Das Fib-Multimediaformat

Version V1.2.2

Betti Österholz

BioKom@gmx.de

www.BioKom.info

Potsdam, 13. Februar 2013

Inhaltsverzeichnis

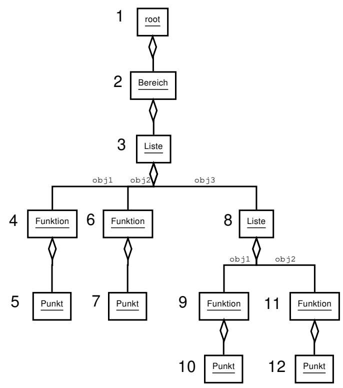
Teil I
Einleitung

1 Zusammenfassung

Das Fib-Multimediasystem zeichnet sich durch seine Universalität und Vielfältigkeit aus. **In diesem Sinne baut das Fib-Multimediasystem auf Vielfalt und nicht auf Spezialisierung auf.**

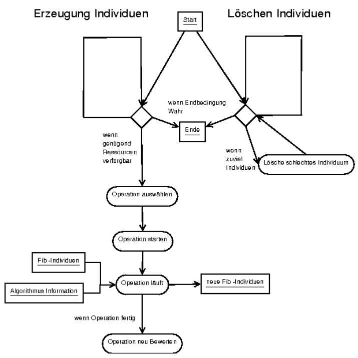
Das Fib-Multimediasystem besteht aus mehreren Komponenten. Die wichtigsten sind im Nachfolgenden kurz vorgestellt.

1.1 Die Fib-Sprache



Das Fib-Multimediaformat dient zum Speichern von Multimediainformationen in strukturierter, funktionaler und hierarchischer Form. Die Struktur des Fib-Multimediaformats unterstützt die Objektsicht auf Dinge. Das Fib-Multimediaformat ist sehr mächtig, da Ausdrücke kombiniert und verschachtelt werden können (Baukastensystem). **Mit Fib ist die Frage nicht mehr, ob Sie etwas machen können, sondern nur noch wie Sie etwas machen können.** Der Speicheraufwand eines Multimediobjekts in Fib ist vielmehr von dessen Komplexität abhängig als von dessen Größe (im Sinne von Ausdehnung in den Dimensionen, also beispielsweise die Höhe und Breite bei Bildern), wie bei üblichen Speicherformaten.
Die einzige Einschränkung bezüglich der Multimediadaten, die im Fib-Multimediaformat darstellbar sind, ist, dass sie als Eigenschaften von Punkten eines endlichen, euklidischen und diskreten (es gibt kleinste Einheiten) Raumes darstellbar sind. Daher können nicht nur Bilder und Töne mit Fib abgespeichert werden, sondern beispielsweise auch Gerüche oder wie weich etwas ist.
Als Grundgerüst von Fib-Multimediadaten dient ein Baum. Die Blätter sind Endpunkte, die für die Darstellung von bzw. Zuordnung zu Punkten oder Teilmultimediaobjekten dienen. In den Ästen und der Ausrichtung dieser, welche z. B. am weitesten links stehen, werden Darstellungsparameter oder Eigenschaften der Blätter kodiert, z. B. wie oft es dargestellt wird und mit welcher Farbe.

1.2 Der genetische Algorithmus



Die zweite wichtige Komponente des Fib-Systems ist der genetische Algorithmus zum Kodieren in und Komprimieren von Fib-Multimediaobjekten. Der große Vorteil dessen ist, dass die Kodierung und Komprimierung nicht mehr an einem bestimmten Algorithmus gebunden ist, sondern dass die eigentlichen Kodierungs- und Komprimierungsalgorithmen im genetischen Algorithmus als Operationen eingebunden werden, welche leicht hinzugefügt und geändert werden können. Dadurch ist es leicht, neue Kodierungs- und Komprimierungsideen einzubringen und eine Vielzahl von diesen auf ein Multimediaobjekt anzuwenden.
In diesem Sinne ist der genetische Algorithmus für Fib ein transgenialer Algorithmus, der darauf angelegt ist, die Ideen von Menschen zur Kodierung von Multimediadaten aus deren Köpfe herauszuholen und in einem Topf zu transportieren bzw. zu sammeln. Damit sollen diese Ideen / Algorithmen mehr leisten können, als sie es einzeln könnten.
Der Algorithmus kann damit mehr Intelligenz in sich vereinen, als es ein Mensch (oder auch eine kleine Gruppe) hervorbringen kann.

1.3 Konvertierungsprogramme

Die Konvertierungsprogramme für Fib dienen zur Übersetzung von anderen Multimediaformaten in das Fib-Multimediaformat und andersherum. Bei dieser Übersetzung sind auch einige Optimierungen der Zielmultimediadaten möglich.

2 Der Ausgangspunkt

Ein (natürliches) Bild besteht im Allgemeinen nicht aus einer Aneinanderreihung zusammenhangloser Pixel (Punkte mit bestimmten Farben), sondern zeigt z. B. Objekte, die gegeneinander abgegrenzt sind und eine Textur haben. Solche Objekte (z. B. Kreise, Linien aber auch komplexere Objekte) wiederholen sich oftmals auf Bildern, sei es als selbstähnliche Kopie oder transformierte Kopie (z. B. Blätter von Bäumen oder Latten im Lattenzaun). Das Gleiche gilt für andere Inhalte von Multimediadaten, wie den Ton.

Des Weiteren können Elemente, die durch Multimediadaten gezeigt werden sollen, oft miteinander assoziiert oder gruppiert werden; z. B. gehören Geräusche oft zu bestimmten dargestellten Objekten (z. B. Entenschnabel), Objekte gehören zu andern Objekten (z. B. Entenschnabel zur Ente) und mehrere Objekte können zu einem Objekt (z. B. eine Wiese mit einer Entengruppe) gruppiert werden.

3 Problem

Wie können Informationen über Objekte aus einem Bild extrahiert werden, das in Form einer Matrix von Pixeln gegeben ist? (Das Bild ist als Pixelbild gegeben, weil es beispielsweise durch Digitalfotografie über einem Fotochip aufgenommen wurde.)

Diese extrahierte Information kann dann z. B. zur Komprimierung benutzt werden. Die Informationen, die benötigt werden, um ein Objekt zu kodieren, ist maximal die Information aller Punkte des Objekts (gegeben als Position und Farbe der Punkte). Jede extrahierte Information aus allen Punkten, muß also Abhängigkeiten zwischen einigen dieser Punkten beschreiben. Kann diese extrahierte Information kürzer dargestellt werden, als die original Information, die sie beschreibt, kann diese extrahierte Information die original Information ersetzen und komprimieren. Wird beispielsweise ein grünes Dreieck im Bild gefunden, das aus 200 grünen Punkten besteht, reicht es im Idealfall, einfach das Dreieck bzw. dessen Punkte durch ein Dreieckobjekt darzustellen, für welches nur seine drei Ecken und seine Farbe benötigt werden.

Man kann mit Information über Objekte im Bild aber auch Objekte auf anderen (Teil-) Bildern wiedererkennen. Oder die Informationen können genutzt werden, um das Bild zu manipulieren, z. B. um Objekte als Ganzes zu entfernen, kopieren oder einzufügen.

4 Lösungsidee

Es wird die Fib-Multimediasprache bzw. das Fib-Multimediaformat definiert, in der Objekte, Transformationen und Zusammenhänge zwischen diesen definiert werden können. Mit Hilfe dieser Multimediasprache entwirft ein Algorithmus (z. B. aus der KI) dann Programme (in der Multimediasprache), testet und verändert diese, so dass sie ein Multimediaobjekt darstellen, das dem zu kodierenden Multimediaobjekt ähnlich genug ist (was dies ist, sollte spezifiziert werden). An dieses Programm können auch noch weitere Einschränkungen erfolgen werden, z. B. Kürze und Abarbeitungsgeschwindigkeit des Programms. Mit diesen Kriterien kann dann festgestellen werden, wie "gut" das Programm ist.

Unter Anderem bieten sich dafür genetische Algorithmen/Programmierung an, da bei genetischen Algorithmen die einzelnen Parameter, Elemente und Konstrukte der Sprache durch Mutation erzeugt und angepasst werden können und die "besten" Programme weiterentwickelt werden. Mit "crossing over", Austausch von Programmcodes zwischen Programmen oder Programmstellen, kann der Umstand berücksichtigt werden, dass die Objekte an anderen Stellen eventuell nur verändert (z. B. vergrößert) wiederverwendet werden.

Warum genau genetische Algorithmen sich anbieten, soll noch im Einzelnen erläutert werden. Es soll auch eine Verbindung zur "biologischen" Evolution/Genetik hergestellt werden. Ein weiterer Bestandteil der Idee ist es, dass nicht nur einem

“Multimediaprogramm/-objekt” als Ganzes eine Fitness zugeordnet wird, sondern auch seinen Teilobjekten, indem bestimmt wird, welcher Multimediaobjektteil von diesem überdeckt wird und wie gut dieser Multimediaobjektteil überdeckt wird.

Die Objekte/Programme mit der höchsten Fitness erhalten höhere Chancen beim “Überleben” (werden seltener gelöscht) und “Vermehren” (werden häufiger für neue Objekte verwendet).

Es ist weiterhin möglich, bei der Kodierung neuer Multimediaobjekte (Teil-) Objekte (z. B. Dreiecke) zu verwenden, die schon in anderen Multimediaobjekten eine hohe Fitness erreichten. (Die Objekte sind dann Vorwissen aus einer Datenbank.)

5 Entstehungsgeschichte dieser Arbeit

Die Idee, etwas über evolutionäre Algorithmen (EA) zu erarbeiten, hat seinen Ursprung darin, dass ich mich für Verfahren interessiere, die zur automatischen Problemlösung dienen, und darin, dass auf dem Gebiet der evolutionären Algorithmen, gerade im Bereich der komplexeren Anwendungen, anscheinend noch wenig Forschung betrieben wurde. Ich suchte also ein Thema, das sich von mir mit evolutionären Algorithmen bearbeiten ließ. Das hieß vor Allem, dass genug “Lernmaterial” für den Algorithmus zugänglich ist, und eventuell noch praktischen Nutzen bringen konnte. So verfiel ich auf die Bildbearbeitung mit evolutionären Algorithmen. Die Bildbearbeitung ist ein großer Sektor in der EDV und der Grundstoff Bilder ist leicht verfügbar.

Die Grundidee ist: Aus speicheraufwändigen Rastergrafikbildern mit evolutionären Algorithmen speichersparende Vektorbilder zu generieren. Daraus entstand die erste Version des Fib-Systems.

Dieses Verfahren wird hier in der zweiten Version auf allgemeine Multimediainhalte erweitert.

In meinen Recherchen habe ich über solche Verfahren nichts gefunden. Es ist also gut möglich, dass ich “Neuland betrete”.

Die nachfolgende Dokumentation dient dazu, die Ideen hinter dem Projekt darzustellen und Anderen die Möglichkeit zu geben, mein Projekt für weitere Forschungen und Entwicklungen auf diesem Gebiet nutzen zu können.

6 Einteilung dieses Projekts

Der Aufbau dieses Dokuments richtet sich nach der Projektstruktur.

Das Projekt is hierarchisch aufgebaut und gliedert sich in zwei große Komponenten. Die Komponenten sind die Multimediabeschreibungssprache Fib und der genetische Algorithmus, um effizente Fib-Kodierungen von Multimediaobjekten zu erzeugen.

Hierarchisch ist das Projekt, also auch jeweils die Komponenten, in mehrere

Ebenen aufgebaut, wobei eine höhere Ebene auf den unteren aufbaut. Dabei werden die Ebenen immer detailierter. Diese Ebenen sind:

1. Eine Zielbeschreibung für die Komponenten (siehe Abschnitt 3 auf Seite 3, sowie der folgende Abschnitt 4 und für Fib Abschnitt 10 auf Seite 22)
2. Die theoretische Beschreibung der Komponente: Diese beschreibt die Komponenten, ohne auf deren möglich Realisierung einzugehen (siehe für die Fib-Sprache Teil III auf Seite 22 und für den denetischen Algorithmus Teil IV auf Seite 95).
3. Ein Entwurf der Realisierung der Komponente: Dies ist ein Entwurf für die Implementierung der Komponenten. Er kann von der tatsächlichen Implementierung abweichen und soll nur den allgemeinen Aufbau sowie die Schnittstellen beschreiben. Implementierungen sollten sich nach diesem richten, so dass sie auf einem allgemeinen Standard fußen. Der Entwurf sollte aber Implementierungen genug weitere Freiräume lassen (siehe allgemein Teil VI auf Seite 172; für die Fib-Sprache Teil VII auf Seite 175 und Teil V auf Seite 105; sowie für den genetischen Algorithmus Teil VIII auf Seite 366, Teil IX auf Seite 432 und Teil XIV auf Seite 473).
4. Die Dokumentation der Realisierung: Sie umfasst die Beschreibung der Schnittstellen der konkreten Implementierung. Eine solche Dokumentation ist nicht Bestandteil dieses Dokuments und ist zur jeweiligen Realisierung zu halten. In der konkreten Realisierung in C++ zu diesem Dokument gibt es eine Dokumentation, die mit Doxygen erstellt wird und im Doxygen Java Stil erfolgt.
5. Die konkrete Realisierungen: Auch hier ist eine Dokumentation der Funktionsweise des Quellcodes sinnvoll.

7 Einteilung dieses Dokuments

In diesem Abschnitt wird eine Übersicht über den Inhalt und Aufbau dieses Dokuments gegeben.

7.1 Einführung in die Grundlagen

Im Teil II auf Seite 8 gibt es eine Einführung in die Grundlagen evolutionärer und genetischer Algorithmen und der Bilddarstellung (als wichtiges Beispiel für Multimediaobjekte).

7.2 Entwurf der Multimediabeschreibungssprache

Der Teil III auf Seite 22 befasst sich mit der Multimediabeschreibungssprache.

In Abschnitt 12 auf Seite 23 werden die einzelnen Elemente und deren mögliche Beziehungen zueinander spezifiziert.

Der Abschnitt 13 auf Seite 72 geht auf die Datenbankfunktion für Fib (-Objekte) ein.

Im nachfolgenden Abschnitt 14 auf Seite 72 werden Definitionen für die Fib-Multimediasprache aufgestellt, die einen Umgang mit ihr erleichtern sollen.

In den Abschnitten 15 auf Seite 84 und 16 auf Seite 89 wird die Multimediabeschreibungssprache näher beleuchtet.

Dabei wird in diesem Teil III :

- Begründet, warum mit dieser zumindest alle möglichen Bilder realisiert werden können.
- Der Entwurf in Hinblick auf die Aufgabenstellung beleuchtet.

Die berücksichtigenden Fragestellungen dabei sind:

- Was ist zu beachten?
- Wie sind die Anforderungen an die Multimediabeschreibungssprache?
- Welche Erweiterungen und Parameter sind möglich?

7.3 Entwurf des genetischen Algorithmus

Im Teil IV auf Seite 95 wird der Entwurf des genetischen Algorithmus beschrieben. Dabei geht es vor allem um die grundlegenden Ideen hinter dem genetischen Algorithmus.

7.4 Dateiformat

Im Teil V auf Seite 105 werden die Dateiformate für die Fib-Multimediasprache dargestellt. Diese sind unabhängig vom Programmsystem oder der Rechnerarchitektur.

7.5 Implementierung der Idee

In dem Teil VII auf Seite 175 wird der Implementierungsentwurf der Fib-Multimediasprache ausgearbeitet. Für den genetischen Algorithmus wird im Teil VIII der Implementierungsentwurf erarbeitet.

7.6 Vorgehensweisen für Fib

Im Teil XII auf Seite 454 werden einige Vorgehensweisen beschrieben, die in verschiedenen Situationen im Umgang mit Fib nützlich sein könnten.

7.7 Operatoren

Der Teil XIV auf Seite 473 dient der Beschreibung einiger Operatoren, die im genetischen Algorithmus zum Einsatz kommen können.

Im vorhergehenden Teil XIII auf Seite 459 werden einige allgemeine Algorithmen vorgestellt und beleuchtet, die in Operatoren zum Einsatz kommen können.

Teil II
Einführung in die Grundlagen

8 Grundlagen genetischer Algorithmen

Evolution ist eines der grundlegensten Naturgesetze unseres Universums. Seit rund 3 Milliarden Jahren gibt es auf der Erde eine genetische Evolution, die, wie wohl nur wenige bestreiten werden, ganz erstaunliche Ergebnisse und Problemlösungen hervorgebracht hat. Deshalb ist es nicht verwunderlich, dass versucht wird, die Methoden und/oder die Wirkungsweise der genetischen Evolution auch in der Informationstechnik einzusetzen. Vielfach werden schon große Fortschritte in der Wissenschaft und Forschung erzielt, indem man sich Prinzipien der Ergebnisse (z. B. Flügelauftrieb) der natürlichen Evolution von der Natur "abgeschaut" hat. Aber auch viele Interpretationen der Evolution wurden schon, teilweise mit Erfolg, versucht anzuwenden.

Die nachfolgende Arbeit ist einer dieser Versuche.

Wie funktioniert Evolution aber oder was ist sie überhaupt genau?

Ob ich diese Fragen zufriedenstellend beantworten kann, weiß ich nicht. Ich will es aber versuchen und vielleicht noch ein paar Denkanstöße geben. Die nachfolgende Sicht von Evolution und genetischer Evolution ist grundlegend für die gesamte vorliegende Arbeit. Sie nimmt aber keinesfalls in Anspruch, vollständig oder "richtig" zu sein. Um genetische Evolution verstehen zu können, muss wahrscheinlich zuerst erklärt werden, was Evolution ist.

8.1 Geschichte

Der berühmteste Forscher, der auf die Evolution aufmerksam wurde und darüber eine Buch verfasste, war Charles Darwin (1809- 1882), der 1859 das Buch "Die Entstehung der Arten"[7] veröffentlichte. Dabei ging es um die Evolution in der Natur und wie Arten aus anderen Arten entstanden sind. Eine Idee, die ihm besonders im Kreise der damals noch wesentlich mächtigeren Kirche viele Unsympathien einbrachte, da diese Idee dem Schöpfungsgedanken zuwiderlief.

Die Idee, Evolution auch zum maschinellen Lernen zu verwenden, war sehr früh in der Informatik vorhanden. Schon 1932 beschrieb W. D. Cannon natürliche Evolution als einen Lernprozess. A. M. Turing (1950) vermutete "an obvious connection between [machine learning] and evolution" (Übersetzung: "eine offensichtliche Verbindung zwischen [maschinellem Lernen] und Evolution"). Leider war zu dieser Zeit die Rechnerleistung bei weitem noch nicht hoch genug, um eine Simulation der Evolution in vernünftigem Rahmen zu gewährleisten.

Erst in den 1980ern war die zur Verfügung stehende Rechenleistung hoch genug, um Evolution zum maschinellen Lernen sinnvoll einsetzen zu können. Danach wurde das Interesse an Evolution in der Informatik größer. Evolution wurde schon in vielen Bereichen mehr oder weniger erfolgreich eingesetzt. Angefangen vom Finden globaler/lokaler Optima von Funktionen, bis hin zum automatischem Schaltungsentwurf oder Roboterkonstruktion.

8.2 Evolution

Evolution ist ein allgemeines Prinzip/Naturgesetz, das man kurz mit "Das Bessere setzt sich durch" erklären kann. Bei genauerer Betrachtung wird der Spruch etwas undeutlich, denn was ist "besser" (für wen) und was ist "sich" oder "durchsetzen".

"Sich" kann auf alles bezogen werden. Ob es nun ein Lebewesen, ein Programm, ein Prinzip oder Sonstiges ist. Für "sich" sind aber die Informationen die es enthält (z. B. Aufbau der Moleküle oder Art des Gemeinschaftslebens) und nicht seine Materie oder Energie (wenn vorhanden) für die Evolution ausschlaggebend. Ein Hund ist nicht so erfolgreich, weil er 25 Kilogramm Materie enthält, sondern weil diese Materie (und seine Energie, wenn diese von Materie getrennt wird) auf ganz spezielle Weise "verteilt/aufgebaut" ist. Auch darf "sich" nicht als ein abgeschlossenes Einzelnes gesehen werden, sondern muss die Umwelt mit einbeziehen. Ameisen sind z. B. so erfolgreich, weil sie in Staaten leben. "Sich" kann auch weitere "sichs" enthalten oder in einem "sich" enthalten sein. So haben viele Säugetiere Darmbakterien, ohne die sie nicht mehr die gleichen Tiere wären. Viele Arten bestehen nur aufgrund dieser Darmbakterien, z. B. Kühe.

Die Frage "Für wen besser?" muss mit "Für sich besser" beantwortet werden. "Für sich besser" ist, wenn "sich" häufiger oder wahrscheinlicher wird.

Damit ist auch gleich die Frage beantwortet, was "durchsetzen" ist: Etwas setzt "sich" durch, wenn es wahrscheinlicher/häufiger in einer Gruppe/Menge von Dingen bzw. anderen "sichs" wird. Die in Australien eingeführten Kaninchen haben sich in den letzten Jahrhunderten gegenüber anderen Tierarten durchgesetzt oder innerhalb dieser Tierartengruppe (Grasfresser) sind sie auf Kosten der Anderen wahrscheinlicher/häufiger geworden. Damit etwas wahrscheinlicher/häufiger werden kann, ist natürlich Zeit (oder Ähnliches) von Nöten. Das heißt, Evolution ist ein zeitlicher Prozess.

Was bei der Erklärung "Das Bessere setzt sich durch" nicht direkt über die Evolution gesagt wird, ist dass das "sich" in mehreren ähnlichen Kopien vorhanden sein muss, damit es "sich" gegen andere "sichs" durchsetzen kann (damit sich ein Grasfresser gegenüber anderen durchsetzen kann, muss es davon mehrere geben). Daraus folgt unter Anderem: Um an einer Evolution teilnehmen zu können, müssen von "sichs" ähnliche Kopien erstellt werden (z. B. durch Vermehrung) und auch mehrere dieser Kopien gleichzeitig vorhanden sein (für Selektion). Nur so kann sich davon "das Bessere" "durchsetzen" und es kann immer wieder "Besseres" entstehen. Zusammenfassend kann gesagt werden, dass Evolution bedeutet: "Dinge", die so sind, dass sie sich selbst in einer Umwelt wahrscheinlicher machen, werden wahrscheinlicher. Insofern ist Evolution trivial.

Das "sich" wird als Individuum bezeichnet.

Wie gut ein Individuum ist, wird als Fitness bezeichnet.

Selektion ist die Auswahl von Individuen, so dass die besseren Individuen wahrscheinlicher/mehr werden.

Technisch gesehen besteht die Evolution aus drei Komponenten oder “Säulen” (siehe “Die Lösung von Darwins Dilemma” [13]):

1. Variation: die Veränderung von Eigenschaften bei Individuen
2. Selektion: Individuen mit (in ihrer jeweiligen Umwelt) besseren Eigenschaften werden bevorzugt bzw. haben eine höhere Überlebenswahrscheinlichkeit.
3. Vererbung: Individuen geben ihre Eigenschaften an ihre Nachkommen ab.

Alle drei Komponenten sind nötig, denn: Ohne Variation entsteht nichts Neues, ohne Selektion kann etwas Neues nicht wahrscheinlicher werden und ohne Vererbung kann sich das Neue nicht ausbreiten.

Als eine Bedingung an die Vererbung müsste noch gestellt werden, dass durch sie die Anzahl der Individuen erhöht werden kann. Dadurch entstehen dann auch automatisch größere Populationen mit einer neuen guten Eigenschaft.

8.3 Genetische Evolution

Genetische Evolution bezieht sich auf eine Evolution, in der die Gene von Lebewesen (Individuen) die Informationsträger darstellen (Genotyp). Alle Gene eines Individuums zusammen stellen dann den Genotyp eines Individuums dar. Diese werden als ein Lebewesen “dekodiert” bzw. führen dann zur Ausprägung des Phänotyps des Individuums in der Umwelt (die Umwelt kann Auswirkungen auf den Phänotypen und dessen Ausbildung haben). Der Phänotyp wiederum bestimmt zusammen mit der Umwelt (ein Fisch an Land hat keine hohe Fitness im Verhältnis zu Landlebewesen) die Fitness (eigentlich nur ein theoretisches Maß) des Individuums. Der Phänotyp zusammen mit der Umwelt hat Auswirkungen darauf, wie wahrscheinlich das Individuum Nachkommen zeugen wird oder wie viele Individuen ein zumindest ähnliches Genom in Zukunft erhalten. Um so mehr Individuen ähnliches Genom erhalten, um so wahrscheinlich wird dieses oder ähnliche Genome in Zukunft sein. So wird der Kreis geschlossen.

Am Beispiel der Evolution in der Natur:

Man nehme z. B. den Schneehasen. Dieser ist besonders gut geeignet für die Evolution, weil er sich so stark vermehrt. Der Hase, das kleine, kuschelige, weiße, hoppelnde Ding, wird Phänotyp genannt, also das Phänomen in der Natur. Die Informationen, die in seinen Genen gespeichert sind, werden Genotyp genannt. (Die Genen sind [spiralförmige] Molekühlketten im Zellkern und aber auch Gene anderswo [z. B. in den Mitochondrien].) Genotyp und Phänotyp zusammen werden als Individuum bezeichnet, also der Hase als Ganzes. Mehrere Individuen, die untereinander in Beziehung stehen und sich untereinander fortpflanzen können, werden als Population bezeichnet. Die Gesamtheit der Genotypen einer Population wird als Genpool der Population charakterisiert. Weil die Evolution (hauptsächlich) auf die Gene bzw. den Genpool wirkt, wird sie als genetische Evolution bezeichnet.

Der Genotyp bestimmt stark den Phänotyp, das Aussehen des Hasen, legt den Phänotyp aber nicht allein fest, denn die Umwelt kann als weiterer Faktor den Phänotyp bestimmen. Wenn ein Hase nur wenig zu fressen bekommt, wird er nicht dick werden, ganz egal, was in seinen Genen steht: Wo nichts ist, kann auch nichts werden. Der Genpool, also die Gesamtheit der Genotypen, ist das, was die Evolution in einer Population verändert oder anpasst. Wird der Phänotyp durch die Umwelt geändert, wird das meist nicht weitergegeben. Wenn ein Hase dünn ist, weil er wenig zu fressen bekommt, können seine Kinder trotzdem dick werden.

Es ist allerdings möglich, dass Mütter erworbene Immunitäten an ihre Kinder weitergeben, z. B. mit der Muttermilch, also etwas, was eigentlich zum Phänotyp gehört und nicht zum Genotyp, da diese Immunität nicht in den Genen kodiert ist. Hier sieht man noch einen wichtigen Punkt der Natur: Die Ausnahme ist meist Regel.

Die Veränderung des Genpools geschieht allerdings über den Umweg der Phänotypen der Population. Unter vielen Komponenten ist die Fortpflanzung dabei die wichtigste. Um so besser der Phänotyp dazu geeignet ist, dass das Individuum sich fortpflanzt, um so eher wird es viele Nachkommen mit seinen Genen haben und um so wahrscheinlicher werden seine Gene in der zukünftigen Population anzutreffen sein.

Ein flinker Hase wird wahrscheinlicher ins geschlechtsreife Alter kommen, da er Jägern besser entkommen kann, als ein nicht so flinker Hase. Dadurch haben die flinken Hasen eine bessere Chance, sich fortzupflanzen und viele Nachkommen zu haben, als langsamere. Flinke Hasen haben auch eher Gene, die sie flinker machen als Langsamere. Dadurch werden Gene, die flink machen, eher vererbt, weshalb es ziemlich schwierig ist, Hasen mit der Hand zu fangen. (Dies weiß ich aus eigener Erfahrung.) Hier wird ein weiterer Punkt der Evolution sichtbar: Es geht immer um Wahrscheinlichkeiten und nicht um “mit Sicherheit”. Hasen die flinker als andere sind, können auch einfach das Glück gehabt haben. Wenn sie beispielsweise, als sie klein waren, an einen Ort gelebt haben, wo es immer gutes Futter gab. Das gute Futter hilft ihnen sicherlich dabei, groß und flink zu werden. Das gute Futter ist dann also ein guter Ersatz für “flinkmachendere Gene”. Ein Hase, der noch so gute flinkmachende Gene hat, kann trotzdem noch als kleiner Hase einem Unglück oder Räuber zum Opfer fallen. Es ist nur wahrscheinlicher, dass ein Hase mit “flinkmachenden Genen” flink wird, länger lebt und mehr Nachkommen zeugt. Das ist auch ein Grund, warum Evolution nur auf viele Individuen wirkt und nicht auf ein einzelnes alleine.

Hasen, die eine bessere Chance als andere haben, sich fortzupflanzen und viele Nachkommen zu zeugen, haben eine höhere Fitness als andere. Das heißt, um so fitter Hasen sind, um so wahrscheinlicher werden ihre Gene in späteren Generationen anzutreffen sein.

Ein weiterer Punkt ist, dass nichts so einfach oder abgeschlossen ist, wie es zunächst scheint. Trotz der Tatsache, dass ein Hase ein Individuum ist, kann er wiederum andere Individuen, sogar ganze Populationen, enthalten. Hasen haben Darmbakterien, die als eigenständige Individuen in einer Umwelt (Hasendarm) angesehen werden. Der Hase ohne seine Darmbakterien ist allerdings nicht mehr das gleiche Individuum aus evolutionstechnischer Sicht, da ohne seine Darmbakterien seine Fitness anders wäre. In diesem Fall wäre sie wahrscheinlich schlechter, weil er seine Nahrung schlechter verdauen könnte. Diese Darmbakterien haben ihre eigenen Gene, bzw. ihren eigenen Genotyp, und ihren eigenen Phänotyp. Als Population haben sie ihre eigene genetische Evolution. Auch können Gruppen von Hasen zu Individuen zusammengefasst werden, die untereinander in evolutionärer Beziehung stehen. Bei solchen "Gruppenindividuen" zählt dann nicht mehr das Überleben des einzelnen Hasen aus der Gruppe, sondern nur noch das der Gruppe, bzw. dessen Genpool. So können Gene, die dazu führen, dass sich einzelne Hasen nicht fortpflanzen (z. B. Gene für einen Beschützerinstinkt, der stärker ist als der Überlebensinstinkt) durchaus sinnvoll sein und erhalten bleiben, wenn sie die Fitness der Gruppe erhöhen.

Evolution wirkt auch bei vielen Vorgängen, in denen sie gar nicht vermutet wird, z. B. hat man bei Ideen oder Programmen meist mehrere von diesen zu einem Thema und mit der Zeit setzen sich dann die "Besseren" (was besser ist, bestimmt die Umwelt) von diesen durch und andere, nicht so gute verschwinden.

Die gewollte und bewusste Anwendung in der Informatik von Wissen über die Evolution heißt evolutionäre Algorithmen, die der genetischen Evolution entsprechend genetische Algorithmen. Deren Grundlagen seien im Nachfolgendem beschrieben.

8.4 Grundlagen evolutionärer Algorithmen (EA)

Evolutionäre Algorithmen arbeiten auf einer Menge (Population) von potentiellen Lösungen (Individuen), die das gegebene Problem mehr oder weniger gut lösen bzw. der optimalen Lösung mehr oder weniger nahe kommen. Von diesen Lösungen werden dann Neue geschaffen, indem einige bis alle zufällig verändert werden. Von allen (potentiellen) Lösungen, Neue und Alte, werden dann in einem Selektionsschritt einige gelöscht. Dabei haben weniger gute Lösungen eine höhere Wahrscheinlichkeit gelöscht zu werden. Dies führt im Allgemeinen dazu, dass die guten Lösungen eine höhere "Überlebenschance" (Fitness), also eine höhere Wahrscheinlichkeit, nicht gelöscht zu werden, haben und damit in der Menge bessere Lösungen zunehmen. Diesen Prozess der Veränderung und Selektion wird dann solange wiederholt, bis eine Endbedingung (Terminalkondition) zutrifft. Die Endbedingung ist meist: Eine bestimmte Anzahl von Durchläufen wurde ausgeführt oder/und eine Lösung überschreitet eine festgelegte Grenze (z. B. wenn die Lösung gut genug ist).

Listing 1: Allgemeiner Algorithmus

```
Initialisieren der Menge der potentiellen Loesungen
loop
  - hinzufuegen neuer Individuen zur Menge, die aus alten
     Individuen, aus der Menge, mit Zufall generiert werden
  - Selektion = zufaelliges Loeschen von Individuen aus der
     Menge, bessere haben eine hoehere Wahrscheinlichkeit,
     nicht geloescht zu werden
until (Terminalkondition erfuellt)
```

Im Listing 1 ist schematisch der allgemeine Algorithmus eines evolutionären Algorithmus in pseudo-C dargestellt.

Bei der Selektion von Individuen findet oft die weel-selection Anwendung. Dabei haben Lösungen/Individuen mit einer niedrigeren Fitness eine höhere Wahrscheinlichkeit, gelöscht zu werden. Der Name weel-selection hat seinen Ursprung darin, dass Wahrscheinlichkeiten für Individuen grafisch in einem Kreis angeordnet werden können (Individuen mit geringerer Fitness bekommen einen größeren Kreisausschnitt). Auf diesem Kreis (bzw. seinem Rand) wird bei der Selektion zufällig ein Punkt ausgewählt und das zugehörige Individuum gelöscht.

8.5 Unterarten

Es werden zwei Arten von evolutionären Algorithmen unterschieden. Die Evolutionsstrategien und die genetische Algorithmen (GA), wobei diese noch in weitere Unterarten unterschieden werden kann (z. B. sequentielle und parallele oder klassische GA und genetische Programmierung). Eine mögliche Einteilung ist in der Abbildung 1 (nach [3]), dabei sind die evolutionären Algorithmen neben den Simulated Annealing als eine Unterart der informellen Zufallssuche eingeordnet.

Wo denn nun genau diese Unterschiede der Arten liegen, ist oft nicht mehr so klar zu erfassen: [9] “Within the past five years, each area of evolutionary computation has borrowed and modified ideas from the others. Over time, an iterative blending has occurred such that all classes of evolutionary algorithms now appear quite similar, if not for all intents and purposes identical” Eine genaue Einteilung von realisierten Algorithmen in die verschiedenen Arten ist also nicht so leicht möglich.

8.5.1 Genetische Algorithmen (GA)

Genetische Algorithmen können einfach beschrieben werden als: evolutionäre Algorithmen + “Crossover”

Vorbild: natürliche Evolution auf Genbasis

Dabei wird als “Crossover” der Vorgang bezeichnet, bei dem von zwei oder mehr Individuen die (Erb-)Informationen kombiniert werden, um ein neues Individuum zu schaffen.

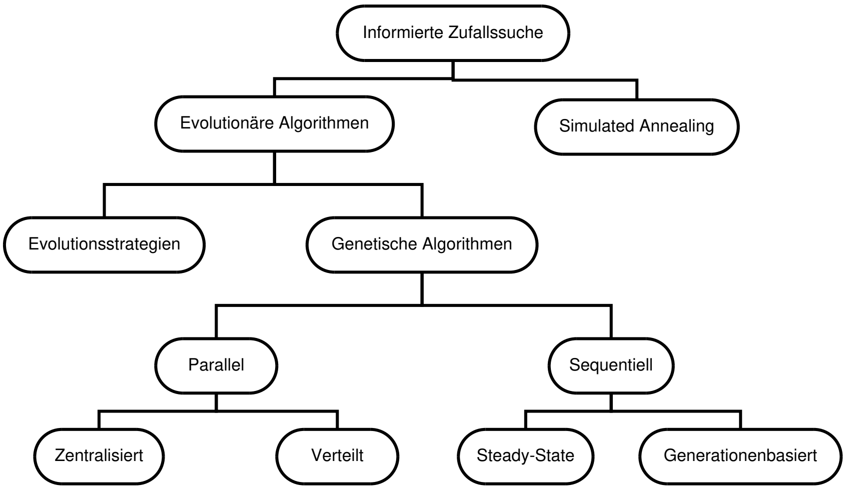


Abbildung 1: Einteilung von Lernverfahren

8.5.2 Klassische genetische Algorithmen (GA)

Die klassischen genetische Algorithmen (GA) sind Algorithmen die auf einer Liste mit Bits arbeiten, welche die potentiellen Problemlösungen/Individuen repräsentieren. Mutation ist dabei das Invertieren von Bits. Beim Crossover gibt es meist one- oder two-point-crossover (bei GA's). Beim one-point-crossover wird die Bitsequenz der beiden Individuen jeweils an einem Punkt, meist die gleiche Position, durchgeschnitten und die beiden Teilhälften eines Individuums mit den entsprechend anderen Teilhälften des anderen Individuums verbunden. Two-point-Crossover ist identisch, nur das hier an zwei Punkten geschnitten wird.

8.6 Anmerkungen zu evolutionären Algorithmen (EA)

Eine geeignete Wahl der Repräsentation von Lösungen (also wie die Individuen aufgebaut sind) und angepasste Operatoren (welche die Individuen verändern bzw. neue Individuen erzeugen) können die Performance von evolutionäre Algorithmen deutlich erhöhen ([9]). Durch eine geeignete Wahl der Repräsentation wird der Hypothesenraum, den diese aufspannen, günstig beeinflusst, z. B. können ungültige oder unerwünschte Lösungen ganz entfallen. Weiterhin ermöglicht eine gute Repräsentation auch einen Hypothesenraum, in dem Operatoren einfacher zu realisieren sind, die im Allgemeinen schneller zur Lösung führen, z. B. ist es dann wahrscheinlicher, dass ähnliche Lösungen auch nahe im Hypothesenraum "beieinanderliegen" (für die Operatoren).

Durch angepasste Operatoren werden die "Wege" (welche diese Operatoren im Hypothesenraum erzeugen) besser, bzw. führen schneller zu besseren Lösungen. Es können beispielsweise als Operatoren schon bekannte Algorithmen zur Optimierung eingesetzt werden oder es kann in den Operatoren Wissen über das

Problem verwendet werden.

Die große Freiheit bei der Wahl der Repräsentation der potentiellen Problemlösung ist einer der größten Vorteile der evolutionären Algorithmen. Theoretisch ist eine beliebige Repräsentation wählbar, solange eine Bewertung von potentiellen Problemlösungen in dieser möglich ist. Andere Lernverfahren ermöglichen meist nur bestimmte Repräsentationen, z. B. können Lernverfahren von neuronalen Netzen nur neuronale Netze als Problemlösung hervorbringen. Bei EAs allerdings können unter anderen Bitlisten, Moleküle, beliebige Programmiersprachen oder mathematische Formeln verwendet werden. Genauso variabel sind damit auch die Operatoren, mit denen die potentiellen Lösungen angepasst werden können. Damit können viel besser Vorwissen oder auch Vermutungen in den Algorithmus eingearbeitet werden.

Weiterhin kann bei der Wahl der Repräsentation auch noch die Anschaulichkeit für den Menschen berücksichtigt werden. Das heißt, die Repräsentation kann so gewählt werden, dass potentielle Problemlösungen einfacher verstanden werden können.

Der große Nachteil von evolutionären Algorithmen ist der hohe Zeitaufwand bzw. Rechenaufwand den sie im Verhältnis zu anderen Algorithmen benötigen. Wo andere Algorithmen die vorhanden Daten nutzen, um aus ihnen einmal eine Lösung zu generieren, generiert ein genetischer Algorithmus ständig neue Lösungen, testet, wie gut diese sind, und wählt möglichst die Guten aus. Solange die Terminalkondition nicht erfüllt ist, hält ein genetischer Algorithmus auch nicht an. Die Terminalkondition kann aber auch sehr schwer oder gar nicht erfüllbar sein. Ein gutes Beispiel ist auch hier die natürliche Evolution: Sie läuft schon seit mindestens 4 Milliarden Jahren auf wenigstens einen ganzen Planeten, und es ist noch kein Ende abzusehen.

9 Multimediaformate

Multimedia ist der Begriff für Informationen, wenn mehrere Medien gebündelt werden. Medien sind Träger von Dingen, die über Sinne wahrnehmbar sind, wobei den verschiedenen Sinnen jeweils ein Medium zugeordnet ist. Multimediaformate sind danach Sätze von Regeln, wie diese Dinge abgespeichert (dauerhaft gemacht) oder übertragen werden können.

Dabei sind unter Anderem Medien für folgende Dinge zu unterscheiden:

- Bild
- Ton
- olefaktorische Medien (Gerüche)
- hauptische Medien (wie sich ein Gegenstand anfühlt, z. B. hart oder rau)

Im Nachfolgenden werden vor Allem das Bild und der Ton beschrieben, da sie die größten Bereiche sind (bzw. in unser heutigen Informationsgesellschaft am häufigsten anzutreffen sind).

9.1 Bilder

9.1.1 Einleitung

Wie auch in vielen anderen Bereichen wurde mit dem Aufkommen leistungsfähiger Computer die Bildverarbeitung revolutioniert. Handelte es sich Anfang der 60er Jahre [19], als die grafische Datenverarbeitung aufkam, noch hauptsächlich um Strichzeichnungen und Diagramme, die erstellt, manipuliert und verarbeitet wurden, so sind es heutzutage ganze Filme, die mit Computern erzeugt werden. Es ist schon heute so gut wie unmöglich, ein durch gute Manipulationen erstelltes Bild von einem "echten" (nicht manipuliertem) Bild zu unterscheiden. Heute stehen eine Unzahl von Programmen zur Erstellung, Manipulation und Verarbeitung von Bildern zur Verfügung und damit einhergehend eine Unzahl von Bildformaten. Alle diese Bildformate haben ihre Vor- und Nachteile und werden in den verschiedensten Bereichen genutzt.

Im Nachfolgenden sollen einige wichtige Eigenschaften von Bildkodierungen näher erläutert werden.

9.1.2 Bildformat

Ein Bildformat oder Bildkodierung ist ein Format (bzw. eine Strukturbeschreibung) für eine Ansammlung von Daten, die ein Bild repräsentieren. Dabei kommt es vor Allem auf die effiziente Organisation der Daten in Bezug auf das Anwendungsgebiet an. Die meisten Bildformate teilen ihre Daten in zwei Blöcke auf: einen Block mit allgemeinen Informationen zum Bild (Farbpalette, Bildgröße, etc.) und einen Block der angibt, was auf dem Bild dargestellt wird (z. B. die Objekte oder die Farbwerte für die einzelnen Pixel). In Bild 2 ist als Beispiel der Aufbau einer Datei im Bitmapformat BMP (*.bmp) angegeben. Eingeleitet wird dieses Format durch den Bitmap Header, der anzeigt, dass es sich um ein Bitmapbild handelt (er besteht aus den Buchstaben "bmp"). Im Informations-Header befinden sich Informationen zum Bild selbst, unter anderem sind dort zu finden: die Höhe und Breite des Bildes, die horizontale und vertikale Auflösung in Pixel pro Meter, der Typ der Komprimierung und die Anzahl der benutzten Farben.

Die Farbpalette definiert jede Farbe durch ihren Anteil an Rot, Grün und Blau. Die Daten enthalten die zeilenweise Rasterinformation des Bildes. Hierbei ist der Ausgangspunkt die linke untere Ecke des Bildes.

9.1.3 Eigenschaften von Bildern

Wichtige Eigenschaften für Bildformate werden im Folgendem aufgeführt. Dabei müssen nicht alle Bildformate alle Eigenschaften unterstützen.

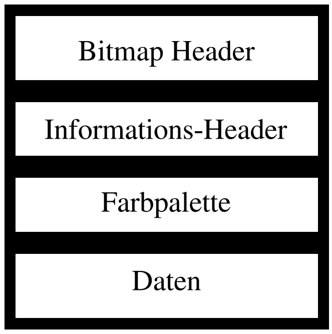


Abbildung 2: Dateiaufbau eines Bitmap-Bildes

Der **Dateityp** (Filetype) ist das, woran im Dateisystem erkannt werden kann (oder zumindest erkannt werden sollte), um welche Art von Bildkodierung es sich handelt. Meist handelt es sich dabei um ein Dateiendungskürzel, das eine Art Abkürzung der vollen Bezeichnung der Bildkodierungsart ist, z. B. die Dateiendung ".bmp" für die Bitmap-Kodierung.

Die **maximale Bildgröße** gibt an, wie die maximale Größe eines Bildes sein darf, um in diesem Bildformat noch abspeicherbar zu sein. Typisch ist hier die maximale Anzahl der Pixel, die ein Bild haben darf. Die maximale Bildgröße hängt vom Speicherformat ab. Da in digitalen Rechnern nicht unendlich große Zahlen abgespeichert werden können (wegen des begrenzten Speicherplatzes), können auch die Zahlen für Koordinaten oder Bildgröße nur endlich sein.

Die **Anzahl der Kanäle** gibt an, aus wie vielen Teilbildern übereinander das Bild besteht. Üblich ist ein Kanal, aber in einigen Anwendungen ist es notwendig, mehrere Bilder übereinander zu legen, um z. B. besser Transparenzen im Bild darstellen zu können.

Mit dem **Farbmodell** werden die Farben angegeben, die verwendet werden können. Der einfachste Fall ist z. B. S/W (nur schwarz und weiß als "Farben").

Die **Farbtiefe** gehört zum Farbmodell und ist die Anzahl möglichen Farben, die einem Pixel oder Objekt zugeordnet werden können. Farbtiefe in Bits ist die Anzahl der Bits pro Farbe, die einem Pixel oder Objekt für dessen Farbe zugeordnet werden. Die beiden Begriffe hängen eng zusammen, da die Bits pro Farbe meist die Farbtiefe bestimmen nach der Formel: $Farbtiefe = 2^{(Bits\ pro\ Farbe)}$

Der **Hersteller** eines Bildformates ist wichtig im Hinblick auf Lizenzen und mögliche Updates. Muss für ein Bildformat bezahlt werden, wenn es in einer Software benutzt werden soll? Wird dieses Bildformat in Zukunft eventuell noch besser ausgebaut usw. ?

Plattformabhängigkeit: Bei vielen unbekannteren Bildformaten gibt es nur für einige Plattformen (z. B. Rechnertypen oder Betriebsysteme) Programme zum Erstellen, Bearbeiten oder Darstellen von Bildern in ihnen.

Als **Komprimierung** wird bezeichnet, wie viel Informationen (Bits) für ein Bild benötigt werden, um es aus diesen wiederherzustellen.

Die **Darstellungsarten** Vektorgrafik oder Rastergrafik beziehen sich auf die Art, wie die Bildinformationen abgespeichert sind. Bei Rastergrafiken wird über das Bild ein Raster gelegt und die einzelnen Punkte abgespeichert. Bei Vektorgrafik werden die "Objekte" (z. B. Linien, Kreise) des Bildes abgespeichert.

9.1.4 Kodierungen in den Darstellungsarten Vektorgrafik oder Rastergrafik

Vektorgrafik: Bei der Vektorgrafik werden die Daten für einzelne Objekte (dessen "Vektoren") gespeichert. Der Vektorgrafiktyp bestimmt die möglichen Objekte. Oft finden dafür Punkte, Linien und Kreise Verwendung. Der Vektorgrafiktyp, und damit die verwendeten Objekte, orientieren sich meist stark an dem Anwendungsgebiet, für das dieser Vektorgrafiktyp geschaffen wurde, z. B. technischer Entwurf von Gebäuden oder Geräten. Die Objekte werden mathematisch durch eine Anzahl von Werten definiert, z. B. Anfangsposition, Länge oder Radius. Vektorgrafiken sind vor allem bei technischen Zeichnungen verbreitet (z. B. CAD-Systeme).

Vorteile:

- Grafikobjekte können (meist) ohne Qualitätsverlust verschoben, skaliert oder mit Farbe versehen werden.
- Ein Vektorbild besteht aus vielen verschiedenen Einzelobjekten, die sich bei Veränderungen gegenseitig nicht beeinflussen.
- Im Vergleich zu Rastergrafiken wird hier meist weniger Speicherplatz benötigt, da eine geringere Anzahl von Informationen gespeichert wird (z. B. einen Kreis als Matrix/Raster aus Farbwerten zu beschreiben, wie bei der Rastergrafik, ist aufwendiger, als einen Kreis mit Mittelpunkt und Radius zu beschreiben).
- Die Vektorgrafiken sind (meist) nicht an eine bestimmte Auflösung gebunden, d. h. sie passen sich den Möglichkeiten des Ausgabegeräts an.

Nachteile:

- Zur Bildschirm- und Druckerausgabe müssen die Vektorgrafiken gerastert werden, da diese Geräte nur Punkte darstellen können. Ein Treppeneffekt ist die Folge.

- Eine direkte Umwandlung von Rastergrafik oder Vektorgrafiken eines Typs in eine Vektorgrafik anderen Typs ist meist nur schwer und mit viel Aufwand zu realisieren. (Bei Vektorgrafiken geht es darum, die Objekte des Bildes darzustellen. Wie findet man aber solche bei Rastergrafiken? Einfach jeden Punkt als Objekt [z. B. als Rechteck] zu deklarieren, würde keinen der oben genannten Vorteile bringen.)
- Die meisten Bilder liegen nach dem Einlesen in einem Rastergrafikformat vor.

Rastergrafik: Bei der Rastergrafik (auch Bitmapgrafik) wird über das Bild ein Raster gelegt und die einzelnen Punkte bzw. ihre Farbe (im Farbmodell) abgespeichert. Damit besteht ein Rasterbild aus einer Menge von Pixeln (Bildpunkten), die eine bestimmte Farbe haben. Bei genügend vielen Pixeln pro Fläche und genügend Farben kann das menschliche Auge die einzelnen Pixel nicht mehr wahrnehmen und es entsteht der Effekt eines "fotorealistischen" Bildes.

Vorteile:

- einfaches Abspeichern von Bildern, es müssen keine Objekte bekannt sein
- Da die meisten Geräte (z. B. Monitor, Scanner, Drucker) zum Darstellen oder Einlesen von Bildern mit Rastern arbeiten, ist die Umwandlung in oder von Rastergrafiken für diese Geräte einfach.

Nachteile:

- Da jeder einzelne Punkt eines Bildes abgespeichert wird, kostet es meist viel Speicherplatz.
- Bei einer starker Vergrößerung eines Rastergrafikbildes kommen die einzelnen Pixel zum Vorschein und die Grafik wirkt eckig.
- Objekte des Bildes können nur noch schwer oder gar nicht automatisch identifiziert werden.

9.1.5 Farbmodell/Farbschema

Das Farbmodell gibt an, wie die Farben im Bild dargestellt werden. Es handelt sich bei diesen Farben um einen Vektor mit diskreten (z. B. ganzen) Zahlen aus einem Definitionsbereich. Dieser Vektor wird dann jeweils einem Pixel oder einem anderem Objekt (bei Vektorgrafik) zugeordnet. Der Darstellbarkeit wegen werden Schwarz und Weiß auch als Farben angesehen. Ihnen ist im allgemeinen auch ein Vektor zugeordnet. Die Anzahl der Farben, die ein Bild haben kann, wirkt sich stark auf die Größe des nötigen Speichers für dieses aus.

Im Nachfolgenden sind einige bekannte Beispiele aufgeführt.

S/W (schwarz/weiß): Das S/W-Farbmodell (S/W steht dabei für schwarz/weiß) ist wohl das älteste, einfachste und am wenigsten speicheraufwändige Farbmodell. Dabei wird pro Objekt (z. B. pro Pixel) ein Bit gespeichert, das je nach Wert (z. B. weiß = true, schwarz = false) angibt, ob das Objekt (z. B. der Punkt) weiß oder schwarz sein soll bzw. heller oder dunkler darzustellen ist, z. B. für Monitore die kein wirkliches Schwarz oder Weiß darstellen können.

Grayscale (Halbton): Beim Grayscale werden 256 Abstufungen von schwarz bis weiß unterschieden. Pro Objekt werden dafür ein Byte oder acht Bit abgespeichert. Dies wird dann als eine dieser Abstufungen interpretiert.

RGB: Das RGB-Farbmodell ist ein additives Verfahren, mit den drei Grundfarben Rot, Grün und Blau. Jeder dieser Grundfarben ist ein Wert zugeordnet: Umso größer dieser ist, desto heller ist der entsprechende Anteil dieser Grundfarbe in der erzeugten Farbe. Mit diesem Farbmodell können alle Farben mit einer Genauigkeit dargestellt werden, die mit der Definitionsbereichsgröße der einzelnen Farben ansteigt. Das Farbmodell hat seinen Ursprung in der Realisierung von Farben auf Monitoren (Braunsche Röhren), da auch dort nur drei verschiedenfarbige Punkte zu einem Pixel zusammenfasst werden: Je stärker der Elektronenstrahl einen dieser Punkte anstrahlt, desto heller erscheint er in seiner Farbe. Daher eignet sich dieses Farbmodell auch besonders gut für diese Monitordarstellungen, es muss nicht mehr viel umgerechnet werden.

Indizierte Farben: Bei indizierten Farben wird nicht immer der gleiche "Farbraum" für jedes Bild verwandt, sondern je nach Bild, je nach dem welche Farben bei ihm benötigt werden, nur diese Farben oder ähnliche. Jede dieser Farben bekommt eine "Nummer" (Indices), die in einer Tabelle der wirklichen Farbe (z. B. im RGB-Farbmodell) zugeordnet wird. Da im Bild nun viel weniger Farben dargestellt werden müssen, wird Speicherplatz gespart.

9.1.6 Komprimierung

Bei der Komprimierung geht es darum, Speicherplatz für ein Bild zu sparen. Umso weniger Speicherplatz ein Speicherformat für ein Bild benötigt, desto höher wird der Kompressionsfaktor dieses Speicherformats angesehen. Ein höherer Kompressionsfaktor eines Speicherformats führt meist zu einen höheren Kompressionsfaktor eines in ihm abgespeicherten Bildes. Leider ist die Festlegung, für welches Speicherformat der Kompressionsfaktor null ist, mehr oder weniger willkürlich, denn es können immer zu einem Speicherformat weitere Informationen über das abgespeicherte Bild hinzugefügt (z. B. Redundanzen, History) und der Kompressionsfaktor verschlechtert werden. Der Ausdruck ein Bild sei "unkomprimiert" ist somit nicht zutreffend. Zutreffender ist, dass solche Bilder einen niedrigen Kompressionsgrad haben.

Es werden zwei große Komprimierungsarten unterschieden.

Verlustfreie Komprimierung: Dabei gehen keine Informationen über das Bild verloren. Das heißt, wenn ein Bild mit einem verlustfreien Verfahren komprimiert und danach wieder hergestellt wird, ist das Abbild mit dem Original identisch.

Verlustbehaftete Komprimierung: Dabei gehen Informationen über das Bild verloren. Das heißt, wenn ein Bild mit einem verlustbehafteten Verfahren komprimiert und danach wieder hergestellt wird, ist das Abbild mit dem Original nicht mehr identisch. Es können z. B. Pixel mit einer leicht abweichenden Farbe im Abbild vorhanden sein.

In dieser Komprimierungsart sind meist höhere Komprimierungsfaktoren möglich als mit der verlustfreie Komprimierung.

9.2 Ton

Töne sind Schwingungen. Die Schwingung kann dabei durch unterschiedliche Medien (z. B. Luft oder Wasser) übertragen werden. Töne setzen sich aus elementaren Schwingungen mit unterschiedlichen Frequenzen, Amplituden, Anfangszeitpunkten und Dauer zusammen.

Deutlich wird dies beispielsweise beim Klavierspielen. Wenn ein Klavierstück gespielt wird, werden die Tasten des Klaviers zu einer bestimmten Zeit gedrückt, woraufhin ein Ton einer bestimmten Länge erklingt. Dieser Ton ist aus unterschiedlichen Frequenzen zusammengesetzt und hat eine bestimmte Lautstärke, welche durch die Amplituden der Frequenzen bestimmt wird.

Töne können demnach abgespeichert werden indem die Schwingungen des Tons aufgezeichnet werden oder die Frequenzen.

Werden die Schwingungen des Tons aufgezeichnet (wie z. B. beim wav-Format), hängt die Quallität von der Dichte der Aufzeichnungspunkte ab. Für jeden Aufzeichnungspunkt wird die Amplitude gespeichert. Werte von 44000 Aufzeichnungspunkten pro Sekunden sorgen für einen Ton in einer guten Qualität. Da pro Aufzeichnungspunkt ein paar Bytes (für die Amplitude) verbraucht werden, ist der Speicherbedarf relativ hoch (allein bei 2 Bytes für die Amplitude $44000\frac{1}{s}*2Byte = 88000Byte/s$, bei einer Aufzeichnung von nur 3 Minuten sind das rund 15,1 MB). Dieser Speicherbedarf verdoppelt sich noch, wenn die Aufzeichnung in Stereo gemacht werden soll.

Mit dem Abspeichern der Frequenzen von Tönen können im Allgemeinen bessere Komprimierungsraten erreicht werden. Diese kann noch gesteigert werden, wenn Frequenzen, die der Mensch normalerweise nicht wahrnimmt, nicht mit abgespeichert werden (wie z. B. beim MPEG-Audio [z. B. mp3]). Diese Art des Abspeicherns ist aber auch rechenaufwändiger.

Teil III
Fib-Sprachbeschreibung

In diesem Teil werden die Elemente der Fib-Multimediasprache vorgestellt und einige grundlegende Aussagen zur Fib-Multimediasprache getroffen.

10 Anforderungen

Da die klassische Bitdarstellung bei genetischen Algorithmen eine geringe Performance erwarten lässt, aber gerade wegen der Komplexität des Problems eine möglichst hohe Performance von Nöten ist, wird eine angepasste Problemdarstellung (Multimediabeschreibungssprache) gewählt.

Die Multimediabeschreibungssprache Fib sollte möglichst einfach gehalten werden, mit möglichst wenigen Alternativen, da mit dem Anstieg der zur Auswahl stehenden Alternativen auch die Anzahl der Alternativen bei der Anwendung der genetischen Operationen steigt und damit wahrscheinlich der Rechenaufwand und der Realisierungsaufwand.

Dennoch sollen mit der Multimediabeschreibungssprache kompakte Ausdrücke für "normale" (in der Anwendung auftretende) Multimediaobjekte möglich sein. Es soll also gute Komprimierungsmöglichkeiten für "normale" Multimediaobjekte geben. Das beinhaltet, dass Zusammenhänge (z. B. Farbverläufe in einer Fläche) zwischen Teilen (z. B. Pixeln der Fläche) eines Multimediaobjekts möglichst einfach dargestellt werden können.

Die Sprache sollte eindeutig, reproduzierbar und auswertbar sein, so dass ein Ausdruck immer zum gleichen Objekt ausgewertet wird. Bei der Eindeutigkeit und Reproduzierbarkeit können in der Implementierung Abstriche gemacht werden (z. B. wegen unterschiedlichen Rundungsfehlern auf unterschiedlichen Architekturen), wenn das erzeugte Multimediaobjekt so gut wie immer (in beispielsweise dem 0,999999sten Anteil der Fälle) dem original Multimediaobjekt sehr ähnlich ist. Die gültigen Sprachobjekte müssen jedoch immer auswertbar sein, da sonst Einschränkungen bei den genetischen Operatoren, die sie erzeugen, gemacht werden müssten oder nicht auswertbare Objekte entstehen könnten.

Es soll mit der Multimediabeschreibungssprache möglich sein, zumindest alle möglichen Rastergrafiken darzustellen. Die Erzeugung einer Rastergrafik aus einem Fib-Multimediaobjekt in der Multimediabeschreibungssprache soll nachvollziehbar sein. Das heißt, die Multimediabeschreibungssprache soll die Unterscheidung einzelner Objekte und deren Zusammenhang unterstützen.

Die Multimediaprogramme verändernden genetischen Operationen sollten, wenn möglich, das Programm so verändern, dass im Hypothesenraum der Multimediaprogramme diese Operatoren einen Gradientenanstieg ermöglichen. Durch Ausführung mehrerer Operatoren sollte also die Hypothese, die das Multimediaprogramm darstellt, allmählich verbessert werden können.

11 Die Multimediabeschreibungssprache

Die Multimediabeschreibungssprache trägt den Namen Fib (für funktionale Interpretation von Bildern oder "functional interpretation of bictures/bitmaps"). Dieser Namen leitet sich noch aus der ersten Version der Multimediabeschreibungssprache her, als sie nur zum Abspeichern von Bildern geeignet war.

Durch einige zusätzliche Elemente wurde die Multimediabeschreibungssprache so erweitert, dass sie beliebige Multimediadaten abspeichern kann. Die einzige Einschränkung bezüglich der Multimediadaten ist, dass sie als Eigenschaften von Punkten eines endlichen, euklidischen und diskreten (es gibt kleinste Einheiten) Raumes darstellbar sind.

Fib ist eine Vektordarstellung von Multimediadaten, also eine Darstellung von Multimediadaten mit Hilfe von Objekten. Als Grundgerüst dient ein Baum. Die Blätter sind Endpunkte, die für die Darstellung von bzw. Zuordnung zu Punkten oder Teilmultimediaobjekten dienen. In den Ästen und der Ausrichtung dieser, welche z. B. am weitesten links stehen, werden Darstellungsparameter oder Eigenschaften der Blätter kodiert, z. B. wie oft es dargestellt wird und mit welcher Farbe.

Jeder Knoten des Baums ist ein Fib-Element. Der Baum wird von der Wurzel zu den Blättern ausgewertet, wobei die Elemente die Auswertung beeinflussen.

Ein vollständiges Fib-Objekt (Baum) repräsentiert ein Multimediaobjekt.

Ein gültiges Fib-Objekt (Baum) ist zyklenfrei, um eine endliche Verarbeitungszeit zu gewährleisten. Die Elemente der Multimediabeschreibungssprache orientieren sich dabei etwas an den üblichen imperativen Programmiersprachen (z. B. C++, Java).

Alle Einheiten sind auf Grundlage des Internationalen Einheitensystems (SI) anzugeben.

Beim Auswerten eines Fib-Objekts werden für das Multimediaobjekt alle Punkte mit ihren Eigenschaften ermittelt. Ein ausgewertetes Multimediaobjekt enthält damit nur eine Liste von konkreten Punkten und ihren Eigenschaften, so dass es direkt angezeigt (durch Darstellung der Eigenschaften an den Punkten/Koordinaten) bzw. ausgewertet werden kann.

Im Nachfolgenden wird als "oben" im Fib-Objekt die Richtung bezeichnet, in der die Wurzel des Baums liegt. Die Richtung "unten" ist damit die entgegengesetzte Richtung (also weg von der Wurzel).

12 Elemente der Fib-Multimediabeschreibungssprache

In diesem Abschnitt werden die Elemente beschrieben, aus denen die Fib-Multimediabeschreibungssprache aufgebaut ist.

Im Nachfolgendem bezeichnet Obj ein Fib-Objekt.

12.1 Vektoren

Vektoren dienen zum Bereitstellen von numerischen Werten. Es gibt verschiedene Typen von Vektoren (z. B. Positionsvektor, Bereichsvektor oder Eigenschaftsvektor für RGB-Farben). Aus dem Typ eines Vektors (und eventuell den Angaben im root-Element) ergeben sich die Anzahl und die Definitionsbereiche der Elemente, die er enthalten kann. Jedes Vektorelemente ist entweder eine Zahl oder eine Variable, welche weiter oben im Fib-Objekt belegt wird.

Liegt der Werte eines ausgewerteten Elements außerhalb des Definitionsbereichs, für den das Vektorelement definiert ist, wird dieser Wert auf den nächsten Wert gerundet, der im Definitionsbereich liegt. Wenn ein Element des Vektors nicht belegt ist, wird es als der Nullwert des Definitionsbereichs ausgewertet.

Vektoren eines Typs können nur in einem ganz bestimmten Fib-Element verwendet werden.

Eine Auflistung der möglichen Vektoren ist in Tabelle 1 zu sehen.

Vektor	Beschreibung	wird verwendet in Fib-Element	Anzahl Elemente	Typ Elemente
Position	gibt die Position eines Punktes an	*point* (Punkt)	beliebig (Vektor mit 0 Elementen ist immer möglich)	Zahlen
Bereich	gibt die Grenzen eines Teilbereichs (inklusive der Grenzen) an	*area* (Bereich)	2	ganze Zahlen
Eigenschafsvektoren (es kann beliebige geben)	definiert die Werte einer Eigenschaft	*property* (Eigenschaften)	beliebig	Zahlen

Tabelle 1: Vektorentypen

12.2 Punkte

Die Punkte sind die darstellenden Elemente. An ihnen werden die gesetzten Eigenschaften ausgewertet. Punkte dienen als Blätter eines Fib-Objekts.
Ein leerer Punkte “point()” ist möglich. Der leere Punkt hat keine Auswirkungen (alle Eigenschaften gehen verloren). Genutzt werden kann er in Kombination mit anderen Elementen (z. B. dem if-Element).
Daneben gibt es auch Punkte mit leerem Positionsvektor “point(())”, diese erzeugen den Hintergrund (z. B. Hintergrundfarbe).

Die Eigenschaften des Punkts (z. B. Farbe, Ton oder/und Geruch) werden im Ast über dem Punkt durch property-Elemente bestimmt.

Syntax: $Obj = point(PositionVector)$

Kurzsyntax: $Obj = p(PositionVector)$

Beschreibung der Elemente:

$PositionVector$: die Koordinatenvektor des Punktes

Beispiel:

- $p((10;20))$; Ein Punkt an der Position $(10, 20)$.
- $p((10;x))$; Das Symbol x steht für eine Variable, die im Ast zu diesem Punkt belegt werden muss.
- $p()$; Der Punkt hat keine Auswirkungen und alle Eigenschaften werden verworfen.
- $p(())$; Der Punkt wirkt sich auf den gesamten Hintergrund aus.

12.3 Eigenschaftselement

Mit dem Eigenschaftselement "property" werden Eigenschaften für Fib-Objekte gesetzt.

Syntax: $Obj = property((value_1, ..., value_n)_{name}, Obj_1)$

Kurzsyntax: $Obj = pr((value_1, ..., value_n)_{name}, Obj_1)$

Beschreibung der Elemente:

$name$ Der $name$ gibt den Namen der Eigenschaft an. Er bestimmt den Typ des Vektors. Alle Eigenschaftsvektortypen sollten zum Vektorobertyp "property" gehören.

$value_i$ Dies ist der Wert i der Eigenschaft.

Obj_1 Das ist das enthaltende Objekt, für welches die Eigenschaft gesetzt wird.

Name	**Wert**	**#El.**	**Beschreibung**	**Beispiel**
whatever	0	0	Die Eigenschaften des Unterobjekts ist egal. Welche Eigenschaften diesem Unterobjekt auch immer zugeordnet werden, sie sind richtig.	pr($()_{whatever}$, Obj)
Farben (Alle Farben überschreiben die für das aktuelle Fib-Objekt gesetzten Farben.)				
colorRGB	1	3	Farbanteile als Rot-, Grün- und Blau-Werte	pr($(255, 16, 0)_{colorRGB}$, Obj)
colorGrayscale	2	1	Helligkeitsanteil	pr($(25)_{colorGrayscale}$, Obj)
weitere Eigenschaften				
layer	100	1	Ebenen für die Punkte (niedrige Ebenen werden von höheren überdeckt)	pr($(2)_{layer}$, Obj)
transparency	200	1	Durchsichtigkeitsanteil (für Farben) der Punkte	pr($(25)_{transparency}$, Obj)
persistent	210	0	Diese Eigenschaft ist nur für einen Zeitraum (bzw. der Dimensionsrichtung Zeit) sinnvoll. Raumpunkte mit diese Eigenschaft verlieren ihre anderen Eigenschaften nur, wenn diese zeitlich später durch jeweils eine Eigenschaft vom gleichen Typ überschrieben wird. Dies gilt natürlich nur, solange der jeweilige überschriebene Punkt die Eigenschaft $persistent$ hat. Sinnvoll ist dies, wenn beispielsweise in einem Film Objekte so lange sichbar sein sollen, wie sie nicht von anderen Objekten überschrieben werden. Dann kann das gesamte Fib-Objekt die Eigenschaft $persistent$ bekommen. Wird dann ein Objekt zu einer Zeit definiert und angezeigt, bleibt es in Zukunft solange bestehen, bis es überschrieben wird.	pr($()_{persistent}$, Obj)
Ton-Eigenschaften				
sound	300	4	ein Ton; Die Werte sind: 1. Frequenz in Hertz ($1/s$), 2. Schalldruck in Pascal Pa ($1Pa = 1N/m^2$), 3. Phasenverschiebung in Bogenmaß 4. Dauer in Sekunden. Ein Ton wirkt additiv mit anderen Tönen.	pr($(5000, 40, 0.5, 50)_{sound}$, Obj)

Name	**Wert**	**#El.**	**Beschreibung**	**Beispiel**
soundPolarized	301	$3 + \sharp Dim$	ein Ton; Die Werte sind: 1. Frequenz in Hertz ($1/s$), 2. Schalldruck in Pascal Pa ($1Pa = 1N/m^2$), 3. Phasenverschiebung in Bogenmaß, 4. Dauer in Sekunden, r = 5 bis ($3 + \sharp Dim$) Polarisierungsanteil (als Winkel in Bogenmaß) in der Dimensionsebene, welche von den jeweiligen Dimensionen $r - 4$ und $r - 3$ aufgespannt wird. ($\sharp Dim$ ist die Anzahl der Dimensionen.) Die Winkelangabe erfolgt von der $r - 3$ Achse in positiver Richtung. Ein Ton wirkt additiv mit anderen Tönen.	pr($(5000, 40, 2, 0.5, 5)_{soundPolarized}$, Obj)
soundAmplitude	305	3	die Amplitude eines Tons; Die Werte sind: 1. Schalldruck in Pascal Pa ($1Pa = 1N/m^2$), 2. Phasenverschiebung in Bogenmaß 3. Dauer in Sekunden. Ein Ton wirkt additiv mit anderen Tönen. Mit dieser Eigenschaft können Töne über ihre Amplituden mit einer bestimmten Abtastrate aufgebaut werden, wie z. B. im WAVE-Dateiformat.	pr($(40, 0.5, 0.0005)_{soundAmplitude}$, Obj)
soundBarrier	310	1	Geschwindigkeit des Schalls in Meter pro Sekunde (m/s); Damit können Objekte die Akustik verändern.	pr($(343)_{soundBarrier}$, Obj)
soundReflected	311	1	Anteil des vom Objekt zurückgeworfener Schalls; Diese Eigenschaft gilt für die Oberfläche/den Rand des Objekts und nicht für alle seine einzelnen Punkte.	pr($(50)_{soundReflected}$, Obj)
soundDamping	312	1	Anteil des von einem Punkt geschluckten Schalls	pr($(2)_{soundDamping}$, Obj)
			Physikalische Eigenschaften	
kelvin	400	1	Temperatur in Kelvin	pr($(300)_{kelvin}$, Obj)

Name	Wert	#El.	Beschreibung	Beispiel
electroMagnetic	410	$3 + \sharp Dim$	eine elektromagnetische Strahlungsquelle; Die Werte sind: 1. Frequenz in Hertz ($1/s$), 2. Amplitude in Candela cd, 3. Phasenverschiebung in Bogenmaß, 4. Dauer in Sekunden, r = 5 bis ($3 + \sharp Dim$), r = 5 bis ($3 + \sharp Dim$) Polarisierungsanteil (als Winkel in Bogenmaß) in der Dimensionsebene, welche von den jeweiligen Dimensionen $r-4$ und $r-3$ aufgespannt wird ($\sharp Dim$ ist die Anzahl der Dimensionen). Die Winkelangabe erfolgt von der $r-3$ Achse in positiver Richtung. Ein elektromagnetische Welle wirkt additiv mit anderen elektromagnetische Wellen.	pr((5,3 $* 10^{14}, 2, 0.5, 0.5, 50$ $)_{electroMagnetic}$, Obj)
			Objektbeschreibende Eigenschaften (Diese beschreiben nur die Teilobjekte ohne weitere Auswirkungen.)	
periodBegin	500	1	Zeit in Sekunden (s) ab dem Anfang des Gesamt-Multimediaobjekts, ab dem das Objekt dargestellt werden soll. Diese Eigenschaft sollte möglichst nahe an der Wurzel des Multimediaobjekt stehen. Mit der Eigenschaft kann beim Abspielen eines Multimediaobjekts bestimmt werden: in welcher Reihenfolge Teilobjekte auszuwerten sind oder/und bis wann ein Teilobjekt auszuwerten ist.	pr($(0.3)_{periodBegin}$, Obj)
periodEnd	501	1	Zeit in Sekunden (s) ab dem Anfang des Gesamt-Multimediaobjekts, bis zu dem das Objekt dargestellt werden soll. Diese Eigenschaft sollte möglichst nahe an der Wurzel des Multimediaobjekts stehen und einer “periodBegin”-Eigenschaft folgen. Mit der Eigenschaft kann beim Abspielen eines Multimediaobjekts bestimmt werden: in welcher Reihenfolge Teilobjekte auszuwerten sind oder/und bis wann ein Teilobjekt vollständig auszuwerten ist.	pr($(0.4)_{periodEnd}$, Obj)

Name	Wert	#El.	Beschreibung	Beispiel
evaluationTime	502	1	Zeit die zum Auswerten eines Multimediaobjekts benötigt wird, bezogen auf ein Multimediaobjekt, welches nur einen Punkt enthält (die Angabe ist als vielfaches der Auswertungszeit für einen Punkt zu sehen). Mit dieser Eigenschaft kann zusammen mit den Eigenschaften "periodBegin" und "periodEnd" beim Abspielen eines Multimediaobjekts eine sinnvolle Auswertungsreihenfolge und die sinnvollen Auswertungszeitpunkte der Teilobjekte bestimmt werden. Diese Eigenschaft sollte direkt nach (bzw. unterhalb /innerhalb) "periodBegin" und "periodEnd" stehen.	pr($(0.15)_{evaluationTime}$, Obj)
Eigenschaften für das komprimierte Abspeichern (Diese haben keine Auswirkungen auf die Punkte.)				
checksum	600	3	Es wird eine Checksumme für das Objekt generiert. Dabei gibt der erste Parameter die Art der Checksumme an. Der zweite Parameter gibt an, alle wieviel Bits eine Checksumme generiert werden soll, und der dritte Parameter, wieviel Bits die Checksumme haben soll. Der letzte Block der Checksummenblöcke wird nach dem Laden mit 0 aufgefüllt, so dass auch er die gewünschte Länge hat. Sind genug Bits zur Korrektur eines Fehlers vorhanden, wird eine Korrektur eines Fehlers beim Laden versucht (siehe Abschnitt 25.3.3 auf Seite 131).	pr($(1, 1024, 16)_{checksum}$, Obj)
boundSize	601	0	Für das enthaltende Objekt wird beim komprimierten Abspeichern die Grenze/Größe in Bits vermerkt. Wenn beim Laden dann ein Fehler im Fib-Objekt auftritt, können die nachfolgenden Fib-Elemente immer noch geladen werden, da deren Anfang bekannt ist (siehe Abschnitt 25.3.3 auf Seite 132).	pr($()_{boundSize}$, Obj)
Sonstige Eigenschaften				

Name	Wert	#El.	Beschreibung	Beispiel
Product Properties	240 bis 255	beliebig	Eigenschaften, die Produktspezifisch sind. Einzelne Hersteller können diese nutzen, ohne mit später definierten allgemeinen Eigenschaften inkompatibel zu werden.	

Tabelle 2: Eigenschaften (der Namensvorsatz “property::“ wurde der Übersichtlichkeit halber weggelassen; #El. ist die Anzahl der Elemente des Eigenschaftsvektors)

Tabelle 2 zeigt mögliche Eigenschaften, welche mit “property” gesetzt werden können (der Namensvorsatz “property::“ bei den Namen wurde der Übersichtlichkeit halber weggelassen). Jede Eigenschaft hat ihren eigenen Vektortyp. Jeder Eigenschaftsvektortyp hat den Obertyp ”property“. Die Definitionsbereiche der Vektortypen werden im root-Element (siehe Abschnitt 12.14 auf Seite 52) gesetzt.

In Tabelle 2 steht $\sharp Dim$ für die Anzahl der Dimensionen im Fib-Multimediaobjekt.

In Tabelle 5 auf Seite 61 sind die Eigenschaftstypen noch einmal mit ihren Standarddefinitionsbereichen aufgeführt.

Existiert an einer Position eine benötigte Eigenschaft nicht, so ist für sie der Nullvektor aus dem gültigen Definitionsbereich (eventuell den entsprechenden Standarddefinitionsbereich) anzunehmen.

Beispiel:

- $pr((255, 0, 0)_{colorRGB}, p((10; 20)))$; ein roter Punkt an der Position $(10, 20)$
- $pr((x)_{colorGrayscale}, p((10; x)))$; Das Zeichen x steht hier für eine Variable, die im Ast zu diesem Punkt belegt werden muss. Diese Variable beinflusst gleichzeitig die Position und Farbe/Helligkeit des Punktes.
- $pr((0, 0, 255)_{colorRGB}, p(()))$; der gesamten Hintergrund wird Blau gezeichnet

12.3.1 Eigenschaften für Anteile

Bei Elementen von Eigenschaften, die sich auf Anteile beziehen, wird die obere Grenze (100%) durch die obere Grenze ihres Definitionsbereichs bestimmt und ihre untere Grenze (0%) durch die untere Grenze ihres Definitionsbereichs. Deshalb ist für Elemente von Eigenschaften, die sich auf Anteile beziehen, immer die untere und obere Grenze für den minimalen und maximalen Wert im Definitionsbereichs anzugeben, auch wenn dieser im Fib-Objekt gar nicht angenommen wird.

Als Beispiel sei die Farbe Rot bei ”colorRGB“ angeführt. Dieser Farbe wird im Fib-Objekt der Definitionsbereichs von Ganzzahlen im Bereich von -10 (untere Grenze) bis 90 (obere Grenze) zugeordnet. Die Farbe Rot ist bei der Anzeige ein

Anteil des Rotwertes eines Punktes, der von $0{,}0$ für kein Rot (untere Grenze) bis $1{,}0$ für maximales Rot (obere Grenze) geht. Dann entspricht die -10 der Eigenschaft dem Wert $0{,}0$ des Rotwertes der Anzeige und 90 entspricht $1{,}0$. Der Zwischenwert 40 der Eigenschaft entspricht $0{,}5$ und der Zwischenwert 0 entspricht $0{,}1$. Zu beachten ist, dass im Definitionsbereich nicht alle Ganzzahlen zwischen -10 bis 90 vorhanden sein müssen. Der Definitionsbereich könnte aus nur acht Zahlen (z. B. $D = \{-10; 0; 3; 4; 5; 21; 40; 90\}$) bestehen, von denen -10 die kleinste und 90 die größte ist. Die -10 und die 90 müssten in jedem Fall im Definitionsbereich sein, um die Grenzen festzulegen, selbst wenn sie nicht im Fib-Objekt auftaucht. Das gleiche Verfahren ist für die anderen Farbwerte der "colorRGB“ Eigenschaft anzuwenden, welche andere Definitionsbereiche haben können.

Elemente von Eigenschaften, die Anteile sind, sind:

- "colorRGB“: alle Vektorelemente; Der Wert $0{,}0$ ist bei der Anzeige die untere Grenze und entspricht "keiner Farbe“.
- "colorGrayscale“: alle Vektorelemente; Der Wert $0{,}0$ ist bei der Anzeige die untere Grenze und entspricht "Schwarz“.
- "transparency“: alle Vektorelemente; Der Wert $0{,}0$ ist bei der Anzeige die untere Grenze und entspricht "total undurchsichtig“, 1 entspricht demnach "total durchsichtig“.
- "soundReflected“: alle Vektorelemente; Der Wert $0{,}0$ ist bei der Anzeige die untere Grenze und entspricht "keine Reflexion“.
- "soundDamping“: alle Vektorelemente; Der Wert $0{,}0$ ist bei der Anzeige die untere Grenze und entspricht "keine Dämpfung“.

12.4 Listenelement

Mit dem Listenelement “list” können mehrere Objekte zu einem Objekt zusammengefügt werden. Dabei wird eine Ausführungsreihenfolge festgelegt, um im Falle einer Überdeckung der Unterobjekte zu gewährleisten, dass immer dieselben Unterobjekte die selben anderen Unterobjekte überdecken und dies z. B. nicht immer wechselt (wegen der Eindeutigkeit von Fib-Objekten).

Syntax: $Obj = list(Obj_1, \ldots, Obj_n)$

Kurzsyntax: $Obj = l(Obj_1, \ldots, Obj_n)$

Obj_i sind die Arme/Zweige des Listenobjekts, mit $i \in \{1, \ldots, n\}$ und $n \geq 2$ (das Listenelement muss mindestens zwei Unterobjekte enthalten). Dabei wird das Obj_i vor Obj_{i+1} ausgewertet und Obj_{i+1} überdeckt Obj_i somit eventuell.

12.5 Anmerkungselement

Das Anmerkungselement (comment) dient zum Benennen oder Beschreiben von Unterobjekten. In ihm können, im Gegensatz zu allen anderen Elementen, beliebige Zeichenketten verwendet werden.

Syntax: $Obj = comment(Key, Value, Obj_1)$

Kurzsyntax: $Obj = c(Key, Value, Obj_1)$

Beschreibung der Elemente:

Key**:** der Schlüssel des Kommentars oder der Beschreibung von Obj_1

$Value$**:** der Wert des Kommentars oder der Beschreibung von Obj_1

Obj_1**:** das enthaltende Objekt, für welches der Bezeichner oder die Beschreibung gilt

Der Schlüssel Key kann eine beliebige Zeichenkette sein, wobei empfohlen wird, ihn aus den vordefinierten Schlüsseln aus Tabelle 3 zu wählen. Auf diese Weise können Schlüssel aus dem gesamten Fib-Objekt herrausgefiltert werden oder zur Suche nach bestimmten Teilobjekten verwendet werden.

Der Wert zu einem Kommentar kann eine beliebige Zeichenkette sein.

Schlüssel	**Beschreibung**	**Beispiel**
unknown	Der Typ des Kommentars ist unbekannt.	c(unknown, “ene mene mu”, Obj)
author	Autor des Fib-Objekts	c(author, “oesterholz”, Obj)
author::email	E-Mail-Adresse des Autors	c(author::email, “author@gmx.com” , Obj)
author::adress	Adresse/Anschrift des Autors	c(author::adress, “Musterstadt 123456;Musterstraße 13”, Obj)
author::telephon	Telefonnummer des Autors	c(author::telephon, “012/345/6789”, Obj)
creation::date	Erzeugungsdatum des Multimediaobjekts	c(creation::date, “2009/10/30”, Obj)
creation::time	Erzeugungszeit des Multimediaobjekts	c(creation::time, “15/57/32”, Obj)
creation:: coordinate	Erzeugungsposition des Multimediaobjekts als Geografische Koordinaten	c(creation::coordinate, “Lat = 47Â° 25Ń, Lon = 010Â° 59É”, Obj)
creation::location	Erzeugungsposition des Multimediaobjekts als Ortsangabe	c(creation::location, “Platz der Republik 1, 10117 Berlin”, Obj)

Schlüssel	**Beschreibung**	**Beispiel**
type	Typ des Objekts	c(type, “tree”, Obj)
description	Beschreibung des Objekts	c(description, “Dies bin ich beim Angeln”, Obj)
name	der Name des Objekts	c(name, “Eifelturm”, Obj)
copyright	Copyright des Objekts	c(copyright, “GPL3”, Obj)
comment	ein allgemeiner Kommentar zum Fib-Objekt	c(comment, “noch mal überarbeiten”, Obj)
link	ein Link der zum Fib-Objekt (dieser kann beim Anklicken des Objekts aufgerufen werden)	c(link, “http://www.fib-development.org”, Obj)
nextElement:: description	eine Beschreibung für das nächste Fib-Element, welches das Kommentarelement enthält	c(nextElement::description, “Dieses Element dient zur Erzeugung von Kopien.”, Obj)
nextElement:: function	Dies ist die Beschreibung der Funktion für das nächste Fib-Element, welches das Kommentarelement enthält. Ist der Schlüssel ein Name einer Dimensionsrichtung, sollte das nächste Fib-Element ein Bereichselement sein, dass eine Variable Definiert, durch welche die entsprechende Dimensionsrichtung erzeugt wird. Ist der Schlüssel beispielsweise “time”, erzeugt eine Variablenbelegung des enthaltenden Bereichselements das Fib-Multimediaobjekt für eine bestimmte Zeit. Ein weiterer mögliche Schlüsselwerte ist “scene”, dann schaltet das Bereichselement durch die Szenen des Multimediaobjekts. Durch eine derartige Kennzeichnung können schnell bestimmte Zeitpunkte oder Szenen angesprungen werden.	c(nextElement::function, “time”, Obj)

Tabelle 3: Schlüssel

12.6 Bereichselement

Das Bereichselement legt für eine Variable aus dem Bereich der ganzen Zahlen die gültigen Werte (diskrete Bereiche) fest, welche sie einnimmt. Das Bereichselement enthält, neben der Variablen und dem Objekt in dem sie gilt, eine Liste von Bereichen, welche die Variable annimmt. Die Variable gilt überall in dem enthaltenden Objekt.

Syntax: $Obj = for(Variable, (B_1, B_2, \ldots, B_n), Obj_1)$

Beschreibung der Elemente:

- $Variable$: Dies ist die Variable, welche das Bereichselement definiert.
- B_i $(i = 1 \ldots n)$: Dies sind die Bereiche. Für den Wert n gilt $n \geq 1$, es gibt also mindestens einen Bereich im Bereichselement.
- Obj_1: Dies ist das Unterobjekt, für das die $Variable$ definiert ist und welches für jede Variablenbelegung aus dem Bereich ausgewertet wird.

Ein Bereich B_i ist ein Vektor vom Grad 2, dessen zwei ganzzahligen Komponenten einen ganzzahligen Bereich festlegen, in dem die Variable gilt. Ein Bereich B_i umfasst die beiden Ganzzahlen des Bereichsvektors B_i und alle Ganzzahlen zwischen diesen. Ist eine Komponente des Vektors eine Variable, die einen nicht ganzzahligen Wert enthält, wird dieser gerundet. Beim Runden bleibt die Ziffer der Vorkommastelle unverändert, wenn die Ziffer der ersten Nachkommastelle 0 bis 4 ist, sonst (bei 5 bis 9) wird die Ziffer der Vorkommastelle bei positiven Zahlen um eins erhöht und bei negativen Zahlen um eins verringert.

Das Bereichselement umfasst als Bereich die Vereinigung aller seiner (Unter-) Bereiche B_i. Es werden also von einem Bereichselement alle Ganzzahlen durchlaufen, die in seinen (Unter-)Bereichen B_i vorkommen.

Beispiel:

- $for(x, [(1; 3), (10; 14)], Obj)$: In diesem Beispiel nimmt die Variable x im Objekt Obj hintereinander die Werte 1, 2, 3, 10, 11, 12, 13, 14 an.
- $for(x, [(1; y = 3.4985)], Obj)$: In diesem Beispiel nimmt die Variable x im Objekt Obj hintereinander die Werte 1, 2, 3 an.
- $for(x, [(1; y = 3.5)], Obj)$: In diesem Beispiel nimmt die Variable x im Objekt Obj hintereinander die Werte 1, 2, 3, 4 an.

Anmerkung: Durch diese Definition des Bereichsobjekts sind kontinuierliche Funktionen nicht zu realisieren, da das Bereichsobjekt nur einzelne Werte (ganze Zahlen) zulässt und keinen kontinuierlichen Übergang. Funktionen können nur kontinuierlich bis zu einer bestimmten Auflösung (z. B. im Bereich der ganzen Zahlen) realisiert werden.

Beispiel: die Funktion $y = x^2$ im Bereich von $x = \{0, 1, 2\}$ (nur zur Verdeutlichung einfach gewählt)

Wenn sie in der Form $for(x, [(0; 2)], fun(y, exp(x; 2), p((x, y))))$ zu realisieren versucht wird, entstehen Lücken (der Übergang von (1;1) zu (2;4), ein Punkt (x;3) fehlt).

Dies kann aber durch “Stauchung des Wertebereichs” behoben werden:
$for(x, [(0; 6)], fun(sx, div(x, 3), fun(y, exp(sx, 2), p((sx, y)))))$

Nun existiert auch ein Punkt (2;3) ($x = 5 \rightarrow sx = 5/3$ aufgerundet 2; $y = (5/3)^2 \simeq$ 2,78 aufgerundet 3)

Dies wurde zugunsten der einfacheren Realisierung und besseren Performance so gewählt. Multimediaobjekte (z. B. Bilder) die in der Fib-Multimediabeschreibungssprache kodiert sind, werden bei der Darstellung aus einzelnen Punkten (Pixeln) bestehen.

Es ist allerdings möglich, die Skalierbarkeit eines Fib-Multimediaobjekts auf andere Weise zu erreichen (siehe Abschnitt XII auf Seite 454).

12.7 Funktionen

Funktionen sind Fib-Elemente, welche einer Variable, die sie definieren, einen Wert zuordnen, der mit Hilfe einer Formel berechnet wird. Eine Funktion enthält dafür eine Unterfunktion. Eine Unterfunktion ist eine Zahl, eine Variable oder eine echte Unterfunktion.

Syntax: $Obj = fun(Variable, UF, Obj_1)$ (Statt “fun” kann auch “fkt” verwendet werden.)

Kurzsyntax: $Obj = f(Variable, UF, Obj_1)$

Beschreibung der Elemente:

- $Variable$: Dies ist die Variable, welche das Funktionselement definiert.
- UF: Dies ist die Unterfunktion der Funktion.
- Obj_1: Dies ist das Unterobjekt, in dem die $Variable$ definiert ist und welches für die ermittelte Variablenbelegung der Funktion ausgewertet wird.

12.7.1 Zahlen oder Variablen als Unterfunktionen

Eine Unterfunktion kann aus einer Zahl oder einer Variablen bestehen. Dies ist normalerweise nur in Verbindung bzw. als Unterfunktion einer echten Unterfunktion sinnvoll. Variablen müssen über dem Funktionselement, in dem sie verwendet werden, definiert werden (z. B. durch ein anderes Funktionselement).

Syntax von Zahlen: X

Beispiel für eine Zahl: 3

Syntax von Variablen: x

Beispiel für eine Variable: x

12.7.2 Echte Unterfunktionen

Jede echte Unterfunktion, wie auch jede Variable, repräsentiert genau einen Wert. Bei der Auswertung der Unterfunktion werden zuerst deren Unterfunktionen zu einem Wert ausgewertet, bevor die eigentliche Unterfunktion ausgewertet wird.

Es gibt keine unberechenbaren/nicht definierten Werte. Funktionswerte, die für die normalen mathematischen Funktion nicht definiert sind, werden nach 0 gemappt. Da die Fib-Objekte nicht den Anspruch erheben, mathematisch korrekt zu sein, sondern nur eindeutig, reproduzierbar und auswertbar zu sein, ist das Füllen von Definitionslücken angebracht.

Im Nachfolgenden sind UF_1 und UF_2 Unterfunktionen.

Addition Die Addition wird als grundlegenste aller Operationen benötigt.

Syntax: $UF = add(UF_1, UF_2)$

Beispiel:

- $fun(x, add(1, 3), Obj)$; realisiert die Funktion $x = 1 + 3 = 4$
- $fun(x, add(y, -3), Obj)$; realisiert die Funktion $x = y + (-3) = y - 3$
- $fun(x, add(add(2, y), add(z, v)), Obj)$; realisiert die Funktion $x = (2 + y) + (z + v) = 2 + y + z + v$

Subtraktion Die Subtraktion subtrahiert zwei Werte.

Syntax: $UF = sub(UF_1, UF_2)$

Beispiel:

- $fun(x, sub(1, 3), Obj)$; realisiert die Funktion $x = 1 - 3 = -2$
- $fun(x, sub(y, -3), Obj)$; realisiert die Funktion $x = y - (-3) = y + 3$
- $fun(x, sub(sub(2, y), add(z, v)), Obj)$; realisiert die Funktion $x = (2 - y) - (z + v) = 2 - y - z - v$

Multiplikation Die Multiplikation multipliziert zwei Werte.

Syntax: $UF = mult(UF_1, UF_2)$

Beispiel:

- $fun(x, mult(2,3), Obj)$; realisiert die Funktion $x = 2*3 = 6$
- $fun(x, mult(y,3), Obj)$; realisiert die Funktion $x = y*3 = 3y$
- $fun(x, add(y, mult(-2,z)), Obj)$; realisiert die Funktion $x = y+(-2z) = y-2z$

Division Die Division könnte zwar durch die Multiplikations- und Exponentialfunktion ersetzt werden ($div(a,b) = mult(a, exp(b,-1))$), da dieses aber aufwändig ist, gibt es diese Funktion separat.

Syntax: $UF = div(UF_1, UF_2)$

Beispiel:

- $fun(x, div(2,3), Obj)$; realisiert die Funktion $x = 2/3$
- $fun(x, div(y,3), Obj)$; realisiert die Funktion $x = y/3$
- $fun(x, mult(4, div(y,z)), Obj)$; realisiert die Funktion $x = 4*(y/z)$
- $fun(x, div(4,0), Obj)$; wird zu 0 ausgewertet, da $x = 4/0$ nicht definiert ist
- $fun(x, div(335,113), Obj)$; realisiert die Funktion $x = 335/113 \simeq 3{,}1415929 \simeq \pi$

Modulo Diese Funktion stellt den symmetrischen Modulo Operator bereit. Der symmetrische Modulo Operator gibt den Rest beim ganzzahliger Division aus. ($mod(x,y) = x - y * int(x/y)$; wobei int das Abschneiden der Nachkommastellen bezeichnet)

Syntax: $UF = mod(UF_1, UF_2)$

Beispiele:

- $fun(x, mod(-12.3,3), Obj)$; der zurückgegebene Wert ist -0.3
- $fun(x, mod(6.5,0.5), Obj)$; der zurückgegebene Wert ist 0
- $fun(x, mod(-4.687,-3), Obj)$; der zurückgegebene Wert ist -1.687

Exponent Die Exponentialfunktion arbeitet auf zwei Unterfunktionen, wobei der erste Wert die Basis darstellt und der zweite den Exponenten.

Syntax: $UF = exp(UF_1, UF_2)$

Beispiel:

- $fun(x, exp(2, 3), Obj)$; realisiert die Funktion $x = 2^3$
- $fun(x, exp(y, 3), Obj)$; realisiert die Funktion $x = y^3$
- $fun(x, exp(4, div(y, z)), Obj)$; realisiert die Funktion $x = 4^{y/z}$

Minimum Die Minimumsfunktion arbeitet auf zwei Unterfunktionen. Sie liefert als Ergebnis den kleinsten Wert der beiden Unterfunktionen.

Syntax: $UF = min(UF_1, UF_2)$

Beispiele:

- $fun(x, min(0, 12), Obj)$; gibt 0 zurück
- $fun(x, min(add(-2, 7), 4), Obj)$; Da der Wert von $add(-2, 7) = -2 + 7 = 5$ ist, wird die 4 zurückgegeben.

Maximum Die Maximumsfunktion arbeitet auf zwei Unterfunktionen. Sie liefert als Ergebnis den größten Wert der beiden Unterfunktionen.

Syntax: $UF = max(UF_1, UF_2)$

Beispiele:

- $fun(x, max(0, 12), Obj)$; gibt 12 zurück
- $fun(x, max(add(-2, 7), 4), Obj)$; Da der Wert von $add(-2, 7) = -2 + 7 = 5$ ist, wird die 5 zurückgegeben.

Logarithmus Die (natürliche) Logarithmusfunktion arbeitet, im Gegensatz zu den bisher vorgestellten Funktionen, nur auf einem Wert. Der natürliche Logarithmus ist zur Basis e zu bestimmen.

Syntax: $UF = ln(UF_1)$

Beispiel:

- $fun(x, ln(2), Obj)$; realisiert die Funktion $x = \ln(2) \simeq 0{,}6931$
- $fun(x, ln(-2), Obj)$; wird zu 0 ausgewertet, da $x = \ln(-2)$ nicht definiert

Sinus Auch die Sinusfunktion arbeitet auf nur einem Wert, der in Bogenmaß angegeben wird.

Syntax: $UF = sin(UF_1)$

Anmerkungen: Da die Sinusfunktionen (bzw. Kosinusfunktion) in Verbindung mit der Fouriertransformation häufig in der Bildverarbeitung vorkommt, dürfte die Sinusfunktion die Fib-Multimediabeschreibungssprache bereichern. Die Kosinusfunktion kann dabei leicht aus der Sinusfunktion gewonnen werden (durch Addition von $\pi/2$: $cos(x) = sin(x + \pi/2)$). Auch die Tangensfunktion kann mit Hilfe der Sinusfunktion hergeleitet werden ($\tan(x) = sin(x)/sin(x + \pi/2)$).

Beispiele:

- $fun(x, sin(0), Obj)$; realisiert die Funktion: $x = \sin 0 = 0$
- $fun(x, sin(1.57), Obj)$; realisiert die Funktion: $x = \sin 1.57 = \sin 3.14/2 \simeq 1$
- $fun(x, sin(y), Obj)$; realisiert die Funktion: $x = \sin y$
- $fun(x, sin(add(y, 1.57)), Obj)$; realisiert die Funktion: $x = \sin(y + 1.57) \simeq \cos y$

Arkussinus Auch die Arkussinusfunktion ist die Umkehrfunktion der Sinusfunktion. Sie gibt Werte in Bogenmaß zurück.

Syntax: $UF = arcsin(UF_1)$

Beispiele:

- $fun(x, arcsin(0), Obj)$; realisiert die Funktion: $x = \arcsin 0 = 0$

Absolutwert Die Absolutwertfunktion liefert eine positive Zahl zurück. Wenn die Eingabe der Absolutwertfunktion negativ war, wird sie mit -1 multipliziert, sonst wird sie direkt ohne Änderungen zurückgegeben.

Syntax: $UF = abs(UF_1)$

Beispiel:

- $fun(x, abs(-124), Obj)$; realisiert die Funktion: $x = |-124| = 124$

Runden Diese Funktion rundet den Wert der enthaltenden Unterfunktion auf einen ganzzahligen Wert. Beim Runden bleibt die Ziffer der Vorkommastelle unverändert, wenn die Ziffer der ersten Nachkommastelle 0 bis 4 ist, sonst (bei 5 bis 9) wird die Ziffer der Vorkommastelle bei positiven Zahlen um eins erhöht und bei negativen Zahlen um eins verringert.

Syntax: $UF = round(UF_1)$

Beispiele:

- $fun(x, round(-12.3), Obj)$; der zurückgegebene Wert ist -12
- $fun(x, round(6.5), Obj)$; der zurückgegebene Wert ist 7
- $fun(x, round(-4.687), Obj)$; der zurückgegebene Wert ist -5

Delay Status: nicht realisiert; zur Realisierung in einer späteren Version vorgesehen

Die Delayunterfunktion holt den früheren Wert einer Variable X zurück. Bei der Auswertung von Fib-Objekten werden einige Zweige mehrfach durchlaufen (z. B. für jeden Wert eines Bereichselements). Die Delayunterfunktion gibt den Wert zurück, den die Variable X vor UF_1-Aufrufen der Delayunterfunktion eingenommen hat.

Syntax: $UF = delay(X, UF_1, UF_2)$

Beschreibung der Elemente:

- X: Dies ist die Variable, deren früherer Wert zurückgegeben werden soll. (Diese kann auch [ausnahmsweise] von einem Fib-Element definiert werden, welches im gleichen Ast ist wie das Fib-Element zur Delayunterfunktion.)
- UF_1: Dies ist eine natürliche Zahl, welche angibt, aus welchem früheren Delayunterfunktionsaufruf X den Wert annehmen soll. Sollte UF_1 keine natürliche Zahl sein, wird sie auf eine natürliche Zahl gerundet.
- UF_2: Dies ist der Standardwert der zurückgegeben wird, wenn es keine frühere Belegung für UF_1 gibt, mit der die Variable X belegt war.

Die Delayunterfunktion speichert sich für jeden Aufruf a eines Durchgangs den Wert W_a, den die Variable X in diesem Aufruf a annimmt. Beim Aufruf n gibt sie den Wert W_{n-UF_1} zurück, den die Variable im Aufruf $n - UF_1$ angenommen hat. Ist $n - UF_1$ kleiner als 1, wird der Wert der Unterfunktion UF_2 zurückgegeben.

Ein Durchgang ist durch die Auswertung des gesamten Fib-Objekts bestimmt. Beispielsweise ist die Auswertung des Fib-Objekts, über das oberste root-Element ein Durchgang. Wird eine solche Auswertung über das oberste root-Element erneut gestartet, wird ein neuer Durchgang begonnen und die Delayunterfunktion verwirft dann alle früheren Werte W_a.

Beispiel:

- $delay(x, 1, 3)$; Die Variable x wird mit dem Wert belegt, den sie beim vorherigen Delayunterfunktionsaufruf hatte oder mit dem Wert 3, wenn die Delayunterfunktion das erste mal ausgewertet wird.

Anmerkung: Mit der Delayunterfunktion in Verbindung mit dem Set-Element sowie Bereichs- und Funktionselementen können beispielsweise Polygonzüge leicht erzeugt werden. Mit dem Set-Element werden die Eckpunkte des Polygonzugs festgelegt. Mit der Delayfunktion werden die Werte für frühere Eckpunkte zurückgeholt, welche dann über Bereichs- und Funktionselemente verbunden werden.

if Die if-Unterfunktion arbeitet ähnlich dem if-Element (siehe Abschnitt 12.8 auf Seite 41) . Dabei wird eine Bedingung ausgewertet und je nachdem, ob sie wahr oder falsch ist, ist der Wert der if-Unterfunktion der Wert ihrer ersten oder zweiten Unterfunktion.

Syntax: $UF = if(Condition, UF_1, UF_2)$

Beschreibung der Elemente:

- $Condition$: Dies ist die Bedingung der if-Unterfunktion, so wie in Abschnitt 12.8 auf Seite 41 beschrieben. Wenn die Bedingung $Condition$ erfüllt/-wahr/true ist, wird der Wert der Unterfunktion UF_1 zurückgegeben, ansonsten der Wert der Unterfunktion UF_2.
- UF_1: Dies ist die erste Unterfunktion der if-Unterfunktion.
- UF_2: Dies ist die zweite Unterfunktion der if-Unterfunktion.

Beispiel:

- $if(lo(x, 7), 1, sin(x))$; Es wird 1 zurückgegeben, wenn die Variable x kleiner 7 ist, sonst wird $sin(x)$ zurückgegeben.

12.8 Bedingungen mit dem if-Element

Um abhängig von Variablen Unterobjekte anzuzeigen oder nicht, dient das if-Element.

Syntax: $Obj = if(Condition, Obj_1, Obj_2)$

Wenn die Bedingung $Condition$ erfüllt/wahr/true ist, wird das Obj_1 ausgewertet, sonst das Obj_2.

Folgende Bedingungen/$Conditions$ sind möglich (UF_i sind Unterfunktionen, wie für Funktionselemente in Abschnitt 12.7 auf Seite 35 beschrieben):

- $true$: Die Bedingung ist wahr.
- $false$: Die Bedingung ist falsch (=nicht wahr).
- $not(Condition_1)$: die Bedingung ist wahr, wenn die $Condition_1$ falsch (bzw. nicht wahr) ist, sonst ist die Bedingung falsch
- $or(Condition_1, Condition_2)$: Die Bedingung ist wahr, wenn mindestens eine der Bedingungen $Condition_1$ oder $Condition_2$ wahr ist, sonst ist sie falsch.
- $and(Condition_1, Condition_2)$: Die Bedingung ist wahr, wenn die beiden Bedingungen $Condition_1$ und $Condition_2$ wahr sind, sonst ist sie falsch.
- $xor(Condition_1, Condition_2)$: Die Bedingung ist wahr, wenn genau eine der Bedingungen $Condition_1$ oder $Condition_2$ wahr ist, sonst ist sie falsch.
- $eqInt(UF_1, UF_2)$: Die Bedingung ist wahr, wenn der auf eine ganze Zahl gerundete Wert von UF_1 gleich dem auf eine ganze Zahl gerundete Wert von UF_2 ist, sonst ist sie falsch. Der direkte Vergleich von Gleitkommazahlen wurde verworfen, da er aufgrund von Rundungsfehlern leicht zu unterschiedlichen Ergebnissen auf unterschiedlichen Systemen führen kann.
- $lo(UF_1, UF_2)$: Die Bedingung ist wahr, wenn UF_1 kleiner als UF_2 ist ($UF_1 < UF_2$), sonst ist sie falsch.
- $gr(UF_1, UF_2)$: Die Bedingung ist wahr, wenn UF_1 größer als UF_2 ist ($UF_1 > UF_2$), sonst ist sie falsch.

Anmerkungen: Das if-Konstrukt ist eines der mächtigsten Programmiersprachenkonstrukte, deshalb wurde es in die Fib-Multimediabeschreibungssprache mit aufgenommen.

Beispiele:

- $Obj = if(lo(x, 7), Obj_1, Obj_2)$; Das Obj_1 wird ausgewertet, wenn die Variable x kleiner 7 ist, sonst wird Obj_2 ausgeführt.
- $Obj = if(and(gr(add(x, 2), 3), lo(x, 7)), Obj_1, Obj_2)$; Das Obj_1 wird ausgeführt, wenn die Variable x plus 2 größer 3 und x kleiner 7 ist (also wenn x eine Zahl zwischen 1 und 7 ist), sonst wird Obj_2 ausgeführt.

12.9 Externe Objekte aufrufen

Externe Objekte sind Fib-Objekte, welche nicht im aktuellen Fib-Teilobjekt definiert werden. Diese können aus dem Wurzelelement ($root$) oder der Fib-Objektdatenbank (siehe Abschnitt 13 auf Seite 72) kommen. Auf diese Weise können Teile von Fib-Objekten im gleichen Fib-Objekt mehrfach verwendet oder von verschiedenen Fib-Objekten wiederverwendet werden.

Syntax:

$$\begin{aligned} Obj &= obj(Identifier, (V_1, \ldots, V_n), (OutVar_1, \ldots \\ &\quad , OutVar_{v_1}, Obj_1), \ldots, (OutVar_1, \ldots, OutVar_{v_m}, Obj_m)) \end{aligned}$$

Beschreibung der Elemente:

- $Identifier$: Dies ist ein Identifizerer (eine eindeutige Ganzzahl) für das Fib-Objekt, welches verwendet werden soll. Es dürfen nur Fib-Objekte verwendet werden, die nach dem aktuellem Fib-Objekt stehen (um Rekursion zu vermeiden), wobei in der Reihenfolge zuerst die Fib-Objekte aus den root-Elementen und dann die aus der Fib-Datenbank betrachtet werden. Von den root-Elementen werden nur die root-Elemente geprüft, in denen das aktuelle Fib-Objekt vorhanden ist, aber keine root-Elemente, welche das aktuelle Fib-Objekt nicht enthalten. Wenn das externe Objekt aus der Fib-Datenbank kommt, ist der $Identifier$ negativ, sonst ist er positiv. Bei der Suche nach dem externen Objekt (mit dem $Identifier$) in den root-Elementen werden jeweils die root-Unterelemente des nächsten root-Elements $root_1$ durchsucht, in dem das Fib-Objekt vorkommt, für welches das externe Objekt benötigt wird, wobei nur root-Unterelemente betrachtet werden, die dabei nach dem root-Unterelement stehen, in dem das obj-Element (Externe-Objekt-Element) vorkommt. Also werden zuerst die $Identifier$ im nächsten root-Element vom externen-Objekt-Element ausgehend untersucht, und dann jeweils im root-Element, in dem das zuletzt untersuchte root-Element $root_1$ existiert, die $Identifier$ der root-Elemente, welche nach dem untersuchten root-Element $root_1$ stehen. Die in Abschnitt 12.14.4 auf Seite 70 definierte Reihenfolge für root-Elemente bestimmt, wie diese bei der Suche nach einem bestimmten $Identifier$ durchlaufen werden. Dort ist auch noch einmal angegeben, welche und in welcher Reihenfolge die root-Elemente auf der Suche nach einem $Identifier$ durchlaufen werden.

- $(V_1, \ldots, V_n)$: Der Vektor mit den Eingabewerten, welche für das verwendete Fib-Objekt benötigt werden.

- V_i: Dies sind die Eingabewerte, welche für das verwendete Fib-Objekt benötigt werden. Dabei wird der Eingabewert V_i der i'ten Eingabevariable des entsprechenden root-Elements zugeordnet.

- Obj_o: Dies sind die Teil-Fib-Objekte, welche für das verwendete Fib-Objekt benötigt werden. Alle Teil-Fib-Objekte werden vor dem aktuellen Objekt aufgelöst. Die Teil-Fib-Objekte sind ganz normale Fib-Objekte, welche selbst wieder externe Objekte enthalten können. Alle in Teil-Fib-Objekten enthaltenden externe Objekte werden zum aktuellen Objekt aufgelöst. "Aufgelöst" meint dabei, dass sie geladen und in das Fib-Objekt eingebunden werden (enthaltende Variablen werden vorhandenen Variablen zugeordnet), ohne dass sie dabei ausgewertet werden. Die Teil-Fib-Objekte müssen im root-Element mit dem entsprechenden Identifizerer $Identifier$ vorgesehen werden. Im entsprechenden root-Element sollte also die gleiche Anzahl von Ausgabevariablenlisten mit der gleichen Anzahl von Ausgabevariablen vorhanden sein.

- $OutVar_{g_o}$: Dies sind die Ausgabevariablen, welche für das verwendete Fib-Objekt Obj_o vom aktuellen externen Fib-Objekt bereitgestellt werden. Sollte eine Ausgabevariable dennoch nicht bereitgestellt werden, so wird sie auf 0 gesetzt.

Die Anzahl und Reihenfolge der Teil-Fib-Objekte, Eingabe- und Ausgabevariablen muss mit der Definition des externen Objekts (dem root-Element, dem der $Identifier$ zugeordnet ist) übereinstimmen.

Anmerkungen: Das obj-Element (Externe-Objekt-Element) ist nach dem root-Element eines der kompliziertesten Fib-Elemente. Durch die Wiederverwendung von Teilobjekten oder -funktionen sollte sich diesen Aufwand aber lohnen.

Beispiele:

- $obj(-3, (x, y))$: Das Datenbank-root-Objekt mit dem $Identifier$ -3 wird hier eingesetzt. Die Variablen x und y sind dabei dessen Eingabeparameter (z. B. können diese die obere linke Ecke der Anzeigeposition des eingesetzten Objekts angeben).

- $obj(5, (x_1, y_1, 3), (x_2, y_2, r, pr((r, 0, 0)_{colorRGB}, p((x_2; y_2))))))$: Das root-Objekt mit dem $Identifier$ 5 wird hier eingesetzt. Dieses Objekt kann das Objekt $pr((r, 0, 0)_{colorRGB}, p((x_2; y_2))$ verwenden, wobei der Rotanteil und die Position des Punktes über Ausgabeparameter/-varaiblen r, x_2 und y_2 bestimmt werden.

12.10 Externe Unterobjekte

Externe Unterobjekte sind Objekte, die beim Auswerten des aktuellen Fib-Objekts schon bereitgestellt werden (siehe die Teil-Fib-Objekte Obj_o unter Abschnitt 12.9 auf Seite 43).

Syntax: $Obj = sub(Number, (V_1, \ldots, V_n))$

Beschreibung der Elemente:

- $Number$: Dies ist die Nummer des externen Unterobjekts (entspricht einem Ausgabevariablenvektor im nächsten root-Element), welches hier eingesetzt werden soll. Für die $Number$ kann nur ein fester Ganzzahlwert eingesetzt werden. Der Definitionsbereich der $Number$ ergibt sich implizit aus der Anzahl der Unterobjekte (bzw. Ausgabevariablenvektoren) des nächsten root-Elements (das root-Element in dem das sub-Element im Haupt-Fib-Objekt vorkommt); siehe Abschnitt 12.14 auf Seite 52 . Sollte der Wert der angegeben Nummer außerhalb des gültigen Definitionsbereichs liegen (dies entspricht einem Fehler), wird er auf den nächsten Wert, welcher im Definitionsbereich liegt, gerundet.

- $(V_1, \ldots, V_n)$: Der Vektor mit den Ausgabewerten, welche für das verwendete Fib-Objekt benötigt werden.

- V_i: Dies sind die Ausgabewerte, welche im entsprechenden Unterobjekt verwendet werden können. Für diese sind Ausgabevariablen ($OutVar_{i_{Number}}$) im externen Objekt (obj) Element definiert. Die Ausgabewerte haben die gleiche Anzahl und Reihenfolge, wie die des entsprechenden $Number$'ten Ausgabevariablen im zugehörigen externen Objekt Element (obj). Sollte dennoch die Anzahl einmal nicht übereinstimmen (entspricht einem fehlerhaften Fib-Objekt), werden Variablen, die zu viel sind, ignoriert und für Variablen, die fehlen, der Nullwert des Definitionsbereichs der entsprechenden Ausgabevariable im root-Element eingesetzt.

Anmerkungen: Über externe Unterobjekte können einem Fib-Objekt Multimediainformationen bereitgestellt werden, welche an bestimmten Stellen eingesetzt werden können. Dies ist beispielsweise nützlich, wenn das aktuelle Fib-Objekt eine Berechnung oder Funktion realisiert, welche umfangreicher ist und in anderen Fib-Objekten öfter benötigt wird.

Sollte ein externes Unterobjekt bei der Auswertung eines root-Elements nicht existieren, wird für dieses der leere Punkt $p()$ (Punkt ohne Auswirkungen) eingesetzt. Externe Objekte können selbst im obersten root-Element nützlich sein, wenn beispielsweise das eigentliche Multimediaobjekt Obj nur einen Rahmen darstellt, in dem ein anderes Bild, ab einer bestimmten Startposition (definiert über die Ausgabewerte) eingesetzt werden soll.

Beispiele:

- $sub(1)$: An dieser Stelle wird das erste Unterobjekt des nächsten root-Elements eingesetzt.

- $sub(1, (x, y, 7))$: An dieser Stelle wird das erste Unterobjekt des nächsten root-Elements eingesetzt. Dabei stehen für das eingesetzte Unterobjekt die

Variablen x, y und der Wert 7 bereit. Diese Werte könnten beispielsweise für die Position (x, y) und den Rotanteil 7 des eingesetzten Punktes verwendet werden.

12.11 Definitionsbereichseigenschaften abrufen

Status: nicht realisiert; zur Realisierung in einer späteren Version vorgesehen

Dieses Fib-Element dient dazu, die Parameter der Definitionsbereiche abzurufen.

Syntax: $Obj = domainProperty(Variable, Type[.Element]*, Mode, Obj_1)$

Syntax: $Obj = dp(Variable, Type[.Element]*, Mode, Obj_1)$

Beschreibung des Elemente:

- $Variable$: Dies ist die Variable, welche das Definitionsbereichselement definiert. Kann kein Eigenschaftswert für die Variable ermittelt werden, wird die Variable mit 0 belegt.
- $Type$: Dies ist der Typ des Definitionsbereichs, dessen Werte ermittelt werden sollen. Mögliche Typen sind in Tabelle 5 auf Seite 61 aufgeführt.
- $Element$: Wenn es sich beim Definitionsbereich um einen Definitionsbereich für Vektoren handelt, muss mit $Element$ die Nummer des Vektorelements angegeben werden, für welche ein Eigenschaftswert zurückgegeben werden soll. Es können auch mehrere $Element$ Angaben aufeinander folgen, wenn mehrere Definitionsbereichsvektoren verschachtelt sind. Da die $Variable$ nur mit Skalaren belegt werden kann, kann auch kein Vektor mit dem Definitionsbereichselement ermittelt werden, sondern es muss immer ein Vektorelement ausgewählt werden. Die Zählung der Vektorelemente eines Vektors beginnt bei 1.
- $Mode$: Dieses Element gibt an, welcher Eigenschaftswert des Definitionsbereichs ausgewählt werden soll. Mögliche Werte sind in Tabelle 4 aufgeführt.
- Obj_1: Dies ist das Unterobjekt in dem die $Variable$ definiert ist und welches für die Variablenbelegung des Definitionsbereichselement ausgewertet wird.

Um den richtigen Definitionsbereich zu finden werden zuerst die Definitionsbereiche des nächst höheren root-Elements durchsucht, dann seine Definitionsbereiche für Werte. Wenn auch darin kein entsprechender Definitionsbereich gefunden wurde, wird das nächst höheret (zum ersten root-Element) root-Element durchsucht und so weiter.

Anmerkungen: Mit dieses Element können Eigenschaften der Umgebung erfragt werden. Dies kann beispielsweise nützlich sein, wenn ein Multimediaobjekt im

Name	Wert	Beschreibung
null	0	Zurückgegeben wird der Nullwert des Definitionsbereichs.
min	1	Zurückgegeben wird das Minimum des Definitionsbereichs.
max	2	Zurückgegeben wird das Maximum des Definitionsbereichs.
size	3	Zurückgegeben wird die Größe (also $Maximum - Minimum$) des Definitionsbereichs.
scaling	4	Zurückgegeben wird der Skalierungsfaktor des Definitionsbereichs. Bei Definitionsbereichen die nicht skaliert werden, wird 1 zurückgegeben.
Unskalierte Werte (Bei Definitionsbereichen, die nicht skaliert werden, entsprechen diese den skalierten Werten [z. B. ist dann die Rückgabe für "unscaled min" gleich der von "min"])		
unscaled null	10	Zurückgegeben wird der unskalierte Nullwert des Definitionsbereichs.
unscaled min	11	Zurückgegeben wird das unskalierte Minimum des Definitionsbereichs.
unscaled max	12	Zurückgegeben wird das unskalierte Maximum des Definitionsbereichs.
unscaled size	13	Zurückgegeben wird die unskalierte Größe (also $(unskaliertes\ Maximum) - (unskaliertes\ Minimum)$) des Definitionsbereichs.

Tabelle 4: Mögliche abrufbare Eigenschaften eines Definitionsbereichs

Nachhinein über die Angabe der Dimensionsdefinitionsbereiche skaliert werden soll. Die dargestellten Objekte können dann automatisch über die Erfragung der Eigenschaften des Dimensionsdefinitionsbereiche in Größe und Position angepasst werden.

Beispiele:

- $dp(x, dim.1, min, Obj_1)$: Die Variable x wird mit dem minimalen Wert des Definitionsbereichs für die erste Dimension belegt.
- $dp(x, property(colorRGB).2, max, Obj_1)$: Die Variable x wird mit dem maximalen Wert des Definitionsbereichs für Grün (zweites Vektorelement) von RGB-Farben belegt.
- $dp(x, property(colorGrayscale).1, size, Obj_1)$: Die Variable x wird mit der Größe des Definitionsbereich für SW-Farben belegt. Hier muss die 1 für das erste Vektorelement mit angegeben werden, da der Definitionsbereichs für SW-Farben ein Vektor ist, auch wenn er nur ein Element enthält.
- $dp(x, matrix.3.2, null, Obj_1)$: Die Variable x wird mit dem Nullwert des zweiten Unterdefinitionsbereichs des dritten Unterdefinitionsbereichs des Matrixdefinitionsbereichs belegt.

12.12 Set-Element

Das set-Element legt für eine Anzahl von Variablen hintereinander Sätze von Werten fest, mit denen die Variablen belegt werden. Die Variablen gelten überall in dem enthaltenden Objekt.

Syntax: $Obj = set((Variable_1, \ldots, Variable_n), [DomainNr,]((W_{1.1}, \ldots, W_{n.1}), \ldots, (W_{1.k}, \ldots, W_{n.k})), Obj_1)$

Beschreibung der Elemente:

- n: Entspricht der Anzahl der Elemente eines Satzes. Es muss mindestens ein Elemente geben ($n \geq 1$).
- k: Entspricht der Anzahl der Sätze von Werten, mit dem die Variablen belegt werden. Es muss mindestens einen Satz geben. ($k \geq 1$)
- $Variable_i$: Dies ist die Variablen, welche das set-Element definiert.
- $DomainNr$: Dies ist die Nummer des Definitionsbereichs für das set-Element. Diese Angabe ist optionale, der Standardwert ist 0. Sollte kein set-Definitionsbereich mit dieser Nummer existieren, wird der set-Definitionsbereich mit der nächst kleineren Nummer verwendet. Durch die Verwendung verschiedener, an die einzelnen set-Elemente angepasster, Definitionsbereiche kann der Speicheraufwand beim komprimierten Abspeichern optimiert werden.

- $W_{i.g}$ mit $(i = 1 \dots n)$ $(g = 1 \dots k)$: Dies sind die zu setzenden Werte für die Variablen.
- $(W_{1.g}, \dots, W_{n.g})$: Dies ist der Vektor mit den zu setzenden Werten.
- Obj_1: Dies ist das Unterobjekt, in dem die Variablen $Variable_i$ definiert sind und welches für jede Variablenbelegung ausgewertet wird.

Die Variablen $(Variable_1, \dots, Variable_n)$ werden nacheinander mit den einzelnen Sätzen von Werten $(W_{1.g}, \dots, W_{n.g})$ belegt, wobei die Variable $Variable_i$ immer nur mit dem Wert $W_{i.g}$ $(g = 1, \dots k)$ belegt wird. Damit gibt es k-Belegungen von Variablen, wobei die Variable $Variable_i$ zuerst mit dem Wert $W_{i.1}$ belegt wird, dann mit $W_{i.2}$ usw. Sollte ein Element $W_{i.g}$ eine Variable sein, so wird die $Variable_i$ entsprechend mit dem Wert der Variable $W_{i.g}$ belegt.

Beispiel:

- $set((x, y), ((1, 2), (3, 8), (3, -8)), Obj)$; In diesem Beispiel nehmen die Variable x und y im Objekt Obj hintereinander die Werte $(x = 1, y = 2)$, $(x = 3, y = 8)$ und dann $(x = 3, y = -8)$ an.
- $set((x, y, z), 3, ((1, 2, 3), (3, 8, 15), (3, -8, -1)), Obj)$; In diesem Beispiel nehmen die Variable x, y und z im Objekt Obj hintereinander die Werte $(x = 1, y = 2, z = 3)$, $(x = 3, y = 8, z = 15)$ und dann $(x = 3, y = -8, z = -1)$ an. Der Definitionsbereich für das Set-Element, ist der dritte set-Definitionsbereich.

Anmerkung: Nicht alle Abhängigkeiten von mehreren Variablen können einfach durch Funktionen dargestellt werden. Deshalb bietet das set-Element die Möglichkeit, mehreren Variablen hintereinander Sätze von Werten zuzuordnen.

Es ist beispielsweise denkbar, ein Datenbankobjekt zu schaffen, welches einen Zeichensatz (“font”) kodiert. Über Eingabeparameter dieses Datenbankobjekts werden der Buchstabe und die Darstellungsposition des Buchstaben festgelegt. Es kann allerdings nicht davon ausgegangen werden, dass die Eingabeparameter für die Buchstaben eines Textes in einer einfachen funktionalen Abhängigkeit zueinander stehen. Mit dem set-Element können einfach für jeden Buchstaben des Textes die Werte für die Eingabeparameter angegeben werden.

12.13 Matrixelement

Das Matrixelement stellt eine Matrix da. Das Matrixelement arbeitet ähnlich wie das Set-Element, nur werden eine Anzahl von Laufvariablen automatisch generiert.

Syntax:

$$
\begin{aligned}
Obj \;=\; & matrix((Variable_1, \ldots, Variable_d, Variable_{d+1}, \ldots, Variable_{d+i}),\\
& [DomainNr,]((Startvalue_1, Endvalue_1), \ldots, (Startvalue_d, Endvalue_d)),\\
& ((W_{1.1}, \ldots, W_{i.1}), \ldots, (W_{1.k}, \ldots, W_{i.k})), Obj_1)
\end{aligned}
$$

Beschreibung der Elemente:

- d: Entspricht der Anzahl der Dimensionen der Matrix. Diese muss mindestens eins sein. ($d \geq 1$).
- i: Entspricht der Anzahl der Werte pro Satz von Werten ($i \geq 0$).
- k: Entspricht der Anzahl der Sätze von Werten, mit dem die Variablen belegt werden ($k \geq 0$).
- $Variable_v$: Dies ist die Variablen, welche das Matrixelement definiert.
- $DomainNr$: Dies ist die Nummer des Definitionsbereichs für das Matrixelement. Diese Angabe ist optionale, der Standardwert ist 0. Sollte kein Matrix-Definitionsbereich mit dieser Nummer existieren, wird der Matrix-Definitionsbereich mit der nächst kleineren Nummer verwendet. Durch die Verwendung verschiedener, an die einzelnen Matrixelemente angepasster, Definitionsbereiche kann der Speicheraufwand beim komprimierten Abspeichern optimiert werden.
- $(Startvalue_h, Endvalue_h)$: Vektor für den zu durchlaufenden Bereich der (/ Matrixgröße in) Dimension h
- $Startvalue_h$: Dies ist der Startwert der Laufvariable für die h'te Dimension.
- $Endvalue_h$: Dies ist der Endwert der Laufvariable für die h'te Dimension.
- $W_{a.b}$ mit $(a = 1 \ldots i)$ und $(b = 1 \ldots k)$: Dies sind die zu setzenden Werte oder Variablen.
- $(W_{1.b}, \ldots, W_{n.b})$: Dies ist der Vektor mit den zu setzenden Werten.
- Obj_1: Dies ist das Unterobjekt in dem die Variablen $Variable_i$ definiert sind und welches für jede Variablenbelegung ausgewertet wird.

Das Matrixelement stellt eine Matrix mit d-Dimensionen dar, dessen Elemente jeweils aus i-Werten bestehen.

Beim Matrixelement durchlaufen die definierten Lauf-/Dimensions-/Indexvariablen $(Variable_1, \ldots, Variable_d)$ jeweils den entsprechenden angegeben Bereich $Startvalue_h$ bis $Endvalue_h$ in Einerschritten. Für jeden Ganzzahlwert der $Variable_h$ werden dabei alle Ganzzahlwerte der Variablen $Variable_{h-1}$ durchlaufen. Für jede Wertebelegung der Dimensionsvariablen $(Variable_1, \ldots, Variable_d)$ werden die Wertevariablen $(Variable_{d+1}, \ldots, Variable_{d+i})$ mit dem nächsten Wertesatz $(W_{1.b}, \ldots, W_{i.b})$ belegt. Dies geschieht solange, bis entweder die Dimensionsvariablen $(Variable_1, \ldots, Variable_d)$ alle Werte durchlaufen haben oder es keinen nächsten Wertesatz $(W_{1.k+1}, \ldots, W_{i.k+1})$ mehr gibt. Sollte ein Element $W_{a.b}$ eine Variable sein, so wird die $Variable_{d+a}$ entsprechend mit dem Wert der Variable $W_{a.b}$ belegt.

Wenn es keine Wertevariablen gibt ($i = 0$), werden einfach nur alle Werte der Dimensionsvariablen $(Variable_1, \ldots, Variable_d)$ durchlaufen, ohne dass die Wertesätze beachtet werden.

In Listing 2 ist die Arbeitsweise des Matrixelements noch einmal mit C-Pseudocode dargestellt.

Listing 2: Pseudoalgorithmus des Matrixelements

```
void evalue( matrix ){
   a = 1;
   for ( int Variable_d = Startvalue_d; Variable_d <= Endvalue_d;
       Variable_d += 1 ){
      ...
         for ( int Variable_1 = Startvalue_1; Variable_1 <=
             Endvalue_1; Variable_1 += 1 ){
            Variable_{d+1} = W_{1.a};
            ...
            Variable_{d+i} = W_{i.a};

            obj_1( Variable_1, ..., Variable_{d+i} );

            a++;
            if ( (k < a) && (0 < i) ){
               return;
            }
         }
      ...
   }
}
```

Beispiel:

- $matrix((x, y, w), (1, 1), (3, 3), ((11), (12), (13), (21), (22), (23), (31), (32), (33)), Obj_1)$: Die Variablen x, y und w nehmen nacheinander die Werte an: $(x = 1, y = 1, w = 11)$, $(x = 2, y = 1, w = 12)$, $(x = 3, y = 1, w = 13)$, $(x = 1, y = 2, w = 21)$, $(x = 2, y = 2, w = 22)$, $(x = 3, y = 2, w = 23)$, $(x = 1, y = 3, w = 31)$, $(x = 2, y = 3, w = 23)$, $(x = 3, y = 3, w = 33)$

- $matrix((x, w), (1), (5), ((1), (2), (3)), Obj_1)$: Die Variablen x und w nehmen nacheinander die Werte an: $(x = 1, w = 1)$, $(x = 2, w = 2)$, $(x = 3, w = 3)$ Ende, da keine weiteren Belegungen für w existieren

- $matrix((x, w), (1), (2), ((1), (2), (3), (4)), Obj_1)$: Die Variablen x und w

nehmen nacheinander die Werte an: $(x = 1, w = 1)$, $(x = 2, w = 2)$ Ende, da keine weiteren Belegungen für x angenommen werden sollen

- $matrix((x, y), (2, 3), (4, 4), (), Obj_1)$: Die Variablen x und y nehmen nacheinander die Werte an: $(x = 2, y = 3)$, $(x = 3, y = 3)$, $(x = 4, y = 3)$, $(x = 2, y = 4)$, $(x = 3, y = 4)$, $(x = 4, y = 4)$

Anmerkung: Matritzen und Vektoren (eindimensionale Matritzen) werden in der Mathematik, Informatik und Bildverarbeitung häufig benutzt. Mit dem Matrixelement können beispielsweise Teilbilder direkt als Rasterbild angegeben werden oder die Werte eines Verlaufsdiagramms.

12.14 Das root-Element

Das root-Element dient als Wurzelelement eines Fib-Objekts. Es sollte alle (Umgebungs-)Informationen, welche für die Auswertung des Fib-Objekt benötigt werden, bereitstellen. Das root-Element selbst kann in keinem Fib-Element außer einem anderem root-Element enthalten sein.

Syntax:

$$\begin{aligned} Rootobj \; = \; & root([Multimediainformation], [Domains], \\ & [DomainsValues], [((InVar_1, S_1), \ldots, (InVar_v, S_v))], Obj, \\ & [((Identifier_1, Rootobj_1), \ldots, (Identifier_n, Rootobj_n))], \\ & [(DB\_Identifier_1, \ldots, DB\_Identifier_d)], [Optionalpart]) \end{aligned}$$

Alle Elemente, außer das eigentliche Multimediaobjekt (Haupt-Fib-Objekt) Obj, sind optional und können daher auch entfallen.

Beschreibung der Elemente:

- Obj: Dies ist das eigentliche Multimediaobjekt (Haupt-Fib-Objekt), welches dargestellt werden soll. Es kann wiederum mehrere (Unter-)Multimediaobjekte (z. B. Bilder) repräsentieren.

- $(Identifier_i, Rootobj_i)$: Der $Identifier_i$ ist ein Identifizierer (eine eindeutige natürliche Zahl), welcher in einem externen Teilobjekt (im obj-Element, siehe Abschnitt 12.9 auf Seite 43) verwendet werden kann. Das $Rootobj_i$ ist das zugehörige externe Teilobjekt (welches selbst durch ein root-Element eingeleitet wird). Wenn im Obj ein externes Objekt noch nicht aufgelöst wurde, werden die $Identifier_i$ nach einem passenden/gleichen Identifizierer wie dem des externen Objekts durchsucht und, wenn einer gefunden wurde, das zugehörige $Rootobj_i$ als externes Objekt aufgelöst. Dabei wird immer vom Fib-Objekt ausgehend, welches das externe Objekt ist, das nächste root-Element, in dem es enthalten ist, durchsucht und dann wiederum die root-Elemente in denen das letzte durchsuchte root-Element enthalten ist. Kann in allen durchsuchten root-Elementen bis zur Fib-Wurzel (allen in denen das externe Fib-Objekt vorhanden ist) kein passender Identifizierer gefunden werden, wird danach die Datenbank durchsucht. Wird auch dort kein passender Identifizierer gefunden, wird anstelle des externen Objekts der leere Punkt $p()$ eingesetzt.

- $Multimediainformation$: Im Teil $Multimediainformation$ sind die zur Auswertung der Multimediadaten benötigten Informationen zu finden (siehe Abschnitt 12.14.1 auf Seite 54), z. B. Fib-Version, Datenbankversion.

- $Domains$: Hier stehen die Definitionsbereiche für Werte und Variablen in den verschiedenen Vektoren (z. B. Positions- und Eigenschaftsvektoren) und Fib-Elementen (siehe Abschnitt 12.14.2 auf Seite 54). Damit wird die Anzahl der Belegungsmöglichkeiten für die einzelnen Elemente eingeschränkt, um so vorab bestimmen zu können, ob die zur Anzeige zur Verfügung stehende Hardware das Multimediaobjekt anzeigen kann.

- $DomainsValues$: Hier stehen die Definitionsbereiche für Werte in den verschiedenen Vektoren (z. B. Positions- und Eigenschaftsvektoren) und Fib-Elementen (siehe Abschnitt 12.14.2 auf Seite 54). Damit wird die Anzahl der möglichen Werte eingeschränkt und damit auch die Anzahl der Bits zum Abspeichern dieser, um so eine bessere Komprimierung erreichen zu können.

- $InVar_i$: Dies sind die Eingabevariablen für das eigentliche Multimediaobjekt (Haupt-Fib-Objekt) Obj. Diese werden vom root-Element für das Haupt-Fib-Objekt Obj definiert und von externen Quellen (z. B. dem externen Objekt in einem anderen Fib-Objekt) belegt. Selbst im obersten root-Element können hier Variablen stehen. Diese stellen Freiheitsgrade für die Multimediadaten dar, welche von externen Programmen belegt werden müssen, z. B. um in einem Fib-Objekt ein Bildarchiv zu bündeln, dessen Teilbilder über eine Eingabevariable $InVar_i$ ausgewählt werden können, oder um verschiedene Sprachen auswählen zu können. Der Definitionsbereiche jeder Eingabevariable sollte in $Domains$ angegeben werden. Der Standarddefinitionsbereich für Eingabevariablen ist $integerB(16)$. Wird für eine Eingabevariable $InVar_i$ bei der Auswertung kein Wert bereitgestellt, wird sie mit ihrem Standardwert S_i belegt.

- S_i: Dies ist der Standardwert der Eingabevariable $InVar_i$. Wenn für $InVar_i$ keine Belegung angegeben wird, wird $InVar_i$ mit S_i belegt. Dabei muss S_i aus dem Definitionsbereich von $InVar_i$ kommen.

- $DB\_Identifier_i$: Identifier der Datenbankobjekten, welche im Haupt-Fib-Objekt Obj verwendet werden

- $Optionalpart$: Hier werden optionale Informationen gespeichert, welche zur Auswertung der Multimediadaten nicht benötigt werden (siehe Abschnitt 12.14.3 auf Seite 66). Dazu gehören beispielsweise das Copyrigth, der Autor oder Beschreibungstexte.

12.14.1 Multimediainformationen

Die $Multimediainformation$ enthalten Informationen, die zum Dekodieren des Multimediaobjekts unbedingt erforderlich sind und die das Ausgabegerät auch verarbeiten können muss.

Syntax:$Multimediainformation = (Fib\_Version, Db\_Version)$

Beschreibung der Elemente:

- $Fib\_Version$: Dies ist eine natürliche Zahl für die Version der Fib-Multimediabeschreibungssprache, mit der das Multimediaobjekt kodiert wurde. Diese Nummer wird mit jeder neuen Version der Fib-Multimediabeschreibungssprache um Eins erhöht.
- $Db\_Version$: Dies ist eine natürliche Zahl für die Version der Fib-Datenbank, welche zur Dekodierung des Multimediaobjekts benötigt wird. Diese Nummer wird mit jeder neuen Version der Fib-Datenbank um Eins erhöht.

Die Versionsnummern können zu einer menschenlesbaren Form (z. B. “Fib V1.2.3”) gemappt werden. Im Element $Multimediainformation$ wird allerdings nur eine Zahl verwendet, da sonst eine bestimmte Form festgelegt würde, die nur noch schwer geändert werden kann. Es kann eine menschenlesbaren Form der Versionsnummern im optionalen Teil angegeben werden (siehe Abschnitt 12.14.3 auf Seite 66).

12.14.2 Definitionsbereiche

$Domains = ((Name_1, Dom_1), \ldots, (Name_k, Dom_k))$

$DomainsValues = ((Name_1, Dom_1), \ldots, (Name_k, Dom_k))$

Es gibt zwei Arten von Definitionsbereichsangaben:

- $Domains$: Definitionsbereiche für die möglichen Werte (auch Werte von Variablen), die ein Element annehmen kann
- $DomainsValues$: Definitionsbereiche für die (echten) Werte, die ein Element enthalten kann

Auf die Arten der Definitionsbereiche wird nachfolgend ausführlicher eingegangen.

Die Angabe eines Definitionsbereichs besteht aus zwei Teilen $Name_i$ und Dom_i.

Der erste Teil des Definitionsbereichseintrags ist der Name $Name_i$ des Elements, für das der Definitionsbereich gilt. Elemente, für die Definitionsbereiche festgelegt werden können, sind in Tabelle 5 auf Seite 61 zu sehen.

Dort sind auch für die Standarddefinitionsbereiche der Elemente eingetragen. Von den angegebenen Standarddefinitionsbereich können die skalaren Unterdefinitionsbereiche geändert werden, es sollten aber nicht die Anzahl der skalaren Unterdefinitionsbereiche geändert werden (wenn nicht anders angegeben, wie z. B. für Positionsvektoren dim). Wenn der Standarddefinitionsbereich eines Elements aus der Menge der skalare Werte kommt (z. B. $IntegerB$), sollte für das Element niemals ein Vektordefinitionsbereich gesetzt werden. Andererseits sollte für Element, deren Standarddefinitionsbereiche für Vektoren mit n-Elementen ist, immer nur Definitionsbereiche für Vektoren mit n-Elementen gesetzt werden. Auf diese Weise können beispielsweise Eigenschaften immer gleich interpretiert werden. Für Elemente (z. B. für eine Eigenschaft), welche nicht in Tabelle 5 aufgeführt werden, können die Definitionsbereiche natürlich frei eingestellt werden.

Der zweite Teil des Definitionsbereichseintrags ist der Definitionsbereich Dom_i, der für das Element $Name_i$ gelten soll. Eine Auflistung der möglichen Definitionsbereiche ist in Tabelle 6 auf Seite 63 zu sehen.

Werte außerhalb ihres Definitionsbereichs werden auf Werte im Definitionsbereich gerundet. Bei Zahlen (Skalaren) ist der gerundete Wert eines nicht gerundeten Werts der Wert mit dem minimalen Abstand zum nicht gerundeten Wert. Wenn es davon mehrere gibt, wird der kleinste Wert von ihnen als gerundeter Wert genommen.

Das Runden bei Vektoren geschieht über deren Abstand. Der Abstand zwischen zwei Vektoren ist die Summe der Abstände ihrer Elemente. Der gerundete Vektor eines Definitionsbereichs ist der Vektor im Definitionsbereichs mit dem kleinsten Abstand zum ungerundeten Vektor. Haben mehrere Vektoren einen minimalen Abstand zu dem ungerundeten Vektor, so ist der gerundete Vektor der, dessen erste n-Elemente einen minimalen Abstand zum ungerundeten Vektor haben. Wobei n, von der Anzahl der Elemente im ungerundeten Vektor ausgehend, solange um 1 verringert wird, bis ein Vektor gefunden wird oder 1 erreicht wird. Sollten auch dabei mehrere Vektoren übrig bleiben, ist der gerundeten Vektor der, dessen k'tes Element kleiner ist als das k'te Element der anderen in Betracht kommenden Vektoren, wobei k von 1 auf die Anzahl der Elemente hochgezählt wird.

Hat der zu rundende Vektor zuviele Elemente, wird die Anzahl seiner Elemente auf die richtige Anzahl reduziert. Sollte der zu rundende Vektor zuwenig Elemente haben, werden die fehlenden Elemente erzeugt und mit dem Nullwert des jeweiligen Elementedefinitionsbereichs belegt.

Jeder Definitionsbereich hat einen Nullwert. Der Nullwert eines skalaren Definitionsbereichs (ein Definitionsbereich für Zahlen), ist der Wert, den 0 gerundet auf einen Wert im Definitionsbereich ergibt. Bei Vektoren ist der Nullvektor der Vektor, der entsteht, wenn ein Vektor, dessen Elemente alle 0 sind, auf einen Vektor im Definitionsbereich gerundet wird.

Jeder skalare Definitionsbereich hat einen maximalen Wert und einen minimalen Wert. Der maximale Wert ist der größte Wert im Definitionsbereich, und der minimale Wert ist der kleinste Wert.

Die Definitionsbereiche, welche nur Ganzzahlen enthalten, haben einen Skalierungsfaktor $Scale_i$, der für das entsprechende Element $Name_i$ gilt. Dieser Faktor $Scale_i$, ist eine Gleitkommazahl, mit dem die Werte des unskalierten Definitionsbereichs multipliziert werden, um den skalierten Definitionsbereich zu erhalten. Als Standard wird der Skalierungsfaktor 1 angenommen. Der Skalierungsfaktor ist beispielsweise nützlich, wenn die horizontale Dimension nicht in Meter m, sondern Millimetern mm gemessen werden soll oder wenn die Temperatur nicht auf Grad Kelvin, sondern auf Zehntel Grad Kelvin genau angegeben ist.

In der nachfolgenden Notation wird der Skalierungsfaktor dadurch angegeben, dass der Definitionsbereich mit ihm multipliziert "$*$" wird. Ein Skalierungsfaktor von 1 wird dabei nicht angegeben. Beispielsweise bedeutet "$integerB(16)*2^{-16}$", dass der 16-Bit-Ganzzahl Definitionsbereich mit dem Skalierungsfaktor 2^{-16} skaliert wird. Dagegen ist bei nur "$integerB(16)$" der Skalierungsfaktor 1, d. h. der Definitionsbereich wird nicht skaliert.

Beispiel: Der Definitionsbereich sind die natürlichen Zahlen von 0 bis 3 (also 4 Werte) mit den Skalierungsfaktor $F = 1/2$. Möglich Werte sind: $\{ 0*1/2 = 0; 1*1/2 = 0{,}5; 2*1/2 = 1; 3*1/2 = 1{,}5 \}$. Ist nun im Element zum Definitionsbereich eine Zahl, die unskaliert 1 bedeuten würde, bedeutet sie mit Skalierung $0{,}5$.

Name	Beschreibung	Standarddefinitionsbereich
dim(Dim, $Dimmap_1$, ..., $Dimmap_{Dim}$)	Dies ist der Definitionsbereich für Positionsvektoren. Dabei gibt Dim die Anzahl der Dimensionen bzw. Elemente eines Positionsvektors an. Der Wert $Dimmap_i$ ist jeweils eine Ganzzahl, wohin (bzw. in welche Richtung) die Dimension i gemappt wird (siehe Tabelle 7 auf Seite 64 für mögliche Werte. Dabei werden dort die Zahlenwerte [in der Spalte "Wert"] angegeben). Der Definitionsbereich ist ein Vektor mit Dim-Zahlen (bzw. Skalaren). Der Standardwert für die Anzahl der Dimensionen Dim ist 2. Dieser wird angenommen, wenn kein Definitionsbereich für Positionsvektoren angegeben wird. Dann (Standardfall) geht die erste Dimension in Richtung horizontal und die zweite in Richtung vertikal. Unabhängig davon überschreiben Richtungsangaben $Dimmap_i$ in den der Definitionsbereichen für Elemente $Domains$ (auch Ererbte) immer die der Definitionsbereiche für Werte $DomainsValues$. Im Normalfall sollten allerdings die Richtungsangaben in allen Positionsvektordefinitionsbereichen gleich sein.	$vector(2, integerB(16), integerB(16))$
subfunction	Werte, die in einer Unterfunktion auftauchen können	$integerB(16)$
area	Dies ist der Definitionsbereich für Bereichselemente (siehe Abschnitt 12.6 auf Seite 34). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 2 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Unterbereiche, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite und letzte Element ist der Definitionsbereich für die Vektoren für die Unterbereiche (B_1) und ist ein Definitionsbereich für Vektoren dessen zwei Elemente ganze Zahlen sind.	$vector(2, naturalNumberB(8), vector(2, integerB(16), integerB(16)))$
inVar(i)	Definitionsbereich der i'ten Eingabevariable ($InVar_i$)	$integerB(16)$
set	Nullter Definitionsbereich für set-Elemente	$vector(3, naturalNumberB(8), naturalNumberB(32), vectorOpenEnd(integerB(32)))$

Name	Beschreibung	Standarddefinitions-bereich
set(i)	i'ter Definitionsbereich für set-Elemente	$vector(3, naturalNumberB(8), naturalNumberB(32), vectorOpenEnd(integerB(32)))$
matrix	nullter Definitionsbereich für Matrixelemente	$vector(4, naturalNumberB(8), naturalNumberB(32), vector(2, integerB(8), integerB(8)), vectorOpenEnd(integerB(32)))$
matrix(i)	i'ter Definitionsbereich für Matrixelemente	$vector(4, naturalNumberB(8), naturalNumberB(32), vector(2, integerB(8), integerB(8)), vectorOpenEnd(integerB(32)))$
extern-Subobject(Number)	Definitionsbereich für das externe Unterobjekt mit angegebenen der Nummer (siehe Abschnitt 12.10 auf Seite 44) des Multimediaobjekts. Mit Werten aus diesem Definitionsbereich können die Ausgabewerte des externe Unterobjekts mit der angegebenen der Nummer belegt werden. Für jeden externSubobjectOutput Definitionsbereich kann es im eigentlichen Haupt-Fib-Objekt Obj (externe) Unterobjekte mit der angebenen Nummer geben. Für jedes externe Unterobjekt (jede Nummer) im Haupt-Fib-Objekt sollte ein externSubobjectOutput Definitionsbereich mit entsprechender Nummer erstellt werden. Zumindest aber für das (externe) Unterobjekte mit der höchsten Nummer muss ein externSubobjectOutput Definitionsbereich erstellt werden, damit bekannt ist wieviele (externe) Unterobjekt im Haupt-Fib-Objekt verwendet werden. Die externSubobjectOutput Definitionsbereiche eines root-Elements werden nicht an seine Unter-root-Elemente weitervererbt.	$vector(0)$
property(Name)	Definitionsbereich eines Eigenschaftselements mit dem gegebenen $Name$; siehe Tabelle 2 auf Seite 30 für mögliche Namen ($Name$)	abhängig vom Typ ($Name$) der Eigenschaft
property(whatever)	Die Eigenschaften dieses Punktes sind egal. Welche Eigenschaften diesem Punkt auch immer zugeordnet werden, sie sind richtig.	$vector(0)$

Name	Beschreibung	Standarddefinitionsbereich
property(colorRGB)	Definitionsbereich für die RGB-Farben	$vector(3\ , integerB(8), integerB(8), integerB(8))$
property(colorGrayscale)	Definitionsbereich für die Schwarz-Weiß-Farbe	$vector(1, integerB(8))$
property(layer)	Anzahl der möglichen Ebenen	$vector(1, integerB(4))$
property(transparency)	Durchsichtigkeitsanteil der Punkte	$vector(1, integerB(8))$
property(persistent)	Diese Eigenschaft ist nur für einen Zeitraum sinnvoll. Raumpunkte mit dieser Eigenschaft verlieren ihre anderen Eigenschaften nur, wenn diese durch jeweils eine Eigenschaft vom gleichen Typ überschrieben wird. Dies gilt natürlich nur, solange der jeweilige Punkt die Eigenschaft $persistent$ hat. Sinnvoll ist dies, wenn beispielsweise in einem Film Objekte so lange sichbar sein sollen, wie sie nicht von anderen Objekten überschrieben werden. Dann kann der gesamte Hintergrund die Eigenschaft $persistent$ bekommen. Wird dann ein Objekt zu einer Zeit definiert und angezeigt, bleibt es in Zukunft so lange bestehen, bis es überschrieben wird. (Die möglichen Werte sind 1 für persistent und 0 für nicht persistent.)	$vector(1)$
property(sound)	ein Ton; Die Werte sind: 1. Frequenz in Hertz $(1/s)$, 2. Schalldruck in Pascal Pa $(1Pa = 1N/m^2)$, 3. Phasenverschiebung in Bogenmaß 4. Dauer in Sekunden. Ein Ton wirkt additiv mit anderen Tönen.	$vector(4, integerB(16), integerB(32), integerB(16) * 2^{-16}, integerB(32) * 2^{-8})$
property(soundPolarized)	ein Ton; Die Werte sind: 1. Frequenz in Hertz $(1/s)$, 2. Schalldruck in Pascal Pa $(1Pa = 1N/m^2)$, 3. Phasenverschiebung in Bogenmaß, 4. Dauer in Sekunden, r = 5 bis $(3 + \sharp Dim)$ Polarisierungsanteil (als Winkel in Bogenmaß) in der Dimensionsebene, welche von den jeweiligen Dimensionen $r-4$ und $r-3$ aufgespannt wird. ($\sharp Dim$ ist die Anzahl der Dimensionen.) Die Winkelangabe erfolgt von der $r-3$ Achse in positiver Richtung. Ein Ton wirkt additiv mit anderen Tönen.	$vector(3 + Dim, integerB(16), integerB(32), integerB(16) * 2^{-16}, integerB(32) * 2^{-8}, [integerB(16) * 2^{-16}]^{Dim-1})$

Name	Beschreibung	Standarddefinitionsbereich
property(soundAmplitude)	die Amplitude eines Tons; Die Werte sind: 1. Schalldruck in Pascal Pa ($1Pa = 1N/m^2$), 2. Phasenverschiebung in Bogenmaß 3. Dauer in Sekunden. Ein Ton wirkt additiv mit anderen Tönen. Mit dieser Eigenschaft können Töne über ihre Amplituden mit einer bestimmten Abtastrate aufgebaut werden, wie z. B. im WAVE-Dateiformat.	$vector(3, integerB(32) * 2^{-16}, integerB(0) * 1/44100, integerB(8) * 2^{-16})$
property(soundBarrier)	Geschwindigkeit des Schalls in Meter pro Sekunde (m/s); Damit können Objekte die Akustik verändern.	$vector(1, integerB(16))$
property(soundReflected)	Anteil des vom Objekt zurückgeworfener Schalls; Diese Eigenschaft gilt für die Oberfläche/den Rand des Objekts und nicht für alle seine einzelnen Punkte.	$vector(1, integerB(16) * 2^{-16})$
property(soundDamping)	Anteil des von einem Punkt geschluckten Schalls	$vector(1, integerB(16) * 2^{-16})$
property(kelvin)	Temperatur in Kelvin	$vector(1, integerB(16) * 2^{-4})$
property(electroMagnetic)	eine elektromagnetische Strahlungsquelle; Die Werte sind: 1. Frequenz in Hertz ($1/s$), 2. Amplitude in Candela cd, 3. Phasenverschiebung in Bogenmaß, 4. Dauer in Sekunden, r = 5 bis ($3 + \sharp Dim$), r = 5 bis ($3 + \sharp Dim$) Polarisierungsanteil (als Winkel in Bogenmaß) in der Dimensionsebene, welche von den jeweiligen Dimensionen $r - 4$ und $r - 3$ aufgespannt wird ($\sharp Dim$ ist die Anzahl der Dimensionen). Die Winkelangabe erfolgt von der $r - 3$ Achse in positiver Richtung. Ein elektromagnetische Welle wirkt additiv mit anderen elektromagnetische Wellen.	$vector(3 + Dim, integerB(64), integerB(64) * 2^{-16}, integerB(16) * 2^{-16}, integerB(32) * 2^{-8}, [integerB(16) * 2^{-16}]^{Dim-1})$
property(periodBegin)	Zeit in Sekunden (s) ab dem Anfang des Gesamt-Multimediaobjekts, ab dem das Objekt dargestellt werden soll. Diese Eigenschaft sollte möglichst nahe an der Wurzel des Multimediaobjekt stehen. Mit der Eigenschaft kann beim Abspielen eines Multimediaobjekts bestimmt werden: in welcher Reihenfolge Teilobjekte auszuwerten sind oder/und bis wann ein Teilobjekt auszuwerten ist.	$vector(1, integerB(24) * 2^{-8})$

Name	Beschreibung	Standarddefinitions-bereich
property(periodEnd)	Zeit in Sekunden (s) ab dem Anfang des Gesamt-Multimediaobjekts, bis zu dem das Objekt dargestellt werden soll. Diese Eigenschaft sollte möglichst nahe an der Wurzel des Multimediaobjekts stehen und einer "periodBegin"-Eigenschaft folgen. Mit der Eigenschaft kann beim Abspielen eines Multimediaobjekts bestimmt werden: in welcher Reihenfolge Teilobjekte auszuwerten sind oder/und bis wann ein Teilobjekt vollständig auszuwerten ist.	$vector(1, integerB(24) * 2^{-8})$
property(evaluation-Time)	Zeit die zum Auswerten eines Multimediaobjekts benötigt wird, bezogen auf ein Multimediaobjekt, welches nur einen Punkt enthält (die Angabe ist als vielfaches der Auswertungszeit für einen Punkt zu sehen). Mit dieser Eigenschaft kann zusammen mit den Eigenschaften "periodBegin" und "periodEnd" beim Abspielen eines Multimediaobjekts eine sinnvolle Auswertungsreihenfolge und die sinnvollen Auswertungszeitpunkte der Teilobjekte bestimmt werden. Diese Eigenschaft sollte direkt nach (bzw. unterhalb /innerhalb) "periodBegin" und "periodEnd" stehen.	$vector(1, integerB(24))$
property(checksum)	Es wird ein Checksumme beim komprimierten Abspeichern für das enthaltende Objekt generiert. Dabei gibt der erste Parameter die Art der Checksumme an. Der zweite Parameter gibt an, alle wieviel Bits eine Checksumme generiert werden soll, und der dritte Parameter, wieviel Bits die Checksumme haben soll. Der letzte Block der Checksummenblöcke wird nach dem Laden mit 0 aufgefüllt, so dass auch er die gewünschte Länge hat. Sind genug Bits zur Korrektur eines Fehlers vorhanden, wird eine Korrektur eines Fehlers beim Laden versucht (siehe Abschnitt 25.3.3 auf Seite 131).	$vector(3, integerB(4), integerB(8), integerB(8))$
property(boundSize)	Für das enthaltende Objekt wird beim komprimierten Abspeichern die Grenze/Größe in Bits vermerkt. Wenn beim Laden dann ein Fehler im Fib-Objekt auftritt, können die nachfolgenden Fib-Elemente immer noch geladen werden, da deren Anfang bekannt ist (siehe Abschnitt 25.3.3 auf Seite 132).	$vector(0)$

Tabelle 5: Namen von Elementen für Definitionsbereiche

Definitionsbereich-name	Beschreibung	Definitionsbereich
naturalNumberB(X) * S	X-Bit-Ganzzahl vorzeichenlos, die mit S skliert werden	$0 \dots S * (2^X - 1)$
integerB(X) * S	X-Bit-Ganzzahl mit Vorzeichen, die mit S skliert werden	$S * -(2^{X-1}) \dots S * (2^{X-1} - 1)$
naturalNumberB(32)	32-Bit-Ganzzahl vorzeichenlos (der Skalierungsfaktor 1 wird weggelassen)	$0 \dots 4294967295$
integerB(32)	32-Bit-Ganzzahl mit Vorzeichen	$-2147483648 \dots 2147483647$
naturalNumber(X) * S	Ganzzahlen von 0 bis X (vorzeichenlos), die mit S skliert werden	$0 \dots S * X$
integer(X, Y) * S	alle Ganzzahlen von X bis Y, die mit S skliert werden	$S * X \dots S * Y$
integerValues($X_1, \dots, X_N$) * S	alle angegebenen Ganzzahlen X_i, die mit S skliert werden	$\{S * X_1, \dots, S * X_N\}$
naturalNumberUL() * S	unbegrenzte positive Ganzzahlen, die mit S skliert werden	alle natürlichen Zahlen
integerUL() * S	unbegrenzte Ganzzahlen, die mit S skliert werden	alle Ganzzahlen
real(D_M, D_E)	Der Definitionsbereich sind Gleitkommazahlen. Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$. Die Definitionsbereiche D_M und D_E sind Definitionsbereiche für Ganzzahlen, wie oben definiert (z. B. integerB(8)). Dabei ist D_M der Definitionsbereich, aus dem die Werte der Mantisse kommen und D_E der Definitionsbereich, aus dem die Werte des Exponenten kommen.	$\{min(D_M) * 2^{min(D_E)}, \dots, max(D_M) * 2^{max(D_E)}\}$
realValues($X_1, \dots, X_N$)	alle angegebenen Gleitkommazahlen X_i	$\{X_1, \dots, X_N\}$
realUL()	alle Gleitkommazahlen	alle Ganzzahlen
vector($E, D_1, \dots, D_E$)	Vektor mit E Elementen, wobei das i'te Element den Definitionsbereich D_i hat	$(D_1, \dots, D_E)$
vectorValues($D_1, \dots, D_T$; $V_1, \dots, V_n$)	Der Grunddefinitionsbereich sind Vektoren, die angegeben Vektoren V_i stellen alle möglichen Vektoren dar. Dabei ist D_i der Definitionsbereich des i'ten Vektorelements (soweit dieses keine Variable ist). Bei Variablen als Vektorelemente wird einfach der Variablenbezeichner weggelassen.	$\{V_1, \dots, V_n\}$; für $i = 1, \dots, n$ $V_i = (E_1, \dots\ E_T)$; mit $E_f \in D_f, f = 1 \dots T$

Definitionsbereich-name	Beschreibung	Definitionsbereich
vectorOpenEnd($E, D_1, \ldots, D_E$)	Dies ist der Definitionsbereich für Vektor mit E oder mehr Elementen, wobei das i'te Element, für i kleiner E, den Definitionsbereich D_i hat und sonst den Definitionsbereich D_E. Dieser Definitionsbereich dient für Elemente, die Vektoren unterschiedlicher Größe enthalten können. Die Anzahl der Elemente des Vektors sollte durch das enthaltende Fib-Element festgelegt werden.	$(D_1, \ldots, D_E, \ldots, D_E)$
domainReference(Name.[Element]*)	Als Definitionsbereich wird der Definitionsbereich des Elements mit dem gegebenen Elementnamen $Name$ genommen. Mit den $Element$ Parametern kann ein entsprechender Unterdefinitionsbereich ausgewählt werden (also Beispielsweise $set.3.1$ für den ersten Unterdefinitionsbereich des dritten Unterdefinitionsbereichs des set-Elements oder $property(colorRGB).2$). Sollte kein Definitionsbereich für das Element mit dem $Name$ existieren, wird der Standarddefinitionsbereich des Elements genommen. Sollte auch dieser Standarddefinitionsbereich nicht existieren wird der allgemeine Standarddefinitionsbereich genommen. Gibt es nicht den Unterdefinitionsbereich für $Element$, wird der letzte Unterdefinitionsbereich für $Element$ genommen, der noch existiert.	der ($Element$ Unter-)Definitionsbereich für das Element mit dem Definitionsbereichnamen $Name$
defaultDomain(Domain)	Der angegebene Definitionsbereich $Domain$ wird nur verwendet, wenn es für das entsprechende Element bisher keinen anderen Definitionsbereich gibt.	$Domain$ wenn für das entsprechende Element bisher kein anderer Definitionsbereich existiert

Tabelle 6: Mögliche Definitionsbereiche

Definitionsbereiche für Elemente In den Definitionsbereichen für die Elemente $Domains$ sind alle Werte angegeben, die ein Element annehmen kann, egal ob dieser Wert explizit angegeben wird oder von einer Variablen kommt. $Domains$ gibt also alle Werte an, welche die Elemente einnehmen können. Über ihn können die

Name	Wert	Beschreibung
none	0	Kein Mapping eingestellt, bzw. es wird nirgendwohin gemappt.
horizontal	1	Die Dimension geht in die horizontale Richtung. Kleinere Werte sind links.
vertikal	2	Die Dimension geht in die vertikale Richtung. Kleinere Werte sind unten.
depth	3	Die Dimension geht in Richtung Tiefe. Kleinere Werte sind hinten.
time	4	Die Dimension ist zeitlich zu interpretieren. Kleinere Werte sind früher.
anywhere	16	Die Dimension kann in jede Richtung oder Dimension gemappt werden.
Product Dimensions	256 bis 511	Dimensionen, die produktspezifisch sind. Einzelne Hersteller können diese nutzen, ohne mit später definierten allgemeinen Dimensionen inkompatibel zu werden.

Tabelle 7: Richtungswerte für Dimensionen

minimalen und maximalen Werte ermittelt werden. Die Darstellung des Multimediageräts kann dann für diese Werte angepasst/skaliert werden. Soll beispielsweise ein Film auf einem 100 cm (Zentimeter) breiten Bildschirm angezeigt werden, und die Dimension der horizontale Dimension wird mit -5 m (Metern) bis +5 m angegeben, so werden Punkte bei dem horizontalen Wert -5 m im Fib-Objekt bei 0 cm auf dem Bildschirm dargestellt und +5 m bei 100 cm (0 m bei 50 cm usw.).

Für die Anzeige sind immer die Definitionsbereiche des obersten root-Elements ausschlaggebend. Enthaltende root-Elemente können zwar auch Definitionsbereiche angeben, diese werden aber bei der Anzeige ignoriert. Daher sind im obersten root-Element alle zur Anzeige relevanten Definitionsbereiche anzugeben.

Die in $Domains$ enthaltenden Definitionsbereiche werden an im aktuellen root-Element enthaltende root-Elemente vererbt. Enthaltende root-Elemente können aber alle ererbten Definitionsbereiche überschreiben. Die Eingabevariablen $InVar_i$ sind root-Element-spezifisch (sie gelten nicht in eventuell enthaltenden root-Elementen), ihre Definitionsbereiche werden deshalb nicht vererbt, da dies weder sinnvoll noch erwünscht ist.

Wenn ein Definitionsbereich eines Elements des Fib-Objekts nicht angegeben oder ererbt wird, hat es seinen Standarddefinitionsbereich. Gibt es für ein spezifisches Element keinen Standarddefinitionsbereich, ist sein Standarddefinitionsbereich $integerB(16)$.

Eine weitere Besonderheit ergibt sich, wenn Datenbankobjekte Definitionsbereiche erben. Denn auch für root-Elemente von Datenbankobjekten ist es sinnvoll, für einige Elemente Definitionsbereiche zu ubernehmen. Dies gilt beispielsweise für die Dimensionen eines Datenbankobjekts, die sich eventuell am angezeigten Multimediaobjekt orientieren sollen. Wenn als Beispiel ein Datenbankobjekt ein allgemeines Dreieck darstellt, dessen Größe bzw. Eckpunkte über Eingabeparameter gesetzt werden, so sind die Dimensionen für dieses Datenbankobjekt nicht sinvoll festzulegen. Aus diesem Grund wird ein Datenbankobjekt bei der Vererbung von Definitionsbereichen so behandelt, als stünde es als Unter-root-Objekt in dem root-Element, in dessen Haupt-Fib-Objekt es verwendet wird. Das Datenbankobjekt erbt also die Definitionsbereiche, welche auch für das einbindende externe Objekt Fib-Element gelten, für das es eingesetzt wird.

Bei einem obersten (Wurzel-)root-Element eines Fib-Objekts (z. B. Datenbankobjekts), können demnach Definitionsbereiche weggelassen werden, wenn diese vom aufrufenden/verwendenden Fib-Objekt oder Mechanismus bestimmt werden sollen. Bei diesen Fib-Objekten sind für die nicht definierten Definitionsbereiche im obersten (Wurzel-)root-Element die entsprechenden Definitionsbereiche für die Elementwerte in $DomainsValues$ anzugeben, um die Objekte abspeichern zu können.

Definitionsbereiche für Werte In den Definitionsbereichen für die Elementwerte $DomainsValues$ sind alle Werte angegeben, die ein echter Wert in einem Element annehmen kann. Diese Definitionsbereiche gelten demnach nicht für Werte, die von Variablen geliefert werden. Die Liste $DomainsValues$ mit Definitionsbereichen für Werte bestimmt damit den Speicherbedarf für einzelne Werte und wird deshalb im Nachfolgenden auch als **Speicherplatzliste** bezeichnet. Da die Werte in den Elementen nicht alle Werte des Definitionsbereichs einnehmen müssen, z. B. wenn meist Variablen in dem Element verwendet werden, kann die Anzahl der unterschiedlichen echten Werten in dem Element deutlich unter den möglichen Werten im Definitionsbereich für das Element $Domains$ liegen. Es ist beispielsweise denkbar, dass ein RGB-Farbbild mit $256*256*256 = 16777216$ möglichen Farben nur 16 Farben benutzt. Um die 16777216 Farben des Definitionsbereichs abspeichern zu können, sind 24 Bit nötig, für die 16 Farben reichen aber 4 Bit (ähnlich wie indexierten Farben in Bildern).

Die Definitionsbereiche für Werte $DomainsValues$ werden von den Definitionsbereichen für Elemente $Domains$ desselben root-Elements ererbt. Wenn für ein Element kein Definitionsbereich in $DomainsValues$ angegeben wird, wird der Definitionsbereich von $Domains$ als Definitionsbereich für Werte des Elements genommen. Die Definitionsbereiche, welche $Domains$ erbt, werden weitervererbt an $DomainsValues$. Vorrang bei der Vererbung haben allerdings vom gleichen root-Element (wie die $Domains$) vererbte $DomainsValues$. Ererbt wird also immer der Definitionsbereich für ein Element, welcher als nächstes im aktuellen oder einem höher gelegenen root-Element auftaucht.

Oder auch: Für ein root-Element werden die angegebenen Definitionsbereiche der $Domains$ im gleichen root-Element an die $DomainsValues$ vererbt. Dann noch fehlende Definitionsbereiche erbt $DomainsValues$ von den, eventuell ererbten, Definitionsbereichen des $DomainsValues$ des nächst höhergelegenen root-Elements.

Die in $DomainsValues$ angegebenen Definitionsbereiche müssen zu den entsprechenden Definitionsbereichen in $Domains$ kompatibl sein. Beispielsweise muss die Anzahl der Elemente in einem Vektor in einem Definitionsbereich zu einem Element in $DomainsValues$ genauso groß sein wie der entsprechende (eventuell ererbte) Definitionsbereich in $Domains$. Insbesondere muss der Wertedefinitionsbereich aus $DomainsValues$ zu einem Element eine Teilmenge des entsprechenden (allgemeinen) Definitionsbereichs aus $Domains$ für das Element sein. (Elemente dürfen nur Werte einnehmen, auf denen sie definiert sind.)

12.14.3 Optionaler Teil

Der $Optionalpart$ enthält Werte, die zur Darstellung der Multimediainformationen nicht notwendig sind. Diese optionalen Angaben können beispielsweise sein: Version des Generators, Copyrigth, Autor, Erstellungsdatum, Beschreibung, Texte, Namen und Bezeichnungen (welche wegen der Komprimierung hierher ausgelagert werden). Um Platz zu sparen, kann der $Optionalpart$ komprimiert oder gar ganz weggelassen werden.

Syntax: $Optionalpart = ((Key_1, Value_1), \ldots, (Key_n, Value_n))$

Beschreibung der Elemente:

- Key_i: Dies ist der Schlüssel des i'ten Informationssatzes. Einige mögliche Schlüssel sind in Tabelle 8 aufgelistet. Der Schlüssel darf nicht mit einem "@" beginnen, dieses Zeichen ist reserviert für ausgelagerte Werte.
- $Value_i$: Dies ist der Wert des i'ten Informationssatzes.

Schlüssel (Key)	Beschreibung	Beispiel
author	Autor des Fib-Objekts	(author, "Oesterholz")
author::email	E-Mail-Adresse des Autors	(author::email , "author@provider.com")
author::adress	Adresse/Anschrift des Autors	(author::adress , "Musterstadt 123456; Musterstraße 13")
author:: telephon	Telefonnummer des Autors	(author::telephon, "123/456/789124")
type	Typ des Objekts	(type, "tree")
description	Beschreibung des Objekts	(description, "dies bin ich beim Angeln")
name	der Name des Objekts	(name, "Eifelturm")
copyright	Copyright des Objekts	(copyright, "GPL3 ...")

Schlüssel (Key)	Beschreibung	Beispiel
version::fib	Dies ist die Menschenlesbare Form der Version der Fib-Multimediasprache, welche für das Laden des Fib-Multimediaobjekts benötigt wird.	(version::fib, “Fib V1.2.3”)
version::fibdb	Dies ist die Menschenlesbare Form der Version der Fib-Datenbank, welche für das Laden des Fib-Multimediaobjekts benötigt wird.	(version::fibdb, “Fib DB V1.2.3”)
version::-enviroment	Menschenlesbare Form der Version des genetischen Algorithmus zu Fib, welche für das Kodieren des Fib-Multimediaobjekts verwendet wurde (siehe Teil IV auf Seite 95).	(version::enviroment, “Fib Env V1.2.3”)
inVarX::N::Art	Dies sind die Informationen zum Fib-Objekt, wenn die Eingabevariable $InVar_X$ mit dem Wert N belegt wird.	(inVar1::17::..., “...”)
inVarX::N::-description	Der Wert enthält Beschreibung zum Fib-Objekt, wenn die Eingabevariable $InVar_X$ mit dem Wert N belegt wird.	(inVar2::8::description , “Dieses Bild zeigt das Wetter am dritten Tag.”)
inVarX::N::typ	Der Eintrag gibt den Typ des Fib-Objekt an, wenn die Eingabevariable $InVar_X$ mit dem Wert N belegt wird.	(inVar2::8::typ , “picture”)
inVarX::N::-identifier	Der Wert ist der Identifier des root-Objekts, welches verwendet wird, wenn die Eingabevariable $InVar_X$ mit der Wert N belegt wird. Das Haupt- Fib-Objekt sollte dann das entsprechende root-Objekt als externes Objekt direkt durch if-Elemente auswählen und aufrufen, ohne dass dafür andere Fib-Elemente, außer if- und root-Elemente Verwendung finden.	(inVar2::4::identifier , “3”)
inVarX::-dimensionD::-points	Die Eingabevariable $InVar_X$ dient zur Angabe der Anzahl von Punkten in Dimension D. Der Wert D gibt die Nummer der Dimension im root-Element an (die Zählung beginnt bei 1), für welche die Auflösung gilt. Der angegebene Wert gibt die Anzahl der Punkte an, die standardmäßig angezeigt werden. Wenn beispielsweise das Fib-Objekt ein Bild darstellt und in der ersten Dimensionsrichtung standardmäßig 1000 Punkte nebeneinander angezeigt werden sollen, ist die Angabe wie im Beispiel gezeigt.	(inVar3::dimension1::points , “1000”)

Schlüssel (Key)	Beschreibung	Beispiel
inVarX::-dimensionD::-resolution	Die Eingabevariable $InVar_X$ dient zur Angabe der Auflösung von Dimension D in Punkten pro einer SI-Einheit der Dimension. Der Wert D gibt die Nummer der Dimension im root-Element an (die Zählung beginnt bei 1), für welche die Auflösung gilt. Der angegebene Wert gibt die Auflösung an, die standardmäßig eingestellt ist. Wenn beispielsweise das Fib-Objekt ein Bild darstellt und die erste Dimension standardmäßig mit 10000 Punkten pro Meter (=10 Punkte pro mm) dargestellt werden soll, ist die Auflösung wie im Beispiel gezeigt 10000.	(inVar3::dimension1::-resolution , “10000”)
inVarX::N::-language	Die Eingabevariable $InVar_X$ dient zur Angabe der Sprache. Dabei steht der Wert N für die angegebene Sprache.	(inVar1::1::language , “deutsch”)
inVarX::N::-subtitle	Die Eingabevariable $InVar_X$ dient zur Angabe des Untertitels. Dabei steht der Wert N für den angegeben Untertitel.	(inVar2::1::subtitle , “deutsch mit Kommentaren”)
inVarX::N::-feature	Die Eingabevariable $InVar_X$ stellt ein Feature /Erweiterungen des Originalmultimediaobjekts bereit. Dabei steht der Wert N für das angegeben Feature. Besseren bzw. neueren Features sollten höhere Werte N zugeordnet werden.	(inVar2::1::feature , “Der Tisch hat jetzt echte Farben.”)
inVarX::N::-bugfix	Die Eingabevariable $InVar_X$ stellt eine Fehlerbehebung des Originalmultimediaobjekts bereit. Dabei steht der Wert N für die angegebe Fehlerbehebung. Besseren bzw. neueren Fehlerbehebungen sollten höhere Werte N zugeordnet werden.	(inVar2::3::bugfix , “Das Auto im Hintergrund ist nicht mehr unscharf.”)
inVarX::Type	Die Eingabevariable ist von einer bestimmten Art, unabhängig von ihrem Wert. Als Art können die oben aufgeführten verwendet werden (z. B. description, typ, identifier). Dadurch können Schlüssel der Art “inVarX::N” verwendet werden, ohne dass jedesmal die Art der Eingabevariable angegeben werden muss.	(inVar1::bugfix, “Die Farbe des Pinguins wird korrigiert.”)
inVarX::N	Dies sind die Informationen zum Fib-Objekt, wenn die Eingabevariable $InVar_X$ mit dem Wert N belegt wird. Die Art der Eingabevariable sollte mit “inVarX::Type” festgelegt werden.	(inVar1::17, “Der Pinguin hat einen schwarzen Rücken und keinen dunkelblauen.”)

Schlüssel (Key)	Beschreibung	Beispiel
inVarX::random	Die Eingabevariable $InVar_X$ sollte mit einem zufälligen Wert aus ihrem Definitionsbereich belegt werden.	(inVarX::random, “”)
subObjN::-description	Dies ist eine Beschreibung des N’ten Unterobjekts, also eines Fib-Objekts, das diesem root-Element von außerhalb bereitgestellt werden sollte. Die Zählung der Unterobjekte beginnt bei 1, wobei jede Liste von Ausgabevariablen $OutVar_{v_k}$ ein Unterobjekt repräsentiert.	(subObject1::description , “Dieses Teilbild wird in der oberen rechten Ecke auf einem Berg dargestellt.”)
subObjN::-outVarX::-description	Dies ist eine Beschreibung der X’ten Ausgabevariable des N’ten Unterobjekts.	(subObj2::outVar3::-description , “Die horizontale Koordinate, an der das eingebundene Bild dargestellt werden soll.”)
preview::XXX	Unter den enthaltenen root-Objekten kann es einige geben, die nicht zur Kodierung des Multimediaobjekts dienen, sondern **Vorschauen** für das Multimediaobjekt darstellen. Dabei kann es beliebig viele Vorschauunterobjekte geben. Um diese schnell erkennen zu können, enthält der optionale Teil einen “preview”-Eintrag. Der Wert des Eintrags ist der $Identifier$ des root-Objekts, welche die Vorschau kodiert. Der zweite Teil des Schlüssels (hier XXX) ist frei wählbar und sollte auf die Art der Vorschau hindeuten. Die “preview”-Einträge sollten im optionalen Teil ganz vorn stehen, um sie schneller abrufen zu können.	(preview::minipicture , “1”)
preview::-minipicture	Das root-Objekt zum angegebenen Identifizierer ist ein Vorschaubild (ohne Größenangabe).	(preview::minipicture , “17”)
preview::-minipictureX	Das root-Objekt zum angegebenen Identifizierer ist ein Vorschaubild X mal X Pixel Größe.	(preview::minipicture32 , “5”)
isPointElement	Das Objekt gibt die Daten der Punkte über Punktelemente zurück. (Gegenstück ist der Schlüssel “isPointSubObject”)	(isPointElement , “”)
isPointSubObject	Das Objekt gibt die Daten der Punkte über ein Unterobjekt zurück. Der Wert ist die Nummer des Unterobjekts für die Punktdaten. Ist der Wert leer, ist das erste Unterobjekt für die Punkte zuständig. (Gegenstück ist der Schlüssel “isPointElement”)	(isPointSubObject , “”)

Schlüssel (Key)	Beschreibung	Beispiel
isAntialiased	Das dargestellte Multimediaobjekt ist Antialiased bzw. verfügt über Kantenglättung. (Gegenstück ist der Schlüssel “isNotAntialiased”)	(isAntialiased , “”)
isNotAntialiased	Das dargestellte Multimediaobjekt ist nicht Antialiased bzw. verfügt nicht über Kantenglättung. (Gegenstück ist der Schlüssel “isAntialiased”)	(isNotAntialiased , “”)
DbObject::XXX	Der Wert zum Schlüssel ist der Identifizerer (eine Zahl) des Objekts, welches bis auf die Eigenschaft XXX identisch zum Objekt des root-Elements ist. Mögliche Werte für XXX sind beispielsweise “isPointSubObject” oder “isAntialiased”. Besitzt also das aktuelle Objekt keine Kantenglättung, so besitzt das Objekt mit dem Identifizerer des Wertes zu “DbObject::isAntialiased” Kantenglättung, ist aber ansonsten identisch.	(DbObject::isAntialiased , “-56”)

Tabelle 8: Schlüssel für den optionalen Teil

12.14.4 Reihenfolge der root-Elemente

Alle root-Elemente sind in einer bestimmten Reihenfolge geordnet. In dieser Reihenfolge kommt zuerst das höchste root-Element und dann die in ihm enthaltenden root-Elemente, mit ihren jeweiligen enthaltenden root-Elementen, in der Reihenfolge, wie sie in den jeweiligen root-Elementen definiert sind. In Diagramm 3 ist ein Beispiel für die Ordnung der root-Elemente dargestellt.

Für ein Fib-Objekt sind jeweils alle root-Objekte sichbar, die nach den Fib-Objekten (in root-Elementen) stehen, in denen es enthalten ist, und die root-Objekte, die in der Datenbank stehen.

Soll ein externes Objekt in einem Fib-Objekt aufgelöst werden, werden zuerst die Identifier im nächsten root-Element durchsucht, in dem das Fib-Objekt enthalten ist, danach im nächsten root-Element usw. Dabei werden nur Identifier überprüft, die nach dem Identifier für das root-Objekt kommen, in dem das Fib-Objekt enthalten ist. Das Haupt-Fib-Objekt steht vor allen enthaltenden root-Objekten (mit Identifiern) eines root-Elements. Am Ende werden noch die Datenbankobjekte überprüft.

Beispiel: Es sei folgende root-Element-Struktur gegeben:

$$root_0 \quad = \quad root(\ldots, Obj_0, ((1, root_1), (2, root_2), (3, root_3)), \ldots)$$

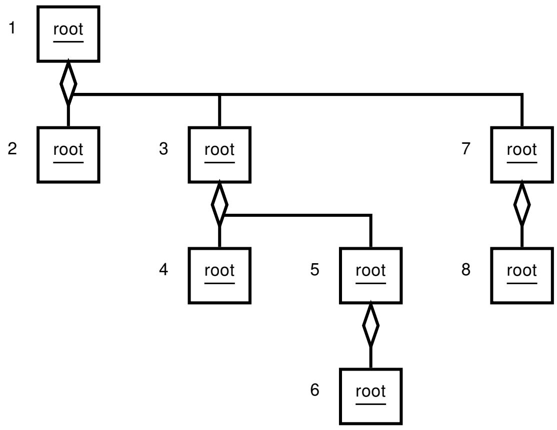


Abbildung 3: Beispiel: Ordnung der root-Elemente

$$
\begin{aligned}
root_1 &= root(\ldots, Obj_1, ((11, root_{11})), \ldots) \\
root_{11} &= root(\ldots, Obj_{11}, ((111, root_{111})), \ldots) \\
root_{111} &= root(\ldots, Obj_{111}, (), \ldots) \\
\\
root_2 &= root(\ldots, Obj_2, ((21, root_{21})), \ldots) \\
root_{21} &= root(\ldots, Obj_{21}, (), \ldots) \\
\\
root_3 &= root(\ldots, Obj_3, ((31, root_{31})), \ldots) \\
root_{31} &= root(\ldots, Obj_{31}, (), \ldots)
\end{aligned}
$$

Das root-Element $root_0$ ist das Wurzel-root-Element. Im Folgenden ist für einige Fib-Objekte angegeben, welche Identifier und damit root-Objekte sie in externen Objekten verwenden können bzw. welche root-Elemente aufgelöst werden können bzw. sichtbar sind. (Es wird hier keine Fib-Datenbank berücksichtigt. Root-Objekte der Datenbank wären für alle Fib-Objekte sichtbar.) Die Identifier und root-Elemente sind in der Reihenfolge angegeben, wie sie bei der Suche durchlaufen werden.

- Obj_0: $(1, root_1)$, $(2, root_2)$, $(3, root_3)$
- Obj_1: $(11, root_{11})$, $(2, root_2)$, $(3, root_3)$
- Obj_{11}: $(111, root_{111})$, $(2, root_2)$, $(3, root_3)$
- Obj_2: $(21, root_{21})$, $(3, root_3)$

- Obj_{21}: $(3, root_3)$
- Obj_3: $(31, root_{31})$
- Obj_{31}: keine

13 Die Fib-Datenbank

Die Fib-Datenbank gehört zu keinem Fib-Multimediaobject. Sie wird mit den Fib-Bibliotheken /dem Fib-System ausgeliefert und enthält häufig verwendete Datenbankobjekte, die in Fib-Objekten verwendet werden können. Fib-Objekte können die Datenbankobjekte verwenden, ohne diese beinhalten zu müssen. Datenbankobjekte können beispielsweise Linien (mit den Eingabeparametern für Start- und Endpunkte), Rechtecke oder Kreise sein, aber auch Bäume, Autos, Zeichensätze (fonts) oder Fraktale. Wird in einem Fib-Objekt beispielsweise ein Kreis benötigt, kann dass entsprechende parametrisierte Datenbankobjekt verwendet werden.

Eine detailierte Beschreibung der Fib-Datenbank kann in Teil X auf Seite 449 gefunden werden.

Die Implementierung der Datenbankobjekte kann an die jeweilige Anwendungsumgebung angepasst sein. Soll zur Anzeige beispielsweise OpenGL verwendet werden, können Datenbankobjekte direkt mit OpenGL-Primitiven (z. B. Dreiecken) umgesetzt werden. Auf diese Weise kann mit Datenbankobjekten die Performance der Anwendung verbessert werden. Bei der Kodierung der Fib-Objekte kann auch gleich darauf geachtet werden, dass Datenbankobjekte mit guter Performance für die Zielanwendung verwendet werden. Diese Fib-Objekte sind dann immer noch auf allen Fib-Systemen mit genügend hoher Datenbankversion anzeigbar, auf einigen jedoch schneller.

Welche Objekte eine Datenbank enthält, sowie die Identifier für diese Objekte, sind mit der Datenbankversion festgelegt. Datenbanken mit höheren Datenbankversionen enthalten dabei alle Datenbankobjekte mit den gleichen Identifiern wie Datenbanken mit niedrigeren Datenbankversionen. Auf diese Weise ist gewährleistet, dass Fib-Objekte immer aufwärtskompatibel zu neueren Datenbankversionen sind.

Alle Identifier von Datenbankobjekten sind negativ.

14 Definitionen für Fib

In diesem Abschnitt werden einige Definitionen für die Fib-Multimediasprache gegeben. Diese sollen den Umgang mit und das Verständnis für Fib erleichtern.

14.1 Definition des korrekten Fib-Objekt

Ein **Fib-Objekt** ist korrekt, wenn es der oben aufgeführten Fib-Syntax entspricht, alle Variablen, die es enthält, über es definiert sind und jedes enthaltende Fib-

Element in ihm ein korrektes Fib-Element ist. Zu einem korrekten Fib-Objekt muss ein korrektes root-Element gehören. Damit ein Fib-Element korrekt ist, muss es zu seinem root-Element passen (das heißt unter Anderem korrekte [Anzahl von] Dimensionen und Definitionsbereiche).

Korrekte Fib-Objekte werden auch kurz nur als Fib-Objekte bezeichnet.

Ein **Fib-Element** entspricht der oben vorgestellten Fib-Syntax, außer dass keine Fib-Elemente in ihm enthalten sind bzw. betrachtet werden.

In den nachfolgenden Beispielen werden Kommentare mit “//” eingeleitet. Diese gehören nicht mehr zu den dargestellten Fib-Elementen.

Beispiele:
Korrekte Fib-Objekte (Es wird angenommen, dass das zugehörige root-Element korrekt und passend ist.):

$$\begin{aligned}
Obj &= pr((3)_{colorGrayscale}, p((1;5))) \\
Obj &= list(for(y, [(4;2)], pr((3,42,125)_{colorRGB}, p((7;y))), \\
&\quad fun(x, add(mult(4, exp(3,-2)), 2), pr((205,x,x)_{colorRGB}, p((3;x))))
\end{aligned}$$

Fib-Element (Elm):

$$\begin{aligned}
Elm &= p((3;2;5)) \text{//3-dimensional} \\
Elm &= for(x, [(3;8)], null) \\
&\quad \text{//null ist kein Objekt (bei der Implementierung ein Nullzeiger)} \\
Elm &= p((2;5))
\end{aligned}$$

Weder ein Fib-Element noch ein korrektes Fib-Objekt (Woe ; $null$ ist kein Objekt [bei der Implementierung Nullzeiger]) :

$$\begin{aligned}
Woe &= list(for(x, [(3;7)], null), null) \\
Woe &= fun(x, add(4, exp(6,3)), for(y, [(6;7)], null)) \\
Woe &= pr((3)_{colorRGB}, p((1;5))) \text{ : //ungültige Syntax des} \\
&\quad \text{Eigenschaftselements: Der colorRGB-Vektor verlangt} \\
&\quad \text{nach 3 Parametern.}
\end{aligned}$$

14.2 Definition des vollständigen Fib-Objekt

Ein vollständiges Fib-Objekt ist ein Fib-Objekt, welches ein darstellbares Multimediaobjekt repräsentiert. Jedes vollständige Fib-Objekt ist auch ein korrektes Fib-Objekt. Zum vollständigen Fib-Objekt gehören unter Anderem alle notwendigen root-Elemente mit den Angaben der richtigen Fib- und Datenbankversion.

14.3 Definition von "unterhalb" und "oberhalb" in einem Fib-Objekt

Man stelle sich ein Fib-Objekt als Baum vor, in dem die Root und Listenelemente die Verzweigungen darstellen. Da in der Informatik im Allgemeinen die Wurzel oben dargestellt wird, sind die Elemente, welche das Element Elm enthält, unten und die Elemente, welche das Element Elm enthalten, für dieses oben.

Unterhalb eines Elements Elm bedeutet also, dass die Elemente gemeint sind, die das Element Elm direkt oder indirekt enthalten. Demgegenüber sind oberhalb eines Elements Elm, die Elemente, die das Element Elm direkt oder indirekt, enthalten.

Dieser Sachverhalt ist in Abbildung 4 dargestellt. Das Listenelement in der Mitte, das mit 1 gekennzeichnet ist, ist das Element bezüglich dessen oberhalb und unterhalb bestimmt wird.

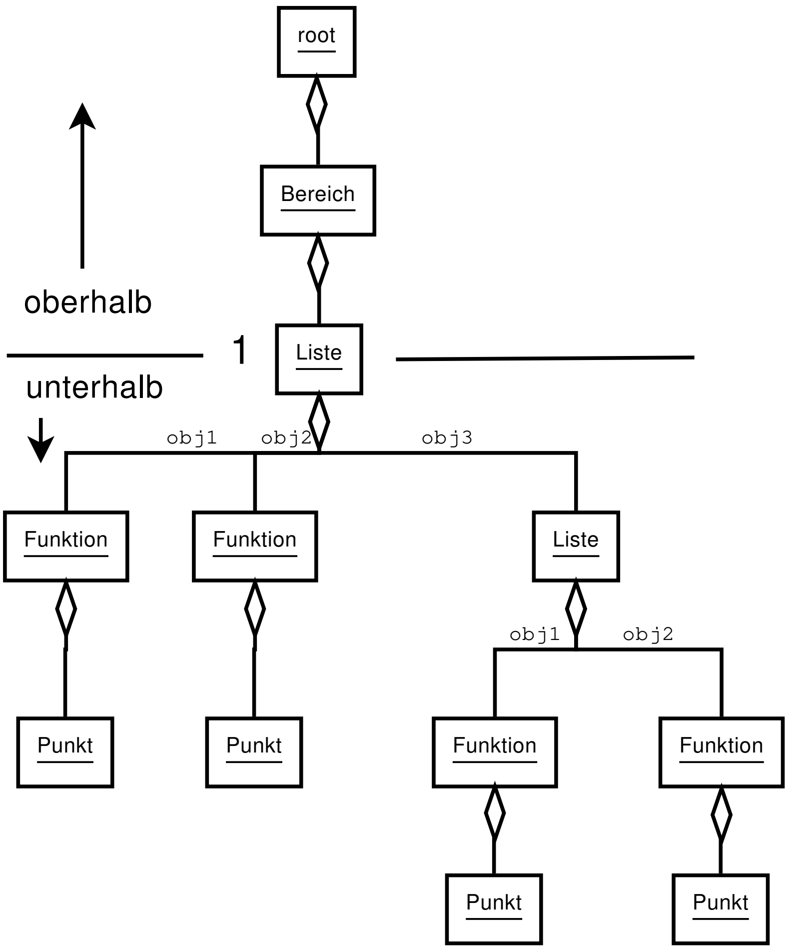


Abbildung 4: Beispiel für oberhalb und unterhalb in einem Fib-Objekt

14.4 Ordnung der Fib-Elemente

Auf den Definitionen von “unterhalb” und “oberhalb” in einem Fib-Objekt wird die Ordnung der Fib-Elemente aufgebaut. Jedem Fib-Element im vollständige Fib-Objekt wird eine eindeutige natürliche Zahl zugeordnet. Wenn in einem Fib-Objekt N Elemente existieren, werden den Fib-Elementen im Fib-Objekt die Zahlen 1 bis N zugeordnet. Fib-Elementen unterhalb eines Fib-Elements werden höhere Werte zugeordnet.

Bei Listenelementen haben Unterobjekte Obj_U des Listenelements mit einer höheren Nummer U als ein anderes Unterobjekt Obj_K ($U > K$) auch höhere Nummern für ihre enthaltenden Elemente, als die Elemente in Obj_K. Die root-Elemente werden in der Ordnung genauso behandelt wie Listenelemente, wobei das Haupt-Fib-Objekt das erste Unterobjekt ist, dem die Unter-root-Objekte in ihrer Reihenfolge (im root-Element) folgen.

Das selbe gilt für alle anderen Zweigelemente. Die oben vorgestellte Syntax gibt dabei die Reihenfolge der Unterobjekte vor.

In Abbildung 5 ist eine Beispielobjekt mit den zugehörigen Zahlen (links neben den Elementen) der Ordnung der Fib-Element dargestellt.

14.5 Ordnung bestimmter Fib-Elemente

Auch Fib-Elemente eines bestimmten Typs sind natürliche Zahlen einer Ordnung zugeordnet. Diese Ordnungen orientieren sich an der Ordnung der Fib-Elemente. Wenn in einem korrekten Fib-Objekt N Fib-Elemente eines Typs existieren, so sind den Fib-Elementen dieses Typs die Zahlen 1 bis N zugeordnet. Ist ein Fib-Element Elm_1 in der Ordnung der Fib-Elemente ein höherer Wert zugeordnet (als einem anderen Fib-Element Elm_2 des gleichen Typs), so ist ihm (Elm_1) auch in der Ordnung der Fib-Elemente des gleichen Typs ein höherer Wert zugeordnet (als dem anderen Fib-Element Elm_2).

In Abbildung 6 ist eine Beispielobjekt mit den zugehörigen Zahlen (links neben den Elementen) der Ordnungen der einzelnen Fib-Element dargestellt.

14.6 Ordnung der Verschiebungspunkte

Eine weitere Ordnung betrifft alle Fib-Elemente, die verschoben werden können. Diese werden Verschiebungspunkte genannt. Die Ordnungen der Verschiebepunkte orientieren sich an der Ordnung der Fib-Elemente. Wenn in einem vollständigen Fib-Objekt N Verschiebepunkte existieren, so sind den Verschiebepunkte, die Zahlen 1 bis N zugeordnet. Ist ein Fib-Element in der Ordnung der Fib-Elemente ein höherer Wert zugeordnet, so ist ihm auch in der Ordnung der Verschiebepunkte ein höherer Wert zugeordnet.

Fib-Elemente, welche verschoben werden können und Verschiebepunkte darstellen, sind alle Zweigelemente (limb Elemente; sie enthalten genau ein Unterobjekt):

- Eigenschaftselement (siehe Abschnitt 12.3 auf Seite 25)

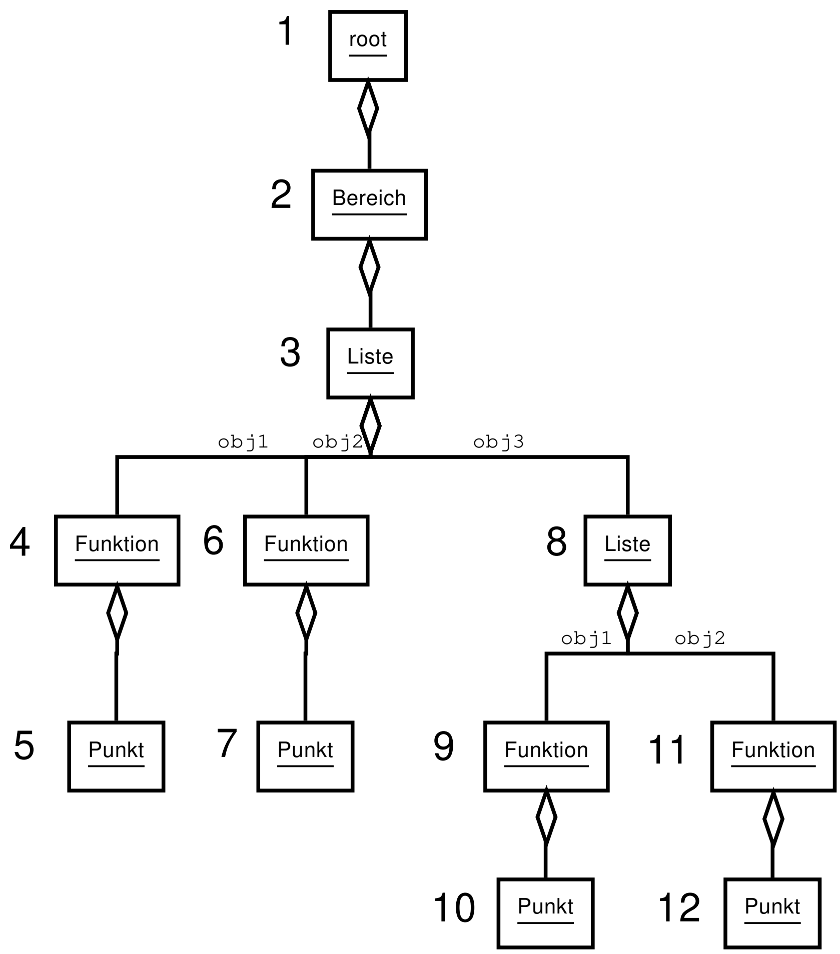


Abbildung 5: Beispiel für die Ordnung der Fib-Elemente

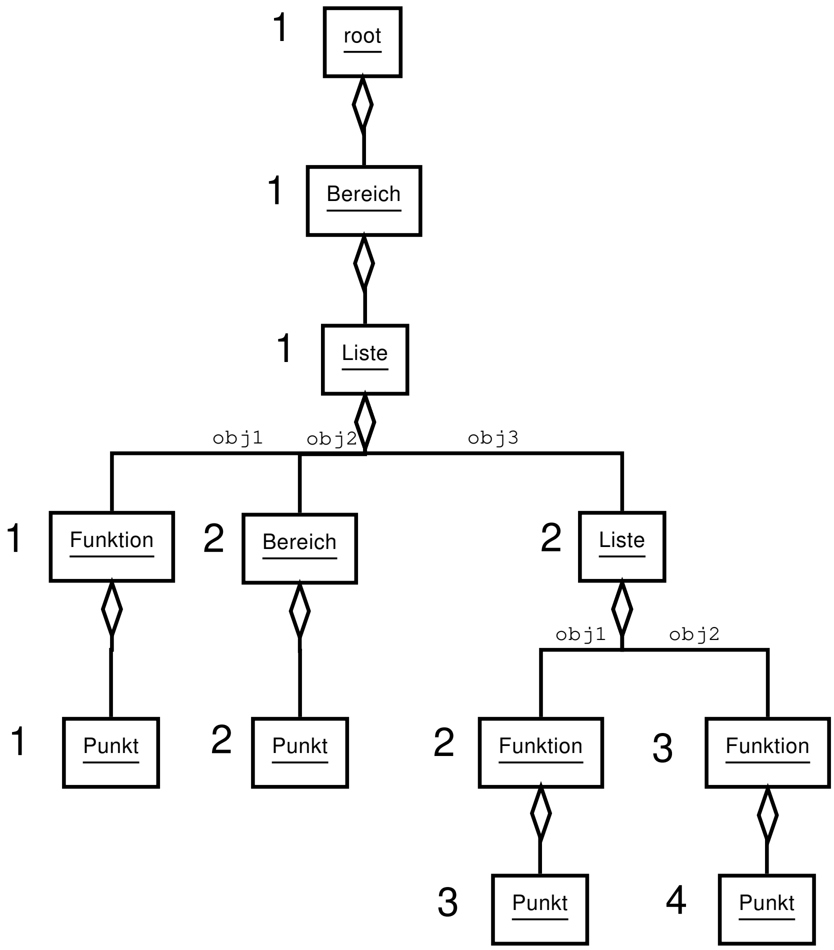


Abbildung 6: Beispiel für die Ordnung bestimmter Fib-Elemente

- Anmerkungselemente (siehe Abschnitt 12.5 auf Seite 32)
- Bereichselemente (siehe Abschnitt 12.6 auf Seite 34)
- Funktionselemente (siehe Abschnitt 12.7 auf Seite 35)
- Fib-Element, um Definitionsbereichseigenschaften abzurufen (siehe Abschnitt 12.11 auf Seite 46)
- Set-Element (siehe Abschnitt 12.12 auf Seite 48)
- Matrixelement (siehe Abschnitt 12.13 auf Seite 49)

Fib-Elemente, welche nicht verschoben werden können und damit keine Verschiebepunkte darstellen, sind:

- Alle Blattelemente:
  - Punkte (siehe Abschnitt 12.2 auf Seite 24)
  - Fib-Elemente für externe Unterobjekte (siehe Abschnitt 12.10 auf Seite 44)
- Alle Verzweigungselemente:
  - Root-Element (siehe Abschnitt 12.14 auf Seite 52)
  - Listenelement (siehe Abschnitt 12.4 auf Seite 31)
  - Das Fib-Element, um externe Objekte aufzurufen (siehe Abschnitt 12.9 auf Seite 43)
  - Bedingungen mit dem if-Element (siehe Abschnitt 12.8 auf Seite 41)

In Abbildung 7 ist ein Beispielobjekt mit den zugehörigen Zahlen (links neben den Elementen) der Ordnung der Verschiebepunkte dargestellt.

14.7 Definition Teilobjekt

Jedes Objekt, das ein ganzer Zweig (z. B. Unterlistenobjekt, Haupt-Fib-Objekt oder Unter-root-Objekt) eines Verzweigungselements ist, ist Teil eines Teilobjekts. Weiterhin gehören zum Teilobjekt alle root-Elemente, in denen es enthalten ist oder die es über externe Objekte benutzt. Auch Fib-Elemente, die Variablen definieren, welche in dem Teilobjekt verwendet werden, gehören dazu. Die Vereinigung zweier Teilobjekte ist wieder ein Teilobjekt. Das ganze Fib-Objekt selbst ist auch ein Teilobjekt. Ein Teilobjekt kann immer zu einem Multimediaobjekt ausgewertet werden.

Ein **echtes Teilobjekt** ist ein Teilobjekt, das nicht das (ganze) Fib-Objekt selbst ist.

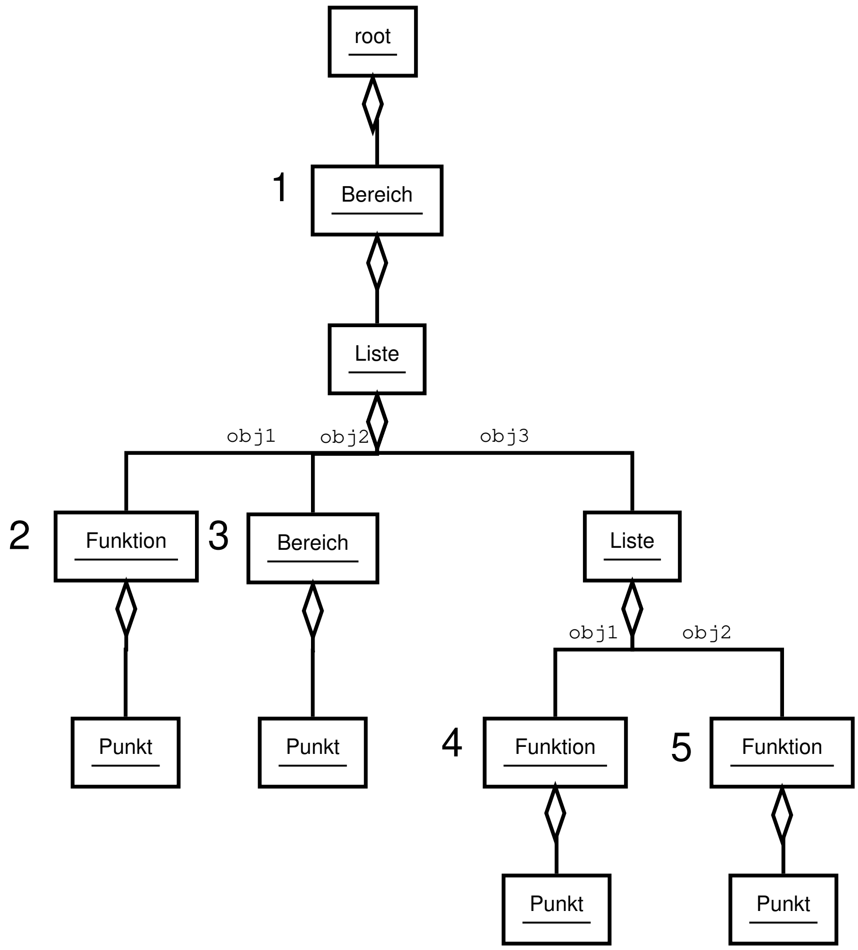


Abbildung 7: Beispiel für die Ordnung der Verschiebungspunkte

Ein **einfaches Teilobjekt** ist ein echtes Teilobjekt, das nur ein Blatt mit einem Punktobjekt enthält.

Ein **zusammenhängendes Teilobjekt** ist ein echtes Teilobjekt, welches das gesamte Objekt eines Arms, genau eines Verzweigungselements (z. B. Listenelements oder root-Elements) enthält und die benötigten Elemente über diesem. Insbesondere ist jedes einfache Teilobjekt ein zusammenhängendes Teilobjekt.

Um ein zusammenhängendes Teilobjekt zu erzeugen, kann aus dem ganzen Fib-Objekt ein Verzweigungselement (z. B. Listenelement oder root-Element) gelöscht und durch ein in ihm enthaltendes Unterobjekt ersetzt werden. Das erzeugte Teilobjekt muss natürlich korrekt sein, um ein zusammenhängendes Teilobjekt zu sein. (Es darf beispielsweise bei einem root-Element nicht das Haupt-Fib-Objekt gelöscht werden, und wenn ein root-Element gelöscht wird, und das nächste Haupt-Fib-Objekt oberhalb durch sein Haupt-Fib-Objekt ersetzt wird, müssen die Definitionsbereiche des ersetzten root-Elements vom nächsten enthaltenden root-Element übernommen werden.)

Mit Abbildung 8 sei diese Definition an einem Beispiel eines Fib-Objekts dargestellt. Im Nachfolgenden werden einige Beispiele für unterschiedliche Typen von Teilobjekte in Abbildung 8 gegeben, wobei für die Fib-Elemente ihre Nummern (aus der Fib-Elementordnung bzw. der Abbildung 8) angegeben werden. Weiterhin wird festgelegt, dass das Punktelement mit der Nummer 10 nicht die Variable benutzt, welche das Bereichselement mit der Nummer 2 definiert. Alle anderen Variablen werden in den Punkten benötigt, welche unter der jeweiligen Variablendefinition stehen.

Teilobjekte:

- 1; 2; 4; 5
- 1; 2; 6; 7
- 1; 2; 3 (nur Unterobjekt 1 und 2); 4; 5; 6; 7
- 1; 9; 10
- 1; 2; 8; 9; 10; 11; 12
- 1; 2; 3 (nur Unterobjekt 1 und 3); 4; 5; 8; 9; 10; 11; 12
- 1; 2; 3 (nur Unterobjekt 1 und 3); 4; 5; 11; 12
- (alle Fib-Elemente) 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12

Echte Teilobjekte:

- 1; 2; 4; 5
- 1; 2; 6; 7

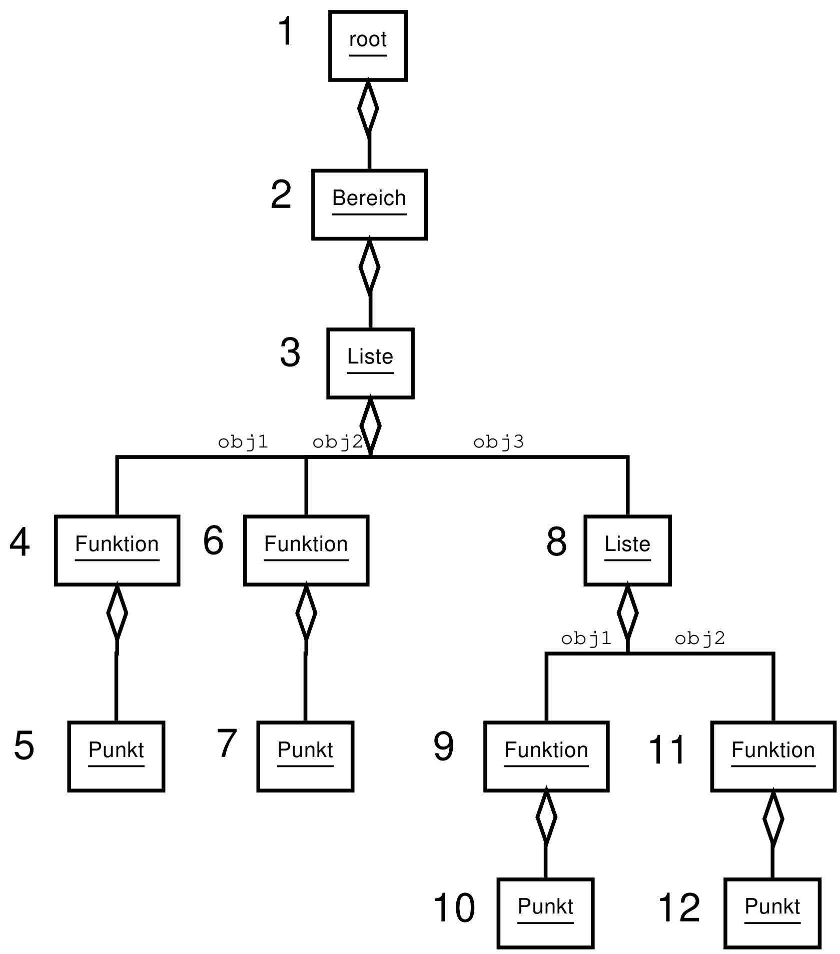


Abbildung 8: Beispiel Objekt für Teilobjekte

- 1; 2; 3 (nur Unterobjekt 1 und 2); 4; 5; 6; 7
- 1; 9; 10
- 1; 2; 8; 9; 10; 11; 12
- 1; 2; 3 (nur Unterobjekt 1 und 3); 4; 5; 8; 9; 10; 11; 12
- 1; 2; 3 (nur Unterobjekt 1 und 3); 4; 5; 11; 12

Einfache Teilobjekte:

- 1; 2; 4; 5
- 1; 2; 6; 7
- 1; 9; 10
- 1; 2; 11; 12

Zusammenhängende Teilobjekte:

- 1; 2; 4; 5
- 1; 2; 6; 7
- 1; 9; 10
- 1; 2; 11; 12
- 1; 2; 8; 9; 10; 11; 12

14.8 Die Ordnung der zusammenhängenden Teilobjekte

Auch auf den zusammenhängenden Teilobjekten gibt es eine Ordnung. Diese wird im Nachfolgenden auch Ordnung der Teilobjekte gennant, da es für die anderen Arten von Teilobjekten keine eigene Ordnung gibt.

Die Ordnungen der (zusammenhängenden) Teilobjekte orientieren sich an der Ordnung der Fib-Elemente. Wenn in einem vollständigen Fib-Objekt N zusammenhängende Teilobjekte existieren, so sind den zusammenhängenden Teilobjekten die Zahlen 1 bis N zugeordnet. Je größer die Nummer eines zusammenhängenden Teilobjekts ist, desto größer ist auch die kleinste Nummer in Ordnung der Fib-Elemente, der Fib-Elemente in ihm. Oder auch: Ist dem obersten Fib-Element des Unterobjekts, über das ein zusammenhängendes Teilobjekt definiert ist, ein höherer Wert zugeordnet, so ist auch dem definierten Teilobjekt ein höherer Wert zugeordnet.

In Abbildung 9 ist ein Beispielobjekt mit den zugehörigen Zahlen (links neben den definierenden Unterobjekten der Teilobjekte) in der Ordnung der Teilobjekte dargestellt.

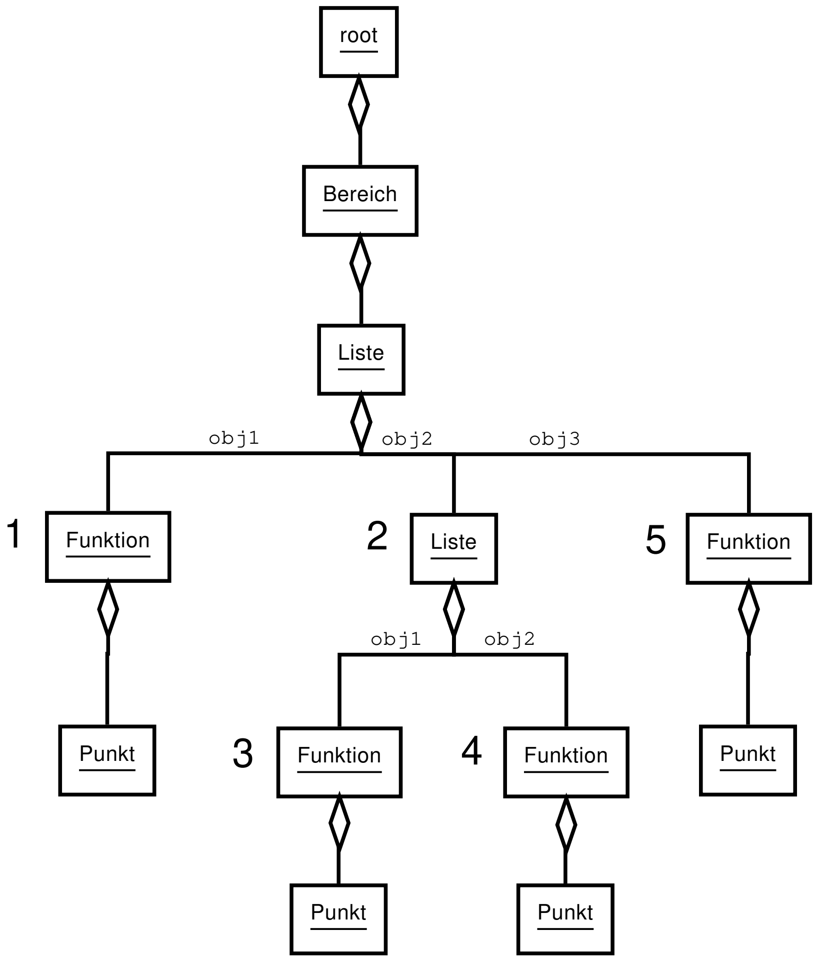


Abbildung 9: Beispiel für die Ordnung der zusammenhängenden Teilobjekte

14.9 Definition Fib-Multimediaobjekt

Wird der Ausdruck Fib-Multimediaobjekt benutzt, ist damit ein Fib-Objekt mit Schwerpunkt auf dem Multimediaobjekt, das es repräsentiert, gemeint.

14.10 Definition richtiges Fib-Multimediaobjekt

Ein richtiges Fib-Multimediaobjekt ist ein Fib-Objekt, welches das originale Multimediaobjekt, das es repräsentieren soll, vollständig wiedergibt. Wenn also das Multimediaobjekt, welches das Fib-Objekt repräsentiert, und das originale Multimediaobjekt verglichen werden, kann kein Unterschied zwischen ihnen feststellt werden.

15 Theoretische Aussagen zur Fib-Multimediabeschreibungssprache

Es folgen einige theoretische Aussagen zur Fib-Multimediabeschreibungssprache, für die aber aus Zeitgründen meistens nicht ein vollständiger Beweis angeführt wird. Mit diesen Aussagen soll ein besseres Verständnis der Fib-Multimediasprache erreicht werden.

15.1 Mächtigkeit der Fib-Multimediasprache auf Bilder

Mit der Fib-Multimediasprache können alle als Rastergrafik darstellbaren Bilder dargestellt werden. Bilder, die als Rastergrafik darstellbar sind, sind die am häufigsten verwendeten Bilder in der elektronischen Datenverarbeitung. Dazu gehören unter anderem Windows Bitmap (BMP, Dateiendung: .bmp), JPEG File Interchange Format (JFIF, Dateiendung: .jpg) und Portable Network Graphics (PNG, Dateiendung: .png).

Allein mit dem Punktelement und dem Listenelement lassen sich bereits alle möglichen Rastergrafiken darstellen.

Beweis: Eine Rastergrafik (euklidisch, zweidimensional, diskret) kann als Matrix dargestellt werden, in der die Spalte die x-Koordinate, die Zeile die y-Koordinate des Punktes angibt und die Werte die Farben der Punkte angeben. Die Anzahl der Punkte in der Rastergrafik ist endlich. Um diese Punkte durch Fib-Multimediasprache darzustellen, ist ein root-Element zu erzeugen, das die Eigenschaften der Rastergrafik enthält (Größe bzw. Dimensionsdefinitionsbereich usw.). In diesem root-Element ist als Haupt-Fib-Objekt ein Listenelement einzufügen, welches für jeden Punkt des Bildes ein Unterobjekt enthält. Dieses Unterobjekt besteht nur aus einem Eigenschaftselement, das die Farbe des Punktes kodiert, und einem enthaltenden Punktelement, welches die Position des Punktes kodiert. So gibt es für jeden Punkt in der Rastergrafik einen entsprechenden Punkt im erzeugten Fib-Objekt, womit das Fib-Objekt die Rastergrafik repräsentiert.

Eine Abbildung in die Fib-Multimediasprache kann beispielsweise mit dem im Listing 3 dargestellten Algorithmus geschehen (Pseudo-C).

Dabei beginnt die Indizierung der Matrix mit (0,0).
Der Farbwert der Koordinate (x,y) kann mit `matrix[ x, y]` bestimmt werden. Mit der Funktion `getColorVector()` wird aus dem Matrix-Farbwert ein Fib-Farbvektor generiert.
Die Syntax der Fib-Objekte entspricht der im Abschnitt III auf Seite 22 angesprochenen möglichen Syntax.

Listing 3: Algorithmus zur Erzeugung eines korrekten Fib-Objekts aus einer Bildmatrix

```
void translate( matrix ){

   int xmax = Anzahl der Spalten der Matrix;
   int ymax = Anzahl der Zeilen der Matrix;
   Fib_Objekt_Pointer obj=new root();

   //Eigenschaften der Rastergrafik setzen
   obj->setBild( xmax, ymax, Farbschema);

   list hauptliste=new list()

   obj->insertMainObject( hauptliste )

   for( int x = 0; x == xmax; x = x + 1 ){

      for( int y = 0; y == xmax; y = y + 1 ){
         hauptliste->insertObject( property( getColorVector(
             matrix[x,y] ) , p( (x,y) ) ) );
      };
   };
};
```

Da die Erweiterung mit Funktionen und anderen Elementen optional ist, wird mit dem angegebenen Algorithmus ein korrektes Fib-Objekt erzeugt.

Zu jedem Punkt in der Matrix gibt es einen gleichfarbigen Punkt im Fib-Objekt mit der entsprechenden Koordinate, aber es gibt keinen Punkt im Fib-Objekt, der nicht in der Matrix vorhanden ist, denn jede Koordinate der Matrix wird mit den zwei for-Schleifen durchlaufen. Dadurch wird für jede Koordinate und damit auch für jeden Punkt in der Rastergrafik ein Punkt im Fib-Objekt eingefügt. Somit gibt es für jeden Punkt in der Rastergrafik einen entsprechenden Punkt in dem Fib-Objekt. Da nur Koordinaten der Matrix in den for-Schleifen durchlaufen werden und nur für diese die entsprechenden Punkte in das Fib-Objekt eingefügt werden, existieren nur Punkte die auch in der Matrix auftauchen und damit auch nur Punkte im Fib-Objekt die auch in der Rastergrafik vorkommen. Es gibt also nur die Punkte aus der Originalrastergrafik im Fib-Objekt und nur diese. Damit repräsentieren das Fib-Objekt und die Originalrastergrafik die gleiche Rastergrafik. Deshalb können alle Rastergrafiken auch mit der Fib-Multimediasprache dargestellt werden.

Ein mit dem Algorithmus generiertes Fib-Objekt, das einer Originalrastergrafik entspricht, stellt eine obere Grenze der minimalen Größe möglicher entsprechender Fib-Objekte da. Das heißt, jede Rastergrafik kann durch ein Fib-Objekt repräsentiert werden, das genauso groß ist, wie ein Fib-Objekt für die Rastergrafik, das mit dem oben aufgeführten Algorithmus generiert wurde, nämlich das generierte Fib-Objekt. Es gibt aber wahrscheinlich noch kürzere Fib-Objekte für die Rastergrafik.

Damit ist die minimale Größe eines Fib-Objekts für eine Rastergrafik maximal:
Fib_max_{min} = (Anzahl der Pixel im Bild) $*$[(Größe eines Punktelements) + (Größe eines Eigenschaftselements für das entsprechende Farbschema)] + (Größe eines Listenelements)+ (Größe eines root-Elements mit gesetzten Werten)

Beispiel: Dargestellt werden soll ein Bild 8 x 8 = 256 Pixel mit 3 Byte für die Farbe (RGB), 1 Byte für die Position (für jede Richtung x und y 4 Bit = 8 mögliche Werte) und 1 Byte für den Objektnamen (z. B. “l” für Listelemente und “p” für Punktelemente). Klammern werden nicht benötigt, da alle Teile eine feste Länge haben. (Die Annahmen über die Größen der Fib-Objekt-Elemente wird hier abgeschätzt. Bei einer Implementierung sind sicherlich bessere (/kleinere) Werte möglich.)

- Für einen Punktelement werden 2 Byte benötigt (1 Byte Position + 1 Byte Elementname).
- Für eine Eigenschaftselement werden 4 Byte benötigt (3 Byte Farbe + 1 Byte Elementname).
- Für ein Listenelement werden 9 Byte benötigt (8 Byte für Angabe der Anzahl der Unterobjekte + 1 Byte Elementname).
- Für das root-Element werden 256 Bytes benötigt. (Da viele Teile des root-Elements leer sind und nur der Definitionsbereichen für 2 Dimensionen und die RGB-Farbdefinitionsbereiche angegeben werden müssen, sollten 256 Bytes genügen.)

Rechnung:
$256(Pixel) * (2Bytes + 4Bytes) + 9Bytes + 256Bytes = 1801Bytes$

Das Bild kann also auf jeden Fall mit einem 1801 Byte großen Fib-Objekt dargestellt werden. Es sind davon abweichende kürzere Fib-Objekt-Darstellungen möglich. Die 1801 Byte sind also die Obergrenze für die minimale Größe, mit der das Bild mit einem Fib-Objekt dargestellt werden kann.

Zum Vergleich: Der Speicherbedarf einer Rastergrafik als Bitmap (nur Punkt-Informationen) beträgt mindestens (Anzahl der Pixel im Bild) * (Größe eines Farbwertes)

Für das obige Beispiel ergibt sich: $256(Pixel) * 3Bytes = 768Bytes$

Das ist ungefähr die Hälfte der 1801 Byte der Obergrenze für die minimale Fib-Objekt-Darstellung.

Indem die Matrix auf 3 Dimensionen erweitert wird, kann die Aussage über die Darstellbarkeit aller Rastergrafiken leicht auf Sequenzen von Rastergrafiken (z. B. die Bilder von Filmen) erweitert werden.

15.2 Mächtigkeit von Fib

Satz: Die Menge der möglichen Fib-Objekte ist abzählbar unendlich.

Beweis Skizze für die Abzählbarkeit:
Jedes Fib-Objekt kann mit einer endlichen Anzahl von Buchstaben und damit auch Bits oder Zahlen repräsentiert werden, und die Menge dieser ist abzählbar. Das folgt daraus, dass die Anzahl der Fib-Elemente in einem Fib-Objekt immer abzählbar ist und jedes Fib-Element aus abzählbar vielen Teilen besteht, welche selbst abzählbar sind, es gibt z. B. nur ganze oder rationale Zahlen, und auch die Menge der Variablen ist abzählbar.

Beweis Skizze für "unendlich":
Es können alle natürlichen Zahlen durch Fib-Objekte repräsentiert werden. Im Nachfolgendem ist eine mögliche Darstellungsart von beliebigen natürlichen Zahlen mithilfe von Fib-Objekten beschrieben.

Ein Punktobjekt alleine stellt die natürliche Zahl 0 da. Wird ein Fib-Objekt in ein neues Funktionselement eingesetzt, stellt das entstehende Fib-Objekt den Nachfolger des ursprünglichen Fib-Objekts da. Damit werden die 0 und die Nachfolgefunktion in der Fib-Multimediasprache nachgebildet, und es können damit alle natürlichen Zahlen dargestellt werden. Da die Menge der natürlichen Zahlen unendlich ist, muss die Menge der Fib-Objekte auch unendlich sein.

Jedes Multimediaobjekt wird sogar durch eine abzählbar unendliche Menge von Fib-Objekten repräsentiert, denn es kann an ein Fib-Objekt, das ein Multimediaobjekt repräsentiert, jedes beliebige Fib-Objekt mit einem Listenelement angehängt werden, solange dieses die Multimediaobjektdarstellung nicht verändert. Es könnte z. B. beliebig oft an ein Fib-Objekt eine Kopie von sich selbst mit Hilfe eines Listenelements angefügt werden, ohne das Multimediaobjekt zu verändern.

15.3 Jedes vollständige Fib-Objekt kann als ein Multimediaobjekt dargestellt werden

Es wird in diesem Abschnitt aufgezeigt, dass es möglich ist, jedes vollständige Fib-Objekt (siehe Abschnitt 14.2 auf Seite 73) immer so zu interpretieren (in ein Multimediaobjekt zu übersetzen), dass nur gültige Multimediaobjekte eines bestimmten Typs (z. B. RGB-Bilder mit 100 x 100 Pixel) entstehen können. Dabei wird die schon angesprochene Einschränkung bezüglich der Multimediadaten vorrausgesetzt, dass die Multimediadaten als Eigenschaften von Punkten eines endlichen, euklidischen, diskreten (es gibt kleinste Einheiten) Raumes darstellbar sind. Diese Einschränkung ist nicht sehr groß, da sie (fast) keine der heutzutage üblichen Multimediadaten ausgrenzt. Die Multimediadaten können also Bilder, Ton und Filme darstellen.

Mit der Einschränkung, dass der Abstand/Unterschied zwischen zwei Eigenschaften des gleichen Typs immer als Zahlenwert bestimmt werden kann, können zwei Multimediaobjekte mit den gleichen Dimensionen immer verglichen werden. Zwei Multimediaobjekte haben die gleichen Dimensionen, wenn für jeden Punkt in dem einem Multimediaobjekt genau ein entsprechender Punkt im jeweils anderen Multimediaobjekt existiert.

Die Voraussetzung des vollständigen Fib-Objekts ist nötig, um zu gewährleisten, dass das Fib-Objekt immer auswertbar ist. Da die Vollständigkeit eines Fib-Objekts über die in Teil III dargestellte Syntax geprüft werden kann, ist die Vollständigkeit eines Fib-Objekts immer feststellbar. Nicht vollständige Fib-Objekte sollten von Algorithmen bzw. den genetischen Operatoren nicht erzeugt werden.

Wenn nun bei einem Fib-Objekt die Dimensionen an die eines Multimediaobjekts angepasst werden (dies sollte immer möglich sein), ist das Fib-Objekt immer mit dem Multimediaobjekt vergleichbar, da es selbst immer als Multimediaobjekt dargestellt werden kann und zwei Multimediaobjekte mit den gleichen Dimensionen immer verglichen und die Ähnlichkeit zueinander bewertet werden kann.

Anmerkung: Dies ist für genetische Algorithmen vorteilhaft. Bei einigen anderen Darstellungsformen, die durch genetische Algorithmen erzeugt werden, können ungültige Objekte (z. B. Programme) entstehen, bei denen eine genauere(/r) Bewertung/Vergleich nicht möglich ist. Eine Population in dieser Darstellungsform kann dann, z. B. eine große Klasse von ungültigen Objekten haben, die alle gleich schlecht sind und damit bei der Selektion gleich berücksichtigt werden. Wenn dann die Population nur aus ungültigen Individuen besteht, ist die Auswahl eines besseren Individuums unmöglich. Der genetische Algorithmus befindet sich dann sozusagen auf einer (Fitness-)Ebene, von der er nur noch schwer herunterfinden kann.

Beweis, dass jedes korrekte Fib-Objekt als ein Multimediaobjekt (eines bestimmten Typs, z. B. als Bild) dargestellt werden kann: Ausgangspunkt ist, dass ein Multimediaobjekt (euklidisch, zweidimensional, diskret) als endliche Menge von Punkten mit ihren (endlich vielen) Eigenschaften dargestellt werden kann. Da es nur endlich viele Punkte mit nur endlich vielen Eigenschaften gibt, ist eine solche endliche Menge immer konstruierbar.

Eine solche endliche Menge von Punkten mit ihren (endlich vielen) Eigenschaften erzeugt auch jedes korrekte Fib-Objekt. Punkte die in dieser Menge zu viel sind, um ein Multimediaobjekt zu repräsentieren, werden aus der Menge gelöscht. Punkte, die in der Menge fehlen, um ein Multimediaobjekt zu repräsentieren, werden in die Menge eingefügt. Eigenschaften von Punkten die zu viele sind, um ein Multimediaobjekt zu repräsentieren, werden gelöscht. Eigenschaften, die bei Punkten fehlen, um ein Multimediaobjekt zu repräsentieren, werden hinzugefügt und mit Standardwerten belegt (z. B. den Nullwerten ihres Definitionsbereichs). Auf diese Weise entsteht eine endliche Menge von Punkten mit ihren (endlich vielen) Eigenschaften, die als Multimediaobjekt repräsentiert werden kann.

Beispiel: Das Fib-Objekt soll ein RGB-Bild mit 100 x 100 Pixel darstellen. Dann werden die Dimensionen des Fib-Objekts so angepasst, das diese 100 x 100 Pixel in horizontaler und vertikaler Richtung umfassen. Das heißt, wenn die Dimension (horizontal oder vertikal) schon existiert, wird der Definitionsbereich jeweils so angepasst, dass er mindestens 100 Werte in gleichmäßigen Abständen umfasst. So wird jedem Pixel ein Wert zugeordnet. Sollte eine Dimension fehlen, so wird sie mit dem entsprechenden Definitionsbereich erzeugt und in allen Punkten für die Dimension der Standardwert 0 eingesetzt. Es gibt dann keine Punkte mit anderen Werten als den Standardwert für diese Dimensionen. Dimensionen, die zuviel sind, werden aus den root-Elementen und den Punkten gelöscht. Bei der Auswertung des so erzeugten Fib-Objekts entsteht eine Menge von Punkten mit ihren Eigenschaften. Bei der Auswertung wird der kleinste Wert W_{min} der jeweiligen Dimension im Fib-Objekt dem Wert 0 als Koordinate in der Menge zugeordnet, dem zweitkleinsten die 1 als Koordinate usw.

Aus dieser Menge werden nun alle Punkte gelöscht, die nicht innerhalb der 100 x 100 Pixel liegen (also Punkte, bei denen ein Wert bzw. eine Koordinate kleiner 0 oder größer 99 ist). Für alle Koordinaten, die noch fehlen (also fehlende Punkte, bei denen die Werte bzw. die Koordinaten zwischen [inklusive] 0 und 99 liegen), werden Punkte eingefügt. Alle Eigenschaften, die nicht RGB-Farben entsprechen, werden gelöscht. Allen Punkten, denen keine Eigenschaft für RGB- Farben zugeordnet ist, wird die Standardfarbe $(0, 0, 0)_{colorRGB}$ zugeordnet. Die so entstandene Menge enthält für jeden Punkt in dem RGB-Bild mit 100 x 100 Pixel einen Punkt mit einer RGB-Farbe, aber keinen anderen Punkt oder Eigenschaften und stellt somit ein RGB-Bild mit 100 x 100 Pixel da.

Dies kann dann mit anderen RGB-Bildern mit 100 x 100 Pixel verglichen und im Bezug auf diese bewertet werden.

16 Weitere Annahmen zu Fib

Die hier aufgestellten Überlegungen sollen als Denkanstöße dienen. Sie erheben nicht den Anspruch, richtig zu sein, sondern sind vielmehr nur Vermutungen. Mit ihnen soll ein tieferes und/oder besseres Verständnis der Fib-Multimediasprache gefördert werden. Des Weiteren sollen Hinweise auf die Möglichkeiten der Fib-Multimediasprache gefunden werden.

16.1 Komprimierungsmöglichkeiten von Fib

Warum ist eine effizientere Kodierung mit Fib als beim "Pixelbild" möglich?
Mit Hilfe von Fib lassen sich zumindest einfache (strukturierte) Bilder mit Fib-Objekten kürzer darstellen.

Beispiel:
Ein RGB-Farben-Bild ([r,g,b]-Vektor) mit 1.000 Pixeln horizontal und vertikal, mit schwarzem Hintergrund und einem weißen Rechteck in der oberen linken Ecke, das bis zur Mitte des Bildes reicht (Siehe Abbildung 10)

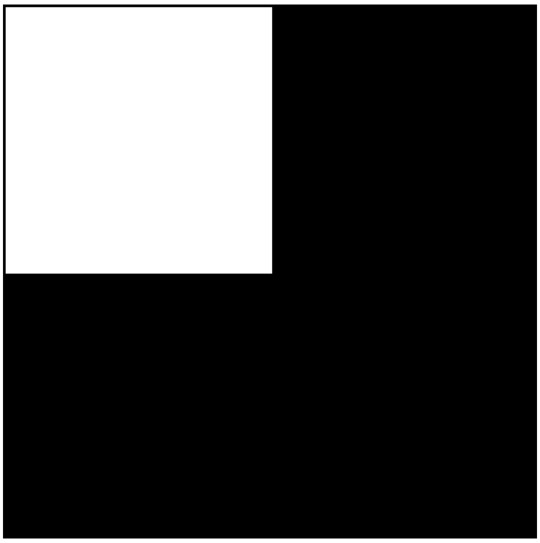


Abbildung 10: Beispielbild: 1000 x 1000 Pixel, schwarzer Hintergrund mit weißem Rechteck

Eine Fib-Darstellung:
root((1, 0) , , ((dim(2, 1, 2), vector(2, naturalNumber(999), naturalNumber(999))), (property(colorRGB), vector(3 , naturalNumber(255), naturalNumber(255) , naturalNumber(255)))), , , , list(for(y, [(0;999)], for(x, [(0;999)], pr($(0;0;0)_{colorRGB}$, p((x,y))))), for(x, [(5;500)], for(y, [(5;500)], pr($(255;255;255)_{colorRGB}$, p((x,y)))))) , ,)

Größenvergleiche einiger Formate:

- Fib: Länge dieses Fib-Objektes: 315 Bytes (Anzahl der Zeichen im String oberhalb, in der er in einer Textdatei gespeichert werden könnte)

- Bitmapbild (ohne Header): 3 Bytes pro Punkt für RGB * 1.000 * 1.000 = 3.000.000 Bytes
- JPG: 7857 Byte mit Gimp: Qualität 85 und optimiert; 7294 Byte mit Gimp: Qualität 85 und optimiert;

An diesem Beispiel ist ersichtlich, dass das Fib-Objekt eine wesentlich kürzere Darstellung ermöglicht. Zumindest für dieses Beispiel ist eine gute Komprimierung mit Fib möglich. Die Fib-Darstellung stellt eine Verkleinerung um den Faktor von rund 9500 zu Bitmapbildern und 25 zu JPG dar.

Beispiel höhere Auflösung:
Um das Verhalten von Fib-Multimediaobjekten bei einer Erhöhung der Auflösung zu veranschaulichen, sei hier das obige Beispiel mit doppelter Auflösung ausgeführt.
Grundlage ist also ein RGB-Farben-Bild ([r,g,b]-Vektor) mit 2.000 Pixeln horizontal und vertikal, mit schwarzem Hintergrund und einem weißen Rechteck in der oberen linken Ecke, dass bis zur Mitte des Bildes reicht (Siehe Abbildung 10).

Eine Fib-Darstellung: root((1, 0) , , ((dim(2, 1, 2), vector(2, naturalNumber(1999), naturalNumber(1999))), (property(colorRGB), vector(3 , naturalNumber(255), naturalNumber(255) , naturalNumber(255)))), , , , list(for(y, [(0;1999)], for(x, [(0;1999)], pr($(0;0;0)_{colorRGB}$, p((x,y))))), for(x, [(10;1000)], for(y, [(10;1000)], pr($(255;255;255)_{colorRGB}$, p((x,y)))))) , ,)

Größenvergleiche einiger Formate:

- Fib: Länge dieses Fib-Objektes: 321 Bytes (Anzahl der Zeichen im String oberhalb, in der er in einer Textdatei gespeichert werden könnte)
- Bitmapbild (ohne Header): 3 Bytes pro Punkt für RGB * 2.000 * 2.000 = 12.000.000 Bytes
- JPG: 29127 Byte mit Gimp: Qualität 85 und optimiert; 26942 Byte mit Gimp: Qualität 15 und optimiert;

Da die Komplexität des Bildes sich nicht erhöht hat, ist die Größe der Fib-Darstellung annähernd gleich geblieben. Es wurde nur etwas mehr Platz für die größeren Werte benötigt. Sowohl für die Bitmapbild- als auch für die JPG-Darstellung hat sich der Speicherbedarf ungefähr vervierfacht. Der Speicherbedarf stieg bei beiden also linear mit der Anzahl der Pixel. Dadurch konnte Fib seinen Vorsprung bei der Komprimierung weiter ausbauen. Die Fib-Darstellung stellt eine Verkleinerung um den Faktor von rund 37000 zu Bitmapbildern und 91 zu JPG dar. Wenn also die Komplexität von Bildern weniger als liniear mit deren Inhaltsgrößen (also der Anzahl der enthaltenden Punkte, d. h. nicht quadratisch mit deren Auflösung) steigt, werden die Komprimierungsvorteile für Fib, zumindest gegenüber Bitmap und JPG, mit zunehmender Größe der Bilder auch zunehmen.

Tauchen Strukturen auf Bildern mehrfach auf, erhöht dies die Komplexität des Bildes nicht so sehr wie völlig neue Strukturen. Mit Fib können mehrfach auftauchende ähnliche Strukturen abgedeckt werden, wahrscheinlich oft durch einen Strukturprototyp, der dann jeweils transformiert wird. Auf den meisten aufgenommenen Bildern ist es üblich, dass Strukturen in ähnlicher Weise häufiger auftauchen. Beispiele dafür sind Vierecke, Fenster eines Hauses oder auch Grashalme. Mit Vergrößerung der Aufnahme wird daher wahrscheinlich nicht die Anzahl der unterschiedlichen Strukturen, und damit die Komplexität für Fib, linear mit der Zunahme an Pixeln steigen, sondern es werden eher weniger neue Strukturen auftauchen, dafür aber werden ähnlich Strukturen öfter auftauchen. Dies gibt natürlich Fib einen Vorteil gegenüber Bitmap und JPG bei der Komprimierung größerer Bilder.

Eine Aussage im Allgemeinen über die Komprimierungsmöglichkeiten von Fib kann aber leider wegen der möglichen großen Komplexität von Fib-Objekten nicht getroffen werden. Es ist aber zu erwarten, dass die Komprimierungsmöglichkeiten mit Verringerung der Komplexität der Multimediaobjekte (z. B. Bilder), bzw. der Objekte auf diesen, zunimmt.

16.2 Überdeckung von Fib-Objekten

Definiert wird eine Überdeckungsrelation $<_{Fib}$ auf Fib-Objekten.

Dabei gilt:

- for(x,$[B_1, \ldots, B_{j-1}, B_j, \ldots, B_k, B_{k+1}, \ldots, B_n]$, Obj_1) $<_{Fib}$ list(for(x,$[B_1, \ldots, B_{j-1}, B_{k+1}, \ldots, B_n]$, Obj_1), for(x,$[B_j, \ldots, B_k]$, Obj_1))

- for(x,$[B_1, \ldots, B_j, (a;b), B_{j+2}, \ldots, B_n]$, Obj_1) $<_{Fib}$ list(for(x,$[B_1, \ldots, B_j, (a;c), B_{j+2}, \ldots, B_n]$, Obj_1), for(x,$[(e;b)]$, Obj_1))
  Dabei entspricht der Bereich (a;b) der Wertemenge $[a, \ldots, c, e, \ldots, b]$.

- for(x,$[(a;b)]$, Obj_1) $<_{Fib}$ list(Obj_1', Obj_1''), dabei entspricht der Bereich $(a;b)$ der Wertemenge $[a,b]$ und im Obj_1' wurden alle Vorkommen von x durch a ersetzt und in Obj_1'' alle Vorkommen von x durch b.

- Wenn zwei Objekte (Obj_1 und Obj_2) je ein Objekt enthalten (Obj_e_1 ist enthalten in Obj_1 und Obj_e_2 ist enthalten in Obj_2), welche die Relation erfüllen (Beispiel 1: $Obj\_e_1 <_{Fib} Obj\_e_2$) und der Rest dieser Objekte (Obj_1, Obj_2) ohne die enthaltenden Objekte (Obj_e_1, Obj_e_2) identisch sind, dann erfüllen auch die beiden Objekte die Relation in der gleichen Richtung (für das Beispiel 1: $Obj_1 <_{Fib} Obj_2$).

Zwei Objekte, welche die Relation $<_{Fib}$ erfüllen, realisieren das gleiche Multimediaobjekt.

Zu jedem Objekt (Obj) existiert eine Klasse von Objekten (Obj'), welche die Gleichung $Obj <_{Fib} Obj'$ erfüllen.

16.3 Struktur der Fib-Objekte im Hypothesenraum

Wenn nun angenommen wird, dass "natürliche" Multimediaobjekte (z. B. Bilder), die etwas darstellen sollen, nicht aus völlig unzusammenhängenden Eigenschaften auf den Punkten (z. B. Farbe auf Pixel) bestehen, sondern Zusammenhänge zwischen den Eigenschaften von (benachbarten) Punkten enthalten sind und diese mit entsprechenden Bereichs- und Funktionselementen dargestellt werden können, So gehören zu solchen Multimediaobjekten viele Fib-Objekte mit relativ vielen Bereichselementen, die eine noch größere Menge von Fib-Objekten überdecken, welche die $<_{Fib}$-Relation erfüllen und damit zu den entsprechenden Klassen gehören.

Die daraus gefolgerte Vermutung ist nun, dass im Hypothesenraum der Fib-Objekte die Fib-Objekte, die "natürlicheren" Multimediaobjekte entsprechen, größere Cluster (z. B. die Menge der bezüglich der $<_{Fib}$-Relation ordnerbaren Fib-Objekte) bilden, als die Fib-Objekte, die Multimediaobjekten mit völlig unzusammenhängenden Punken entsprechen. Dies würde bedeuten, dass bei der Wanderung durch den Hypothesenraum Fib-Objekte von "natürlichen" Multimediaobjekten auf mehr Wegen in Fib-Objekte umgewandelt werden können, die ähnlichen Multimediaobjekten entsprechen, als Fib-Objekte, die zufällige Multimediaobjekte (diese haben zufällige Eigenschaften für ihre Punkte) repräsentieren.

Wie in Abschnitt 15.1 auf Seite 84 aufgeführt können zumindest alle Rastergrafiken mit entsprechenden Fib-Objekten dargestellt werden. Es wurde auch ein Algorithmus zur Erzeugung eines (ineffizienten, weil großen) Fib-Objekts zu einer Rastergrafik vorgestellt (das Fib-Objekts kodiert dann die Rastergrafik). Dieser Ansatz kann wahrscheinlich auf beliebige, mit der Fib-Multimediasprache darstellbare, Multimediaobjekte erweitert werden. Auf diese Weise lässt sich zumindest ein Anfangspunkt (Fib-Objekt Obj) in einem Cluster zu einem Multimediaobjekt finden. Dieses Fib-Objekt Obj kann dann reduziert werden, indem beispielsweise die Fib-Objekte Obj' ermittelt werden, die nach der $<_{Fib}$-Relation kleiner sind ($Obj' <_{Fib} Obj$). Diese Vorgehensweise sollte dadurch vereinfacht werden, dass "natürliche" Multimediaobjekte wahrscheinlich größere solcher Cluster haben.

Zur Dichte der Repräsentationen eines Multimediaobjekts im Hypothesenraum kann weiterhin noch Folgendes gesagt werden:

Unter den richtigen Voraussetzungen symbolisiert jedes Individuum ein Multimediaobjekt vom gesuchten Typ (siehe Abschnitt 15.3 auf Seite 88). Das heißt: Wenn es beispielsweise n mögliche Rastergrafiken gibt (z. B. $n = 100 * 100$ Bildpunkte $*8$ Farben$= 80.000$ Möglichkeiten), gibt es auch nur n Fib-Objektgruppen im Hypothesenraum, die unterschiedliche Rastergrafiken des Typs symbolisieren. Wenn jeder Rastergrafik eine Klasse von Fib-Objekten, die dieses repräsentieren, zugeordnet wird, so gibt es im ganzen unendlichen Hypothesenraum nur endlich (n) viele verschiedene solcher Klassen, womit das Finden eines Fib-Objektes im unendlichen Hypothesenraum, das eine bestimmte Rastergrafik repräsentiert, sich nicht mehr als ganz so unmöglich darstellt.

16.4 Annahme über verdeckte Objekte

Annahme: Im genetischen Algorithmus können verdeckte Fib-Teilobjekte als Ressource für neue Individuen dienen.

Im verdeckten Multimediaobjektbereich können sich neue Teilobjekte entwickeln, die sich nicht negativ auf die Ähnlichkeit zum Originalmultimediaobjekt auswirken. Wenn ”Superindividuen“ vorhanden sind (starkes lokales Optimum), die wegen ihrer Ähnlichkeit zum Originalbild die ganze Population dominieren, können sie das Bilden von neuen, weniger ähnlichen Individuen unterdrücken. Trotzdem können Individuen, die das gleiche Multimediaobjekt repräsentieren, entstehen, die verdeckte Teilobjekte enthalten, welche die Überwindung des aktuellen lokalen Optimums ermöglichen. Dies setzt natürlich voraus, dass die Ähnlichkeit von Individuen höher bewertet wird als die Größe und dass kleine Änderungen in der Größe von Individuen die Fitness des Individuums nicht extrem verändern.

Dies sollte ”Monokulturen“ etwas entgegenwirken.

Dieser Effekt hat eventuell Ähnlichkeit mit dem Effekt, den eine veränderbare Populationsgröße bei genetischen Algorithmen hat. Nur wird nicht die Zahl der Individuen verändert, sondern die Anzahl der ”Teilgene“ dieser Individuen.

Teil IV
Der genetische Algorithmus

In diesem Abschnitt werden allgemeine Entwurfsentscheidungen und erste Analysen für den genetischen Algorithmus aufgestellt. Der realisierte genetische Algorithmus ist flexibel und erweiterbar gestaltet.

Der genetische Algorithmus ist natürlich auch ein evolutionärer Algorithmus. Die Bezeichnung “genetisch” bezieht sich auf die Fähigkeit des Algorithmus, Informationen von zwei oder mehr Individuen in einem neuen Individuum zu kodieren und darauf, dass er auf Informationen arbeitet, die ein (Multimedia-) Objekt kodieren und diese Objekte nicht direkt darstellen.

Der Algorithmus dient zur Erzeugung/Generierung von Fib-Objekten, welche ein Multimediaobjekt möglichst gut darstellen. Dem Algorithmus wird dafür ein bestimmtes Multimediaobjekt vorgegeben, für dass er Fib-Objekte/Individuen generiert, von denen gute selektiert werden. Die Generierung von Individuen kann auch Analysen des Multimediaobjekts beinhalten und die Nutzung oder Analyse von Informationen anderer Individuen. Welche Individuen gut sind, wird mithilfe von vorgegebenen Parametern entschieden (durch den Bewerter für Individuen).

Der Algorithmus besteht aus fünf separaten Teilen:

- dem Kernalgorithmus
- dem Bewerter für Individuen
- dem Mortalitätsbewerter
- dem Bewerter für Operatoren
- der Menge der Operatoren

In der Abbildung 11 ist eine Ablaufskizze des genetischen Algorithmus dargestellt.

Im Nachfolgenden werden die einzelnen Fib-Objekte als Individuen bezeichnet. Die Menge aller Individuen, die zu einem Zeitpunkt im Algorithmus vorhanden sind, wird kurz als Population bezeichnet.

17 Kernalgorithmus

Der Kernalgorithmus nutzt die Bewerter und die Operatoren, um den genetischen Algorithmus zu realisieren. Er soll möglichst einfach gehalten und flexibel sein.

Die Bewerter für Individuen oder Operatoren sind (über Parameter) austauschbar.

Der Kernalgorithmus beinhaltet einmal die Hauptschleife des genetischen Algorithmus, in dem die Operatoren aufgerufen und Individuen erzeugt werden.

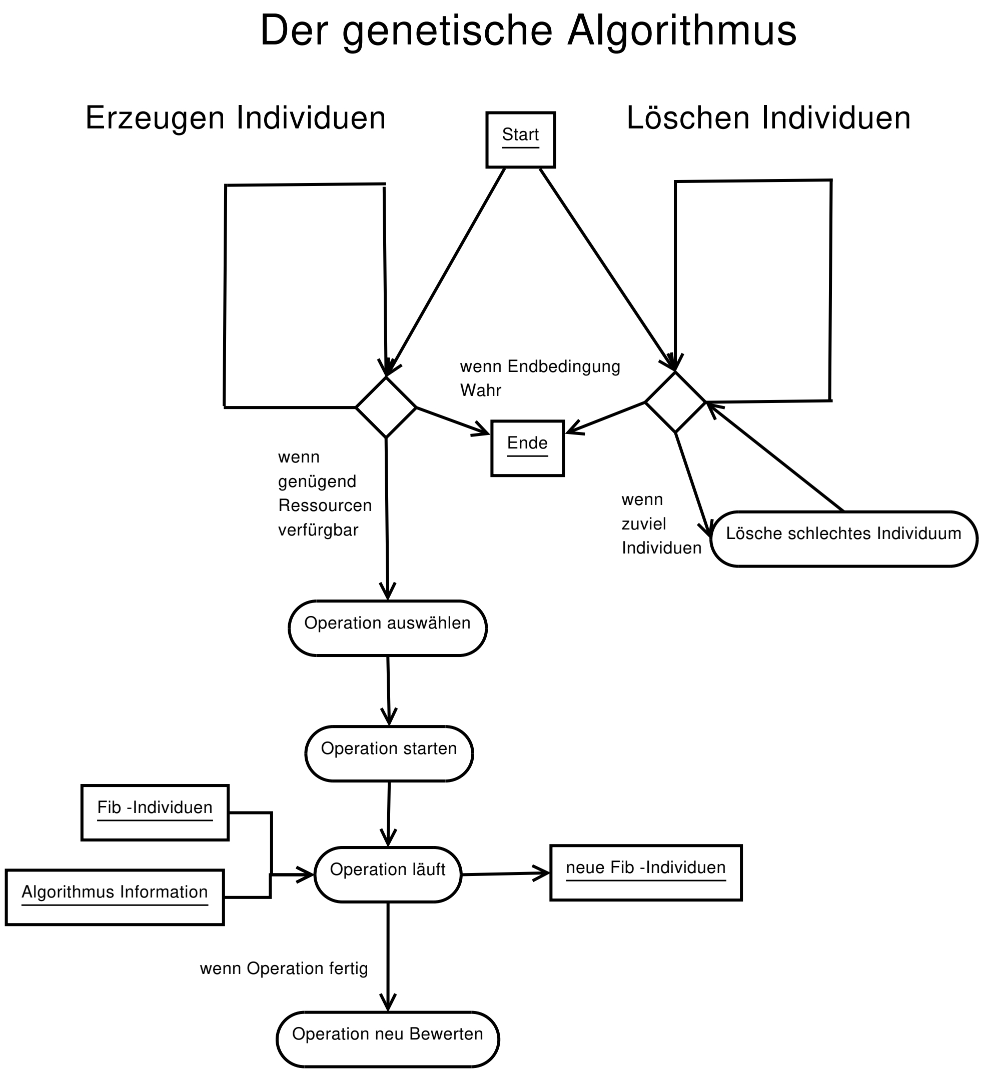


Abbildung 11: Ablaufskizze des genetischen Algorithmus

Als zweite Schleife existiert im Algorithmus die Selektionsschleife. Durch sie werden Individuen gelöscht.

Des Weiteren stellt der Kernalgorithmus Funktionen für die Operatoren bereit. Die Operatoren werden vom Kernalgorithmus über eine Methode aufgerufen, die für alle Operatoren gleich ist. Danach holen sich die Operatoren über Funktionen des Kernalgorithmus spezifische Werte aus dem Algorithmus, wie z. B. die Individuen, auf denen die Operationen arbeiten. Auf diese Weise sehen die Operatoren für den Kernalgorithmus immer jeweils gleich aus, und er benötigt keine Logik speziell für einen bestimmten Operator. Damit werden auch zukünftige Anpassungen der Operatoren vereinfacht, da nicht die Schnittstelle aller Operatoren angepasst werden muss, sondern nur die Funktionen, welche der Kernalgorithmus bereitstellt.

Die Aufrufmethode für die Operatoren soll flexibel sein, so dass auch Operatoren nebenläufig gestartet werden können und nicht auf die Beendigung jedes Operatoraufrufs gewartet werden muss. Die Operatoren sollten getrennt vom Kernalgorithmus laufen und diesen nur über die vorgegebene Schnittstelle beeinflussen können. Ein fehlerhafter Operator soll also nicht zu Fehlern im oder zur Beendigung des Gesamtsystems führen.

18 Bewerten von Individuen

Der konkrete Bewertungsalgorithmus sollte einfach auszutauschen sein, um verschiedenen Problemstellungen Rechnung tragen zu können. Der jeweilige Bewertungsalgorithmus wird zur Berechnung der Fitness eines Individuums benötigt.

Über Parameter für spezielle Bewerter können diese weiter angepasst werden (beispielsweise damit eingestellt werden kann, wie wichtig eine geringe Größe gegenüber eine schnellen Auswertung von Individuen ist).

18.1 Die Fitness eines Individuums

Die Fitness eines Individuums ist durch verschiedene Fitnessfaktoren gegeben.

Einer der wichtigsten ist, inwieweit das Individuum den gewünschten Originalmultimediadaten (z. B. dem Originalbild) ähnelt. Je mehr das Individuum (bzw. der Phänotyp dessen) den Originalmultimediadaten ähnelt, umso höher sollte die Fitness sein und umso geringer ist der Fehler, den das Individuum für die Darstellung der Originalmultimediadaten macht.

Dieser Fehler (und damit die Fitness des Individuums) kann z. B. über die Summe der (quadratischen) Abweichungen (nicht definierte Punkte liefern einen maximalen Fehler) in den Farben zu einem Punkt zwischen dem Originalbild und dem vom Individuum erzeugten Bild bestimmt werden oder über ein anderes selbstbestimmtes Abstandsmaß.

Wenn die Fitness für einzelne Teilobjekte des Individuums bestimmt wird, kann dies z. B. dadurch realisiert werden, dass in die Rechnung nur der Bereich einbezogen wird, der durch das Teilobjekt überdeckt wird, ein Rand um diesen Bereich noch mit einbezogen wird, oder auch nur das kleinste Quadrat benutzt wird, das dieses Objekt umschließt.

Ein anderer sinnvoller Fitnessfaktor ist die Größe (steigt mit der Anzahl der Elemente) der einzelnen Individuen, um größeren Individuen eine geringere Fitness zu geben als kleineren Individuen, mit dem gleichen Fehler auf den Originalmultimediadaten, und die kleineren Individuen so zu bevorzugen.

Ein weiterer Fitnessfaktor der einbezogen werden kann, ist eine Schätzung über die Zeit, die für das Individuum zur Berechnung des dargestellten Multimediaobjekts (Phänotyp) benötigt wird. Damit kann eventuell sogar die Ausführungsgeschwindigkeit des Algorithmus erhöht werden.

18.2 Selektion durch Löschen von Individuen

Um die Ressourcen (Arbeitspeicher, Rechenzeit) zu schonen, ist es notwendig, die Anzahl der Individuen (Fib-Objekte) im Bearbeitungsprozess (Individuen die noch am genetischen Algorithmus teilnehmen) zu beschränken. Deshalb müssen Individuen aus diesem (nach Bedarf) entfernt werden. Dabei sind Individuen mit einer niedrigen Fitness zu bevorzugen.

Deshalb gibt es für den Algorithmus einen **Mortalitätsbewerter**. Dieser bestimmt, mit welcher Wahrscheinlichkeit ein Individuen gelöscht wird. Um Individuen zu löschen, gibt es im Algorithmus eine seperate Schleife, in der geprüft wird, ob die maximale Anzahl von Individuen überschritten ist. Wenn dies der Fall ist, werden so viele Individuen gelöscht, bis die maximale Anzahl von Individuen wieder eingehalten wird. Dabei haben Individuen mit einer hohen Bewertung durch den Mortalitätsbewerter auch eine hohe Wahrscheinlichkeit, gelöscht zu werden.

Der Mortalitätsbewerter orientiert sich für seine Bewertung an der Fitness der Individuen. Er kann allerdings durch andere Mortalitätsbewerter ausgetauscht oder durch Parameter gesteuert werden. Es ist beispielsweise auch möglich, einige Individuen als unsterblich bzw. nicht löschbar zu deklarieren. Indem z. B. die n ($n > 0$) besten Individuen als unsterblich deklariert werden, kann vermieden werden, dass diese gelöscht werden und somit eines der besten Individuen verloren geht.

Der Mortalitätsbewerter kann weiterhin über die Operatorschnittstelle auf Statusinformationen des Algorithmus zugreifen, um z. B. die Anzahl der bisher erzeugten Individuen zu ermitteln.

19 Bewerter für Operatoren

Um zu bestimmen, welcher Operator als nächstes ausgewählt wird, sind diese zu bewerten. Operatoren, die besser in einer Situation bewertet werden, haben eine höhere Wahrscheinlichkeit, in einer ähnlichen Situation ausgewählt bzw. ausgeführt zu werden.

Die Situation kann umfassen:

- die wievielte Operation bzw. Interaktion ausgeführt wird.
- welcher Art das Originalmultimediaobjekt ist (über die Definitionen für die Umgebung in den root-Elementen dieses):
  - es enthält Farben
  - es ist Schwarz/Weiß
  - es enthält Ton
  - es handelt sich um einen Film
  - ...
- die durchschnittliche (relative) Fitness der Individuen.
- die Fitness des besten/schlechtesten Individuums.
- die Standardabweichung der Fitnesswerte in der Population.
- die Anzahl der Individuen.
- Operationen, die bisher angewendet wurden.
- ...

Der spezielle Bewertungsalgorithmus kann ausgewählt werden. So können verschiedene Bewertungsalgorithmen leicht gegeneinander ausgetauscht und verglichen werden.

Zur Bewertung der Operatoren werden (von einzelnen Bewertern) eventuell Daten über ihre bisherigen Anwendungen permanent gehalten. Auf diese Weise kann der Algorithmus aus vorhergehenden Operatoraufrufen lernen.

Die Bewertung der Operatoren sollte möglichst Systemunabhängig erfolgen, also unabhängig vom Rechner, auf dem der Algorithmus gerade läuft.

Die Bewertungskriterien können sein:

- Ausführungszeit der Operation
- erreichte Verschlechterungen oder Verbesserungen
- Zuverlässigkeit des Operators (Gibt er immer ein Ergebnis zurück? Stürzt er manchmal ab?)
- ...

20 Die genetischen Operationen auf Fib

Die Operatoren sind vom genetischen Algorithmus getrennt zu sehen. Es sollte möglich sein, beliebig viele Operatoren zum genetischen Algorithmus hinzuzufügen, ohne diesen anpassen zu müssen.

Jeder Operator hat eine eindeutige Kennung bzw. ID, über die ihm zugehörige Werte (z. B. seine bisherige Performance) zugeordnet werden können.

Wird ein Operator ausgeführt, ist dies eine Operation.

Diese Operationen sind dazu gedacht, Kodierungsalgorithmen zu implementieren. Operatoren sollten also nicht möglichst einfach sein, sondern können durchaus komplexe Algorithmen beinhalten.

Die Operatoren sollen auch möglichst zahlhaft sein, und der Algorithmus ist für die Auswahl guter Operatoren verantwortlich. Daher ist es auch erwünscht, mit sinvollen Teilen von Operatoren eigene Operatoren zu erstellen. Dabei sollten sowohl die Originaloperatoren als auch der Operator, der den extrahierten Teil enthält, im Algorithmus verbleiben.

Für Spezialanwendungen kann ein genetischer Algorithmus verwendet werden, dessen Menge der Operatoren auf die nützlichsten für diese Anwendung eingeschränkt wurde. Auch können für eine Anwendung nützliche Operatoren zu einem (nicht genetischen) Algorithmus kombinert werden, der diese in determinerter Weise verbindet.

20.1 Vermehrung

Bei der “Vermehrung” wird im Allgemeinen ein Individuum aus der Population genommen, bevorzugt Individuen mit hoher Fitness, dieses kopiert und durch eine Operation verändert. Ist das entstehende Individuum ein “neues” Objekt (wenn es kein gleiches in der Population gibt), wird es zur Menge hinzugenommen. Es können auch bereits vorhandene Individuen übernommen werden.

Im Algorithmus wird die Vermehrung dadurch realisiert, dass eine Operation ausgeführt wird. Diese Operation holt sich dann über die Operationenschnittstelle vom Algorithmus alle benötigten Daten. Zu den benötigten Daten können ein oder mehrere Individuen gehören oder Daten über den bisherigen Verlauf des Algorithmus (z. B. die Nummer der bisherigen Interaktionen bzw. Operationen).

21 Der soziale Aspekt des genetische Algorithmus

Die strikte Trennung der Operatoren vom Algorithmus und die Möglichkeit, Operationen leicht hinzufügen zu können, hat seinen Grund in eher sozialen Überlegungen.

Normale Kodierungsalgorithmen beschränken sich bei den Kodierungsmethoden auf eine oder nur wenige Ideen von einigen (in der Größenordnung von 1 bis 100) Menschen (dem Entwicklerteam). Da es aber viel mehr (in der Größenordnung von wahrscheinlich 100000) Menschen weltweit gibt, die sich im weiteren

Sinne mit der effizenten Kodierung von Multimediaobjekten beschäftigen, bleiben dadurch zwangsläufig auch viele Ideen zur Kodierung unberücksichtigt. Selbst wenn ein Teil dieser Menschen daran Interesse hat, ihre Ideen einzubringen, ist dies nur sehr schwer bis unmöglich zu realisieren.

In den Fib-Algorithmus kann aber jeder neue Ideen in Form neuer Operatoren einbringen (ein Begriff dafür ist “Crowdsourcing”). Dafür sind lediglich ausreichende Kenntnisse der Fib-Multimediabeschreibungssprache und der Schnittstelle des Algorithmus für die Operatoren nötig. Um neue Ideen bzw. Operatoren zur effizenten Kodierung von Multimediaobjekten einzubringen, sollten keine Anpassungen am Algorithmus oder anderen Operatoren nötig sein. Dadurch werden Seiteneffekte beschränkt und jede Implementierung einer Idee hat sich nur um die Idee bzw. deren Operator zu kümmern. Es sind also keine weiteren Kenntnisse des Algorithmus oder gar anderer Operatoren von Nöten. Auf diese Weise kann das System wachsen, ohne dass sich die Komplexität des (Kern-)Systems vergrößert und die Wartung und Erweiterung des Systems schwieriger wird.

Der Algorithmus und die Bildbeschreibungssprache sollten so angelegt sein, dass sich auch Laien ohne größeren Aufwand einarbeiten und neue Ideen bzw. Operatoren realisieren können. Insbesondere sollten Studenten und Studierende (z. B. Menschen mit Informatik als Hobby) der Informatik, sich innerhalb weniger Tage soweit einarbeiten können, dass sie einen eigenen Operator realisieren und einbinden können. (Wie nützlich oder effizient dieser ist, sei erst einmal dahingestellt.)

Die GNU-GPL-Lizenz, unter dem der Algorithmus steht, klärt die rechtliche Situation, unter der neue Operatoren stehen. Dadurch können neue Operatoren legal eingebunden werden, solange nicht andere Rechte verletzt werden (eine Verletzung wäre beispielsweise, dass in den Operatoren verwendete Algorithmen oder Codes unter inkompatiblen Lizenzen stehen). Neue Operatoren können auch der Allgemeinheit zur Verfügung gestellt werden.

Die Bewertung der Operatoren sollte motivieren, eigene Operatoren einzubinden. Dadurch kann jeder, der einen Operator hinzugefügt hat, realistisch einschätzen, wie gut sein Operator sich im Verhältnis zu anderen Operatoren in einer Situation macht. Der Wettbewerb unter Operatorenautoren sollte für neue bessere Operatoren förderlich sein.

All dies sollte dazu führen, dass nicht nur ein kleines Entwicklerteam zur Verbesserung der Kodierung von Fib-Objekten beiträgt, sondern dass ein viel größerer Kreis von Menschen sich mit diesen Thema beschäftigt. Dies sollte der Entwicklung und Verbreitung von Fib einen weiteren Schub geben.

In diesem Sinne ist der genetische Algorithmus für Fib ein transgenialer Algorithmus, der darauf angelegt ist, die Ideen von Menschen zur Kodierung von Multimediadaten aus deren Köpfen herauszuholen und in einem Topf zu transportieren/sammeln. Damit sollen diese Ideen/Algorithmen mehr leisten können, als sie es einzeln könnten. Der Algorithmus kann damit mehr Intelligenz in sich vereinen, als es ein Mensch (oder auch eine kleine Gruppe) hervorbringen kann.

22 Warum sich genetische Algorithmen zur Multimediakodierung anbieten

Es gibt im Allgemeinen keinen Algorithmus der Rastergrafiken direkt in speicherschonende Vektorbilder umwandeln kann, bei denen die Stärken der entsprechenden Vektorbeschreibungssprache wirklich genutzt werden. Für die Umwandlung von einem Multimediaformat in ein anderes ergibt sich oft eine ähnliche Problematik, insbesondere wenn das erste Multimediaformat die Eigenschaften von Punkten eines euklidischen (diskreten) Raumes direkt angibt (z. B. um Aufnahmen direkt abspeichern zu können) und das zweite Multimediaformat komplexe Objekte (z. B. Rechtecke oder/und Kreise) in Multimediaobjekten kodiert.

Da ein genetischer Algorithmus das Potential hat, alle möglichen Beschreibungen einer Multimediasprache zu generieren und unter diesen natürlich auch gute Beschreibungen sind, welche die Stärken der Multimediasprache in Bezug auf das Originalobjekt nutzen, hat ein genetischer Algorithmus natürlich auch das Potential, gute Beschreibungen zu generieren. Wenn im genetischen Algorithmus vorhandenes Wissen gut eingebaut wurde, kann er gute Beschreibungen wahrscheinlich auch schneller finden als eine reine Zufallssuche.

Noch ein weiterer Vorteil von genetischen Algorithmen ist die große Freiheit bei der Wahl der Problembeschreibung (Multimediasprache) und der möglichen Operatoren auf dieser. So kann völlig frei eine Multimediasprache nach eigenen Vorstellungen entworfen werden, die bestimmte Eigenschaften hat, z. B. Lesbarkeit, Einfachheit. Bei den Operatoren kann beliebig viel Wissen eingebaut werden. So ist es unter anderem auch möglich, schon bekannte gute Algorithmen zur Übersetzung von Rastergrafiken in Vektorbilder oder Teile von ihnen in Operatoren zu verwenden, so dass der genetische Algorithmus auf Rastergrafiken vom Ergebnis her mindestens so gut wird wie der verwendete Algorithmus, aber noch bessere Ergebnisse erzeugen kann.

Der große Nachteil von genetischen Algorithmen, dass sie sehr viel Zeit oder Rechenaufwand benötigen, wird dadurch abgeschwächt, dass dieser anfänglich hohe Aufwand "billig" sein kann und sich später auszahlt. Der genetische Algorithmus kann z. B. als Hintergrundprozess mit niedriger Priorität laufen, so dass er nur überflüssige Rechnerleistung verbraucht. Später kann durch das Ergebnis, das er geliefert hat, viel Übertragungsbandbreite eingespart werden.

Dies alles spricht für den Versuch, genetische Algorithmen zur Multimediakodierung zu verwenden.

23 Komplexitätsabschätzung

Mit dem Ansatz aus Abschnitt 15.1 auf Seite 84 kann eine beliebiges Rastergrafik in ein Fib-Objekt konvertiert werden, wobei der Kodierungsaufwand linear mit der Anzahl der Punkte und Eigenschaften wächst, da jeder Punkt mit seinen Eigenschaften einfach in ein Listenelement eingefügt werden kann und die Erzeugung der sonstigen Fib-Elemente (des root-Elements) konstante Zeit benötigt.

Dieser Ansatz ist auf beliebige Multimediadaten erweiterbar, die als Eigenschaften von Punkten eines endlichen, euklidischen, diskreten Raumes darstellbar sind. So kann immer ein Operator konstruiert werden, der ein Multimediaobjekt in linearer Zeit mit der Anzahl der Punkte und Eigenschaften in ein entsprechendes Fib-Objekt umwandelt. Dieses Fib-Objekt wächst auch von der Größe her nur linear mit der Anzahl der Punkte und Eigenschaften des Originalmultimediaobjekts. Dieser Operator erzeugt allerdings keine effiziente Fib-Objektdarstellung, da er nicht die Möglichkeiten von Fib nutzt.

Wieviel Aufwand die Erzeugung besserer Fib-Objekte benötigt, kann nur schwer abgeschätzt werden, da sowohl die Operatoren als auch die Multimediaobjekte beliebig sein können. Es kann aber davon ausgegangen werden, dass für Multimediaobjekte mit einfacheren Strukturen der Aufwand auch geringer ist.

24 Parallelen zur natürlichen Evolution

Als Parallele seien hier die Bakterienstämme (z. B. E. coli) aufgeführt. Sie besitzen Gene, aufgrund derer bei ihnen genetische Evolution auftritt.

Bei diesen Bakterienstämmen (z. B. E. coli) kann es auch zu einem Gentransfer kommen, dabei wird ein Teil der Geninformationen (des Genmaterials oder dessen Kopien) von einer Bakterie zu einer anderen übertragen. Daneben gibt es natürlich auch Mutationsprozesse.

Ein Fib-Objekt kann als Information betrachtet werden, die ein Multimediaobjekt kodiert, so wie Bakteriengene eine Bakterie kodieren (Aufbau und Verhalten).

Die einzelnen Fib-Elemente sind dabei weniger als die Basen der Gene zu sehen, sondern vielmehr als die Funktionalität von Genen oder Genmengen. So wie bei den Bakterien durch Kombination der Gene bzw. der Dinge, die sie kodieren (z. B. Enzyme), erst komplexere Funktionalitäten definiert werden (z. B. Umsetzung von Zucker in Bewegungsenergie), entstehen bei Fib-Objekten erst durch Kombination der Elemente (Teil-)Multimediaobjekte (z. B. [Teil-]Bilder).

Fib-Teilobjekte können auch, wie beim Gentransfer bei den Bakterien, in andere Fib-Objekte übertragen werden und unterliegen einer Mutation, wobei auch bei den Bakterien schwer zu sagen ist, ob die Mutation der Gene wirklich völlig zufällig ist oder ob im Laufe der Jahrmillionen nicht Mechanismen entstanden sind, die zu einer "intelligenteren" Mutation führen, bzw. worin das zufällige Element besteht. Einzelne Fib-Objekte können dabei nicht nur als einzelne Bakterien angesehen werden, sondern auch als alle Bakterien mit identischen Geninformationen.

Das Originalmultimediaobjekt stellt die Nische dar, an welche sich die Fib-Objekte (Bakterien) anpassen sollen. Sie können dies auf viele unterschiedliche Arten tun, und die Anpassung muss auch nicht perfekt sein.

Allerdings werden beim genetischen Algorithmus für Fib-Objekte Operatoren angestrebt, die gezielte Verbesserungen vornehmen. Ein solcher explizit gerichteter Mechanismus ist bei Bakterien nicht zu erwarten.

Für weitere Informationen zur Genetik bei Bakterien sei auf “Einführung in die Mikrobiologie”([1]) verwiesen. Ein gutes Buch zur Evolution im allgemeinen ist “Die Lösung von Darwins Dilemma” [13] .

Teil V
Fib-Dateiformate

In diesen Abschnitt werden die Speicherformate vorgestellt, in denen vollständige Fib-Objekte abgespeichert werden. Davon gibt es zwei: ein Format zum möglichst komprimierten Abspeichern (siehe Abschnitt 25 auf Seiet 105) und ein Format, um ein Fib-Objekt gut lesbar als XML-Struktur darzustellen (siehe Abschnitt 26 auf Seiet 141).

25 Komprimierte Fib-Daten

Da eine Speicherplatz sparende Repräsentation eines der Hauptziele der Fib-Multimediabeschreibungssprache ist, sollten die Fib-Elemente mit möglichst wenig Bits bzw. Bytes gespeichert werden können. Dafür wird hier ein komprimierendes Speicherformat definiert. Es wird kein Standardkomprimierungsalgorithmus verwendet, da dieser keine Rücksicht auf Besonderheiten der Fib-Multimediabeschreibungssprache nehmen würde.

Die Darstellung von **ganzen Zahlen** erfolgt im Zweierkomplement System (auch 2-Komplement, Zweikomplement, Basiskomplement, K2-Zahl, 2K-Zahl, two's complement). Für **natürliche Zahlen** (inklusive der 0) wird das Dualsystem verwendet. Die Anzahl der Bits von Vektorelementen oder Variablen wird jeweils durch den entsprechenden Definitionsbereich bestimmt (z. B. für Zahlen für die horizontale Dimension oder einen Grauwert) oder fest vorgegeben (z. B. für Zahlen für das Byte Offset in den root-Elementen).

Reelle Zahlen werden als Gleitkommazahlen (auch Fließkommazahl engl. floating point number) dargestellt. Die Anzahl der Bits für den Exponenten und die Mantisse werden durch die jeweilige Definitionsbereichsdefinition angegeben.

Die Zahlen werden im Little Endian Format gespeichert.

Alle **Texte** werden in Unicode kodiert.

Für die nachfolgende Beschreibung ist leider die **Schreibweise von Bits** nicht so einfach umzusetzen. In der normalen Schreibweise werden Zahlen von rechts nach links, Texte allerdings von links nach rechts geschrieben. Dies führt dazu, dass kurze Bitfolgen, so wie sie in einem Byte auftauchen, eher wie Zahlen von rechts nach links geschrieben werden. Wenn die Bitfolgen allerdings länger werden bzw. wenn Bits in einer Datei oder einem Datenstrom betrachtet werden, ist die Schreibweise wie in Texten von links nach rechts eher angebracht. Ansonsten, wenn die Zahlenschreibweise angewendet wird, steht am Ende einer langen Zeile von Bits das erste Bit aus der Datei.

Im Nachfolgenden werden deshalb die Bits einzelner Elemente wie Zahlen von von rechts nach links geschrieben, da diese Bitfolgen meist kurz sind. Einzelne Elemente werden am Computer auch meist im Hexadezimalsystem angezeigt und verarbeitet, welches auch wie Zahlen von rechts nach links arbeitet. Die umgekehrte

Schreibweise würde die Arbeit mit den komprimierten Elementen nur erschweren. Die Bits einzelner Bytes werden meistens der Übersichtlichkeit halber mit einem Semikolon (';') getrennt.

Wenn allerdings ein Datenstrom aus mehreren Elementen dargestellt wird, sind die Bits wie bei Texten von links nach rechts angeordnet. Dies führt leider dazu, dass die Bits der einzelnen Elemente in umgekehrter Reihenfolge dargestellt werden.

25.1 Dateiheader

Jede komprimierte Fib-Datenstrom beginnt mit den drei Buchstaben "fib". Die Dateiendung für komprimierte Fib-Dateien sollte ".fib" sein.

25.2 Root-Element

Zur Beschreibung des root-elements siehe Abschnitt 12.14 auf Seite 52 .

Das Fib-root-Element braucht keine separate Einleitung. Die Daten des obersten root-Elements beginnen mit dem dritten Byte (die Zählung beginnt bei 0) direkt nach der Datenstromeinleitung "fib". Andere root-Elemente folgen auf ihren Identifier.

Für das root-Element werden nacheinander folgende Felder geschrieben (dabei wird jedes Element jeweils fals nötig auf ein volles Byte mit 0 aufgefüllt):

1. 16-Bit-Feld zur Angabe der optionalen Informationsfelder (siehe Abschnitt 25.2.1)

2. 64-Bit-Feld zur Angabe weiterer optionaler Informationsfelder (siehe Abschnitt 25.2.1); nur vorhanden, wenn Bit 16 der optionalen Informationsfelder gesetzt ist

3. ein $144(= 16 + 64 + 64)$-Bit-Feld für die Checksumme (siehe Abschnitt 25.2.2 auf Seite 111); nur vorhanden, wenn Bit 1 der optionalen Informationsfelder gesetzt ist

4. Nummer des Bytes des root-Objekts (Offset), ab dem die Definitionsbereiche ($Domains$ und $DomainsValues$) definiert werden; nur vorhanden, wenn Bit 3 der optionalen Informationsfelder gesetzt ist

5. Nummer des Bytes des root-Objekts (Offset), ab dem die Eingabevariablen definiert werden; nur vorhanden, wenn Bit 4 der optionalen Informationsfelder gesetzt ist

6. Nummer des Bytes des root-Objekts (Offset), ab dem das Haupt-Fib-Objekt definiert wird

7. Nummer des Bytes des root-Objekts (Offset), ab dem die weiteren root-Objekte definiert werden; nur vorhanden, wenn Bit 6 der optionalen Informationsfelder gesetzt ist

8. Nummer des Bytes des root-Objekts (Offset), ab dem die Identifier von verwendeten Datenbankobjekten aufgelistet werden; nur vorhanden, wenn Bit 7 der optionalen Informationsfelder gesetzt ist

9. Nummer des Bytes des root-Objekts (Offset), ab dem der optionale Teil (Optionalpart) kommt; nur vorhanden, wenn Bit 8 der optionalen Informationsfelder gesetzt ist

10. Nummer des Bytes des root-Objekts (Offset), ab dem das root-Objekt zu Ende ist, bzw. Anzahl der Bytes, welche das root-Objekt lang ist

11. Multimediainformationen (siehe Abschnitt 25.2.3 auf Seite 111); nur vorhanden, wenn Bit 2 der optionalen Informationsfelder gesetzt ist

12. Definitionsbereiche (siehe Abschnitt 25.2.4 auf Seite 113); nur vorhanden, wenn Bit 3 der optionalen Informationsfelder gesetzt ist

13. Eingabevariablen (siehe Abschnitt 25.2.5 auf Seite 126); nur vorhanden, wenn Bit 4 der optionalen Informationsfelder gesetzt ist

14. Haupt-Fib-Objekt (siehe Abschnitt 25.2.6 auf Seite 126)

15. weitere root-Objekte (siehe Abschnitt 25.2.7 auf Seite 126); nur vorhanden, wenn Bit 6 der optionalen Informationsfelder gesetzt ist

16. Identifier von verwendeten Datenbankobjekten (siehe Abschnitt 25.2.8 auf Seite 127); nur vorhanden, wenn Bit 7 der optionalen Informationsfelder gesetzt ist

17. der optionale Teil (siehe Abschnitt 25.2.9 auf Seite 127); nur vorhanden, wenn Bit 8 der optionalen Informationsfelder gesetzt ist

Wenn ein einzelnes Bit in den optionalen Informationsfeldern für ein Feld nicht gesetzt ist, entfällt diese Feld.

Für die Felder mit der “Nummer des Bytes des root-Elements (Offset)” werden jeweils 8 Byte oder 64 Bit verwendet. Die Zahl im Feld gehört zu den natürlichen Zahlen. Angegeben wird jeweils die Nummer des Bytes ab Anfang des root-Elements, ab dem das entsprechende Element beginnt (also für das erste Byte des Elements). Die Zählung der Bytes im root-Element beginnt bei 0. Das optionale Informationsfeld hat also den Offset 0.

Alle Texte, die nicht im optionalen Teil stehen, werden in den optionalen Teil ausgelagert (siehe Abschnitt 25.2.9 auf Seite 127).

25.2.1 Optionale Informationsfelder

An dieser Stelle steht ein 16-Bit-Feld, dessen Bits das Vorhandensein von optionalen Informationsfeldern im root-Element anzeigt.

Dabei schalten die einzelnen Bits (die Zählung beginnt bei 1) die folgenden Informationsfelder:

1 Checksumme (siehe Abschnitt 25.2.2 auf Seite 111)

2 Multimediainformationen (siehe Abschnitt 25.2.3 auf Seite 111)

3 Definitionsbereiche (siehe Abschnitt 25.2.4 auf Seite 113)

4 Eingabevariablen (siehe Abschnitt 25.2.5 auf Seite 126)

5 frei

6 weitere root-Objekte (siehe Abschnitt 25.2.7 auf Seite 126)

7 Identifier von verwendeten Datenbankobjekten (siehe Abschnitt 25.2.8 auf Seite 127)

8 optionaler Teil (siehe Abschnitt 25.2.9 auf Seite 127)

- 9 bis 15 Bits sind noch nicht belegt und stehen für zukünftige Belegungen zur Verfügung

16 weitere optionale Felder, angegeben durch ein folgendes 64-Bit-Feld für weitere optionale Informationsfelder

Im Nachfolgenden werden die einzelnen Bits, wann sie eingesetzt werden und ihre Auswirkung beschrieben.

1. Checksummenbit

Gesetzt wenn: Ein Checksummenfeld im root-Element existiert.

Auswirkungen wenn gesetzt: Ein Checksummenfeld (siehe Abschnitt 25.2.2 auf Seite 111) ist vorhanden.

Nicht gesetzt wenn: Es existiert kein Checksummenfeld im root-Element.

Auswirkungen wenn nicht gesetzt: Ein Checksummenfeld ist nicht vorhanden.

2. Multimediainformationenbit

Gesetzt wenn: Für das aktuelle root-Element existieren Multimediainformationen, welche von den ererbten Multimediainformationen abweichen. Dabei werden Multimediainformationen eines root-Elements jeweils an in diesem enthaltende root-Elemente vererbt, wenn die enthaltenden root-Elemente nicht selbst (irgendwelche) abweichende Multimediainformationen definieren.

Auswirkungen wenn gesetzt: Es werden die Multimediainformationen für das aktuelle root-Element angegeben (siehe Abschnitt 25.2.3 auf Seite 111).

Nicht gesetzt wenn: Die Multimediainformationen entsprechen den ererbten Multimediainformationen.

Im obersten root-Element (bzw. in einem root-Element, das in keinem anderen root-Element vorhanden ist) muss das Bit immer gesetzt sein und damit die Multimediainformationen vorhanden sein.

Auswirkungen wenn nicht gesetzt: Es werden keine Multimediainformationen für das aktuelle root-Element angegeben. Die gültigen Multimediainformationen für das root-Element werden von dem root-Element, in dem es vorhanden ist, ererbt.

3. Definitionsbereiche

Gesetzt wenn: Für das aktuelle root-Element existieren Definitionsbereiche.

Auswirkungen wenn gesetzt: Es werden die Definitionsbereiche für das aktuelle root-Element ausgegeben (siehe Abschnitt 25.2.4 auf Seite 113) und der Offset, wann diese im root-Element beginnen.

Nicht gesetzt wenn: Für das aktuelle root-Element existieren keine Definitionsbereiche.

Auswirkungen wenn nicht gesetzt: Es wird kein Offset ausgegeben, wann die Angabe der Definitionsbereiche im root-Element beginnt, und es werden keine Definitionsbereiche für das aktuelle root-Element angegeben.

4. Eingabevariablen

Gesetzt wenn: Für das aktuelle root-Element existieren Eingabevariablen.

Auswirkungen wenn gesetzt: Es werden die Eingabevariablen für das aktuelle root-Element angegeben (siehe Abschnitt 25.2.5 auf Seite 126) und der Offset, wann diese im root-Element beginnen.

Nicht gesetzt wenn: Für das aktuelle root-Element existieren keine Eingabevariablen.

Auswirkungen wenn nicht gesetzt: Es wird kein Offset ausgegeben, wann die Angabe der Eingabevariablen im root-Element beginnt, und es werden keine Eingabevariablen für das aktuelle root-Element angegeben.

6. weitere root-Objekte

Gesetzt wenn: Für das aktuelle root-Element existieren Unter-root-Objekte.

Auswirkungen wenn gesetzt: Es werden die Unter-root-Objekte für das aktuelle root-Element angegeben (siehe Abschnitt 25.2.7 auf Seite 126) und der Offset, wann diese im root-Element beginnen.

Nicht gesetzt wenn: Für das aktuelle root-Element existieren keine Unter-root-Objekte.

Auswirkungen wenn nicht gesetzt: Es wird kein Offset ausgegeben, wann die Angabe der Unter-root-Objekte im root-Element beginnt, und es werden keine Unter-root-Objekte für das aktuelle root-Element angegeben.

7. Identifier von verwendeten Datenbankobjekten

Gesetzt wenn: Für das aktuelle root-Element werden Identifier von verwendeten Datenbankobjekten angegeben.

Auswirkungen wenn gesetzt: Es werden die Identifier von verwendeten Datenbankobjekten für das aktuelle root-Element angegeben (siehe Abschnitt 25.2.8 auf Seite 127) und der Offset, wann diese im root-Element beginnen.

Nicht gesetzt wenn: Für das aktuelle root-Element existieren keine Identifier von verwendeten Datenbankobjekten.

Auswirkungen wenn nicht gesetzt: Es wird kein Offset ausgegeben, wann die Angabe der Identifier von verwendeten Datenbankobjekten im root-Element beginnt, und es werden keine Identifier von verwendeten Datenbankobjekten für das aktuelle root-Element angegeben.

8. optionaler Teil

Gesetzt wenn: Für das aktuelle root-Element wird der optionale Teil angegeben.

Auswirkungen wenn gesetzt: Es wird der optionale Teil für das aktuelle root-Element angegeben (siehe Abschnitt 25.2.9 auf Seite 127) und der Offset, wann diese im root-Element beginnen.

Nicht gesetzt wenn: Für das aktuelle root-Element ist kein optionaler Teil vorhanden, weil diese beispielsweise nicht mit abgespeichert wurden.

Auswirkungen wenn nicht gesetzt: Es wird kein Offset ausgegeben, wann die Angabe der optionale Informationen im root-Element beginnt, und es werden keine optionalen Informationen für das aktuelle root-Element angegeben.

16. weitere optionale Felder

Gesetzt wenn: 64 weitere optionale Informationsbits sind vorhanden.

Auswirkungen wenn gesetzt: Auf dem 16-Bit-Feld zur Bestimmung der vorhandenen optionalen Informationen folgt noch ein weiteres 64-Bit-Feld zur Bestimmung von weiteren optionalen Informationen. Diese Bits sind für zukünftige Benutzungen gedacht und werden zur Zeit noch nicht verwendet.

Nicht gesetzt wenn: Die 64 weiteren optionalen Informationsbits sind nicht vorhanden.

Auswirkungen wenn nicht gesetzt: Auf den 16-Bit-Feld folgen keine weiteren Bits zur Anzeige der vorhandenen optionalen Informationen.

25.2.2 Checksummenfelder

Mit diesem Feld kann auch das root-Element mit einer Checksumme versehen werden.

Das Verfahren entspricht dem der Checksummeneigenschaft aus dem Abschnitt 25.3.3 auf Seite 131.

An dieser Stelle stehen 3 Parameter, welche eine 16 Bit und zwei 64 Bit natürliche Zahlen sind. Der erste Parameter A gibt die Art der Checksumme an. Der zweite Parameter B gibt an, alle wieviel Bits eine Checksumme generiert werden soll und der dritte Parameter C, wieviel Bits die Checksumme haben soll. Die Angaben gelten für den Bereich nach den drei Parametern (auch in Unter-root-Elementen). Die Checksumme wird so wie in Abschnitt 25.3.3 auf Seite 131 beschrieben umgesetzt.

25.2.3 Multimediainformationen

In der Tabelle 9 ist der Aufbau des Multimediainformationsabschnitts eines root-Elements beschrieben. Die Größe eines Multimediabschnitts ist $2 * 64 = 128$ Bit oder 16 Byte.

Element	Anzahl	Bits	Typ	Beschreibung
Fib-Version	1	64	natürliche Zahl	Die Versionsnummer der Fib-Multimediabeschreibungssprache, die zum Laden des Fib-Objekts benötigt wird. Diese Nummer wird mit jeder neuen Version der Fib-Multimediabeschreibungssprache um eins erhöht. Sie kann zu einer menschenlesbaren Form (z. B. “Fib V1.2.3”) gemappt werden. Hier wird allerdings nur eine Zahl verwendet, da ansonsten eine bestimmte Form festgelegt würde, die nur noch schwer geändert werden kann. Es kann eine menschenlesbaren Form der Version im optionalen Teil angegeben werden.
DB-Version	1	64	natürliche Zahl	Die Versionsnummer der Fib-Datenbank, die zum Laden des Fib-Objekts benötigt wird. Diese Nummer wird mit jeder neuen Version der Fib-Datenbank um eins erhöht. Sie kann zu einer menschenlesbaren Form (z. B. “Fib DB V1.2.3”) gemappt werden. Hier wird allerdings nur eine Zahl verwendet, da ansonsten eine bestimmte Form festgelegt würde, die nur noch schwer geändert werden kann. Es kann eine menschenlesbaren Form der Version im optionalen Teil angegeben werden.

Tabelle 9: Daten der Multimediainformationen

25.2.4 Definitionsbereiche

Der Definitionsbereichsabschnitt besteht aus zwei Listen mit unterschiedlich langen Einträgen. Die Länge der Einträge wird von ihrem Inhalt bestimmt.

Der Definitionsbereichsabschnitt wird mit einer 64 Bit natürlichen Zahl eingeleitet, welche die Anzahl der Einträge für die Definitionsbereichsliste angibt. Danach folgt die Definitionsbereichsliste.

Auf diese wiederum folgt eine 64 Bit natürliche Zahl, welche die Anzahl der Elemente in der Speicherplatzliste (die Definitionsbereichen für Werte) angibt, und dann die Speicherplatzliste.

Dabei können eine oder beide Listen auch leer sein. Für leere Listen wird dann entsprechend nur die einleitende 64 Bit natürliche Zahl gespeichert, die dann 0 ist

Die Definitionsbereichsliste gibt die Definitionsbereiche der einzelnen Elemente an. Ist ein Wert (z. B. einer Variable) außerhalb dieses Definitionsbereichs, wird er auf einen Wert innerhalb des Definitionsbereichs gerundet. Werte außerhalb dieser Definitionsbereiche können also für das Element nicht auftreten.

Die Definitionsbereiche für Werte (Speicherplatzliste) enthalten auch Definitionsbereiche, nur gelten diese Definitionsbereiche nicht für Elemente, die Variablen enthalten. Die Definitionsbereiche für Werte bestimmen, wieviel Bits zum Speichern eines Elements, welches einen Wert enthält, benötigt wird. Die Definitionsbereiche für Werte sind also dann sinnvoll, wenn die Werte eines Elements nicht den vollen möglichen Definitionsbereich für das Element ausschöpfen. Wenn beispielsweise ein Unterobjekt nur Punkte enthält, deren Positionsvektoren nur Variablen und Ganzzahlwerte zwischen 0 und 10 enthalten, kann der Definitionsbereich für Werte für Positionsvektoren auf "integerB" mit 4 Bit gesetzt werden, auch wenn die Variablen der Positionsvektoren Werte über 100 einnehmen können.

Wenn beim Abspeichern ein Wert im Haupt-Fib-Objekt gefunden wird, der außerhalb des für sein Element festgelegten Definitionsbereichs liegt, wird der Definitionsbereich automatisch erweitert, so dass er auch den Wert umfasst. Wenn der Definitionsbereich ererbt wurde, wird im root-Element ein entsprechender neuer Definitionsbereich angelegt, der den Wert und den ererbten Definitionsbereich umfasst.

Die Definitionsbereiche für Werte werden beim Abspeichern eventuell erzeugt und optimiert, um den Speicherplatz für das Fib-Objekt möglichst gering zu halten. Ein Definitionsbereich für Werte wird beim Abspeichern erzeugt, wenn durch seine Erzeugung wahrscheinlich Speicherplatz eingespart wird.

Da zu einem Fib-Multimediaobjekt viele Definitionsbereiche gehören können, wird bei ihnen mehr auf den Speicherplatz geachtet. Um zukünftige Erweiterungen einfach zu machen, wird Wert auf Flexibilität gelegt.

Der Grund, für die Einführung von zentral (in den root-Elementen) festgelegten Definitionsbereichen ist, dass einerseits beim Abspeichern des Fib-Objekts möglichst wenig Speicherplatz für die Werte verbraucht wird, ohne die Belegungsmöglichkeiten für die Werte drastisch einzuschränken und andererseits vorab bestimmt werden kann, ob und wie das Multimediaobjekt angezeigt werden kann (z. B. wie es skaliert werden muss oder ob die Anzeige aller Werte unmöglich ist). Wenn beispielsweise ein Dimensionsdefinitionsbereich nur Ganzzahlwerte zwischen 0 und 50 einnimmt (z. B. die Horizontale in einem Bild), dann reichen 6 Bit zum Abspeichern der Werte für die Dimension aus. Bei größeren Bildern können einfach mehr Bits für die Werte der Dimension verwendet werden.

Jeder Eintrag besteht aus zwei Teilen:

1. Dem Namen des Elements, für den der Definitionsbereich gilt (siehe Abschnitt 25.2.4).
2. Die Angabe des Definitionsbereichs für das Element (siehe Abschnitt 25.2.4).

Elementennamen Der Elementenname kann nur aus der Angabe eines festen Elements oder aus einem festen Element und einem Parameter bestehen.

Das erste Bit (die Zählung beginnt bei 1) des festen Elements bestimmt dessen Länge:

- Ist es 0, ist der Namen 8 Bit lang. Mögliche Werte sind in Tabelle 10 zu sehen.
- Ist es 1, ist der Namen 64 Bit lang. Dies ist zur Zeit nur für eine zukünftige Nutzung eingeplant.

Das zweite und dritte Bit bestimmen die Länge des Parameters:

00 es gibt kein Parameter

01 Parameter mit einer Gesamtlänge von 8 Bit folgen

10 Parameter mit einer Gesamtlänge von 64 Bit folgen

11 Die auf das Elementnamenfeld folgende 16 Bit natürliche Zahl gibt die Parameterlänge in Byte an. Dabei werden die 16 Bit der Parameterlängenangabe nicht zur Parameterlänge mit eingerechnet.

Name	Wert Bit 4 bis 8	Beschreibung
dim	0000 1	Der Definitionsbereich für Positionsvektoren (siehe Abschnitt 12.2 auf Seite 24 und Abschnitt 25.3.2 auf Seite 130) bzw. die Dimensionen. Die Länge der Parameterliste ist variabel. Das zweite und dritte Bit ist also 11 und auf Bit 8 folgt eine 16 Bit natürliche Zahl L, welche die Länge der Parameterliste in Byte angibt. Der erste folgende Parameter ist eine 16 Bit natürliche Zahl und gibt die Anzahl der Dimension Dim an. Darauf folgen Dim weitere Parameter $Dimmap_1$ bis $Dimmap_{Dim}$ als natürliche Zahl mit jeweils der Länge $L_{Dimmap} = \lfloor((L-2)*8)/Dim\rfloor$ (also die noch übrigen Bits für die Parameter L durch die Anzahl der Dimensionen und das Ergebnis auf eine Ganzzahl abgerundet). Die Werte, welche die $Dimmap_i$ Parameter einnehmen können, sind in Tabelle 7 auf Seite 64 beschrieben. Die Länge der Parameterliste L ist so zu bestimmen, dass gerade genug Platz für alle Parameter ist.
subfunction	0001 0	Definitionsbereich für die Elemente von Unterfunktionen (siehe Abschnitt 12.7 auf Seite 35 und Abschnitt 25.3.7 auf Seite 134)
property	0010 0	Dies ist der Definitionsbereich eines Eigenschaftselements mit dem gegebenen Namen (siehe Abschnitt 12.3 auf Seite 25 und Abschnitt 25.3.3 auf Seite 130). Der Wert für den Namen bzw. den Eigenschaftstyp wird im Parameter übergeben. Mögliche Werte sind in Tabelle 2 auf Seite 30 zu sehen. Der Wert, der als Parameter angegeben wird, ist eine natürliche Zahl. Der Parameter ist nur so lang (z. B. 8 Bit), wie für die Darstellung des Wertes aus Tabelle 2 als natürliche Zahl benötigt wird.
inVar	0010 1	Dies ist der Definitionsbereich für die i'ten Eingabevariable (siehe Abschnitt 12.14 auf Seite 52 und Abschnitt 25.2.5 auf Seite 126). Der folgende Parameter ist eine natürliche Zahl und gibt die Nummer i der Eingabevariable an. (Die Zählung der Eingabevariablen eines root-Elements beginnt bei 1 .)
Namen von Elementen für Definitionsbereiche die beim Abspeichern (wenn nötig) erstellt werden		

Name	Wert Bit 4 bis 8	Beschreibung
area	0001 1	Dieser Typ ist für den Definitionsbereich für das Bereichselement (siehe Abschnitt 12.6 auf Seite 34 und Abschnitt 25.3.6 auf Seite 133). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 2 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Unterbereiche, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite und letzte Element ist der Definitionsbereich für die Vektoren für die Unterbereiche (B_1) und ist ein Definitionsbereich für Vektoren dessen zwei Elemente ganze Zahlen sind.
variable	1000 1	Werte, die benötigt werden, um Variablen zu kodieren. Der Definitionsbereich sollte die natürliche Zahlen von 0 bis maximale Anzahl der definierten Variablen in den Fib-Blättern im Haupt-Fib-Objekt beinhalten. Das Fib-Baum-Blatt im Haupt-Fib-Objekt, über das die meisten Variablen definiert werden, bzw. der Ast mit den meisten definierten Variablen, bestimmt also den Definitionsbereich. Dieser Eintrag wird beim Abspeichern erstellt.
comments	1001 0	Werte, die benötigt werden, um Kommentare zu kodieren (siehe Abschnitt 12.5 auf Seite 32, Abschnitt 25.2.9 auf Seite 127 und Abschnitt 25.3.5 auf Seite 133) . Der Definitionsbereich sollte die natürlichen Zahlen von 0 bis Anzahl der Kommentare im Haupt-Fib-Objekt beinhalten. Dieser Eintrag wird beim Abspeichern erstellt.
externObject	1100 0	Dies ist der Definitionsbereich für externe Objekte (siehe Abschnitt 12.9 auf Seite 43 und Abschnitt 25.3.9 auf Seite 138) im Haupt-Fib-Objekt. Der Definitionsbereich ist ein Vektor mit 4 Elementen. Die Vektorelemente dienen der Reihenfolge für den Identifier, die Anzahl der Eingabewerte, die Anzahl der Unterobjekte und die Anzahl der Ausgabevariablen. Alle Vektorelementdefinitionsbereiche, außer der für den Identifier, kommen aus den natürlichen Zahlen. Der Vektorelementdefinitionsbereich für den Identifier kommt aus den Ganzzahlen. Dieser Definitionsbereich wird normalerweise beim Abspeichern erstellt.

Name	Wert Bit 4 bis 8	Beschreibung
externObject-Input	1110 0	Dieser Typ ist für den Definitionsbereich für die Eingabewerte für externe Objekte (siehe Abschnitt 12.9 auf Seite 43 und Abschnitt 25.3.9 auf Seite 138). Der Definitionsbereich ist ein Vektordefinitionsbereich und wird normalerweise beim Abspeichern erstellt. Der folgende Parameter ist eine Ganzzahl und gibt den Identifier des externe Objekts an, für dessen Element der Definitionsbereich ist.
externSubobject	1100 1	Dieser Typ ist für den Definitionsbereich für die Eingabewerte für externe Unterobjekte (siehe Abschnitt 12.10 auf Seite 44 und Abschnitt 25.3.10 auf Seite 139). Der Definitionsbereich ist ein Vektordefinitionsbereich und wird normalerweise beim Abspeichern erstellt. Der folgende Parameter ist eine natürliche Zahl und gibt die Nummer des externe Unterobjekts an, für das der Definitionsbereich ist.
setElement	1101 0	Dieser Typ ist für den Definitionsbereich für das Set-Element (siehe Abschnitt 12.12 auf Seite 48 und Abschnitt 25.3.12 auf Seite 140). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 3 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Variablen und der zu setzenden Werte pro Satz, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich dient für die Anzahl (k) der Sätze mit zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{i.g}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind. Des weiteren kann als Parameter eine natürliche Zahl für den Definitionsbereichnummer $DomainNr$ angegeben werden. Wenn der Parameter fehlt, ist die die Definitionsbereichnummer $DomainNr$ gleich 0 .

Name	Wert Bit 4 bis 8	Beschreibung
matrixElement	1101 1	Dieser Typ ist für den Definitionsbereich für das Matrixelement (siehe Abschnitt 12.13 auf Seite 49 und Abschnitt 25.3.13 auf Seite 140). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 4 Elementen/Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich ist für die Anzahl (d) der Dimensionsvariablen, die Anzahl (i) der Wertevariablen und die zu setzenden Werte i pro Satz. Er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich ist für die Anzahl (k) der Sätze mit den zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte Element ist der Definitionsbereich für die Bereiche bzw. Start und Endwerte für die einzellenen Dimensionsvariablen, er ist ein Vektordefinitionsbereich mit zwei Elementen, welche jeweils aus den ganzen Zahlen kommen. Das vierte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{a.b}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind. Des weiteren kann als Parameter eine natürliche Zahl für den Definitionsbereichnummer $DomainNr$ angegeben werden. Wenn der Parameter fehlt, ist die die Definitionsbereichnummer $DomainNr$ gleich 0 .

Tabelle 10: Namen von festen 8-Bit-Elementen für Definitionsbereiche

Definitionsbereiche Der Definitionsbereich besteht aus einem Grunddefinitionsbereich und eventuell Parametern, die den Grunddefinitionsbereich weiter spezifizieren.

Das erste Bit (die Zählung beginnt bei 1) des Grunddefinitionsbereichsfeldes bestimmt dessen Länge:

- Ist es 0, ist der Grunddefinitionsbereichsfeld 8 Bit lang. Mögliche 8-Bit-Grunddefinitionsbereiche sind in Tabelle 11 zu sehen.
- Ist es 1, ist der Grunddefinitionsbereichsfeld 64 Bit lang. Dies ist zur Zeit nur für eine zukünftige Nutzung eingeplant.

Das zweite Bit gibt an, ob der Definitionsbereich skaliert wird:

- Ist es 0, wird der Definitionsbereich nicht skaliert.
- Ist es 1, wird der Definitionsbereich skaliert.

Wenn der Definitionsbereich skaliert wird, wird ein Skalierungsfaktor S angegeben. Dieser Faktor S ist eine Gleitkommazahl, mit dem die Werte des unskalierten Definitionsbereichs multipliziert werden, um (die Werte des/) den skalierten Definitionsbereich zu erhalten.

Der Skalierungsteil wird hinter den Definitionsbereichsangaben (inklusive Parametern) angegeben. Er besteht aus 3 Feldern: dem Längenfeld, dem Mantissenfeld und dem Exponentenfeld. Das erste Feld (das Längenfeld) ist 8 Bit lang, stellt eine natürliche Zahlen dar und gibt die Anzahl der Bytes für sowohl das Mantisse- als auch das Exponentenfeld an. Das Mantissenfeld (zweites Feld) S_M und das Exponentenfeld (drittes Feld) S_E sind jeweils Ganzzahlen. Der Skalierungsfaktor S ergibt sich dann zu $S = S_M * 2^{S_E}$.
Beispiel: Der Definitionsbereich sind 2 Bit natürliche Zahlen mit dem Skalierungsfaktor $F = 1*2^{-1} = 1/2$ (die Mantisse ist 1 und der Exponent ist -1, das Längenfeld ist 1, für ein Byte Skalierungsfaktorfeld). Die Bits des Skalierungsfaktors sind damit: 0000 0001; 0000 0001; 1111 1111. Möglich Werte für das Element zum Definitionsbereich sind: $\{ 0*1/2 = 0; 1*1/2 = 0,5; 2*1/2 = 1; 3*1/2 = 1,5 \}$. Ist nun im Element zum Definitionsbereich eine Zahl, die unskaliert 1 bedeuten würde, bedeutet sie mit Skalierung $0,5$. Die Bits des gesamten Definitionsbereichs sind: 0; 1; 000000; 00000010; 00000001; 00000001; 11111111 (Felder in dieser Reihenfolge: 8-Bit-Grunddefinitionsbereiche; skaliert; natNumberB; mit 2 Bit pro Element; Skalierungsfaktor: 1 Byte pro Feld, $S_M = 1$, $S_E = -1$) (in der Datei, erste Bit vorn: 01000000 01000000 10000000 10000000 11111111)

Die einzelnen Definitionsbereiche (inklusive Skalierungsfaktor) werden mit 0 auf ganzzahlige Byts aufgefüllt. Ist also eine Definitionsbereichsangabe 13 Bit lang, so werden die verbleibenden 3 Bit mit 0 gefüllt, so dass ein 16-Bit (=2 Byte) Definitionsbereichsfeld entsteht.

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
naturalNumberB	0000 00	Der Grunddefinitionsbereich sind die natürlichen Zahlen.	Der folgende 8-Bit-Parameter ist eine natürliche Zahl X, welche die Anzahl der Bits für Werte des Definitionsbereichs angibt. Der zugehörige Grunddefinitionsbereich ist dann $0 \ldots (2^X - 1)$.

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
integerB	0100 00	Der Grunddefinitionsbereich sind die Ganzzahlen.	Der folgende 8-Bit-Parameter ist eine natürliche Zahl X, welche die Anzahl der Bits für Werte des Definitionsbereichs angibt. Der zugehörige Grunddefinitionsbereich ist dann $-(2^{X-1}) \ldots (2^{X-1} - 1)$.
naturalNumber	0000 01	Der Grunddefinitionsbereich sind die natürlichen Zahlen.	Der folgende 64-Bit-Parameter ist eine natürliche Zahl X, welche die größte natürliche Zahl des Definitionsbereichs angibt. Der zugehörige Grunddefinitionsbereich ist dann $0 \ldots X$. Die Bits, die pro Wert des Definitionsbereichs benötigt werden, sind $\lceil \log_2(X + 1) \rceil$.
integer	0100 01	Der Grunddefinitionsbereich sind die Ganzzahlen.	Die folgenden zwei 64-Bit-Parameter sind zwei Ganzzahlen X und Y. Der erste Parameter X gibt die untere Grenze des Grunddefinitionsbereich an. Der zweite Parameter Y gibt die obere Grenze des Grunddefinitionsbereich an. Der zugehörige Grunddefinitionsbereich ist dann $Y \ldots X$. Die Bits, die pro Wert des Definitionsbereichs benötigt werden, sind $\lceil \log_2(Y - X + 1) \rceil$. Bei der Interpretation der Bits werden alle Werte W (W ist der Wert der Bits als natürliche Zahl) größer als X als negative Werte mit dem Wert $W-(X-Y+1)$ umgesetzt. Wenn die kleinste Zahl X größer als 0 oder die größte Zahl Y kleiner als 0 ist ist, wird kleinste Zahl X von alle Zahlen zum Speichern abgezogen ($W + X$).

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
integerValues	0100 10	Der Grunddefinitionsbereich sind die Ganzzahlen.	Der erste folgende 64-Bit-Parameter ist eine natürliche Zahl N, welche die Anzahl der möglichen Werte angibt. Der zweite folgende 8-Bit-Parameter B gibt die Anzahl der Bits pro nachfolgenden Wert an. Nach den ersten beiden Parametern folgen N Ganzzahlen mit je B Bits in der Zweierkomplementdarstellung. Diese N Ganzzahlen sind alle Werte, die Werte zu dem Grunddefinitionsbereich einnehmen können. Dabei wird der Wert W einer Zahl in einem Element zum Definitionsbereich auf die W'te Zahl in der Liste gemappt/interpretiert (die Zählung beginnt bei 0). Die Bits, die pro Wert W des Definitionsbereichs benötigt werden, sind $\lceil \log_2(N) \rceil$.
real	1000 00	Der Grunddefinitionsbereich sind Gleitkommazahlen. Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$.	Es folgen zwei Parameter. Der erste Parameter gibt den Definitionsbereich der Mantisse und der zweite des Exponenten an. Die Angabe der Definitionsbereiche ist so wie in dieser Tabelle beschrieben (ohne dass die Definitionsbereiche einzeln mit 0 aufgefüllt werden). Beide Definitionsbereiche müssen aus den Ganzzahlen kommen (also integer... oder naturalNumber...).

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
realValues	1000 01	Der Grunddefinitionsbereich sind Gleitkommazahlen. Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$.	Der erste folgende 64 Bit Parameter ist eine natürliche Zahl N, welche die Anzahl der möglichen Werte angibt. Es folgen 2 Parameter. Der erste Parameter gibt den Definitionsbereich der Mantisse und der zweite des Exponenten an. Beide Definitionsbereiche müssen aus den Ganzzahlen kommen (also integer... oder naturalNumber...) . Nach den ersten drei Parameter folgen N Gleitkommazahlen mit je $B = B_M + B_E$ Bits in der Gleitkommazahlendarstellung für die gegebenen Mantissa and Exponenten Definitionsbereiche. Dabei sind B_M die Bits pro Mantisse und B_E die Bits pro Exponent. Diese N Gleitkommazahlen sind alle Werte, die Werte zu dem Definitionsbereich einnehmen können. Dabei wird der Wert W, einer Zahl in einem Element zum Definitionsbereich, auf die W'te Zahl in der Liste gemappt/ interpretiert (die Zählung beginnt bei 0). Die Bits, die pro Wert des Definitionsbereichs benötigt werden, sind $\lceil \log_2(N) \rceil$.
vector	1100 00 und 1100 01	Der Grunddefinitionsbereich sind Vektoren.	Der folgende Parameter E gibt die Anzahl der Elemente des Vektors an. Der (Anzahl-)Parmeter ist bei der Einleitung "1100 00" 8 Bit lang und bei der Einleitung "1100 01" 64 Bit lang. Auf diesen Parameter folgt eine Liste mit E Definitionsbereichen, so wie sie in diesem Abschnitt definiert sind. Alle Werte des Definitionsbereichs haben dann die Form $(D_1, \dots, D_E)$, wobei D_i ein Wert aus dem i'ten Definitionsbereich der Liste ist.

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
vectorValues	1100 10 und 1100 11	Der Grunddefinitionsbereich sind Vektoren.	Der erste folgende Parameter E gibt die Anzahl der Elemente des Vektors an. Der zweite Parameter N die Anzahl der möglichen Vektoren. Die (Anzahl-)Parmeter sind bei der Einleitung “1100 00” 8 Bit lang und bei der Einleitung “1100 01” 64 Bit lang. Auf den zweiten Parameter folgt eine Liste mit E Definitionsbereichen, so wie sie in diesem Abschnitt definiert sind. Darauf folgt eine Liste mit N Vektoren, so wie im Abschnitt 25.3.1 auf Seite 129 beschreiben. Die Definitionsbereiche der Vektoren sind die angegebenen Definitionsbereich der vorhergehenden Liste. Es können auch Variablen in den abgespeicherten Vektoren vorkommen. Der Definitionsbereich für Variablen in der Liste hat 0 Bits. Für eine Variable im Vektor wird also nur die einleitende 1 geschrieben. Ein Wert W zum Definitionsbereich in einem Fib-Element wird dann als W’ter Vektor der zweiten Liste interpretiert. (Die Zählung beginnt bei 0 .) Enthält der angegebene Vektor Variablen, folgen auf den Wert W direkt die Variablenbezeichner der Variablen, so wie sie über dem aktuellen Fib-Element definiert wurde. Die Anzahl der Bits der Variable wird durch den entsprechenden Definitionsbereich für Variablen “variable” bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
vectorOpenEnd	1110 00 und 1110 01	Der Grunddefinitionsbereich sind Vektoren.	Der folgende Parameter E gibt die mindest Anzahl der Elemente des Vektors an. Der (Anzahl-)Parmeter ist bei der Einleitung “1100 00” 8 Bit lang und bei der Einleitung “1100 01” 64 Bit lang. Auf diesen Parameter folgt eine Liste mit E Definitionsbereichen, so wie sie in diesem Abschnitt definiert sind. Alle Werte des Definitionsbereichs haben dann die Form $(D_1, \ldots, D_E, \ldots, D_E)$, wobei D_i ein Wert aus dem i’ten Definitionsbereich der Liste ist. Dieser Definitionsbereich dient für Elemente, die Vektoren unterschiedlicher Größe enthalten können. Die Anzahl der Elemente eines Vektors zu diesem Definitionsbereich wird durch das Element bestimmt, welches den Vektor enthält.

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
domainReference	1111 00	Ein Verweis auf den (Unter-)Definitionsbereich eines anderen Elements. Als Definitionsbereich wird der Definitionsbereich des Elements mit dem gegebenen Definitionsbereichnamen $Name$ genommen.	Als erster Parameter folgt der im komprimierte Dateiformat kodierte $Name$ des Elements von dem der Definitionsbereich verwendet wird (siehe Abschnitt 25.2.4 auf Seite 114, ohne auffüllen auf ein ganzes Byte). Danach folgt die Angabe der $Element$ Parameter, für die Auswahl eines Unterdefinitionsbereichs. Als erstes folgt dafür ein $Element$-Einleitungsbit, welches angibt ob ein weiterer $Element$ Parameter folgen. Ist es 0 folgt kein $Element$ Parameter, ist es 1 folgt ein $Element$ Parameter. Wenn es 1 ist folgt auf dieses erste $Element$-Einleitungsbit eine 1 Byte (8-Bit) lange natürliche Zahl $Bits$, die Angibt, wieviele Bits pro $Element$ Parameter verwendet werden. Danach folgt der erste $Element$ Parameter. Nach jedem $Element$ Parameter (abgespeichert mit dem Definitionsbereich $naturalNumberB(Bits)$) folgt wieder ein $Element$-Einleitungsbit und danach eventuell der nächste $Element$ Parameter usw. . Beispiel $matrix.3.1$: zwei Bits werden benötigt um die $Element$ Parameter abzuspeichern; die Bits sind (erstes Bit vorn): 0 0 001111 0 00 11011 1 00000010 11 1 10 0 (In ihrer Reihenfolge stehen die Felder für: Definitionsbereichsfeld 8 Bit lang; nicht skaliert; domainReference; 8 Bit Elementname; keine Parameter; matrixElement; $Element$ Parameter folgt; mit je 2 Bit; 3'ter Unterdefinitionsbereich; $Element$ Parameter folgt; erster Unterdefinitionsbereich; kein $Element$ Parameter folgt)

Name	Wert Bits 3 bis 8	Beschreibung	Parameter
defaultDomain	1111 01	Der angegebene Definitionsbereich wird nur verwendet, wenn es für das entsprechende Element bisher keinen anderen Definitionsbereich gibt.	Als Parameter folgt ein Definitionsbereich wie in dieser Tabelle beschrieben. Der angegeben Definitionsbereich wird nur verwendet, wenn es für das entsprechende Element bisher keinen anderen Definitionsbereich gibt.

Tabelle 11: 8 Bit Parameter für Definitionsbereiche

25.2.5 Eingabevariablen

In komprimierten Fib-Objekten werden Variablen durchnummeriert und als natürliche Zahlen dargestellt. Dabei sind die ersten n Variablenwerte im Fib-Objekten für die Eingabenvariablen reserviert. Deshalb ist für die Variablen nur die Anzahl der Eingabevariablen V_E an dieser Stelle interessant. Dafür wird eine 64 Bit natürliche Zahl an dieser Stelle abgespeichert, welche die Anzahl der Eingabevariablen enthält. Nach dem Feld für die Anzahl der Eingabevariablen (V_E) folgen V_E Werte für die Standardbelegungen S_i der Eingabevariable $inVar_i$ (in der jeweiligen Reihenfolge). Die Anzahl der Bits und die Kodierung des Standardwerts S_i folgt aus dem Definitionsbereich der Eingabevariable $inVar_i$.

Das Eingabevariablenfeld wird mit dem 0 Bit auf volle Bytes aufgefüllt.

25.2.6 Haupt-Fib-Objekt

An dieser Stelle kommen die Daten des Haupt-Fib-Objekts in Form seiner Elemente und deren Parameter, so wie es weiter unten im Abschnitt 25.3 auf Seite 129 beschrieben wird.

Das Feld für das Haupt-Fib-Objekts wird mit dem 0 Bit auf volle Bytes aufgefüllt.

25.2.7 Weitere root-Objekte

Hier folgt eine Liste mit weiteren root-Objekten und ihren Identifizierern. Eingeleitet wird die Liste mit einer 64 Bit natürlichen Zahl N, welche die Anzahl der Elemente der Liste angibt. Darauf folgt eine 8 Bit natürliche Zahl B, welche die Bytes pro Identifizierer angibt.

Darauf folgt die Liste von N Paare mit Identifizierer und root-Element. Der Identifizierer in jedem Paar ist eine Ganzzahl (im root-Element immer größergleich 0, in der Datenbank kleiner 0). Er kommt in den Listenelementen zuerst und ist B Bytes lang. Darauf folgt das root-Element, so wie es hier im Abschnitt 25.2 (ab Seite 106) beschrieben wird. Jedes Paare wird mit dem 0 Bit auf volle Bytes aufgefüllt.

25.2.8 Identifier von verwendeten Datenbankobjekten

An dieser Stelle steht eine Liste aller Identifier von Fib-Datenbankobjekten, welche im Haupt-Fib-Objekt oder auch in Unter-root-Objekten verwendet werden. Diese Angaben der Identifiern ist optional. Wenn Identifier vorhanden sind, kann von Vornherein geprüft werden, ob alle externen Fib-Objekte aus der Datenbank, welche benötigt werden, existieren oder ob es wahrscheinlich zu Anzeigefehlern kommt, da Fib-Datenbankobjekte fehlen. Ob die Identifier von Fib-Datenbankobjekten, die in einem Haupt-Fib-Objekt eines root-Elements verwendet werden, in diesem root-Element angegeben werden oder in einem höheren root-Element, hängt von verschiedenen Gesichtspunkten ab. Für die Veränderung von Fib-Objekten, ist es vorteilhaft Identifier von Fib-Datenbankobjekten in einem root-Element, welches möglichst nahe am Verwendungsort liegt, anzugeben. Aus Platzgründen kann es aber sinnvoll sein, Identifier von Fib-Datenbankobjekten in möglichst wenigen root-Elementen zusammenzufassen.

Eingeleitet wird die Liste mit einer 64 Bit natürlichen Zahl N, welche die Anzahl der Elemente der Liste angibt. Darauf folgt eine 8 Bit natürliche Zahl B, welche die Bits pro Identifizierer angibt. Darauf folgt die Liste von N je B Bits langen ganzzahl Identifizierern.

Das Feld für die Identifizierer von verwendeten Datenbankobjekten wird mit dem 0 Bit auf volle Bytes aufgefüllt.

25.2.9 Optionaler Teil

Der optionale Teil ist der letzte Teil eines root-Elements. Er sollte für die Darstellung des Multimediaobjekts keine wichtige Informationen enthalten, so dass er beim Abspeichern ganz weggelassen werden kann. Auf diese Weise kann Speicherplatz gespart werden.

Die Länge des optionale Teil ergibt sich aus der Differenz zwischen seinem Anfangsbyte und dem Ende des root-Elements (zum Offset siehe Überabschnitt 25.2 ab Seite 106).

Eingeleitet wird der Optionale Teil durch eine 16 Bit natürliche Zahl C, welche die Kompressionsart des optionale Teils angibt. Mögliche Werte für die Kompressionsart C sind in Tabelle 12 aufgeführt. Der restliche optionale Teil wird dann vollständig mit dem angegebenen Verfahren komprimiert.

Der dekomprimierte optionale Teil besteht aus einer Liste von Key, Value-Paaren. Sie wird eingeleitet durch eine 64 Bit natürliche Zahl N, welche die Anzahl

Wert C	Beschreibung
0	keine Komprimierung
1	Die Daten sind im zlib Format komprimiert. Dieser ist ein Warper (dokumentiert im RFC 1950) für einen Deflate Datenstrom (verlustlosen Datenkompression, dokumentiert in RFC 1951).

Tabelle 12: Komprimierungsarten

der Elemente in der Liste angibt. Als zweiter Parameter folgt eine 16 Bit natürliche Zahl, welche die Kodierungsart bestimmt. Mögliche Werte für die Kodierung sind in Tabelle 13 aufgeführt. Dabei sollte UTF-8 als Standardkodierung angesehen werden und andere Kodierungen nur gewählt werden, wenn UTF-8 nicht mehr ausreichend ist. Danach folgt die Liste mit den Paaren/ Elementen.

Jedes Listenelement (-Paar) enthält zwei nullterminierte Zeichenketten in der angegeben Kodierung. Die erste Zeichenkette ist dabei der Schlüssel/ Key des Elements und die zweite der Wert/ Value.

Name	Wert	Beschreibung
UTF-8	0	8-Bit Unicode Transformation Format
UTF-16	1	Universal Multiple-Octet Coded Character Set (UCS) Transformation Format for 16 Planes of Group 00
UTF-32	2	Unicode kodiert ein Zeichen mit 32 Bit

Tabelle 13: Kodierungsarten für Zeichenketten

Aus dem Haupt-Fib-Objekt werden alle Texte aus den Anmerkungen (siehe Abschnitt 25.3.5 auf Seite 133) in den optionalen Teil verlagert. Die Anmerkungen enthalten auch Schlüssel (Key), Wert ($Value$)-Paare. Jedem dieser Schlüssel, Wert Paare aus den Anmerkungen wird eine natürliche Zahl K zugeordnet (beginnend mit 0). Im optionalen Teil wird dann für das ausgelagerte Paar ein eigener Listeneintrag generiert. Dabei ist der Schlüssel des Listeneintrags "@" gefolgt von der der Nummer der Anmerkung A, dann folgt wieder ein "@" und danach der ursprünglich Schlüssel (Key) der Anmerkung. Der Wert des generierten Listeneintrag ist gleich dem Wert aus der Anmerkung. Wenn also das Schlüssel, Wert Paar in der A'ten Anmerkung $(Key, Value)$ ist, ist der Eintrag im optionalen Teil dafür $(@A@Key, Value)$.

Beispiel: Die 2. Anmerkung ist: "c("autor","ich", Obj)", der daraus generierte Listeneintrag im optionalen Teil ist: "("@2@autor","ich")".

Ausgelagerte Texte sollten am Ende des optionalen Teils stehen.

25.3 Fib-Elemente

Alle Fib-Elemente, außer dem root-Element, werden durch ein (mindestens 4 Bit) Bitfeld eingeleitet. Vektoren werden direkt in den Fib-Elementen eingebunden, ohne eigene Einleitung.

Die Einleitung 0000 ist reserviert, damit mit 0 aufgefüllte Bereiche nicht als Fib-Element interpretiert werden können. Die Einleitung 1111 ist für spätere Benutzung reserviert und kündigt an, dass die Einleitung 16 Bit lang ist und nicht 4 Bit.

Alle Fib-Elemente und Vektorelemente folgen direkt hintereinander. Wenn Stellen mit dem 0 Bit aufgefüllt werden, wird dies im Nachfolgenden explizit angeführt.

25.3.1 Vektoren

Vektoren bestehen aus mehreren Elementen, wobei jedes Element entweder eine Wert oder eine Variable sein kann. Die Anzahl der Elemente des Vektors ist entweder Fib-Element spezifisch oder wird durch einen zugehörigen Definitionsbereich bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).
Für jedes Element gibt es zwei Felder:

- ein 1 Bitfeld welches bestimmt, ob es sich bei dem Element um einen Wert (Bit ist 0) oder um eine Variable (Bit ist 1) handelt
- das Feld mit dem Wert für das Element (dem wahren Wert oder einer Zahl für die Variable)

Ist das Element ein Wert, folgt auf das 1 Bitfeld mit 0 direkt ein Wert. Die Anzahl der Bits des Werts wird durch den entsprechenden Definitionsbereich bestimmt. Der Wert entstammt dem unskalierten Definitionsbereich und wird erst nach dem Laden skaliert.

Ist das Element eine Variable, folgt auf das 1 Bitfeld mit 1 direkt ein Variablenbezeichner (dies ist die gleiche natürliche Zahl wie bei der Variablendefinition) einer Variable, die über dem aktuellen Fib-Element definiert wurde. Die Anzahl der Bits der Variable wird durch den entsprechenden Definitionsbereich für Variablen "variable" bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Die im jeweiligen Fib-Objekt-Zweig am weitesten oben definierten Variable hat den Wert bzw. Variablenbezeichner 1 . Alle anderen Variablen haben den Wert $i+1$, wobei i der Wert der als nächste über der Variable definierten Variable ist. Es werden also in den jeweiligen Fib-Objekt-Zweigen die Variablen von oben (ausgehend von der Wurzel) nach unten (zu den Blättern) durchnummeriert, wobei mit 1 begonnen wird.

25.3.2 Punkte

Einleitung normaler Punkt: 0001
Fünf Bit Einleitung für einen Punkt mit leerem Positionsvektor ($point(())$): 00010
Fünf Bit Einleitung für einen Punkt ohne Positionsvektor ($point()$): 10010

Zur Beschreibung des Punktelements siehe Abschnitt 12.2 auf Seite 24 .

Normale Punkte enthalten einen Positionsvektor (siehe 25.3.1). Die Anzahl der Vektorelemente wird durch die Anzahl der Dimensionen bzw. Elemente im Definitionsbereich der Dimension ("dim") bestimmt. Die Bits pro Positionsvektorelement wird durch das jeweile Element des Definitionsbereich der Dimension ("dim") oder durch den Definitionsbereich für Variablen ("variable") bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Die einzelnen Vektorelemente folgen direkt auf die Einleitung des Punktes.

Punkt mit leeren oder ohne Positionsvektor enthalten keine Bits für einen Positionsvektor. Bei diesen Punkten gibt es also nur die jeweilige 5 Bit Einleitung.

Beispiel: Ein Punkt wird kodiert mit den Positionswerten (2; 10). Das Multimediaobjekt hat 2 Dimensionen. Für beide Dimensionen werden 5 Bit benötigt, es sind natürliche Zahlen.

- Einleitung Punkt: 0001
- Erstes Positionsvektorelement (führende 0 für: "Vektorelement ist ein Wert")
  2 : 0 00010
- Zweites Positionsvektorelement (führende 0 für: "Vektorelement ist ein Wert")
  10 : 0 01010
- Kodiertes Punktobjekt (erstes Bit vorn): 1000010000010100

25.3.3 Eigenschaft

Einleitung: 0011

Zur Beschreibung des Eigenschaftselements siehe Abschnitt 12.3 auf Seite 25 .

Der Eigenschaftsvektortyp wird durch eine natürliche Zahl bestimmt die direkt auf die Einleitung folgt. In den gültigen Definitionsbereichen werden die mögliche Eigenschaften aufgelistet (siehe Abschnitt 12.14.2 auf Seite 54, Tabelle 2 auf Seite 30 und Tabelle 5 auf Seite 61). Die Zahl für den Eigenschaftsvektortyp ist die Nummer der Eigenschaft in der gültigen Definitionsbereichsliste. Gezählt werden dabei nur Eigenschaftsdefinitionsbereiche und angefangen wird mit der Zählung bei 0.

In der gültigen Definitionsbereichsliste werden zuerst die Definitionsbereiche für Werte (siehe Abschnitt 12.14.2 auf Seite 65) aufgelistet und danach Definitionsbereiche für Elemente (siehe Abschnitt 12.14.2 auf Seite 63) . Ererbte Eigenschaftsdefinitionsbereich, die nicht überschrieben werden, werden gezählt, als ob sie hinter den Definitionsbereichen des aktuellen (bzw. nächsten) root-Elements stehen. (Um so näher das vererbende root-Element dem erbenden ist, um so kleiner sind die Zahlen der vererbten Eigenschaftsdefinitionsbereiche [jeweils für Werte oder Elemente Definitionsbereiche getrennt] .)

Für Eigenschaftstypen, welche im Fib-Objekt vorhanden sind, aber für die kein Definitionsbereich existiert, muß ein Definitionsbereich (z. B. der Standarddefinitionsbereich) in den Wertedefinitionsbereichen eingefügt werden. Nur so können die Eigenschaftsvektortypen durchnummeriert werden.

Die Anzahl Bits, welche für die Zahl des Eigenschaftsvektortyp benötigt werden, ergibt sich aus der Anzahl der definierten (inklusive ererbten) Eigenschaftsdefinitionsbereiche WE insgesamt, sie ist $\lceil \log_2(WE) \rceil$. Werden beispielsweise nur die Eigenschaftsdefinitionsbereiche `colorRGB`, `layer` und `sound` definiert. Dann werden $\lceil \log_2(3) \rceil = 2$ Bit zum Abspeichern des Eigenschaftsvektortyp benötigt und der Eigenschaftsvektortyp `layer` hat den Wert 01.

Auf dem Eigenschaftsvektortyp folgen die Elemente des Eigenschaftsvektors (siehe Abschnitt 25.3.1 auf Seite 129). Wobei die Anzahl und Bits durch die jeweiligen Definitionsbereichsdefinition bestimmt wird (siehe Abschnitt 25.2.4 auf Seite 113).

Am Ende folgt das Unterobjekt der Eigenschaft.

Bei den Eigenschaften gibt es noch zwei besondere Eigenschaften, die Einfluss auf die Decodierung des Fib-Objekts haben. Diese Eigenschaften sind “checksum” und “boundSize”.

Checksummen Die Checksummeneigenschaft dient zum Finden von fehlerhaften Fib-Objekten und zum eventuellen korrigieren der Fehler.

Ihre Bezeichnung ist “checksum”. Sie hat 3 Parameter. Der erste Parameter A gibt die Art der Checksumme an. Der zweite parameter B gibt an alle wieviel Bits eine Checksumme generiert werden soll und der dritte Parameter C wieviel Bits die Checksumme haben soll.

Checksummen der Art 1 Ist der erste Parameter A gleich 1 wird die Checksumme nach der in diesem Abschnitt beschriebenen Art generiert.

Die Checksumme steht jeweils vor dem zu ihr gehörenden Block.

Die Checksummeneigenschaft gilt für das gesamte Fib-Objekt bzw. Fib-Zweig das im Eigenschaftselement vorhanden ist (dessen gesamtes Unterobjekt), ausgenommen Bereiche die eine separate Checksummeneigenschaft haben. Beim Laden des Fib-Objekts werden immer Blöcke von $C + B$ Bits geladen, wobei Blöcke die am Ende des Checksummenbereichs liegen mit 0 aufgefüllt werden. Vom Fib-Objekt wird also jeweils nur soviel gelesen wie noch zum Checksummenbereich gehört und der Rest des Blocks mit 0 aufgefüllt.

Wird der Checksummenbereich Cb_1 durch ein anderen Checksummenbereich CB_2 unterbrochen, wird nach dem Checksummenbereich CB_2 ein neuer Block für den Checksummenbereich Cb_1 angefangen und eingelesen.

Beim Abspeichern wird das gesamte Unterobjekt des Eigenschaftselement der Checksumme zuerst in komprimierte Form gebracht, so wie in diesem Abschnitt 25 beschrieben. Aus dem entstandende Bitfeld werden alle Bereiche ausgeschnitten die nicht mehr zu der Checksummeneigenschaft gehören, da sie zu einer anderen Checksummeneigenschaften gehören, diese werden dann separat behandelt. Dadurch entstehen aus dem komprimierten Unterobjekt der Checksummeneigenschaft mehrere Bitfelder Bf_i für die Checksummeneigenschaft. Diese Bitfelder Bf_i werden in Blöcke von je B Bits unterteilt, wobei der jeweils letzte Block eines Bitfeldes Bf_i mit dem 0 Bit auf B Bits aufgefüllt wird. Für die Blöcke wird dann jeweils die Checksumme von C Bits berechnet. Die Blöcke werden danach in ihrer ursprünglichen Reihenfolge wieder zu Bitfelder entsprechend zu den Bf_i Blöcken zusammengefügt, wobei jeweils zuerst das Checksummenfeld des Blocks und dann der zugehörige Block kommt. Diese Bitfelder werden anschließend in ihrer ursprünglichen Reihenfolge mit den zu anderen Checksummeneigenschaften Bereichen, die separat behandelt wurden, zu einem Bitfeld für das gesamte Unterobjekt zusammengefügt.

Die Checksumme gilt nicht nur zur Fehlererkennung, sonder nach Möglichkeit (wenn genug Checksummenbits vorhanden sind) zu Fehlerkorrektur.

Größe des Fib-Objekts Mit der Eigenschaft "boundSize" wird angegeben, dass für ein Fib-Objekt die Größe in Bit angegeben werden sollen. Obwohl diese Eigenschaft im erstellten Fib-Objekt kein Element im Vektor besitzt, wird beim Abspeichern des Vektors ein Element erzeugt, für das auch ein Definitionsbereich im root-Element erstellt wird (siehe Abschnitt 25.2.4 auf Seite 113).

Das Vektorelement im komprimierten Format enthält die Anzahl der Bits, die das enthaltende Fib-Objekt lang ist (inklusive eventueller Checksummen). Sollte das Fib-Objekt unreparabel Fehlerhaft sein, so kann auf diese Weise dennoch der Anfang des folgenden Fib-Objekts gefunden werden und dieses geladen werden. Ohne die Angabe der Länge des Fib-Objekts Obj_1 kann wahrscheinlich nicht der Anfang des im Datenstrom folgenden Fib-Objekts Obj_2 bestimmt werden, da unbekannt ist, wie lang die Elemente von Obj_1 sind. Wenn Obj_1 unreparabel Fehlerhaft ist, aber der Anfangspunkt bzw. das Anfangsbit des folgenden Fib-Objekts Obj_2 bekannt ist, kann dann zwar das Obj_1 nicht dargestellt werden, aber dies hat dann keine Auswirkungen auf Multimediainformationen außerhalb von Obj_1, so dass sich der Fehler auf Obj_1 beschränkt.

Die Eigenschaft "boundSize" ist sinvollerweise als erstes/ oberstes Fib-Element von Unterobjekten in Listen oder root-Elementen einzusetzen.

25.3.4 Listenelement

Mögliche Einleitungen:

- 0100: Einleitung für zwei Unterobjekte
- 0101: weniger als 256 Unterobjekte

- 0110: weniger als 2^{64} Unterobjekte

Zur Beschreibung des Listenelements siehe Abschnitt 12.4 auf Seite 31 .

Das Listenelement gibt an, dass im Nachfolgenden mehrere Fib-Objekte hintereinander kommen. Wobei direkt auf die Einleitung des Listenelements ein natürliche Zahl folgt, welche die Anzahl der folgenden Fib-Unterobjekte angibt. Die Länge des Feldes, für die Anzahl der Fib-Unterobjekte, hängt von der Einleitung des Listenelements ab.

Folgende Längen für das Anzahlfeld gibt es:

- Einleitung Listenelement 0100: Das Feld für die Anzahl der Fib-Unterobjekte ist 0 Bit lang, bzw. wird weggelassen.
- Einleitung Listenelement 0101: Das Feld für die Anzahl der Fib-Unterobjekte ist 8 Bit lang.
- Einleitung Listenelement 0110: Das Feld für die Anzahl der Fib-Unterobjekte ist 64 Bit lang.

Auf die Angabe der Anzahl Fib-Unterobjekte folgen die Fib-Unterobjekte. Wobei das erste folgende Fib-Unterobjekte, das erste Unterobjekt des Listenelements ist, darauf folgt dann das zweite Unterobjekt des Listenelements usw. .

Weitere Angaben oder Informationen werden zum Listenelement nicht benötigt, da die Länge der Unterobjekte aus den Unterobjekten selbst hervorgeht.

25.3.5 Anmerkungselement

Einleitung: 0111

Zur Beschreibung des Anmerkungselements siehe Abschnitt 12.5 auf Seite 32 .

Die Anmerkungselemente enthalten nach der Einleitung eine natürliche Zahl N, welche die Nummer der Anmerkung angibt. Der Definitionsbereich "comments" dieser Zahl wird durch das root-Element bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Darauf folgt das Unterobjekt des Anmerkungselements.

Im optionalen Teil des zugehörigen root-Elements wird für jede Anmerkung ein Eintrag erstellt. Mehr Details sind im Abschnitt 25.2.9 auf Seite 128 nachzulesen.

25.3.6 Bereiche

Einleitungen: 1000

Zur Beschreibung des Bereichselements siehe Abschnitt 12.6 auf Seite 34 .

Direkt auf die Einleitungen folgt eine natürliche Zahl, welche die Anzahl der Teilbereiche angibt. Die Bits für diese Zahl ergeben sich aus dem ersten Element /Unterdefinitionsbereich des Vektordefinitionsbereichs für “area”.

Dannach folgen die Teilbereiche. Die Teilbereiche sind Vektoren, welche aus zwei Elementen bestehen (zu Vektoren siehe Abschnitt 25.3.1 auf Seite 129). Die entsprechenden Definitionsbereiche für die Elemente der Teilbereichsvektoren sind durch den zweiten Unterdefinitionsbereiche des Definitionsbereichs “area” für die Werte oder den Definitionsbereich “variable” für die Variablen bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Am Ende folgt das Unterobjekt des Bereichsobjekts.

25.3.7 Funktionen

Einleitung: 1011

Zur Beschreibung des Funktionselements siehe Abschnitt 12.7 auf Seite 35 .

Nach der Einleitung folgt die Unterfunktion der Funktion.

Am Ende folgt das Unterobjekt des Funktionsobjekts.

Unterfunktionen

Unterfunktionen werden durch 2 Bit für Art der Unterfunktion eingeleitet:

- 00: Wert
- 01: Variable
- 10: Unterfunktion einstellig
- 11: Unterfunktion zweistellig

Ist die Unterfunktion ein Wert (Einleitung: 00), folgt auf die Einleitung 00 direkt ein Wert. Die Anzahl der Bits des Werts wird durch den entsprechenden Definitionsbereich für Unterfunktionen “underfunction” bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Ist die Unterfunktion eine Variable (Einleitung: 01), folgt auf die Einleitung 01 direkt ein Variablenbezeichner, einer Variable die über dem Funktionselement definiert wurde. Die Anzahl der Bits des Variablenbezeichners wird durch den entsprechenden Definitionsbereich für Variablen “variable” bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Ist die Unterfunktion eine einstellig Unterfunktion (Einleitung: 10), folgt auf die Einleitung 10 direkt ein (mindestens 2 Bit) Bitfeld, welches den Typ der Unterfunktion angibt. Nach dem Bitfeld folgt direkt eine weitere Unterfunktion (Eingeleitet durch die 2 Bit für deren Art), welche die Unterfunktion der einstellig Unterfunktion ist.

Werte für den Typ von einstellig Unterfunktion:

- 00: Absolutwert
- 01: Sinusfunktion
- 10: Einleitung für seltenere einstellige Funktionen. Die folgenden 2 Bit geben den Typ der einstelligen Funktion an:
  - 00 10: Logarithmus
  - 01 10: Arkussinus
  - 10 10: Runden
  - 11 10: Frei für zukünftige Belegungen.
- 11: Einleitung für seltenere einstellige Funktionen. Die folgenden 6 Bit geben den Typ der einstelligen Funktion an:
  - **** ** 11: Frei für zukünftige Belegungen.

Ist die Unterfunktion eine zweistellig Unterfunktion (Einleitung: 11), folgt auf die Einleitung 11 direkt ein (mindestens 3 Bit) Bitfeld welches den Typ der Unterfunktion angibt. Nach dem Bitfeld folgt direkt zwei weitere Unterfunktion (Eingeleitet jeweils durch die 2 Bit für deren Art), welche die Unterfunktionen der zweistellig Unterfunktion sind.

Werte für den Typ von zweistellig Unterfunktion:

- 000: Addition
- 001: Subtraktion
- 010: Multiplikation
- 011: Division
- 100: Exponent
- 101: Minimum
- 110: Maximum
- 111: Einleitung für seltenere einstellige Funktionen. Die folgenden 5 Bit geben den Typ der zweistelligen Funktion an:
  - 0000 0 111: If-Funktion (vor den zwei weitere Unterfunktion wird zuerst die Bedingung der if-Unterfunktion ausgegeben)
  - 0000 1 111: Delay
  - 0001 0 111: Modulo
  - **** * 111: Frei für zukünftige Belegungen.

Beispiel:

- Beispielobjekt: fun(x_3 , mult(sin(5), x_1), ...)
- Bits für "fun" (Einleitung): 1011
- Bits für "mult" (Einleitungen für "Unterfunktion zweistellig" und "mult"): 11 010
- Bits für "sin" (Einleitungen für "Unterfunktion einstellig" und "sin"): 10 01
- Bits für den Wert 5 (Einleitungen für "Wert" und 5 als 4 Bit natürliche Zahl mit dem Werte Definitionsbereich "underfunction" als "naturalNumberB(4)", also eine 4 Bit Zahl): 00 0101
- Bits für die Variable x_1 (die erste Variable im Teilobjektarm, Einleitungen für "Variable" und der Variablenwert): 01 0001
- Bits für das ganze Funktionselement: 1011 11 010 10 01 00 0101 01 0001 ... (=Bits für das Unterobjekt)
- Bits für das ganze Funktionselement (erste bit vorn): 1101 1101 0011 0001 0101 0100 0 ... (=Bits für das Unterobjekt)

25.3.8 If-Bedingung

Einleitung: 1100

Zur Beschreibung des If-Elements siehe Abschnitt 12.8 auf Seite 41 .

Nach der Einleitung folgt die Bedingung des If-Elements.

Am Ende folgen die beiden Unterobjekte des If-Objekts. Wenn die Bedingung Wahr ist, wird nur das erste Unterobjekt ausgewertet, wenn die Bedingung Falsch ist, nur das zweite Unterobjekt.

Bedingungen Bedingungen werden durch einen 4 Bit Wert für die Art der Bedingungen eingeleitet. Nach der Einleitung der Bedingung folgen als Parameter entweder 0 bis 2 Unterbedingungen oder zwei Unterfunktionen (siehe Abschnitt 25.3.7 auf Seite 134).

Die möglichen Bedingungen werden in Tabelle 14 aufgeführt.

Beispiel:

- Beispielobjekt: if(and(true, gr(x_1, 3)), ...)
- Bits für "if" (Einleitung): 1101
- Bits für "and" (Einleitungen für "and"): 0011

Name	Einleitung	Beschreibung	Parameter
false	0000	Die Bedingung ist Falsch.	keine
true	1111	Die Bedingung ist Wahr.	keine
not	0001	Die Bedingung ist genau dann Wahr, wenn die Unterbedingung Falsch ist, sonst ist sie Falsch.	eine Unterbedingung
or	0010	Die Bedingung ist genau dann Wahr, wenn mindestens eine der beiden Unterbedingungen Wahr ist, sonst ist sie Falsch.	zwei Unterbedingung
and	0011	Die Bedingung ist genau dann Wahr, wenn die beiden Unterbedingungen Wahr sind, sonst ist sie Falsch.	zwei Unterbedingung
xor	0100	Die Bedingung ist genau dann Wahr, wenn genau eine der beiden Unterbedingungen Wahr sind, sonst ist sie Falsch.	zwei Unterbedingung
eqInt	1000	Vergleich zweier auf Ganzzahlen gerundeter Zahlen auf Gleichheit.	zwei Unterfunktionen, so wie in Abschnitt 25.3.7 auf Seite 134 beschrieben
lo	1001	Vergleich, ob die erste Zahl kleiner als die zweite ist.	zwei Unterfunktionen, so wie in Abschnitt 25.3.7 auf Seite 134 beschrieben
gr	1010	Vergleich, ob die erste Zahl größer als die zweite ist.	zwei Unterfunktionen, so wie in Abschnitt 25.3.7 auf Seite 134 beschrieben
	0101 bis 0111 und 1011 bis 1110	Frei für zukünftige Bedingungen.	

Tabelle 14: Einleitungen für Bedingungen

- Bits für “true” (Einleitungen für “true”): 1111
- Bits für “gr” (Einleitungen für “gr”): 0111
- Bits für die Variable x_1 (die erste Variable im Teilobjektarm, Einleitungen für die Unterfunktion “Variable” und der Variablenwert; der Variable Definitionsbereich “variable” ist “naturalNumberB(4)”, also eine 4 Bit Zahl): 01 0001
- Bits für den Wert 3 (Einleitungen für die Unterfunktion “Wert” und 3 als 4 Bit natürliche Zahl; der Werte Definitionsbereich “underfunction” ist “naturalNumberB(4)”, also eine 4 Bit Zahl): 00 0011
- Bits für die ganze If-Bedingung (erstes Bit vorn):
  10111100 11111110 10100000 1100 ... (=Bits für die zwei Unterobjekte)

25.3.9 Externe Objekte

Einleitung: 1101

Zur Beschreibung des Fib-Elements für externe Objekte siehe Abschnitt 12.9 Seite 43 .

Nach der Einleitung folgt direkt eine Zahl für den Identifier des externen Objekts, der Definitionsbereich, und damit die Bits für die Zahl, wird durch den Definitionsbereich des ersten Elements des den Definitionsbereich “externObject” bestimmt (siehe Abschnitt 25.2.4 auf Seite 113).

Nach der Identifier folgt die Anzahl der Eingabewerte E, welche den Definitionsbereich des zweiten Elements des Definitionsbereichs “externObject” hat. Ist dieser Wert 0 wird die Anzahl der Eingabewerte aus der Anzahl der Vektorelemente des Vektors für die Eingabewerte bestimmt, also den Elementen des Definitionsbereichs für ”externObjectInput” mit dem entsprechenden Identifier, wenn kein solcher existiert ist davon auszugehen das keine / 0 Eingabewerte ($E = 0$) existieren.

Danach folgt der Vektor (siehe Abschnitt 25.3.1 auf Seite 129) mit den Eingabewerten. Der Definitionsbereich für den Vektor ist der Definitionsbereich für das “externObjectInput” mit dem entsprechenden Identifier oder $vectorOpenEnd(integerB(8))$, wenn kein solcher Definitionsbereich existiert.

Auf den Vektor mit den Eingabewerten folgen die Unterobjekte des externen Objekts. Diese werden durch eine Zahl für die Anzahl der Unterobjekte U eingeleitet. Der Definitionsbereich dieser Zahl U ist der Definitionsbereich des dritten Elements des den Definitionsbereich “externObject”. Danach folgen hintereinander die U Datensätze der U Unterobjekte.

Jeder Datensatz für ein Unterobjekt wird durch eine Zahl für die Anzahl der Ausgabevariablen A_i ($i = 1 \dots U$) eingeleitet. Der Definitionsbereich dieser Zahl

A_i ist der Definitionsbereich des vierten Elements des den Definitionsbereich "externObject".

Danach folgt das Unterobjekt des Datensatzes, welches ein normales Fib-Objekt ist.

25.3.10 Externe Unterobjekte

Einleitung: 1110

Zur Beschreibung des Fib-Elements für externe Unterobjekte siehe Abschnitt 12.10 Seite 44 .

Auf die Einleitung folgt eine natürliche Zahl für die Nummer des externen Unterobjekts. Der Definitionsbereich dieser Nummer ergibt sich implizit aus der Anzahl N der Unterobjekte (bzw. der höchsten Nummer für einen "externSubobject" Definitionsbereich) im aktuellen/ nächsten root-Element , siehe Abschnitt 25.2.4 auf Seite 118 . Der Definitionsbereich ist: $naturalNumber(N)$, siehe Tabelle 11 auf Seite 126 .

Danach folgt der Vektor mit den Ausgabewerte (siehe Abschnitt 25.3.1 auf Seite 129), bzw. Eingabevariablen für das Unterobjekt. Der Definitionsbereich für den Vektor ist der Definitionsbereich für das "externSubobject" mit dem entsprechenden Nummer oder der Standarddefinitionsbereich für "externSubobject", wenn kein solcher Definitionsbereich existiert.

25.3.11 Definitionsbereichseigenschaften abrufen

Einleitung: 0000 0000 0000 1111

Zur Beschreibung des Definitionsbereichseigenschaftselements sei auf Abschnitt 12.11 auf Seite 46 verwiesen.

Nach der Einleitung folgt der Definitionsbereich (die Domain), von welcher der Wert zurückgegeben werden soll. Die Angabe des Definitionsbereichs erfolgt über den Elementnamen wie in Abschnitt 25.2.4 auf Seite 114 beschreiben.

Danach folgt die Angabe der $Element$ Parameter, für die Auswahl eines Unterdefinitionsbereichs. Als erstes folgt dafür ein $Element$-Einleitungsbit, welches angibt ob ein weiterer $Element$ Parameter folgen. Ist es 0 folgt kein $Element$ Parameter, ist es 1 folgt ein $Element$ Parameter. Wenn es 1 ist folgt auf dieses erste $Element$-Einleitungsbit eine 1 Byte (8-Bit) lange natürliche Zahl $Bits$, die Angibt, wieviele Bits pro $Element$ Parameter verwendet werden. Danach folgt der erste $Element$ Parameter. Nach jedem $Element$ Parameter (abgespeichert mit dem Definitionsbereich $naturalNumberB(Bits)$) folgt wieder ein $Element$-Einleitungsbit und danach eventuell der nächste $Element$ Parameter usw. . Beispiel $matrix.3.1$: zwei Bits werden benötigt um die $Element$ Parameter abzuspeichern; die Bits sind (erstes Bit vorn): 1111 0000 0000 0000 0 00 11011 1 00000010 11 1 10 0 (In ihrer Reihenfolge stehen die Felder für: Definitionsbereichseigenschaftselement; 8 Bit Elementname; keine Parameter; matrixElement;

$Element$ Parameter folgt; mit je 2 Bit; 3'ter Unterdefinitionsbereich; $Element$ Parameter folgt; erster Unterdefinitionsbereich; kein $Element$ Parameter folgt)

Nach der Angabe für das Element folgt der $Mode$, also welcher Eigenschaftswert des Definitionsbereichs ausgewählt wird. Dies ist eine 8 Bit natürliche Zahl, so wie in Tabelle 4 auf Seite 47 in der Spalte "Wert" beschrieben.

Am Ende folgt das Unterobjekt des Definitionsbereichseigenschaftselements.

25.3.12 Set-Element

Einleitung: 0000 0000 0001 1111

Zur Beschreibung des Set-Element siehe Abschnitt 12.12 auf Seite 48 .

Der Definitionsbereich für das Set-Element ist "setElement" (siehe Abschnitt 25.2.4 auf Seite 113).

Nach der Einleitung folgt zuerst ein Bit, dass angibt ob ein besonderer Definitionsbereich verwendet wird. Ist dieses Bit 0 wird keine Definitionsbereichsnummer $DomainNr$ angegeben und die $DomainNr$ wird mit 0 belegt. Sonst, wenn es 1 ist, folgt eine natürliche Zahl für die Nummer des verwendeten Definitionsbereichs $DomainNr$. Die Anzahl der Bit für diese Nummer ergibt sich aus der größten Nummer $max(i)$ für einen Definitionsbereich für das Set-Element und ist $\lceil \log_2(max(i)) \rceil$. Aus der $DomainNr$ ergibt sich der Definitionsbereich des Elements. Er ist der erste "setElement" Definitionsbereich der eine Definitionsbereichnummer kleiner gleich der gesetzen (geladenen) Definitionsbereichsnummer $DomainNr$ hat.

Danach folgen zwei natürlich Zahlen.

Die erste Zahl gibt die Anzahl (n) der Variablen und der zu setzenden Werte pro Satz an. Die Bits für diese Zahl ergeben sich aus dem ersten Element /Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Die zweite Zahl gibt die Anzahl (k) der Sätze mit zu setzenden Werten an. Die Bits für diese Zahl ergeben sich aus dem zweiten Element /Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Nach den beiden Anzahlwerten folgen k Vektoren/Sätze (siehe Abschnitt 25.3.1 auf Seite 129) mit je n Werten (bzw. es folgen $k * n$ Werte/Vektorelemente direkt hintereinander). Der Definitionsbereich für die Vektoren/Sätze ist das dritte Element / Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Am Ende folgt das Unterobjekt des Set-Elements.

25.3.13 Matrixelement

Einleitung: 0000 0000 0010 1111

Zur Beschreibung des Matrixelement siehe Abschnitt 12.13 auf Seite 49 .

Der Definitionsbereich für das Matrixelement ist "matrixElement" (siehe Abschnitt 25.2.4 auf Seite 113).

Nach der Einleitung folgt zuerst ein Bit, dass angibt ob ein besonderer Definitionsbereich verwendet wird. Ist dieses Bit 0 wird keine Definitionsbereichsnummer $DomainNr$ angegeben und die $DomainNr$ wird mit 0 belegt. Sonst, wenn es 1 ist, folgt eine natürliche Zahl für die Nummer des verwendeten Definitionsbereichs $DomainNr$. Die Anzahl der Bit für diese Nummer ergibt sich aus der größten Nummer $max(i)$ für einen Definitionsbereich für das Matrixelement und ist $\lceil \log_2(max(i)) \rceil$. Aus der $DomainNr$ ergibt sich der Definitionsbereich des Elements. Er ist der erste "matrixElement" Definitionsbereich der eine Definitionsbereichnummer kleiner gleich der gesetzen (geladenen) Definitionsbereichsnummer $DomainNr$ hat.

Danach folgen drei natürlich Zahlen.

Die erste Zahl gibt die Anzahl (d) der Dimensionen /Dimensionsvariablen an.

Die zweite Zahl gibt die Anzahl (i) der Wertevariablen und der zu setzenden Werte pro Satz an. Die Bits für die ersten beiden Zahlen ergeben sich aus dem ersten Element /Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Die dritte Zahl gibt die Anzahl (k) der Sätze mit zu setzenden Werten an. Die Bits für diese Zahl ergeben sich aus dem zweiten Element /Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Nach den Anzahlwerten folgen d Vektoren (siehe Abschnitt 25.3.1 auf Seite 129) für die Bereiche bzw. Start und Endwerte für die jeweiligen Dimensionen. Dabei stehet der erste Vektor für die erste Dimensionsvariable, der zweite Vektor für die zweite Dimensionsvariable usw. . Der Definitionsbereich für die Vektoren/-Sätze ist das dritte Element / Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Nach den Vektoren für die Bereich folgen k Vektoren/Sätze (siehe Abschnitt 25.3.1 auf Seite 129) mit je i Werten (bzw. es folgen $k * i$ Werte/Vektorelemente direkt hintereinander). Der Definitionsbereich für die Vektoren/Sätze ist das vierte Element / Unterdefinitionsbereich des Vektordefinitionsbereichs für das Element.

Am Ende folgt das Unterobjekt des Matrixelement.

26 XML-Format

Jedes Fib-Element und jeder Vektor hat im XML-Format sein eigenes XML-Element. Einfache Werte der Fib-Elemente und Vektoren werden als Attribute gespeichert.

Im Folgendem Listing ist ein einfaches Beispiel eines Fib-Objekts im XML-Format angegeben.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<fib_object
   xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://www.fib-development.org/"
```

```
   xsd:schemaLocation="http://www.fib-development.org/fib.xsd">

   <root>
      <multimedia_info fib_version="3" db_version="2"/>
      <optionalpart>
         <pair key="autor" value="Oesterholz"/>
      </optionalpart>
      <domains>
         <dim count="2">
            <dimension number="1" direction="horizontal"/>
            <dimension number="2" direction="vertical"/>
            <vector elements="2">
               <naturalNumberB bit="8">
               <integer min="10" max="123">
            </vector>
         </dim>
      </domains>
      <main_fib_object>
         <area defined_variable="1">
            <vector type="subarea">
               <value>3</value>
               <value>15</value>
            </vector>

            <property>
               <vector type="property.colorGrayscale">
                  <value>200</value>
               </vector>
               <point>
                  <vector type="position">
                     <value>4</value>
                     <variable>1</variable>
                  </vector>
               </point>
            </property>
         </area>
      </main_fib_object>
   </root>
</fib_object>
```

26.1 XML-Header

Der Header des Fib-XML-Formats ist immer gleich aufgebaut. Er spezifiziert das XML-Format. Das oberste Element des Fib-XML-Formats ist immer ein Element mit dem Namen `fib_object`. Dieses enthält die Attribute, um das XML-Format spezifizieren und ein root-Objekt.

Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<fib_object
   xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://www.fib-development.org/"
   xsd:schemaLocation="http://www.fib-development.org/fib.xsd">

      ... <!-- root-Element -->

</fib_object>
```

26.2 Root-Element

Zur Beschreibung des root-elements siehe Abschnitt 12.14 auf Seite 52 .

Das Element für das root-Element hat den Namen `root`.

Das root-Element kann folgende Elemente, in der nachfolgenden Reihenfolge, enthalten:

1. Multimediainformationen (siehe Abschnitt 26.2.1 auf Seite 144)
2. Der optionale Teil (siehe Abschnitt 26.2.2 auf Seite 144)
3. Definitionsbereiche (siehe Abschnitt 26.2.3 auf Seite 144)
4. Definitionsbereiche für Werte (siehe Abschnitt 26.2.3 auf Seite 144)
5. Eingabevariablen (siehe Abschnitt 26.2.4 auf Seite 154)
6. Haupt-Fib-Objekt (siehe Abschnitt 26.2.5 auf Seite 154); dieses Element ist immer vorhanden
7. weitere root-Objekte (siehe Abschnitt 26.2.6 auf Seite 154)
8. Identifier von verwendeten Datenbankobjekten (siehe Abschnitt 26.2.7 auf Seite 155)
9. eventuell ein Checksummenelement (siehe Abschnitt 26.2.8 auf Seite 155), dieses ist für das Fib-XML-Format ohne Bedeutung und dient nur der Datenhaltung

26.2.1 Multimediainformationen

Das Element für Multimediainformationen hat den Namen `multimedia_info`.

Das Element für Multimediainformationen hat zwei Attribute für die Version von Fib und der Fib-Datenbank.

Das Attribute `fib_version` gibt die Fib-Version des Fib-Objekts an und das Attribute `db_version` die Datenbankversion. Beide Versionsnummern sind natürliche Zahlen. Sie können zu einer menschenlesbaren Form (z. B. “Fib V1.2.3”) gemappt werden. Hier wird allerdings nur jeweils eine Zahl verwendet, da sonst eine bestimmte Form festgelegt würde, die nur noch schwer geändert werden kann. Es kann eine menschenlesbaren Form der Version im optionalen Teil angegeben werden.

Ein Beispiel für die Multimediainformationen:

```
<multimedia_info fib_version="3" db_version="2"/>
```

26.2.2 Optionaler Teil

Das Element für den optionalen Teil hat den Namen `optionalpart`.

Es enthält eine Liste mit Elementen für die Eintrage im optionalen Teil. Diese Elemente haben den Namen `pair`. Die `pair`-Elemente haben die zwei Attribute `key` und `value`. Das Attribut `key` enthält den Schlüssel und das Attribut `value` enthält den Wert des Eintrags.

Ein Beispiel für die Optionaleninformationen:

```
<optionalpart>
   <pair key="copyright" value="GNU GPL 3"/>
   <pair key="type" value="die Berliner Mauer"/>
</optionalpart>
```

26.2.3 Definitionsbereiche

Das Definitionsbereichselement hat den Namen `domains` und besteht aus einer Definitionsbereichsliste.

Das Definitionsbereichselement für Wertedefinitionsbereiche hat den Namen `valuedomains` und ist genauso wie das Definitionsbereichselement mit den Namen `domains` aufgebaut. Dieses Element (`valuedomains`) hat auf das Abspeichern im XML-Format keine Auswirkung, sondern dient nur der Informationshaltung.

Die Definitionsbereichsliste gibt die Definitionsbereiche der einzelnen Elemente an, ist ein Wert (z. B. einer Variable) außerhalb dieses Definitionsbereichs, wird er auf einen Wert innerhalb des Definitionsbereichs gerundet. Werte außerhalb dieser Definitionsbereiche können also für das Element nicht auftreten.

Die Wertedefinitionsbereiche enthalten auch Definitionsbereiche, nur gelten diese Definitionsbereiche nicht für Elemente die Variablen enthalten. Die Wertedefinitionsbereiche bestimmen, wieviel Bits zum komprimierten Speichern (siehe Abschnitt 25 auf Seite 105) eines Elements, welches ein Wert enthält, benötigt wird. Die Wertedefinitionsbereiche sind also dann sinnvoll, wenn die Werte eines Elements nicht den vollen möglichen Definitionsbereich für das Element ausschöpfen. Wenn beispielsweise ein Unterobjekt nur Punkt enthält, deren Positionsvektoren nur Variablen und Werte zwischen 0 und 10 enthalten, kann der Definitionsbereich der Speicherplatzliste für Positionsvektoren auf "integerB" mit 4 Bit gesetzt werden, auch wenn die Variablen der Positionsvektoren Werte über 100 einnehmen können.

Die Wertedefinitionsbereiche werden beim Abspeichern im komprimierten Format eventuell erzeugt und optimiert, um den Speicherplatz für das Fib-Objekt möglichst gering zu halten.

Der Grund, für die Einführung von zentral (in den root-Elementen) festgelegten Definitionsbereiche für Werte, ist, das einerseits beim komprimierten Abspeichern des Fib-Objekts möglichst wenig Speicherplatz für die Werte verbraucht wird, ohne die Belegungsmöglichkeiten für die Werte drastisch einzuschränken und andererseits vorab bestimmt werden kann wie und ob das Multimediaobjekt angezeigt werden kann (z. B. wie es Skaliert werden muss oder ob die Anzeige aller Werte unmöglich ist).

Jede der beiden Listen enthält eine Anzahl von Definitionsbereichsdefinitionen. Eine Definitionsbereichsdefinition ist ein XML-Element für den Typ des Fib-Vektorelements für das es gilt. Dieses XML-Element enthält dann den eigentlichen Definitionsbereich.

XML-Elementnamen für Fib-Vektorelemente Die Tabelle 15 listet die Namen der XML-Elemente und die Attribute der XML-Elemente für die verschiedenen Typen bzw. den Fib-Elementen auf.

Name	Beschreibung
dim	Der Definitionsbereich für Positionsvektoren bzw. die Dimensionen. Das Attribut ist `count`, welches die Anzahl der Dimensionen angibt. Dieses Element enthält für jede Dimension ein Element mit dem Namen `dimension`. Die `dimension`-Elemente haben die zwei Attribute `number` und `direction`. Das Attribut `number` ist eine natürliche Zahl größer 0, welche die Nummer der Dimension angibt. In Positionsvektoren steht das `number`'te Element für die Dimension. Das Attribut `direction` gibt die Richtung an, in welche die Dimension geht. Dies ist eine natürliche Zahl, welche angibt auf was die `number`'te Dimension gemappt wird. Die möglichen Werte sind in Tabelle 7 auf Seite 64 beschrieben.
subfunction	Definitionsbereich für die Elemente von Unterfunktionen
property	Definitionsbereich eines Eigenschaftselements mit dem gegebenen Namen. Das Attribut `name` enthält den Namen der Eigenschaft. Mögliche Namen sind in Tabelle 2 auf Seite 30 zu sehen.
inVar	Definitionsbereich für die `number`'ten Eingabevariable. Das Attribut ist `number`, welches die Nummer der Eingabevariable angibt. (Die Zählung der Eingabevariablen eines root-Elements beginnt bei 1 .)
Namen von XML-Elementen für Definitionsbereiche die beim komprimierten Abspeichern relevant sind	
area	Dieser Typ ist für den Definitionsbereich für das Bereichselement (siehe Abschnitt 12.6 auf Seite 34). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 2 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Unterbereiche, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite und letzte Element ist der Definitionsbereich für die Vektoren für die Unterbereiche (B_1) und ist ein Definitionsbereich für Vektoren dessen zwei Elemente ganze Zahlen sind.
variable	Werte die benötigt werden, um Variablen zu kodieren. Der Definitionsbereich sollten die natürliche Zahlen von 0 bis maximale Anzahl der definierten Variablen in den Fib-Blättern im Haupt-Fib-Objekt sein. Das Fib-Baum-Blatt im Haupt-Fib-Objekt über dem meisten Variablen definiert werden, bestimmt also den Definitionsbereich, bzw. der Ast mit den meisten definierten Variablen.
comments	Werte die benötigt werden, um Kommentare zu kodieren. Der Definitionsbereich sollten die natürliche Zahlen von 0 bis Anzahl der Kommentare im Haupt-Fib-Objekt sein.

Name	Beschreibung
externObject	Definitionsbereich für externe Objekte im Haupt-Fib-Objekt. Der Definitionsbereich ist ein Vector mit 4 Elementen. Die Vektorelemente dienen der Reihenfolge für den Identifier, die Anzahl der Eingabewerte NumberInVal, die Anzahl der Unterobjekte NumberSubobjects und die Anzahl der Ausgabevariablen NumberOutVar. Alle Vektorelementdefinitionsbereiche, außer dem für den Identifier, kommen aus den natürlichen Zahlen. Der Vektorelementdefinitionsbereich für den Identifier kommt aus den Ganzzahlen.
externObjectInput	Dieser Typ ist für den Definitionsbereich für die Eingabewerte für externe Objekte (siehe Abschnitt 12.9 auf Seite 43). Der Definitionsbereich ist ein Vektordefinitionsbereich. Das Attribute `identifier` ist eine Ganzzahl und gibt den Identifier des externe Objekts an, für dessen Element der Definitionsbereich ist.
externSubobject	Dieser Typ ist für den Definitionsbereich für die Eingabewerte für externe Unterobjekte (siehe Abschnitt 12.10 auf Seite 44). Der Definitionsbereich ist ein Vektordefinitionsbereich. Das Attribute `number` ist eine natürliche Zahl und gibt die Nummer des externe Unterobjekts an, für das der Definitionsbereich ist.
setElement	Dieser Typ ist für den Definitionsbereich für das Set-Element (siehe Abschnitt 12.12 auf Seite 48). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 3 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Variablen und der zu setzenden Werte pro Satz, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich dient für die Anzahl (k) der Sätze mit zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{i.g}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind. Des weiteren kann als Attribut `domainNr` eine natürliche Zahl für den Definitionsbereichnummer $DomainNr$ angegeben werden. Wenn das Attribut `domainNr` fehlt, ist die die Definitionsbereichnummer $DomainNr$ gleich 0 .

Name	Beschreibung
matrixElement	Dieser Typ ist für den Definitionsbereich für das Matrixelement (siehe Abschnitt 12.13 auf Seite 49). Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 4 Elementen/Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich ist für die Anzahl (d) der Dimensionsvariablen, die Anzahl (i) der Wertevariablen und die zu setzenden Werte i pro Satz. Er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich ist für die Anzahl (k) der Sätze mit den zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte Element ist der Definitionsbereich für die Bereiche bzw. Start und Endwerte für die einzellenen Dimensionsvariablen, er ist ein Vektordefinitionsbereich mit zwei Elementen, welche jeweils aus den ganzen Zahlen kommen. Das vierte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{a.b}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind. Des weiteren kann als Attribut `domainNr` eine natürliche Zahl für den Definitionsbereichnummer $DomainNr$ angegeben werden. Wenn das Attribut `domainNr` fehlt, ist die die Definitionsbereichnummer $DomainNr$ gleich 0 .

Tabelle 15: Namen von XML-Elementen für Definitionsbereiche

Möglich Definitionsbereiche Jedem Definitionsbereich ist eine eigenes XML-Element zugeordnet. Jedes Definitionsbereich XML-Element kann ein Attribut für den Skalierungsfaktor `scalingfactor` besitzen. Dieses kann weggelassen werden, wenn der Skalierungsfaktor 1 ist, bzw. der Skalierungsfaktor ist 1, wenn es weggelassen wird.

Der Skalierungsfaktor `scalingfactor` ist eine Gleitkommazahl mit dem die Werte des unskalierten Definitionsbereich bzw. dem Grunddefinitionsbereich multipliziert werden, um den skalierten Definitionsbereich zu erhalten.
Beispiel: Der Definitionsbereich sind 2 Bit natürliche Zahlen mit den Skalierungsfaktor $scalingfactor = 1 * 2^{-1} = 1/2$. Möglich Werte sind dann: $\{\ 0 * 1/2 = 0;$ $1 * 1/2 = 0,5; 2 * 1/2 = 1; 3 * 1/2 = 1,5\ \}$.

Die Tabelle 16 listet die Namen der XML-Elemente und die Attribute der XML-Elemente für die verschiedenen Definitionsbereiche auf.

Name	Beschreibung	Parameter
naturalNumberB	Der Grunddefinitionsbereich sind die natürlichen Zahlen.	Das Attribute `bit` ist eine natürliche Zahl, welche die Anzahl der Bits angibt, den Werte des Definitionsbereichs lang sind. Der zugehörige Grunddefinitionsbereich ist dann $0\ldots(2^{bit}-1)$.
integerB	Der Grunddefinitionsbereich sind die Ganzzahlen.	Das Attribute `bit` ist eine natürliche Zahl, welche die Anzahl der Bits angibt, den Werte des Definitionsbereichs lang sind. Der zugehörige Grunddefinitionsbereich ist dann $-(2^{bit-1})\ldots(2^{bit-1}-1)$.
naturalNumber	Der Grunddefinitionsbereich sind die natürlichen Zahlen.	Das Attribute `max` ist eine natürliche Zahl, welche die größte natürliche Zahl des Grunddefinitionsbereich angibt. Der zugehörige Grunddefinitionsbereich ist dann $0\ldots max$.
integer	Der Grunddefinitionsbereich sind die Ganzzahlen.	Das Attribute `min` und `max` sind zwei Ganzzahlen. Das Attribute `min` gibt die untere Grenze des Grunddefinitionsbereich an. Das Attribute `max` gibt die obere Grenze des Grunddefinitionsbereich an. Der zugehörige Grunddefinitionsbereich ist dann $min\ldots max$.
integerValues	Der Grunddefinitionsbereich sind die Ganzzahlen.	Für jeden Wert des Grunddefinitionsbereichs enthält das `integerValues`-Element ein `value`-Element in dem der Wert steht (z. B. `<value>17</value>`).
real	Der Grunddefinitionsbereich sind Gleitkommazahlen. Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$.	Dieses Element enthält zwei Elemente. Das erste enthaltende Element gibt den Definitionsbereich der Mantisse und der zweite des Exponenten an. Die Angabe der Definitionsbereiche ist so wie in dieser Tabelle beschrieben. Beide Definitionsbereiche müssen aus den Ganzzahlen kommen. (also integer... oder naturalNumber...) Beispiel: `<real>` `<integer min="-100" max="100"/>` `<naturalNumberB scalingfactor="0.1"` `bit="8"/></real>` Die Mantisse ist eine natürliche Zahl von -100 bis 100 und der Exponenten ist eine acht Bit natürliche Zahl die mit 0.1 skaliert wird.

Name	Beschreibung	Parameter
realValues	Der Grunddefinitionsbereich sind Gleitkommazahlen. Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$.	Dieses Element enthält mehrere Elemente. Das erste enthaltende Element gibt den Definitionsbereich der Mantisse und der zweite des Exponenten an. Die Angabe der Definitionsbereiche ist so wie in dieser Tabelle beschrieben. Beide Definitionsbereiche müssen aus den Ganzzahlen kommen. (also integer... oder naturalNumber...) Nach diese Elementen folgen für jeden Wert des Grunddefinitionsbereichs ein `value`-Element in dem der Wert steht (z. B. `<value>17.14</value>`). Beispiel: `<real> <integer min="-100" max="100"/> <naturalNumberB scalingfactor="0.1" bit="8"/> <value>7*2^0.5</value> <value>31*2^10</value> </real>` Die Mantisse ist eine natürliche Zahl von -100 bis 100 und der Exponenten ist eine acht Bit natürliche Zahl die mit 0.1 skaliert wird. Die Werte im Definitionsbereich sind $7 * 2^{0.5} \approx 9.9$ und $31 * 2^{10} \approx 31744$.
vector	Der Grunddefinitionsbereich sind Vektoren.	Dieses Element besitzt das Attribut `elements`, welches die Anzahl der Elemente E des Vektors angibt. Das XML-Element `vector` enthält weiterhin für jedes Elemente des Vektors (also `elements` Stück) ein XML-Element mit dem Definitionsbereich des Elements des Vektors. Wobei das erste enthaltende XML-Element den Definitionsbereich des ersten Vektorelements angibt, das Zweite des Zweiten usw. . Alle Werte des Definitionsbereichs haben dann die Form $(D_1, \ldots, D_E)$, wobei D_i ein Wert aus dem i'ten Definitionsbereich der Liste ist.

Name	Beschreibung	Parameter
vectorValues	Der Grunddefinitionsbereich sind Vektoren.	Dieses Element besitzt das Attribut `elements`, welches die Anzahl der Elemente des Vektors angibt. Das XML-Element `vectorValues` enthält weiterhin für jedes Elemente des Vektors (also `elements` Stück) ein XML-Element mit dem Definitionsbereich des Elements des Vektors. Wobei das erste enthaltende XML-Element den Definitionsbereich des ersten Vektorelements angibt, das Zweite des Zweiten usw. . Darauf folgt eine Liste mit den Vektoren, so wie im Abschnitt 26.3.1 auf Seite 156 beschreiben, ohne das `type` Attribute. Die Definitionsbereiche der Vektoren sind die angegebenen Definitionsbereich der vorhergehenden Liste. Es können auch Variablen in den abgespeicherten Vektoren vorkommen. Bei einer Variable im Vektor wird der Variablenbezeichner weggelassen. Beispiel: `<vectorValues elements="2"> <integer min="-100" max="100"/> <naturalNumberB scalingfactor="0.1" bit="8"/> <vector> <value number="1">99</value> <value number="2">2</value> </vector> <vector> <value number="1">-17</value> <variable number="2"></variable> </vector> </vectorValues>` ; An der Angabe der Vektoren in Fib-Elementen ändert dieser Definitionsbereich beim XML-Format nichts. Vektoren werden abgespeichert wie mit dem `vector` Definitionsbereich.

Name	Beschreibung	Parameter
vectorOpenEnd	Der Grunddefinitionsbereich sind Vektoren.	Dieses Element besitzt das Attribut `elements`, welches die Anzahl der mindest Elemente E des Vektors angibt. Das XML-Element `vectorOpenEnd` enthält weiterhin für jedes der ersten E Elemente des Vektors (also `elements` Stück) ein XML-Element mit dem Definitionsbereich des Elements des Vektors. Wobei das erste enthaltende XML-Element den Definitionsbereich des ersten Vektorelements angibt, das Zweite des Zweiten usw. und der E'te den Definitionsbereich der E'ten und folgenden Elemente. Alle Werte des Definitionsbereichs haben dann die Form $(D_1, \ldots, D_E, \ldots, D_E)$, wobei D_i ein Wert aus dem i'ten Definitionsbereich der Liste ist. Dieser Definitionsbereich dient für Elemente, die Vektoren unterschiedlicher Größe enthalten können. Die Anzahl der Elemente eines Vektors zu diesem Definitionsbereich wird durch das Element bestimmt, welches den Vektor enthält.
domainReference	Ein Verweis auf den (Unter-)Definitionsbereich eines anderen Elements. Als Definitionsbereich wird der Definitionsbereich des Elements mit dem gegebenen Definitionsbereichnamen genommen.	Als weiters kann dieses Element ein optionales Attribut `subdomain` enthalten. Das Attribut `subdomain` spezifiziert den Unterefinitionsbereich auf welchen verwiesen wird. Dabei können die Nummern (die Zählung beginnt bei 1) von Unterdefinitionsbereichen auf meheren Stufen durch einen Punkt getrennt angegeben werden. Wobei die Nummer eines Unterdefinitionsbereichs vor der Nummer des in ihm enthaltenden Unterdefinitionsbereichs steht. Das `domainReference` enthält eine XML-Element für den Definitionsbereichsnamen auf welchen verwiesen wird. Dieses XML-Element wird wie in Tabelle 15 auf Seite 148 beschrieben erzeugt. Beispiel `<domainReference subdomain="3.1"> <matrixElement/> </domainReference>`: Verwendet wird der erste Unterdefinitionsbereich des dritten Unterdefinitionsbereichs des matrix Elements.

Name	Beschreibung	Parameter
defaultDomain	Der angegebene Definitions-bereich wird nur verwendet, wenn es für das entsprechende Element bisher keinen anderen Definitionsbereich gibt.	Dieses Element enthält ein XML-Element für einen Definitionsbereich wie in dieser Tabelle beschrieben. Der enthaltende Definitionsbereich wird nur verwendet, wenn es für das entsprechende Element bisher keinen anderen Definitionsbereich gibt.

Tabelle 16: Elemente für Definitionsbereiche

Ein Beispiel für die Definitionsbereiche:

```
<domains>
   <dim count="2">
      <dimension number="1" direction="horizontal"/>
      <dimension number="2" direction="vertical"/>
      <vector elements="2">
         <naturalNumberB scalingfactor="0.1" bit="8"/>
         <integer scalingfactor="0.1" min="10" max="110"/>
      </vector>
   </dim>
   <property name="colorRGB">
      <vector elements="3">
         <naturalNumberB bit="8"/>
         <integer min="-10" max="246"/>
         <naturalNumberB bit="9"/>
      </vector>
   </property>
</domains>
<valuedomains>
   <property name="colorRGB">
      <vector elements="3">
         <naturalNumberB bit="4"/>
         <integer min="-10" max="10"/>
         <integerValues count="4">
            <value>17</value>
            <value>-3</value>
```

```
            <value>12</value>
            <value>124</value>
         </integerValues>
      </vector>
   </property>
</valuedomains>
```

26.2.4 Eingabevariablen

Das XML-Element für die Eingabevariablen trägt den Namen `input_variables`. Es enthält für jede Eingabevariable ein XML-Element mit dem Namen `variable`. Die `variable`-Elemente haben die zwei Attribute `number` und `default`. Das Attribut `number` gibt die Nummer der Eingabevariable an. Mit dem Attribut `default` wird der Standardwert der Eingabevariable festgelegt.

Ein Beispiel für das Eingabevariablenelement:

```
<input_variables>
   <variable number="1" default="3"/>
   <variable number="2" default="17.3"/>
</input_variables>
```

26.2.5 Haupt-Fib-Objekt

An dieser Stelle kommen die Daten des Haupt-Fib-Objekts in Form seiner Elemente und deren Parameter, so wie es weiter unten im Abschnitt 26.3 auf Seite 156 beschrieben wird. Es ist in einem Xml-Element mit dem Namen `main_fib_object` verpackt.

Ein Beispiel für Haupt-Fib-Objekt:

```
<main_fib_object>
   ...<!-- Haupt-Fib-Objekt -->
</main_fib_object>
```

26.2.6 Weitere root-Objekte

Das XML-Element für die Unter-root-Objekte heißt `sub_roots`. Es enthält für jedes Unter-root-Objekt ein XML-Element mit dem Namen `sub_root`. Das Element `sub_root` hat das Attribut `identifier`, welches den Identifizierer der Unter-root-Objekts angibt (dies ist eine Ganzzahl). Weiterhin enthält das Element `sub_root` das entsprechende root-Objekt, so wie in dem Überabschnitt 26.2 beschrieben.

Ein Beispiel für Unter-root-Objekte:

```
<sub_roots>
   <sub_root identifier="1">
      <root>
          ...
      <\root>
   </sub_root>
   <sub_root identifier="3">
      <root>
          ...
      <\root>
   </sub_root>
</sub_roots>
```

26.2.7 Identifier von verwendeten Datenbankobjekten

Hier folgt eine Liste aller Identifier von Fib-Datenbankobjekten, welche im Haupt-Fib-Objekt oder auch in Unter-root-Objekten verwendet werden. Die Angaben dieser Identifier ist optional. Wenn Identifier vorhanden sind, kann von Vornherein geprüft werden, ob alle externen Fib-Objekte aus der Datenbank, welche benötigt werden, existieren oder ob es wahrscheinlich zu Anzeigefehlern kommt, da Fib-Datenbankobjekte fehlen. Ob Identifier von Fib-Datenbankobjekten, die in einem Haupt-Fib-Objekt eines root-Elements verwendet werden, in diesem root-Element angegeben werden oder in einem höheren root-Element, hängt von verschiedenen Gesichtspunkten ab. Für die Veränderung von Fib-Objekten, ist es vorteilhaft Identifier von Fib-Datenbankobjekten in einem root-Element, welches möglichst nahe am Verwendungsort liegt, anzugeben. Aus Platzgründen bei der komprimierten Speicherform kann es aber sinnvoll sein, Identifier von Fib-Datenbankobjekten in möglichst wenigen root-Elementen zusammenzufassen.

Der Name des entsprechenden XML-Elements für die Datenbankidentifier ist `database_identifiers`. Es enthält für jeden angegebenen Datenbankidentifier ein XML-Element mit dem Namen `identifier`. Dieses XML-Element `identifier` enthält wiederum eine negative Ganzzahl für den verwendeten Datenbankidentifier.

Ein Beispiel für die Identifier von verwendeten Datenbankobjekte:

```
<database_identifiers>
   <identifier>-21</identifier>
   <identifier>-632</identifier>
</database_identifiers>
```

26.2.8 Checksummenelement

Mit diesem Element kann auch das root-Element im komprimierten Speicherformat mit einer Checksumme versehen werden. Für das Fib-XML-Format sind diese

Angaben ohne Bedeutung.

Das Checksummenelement ist ein Eigenschaftsvektor für Checksummen, wie im Abschnitt 26.3.3 auf Seite 157 beschrieben.

Ein Beispiel für das Checksummenelement bzw. einen Checksummenvektor:

```
<vector type="property.checksum">
   <value number="1">1</value>
   <value number="2">256</value>
   <value number="3">64</value>
</vector>
```

26.3 Fib-Elemente

Jedes Fib-Element hat sein eigenes XML-Element. Deren Aufbau werden im Nachfolgenden beschrieben.

26.3.1 Vektoren

Vektoren bestehen aus mehreren Elementen, wobei jedes Element entweder eine Wert oder eine Variable sein kann. Die Anzahl der Elemente des Vektors ist entweder Fib-Element spezifisch oder wird durch einen zugehörigen Definitionsbereich bestimmt (siehe Abschnitt 12.14.2 auf Seite 54).

Alle Vektor-XML-Elemente haben den Namen `vector`. Das `vector`-Element hat das Attribut `type`, welches den Typ des Vektors angibt (Bsp.: `position` oder `property.colorRGB`). Des weiteren enthält das `vector`-Element für jedes Element des Vektors ein XML-Element. Das XML-Element für Vektorelemente, die Werte sind, hat den Namen `value`. Dagegen hat das XML-Element für Vektorelemente die Variablen sind den Namen `variable`. Beide XML-Element für Vektorelemente haben das optionale Attribut `number`, welches die Nummer (die Zählung beginnt bei 1) des Elements im Vektor angibt. Weiterhin enthalten beide den Wert des Vektorelements. Wobei bei Variablen die Nummer der Variable angegeben wird, so wie sie in der Definition des Variable festgelegt wurde.

Ein Beispiel für einen Vektor:

```
<vector type="position">
   <value>17</value>
   <variable>3</variable>
</vector>
```

Ein Beispiel für einen Vektor mit dem optionalen Attribut `number`:

```
<vector type="position">
   <variable number="1">5</variable>
   <value number="2">3.9</value>
</vector>
```

26.3.2 Punkte

Zur Beschreibung des Punktelements siehe Abschnitt 12.2 auf Seite 24 .

Ein normales Punktelement hat den Namen `point`. Ein Punkt ohne Positionsvektor ($point()$) hat auch den Namen `point` enthält aber keinen Positionsvektor. Wenn der Punkt für den gesamten Hintergrund ($point(())$ der also keine Positionsvektorelemente hat) gilt, ist der Name des XML-Elements `background`.

Normale Punkte enthalten einen Positionsvektor (siehe Abschnitt 26.3.1). Die Anzahl der Vektorelemente wird durch die Anzahl der Dimensionen bestimmt (siehe Abschnitt 26.2.3 auf Seite 144) die Anzahl der möglichen Werte pro Positionsvektorelement, wird durch den Definitionsbereich der Dimension bestimmt (siehe Abschnitt 26.2.3 auf Seite 148).

Ein Beispiel für einen normalen Punkt:

```
<point>
   <vector type="position">
      <variable>2</variable>
      <value>21</value>
   </vector>
</point>
```

Ein Beispiel für einen Punkt ohne Auswirkungen bzw. einen Punkt ohne Positionsvektor:

```
<point/>
```

Ein Beispiel für einen Punkt mit leerem Positionsvektor für den gesamten Hintergrund:

```
<background/>
```

26.3.3 Eigenschaft

Zur Beschreibung des Eigenschaftselements siehe Abschnitt 12.3 auf Seite 25 .

Der Name des Eigenschaftselements ist `property`. Es enthält einen Eigenschaftsvektor und ein Fib-Objekt (sein Unterobjekt).

Der Eigenschaftsvektor hat ein Attribut mit dem Namen `type` für den Typ der Property. Der Typ is immer `"property."` gefolgt von dem Namen der Eigenschaft, so wie in Tabelle 2 auf Seite 30 aufgeführt.

Ein Beispiel für einen Eigenschaftselement:

```
<property>
   <vector type="property.colorGrayscale">
      <value>200</value>
   <vector>
   ...<!-- Fib-Unterobjekt -->
</property>
```

Bei den Eigenschaften gibt es noch zwei besondere Eigenschaften, die Einfluss auf die Decodierung des komprimierten Fib-Objekts haben. Diese Eigenschaften sind “checksum” (siehe Abschnitt 25.3.3 auf Seite 131) und “boundSize” (siehe Abschnitt 25.3.3 auf Seite 132) . Für das XML-Format haben sie jedoch keine besondere Bedeutung, sondern dienen nur zur Informationshaltung. Der Vektor für die Eigenschaft “boundSize” besitzt dabei 0 Vektorelemente.

26.3.4 Listenelement

Zur Beschreibung des Listenelements siehe Abschnitt 12.4 auf Seite 31 .

Der Name des Listenelements ist `list`. Es enthält eine Anzahl von Unterobjekten die in ihrer Reihenfolge aufgeführt werden. (Das erste Element im XML-Listenelements ist auch das erste Unterobjekt des Listenelements usw. .)

Ein Beispiel für einen Listenelementelement:

```
<list>
   <!-- 1. Listenunterobjekt -->
   <property>
      <vector type="property.boundSize">
      </vector>
      ...<!-- Fib-Objekt -->
   </property>

   <!-- 2. Listenunterobjekt -->
   <point>
      <vector type="position">
         <variable>1</variable>
         <variable>2</variable>
      </vector>
   </point>

   <!-- 3. Listenunterobjekt -->
   <point>
      <vector type="position">
         <variable>52</variable>
         <value>36</value>
```

```
      </vector>
   </point>

</list>
```

26.3.5 Anmerkungselement

Zur Beschreibung des Anmerkungselements siehe Abschnitt 12.5 auf Seite 32 .

Anmerkungselements hat den Namen `comment`. Es hat zwei Attribute `key` und `value`. Das Attribut `key` gibt den Schlüssel des Anmerkungselements an und das Attribut `value` den Wert. Statt mit dem Attribut `value` kann der Wert des Anmerkungselements auch in einem XML-Element mit dem Namen `value` als Text abgespeichert werden. Des weiteren enthält das Anmerkungselements ein Fib-Objekt (sein Unterobjekt).

Ein Beispiel für einen Anmerkungselement:

```
<comment key="name" value="der Kasperle">
   ...<!-- Fib-Unterobjekt -->
</comment>
```

Ein Beispiel für einen Anmerkungselement mit einem `value` XML-Element:

```
<comment key="description">
   <value>
      Dies ist ein laengerer Text
      Zeile 2
   </value>
   ...<!-- Fib-Unterobjekt -->
</comment>
```

26.3.6 Bereiche

Zur Beschreibung des Bereichselements siehe Abschnitt 12.6 auf Seite 34 .

Der Name des XML-Bereichselements ist `area`. Das XML-Bereichselement hat ein Attribut `defined_variable`. Das Attribut `defined_variable` gibt an, welche Variable vom Bereichselement definiert wird. Es ist eine natürliche Zahl für eine Variable, welche noch nicht über dem Bereichselement, für eine andere Variable, verwendet wird.

Des weiteren enthält das Bereichselement für jeden seiner Unterbereiche einen Vektor vom Typ `subarea`. Wobei der erste Unterbereich zuerst im XML-Element kommt, dann das Zweite usw. .

Nach den Unterbereichen folgt das XML-Element für das enthaltende Unterobjekt.

Ein Beispiel für einen Bereichselement:

```
<area defined_variable="1">
   <vector type="subarea">
      <value>4</value>
      <value>10</value>
   </vector>
   <vector type="subarea">
      <value>15</value>
      <value>19</value>
   </vector>
   ...<!-- Fib-Unterobjekt -->
</area>
```

26.3.7 Funktionen

Zur Beschreibung des Funktionselements siehe Abschnitt 12.7 auf Seite 35 .

Der Name des XML-Funktionselements ist `function`. Das XML-Funktionselement hat ein Attribut namens `defined_variable`. Es gibt an, welche Variable vom Funktionselement definiert wird. Es ist eine natürliche Zahl für eine Variable, welche noch nicht über dem Funktionselement, für eine andere Variable, verwendet wird.

Des weiteren enthält das Funktionselement ein Unterfunktionselement (dieses wird beschrieben in Unterabschnitt 26.3.7), welches in ein XML-Element mit dem Namen `underfunction` eingebettet ist.

Nach dem Unterfunktionselement folgt das XML-Element für das enthaltende Unterobjekt.

Ein Beispiel für einen Funktionselement:

```
<function defined_variable="2">
   <subfunction>
      <add>
         <value>4</value>
         <div>
            <variable>1</variable>
            <value>3.14</value>
         </div>
      </add>
   </subfunction>
   ...<!-- Fib-Unterobjekt -->
</function>
```

Unterfunktionen

Die XML-Unterfunktionselement für Variablen und Werte (siehe Abschnitt 12.7.1 auf Seite 35) haben die Namen:

- `value`: Wert
- `variable`: Variable

Sie enthalten jeweils eine Zahl. Wobei bei der Variableunterfunktion (`variable`) der enthaltende Wert die Nummer der Variable ist, so wie sie definiert wird.

Weiterhin gibt es echte Unterfunktionen (siehe Abschnitt 12.7.2 auf Seite 36). Diese enthalten für jede ihrer Unterfunktionen ein XML-Element.

Namen von einstellig XML-Unterfunktionselement sind:

- `abs`: Absolutwert
- `sin`: Sinusfunktion
- `arcsin`: Arkussinus
- `log`: Logarithmus
- `round`: Runden
- `delay`: Delayunterfunktion

Namen von zweistellig XML-Unterfunktionselement sind:

- `add`: Addition
- `sub`: Subtraktion
- `mult`: Multiplikation
- `div`: Division
- `mod`: Modulo
- `exp`: Exponent
- `min`: Minimum
- `max`: Maximum

Bei den `sub`, `div` und `exp` Unterfunktionen ist die Reihenfolge der enthaltenden XML-Elemente wichtig. Dabei ist das erste enthaltende XML-Element die erste enthaltende Unterfunktion der Unterfunktion.

Weitere Unterfunktionen:

- `if`: If-Funktion (vor den zwei Unterfunktionen wird zuerst die Bedingung der If-Funktion ausgegeben, die Reihenfolge der enthaltenden XML Elements ist: Bedingung, wahr (true) Fall (Unterfunktion), falsch (false) Fall (Unterfunktion); siehe Abschnitt 26.3.8 auf Seite 162)

26.3.8 If-Bedingung

Zur Beschreibung des If-Elements siehe Abschnitt 12.8 auf Seite 41 .

Der Name des XML-If-Elements ist `if`.

Das If-Element enthält ein Bedingungselement `condition` (dieser wird beschrieben in Unterabschnitt 26.3.8).

Nach dem Bedingungselement folgen zwei XML-Element für die enthaltende Unterobjekte. Das erste XML-Element wird ausgewertet, wenn die Bedingung Wahr (=true) ist, und das zweite sonst (wenn die Bedingung Falsch (=false) ist).

Ein Beispiel für einen If-Element:

```
<if>
   <condition>
      <and>
         <true/>
         <lo>
            <variable>1</variable>
            <value>3.14</value>
         </lo>
      </and>
   </condition>
   ...<!-- Fib-Unterobjekt,
      wenn die Bedingung Wahr (=true) ist -->
   ...<!-- Fib-Unterobjekt,
      wenn die Bedingung Falsch (=false) ist -->
</if>
```

Bedingungen Bedingungen enthalten entweder XML-Elemente für 0 bis 2 Unterbedingungen (bzw. Bedingungen) oder XML-Elemente für zwei Unterfunktionen (siehe Abschnitt 26.3.7 auf Seite 160).

Die möglichen XML-Elemente für Bedingungen werden in Tabelle 17 aufgeführt.

26.3.9 Externe Objekte

Zur Beschreibung des Fib-Elements für externe Objekte siehe Abschnitt 12.9 auf Seite 43 .

Die XML-Elemente für externe Objekte besitzen den Namen `obj`. Sie haben ein Attribut namens `identifier`, welches den Identifier des einzusetzenden externen Objekts angibt. Des weiteren enthält das `obj` XML-Elemente eine XML-Element für die Eingabevariablen und eine XML-Element für jedes Teil-Fib-Objekte bzw. Fib-Unterobjekt, welches vom externen Objekt verwendet werden kann.

Elementname	Beschreibung	Enthaltende XML-Elemente
`false`	Die Bedingung ist Falsch.	keine
`true`	Die Bedingung ist Wahr.	keine
`not`	Die Bedingung ist genau dann Wahr, wenn die Unterbedingung Falsch ist, sonst ist sie Falsch.	eine Unterbedingung
`or`	Die Bedingung ist genau dann Wahr, wenn mindestens eine der beiden Unterbedingungen Wahr ist, sonst ist sie Falsch.	zwei Unterbedingung
`and`	Die Bedingung ist genau dann Wahr, wenn die beiden Unterbedingungen Wahr sind, sonst ist sie Falsch.	zwei Unterbedingung
`xor`	Die Bedingung ist genau dann Wahr, wenn nur eine der beiden Unterbedingungen Wahr sind, sonst ist sie Falsch.	zwei Unterbedingung
`eqInt`	Vergleich zweier auf Ganzzahlen gerundeter Zahlen auf Gleichheit. (zur Rundung siehe Abschnitt 12.6 auf Seite 34)	zwei Unterfunktionen, so wie in Abschnitt 26.3.7 auf Seite 160 beschrieben
`lo`	Vergleich, ob die erste Zahl kleiner als die zweite ist.	zwei Unterfunktionen, so wie in Abschnitt 26.3.7 auf Seite 160 beschrieben
`gr`	Vergleich, ob die erste Zahl größer als die zweite ist.	zwei Unterfunktionen, so wie in Abschnitt 26.3.7 auf Seite 160 beschrieben

Tabelle 17: XML-Elementnamen für Bedingungen

Das entsprechende XML-Element für die Eingabewerte ist ein Vektor (siehe Abschnitt 26.3.1 auf Seite 156) vom Typ `externObjectInput` (`<vector type="externObjectInput">`), er enthält die Eingabewerte.

Ein XML-Elemente für ein Teil-Fib-Objekt hat den Namen `subobject`. Das `subobject` XML-Element hat das optionale Attribut `number`, welches die Nummer (die Zählung beginnt bei 1) des Teil-Fib-Objekts angibt. Des weiteren enthält das `subobject` XML-Elemente eine XML-Element für die Ausgabevariablen des externe Objekts und eine XML-Element für das Fib-Unterobjekt, welches ein ganz normales Fib-Objekt ist, wie im Überabschnitt 26.3 beschrieben.

Das entsprechende XML-Element für die Ausgabevariablen hat den Elementnamen `output_variables`. Es einthält für jede Ausgabevariable ein XML-Element namens `variable`. Das `variable` XML-Element hat das optionale Attribut `number`, welches die Nummer (die Zählung beginnt bei 1) der Ausgabevariable angibt. Weiterhin enthält das `variable` XML-Element einen Wert für die Nummer der definierten Variable.

Ein Beispiel für ein externes Objekt:

```
<obj identifier="123">
   <vector type="externObjectInput">
      <value>5</value>
      <variable>1</variable>
      <variable>3</variable>
   </vector>
   <subobject>
      <output_variables>
         <variable>6</variable>
         <variable>7</variable>
         <variable>8</variable>
         <variable>9</variable>
      </output_variables>
      ...<!-- Fib-Unterobjekt nummer 1 -->
   </subobject>
   <subobject>
      <output_variables/>
      ...<!-- Fib-Unterobjekt nummer 2 -->
   </subobject>
   <subobject>
      <output_variables>
         <variable>6</variable>
         <variable>7</variable>
      </output_variables>
      ...<!-- Fib-Unterobjekt nummer 3 -->
   </subobject>
</obj>
```

Ein Beispiel für ein externes Objekt mit dem optionalen Attribut `number`:

```
<obj identifier="34">
   <vector type="externObjectInput">
      <variable number="1">2</variable>
      <value number="2">3.7</value>
      <variable number="3">5</variable>
   </vector>
   <subobject number="1">
      <output_variables>
         <variable number="1">6</variable>
         <variable number="2">7</variable>
         <variable number="3">8</variable>
      </output_variables>
      ...<!-- Fib-Unterobjekt nummer 1 -->
   </subobject>
   <subobject number="2">
      <output_variables>
         <variable number="1">6</variable>
         <variable number="2">7</variable>
      </output_variables>
      ...<!-- Fib-Unterobjekt nummer 2 -->
   </subobject>
</obj>
```

26.3.10 Externe Unterobjekte

Zur Beschreibung des Fib-Elements für externe Unterobjekte siehe Abschnitt 12.10 auf Seite 44 .

Die XML-Elemente für externe Unterobjekte hat den Namen `subobject`. Sie haben ein Attribut namens `number`, das die Nummer des externen Unterobjekts angibt, welches hier eingesetzt werden soll. Des weiteren enthält das `subobject` XML-Elemente eine XML-Element für die Ausgabevariablen.

Das entsprechende XML-Element für die Ausgabewerte ist ein Vektor (siehe Abschnitt 26.3.1 auf Seite 156) vom Typ `externSubobject` (`<vector type="externSubobject">`), er enthält die Ausgabewerte.

Ein Beispiel für ein externes Unterobjekt:

```
<subobject number="6">
   <vector type="externSubobject">
      <value>5</value>
      <variable>7</variable>
   </vector>
</subobject>
```

Ein Beispiel für ein externes Unterobjekt mit dem optionalen Attribut `number`:

```
<subobject number="2">
   <vector type="externSubobject">
      <variable number="1">2</variable>
      <value number="2">3.7</value>
      <variable number="3">5</variable>
   </vector>
</subobject>
```

26.3.11 Definitionsbereichseigenschaftselement

Zur Beschreibung des Fib-Elements für Definitionsbereichseigenschaften siehe Abschnitt 12.11 auf Seite 46 .

Das entsprechende XML-Element für Definitionsbereichseigenschaften hat den Namen `domainProperty`. Das XML-Element hat zwei bis drei Attribute. Das erste Attribut hat den Namen `defined_variable` und gibt an, welche Variable vom Definitionsbereichseigenschaftselement definiert wird. Es ist eine natürliche Zahl für eine Variable, welche noch nicht über dem Definitionsbereichseigenschaftselement für eine andere Variable verwendet wird.

Als weiters kann dieses Element ein optionales Attribut `subdomain` enthalten. Das Attribut `subdomain` spezifiziert den Unterefinitionsbereich auf welchen verwiesen wird. Dabei können die Nummern (die Zählung beginnt bei 1) von Unterdefinitionsbereichen auf meheren Stufen durch einen Punkt getrennt angegeben werden. Wobei die Nummer eines Unterdefinitionsbereichs vor der Nummer des in ihm enthaltenden Unterdefinitionsbereichs steht.

Das Attribut `mode` gibt den Type der Eigenschaft an, welche zurückgegeben werden soll. Der Wert diese Attributs kommt aus der Spalte “Name” der Tabelle 4 auf Seite 47 .

Das Fib-Elements für Definitionsbereichseigenschaften enthält zwei XML-Elemente.

Das Erste ist für den Typ des Fib-Elements auf dessen Definitionsbereich verwiesen wird. Es hat den Namen `type` und enthält das XML-Element für den Typ. Dieses XML-Element wird wie in Tabelle 15 auf Seite 148 beschrieben erstellt.

Das Definitionsbereichseigenschaftselement enthält weiterhin das XML-Element für das enthaltende Unterobjekt.

Ein Beispiel für ein Definitionsbereichseigenschaftselement:

```
<domainProperty defined_variable="3" subdomain="2"
      mode="unscaled max">
   <type>
      <dim count="2">
         <dimension number="1" direction="horizontal"/>
```

```
         <dimension number="2" direction="vertical"/>
      </dim>
   </type>
   ...<!-- Fib-Unterobjekt -->
</subobject>
```

Beispiel in dem der erste Unterdefinitionsbereich des dritten Unterdefinitionsbereichs des Matrixelements verwendet wird:

```
<domainProperty defined_variable="1" subdomain="3.1"
      mode="unscaled min">
   <type>
      <matrixElement/>
   </type>
   ...<!-- Fib-Unterobjekt -->
</subobject>
```

Beispiel ohne Unterdefinitionsbereiche:

```
<domainProperty defined_variable="14" mode="scaling">
   <type>
      <inVar number="4"/>
   </type>
   ...<!-- Fib-Unterobjekt -->
</subobject>
```

26.3.12 Set-Element

Zur Beschreibung des Set-Element siehe Abschnitt 12.12 auf Seite 48 .

Der Name des Set-Elements ist `set`. Es hat ein optionales Attribut mit dem Namen `domainNr`, welches die Nummer des Definitionsbereichs für das set-Element angibt. Wenn das Attribut `domainNr` fehlt, ist die die Nummer des Definitionsbereichs 0 .

Das Set-Xml-Element enthält als erstes ein Xml-Element für den definierten Variablen mit dem Namen `defined_variables`, welches für jede Variable, die das Set-Element definiert, in ihrer Reihenfolge ein `variable`-Element enthält. Diese haben das optinale Attribut `number`, welches die Nummer (die Zählung beginnt bei 1) des Elements bzw. der definierten Variable im Set-Element angibt. Weiterhin enthält das `variable`-Element eine natürliche Zahl für die Variable, welche noch nicht über oder im Set-Element verwendet wird.

Nach dem Xml-Element für Variablendefinitionen `defined_variables` folgt ein Xml-Element namens `values`, welches die Vektoren mit den Werten, mit denen die Variablen belegt werden sollen enthält. Sie stehen in ihrer Reihenfolge im Set-Element. Die Vektoren haben alle den Typ `set` (`type="set"`).

Am Ende, nach den Xml-Element für die Wertevektoren `values`, enthält das Set-Element das XML-Element für das enthaltende Unterobjekt.

Ein Beispiel für einen Set-Element:

```
<set>
   <defined_variables>
      <variable>7</variable>
      <variable>8</variable>
      <variable>9</variable>
   </defined_variables>
   <values>
      <vector type="set">
         <variable>1</variable>
         <value>3</value>
         <value>26.14</value>
      </vector>
      <vector type="set">
         <value>33.4</value>
         <value>-47</value>
         <variable>4</variable>
      </vector>
   </values>

   ...<!-- Fib-Unterobjekt -->
</set>
```

Ein Beispiel für einen Set-Element mit dem optionalen Attributen `domainNr` und `number`:

```
<set domainNr="5">
   <defined_variables>
      <variable number="1">2</variable>
      <variable number="2">3</variable>
      <variable number="3">4</variable>
      <variable number="4">5</variable>
   </defined_variables>
   <values>
      <vector type="set">
         <variable>1</variable>
         <variable>1</variable>
         <variable>1</variable>
         <variable>1</variable>
      </vector>
      <vector type="set">
```

```
            <value>8</value>
            <value>4</value>
            <variable>1</variable>
            <value>3</value>
        </vector>
    </values>

    ...<!-- Fib-Unterobjekt -->
</set>
```

26.3.13 Matrixelement

Zur Beschreibung des Matrixelement siehe Abschnitt 12.13 auf Seite 49 .

Der Name des Matrixelement ist `matrix`. Es hat das Attribut `dimensions`, welches angibt, wieviel Dimensionen d das Matrixelement hat. Des weiteren hat es ein weiteres optionales Attribut mit dem Namen `domainNr`, welches die Nummer des Definitionsbereichs für das Matrixelement angibt. Wenn das Attribut `domainNr` fehlt, ist die die Nummer des Definitionsbereichs 0 .

Das Xml-Matrixelement enthält als erstes ein Xml-Element mit dem Namen `defined_variables`, welches für jede Variable, die das Matrixelement definiert, in ihrer Reihenfolge ein `variable`-Element enthält. Dieses hat das optinale Attribut `number`, welches die Nummer (die Zählung beginnt bei 1) des Elements bzw. der definierten Variable im Matrixelement angibt. Weiterhin enthält das `variable`-Element eine natürliche Zahl für die Variable, welche noch nicht über oder im Matrixelement verwendet wird.

Nach den Xml-Element für die Variablendefinitionen `defined_variables` folgt das Xml-Element namens `areas`, welches die Vektoren mit den Bereichen für die Dimensionen $((Startvalue_k, \ldots, Endvalue_k)$ mit $k = 1 \ldots d)$ enthält, mit denen die Dimensionsvariablen $(Variable_1, \ldots, Variable_d)$ belegt werden sollen. Sie stehen in ihrer Reihenfolge im Matrixelement. Die Vektoren haben alle den Typ `area` (`type="area"`).

Nach den Xml-Element für Vektoren mit den Bereichen `areas` folg das Xml-Element namens `values`, welches die Vektoren mit den Werten enthält, mit denen die Wertevariablen $(Value_{d+1}, \ldots, Value_{d+i})$ belegt werden sollen. Sie stehen in ihrer Reihenfolge im Matrixelement. Die Vektoren haben alle den Typ `matrix` (`type="matrix"`).

Am Ende, nach den Xml-Element für die Wertevektoren `values`, enthält das Matrixelement das XML-Element für das enthaltende Unterobjekt.

Ein Beispiel für einen Matrixelement:

```
<matrix dimensions="2">
    <defined_variables>
        <variable>7</variable>
```

```
      <variable>8</variable>
      <variable>9</variable>
      <variable>10</variable>
      <variable>11</variable>
   </defined_variables>
   <areas>
      <vector type="area">
         <value>3</value>
         <value>4</value>
      </vector>
      <vector type="area">
         <variable>2</variable>
         <value>7</value>
      </vector>
   </areas>
   <values>
      <vector type="matrix">
         <variable>1</variable>
         <value>3</value>
         <value>26.14</value>
      </vector>
      <vector type="matrix">
         <value>33.4</value>
         <value>-47</value>
         <variable>4</variable>
      </vector>
   </values>
  ...
   ...<!-- Fib-Unterobjekt -->
</matrix>
```

Ein Beispiel für einen Matrixelement mit dem optionalen Attributen `domainNr` und `number`:

```
<matrix dimensions="2" domainNr="7">
   <defined_variables>
      <variable number="1">8</variable>
      <variable number="2">9</variable>
      <variable number="3">10</variable>
      <variable number="4">11</variable>
   </defined_variables>
   <areas>
      <vector type="area">
         <variable>3</variable>
         <variable>2</variable>
```

```
      </vector>
      <vector type="area">
         <value>-2</value>
         <value>0</value>
      </vector>
   </areas>
   <values>
      <vector type="matrix">
         <value>3</value>
         <variable>1</variable>
      </vector>
      <vector type="matrix">
         <value>0.44</value>
         <value>-7</value>
      </vector>
      <vector type="matrix">
         <value>6</value>
         <value>5</value>
      </vector>
      <vector type="matrix">
         <value>-1</value>
         <value>-1</value>
      </vector>
   </values>
   ...
   ...<!-- Fib-Unterobjekt -->
</matrix>
```

Teil VI
Projektstruktur der Implementation

Das Projekt ist in unterschiedliche Module organisiert. Diese Module sollen weitesgehende separate Einheiten darstellen, deren Beziehungen untereinander klar definiert sind.

Jedes Modul besitzt sein eigenen Namensraum.
Diese Module sind (die Namen der Namensräume werden vorn, vor dem Doppelpunkt angegeben):

- “fib”: die Fib-Multimediasprache (siehe Abschnitt VII auf Seite 175 für den Implementationsentwurf, sowie Abschnitt III auf Seite 22 für die Beschreibung)
- “enviroment”: der allgemeine genetische Algorithmus (siehe Abschnitt VIII auf Seite 366 für den Implementationsentwurf, sowie Abschnitt IV auf Seite 95 für die Beschreibung)
- “operator”: Operatoren für den allgemeine genetische Algorithmus (siehe Abschnitt 78 auf Seite 427)
- “enviroment.fib”: der genetische Algorithmus für Fib (siehe Abschnitt IX auf Seite 432)
- “fib.operator”: Operatoren für den genetische Algorithmus für Fib (siehe Abschnitt 87 auf Seite 448, sowie Abschnitt XIV auf Seite 473 für die Beschreibung einzelner Operatoren)
- “fib.algorithm”: Algorithmen zur Verarbeitung von Fib-Objekten (noch keine Entwürfe vorhanden)
- “fib.converter”: Konvertierer in und aus der Fib-Multimediasprache (siehe Abschnitt XI auf Seite 451)
- “fib.player”: Ein Abspielprogramm für Fib-Multimediaobjekte (noch kein Entwurf vorhanden)

27 Abhängigkeiten der Module

In Abbildung 12 sind die Abhängigkeiten der einzelnen Module dargestellt. Hier wurde der Vererbungspfeil zwischen den “enviroment” und “enviroment.fib” sowie den “operation” und “fib.operator” Modulen benutzt, um darzustellen, dass die entsprechenden Fib-Module Spezialisierungen der “enviroment” sowie “operator” Module darstellen.

Die Abbildung 13 zeigt die Abhängigkeiten der Hauptmodule noch einmal in einer anderen grafisch Form.

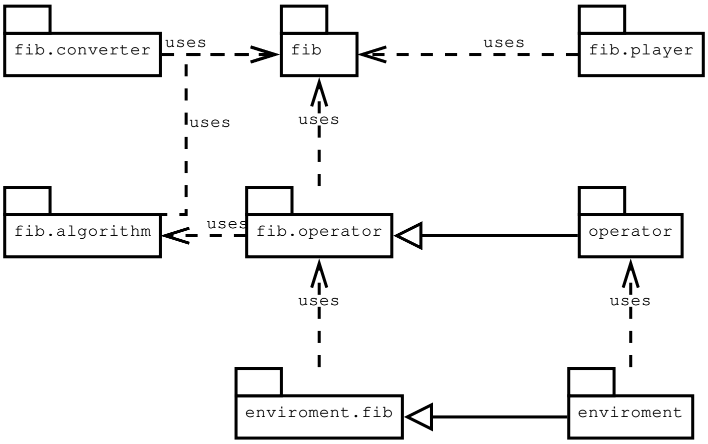


Abbildung 12: Abhängigkeiten der Module

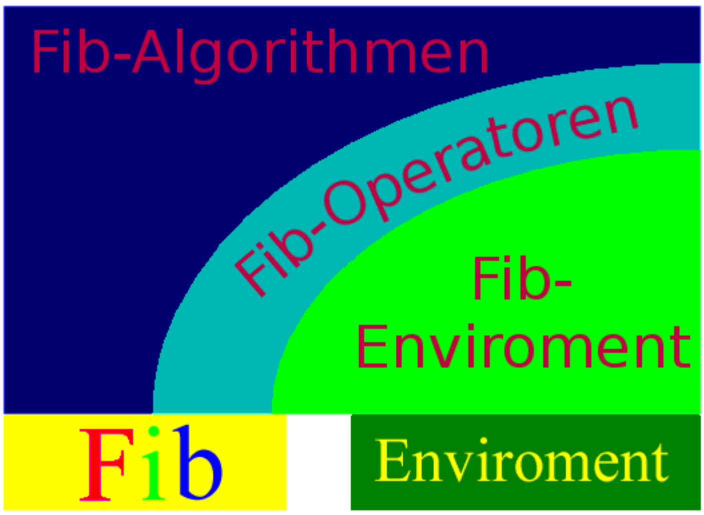


Abbildung 13: Abhängigkeiten der Hauptmodule

Das “enviroment.fib” Modul benötigt zwar für seine Individuen die Fib-Objekte, dies jedoch nur als Namen (für Fib-Objekte). Das “enviroment.fib”-Modul benötigt also nur einen Namen für Fib-Objekte und kein Wissen über die Funktionsweise (Methoden) der Fib-Objekte. Daher ist dass “enviroment.fib”-Modul nicht von “fib”-Modul abhängig.

Teil VII
Implementation Fib-Sprachelemente

In diesem Teil wird der Implementationsentwurf einer Bibliothek für die Fib-Sprachelemente vorgestellt. Dazu gehört die Klassenhierarchien und die Schnittstellenbeschreibungen. Die Schnittstellenbeschreibungen geschieht in pseudo C++.

Alle Fib-Element und Fib-Vektoren werden einem Namensraum mit dem Namen “fib” zugeordnet.

Alle Methoden sollten nach Möglichkeit so definiert werden, dass es unmöglich ist, ein ungültiges Fib-Objekt zu erzeugen. Das macht den Umgang mit den Methoden sicherlich schwieriger, aber auch sicherer.

28 Hilfsmethoden

Hilfsmethoden die bei der Implementierung Hilfreich sind, aber nicht zur eigentlichen Schnittstelle gehören, werden nach Bedarf implentiert, hier aber nicht aufgeführt. Sie sollten nach Möglichkeit von Außen nicht zugreifbar sein (also z. B. `privat` oder `protected` deklariert sein).

29 Copyright

Alle Fib-Sprachelemente werden unter die LGPL (siehe Abschnitt 106 auf Seite 500) gestellt. Auf diese Weise können sie als Bibliothek in beliebige andere Applikationen eingebunden werden.

Dies ist besonders für Anzeigeprogramme/Abspielprogramme wichtig, da so die Copyright des Programms unabhängig von der Fib-Bibliothek ist.

30 Konstanten

Im Fib-Namensraum werden ein paar Konstanten definiert:

- `FIB_VERSION`: Version der Multimediabeschreibungssprache Fib. Es handelt sich hier um eine positive ganze Zahl.
- `FIB_DB_VERSION`: Version der Datenbank der Multimediabeschreibungssprache Fib. Es handelt sich hier um eine positive ganze Zahl.
- `FIB_VERSION_NAME`: Name der Version der Multimediabeschreibungssprache Fib. Es handelt sich um eine Zeichenkette der Form “Fib V1.0.2” .
- `FIB_DB_VERSION_NAME`: Name der Version der Datenbank der Multimediabeschreibungssprache Fib. Es handelt sich hier um eine Zeichenkette der Form “Fib DB V2.3.1”

31 Einfache Datentypen für Zahlen

Als primitive Datentypen für Zahlen werden Fib eigene Namen benutzt, um eine einfache Erweiterbarkeit (z. B. Ausweitung des Definitionsbereichs) zu gewährleisten.

Diese Datentypen sind:

- `intFib`: Ganzzahlen
- `longFib`: natürliche Zahlen
- `unsignedIntFib`: große Ganzzahlen
- `unsignedLongFib`: große natürliche Zahlen
- `doubleFib`: rationale Zahlen

Der Namensteil "int" deutet auf kleine Ganzzahlen hin und "long" auf große. Der "double" Namensanfang steht für Gleitkommazahlen. Beinhaltet der Name "unsigned" sind die Zahlen vorzeichenlos, bzw. nur positiv.

Mögliche Entsprechungen in C++ könnten sein:

- intFib: int
- longFib: long long
- doubleFib: double
- unsignedIntFib: unsigned int
- unsignedLongFib: unsigned long long

32 Klassen für Fib-Elemente

Sie Klasse `cFibElement` ist die Basisklasse aller Fib-Elementklassen.

Es gibt folgende Fib-Elementklassen:

- `cPoint`: für Punktelemente (siehe Abschnitt 12.2 auf Seite 24)
- `cProperty`: für Eigenschaftselemente (siehe Abschnitt 12.3 auf Seite 25)
- `cList`: eine Liste von Fib-Objekten bzw. das Listenelement (siehe Abschnitt 12.4 auf Seite 31)
- `cComment`: für Anmerkungselemente (siehe Abschnitt 12.5 auf Seite 32)
- `cArea`: für Bereichselement (siehe Abschnitt 12.6 auf Seite 34)

- `cFunction`: für Funktionselement (siehe Abschnitt 12.7 auf Seite 35)
- `cIf`: für if-Bedingungselemente (siehe Abschnitt 12.8 auf Seite 41)
- `cExtObject`: für externe Objekte (siehe Abschnitt 12.9 auf Seite 43)
- `cExtSubobject`: für externe Unterobjekte (siehe Abschnitt 12.10 auf Seite 44)
- `cRoot`: für Wurzelelemente (siehe Abschnitt 12.14 auf Seite 52)

Die Fib-Elementklassen werden Klassen untergeordnet, welche ihre Struktur beschreiben:

- `cFibLeaf`: für Fib-Elementklassen die Blätter sind, also keine Unterobjekte enthalten; die abgeleitete Klassen sind:
  - `cPoint`
  - `cExtSubobject`
- `cFibLimb`: für Fib-Elementklassen die (gerade) Zweige sind, also genau ein Unterobjekt enthalten; die abgeleitete Klassen sind:
  - `cProperty`
  - `cFibList`
  - `cComment`
  - `cArea`
  - `cFunction`
- `cFibBranch`: für Fib-Elementklassen die Verzweigungen darstellen, also mehrere Unterobjekte haben können; die abgeleitete Klassen sind:
  - `cFibList`
  - `cIf`
  - `cRoot`
  - `cExtObject`

32.1 Klassenhierarchie

In Abbildung 14 ist die Vererbungsstruktur der Klassen für Fib-Elemente zu sehen. Wie zu sehen werden alle Fib-Elementklassen von der Klasse `cFibElement` abgeleitet.

33 Fib-Elemente Klasse cFibElement

Die Klasse `cFibElement` ist die Basisklasse aller Fib-Element. Sie ist eine abstrakte Klasse, von ihr selbst können keine Objekte erzeugt werden.

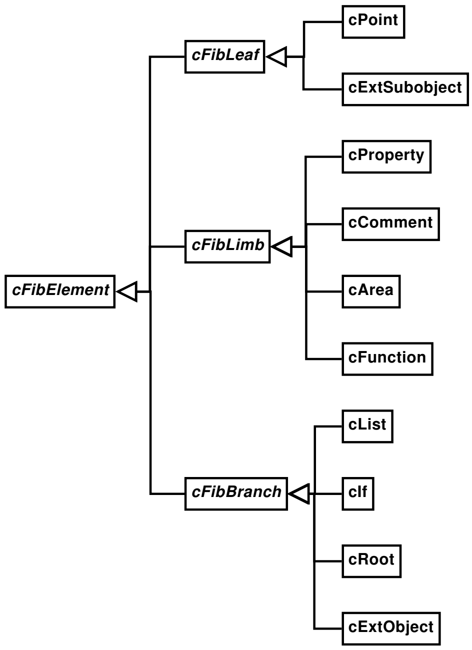


Abbildung 14: Klassengraph Fib-Elemente

33.1 Schnittstellenbeschreibung

Im Nachfolgenden werden die Methoden und Funktionen beschrieben, welche von allen Fib-Elementen bereitgestellt und von der Klasse `cFibElement` deklariert werden. Diese Methoden und Funktionen dienen hauptsächlich dazu, die Struktur eines Fib-Objektbaums zu analysieren und anzupassen.

Externe Unterobjekte werden nur von Methoden aufgelöst, bei denen es explizit erwähnt wird.

Alle Zahlen mit denen Fib-Elemente oder Objekte identifiziert werden (siehe Abschnitte über die Ordnung der Fib-Elemente ab Seite 75), werden bezüglich des Fib-Elements angegeben, bei dem die entsprechende Methode aufgerufen wird, und nicht dem obersten root-Element. Soll beispielsweise das dritte Punktelement zurückgegeben werden, wird das dritte Punktelement bezüglich des Fib-Elements zurückgegeben, an dem die entsprechende Methode aufgerufen wurde, und nicht bezüglich des obersten root-Elements, für welches dieses dritte Punktelement bezüglich des Fib-Elements das neunte Punktelement sein könnte. Dies gilt natürlich nur, wenn nicht explizit ein anderer Bezugspunkt angegeben wird.

33.1.1 Typ edDirection

Der Typ `edDirection` dient zur Angabe einer Richtung im Fib-Objekt. Diese Angabe sollte zu einem Referenz-Fib-Element im Fib-Objekt geschehen. (Zur

Definition von “Oben” und “Unten” siehe Abschnitt 14.3 auf Seite 74 .)

Gültige Werte sind:

- `ED_ALL`: in alle Richtungen im Fib-Objekt vom Referenz-Fib-Element ausgehend
- `ED_POSITION`: das Referenz-Fib-Element
- `ED_BELOW`: alle Fib-Elemente unter dem Referenz-Fib-Element
- `ED_HIGHER`: alle Fib-Elemente über dem Referenz-Fib-Element
- `ED_BELOW_EQUAL`: alle Fib-Elemente unter dem Referenz-Fib-Element und das Referenz-Fib-Element
- `ED_HIGHER_EQUAL`: alle Fib-Elemente über dem Referenz-Fib-Element und das Referenz-Fib-Element

33.1.2 getType

Syntax: `char getType() const`

Über die Methode kann der Typ des Objekts bzw. des Fib-Elements ermittelt werden.

Mögliche Typen sind:

’u’: (=117) unbekannten Type; als Parameter einer Methode steht dieser Wert für Fib-Elemente aller Typen

’p’: (=112) für Punkt-Elemente (cPoint)

’y’: (=121) für Eigenschaftselemente (cProperty)

’l’: (=108) für Listenelemente (cListen)

’c’: (=110) für Anmerkungselemente (cComment)

’a’: (=97) für Bereichselemente (cArea)

’f’: (=102) für Funktionselemente (cFunction)

’i’: (=105) für if-Bedingungen (cIf)

’o’: (=111) für externe Objekte (cExtObject)

’s’: (=115) für externe Unterobjekte (cExtSubobject)

’v’: (=118) für set-Elemente (cFibSet)

'm': (=109) für Matrixelement (cFibMatrix)

'r': (=114) für Wurzelelemente (cRoot)

Eingabeparameter: keine

Rückgabe: Es wird eine Konstante zum Identifizieren des Objekts zurückgegeben.

33.1.3 getTypeName

Syntax: `static string getTypeName( char cType ) const`

Diese Methode wandelt das Typezeichen eines Fib-Elements in dessen Bezeichnung um.

Eingabeparameter:

- `cType`: Das Typzeichen, welches umzuwandeln ist.

Rückgabe: Der Text für die Elementbezeichnung für den eingegeben Typen:

Typzeichen	Elementbezeichnung
'u'	"unknown"
'p'	"point"
'y'	"property"
'l'	"list"
'm'	"comment"
'a'	"area"
'f'	"function"
'i'	"if"
'o'	"extern object"
's'	"extern subobject"
'r'	"root"

33.1.4 getSuperiorFibElement

Syntax: `cFibElement* getSuperiorFibElement()`

Die Methode holt das nächste Fib-Element im Fib-Baum, welches dieses Fib-Element enthält. Wenn kein solches Fib-Element existiert, wird ein Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf das nächste Fib-Element, welches dieses Fib-Element enthält, oder der Nullpointer `NULL`, wenn kein solches Fib-Element existiert.

33.1.5 getNextFibElement

Syntax: `cFibElement* getNextFibElement()`

Die Methode holt das nächste Fib-Element im Fib-Baum. Wenn kein solches Fib-Element existiert, wird ein Nullpointer `NULL` zurückgegeben. Die Reihenfolge bezieht sich auf die Ordnung der Fib-Elemente.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf das nächste Fib-Element oder der Nullpointer `NULL`, wenn kein solches Fib-Element existiert.

33.1.6 getNextFibElement

Syntax: `cFibElement* getNextFibElement( char cType )`

Die Methode holt das nächste Fib-Element im Fib-Baum vom angegeben Typ. Wenn kein solches Fib-Element existiert, wird ein Nullpointer `NULL` zurückgegeben. Die Reihenfolge bezieht sich auf die Ordnung der Fib-Elemente.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, welches zurückgegeben werden soll. Der Typ 'w' steht für das nächste Falsche (="wrong") Fib-Element, also ein Fib-Element, welches noch nicht korrekt mit Parametern beladen ist (es fehlt beispielsweise ein Unterobjekt).

Rückgabe: Einen Zeiger auf das nächste Fib-Element mit dem angegeben Typ oder der Nullpointer `NULL`, wenn kein solches Fib-Element existiert.

33.1.7 getFibElement

Syntax: `cFibElement* getFibElement( longFib lNumber, bool bAbsolute=false )`

Die Methode holt das `lNumber`'te Fib-Element. Wenn kein solches Fib-Element existiert, wird ein Nullpointer `NULL` zurückgegeben.

Eingabeparameter:

- `lNumber`: Die Nummer, die das zu holende Fib-Element im aktuellen Fib-Baum/Objekt hat.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` ist, bezieht sich die `lNumber` auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` ist, bezieht sich `lNumber` auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Einen Zeiger auf das Fib-Element, vom Typ des aktuellen Fib-Elements, welches an der angegebene Position `lNumber` steht oder der Nullpointer `NULL`, wenn kein solches Fib-Element existiert.

33.1.8 getFibElement

Syntax:
```
cFibElement* getFibElement(
 char cType, longFib lNumber,
 bool bAbsolute=false )
```

Die Methode holt das `lNumber`'te Fib-Element von dem angegebenen Typ `cType`. Wenn kein solches Fib-Element existiert, wird ein Nullpointer `NULL` zurückgegeben.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, welches zurück geliefert werden soll. Der Typ 'w' steht für das nächste Falsche (="wrong") Fib-Element, also ein Fib-Element, welches noch nicht korrekt mit Parametern beladen ist (es fehlt beispielsweise ein Unterobjekt).
- `lNumber`: Die Nummer, die das zu holende Fib-Element im aktuellen Fib-Baum/Objekt hat.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` ist, bezieht sich die `lNumber` auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` ist, bezieht sich `lNumber` auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Einen Zeiger auf das Fib-Element, vom angegeben Typ `cType`, welches an der angegebene Position `lNumber` steht oder der Nullpointer `NULL`, wenn kein solches Fib-Element existiert.

33.1.9 getAllFibElements

Syntax:
```
list<cFibElement*> getAllFibElements(
 char cTypeBasis='u',
 longFib lNumber=0, char cType='u',
 edDirection direction=ED_ALL , unsignedLongFib
 lNumberOfMaxReturnedElements=0
 bool bAbsolute=false )
```

Die Methode gibt eine Liste mit den Zeigern zu den Fib-Elemente vom angegebenen Typ `cType` zurück, welche sich in der Richtung `direction` (siehe 33.1.1 Seite 178) vom `lNumber`'te Objekt vom angegeben Typ `cTypBasis` befinden.

Dabei werden niemals mehr als lNumberOfMaxReturnedElements Fib-Element zurückgegeben, wobei der Wert 0 bei lNumberOfMaxReturnedElements für unendlich steht. Alle zurückgegebenen Fib-Elemente stammen aus dem Ast, in dem sich auch das Referenz-Fib-Element befindet.

Wenn von mehreren Richtungen oder/und Unterobjekten Fib-Elemente zurückzugeben sind, wird versucht von jeder Richtung oder/und Unterobjekt die gleiche Anzahl Fib-Elemente zurückzugeben.

Eingabeparameter:

- `cTypeBasis`: Der Typ, welchen das Referenz-Fib-Element im aktuellen Fib-Baum/Objekt hat. Standardwert ist 'u', womit Fib-Elemente aller Typen (bzw. keines bestimmten Typs) betrachtet werden.
- `lNumber`: Die Nummer, welches das Referenz-Fib-Element im aktuellen Fib-Baum/Objekt hat. Standardwert ist 0, womit alle Fib-Elemente vom aktuellen Fib-Element ausgehend zurückgeliefert werden, unabhängig vom angegeben Typ `cTypeBasis`.
- `cType`: Der Typ der Fib-Elemente, welche zurück geliefert werden soll. Standardwert ist 'u', womit Fib-Elemente aller Typen (bzw. keines bestimmten Typs) zurückgegeben werden. Der Typ 'w' steht für Falsche (="wrong") Fib-Elemente, also Fib-Elemente, welche noch nicht korrekt mit Parametern beladen sind (es fehlt beispielsweise ein Unterobjekt).
- `direction`: Die Richtung vom Referenz-Fib-Element ausgehend, von der alle Fib-Elemente mit dem angegeben Typ zurückzugeben sind. Standardwert ist hierfür `ED_ALL`, so dass Fib-Elemente aus dem gesamten Ast des Referenz-Fib-Elements zurückgegeben werden.
- `lNumberOfMaxReturnedElements`: Die Anzahl der maximal zurückzugebenden Fib-Elemente. Ist der Wert 0, gibt es keine Begrenzung für die Anzahl der zurückzugebenden Fib-Elemente. Standardwert ist 0 (=unbegrenzt).
- `bAbsolute`: Wenn `bAbsolute` gleich `true` ist, bezieht sich die `lNumber` auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` ist, bezieht sich `lNumber` auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Eine Liste mit den entsprechenden Fib-Elementen.

33.1.10 evalueObject

Syntax: `bool evalueObject(`
`iEvaluePosition & evaluePosition,`

```
const unsignedIntFib objectPoint=0,
list<cVectorProperty> &liVecProperties=
list<cVectorProperty>() ) const
```

Über die Methode `evalueObject` können Fib-Objekte ausgewertet werden.

Der Methode wird eine Referenz auf ein Object (`evaluePosition`) vom Type `iEvaluePosition` übergeben, über welche die einzelnen Punkt mit ihren Eigenschaften ausgewertet werden. Jedes mal, wenn ein Punkt dargestellt/ ausgewertet werden soll, wird die Methode `evaluePosition()` des Objects mit der Position des Punktes und einer Liste seiner Eigenschaften aufgerufen. Auf diese Weise kann die Methode `evalueObject` für verschiedene Aufgaben genutzt werden. Soll ein Objekt als Bild ausgewertet werden, stellt die Methode `evaluePosition()` einen Punkt im Bild dar. Wenn das Objekt mit einem anderen Multimediaobjekt verglichen werden soll, vergleicht die Methode `evaluePosition()` die einzelnen Punkte (Vorsicht ist geboten, wenn ein Punkt einen anderen überdeckt). Die Methode `evaluePosition()` muss nur entsprechend implementiert werden. Gibt ein Aufruf der Methode `evaluePosition()` false zurück, wird die Auswertung des Fib-Objekts mit `evalueObject` abgebrochen. Näheres zu Objekten, welche das Interface `iEvaluePosition` implementieren im Abschnitt 34 auf Seite 214 .

Von der Methode `evalueObject` werden auch externe Objekte aufgelöst. Auf diese Weise wird das komplette Multimediaobjekt ausgewertet, welches des Fib-Objekt darstellt.

Eingabeparameter:

- `evaluePosition`: Eine Referenz auf ein Object zum auswerten/ speichern der einzelnen Punkte mit ihren Eigenschaften. Dieses Object stellt die Methode `evaluePosition()` bereit, der für jeden Punkt, der ausgewertet werden soll, die Position und eine Liste mit Eigenschaften des Punktes übergeben wird.
- `objectPoint`: Der Objektpunkt des echten Teilobjekts, welches ausgewertet werden soll. Standardmäßig wird dieser Wert auf 0 gesetzt und damit das gesamt Objekt ausgewertet.
- `liVecProperties`: Eine Liste mit den Eigenschaften, die das auszuwertende Objekt global besitzen soll. Diese Eigenschaften werden jeden Punkt zugeordnet, soweit sie nicht überschrieben werden. Standardmäßig wird hier eine leere Liste, also keine globalen Eigenschaften, übergeben.

Rückgabe: Diese Methode gibt `true` (=wahr) zurück, wenn die Auswertung des Objekts erfolgreich war, sonst `false` (=falsch).

33.1.11 evalueObject

Syntax: `bool evalueObject(`
```
 iEvalueFibElement & evalueFibElement,
 const unsignedIntFib objectPoint=0,
 list<cVectorProperty> &liVecProperties=
 list<cVectorProperty>(),
 conat list<char> liCFibElementTyps )
```

Über die Methode `evalueObject` können Fib-Objekte ausgewertet werden.

Der Methode wird eine Referenz auf ein Object (`evalueFibElement`) vom Type `iEvalueFibElement` übergeben, über welche die einzelnen Fib-Elemente mit ihren Eigenschaften ausgewertet werden. Jedes mal, wenn ein Fib-Elemente dargestellt/ ausgewertet werden soll, wird die Methode `evalueElement()` des Objects mit einem Zeiger auf des Fib-Elemente und einer Liste seinen Eigenschaften aufgerufen. Auf diese Weise kann die Methode `evalueObject` für verschiedene Aufgaben genutzt werden. Gibt ein Aufruf der Methode `evalueElement()` false zurück, wird die Auswertung des Fib-Objekts mit `evalueObject` abgebrochen. Näheres zu Objekten, welche das Interface `iEvalueFibElement` implementieren im Abschnitt 35 auf Seite 215 .

Über die Liste `liCFibElementTyps` wird bestimmt, welche Fib-Element zurückgegeben werden sollen. Für jedes Fib-Element das zurückgegeben werden soll, steht in `liCFibElementTyps` sein Typzeichen (siehe Abschnitt 33.1.2 auf Seite 179). Punktelemente mit Positionsvektoren werden immer zurückgegeben. Diese müßen nicht seperat in `liCFibElementTyps` aufgeführt werden.

Eingabeparameter:

- `evalueFibElement`: Eine Referenz auf ein Object zum auswerten/ speichern der einzelnen Fib-Elemente mit ihren Eigenschaften. Dieses Object stellt die Methode `evalueElement()` bereit, der für jedes Fib-Element, mit einem Type aus der Liste `liCFibElementTyps` oder das ein Punkt ist, ein Zeiger auf das Fib-Element und eine Liste mit Eigenschaften des Fib-Elements übergeben wird.

- `objectPoint`: Der Objektpunkt des echten Teilobjekts, welches ausgewertet werden soll. Standardmäßig wird dieser Wert auf 0 gesetzt und damit das gesamt Objekt ausgewertet.

- `liVecProperties`: Eine Liste mit den Eigenschaften, die das auszuwertende Objekt global besitzen soll. Diese Eigenschaften werden jeden Blattelement zugeordnet, soweit sie nicht überschrieben werden. Standardmäßig wird hier eine leere Liste, also keine globalen Eigenschaften, übergeben.

- `liCFibElementTyps`: Die Liste mit den Typzeichen für die Fib-Element, für die `auf()` aufgerufen werden soll. Die möglichen Zeichen werden in Abschnitt 33.1.2 auf Seite 179 aufgeführt.

Rückgabe: Diese Methode gibt `true` (=wahr) zurück, wenn die Auswertung des Objekts erfolgreich war, sonst `false` (=falsch).

33.1.12 getTimeNeed

Syntax: `unsignedLongFib getTimeNeed( unsignedLongFib lMaxTime=0 ) const`

Diese Methode berechnet einen Wert für die Zeit, welche zur Auswertung des Fib-Objekts benötigt wird.

Von der Methode `getTimeNeed` werden auch externe Objekte aufgelöst und in der Berechnung berücksichtigt.

Der Eingabeparameter `lMaxTime` dient dazu, den Aufwand zum Berechnen der Zeit zu begrenzen. Wenn ein Fib-Objekt sehr Zeitaufwändig auszuwerten ist, würde auch das Ermitteln der Zeitstatistik entsprechend lange dauern. Mit dem Parameter `lMaxTime` wird die Zeit zur Berechnung der Zeitstatistik begrenzt bzw. abgebrochen, wenn `lMaxTime` erreicht wurde. Wenn der Wert `lMaxTime` nicht angegeben wird oder 0 ist, wird die Berechnungszeit nicht begrenzt.

Nützlich ist diese Begrenzung, da eine genetische Operation die Auswertungszeit sehr stark erhöhen kann. Durch die Begrenzung kann die Zeit zum Ermitteln der Zeitstatistik begrenzt werden und alle Individuen, die diese Grenze erreichen sofort verworfen/ gelöscht werden.

In Tabelle 18 sind die Formeln für die benötigten Zeiten für die einzellnen Elemente aufgeführt. Dabei bezeichnet T die Auswertungszeit eines Fib-Elements und $T(Obj_i)$ die Auswertungszeit des i'ten enthaltenden Fib-Objekts.

Fib-Elementklasse	Zeitformel	Anmerkung
`cPoint`	$T(Obj) = 1 + E_V$	E_V ist die Anzahl der Elemente des Positionsvektors
`cProperty`	$T(Obj) = 1 + E_V + T(Obj_1)$	E_V ist die Anzahl der Elemente des Eigenschaftsvektor; $T(Obj_1)$ die Auswertungszeit des enthaltenden Fib-Objekts Obj_1
`cFibList`	$T(Obj) = 1 + \sum_{i=1}^{n} T(Obj_i)$	$T(Obj_i)$ sind die Auswertungszeiten der enthaltenden Fib-Unterobjekte Obj_i; n ist die Anzahl der enthaltenden Fib-Unterobjekte
`cComment:`	$T(Obj) = 1 + T(Obj_1)$	$T(Obj_1)$ ist die Auswertungszeit des enthaltenden Fib-Objekts Obj_1

Fib-Element-klasse	Zeitformel	Anmerkung
`cArea:`	$T(Obj) = 1 + \sum_{W \in B_1 \dots B_n} T(Obj_1(W))$	B_i sind die Unterbereiche des Bereichs, mit n der Anzahl der Unterbereiche; W sind dann die einzelnen Werte der Bereiche; $T(Obj_1(W))$ ist die Auswertungszeit des enthaltenden Fib-Objekts Obj_1, wenn die Variable des Bereichselements auf W gesetzt wurde
`cFunction`	$T(Obj) = T(UF) + T(Obj_1(W))$	$T(UF)$ ist die Zeit, welche zu Berechnen der Unterfunktion benötigt wird; $T(Obj_1(W))$ ist die Auswertungszeit des enthaltenden Fib-Objekts Obj_1, wenn die Variable des Funktionselements auf W, den Wert der Funktion, gesetzt wurde
`cIf`	$T(Obj) = T(UB) + if(UB == true)\{T(Obj_1)\}else\{T(Obj_2)\}$	$T(UB)$ ist die Zeit, welche zu Berechnen der Unterbedingung benötigt wird; $T(Obj_1)$ und $T(Obj_2)$ die Auswertungszeiten der enthaltenden Fib-Objekte Obj_1 und Obj_2, es wird jeweils die Auswertungszeit desjenigen Objekts hinzuaddiert, welches bei der Unterbedingungsauswertung auch ausgewertet würde (als Obj_1 wenn die Unterbedingung wahr ist und sonst Obj_2)
`cExtObject`	$T(Obj) = 5 + T(Obj_1)$	$T(Obj_1)$ ist die Auswertungszeit des Fib-Objekts Obj_1, welches durch das externe Objekte repräsentiert wird
`cExtSub-object`	$T(Obj) = 2 + T(Obj_1)$	$T(Obj_1)$ ist die Auswertungszeit des Fib-Objekts Obj_1, welches hier jeweils eingesetzt bzw. extern bereitgestellt wird

Fib-Element-klasse	Zeitformel	Anmerkung
`cFibSet`	$T(Obj) = n * \sharp DefVar + \sum_{V \in V_1 \dots V_n} T(Obj_1(V))$	$\sharp DefVar$ ist die Anzahl der vom set-Element definierten Variablen;V_i sind die Vektoren mit den zu setzenden Werten des set-Elements, mit n der Anzahl der Vektoren; V sind dann die einzelnen Vektoren; $T(Obj_1(V))$ ist die Auswertungszeit des enthaltenden Fib-Objekts Obj_1, wenn die definierten Variable des set-Elements auf die entsprechenden Elementwerte von V gesetzt wurden

Fib-Element-klasse	Zeitformel	Anmerkung
`cFibMatrix`	$T(Obj) = if(inot0)\{ \sum_{V \in V_1 \dots V_k}(d + i + T(Obj_1(D(V), V))) \}else\{ \sum_{v_1=Startvalue_1 \dots Endvalue_1} \cdots sum_{v_d=Startvalue_d \dots Endvalue_d}(d + T(Obj_1(v_1, \dots, v_d)))\}$	Die Zeitbereichnung von Matrixelementen ist davon Abhängig, ob Variablen für die Werte vorhanden sind ($0 < i$). Sind Variablen für die Werte vorhanden ($0 < i$), wird für jeden Wertevektor $V = (W_{1.b}, \dots, W_{n.b})$ das enthaltende Objekt Obj_1 mit aufgerufen / ausgewertet bzw. die Zeit für diesen Aufruf addiert. Dabei werden die Wertevariablen mit den entsprechenden Werten belegt und die Dimensionsvariablen mit den entsprechend hochgezählten Werten $D(V)$ (in $Obj_1(D(V), V)$). Zu der Zeit des Aurufs des enthaltende Objekts Obj_1 wird noch die Zeit zur Belegung der Variablen addiert ($d + i$). Wenn keine Variablen für die Werte vorhanden sind ($i == 0$), durchlaufen die Dimensionsvariablen jeweils ihre Bereiche, wobei für jede Belegung das enthaltende Objekt ausgewertet wird. Die Zeit ist also für jede Dimensionsvariablenbelegung die Zeit für die Belegung dieser d plus die Auswertungzeit des enthaltende Objekts für diese Belegung.
`cRoot`	$T(Obj) = 3 + T(Obj_1)$	$T(Obj_1)$ ist die Auswertungszeit des enthaltenden Haupt-Fib-Objekts Obj_1
nullstellige Unterfunktionen	$T(UF) = 1$	Für jede Unterfunktionen oder Unterbedingungen wird jeweils 1 hinzuaddiert
einstellige Unterfunktionen	$T(UF) = 1 + T(UF_1)$	UF_1 ist die enthaltende Unterfunktion

Fib-Element-klasse	Zeitformel	Anmerkung
zweistellige Unterfunktionen	$T(UF) = 1 + T(UF_1) + T(UF_2)$	UF_1 und UF_2 sind die enthaltenden Unterfunktionen
nullstellige Unterbedingungen	$T(UB) = 1$	Für jede Unterfunktionen oder Unterbedingungen wird jeweils 1 hinzuaddiert
einstellige Unterbedingungen	$T(UB) = 1 + T(UB_1)$	UB_1 ist die enthaltende Unterbedingung
zweistellige Unterbedingungen	$T(UB) = 1 + T(UB_1) + T(UB_2)$	UB_1 und UB_2 sind die enthaltenden Unterbedingungen
Vergleiche	$T(UB) = 1 + T(UF_1) + T(UF_2)$	UF_1 und UF_2 sind die enthaltenden Unterfunktionen

Tabelle 18: Auswertungszeiten $T(Obj)$ der Fib-Objekts Obj

Eingabeparameter:

- `lMaxTime`: Der Wert `lMaxTime` der bei der Zeitberechnung nicht überschritten werden soll. Ist `lMaxTime=0` (Standardwert), wird die Zeitberechnung nicht begrenzt.

Rückgabe: Diese Methode berechnet einen Wert für die Zeit, welche zur Auswertung des Fib-Objekts benötigt wird.

33.1.13 getCompressedSize

Syntax: `unsignedLongFib getCompressedSize() const`

Diese Methode gibt die Anzahl der Bits für die Größe des Fib-Objekts zurück. Dabei wird für jedes Element (Fib-Element, Fib-Vektoren, usw.) ein Wert aufaddiert, der den Speicherplatzbedarf in Bit beim komprimierten Abspeichern des Elements entspricht (siehe Abschnitt 25 auf Seite 105).

Bei der Berechnung der Größe werden keine externen Objekte aufgelöst, sondern nur die Struktur des aktuellen Fib-Objekts ausgewertet. Weiterhin werden die optionale Information ($Optionalpart$) der root-Elemente nicht mit eingerechnet, da sie als optionale Informationen weggelassen werden können.

Eingabeparameter: keine

Rückgabe: Diese Methode liefert einen Wert für die Größe in Bits des Fib-Objekts zurück.

33.1.14 isUsedVariable

Syntax: `bool isUsedVariable( const cFibVariable *variable, edDirection direction=ED_POSITION ) const`

Diese Methode prüft, ob die gegebene Variable `variable` in den Fib-Elementen, in der angegeben Richtung `direction` vom Fib-Element (von dem die Methode aufgerufen wird) ausgehend, benutzt wird (z. B. in einem Vektor vorkommt).

Diese Methode kann beispielsweise dazu verwendet werden, um zu Prüfen, ob ein Fib-Element, das eine Variable definiert, gelöscht werden kann.

Eingabeparameter:

- `variable`: Die Variable für die geprüft werden soll, ob sie benutzt wird.
- `direction`: Die Richtung, vom aktuellem Fib-Element ausgehend, in der sich die zu prüfenden Fib-Elemente befinden. Standardmäßig wird nur das aktuelle Fib-Element geprüft.

Rückgabe: Die Methode gibt `true` (=wahr) zurück, wenn die Variable in den Fib-Elementen benutzt wird, sonst `false` (=falsch).

33.1.15 getUsedVariables

Syntax: `set<cFibVariable*> getUsedVariables( edDirection direction=ED_POSITION )`

Diese Methode gibt die Menge aller Variablen zurück, die in den Fib-Elementen in der angegeben Richtung `direction` vom Fib-Element, von dem die Methode aufgerufen wird, ausgehend, benutzt werden (bzw. in einem Vektor vorkommt).

Eingabeparameter:

- `direction`: Die Richtung, vom aktuellem Fib-Element ausgehend, in der sich die Fib-Elemente befinden, von denen die benutzten Variablen zurückgegeben werden sollen. Standardmäßig wird nur das aktuelle Fib-Element geprüft.

Rückgabe: Diese Methode gibt die Menge aller Variablen zurück, die in den Fib-Elementen in der angegeben Richtung `direction` vom Fib-Element, von dem die Methode aufgerufen wird, ausgehend, benutzt werden (bzw. in einem Vektor vorkommt).

33.1.16 isDefinedVariable

Syntax: `bool isDefinedVariable(`
` const cFibVariable *variable,`
` edDirection direction=ED_POSITION) const`

Diese Methode prüft, ob die gegebene Variable `variable` in einem Fib-Element definiert ist, welches in der angegeben Richtung `direction` vom aktuellen Fib-Element ausgehend stehen.

Variablen werden beispielsweise von Bereichs- oder Funktionselementen definiert.

Variablen, die ein root-Element definiert, liegen nicht über den in diesem root-Element enthaltenden unter-root-Elemente. Daher kann keine Variable über einem root-Element definiert sein. Dies liegt daran, dass root-Element die Variablen, die sie definieren, nicht an enthaltende root-Elemente vererben, daher können sie auch nicht in enthaltenden root-Elementen benutzt werden.

Eingabeparameter:

- `variable`: Die Variable für die geprüft werden soll, ob sie definiert wird.
- `direction`:Die Richtung, vom aktuellem Fib-Element ausgehend, in der sich die zu prüfenden Fib-Elemente befinden. Standardmäßig wird das aktuelle Fib-Elemente geprüft.

Rückgabe: Die Methode gibt `true` (=wahr) zurück, wenn die Variable in einem der geprüften Fib-Elemente definiert wird, sonst `false` (=falsch).

33.1.17 getDefinedVariables

Syntax: `list<cFibVariable*> getDefinedVariables(`
` edDirection direction=ED_HIGHER )`

Diese Methode gibt die Menge aller Variablen zurück, die in den Fib-Elementen in der angegeben Richtung `direction` vom Fib-Element, von dem die Methode aufgerufen wird, ausgehend, definiert werden.

Variablen, die ein root-Element definiert, liegen nicht über den in diesem root-Element enthaltenden unter-root-Elemente. Daher kann keine Variable über einem root-Element definiert sein. Dies liegt daran, dass root-Element die Variablen, die sie definieren, nicht an enthaltende root-Elemente vererben, daher können sie auch nicht in enthaltenden root-Elementen benutzt werden.

Eingabeparameter:

- `direction`: Die Richtung, vom aktuellem Fib-Element ausgehend, in der sich die Fib-Elemente befinden, von denen die definierten Variablen zurückgegeben werden sollen. Standardmäßig werden alle Fib-Element, die das aktuelle Fib-Element direkt oder indirekt enthalten, geprüft.

Rückgabe: Diese Methode gibt eine Liste aller Variablen zurück, die in den Fib-Elementen in der angegeben Richtung `direction` vom Fib-Element, von dem die Methode aufgerufen wird, ausgehend, definiert werden.

33.1.18 replaceVariable

Syntax: `bool replaceVariable( cFibVariable *variableOld, cFibVariable *variableNew )`

Diese Methode ersetzt im Fib-Object die Vorkommen der Variable `variableOld` durch die Variable `variableNew`.

Dabei bleiben Definitionen von Variablen `variableOld` unberührt. Es werden also nur Vorkommen geändert an denen die Variable `variableOld` benutzt wird.

Eingabeparameter:

- `variableOld`: Die Variable, die ersetzt werden soll.
- `variableNew`: Die Variable, welche die Variable `variableOld` ersetzen soll.

Rückgabe: Die Methode gibt `true` (=wahr) zurück, wenn die `variableOld` Variable dem Fib-Object durch die Variable `variableNew` ersetzt wurde, sonst `false` (=falsch).

33.1.19 getNumberOfElement

Syntax: `unsignedIntFib getNumberOfElement( bool bOfType=false )const`

Diese Methode gibt die Nummer des Fib-Elements in der Ordnung der Fib-Elemente zurück.

Eingabeparameter:

- `bOfType`: Wenn `bOfType` falsch (=`false`) ist, wird die Nummer bezüglich der Ordnung aller Fib-Elemente zurückgegeben. Ist dagegen `bOfType` wahr (=`true`), wird die Nummer bezüglich der Ordnung der Fib-Elemente vom gleichem Typ wie das aktuelle Fib-Element zurückgegeben. Der Standardwert ist falsch (=`false`),

Rückgabe: Die Nummer des Fib-Elements in der Ordnung der Fib-Elemente.

33.1.20 getNumberOfMovePoint

Syntax: `unsignedIntFib getNumberOfMovePoint() const`

Gibt die Nummer des Fib-Elements in der Ordnung der Verschiebungspunkte zurück.

Verschiebungspunkte, sind Stellen (Positionen von Fib-Elementen) von denen und zu denen Fib-Elemente verschoben werden können. Dies sind in der Regel alle (geraden) Astelemente `cFibLimb` (siehe Abschnitt 38 auf Seite 216). Beispielsweise sind alle Bereichsobjekte Verschiebungspunkte, Punktobjekte hingegen nicht. Zm Verschieben von Fib-Elementen siehe auch Abschnitt 33.1.37 auf Seite 206 .

Es wird 0 zurückgegeben, wenn das aktuelle Fib-Element kein Verschiebepunkt ist, bzw. nicht verschoben werden kann.

Eingabeparameter: keine

Rückgabe: Die Nummer des Fib-Elements in der Ordnung der Verschiebungspunkte oder 0, wenn das Fib-Element kein Verschiebepunkt ist.

33.1.21 getNumberOfObjectPoint

Syntax: `unsignedIntFib getNumberOfObjectPoint() const`

Diese Methode gibt die Nummer des zusammenhängenden Teilobjekte in der Ordnung der zusammenhängenden Teilobjekte zurück. Das zusammenhängenden Teilobjekte, auf das sich bezogen wir, ist das zusammenhängenden Teilobjekte, welches über dem aktuellen Fib-Element als nächstes über ein Fib-Unterobjekt definiert ist. Also das Teilobjekt, welches das aktuelle Fib-Element enthält, aber kein zusammenhängendes Teilobjekt enthält, welches auch das aktuelle Fib-Element enthält.

Zur Beschreibung der Ordnung von zusammenhängenden Teilobjekte siehe Abschnitt 14.8 auf Seite 82 .

Eingabeparameter: keine

Rückgabe: Die Nummer in der Ordnung der zusammenhängenden Teilobjekte des nächsten Teilobjekts in dem das aktuelle Fib-Element vorhanden ist.

33.1.22 getNumberOfElements

Syntax:
```
unsignedIntFib getNumberOfElements(
 char cType='u' )
 const
```

Gibt die Anzahl der Fib-Elemente vom angegebenen Typ im aktuellem Fib-Objekt zurück.

Standardmäßig wird die Anzahl aller Fib-Elemente zurückgegeben. Der Standardtyp zum zurückgeben der Anzahl aller Fib-Elemente ist 'u' für "unknown", damit wird angezeigt, dass man sich auf keinen Typ festlegt, bzw. allw Fib-Elemente gezählt werden.

Eingabeparameter:

- `cType`: Der Typ der Elemente, deren Anzahl zurückgeliefert werden soll. Der Standardwert ist 'u', damit alle Fib-Elemente gezählt werden.

Rückgabe: Die Anzahl vom Fib-Elemente vom angegebenen Typ.

33.1.23 getNumberOfMovePoints

Syntax: `unsignedIntFib getNumberOfMovePoints() const`

Liefert die Anzahl der Verschiebungspunkte im aktuellem Objekt. Verschiebungspunkte, sind Stellen (Positionen von Fib-Elementen) von denen und zu denen Fib-Elemente verschoben werden können. Dies sind in der Regel alle (geraden) Astelemente `cFibLimb` (siehe Abschnitt 38 auf Seite 216). Beispielsweise sind alle Bereichsobjekte Verschiebungspunkte, Punktobjekte hingegen nicht.

Zm Verschieben von Fib-Elementen siehe auch Abschnitt 33.1.37 auf Seite 206
.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Verschiebungspunkte im aktuellem Objekt.

33.1.24 getNumberOfObjectPoints

Syntax: `unsignedIntFib getNumberOfObjectPoints() const`

Diese Methode gibt die Anzahl der zusammenhängenden Teilobjekte im Fib-Objekt zurück.

Zur Beschreibung der Ordnung von zusammenhängenden Teilobjekte siehe Abschnitt 14.8 auf Seite 82 .

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Anzahl der zusammenhängenden Teilobjekte im Fib-Objekt.

33.1.25 typeElementPointToElementPoint

Syntax: `unsignedIntFib typeElementPointToElementPoint( const char cType, const unsignedIntFib elementPoint, bool bAbsolute=false ) const`

Für das `elementPoint`'te Element vom angegebenen Typ (`cType`) im Fib-Objekt, wird die Nummer zurückgegeben, welche es unter allen Fib-Elementen im Fib-Objekt hat.

Damit kann beispielsweise ermittelt werden, das wievielte Fib-Element das dritte Listenelement im Fib-Objekt ist.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, bezüglich welchem die angegebene Nummer `elementPoint` gilt.
- `elementPoint`: Die Nummer des Fib-Elements unter den Fib-Elemente vom angegeben Typ `cType`.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, beziehen sich die Ordnungen auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, beziehen sich Ordnungen auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Die Nummer des `elementPoint`'ten Fib-Elements vom angegeben Typ `cType` in der allgemeinen Fib-Elementenfolge oder 0 , wenn der Fib-Elementpunkt nicht konvertiert werden kann (da er z. B. nicht existiert).

33.1.26 elementPointToObjectPoints

Syntax: `list<unsignedIntFib> elementPointToObjectPoints( const char cType='u', const unsignedIntFib elementPoint, bool bAbsolute=false ) const`

Diese Methode gibt die Liste mit den Nummern aller zusammenhängenden Teilobjekte zurück, in denen das `elementPoint`'te Element vom angegebenen Typ (`cType`) im Fib-Objekt enthalten ist.

Damit kann beispielsweise ermittelt werden, in welchen zusammenhängenden Teilobjekten sich das dritte Funktionselement im Fib-Objekt befindet.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, bezüglich welchem die angegebene Nummer `elementPoint` gilt. Standardmäßig wird angenommen, dass es ein unspezifischer Elementtyp ist. (alle Elemente werden gezählt)

- `elementPoint`: Die Nummer des Fib-Elements unter den Fib-Elemente vom angegeben Typ `cType`.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, beziehen sich die Ordnungen auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, beziehen sich Ordnungen auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Die Liste mit den Nummern aller zusammenhängenden Teilobjekte, in denen das `elementPoint`'te Element vom angegebenen Typ (`cType`) im Fib-Objekt enthalten ist.

33.1.27 objectPointToElementPoint

Syntax: `unsignedIntFib objectPointToElementPoint(`
`const unsignedIntFib objectPoint,`
`bool bAbsolute=false ) const`

Für den zusammenhängenden Teilobjektpunkt mit der Nummer `objectPoint`, wird der Elementpunkt des Elements zurückgegeben, welches als ein zusammenhängenden Teilobjekt in einem Verzweigungselement `cFibBranch` (siehe Abschnitt 39 auf Seite 217) steht, dessen andere Teilobjekte nicht zum Teilobjekt des zusammenhängenden Teilobjektpunktes gehört.

Zu jedem zusammenhängenden Teilobjekt gibt es ein Verzweigungselement über das es Definiert wird. Das definierende Verzweigungselement enthält in einem Zweig (=Unterobjekt) ein Fib-Objekt, welches vollständig zum zusammenhängenden Teilobjekt gehört, und dessen anderen Zweig-Fib-Objekte (=Unterobjekte) überhaupt nicht zum zusammenhängenden Teilobjekt gehört.

Über diese Methode kann das erste Fib-Element ermittelt werden, welches nur noch Auswirkungen auf das Objekt zum zusammenhängenden Teilobjektpunkt und dessen Teilobjekte hat, aber auf keine anderen zusammenhängenden Teilobjekte.

Eingabeparameter:

- `objectPoint`: Die Nummer des zusammenhängenden Teilobjekts unter den zusammenhängenden Teilobjekten.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, beziehen sich die Ordnungen auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, beziehen sich Ordnungen auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Zurückgegeben wird die Nummer des Fib-Elements, welches das erste Fib-Element im Unterobjekt eines Listenelements ist. Wobei das Unterobjekt das zusammenhängenden Teilobjekt mit der zusammenhängenden Teilobjektnummer `objectPoint` definiert. Oder 0 wenn ein zusammenhängenden Teilobjekt mit der Nummer `objectPoint` im Fib-Objekt nicht existiert.

33.1.28 insertElement

Syntax: `bool insertElement( cFibElement *fibElement, const char cType='u', const unsignedIntFib elementPoint=0, bool bAbsolute=false )`

Diese Methode fügt das übergebene Fib-Element an der Stelle des Fib-Elements vom angegeben Typ `cType` ein, welches das `elementPoint`'te Fib-Element vom Typ `cType` ist. Das Fib-Element, welches vorher an der Position stand, wird in das übergebene Fib-Element eingefügt. Das einzufügende Fib-Element darf weder ein Fib-Element enthalten noch in einem enthalten sein.

Das Einfügen des Fib-Elements schlägt fehl, wenn dadurch ein ungültiges Fib-Objekt entstehen würde. Insbesondere müssen alle Variablen die im einzufügenden Fib-Element verwendet werden, an der Position, wo es eingefügt wird, definiert sein.

Alle Fib-Baum Blattelemente, wie Punktelemente, externe Objekte oder externe Unterobjekt, können nur mit dieser Methode in ein Fib-Element eingefügt werden, wenn `elementPoint=0` und das im aktuellen Fib-Element enthaltende Unterobjekt `NULL` ist bzw. nicht existiert, da sie keine Unterobjekte enthalten und damit auch keine aufnehmen können. Sonst müssen sie mit der Methode `insertObjectInElement` eingefügt werden, da sie, wenn sie als Fib-Element korrekt sind, schon ein korrektes Fib-Objekt sind.

Listenelement können niemals mit der Methode eingefügt werden, da sie zwei Unterobjekt benötigen, aber nur eins zur Verfügung steht. Listenelement sind mit der Methode `insertObjectInElement` zu erzeugen.

Eingabeparameter:

- `fibElement`: Das einzufügende Fib-Element.
- `cType`: Der Typ des Fib-Elements, an wessen Stelle das übergebene Fib-Element eingefügt werden soll. Standardmäßig werden Fib-Elemente aller Typen betrachtet/ gezählt.
- `elementPoint`: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom `cType` haben soll. Standardmäßig wird diese mit 0 belegt und damit das `fibElement` im aktuellem Fib-Element eingefügt (bei Verzweigungselement `cFibBranch` (siehe Abschnitt 39 auf Seite 217) im ersten Teilobjekt).
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, bezieht sich Ordnung auf das Fib-Element, von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Wenn das Fib-Element eingefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.29 insertObjectInElement

Syntax: `bool insertObjectInElement(`
```
cFibElement * fibObject,
const char cType='u', unsignedIntFib elementPoint=0,
bool first=true, bool bAbsolute=false )
```

Diese Methode fügt das übergebene `fibObject` unter dem Fib-Element an der angegeben Stelle ein. Das Fib-Element unter dem das Fib-Objekt `fibObject` eingefügt wird, ist das `elementPoint`'te Fib-Element vom angegebenen Typ `cType`. In diesem Fib-Element wird ein neues Listenelement eingefügt. Wenn `first=true` (wahr), wird das übergebene Fib-Objekt `fibObject` als erstes Teilobjekt im Listenelement eingesetzt, sonst als zweite. Das andere Teilobjekt im neuem Listenelement ist das Objekt, welches durch das Listenelement ersetzt wurde bzw. das Objekt was ursprünglich im Fib-Element zu Position stand. Auf diese Weise enthält des Fib-Element nach der Operation sowohl das ursprünglich enthaltende Fib-Objekt wie auch das neue `fibObject`.

Ist das Fib-Element an der angegeben Stelle schon Unterobjekt eines Listenelements, wird das `fibObject` einfach in dieses Listelement eingefügt. Wobei es bei `first=true` an der Stelle des Fib-Elements eingefügt wird und sonst (`first=false`) im Listelement direkt hinter der Stelle des Fib-Elements.

Steht an der angegeben Stelle (`elementPoint=0`) bisher kein Unterobjekt, wird das `fibObject` als Unterobjekt eingefühgt, ohne ein weiteres Listenelement.

Zurückgegeben wird, ob die Operation erfolgreich war. Diese Operation ist beispielsweise niemals auf Punktelementen, als Fib-Element in denen das Fib-Objekt eingefügt werden soll, erfolgreich, da diese keine Fib-Objekte enthalten können.

Das übergebene `fibObject` wird für das Einfügen nicht kopiert und darf daher nicht einfach gelöscht werden (z. B. mit `delete fibObject`).

Im Bild 15 ist ein Beispielaufruf dargestellt.

Eingabeparameter:

- `fibObject`: Das Fib-Objekt, welches neu unter dem Fib-Element an der angegebenen Position eingefügt werden soll.
- `cType`: Der Typ des Fib-Elements unter dem das Fib-Objekt `fibObject` eingefügt werden soll. Standardmäßig werden Fib-Elemente aller Typen betrachtet/gezählt.
- `elementPoint`: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom `cType` haben soll. Standardmäßig wird diese mit 0 belegt und damit das `fibObject` im aktuellem Fib-Element eingefügt.

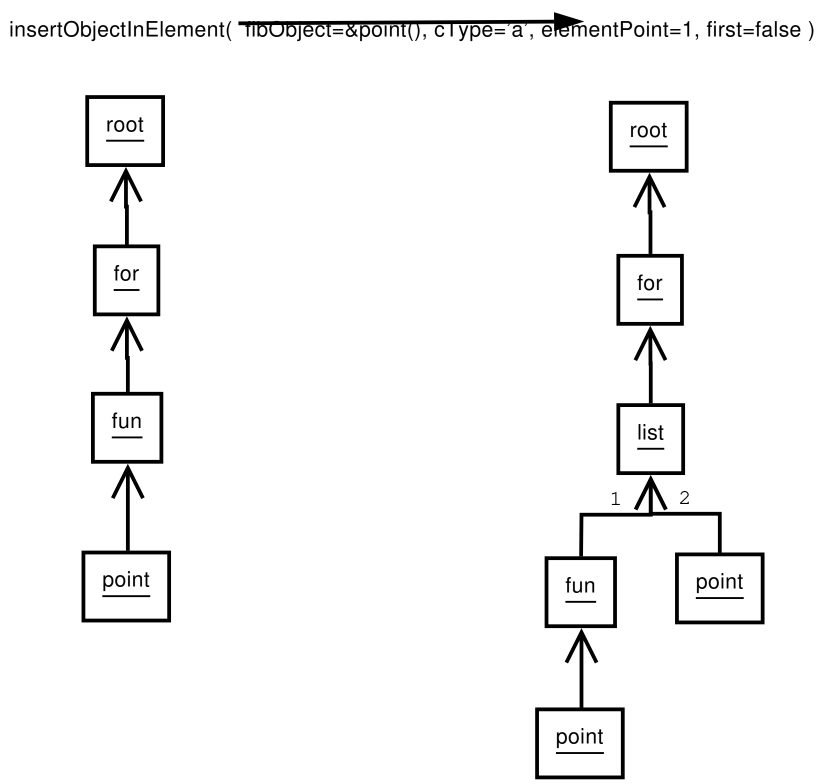


Abbildung 15: Beispiel für insertObjectInElement()

- `first`: Wenn `first` gleich `true` (=wahr) ist, wird das `fibObject` als erstes Unterobjekt im (neuem) Listenelement eingesetzt, und damit die anderen Unterobjekte des Listenelements eventuell überdeckt. Sonst, wenn `first` gleich `false` (=falsch) ist, wird das `fibObject` als letztes Unterobjekt im Listenelement eingesetzt, und damit eventuell selbst überdeckt werden.

- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, bezieht sich Ordnung auf das Fib-Element, von dem aus die Methode aufgerufen wurde. Der Standardwert ist `false`.

Rückgabe: Wenn das Fib-Objekt `fibObject` eingefügt wurde wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.30 overwriteObjectWithObject

Syntax:
```
bool overwriteObjectWithObject(
 cFibElement* fibObject,
 const char cType='u',
 unsignedIntFib elementPoint=0 ,
 bool bDeleteOld=true,
 bool bAbsolute=false )
```

Diese Methode fügt das übergebene `fibObject` an der angegeben Stelle ein. Das Fib-Element, welches vorher an der Stelle stand, wird entfernt und, wenn `bDeleteOld` gleich `true` (=wahr) ist, gelöscht (inklusive aller enthaltender Fib-Elemente). Die Position an der das `fibObject` eingefügt wird, ist die Position an der vorher das Fib-Element stand, welches das `elementPoint`'te Fib-Element vom angegebenen Typ `cType` ist.

Zurückgegeben wird, ob die Operation erfolgreich war. Es kann beispielsweise niemals das aktuelle Fib-Objekt überschrieben werden oder anstatt dessen ein neues Fib-Objekt eingefügt werden.

Eingabeparameter:

- `fibObject`: Das Fib-Objekt, welches anstatt des Fib-Elements an der angegebenen Position eingefügt werden soll.

- `cType`: Typ des Fib-Elements, welches durch das Fib-Objekt `fibObject` ersetzt werden soll. Standardmäßig werden Fib-Elemente aller Typen betrachtet/gezählt.

- `elementPoint`: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom `cType` haben soll. Standardmäßig wird diese mit 0 belegt und damit das `fibObject` im aktuellem Fib-Element eingefügt.

- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird das alte Fib-Objekt (inklusive enthaltender Fib-Elemente) aus dem Speicher gelöscht, sonst verbleibt es im Speicher. Standardwert ist `true` (=wahr), um das alte Fib-Objekt zu löschen.

- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=False) ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Der Standardwert ist `false`.

Rückgabe: Wenn das Fib-Objekt `fibObject` eingefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.31 removeObject

Syntax: `bool removeObject( unsignedIntFib objectPoint,`
`bool bDeleteOld=true,`
`bool bAbsolute=false )`

Diese Methode entfernt das zusammenhängenden Teilobjekt, welches die Nummer `objectPoint` in der Ordnung der zusammenhängenden Teilobjekte hat. Für den `objectPoint` sollte es ein Listenobjekt geben, welches eine Unterobjekt enthält, welches nur Fib-Elemente des zusammenhängenden Teilobjekt enthält, und dessen andere Unterobjekte, nur Fib-Elemente enthalten die nicht zum zusammenhängenden Teilobjekt gehören. Das Unterobjekt, welches nur Fib-Elemente des zusammenhängenden Teilobjekt enthält, wird aus dem Listenelement entfernt. Würde das Listenelement nach der Operation nur noch ein Unterobjekt enthalten, wird das entsprechende Listenelement durch das übrigbleibende Unterobjekt ersetzt und gelöscht.

Wenn das echten Objekt mit der Objektpunktnummer `objectPoint` entfernt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch). Sollte beispielsweise kein zusammenhängenden Teilobjekt mit der Nummer `objectPoint` existieren oder dieses ein Haup-Fib-Objekt in einem root-Element sein, wird `false` zurückgegeben.

Zur Beschreibung der Ordnung von zusammenhängenden Teilobjekte siehe Abschnitt 14.8 auf Seite 82 .

Eingabeparameter:

- `objectPoint`: Die Nummer des zu löschenden zusammenhängenden Teilobjekts, welche es unter den zusammenhängenden Teilobjekten hat.

- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird das entfernte Fib-Objekt (inklusive enthaltender Fib-Elemente und einem eventuell entfernten Listenelement) aus dem Speicher gelöscht, sonst verbleibt es im Speicher. Standardwert ist `true` (=wahr), um das alte Fib-Objekt zu löschen.

- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Wenn das zusammenhängenden Teilobjekt mit der zusammenhängenden Teilobjektpunktnummer `objectPoint` gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.32 hasUnderAllObjects

Syntax: `bool hasUnderAllObjects() const`

Diese Methode prüft, ob in den Fib-Elementen des aktuelle Fib-Objekts alle Unterobjekte vorhanden sind. Sind im Fib-Objekt nicht alle Unterobjekte vorhanden, ist es kein korrektes Fib-Objekt.

Eingabeparameter: keine

Rückgabe: Wenn im Fib-Objekt alle Unterobjekte vorhanden sind, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.33 isRemovableElement

Syntax:
```
bool isRemovableElement( const char cType='u',
 const unsignedIntFib elementPoint=0,
 bool bAbsolute=false ) const
```

Diese Methode überprüft, ob das Fib-Element vom Typ `cType`, welches die Nummer `elementPoint` in der Ordnung der Fib-Element vom angegebenen Typ `cType` hat, löschbar ist.

Blattelemente von der Klasse `cFibLeaf` (z. B. Punkte) und korrekte Verzweigungselemente von der Klasse `cFibBranch` (z. B. Listenelemente) sind niemals löschbar. Alle (geraden) Astelemente `cFibLimb` (siehe Abschnitt 38 auf Seite 216 z. B. Funktions- und Bereichselemente) sind löschbar, wenn die Variable, die sie definieren, nirgendwo sonst benutzt wird.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, für welches geprüft werden soll, ob es löschbar ist. Standardmäßig werden Fib-Elemente aller Typen betrachtet/gezählt.

- `elementPoint`: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom `cType` haben soll. Standardmäßig wird diese mit 0 belegt und damit das `fibObject` im aktuellem Fib-Element gelöscht.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false` (=falsch).

Rückgabe: Wenn das angegebene Fib-Element vom Typ `cType`, welches die Nummer `elementPoint` in der Ordnung der Fib-Element vom angegebenen Typ `cType` hat, gelöscht werden kann, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.34 removeElement

Syntax: `bool removeElement( const char cType='u',`
`const unsignedIntFib elementPoint=0,`
`bool bAbsolute=false )`

Diese Methode löscht das Fib-Element vom Typ `cType`, welches die Nummer `elementPoint` in der Ordnung der Fib-Element vom angegebenen Typ `cType` hat. An der Stelle des Fib-Elements steht nach der Operation das Fib-Element, welches das gelöschte Fib-Element als Unterobjekt enthalten hat. Es wird also das Fib-Element, das gelöscht werden soll, durch das Fib-Element, welches es enthält, ersetzt und erst dann wird das zu löschende Fib-Element gelöscht. Das Fib-Element kann nicht gelöscht werden, wenn dadurch ein ungültiges Fib-Objekt entstehen würde.

Wenn das Fib-Element vom Typ `cType`, mit der Nummer `elementPoint` in der Ordnung der Fib-Element vom angegebenen Typ `cType` , gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch). Sollte beispielsweise versucht werden ein Listenelement zu löschen, schlägt die Operation fehl und es wird `false` zurückgegeben, da Listenelemente mindestens zwei Unterobjekte haben, aber nur eins das Listenelement ersetzen könnte. Des weiteren können auch keine Fib-Elemente gelöscht werden, welche Variablen definieren, die noch benötigt werden. Das aktuelle Fib-Element kann natürlich auch nicht gelöscht werden.

Durch die Methode `isRemovableElement` (siehe Abschnitt 33.1.33 auf Seite 203) kann geprüft werden, ob ein Fib-Element löschbar ist.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, welches gelöscht werden soll. Standardmäßig (für den Standardwert `'u'`) werden Fib-Elemente aller Typen betrachtet/gezählt.

- elementPoint: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom cType haben soll. Standardmäßig wird diese mit 0 belegt, damit das erste Fib-Element des ersten Unterobjekts im aktuellem Fib-Element gelöscht wird.
- bAbsolute: Wenn bAbsolute gleich true (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn bAbsolute gleich false ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist false (=falsch).

Rückgabe: Wenn das entsprechende Fib-Element vom Typ cType, welches die Nummer elementPoint in der Ordnung der Fib-Element vom angegebenen Typ cType hat, gelöscht wurde, wird true (=wahr) zurückgegeben, sonst false (=falsch).

33.1.35 cutElement

Syntax:
```
cFibElement * cutElement( const char cType='u',
 const unsignedIntFib elementPoint=0,
 bool bAbsolute=false )
```

Diese Methode entfernt das Fib-Element vom Typ cType, welches die Nummer elementPoint in der Ordnung der Fib-Element vom angegebenen Typ cType hat, aus dem Fib-Objekt und gibt es zurück. An der Stelle des Fib-Elements steht nach der Operation, das Fib-Element (Unterobjekt), welches das entfernt Fib-Element enthalten hat. Es wird also das Fib-Element, das entfernt werden soll, durch das Fib-Element, welches es enthält, ersetzt und dann das entfernte Fib-Element zurückgegeben. Das Fib-Element kann nicht entfernt werden, wenn dadurch ein ungültiges Fib-Objekt entstehen würde.

Wenn das Fib-Element vom Typ cType, mit der Nummer elementPoint in der Ordnung der Fib-Element vom angegebenen Typ cType , entfernt wurde, wird es bzw. ein Zeiger auf es zurückgegeben, sonst wird NULL zurückgegeben. Sollte beispielsweise versucht werden ein Listenelement zu entfernen, schlägt die Operation fehl und es wird NULL zurückgegeben, da Listenelemente mindestens zwei Unterobjekte haben, aber nur eins das Listenelement ersetzen könnte. Des weiteren können auch keine Fib-Elemente entfernt werden, welche Variablen definieren, die noch benötigt werden.

Eingabeparameter:

- cType: Der Typ des Fib-Elements, welches entfernt werden soll. Standardmäßig ('u') werden Fib-Elemente aller Typen betrachtet/ gezählt.
- elementPoint: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom cType haben soll. Standardmäßig wird diese mit 0 belegt und damit das Fib-Element im aktuellem Fib-Element ausgeschnitten.

- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Wenn das Fib-Element vom angegebenen Typ `cType`, welches die Nummer `elementPoint` in der Ordnung der Fib-Element vom angegebenen Typ `cType` hat, entfernt wurde, wird es bzw. ein Zeiger auf es zurückgegeben, sonst wird `NULL` zurückgegeben.

33.1.36 deleteObject

Syntax: `static void deleteObject( cFibElement * fibObject )`

Diese Methode löscht das gesamte übergebene Fib-Objekt `fibObject` mit allen seinen enthaltenden Elementen. Auf diese Weise muss nicht jedes Fib-Element im Fib-Objekt einzeln über seinen Destruktor gelöscht werden.

Achtung: Existiert das `fibObject` noch in anderen Fib-Elementen bzw. Fib-Objekten werden auch diese mitgelöscht.

Eingabeparameter:

- `fibObject`: Einen Zeiger auf das zu löschende Fib-Objekt.

Rückgabe: keine

33.1.37 moveLimbElement

Syntax:
```
intFib moveLimbElement( const char cType='u',
 const unsignedIntFib elementPoint=0,
 const intFib iHowfar=1,
 bool bAbsolute=false )
```

Diese Methode verschiebt das Fib-Element vom angegeben Typ `cType`, welches das `elementPoint`'te Fib-Element vom Typ `cType` ist, über `iHowfar` Fib-Element. Wenn `iHowfar` positiv ist, wird das Fib-Element nach unten verschoben, sonst nach oben. Es können nur `cFibLimb` verschoben werden, also nur Fib-Element die genau ein Unterobject haben.

Das Verschieben des Fib-Elements wird abgebrochen, wenn durch ein weiteres Verschieben ein ungültiges Fib-Objekt entstehen würde. Die Anzahl der Fib-Elemente, über die das zu verschiebende Fib-Element verschoben wurde, wird zurückgegeben. Wobei der zurückgegebene Wert negativ ist, wenn nach oben verschoben wurde. (für die Definition von Oben und Unten siehe Abschnitt 14.3 auf Seite 74)

Ein Fib-Element kann beispielsweise nicht verschoben werden, wenn es ein Punktelement ist oder wenn es über ein Fib-Element verschoben werden soll, welches eine Variable definiert, die das zu verschiebende Fib-Element benötigt.

Eine besondere Betrachtung beim Verschieben benötigt die Verzweigungselemente (cFibBranch). Wird ein Fib-Element über ein Verzeweigungselemente nach unter verschoben, wandert es in alle Unterobjekte des Verzweigungselemente, in denen es bzw. eine Variable die es definiert benötigt wird, wobei in allen außer einem Unterobjekt natürlich Kopien des Fib-Element verwendet werden. Wird die vom zu verschiebenden Fib-Element definierte Variable in keinem Unterobjekt benötigt, wird es nur in das erste Unterobjekt verschoben. Das zu verschiebende Fib-Element wird dann in allen Unterobjekt die noch verbleibenden Schritte $iHowfar - 1$ weiter nach unten verschoben. Zurückgegeben wird die Summe der Fib-Elemente, über die das zu verschiebende Fib-Element oder eine Kopie dessen verschoben wurde.

Auch eine Verschiebung über das und vom aktuellem Fib-Element ist möglich.

Eingabeparameter:

- cType: Der Typ des Fib-Elements, welches verschoben werden soll. Standardmäßig werden Fib-Elemente aller Typen betrachtet/ gezählt.
- elementPoint: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom cType haben soll. Standardmäßig wird diese mit 0 belegt und damit das aktuelle Fib-Element verschoben.
- iHowfar: Die Anzahl der Fib-Elemente über die das zu verschiebende Fib-Element verschoben werden soll. Wenn iHowfar positiv ist, wird das Fib-Element nach unten verschoben, sonst nach oben. Standardmäßig wird das zu verschiebene Fib-Element um ein Fib-Element nach unter verschoben.
- bAbsolute: Wenn bAbsolute gleich true (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn bAbsolute gleich false (=falsch) ist, bezieht sich Ordnung auf das Fib-Element von dem aus die Methode aufgerufen wurde. Standardwert ist false.

Rückgabe: Die Anzahl der Fib-Elemente, über die das zu verschiebene Fib-Element verschoben wurde. Wobei der zurückgegebene Wert negativ ist, wenn nach oben verschoben wurde, und sonst positiv ist.

33.1.38 clone

Syntax: `cFibElement *clone() const`

Diese Methode stellt eine Kopie des gesamten Fib-Objekts her. Dabei werden alle Elemente (auch Vektoren und Variablen) von der Wurzel bzw. dem höchsten root-Element an dupliziert.

Eingabeparameter: keine

Rückgabe: Ein Zeiger auf die Kopie des gesamten Fib-Objekts.

33.1.39 copy

Syntax: `cFibElement *copy( const unsignedIntFib iObjectPoint=0 ) const`

Diese Methode stellt eine Kopie zusammenhängenden Teilobjekts her, welches das `iObjectPoint`'te unter den zusammenhängenden Teilobjekten ist. Dabei werden alle Elemente (auch Vektoren und Variablen) von aktuellen Fib-Element an dupliziert, welche zum zusammenhängenden Teilobjekt gehören.

Sollte es kein zusammenhängenden Teilobjekt mit der Nummer `iObjectPoint` geben, wird der Nullpointer `NULL` zurückgegeben.

Variablen die das Fib-Objekt verwendet aber nicht in ihm definiert werden, behalten ihre Referenz auf ihre alte Definition.

Eingabeparameter:

- `iObjectPoint`: Die Nummer, die das zu kopierende zusammenhängenden Teilobjekt unter den zusammenhängenden Teilobjekten hat. Standardmäßig wir dieser Parameter mit 0 belegt und damit das gesamte aktuelle Fib-Objekt kopiert.

Rückgabe: Ein Zeiger auf eine Kopie des zusammenhängenden Teilobjekts mit der Nummer `iObjectPoint` in der Ordnung der zusammenhängenden Teilobjekte, oder `NULL`, wenn ein solches zusammenhängenden Teilobjekt nicht existiert.

33.1.40 copyElement

Syntax: `cFibElement *copyElement( const char cType='u', const unsignedIntFib elementPoint=0, bool bAbsolute=false ) const`

Diese Methode stellt eine Kopie des Fib-Elements vom angegeben Typ `cType` her, welches das `elementPoint`'te Fib-Element vom Typ `cType` ist. Dabei wird das Fib-Element mit seinen enthaltenden Werten und Vektoren aber nicht enthaltende Fib-Elemente kopiert. Verwendete Variablen behalten ihre Referenz bei.

Das zurückgegebene Fib-Element hat nach der Kopie den Nullpointer `NULL` für seine (eventuell vorhandenen) Unterobjekte.

Sollte kein Fib-Element an der angegebenen Position existieren, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter:

- `cType`: Der Typ des Fib-Elements, welches kopiert werden soll. Standardmäßig werden Fib-Elemente aller Typen betrachtet/ gezählt.
- `elementPoint`: Die Nummer des Fib-Elements, die es unter den Fib-Elementen vom `cType` haben soll. Standardmäßig wird diese mit 0 belegt und damit das aktuellem Fib-Element kopiert.
- `bAbsolute`: Wenn `bAbsolute` gleich `true` (=wahr) ist, bezieht sich die Ordnung auf das gesamte Fib-Objekt. Ansonsten, wenn `bAbsolute` gleich `false` (=falsch) ist, bezieht sich Ordnung auf das Fib-Element, von dem aus die Methode aufgerufen wurde. Standardwert ist `false`.

Rückgabe: Ein Zeiger auf eine Kopie des Fib-Elements an der angegebenen Position, oder `NULL`, wenn ein solches Fib-Element nicht existiert.

33.1.41 equal

Syntax: `bool equal( const cFibElement* fibObject ) const`

Diese Methode prüft, ob das übergebene Fib-Objekt gleich zum aktuellen Fib-Objekt ist. Dabei müssen alle Elemente (Fib-Elemente, Vektoren, Werte) und die Struktur gleich sein, bis auf die Referenzen für die Variablen. Referenzen für Variablen müssen auf zum übergebenen Fib-Objekt entsprechende Definitionen in Fib-Elementen im eigenen Fib-Objekt verweisen.
Die folgenden Fib-Objekte sind demanch gleich:

- $for(x_1, p((1, x_1)))$
- $for(x_3, p((1, x_3)))$

Demgegenüber sind die Fib-Objekte nicht gleich:

- $for(x_1, p((1, x_1)))$
- $fun(x_3, p((1, x_3)))$

Es wird nur das Fib-Objekt unterhalb dem aktuellen und übergebenen Fib-Element geprüft und Fib-Elemente oberhalb der Fib-Elemente, welche Variablen definieren, die in dem Fib-Objekt unterhalb der Fib-Elemente direkt oder indirekt benötigt werden.

Eingabeparameter:

- `fibObject`: Das Fib-Objekt, zu dem das aktuelle Fib-Objekt gleich sein soll.

Rückgabe: Wenn das aktuelle Fib-Objekt zum übergebenen Fib-Objekt gleich ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.42 storeXml

Syntax: `bool storeXml( ostream& outstream ) const`

Diese Methode speichert das aktuelle Fib-Objekt im XML-Format in den übergebenen `outstream`.

Für den Aufbau der XML-Daten siehe Abschnitt 26 auf Seite 141.

Eingabeparameter:

- `outstream`: Der Datenstrom, in dem das Fib-Objekt gespeichert werden soll.

Rückgabe: Wenn das aktuelle Fib-Objekt erfolgreich gespeichert wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.43 restoreXml

Syntax: `static cFibElement* restoreXml(`
`istream& instream , intFib *outStatus=NULL )`

Diese Methode lädt ein Fib-Objekt im XML-Format aus dem übergebenen `instream` und gibt eine Referenz darauf zurück. Konnte das Fib-Objekt nicht erfolgreich geladen werden, wird der Nullpointer `NULL` zurückgegeben.

Für den Aufbau der XML-Daten siehe Abschnitt 26 auf Seite 141.

Eingabeparameter:

- `instream`: Der Datenstrom, aus dem das Fib-Objekt geladen werden soll.
- `outStatus`: Ein Integerfeld zum Speichern des Ladestatus. Standardmäßig wird das Feld auf `NULL` gesetzt und damit kein Ladestatus zurückgegeben.

Rückgabe: Wenn das Fib-Objekt erfolgreich geladen wurde, wird ein Zeiger darauf zurückgegeben, sonst der Nullpointer `NULL`. Wenn `outStatus` beim Aufruf nicht `NULL` war, steht nach der Ladeoperation der Ladestatus in dieser Variable. Negative Werte sind Fehler und positive Warnungen.

Mögliche Werte für `outStatus`:

0 Laden erfolgreich

-1 Laden fehlgeschlagen, der Stream `instream` ist ungültig

-2 Laden fehlgeschlagen, Daten sind Fehlerhaft

1 Daten im Stream `instream` sind Fehlerhaft, Fehler konnten aber korrigiert werden

2 Daten im Stream `instream` sind Fehlerhaft, Fib-Objekt konnte aber geladen werden; Fehler konnten nicht korrigiert werden, das Multimediaobjekt ist eventuell fehlerhaft

33.1.44 store

Syntax: `bool store( ostream& outstream ) const`

Diese Methode speichert das aktuelle Fib-Objekt im komprimierter Form in den übergebenen `outstream`.

Für den Aufbau der komprimierten Daten siehe Abschnitt 25 auf Seite 105.

Eingabeparameter:

- `outstream`: Der Datenstrom, in dem das Fib-Objekt gespeichert werden soll.

Rückgabe: Wenn das aktuelle Fib-Objekt erfolgreich gespeichert wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

33.1.45 restore

Syntax: `static cFibElement* restore( istream& instream , intFib *outStatus=NULL )`

Diese Methode läd ein Fib-Objekt in komprimierter Form aus dem übergebenen `instream` und gibt eine Referenz darauf zurück. Konnte das Fib-Objekt nicht erfolgreich geladen werden, wird der Nullpointer `NULL` zurückgegeben.

Für den Aufbau der komprimierten Daten siehe Abschnitt 25 auf Seite 105.

Eingabeparameter:

- `instream`: Der Datenstrom, aus dem das Fib-Objekt geladen werden soll.
- `outStatus`: Ein Integerfeld zum Speichern des Ladestatus. Standardmäßig wird das Feld auf `NULL` gesetzt und damit kein Ladestatus zurückgegeben.

Rückgabe: Wenn das Fib-Objekt erfolgreich geladen wurde, wird ein Zeiger darauf zurückgegeben, sonst der Nullpointer `NULL`. Wenn `outStatus` beim Aufruf nicht `NULL` war, steht nach der Ladeoperation der Ladestatus in dieser Variable. Negative Werte sind Fehler und positive Warnungen.

Mögliche Werte für `outStatus`:

0 Laden erfolgreich

-1 Laden fehlgeschlagen, der Stream `instream` ist ungültig

-2 Laden fehlgeschlagen, Daten sind Fehlerhaft

1 Daten im Stream `instream` sind Fehlerhaft, Fehler konnten aber korrigiert werden

2 Daten im Stream `instream` sind Fehlerhaft, Fib-Objekt konnte aber geladen werden; Fehler konnten nicht korrigiert werden, das Multimediaobjekt ist eventuell fehlerhaft

33.1.46 getAllRootObjectIdentifiers

Syntax: `list<longFib> getAllRootObjectIdentifiers()`

Diese Methode gibt eine Liste mit den Identifiern aller root-Elementen im gesamten Fib-Objekt und der Datenbank zurück.

Dabei können einige Identifier doppelt auftauchen, wenn sie doppelt im gesamten Fib-Objekt vorhanden sind.

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Identifiern aller root-Objekte im gesamten Fib-Objekt und der Datenbank.

33.1.47 getAllDatabaseObjectIdentifiers

Syntax: `static list<longFib> getAllDatabaseObjectIdentifiers()`

Diese Methode gibt eine Liste mit den Identifiern aller Fib-Objekte in der Fib-Datenbank zurück.

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Identifiern aller Fib-Objekte in der Fib-Datenbank.

33.1.48 getRootObject

Syntax: `cRoot * getRootObject( longFib lIdentifier )`

Diese Methode gibt das root-Objekt zurück, welches den angegebenen Identifier `lIdentifier` besitzt. Dabei werden zuerst die vom aktuellem Fib-Element zugreifbaren root-Objekte im aktuellem Fib-Objekt, dann das gesamte Fib-Objekt und dann die Datenbank durchsucht. Wird bei der Suche ein passendes root-Objekt gefunden, wird die Suche abgebrochen und das root-Objekt zurückgegeben. Wird kein passendes root-Objekt gefunden, wird `NULL` zurückgegeben.

Eingabeparameter:

- `lIdentifier`: Der Identifier, der zum zurückzugebenden root-Objekt gehören soll.

Rückgabe: Das root-Objekt das zum `lIdentifier` gehört oder `NULL`, wenn kein solches root-Objekt existiert.

33.1.49 getAllAccessibleRootObjectIdentifiers

Syntax: `list<longFib> getAllAccessibleRootObjectIdentifiers()`

Diese Methode gibt eine Liste mit den Identifiern aller root-Elementen zurück, welche vom aktuellem Fib-Element zugreifbar sind. Welche root-Elemente sichbar sind, wird in Abschnitt 12.14.4 auf Seite 70 beschrieben. Solche root-Elemente können dann über externe Objekte in den aktuellen Fib-Element eingefügt werden, siehe Abschnitt 12.9 auf Seite 43 .

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Identifiern aller root-Objekte, welche vom aktuellem Fib-Element zugegriffen werden können.

33.1.50 getAccessibleRootObject

Syntax: `cRoot * getAccessibleRootObject( longFib lIdentifier )`

Diese Methode gibt das root-Objekt zurück, welches den angegebenen Identifier `lIdentifier` besitzt. Dabei werden nur die vom aktuellen Fib-Element zugreifbaren root-Objekte durchsucht. Welche root-Elemente sichbar sind und in welcher Reihenfolge sie durchsucht werden, wird in Abschnitt 12.14.4 auf Seite 70 beschrieben. Wird bei der Suche ein passendes root-Objekt gefunden, wird die Suche abgebrochen und das root-Objekt zurückgegeben. Wird kein passendes root-Objekt gefunden, wird `NULL` zurückgegeben.

Eingabeparameter:

- `lIdentifier`: Der Identifier, der zum zurückzugebenden root-Objekt gehören soll.

Rückgabe: Das root-Objekt das zum `lIdentifier` gehört oder `NULL`, wenn kein solches root-Objekt vom aktuellen Fib-Element zugreifbar ist.

33.1.51 getValidDomains

Syntax: `cDomains getValidDomains() const`

Diese Methode gibt eine Referenz auf die Definitionsbereiche, die für das Fib-Element gelten, zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf die Definitionsbereiche, die für das Fib-Element gelten.

33.1.52 getNumberOfDimensions

Syntax: `unsignedIntFib getNumberOfDimensions() const`

Diese Methode gibt die Anzahl der Dimensionen des Fib-Objekts zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Dimensionen des Fib-Objekts.

33.1.53 getDimensionMapping

Syntax: `unsignedIntFib getDimensionMapping( unsignedIntFib iDimensionNumber ) const`

Diese Methode gibt den Mappingwert für die Dimension `iDimensionNumber` zurück. Existiert keine solche Dimension, wird 0 (für "none") zurückgegeben.

Die Werte werden in Tabelle 7 auf Seite 64 definiert.

Eingabeparameter:

- `iDimensionNumber`: Nummer der Dimension, für welche das Mapping zurückgegeben werden soll.

Rückgabe: Den Mappingwert für die Dimension `iDimensionNumber`, existiert keine solche Dimension wird 0 zurückgegeben.

34 Das Interface iEvaluePosition

Das Interface `iEvaluePosition` dient zum Auswerten von Fib-Objekten. Es stellt die Methode `evaluePosition()` bereit.

Jede Klasse die das Interface `iEvaluePosition` implementiert, hat die Methode `evaluePosition()` zu implementieren. Objekte der abgeleiteten Klasse können dann der Methode `evalueObject()` (siehe Abschnitt 33.1.10 auf Seite 183) übergeben werden, um das Fib-Objekt auszuwerten.

Wenn die Methode `evalueObject()` einen Punkt auswertet, ruft sie die Methode `evaluePosition()` des übergebenen Objekts mit der Position der Punktes und seinen Eigenschaften auf.

34.1 evaluePosition

Syntax: `bool evaluePosition(`
`const cVectorPosition & vPosition,`
`const list<cVectorProperty> & vProperties )`

Diese Methode wird von der Methode `evalueObject()` (siehe Abschnitt 33.1.10 auf Seite 183) aufgerufen, wenn ein Punkt ausgewertet werden soll.

Eingabeparameter:

- `vPosition`: Die Position des Punktes, der ausgewertet werden soll.
- `vProperties`: Eine Liste mit Eigenschaften des Punktes.

Rückgabe: Zurückgegeben wird `true` (=wahr) wenn die Auswertung vortgesetzt werden soll, sonst false.

35 Das Interface iEvalueFibElement

Das Interface `iEvalueFibElement` dient zum Auswerten von Fib-Objekten. Es stellt die Methode `evalueElement()` bereit.

Jede Klasse die das Interface `iEvalueFibElement` implementiert, hat die Methode `evalueElement()` zu implementieren. Objekte der abgeleiteten Klasse können dann der Methode `evalueObject()` (siehe Abschnitt 33.1.11 auf Seite 185) übergeben werden, um das Fib-Objekt auszuwerten.

Wenn die Methode `evalueObject()` ein Fib-Element, vom angegeben Type oder ein Punktelement, auswertet, ruft sie die Methode `evalueElement()` des übergebenen Objekts mit einem Zeiger auf des Fib-Element und seinen Eigenschaften auf.

35.1 evaluePosition

Syntax: `bool evalueElement( cFibElement & pFibElement,`
`const list<cVectorProperty> & vProperties )`

Diese Methode wird von der Methode `evalueObject()` (siehe Abschnitt 33.1.11 auf Seite 185) aufgerufen, wenn ein Fib-Element, vom angegeben Type oder ein Punktelement, ausgewertet werden soll.

Eingabeparameter:

- `pFibElement`: Eine Refernz auf das Fib-Element, welches ausgewertet werden soll.
- `vProperties`: Eine Liste mit Eigenschaften des Fib-Elements.

Rückgabe: Zurückgegeben wird `true` (=wahr) wenn die Auswertung vortgesetzt werden soll, sonst false.

36 Abhängigkeiten

Jedes Fib-Element (auch die root-Elemente) hält jeweils eine Referenzen auf sein Vorgänger-, Nachfolger-, nächst höheres Fib-Element und nächst höheres root-Element. Das zu einem Fib-Element gehörende root-Element ist das nächste root-Element, welches das Fib-Element enthält. die Vorgänger- und Nachfolgerelemente sind die entsprechenden Elemente in der Ordnung der Fib-Elemente (siehe Abschnitt 14.4 auf Seite 75).

Um zu gewährleisten, dass ein Fib-Element immer die Referenz auf das richtige Vorgänger-, Nachfolger-, nächst höheres Fib-Element und root-Element hält, registriert sich ein neues Fib-Element oder -Objekt beim gesamten Fib-Objekten. Die Registrierung wird an alle Fib-Elemente weitergegeben, welche jeweils ihre abhängigen Referenzen und Daten aktualisieren.

37 cFibLeaf

Die Klasse `cFibLeaf` ist die Elternklasse für alle Fib-Elementklassen die Blätter sind, also kein Unterobjekt einthalten. Von `cFibLeaf` selbst können keine Instnzen erzeugt werden.

Elternklasse: `cFibElement`

Die Abgeleiteten Klassen sind:

- `cPoint`
- `cExtSubobject`

38 cFibLimb

Die Klasse `cFibLimb` ist die Elternklasse für alle Fib-Elementklassen die (gerade) Zweige sind, also genau ein Unterobjekt enthalten. Von `cFibLimb` selbst können keine Instnzen erzeugt werden.

Elternklasse: `cFibElement`

Die Abgeleiteten Klassen sind:

- `cProperty`
- `cFibList`
- `cComment`
- `cArea`
- `cFunction`
- `cIf`

39 cFibBranch

Die Klasse `cFibBranch` ist die Elternklasse für alle Fib-Elementklassen die Verzweigungen darstellen, also mehrere Unterobjekte haben können. Von der Klasse `cFibBranch` selbst können keine Instnzen erzeugt werden.

Elternklasse: `cFibElement`

Die Abgeleiteten Klassen sind:

- `cList`
- `cIf`
- `cRoot`
- `cExtObject`

40 cPoint

Die Klasse `cPoint` realisiert das Punktelement. Zur Beschreibung des Punktelements siehe Abschnitt 12.2 auf Seite 24 .

Elternklasse: `cFibLeaf`

40.1 Schnittstellenbeschreibung

40.1.1 cPoint

Syntax: `cPoint( const cVectorPosition *vecPositon=NULL )`

Der Konstruktor des Punktelements, er erstellt einen Punkt. Eine Kopie eines eventuell übergebene Positionsvektor wird als Position des Punktes eingesetzt.

Eingabeparameter:

- `vecPositon`: Einen Zeiger auf den Positionsvektor, den der erzeugte Punkt erhalten soll. Standardmäßig wird `NULL` übergeben. Ist der Positionsvektor `NULL` wird ein Punkt ohne Positionsvektor erzeugt (der Punkt hat keine Auswirkungen) .

Rückgabe: keine

40.1.2 getPosition

Syntax: `cVectorPosition *getPosition()`

Über diese Methode kann der Positionsvektor des Punktes ermittelt werden.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf den Positionsvektor des Punktes oder `NULL`, wenn der Punkt keinen Positionsvektor enthält.

40.1.3 setPosition

Syntax: `void setPosition( const cVectorPosition *vecPositon=NULL )`

Über diese Methode kann der Positionsvektor des Punktes gesetzt werden. Eine Kopie eines eventuell übergebene Positionsvektor wird als Position des Punktes eingesetzt. Wird kein Positionsvektor (`NULL`) übergeben, hat der Punkt keinen Positionsvektor (der Punkt hat keine Auswirkungen) .

Eingabeparameter:

- `vecPositon`: Einen Zeiger auf den Positionsvektor, den der Punkt erhalten soll. Standardmäßig wird `NULL` übergeben. Ist der Positionsvektor `NULL`, entsteht ein Punkt ohne Positionsvektor (der Punkt hat keine Auswirkungen).

Rückgabe: keine

41 cProperty

Die Klasse `cProperty` realisiert das Eigenschaftselement. Zur Beschreibung des Eigenschaftselements siehe Abschnitt 12.3 auf Seite 25 .

Elternklasse: `cFibLimb`

41.1 Schnittstellenbeschreibung

41.1.1 cProperty

Syntax: `cProperty( const cVectorProperty &vecProperty, cFibElement *pFibUnderObject )`

Der Konstrukter des Eigenschaftselements, er erstellt einen Eigenschaftselement. Eine Kopie eines übergebene Eigenschaftsvektors `vecProperty` wird als Eigenschaft des Eigenschaftselements eingesetzt.

Von den übergebenen Unterobjekten `pFibUnderObject` wird keine Kopie erstellt, daher darf es nicht einfach gelöscht werden.

Eingabeparameter:

- `vecProperty`: Der einzusetzende Eigenschaftsvektor, den das erzeugte Eigenschaftselement erhalten soll.
- `pFibUnderObject`: Ein Zeiger auf das einzusetzende Unterobjekt, den das erzeugte Eigenschaftselement enhalten soll. Der Standardwert ist `NULL`. Ist der Wert `NULL`, hat das Eigenschaftselement zunächst kein Unterobjekt, dieses muss dann später gesetzt werden.

Rückgabe: keine

41.1.2 getProperty

Syntax: `cVectorProperty *getProperty()`

Über diese Methode kann der Eigenschaftsvektor der Eigenschaft ermittelt werden.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf den Eigenschaftsvektor der Eigenschaft.

42 cList

Mit der Klasse `cList` wird das Listenelement realisiert. Zur Beschreibung des Listenelements siehe Abschnitt 12.4 auf Seite 31 .

Elternklasse: `cFibBranch`

42.1 Schnittstellenbeschreibung

42.1.1 cList

Syntax: `cList( cFibElement * fibObject1, cFibElement * fibObject2 )`

Der Konstruktor des Listenelements, er erstellt einen Listenobjekt. Die erstellte Liste hat die beiden übergebenen Fib-Objekte als Teilobjekte. Von den übergebenen Unterobjekten `fibObject1` und `fibObject2` wird keine Kopie erstellt, sie dürfen daher nicht einfach gelöscht werden.

Eingabeparameter:

- `fibObject1`: Ein Zeiger auf das erste Fib-Objekt bzw. Unterobjekt des erstellten Listenobjekts.
- `fibObject2`: Ein Zeiger auf das zweite Fib-Objekt bzw. Unterobjek des erstellten Listenobjekts.

Rückgabe: keine

42.1.2 getNumberOfUnderObjects

Syntax: `unsignedIntFib getNumberOfUnderObjects() const`

Diese Methode gibt die Anzahl der Unterobjekt im aktuellen Listenelement zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Unterobjekt im aktuellen Listenelement.

42.1.3 getUnderObject

Syntax: `cFibElement* getUnderObject( unsignedIntFib iNumberOfUnderObject=1 )`

Diese Methode gibt das `iNumberOfUnderObject`'te Unterobjekt des Listenelement zurück.

Eingabeparameter:

- `iNumberOfUnderObject`: Die Nummer des Unterobjekts in der Liste des Listenelements, welches zurückgegeben werden soll. Wird keine Nummer angegeben, wird das erste Unterobjekt aus der Liste zurückgegeben.

Rückgabe: Das `iNumberOfUnderObject`'te Unterobjekt des Listenelement oder `NULL`, wenn kein `iNumberOfUnderObject`'tes Unterobjekt existiert.

42.1.4 addUnderObject

Syntax: `bool addUnderObject( cFibElement * pUnderObject, unsignedIntFib iPosition=1 )`

Diese Methode fügt das übergebene Fib-Objekt `pUnderObject` an der Stelle `iPosition` in der Unterobjektliste ein.

Von den übergebenen Unterobjekt `pUnderObject` wird keine Kopie erstellt, es darf daher nicht einfach gelöscht werden.

Eingabeparameter:

- `pUnderObject`: Ein Zeiger auf das Unterobjekt, welches in die Liste eingefügt werden soll.
- `iPosition`: Die Position, an der das Unterobjekt in die Liste eingefügt werden soll. Wird keine Nummer angegeben, wird das Unterobjekt als Erstes in der Liste eingefügt. Ist die übergebene Zahl größer als die Anzahl der Unterobjekte in der Liste, wird das übergebene Unterobjekt `pUnderObject` als letztes an die Liste angehängt.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn das übergebene Unterobjekt `pUnderObject` in die Liste eingefügt wurde, sonst `false` (=falsch).

42.1.5 deleteUnderObject

Syntax: `bool deleteUnderObject( unsignedIntFib iPositionUnderObject, bool bDeleteOld=true )`

Diese Methode löscht das Unterobjekt an der Stelle `iPositionUnderObject` in der Unterobjektliste.

Dabei können alle bis auf zwei Unterobjekt aus der Liste gelöscht werden. Da ein Listenobjekt mit weniger als zwei Unterobjekten kein gültiges Fib-Objekt mehr darstellt. Soll dennoch nur noch ein Unterobjekt verwendet werden, ist das Listenelement (das Fib-Element) zu löschen.

Eingabeparameter:

- `iPositionUnderObject`: Die Position, von der das Unterobjekt aus der Liste gelöscht werden soll. Ist die übergebene Zahl größer als die Anzahl der Unterobjekte in der Liste oder 0, wird kein Unterobjekt gelöscht.
- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird das entfernte Fib-Objekt (inklusive enthaltender Fib-Elemente) aus dem Speicher gelöscht, sonst verbleibt es im Speicher. Standardwert ist `true` (=wahr), um das alte Fib-Objekt zu löschen.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn aus der Liste das entsprechende `iPositionUnderObject`'te Unterobjekt gelöscht wurde, sonst wird `false` (=falsch) zurück gegeben.

43 cComment

Die Klasse `cComment` realisiert das Anmerkungselement. Zur Beschreibung des Anmerkungselements siehe Abschnitt 12.5 auf Seite 32 .

Elternklasse: `cFibLimb`

43.1 Schnittstellenbeschreibung

43.1.1 cComment

Syntax: `cComment( const string szKey, const string szValue, cFibElement * pFibUnderObject )`

Der Konstruktor des Anmerkungselements, er erstellt einen Anmerkungselement.

Von den übergebenen Unterobjekten `pFibUnderObject` wird keine Kopie erstellt, daher darf es nicht einfach gelöscht werden.

Eingabeparameter:

- `szKey`: Der einzusetzende Schlüssel, den das erzeugte Anmerkungselement erhalten soll.
- `szValue`: Der einzusetzende Wert, den das erzeugte Anmerkungselement erhalten soll.
- `pFibUnderObject`: Ein Zeiger auf das einzusetzende Unterobjekt, den das erzeugte Anmerkungselement enthalten soll. Ist der Wert `NULL`, hat das Anmerkungselement zunächst kein Unterobjekt, dieses muss dann später gesetzt werden.

Rückgabe: keine

43.1.2 getKey

Syntax: `string getKey() const`

Diese Methode liefert den Schlüssel des Kommentars zurück.

Eingabeparameter: keine

Rückgabe: Der Schlüssel des Kommentars.

43.1.3 setKey

Syntax: `void setKey( const string szKey )`

Diese Methode setzt den Schlüssel des Kommentars auf `szKey`.

Eingabeparameter:

- `szKey`: Der Schlüssel den das Kommentar haben soll.

Rückgabe: keine

43.1.4 getValue

Syntax: `string getValue() const`

Diese Methode liefert den Wert des Kommentars zurück.

Eingabeparameter: keine

Rückgabe: Der Wert des Kommentars.

43.1.5 setValue

Syntax: `void setValue( const string szValue )`

Diese Methode setzt den Wert des Kommentars auf `szValue`.

Eingabeparameter:

- `szValue`: Der Wert den das Kommentar haben soll.

Rückgabe: keine

44 cArea

Die Klasse `cArea` realisiert das Bereichselement. Zur Beschreibung des Bereichselement siehe Abschnitt 12.6 auf Seite 34 .

Zu beachten ist, dass die Reihenfolge der Unterbereiche eines Bereichselements durch Überdeckungen Auswirkungen auf das angezeigte Multimediaobjekt haben kann. Darauf sollte beispielsweise geachtet werden, wenn neue Unterbereiche eingefügt oder die Unterbereiche sortiert werden.

Elternklasse: `cFibLimb`

44.1 Schnittstellenbeschreibung

44.1.1 cArea

Syntax: `cArea( const cVectorArea &vecSubarea, cFibElement * pFibUnderObject )`

Der Konstruktor des Bereichselements, er erstellt einen Bereichselement.

Von den übergebenen Unterobjekten `pFibUnderObject` wird keine Kopie erstellt, daher darf es nicht einfach gelöscht werden.

Eingabeparameter:

- `vecSubarea`: Der einzusetzende Bereich, den das erzeugte Bereichselement erhalten soll. Von diesem wird eine Kopie in das neue Bereichselement eingefügt.
- `pFibUnderObject`: Ein Zeiger auf das einzusetzende Unterobjekt, den das erzeugte Bereichselement enthalten soll. Ist der Wert `NULL`, hat das Bereichselement zunächst kein Unterlobjekt, dieses muss dann später gesetzt werden. Das Fib-Objekt `pFibUnderObject` wird direkt in das neue Bereichselement als Unterobjekt eingefügt, ohne dass davon eine Kopie erstellt wird.

Rückgabe: keine

44.1.2 getNumberOfSubareas

Syntax: `unsignedIntFib getNumberOfSubareas() const`

Diese Methode liefert die Anzahl der Teilbereiche zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Teilbereiche.

44.1.3 getSubarea

Syntax: `cVectorArea * getSubarea( unsignedIntFib iSubarea=1 )`

Diese Methode liefert ein Zeiger auf den `iSubarea`'ten Teilbereich zurück (die Zählung beginnt bei 1).

Eingabeparameter:

- `iSubarea`: Die Nummer des Teilbereichs der zurückgegeben werden soll. Wird kein Wert angegeben wird der erste Teilbereiche zurückgegeben. (Der Standardwert ist 1 .)

Rückgabe: Den `iSubarea`'ten Teilbereich oder `NULL`, wenn kein entsprechender `iSubarea`'ter Teilbereich existiert.

44.1.4 addSubarea

Syntax: `bool addSubarea( const cVectorArea & underArea, unsignedIntFib uiPosition=0 )`

Diese Methode fügt den übergebenen Teilbereich dem Bereich hinzu. Der Teilbereich wird an der `uiPosition`'ten Stelle in der Liste der Teilbereich hinzugefügt.

Eingabeparameter:

- `underArea`: Der Teilbereich, der zum Bereich hinzugefügt werden soll. Von diesem wird eine Kopie als neuer Teilbereich eingefügt.
- `uiPosition`: Die Position, an welcher der Teilbereich `underArea` in der Liste der Teilbereich hinzugefügt werden soll. Wenn dieser Wert 0 oder größer der Anzahl der Teilbereich ist, wird der Teilbereich `underArea` an das Ende der Liste der Teilbereich hinzugefügt. Standardwert ist 0, um den Teilbereich `underArea` an das Ende der Liste der Teilbereich hinzuzufügen.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der übergebene Teilbereich `underArea` zum Bereich hinzugefügt wurde, sonst `false` (=falsch).

44.1.5 deleteSubarea

Syntax: `bool deleteSubarea( unsignedIntFib uiSubareaPosition )`

Diese Methode löscht den `uiSubareaPosition`'ten Teilbereich aus dem Bereich. Es können alle Teilbereich gelöscht werden, solange danach noch ein Teilbereich vorhanden ist.

Eingabeparameter:

- `uiSubareaPosition`: Die Nummer des Teilbereichs, der aus dem Bereich gelöscht werden soll.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der Teilbereich an der Position `uiSubareaPosition` aus dem Bereich gelöscht wurde, sonst wird `false` (=falsch) zurückgegeben.

44.1.6 sort

Syntax: `bool sort()`

Diese Methode ordnet die Teilbereiche des Bereichs.

Zum Ordnen gehört:

- Bei allen Teilbereich deren Grenzen zwei Werte sind, ist der erste Wert kleiner als der zweite.
- Zwei Teilbereich, deren Grenzen jeweils zwei Werte sind und die sich überschneiden, werden zu einem zusammengefasst.
- Die Liste der Werte der Teilbereiche ist aufsteigend sortiert. (Teilbereiche mit kleineren Grenzwerten kommen vor Teilbereichen mit größeren Grenzwerten.)

Variablen können natürlich beim Ordnen nicht berücksichtigt werden. (Da ihr Wert unbekannt ist.)

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn die Teilbereiche im Bereich geordnet wurden, sonst `false` (=falsch).

44.1.7 getDefinedVariable

Syntax: `cFibVariable * getDefinedVariable()`

Diese Methode gibt einen Zeiger auf die vom Bereichsobjekt definierte Variable zurück.

Diese Variable kann im gesamten Unterobjekt des Bereichselements verwendet/ eingesetzt werden. Bei der Auswertung wird die Variable nacheinander mit den Werten des Bereichs belegt.

Eingabeparameter: keine

Rückgabe: Es wird ein Zeiger auf die vom Bereichsobjekt definierte Variable zurückgegeben.

45 cFunction

Die Klasse `cFunction` implementiert das Funktionselement. Zur Beschreibung des Funktionselements siehe Abschnitt 12.7 auf Seite 35 .

Elternklasse: `cFibLimb`

45.1 Schnittstellenbeschreibung

45.1.1 cFunction

Syntax: `cFunction(`
`const cUnderFunction &underFunction,`
`cFibElement * pFibUnderObject )`

Der Konstruktor des Funktionselements, er erstellt einen Funktionselement.

Von den übergebenen Unterobjekten `pFibUnderObject` wird keine Kopie erstellt, daher darf es nicht einfach gelöscht werden.

Eingabeparameter:

- `underFunction`: Die einzusetzende Unterfunktion, welche das erzeugte Funktionselement erhalten soll. Von diesem wird eine Kopie in das neue Funktionselement eingefügt.
- `pFibUnderObject`: Ein Zeiger auf das einzusetzende Unterobjekt, den das erzeugte Funktionselement enthalten soll. Ist der Wert `NULL`, hat das Funktionselement zunächst kein Unterobjekt, dieses muss dann später gesetzt werden. Das Fib-Objekt `pFibUnderObject` wird direkt in das neue Funktionselement als Unterobjekt eingefügt, ohne dass davon eine Kopie erstellt wird.

Rückgabe: keine

45.1.2 getDefinedVariable

Syntax: `cFibVariable* getDefinedVariable()`

Diese Methode gibt die Variable zurück, welche von der Funktion definiert wird.

Eingabeparameter: keine

Rückgabe: Die Variable, welche von der Funktion definiert wird.

45.1.3 getUnderFunction

Syntax: `cUnderFunction * getUnderFunction()`

Diese Methode gibt die Referenz auf die Unterfunktion der Funktion zurück.

Eingabeparameter: keine

Rückgabe: Die Referenz auf die Unterfunktion der Funktion.

45.1.4 setUnderFunction

Syntax: `void setUnderFunction(`
`const cUnderFunction & underFunction)`

Diese Methode setzt die Unterfunktion der Funktion auf die übergebene Funktion. Eingesetzt wird eine Kopie der übergebenen Unterfunktion `underFunction` . Die ersetzte Unterfunktion wird gelöscht.

Eingabeparameter:

- `underFunction`: Die Unterfunktion, welche die aktuelle Unterfunktion der Funktion ersetzen soll.

Rückgabe: keine

46 cIf

Die Klasse `cIf` realisiert das If-Element. Zur Beschreibung des If-Elements siehe Abschnitt 12.8 auf Seite 41 .

Elternklasse: `cFibBranch`

46.1 Schnittstellenbeschreibung

46.1.1 cIf

Syntax: `cIf( const cCondition &condition,`
`cFibElement * pFibObjectTrueCase,`
`cFibElement * pFibObjectFalseCase )`

Der Konstruktor des If-Elements, er erstellt einen If-Element.

Von den beiden übergebenen Unterobjekten `pFibObjectTrueCase` sowie `pFibObjectFalseCase` wird keine Kopie erstellt, sie dürfen daher nicht einfach gelöscht werden.

Eingabeparameter:

- `condition`: Die einzusetzende Bedingung, welche das erzeugte If-Element erhalten soll. Von dieser wird eine Kopie eingesetzt.
- `pFibObjectTrueCase`: Ein Zeiger auf das für einzusetzende Unterobjekt, welches das erzeugte If-Element enthalten soll, für den Fall das die Bedingung wahr (=`true`) ist. Ist der Wert `NULL`, hat das If-Element zunächst kein Unterobjekt für den wahr-Fall, dieses muss dann später gesetzt werden.

- `pFibObjectFalseCase`: Ein Zeiger auf das für einzusetzende Unterobjekt, welches das erzeugte If-Element enthalten soll, für den Fall das die Bedingung falsch (`=false`) ist. Ist der Wert `NULL`, hat das If-Element zunächst kein Unterobjekt für den falsch-Fall, dieses muss dann später gesetzt werden.

Rückgabe: keine

46.1.2 getCondition

Syntax: `cCondition *getCondition()`

Diese Methode gibt die Referenz auf die Bedingung des If-Elements zurück.

Eingabeparameter: keine

Rückgabe: Die Referenz auf die Bedingung des If-Elements.

46.1.3 setCondition

Syntax: `void setCondition( const cCondition & condition)`

Diese Methode setzt die Bedingung auf die übergebene Bedingung `condition`. Eingesetzt wird eine Kopie der übergebenen Bedingung `condition` . Die ersetzte Bedingung wird gelöscht.

Eingabeparameter:

- `condition`: Die Bedingung, welche die aktuelle Bedingung des If-Elements ersetzen soll.

Rückgabe: keine

46.1.4 getTrueCase

Syntax: `cFibElement* getTrueCase()`

Diese Methode gibt die Referenz auf das Fib-Objekt zurück, welches ausgewertet wird, wenn die Bedingung wahr ist.

Eingabeparameter: keine

Rückgabe: Die Referenz auf das Fib-Objekt, welches ausgewertet wird, wenn die Bedingung wahr ist.

46.1.5 setTrueCase

Syntax: `bool setTrueCase( cFibElement* fibObject, bool bDeleteOld=true )`

Diese Methode ersetzt das Fib-Objekt, welches ausgewertet wird, wenn die Bedingung wahr ist, auf das übergebenen Fib-Objekt `fibObject`.

Das übergebene `fibObject` wird für das Einfügen nicht kopiert.

Eingabeparameter:

- `fibObject`: Ein Zeiger auf das Fib-Objekt, welches das Unterobjekt ersetzen soll, welches ausgewertet wird, wenn die Bedingung wahr ist.
- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird vor dem Einsetzen des Fib-Objekts `fibObject` das zu ersetzende Unterobjekt (inklusive aller enthaltenden Fib-Elemente) gelöscht.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn das übergebene Fib-Objekt `fibObject` als Unterobjekt eingesetzt wurde, sonst `false` (=falsch).

46.1.6 getFalseCase

Syntax: `cFibElement* getFalseCase()`

Diese Methode gibt die Referenz auf das Fib-Objekt zurück, welches ausgewertet wird, wenn die Bedingung falsch ist.

Eingabeparameter: keine

Rückgabe: Die Referenz auf das Fib-Objekt, welches ausgewertet wird, wenn die Bedingung falsch ist.

46.1.7 setFalseCase

Syntax: `bool setFalseCase( cFibElement* fibObject, bool bDeleteOld=true )`

Diese Methode ersetzt das Fib-Objekt, welches ausgewertet wird, wenn die Bedingung falsch ist, auf das übergebenen Fib-Objekt `fibObject`.

Das übergebene `fibObject` wird für das Einfügen nicht kopiert.

Eingabeparameter:

- `fibObject`: Ein Zeiger auf das Fib-Objekt, welches das Unterobjekt ersetzen soll, welches ausgewertet wird, wenn die Bedingung falsch ist.

- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird vor dem Einsetzen des Fib-Objekts `fibObject` das zu ersetzende Unterobjekt (inklusive aller enthaltenden Fib-Elemente) gelöscht.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn das übergebene Fib-Objekt `fibObject` als Unterobjekt eingesetzt wurde, sonst `false` (=falsch).

47 cExtObject

Die Klasse `cExtObject` realisiert das Externe-Objekt-Element. Zur Beschreibung des externen Objekt Elements siehe Abschnitt 12.9 auf Seite 43 .

Elternklasse: `cFibBranch`

47.1 Schnittstellenbeschreibung

47.1.1 cExtObject

Syntax: `cExtObject( longFib lIdentifier,`
`const cVectorExtObject & vecInputValues )`

Der Konstruktor des externes Objekt Elements, er erstellt einen externes Objekt Element.

Eingabeparameter:

- `lIdentifier`: Der Identifier des externe Objekts.
- `vecInputValues`: Der Vektor der Eingabewerte für das externe Objekt.

Rückgabe: keine

47.1.2 cExtObject

Syntax: `cExtObject( longFib lIdentifier,`
`const unsignedIntFib uiNumberOfInputValues=0 )`

Der Konstruktor des externes Objekt Elements, er erstellt einen Externes-Objekt-Element.

Eingabeparameter:

- `lIdentifier`: Der Identifier des externe Objekts.
- `uiNumberOfInputValues`: Die Anzahl der Eingabewerte für das externe Objekt. Als Standard gibt es keine (0) Eingabewerte.

Rückgabe: keine

47.1.3 getIdentifier

Syntax: `longFib getIdentifier()`

Diese Methode gibt den Identifier des externe Objekts zurück.

Eingabeparameter: keine

Rückgabe: Der Identifier des externe Objekts.

47.1.4 setIdentifier

Syntax: `void setIdentifier( longFib lIdentifier )`

Diese Methode setzt die Identifier das externe Objekts auf den übergebenen Wert `lIdentifier`.

Eingabeparameter:

- `lIdentifier`: Der Identifier, den das externen Objekt haben soll.

Rückgabe: keine

47.1.5 getInputVector

Syntax: `cVectorExtObject * getInputVector()`

Diese Methode liefert ein Zeiger auf den Vektor mit den Eingabewerten des externen Objekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird einen Zeiger auf den Vektor mit den Eingabewerten.

47.1.6 getNumberOfInputValues

Syntax: `unsignedIntFib getNumberOfInputValues()`

Diese Methode gibt die Anzahl der Eingabewerte für das externe Objekt zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Eingabewerte für das externe Objekt.

47.1.7 setNumberOfInputValues

Syntax: `void setNumberOfInputValues( unsignedIntFib uiNumberOfValues )`

Über diese Methode wird die Anzahl der Eingabewerte des externen Objektelements auf `uiNumberOfValues` gesetzt.

Alle vorher nicht existierenden Eingabewerte werden erzeugt und auf den Wert 0 gesetzt. Eingabewerte, welche zuviel sind, werden gelöscht.

Eingabeparameter:

- `uiNumberOfValues`: Die Anzahl der Eingabewerte, welche das externe Objekt haben soll.

Rückgabe: keine

47.1.8 getNumberOfSubobjects

Syntax: `intFib getNumberOfSubobjects()`

Diese Methode gibt die Anzahl der Unterobjekte des externe Objekts zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Unterobjekte des externe Objekts.

47.1.9 setNumberOfSubobjects

Syntax: `void setNumberOfSubobjects( unsignedIntFib iNumberOfSubobjects, bool bDeleteOld=true )`

Über diese Methode wird die Anzahl der Unterobjekte des externen Objektelements auf `iNumberOfSubobjects` gesetzt.

Alle vorher nicht existierenden Unterobjekte werden erzeugt und auf `NULL` mit 0 Ausgabevariablen gesetzt (diese Unterobjekte existieren dann noch nicht). Unterobjekte, welche zuviel sind, werden gelöscht.

Eingabeparameter:

- `iNumberOfSubobjects`: Die Anzahl der Unterobjekte, welche das externe Objekt haben soll.
- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird vor dem Einsetzen des Fib-Objekts `fibObject` das zu ersetzende Unterobjekt (inklusive aller enthaltenden Fib-Elemente) gelöscht.

Rückgabe: keine

47.1.10 getSubobject

Syntax: `cFibElement* getSubobject( unsignedIntFib iNumberSubobject )`

Diese Methode liefert das `iNumberSubobject`'te Unterobjekt des externe Objekts zurück.

Existiert kein Unterobjekt mit der Nummer `iNumberSubobject` wird `NULL` zurückgegeben.

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts, welches zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Ein Zeiger auf das `iNumberSubobject`'te Unterobjekt des externe Objekts.

47.1.11 setSubobject

Syntax: `bool setSubobject( unsignedIntFib iNumberSubobject, cFibElement* pFibObject, bool bDeleteOld=true )`

Über diese Methode wird das `iNumberSubobject`'te Unterobjekt für das Externe-Objekt auf das übergebene Fib-Objekt `pFibObject` gesetzt.

Existiert kein Unterobjekt mit der Nummer `iNumberSubobject` wird die Anzahl der Unterobjekte auf den Wert von `iNumberSubobject` erhöht. Das `iNumberSubobject`'te Unterobjekt wird auf `pFibObject` gesetzt und alle anderen erzeugten Unterobjekte auf `NULL` mit 0 Ausgabevariablen (diese Unterobjekte existieren dann noch nicht).

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts, welches ersetzt werden soll. (Die Zählung beginnt bei 1 .)
- `pFibObject`: Ein Zeiger auf das Fib-Objekt, welches das externe Objekt als Unterobjekt verwenden soll. Von diesem wird keine Kopie erstellt, sondern es wird direkt eingefügt.
- `bDeleteOld`: Wenn `bDeleteOld` gleich `true` (=wahr) ist, wird vor dem Einsetzen des Fib-Objekts `fibObject` das zu ersetzende Unterobjekt (inklusive aller enthaltenden Fib-Elemente) gelöscht.

Rückgabe: Wenn das Fib-Objekte `pFibObject` eingesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch) .

47.1.12 getNumberOfOutputVariables

Syntax: `unsignedIntFib getNumberOfOutputVariables( unsignedIntFib iNumberSubobject ) const`

Diese Methode gibt die Anzahl der Ausgabevariablen zurück, welche das entsprechende `iNumberSubobject`'te Unterobjekt des externe Objekts besitzt.

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts, von welchem die Anzahl der Ausgabevariablen zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Die Anzahl der Ausgabevariablen für das `iNumberSubobject`'te Unterobjekte des externe Objekts.

47.1.13 setNumberOfOutputVariables

Syntax: `unsignedIntFib setNumberOfOutputVariables( unsignedIntFib iNumberSubobject, unsignedIntFib iNumberOutputVariables )`

Durch diese Methode wird die Anzahl der (verwendbaren) Ausgabevariablen für das `iNumberSubobject`'te Unterobjekt des externe Objekts auf die übergeben Anzahl `iNumberOutputVariables` gesetzt. Diesem Unterobjekt werden vom Externen-Objekt-Element dann also `iNumberOutputVariables` Variablen definiert.

Wurden vor dem Methodenaufruf weniger als `iNumberOutputVariables` Variablen definiert, werden soviele neue Variablen erzeugt und am Ende der Ausgabevariablenliste des Unterobjekt angehängt bis `iNumberOutputVariables` Variablen existieren. Wurden vorher mehr als `iNumberOutputVariables` Variablen definiert, werden vom Ende der Ausgabevariablenliste des Unterobjekts soviele Variablen gelöscht, bis in der List nur noch `iNumberOutputVariables` Variablen vorhanden sind (bzw. definiert werden) oder eine Variable versucht wird zu Löschen, die noch verwendet wird.

Wenn kein `iNumberSubobject`'tes Unterobjekt existiert, wird falsch (=`false`) zurückgegeben, ohne das Änderungen vorgenommen werden.

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts des Externen-Objekts, für das die Anzahl der Ausgabevariablen geändert werden soll. (Die Zählung beginnt bei 1 .)

- `iNumberOutputVariables`: Die Nummer der Variablen, die das Externe-Objekt für das `iNumberSubobject`'te Unterobjekt bereitstellen/ definieren soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird die neue Anzahl der Ausgabevariablen für das `iNumberSubobject`'te Unterobjekte des externe Objekts zurückgegeben.

47.1.14 getOutputVariables

Syntax: `vector<cFibVariable*> getOutputVariables( unsignedIntFib iNumberSubobject )`

Diese Methode liefert einen Vektor mit den Ausgabevariablen zurück, welche für das `iNumberSubobject`'te Unterobjekt des externe Objekts bereitgestellt werden.

Von der Methode wird ein leerer Vektor zurückgegeben, falls kein entsprechendes `iNumberSubobject`'tes Unterobjekt existiert oder für dieses keine Ausgabevariablen definiert sind.

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts des Externen-Objekts für das die Ausgabevariablen zurückgegeben werden soll.

Rückgabe: Zurückgegeben wird ein Vektor mit den Ausgabevariablen für das `iNumberSubobject`'te Unterobjekt des externe Objekts.

47.1.15 getOutputVariable

Syntax: `cFibVariable * getOutputVariable( unsignedIntFib iNumberSubobject, unsignedIntFib iNumberVariable )`

Diese Methode liefert die `iNumberVariable`'te Ausgabevariable zurück, welches das `iNumberSubobject`'te Unterobjekt des externe Objekts enthält.

Wenn keine solche Ausgabevariable existiert wird `NULL` zurückgegeben.

Eingabeparameter:

- `iNumberSubobject`: Die Nummer des Unterobjekts des Externen-Objekts, für das die Ausgabevariablen zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)
- `iNumberVariable`: Die Nummer der Ausgabevariable des Unterobjekts des Externen-Objekts, welche zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Die `iNumberVariable`'te Ausgabevariable für das vom externen Objekt verwendete `iNumberSubobject`'te Unterobjekt oder `NULL`, wenn keine solche Ausgabevariable existiert.

48 cExtSubobject

Die Klasse `cExtSubobject` realisiert das Unterobjekt-Element. Zur Beschreibung des Unterobjekt-Elements siehe Abschnitt 12.10 auf Seite 44 .

Wenn dieses Element in ein Fib-Objekt eingefügt oder es geändert wird, werden die Eingabevariablen des zugehörigen root-Elements gegebenenfalls angepasst.

Elternklasse: `cFibLeaf`

48.1 Schnittstellenbeschreibung

48.1.1 cExtSubobject

Syntax: `cExtSubobject( intFib iSubobjectNumber,`
`const cVectorExtSubobject & vecOutputValues )`

Der Konstruktor des Unterobjekt-Elements, er erstellt einen Unterobjekt-Element.

Eingabeparameter:

- `iSubobjectNumber`: Die Nummer des Unterobjekts, welches hier eingesetzt werden soll. (Die Zählung beginnt bei 1 .)
- `vecOutputValues`: Der Vektor mit den Ausgabewerten für das Unterobjekt.

Rückgabe: keine

48.1.2 cExtSubobject

Syntax: `cExtSubobject( intFib iSubobjectNumber,`
`const unsigendIntFib uiNumberOfOutputValues=0 )`

Der Konstruktor des Unterobjekt-Elements, er erstellt einen Unterobjekt-Element.

Eingabeparameter:

- `iSubobjectNumber`: Die Nummer des Unterobjekts, welches hier eingesetzt werden soll. (Die Zählung beginnt bei 1 .)
- `uiNumberOfOutputValues`: Die Anzahl der Ausgabewerte für das Unterobjekt. Als Standard gibt es keine (0) Ausgabewerte.

Rückgabe: keine

48.1.3 getNumberSubobject

Syntax: `unsignedIntFib getNumberSubobject() const`

Diese Methode gibt die Nummer des Unterobjekts zurück, also das wievielte Unterobjekt aus dem nächsten root-Element hier verwendet wird.

Eingabeparameter: keine

Rückgabe: Die Nummer des Unterobjekts.

48.1.4 setNumberSubobject

Syntax: `bool setNumberSubobject( unsignedIntFib iSubobjectNumber )`

Diese Methode setzt die Nummer des Unterobjekts, also das wievielte Unterobjekt aus dem nächsten root-Element hier verwendet wird.

Eingabeparameter:

- `iSubobjectNumber`: Die Nummer, welches das Unterobjekt sein soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn die Nummer des Unterobjekts auf den Wert `iUnterobjectNumber` gesetzt wurden, sonst `false` (=falsch).

48.1.5 getOutputVector

Syntax: `cVectorExtObject * getOutputVector()`

Diese Methode liefert ein Zeiger auf den Vektor mit den Ausgabewerten des externen Unterobjekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird einen Zeiger auf den Vektor mit den Ausgabewerten.

49 cFibSet

Die Klasse `cFibSet` realisiert das set-Element. Zur Beschreibung des set-Elements siehe Abschnitt 12.12 auf Seite 48 .

Elternklasse: `cFibLimb`

49.1 Schnittstellenbeschreibung

49.1.1 cFibSet

Syntax: `cFibSet( const unsignedIntFib iuNumberOfVariables )`

Der Konstruktor des set-Elements, er erstellt einen Set-Element.

Eingabeparameter:

- `iuNumberOfVariables`: Die Anzahl der definierten Variablen bzw. Vektorelemente, welche das set-Element enthalten soll. Dieser Wert kann nur über diesen Konstruktor gesetzt werden.

Rückgabe: keine

49.1.2 getNumberOfVariables

Syntax: `unsignedIntFib getNumberOfVariables() const`

Diese Methode gibt die Anzahl der definierten Variablen bzw. Vektorelemente zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der definierten Variablen.

49.1.3 getDefinedVariable

Syntax: `cFibVariable * getDefinedVariable( const unsignedLongFib uiPosition`

Diese Methode gibt einen Zeiger auf die `uiPosition`'te vom set-Element definierte Variable zurück.

Eingabeparameter:

- `uiPosition`: Die Nummer der von set-Element definierten Variable, welche zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Zurückgegeben wird ein Zeiger auf die `uiPosition`'te vom set-Element definierten Variable oder der Nullzeiger `NULL`, wenn kein solche existiert.

49.1.4 getDomainNr

Syntax: `unsignedIntFib getDomainNr() const`

Diese Methode gibt die Nummer des Definitionsbereichs für das set-Element zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Nummer des Definitionsbereichs für das set-Element.

49.1.5 setDomainNr

Syntax: `void setDomainNr( const unsignedLongFib uiDomainNumber )`

Diese Methode setzt die Nummer des Definitionsbereichs für das set-Element.

Eingabeparameter:

- `vecOutputVariables`: Die Nummer des Definitionsbereichs für das set-Element. Wenn 0 übergeben wird, wird der Standarddefinitionsbereich genutzt.

Rückgabe: keine

49.1.6 getNumberOfVectors

Syntax: `unsignedLongFib getNumberOfVectors() const`

Diese Methode gibt die Nummer der Vektoren bzw. Sätze von Werten zurück, die das set-Element enthält.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Nummer der Vektoren bzw. Sätze von Werten, die das set-Element enthält.

49.1.7 getVectors

Syntax: `const vector< cVectorFibSet > getVectors()`

Diese Methode gibt die Vektoren bzw. die Sätze von Werten zurück, die das set-Element enthält.

Eingabeparameter: keine

Rückgabe: Zurückgegeben werden die Vektoren bzw. die Sätze von Werten, die das set-Element enthält.

49.1.8 setVectors

Syntax: `bool setVectors( const vector< cVectorFibSet > & vecSets )`

Diese Methode setzt die Vektoren bzw. Sätze von Werten, die das set-Element enthalten soll.

Eingabeparameter:

- `vecSets`: Die Vektoren bzw. Sätze von Werten, die das set-Element enthalten soll.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn die Vektoren des set-Elements auf die Vektoren in `vecSets` gesetzt wurden, sonst `false` (=falsch).

49.1.9 getVector

Syntax: `cVectorFibSet * getVector( const unsignedLongFib uiPosition )`

Diese Methode gibt einen Zeiger auf den `uiPosition`'te Vektor bzw. Satz von Werten zurück, den das set-Element enthält.

Eingabeparameter:

- `uiPosition`: Die Nummer des Vektors, welcher zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Zurückgegeben wird ein Zeiger auf den `uiPosition`'te Vektor bzw. Satz von Werten, den das set-Element enthält, oder der Nullzeiger `NULL`, wenn kein solcher existiert.

49.1.10 setVector

Syntax: `bool setVector( const cVectorFibSet & vecSet, const unsignedLongFib uiPosition )`

Diese Methode setzt den `uiPosition`'te Vektor bzw. Satz von Werten, den das set-Element enthält, auf den übergebenen Vektor `vecSet`.

Eingabeparameter:

- `vecSet`: Der Vektor, der als `uiPosition`'ter Vektor gesetzt werden soll.
- `uiPosition`: Die Nummer des Vektors, welcher gesetzt werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der `uiPosition`'te Vektor des set-Elements auf `vecSet` gesetzt wurde, sonst `false` (=falsch).

49.1.11 addVector

Syntax: `bool addVector( const cVectorFibSet & vecSet, const unsignedLongFib uiPosition=0 )`

Diese Methode fügt den übergebenen Vektor `vecSet` (bzw. Satz von Werten) als `uiPosition`'ten Vektor im set-Element hinzu.

Eingabeparameter:

- `vecSet`: Der Vektor, der als `uiPosition`'ter Vektor hinzugefügt werden soll.
- `uiPosition`: An welcher Stelle der Vektor `vecSet` hinzugefügt werden soll. (Die Zählung beginnt bei 1 .) Wenn der Wert 0 (Standardwert) oder oder größer als die Anzahl von Vektoren im Mset-Element ist, wird der neue Vektor als letzten Vektor hinzugefügt.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn an der `uiPosition`'te Position des set-Elements der Vektor `vecSet` hinzugefügt wurde, sonst `false` (=falsch).

49.1.12 deleteVector

Syntax: `bool deleteVector( const unsignedLongFib uiPosition )`

Diese Methode löscht den `uiPosition`'ten Vektor (bzw. Satz von Werten) im set-Element.

Eingabeparameter:

- `uiPosition`: An welcher Stelle der Vektor gelöscht werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der `uiPosition`'te Vektor gelöscht wurde, sonst `false` (=falsch).

50 cFibMatrix

Die Klasse `cFibMatrix` realisiert das Matrixelement. Zur Beschreibung des Matrixelements siehe Abschnitt 12.13 auf Seite 49 .

Elternklasse: `cFibLimb`

50.1 Schnittstellenbeschreibung

50.1.1 cFibMatrix

Syntax: `cFibMatrix( const unsignedIntFib iuNumberOfDimensions, const unsignedIntFib iuNumberOfVectorElements )`

Der Konstruktor des Matrixelements, er erstellt einen Matrixelement.

Eingabeparameter:

- `iuNumberOfDimensions`: Die Anzahl der Dimensionsvariablen d, welche das Matrixelement enthalten soll. Dieser Wert kann nur über diesen Konstruktor gesetzt werden.
- `iuNumberOfVectorElements`: Die Anzahl der Werte pro Satz bzw. Vektorelemente i, welche die Vektoren des Matrixelement enthalten soll. Dieser Wert kann nur über diesen Konstruktor gesetzt werden.

Rückgabe: keine

50.1.2 getNumberOfMatrixDimensions

Syntax: `unsignedIntFib getNumberOfMatrixDimensions() const`

Diese Methode gibt die Anzahl der definierten Dimensionsvariablen d zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Dimensionsvariablen d.

50.1.3 getNumberOfVectorElements

Syntax: `unsignedIntFib getNumberOfVectorElements() const`

Diese Methode gibt die Anzahl der Werte pro Satz bzw. Vektorelemente i zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Werte pro Satz bzw. Vektorelemente i.

50.1.4 getDefinedVariable

Syntax: `cFibVariable * getDefinedVariable( const unsignedLongFib uiPosition`

Diese Methode gibt einen Zeiger auf die `uiPosition`'te vom Matrixelement definierte Variable zurück.

Eingabeparameter:

- `uiPosition`: Die Nummer der von Matrixelement definierten Variable, welche zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Zurückgegeben wird ein Zeiger auf die `uiPosition`'te vom Matrixelement definierten Variable oder der Nullzeiger `NULL`, wenn kein solche existiert.

50.1.5 getDomainNr

Syntax: `unsignedIntFib getDomainNr() const`

Diese Methode gibt die Nummer des Definitionsbereichs für das Matrixelement zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Nummer des Definitionsbereichs für das Matrixelement.

50.1.6 setDomainNr

Syntax: `void setDomainNr( const unsignedLongFib uiDomainNumber )`

Diese Methode setzt die Nummer des Definitionsbereichs für das Matrixelement.

Eingabeparameter:

- `vecOutputVariables`: Die Nummer des Definitionsbereichs für das Matrixelement. Wenn 0 übergeben wird, wird der Standarddefinitionsbereich genutzt.

Rückgabe: keine

50.1.7 getSubarea

Syntax: `cVectorArea * getArea( const unsignedIntFib uiDimension=1 )`

Diese Methode liefert ein Zeiger auf den `uiDimension`'ten Bereich $((Startvalue_{uiDimension}, Endvalue_{uiD}$ zurück (die Zählung beginnt bei 1).

Eingabeparameter:

- `uiDimension`: Die Nummer des Bereich der zurückgegeben werden soll. Wird kein Wert angegeben wird der erste Bereich zurückgegeben. (Der Standardwert ist 1 .)

Rückgabe: Den `uiDimension`'ten Bereich oder `NULL`, wenn kein entsprechender `uiDimension`'ter Bereich existiert.

50.1.8 getNumberOfVectors

Syntax: `unsignedLongFib getNumberOfVectors() const`

Diese Methode gibt die Nummer der Vektoren bzw. Sätze von Werten zurück, die das Matrixelement enthält.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Nummer der Vektoren bzw. Sätze von Werten, die das Matrixelement enthält.

50.1.9 getVectors

Syntax: `const vector< cVectorFibMatrix > getVectors()`

Diese Methode gibt die Vektoren bzw. die Sätze von Werten zurück, die das Matrixelement enthält.

Eingabeparameter: keine

Rückgabe: Zurückgegeben werden die Vektoren bzw. die Sätze von Werten, die das Matrixelement enthält.

50.1.10 setVectors

Syntax: `bool setVectors( const vector< cVectorFibMatrix > & vecSets )`

Diese Methode setzt die Vektoren bzw. Sätze von Werten, die das Matrixelement enthalten soll.

Eingabeparameter:

- `vecSets`: Die Vektoren bzw. Sätze von Werten, die das Matrixelement enthalten soll.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn die Vektoren des Matrixelements auf die Vektoren in `vecSets` gesetzt wurden, sonst `false` (=falsch).

50.1.11 getVector

Syntax: `cVectorFibMatrix * getVector( const unsignedLongFib uiPosition )`

Diese Methode gibt einen Zeiger auf den `uiPosition`'te Vektor bzw. Satz von Werten zurück, den das Matrixelement enthält.

Eingabeparameter:

- `uiPosition`: Die Nummer des Vektors, welcher zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Zurückgegeben wird ein Zeiger auf den `uiPosition`'te Vektor bzw. Satz von Werten, den das Matrixelement enthält, oder der Nullzeiger `NULL`, wenn kein solcher existiert.

50.1.12 setVector

Syntax: `bool setVector( const cVectorFibMatrix & vecSet, const unsignedLongFib uiPosition )`

Diese Methode setzt den `uiPosition`'te Vektor bzw. Satz von Werten, den das Matrixelement enthält, auf den übergebenen Vektor `vecSet`.

Eingabeparameter:

- `vecSet`: Der Vektor, der als `uiPosition`'ter Vektor gesetzt werden soll.
- `uiPosition`: Die Nummer des Vektors, welcher gesetzt werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der `uiPosition`'te Vektor des Matrixelements auf `vecSet` gesetzt wurde, sonst `false` (=falsch).

50.1.13 addVector

Syntax: `bool addVector( const cVectorFibMatrix & vecSet, const unsignedLongFib uiPosition=0 )`

Diese Methode fügt den übergebenen Vektor `vecSet` (bzw. Satz von Werten) als `uiPosition`'ten Vektor im Matrixelement hinzu.

Eingabeparameter:

- `vecSet`: Der Vektor, der als `uiPosition`'ter Vektor hinzugefügt werden soll.
- `uiPosition`: An welcher Stelle der Vektor `vecSet` hinzugefügt werden soll. (Die Zählung beginnt bei 1 .) Wenn der Wert 0 (Standardwert) oder oder größer als die Anzahl von Vektoren im Matrixelement ist, wird der neue Vektor als letzten Vektor hinzugefügt.

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn an der `uiPosition`'te Position des Matrixelements der Vektor `vecSet` hinzugefügt wurde, sonst `false` (=falsch).

50.1.14 deleteVector

Syntax: `bool deleteVector( const unsignedLongFib uiPosition )`

Diese Methode löscht den `uiPosition`'ten Vektor (bzw. Satz von Werten) im Matrixelement.

Eingabeparameter:

- `uiPosition`: An welcher Stelle der Vektor gelöscht werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurück gegeben, wenn der `uiPosition`'te Vektor gelöscht wurde, sonst `false` (=falsch).

51 cRoot

Die Klasse `cRoot` realisiert das root-Element. Zur Beschreibung des root-Elements siehe Abschnitt 12.14 auf Seite 52 .

Elternklasse: `cFibBranch`

51.1 Schnittstellenbeschreibung

51.1.1 cRoot

Syntax: `cRoot( cFibObject * pMainFibObject )`

Der Konstruktor des root-Elements, er erstellt einen root-Element.

Eingabeparameter:

- `pMainFibObject`: Ein Zeiger auf das einzusetzende Haupt-Fib-Objekt, welches das erzeugte root-Element enhalten soll. Ist der Wert `NULL`, hat das root-Element zunächst kein Haupt-Fib-Objekt, dieses muss dann später gesetzt werden. Das Fib-Objekt `pMainFibObject` wird direkt in das neue root-Element als Unterobjekt eingefügt, ohne dass davon eine Kopie erstellt wird.

Rückgabe: keine

51.1.2 setMainFibObject

Syntax: `bool setMainFibObject( cFibObject * pInMainFibObject )`

Diese Methode setzt das Haup-Fib-Object des root-Elements auf das übergebene `pInMainFibObject`.

Eingabeparameter:

- `pInMainFibObject`: Ein Zeiger auf das Fib-Objekt, welches das Haup-Fib-Object des root-Elements sein soll.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn das Haup-Fib-Object des root-Elements `pInMainFibObject` ist, sonst `false` (=falsch).

51.1.3 getMultimediaInfo

Syntax: `cMultimediaInfo *getMultimediaInfo()`

Diese Methode gibt eine Referenz auf die Multimediainformationen des root-Elements zurück.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf die Multimediainformationen des root-Elements.

51.1.4 getOptionalPart

Syntax: `cOptionalPart *getOptionalPart()`

Diese Methode gibt eine Referenz auf den optionalen Teil des root-Elements zurück.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf den optionalen Teil des root-Elements.

51.1.5 getDomains

Syntax: `cDomains * getDomains()`

Diese Methode gibt eine Referenz auf die Definitionsbereiche des root-Elements zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf die Definitionsbereiche des root-Elements.

51.1.6 getValueDomains

Syntax: `cDomains * getValueDomains()`

Diese Methode gibt eine Referenz auf die Definitionsbereiche für Werte des root-Elements zurück.

Diese Wertedefinitionsbereiche sind immer Untermengen der Definitionsbereiche. Die Definitionsbereiche für Werte sollten so klein wie möglich sein, um alle Werte (nicht aber möglich Variableninhalte) in Elementen des Fib-Objekts zu erfassen. Auf diese Weise kann Speicherplatz beim Abspeichern des Fib-Objekts gespart werden. (siehe auch 25.2.4 auf Seite 113)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf die Definitionsbereiche für Werte des root-Elements.

51.1.7 getNumberOfInputVariables

Syntax: `unsignedIntFib getNumberOfInputVariables()`

Diese Methode gibt die Anzahl der Eingabevariablen des root-Elements zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Eingabevariablen des root-Elements wird zurückgegeben.

51.1.8 setNumberOfInputVariables

Syntax: `bool setNumberOfInputVariables( unsignedIntFib iNumberOfInputVariables )`

Diese Methode setzt die Anzahl der Eingabevariablen des root-Elements.

Wurden vor dem Aufruf weniger als `iNumberOfInputVariables` Variablen definiert, werden soviele neue Variablen erzeugt und am Ende der Eingabevariablenliste des root-Elements angehängt bis `iNumberOfInputVariables` Variablen existieren/ definiert werden. Der Standardwert der Eingabevariablen ist dabei jeweils der Nullwert des (Standard-)Definitionsbereichs für die Eingabevariable.

Wurden vor den Methodenaufruf mehr als `iNumberOfInputVariables` Variablen definiert, werden vom Ende Eingabevariablenliste des root-Elements soviele Variablen gelöscht, bis in der List nur noch `iNumberOfInputVariables` Variablen vorhanden sind oder eine Variable versucht wird zu Löschen, die noch verwendet wird. (Die Standardwerte der Eingabevariablen werden natürlich mitgelöscht.)

Eingabeparameter:

- `iNumberOfInputVariables`: Die Nummer der Eingabevariablen, die das root-Elements für das Haupt-Fib-Objekt bereitstellen/ definieren soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Anzahl der Eingabevariablen `iNumberOfInputVariables` ist, sonst `false` (=falsch).

51.1.9 getInputVariables

Syntax: `list<cFibVariable*> getInputVariables()`

Diese Methode gibt eine Liste mit den Eingabevariablen des root-Elements zurück.

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Eingabevariablen des root-Elements.

51.1.10 getInputVariable

Syntax: `cFibVariable* getInputVariable( unsignedIntFib iNumberInputVariable )`

Diese Methode gibt die `iNumberInputVariable`'te Eingabevariable des root-Elements zurück. (Die Zählung beginnt bei 1 .)

Eingabeparameter:

- `iNumberInputVariable`: Die Nummer der Eingabevariable, welche zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Die `iNumberInputVariable`'te Eingabevariable des root-Elements.

51.1.11 getStandardValueOfInputVariable

Syntax: `doubleFib getStandardValueOfInputVariable( unsignedIntFib iNumberInputVariable )`

Diese Methode gibt den Standardwert der `iNumberInputVariable`'ten Eingabevariable zurück.

Eingabeparameter:

- `iNumberInputVariable`: Die Nummer der Eingabevariable, für welche der Standardwert zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Der Standardwert der `iNumberInputVariable`'ten Eingabevariable.

51.1.12 setStandardValueOfInputVariable

Syntax: `bool setStandardValueOfInputVariable( unsignedIntFib iNumberInputVariable, doubleFib dValue )`

Diese Methode setzt den Standardwert der `iNumberInputVariable`'ten Eingabevariable auf den Wert `dValue`.

Der Wert `dValue` muss im Definitionsbereich der Eingabevariable liegen, um gesetzt werden zu können. Wenn der Wert `dValue` nicht im Definitionsbereich der Eingabevariable liegt, wird das Vektorelement nicht verändert und `false` (=falsch) zurückgegeben. (Dann sollte eventuell der Definitionsbereich angepasst werden.) Werte die mit der `round`-Methode des zur Eingabevariable gehörenden Definitionsbereichs erzeugt wurden (bzw. bei denen die `isElement`-Methode `true` zurückgibt), können immer gesetzt werden.

Eingabeparameter:

- `iNumberInputVariable`: Die Nummer der Eingabevariable, für welche der Standardwert `dValue` gesetzt werden soll. (Die Zählung beginnt bei 1 .)
- `dValue`: Der Wert `dValue`, welcher als Standardwert gesetzt werden soll.

Rückgabe: Wenn der Standardwert der `iNumberInputVariable`'te Eingabevariable auf den Wert `dValue` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.1.13 getNumberOfExternSubobjects

Syntax: `unsignedIntFib getNumberOfExternSubobjects()`

Diese Methode gibt die Anzahl der möglichen verschiedenen Unterobjekte zurück, welche das Haupt-Fib-Objekt des root-Elements enthalten kann (und sollte).

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Anzahl der möglichen verschiedenen Unterobjekte.

51.1.14 checkExternSubobjects

Syntax: `unsignedIntFib checkExternSubobjects( intFib *iErrorNumber=NULL )`

Diese Methode prüft die Unterobjekte des root-Objekts.

Dabei wird geprüft:

1. Ob alle Unterobjekte, die im Haupt-Fib-Objekt des root-Elements vorkommen, auch im root-Element definiert werden. Wenn nicht, wird die Nummer, welche das ersten fehlende Unterobjekt hat, zurückgegeben.
2. Ob die Unterobjekte im Haupt-Fib-Objekt die gleiche Anzahl von Ausgabevariablen haben, wie in ihrer jeweiligen Definition im root-Element angegeben. Wenn nicht, wird die Nummer des ersten Unterobjekts, bei dem die Anzahl nicht übereinstimmt, zurückgegeben.
3. Ob alle Unterobjekte, welche im root-Element definiert werden, auch im Haupt-Fib-Objekt verwendet werden. Wenn nicht, wird die Nummer des ersten Unterobjekts, welches nicht verwendet wird, zurückgegeben.

Wenn während der Prüfung ein Fehler festgestellt wird, wird die Prüfung unterbrochen und die Nummer des Unterobjekts (nicht die Nummer in der Ordnung der Fib-Elemente) zurückgegeben bei dem der Fehler auftrat.

Im Feld auf das `iErrorNumber` zeigt wird eine eventuelle Fehlernummer zurückgegeben. Wenn der Zeiger `iErrorNumber` gleich `NULL` ist, wird keine Fehlernummer zurückgegeben. Negative Fehlernummern deuten auf Fehler hin, die behoben werden müssen, und positive Fehlernummern auf Wahrnungen.
Mögliche Fehlernummern für `iErrorNumber` sind:

0 alles in Ordnung

-1 Es gibt kein Haupt-Fib-Objekt.

-10 Ein Unterobjekt, das im Haupt-Fib-Objekt des root-Elements vorkommt, wird nicht im root-Element definiert.

-11 Ein Unterobjekte im Haupt-Fib-Objekt hat nicht die gleiche Anzahl von Ausgabevariablen, wie in seiner Definition im root-Element angegeben.

-12 Es werden nicht alle Unterobjekte, welche im root-Element definiert sind, auch im Haupt-Fib-Objekt verwendet.

-13 Es gibt Lücken bei der Nummerierung der Unterobjekte oder die Nummerierung beginnt nicht bei 1.

Eingabeparameter:

- `iErrorNumber`: Einen Zeiger auf ein Zahlenfeld in dem die Fehlernummer gespeichert werden kann. Wenn der Zeiger `iErrorNumber` gleich `NULL` ist, wird keine Fehlernummer zurückgegeben. Standardwert ist `NULL`, um keinen Fehler zurückzugeben.

Rückgabe: Zurückgegeben wird die Nummer des ersten fehlerhaften Unterobjekts des root-Elements oder 0, wenn alle Unterobjekte in Ordnung sind.

51.1.15 generateExternSubobjectsDefinitions

Syntax: `unsignedIntFib generateExternSubobjectsDefinitions( intFib *iErrorNumber=NULL )`

Diese Methode generiert die Definitionen der Unterobjekte im root-Element.

Dabei wird das Haupt-Fib-Objekt nach externen Unterobjekten durchsucht. Und für jedes gefundene Unterobjekt ein Eintrag an der richtigen Stelle (=Unterobjektnummer) in die Liste der Unterobjekte im root-Element gemacht.

Wenn externe Unterobjekten im Haupt-Fib-Objekt sich wiedersprechen, wird die Generierung abgebrochen und `false` (=falsch) zurückgegeben. Das root-Element wird dann nicht geändert.

Mögliche Fehlerquellen/ Abbruchgründe sind:

1. die Anzahl der Ausgabevariablen ist unterschiedlich für das gleiche Unterobjekt (/gleiche Unterobjektnummer)
2. es gibt Lücken bei der Nummerierung der Unterobjekt oder die Nummerierung beginnt nicht bei 1

Im Feld auf das `iErrorNumber` zeigt, wird eine eventuelle Fehlernummer zurückgegeben. Wenn der Zeiger `iErrorNumber` gleich `NULL` ist, wird keine Fehlernummer zurückgegeben. Negative Fehlernummern deuten auf Fehler hin, die behoben werden müssen, und positive Fehlernummern auf Wahrnungen.
Mögliche Fehlernummern für `iErrorNumber` sind:

0 alles in Ordnung

-1 Es gibt kein Haupt-Fib-Objekt.

-11 Die Anzahl der Ausgabevariablen ist unterschiedlich für das gleiche Unterobjekt (das heißt für die gleiche Unterobjektnummer) .

-13 Es gibt Lücken bei der Nummerierung der Unterobjekt oder die Nummerierung beginnt nicht bei 1 .

Eingabeparameter:

- `iErrorNumber`: Einen Zeiger auf ein Zahlenfeld in dem die Fehlernummer gespeichert werden kann. Wenn der Zeiger `iErrorNumber` gleich `NULL` ist, wird keine Fehlernummer zurückgegeben. Standardwert ist `NULL`, um keinen Fehler zurückzugeben.

Rückgabe: Zurückgegeben wird die Nummer des ersten fehlerhaften Unterobjekts des root-Elements oder 0, wenn alle Unterobjekte in Ordnung sind.

51.1.16 getNumberOfSubRootObjects

Syntax: `unsignedIntFib getNumberOfSubRootObjects() const`

Diese Methode gibt die Anzahl der im root-Element direkt enthaltenden Unter-root-Objekte zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der im root-Element direkt enthaltenden Unter-root-Objekte.

51.1.17 getSubRootObject

Syntax: `pair<longFib lIdentifier, cRoot* rootObject> getSubRootObject( unsignedIntFib iSubRootObjectNumber )`

Diese Methode gibt das `iSubRootObjectNumber`'te Unter-root-Objekte zurück, welches im root-Element direkt enthalten ist.

Die Rückgabe ist dabei das Paar von root-Objekt und seinem zugehörigen Identifizierer. Wenn es kein `iSubRootObjectNumber`'tes root-Objekt gibt, wird das Paar `(0,NULL)` zurückgegeben.

Eingabeparameter:

- `iSubRootObjectNumber`: Die Nummer des direkt enthaltenden root-Objekts, welches zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Zurückgegeben wird das im aktuellen root-Element direkt enthaltenden `iSubRootObjectNumber`'te Unter-root-Objekte oder, wenn es kein solches Unter-root-Objekte gibt, das Paar `(0,NULL)`.

51.1.18 getSubRootObjectNumber

Syntax: `unsignedIntFib getSubRootObjectNumber( longFib lIdentifier ) const`

Diese Methode gibt die Nummer des im root-Element direkt enthaltenden Unter-root-Objekte mit dem Identifizierer `lIdentifier` zurück oder 0, wenn kein Unter-root-Objekte mit dem Identifizierer `lIdentifier` direkt im root-Element existiert.

Eingabeparameter:

- `lIdentifier`: Der Identifizierer des root-Objekts, von welchem die Nummer (die Zählung beginnt bei 1) zurückgegeben werden soll.

Rückgabe: Die Nummer des im root-Element direkt enthaltenden Unter-root-Objekte mit dem Identifizierer `lIdentifier` oder 0, wenn kein Unter-root-Objekte mit dem Identifizierer `lIdentifier` existiert.

51.1.19 getSubRootObjectForIdentifier

Syntax: `cRoot* getSubRootObjectForIdentifier( longFib lIdentifier )`

Diese Methode gibt das im root-Element direkt enthaltenden Unter-root-Objekte mit dem Identifizierer `lIdentifier` zurück oder `NULL`, wenn kein Unter-root-Objekte mit dem Identifizierer `lIdentifier` direkt im root-Element existiert.

Eingabeparameter:

- `lIdentifier`: Der Identifizierer des root-Objekts, von welchem die Nummer zurückgegeben werden soll.

Rückgabe: Zurückgegeben wird das im root-Element direkt enthaltenden Unter-root-Objekte mit dem Identifizierer `lIdentifier` oder `NULL`, wenn kein Unter-root-Objekte mit dem Identifizierer `lIdentifier` direkt im root-Element existiert.

51.1.20 addSubRootObject

Syntax: `bool addSubRootObject( longFib lIdentifier, cRoot * pRootObject, unsignedIntFib uiPosition=0 )`

Diese Methode fügt an die Position `uiPosition` der Liste der Unter-root-Objekte des root-Elements, das übergebenen root-Objekt `pRootObject` mit dem gegebenen Identifizierer `lIdentifier` ein.

Das root-Objekt kann nicht eingefügt werden, wenn der übergebene Identifizierer `lIdentifier` im aktuellem root-Element, in dem es eingefügt werden soll, schon verwendet wird.

Eingabeparameter:

- `lIdentifier`: Der Identifizierer des root-Objekts, welches eingefügt werden soll.
- `pRootObject`: Das root-Objekt, welches eingefügt werden soll.
- `uiPosition`: Die Position in der Liste der Unter-root-Objekte des root-Elements an der das root-Objekt eingefügt werden soll. Ist `uiPosition` gleich 0 oder größer der Anzahl der Elemente in der Liste, wird das root-Objekt an das Ende der Liste angefügt. Standardwert für `uiPosition` ist 0, womit das root-Objekt an das Ende der Liste angefügt wird.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn das root-Objekt eingefügt wurde, sonst `false` (=falsch).

51.1.21 deleteSubRootObject

Syntax: `bool deleteSubRootObject( unsignedIntFib uiSubRootObjectNumber, bool bDeleteObject=true )`

Diese Methode löscht das `uiSubRootObjectNumber`’te root-Objekt aus der Liste der Unter-root-Objekte.

Das Löschen schlägt fehl, wenn es kein `uiSubRootObjectNumber`’tes root-Element in der Liste der Unter-root-Objekte gibt.

Eingabeparameter:

- `uiSubRootObjectNumber`: Die Nummer des Unter-root-Objekts, welches gelöscht werden soll. (Die Zählung beginnt bei 1 .)
- `bDeleteObject`: Wenn `true` (=wahr) wird das root-Objekt nicht nur aus der Liste entfernt, sondern auch vollständig (aus dem Speicher) gelöscht, sonst bei `false` wird es nur aus der Liste entfernt. Der Standardwert ist `true` , d.h. Standardmäßig wird das root-Objekt (aus dem Speicher) gelöscht.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn aus der Liste der Unter-root-Objekte das `uiSubRootObjectNumber`'te root-Objekt gelöscht wurde, sonst `false` (=falsch).

51.1.22 getAllSubRootObjectIdentifiers

Syntax: `list<longFib> getAllSubRootObjectIdentifiers() const`

Diese Methode liefert eine Liste mit den Identifizierern aller im root-Element direkt enthaltenden Unter-root-Objekte zurück.

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Identifizierern aller im root-Element direkt enthaltenden Unter-root-Objekte.

51.1.23 getUsedDatabaseIdentifiers

Syntax: `set< longFib > getUsedDatabaseIdentifiers()`

Diese Methode gibt alle Datenbankidentifizierer, von benutzten Datenbankobjekten, zurück, die im aktuellen root-Element angegeben sind.

Eingabeparameter: keine

Rückgabe: Zurückgegeben werden alle Datenbankidentifizierer, von benutzten Datenbankobjekten, die im aktuellen root-Element angegeben sind.

51.1.24 addUsedDatabaseIdentifier

Syntax: `void addUsedDatabaseIdentifier( const longFib lIdentifier )`

Diese Methode fügt den übergebenen Datenbankidentifizierer zu den vom aktuellen root-Element benutzten Datenbankidentifizierer hinzu.

Eingabeparameter:

- `lIdentifier`: Der Datenbankidentifizierer eines benutzten Datenbankobjekts der im root-Objekt eingefügt werden soll.

Rückgabe: keine

51.1.25 deleteUsedDatabaseIdentifier

Syntax: `bool deleteUsedDatabaseIdentifier( const longFib lIdentifier )`

Diese Methode löscht den übergebenen Datenbankidentifizierer von den vom aktuellen root-Element benutzten Datenbankidentifizierer.

Eingabeparameter:

- `lIdentifier`: Der Datenbankidentifizierer eines benutzten Datenbankobjekts der vom root-Objekt gelöscht werden soll.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Datenbankidentifizierer gelöscht wurde, sonst `false` (=falsch).

51.1.26 getChecksum

Syntax: `cVectorChecksum * getChecksum()`

Diese Methode gibt den Vektor für die Checksummeneigenschaft des root-Elements zurück oder `NULL`, falls das root-Element keine Checksummeneigenschaft hat. (siehe Abschnitt 25.2.2 auf Seite 111)

Eingabeparameter: keine

Rückgabe: Der Vektor für die Checksummeneigenschaft des root-Elements oder `NULL`, falls das root-Element keine Checksummeneigenschaft hat.

51.1.27 setChecksum

Syntax: `void setChecksum( cVectorChecksum * pChecksum=NULL )`

Diese Methode setzt die Checksummeneigenschaft des root-Elements. (siehe Abschnitt 25.2.2 auf Seite 111) Wenn der übergebene Parameter `NULL` ist, hat das root-Element keine Checksummeneigenschaft.

Der Vektor für die Checksummeneigenschaft `pChecksum` wird, wenn vorhanden, vor dem einfügen kopiert, so dass die Kopie eingefügt wird.

Eingabeparameter:

- `pChecksum`: Ein Zeiger auf den Vektor für die Checksummeneigenschaft oder `NULL`, wenn das root-Element keine Checksummeneigenschaft haben soll. Standardwert ist `NULL`.

Rückgabe: keine

51.2 cMultimediaInfo

Die Klasse `cMultimediaInfo` realisiert die Multimediainformationen des root-Elements. Zur Beschreibung der Multimediainformationen des root-Elements siehe Abschnitt 12.14.1 auf Seite 54 .

Von dieser Klasse sollte nur von root-Elementen eine Instanz erzeugbar sein, da Multimediainformationen Teil von root-Elementen ist.

51.2.1 cMultimediaInfo

Syntax: `protected cMultimediaInfo( cRoot * pMasterRoot )`

Der Konstruktor des Multimediainformationenobjekt, er erstellt einen Multimediainformationenobjekts.

Als Defaultwerte für die Fib- und Datenbankversion werden die Versionen des Programmsystems eingesetzt.

Eingabeparameter:

- `pMasterRoot`: Ein Zeiger auf das root-Element, welches diese Multimediainformationen enthält.

Rückgabe: keine

51.2.2 getFibVersion

Syntax: `unsignedLongFib getFibVersion() const`

Diese Methode gibt die Versionsnummer der Fib-Multimediabeschreibungssprache zurück, welche das Fib-Objekt benötigt.

Eingabeparameter: keine

Rückgabe: Die Versionsnummer der Fib-Multimediabeschreibungssprache, welches das Fib-Objekt benötigt.

51.2.3 setFibVersion

Syntax: `void setFibVersion( unsignedLongFib ulFibVersion )`

Mit diese Methode wird die Versionsnummer der Fib-Multimediabeschreibungssprache, welche das Fib-Objekt benötigt, auf den Wert `ulFibVersion` gesetzt.

Eingabeparameter:

- `ulFibVersion`: Die Versionsnummer der Fib-Multimediabeschreibungssprache, welche das Fib-Objekt benötigt.

Rückgabe: keine

51.2.4 getDatabaseVersion

Syntax: `unsignedLongFib getDatabaseVersion() const`

Diese Methode gibt die Versionsnummer der Fib-Objektdatenbank zurück, welche das Fib-Objekt benötigt. Dies sollte die älteste Datenbank sein, welche alle Fib-Objekte (bzw. Identifier) enthält, welche das Fib-Objekt benötigt.

Eingabeparameter: keine

Rückgabe: Die Versionsnummer der Fib-Objektdatenbank, welche das Fib-Objekt benötigt.

51.2.5 setDatabaseVersion

Syntax: `void setDatabaseVersion( unsignedLongFib ulFibDbVersion )`

Mit diese Methode wird die Versionsnummer der Fib-Objektdatenbank auf den übergebenen Wert `ulFibDbVersion` gesetzt. Dies sollte die älteste Datenbankversion sein, welche alle Fib-Objekte (bzw. Identifier) enthält, welche das Fib-Objekt benötigt.

Eingabeparameter:

- `ulFibDbVersion`: Die Versionsnummer der Fib-Objektdatenbank, welche das Fib-Objekt benötigt.

Rückgabe: keine

51.2.6 evalueMinVersionsNumbers

Syntax: `bool evalueMinVersionsNumbers()`

Diese Methode ermittelt und setzt möglichst kleine Fib-Multimediabeschreibungssprache und Fib-Objektdatenbankversionsnummern, welche das Fib-Objekt benötigt. Die Fib-Multimediabeschreibungssprache zur Fib-Versionsnummer sollte alle Elemente des Fib-Objekts enthalten. Die Fib-Objektdatenbank zur Datenbankversionsnummern sollte alle Fib-Objekte bzw. Identifier enthalten, welche für das Fib-Objekts noch extern bereitgestellt werden müssen.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Versionsnummern ermittelt und gesetzt wurden, sonst `false` (=falsch).

51.3 cOptionalPart

Die Klasse `cOptionalPart` realisiert die optionalen Informationen des root-Elements. Der Teil für die optionalen Informationen wird im Abschnitt 12.14.3 auf Seite 66 beschrieben.

Von dieser Klasse sollte nur von root-Elementen eine Instanz erzeugbar sein, da die optionalen Informationen Teil von root-Elementen sind.

51.3.1 cOptionalPart

Syntax: `protected cOptionalPart()`

Der Konstruktor des Optionalen-Informationenobjekts, er erstellt einen leeres Optionalen-Informationenobjekt.

Eingabeparameter: keine

Rückgabe: keine

51.3.2 getNumberOfEntries

Syntax: `unsignedLongFib getNumberOfEntries() const`

Diese Methode gibt die Anzahl der Einträge/ Elemente im optionalen Teil zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Einträge/ Element im optionalen Teil.

51.3.3 getEntry

Syntax: `pair<string=Key, string=Value> getEntry( unsignedLongFib ulEntryNumber ) const`

Diese Methode gibt das `ulEntryNumber`'ten Element im optionalen Teil zurück. (Die Zählung beginnt bei 1 .)

Zurückgegeben wird ein Paar von Werten/ Zeichenketten, wobei das erste Element des Paars der Schlüssel und der zweite der Wert des Eintrags ist. Ist kein `ulEntryNumber`'tes Element in der Liste der optionalen Angaben vorhanden, wird das Paar `<"","">` (=zwei leer Zeichenketten) zurückgegeben.

Eingabeparameter:

- `ulEntryNumber`: Die Nummer des Eintrags, der zurückgegeben werden soll.

Rückgabe: Das `ulEntryNumber`'ten Element im optionalen Teil oder `<"","">` , wenn kein solches Element existiert.

51.3.4 getEntries

Syntax: `list< pair<string=Key, string=Value> >`
`getEntries( string szKey, bool bFullBegin=true ) const`

Diese Methode gibt die Werte aller Elemente mit dem Schlüssel `szKey` (oder den Anfang `szKey`, wenn `bFullBegin` gleich `false` ist) in einer Liste zurück.

Eingabeparameter:

- `szKey`: Der Schlüssel, für den die Werte zurückzugeben sind.
- `bFullBegin`: Wenn `bFullBegin` gleich `true` (=wahr) ist, ist der angegebene Schlüssel der vollständige Schlüsse und es werden nur Werte zurückgegeben deren Schlüssel gleich `szKey` ist. Sonst, wenn `bFullBegin` gleich `false` (=falsch) ist, werden alle Werte zurückgegeben deren Schlüssel mit `szKey` beginnt (der Schlüssel des Eintrags kann dann auch gleich `szKey` oder länger als `szKey` sein). Der Standardwert für `bFullBegin` ist `true` (=wahr).

Rückgabe: Eine Liste mit allen Elemente mit dem Schlüssel `szKey`.

51.3.5 findKeyPart

Syntax: `list< pair<string=Key, string=Value> >`
`findKeyPart( string szKeyPart ) const`

Diese Methode gibt alle Elemente, deren Schlüssel den Teil `szKeyPart` enthält, in einer Liste zurück. Damit können beispielsweise alle optionalen Informationen zu einer Eingabevariable ("`inVarX`") zusammengesucht werden.

Eingabeparameter:

- `szKeyPart`: Ein Teil des oder der Schlüssel, für welchen die Elemente zurückzugeben sind.

Rückgabe: Zurückgegeben wird eine Liste mit allen Elemente, deren Schlüssel den Teil `szKeyPart` enthält.

51.3.6 addEntry

Syntax: `unsignedLongFib addEntry( string szKey,`
`string szValue, unsignedLongFib ulPosition=0 )`

Diese Methode fügt einen Eintrag in die Liste mit den optionalen Informationen ein. Zurückgegeben wird die Position, an welcher der Eintrag eingefügt wurde.

Eingabeparameter:

- `szKey`: Der Schlüssel des einzufügenden Eintrags.
- `szValue`: Der Wert des einzufügenden Eintrags.
- `ulPosition`: Die Position, an welcher der neue Eintrg einzufügen ist. Wenn `ulPosition` gleich 0 oder größer als die Anzahl der Elemente in der Liste mit den optionalen Informationen ist, wird der Eintrag an das Ende der Liste eingefügt. Standardwert ist 0, so dass der Eintrag am Ende der Liste eingefügt wird.

Rückgabe: Die Position, an welcher der Eintrag eingefügt wurde.

51.3.7 deleteEntry

Syntax: `bool deleteEntry( unsignedLongFib lEntryNumber )`

Diese Methode löschte den `lEntryNumber`'ten Eintrag aus den optionalen Informationen. Es wird `true` (=wahr) zurückgegeben, wenn der Eintrag gelöscht wurde, sonst `false` (=falsch).

Eingabeparameter:

- `lEntryNumber`: Die Nummer des zu löschenden Eintrags.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Eintrag gelöscht wurde, sonst `false` (=falsch).

51.3.8 deleteEntries

Syntax: `unsignedLongFib deleteEntries( string szKey,`
`bool bFullBegin=true)`

Diese Methode löscht alle Elemente mit dem Schlüssel `szKey` (oder den Anfang `szKey`, wenn `bFullBegin` gleich `false` ist) aus den optionalen Informationen.

Eingabeparameter:

- `szKey`: Der Schlüssel für den die Elemente gelöscht werden sollen.
- `bFullBegin`: Wenn `bFullBegin` gleich `true` (=wahr) ist, ist der angegebene Schlüssel der vollständige Schlüsse und es werden nur Elemente gelöscht deren Schlüssel gleich `szKey` ist. Sonst, wenn `bFullBegin` gleich `false` (=falsch) ist, werden alle Elemente gelöscht der Schlüssel mit `szKey` beginnt (der Schlüssel kann dann auch gleich `szKey` oder länger als `szKey` sein). Der Standardwert für `bFullBegin` ist `true` (=wahr).

Rückgabe: Es wird die Anzahl der gelöschten Einträge zurückgegeben.

51.4 cDomains

Die Klasse `cDomains` hält eine Liste von Definitionsbereichen bereit (z. B. für ein root-Element). Die Definitionsbereiche werden im Abschnitt 12.14.2 auf Seite 54 beschrieben.

51.4.1 cDomains

Syntax: `cDomains()`

Der Konstruktor des Definitionsbereichsobjekts, er erstellt einen Definitionsbereichsobjekt.

Eingabeparameter: keine

Rückgabe: keine

51.4.2 getNumberOfDomains

Syntax: `unsignedIntFib getNumberOfDomains() const`

Diese Methode gibt die Anzahl der Definitionsbereiche zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Definitionsbereiche.

51.4.3 getDomain

Syntax: `cDomain* getDomain(`
`unsignedIntFib iDomainNumber ) const`

Diese Methode gibt einen Zeiger auf den `iDomainNumber`'ten Definitionsbereich zurück oder `NULL` , wenn kein solcher Definitionsbereich existiert.

Sollte der `iDomainNumber`'te Definitionsbereich eine Referenz sein, wird diese direkt aufgelöst und der Definitionsbereich auf den verwiesen wird zurückgeliefert. Sollte kein Definitionsbereich für das Element existieren auf das verwiesen wird, wird der Standarddefinitionsbereich des Elements genommen. Sollte auch dieser Standarddefinitionsbereich nicht existieren wird der allgemeine Standarddefinitionsbereich genommen. Gibt es nicht den angegebenen Unterdefinitionsbereich ($Element$), wird der letzte Unterdefinitionsbereich genommen, der noch existiert.

Eingabeparameter:

- `iDomainNumber`: Die Nummer des Definitionsbereiches, der zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Einen Zeiger auf den `iDomainNumber`'ten Definitionsbereich oder `NULL` , wenn kein solcher Definitionsbereich existiert.

51.4.4 getDirectDomain

Syntax: `cDomain * getDirectDomain( unsignedIntFib iDomainNumber ) const`

Diese Methode gibt einen Zeiger auf den `iDomainNumber`'ten direkten Definitionsbereich zurück, ohne diesen aufzulösen oder zu verarbeiten.

Da die Method `getDomain()` und `getDomainForElement()` beispielsweise Referenzdefinitionsbereiche `cDomainReference` direkt auflösen, können mit dieser Methode die wirklichen Definitionsbereiche ermittelt werden.

Eingabeparameter:

- `iDomainNumber`: Die Nummer des Definitionsbereich, der zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Einen Zeiger auf den `iDomainNumber`'ten direkten Definitionsbereich.

51.4.5 getType

Syntax: `cTypeElement * getType( unsignedIntFib iDomainNumber ) const`

Diese Methode gibt einen Zeiger auf den `iDomainNumber`'ten Definitionsbereichstyp (bzw. dem Elementtyp zu dem der `iDomainNumber`'ten Definitionsbereich gehört) zurück oder `NULL` , wenn kein solcher Definitionsbereich existiert.

Eingabeparameter:

- `iDomainNumber`: Die Nummer des Definitionsbereichstyps, der zurückgegeben werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Einen Zeiger auf den `iDomainNumber`'ten Definitionsbereichstyp oder `NULL` , wenn kein solcher Definitionsbereich existiert.

51.4.6 getDomainForElement

Syntax: `cDomain* getDomainForElement( const cTypeElement & elementType ) const`

Diese Methode gibt einen Zeiger auf den Definitionsbereich für den angegebene Elementtyp `elementType` zurück oder `NULL` , wenn kein solcher Definitionsbereich existiert.

Sollte der Definitionsbereich für den angegebene Elementtyp `elementType` eine Referenz sein, wird diese direkt aufgelöst und der Definitionsbereich auf den verwiesen wird zurückgeliefert. Sollte kein Definitionsbereich für das Element auf das verwiesen wird existieren, wird der Standarddefinitionsbereich des Elements genommen. Sollte auch dieser Standarddefinitionsbereich nicht existieren wird der allgemeine Standarddefinitionsbereich genommen. Gibt es nicht den angegebenen Unterdefinitionsbereich ($Element$), wird der letzte Unterdefinitionsbereich genommen, der noch existiert.

Eingabeparameter:

- `elementType`: Der Elementtyp, für den der Definitionsbereich zurückgegeben werden soll.

Rückgabe: Einen Zeiger auf den Definitionsbereich für den angegebene Elementtyp `elementType` oder `NULL` , wenn kein solcher Definitionsbereich existiert.

51.4.7 addDomain

Syntax: `unsignedIntFib addDomain(`
```
 const cTypeElement & elementType,
 const cDomain & domain )
```

Diese Methode fügt einen neuen Definitionsbereich zu den Definitionsbereichen hinzu.

Dabei kann für jeden Elementtentyp nur ein Definitionsbereich in den jeweiligen Definitionsbereichen existieren. (Also je Definitionsbereichsliste maximal ein Eintrag je Elementtentyp.) Existiert schon ein Eintrag für den Elementtentyp in den Definitionsbereichen, wird dessen Definitionsbereich mit dem übergebenen Definitionsbereich `domain` überschrieben.

Eingabeparameter:

- `elementType`: Der Elementtype, für den der Definitionsbereich `domain` gelten soll.
- `domain`: Der Definitionsbereich, der hinzugefügt werden soll.

Rückgabe: Die Position an welcher der Definitionsbereich hinzugefügt wurde oder 0, wenn der Definitionsbereich nicht eingefügt wurde.

51.4.8 setStandardDomain

Syntax: `unsignedIntFib setStandardDomain( const cTypeElement & elementType )`

Diese Methode setzt für den Elementtyp `elementType` seinen Standarddefinitionsbereich. Existiert für den Elementtypen `elementType` noch kein Definitionsbereichseintrag, so wird dieser mit den Standarddefinitionsbereich erzeugt.

Eingabeparameter:

- `elementType`: Der Elementtyp, für den der Standarddefinitionsbereich gesetzt werden soll.

Rückgabe: Die Position (die Zählung beginnt bei 1), an welcher der Definitionsbereich gesetzt wurde.

51.4.9 deleteDomain

Syntax: `bool deleteDomain( unsignedIntFib iDomainNumber )`

Diese Methode löscht den `iDomainNumber`'ten Definitionsbereich.

Eingabeparameter:

- `iDomainNumber`: Die Nummer der Position, von der der Definitionsbereich gelöscht werden soll. (Die Zählung beginnt bei 1 .)

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Definitionsbereich gelöscht wurde, sonst `false` (=falsch).

51.4.10 deleteDomainForElement

Syntax: `bool deleteDomainForElement( const cTypeElement & elementType )`

Diese Methode löscht den Definitionsbereich für den `elementType` Elemententyp.

Eingabeparameter:

- `elementType`: Der Elemententyp, für den der Definitionsbereich gelöscht werden soll.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Definitionsbereich gelöscht wurde, sonst `false` (=falsch).

51.5 cDomainElement

Die Klasse `cDomainElement` bestimmt einen Definitionsbereich für ein Element.

Sie ist eine Hilfsklasse der Klasse `cDomains` und kann damit nur von dieser erzeugt werden.

51.5.1 cDomainElement

Syntax: `protected cDomainElement(`
```
 const cTypeElement & elementType,
 const cDomain *domain=NULL )
```

Der Konstruktor des Definitionsbereichsobjekts, er erstellt einen Definitionsbereichsobjekts.

Wenn der Definitionsbereich `domain` gleich `NULL` (Standardwert) oder inkompatibel zum gegebenen Elementtyp `elementType` ist, wird der Standarddefinitionsbereich für den angegebenen `elementType` gesetzt. (siehe Tabelle 5 auf Seite 61)

Eingabeparameter:

- `elementType`: Eine Referenz auf den Typ des Elements, für den der Definitionsbereich steht.
- `domain`: Einen Zeiger auf den Definitionsbereich für den übergebenen Elementtyp `elementType`. Standardwert ist `NULL`, womit der Standarddefinitionsbereich für den Elementtyp gesetzt wird.

Rückgabe: keine

51.5.2 getElementType

Syntax: `cTypeElement * getElementType() const`

Diese Methode liefert den Typ des Elements zurück, für den der Definitionsbereich gilt.

Eingabeparameter: keine

Rückgabe: Der Typ des Elements für den der Definitionsbereich gilt.

51.5.3 getDomain

Syntax: `cDomain * getDomain() const`

Diese Methode liefert ein Zeiger auf den Definitionsbereich zurück, der für den Typ des Elements gilt.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird einen Zeiger auf den Definitionsbereich, der für den Typ des Elements gilt.

51.5.4 setDomain

Syntax: `bool setDomain( const cDomain & domain )`

Diese Methode setzt den der Definitionsbereich der für den Typ des Elements gilt.

Dabei wird der Definitionsbereich dahingehend geprüft, ob er zum Typ des Elements kompatible bzw. auf in anwendbar ist. Sollte der Definitionsbereich zum Typ inkompatibel sein (z. B. ein Definitionsbereich mit Vektoren für die Unterbereichselemente) wird der Definitionsbereich nicht gesetzt und `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `domain`: Der Definitionsbereich, welcher für den Typ des Elements gelten soll.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Definitionsbereich gesetzt wurde, sonst `false` (=falsch).

51.6 cTypeElement

Die Klasse `cTypeElement` ist die Basisklasse für alle Elementtypen. Alle Klassen welche einen Typ für ein Fib-Element darstellen, für das Definitionsbereiche festgelegt werden können, sind von der Klasse `cTypeElement` abgeleitet.

Von der Klasse `cTypeElement` selbst können keine Instanzen erzeugt werden.

Im Nachfolgenden werden die Methoden von `cTypeElement` beschrieben.

Danach folgen Kurz die abgeleiteten Klassen bzw. Elementtypen. Mit einer Beschreibung für welche Elemente sie gelten, welchen Standarddefinitionsbereich sie haben und zu welchen Definitionsbereiche sie kompatibel sind. Diese Parameter bestimmen auch das Verhalten ihrer Methoden.

Für die Angabe der möglichen Typen sei auf Tabelle 5 auf Seite 61 verwiesen.

In Diagramm 16 ist eine Klassenhierarchie der Elementtypen dargestellt.

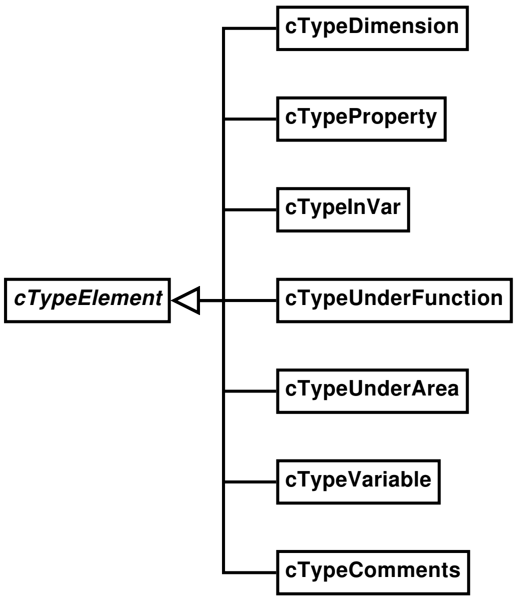


Abbildung 16: Klassengraph Fib-Elementtypen

51.6.1 getType

Syntax: `unsignedIntFib getType()`

Diese Methode gibt eine Nummer für den Typ des Elements zurück.

Die Nummern für die Typen sind:

- 1 für cTypeDimension
- 2 für cTypeSubarea
- 3 für cTypeUnderFunction
- 5 für cTypeInVar
- 6 für cTypeProperty
- 10 für cTypeVariable
- 11 für cTypeComments
- 12 für cTypeExternObject
- 13 für cTypeExtSubobject
- 14 cTypeFibSet

- 15 cTypeFibMatrix
- 16 cTypeExtObjectInput

Eingabeparameter: keine

Rückgabe: Der Typ des Vektors.

51.6.2 isCompatible

Syntax: `bool isCompatible( const cDomain & domain )`

Diese Methode gibt zurück, ob der Elemententyp kompatibel zum angegebenen Definitionsbereiche `domain` ist.

Eingabeparameter:

- `domain`: Eine Referenz auf den Definitionsbereich, welcher kompatibel zum Typ des Elements sein soll.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Definitionsbereich `domain` kompatibel ist, sonst `false` (=falsch).

51.6.3 getStandardDomain

Syntax: `cDomain * getStandardDomain()`

Diese Methode gibt einen Zeiger auf den Standarddefinitionsbereich des Elemententyps zurück.

Das Objekt zum Standarddefinitionsbereich wird nicht automatisch gelöscht. Das Löschen des Objekts sollte also nach der Verwendung sichergestellt werden.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf den Standarddefinitionsbereich des Elemententyps.

51.6.4 equalElementType

Syntax: `bool equalElementType(`
` const cTypeElement & typeElement ) const`

Diese Methode prüft, ob der übergebene Elementtyp für `typeElement` mit dem aktuellen Elementtyp übereinstimmt. Zwei Elementtype stimmen überein, wenn sie zum gleichen Element gehören. Demnach darf es keine zwei Definitionsbereich zum gleichen Elementtyp gleichzeitig geben.

Der ein Positionselementtyp ist beispielsweise zu allen anderen möglichen Positionselementtypen gleich, unabhängig von deren Dimension und Dimensionsmapping, da es nur eine Art von Positionsvektoren geben darf. Es gibt aber viele verschiedene Elementtypen für die Eigenschaftstypen, da es für verscheidene Eigenschaften verschiedene Vektoren geben kann.

Eingabeparameter:

- `typeElement`: Der Elementtyp, zu mit dem aktuelle Elementtyp übereinstimmt soll.

Rückgabe: Wenn der aktuelle Elementtyp mit dem übergebenen `typeElement` Elementtyp übereinstimmt, wird `true` (=wahr) zurückgegeben, sonst wird `false` (=falsch) zurückgegeben.

51.6.5 equal

Syntax: `bool equal( const cTypeElement & typeElement)const`
Syntax: `bool operator==( const cTypeElement & typeElement ) const`

Diese Methode prüft, ob der übergebene Elementtyp `typeElement` gleich zum aktuellen Elementtyp ist.

Eingabeparameter:

- `typeElement`: Der Elementtyp, zu dem der aktuelle Elementtyp gleich sein soll.

Rückgabe: Wenn der aktuelle Elementtyp zum Elementtyp `typeElement` gleich ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.6.6 cTypeDimension

Typ für Element: cVectorPosition; Die Nummer für die Richtungen der Dimension werden über den Konstruktor bestimmt.

Standarddefinitionsbereich:
$vector(2, naturalNumberB(16), naturalNumberB(16))$

Kompatible Definitionsbereiche: Vektoren

Inkompatible Definitionsbereiche: alle einzelnen Zahlen

Konstanten Folgende Konstanten werden von der Klasse `cTypeDimension` definiert.

- `static const unsignedIntFib DIRECTION_NONE  = 0`
- `static const unsignedIntFib DIRECTION_HORIZONTAL=1`
- `static const unsignedIntFib DIRECTION_VERTICAL=2`
- `static const unsignedIntFib DIRECTION_DEPTH = 3`
- `static const unsignedIntFib DIRECTION_TIME  = 4`

cTypeDimension

Syntax: `cTypeDimension( unsigendIntFib iNumberOfDimensions=2 )`

Der Konstruktor des Dimensionstypobjekts, er erstellt einen Dimensionstypobjekt.

Den Dimensionen werden Initial die Werte 1 bis `iNumberOfDimensions` zugeordnet, wie sie in Tabelle 7 auf Seite 64 aufgeführt sind. Bei zwei Dimensionen (Standardwert) wäre nach dem Erzeugen des Multimediainformationenobjekts die erste Dimension in Richtung “horizontal” und die zweite in Richtung “vertikal”.

Eingabeparameter:

- `iNumberOfDimensions`: Die Anzahl der Dimensionen, welches das Fib-Objekt haben soll. Standardwert ist 2 .

Rückgabe: keine

cTypeDimension

Syntax: `cTypeDimension( vector<unsigendIntFib> vecDimensionsMapping )`

Der Konstruktor des Dimensionstypobjekts, er erstellt einen Dimensionstypobjekt.

Eingabeparameter:

- `vecDimensionsMapping`: Der Vektor mit den Mappingwerten für die Dimensionen. Die Werte werden in Tabelle 7 auf Seite 64 definiert. Der i'te Vektorwert steht für die $i+1$'te Dimension. Über diesen Vektor wird auch die Anzahl der Dimensionen bestimmt. Wiederholt auftauchende Mappingwerte werden auf den Mappingwert 0 für “none” gesetzt.

Rückgabe: keine

getNumberOfDimensions

Syntax: `unsigendIntFib getNumberOfDimensions() const`

Diese Methode gibt die Anzahl der Dimensionen zurück, für welche das Objekt steht.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Dimension, für welche das Objekt steht.

getDimensionMapping

Syntax: `unsignedIntFib getDimensionMapping( unsignedIntFib iDimensionNumber ) const`

Diese Methode gibt den Mappingwert für die Dimension `iDimensionNumber` zurück. Existiert keine solche Dimension, wird 0 für `none` zurückgegeben.

Die Werte werden in Tabelle 7 auf Seite 64 definiert.

Eingabeparameter:

- `iDimensionNumber`: Nummer der Dimension, für welche der Mappingwert zurückgegeben werden soll.

Rückgabe: Den Mappingwert für die Dimension `iDimensionNumber`, existiert keine solche Dimension wird 0 zurückgegeben.

setDimensionMapping

Syntax: `bool setDimensionMapping( unsignedIntFib iDimensionNumber, unsignedLongFib lMapping )`

Diese Methode setzt den Mappingwert für die Dimension `iDimensionNumber`. Die Werte werden in Tabelle 7 auf Seite 64 definiert.

Wird versucht ein Mapping einzustellen, auf welches schon ein anderen Dimension gesetzt ist, wird das Mapping nicht geändert und `false` (=falsch) zurückgegeben. Jede Dimension kann allerdings immer auf 0 (=kein Mapping) gesetzt werden.

Wenn kein entsprechendes `iDimensionNumber`'tes Dimensionsmapping existiert, wird `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `iDimensionNumber`: Nummer der Dimension, für welche das Mapping geändert werden soll.
- `lMapping`: Neuer Wert für das Mapping der Dimension.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn das Mappingwert gesetzt wurden, sonst `false` (=falsch).

getDimensionMappingName

Syntax: `static string getDimensionMappingName( unsignedLongFib ulMapping )`

Diese Methode liefert Namen für den den Mappingwert `ulMapping`. Die Werte und Namen werden in Tabelle 7 auf Seite 64 definiert und einander zugeordnet.

Eingabeparameter:

- `ulMapping`: Der Wert für das Mapping, für welche der Name zurückgegeben werden soll.

Rückgabe: Den Namen für den Mappingwert `ulMapping`.

getUnit

Syntax: `vector<string> getUnit() const`

Diese Methode gibt einen Vektor mit den Zeichenketten mit den (SI-) Einheiten für die Vektorelemente/Dimensionen des Typs zurück.

Ist eine Einheit unbekannt (bzw. die Eigenschaft ist kein Standardtyp), wird eine leere Zeichenkette für sie zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Vektor mit den Zeichenketten mit den (SI-) Einheiten für die Vektorelemente des Typ.

51.6.7 cTypeArea

Typ für Element: cArea

Standarddefinitionsbereich: $vector(2, naturalNumberB(8), vector(2, integerB(16), integerB(16)))$

Dieser Typ ist für Definitionsbereich für das Bereichselement (siehe Abschnitt 12.6 auf Seite 34 und Abschnitt 25.3.6 auf Seite 133) im Haupt-Fib-Objekt. Der zugehörige Definitionsbereich ist ein Vektordefinitionsbereich mit 2 Elementen /Unterdefinitionsbereichen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Unterbereiche, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite und letzte Element ist der Definitionsbereich für die Vektoren für die Unterbereiche (B_1) und ist ein Definitionsbereich für Vektoren dessen zwei Elemente ganze Zahlen sind.

Kompatible Definitionsbereiche: alle Vektoren mit zwei Elementen, deren erstes Element natürliche Zahlen ist und deren zweites Element ein Vektor mit zwei ganzen Zahlen ist

Inkompatible Definitionsbereiche: alle einzelnen Zahlen, alle Vektoren mit nicht zwei Unterdefinitionsbereichen und Vektoren in denen das erstes Element keine natürliche Zahlen ist oder der zweite Unterdefinitionsbereich kein Vektor mit zwei ganzen Zahlen ist

51.6.8 cTypeUnderFunction

Typ für Element: cUnderFunction; Die Zahlen in Funktionen.

Standarddefinitionsbereich: $naturalNumberB(16)$

Kompatible Definitionsbereiche: alle einzelnen Zahlen

Inkompatible Definitionsbereiche: Vektoren

51.6.9 cTypeInVar

Typ für Element: Eingabevariable $inVar_i$; Die Nummer der Eingabevariable wird über den Konstruktor bestimmt. Nur Eingabevariablen des aktuellen root-Elements können hier auftauchen.

Standarddefinitionsbereich: $naturalNumberB(16)$

Kompatible Definitionsbereiche: alle einzelnen Zahlen

Inkompatible Definitionsbereiche: Vektoren

cTypeInVar

Syntax: `cTypeInVar( unsignedIntFib iNumberInputVariable )`

Der Konstruktor des Eingabevariablentypobjekts, er erstellt einen Eingabevariablentypobjekt.

Eingabeparameter:

- `iNumberInputVariable`: Die Nummer der Eingabevariable, für welche das Objekt steht.

Rückgabe: keine

getNumberOfInputVariable

Syntax: `unsignedIntFib getNumberOfInputVariable() const`

Diese Methode gibt die Nummer der Eingabevariable zurück, für das das Objekt steht.

Eingabeparameter: keine

Rückgabe: Die Nummer der Eingabevariable, für welche das Objekt steht.

51.6.10 cTypeProperty

Typ für Element: cVectorProperty; Der Typ der Eigenschaft wird über den Konstruktor bestimmt.

Standarddefinitionsbereich: $vector(1, naturalNumberB(16))$
Für einige Eigenschaften ist der Standarddefinitionsbereich des Eigenschaftsvektors von $vector(1, naturalNumberB(16))$ abweichend. In Tabelle 5 auf Seite 61 sind die abweichenden Standarddefinitionsbereiche aufgeführt.

Kompatible Definitionsbereiche: Vektoren deren Elemente einzelnen Zahlen sind

Inkompatible Definitionsbereiche: alle einzelnen Zahlen

Konstanten Folgende Konstanten werden von der Klasse `cTypeProperty` definiert:

- `static const unsignedIntFib COLOR_RGB = 1`
- `static const unsignedIntFib COLOR_GRAYSCALE  = 2`
- `static const unsignedIntFib LAYER     = 100`
- `static const unsignedIntFib TRANSPARENCY    = 200`
- `static const unsignedIntFib SOUND     = 300`
- `static const unsignedIntFib SOUND_POLARIZED = 301`
- `static const unsignedIntFib SOUND_AMPLITUDE = 305`

- `static const unsignedIntFib SOUND_BARRIER   = 310`
- `static const unsignedIntFib SOUND_REFLECTED = 311`
- `static const unsignedIntFib SOUND_DAMPING   = 312`
- `static const unsignedIntFib KELVIN    = 400`
- `static const unsignedIntFib ELECTRO_MAGNETIC= 410`

cTypeProperty

Syntax: `cTypeProperty( intFib iTypeProperty )`

Der Konstruktor des Eigenschaftstypobjekts, er erstellt einen Eigenschaftstypobjekt.

Die Nummer des Eigenschaftstyps sind die Werte aus Tabelle 2 auf Seite 30 .

Eingabeparameter:

- `iTypeProperty`: Die Nummer des Typs, der Eigenschaft für das das Objekt steht. Die möglichen Werte für die unterschiedlichen Eigenschaften sind aus Tabelle 2 auf Seite 30 aus der Spalte “Wert” zu entnehmen.

Rückgabe: keine

cTypeProperty

Syntax: `cTypeProperty( unsignedIntFib uiPropertyType, unsignedIntFib uiNumberOfDimensions=2 )`

Der Konstruktor des Eigenschaftstypobjekts, er erstellt einen Eigenschaftstypobjekts.

Eingabeparameter:

- `uiPropertyType`: Die Nummer der Eigenschaft, für welche das Objekt steht.
- `uiNumberOfDimensions`: Die Nummer der der Dimensionen der Eigenschaft. Bei manche Eigenschaften ist die Anzahl der Element von der Dimension abhängig (z.B $soundPolarized$), deshalb wird diee Angabe benötigt.

Rückgabe: keine

getNumberOfProperty

Syntax: `unsignedIntFib getNumberOfProperty()`

Diese Methode gibt die Nummer der Eigenschaft zurück, für welche das Objekt steht. In Tabelle 2 auf Seite 30 sind die Nummern der Eigenschaften ind der Spalte "Wert" aufgeführt.

Eingabeparameter: keine

Rückgabe: Die Nummer der Eigenschaft für das das Objekt steht.

getNumberOfProperty

Syntax: `intFib getNumberOfTypeProperty() const`

Diese Methode gibt die Nummer des Typs der Eigenschaft zurück, für welchen das Objekt steht.

Die Nummer des Eigenschaftstyps sind die Werte aus Tabelle 2 auf Seite 30 in der Spalte "Wert" aufgeführt.

Eingabeparameter: keine

Rückgabe: Die Nummer der des Typs der Eigenschaft, für welche das Objekt steht.

getUnit

Syntax: `vector<string> getUnit() const`

Diese Methode gibt einen Vektor mit den Zeichenketten mit den (SI-) Einheiten für die Vektorelemente des Typs zurück.

Ist eine Einheit unbekannt (bzw. die Eigenschaft ist kein Standardtyp), wird eine leere Zeichenkette für sie zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Vektor mit den Zeichenketten mit den (SI-) Einheiten für die Vektorelemente des Eigenschaftstyps.

51.6.11 cTypeVariable

Typ für Element: cVariable

Standarddefinitionsbereich: $naturalNumberB(8)$

Der zugehörige Definitionsbereich dient für die Werte die benötigt werden, um Variablen im komprimierten Speicherformat zu kodieren. Der Definitionsbereich sollten die natürliche Zahlen von 0 bis maximale Anzahl der definierten Variablen in den Fib-Blättern im Haupt-Fib-Objekt sein. Das Fib-Baum-Blatt im Haupt-Fib-Objekt über dem meisten Variablen definiert werden, bestimmt also den Definitionsbereich, bzw. der Ast mit den meisten definierten Variablen. Dieser Eintrag wird beim Abspeichern erstellt.

Kompatible Definitionsbereiche: alle natürlichen, unskalierte Zahlen

Inkompatible Definitionsbereiche: alle einzelnen Zahlen die nicht natürlichen, unskalierte Zahlen sind und alle Vektoren

51.6.12 cTypeComments

Typ für Element: cComment

Standarddefinitionsbereich: $naturalNumberB(8)$

Der zugehörige Definitionsbereich dient für die Werte die benötigt werden, um Kommentare im komprimierten Speicherformat zu kodieren. Der Definitionsbereich sollten die natürliche Zahlen von 0 bis Anzahl der Kommentare im Haupt-Fib-Objekt sein. Dieser Eintrag wird beim Abspeichern erstellt.

Kompatible Definitionsbereiche: alle natürlichen, unskalierte Zahlen

Inkompatible Definitionsbereiche: alle einzelnen Zahlen die nicht natürlichen, unskalierte Zahlen sind und alle Vektoren

51.6.13 cTypeExtObjekt

Typ für Element: cExtObjekt

Standarddefinitionsbereich:

$vector(4, integerB(32), naturalNumberB(4), naturalNumberB(2), naturalNumberB(2))$

Dieser Typ ist für Definitionsbereich für externe Objekte im Haupt-Fib-Objekt. Der Definitionsbereich ist ein Vector mit 4 Elementen. Die Vektorelemente dienen der Reihenfolge für den Identifier, die Anzahl der Eingabevariablen NumberInVar, die Anzahl der Unterobjekte NumberUnderObjects und die Anzahl der Ausgabevariablen NumberOutVar. Alle Vektorelementdefinitionsbereiche, außer dem für den Identifier, kommen aus den natürlichen Zahlen. Der Vektorelementdefinitionsbereich für den Identifier kommt aus den Ganzzahlen. Dieser Definitionsbereich wird normalerweise beim Abspeichern erstellt.

Kompatible Definitionsbereiche: Vektoren mit 4 Elementen deren erstes Elemente eine einzelnen Ganzzahlen ist und dessen anderen Elemente natürliche Zahlen sind, alle Vektorelemente sind unskaliert

Inkompatible Definitionsbereiche: alle einzelnen Zahlen, alle Vektoren die nicht kompatible sind

51.6.14 cTypeExtObjectInput

Typ für Element: Eingabewerte von externen Objekten (cExtObjekt); Der Identifier des externen Objekts wird über den Konstruktor bestimmt.

Standarddefinitionsbereich: $vectorOpenEnd(integerB(8))$

Kompatible Definitionsbereiche: Vektoren deren Elemente einzelnen Zahlen sind

Inkompatible Definitionsbereiche: alle einzelnen Zahlen

cTypeExtObjectInput

Syntax: `cTypeExtObjectInput( long lIdentifier )`

Der Konstruktor des Eingabewertetypobjekts für externen Objekten, er erstellt einen Eingabewertetypobjekt für externen Objekten mit dem Identifier lIdentifier.

Eingabeparameter:

- `lIdentifier`: Der Identifier des externen Objekts, für welche das Objekt steht.

Rückgabe: keine

getIdentifier

Syntax: `long getIdentifier() const`

Diese Methode gibt den Identifier des externen Objekts zurück, für dessen Eingabewerte das Objekt steht.

Eingabeparameter: keine

Rückgabe: Die Identifier des externen Objekts, für dessen Eingabewerte das Objekt steht.

51.6.15 cTypeExtSubobject

Typ für Element: für das externe Subobjekt (cExtSubobject); Die Nummer des externen Subobjekts wird über den Konstruktor bestimmt. Nur externe Subobjekte des Haup-Fib-Objects des aktuellen root-Elements können hier auftauchen.

Standarddefinitionsbereich: $vector(0)$

Kompatible Definitionsbereiche: Vektoren deren Elemente einzelnen Zahlen sind

Inkompatible Definitionsbereiche: alle einzelnen Zahlen

cTypeExtSubobject

Syntax: `cTypeExtSubobject( const unsignedIntFib iNumberExtSubobject )`

Der Konstruktor des externen Subobjekt Typs, er erstellt einen externe Subobjekt Type.

Eingabeparameter:

- `iNumberExtSubobject`: Die Nummer des externen Subobjekts, für welche das Objekt steht.

Rückgabe: keine

getNumberOfExtSubobject

Syntax: `unsignedIntFib getNumberOfExtSubobject() const`

Diese Methode gibt die Nummer des externen Subobjekts zurück, für das das Objekt steht.

Eingabeparameter: keine

Rückgabe: Die Nummer der externen Subobjekts, für welche das Objekt steht.

51.6.16 cTypeFibSet

Typ für Element: cTypeFibSet

Standarddefinitionsbereich: $vector(3, naturalNumberB(8), naturalNumberB(32), vectorOpenEnd(1, integerB(32)))$

Dieser Typ ist für Definitionsbereich für das set-Element (siehe Abschnitt 12.12 auf Seite 48 und Abschnitt 49 auf Seite 238) im Haupt-Fib-Objekt. Der Definitionsbereich ist ein Vector mit 3 Elementen. Das erste Element bzw. der erste Unterdefinitionsbereich dient für die Anzahl (n) der Variablen und der zu setzenden Werte pro Satz, er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich dient für die Anzahl (k) der Sätze mit zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{i.g}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind.

Kompatible Definitionsbereiche: alle Vektoren mit drei Elementen, deren ersten beiden Elemente natürliche Zahlen sind und deren drittes Element ein beliebiger Vektor mit Zahlen ist

Inkompatible Definitionsbereiche: alle einzelnen Zahlen, alle Vektoren mit nicht drei Unterdefinitionsbereichen und Vektoren in denen die ersten beiden Elemente keine natürliche Zahlen sind oder der dritte Unterdefinitionsbereich kein Vektor von Zahlen ist

51.6.17 cTypeFibMatrix

Typ für Element: cTypeFibMatrix

Standarddefinitionsbereich: $vector(4, naturalNumberB(8), naturalNumberB(32), vector(2, integerB(16), integerB(16)), vectorOpenEnd(1, integerB(32)))$

Dieser Typ ist für Definitionsbereich für das Matrixelement (siehe Abschnitt 12.13 auf Seite 49 und Abschnitt 50 auf Seite 242) im Haupt-Fib-Objekt. Der Definitionsbereich ist ein Vector mit 4 Elementen. Das erste Element bzw. der erste Unterdefinitionsbereich ist für die Anzahl (d) der Dimensionsvariablen, die Anzahl (i) der Wertevariablen und die zu setzenden Werte i pro Satz. Er ist ein Definitionsbereich aus den natürlichen Zahlen. Das zweite Element bzw. der zweite Unterdefinitionsbereich ist für die Anzahl (k) der Sätze mit den zu setzenden Werten. Er ist auch ein Definitionsbereich aus den natürlichen Zahlen. Das dritte Element ist der Definitionsbereich für die Bereiche bzw. Start und Endwerte für die einzellenen Dimensionsvariablen, er ist ein Vektordefinitionsbereich mit zwei Elementen, welche jeweils aus den ganzen Zahlen kommen. Das vierte und letzte Element ist der Definitionsbereich für die Vektoren für die zu setzenden Werte ($W_{a.b}$) und ist ein Definitionsbereich für Vektoren deren Elemente einfache Zahlen (skalare) sind.

Kompatible Definitionsbereiche: alle Vektoren mit vier Elementen, deren ersten beiden Elemente natürliche Zahlen sind, deren drittes Element ein Vektor mit zwei Ganzzahlen und deren viertes Element ein beliebiger Vektor mit Zahlen ist

Inkompatible Definitionsbereiche: alle einzelnen Zahlen, alle Vektoren mit nicht vier Unterdefinitionsbereichen und Vektoren in denen die ersten beiden Elemente keine natürliche Zahlen sind, Vektoren in denen der dritte Unterdefinitionsbereich kein Vektor von zwei Ganzzahlen ist oder der vierte Unterdefinitionsbereich kein Vektor von Zahlen ist

51.7 Definitionsbereiche cDomain

Definitionsbereiche dienen zur Eingrenzung der möglichen Werte, welche ein Element einnehmen kann.

Es gibt zwei große Klassen von Definitionsbereichen: Vektoredefinitionsbereiche und Definitionsbereiche für einzelne (skalare) Werte/ Zahlen.

In Diagramm 17 ist eine Klassenhierarchie der Definitionsbereiche dargestellt.

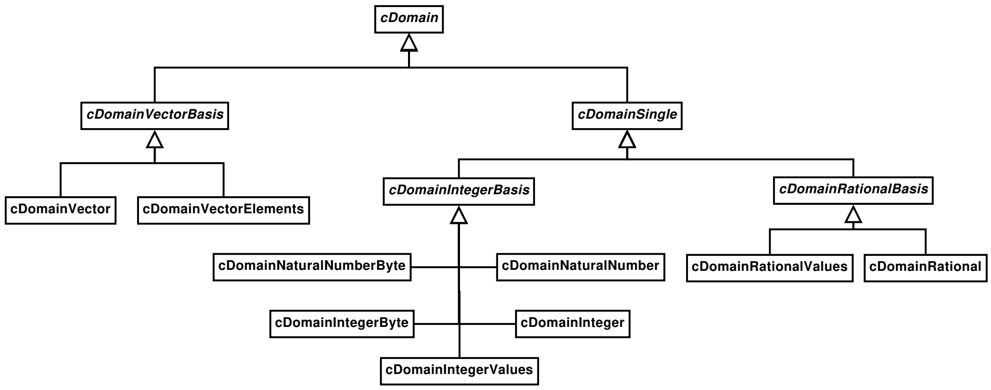


Abbildung 17: Klassengraph der Fib-Definitionsbereiche

51.7.1 getType

Syntax: `string getType()`

Diese Methode gibt den Typ des Definitionsbereichs als Zeichenkette zurück. Dies ist der Klassenname ohne das vorangestellte Zeichen "c". Beispielsweise geben Objekte der Klasse `cDomainNaturalNumberBit` die Zeichenkette "DomainNaturalNumberBit" als Typ zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Typ des Definitionsbereichs als Zeichenkette.

51.8 Einfache Definitionsbereiche cDomainSingle

Die Klasse `cDomainSingle` ist der Basisklasseh aller einzelnen Zahlen (Skalare). Von `cDomainSingle` werden alle Definitionsbereiche für einfache Zahlen abgeleitet. Von `cDomainSingle` selbst kann jedoch keine Instanz erzeugt werden.

Nach den folgenden Methoden der Klasse `cDomainSingle` sind die von ihr abgeleiteten Klassen aufgeführt.

51.8.1 isElement

Syntax: `bool isElement( const doubleFib dValue ) const`

Diese Methode prüft, ob der übergebene Wert `dValue` im (skalierten) Definitionsbereich liegt.

Diese Methode ist mit Vorsicht zu verwenden, da aufgrund von Rundungsfehlern bei Gleitkommawerten (`doubleFib`) diese leicht voneinander abweichen können. Für Werte die von der `round()` Methode zurückgegeben werden, gibt die `isElement()` Methode `true` (=wahr) zurück. Wenn aber beispielsweise zum Rückgabewert W der `round()` Methode 3 addiert und wieder abgezogen wird ($W + 3 - 3$), ist das Ergebnis der `isElement()` Methode unbestimmt.

Eingabeparameter:

- `dValue`: Der Wert, für den zu prüfen ist, ob er im Definitionsbereich liegt.

Rückgabe: Wenn der übergebene Wert `dValue` im Definitionsbereich liegt, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.8.2 round

Syntax: `doubleFib round( const doubleFib dValue ) const`

Diese Methode rundet den übergebenen Wert `dValue` auf einen Wert, der im (skalierten) Definitionsbereich liegt und den minimalen Abstand zum übergebenen Wert `dValue` hat. Gibt es mehrere Werte mit minimalen Abstand zum Wert `dValue`, wird der kleinste genommen.

Eingabeparameter:

- `dValue`: Der Wert, der auf einen Wert im Definitionsbereich zu runden ist.

Rückgabe: Ein Wert der im (skalierten) Definitionsbereich liegt und dem übergebenen Wert `lValue` möglichst nahe kommt.

51.8.3 getMaximum

Syntax: `doubleFib getMaximum() const`

Diese Methode liefert den größten Wert des (skalierten) Definitionsbereichs.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der größte Wert des Definitionsbereichs.

51.8.4 getMinimum

Syntax: `doubleFib getMinimum() const`

Diese Methode liefert den kleinsten Wert des (skalierten) Definitionsbereichs.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der kleinste Wert des Definitionsbereichs.

51.8.5 getNull

Syntax: `doubleFib getNull() const`

Diese Methode liefert den Nullwert des (skalierten) Definitionsbereichs.

Der Nullwert ist der Wert der entsteht, wenn die Zahl 0 auf einen Wert im Definitionsbereich gerundet wird.

Der Nullwert wird beispielsweise verwendet, wenn eine Eingabevariable nicht belegt wurde.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Nullwert des Definitionsbereichs.

51.8.6 cDomainIntegerBasis

Die Klasse `cDomainIntegerBasis` ist die Basisklasse der Definitionsbereiche für skalierte ganze Zahlen. Von dieser Klasse `cDomainIntegerBasis` werden alle Definitionsbereiche für skalierte ganze Zahlen abgeleitet. Von der Klasse `cDomainIntegerBasis` selbst kann jedoch keine Instanz erzeugt werden.

Zahlen aus diesem Definitionsbereich sind Zahlen, die mit einem Skalierungsfaktor S skaliert werden. DEr Skalierungsfaktor muß immer größer als 0 sein. Standardwert für den Skalierungsfaktor S ist 1, wodurch die Werte des Definitionsbereichs nicht skaliert werden. Die Skalierung dient dazu, um (ganze) Zahlen besser auf die SI-Einheiten Mappen zu können. Er kann ignoriert werden, wenn z. B. das Anzeigegerät zu klein oder viel zu groß ist, für eine vollständige Anzeige aller Werte des Definitionsbereichs. Beispielsweise wenn der Definitionsbereich für die horizontale Dimension von 0 m bis 10 m (Meter) geht, aber der Monitor nur 30 cm Breit ist.

Für die von `cDomainIntegerBasis` abgeleiteten Klassen gibt zwei Gruppen von Methoden, eine, welche die Werte ohne Skalierung zurückgibt, und eine, welche die Werte skaliert zurückgibt. Die Funktionen, welche auf nicht skalierten Werten arbeiten, haben "Unscaled" in ihrem Namen. Sind im Definitionsbereich nur Ganzzahlen (z. B. bei Unterbereichen) ist es besser die "Unscaled" Methoden zu verwenden. Im allgemeinen sind Werte der Methoden bzw. ihre Rückgabe auf dem skalierten Definitionsbereich nicht kompatibel zu ihren "Unscaled" Methoden. Gibt beispielsweise die `isElement()` Methode für einen Wert `true` zurück, ist die Rückgabe von `isUnscaledElement()` nicht unbedingt auch `true` , selbst wenn der Skalierungsfaktor 1 ist.

Die skalierte Ganzzahlen sind von Gleitkommazahlen zu unterscheiden. Bei skalierte Ganzzahlen haben die einzelnen Werte immer einen Abstand der ein vielfaches des Skalierungsfaktors S ist, dies gilt nicht für Gleitkommazahlen. Skalierte Ganzzahlen sind in gewisser Weise Ganzzahlen mit einer anderen Einheit, welche sich aus der ursprünglichen SI-Einheit durch Multiplikation mit dem Skalierungsfaktor S ergibt.

getScalingFactor

Syntax: `doubleFib getScalingFactor() const`

Diese Methode gibt den Skalierungsfaktor des Definitionsbereichs zurück.

Eingabeparameter: keine

Rückgabe: Den Skalierungsfaktor des Definitionsbereichs.

scale

Syntax: `doubleFib scale( const longFib lValue ) const`

Diese Methode skaliert den übergebenen Wert `lValue` auf einen Wert der im skalierten Definitionsbereich liegt und den minimalen Abstand zum skalierten Wert ($lValue * S$) des übergebenen Wert `lValue` hat. Gibt es mehrere skalierte Werte mit minimalen Abstand zum skalierten Wert von `lValue` , wird der kleinste genommen.

Beispiele für ein Definitionsbereich mit den natürlichen unskalierten Zahlen von 0 bis 4 und den Skalierungsfaktor S von 5:

- $scale(-2) = 0$
- $scale(0) = 0$
- $scale(1) = 5$
- $scale(2) = 10$
- $scale(4) = 20$
- $scale(10) = 20$

Eingabeparameter:

- `lValue`: Der Wert, welcher zu skalieren ist.

Rückgabe: Ein Wert der im skalierten Definitionsbereich liegt und dem skalierten Wert des übergebenen Werts `lValue` möglichst nahe kommt.

isUnscaledElement

Syntax: `bool isUnscaledElement( const longFib lValue ) const`

Diese Methode prüft, ob der übergebene Wert `lValue` im unskalierten Definitionsbereich liegt.

Eingabeparameter:

- `lValue`: Der Wert, für den zu prüfen ist, ob er im unskalierten Definitionsbereich liegt.

Rückgabe: Wenn der übergebene Wert `lValue` im unskalierten Definitionsbereich liegt, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

roundUnscaled

Syntax: `longFib roundUnscaled( const longFib lValue ) const`

Diese Methode rundet den übergebenen Wert `lValue` auf einen Wert der im unskalierten Definitionsbereich liegt und den minimalen Abstand zum übergebenen Wert `lValue` hat. Gibt es mehrere Werte mit minimalen Abstand zum Wert `lValue` wird der kleinste genommen.

Eingabeparameter:

- `lValue`: Der Wert, welcher auf einen Wert im unskalierten Definitionsbereich zu runden ist.

Rückgabe: Ein Wert der im unskalierten Definitionsbereich liegt und dem übergebenen Wert `lValue` möglichst nahe kommt.

getMaximumUnscaled

Syntax: `longFib getMaximumUnscaled() const`

Diese Methode liefert den größten Wert des unskalierten Definitionsbereichs.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der größte Wert des unskalierten Definitionsbereichs.

getMinimumUnscaled

Syntax: `longFib getMinimumUnscaled() const`

Diese Methode liefert den kleinsten Wert des unskalierten Definitionsbereichs.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der kleinste Wert des unskalierten Definitionsbereichs.

getNullUnscaled

Syntax: `longFib getNullUnscaled() const`

Diese Methode liefert den Nullwert des unskalierten Definitionsbereichs.

Der Nullwert ist der Wert, der entsteht, wenn die Zahl 0 auf einen Wert im unskalierten Definitionsbereich gerundet wird.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Nullwert des unskalierten Definitionsbereichs.

cDomainNaturalNumberBit Dieser Definitionsbereich hat als unskalierte Grundmenge natürliche Zahlen aus dem Bereich $0 \ldots (2^B - 1)$, wobei B die Anzahl der Bits pro Zahl ist.

cDomainNaturalNumberBit

Syntax: `cDomainNaturalNumberBit( unsignedIntFib iBits )`

Der Konstruktor des Definitionsbereichs für natürliche Zahlen mit `iBits` Bits. Er erstellt einen Definitionsbereich für natürliche Zahlen.

Der Definitionsbereich, welcher durch diesen Konstruktor erzeugt wird, wird nicht skaliert, bzw. enthält nur natürliche Zahlen aus dem Bereich $0 \ldots (2^{iBits} - 1)$.

Eingabeparameter:

- `iBits`: Die Anzahl der Bits, welche für die Zahlen des Definitionsbereichs verwendet werden.

Rückgabe: keine

cDomainNaturalNumberBit

Syntax: `cDomainNaturalNumberBit( unsignedIntFib iBits, doubleFib dScalingFactor )`

Der Konstruktor des Definitionsbereichs für skalierte natürliche Zahlen mit `iBits` Bits, er erstellt einen Definitionsbereich für skalierte natürliche Zahlen.

Die Werte des Definitionsbereichs sind: $0*dScalingFactor, 1*dScalingFactor, 2*dScalingFactor, \ldots, (2^{iBits-1})*dScalingFactor$

Eingabeparameter:

- `iBits`: Die Anzahl der Bits, die für die Zahlen des Definitionsbereichs verwendet werden.
- `dScalingFactor`: Der Skalierungsfaktor, welcher für die Zahlen des Definitionsbereichs verwendet wird.

Rückgabe: keine

cDomainIntegerBit Dieser Definitionsbereich hat als unskalierte Grundmenge Ganzenzahlen aus dem Bereich $-(2^{B-1}), \ldots, (2^{B-1}-1)$, wobei B die Anzahl der Bits pro Zahl ist.

cDomainIntegerBit

Syntax: `cDomainIntegerBit( unsignedIntFib iBits )`

Der Konstruktor des Definitionsbereichs für Ganzzahlen mit `iBits` Bits, er erstellt einen Definitionsbereich für Ganzenzahlen.

Der Definitionsbereich wird nicht skaliert, bzw. enthält nur natürliche Zahlen aus dem Bereich $-(2^{iBits-1}) \ldots (2^{iBits-1}-1)$.

Eingabeparameter:

- `iBits`: Die Anzahl der Bits, welche für die Zahlen des Definitionsbereichs verwendet werden.

Rückgabe: keine

cDomainIntegerBit

Syntax: `cDomainIntegerBit( unsignedIntFib iBits, doubleFib dScalingFactor )`

Der Konstruktor des Definitionsbereichs für skalierte Ganzenzahlen mit `iBits` Bits. Er erstellt einen Definitionsbereich für Ganzzahlen, welche mit dem Skalierungsfaktor `dScalingFactor` skaliert werden.

Die Werte des Definitionsbereichs sind: $-(2^{iBits-1}) * dScalingFactor, \ldots, (2^{iBits-1} - 1) * dScalingFactor$

Eingabeparameter:

- `iBits`: Die Anzahl der Bits, welche für die Zahlen des Definitionsbereichs verwendet werden.
- `dScalingFactor`: Der Skalierungsfaktor, der für die Zahlen des Definitionsbereichs verwendet wird.

Rückgabe: keine

cDomainNaturalNumber Dieser Definitionsbereich hat als unskalierte Grundmenge natürliche Zahlen aus dem Bereich $0, \ldots, X$, wobei X die größte natürliche Zahl des Bereichs ist.

cDomainNaturalNumber

Syntax: `cDomainNaturalNumber( unsignedLongFib lMaxNumber )`

Der Konstruktor des Definitionsbereichs für natürliche Zahlen, er erstellt einen Definitionsbereich für natürliche Zahlen.

Der Definitionsbereich, welcher durch den Konstruktor erzeugt wird, wird nicht skaliert, bzw. enthält nur natürliche Zahlen aus dem Bereich $0, \ldots, lMaxNumber$.

Eingabeparameter:

- `lMaxNumber`: Die größte Zahl im Definitionsbereich.

Rückgabe: keine

cDomainNaturalNumber

Syntax: `cDomainNaturalNumber( unsignedLongFib lMaxNumber, doubleFib dScalingFactor )`

Der Konstruktor des Definitionsbereichs für skalierte natürliche Zahlen, er erstellt einen Definitionsbereich für skalierte natürliche Zahlen.

Die Werte des Definitionsbereichs sind: $0 * dScalingFactor$, $1 * dScalingFactor$, $2 * dScalingFactor, \ldots, lMaxNumber * dScalingFactor$

Eingabeparameter:

- `lMaxNumber`: Die größte Zahl im Definitionsbereich.
- `dScalingFactor`: Der Skalierungsfaktor, der für die Zahlen des Definitionsbereichs verwendet wird.

Rückgabe: keine

cDomainInteger Dieser Definitionsbereich hat als unskalierte Grundmenge Ganzzahlen aus dem Bereich $X \ldots Y$, wobei X die kleinste Ganzzahl und Y die größte Ganzzahl des Bereichs ist.

cDomainInteger

Syntax: `cDomainInteger( longFib lMinNumber, longFib lMaxNumber )`

Der Konstruktor des Definitionsbereichs für Ganzzahlen, er erstellt einen Definitionsbereich für Ganzzahlen.

Der Definitionsbereich, welcher erzeugt wird, wird nicht skaliert, bzw. enthält nur Ganzzahlen aus dem Bereich $lMinNumber, \ldots, lMaxNumber$.

Eingabeparameter:

- `lMinNumber`: Die kleinste Zahl im unskalierten Definitionsbereich.
- `lMaxNumber`: Die größte Zahl im unskalierten Definitionsbereich.

Rückgabe: keine

cDomainInteger

Syntax: `cDomainInteger( longFib lMinNumber, longFib lMaxNumber, doubleFib dScalingFactor )`

Der Konstruktor des Definitionsbereichs für skalierte Ganzzahlen, er erstellt einen Definitionsbereich für skalierte Ganzzahlen.

Die Werte des Definitionsbereichs sind: $lMinNumber * dScalingFactor$, $(lMinNumber + 1) * dScalingFactor, \ldots, lMaxNumber * dScalingFactor$

Eingabeparameter:

- `lMinNumber`: Die kleinste Zahl im Definitionsbereich.
- `lMaxNumber`: Die größte Zahl im Definitionsbereich.
- `dScalingFactor`: Der Skalierungsfaktor, der für die Zahlen des Definitionsbereichs verwendet wird.

Rückgabe: keine

cDomainIntegerValues Die unskalierte Grundmenge dieses Definitionsbereichs sind Ganzzahlen aus einer vorgegeben Menge. Die Menge darf dabei nicht leer sein.

cDomainIntegerValues

Syntax: `cDomainIntegerValues( const list<longFib> &liUnscaledValues )`

Der Konstruktor des Definitionsbereichs für Ganzzahlen, er erstellt einen Definitionsbereich für Ganzzahlen.

Der Definitionsbereich, der erzeugt wird, wird nicht skaliert, bzw. enthält nur Ganzzahlen aus der übergebenen Liste `liUnscaledValues`.

Wenn die übergebene Liste `liUnscaledValues` leer ist, wird 0 zu der Menge der Ganzzahlen des Definitionsbereichs hinzugefügt.

Eingabeparameter:

- `liUnscaledValues`: Die Liste mit den unskalierten Werten des Definitionsbereichs.

Rückgabe: keine

cDomainIntegerValues

Syntax: `cDomainIntegerValues( const list<longFib> & liUnscaledValues, doubleFib dScalingFactor)`

Der Konstruktor des Definitionsbereichs für skalierte Ganzzahlen, er erstellt einen Definitionsbereich für skalierte Ganzzahlen.

Der Definitionsbereich, der durch den Konstruktor erzeugt wird, wird skaliert, bzw. enthält nur Ganzzahlen aus der übergebenen Liste `liUnscaledValues` skaliert mit dem Faktor `dScalingFactor`.

Wenn die übergebene Liste `liUnscaledValues` leer ist, wird 0 zu der Menge der Ganzzahlen des Definitionsbereichs hinzugefügt.

Eingabeparameter:

- `liUnscaledValues`: Die Liste mit den unskalierten Werten des Definitionsbereichs.
- `dScalingFactor`: Der Skalierungsfaktor, der für die Zahlen des Definitionsbereichs verwendet wird.

Rückgabe: keine

addUnscaledValue

Syntax: `bool addUnscaledValue( longFib lUnscaledValue )`

Diese Methode fügt einen unskalierten Wert `lUnscaledValue` zur Menge der unskalierten Werte hinzu.

Die Methode gibt `false` zurück, wenn vor dem Methodenaufruf der Wert schon in der Menge der unskalierten Werte war.

Eingabeparameter:

- `lUnscaledValue`: Der Wert, der zur Menge der unskalierten Werte hinzugefügt werden soll.

Rückgabe: Wenn der übergebene Wert `lUnscaledValue` zum unskalierten Definitionsbereich hinzugefügt wurde, wird `true` (=wahr) zurückgegeben, sonst wird `false` (=falsch) zurückgegeben.

deleteUnscaledValue

Syntax: `bool deleteUnscaledValue( longFib lUnscaledValue)`

Diese Methode löscht einen unskalierten Wert `lUnscaledValue` aus der Menge der unskalierten Werte.

Die Methode gibt `false` zurück, wenn der Wert `lUnscaledValue` nicht in Menge der unskalierten Werte existiert.

Es kann niemals der einzige Wert in der Menge der unskalierten Werte gelöscht werden. Wird versucht den einzigen Wert, den die Menge der unskalierten Werte enthält, zu löschen, wird `false` (=falsch) zurückgegeben und keine Änderung vorgenommen.

Eingabeparameter:

- `lUnscaledValue`: Der Wert, der aus der Menge der unskalierten Werte gelöscht werden soll.

Rückgabe: Wenn der übergebene Wert `lUnscaledValue` aus dem unskalierten Definitionsbereich gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

addValue

Syntax: `bool addValue( doubleFib dValue )`

Diese Methode fügt einen Wert `dValue` zum Definitionsbereich hinzu.

Die Methode gibt `false` zurück, wenn vor dem Methodenaufruf der Wert schon im Definitionsbereich war oder der Wert keine Ganzzahl mal Skalierungsfaktor entspricht.

Eingabeparameter:

- `dValue`: Der Wert, der zum Definitionsbereich hinzugefügt werden soll.

Rückgabe: Wenn der übergebene Wert `dValue` zum Definitionsbereich hinzugefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

deleteValue

Syntax: `bool deleteValue( doubleFib dValue )`

Diese Methode löscht einen Wert `dValue` aus dem Definitionsbereich.

Die Methode gibt `false` zurück, wenn der Wert `dValue` nicht im Definitionsbereich existiert.

Es kann niemals der einzige Wert in der Menge des Definitionsbereichs gelöscht werden. Wird versucht den einzigen Wert, den die Menge des Definitionsbereichs enthält, zu löschen, wird `false` (=falsch) zurückgegeben und keine Änderung vorgenommen.

Eingabeparameter:

- `dValue`: Der Wert, der aus dem Definitionsbereich gelöscht werden soll.

Rückgabe: Wenn der übergebene Wert `dValue` aus dem Definitionsbereich gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.8.7 cDomainRationalBasis

Die Klasse `cDomainRationalBasis` ist die Basisklasse alle Klassen, welche Definitionsbereiche von Gleitkommazahlen darstellen. Von dieser Basisklasse `cDomainRationalBasis` werden alle Definitionsbereiche für Gleitkommazahlen abgeleitet. Von `cDomainRationalBasis` kann jedoch keine Instanz erzeugt werden.

Eine Gleitkommazahl besteht aus zwei Ganzzahlfeldern, eines, das Erste, für den Exponent E und eines, das Zweite, für die Mantisse M. Die Gleitkommazahl Z ergibt sich dann zu $Z = M * 2^E$.

cDomainRational Dieser Definitionsbereich D_G enthält Gleitkommazahlen der Form $Z = M * 2^E$, wobei die Mantisse M aus einem Definitionsbereich D_M für die Mantisse und der Exponent E aus einem Definitionsbereich D_E für Exponenten kommt. $D_G = \{M * 2^E | (M \in D_M) \wedge (E \in D_E)\}$ Beide Definitionsbereich D_M und D_E kommen aus dem Definitionsbereich der (unskalierten) Ganzenzahlen.

cDomainRational

Syntax: `cDomainRational(`
```
 const cDomainIntegerBasis &dfMantissa ,
 const cDomainIntegerBasis &dfExponent )
```

Der Konstruktor des Definitionsbereichs für Gleitkommazahlen, er erstellt einen Definitionsbereich für Gleitkommazahlen.

Definitionsbereich:$D_G = \{M*2^E | (M \in dfMantissa) \wedge (E \in dfExponent)\}$

Die dem Konstruktor übergebenen Definitionsbereiche von Ganzahlen für Mantisse `dfMantissa` und Exponenten `dfExponent` sollten unskaliert sein. Sollten eine oder beide dennoch skaliert sein, so wird diese Skalierung für den Gleitkommazahlendefinitionsbereich aufgehoben bzw. auf 1 gesetzt.

Eingabeparameter:

- `dfMantissa`: Der (unskalierter) Ganzzahldefinitionsbereich aus dem die Mantisse kommt.
- `dfExponent`: Der (unskalierter) Ganzzahldefinitionsbereich aus dem der Exponent kommt.

Rückgabe: keine

cDomainRationalValues Dieser Definitionsbereich gehören Gleitkommazahlen aus einer vorgegeben Menge an. Die Menge des Definitionsbereichs darf nicht leer sein.

cDomainRationalValues

Syntax: `cDomainRationalValues( const list<doubleFib> &liValues )`

Der Konstruktor des Definitionsbereichs für Gleitkommazahlen, er erstellt einen Definitionsbereich für Gleitkommazahlen.

Der Definitionsbereich, welcher durch diesen Konstruktor erzeugt wird, enthält nur Gleitkommazahlen aus der übergebenen Liste `liValues`.

Ist die übergebenen Liste `liValues` leer, so wird der Wert 0.0 zum Definitionsbereich hinzugefügt.

Eingabeparameter:

- `liValues`: Die Liste mit den Werten des Definitionsbereichs.

Rückgabe: keine

addValue

Syntax: `bool addValue( doubleFib dValue )`

Diese Methode fügt einen Wert `dValue` zum Definitionsbereich hinzu.

Die Methode gibt `false` zurück, wenn vor dem Methodenaufruf der Wert schon im Definitionsbereich war.

Eingabeparameter:

- `dValue`: Der Wert, der zum Definitionsbereich hinzugefügt werden soll.

Rückgabe: Wenn der übergebene Wert `dValue` zum Definitionsbereich hinzugefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

deleteValue

Syntax: `bool deleteValue( doubleFib dValue )`

Diese Methode löscht einen Wert `dValue` aus dem Definitionsbereich.

Die Methode gibt `false` zurück, wenn der Wert `dValue` nicht im Definitionsbereich existiert.

Enthält die Menge des Definitionsbereichs nur noch einen Wert, kann diese niemals gelöscht werden. Wird versucht den einzigen Wert in der Menge des Definitionsbereichs zu löschen, so wird `false` (=falsch) zurückgegeben und keine Änderung vorgenommen.

Eingabeparameter:

- `dValue`: Der Wert, der aus dem Definitionsbereich gelöscht werden soll.

Rückgabe: Wenn der übergebene Wert `dValue` aus dem Definitionsbereich gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.9 Vektordefinitionsbereich cDomainVectorBasis

Von diesem Definitionsbereich werden alle Definitionsbereiche für Vektoren abgeleitet.

Vektoren bestehen immer aus einer Anzahl von Elementen.

Der Aufbau und die Bedeutung von Definitionsbereichen wird in Abschnitt 12.14.2 auf Seite 54 beschrieben.

51.9.1 isElement

Syntax: `bool isElement(const cFibVector &fibVector )const`

Diese Methode prüft, ob der übergebene Vektor im Definitionsbereich liegt. Dabei werden Variablen im Vektor ignoriert bzw. nicht berücksichtigt. Denn Vektorelemente, die Variablen sind, können durch die Variablenbelegung immer auf einen gültigen Wert gesetzt werden.

Eingabeparameter:

- `fibVector`: Eine Referenz auf den Vektor für den zu prüfen ist, ob er im Definitionsbereich liegt.

Rückgabe: Wenn der übergebene Vektor `fibVector` im Definitionsbereich liegt, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.9.2 round

Syntax: `cFibVector * round( cFibVector & fibVector ) const`

Diese Methode rundet den übergebenen Vektor `fibVector` auf einen Vektor der im Definitionsbereich liegt und den minimalen Abstand zum übergebenen Vektor `fibVector` hat.

Der Abstand zwischen zwei Vektoren ist die Summe der Abstände ihrer Elemente. Haben mehrere Vektoren einen minimalen Abstand zu dem Eingabevektor `fibVector`, so wird von diesen der Vektor zurückgegeben, dessen ersten n Elemente einen minimalen Abstand zum Eingabevektor `fibVector` hat, wobei n, von der Anzahl der Elemente im Eingabevektor ausgehend, solange um 1 verringert wird, bis ein Vektor gefunden wird oder 1 erreicht wird. Sollte auch dabei mehrere Vektoren übrig bleiben, wird der Vektor zurückgegeben, dessen k'tes Element kleiner ist als das k'tes Element der anderen in betracht kommenden Vektoren, wobei k von 1 auf die Anzahl der Elemente hochgezählt wird.

Sollte der Eingabevektor `fibVector` mehr Elemente enthalten, als die Vektoren des Definitionsbereichs, werden nur die ersten E Elemente des Eingabevektors `fibVector` betrachtet, wobei E die Anzahl der Elemente der Vektoren im Definitionsbereich ist.

Sollte der Eingabevektor `fibVector` weniger Elemente enthalten, als die Vektoren des Definitionsbereichs, werden die fehlenden Elemente des Eingabevektor `fibVector` mit 0 aufgefüllt.

Variablen werden beim Runden ignoriert. Ist ein Vektorelement vor dem Runden eine Variable, so ist das Vektorelement auch nach dem Runden die gleiche Variable.

Der gerundete Vektor hat den gleichen Typ (z. B. Positionsvektor oder Unterbereichsvektor) wie der Eingabevektor.

Eingabeparameter:

- `fibVector`: Eine Referenz auf den Vektor, der auf einen Vektor im Definitionsbereich zu Runden ist.

Rückgabe: Ein Zeiger auf einen Vektor der im Definitionsbereich liegt und dem übergebenen Vektor `fibVector` möglichst nahe kommt.

51.9.3 getNumberOfElements

Syntax: `unsignedIntFib getNumberOfElements() const`

Diese Methode liefert die Anzahl der Elemenete der Vektoren im Definitionsbereich zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Anzahl der Elemenete der Vektoren im Definitionsbereich.

51.9.4 getElement

Syntax: `cDomain * getElementDomain( unsignedIntFib iNumberOfElement )`

Diese Methode gibt den Definitionsbereich des `iNumberOfElement`'ten Elements des Vektordefinitionsbereichs zurück.

Eingabeparameter:

- `iNumberOfElement`: Die Nummer des Elements (die Zählung beginnt bei 1) dessen Definitionsbereich zurückgegeben werden soll.

Rückgabe: Den Definitionsbereich des `iNumberOfElement`'ten Elements des Vektordefinitionsbereichs.

51.9.5 cDomainVector

Die durch `cDomainVector` definierten Definitionsbereiche stehen für Vektoren.

cDomainVector

Syntax: `cDomainVector( const vector<cDomain> & vElementDomains )`

Der Konstruktor des Vektordefinitionsbereichs, er erstellt einen Vektordefinitionsbereich.

Eingabeparameter:

- `vElementDomains`: Der Vektor mit den Definitionsbereichen für die Elemente des Vektors. Der n'te Vektordefinitionsbereich wird dabei dem n'ten Element zugeordnet (die Zählung beginnt bei 1).

Rückgabe: keine

51.9.6 cDomainVectorElements

Die durch `cDomainVectorElements` definierten Definitionsbereiche stehen für Vektoren. Dabei ist nur eine vorgegebene Anzahl von festgelegten Vektoren erlaubt. Werte/Vektoren aus einem `cDomainVectorElements` müssen also aus einer Menge von festgelegten Vektoren kommen.

In einem `cDomainVectorElements`-Objekt wird für die einzelnen möglichen/ erlaubten Vektoren intern eine Liste gehalten. In dieser Liste sind die Vektoren nach Größe geordnet. Der kleinste Vektor kommt zuerst. Doppelte /gleiche Vektoren sind nicht erlaubt. Ein Vektor V_1 ist größer als ein zweiter Vektor V_2, wenn das n'te Element im Vektor V_1 größer als im Vektor V_2 ist und alle Elemente vor dem n'te in beiden Vektoren gleich sind. Leer Vektorlisten sind nicht erlaubt, jede Vektorliste muß mindestens einen Vektor enthalten.

Alle Vektoren in der Grundmenge bzw. der gehaltenen Liste sollten vom gleichen Typ sein. Ist der Typ nicht gleich, hat dies allerdings auch keine weiteren Auswirkungen, da der Vektortyp für den Definitionsbereich nicht relevant ist.

cDomainVectorElements

Syntax: `cDomainVectorElements(`
```
 const vector<cDomain * > & domains,
 const vector< cFibVector * > & vecFibVectors )
```

Der Konstruktor des Vektordefinitionsbereichs, er erstellt einen Vektordefinitionsbereich.

Wenn die Liste `vecFibVectors` leer ist, wird der Nullvektor zur Vektorliste hinzugefügt. Der Nullvektor ist der Vektor, dessen Elemente auf die Nullwerte ihres jeweiligen Elementedefinitionsbereichs gesetzt sind.

Eingabeparameter:

- `domains`: Der Vektor mit den Zeigern auf die Definitionsbereiche für die Elemente des Vektors. Der n'te Vektordefinitionsbereich wird dabei dem n'ten Element zugeordnet (die Zählung beginnt bei 1). Diese Elementedefinitionsbereich werden für den Vektordefinitionsbereich kopiert.

- `vecFibVectors`: Die Vektoren, welche der Definitionsbereich erlauben soll. Die übergebenen Vektoren in der Liste werden für den Vektordefinitionsbereich kopiert.

Rückgabe: keine

getNumberOfVectors

Syntax: `unsignedLongFib getNumberOfVectors()`

Diese Methode gibt die Anzahl der Vektoren in der Liste der möglichen Vektoren zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Anzahl der Vektoren in der Liste der möglichen Vektoren.

getVectorNumber

Syntax: `unsignedLongFib getVectorNumber( const cFibVector & fibVector )`

Diese Methode gibt die Nummer des Vektors `fibVector` in der Liste der möglichen Vektoren zurück.

Existiert der Vektor `fibVector` nicht in der Liste, wird der Vektor auf einen Vektor im Definitionsbereich gerundet und dessen Nummer zurückgegeben.

Eingabeparameter:

- `fibVector`: Eine Referenz auf den Vektor, für den die Nummer zurückgegeben werden soll.

Rückgabe: Die Nummer des Vektors `fibVector` in der Liste der möglichen Vektoren.

getVectorForNumber

Syntax: `cFibVector * getVectorForNumber( unsignedLongFib lVectorNumber )`

Diese Methode gibt den Vektor mit der Nummer `lVectorNumber` in der Liste der möglichen Vektoren zurück.

Existieren keine `lVectorNumber` Vektoren in der Liste der möglichen Vektoren, wird der letzte Vektor aus der Liste zurückgegeben.

Ist `lVectorNumber` gleich 0 wird der erste Vektor aus der Liste zurückgegeben.

Eingabeparameter:

- `lVectorNumber`: Die Nummer des Vektors, der zurückgegeben werden soll.

Rückgabe: Einen Zeiger auf den Vektor mit der Nummer `lVectorNumber` in der Liste der möglichen Vektoren.

addVector

Syntax: `bool addVector( const cFibVector & fibVector )`

Diese Methode fügt den übergebenen Vektor `fibVector` zur Liste der möglichen Vektoren hinzu. Wobei seine Elemente auf Zahlen aus dem Definitionsbereich des jeweiligen Elements gerundet werden.

Der Vektor kann nicht eingefügt werden, wenn ein Vektor mit identischen Elementen in die Liste der möglichen Vektoren schon existiert oder er nicht die gleiche Anzahl Elemente wie die Vektoren im Definitionsbereich hat. Dann wird `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `fibVector`: Eine Referenz auf den Vektor, der in die Liste der möglichen Vektoren eingefügt werden soll. Dieser Vektor wird vor dem Einfügen in den Vektordefinitionsbereich kopiert.

Rückgabe: Wenn der übergebene Vektor `fibVector` im Definitionsbereich eingefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

deleteVector

Syntax: `bool deleteVector( const cFibVector & fibVector )`

Diese Methode löscht den übergebenen Vektor `fibVector` aus der Liste der möglichen Vektoren. Die Elemente werden dabei zuerst auf Zahlen aus den Definitionsbereichen des jeweiligen Elements gerundet, und danach wird ein Vektor mit genau diesen Elementen gesucht. Wird ein solcher Vektor gefunden, wird er gelöscht und `true` (=wahr) zurückgegeben, sonst wird `false` (=falsch) zurückgegeben.

Wenn die Vektorliste des Definitionsbereichs nur noch ein Element enthält, kann dies niemals gelöscht werden.

Eingabeparameter:

- `fibVector`: Eine Referenz auf den Vektor, der aus der Liste der möglichen Vektoren gelöscht werden soll.

Rückgabe: Wenn der übergebene Vektor `fibVector` aus dem Definitionsbereich gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

51.9.7 cDomainVectorOpenEnd

Die durch `cDomainVectorOpenEnd` definierten Definitionsbereiche stehen für Vektoren mit variabler Länge (vectorOpenEnd($E, D_1, \ldots, D_E$)).

cDomainVectorOpenEnd

Syntax: `cDomainVectorOpenEnd( const vector<cDomain> & vElementDomains )`

Der Konstruktor des Vektordefinitionsbereichs mit variabler Länge.

Der erzeugte Definitionsbereich ist für Vektor mit E (=`vElementDomains.size()`) oder mehr Elementen, wobei das i'te Element, für i kleiner E, den Definitionsbereich D_i hat und sonst den Definitionsbereich D_E. Dieser Definitionsbereich dient für Elemente, die Vektoren unterschiedlicher Größe enthalten können. Die Anzahl der Elemente des Vektors sollte durch das enthaltende Fib-Element festgelegt werden.

Eingabeparameter:

- `vElementDomains`: Der Vektor mit den Definitionsbereichen für die Elemente des Vektors. Der i'te Vektordefinitionsbereich wird dabei dem i'ten Element zugeordnet (die Zählung beginnt bei 1). Der letzte (E'te) Definitionsbereich in diesem Vektor `vElementDomains` gilt auch für alle Vektorelemente mit einem größeren Index ($E \leq i$) .

Rückgabe: keine

51.10 Verweise auf ander Definitionsbereiche cDomainReference

Der `cDomainReference` Definitionsbereiche verweist auf einen Definitionsbereich eines anderen Elements / Typs (siehe Abschnitt 51.6 auf Seite 269). Als Definitionsbereich wird der Definitionsbereich des Elements mit dem gegebenen Type genommen.

Von den Methoden `getDomain()` und `getDomainForElement()` der `cDomains` Klasse wird jeweils direkt der Definitionsbereich zurückgegeben auf den verwiesen wird. Auf diese Weise bedarf `cDomainReference` keine Sonderbehandlung bei der Benutzung. Die Klasse `cDomainReference` ist somit eine interne Referenz der Klasse `cDomains` die außerhalb von `cDomains` wie die Zieldomain, auf die verwiesen wird, behandelt werden kann.

cDomainReference

Syntax: `cDomainReference( const cTypeElement * pOriginalType, const vector<unsignedIntFib> vecElementDomains=vector<cDomain>() )`

Der Konstruktor des Verweisdefinitionsbereichs.

Der erzeugte Definitionsbereich verweist auf den Definitionsbereich des Typs `pOriginalType`, wobei mit den Zahlen in `vecElementDomains` der Unterdefinitionsbereich spezifiziert wird.

Beispiel: `cDomainReference( new cTypeFibMatrix(), [ 4, 2 ] ):` Der Definitionsbereich verweist auf den zweiten Unterdefinitionsbereich des vierten Unterdefinitionsbereich des Matrix-Elements.

Eingabeparameter:

- `pOriginalType`: Ein Zeiger auf den Type des Definitionsbereichs auf den verwiesen wird.
- `vecElementDomains`: Der Vektor mit der Angabe des Unterdefinitionsbereichen auf den verwiesen wird. Die Zahlen sind dabei jeweils die Nummer (die Zählung beginnt bei 1) des auszuwählenden Unterdefinitionsbereichs. Dabei haben Unterdefinitionsbereiche eines höheren Levels jeweils einen niedrigeren Index.

Rückgabe: keine

getTypeElement

Syntax: `const cTypeElement * getTypeElement() const;`

Diese Methode liefert einen Zeiger auf den Typ des Definitionsbereich auf den verwiesen wird.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeiger auf den Typ des Definitionsbereichs.

getElement

Syntax: `vector<unsignedIntFib> getElement() const;`

Diese Methode liefert einen Vektor mit den Indexnummern (die Zählung beginnt bei 1) auf Unterdefinitionsbereich auf den verwiesen wird. Dabei haben Unterdefinitionsbereiche eines höheren Levels jeweils einen niedrigeren Index.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Vektor mit den Indexnummern (die Zählung beginnt bei 1) auf Unterdefinitionsbereich.

51.11 Standarddefinitionsbereiche cDomainDefault

Mit dem `cDomainDefault` Definitionsbereich kann ein Standarddefinitionsbereich für ein Element festgelegt werden. Der in `cDomainDefault` enthaltende Definitionsbereich wird nur verwendet, wenn zuvor kein anderer Definitionsbereich definiert war.

Von den Methoden `getDomain()` und `getDomainForElement()` der `cDomains` Klasse wird jeweils direkt der Definitionsbereich zurückgegeben der gültig ist. Auf diese Weise bedarf `cDomainDefault` keine Sonderbehandlung bei der Benutzung.

cDomainDefault

Syntax: `cDomainDefault( const cDomain * pDefaultDomain )`

Der Konstruktor des Standarddefinitionsbereichs.

Der erzeugte Definitionsbereich enthält den Definitionsbereich `pDefaultDomain`. Wenn also kein anderer Definitionsbereich existiert, wird der `pDefaultDomain` Definitionsbereich verwendet.

Eingabeparameter:

- `pDefaultDomain`: Ein Zeiger auf den Standarddefinitionsbereiche des Definitionsbereichs. Dieser wird verwendet, wenn kein anderer Definitionsbereich für das Element vorher existierte.

Rückgabe: keine

getDefaultDomain

Syntax: `const cDomain * getTypeElement() const;`

Diese Methode liefert einen Zeiger auf den Standarddefinitionsbereich zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeiger auf den Standarddefinitionsbereich.

52 Fib-Vektoren cFibVector

Die Fib-Vektoren werden durch eine Klasse bereitgestellt, welche sich den Vektorentyp merkt.

Über den zu einem Vektor gehörenden Definitionsbereich im entsprechenden root-Elemente kann der Definitionsbereich des Vektors ermittelt werden. Nur Werte aus dem Definitionsbereich können von Elementen (Werte oder Variablenwerte) des Vektors angenommen werden, sollen Werte außerhalb des Definitionsbereichs Verwendung finden, ist der Definitionsbereich entsprechend zu erweitern.

Die Klasse `cFibVector` dient als Basisklasse aller Fib-Vektoren. Von der Klasse `cFibVector` können keine Instanzen erzeugt werden. In Abbildung 18 ist ein Klassendiagramm der Fib-Vektoren zu sehen.

Die Zählung der Vektorelemente beginnt bei 1.

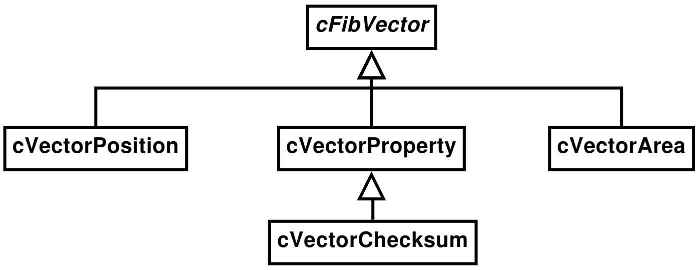


Abbildung 18: Klassengraph der Fib-Vektoren

52.1 Schnittstellenbeschreibung

52.1.1 cFibVector

Syntax: `protected cFibVector(`
`unsignedIntFib iNumberOfVectorElements=1,`
`const cFibElement * pDefiningFibElement )`

Konstruktor eines Vektors, er erzeugt einen Vektors. Da ein allgemeiner Vektor `cFibVector` nicht erzeugt werden kann und soll, ist der Konstruktor außerhalb der von `cFibVector` abgeleiteten Klassen nicht aufrufbar.

Alle Vektorelemente werden mit 0 initialisiert.

Eingabeparameter:

- `iNumberOfVectorElements`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 1 .
- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.1.2 getElementType

Syntax: `cTypeElement getElementType() const`

Diese Methode gibt den Typ des Vektors zurück.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf den Typ des Vektors.

52.1.3 getNumberOfElements

Syntax: `unsignedIntFib getNumberOfElements() const`

Diese Methode gibt die Anzahl der Elemente des Vektors zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der Elemente des Vektors.

52.1.4 isVariable

Syntax: `bool isVariable( unsignedIntFib iNumberElement ) const`

Diese Methode gibt zurück, ob das `iNumberElement`'te Element des Vektors eine Variable ist.

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, für das zurückgegeben werden soll, ob es eine Variable ist.

Rückgabe: Wenn das `iNumberElement`'te Element des Vektors eine Variable ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.1.5 getValue

Syntax: `doubleFib getValue( unsignedIntFib iNumberElement ) const`

Diese Methode gibt den Wert des `iNumberElement`'ten Elements zurück.

Dabei wird bei einer Variable ihre Belegung ausgegeben. Ist die Variable nicht belegt, wird der Nullwert des Definitionsbereiches zum Vektorelement zurückgegeben.

Der zurückgegebene Wert liegt immer im Definitionsbereich des Vektors (auch bezüglich der anderen Vektorelemente).

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, für das der Wert zurückgegeben werden soll.

Rückgabe: Den Wert des `iNumberElement`'ten Elements.

52.1.6 setValue

Syntax: `bool setValue( unsignedIntFib iNumberElement, doubleFib dValue )`

Diese Methode setzt das `iNumberElement`'ten Vektorelement auf den Wert `dValue`.

Der Wert `dValue` muss im Definitionsbereich des Vektorelements liegen, um gesetzt werden zu können. Wenn der Wert `dValue` nicht im Definitionsbereich des Vektorelements liegt, wird das Vektorelement nicht verändert und `false` (=falsch) zurückgegeben. (Dann sollte eventuell der Definitionsbereich angepasst werden.) Werte die mit der `round`-Methode des zum Vektorelement gehörenden Definitionsbereich erzeugt wurden (bzw. bei denen die `isElement`-Methode `true` zurückgibt), können immer gesetzt werden.

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, für das der Wert `dValue` gesetzt werden soll.
- `dValue`: Der Wert `dValue`, welcher gesetzt werden soll.

Rückgabe: Wenn das `iNumberElement`'te Element des Vektors auf den Wert `dValue` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.1.7 getVariable

Syntax: `cFibVariable *getVariable( unsignedIntFib iNumberElement )`

Diese Methode liefert einen Zeiger auf die Variable des `iNumberElement`'ten Vektorelements zurück oder `NULL` (Nullpointer), wenn das `iNumber`'te Vektorelement keine Variable ist.

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, von dem die Variable zurückgegeben werden soll.

Rückgabe: Einen Zeiger auf die Variable des `iNumberElement`'ten Vektorelements oder `NULL` (Nullpointer), wenn das `iNumberElement`'te Vektorelement keine Variable ist.

52.1.8 setVariable

Syntax: `bool setVariable( unsignedIntFib iNumberElement, cFibVariable *pVariable )`

Diese Methode setzt das `iNumberElement`'te Vektorelement auf die Variable `pVariable`.

Gibt es kein `iNumberElement`'tes Vektorelement wird `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, für das die Variable `pVariable` gesetzt werden soll.
- `pVariable`: Der Zeiger auf die zu setzende Variable.

Rückgabe: Wenn das `iNumberElement`'te Element des Vektors auf die Variable `pVariable` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.1.9 getDomain

Syntax: `cDomainVectorBasis * getDomain()`

Diese Methode gibt eine Referenz auf den Definitionsbereich des Vektors zurück oder den Nullpointer `NULL`, wenn keine Definitionsbereich für den Vektor definiert ist. Wenn der Nullpointer `NULL` zurückgegeben wurde ist die Standarddomain für den vektor gültig.

Eingabeparameter: keine

Rückgabe: Es wird eine Referenz auf den Definitionsbereich des Vektors zurückgegeben oder den Nullpointer `NULL`, wenn keine Definitionsbereich für den Vektor definiert ist.

52.1.10 getDomain

Syntax: `cDomainSingle *getDomain( unsignedIntFib iNumberElement )`

Diese Methode gibt einen Zeiger auf den Definitionsbereich des entsprechenden `iNumberElement`'ten Vektorelements zurück oder den Nullpointer `NULL`, wenn keine Definitionsbereich für den Vektor definiert ist. Wenn der Nullpointer `NULL` zurückgegeben wurde ist die Standarddomain für den vektor gültig.

Eingabeparameter:

- `iNumberElement`: Die Nummer des Vektorelements, für das der Definitionsbereich zurückgegeben werden soll.

Rückgabe: Es wird einen Zeiger auf den Definitionsbereich des entsprechenden `iNumberElement`'ten Vektorelements oder den Nullpointer `NULL` (wenn keine Definitionsbereich für den Vektor definiert ist) zurückgegeben.

52.1.11 getStandardDomain

Syntax: `cDomainVectorBasis *getStandardDomain() const`

Diese Methode gibt eine Referenz auf den Standarddefinitionsbereich des Vektors zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf den Standarddefinitionsbereich des Vektors.

52.1.12 getDefiningFibElement

Syntax: `cFibElement *getDefiningFibElement() const`

Diese Methode gibt eine Referenz auf das Fib-Element zurück, welches den Vektor definiert bzw. ihn enthält.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf das Fib-Element, welches den Vektor definiert bzw. enthält.

52.1.13 setDefiningFibElement

Syntax:
```
void setDefiningFibElement(
 cFibElement * fibElement=NULL );
```

Diese Methode setzt das Fib-Element, welches den Vektor definiert bzw. enthält.

Eingabeparameter:

- `fibElement`: Ein Zeiger auf das Fib-Element, welches den Vektors enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen das noch kein Fib-Element, welches den Vektors enthält, bekannt ist.

Rückgabe: keine.

52.2 cVectorPosition

Elternklasse: cFibVector

Standarddefinitionsbereich: $vector(2, integerB(16), integerB(16))$

Dieser Vektor stellt einen Positionsvektor dar. Er kann nur in Punktelementen verwendet werden.

52.2.1 cVectorPosition

Syntax: `cVectorPosition(`
```
 unsignedIntFib iNumberOfDimensions=2,
 cFibElement * pDefiningPointElement=NULL )
```

Der Konstruktor des Positionsvektors, er erstellt einen Positionsvektor.
Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `iNumberOfDimensions`: Die Anzahl der Dimensionen bzw. Elemente, welche der Positionsvektor enthalten soll. Standardwert ist 2 .
- `pDefiningPointElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.3 cVectorProperty

Elternklasse: cFibVector

Standarddefinitionsbereich: $vector(0)$

Mit diesem Vektor werden die Werte von Eigenschaften gespeichert. Dabei wird der Typ der Eigenschaft, für den der Vektor steht, beim Erzeugen des Objekts festgelegt.

52.3.1 cVectorProperty

Syntax: `cVectorProperty( unsignedIntFib uiProperyType, cFibElement * pDefiningFibElement=NULL )`

Der Konstruktor des Eigenschaftsvektors, er erstellt einen Eigenschaftsvektor.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `uiProperyType`: Der Typ der Eigenschaft. Dieses ist ein Wert, so wie er in Tabelle 2 auf Seite 30 angegeben wird.
- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.3.2 cVectorProperty

Syntax: `cVectorProperty( unsignedIntFib uiProperyType, unsignedIntFib iNumberOfVectorElements=1, cFibElement * pDefiningFibElement=NULL )`

Der Konstruktor des Eigenschaftsvektors, er erstellt einen Eigenschaftsvektor.

Dieser Konstruktor sollte verwendet werden, wenn für die Eigenschaft vom Typ `iProperyType` kein korrekter Standarddefinitionsbereich existiert. (z. B. wie bei den "Product Properties" siehe Tabelle 2 auf Seite 30)

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `uiProperyType`: Der Typ der Eigenschaft. Dieses ist ein Wert, so wie er in Tabelle 2 auf Seite 30 angegeben wird.
- `iNumberOfVectorElements`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 1 .
- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, das noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.3.3 getPropertyType

Syntax: `unsignedIntFib getPropertyType()`

Diese Methode gibt den Typ der Eigenschaft zurück, so wie er in Tabelle 2 auf Seite 30 angegeben wird.

Eingabeparameter: keine

Rückgabe: Den Typ der Eigenschaft.

52.3.4 cVectorChecksum

Elternklasse: cVectorProperty

Standarddefinitionsbereich:
$vector(3, integerB(4), integerB(8), integerB(8))$

Kompatible zu: Alle Vektoren mit drei Elementen, deren Definitionsbereich aus den natürlichen Zahlen kommen.

Dieser Vektor dient der Typsicherheit des Checksummenvektors im root-Element. Er hat keine eigenen Methoden.

cVectorChecksum

Syntax: `cVectorChecksum(`
`cFibElement * pDefiningFibElement=NULL )`

Der Konstruktor des Eigenschaftsvektors für die Checksummeneigenschaft, er erstellt einen Eigenschaftsvektor für die Checksummeneigenschaft.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.4 cVectorArea

Elternklasse: cFibVector

Standarddefinitionsbereich: $vector(2, integerB(32), integerB(32))$ (Standarddefinitionsbereich von `cTypeArea`)

Dieser Vektor hält die Werte für die untere und obere Grenze eines Unterbereichs (inklusive Grenzen). Beide Werte für die Grenzen sind Ganzzahlen.

52.4.1 cVectorArea

Syntax 1: `cVectorArea( longFib lLowerBound,`
`longFib lUpperBound,`
`cFibElement * pDefiningFibElement=NULL )`
Syntax 2: `cVectorArea( longFib lLowerBound,`
`cFibVariable & vUpperBound,`
`cFibElement * pDefiningFibElement=NULL ) )`
Syntax 3: `cVectorArea( cFibVariable & vLowerBound,`
`longFib lUpperBound,`
`cFibElement * pDefiningFibElement=NULL ) )`
Syntax 4: `cVectorArea( cFibVariable & vLowerBound,`
`cFibVariable & vUpperBound,`
`cFibElement * pDefiningFibElement=NULL ) )`

Der Konstruktoren des Bereichsvektors, sie erstellen jeweils einen Bereichsvektor.

Eingabeparameter:

- `lLowerBound`: Die unter Grenze des Bereichs, den der Vektor repräsentiert.
- `vLowerBound`: Eine Referenz auf die Variable, welche die unter Grenze des Bereichs bestimmt, den der Vektor repräsentiert.
- `lUpperBound`: Die ober Grenze des Bereichs, den der Vektor repräsentiert.
- `vUpperBound`: Eine Referenz auf die Variable, welche die obere Grenze des Bereichs bestimmt, den der Vektor repräsentiert.
- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.4.2 getLowerBound

Syntax: `longFib getLowerBound() const`

Diese Methode gibt die untere Grenze des Bereichs zurück.

Dabei wird bei einer Variable ihre Belegung ausgegeben. Ist die Variable nicht belegt, wird der Nullwert des Definitionsbereiches zum Bereichstyp (bzw. des Vektorelements) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Die untere Grenze des Bereichs.

52.4.3 getUpperBound

Syntax: `longFib getUpperBound() const`

Diese Methode gibt die obere Grenze des Bereichs zurück.

Dabei wird bei einer Variable ihre Belegung ausgegeben. Ist die Variable nicht belegt, wird der Nullwert des Definitionsbereiches zum Bereichstyp (bzw. des Vektorelements) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Die ober Grenze des Bereichs.

52.4.4 getAreaValues

Syntax: `list<longFib> getAreaValues() const`

Diese Methode gibt alle Werte die im Bereich liegen zurück.

Dies sind alle Ganzzahlen zwischen der oberen und unteren Grenze des Bereichs (inclusive der Grenzen, wenn diese jeweils Ganzzahlen ist).

Dabei wird bei einer Variable als Grenze ihre Belegung berücksichtigt. Ist die Variable nicht belegt, wird der Nullwert des Definitionsbereiches zum Bereichstyps (bzw. des Vektorelements) berücksichtigt.

Eingabeparameter: keine

Rückgabe: Zurückgegeben werden alle Werte, die im Bereich liegen.

52.4.5 setLowerBoundValue

Syntax: `bool setLowerBoundValue( longFib lValue )`

Diese Methode setzt die untere Grenze des Bereichs auf den Wert `lValue`.

Der Wert `lValue` muss im Definitionsbereich des Vektorelements bzw. des Bereichstyps liegen, um gesetzt werden zu können. Wenn der Wert `lValue` nicht im Definitionsbereich des Vektorelements liegt, wird das Vektorelement nicht verändert und `false` (=falsch) zurückgegeben. (Dann sollte eventuell der Definitionsbereich entsprechend angepasst werden.) Werte die mit der `round`-Methode des zum Vektorelement gehörenden Definitionsbereich erzeugt wurden (bzw. bei denen die `isElement`-Methode `true` zurückgibt), können immer gesetzt werden.

Eingabeparameter:

- `lValue`: Der für die untere Grenze des Bereichs zu setzende Wert.

Rückgabe: Wenn die untere Grenze des Bereichs auf den Wert `lValue` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.4.6 setUpperBoundValue

Syntax: `bool setUpperBoundValue( longFib lValue )`

Diese Methode setzt die obere Grenze des Bereichs auf den Wert `lValue`.

Der Wert `lValue` muss im Definitionsbereich des Vektorelements bzw. des Bereichstyps liegen, um gesetzt werden zu können. Wenn der Wert `lValue` nicht im Definitionsbereich des Vektorelements liegt, wird das Vektorelement nicht verändert und `false` (=falsch) zurückgegeben. (Dann sollte eventuell der Definitionsbereich entsprechend angepasst werden.) Werte die mit der `round`-Methode des zum Vektorelement gehörenden Definitionsbereich erzeugt wurden (bzw. bei denen die `isElement`-Methode `true` zurückgibt), können immer gesetzt werden.

Eingabeparameter:

- `lValue`: Der für die obere Grenze des Bereichs zu setzende Wert.

Rückgabe: Wenn die obere Grenze des Bereichs auf den Wert `lValue` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.4.7 setLowerBoundVariable

Syntax: `bool setLowerBoundVariable( cFibVariable *pVariable )`

Diese Methode setzt die untere Grenze des Bereichs auf die Variable `pVariable`.

Eingabeparameter:

- `pVariable`: Der Zeiger auf die zu setzende Variable.

Rückgabe: Wenn unter Grenze des Bereichs auf die Variable `pVariable` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.4.8 setUpperBoundVariable

Syntax: `bool setUpperBoundVariable( cFibVariable *pVariable )`

Diese Methode setzt die obere Grenze des Bereichs auf die Variable `pVariable`.

Eingabeparameter:

- `pVariable`: Der Zeiger auf für die zu setzende Variable.

Rückgabe: Wenn obere Grenze des Bereichs auf die Variable `pVariable` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

52.5 cVectorExtObject

Elternklasse: cFibVector

Standarddefinitionsbereich: $vectorOpenEnd(integerB(8))$ (Standarddefinitionsbereich von `cTypeExtObjectInput`)

Dieser Vektor stellt einen Satz bzw. Vektor von Zahlen dar. Er kann nur im externen Objekt Element verwendet werden (siehe Abschnitt 47 auf Seite 231 und Abschnitt 12.9 auf Seite 43) .

52.5.1 cVectorExtObject

Syntax: `cVectorExtObject(`
` unsignedIntFib iNumberOfElements=0,`
` cFibElement * pDefiningMatrixElement=NULL )`

Der Konstruktor des Vektors für das externen Objekt Element.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `iNumberOfDimensions`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 0 .
- `pDefiningPointElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.6 cVectorExtSubobject

Elternklasse: cFibVector

Standarddefinitionsbereich: $vector(0)$ (Standarddefinitionsbereich von `cTypeExtSubobject`)

Dieser Vektor stellt einen Satz bzw. Vektor von Zahlen dar. Er kann nur im externen Unterobjekt Element verwendet werden (siehe Abschnitt 48 auf Seite 237 und Abschnitt 12.10 auf Seite 44) .

52.6.1 cVectorExtSubobject

Syntax: `cVectorExtSubobject(`
` unsignedIntFib iNumberOfElements=0,`
` cFibElement * pDefiningMatrixElement=NULL )`

Der Konstruktor des Vektors für das externen Unterobjekt Element.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `iNumberOfDimensions`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 0 .

- `pDefiningPointElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.7 cVectorFibSet

Elternklasse: cFibVector

Standarddefinitionsbereich: $vectorOpenEnd(1, integerB(32))$ (dritter Unterdefinitionsbereich vom Standarddefinitionsbereich von `cTypeFibSet`)

Dieser Vektor stellt einen Satz bzw. Vektor von Zahlen dar. Er kann nur im set-Element verwendet werden (siehe Abschnitt 49 auf Seite 238 und Abschnitt 12.12 auf Seite 48) .

52.7.1 cVectorFibSet

Syntax: `cVectorFibSet(`
```
 unsignedIntFib iNumberOfElements=2,
 cFibElement * pDefiningSetElement=NULL )
```

Der Konstruktor des Vektors für das set-Element.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `iNumberOfElements`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 2 .
- `pDefiningPointElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

52.8 cVectorFibMatrix

Elternklasse: cFibVector

Standarddefinitionsbereich: $vectorOpenEnd(1, integerB(32))$ (vierter Unterdefinitionsbereich vom Standarddefinitionsbereich von `cTypeFibMatrix`)

Dieser Vektor stellt einen Satz bzw. Vektor von Zahlen dar. Er kann nur im Matrixelement verwendet werden (siehe Abschnitt 50 auf Seite 242 und Abschnitt 12.13 auf Seite 49) .

52.8.1 cVectorFibMatrix

Syntax: `cVectorFibMatrix(`
```
 unsignedIntFib iNumberOfElements=2,
 cFibElement * pDefiningMatrixElement=NULL )
```

Der Konstruktor des Vektors für das Matrixelement.

Initial werden alle Elemente auf den Nullwerte des entsprechenden Definitionsbereichs gesetzt.

Eingabeparameter:

- `iNumberOfElements`: Die Anzahl der Elemente, welche der Vektor enthalten soll. Standardwert ist 2 .
- `pDefiningPointElement`: Ein Zeiger auf das Fib-Element, welches den Vektor enthält. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass noch kein Fib-Element bekannt ist, welches den Vektor enthält.

Rückgabe: keine

53 cFibVariable

Die Klasse `cFibVariable` stellt Variablen für Fib-Objekte dar. Jede Variable muss von einem Fib-Element definiert werden. Nur Fib-Elemente können deshalb Instanzen von `cFibVariable` erzeugen.

Wenn Variablen in einem Fib-Element verwendet werden, sollte die Variable immer über dem Fib-Element definiert werden.

Jede Fib-Variable hält sich eine Liste mit Fib-Elementen in den die Variable verwendet wird, sowie eine Referenz auf das Fib-Element in dem die Fib-Variable definiert wird.

53.1 Schnittstellenbeschreibung

53.1.1 cFibVariable

Syntax: `cFibVariable(`
```
 const cFibElement * pDefiningFibElement )
```

Konstruktor einer Variable, er erzeugt eine Variable.

Eingabeparameter:

- `pDefiningFibElement`: Ein Zeiger auf das Fib-Element, das diese Variable definiert.

Rückgabe: keine

53.1.2 getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Wert der Variable zurück oder 0 wenn, der Variablenwert undefiniert ist.

Eingabeparameter: keine

Rückgabe: Der Wert der Variable oder 0, wenn der Variablenwert undefiniert ist.

53.1.3 getIntegerValue

Syntax: `longFib getIntegerValue() const`

Diese Methode gibt den Wert der Variable als Ganzzahl zurück oder 0 wenn, der Variablenwert undefiniert ist.

Ist der Inhalt der Variable eine Gleitkommazahl oder skalierte Ganzzahl, wird er auf eine Ganzzahl gerundet und ausgegeben.

Dies ist beispielsweise nützlich, wenn der Vektor, in dem die Variable verwendet wird, nur Ganzzahlen enthalten darf (wie z. B. bei Unterbereichsvektoren `cVectorArea`).

Eingabeparameter: keine

Rückgabe: Der Wert der Variable als Ganzzahl oder 0, wenn der Variablenwert undefiniert ist.

53.1.4 setValue

Syntax: `void setValue( doubleFib dValue )`

Diese Methode setzt den Wert der Variable auf den übergebenen Wert `dValue`.

Diese Methode ist beispielsweise bei Eingabevariablen des obersten root-Elements nützlich.

Eingabeparameter:

- `dValue`: Der Wert, auf den die Variable zu setzen ist.

Rückgabe: keine

53.1.5 setIntegerValue

Syntax: `void setIntegerValue( longFib lValue )`

Diese Methode setzt den Wert der Variable auf den übergebenen Ganzzahlwert `lValue`.

Diese Methode ist beispielsweise bei Eingabevariablen des obersten root-Elements nützlich.

Durch Eingabe eines Ganzzahlwertes könnte unter anderem die Sprache oder der Untertitel ausgewählt werden. Gleitkommazahlen könnten in solchen Situationen zu Rundungsfehlern führen.

Eingabeparameter:

- `lValue`: Der Ganzzahlwert, auf den die Variable zu setzen ist.

Rückgabe: keine

53.1.6 getDefiningElement

Syntax: `cFibElement * getDefiningElement() const`

Diese Methode gibt ein Zeiger auf das Fib-Element zurück, in dem die Variable definiert wird.

Eingabeparameter: keine

Rückgabe: Ein Zeiger auf das Fib-Element, in dem die Variable definiert wird.

53.1.7 getNumberOfUsingElements

Syntax: `unsignedIntFib getNumberOfUsingElements()`

Diese Methode gibt die Anzahl der Fib-Elemente zurück, in denen die Variable benutzt wird.

Wenn dieser Wert beispielsweise 0 ist, kann das Fib-Element, das die Variable definiert, gelöscht werden.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Anzahl der Fib-Element in denen die Variable benutzt wird.

53.1.8 registerUsingElement

Syntax: `protected void registerUsingElement( cFibElement * usingElement )`

Diese Methode registriert das Fib-Element `usingElement` als ein Fib-Element, dass die Variable benutzt.

Eingabeparameter:

- `usingElement`: Ein Zeiger auf das Fib-Element, das diese Variable benutzt.

Rückgabe: keine

53.1.9 unregisterUsingElement

Syntax: `protected void unregisterUsingElement( cFibElement * usingElement )`

Diese Methode deregistriert das Fib-Element `usingElement` als ein Fib-Element, dass die Variable benutzt. Damit wird der Variable angezeigt, dass das Fib-Element `usingElement` die Variable nicht mehr benötigt.

Eingabeparameter:

- `usingElement`: Ein Zeiger auf das Fib-Element, das diese Variable nicht mehr benutzt.

Rückgabe: keine

53.1.10 getUsingElements

Syntax: `set< cFibElement* > getUsingElements()`

Diese Methode gibt eine Menge mit den Zeigern auf die Fib-Elemente zurück, in denen die Variable benutzt wird.

Dadurch kann die Variable beispielsweise leichter ersetzt werden, als wenn der gesamte Fib-Unterbaum, über den die Variable definierenden Fib-Element, nach der Variable durchsucht werden müsste.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Menge mit den Zeigern auf die Fib-Elemente, in denen die Variable benutzt wird.

54 Fib-Unterfunktionen cUnderFunction

Die Klasse `cUnderFunction` ist die Basisklasse aller Unterfunktionen. Aus Unterfunktionen werden Funktionen (mit der Struktur von Bäumen) zusammengestellt. (siehe Abschnitt 12.7 auf Seite35)

Die Klasse `cUnderFunction` dient als Basisklasse aller Unterfunktionen. Von der Klasse `cUnderFunction` können keine Instanzen erzeugt werden.

Bei der Auswertung von Unterfunktionen ist für den erzeugten Wert der Definitionsbereich für Unterfunktionenen nicht relevant. Wenn also beispielsweise der Definitionsbereich für Unterfunktionenen nur Ganzzahlen enthält und eine Unterfunktionen zu $1,5$ ausgewertet wird, wird dieser Wert $1,5$ sowohl von anderen Unterfunktionen direkt (ohne Rundung) verwendet werden, als auch die definierte Variable eines Funktionselements den Wert $1,5$ annehemen. Der Definitionsbereich für Unterfunktionenen kommt nur zur Anwendung, wenn die Unterfunktion einen Wert enthält (also wenn die Unterfunktion `cFunctionValue` ist), oder wenn eine Unterfunktion fehlt und desshalb anstatt ihr der Nullwert des Definitionsbereichs eingesetzt werden soll.

In Abbildung 19 ist ein Klassendiagramm der Unterfunktionen zu sehen.

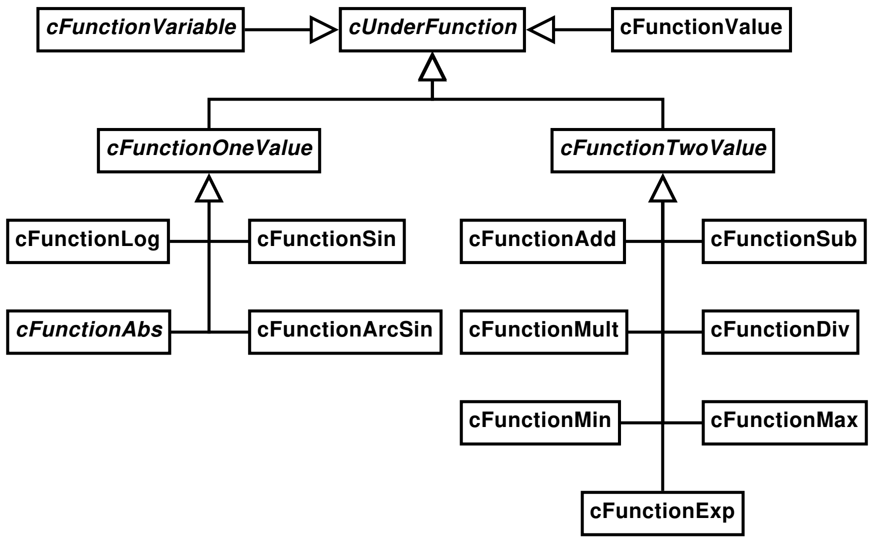


Abbildung 19: Klassengraph der Unterfunktionen

Es gibt drei Arten von Unterfunktionen:

- Nullstellige (keine enthaltenden Unterfunktionen)
- Einstellige (einen Eingabeparameter bzw. enthaltende Unterfunktionen)
- Zweistellige (zwei Eingabeparameter bzw. enthaltende Unterfunktionen)

Die Klassen für die nullstelligen Funktionen sind:

- `cFunctionValue`: Für einen Wert bzw. eine Konstante.
- `cFunctionVariable`: Für eine Variable, die über dem Fib-Element, welches die Unterfunktion enthält, definiert wird.

Die Klassen, welche die einstelligen Funktionen implementieren, werden von der Basisklasse `cFunctionOneValue` abgeleitet.

Zu diesen einstelligen Funktionen gehören:

- `cFunctionLog`: Logarithmus
- `cFunctionSin`: Sinusfunktion
- `cFunctionArcsin`: Arkussinusfunktion
- `cFunctionAbs`: Absolutwert
- `cFunctionRound`: Runden

Die Klassen, welche die zweistelligen Funktionen implementieren, werden von der Basisklasse `cFunctionTwoValue` abgeleitet.

Zu diesen zweistelligen Funktionen gehören:

- `cFunctionAdd`: Addition
- `cFunctionSub`: Subtraktion
- `cFunctionMult`: Multiplikation
- `cFunctionDiv`: Division
- `cFunctionExp`: Exponent
- `cFunctionMin`: Minimum
- `cFunctionMax`: Maximum
- `cFunctionIf`: If-Funktion
- `cFunctionDelay`: Delay-Funktion

54.1 Schnittstellenbeschreibung

54.1.1 getValue

Syntax: `doubleFib getValue() const`

Diese Methode berechnet den Wert der Unterfunktion. Dabei werden in der Unterfunktion enthaltenden Unterfunktionen ausgewertet bzw. deren Wert berechnet.

Wenn eine Unterfunktion der Unterfunktion nicht existiert (dies wäre eine ungültige Unterfunktion), wird für sie der Nullwert des Definitionsbereichs für Unterfunktionen angenommen.

Eingabeparameter: keine

Rückgabe: Den berechneten Wert der Unterfunktion.

54.1.2 getType

Syntax: `intFib getType() const`

Diese Methode gibt eine Ganzzahl zurück, über die der Typ der Unterfunktion bestimmt werden kann. Für diese Typen werden auch Konstanten in der Klasse `cUnderFunction` definiert.

Mögliche Typen sind:

Typ-zahl	Unterfunktion	Konstante
00	`cFunctionValue`: Für einen Wert bzw. eine Konstante.	`FUNCTION_VALUE`
01	`cFunctionVariable`: Für eine Variable, die über dem Fib-Element, welches die Unterfunktion enthält, definiert wird.	`FUNCTION_VARIABLE`
10	`cFunctionLog`: Logarithmus	`FUNCTION_LOG`
11	`cFunctionSin`: Sinusfunktion	`FUNCTION_SIN`
12	`cFunctionAbs`: Absolutwert	`FUNCTION_ABS`
13	`cFunctionArcsin`: Arkussinusfunktion	`FUNCTION_ARCSIN`
14	`cFunctionRound`: Rundungsfunktion	`FUNCTION_ROUND`
20	`cFunctionAdd`: Addition	`FUNCTION_ADD`
21	`cFunctionSub`: Subtraktion	`FUNCTION_SUB`
22	`cFunctionMult`: Multiplikation	`FUNCTION_MULT`
23	`cFunctionDiv`: Division	`FUNCTION_DIV`
24	`cFunctionExp`: Exponent	`FUNCTION_EXP`
25	`cFunctionMin`: Minimum	`FUNCTION_MIN`
26	`cFunctionMax`: Maximum	`FUNCTION_MAX`
30	`cFunctionIf`: If-Funktion	`FUNCTION_IF`
31	`cFunctionMod`: Modulo-Funktion	`FUNCTION_MOD`

Eingabeparameter: keine

Rückgabe: Eine Ganzzahl, über die der Typ der Unterfunktion bestimmt werden kann.

54.1.3 isValid

Syntax: `bool isValid() const`

Diese Methode prüft, ob die Unterfunktion korrekt ist.

Damit eine Unterfunktion korrekt ist, müssen alle direkt oder indirekt enthaltenden Unterfunktionen und Bedingungen (cCondition siehe Abschnitt 55 auf Seite 349) korrekt sein. Weiterhin müssen alle konstanten Unterfunktionen `cFunctionValue` Werte im zugehörigem Definitionsbereich für Unterfunktionen "underfunction" enthalten. Außerdem müssen alle Unterfunktionen für Variablen `cFunctionVariable` ein gültige Variable enthalten, welche in einem Fib-Element über dem Fib-Element des Funktionsobjekts definiert ist. Eine korrekte Unterfunktion darf sich nicht selbst enthalten und keine direkt oder indirekt enthaltende Unterfunktion mehr als einmal enthalten. Das heißt, die direkt oder indirekt enthaltenden Unterfunktionen haben die Struktur eines (zyklenfreien) Baums.

Eingabeparameter: keine

Rückgabe: Wenn die Unterfunktion korrekt ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

54.1.4 getDomain

Syntax: `cDomain * getDomain()`

Diese Methode gibt eine Referenz auf den Definitionsbereich der Unterfunktion zurück.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf den Definitionsbereich der Unterfunktion .

54.2 cFunctionValue

Die Klasse `cFunctionValue` stellt eine konstante Unterfunktion bzw. einen Wert dar. Sie ist ein Blatt im Unterfunktionsbaum, da sie keine weiteren Unterfunktionen enthält. Ihr Wert (Rückgabe von `getValue()`) ist immer der Wert, der für sie angegeben wurde.

54.2.1 cFunctionValue

Syntax: `cFunctionValue( doubleFib dValue )`

Der Konstruktor der konstante Unterfunktion, er erstellt einen konstante Unterfunktion.

Wenn der übergebene Wert außerhalb des Definitionsbereichs für Unterfunktionen liegt, wird er auf einen Wert im Definitionsbereichs gerundet.

Eingabeparameter:

- `dValue`: Der Wert, den die konstante Unterfunktion zurückgeben bzw. repräsentieren soll.

Rückgabe: keine

54.2.2 getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Wert der konstanten Unterfunktion zurück.

Eingabeparameter: keine

Rückgabe: Der Wert der konstanten Unterfunktion.

54.2.3 setValue

Syntax: `bool setValue( doubleFib dValue )`

Der Methode setzt den Wert der konstanten Unterfunktion auf `dValue`.

Der Wert `dValue` muss im Definitionsbereich für Unterfunktion liegen, um gesetzt werden zu können. Wenn der Wert `dValue` nicht im Definitionsbereich für Unterfunktion liegt, wird die Unterfunktion nicht verändert und `false` (=falsch) zurückgegeben. (Dann sollte eventuell der Definitionsbereich angepasst werden.) Werte die mit der `round`-Methode des zum Unterfunktionen gehörenden Definitionsbereich erzeugt wurden (bzw. bei denen die `isElement`-Methode `true` zurückgibt), können immer gesetzt werden.

Eingabeparameter:

- `dValue`: Der Wert, den die konstante Unterfunktion zurückgeben bzw. repräsentieren soll.

Rückgabe: Wenn der Wert der Unterfunktion auf den Wert `dValue` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

54.3 cFunctionVariable

Die Klasse `cFunctionVariable` stellt eine Variable dar. Der Inhalt dieser Variable stellt den Wert (Rückgabe von `getValue()`) der Unterfunktion dar. Sie ist ein Blatt im Unterfunktionsbaum, da sie keine weiteren Unterfunktionen enthält.

54.3.1 cFunctionVariable

Syntax: `cFunctionVariable( cFibVariable *fibVariable )`

Der Konstruktor der Unterfunktion für Variablen, er erstellt einen Unterfunktion für Variablen.

Eingabeparameter:

- `fibVariable`: Einen Zeiger auf die Variable, welche die Unterfunktion repräsentieren soll. Der Inhalt dieser Variable stellt den Wert der Unterfunktion dar.

Rückgabe: keine

54.3.2 getVariable

Syntax: `cFibVariable * getVariable()`

Der Methode gibt einen Zeiger auf die Variable, welche die Unterfunktion darstellt, zurück.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die Variable, welche die Unterfunktion darstellt.

54.3.3 setVariable

Syntax: `bool setVariable( cFibVariable * pFibVariable )`

Der Methode setzt die Variable, welche die Unterfunktion darstellt, auf die übergebene Variable `pFibVariable`.

Diese Operation schlägt fehl und gibt `false` (=falsch) zurück, wenn die Variable `pFibVariable` ein Nullpointer `NULL` ist.

Eingabeparameter:

- `pFibVariable`: Einen Zeiger auf die Variable, welche die Unterfunktion darstellen soll.

Rückgabe: Wenn die Variable, welche die Unterfunktion darstellt, auf die Variable `pFibVariable` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

54.3.4 getValue

Syntax: `doubleFib getValue()`

Diese Methode gibt den Wert der Variable, welche die Unterfunktion darstellt, zurück.

Existiert keine Variable, welche die Unterfunktion darstellt, oder ist die Variable nicht mit einem Wert belegt, wird für sie der Nullwert des Definitionsbereichs für Unterfunktionen zurückgegeben.

Eingabeparameter: keine

Rückgabe: Den Wert der Variable, welche die Unterfunktion darstellt.

54.4 cFunctionOneValue

Die Klasse `cFunctionOneValue` ist die Basisklasse aller einstelligen Unterfunktionen. Von der Klasse `cFunctionOneValue` können keine Instanzen erzeugt werden.

Einstellig Unterfunktionen enthalten eine Unterfunktion. Beim Ermitteln des Wertes der Unterfunktion mit `getValue()` wird zuerst der Wert der enthaltenden Unterfunktion ermittelt (mit `getValue()`). Existiert keine enthaltenden Unterfunktion, wird der Nullwert des Definitionsbereichs für Unterfunktionen als Wert der enthaltenden Unterfunktion angenommen. Bei der Berechnung des Funktionswerts, wird also für fehlende Unterfunktionen der Nullwert eingesetzt.

Einstellige Unterfunktionen sind:

- `cFunctionLog`: Logarithmus
- `cFunctionSin`: Sinusfunktion
- `cFunctionArcsin`: Arcussinusfunktion
- `cFunctionAbs`: Absolutwert

Im Folgenden werden zuerst die Methoden für einstellige Unterfunktionen vorgestellt und dann die Klassen der möglichen einstellige Unterfunktionen.

54.4.1 getUnderFunction

Syntax: `cUnderFunction * getUnderFunction()`

Diese Methode gibt einen Zeiger auf die in der einstelligen Unterfunktion enthaltende Unterfunktion zurück.

Existiert keine enthaltende Unterfunktion, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die in der einstelligen Unterfunktion enthaltende Unterfunktion oder der Nullpointer `NULL` , wenn keine enthaltende Unterfunktion existiert.

54.4.2 setUnderFunction

Syntax: `void setUnderFunction(`
```
cUnderFunction * pUnderFunction,
bool bDeleteOld=true )
```

Diese Methode setzt die in der einstelligen Unterfunktion enthaltende Unterfunktion auf `pUnderFunction`. Dabei wird keine Kopie von `pUnderFunction` erstellt.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die einstelligen Unterfunktion enthalten soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte enthaltende Unterfunktion gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die alte enthaltende Unterfunktion (aus dem Arbeitsspeicher) gelöscht, sonst, wenn er `false` (=falsch) ist, nicht. Standardwert ist `true`, um die alte enthaltende Unterfunktion zu löschen.

Rückgabe: keine

54.4.3 cFunctionLog für Logarithmus

Die Klasse `cFunctionLog` realisiert den natürliche Logarithmus. Der natürliche Logarithmus ($\ln X$) ist zur Basis e zu bestimmen.

cFunctionLog

Syntax: `cFunctionLog( cUnderFunction * pUnderFunction )`

Der Konstruktor der Logarithmusunterfunktion, er erstellt eine Logarithmusunterfunktion.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die Logarithmusunterfunktion enthalten soll. Die Unterfunktion `pUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den natürliche Logarithmus des Wertes zurück, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

Eingabeparameter: keine

Rückgabe: Den natürliche Logarithmus des Wertes, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

54.4.4 cFunctionSin für Sinusfunktion

Die Klasse `cFunctionSin` realisiert die Sinusfunktion. Gerechnet wird dabei mit Bogenmaß.

cFunctionSin

Syntax: `cFunctionSin( cUnderFunction * pUnderFunction )`

Der Konstruktor der Sinusunterfunktion, er erstellt eine Sinusunterfunktion.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die Sinusunterfunktion enthalten soll. Die Unterfunktion `pUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Sinuswert des Wertes zurück, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird. Gerechnet wird dabei mit Bogenmaß.

Eingabeparameter: keine

Rückgabe: Den Sinuswert des Wertes, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

54.4.5 cFunctionArcsin für Arkussinus

Die Klasse `cFunctionArcsin` realisiert die Arkussinusfunktion.

cFunctionArcsin

Syntax: `cFunctionArcsin( cUnderFunction * pUnderFunction )`

Der Konstruktor der Arkussinusfunktion, er erstellt eine Arkussinusfunktion.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die Arkussinusfunktion enthalten soll. Die Unterfunktion `pUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Arkussinusfunktion des Wertes zurück, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird. Zurückgegben wird dabei ein Wert in Bogenmaß.

Eingabeparameter: keine

Rückgabe: Den Arkussinus des Wertes, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

54.4.6 cFunctionAbs für Absolutwert

Die Klasse `cFunctionAbs` realisiert die Absolutwertfunktion.

cFunctionAbs

Syntax: `cFunctionAbs( cUnderFunction * pUnderFunction )`

Der Konstruktor der Absolutwertunterfunktion, er erstellt eine Absolutwertunterfunktion.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die Absolutwertunterfunktion enthalten soll. Die Unterfunktion `pUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Absolutwert des Wertes zurück, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird. Für $x \geq 0$ ist der Absolutwert x, sonst bei $x < 0$ ist der Absolutwert $-x$.

Eingabeparameter: keine

Rückgabe: Den Absolutwert des Wertes, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

54.4.7 cFunctionRound für Rundung

Die Klasse `cFunctionRound` realisiert das Runden von zahlen.

cFunctionRound

Syntax: `cFunctionRound( cUnderFunction * pUnderFunction )`

Der Konstruktor der Rundungsunterfunktion, er erstellt eine Rundungsunterfunktion.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche die Rundungsunterfunktion enthalten soll. Die Unterfunktion `pUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den gerundeten Wert des Wertes zurück, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird. Beim Runden bleibt die Ziffer der Vorkommastelle unverändert, wenn die Ziffer der ersten Nachkommastelle 0 bis 4 ist, sonst (bei 5 bis 9) wird die Ziffer der Vorkommastelle bei positiven Zahlen um eins erhöht und bei negativen Zahlen um eins verringert.

Eingabeparameter: keine

Rückgabe: Den gerundeten Wert des Wertes, der durch die enthaltende Unterfunktion (mit `getValue()`) ermittelt wird.

54.5 cFunctionTwoValue

Die Klasse `cFunctionTwoValue` ist die Basisklasse aller zweistelligen Unterfunktionen. Von der Klasse `cFunctionTwoValue` können keine Instanzen erzeugt werden.

Zweistellig Unterfunktionen enthalten zwei Unterfunktionen. Beim Ermitteln des Wertes der Unterfunktion mit `getValue()` , werden zuerst die Werte der enthaltenden Unterfunktionen ermittelt (mit `getValue()`), wobei der Wert der ersten enthaltenden Unterfunktion zuerst ermittelt wird. Existiert eine enthaltenden Unterfunktion nicht, wird der Nullwert des Definitionsbereichs für Unterfunktionen als Wert dieser enthaltenden Unterfunktion angenommen. Bei der Berechnung des Funktionswerts, wird also für fehlende Unterfunktionen der Nullwert eingesetzt.

Zweistellige Unterfunktionen sind:

- `cFunctionAdd`: Addition
- `cFunctionSub`: Subtraktion
- `cFunctionMult`: Multiplikation
- `cFunctionDiv`: Division
- `cFunctionExp`: Exponent
- `cFunctionMin`: Minimum
- `cFunctionMax`: Maximum

Im Folgenden werden zuerst die Methoden für zweistellige Unterfunktionen vorgestellt und dann die Klassen der möglichen zweistellige Unterfunktionen.

54.5.1 getFirstUnderFunction

Syntax: `cUnderFunction * getFirstUnderFunction()`

Diese Methode gibt einen Zeiger auf die erste, in der zweistelligen Unterfunktion enthaltende, Unterfunktion zurück.

Existiert keine erste enthaltende Unterfunktion, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die erste in der zweistelligen Unterfunktion enthaltende Unterfunktion oder der Nullpointer `NULL` , wenn keine erste enthaltende Unterfunktion existiert.

54.5.2 setFirstUnderFunction

Syntax: `void setFirstUnderFunction( const cUnderFunction &underFunction, bool bDeleteOld=true )`

Diese Methode setzt die erste, in der zweistelligen Unterfunktion enthaltende, Unterfunktion auf `underFunction`. Dabei wird keine Kopie von der übergebenen `underFunction` erstellt.

Eingabeparameter:

- `underFunction`: Eine Referenz auf die Unterfunktion, welche die erste einstelligen Unterfunktion sein soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die erste alte enthaltende Unterfunktion gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die erste alte enthaltende Unterfunktion (aus dem Arbeitsspeicher) gelöscht, sonst, wenn der Wert `false` (=falsch) ist, nicht. Standardwert ist `true`, um die erste alte enthaltende Unterfunktion zu löschen.

Rückgabe: keine

54.5.3 getSecondUnderFunction

Syntax: `cUnderFunction * getSecondUnderFunction()`

Diese Methode gibt einen Zeiger auf die zweite in der zweistelligen Unterfunktion enthaltende Unterfunktion zurück.

Existiert keine zweite enthaltende Unterfunktion, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die zweite in der zweistelligen Unterfunktion enthaltende Unterfunktion oder der Nullpointer `NULL` , wenn keine zweite enthaltende Unterfunktion existiert.

54.5.4 setSecondUnderFunction

Syntax: `void setSecondUnderFunction( const cUnderFunction &underFunction, bool bDeleteOld=true )`

Diese Methode setzt die zweite, in der zweistelligen Unterfunktion enthaltende, Unterfunktion auf `underFunction`. Dabei wird keine Kopie von der übergebenen `underFunction` erstellt.

Eingabeparameter:

- `underFunction`: Eine Referenz auf die Unterfunktion, welche die zweite einstelligen Unterfunktion sein soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte zweite enthaltende Unterfunktion gelöscht werden soll. Wenn der Wert `true` (=wahr) ist wird die zweite alte enthaltende Unterfunktion (aus dem Arbeitsspeicher) gelöscht, sonst, wenn der Wert `false` (=falsch) ist, nicht. Standardwert ist `true`, um die zweite alte enthaltende Unterfunktion zu löschen.

Rückgabe: keine

54.5.5 cFunctionAdd für Addition

Die Klasse `cFunctionAdd` realisiert die Addition zweier Zahlen bzw. Unterfunktionswerte.

cFunctionAdd

Syntax: `cFunctionAdd(`
```
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Additionsunterfunktion, er erstellt eine Additionsunterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Additionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Additionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt die Summe der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Die Summe der Werte der enthaltenden Unterfunktionen.

54.5.6 cFunctionSub für Subtraktion

Die Klasse `cFunctionSub` realisiert die Subtraktion zweier Zahlen bzw. Unterfunktionswerte. Wobei der Wert der zweiten Unterfunktion von dem der ersten abgezogen wird.

cFunctionSub

Syntax: `cFunctionSub(`
```
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Subtraktionsunterfunktion, er erstellt eine Subtraktionsunterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Subtraktionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Subtraktionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue()`

Diese Methode gibt die Differenz der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Die Differenz der Werte der enthaltenden Unterfunktionen.

54.5.7 cFunctionMult für Multiplikation

Die Klasse `cFunctionMult` realisiert die Multiplikation zweier Zahlen bzw. Unterfunktionswerte.

cFunctionMult

Syntax: `cFunctionMult(`
```
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Multiplikationsunterfunktion, er erstellt eine Multiplikationsunterfunktion. Von diesem Konstruktor wird keine Kopie von den beiden übergebenen Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Multiplikationsunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Multiplikationsunterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt das Produkt der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Das Produkt der Werte der enthaltenden Unterfunktionen (diese Werte werden mit `getValue()` der Unterfunktionen ermittelt).

54.5.8 cFunctionDiv für Division

Die Klasse `cFunctionDiv` realisiert die Division zweier Zahlen bzw. Unterfunktionswerte. Wobei der Wert W_1 der ersten Unterfunktion durch den Wert W_2 der zweiten Unterfunktion dividiert wird (W_1/W_2). Die ersten Unterfunktion liefert also den Dividend und die zweite den Divisor des Quotienten.

cFunctionDiv

Syntax:
```
cFunctionDiv(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Divisionsunterfunktion, er erstellt eine Divisionsunterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Divisionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Divisionsunterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Wert des Quotienten der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Den Wert des Quotienten der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt).

54.5.9 cFunctionMod für Modulo

Die Klasse `cFunctionMod` realisiert den symmetrischen Modulo Operator zweier Zahlen bzw. Unterfunktionswerte. Der symmetrische Modulo Operator gibt den Rest beim ganzzahliger Division von W_1 und W_2 aus. ($mod(W_1, W_2) = W_1 - W_2 * int(W_1/W_2)$; wobei int das Abschneiden der Nachkommastellen bezeichnet) Die ersten Unterfunktion liefert also den Dividend und die zweite den Divisor des Quotienten.

cFunctionMod

Syntax:
```
cFunctionMod(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Modulounterfunktion, er erstellt eine Modulounterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Modulounterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Modulounterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt den Wert der symmetrische Modulo Operator der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück. ($mod(W_1, W_2) = W_1 - W_2 * int(W_1/W_2)$; wobei int das Abschneiden der Nachkommastellen bezeichnet)

Eingabeparameter: keine

Rückgabe: Den Wert der symmetrische Modulo Operator der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt).

54.5.10 cFunctionExp für Exponent

Die Klasse `cFunctionExp` realisiert die Exponentialfunktion zweier Zahlen bzw. Unterfunktionswerte. Wobei der Wert W_1 der ersten Unterfunktion hoch dem Wert W_2 der zweiten Unterfunktion genommen wird ($W_1^{W_2}$). Der Wert der ersten Unterfunktion ist also die Basis und der Wert der zweiten Unterfunktion der Exponent der Exponentialfunktion.

cFunctionExp

Syntax:
```
cFunctionExp(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Exponentialunterfunktion, er erstellt eine Exponentialunterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Exponentialunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die Exponentialunterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt das Ergebnis des Wertes der ersten enthaltenden Unterfunktionen hoch dem Wert der zweiten enthaltenden Unterfunktionen zurück.

Eingabeparameter: keine

Rückgabe: Das Ergebnis des Wertes der ersten enthaltenden Unterfunktionen hoch dem Wert der zweiten enthaltenden Unterfunktionen.

54.5.11 cFunctionMin für Minimum

Die Klasse `cFunctionMin` realisiert das Minimum zweier Zahlen bzw. Unterfunktionswerte. Es wird also der kleinste Wert zurückgegeben, der von den enthaltenden Unterfunktionen zurückgegeben wurde.

cFunctionMin

Syntax: `cFunctionMin(`
```
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Minimumsunterfunktion, er erstellt eine Minimumsunterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die Minimumsunterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.

- pSecondUnderFunction: Einen Zeiger auf die zweite Unterfunktion, welche die Minimumsunterfunktion enthalten soll. Die übergebene Unterfunktion pSecondUnderFunction wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt das Minimum der Werte der enthaltenden Unterfunktionen (mit getValue() dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Das Minimum der Werte der enthaltenden Unterfunktionen (diese Werte werden mit getValue() der Unterfunktionen ermittelt).

54.5.12 cFunctionMax für Maximum

Die Klasse cFunctionMax realisiert das Maximum zweier Zahlen bzw. Unterfunktionswerte. Es wird also der größte Wert zurückgegeben, der von den enthaltenden Unterfunktionen zurückgegeben wurde.

cFunctionMax

Syntax:
```
cFunctionMax(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der Maximumsunterfunktion, er erstellt eine Maximumsunterfunktion. Dabei wird keine Kopie von den Objekten pFirstUnderFunction oder pSecondUnderFunction erstellt.

Eingabeparameter:

- pFirstUnderFunction: Einen Zeiger auf die erste Unterfunktion, welche die Maximumsunterfunktion enthalten soll. Die übergebene Unterfunktion pFirstUnderFunction wird vor dem Einfügen nicht kopiert.
- pSecondUnderFunction: Einen Zeiger auf die zweite Unterfunktion, welche die Maximumsunterfunktion enthalten soll. Die übergebene Unterfunktion pSecondUnderFunction wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt das Maximum der Werte der enthaltenden Unterfunktionen (mit `getValue()` dieser ermittelt) zurück.

Eingabeparameter: keine

Rückgabe: Das Maximum der Werte der enthaltenden Unterfunktionen (diese werden mit `getValue()` ermittelt).

54.5.13 cFunctionIf für die if-Funktion

Die Klasse `cFunctionIf` realisiert die if-Funktion zweier Zahlen bzw. Unterfunktionswerte. Wenn die Bedingung wahr ist, wird der Wert der ersten Unterfunktion zurückgegeben, sonst der der zweiten.

cFunctionIf

Syntax: `cFunctionIf(`
```
cCondition * pCondition,
cUnderFunction * pFirstUnderFunction,
cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor der if-Unterfunktion, er erstellt eine if-Unterfunktion. Dabei wird keine Kopie von den Objekten `pFirstUnderFunction` oder `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pCondition`: Einen Zeiger auf die Bedingung der if-Unterfunktion. Die übergebene Bedingung `pCondition` wird vor dem Einfügen nicht kopiert.
- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche die if-Unterfunktion enthalten soll. Die übergebene Unterfunktion `pFirstUnderFunction` wird vor dem Einfügen nicht kopiert.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche die if-Unterfunktion enthalten soll. Die übergebene Unterfunktion `pSecondUnderFunction` wird vor dem Einfügen nicht kopiert.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Wenn die Bedingung wahr ist, wird der Wert der ersten Unterfunktion zurückgegeben (mit `getValue()` dieser ermittelt), sonst der der zweiten.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird wenn die Bedingung wahr ist, der Wert der ersten Unterfunktion (mit `getValue()` dieser ermittelt), sonst der der zweiten.

54.5.14 getCondition

Syntax: `cCondition * getCondition()`

Diese Methode gibt einen Zeiger auf die Bedingung der If-Unterfunktion zurück.

Eingabeparameter: keine

Rückgabe: Den Zeiger auf die Bedingung des If-Unterfunktion.

54.5.15 setCondition

Syntax: `void setCondition( const cCondition & condition)`

Diese Methode setzt die Bedingung auf die übergebene Bedingung `condition`. Eingesetzt wird eine Kopie der übergebenen Bedingung `condition` . Die ersetzte Bedingung wird gelöscht.

Eingabeparameter:

- `condition`: Die Bedingung, welche die aktuelle Bedingung der If-Unterfunktion ersetzen soll.

Rückgabe: keine

54.5.16 cFunctionDelay für die Delayfunktion

Die Klasse `cFunctionDelay` realisiert die Delayfunktion für eine Variable. Bei der Auswertung von Fib-Objekten werden einige Zweige mehrfach durchlaufen (z. B. für jeden Wert eines Bereichselements). Die Delayunterfunktion gibt den Wert zurück, den die Variable vor UF_1-Aufrufen der Delayunterfunktion eingenommen hat. Der Wert UF_1 wird dabei durch die erste Unterfunktion bestimmt.

cFunctionDelay

Syntax: `cFunctionDelay(`
`cFibVariable * pVariable,`
`cUnderFunction * pUnderFunctionDelay,`
`cUnderFunction * pUnderFunctionDefault, unsignedLongFib uiMaxDelay=0 )`

Der Konstruktor der Delay-Unterfunktion, er erstellt eine Delay-Unterfunktion. Dabei wird keine Kopie von den Objekten `pUnderFunctionDelay` oder `pUnderFunctionDefault` erstellt.

Eingabeparameter:

- `pVariable`: Einen Zeiger auf die Variable X der Delayunterfunktion.
- `pUnderFunctionDelay`: Einen Zeiger auf die erste Unterfunktion UF_1, welche die Delayunterfunktion enthalten soll. Die übergebene Unterfunktion `pUnderFunctionDelay` wird vor dem Einfügen nicht kopiert. Der Wert dieser Unterfunktion gibt an, wieviel Durchläufe (UF_1) der zurückzugebene Variablenwert her sein soll.
- `pUnderFunctionDefault`: Einen Zeiger auf die zweite Unterfunktion UF_2, welche die Delayunterfunktion enthalten soll. Die übergebene Unterfunktion `pUnderFunctionDefault` wird vor dem Einfügen nicht kopiert. Der Wert dieser Unterfunktion gibt den Wert an, den die Delayunterfunktion einnehmen soll, wenn keine Variablenbelegung von vor UF_1-Aufrufen existiert.
- `uiMaxDelay`: Eine Zahl, welche die maximale Anzahl der zu speicherenden Variablenbelegungen von `pVariable` angibt. Wenn 0 (Standardwert) werden alle Variablenbelegungen von `pVariable` gespeichert. Wenn der Wert von `pUnderFunctionDelay` größer als `uiMaxDelay` ist wird der Wert von `pUnderFunctionDefault` zurückgegeben.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Die Delayunterfunktion speichert sich für jeden Aufruf a eines Durchgangs den Wert W_a, den die Variable X in diesem Aufruf a annimmt. Beim Aufruf n gibt sie den Wert W_{n-UF_1} zurück, den die Variable im Aufruf $n - UF_1$ angenommen hat. Ist $n - UF_1$ kleiner als 1 oder UF_1 größer als `uiMaxDelay`, wird der Wert der Unterfunktion UF_2 zurückgegeben.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Wert zurück, den die Variable X vor UF_1-Aufrufen der Delayunterfunktion eingenommen hat oder, wenn dies nicht möglich ist, der Standardwert UF_2.

54.5.17 getVariable

Syntax: `cFibVariable * getVariable()`

Diese Methode gibt einen Zeiger auf die Variable der Delayunterfunktion zurück.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die Variable der Delayunterfunktion.

54.5.18 setVariable

Syntax: `void setVariable( cFibVariable * pVariable)`

Diese Methode setzt die Variable auf die übergebene Variable `pVariable`.

Eingabeparameter:

- `pVariable`: Einen Zeiger auf die Variable für die Delayunterfunktion.

Rückgabe: keine

55 Die Klasse cCondition für Bedingungen

Die Klasse `cCondition` ist die Basisklasse aller Bedingungen. Aus einzelnen Bedingungen werden zusammengesetzte Bedingungen (mit der Struktur von Bäumen) zusammengestellt.

Die Klasse `cCondition` dient als Basisklasse aller Bedingungen. Von der Klasse `cCondition` können keine Instanzen erzeugt werden.

In Abbildung 20 ist das Klassendiagramm der Bedingungen zu sehen.

Es gibt vier Arten von Bedingungen:

- Nullstellige (keine Eingabeparameter bzw. enthaltende Bedingungen)
- Einstellige (einen Eingabeparameter bzw. enthaltende Bedingungen)
- Zweistellige (zwei Eingabeparameter bzw. enthaltende Bedingungen)
- Vergleiche (arbeiten auf zwei Unterfunktionen, welche Werte/ Zahlen bereitstellen)

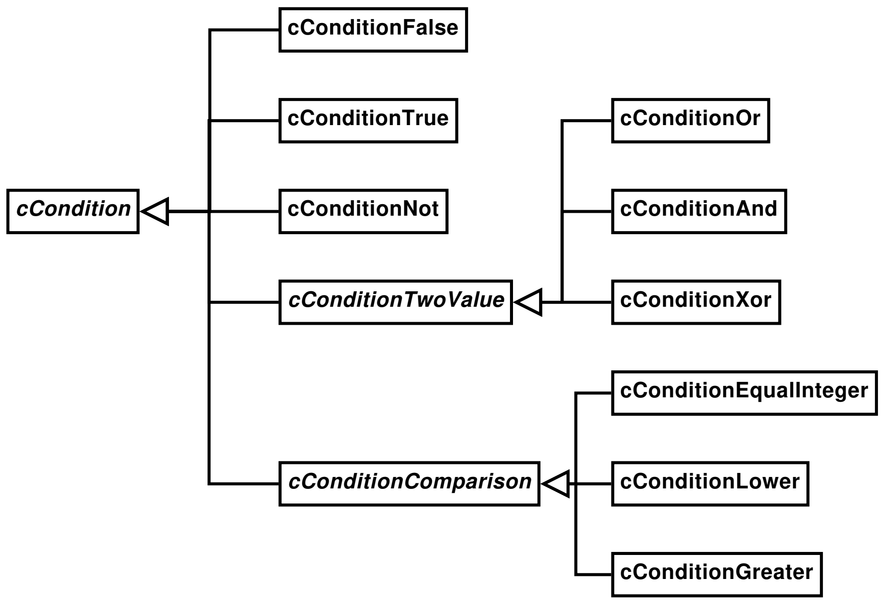


Abbildung 20: Klassengraph der Bedingungen

Die Klassen für die nullstelligen Bedingungen realisieren Wahrheitswerte, diese sind:

- `cConditionFalse`: Diese Bedingung ist immer falsch/ Unwahr.
- `cConditionTrue`: Diese Bedingung ist immer wahr.

Die Klasse für die einstelligen Bedingung ist `cConditionNot` . Sie realisiert die Negation einer Aussage.

Die Klassen, welche zweistelligen Bedingung darstellen, werden von der Basisklasse `cConditionTwoValue` abgeleitet.

Zu diesen zweistelligen Bedingungen gehören:

- `cConditionOr`: Die “Oder” Verknüpfung zweier Wahrheitswerte.
- `cConditionAnd`: Die “Und” Verknüpfung zweier Wahrheitswerte.
- `cConditionXor`: Die “Entweder oder” Verknüpfung zweier Wahrheitswerte.

Die Klassen, welche Vergleiche von Werten implementieren, werden von der Basisklasse `cConditionComparison` abgeleitet.

Zu diesen Vergleichen gehören:

- `cConditionEqualInteger`: Prüfung auf Gleichheit, zweier auf Ganzzahlen gerundeter Werte.
- `cConditionLower`: Prüfung, ob der erste Wert kleiner als der zweite ist.
- `cConditionGreater`: Prüfung, ob der erste Wert größer als der zweite ist.

55.1 Schnittstellenbeschreibung

55.1.1 getValue

Syntax: `bool getValue() const`

Diese Methode wertet die Bedingung aus. Dabei werden auch in der Bedingung enthaltenden Bedingung ausgewertet.

Wenn eine enthaltende Bedingung der Bedingung nicht existiert, wird für sie der `false` (=falsch) angenommen.

Eingabeparameter: keine

Rückgabe: Der Wahrheitswert zu dem die Bedingung ausgewertet wurde.

55.1.2 getType

Syntax: `intFib getType() const`

Diese Methode gibt eine Ganzzahl zurück, über die der Typ der Bedingung bestimmt werden kann. Für diese Typen werden auch Konstanten in der Basisklasse `cCondition` definiert.

Mögliche Typen sind:

Typzeichen	Bedingung	Konstante
00	`cConditionFalse`: Diese Bedingung ist immer ist falsch/ Unwahr.	`CONDITION_FALSE`
01	`cConditionTrue`: Diese Bedingung ist immer ist wahr.	`CONDITION_TRUE`
10	`cConditionNot`: Diese Bedingung realisiert die Negation einer Aussage.	`CONDITION_NOT`
20	`cConditionOr`: Die “Oder” Verknüpfung zweier Wahrheitswerte.	`CONDITION_OR`
21	`cConditionAnd`: Die “Und” Verknüpfung zweier Wahrheitswerte.	`CONDITION_AND`
22	`cConditionXor`: Die “Entweder oder” Verknüpfung zweier Wahrheitswerte.	`CONDITION_XOR`
30	`cConditionEqualInteger`: Prüfung auf Gleichheit, zweier auf Ganzzahlen gerundeter Werte.	`CONDITION_EQUAL_INTEGER`
31	`cConditionLower`: Prüfung, ob der erste Wert kleiner als der zweite ist.	`CONDITION_LOWER`
32	`cConditionGreater`: Prüfung, ob der erste Wert größer als der zweite ist.	`CONDITION_GREATER`

Eingabeparameter: keine

Rückgabe: Eine Ganzzahl, über die der Typ der Bedingung bestimmt werden kann.

55.1.3 isValid

Syntax: `bool isValid() const`

Diese Methode prüft, ob die Bedingung korrekt ist.

Damit eine Bedingung korrekt ist, müssen alle direkt oder indirekt enthaltenden Bedingung korrekt sein. Die in einer korrekten Bedingung enthaltende Vergleichen (von der Klasse `cConditionComparison` abgeleitet Unterklassen) enthaltenden Elemente sind korrekte Unterfunktionen (dies wird mit der `isValid()`-Methode der Unterfunktionen geprüft). Eine korrekte Bedingung darf sich nicht selbst enthalten und keine direkt oder indirekt enthaltende Bedingung oder Unterfunktionen mehr als einmal enthalten. Das heißt, die direkt oder indirekt enthaltenden Bedingung und Unterfunktionen haben die Struktur eines (zyklenfreien) Baums.

Eingabeparameter: keine

Rückgabe: Wenn die Bedingung korrekt ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

55.2 cConditionFalse

Die Klasse `cConditionFalse` stellt eine Bedingung dar, die immer falsch ist.

55.2.1 cConditionFalse

Syntax: `cConditionFalse()`

Der Konstruktor der falschen Bedingung, er erstellt einen falschen Bedingung.

Eingabeparameter: keine

Rückgabe: keine

55.2.2 getValue

Syntax: `bool getValue() const`

Diese Methode gibt immer `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird immer `false` (=falsch) zurückgegeben.

55.3 cConditionTrue

Die Klasse `cConditionTrue` stellt eine Bedingung dar, die immer wahr ist.

55.3.1 cConditionTrue

Syntax: `cConditionTrue()`

Der Konstruktor der wahre Bedingung, er erstellt einen wahre Bedingung.

Eingabeparameter: keine

Rückgabe: keine

55.3.2 getValue

Syntax: `bool getValue() const`

Diese Methode gibt immer `true` (=wahr) zurück.

Eingabeparameter: keine

Rückgabe: Es wird immer `true` (=wahr) zurückgegeben.

55.4 cConditionNot

Die Klasse `cConditionNot` stellt eine Bedingung dar, welche den entgegengesetzten Wahrheitswert hat, wie die enthaltende Bedingung. Ist die enthaltende Bedingung wahr ist die `cConditionNot` Bedingung falsch. Ist die enthaltende Bedingung falsch ist die `cConditionNot` Bedingung wahr.

55.4.1 cConditionNot

Syntax: `cConditionNot( cCondition * pSubCondition )`

Der Konstruktor der Negationsbedingung, er erstellt eine Negationsbedingung.

Von dem Objekt `pSubCondition` wird dabei keine Kope erstellt.

Eingabeparameter:

- `pSubCondition`: Einen Zeiger auf die Bedingung, welche die Negationsbedingung enthalten soll.

Rückgabe: keine

55.4.2 getValue

Syntax: `bool getValue() const`

Diese Methode gibt den Wahrheitswert zurück, der das Gegenteil des Wahrheitswerts der enthaltenden Bedingung (mit `getValue()` dieser ermittelt) ist.

Ist die enthaltende Bedingung `true` (=wahr), ist die `cConditionNot` Bedingung `false` (=falsch). Ist die enthaltende Bedingung `false` (=falsch), ist die `cConditionNot` Bedingung `true` (=wahr).

Eingabeparameter: keine

Rückgabe: Den Wahrheitswert, der das Gegenteil des Wahrheitswerts der enthaltenden Bedingung ist (mit `getValue()` dieser ermittelt).

55.4.3 getSubCondition

Syntax: `cCondition * getSubCondition()`

Diese Methode gibt einen Zeiger auf die in der Negationsbedingung enthaltende Bedingung zurück.

Existiert keine enthaltende Bedingung, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die in der Negationsbedingung enthaltende Bedingung oder der Nullpointer `NULL` , wenn keine enthaltende Bedingung existiert.

55.4.4 setSubCondition

Syntax: `bool setSubCondition( cCondition * pSubCondition, bool bDeleteOld=true )`

Diese Methode setzt die in der Negationsbedingung enthaltende Bedingung auf `pSubCondition`. Dabei wird keine Kopie von `pSubCondition` erstellt.

Eingabeparameter:

- `pSubCondition`: Einen Zeiger auf die Bedingung, welche die Negationsbedingung enthalten soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte enthaltende Bedingung gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die alte enthaltende Bedingung (aus dem Arbeitsspeicher) gelöscht. Standardwert ist `true`, um die alte enthaltende Bedingung zu löschen.

Rückgabe: Wenn die entahltende Bedingung auf `pSubCondition` gesetzt wurde `true` (=wahr), sonst `false` (=falsch)

55.5 cConditionTwoValue

Die Klasse `cConditionTwoValue` ist die Basisklasse aller zweistelligen Bedingungen. Von der Klasse `cConditionTwoValue` können keine Instanzen erzeugt werden.

Zweistellig Bedingungen enthalten zwei Bedingungen. Beim Ermitteln des Wahrheitswerts der Bedingungen mit `getValue()` , werden zuerst die Wahrheitswerts der enthaltenden Bedingungen ermittelt (mit `getValue()`), wobei der Wahrheitswerts der ersten enthaltenden Bedingungen zuerst ermittelt wird. Existiert eine enthaltenden Bedingungen nicht, wird der Wahrheitswert falsch (=`false`) als Wert dieser enthaltenden Bedingungen angenommen. Bei der Berechnung des

Wahrheitswerts der Bedingung, wird also für fehlende Bedingungen der Wahrheitswert falsch (=`false`) eingesetzt.

Zweistellige Bedingungen sind:

- `cConditionOr`: Die “Oder” Verknüpfung zweier Wahrheitswerte.
- `cConditionAnd`: Die “Und” Verknüpfung zweier Wahrheitswerte.
- `cConditionXor`: Die “Entweder oder” Verknüpfung zweier Wahrheitswerte.

Im Folgenden werden zuerst die Methoden für zweistellige Bedingungen vorgestellt und dann die Klassen der möglichen zweistellige Bedingungen.

55.5.1 getFirstSubCondition

Syntax: `cCondition * getFirstSubCondition()`

Diese Methode gibt einen Zeiger auf die erste in der Bedingungen enthaltende Bedingung zurück.

Existiert keine erste enthaltende Bedingung, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die erste in der Bedingungen enthaltende Bedingung oder der Nullpointer `NULL` , wenn keine erste enthaltende Bedingung existiert.

55.5.2 setFirstSubCondition

Syntax: `bool setFirstSubCondition(`
`cCondition * pSubCondition, bool bDeleteOld=true )`

Diese Methode setzt die erste in der Bedingungen enthaltende Bedingung auf die übergebende Unterbedingung `pSubCondition`. Dabei wird keine Kopie von `pSubCondition` erstellt.

Eingabeparameter:

- `pSubCondition`: Einen Zeiger auf die Bedingung, welche die erste Unternbedingungen enthalten soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte erste enthaltende Bedingung gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die erste alte enthaltende Bedingung (aus dem Arbeitsspeicher) gelöscht, sonst, wenn er `false` (=falsch) ist, nicht. Standardwert ist `true`, um die erste alte enthaltende Bedingung zu löschen.

Rückgabe: Wenn die entahltende erste Bedingung auf `pSubCondition` gesetzt wurde `true` (=wahr), sonst `false` (=falsch)

55.5.3 getSecondSubCondition

Syntax: `cCondition * getSecondSubCondition()`

Diese Methode gibt einen Zeiger auf die zweite in der Bedingungen enthaltende Bedingung zurück.

Existiert keine zweite enthaltende Bedingung, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die zweite in der Bedingungen enthaltende Bedingung oder der Nullpointer `NULL` , wenn keine zweite enthaltende Bedingung existiert.

55.5.4 setSecondSubCondition

Syntax: `bool setSecondSubCondition(`
`cCondition * pSubCondition, bool bDeleteOld=true )`

Diese Methode setzt die zweite in der Bedingungen enthaltende Bedingung auf `pSubCondition`. Dabei wird keine Kopie von `pSubCondition` erstellt.

Eingabeparameter:

- `pSubCondition`: Einen Zeiger auf die Bedingung, welche die zweite Unternbedingungen enthalten soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte zweite enthaltende Bedingung gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die zweite alte enthaltende Bedingung (aus dem Arbeitsspeicher) gelöscht, sonst, wenn er `false` (=falsch) ist, nicht. Standardwert ist `true`, um die zweite alte enthaltende Bedingung zu löschen.

Rückgabe: Wenn die entahltende zweite Bedingung auf `pSubCondition` gesetzt wurde `true` (=wahr), sonst `false` (=falsch)

55.5.5 cConditionOr

Die Klasse `cConditionOr` realisiert die "Or"-Bedingung zweier Unterbedingungen. Die "Or"-Bedingung ist wahr (=`true`), wenn mindestens eine der Unterbedingungen wahr (=`true`) ist, sonst ist sie falsch (=`false`).

cConditionOr

Syntax: `cConditionOr( cCondition * pFirstSubCondition, cCondition * pSecondSubCondition )`

Der Konstruktor der “Or”-Bedingung, er erstellt eine “Or”-Bedingung. Es wird keine Kopie von `pFirstSubCondition` oder `pSecondSubCondition` erstellt.

Eingabeparameter:

- `pFirstSubCondition`: Einen Zeiger auf die erste Unterbedingung, welche die “Or”-Bedingung enthalten soll.
- `pSecondSubCondition`: Einen Zeiger auf die zweite Unterbedingung, welche die “Or”-Bedingung enthalten soll.

Rückgabe: keine

getValue

Syntax: `bool getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn mindestens einer der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) ist, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn mindestens einer der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) ist, sonst `false` (=falsch).

55.5.6 cConditionAnd

Die Klasse `cConditionAnd` realisiert die “And”-Bedingung zweier Unterbedingungen. Die “And”-Bedingung ist wahr (=`true`), wenn beide der Unterbedingungen wahr (=`true`) sind, sonst ist sie falsch (=`false`).

cConditionAnd

Syntax: `cConditionAnd( cCondition * pFirstSubCondition, cCondition * pSecondSubCondition )`

Der Konstruktor der “And”-Bedingung, er erstellt eine “And”-Bedingung. Dabei wird keine Kopie von dem Objekt `pFirstSubCondition` oder dem Objekt `pSecondSubCondition` erstellt.

Eingabeparameter:

- `pFirstSubCondition`: Einen Zeiger auf die erste Unterbedingung, welche die “And”-Bedingung enthalten soll.
- `pSecondSubCondition`: Einen Zeiger auf die zweite Unterbedingung, welche die “And”-Bedingung enthalten soll.

Rückgabe: keine

getValue

Syntax: `bool getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn beide der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) sind, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn beide der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) sind, sonst `false` (=falsch).

55.5.7 cConditionXor

Die Klasse `cConditionXor` realisiert die “Xor”-Bedingung zweier Unterbedingungen. Die “Xor”-Bedingung ist wahr (=`true`), wenn genau eine der Unterbedingungen wahr (=`true`) ist, sonst ist sie falsch (=`false`).

cConditionXor

Syntax: `cConditionXor( cCondition * pFirstSubCondition, cCondition * pSecondSubCondition )`

Der Konstruktor der “Xor”-Bedingung, er erstellt eine “Xor”-Bedingung. Dabei wird keine Kopie von dem Objekt `pFirstSubCondition` oder dem Objekt `pSecondSubCondition` erstellt.

Eingabeparameter:

- `pFirstSubCondition`: Einen Zeiger auf die erste Unterbedingung, welche die “Xor”-Bedingung enthalten soll.
- `pSecondSubCondition`: Einen Zeiger auf die zweite Unterbedingung, welche die “Xor”-Bedingung enthalten soll.

Rückgabe: keine

getValue

Syntax: `bool getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn genau einer der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) ist, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn genau einer der Wahrheitswerte der enthaltenden Unterbedingungen (mit `getValue()` dieser ermittelt) `true` (=wahr) ist, sonst `false` (=falsch).

55.6 cConditionComparison

Die Klasse `cConditionComparison` ist die Basisklasse aller Vergleiche. Von der Klasse `cConditionComparison` können keine Instanzen erzeugt werden.

Vergleiche geschehen immer auf zwei Unterfunktionen.

Vergleiche enthalten zwei Unterfunktionen. Beim Ermitteln des Wahrheitswerts des Vergleichs mit `getValue()` , werden zuerst die Werte der enthaltenden Unterfunktionen ermittelt (mit `getValue()`), wobei der Wert der ersten enthaltenden Unterfunktion zuerst ermittelt wird. Existiert eine enthaltenden Unterfunktion nicht, wird der Nullwert des Definitionsbereichs für Unterfunktionen als Wert dieser Unterfunktion angenommen. Bei der Berechnung des Wahrheitswerts, wird also für fehlende Unterfunktionen der Nullwert eingesetzt.

Vergleiche sind:

- `cConditionEqualInteger`: Prüfung auf Gleichheit, zweier auf Ganzzahlen gerundeter Werte.
- `cConditionLower`: Prüfung, ob der erste Wert kleiner als der zweite ist.
- `cConditionGreater`: Prüfung, ob der erste Wert größer als der zweite ist.

Im Folgenden werden zuerst die Methoden für Vergleiche vorgestellt und dann die Klassen der möglichen Vergleiche.

55.6.1 getFirstUnderFunction

Syntax: `cUnderFunction * getFirstUnderFunction()`

Diese Methode gibt einen Zeiger auf die erste in dem Vergleich enthaltende Unterfunktion zurück.

Existiert keine erste enthaltende Unterfunktion, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die erste in dem Vergleich enthaltende Unterfunktion oder der Nullpointer `NULL` , wenn keine erste enthaltende Unterfunktion existiert.

55.6.2 setFirstUnderFunction

Syntax: `void setFirstUnderFunction( cUnderFunction *pUnderFunction, bool bDeleteOld=true )`

Diese Methode setzt die erste in dem Vergleich enthaltende Unterfunktion auf `pUnderFunction`. Dabei wird keine Kopie von `pUnderFunction` erstellt.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche als die erste Unterfunktion enthalten sein soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die erste alte enthaltende Unterfunktion gelöscht werden soll. Wenn der Wert `true` (=wahr) ist, wird die erste alte enthaltende Unterfunktion (aus dem Arbeitsspeicher) gelöscht, sonst, wenn der Wert `false` (=falsch) ist, nicht. Standardwert ist `true`, um die erste alte enthaltende Unterfunktion zu löschen.

Rückgabe: keine

55.6.3 getSecondUnderFunction

Syntax: `cUnderFunction * getSecondUnderFunction()`

Diese Methode gibt einen Zeiger auf die zweite in dem Vergleich enthaltende Unterfunktion zurück.

Existiert keine zweite enthaltende Unterfunktion, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die zweite in dem Vergleich enthaltende Unterfunktion oder der Nullpointer `NULL` , wenn keine zweite enthaltende Unterfunktion existiert.

55.6.4 setSecondUnderFunction

Syntax: `void setSecondUnderFunction( cUnderFunction *pUnderFunction, bool bDeleteOld=true )`

Diese Methode setzt die zweite in dem Vergleich enthaltende Unterfunktion auf `pUnderFunction`. Dabei wird keine Kopie von `pUnderFunction` erstellt.

Eingabeparameter:

- `pUnderFunction`: Einen Zeiger auf die Unterfunktion, welche als die zweite Unterfunktion enthalten sein soll.
- `bDeleteOld`: Dieser Parameter gibt an, ob die alte zweite enthaltende Unterfunktion gelöscht werden soll. Wenn der Wert `true` (=wahr) ist wird die zweite alte enthaltende Unterfunktion (aus dem Arbeitsspeicher) gelöscht, sonst, wenn der Wert `false` (=falsch) ist, nicht. Standardwert ist `true`, um die zweite alte enthaltende Unterfunktion zu löschen.

Rückgabe: keine

55.6.5 cConditionEqualInteger

Die Klasse `cConditionEqualInteger` realisiert den Vergleich auf Gleichheit der auf Ganzahlen gerundeten Werte zweier Unterfunktionen. Der Vergleich `cConditionEqualInteger` ist wahr (=`true`), wenn die Werte beider Unterfunktionen auf Ganzzahlen gerundet gleich sind, sonst ist sie falsch (=`false`).

Wenn die Werte der Unterfunktionen gerundet werden, wird ihr Wert auf die Ganzzahl gerundet, die ihm am nächsten ist (=kleinster Abstand). Gibt es mehrere nächste Ganzzahlen, so wird die kleinste zum Vergleich genommen (bei $0,5$ wird immer abgerundet). Nach Möglichkeit sollten die Werte der Unterfunktionen aber schon Ganzzahlen entsprechen, so dass es nicht zu Rundungsfehlern kommen kann.

cConditionEqualInteger

Syntax: `cConditionEqualInteger( cUnderFunction * pFirstUnderFunction, cUnderFunction * pSecondUnderFunction )`

Der Konstruktor des Vergleichs auf Gleichheit, er erstellt eine Vergleich auf Gleichheit. Dabei wird keine Kopie von dem Objekt `pFirstUnderFunction` oder dem Objekt `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche im Vergleich auf Gleichheit enthalten sein soll.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche im Vergleich auf Gleichheit enthalten sein soll.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn die Werte beider Unterfunktionen (mit `getValue()` dieser ermittelt) auf Ganzzahlen gerundet gleich sind, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Werte beider Unterfunktionen (mit `getValue()` dieser ermittelt) auf Ganzzahlen gerundet gleich sind, sonst gibt die Methode `false` (=falsch) zurück.

55.6.6 cConditionLower

Die Klasse `cConditionLower` realisiert den Vergleich auf Kleiner der Werte zweier Unterfunktionen. Der Vergleich `cConditionLower` ist wahr (=`true`), wenn die Werte der ersten Unterfunktionen kleiner als der Wert der zweiten Unterfunktionen ist, sonst ist sie falsch (=`false`).

cConditionLower

Syntax:
```
cConditionLower(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor des Vergleichs auf kleiner, er erstellt eine Vergleich auf kleiner. Dabei wird keine Kopie von dem Objekt `pFirstUnderFunction` oder dem Objekt `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche im Vergleich auf kleiner enthalten sein soll.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche im Vergleich auf kleiner enthalten sein soll.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn die Werte der ersten Unterfunktionen (mit `getValue()` dieser ermittelt) kleiner als der Wert der zweiten Unterfunktionen (mit `getValue()` dieser ermittelt) ist, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Werte der ersten Unterfunktionen (mit `getValue()` dieser ermittelt) kleiner als der Wert der zweiten Unterfunktionen (mit `getValue()` dieser ermittelt) ist, sonst gibt die Methode `false` (=falsch) zurück.

55.6.7 cConditionGreater

Die Klasse `cConditionGreater` realisiert den Vergleich auf Größer der Werte zweier Unterfunktionen. Der Vergleich `cConditionGreater` ist wahr (=`true`), wenn die Werte der ersten Unterfunktionen größer als der Wert der zweiten Unterfunktionen ist, sonst ist sie falsch (=`false`).

cConditionGreater

Syntax:
```
cConditionGreater(
 cUnderFunction * pFirstUnderFunction,
 cUnderFunction * pSecondUnderFunction )
```

Der Konstruktor des Vergleichs auf größer, er erstellt eine Vergleich auf größer. Dabei wird keine Kopie von dem Objekt `pFirstUnderFunction` oder dem Objekt `pSecondUnderFunction` erstellt.

Eingabeparameter:

- `pFirstUnderFunction`: Einen Zeiger auf die erste Unterfunktion, welche im Vergleich auf größer enthalten sein soll.
- `pSecondUnderFunction`: Einen Zeiger auf die zweite Unterfunktion, welche im Vergleich auf größer enthalten sein soll.

Rückgabe: keine

getValue

Syntax: `doubleFib getValue() const`

Diese Methode gibt `true` (=wahr) zurück, wenn die Werte der ersten Unterfunktionen (mit `getValue()` dieser ermittelt) größer als der Wert der zweiten Unterfunktionen (mit `getValue()` dieser ermittelt) ist, sonst gibt sie `false` (=falsch) zurück.

Eingabeparameter: keine

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Werte der ersten Unterfunktionen (mit `getValue()` dieser ermittelt) größer als der Wert der zweiten Unterfunktionen (mit `getValue()` dieser ermittelt) ist, sonst gibt die Methode `false` (=falsch) zurück.

Teil VIII
Implementation des genetische Algorithmus

In diesem Teil wird der Implementationsentwurf des genetischen Algorithmus vorgestellt. Dazu gehören die Klassenhierarchien und die Schnittstellenbeschreibungen. Die Schnittstellenbeschreibungen geschieht in pseudo C++.

Der genetische Algorithmus wird dem separaten Namensraum mit dem Namen “enviroment” zugeordnet. Er wird generisch (bzw. allgemein) implementiert, so dass er auch für andere Anwendungen außer Fib verwendet werden kann. In Abbildung 21 ist eine Ablaufskizze des genetischen Algorithmus dargestellt.

Des Weiteren wird in diesem Abschnitt auch der allgemeine Entwurf für die Operatoren vorgestellt, da der genetische Algorithmus nur schwer von diesen getrennt gesehen werden kann.

56 Hilfsmethoden

Hilfsmethoden, die bei der Implementierung hilfreich sind, aber nicht zur eigentlichen Schnittstelle gehören, werden nach Bedarf implementiert, hier aber nicht aufgeführt. Sie sollten nach Möglichkeit von außen nicht zugreifbar sein (also `privat` oder `protected` deklariert sein).

57 Copyright

Der genetische Algorithmus wird unter die GPL (siehe Abschnitt 105 auf Seite 486) gestellt. Er kann also frei angepasst, verändert und eingebunden werden, solange das Ergebnis wieder unter der GPL steht. Da der genetische Algorithmus als eigenständiges Programm zu sehen ist, sollten die zusätzlichen Freiheiten der LGPL (siehe Abschnitt 106 auf Seite 500) für ihn nicht nötig sein.

58 Allgemeine Regeln

Für die Implementation des genetischen Algorithmus gibt es ein paar allgemeine Regeln:

- Der Algorithmus ist so zu modularisieren, dass Module einfach ausgetauscht werden können, um ihn an andere Problemstellungen möglichst leicht anzupassen zu können.
- Nur der Kernalgorithmus kann Individuen löschen.
- Nur der Algorithmus bewertet die Operatoren.

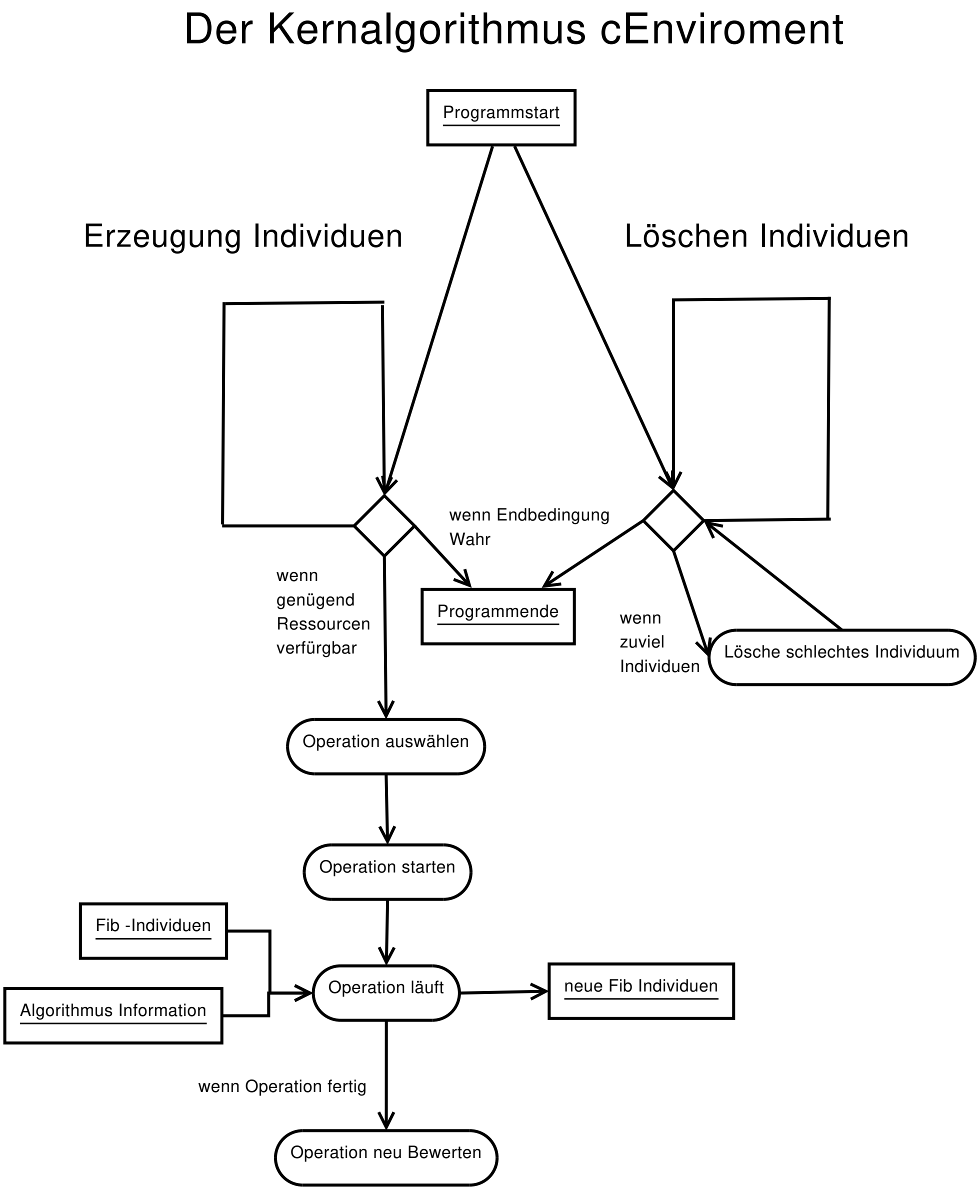


Abbildung 21: Ablaufdiagramm des genetischen Algorithmus

- Nur der Algorithmus kann eine Operation zum Erzeugen neuer Individuen starten.
- Nur der Algorithmus entscheidet, wann er fertig ist und beendet sich. Er kann dabei aber auch auf Signale von außen reagieren.

59 Module

Um den Algorithmus so allgeimein wie möglich zu implementieren, werden einige Funktionen von externen Modulen bzw. Klassen realisiert. Im Nachfolgenden werden diese Module vorgestellt.

59.1 Modul für die Individuen

Die Individuen sind die zentralen Objekte im Algorithmus. Um verschiedene Problemstellungen bearbeiten zu können, müssen Individuen an die jeweile Problemstellung angepasst werden können. Näheres dazu in Abschnitt 60 auf Seite 370.

59.2 Modul für die Initialisierung

Bei Start des Algorithmus (Programmstart) sind einige Aufgaben auszuführen. Dazu gehört beispielsweise, dass ein Individuum erzeugt und in die Individuenpopulation (bzw. Individuenliste) als erstes Individuum eingefügt wird. Näheres dazu in Abschnitt 65 auf Seite 385.

59.3 Module für Entscheidungen

Diese Module werden verwendet, wenn etwas entschieden werden soll. Sie enthalten im Allgemeinen nur eine Funktion, mit der eine Bedingung geprüft werden kann.

59.3.1 Modul für die Ressourcenprüfung

Dieses Modul ist dafür zuständig zu prüfen, ob genug System-/Rechnerressourcen vorhanden sind, um weitere Operationen zu starten und auszuführen. Näheres dazu in Abschnitt 74 auf Seite 415.

Über das Modul zum Auswählen einer Operation werden dann neue Operationen gestartet, wenn genug Ressourcen vorhanden sind.

59.3.2 Modul für die Endbedingung

Das Modul zum Prüfen der Endbedingung zeigt an, wann die Endbedingung erfüllt ist und der Algorithmus beendet werden soll. Näheres dazu in Abschnitt 66 auf Seite 386.

59.3.3 Modul für die maximale Populationsgröße

Damit die Populationsgröße nicht in das Unendliche anwächst, muss diese beschränkt werden. Um zu bewerten, wann "zuviele Individuen" vorhanden sind, dient das Modul für die maximale Populationsgröße. Näheres dazu in Abschnitt 71 auf Seite 410.

Das Löschen von Individuen, um die Population zu beschränken, geschieht allerdings mit dem Modul zum Löschen von Individuen.

59.4 Module für Vorgänge

Die folgenden Module dienen dazu, bestimmte Vorgänge oder Aufgaben auszuführen bzw. zu erfüllen.

59.4.1 Modul zum Bewerten eines Individuums

Um Individuen vergleichen zu können, müssen diese bewertet werden. Dazu dient das Modul zum Bewerten von Individuen. Jedes neu erzeugte Individuum sollte bewertet werden. Diese Bewertung ist dann zum Individuum zu speichern. Das bedeutet, dass dieses Modul aus zwei Teilen besteht: dem Bewerter und der Bewertung für Individuen.

Mehr zur Bewertung im Abschnitt 62 auf Seite 378 und zum Bewerter im Abschnitt 64 auf Seite 382.

59.4.2 Modul zum Auswählen eines guten Individuums

Einige Operatoren werden Informationen aus bereits existierenden Individuen nutzen wollen, um neue Individuen zu erzeugen. Damit die Individuen unabhängig von der nachfragenden Operation gewählt werden, ist die Wahl in einem seperaten Modul im Algorithmus zu implementieren. Dieses Modul sollte gute Individuen bevorzugen und austauschbar sein, um auch andere Wahlstrategien leicht einsetzen zu können. Davon unabhängig können Operationen natürlich ihre eigenen Wahlmechanismen implementieren.

Näheres dazu in Abschnitt 69 auf Seite 408.

59.4.3 Modul zum Löschen von Individuen

Wenn die maximale Populationsgröße überschritten ist, müssen existierende Individuen gelöscht werden. Die Auswahl der zu löschenden Individuen geschieht über ein austauschbares Modul. Die Löschoperation selbst wird im Kernalgorithmus ausgeführt, um zu vermeiden, dass externe Module über eine Schnittstelle Individuen löschen können.

Im Abschnitt 72 auf Seite 412 wird dieses Modul beschrieben.

59.4.4 Modul zum Bewerten einer Operation

Um zu bestimmen, wie gut Operatoren in bestimmten Situationen arbeiten, müssen die Operatoren anhand ihrer Leistung bewertet werden. Da die Art der Operatoren und damit die Bewertungskriterien sich ändern können, muss dieses Modul austauschbar sein.

Das Modul zum Bewerten von Operatoren wird im Abschnitt 75 auf Seite 416 beschrieben.

59.4.5 Modul zum Auswählen einer Operation

Um neue Individuen zu erzeugen, müssen Operatoren ausgewählt werden. Da die Art der Operatoren sich ändern kann, muss dieses Modul austauschbar sein.

Dieses Modul wird im Abschnitt 77 auf Seite 425 beschrieben.

60 Inividuen cIndividual

Der Algorithmus arbeitet auf der Basis von Individuen. Deshalb sind diese von zentraler Bedeutung für den Algorithmus. Individuen sind es, die er schließlich verbessern soll, und das Ergebnis des Algorithmus soll auch ein Individuum sein. Deshalb sind Individuen von zentraler Bedeutung für den Algorithmus.

Die Klasse `cIndividual` repräsentiert ein Individuum in einer abstrakten Form. Dabei hält sie nicht nur die Informationen eines vollständigen Objekts, welches verbessert werden soll, sondern auch weitere Angaben. Mit diesen Angaben soll eine beschleunigte Verarbeitung von Individuen ermöglicht werden.

Zu den Informationen, welche die Klasse `cFibIndividual` bereitstellt, gehören:

- das Objekt, welches das Individuum repräsentiert.
- Zusatzinformationen zu einem Individuum `cIndividualInfo` (Siehe Abschnitt 61 auf Seite 372)

60.1 Schnittstellenbeschreibung

60.1.1 cIndividual

Syntax: `cIndividual( void * pObject,`
`const cIndividualInfo & individualInfo )`

Der Konstruktor für ein Indiviuum; er erzeugt ein Individuum.

Weder von `pObject` noch von `individualInfo` werden Kopien angefertigt. Beide Objekte werden direkt für das Individuum verwendet.

Eingabeparameter:

- `pObject`: Einen Zeiger auf das Objekt, welches das Individuum repräsentieren soll.
- `individualInfo`: Eine Refernz, auf die Zusatzinformationen für das Objekt.

Rückgabe: keine

60.1.2 equal

Syntax: `boolen equal( const cIndividual &individuum, bool bCheckIdentifiers=true )`
oder `bool operation==( const cIndividual &individuum )`

Diese Methoden prüfen, ob das aktuelle und übergebene Individuum gleich sind.

Eingabeparameter:

- `individuum`: Das Individuum, welches gleich zum aktuellen Individuum sein soll.
- `bCheckIdentifiers`: Dieser Wert gibt an, ob bei der Gleichheitsprüfung die beiden Identifier der Objekte (`cIndividualIdentifier` und `cOperationIdentifier`) mit berücksichtigt werden sollen. Wenn er `true` (=wahr) ist, werden die Identifier bei der Prüfung mit berücksichtigt. Sonst, wenn er `false` (=falsch) ist, werden die Identifier bei der Prüfung nicht berücksichtigt. Es werden dann nur die Strukturen der Individuen auf Gleichheit geprüft. Standardwert ist `true` (=wahr), womit auch die Identifier bei der Prüfung berücksichtigt werden.

Rückgabe: Die Methode gibt `true` (=wahr) zurück, wenn das aktuelle Individuum gleich dem übergebenen Individuum ist, sonst `false` (=falsch).

60.1.3 getObjekt

Syntax: `void * getObject()`

Diese Methode gibt einen Zeiger auf das Objekt des Individuums zurück. Das Objekt ist der eigentliche Gegenstand des Algorithmus. Auf dem Individuum arbeiten die Operatoren. Das Objekt ist das, was “verbessert” werden soll. Die anderen Informationen des Individuums dienen nur dazu, dem Algorithmus Zusatzinformationen zum Objekt zu liefern und die Abarbeitung zu beschleunigen.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf das Objekt des Individuums.

60.1.4 getInfo

Syntax: `cIndividualInfo * getInfo()`

Diese Methode liefert einen Zeiger auf die Zusatzinformationen des Individuums zurück. Dies sind Informationen zum Objekt des Individuums, welche dem Algorithmus und den Operatoren die Arbeit erleichtern. Sie enthalten auch allgemeine Informationen (wie z. B. die Erzeugungszeit des Objekts).

Eingabeparameter: keine

Rückgabe: Einen Zeiger auf die Zusatzinformationen des Individuums.

60.1.5 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cIndividual" des Individuumobjekts zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassenname zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cIndividual" des Individuumobjekts.

61 Zusatzinformationen zu einem Individuum cIndividualInfo

Die Klasse `cIndividualInfo` stellt zusätzliche Informationen zu einem Individuum (bzw. Objekt) bereit.

Zu den Informationen, welche die Klasse `cIndividualInfo` bereitstellt, gehören:

- ein Wahrheitswert (boolean), der angibt, ob das Individuum noch lebt bzw. ob das Objekt noch existiert oder es tot ist, bzw. das Objekt gelöscht wurde. Die Klasse `cIndividualInfo` kann noch über das Ableben/ Löschen des Individuums hinaus leben, um beispielsweise spätere statistische Auswertungen zu erlauben.
- ein Identifier `cIndividualIdentifier` für das Individuum. Dies ist eine eindeutige natürliche Zahl für das Inividuum und eine Zahl für die genetischen Algorithmusinstanz, in dem es erzeugt wurde. Die genetische Algorithmusinstanz benötigt einen seperaten Identifier, da es vorgesehen ist,

dass Individuen zwischen genetischen Algorithmusinstanzen migrieren können (zur Klasse `cIndividualIdentifier` siehe Abschnitt 63 auf Seite 381).

- die Fitness des Objekts (zur Klasse `cObjectFitness` siehe Abschnitt 62 auf Seite 378).
- der (lokale) Zeitpunkt der Erzeugung des Objekts.
- die Anzahl der bisher vom Algorithmus ausgeführten Operationen.
- eine Liste mit den Identifieren der direkten Vorfahren des Inividuums. Diese können beispielsweise für Heuristiken interessant sein.
- der Operatorname der Operation, mit der das Objekt erzeugt wurde, inklusive einem freien Feld (eine Unicodezeichenkette), in dem vom Operator weitere Einträge vorgenommen werden können. In dem freien Feld kann beispielsweise von der Operation eingetragen werden, welche Unteroperatoren sie mit welchen Parametern zur Erzeugung des Objekts benutzt hat. Diese Informationen (Name und freies Feld) werden als zwei Zeichenketten abgespeichert.
- der Identifier `cOperationIdentifier` der erzeugenden Operation.
- der zur Erzeugung benötigte Aufwand A (als Systemunabähngige Gleitkommazahl). Dieser wird in Abschnitt 61.1 auf Seite 373 beschrieben.
- die Fitness des besten Individuums zum Erzeugungszeitpunkt, um die Verbesserung einschätzen zu können, welche das Individuum darstellt.

Die Zusatzinformationen `cIndividualInfo` zu einem Individuum sollten auch nach dem Löschen des Individuums/Objekts persistent auf der Festplatte erhalten bleiben. Dafür ist ein Limit für den zur Verfügung stehenden Festplattenplatz zu setzen. Wird dieses Limit für die Zusatzinformationen überschritten, sind die ältesten Zusatzinformationen von der Platte zu löschen, bis das Limit wieder eingehalten wird. Auf diese Weise stehen Informationen zur Verfügung, um die Operationen zu bewerten.

61.1 Der Aufwand von Operationen

Der Aufwand von Operatoren sollte systemunabhängig ermittelt werden, um auch zwischen einzelnen Systemen den Aufwand vergleichen zu können. Dies ist nötig, da der Algorithmus auch verteilt auf mehreren Systemen laufen kann und auch dann die Operatoren vergleichbar bleiben sollten. Der Aufwand wird mit einer Gleitkommazahl repräsentiert.

Der Aufwand für Operationen wird ermittelt, indem die Prozessorzeit (CPU Time“) der Operation mit einem durch einen Benchmark ermittelten Wert multipliziert wird.

Wenn eine Operation mehrere Individuen erzeugt, muss der gesamte Aufwand der Operation über diese Individuen verteilt werden. Die Summe der Aufwände aller von der Operation erzeugten Individuen ist gleich oder größer dem Gesamtaufwand der Operation, welche diese Individuen erzeugt hat. Kein Individuum darf aber weniger Aufwand zugeschrieben werden als $1/16$ des Durchschnittsaufwands der Individuen zu seiner Operation. Wenn also eine Operation I Individuen erzeugte und dazu einen Gesamtaufwand von A benötigte, darf keinem Individuum ein Aufwand A_i kleiner als $A/I * 1/16$ zugeschrieben werden. Mit dieser Regel soll vermieden werden, dass Operatoren einzelne Individuen ”schönrechnen“.

61.1.1 Benchmark

Der Benchmark dient zum Ermitteln eines Benchmarkwertes, mit dem die Prozessorzeit von Operationen skaliert werden kann, um so eine Systemunabhängigkeit der Aufwandwerte zu gewährleisten.

Dafür wird der Benchmark jedes Mal ermittelt, wenn sich seit dem letzten Benchmark leistungsrelevante Rechnerkomponenten geändert haben.

Zu den leistungsrelevanten Rechnerkomponenten gehören:

- CPU (Art, Anzahl, Taktung); alle an der Berechnung beteiligten CPU's sind zu berücksichtigen, eventuell also auch Grafikprozessoren
- Mainbord
- Arbeitsspeicher (Größe, Art, Taktung); aller an der Berechnung beteiligter Arbeitsspeicher ist zu berücksichtigen, eventuell also auch Arbeitsspeicher der Grafikkarte
- neue Compilierung des Algorithmus (neue Algorithmusversion oder Compilerversion)

Um den Benchmarkwert zu ermitteln, wird auf einem festen Objekt/Individuum eine Reihe von deterministischen Operationen ausgeführt und die dafür verbrauchte Prozessorzeit T_B (CPU Time“) gemessen. Das Reziproke der Prozessorzeit ($1/T_B = B$) ist dann der Benchmarkwert B, mit dem die Prozessorzeit (CPU Time“) von Operationen T_O multipliziert wird, um ihren Aufwand zu ermitteln ($A = T_O * B = T_O/T_B$ mit A als systemunabähngige Gleitkommazahl für den Aufwand der Operation).

61.2 Schnittstellenbeschreibung

61.2.1 cIndividualInfo

Syntax: `cIndividualInfo(`
```
unsigned long ulAlgorithmusIdentifier,
list<cIndividualIdentifier> liIdentifierOfParents,
```

```
cObjectFitness fitness,
string szOperationName, string szOperationInfo,
cOperationIdentifier operationIdentifier,
time creationTime, double dOperationCost,
const cObjectFitness * pFitnessOfBestAtCreationTime )
```

Der Konstruktor für ein Objekt für die Zusatzinformationen zu einem Individuum.

Das erzeugte Individuum lebt immer.

Nur über den Konstruktor des Objekts für Zusatzinformationen zu einem Individuum können die Daten dessen gesetzt werden. Für `cIndividualInfo` gibt es keine "set"-Methoden.

Der Identifier des Individuums wird automatisch erzeugt. Zusammen mit dem Identifier des Algorithmus `ulAlgorithmusIdentifier` kann damit das Individuum eindeutig identifiziert werden.

Eingabeparameter:

- `ulAlgorithmusIdentifier`: Der Identifier des Algorithmus.
- `liIdentifierOfParents`: Eine Liste mit den Identifiern der direkten Vorfahren des Inividuums. Die Operation hat diese benutzt, um das Individuum zu erzeugen.
- `fitness`: Die Fitness des Objekts.
- `szOperationName`: Der Name des Operators, mit dem das Individuum erzeugt wurde.
- `szOperationInfo`: Weitere Informationen zur Operation, mit dem das Individuum erzeugt wurde.
- `operationIdentifier`: Der Identifier der Operation, mit der das Individuum erzeugt wurde.
- `creationTime`: Der Zeitpunkt zu dem das Individuum erzeugt wurde.
- `dOperationCost`: Der zur Erzeugung des Individuums benötigte Aufwand.
- `pFitnessOfBestAtCreationTime`: Die Fitness des besten Individuums im Algorithmus zum Zeitpunkt, als das zu `cIndividualInfo` gehörende Individuum erzeugt wurde.

Rückgabe: keine

61.2.2 getIdentifier

Syntax: `cIndividualIdentifier getIdentifier() const`

Diese Funktion liefert den Identifizierer des Individuums. Dies ist eine eindeutige natürliche Zahl für das Inividuum und eine Zahl für die genetische Algorithmusinstanz, in dem es erzeugt wurde (siehe Abschnitt 63 auf Seite 381).

Eingabeparameter: keine

Rückgabe: Den Identifizierer des Individuums.

61.2.3 kill

Syntax: `bool kill()`

Diese Methode deklariert das Individuum für tot bzw. nicht lebendig.

Eingabeparameter: keine

Rückgabe: Wenn das Individuum getötet werden konnte (bzw. es vorher nicht tot war), wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

61.2.4 isLiving

Syntax: `bool isLiving() const`

Diese Funktion gibt zurück, ob das Individuum noch lebt.

Eingabeparameter: keine

Rückgabe: Wenn das Individuum lebt, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

61.2.5 getIdentifiersOfParents

Syntax: `list<cIndividualIdentifier> getIdentifiersOfParents() const`

Diese Funktion gibt eine Liste mit den Identifiern der Elternindividuen zurück. Elternindividuen sind Individuen von, denen die Operation Informationen benutzt hat, um das Individuum zu erzeugen. (Siehe Abschnitt 63 auf Seite 381.)

Eingabeparameter: keine

Rückgabe: Eine Liste mit den Identifiern der Elternindividuen.

61.2.6 getFitness

Syntax: `const cObjectFitness * getFitness() const`

Diese Funktion gibt eine Referenz auf die Fitness des Individuums zurück (siehe Abschnitt 62 auf Seite 378).

Eingabeparameter: keine

Rückgabe: Eine Referenz auf die Fitness des Individuums.

61.2.7 getOperatorName

Syntax: `string getOperatorName() const`

Diese Funktion gibt den Namen des Operators zurück, von dem das Individuum erstellt wurde.

Eingabeparameter: keine

Rückgabe: Den Namen des Operators, von dem das Individuum erstellt wurde.

61.2.8 getOperatorInfo

Syntax: `string getOperatorInfo() const`

Diese Funktion gibt weitere Informationen zu der Operation zurück, von der das Individuum erstellt wurde. Dies können beispielsweise Erstellungsparameter sein.

Eingabeparameter: keine

Rückgabe: Weitere Informationen zu der Operation, von der das Individuum erstellt wurde.

61.2.9 getOperatorIdentifier

Syntax: `cOperationIdentifier getOperatorIdentifier() const`

Diese Funktion gibt den Identifier der Operation zurück, von der das Individuum erstellt wurde (siehe Abschnitt 78.2 auf Seite 430).

Eingabeparameter: keine

Rückgabe: Den Identifier der Operation, von der das Individuum erstellt wurde.

61.2.10 getCreationTime

Syntax: `time getCreationTime() const`

Diese Funktion gibt den Zeitpunkt zurück, an dem das Individuum erzeugt wurde.

Eingabeparameter: keine

Rückgabe: Den Zeitpunkt, an dem das Individuum erzeugt wurde.

61.2.11 getOperationCost

Syntax: `double getOperationCost() const`

Diese Funktion gibt den Aufwand zurück, der zur Erzeugung des Individuums benötigt wurde (siehe Abschnitt 61.1 auf Seite 373).

Eingabeparameter: keine

Rückgabe: Den Aufwand, der zur Erzeugung des Individuums benötigt wurde.

61.2.12 getFitnessOfBestAtCreationTime

Syntax: `const cObjectFitness * getFitnessOfBestAtCreationTime() const`

Diese Funktion gibt eine Referenz auf die Fitness des besten Individuums im Algorithmus zum Erzeugungszeitpunkt des entsprechenden zum `cIndividualInfo` Objekt gehörenden Individuums zurück.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf die Fitness des besten Individuums im Algorithmus zum Erzeugungszeitpunkt des entsprechenden zum `cIndividualInfo` Objekt gehörenden Individuums.

62 Die Fitness von Individuen cObjectFitness

Die Klasse `cObjectFitness` stellt die Fitness eines Objekts (bzw. Individuums) dar. Sie ist die Basisklasse für alle Klassen für die Fitness von Objekten. Von der Klasse `cObjectFitness` selbst oder einer abgeleiteten Klasse kann keine Instanz direkt erzeugt werden. Instanzen von Objekten der Klasse `cObjectFitness` können nur durch die Berechnung der Fitness mit den Klassen für die Algorithmen `cObjectFitnessAlgorithm` (siehe Abschnitt 64 auf Seite 382) oder durch das Laden einer existierenden Fitness aus einem Datenstrom erzeugt werden.

Zu den Fitnessinformationen gehört die Gesamtfitness des Objekts. Abgeleitete Klasse können aber mehrere Fitnesswerte enthalten.

62.1 Schnittstellenbeschreibung

62.1.1 cObjectFitness

Syntax: `cObjectFitness( double dFittness, cObjectFitnessAlgorithm *pObjectFitnessAlgorithmus=NULL )`

Der Konstruktor für ein Fitnessobjekt für ein Individuum.

Nur über den Konstruktor des Fitnessobjekt können die Daten dessen gesetzt werden. Es gibt für `cObjectFitness` keine "set"-Methoden.

Eingabeparameter:

- `dFittness`: Der Fitnesswert für das Inividuum.
- `pObjectFitnessAlgorithmus`: Einen Zeiger auf das Algorithmusobjekt `cObjectFitnessAlgorithm`, mit dem die Fitness erzeugt wurde. Standardwert ist der Nullpointer `NULL` um anzuzeigen, dass kein Algorithmusobjekt diese Fitness erzeugt hat.

Rückgabe: keine

62.1.2 getFitness

Syntax: `double getFitness() const`

Diese Funktion gibt die Fitness des zugehörigen Individuums zurück.

In abgeleiteten Klassen gibt diese Methode die Gesamtfitness des Individuums zurück. Dieser Wert sollte linear sein, d. h. wenn der Wert halb so groß ist, sollte das Individuum auch nur halb so gut sein.

Eingabeparameter: keine

Rückgabe: Die Fitness des zugehörigen Individuums.

62.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cObjectFitness" des Fitnessobjekts zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassenname zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cObjectFitness" des Fitnessobjekts.

62.1.4 getFitnessAlgorithmus

Syntax: `const cObjectFitnessAlgorithm * getFitnessAlgorithmus() const`

Diese Funktion gibt einen Zeiger auf das Algorithmusobjekt (siehe Abschnitt 64 auf Seite 382) zurück, mit das Fitnessobjekte erzeugt wurde. Für Fitnessobjekte vom Typ `cObjectFitness` wird eine Referenz auf das Algorithmusobjekt vom Typ `cObjectFitnessAlgorithm` zurückgegeben.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass das jeweilige Algorithmusobjekt zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf das Algorithmusobjekt vom Typ `cObjectFitnessAlgorithm`.

62.1.5 Gleichheit von cObjectFitness Objekten

Syntax: `bool equal( const cObjectFitness &fitness ) const`
`bool operation==( const cObjectFitness &fitness ) const`

Diese Methoden vergleichen zwei `cObjectFitness` Objekte. Wenn der Typ und die Fitnesswerte im übergebenen Fitnessobjekt gleich dem des aktuellen Objekts vom Typ `cObjectFitness` sind, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

Eingabeparameter: keine

Rückgabe: Wenn das übergebene `cObjectFitness` Objekt gleich dem aktuellen `cObjectFitness` Objekt ist, wird `true` (=wahr) zurückgegeben, ansonsten wird `false` (=falsch) zurückgegeben.

62.1.6 Kleiner Vergleich von cObjectFitness Objekten

Syntax: `bool operation<( const cObjectFitness &fitness ) const`

Dieser Operator vergleicht zwei `cObjectFitness` Objekte. Wenn die Gesamtfitnesswert im übergebenen Objekt größer als das aktuelle `cObjectFitness` ist, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

Eingabeparameter: keine

Rückgabe: Wenn der Gesamtfitnesswert im übergebenen Objekt größer als das aktuelle `cObjectFitness` ist, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

63 cIndividualIdentifier

Die Klasse `cIndividualIdentifier` stellt einen Identifier für das Individuum bereit. Dies ist eine eindeutige natürliche Zahl für das Inividuum und eine natürliche Zahl für die genetische Algorithmusinstanz, in der es erzeugt wurde. Die genetische Algorithmusinstanz benötigt einen seperaten Identifier, da es vorgesehen ist, dass Individuen zwischen genetischen Algorithmusinstanzen migrieren können.

Einen Identifier für ein Individuum sollte es nur maximal einmal geben. Weder auf anderen Rechnern noch für andere Algorithmusläufe sollten Identifier doppelt erzeugt werden.

63.1 Schnittstellenbeschreibung

63.1.1 cIndividualIdentifier

Syntax: `cIndividualIdentifier(`
`unsigned long ulAlgorithmusIdentifier )`

Der Konstruktor für einen Identifier für ein Individuum. Er erzeugt einen eindeutigen Identifier für ein Individuum.

Eingabeparameter:

- `ulAlgorithmusIdentifier`: Der Identifier des Algorithmus.

Rückgabe: keine

63.1.2 getNoIndividualIdentifier

Syntax: `static cIndividualIdentifier &`
`getNoIndividualIdentifier()`

Diese Methode gibt eine Referenz für einen `cIndividualIdentifier` zurück, der für kein Individuum steht.

Dieser Identifier ist kleiner als alle anderen `cIndividualIdentifier` und sollte niemals einem Individuum zugeordnet werden.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf einen fest definierten Identifier `cIndividualIdentifier`, der für kein Individuum steht.

63.1.3 Vergleiche von cIndividualIdentifier Objekten

Syntax: `bool equal(`
` const cIndividualIdentifier &identifier ) const`
`bool operation==(`
` const cIndividualIdentifier &identifier ) const`

Diese Methoden vergleichen zwei `cIndividualIdentifier` Objekte. Wenn das übergebene Objekt gleich dem aktuellen `cIndividualIdentifier` Objekt ist, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

Eingabeparameter: keine

Rückgabe: Wenn das übergebene `cIndividualIdentifier` Objekt gleich dem aktuellen `cIndividualIdentifier` Objekt ist, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

64 Algorithmus für einen Bewerter von Individuen cObjectFitnessAlgorithm

Die Klasse `cObjectFitnessAlgorithm` ist die Basisklasse der Bewerter für die Fitness von Individuen. Von ihr selbst kann keine Instanz erzeugt werden.

64.1 Schnittstellenbeschreibung

64.1.1 cObjectFitnessAlgorithm

Syntax: `cObjectFitnessAlgorithm()`

Der Konstruktor für ein Objekt zum Bewerter von Individuen.

Eingabeparameter: keine

Rückgabe: keine

64.1.2 cObjectFitnessAlgorithm

Syntax: `cObjectFitnessAlgorithm(`
` const cIndividual & original )`

Der Konstruktor für ein Objekt zum Bewerter von Individuen.

Eingabeparameter:

- `original`: Eine Referenz auf das Individuum, welches als Originalindividuum gesetzt werden soll.

Rückgabe: keine

64.1.3 evalueFitness

Syntax: `cObjectFitness * evalueFitness( const cIndividual &individual )`

Diese Methode berechnet die Fitness eines Individuums `individual`. Sie wird nicht von `cObjectFitnessAlgorithm` implementiert und muss daher in den von `cObjectFitnessAlgorithm` abgeleiteten Klassen implementiert werden.

Eingabeparameter:

- `individual`: Eine Referenz auf das Individuum, welches bewertet werden soll.

Rückgabe: Zurückgegeben wird ein Zeiger auf ein Fitnessobjekt für die Fitness des Individuums `individual`, berechnet nach dem aktuellen Individuumbewerter bezüglich des gesetzten Originalindividuums.

64.1.4 setOriginalIndividual

Syntax: `bool setOriginalIndividual( const cIndividual &original )`

Diese Methode wird das Originalindividuum gesetzt. Bezüglich dieses Individuums werden andere Individuen vom Bewerteralgorithmus bewertet.

Das zu setzende Originalindividuum `original` muss den richtigen Typ für den Bewerter haben, um gesetzt werden zu können. Ist der Typ falsch wird das Individuum `original` nicht als Origianl gesetzt und `false` zurückgegeben.

Eingabeparameter:

- `original`: Eine Referenz auf das Individuum, welches als Originalindividuum gesetzt werden soll.

Rückgabe: Wenn das übergebene Individuum als Originalindividuum gesetzt wurde, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

64.1.5 getOriginalIndividual

Syntax: `cIndividual * getOriginalIndividual()`

Diese Methode gibt eine Referenz auf das Originalindividuum zurück. Bezüglich dieses Individuums werden andere Individuen vom Bewerteralgorithmus bewertet.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf das Originalindividuum.

64.1.6 getClassName

Syntax: `string getClassName()`

Diese Methode gibt eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Individuen zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Individuen.

64.1.7 getBestFitness

Syntax: `const cObjectFitness * getBestFitness() const`

Diese Methode gibt die bestmögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass es nicht möglich ist, eine bessere Fitness zu generieren.

Diese Fitness entspricht im Normalfall einem Individuum, welches das Originalobjekt perfekt wiedergibt und dabei keine Recourcen verbraucht.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf die bestmögliche Fitness eines Individuums bezüglich des Originalobjekts oder der Nullzeiger `NULL`, wenn diese nicht berechnet werden kann.

64.1.8 getWorstCaseFitness

Syntax: `const cObjectFitness * getWorstCaseFitness() const`

Diese Methode gibt die schlechtestmögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass sie immer erreicht werden kann.

Im Normalfall wird der Algorithmus die Fitness des Originalindividuums zurückgeben, da dieses ja schon in der entsprechenden Repräsentation vorliegt und verbessert werden soll.

Der Algorithmus kann dennoch durch `evalueFitness()` Fitnessobjekte generieren, die noch schlechter sind.

Es wird der Nullzeiger `NULL` zurückgegeben, wenn die schlechtestmögliche Fitness nicht berechnet werden kann. Dies kann beispielsweise der Fall sein, wenn kein Originalobjekt für den Algorithmus gesetzt wurde.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf die schlechtestmögliche Fitness eines Individuums bezüglich des Originalobjekts oder der Nullzeiger `NULL`, wenn diese nicht berechnet werden kann.

65 Initialisieren des Algorithmus cInitEnviroment

Die Klasse `cInitEnviroment` ist die Basisklasse zur Initialisierung des Algorithmus. Von der Klasse `cInitEnviroment` werden alle Klassen zur Initialisierung des Algorithmus abgeleitet. Von `cInitEnviroment` selbst können allerdings keine Instanzen erzeugt werden, da eine allgemeine Initialisierung nich möglich ist.

Für die Initialisierung steht die Methode `initEnviroment()` bereit.

Die Initialisierung umfasst:

- die Bereitstellung des Originalindividuums. Die Daten für das Originalindividuum kï¿½nnen der von `cInitEnviroment` abgeleiteten Klasse über ihren Konstruktor übergeben werden.
- Erzeugen und Einfügen mindestens eines Individuums für und in die Population. So hat der Algorithmus von Anfang an ein Individuum zum Zurückgeben. Diese Individuen können auch aus einem früheren Lauf des Algorithmus kommen und/oder aus einer Datei geladen werden.
- Aufrufen, wenn vorhanden, des Initialsierungsoperators `cInitOperator`.

65.1 Schnittstellenbeschreibung

65.1.1 cInitEnviroment

Syntax: `cInitEnviroment()`

Der Konstruktor für das Initialisierungsobjekt des Algorithmus (`cEnviroment` siehe Abschnitt 67 auf Seite 393).

Eingabeparameter: keine

Rückgabe: keine

65.1.2 initEnviroment

Syntax: `bool initEnviroment()`

Diese Methode initialisiert den Algorithmus.

Bevor die Initialisierung ausgeführt werden kann, müßen die gewünschten Parameter zur Initialisierung über die "set"-Methoden oder/und die Konstruktoren des von `cInitEnviroment` abgeleiteten Objekts gesetzt werden.

Weiterhin muss geprüft werden, ob die Initialisierung zur aktuellen Instanz von `cEnviroment` passt. Dazu gehört beispielsweise, dass der Typ der bei der Initialisierung erzeugten Individuen zur aktuellen `cEnviroment` Instanz passt.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus `cEnviroment` initialisiert wurde, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

65.1.3 getClassName

Syntax: `string getClassName()`

Diese Methode gibt eine Zeichenkette mit dem Klassennamen des aktuellen Objekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Individuen.

66 Prüfen der Endbedingung mit cEndConditionCheck

Die Klasse `cEndConditionCheck` und von ihr abgeleitete Klassen stellen die Methode `endConditionCheck()` zum Prüfen der Endbedingung bereit. Wenn die Methode `endConditionCheck()` wahr (=`true`) zurückgibt, sollte sich der Algorithmus beenden. Die Methode `endConditionCheck()` sollte in regelmäßigen Abständen (im Zehntelsekundenbereich) vom Algorithmus aufgerufen werden, so dass sich der Algorithmus zeitnah beendet.

Bei folgenden Bedingungen ist die Endbedingung erfüllt und die Methode zur Prüfen der Endbedingung `endConditionCheck()` gibt wahr `true` zurück:

- Es wird ein Signal zum Beenden empfangen (unter Linux: HUP=1, QUIT=3, TERM=15).
- Es wird eine maximale Fitness überschritten.

- Eine maximal Anzahl von Operatorenaufrufen (bzw. Operationen) wird überschritten.
- Eine maximale Prozessorlaufzeit ist überschritten.
- Eine maximale Laufzeit (Realzeit) ist überschritten.
- Ein bestimmter Zeitpunkt ist überschritten.

66.1 Schnittstellenbeschreibung

66.1.1 cEndConditionCheck

Syntax: `cEndConditionCheck()`

Der Konstruktor für ein Objekt zum Prüfen der Endbedingung für den Algorithmus (`cEnviroment` siehe Abschnitt 67 auf Seite 393).

In dem neu konstruierten Objekt ist nur die Bedingung des "Signal zum Beenden" aktiviert. Alle anderen Endbedingungen müssen über die "set"-Methoden gesetzt werden.

Eingabeparameter: keine

Rückgabe: keine

66.1.2 endConditionCheck

Syntax: `bool endConditionCheck()`

Diese Methode gibt zurück, ob der Algorithmus beendet werden soll. Der Algorithmus sollte beendet werden, wenn eine der oben aufgeführten Bedingung erfüllt ist.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus `cEnviroment` beendet werden soll, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

66.1.3 getClassName

Syntax: `string getClassName()`

Diese Methode gibt eine Zeichenkette mit dem Klassennamen des aktuellen Objekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Individuen.

66.1.4 getMaxFitness

Syntax: `cObjectFitness *getMaxFitness() const`

Diese Methode gibt eine Referenz auf ein Fitnessobjekt zurück, dessen Wert von den Individuen nicht überschritten werden soll. Ist die übergebene Referenz `NULL` wird die Prüfung auf die maximale Fitness nicht ausgeführt.

Die Fitnessobjekte der Individuen werden mit dem Fitnessobjekt der maximalen Fitnessbedingung über den kleineren Operator der Fitnessobjekte verglichen. Wird also die Endbedingung mit der `endConditionCheck()` Methode geprüft, und die maximale Fitnessbedingung ist aktiv (bzw. das maximale Fitnessobjekt ist nicht `NULL`), dann wird das Individuum mit der höchsten Fitness im Algorithmus ermittelt und dessen Fitness mit dem maximalen Fitnessobjekt verglichen. Ist die Fitness des maximalen Fitnessobjekts kleiner als die des ermittelten Individuums, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf ein Fitnessobjekt, dessen Wert von den Individuen nicht überschritten werden soll.

66.1.5 setMaxFitness

Syntax: `bool setMaxFitness( cObjectFitness *fitness=NULL )`

Diese Methode setzt das Fitnessobjekt, dessen Wert von den Individuen nicht überschritten werden soll. Ist die übergebene Referenz `fitness` gleich `NULL`, wird die Prüfung auf die maximale Fitness nicht ausgeführt. Das Objekt zur übergebenen Referenz wird, wenn vorhanden, dabei kopiert.

Die Fitnessobjekte der Individuen werden mit dem Fitnessobjekt der maximalen Fitnessbedingung über den kleineren Operator der Fitnessobjekte verglichen. Wird also die Endbedingung mit der `endConditionCheck()` Methode geprüft, und die maximale Fitnessbedingung ist aktiv (bzw. das maximale Fitness Objekt ist nicht `NULL`), dann wird das Individuum mit der höchsten Fitness im Algorithmus ermittelt und dessen Fitness mit dem maximalen Fitnessobjekt verglichen. Ist die Fitness des maximalen Fitnessobjekts kleiner als die des ermittelten Individuums, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter:

- `fitness`: Eine Referenz auf das Fitnessobjekt, dessen Wert von den Individuen nicht überschritten werden soll. Ist die übergebene Referenz `fitness` gleich `NULL`, wird die Prüfung auf die maximale Fitness nicht ausgeführt.

Standardwert von `fitness` ist `NULL`, um die die maximale Fitnessprüfung nicht auszuführen.

Rückgabe: Wenn die maximale Fitness auf `fitness` gesetzt wurde, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

66.1.6 getMaxOperationCalls

Syntax: `unsigned long getMaxOperationCalls() const`

Diese Methode gibt die maximale Anzahl der Opertorenaufrufe (=Operationen) zurück. Ist dieser Wert 0 , wird die Prüfung auf die maximale Operationen nicht ausgeführt bzw. es können beliebig viele Operationen ausgeführt werden.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn die maximale Anzahl von Operationen größer als 0 ist und die Anzahl der Opertionen, die der Algorithmus bisher ausgeführt hat, größer als die maximale Anzahl von Operationen ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Die maximale Anzahl der Opertorenaufrufe (=Operationen).

66.1.7 setMaxOperationCalls

Syntax:
```
bool setMaxOperationCalls(
 unsigned long ulMaxCalls=0 )
```

Diese Methode setzt die maximale Anzahl von Opertorenaufrufen (=Operationen). Ist dieser Wert 0 (Standardwert), wird die Prüfung auf die maximale Operationen nicht ausgeführt bzw. es können beliebig viele Operationen ausgeführt werden.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn die maximale Anzahl von Operationen größer als 0 ist und die Anzahl der Operationen, die der Algorithmus bisher ausgeführt hat, größer als die maximale Anzahl von Operationen ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter:

- `ulMaxCalls`: Die maximale Anzahl von Opertorenaufrufen (=Operationen), welche gesetzt werden soll. Ist der übergebene Wert `ulMaxCalls` gleich 0, wird die Prüfung auf die maximale Operationen nicht ausgeführt bzw. es können beliebig viele Operationen ausgeführt werden. Standardwert von `ulMaxCalls` ist 0, um die maximale Operationenprüfung nicht auszuführen.

Rückgabe: Wenn der die maximale Anzahl von Operationen auf `ulMaxCalls` gesetzt wurde, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

66.1.8 getMaxCpuRuntime

Syntax: `double getMaxCpuRuntime() const`

Diese Methode gibt die maximale CPU-Zeit in Sekunden zurück, die der Algorithmus verbrauchen darf. Ist dieser Wert negativ, wird die Prüfung auf die maximale CPU-Zeit nicht ausgeführt bzw. der Algorithmus kann beliebig viel CPU-Zeit verbrauchen.

Wird die Endbedingung mit der `endConditionCheck()`-Methode geprüft, wenn die maximale CPU-Zeit größer als 0 ist und die aktuelle CPU-Zeit, die der Algorithmus bisher verbraucht hat, größer als die CPU-Zeit ist, wird von von der Methode `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Die maximale CPU-Zeit, die der Algorithmus verbrauchen darf.

66.1.9 setMaxCpuRuntime

Syntax: `bool setMaxCpuRuntime( double dMaxCpuTime=-1.0 )`

Diese Methode setzt die maximale CPU-Zeit in Sekunden, die der Algorithmus verbrauchen darf. Ist dieser Wert negativ (Standardwert), wird die Prüfung auf die maximale CPU-Zeit nicht ausgeführt, bzw. der Algorithmus kann beliebig viel CPU-Zeit verbrauchen.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn die maximale CPU-Zeit größer als 0 ist und die aktuelle CPU-Zeit, die der Algorithmus bisher verbraucht hat, größer als die CPU-Zeit ist, wird von der Methode `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter:

- `dMaxCpuTime`: Die maximale CPU-Zeit in Sekunden, die der Algorithmus verbrauchen darf. Ist der übergebene Wert `dMaxCpuTime` negativ, wird die Prüfung auf die maximale CPU-Zeit nicht ausgeführt, bzw. der Algorithmus kann beliebig viel CPU-Zeit verbrauchen. Standardwert von `dMaxCpuTime` ist -1.0, um die maximale CPU-Zeitprüfung nicht auszuführen.

Rückgabe: Wenn der die maximale CPU-Zeit auf `dMaxCpuTime` gesetzt wurde, wird `true` (=wahr) zurückgegeben, ansonsten `false` (=falsch).

66.1.10 getMaxRuntime

Syntax: `double getMaxRuntime() const`

Diese Methode gibt die maximale Laufzeit in Sekunden zurück, die der Algorithmus verbrauchen darf. Die Laufzeit ist die Zeit, die vergeht, während der Algorithmus läuft (das bedeuted nicht: wärend der Algorithmus gestartet ist und arbeitet). Ist dieser Wert negativ, wird die Prüfung auf die maximale Laufzeit nicht ausgeführt, bzw. der Algorithmus kann beliebig viel Laufzeit verbrauchen.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn die maximale Laufzeit größer als 0 ist und die Laufzeit, die der Algorithmus bisher verbraucht hat, größer als die Laufzeit ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die maximale Laufzeit, die der Algorithmus verbrauchen darf.

66.1.11 setMaxRuntime

Syntax: `bool setMaxRuntime( double dMaxTime=-1.0 )`

Diese Methode setzt die maximale Laufzeit in Sekunden, die der Algorithmus verbrauchen darf. Laufzeit ist die Zeit die vergeht, während der Algorithmus läuft (das bedeuted nicht: wärend der Algorithmus gestartet ist und arbeitet). Ist dieser Wert negativ (Standardwert), wird die Prüfung auf die maximale Laufzeit nicht ausgeführt, bzw. der Algorithmus kann beliebig viel Laufzeit verbrauchen.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn die maximale Laufzeit größer als 0 ist und die Laufzeit, die der Algorithmus bisher verbraucht hat, größer als die Laufzeit ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter:

- `dMaxTime`: Die maximale Laufzeit in Sekunden, die der Algorithmus verbrauchen darf. Ist der übergebene Wert `dMaxTime` negativ, wird die Prüfung auf die maximale Laufzeit nicht ausgeführt, bzw. der Algorithmus kann beliebig viel Laufzeit verbrauchen. Standardwert von `dMaxTime` ist -1.0, um die maximale Laufzeitprüfung nicht auszuführen.

Rückgabe: Wenn der die maximale Laufzeit auf `dMaxTime` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

66.1.12 getMaxDate

Syntax: `time getMaxDate() const`

Diese Methode gibt das Enddatum (inclusive dem Zeitpunkt) zurück, bis zu dem der Algorithmus laufen darf. Ist dieser Wert 0 (als unkonvertierter Datumswert), wird die Prüfung auf das Enddatum nicht ausgeführt, bzw. der Algorithmus kann beliebig lange laufen.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn das Enddatum größer als 0 ist und das aktuelle Datum größer/ später als das Enddatum ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Enddatum, bis zu dem der Algorithmus laufen darf.

66.1.13 setMaxDate

Syntax: `bool setMaxDate( time tMaxDate=0 )`

Diese Methode setzt das Enddatum (inclusive dem Zeitpunkt), bis zu dem der Algorithmus laufen darf. Ist dieser Wert 0 (als unkonvertierter Datumswert), wird die Prüfung auf das Enddatum nicht ausgeführt, bzw. der Algorithmus kann beliebig lange laufen.

Wird die Endbedingung mit der `endConditionCheck()` Methode geprüft, wenn das Enddatum größer als 0 ist und das aktuelle Datum größer/ später als das Enddatum ist, wird von `endConditionCheck()` wahr (=`true`) zurückgegeben.

Eingabeparameter:

- `tMaxDate`: Das Enddatum (inclusive dem Zeitpunkt), bis zu dem der Algorithmus laufen darf. Ist der übergebene Wert `tMaxDate` 0 (als unkonvertierter Datumswert), wird die Prüfung auf das Enddatum nicht ausgeführt, bzw. der Algorithmus kann beliebig lange laufen. Standardwert von `tMaxDate` ist 0, um die Enddatumsprüfung nicht auszuführen.

Rückgabe: Wenn der das Enddatum auf `tMaxDate` gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

67 Kernalgorithmus cEnviroment

Der Kernalgorithmus `cEnviroment` implementiert den genetischen Algorithmus unter zuhilfenahme der Bewerter und Operatoren. Es kann vom Kernalgorithmus bzw. der Umwelt immer nur eine Instanz (im gültigen Prozessraum) existieren.

Im Kernalgorithmus werden zwei Schleife solange ausgeführt, bis die Endbedingung zutrift oder sie, über die Methode `stop()`, gestoppt wird.

Eine Schleife dient zum Erzeugen von Individuen und die andere zum Löschen von Individuen. Dafür werden Operationen ausgewählt (das Auswählen geschieht mit Hilfe von `cChoosOperation` abgeleiteten Klassen) und gestartet, solange noch genügend Rechnerresourcen (Prüfung mit von `cResouceCheck` abgeleiteten Klassen) vorhanden sind. Gestartete Operationen werden vermerkt. Wird eine Operation beendet, so werden die von ihr erzeugten Individuen geprüft und es wird die Operation mit den von ihr erzeugten Individuen bewertet. Wenn eine Operation kein Individum erzeugt, wird für sie vom Kernalgorithmus ein Dummyindividuum eingefügt und die Operation bezüglich dieser bewertet. Dieses Dummyindividuum besteht nur aus Individueninformationen `cIndividualInfo` über ein totes Individuum. Die Fitness dieses Dummyindividuum ist die minimal mögliche Fitness.

Sollte eine Operation schon sehr lange laufen, wird geprüft ob die Operation noch reagiert (mit der `isRunning()` Methode der Operation). Dann wird eventuell versucht die Operation mit ihrer `stop()`-Methode zu beenden. Wenn auch dies nicht zum Erfolg führt, wird der Prozess in dem die Operation läuft beendet.

Der Kernalgorithmus ist generisch für ein von der Klasse `cIndividual` abgeleiteten Individuentyp geschrieben.

67.1 Schnittstelle

67.1.1 cEnviroment

Syntax: `protected cEnviroment()`

Der Konstruktor für ein Kernalgorithmusobjekt. Diese Methode kann nicht von außerhalb der `cEnviroment` Klasse oder von ihr abgeleiteten Klassen aufgerufen werden. Da der Kernalgorithmus nur einmal existieren kann (bzw. ein singelton Objekt ist), kann nur eine Instanz von ihm existieren. Diese Instanz des Kernalgorithmus kann über die Methode `getInstance()` erhalten werden.

Über die `setParameter()` Methode die Parameter des Kernalgorithmus gesetzt werden, bevor dieser gestartet werden kann (mit `start()`).

Eingabeparameter: keine

Rückgabe: keine

67.1.2 setParameter

Syntax:
```
static bool setParameter( const cInitEnviroment * pInit,
 const cObjectFitnessAlgorithm * pObjectFitnessAlgorithmus,
 const cEndConditionCheck * pEndCondition=NULL ,
 const cIndividualSelection * pIndividualSelection=NULL,
 const cMaximumReached * pMaximumIndividuals=NULL,
 const cSelectIndividualToDelete * pSelectIndividualToDelete=NULL,
 const cOperatorFitnessAlgorithm * pOperationFitnessAlgorithm=NULL,
 const cChoosOperator * pChoosOperator=NULL,
 const cResourceCheck * pResourceCheck=NULL )
```

Diese Methode setzt die Parameter für den Kernalgorithmus.

Erst nachdem die Parameter gesetzt wurden kann der Kernalgorithmus (mit `start()`) gestartet werden.

Wenn ein oder mehrere Parameter nicht gesetzt werden können, wird `false` (=falsch) zurückgegeben. In diesem Fall werden die Parameter des Algorithmus nicht verändert, d. h. der Aufruf von `setParameter` bewirkt dann keine Änderung.

Alle Parameter sind Zeiger, da für sie auch abgeleitete Klassen verwendet werden können. Von allen Parametern werden Kopien erzeugt und für den Algorithmus verwendet.

Eingabeparameter:

- `pInit`: Das Objekt zum Initialisieren des Kernalgorithmus. (siehe Abschnitt 65 auf Seite 385)

- `pObjectFitnessAlgorithmus`: Mit diesem Objekt wird der Typ der Fitnessobjekte der Individuen im Kernalgorithmus festgelegt. Das Objekt stellt den Algorithmus bzw. eine Funktion zum Erzeugen der Fitness für Individuen bereit. (siehe Abschnitt 64 auf Seite 382)

- `pEndCondition`: Das Objekt zum Prüfen der Endbedingung für den Kernalgorithmus. (siehe Abschnitt 66 auf Seite 386) Als Standardobjekt (wenn `NULL`) wird ein mit dem Standardkonstruktor der Klasse `cEndConditionCheck` erzeugtes Objekt genommen.

- `pIndividualSelection`: Diese Objekt dient der Auswahl von guten Individuen aus der Individuenmenge des Kernalgorithmus. (siehe Abschnitt 69 auf Seite 408) Als Standardobjekt (wenn `NULL`) wird ein mit dem Standardkonstruktor der Klasse `cIndividualSelectionWeel` erzeugtes Objekt genommen.

- `pMaximumIndividuals`: Mit diesem Objekt wird eine Funktion bereitgestellt, über welche die maximale Anzahl von Individuen im Kernolgorithmus begrenzt werden kann. (siehe Abschnitt 71 auf Seite 410) Als Standardobjekt (wenn NULL) wird ein mit dem Standardkonstruktor der Klasse cMaximumReached erzeugtes Objekt genommen.

- pSelectIndividualToDelete: Diese Objekt dient der Auswahl von Individuen aus der Individuenmenge des Kernalgorithmus, welche gelöscht werden sollen. (siehe Abschnitt 72 auf Seite 412) Als Standardobjekt (wenn NULL) wird ein mit dem Standardkonstruktor der Klasse cSelectIndividualToDeleteWeel erzeugtes Objekt genommen.

- pOperationFitnessAlgorithm: Mit diesem Objekt wird der Typ der Fitnessobjekte der Operatoren festgelegt. Das Objekt stellt den Algorithmus bzw. eine Funktion zum Erzeugen der Fitness für Operatoren bereit. (siehe Abschnitt 75 auf Seite 416) Als Standardobjekt (wenn NULL) wird ein mit dem Standardkonstruktor der Klasse cOperatorFitnessAlgorithmBasic erzeugtes Objekt genommen.

- pChoosOperator: Diese Objekt dient der Auswahl von guten Operatoren aus der Operatorenmenge. (siehe Abschnitt 77 auf Seite 425) Als Standardobjekt (wenn NULL) wird ein mit dem Standardkonstruktor der Klasse cChoosOperator erzeugtes Objekt genommen.

- pResourceCheck: Dieses Objekt dient zum Prüfen, ob noch Ressourcen vorhanden sind, um weitere Operationen zu starten. (siehe Abschnitt 74 auf Seite 415) Als Standardobjekt (wenn NULL) wird ein mit dem Standardkonstruktor der Klasse cResourceCheck erzeugtes Objekt genommen.

Rückgabe: Wenn die Parameter gesetzt wurden, wird true (=wahr) zurückgegeben, sonst false (=falsch).

67.1.3 getInstance

Syntax: `static cEnviroment* getInstance()`

Diese Funktion gibt eine Referenz auf die Instanz des Kernalgorithmus bzw. der Umwelt zurück.

Nur ein Kernalgorithmus dessen Parameter mit setParameter() gültig gesetzt wurden, gilt als funktionierender Kernalgorithmus. Sollten die Parameter nicht gesetzt sein, wird deshalb dein Nullpointer NULL zurückgegeben, um anzuzeigen, dass es kein funktionierender Kernalgorithmus gibt.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf die Instanz des Kernalgorithmus oder NULL, wenn kein funktionierender Kernalgorithmus existiert.

67.1.4 start

Syntax: `bool start()`

Diese Methode startet den Algorithmus und kehrt zurück. Der Algorithmus wird dann im Hintergrund laufen, bis er gestoppt wird. Nach dem Starten wird der Algorithmus anfangen seine Schleifen zu durchlaufen (also z. B. Operationen aufzurufen).

Das Starten kann fehlschlagen, wenn beispielsweise Parameter falsch gesetzt sind, dann wird `false` (=falsch) zurückgegeben und der Algorithmus wird nicht laufen bzw. gestartet.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus gestartet wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

67.1.5 stop

Syntax: `bool stop()`

Diese Methode hält den Algorithmus an und kehrt danach zurück. Die Methode kehrt also erst zurück, wenn alle Operationen des Algorithmus beendet bzw. angehalten sind.

Das Anhalten kann fehlschlagen, wenn beispielsweise Parameter das Anhalten nicht zulassen.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus gestoppt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

67.1.6 isRunning

Syntax: `bool isRunning()`

Diese Methode gibt zurück, ob der Algorithmus gerade läuft, bzw. ob Operationen vom Algorithmus ausgeführt werden.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus läuft, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

67.1.7 getAlgorithmusIdentifier

Syntax: `unsigned long getAlgorithmusIdentifier()`

Diese Methode gibt den Identifier des Algorithmus zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Identifier des Algorithmus.

68 Schnittstelle für die Operatoren

Die nachfolgenden Methoden gehören zum Kernalgorithmus `cEnviroment` und dienen den Operatoren um Informationen über den Kernalgorithmus zu erhalten.

Den Operatoren werden vom Kernalgorithmus Individuen bzw. Objekte und möglichkeiten zur Bewertung dieser bereitgestellt. Ob und welche dieser Möglichkeiten die Operatoren nutzen, bleibt ihnen überlassen. Ihnen steht es auch frei die Erstellung der Fitness eines neuen Objekts selbst auszuführen.

Der Kernalgorithmus wird die von dem Operator ermittelte Fitness allerdings überprüfen. Ist die Fitness dann falsch, wird sie korrigiert. Die Zeit zum Ausführnung der Prüfung wird auf die Erzeugungszeit des Opertors für das Individuum mit aufgerechnet. Die Prüfung geschiet bei neuen Operatoren für jedes Inividuum. Wenn der Operator die Fitness sehr oft hintereinander nahe der richtigen Fitness bestimmt, nimmt der Anteil der Fitnessprüfungen mit der Zeit ab. Wird die Fitness allerdings vom Operator falsch bestimmt, nimmt der Anteil der Fitnessprüfungen für den Operator wieder zu.

68.1 Methoden rund um die Individuen

Die folgenden Methoden der Umwelt `cEnviroment` dienen dazu, um die Informationen über die Individuen der Umwelt bzw. des Algorithmus abzufragen.

Im Allgemeinen ist es schneller Informationen über die Individuen vom Typ `cIndividualInfo` zu bekommen, als die zugehörigen ganzen Individuen von Typ `cIndividual`. Da die Individuen `cIndividual` auch immer ihre Informationen `cIndividualInfo` enthalten und somit die Individuen `cIndividual` größer sind.

68.1.1 getIndividualInfos

Syntax: `list<const cIndividualInfo*> getIndividualInfos( short iLive=0 ) const`

Diese Methode gibt eine Liste mit den Individueninformationen `cIndividualInfo` zu den Individuen im Algorithmus zurück.

Eingabeparameter:

- `iLive`: Der Wert von `iLive` gibt an, ob die Individueninformationen von lebende, tote oder allen zurückgegeben werden sollen. Zu toten Individuen existiert kein Individuenobjekt mehr, sondern nur noch die Individueninformationen. Standardwert von `iLive` ist 0, so daß die Individueninformationen aller Individuen zurückgegeben werden. Folgende Werte von `iLive` sind zulässig:

  0 : Wenn `iLive` gleich 0 ist, werden die Individueninformationen aller Individuen zurückgegeben, egal ob sie leben oder tot sind.

  1 : Wenn `iLive` gleich 1 ist, werden die Individueninformationen aller noch lebenden Individuen zurückgegeben.

  -1 : Wenn `iLive` gleich -1 ist, werden die Individueninformationen aller nicht mehr lebenden bzw. toten Individuen zurückgegeben.

  – In allen andern Fällen wird eine leere Liste zurückgegeben. Dies sollte nicht benutzt werden, da diese Bereich für spätere Erweiterungen vorgesehen sind.

Rückgabe: Eine Liste mit den Individueninformationen `cIndividualInfo` zu den Individuen im Algorithmus.

68.1.2 getIndividualInfo

Syntax: `const cIndividualInfo * getIndividualInfo( const cIndividualIdentifier * identifier ) const`

Diese Methode gibt eine Referenz auf die entsprechenden Individueninformationen `cIndividualInfo` des Individuums mit dem Identifier `identifier` zurück. Wenn kein solches Individuum existiert, wird der Nullpointer `NULL` zurückgegeben.

Eingabeparameter:

- `identifier`: Der Identifier des Individuums, für das die Individueninformationen `cIndividualInfo` zurückgegeben werden sollen.

Rückgabe: Eine Referenz auf die Individueninformationen `cIndividualInfo` des Individuums mit dem Identifier `identifier` oder der Nullpointer `NULL`, wenn kein solches Individuum existiert.

68.1.3 getBestIndividualInfo

Syntax: `const cIndividualInfo * getBestIndividualInfo( unsigned long lNumber=1, short iLive=1 ) const`

Diese Methode gibt eine Referenz auf die Individueninformationen `cIndividualInfo` des `lNumber`'ten besten Individuums zurück. (die Zählung beginnt bei 1) Wenn kein solches Individuum existiert, wird der Nullpointer `NULL` zurückgegeben.

Wenn also alle Individuen nach ihrer Fitness absteigend geordnet werden, so dass das beste Individuum bei 1 vorn steht, werden die Individueninformationen des `lNumber`'ten Individuums zurückgegeben.

Der Parameter `iLive` gibt an, welcher Art die Individuen sein sollen, welche betrachtet werden. Betrachtet werden standardmaßig nur alle lebenden Individuen.

Eingabeparameter:

- `lNumber`: Die Nummer die angibt, von welchen `lNumber`'te besten Individuum die Individueninformationen `cIndividualInfo` zurückgegeben werden sollen. Standardmäßig ist der Wert 1 und es werden die Individueninformationen des besten Individuums zurückgegeben.

- `iLive`: Der Wert von `iLive` gibt an, ob die Individueninformationen von lebende, tote oder allen betrachtet werden sollen. Zu toten Individuen existiert kein Individuenobjekt mehr, sondern nur noch die Individueninformationen. Folgende Werte von `iLive` sind zulässig:

  0 : Wenn `iLive` gleich 0 ist, werden die Individueninformationen aller Individuen betrachtet, egal ob sie leben oder tot sind.

  1 : Wenn `iLive` gleich 1 ist, werden die Individueninformationen aller noch lebenden Individuen betrachtet.

  -1 : Wenn `iLive` gleich -1 ist, werden die Individueninformationen aller nicht mehr lebenden bzw. toten Individuen betrachtet.

  – In allen andern Fällen wird der Nullpointer `NULL` zurückgebenen. Dies sollte nicht benutzt werden, da diese Bereich für spätere Erweiterungen vorgesehen sind.

Rückgabe: Eine Referenz auf die Individueninformationen `cIndividualInfo` des `lNumber`'ten besten Individuums oder der Nullpointer `NULL`, wenn kein solches Individuum existiert.

68.1.4 getIndividual

Syntax: `cIndividual * getIndividual() const`

Diese Methode gibt eine Referenz ein Individuum zurück. Zurückgegeben wird eine Kopie des Individuums (inclusive des enthaltenden Objekts), die vom Benutzer gelöscht werden muss.

Die Auswahl des Individuums aus der Menge der Individuen des Algorithmus geschieht über die, von der `cIndividualSelection` Klasse abgeleitete, übergebene Instanz zur Auswahl eines Individuums, welche mit `setParameter()` von `cEnviroment` gesetzt wurde.

Dabei werden nur lebende Individuen betrachtet.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf ein Individuum.

68.1.5 getIndividual

Syntax: `cIndividual * getIndividual( cIndividualIdentifier identifier ) const`

Diese Methode gibt eine Referenz auf das Individuum mit dem angegebenen Identifier `identifier` zurück. Wenn kein solches Individuum existiert, wird der Nullpointer `NULL` zurückgegeben.

Zurückgegeben wird eine Kopie des Individuums (inclusive des enthaltenden Objekts), die vom Benutzer gelöscht werden muss.

Eingabeparameter:

- `identifier`: Der Identifier des Individuums, welches zurückgegeben werden sollen.

Rückgabe: Eine Referenz auf das Individuum mit dem Identifier `identifier` oder der Nullpointer `NULL`, wenn kein solches Individuum existiert.

68.1.6 getBestIndividual

Syntax: `cIndividual * getBestIndividual( unsigned long lNumber=1 ) const`

Diese Methode gibt eine Referenz auf das `lNumber`'ten besten Individuum zurück. (die Zählung beginnt bei 1) Wenn kein solches Individuum existiert, wird der Nullpointer `NULL` zurückgegeben.

Zurückgegeben wird eine Kopie des Individuums (inclusive des enthaltenden Objekts), die vom Benutzer gelöscht werden muss.

Wenn also alle Individuen nach ihrer Fitness absteigend geordnet werden, so dass das beste Individuum bei 1 vorn steht, wird das `lNumber`'te Individuum zurückgegeben. Dabei werden nur lebende Individuen betrachtet.

Eingabeparameter:

- `lNumber`: Die Nummer die angibt, welches `lNumber`'te besten Individuum zurückgegeben werden sollen. Standardmäßig ist der Wert 1 und es wird das besten Individuum zurückgegeben.

Rückgabe: Eine Referenz auf das `lNumber`'ten besten Individuum oder der Nullpointer `NULL`, wenn kein solches Individuum existiert.

68.1.7 getFitnessAlgorithmus

Syntax: `cObjectFitnessAlgorithm & getFitnessAlgorithmus() const`

Diese Methode gibt eine Referenz auf den die Algorithmusinstanz zum Bewerten von Individuen (diese ist vom Typ `cObjectFitnessAlgorithm`) zurück.

Diese Algorithmusinstanz wurde mit der `setParameter()`-Methode von `cEnviroment` gesetzt.

Eingabeparameter: keine

Rückgabe: Eine Referenz auf den die Algorithmusinstanz zum Bewerten von Individuen.

68.1.8 insertIndividual

Syntax: `bool insertIndividual( const cIndividual * individual )`

Diese Methode fügt eine Kopie des übergebenen Individuums in die Menge der Individuen des Kernalgorithmus bzw. der Umwelt ein.

Passt der Typ des Individuums nicht zum Typ der Individuen im Kernalgorithmus, wird das übergebene Individuum nicht eingefügt und `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `individual`: Ein Zeiger auf das Individuum, welches in den Algorithmus eingefügt werden soll.

Rückgabe: Wenn das übergebene Individuum in den Kernalgorithmus eingefügt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

68.1.9 getNumberOfIndividuals

Syntax: `unsigned long getNumberOfIndividuals( short iLive=1 )`

Diese Methode liefert die Anzahl der Individuen im Algorithmus zurück.

Eingabeparameter:

- `iLive`: Der Wert von `iLive` gibt an, ob die Anzahl der Individuen von lebende, tote oder allen zurückgegeben werden sollen. Zu toten Individuen existiert kein Individuenobjekt mehr, sondern nur noch die Individueninformationen. Standardwert von `iLive` ist 0, so daß die Anzahl aller Individuen zurückgegeben wird. Folgende Werte von `iLive` sind zulässig:
  - 0 : Wenn `iLive` gleich 0 ist, wird die Anzahl aller Individuen zurückgegeben, egal ob sie leben oder tot sind.
  - 1 : Wenn `iLive` gleich 1 ist, wird die Anzahl aller noch lebenden Individuen zurückgegeben.
  - -1 : Wenn `iLive` gleich -1 ist, wird die Anzahl aller nicht mehr lebenden bzw. toten Individuen zurückgegeben.
  - – In allen andern Fällen wird 0 zurückgegeben. Dies sollte nicht benutzt werden, da diese Bereich für spätere Erweiterungen vorgesehen sind.

Rückgabe: Die Anzahl der Individuen im Algorithmus.

68.2 Informationen über den Algorithmus

Die im diesem Abschnitt zusammengestellten Methoden liefern allgemeine Informationen über den Algorithmus. Diese Methoden sind nicht vorranging für die Operatoren vorgesehen, sondern eher für die verschiedenen Module (z. B. für `cResourceCheck`) des Algorithmus.

68.2.1 getNumberOfRunningOperations

Syntax: `unsigned int getNumberOfRunningOperations()`

Diese Methode liefert die Anzahl der zur Zeit laufenden Operationen zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der zur Zeit laufenden Operationen.

68.2.2 getNumberOfCalledOperations

Syntax: `unsigned long getNumberOfCalledOperations()`

Diese Methode liefert die Anzahl der bisher vom Algorithmus aufgerufenen Operatoren bzw. der Operationen zurück.

Eingabeparameter: keine

Rückgabe: Die Anzahl der bisher vom Algorithmus aufgerufenen Operatoren bzw. der Operationen.

68.2.3 getCpuRunTime

Syntax: `double getCpuRunTime() const`

Diese Methode liefert Zeit in Sekunden zurück, die der Algorithmus (inklusive Operationen) bisher gelaufen ist. Zu dieser Zeit gibt die Methode `isRunning()` `true` (=wahr) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegebenen wird die Zeit in Sekunden die der Algorithmus (inklusive Operationen) bisher gelaufen ist.

68.2.4 getLastStartTime

Syntax: `time getLastStartTime() const`

Diese Methode liefert den Zeitpunkt an dem der Algorithmus das letzte mal gestartet wurde.

Eingabeparameter: keine

Rückgabe: Den Zeitpunkt an dem der Algorithmus zuletzt gestartet wurde.

68.2.5 getFirstStartTime

Syntax: `time getFirstStartTime() const`

Diese Methode liefert den Zeitpunkt an dem der Algorithmus das Erste mal gestartet wurde.

Eingabeparameter: keine

Rückgabe: Den Zeitpunkt an dem der Algorithmus das Erste mal gestartet wurde.

68.2.6 getInitEnviroment

Syntax: `static cInitEnviroment * getInitEnviroment()`

Diese Methode liefert eine Kopie des Initialisierungsobjekt des Kernalgorithmuss zurück. (siehe Abschnitt 65 auf Seite 385)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf des Initialisierungsobjekt des Kernalgorithmuss.

68.2.7 getEndConditionCheck

Syntax: `static cEndConditionCheck * getEndConditionCheck()`

Diese Methode gibt eine Kopie des Objekts zurück, das die Endbedingung des Kernalgorithmuss bereitstellt. (siehe Abschnitt 66 auf Seite 386)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, das die Endbedingung des Kernalgorithmuss bereitstellt.

68.2.8 getIndividualSelection

Syntax: `static cIndividualSelection * getIndividualSelection()`

Diese Methode gibt eine Kopie des Objekts zurück, welches die Standardauswahl von Individuen aus den Kernalgorithmus realisiert. (siehe Abschnitt 69 auf Seite 408)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, welches die Standardauswahl von Individuen aus den Kernalgorithmus realisiert.

68.2.9 getMaximumReached

Syntax: `static cMaximumReached * getMaximumReached()`

Diese Methode gibt eine Kopie des Objekts zurück, welches prüft, ob die maximale Anzahl von Individuen erreicht ist. (siehe Abschnitt 71 auf Seite 410)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, welches prüft, ob die maximale Anzahl von individuen erreicht ist.

68.2.10 getSelectIndividualToDelete

Syntax: `static cSelectIndividualToDelete * getSelectIndividualToDelete()`

Diese Methode gibt eine Kopie des Objekts zurück, welches ein Individuum zum Löschen aus dem Kernalgorithmus auswählt. (siehe Abschnitt 72 auf Seite 412)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, welches ein Individuum zum Löschen aus dem Kernalgorithmus auswählt.

68.2.11 getOperatorFitnessAlgorithm

Syntax: `static cOperatorFitnessAlgorithm * getOperatorFitnessAlgorithm()`

Diese Methode gibt eine Kopie des Objekts zurück, mit welchem die Fitness von Operatoren berechnet wird. (siehe Abschnitt 75 auf Seite 416)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, mit welchem die Fitness von Operatoren berechnet wird.

68.2.12 getChoosOperator

Syntax: `static cChoosOperator * getChoosOperator()`

Diese Methode gibt eine Kopie des Objekts zurück, mit welchem gute Operatoren ausgewählt werden. (siehe Abschnitt 77 auf Seite 425)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, mit welchem gute Operatoren ausgewählt werden.

68.2.13 getResourceCheck

Syntax: `static cResourceCheck * getResourceCheck()`

Diese Methode gibt eine Kopie des Objekts zurück, welches die Verfügbarkeit von Rechnerresourcen prüft. (siehe Abschnitt 74 auf Seite 415)

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf das Objekt, welches die Verfügbarkeit von Rechnerresourcen prüft.

68.2.14 removeIndividual

Syntax: `protected bool removeIndividual( const cIndividualIdentifier IndividualId )`

Diese Methode löscht das Inidividuum für den die übergebenen Identifier aus die Menge der Individuen des Kernalgorithmus bzw. der Umwelt.

Ist kein entsprechendes Individuum im Kernalgorithmus vorhanden, wird das übergebene Individuum nicht gelöscht und `false` (=falsch) zurückgegeben.

Eingabeparameter:

- `IndividualId`: Der Identifier für das Individuum, welches aus dem Algorithmus gelöscht werden soll.

Rückgabe: Wenn das Individuum aus dem Kernalgorithmus gelöscht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

68.2.15 registerIndividualListener

Syntax: `bool registerIndividualListener( clNewIndividualListener * individualListener )`

Diese Methode registriert einen `clNewIndividualListener`-Listener. Über diesen Listener werden Nachrichten über Änderungen in der Menge der Individuen weitergegeben werden.

Eingabeparameter:

- `individualListener`: Dies ist ein Zeiger auf den entsprechenden `clNewIndividualListene` Listener, welcher im Algorithmus registriert werden soll.

Rückgabe: Wenn der übergebene `clNewIndividualListener`-Listener im Kernalgorithmus registriert wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

68.2.16 unregisterIndividualListener

Syntax: `bool unregisterIndividualListener( clNewIndividualListener * individualListener )`

Diese Methode deregistriert einen `clNewIndividualListener`-Listener. Über den deregistrierten Listener werden keine Nachrichten vom Algorithmus mehr weitergegeben.

Eingabeparameter:

- `individualListener`: Dies ist ein Zeiger auf den entsprechenden `clNewIndividualListener`-Listener, welcher im Algorithmus deregistriert werden soll.

Rückgabe: Wenn der übergebene `clNewIndividualListener`-Listener im Kernalgorithmus deregistriert wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

68.2.17 OPERATION_DOMAIN

Syntax: `const string OPERATION_DOMAIN=""`

Diese Konstante gibt die Art der Individuen an auf denen der Kernalgorithmus arbeitet. Dies ist der Klassenname der Individuen auf denen der Kernalgorithmus arbeitet. Diese Angabe sollte zu den jeweiligen Angaben der verwendeten Operatoren passen.

68.2.18 Der Listener clNewIndividualListener

Die Klasse `clNewIndividualListener` realisiert einen Listener. Über diesen Listener wird beim Hinzufügen oder Löschen eines Individuums, aus der Menge der Individuen im Algorithmus, eine Nachricht weitergegeben.

individualAdded

Syntax: `void individualAdded( const cIndividual * pIndividual )`

Diese Methode wird vom Algorithmus aufgerufen, wenn ein neues Individuum in die Menge der Individuen im Algorithmus hinzugefügt wurde.

Eingabeparameter:

- `pIndividual`: Einen Zeiger auf das Individuum, welches zum Algorithmus hinzugefügt wurde.

Rückgabe: keine

individualRemoved

Syntax: `void individualRemoved( const cIndividual * pIndividual )`

Diese Methode wird vom Algorithmus aufgerufen, wenn ein Individuum aus der Menge der Individuen im Algorithmus gelöscht wurde.

Eingabeparameter:

- `pIndividual`: Einen Zeiger auf das Individuum, welches aus dem Algorithmus gelöscht wurde.

Rückgabe: keine

69 Auswählen eines Individuums mit cIndividualSelection

Die abstrakte Klasse `cIndividualSelection` stellt eine Methode bereit, um ein Individuum auszuwählen. Bei der Auswahl des Individuums werden gute Individuen bevorzugt. Auf diese Weise erhalten die Operatoren, über die Algorithmenschnittstelle für Operatioren, die Möglichkeit ein wahrscheinlich gutes Individuum zu bekommen.

69.1 Schnittstellenbeschreibung

69.1.1 cIndividualSelection

Syntax: `cIndividualSelection()`

Der Konstruktor für ein Objekt zum Auswählen von Individuen.

Eingabeparameter: keine

Rückgabe: keine

69.1.2 getIndividual

Syntax: `cIndividual * getIndividual() const`

Diese Methode gibt eine Referenz auf ein ausgewähltes Individuum zurück.

Von dieser Methode werden die Individuen mit einer Auswahlmethode ausgewählt.

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf ein ausgewähltes Individuum.

69.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cIndividualSelection" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cIndividualSelection" des Objects.

70 Weel-selection auswählen eines Individuums mit cIndividualSelectionWeel

Elternklasse: `cIndividualSelection`

Die Klasse `cIndividualSelectionWeel` implementiert eine Methode bereit, um ein Individuum nach der weel-selection Auswahlmethode auszuwählen. Sie wird von der Klasse `cIndividualSelection` (siehe Abschnitt 70 auf Seite 409) abgeleitet.

Von der Klasse `cIndividualSelectionWeel` werden die Individuen mit der weel-selection Auswahlmethode ausgewählt. Dabei hat jedes Individuum eine Auswahlwahrscheinlichkeit die proportional zu seiner Gesamtfitness ist und die Summe der Gesamtfitness aller Individuen steht für die Wahrscheinlichkeit 1 .

Die schlechtest mögliche Fitness von Individuen wird durch die Methode `getWorstCaseFitness()` des entsprechenden Fitnessalgorithmus (siehe 64 auf Seite 382) zur jeweiligen Fitness ermittelt. Von der Fitness aller Individuen wird der Wert der schlechtest mögliche Fitness abgezogen, um einen Nullpunkt zu erhalten. Bei Individuen deren Auswahlwahrscheinlichkeit p dann kleiner als 1 durch die Anzahl der zur Auswahl stehenden Individuen ($\sharp Individuen$) durch 1024 ist, wird die Auswahlwahrscheinlichkeit auf diesen Wert gesetzt. ($p_{Min} = 1/(\sharp Individuen * 1024)$; if $p < p_{Min}$ then $p = p_{Min}$) So dass jedes Individuum eine Auswahlwahrscheinlichkeit ungleich 0 hat. Existiert kein Fitnessalgorithmus oder schlechtest mögliche Fitness zur Fitness, wird deren Auswahlwahrscheinlichkeit auf p_{Min} gesetzt.

Die schlechtest mögliche Fitness kann dabei durchaus von Individuen unterschritten werden. Sie dient in `cIndividualSelectionWeel` nur zum Festlegen eines Nullpunktes für die Fitnesswerte, um die weel-selection ausführen zu können.

70.1 Schnittstellenbeschreibung

70.1.1 cIndividualSelectionWeel

Syntax: `cIndividualSelectionWeel()`

Der Konstruktor für ein Objekt zum Auswählen von Individuen.

Eingabeparameter: keine

Rückgabe: keine

70.1.2 getIndividual

Syntax: `cIndividual * getIndividual() const`

Diese Methode gibt eine Referenz auf ein ausgewähltes Individuum zurück.

Die schlechtest mögliche Fitness von Individuen wird durch die Methode `getWorstCaseFitness()` des entsprechenden Fitnessalgorithmus (siehe 64 auf Seite 382) zur jeweiligen Fitness ermittelt. Von der Fitness aller Individuen wird der Wert der schlechtest mögliche Fitness abgezogen, um einen Nullpunkt zu erhalten. Bei Individuen deren Auswahlwahrscheinlichkeit p dann kleiner als 1 durch die Anzahl der zur Auswahl stehenden Individuen ($\sharp Individuen$) durch 1024 ist, wird die Auswahlwahrscheinlichkeit auf diesen Wert gesetzt. ($p_{Min} = 1/(\sharp Individuen * 1024)$; if $p < p_{Min}$ then $p = p_{Min}$) So dass jedes Individuum eine Auswahlwahrscheinlichkeit ungleich 0 hat. Existiert kein Fitnessalgorithmus oder schlechtest mögliche Fitness zur Fitness, wird deren Auswahlwahrscheinlichkeit auf p_{Min} gesetzt.

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf ein ausgewähltes Individuum.

70.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cIndividualSelectionWeel" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cIndividualSelectionWeel" des Objects.

71 Prüfung auf ob die maximale Populationsgröße überschritten ist mit cMaximumReached

Die Population/ Liste mit Individuen sollte beschränkt sein und nicht ins unendliche wachsen können. Um zu ermitteln, ob das Maximum an Individuen erreicht ist, dient die Klasse `cMaximumReached` und von ihr abgeleitete Klassen. Dieses Maximum muss nicht statisch sein, sondern kann sich durchaus im Verlauf des Algorithmus anpassen.

Die Klasse `cMaximumReached` (aber nicht unbedingt abgeleitete Klassen) realisiert 3 Bedingungen für die maximale Populationsgröße:

1. Es sind mindestens 16 Individuen möglich.
2. Es sind Maximal 65 536 Individuen möglich.
3. Bei mehr als 16 Individuen muß mindestens 10 % des Arbeitsspeichers frei bleiben.

71.1 Schnittstellenbeschreibung

71.1.1 cMaximumReached

Syntax: `bool cMaximumReached()`

Der Konstruktor für ein Objekt zum Prüfen der maximalen Populationsgrößen.

Eingabeparameter: keine

Rückgabe: keine

71.1.2 maximumReached

Syntax: `bool maximumReached()`

Von diese Methode prüft, ob die maximale Populationsgröße erreicht ist. Wenn die maximale Populationsgröße erreicht ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Wenn die maximale Populationsgröße erreicht wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

71.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cMaximumReached“ des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cMaximumReached“ des Objects.

72 Auswählen eines zu löschenden Individuums mit cSelectIndividualToDelete

Die Klasse `cSelectIndividualToDelete` stellt eine Methode bereit, um ein Individuum auszuwählen, das gelöscht werden soll. Alle Klassen zum Auswählen eines zu Löschenden Individuums müssen von `cSelectIndividualToDelete` abgeleitet werden. Bei dieser Auswahl werden schlechte Individuen bevorzugt. Auch Individuen, von denen mehrere ähnliche Individuen im Algorithmus existieren, können bevorzugt werden.

Von der Klasse `cSelectIndividualToDelete` werden alle Klassen zum Auswählen von Individuen zum Löschen abgeleitet. Von `cSelectIndividualToDelete` selbst können aber keine Instanzen erzeugt werden.

Wenn im Algorithmus nur noch ein Individuum vorhanden ist, wird dies niemals ausgewählt. Da der Algorithmus immer mindestens ein Individuum enthalten sollte.

72.1 Schnittstellenbeschreibung

72.1.1 cSelectIndividualToDelete

Syntax: `cIndividualSelection()`

Der Konstruktor für ein Objekt zum Auswählen von Individuen zum Löschen.

Eingabeparameter: keine

Rückgabe: keine

72.1.2 getIndividual

Syntax: `cIndividual * getIndividual()`

Diese Methode gibt eine Referenz auf ein ausgewähltes Individuum zurück.

Wenn der Algorithmus nur noch ein Individuum enthält, wird der Nullpointer `NULL` zurückgegeben. Da das letzte Individuum nicht gelöscht werden kann, weil der Algorithmus mindestens ein Individuum enthalten muss.

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf eines zum Löschen ausgewählten Individuums oder der Nullpointer `NULL`, wenn kein Individuum zum Löschen existiert.

72.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cSelectIndividualToDelete" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cSelectIndividualToDelete" des Objects.

73 Weel-selection auswählen eines zu löschenden Individuums mit cSelectIndividualToDeleteWeel

Elternklasse: `cSelectIndividualToDelete`

Die Klasse `cSelectIndividualToDeleteWeel` stellt eine Methode bereit, um ein Individuum auszuwählen, das gelöscht werden soll. Sie wird von der Klasse `cSelectIndividualToDelete` abgeleitet.

Wenn im Algorithmus nur noch ein Individuum vorhanden ist, wird dies niemals ausgewählt. Da der Algorithmus immer mindestens ein Individuum enthalten sollte.

Von der Klasse `cSelectIndividualToDeleteWeel` werden die Individuen mit der weel-selection Auswahlmethode ausgewählt. Dabei hat jedes Individuum eine Auswahlwahrscheinlichkeit die umgekehrt proportional zu seiner Gesamtfitness ist und die Wahrscheinlichkeiten aller Individuen summiert sich zu `1`.

Die best mögliche Fitness von Individuen wird durch die Methode `getBestFitness()` des entsprechenden Fitnessalgorithmus (siehe 64 auf Seite 382) zur jeweiligen Fitness ermittelt. Von der best mögliche Fitness wird jeweils der Wert Fitness aller Individuen abgezogen, um einen Nullpunkt zu erhalten. Bei Individuen deren Auswahlwahrscheinlichkeit p dann kleiner als 1 durch die Anzahl der zur Auswahl stehenden Individuen ($\sharp Individuen$) durch 1024 ist, wird die Auswahlwahrscheinlichkeit auf diesen Wert gesetzt. ($p_{Min} = 1/(\sharp Individuen * 1024)$; if $p < p_{Min}$ then $p = p_{Min}$) So dass jedes Individuum eine Auswahlwahrscheinlichkeit ungleich 0 hat. Existiert kein Fitnessalgorithmus oder best mögliche Fitness zur Fitness, wird deren Auswahlwahrscheinlichkeit auf p_{Min} gesetzt.

Die best mögliche Fitness kann dabei durchaus von Individuen überschritten werden. Sie dient in `cSelectIndividualToDeleteWeel` nur zum Festlegen eines Nullpunktes für die Fitnesswerte, um die weel-selection ausführen zu können.

73.1 Schnittstellenbeschreibung

73.1.1 cSelectIndividualToDeleteWeel

Syntax: `cSelectIndividualToDeleteWeel()`

Der Konstruktor für ein Objekt zum Auswählen von Individuen zum Löschen.

Eingabeparameter: keine

Rückgabe: keine

73.1.2 getIndividual

Syntax: `cIndividual * getIndividual()`

Diese Methode gibt eine Referenz auf ein ausgewähltes Individuum zurück.

Die best mögliche Fitness von Individuen wird durch die Methode `getBestFitness()` des entsprechenden Fitnessalgorithmus (siehe 64 auf Seite 382) zur jeweiligen Fitness ermittelt. Von der best mögliche Fitness wird jeweils der Wert Fitness aller Individuen abgezogen, um einen Nullpunkt zu erhalten. Bei Individuen deren Auswahlwahrscheinlichkeit p dann kleiner als 1 durch die Anzahl der zur Auswahl stehenden Individuen ($\sharp Individuen$) durch 1024 ist, wird die Auswahlwahrscheinlichkeit auf diesen Wert gesetzt. ($p_{Min} = 1/(\sharp Individuen * 1024)$; if $p < p_{Min}$ then $p = p_{Min}$) So dass jedes Individuum eine Auswahlwahrscheinlichkeit ungleich 0 hat. Existiert kein Fitnessalgorithmus oder best mögliche Fitness zur Fitness, wird deren Auswahlwahrscheinlichkeit auf p_{Min} gesetzt.

Die best mögliche Fitness kann dabei durchaus von Individuen überschritten werden. Sie dient in `cSelectIndividualToDeleteWeel` nur zum Festlegen eines Nullpunktes für die Fitnesswerte, um die weel-selection ausführen zu können.

Wenn der Algorithmus nur noch ein Individuum enthält, wird der Nullpointer `NULL` zurückgegeben. Da das letzte Individuum nicht gelöscht werden kann, weil der Algorithmus mindestens ein Individuum enthalten muss.

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf ein zum Löschen ausgewählten Individuum oder der Nullpointer `NULL`, wenn kein Individuum zum Löschen existiert.

73.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cSelectIndividualToDeleteWeel" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cSelectIndividualToDeleteWeel" des Objects.

74 Ressourcenprüfung mit cResourceCheck

Die Klasse `cResourceCheck` ist die Basisklasse aller Klassen zum Prüfen der Ausführungsressourcen. Dabei wird geprüft, ob genug Rechnerressourcen vorhanden sind, um weitere Operatoren zu starten. Alle Klassen zum Prüfen der Ausführungsressourcen müssen von `cResourceCheck` abgeleitet werden.

Die Klasse `cResourceCheck` realisiert folgende Bedingungen mit den die Rechnerressourcen geprüft werden:

1. Wenn mehr als 8 Operatoren laufen, werden die Rechnerressourcen als belegt angenommen.

74.1 Schnittstellenbeschreibung

74.1.1 cResourceCheck

Syntax: `cResourceCheck()`

Der Konstruktor für ein Objekt zum Prüfen, ob noch genug Rechnerressourcen vorhanden sind.

Eingabeparameter: keine

Rückgabe: keine

74.1.2 enoughResources

Syntax: `bool enoughResources()`

Diese Methode prüft, ob genug Rechnerressourcen vorhanden sind, um weitere Operatoren zu starten. Wenn noch Rechnerressourcen vorhanden sind, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus und der Rechnerhardware.

Eingabeparameter: keine

Rückgabe: Wenn noch Rechnerressourcen vorhanden sind, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

74.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cResourceCheck" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cResourceCheck" des Objects.

75 Bewerter für Operatoren cOperatorFitnessAlgorithm

Mit der Klasse `cOperatorFitnessAlgorithm` ist die Basisklasse für alle Klassen zum Bewerten von Operatoren. Alle Klassen zum Bewerten von Operatoren müssen von `cOperatorFitnessAlgorithm` abgeleitet werden. Von `cOperatorFitnessAlgorithm` können allerdings keine Instanzen erzeugt werden. Die Bewertung kann bei abgeleiteten Klassen auch Situationsabhängig sein.

Der Bewerter hält sich dabei intern für alle Operatoren, die schon bewertet wurden, ein Fitnesswert vor. Dadurch kann beim Aktualisieren der Fitness für neu erzeugte Individuen Rechnerzeit gespaart werden, indem die Fitness nur hinsichtlich des neuen Individuums angepasst wird.

Operatoren die noch nicht bewertet wurden, haben als Fitnesswert den Durchschnitt der Fitnesswerte der schon bewerteten Operatoren.

Objekte von der Klasse `cOperatorFitnessAlgorithm` verwalten zusätzlich eine Liste mit allen möglichen/ aufrufbaren Operatoren.

75.1 Schnittstellenbeschreibung

75.1.1 cOperatorFitnessAlgorithm

Syntax: `cOperatorFitnessAlgorithm()`

Der Konstruktor für ein Objekt zum Bewerten von Operatoren.

Eingabeparameter: keine

Rückgabe: keine

75.1.2 evalueFitness

Syntax: `double evalueFitness( const cOperation &operator, const list<cIndividualInfo> &liCreatedIndividualInfos )`

Diese Methode berechnet die Fitness des Operators zur Operation `operator` neu. Dazu wird angenommen, dass der Operator bisher nur die Individuen erzeugt hat, welchen Informationen in der übergebenen Liste `liCreatedIndividualInfos` stehen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator, für den die Fitness berechnet werden soll.
- `liCreatedIndividualInfos`: Eine Liste mit den Informationen zu den Individuen, die der Operator bisher erzeugt hat.

Rückgabe: Die neu berechnete Fitness des Operators `operator`.

75.1.3 updateFitness

Syntax: `double updateFitness( const cOperation &operator, const cIndividualInfo &createdIndividualInfo )`

Diese Methode berechnet die Fitness des Operators zur Operation `operator` auf Basis der bisherigen von der Operation erzeugten Individuen und dem übergebenen Individuum `createdIndividualInfo`. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher (durch frühere Methodenaufrufe) dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden) und zusätzlich noch das übergebene Individuum `createdIndividualInfo`.

Auf diese Weise muß dem `cOperatorFitnessAlgorithm`-Algoritmus zur Berechnung der Fitness nicht immer alle neu erzeugten Individuen übergeben werden, sondern nur jeweils die zuletzt neu erzeugten Individuen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator, für den die Fitness berechnet bzw. angepasst werden soll.
- `createdIndividualInfo`: Die Informationen zu dem Individuum, die der Operator neu erzeugt hat. Diese wurden noch nicht dem Algorithmus, zur Bewertung der Fitness der Operatoren, übergeben.

Rückgabe: Die berechnete Fitness des Operators `operator`.

75.1.4 updateFitness

Syntax: `double updateFitness( const cOperation &operator, const list<cIndividualInfo> &liCreatedIndividualInfos )`

Diese Methode berechnet die Fitness des Operators zur Operation `operator` auf Basis der bisherigen von dem Operator erzeugten Individuen und dem übergebenen Individuen aus der Liste `liCreatedIndividualInfos`. Dazu wird angenommen, dass die Operation bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden) und zusätzlich noch die übergebenen Individuen aus der Liste `liCreatedIndividualInfos`.

Auf diese Weise muß dem `cOperatorFitnessAlgorithm`-Algoritmus zur Berechnung der Fitness nicht immer alle neu erzeugten Individuen übergeben werden, sondern nur jeweils die zuletzt neu erzeugten Individuen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator für den die Fitness berechnet bzw. angepasst werden soll.
- `liCreatedIndividualInfos`: Eine Liste mit den Informationen zu den Individuen, die der Operator neu erzeugt hat. Diese wurden noch nicht dem Algorithmus, zur Bewertung der Fitness der Operatoren, übergeben.

Rückgabe: Die berechnete Fitness des Operators `operator`.

75.1.5 getFitness

Syntax: `double getFitness( const cOperation &operator )`

Diese Methode berechnet die Fitness des Operators zur Operators zu `operator` auf Basis der bisherigen von dem Operator erzeugten Individuen. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden).

Abgeleitete Klassen wird es überlassen die Fitness erneut zu berechnen oder nicht. Wenn beispielsweise die Fitness eines Operators abhängig von der aktuelle Situation (z. B. der Anzahl der bisherigen Operationsaufrufe) ist, ist eine erneute Berechnung sinnvoll.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator für den die Fitness zurückgegeben werden soll.

Rückgabe: Die berechnete Fitness des Operators `operator`.

75.1.6 getFitness

Syntax: `list< pair< const cOperation*, double > > getFitness()`

Diese Methode berechnet die Fitness aller Operatoren auf Basis der bisherigen von dem Operator erzeugten Individuen. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden).

Abgeleitete Klassen wird es überlassen die Fitness erneut zu berechnen oder nicht. Wenn beispielsweise die Fitness eines Operators abhängig von der aktuelle Situation (z. B. der Anzahl der bisherigen Operationsaufrufe) ist, ist eine erneute Berechnung sinnvoll.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter: keine

Rückgabe: Die berechnete Fitness aller Operatoren `operator` in einer Liste aus Paaren. Das erste Element des jeweiligen Paars ist ein Zeiger auf den Operator und das Zweite auf dessen Fitness.

75.1.7 getClassName

Syntax: `string getClassName() const`

Diese Methode gibt eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Operationen zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette mit dem Klassennamen des aktuellen Bewerters für Operationen.

75.1.8 getPossibleOperators

Syntax: `list<const cOperation*> getPossibleOperators() const`

Diese Methode gibt eine Liste mit den Referenzen auf alle möglichen/aufrufbaren Operator zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Liste mit den Referenzen auf alle möglichen/aufrufbaren Operator.

75.1.9 getMinFitness

Syntax: `double getMinFitness() const`

Diese Methode gibt die kleinst mögliche Fitness zurück, welche ein Operator haben kann.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die kleinst mögliche Fitness für Operatoren.

75.1.10 getFitnessSum

Syntax: `double getFitnessSum() const`

Diese Methode gibt die Summe aller Fitnesswerte aller Operatoren zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Summe aller Fitnesswerte aller Operatoren.

76 Bewerter für Operatoren cOperatorFitnessAlgorithmBasic

Elternklasse: `cOperatorFitnessAlgorithm`

Mit der Klasse `cOperatorFitnessAlgorithmBasic` werden Operatoren bewertet. Sie wird von der Klasse `cOperatorFitnessAlgorithm` abgeleitet.

Der Bewerter hält sich dabei intern für alle Operatoren, die schon bewertet wurden, ein Fitnesswert vor. Dadurch kann beim Aktualisieren der Fitness für neu erzeugte Individuen Rechnerzeit gespaart werden, indem die Fitness nur hinsichtlich des neuen Individuums angepasst wird.

Operatoren die noch nicht bewertet wurden, haben als Fitnesswert den Durchschnitt der Fitnesswerte der schon bewerteten Operatoren.

Die Klasse `cOperatorFitnessAlgorithmBasic` erstellt nur eine einfache Bewertung/ Fitness der Operatoren. Folgende Werte der Operation und Funktionen dieser sind dafür ausschlaggebend:

- $NF()$: Normalisiert einen Fitnesswert F_I eines Individuums, so dass dieser größer 0 ist. Dies geschieht indem von Fitnesswert F_I der minimale Fitnesswert für Individuen F_{min} abgezogen wird. Ist der sich ergebene Wert

$F_I - F_{min}$ dann kleiner als 1 wird er auf 1 gerundet. $N(F_I) = \max(1, (F_I - F_{min}))$

- V_i: Die Verbesserung, welche durch eine Operation erreicht wurde. Dies ist der Qutient der normalisierte Fitness des erzeugten Individuums F_N durch die normalisierte Fitness des besten Individuums F_B, welches bei der Erzeugung des Individuums F_N im Algorithmus vorhanden war und lebte. $V_i = NF(F_N)/NF(F_B)$
- B_{ins}: Summe aller Fitnessverbesserungen die von der Operation bisher realisiert wurden. Für alle V_i die größer 1 sind, wird der Wert $V_i - 1$ aufsummiert (also nur die Verbesserung).
- A_{ins}: Aufwand den die Operation bisher benötigt hat. Dies sind die summierten Werte des Aufwände der Operation (Siehe Abschnitt 61.1 auf Seite 373).
- F_{ins}: Erreichte Fitness insgesamt. Dies ist der Qutient der erreichten Verbesserung insgesamt B_{ins} durch den Aufwand insgesamt $F_{ins} = B_{ins}/A_{ins}$
- A_P: Aufwand den die Operation bisher für Verbesserungen benötigt hat. Dies sind die summierten Werte des Aufwände der Operation, wenn die Operation ein verbessertes Individuum hervorgebracht hat. Also nur wenn V_i größer 1 für den Operationlauf war, wird der Aufwand der Operation zu A_P hinzuaddiert.
- F_P: Erreichte Verbesserungsfitness insgesamt. Dies ist der Qutient der erreichten Verbesserung insgesamt B_{ins} durch den Aufwand für Verbesserungen A_P: $F_P = B_{ins}/A_P$
- F: F ist die Fitness des Operators. Diese ergibt sich Linear aus der erreichte Fitness insgesamt und der erreichte Verbesserungsfitness insgesamt: $F = A * F_{ins} + B * F_P$. Mit den Faktoren A und B kann eingestellt werden, mit welchen Gewicht die jeweiligen Teilfitnesswerte F_{ins} und F_P in die Fitness des Operators F eingehen. Standardwerte für A und B sind 0.5. Über den konstruktor können aber auch andere Werte für A und B festgelegt werden.

76.1 Schnittstellenbeschreibung

76.1.1 cOperatorFitnessAlgorithmBasic

Syntax: `cOperatorFitnessAlgorithmBasic(`
`double dFaktorAllFitnessPart=0.5, double dFaktorPositivFitnessPart=0.5 )`

Der Konstruktor für ein Objekt zum Bewerten von Operatoren.

Eingabeparameter:

- `dFaktorAllFitnessPart`: Gewicht A für erreichte Fitness insgesamt F_{ins} (in $F = A * F_{ins} + B * F_P$). Standardwert ist 0.5 .
- `dFaktorPositivFitnessPart`: Gewicht B für erreichte Verbesserungsfitness insgesamt F_P (in $F = A * F_{ins} + B * F_P$). Standardwert ist 0.5 .

Rückgabe: keine

76.1.2 evalueFitness

Syntax: `bool evalueFitness( const cOperation &operator, const list<cIndividualInfo> &liCreatedIndividualInfos )`

Diese Methode berechnet die Fitness F des Operators zur Operation `operator` neu. Dazu wird angenommen, dass der Operator bisher nur die Individuen erzeugt hat, welchen Informationen in der übergebenen Liste `liCreatedIndividualInfos` stehen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator, für den die Fitness berechnet werden soll.
- `liCreatedIndividualInfos`: Eine Liste mit den Informationen zu den Individuen, die der Operator bisher erzeugt hat.

Rückgabe: Die neu berchnete Fitness F des Operators `operator`.

76.1.3 updateFitness

Syntax: `double updateFitness( const cOperation &operator, const cIndividualInfo &createdIndividualInfo )`

Diese Methode berechnet die Fitness F des Operators zur Operation `operator` auf Basis der bisherigen von dem Operator erzeugten Individuen und dem übergebenen Individuum `createdIndividualInfo`. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden) und zusätzlich noch das übergebene Individuum `createdIndividualInfo`.

Auf diese Weise muß dem `cOperatorFitnessAlgorithm`-Algoritmus zur Berechnung der Fitness nicht immer alle neu erzeugten Individuen übergeben werden, sondern nur jeweils die zuletzt neu erzeugten Individuen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator, für den die Fitness berechnet bzw. angepasst werden soll.
- `createdIndividualInfo`: Die Informationen zu dem Individuum, die der Operator neu erzeugt hat. Diese wurden noch nicht dem Algorithmus, zur Bewertung der Fitness der Operatoren, übergeben.

Rückgabe: Die berechnete Fitness F des Operators `operator`.

76.1.4 updateFitness

Syntax: `double updateFitness( const cOperation &operator, const list<cIndividualInfo> &liCreatedIndividualInfos )`

Diese Methode berechnet die Fitness F des Operators zur Operation `operator` auf Basis der bisherigen von dem Operator erzeugten Individuen und dem übergebenen Individuen aus der Liste `liCreatedIndividualInfos`. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden) und zusätzlich noch die übergebenen Individuen aus der Liste `liCreatedIndividualInfos`.

Auf diese Weise muß dem `cOperatorFitnessAlgorithm`-Algoritmus zur Berechnung der Fitness nicht immer alle neu erzeugten Individuen übergeben werden, sondern nur jeweils die zuletzt neu erzeugten Individuen.

Die Methode holt sich selbständig alle weiteren benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `operator`: Der Operator, für den die Fitness berechnet bzw. angepasst werden soll.
- `liCreatedIndividualInfos`: Eine Liste mit den Informationen zu den Individuen, die der Operator neu erzeugt hat. Diese wurden noch nicht dem Algorithmus, zur Bewertung der Fitness der Operatoren, übergeben.

Rückgabe: Die berechnete Fitness F des Operators `operator`.

76.1.5 getFitness

Syntax: `double getFitness( const cOperation &operator )`

Diese Methode berechnet die Fitness F des Operators `operator` auf Basis der bisherigen von dem Operator erzeugten Individuen. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden).

In Objekten der Klasse `cOperatorFitnessAlgorithmBasic` wird die zuletzt berechnete Fitness zurückgegeben.

Eingabeparameter:

- `operator`: Der Operator für den die Fitness zurückgegeben werden soll.

Rückgabe: Die berechnete Fitness F des Operators `operator`.

76.1.6 getFitness

Syntax: `list< pair< const cOperation*, double > > getFitness()`

Diese Methode berechnet die Fitness aller Operatoren auf Basis der bisherigen von dem Operator erzeugten Individuen. Dazu wird angenommen, dass der Operator bisher die Individuen erzeugt hat, welchen bisher dem `cOperatorFitnessAlgorithm`-Objekt bekannt sind (da sie ihm früher übergeben wurden).

In Objekten der Klasse `cOperatorFitnessAlgorithmBasic` wird die zuletzt berechnete Fitness zurückgegeben.

Eingabeparameter: keine

Rückgabe: Die berechnete Fitness aller Operatoren `operator` in einer Liste aus Paaren. Das erste Element des jeweiligen Paars ist ein Zeiger auf den Operator und das Zweite auf dessen Fitness.

76.1.7 getClassName

Syntax: `string getClassName() const`

Diese Methode gibt eine Zeichenkette mit dem Klassennamen "cOperatorFitnessAlgorithmBasic“ des aktuellen Bewerters für Operationen zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette mit dem Klassennamen "cOperatorFitnessAlgorithmBasic“ des aktuellen Bewerters für Operationen.

76.1.8 getMinFitness

Syntax: `double getMinFitness() const`

Diese Methode gibt 0.0 als die kleinst mögliche Fitness zurück, welche ein Operator haben kann.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird 0.0 .

76.1.9 getFitnessSum

Syntax: `double getFitnessSum() const`

Diese Methode gibt die Summe aller Fitnesswerte aller Operatoren zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Summe aller Fitnesswerte aller Operatoren.

76.1.10 getFaktorAllFitnessPart

Syntax: `double getFaktorAllFitnessPart() const`

Diese Methode gibt das Gewicht A für erreichte Fitness insgesamt F_{ins} (in $F = A * F_{ins} + B * F_P$) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht A für erreichte Fitness insgesamt F_{ins} (in $F = A * F_{ins} + B * F_P$).

76.1.11 getFaktorPositivFitnessPart

Syntax: `double getFaktorPositivFitnessPart() const`

Diese Methode gibt das Gewicht B für erreichte Verbesserungsfitness insgesamt F_P (in $F = A * F_{ins} + B * F_P$) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht B für erreichte Verbesserungsfitness insgesamt F_P (in $F = A * F_{ins} + B * F_P$).

77 Auswählen von Operatoren mit cChoosOperator

Die Klasse `cChoosOperator` stellt eine Methode bereit, um ein Operator auszuwählen. Bei der Auswahl die Operatoren werden gute Operatoren bevorzugt.

Alle Klassen zum Auswählen von Operatoren müssen von `cChoosOperator` abgeleitet werden.

Von der Klasse `cChoosOperator` werden die Operatoren mit der weel-selection Auswahlmethode ausgewählt. Dabei hat jeder Operator eine Auswahlwahrscheinlichkeit die proportional zu seiner Fitness ist und die Summe der Fitness aller Operatoren steht für die Wahrscheinlichkeit 1 .

77.1 Schnittstellenbeschreibung

77.1.1 cChoosOperator

Syntax: `cChoosOperator()`

Der Konstruktor für ein Objekt zum Auswählen von Operatoren.

Eingabeparameter: keine

Rückgabe: keine

77.1.2 choosOperator

Syntax: `cOperation *choosOperator()`

Diese Methode gibt eine Zeiger auf eine Operation eines ausgewählten Operators zurück.

Von der Klasse `cChoosOperator` werden die Operatoren mit der weel-selection Auswahlmethode ausgewählt. Dabei hat jeder Operator eine Auswahlwahrscheinlichkeit die proportional zu seiner Fitness ist und die Summe der Fitness aller Operatoren steht für die Wahrscheinlichkeit 1 .

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus. Insbesondere den verwendeten Allgorithmus zum Bewerten von Operatoren (gehören zur Klasse `cOperatorFitnessAlgorithm`).

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeiger auf eine Operation eines ausgewählten Operators.

77.1.3 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cChoosOperator" des Objects zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cChoosOperator" des Objects.

78 Die Operatoren cOperation

Alle Operatoren werden von der Basisklasse `cOperation` abgeleitet. Von der Klasse `cOperation` selbst können keine Instanzen erzeugt werden.

Operatoren werden vom Algorithmus gestartet, um neue bessere Individuen zu erzeugen. Ob und wieviele neue Individuen ein Operation erzeugt und welche Informationen sie dafür benutzt bleibt der Operation überlassen. Wenn eine Operation kein Individuum erzeugt, wird für sie vom Kernalgorithmus ein Dummyindividuum eingefügt, um die Operation bewerten zu können. Dieses Dummyindividuum besteht nur aus Individueninformationen `cIndividualInfo` über ein totes Individuum. Die Fitness dieses Dummyindividuums ist die minimal mögliche Fitness.

Jeder Operation muss ein eindeutiger Identifier `cOperationIdentifier` (siehe Abschnitt 78.2 auf Seite 430) bei der Erzeugung zugeordnet werden. Jeder Operatorklasse darf nur einen Konstruktor haben dem als einziger Parameter der Operationsidentifier `cOperationIdentifier` übergeben wird. Auf diese Art können die Operationsobjekte/-instanzen zu den Operatoren immer auf die gleiche Weise (von einer Factory) erzeugt werden.

Wenn für Operationen Funktionalität verwendet wird, die auch für andere Operatoren interessant sein könnte, ist diese Funktionalität in seperate Module oder Klassen auszulagern. Die Operatoren sollten nur zum Verwalten der Operation dienen und nicht zum Verarbeiten von weiteren Informationen. Wenn beispielsweise eine Operation auf Bildern gleichfarbige Flächen sucht und zu kompakten Fib-Objekten zusammenfasst. So sollte der Quellkode aus der von `cOperation` abgeleiteten Klasse für die Operation nicht viel mehr tun, als Funktionen aus anderen Modulen aufzurufen, welche die gleichfarbigen Flächen suchen und diese zusammenfassen.

Über die Konstante `OPERATION_DOMAIN` kann die Art der Individuen ermittelt werden, auf denen die Operation arbeitet. Dies ist der Klassenname der Individuen (z. B. `cIndividual`) auf denen die Operation arbeitet.

Neu Operatoren sollten leicht hinzugefügt werden können.

78.1 Schnittstellenbeschreibung

78.1.1 cOperation

Syntax: `protected cOperation( const cOperationIdentifier & operationId )`

Dieser Konstruktor erstellt die Operation.

Eingabeparameter:

- `operationId`: Der Operationsidentifier, welcher der Operation zugeordnet werden soll.

Rückgabe: keine

78.1.2 createInstance

Syntax: `cOperation * createInstance( const cOperationIdentifier & operationId ) const`

Diese Methode gibt eine Instanz auf eine neue Operation vom gleichen Typ wie die aktuelle Operation (von der sie aufgerufen wird) zurück.

Sie arbeitet ähnlich wie eine `clone()` Methode, nur das die Objektwerte (Klassenvariablen) auf ihre Initialbelegung gesetzt werden. Mit dieser Methode können von den zurückgegebene Operatoren, der Methode `getPossibleOperators()` einer `cOperatorFitnessAlgorithm` Instanz (siehe Abschnitt 75.1.8 auf Seite 419), Operatoren zum Ausführen erzeugt werden.

Eingabeparameter:

- `operationId`: Der Operationsidentifier, welcher der neuen Operation zugeordnet werden soll.

Rückgabe: Zurückgegeben wird eine Instanz auf eine neue Operation vom gleichen Typ wie die aktuelle Operation.

78.1.3 start

Syntax: `bool start()`

Diese Methode startet den Operator bzw. leitet die Operation ein und kehrt zurück, ohne deren Beendigung abzuwarten.

Eingabeparameter: keine

Rückgabe: Wenn der Operator gestartet wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.1.4 run

Syntax: `bool run()`

Diese Methode führt den Operator bzw. die Operation aus. Sie wartet auf die Beendigung der Operation.

Eingabeparameter: keine

Rückgabe: Wenn der Operator erfolgreich ausgeführt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.1.5 stop

Syntax: `bool stop()`

Diese Methode beendet die Operation und kehrt danach zurück.

Eingabeparameter: keine

Rückgabe: Wenn der Operator gestopt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.1.6 isRunning

Syntax: `bool isRunning() const`

Diese Methode prüft, ob die Operation gerade läuft, bzw. erfolgreich gestartet wurde aber noch nicht beendet.

Eingabeparameter: keine

Rückgabe: Wenn der Operator läuft, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.1.7 hasRun

Syntax: `bool hasRun() const`

Diese Methode prüft, ob die Operation schon gelaufen ist, bzw. erfolgreich gestartet wurde und dann beendet wurde.

Eingabeparameter: keine

Rückgabe: Wenn der Operator fertig gelaufen ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.1.8 getName

Syntax: `string getName() const`

Diese Methode gibt den Namen der Operation zurück. Dies ist der Klassenname der Operation.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Namen der Operation.

78.1.9 getOperationIdentifier

Syntax: `const cOperationIdentifier *getOperationIdentifier() const`

Diese Methode gibt den Identifier der Operation zurück oder der Nullpointer `NULL`, wenn die Operation noch kein Identifier hat.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Identifier der Operation oder der Nullpointer `NULL`, wenn die Operation noch kein Identifier hat.

78.1.10 OPERATION_DOMAIN

Syntax: `const string OPERATION_DOMAIN`

Diese Konstante gibt die Art der Individuen an auf denen der Operator arbeitet. Dies ist der Klassenname der Individuen auf denen der Operator arbeitet. Diese Angabe sollte zu der Angabe des Kernalgorithmus passen.

78.2 Der Identifier einer Operation cOperationIdentifier

Die Klasse `cOperationIdentifier` stellt ein Identifier für einen Lauf eines Operators bereit. Dies ist eine eindeutige natürliche Zahl für die Operation und eine natürliche Zahl für die genetischen Algorithmusinstanz in dem er gestartet wurde. Die genetische Algorithmusinstanz benötigt eine seperaten Identifier, da es vorgesehen ist, dass Individuen zwischen genetische Algorithmusinstanzen migrieren können und mit ihnen auch die Informationen der erzeugenden Operation migriert wird.

Einen Identifier für einer Operation sollte es nur maximal einmal geben. Weder auf anderen Rechnern noch für andere Algorithmusläufe sollten Identifier doppelt erzeugt werden.

78.2.1 Schnittstellenbeschreibung

78.2.2 cOperationIdentifier

Syntax:
```
cOperationIdentifier(
 unsigned long ulAlgorithmusIdentifier )
```

Dieser Konstruktor erstellt einen eindeutigen Identifierobjekt für eine Operation.

Die Methode holt sich selbständig alle benötigten Daten vom Kernalgorithmus.

Eingabeparameter:

- `ulAlgorithmusIdentifier`: Der Identifier des Algorithmus.

Rückgabe: keine

78.2.3 Vergleiche von cOperationIdentifier Objekten

Syntax: `bool equal(`
` const cOperationIdentifier &identifier ) const`
Syntax: `bool operation==(`
` const cOperationIdentifier &identifier ) const`

Diese Methoden vergleichen zwei `cOperationIdentifier` Objekte. Wenn das übergebene Objekt gleich dem aktuellen `cOperationIdentifier` Objekt ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

Eingabeparameter: keine

Rückgabe: Wenn das übergebene `cOperationIdentifier` Objekt gleich dem aktuellen `cOperationIdentifier` Objekt ist, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

78.3 Der Initialsierungsoperator cInitOperator

Der Initialsierungsoperator `cInitOperator` wird immer beim Start des Algorithmus aufgerufen. Er sollte je nach Multimediatyp Funktionen oder andere Operatoren aufrufen, welche beim Start des Algorithmus (meist) gute Individuen erzeugen.

Beispielsweise kann es sinnvoll sein beim Start des Algorithmus alle Bereiche mit gleichen Eigenschaften auf mehreren Punktelementen mittels Bereichs- und Funktionselemente jeweils zusammenzufassen. Wenn ein Bereich im Fib-Objekt vorher durch viele Punktelemente mit gleicher Eigenschaft realisiert wurde, gibt es für ihn dann im neuen Fib-Objekt nur ein Punktelement, dass mittels Variablen aus Bereichs- und Funktionselementen den gesamten Bereich abdeckt.

Teil IX
Erweiterungen des genetische Algorithmus für Fib

In diesem Teil werden die für Fib nötigen Erweiterungen für den genetische Algorithmus vorgestellt. Dazu gehört die Klassenhierarchien und die Schnittstellenbeschreibungen. Die Schnittstellenbeschreibungen geschieht in pseudo C++.

Die zu Fib zugehörigen Erweiterungen des genetische Algorithmus werden dem separaten Namensraum mit dem Namen “enviroment.fib” zugeordnet.

79 Hilfsmethoden

Hilfsmethoden die bei der Implementierung Hilfreich sind, aber nicht zur eigentlichen Schnittstelle gehören, werden nach Bedarf implentiert, hier aber nicht aufgeführt. Sie sollten nach Möglichkeit von Außen nicht zugreifbar sein (also `privat` oder `protected` deklariert sein).

80 Individuen für Fib-Objekte cFibIndividual

Elternklasse: `cIndividual`

Die Klasse `cFibIndividual` stellt ein Fib-Individuum für ein Fib-Objekt bereit. Ihre Elterklasse ist `cIndividual`, von der sie alle Methoden übernimmt. Die Klasse `cFibIndividual` dient hauptsächlich der Typsicherheit.

80.1 Schnittstellenbeschreibung

80.1.1 cFibIndividual

Syntax: `cFibIndividual( cRoot * pFibObject, cIndividualInfo * individualInfo )`

Der Konstruktor für ein Fib-Indiviuum, er erzeugt ein Fib-Individuum.

Weder von `pFibObject` noch `individualInfo` werden Kopien gemacht, beide Objekte werden direkt für das Fib-Individuum verwendet.

Eingabeparameter:

- `pFibObject`: Einen Zeiger auf das Fib-Objekt, welches das Individuum repräsentieren soll.
- `cIndividualInfo`: Einen Zeiger auf die Zusatzinformationen für das Fib-Objekt.

Rückgabe: keine

80.1.2 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cFibIndividual" des Inividuumobjekts zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cFibIndividual" des Fitnessobjekts.

81 Benchmark

Für den genetischen Algorithmus auf Fib-Objekte wird ein spezieller Benchmark für Fib-Objekte ausgeführt.

Um den Benchmarkwert zu ermitteln, wird auf einem festen Fib-Objekt/ Individuum eine Reihe von deterministischen Operationen ausgeführt und die dafür verbrauchte Prozessorzeit ("CPU Time") gemessen. Diese Prozessorzeit ist dann der Benchmarkwert mit dem die Prozessorzeit ("CPU Time") von Operationen multipliziert wird, um ihren Aufwand zu ermitteln.

82 Die Fitness von Fib-Objekten cFibObjectFitness

Elternklasse: `cObjectFitness`

Die Klasse `cFibObjectFitness` stellt die Fitness eines Fib-Objekts dar und wird von der Klasse `cObjectFitness` abgeleitet. Sie ist die Basisklasse für alle Klassen für die Fitness von Fib Objekten. Die Klasse `cFibObjectFitness` stellt selbst aber keine neuen Methoden bereit.

Von der Klasse `cFibObjectFitness` selbst oder einer abgeleiteten Klasse kann keine Instanz direkt erzeugt werden. Instanzen von Objekten der Klasse `cFibObjectFitness` können nur durch die Berechnung der Fitness mit den Klassen für die Algorithmen `cFibObjectFitnessAlgorithm` (siehe Abschnitt 84 auf Seite 440) oder durch das Laden einer existierenden Fitness aus einem Datenstrom erzeugt werden.

82.1 Schnittstellenbeschreibung

82.1.1 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cFibObjectFitness" des Fitnessobjekts zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Klassennamen "cFibObjectFitness" des Fitnessobjekts.

83 Die einfache Fitness von Individuen cFibObjectFitnessBasic

Elternklasse: `cObjectFitnessBasic`

Die Klasse `cFibObjectFitnessBasic` stellt eine einfach berechnete Fitness für Individuen dar. Sie wird erzeugt durch die Bewerteralgorithmenklasse `cFibObjectFitnessAlgorit` von Abschnitt 85 auf Seite 442. Ihre Elternklasse ist `cFibObjectFitness`.

Zu den Fitnessinformationen gehören:

- die gesamt Fitness des Fib-Objekts
- ein Wert für die Abweichungen vom original Multimediaobjekt
- ein Wert für die Größe des Fib-Objekts (die Anzahl der Bits, welche das Fib-Objekts beim Abspeichern im komprimierten Format benötigt)
- ein Wert für die benötige Auswertungszeit des Fib-Objekts

Die Gesamtfitness ergibt sich aus der Summe der gewichteten Einzelfitnesse: $Fitness = lDistanceToOriginal * dWeightDistanceToOriginal + lSize * dWeightSize + lEvaluationTime * dWeightEvaluationTime$

83.1 Schnittstellenbeschreibung

83.1.1 cFibObjectFitnessBasic

Syntax:
```
cFibObjectFitnessBasic( double lDistanceToOriginal,
 long lSize, long lEvaluationTime,
 double dWeightDistanceToOriginal=1.0,
 double dWeightSize=1.0,
 double dWeightEvaluationTime=1.0 ,
 cFibObjectFitnessBasicAlgorithm *
 objectFitnessAlgorithm=NULL )
```

Der Konstruktor für ein Fitnessobjekt für ein Fib-Individuum.

Die Gesamtfitness ist die Summe der gewichteten Einzelfitnesse: $Fitness = lDistanceToOriginal * dWeightDistanceToOriginal + lSize * dWeightSize + lEvaluationTime * dWeightEvaluationTime$

Eingabeparameter:

- `lDistanceToOriginal`: Der Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt.
- `lSize`: Die Größe des Fib-Objekt des Fib-Individuums.
- `lEvaluationTime`: Ein Wert für die Auswertungszeit des Fib-Objekt des Fib-Individuums.
- `dWeightDistanceToOriginal`: Das Gewicht mit dem der Abstand zum Multimediaobjekt `lDistanceToOriginal` in die gesamt Fitness eingeht.
- `dWeightSize`: Das Gewicht mit dem die Größe `lSize` in die gesamt Fitness eingeht.
- `dWeightEvaluationTime`: Das Gewicht mit dem die Auswertungszeit `lEvaluationTime` in die gesamt Fitness eingeht.
- `objectFitnessAlgorithm`: Dies ist ein Zeiger auf das Algorithmusobjekt der Klasse`cFibObjectFitnessBasicAlgorithm` Algorithmusobjekt mit dem die Fitness erzeugt wurde. Standardwert ist der Nullpointer `NULL`, um anzuzeigen, dass kein Algorithmusobjekt diese Fitness erzeugt hat.

Rückgabe: keine

83.1.2 cFibObjectFitnessBasic

Syntax: `cFibObjectFitnessBasic(`
` const cFibObjectFitnessBasic & fibObjectBasic )`

Der Kopierkonstruktor für ein Fitnessobjekt für ein Fib-Individuum. Er erzeugt ein neues `cFibObjectFitnessBasic` Fitnessobjekt mit den Werten des übergebenen Fitnessobjekts `fibObjectBasic`.

Eingabeparameter:

- `fibObjectBasic`: Eine referenz asuf das Fitnessobjekt, dessen Werte das erzeugte Fitnessobjekt haben soll.

Rückgabe: keine

83.1.3 getFitness

Syntax: `double getFitness() const`

Diese Funktion gibt die Gesamtfitness des zugehörigen Fib-Individuums zurück.

Die Gesamtfitness ist die Summe der gewichteten Einzelfitnesse: $Fitness = -1*(lDistanceToOriginal*dWeightDistanceToOriginal+lSize*dWeightSize+lEvaluationTime*dWeightEvaluationTime)$

Eingabeparameter: keine

Rückgabe: Die Gesamtfitness des zugehörigen Fib-Individuums.

83.1.4 getClassName

Syntax: `string getClassName() const`

Diese Funktion gibt den Klassennamen "cFibObjectFitnessBasic“ des Fitnessobjekts zurück.

In abgeleiteten Klassen sollte diese Methode immer überschrieben werden, so dass der jeweilige Klassennahmen zurückgegeben wird.

Eingabeparameter: keine

Rückgabe: Den Klassennamen "cFibObjectFitnessBasic“ des Fitnessobjekts.

83.1.5 getFitnessAlgorithmus

Syntax: `const cObjectFitnessAlgorithm * getFitnessAlgorithmus() const`

Diese Funktion gibt einen Zeiger auf das Algorithmusobjekt (siehe Abschnitt 85 auf Seite 442) zurück, mit dem das Fitnessobjekte erzeugt wurde. Für Fitnessobjekte vom Typ `cFibObjectFitnessBasic` wird eine Referenz auf das Algorithmusobjekt vom Typ `cFibObjectFitnessBasicAlgorithm` zurückgegeben.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf das Algorithmusobjekt vom Typ `cFibObjectFitnessBasicAlgorithm`.

83.1.6 getDifferenceToOriginal

Syntax: `double getDifferenceToOriginal() const`

Diese Methode gibt den Abstand des Multimediaobjekts, welches vom zugehörigem Fib-Individuum realisiert wird, zum originalen Multimediaobjekt zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird der Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt.

83.1.7 changeDifferenceToOriginal

Syntax: `bool changeDifferenceToOriginal(`
` long lDeltaToOriginal )`

Diese Methode ändert den Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt um den gegebenen Betrag `lDeltaToOriginal`.

Eingabeparameter:

- `lDeltaToOriginal`: Der Betrag, um den der Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt geändert wurde.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn der Abstand des Multimediaobjekt zum originalen Multimediaobjekt um `lDeltaToOriginal` geändert wurde, sonst `false` (=falsch).

83.1.8 getWeightDistanceToOriginal

Syntax: `double getWeightDistanceToOriginal() const`

Diese Methode gibt das Gewicht für den Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für den Abstand des Multimediaobjekts, welches vom Fib-Individuum realisiert wird, zum originalen Multimediaobjekt.

83.1.9 getSize

Syntax: `long getSize() const`

Diese Methode gibt die Größe des Fib-Objekt des Individuums zurück. Die Größe ist die Anzahl der Bits, welche das Fib-Objekts beim Abspeichern im komprimierten Format benötigt.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird die Größe des Fib-Objekts.

83.1.10 changeSize

Syntax: `bool changeSize( long lDeltaSize )`

Diese Methode ändert die vom Fitnessobjekt vermerkte Größe des Fib-Objekt des Individuums um den gegebenen Betrag `lDeltaSize`.

Diese Methode ist beispielsweise dann sinnvoll, wenn in einer Kopie eines Individuums mit korrekter Fitness nur ein kleiner Teil (z. B. ein Listenunterobjekt) des Fib-Objekts geändert wurde und die Größe nicht komplett neu berechnet werden soll.

Eingabeparameter:

- `lDeltaSize`: Der Betrag, um den die Größe des Fib-Objekts geändert wurde.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die gemerkte Größe des Fib-Objekts um `lDeltaSize` geändert wurde, sonst `false` (=falsch).

83.1.11 getWeightSize

Syntax: `double getWeightSize() const`

Diese Methode gibt das Gewicht für die Größe des Fib-Objekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für die Größe des Fib-Objekts.

83.1.12 getTime

Syntax: `double getTime() const`

Diese Methode gibt ein Wert für die benötige Auswertungszeit des Fib-Objekts des Individuums zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Wert für die benötige Auswertungszeit des Fib-Objekts.

83.1.13 changeTime

Syntax: `bool changeTime( long lDeltaEvalueTime )`

Diese Methode ändert den von dem Fitnessobjekt gemerkten Wert für die Auswertungszeit um den gegebenen Betrag `lDeltaEvalueTime`.

Diese Methode ist beispielsweise dann sinnvoll, wenn in einer Kopie eines Individuums mit korrekter Fitness nur ein kleiner Teil (z. B. ein Unterobjekt) des Fib-Objekts geändert wurde und die Auswertungszeit nicht komplett neu berechnet werden soll.

Eingabeparameter:

- `lDeltaEvalueTime`: Der Betrag, um den die Auswertungszeit des Fib-Objekts geändert wurde.

Rückgabe: Es wird `true` (=wahr) zurückgegeben, wenn die Auswertungszeit des Fib-Objekts um `lDeltaEvalueTime` geändert wurde, sonst `false` (=falsch).

83.1.14 getWeightEvaluationTime

Syntax: `double getWeightEvaluationTime() const`

Diese Methode gibt das Gewicht für die Auswertungszeit des Fib-Objekts zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für die Auswertungszeit des Fib-Objekts.

84 Algorithmus für einen Bewerter von Fib-Individuen cFibObjectFitnessAlgorithm

Elternklasse: `cObjectFitnessAlgorithm`

Die Klasse `cFibObjectFitnessAlgorithm` ist die Basisklasse der Bewerter zur Erzeugung von Fitnessobjekten für Fib-Individuen. Von der Klasse `cFibObjectFitnessAlgorithm` selbst kann keine Instanz erzeugt werden. Sie wird von der Basisklasse `cObjectFitnessAlgorithm` abgeleitet.

84.1 Schnittstellenbeschreibung

84.1.1 evalueFitness

Syntax: `cFibObjectFitness * evalueFitness( const cFibIndividual &individual )`

Diese Methode berechnet die Fitness eines Fib-Individuums `individual`.

Eingabeparameter:

- `individual`: Eine Referenz auf das Fib-Individuum, welches bewertet werden soll.

Rückgabe: Zurückgegeben wird ein Zeiger auf ein Fitnessobjekt für die Fitness des Fib-Individuums `individual`, berechnet bezüglich des gesetzten Originalindividuums.

84.1.2 setOriginalIndividual

Syntax: `bool setOriginalIndividual( const cFibIndividual &original )`

Diese Methode wird das Fib-Originalindividuum gesetzt. Bezüglich dieses Fib-Individuums werden andere Fib-Individuen vom Bewerteralgorithmus bewertet.

Eingabeparameter:

- `original`: Dies ist eine Referenz auf das Fib-Individuum, welches als Fib-Originalindividuum gesetzt werden soll.

Rückgabe: Wenn das übergebene Fib-Individuum als Originalindividuum gesetzt wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

84.1.3 getOriginalIndividual

Syntax: `cFibIndividual * getOriginalIndividual()`

Diese Methode gibt eine Referenz auf das Fib-Originalindividuum zurück. Bezüglich dieses Fib-Individuums werden andere Fib-Individuen vom Bewerteralgorithmus bewertet.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf das Fib-Originalindividuum.

84.1.4 getOriginalIndividualRoot

Syntax: `cRoot * getOriginalIndividualRoot()`

Diese Methode liefert nur die root-Elemente des original Individuums/ Multimediaobjekts zurück. Die Haupt-Fib-Objekte der root-Elemente sind jeweils leere Punkte (Punkte ohne Positionsvektor). Dies dient zur Ermittlung des Multimediaformats auf das der Algorithmus arbeitet, ohne das die gesamten Daten des original Individuums/ Multimediaobjekts übermittelt werden müßen.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Referenz auf die root-Elemente des original Individuums/ Multimediaobjekts.

84.1.5 getClassName

Syntax: `string getClassName()`

Diese Methode gibt eine Zeichenkette "cFibObjectFitnessAlgorithm“ mit dem Klassennamen des aktuellen Bewerters für Individuen zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette "cFibObjectFitnessAlgorithm“.

84.1.6 getBestFitness

Syntax: `const cFibObjectFitness * getBestFitness() const`

Diese Methode gibt die best mögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass es nicht möglich ist eine bessere Fitness zu generieren.

Diese Fitness entspricht im Normalfall einem Individuum, welches das Originalobjekt perfekt wiedergibt und dabei keine Recourcen verbraucht.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf die best mögliche Fitness eines Individuums bezüglich des Originalobjekts oder der Nullzeiger `NULL`, wenn diese nicht berechnet werden kann.

84.1.7 getWorstCaseFitness

Syntax: `const cFibObjectFitness * getWorstCaseFitness() const`

Diese Methode gibt die schlechtest mögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass sie immer erreicht werden kann.

Im Normalfall wird der Algorithmus die Fitness des Originalindividuums zurückgeben, da dieses ja schon in der entsprechenden Repräsentation vorliegt und verbessert werden soll.

Der Allgorithmus kann dennoch durch `evalueFitness()` Fitnessobjekt generieren die noch schlechter sind.

Es wird der Nullzeiger `NULL` zurückgegeben, wenn die schlechtest mögliche Fitness nicht berechnet werden kann. Dies kann beispielsweise der Fall sein, wenn kein Originalobjekt für den Algorithmus gesetzt wurde.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf die schlechtest mögliche Fitness eines Individuums bezüglich des Originalobjekts oder der Nullzeiger `NULL`, wenn diese nicht berechnet werden kann.

85 Einfacher Algorithmus für einen Bewerter von Fib-Individuen cFibObjectFitnessBasicAlgorithm

Elternklasse: `cFibObjectFitnessAlgorithm`

Die Klasse `cFibObjectFitnessBasicAlgorithm` dient zum Berechnen der `cFibObjectFitnessBasic` Fitness (siehe Abschnitt 83 auf Seite 434) eines Individuums. Sie wird von der Klasse `cFibObjectFitnessAlgorithm` abgeleitet.

Zu den Fitnessinformationen der vom Algorithmus erzeugten Fitnessobjekte `cFibObjectFitnessBasic` (siehe Abschnitt 83 auf Seite 434) gehören:

- die gesamt Fitness des Fib-Objekts

- ein Wert für die Abweichungen vom original Multimediaobjekt
- ein Wert für die Größe des Fib-Objekts (die Anzahl der Bits, welche das Fib-Objekts beim Abspeichern im komprimierten Format benötigt)
- ein Wert für die benötige Auswertungszeit des Fib-Objekts

Die Gesamtfitness ergibt sich aus der Summe der gewichteten Einzelfitnesse: $Fitness = lDistanceToOriginal * dWeightDistanceToOriginal + lSize * dWeightSize + lEvaluationTime * dWeightEvaluationTime$

Diese Rechnung wird allerdings nicht direkt von den Instanzen der Klasse `cFibObjectFitnessBasicAlgorithm` vorgenommen, sondern direkt von den erzeugten Fitnessobjekten `cFibObjectFitnessBasic` (siehe Abschnitt 83 auf Seite 434).

85.1 Schnittstellenbeschreibung

85.1.1 cFibObjectFitnessBasicAlgorithm

Syntax:
```
cFibObjectFitnessBasicAlgorithm( cFibElement * pOriginalFibElement,
 double dWeightDistanceToOriginal=1.0,
 double dWeightSize=1.0,
 double dWeightEvaluationTime=1.0 )
```

Der Konstruktor für ein Objekt zum Bewerter von Fib-Individuen.

Eingabeparameter:

- `pOriginalFibElement`: Eine Referenz auf das Fib-Element, welches als Originalindividuum gesetzt werden soll.
- `dWeightDistanceToOriginal`: Das Gewicht mit dem der Abstand zum Multimediaobjekt `lDistanceToOriginal` in die gesamt Fitness eingeht.
- `dWeightSize`: Das Gewicht, mit dem die Größe `lSize` in die gesamt Fitness eingeht.
- `dWeightEvaluationTime`: Das Gewicht, mit dem die Auswertungszeit `lEvaluationTime` in die gesamt Fitness eingeht.

Rückgabe: keine

85.1.2 evalueFitness

Syntax: `cFibObjectFitnessBasic evalueFitness( const cFibIndividual &individual )`

Diese Methode berechnet die Fitness eines Fib-Individuums `individual`.

Eingabeparameter:

- `individual`: Eine Referenz auf das Fib-Individuum, welches bewertet werden soll.

Rückgabe: Zurückgegeben wird ein Fitnessobjekt für die ermittelte Fitness des Fib-Individuums `individual`, berechnet bezüglich des gesetzten Originalindividuums.

85.1.3 evalueDistance

Syntax: `double evalueDistance( const cFibElement &fibElement, const cFibElement *fibElementArea=NULL )`

Diese Methode berechnet den Abstand zum original Multimediaobjekt eines Fib-Objekts `fibElement` auf dem durch `fibElementArea` angegeben Bereich. Es wird also nur der Abstand von Eigenschaften von Punkten berechnet, welche durch `fibElementArea` erstellt werden. Wenn `fibElementArea` allerdings gleich `NULL` ist, werden alle Punkte des Fib-Objekts `fibElement` verglichen.

Eingabeparameter:

- `fibElement`: Eine Referenz auf das Fib-Objekt, welches bewertet werden soll.
- `fibElementArea`: Dieses Fib-Objekt bestimmt den zu überprüfenden Bereich. Nur der Bereich den dieses Individuum mit Punkten überdeckt ist zu prüfen. Wenn `fibElementArea` gleich dem Nullpointer `NULL` (Standardwert) ist, wird der Abstand des gesamte Bereichs bzw. zum gesamte original Multimediaobjekt ermittelt.

Rückgabe: Zurückgegeben wird der Abstand zum original Multimediaobjekt eines Fib-Objekts `fibElement` auf dem durch `fibElementArea` angegeben Bereich.

85.1.4 getClassName

Syntax: `string getClassName()`

Diese Methode gibt eine Zeichenkette "cFibObjectFitnessAlgorithm“ mit dem Klassennamen des aktuellen Bewerters für Individuen zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird eine Zeichenkette "cFibObjectFitnessBasicAlgorithm“.

85.1.5 getBestFitness

Syntax: `const cFibObjectFitnessBasic * getBestFitness() const`

Diese Methode gibt die best mögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass es nicht möglich ist eine bessere Fitness zu generieren.

Diese Fitness ist im Fall von `cFibObjectFitnessBasicAlgorithm` einem `cFibObjectFitnessBasic` Object dessen (Teil-)Fitnesswerte alle 0 sind.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger ein `cFibObjectFitnessBasic` Object dessen (Teil-)Fitnesswerte alle 0 sind.

85.1.6 getWorstCaseFitness

Syntax: `const cFibObjectFitnessBasic * getWorstCaseFitness() const`

Diese Methode gibt die schlechtest mögliche Fitness eines Individuums bezüglich des Originalobjekts zurück. Diese Fitness zeichnet aus, dass sie immer erreicht werden kann.

Diese Fitness ist im Fall von `cFibObjectFitnessBasicAlgorithm` die Fitness des Originalindividuums, da dieses ja schon in der entsprechenden Repräsentation vorliegt und verbessert werden soll.

Der Allgorithmus kann dennoch durch `evalueFitness()` Fitnessobjekt generieren die noch schlechter sind.

Es wird der Nullzeiger `NULL` zurückgegeben, wenn die schlechtest mögliche Fitness nicht berechnet werden kann. Dies ist der Fall sein, wenn kein Originalobjekt für den Algorithmus gesetzt wurde.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird ein Zeiger auf die Fitness des Originalindividuums, wenn dies vorhanden ist, oder der Nullzeiger `NULL`, wenn kein Originalindividuum gesetzt wurde.

85.1.7 getWeightDistanceToOriginal

Syntax: `double getWeightDistanceToOriginal() const`

Diese Methode gibt das Gewicht für den Abstand zum Original ($dWeightDistanceToOriginal$) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für den Abstand zum Original ($dWeightDistanceToOriginal$)

85.1.8 getWeightSize

Syntax: `double getWeightSize() const`

Diese Methode gibt das Gewicht für die Größe des Fib-Objekts ($dWeightSize$) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für die Größe des Fib-Objekts ($dWeightSize$).

85.1.9 getWeightEvaluationTime

Syntax: `double getWeightEvaluationTime() const`

Diese Methode gibt das Gewicht für die Auswertungszeit des Fib-Objekts ($dWeightEvaluationTime$) zurück.

Eingabeparameter: keine

Rückgabe: Zurückgegeben wird das Gewicht für die Auswertungszeit des Fib-Objekts ($dWeightEvaluationTime$).

86 Initialisieren des Algorithmus cInitFibEnviroment

Elternklasse: `cInitEnviroment`

Die Klasse `cInitFibEnviroment` ist die Basisklasse zur Initialisierung des Algorithmus für Fib-Individuen. Von ihr werden alle Klassen zur Initialisierung des Algorithmus für Fib-Individuen abgeleitet.

Für die Initialisierung steht die Methode `initEnviroment()` bereit.

Die Initialisierung umfasst:

- Die Bereitstellung des Originalindividuums/ -multimediaobjekts.
- Erzeugen und Einfügen mindestens eines Fib-Individuums für und in die Population. So hat der Algorithmus von Anfang an ein Fib-Individuum zum zurückgeben. Diese Fib-Individuen können auch einem früheren Lauf des Algorithmus kommen und/oder aus einer Datei geladen werden.
- Aufrufen, wenn vorhanden, des Initialsierungsoperators `cInitOperator`.

86.1 Schnittstellenbeschreibung

86.1.1 cInitFibEnviroment

Syntax: `cInitFibEnviroment( const cRoot &fibOriginal )`

Der Konstruktor für die Fib-Initialisierungsobjekt des Algorithmus (`cEnviroment` siehe Abschnitt 67 auf Seite 393).

Eingabeparameter:

- `fibOriginal`: Eine Referenz auf das original Multimediaobjekt.

Rückgabe: keine

86.1.2 initEnviroment

Syntax: `bool initEnviroment()`

Diese Methode Initialisiert den Algorithmus mit Fib-Objekten.

Es wird geprüft, ob die Initialisierung zur aktuellen Instanz von `cEnviroment` passt. Dazu gehört beispielsweise, dass die aktuellen `cEnviroment` Instanz auf Fib-Individuen arbeitet.

Bei der Initialisierung wird das dem Konstruktor übergebene Fib-Objekt als original Individuum eingesetzt und als erstes Individuum in den Kernallgorithmus (mit der `insertIndividual()`-Methode) eingefügt.

Aufrufen, wenn vorhanden, des Initialsierungsoperators `cFibInitOperator`.

Eingabeparameter: keine

Rückgabe: Wenn der Algorithmus `cEnviroment` initialisiert wurde, wird `true` (=wahr) zurückgegeben, sonst `false` (=falsch).

87 Die Operatoren für Fib cOperationFib

Elternklasse: `cOperation`

Alle Operatoren für Fib-Individuen werden von der Basisklasse `cOperationFib` abgeleitet, welche wiederum von der Basisklasse aller Operatoren `cOperation` abgeleitet ist. Von `cOperationFib` selbst können keine Instanzen erzeugt werden. Die Klasse `cOperationFib` übernimmt alle Methoden von ihrer Elternklasse `cOperation` und definiert keine eigenen Methoden.

Operatoren werden vom Algorithmus gestartet, um neue bessere Fib-Individuen zu erzeugen. Ob und wieviele neue Individuen eine Operation erzeugt und welche Informationen sie dafür benutzt, bleibt der Operation überlassen.

Teil X
Fib-Datenbank

Die Fib-Datenbank bietet die Möglichkeit häufig verwendete Fib-Objekte vorzuhalten. Auf dort abgelegte Fib-Objekte kann von einem Fib-Multimediaobjekt zurückgegriffen werden, ohne dass diese mit dem Multimediaobjekt übertragen oder gespeichert werden müssen. Auf diese Weise kann nicht nur Übertragungsbandbreite gespaart werden, sondern mit dem Wissen über die festen Datenbankobjekte kann auch weitere Informationen aus dem Multimediaobjekt extrahiert werden.

Die Fib-Datenbank gehört zu keinem Fib-Multimediaobject. Sie wird mit den Fib-Bibliotheken /dem Fib-System ausgeliefert und enthält häufig verwendete Datenbankobjekte, die in Fib-Objekten verwendet werden können. Datenbankobjekte können beispielsweise Linien (mit den Eingabeparametern für Start- und Endpunkte), Rechtecke oder Kreise sein, aber auch Bäume, Autos, Zeichensätze (fonts) oder Fraktale. Wird in einem Fib-Objekt beispielsweise ein Kreis benötigt, kann dass entsprechende parametrisierte Datenbankobjekt verwendet werden.

Die Implementierung der Datenbankobjekte kann an die jeweilige Anwendungsumgebung angepasst sein. Soll zur Anzeige beispielsweise OpenGL verwendet werden, können Datenbankobjekte direkt mit OpenGL-Primitiven (z. B. Dreiecken) umgesetzt werden. Auf diese Weise kann mit Datenbankobjekten die Performance der Anwendung verbessert werden. Bei der Kodierung der Fib-Objekte kann auch gleich darauf geachtet werden, dass Datenbankobjekte mit guter Performance für die Zielanwendung verwendet werden. Diese Fib-Objekte sind dann immer noch auf allen Fib-Systemen mit genügend hoher Datenbankversion anzeigbar, auf einigen jedoch schneller.

Welche Objekte eine Datenbank enthält, sowie die Identifier für diese Objekte, sind mit der Datenbankversion festgelegt. Datenbanken mit höheren Datenbankversionen enthalten dabei alle Datenbankobjekte mit den gleichen Identifiern wie Datenbanken mit niedrigeren Datenbankversionen. Auf diese Weise ist gewährleistet, dass Fib-Objekte immer aufwärtskompatibel zu neueren Datenbankversionen sind.

Alle Identifier von Datenbankobjekten sind negativ.

88 Struktur

Die Fib-Datenbank ist in verschiedene Abschnitte strukturiert. Dabei sollten ähnliche Objekte auch nahe beieinander stehen und wahrscheinlich häufig verwendete Objekte eine möglichst große Identifier haben (also vom Absolutwert kleinen Identifier, da die Datenbankobjekidentifier alle negativ sind).

Die Beschreibung der einzelnen Bereiche (vorne stehen die Absolutwerte der Identifier (=Id), also 10 - 19 für Id -10 bis -19):

- 0 ein Objekt für nichts (leerer Punkt)
- 1 - 9 Punkte (Id = Anzahl der Dimensionen)
- 10 - 19 Beschreibungen (wichtigste Lizenzen)
- 20 - 999 2 dimensionale Objekte:
  - 20 - 29 Linien
  - 30 - 39 Dreiecke
  - 40 - 49 Vierecke
  - 50 - 59 Kreise
  - 60 - 89 offen
  - 90 - 99 Funktionen
  - 100 - 199 Beschreibungen 2 (z. B. Lizenzen)
  - 200 - 299 Linien 2
  - 300 - 399 Dreiecke 2
  - 400 - 499 Vierecke 2
  - 500 - 599 Kreise 2
  - 600 - 899 offen
  - 900 - 999 Funktionen 2
- 1000 - 1999 Algorithmen
- 2000 - 2999 Beschreibungen
- 3000 - 3999 3 dimensionale Objekte
- 4000 - 4999 4 dimensionale Objekte

Teil XI
Konvertierungsfunktionen

In diesem Teil werden Algorithmen beschrieben die zur Konvertierung von Multimediaobjekten in Fib-Objekte und von Fib-Objekte in Multimediaobjekten in anderen Multimediabeschreibungsformate/ -sprachen.

89 Konvertierung zu Fib-Multimediaobjekten

Bei der Konvertierung von Multimediaobjekte zu Fib-Multimediaobjekten geht es vorrangig darum, das original Multimediaobjekt mit der Fib-Multimediasprache korrekt darzustellen. Die Optimierung vom erzeugten Fib-Multimediaobjekt ist zweitrangig und kann in einem nachgeschalteten Schritt geschehen (z. B. durch den genetischen Algorithmus für Fib).

Diese Umsetzung in Fib wird im allgemeinen dadurch geschehen, dass die Punkte des original Multimediaobjekts mit ihren Eigenschaften zu ein Fib-Objekt zusammengefügt werden. Dabei ist darauf zu achten das die Eigenschaften der Punkte korrekt umgesetzt werden. Ebenen in Bildern haben beispielsweise nichts mit der Überdeckung von Teilbildern in Fib-Objekten zu tun, sondern sind Eigenschaften von Punkten.

89.1 Konvertierung von Rastergrafik in Fib-Multimediaobjekte

Rastergrafik ist dadurch gekenzeichnet, dass die Daten für das Bild in einer 2 Dimensionalen Matrix darstellbar ist, dessen Elemente Farbwerte sind.

Eine einfache Abbildung von Rastergrafik in die Fib-Multimediasprache kann mit dem im Listing 4 dargestellten Algorithmus geschehen (Pseudo-C).

Dabei beginnt die Indizierung der Matrix mit (0,0).
Der Farbwert der Koordinate (x,y) kann mit `matrix[ x, y]` bestimmt werden. Mit der Funktion `getColorVector()` wird aus dem Matrix Farbwert ein Fib-Farbvektor generiert.
Die Syntax der Fib-Objekte entspricht der im Abschnitt III auf Seite 22 angesprochenen möglichen Syntax.

Listing 4: Algorithmus zur Erzeugung eines korrekten Fib-Objekts aus einer Bildmatrik

```
void translate( matrix, farbschema ){

   int xmax = Anzahl der Spalten der Matrix;
   int ymax = Anzahl der Zeilen der Matrix;
   Fib_Objekt_Pointer obj = new root();

   //Eigenschaften der Rastergrafik setzen
```

```
    obj->addDomain( dim( 2, horizontal, vertikal ), vector( 2,
        naturalNumber( xmax ), naturalNumber( ymax ) ) );
    obj->addDomain( property( farbschema.type ), farbschema.domain
        );

    list hauptliste = new list();

    obj->setMainFibObject( hauptliste )

    for( int x = 0; x == xmax; x = x + 1 ){

        for( int y = 0; y == xmax; y = y + 1 ){
            hauptliste->insertObject( property( getColorVector(
                farbschema.type, matrix[x,y] ) , p( (x,y) ) ) );
        };
    };
};
```

Nach dieser Umsetzung kann das erzeugte Fib-Multimediaobjekt mit Operatoren, welche das Fib-Objekt vereinfachen, aber nicht die dargestellte Rastergrafik beeinflussen, weiter vereinfacht werden.

Eine sinvollerweise einzusetzende Operation, wäre beispielsweise das Zusammenfassen von gleichfarbigen Bereichen.

90 Konvertierung von Fib-Multimediaobjekten

Die in Fib-Multimediaobjekten kodierten Objekte müßen auch wieder aus der Fib-Darstellung erzeugt werden können, um sie beispielsweise Darzustellen.

Die allgemeine Vorgehensweise dafür ist, das Fib-Multimediaobjekt auszuwerten und die erzeugten Punkte in das Zielformat einzufügen. Dies ist natürlich nur möglich, wenn das Zielformat die Eigenschaften, welche Fib-Multimediaobjekt auftauchen, auch kodieren kann. Wenn nicht, gehen Informationen verlohren. Des weiteren kann es sinnvoll sein, nicht alle einzelnen Punkte in das Zielformat einzfügen, sondern ganze Teilobjekte (z. B. Rechtecke oder Kreise) aus dem Fib-Multimediaobjekt, wenn das Zielformat diese unterstützt.

90.1 Konvertierung von Fib-Multimediaobjekte in Rastergrafik

Bei der Konvertierung von Fib-Multimediaobjekten in Rastergrafik ist zuerst eine Matrix für die Punkte mit ihren Eigenschaften zu erzeugen. Die Matrix hält die Informationen der Rastergrafik. Die Größe der Matrix und die Art der Eigenschaften, welche sie in ihren Koordinaten aufnehemen kann, wird von den Informationen in den root-Elementen des zu konvertierenden Fib-Multimediaobjekts bestimmt. Dabei müssen allerdings alle Eigenschaften, welche nicht Farben oder/und Helligskeitwerte repräsentieren ignoriert werden. Das weiteren können natürlich Fib-Multimediaobjekte, welche nicht genau zwei Dimensionen besitzen, nicht direkt in Rastergrafik konvertiert werden.

Die erzeugte Matrix ist dann sinnvollerweise mit einem Standardwert/-eigenschaft zu initialisieren. Dann ist das Fib-Multimediaobjekt auszuwerten und für die erzeugten Punkte (inklusive der Hintergrundeigenschaften), an der entsprechenden Position in der Matrix, ihre Eigenschaft einzufügen.

Die erzeugte Matrix ist am Ende im gewünschten Rastergrafikformat abzuspeichen.

Teil XII
Vorgehensweisen für Fib

In diesem Teil werden für Fib einige Vorgehensweisen für bestimmte Aufgaben vorgestellt. Es kann als eine Art “Tips und Tricks” Sammlung verstanden werden.

91 Parallele Auswertung von Fib-Teilobjekten

Problem: Heutzutage sind Parallelrechner schon weit verbreitet. Um dies bei der Auswertung von Fib-Objekten auszunutzen, sind einige Anpassungen nötig.

Lösungskizze: Anstatt das gesamte Fib-Objekt mit in einem Proßess auszuwerten wird dieses in Teilobjekten aufgespalten und diese seperat ausgewertet. Dabei gibt es 2 Schritte: Ermittlung der Teilobjekte und Auswertung der ermittelten Teilobjekte.

Lösung mit getTimeNeed():

0 ermittle die Anzahl A der Teilobjekte die gleichzeitig ausgewerte werden sollen, dies Anzahl A kann beispielsweise mit der Anzahl Prozessorkerne übereinstimmen

1 Ermittlung der Anzahl N der benötigten Teilobjekte: Diese Zahl sollte ein Vielfaches der Anzahl A der Prozesse sein. Wenn beispielsweise 4 Prozesse gleichzeitig laufen können und ein Multiplikator von 10 festgelegt wurde, sollte $N = 10 * A = 40$ sein. In diesem Fall sollten also rund 40 Teilobjekte gesucht werden, die ungefähr die gleiche Abarbeitungszeit haben.

2 T ist die Liste der Teilobjekte. Die Liste T wird am Anfang mit dem obersten root-Element als Teilobjekt initialisiert. Zu jedem Teilobjekt gehört der Zeiger auf das definierende Unterobjekt, seine Nummer in der Ordnung der zusammenhängenden Teilobjekte (siehe Abschnitt 14.8 auf Seite 82) und (ein Wert für) die zur Auswertung benötigte Zeit (ermittelt mit `getTimeNeed()` für das jeweilige Teilobjekt).

3 Ermittlung der Teilobjekte:

 3.1 nehme und entferne das Teilobjekt T_m aus T, *welches den größten Wert für die zur Auswertung benötigte Zeit hat*

 3.2 ermittle das erste/ nächste Listenelement L in T_m; wenn es kein erste/ nächste Listenelement L in T_m gibt, füge T_m an das Ende von T_O an und, wenn T leer ist, gehe zu Schritt 4, sonst gehe zu Schritt 3.1

3.3 ermittle alle Unterobjekte des Listenelements L und füge jeweils für jedes ermittelte Unterobjekt das zusammenhängende Teilobjekt, das es definiert, und die Nummer des zusammenhängenden Teilobjekts in die Liste T an der Stelle an der vorher T_m stand ein

3.4 wenn die Anzahl der Teilobjekte in T größer oder gleich N ist gehe zu Schritt 4

3.5 wenn die Anzahl der Teilobjekte in T kleiner als N ist gehe zu Schritt 3.1

4 füge alle Elemente von T_O an ihrer entsprechenden Stelle von T ein

5 Starten der Auswertung der ermittelten Teilobjekte (i wird auf 1 gesetzt):

5.1 Wenn T leer ist gehe zu Schritt 6

5.2 Wenn weniger als A Teilobjekte zur Zeit ausgewertet werden, gehe Schritt 5.4

5.3 Warte kurz und gehe dann zu Schritt 5.2

5.4 nehme und entferne das erste Teilobjekt T_1 aus T

5.5 Starte die Auswertung des Teilobjekt T_1 und sammle die Ergebnispunkte in der Liste LP_i

5.5 erhöhe i um 1

5.6 gehe zu Schritt 5.1

6 warten bis alle Teilobjekte fertig ausgewertet sind

7 erstelle die Liste mit Ergebnispunkten $LP = \bigcup_{a=1}^{i-1} LP_a$, wobei die Elemente der Liste LP_{i+1} in der Ergebnisliste hinter denen der Liste LP_i stehen

8 zeige die Punkte in LP an

9 Ende

Lösung ohne getTimeNeed():

0 ermittle die Anzahl A der Teilobjekte die gleichzeitig ausgewerte werden sollen, dies Anzahl A kann beispielsweise mit der Anzahl Prozessorkerne übereinstimmen

1 Ermittlung der Anzahl N der benötigten Teilobjekte: Diese Zahl sollte ein Vielfaches der Anzahl A der Prozesse sein. Wenn beispielsweise 4 Prozesse gleichzeitig laufen können und ein Multiplikator von 10 festgelegt wurde, sollte $N = 10 * A = 40$ sein. In diesem Fall sollten also rund 40 Teilobjekte gesucht werden.

2 T ist die Liste der Teilobjekte. Die Liste T wird am Anfang mit dem obersten root-Element als Teilobjekt initialisiert. Zu jedem Teilobjekt gehört der Zeiger auf das definierende Unterobjekt und seine Nummer in der Ordnung der zusammenhängenden Teilobjekte (siehe Abschnitt 14.8 auf Seite 82).

3 Ermittlung der Teilobjekte:

3.1 nehme und entferne das i'te Teilobjekt T_i aus T

3.2 ermittle das erste/ nächste Listenelement L in T_i; wenn es kein erste/ nächste Listenelement L in T_i gibt, markiere T_i; wenn alle T_j markiert sind gehe zu Schritt 4, sonst gehe zu Schritt 3.1

3.3 ermittle alle Unterobjekte des Listenelements L und füge jeweils für jedes ermittelte Unterobjekt das zusammenhängenden Teilobjekte, das es definiert, und die Nummer des zusammenhängenden Teilobjekts in ihrer Reihenfolge an den entsprechenden Stellen (i+Anzahl bisher eingefügte Teilobjekte) in T ein

3.4 wenn die Anzahl der Teilobjekte in T größer oder gleich N ist gehe zu Schritt 4

3.5 wenn die Anzahl der Teilobjekte in T kleiner als N ist gehe zu Schritt 3.1

4 Starten der Auswertung der ermittelten Teilobjekte (i wird auf 1 gesetzt):

4.1 Wenn T leer ist gehe zu Schritt 5

4.2 Wenn weniger als A Teilobjekte zur Zeit ausgewertet werden, gehe Schritt 4.4

4.3 Warte kurz und gehe dann zu Schritt 4.2

4.4 nehme und entferne das erste Teilobjekt T_1 aus T

4.5 Starte die Auswertung des Teilobjekt T_1 und sammle die Ergebnispunkte in der Liste LP_i

4.5 erhöhe i um 1

4.6 gehe zu Schritt 4.1

5 warten bis alle Teilobjekte fertig ausgewertet sind

6 erstelle die Liste mit Ergebnispunkten $LP = \bigcup_{a=1}^{i-1} LP_a$, wobei die Elemente der Liste LP_{i+1} in der Ergebnisliste hinter denen der Liste LP_i stehen

7 zeige die Punkte in LP an

8 Ende

92 Auswertung von zeitlichen Fib-Objekten

Problem: Fib-Objekte können Multimediaobjekte repräsentieren die über eine Zeit hinweg "abgespielt" werden sollen. Ein Beispiel dafür sind Filme. Diese Objekte müssen nicht schon zu Begin vollständig ausgewertet werden, sondern es reich nur die Abschnitte auszuwerten, welche als nächstes abgespielt werden müssen.

Lösungsskizze 1: Über `periodBegin`, `periodEnd` und `evaluationTime` Eigenschaftselemente werden die Fib-Unterobjekte in Zeitintervallen eingeteilt. Und jeweils die nächsten anzuzeigenden Unterobjekte ausgewertet. Die Unterteilung geschieht dabei direkt in den Unterobjekten eines Listenelements, welches das Haup-Fib-Objekt des obersten root-Elements ist.

Lösungsskizze 2: Das Haupt-Fib-Objekt des Fib-Multimediaobjekt wird durch ein Kommentarelement, welches ein Bereichselement enthält, eingeleitet. Durch das Kommentarelement wird spezifiziert, dass das enthaltenede Bereichselement verschiedene aufeinanderfolgende Zeitsegmente erzeugt (z. B. könnte es lauten $m($ $nextElement :: function, “time'', Obj))$. Das enthaltende Bereichselement definiert eine Variable, deren verschiedene Belegung verschiedene aufeinanderfolgende Zeitabschnitte erzeugt. Bei der Auswertung kann die Variable des Bereichselements jeweils mit dem nächsten Wert belegt werden, um den nächsten Zeitabschnitt zu erzeugen. Spätere Belegungen der Variable, werden vorerst nicht benötigt, da sie spätere, noch nicht benötigte, Zeitabschnitte erzeugen.

93 Skalierung von Fib-Objekten

Problem: Da Fib-Objekt immer eine kleinste Einheit in den Dimensionsrichtungen haben, ist eine einfache Skalierung im allgemeinen nicht möglich. Es können aber Fib-Objekte speziell so erstellt werden, dass eine Skalierung möglich ist.

Lösungsskizze: Vorgehensweise bei der Kodierung des Fib-Objekts, um die Dimension D skalieren zu können:

- Im obersten root-Element wird für die Dimension D kein Definitionsbereich angegeben. Im Fib-Objekt sollten Werte für die Dimension D aus dem Standarddefinitionsbereich der Dimension D kommen. Kann ein Anzeigeprogramm dann die Dimension des Fib-Objekts nicht Skalieren wird des Fib-Objekt trotzdem noch angezeigt.
- Es wird eine Eingabevariable $inVarX$ angegeben, über welche die Anzahl der Punkte in der Dimension D verändert werden kann. Für diese Eingabevariable wird ein Standardwert festgelegt, bei dem das Fib-Objekt so angezeigt wird das der Standarddefinitionsbereich der Dimension D ausgenutzt wird.

Für diese Eingabevariable wird ein Eintrag in den optionalen Informationen des root-Elements der Form “inVarX::dimensionD::points” gemacht.

Teil XIII
Die Fib-Algorithmen

Viele Operatoren bauen auf Überlegungen und Algorithmen auf, welche in diesem Teil beschrieben werden.

Diese Algorithmen können von verschiedenen Operatoren genutzt und miteinander kombiniert werden.

Für die meisten Algorithmen gibt es gute Beschreibungen in entsprechender Fachliteratur. Allerdings sind diese Beschreibungen meist schlecht direkt für Fib anzuwenden. Das Problem liegt in der Art der verarbeiteten Daten. Während viele Algorithmenbeschreibungen für reele Zahlen und abstrakte oder exakte Probleme ausgelegt sind, werden bei der Umsetzung von Multimediadaten in Fib die Daten meist Ganzzahlen oder zumindest skalierte Ganzzahlen sein. Zusätzlich ist eine speicherplatzspaarende Darstellung oft wichtiger, als eine exakte Umsetzung der Multimediadaten. Deshalb sind die Allgorithmen meist für Fib anzupassen.

94 Ableitung

Da Fib nicht nich auf reelen Zahlen arbeiten kann, vereinfacht sich der Begriff der Ableitung. Als Ableitung einer Funktion f wird im Folgenden die Funktion f' verstanden, welche die Differenz zweier benachbarter Eingabewerte von f berechnet.

Meist werden die Funktion für Fib als Eingabevariablen wohl Werte haben, welche auf die Dimensionen von Fib-Objekten zurückgehen. Dies ist beispielsweise der Fall, wenn die Farbe (Ausgabewert) eines Punktes anhand seiner Position (Eingabewert) oder die Position in der zweiten Dimension (Ausgabewert) anhand der Position in der ersten Dimension (Eingabewert) mithilfe einer Funktion ermittelt werden soll.

Daher sind als Ausgangsfunktion f zwei Funktionsklassen für Fib zu unterscheiden. Funktionen die auf Ganzzahlen arbeiten und Funktionen, welche auf ratinalen Zahlen arbeiten. Diese werden im Nachfolgenden genauer betrachtet.

94.1 Ganzzahlfunktionen

Bei diesen Funktionen, sind die ganzzahlige Funktionswerte als n dimensionale Matrix gegeben. Die Anzahl der Dimensionen wird meist eins oder zwei sein.

Soll mit den Werten der Ableitung ein Taylorpolynom oder andere rationale Funktion erstellt werden, so ist zu beachten, dass rationale Werte auf ganzzahlige Werte gerundet werden. Alle rationalen Zahlen x die $n - 0.5 < x < n + 0.5$ erfüllen werden zu der Ganzzahl n gerundet. Durch eine Ableitung verdoppelt sich der möglich Bereich, für den die Zahlen in der Ableitung noch auf eine bestimmte Ganzzahl gerundet werden können. Wenn beispielsweise ein Punkt den Wert 1.51 (gerundet 2) hat und sein Nachbarpunkt 2.49 (gerundet 2), ist die Differenz dieser beiden Punkte ungefähr 1. Bis zu einen Abstand von 1 ($n' - 1 < x' < n' +$

1) können also in der Ableitung einer Funktionen die Funktion noch als konstant angesehen werden.

Eine Taylorentwicklung nach den exakten (Ganzahl-)Werten, berücksichtig also nicht die Freiheitsgrade durch die Rundung. Eine solche Taylorentwicklung könnte nur schwehr die Treppenartigen Funktionen von Ganzzahlen erfassen.

Daher ist mit den Werten der Ableitung bei Ganzzahlfunktionen mit Vorsicht umzugehen.

94.2 Rationale Funktionen

Die Werte der rationalen Funktionen in Fib-Objekten werden meist in Form einer Matrix vorliegen und daher über die Distanz von Nachparpunkten berechnet werden.

Besonderen Augenmerk ist aber darauf zu verwenden, ob es sich wirklich um rationale Werte handelt oder im Grunde um skalierte Ganzzahlen. Handelt es sich um skalierte Ganzzahlen, so sollten die Überlegungen aus dem vorhergehenden Abschnitt 94.1 beachtet werden.

95 Lösen von linearen Ungleichungssystemen

In diesem Abschnitt wird ein Verfahren zum Lösen eines linearen Ungleichungssystemen vorgestellt. Wenn es keine Lösung für das Ungleichungssystem gibt, liefert das Verfahren eine Lösung für die ersten s Ungleichungen, die noch lösbar sind (s ist maximal).

Das Verfahren ist Iterativ über den Ungleichungen. Außerdem ist der Berechnungsaufwand des Verfahren vermutlich Liniar mit der Anzahl der Ungleichungen (oder besser s) und der Anzahl (d) der Parameter (a_i).

95.1 Problem

Das Kernproblem ist die Lösung der Ungleichungen U_i mit $i = 1, \ldots, s$ mit maximalen s und $s \leq n$.

Ungleichungsystem:

$$\begin{aligned}
U_1 : y_1 &\leq a_1 * x_{1.1} + a_2 * x_{1.2} + \ldots + a_d * x_{1.d} \\
U_2 : y_2 &\leq a_1 * x_{2.1} + a_2 * x_{2.2} + \ldots + a_d * x_{2.d} \\
&\vdots \\
U_n : y_n &\leq a_1 * x_{n.1} + a_2 * x_{n.2} + \ldots + a_d * x_{n.d}
\end{aligned}$$

Gegeben: y_i und $x_{i.k}$
Gesucht: a_i

95.2 Vorrausetzungen

Die Lösung des Problems setzt folgende Dinge vorraus:

- Eine Anzahl von lineare Ungleichungen U_i, welche die Lösung einschränken
- Keine weiteren Einschränkungen der Lösung (sollten welche existieren, kann das Verfahren eventuell angepasst werden)
- Feste Anzahl d von Parametern a_i, welche die Dimension d des Lösungsraums bestimmen
- Feste Werte für die Faktoren x_i und $y_{i.k}$
- Eine (endliche) maximale Lösungsmenge (Hyperkörper der maximal möglichen Lösungen)

95.3 Ursprüngliches Problem

Das ursprüngliches Problem, zu dessen Lösung dieses Verfahren von mir entwickelt wurde, ist die Polynomapproximation (und Splineapproximation) mit festen Freiheitsgraden. Dabei sind für die Kurve nicht feste Punkte vorgegeben, sonder für jeden Punkt ein Wert für die Y-Koordinate und ein möglicher Bereich für die X-Koordinate. Der Bereich der X-Koordinate muß dabei eingehalten werden.

Beispielsweise kann sich der Bereich für die X-Koordinate einfach schon daraus ergeben, dass die Werte Ganzzahlen sind und die Funktionswerte des Polynoms auf Ganzzahlen gerundet werden. Dann wäre es unvernunnftig die Ganzzahlwerte mit dem Polynom interpolieren zu wollen, da dann das Polynom viel komplizierter wird, als es eigendlich müsste.

Damit ergibt sich für eine Kurve für die unterschiedlichen Werte der Y-Koordinate jeweils eine Ungleichung der Form: $y_u \leq a_0 + a_1 * x^1 + a_2 * x^2 + \ldots + a_d * x^d \leq y_o$ Diese kann jeweils in zwei Ungleichungen aufgespalten werden der Form: $y_u \leq a_0 + a_1 * x^1 + a_2 * x^2 + \ldots + a_d * x^d$ und $-y_u \leq -a_0 - a_1 * x^1 - a_2 * x^2 - \ldots - a_d * x^d$

So ergeben sich für die p Punkte $2p$ (=n) Ungleichungen für das hier vorgestellte Verfahren.

95.4 Lösungsansatz

Für die Lösung werden die Ungleichungen als lineare Hyperflächen angesehen, welche den konvexen Hyperkörper der Lösungen begrenzen. Dabei ist die Anzahl der Dimension gleich der Anzahl der Parameter d.

Um den Hyperkörper der Lösungen zu bestimmen, werden nacheinander die Hyperflächen (zu den Ungleichungen) hinzugefügt. Dabei wird der Hyperkörper der Lösungen immer weiter eingeschränkt. Wenn der Hyperkörper der Lösungen durch das hinzufügen der nächsten Ungleichung verschwinden würde (dann würde keine Lösungen mehr existieren), wird das Verfahren gestoppt und ein Punkt aus dem Hyperkörper der Lösungen als Lösung zurückgegeben.

95.5 Grundlagen

Das Verfahren baut auf Folgenden Grundlagen auf: (Im Nachfolgenden ist mit Hyperkörper ein Hyperkörper gemeint, der endlich ist und durch Randhyperebenen definiert bzw. gebildet wird, und Schnittflächen sind Schnittflächen von Hyperebenen.)

- Jeder ebene Schnitt durch konvexen Hyperkörper hat wieder als Schnittfläche einenen konvexen Hyperkörper
  - Geraden (ein dimensionale Hyperkörper) haben immer maximal zwei Schnittpunkte
- Die Orrientierung von linearen Hyperflächen durch Normalform (Richtung des Normalvektors immer nach innen vom Hyperkörper)
  - Alle Eckpunkte des konvexen Hyperkörpers liegen immer in Richtung der Normalvektoren seiner Seitenflächen oder auf diesen
- Liniare Hyperebenen (z. B. Flächen) außerhalb des Hyperkörpers können nicht mehr zu seiner Form beitragen (können ignoriert werden)
  - Hyperflächen, welche keine Schnittpunkte mit dem Hyperkörper haben, liegen außerhalb dieses
- Neue k dimensionale Schnittkörper müssen immer auf neuen h dimensionale Hyperflächen liegen, mit $k < l$ (neue Hyperflächen erzeugen nur Schnittkörper, welche auf ihnen liegen)
  - Alle k dimensionale Randflächen des Hyperkörpers liegen auf $k + 1$ dimensionale Randflächen des Hyperkörpers oder auf keinen Randflächen. Zwar können g dimensionale Hyperflächen in Räumen mit mehr wie drei Dimensionen auch Schnittflächen haben, deren Dimension kleiner als $g - 1$ ist, nur können diese Schnittflächen nicht die g dimensionalen Hyperflächen beranden/eingrenzen und damit auch keine Randflächen eines endlichen Hyperkörpers sein, bei dem die g dimensionalen Hyperflächen Randflächen sind. Dies folgt daraus, dass ein endlicher Hyperkörper auch immer nur endliche Randflächen hat, die zu ihm gehören. Um eine g dimensionalen Randfläche endlich zu machen, muss sie durch $g - 1$ dimensionalen Randflächen begrenzt sein. Nur eindimensionale Punkte benötigen keine Randflächen, da sie schon endlich sind.
  - Punkte begrenzen und definieren jeden k dimensionalen Hyperkörper. Punkte benötigen keine begrenzenden Randflächen, da sie schon endlich sind. Jede g dimensionalen Randebene (mit g größer 0) hat $g - 1$ dimensionale Randebenen, die Sie begrenzen und definieren. Damit wird jede g dimensionalen Randebene auch durch die (0 dimensionale)

Randpunkte, die sie entält, begrenzt und definiert. Zur Untersuchung eines konvexen Hyperkörpers, der durch Hyperebenen gebildet wird, reicht also die Untersuchung seiner Eckpunkte aus.

- Liniare Hyperebenen können nur maximal eine Schnitthyperebenen mit anderen liniare Hyperebenen haben
  - Neue Hyperebnen erhöhen die Anzahl der Schnitthyperebenen auf anderen Hyperebenen maximal um Eins

95.6 Lösung

95.6.1 Eingabeparameter:

- Anzahl der Parameter d
- Anzahl der Ungleichungen p
- Den Vektor U der Ungleichungen U_i
- Ein Wert v_{max} für den maximalen Wert der angenommen werden kann (in allen Dimensionen), alternativ direkt einen Hyperkubus oder -körper der maximalen Lösungsmenge

95.6.2 Wichtige Daten:

- Zähler i für die aktuell zu integrierende Ungleichung, initial auf 1
- Die Menge K der Hyperflächen (der Dimensionalität dim) die den konvexer Hyperkörper der möglichen Lösungen begrenzen, mit je Verweise zu den Hyperflächen, die sie enthalten ($dim - 1$) und in den sie enthalten sind ($dim + 1$)
- Der Vektor E mit allen Eckpunkte E_k des konvexer Hyperkörper der möglichen Lösungen
- Listen LS_g mit noch zu prüfenden Schnittebenen S, welche g Dimensional sind
- Die aktuelle Hyperebene H mit dem Normalenvektors N zur aktuellen Ungleichung U_i

95.6.3 Vorgehen:

Das hier vorgestellte Verfahren ist optimiert. Alle überflüssigen Hyperebenen werden in ihm ignoriert.

0 Initialisiere die Hyperebenen (K und E) mit dem Hyperkörper der maximalen Lösungsmenge (eventuell unter Zuhilfenahme von v_{max})

1 Prüfe i:

1.1 Fall 1: Wenn i größer n, setze i auf n und gehe zu “8. Lösung ermitteln”

1.2 Sonst: Nehme nächste Ungleichung U_i

1.2.1 Erstelle eine orientierte Hyperfläche H für aktuelle Ungleichung $U_i = U[i]$ als Normalgleichung, mit dem Normalenvektor N in Richtung der zulässigen Lösungen. Aus der Ungleichung U_i in der Form $y_i \leq a_1 * x_{i.1} + a_2 * x_{i.2} + \ldots + a_d * x_{i.d}$ wird die Normalengleichung $\vec{x_i} \cdot \vec{a} - y_i = 0$ mit $\vec{x_i} = (x_{i.1}, \ldots, x_{i.d})^T$ und $\vec{a} = (a_1, \ldots, a_d)^T$

2. Prüfe für alle Punkte auf welcher Seite der Hyperflache H sie liegen

2.1 Fall 1: alle Eckpunkte E_k liegen in Richtung des Normalenvektors N oder auf der Hyperebene -> erhöhe Zähler i und gehe zu 1. (die Hyperfläche H schränkt die Lösungsmenge nicht weiter ein, d. h. H ist nicht relevant)

2.2 Fall 2: alle Eckpunkte E_k liegen nicht in Richtung des Normalenvektors N -> erniedrige Zähler i und gehe zu “8. Lösung ermitteln” (durch die Hinzunahme der Ungleichung U_i gäbe es keine Lösungen mehr, bzw. die Ungleichung U_i macht das bisherige Gleichungsystem unlösbar)

2.3 Sonst: einige Eckpunkte E_k liegen in Richtung des Normalenvektors N, andere nicht -> gehe zu Schritt 3., bzw. integriere die Hyperfläche H in den konvexer Hyperkörper der möglichen Lösungen

3. Für jede Hyperfläche H_e mit der Dimensionalität $n-1$ (diese entsprechen den Ungleichungen) aus K (konvexer Hyperkörper der möglichen Lösungen):

3.1 Berechne die Schnittebene (Dimensionalität ist $n-2$) von H_e und H

3.1.1 Fall 1: wenn keine Schnittebene existiert -> gehe zu 3.1 und prüfe die nächste Hyperfläche H_e

3.1.2 Sonst: wenn eine Schnittebene S existiert:

3.1.2.1 Wenn eine identische Hyperfläche wie die Schnittebene S in den Hyperflächen K existiert (bzw. es existiert keine neue Hyperfläche)

3.1.2.1.1 Vermerke zu der gefundenen Schnittebene als Verweise die Hyperfläche H als die enthaltende Hyperfläche

3.1.2.1.2 Füge zu der Hyperfläche H als Verweise die Schnittebene S als eine enthaltende Hyperflächen hinzu

3.1.2.1.3 gehe zu 3.1 und prüfe die nächste Hyperfläche H_e

3.1.2.2 Vermerke zu der Schnittebene S als Verweise die Hyperflächen H und H_e als die enthaltenden Hyperflächen

3.1.2.3 Füge zu den Hyperflächen H und H_e als Verweise die Schnittebene S als eine enthaltende Hyperflächen hinzu

3.1.2.4 Füge Schnittebene S zu der Liste LS_{n-2} mit noch zu prüfenden Schnittebene hinzu

3.1.2.5 Füge die Schnittebene S zu den Hyperflächen K hinzu

3.1.2.6 Gehe zu 3.1 und prüfe die nächste Hyperfläche H_e

4. Füge Schnittebenen der Schnittebenen hinzu: für $d_s = n - 2$ bis $d_s = 1$ (d_s ist die Dimensionalität der geprüften Schnittebenen)

4.1 Nehme und entferne die letzte Schnittebene S von der List LS_{d_s} (mit den noch zu prüfenden Schnittebene der Dimensionalität d_s):

4.1.1 Für jede Hyperfläche HS die S enthält:

4.1.1.1 Für jede Hyperfläche HU die in HS enthalten ist (HU hat Dimensionalität d_s)

4.1.1.1.1 Berechne Schnittebene SU (hat Dimensionalität $d_s - 1$) von S und HU

4.1.1.1.1 Fall 1: wenn keine Schnittebene SU existiert -> gehe zu 4.1.1.1 und prüfe die nächste Hyperfläche HU

4.1.1.1.1 Sonst: wenn eine Schnittebene SU existiert:

4.1.1.1.1.1 Wenn eine identische Hyperfläche (diese sollte in HU schon enthalten sein) wie die Schnittebene SU in den Hyperflächen K existiert (bzw. es existiert keine neue Schnitthyperfläche)

4.1.1.1.1.1.1 Vermerke zu der gefundenen Schnittebene als Verweise die Hyperfläche S als die enthaltende Hyperfläche

4.1.1.1.1.1.2 Füge zu der Hyperfläche S als Verweis die Schnittebene SU für eine enthaltende Hyperflächen hinzu

4.1.1.1.1.1.3 gehe zu 4.1.1.1 und prüfe die nächste Hyperfläche HU

4.1.1.1.1.2 Vermerke zu der Schnittebene SU als Verweise die Hyperflächen S und HU als die enthaltenden Hyperflächen

4.1.1.1.1.3 Füge zu den Hyperflächen S und HU als Verweise die Schnittebene SU für eine enthaltende Hyperflächen hinzu

4.1.1.1.1.4 Füge Schnittebene SU zu der Liste LS_{d_s-1} mit noch zu prüfenden Schnittebene hinzu

4.1.1.1.1.5 Füge die Schnittebene SU zu den Hyperflächen K hinzu

4.1.1.1.1.6 Gehe zu 4.1.1.1 und prüfe die nächste Hyperfläche HU

4.2 Wenn LS_{d_s} nicht leer ist, gehe zu 4.1

4.3 Wenn d_s größer 1 ist ernidrige d_s und gehe zu 4

5. Entferne alle nicht mehr benötigten Eckpunkte:

5.1 Entferne alle Eckpunkte (bzw. Hyperflächen der Dimensionalität 0), inclusive Verweise auf sie, aus K und E die nicht in Richtung des Normalenvektors N der Hyperfläche H von einer Seite von H liegen (bzw. entferne die Eckpunkte, welche die Hyperfläche H aus dem Hyperkörper der möglichen Lösungen K abschneidet)

5.2 Entferne die Überflüssigen Eckpunkte die durch H hinzukamen, nehme und entferne den letzten Eckpunkte E von der List LS_0 mit noch zu prüfenden Schnittebene der Dimensionalität 0:

5.2.1 Für jede Hyperfläche bzw. Gerade G die E enthält:

5.2.1.1 Fall 1: Enthält die Gerade G maximal zwei Eckpunkte bzw. Hyperebenen -> gehe zu 5.2.1 und prüfe die nächste Gerade G

5.2.1.2 Sonst: Enthält die Gerade G mehr als zwei (bzw. drei) Eckpunkte bzw. Hyperebenen (die Eckpunkte bilden dann keinen konvexen Körper auf G mehr):

5.2.1.2.1 Ermittle den mittleren Punkt PM der drei Schnittpunkte auf G (bzw. enthaltenden Hyperebenen) und die beiden anderen Punkte P_1 und P_2

5.2.1.2.2 Für jede Hyperfläche H_e mit der Dimensionalität $n-1$ (diese entsprechen den begrenzenden Ungleichungen U_i) aus K die den Punkt PM (indirekt) enthält:

5.2.1.2.2.1 Fall 1: beide Punkt P_1 und P_2 liegen in Richtung des Normalenvektors N_e der Hyperfläche H_e von einer Seite von H_e oder auf H_e (also in der Lösungsmenge) -> gehe zu 5.2.1.2.2 und prüfe die nächste Hyperfläche H_e

5.2.1.2.2.2 Sonst: der Punkt P_r liegen nicht in Richtung des Normalenvektors N_e der Hyperfläche H_e von einer Seite von H_e oder auf H_e (also ist er nicht in der Lösungsmenge)

5.2.1.2.2.2.1 Entferne den Punkt P_r aus der Menge der Eckpunkte E (inclusive Verweise auf ihn)

5.2.1.2.2.2.1 Wenn E nicht entfernt wurde ($E \neq P_r$), gehe zu Schritt 5.2.1 und prüfe die nächste Gerade G, sonst gehe zu 5.2 und prüfe den nächsten Eckpunkt E

6. Entferne alle nicht mehr benötigten Hyperflächen: für $d_h = 1$ bis $d_h = (n-1)$ (Dies sind alle Hyperflächen, die keine Eckpunkte des konvexen Hyperkörper der Lösungen enthalten und damit nicht zu ihm gehören.)

6.1 Entferne alle Hyperflachen, inclusive Verweise zu ihnen, der Dimensionalität d_h die keine Hyperflächen (diese haben Dimensionalität $d_h - 1$) mehr enthalten

7. wenn $i < n$: erhöhe Zähler i und gehe zu 1. (sonst 8.)

8. **Lösung ermitteln:**

8.1. Alternative 1:

8.1.1. Gebe E als Eckpunkte der (konvexen) Lösungsmenge und i als Anzahl der erfüllten Ungleichungen zurück (eine Lösung kann dann aus dem Bereich E ermittelt werden)

8.1. Alternative 2:

8.1.1. Nehme den Mittelpunkt /Schwerpunkt M aller Eckpunkte E_k: $M = 1/\sharp E * \sum_{i=1}^{\sharp E} E_k$

8.1.2. Gebe M als Lösung und i als Anzahl der erfüllten Ungleichungen zurück

95.7 Optimierte Lösung

Für die optimierte Lösung werden alle Randhyperebenen außer Punkten, Geraden und Hyperebenen der Dimensionalität $d-1$ weggelassen. Der hier gegebene Algorithmus ist für mindestens drei Parameter ($3 \leq d$). Wenn die Lösung nur zwei oder ein Parameter hat, kann ein anderer Algorithmus verwendet werden, bzw. der nachfolgende Algorthmus angepasst werden.

0 Initialisiere die Hyperebenen (K und E) mit dem Hyperkörper der maximalen Lösungsmenge (eventuell unter Zuhilfenahme von v_{max})

1 Prüfe i:

1.1 Fall 1: Wenn i größer n, setze i auf n und gehe zu “8. Lösung ermitteln”

1.2 Sonst: Nehme nächste Ungleichung U_i

1.2.1 Erstelle eine orientierte Hyperfläche H für aktuelle Ungleichung $U_i = U[i]$ als Normalgleichung, mit dem Normalenvektor N in Richtung der zulässigen Lösungen. Aus der Ungleichung U_i in der Form $y_i \leq a_1 * x_{i.1} + a_2 * x_{i.2} + \ldots + a_d * x_{i.d}$ wird die Normalengleichung $\vec{x_i} \cdot \vec{a} - y_i = 0$ mit $\vec{x_i} = (x_{i.1}, \ldots, x_{i.d})^T$ und $\vec{a} = (a_1, \ldots, a_d)^T$

2. Prüfe für alle Punkte auf welcher Seite der Hyperflache H sie liegen

2.1 Fall 1: alle Eckpunkte E_k liegen in Richtung des Normalenvektors N oder auf der Hyperebene -> erhöhe Zähler i und gehe zu 1. (die Hyperfläche H schränkt die Lösungsmenge nicht weiter ein, d. h. H ist nicht relevant)

2.2 Fall 2: alle Eckpunkte E_k liegen nicht in Richtung des Normalenvektors N -> erniedrige Zähler i und gehe zu “8. Lösung ermitteln” (durch die Hinzunahme der Ungleichung U_i gäbe es keine Lösungen mehr, bzw. die Ungleichung U_i macht das bisherige Gleichungsystem unlösbar)

2.3 Sonst: einige Eckpunkte E_k liegen in Richtung des Normalenvektors N, andere nicht -> gehe zu Schritt 3., bzw. integriere die Hyperfläche H in den konvexer Hyperkörper der möglichen Lösungen

3 ermittle alle (Rand-) Linien/Geraden L_z des Hyperkörpers K zwischen den Eckpunkten E_o nicht in Richtung des Normalenvektors N und Eckpunkten E_i in dessen Richtung (ignoriere Eckpunkte E_{ho} auf H)

3.1 für jede dieser Geraden L_z: ermittle Schnittpunkte E_{hn} mit der Hyperfläche H und füge sie zum Hyperkörper hinzu

3.2 finde Geraden zwischen den Schnittpunkten auf H, welche einen konvexen Körper bilden: für jeden Schnittpunkt E_{h1} auf H ($\{E_{h1}\} = \{E_{h2}\} = \{E_{ho}\} \cup \{E_{hn}\}$) ermittle alle Schnittpunkt E_{h2} auf H die gemeinsam auf mindestens $d - 2$ anderen Hyperflächen H_a ($H \neq H_a$) liegen. (Diese Hyperflächen sollten zusammen mit der Hyperfläche H als Schnittfläche eine Gerade haben. Da H die Dimensionalität $d - 1$ hat und der Schnitt mit jeder weiteren ($d - 1$ dimensionalen) Hyperflächen die Dimensionalität der Schnittfläche um eins verringert.)

3.2.1 ermittle Geraden G zwischen E_{h1} und E_{h2} die gemeinsam auf gemeinsammen anderen Hyperfläche H_l liegen

3.2.2 füge Geraden G zu K und als auf H liegend hinzu

3.2.3 füge E_{h1} und E_{h2} als auf G liegend hinzu

4. entferne alle Eckpunkte E_k die nicht in Richtung des Normalenvektors N von H liegen

5. entferne alle Gerade G die nur noch einen Eckpunkt besitzen

6. entferne alle Hyperflächen auf denen keine Geraden mehr liegen

7. wenn $i < n$: erhöhe Zähler i und gehe zu 1. (sonst 8.)

8. **Lösung ermitteln:**

8.1. Alternative 1:

8.1.1. Gebe E als Eckpunkte der (konvexen) Lösungsmenge und i als Anzahl der erfüllten Ungleichungen zurück (eine Lösung kann dann aus dem Bereich E ermittelt werden)

8.1. Alternative 2:

8.1.1. Nehme den Mittelpunkt /Schwerpunkt M aller Eckpunkte E_k: $M = 1/\sharp E * \sum_{i=1}^{\sharp E} E_k$

8.1.2. Gebe M als Lösung und i als Anzahl der erfüllten Ungleichungen zurück

96 Bereiche Approximieren

Wurde ein Fib-Multimidiaobjekt analysiert und Bereiche in ihm gefunden, so sollten diese möglichst günstig mit Fib-Elementen darstellbar sein.

Kombinationen aus dem Bereichs- und Funktionselement bieten sich dafür an. Bereichselemente erzeugen Punktmengen, die mit den Funktionselementen so angepasst werden, dass sie den gegebenen Bereich überdecken.

97 Bereiche for-2fun-for 2 D Approximieren mit Taylorpolynomen

Die Idee dieser approximationsmethode ist es einen zweidimensionalen Bereich mit zwei Bereichselementen und zwei Funktionselementen zu überdeken.

Das erste Areaelement gibt den Grundbereich in einer Richtung x vor. Für jeden x-Wert hat der Bereich eine bestimmte Ausdehnung $y(x)_1$ bis $y(x)_2$ (die Größte und kleinste Ausdehung für den x Wert). Diese Werte $y(x)_1$ und $y(x)_2$ werden mit den zwei Funktionselementen abhängig von den x Werten berechnet und als Grenzen des zweiten Bereichselements genommen. (z. B. $for(x, (x_1, x_2),$ $fun(y_1, ..., fun(y_2, ..., for(y, (y_1, y_2), p(x, y))))))$)

Vorraussetzung für dieses Vorgehen ist, dass der Bereich für jede x Position durch zwei Werte (y_1, y_2) begrenzt ist und alle Werte zwischen y_1 und y_2 zum Bereich gehören. Diese Verraussetzung kann etwas entschärft werden, wenn der Bereich nich hundertprozentig genau approximiert werden muß, sonder auch einige Punkte nicht im Bereich überdeckt werden können. Ansonsten muß der Bereich in mehrere Unterbereiche aufgespalten werden. Die Anzahl der Unterbereich ist dabei mindestens die maximale Anzahl der Unterbrechungen in den (y_1, y_2) Bereichen.

Die Richtung /Dimension der x und y Werte kann frei gewählt werden. Mit der wahl der x und y-Richtung kann daher eventuell die Anzahl der Unterbereiche verringert werden.

98 Polynom Approximation

Allgorithmen für Polynominterpolationen sind in der Mathematik gut bekannt. In zusammenhang mit Fib sind aber einige Besonderheiten zu beachten. Einige der besonderheiten wurden schon im Abschnitt 94 über Ableitungen angesprochen.

98.1 Eindimensionale Polynome mit Y-Bereichen

In Fib gibt es die Problemstellung, dass ein Polynom ($f(x) = y$) interpoliert werden soll, das eine feste Zahl für die Eingabe x hat, aber deren Ausgabewerte y in einem Bereich liegen können. Beispielsweise ist bei Funktionen auf Ganzzahlen die Eingabe eine Ganzzahl und die Ausgabe wird auf eine Ganzzahl y gerundet, wenn sie im Bereich $(y - 0.5, y + 0.5)$ liegt. Dies kommt in Fib unter anderen vor, wenn die Helligkeit eines Punktes in Abhängigkeit von seiner Position berechnet werden soll (z. B. mit "$for(\ x, [1, 20], fun(y\ , mult(x, 0.5), pr(\ (y)_{colorGrayscale}, p(x, 1))))$"). Wird in diesen Fällen nicht der Freiheitsgrad für die Ausgabe ausgenutzt, sondern nur ein fester Wert aus dem möglichen Ausgabewert zur Polynominterpolation gewählt (so dass ein übliches Verfahren angewendet werden kann), kann es zu starken Schwingungen des Polynoms kommen. Dann hat das Polynom viele unnötige Faktoren, die den Speicherplatzverbrach für ein Fib-Object nur unnötig erhöhen.

Deshalb wird im Nachfolgenden ein Verfahren vorgestellt, dass dies Freiheitsgraden berücksichtigt.

gegeben:

- Punkte (mit den Bereichen für den Ausgabewert): $P_1(x_1, (yu_1, yo_1)), \ldots, P_p(x_p, y_p \in (yu_p, yo_p))$
- der maximale Grad des Polynoms n, mit $n < p$

Gesucht wird das Polynom $f(x) = \sum_{i=0}^{n}(a_i(x)^i)$; bzw. dessen Faktoren: a_i mit $i = 0, \ldots, n$.

Lösung:

setze untere Grenze a_n auf $-inf$ und die obere auf $+inf$

1. Lösungschema aufbauen:

$$\begin{array}{lll}
yu_1 & <= a_0 + a_1(x_1) + \ldots + a_n(x_1)^n & <= yo_1 \\
yu_2 & <= a_0 + a_1(x_2) + \ldots + a_n(x_2)^n & <= yo_2 \\
\ldots & & \\
yu_p & <= a_0 + a_1(x_p) + \ldots + a_n(x_p)^n & <= yo_p
\end{array}$$

2. in Dreiecksform bringen: ($<?>$: Vergleich wird entsprechend gedreht wenn durch negative x_i geteilt wird)

- mit a_k ist das a_i mit groessten i, dessen untere und ober Grenze nicht gleich ist
- as_n ist der konstante Faktor, der sich aus den a_i mit jeweils gleicher untere und ober Grenze und dem x Wert der Ungleichung ergibt

$$\begin{array}{l}
yu_1 <= a_0 + a_1(x_1) + \ldots + a_n(x_1)^n <= yo_1 \\
(yu_2 - yu_1)/x_2 <?> a_1 + \ldots + 2 * a_n(x_2)^{n-1} <?> (yo_2 - yo_1)/x_2 \\
yu_3/x_3 - yu_2/x_2 + yu_1 * (1/x_2 - 1/x_3) <?> a_2 + \ldots + 3 * a_n(x_3)^{n-2} \\
<?> yo_3/x_3 - yo_2/x_2 + yo_1 * (1/x_2 - 1/x_3) \ldots \\
yu_n/x_n^n + g_n(yu_1, \ldots, yu_{n-1}, x_1, \ldots, x_{n-1}) <?> k * (k-1)/2 * a_k + as_k \\
<?> yo_n/x_n^n - g_n(yo_1, \ldots yo_{n-1}, x_1, \ldots, x_{n-1}) \\
yu_{n+1}/x_{n+1}^n + g_{n+1}(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_{n+1}) \\
<?> k * (k-1)/2 * a_k + as_{k+1} \\
<?> yo_{n+1}/x_{n+1}^n + g_{n+1}(yo_1, \ldots, yo_n, x_1, \ldots, x_n, x_{n+1}) \\
\ldots \\
yu_p/x_p^n + g_p(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) \\
<?> k * (k-1)/2 * a_k + as_p \\
<?> yo_p/x_p^n + g_p(yo_1, \ldots, yo_n, x_1, \ldots, x_n, x_p)
\end{array}$$

2.b So Umformen, dass in jeder Ungleichung ein a_i (b_i Faktor fuer die a_i)

$$\begin{array}{l}
g_0(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) <?> b_0 * a_0 + as_0 <?> g_0(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) \\
g_1(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) <?> b_1 * a_1 + as_1 <?> g_1(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) \\
g_2(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p) <?> b_2 * a_2 + as_2 <?> g_2(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p)
\end{array}$$

$$\ldots$$
$$yu_n/x_n^n + g_n(yu_1, \ldots, yu_{n-1}, x_1, \ldots, x_{n-1})$$
$$<?> k*(k-1)/2*a_k + as_k <?> yo_n/x_n^n - g_n(yo_1, \ldots, yo_{n-1}, x_1, \ldots, x_{n-1})$$
$$yu_{n+1}/x_{n+1}^n + g_{n+1}(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_{n+1})$$
$$<?> k*(k-1)/2*a_k + as_{k+1} <?> yo_{n+1}/x_{n+1}^n +$$
$$+g_{n+1}(yo_1, \ldots, yo_n, x_1, \ldots, x_n, x_{n+1})$$
$$\ldots$$
$$yu_p/x_p^n + g_p(yu_1, \ldots, yu_n, x_1, \ldots, x_n, x_p)$$
$$<?> k*(k-1)/2*a_k + as_p <?> yo_p/x_p^n + g_p(yo_1, \ldots, yo_n, x_1, \ldots, x_n, x_p)$$

3. Lösungsschema umformen (zu $0 <= p$)

4. Faktoren Eingrenzen:

   1. Fuer jede Ungleichung mit maximal einen Faktor mit einer Grenze im Unendlichen:

      (a) Passe Faktoren an; für jeden Faktor $f = a_i(miti = 1, \ldots n)$ und y*; (enthaelt die Ungleichung einen inf Faktor kann nur dieser gewaehlt werden):

         i. setze anderen nicht f Faktoren so (auf jeweils ihre Ober- oder Untergrenze), dass sie Formel maximieren; bestimme dann f neu

   2. wenn sich in 4. Aenderungen ergeben haben zurück zu 4.

   3. beginne dann mit diesem Wert Schritt 1. “Lösungschema aufbauen” erneut solange für ein a_i sich zuletzt Aenderungen ergeben haben oder es noch nicht eingegrenzt wurde

   4. Wenn alle a_i mit Grenzen belegt und keine Änderungen, wähle jeweils einen Wert (den Mittelwert) fuer a_i aus seinen Bereich aus; beginne mit a_i mit kleinsten i dessen Grenzen noch nicht gleich sind und gehe dann jeweils wieder zum Schritt 4

Teil XIV
Die Fib-Operatoren

In diesem Teil werden einige zur Realisierung vorgesehene Operatoren vorgestellt. Für weitere Operatoren sind der Fantasie natürlich keine Grenzen gesetzt.

Diese Operatoren sollen im genetische Algorithmus (siehe Abschnitt IV auf Seite 95) verwendet werden, um (gute) Individuen bzw. Fib-Multimediadarstellungen in ihm anzupassen, zu verändern oder zu erstellen. Dabei werden natürlich nicht Individuen im genetische Algorithmus direkt verändert, sondern immer nur Kopien dieser. All Individuen im genetische Algorithmus, die zur Erzeugung eines neuen Individuums I verwendet wurden, sind die Elternindividuen dieses (I).

99 Ein Teilobjekt einfügen (Genimport)

Operatorname: `cOpFibImportPartObject`

Status: nicht Implementiert

Um zu gewährleisten, dass sich das kopierte Individuum an das Originalbild "anpassen" kann, können in dieses neue Teilobjekte aus anderen Individuen einfügt werden.

Dies wird realisiert werden, indem an einer ausgewählten Stelle im Fib-Objekt, an der ein Fib-Unterobjekt steht, des Individuums ein (neues) Listenelement einfügt wird. Wobei ein Teil der Listenunterobjekte das/die Objekt(e), das/die an dieser Stelle standen, weiterführt/-führen und ein Listenunterobjekt ein neues Objekt enthält, welches dadurch gebildet wird, dass ein Teilobjekt, aus einem anderen Individuum, hineinkopiert wird.

100 Einen neuen Punkt einfügen

Operatorname: `cOpFibCreatePoint`

Status: nicht Implementiert

Dieser Operator fügt an einer ausgewählten Stelle im Individuum ein (neues) Listenelement ein. Wobei ein Teil der Listenunterobjekte das/die Objekt(e), das/die an dieser Stelle standen, weiterführt/-führen und ein Listenunterobjekt ein neues Objekt enthält, welches dadurch gebildet wird dass ein neuer Punkt mit zufällig gewählten Werten einfügt wird.

101 Einfache Änderungen an einem Fib-Objekt: Mutation

Die Nachfolgenden Operatoren ändern einzelne Teile des Fib-Objekts (eines Individuums) zufällig.

101.1 Löschung eines Teilobjektes

Operatorname: `cOpFibDeletePartObject`

Status: nicht Implementiert

Um einzelne Individuen möglichst klein zu halten, damit sie nicht über alle Grenzen wachsen ohne den Fehler zum Originalmultimediaobjekt zu verkleinern, ist es sinnvoll ein neues Individuum zu erzeugen, indem ein Elternindividuum Kopiert und aus dieser Kopie ein Teilobjekt gelöscht wird. Dabei sind Teilobjekte mit niedriger Fitness zu bevorzugen. Wenn das neu entstandene Individuum eine höhere Fitness als das Alte hat, erhält es auch höhere "Überlebenschancen" und das alte Individuum wird eher gelöscht.

101.2 Ändern eines Wertes

Operatorname: `cOpFibChangeValueRandom`

Status: nicht Implementiert

Bei dieser Operation wird aus dem kopierten Elternindividuum ein Wert ausgewählt und dieser verändert, indem ein zufälliger Wert hinzu addiert wird.

101.3 Ersetzen eines Wertes durch eine Variable

Operatorname: `cOpFibReplaceValueWithVariable`

Status: nicht Implementiert

Bei dieser Operation wird aus dem kopierten Elternindividuum ein Wert ausgewählt und dieser durch eine, für das entsprechende Fib-Element definierte, zufällige Variable ersetzt.

101.4 Ändern einer Variable

Operatorname: `cOpFibChangeVariableRandom`

Status: nicht Implementiert

Bei dieser Operation wird aus dem kopierten Elternindividuum eine Variable ausgewählt und diese durch eine zufällige andere Variable ersetzt, welche auch für das entsprechende Fib-Element definiert ist.

101.5 Ersetzen einer Variable durch einen Wertes

Operatorname: `cOpFibReplaceVariableWithValue`

Status: nicht Implementiert

Bei dieser Operation wird aus dem kopierten Elternindividuum eine Variable ausgewählt und diese durch einen Wert ersetzt. Der Wert sollte nach Möglichkeit ein Wert sein, den die Variable durchlaufen würde.

101.6 Einfügen einer Variablen

Operatorname: `cOpFibCreateFunctionElement`

Status: nicht Implementiert

Operatorname: `cOpFibCreateAreaElement`

Status: nicht Implementiert

Beim Einfügen einer Variablen, wird entweder eine schon unter dem Objekt (in den Elementen die dieses enthalten) existierende Variable eingefügt oder eine neue Variable gebildet.

Beim Bilden einer neuen Variablen wird ein Fib-Element, das die Variable definiert, in der Kopie des Elternindividuum an einer ausgewählten Stelle einfügt und zwar oberhalb der Stelle an der seine Variable eingefügt wurde. So dass es das Fib-Element enthält, in dem seine Variable eingefügt wurde.

Es können Funktions- und Bereichsobjekte eingefügt werden.

Beim Einfügen einer Funktion (`cOpFibCreateFunctionElement`) ist es sinnvoll, diese so zu bilden, dass sie den gleichen Wert repräsentiert, der ersetzt wurde.

Beim Einfügen eines Bereichsobjekts (`cOpFibCreateAreaElement`) ist es sinnvoll, dass der verwendete Bereich, (fast nur) den Wert überdeckt, welcher ersetzt wird. Dies kann z. B. dadurch erreicht werden, dass dieses Bereichsobjekt nur ein Bereich mit dem Vektor (Wert, Wert) oder ($Wert - x, Wert + y$) enthält.

101.7 Löschung eines Funktionselements

Operatorname: `cOpFibDeleteUnusedFunctionElement`

Status: nicht Implementiert

Funktionselemente (`cOpFibDeleteUnusedFunctionElement`) können gelöscht werden, wenn kein Objekt unter ihnen mehr ihre Variable, die sie definieren, enthält. Beim Anwenden dieser Operation, ist es sinnvoll, mehrmals ein Funktionselement aus dem Individuum herauszugreifen, um zu versuchen, dieses zu entfernen.

101.8 Löschung eines Bereichselements

Operatorname: `cOpFibDeleteUnusedAreaElement`

Status: nicht Implementiert

Bereichselemente (`cOpFibDeleteUnusedAreaElement`) können gelöscht werden, wenn kein Objekt unter ihnen mehr ihre Variable, die sie definieren, enthält. Beim Anwenden dieser Operation, ist es sinnvoll, mehrmals ein Bereichselement aus dem Individuum herauszugreifen, um zu versuchen, dieses zu entfernen.

101.9 Hinzufügen eines Unterbereichs zu einem Bereich

Operatorname: `cOpFibCreateNewUnderarea`

Status: nicht Implementiert

Listen mit Unterbereichen, die in Bereichselementen enthaltenen sind, können durch neue Vektoren/ Unterbereiche erweitert werden.

Um einen Gradientenanstieg leichter zu ermöglichen, ist es sinnvoll die Vektoren zum Erweitern sinnvoll zu wählen. Bei Bereichsoperatoren ist es z. B. sinnvoll nur Vektoren einzufügen, die in der Nähe der schon vorhandenen Bereiche liegen, z. B. vorher [(1; 5)] nachher [(1; 5),(7; 8)], da so die Bereiche die "Umgebung ertastend" erweitert werden.

101.10 Löschen eines Unterbereichs aus einem Bereich

Operatorname: `cOpFibDeleteUnderarea`

Status: nicht Implementiert

Unterbereiche, die in Listen von Bereichselementen enthalten sind, können gelöscht werden. Bei Bereichselementen ist es sinnvoll, Vektoren die aneinander grenzen, überschneiden oder nahe beieinander liegen zusammenzufügen (neuer Vektor z. B. (kleinste untere Grenze beider Vektoren; größte obere Grenze beider Vektoren)) oder Vektoren für kleine Bereiche beim Löschen zu bevorzugen z. B. (6; 6).

101.11 Verschieben eines Elements

Operatorname: `cOpFibMoveRandomElement`

Status: nicht Implementiert

Einzelne Elemente (außer Punkte) können in der Hierarchie der Fib-Elemente des Objekts nach oben oder unten verschoben werden.

Dabei muss beachtet werden, dass die Fib-Objekte nicht fehlerhaft werden, d.h. beispielsweise:

- es darf kein Fib-Element nach unten über ein Fib-Element verschoben werden, das die Variable enthält, die das verschobene Fib-Element definiert
- es darf kein Fib-Element nach oben über ein Fib-Element verschoben werden, das eine Variable enthält, die das verschobene Fib-Element benötigt
- wenn ein Listenelement nach unten verschoben wird, werden die Fib-Elemente, die es enthält, abwechselnd in das Fib-Element über es verschoben (die Variablenmenge, die durch Elemente unter Listenelement realisiert werden, sind disjunkt)
- wenn ein Listenelement nach oben verschoben wird, werden Fib-Elemente, über die es wandert, in die Unterobjekte, die es enthält, verschoben, in denen die Variable, die diese Fib-Elemente definieren, verwendet wird (oder auch einfacher möglich: das Fib-Element wird in alle Listen-Unterobjekte verschoben)
- wenn ein Fib-Element, das eine Variable definiert, nach unten über ein Listenelement verschoben wird, muss es in die Objekte kopiert werden, die das Listenelement enthält und die Variablen verwenden, die dieses Fib-Element definiert

101.12 Vertauschen der Unterobjekte im Listenelement

Operatorname: `cOpFibFlipListUnderobjects`

Status: nicht Implementiert

Innerhalb der Listenelement können die einzelnen Unterobjekte, die es enthält, vertauscht werden. Da es für diese Unterobjekte eine feste Reihenfolge der Auswertung gibt und das Vertauschen somit das produzierte Multimediaobjekt verändern kann, bewirkt diese Operation eventuell etwas.

102 Fib-Objekte Vereinfachen

Operatoren die Fib-Objekte vereinfachen verwenden als Grundlage existierende Fib-Objekte. Von diesen werden bestimmte Aspekte analysiert, um Teile des Fib-Objekts durch einfachere, kürzere oder/ und schneller auswertbare Teile zu ersetzen.

102.1 Löschen von überflüssigen Elementen

Teile des Fib-Objekts, die bei der Auswertung nicht im erzeugten Multimediaobjekt auftauchen, können gelöscht werden. Operationen, welche dies bewältigen, werden in diesem Abschnitt vorgestellt.

102.1.1 Entfernen von überflüssigen Bereichs- oder Funktionselemenenten

Operatorname: `cOpFibDeleteUnusedVariableDefinitions`

Status: nicht Implementiert

Bereichs- und Funktionselemenenten sind überflüssig, wenn die Variable, welche sie definieren, nicht mehr verwendet wird.

Diese Operation führt also eine Suche nach Bereichs- oder Funktionselemenenten durch, deren definierte Variable nicht Verwendet wird, und entfernt dann diese. Dafür werden zuerst alle Bereichs- und dann alle Funktionselemenente im Fib-Objekt durchlaufen und geprüft, ob die Variable, welche sie definieren, noch verwendet wird. Wird ein Bereichs- oder Funktionselemenent gefunden deren definierte Variable nicht mehr verwendet wird, wird das Elemenent gelöscht.

102.1.2 Suche nach und Entfernen von Eigenschaftselementen die später überschrieben werden

Operatorname: `cOpFibDeleteOverwritenProperties`

Status: nicht Implementiert

Viele Eigenschaften können nur einmal einem Punkt zugewiesen werden. Wenn also in einem Zweig zweimal Eigenschaften vom gleichen Typ auftaucht, die nicht additiv wirken und keine weiteren Verzweigungen zwischen sich haben, ist die Obere davon überflüssig, da sie von der Unteren überschrieben wird. Das obere Eigenschaftselement kann also ohne Auswirkungen auf das dargestellte Multimediaobjekt gelöscht werden.

Beispielsweise können Farbeigenschaften gelöscht werden, welche durch weiter unten im Fib-Baum stehende Farbeigenschaften überdeckt werden. Toneigenschaften können hingegen niemals gelöscht werden, da sie additiv mit anderen Tönen wirken.

102.1.3 Festlegen einer Hintergrindfarbe

Operatorname: `cOpFibSetBackground`

Status: nicht Implementiert

Dieser Operator seperiert für jeweils eine Eigenschaft, für die es noch keine Hintergrundfestlegung gibt, das Fib-Object nach Teilen die jeweils eine bestimmte Ausprägung dieser Eigenschaft annehmen. In den separierten Teilen kommen dann für die Eigenschaft, nur noch gleiche Vektoren vor.

Der Teil, welcher am größten ist und/oder die längste Auswertungszeit hat, wird aus dem Fib-Objekts entfernt und dafür wird der Hintergrund auf die Eigenschaftsausprägung gesetzt.

102.2 Löschen von überflüssigen zusammenhängenden Teilobjekten

Operatorname: `cOpFibDeleteNonessentialPartObject`

Status: nicht Implementiert

Im Gegensatz zum Operator `cOpFibDeletePartObject` auf Seite 474 werden durch den Operator `cOpFibDeleteNonessentialPartObject` nur zusammenhängenden Teilobjekte gelöscht, welche keine oder negative Auswirkung auf das dargestellte Multimediaobjekt haben.

Dafür wird für jedes enthaltende zusammenhängenden Teilobjekt eines Fib-Objekts geprüft, welche Auswirkungen es auf das dargestelle Multimediaobjekt hat. Hat es keine oder negative Auswirkung, wird das zusammenhängenden Teilobjekt gelöscht.

102.3 Operatoren die Fib-Ausdrücke vereinfachen

Operatoren die Fib-Ausdrücke bzw. Fib-Objekte vereinfachen. Analysieren die Struktur der Fib-Objekte und vereinfachen diese. Auf diese Weise wird das Fib-Objekte durch die Operation wahrscheinlich kleiner oder schneller auszuwerten.

102.3.1 Zusammenfassen von verschachtelten Funktionen

Operatorname: `cOpFibCombineFunctions`

Status: nicht Implementiert

Wird von einem Funktionselement F_1 eine Variable V_1 definiert, die nur noch von einem anderem Funktionselement F_2 genutzt wird, so können die beiden Funktionselemente zu einem Funktionselement zusammengefasst werden.

Dafür wird in dem Funktionselement F_2 an allen Stellen an denen die Unterfunktion für die Variable V_1 auftauch die Unterfunktion von F_1 eingesetzt. Danach wird das Funktionselement F_1 gelöscht.

102.3.2 Zusammenfassen von ähnlichen Funktionen

Operatorname: `cOpFibCombineSimilarFunctions`

Status: nicht Implementiert

In einem zusammenhängenden Teilobjekt tauchen eventuell zwei Funktionselemente auf, welche die gleiche oder zumindest sehr ähnliche Funktionen realisieren (bezogen auf die Werte, welche die Variablen, die sie verwenden, einnehmen). Von diesen beiden Funktionen kann dann eine Funktion F_A ausgewählt werden. Diese Funktion F_A muß dann weit genug im Teilobjekt nach oben verschoben werden, so dass die Variable, die es definiert, an allen Stellen definiert ist, an denen die Variable des nicht ausgewählten Funktionselements verwendet wird. Dann wird die Variable, welche das andere Funktionselement definiert, durch die Variable der ausgewählten Funktionselement F_A ersetzt. Das nicht ausgewählte Funktionselement kann dann gelöscht werden. (Eventuell können auch weitere Fib-Elemente gelöscht werden, welche nun Variablen definieren, die nicht mehr benötigt werden.)

102.4 Operatoren die Fib-Ausdrücke aufräumen

Da Fib-Operatoren nicht in den Fib-Objekten, die sie erzeugen, eingeschränkt werden. Können leicht Fib-Objekten entstehen, die unnötige Teile enthalten (z. B. Teile die nicht dargestellt werden). Diese Teile können von Operatoren entfernt werden, ohne das sich die Ähnlichkeit zum original Multimediaobjekt verschlechtert. Dadurch wird das Fib-Objekt aufgeräumt.

102.4.1 Nur Bereiche zu jedem Punkt, wie sie von diesem benötigt werden

Operatorname: `cOpFibMinimizeAreas`

Status: nicht Implementiert

Bereichsobjekt in Fib-Objekten können Unterbereiche enthalten, die mehr Zahlen enthalten als eigentlich benötigt werden. Beispielsweise kann ein Unterbereich Punkte erzeugen die über den Rand des Multimediaobjekts hinausgehen.

Diese Operation wählt eine Anzahl (eventuell auch alle) Unterbereiche eines Fib-Objekts. Für jeden Bereich wird geprüft, ob er nicht verkleinert werden kann, ohne die Fitness des Fib-Objekts zu verschlechtern. Dafür wird schrittweise die eine Grenze des Bereichs in Richtung der anderen bewegt und die Fitness geprüft. Wird die Fitness verringert, wird die Änderung verworfen. Enthält ein Bereich nur noch ein Element, wird entweder der Bereich aus dem Bereichselement gelöscht, wenn das Bereichselement noch andere Unterbereich enthält, oder das ganze Bereichselement, wenn das Bereichselement keine anderen Unterbereich enthält, gelöscht und die Variable, die es definiert, durch den Wert des Bereichs ersetzt.

103 Analysieren der Originalmultimediaobjekts

Bei Operatoren die das Originalmultimediaobjekt analysieren, werden aus dem Originalmultimediaobjekt Informationen gewonnen.

103.1 Suche nach falschen Punkten

Operatorname: `cOpFibFindWrongPoints`

Status: nicht Implementiert

Parameter:

- Die Anzahl der maximal zu setzenden Punkte.
- Die minimale Abweichung für einen Punkt, die überschritten sein muß, um den Punkt mit den richtigen Eigenschaften einzusetzen.

Bei diesem Operator wird ein Fib-Individuum ausgewält und Punkte, die in ihm Falsch sind, durch die richtigen Punkte ersetzt.

Dafür wird für jeden Punkt im original Multimediaobjekt die Abweichung zum gewählten Individuum geprüft (mithilfe des Aktuell gewählten Inividuum Bewerters). Ist diese Abweichung größer als die minimal zulässige Abweichung, wird der Punkt mit den Eigenschaften des original Multimediaobjekt am Ende des ersten Listenelements eingefügt. (Sollte kein Listenelement existieren wird eines erzeugt.) Dies wird solange fortgesetzt bis die Anzahl der maximal zu setzenden Punkte gesetzt wurde.

103.2 Suche nach Bereichen mit Eigenschaftswerten die von einander Abhängig sind

Bereiche deren Eigenschaften voneinander abhängen, können wahrscheinlich in einfachere Fib-Strukturen umgesetzt werden. Diese zu finden und in Fib-Strukturen Umzusetzen, ist ein guter Ansatz für die Vorgehensweise von Operatoren.

103.2.1 Suche nach einem Bereich mit gleichem Eigenschaftswert

Operatorname: `cOpFibFindEvenArea`

Status: nicht Implementiert

Parameter:

- Anfangspunkt (zufällig gewählt)
- Anteil der falsch überdeckten Punkte (Punkte die durch den erzeugten Bereich falsch überdeckt werden)

- maximale Komplexität des erzeugten Fib-Objekts

Für diese Operation wird ein zufälliger Punkt P_a der im Originalmultimediaobjekt liegt ausgewählt und in die Liste der zu prüfenden Punkte L_p eingefügt. Des weiteren wird eine Eigenschaft E, welche der ausgewählte Punkt hat, ausgewählt.

Danach wird eine Schleife solange durchlaufen bis die Liste der zu prüfenden Punkte L_p leer ist. In dieser Schleife wird der erste Punkt P_1 aus der Liste der zu prüfenden Punkte L_p entfernt. Dieser Punkt P_1 wird an das Ende der Liste der geprüften Punkte L_g angefügt. Weiterhin werden alle Nachbarn des Punktes P_1, welche die gleiche Eigenschaft E haben und die nicht in der Liste der zu prüfenden Punkte L_p oder der Liste der geprüften Punkte L_g sind, in die Liste der zu prüfenden Punkte L_p eingefügt.

Wenn die Schleife beendet ist und die Liste der zu prüfenden Punkte L_p leer ist, wird versucht die Punkte in der Liste der geprüften Punkte L_g zu einem Bereich zusammen zu fassen. Der erstellte Bereich wird durch maximal d Bereichs- und maximal d Funktionselementen dargestellt, wobei d die Nummer der Dimensionen im Originalmultimediaobjekt ist.

Dafür wird der erste Punkt aus L_g (der Anfangspunkt P_a) genommen. Dessen Koordinaten werden mit den definierten Variablen der zur Verfügung stehenden Bereichs- und Funktionselementen dargestellt.

Es wird dann versucht die Bereiche der Bereichselemente so auszudehnen, dass sie transformiert durch die Funktionselemente einen möglichst großen Teil der gefundenen Punkte in L_g überdecken.

Alle Bereichselement, welche nur einen Wert in ihren Bereich haben, werden gelöscht und das Auftreten der von ihnen definierten Variable wird durch den Wert des Bereichselements ersetzt. Alle Funktionselemente, welche nun keinen Variablen mehr enthalten, werden gelöscht und das Auftreten der von ihnen definierten Variable wird durch den Wert ersetzt, den das Funktionselement berechnet hat.

Für alle Eigenschaften, welche auf allen Punkten im Bereich gleich sind, wird jeweils ein Eigenschaftselement für die Eigenschaften erstellt und das erstellte Fib-Objekt in dieses eingefügt.

103.2.2 Suche nach Bereichen mit Eigenschaftswerten die von einander polynomiell Abhängig sind

Operatorname: `cOpFibFindPolynomialAreaDependencies`

Status: nicht Implementiert

Äbhängigkeiten: Polynom vom Grad n

Polynomiale Abhängigkeiten können durch Differenzieren gefunden werden. Dafür wird für die einzelnen Eigenschaftswerte die Differenz von in einer Richtung bzw. Dimension benachbarten Punkten gebildet. Diese Differenz wird wird für jede Dimensionsrichtung gebildet. Auf diese Weise wird für jede Dimension D

und für jedes Eigenschaftselement E (Element in einem möglichen Eigenschaftsvektor) eine Ableitung $f^1_{D,E}$ definiert. Durch wiederholtes n-maliges Ableiten in einer Dimension kann die n'te Ableitung $f^n_{D,E}$ erstellt werden. Die Funktion $f_{D,E}$ gibt die nicht abgeleiteten Eigenschaftswerte der Punkte an.

103.2.3 Suche nach Bereichen mit Eigenschaftswerten die von einander trigonometrischen Abhängig sind

Operatorname: `cOpFibFindTrigonometricAreaDependencies`

Status: nicht Implementiert

Äbhängigkeiten: Sinusfunktionen

Teil XV
Anhang

104 Implemtierungskonventionen

104.1 Allgemeines

Zur Dokumentierung des Quelltextes wird **doxygen** im Java Stil verwendet.

Nur Kommentare die sich direkt auf spezifische Codeabschnitte beziehen erfolgen nicht im doxygen Stil.

Beim Quelltext wird großer Wert auf Verständlichkeit und Übersichtlichkeit gelegt. Die Sprache des Quelltexts (inclusive Quelltextkommentare) ist Englisch.

104.2 Einleitung des Quelltextes

Jede Quelltextdatei beginnt mit einem Informationsheader beziehungsweise einer Dateieinleitung. In ihm steht der Name der Datei, des Autors und das Erstellungsdatum der Datei, gefolgt einer Kurzbeschreibung und den Kopierrechten(Copyright). Nach den Kopierrechten wird der Inhalt der Datei näher beschrieben.

Am Ende der Dateieinleitung folgt ein Änderungsprotokoll, welches nicht in doxygen Stiel geschrieben wird.
Beispiel:

```
/**
 * @class cDomains
 * file name: cDomains.cpp
 * @author Betti Oesterholz
 * @date 09.06.2009
 * @mail webmaster@BioKom.info
 *
 * System: C++
 *
 * This class represents a list of domains.
 * Copyright (C) @c LGPL3 2009 Betti Oesterholz
 *
 * This program is free software: you can redistribute
 * it and/or modify it under the terms of the
 * GNU Lesser General Public License (LGPL) as published
 * by the Free Software Foundation, either version 3 of
 * the License, or any later version.
 *
 * This program is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even
```

```
 * the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU Lesser General
 * Public License for more details.
 *
 * You should have received a copy of the GNU Lesser
 * General Public License along with this program.
 * If not, see <http://www.gnu.org/licenses/>.
 *
 *
 * This class represents a list of domains for an
 * root-element.
 * The domain list consists of a number of domains with
 * a type.
 *
 */
/*
History:
09.06.2009  Oesterholz  created
12.07.2009  Schmidt     new method to get the size of
   an domainelement
*/
```

104.3 Formatierung

Gearbeitet wird mit Tabulatoren. Jeder Block, außer Methodenimplementationen und einzeilige Blöcke, wird durch ein Tab eingerückt.

Die einzelenen Elemente in einer Zeile werden nach Möglichkeit durch Leerzeichen seperiert. Die Länge einer Zeile sollte möglichst 75 Zeichen nicht überschreiten.
Beispiel:

```
if ( a == b ){
   //it makes something
   do_it;
   call( parameter );
}
```

104.4 Kommentare

Alle Kommentare sind in Englisch gehalten.

Jede Methode beginnt mit einem Informationsheader im **doxygen** im Java Stil, in dem eine kurze Beschreibung der Methode steht.

Kommentare zu eine Quelltextsegment stehen in einer Zeile direkt vor diesem.

104.5 Namen

Namen sind Bezeichner von Klassen, Methoden, Variablen, Konstanten oder Makros.

104.5.1 Klassennamen

Alle Klassennamen beginnen mit einem Kleinen „c“ für „class“, die Teilworte des Klassennamen beginnen mit einem großen Buchstaben (Camelcase).
Beispiel: cDomain; cDomainInteger

104.5.2 Methodennamen

Methoden beginnen mit einem Kleinbuchstaben. Teilnamen im Methodennamen beginnen mit einem großen Buchstaben und grenzen unmittelbar aneinander. (Camelcase)

Methoden, die dazu dienen Werte zurückzugeben, beginnen mit einem „get“. Demgegenüber beginnen Methoden, mit denen Werten gesetzt werden, sollen mit einem „set“.
Beispiel:

- eineMethod();
- getSomeThing();

104.5.3 Variablennamen

Variablennamen beginnen kleingeschriebenen, Teilnamen im Variablennamen beginnen mit einem großen Buchstaben und grenzen unmittelbar aneinander. (Camelcase) Namen von Variablen von grundlegenden Datentypen, wie Beispielsweise `int`, beginnen mit einem Kürzel für diesen Datentypen (Siehe Tabelle 19). Im Allgemeinen sollte am Variablennamenanfang der Typ der Variable erkennbar sein. Auch sollte der Variablennamen nach Möglichkeit die Funktion der Variable beschreiben.
Beispiel: variableName, uiCounterDomain

105 GNU GENERAL PUBLIC LICENSE

Im Nachfolgenden ist der englische Originaltext der GPL (GNU GENERAL PUBLIC LICENSE) aufgeführt.

Dieser Text wurde von der GNU Seite `www.gnu.org` übernommen und in das Dokumentenlayout eingepasst. Auf der GNU Seite `www.gnu.org` ist auch eine deutsche Übersetzung der GPL zu finden, sowie weitere Informationen über die GPL und ihre Schwesterlizenzen.

Kürzel	Datentyp
i	int
s	short
c	char
l	long
u	unsigned
p	pointer / Zeiger
sz	string / Zeichenketten
li	list / Listen
vec	vector / Vektoren
str	stream / Datenstrom

Tabelle 19: Prefixe für Variablennamen

105.1 "GNU GENERAL PUBLIC LICENSE" englischer Originaltext

Date: Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others’ Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER

OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program’s name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

106 GNU LESSER GENERAL PUBLIC LICENSE

Im Nachfolgenden ist der englische Originaltext der LGPL (GNU LESSER GENERAL PUBLIC LICENSE) aufgeführt.

Dieser Text wurde von der GNU Seite `www.gnu.org` übernommen und in das Dokumentenlayout eingepasst. Auf der GNU Seite `www.gnu.org` ist auch eine deutsche Übersetzung der LGPL zu finden, sowie weitere Informationen über die LGPL und ihre Schwesterlizenzen.

106.1 "GNU LESSER GENERAL PUBLIC LICENSE" englischer Originaltext

Date: Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

   As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

   "The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

   You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

   If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

   (a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
   (b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

   The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

   (a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

(b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

   You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

   (a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
   (b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
   (c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
   (d) Do one of the following:

       . Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

       i. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

   (e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

(a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

(b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

   The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

   If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

107 GNU Free Documentation License

In diesem Abschnitt wird die „GNU Free Documentation License“ wiedergegeben, unter der dieses Dokument steht.

Dieser Text wurde von der GNU Seite `www.gnu.org` übernommen und in das Dokumentenlayout eingepasst. Auf der GNU Seite `www.gnu.org` sind auch weitere Informationen über die GPL und ihre Schwesterlizenzen zu finden.
Eine deutsche Übersetzung der „GNU Free Documentation License“ ist unter `de.wikipedia.org/wiki/GNU_FDL/Text` zu finden. Diese ist eine inoffizielle deutsche Übersetzung der GNU Free Documentation License. Sie ist nicht von der Free Software Foundation herausgegeben und erläutert nicht die Bedingungen der GNU FDL – Dies tut nur der nachfolgend original englische Text der GNU FDL.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document „free“ in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of „copyleft“, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The „Document“, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as „you“. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A MModified Versionöf the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A SSecondary Sectionïs a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The „Invariant Sections“ are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The „Cover Texts“ are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A „Transparent“ copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not „Transparent“ is called „Opaque“.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The „Title Page“ means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, „Title Page“ means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section „Entitled XYZ“ means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as „Acknowledgements“, „Dedications“, „Endorsements“, or „History“.) To „Preserve the Title“ of such a section when you modify the Document means that it remains a section „Entitled XYZ“ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled „History“, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled „History“ in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the „History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled „Acknowledgements" or „Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled „Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled „Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled „Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled „History" in the various original documents, forming one section Entitled „History"; likewise combine any sections Entitled „Acknowledgements", and any sections Entitled „Dedications". You must delete all sections Entitled „Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an „aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled „Acknowledgements“, „Dedications“, or „History“, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version“ applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.

Permission is granted to copy, distribute
and/or modify this document under the
terms of the GNU Free Documentation License,
Version 1.2 or any later version published
by the Free Software Foundation; with no
Invariant Sections, no Front-Cover Texts,
and no Back-Cover Texts. A copy of the
license is included in the section entitled
"`GNU Free Documentation License"'.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the „with...Texts.“ line with this:

```
with the Invariant Sections being LIST
THEIR TITLES, with the Front-Cover
Texts being LIST, and with the
Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Literatur

[1] *Einführung in die Mikrobiologie*. http://www.biologie.hu-berlin.de/~baktgen/texte/Vorlesungen/Mikro/3_gen_trans.pdf.

[2] *Genetic Programming for Feature Discovery and Image Discrimination Walter Alden Tackett*. http://citeseer.ist.psu.edu/tackett93genetic.html.

[3] *Genetische und Evolutionäre Algorithmen*. http://www.bwl.uni-mannheim.de/Heinzl/de/downloads/ki-kapitel-4-6.pdf, WS 2002/03.

[4] BAUER, HEINZ: *Wahrscheinlichkeitstheorie*. New York, 4 Auflage, 1991.

[5] BEHRENDS, ERHARD: *Überall Zufall: Eine Einführung in die Wahrscheinlichkeitsrechnung*. Mannheim, 1994.

[6] CANNON, W.D.: *The Wisdom of the Body*. W.W. Norton, New York, 1932.

[7] DARWIN, CHARLES: *Die Entstehung der Arten*. Nikol, 1963.

[8] FOGEL, DAVID B.: *Evolutionary Computation - The Fossil Record*. New York, 1998.

[9] FOGEL, DAVID B.: *Evolutionary computation: toward a new philosophy of machine intelligence - 2nd ed.* Institute of Electrical and Electronics Engineers, 2000.

[10] GÖRZ, GÜNTHER, CLAUS-RAINER ROLLINGER und JOSEF SCHNEEBERGER: *Handbuch der Künstlichen Intelligenz*. Oldenbourg Verlag München Wien, Oldenbourg Wissenschaftsverlag GmbH; Rosenheimer Str. 145; D-81671 München, 4 Auflage, 2003.

[11] HEITKÖTTER, JÖRG und DAVID BEASLEY: *The hitch-Hiker's Guide to Evolutionary Computation*. http://surf.de.uu.net/hhg2ec/.

[12] HOLLAND, JOHN H.: *Genetic Programming*. 4 Auflage.

[13] KIRSCHER, MARC W. und JOHN C. GERHART: *Die Lösung von Darwins Dilemma*. Rowohlt Taschenbuch Verlag, 2007.

[14] MCCONNELL, STEVE: *Code Complete*. Microsoft Press Deutschland, 2 Auflage, 2004.

[15] MICHALEWICZ, ZBIGNIEW: *Genocop – Optimization via Genetic Algorithms*. http://www.cs.sunysb.edu/~algorith/implement/genocop/implement.shtml.

[16] MICHALEWICZ, ZBIGNIEW: *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Third Revision Auflage, 1996.

[17] NAWROTZKI, KURT: *Lehrbuch der Stochastik: eine Einführung in die Wahrscheinlichkeitstheorie und die mathematische Statistik*. Harri Deutsche, 1994.

[18] NISCHWITZ, A., M. FISCHER und P. HABERÄCKER: *Computergrafik und Bildverarbeitung*. Friedr. Vieweg und Sohn Verlag; GWV Fachverlag GmbH, Wiesbaden 2007, 2 Auflage, 2007.

[19] PFIESTER, PAULA, NINA TRETTER, TINA KOPPENHÖFER, MATTHIAS ECKER und PROJEKTBETREUERIN ANNETT STEINER: *Projekt Bildformate*. `http://www.scheib.info/downloads/Projekt_Bildformate.pdf`, 10.2001-2.2002.

[20] POOLE, DAVID, ALAN MACKWORTH und RANDY GOEBEL: *Computational intelligence: a logical approach*. Oxford University Press, 1998.

[21] PRESSMAN, ROGER S.: *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 6 Auflage, 2005.

[22] RUSSELL, STUART und PETER NORVIG: *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition Auflage, 2003.

[23] SCHILDT, HERBERT: *C++ Die professionelle Referenz*. Osborne/McGraw-Hill, 1 Auflage, 2004.

[24] SCHMID, FRIEDRICH und MARK TREDE: *Skript zur Vorlesung: Einführung in die Stochastik der Finanzmärkte: Stochastische Prozesse, Simulation und Anwendungen*. `http://www.wiwi.uni-muenster.de/~05/hauptstudium/vorlesungen/stofin/stofin.pdf`, SS 2000.

[25] TANENBAUM, ANDREW S. und MAARTEN VAN STEEN: *Verteilte Systeme*. Pearson Education Deutschland GmbH, 1 Auflage, 2003.

[26] TELLER, ASTRO und MANUELA VELOSO: *Algorithm Evolution for Face Recognition: What Makes a Picture Difficult*. `http://www.cs.cmu.edu/~mmv/papers/tellerICEC95.pdf`.

[27] TELLER, ASTRO und MANUELA M. VELOSO: *A Controlled Experiment: Evolution for Learning Difficult Image Classification*. In: *Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, Seiten 165–176. Springer Verlag, October 1995.

[28] TURING, A.M.: *Computing Machinery and Intelligence*. Mind, 59, 1950.

[29] WIELAND, THOMAS: *C++ Entwicklung mit Linux*. dpunkt.verlag, 2001.

Index