OFML – Standardisiertes Datenbeschreibungsformat der Büromöbelindustrie
Version 2.0
3. überarbeitete Auflage

4. November 2015

Verband Büro-, Sitz- und Objektmöbel e.V. (BSO)
Bierstadter Straße 3
D-65189 Wiesbaden
www.buero-forum.de

Die wissenschaftliche Betreuung und Koordinierung der Entwicklung des Datenstandards OFML erfolgte durch Dr. Ing. habil. Ekkehard Beier, ehemals Institut Praktische Informatik und Medieninformatik der Fakultät Informatik und Automatisierung der Technischen Universität Ilmenau.

Der Standard OFML (Parts I-III) wurde im Auftrag des Verbandes Büro-, Sitz- und Objektmöbel e.V. (BSO) durch die EasternGraphics GmbH entwickelt.

Der Standard OFML wurde mit großer Sorgfalt erarbeitet. Dennoch sind Fehler oder Inkonsistenzen nicht auszuschließen. Eine diesbezügliche Haftung wird sowohl von der Wirtschaftsvereinigung Büro-, Sitz- und Objektmöbel e.V., als auch von der EasternGraphics GmbH abgelehnt.

Inhaltsverzeichnis

Referenzen

[GO] EasternGraphics GmbH: *GO – Generische OFML-Typen (OFML Part II).*

[OAM] Verband Büro-, Sitz- und Objektmöbel e.V.: *OAM – OFML Article Mappings (OFML Part VI).*

[OAS] Verband Büro-, Sitz- und Objektmöbel e.V.: *OAS – OFML Article Selection (OFML Part V).*

[OCD] Verband Büro-, Sitz- und Objektmöbel e.V.: *OCD – OFML Commercial Data (OFML Part IV).*

[ODB] EasternGraphics GmbH: *ODB – OFML-Datenbank (OFML Part I).*

[OEX] Verband Büro-, Sitz- und Objektmöbel e.V.: *OEX – OFML Business Data Exchange (OFML Part VII).*

[Rumb91] J.Rumbaugh et al: *Object–Oriented Modelling And Design.* Prentice–Hall, New Jersey, 1991

Kapitel 1

Einleitung

Die Motivation für den neuen Standard der Büromöbelindustrie (OFML[1]) ergibt sich aus einer Reihe von Anforderungen, denen bisherige Lösungen im allgemeinen nicht gerecht wurden:

- Die neuen Anforderungen im Bereich der Planung und Visualisierung von (Büro-)Möbeln können durch CAD-basierte Systeme nur unzureichend erfüllt werden. Die Hauptprobleme CAD-basierter Lösungen sind die immense Datengröße, die mangelnde Parametrisierbarkeit und Konfigurierbarkeit, die unzureichende Abdeckung von Produktlogiken, die unzureichende Darstellungsqualität im interaktiven Bereich, die komplizierte Bedienbarkeit und die kostenintensive Lizensierung.
- Die oben genannten Nachteile verstärken sich im Bereich Marketing-orientierter Lösungen, in dessen Rahmen z.B. Endkunden-orientierte Systeme per CD-ROM oder Internet zum Einsatz kommen sollen.
- Ein plattform- und (Software-)herstellerunabhängiges Datenformat ermöglicht einer unbeschränkten Zahl von Software-Herstellern Systeme und Lösungen anzubieten, so daß monopolartige Zustände vermieden bzw. beseitigt werden können.
- Weiterhin ermöglicht das neue Datenformat die Implementierung einer ganzen Reihe von Applikationen, die trotz unterschiedlicher Ausrichtung kompatibel bezüglich der Daten sind. Auf diese Weise kann eine Kompatibilität und somit technologische Durchgängigkeit zwischen Hersteller-, Handels- und Endkunden-Systemen erreicht werden.

(Traditionelle) CAD-Systeme haben nach wie vor eine Daseinsberechtigung, insbesondere durch ihre Fähigkeiten im konstruktiven und fertigungsnahen Bereich. Folglich besteht der Anspruch des neuen Standards nicht in einer vollständigen Ablösung der existierenden CAD-basierten Lösungen. Statt dessen wird eine Koexistenz zwischen traditionellen CAD-Lösungen und dem neuen Standard angestrebt. Die Koexistenz soll auf der Basis von direkt kompatiblen Datenformaten bzw. geeigneten Konvertierungswerkzeugen realisiert werden.

Die Merkmale des Standards OFML sind im einzelnen:

[1] *Office Furniture Modeling Language* – Modellierungssprache der Büromöbelindustrie

- konsequente Anwendung des objektorientierten Paradigmas,
- Umsetzung von Konzepten der semantischen Modellierung, um eine Übereinstimmung von virtuellen Objekten mit den tatsächlichen Produkten zu erreichen,
- Zusammenfassung von geometrischen, visuellen, interaktiven und semantischen Merkmalen realer Produkte in einem einheitlichen und ganzheitlichen Datenmodell,
- Abbildung realer Konfigurationslogiken und Parametriken,
- Unabhängigkeit von System- oder Oberflächenplattformen und
- Unabhängigkeit von einer konkreten Laufzeitumgebung.

Der OFML–Standard besteht aus folgenden **Teilen (Parts)**, die jeweils unterschiedliche Aspekte der OFML-Datenanlage bzw. verschiedene Anwendungsprozesse abdecken. Die Parts sind mehr oder weniger stark miteinander verbunden, vornehmlich mittels Querverweisen wie Artikelnummern und Typbezeichnern.

1. OFML Datenbank (ODB)

   Die OFML Datenbank [ODB] definiert eine tabellenbasierte Schnittstelle zur Beschreibung von hierarchischen Geometrien in 2D und 3D.

2. Generic Office library (GO)

   Die Klassenbibliothek GO [GO] stellt grundlegende Funktionalität für den Anwendungsbereich der Büromöbelindustrie bereit.

3. Objektmodell

   Dieser Part definiert eine vollständige Programmiersprache, die grundlegenden Schnittstellen von OFML-Typen, vordefinierte Regelursachen, globale (typunabhängige) Funktionen sowie eine Menge von Basistypen. Auf der Basis dieses Objektmodells können beliebig komplexe Daten erstellt und externe kaufmännische Daten integriert werden.

4. OFML Commercial Data (OCD)

   OCD [OCD] definiert eine Menge von Tabellen zur Anlage von (kaufmännischen) Produktdaten, die in Geschäftsprozessen des (Möbel-)Handels benötigt und ausgetauscht werden. Unterstützt werden dabei Aufgabenkomplexe wie Konfiguration komplexer Artikel, Preisermittlung und Erstellung von Angebots- bzw. Bestellformularen.

5. OFML Article Selection (OAS)

   OAS [OAS] beschreibt ein Format zur strukturierten Darstellung und Auswahl von Artikeln in digitalen Katalogen.

6. OFML Article Mappings (OAM)

   Die in diesem Part definierten Tabellen [OAM] dienen zur Festlegung von komplexeren Zusammenhängen zwischen Daten, die gemäß der Spezifikation verschiedener anderer OFML–Parts angelegt wurden.

7. OFML Business Data Exchange (OEX)

   OEX [OEX] beschreibt ein Format für den elektronischen Austausch von Geschäftsdokumenten, wie z.B. Bestellungen und Rechnungen.

Die Parts I-III wurden im Auftrag des Verbandes Büro-, Sitz- und Objektmöbel e.V. (BSO) durch die EasternGraphics GmbH entwickelt. Alle anderen Parts werden durch den Normungsausschuß des BSO spezifiziert.

In diesem Dokument wird im Weiteren nur das Objektmodell (Part III) beschrieben. Alle anderen Parts werden in separaten Dokumenten spezifiziert (siehe Literaturverweise oben).

Dieses Dokument ist wie folgt aufgebaut:

Einleitung und Übersicht

- Dieses Kapitel beschreibt Motivation, Merkmale und Parts des OFML-Standards und gibt einen Überblick über die Struktur des Dokuments.
- Kapitel 2 faßt die relevanten Konzepte und Metaphern des Objektmodells zusammen.

OFML-Part III (Objektmodell)

- Kapitel 3 beschreibt die Basissyntax und -semantik der OFML zugrundeliegenden Programmiersprache.
- Kapitel 4 stellt die grundlegenden Schnittstellen zusammen, auf denen die konkreten Typen des Standards basieren.
- Kapitel 5 beschreibt die Menge der vordefinierten Regelursachen.
- Kapitel 6 beschreibt die für OFML vordefinierte Menge von typunabhängigen Funktionen.
- Die Kapitel 7 und 8 beschreiben die vollständige Menge der OFML-Basistypen.
- Kapitel 9 spezifiziert Typen, die für den Zugriff auf externe Produktdaten notwendig sind.
- Kapitel 10 beschreibt die generischen Planungsumgebungstypen.

Anhang

- Anhang A beschreibt ein generisches Format zur externen Beschreibung von Produktdaten.
- Anhang B dokumentiert eine explizite 2D-Programmierschnittstelle, die in OFML zur Verfügung steht.
- Anhang C dokumentiert ein Metafile-Format, das im Kontext von OFML zur Beschreibung von 2D-Vektorgrafiken verwendet wird.
- Anhang D dokumentiert die Menge der externen Datenformate sowie deren Verwendung.

- Anhang E beschreibt die für die Verwendung von Properties relevanten Formate.
- Anhang F beschreibt die zusätzlichen Typen, die im Rahmen von OFML Anwendung finden können.
- Anhang G beschreibt die im Rahmen dieses Standards verwendeten Notationen.
- Anhang H beschreibt die im Rahmen dieses Standards vordefinierten Kategorien.
- Anhang I definiert die wichtigsten im Rahmen dieses Standards verwendeten Begriffe.

Kapitel 2

Konzepte

In diesem Kapitel werden die grundlegenden OFML-Konzepte beschrieben. Alle in späteren Kapiteln dokumentierten Konzepte basieren letztendlich auf diesen Grundlagen. Folglich ist ein Verständnis dieser Konzepte eine notwendige Grundlage für die weitere Auseinandersetzung mit dem Standard.

2.1 Typen

Ein Typ[1] ist eine Zusammenfassung von gleichartigen Instanzen. Für diese Instanzen definiert ein Typ:

- eine Menge von Methoden,
- eine Menge von Regeln,
- eine Menge von Instanzvariablen und
- genau eine Initialisierungsfunktion.

Jede Instanz gehört zu genau einem unmittelbaren Typ. Ein Typ kann genau einen direkten Supertyp besitzen; von diesem werden die Eigenschaften auf bestimmte Weise geerbt. Folglich ist ein Typ immer in Zusammenhang mit seinen (direkten oder indirekten) Supertypen zu betrachten.

Ein Typname muß innerhalb des definierenden Moduls eindeutig sein. Ein Typname muß weiterhin im globalen Kontext eindeutig sein. Dies wird durch einen einheitlichen Namenspräfix oder durch die Einbeziehung in einen Namensbereich erreicht.

Ein Typ ist entweder abstrakt oder konkret. Von einem abstrakten Typ können keine Instanzen gebildet werden.

[1] Die Begriffe Typ und Klasse sind Synonyme.

Beispiel: Der Begriff eines Korpusschrankes kann als Typ interpretiert werden. Ein bestimmter Korpusschrank(-typ), der auch eine entsprechende Bestellnummer hat, ist ein Beispiel für einen konkreten Typ. Die Generalisierung aller Korpusschranktypen ist ein Beispiel für einen abstrakten Typ.

Der Begriff Schnittstelle ähnelt in seiner Verwendung innerhalb von OFML dem Begriff Typ. Allerdings gibt es die folgenden Ausnahmen:

- Eine Schnittstelle ist ein Beschreibungshilfsmittel und korrespondiert nicht notwendigerweise zu einem Typ.
- Eine Schnittstelle ist nicht von einer anderen Schnittstelle abgeleitet.
- Der Name einer Schnittstelle bekommt keinen Namenspräfix.

2.2 Instanzen

Eine Instanz[2] ist eine konkrete Ausprägung eines Typs. Sie unterscheidet sich von anderen Instanzen durch ihre Identität, die durch einen hierarchischen Namen realisiert ist. Sie unterscheidet sich im allgemeinen weiterhin durch die Belegung der Instanzvariablen, von denen sie jeweils eine eigene Kopie besitzt.

Beispiel: Zwei Korpusschränke mit der gleichen Bestellnummer werden als Instanzen eines gemeinsamen Typs bezeichnet. Sie haben gemeinsame Merkmale, wie z.B. die gleiche Bestellnummer oder die gleichen geometrischen Abmessungen. Sie unterscheiden sich voneinander z.B. durch die Position oder die Materialausführung.

Instanzen sollten im allgemeinen **topologisch unabhängig** sein. Dies bedeutet:

- Eine Instanz soll in ihren Instanzvariablen keine Objektreferenzen, d.h. namentliche Bezüge auf Objekte, speichern.

  Beispiel: Dies würde verletzt, wenn sich eine Instanz eine bestimmte andere Instanz (z.B. auf derselben topologischen Ebene) *merkt*.

- Eine Instanz kann nicht davon ausgehen, daß ihre topologischen Vorfahren von einem bestimmten Typ sind.

  Beispiel: Im Rahmen der temporären Erzeugung von Instanzen kann jede beliebige Instanz Vorfahr einer Instanz sein.

Unter besonderen Umständen können Verletzungen dieser Regeln vorgenommen werden. Die sich dann ergebenden Konsequenzen können beispielsweise sein:

- der Verlust der Speicherbarkeit und
- unkorrektes Verhalten beim Kopieren und Einfügen.

[2] Die Begriffe Instanz und Objekt sind Synonyme.

2.2.1 Kinder

Eine Instanz kann eine Menge von Kindern haben. Ein Kind ist eine Instanz, die im Namensraum des Vaterobjekts existiert. Die Vater-Kind-Relation wird in OFML wie folgt beschrieben:

- Kinder werden zur Laufzeit erzeugt, modifiziert und gelöscht. Folglich ist die Menge der Kinder zeitabhängig.
- Der Vater muß bei der Erzeugung einer Instanz angegeben werden und kann danach nicht mehr verändert werden.
- Das Löschen einer Instanz zieht automatisch das Löschen ihrer Kinder nach sich.
- Ein Kind erbt die Eigenschaften seines Vaters auf bestimmte Weise. Beispielsweise ergibt sich die globale räumliche Modellierung des Kindes aus der Verkettung der globalen räumlichen Modellierung des Vaters und der lokalen räumlichen Modellierung des Kindes.
- Ein Kind kennt seinen Vater. Dies kann für eine Aufwärtstraversierung innerhalb der Szene verwendet werden.
- Ein Vater kennt seine Kinder. Dies kann für eine Abwärtstraversierung innerhalb der Szene verwendet werden.

Instanzen werden in einer Szene plaziert. Aufgrund der oben beschriebenen Merkmale der Vater-Kind-Relation ist die resultierende Szenentopologie eine Menge von Bäumen.

Die Menge der Elemente ist eine Teilmenge der Menge der Kinder. Ein Element ist ein spezielles Kind, dessen Erzeugung und Beseitigung über Regeln (s.u.) kontrolliert werden kann. Jedes Element ist also ein Kind, aber nicht jedes Kind ein Element. Elemente werden üblicherweise für zugreifbare Komponenten einer komplexen Instanz verwendet. Nicht-Elemente sind üblicherweise Komponenten einer zusammengesetzten Instanz, die sich dem Zugriff des Anwenders entziehen.

Beispiel: Die Kinder eines Korpusschrankes sind Wangen, Rückwand, Sockel, Front und Einbauteile. Die Fachböden sind Elemente, da sie sich separat einfügen, bewegen und löschen lassen.

Die einzelnen Bretter des Korpus sind dagegen Nicht-Elemente, da auf diese in der Regel kein Zugriff möglich ist.

Die Verschiebung eines Fachbodens wird durch den Vater des Fachbodens, also der Korpusschrank, kontrolliert (Rasterisierung, Kollisionsvermeidung, Bereichsüberwachung). Der Fachboden muß also seinen Vater kennen, um ihm die Kontrolle über die gewünschte Verschiebung zu übergeben.

Bei Verschiebung des Korpusschrankes müssen die Kinder entsprechend verschoben werden. Zu diesem Zweck müssen die Kinder dem Vater bekannt sein.

Beim Löschen eines Korpusschrankes werden automatisch alle Fachböden, etc. dieses Korpusschrankes mitgelöscht.

Kinder werden syntaktisch wie Instanzvariablen behandelt (Abschn. 3.8.4). Da sie dynamisch erzeugt werden, muß der namentliche Zugriff innerhalb von Methoden über ein vorangestelltes *self* zuzüglich eines Zugriffsoperators erfolgen, z.B. für das Kind *b5*: *self.b5*.

2.2.2 Instanzidentität

Die Identität einer Instanz wird durch einen hierarchischen Namensraum realisiert. Jeder Name innerhalb dieses Namensraumes korrespondiert eineindeutig zu einer topologischen Position in der Objektwelt (Szene). Der Name einer Instanz ergibt sich entsprechend der folgenden Vorschrift:

```
Name        : Name(Vater) '.' LokalerName
            | LokalerName

LokalerName : Buchstabe
            | LokalerName Buchstabe

Buchstabe   : 'A' - 'z' | '0' - '9' | '_'
```

Folglich ergibt sich der Name einer Instanz durch die Verkettung des Namens des Vaters, sofern dieser existiert, über einen Punktoperator mit dem lokalen Namen.

Beispiel:

- *env* – ist der Name eines vaterlosen Wurzelobjekts.
- *env.sky* – ist der Name eines Kindes von *env*. Der lokale Name ist *sky*.
- *env.sky-1* – ist kein gültiger Name.
- *env.sky_1* – ist ein gültiger Name.
- *env.env* – ist der Name eines Kindes von *env* und bezeichnet gleichzeitig ein Geschwisterobjekt von *env.sky*.
- *top* – ist der Name eines weiteren vaterlosen Wurzelobjekts.
- _ – ist nicht erlaubt, weder als globaler noch als lokaler Name.

Die folgenden absoluten Namen sind vordefiniert:

- t – ist das Wurzelobjekt, das die Planungshierarchie zusammenfaßt.
- e – ist das Wurzelobjekt, das ggf. die Umgebungshierarchie zusammenfaßt.
- m – ist das Wurzelobjekt, das ggf. die Bemaßungshierarchie zusammenfaßt.

Daneben können weitere Wurzelobjekte für spezifische Anwendungen definiert werden.

Einschränkung: Die (lokalen) Namen mit der Form $e<n>$, wobei n eine natürliche Zahl ist, sind reserviert und dürfen nicht explizit vergeben werden. Diese Namen werden bei der Erzeugung von Elementen automatisch vergeben.

2.2.3 Instanzvariablen

Ein Typ (in Verbindung mit seinen Supertypen) definiert eine Menge von Instanzvariablen, von denen jede Instanz eine eigene Kopie besitzt. Per Konvention ergibt sich der Name einer Instanzvariablen aus dem Präfix *m* zuzüglich einer nichtleeren Menge von Worten, die jeweils mit einem Großbuchstaben beginnen. Weiterhin ist der Name einer Instanzvariablen ein gültiger Bezeichner im Sinne der Basissyntax (Kap. 3). Beispiele für legale Namen von Instanzvariablen sind: *mWidth* und *mIsCutable*.

Eine Instanzvariable, die in einem Typ definiert wird, darf nicht in einem abgeleiteten Typ neu definiert werden. Des weiteren muß eine Instanzvariable mindestens in dem Typ, in dem sie definiert wurde, initialisiert werden. Der direkte Zugriff auf eine Instanzvariable ist nur innerhalb des definierenden Typs erlaubt. Ein externer Zugriff ist nur über entsprechende Methoden möglich.

Instanzvariablen können ebenfalls durch Schnittstellen definiert werden.

Beispiel: In einem Rollcontainer könnte per Instanzvariable definiert sein, ob dieser eine Drehstangenverriegelung besitzt oder nicht.

2.3 Merkmal

Ein Merkmal (*property*) ist eine spezielle Instanzvariable, die eine implizite Schnittstelle einer Instanz zur (grafischen) Benutzungsoberfläche darstellt. Ein Merkmal besitzt einen Typ, einen symbolischen Bezeichner und einen aktuellen Wert. Meistens ist einem Merkmal ein diskreter Wertebereich zugeordnet. Weitere optionale Eigenschaften eines Merkmals sind die initiale Belegung und – normalerweise bei geometrischen Merkmalen – der minimale Wert und der maximale Wert.

Die aktuelle Ausprägung der Menge der Merkmale einer Instanz korrespondiert im allgemeinen zu einer konkreten Artikelnummer.

Merkmale werden von einem Merkmalseditor (*property editor*) ausgelesen und können durch diesen interaktiv gesetzt werden.

Durch das Konzept der Merkmale kann eine beliebig große Menge von Konfigurationen, die jeweils zu genau einer Artikelnummer korrespondieren, durch einen Typ zusammengefaßt werden, der sämtliche möglichen Konfigurationen abdeckt und dabei auch Abhängigkeiten zwischen einzelnen Merkmalen berücksichtigt.

Beispiel: Die (interaktive) Konfigurierbarkeit eines Korpusschrankes kann durch die drei Merkmale *Breite*, *Höhe* und *Tiefe* realisiert werden. Im allgemeinen ist dann für jedes dieser Merkmale ein herstellerspezifischer diskreter Wertebereich definiert, z.B. für die Breite: *600 mm*, *800 mm*, *1000 mm* und *1200 mm*.

2.4 Methoden

Ein Typ (in Verbindung mit seinen Supertypen) definiert eine Menge von Methoden bzw. typgebundenen Funktionen (Abschn. 3.8.4). Der Name einer Methode ergibt sich aus einer nichtleeren Menge von Worten, von denen alle außer dem ersten mit einem Großbuchstaben beginnen. Des weiteren ist der Name einer Methode ein gültiger Bezeichner im Sinne der Basissyntax (Kap. 3). Beispiele für legale Namen von Methoden sind: *selectable()* und *isSelectable()*.

Eine Methode, die in einem Typ definiert wird, darf in einem abgeleiteten Typ genau dann neu definiert werden, wenn sie dieselbe Signatur besitzt. Im Fall von OFML bedeutet dies, daß Anzahl, Format und Semantik der Parameter gleich sein müssen.

Methoden können ebenfalls durch Schnittstellen definiert werden.

Beispiel: Der Anschlagwechsel einer Tür kann über eine entsprechende Methode realisiert werden. Über diese Methode wird dann der Anschlagwechsel realisierbar, ohne daß der interne Aufbau der Tür nach außen hin bekannt ist.

2.5 Regeln

Ein Typ (in Verbindung mit seinen Supertypen) definiert eine Menge von Regeln. Eine Regel ist ein prozeduraler Konstrukt, der analog zu einer Methode im Bereich eines Typs definiert ist. Eine Regel unterscheidet sich von einer Methode durch die folgenden Merkmale:

- Eine Regel ist ein typgebundener Konstrukt, dessen Signatur aus einer Regelursache in Form eines vordefinierten oder nutzerdefinierten Symbols, einem optionalen spezifischen Regelparameter und einem formalen Parameter besteht.
- Der Rückgabewert einer Regel ist vom Typ *Int*. Der Wert 0 signalisiert die erfolgreiche Bearbeitung der Regel. Der Wert -1 kennzeichnet eine fehlgeschlagene Regel. Das Fehlschlagen einer Regel kann dem Nutzer bei Bedarf durch eine entsprechende Textausgabe mitgeteilt werden.
- Innerhalb eines Typs bzw. einer Hierarchie von Typen kann es mehrere Regeln für ein und dieselbe Regelursache geben.
- Eine Regel kann nicht überschrieben werden, beispielsweise durch eine ursachenidentische Regel in einem abgeleiteten Typ.
- Eine Regel wird als Vorher-Regel oder als Nachher-Regel klassifiziert. Eine Vorher-Regel wird aufgerufen, bevor eine Aktion durchgeführt wird. Das Fehlschlagen einer Vorher-Regel unterbindet die Durchführung der entsprechenden Aktion. Eine Nachher-Regel wird nach der Durchführung einer Aktion aufgerufen. Folglich kann diese Aktion nicht unterbunden werden. Allerdings kann die Wirkung der Aktion durch eine geeignete Gegenaktion rückgängig gemacht werden.

Für eine durchgeführte oder noch durchzuführende Aktion und eine gegebene Instanz wird zunächst eine Liste zusammengestellt, die die durch den Typ und seine Supertypen definierten Regeln für die entsprechende Ursache enthält. Die Reihenfolge der Regeln in der Liste korrespondiert zur Ableitungshierarchie der jeweiligen Typen. Das heißt, eine Regel, die von einem bestimmten Typ definiert wird, steht in der Liste vor einer Regel, die von einem abgeleiteten Typ definiert wird. Die Liste der Regeln wird nun der Reihe nach abgearbeitet. Die Abarbeitung wird abgebrochen, sofern eine Regel fehlgeschlagen ist. In diesem Fall und sofern es sich um eine Vorher-Regel handelt, wird die entsprechende Aktion nicht durchgeführt.

Die in OFML vordefinierten Regelursachen werden in Kapitel 5 dokumentiert.

Beispiel: Das Einfügen eines beliebigen Objekts, z.B. in einen Korpusschrank, kann durch eine entsprechende Vorher-Regel kontrolliert werden. In dieser Regel kann der Korpusschrank beispielsweise sicherstellen, daß nur Fachböden eines bestimmten Typs in einer bestimmten Anzahl einfügbar sind.

Die Verschiebung eines Objekts kann durch eine entsprechende Nachher-Regel kontrolliert werden. Falls beispielsweise durch die Verschiebung eine Kollision auftritt, soll die Verschiebung nachträglich entsprechend korrigiert werden.

2.6 Kategorien

Eine Kategorie ist eine Klassifizierung von Typen bzw. Instanzen, die sich durch eine gewisse Betrachtungsperspektive ergibt.

Kategorien stellen eine Erweiterung zum Konzept der Typen dar: Typen, die zu einer gemeinsamen Kategorie gehören, müssen nicht von einem gemeinsamen Typ abgeleitet sein. Des weiteren kann ein Typ mehreren Kategorien zugeordnet sein.

Die Zugehörigkeit zu einer Kategorie wird von jedem Typ selber festgelegt. Für eine Instanz kann ermittelt werden, ob dessen Typ oder Supertypen einer bestimmten Kategorie angehören (Abschn. 4.1).

Das Konzept der Kategorien kann verwendet werden, um bei der Klassifikation von Instanzen auf Basis orthogonaler Kategorisierungskriterien die Beschränkung der Einfachvererbung von Typen zu umgehen. Weiterhin ist sie sinnvoll, wenn *Rollen* modelliert werden sollen.

Beispiele: Material- und Planungskategorien (siehe Anhang H).

2.7 Initialisierung

Die Initialisierung einer Instanz erfolgt über die Prozedur *initialize()*. Aufgaben der Initialisierung sind im wesentlichen die Initialisierung von Instanzvariablen und die Erzeugung von Kindobjekten. Folgende Merkmale betreffen die Initialisierung:

- Pro Typ gibt es genau eine Initialisierungsfunktion. Diese hat den Namen *initialize()*.

- Innerhalb der Implementierung der Initialisierungsfunktion wird zuerst die Initialisierungsfunktion des direkten Supertyps aufgerufen.

Die Standardsignatur für die Initialisierungsfunktion ist wie folgt:

$$initialize(pFather(MObject),\ pName(Symbol)) \rightarrow MObject$$

Dabei ist *pFather* das Vaterobjekt und *pName* der lokale Name des neu zu erzeugenden Objekts. Der Rückgabewert der Initialisierungsfunktion ist eine Referenz auf das erzeugte Objekt.

Bei Bedarf können weitere beliebige Parameter für die Initialisierungsfunktion eines Typs definiert werden. Dies ist jedoch nur für abstrakte Typen oder interne Komponenten erlaubt. Alle interaktiv instantiierbaren Typen müssen zur Standardsignatur der Initialisierungsfunktion konform sein.

Beispiel: Die Initialisierungsfunktion eines Korpusschrankes muß die entsprechenden Kinder (Wangen, Sockel, Rückwand, etc.) erzeugen und parametrisieren. Die Erzeugung der Fachböden ist dagegen interaktiv zu einem späteren Zeitpunkt möglich.

2.8 Interaktoren

In OFML stellen Interaktoren einen speziellen Typ dar, der, im Gegensatz zu den meisten anderen OFML-Objekten, kein Objekt der realen Welt repräsentiert. Interaktoren sind Objekte, die nur zur Laufzeit existieren und dem Benutzer auf einfache Weise Aktionen ermöglichen, die über elementare Manipulationen wie Translation und Rotation hinausgehen. Beispiele dafür sind das Markieren von Anfügepunkten oder „Griffe“ zum interaktiven Ändern der Größe eines Objekts.

Interaktoren zeichnen sich gegenüber anderen Objekten durch folgende Eigenschaften aus:

- Sie werden nicht persistent abgespeichert.
- Sie sind nicht direkt selektierbar. Der Versuch, einen Interaktor zu selektieren, löst beim Vater die Regel *INTERACTOR* (Abschn. 5.5) aus.
- Interaktoren können keine Kollision verursachen.
- Sie werden bei der photorealistischen Ausgabe und beim Export in ein externes Datenformat ignoriert.

Beispiel: An einer Organisationswand können an verschiedenen Positionen Aufbauten angebaut werden. Werden für diese Positionen Interaktoren definiert, kann der Nutzer interaktiv den gewünschten Anfügepunkt selektieren.

Kapitel 3

Basissyntax und -semantik

3.1 Einführung

Dieses Kapitel beschreibt die programmiersprachlichen Grundlagen von OFML, die sich syntaktisch an den Programmiersprachen C, C++ und Java orientiert. Aus semantischer Sicht ähnelt OFML Smalltalk oder Python, da ein dynamisches Typkonzept zugrunde liegt.

3.1.1 Darstellung der Syntax

Zur Darstellung der Syntax wird in diesem Dokument eine leicht modifizierte Variante der bekannten Backus–Naur–Form verwendet. Es gelten folgende typographische Konventionen: Reservierte Bezeichner, Zeichen und Zeichenkombinationen werden in `Schreibmaschine` dargestellt. Ansonsten werden alle Grammatiksymbole *kursiv* geschrieben. Mehrere Alternativen für die rechte Seite einer Produktion werden entweder durch Zeilentrennung und Einrückung oder durch „|“ innerhalb einer Zeile getrennt. Optionale Symbole werden durch ein tiefgestelltes „*opt*“ gekennzeichnet:

`{` $stmt_{opt}$ `}`

3.1.2 Implementierung

Die Sprachdefinition von OFML geht davon aus, daß ein OFML–Programm von einem Übersetzungsprogramm in eine abarbeitbare Form konvertiert wird[1]. Dies erfolgt in zwei Phasen:

1. Übersetzung aller Definitionen auf Modul- und Klassenebene. Ausführbare Anweisungen sowie Definitionen innerhalb von Verbundanweisungen werden in diesem Schritt gar nicht oder nur teilweise übersetzt.

[1] Dabei kann es sich z.B. um Bytekode, Maschinenkode oder auch gerichtete Graphen handeln.

2. Übersetzung aller ausführbaren Anweisungen und Definitionen innerhalb von Verbundanweisungen. Dieser Schritt kann abhängig von der Implementierung für jede einzelne Verbundanweisung bis unmittelbar vor ihre erste Abarbeitung hinausgezögert werden.

Zweck dieser Zweiteilung ist die Behandlung von Übersetzungseinheiten, die durch sie definierte Variablen, Funktionen oder Klassen gegenseitig referenzieren. Nicht erlaubt sind Übersetzungseinheiten, die auf Grund der gegenseitigen Referenzierung von Oberklassen einen Zyklus bilden.

Ein weiteres Anliegen für die Zweiteilung ist, die für die Übersetzung des Programms benötigte Zeit teilweise auf die Laufzeit zu verteilen, was durch die verzögerte Übersetzung von Verbundanweisungen erreicht werden kann.

3.1.3 Programmstruktur

Ein OFML–Programm besteht aus einer oder mehreren Übersetzungseinheiten. Jede Einheit stellt dabei eine Folge von Zeichen des Zeichensatzes (Abschn. 3.2.1) dar, die sowohl in Form einer Datei als auch als Zeichenkette vorliegen kann. Jede Übersetzungseinheit wird konzeptionell durch das Endezeichen *EOF* abgeschlossen. Dieses Zeichen ist nicht Bestandteil des Quelltextes, sondern dient lediglich zur Darstellung des Endes vom Eingabestrom in der Syntaxbeschreibung.

3.2 Lexikalische Struktur

Der erste Paß in der Verarbeitung liest eine Folge von Eingabezeichen und produziert als Ergebnis eine Folge lexikalischer Symbole (*Token*)[2].

3.2.1 Zeichensatz

Der vom Übersetzungsprogramm verarbeitete Zeichensatz ist die Menge der druckbaren ASCII–Zeichen, d.h. 8 bit–Zeichen mit einem ganzzahligen Wert von 32 bis 126, sowie die in Abschnitt 3.2.1 genannten Steuerzeichen. Ausnahmen sind lediglich in Kommentaren und literalen Zeichen- und Zeichenkettenkonstanten erlaubt. Im letzten Falle ist der Programmierer dafür verantwortlich, daß die entsprechenden Zeichen korrekt von der Laufzeitumgebung verarbeitet werden. Nicht druckbare ASCII–Zeichen werden im folgenden als Hexadezimalzahlen in der in C üblichen Weise dargestellt; das Grammatiksymbol *any-chars* bezeichnet eine beliebige Folge von Zeichen aus dem gesamten Zeichensatz der Implementierung.

Alphanumerische Zeichen

Die folgenden Produktionen definieren Buchstaben (*alpha*), Zahlen (*num*) und alphanumerische Zeichen (*alnum*). Man beachte, daß der Unterstrich auch zu den Buchstaben zählt.

[2]Um Verwechslungen mit OFML–Symbolen zu vermeiden, wird hier der englische Begriff verwendet.

alpha:

A | B | C | D | E | F | G | H | I | J | K | L | M
N | O | P | Q | R | S | T | U | V | W | X | Y | Z
a | b | c | d | e | f | g | h | i | j | k | l | m
n | o | p | q | r | s | t | u | v | w | x | y | z
_

num:

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

alnum:

alpha | *num*

Zwischenräume

Folgende Zeichen bilden, als Folge oder zusammen mit Kommentaren (Abschn. 3.2.1) *Zwischenräume* (*white-space*): Horizontal–Tabulator (HT), Zeilenwechsel (NL), Vertikal–Tabulator (VT), Seitenvorschub (FF), Wagenrücklauf (CR) und Leerzeichen (SP). Folgt ein Bezeichner oder Schlüsselwort auf einen Bezeichner, ein Schlüsselwort oder ein Symbol, so sind beide durch einen Zwischenraum zu trennen. Das gleiche gilt für ganzzahlige Konstanten (ausgeschlossen Zeichenkonstanten) und Gleitkommakonstanten. Ansonsten haben Zwischenräume keinerlei Bedeutung, sondern dienen lediglich der besseren Lesbarkeit von Programmen.

white-space:

HT | NL | VT | FF | CR | SP | *comment*

Kommentare

Kommentare beginnen mit der Zeichenkombination `//` und enden mit einem Zeilenwechsel (NL) oder Wagenrücklauf (CR) oder einer Kombinationen von beiden.

comment:

`//` *any-chars eol*

eol:

CR | NL | CR NL | NL CR

Eine Sonderstellung nimmt das Zeichen `#` ein: Tritt es am Anfang der ersten Zeile einer Datei auf, wird der Rest dieser Zeile ebenfalls als Kommentar interpretiert.

3.2.2 Token

Es werden folgende Klassen von Token unterschieden: Schlüsselworte, Bezeichner, literale Konstanten, Operatoren und Begrenzer.

3.2.3 Bezeichner

Bezeichner *ident* beginnen mit einem Buchstaben und können danach eine beliebig lange Folge alphanumerischer Zeichen enthalten.

ident:
    alpha alnum-seq
alnum-seq:
    alnum alnum-seq$_{opt}$

Ausgeschlossen von der Benutzung als Bezeichner sind die im nächsten Abschnitt genannten Schlüsselworte.

3.2.4 Schlüsselworte

Die folgenden Schlüsselworte gelten als reserviert und dürfen nicht als Bezeichner verwendet werden:

```
abstract   break      case         catch    class
continue   default    do           else     final
finally    for        foreach      func     goto
if         import     instanceof   native   operator
package    private    protected    public   return
rule       self       static       super    switch
throw      transient  try          var      while
```

3.2.5 Literale Konstanten

In OFML sind literale Konstanten der folgenden Typen (Abschn. 3.3) enthalten: Ganzzahlen, Gleitkommazahlen, Zeichenketten und Symbole.

constant:
    integer-constant
    float-constant
    string-constant
    symbol-constant

Ganzzahlige Konstanten

Ganzzahlige Konstanten (*integer-constant*) können in drei verschiedenen Zahlensystemen angegeben werden: dezimal, oktal und hexadezimal. Da OFML keine Zeichentypen kennt, werden Zeichenkonstanten (*character-constant*) ebenfalls als Ganzzahlen interpretiert.

integer-constant:
dec-constant
oct-constant
hex-constant
character-constant

Dezimalzahlen beginnen mit einer Ziffer ungleich `0`, gefolgt von einer beliebigen Ziffernfolge:

dec-constant:
dec-start *dec-rest*$_{opt}$
dec-start:
`1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
dec-rest:
num *dec-rest*$_{opt}$

Oktalzahlen beginnen mit der Ziffer `0`, gefolgt von einer beliebigen Folge der Ziffern `0`–`7`:

oct-constant:
`0` *oct-rest*$_{opt}$
oct-rest:
oct-num *oct-rest*$_{opt}$
oct-num:
`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

Hexadezimalzahlen beginnen mit der Zeichenfolge `0x` oder `0X`, gefolgt von einer beliebigen Folge von Ziffern oder der Buchstaben `A`–`Z` bzw. `a`–`z`:

hex-constant:
hex-start *hex-rest*
hex-start:
`0X` | `0x`
hex-rest:
hex-num *hex-rest*$_{opt}$
hex-num:
num | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

Eine ganzzahlige Konstante muß kleiner oder gleich dem größten in der Implementierung darstellbaren Wert sein. Andernfalls wird während der Übersetzung ein Fehler erzeugt.

Zeichenkonstanten bestehen aus einem Zeichen, eingeschlossen in Hochkommas „'“:

char-constant:
'*char-char*'

Die Menge der in Zeichenkonstanten erlaubten Zeichen wird mit *char-char* bezeichnet. Das Hochkomma selbst, sowie Zeilentrenner, sind in Zeichenkonstanten nicht erlaubt. Um diese und weitere Sonderzeichen darzustellen sind die in Abschnitt 3.2.5 beschriebenen Escape–Folgen zu verwenden.

Der Wert einer Zeichenkonstante ist der numerische Wert des Zeichens im Zeichensatz der Laufzeitumgebung.

Gleitkommakonstanten

Gleitkommakonstanten beginnen mit einem ganzzahligen Teil, gefolgt von einem Dezimalpunkt, dem gebrochenen Teil und dem Exponenten. Der Exponent besteht aus dem Zeichen `E` oder `e`, einem optionalen Vorzeichen und einem ganzzahligen Wert. Der ganzzahlige oder der gebrochene Teil können entfallen, nicht aber beide. Ebenso kann nur der Dezimalpunkt oder nur der Exponent entfallen.

float-constant:
dec-rest `.` *dec-rest float-exp*$_{opt}$
dec-rest `.` *float-exp*$_{opt}$
`.` *dec-rest float-exp*$_{opt}$
dec-rest float-exp
float-exp:
exp-char sign$_{opt}$ *dec-rest*
exp-char:
`E` | `e`
sign:
`+` | `-`

Tritt bei der Konvertierung der Gleitkommakonstanten in die interne Darstellung ein Unterlauf auf, ist der Wert der Konstanten `0.0`. Im Falle eines Überlaufs ist er `Float::HUGE_VAL`. Ist die Genauigkeit der Gleitkommakonstanten größer als durch die interne Darstellung unterstützt, werden überflüssige Stellen ignoriert.

Konstante Zeichenketten

Konstante Zeichenketten (*string-constant*) bestehen aus einer Folge von in Anführungszeichen („`"`“) eingeschlossenen Zeichen. Das Anführungszeichen selbst ist in Zeichenketten nicht erlaubt. Um dieses und bestimmte Sonderzeichen darzustellen, sind die folgenden Escape–Sequenzen zu verwenden:

`\a`	Klingelzeichen (BEL)
`\b`	Rückschritt (BS)
`\t`	Horizontal–Tabulator (HT)
`\n`	Zeilenwechsel (NL)
`\v`	Vertikal–Tabulator (VT)
`\f`	Seitenvorschub (FF)
`\r`	Wagenrücklauf (CR)
`\"`	Anführungszeichen
`\'`	Hochkomma
`\\`	Gegenschrägstrich
`\`*oct-rest*	Oktaler Zeichencode
`\x`*hex-rest*	Hexadezimaler Zeichencode

Die Menge der in konstanten Zeichenketten zulässigen Zeichen wird mit *string-char* bezeichnet.

Der oktale Zeichencode *oct-rest* besteht aus einer Folge von bis zu drei oktalen Ziffern und wird durch das erste nicht–oktale Zeichen beendet. Der hexadezimale Zeichencode *hex-rest* besteht aus einer Folge von hexadezimalen Ziffern unbegrenzter Länge und wird durch das erste nicht–hexadezimale Zeichen beendet.

Tritt während der Übersetzung bei der Konvertierung eines oktalen oder hexadezimalen Zeichenkodes in ein Zeichen ein Überlauf auf, wird ein Fehler erzeugt.

string-constant:
"*string-char-seq*$_{opt}$"
string-char-seq:
string-char string-char-seq$_{opt}$

Literale Symbole

Literale Symbole in OFML beginnen stets mit dem Sonderzeichen „`@`“, direkt gefolgt von einer Zeichenfolge, die den Regeln für Bezeichner entspricht.[3]

symbol-constant:
`@`*ident*

3.2.6 Operatoren

Die folgenden Token werden von OFML als Operatoren behandelt:

[3] Mit Hilfe des Symbol(...)–Konstruktors ist es möglich, Symbole aus beliebigen Zeichenketten zu erzeugen, Abschnitt 3.3.3.

operator:

```
.   | (   | [   | ++  | --  | !   | !!  | ~   | $
*   | /   | %   | +   | -   | <<  | >>  | >>> | <
<=  | >=  | >   | ==  | !=  | ~=  | <?  | >?  | &
^   | |   | &&  | ||  | =>  | ?   | :   | *=  | /=
%=  | +=  | -=  | &=  | ^=  | |=  | <<= | >>= | >>>=
=   | ,   | @(  | ::  | instanceof
```

3.2.7 Begrenzer

Die folgenden Token stellen in OFML Begrenzer dar:

delimiter:

```
:: | { | } | ; | ) | ]
```

3.3 Typen

OFML ist eine dynamisch typisierte Sprache, d.h., daß der Typ einer Variablen oder eines Ausdrucks im allgemeinen erst zur Laufzeit bekannt ist.

Abgesehen von der Klassendefinition gibt es für Typen in OFML keine speziellen syntaktischen Konstrukte. Typen sind Objekte und werden daher wie alle anderen Objekte auch in Variablen gespeichert. Im Rahmen der für Typen definierten Operationen kann mit ihnen wie mit jedem anderen Objekt verfahren werden. Insbesondere bedeutet dies, daß sie zugewiesen, an Funktionen übergeben und aufgerufen werden können.

OFML unterscheidet zwei grundlegende Arten von Typen: einfache Typen und Referenztypen. Einfache Typen sind die numerischen Typen, der Symboltyp und der Typ `Void`. Die Referenztypen sind vordefinierte Referenztypen oder nutzerdefinierte Klassen.

3.3.1 Objekte und Variablen

Ein Objekt ist eine Instanz einer Klasse. Es wird durch Aufruf der entsprechenden Klasse erzeugt. Auf Objekte wird über Referenzen zugegriffen.

Eine Variable ist ein Speicherbereich, in dem ein Wert eines einfachen Typs oder eine Referenz auf ein Objekt eines Referenztyps gespeichert ist.

Es wird zwischen benannten und unbenannten Variablen unterschieden. Benannte Variablen sind alle die Variablen, die durch einen Bezeichner spezifiziert werden können. Auf unbenannte Variablen muß über einen Operator zugegriffen werden (z.B. den Index–Operator `[]`).

3.3.2 Operationen für alle Typen

Alle Typen erben vom Wurzeltyp `Object`. Damit sind folgende Operationen für alle Typen verfügbar:

- Der Konstruktor. Dies ist eine Funktion (Abschn. 3.6.3), die eine typspezifische Anzahl von Parametern verlangt und bei einfachen Typen einen neuen Wert des Typs bzw. bei Referenztypen eine Referenz auf ein neu erzeugtes Objekt liefert.
- Die Zuweisung durch den Operator `=` (Abschn. 3.6). Dabei wird der Variablen auf der linken Seite des Zuweisungsoperators der Wert vom Ergebnis des Ausdrucks auf der rechten Seite zugewiesen. Hat das Ergebnis einen Referenztyp, wird die Referenz zugewiesen, ohne eine neue Instanz des referenzierten Objektes zu erzeugen.
- Die Übergabe als Argument an eine Funktion. Sie erfolgt entsprechend den Regeln der Zuweisung durch den Operator `=`, wobei das Argument dem entsprechenden formalen Parameter der Funktion zugewiesen wird.
- Der Vergleich durch die Operatoren `==` bzw. `!=`. Bei einfachen Typen werden die Werte selbst verglichen, während, falls nicht anders definiert, bei Referenztypen auf Objektidentität geprüft wird.
- Die Überprüfung des Typs mittels des Operators `instanceof`.

3.3.3 Einfache Typen

Alle einfachen Typen sind in dem Paket `::cobra::lang` definiert.

Der leere Typ `Void`

Der leere Typ `Void` wird immer dann verwendet, wenn eine Variable keinen konkreten Wert haben soll. Der einzige mögliche Wert des Typs `Void` ist `NULL`.

Ganze Zahlen

Ganze Zahlen werden durch den Typ `Int` repräsentiert und haben eine durch die jeweilige Maschine gegebene Größe[4]. Der verfügbare Wertebereich kann über die statischen Member `Int::MIN_VALUE` (betragsmäßig größter darstellbarer negativer Wert) bzw. `Int::MAX_VALUE` (größter darstellbarer positiver Wert) in Erfahrung gebracht werden.

Der Konstruktor `Int()` kann entweder ohne Argumente aufgerufen werden (in diesem Fall wird der Wert 0 geliefert) oder mit einem Argument mit einem der folgenden Typen:

- `Int`: Der Wert des Arguments wird kopiert.

[4]Bei den meisten momentan verbreiteten Architekturen sind dies 32 bit-Zahlen im Zweierkomplement. Der verfügbare Wertebereich ist somit $[-2147483648, 2147483647]$.

- `Float`: Es wird eine Konvertierung von `Float` nach `Int` vorgenommen, wobei der gebrochene Anteil abgeschnitten wird. Bei Überschreitung des verfügbaren Wertebereichs ist das Ergebnis undefiniert.
- `Symbol`: Es wird eine Zahl geliefert, die dem Symbol eindeutig zugeordnet ist.
- `String`: Es wird versucht, die Zeichenkette als ganzzahlige Konstante zu interpretieren. Werden dabei die in Abschnitt 3.2.5 angegebenen Regeln verletzt, wird eine Ausnahme ausgelöst (Abschn. 3.5.3).

Auf den Typ `Int` können die folgenden Operatoren (Abschn. 3.6) angewandt werden:

- Die arithmetischen Operatoren: die Operatoren `+` und `-` in Präfix- und Infixform, die Operatoren `++` und `--` in Präfix- und Postfixform und die Infixoperatoren `*`, `/` und `%`.
- Die Vergleichsoperatoren: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` und `>?`.
- Die logischen Operatoren: `!` und `!!`.
- Die bitweisen Operatoren: `&`, `|`, `^`, `~`, `<<`, `>>` und `>>>`.
- Alle kombinierten Zuweisungen, die mit den oben genannten Operatoren gebildet werden können.

Gleitkommazahlen

Gleitkommazahlen werden durch den Typ `Float` repräsentiert und haben eine durch die jeweilige Maschine gegebene Größe[5]. Der verfügbare Wertebereich kann über die statischen Member `Float::MIN_VALUE` (kleinster darstellbarer normalisierter positiver Wert) bzw. `Float::MAX_VALUE` (größter darstellbarer positiver Wert) in Erfahrung gebracht werden.

Das statische Member `Float::HUGE_VAL` ist abhängig von der Implementierung entweder positiv unendlich oder der größte darstellbare positive Wert. Er wird, ggf. mit negativem Vorzeichen, durch arithmetische Operationen auf Gleitkommawerten genutzt, um einen Überlauf zu signalisieren.

Der Konstruktor `Float()` kann entweder ohne Argumente aufgerufen werden (in diesem Fall wird der Wert 0.0 geliefert) oder mit einem Argument mit einem der folgenden Typen:

- `Float`: Der Wert des Arguments wird kopiert.
- `Int`: Es wird eine Konvertierung von `Int` nach `Float` vorgenommen.
- `String`: Es wird versucht, die Zeichenkette als Gleitkommakonstante zu interpretieren. Werden dabei die in Abschnitt 3.2.5 angegebenen Regeln verletzt, wird eine Ausnahme ausgelöst (Abschn. 3.5.3).

Auf den Typ `Float` können folgende Operatoren (Abschn. 3.6) angewandt werden:

[5] Bei den meisten momentan verbreiteten Architekturen liegt die vom Betrag her kleinste darstellbare Zahl bei $\pm 2.2 \cdot 10^{-308}$, die größte bei $\pm 1.8 \cdot 10^{308}$ und die Genauigkeit bei 15 Dezimalstellen.

- Die arithmetischen Operatoren: die Operatoren `+` und `-` in Präfix- und Infixform, die Operatoren `++` und `--` in Präfix- und Postfixform, und die Infixoperatoren `*`, `/` und `%`.
- Die Vergleichsoperatoren: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` und `>?`.
- Die logischen Operatoren: `!` und `!!`.
- Alle kombinierten Zuweisungen, die mit den oben genannten Operatoren gebildet werden können.

Arithmetik und Typumwandlung

Abhängig von den Typen der Operanden werden arithmetische Berechnungen entweder in `Int` oder `Float` durchgeführt. `Float` wird verwendet, wenn wenigstens einer der Operanden vom Typ `Float` ist, außer bei kombinierten Zuweisungen, bei denen auf der linken Seite ein Wert vom Typ `Int` steht.

Implizite Typumwandlungen für numerische Typen finden in folgenden Fällen statt:

- Ist einer der Operanden vom Typ `Int` und findet die Berechnung in `Float` statt, wird dieser Operand nach `Float` konvertiert.
- Ist einer der Operanden vom Typ `Float` und findet die Berechnung in `Int` statt, wird dieser Operand nach `Int` konvertiert. Der gebrochene Anteil wird dabei abgeschnitten. Wenn bei der Konvertierung ein Überlauf auftritt, ist das Ergebnis undefiniert.

Für Berechnungen in `Int` gelten folgende Regeln:

- Für die interne Darstellung ganzzahliger Werte wird die Zweierkomplementdarstellung verwendet.
- Das Ergebnis ist undefiniert, wenn es im Wertebereich von `Int` nicht darstellbar ist. Eine Ausnahme stellen Additions- und Subtraktionsoperationen dar, bei denen das Ergebnis aus den niederwertigsten Bits eines ganzzahligen Wertes hinreichender Größe besteht.
- Bei der Division durch `0` wird eine Ausnahme ausgelöst.

Für Berechnungen in `Float` gelten folgende Regeln:

- Wenn das Ergebnis nicht exakt darstellbar ist, wird implementierungsabhängig entweder der nächsthöhere oder nächstniedrigere darstellbare Wert verwendet[6].
- Der Betrag des Ergebnisses ist `Float::HUGE_VAL`, wenn es auf Grund eines Überlaufs im Wertebereich von `Float` nicht darstellbar ist. Das Vorzeichen entspricht dem des korrekten Wertes.
- Das Ergebnis ist `0`, wenn es auf Grund eines Unterlaufs im Wertebereich von `Float` nicht mehr darstellbar ist. Ob das Vorzeichen erhalten bleibt, ist implementierungsabhängig.
- Liegt ein Operand nicht im Definitionsbereich der Operation, wird eine Ausnahme ausgelöst.

[6] Die Rundungsrichtung kann sich von Operation zu Operation unterscheiden und ist auch nicht abhängig vom Betrag der Differenz zum nächstniedrigeren oder nächsthöheren darstellbaren Wert.

Symbole

Symbole stellen das dynamische Gegenstück zu Aufzählungskonstanten in statisch typisierten Sprachen dar. Intern werden sie durch eine eindeutige ganze Zahl dargestellt, die über die Funktion `Int()` auch nach außen hin verfügbar ist (siehe Abschn. 3.3.3). Durch diese Repräsentation ist ein sehr schneller Vergleich von Symbolen (im Gegensatz zum Vergleich von Zeichenketten) möglich.

In verschiedenen Instanzen eines OFML–Programmes kann die Konvertierung der gleichen Zeichenkette zu einem Symbol zu verschiedenen für die interne Darstellung verwendeten ganzen Zahlen führen. Daher ist das Ergebnis von Vergleichen auf Symbolen, die eine Ordnung zugrunde legen, in verschiedenen Instanzen eines OFML–Programmes nicht reproduzierbar.

Der Konstruktor `Symbol()` verlangt ein Argument mit einem der folgenden Typen:

- `Symbol`: Der Wert des Arguments wird kopiert.
- `String`: Die Zeichenkette (ohne führendes `@`) wird in ein Symbol umgewandelt. Die Ausdrücke `@foo` und `Symbol("foo")` sind damit äquivalent. Auf diese Weise ist es auch möglich, Zeichenketten in Symbole umzuwandeln, die nicht den Vorschriften für Bezeichner entsprechen, z.B. `Symbol("500 Motels")`.

Auf den Typ `Symbol` können folgende Operatoren (Abschn. 3.6) angewandt werden:

- Die Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` und `>?`.

3.3.4 Referenztypen

Automatische Speicherbereinigung

Die Sprachdefinition von OFML verlangt die Implementierung einer automatischen Speicherbereinigung. Objekte von Referenztypen werden implizit mit dem Aufruf des Konstruktors (Ausnahme s.u.) erzeugt. Es gibt keine Möglichkeit zur expliziten Freigabe von Objekten. Statt dessen können sie vom System automatisch freigegeben werden, sobald keine Referenzen auf das Objekt mehr existieren. Es ist jedoch nicht festgelegt, wann und ob überhaupt nicht länger referenzierte Objekte freigegeben werden[7].

Die Sprachdefinition stellt die Art und Weise der Implementierung der automatischen Speicherbereinigung frei.

[7] Dies variiert stark mit dem für die automatische Speicherbereinigung verwendeten Algorithmus. Bei der Verwendung von Referenzzählern werden Objekte in der Regel freigegeben, sobald sie nicht mehr referenziert werden. Allerdings gibt eine nur auf Referenzzählern basierende Speicherbereinigung Datenstrukturen mit Zyklen nicht frei. Diese Zyklen sind daher durch den Programmierer zu brechen, bevor die letzte Referenz auf eine derartige Datenstruktur freigegeben wird.

Andere Verfahren, bei denen ausgehend von einer bekannten Menge referenzierter Objekte alle erreichbaren (und damit referenzierten) Objekte bestimmt werden, verzögern die Freigabe nicht mehr referenzierter Objekte und geben bei der Verwendung konservativer Algorithmen unter Umständen auch nicht alle Objekte frei.

Eine Kombination aus beiden Verfahren ist ebenfalls denkbar.

Operatoren auf Referenztypen

Das Verhalten der Operatoren, die innerhalb von Ausdrücken verwendet werden können, ist für die einfachen Typen fest definiert. Liefert der Operand eines unären Operators oder der linke Operand eines binären Operators einen Referenztyp, so wird für diesen eine für den Operator spezifische instanzbezogene Methode aufgerufen. Diese Methoden können für Klassen frei definiert werden. Ausnahmen sind die Operatoren `$` (Symbol–Auflösungsoperator), `!` (logische Negation), `instanceof` (Typprüfung), `>?` (Maximum), `<?` (Minimum), `&&` (logisches Und), `||` (logisches Oder), `?:` (bedingter Ausdruck), `=` (Zuweisung) und `,` (Komma–Operator), deren Verhalten entweder für Referenztypen fest vorgegeben ist, die auf andere Operatoren abgebildet werden oder die grundsätzlich auf Referenztypen nicht angewendet werden können.

Die Operatormethoden, die in Klassendefinitionen zu verwenden sind, sind in Abschnitt 3.6 unter dem jeweiligen Operator beschrieben.

Sequenztypen

Sequenztypen sind alle die Referenztypen, die als eine Folge von Objekten angesehen werden können. Dazu müssen sie die folgenden Bedingungen erfüllen:

- Die Methode `size()` muß definiert sein und einen nichtnegativen `Int`–Wert *size* liefern.
- Die Index–Operatoren *operator[](pIdx(Int))* und *operator[](pIdx(Int), pValue(Object))* müssen für jeden ganzzahligen Index *pIdx* im Bereich $[0, size)$ definiert sein.
- Der sequentielle Zugriff über die Index–Operatoren, vorwärts sowie rückwärts, sollte konstante Zeit benötigen.

Von den in OFML vordefinierten Typen sind `String`, `Vector` und `List` Sequenztypen.

3.4 Vordefinierte Referenztypen

Die folgenden Abschnitte beschreiben die in OFML vordefinierten Referenztypen, nutzerdefinierte Klassen werden in Abschnitt 3.8 beschrieben.

Alle vordefinierten Referenztypen sind in dem Paket `::cobra::lang` definiert.

3.4.1 Der Metatyp `Type`

Alle Typen, einschließlich des Typs `Type`, sind Instanzen des Typs `Type`.

Typenamen in OFML sind Variablen, die eine Referenz auf eine Instanz von `Type` enthalten.

Auf Instanzen von `Type` können die folgenden Operatoren (siehe Abschn. 3.6) und Methoden (siehe Abschn. 3.8.4) angewendet werden:

operator()(parameters) → Object
: Der für alle Typen definierte Funktionsaufrufoperator wird als Konstruktor bezeichnet. Der Konstruktor erzeugt eine Instanz des Typs, für den er aufgerufen wird und ruft für diese Instanz, wenn vorhanden, die Initialisierungsmethode `initialize()` auf. Die an den Konstruktor übergebenen Argumente werden an die Initialisierungsmethode weitergeleitet.

getName() → Symbol
: Die Methode *getName*() gibt den einfachen Namen des Typs als `Symbol` zurück.

getFullName() → String
: Die Methode *getFullName*() gibt den voll qualifizierten Namen des Typs als Zeichenkette zurück.

subClassOf(pType(Type)) → Int
: Die Methode *subClassOf*() gibt 1 zurück, wenn der Typ, für den sie aufgerufen wird, entweder mit dem als Argument übergebenen Typ identisch oder von diesem abgeleitet ist. Andernfalls gibt sie 0 zurück. Ist das Argument nicht vom Typ `Type`, wird eine Ausnahme ausgelöst.

3.4.2 Funktionen

Funktionen in OFML werden durch die Typen `Func` und `CFunc` repräsentiert. `Func` ist der Typ von in OFML definierten Funktionen, während `CFunc` der Typ von vordefinierten Funktionen ist. Neben den für alle Typen verfügbaren Operatoren (siehe Abschn. 3.3.2) implementieren `Func` und `CFunc` den Funktionsaufruf–Operator „`()`“.

3.4.3 Zeichenketten

Zeichenketten werden durch den Typ `String` repräsentiert und werden intern durch eine Folge von 8 bit–Werten dargestellt, wobei jeder Wert einem Zeichen entspricht. Ob das Null–Zeichen (`'\0'`) Bestandteil einer Zeichenkette sein darf, ist implementierungsabhängig.

Der Konstruktor `String()` kann entweder ohne Argumente aufgerufen werden (in diesem Fall wird die leere Zeichenkette `""` geliefert) oder mit einem Argument mit einem der folgenden Typen:

- `String`: Es wird eine Kopie der als Argument übergebenen Zeichenkette erzeugt.
- `Symbol`: Es wird eine neue Zeichenkette erzeugt, deren Inhalt gleich der durch das Symbol repräsentierten Zeichenfolge ist.
- `Int`, `Float`: Es wird eine neue Zeichenkette erzeugt, die das Ergebnis der Konvertierung der Zahlen in eine Zeichenkette enthält.

Eine Zeichenkettenkonstante in einem Ausdruck bewirkt einen impliziten Aufruf des Konstruktors `String()`, wobei die Zeichenkette als Argument übergeben wird.

Auf den Typ `String` können folgende Operatoren (Abschn. 3.6) und Methoden (Abschn. 3.8.4) angewandt werden:

operator==(pString(String)) → *Int*
operator!=(pString(String)) → *Int*
operator<(pString(String)) → *Int*
operator<=(pString(String)) → *Int*
operator>=(pString(String)) → *Int*
operator>(pString(String)) → *Int*

Das Ergebnis ist 1, wenn der zeichenweise Vergleich beider Zeichenketten Gleichheit ergibt. Andernfalls ist das Ergebnis 0. Der zeichenweise Vergleich zweier Zeichenketten ist auf Seite 37 unter der Funktion *compare*() beschrieben.

operator+(pString(String)) → *String*

Der Additions–Operator `+` erwartet auf der rechten Seite eine Zeichenkette. Andernfalls wird eine Ausnahme ausgelöst. Er erzeugt dann eine neue Zeichenkette, die aus der Verknüpfung der Zeichenketten auf der linken und der rechten Seite des Operators besteht.

operator+=(pString(String)) → *String*

Der Additions–Operator `+=` erwartet auf der rechten Seite eine Zeichenkette. Andernfalls wird eine Ausnahme ausgelöst. Er fügt dann die Zeichenkette auf der rechten Seite des Operators an die Zeichenkette auf der linken Seite an. Das Ergebnis ist die zusammengesetzte Zeichenkette.

operator[](pIdx(Int)) → *Int*
operator[](pIdx(Int), pChar(Int))

Die Index–Operatoren erwarten als Index $pIdx$ einen Wert vom Typ `Int`. Andernfalls wird eine Ausnahme ausgelöst. Die Länge der Zeichenkette sei len. Ist $pIdx < 0$, wird $pIdx = pIdx + len$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq len$, wird eine Ausnahme ausgelöst. Andernfalls wird das Zeichen auf Position $pIdx$ als positiver `Int` zurückgegeben.

Wird der Index–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite Form des Index–Operators), muß der Ausdruck auf der rechten Seite des Zuweisungsoperators einen Wert vom Typ `Int` liefern. Andernfalls wird eine Ausnahme ausgelöst. Dieser Wert modulo 2^8 wird dem Zeichen auf Position $pIdx$ zugewiesen.

operator[:](pBegin(Int), pEnd(Int)) → *String*
operator[:](pBegin(Int), pEnd(Int), pChar(Int))
operator[:](pBegin(Int), pEnd(Int), pString(String))

Der Bereichs–Operator `[:]` erwartet zwei Indizes $pBegin$ und $pEnd$ vom Typ `Int`. Die Länge der Zeichenkette sei len. Ist $pBegin < 0$, wird $pBegin = pBegin + len$ gesetzt. Ebenso wird $pEnd = pEnd + len$ gesetzt, wenn $pEnd < 0$ ist. Ist danach $pBegin < 0 \vee pBegin > len$ oder $pEnd < 0 \vee pEnd > len$ oder $pBegin > pEnd$, wird eine Ausnahme ausgelöst. Andernfalls wird eine Teilzeichenkette, beginnend mit Position $pBegin$ und endend mit Position $pEnd-1$, zurückgegeben.

Wird der Bereichs–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite und dritte Form des Bereichs–Operators), muß der Wert des Ausdrucks auf der rechten Seite des Zuweisungsoperators entweder ein `Int`–Wert oder eine Zeichenkette beliebiger Länge sein. Andernfalls wird eine Ausnahme ausgelöst. Die durch die Indizes $pBegin$ und $pEnd$ spezifizierte Teilzeichenkette wird durch das durch den ganzzahligen Wert angegebene Zeichen modulo 2^8 oder durch die Zeichenkette ersetzt.

operator!!() → Int

Der Test–Operator `!!` liefert 1, wenn die Länge der Zeichenkette ungleich Null ist. Andernfalls liefert er 0.

getAt(pIdx(Int)) → Int

Ist $pIdx < 0$, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$, wird eine Ausnahme ausgelöst. Andernfalls wird das Zeichen auf Position $pIdx$ als positiver `Int` zurückgegeben.

setAt(pIdx(Int), pChar(Int))

Ist $pIdx < 0$, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$ oder $pChar < 0 \vee pChar \geq 2^8$, wird eine Ausnahme ausgelöst. Andernfalls wird $pChar$ dem Zeichen auf Position $pIdx$ zugewiesen.

size() → Int

liefert die Anzahl der Zeichen in der Zeichenkette.

empty() → Int

gibt 1 zurück, wenn die Länge der Zeichenkette Null ist, andernfalls 0.

resize(pSize(Int), pChar(Int) = ' ')

Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird die neue Länge der Zeichenkette auf $pSize$ gesetzt. Ist der neue Wert größer als die alte Länge, wird mit dem Zeichen $pChar$ aufgefüllt.

append(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)

Wenn $pPos < 0 \vee pPos > pString.size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls ist $pLen = \min(pLen, pString.size() - pPos)$. Die Teilzeichenkette von $pString$ der Länge $pLen$ wird dann, beginnend ab Position $pPos$, an die Zeichenkette angefügt.

append(pNum(Int), pChar(Int) = ' ')

Wenn $pNum < 0$ oder $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das Zeichen $pChar$ $pNum$–mal an die Zeichenkette angefügt.

assign(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)

Wenn $pPos < 0 \vee pPos > pString.size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls ist $pLen = \min(pLen, pString.size() - pPos)$. Die Zeichenkette wird dann auf die Teilzeichenkette von $pString$ gesetzt, die an Position $pPos$ beginnt und eine Länge von $pLen$ hat.

assign(pNum(Int), pChar(Int) = ' ')

Wenn $pNum < 0$ oder $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird die Zeichenkette auf eine Folge von $pLen$ mal dem Zeichen $pChar$ gesetzt.

insert(pPos1(Int), pString(String), pPos2(Int) = 0, pLen(Int) = Int::MAX_VALUE)

Wenn $pPos1 < 0 \vee pPos1 > size()$ oder $pPos2 < 0 \vee pPos2 > pString.size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, pString.size() - pPos2)$ gesetzt. Anschließend wird die Teilzeichenkette aus $pString$, beginnend auf Position $pPos2$ und mit der Länge $pLen$, an Position $pPos1$ eingefügt.

insert(pPos(Int), pNum(Int), pChar(Int) = ' ')
Wenn $pPos < 0 \vee pPos > size()$ oder $pNum < 0$ oder $pChar < 0 \vee pChar >= 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das Zeichen $pChar$ $pNum$–mal an Position $pPos$ eingefügt.

remove(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
Wenn $pPos < 0 \vee pPos > size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, size() - pPos)$ gesetzt. Anschließend werden ab Position $pPos$ $pLen$ Zeichen entfernt.

replace(pPos1(Int), pLen1(Int), pString(String), pPos2(Int) = 0, pLen2(Int) = Int::MAX_VALUE)

Wenn $pPos1 < 0 \vee pPos1 > size()$ oder $pPos2 < 0 \vee pPos2 > pString.size()$ oder $pLen1 < 0$ oder $pLen2 < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls werden $pLen1 = \min(pLen1, size() - pPos1)$ und $pLen2 = \min(pLen2, pString.size() - pPos2)$ gesetzt. Anschließend werden ab Position $pPos1$ $pLen1$ Zeichen durch eine Teilzeichenkette aus $pString$, die auf Position $pPos2$ beginnt und $pLen2$ Zeichen lang ist, ersetzt.

replace(pPos(Int), pLen(Int), pNum(Int), pChar(Int) = ' ')
Wenn $pPos < 0 \vee pPos > size()$ oder $pLen < 0$ oder $pNum < 0$ oder $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, size() - pPos)$ gesetzt. Anschließend werden ab Position $pPos$ $pLen$ Zeichen durch $pNum$ neue Zeichen $pChar$ ersetzt.

swap(pString(String))
vertauscht den Inhalt zweier Zeichenketten.

find(pString(String), pPos(Int) = 0) → Int
Wenn möglich, wird der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res + pString.size() \leq size()$ und
$getAt(res + i) = pString.getAt(i)$ für alle $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

find(pChar(Int), pPos(Int) = 0) → Int
Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res < size()$ und $getAt(res) = pChar$
Andernfalls wird -1 zurückgegeben.

rfind(pString(String), pPos(Int) = Int::MAX_VALUE) → Int
Wenn möglich, wird der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge res + pString.size() \leq size()$ und
$getAt(res + i) = pString[i]$ für alle $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

rfind(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int
Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge res < size()$ und $getAt(res) = pChar$
Andernfalls wird -1 zurückgegeben.

findFirstOf(pString(String), pPos(Int) = 0) → Int

Wenn möglich, wird der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res < size()$ und
$getAt(res) = pString.getAt(i)$ für wenigstens ein $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

findFirstOf(pChar(Int), pPos(Int) = 0) → Int

Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res < size()$ und $getAt(res) = pChar$ Andernfalls wird -1 zurückgegeben.

findLastOf(pString(String), pPos(Int) = Int::MAX_VALUE) → Int

Wenn möglich, wird der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge pPos < size()$ und
$getAt(res) = pString.getAt(i)$ für wenigstens ein $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

findLastOf(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int

Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge pPos < size()$ und $getAt(res) = pChar$.
Andernfalls wird -1 zurückgegeben.

findFirstNotOf(pString(String), pPos(Int) = 0) → Int

Wenn möglich, wird der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res < size()$ und
$getAt(res) = pString.getAt(i)$ für kein $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

findFirstNotOf(pChar(Int), pPos(Int) = 0) → Int

Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der kleinste Wert res zurückgegeben, für den gilt:
$res \geq pPos \wedge res < size()$ und $getAt(res) \neq pChar$
Andernfalls wird -1 zurückgegeben.

findLastNotOf(pString(String), pPos(Int) = Int::MAX_VALUE) → Int

Wenn möglich, wird der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge pPos < size()$ und
$getAt(res) = pString.getAt(i)$ für kein $i \geq 0 \wedge i < pString.size()$
Andernfalls wird -1 zurückgegeben.

findLastOf(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int

Wenn $pChar < 0 \vee pChar \geq 2^8$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird, wenn möglich, der größte Wert res zurückgegeben, für den gilt:
$res \leq pPos \wedge pPos < size()$ und $getAt(res) \neq pChar$
Andernfalls wird -1 zurückgegeben.

substr(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE) → String

Wenn $pPos < 0 \vee pPos > size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, size() - pPos)$ gesetzt. Anschließend wird eine neue Zeichenkette

erzeugt und zurückgegeben, deren Inhalt der bei Position $pPos$ beginnenden Teilzeichenkette der Länge $pLen$ entspricht.

toUpper(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
Wenn $pPos < 0 \vee pPos > size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, size() - pPos)$ gesetzt. Anschließend werden, wenn $pLen > 0$, ab der Position $pPos$ bis einschließlich der Position $pPos + pLen - 1$ alle Kleinbuchstaben in Großbuchstaben umgewandelt.

toLower(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
Wenn $pPos < 0 \vee pPos > size()$ oder $pLen < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird $pLen = \min(pLen, size() - pPos)$ gesetzt. Anschließend werden, wenn $pLen > 0$, ab der Position $pPos$ bis einschließlich der Position $pPos + pLen - 1$ alle Großbuchstaben in Kleinbuchstaben umgewandelt.

compare(pPos1(Int), pLen1(Int), pString(String), pPos2(Int) = 0, pLen2(Int) = Int::MAX_VALUE) $\rightarrow$ *Int*

Führt einen zeichenweisen Vergleich der beiden Zeichenketten $pStr1 = substr(pPos1, pLen1)$ und $pStr2 = pString.substr(pPos2, pLen2)$ durch. Das Ergebnis ist -1, wenn $pStr1$ kleiner als $pStr2$ ist. Es ist $+1$, wenn $pStr1$ größer als $pStr2$ ist. Und es ist 0, wenn $pStr1$ und $pStr2$ gleich sind.

Beim zeichenweisen Vergleich zweier Zeichenketten werden die Zeichen beider Zeichenketten, beginnend ab Position 0, paarweise miteinander verglichen. Der Vergleich bricht ab, sobald zwei Zeichen nicht gleich sind oder das Ende wenigstens einer Zeichenkette erreicht wurde. Im ersten Fall ist das Ergebnis des Vergleichs -1, wenn die Kodierung des Zeichens der ersten (oder linken) Zeichenkette kleiner ist als die Kodierung des Zeichens der zweiten (oder rechten) Zeichenkette. Entsprechend ist das Ergebnis $+1$, wenn die Kodierung des Zeichens der ersten Zeichenkette größer ist als die des Zeichens der zweiten Zeichenkette. Im zweiten Fall ist das Ergebnis 0, wenn gleichzeitig das Ende beider Zeichenketten erreicht wurde. Es ist -1, wenn das Ende der ersten Zeichenkette erreicht wurde und $+1$, wenn das Ende der zweiten Zeichenkette erreicht wurde.

compare(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE) $\rightarrow$ *Int*
Entspricht dem Aufruf von $compare(0, \texttt{Int::MAX\_VALUE}, pString, pPos, pLen)$.

getHashValue() $\rightarrow$ *Int*
Die Methode $getHashValue()$ gibt einen Hash–Wert für die Zeichenkette zurück. Sie gibt für entsprechend des Operators `==` gleiche Zeichenketten immer den gleichen Hash–Wert zurück, während sie den gleichen Hash–Wert auch für nicht gleiche Zeichenketten zurückgeben kann.

3.4.4 Vektoren

Der Typ `Vector` stellt eindimensionale Vektoren dar. Mehrdimensionale Felder können aus Vektoren von Vektoren gebildet werden, wobei die Dimensionen der einzelnen Vektoren nicht identisch sein müssen.

Wahlfreier Zugriff auf einzelne Vektorelemente über ihren Index sowie Einfüge- und Löschoperationen am Ende des Vektors erfordern konstante Zeit. Für Einfüge- und Löschoperationen am

Anfang oder in der Mitte des Vektors ist die benötigte Zeit proportional zur Anzahl der folgenden Vektorelemente.

Einfügeoperationen können zusätzlich Zeit zur Reallozierung des Vektors erfordern.

Vektoren können auf zwei verschiedene Arten erzeugt werden:

- Durch Aufruf des `Vector`–Konstruktors. *Vector(pSize(Int), ...)* erzeugt einen Vektor mit *pSize* Elementen, die mit `NULL` initialisiert werden. Die Angabe eines zweiten Arguments $pSize2$ vom Typ `Int` initialisiert den Vektor mit Vektoren der Größe $pSize2$ und erzeugt somit ein zweidimensionales Feld. Dies kann rekursiv durch Angabe von drei und mehr Argumenten vom Typ `Int` zur Erzeugung von drei- und mehrdimensionalen Feldern fortgeführt werden.
- Durch Angabe der Elemente in eckigen Klammern, getrennt durch Komma. Für jedes Element kann dabei ein beliebiger Zuweisungsausdruck stehen, dessen Ergebnis zur Initialisierung des Elements benutzt wird.

special-ctor:
`[` *arg-expr-list*$_{opt}$ `]`
arg-expr-list:
assign-expr
arg-expr-list `,` *assign-expr*

Auf den Typ `Vector` können folgende Operatoren (Abschn. 3.6) und Methoden (Abschn. 3.8.4) angewandt werden:

operator==(pSeq(Object)) $\rightarrow$ *Int*
operator!=(pSeq(Object)) $\rightarrow$ *Int*

Die Vergleichsoperatoren `==` und `!=` mit einem Vektor *vec* auf der linken Seite erwarten eine Instanz *pSeq* eines Sequenztyps (siehe Abschn. 3.3.4) auf der rechten Seite. Der Vektor *vec* und die Sequenz *pSeq* gelten als gleich, wenn:

- Die Länge von *vec* gleich der Länge von *pSeq* ist.
- Für jeden ganzzahligen Index *idx* im Bereich von $[0, vec.size())$ der Vergleich von $vec[idx]$ mit $pSeq[idx]$ mittels des Operators `==` *wahr* ($\neq 0$) ergibt. Der erste Vergleich von Elementen, der eine Ausnahme auslöst oder nicht *wahr* ergibt, bricht den Vergleich des Vektors *vec* mit der Sequenz *pSeq* ab.

operator[](pIdx(Int)) $\rightarrow$ *Object*
operator[](pIdx(Int), pObj(Object))

Der Index–Operator `[]` erwartet als Index $pIdx$ einen Wert vom Typ `Int`. Die Länge des Vektors sei *len*. Ist $pIdx < 0$, wird $pIdx = pIdx + len$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq len$, wird eine Ausnahme ausgelöst. Andernfalls wird das durch $pIdx$ indizierte Vektorelement zurückgegeben.

Wird der Index–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite Form des Index–Operators), wird das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators dem durch $pIdx$ indizierten Vektorelement zugewiesen.

operator[:](pBegin(Int), pEnd(Int)) $\rightarrow$ *Vector*
operator[:](pBegin(Int), pEnd(Int), pSeq(Object))

Der Bereichs–Operator `[:]` erwartet zwei Indizes $pBegin$ und $pEnd$ vom Typ `Int`. Die Länge des Vektors sei len. Ist $pBegin < 0$, wird $pBegin = pBegin + len$ gesetzt. Ebenso wird $pEnd = pEnd + len$ gesetzt, wenn $pEnd < 0$ ist. Ist danach $pBegin < 0 \vee pBegin > len$ oder $pEnd < 0 \vee pEnd > len$ oder $pBegin > pEnd$, wird eine Ausnahme ausgelöst. Andernfalls wird ein Vektor zurückgegeben, der aus den durch $pBegin$ bis $pEnd-1$ indizierten Elementen des ursprünglichen Vektors besteht.

Wird der Bereichs–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite Form des Bereichs–Operators), muß das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators eine Sequenz (siehe Abschn. 3.3.4) beliebiger Länge sein. Die durch $pBegin$ bis $pEnd - 1$ indizierten Elemente des Vektors werden durch alle Elemente der Sequenz ersetzt.

operator!!() $\rightarrow$ *Int*
Der Test–Operator `!!` liefert 1, wenn die Länge des Vektors ungleich Null ist. Andernfalls liefert er 0.

getAt(pIdx(Int)) $\rightarrow$ *Object*
Wenn $pIdx < 0$ ist, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$, wird eine Ausnahme ausgelöst. Andernfalls wird das Element mit dem Index $pIdx$ zurückgegeben.

setAt(pIdx(Int), pObject(Object))
Wenn $pIdx < 0$ ist, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$, wird eine Ausnahme ausgelöst. Andernfalls wird $pObject$ dem Element mit dem Index $pIdx$ zugewiesen.

size() $\rightarrow$ *Int*
Es wird die Anzahl der Elemente des Vektors zurückgegeben.

empty() $\rightarrow$ *Int*
Wenn $size() = 0$ ist, wird 1 zurückgegeben, andernfalls 0.

front() $\rightarrow$ *Object*
Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das erste Element des Vektors zurückgegeben.

back() $\rightarrow$ *Object*
Wenn $size() == 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das letzte Element des Vektors zurückgegeben.

pushBack(pObject(Object))
Es wird $pObject$ als letztes Element an den Vektor angefügt.

popBack() $\rightarrow$ *Object*
Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das letzte Element des Vektors von diesem entfernt und zurückgegeben.

insert(pPos(Int), pNum(Int) = 1, pObj(Object))
Wenn $pPos < 0 \vee pPos > size()$ oder $pNum < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird *pObj* $pNum$–mal an dem Index $pPos$ eingefügt.

erase(pBegin(Int), pEnd(Int) = pBegin + 1)
Wenn $pBegin < 0 \vee pBegin > size()$ oder $pEnd < 0 \vee pEnd > size()$ oder $pBegin > pEnd$ ist, wird eine Ausnahme ausgelöst. Andernfalls werden, wenn $pBegin < pEnd$, die durch $pBegin$ bis $pEnd - 1$ indizierten Elemente aus dem Vektor gelöscht.

swap(pVec(Vector))
Wenn $pVec$ keine Instanz vom Typ `Vector` ist, wird eine Ausnahme ausgelöst. Andernfalls werden die Inhalte beider Vektoren ausgetauscht.

3.4.5 Listen

Der Typ `List` stellt doppelt verkettete Listen dar.

Sequentieller Zugriff auf einzelne Elemente einer Liste, sowohl vorwärts als auch rückwärts, sowie Einfüge- und Löschoperationen an beliebiger Position, erfordern konstante Zeit. Die für den wahlfreien Zugriff auf Listenelemente benötigte Zeit ist höchstens proportional zum Minimum des Abstands zum Anfang oder zum Ende der Liste.

Listen können auf zwei verschiedene Arten erzeugt werden:

- Durch Aufruf des `List`–Konstruktors. Dieser kann mit Null oder mehr Argumenten aufgerufen werden. Die Argumente bilden die einzelnen Elemente der Liste.
- Durch Angabe der Elemente in `@()`, getrennt durch Komma. Für jedes Element kann dabei ein beliebiger Zuweisungsausdruck stehen, dessen Ergebnis zur Initialisierung des Elements benutzt wird.

special-ctor:
`@(` *arg-expr-list*$_{opt}$ `)`
arg-expr-list:
assign-expr
arg-expr-list `,` *assign-expr*

Auf den Typ `List` können folgende Operatoren (Abschn. 3.6) und Methoden (Abschn. 3.8.4) angewandt werden:

operator==(pSeq(Object)) → *Int*
operator!=(pSeq(Object)) → *Int*

Die Vergleichsoperatoren `==` und `!=` mit einer Liste *list* auf der linken Seite erwarten eine Instanz *pSeq* eines Sequenztyps (siehe Abschn. 3.3.4) auf der rechten Seite. Die Liste *list* und die Sequenz *pSeq* gelten als gleich, wenn:

- Die Länge von *list* gleich der Länge von *pSeq* ist.

- Für jeden ganzzahligen Index idx im Bereich von $[0, list.size())$ der Vergleich von $list[idx]$ mit $pSeq[idx]$ mittels des Operators `==` *wahr* ($\neq 0$) ergibt. Der erste Vergleich von Elementen, der eine Ausnahme auslöst oder nicht *wahr* ergibt, bricht den Vergleich der Liste *list* mit der Sequenz *pSeq* ab.

operator[](pIdx(Int)) → Object
operator[](pIdx(Int), pObject(Object))

Der Index–Operator `[]` erwartet als Index $pIdx$ einen Wert vom Typ `Int`. Die Länge der Liste sei *len*. Ist $pIdx < 0$, wird $pIdx = pIdx + len$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq len$, wird eine Ausnahme ausgelöst. Andernfalls wird das durch $pIdx$ indizierte Listenelement zurückgegeben.

Wird der Index–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite Form des Index–Operators), wird das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators dem durch $pIdx$ indizierten Listenelement zugewiesen.

operator[:](pBegin(Int), pEnd(Int)) → List
operator[:](pBegin(Int), pEnd(Int), pSeq(Object))

Der Bereichs–Operator `[:]` erwartet zwei Indizes $pBegin$ und $pEnd$ vom Typ `Int`. Die Länge der Liste sei *len*. Ist $pBegin < 0$, wird $pBegin = pBegin + len$ gesetzt. Ebenso wird $pEnd = pEnd + len$ gesetzt, wenn $pEnd < 0$ ist. Ist danach $pBegin < 0 \vee pBegin > len$ oder $pEnd < 0 \vee pEnd > len$ oder $pBegin > pEnd$, wird eine Ausnahme ausgelöst. Andernfalls wird eine Liste zurückgegeben, die aus den durch $pBegin$ bis $pEnd - 1$ indizierten Elementen der ursprünglichen Liste besteht.

Wird der Bereichs–Operator auf der linken Seite des Zuweisungsoperators verwendet (zweite Form des Bereichs–Operators), muß das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators eine Sequenz (siehe Abschn. 3.3.4) beliebiger Länge sein. Die durch $pBegin$ bis $pEnd - 1$ indizierten Elemente der Liste werden durch alle Elemente der Sequenz ersetzt.

operator!!() → Int
Der Test–Operator `!!` liefert 1, wenn die Länge der Liste ungleich Null ist. Andernfalls liefert er 0.

getAt(pIdx(Int)) → Object
Wenn $pIdx < 0$ ist, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$, wird eine Ausnahme ausgelöst. Andernfalls wird das Element mit dem Index $pIdx$ zurückgegeben.

setAt(pIdx(Int), pObject(Object))
Wenn $pIdx < 0$ ist, wird $pIdx = pIdx + size()$ gesetzt. Ist danach $pIdx < 0 \vee pIdx \geq size()$, wird eine Ausnahme ausgelöst. Andernfalls wird *pObject* dem Element mit dem Index $pIdx$ zugewiesen.

size() → Int
Es wird die Anzahl der Elemente der Liste zurückgegeben.

empty() → Int
Wenn $size() = 0$ ist, wird 1 zurückgegeben, andernfalls 0.

front() $\rightarrow$ *Object*
: Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das erste Element der Liste zurückgegeben.

back() $\rightarrow$ *Object*
: Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das letzte Element der Liste zurückgegeben.

pushFront(pObject(Object))
: Es wird *pObject* als erstes Element an die Liste angefügt.

pushBack(pObject(Object))
: Es wird *pObject* als letztes Element an die Liste angefügt.

popFront() $\rightarrow$ *Object*
: Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das erste Element der Liste von dieser entfernt und zurückgegeben.

popBack() $\rightarrow$ *Object*
: Wenn $size() = 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird das letzte Element der Liste von dieser entfernt und zurückgegeben.

insert(pPos(Int), pNum(Int) = 1, pObj(Object))
: Wenn $pPos < 0 \vee pPos > size()$ oder $pNum < 0$ ist, wird eine Ausnahme ausgelöst. Andernfalls wird *pObj* $pNum$–mal an dem Index *pPos* eingefügt.

erase(pBegin(Int), pEnd(Int) = pBegin + 1)
: Wenn $pBegin < 0 \vee pBegin > size()$ oder $pEnd < 0 \vee pEnd > size()$ oder $pBegin > pEnd$ ist, wird eine Ausnahme ausgelöst. Andernfalls werden, wenn $pBegin < pEnd$ ist, die durch *pBegin* bis $pEnd - 1$ indizierten Elemente aus der Liste gelöscht.

swap(pList(List))
: Wenn *pList* keine Instanz vom Typ `List` ist, wird eine Ausnahme ausgelöst. Andernfalls werden die Inhalte beider Listen ausgetauscht.

splice(pPos(Int), pList(List), pBegin(Int) = 0, pEnd(Int) = pBegin + 1)
: Wenn $pPos < 0 \vee pPos > size()$ oder $pBegin < 0 \vee pBegin > pList.size()$ oder $pEnd < 0 \vee pEnd > pList.size()$ oder $pBegin > pEnd$ ist, wird eine Ausnahme ausgelöst. Andernfalls werden, wenn $pBegin < pEnd$ ist, die Elemente aus *pList*, die durch *pBegin* bis $pEnd - 1$ indiziert werden, aus *pList* entfernt und in der gleichen Reihenfolge ab *pPos* eingefügt.

remove(pObj(Object))
: Vergleicht jedes Element der Liste, beginnend mit dem ersten und sequentiell aufsteigend bis zum letzten, unter Verwendung des Operators `==` mit *pObj*, wobei das Listenelement auf der linken und *pObj* auf der rechten Seite des Vergleichsoperators erscheint. Wird durch einen Vergleich eine Ausnahme ausgelöst, kehrt die Funktion sofort zurück. Andernfalls entfernt sie das aktuelle Listenelement aus der Liste, wenn der Vergleich *wahr* ($\neq 0$) ergab.

removeIf(pPred(Func))
: Das Argument *pPred* muß eine Funktion sein, die ein Argument erwartet und entweder *wahr* ($\neq 0$) oder *falsch* (0) zurückgibt (unäres Prädikat).

Die Methode *removeIf*() ruft für jedes Element der Liste, beginnend mit dem ersten und sequentiell aufsteigend bis zum letzten, die Funktion *pPred* auf, wobei das Listenelement als Argument an die Funktion übergeben wird. Wird durch die Funktion eine Ausnahme ausgelöst, kehrt *removeIf()* sofort zurück. Ist der Rückgabewert der Funktion nicht vom Typ `Int`, wird eine Ausnahme ausgelöst. Andernfalls entfernt sie das aktuelle Listenelement aus der Liste, wenn der Rückgabewert der Funktion *wahr* ($\neq 0$) ist.

unique()

Entfernt von jeder Folge aufeinanderfolgender gleicher Elemente alle außer dem ersten. Vergleicht dazu unter Verwendung des Operators `==` das jeweils aktuelle Element mit dem unmittelbar folgenden Element, wobei das aktuelle Element auf der linken und das folgende Element auf der rechten Seite des Vergleichsoperators erscheint. Wird durch den Vergleichsoperator eine Ausnahme ausgelöst, kehrt die Funktion sofort zurück. Andernfalls wird das folgende Element gelöscht, wenn der Vergleich *wahr* ergab oder das folgende Element zum aktuellen Element gemacht, wenn der Vergleich *falsch* ergab.

unique(pPred(Func))

Das Argument *pPred* muß eine Funktion sein, die zwei Argumente erwartet und *wahr* ($\neq 0$) zurückgibt, wenn beide Argumente gleich sind oder *falsch* (0), wenn sie verschieden sind (binäres Prädikat). Ist der Rückgabewert nicht vom Typ `Int`, wird durch *unique*() eine Ausnahme ausgelöst.

Die Methode *unique*() mit *pPred* als Argument verhält sich identisch zu der ohne Argument, mit dem Unterschied, daß statt des Operators `==` die Funktion *pPred* aufgerufen wird, dem das aktuelle Element als erstes und das folgende Element als zweites Argument übergeben wird.

merge(pList(List))

Die Methode *merge*() fügt zwei aufsteigend sortierte Listen zu einer sortierten Liste zusammen. Sie verwendet den Operator `<`, auf dessen linker Seite jeweils ein Element von *pList* und auf dessen rechter Seite jeweils ein Element von *self* übergeben wird. Wird durch den Vergleichsoperator eine Ausnahme ausgelöst, kehrt *merge*() sofort zurück und der Inhalt beider Listen ist undefiniert. Die Argumentliste *pList* ist nach Rückkehr von *merge*() leer. Sind in beiden Listen äquivalente Elemente, befinden sich in der Ergebnisliste die Elemente aus *self* vor denen aus *pList*. Die Reihenfolge von Elementen einer Liste bleibt in der Ergebnisliste erhalten.

merge(pList(List), pPred(Func))

Das zweite Argument *pPred* muß eine Funktion sein, die zwei Argumente erwartet und *wahr* ($\neq 0$) zurückgibt, wenn das erste Argument kleiner als das zweite Argument ist, andernfalls *falsch* (0) (binäres Prädikat). Ist der Rückgabewert nicht vom Typ `Int`, wird durch *merge*() eine Ausnahme ausgelöst.

Die Methode *merge*() mit zwei Argumenten verhält sich identisch zu der mit nur einem Argument, nur daß anstatt des Operators `<` die Funktion *pPred* aufgerufen wird.

sort()

Sortiert die Liste unter Verwendung des Vergleichsoperators `<`, der für beliebige Elemente der Liste aufgerufen werden kann. Wird durch *sort*() direkt oder indirekt eine Ausnahme

ausgelöst, ist der Inhalt der Liste undefiniert. Die Reihenfolge gleicher Elemente in der unsortierten Liste bleibt in der sortierten Liste erhalten. Die Komplexität von *sort*() ist ungefähr $size() \cdot \log(size())$ Vergleichsoperationen.

sort(pPred(Func))
: Das Argument *pPred* muß eine Funktion sein, die zwei Argumente erwartet und *wahr* ($\neq 0$) zurückgibt, wenn das erste Argument kleiner als das zweite Argument ist, andernfalls *falsch* (0) (binäres Prädikat). Ist der Rückgabewert nicht vom Typ `Int`, wird durch *sort*() eine Ausnahme ausgelöst.

  Die Methode *sort*() mit einem Argument verhält sich identisch zu der ohne Argument, nur daß statt des Vergleichsoperators `<` die Funktion *pPred* aufgerufen wird.

reverse()
: Die Methode *reverse*() kehrt die Ordnung der Elemente der Liste um.

3.4.6 Hash–Tabellen

Der Typ `Hash` stellt Hash–Tabellen zur Verfügung. Eine Hash–Tabelle enthält eine Menge von paarweisen Einträgen. Jeder Eintrag besteht aus einem Schlüssel und einem Wert. Der Schlüssel wird genutzt, um auf den Wert lesend oder schreibend zuzugreifen.

Als Schlüssel können Werte der einfachen Typen `Int`, `Float` und `Symbol` verwendet werden, wie auch alle Referenztypen, die die instanzbezogene Methode `getHashValue()` definieren. Die Methode `getHashValue()` muß einen Wert vom Typ `Int` zurückgeben, der für zwei Schlüssel, auf die der Gleichheitsoperator `==` angewandt *wahr* ergibt, gleich ist.

In einer Hash–Tabelle können Schlüssel verschiedener Typen verwendet werden. Zwei Schlüssel werden als gleich angesehen, wenn ihre Typen identisch sind und der Gleichheitsoperator `==`, angewandt auf beide Schlüssel, *wahr* ergibt.

Der Konstruktor *Hash()* erzeugt eine leere Hash–Tabelle. Die anfängliche Größe der Hash–Tabelle ist implementierungsabhängig. Sie wächst mit der Anzahl der in der Hash–Tabelle gespeicherten Werte, wobei die Zeit zum Vergrößern der Hash–Tabelle auf aufeinanderfolgende lesende oder schreibende Zugriffe verteilt wird, wobei die so zusätzlich in einem Zugriff verbrauchte Zeit im Durchschnitt unabhängig von der Größe der Hash–Tabelle ist.

Auf dem Typ `Hash` können folgende Operatoren (siehe Abschn. 3.6) und Methoden (siehe Abschn. 3.8.4) angewandt werden:

operator[](pKey(Object)) $\rightarrow$ *Object*
operator[](pKey(Object), pValue(Object))
: Der Index–Operator erwartet einen Schlüssel als Index–Wert, der die oben genannten Anforderungen an einen Schlüssel erfüllt. Wenn die Hash–Tabelle einen Eintrag mit diesem Schlüssel enthält, wird der für diesen Eintrag gespeicherte Wert zurückgegeben. Andernfalls wird eine Ausnahme ausgelöst.

  Wird der Index–Operator auf der linken Seite des Zuweisungsoperators verwendet, wird das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators in der Hash–Tabelle als Wert unter dem angegebenen Schlüssel gespeichert. Existiert noch kein Eintrag für diesen Schlüssel, wird ein neuer Eintrag angelegt.

operator!!() → Int
: Der Test–Operator liefert 1, wenn die Hash–Tabelle mindestens einen Eintrag enthält. Andernfalls liefert er 0.

getAt(pKey(Object)) → Object
: Die Methode *getAt*() erwartet einen die oben genannten Anforderungen erfüllenden Schlüssel als Argument. Existiert kein Eintrag mit diesem Schlüssel, wird eine Ausnahme ausgelöst. Andernfalls wird der Wert dieses Eintrags zurückgegeben.

setAt(pKey(Object), pValue(Object))
: Die Methode *setAt*() erwartet einen die oben genannten Anforderungen erfüllenden Schlüssel als erstes Argument und ein Objekt eines beliebigen Typs als zweites Argument. Existiert kein Eintrag mit diesem Schlüssel, wird ein neuer angelegt. Anschließend wird der Wert des zweiten Arguments in diesem Eintrag als Wert gespeichert.

size() → Int
: Es wird die Anzahl Einträge in der Hash–Tabelle zurückgegeben.

empty() → Int
: Wenn *size*() = ist, wird 1 zurückgegeben, andernfalls 0.

hasKey(pKey(Object)) → Int
: Die Methode *hasKey*() erwartet einen die oben genannten Anforderungen erfüllenden Schlüssel als Argument. Sie gibt 1 zurück, wenn in der Hash–Tabelle ein Eintrag mit diesem Schlüssel existiert, andernfalls 0.

keys() → Vector
: Die Methode *keys*() gibt einen `Vector` zurück, dessen einzelne Elemente die Schlüssel aller Einträge der Hash–Tabelle sind.

values() → Vector
: Die Methode *values*() gibt einen `Vector` zurück, dessen einzelne Elemente die Werte aller Einträge der Hash–Tabelle sind.

swap(pHash(Hash))
: Wenn das Argument nicht vom Referenztyp `Hash` ist, wird eine Ausnahme ausgelöst. Andernfalls werden die Einträge beider Hash–Tabellen ausgetauscht.

remove(pKey(Object))
: Das Argument muß ein die oben genannten Anforderungen erfüllender Schlüssel sein. Existiert kein Eintrag zu diesem Schlüssel, wird eine Ausnahme ausgelöst. Andernfalls wird der entsprechende Eintrag gelöscht.

Nur wenn die Methoden `keys()` und `values()` ohne dazwischen liegenden Aufruf einer anderen Methode der Hash–Tabelle, einschließlich der Index–Operatoren, aufgerufen werden, ist garantiert, daß die Reihenfolge der durch beide Methoden zurückgegebenen Schlüssel und Werte identisch ist.

3.5 Anweisungen

Die Übersetzungseinheit (*translation-unit*) bildet das Eintrittssymbol der OFML–Grammatik (Abschn. 3.1.3). Jede Übersetzungseinheit besteht aus einer optionalen Paketanweisung, einer (möglicherweise leeren) Folge von Importanweisungen (*import-stmts*) und einer (möglicherweise leeren) Folge von anderen Anweisungen (*stmt-list*). Syntax und Semantik von Paket- und Importanweisung werden in Abschnitt 3.7 beschrieben.

translation-unit:
package-stmt$_{opt}$ *import-stmts*$_{opt}$ *stmt-list*$_{opt}$
import-stmts:
import-stmts$_{opt}$ *import-stmt*
stmt-list:
stmt-list$_{opt}$ *stmt*

Eine OFML–Anweisung kann folgendes beinhalten: eine Definition (*definition-stmt*), einen Ausdruck (*expr-stmt*), eine Steueranweisung (*ctrl-stmt*) oder eine Verbundanweisung (*compound-stmt*).

stmt:
definition-stmt
expr-stmt
ctrl-stmt
compound-stmt

Definitionen werden durch das Übersetzungsprogramm behandelt. Alle anderen Anweisungen werden in der Reihenfolge ihres textuellen Auftretens zur Laufzeit ausgeführt.

In einigen Fällen wird als Abschluß einer Anweisung entweder ein Semikolon oder das Dateiende erwartet[8].

eox:
; | EOF

3.5.1 Definitionen

Folgende Elemente können durch eine Definition eingeführt werden: Variablen (*var-def*), benannte Funktionen (*named-func-def*), Klassen (*class-def*), der Name des Pakets, zu dem die Übersetzungseinheit gehört (*package-stmt*) und die von der Übersetzungseinheit importierten Pakete (*import-stmt*). Paket- und Importanweisung werden in Abschnitt 3.7, Klassendefinitionen in Abschnitt 3.8 beschrieben.

[8]Der Abschluß einer Anweisung durch das Dateiende wird erlaubt, um im interaktiven Betrieb auf die Eingabe des abschließenden Semikolons verzichten zu können.

definition-stmt:
var-def
named-func-def
class-def

Definition von Variablen

Eine Variablendefinition beginnt mit einer optionalen Folge von Modifikatoren und dem Schlüsselwort `var`, gefolgt von einem oder mehreren durch Komma getrennten Initialisierungs–Ausdrücken (*init-expr*). Der letzte Ausdruck wird durch ein Semikolon oder das Dateiende (*eox*) abgeschlossen. Jeder Initialisierungs–Ausdruck besteht aus einem Bezeichner (*ident*), optional gefolgt vom Zuweisungsoperator und einem im Wert–Kontext (siehe Abschn. 3.6.1) ausgewerteten Ausdruck (*expr*). Letzterer wird benutzt, um den Anfangswert der Variablen zu setzen. Fehlen Zuweisungsoperator und Ausdruck (*expr*, Abschn. 3.6), erhält die Variable den Wert `NULL`. Der Bezeichner wird unmittelbar nach dem ihn enthaltenden Initialisierungs–Ausdruck gültig.

var-def:
global-modifiers$_{opt}$ `var` *init-expr-list eox*
init-expr-list:
init-expr
init-expr-list `,` *init-expr*
init-expr:
ident
ident `=` *expr*

Modifikatoren werden in den Abschnitten 3.7.6 und 3.8 beschrieben.

Definition benannter Funktionen

Die Definition einer benannten Funktion beginnt mit einer optionalen Folge von Modifikatoren und dem Schlüsselwort `func`, gefolgt vom Namen der Funktion, der damit im aktuellen Namensraum (Abschn. 3.7) gültig wird. Es folgt ein Paar runder Klammern, das evtl. vorhandene Parameter einschließt, sowie eine Verbundanweisung, die den Funktionskörper repräsentiert.

named-func-def:
global-modifiers$_{opt}$ `func` *ident* `(` *param-list*$_{opt}$ `)` *compound-stmt*
global-modifiers$_{opt}$ `func` *ident* `(` *param-list* `, ...` `)` *compound-stmt*
`native` *global-modifiers*$_{opt}$ `func` *ident* `( ) ;`
param-list:
ident
param-list `,` *ident*

Modifikatoren werden in den Abschnitten 3.7.6 und 3.8 beschrieben.

Die zweite Form der Funktionsdefinition, bei der einer nicht–leeren Parameterliste, von dieser getrennt durch ein Komma, eine Ellipse (`...`) folgt, definiert eine Funktion mit variabler Anzahl von Argumenten. Wenn die darart definierte Funktion in ihrer Parameterliste n Parameter hat, so ist sie mit mindestens $n-1$ Argumenten aufzurufen. Für den n-ten Parameter wird ein Vektor erzeugt, der alle weiteren Argumente aufnimmt.

Die dritte Form der Funktionsdefinition, die durch das Schlüsselwort `native` eingeleitet wird, enthält keine Parameterdeklarationen[9] und keinen Funktionskörper. Statt dessen wird ihre Definition mit einem Semikolon abgeschlossen.

Eine Funktion, die als `native` definiert wird, ist in plattformabhängigem Kode implementiert. In der Regel ist dies eine andere Programmiersprache wie C, C++ oder Assembler.

3.5.2 Ausdrücke als Anweisungen

Die meisten Anweisungen in OFML bestehen aus einem Ausdruck (*expr*, der im Nebenwirkungs–Kontext (siehe Abschn. 3.6.1) ausgewertet wird und der durch ein Semikolon oder das Dateiende abgeschlossen wird.

expr-stmt:
$expr_{opt}$ *eox*

Fehlt der Ausdruck, handelt es sich um eine leere Anweisung, die dort benutzt werden kann, wo die Syntax eine Anweisung verlangt, aber keine Aktion gewünscht ist (z.B. bei einem leeren Schleifenkörper).

3.5.3 Steueranweisungen

Steueranweisungen dienen dazu, den Programmablauf dynamisch zu steuern und lassen sich grob in drei Kategorien einteilen: Auswahlanweisungen (*select-stmt*), Schleifenanweisungen (*loop-stmt*), Sprunganweisungen (*jump-stmt*) und Ausnahmeanweisungen (*exception-stmt*). Letztere werden in Abschnitt 3.5.3 beschrieben.

ctrl-stmt:
select-stmt
loop-stmt
jump-stmt
exception-stmt

[9] Dies bedeutet nicht, daß einer als `native` definierten Funktion keine Argumente übergeben werden können. Die Parameter werden nicht deklariert, da es Aufgabe des plattformabhängigen Kodes ist, die Anzahl der Argumente (und ihre Typen) zu überprüfen.

Auswahlanweisungen

Auswahlanweisungen wählen einen von mehreren Programmabläufen.

select-stmt:
`if (` *expr* `)` $stmt_1$
`if (` *expr* `)` $stmt_1$ `else` $stmt_2$
$label_{opt}$ `switch (` *expr* `) {` *switch-stmt-list* `}`

Bei beiden Formen der `if`–Anweisung wird der Ausdruck *expr* im Test–Kontext (siehe Abschn. 3.6.1) berechnet. Ergibt der Ausdruck *wahr*, wird die Anweisung $stmt_1$ ausgeführt. In der zweiten Form wird $stmt_2$ ausgeführt, wenn der Ausdruck *falsch* ergibt. Die syntaktische Mehrdeutigkeit bei `else` wird aufgelöst, indem ein `else` stets dem zuletzt aufgetretenen `if` ohne `else` auf der gleichen Blockschachtelungsebene zugeordnet wird.

Die Anweisungen $stmt_1$ und $stmt_2$ der `if`–Anweisung dürfen keine Definitionen (*definition-stmt*) sein.

Die `switch`–Anweisung berechnet den `switch`–Ausdruck *expr* im Wert–Kontext und verzweigt abhängig von seinem Ergebnis zu einer Marke (*switch-label*) innerhalb der folgenden, in geschweiften Klammern eingeschlossenen, Anweisungsliste (`switch-stmt-list`). Sie kann optional mit einer Marke versehen werden, auf die sich `break`– und `continue`–Anweisungen innerhalb der `switch`–Anweisungsliste beziehen können (siehe Abschn. 3.5.3).

switch-stmt-list:
$switch\text{-}stmt\text{-}list_{opt}$ *switch-stmt*
switch-stmt:
expr-stmt
ctrl-stmt
compound-stmt
switch-label
switch-label:
`case` *expr* `:`
`default :`

Dazu werden die Ausdrücke (*expr*) der `case`–Marken in der Reihenfolge ihres Auftretens berechnet und mit dem Ergebnis des `switch`–Ausdrucks auf Gleichheit verglichen, wobei das Ergebnis des `switch`–Ausdrucks auf der linken Seite des Vergleichsoperators erscheint. Ergibt der Vergleich *wahr*, wird mit der auf die `case`–Marke unmittelbar folgenden Anweisung (*switch-stmt*) fortgefahren. Andernfalls wird mit der folgenden `case`–Marke fortgefahren.

Nachdem alle `case`–Marken abgearbeitet wurden, ohne daß Gleichheit auftrat, wird mit der einer eventuell vorhandenen `default`–Marke folgenden Anweisung fortgefahren. Ist keine solche Marke vorhanden, wird keine Anweisung der Anweisungsliste abgearbeitet.

Innerhalb der Anweisungsliste einer `switch`–Anweisung darf maximal eine `default`–Marke vorkommen.

Ausnahmen, die durch den `switch`–Ausdruck, die `case`–Ausdrücke oder den Vergleichsoperator `==`, der auf die Ergebnisse beider Ausdrücke angewandt wird, ausgelöst wurden, werden nicht abgefangen.

Schleifenanweisungen

Schleifenanweisungen dienen der wiederholten Ausführung von Anweisungen. Alle Schleifenanweisungen können optional mit einer Marke versehen werden, auf die sich `break`– und `continue`–Anweisungen innerhalb des Schleifenkörpers beziehen können (siehe Abschn. 3.5.3).

Die den Schleifenkörper bildende Anweisung *stmt* darf keine Definition (*definition-stmt*) sein.

labeled-loop-stmt:
    $label_{opt}$ *loop-stmt*
label:
    ident `:`
loop-stmt:
    `while (` *expr* `)` *stmt*
    `do` *stmt* `while (` *expr* `)`
    `for (` $expr_{1opt}$`;` $expr_{2opt}$`;` $expr_{3opt}$ `)` *stmt*
    `foreach (` *name* `;` *expr* `)` *stmt*

Die Ausdrücke *expr* der `while`– bzw. `do`–`while`–Anweisungen und der zweite Ausdruck $expr_2$ der `for`–Anweisung werden im Test–Kontext (siehe Abschn. 3.6.1) ausgewertet.

Bei der `while`–Anweisung wird die Anweisung *stmt* so oft wiederholt, bis der Ausdruck *expr falsch* ergibt. Die Auswertung des Ausdrucks erfolgt **vor** der ersten Ausführung der Anweisung.

Die `do`–`while`–Anweisung gleicht der `while`–Anweisung, mit dem Unterschied, daß der Ausdruck **nach** der Ausführung der Anweisung *stmt* ausgewertet wird. Damit wird die Anweisung in jedem Fall mindestens einmal ausgeführt.

Bei der `for`–Anweisung wird der erste Ausdruck ($expr_1$) zuerst im Nebenwirkungs–Kontext ausgewertet. Er dient (i.a.) zur Initialisierung der Schleife. Der zweite Ausdruck ($expr_2$) wird vor jeder Abarbeitung des Schleifenkörpers berechnet. Ergibt er *falsch*, wird die `for`–Schleife abgebrochen. Andernfalls wird der Schleifenkörper abgearbeitet und anschließend der dritte Ausdruck ($expr_3$), der (i.a.) zur Reinitialisierung der Schleife dient, im Nebenwirkungskontext abgearbeitet.

Alle drei Ausdrücke der `for`–Anweisung können entfallen. Fehlt der zweite Ausdruck ($expr_2$), entspricht dies dem Testergebnis *wahr*.

Enthält die Anweisung keine `continue`-Anweisung, ist die `for`-Anweisung äquivalent zu:

```
expr1;
while ( expr2 ) {
    stmt
    expr3;
}
```

Die `foreach`-Anweisung dient der Iteration über eine Sequenz. Der erste Ausdruck muß dabei ein (ggf. qualifizierter) Name sein. Das Ergebnis des im Wert–Kontext abgearbeiteten zweiten Ausdrucks muß die Anforderungen an einen Sequenztyp erfüllen (siehe Abschn. 3.3.4), andernfalls wird (ggf. nach einer oder mehreren Iterationen) eine Ausnahme ausgelöst.

Die Implementierung muß sich so verhalten, als ob sie eine temporäre Variable *idx* anlegt, der vor der Abarbeitung der Schleife der `Int`–Wert -1 zugewiesen und die vor jedem Schleifendurchlauf um 1 erhöht wird. Der zweite Ausdruck wird einmal vor der Abarbeitung der Schleife ausgewertet und sein Ergebnis in der temporären Variable *seq* gespeichert. Die Schleife wird abgebrochen, wenn *idx* nach der Erhöhung um 1 gleich oder größer der aktuellen Länge der in *seq* gespeicherten Sequenz ist. Andernfalls wird das durch *idx* indizierte Element der Sequenz der durch den ersten Ausdruck bestimmten Variable zugewiesen. Anschließend wird der Schleifenkörper abgearbeitet.

Enthält die `foreach`–Anweisung keine `continue`–Anweisung, so ist sie äquivalent zu[10]:

```
seq = expr;
for (idx = 0; idx < seq.size(); idx++) {
    name = seq[idx];
    stmt
}
```

Sprunganweisungen

Durch Sprunganweisungen wird der Programmablauf unbedingt an einer anderen Stelle fortgesetzt.

jump-stmt:
continue-stmt
break-stmt
return-stmt

Die `continue`–Anweisung darf nur innerhalb einer `while`–, `do`–`while`–, `for`– oder `switch`–Anweisung auftreten. Bei `while`– und `do`–`while`–Schleifen setzt sie den Programmlauf mit der Auswertung des Testausdrucks fort, bei der `for`–Schleife mit der Auswertung des Reinitialisierungsausdrucks und bei der `switch`–Anweisung mit der erneuten Abarbeitung der gesamten `switch`–Anweisung.

continue-stmt:
`continue ;`
`continue` *ident* `;`

Eine `continue`–Anweisung ohne Bezeichner übergibt die Kontrolle an die innerste der oben aufgeführten Anweisungen. Existiert keine derartige Anweisung, tritt ein Übersetzungsfehler auf.

[10] Die Bezeichner sind nur zur Demonstration gewählt, prinzipiell erzeugt OFML interne Variablen, die nicht mit nutzerdefinierten Variablen in Konflikt geraten können.

Eine `continue`–Anweisung mit Bezeichner *ident* übergibt die Kontrolle an die innerste der oben aufgeführten Anweisungen, die mit dem gleichen Bezeichner als Marke versehen ist. Existiert keine derartige Anweisung, tritt ein Übersetzungsfehler auf.

Die `break`-Anweisung darf nur innerhalb einer `while`–, `do`–`while`–, `for`– oder `switch`–Anweisung auftreten.

break-stmt:
`break ;`
`break` *ident* `;`

Eine `break`–Anweisung ohne Bezeichner setzt den Programmlauf unmittelbar nach der innersten der oben aufgeführten Anweisungen fort. Existiert keine derartige Anweisung, tritt ein Übersetzungsfehler auf.

Eine `break`–Anweisung mit Bezeichner *ident* setzt den Programmlauf unmittelbar nach der innersten der oben aufgeführten Anweisungen, die mit dem gleichen Bezeichner als Marke versehen ist, fort. Existiert keine derartige Anweisung, tritt ein Übersetzungsfehler auf.

Die bei `continue`– und `break`–Anweisungen angegebenen und vor `while`–, `do`–`while`–, `for`– und `switch`–Anweisungen als Marken verwendeten Bezeichner liegen in einem separaten Namensraum, in dem sie beliebig oft verwendet werden können.

Die `return`–Anweisung beendet die Ausführung einer Funktion. Enthält die Anweisung einen Ausdruck (optional), wird dieser im Wert–Kontext (siehe Abschn. 3.6.1) abgearbeitet und sein Wert als Rückgabewert der Funktion geliefert. Fehlt der Ausdruck oder wird das Ende der Funktion ohne `return`-Anweisung erreicht, liefert die Funktion den Wert `NULL`.

Eine `return`–Anweisung außerhalb einer Funktion verursacht einen Übersetzungsfehler.

return-stmt:
`return ;`
`return` *expr* `;`

Ausnahmen

Ausnahmen können entweder durch interne Fehler (z.B. Fehler beim Laden von Übersetzungseinheiten oder Division durch Null) oder vom Programmierer explizit durch Ausführung der `throw`–Anweisung ausgelöst werden. Sie verursachen eine nichtlokale[11] Übergabe des Programmablaufs von der Stelle, an der die Ausnahme ausgelöst wurde, an die Stelle, an der sie abgefangen wird. Letztere wird zur Laufzeit des Programms ermittelt.

exception-stmt:
try-stmt
throw-stmt

[11] Nichtlokale Übergabe heißt, daß sich die abfangende `try`–Anweisung in einem direkten oder indirekten Aufrufer der die Ausnahme auslösenden Funktion befinden kann.

Die `try`–Anweisung ermöglicht die nutzerdefinierte Behandlung von Ausnahmen. Durch mehrere optionale `catch`–Teile ist es dabei möglich, unterschiedliche Typen von Ausnahmen getrennt zu behandeln. Dabei ist *name* der Name eines Typs und *ident* der Name einer nur innerhalb des `catch`–Blocks gültigen lokalen Variable, die den Wert der Ausnahme enthält.

try-stmt:
`try` *compound-stmt catch-stmts*$_{opt}$
catch-stmts:
catch-stmt catch-stmts$_{opt}$
catch-stmt:
`catch (` *name ident* `)` *compound-stmt*

Als Typname in der `catch`–Anweisung sind nur Referenztypen erlaubt. Andere Typen führen zu einem Übersetzungsfehler. Mit einer `catch`–Anweisung werden alle die Ausnahmen abgefangen, die Instanz der über den Typnamen angegebenen Klasse oder einer von dieser abgeleiteten Klasse sind.

Hat eine `try`–Anweisung mehrere `catch`–Anweisungen, so wird der Körper der ersten passenden `catch`–Anweisung ausgeführt, auch wenn eine folgende `catch`–Anweisung der gleichen `try`–Anweisung eine exaktere Übereinstimmung zwischen dem Typ des `catch`–Parameters und der Klasse der Ausnahme liefern würde.

Eine `try`–Anweisung ohne `catch`–Anweisung fängt alle Ausnahmen ab. Kann in einer `try`–Anweisung mit mindestens einer `catch`–Anweisung keine Übereinstimmung zwischen wenigstens einem der Typen der `catch`–Parameter und der Klasse der Ausnahme festgestellt werden, wird die Ausnahme durch diese `try`–Anweisung nicht abgefangen.

Die `throw`–Anweisung erlaubt es dem Programmierer, selbst Ausnahmen auszulösen. Der Wert des im Wert–Kontext (siehe Abschn. 3.6.1) abgearbeiteten Ausdrucks *expr* wird dabei als Wert der Ausnahme übergeben.

throw-stmt:
`throw` *expr eox*

Das Ergebnis des Ausdrucks einer `throw`–Anweisung muß einen Referenztyp haben. Andernfalls wird eine andere Ausnahme ausgelöst. Die `throw`–Anweisung übergibt den Programmlauf an die sie dynamisch umschließende `try`–Anweisung, die entweder keine oder eine passende `catch`–Anweisung enthält. Ist keine derartige `try`–Anweisung vorhanden, wird die Ausnahme implementierungsabhängig durch das Laufzeitsystem behandelt, z.B. durch Ausgabe einer Fehlermeldung und evtl. Abbruch der Programmausführung.

Verbundanweisungen

Verbundanweisungen dienen dazu, eine Folge von mehreren Anweisungen an Stellen einzufügen, an denen syntaktisch nur eine Anweisung erlaubt ist, z.B. als Körper einer Schleife.

compound-stmt:
`{` *stmt-list* `}`

Verbundanweisungen stellen einen neuen Namensraum bereit, in dem innerhalb dieser Verbundanweisung definierte Variablen eingetragen werden. Beim Binden eines Bezeichners an eine Variable wird nacheinander von innen nach außen in den Namensräumen der statisch umschließenden Verbundanweisungen gesucht.

Variablen mit gleichem Bezeichner dürfen in einer Verbundanweisung nicht mehrfach definiert werden. Verbundanweisungen dürfen keine Funktions– oder Klassendefinitionen enthalten.

3.6 Ausdrücke

Der folgende Abschnitt beschreibt die Operatoren von OFML, geordnet nach ihrem Vorrang. Vorrang, Assoziativität und Auswertungsreihenfolge der Operanden sind fest vorgeschrieben. Falls nicht anders angegeben, werden Operanden von links nach rechts ausgewertet, wobei die Auswertung eines Operanden mit allen Seiteneffekten abgeschlossen sein muß, bevor mit der Auswertung des nächsten begonnen wird. Dies trifft auch für die Argumente von Funktionen bzw. Methoden zu. Mit wenigen Ausnahmen, die speziell genannt werden, werden stets alle Operanden eines Operators ausgewertet.

Das Verhalten von unären Operatoren richtet sich nach dem Typ des Ergebnisses des Operanden. Bei binären Operatoren richtet es sich nach dem Typ des Ergebnisses des linken Operanden. Hat das entsprechende Ergebnis einen vordefinierten Referenztyp, so ist das genaue Verhalten des jeweiligen Operators, sofern definiert, in Abschnitt 3.4 beschrieben.

3.6.1 Wert–, Test– und Nebenwirkungs–Kontext

Ausdrücke und Teilausdrücke werden in einem von 3 verschiedenen Kontexten abgearbeitet:

Wert–Kontext Der Ausdruck muß einen Wert eines der einfachen Typen oder einen Referenztyp liefern. Ist das Ergebnis des Ausdrucks ein Wahrheitswert, so wird dieser in den `Int`–Wert 1 konvertiert, wenn er *wahr* ist und in den `Int`–Wert 0, wenn er *falsch* ist.

Test–Kontext Der Ausdruck muß einen Wahrheitswert liefern, d.h. entweder *wahr* oder *falsch*. Ist das Ergebnis des Ausdrucks ein Wert eines Referenztyps, so wird für diesen die Operatorfunktion `operator!!()` aufgerufen und im weiteren deren Rückgabewert betrachtet. Ist das Ergebnis jetzt kein `Int` oder `Float`, wird eine Ausnahme ausgelöst. Andernfalls wird das Ergebnis des Ausdrucks *wahr*, wenn der `Int` oder `Float` ungleich Null ist, andernfalls *falsch*.

Nebenwirkungs–Kontext Der Ausdruck wird ausgewertet, um einen Seiteneffekt (Nebenwirkung) zu erzielen. Das Ergebnis des Ausdrucks, das sowohl einen Wahrheitswert darstellen als auch ein Wert eines einfachen Typs oder eines Referenztyps sein kann, wird ignoriert.

Wenn nicht anders erwähnt, arbeiten Operatoren ihre Operanden im Wert–Kontext ab.

3.6.2 Primäre Ausdrücke

Primäre Ausdrücke sind Bezeichner (Abschn. 3.7.2 und 3.7.5), literale Konstanten (Abschn. 3.2.5), spezielle Konstruktoren für Vektoren (Abschn. 3.4.4) und Listen (Abschn. 3.4.5) oder geklammerte Ausdrücke:

primary-expr:
name
constant
special-ctor
`(` *expr* `)`
special-ctor:
`[` *arg-expr-list*$_{opt}$ `]`
`@(` *arg-expr-list*$_{opt}$ `)`
arg-expr-list:
assign-expr
arg-expr-list `,` *assign-expr*

Namen (*name*) sind in Abschnitt 3.7.2 beschrieben, Konstanten (*constant*) in Abschnitt 3.2.5 und die speziellen Konstruktoren (*special-ctor*) in den Abschnitten 3.4.4 und 3.4.5. Der Wert eines geklammerten Ausdrucks ist gleich dem Wert des Ausdrucks *expr* innerhalb der Klammern.

3.6.3 Postfix–Ausdrücke

Postfix–Ausdrücke sind linksassoziativ.

postfix-expr:
primary-expr
postfix-expr `[` *expr* `]`
postfix-expr `[` *expr*1_{opt} `:` *expr*2_{opt} `]`
postfix-expr `(` *arg-expr-list*$_{opt}$ `)`
postfix-expr `.` *ident*
postfix-expr `++`
postfix-expr `--`

Der Operator zum Zugriff auf Attribute „`.`“ wird in Abschnitt 3.8 beschrieben.

Index–Ausdrücke

Der *postfix-expr* im Index–Ausdruck *postfix-expr* `[` *expr* `]` muß einen Referenztyp liefern. Andernfalls wird eine Ausnahme ausgelöst.

Für Referenztypen können zwei Operatormethoden definiert werden, die zum Abfragen und Setzen eines Objekts in einer Sequenz anhand eines Index dienen:

`operator[](`*idx*`)` wird für das Ergebnis des *postfix-expr* aufgerufen, wenn der indizierte Wert gelesen werden soll. Das Ergebnis von *expr* wird an den Parameter *idx* übergeben. Der Rückgabewert der Operatormethode ist der Wert des Index–Ausdrucks.

`operator[](`*idx*`,` *value*`)` wird für das Ergebnis des *postfix-expr* aufgerufen, wenn der indizierte Wert geschrieben werden soll[12]. Das Ergebnis von *expr* wird an den Parameter *idx* übergeben und der zu schreibende Wert an den Parameter *value*. Ein eventueller Rückgabewert wird ignoriert.

Bereichs–Ausdrücke

Der *postfix-expr* im Bereichs–Ausdruck *postfix-expr* `[` $expr_{1opt}$ `:` $expr_{2opt}$ `]` muß einen Referenztyp liefern. Andernfalls wird eine Ausnahme ausgelöst.

Wenn $expr_1$ nicht angegeben wurde, wird der `Int`–Wert 0 an den Bereichs–Operator als Bereichsanfang übergeben. Analog wird, wenn $expr_2$ nicht angegeben wurde, der Rückgabewert der Methode *size()*, angewandt auf das Ergebnis des *postfix-expr*, als Bereichsende an den Bereichs–Operator übergeben. Eine Ausnahme wird ausgelöst, wenn die Methode *size()* nicht existiert.

Für Referenztypen können zwei Operatormethoden definiert werden, die zum Abfragen und Setzen eines Bereiches einer Sequenz an Hand von Anfangs– und Ende–Index dienen:

`operator[:](`*begin*`,` *end*`)` wird für das Ergebnis des *postfix-expr* aufgerufen, wenn der Bereich gelesen werden soll. Das Ergebnis von $expr_1$ wird an den Parameter *begin* und das Ergebnis von $expr_2$ an den Parameter *end* übergeben. Der Rückgabewert der Operatormethode ist der Wert des Bereichs–Ausdrucks.

`operator[:](`*begin*`,` *end*`,` *value*`)` wird für das Ergebnis des *postfix-expr* aufgerufen, wenn der Bereich geschrieben werden soll. Das Ergebnis von $expr_1$ wird an den Parameter *begin*, das Ergebnis von $expr_2$ an den Parameter *end* und der zu schreibende Wert an den Parameter *value* übergeben. Ein eventueller Rückgabewert wird ignoriert.

Funktionsaufrufe

Bei Funktionsaufrufen muß der erste Ausdruck (*postfix-expr*) ein Objekt eines Referenztyps liefern, der den Funktionsaufruf–Operator (`operator()`) implementiert, wie die vordefinierten Funktionstypen `Func` und `CFunc`. Ein derartiges Objekt wird im folgenden als Funktion bezeichnet.

Bei Funktionsaufrufen kann bezüglich der aufgerufenen Funktion zwischen folgenden beiden Fällen unterschieden werden:

- Die aufgerufene Funktion ist eine gewöhnliche Funktion oder eine klassenbezogene (statischen) Methode.
- Die aufgerufene Funktion ist eine instanzbezogene Methode.

[12] Dies ist z.B. der Fall, wenn der Index–Operator auf der linken Seite des Zuweisungsoperators verwendet wird. Das Ergebnis des Ausdrucks auf der rechten Seite des Zuweisungsoperators wird dann als *value* an den Index–Operator übergeben.

Der Unterschied beim Aufruf einer instanzbezogenen Methode zum Aufruf einer klassenbezogenen (statischen) Methode oder einer gewöhnlichen Funktion besteht darin, daß an eine instanzbezogene Methode das Objekt, für das die Methode aufgerufen wird, implizit als Parameter `self` übergeben wird.

Ist die aufzurufende Funktion eine instanzbezogene Methode und ist der die Methode liefernde Ausdruck in der Form *postfix-expr*$_2$ `.` *ident*, wird das Ergebnis von *postfix-expr*$_2$ als Parameter `self` übergeben. Andernfalls muß es sich bei dem Aufrufer um eine instanzbezogene Methode handeln und es wird `self` der aufrufenden Methode als Parameter `self` der aufzurufenden instanzbezogenen Methode übergeben.

Eine Ausnahme wird ausgelöst, wenn für eine instanzbezogene Methode kein Objekt als `self` übergeben werden kann[13], oder wenn die Klasse des als `self` übergebenen Objekts nicht gleich der Klasse, oder einer von dieser abgeleiteten Klasse, ist, für die die aufgerufene instanzbezogene Methode definiert wurde.

Die Argumentübergabe erfolgt analog zur Zuweisung bei einfachen Typen als Wert (*call by value*) und bei Referenztypen als Referenz (*call by reference*). Es müssen exakt die gleiche Anzahl Argumente übergeben werden, wie in der Funktionsdefinition angegeben, es sei denn, die Funktion ist als Funktion mit variabler Argumentanzahl definiert worden. In diesem Fall darf die Anzahl der übergebenen Argumente höchstens um eins kleiner sein als die Anzahl der deklarierten Parameter. Alle weiteren Argumente werden in Form eines Vektors dem letzten Parameter zugewiesen.

Der Rückgabewert eines Funktionsaufrufs ist der an die in der aufgerufenen Funktion an die `return`–Anweisung übergebene Wert oder `NULL` (siehe Abschn. 3.5.3).

Für Klassen kann der Funktionsaufruf–Operator wie folgt definiert werden:

`operator()(`*parameters*`)` wird für die Instanz einer Klasse aufgerufen, wenn diese Instanz das Ergebnis des *postfix-expr* ist. Die Argumente des Funktionsaufrufs werden in der oben beschriebenen Art und Weise an die an Stelle von *parameters* deklarierten Parameter des Funktionsaufruf–Operators übergeben. Der Rückgabewert der Funktionsaufrufs–Operatormethode ist das Ergebnis des Funktionsaufrufs.

Postfix–Inkrementierung und –Dekrementierung

Der Operand eins Postfix–Inkrement– oder –Dekrement–Operators muß eine Variable, ein Index–Ausdruck oder ein Bereichs–Ausdruck sein. Andernfalls tritt ein Übersetzungsfehler auf.

Die Postfix–Inkrement– und –Dekrement–Operatoren `++` und `--` verhalten sich abhängig vom Typ des Operanden wie folgt:

Hat der Wert des Operanden einen einfachen Typ, so muß er `Int` oder `Float` sein. Andernfalls wird eine Ausnahme ausgelöst. Für die Abarbeitung des Operators gilt dann die folgende Äquivalenz:

$$expr \oplus\oplus \equiv \texttt{(}tmp \texttt{ = } expr\texttt{, } expr \texttt{ = } tmp \oplus \texttt{ 1, } tmp\texttt{)},$$

wobei *tmp* eine für die Dauer der Abarbeitung dieses Teilausdrucks dynamisch erzeugte unbenannte Variable ist und Teilausdrücke des Ausdrucks *expr* nur einmal berechnet werden. Die Addition oder Subtraktion folgt den in Abschnitt 3.3.3 aufgeführten Regeln.

[13] Dies ist der Fall, wenn es sich bei der aufrufenden Funktion um eine gewöhnliche Funktion oder um eine klassenbezogene (statische) Methode handelt.

Hat der Wert des Operanden einen Referenztyp, so wird die Operatormethode `operator++(`*value*`)` oder `operator--(`*value*`)` für den Postfix–Inkrement– oder –Dekrement–Operator für diesen Referenztyp aufgerufen. Dem Parameter *dummy* wird `NULL` übergeben. Er dient lediglich zur Unterscheidung vom entsprechenden Präfix–Inkrement– oder –Dekrement–Operator. Der Rückgabewert der Operatormethode ist das Ergebnis des Operators.

3.6.4 Unäre Operatoren

Unäre Ausdrücke sind rechtsassoziativ.

unary-expr:
    postfix-expr
    unary-op unary-expr
unary-op:
    `+` | `-` | `++` | `--` | `~` | `!` | `!!` | `$`

Unärer Plus– und Minus–Operator

Die unären Plus– und Minus–Operatoren `+` und `-` verhalten sich abhängig vom Typ des Operanden wie folgt:

Hat der Wert des Operanden einen einfachen Typ, so muß er `Int` oder `Float` sein. Andernfalls wird eine Ausnahme ausgelöst. Im Fall des Plus–Operators ist dann der Wert des Operanden gleich dem Ergebnis des Operators. Im Fall des Minus–Operators (arithmetischer Negationsoperator) ist das Ergebnis des Operators gleich dem Wert des Operanden multipliziert mit -1[14].

Hat der Wert des Operanden einen Referenztyp, so wird die Operatormethode `operator+(`*value*`)` oder `operator-(`*value*`)` für den unären Plus– oder Minus–Operator für diesen Referenztyp aufgerufen. Der Wert des Operanden wird als Parameter *value* an diese Methode übergeben, deren Rückgabewert das Ergebnis des Operators ist.

Präfix–Inkrementierung und –Dekrementierung

Der Operand eines Präfix–Inkrement– oder –Dekrement–Operators muß eine Variable, ein Index–Ausdruck oder ein Bereichs–Ausdruck sein. Andernfalls tritt ein Übersetzungsfehler auf.

Die Präfix–Inkrement– und Dekrement–Operatoren `++` und `--` verhalten sich abhängig vom Typ des Operanden wie folgt:

Hat der Wert des Operanden einen einfachen Typ, so muß er `Int` oder `Float` sein. Andernfalls wird eine Ausname ausgelöst. Für die Abarbeitung des Operators gilt dann die folgende Äquivalenz:

$$\oplus\oplus expr \equiv \texttt{(}expr\ \texttt{=}\ tmp\ \texttt{=}\ expr \oplus \texttt{1,}\ tmp\texttt{)},$$

[14] Aufgrund der Verwendung des Zweierkomplements zur Darstellung ganzer Zahlen ist die arithmetische Negation des betragsmäßig größten darstellbaren negativen Wertes gleich diesem Wert.

wobei *tmp* eine für die Dauer der Abarbeitung dieses Teilausdrucks dynamisch erzeugte unbenannte Variable ist und Teilausdrücke des Ausdrucks *expr* nur einmal berechnet werden. Die Addition oder Subtraktion folgt den in Abschnitt 3.3.3 aufgeführten Regeln.

Hat der Wert des Operanden einen Referenztyp, so wird die Operatormethode `operator++()` oder `operator--()` für den Präfix–Inkrement– oder –Dekrement–Operator für diesen Referenztyp aufgerufen. Ein eventueller Rückgabewert dieser Methoden wird ignoriert. Der Wert des Operanden ist gleich dem Ergebnis des Operators.

Bitweise Negation

Der bitweise Negationsoperator `~` verhält sich abhängig vom Typ des Operanden wie folgt:

Hat der Wert des Operanden einen einfachen Typ, so muß er `Int` sein. Andernfalls wird eine Ausnahme ausgelöst. Das Ergebnis des Operators ist dann gleich der bitweisen Negation des Wertes des Operanden.

Hat der Wert des Operanden einen Referenztyp, so wird die Operatormethode `operator~()` für die bitweise Negation für diesen Referenztyp aufgerufen. Der Rückgabewert dieser Methode ist gleich dem Ergebnis des Operators.

Logische Negation

Der Operand des logischen Negationsoperators `!` wird im Test–Kontext ausgewertet. Sein Ergebnis ist *wahr*, wenn der Wert des Operanden *falsch* ist und *falsch*, wenn der Wert des Operanden *wahr* ist.

Der Test–Operator

Der Operand des Test–Operators `!!` wird im Test–Kontext ausgewertet. Sein Ergebnis ist identisch mit dem Wert des Operanden.

Der Symbol–Auflösungsoperator

Der Symbol–Auflösungsoperator `$` verlangt ein Argument vom Typ `Symbol`. Andernfalls wird eine Ausnahme ausgelöst. Er kann nicht für Referenztypen redefiniert werden.

Der Symbol–Auflöstungsoperator bindet das Symbol, das durch seinen Operanden geliefert wird, dynamisch an eine Variable. Dies geschieht nach den in Abschnitt 3.7.5 angegebenen Regeln zum Binden von Bezeichnern.

3.6.5 Multiplikative Operatoren

Multiplikative Ausdrücke sind linksassoziativ.

mul-expr:
    unary-expr
    mul-expr mul-op unary-expr
mul-op:
    `*` | `/` | `%`

Die multiplikativen Operatoren `*`, `/` und `%` verhalten sich abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen einfachen Typ, so muß er `Int` oder `Float` sein. Der Wert des rechten Operanden muß dann ebenfalls ein `Int` oder `Float` sein. Bei Verletzung einer der beiden Bedingungen wird eine Ausnahme ausgelöst. Andernfalls erfolgt die Operation in `Int`, wenn beide Operanden vom Typ `Int` sind oder in `Float`, wenn wenigstens einer der Operanden vom Typ `Float` ist. Im zweiten Fall wird ein Operand vom Typ `Int` vor der Operation nach `Float` konvertiert.

Der Operator `*` bewirkt eine Multiplikation der beiden Operanden.

Der Operator `/` bewirkt eine Division der beiden Operanden, wobei der linke Operand der Dividend und der rechte Operand der Divisor ist. Bei ganzzahliger Division wird gegen 0 gerundet.

Der Operator `%` ermittelt den Rest einer impliziten Division, bei der der linke Operand der Dividend und der rechte Operand der Divisor ist.

Wird die Rest–Operation in `Int` ausgeführt, gilt für den durch den Rest–Operator berechneten Wert $(a/b) * b + (a\%b) = a$. Daraus folgt, daß das Vorzeichen des Rests gleich dem Vorzeichen des Dividenden ist. Weiterhin ist der Betrag des Rests immer kleiner als der Betrag des Divisors.

Wird die Rest–Operation in `Float` ausgeführt, ist das Ergebnis der Wert $a - i * b$, wobei der ganzzahlige Wert i so gewählt wird, daß das Ergebnis das gleiche Vorzeichen wie a hat und der Betrag des Ergebnisses kleiner dem Betrag von b ist.

Die Berechnungen erfolgen entsprechend den in Abschnitt 3.3.3 aufgestellten Regeln.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator*(`*rhs*`)` (Multiplikation),

`operator/(`*rhs*`)` (Division) oder

`operator%(`*rhs*`)` (Restwertberechnung)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode ist das Ergebnis des Operators.

3.6.6 Additive Operatoren

Additive Ausdrücke sind linksassoziativ.

add-expr:
    mul-expr
    add-expr add-op mul-expr
add-op:
    `+` | `-`

Die additiven Operatoren `+` und `-` verhalten sich abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen einfachen Typ, so muß er `Int` oder `Float` sein. Der Wert des rechten Operanden muß dann ebenfalls ein `Int` oder `Float` sein. Bei Verletzung einer der beiden Bedingungen wird eine Ausnahme ausgelöst. Andernfalls erfolgt die Operation in `Int`, wenn beide Operanden vom Typ `Int` sind oder in `Float`, wenn wenigstens einer der Operanden vom Typ `Float` ist. Im zweiten Fall wird ein Operand vom Typ `Int` vor der Operation nach `Float` konvertiert.

Der Operator `+` bewirkt eine Addition beider Operanden.

Der Operator `-` bewirkt eine Subtraktion beider Operanden, wobei der linke Operand der Minuend und der rechte Operand der Subtrahend ist.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator+(`*rhs*`)` (Addition) oder

`operator-(`*rhs*`)` (Subtraktion)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode ist das Ergebnis des Operators.

3.6.7 Bitweise Verschiebung

Ausdrücke zur bitweisen Verschiebung sind linksassoziativ.

shift-expr:
    add-expr
    shift-expr shift-op add-expr
shift-op:
    `<<` | `>>` | `>>>`

Die Verschiebe–Operatoren `<<` (Linksverschiebung), `>>` (vorzeichenbehaftete Rechtsverschiebung) und `>>>` (vorzeichenlose Rechtsverschiebung) verhalten sich abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen einfachen Typ, so müssen beide Operanden vom Typ `Int` und der rechte Operand nicht–negativ sein. Sind diese Bedingungen nicht erfüllt, wird eine Ausnahme ausgelöst. Andernfalls wird der linke Operand als Bitfolge aufgefaßt, die um die durch den rechten Operanden angegebene Anzahl Positionen nach links (`<<`) oder nach rechts (`>>` und `>>>`) verschoben wird. Die Operatoren `<<` und `>>>` füllen frei werdende Positionen mit 0–Bits, während der Operator `>>` frei werdende Positionen mit dem Wert des höchstwertigsten Bits vor der Operation füllt.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator<<(`*rhs*`)` (Linksverschiebung),

`operator>>(`*rhs*`)` (vorzeichenbehaftete Rechtsverschiebung) oder

`operator>>>(`*rhs*`)` (vorzeichenlose Rechtsverschiebung)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode ist das Ergebnis des Operators.

3.6.8 Vergleichsoperatoren

Vergleichsausdrücke sind linksassoziativ[15].

comp-expr:
shift-expr
comp-expr comp-op shift-expr
comp-expr `instanceof` *shift-expr*
comp-op:
`<` | `>` | `<=` | `>=`

Die Vergleichsoperatoren `<` (kleiner als), `<=` (kleiner als oder gleich), `>=` (größer als oder gleich) und `>` (größer als) verhalten sich abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen einfachen Typ, so muß er `Int`, `Float` oder `Symbol` sein. Der Wert des rechten Operanden muß dann ein `Int` oder `Float` sein, wenn der linke Operand ein `Int` oder `Float` ist, oder er muß ein `Symbol` sein, wenn der linke Operand ein `Symbol` ist. Bei Verletzung einer der Bedingungen wird eine Ausnahme ausgelöst.

Ist keiner der Operanden ein `Symbol`, erfolgt die Vergleichsoperation in `Int`, wenn beide Operanden vom Typ `Int` sind oder in `Float`, wenn wenigstens einer der Operanden vom Typ `Float` ist. Im zweiten Fall wird ein Operand vom Typ `Int` vor der Operation nach `Float` konvertiert.

Sind die Operanden vom Typ `Symbol`, erfolgt ein Vergleich der internen Repräsentation beider Symbole. Das Ergebnis dieses Vergleichs ist nur in einer Instanz eines OFML–Programms garantiert reproduzierbar.

[15] Man beachte, daß aneinandergereihte Vergleichsausdrücke nicht die aus der gewohnten mathematischen Schreibweise übliche Bedeutung haben: `0 < x < 5` wird als `(0 < x) < 5` interpretiert und liefert immer den Wert 1.

Der Operator `<` ergibt *wahr*, wenn der Wert des linken Operanden kleiner ist als der Wert des rechten Operanden.

Der Operator `<=` ergibt *wahr*, wenn der Wert des linken Operanden kleiner oder gleich dem Wert des rechten Operanden ist.

Der Operator `>=` ergibt *wahr*, wenn der Wert des linken Operanden größer oder gleich dem Wert des rechten Operanden ist.

Der Operator `>` ergibt *wahr*, wenn der Wert des linken Operanden größer ist als der Wert des rechten Operanden.

Wenn der Vergleichsoperator nicht *wahr* ergibt, ergibt er *falsch*.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator<(`*rhs*`)` (kleiner als),

`operator<=(`*rhs*`)` (kleiner als oder gleich),

`operator>=(`*rhs*`)` (größer als oder gleich) oder

`operator>(`*rhs*`)` (größer als)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode wird im Test–Kontext interpretiert und entsprechend Abschnitt 3.6.1 in einen Wahrheitswert umgewandelt[16].

Typprüfung

Der Operator `instanceof` erwartet als Wert des rechten Ausdrucks einen von dem Typ `Type` abgeleiteten Typ (siehe Abschn. 3.4.1). Andernfalls wird eine Ausnahme ausgelöst. Das Ergebnis des `instanceof`–Operators ist *wahr*, wenn

- der linke Ausdruck einen Wert eines einfachen Typs liefert, dessen Typ identisch ist mit dem durch den rechten Ausdruck gelieferten Typ oder
- der linke Ausdruck einen Wert eines beliebigen Referenztyps liefert, der entweder identisch mit dem durch den rechten Ausdruck gelieferten Typ ist oder von diesem abgeleitet wurde.

Andernfalls ist das Ergebnis des Operators *falsch*.

3.6.9 Äquivalenzvergleiche

Äquivalenzvergleiche sind linksassoziativ.

[16] Der Rückgabewert sollte folglich ein `Int`–Wert sein, obwohl dies nicht zwingend erforderlich ist.

equiv-expr:
comp-expr
equiv-expr equiv-op comp-expr
equiv-op:
`==` | `!=` | `~=`

Ist der Wert des rechten Operanden der Vergleichsoperatoren `==` (Gleichheit) und `!=` (Ungleichheit) `NULL`, so werden beide Operanden als gleich angesehen, wenn auch der Wert des linken Operanden `NULL` ist. Der Operator `==` liefert bei Gleichheit *wahr*, andernfalls *falsch*. Der Operator `!=` liefert bei Gleichheit *falsch*, andernfalls *wahr*.

Andernfalls verhalten sich die Vergleichsoperatoren `==`, `!=` und `~=` (Muster–Übereinstimmung) abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen Referenztyp, so wird eine der Operatormethoden

`operator==(`*rhs*`)` (Gleichheit),

`operator!=(`*rhs*`)` (Ungleichheit) oder

`operator =(`*rhs*`)` (Muster–Übereinstimmung)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode wird im Test–Kontext interpretiert und entsprechend Abschnitt 3.6.1 in einen Wahrheitswert umgewandelt.

Hat der Wert des linken Operanden einen einfachen Typ, so wird im Falle des Operators `~=` eine Ausnahme ausgelöst. Im Falle des Operators `!=` ist das Ergebnis die logische Negation des Ergebnisses des Operators `==`, angewandt auf die gleichen Operanden. Der Operator `==` verhält sich wie folgt:

Ist der Wert des linken Operanden `NULL`, so ist das Ergebnis *wahr*, wenn der Wert des rechten Operanden ebenfalls `NULL` ist oder *falsch*, wenn der Wert des rechten Operanden von `NULL` verschieden ist.

Ist der Wert des linken Operanden vom Typ `Symbol`, so muß der Wert des rechten Operanden ebenfalls vom Typ `Symbol` sein. Andernfalls wird eine Ausnahme ausgelöst. Das Ergebnis ist *wahr*, wenn beide Symbole die gleiche Zeichenfolge verkörpern, andernfalls *falsch*.

Ist der Wert des linken Operanden vom Typ `Int` oder `Float`, so muß der Wert des rechten Operanden ebenfalls vom Typ `Int` oder `Float` sein. Andernfalls wird eine Ausnahme ausgelöst. Der Vergleich erfolgt in `Int`, wenn beide Operanden vom Typ `Int` sind, andernfalls in `Float`. Im zweiten Fall wird ein eventueller Operand vom Typ `Int` nach `Float` konvertiert. Das Ergebnis ist *wahr*, wenn beide Operanden (ggf. nach der Konvertierung) exakt den gleichen Wert haben, andernfalls *falsch*.

3.6.10 Minimum und Maximum

Minimum- und Maximumoperator sind linksassoziativ.

minmax-expr:
    equiv-expr
    minmax-expr minmax-op equiv-expr
minmax-op:
    `<?` | `>?`

Für die Operatoren `<?` (Minimum) und `>?` (Maximum) gelten für die Abarbeitung der Minimum– und Maximum–Operatoren die folgenden Äquivalenzen:

a `<?` $b \equiv$ `(`tmp_1 `=` a`,` tmp_2 `=` b`,` tmp_1 `<` tmp_2 `?` tmp_1 `:` tmp_2`)`

a `>?` $b \equiv$ `(`tmp_1 `=` a`,` tmp_2 `=` b`,` tmp_1 `>` tmp_2 `?` tmp_1 `:` tmp_2`)`

Dabei sind tmp_1 und tmp_2 für die Dauer der Abarbeitung dieses Teilausdrucks dynamisch erzeugte unbenannte Variablen.

3.6.11 Bitweise Verknüpfungen

Ausdrücke zur bitweisen Verknüpfung sind linksassoziativ.

bit-and-expr:
    minmax-expr
    bit-and-expr `&` *minmax-expr*
bit-xor-expr:
    bit-and-expr
    bit-xor-expr `^` *bit-and-expr*
bit-or-expr:
    bit-xor-expr
    bit-or-expr `|` *bit-xor-expr*

Die bitweisen Verknüpfungsoperatoren `&` (bitweises Und), `^` (bitweises exklusives Oder) und `|` (bitweises Oder) verhalten sich abhängig vom Typ des linken Operanden wie folgt:

Hat der Wert des linken Operanden einen einfachen Typ, so müssen beide Operanden vom Typ `Int` sein. Andernfalls wird eine Ausnahme ausgelöst. Das Ergebnis ist vom Typ `Int`.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator&(`*rhs*`)` (bitweises Und),

`operator^(`*rhs*`)` (bitweises exklusives Oder) oder

`operator|(`*rhs*`)` (bitweises Oder)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode ist das Ergebnis des Operators.

3.6.12 Logische Verknüpfungen

Ausdrücke zur logischen Verknüpfung sind linksassoziativ.

logic-and-expr:
    bit-or-expr
    logic-and-expr `&&` *bit-or-expr*
logic-or-expr:
    logic-and-expr
    logic-or-expr `||` *logic-and-expr*

Die Operatoren `&&` (logisches Und) und `||` (logisches Oder) werten ihren linken Operanden im Test–Kontext aus. Ist dessen Wert beim Operator `&&` *falsch*, wird der rechte Operand nicht ausgewertet und das Ergebnis des Operators ist *falsch*. Entsprechend wird beim Operator `||`, wenn der linke Operand *wahr* ergibt, der rechte Operand nicht ausgewertet und das Ergebnis des Operators ist *wahr*. Andernfalls werten beide Operatoren ihren rechten Operanden im Test–Kontext aus und ihr Ergebnis ist gleich dem Wert des rechten Operanden.

Der rechte Operand wird grundsätzlich nicht ausgewertet, wenn das Ergebnis des Operators durch das Ergebnis des linken Operanden bestimmt ist.

3.6.13 Bedingter Ausdruck

Bedingte Ausdrücke sind rechtsassoziativ.

cond-expr:
    logic-or-expr
    logic-or-expr `?` *expr* `:` *cond-expr*

Beim bedingten Ausdruck wird der erste Operand (*logic-or-expr*) im Test–Kontext ausgewertet. Ergibt die Auswertung *wahr*, wird der zweite Operand (*expr*) ausgewertet und das Ergebnis des bedingten Ausdrucks ist gleich dem Wert des zweiten Operanden. Ergibt die Auswertung des ersten Operanden *falsch*, wird der dritte Operand (*cond-expr*) ausgewertet und das Ergebnis des bedingten Ausdrucks ist gleich dem Wert des dritten Operanden.

Es wird immer nur entweder der zweite oder der dritte Operand ausgewertet.

3.6.14 Zuweisungsoperatoren

Alle Zuweisungsoperatoren sind rechtsassoziativ.

assign-expr:
    cond-expr
    unary-expr assign-op assign-expr
assign-op:
    `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `<<=` | `>>=` | `&=` | `^=` | `|=`

Der linke Operand einer Zuweisung muß eine Variable, ein Index–Ausdruck oder ein Bereichs–Ausdruck sein. Andernfalls tritt ein Übersetzungsfehler auf.

Ist der linke Operand eine Variable, wird durch den Zuweisungsoperator `=` der Wert des rechten Operanden berechnet und der Variable zugewiesen. Dieser Wert ist das Ergebnis des Zuweisungsoperators.

Ist der linke Operand ein Index– oder Bereichs–Ausdruck, wird durch den Zuweisungsoperator zuerst der rechte Operand berechnet. Anschließend werden von dem linken Operanden die Teilausdrücke (Sequenz, Index oder Indizes) berechnet und der Index– oder Bereichs–Operator zum Setzen eines Wertes mit dem Wert des rechten Operanden als Argument aufgerufen. Der Wert des rechten Operanden ist das Ergebnis des Zuweisungsoperators.

Die kombinierten Zuweisungsoperatoren `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=` und `|=` berechnen zuerst den Wert des rechten Operanden. Anschließend wird der Wert des linken Operanden berechnet. Abhängig von dessen Typ wird wie folgt verfahren:

Hat der Wert des linken Operanden einen einfachen Typ, so gilt für die Abarbeitung des kombinierten Zuweisungsoperators die folgende Äquivalenz:

$$lhs \oplus\texttt{=}\ rhs \equiv \texttt{(}tmp_1\ \texttt{=}\ rhs\texttt{,}\ lhs\ \texttt{=}\ tmp_2\ \texttt{=}\ lhs \oplus tmp_1\texttt{,}\ tmp_2\texttt{)}$$

Dabei sind tmp_1 und tmp_2 für die Dauer der Abarbeitung dieses Teilausdrucks dynamisch erzeugte unbenannte Variablen. Teilausdrücke des linken Operanden (a) werden nur einmal berechnet.

Hat der Wert des linken Operanden einen Referenztyp, so wird für den linken Operanden eine der Operatormethoden

`operator*=(`*rhs*`)` (Operator `*=`),

`operator/=(`*rhs*`)` (Operator `/=`),

`operator%=(`*rhs*`)` (Operator `\%=`),

`operator+=(`*rhs*`)` (Operator `+=`),

`operator-=(`*rhs*`)` (Operator `-=`),

`operator<<=(`*rhs*`)` (Operator `<<=`),

`operator>>=(`*rhs*`)` (Operator `>>=`),

`operator>>>=(`*rhs*`)` (Operator `>>>=`),

`operator&=(`*rhs*`)` (Operator `&=`),

`operator^=(`*rhs*`)` (Operator `^=`) oder

`operator|=(`*rhs*`)` (Operator `|=`)

aufgerufen, wobei der Wert des rechten Operanden als Parameter *rhs* übergeben wird. Der Rückgabewert der Operatormethode ist das Ergebnis des kombinierten Zuweisungsoperators.

3.6.15 Der Komma-Operator

Der Komma-Operator ist linksassoziativ.

expr:
assign-expr
expr `,` *assign-expr*

Der linke Operand wird im Nebenwirkungs–Kontext ausgewertet. Anschließend wird der rechte Operand ausgewertet. Sein Wert ist das Ergebnis des Komma–Operators.[17]

Tabelle 3.1 faßt noch einmal Vorrang und Assoziativität für alle Operatoren zusammen. Dabei bedeutet die kleinste Zahl die höchste Priorität.

Operatoren	Vorrang	Assoziativität
`::`	1	links
`() @() [] .`	2	links
`! !! ~ ++ -- + - $`	3	rechts
`* / %`	4	links
`+ -`	5	links
`<< >> >>>`	6	links
`< <= > >= instanceof`	7	links
`== != ~=`	8	links
`>? <?`	9	links
`&`	10	links
`^`	11	links
`|`	12	links
`&&`	13	links
`||`	14	links
`?:`	15	rechts
`= *= /= %= += -= <<= >>= >>>= &= ^= |=`	16	rechts
`,`	17	links

Tabelle 3.1: Operatoren

3.7 Pakete und Namensräume

3.7.1 Module

Jede Übersetzungseinheit bildet einen Modul. Ein Modul gehört zu einem Paket, das optional am Anfang des Moduls mit der `package`–Anweisung angegeben wird (siehe Abschn. 3.7.3).

[17] Man beachte, daß Komma-Ausdrücke entsprechend der hier definierten Grammatik in Kontexten, in denen das Komma eine andere syntaktische Bedeutung hat (z.B. in Argumentlisten von Funktionsaufrufen) geklammert werden müssen, um den gewünschten Effekt zu erzielen.

Ein Modul bildet einen Namensraum, der implizit alle innerhalb seines Pakets als `public` oder privat zum Paket (d.h. ohne `public` oder `private`) definierten Namen enthält. Zusätzlich zu diesen können weitere Namen aus anderen Paketen implizit oder explizit importiert werden (siehe Abschn. 3.7.4), sowie neue Namen innerhalb eines Moduls definiert werden.

Auf Namen im Namensraum eines Moduls kann nicht qualifiziert zugegriffen werden.

Wird versucht, innerhalb eines Moduls einen explizit importierten Namen erneut explizit zu importieren oder zu definieren, tritt ein Übersetzungsfehler auf. Implizit importierte Namen können mehr als einmal implizit importiert und höchstens einmal explizit importiert oder definiert werden.

Ein explizit importierter oder definierter Name verdeckt alle gleichnamigen implizit importierten Namen.

Werden mehrere gleiche Namen aus verschiedenen Paketen implizit importiert, so sind diese implizit importieren Namen nicht mehr sichtbar.

Ein Modul gilt als *geladen*, wenn er soweit übersetzt wurde, daß in ihm alle Definitionen auf Modul- und Klassenebene abgearbeitet wurden und die entsprechenden Namen durch den Übersetzer bei der Übersetzung anderer Module referenziert werden können. Verbundanweisungen müssen noch nicht übersetzt worden sein.

3.7.2 Pakete und Namensräume

In OFML existieren die folgenden Namensräume: Paket (siehe Abschn. 3.7.3), Modul (siehe Abschn. 3.7.1), Klasse (siehe Abschn. 3.8) und Verbundanweisung (siehe Abschn. 3.5.3).

Die Namensräume von Paketen und Klassen bilden eine Hierarchie, wobei die einzelnen Komponenten eines Namens durch doppelten Doppelpunkt `::` voneinander getrennt sind. Beginnt ein derartiger Name mit einem `::`, wird in der Root–Package mit der Suche begonnen, andernfalls in der Package, zu der der übersetzte Modul gehört. Steht der Operator `::` in der Mitte eines Namens, so wird in dem auf seiner linken Seite angegebenen Paket oder in der auf seiner linken Seite angegebenen Klasse nach dem auf seiner rechten Seite angegebenen Bezeichner gesucht. Die Suche nach Namen ohne `::` unterliegt anderen Regeln (siehe Abschn. 3.7.5). Die Syntax für Namen ist:

name:
    ident
    qualified-name
    fully-qualified-name
fully-qualified-name:
    `::` *ident*
    `::` *qualified-name*
qualified-name:
    name-qualifier `::` *ident*
name-qualifier:
    ident
    name-qualifier `::` *ident*

Bei der Verwendung eines nicht voll qualifizierten Namens muß es sich bei dessen ersten Komponente um eine Klasse handeln, die in dem Paket, zu dem der übersetzte Modul gehört, definiert wurde oder um ein direktes Unter–Paket dieses Paketes.

Beim (qualifizierten) Zugriff auf einen Namen, der noch nicht geladen wurde, wird versucht, ihn in der entsprechenden Package zu laden. Bei nicht qualifizierten Namen ist dies die Package, zu der der zugreifende Modul gehört. Bei qualifizierten Namen ist es die Package, die durch den Qualifikator angegeben wird. Zum Laden wird der Name über einen implementationsabhängigen Mechanismus auf den Namen eines Moduls abgebildet, der dann geladen wird.

3.7.3 Paket–Anweisung

Die Syntax einer Paket–Anweisung ist:

package-stmt:
`package` *fully-qualified-name* `;`

Der Modul wird in dem spezifizierten Paket übersetzt. Ist keine Paket–Anweisung vorhanden, wird der Modul in der Root–Package (oder Default–Package) übersetzt.

Bei der Übersetzung der Paket–Anweisung werden alle innerhalb des spezifizierten Pakets als `public` oder privat zum Paket (d.h. ohne `public` oder `private`) definierten Namen des Pakets implizit in den übersetzten Modul importiert. Ist keine Paket–Anweisung vorhanden, werden die Namen der Root–Package implizit importiert.

3.7.4 Import–Anweisung

Die Syntax der Import–Anweisungen ist:

import-stmt:
`import` *fully-qualified-name* `;`
`import :: * ;`
`import` *fully-qualified-name* `:: * ;`

Die erste Form der Import–Anweisung überprüft für den angegebenen Namen, ob er bereits definiert wurde. Wenn nicht, wird über einen implementierungsabhängigen Mechanismus der Modul bestimmt, der diesen Namen definiert. Dieser Modul wird geladen. Anschließend wird der Name in den Namensraum des importierenden Moduls übernommen. Es ist ein Fehler, mit dieser Form der Import–Anweisung Namen zu importieren, deren letzte Komponente identisch ist oder innerhalb des importierenden Moduls definiert wird.

Die zweite und dritte Form der Import–Anweisung stellt zuerst sicher, daß alle Module des angegebenen Pakets geladen worden sind. Anschließend werden alle als `public` definierten Namen dieses Pakets in den importierenden Modul übernommen.

Es ist nicht möglich, einen Namen zu importieren, der zum gleichen Paket gehört wie der importierende Modul.

Es ist zu beachten, daß die Import–Anweisung die Namen in den Namensraum des Moduls und nicht in den des Pakets, zu dem der Modul gehört, importiert. Dies ist notwendig, um eine gegenseitige Beeinflussung von Modulen durch Import–Anweisungen zu vermeiden.

3.7.5 Statisches und dynamisches Binden

Namen (sowohl einfache als auch (voll) qualifizierte) werden grundsätzlich statisch gebunden.

Der Punkt–Operator (Abschn. 3.6 und 3.8.3) kann als binärer Operator aufgefaßt werden, der auf der linken Seite ein Objekt und auf der rechten Seite einen Bezeichner erwartet. Das Binden des Bezeichners an das entsprechende Attribut des Objektes erfolgt dynamisch zur Laufzeit.

Die Definition von OFML läßt zu, daß die Implementierung zur Laufzeit dynamisch Attribute erzeugt. Um eine Überprüfung auf nichtdefinierte Namen vornehmen zu können, muß auf derartige Attribute über `self` zugegriffen werden (Abschn. 3.8.3).

Das Binden eines einfachen (nicht qualifizierten) Namens findet in der folgenden Reihenfolge statt:

1. Innerhalb von Funktionen und Methoden wird zuerst in dem Namensraum der innersten Verbundanweisung gesucht. Wird der Name in diesem nicht gefunden, wird die Suche von innen nach außen bis zum Namensraum der den Funktionskörper bildenden Verbundanweisung fortgesetzt.

   Kann der Name innerhalb einer Methode (sowohl instanzbezogen als auch klassenbezogen) nicht gefunden werden, wird die Suche im Namensraum der Klasse, zu der die Methode gehört, fortgesetzt.

   Kann der Name innerhalb einer (gewöhnlichen) Funktion nicht gefunden werden, wird die Suche innerhalb des Moduls, in dem die Funktion definiert wurde, fortgesetzt.

2. Innerhalb von Klassen wird im Namensraum der Klasse gesucht. Dieser enthält alle von Oberklassen geerbten Namen. Wird von einer klassenbezogenen Methode oder einem klassenbezogenen Initialisierer eine instanzbezogene Methode oder Variable gefunden, wird ein Übersetzungsfehler ausgelöst.

   Wird der Name innerhalb des Namensraumes der Klasse nicht gefunden, wird die Suche in dem Modul fortgesetzt, in dem die Klasse definiert wurde.

3. Die Suche im Modul erfolgt ausschließlich im Namensraum des Moduls. Dieser enthält alle importierten Namen[18].

[18] Um Speicherplatz zu sparen, muß die Implementierung nicht notwendigerweise jeden einzelnen Namen zur Übersetzungszeit übernehmen. Es ist dann jedoch notwendig, beim Auflösen eines Namens alle importierten Module nach diesem Namen zu durchsuchen, um Mehrfachdefinitionen und damit Mehrdeutigkeiten auszuschließen. Für Namen, für die dies bereits getan wurde, kann in der Symboltabelle des Moduls ein Eintrag angelegt werden, um bei der nächsten Verwendung des gleichen Namens den Aufwand zu verringern.

Das gleiche gilt für Namen, die zum Paket des Moduls gehören, aber nicht durch den Modul definiert wurden. Namen, die durch den Modul definiert wurden, befinden sich ohnehin in der Symboltabelle des Moduls.

Die Suchreihenfolge beim Zugriff auf einen nicht qualifizierten Namen ist dann wie folgt: 1. Symboltabelle des Moduls, 2. Symboltabelle des Pakets, 3. alle importierten Module.

3.7.6 Sichtbarkeit und Zugreifbarkeit

Ein einfacher Name ist sichtbar, wenn er nach den in Abschnitt 3.7.5 beschriebenen Regeln gebunden werden kann.

Ein qualifizierter Name ist sichtbar, wenn er entweder nicht als `private` definiert wurde oder sich ein nur aus der letzten Komponente des qualifizierten Namens bestehender einfacher Name auf die gleiche Definition bezieht und sichtbar ist.

Ein Name auf der rechten Seite des Punkt–Operators ist sichtbar, wenn er im Namensraum der Klasse des Objekts auf der linken Seite des Punkt–Operators existiert und er entweder nicht als `private` definiert wurde oder der Zugriff aus der gleichen Klasse heraus erfolgt.

Die Sichtbarkeit eines einfachen Namens kann eingeschränkt werden, wenn er durch den gleichen Namen in einem anderen Namensraum, in dem entsprechend Abschnitt 3.7.5 eher gesucht wird, verdeckt wird.

Ein Name ist zugreifbar, wenn er sichtbar ist und der Zugriff erlaubt ist. Beim nicht–qualifizierten Zugriff ist jeder sichtbare Name auch zugreifbar. Beim qualifizierten Zugriff oder beim Zugriff mittels des Punkt–Operators ist ein sichtbarer Name unter Umständen nicht zugreifbar.

Zugreifbarkeit und Sichtbarkeit werden durch Modifikatoren am Anfang einer Definition gesteuert. In diesem Abschnitt werden nur die Modifikatoren für die Definition von Variablen, Funktionen und Klassen auf der Ebene von Modulen beschrieben. Modifikatoren für Attribute von Klassen werden in Abschnitt 3.8 erläutert.

global-modifiers:
    global-modifiers$_{opt}$ *global-modifiers*
global-modifier:
    `final` | `public` | `private`

Das Schlüsselwort `final` bewirkt bei Variablen, daß diesen nach der innerhalb der Variablendefinition geforderten Initialisierung kein neuer Wert zugewiesen werden kann. OFML erlaubt jedoch, daß Variablen wie auch Funktionen und Klassen auf der Ebene von Modulen durch verschiedene Module oder durch erneute Übersetzung des (ggf. geänderten) gleichen Moduls redefiniert werden können, in welchem Fall sich dann auch der Wert von mit `final` definierten Variablen ändern kann.

Als `final` definierte Klassen dürfen nicht als Oberklassen von anderen Klassen verwendet werden.

Bei Funktionsdefinitionen darf `final` nicht angewendet werden.

Die Variablen, in denen Funktionen und Klassen gespeichert sind, werden implizit als `final` definiert.

Als `public` definierte Variablen, Funktionen und Klassen sind in allen Modulen, auch denen anderer Pakete, zugreifbar.

Als `private` definierte Variablen, Funktionen und Klassen sind nur innerhalb des definierenden Moduls sichtbar und zugreifbar.

Ist eine Variable, Funktion oder Klasse weder `public` noch `private` definiert, so gilt sie als privat zum Paket. Derartig definierte Namen sind generell sichtbar, jedoch nur von zum gleichen Paket gehörenden Modulen aus zugreifbar[19].

3.8 Klassen

3.8.1 Klassendefinitionen

Klassendefinitionen definieren neue Referenztypen und beschreiben deren Implementierung.

class-def:
global-modifiers$_{opt}$ `class` *ident super-class*$_{opt}$ *class-body*

3.8.2 Oberklassen

super-class:
`:` *ident*

In einer Klassendefinition kann optional der Name einer Oberklasse angegeben werden. Diese Oberklasse darf nicht als `final` definiert sein. Wird die Angabe einer Oberklasse weggelassen, erbt die Klasse automatisch von der Wurzelklasse `Object` (Abschn. 3.3.2).

Wenn die Oberklasse innerhalb der gleichen Übersetzungseinheit definiert ist, muß ihre Definition vor der Definition der abgeleiteten Klasse erscheinen. Des weiteren ist die in Abschnitt 3.1.2 genannte Einschränkung bezüglich der Verwendung von Oberklassen zu beachten.

Eine Klasse erbt von ihrer Oberklasse alle nicht als `private` definierten Attribute. Diese werden in den Namensraum der Unterklasse übernommen, sind somit in der Unterklasse zugreifbar. Als `private` definierte Attribute der Oberklasse sind in der Unterklasse nicht sichtbar.

3.8.3 Attribute

Der Körper eine Klassendefinition besteht aus einer Folge von Attributdefinitionen und klassenbezogenen Initialisierern.

class-body:
`{` *member-def-stmts*$_{opt}$ `}`
member-def-stmts:
member-def-stmts$_{opt}$ *member-def-stmt*
member-def-stmt:
field-def
method-def
static-initializer

[19] Obwohl diese Namen für Import–Anweisungen zum Importieren aller Namen eines Pakets (Stern–Form) sichtbar sind, versuchten diese Importanweisungen nicht, auf sie zuzugreifen (sie zu importieren).

3.8.4 Datenfelder

field-def:
field-modifiers$_{opt}$ `var` *init-expr-list* `;`
field-modifiers:
field-modifiers$_{opt}$ *field-modifiers*
field-modifier:
`public` | `protected` | `private` | `final` | `static`

Die Definition von Datenfeldern entspricht syntaktisch der Definition von Variablen (siehe Abschn. 3.5.1), wobei die zusätzlichen Modifikatoren `protected` und `static` möglich sind.

Die Initialisierung von klassenbezogenen Datenfeldern erfolgt unmittelbar nach dem Laden des die Klassendefinition enthaltenden Moduls. Die Initialisierung von instanzbezogenen Datenfeldern, einschließlich der Auswertung des Initialisierungsausdrucks, erfolgt in der Reihenfolge ihres Auftretens unmittelbar nach dem Erzeugen einer neuen Instanz und vor dem eventuellen Aufruf einer `initialize()`–Methode.

Als `static` definierte Datenfelder beschreiben klassenbezogene Variablen, die nur einmal je Klasse existieren. Andernfalls handelt es sich um instanzbezogene Variablen, die für jede Instanz der Klasse neu erzeugt werden.

Als `final` definierten Datenfeldern darf nach der innerhalb der Variablendefinition geforderten Initialisierung kein neuer Wert zugewiesen werden.

Datenfelder, die als `public` definiert wurden, sind aus allen Modulen, auch denen anderer Pakete, heraus zugreifbar.

Datenfelder, die als `protected` definiert wurden, sind generell sichtbar, jedoch nur aus Methoden, klassenbezogenen Initialisierern und Initialisierungsausdrücken von Datenfeldern der gleichen Klasse oder von ihr abgeleiteten Klassen aus zugreifbar.

Datenfelder, die als `private` definiert wurden, sind nur in Methoden, klassenbezogenen Initialisierern und Initialisierungsausdrücken von Datenfeldern der gleichen Klasse sichtbar und zugreifbar.

Datenfelder, die als privat zum Paket (d.h. ohne eines der Schlüsselworte `public`, `protected` und `private`) definiert wurden, sind generell sichtbar, jedoch nur aus Methoden, klassenbezogenen Initialisierern und Initialisierungsausdrücken von Datenfeldern der gleichen Klasse oder von ihr abgeleiteten Klassen, sowie aus allen zum gleichen Paket gehörenden Modulen aus zugreifbar.

Methoden

method-def:
`public` | `protected` | `private` | `final` | `static`
method-name:
ident
`operator` *method-operator*
method-operator:
`++` | `--` | `!!` | `~` | `*` | `/` | `%` | `+` | `-` | `<<` | `>>` | `>>>`
`<` | `<=` | `>=` | `>` | `==` | `!=` | `~=` | `&` | `^` | `|`
`*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `>>>=` | `&=` | `^=` | `|=`

Die Definition von Methoden entspricht syntaktisch der Definition von benannten Funktionen (Abschn. 3.5.1), wobei zusätzliche Modifikatoren und spezielle Namen zum Redefinieren von Operatoren möglich sind.

Als `static` definierte Methoden beschreiben klassenbezogene Methoden, die nicht auf instanzbezogene Variablen zugreifen können. Klassenbezogene Methoden werden im Zusammenhang mit dem Typ aufgerufen (s.u.). Andernfalls handelt es sich um eine instanzbezogene Methode, die unter Bezug auf ein bestimmtes Objekt, das eine Instanz der Klasse oder einer von dieser abgeleiteten Klasse ist, aufgerufen wird.

Als `final` definierte Methoden dürfen in Unterklassen nicht redefiniert werden.

Die Modifikatoren zur Steuerung des Zugriffs haben die gleiche Bedeutung wie bei Datenfeldern (s.o.).

Redefinition von Operatoren

Die meisten der durch die Grammatik von OFML unterstützten Operatoren können für Referenztypen redefiniert werden. Dazu müssen in der entsprechenden Klassendefinition instanzbezogene Methoden definiert werden, deren Namen sich aus dem Schlüsselwort `operator`, gefolgt von dem zu redefinierenden Operator, zusammensetzen. Die Anzahl der für Operator–Methoden zu deklarierenden Parameter ist in Abschnitt 3.6 festgelegt.

Wird eine Operator–Methode als `static` (klassenbezogen) oder mit einer nicht erlaubten Anzahl von Parameter definiert, tritt ein Übersetzungsfehler auf.

Konstruktoren

Konstruktoren werden nie explizit definiert, sondern immer automatisch erzeugt. Nutzerdefinierte Operationen zur Initialisierung von Instanzen können innerhalb der speziellen Instanzmethode

`initialize()` erfolgen, der die an den Konstruktor übergebenen Argumente übergeben werden. Ist eine `initialize()`–Methode definiert, so wird sie automatisch im Konstruktor aufgerufen. Ist keine `initialize()`–Methode definiert, so dürfen an den Konstruktor keine Argumente übergeben werden. `initialize()`–Methoden von Oberklassen werden **nicht** automatisch aufgerufen. Dieser Aufruf muß explizit in der `initialize()`–Methode der jeweiligen Unterklasse erfolgen!

Klassenbezogene Initialisierer

static-initializer:
`static` *compound-stmt*

Klassenbezogene Initialisierer bestehen aus dem Schlüsselwort `static`, gefolgt von einer Verbundanweisung. Klassenbezogene Initialisierer werden nach dem Laden des Moduls, der die Klassendefinition enthält, zusammen mit anderen ausführbaren Anweisungen auf Modul–Ebene und Initialisierungen von klassenbezogenen Variablen in der Reihenfolge ihres Auftretens innerhalb des Moduls abgearbeitet.

Das folgende Beispiel veranschaulicht die eben eingeführten Konzepte:

```
public class MyInt {
  private var value = 0;          // Instanzvariable
  private static var num;         // Klassenvariable
  public func initialize() {      // Initialisierungsmethode
    numInts++;
  }
  public func getValue() {        // normale Instanzmethode
    return (value);
  }
  public func incr(i) {
    value += i;
  }
  public static func getNum() {   // Klassenmethode
    return (num);
  }
  static {                        // klassenbezogener Initialisierer
    num = 0;
  }
}
```

Zugriff auf Attribute

Innerhalb von Instanzmethoden derselben Klasse erfolgt der Zugriff auf Attribute im allgemeinen direkt, d.h. diese befinden sich innerhalb des aktuellen Namensraumes. Alternativ dazu kann mittels des speziellen Schlüsselwortes `self` und des Zugriffsoperators „`.`“ auf Instanzvariablen und

-methoden zugegriffen werden. Werden Instanzvariablen dynamisch von der Implementierung erzeugt, wie dies bei Kindobjekten in *OFML* der Fall ist, muß auf diese mittels `self` zugegriffen werden. Im obigen Beispiel könnte demnach statt `return (value);` ebenso `return (self.value);` geschrieben werden.

Auf Instanzmethoden und -variablen erfolgt der Zugriff über den Operator „`.`“. Dabei ist der linke Operand ein beliebiger Ausdruck, dessen Typ das jeweilige Attribut besitzen muß, der rechte Operand ist der Name des Attributs, z.B. `i.getValue()`.

Auf Klassenmethoden und -variablen erfolgt der Zugriff über den Operator „`::`“ entsprechend den Regeln für qualifizierte Namen (Abschn. 3.7.2), wobei der Klassenname als Qualifikator benutzt wird z.B. `MyInt::getNum()`.

3.9 Vordefinierte Funktionen

3.9.1 Standardfunktionen

Standardfunktionen sind in dem Paket `::cobra::lang` definiert.

typeOf(pObject(Object)) $\rightarrow$ *Type*
: Die Funktion *typeOf*() erwartet einen beliebigen Wert eines einfachen Typs oder Referenztyps als Argument und gibt den Typ des Arguments zurück.

3.9.2 Numerische Standardfunktionen

Alle vordefinierten numerischen Standardfunktionen sind in dem Paket `::cobra::math` definiert.

Fehlerbehandlung

Bereichsfehler: Wenn ein Argument einer numerischen Standardfunktion außerhalb des Definitionsbereiches der Funktion liegt, wird eine Ausnahme ausgelöst.

Über- und Unterlauffehler: Ein Über- oder Unterlauffehler tritt auf, wenn das Ergebnis einer Funktion nicht als `Float` dargestellt werden kann. Wenn ein Überlauf aufgetreten ist (der Betrag des Ergebnisses ist so groß, daß er nicht in einem `Float` darstellbar ist), gibt die Funktion den Wert `Float::HUGE_VAL` zurück, mit dem gleichen Vorzeichen wie der korrekte Wert der Funktion (außer bei `tan()`). Im Falle eines Unterlaufs ist das Ergebnis `0`.

Argumentkonvertierung

Wenn an eine der numerischen Standardfunktionen an Stelle eines `Float`–Wertes ein `Int`–Wert übergeben wird, wird dieser `Int`–Wert durch die Funktion implizit in einen `Float`–Wert konvertiert.

Trigonometrische Funktionen

acos(x(Float)) → Float
Die Funktion *acos*() berechnet den Arcuscosinus von x im Bogenmaß. Eine Ausnahme wird ausgelöst, wenn x nicht im Intervall $[-1, +1]$ liegt. Das Ergebnis liegt im Intervall $[0, \pi]$.

asin(x(Float)) → Float
Die Funktion *asin*() berechnet den Arcussinus von x im Bogenmaß. Eine Ausnahme wird ausgelöst, wenn x nicht im Intervall $[-1, +1]$ liegt. Das Ergebnis liegt im Intervall $[-\pi/2, +\pi/2]$.

atan(x(Float)) → Float
Die Funktion *atan*() berechnet den Arcustangens von x im Bogenmaß. Das Ergebnis liegt im Intervall $(-\pi/2, +\pi/2)$.

atan2(y(Float), x(Float)) → Float
Die Funktion *atan*2() berechnet den Arcustangens von y/x im Bogenmaß. Sie benutzt die Vorzeichen beider Argumente, um den Quadranten des Rückgabewertes zu berechnen. Wenn beide Argumente 0 sind, wird eine Ausnahme ausgelöst. Das Ergebnis liegt im Intervall $[-\pi, +\pi]$.

cos(x(Float)) → Float
Die Funktion *cos*() berechnet den Cosinus von x (angegeben im Bogenmaß).

sin(x(Float)) → Float
Die Funktion *sin*() berechnet den Sinus von x (angegeben im Bogenmaß).

tan(x(Float)) → Float
Die Funktion *tan*() berechnet den Tangens von x (angegeben im Bogenmaß).

acosh(x(Float)) → Float
Die Funktion *acosh*() berechnet den Arcuscosinus Hyperbolicus von x. Wenn x nicht im Intervall $[1, \infty)$ liegt, wird eine Ausnahme ausgelöst.

asinh(x(Float)) → Float
Die Funktion *asinh*() berechnet den Arcussinus Hyperbolicus von x.

atanh(x(Float)) → Float
Die Funktion *atanh*() berechnet den Arcustangens Hyperbolicus von x. Wenn x nicht im Intervall $(-1, +1)$ liegt, wird eine Ausnahme ausgelöst.

cosh(x(Float)) → Float
Die Funktion *cosh*() berechnet den Cosinus Hyperbolicus von x.

sinh(x(Float)) → Float
Die Funktion *sinh*() berechnet den Sinus Hyperbolicus von x.

tanh(x(Float)) → Float
Die Funktion *tanh*() berechnet den Tangens Hyperbolicus von x.

Exponentialfunktionen und logarithmische Funktionen

exp(x(Float)) → Float
Die Funktion $exp()$ berechnet die Exponentialfunktion von x (d.h. e^x).

frexp(x(Float)) → [Float, Int]
Die Funktion $frexp()$ teilt eine Gleitkommazahl in einen normalisierten gebrochenen Anteil ($frac$) und eine ganzzahlige Potenz von 2 (exp) auf, wobei $x = frac \cdot 2^{exp}$. Beide Werte werden als Vektor $[frac, exp]$ zurückgebeben.

Wenn x 0 ist, sind beide Teile des Ergebnisses 0.

ldexp(x(Float), exp(Int)) → Float
Die Funktion $ldexp()$ multipliziert die Gleitkommazahl x mit der ganzzahligen Potenz exp von 2.

log(x(Float)) → Float
Die Funktion $log()$ berechnet den natürlichen Logarithmus von x. Wenn das Argument negativ ist, wird eine Ausnahme ausgelöst. Ist es 0, ist das Ergebnis $-Float :: HUGE\_VAL$.

log10(x(Float)) → Float
Die Funktion $log10()$ berechnet den dekadischen Logarithmus von x. Wenn das Argument negativ ist, wird eine Ausnahme ausgelöst. Ist es 0, ist das Ergebnis $-Float :: HUGE\_VAL$.

modf(x(Float)) → [Float, Float]
Die Funktion $modf()$ teilt das Argument in einen ganzzahligen Anteil (int) und einen gebrochenen Anteil ($frac$) auf, von denen beide das gleiche Vorzeichen wie das Argument haben. Beide Werte werden als Vektor $[int, frac]$ zurückgegeben.

Potenzfunktionen

pow(x(Float), y(Float)) → Float
Die Funktion $pow()$ berechnet x hoch y. Eine Ausnahme wird ausgelöst, wenn x negativ ist und y kein ganzzahliger Wert ist, oder x 0 und y negativ ist. Das Ergebnis ist 1.0, wenn sowohl x und y 0 sind.

sqrt(x(Float)) → Float
Die Funktion $sqrt()$ berechnet die nicht–negative Quadratwurzel von x. Wenn x negativ ist, wird eine Ausnahme ausgelöst.

Rundung, Absolutwert und Rest

ceil(x(Float)) → Float
Die Funktion $ceil()$ berechnet den kleinsten ganzzahligen Wert, der nicht kleiner als x ist.

fabs(x(Float)) → Float
Die Funktion $fabs()$ berechnet den Betrag von x.

floor(x(Float)) → Float
Die Funktion $floor()$ berechnet den größten ganzzahligen Wert, der nicht größer als x ist.

fmod(x(Float), y(Float)) → Float

Für y ungleich 0 berechnet die Funktion $fmod()$ den Wert $x - i \cdot y$, so daß für einen ganzzahligen Wert i das Ergebnis das gleiche Vorzeichen wie x und einen Betrag kleiner dem von y hat. Wenn y 0 ist, wird eine Ausnahme ausgelöst.

Kapitel 4

Basisschnittstellen

Die nachfolgend beschriebenen Basisschnittstellen realisieren grundlegende Konzepte, auf denen die eigentlichen Typen des OFML-Standards basieren. Ein derartiger Typ implementiert eine oder mehrere dieser Basisschnittstellen.

Jeder Schnittstelle ist eine Schnittstellenkategorie zugeordnet (Anh. H). Mittels der unten beschriebenen allgemeinen Methode zur Bestimmung der Kategorie-Zugehörigkeit eines Typs bzw. einer Instanz kann somit – alternativ zur Bestimmung der Typidentität – ermittelt werden, ob ein Typ eine bestimmte Schnittstelle implementiert bzw. ob eine Instanz des Typs die Funktionalität der Schnittstelle bereitstellt.

4.1 MObject

Die Schnittstelle *MObject* definiert die grundlegenden Schnittstellen aller OFML-Typen. Folglich implementiert jeder OFML-Typ zumindest diese Schnittstelle.

4.1.1 Typidentität und Kategorie-Zugehörigkeit

- *getType()* → *Type*
  Die Funktion liefert den unmittelbaren Typ der impliziten Instanz.
- *getClass()* → *String*
  Die Funktion liefert den Namen des unmittelbaren Typ der impliziten Instanz.

  Hinweis: Equivalent zu `String(getType().getName())`.

- *isA(pType(Type))* → *Int*
  Die Funktion überprüft die Zugehörigkeit zu einem übergebenen Typ *pType*. *isA()* liefert 1, wenn *pType* der direkte Typ der impliziten Instanz bzw. ein Supertyp derselben ist. Sonst ist das Ergebnis 0.

- *isCat(pCat(Symbol)) → Int*

  Die Funktion liefert 1, wenn die implizite Instanz der übergebenen Kategorie angehört.

  Hinweis: In der Regel erbt ein Typ die Zugehörigkeit zu Kategorien von seinem unmittelbaren Supertyp. Deswegen ist im Allgemeinen beim Überschreiben der Funktion darauf zu achten, daß bei Übergabe einer Kategorie, die nicht von dem konkreten Typ selber definiert wird, die geerbte Implementierung der Funktion aufgerufen wird.

4.1.2 Instanzidentität und -hierarchie

- *final getName() → String*

  Die Funktion gibt den absoluten Namen der impliziten Instanz zurück.

- *final getFather() → MObject*

  Die Funktion liefert eine Referenz auf das Vaterobjekt. Falls die implizite Instanz keinen Vater besitzt, ist das Ergebnis $NULL$.

- *final getRoot() → MObject*

  Die Funktion liefert eine Referenz auf die Wurzelinstanz der Hierarchie, in der sich die implizite Instanz befindet.

- *final getChildren() → MObject[]*

  Die Funktion gibt eine Liste von Objektreferenzen zurück, die die direkten Kinder der impliziten Instanz repräsentieren. Falls keine Kinder vorhanden sind, wird eine leere Liste zurückgegeben.

- *final getElements() → MObject[]*

  Die Funktion gibt eine Liste von Objektreferenzen zurück, die diejenigen direkten Kinder der impliziten Instanz repräsentieren, die gleichzeitig Elemente sind. Falls keine Elemente vorhanden sind, wird eine leere Liste zurückgegeben.

- *final add(pType(Type) ...) → MObject*

  Die Funktion erzeugt ein Kind der impliziten Instanz vom Typ *pType* und registriert es weiterhin als Element. Der lokale Name des Kindes wird automatisch gewählt. Sofern ein Typ zu seiner Instantiierung zusätzliche Parameter benötigt, müssen diese beim Aufruf von *add()* nach *pType* angegeben werden.

  Der Rückgabewert der Funktion ist eine Referenz auf das erzeugte Objekt oder $NULL$.

- *final remove(pChild(MObject)) → self*

  Die Funktion entfernt das spezifizierte Objekt, das ein Kind der impliziten Instanz ist, aus der Liste der Kinder der impliziten Instanz. Sofern es gleichzeitig ein Element ist, wird es auch aus der Liste der Elemente entfernt.

4.2 Base

Die Schnittstelle *Base* repräsentiert als Erweiterung der Schnittstelle *MObject* ebenfalls eine grundlegende Schnittstelle der OFML-Typen, die von den allermeisten OFML-Typen implementiert wird.

Jeder Typ, der die Schnittstelle *Base* implementiert, implementiert **auch** die Schnittstelle *MObject*.

4.2.1 Instanzvariablen

- *mIsCutable(Int)*

  Die Variable legt die Eigenständigkeit der Instanz in Bezug auf die Cut-Operation des Clipboards fest (Funktionen *setCutable()* und *isCutable()* in Abschn. 4.2.2).

- *[static] eps(Float) = 0.005*

  Die statische Variable *eps* muß für geometrische Vergleichsoperationen aufgrund der limitierten Darstellungsgenauigkeit von Gleitkommazahlen in OFML verwendet werden. Diese Variable, wie auch die folgenden, darf nicht redefiniert werden. Nicht redefinierbare Variablen können mit *final* gekennzeichnet werden.

- *[static] sPi4(Float)* = $\frac{\pi}{4}$
- *[static] sPi2(Float)* = $\frac{\pi}{2}$
- *[static] sPi(Float)* = π
- *[static] s2Pi(Float)* = 2π

4.2.2 Selektierbarkeit

- *final selectable()* → *self*

  Die Funktion erlaubt die Selektion der impliziten Instanz.

- *final notSelectable()* → *self*

  Die Funktion verbietet die Selektion der impliziten Instanz. Im Fall einer versuchten Selektion der impliziten Instanz wird die im Rahmen einer Aufwärtstraversierung als erste selektierbare Instanz selektiert.

- *final hierSelectable()* → *self*

  Die Funktion erlaubt die Selektion aller Instanzen der Teilhierarchie, dessen Wurzelobjekt die implizite Instanz ist. Ob eine einzelne Instanz tatsächlich selektiert werden kann, wird durch den Status bestimmt, der via *selectable()* bzw. *notSelectable()* gesetzt wurde.

- *final notHierSelectable()* → *self*

  Die Funktion verbietet die Selektion aller Instanzen der Teilhierarchie, dessen Wurzelobjekt die implizite Instanz ist. Das Verbot gilt für alle Instanzen der Teilhierarchie, auch wenn die Selektion einer einzelnen Instanz via *selectable()* prinzipiell erlaubt ist. *notHierSelectable()* hat, bezogen auf eine einzelne Instanz, also Vorrang vor *selectable()*.

- *final isSelectable() → Int*

  Die Funktion liefert True, wenn die implizite Instanz selektiert werden kann. Dies ist der Fall, wenn für die Instanz *selectable()* gerufen wurde und für kein Objekt in der Hierarchie über der Instanz *notHierSelectable()* gerufen wurde.

  Instanzen sind initial selektierbar.

4.2.3 Löschbarkeit

- *setCutable(pMode(Int)) → Void*

  Die Funktion legt die Eigenständigkeit der impliziten Instanz in Bezug auf die Cut-Operation des Clipboards fest und speichert den übergebenen Modus in der Instanzvariablen *mIsCutable* ab. Die möglichen Werte sind:

-1	Die implizite Instanz darf generell nicht gelöscht werden.
0	Die implizite Instanz selber darf nicht gelöscht werden, kann aber im Rahmen einer übergeordneten Instanz gelöscht werden. Im Fall einer versuchten Cut-Operation der impliziten Instanz wird die Operation auf die im Zuge einer Aufwärtstraversierung erste Instanz angewendet, für die *isCutable()* 1 liefert.
1	Die implizite Instanz kann gelöscht und ins Clipboard kopiert werden. Dies ist der Initialzustand.
2	Die implizite Instanz kann gelöscht werden, darf aber nicht ins Clipboard kopiert werden.

  Beispiel: Modus 2 wird für Objekte wie Kranzprofile verwendet, die hinsichtlich einer Menge von Basisobjekten konstruiert werden und folglich nicht ohne weiteres an eine andere räumliche bzw. topologische Position kopiert werden können.

- *isCutable() → Int*

  Die Funktion fragt die Eigenständigkeit der impliziten Instanz in Bezug auf die Cut-Operation des Clipboards ab. Sie liefert den Wert der Instanzvariablen *mIsCutable*, die mit Hilfe der Funktion *setCutable()* beschrieben werden kann.

- *removeValid() → Int*

  Die Funktion liefert True, wenn die implizite Instanz gelöscht werden darf.

  Instanzen sind initial löschbar.

  Im Gegensatz zur Funktion *isCutable()*, die die prinzipielle Löschbarkeit eines Objektes spezifiziert und von der Applikation vor einer Cut-Operation auf dem selektierten Objekt aufgerufen wird, dient die Funktion *removeValid()* zur Modellierung dynamischer Aspekte der Löschbarkeit und wird innerhalb von REMOVE_ELEMENT Regeln von Vaterinstanzen aufgerufen.

4.2.4 Sichtbarkeit

- *final hide() → self*

  Die Funktion verbirgt die implizite Instanz einschließlich ihrer Kinder und Kindeskinder. Das Verbergen von Instanzen hat keinen Einfluß auf die Kollisionserkennung.

- *final show() → self*

  Die Funktion macht die implizite Instanz wieder sichtbar, falls sie vorher verborgen war.

- *final isHidden() → Int*

  Die Funktion drückt durch ihren Rückgabewert aus, ob die implizite Instanz sichtbar (0) oder unsichtbar (1) ist.

4.2.5 Auflösung

Über die nachfolgenden Funktionen kann die Objektraumauflösung eines Objekts gesetzt bzw. abgefragt werden. Dies betrifft im allgemeinen die Abbildung einer analytischen oder parametrischen Primitive in eine stückweise lineare Approximation. Die direkte Umsetzung erfolgt nur für die geometrischen Grundtypen (Kap. 7). Alle anderen Typen bzw. Instanzen leiten die Auflösung lediglich weiter. Importierte polygonale Datensätze bleiben davon unbeeinflußt.

Normalerweise soll die Auflösung nur für die Wurzel einer Objekthierarchie direkt gesetzt werden. Das direkte Setzen der Objektraumauflösung für ein Nicht-Wurzel-Objekt ist dennoch erlaubt.

Die Auflösung wird durch eine Gleitkommazahl r im Bereich $0.0 \leq r \leq 1.0$ angegeben. Dabei steht 0.0 für die minimal mögliche Auflösung und 1.0 für die maximale Auflösung. Wird für eine parametrische Primitive die Auflösung 0.0 angegeben, entspricht ihre Repräsentation dem polygonalen Körper, der sich durch entsprechende Verbindung der definierenden Vertices ergibt. Die initiale Auflösung beträgt 0.1.

- *final setResolution(pRes(Float)) → self*

  Die Funktion setzt die Objektraumauflösung für den durch die implizite Instanz vorgegebenen Teilbaum. Diese Auflösung wird im Teilbaum entsprechend weiter vererbt. Enthält ein Nachfahr bereits eine explizit zugewiesene Objektraumauflösung, wird die rekursive Vererbung an dieser Stelle und für diesen Pfad des Teilbaumes beendet.

- *final getResolution() → Float*

  Die Funktion gibt die für die implizite Instanz gültige Objektraumauflösung zurück.

4.2.6 Änderungsstatus

- *final setChanged() → self*

  Die Funktion markiert die implizite Instanz explizit als verändert bezüglich dem Zeitpunkt unmittelbar nach der Ausführung der Initialisierung. Ein expliziter Aufruf von *setChanged()*

ist notwendig, wenn Instanzvariablen direkt beschrieben werden, ohne daß andere verändernde Operationen ausgeführt wurden (z.B. Erzeugung von Kindern, Verschiebung). Der Änderungsstatus wird ausgewertet, um eine effiziente Abspeicherung von Instanzhierarchien zu ermöglichen, was bei Clipboard- und Persistenz-Operationen Anwendung findet.

- *final setUnchanged() → self*

  Die Funktion setzt den Änderungszustand der impliziten Instanz auf den Zustand unmittelbar nach der Ausführung der Initialisierung zurück. D.h., die Instanz gilt nun als unverändert in Bezug auf den Zeitpunkt unmittelbar nach der Initialisierung.

4.2.7 Kollisionserkennung

- *final disableCD() → Void*

  Die Funktion schaltet die Kollisionserkennung für die implizite Instanz aus. Danach wird die implizite Instanz einschließlich ihrer Kinder von der Kollisionserkennung ignoriert.

- *final enableCD() → Void*

  Die Funktion schaltet die Kollisionserkennung für die implizite Instanz (wieder) ein.

- *final isEnabledCD() → Int*

  Die Funktion liefert 0, falls die implizite Instanz von der Kollisionserkennung ausgeschlossen ist, ansonsten 1.

4.2.8 Bemaßung

- *measure(pMode(Symbol)) → Void*

  Die Funktion schaltet die Bemaßung der impliziten Instanz ein. Über einen implementierungsabhängigen Wert *pMode* können ggf. verschiedene Arten der Bemaßung gewählt werden. Existiert nur eine Art der Bemaßung, kann der Parameter ignoriert werden.

  Folgende Symbole sind für die Bemaßung vordefiniert:

  - *@ISO* – Die Bemaßung erfolgt in Meter.
  - *@INCH* – Die Bemaßung erfolgt in Zoll.

- *unMeasure() → Void*

  Schaltet die Bemaßung der impliziten Instanz aus.

4.2.9 Räumliche Modellierung

- *final setPosition(pPosition(Float[3])) → self*

  Die Funktion setzt die lokale Position[1] der impliziten Instanz unbedingt, d.h., es werden keine Regeln aufgerufen und die Translationsfreiheitsgrade werden ignoriert auf die Position

[1] Tatsächlich wird jedoch das lokale Koordinatensystem an die entsprechende Position relativ zum Vater verschoben.

pPosition. Diese stellt gleichzeitig die Verschiebung der impliziten Instanz gegenüber dem Vater der impliziten Instanz dar. Initial hat eine Instanz keine Verschiebung bezüglich ihres Vaters. Falls kein Vater existiert, dient das Weltkoordinatensystem als Bezug.

Die Funktion wird für die explizite Positionierung innerhalb von Funktionen verwendet.

- *final getPosition() → Float[3]*

  Die Funktion liefert die aktuelle Verschiebung der impliziten Instanz bezüglich ihres Vaters, sofern dieser existiert oder bezüglich des Weltkoordinatensystems.

- *final translate(pVector(Float[3])) → self*

  Die Funktion verschiebt die implizite Instanz bedingt um den im Weltkoordinatensystem definierten Vektor *pVector*. Die Bedingtheit der Verschiebung ergibt sich durch eine mögliche Rasterisierungs- und Snapping-Funktionalität, sofern von der OFML-Laufzeitumgebung unterstützt, translatorische Freiheitsgrade (*setTrAxis()*), sowie durch das Vorhandensein von Regeln der Ursache *TRANSLATE* (Kap. 5). Sofern *TRANSLATE*-Regeln für die implizite Instanz definiert sind, werden diese nach Ausführung der Translation unmittelbar aufgerufen.

  Die Funktion wird für die interaktive Positionierung über direkte Manipulation oder Benutzungsoberfläche verwendet. Der Vektor *pVector* wird aus dem Weltkoordinatensystem in das lokale Koordinatensystem der impliziten Instanz unter Berücksichtigung der aktuellen geerbten und lokalen Modellierung der impliziten Instanz transformiert. Somit wird eine intuitive Modellierung über die Funktion *translate()* sichergestellt.

- *final moveTo(pPosition(Float[3])) → self*

  Die Funktion verschiebt die implizite Instanz bedingt an die im Weltkoordinatensystem definierte Position *pPosition*. Die Semantik der Funktion entspricht vollständig dem Aufruf von *translate()* mit dem Vektor *pPosition – getWorldPosition()*.

- *final setTrAxis(pAxis(Int)) → self*

  Die Funktion erlaubt oder verbietet die Verschiebbarkeit der impliziten Instanz für einzelne Achsen des lokalen Koordinatensystems. Der Parameter *pAxis* ergibt sich aus der Addition der erlaubten Achsen, wobei x-, y- und z-Achse durch 1, 2 und 4 repräsentiert werden. Falls *pAxis* den Wert 0 besitzt, kann das Objekt nicht transliert werden.

- *final getTrAxis() → Int*

  Die Funktion liefert die aktuelle Verschiebbarkeit der impliziten Instanz.

- *final rotate(pAxis(Symbol), pArc(Float)) → self*

  Die Funktion verdreht die implizite Instanz bedingt um den bezüglich der lokalen Koordinatenachse *pAxis* im Bogenmaß definierten Winkel *pArc*[2]. Die Bedingtheit der Verdrehung ergibt sich durch eine mögliche Rasterisierungs- und Snapping-Funktionalität, sofern von der OFML-Laufzeitumgebung unterstützt, rotatorische Freiheitsgrade (*setRtAxis()*), sowie durch das Vorhandensein von Regeln der Ursache *ROTATE* (Kap. 5). Sofern *ROTATE*-Regeln für die implizite Instanz definiert sind, werden diese nach Ausführung der Drehung unmittelbar aufgerufen.

[2] Tatsächlich wird jedoch das lokale Koordinatensystem entsprechend verdreht.

pAxis ist entweder *@PX*, *@PY* oder *@PZ*, soweit eine Drehung um die (positive) x-, y- oder z-Achse erfolgt. Alternativ kann eine Drehung um eine entgegengesetzte Achse erfolgen. Die dementsprechenden Symbole sind *@NX*, *@NY* und *@NZ*. Die Verdrehung um beliebige Achsen wird durch entsprechende aufeinanderfolgende Drehungen um die elementaren Achsen erreicht.

Im Gegensatz zur Translation gibt es für die Rotation keine Funktion zum unbedingten Setzen. Das Setzen einer bestimmten Orientierung erfolgt entweder initial, d.h., wenn noch keine Verdrehung gegenüber dem Vater stattgefunden hat, oder durch eine Subtraktion der aktuellen Orientierung von der zu setzenden Orientierung. Die oben beschriebene Bedingtheit wird dadurch aber nicht außer Kraft gesetzt.

Innerhalb von Regeln erfolgt die Korrektur von Orientierungen durch erneute Anwendung der *rotate()*-Funktion. Dabei ist vom OFML-Laufzeitsystem sicherzustellen, daß kein erneuter Aufruf der *ROTATE*-Regeln stattfindet.

Ein allgemeines Problem der Verdrehung um kartesische Achsen besteht in der Überlagerung der drei elementaren Rotationen. Aus diesem Grund und um eine korrekte Funktionsweise der Rotationen sicherzustellen wird empfohlen, zu einem Zeitpunkt nur höchstens eine Rotationsachse freizugeben (*setRtAxis()*).

Die Funktion wird sowohl für die interaktive Positionierung über direkte Manipulation oder Benutzungsoberfläche als auch für die explizite Positionierung innerhalb von Funktionen verwendet.

- *final getRotation(pAxis(Symbol)) → Float*

  Die Funktion liefert die aktuelle Verdrehung im Bogenmaß um die durch *pAxis* spezifizierte Rotationsachse.

  Achtung: Falls eine Instanz um mehr als eine Achse gedreht wurde, liefert *getRotation()* möglicherweise unerwartete Ergebnisse. Dies liegt an dem prinzipiellen Problem der Überlagerung der drei elementaren kartesischen Rotationen.

- *final setRtAxis(pAxis(Int)) → self*

  Die Funktion erlaubt oder verbietet die Drehbarkeit der impliziten Instanz für einzelne Achsen des lokalen Koordinatensystems. Der Parameter *pAxis* ergibt sich aus der Addition der erlaubten Achsen, wobei x-, y- und z-Achse durch 1, 2 und 4 repräsentiert werden. Falls *pAxis* den Wert 0 besitzt, kann das Objekt nicht gedreht werden.

  Instanzen sollen maximal um eine Rotationsachse drehbar sein. Allerdings kann diese über die Zeit verändert werden.

- *final getRtAxis() → Int*

  Die Funktion liefert die aktuelle Drehbarkeit der impliziten Instanz.

- *final getLocalBounds() → Float[2][3]*

  Die Funktion liefert das minimale achsenorthogonale Begrenzungsvolumen der impliziten Instanz bezogen auf deren lokales Koordinatensystem. Das Begrenzungsvolumen schließt die Kinder und den Ursprung des lokalen Koordinatensystems mit ein.

  Der Rückgabewert ist ein Vektor aus zwei Elementen. Das erste Element ist die minimale Koordinate innerhalb des lokalen Begrenzungsvolumens. Das zweite Element ist die maximale Koordinate innerhalb des lokalen Begrenzungsvolumens.

Die OFML-Laufzeitumgebung muß sicherstellen, daß sich das lokale Begrenzungsvolumen stets in einem konsistenten Zustand befindet.

- *final getLocalGeoBounds() → Float[2][3]*

  Die Funktion liefert das minimale achsenorthogonale Begrenzungsvolumen der impliziten Instanz bezogen auf deren lokales Koordinatensystem. Im Gegensatz zu *getLocalBounds()* schließt das Begrenzungsvolumen aber den Ursprung des lokalen Koordinatensystems und Kinder mit leerer Geometrie nicht mit ein.

- *final getWorldBounds() → Float[2][3]*

  Die Funktion liefert das minimale achsenorthogonale Begrenzungsvolumen der impliziten Instanz bezogen auf das Weltkoordinatensystem. Das Begrenzungsvolumen schließt die Kinder mit ein.

  Der Rückgabewert ist ein Vektor aus zwei Elementen. Das erste Element ist die minimale Koordinate innerhalb des globalen Begrenzungsvolumens. Das zweite Element ist die maximale Koordinate innerhalb des globalen Begrenzungsvolumens.

  Die OFML-Laufzeitumgebung muß sicherstellen, daß sich das globale Begrenzungsvolumen stets in einem konsistenten Zustand befindet.

- *final getWorldGeoBounds() → Float[2][3]*

  Die Funktion liefert das minimale achsenorthogonale Begrenzungsvolumen der impliziten Instanz bezogen auf das Weltkoordinatensystem. Im Gegensatz zu *getWorldBounds()* schließt das Begrenzungsvolumen aber Kinder mit leerer Geometrie nicht mit ein.

- *final getDistance(pDirection(Symbol)) → Float*

  Die Funktion ermittelt den kürzesten Abstand der impliziten Instanz entlang einer von sechs Richtungen, ausgehend vom lokalen Begrenzungsvolumen, zu einer anderen Instanz in der Szene. Die durch *pDirection* angegebene Richtung hat dabei einen der folgenden Werte: *@NX*, *@PX*, *@NY*, *@PY*, *@NZ*, *@PZ*.

  Der Rückgabewert ist der Abstand, sofern eine andere Instanz ermittelt werden konnte oder -1.

4.2.10 Regelaufruf

- *final callRules(pReason(Symbol), pArg(Any)) → Int*

  Die Funktion löst die Abarbeitung der für die Ursache *pReason* definierten Regeln aus. *pReason* ist entweder eine vordefinierte Regel (Kap. 5) oder eine benutzerdefinierte Regel.

  Der explizite Aufruf einer vordefinierten Regelursache kann beispielsweise verwendet werden, um ein Snapping-Verhalten, das durch eine *TRANSLATE*-Regel implementiert wird, nach der initialen Positionierung der Instanz explizit anzufordern. Bei Aufruf einer vordefinierten Regelursache muß *pArg* der Spezifikation in Kap. 5 entsprechen.

  Der explizite Aufruf einer benutzerdefinierten Regelursache über *callRules()* ist die einzige Möglichkeit, die entsprechenden Regeln zur Abarbeitung zu bringen. Eine prinzipielle Anwendbarkeit benutzerdefinierter Regelursachen besteht darin, eine Kommunikation zwischen

Instanzen zu ermöglichen, die flexibler und robuster als die Kommunikation über Funktionen der Typen ist. Hierbei entfällt die Notwendigkeit zur Überprüfung von Typkompatibilität; falls für eine bestimmte Ursache keine Regeln in einem Typ definiert sind, kommt es nicht zu einem Fehler. Dagegen führt der Aufruf einer Funktion für einen Typ, der diese Funktion nicht definiert, stets zu einem Fehler.

Beispiel: In einem System können Strahler verplant werden, die normalerweise nicht verschoben werden können. Als Kinder einiger weniger Typen können sie aber auf spezifische Weise verschoben werden. In der *TRANSLATE*-Regel des Strahlers wird die Verschiebung zurückgesetzt; der Strahler wird also nicht verschoben. Danach folgt ein Aufruf des Vaters mit *callRules()*, wobei die benutzerdefinierte Ursache *MOVE_SPOT* die gewünschte neue Position des Strahlers und der Strahler selbst übergeben werden. Sofern der Vater eine Bewegung des Strahlers erlaubt, kann er diese durch eine entsprechende *MOVE_SPOT*-Regel kontrollieren. Andernfalls bleibt der Strahler unverändert.

Der Rückgabewert ist -1, falls eine Regel der Ursache *pReason* fehlschlug. Andernfalls ist der Rückgabewert 0.

4.2.11 Dynamische Merkmale

Eigenschaften einer Instanz, deren jeweils aktuelle Ausprägung in einer entsprechenden Instanzvariablen gespeichert sind (und über entsprechende set- und get-Funktionen zugewiesen bzw. abgerufen werden können), werden als *statische* Eigenschaften bzw. Merkmale der Instanz bezeichnet. Im Gegensatz dazu ist es manchmal erforderlich, einer Instanz während der Zeit seiner Existenz dynamisch Merkmale und Werte zuzuweisen. Die Schnittstelle *Base* verwaltet dazu pro Instanz eine interne Hash-Tabelle, in der solche Merkmale angelegt werden können. Ein Merkmal wird dabei über seinen eindeutigen Schlüssel vom Typ *Symbol* definiert und angesprochen. Der in der Tabelle zu dem Schlüssel eingetragene Wert kann von einem einfachen Typ oder von den Referenztypen *String*, *Vector*, *List* und *Hash* sein.

- *getDynamicProps()* $\rightarrow$ *Hash*

  Die Funktion liefert die (Referenz auf die) Hash-Tabelle für dynamische Merkmale.

4.2.12 2D–Darstellung und ODB

Objekte, die die Schnittstelle *Base* implementieren, können über eine 2D–Repräsentation verfügen. Diese wird mittels der Methoden *getOdbInfo()*, *getPictureInfo()*, *invalidatePicture()* und den Methoden zum Erzeugen von primitiven 2D–Objekten, wie in Anhang B beschrieben, erzeugt.

Über *getOdbInfo()* und *getPictureInfo()* erzeugte 2D–Symbole lassen sich nicht gleichzeitig verwenden. Wird durch *getOdbInfo()* eine Hash–Tabelle zurückgegeben, wird ein eventuell mittels getPictureInfo() angegebenes Symbol nicht dargestellt.

- *getOdbInfo()* $\rightarrow$ *Hash*

  Die Funktion wird durch das Kernsystem zu beliebigen Zeitpunkten aufgerufen, um die aktuelle ODB–Information, die zur Erstellung eines 2D–Symbols oder der 3D–Geometrien

benötigt wird, abzufragen. Die Funktion gibt die ODB–Information in Form einer Hash–Tabelle zurück, oder *NULL*, wenn keine ODB–Information für das Objekt verfügbar ist. Die Verwendung der ODB ist in [ODB] beschrieben.

- *getPictureInfo()* → *Vector*

  Die Funktion wird durch das Kernsystem zu beliebigen Zeitpunkten aufgerufen, um Informationen über das für dieses Objekt zu verwendende 2D–Symbol abzufragen. Der Rückgabewert ist ein aus drei Elementen bestehender Vektor:

  – Das erste Element ist entweder *NULL*, in welchem Fall dieses Objekt kein 2D–Symbol besitzt, oder der voll qualifizierte Name des Symbols. Mit dem Namen wird nach einem entsprechenden EGM–, DMP– oder FIG–Symbol gesucht. Das zuerst gefundene Symbol wird für die Darstellung im 2D–Mode verwendet. Kann kein Symbol gefunden werden, wird ausgehend von den 3D–Geometrien für das Objekt und alle seine aktuellen Kind–Objekte ein Symbol automatisch generiert.

  – Das zweite Element ist entweder *@TRAVERSAL_STOP* oder *@TRAVERSAL_CONT*. Es bestimmt, ob bei der Darstellung im 2D–Mode eventuell vorhandene Symbole von Kindobjekten dargestellt werden sollen (*@TRAVERSAL_CONT*) oder nicht (*@TRAVERSAL_STOP*). Wird für das Objekt ein Symbol automatisch generiert, sollte dieser Wert auf jeden Fall auf *@TRAVERSAL_STOP* gesetzt werden.

  – Das dritte Element ist entweder *@SHARE_ON* oder *@SHARE_OFF*. Es legt fest, ob Symbole mit gleichem Namen von verschiedenen Objekten gemeinsam genutzt werden sollen (*@SHARE_ON*) oder ob das Symbol für jedes Objekt erneut geladen oder erzeugt wird (*@SHARE_OFF*). Für Symbole, die aus Dateien geladen werden, sollte hier für gewöhnlich *@SHARE_ON* angegeben werden. Für Objekte, deren 2D–Symbol automatisch generiert wird, kann *@SHARE_ON* angegeben werden, wenn verschiedene Instanzen mit gleichem Symbol–Namen immer über die gleiche 3D–Geometrie verfügen, andernfalls sollte *@SHARE_OFF* verwendet werden. Es ist allerdings zu beachten, daß Kindobjekte, z.B. Zubehör, Bestandteil des Symbols werden, so daß bei der gemeinsamen Benutzung von automatisch generierten Symbolen für Objekte, die unter Umständen zusätzliche Kinder enthalten können, diese Kinder entweder bei allen Objekten oder bei keinem sichtbar sind.

  Standardmäßig gibt *getPictureInfo()* als Symbolnamen den Typ der Klasse der impliziten Instanz zurück, erlaubt das Traversieren von Kindobjekten bei der Darstellung im 2D–Mode und unterbindet die gemeinsame Nutzung von Symbolen.

  Hinweis: Eine Änderung in den Kind–Objekten bewirkt nicht automatisch eine Anpassung eines automatisch generierten Symbols.

  Auf die automatische Generierung von 2D–Symbolen sollte nach Möglichkeit verzichtet werden, da sie insbesondere bei gehäufter Anwendung zu spürbaren Verzögerungen führen kann und das Ergebnis für eine effektive Planung im 2D–Mode in der Regel unbefriedigend ist.

- *invalidatePicture()* → *Void*

  Die Funktion muß aufgerufen werden, nachdem sich Eigenschaften des Objekts, die Einfluß auf die 2D– oder 3D–Geometrie haben, geändert haben. Das Kernsystem verwirft alle gepspeicherten Informationen (Rückgabewerte von *getOdbInfo()* und *getPictureInfo()* sowie

2D–Symbole (ODB, EGM, DMP, FIG, und generierte)). Bei Bedarf werden diese Informationen erneut abgefragt und die 2D–Symbole erzeugt.

- *createOdbObjects(pUpdate(Int))* → *Void*

  Die Funktion erzeugt Kindobjekte entsprechend der Spezifikation in der ODB. Wenn der Parameter *pUpdate* 0 ist (falsch), werden all derzeit existierenden durch die ODB erzeugten Kindobjekte gelöscht und anschließend neu erzeugt. Ist der Parameter *pUpdate* 1 (wahr), erfolgt eine Anpassung der existierenden durch die ODB erzeugten Kindobjekte.

  Hinweis: In der aktuellen OFML-Implementierung der EasternGraphics GmbH werden unabhängig vom Parameter *pUpdate* alle von der ODB erzeugten Kindobjekte gelöscht und anschließend durch die ODB neu erzeugt.

4.3 Material

Die Schnittstelle *Material* definiert die Funktionen zur Behandlung von Materialeigenschaften (Oberflächenbeschaffenheiten) auf der Basis von Materialkategorien. Alle Typen, deren Instanzen Materialeigenschaften zugewiesen werden können, müssen diese Schnittstelle implementieren.

In einer Materialkategorie werden alle Möbel bzw. Möbelteile zusammengefaßt, deren Materialien aus funktionalen und/oder ästhetischen Gesichtspunkten gleich behandelt werden sollen. Eine Instanz kann dabei einer oder mehreren Materialkategorien angehören. Materialkategorien werden durch Symbole bezeichnet.

Vordefinierte Materialkategorien sind in Anhang H aufgelistet.

Beispiel: Typische Materialkategorien sind: *Korpus*, *Front*, *Sockel*, *Arbeitsplatte*. Instanzen eines Schranktyps mit Türen und/oder Auszügen würden dann den Kategorien *Korpus*, *Front* und *Sockel* zugehören, während z.B. die Kindinstanzen, die den Korpus realisieren, nur zur Kategorie *Korpus* gehören. Die korrespondierenden OFML-Materialkategorien könnten lauten: *@KORPUS*, *@FRONT*, *@SOCKEL* und *@PLATTE*.

Hinweis: Die „allgemeingültige“ Materialkategorie *@ANY* ist vordefiniert (Funktion *setCMaterial()*) und darf nicht anderweitig verwendet werden.

Für jede Materialkategorie wird eine eingeschränkte Menge möglicher Materialien festgelegt, die ebenfalls durch Symbole bezeichnet werden (Funktion *getMatCategories()*). Materialbezeichner sind über alle Materialkategorien hinweg eindeutig. Die visuellen Eigenschaften eines Materials werden in einer separaten Datei spezifiziert, deren Format im Anhang D.2 beschrieben ist. Jedem Materialbezeichner muß ein Materialname zugeordnet werden (Funktion *getMatName()*), um anhand des Namens zur Laufzeit die entsprechende Materialbeschreibungsdatei lesen zu können.

Beispiel: Für die Kategorie *Korpus* sind z.B. die Materialien „Laminat grau“ und „Furnier Buche hell“ vorgesehen, für die Kategorie *Front* nur das Material „Furnier Buche hell“. Mögliche Bezeichner für diese

Materialien wären *@LGrau* bzw. *@FBhell*[3]. Entsprechende Materialnamen wären „Laminat grau“ bzw. „Buche hell“ und die zugehörigen Materialbeschreibungsdateien `laminatgrau.mat` bzw. `buchehell.mat`.

4.3.1 Materialkategorien

- *getMatCategories() → Symbol[]*

  Liefert die Liste der aktuell für die implizite Instanz definierten Materialkategorien. Eine Instanz, für die selber keine Materialkategorien definiert sind, liefert entweder eine leere Liste oder *Void*. In letzterem Fall sollen für die implizite Instanz die für die Vaterinstanz definierten Materialkategorien angewendet werden.

  Die Menge der für die implizite Instanz definierten Materialkategorien kann sich dynamisch ändern und sich somit von der Menge aller potentiell möglichen Materialkategorien unterscheiden (Funktion *getAllMatCats()*).

- *isMatCat(pCat(Symbol)) → Int*

  Liefert 1, wenn die übergebene Materialkategorie zu der Menge der aktuell für die implizite Instanz definierten Materialkategorien gehört, andernfalls 0.

- *getAllMatCats() → Symbol[]*

  Liefert die Liste aller potentiell für die implizite Instanz definierbaren Materialkategorien (siehe auch Funktion *getMatCategories()*).

- *getCMaterials(pCat(Symbol)) → Symbol[]*

  Liefert die Liste aller innerhalb der übergebenen Materialkategorie für die implizite Instanz anwendbaren Materialien. Der Rückgabewert ist vom Typ *Void*, wenn die übergebene Materialkategorie nicht zu den aktuell für die implizite Instanz definierten Materialkategorien gehört.

4.3.2 Materialien

- *setCMaterial(pCat(Symbol), pMat(Symbol)) → Int*

  Weist der impliziten Instanz in der übergebenen Materialkategorie das angegebene Material zu. Die Operation wird rekursiv auf alle Kinder und Kindeskinder angewendet. Die Funktion hat keinen Effekt, wenn weder die implizite Instanz noch eines der Kinder der übergebenen Materialkategorie angehört. Der Rückgabewert ist 1, wenn der impliziten Instanz oder mindestens einem Nachfahr (Kind, Enkel, etc.) derselben das Material zugewiesen werden konnte, andernfalls 0.

  Die vordefinierte „allgemeingültige“ Materialkategorie *@ANY* kann zum expliziten Zuweisen eines Materials ohne Berücksichtigung der Zugehörigkeit der impliziten Instanz zu einer konkreten Materialkategorie verwendet werden.

[3] Für die symbolischen Materialbezeichner bieten sich die bereits in Herstellerunternehmen verwendeten Materialkürzel an.

- *getCMaterial(pCat(Symbol))* → *Symbol*

  Liefert das der impliziten Instanz in der übergebenen Materialkategorie aktuell zugewiesene Material bzw. einen Wert vom Typ *Void*, wenn die implizite Instanz aktuell nicht der übergebenen Materialkategorie angehört.

- *getMatName(pMat(Symbol))* → *String*

  Liefert für die implizite Instanz den Materialnamen zu dem übergebenen Material bzw. einen Wert vom Typ *Void*, wenn es sich um ein unbekanntes Material handelt. Die Standardimplementierung ruft die gleichnamige Funktion des Vaters auf.

4.4 Property

Properties sind Objekteigenschaften, die interaktiv durch den Systemanwender mit Hilfe von geeigneten Dialogen (Property-Editoren) geändert werden können. Die Schnittstelle *Property* definiert die Funktionen zum Umgang mit Properties. Properties können mit Merkmalen in Produktdatenbanken assoziiert sein (Kap. 9).

4.4.1 Festlegen von Properties

- *setupProperty(pKey(Symbol), pDef(Any[5]), pPos(Int))* → *Void*

  Die Funktion legt eine Property mit dem angegebenen Schlüssel (Identifikator) und der übergebenen Spezifikation an. Ist bereits eine Property mit dem angegebenen Schlüssel registriert, wird deren Spezifikation durch die Parameterwerte überschrieben.

  Die Definition einer Property (Parameter *pDef*) ist ein Vektor aus fünf Werten:

pName(String)	der Name der Property (erscheint im Property-Editor). Dabei kann es sich um einen Platzhalter handeln, der über eine externe Ressourcen-Datei aufgelöst wird (Anh. D).
pMin(Any)	untere (inklusive) Grenze des Wertebereichs
pMax(Any)	obere (inklusive) Grenze des Wertebereichs bzw. maximale Länge bei String-Properties
pFmt(String)	gewünschtes spezielles Ein-/Ausgabeformat (Syntax und Bedeutung lt. Anh. E.1)
pType(String)	der Typ der Property: *b* boolescher Wert *i* ganze Zahl *f* reelle Zahl *s* String *ch* Auswahlliste (*choice list*) Der Typangabe folgt nach einem Leerzeichen die Liste der Auswahlwerte. Jeder Auswahlwert ist entweder eine durch ein vorangestelltes @-Zeichen gekennzeichnete (sprachneutrale) String-ID oder ein

durch Leerzeichen getrenntes Paar aus String-ID und sprachabhängiger Bezeichnung (Anh. D). Die Auswahlwerte sind durch Leerzeichen getrennt. Ist für einen Wert keine sprachabhängige Bezeichnung angegeben, wird diese anhand der String-ID aus sprachabhängigen Bezeichnungsdateien gelesen.

chf Auswahlliste über Funktion
Der Typangabe folgt der Name einer Funktion, die, für die implizite Instanz aufgerufen, die Liste der Auswahlwerte in derselben Form liefert wie die explizite Angabe in einer Property vom Typ *ch*.

u Spezial-Typ (*user defined*)
Der Typangabe folgt nach einem Leerzeichen die ID des geforderten Spezial-Editors und (nach einem weiteren Leerzeichen) evtl. weitere Angaben für den Spezial-Editor.

Hinweis: Es ist nicht garantiert, daß der Spezial-Editor in der jeweils verwendeten OFML–Laufzeitumgebung implementiert ist.

Neben der eigentlichen Property-Definition kann im Parameter *pPos* die gewünschte Position in der Property-Liste angegeben werden. Für das Setzen der Position gilt dieselbe Spezifikation wie für die Funktion *setPropPosOnly()*, mit der die Position für eine existierende Property einzeln gesetzt werden kann.

Die Wertebereichsgrenzen, das Format und die Position sind optional. Fehlende Angaben werden durch einen Parameter vom Typ *Void* gekennzeichnet.

Zu jeder Property kann im Typ der impliziten Instanz eine *set*- und eine *get*-Methode definiert sein:

- *set<Key>(pValue(Any)) → Void*
  Die Funktion wird aufgerufen, wenn der Wert der Property <Key> geändert wurde.

  Hinweis: In dieser Funktion erfolgt in der Regel eine Zuweisung des Wertes an eine entsprechende Instanzvariable. Jede weitere Semantik, wie beispielsweise die Neugenerierung der Geometrie oder entsprechende Kollisionstests, ist der Funktion *propsChanged()* vorbehalten.

- *get<Key>() → Any*
  Liefert den aktuell in der impliziten Instanz gespeicherten Wert für die Property <Key>.

  Hinweis: Die Funktion liefert in der Regel den Inhalt einer entsprechenden Instanzvariable.

  Ein Rückgabewert vom Typ *Void* kennzeichnet ein nicht spezifiziertes Property, z.B. bei optionalen Merkmalen.

- *setPropPosOnly(pKey(Symbol), pPos(Int)) → Int*

  Die Funktion legt für die Property mit dem angegebenen Schlüssel die gewünschte Position in der Property-Liste fest. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, hat die Funktion keinen Effekt und der Rückgabewert ist vom Typ *Void*. Ist für die implizite Instanz eine Property mit dem angegebenen Schlüssel definiert, wird die alte Positionsangabe überschrieben. Ist *pPos* eine ganze Zahl größer gleich 0 und wurde

die gewünschte Position bereits für eine andere Property vergeben, so werden diese und alle in der Positionsliste nachfolgenden Properties um eine Position nach hinten verschoben. Ist *pPos* vom Typ *Void* oder hat es den Wert -1, ist für die Property keine spezielle Position gefordert. Sie wird dann in der Property-Liste nach den Properties einsortiert, für die explizit eine Position angefordert wurde. Rückgabewert ist die neue Position der Property bzw. -1, wenn keine spezielle Position gefordert ist.

- *setExtPropOffset(pOffset(Int))* $\rightarrow$ *Void*

  Mit dieser Funktion wird der impliziten Instanz ein Offset für Positionen von extern definierten Properties zugewiesen, d.h. von Properties, die von einer anderen als der impliziten Instanz für diese definiert werden. Der Offset gibt die kleinste Positionsnummer an, die für extern definierte Properties verwendet werden darf.

  Beispiel: Ein typisches Beispiel für extern definierte Properties sind solche, die seitens einer globalen Produktdatenverwalter-Instanz (Abschn. 9.1) zur Abbildung von Produktmerkmalen aus der Produktdatenbank für die implizite Instanz definiert werden.

- *removeProperty(pKey(Symbol))* $\rightarrow$ *Void*

  Die Funktion entfernt die durch den angegebenen Schlüssel spezifizierte Property aus der Property-Liste. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, hat die Funktion keinen Effekt.

- *clearProperties()* $\rightarrow$ *Void*

  Die Funktion entfernt alle Properties aus der Property-Liste.

4.4.2 Abfragen von Properties

- *hasProperties()* $\rightarrow$ *Int*

  Die Funktion liefert 1, wenn für die implizite Instanz Properties definiert sind, andernfalls 0.

- *hasProperty(pKey(Symbol))* $\rightarrow$ *Int*

  Die Funktion liefert 1, wenn für die implizite Instanz eine Property mit angegebenen Schlüssel definiert ist, andernfalls 0.

- *getPropertyDef(pKey(Symbol))* $\rightarrow$ *Any[]*

  Die Funktion liefert die Definition der Property mit angegebenen Schlüssel. Der Aufbau des zurückgegebenen Vektors entspricht dem Aufbau des als Property-Definition an die Funktion *setupProperty()* übergebenen Parameters *pDef*. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, ist der Rückgabewert vom Typ *Void*.

- *getPropertyPos(pKey(Symbol))* $\rightarrow$ *Int*

  Die Funktion liefert die Position der Property mit angegebenen Schlüssel. Wurde für die Property keine spezielle Position angefordert, ist der Rückgabewert -1. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, ist der Rückgabewert vom Typ *Void*.

- *getExtPropOffset()* → *Int*

  Die Funktion liefert den Offset für Positionen von extern definierten Properties, d.h. von Properties, die von einer anderen als der impliziten Instanz für diese definiert werden. Der Offset gibt die kleinste Positionsnummer an, die für extern definierte Properties verwendet werden darf. Dieser Offset ist von einer externen Instanz vor der Definition einer Property für die implizite Instanz abzurufen und bei der Vergabe von expliziten Positionen zu berücksichtigen.

  Wurde mittels *setExtPropOffset()* kein anderer Wert zugewiesen, ist der Default-Rückgabewert gleich 0.

- *getPropertyKeys()* → *Symbol[]*

  Die Funktion liefert eine Liste der Schlüssel aller aktuell für die implizite Instanz definierten Properties.

  Die Properties werden dabei nach ihren expliziten Positionen in aufsteigender Reihenfolge sortiert. Die Properties ohne explizite Position erscheinen am Ende der Liste in undefinierter Reihenfolge.

- *getProperties()* → *String*

  Die Funktion liefert eine Beschreibung aller aktuell für die implizite Instanz definierten Properties. Das Format dieser Beschreibung ist in Anhang E.2 erklärt.

- *getPropTitle()* → *String*

  Die Funktion liefert eine Kurzbeschreibung der Instanz zur Verwendung in der Kopfzeile von Property-Editoren.

  Hinweis: Die beiden voran beschriebenen Funktionen werden von den Property-Editoren zum Aufbau des Dialog-Fensters verwendet.

4.4.3 Property–Werte

- *getPropValue(pKey(Symbol))* → *Any*

  Die Funktion liefert den aktuell in der impliziten Instanz gespeicherten Wert für die Property mit dem angegebenen Schlüssel. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, ist der Rückgabewert vom Typ *Void*.

  Hinweis: Die Funktion nutzt die get-Methode der Property (siehe Funktion *setupProperty()*). Besitzt der Typ der impliziten Instanz keine solche Methode, wird der Wert aus der Hash-Tabelle der dynamischen Properties ermittelt (siehe Funktion *getDynamicProps()* in der Schnittstelle *Base*).

- *setPropValue(pKey(Symbol), pValue(Any))* → *Int*

  Die Funktion weist der impliziten Instanz einen neuen Wert für die Property mit dem angegebenen Schlüssel zu.

  Ist die Property mit einem Merkmal in einer Produktdatenbank assoziiert, werden anschließend vom globalen Produktdatenmanager (Kap. 9) Beziehungen zwischen Merkmalen und

Merkmalswerten ausgewertet (infolge dessen sich andere Properties bzw. deren Werte ändern können). Danach wird die Funktion *propsChanged()* (s.u.) zur Durchführung von Spezialbehandlungen aufgerufen. Für den Parameter *pDoChecks* wird dabei True übergeben. Wurde die Wertzuweisung vom Produktdatenmanager oder von *propsChanged()* zurückgewiesen, werden alle Properties auf den zu Beginn der Funktion gespeicherten Zustand zurückgesetzt und abschließend noch einmal die Funktion *propsChanged()* gerufen, wobei für den Parameter *pDoChecks* diesmal False übergeben wird.

Rückgabewert der Funktion ist True, wenn sich die Definition einer oder mehrerer Properties geändert hat bzw. wenn Properties hinzugekommen oder weggefallen sind.

Hinweis: Die Funktion nutzt die set-Methode der Property (siehe Funktion *setupProperty()*) zum eigentlichen Zuweisen des neuen Wertes an die entsprechende Instanzvariable. Besitzt der Typ der impliziten Instanz keine solche Methode, wird der Wert unter dem Schlüssel der Property in die Hash-Tabelle der dynamischen Properties geschrieben (siehe Funktion *getDynamicProps()* in der Schnittstelle *Base*).

- *propsChanged(pPKeys(Symbol[]), pDoChecks(Int))* → *Int*

  Die Funktion führt Spezialbehandlungen und Prüfungen nach der Änderung von Property–Werten durch. Die Properties, deren Werte sich geändert haben, werden durch ihre Schlüssel spezifiziert. Der Parameter *pDoChecks* gibt an, ob Prüfungen durchgeführt werden müssen oder lediglich auf die Änderung der Property–Werte reagiert werden muß, z.B. durch Geometrieanpassungen. Der Rückgabewert ist 1, falls die neuen Property–Werte gültig sind, andernfalls 0.

  Hinweis: Die Funktion wird am Ende der Funktion *setPropValue()* gerufen. In ihr werden in der Regel Anpassungen der Geometrie oder der Materialeigenschaften der impliziten Instanz vorgenommen.

- *changedPropList()* → *Symbol[]*

  Die Funktion liefert die Referenz auf die Liste der Properties, deren Werte sich während der Abarbeitung der Funktion *setPropValue()* geändert haben. Die Properties werden in der Liste anhand ihrer Schlüssel vermerkt.

  Hinweis: Die Funktion wird in der Regel nur von Produktdatenmanagern (Kap. 9) während der Auswertung von Wissen über Produktdatenbeziehungen innerhalb der Funktion *setPropValue()* benutzt.

  Die Liste wird zu Beginn jeder Ausführung von *setPropValue()* geleert.

4.4.4 Aktivierungsstatus

Eine Property kann *aktiv* sein oder nicht. Für eine aktive Property kann deren Wert interaktiv geändert werden. Bei nicht-aktiven Properties werden deren aktuelle Werte nur angezeigt, können aber nicht interaktiv geändert werden. Der initiale Zustand nach der Definition einer Property ist „aktiv“.

- *setPropState(pKey(Symbol), pState(Int)) → Void*

  Die Funktion setzt für die implizite Instanz den Aktivierungsstatus der Property mit dem angegebenen Schlüssel auf den übergebenen Wert. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, hat die Funktion keinen Effekt.

- *getPropState(pKey(Symbol)) → Int*

  Die Funktion liefert 1, wenn die implizite Instanz eine Property mit dem angegebenen Schlüssel besitzt und diese aktiv ist. Die Funktion liefert 0, wenn die implizite Instanz eine Property mit dem angegebenen Schlüssel besitzt und diese nicht aktiv ist. Ist für die implizite Instanz keine Property mit dem angegebenen Schlüssel definiert, ist der Rückgabewert -1.

4.4.5 Informationen über Properties und Property-Werte

- *getPropInfo(pKey(Symbol), pPropValue(Any), pInfoType(Symbol)) → Any*

  Die Funktion liefert für die implizite Instanz die Information des angeforderten Typs zu dem spezifizierten Property-Wert. Der Rückgabewert ist vom Typ *Void*, wenn die Instanz nicht die spezifizierte Property besitzt oder wenn keine Information des angeforderten Typs verfügbar ist.

  Default-Implementierungen dieser Funktion delegieren den Aufruf an die Methode *getPropInfo4Obj()* der für die Instanz zuständigen OiProgInfo–Instanz (insofern vorhanden), siehe Kapitel 8.

  Es sind folgende Standard-Infotypen vordefiniert:

  @Picture
  : Name der Bilddatei, die den Property-Wert veranschaulicht (String)

  @Text
  : textuelle Beschreibung (String, kann Text-Resource sein)

  @HTML
  : URL der HTML-Beschreibung (String)

4.5 Complex

Die Schnittstelle *Complex* beschreibt die notwendige Funktionalität von komplexen Objekten, d.h. von Objekten, die sich aus ein oder mehreren zugreifbaren Teilobjekten (Kindern) zusammensetzen. Dies betrifft im Prinzip alle Typen, deren Instanzen zur Laufzeit zusammengesetzt, erweitert oder zerlegt werden können.

4.5.1 Räumliches Modell

Die Funktionen dieser Gruppe dienen zum einen dem effektiveren Zugriff auf die räumlichen Maße von Objekten, die ansonsten durch die zeitaufwendigere Funktion *getLocalBounds()* der Schnittstelle *Base* ermittelt werden müßten. Zum anderen gestatten sie es, Maße zu verwenden, die von den exakten geometrischen Maßen gemäß *getLocalBounds()* abweichen.

- *getWidth() → Float*

  Die Funktion liefert die Breite der impliziten Instanz.

- *getHeight() → Float*

  Die Funktion liefert die Höhe der impliziten Instanz.

- *getDepth() → Float*

  Die Funktion liefert die Tiefe der impliziten Instanz.

4.5.2 Dynamische Kinderzeugung und -verwaltung

- *checkAdd((pType(Type), pObj(MObject), pPosRot(Any[2]), pParams(Any)) → Float[3]*

  Die Funktion prüft, ob eine Instanz des angegebenen Typs als Kind an die implizite Instanz angefügt werden kann und liefert im positiven Fall eine gültige Position für die Kindinstanz (im lokalen Koordinatensystem der impliziten Instanz). Kann keine Instanz des angegebenen Typs als Kind angefügt werden bzw. kann keine freie gültige Position ermittelt werden, gibt die Funktion einen Wert vom Typ *Void* zurück.

  Falls das Argument *pObj* nicht vom Typ *Void* ist, spezifiziert es eine bereits existierende Instanz, die zur Positionsfindung herangezogen werden soll. Falls das Argument *pPosRot* nicht vom Typ *Void* ist, spezifiziert es eine Vorschlagsposition und -rotation in Bezug auf das lokale Koordinatensystem der impliziten Instanz. Das erste Element des Parametervektors enthält dabei die Vorschlagsposition (Float[3]) und das zweite Element die Vorschlagsrotation bezüglich positiver Y-Achse. Falls das Argument *pParams* nicht vom Typ *Void* ist, enthält es zusätzliche Parameter für die Initialisierungsfunktion des Typs *pType*.

  Zur Prüfung, ob eine Instanz des übergebenen Typs angefügt werden darf, kann es notwendig sein, während der Ausführung der Funktion eine temporäre Instanz des Typs zu erzeugen, um z.B. durch Funktionsaufrufe auf dieser Instanz Aussagen über das zu erzeugende Kind treffen zu können. Die Art und Weise der Erzeugung einer solchen temporären Kindinstanz wird durch den sogenannten *Paste-Modus* gesteuert, der vor Aufruf von *checkAdd()* seitens des Clienten mittels Aufruf der Funktion *setPasteMode()* zugewiesen wird. Repräsentiert die einzufügende Instanz einen Artikel (siehe Schnittstelle *Article*), reicht dabei mitunter eine einfache Instanziierung des übergebenen Typs nicht aus, sondern die temporäre Kindinstanz muß auch die Konfiguration des einzufügenden Artikels annehmen. Dazu wird vom Clienten vor Aufruf von *checkAdd()* der impliziten Instanz mittels Aufruf der Funktion *setTempArticleSpec()* die gewünschte Artikelspezifikation übergeben.

  Sofern der einzufügende Typ Planungskategorien (Anh. H) definiert, können diese bei der Implementierung von *checkAdd()*-Funktionen berücksichtigt werden.

  Hinweis: Diese Funktion wird in der Regel von der Laufzeitumgebung aufgerufen, wenn der Anwender den Befehl zum Einfügen eines Objekts eines gewählten Typs in die Szene bzw. in ein selektiertes Objekt gegeben hat. Liefert die Funktion eine gültige Position, wird im nächsten Schritt von der Laufzeitumgebung eine Instanz des angegebenen Typs erzeugt und an die ermittelte Position gesetzt. Kann das neue Objekt nicht in das selektierte Objekt eingefügt werden, wird versucht, es in dessen Vaterinstanz einzufügen usw.

- *setPasteMode(pMode(Symbol)) → Void*

  Die Funktion setzt den *Paste*-Modus für das Einfügen von temporären Kindinstanzen in die implizite Instanz. Die möglichen Modi sind:

  `@CR` Die Kindinstanz soll als Instanz des an die Funktion *checkAdd()* übergebenen Typs neu erzeugt werden. Dies ist die Standardeinstellung.

  `@PA` Die Kindinstanz soll als Kopie eines bereits existierenden Objekts erzeugt werden, dessen Repräsentation sich im Clipboard der Applikation befindet. In diesem Fall wird die Kindinstanz durch Evaluierung des Clipboards mittels der globalen Funktion *oiApplPaste()* erzeugt.

- *getPasteMode() → Symbol*

  Die Funktion liefert den aktuellen *Paste*-Modus für das Einfügen von temporären Kindinstanzen in die implizite Instanz.

- *setTempArticleSpec(pArticle(Vector[2])) → Void*

  Die Funktion weist der impliziten Instanz die Artikelspezifikation zu, die der temporären Kindinstanz nach deren Erzeugung zuzuweisen ist (siehe Funktion *setXArticleSpec()* der Schnittstelle *Article*, Abschn. 4.6). Der Parameter *pArticle* enthält einen Vektor, dessen erstes Element die Grundartikelnummer angibt, während das zweite den Variantencode des Artikels spezifiziert.

- *getTempArticleSpec() → Vector[2]*

  Die Funktion gibt die Artikelspezifikation für die temporäre Kindinstanz zurück, die mittels der Funktion *setTempArticleSpec()* zugewiesen wurde.

- *setMethod(pMethod(String)) → Void*

  Die Funktion setzt den Methodenaufruf einschließlich der Parameter gemäß Basissyntax (Kap. 3), der nach der Erzeugung und initialen Positionierung einer Kindinstanz nach einer vorangegangenen Ausführung der Funktion *checkAdd()* für diese Kindinstanz ausgeführt werden soll.

- *getMethod() → String*

  Die Funktion liefert den Code gemäß Basissyntax (Kap. 3), der nach der Erzeugung und initialen Positionierung einer Kindinstanz nach einer vorangegangenen Ausführung der Funktion *checkAdd()* für diese Kindinstanz ausgeführt werden soll. Soll keine Methode ausgeführt werden, wird ein leerer String zurückgegeben.

  Hinweis: Der auszuführende Methodenaufruf wird durch die vorangegangene Ausführung der Funktion *checkAdd()* bereitgestellt. Er beinhaltet Aktionen, die über die Positionierung der Kindinstanz hinausgehen, z.B. die Drehung der Kindinstanz um einen geforderten Winkel.

- *clearMethod() → Void*

  Die Funktion setzt einen gegebenenfalls nach der Erzeugung einer Kindinstanz für die Kindinstanz auszuführenden Methodenaufruf zurück. Dabei wird als Methodenaufruf ein leerer String gesetzt.

- *addPart(pType(Type), pParams(Any)) → MObject*

  Die Funktion fügt, falls möglich, eine Instanz des angegebenen Typs als Kind an die implizite Instanz an. Falls das Argument *pParams* nicht vom Typ *Void* ist, enthält es zusätzliche Parameter für die Initialisierungsfunktion des Typs *pType*. Kann keine Instanz des angegebenen Typs als Kind angefügt werden, gibt die Funktion einen Wert vom Typ *Void* zurück.

  Hinweis: Die Funktion nutzt die Funktion *checkAdd()* zur Ermittlung einer gültigen Position und führt bei Rückgabe einer selbigen nach der initialen Positionierung ggf. den durch die Funktion *getMethod()* spezifizierten Code aus.

- *checkElPos(pEl(MObject), pOldPos(Float[3])) → Int*

  Die Funktion prüft die Gültigkeit der aktuellen lokalen Position der übergebenen Kindinstanz. Die Funktion liefert 1, wenn die aktuelle Position erlaubt ist, andernfalls 0.

  Hinweis: Die Funktion dient vorrangig zum Prüfen der neuen Position einer Kindinstanz nach einer Translation oder Rotation derselben. In der Regel erfolgt eine Kollisionsprüfung. Weitere, typabhängige Prüfungen sind möglich, z.B. die Überwachung der Einhaltung eines vorgegebenen Rasters. Gegebenenfalls kann, unter Zuhilfenahme der im Parameter *pOldPos* übergebenen Position vor der Transformation, eine Korrektur der Position stattfinden, z.B. eine Einstellung auf die nächstgelegene Rasterposition.

4.5.3 Kollisionsermittlung

- *disableChildCD() → Void*

  Die Funktion schaltet die Kollisionsermittlung für Kinder der impliziten Instanz, die über *checkChildColl()* erfolgt, aus.

- *enableChildCD() → Void*

  Die Funktion schaltet die Kollisionsermittlung für Kinder der impliziten Instanz, die über *checkChildColl()* erfolgt, (wieder) ein.

- *isEnabledChildCD() → Int*

  Die Funktion liefert 1, falls die Kollisionsermittlung für die implizite Instanz eingeschaltet ist, ansonsten 0.

- *isValidForCollCheck(pObj(MObject)) → Int*

  Die Funktion liefert 1, wenn die angegebene (Kind)Instanz bei der Kollisionsprüfung berücksichtigt werden soll, andernfalls 0.

  Hinweis: Die Funktion ist eine Hakenfunktion, die von der Funktion *checkChildColl()* gerufen wird. Standardimplementierungen dieser Funktion liefern immer 1.

- *checkChildColl(pObj(MObject), pExclObj(MObject[])) → MObject*

Die Funktion prüft, ob eine Kollision der übergebenen (Kind)Instanz mit anderen Objekten vorliegt. Enthält das Argument *pExclObj* eine nicht-leere Menge von Objekten, werden diese von der Kollisionsprüfung ausgeschlossen.

Die Funktion prüft zunächst auf Kollision mit den Kindern der impliziten Instanz. Die Prüfung findet jedoch nur statt, wenn folgende Bedingungen erfüllt sind:

- *isEnabledChildCD()* der impliziten Instanz liefert True
- *isValidForCollCheck()* der impliziten Instanz liefert für die übergebene (Kind)Instanz True
- *isEnabledCD()* der übergebenen (Kind)Instanz liefert True

Von der Kollisionsprüfung werden folgende Kinder ausgeschlossen:

- Kinder, für die die Funktion *isValidForCollCheck()* der impliziten Instanz False liefert
- Kinder, deren Funktion *isEnabledCD()* False liefert
- Kinder, die im Argument *pExclObj* angegeben sind

Liefert *isEnabledCD()* der impliziten Instanz True, wird anschließend die gleichnamige Funktion der Vaterinstanz gerufen (falls diese existiert und deren Typ die Schnittstelle *Complex* implementiert).

Rückgabewert ist das erste aufgefundende Objekt, mit dem die übergebene Instanz kollidiert oder ein Wert vom Typ *Void*, falls keine Kollision entdeckt wurde bzw. die Kollisionserkennung ausgeschaltet ist.

4.6 Article

Die Schnittstelle *Article* umfaßt eine Menge von Funktionen, die aus kaufmännischer Sicht die notwendigen Informationen über ein Planungsobjekt bereitstellen.

4.6.1 Programm-Zugriff

- *getProgram()* → *Symbol*

  Die Funktion liefert die ID des Programms (Anh. I), dem die implizite Instanz angehört.

4.6.2 Bestelllisten-Struktur

- *setOrderID(pID(Symbol))* → *Void*

  Die Funktion weist der impliziten Artikelinstanz eine eindeutige ID zu, die in Bestelllisten-Strukturen zur Zuordnung eines Artikel-Postens der Bestellliste zu der Instanz dient, die den Artikel in der Planung repräsentiert.

  Die Bestell-ID wird der Artikelinstanz unmittelbar nach deren Erzeugung zugewiesen und während der Dauer der Existenz der Artikelinstanz nicht geändert. Ändert sich die Lage

des Artikels in der Planungshierarchie (z.B. bei Gruppierungsaktionen) wird die Bestell-ID bei der dabei stattfindenden Cut/Paste-Operation von der zerstörten Instanz auf die neu erzeugte Klon-Instanz übertragen.

- *getOrderID() → Symbol*

  Die Funktion liefert die eindeutige Bestell-ID der impliziten Artikelinstanz.

4.6.3 Produktdaten

- *getArticleSpec() → String*

  Die Funktion liefert den Namen des Artikels (Grundartikelnummer), dem die implizite Instanz entspricht bzw. einen Wert vom Typ *Void*, falls keine Artikelspezifikation für die implizite Instanz vorliegt.

  Ist das Ergebnis der Funktion ein Wert vom Typ *Void*, wird für die Instanz kein Eintrag in Bestelllisten generiert.

- *getXArticleSpec(pType(Symbol)) → String*

  Die Funktion liefert die Spezifikation des geforderten Typs für den Artikel, dem die implizite Instanz entspricht bzw. einen Wert vom Typ *Void*, falls keine Artikelspezifikation des geforderten Typs für die implizite Instanz vorliegt.

  Folgende Spezifikationstypen sind vordefiniert:

  `@Base`
  : Grundartikelnummer, kennzeichnet das Modell des Artikels ohne Bezug auf eine konkrete Ausprägung/Konfiguration (entspricht dem Rückgabewert von *getArticleSpec()*)

  `@VarCode`
  : Variantencode, beschreibt die konkrete Ausprägung/Konfiguration des Artikels in Bezug auf die Grundartikelnummer.

  `@Final`
  : Endartikelnummer, kennzeichnet das Modell des Artikels und beschreibt seine konkrete Ausprägung/Konfiguration

  Hinweis: Normalerweise setzt sich die Endartikelnummer aus der Grundartikelnummer und dem Variantencode zusammen. Dies ist jedoch abhängig von dem zugrunde liegenden Produktdatensystem. Falls dieses keine solche strikte Definition vorsieht, sind Variantencode und Endartikelnummer gleich.

- *setArticleSpec(pSpec(String)) → Void*

  Die Funktion weist der impliziten Instanz eine neue Grundartikelnummer zu.

  Hinweis: Die Funktion ist nur für Typen relevant, deren Instanzen verschiedene Artikel(nummern) repräsentieren können. Die Zuweisung einer neuen Artikelnummer führt in der Regel zur Änderung bestimmter Merkmale der Instanz und ggf. auch zu einer neuen geometrischen Repräsentation.

- *setXArticleSpec(pType(Symbol), pSpec(String)) → Void*

  Die Funktion weist der impliziten Instanz eine neue Artikelspezifikation des angegebenen Typs zu.

  Die möglichen Spezifikationstypen sind bei der Funktion *getXArticleSpec()* beschrieben. Die Funktion verhält sich bei Übergabe einer Artikelspezifikation des Typs @Base wie die Funktion *setArticleSpec()* oben.

  Hinweis: Die Zuweisung einer neuen Endartikelnummer oder eines neuen Variantencodes (Spezifikationstypen @Final bzw. @VarCode) führt in der Regel zur Änderung bestimmter Merkmale der Instanz und ggf. auch zu einer neuen geometrischen Repräsentation.

- *getArticleParams() → Any*

  Die Funktion liefert die Parameter der impliziten Instanz, die neben dem Typ der Instanz zur Ermittlung der Artikelnummer (siehe Funktion *getArticleSpec()*) verwendet werden sollen. Rückgabewert ist ein Vektor mit den Parameterwerten oder ein String, der bereits die in das jeweilige Speicherformat konvertierten Parameterwerte enthält. Werden für die Ermittlung der Artikelnummer keine Parameter benötigt, liefert die Funktion einen Wert vom Typ *Void*.

- *getArticlePrice(pLanguage(String), ...) → Any[]*

  Die Funktion liefert Preisinformationen für die implizite Instanz in der angegebenen Sprache. Ist ein weiterer optionaler Parameter angegeben, so spezifiziert er die gewünschte Währung. Die Preisinformationen müssen von der Funktion jedoch nicht in dieser Währung geliefert werden (wenn z.B. die zugrunde liegende Produktdatenbank keine Preise in dieser Währung bereitstellen kann). In diesem Fall muß der Client der Funktion anhand von Umtauschsätzen selber eine Umrechnung in die gewünschte Währung vornehmen.

  Rückgabewert ist eine Liste, die die einzelnen Preiskomponenten beinhaltet. Jeder Listeneintrag ist ein Vektor aus drei Elementen:

  1. eine Beschreibung (*String*), die die Art bzw. den Existenzgrund der Preiskomponente spezifiziert, z.B. den Grund eines Aufpreises.
  2. der Verkaufspreis der Preiskomponente (*Float*)
  3. der Einkaufspreis der Preiskomponente (*Float*)

  Eine Ausnahme bildet der erste Eintrag, der anstelle der Preise die verwendete Währung (*String*) enthält. Der letzte Eintrag der Liste spezifiziert den (akkumulierten) Endpreis. Die optionalen Einträge dazwischen spezifizieren die einzelnen Preiskomponenten (Basispreis, Aufpreise, Rabatte usw.). Enthält eine solche Preiskomponente den Bezeichner `"@baseprice"`, so ist sie explizit als Basispreis gekennzeichnet.

  Hinweis: Die explizite Kennzeichnung der Preiskomponente Basispreis kann von der jeweiligen Applikation dazu genutzt werden, den Basispreis bei der Darstellung von Bestelllisten gesondert zu behandeln.

  Die Funktion liefert einen Wert vom Typ *Void*, falls keine Preisinformationen für die implizite Instanz vorliegen.

- *getArticleText(pLanguage(String), pForm(Symbol)) → String[]*

  Die Funktion liefert in der angegebenen Sprache eine textuelle Beschreibung der gewünschten Form für den Artikel, der durch die implizite Instanz repräsentiert wird.

  Mögliche Werte für den Paramter *pForm* sind:

  - `@s` Kurzbeschreibung
  - `@l` Langbeschreibung

  Rückgabewert ist eine Liste von Strings, die die einzelnen Zeilen der Beschreibung enthalten bzw. ein Wert vom Typ *Void*, falls keine Artikelbeschreibung für die implizite Instanz vorliegt.

  Hinweis: Die von dieser Funktion gelieferte Artikelbeschreibung beinhaltet (typischerweise in der Langform) nur Angaben zu den festen Eigenschaften des Artikels. Eine Beschreibung der konkreten, aktuellen Ausprägungen der veränderbaren/konfigurierbaren Eigenschaften des Artikels wird von der Funktion *getArticleFeatures()* geliefert.

- *getArticleFeatures(pLanguage(String)) → Any*

  Die Funktion liefert für den Artikel, der durch die implizite Instanz repräsentiert wird, in der angegebenen Sprache eine Beschreibung der aktuellen Ausprägung der Produktmerkmale, die am Artikel verändert/konfiguriert werden können.

  Rückgabewert ist eine Liste von zweistelligen Vektoren, deren erstes Element (*String*) die Eigenschaft benennt, während das zweite Element den aktuellen Wert (als Zeichenkette) der Eigenschaft beinhaltet. Enthält der Parameter *pLanguage* einen Wert vom Typ *Void*, werden sprachunabhängige Bezeichner für Eigenschaft und Wert geliefert. Die Funktion liefert einen Wert vom Typ *Void*, falls keine Eigenschaftsbeschreibung für die implizite Instanz vorliegt.

  Unmittelbar nacheinander folgende Aufrufe der Funktion mit unterschiedlichen Parametern für die Sprache liefern Listen der gleichen Länge und beinhalten die Eigenschaften in derselben Reihenfolge. Liegt bei einem Sprachparameter vom Typ *Void* für eine Eigenschaft kein sprachunabhängiger Bezeichner für den Wert vor, so ist der entsprechende Eintrag in der Rückgabeliste kein Vektor sondern ein Wert vom Typ *Void*.

  Hinweis: Die von der Funktion bei einem Sprachparameter vom Typ *Void* gelieferten sprachunabhängigen Bezeichner (Kürzel) werden in der Regel von Export-Routinen der Applikation verwendet, um eine komplette Beschreibung eines Artikels zu erzeugen, die an ein externes PPS, z.B. zur Auftragsabwicklung, exportiert werden kann.

4.6.4 Konsistenzprüfung

- *checkConsistency() → Int*

  Die Funktion prüft die Konsistenz und Vollständigkeit des Planungselements. Gegebenenfalls werden Korrekturen oder Ergänzungen vorgenommen bzw. Fehlermeldungen generiert.

  Wurde von der übergeordneten Instanz, die die Konsistenzprüfung der impliziten Instanz veranlaßt hat, ein *Fehler-Log* angelegt, so sind die Fehlermeldungen in dieses Fehler-Log

zu schreiben, anderenfalls können sie mittels *oiOutput()* direkt an den Anwender ausgegeben werden. Das zu verwendende Fehler-Log muß mittels Funktion *getErrorLog()* von der globalen Planungsinstanz (Typ *OiPlanning*, Abschn. 8.1) abgerufen werden. Die für *checkConsistency()* festgelegte Datenstruktur des Fehler-Logs ist eine Hash-Tabelle, in der für jede Artikelinstanz die betreffenden Meldungen unter ihrer Bestell-ID (siehe Funktion *getOrderID()*) als Schlüssel eingetragen sind. Der Wert zu diesem Schlüssel ist eine Liste von dreistelligen Vektoren:

1. die Fehlermeldung (String)
2. der Name des Objekts, das den Fehler gemeldet hat (String)
3. der Name der Methode, in der der Fehler festgestellt wurde (String)

Hinweis: Mit den letzten beiden Angaben können bei Bedarf detailierte Reports für Fehleranalysen generiert werden.

Kapitel 5

Vordefinierte Regelursachen

In diesem Kapitel werden vordefinierte Regelursachen beschrieben. Die Merkmale der vordefinierten Regelursachen sind:

- Sie entsprechen in ihrer Gesamtheit den prinzipiellen Basisinteraktionen wie Selektion, Bewegen, Kopieren, Einfügen, usw.
- Sie werden von der Laufzeitumgebung automatisch aufgerufen, sofern eine entsprechende Aktion aufgetreten ist (impliziter Aufruf).
- Sie können auch explizit aufgerufen werden.

Des weiteren kann es benutzerdefinierte Regelursachen geben. Die Merkmale benutzerdefinierter Regelursachen sind:

- Sie werden stets explizit aufgerufen.
- Die Definition benutzerdefinierter Regelursachen verletzt nicht die Kompatibilität der OFML-Daten.

5.1 Elementregeln

CREATE_ELEMENT

Die Regeln der Ursache *CREATE_ELEMENT* werden für ein Objekt O aufgerufen, *bevor* ein Objekt E vom Typ T_E als Element von O erzeugt wird. Die korrespondierende Interaktion ist die Erzeugung von Objekten im allgemeinen, z.B. durch Einfügen eines Objekts aus dem Clipboard. Der Parameter der Regeln ist der Typ T_E. Regeln dieser Ursachen können verwendet werden, um die Aggregation dynamisch und abhängig vom Zustand des Objekts O zu kontrollieren. Ursachen für das Fehlschlagen derartiger Regeln können sein:

- Instanzen des Typs T_E können überhaupt nicht in O aggregiert werden.

  Beispiel: In einem Korpusschrank können keine Schreibtischlampen verplant werden.

- Instanzen des Typs T_E können nur in O aggregiert werden, sofern bestimmte (geometrische) Regeln eingehalten werden, z.B. eine lineare Abhängigkeit zwischen der Breite von O und der Breite der Instanz von T_E. Falls eine solche Bedingung verletzt ist, schlägt folglich die Regel fehl.

  Beispiel: In einem Korpusschrank mit einer gewissen Breite können im allgemeinen nur die Fachböden mit der entsprechenden Breite verplant werden.

- Instanzen des Typs T_E können prinzipiell in O aggregiert werden; allerdings würde das Einfügen einen Konflikt mit bereits existierenden Kindern hervorrufen.

  Beispiel: In einem Korpusschrank, der bereits an jeder Rasterposition einen Fachboden besitzt, kann kein weiterer Fachboden verplant werden.

In diesen Fällen wird die weitere Bearbeitung der Liste der Regeln abgebrochen und es wird keine Instanz von T_E als Element von O erzeugt.

NEW_ELEMENT

Die Regeln der Ursache *NEW_ELEMENT* werden für ein Objekt O aufgerufen, *bevor* ein Kind E von O in die Liste der Elemente von O aufgenommen wird. Entsprechend Kap. 2 ist ein Element ein spezielles Kind insofern, daß Elemente von außerhalb von O zugreifbar, d.h. erzeugbar oder löschbar, sind. Die korrespondierende Interaktion ist die Erzeugung von Objekten im allgemeinen, z.B. durch Einfügen eines Objekts aus dem Clipboard. Die Regeln der Ursache *NEW_ELEMENT* werden nach dem Aufruf von Regeln der Ursache *CREATE_ELEMENT* aufgerufen. Durch eine *NEW_ELEMENT*-Regel kann die Erzeugung einer Instanz nicht verhindert werden. Dafür bietet die *NEW_ELEMENT*-Ursache gegenüber der Ursache *CREATE_ELEMENT* erweiterte Möglichkeiten zur Ableitung von Funktionalität innerhalb der entsprechenden Regeln. Da hierbei eine tatsächliche Instanz anstelle eines Typs als Parameter übergeben wird, können Abfragen implementiert werden, die über das Vergleichen von Typen hinausgehen, z.B. die Abfrage der Typkompatibilität zu abstrakten Supertypen, die Abfrage geometrischer Parameter sowie sonstige (typabhängige) Abfragen.

Der Regelparameter ist ein bereits existierendes Kind von O, das als Element von O aufgenommen werden soll. Falls eine Regel fehlschlägt, wird E nicht aufgenommen.

Beispiel: Die automatische Erzeugung von Bauteilen wie Montageschienen, die zur Befestigung von Anbauteilen notwendig sind, kann über *NEW_ELEMENT* bzw. *CREATE_ELEMENT* realisiert werden.

REMOVE_ELEMENT

Die Regeln der Ursache REMOVE_ELEMENT werden für ein Objekt O aufgerufen, *bevor* ein Element E gelöscht wird. Die korrespondierende Interaktion ist die Beseitigung von Objekten im allgemeinen, z.B. durch Operationen wie Ausschneiden oder Löschen.

Der Regelparameter ist eine Referenz auf das bereits existierende Element E von O, das gelöscht werden soll. Die Regel kann fehlschlagen, falls andere Elemente in O von E abhängen. Falls eine Regel fehlschlägt, wird E nicht gelöscht.

Beispiel: Ein Bettkasten als Element eines Bettes kann nicht gelöscht werden, solange sich noch Bettrahmen, Matratzen, Kopfteile, Rückenpaneele, etc. darin befinden.

5.2 Auswahlregeln

PICK

Die Regeln der Ursache PICK werden aufgerufen, *nachdem* ein Objekt ausgewählt bzw. selektiert wurde. Die korrespondierende Interaktion ist die Auswahl eines Objekts im allgemeinen, z.B. auf direktmanipulative Weise (2D/3D-Interaktion) oder über eine grafische Benutzungsoberfläche. Der Regelparameter ist vom Typ *Float[3]* und gibt die lokalen Koordinaten an, an denen das Objekt selektiert wurde.

PICK-Regeln können definiert werden, um ein spezielles Feedback zu generieren, z.B. die Änderung der Materialeigenschaften oder der Geometrie. Ein solches Feedback ist unabhängig vom allgemeinen Feedback, das durch die OFML-Laufzeitumgebung bereitgestellt wird. Weiterhin können beliebige Aktionen durch eine PICK-Regel ausgelöst werden, z.B. die Anzeige von Objektmerkmalen innerhalb der grafischen Benutzungsoberfläche oder die Veränderung des globalen Zustandes.

UNPICK

Die Regeln der Ursache UNPICK werden aufgerufen, *nachdem* ein Objekt deselektiert wurde, z.B. durch die Auswahl eines anderen Objekts. Der Regelparameter ist undefiniert.

UNPICK-Regeln sind im allgemeinen eine Umkehrung der entsprechenden PICK-Regeln. Beispielsweise kann das durch die PICK-Regel erzeugte Feedback wieder zurückgesetzt werden.

5.3 Bewegungsregeln

TRANSLATE

Die Regeln der Ursache *TRANSLATE* werden aufgerufen, *nachdem* ein Objekt O verschoben wurde. Die korrespondierende Interaktion ist die translatorische Bewegung von Objekten über direkte

oder indirekte Manipulation. Der Regelparameter ist die lokale Position von O vor der Verschiebung.

Durch die Definition von *TRANSLATE*-Regeln kann die translatorische Bewegung eines Objekts auf beliebige Weise kontrolliert werden, z.B.:

- homogene oder inhomogene Rasterisierung,
- Beschränkung auf einen Bereich,
- Initiierung einer Kollisionserkennung mit entsprechender Korrektur der Position,
- Snapping auf Objekte oder Positionen.

Die verschiedenen Möglichkeiten können innerhalb einer einzigen Regel kombiniert werden, um beispielsweise mehrdimensionale Bewegungen zu ermöglichen. Außerdem kann innerhalb der Regel der Vater aufgerufen und an diesen die Regelfunktionalität delegiert werden.

Weiterhin kann sich O auf beliebige Weise an die neue Position anpassen. Dies kann lokale Merkmale wie Geometrie betreffen oder die Eigenschaften von Kindern, z.B. die Position eines Kindes relativ zu seinem Vater O. Aus dem Vektor, der sich aus neuer und bisheriger Position ergibt, kann eine Richtungsinformation abgeleitet und bei Bedarf angewendet werden.

Beispiel: Die Fachböden eines Korpusschrankes können im Raster von 32 mm, beginnend ab einer Höhe von 80 mm bewegt werden. Die maximale Einbauhöhe ergibt sich aus der inneren Höhe des Korpusschrankes abzüglich 80 mm.

ROTATE

Die Regeln der Ursache *ROTATE* werden aufgerufen, *nachdem* ein Objekt O gedreht wurde. Die korrespondierende Interaktion ist die rotatorische Bewegung von Objekten über direkte oder indirekte Manipulation. Der Regelparameter ist die lokale Orientierung von O entlang der verwendeten Rotationsachse vor der Verdrehung.

Durch die Definition von *ROTATE*-Regeln kann die rotatorische Bewegung eines Objekts auf beliebige Weise kontrolliert werden, z.B.:

- homogene oder inhomogene Rasterisierung,
- Beschränkung auf einen Bereich,
- Initiierung einer Kollisionserkennung und entsprechender Korrektur der Orientierung,
- Snapping auf Objekte oder Positionen.

Die verschiedenen Möglichkeiten können innerhalb einer einzigen Regel kombiniert werden, um beispielsweise eine rasterisierte Rotation innerhalb eines bestimmten Bereiches zu ermöglichen. Außerdem kann innerhalb der Regel der Vater aufgerufen und an diesen die Regelfunktionalität delegiert werden.

In Analogie zur *TRANSLATE*-Regel kann sich ein Objekt an die neue Orientierung auf beliebige Weise anpassen.

Beispiel: Die Tür eines Korpusschrankes läßt sich im Winkel von 0 bis 90 Grad öffnen. Bei Winkeln unter 10 Grad erfolgt automatisch ein Snapping auf 0 Grad. Bei Winkeln über 80 Grad erfolgt automatisch ein Snapping auf 90 Grad. Durch das Snapping-Verhalten kann das Einrasten an den Endpositionen simuliert werden.

SPATIAL_MODELING

Die Regeln der Ursache SPATIAL_MODELING werden aufgerufen, *nachdem* ein Objekt O indirekt bewegt, d.h. verschoben oder gedreht wurde. Eine indirekte Bewegung findet statt, wenn ein Vorfahr (Vater, Großvater, etc.) transliert oder rotiert wurde. Wiederum kann eine Anpassung von O erfolgen. Der Regelparameter ist undefiniert.

Beispiel: Die Griffe der Tür eines Baukastens innerhalb einer Regalplanung werden in Abhängigkeit von der Einbauhöhe des Baukastens plaziert. Bei einer Höhe, die kleiner als 1.40 Meter ist, erfolgt die Plazierung am oberen Ende der Tür. Andernfalls erfolgt die Plazierung am unteren Ende. Durch Anwendung einer *SPATIAL_MODELING*-Regel kann diese Anpassung automatisch realisiert werden.

5.4 Persistenzregeln

Die Persistenzregeln dienen zur Konvertierung der Instanzvariablen von einer Repräsentation, die zur Laufzeit verwendet wird, in eine persistente Repräsentation und umgekehrt. Dazu zählt insbesondere die Konvertierung von Objektreferenzen in speicherbare, restaurierbare Werte wie *String* oder *Int*. Weiterhin können insbesondere die **_EVAL*-Regeln zur Anpassung von gespeicherten Szenen verwendet werden, indem bisher nicht vorhandene Instanzvariablen entsprechend initialisiert werden.

Die Definition von Persistenzregeln ist nur in Ausnahmefällen erforderlich.

Der Regelparameter von Persistenzregeln ist undefiniert.

START_DUMP

Die Regeln der Ursache START_DUMP werden *vor* der Generierung einer persistenten Repräsentation des Objekts O aufgerufen, z.B. im Rahmen einer Szenen-/Objektspeicherung oder einer Clipboard-Operation (z.B. Ausschneiden, Kopieren). Nach Bearbeitung der Regeln müssen die Instanzvariablen in einer speicherbaren Repräsentation vorliegen.

FINISH_DUMP

Die Regeln der Ursache FINISH_DUMP werden *nach* der Generierung einer persistenten Repräsentation des Objekts O und seiner Kinder aufgerufen. Nach Bearbeitung der Regeln müssen die Instanzvariablen wieder in der Repräsentation vorliegen, die für die normale Arbeitsweise erforderlich ist.

START_EVAL

Die Regeln der Ursache START_EVAL werden *vor* der Abarbeitung einer persistenten Repräsentation des Objekts O aufgerufen, z.B. im Rahmen des Ladens einer Szenen-/Objektspeicherung oder einer Clipboard-Operation (z.B. Einfügen). Der Aufruf erfolgt unmittelbar nach der Erzeugung des Objekts O und vor der Zuweisung von Attributen, Kindern, etc.

FINISH_EVAL

Die Regeln der Ursache FINISH_EVAL werden *nach* der Abarbeitung einer persistenten Repräsentation des Objekts O und seiner Kinder aufgerufen. Nach Bearbeitung der Regeln müssen die Instanzvariablen in der Repräsentation vorliegen, die für die normale Arbeitsweise erforderlich ist.

Beispiel: Bei einem bestimmten Rollcontainertyp kann in der neuen Version optional eine Drehstangenverriegelung konfiguriert werden. Folglich definiert dieser Typ in der neuen Version eine zusätzliche Instanzvariable, die über ein Symbol beschreibt, ob die Drehstangenverriegelung gewünscht wird oder nicht. Durch eine *FINISH_EVAL*-Regel kann sichergestellt werden, daß Speicherungen der alten Version diesbezüglich nachinitialisiert werden.

5.5 Sonstige Regeln

SENSOR

Die Regeln der Ursache *SENSOR* werden aufgerufen, falls irgendein Objekt M direkt bewegt wurde. Der Regelparameter ist eine Referenz auf M.

Sensorische Objekte, d.h. Objekte mit mindestens einer *SENSOR*-Regel, können autonom auf Veränderungen der Umgebung reagieren.

Beispiel: Die Tür eines Raumes öffnet automatisch, falls sich ein Objekt im Umkreis von 5 Metern befindet.

TIMER

Die Regeln der Ursache *TIMER* werden aufgerufen, falls das in der jeweiligen Regelsignatur definierte Zeitintervall mindestens einmal abgelaufen ist. Die Anzahl der verstrichenen Intervalle (typischerweise 1 für nicht zu kleine Zeitintervalle) wird als Parameter an die Regel(n) übergeben. Eine Instanz (bzw. ein Typ) mit mindestens einer *TIMER*-Regel ist zeitabhängig. Durch die Erzeugung und Beseitigung zeitabhängiger Kinder kann eine dynamische indirekte Zeitabhängigkeit eines Objekts realisiert werden.

Beispiel: Eine Instanz vom Typ Uhr zeigt die aktuelle Zeit an. Zur Aktualisierung wird eine *TIMER*-Regel verwendet.

INTERACTOR

Die Regeln der Ursache *INTERACTOR* werden beim Versuch einen Interaktor zu selektieren für den Vater des Interaktors aufgerufen. Sie dienen typischerweise dazu, den Interaktor zu aktivieren (Abschn. F.1).

Als Regelparameter wird der ausgewählte Interaktor als Referenz übergeben.

Beispiel: An einer Organisationswand können an verschiedenen Positionen Aufbauten angebaut werden. Werden für diese Positionen Interaktoren definiert, kann der Nutzer interaktiv den gewünschten Anfügepunkt selektieren.

Kapitel 6

Globale Funktionen

6.1 Formatierte Ausgabe

Einige der im folgenden beschriebenen Funktionen verwenden spezielle Zeichenketten zur Steuerung der Formatierung. Die Format-Zeichenkette enthält zwei Arten von Komponenten: normale Zeichen, die unverändert in die Ausgabe übernommen werden, und Formatierungsfolgen, die jeweils die Umwandlung eines der folgenden Argumente steuern. Jede Formatierungsfolge beginnt mit dem Zeichen `%` und endet mit einem Formatierungszeichen. Zwischen dem Zeichen `%` und dem Umwandlungszeichen können, in der angegebenen Reihenfolge folgende optionale Zeichen angegeben werden:

- Steuerzeichen (in beliebiger Reihenfolge), die die Umwandlung modifizieren:

  `-` Das umgewandelte Argument wird nach links ausgerichtet.

  `+` Die Zahl wird immer mit Vorzeichen ausgegeben.

  Leerzeichen Wenn das erste zeichen kein Vorzeichen ist, wird ein Leerzeichen vorangestellt.

  `0` Zahlen werden bis zur Feldbreite mit Nullen aufgefüllt.

  `\#` Erzeugt eine alternative Form der Umwandlung, abhängig vom Formatierungszeichen (s.u.). Bei `o` ist die erste Ziffer eine Null. Bei `x` oder `X` werden `0x` oder `0X` einem von Null verschiedenen Resultat vorangestellt. Bei `e`, `E`, `f`, `g` und `G` enthält die Ausgabe immer einen Dezimalpunkt; bei `g` und `G` werden Nullen am Schluß nicht unterdrückt.

- Eine Zahl, die die minimale Feldbreite festlegt. Es werden mindestens so viele Zeichen ausgegeben wie angegeben, bei Notwendigkeit auch mehr (d.h. es werden nie Zeichen abgeschnitten). Ist das umgewandelte Argument kürzer, wird bis auf die Feldbreite aufgefüllt. Ausrichtung und Füllzeichen richten sich nach den Formatierungs- und Steuerzeichen.

- Ein Punkt, der die Feldbreite von der Genauigkeit trennt.

- Eine Zahl mit folgender Bedeutung: Bei `e`, `E` oder `f` die Anzahl der Ziffern nach dem Dezimalpunkt. Bei `g` oder `G` die Anzahl signifikanter Ziffern. Bei ganzzahligen Werten die minimal auszugebende Anzahl von Ziffern. In den übrigen Fällen gibt die Zahl die maximale Anzahl von Zeichen an, die von einer Zeichenkette ausgegeben werden.

Als Feldbreite oder Genauigkeit kann jeweils `*` angegeben werden, damit wird der Wert durch das nächste bzw. die nächsten zwei Argumente bestimmt, die vom Typ `Int` sein müssen.

Tabelle 6.1 erläutert die Formatierungszeichen. Ein Zeichen, das nach `%` folgt und kein Formatierungszeichen ist, stellt einen Fehler dar.

Zeichen	Argumenttyp	Formatierung
`d`, `i`	`Int`	dezimal mit Vorzeichen
`o`	`Int`	oktal ohne Vorzeichen, führende Null optional
`x`, `X`	`Int`	hexadezimal ohne Vorzeichen, `0x`, `0X` optional, mit `abcdef` bei `x` bzw. `ABCDEF` bei `X`
`u`	`Int`	dezimal ohne Vorzeichen
`c`	`Int`	einzelnes Zeichen (Abschn. 3.2.1)
`s`	`String`	Zeichenkette
`f`	`Float`	dezimal als `[-]`$mmm.ddd$, die Genauigkeit legt die Anzahl der d fest, Voreinstellung: 6, bei 0 entfällt der Dezimalpunkt
`e`, `E`	`Float`	dezimal als `[-]`$mmm.ddd$`e`$\pm xx$ bzw. `[-]`$mmm.ddd$`E`$\pm xx$ die Genauigkeit legt die Anzahl der d fest, Voreinstellung: 6, bei 0 entfällt der Dezimalpunkt
`g`, `G`	`Float`	entspricht `%e`, `%E` falls der Exponent kleiner als -4 oder nicht kleiner als die Genauigkeit ist, sonst `%f`. Null und Dezimalpunkt am Schluß werden nicht ausgegeben.
`%`	-	gibt `%` aus

Tabelle 6.1: Formatierungszeichen

6.2 oiApplPaste()

- *oiApplPaste(pFather(MObject), pName(Symbol))* $\rightarrow$ *Int*

  Die Funktion evaluiert das Clipboard der Applikation und erzeugt ein neues Objekt als Kind von *pFather*. Der lokale Name des neuen Objekts wird durch *pName* spezifiziert. Falls ein Objekt mit dem sich ergebenden globalen Namen schon existiert oder das Applikations–Clipboard leer ist, entsteht ein Laufzeitfehler. Wird für *pName* $NULL$ angegeben, wird automatisch ein gültiger Name gewählt. Rückgabewert der Funktion ist 1, wenn ein Objekt erzeugt werden konnte, andernfalls 0.

  Der Zustand des Clipboards wird nicht verändert.

  Hinweis: Das Applikations–Clipboard wird von der Laufzeitumgebung implementiert und hat keinen Bezug zu dem globalen OFML-Clipboard, das mit Hilfe der Funktion *oiCopy()*, *oiCut()* und *oiPaste()* manipuliert bzw. ausgewertet werden kann.

6.3 oiClone()

- *oiClone(pSrc(MObject), pDest(String)) → MObject*

  Die Funktion erzeugt eine identische Kopie von Objekt *pSrc* unter dem globalen Namen *pDest* und gibt die entsprechende Objektreferenz zurück. Die unmittelbar vorherige Existenz eines Objekts mit dem Namen *pDest* führt zu einem Laufzeitfehler.

  Der Zustand des OFML-Clipboards wird nicht verändert.

6.4 oiCollision()

- *oiCollision(pObject1(MObject), pObject2(MObject)) → Int*

  Die Funktion überprüft die Kollision zwischen zwei Objekten *pObject1* und *pObject2*. Atomares Element der Kollisionsprüfung sind die Polygone der geometrischen Grundprimitiven. Im Falle der parametrischen Primitiven *OiRotation*, *OiSweep* und *OiSurface* ergeben sich diese Polygone aus den Definitionskoordinaten bzw. -flächen. Das heißt, die tatsächliche Abbildung in eine stückweise lineare Approximation bleibt unberücksichtigt.

  Im Falle einer Kollision wird 1 zurückgeliefert, sonst 0.

  Die Funktion liefert stets 1, falls *pObject1* ein Vorfahr (Vater, Großvater, etc.) oder ein Nachfolger (Kind, Enkel, etc.) von *pObject2* ist und umgekehrt.

6.5 oiCopy()

- *oiCopy(pObject(MObject)) → Void*

  Die Funktion schreibt eine adäquate Beschreibung von *pObject* in das globale OFML-Clipboard.

  Der bisherige Zustand des Clipboards geht verloren.

  Da das OFML-Clipboard eine globale Datenstruktur ist, müssen entsprechende Operationen unmittelbar aufeinanderfolgen. Sonst kann die Korrektheit der Operationen nicht garantiert werden.

6.6 oiCut()

- *oiCut(pObject(MObject)) → Void*

  Die Funktion schreibt eine adäquate Beschreibung von *pObject* in das globale OFML-Clipboard und löscht danach Objekt *pObject*.

  Der bisherige Zustand des Clipboards geht verloren.

  Da das OFML-Clipboard eine globale Datenstruktur ist, müssen entsprechende Operationen unmittelbar aufeinanderfolgen. Sonst kann die Korrektheit der Operationen nicht garantiert werden.

6.7 oiDialog()

- *oiDialog(pDialog(Symbol), pIcon(Symbol), pMessage(String)) → Symbol*

  Diese Funktion veranlaßt die Reaktion des Anwenders auf einen modalen Dialog, der durch die OFML-Laufzeitumgebung generiert wird.

  Der Parameter *pDialog* spezifiziert den Dialog durch eines der folgenden Symbole. Die möglichen Rückgabewerte sind dabei in Klammern aufgeführt.

  - *@OK* - Bestätigung (*@OK*).
  - *@OK_CAN* - Bestätigung oder Abbruch (*@OK*, *@CANCEL*).
  - *@ABT_IGN* - Abbruch oder Ignorieren (*@ABORT*, *@IGNORE*).
  - *@YES_NO_CAN* - Ja oder Nein oder Abbruch (*@YES*, *@NO*, *@CANCEL*).
  - *@YES_NO* - Ja oder Nein (*@YES*, *@NO*).

  Wird kein gültiger Wert für *pDialog* übergeben, wird kein Dialog gestartet und der Rückgabewert ist *@INVALID_DIALOG*.

  Über den Parameter *pIcon* wird zusätzlich die visuelle Darstellung des Dialoges spezifiziert. Diese kann durch ein entsprechendes Icon erfolgen und ist unabhängig vom Wert von *pDialog* stets bindend. Der Wertebereich von *pIcon* ist wie folgt festgelegt:

  - *@NONE* - Keine Ausgabe eines speziellen Zeichens.
  - *@STOP* - Ausgabe eines Stopp-Zeichens.
  - *@QUESTION* - Ausgabe eines Fragezeichens.
  - *@WARNING* - Ausgabe eines Ausrufezeichens.
  - *@INFO* - Ausgabe eines Informationszeichens (ein kleines *i* innerhalb eines Kreises).

  Wird kein gültiger Wert für *pIcon* übergeben, wird kein Dialog gestartet und der Rückgabewert ist *@INVALID_ICON*.

  Der Parameter *pMessage* gibt die auszugebende Botschaft an. Sofern das erste Zeichen des Strings *pMessage* ein `@` ist, wird der String als Referenz betrachtet, die durch einen Zugriff auf eine externe Datenbasis (Anh. D) aufgelöst wird.

  pMessage muß entweder gültiger String im Sinne der Basissyntax (Kap. 3) oder ein Vektor sein. Im ersten Fall sind keine Umlaute erlaubt. Die Angabe eines Vektors dient zur formatierten Ausgabe. Dabei ist das erste Element die Format-Zeichenkette (Abschn. 6.1), die restlichen Elemente sind die zu formatierenden Argumente. Beginnt das erste Element des Vektors mit `@`, wird die Format-Zeichenkette, wie oben angegeben, aus der externe Datenbasis gelesen. Wird kein gültiger Wert für *pMessage* übergeben, wird im Dialog eine leere Zeichenkette ausgegeben.

  Rückgabewert ist ein Symbol, das die gewählte Antwort im Sinne der oben genannten Alternativen beschreibt.

6.8 oiDump2String()

- *oiDump2String(pObj(MObject)) → String*

  Die Funktion liefert die (implementationsabhängige) Dump-Repräsentation der übergebenen Instanz.

  Zusammen mit der Funktion *oiReplace()* können damit Objektzustände gespeichert und wiederhergestellt werden, was z.B. in Problemfällen zur Realisierung von Undo-fähigen Operationen genutzt werden kann.

6.9 oiExists()

- *oiExists(pName(String)) → Int*

  Die Funktion prüft die Existenz des Objekts, dessen absoluter Name als String im Parameter *pName* übergeben wird. Falls das Objekt existiert, ist der Rückgabewert 1, sonst 0. Die Existenzprüfung ist gegebenenfalls notwendig, da ein Zugriff auf ein nicht existierendes Objekt einen Laufzeitfehler verursacht.

6.10 oiGetDistance()

- *oiGetDistance(pPosition(Float[3]), pDirection(Float[3])) → Float*

  Die Funktion ermittelt den ersten Schnittpunkt eines Strahls, der seinen Ursprung im Weltkoordinatenpunkt *pPosition* hat und entlang dem normierten Vektor *pDirection* verläuft, mit den Objekten der Szene. Der Rückgabewert ist die Entfernung entlang des Strahls bis zum ersten Schnittpunkt oder -1, falls kein Schnittpunkt gefunden wurde.

6.11 oiGetNearestObject()

- *oiGetNearestObject(pPosition(Float[3]), pDirection(Float[3])) → MObject*

  Die Funktion ermittelt das erste getroffene Objekt bei der Verfolgung eines Strahls, der seinen Ursprung im Weltkoordinatenpunkt *pPosition* hat und entlang dem normierten Vektor *pDirection* verläuft. Der Rückgabewert ist eine Referenz auf das zuerst getroffene Objekt oder $NULL$, falls kein Objekt getroffen wurde.

6.12 oiGetRoots()

- *oiGetRoots() → MObject[]*

  Die Funktion ermittelt die in der Szene vorhandenen Wurzelobjekte.

6.13 oiGetStringResource()

- *oiGetStringResource(pStr(String), pLanguage(String), ...) → String*

  Die Funktion liefert den in einer externen Ressourcen-Datei abgelegten Text zu der übergebenen Text-Ressource in der angegebenen Sprache bzw. die Text-Ressource, falls kein Text zu der Ressource gefunden werden konnte oder ein ungültiger Wert für die Sprache übergeben wurde.

  Ist ein weiterer optionaler Parameter angegeben, so spezifiziert er eine Instanz, in deren Namensraum der Text gesucht wird, falls die Text-Ressource nicht vollqualifiziert ist (siehe Anh. D).

6.14 oiLink()

- *oiLink(pURL(String)) → Void*

  Die Funktion lädt die durch den String *pURL* spezifizierte Datei. Im Ergebnis kann die aktuelle Szene durch eine neue Szene bzw. ein anderes Dokument ersetzt werden.

6.15 oiOutput()

- *oiOutput(pLevel(Symbol), pMessage(String)) → Void*

  Diese Funktion veranlaßt die Ausgabe einer textuellen Botschaft durch die OFML-Laufzeitumgebung. Die Ausgabe sollte durch einen modalen Dialog realisiert sein. Das Symbol *pLevel* beschreibt die Kategorie der Ausgabe wie folgt:

  – *@MESSAGE* - Ausgabe einer Mitteilung.
  – *@WARNING* - Ausgabe einer Warnung.
  – *@ERROR* - Ausgabe einer Fehlermeldung.
  – *@FATAL* - Ausgabe einer Fehlermeldung. Nach Quittieren des modalen Dialogs muß die Laufzeitumgebung terminieren.

  Sofern das erste Zeichen des Strings *pMessage* ein `@` ist, wird der String als Referenz betrachtet, die durch einen Zugriff auf eine externe Datenbasis (Anh. D) aufgelöst wird.

  pMessage muß entweder gültiger String im Sinne der Basissyntax (Kap. 3) oder ein Vektor sein. Im ersten Fall sind keine Umlaute erlaubt. Die Angabe eines Vektors dient zur formatierten Ausgabe. Dabei ist das erste Element die Format-Zeichenkette (Abschn. 6.1), die restlichen Elemente sind die zu formatierenden Argumente. Beginnt das erste Element des Vektors mit `@`, wird die Format-Zeichenkette, wie oben angegeben, aus der externe Datenbasis gelesen.

  Wird als Message `"::ofml::app::@none"` übergeben, so erfolgt durch die Applikation keine Ausgabe einer Meldung.

Hinweis: Diese kann genutzt werden, um z.B. via `oiOutput(@ERROR, "::ofml::app::@none")` der Applikation einen „Fehlerzustand“ anzuzeigen, für den seitens OFML bereits ein Dialog (Funktion *oiDialog()*) durchgeführt wurde (z.B. ein Abbrechen-Dialog während *checkAdd()*, siehe Schnittstelle *Complex*), und eine weitere Meldung nicht erwünscht ist.

6.16 oiPaste()

- *oiPaste(pFather(MObject), pName(Symbol)) → MObject*

  Die Funktion evaluiert das globale OFML-Clipboard und erzeugt ein neues Objekt als Kind von *pFather*. Der lokale Name des neuen Objekts wird durch *pName* spezifiziert. Falls ein Objekt mit dem sich ergebenden globalen Namen schon existiert, entsteht ein Laufzeitfehler. Wird für *pName NULL* angegeben, wird automatisch ein gültiger Name gewählt. Rückgabewert der Funktion ist eine Referenz auf das erzeugte Objekt.

  Der Zustand des Clipboards wird nicht verändert.

  Da das OFML-Clipboard eine globale Datenstruktur ist, müssen entsprechende Operationen unmittelbar aufeinander folgen. Sonst kann die Korrektheit der Operationen nicht garantiert werden.

6.17 oiReplace()

- *oiReplace(pObj(MObject), pDump(String)) → Void*

  Die Funktion ersetzt die übergebene Instanz durch ein Objekt, dessen Dump-Repräsentation im übergebenen Puffer enthalten ist.

  Eine (implementationsabhängige) Dump-Repräsentation kann mit der Funktion *oiDump2String()* erstellt werden.

6.18 oiSetCheckString()

- *oiSetCheckString(pString(String) → Void*

  Die Funktion setzt einen String, der von der jeweiligen OFML-Laufzeitumgebung zu verifizieren ist. Dieser String, der üblicherweise in persistenten OFML-Szenenrepräsentationen gesetzt wird, kann zur Prüfung der Konsistenz oder Gültigkeit einer Szenenrepräsentation verwendet werden. Ein in diesem Sinne fehlerhafter String muß den Abbruch des Einlesens der persistenten Szenenrepräsentation zur Folge haben.

6.19 oiTable()

- *oiTable(pRequest(Symbol), pArgs(List)) → List*

Die Funktion *oiTable* realisiert den lesenden Zugriff auf Daten aus einer externen relationalen Datenbank (Anh. D).

Über den Parameter *pRequest* wird die gewünschte Tabellenoperation spezifiziert und über *pArgs* die entsprechenden Argumente. Folgende Aufzählung gibt die möglichen Operationen und zugehörigen Argumente an:

```
@openTbl     List of TableEntry
@closeTbl    List of TableID
@readTE      List of TableEntry
```

Ein *TableEntry* wird als Vektor `[tableID, attributeList]` übergeben, wobei *tableID* als String und *attributeList* als Liste von *TableAttributen* angegeben werden.

Ein *TableAttribute* wird als Vektor `[name, isPrimKey, isKey, type, value, format]` übergeben, wobei *name* als String, *isPrimKey* und *isKey* als (boolesche) Int, *type* als Symbol, *value* als Objekt gemäß *type* und *format* als String angegeben werden.

Folgende Attributtypen und zugehörigen Formatstrings sind definiert:

```
Int:    type = @i, format = maximale Anzahl der Stellen
Float:  type = @f, format = Anzahl Stellen gesamt.Anzahl Stellen nach Komma
String: type = @s, format = maximale Laenge
```

Ein TableID–String setzt sich aus drei durch Leerzeichen getrennte Bestandteile zusammen:

- dem Bezeichner für den Typ der Datenbank, der in OFML stets `"FTXT"` (Textdatei mit fester Feldlänge) ist,
- dem Lokalisationspfad für die zu verwendende Datenbank und
- dem eigentlichen Namen der interessierenden Tabelle.

Vor dem ersten Zugriff muß eine Tabelle geöffnet werden. Die Operation `@openTbl` öffnet eine Tabelle zum Lesen, wobei deren Struktur über die Liste der Attribute definiert wird. Die Attribute *isPrimKey* und *isKey* von *TableAttribute* werden nur während der Tabellendefinition in `@openTbl` ausgewertet.

Die Operation `@closeTbl` verlangt als Parameter eine Liste von TableIDs der zu schließenden Tabellen. Alle zur Verwaltung dieser Tabellen verwendeten Systemressourcen werden freigegeben, ein Zugriff ist danach nicht mehr möglich.

Die Operation `@readTbl` dient dem Lesen von Tabellenzeilen. Welche Zeilen gelesen werden sollen, wird durch die in der Liste übergebenen TableEntries spezifiziert. `@readTbl` ist dabei die einzige Operation, bei der die übergebenen TableEntries keine vollständige Zeilenbeschreibung gemäß Tabellendefinition enthalten müssen. Vielmehr müssen nur diejenigen TableAttributes angegeben werden, die als Schlüssel für den Zugriff dienen sollen. Für einen übergebenen TableEntry werden alle Zeilen geliefert, deren Werte in den angegebenen Spalten mit den in den übergebenen TableAttributen angegebenen Werten übereinstimmen. Die Operation `@readTbl` stellt somit bereits eine einfache, Tabellen-lokale Suchfunktion bereit. Nur wenn der TableEntry ein bei der Tabellendefinition als Primärschlüssel markiertes TableAttribute enthält, ist ein eindeutiges Ergebnis zu erwarten. Werden an `@readTbl` mehrere TableEntry–Objekte übergeben, wird für jedes Objekt die entsprechende Abfrage ausgeführt und die Ergebnisse in der zurückgegebenen Liste verknüpft.

Kapitel 7

Geometrische Typen

Dieses Kapitel beschreibt die Hierarchie der geometrisch orientierten Typen. Eine geometrische Instanz ist direkt durch ihre Geometrie und gegebenenfalls durch ihre Kinder sichtbar. Die Instanzen geometrischer Typen befinden sich im allgmeinen auf der niedrigsten Stufe in hierarchischen Produktmodellen.

7.1 OiGeometry

Beschreibung

- Der abstrakte Typ *OiGeometry* ist der Basistyp für die im folgenden beschriebenen geometrisch-orientierten Typen. *OiGeometry* darf nicht direkt instantiiert werden. Die Ableitung anwendungsspezifischer Typen von *OiGeometry* ist erlaubt. Hierbei erfolgt eine Implementierung eines anwendungsspezifischen, abgeleiteten Typs durch Parametrisierung und Aggregation einer oder mehrerer *OiGeometry*-kompatibler Instanzen.

  Die Instanzen des Typs *OiGeometry* können zur Implementierung der Geometrie ein Kind mit dem lokalen Namen *geo* besitzen. Dieser Name darf nicht anderweitig vergeben werden. Außerdem ist bei Iterationen auf der Liste der Kinder die potentielle Existenz von *geo* zu beachten.

- **Schnittstelle(n):** Base, Material

Initialisierung

- *OiGeometry(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine indirekte Instanz vom Typ *OiGeometry*. Initial ist die Selektierbarkeit ausgeschaltet. Die initiale Materialkategorie ist *@ANY*. Im Normalfall muß diese über *setMatCat()* entsprechend geändert werden. Die initiale Ausrichtung ist nicht einheitlich definiert und richtet sich nach der entsprechenden Primitive.

Methoden

- *setMatCat(pCat(Symbol)) → Void*

  Die Funktion überschreibt die initiale Materialkategorie *@ANY* bzw. eine bereits gesetzte Kategorie durch den Wert von *pCat*.

- *setAlignment(pAlignment(Symbol[3])) → Void*

  Die Funktion erlaubt die Ausrichtung der Geometrie bezüglich der lokalen Achsen. Dabei werden folgende Symbole für Element von *pAlignment* unterstützt (jeweils bezüglich einer der drei Achsen):

  – *@C* – Der Ursprung des Objekts befindet sich in der Mitte des lokalen Begrenzungsvolumens.
  – *@I* – Der Ursprung des Objekts befindet sich im Minimum des lokalen Begrenzungsvolumens.
  – *@A* – Der Ursprung des Objekts befindet sich im Maximum des lokalen Begrenzungsvolumens.

  Eine nachfolgende Änderung der Geometrie führt nicht zur Anpassung entsprechend der gesetzten Ausrichtung. Eventuell vorhandene Kinder können beim Setzen der Ausrichtung zu unerwarteten Ergebnissen führen.

7.2 OiBlock

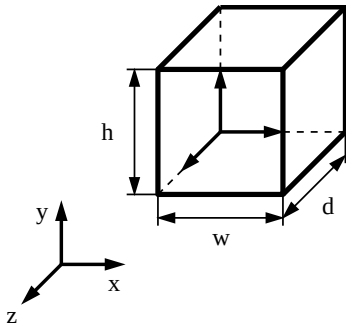

Abbildung 7.1: Der geometrische Typ *OiBlock*

Beschreibung

- *OiBlock* repräsentiert einen orthogonalen Quader, der im Ursprung des lokalen Koordinatensystems beginnt und sich entlang der positiven Achsen des lokalen Koordinatensystems entsprechend ausdehnt. Die Abmessungen des Quaders können nach der Erzeugung geändert werden.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiBlock(pFather(MObject), pName(Symbol), pDimensions(Float[3]))*

  Die Funktion initialisiert eine Instanz vom Typ *OiBlock*. Die initialen Abmessungen des Quaders werden durch einen Vektor von drei positiven Zahlen angegeben.

Methoden

- *setDimensions(pDimensions(Float[3])) → Void*

  Die Funktion setzt die Abmessungen des Quaders. *pDimensions* muß ein Vektor von drei positiven Zahlen sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getDimensions() → Float[3]*

  Die Funktion liefert die aktuellen Abmessungen des Quaders.

7.3 OiCylinder

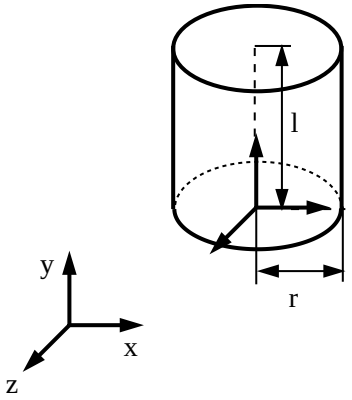

Abbildung 7.2: Der geometrische Typ *OiCylinder*

Beschreibung

- *OiCylinder* repräsentiert einen abgeschlossenen homogenen Zylinder, der im Ursprung des lokalen Koordinatensystems beginnt und sich entlang der positiven y-Achse des lokalen Koordinatensystems zentriert ausdehnt. Die Abmessungen des Zylinders können nach der Erzeugung geändert werden.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiCylinder(pFather(MObject), pName(Symbol), pLength(Float), pRadius(Float))*

  Die Funktion initialisiert eine Instanz vom Typ *OiCylinder*. Die initialen Abmessungen des Zylinders werden durch die Parameter Länge und Radius angegeben. Dabei sind nur positive Zahlen erlaubt.

Methoden

- *setLength(pLength(Float))* $\rightarrow$ *Void*

  Die Funktion setzt die Länge des Zylinders. *pLength* muß eine positive Zahl sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getLength()* $\rightarrow$ *Float*

  Die Funktion liefert die aktuelle Länge des Zylinders.

- *setRadius(pRadius(Float))* $\rightarrow$ *Void*

  Die Funktion setzt den Radius des Zylinders. *pRadius* muß eine positive Zahl sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getRadius()* $\rightarrow$ *Float*

  Die Funktion liefert den aktuellen Radius des Zylinders.

7.4 OiEllipsoid

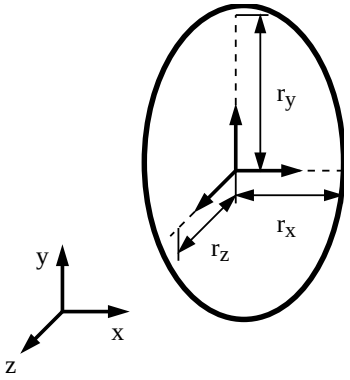

Abbildung 7.3: Der geometrische Typ *OiElliposid*

Beschreibung

- *OiEllipsoid* repräsentiert einen homogenen Ellipsoid, dessen Zentrum sich im Ursprung des lokalen Koordinatensystems befindet und der sich nach allen sechs Seiten des lokalen Koordinatensystems entsprechend ausdehnt. Die Abmessungen des Ellipsoids können nach der Erzeugung geändert werden.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiEllipsoid(pFather(MObject), pName(Symbol), pDimensions(Float[3]))*

  Die Funktion initialisiert eine Instanz vom Typ *OiEllipsoid*. Die initialen Abmessungen des Ellipsoids werden durch einen Vektor von drei positiven Zahlen angegeben.

Methoden

- *setDimensions(pDimensions(Float[3])) → Void*

  Die Funktion setzt die Abmessungen des Ellipsoids. *pDimensions* muß ein Vektor von drei positiven Zahlen sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getDimensions() → Float[3]*

  Die Funktion liefert die aktuellen Abmessungen des Ellipsoids.

7.5 OiFrame

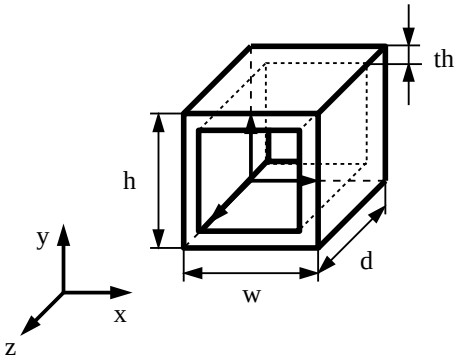

Abbildung 7.4: Der geometrische Typ *OiFrame*

Beschreibung

- *OiFrame* repräsentiert einen Rahmen, der im Ursprung des lokalen Koordinatensystems beginnt und sich entlang der positiven Achsen des lokalen Koordinatensystems entsprechend ausdehnt. Dabei wird aus dem Körper in der lokalen x-y-Ebene ein orthogonales Volumen subtrahiert. Die Dicke des Rahmens in x- und y-Richtung ist gleich. Für die Abmessungen in x- und y-Richtung w, h und die x/y-Dicke th muß stets gelten: $w, h > 2*th$. Die Dimensionen des Rahmens können nach der Erzeugung geändert werden.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiFrame(pFather(MObject), pName(Symbol), pDimensions(Float[3]), pThickness(Float))*

  Die Funktion initialisiert eine Instanz vom Typ *OiFrame*. Die initialen äußeren Abmessungen des Rahmens werden durch einen Vektor von drei positiven Zahlen angegeben. Die initiale Dicke des Rahmens in der lokalen x- und y-Richtung wird durch eine positive Zahl angegeben.

Methoden

- *setDimensions(pDimensions(Float[3])) → Void*

  Die Funktion setzt die äußeren Abmessungen des Rahmens. *pDimensions* muß ein Vektor von drei positiven Zahlen sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getDimensions() → Float[3]*

  Die Funktion liefert die aktuellen äußeren Abmessungen des Rahmens.

- *setThickness(pThickness(Float)) → Void*

  Die Funktion setzt die Rahmendicke in der lokalen x- und y-Richtung. *pThickness* muß eine positive Zahl sein.

- *getThickness() → Float*

  Die Funktion liefert die aktuelle Rahmendicke in der lokalen x- und y-Richtung.

7.6 OiHole

Beschreibung

- *OiHole* realisiert kreisförmige oder rechteckförmige Durchbrüche in kreisförmigen oder rechteckförmigen Bereichen. Damit können boolesche Operationen, insbesondere die Subtraktion in Spezialfällen simuliert werden. Es erfolgt jedoch keine tatsächliche Subtraktion im Sinne einer booleschen Operation. Der eigentliche Sinn von *OiHole* besteht in der Generierung der

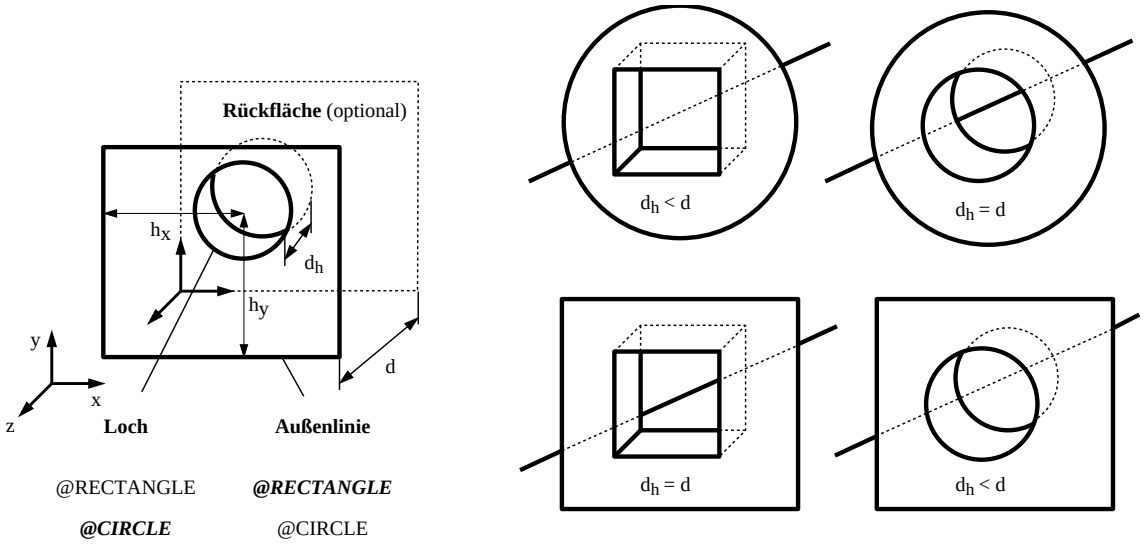

Abbildung 7.5: Der geometrische Typ *OiHole*

Flächen für die Kombinationen kreisförmige Außenlinie – rechteckförmiges Loch und rechteckförmige Außenlinie – kreisförmiges Loch. *OiHole* realisiert keine Außenflächen entlang der Außenlinie in der lokalen z-Richtung. *OiHole*-Instanzen beginnen im Ursprung des lokalen Koordinatensystems und breiten sich entsprechend der äußeren Dimensionen entlang der positiven x-, y- und z-Achse aus.

- **Supertyp:** *OiGeometry*

Initialisierung

- *OiHole(pFather(MObject), pName(Symbol), pOMode(Symbol), pODim(Float[3]), pBack(Int), pHMode(Symbol), pHDim(Float[3]), pHOffset(Float[2])*

  Die Funktion initialisiert eine Instanz vom Typ *OiHole*. Dazu sind folgende spezifische Parameter anzugeben:

  – Der Parameter *pOMode* gibt den Modus der Außenlinie an. Erlaubte Ausprägungen für *pOMode* sind:

    * *@RECTANGLE* – Die Außenlinie entspricht einem Rechteck.
    * *@CIRCLE* – Die Außenlinie entspricht einem Kreis.

  – Der Parameter *pODim* bestimmt die äußeren Dimensionen des Körpers bestehend aus der Breite *w*, der Höhe *h* und der Tiefe *d*. Alle Dimensionen müssen positive Zahlen sein. Im Falle einer kreisförmigen Außenlinie legt die Breite gleichzeitig die Höhe fest.

  – Der Parameter *pBack* gibt an, ob die äußere Rückfläche erzeugt wird (*pBack == 1*) oder nicht (*pBack == 0*).

– Der Parameter *pHMode* gibt den Modus des Loches an. Erlaubte Ausprägungen für *pHMode* sind:
  * *@RECTANGLE* – Es wird ein rechteckförmiges Loch generiert.
  * *@CIRCLE* – Es wird ein kreisförmiges Loch generiert.
– Der Parameter *pHDim* bestimmt die Dimensionen des Loches bestehend aus der Breite w_h, der Höhe h_h und der Tiefe d_h. Alle Dimensionen müssen positive Zahlen sein. Im Falle eines kreisförmigen Loches legt die Breite gleichzeitig die Höhe fest. Die Lochbreite w_h muß kleiner als die Gesamtbreite w sein. Die Lochhöhe h_h muß kleiner als die Gesamthöhe h sein. Die Lochtiefe d_h darf nicht größer als die Gesamttiefe d sein. Falls sie kleiner als diese ist, wird automatisch eine Lochrückfläche generiert.
– Der Parameter *pHOffset* definiert den Offset vom Mittelpunkt des Loches zum lokalen Ursprung der Primitive. Das Loch darf nicht über den Bereich des äußeren Volumens hinausragen.

7.7 OiHPolygon

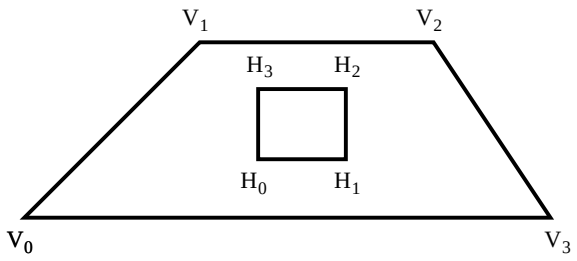

Abbildung 7.6: Der geometrische Typ *OiHPolygon*

Beschreibung

- *OiHPolygon* repräsentiert ein einseitiges, einfaches, planares und konvexes Polygon, aus dem eine Menge einfacher, planarer und konvexer Polygone ausgeschnitten werden können.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiHPolygon(pFather(MObject), pName(Symbol), pMosaic(Int), pOutline(Float[3][]), pHoles(Float[3][][]))*

  Die Funktion initialisiert eine Instanz vom Typ *OiHPolygon*. Der Parameter *pMosaic* steuert die Tesselierung des resultierenden Polygonnetzes. Bekommt *pMosaic* den Wert 0 erfolgt eine Triangulierung. Andernfalls wird die Anzahl der internen Polygone minimiert. *pOutline* beschreibt das äußere Polygon in Uhrzeigerrichtung. *pHoles* ist ein optional leerer Vektor von

Polygonen, welche jeweils einen Ausschnitt beschreiben. Diese Polygone müssen entgegengesetzt der Uhrzeigerrichtung definiert werden.

7.8 OiImport

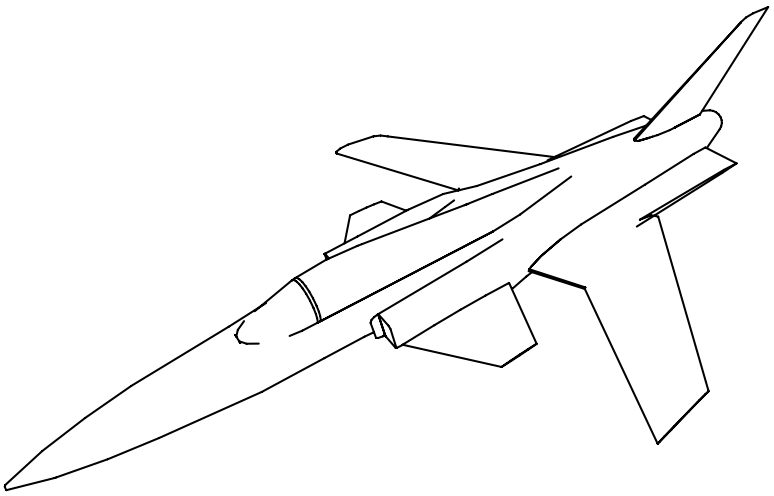

Abbildung 7.7: Der geometrische Typ *OiImport*

Beschreibung

- *OiImport* importiert eine externe Datei in einem geometrischen Format. Sofern diese keine Materialien beinhaltet, kann ein Material über das Interface *Material* gesetzt werden.

  OiImport unterstützt optional zu der tatsächlichen Geometrie genau eine stark auflösungsreduzierte Geometrie. Falls diese vorhanden ist, kann sie bei der geschwindigkeitsoptimierten Darstellung verwendet werden. Die Verwendung ist allerdings abhängig von der jeweiligen Darstellungssoftware.

- **Supertyp:** *OiGeometry*

Initialisierung

- *OiImport(pFather(MObject), pName(Symbol), pMode(Symbol), pGeometry(String))*

Die Funktion initialisiert eine Instanz vom Typ *OiImport*. *pGeometry* beschreibt den Namen einer Geometrie-Datei in Form eines einfachen Strings ohne Pfad- oder Extension-Angaben, z.B. "wheel". Der Dateityp wird durch den Parameter *pMode* festgelegt. Erlaubte Ausprägungen für *pMode* sind:

- *@OFF* – Die Geometrie befindet sich im Object File Format.
- *@G3DS* – Die Geometrie befindet sich im 3D Studio-Format.

Die optionale auflösungsreduzierte Geometrie-Datei besitzt zusätzlich einen Unterstrich zu Beginn des Namens, z.B. "_wheel".

Der Datensatz wird entsprechend den Definitionen für externe Daten (Kap. D) geladen und muß vollständig qualifiziert sein.

Methoden

- *setScale(pFactor(Float[3])) → Void*

  Die Funktion erlaubt die Skalierung von *OiImport*-Objekten. Die Elemente des Vektors *pFactor* müssen relle, positive Zahlen sein. Die initiale Skalierung ist 1.0 in allen drei Dimensionen.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getScale() → Float[3]*

  Die Funktion liefert die aktuelle Skalierung der impliziten Instanz.

7.9 OiPolygon

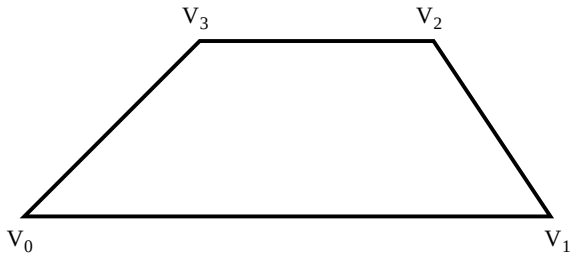

Abbildung 7.8: Der geometrische Typ *OiPolygon*

Beschreibung

- *OiPolygon* repräsentiert ein einseitiges, einfaches, planares und konvexes Polygon. Diese Primitive ist nur in Ausnahmefällen zu verwenden, da eine Menge von *OiPolygonen* im Vergleich zu anderen Polygonmengen (z.B. auf Basis von *OiImport*) erheblich ineffizient ist. Außerdem beschreibt ein singuläres *OiPolygon* keinen Körper, was der generellen Intention von OFML widerspricht.

- **Supertyp:** *OiGeometry*

Initialisierung

- *OiPolygon(pFather(MObject), pName(Symbol), pPoints(Float[3][]))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPolygon*. Der Parameter *pPoints* definiert ein einseitiges, einfaches, planares und konvexes Polygon. Dabei werden der letzte und der erste Punkt automatisch verbunden. Die Sichtbarkeit ergibt sich gemäß der Rechte-Hand-Regel. Wenn die Krümmung der rechten Hand der Vertex-Liste folgt, gibt der Daumen der rechten Hand die sichtbare Seite an.

7.10 OiRotation

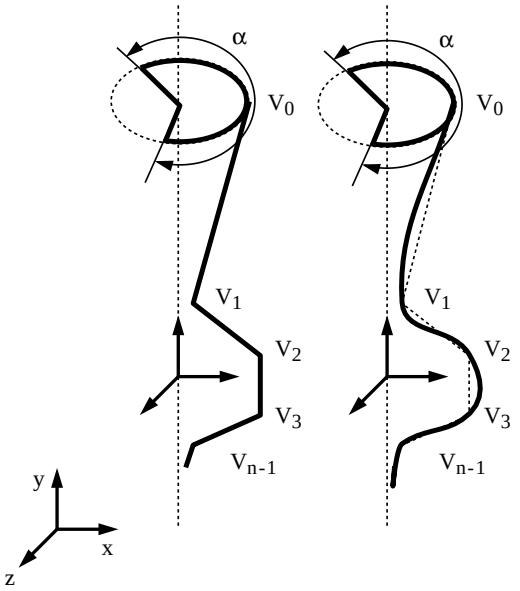

Abbildung 7.9: Der geometrische Typ *OiRotation*

Beschreibung

- *OiRotation* beschreibt einen Solid-Körper, der durch die planare Rotation einer dreidimensionalen planaren Kurve um eine Achse definiert wird. Die Kurve muß nach der Rechte-Hand-Regel wie folgt definiert sein: Wenn der Daumen der rechten Hand in Richtung der Drehung zeigt, geben die restlichen Finger der Hand die Orientierung an. Andernfalls muß eine Invertierung erfolgen.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiRotation(pFather(MObject), pName(Symbol), pMode(Symbol), pAxis(Float[3]), pPoints(Float[3][]), pArc(Float), pUWMode(Symbol[2]), pCMode(Symbol[2]), pFlip(Int))*

  Die Funktion initialisiert eine Instanz vom Typ *OiRotation*. Dazu sind folgende spezifische Parameter anzugeben:

  - *pMode* spezifiziert, ob der Körper entlang der Definitionskurve geglättet sein soll (*@SMOOTH*) oder nicht (*@LINEAR*).
  - *pAxis* definiert die Rotationsachse in Bezug auf das lokale Koordinatensystem.
  - *pPoints* beschreibt die Definitionskurve. Hierbei sind Punkte auf der Rotationsachse nicht erlaubt.
  - *pArc* setzt den Winkel der Drehung der Definitionskurve. *pArc* muß positiv und kleiner oder gleich 2π sein.
  - *pUWMode* definiert die Offenheit (*@OPEN*) oder Geschlossenheit (*@CLOSED*) des Körpers entlang zweier Kurven. *pUWMode[0]* spezifiziert, ob der Körper entlang der Drehachse geschlossen ist. Dies ist für Körper mit $pArg = 2\pi$ im allgemeinen der Fall. *pUWMode[1]* spezifiziert, ob sich eine Geschlossenheit des Körpers bezüglich der Definitionskurve ergibt (durch Zusammenführung des ersten und letztens Punktes). Dies ist im allgemeinen nicht der Fall.
  - *pCMode* definiert die Offenheit (*@OPEN*) oder Geschlossenheit (*@CLOSED*) des Körpers bezüglich zweier Flächen. *pCMode[0]* spezifiziert, ob sich eventuell ergebende Schnittflächen des Körpers geschlossen werden sollen. Dies ist nur für Körper mit $pArg = 2\pi$ notwendig. *pCMode[1]* spezifiziert, ob die Deckelflächen des Körpers erzeugt werden sollen oder nicht.
  - *pFlip* erzwingt eine Invertierung der Reihenfolge in *pPoints*, wenn es den Wert 1 hat. Ansonsten muß der Wert 0 sein.

7.11 OiSphere

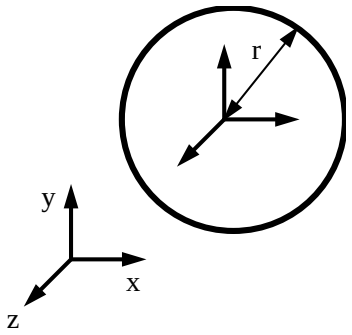

Abbildung 7.10: Der geometrische Typ *OiSphere*

Beschreibung

- *OiSphere* repräsentiert eine homogene Kugel, die um den Ursprung des lokalen Koordinatensystems zentriert ist. Der Radius der Kugel kann nach der Erzeugung geändert werden.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiSphere(pFather(MObject), pName(Symbol), pRadius(Float))*

  Die Funktion initialisiert eine Instanz vom Typ *OiSphere*. Der initiale Radius der Kugel wird durch die positive Zahl *pRadius* angegeben.

Methoden

- *setRadius(pRadius(Float))* $\rightarrow$ *Void*

  Die Funktion setzt den Radius der Kugel. *pRadius* muß eine positive Zahl sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getRadius()* $\rightarrow$ *Float*

  Die Funktion liefert den aktuellen Radius der Kugel.

7.12 OiSweep

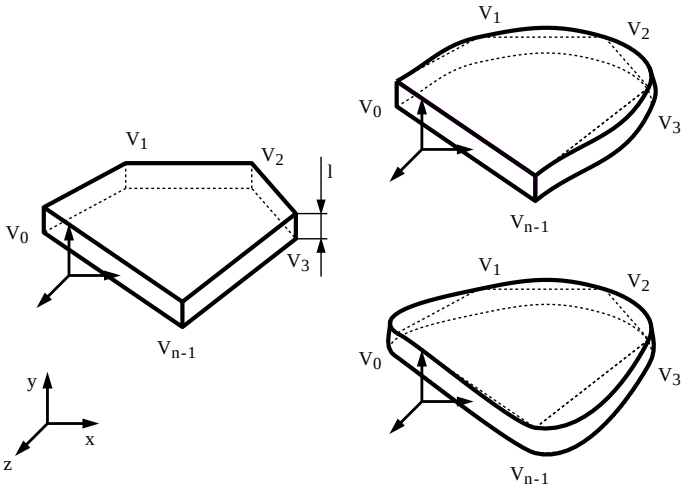

Abbildung 7.11: Der geometrische Typ *OiSweep*

Beschreibung

- *OiSweep* beschreibt einen Solid-Körper, der durch die planare Verschiebung einer dreidimensionalen planaren Kurve entlang einer Achse definiert wird. Die Kurve muß nach der Rechte-Hand-Regel wie folgt definiert sein: Wenn der Daumen der rechten Hand in Richtung der Verschiebung zeigt, geben die restlichen Finger der Hand die Orientierung an. Andernfalls muß eine Invertierung erfolgen.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiSweep(pFather(MObject), pName(Symbol), pMode(Symbol), pAxis(Float[3]), pLength(Float), pPoints(Float[3][]), pUMode(Symbol), pCMode(Symbol[2]), pFlip(Int))*

  Die Funktion initialisiert eine Instanz vom Typ *OiSweep*. Dazu sind folgende spezifische Parameter anzugeben:

  - *pMode* spezifiziert, ob der Körper entlang der Definitionskurve geglättet sein soll (*@SMOOTH*) oder nicht (*@LINEAR*).
  - *pAxis* definiert die Verschiebeachse in Bezug auf das lokale Koordinatensystem.
  - *pLength* setzt die Länge des Körpers entlang der Verschiebeachse. *pLength* muß eine positive Zahl sein.
  - *pPoints* beschreibt die Definitionskurve.
  - *pUMode* definiert die Offenheit (*@OPEN*) oder Geschlossenheit (*@CLOSED*) des Körpers entlang der Definitionskurve. Ist *pUMode = @OPEN*, werden Endpunkt und Anfangspunkt von *pPoints* durch eine Gerade verbunden. Im anderen Fall erfolgt eine entsprechend weiche Verbindung.
  - *pCMode* definiert die Offenheit (*@OPEN*) oder Geschlossenheit (*@CLOSED*) des Körpers bezüglich zweier Flächen. *pCMode[0]* spezifiziert, ob die Seitenflächen des Körpers geschlossen werden sollen oder nicht. *pCMode[1]* spezifiziert, ob die Verbindung des letzten Punktes mit dem ersten Punkt geschlossen werden soll oder nicht.
  - *pFlip* erzwingt eine Invertierung der Reihenfolge in *pPoints*, wenn es den Wert 1 hat. Ansonsten muß der Wert 0 sein.

Methoden

- *setLength(pLength(Float))* $\rightarrow$ *Void*

  Die Funktion setzt die Länge des Objekts entlang der Verschiebeachse. *pLength* muß eine positive Zahl sein.

  Gegebenenfalls muß danach eine Anpassung der Ausrichtung erfolgen (*setAlignment()*).

- *getLength()* $\rightarrow$ *Float*

  Die Funktion liefert die aktuelle Länge des Objekts entlang der Verschiebeachse.

7.13 OiSurface

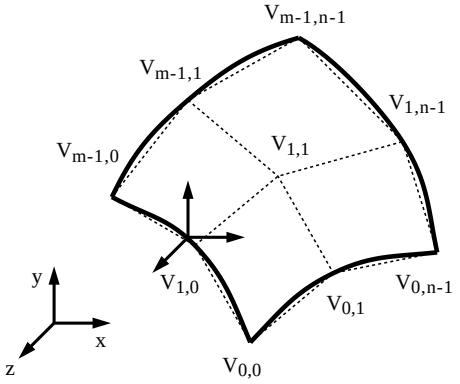

Abbildung 7.12: Der geometrische Typ *OiSurface*

Beschreibung

- *OiSurface* beschreibt eine Primitive, die durch ein zweidimensionales Netz von dreidimensionalen Stützpunkten definiert wird. Dabei sind u und w die Dimensionen des Netzes.
- **Supertyp:** *OiGeometry*

Initialisierung

- *OiSurface(pFather(MObject), pName(Symbol), pUDim(Int), pWDim(Int), pPoints(Float[3][pUDim(pWDim]), pUWMode(Symbol[2])*

  Die Funktion initialisiert eine Instanz vom Typ *OiSurface*. Dazu sind folgende spezifische Parameter anzugeben:

  - *pUDim* definiert die u-Dimension des Netzes.
  - *pWDim* definiert die w-Dimension des Netzes.
  - *pPoints* beschreibt ein Array mit den Definitionspunkten. Innerhalb eines Patches gibt die Rechte-Hand-Regel die Orientierung an, d.h., wenn der Daumen der rechten Hand senkrecht auf dem Patch steht, geben die restlichen Finger der Hand die Orientierung an.
  - *pUWMode* definiert die Offenheit (*@OPEN*) oder Geschlossenheit (*@CLOSED*) des Primitivs entlang u- und w-Dimension. Ist *pUWMode[0] = @OPEN*, erfolgt keine Verbindung des Netzes in u-Richtung. Ist *pUWMode[1] = @OPEN*, erfolgt keine Verbindung des Netzes in w-Richtung.

Kapitel 8

Globale Planungstypen

Dieses Kapitel beschreibt globale, planungsübergreifende Basistypen. Diese Basistypen sind unabhängig von konkreten Planungselementen (Möbelstücken) und damit auch unabhängig von konkreten geometrischen Ausprägungen.

Den hier beschriebenen Typen liegt das in Abb. 8.1 dargestellte konzeptuelle Modell zugrunde.

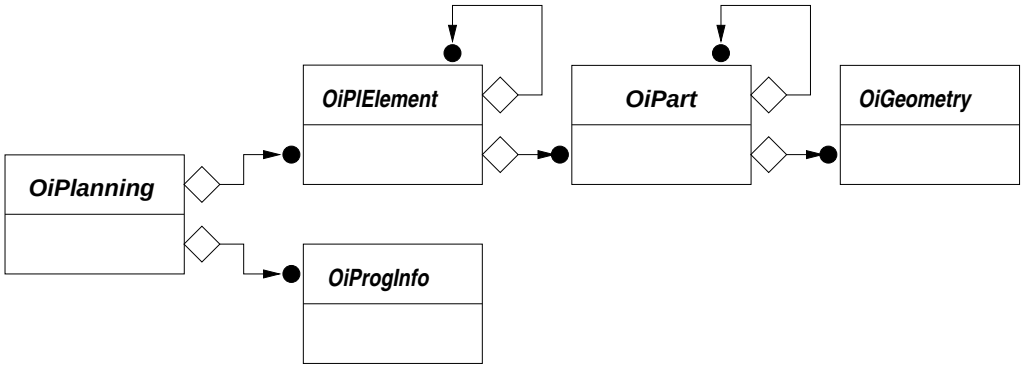

Abbildung 8.1: Konzeptuelles Modell der globalen Planungstypen

8.1 OiPlanning

Beschreibung

- Eine Instanz dieses Typs fungiert als Wurzelobjekt einer kompletten Planungshierarchie und realisiert globale Planungslogiken für die Elemente der Planung (Typ *OiPlElement*).
- Zu den weiteren Aufgaben des globalen Planungsobjektes gehören:

  - Die Definition einer *Planungsgrenze*, die den Raum festlegt, in dessem Rahmen die Planungselemente plaziert werden können.
  - Die Überwachung und Behandlung der Transformation von Planungselementen zum Zwecke der Vermeidung von Kollisionen und Planungsgrenzüberschreitungen.
  - Die Verwaltung und Verwertung von Informationen über Eigenschaften und Bedingungen der (Möbel)Programme, zu denen die in der Planung vertretenen Elemente gehören (Typ *OiProgInfo*).
  - Verwendung eines Produktdatenmanagers zum Zugriff auf Produktdaten (s.a. Kap. 9).

- **Schnittstelle(n):** Base, Complex, Property, Material

Initialisierung

- *OiPlanning(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPlanning*. Die Instanz ist nicht selektierbar. Die initiale Planungsgrenze liegt im Unendlichen.

Methoden

Allgemeine Methoden

- *setLanguage(pLang(String))* → *Void*

  Die Funktion spezifiziert die Sprache, die nachfolgend für Meldungen sowie Beschriftungen verwendet werden soll. Der Parameter *pLang* beschreibt die Landessprache durch einen String entsprechend den ISO-639-Festlegungen.

  Beispiele:

  - `"de"` – deutsch
  - `"en"` – englisch
  - `"nl"` – holländisch

- *getLanguage()* → *String*

  Die Funktion liefert die Sprache, die aktuell für Meldungen sowie Beschriftungen verwendet wird.

- *setRegion(pRegion(String))* → *Void*

  Die Funktion spezifiziert den Vertriebsbereich, d.h. im allgemeinen ein Land, für den die aktuelle Planung erstellt wird. Der Parameter *pRegion* beschreibt den Vertriebsbereich durch einen String entsprechend den ISO-3166-Festlegungen (ISO Code 2) oder diesbezüglicher Erweiterungen zur Bezeichnung von Bundesländern, etc.

  Beispiele:

  - `"DE"` – Deutschland

  - `"UK"` – England
  - `"NL"` – Holland

- *getRegion() → String*

  Die Funktion liefert den aktuellen Vertriebsbereich. Falls kein Vertriebsbereich spezifiziert wurde, ist der Rückgabewert vom Typ *Void*.

- *setProgram(pProgr(Symbol)) → Void*

  Die Funktion spezifiziert das aktuell relevante Programm.

  Hinweis: Programme werden anhand ihrer ID (Identifikationssymbol) unterschieden. Die Programm–ID muß über alle Hersteller hinweg eindeutig sein. Sie beginnt deswegen mit einem zweistelligen Herstellerkürzel, dem das Kürzel für das eigentliche Programm folgt.

  Die Programm–ID wird bei bestimmten Operationen zur Delegation von Funktionalität an programmspezifische Informations- oder ähnliche Objekte verwendet. Das aktuell relevante Programm kann entweder extern durch die Laufzeitumgebung, aber auch intern aus einem bestimmten Kontext heraus bestimmt werden, z.B. aus der Zugehörigkeit des aktuell bearbeiteten Planungselements zu einem Programm.

- *delegationDone() → Void*

  Die Funktion signalisiert der impliziten Planungsinstanz die erfolgreiche Abarbeitung einer an eine andere Instanz (*Delegat*) delegierten Funktionalität.

  Hinweis: Die Funktion wird von der Delegatinstanz bei erfolgreicher Abarbeitung der delegierten Funktionalität aufgerufen.

Fehler-Log

Bei komplexen Prüfalgorithmen, die über einer Objektstruktur ausgeführt werden, kann es zu Fehlermeldungen betreffs verschiedener Objekte der Struktur kommen. Anstatt der Ausgabe einer Fehlermeldung aus jeder Prüfmethode für die betreffenden Objekte ist es in der Regel wünschenswert, diese Meldungen aufzusammeln und zusammen in einem Dialog anzuzeigen. Dazu wird von der globalen Planungsinstanz ein sogenanntes *Fehler-Log* verwaltet. Die Instanz bzw. Methode, die einen globalen Prüfvorgang initiiert, erzeugt die für das Log des Prüfvorgangs zu verwendende Datenstruktur und übergibt diese an die Funktion *setErrorLog()* bevor die Ausführung der Prüfung an eine andere Instanz delegiert wird bzw. geerbte Implementierungen gerufen werden. In Implementierungen des Prüfalgorithmus muß dann zunächst mittels der Funktion *getErrorLog()* geprüft werden, ob ein übergeordnetes Log angelegt wurde, in welchem Fall die generierten Meldungen in diesen einzutragen sind und kein eigener Dialog zur Anzeige gestartet werden darf. Ist kein übergeordnetes Log vorhanden, ist diese vor eventuell stattfindenden Delegationen anzulegen und am Ende des Prüfvorgangs ein Dialog zur Anzeige der Meldungen aus dem Log zu starten. Die Datenstruktur, die für das Log verwendet wird, kann für jeden Prüfalgorithmus separat definiert werden.

Hinweis: Ein Beispiel für die Anwendung des Fehler-Logs befindet sich in der Schnittstelle *Article* bei der Durchführung von Konsistenzprüfungen.

- *setErrorLog(pLog(Any)) → Void*

  Die Funktion weist der impliziten Planungsinstanz eine neue Datenstruktur für das Fehler-Log zu.

- *getErrorLog() → Any*

  Die Funktion gibt die Refernz auf die zuletzt mittels Funktion *setErrorLog()* zugewiesene Datenstruktur für das Fehler-Log zurück.

Instanzhierarchie

- *getEnvironment() → MObject*

  Die Funktion liefert das Wurzelobjekt der Hierarchie der Planungsumgebung. Die Funktion liefert einen Wert vom Typ *Void*, falls keine Planungsumgebung existiert.

- *getPlElementUp(pObj(MObject)) → MObject*

  Die Funktion traversiert die Instanzhierarchie beginnend mit der übergebenen Instanz aufwärts bis zum Wurzelobjekt und liefert die erste Instanz, die vom Typ *OiPlElement* ist. Wurde im Traversierungspfad keine solche Instanz gefunden, liefert die Funktion einen Wert vom Typ *Void*.

- *getTopPlElement(pObj(MObject)) → MObject*

  Die Funktion traversiert die Instanzhierarchie beginnend mit der übergebenen Instanz aufwärts bis zum Wurzelobjekt und liefert die oberste Instanz, die vom Typ *OiPlElement* ist. Wurde im Traversierungspfad keine solche Instanz gefunden, liefert die Funktion einen Wert vom Typ *Void*.

- *getPropObj(pObj(MObject)) → MObject*

  Die Funktion traversiert die Instanzhierarchie beginnend mit der übergebenen Instanz aufwärts bis zum Wurzelobjekt und liefert die erste Instanz, die Properties besitzt. Wurde im Traversierungspfad keine solche Instanz gefunden, liefert die Funktion einen Wert vom Typ *Void*.

Planungsgrenze

- *setBorder(pBorder(Float[2][3]) → Void*

  Die Funktion weist der impliziten Planungsinstanz einen neuen Wert für das (achsenorthogonale) Planungsgrenzvolumen zu.
  Der Parameter *pBorder* ist ein Vektor, der aus zwei Vektoren zu je drei *Float*-Werten besteht. Der erste *Float*-Vektor spezifiziert den Ursprung des erlaubten Planungsvolumens in Weltkoordinaten. Der zweite *Float*-Vektor legt die maximale Ausdehnung des erlaubten Planungsvolumens entlang den Achsen x, y und z fest. Spielt die Lage der Planung im Raum keine Rolle, kann anstelle des ersten Vektors auch ein Wert vom Typ *Void* übergeben werden.

- *getBorder() → Float[2][3]*

  Die Funktion liefert die aktuell in der impliziten Planungsinstanz verwendete Spezikation für das Planungsgrenzvolumen. Aufbau und Semantik des Rückgabewertes entsprechen dem Parameter für die Funktion *setBorder()*.

- *checkBorder() → String*

  Die Funktion prüft, ob die Planungsgrenze durch die aktuell in der impliziten Planungsinstanz enthaltenen Planungselemente eingehalten wird.
  Im Fall einer Grenzüberschreitung liefert die Funktion einen String, der eine entsprechende Fehlermeldung enthält. Ist die Grenze eingehalten, liefert die Funktion einen Wert vom Typ *Void*.

Verwaltung von Programm-Informationen

- *addInfoObj(pType(Type), pID(Symbol)) → Void*

  Die Funktion fügt eine Instanz des angegebenen Typs der Menge der Programm-Infoobjekte hinzu und registriert diese unter der angegebenen Programm–ID. Existiert bereits ein Programm-Infoobjekt mit der angegebenen Programm–ID, wird dieses vor dem Einfügen des neuen Objektes beseitigt.

  Hinweis: Die Programm-Infoobjekte werden als nicht-grafische (und damit nicht-sichtbare) Objekte in die Instanzhierarchie der Planung eingefügt. Nach dem Speichern und Neuladen der Planung sind diese Infoobjekte somit sofort verfügbar.

- *delInfoObj(pID(Symbol)) → Void*

  Die Funktion entfernt das Programm-Infoobjekt mit der angegebenen Programm–ID.

- *clearInfoObjs() → Void*

  Die Funktion entfernt alle Programm-Infoobjekte.

- *getInfoIDs() → Symbol[]*

  Die Funktion liefert die Programm–IDs aller registrierten Programm-Infoobjekte.

- *getInfo(pID(Symbol)) → MObject*

  Die Funktion liefert das Programm-Infoobjekt mit der angegebenen Programm–ID bzw. einen Wert vom Typ *Void*, falls kein Programm-Infoobjekt unter der angegebenen Programm–ID registriert ist.

Materialien

- *Material::getMatCategories() → Symbol[]*
- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*
- *Material::getCMaterial(pCat(Symbol)) → Symbol*

- *Material::setCMaterial(pCat(Symbol), pMat(Symbol))* → *Int*
- *Material::getMatName(pMat(Symbol))* → *String*
  Die Standardimplementierung führt eine String-Konvertierung des Symbols durch.

Diese Funktionen implementieren die entsprechenden Funktionen der Schnittstelle *Material* mittels Delegation an die gleichnamigen Funktionen des Programm-Infoobjektes (Typ *OiProgInfo*) des aktuell relevanten Programms (Funktion *setProgram()*).

Element–Verwaltung und Kollisionserkennung

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any))* → *Float[3]*

  Die Funktion prüft, ob eine Instanz des angegebenen Typs als Element in die Planung eingefügt werden kann und liefert im positiven Fall eine gültige Position für das Element.
  (Mehr Informationen zur Semantik der Funktion bzw. deren Parameter siehe Schnittstelle *Complex*.)
  Die Funktion ruft zunächst die gleichnamige Funktion des Programm-Infoobjektes (Typ *OiProgInfo*) für das Programm, dem die im Parameter *pObj* übergebene Instanz angehört. Anschließend erfolgt ggf. eine programmunabhängige Prüfung entsprechend einer globalen Planungslogik. Dazu wird die Hakenfunktion *doCheckAdd()* aufgerufen.
- *doCheckAdd(pType(Type), pObj(MObject), pPos(Float[3]), pParams (Any))* → *Float[3]*

  Die Funktion prüft, unabhängig von konkreten Möbelprogrammen, ob eine Instanz des angegebenen Typs als Element in die Planung eingefügt werden kann und liefert im positiven Fall eine gültige Position für das Element. Die Semantik der Parameter entspricht der Funktion *checkAdd()*. Die Standardimplementierung realisiert das Anfügen des neuen Elements rechts an die bestehende Planung.

  Hinweis: Die Funktion wird von *checkAdd()* gerufen und kann, im Gegensatz zu *checkAdd()*, in Untertypen redefiniert werden, wobei im Fall der Nichtanwendbarkeit der durch den Untertypen implementierten speziellen Planungslogik die gleichnamige Funktion des unmittelbaren Supertyps gerufen werden sollte.

- *Complex::checkChildColl(pObj(MObject), pExclObj(MObject))* → *MObject*

  Die Funktion prüft, ob eine Kollision der übergebenen (Kind)Instanz mit anderen Objekten vorliegt. Enthält das Argument *pExclObj* eine nicht-leere Menge von Objekten, werden diese von der Kollisionsprüfung ausgeschlossen. Die Funktion prüft zunächst auf Kollision mit den Kindern der impliziten Instanz. Objekte, für die die Hakenfunktion *isValidForCollCheck()* den Wert 0 liefert, werden dabei von der Kollisionsprüfung ausgeschlossen. Vor und nach dieser Prüfung werden die Funktionen *startCollCheck()* bzw. *finishCollCheck()* des Programm-Infoobjektes für das Programm gerufen, dem die im Parameter *pObj* übergebene Instanz angehört. Anschließend wird die gleichnamige Funktion des Wurzelobjektes der Hierarchie der Planungsumgebung gerufen (falls dieses existiert und dessen Typ die Schnittstelle *Complex* implementiert). Rückgabewert ist das erste gefundende Objekt, mit dem die übergebene Instanz kollidiert oder ein Wert vom Typ *Void*, falls keine Kollision entdeckt wurde bzw. die Kollisionserkennung ausgeschaltet ist.

- *Complex::isValidForCollCheck(pObj(MObject)) → Int*

  Diese Funktion implementiert die entsprechende Funktion der Schnittstelle *Complex* mittels Delegation an die gleichnamige Funktion des Programm-Infoobjektes (Typ *OiProgInfo*) für das Programm, dem die im Parameter *pObj* übergebene Instanz angehört.

- *Complex::checkElPos(pEl(MObject), pOldPos(Float[3])) → Int*

  Die Funktion implementiert die entsprechende Funktion der Schnittstelle *Complex* mittels Kollisionserkennung und Planungsgrenzenüberwachung (Funktionen *checkChildColl()* und *checkBorder()*).

Element–Transformationen

- *elemTranslation(pEl(MObject), pOldPos(Float[3])) → Void*

  Die Funktion behandelt eine (bereits erfolgte) Translation des angegebenen Planungselements auf folgende Weise.

  – Zunächst wird die generelle Zulässigkeit der Translation des Planungselements überprüft (s.a. Funktion *translateValid()* des Typs *OiPlElement*).
  – Ist die Translation prinzipiell zulässig, wird nun die Funktion *translated()* des übergebenen Planungselements aufgerufen (siehe ebenfalls Typ *OiPlElement*).
  – Wurde von der Funktion *translated()* der Wert 0 zurückgegeben, wird nun von der impliziten Instanz die Gültigkeit der aktuellen Position des Planungselements geprüft (Kollisionserkennung, Planungsgrenzeneinhaltung u.a.). Dabei kann es ggf. unter Zuhilfenahme der im Parameter *pOldPos* übergebenen Position des Planungselements vor der Translation zu einer Korrektur der aktuellen Position kommen.

  Ist das angegebene Objekt keine Instanz des Typs *OiPlElement*, hat die Funktion keinen Effekt.

  Hinweis: Die Funktion wird aus der *TRANSLATE*-Regel des übergebenen Planungselements aufgerufen (Typ *OiPlElement*).

- *elemRotation(pEl(MObject), pOldRot(Float)) → Void*

  Die Funktion behandelt die Rotation des angegebenen Planungselements auf folgende Weise.

  – Zunächst wird die generelle Zulässigkeit der Rotation des Planungselements überprüft (s.a. Funktion *rotateValid()* des Typs *OiPlElement*).
  – Ist die Rotation prinzipiell zulässig, wird nun die Funktion *rotated()* des übergebenen Planungselements aufgerufen (siehe ebenfalls Typ *OiPlElement*).
  – Wurde von der Funktion *rotated()* der Wert 0 zurückgegeben, wird nun von der impliziten Instanz die Gültigkeit des aktuellen Rotationswinkels des Planungselements geprüft (Kollisionserkennung, Planungsgrenzeneinhaltung u.a.). Dabei kann es ggf. unter Zuhilfenahme des im Parameter *pOldRot* übergebenen Rotationswinkels des Planungselements vor der Rotation zu einer Korrektur des aktuellen Winkels kommen.

Ist das angegebene Objekt keine Instanz des Typs *OiPlElement*, hat die Funktion keinen Effekt.

Hinweis: Die Funktion wird aus der *ROTATE*-Regel des übergebenen Planungselements aufgerufen (Typ *OiProgInfo*).

- *checkPosition(pEl(MObject), pPos(Float[3]), pAngles(Float[3])) → Float[2][3]*

  Die Funktion prüft, ob für das übergebene Planungselement die angegebene Position sowie die angegebenen Rotationswinkel (pro Achse) erlaubt sind.

  Rückgabewert ist ein Vektor, der aus zwei Vektoren zu je drei *Float*-Werten besteht. Der erste Vektor spezifiziert eine erlaubte Position, der zweite die Rotationswinkel (pro Achse). Die zurückgegebenen Werte können zur Vermeidung von Kollisionen u.ä. Konflikten von den in den Parametern übergebenen gewünschten Werten in gewissen Grenzen abweichen. Kann das Planungselement prinzipiell nicht an der gewünschten Position (oder in seiner Nähe) plaziert werden, enthält der Rückgabevektor anstelle der Positionsangabe einen Wert vom Typ *Void*.

  Hinweis: Die Funktion wird von der Laufzeitumgebung während eines Dialogs zur expliziten Positionierung eines Planungselements aufgerufen.

Produktdatenmanagement

- *setPDManager(pType(Type)) → Void*

  Die Funktion erzeugt eine Instanz des angegebenen Typs zur Verwendung als globaler Produktdatenmanager (Typ *OiPDManager*). Existiert bereits eine Produktdatenmanager-Instanz, wird diese vorher beseitigt.

- *getPDManager() → MObject*

  Die Funktion liefert die globale Produktdatenmanager–Instanz bzw. einen Wert vom Typ *Void*, falls keine solche Instanz registriert ist.

- *article2Class(pArticle(String)) → String*

  Die Funktion liefert den Namen des Typs, der den anhand seiner Artikelnummer spezifizierten Artikel modelliert bzw. einen Wert vom Typ *Void*, falls für den Artikel keine Zuordnung gefunden wurde. Ist eine globale Produktdatenmanager–Instanz registriert, wird die Anfrage an diese Instanz delegiert.

- *addProductDB(pType(Type), pID(Symbol), pPath(String), pProgList(Symbol[]) → MObject*

  Die Funktion erzeugt eine Instanz des übergebenen Typs (Untertyp von *OiProductDB*) und registriert sie unter der angegebenen ID beim globalen Produktdatenmanager. Im Parameter *pPath* wird der Dateisystem–Pfad des Verzeichnisses übergeben (relativ zum Wurzelverzeichnis der Laufzeitumgebung), das die Dateien der Produktdatenbank enthält. Der zusätzliche Parameter *pProgList* spezifiziert die Programme (IDs), die in der Datenbank repräsentiert sind. Ist bereits eine Produktdatenbank unter der angegebenen ID registriert, wird die Liste der Programme der Produktdatenbank ggf. erweitert.

  Rückgabewert ist die Referenz auf die (erzeugte) Produktdatenbank-Instanz.

Hinweis: Die Funktion bewirkt denselben Effekt wie die gleichnamige Funktion des Typs *OiPDManager*.

Verschiedenes

- *checkConsistency() → Int*

  Die Funktion prüft die Konsistenz und Vollständigkeit der Planung. Gegebenenfalls werden Korrekturen oder Ergänzungen vorgenommen bzw. Fehlermeldungen generiert. Die Funktion liefert True, wenn die Planung konsistent ist, andernfalls False.

  Die Funktion ruft zunächst die gleichnamige Funktion auf allen registrierten Instanzen von *OiProgInfo* auf und anschließend auf allen Kindern vom Typ *OiPlElement*. Das Ergebnis der Prüfung ist False, wenn mindestens für eine Instanz die Prüfung nicht erfolgreich war. Bei der Prüfung wird das in der Schnittstelle *Article* bei der Funktion *checkConsistency()* beschriebene Fehler-Log verwendet.

  Hinweis: Die Funktion wird typischerweise von der Laufzeitumgebung vor der Erstellung einer Bestellliste aufgerufen.

- *checkObjConsistency(pObj(MObject) → Int*

  Die Funktion führt eine Konsistenzprüfung auf der übergebenen Instanz vom Typ *OiPlElement* oder *OiPart* aus. Neben dem Aufruf der Methode *checkConsistency()* auf der übergebenen Instanz kann die Funktion weitere zusätzliche Aktionen durchführen, z.B. das Anzeigen oder Entfernen eines visuellen Feedbacks bei fehlerhaften Artikeln bzw. das Hinzufügen oder Entfernen eines Eintrags im globalen Fehler-Log.

- *doSpecial(pPID(Symbol), pOp(Symbol), pArgs(Any)) → Any*

  Die Funktion führt unter Verwendung der übergebenen Argumente die angegebene Operation bezüglich des durch die übergebene ID spezifizierten Programms durch. Ist für das Programm ein Programm-Infoobjekt registriert, wird die Funktion an dieses delegiert (Typ *OiProgInfo*). Der Rückgabewert ist operationsabhängig.

  Hinweis: Die Funktion kann zur Erweiterung der Funktionalität eines Planungssystems verwendet werden, ohne dazu die Schnittstelle zwischen Laufzeitumgebung und globaler Planungsinstanz erweitern zu müssen.

Regeln

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  Die Regel verhindert das Entfernen von Planungselementen, deren Funktion *removeValid()* den Wert 0 liefert.

8.2 OiProgInfo

Beschreibung

- Instanzen dieses Typs verwalten Informationen über ein (Möbel)Programm (Anh. I) bzw. realisieren auf Anforderung der globalen Planungsinstanz (Typ *OiPlanning*) programmspezifische Funktionen.
- **Schnittstelle(n):** *MObject*, *Property*

Initialisierung

- *OiProgInfo(pFather(MObject), pName(Symbol), pPID(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiProgInfo* mit der angegebenen Programm–ID. Die Programm–ID kann nachträglich nicht mehr geändert werden.

Methoden

Allgemeine Methoden

- *getID()* $\rightarrow$ *Symbol*

  Die Funktion liefert die ID des Programms, für welches die implizite Instanz verantwortlich ist.

- *getPlanning()* $\rightarrow$ *MObject*

  Die Funktion liefert das Wurzelobjekt der Planungshierarchie (*t*), falls dieses eine Instanz vom Typ *OiPlanning* ist, andernfalls einen Wert vom Typ *Void*.

- *checkConsistency()* $\rightarrow$ *Int*

  Die Funktion führt eine programmspezifische Konsistenzprüfung durch. Sie wird von der globalen Planungsinstanz vom Typ *OiPlanning* bei einer globalen Konsistenzprüfung vor der Durchführung der Prüfung auf den Planungselementen gerufen.

- *doSpecial(pOp(Symbol), pArgs(Any))* $\rightarrow$ *Any*

  Die Funktion führt unter Verwendung der übergebenen Argumente die angegebene Operation durch (s.a. gleichnamige Funktion des Typs *OiPlanning*).

Materialien

- *getMatCategories()* $\rightarrow$ *Symbol[]*
- *getCMaterials(pCat(Symbol))* $\rightarrow$ *Symbol[]*
- *setCMaterial(pCat(Symbol), pMat(Symbol))* $\rightarrow$ *Int*
- *getCMaterial(pCat(Symbol))* $\rightarrow$ *Symbol*

- *getMatName(pMat(Symbol)) → String*

  Die Standardimplementierung führt eine String-Konvertierung des Symbols durch.

Diese Funktionen stellen programmspezifische Versionen der entsprechenden Funktionen der Schnittstelle *Material* dar und werden von den gleichnamigen Funktionen der globalen Planungsinstanz (Typ *OiPlanning*) aufgerufen.

Element–Verwaltung und Kollisionserkennung

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  Die Funktion prüft, ob eine Instanz des angegebenen Typs als Nachbarelement des im Parameter *pObj* übergebenen Programmelements in die Planung eingefügt werden kann und liefert im positiven Fall eine gültige Position für das Element.
  Semantik der Funktion bzw. deren Parameter entspricht der gleichnamigen Funktion der globalen Planungsinstanz (Typ *OiPlanning*) und wird von dieser gerufen.

- *isValidForCollCheck(pObj(MObject)) → Int*

  Die Funktion liefert 1, wenn das angegebene Programmelement bei der Kollisionsprüfung berücksichtigt werden soll, andernfalls 0. Die Funktion wird von der gleichnamigen Funktion der globalen Planungsinstanz (Typ *OiPlanning*) aufgerufen.

- *startCollCheck(pObj(MObject)) → Void*

  Die Funktion führt erforderliche Aktionen aus, bevor das angegebene Programmelement auf Kollision mit anderen Planungselementen geprüft wird.
  Sie wird von der Funktion *checkChildObj()* der globalen Planungsinstanz (Typ *OiPlanning*) aufgerufen. Die Standardimplementierung der Funktion führt keine Aktionen aus.

- *finishCollCheck(pObj(MObject)) → Void*

  Die Funktion führt erforderliche Aktionen aus, nachdem das angegebene Programmelement auf Kollision mit anderen Planungselementen geprüft wurde.
  Sie wird von der Funktion *checkChildObj()* der globalen Planungsinstanz (Typ *OiPlanning*) aufgerufen. Die Standardimplementierung der Funktion führt keine Aktionen aus.

8.3 OiPlElement

Beschreibung

- Instanzen des Typs *OiPlElement* repräsentieren selbständige Elemente einer Planung.
- Planungselemente kooperieren auf definierte Weise mit der globalen Planungsinstanz (Typ *OiPlanning*).
- **Schnittstelle(n):** Base, Complex, Material, Property, Article

Initialisierung

- *OiPlElement(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPlElement*.

  Die Initialisierungsfunktion von konkreten Untertypen muß die Eigenschaften des Planungselements definieren. Dies erfolgt entweder mittels der Funktion *setupProperty()* der Schnittstelle *Property* oder durch Delegation an die Funktion *setupProps()* des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

Methoden

Allgemeine Methoden

- *getPlanning()* $\rightarrow$ *MObject*

  Die Funktion liefert das Wurzelobjekt der Planungshierarchie (t), falls dieses eine Instanz von *OiPlanning* ist, andernfalls einen Wert vom Typ *Void*.

- *setPlProgram()* $\rightarrow$ *Void*

  Die Funktion weist der globalen Planungsinstanz mittels der Funktion *setProgram()* (Typ *OiPlanning*) als aktuell relevantes Programm das durch die Funktion *getProgram()* spezifizierte eigene Programm zu.

Räumliches Modell

- *setWidth(pWidth(Float))* $\rightarrow$ *Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Breite zu.

- *Complex::getWidth()* $\rightarrow$ *Float*

  Die Funktion liefert die Breite der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setWidth()* ein Wert für die Breite zugewiesen, wird dieser zurückgegeben, andernfalls die Breite des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setHeight(pHeight(Float))* $\rightarrow$ *Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Höhe zu.

- *Complex::getHeight()* $\rightarrow$ *Float*

  Die Funktion liefert die Höhe der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setHeight()* ein Wert für die Höhe zugewiesen, wird dieser zurückgegeben, andernfalls die Höhe des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setDepth(pDepth(Float)) → Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Tiefe zu.

- *Complex::getDepth() → Float*

  Die Funktion liefert die Tiefe der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setDepth()* ein Wert für die Tiefe zugewiesen, wird dieser zurückgegeben, andernfalls die Tiefe des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setOrigin(pOrigin(Float[3])) → Void*

  Die Funktion weist der impliziten Instanz einen Offset des Bezugsursprungs bezüglich des Minimums des lokalen Begrenzungsvolumens zu.

- *getOrigin() → Float[3]*

  Die Funktion liefert den Offset des Bezugsursprungs der impliziten Instanz bezüglich des Minimums des lokalen Begrenzungsvolumens. Wurde während oder nach der Initialisierung mittels der Methode *setOrigin()* ein Wert für den Offset zugewiesen, wird dieser zurückgegeben, andernfalls wird er mit Hilfe der Methode *getLocalBounds()* der Schnittstelle *Base* ermittelt.

Materialien

In den folgenden Funktionen erfolgt zu Beginn jeweils ein Aufruf von *setPlProgram()*.

- *Material::getMatCategories() → Symbol[]*

  Liefert die Liste der aktuell für die implizite Instanz definierten Materialkategorien (ausführliche Spezifikation siehe Schnittstelle *Material*). Die Standardimplementierung liefert einen Wert vom Typ *Void*.

- *Material::getAllMatCats() → Symbol[]*

  Liefert die Liste *aller* potentiell für die implizite Instanz definierbaren Materialkategorien. Die Standardimplementierung liefert den Rückgabewert der Funktion *getMatCategories()*.

- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*

  Liefert die Liste aller innerhalb der übergebenen Materialkategorie für die implizite Instanz anwendbaren Materialien (ausführliche Spezifikation siehe Schnittstelle *Material*). Die Standardimplementierung liefert den Rückgabewert der gleichnamigen Funktion der globalen Planungsinstanz, wenn deren Typ *OiPlanning* ist, ansonsten einen Wert vom Typ *Void*.

- *Material::getCMaterial(pCat(Symbol)) → Symbol*

  Liefert das der impliziten Instanz in der übergebenen Materialkategorie aktuell zugewiesene Material bzw. einen Wert vom Typ *Void*, wenn die implizite Instanz aktuell nicht der übergebenen Materialkategorie angehört. Die Standardimplementierung liefert den Rückgabewert der gleichnamigen Funktion der globalen Planungsinstanz, wenn deren Typ *OiPlanning* ist, ansonsten einen Wert vom Typ *Void*.

Hinweis: Konkrete Unterklassen müssen diese Methode so überschreiben, daß das im Objekt für diese Kategorie aktuell gesetzte Material geliefert wird. Nur bei noch nicht erfolgter expliziter Zuweisung eines Materials in dieser Kategorie an das Objekt muß die Standardimplementierung (Aufruf des Vaterobjekts) erfolgen.

- *Material::getMatName(pMat(Symbol))* → *String*

  Liefert für die implizite Instanz den Materialnamen zu dem übergebenen Material bzw. einen Wert vom Typ *Void*, wenn es sich um ein unbekanntes Material handelt. Die Standardimplementierung liefert den Rückgabewert der gleichnamigen Funktion der globalen Planungsinstanz, wenn deren Typ *OiPlanning* ist, ansonsten den Rückgabewert der gleichnamigen Funktion der Vaterinstanz, falls deren Typ die Schnittstelle *Material* implementiert.

Elementerzeugung

- *isElemCatValid(pCat(Symbol))* → *Int*

  Die Funktion liefert 1, wenn Instanzen der angegebenen Kategorie der impliziten Instanz als Elemente zugefügt werden können, andernfalls 0.

  Die Standardimplementierung liefert 0.

  Beispiel: Die Funktion *isElemCatValid()* eines Tischtyps, auf den Instanzen der Kategorie *@TOP_ELEM* gestellt werden können, muß für diese Kategorie 1 liefern.

  Hinweis: Beim Überschreiben der Funktion in abgeleiteten Typen muß nach dem Testen auf spezielle Kategorien immer die geerbte Funktion gerufen werden, damit auch für die von Supertypen erlaubten Kategorien 1 geliefert wird.

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any))* → *Float[3]*

  Die Funktion prüft, ob eine Instanz des angegebenen Typs als Element in die Planung eingefügt werden kann und liefert im positiven Fall eine gültige Position für das Element.

  Die Standardimplementierung realisiert die Plazierung von Elementen der Kategorie *@TOP_ELEM*, falls die Funktion *isElemCatValid()* für diese Kategorie 1 liefert. Dabei werden die Funktionen *getWidth()*, *getHeight()*, *getDepth()* und *getOrigin()* verwendet.

- *getPDistance()* → *Float*

  Die Funktion liefert den gewünschten initialen Abstand zum Vorgängerelement.
  Die Standardimplementierung liefert den minimalen x-Wert des lokalen Begrenzungsvolumens.

  Hinweis: Die Funktion kann innerhalb der Funktion *checkAdd()* der Vaterinstanz verwendet werden. Der von der Funktion gelieferte Wert kann vor dem Einfügen des neuen Elements durch einen Dialog vom Nutzer abgefragt werden. Dazu muß von Untertypen eine entsprechende set-Funktion bereitgestellt werden.

- *getWallOffset() → Float*

  Die Funktion liefert den gewünschten initialen Abstand zu einem Wandelement, vor dem die implizite Instanz plaziert werden soll.
  Die Standardimplementierung liefert 0.01 minus den minimalen z-Wert des lokalen Begrenzungsvolumens.

  Hinweis: Die Funktion kann innerhalb der Funktion *checkAdd()* der Vaterinstanz verwendet werden. Der von der Funktion gelieferte Wert kann vor dem Einfügen des neuen Elements durch einen Dialog vom Nutzer abgefragt werden. Dazu muß von Untertypen eine entsprechende set-Funktion bereitgestellt werden.

- *onCreate(pRot(Float), pObj(MObject), pParams(Any)) → Void*

  Die Funktion kann nach der Erzeugung der impliziten Instanz aufgerufen werden und beendet den Gesamtvorgang des interaktiven Einfügens der Instanz in die Planung. Übergeben werden die geforderte Rotation bezüglich y-Achse in positiver Richtung, das Nachbarelement, an welches die implizite Instanz angefügt wurde, sowie ein weiterer beliebiger Parameter. Die Standardimplementierung realisiert die geforderte Rotation bezüglich y-Achse in positiver Richtung.

  Hinweis: Die Funktion dient zur Einstellung von Objekteigenschaften, die während der Objekterzeugung (in der Funktion *initialize()*) aus Unkenntnis über den Planungskontext nicht vorgenommen werden kann. Die Funktion wird typischerweise zusammen mit den passenden Argumenten während des *checkAdd()* von *OiPlanning* mittels Aufruf von *setMethod()* (Interface *Complex*) gesetzt.

Elementsteuerung

- *elRemoveValid(pObj(MObject)) → Int*

  Die Funktion liefert True, wenn die übergebene Kindinstanz entfernt werden kann. Die Funktion wird in REMOVE_ELEMENT Regeln zusätzlich zur Funktion *removeValid()* der Schnittstelle *Base* für die zu entfernende Instanz gerufen.

  Beispiel: Eine Schrankwandteilplanung kann damit z.B. erreichen, daß nur Elemente am linken und rechten Ende entfernt werden.

  Die Standardimplementierung liefert True.

- *isElOrderSubPos(pObj(MObject)) → Int*

  Die Funktion liefert True, wenn die übergebene Kindinstanz nicht als Unterposten in einer Bestellliste erscheinen darf.

  Beispiel: Die Funktion kann in Algorithmen zur Generierung von Bestelllisten genutzt werden, um bestimmte Teile an spezielle Stellen in der Bestellliste zu verschieben.

  Die Standardimplementierung liefert True.

Produktdaten

- *Article::getArticleSpec() → String*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die Funktion *class2Article()* des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::setArticleSpec(pSpec(String)) → Void*

  Die Standardimplementierung führt keine Aktionen aus.

- *Article::getArticleParams() → Any*

  Die Standardimplementierung liefert einen Wert vom Typ *Void*.

- *Article::getArticlePrice(pLanguage(String)) → Any[]*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::getArticleText(pLanguage(String), pForm(Symbol)) → String[]*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::getArticleFeatures(pLanguage(String)) → Any*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

Konsistenzprüfung

- *Article::checkConsistency() → Int*

  Die Funktion prüft die Konsistenz und Vollständigkeit des Planungselements und wird von *OiPlanning::checkConsistency()* aufgerufen. Gegebenenfalls werden Korrekturen oder Ergänzungen vorgenommen bzw. Fehlermeldungen generiert.

  Die Standardimplementierung delegiert an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*).

Kind–Transformationen

- *elemTranslation(pEl(MObject), pOldPos(Float[3])) → Void*

  Die Funktion behandelt eine (bereits erfolgte) Translation der übergebenen Kindinstanz vom Typ *OiPlElement* oder *OiPart*.

  Für Instanzen vom Typ *OiPlElement* erfolgt dies auf dieselbe Weise wie in der gleichnamigen Funktion von *OiPlanning*. Für Instanzen vom Typ *OiPart* wird die Funktion *onTranslate()* gerufen.

Hinweis: Die Funktion wird aus der *TRANSLATE*-Regel der übergebenen Kindinstanz aufgerufen.

- *elemRotation(pEl(MObject), pOldRot(Float)) → Void*

  Die Funktion behandelt die (bereits erfolgte) Rotation der übergebenen Kindinstanz vom Typ *OiPlElement* oder *OiPart*.

  Für Instanzen vom Typ *OiPlElement* erfolgt dies auf dieselbe Weise wie in der gleichnamigen Funktion von *OiPlanning*. Für Instanzen vom Typ *OiPart* wird die Funktion *onRotate()* gerufen.

  Hinweis: Die Funktion wird aus der *ROTATE*-Regel der übergebenen Kindinstanz aufgerufen.

Translation und Rotation

- *translateValid(pOldPos(Float[3])) → Int*

  Die Funktion liefert 1, wenn das Planungselement von der übergebenen alten Position an die neue, aktuelle Position verschoben werden kann, andernfalls 0.
  Die Standardimplementierung liefert 1.

  Hinweis: Die Funktion wird innerhalb der Funktion *elemTranslation()* der globalen Planungsinstanz verwendet (Typ *OiPlanning*).

- *translated(pOldPos(Float[3])) → Int*

  Die Funktion wird von der Funktion *elemTranslation()* der globalen Planungsinstanz gerufen, um dem Planungselement die Möglichkeit zu geben, individuell auf seine Translation zu reagieren. Der Rückgabewert ist 1, falls die Funktion bereits die Gültigkeit der neuen Position geprüft hat, andernfalls 0.

- *rotateValid(pOldPos(Float)) → Int*

  Die Funktion liefert 1, wenn das Planungselement vom übergebenen alten Rotationswinkel in den neuen, aktuellen Rotationswinkel gedreht werden kann, andernfalls 0.
  Die Standardimplementierung liefert 1.

  Hinweis: Die Funktion wird innerhalb der Funktion *elemRotation()* der globalen Planungsinstanz verwendet (Typ *OiPlanning*).

- *rotated(pOldPos(Float)) → Int*

  Die Funktion wird von der Funktion *elemRotation()* der globalen Planungsinstanz gerufen, um dem Planungselement die Möglichkeit zu geben, individuell auf seine Rotation zu reagieren. Der Rückgabewert ist 1, falls die Funktion bereits die Gültigkeit des neuen Rotationswinkels geprüft hat, andernfalls 0.

Regeln

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  Die Regel verhindert das Entfernen von Kindinstanzen, deren Funktion *removeValid()* False liefert oder für die die Funktion *elRemoveValid()* False liefert.

- *TRANSLATE(pValue(Float[3])) → Int*

  Die Regel delegiert die Behandlung der Translation an die Funktion *elemTranslation()* der globalen Planungsinstanz (Typ *OiPlanning*).

- *ROTATE(pValue(Float)) → Int*

  Die Regel delegiert die Behandlung der Rotation an die Funktion *elemRotation()* der globalen Planungsinstanz (Typ *OiPlanning*).

8.4 OiPart

Beschreibung

- Der Typ *OiPart* ist der Basistyp für funktionale Grundtypen, die als Teile in Planungselementen (Klasse *OiPlElement*) Verwendung finden.
- **Schnittstelle(n):** Base, Complex, Material, Property, Article

Initialisierung

- *OiPart(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPart*.

Methoden

Allgemeine Methoden

- *getPlanning() → MObject*

  Die Funktion liefert das Wurzelobjekt der Planungshierarchie (t), falls dieses eine Instanz von *OiPlanning* ist, andernfalls einen Wert vom Typ *Void*.

Räumliches Modell

- *setWidth(pWidth(Float)) → Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Breite zu.

- *Complex::getWidth() → Float*

  Die Funktion liefert die Breite der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setWidth()* ein Wert für die Breite zugewiesen, wird dieser zurückgegeben, andernfalls die Breite des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setHeight(pHeight(Float)) → Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Höhe zu.

- *Complex::getHeight() → Float*

  Die Funktion liefert die Höhe der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setHeight()* ein Wert für die Höhe zugewiesen, wird dieser zurückgegeben, andernfalls die Höhe des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setDepth(pDepth(Float)) → Void*

  Die Funktion weist der impliziten Instanz einen expliziten Wert für die Ausdehnung in der Tiefe zu.

- *Complex::getDepth() → Float*

  Die Funktion liefert die Tiefe der impliziten Instanz. Wurde während oder nach der Initialisierung mittels der Methode *setDepth()* ein Wert für die Tiefe zugewiesen, wird dieser zurückgegeben, andernfalls die Tiefe des durch die Methode *getLocalBounds()* (Schnittstelle *Base*) ermittelten Begrenzungsvolumens.

- *setOrigin(pOrigin(Float[3])) → Void*

  Die Funktion weist der impliziten Instanz einen Offset des Bezugsursprungs bezüglich des Minimums des lokalen Begrenzungsvolumens zu.

- *getOrigin() → Float[3]*

  Die Funktion liefert den Offset des Bezugsursprungs der impliziten Instanz bezüglich des Minimums des lokalen Begrenzungsvolumens. Wurde während oder nach der Initialisierung mittels der Methode *setOrigin()* ein Wert für den Offset zugewiesen, wird dieser zurückgegeben, andernfalls wird er mit Hilfe der Methode *getLocalBounds()* der Schnittstelle *Base* ermittelt.

Materialien

- *Material::getMatCategories() → Symbol[]*

  Liefert die Liste der aktuell für die implizite Instanz definierten Materialkategorien (ausführliche Spezifikation siehe Schnittstelle *Material*). Die Standardimplementierung liefert einen Wert vom Typ *Void*.

- *Material::getAllMatCats() → Symbol[]*

  Liefert die Liste *aller* potentiell für die implizite Instanz definierbaren Materialkategorien. Die Standardimplementierung liefert den Rückgabewert der Funktion *getMatCategories()*.

- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*

  Liefert die Liste aller innerhalb der übergebenen Materialkategorie für die implizite Instanz anwendbaren Materialien (ausführliche Spezifikation siehe Schnittstelle *Material*). Die Standardimplementierung liefert den Rückgabewert der gleichnamigen Funktion der Vaterinstanz, sofern deren Typ die Schnittstelle *Material* implementiert, ansonsten einen Wert vom Typ *Void*.

- *Material::getCMaterial(pCat(Symbol)) → Symbol*

  Liefert das der impliziten Instanz in der übergebenen Materialkategorie aktuell zugewiesene Material bzw. einen Wert vom Typ *Void*, wenn die implizite Instanz aktuell nicht der übergebenen Materialkategorie angehört. Die Standardimplementierung liefert den durch die gleichnamige Funktion von der Vaterinstanz erhaltenen Rückgabewert.

- *Material::getMatName(pMat(Symbol)) → String*

  Liefert für die implizite Instanz den Materialnamen zu dem übergebenen Material bzw. einen Wert vom Typ *Void*, wenn es sich um ein unbekanntes Material handelt. Die Standardimplementierung liefert den Rückgabewert der gleichnamigen Funktion der globalen Planungsinstanz, wenn deren Typ *OiPlanning* ist, ansonsten den Rückgabewert der gleichnamigen Funktion der Vaterinstanz, falls deren Typ die Schnittstelle *Material* implementiert.

Elementerzeugung

- *isElemCatValid(pCat(Symbol)) → Int*

  Die Funktion liefert 1, wenn Instanzen der angegebenen Kategorie der impliziten Instanz als Elemente zugefügt werden können, andernfalls 0.

  Die Standardimplementierung liefert 0.

  Beispiel: Die Funktion *isElemCatValid()* eines Tischtyps, auf den Instanzen der Kategorie *@TOP_ELEM* gestellt werden können, muß für diese Kategorie 1 liefern.

  Hinweis: Beim Überschreiben der Funktion in abgeleiteten Typen muß nach dem Testen auf spezielle Kategorien immer die geerbte Funktion gerufen werden, damit auch für die von Supertypen erlaubten Kategorien 1 geliefert wird.

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  Die Funktion prüft, ob eine Instanz des angegebenen Typs als Element in die Planung eingefügt werden kann und liefert im positiven Fall eine gültige Position für das Element.

  Die Standardimplementierung realisiert die Plazierung von Elementen der Kategorie *@TOP_ELEM*, falls die Funktion *isElemCatValid()* für diese Kategorie 1 liefert. Dabei werden die Funktionen *getWidth()*, *getHeight()*, *getDepth()* und *getOrigin()* verwendet.

Elementsteuerung

- *elRemoveValid(pObj(MObject)) → Int*

  Die Funktion liefert True, wenn die übergebene Kindinstanz entfernt werden kann. Die Funktion wird in REMOVE_ELEMENT Regeln zusätzlich zur Funktion *removeValid()* der Schnittstelle *Base* für die zu entfernende Instanz gerufen.

  Beispiel: Eine Schrankwandteilplanung kann damit z.B. erreichen, daß nur Elemente am linken und rechten Ende entfernt werden.

  Die Standardimplementierung liefert True.

- *isElOrderSubPos(pObj(MObject)) → Int*

  Die Funktion liefert True, wenn die übergebene Kindinstanz nicht als Unterposten in einer Bestellliste erscheinen darf.

  Beispiel: Die Funktion kann in Algorithmen zur Generierung von Bestelllisten genutzt werden, um bestimmte Teile an spezielle Stellen in der Bestelllisten zu verschieben.

  Die Standardimplementierung liefert True.

Produktdaten

- *Article::getArticleSpec() → String*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die Funktion *class2Article()* des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::setArticleSpec(pSpec(String)) → Void*

  Die Standardimplementierung führt keine Aktionen aus.

- *Article::getArticleParams() → Any*

  Die Standardimplementierung liefert einen Wert vom Typ *Void*.

- *Article::getArticlePrice(pLanguage(String)) → Any[]*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::getArticleText(pLanguage(String), pForm(Symbol)) → String[]*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

- *Article::getArticleFeatures(pLanguage(String)) → Any*

  Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*), insofern eine solche Instanz existiert.

Konsistenzprüfung

- *Article::checkConsistency() → Int*

  Die Funktion prüft die Konsistenz und Vollständigkeit des Planungselements und wird von *OiPlanning::checkConsistency()* aufgerufen. Gegebenenfalls werden Korrekturen oder Ergänzungen vorgenommen bzw. Fehlermeldungen generiert.

  Die Standardimplementierung delegiert an die gleichnamige Funktion des globalen Produktdatenmanagers (Typ *OiPDManager*).

Translation und Rotation

- *onTranslate(pOldPos(Float[3])) → Void*

  Die Funktion wird von der Translationsregel aufgerufen und dient in abgeleiteten Klassen dazu, ein spezifisches Verhalten bei Verschiebung zu realisieren.

- *onRotate(pOldRot) → Void*

  Die Funktion wird von der Rotationsregel aufgerufen und dient in abgeleiteten Klassen dazu, ein spezifisches Verhalten bei der Rotation zu realisieren.

Regeln

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  Die Regel verhindert das Entfernen von Kindinstanzen, deren Funktion *removeValid()* False liefert oder für die die Funktion *elRemoveValid()* False liefert.

- *TRANSLATE(pValue(Float[3])) → Int*

  Ist die Vaterinstanz vom Typ *OiPlElement*, wird die Behandlung der Translation an dessen Funktion *elemTranslation()* delegiert, anderenfalls an die Funktion *onTranslate()* der impliziten Instanz.

- *ROTATE(pValue(Float)) → Int*

  Ist die Vaterinstanz vom Typ *OiPlElement*, wird die Behandlung der Rotation an dessen Funktion *elemRotation()* delegiert, anderenfalls an die Funktion *onRotate()* der impliziten Instanz.

8.5 OiUtility

Beschreibung

- Der Typ *OiUtility* ist der Basistyp für Typen, die für spezielle Aufgaben Verwendung finden, z.B. zur Repräsentation und Speicherung der globalen Daten eines Programmes.
- **Schnittstelle(n):** MObject

Initialisierung

- *OiUtility(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiUtility*.

8.6 OiPropertyObj

Beschreibung

- Der Typ *OiPropertyObj* ist der Basistyp für Typen, die für spezielle Aufgaben Verwendung finden und Properties besitzen.
- **Supertyp:** OiUtility
- **Schnittstelle(n):** MObject (geerbt), Property

Initialisierung

- *OiPropertyObj(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPropertyObj*.

Methoden

Allgemeine Methoden

- *getPlanning()* $\rightarrow$ *MObject*

  Die Funktion liefert das Wurzelobjekt der Planungshierarchie (t), falls dieses eine Instanz von *OiPlanning* ist, andernfalls einen Wert vom Typ *Void*.

- *isCutable()* $\rightarrow$ *Int*

  Siehe gleichnamige Funktion der Schnittstelle *Base*.

- *removeValid()* $\rightarrow$ *Int*

  Siehe gleichnamige Funktion der Schnittstelle *Base*.

8.7 OiOdbPlElement

Beschreibung

- Der Typ *OiOdbPlElement* ist der Basistyp für Planungselemente, deren Geometrien durch die ODB generiert werden.
- **Supertyp:** OiPlElement

- **Schnittstelle(n):** Base, Complex, Material, Property, Article
- Die wesentlichste Funktion der Klasse *OiOdbPlElement* ist die Bereitstellung der ODB–Information in Form einer durch *getOdbInfo()* zurückgegebenen Hash–Tabelle. Diese enthält einen Eintrag für den ODB–Namen und einen weiteren Eintrag für jede Property, wobei der Property–Key auch als Key in der Hash–Tabelle Verwendung findet. Somit stehen die Werte der Properties in der ODB zur Parametrisierung der Geometrien zur Verfügung.

Initialisierung

- *OiOdbPlElement(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiOdbPlElement* analog zu *OiPlElement*. Zusätzlich wird die Translation in X– und Z–Richtung freigeschaltet und in Y–Richtung gesperrt.

Methoden

Allgemeine Methoden

- *setOdbType(pArticle(String))* → *Void* (protected)

  Die Funktion setzt den ODB–Namen anhand des übergebenen Artikels. Der ODB–Name wird durch Aufruf der Methode *article2ODBType()* auf dem PD–Manager ermittelt. Liefert diese Methode keinen ODB–Namen, wird standardmäßig ein Name generiert. Dieser setzt sich aus der Serie des Artikels und der Artikelbezeichnung zusammen, wobei alle Zeichen in der Artikelbezeichnung außer Buchstaben, Ziffern und dem Unterstrich durch `_XX` ersetzt werden. `XX` steht dabei für die hexadezimale Darstellung des Codes des jeweiligen Zeichens. Der Unterstrich selbst wird durch zwei aufeinanderfolgende Unterstriche ersetzt.

- *setArticleSpec(pArticle(String))* → *Void*

  Die Funktion weist der impliziten Instanz eine neue Grundartikelnummer zu. Dies bewirkt die Initialisierung der ODB–Info und die anschließende Erzeugung der Geometrien mittels eines Aufrufs von *createOdbChildren(@NEW)*.

- *getArticleSpec()* → *String*

  Die Funktion liefert den Namen des Artikels (Grundartikelnummer), dem die implizite Instanz entspricht bzw. einen Wert vom Typ *Void*, falls keine Artikelspezifikation für die implizite Instanz vorliegt.

  Der Name des Artikels wird anhand des ODB–Namens bestimmt.

- *propsChanged(pPKeys(Symbol[]), pDoChecks(Int))* → *Int*

  Wenn die Liste der Property–Keys *pPKeys* nicht leer ist, wird die Funktion *createOdbChildren(@INCR)* aufgerufen, um die Geometrien für diesen Artikel neu zu erzeugen. In der gegenwärtigen Implementierung wird der Parameter *pDoChecks* ignoriert, und der Rückgabewert ist immer 1.

- *setPropValue(pKey(Symbol), pValue(Any)) → Int*

  Zuerst wird die Methode *setPropValue()* der Oberklasse *OiPlElement* aufgerufen. Anschließend wird über alle direkten Kinder der impliziten Instanz iteriert und die Methode *setPropValue()* für jedes Kind aufgerufen, das entweder ein *OiPlElement* oder ein *OiPart* ist.

- *getOdbInfo() → Hash*

  Die Funktion gibt eine Hash–Tabelle mit den ODB–Parametern zurück. Sie enthält den aus der Grundartikelnummer ermittelten ODB–Namen und die aktuellen Property–Werte.

- *createOdbChildren(pVal(Symbol)) → Void*

  Die Funktion steuert die Erzeugung der Kindobjekte über die ODB. Abhängig vom Parameter *pVal* werden die Kindobjekte entweder vollständig neu erzeugt (*@NEW* und *@RULE*) oder an die neue ODB–Info angepaßt (*@INCR*). Normalerweise wird als Argument entweder *@NEW* oder *@INCR* übergeben. Das Argument *@RULE* ist für die Verwendung in der *FINISH_EVAL*–Regel vorgesehen.

  Hinweis: Mit der derzeitigen Implementierung erfolgt immer eine vollständige Neuerzeugung der Kindobjekte.

  Beim erneuten Erzeugen der Kindobjekte werden Kindobjekte, die nicht zu dem Artikel gehören, wie z.B. Zubehörteile, gelöscht und nicht wieder neu erzeugt. Da die erneute Erzeugung der Kindobjekte zu beliebigen Änderungen in der Geometrie des Artikels führen können, ist eine allgemeine Behandlung dieses Problems nicht möglich.

Translation und Rotation

- *translated(pOldPos(Float[3])) → Int*

  Die Funktion wird nach jeder Translation aufgerufen und überprüft, ob das Objekt an der aktuellen Position eine Kollision verursacht. Ist dies der Fall, versucht die Funktion auf der Linie zwischen alter und aktueller Position eine neue Position zu ermitteltn, die möglichst nahe der aktuellen Position ist und auf der das Objekt nicht kollidiert.

Kapitel 9

Typen für Produktdatenmanagement

OFML erlaubt prinzipiell die vollständige Beschreibung von Logiken und Abhängigkeiten von Typen ohne externe Datensätze. Dennoch kann aus verschiedenen Gründen eine Spezifikation von Produktmerkmalen über externe Datensätze gewünscht werden, z.B. um eine existierende Datenanlage direkt in *OFML* verwenden zu können.

Zu diesem Zweck definiert *OFML* eine leistungsfähige, generische Produktdatenmanagement–Schnittstelle. Das Konzept des Produktdatenmanagements (siehe auch Abb. 9.1) sieht vor, daß es eine Menge externer Produktdatenbanken (in möglicherweise verschiedenen Datenformaten) gibt, die jedoch durch einen globalen Produktdatenmanager verwaltet werden und mit diesem über eine einheitliche generische Schnittstelle kommunizieren. Für jedes konkrete Datenformat (aber nicht für jede externe Produktdatenbank) muß nun ein spezieller Schnittstellentyp implementiert werden (Untertypen von *OiProductDB*), der auf *OFML*-Ebene die Interpretation des Datenformates übernimmt.

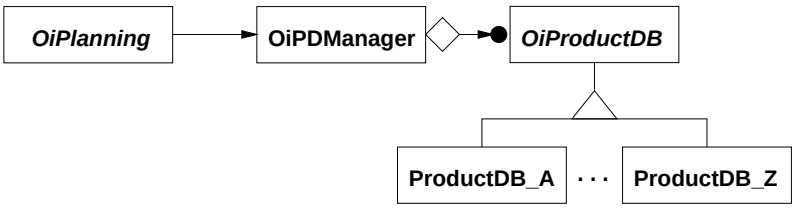

Abbildung 9.1: Konzeptuelles Modell der Produktdatenmanagement-Typen

Beispiel: Ein konkretes Beispiel ist ein Datenformat, das unter Einhaltung des physischen Grundformates aus einem SAP/R3-System generiert wurde. Die Daten sind auf mehrere Tabellen verteilt, die untereinander verknüpft sind. Das Beziehungswissen wird dabei in Ausdrücken der Sprache ABAP/4 abgelegt. Durch die Implementierung eines entsprechenden Untertyps von *OiProductDB* kann nun dieses Format auf *OFML*-Ebene eingelesen werden. Dies beinhaltet die Implementierung eines ABAP/4-Parsers.

9.1 OiPDManager

Beschreibung

- Eine Instanz des Typs *OiPDManager* verwaltet eine Menge von externen Produktdatenbanken (Typ *OiProductDB*) und erlaubt den Zugriff auf die in diesen Datenbanken gelagerten Produktdaten.
- Pro Planung existiert genau eine Instanz dieses Typs. Diese Instanz wird als der Produktdatenmanager bezeichnet. Sie wird mittels der Funktion *setPDManager()* des Typs *OiPlanning* erzeugt.
- Der Produktdatenmanager verwaltet weiterhin die Zuordnung von Typen zu Artikeln und umgekehrt.
- **Schnittstelle(n):** *MObject*

Initialisierung

- *OiPDManager(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiPDManager*.

Methoden

Verwaltung der Produktdatenbanken

- *addProductDB(pType(Type), pID(Symbol), pPath(String), pProgList(Symbol[]) → MObject*

  Die Funktion erzeugt eine Instanz des übergebenen Typs (Untertyp von *OiProductDB*) und registriert sie unter der angegebenen ID. Im Parameter *pPath* wird der Dateisystem–Pfad des Verzeichnisses übergeben (relativ zum Wurzelverzeichnis der Laufzeitumgebung), das die Dateien der Produktdatenbank enthält. Der zusätzliche Parameter *pProgList* spezifiziert die Programme (IDs), die in der Datenbank repräsentiert sind. Ist bereits eine Produktdatenbank unter der angegebenen ID registriert, wird die Liste der Programme der Produktdatenbank ggf. erweitert.

  Rückgabewert ist die Referenz auf die (erzeugte) Produktdatenbank-Instanz.

- *delProductDB(pID(Symbol)) → Void*

  Die Funktion beseitigt und entfernt die Produktdatenbank mit der angegebenen ID von der Menge der registrierten Produktdatenbanken.

- *clearProductDBs() → Void*

  Die Funktion beseitigt und entfernt alle registrierten Produktdatenbanken.

- *getPDB_IDs() → Symbol[]*

  Die Funktion liefert die IDs aller registrierten Produktdatenbanken.

- *getProductDB(pID(Symbol)) → MObject*

  Die Funktion liefert die Produktdatenbank-Instanz, die unter der angegebenen ID registriert ist bzw. einen Wert vom Typ *Void*, falls keine solche vorhanden ist.

- *getPDBFor(pObj(MObject)) → MObject*

  Die Funktion liefert die Produktdatenbank-Instanz, die für das übergebene Planungselement verantwortlich ist bzw. einen Wert vom Typ *Void*, falls keine solche Produktdatenbank gefunden wird. Die Verantwortlichkeit ergibt sich aus der Programmzugehörigkeit des Planungselements.

- *getProgPDB(pPID(Symbol)) → MObject*

  Die Funktion liefert die Produktdatenbank-Instanz, die für das durch die übergebene ID spezifizierte Programm verantwortlich ist bzw. einen Wert vom Typ *Void*, falls keine solche Produktdatenbank gefunden wird.

Zuordnung von Typen zu Artikeln

- *article2Class(pArticle(String)) → String*

  Die Funktion liefert den Namen des Typs, der den anhand seiner Artikelnummer spezifizierten Artikel modelliert bzw. einen Wert vom Typ *Void*, falls keine Zuordnung gefunden wurde.

- *article2Params(pArticle(String)) → String*

  Die Funktion liefert die Parameterwerte für den Typ, der den anhand seiner Artikelnummer spezifizierten Artikel modelliert. Rückgabewert ist ein String, der die in den Produktdaten gespeicherte Darstellung der Parameterwerte enthält. Die Funktion liefert einen Wert vom Typ *Void*, falls in den Produktdaten kein Eintrag für den Artikel gefunden wurde.

- *object2Article(pObj(MObject)) → String*

  Die Funktion liefert den Namen des Artikels (Artikelnummer), dem das übergebene Planungselement entspricht.

  Die Zuordnung ergibt sich aus der Programmzugehörigkeit, dem unmittelbaren Typ und den relevanten Parametern (siehe Funktion *OiPlElement::getArticleParams()*) des Planungselements.

- *class2Articles(pObj(MObject)) → String[]*

  Die Funktion liefert die Liste der Artikel(nummern), die durch die Klasse des übergebenen Planungselementes repräsentiert werden. Der Rückgabewert ist ein Wert vom Typ *Void*, wenn keine Artikel–Zuordnung für die Klasse existiert.

Properties

- *setupProps(pObj(MObject)) → Void*

  Die Funktion definiert die initialen Properties für das angegebene Planungselement anhand der Produktdaten für den Artikel, der dem Typ des Planungselements und seiner Programmzugehörigkeit entspricht. Für Bezeichnungen (Beschriftung, Werte) wird die aktuell in der globalen Planungsinstanz (siehe Typ *OiPlanning*) eingestellte Sprache verwendet.

- *evalPropValue(pObj(MObject), pPKey(Symbol), pValue(Any), pOldValue(Any), pOldArticle(String)) → Int*

  Die Funktion wertet Beziehungswissen in den Produktdaten aus, nachdem die durch ihren Schlüssel spezifizierte Property des angegebenen Planungselements auf den übergebenen neuen Wert gesetzt wurde. Zusätzlich wird der alte Property-Wert sowie die Grundartikelnummer vor der Wertzuweisung übergeben. Die Auswertung der Wertzuweisung kann zu Änderungen der Definition (Wertebereiche) bzw. aktuellen Werte von anderen Properties des Planungselements führen. In diesem Fall liefert die Funktion 0 zurück, andernfalls 1.

  Hinweis: Die Funktion wird aus der Funktion *setPropValue()* der Schnittstelle *Property* aufgerufen.

- *checkConsistency(pObj(MObject)) → Int*

  Die Funktion prüft die Korrektheit der Produktdaten des durch die übergebene Instanz repräsentierten Artikels. Dabei wird für Fehlermeldungen das globale Fehler-Log verwendet (siehe gleichnamige Funktion der Schnittstelle *Article*). Die Standardimplementierung delegiert an die gleichnamige Funktion der für den Artikel zuständigen Produktdatenbank (Typ *OiProductDB*).

Artikelinformationen

- *getXArticleSpec(pObj(MObject), pType(Symbol)) → String*

  Die Funktion liefert die Spezifikation des geforderten Typs für den Artikel, der durch die übergebene Instanz repräsentiert wird bzw. einen Wert vom Typ *Void*, falls keine Artikelspezifikation des geforderten Typs für die implizite Instanz vorliegt. Semantik und Rückgabewert der Funktion entsprechen der gleichnamigen Funktion der Schnittstelle *Article*, wobei nur die Spezifikationstypen @VarCode und @Final erlaubt sind. Die Standardimplementierung der Funktion delegiert die Abfrage beim Spezifikationstyp @VarCode an die Funktion *getVarCode()* und beim Spezifikationstyp @Final an die Funktion *getFinalArticleSpec()* der für die Artikelinstanz zuständigen Produktdatenbank (Typ *OiProductDB*), insofern eine solche Instanz existiert. Dabei wird anstelle der Artikelinstanz seine Grundartikelnummer und eine Liste seiner aktuellen Property-Werte übergeben.

- *setXArticleSpec(pObj(MObject), pType(Symbol), pSpec(String)) → Void*

  Die Funktion weist der übergebenen Artikelinstanz eine neue Artikelspezifikation des angegebenen Typs zu. Semantik und Rückgabewert der Funktion entsprechen der gleichnamigen Funktion der Schnittstelle *Article*, wobei nur der Spezifikationstyp @VarCode erlaubt ist. Die Standardimplementierung der Funktion verwendet dabei die Funktion *varCode2PValues()* der für die Artikelinstanz zuständigen Produktdatenbank (Typ *OiProductDB*), um die zu dem übergebenen Variantencode passenden Produkteigenschaften zu ermitteln. Unterscheiden sich die erhaltenen Werte von den aktuellen Werten der betreffenden Properties, werden diese mittels der Funktion *setPropValue()* (Schnittstelle *Property*) der übergebenen Artikelinstanz neu zugewiesen.

- *getArticlePrice(pObj(MObject), pLanguage(String), ...) → Any[]*

Die Funktion liefert Preisinformationen für das übergebene Planungselement in der angegebenen Sprache. Semantik und Rückgabewert der Funktion entsprechen der gleichnamigen Funktion der Schnittstelle *Article*. Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion der für das Planungselement zuständigen Produktdatenbank (Typ *OiProductDB*), insofern eine solche Instanz existiert. Dabei wird anstelle des Planungselements seine Grundartikelnummer und eine Liste seiner aktuellen Property-Werte übergeben.

- *getArticleText(pObj(MObject), pLanguage(String), pForm(Symbol)) → String[]*

  Die Funktion liefert beschreibende Artikelinformationen für das übergebene Planungselement in der angegebenen Sprache und in der angegebenen Form. Semantik und Rückgabewert der Funktion entsprechen der gleichnamigen Funktion der Schnittstelle *Article*. Die Standardimplementierung der Funktion delegiert die Abfrage an die gleichnamige Funktion der für das Planungselement zuständigen Produktdatenbank (Typ *OiProductDB*), insofern eine solche Instanz existiert. Dabei wird anstelle des Planungselements seine Grundartikelnummer übergeben.

- *getArticleFeatures(pObj(MObject), pLanguage(String)) → Any*

  Die Funktion liefert eine Beschreibung der konfigurierbaren Produkteigenschaften für das übergebene Planungselement in der angegebenen Sprache. Semantik und Rückgabewert der Funktion entsprechen der gleichnamigen Funktion der Schnittstelle *Article*. Die Standardimplementierung der Funktion delegiert die Abfrage an die Funktion *getPropDescription()* der für das Planungselement zuständigen Produktdatenbank (Typ *OiProductDB*), insofern eine solche Instanz existiert. Dabei wird anstelle des Planungselements seine Grundartikelnummer und eine Liste seiner aktuellen Property-Werte übergeben.

9.2 OiProductDB

Beschreibung

- Eine Instanz des Typs *OiProductDB* verwaltet genau eine Produktdatenbank und bietet Dienste zum Zugriff und zur Bewertung von Informationen über Artikel und deren Merkmale.
- **Schnittstelle(n):** *MObject*

Initialisierung

- *OiProductDB(pFather(MObject), pName(Symbol), pID(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *OiProductDB* mit der angegebenen ID. Die ID kann nachträglich nicht mehr geändert werden.

Methoden

Artikelkonfiguration

Einige der unten beschriebenen Funktionen erwarten einen Parameter *pPValues*, der die aktuelle Artikelkonfiguration beinhaltet. Dieser Parameter ist eine Liste, die für jedes Produktmerkmal einen Vektor aus folgenden Elementen enthält:

1. die Merkmalsklasse (*String* oder *Void*, falls nicht relevant)
2. der (sprachunabhängige) Bezeichner des Merkmals (*String*)
3. der Wert des Merkmals (*Any*)
4. die Liste der aktuell möglichen Werte (*List* oder *Void*, falls nicht relevant)
5. der Aktivierungszustand der dem Merkmal zugeordneten Property (*Int*)

Allgemeine Methoden

- *getID()* → *Symbol*

  Die Funktion liefert die ID der Produktdatenbank.

- *setPrograms(pProgList(Symbol[]))* → *Void*

  Die Funktion weist der impliziten Instanz die Menge der Programme (IDs) zu, die in der Produktdatenbank vertreten sind.

- *getPrograms()* → *Symbol[]*

  Die Funktion liefert die Menge der Programme (IDs), die in der Produktdatenbank vertreten sind.

- *setDataBasePath(pDir(String))* → *Void*

  Die Funktion weist der impliziten Instanz das Wurzelverzeichnis der Produktdaten zu.

- *getDataBasePath()* → *String*

  Die Funktion liefert das Wurzelverzeichnis der Produktdaten.

- *getPDManager()* → *MObject*

  Die Funktion liefert die Referenz auf den globalen Produktdatenmanager.

Merkmale und Beziehungswissen

- *hasProductKnowledge()* → *Int*

  Die Funktion liefert True, wenn die Produktdatenbank Beziehungswissen enthält, welches bei der Änderung von Merkmalswerten ausgewertet werden muß.

- *getArticlePropClasses(pArticle(String))* → *Any*

  Die Funktion liefert eine Liste mit den Merkmalsklassen, denen der angegebene Artikel (Grundartikelnummer) zugeordnet ist.

- *getPropDefs(pArticle(String), pPropOffset(Int), pLanguage(String), pChangedProp(Any[]), pPValues(Vector[]))* → *Any*

  Die Funktion liefert die Property-Definitionen für alle Merkmale des übergebenen Artikels (Grundartikelnummer).

  Der Parameter *pPropOffset* spezifiziert die Nummer, ab der Positionen für die Properties vergeben werden können. Für Bezeichnungen (Beschriftung, Werte) wird die angegebene Sprache (ISO-Code) verwendet. Ist keine Sprache spezifiziert (*Void*), wird Englisch verwendet. Ist der Parameter *pChangedProp* kein Wert vom Typ *Void*, so spezifiziert er ein Merkmal, dessen Wert geändert wurde und infolge dessen der Aufruf der Funktion erfolgt. Der Parameter enthält dann einen dreistelligen Vektor aus (sprachunabhängigem) Bezeichner des Merkmals, neuem und alten Merkmalswert. Der Parameter *pPValues* beschreibt die aktuelle Artikelkonfiguration (s.o.) oder ist ein Wert vom Typ *Void*, wenn die Funktion für einen noch nicht initialisierten Artikel gerufen wird.

  Rückgabewert ist eine Liste von siebenstelligen Vektoren. Jeder Vektor beschreibt ein Merkmal und ist wie folgt aufgebaut:

  1. Merkmalsklasse (*String* oder *Void*, wenn nicht relevant).
  2. (sprachunabhängige) Bezeichnung des Merkmals (*String*).
  3. Spezifikation der zugehörigen Property (*Any[5]*) gemäß Funktion *setupProperty()* der Schnittstelle *Property*.
  4. (initialer) Wert des Merkmals bzw. *Void*, wenn kein Wert (vor)definiert ist.
  5. Liste aller möglichen Werte, insofern für das Merkmal mehrere Werte definiert sind (*List* oder *Void*, wenn nicht relevant).
     Die Einträge sind zweistellige Vektoren, die den Wert und die sprachabhängige Beschreibung des Wertes beinhalten. Bei optionalen Merkmalen muß die Liste den Wert „nicht ausgewählt“ enthalten, der als `[@VOID, "@VOID"]` anzugeben ist.
  6. Position in der Property–Liste (*Int*).
  7. Aktivierungsstatus für die Property (*Int*).

- *checkConsistency(pArticle(String), pPValues(Vector[]), pLanguage(String), pErrorList(String[]))* → *Int*

  Die Funktion liefert True, wenn die übergebene Artikelkonfiguration für den angegebenen Artikel (Grundartikelnummer) aus Produktsicht korrekt ist. Fehlermeldungen werden an die im Parameter *pErrorList* übergebene Liste angehängt. Dabei wird die im Parameter *pLanguage* spezifizierte Sprache verwendet.

Artikelinformationen

- *getVarCode(pArticle(String), pPValues(Any[]), ...)* → *String*

Die Funktion liefert den Variantencode für den übergebenen Artikel (Grundartikelnummer) und die übergebene Artikelkonfiguration. Ist ein weiterer optionaler Parameter angegeben, so spezifiziert er, ob die in der Artikelkonfiguration enthaltenen Merkmalswerte OFML-Werte der zugehörigen Property sind (True) oder ob sie in der von der Produktdatenbank verwendeten Form angegeben sind (False). Bei keiner Angabe wird True angenommen.

- *varCode2PValues(pArticle(String), pVarcode(String)) → Any[]*

  Die Funktion liefert die Merkmalswerte zu dem übergebenen Variantencode für den angegebenen Artikel (Grundartikelnummer).

  Rückgabewert ist eine Liste, die für jedes Produktmerkmal einen Vektor aus folgenden Elementen enthält:

  1. die Merkmalsklasse (*String* oder *Void*, falls nicht relevant)
  2. der (sprachunabhängige) Bezeichner des Merkmals (*String*)
  3. der Wert des Merkmals (*Any*)

- *getFinalArticleSpec(pArticle(String), pPValues(Any[])) → String*

  Die Funktion liefert die Endartikelnummer für den übergebenen Artikel (Grundartikelnummer) und die übergebene Artikelkonfiguration.

- *getArticlePrice(pArticle(String), pPValues(Any[]), pLanguage(String), ...) → Any[]*

  Die Funktion liefert Preisinformationen für den übergebenen Artikel (Grundartikelnummer) und die übergebene Artikelkonfiguration in der angegebenen Sprache. Ist ein weiterer optionaler Parameter angegeben, so spezifiziert er die gewünschte Währung.

  Der Rückgabewert entspricht der gleichnamigen Funktion der Schnittstelle *Article*.

- *getArticleText(pArticle(String), pLanguage(String), pForm(Symbol)) → String[]*

  Die Funktion liefert die Artikelbeschreibung für den übergebenen Artikel (Grundartikelnummer) in der angegebenen Sprache und in der angegebenen Form. Mögliche Werte für den Paramter *pForm* sind:

  – `@short` Kurzbeschreibung
  – `@long` Langbeschreibung

  Rückgabewert ist eine Liste von Strings, die die einzelnen Zeilen der Beschreibung enthalten bzw. ein Wert vom Typ *Void*, falls keine Artikelbeschreibung für den Artikel vorliegt.

- *getPropDescription(pArticle(String), pPValues(Any[]), pNeedSymbols(Int), pLanguage(String)) → Any*

  Die Funktion liefert eine Beschreibung der übergebenen Artikelkonfiguration für den spezifizierten Artikel (Grundartikelnummer) in der angegebenen Sprache.

  Rückgabewert ist eine Liste von zweistelligen Vektoren, deren erstes Element (*String*) das Merkmal benennt, während das zweite Element den aktuellen Wert (als Zeichenkette) des Merkmals beinhaltet. Enthält der Parameter *pLanguage* einen Wert vom Typ *Void*, werden sprachunabhängige Bezeichner für Merkmal und Wert geliefert.

  Hat der Parameter *pNeedSymbols* den Wert 1, bestehen die Listeneinträge aus vierstelligen Vektoren mit folgenden Feldern in der angegebenen Reihenfolge:

1. sprachunabhängiges Symbol des Merkmals
2. sprachunabhängige Bezeichnung des Merkmals
3. sprachunabhängiges Symbol des aktuellen Wertes des Merkmals
4. sprachunabhängige Bezeichnung des aktuellen Wertes des Merkmals

Die Funktion liefert einen Wert vom Typ *Void*, falls keine Beschreibungen für die Merkmale vorliegen.

Kapitel 10

Typen der Planungsumgebung

10.1 Die Schnittstelle Wall

Wall definiert die Schnittstelle einer Wand oder einzelner ihrer Teile (z.B. Seiten) zur Möbelplanung.

- *getWallParams()) → [Float, Float, Float[3]]*
  Liefert die geometrischen Parameter um im Laufe des Planungsprozesses Möbel an der Wand zu plazieren. Der Rückgabewert ist ein Vektor mit drei Elementen:

  1. Breite.
  2. Rotationswinkel (in positiver Orientierung um die y-Achse).
  3. Position (Ursprung des lokalen Koordinatensystems).

10.2 OiLevel

Beschreibung

- *OiLevel* modelliert eine Etage eines Gebäudes, die aus einem oder mehreren Räumen bestehen kann.
- **Schnittstelle(n):** Base, Complex

Initialisierung

- *OiLevel(pFather(MObject), pName(Symbol))*
  Die Funktion initialisiert eine Instanz vom Typ *OiLevel*.

Methoden

- *setDefaultHeight(pHeight(Float)) → Void*

  Die Funktion setzt eine Vorgabe für die Höhe neu zu erzeugender Wände. Dieser Wert ist nur wirksam, wenn keine Wände in der Etage vorhanden sind, andernfalls wird die Höhe der Planungswand (s.u.) als Vorgabe verwendet.

- *Complex::getHeight() → Float*

  Die Funktion liefert die maximale Wandhöhe innerhalb der Etage oder, falls keine Wände vorhanden sind, die voreingestellte Höhe.

- *setPlanningWall(pWall(MObject)) → MObject*

  Die Funktion selektiert eine Wand, an die im folgenden Möbel angefügt werden sollen. *pWall* muß ein Objekt sein, daß die Schnittstelle *Wall* (Abschn. 10.1) implementiert. Als spezieller Wert für *pWall* ist $NULL$ zulässig. In diesem Fall wird eine eventuell vorhandene Einstellung gelöscht. Die Funktion liefert als Rückgabewert die neue Planungswand.

- *getPlanningWall() → MObject*

  Die Funktion liefert die eingestellte Planungswand (s.o.). Ist keine Planungswand explizit gesetzt, wird die zuletzt erzeugte Wand verwendet.

- *setPlanningMode(pMode(Int)) → Void*

  Die Funktion setzt den Planungsmodus. Als Minimalforderung müssen die Werte 0 (schaltet Möbelplanung ein) und 1 (schaltet in den Grundmodus der Raumplanung) erkannt werden. Werte > 1 sind je nach Implementierung zulässig.

- *Complex::checkAdd(pType(Type), pObj(MObject), pPos(Float[3]), pParams(Float[])) → Float[3]*

  Die Funktion überprüft das Einfügen einer neuen Wand. *pType* muß ein Subtyp von OiWall sein. *pObj* muß Instanz eines Subtyps von *OiWall* sein, an welche die neue Wand angefügt werden soll. Wird für *pObj* $NULL$ übergeben, wird an die eingestellte Planungswand angefügt. (Abschn. 10.2). *pPos* wird ignoriert und darf $NULL$ sein. *pParams* ist $NULL$ oder enthält eine optionale Liste von Vorgabeparametern. Falls vorhanden, werden diese Parameter wie folgt interpretiert:

  1. Breite.
  2. Anfügewinkel.
  3. Dicke.

  Ist ein Einfügen mit den gegebenen Parametern möglich, wird die Anfügeposition geliefert, sonst $NULL$.

- *objInLevel(pObj(MObject)) → Int*

  Die Funktion liefert 1, wenn das Objekt *pObj* innerhalb der Etage liegt, sonst 0. Dabei sind vereinfachte Tests (z.B. Einschränkung auf umschließendes Rechteck oder Bounding-Box) möglich, Kollision muß nicht beachtet werden.

10.3 OiWall

Beschreibung

- *OiWall* repräsentiert eine Wand als Bestandteil einer Etage. Dies kann sowohl eine Außenwand als auch eine Trennwand sein. In eine Wand können als Kinder Fenster, Türen, usw. eingefügt werden.
- **Schnittstelle(n):** Base, Complex, Properties, Material, Wall

Initialisierung

- *OiWall(pFather(MObject), pName(Symbol))*
  Die Funktion initialisiert eine Instanz vom Typ *OiWall*.

10.4 OiWallSide

Beschreibung

- Eine einzelne Seite einer Wand, an die im Laufe des Planungsprozesses Möbel gestellt werden können.
- **Schnittstelle(n):** Base, Properties, Material, Wall

Initialisierung

- *OiWallSide(pFather(MObject), pName(Symbol))*
  Die Funktion initialisiert eine Instanz vom Typ *OiWallSide*.

Anhang A

Produktdatenmodell

Dieser Anhang beschreibt das den Typen des Produktdatenmanagements (Kap. 9) zugrundeliegende Datenmodell. Abbildung A.1 zeigt eine grafische Darstellung des Modells, die die wesentlichen Konzepte und Begriffe veranschaulicht[1].

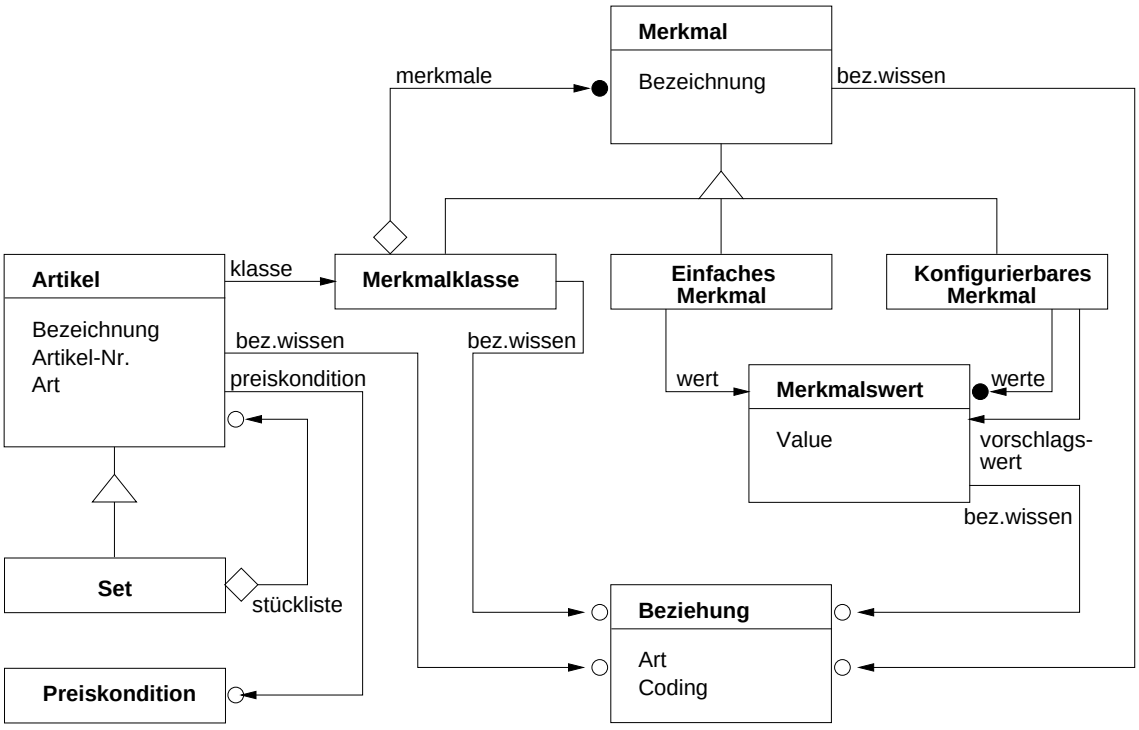

Abbildung A.1: Produktdatenmodell

[1] Die verwendete Notation ist in Anhang G erklärt.

Ergänzungen und Erläuterungen

Jeder Artikel wird einer bestimmten Artikelart zugeordnet, die festlegt, welche Aktionen zu dem Artikel erlaubt sind bzw. welche Bedeutung bestimmte Modelleigenschaften besitzen. Die wesentlichen Artikelarten sind „konfigurierbarer Artikel“, „Fertigteil“ und „Handelsware“.

Merkmale beschreiben die Eigenschaften von Artikeln und werden zu Merkmalsklassen zusammengefaßt. Ein Merkmal in einer Klasse kann wieder eine Klasse sein. Jedem Artikel wird eine oder mehrere Merkmalsklassen zugeordnet.

In Preiskonditionen werden der Grundpreis, sowie für konfigurierbare Artikel über Variantenkonditionen Auf- und Abschläge definiert. Über Beziehungswissen muß dabei die Beziehung zu den entsprechenden Merkmalen bzw. Merkmalswerten hergestellt werden.

Beziehungswissen wird mittels fünf Arten von Beziehungen abgebildet:

- *Vorbedingungen*
  legen fest, ob ein Merkmal bewertet werden darf bzw. ob ein Merkmalswert gesetzt werden darf.
- *Auswahlbedingungen*
  legen fest, daß ein Merkmal bewertet werden muß bzw., daß eine Stücklistenposition ausgewählt werden muß.
- *Aktionen* und *Prozeduren*
  dienen zur Herleitung von Merkmalswerten und werden ausgeführt, wenn ein Merkmalswert gewählt bzw. ein Merkmal bewertet wurde. Aktionen haben dabei deklarativen Charakter und sind unabhängig von der Reihenfolge der Auswertung. Prozeduren hingegen realisieren komplexere Algorithmen und werden nur zu bestimmten Zeitpunkten ausgeführt.
- *Constraints*
  dienen zur Überwachung der Konsistenz einer Konfiguration und können somit nur über das Konfigurationsprofil an einen konfigurierbaren Artikel gebunden werden.

Anhang B

Die 2D-Schnittstelle

B.1 Einleitung

Die in diesem Kapitel beschriebene 2D-Schnittstelle erlaubt die Programmierung von 2D-Objekten. Insgesamt ist die Erzeugung von 2D-Objekten in OFML somit auf folgende Weisen möglich:

- durch Generierung ausgehend von einer (3D-) OFML-Geometrie,
- durch Beschreibung über die OFML-Datenbank ODB [ODB]
- durch Import eines externen 2D-Vektordatensatzes (Kap. C), sowie
- durch Programmierung.

Eine Besonderheit dieser 2D-Programmierschnittstelle besteht darin, daß die erzeugten 2D-Objekte nicht persistent abgespeichert werden. Sie müssen also in den geeigneten Regeln (Kap. 5) gegebenenfalls restauriert werden.

B.2 Die 2D-Objekt-Hierarchie

Die 2D-Objekte werden in der Regel in einem Baum angeordnet, wobei die Knoten des Baums vom Typ `G2DCompound` und die Blätter sinnvollerweise von einem von `G2DPrimitive` abgeleiteten Typ sind. Die Wurzel des Baums hängt immer an einem OFML-Objekt, das die Schnittstelle *MObject* unterstützt, so daß jedes 2D–Objekt direkt oder indirekt einem *MObject*-Objekt zugeordnet werden kann.

Aus OFML heraus werden die 2D-Objekte über ganzzahlige Id's referenziert. Einem *MObject*-Objekt sind nicht zwei 2D-Objekte mit der gleichen Id zugeordnet, d.h., daß unterhalb eines *MObject*-Objekts die Id's eindeutig sind. Die vergebenen Id's sind nicht monoton wachsend, d.h., daß ein neues Objekt die Id eines alten Objekts, das inzwischen gelöscht wurde, erhalten kann.

Das Objekt mit der Id 0 existiert immer[1] und ist vom Typ `G2DCompound`.

[1] Tatsächlich wird es bei Bedarf erzeugt.

B.3 Koordinaten

Alle Koordinaten werden im rechtwinkligen X/Y–Koordinatensystem angegeben, wobei die positive X–Achse nach rechts und die positive Y–Achse nach oben zeigt. Winkelangaben erfolgen grundsätzlich im Bogenmaß und sind mathematisch positiv (entgegen dem Uhrzeigersinn). Der Null–Winkel zeigt in Richtung der positiven X–Achse.

B.4 Methoden

Die Manipulation der 2D–Objekte erfolgt über die in den folgenden Unterabschnitten aufgeführten Methoden.

B.4.1 `new2DObj`

`t.new2DObj(`*parent_id*`, `*object_type*`, ...)`

Alle 2D–Objekte werden mit der Methode `new2DObj` erzeugt. Ihr erstes Argument ist die Id des Vaterobjekts, das vom Typ `G2DCompound` sein muß. Das zweite Argument ist ein Symbol, das die Art des zu erzeugenden 2D–Objekts bestimmt. Die restlichen Argumente hängen vom Typ des zu erzeugenden Objekts ab. Der Rückgabewert ist die Id des neu erzeugten Objekts.

Die genaue Form der Aufrufe von `new2DObj` ist in Abschnitt B.5 in den jeweiligen objekttypspezifischen Unterabschnitten beschrieben.

B.4.2 `delete2DObj`

`t.delete2DObj(`*obj_id*`)`

Die Methode `delete2DObj` entfernt das Objekt mit der angegebenen Id und gegebenenfalls rekursiv alle existierenden Kind–Objekte dieses Objekts.

B.4.3 `set2DObjAttr`

`t.set2DObjAttr(`*obj_id*`, `*attr_type*`, ...)`

Mit der Methode `set2DObjAttr` werden die Attribute bestehender Objekte gesetzt. Das erste Argument ist die Id des Objekts, vom dem ein Attribut gesetzt werden soll. Das zweite Argument ist ein Symbol, welches den Typ des zu setzenden Attributs bestimmt. Die restlichen Argumente hängen vom Attributtyp ab.

Die genaue Form der Aufrufe von `set2DObjAttr` ist in Abschnitt B.6 in den jeweiligen attributspezifischen Unterabschnitten beschrieben.

B.4.4 translate2DObj

`t.translate2DObj(`*obj_id*`, [`x`, `y`])`

Mit der Methode `translate2DObj` wird das `G2DCompound`–Objekt mit der Id *obj_id* relativ zur aktuellen Position im Koordinatensystem des Vaterobjekts um $x; y$ verschoben.

B.4.5 rotate2DObj

`t.rotate2DObj(`*obj_id*`, `*angle*`)`

Mit der Methode `rotate2DObj` wird das `G2DCompound`–Objekt mit der Id *obj_id* relativ zur aktuellen Rotation um den Winkel *angle* gedreht. Die Rotation erfolgt um den Ursprung des Koordinatensystem des Vaterobjektes.

B.5 Objekttypen

In den folgenden Unterabschnitten sind die verfügbaren 2D–Objekttypen mit ihren Attributen aufgeführt. Neben den angegebenen Attributen hat jeder Typ die Attribute *Pickable* und *Snapable*.

B.5.1 G2DCompound

Ein `G2DCompound`–Objekt unterscheidet sich von allen anderen Objekten, die von `G2DPrimitive` abgeleitet sind, dadurch, daß es

- weitere 2D–Objekte als Kinder haben kann und
- transliert, rotiert und skaliert werden kann.

Ein neues `G2DCompound`–Objekt wird mit der Methode

`t.new2DObj(`*parent_id*`, @COMPOUND)`

erzeugt.

B.5.2 G2DPoints

Ein Objekt vom Typ `G2DPoints` besteht aus einer Liste von X/Y–Koordinaten, die die Mittelpunkte der einzelnen Punkte beschreiben. Des weiteren hat es die Attribute *Color*, *PointSize* und *PointSmooth*.

Ein neues `G2DPoints`–Objekt mit n Punkten wird mit der Methode

`t.new2DObj(`*parent_id*`, @POINTS, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

erzeugt.

B.5.3 G2DLines

Ein Objekt vom Typ `G2DLines` besteht aus einzelnen Liniensegmenten, die über ihre Anfangspunkte und Endpunkte im X/Y–Koordinatensystem definiert werden. Es hat die Attribute *Color*, *LineWidth* und *LineStyle*.

Ein neues `G2DLines`–Objekt mit n Linien wird mit der Methode

`t.new2DObj(`*parent_id*`, @LINES, [[`x_0`, `y_0`], ..., [`x_{2n-1}`, `y_{2n-1}`]])`

erzeugt, wobei $x_{2i}; y_{2i}$ den Anfangs- und $x_{2i+1}; y_{2i+1}$ den Endpunkt einer Linie festlegt.

B.5.4 G2DLineStrip

Ein Objekt vom Typ `G2DLineStrip` besteht aus einer Folge von mindestens zwei Punkten, die in der angegebenen Reihenfolge durch einzelne Liniensegmente miteinander verbunden sind, wobei im Gegensatz zu `G2DLineLoop` der letzte Punkt nicht mit dem ersten Punkt verbunden ist. Es hat die Attribute *Color*, *LineWidth* und *LineStyle*.

Ein neues `G2DLineStrip`–Objekt mit n Punkten wird mit der Methode

`t.new2DObj(`*parent_id*`, @LINE_STRIP, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

erzeugt.

B.5.5 G2DLineLoop

Ein Objekt vom Typ `G2DLineLoop` besteht aus einer Folge von mindestens zwei Punkten, die in der angegebenen Reihenfolge durch einzelne Liniensegmente miteinander verbunden sind, wobei im Gegensatz zu `G2DLineStrip` auch der letzte Punkt mit dem ersten Punkt verbunden ist. Es hat die Attribute *Color*, *LineWidth* und *LineStyle*.

Ein neues `G2DLineLoop`–Objekt mit n Punkten wird mit der Methode

`t.new2DObj(`*parent_id*`, @LINE_LOOP, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

erzeugt.

B.5.6 G2DConvexPolygon

Ein Objekt vom Typ `G2DConvexPolygon` wird durch eine Folge von mindestens drei Punkten beschrieben, die miteinander verbunden, einschließlich des letzten mit dem ersten Punkt, ein konvexes Polygon ergeben müssen. Es hat die Attribute *Color*, *PointSize*, *PointSmooth*, *LineWidth*, *LineStyle*, *FillStyle* und *PolygonMode*. Abhängig von *PolygonMode* wird jeweils nur eine Untermenge der Attribute verwendet.

Ein neues `G2DConvexPolygon`–Objekt mit n Eckpunkten wird mit der Methode

`t.new2DObj(`*parent_id*`, @POLYGON, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

erzeugt.

B.5.7 G2DRectangle

Ein Objekt vom Typ `G2DRectangle` wird durch zwei diagonal gegenüberliegende Eckpunkte beschrieben. Es hat die Attribute *Color*, *PointSize*, *PointSmooth*, *LineWidth*, *LineStyle*, *FillStyle* und *PolygonMode*. Abhängig von *PolygonMode* wird jeweils nur eine Untermenge der Attribute verwendet.

Ein neues `G2DRectangle`–Objekt wird mit der Methode

`t.new2DObj(`*parent_id*`, @RECTANGLE, [[`x_0`, `y_0`], [`x_1`, `y_1`]])`

erzeugt, wobei das Rechteck die vier Eckpunkte $x_0; y_0$, $x_0; y_1$, $x_1; y_0$ und $x_1; y_1$ hat.

B.5.8 G2DText

Ein Objekt vom Typ `G2DText` besteht aus einem relativ zu einem Bezugspunkt positionierten ASCII–Text. Es hat die Attribute *Color*, *Text*, *Position*, *Height*, *AspectRatio* und *Alignment*.

Ein neues `G2DText`–Objekt wird mit der Methode

`t.new2DObj(`*parent_id*`, @TEXT, [`x`, `y`], `*text*`)`

erzeugt, wobei $x; y$ die Position und *text* eine Zeichenkette mit dem darzustellenden Text ist.

B.5.9 G2DArc

Ein Objekt vom Typ `G2DArc` stellt den Bogen eines Kreises dar, der durch seinen Mittelpunkt, den Radius, den Anfangs- und den Endwinkel beschrieben wird. Das Kreissegment wird in mathematisch positiver Drehrichtung vom Anfangs- zum Endwinkel gezeichnet. `G2DArc`–Objekte haben die Attribute *Color*, *LineWidth* und *LineStyle*.

Ein neues `G2DArc`–Objekt wird mit der Methode

`t.new2DObj(`*parent_id*`, @ARC, [`x_{center}`, `y_{center}`], `*radius*`, `*start*`, `*end*`)`

erzeugt. Mit der Methode

`t.new2DObj(`*parent_id*`, @CIRCLE, [`x_{center}`, `y_{center}`], `*radius*`)`

wird der Spezialfall eines Kreisbogens, bei dem $start = 0$ und $end = 2\pi$ ist, erzeugt.

B.5.10 G2DEllipse

Ein Objekt vom Typ `G2DEllipse` stellt eine Ellipse dar, die durch ihren Mittelpunkt, ihren Radius in x– und y–Richtung und ihren Rotationswinkel um den Mittelpunkt beschrieben wird.

Ein neues `G2DEllipse`–Objekt wird mit der Methode

`t.new2DObj(`*parent_id*`, @ELLIPSE, [`x_{center}`, `y_{center}`], [`x_{radius}`, `y_{radius}`], `*angle*`)`

erzeugt.

B.6 Attribute

In den folgenden Unterabschnitten werden die für 2D–Objekte unterstützten Attribute beschrieben.

B.6.1 *Color*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @COLOR, [`*red*`, `*green*`, `*blue*`])`

wird die Farbe des durch die Id *obj_id* spezifizierten Objekts auf den RGB–Wert *red*; *green*; *blue* gesetzt, wobei die drei Farbkomponenten als Gleitkommawerte im Intervall $[0.0, 1.0]$ anzugeben sind.

B.6.2 *PointSize*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @POINT_SIZE, `*point_size*`)`

wird die Größe eines Punktes auf den Gleitkommawert *point_size* gesetzt. Bei der Bildschirmdarstellung entspricht ein Wert von 1.0 einer Punkt–Größe von einem Pixel. Beim Ausdruck sollte 1.0 einer Größe von 1pt (1/72in) entsprechen.

Der Standardwert für die Punktgröße ist 1.0

B.6.3 *PointSmooth*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @POINT_SMOOTH, `*smooth*`)`

wird für Punkte, deren Größe nicht 1.0 ist, festgelegt, ob der Punkt als Quadrat ($smooth = 0$) oder als ausgefüllter Kreis mit geglättetem Rand ($smooth \neq 0$) dargestellt werden soll. Diese Einstellung ist nur für die Bildschirmdarstellung mit OpenGL relevant. Die Bildschirmdarstellung mit anderen Treibern oder die Druckausgabe können das *PointSmooth*–Flag ignorieren, wenn der Punkt immer als ausgefüllter Kreis dargestellt wird.

Der Standardwert für das *PointSmooth*–Flag ist 0.

B.6.4 *LineWidth*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @LINE_WIDTH, `*line_width*`)`

wird die Linienstärke auf *line_width* Pixel (Bildschirmdarstellung) oder Punkte (1pt = 1/72in) (Druck) gesetzt. Die Angabe von *line_width* erfolgt als Gleitkommawert.

Der Standardwert für die Linienstärke ist 1.0.

B.6.5 *LineStyle*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @LINE_STYLE,` *factor*`,` *pattern*`)`

wird die Linienart gesetzt. Dabei ist *pattern* ein Symbol, das die in Tabelle B.1 aufgeführten Werte annehmen kann.

Muster	Beschreibung
`@DEFAULT`	Es wird die voreingestellte Linienart verwendet.
`@SOLID`	Es wird eine durchgehende Linie gezeichnet.
`@DASHED`	Es wird eine gestrichelte Linie gezeichnet. Der Faktor *factor* bestimmt die Länge der dargestellten und nicht dargestellten Liniensegmente.
`@DOTTED`	Es wird eine gepunktete Linie gezeichnet. Der Faktor *factor* bestimmt den Abstand zwischen den Mittelpunkten zweier benachbarter Punkte.
`@DASH_DOTTED`	Es wird eine Strich–Punkt Linie gezeichnet. Der Faktor *factor* bestimmt die Länge des dargestellten Liniensegments und die halbe Länge der nicht dargestellten Liniensegmente.
`@DASH_DOUBLE_DOTTED`	Es wird eine Strich–2–Punkt Linie gezeichnet. Der Faktor *factor* bestimmt die Länge des dargestellten Liniensegments und ein Drittel der Länge der nicht dargestellten Segmente.
`@DASH_TRIPLE_DOTTED`	Es wird eine Strich–3–Punkt Linie gezeichnet. Der Faktor *factor* bestimmt die Länge des dargestellten Liniensegments und ein Viertel der Länge der nicht dargestellen Segmente.

Tabelle B.1: Linienarten

Der Faktor *factor* wird als Gleitkommawert angegeben. Seine Einheit ist bei der Bildschirmdarstellung ein Pixel und beim Druck ein Punkt (1pt = 1/72in).

Der Standardwert für *factor* ist 4.0 und für *pattern* `@DEFAULT`.

B.6.6 *FillStyle*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @FILL_STYLE,` *style*`)`

wird das Füllmuster für Polygone gesetzt. Dabei ist *style* ein Symbol, das die in Tabelle B.2 aufgeführten Werte annehmen kann.

Der horizontale Abstand zwischen schrägen oder vertikalen Linien im Füllmuster, oder der vertikale Abstand zwischen horizontalen Linien, sollte bei der Bildschirmausgabe acht Pixel und beim Druck acht Punkte (1pt = 1/72in) sein.

Standardmäßig wird ein Polygon vollständig gefüllt dargestellt.

Füllmuster	Beschreibung
`@LEFT_30`	diagonale Linien von links oben nach rechts unten in einem Winkel von 30 Grad zur Horizontalen
`@RIGHT_30`	diagonale Linien von links unten nach rechts oben in einem Winkel von 30 Grad zur Horizontalen
`@CROSS_30`	sich kreuzende diagonale Linien von links oben nach rechts unten und von links unten nach rechts oben, beide in einem Winkel von 30 Grad zur Horizontalen
`@LEFT_45`	diagonale Linien von links oben nach rechts unten in einem Winkel von 45 Grad zur Horizontalen
`@RIGHT_45`	diagonale Linien von links unten nach rechts oben in einem Winkel von 45 Grad zur Horizontalen
`@CROSS_45`	sich kreuzende diagonale Linien von links oben nach rechts unten und von links unten nach rechts oben, beide in einem Winkel von 45 Grad zur Horizontalen
`@H_LINES`	horizontale Linien
`@V_LINES`	vertikale Linien
`@CROSS`	gekreuzte horizontale und vertikale Linien

Tabelle B.2: Füllmuster

B.6.7 *PolygonMode*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @POLYGON_MODE,` *mode*`)`

wird über den Parameter *mode* für Polygone eingestellt, ob die Eckpunkte (*mode* = `@POINT`), die Begrenzungslinien (*mode* = `@LINE`) oder die Fläche (*mode* = `@FILL`) dargestellt werden soll. Die abhängig vom *PolygonMode* verwendeten Attribute sind in Tabelle B.3 aufgeführt.

PolygonMode	verwendete Attribute
`@POINT`	*Color, PointSize, PointSmooth*
`@LINE`	*Color, LineWidth, LineStyle*
`@FILL`	*Color, FillStyle*

Tabelle B.3: In Abhängigkeit vom *PolygonMode* verwendete Attribute

Der Standardwert für *PolygonMode* ist `@FILL`.

B.6.8 *Text*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @TEXT,` *text*`)`

wird für ein Objekt vom Typ `G2DText` der darzustellende Text *text* gesetzt. Das Argument *text* muß eine Zeichenkette sein, deren Zeichen aus dem ASCII–Zeichensatz stammen.

B.6.9 *Position*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @POSITION, [`*x*`, `*y*`])`

wird für ein Objekt vom Typ `G2DText` die Position (der Bezugspunkt) gesetzt.

B.6.10 *Height*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @HEIGHT, `*height*`)`

wird für ein Objekt vom Typ `G2DText` die Höhe eines Großbuchstabens ohne Unterlänge gesetzt.

Der Standardwert für die Texthöhe ist 0.1.

B.6.11 *AspectRatio*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @ASPECT_RATIO, `*ratio*`)`

wird für ein Objekt vom Typ `G2DText` die Streckung bzw. Stauchung in x–Richtung bestimmt. Der Gleitkommawert von *ratio* muß größer 0.0 sein. Beim Standardwert von 1.0 erscheint der Text in den normalen Proportionen, die durch den verwendeten Font vorgegeben werden. Bei Werten größer 1.0 wird er in x–Richtung gestreckt, bei Werten kleiner 1.0 und größer als 0.0 wird er in x–Richtung gestaucht.

B.6.12 *Alignment*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @ALIGNMENT, `*align*`)`

wird für ein Objekt vom Typ `G2DText` die Ausrichtung des Textes relativ zum Bezugspunkt bestimmt.

Wenn *width* die Breite des Textes ist, so wird die Verschiebung Δx der linken Seite des ersten Buchstabens auf der Grundlinie relativ zum Bezugspunkt wie folgt ermittelt:

$$\Delta x = -(alignment + 1.0) \times (width/2.0)$$

Der Standardwert für *align* ist -1.0, wodurch der Text links ausgerichtet erscheint.

B.6.13 *Pickable*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @PICKABLE,` *pickable*`)`

kann für jedes Objekt festgelegt werden, ob es im 2D–Mode bei der Ermittlung des mit der Maus zu selektierenden Objekts berücksichtigt werden soll. Ist das ganzzahlige Argument *pickable* Null, wird das entsprechende 2D–Objekt nicht berücksichtigt, Ist es ungleich Null, wird es berücksichtigt.

Ist das *pickable*–Flag eines Objekts vom Typ `G2DCompound` nicht gesetzt, so wird keines der Kindobjekte berücksichtigt. Ist es gesetzt, so ist das *pickable*–Flag des jeweiligen Kind–Objektes entscheidend.

Der Standardwert für *pickable* ist 1.

B.6.14 *Snapable*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @SNAPABLE,` *snapable*`)`

kann für jedes Objekt festgelegt werden, ob es bei der Ermittlung eines Fangpunktes berücksichtigt werden soll. Ist das ganzzahlige Argument *snapable* Null, wird das entsprechende 2D–Objekt nicht berücksichtigt. Ist es ungleich Null, wird es berücksichtigt.

Ist das *snapable*–Flag eines Objekts vom Typ `G2DCompound` nicht gesetzt, so wird keines der Kindobjekte berücksichtigt. Ist es gesetzt, so ist das *snapable*–Flag des jeweiligen Kind–Objektes entscheidend.

Der Standardwert für *snapable* ist 1.

B.6.15 *Exportable*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @EXPORTABLE,` *exportable*`)`

kann für jedes Objekt festgelegt werden, ob es beim Export in ein 2D–Vektorformat[2] mit exportiert werden soll oder nicht. Ist das ganzzahlige Argument *exportable* Null, wird das entsprechende 2D–Objekt nicht exportiert. Ist es ungleich Null, wird es exportiert.

Ist das *exportable*–Flag eines Objekts vom Typ `G2DCompound` nicht gesetzt, so wird auch keines der Kindobjekte exportiert. Ist es gesetzt, so ist das *exportable*–Flag des jeweiligen Kind–Objektes ausschlaggebend.

Der Standardwert für *exportable* ist 1.

[2]Da die Druckausgabe auch den Export in ein 2D–Vektorformat benutzt, hat das *exportable*–Flag auch auf die beim Ausruck erscheinenden Objekte Einfluß.

B.6.16 *Layer*

Mit dem Aufruf

`t.set2DObjAttr(`*obj_id*`, @LAYER,` *layer*`)`

kann für jedes Objekt der Layer, über den seine Sichtbarkeit gesteuert werden kann, festgelegt werden. Das Argument *layer* ist dabei ein Symbol, das ausschließlich aus Buchstaben, Ziffern und Unterstrich bestehen sollte.

Wurde der Layer für ein Objekt vom Typ `G2DCompound` gesetzt, so findet dieser Layer für alle direkten und indirekten Kindobjekte, für die nicht explizit ein Layer gesetzt wurde, Verwendung.

Anhang C

Das 2D-Vektor-Dateiformat

C.1 Einleitung

Das EasternGraphics Metafile (EGM) ist ein erweiterbares Dateiformat zur Speicherung grafischer und nicht–grafischer Daten. Auf Grund seiner Erweiterbarkeit erlaubt es auch die Einbindung von Daten in anderen Dateiformaten.

Das EGM ist so aufgebaut, daß ein EGM–Parser nicht alle Elemente eines EGM interpretieren können muß, um es bis zu seinem Ende einlesen zu können. Ihm nicht bekannte Elemente sollte er dabei im allgemeinen ignorieren können.

Das EGM unterstützt die hierarchische Speicherung von Daten. Dadurch ist es z.B. möglich, grafische 2D–Symbole sowohl auf oberster Ebene als auch eingebettet in ein Szenen–Element innerhalb eines EGM zu speichern. Der erste Fall wird verwendet, wenn das EGM lediglich ein einzelnens 2D–Symbol für die Verwendung im GF beschreibt. Der zweite Fall kann dann interessant werden, wenn im EGM eine komplette Szene beschrieben wird, die unter anderem auch nutzerdefinierte 2D–Symbole enthält.

Das EGM ist sowohl als Binärformat als auch als Textformat spezifiziert. Das Binärformat dient der effizienten Speicherung von Daten, während das Textformat in erster Linie während der Entwicklung Verwendung finden kann.

C.2 Datentypen

Dieser Abschnitt beschreibt die innerhalb eines EGM verwendeten Datentypen, insbesondere ihre Repräsentation im Binärformat und im Textformat des EGM.

Beide Formate, binär wie auch Text, sind so definiert, daß sie plattformunabhängig sind. Dadurch ist die Möglichkeit des problemlosen Datenaustausches zwischen verschiedenen Rechnerplattformen gegeben.

Alle numerischen Werte werden im Binärformat so abgespeichert, daß das höchstwertigste Byte zuerst kommt, gefolgt von allen anderen Bytes mit abnehmender Wertigkeit. Dies ist auch als „Network Byte Order“ oder „Big–Endian“ bekannt.

Der Zeichensatz für das Textformat muß eine Obermenge des ASCII–Zeichensatzes sein, was für die meisten, wenn nicht alle, ISO–8859–x–Zeichensätze der Fall ist. Um eine einfache Konvertierung zwischen Textformat und Binärformat zu ermöglichen, gilt die gleiche Einschränkung auch für den Datentyp *String* des Binärformats.

C.2.1 Einfache Datentypen

Byte

Ein *Byte* ist ein ganzzahliger 8 Bit Wert, der entweder als vorzeichenloser Wert im Bereich von 0 bis $2^8 - 1$ ($[0, 255]$) oder als vorzeichenbehafteter Wert im Zweierkomplement im Bereich von -2^7 bis $2^7 - 1$ ($[-128, 127]$) interpretiert wird. Werte vom Typ *Byte* werden im Binärformat des EGM, wie in Bild C.1 dargestellt, als ein einzelnes Byte gespeichert, wobei I_7 das höchstwertigste Bit und I_0 das niederwertigste Bit ist.

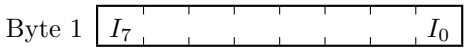

Abbildung C.1: Werte vom Typ *Byte*

Im Textformat erscheint ein *Byte* als eine Dezimal-, Oktal- oder Hexadezimalzahl, der optional ein Minus vorangestellt sein kann. Eine Oktalzahl wird durch eine führende `0` und eine Hexadezimalzahl durch ein führendes `0x` oder `0X` gekennzeichnet, wobei in Oktalzahlen nur die Ziffern `0` bis `7` und in Hexadezimalzahlen nur die Ziffern `0` bis `9`, `A` bis `F` und `a` bis `f` erlaubt sind.

Als Typangabe wird in dieser Spezifikation des EGM für vorzeichenlose *Byte*–Werte der Bezeichner UINT8 und für vorzeichenbehaftete *Byte*–Werte der Bezeichner INT8 verwendet.

Word

Ein *Word* ist ein ganzzahliger 16 Bit Wert, der entweder als vorzeichenloser Wert im Bereich von 0 bit $2^{16} - 1$ ($[0, 65535]$) oder als vorzeichenbehafteter Wert im Zweierkomplement im Bereich von -2^{15} bis $2^{15} - 1$ ($[-32768, 32767]$) interpretiert wird. Werte vom Typ *Word* werden im Binärformat des EGM, wie in Bild C.2 dargestellt, als zwei aufeinanderfolgende Bytes gespeichert, wobei I_{15} das höchstwertigste Bit und I_0 das niederwertigste Bit ist.

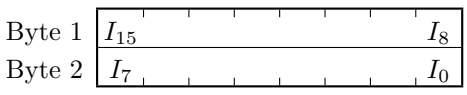

Abbildung C.2: Werte vom Typ *Word*

Im Textformat erscheint ein *Word* als eine Dezimal-, Oktal- oder Hexadezimalzahl, der optional ein Minus vorangestellt sein kann. Eine Oktalzahl wird durch eine führende `0` und eine Hexadezimalzahl durch ein führendes `0x` oder `0X` gekennzeichnet, wobei in Oktalzahlen nur die Ziffern `0` bis `7` und in Hexadezimalzahlen nur die Ziffern `0` bis `9`, `A` bis `F` und `a` bis `f` erlaubt sind.

Als Typangabe wird in dieser Spezifikation des EGM für vorzeichenlose *Word*–Werte der Bezeichner UINT16 und für vorzeichenbehaftete *Word*–Werte der Bezeichner INT16 verwendet.

Double Word

Ein *Double Word* ist ein ganzzahliger 32 Bit Wert, der entweder als vorzeichenloser Wert im Bereich von 0 bis $2^{32} - 1$ ($[0, 4294967295]$) oder als vorzeichenbehafteter Wert im Zweierkomplement im Bereich von -2^{31} bis $2^{31}-1$ ($[-2147483648, 2147483647]$) interpretiert wird. Werte vom Typ *Double Word* werden im Binärformat des EGM, wie in Bild C.3 dargestellt, als vier aufeinanderfolgende Bytes gespeichert, wobei I_{31} das höchstwertigste Bit und I_0 das niederwertigste Bit ist.

Byte 1	I_{31}	I_{24}
Byte 2	I_{23}	I_{16}
Byte 3	I_{15}	I_8
Byte 4	I_7	I_0

Abbildung C.3: Werte vom Typ *Double Word*

Im Textformat erscheint ein *Double Word* als eine Dezimal-, Oktal- oder Hexadezimalzahl, der optional ein Minus vorangestellt sein kann. Eine Oktalzahl wird durch eine führende `0` und eine Hexadezimalzahl durch ein führendes `0x` oder `0X` gekennzeichnet, wobei in Oktalzahlen nur die Ziffern `0` bis `7` und in Hexadezimalzahlen nur die Ziffern `0` bis `9`, `A` bis `F` und `a` bis `f` erlaubt sind.

Als Typangabe wird in dieser Spezifikation des EGM für vorzeichenlose *Double Word*–Werte der Bezeichner UINT32 und für vorzeichenbehaftete *Double Word*–Werte der Bezeichner INT32 verwendet.

Single Precision Floating Point

Werte vom Typ *Single Precision Floating Point* werden entsprechend dem IEEE 754 Standard dargestellt. Der Absolutwert liegt im Bereich von $1.17549435 \times 10^{-38}$ bis $3.40282347 \times 10^{38}$ mit mindestens 6 signifikanten Dezimalstellen in der Mantisse. Im Binärformat des EGM werden Gleitkommawerte einfacher Genauigkeit, wie in Bild C.4 dargestellt, als vier aufeinanderfolgende Bytes gespeichert.

Der Wert ist 0.0, wenn Exponent und Mantisse 0 sind. Andernfalls berechnet er sich nach $(-1)^s \times 1.f \times 2^{e-127}$. S ist das Vorzeichenbit, f die Mantisse ($F_1 \dots F_{23}$ mit F_1 als höchstwertigstem Bit) und e der Exponent ($E_7 \dots E_0$ mit E_7 als höchstwertigstem Bit).

Im Textformat besteht eine Gleitkommazahl einfacher Genauigkeit aus einem optionalen führenden Minus– oder Plus–Zeichen, einem ganzzahligen dezimalen Anteil, einem Dezimalpunkt, einem

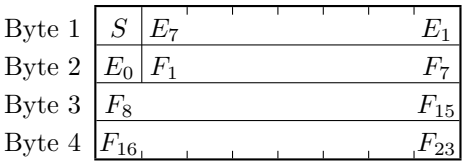

Abbildung C.4: Gleitkommawert einfacher Genauigkeit

gebrochenen dezimalen Anteil und einem optionalen Exponenten. Der Exponent besteht aus einem der Zeichen `e` oder `E`, gefolgt von einem optionalen Minus– oder Plus–Zeichen, gefolgt von einer ganzzahligen Dezimalzahl. Der ganzzahlige oder gebrochene Anteil können entfallen, jedoch nicht beide. Der Dezimalpunkt kann entfallen, wenn der gebrochene Anteil entfällt und ein Exponent vorhanden ist.

Als Typangabe für Gleitkommawerte einfacher Genauigkeit wird in dieser Spezifikation des EGM der Bezeichner FLOAT32 verwendet.

Double Precision Floating Point

Werte vom Typ *Double Precision Floating Point* werden entsprechend dem IEEE 754 Standard dargestellt. Der Absolutwert liegt im Bereich von $2.2250738585072014 \times 10^{-308}$ bis $1.7976931348623157 \times 10^{308}$ mit mindestens 15 signifikanten Dezimalstellen in der Mantisse. Im Binärformat des EGM werden Gleitkommawerte doppelter Genauigkeit, wie in Bild C.5 dargestellt, als acht aufeinanderfolgende Bytes gespeichert.

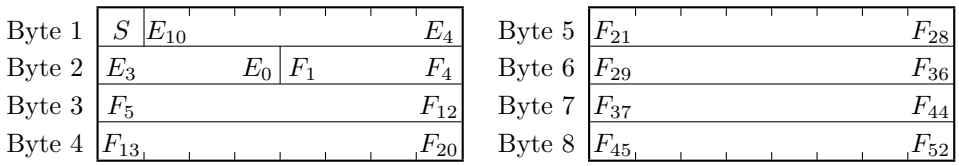

Abbildung C.5: Gleitkommawert doppelter Genauigkeit

Der Wert ist 0.0, wenn Exponent und Mantisse 0 sind. Andernfalls berechnet er sich nach $(-1)^s \times 1.f \times 2^{e-1023}$. S ist das Vorzeichenbit, f die Mantisse ($F_1 \dots F_{52}$ mit F_1 als höchstwertigstem Bit) und e der Exponent ($E_{10} \dots E_0$ mit E_{10} als höchstwertigstem Bit).

Die Darstellung einer Gleitkommazahl doppelter Genauigkeit im Textformat entspricht der Darstellung einer Gleitkommazahl einfacher Genauigkeit wie oben beschrieben.

Als Typangabe für Gleitkommawerte doppelter Genauigkeit wird in dieser Spezifikation des EGM der Bezeichner FLOAT64 verwendet.

Symbol

Ein Symbol ist eine Folge von Zeichen und wird im Binärformat des EGM, wie in Bild C.6 dargestellt, gespeichert. Die in den ersten beiden Bytes als vorzeichenloser Wert abgespeicherte Länge ($U_{15} \ldots U_0$ mit U_{15} als höchstwertigstem Bit) schließt weder sich selbst noch das das Symbol abschließende NUL–Zeichen mit ein.

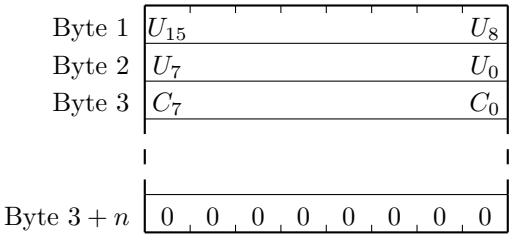

Abbildung C.6: Symbol

Im Textformat besteht ein Symbol aus einer Folge von ASCII–Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen keine Ziffer sein darf. Groß- und Kleinschreibung ist signifikant.

Als Typangabe für Symbole wird in dieser Spezifikation des EGM der Bezeichner SYMBOL verwendet.

String

Eine Zeichenkette wird im Binärformat des EGM, wie in Bild C.7 dargestellt, gespeichert. Die in den ersten beiden Bytes als vorzeichenloser Wert abgespeicherte Länge ($U_{15} \ldots U_0$ mit U_{15} als höchstwertigstem Bit) schließt weder sich selbst noch das die Zeichenkette abschließende NUL–Zeichen mit ein.

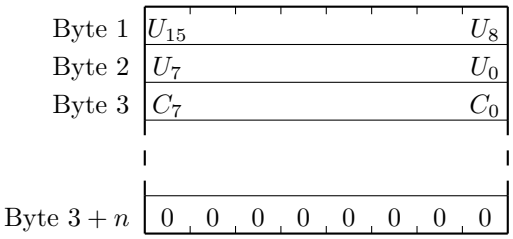

Abbildung C.7: Zeichenkette

Im Textformat des EGM werden Zeichenketten als Folge von beliebig vielen Zeichen (auch keinen), die von doppelten Anführungszeichen umgeben sind, dargestellt. Nicht druckbare Zeichen werden dabei durch Fluchtzeichenfolgen repräsentiert, von denen die in Tabelle C.1 aufgeführten Verwendung finden können. Es ist zu beachten, daß die in runden Klammern angegebene oktale

`\a`	Klingelzeichen	(\7)	`\\`	Gegenschrägstrich	(\)
`\b`	Rückschritt	(\8)	`\?`	Fragezeichen	(?)
`\f`	Seitenvorschub	(\14)	`\'`	Anführungszeichen	(')
`\n`	Zeilentrenner	(\12)	`\"`	doppeltes Anführungszeichen	(")
`\r`	Wagenrücklauf	(\15)	`\`*ooo*	oktale Zahl	
`\t`	Tabulatorzeichen	(\9)	`\x`*hh*	hexadezimale Zahl	
`\v`	Vertikaltabulator	(\13)			

Tabelle C.1: Fluchtzeichenfolgen für Zeichenketten

Kodierung von Plattform zu Plattform varieren kann. So ist z.B. unter MacOS$^{(R)}$ die Kodierung von `\n` und `\r` vertauscht.

Bei der Fluchtzeichenfolge `\`*ooo* steht *ooo* für eine Folge von ein bis drei oktalen Ziffern (0 ... 7) und bei `\x`*hh* steht *hh* für eine Folge von einer oder mehreren hexadezimalen Ziffern (0 ... 9, A ... F, a ... f). Es sollte vorzugsweise die Form `\`*ooo* mit drei oktalen Ziffern Verwendung finden, da nur so eine korrekte Kodierung des Zeichens gewährleistet werden kann, ohne auf das folgende Zeichen Rücksicht nehmen zu müssen.

Als Typangabe für Zeichenketten wird in dieser Spezifikation des EGM der Bezeichner STRING verwendet.

C.2.2 Strukturierte Datentypen

Strukturierte Datentypen bestehen aus einer Folge von einfachen Datentypen. Jeder strukturierte Datentyp wird durch eine Kombination aus Typklasse und Objekttyp beschrieben. Die Typklasse dient zur Klassifizierung von Objekttypen und die Objekttypen dienen zur Definition des Aufbaus und der Bedeutung strukturierter Datentypen. Eine Typklasse kann zum Beispiel alle für die Beschreibung von grafischen 2D–Primitiven definierten Objekttypen zusammenfassen, wobei jeder dieser Objekttypen die Struktur des jeweiligen 2D–Objekts innerhalb des EGM beschreibt.

Im Binärformat besteht jeder strukturierte Datentyp aus dem Strukturkopf und dem Strukturkörper. Der Strukturkopf ist acht Byte lang und enthält Informationen über den Typ der Struktur und über ihre Länge. Der Strukturkörper enthält die eigentlichen Daten. Innerhalb des Strukturkörpers ist jedes Datum relativ zum Strukturanfang an einem Vielfachen seiner eigenen Größe ausgerichtet, mit Ausnahme von Zeichenketten, die an einem Vielfachen von zwei ausgerichtet sind.

Byte 1	R_7								R_0
Byte 2	C_7								C_0
Byte 3	B	E	S	T_{12}					T_8
Byte 4	T_7								T_0

Byte 5	F_7	F_0
Byte 6	L_{23}	L_{16}
Byte 7	L_{15}	L_8
Byte 8	L_7	L_0

Abbildung C.8: Strukturkopf

In Bild C.8 ist der Aufbau des Strukturkopfes dargestellt.

Die Bits $R_7 \ldots R_0$ sind für zukünftige Verwendung reserviert und sollten auf 0 gesetzt sein.

Die Bits $C_7 \ldots C_0$ enthalten die Typklasse, die Bits $T_{12} \ldots T_0$ den Objekttyp. Die Objekttypen müssen innerhalb der Typklasse eindeutig sein.

Die B– und E–Bits dienen, wie in Abschnitt C.2.3 beschrieben, zur Kennzeichnung des Anfangs und Endes eines Verbundobjekts.

Das S–Bit zeigt an, ob Gleitkommaparameter in einfacher Genauigkeit (S–Bit ist gesetzt) oder doppelter Genauigkeit (S–Bit ist zurückgesetzt) vorliegen[1]. Nicht jeder Objekttyp, der Gleitkommaparameter hat, muß sowohl einfache und doppelte Genauigkeit unterstützen.

Die Bits $F_7 \ldots F_0$ haben keine vordefinierte Bedeutung und können abhängig vom Objekttyp zum Beispiel für Flag–Bits verwendet werden.

Die Bits $L_{23} \ldots L_0$ schließlich enthalten die komplette Länge der Struktur.

Der Strukturkopf selbst ist immer auf ein Vielfaches von acht relativ zum Anfang der Datei ausgerichtet und die gesamte Länge der Struktur ist ebenfalls ein Vielfaches von acht, wozu eine Struktur, wenn notwendig, an ihrem Ende mit Null–Bytes aufgefüllt wird. Damit ist sichergestellt, daß das EGM direkt in den Speicher gemappt werden kann und es beim direkten Zugriff auf einzelne Daten zu keinen Problemen mit der Ausrichtung der Daten und somit ggf. zu einem Bus–Fehler kommt.

Im Textformat besteht jeder strukturierte Datentyp aus einer oder mehreren Zeilen, wobei alle bis auf die letzte Zeile unmittelbar vor dem Zeilenendezeichen (oder den Zeilenendezeichen) einen Rückwärtsstrich (\) haben müssen. Diese Zeilen werden zu einem Datensatz zusammengefaßt, wobei die Rückwärtsstriche und das folgende Zeilenende entfernt werden.

Jede einzelne Zeile darf einschließlich des Zeilenendezeichens nicht länger als 2047 Zeichen sein. Die Anzahl der Zeichen pro Zeile ohne Zeilenendezeichen sollte folglich nicht größer als 2045 sein, da auf manchen Plattformen für die Kennzeichnung des Zeilenendes zwei Zeichen verwendet werden. Die Länge des gesamten Datensatzes ist theoretisch unbegrenzt.

Zeilen werden entweder durch `\x0A`, durch `\x0D` oder durch `\x0D\x0A` beendet.

Die einzelnen einfachen Daten innerhalb eines Datensatzes werden durch jeweils ein oder mehrere Trennzeichen getrennt, wobei als Trennzeichen das Leerzeichen (`\40`) und das Tabulatorzeichen (`\t`) Verwendung finden.

Am Anfang des Datensatzes stehen, durch ein oder mehrere Trennzeichen voneinander getrennt, Typklasse und Objekttyp, gefolgt von den als vorzeichenlose Dezimal-, Oktal- oder Hexadezimalzahl im Bereich von 0 bis 255 angegebenen Flags F_0 bis F_7, wobei F_7 das höchstwertigste Bit ist. Sowohl Typklasse als auch Objekttyp können als Bezeichner, bei denen Groß– und Kleinschreibung signifikant ist, oder als Dezimal-, Oktal- oder Hexadezimalzahlen angegeben werden. Der weitere Aufbau des Datensatzes ist durch den Objekttyp, dessen Bezeichner und Kodierung nur innerhalb der Typklasse eindeutig sein müssen, definiert.

Eine Oktalzahl wird durch eine führende `0` und eine Hexadezimalzahl durch ein führendes `0x` oder `0X` gekennzeichnet, wobei in Oktalzahlen nur die Ziffern `0` bis `7` und in Hexadezimalzahlen nur die Ziffern `0` bis `9`, `A` bis `F` und `a` bis `f` erlaubt sind.

[1] Bei Typangaben, bei denen abhängig vom S–Bit entweder FLOAT32 oder FLOAT64 verwendet wird, wird nur FLOAT geschrieben.

C.2.3 Verbundtypen

Verbundtypen bestehen aus einer Folge von strukturierten Datentypen, die durch ein *Begin*– und ein *End*–Objekt des gleichen strukturierten Datentyps eingeschlossen werden. Diese *Begin*– und *End*–Objekte müssen immer paarweise auftreten.

Im Binärformat des EGM ist das *Begin*–Objekt durch ein gesetztes *B*–Bit im Strukturkopf gekennzeichnet, und das *End*–Objekt dementsprechend durch ein gesetztes *E*–Bit.

Im Textformat des EGM beginnt der Datensatz des *Begin*–Objekts mit dem Bezeichner `begin`, dem, getrennt durch ein oder mehrere Trennzeichen, die Typklasse folgt. Der Datensatz des *End*–Objekts beginnt analog mit dem Bezeichner `end`, gefolgt von einem oder mehreren Trennzeichen und der Typklasse. Statt `begin` kann auch ein einzelnes Plus–Zeichen (`+`), statt `end` ein einzelnes Minus–Zeichen (`-`) stehen.

C.3 Dateikopf

Das Binärformat des EGM beginnt mit der folgenden Struktur:

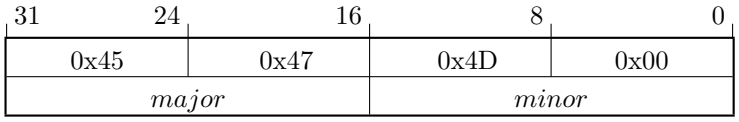

Abbildung C.9: Kopf des binärem EGM

Dabei ist *major* die Hauptversionsnummer und *minor* die Unterversionsnummer. Die in diesem Dokument beschriebene Version des EGM ist 1.0 ($major = 1; minor = 0$).

Im Textformat des EGM enthält die erste Zeile die Bezeichner `EGM` und `version`, wobei `EGM` unmittelbar am Zeilenanfang steht und `version` von `EGM` durch ein Leerzeichen getrennt ist. Es folgen, getrennt durch die üblichen Trennzeichen Hauptversionsnummer und Unterversionsnummer. Die erste Zeile des EGM in der Version 1.0 ist:

```
EGM version 1 0
```

C.4 Allgemeine strukturierte Datentypen

In diesem Abschnitt sind allgemeine strukturierte Datentypen beschrieben, die keinen Bezug zu konkreten Typklassen haben. Diese Typen haben die Typklasse 1 mit dem Bezeichner `common`.

C.4.1 Kommentar

Typklasse: 1 / `common`

Objekttyp: 1 / `comment`

Parameter:

Offset	Typ	Parameter
8	STRING	*comment*
roundup$(11 + len, 8)$	Strukturende	

Kommentare werden beim Einlesen ignoriert.

Als Spezialfall können im Textformat Kommentare aus einem Datensatz bestehen, der mit einem Doppelkreuz (`#`) beginnt, dem der eigentliche Kommentar unmittelbar folgt, ggf. von dem Doppelkreuz durch ein oder mehrere Trennzeichen getrennt.

C.4.2 EGM–Typ

Typklasse: 1 / `common`

Objekttyp: 2 / `egm_type`

Parameter:

Offset	Typ	Parameter
8	STRING	*egmtype*
roundup$(11 + len, 8)$	Strukturende	

Unmittelbar nach dem in Abschnitt C.3 beschriebenen Dateikopf kann ein Objekt vom Typ *EGM-Type* folgen. Die Zeichenkette *egmtype* beschreibt den Typ des EGM. Derzeit sind die in Tabelle C.2 aufgeführten Typen definiert.

Bezeichner	Beschreibung
`x2DSYMBOL`	Das EGM beschreibt ein 2D–Symbol. Es sollte nur Objekte der Typklassen `common` und `gr2dobj` enthalten. Objekte anderer Typklassen werden beim Einlesen ignoriert. EGM–Dateien vom Typ `x2DSYMBOL` enthalten genau ein 2D–Objekt. Wenn dieses aus mehreren primitiven 2D–Objekten besteht, müssen diese durch ein *Compound*–Objekt gekapselt werden.

Tabelle C.2: EGM–Typen

C.5 Grafische 2D–Objekte

Die grafischen 2D–Objekte werden zu einer Typklasse mit der Nummer 2 und dem Bezeichner `gr2dobj` zusammengefaßt.

Die 2D–Objekte werden in einem x/y–Koordinatensystem beschrieben, wobei die x–Achse nach rechts und die y–Achse nach oben zeigt. Winkelangaben erfolgen im Bogenmaß, sind mathematisch

positiv (entgegen dem Uhrzeigersinn) und, wenn nicht anders angegeben, relativ zur positiven x–Achse.

Alle Koordinaten werden als Gleitkommawerte mit einfacher oder doppelter Genauigkeit angegeben. Die verwendete Genauigkeit wird in jedem einzelnen 2D–Objekt durch das S–Bit des Strukturkopfes spezifiziert.

C.5.1 Compound

Grafische 2D–Objekte werden durch den Verbundtyp *Compound* verschachtelt. Die Verschachtelung kann auch rekursiv angewendet werden.

Ein *Compound*–Objekt hat keine geometrische Repräsentation. Üblicherweise schließt es mindestens ein Objekt der Typklasse `gr2dobj` ein. Andere eingeschlossene Objekte sind nicht erlaubt.

Das *Compound*–Objekt kann optional eine geometrische Transformation, die auf die eingeschlossenen Objekte angewendet werden soll, enthalten. Die Transformation wird entweder als Rotation der eingeschlossenen Objekte mit anschließender Translation oder als 3×3 Transformationsmatrix plus optionaler inverser Transformationsmatrix angegeben.

Typklasse: 2 / `gr2dobj`

Objekttyp: 1 / `compound`

Flags:
$F_1F_0 = 00$: keine Transformation
$F_1F_0 = 01$: Rotation und Translation
$F_1F_0 = 10$: Transformationsmatrix
$F_1F_0 = 11$: Transformationsmatrix und inverse Transformationsmatrix

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
mit $F_1F_0 = 01$:			
8	8	FLOAT	α
12	16	FLOAT	x_{offs}, y_{offs}
24	32	Strukturende	
mit $F_1F_0 = 10$:			
8	8	FLOAT	$mat_{2\times3}$
32	56	Strukturende	
mit $F_1F_0 = 11$:			
8	8	FLOAT	$mat_{2\times3}$
32	56	FLOAT	$mat^{-1}_{2\times3}$
56	104	Strukturende	

Erfolgt die Angabe der Transformation über Rotation und Translation, so werden die Koordinaten der eingeschlossenen Objekte im Koordinatensystem des *Compound*–Objekts wie folgt berechnet:

$$x' = x\cos\alpha - y\sin\alpha + x_{offs}$$
$$y' = x\sin\alpha + y\cos\alpha + y_{offs}$$

Wenn die Angabe der Transformation über die Transformationsmatrix erfolgt, so berechnen sich die Koordinaten der eingeschlossenen Objekte im Koordinatensystem des *Compound*–Objekts wie folgt:

$$x' = m_{11}x + m_{12}y + m_{13}$$
$$y' = m_{21}x + m_{22}y + m_{23}$$

Die Transformationsmatrizen, wie auch ihre Inversen, werden im EGM zeilenweise, beginnend mit dem Element in der linken oberen Ecke (m_{11}), abgespeichert. Die dritte Zeile wird nicht mit abgespeichert, da sie immer $0.0, 0.0, 1.0$ ist.

Die inverse Matrix kann im EGM mit enthalten sein, um sie nicht während des Lesens des EGM berechnen zu müssen.

C.5.2 Grafische Primitive

In diesem Abschnitt sind alle die 2D–Objekte beschrieben, die über eine grafische Repräsentation verfügen.

Die Objektdefinitionen enthalten nur die geometrischen Informationen wie Punktkoordinaten und Winkel. Attribute, wie die Farbe, werden in einem *Attribut–Set* gespeichert und in diesem über spezielle Attribut–Objekte gesetzt. Die grafischen Primitiven verwenden die zum Zeitpunkt ihres Auftretens in dem Attribut–Set für sie relevanten Attribute, welche für jede einzelne grafische Primitive unter „Attribute:“ aufgelistet sind.

Lines

Typklasse: 2 / `gr2dobj`

Objekttyp: 256 / `lines`

Attribute: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	n
12	16	FLOAT	$x1_0, y1_0, x2_0, y2_0$
$12 + (n-1) \times 16$	$16 + (n-1) \times 32$	FLOAT	$x1_{n-1}, y1_{n-1},$ $x2_{n-1}, y2_{n-1}$
$16 + n \times 16$	$16 + n \times 32$	Strukturende	

Objekte vom Typ *Lines* stellen ein oder mehrere getrennte Liniensegmente dar. Der Parameter n spezifiziert die Anzahl der Liniensegmente. Sein Wert muß größer oder gleich 1 sein. Jedes einzelne Liniensegment beginnt bei $x1_i, y1_i$ und endet bei $x2_i, y2_i$.

Polyline

Typklasse: 2 / `gr2dobj`

Objekttyp: 257 / `polyline`

Attribute: *Color, LineWidth, LineStyle*

Flags: $F_0 = 0$: offen
$F_0 = 1$: geschlossen

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n-1) \times 8$	$16 + (n-1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	Strukturende	

Objekte vom Typ *Polyline* stellen, abhängig vom Flag F_0, einen offenen oder geschlossenen Linienzug dar. Der Parameter n ist um 1 größer als die Anzahl der Liniensegmente. Er muß größer oder gleich 2 sein. Wenn das Flag F_0 gesetzt ist, wird der letzte Punkt (x_{n-1}, y_{n-1}) mit dem ersten Punkt (x_0, y_0) verbunden.

Points

Typklasse: 2 / `gr2dobj`

Objekttyp: 258 / `points`

Attribute: *Color, PointSize, PointType*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n-1) \times 8$	$16 + (n-1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	Strukturende	

Objekte vom Typ *Points* stellen einen oder mehrere Punkte dar. Der Parameter n spezifiziert die Anzahl der Punkte. Sein Wert muß größer oder gleich 1 sein.

Circle

Typklasse: 2 / `gr2dobj`

Objekttyp: 259 / `circle`

Attribute: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	r
24	32	Strukturende	

Objekte vom Typ *Circle* stellen einen Kreis mit dem Radius r dar, dessen Mittelpunkt durch x_{center}, y_{center} bestimmt wird. Der Radius r muß größer als 0.0 sein.

Arc

Typklasse: 2 / `gr2dobj`

Objekttyp: 260 / `arc`

Attribute: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	r
20	32	FLOAT	$\alpha_{start}, \alpha_{end}$
24	48	Strukturende	

Objekte vom Typ *Arc* stellen einen Kreisbogen mit dem Radius r dar, dessen Mittelpunkt durch x_{center}, y_{center} bestimmt wird. Der Kreisbogen wird vom Winkel α_{start} zum Winkel α_{end} in mathematisch positiver Richtung gezeichnet.

Ellipse

Typklasse: 2 / `gr2dobj`

Objekttyp: 261 / `ellipse`

Attribute: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	x_{radius}, y_{radius}
24	40	FLOAT	α
32	48	Strukturende	

Objekte vom Typ *Ellipse* stellen eine Ellipse dar, deren Mittelpunkt durch x_{center}, y_{center} bestimmt wird. Der Radius der nicht gedrehten Ellipse in Richtung der x–Achse ist x_{radius}, der Radius in Richtung der y–Achse y_{radius}. Der Rotationswinkel der Ellipse um ihren Mittelpunkt ist α.

Text

Typklasse: 2 / `gr2dobj`

Objekttyp: 262 / `text`

Attribute: *Color, Font, FontHeight, FontAspectRatio, FontAlignment*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	x_{origin}, y_{origin}
16	24	FLOAT	α
20	32	FLOAT	*width*
20	32	STRING	*text*
roundup$(23 + len, 8)$	roundup$(35 + len, 8)$	Strukturende	

Objekte vom Typ *Text* stellen den Text *text* dar, dessen Grundlinie durch den Bezugspunkt x_{origin}, y_{origin} verläuft und um den Bezungspunkt um den Winkel α gedreht ist. Der Parame-

ter *width* gibt die Breite des nicht gedrehten Textes an. Die Position der linken Seite des ersten Buchstabens auf der Grundlinie wird, wie in Abschnitt C.5.3 beschrieben, ermittelt.

Ein System, das die gleichen Fonts wie EGR GF benutzt, kann beim Einlesen den Parameter *width* ignorieren, da die Breite des Textes durch das Attribut *FontAspectRatio* bestimmt wird. Andere Systeme müssen das Attribut *FontAspectRatio* ignorieren und statt dessen den Text an die im Parameter *width* angegebene Breite anpassen.

Convex Polygon

Typklasse: 2 / `gr2dobj`

Objekttyp: 263 / `cvx_polygon`

Attribute: *Color, FillStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n-1) \times 8$	$16 + (n-1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	Strukturende	

Objekte vom Typ *Convex Polygon* stellen ein konvexes Polygon dar. Bei einem konvexen Polygon schneiden sich keine Kanten, und alle Innenwinkel sind kleiner oder gleich π.

Der Parameter n, der die Anzahl der Eckpunkte des Polygons enthält, muß größer als 2 sein.

C.5.3 Attribute

Wie in Abschnitt C.5.2 beschrieben, enthalten die grafischen 2D–Primitiven keine Attribute, sondern nur reine Geometrie–Informationen. Sie benutzten die für sie relevanten Attribute, die zum Zeitpunkt ihres Auftretens im EGM im aktuellen Attribut–Set gesetzt sind.

Da das EGM die hierarchische Strukturierung von Daten in Form eines Baumes erlaubt, muß es auch eine Hierarchie von Attribut–Sets unterstützen. Diese Hierarchie wird während des Lesens des EGM in Baumform aufgebaut, wobei zu einem Zeitpunkt nur alle Knoten des Baumes, die sich auf dem Pfad von der Wurzel zum aktuellen Blatt befinden, existieren. Dies wird durch einen Stack von Attribut–Sets realisiert.

Um den Stack der Attribut–Sets mit der Hierarchie des EGM konsistent zu halten, entfernt jedes *End*–Objekt eines Verbundtyps die notwendige Anzahl von Attribut–Sets vom Stack, bis die Anzahl der Attribut–Sets auf dem Stack identisch ist mit der Anzahl, die das korresponierende *Begin*–Objekt auf dem Stack vorgefunden hat. Ebenso tritt ein Fehler auf, wenn ein Attribut–Set vom Stack entfernt wird, so daß die Anzahl der Attribut–Sets auf dem Stack geringer wird als die Anzahl der Attribut–Sets, die das *Begin*–Objekt des innersten Verbundtyps auf dem Stack vorgefunden hat.

Attributwerte

Die bei den grafischen 2D–Primitiven angegebenen Koordinatenwerte können in der Regel (d.h., wenn keine Skalierung stattfindet) als Meter interpretiert werden. Die Abmessungen bei der Darstellung auf dem Bildschirm, dem Drucker oder dem Plotter hängen von dem verwendeten Maßstab ab.

Viele Attributwerte für die grafischen 2D–Objekte werden demgegenüber unabhängig vom Maßstab angegeben, da dies einerseits den Fähigkeiten üblicher Ausgabegeräte entgegenkommt und andererseits eine Skalierung mit dem Maßstab oftmals nicht erwünscht ist. Die in diesem Fall zugrundegelegte Einheit ist der Punkt. Für die Größe eines Punktes gelten die folgenden Beziehungen:

$1\text{pt} = 0.0352\overline{7}\text{cm}$	$1\text{pt} = 0.352\overline{7}\text{mm}$	$1\text{pt} = 0.013\overline{8}\text{in}$
$1\text{cm} = 28.346457\text{pt}$	$1\text{mm} = 2.8346457\text{pt}$	$1\text{in} = 72\text{pt}$

Diese Definition eines Punktes ist kompatibel mit der eines PostScript Punktes, unterscheidet sich aber leicht von der im Buchdruck verwendeten Defintion, bei der 1in = 72.27pt und 1in = 72bp gilt, wobei bp für *Big Point* steht.

Bei der Ausgabe auf Drucker oder Plotter sollte die Größe eines Punktes möglichst genau eingehalten werden. Bei der Ausgabe auf dem Bildschirm ist es akzeptabel, daß bei einer üblichen Auflösung von 75 bis 100 dpi ein Punkt als ein Pixel dargestellt wird.

Push Attrib

Typklasse: 2 / `gr2dobj`

Objekttyp: 512 / `push_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	Strukturende	

Wird ein *PushAttrib*–Objekt gelesen, so wird der aktuelle Zustand des Attribut–Sets auf den Attribut–Stack gelegt.

Pop Attrib

Typklasse: 2 / `gr2dobj`

Objekttyp: 513 / `pop_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	Strukturende	

Wenn ein *PopAttrib*–Objekt gelesen wird, so wird das oberste Attribut–Set in das aktuelle Attribut–Set kopiert und vom Stack entfernt. Ein Fehler tritt auf, wenn anschließend die Anzahl der Attribut–Sets auf dem Stack kleiner ist als zu dem Zeitpunkt, als das *Begin*–Objekt des innersten Verbundtyps gelesen wurde.

Init Attrib

Typklasse: 2 / `gr2dobj`

Objekttyp: 514 / `init_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	Strukturende	

Wenn ein *InitAttrib*–Objekt gelesen wird, so wird das aktuelle Attribut–Set auf die Standardwerte zurückgestellt. Die Standardwerte sind bei den einzelnen Attributen angegeben.

Color

Typklasse: 2 / `gr2dobj`

Objekttyp: 515 / `color`

Standard: $r/g/b = 0/0/0$

Parameter:

Offset	Typ	Parameter
8	UINT16	*red*
10	UINT16	*green*
12	UINT16	*blue*
16	Strukturende	

Die Angabe von Farben erfolgt im RGB–System, wobei für jede Farbkomponente ein vorzeichenloser ganzzahliger Wert im Bereich von 0 bis 65535 angegeben wird. Dabei entspricht 0 minimaler und 65535 maximaler Intensität.

Line Width

Typklasse: 2 / `gr2dobj`

Objekttyp: 516 / `line_width`

Standard: 1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	*linewidth*
16	16	Strukturende	

Objekte vom Typ *LineWidth* setzen im aktuellen Attribut–Set das Attribut für die Linienstärke. Die Angabe der Linienstärke erfolgt in Punkten (pt). Der Wert muß größer als 0.0 sein. Ungültige Werte werden als 1.0 interpretiert.

Line Style

Typklasse: 2 / `gr2dobj`

Objekttyp: 517 / `line_style`

Standard: -1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	*linestyle*
12	16	FLOAT	$factor$
16	24	Strukturende	

Objekte vom Typ *LineStyle* setzen im aktuellen Attribut–Set den Linientyp.

Für der Parameter *linestyle* sind die folgenden Werte vordefiniert:

Konstante	Linientyp
-1	Standardwert
0	durchgehende Linie
1	gestrichelte Linie
2	gepunktete Linie
3	Strich–Punkt–Linie
4	Strich–Punkt–Punkt–Linie
5	Strich–Punkt–Punkt–Punkt–Linie

Tabelle C.3: Vordefinierte Linientypen

Wenn der *LineStyle* -1 ist, findet der vom übergeorneten Objekt eingestellte *LineStyle* Verwendung.

Der Parameter wird in Punkten (pt) angegeben und bestimmt, wie die Linie gestreckt wird. Abhängig vom Linientyp beeinflußt er die Darstellung der Linie wie folgt:

Linientyp	Bedeutung vom Faktor $factor$
gestrichelt	Länge der dargestellten und nicht dargestellten Segmente
gepunktet	Abstand zwischen den Mittelpunkten zweier benachbarter Punkte
Strich–Punkt	Länge des dargestellten Liniensegments und halbe Länge der nicht dargestellten Liniensegmente
Strich–2–Punkt	Länge des Liniensegments und ein Drittel der nicht dargestellten Liniensegmente
Strich–3–Punkt	Länge des Liniensegments und ein Viertel der nicht dargestellten Liniensegmente

Tabelle C.4: Auswirkung des Faktors auf den Linientyp

Point Size

Typklasse: 2 / `gr2dobj`

Objekttyp: 518 / `point_size`

Standard : 0.1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	*pointsize*
16	16	Strukturende	

Objekte des Typs *PointSize* setzen im aktuellen Attribut–Set die Größe eines Punktes. Die Punktgröße findet nur dann Verwendung, wenn es sich beim eingestellten Punkttyp um einen Vektor–Punkt handelt. In diesem Fall werden die Vertices des Punktes, wie in Tabelle C.6 beschrieben, berechnet, wobei die Variable d die mittels *PointSize* eingestellte Punktgröße ist.

Point Type

Typklasse: 2 / `gr2dobj`

Objekttyp: 522 / `point_style`

Standard: 0xffffffff

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	UINT32	*pointstyle*
16	16	Strukturende	

Bei Punkten wird zwischen Bitmap–Punkten und Vektor–Punkten unterschieden. Bitmap–Punkte sind immer gleich ausgerichtet und unabhängig von der Skalierung und der eingestellten Punktgröße immer gleich groß. Die Größe und Ausrichtung von Vektor–Punkten wird durch die eingestellte Punktgröße sowie ggf. durch die Transformation eines übergeordneten Verbundobjektes bestimmt.

Die Angabe von Punkten erfolgt durch eine Bitmaske, so daß sowohl verschiedene Bitmap–Punkte wie auch verschiedene Vektorpunkte jeweils untereinander kombiniert werden können[2]. Die Kombinationsmöglichkeiten bei Vektor–Punkten sind beliebig, bei Bitmap–Punkten beschränken sie sich auf Punkttypen verschiedener Klassen.

Die Tabellen C.5 und C.6 enthalten die Konstanten für die Spezifikation von Bitmap– und Vektor–Punkten. Die Tabelle C.6 enthält die Vorschrift für die Berechnung der Vertices der Punkte, wobei die Variable d die mittels *PointSize* eingestellte Punktgröße ist.

Font

Derzeit werden keine verschiedenen Fonts unterstützt.

[2] D.h., daß Bitmap– und Vektor–Punkte nicht kombiniert werden können.

Konstante	Durchmesser	Konstante	Durchmesser
-1	Standardwert		
gefüllter Kreis:		Kreuz:	
0x40000001	1 Pixel	0x40000008	5 Pixel
0x40000002	3 Pixel	0x40000010	10 Pixel
0x40000003	5 Pixel	0x40000018	15 Pixel
0x40000004	7 Pixel	0x40000020	20 Pixel
0x40000005	9 Pixel	0x40000028	25 Pixel
0x40000006	11 Pixel	0x40000030	30 Pixel
0x40000007	13 Pixel	0x40000038	40 Pixel
diagonales Kreuz:		Kreis:	
0x40000040	5 Pixel	0x40000200	5 Pixel
0x40000080	10 Pixel	0x40000400	10 Pixel
0x400000c0	15 Pixel	0x40000600	15 Pixel
0x40000100	20 Pixel	0x40000800	20 Pixel
0x40000140	25 Pixel	0x40000a00	25 Pixel
0x40000180	30 Pixel	0x40000c00	30 Pixel
0x400001c0	40 Pixel	0x40000e00	40 Pixel
Quadrat:			
0x40001000	5 Pixel	0x40002000	10 Pixel
0x40003000	15 Pixel	0x40004000	20 Pixel
0x40005000	25 Pixel	0x40006000	30 Pixel
0x40007000	40 Pixel		

Tabelle C.5: Typen von Bitmap-Punkten und ihre Konstanten

Font Height

Typklasse: 2 / `gr2dobj`

Objekttyp: 519 / `font_height`

Standard: 0.1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	*fontheight*
16	16	Strukturende	

Objekte des Typs *FontHeight* setzen im aktuellen Attribut–Set die Höhe des Fonts, gemessen von der Grundlinie bis zur Oberkante normaler Großbuchstaben. Die Höhenangabe für den Font berücksichtigt somit weder Über– noch Unterlängen.

Die Angabe der Font–Höhe erfolgt **nicht** in Punkten.

Konstante	Typ
0x00000001	kleines Kreuz: $[(-0.5 \times d, 0), (0.5 \times d, 0)], [(0, -0.5 \times d), (0, 0.5 \times d)]$
0x00000002	großes Kreuz: $[(-\sqrt{0.5} \times d, 0), (\sqrt{0.5} \times d, 0)], [(0, -\sqrt{0.5} \times d), (0, \sqrt{0.5} \times d)]$
0x00000004	kleines um 45° gedrehtes Kreuz: $[(-\sqrt{0.125} \times d, -\sqrt{0.125} \times d), (\sqrt{0.125} \times d, \sqrt{0.125} \times d)], [(-\sqrt{0.125} \times d, \sqrt{0.125} \times d), (\sqrt{0.125} \times d, -\sqrt{0.125} \times d)]$
0x00000008	großes um 45° gedrehtes Kreuz: $[(-0.5 \times d, -0.5 \times d), (0.5 \times d, 0.5 \times d)], [(-0.5 \times d, 0.5 \times d), (0.5 \times d, -0.5 \times d)]$
0x00000010	Kreis, Durchmesser ist d, Mittelpunkt liegt bei $(0, 0)$
0x00000020	Quadrat: $[(0.5 \times d, 0.5 \times d), (-0.5 \times d, 0.5 \times d), (-0.5 \times d, -0.5 \times d), (0.5 \times d, -0.5 \times d)]$
0x00000040	um 45° gedrehtes Quadrat: $[(0.5 \times d, 0), (0, 0.5 \times d), (-0.5 \times d, 0), (0, -0.5 \times d)]$
0x00000080	gleichschenkliges Dreieck, Spitze nach rechts: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0.5 \times d, 0), (c, 0.5 \times d), (c, -0.5 \times d)]$
0x00000100	gleichschenkliges Dreieck, Spitze nach oben: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0, 0.5 \times d), (-0.5 \times d, c), (0.5 \times d, c)]$
0x00000200	gleichschenkliges Dreieck, Spitze nach links: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(-0.5 \times d, 0), (c, -0.5 \times d), (c, 0.5 \times d)]$
0x00000400	gleichschenkliges Dreieck, Spitze nach unten: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0, -0.5 \times d), (0.5 \times d, c), (-0.5 \times d, c)]$

Tabelle C.6: Typen von Vektor-Punkten und ihre Konstanten

Font Aspect Ratio

Typklasse: 2 / `gr2dobj`

Objekttyp: 520 / `font_aspect_ratio`

Standard: 1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	*aspectratio*
16	16	Strukturende	

Objekte des Typs *FontAspectRatio* setzen im aktuellen Attribut–Set das Seitenverhältnis des Fonts, welches bestimmt, ob Text in der Breite gestaucht ($aspectratio < 1.0$), gestreckt ($aspectratio > 1.0$) oder normal ($aspectratio = 1.0$) erscheint.

Font Alignment

Typklasse: 2 / `gr2dobj`

Objekttyp: 521 / `font_alignment`

Standard: -1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	FLOAT	*alignment*
16	16	Strukturende	

Objekte des Typs *FontAlignment* setzen im aktuellen Attribut–Set die horizontale Ausrichtung des Texts relativ zu seinem Ursprung.

Wenn *width* die Breite des Textes ist, so wird die Verschiebung Δx der linken Seite des ersten Buchstabens auf der Grundlinie relativ zum Bezugspunkt wie folgt ermittelt:

$$\Delta x = -(alignment + 1.0) \times (width/2.0)$$

Layer

Typklasse: 2 / `gr2dobj`

Objekttyp: 523 / `layer`

Standard: `DEFAULT`

Parameter

Offset ($S = 1$)	Offset ($S = 0$)	Typ	Parameter
8	8	SYMBOL	*layer*
roundup$(11 + len, 8)$	roundup$(11 + len, 8)$	Strukturende	

Objekte des Typs *Layer* setztn im aktuellen Attribut–Set den Layer. Der Name des Layers sollte ausschließlich aus ASCII–Buchstaben, Ziffern und dem Unterstrich bestehen und nicht mit einer Ziffer beginnen. Groß- und Kleinschreibung ist signifikant.

Der Layer wird benutzt, um die Sichtbarkeit von Objekten zu steuern. Grundsätzlich ist es möglich, alle grafischen Objekte, die einem Layer angehören, gemeinsam ein- und auszublenden. Es findet keine Zuordnung von Attributen über den Layer an Objekte statt.

In einer hierarchischen Anordnung von Objekten erben Objekte, die zum Standardlayer gehören[3], den Layer vom übergeordneten Objekt, bei dem es sich selber wieder um den Standardlayer handeln kann. Ist dies der Fall, wird das entsprechende Objekt immer dargestellt[4].

[3] Der Name des Standardlayers ist `DEFAULT`.

[4] Der Standardlayer kann nicht ausgeblendet werden.

Anhang D

Externe Datenformate

OFML definiert die nachfolgend beschriebenen externen Datenformate. Die entsprechenden Dateien befinden sich in einem bibliotheksspezifischen Verzeichnis oder in einem bibliotheksspezifischen Archiv. Allgemeine Materialdefinitionen befinden sich in einem globalen Verzeichnis mit dem relativen Pfad *data/material*. Die vordefinierten Fonts befinden sich in einem globalen Verzeichnis mit dem relativen Pfad *data/font*.

Externe Daten sind vollständig zu qualifizieren. Beispielsweise muß die angenommene Text-Ressource *@collision* aus dem Paket *::ofml::xoi* beim Aufruf wie folgt qualifiziert werden:

"::ofml::xoi::@collision"

Ist eine Text-Ressource nicht qualifiziert, so wird die Ressourcen-Datei zunächst im Paket des unmittelbaren Typs der Instanz gesucht, für das die Text-Ressource aufgelöst werden soll[1]. Kann die Ressourcen-Datei nicht in diesem Paket gefunden werden, so wird weiter in den Paketen der Supertypen gesucht.

D.1 Geometrien

- Geometriebeschreibungsdateien definieren polygonale Geometrien, die direkt in *OFML* eingelesen werden können.
- **Namensgebung:** Der Name von geometrischen Definitionsdateien ergibt sich aus dem Namen der Geometrie, wie er in *OiImport* verwendet wird, ohne Pfad und Erweiterung. Dabei sind nur ASCII-Zeichen erlaubt. Allerdings sind dabei keine Leerzeichen erlaubt. Die Erweiterung richtet sich nach der jeweiligen Datei. Erlaubte Erweiterungen sind:
  - *geo* – polygonale Geometrien (OFF-Format)
    Dabei müssen Polygone einfach, planar, convex und im Uhrzeigersinn definiert sein.

[1] z.B. die Instanz, die eine Message mittels *oiOutput()* ausgibt, oder die Instanz, die an die Funktion *oiGetStringResource()* übergeben wird

  - *ipc* – optionale Polygon-Farben (OFF-Format)
    Falls Polygon-Farben definiert sind, kann ein Material auf *OFML*-Ebene zugewiesen, aber nicht visualisiert werden.
  - *vnm* – optionale Vertex-Normalen (OFF-Format)
    Falls keine Vertex-Normalen definiert sind, werden sie generiert.
  - *3ds* – polygonale Geometrien (3DS-Format)
    Es werden nur Geometrien und Materialien (einschließlich Texturen) übernommen. Falls Polygon-Farben definiert sind, kann ein Material auf *OFML*-Ebene zugewiesen, aber nicht visualisiert werden.

- **Format:** Die Formate entsprechen den jeweiligen Definitionen des 3DS- und des OFF-Formates.

D.2 Materialien

- Materialdefinitionsdateien substituieren einen String-Bezeichner aus *OFML* durch einen entsprechenden Satz von Materialparametern.
- **Namensgebung:** Der Name einer Materialdefinitionsdatei ergibt sich aus dem Namen des Materials in Kleinschreibweise. Dabei sind nur ASCII-Zeichen erlaubt. Sofern ein Materialname aus mehr als einem Wort besteht, werden die Worte aneinander gehangen. Dabei entfallen die Leerzeichen. Die Dateierweiterung lautet *mat*.

  Beispiel: Die Datei "eschenatur.mat" enthält die Definition des Materials *Esche Natur*.
- **Format:** Materialdefinitionsdateien sind zeilenweise aufgebaut und bestehen aus der Benennung des Materials sowie einer beliebigen Anzahl von Materialparameterspezifikationen. Eine Materialparameterspezifikation überschreibt den Initialwert des entsprechenden Materialparameters. Die nachfolgenden Spezifikationen sind zulässig:
  - *amb Red(Float) Green(Float) Blue(Float)*
    Der Schlüssel *amb* spezifiziert die ambiente Farbe des Materials. Die Komponenten sind Gleitkommazahlen im Bereich $0 \leq z \leq 1$. Die initiale ambiente Farbe ist Weiß (1.0 1.0 1.0).
  - *dif Red(Float) Green(Float) Blue(Float)*
    Der Schlüssel *dif* spezifiziert die diffuse Farbe des Materials. Die Komponenten sind Gleitkommazahlen im Bereich $0 \leq z \leq 1$. Die initiale diffuse Farbe ist Weiß (1.0 1.0 1.0). Normalerweise sind die ambiente und die diffuse Farbe gleich.
  - *spe Red(Float) Green(Float) Blue(Float)*
    Der Schlüssel *spe* spezifiziert die spekulare Farbe des Materials. Die Komponenten sind Gleitkommazahlen im Bereich $0 \leq z \leq 1$. Die initiale spekulare Farbe ist Schwarz (0.0 0.0 0.0).
  - *shi Shininess(Float)*
    Der Schlüssel *shi* spezifiziert den spekularen Exponent durch eine positive Gleitkommazahl. Je höher der Exponent ist, desto geringer ist die Ausbreitung der spekularen Highlights. Der initiale spekulare Exponent hat den Wert 30.0.

– *tra Transparency(Float)*

  Der Schlüssel *tra* spezifiziert die Transparenz durch eine nichtnegative Gleitkommazahl, die kleiner oder gleich 1 ist. Der Wert 0.0 steht für völlige Undurchlässigkeit; der Wert 1 bedeutet völlige Transparenz. Die initiale Transparenz hat den Wert 0.0.

– *ref Refraction(Float)*

  Der Schlüssel *ref* spezifiziert die Brechung durch eine positive Gleitkommazahl. Die initiale Brechung hat den Wert 1.0 und entspricht dem Brechungswert im Vakuum.

– *tex image Format(String) Name(String)*

  Der Schlüssel *tex* spezifiziert eine Imagemap-Textur. Initial wird keine Textur im Rahmen des zu definierenden Materials verwendet. Die unterstützten Formate sind Targa (*tga*), BMP (*bmp*), JPEG (*jpg*) und SGI RGB (*rgb*). Der Parameter *Name* gibt den Namen des Images ohne Pfad und Erweiterung an.

– *scale X(Float) Y(Float) Z(Float)*

  Falls über den Schlüssel *tex* eine Textur definiert wurde, spezifiziert der Schlüssel *scale* die Skalierung der Textur. Diese erfolgt durch einen positiven Skalar für jede Dimension. Der initiale Wert ist jeweils 1.0, wobei das Bild, unabhängig von seiner Auflösung auf eine Größe von 1x1m skaliert wird.

– *rotate AngleX(Float) AngleY(Float) AngleZ(Float)*

  Falls über den Schlüssel *tex* eine Textur definiert wurde, spezifiziert der Schlüssel *rotate* die Rotation der Textur um den in Grad anzugebenden Winkel um die jeweilige Achse. Der initiale Wert ist 0.00.00.0.

– *prjx*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *prjx* die Projektion des Bildes auf die x-Achse.

– *prjy*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *prjy* die Projektion des Bildes auf die y-Achse.

– *prjz*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *prjz* die Projektion des Bildes auf die z-Achse.

– *prj X(Float) Y(FLoat) Z(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *prj* die Projektion des Bildes auf die durch X, Y und Z angegebene Achse.

– *circ R(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *circ* die Abbildung des Bildes auf einen Kreis mit dem Radius R.

– *sph R(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *sph* die Abbildung des Bildes auf eine Kugel mit dem Radius R.

– *cyl R(Float) H(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *cyl* die Abbildung des Bildes auf einen Zylinder mit dem Radius R und der Höhe H.

- *cone R1(Float) R2(Float) H(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *cone* die Abbildung des Bildes auf einen Kegel mit den Radii *R1* und *R2*, sowie der Höhe *H*.

- *quad X1(Float) Y1(Float) Z1(Float) X2(Float) Y2(Float) Z2(Float) X3(Float) Y3(Float) Z3(Float) X4(Float) Y4(Float) Z4(Float)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *quad* die Abbildung des Bildes auf eine allgemeine quadrilaterale Fläche mit den entsprechenden Koeffizienten.

- *interp Mode(Int)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *interp* ob eine Interpolation erfolgt (1) oder nicht (0). Der initiale Wert ist 1.

- *once Mode(Int)*

  Falls über den Schlüssel *tex* ein Image-Mapping definiert wurde, spezifiziert der Schlüssel *once* ob eine wiederholte Abbildung des Images erfolgt (1) oder nicht (0). Der initiale Wert ist 1.

Die Anwendung der Materialparameter ist abhängig vom jeweilig angewendeten Darstellungsverfahren. Sofern Objekte bereits eigene Farben/Materialien definieren, dies betrifft eventuell *OiImport*, werden die hiermit definierten Materialien nicht übernommen.

Die Angabe von Materialien kann in Ausnahmefällen alternativ ohne externe Dateien erfolgen. Dabei können die Parameterspezifikationen direkt anstelle des Materialnamens eingegeben werden. Ein derartig definiertes Material muß mit einem '$'-Zeichen beginnen. Anstelle der Zeilenenden werden Semikolons verwendet. Die Verwendung des Schlüssels *mat* ist dabei unzulässig.

Beispiel: Der String "$ amb 1.0 0.0 0.0; dif 1.0 0.0 0.0" setzt eine rote Farbe als Material ohne externe Materialdefinitionsdatei.

D.3 Fonts

- Die in *OFML* unterstützten Fonts basieren auf den von Dr. A. V. Hershey (U.S. National Bureau of Standards) geschaffenen Fonts. Dabei handelt es sich um Vektor-Fonts, die zweidimensionale Linienzüge beschreiben.

  Die folgenden Fonts sind durch eine *OFML*-konforme Laufzeitumgebung bereitzustellen:

  - *default*
  - *cyrillic*
  - *cursive*
  - *timesg*
  - *timesi*
  - *timesib*

- *timesr*
- *timesrb*

- **Namensgebung:** Der Name eines Fonts ergibt sich aus einem Bezeichner (dem Namen des Fonts), in dem alle Buchstaben kleingeschrieben werden. Eine Erweiterung ist nicht vorgesehen.
- **Format:** Das Format entspricht der Definition des Hershey-Font-Formates.

D.4 Externe Tabellen

- Externe Tabellen, z.B. Produktdatenbanken, werden in einem einfachen Textformat abgespeichert. Datensätze werden durch Zeilenwechsel getrennt. Die einzelnen Felder eines Datensatzes besitzen eine feste Länge, es gibt keine Feldtrenner. Felder, die kürzer als die jeweilige Länge sind, werden bis zu dieser Länge aufgefüllt.

  Dieses generische Tabellenformat kann innerhalb von *OFML* über die globale Funktion *oiTable()* gelesen werden (Kap. 6).

- **Namensgebung:** Der Name einer Produktdatenbank kann frei gewählt werden.
- **Format:** Es werden folgende Feldtypen gelesen:
  - Zeichenketten. Diese sind linksbündig und werden ggf. mit Leerzeichen aufgefüllt, es sei denn, es handelt sich um das letzte Feld innerhalb eines Datensatzes. Im letzten Fall wird die Zeichenkette mit dem Zeilenende abgeschlossen.

    Ist das letzte Feld die leere Zeichenkette, kann dieses Feld ganz weggelassen werden. In diesem Fall wird das vorletzte Feld als letztes Feld gemäß der oben genannten Regeln behandelt. Ist das vorletzte Feld ebenfalls leer, sind die Regeln wiederholt anwendbar.
  - Ganzzahlen sind rechtsbündig und werden mit Nullen aufgefüllt.
  - Festpunktzahlen sind rechtsbündig und mit Nullen aufgefüllt, der Dezimalpunkt entfällt.
  - Felder, die als erster Schlüssel für den Zugriff dienen, müssen in aufsteigender Folge sortiert sein.

D.5 Text-Ressourcen

- Text-Ressourcen substituieren einen Symbol-Bezeichner aus *OFML* durch einen entsprechenden Text aus einer externen Datei. Dies findet beispielsweise Anwendung bei Property-Namen, bei Beschreibungstexten, sowie bei Ausgabetexten.
- **Namensgebung:** Der Name einer Ressourcen-Datei ergibt sich aus der Verkettung des Bibliotheksnamens mit dem jeweiligen ISO-Landeskurzzeichen, separiert durch einen Unterstrich. Die Dateierweiterung lautet *sr*. Alle Buchstaben werden klein geschrieben.

Beispiel: Die Datei "room_de.sr" enthält die deutschen Text-Ressourcen für die Bibliothek *Room*.

- **Format:** Die relevanten Zeilen haben das Format

  `@SYMBOL=<Text>`

  Dabei ist der linke Ausdruck der Zuweisung ein gültiges Symbol im Sinne von *OFML*. Der rechte Ausdruck ist ein Text in einem beliebigen 8-bit-Zeichenformat. Für Westeuropa wird dabei der Zeichensatz ISO-Latin 1 (ISO 8859-1) angewendet. Ein weiteres gültiges Zeichenformat ist beispielsweise UTF 8. Zur Verwendung mit formatierter Ausgabe kann der Text eine Format-Zeichenkette sein (Abschn. 6.1). Alle weiteren Zeilen werden ignoriert und können zu Strukturierungs- und Kommentierungszwecken verwendet werden. Per Konvention kennzeichnet ein singuläres Doppelkreuz einen Kommentar. Ein zweifaches Doppelkreuz führt zu einer Strukturierung der Ressource-Datei. Per Konvention werden die folgenden Strukturierungen eingeführt:

  – *## messages:* – Nachfolgend stehen Zeichenketten für Meldungen, Warnungen, usw.
  – *## properties:* – Nachfolgend stehen Zeichenketten für Property-Bezeichnungen.

D.6 Archive

- Archive stellen Behälter dar, in denen normalerweise alle zu einer Bibliothek gehörenden Dateien enthalten sind. Ihr Aufbau entspricht dem durch das UNIX SVR4 Dienstprogramm *ar* verwendete Format.
- **Namensgebung:** Der Name eines Archives wird klein geschrieben. Die Erweiterung ist *alb*. Weitere Festlegungen existieren nicht.
- **Format:** Alle Archive beginnen mit der Zeichenkette `!<arch>\n`. Der Rest des Archivs setzt sich aus Objekten zusammen, von denen jedes aus einem Kopf und dem eigentlichen Inhalt der Datei besteht.

  Der Kopf besteht aus sechs Feldern von ASCII-Zeichen mit fester Länge. Diese Felder enthalten (mit zwei Ausnahmen, s.u.) den Dateinamen (16 Zeichen), die Zeit der letzten Änderung der Datei (12 Zeichen), die Nutzer- und Gruppennummer des Dateibesitzers (je 6 Zeichen), den Zugriffsmodus (8 Zeichen) und die Größe der Datei in byte. Alle numerischen Felder sind dezimal, mit Ausnahme des Zugriffsmodus, der oktal angegeben wird. Der Kopf wird mit der Zeichenfolge `‘\n` abgeschlossen.

  Eine Sonderbehandlung erfolgt für Dateinamen, die länger als 16 Zeichen sind. Ist mindestens eine derartige Datei im Archiv vorhanden, ist das erste Objekt im Archiv keine Datei, sondern eine Tabelle, die die langen Dateinamen enthält und dessen Name `//` ist. An Stelle des Dateinamens befindet sich im Kopf der jeweiligen Datei das Zeichen `/`, gefolgt von einer Zahl, die den Offset bezüglich der Tabelle der Dateinamen angibt.

  Dateien mit einer ungeradzahligen Anzahl von Bytes werden mit einem Zeilentrenner aufgefüllt, was allerdings keinen Einfluß auf die im Kopf angegebene Größe hat. Damit beginnt jedes Objekt auf einer geradzahligen Adresse.

  Von Seiten der *OFML*-Laufzeitumgebung enthält jedes Archiv eine spezielle Datei mit Namen `__attrib`, die Attribute des Archivs enthält. Jedes Attribut beansprucht eine Zeile und

enthält ein durch Leerzeichen getrenntes Paar von Schlüssel und Wert. Folgende Attribute sind standardisiert:

`version`	Die Version des Archivs, bestehend aus zwei durch Punkt getrennte Zahlen.
`valid_span`	Die Gültigkeitsspanne des Archivs, bestehend aus zwei durch Unterstrich getrennte Datumsangaben.
`pwdcheck`	Ein Prüfwort zur Überprüfung der Entschlüsselung.
`md5sum`	Die MD5-Prüfsumme des Archivs.

Alle Attribute sind optional. Es können beliebige Attribute hinzugefügt werden.

Anhang E

Formatspezifikationen

E.1 Formatspezifikationen für Properties

Dieser Abschnitt beschreibt Syntax und Bedeutung von Formatspezifikationen für Properties. Formatspezifikationen können bei der Festlegung von Properties in der Funktion *setupProperty()* in der Schnittstelle *Property* (Abschn. 4.4) angegeben werden.

Die Formatspezifikation hat eine der drei folgenden Formen:

property-format:
"*@L*"
"*@A*"
"*%[-][width][.prec]type*"

Die ersten beiden Formate können bei Properties vom Grundtyp `"f"` verwendet werden und geben an, daß es sich bei dem Property-Wert um eine Längen- bzw. Winkelmaßangabe handelt und daß zur Darstellung bzw. Eingabe des Wertes die vom Nutzer eingestellte Maßeinheit verwendet werden soll. Der Property-Editor muß dann eine Konvertierung zwischen der nutzerdefinierten Maßeinheit und der in OFML für Längen- bzw. Winkelmaßangaben verwendeten Maßeinheit (m bzw. rad) vornehmen.

Die dritte Form wird verwendet, wenn ein OFML-Objekt ein spezielles Format für die Darstellung bzw. Eingabe von Property-Werten erzwingen möchte. Die Format-Zeichenkette dieser Form beginnt mit einem %-Zeichen. Danach sind die folgenden Spezifikatoren in entsprechender Reihenfolge erlaubt:

- Ein optionaler Linksausrichtungsindikator – "*[-]*"
- Ein optionaler Breitenspezifikator – *[width]*
- Ein optionaler Präzisionsspezifikator – *[.prec]*

- Ein notwendiger Typspezifikator – *type*

  Für den Typspezifikator ist der folgende diskrete Wertebereich vorgegeben:

  – Dezimalzahl (*Int*) – *d*

    Das Argument muß ein *Int*-Wert sein. Der Wert wird in eine Zeichenkette konvertiert, die die dezimalen Stellen enthält. Falls die Formatspezifikation einen Präzisionsspezifikator enthält, gibt dieser an, daß die resultierende Zeichenkette mindestens die spezifizierte Anzahl der Stellen besitzt. Sofern der Wert weniger Stellen besitzt, erfolgt in Abhängigkeit vom optionalen Linksausrichtungsindikator ein Auffüllen mit Nullen. Ist der Linksausrichtungsindikator angegeben, werden rechtsseitig Nullen aufgefüllt. Andernfalls werden linksseitig Nullen aufgefüllt.

    Falls der Breitenspezifikator verwendet wird, gibt er die maximale Anzahl der Stellen an, die die resultierende Zeichenkette besitzen darf. Werden Breiten- und Präzisionsspezifikator verwendet, dann muß gelten: $width \geq prec$.

  – Gleitkommazahl (*Float*) – *f*

    Das Argument muß eine Gleitkommazahl sein. Der Wert wird in eine Zeichenkette der Form "*-ddd.ddd...*" konvertiert. Die resultierende Zeichenkette beginnt mit einem Minuszeichen, sofern die Zahl negativ ist. Die Anzahl der Stellen nach dem Dezimalpunkt wird durch den Präzisionsspezifikator angegeben. Falls kein Präzisionsspezifikator angegeben wurde, wird 2 als Anzahl der Dezimalstellen nach dem Komma angenommen.

    Sofern ein Breitenspezifikator verwendet wird, gibt er die exakte Breite der resultierenden Zeichenkette an. Dabei wird das Minuszeichen mitgezählt, der Dezimalpunkt jedoch nicht. Falls der Wert weniger Stellen hat, werden linksseitig Nullen aufgefüllt. Der Linksausrichtungsindikator wird, falls vorhanden, ignoriert. Falls der Wert mehr Stellen hat, werden die führenden Stellen unterdrückt.

  – Zeichenkette (*String*) – *s*

    Das Argument muß eine Zeichenkette sein. Diese wird anstelle des Formatspezifikators eingefügt. Falls der Präzisionsspezifikator angegeben wurde, definiert er die maximale Länge der resultierenden Zeichenkette. Falls die Länge des Arguments die maximale Länge überschreitet, wird die Zeichenkette entsprechend abgeschnitten.

    Falls die Formatspezifikation einen Breitenspezifikator enthält, gibt dieser die minimale Anzahl der Zeichen der resultierenden Zeichenkette an. Falls die Zeichenkette weniger Zeichen besitzt, wird die resultierende Zeichenkette linksseitig (ohne gesetzten Linksausrichtungsindikator) oder rechtsseitig (bei gesetztem Linksausrichtungsindikator) mit Leerzeichen aufgefüllt.

E.2 Definitionsformat für Properties

Dieser Abschnitt beschreibt das Format einer Property-Definitionsbeschreibung, die alle Properties einer Instanz beschreibt und als Ergebnis der Funktion *getProperties()* in der Schnittstelle *Property* (Abschn. 4.4) geliefert wird.

Für das Format der Definition von Properties gelten folgende Regeln:

- Die Beschreibung aller Properties besteht aus den Beschreibungen für jedes einzelne Property, die durch Semikolons voneinander getrennt sind.
- Jede Property-Definition reflektiert die Daten, die an die Funktion *Property::setupProperty()* übergeben werden und besteht aus einer Menge von notwendigen und optionalen Spezifikatoren, die durch Semikolons getrennt werden.
- Nach einem Semikolon kann eine beliebige Anzahl von Leerzeichen stehen.
- Der erste Spezifikator einer Property-Definition ist der Schlüsselspezifikator:
  - *k <str>* – Schlüssel des spezifizierten Propertys.
- Der letzte Spezifikator einer Property-Spezifikation ist der Typspezifikator. Dieser muß einen der folgenden Werte besitzen:
  - *b* – ein boolescher Typ (0 oder 1).
  - *i* – ein Dezimaltyp.
  - *f* – ein Gleitkommazahltyp.
  - *s* – ein Zeichenkettentyp.
  - *ch <str>*n* – eine Auswahlliste mit $n, n > 0$ Zeichenketten zur Verwendung bei Zeichenketteneingabe.
  - *chf <str>* – eine Auswahlliste, deren mögliche Zeichenketten durch die angeführte Funktion geliefert werden.
  - *u* – ein anwenderdefinierter Typ mit einer gegebenen Editor-Identifikation.
- Weitere optionale Spezifikatoren zwischen Schlüssel- und Typspezifikator sind:
  - *n <str>*n* – der Name des Propertys.
  - *d <str>*n* – der initiale Wert des Propertys.
  - *mn <str>* – minimaler Wert einer Dezimal- oder Gleitkommazahl bzw. minimale Anzahl von Zeichen in einem Zeichenketten-Property.
  - *mx <str>* – maximaler Wert einer Dezimal- oder Gleitkommazahl bzw. maximale Anzahl von Zeichen in einem Zeichenketten-Property.
  - *fmt <str>* – C-ähnlicher Formatspezifikator (Abschn. E.1)

Anhang F

Zusätzliche Typen

Die nachfolgend definierten Typen sind nicht direkter Bestandteil von OFML; können aber in OFML-konformen Bibliotheken verwendet werden. Die Schnittstelle *Base* wird implementiert, jedoch mit einigen jeweils spezifischen Einschränkungen.

F.1 Interactor

Beschreibung

- *Interactor* realisiert die Basisklasse für Interaktoren.
- **Schnittstelle(n):** *Base* mit Einschränkungen:

  Die Funktionen *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()* und *unMeasure()* sind nicht vorhanden. Die Instanzvariable *mIsCutable* ist nicht vorhanden.

Initialisierung

- *Interactor(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *Interactor*.

Methoden

- *final makeVisible(pType(Type) ...)* $\rightarrow$ *Void*

  Die Funktion erzeugt eine Instanz des angegebenen Typs als Element, welches die Geometrie des Interaktors repräsentiert. Nach dem Typ-Argument können weitere Konstruktor-Argumente folgen. Ist der Interaktor bereits sichtbar, hat die Funktion keine Wirkung. Die Übergabe von NULL macht den Interaktor unsichtbar.

- *final isVisible() → Int*

  Die Funktion liefert 1, falls der Interaktor sichtbar ist, sonst 0.

- *final getState() → Symbol*

  Die Funktion liefert den Zustand des Interaktors, der durch eines der Symbole *@ENABLED*, *@DISABLED* oder *@ACTIVE* beschrieben wird.

- *final enable() → Int*

  Setzt den Interaktor in den Zustand „frei“ (*@ENABLED*) und liefert immer 1 (Erfolg).

- *final disable() → Int*

  Setzt den Interaktor in den Zustand „gesperrt“ (*@DISABLED*) und liefert immer 1 (Erfolg).

- *final activate() → Int*

  Setzt den Interaktor in den Zustand „aktiv“ (*@ACTIVE*), wobei er sich bereits im Zustand *@ENABLED* befinden muß. Liefert 1 bei Erfolg und 0, wenn der Interaktor gesperrt war.

F.2 Light

Beschreibung

- *Light* ist eine global wirkende aktive Lichtquelle, die allerdings in eine Instanzhierarchie eingebunden ist. Bei Umsetzung der Lichtquelle in einem lokalen Beleuchtungsmodell gilt folgende Vorgehensweise: Falls die Lichtquelle Kinder hat, handelt es sich um eine gerichtete Punktlichtquelle. Diese befindet sich im lokalen Ursprung und leuchtet entlang der lokalen negativen y-Achse. Der Öffnungswinkel des Lichtkegels ergibt sich aus dem Arcus Tangens des Verhältnisses des maximalen z-Wertes des lokalen Begrenzungsvolumens der Lichtquelle zur negativen minimalen y-Koordinate des lokalen Begrenzungsvolumens.

  Falls die Lichtquelle keine Kinder besitzt oder die minimale y-Koordinate des lokalen Begrenzungsvolumens gleich 0.0 ist, handelt es sich um eine ungerichtete Punktlichtquelle.

  In einem globalen Beleuchtungsmodell erübrigt sich diese explizite Unterscheidung.

- Der Typ *Light* darf **nicht** abgeleitet werden.

- **Schnittstelle(n):** *Base* mit Einschränkungen:

  Die Funktionen *getType()*, *isCat()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()* und *unMeasure()* sind nicht vorhanden. Die Instanzvariable *mIsCutable* ist nicht vorhanden.

Initialisierung

- *Light(pFather(MObject), pName(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *Light*.

Methoden

- *final setColor(pColor(Float[3])) → self*

  Die Funktion setzt die Farbe der Lichtquelle. Die Elemente des Vektors *pColor* müssen reelle Zahlen im Intervall von 0.0 bis 1.0 sein, wobei die Intervallgrenzen erlaubte Werte sind. Die Vektorelemente werden als Amplituden der Wellenlängen Rot, Grün und Blau interpretiert. Ihre Linearkombination ergibt die tatsächliche Farbe. Die initiale Lichtfarbe ist Weiß.

- *final getColor() → Float[3]*

  Die Funktion liefert die aktuelle Lichtfarbe der impliziten Instanz.

- *final on() → self*

  Die Funktion aktiviert die Lichtquelle.

- *final off() → self*

  Die Funktion deaktiviert die Lichtquelle.

- *final isOn() → Int*

  Die Funktion signalisiert über ihren Rückgabewert den Status der Lichtquelle: eingeschaltet (1) oder ausgeschaltet (0).

F.3 MLine

Beschreibung

- *MLine* realisiert eine automatische Bemaßungsprimitive, die das in der Hierarchie übergeordnete Objekt automatisch bemaßt.

  Die Liniendicke beträgt 1 in der jeweilig kleinsten Darstellungseinheit des Bildraumes, z.B. 1 Pixel.

- Der Typ *MLine* darf **nicht** abgeleitet werden.

- **Schnittstelle(n):** *Base* mit Einschränkungen:

  Die Funktionen *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()* und *unMeasure()* sind nicht vorhanden. Die Instanzvariable *mIsCutable* ist nicht vorhanden.

Initialisierung

- *MLine(pFather(MObject), pName(Symbol), pDirection(Symbol))*

  Die Funktion initialisiert eine Instanz vom Typ *MLine*. Der Parameter *pDirection* definiert, auf welche Weise die topologisch übergeordnete Primitive bemaßt wird. Es wird entweder die Breite, die Höhe oder die Tiefe des lokalen Begrenzungsvolumens des Vaters bemaßt. Die folgenden Symbole sind erlaubt:

@NX Die Breite wird unten hinten bemaßt. Die Bemaßung liegt in der lokalen x-y-Ebene des Vaters und ist von vorn lesbar.

@NXG Die Breite wird unten hinten bemaßt. Die Bemaßung liegt in der lokalen x-z-Ebene des Vaters und ist von vorn und oben lesbar.

@NXT Die Breite wird unten hinten bemaßt. Die Bemaßung liegt in der lokalen x-z-Ebene des Vaters und ist von hinten und oben lesbar.

@PX Die Breite wird oben hinten bemaßt. Die Bemaßung liegt in der lokalen x-y-Ebene des Vaters und ist von vorn lesbar.

@PXT Die Breite wird unten vorn bemaßt. Die Bemaßung liegt in der lokalen x-z-Ebene des Vaters und ist von vorn und oben lesbar.

@NY Die Höhe wird links hinten bemaßt. Die Bemaßung liegt in der lokalen x-y-Ebene des Vaters, ist von vorn lesbar und ist von unten nach oben ausgerichtet.

@PY Die Höhe wird rechts hinten bemaßt. Die Bemaßung liegt in der lokalen x-y-Ebene des Vaters, ist von vorn lesbar und ist von unten nach oben ausgerichtet.

@NZ Die Tiefe wird unten links bemaßt. Die Bemaßung liegt in der lokalen y-z-Ebene des Vaters und ist von links lesbar.

@NZT Die Tiefe wird unten links bemaßt. Die Bemaßung liegt in der lokalen x-z-Ebene des Vaters und ist von links und oben lesbar.

@PZ Die Tiefe wird unten rechts bemaßt. Die Bemaßung liegt in der lokalen y-z-Ebene des Vaters und ist von rechts lesbar.

@PZT Die Tiefe wird unten rechts bemaßt. Die Bemaßung liegt in der lokalen x-z-Ebene des Vaters und ist von rechts und oben lesbar.

Methoden

- *final setMaterial(pMaterial(String)) → self*

  Das spezifizierte Material wird zugewiesen. Die ambiente Komponente des Materials wird bei der Darstellung als Farbe zugewiesen. Die Darstellung soll ohne Berücksichtigung von Beleuchtung und Schattierung erfolgen.

- *final getMaterial() → String*

  Die Funktion liefert das aktuell gültige Material der impliziten Instanz.

- *final setOffset(pOffset(Float)) → self*

  Diese Funktion setzt den Offset der Maßlinie in Bezug auf die zu bemaßende Kante. Der initiale Offset beträgt 0.1.

- *final getOffset() → Float*

  Die Funktion liefert den aktuellen Offset der impliziten Instanz.

- *final setText(pText(String)) → self*

  Instanzen von *MLine* bemaßen initial automatisch die entsprechende Kante des Begrenzungsvolumens des Vaterobjektes und passen sich an Größenveränderungen automatisch an. Durch Verwendung dieser Funktion kann der Text jedoch explizit gesetzt werden. Dabei repräsentiert der Parameter *pText* den anzuzeigenden Text durch eine ASCII-Zeichenkette.

- *final getText() → String*

  Die Funktion liefert den aktuell angezeigten Text.

F.4 MSymbol

Beschreibung

- *MSymbol* realisiert eine polymorphe Bemaßungsprimitive. Alle Varianten werden in der lokalen x-y-Ebene erzeugt. Die z-Koordinate ist stets 0. Koordinaten bezüglich dieser Ebene werden durch einen Vektor von 2 Elementen repräsentiert (x- und y-Wert in dieser Reihenfolge).

  Die Liniendicke beträgt 1 in der jeweilig kleinsten Darstellungseinheit des Bildraumes, z.B. 1 Pixel.

- Der Typ *MSymbol* darf **nicht** abgeleitet werden.

- **Schnittstelle(n):** *Base* mit Einschränkungen:

  Die Funktionen *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()* und *unMeasure()* sind nicht vorhanden. Die Instanzvariable *mIsCutable* ist nicht vorhanden.

Initialisierung

- *MSymbol(pFather(MObject), pName(Symbol), pMode(Symbol), pValues(Float[][2]))*

  Die Funktion initialisiert eine Instanz vom Typ *MSymbol*. Der Parameter *pMode* spezifiziert in Zusammenhang mit dem variablen Parameter *pValues* die Ausprägung von *MSymbol*. Die Auswertung von *pValues* ist abhängig von der Belegung von *pMode*. Folgende Symbole sind für *pMode* möglich:

@ARCLINE Es wird die Umrißlinie eines Kreissegments erzeugt. Der Ursprung des entsprechenden Kreises wird durch *pValues[0]* angegeben. *pValues[1][0]* definiert den Radius des Kreises durch eine positive Zahl. *pValues[1][1]* definiert die Länge der Linie im Bogenmaß durch eine nichtnegative Zahl. Ist die Länge positiv, beginnt die Linie bei

(pValues[0][0], pValues[0][1]+pValues[1][0])

in Uhrzeigerrichtung. Im negativen Fall beginnt sie im selben Punkt, verläuft aber in entgegengesetzter Richtung.

@CIRCLE Es wird ein gefüllter Kreis im lokalen Ursprung erzeugt. *pValues[0][0]* definiert den Radius des Kreises durch eine positive Zahl.

@POLYLINE Es wird ein Linienzug erzeugt, der die angegebenen Punkte in der entsprechenden Reihenfolge verbindet. Es erfolgt keine Verbindung vom letzten zum ersten Punkt. Die Orientierung wird nicht berücksichtigt.

@RECTANGLE Es wird ein gefülltes Rechteck erzeugt. *pValues[0]* beschreibt die linke, untere Ecke. *pValues[1]* beschreibt die rechte, obere Ecke.

@X_CIRCLE Es wird ein Kreis erzeugt. *pValues[0]* definiert den Ursprung des Kreises in Bezug auf das lokale Koordinatensystem. *pValues[1][0]* definiert den Radius des Kreises durch eine positive Zahl. Ist *pValues[1][1]* gleich 0.0, wird nur die Umrißlinie gezeichnet. Sonst wird ein gefüllter Kreis gezeichnet.

Methoden

- *final setMaterial(pMaterial(String))* → *self*

  Das spezifizierte Material wird zugewiesen. Die ambiente Komponente des Materials wird bei der Darstellung als Farbe zugewiesen. Die Darstellung soll ohne Berücksichtigung von Beleuchtung und Schattierung erfolgen.

- *final getMaterial()* → *String*

  Die Funktion liefert das aktuell gültige Material der impliziten Instanz.

F.5 MText

Beschreibung

- *MText* realisiert eine Vektor-Text-Primitive.

  Die Liniendicke beträgt 1 in der jeweilig kleinsten Darstellungseinheit des Bildraumes, z.B. 1 Pixel.

- Der Typ *MText* darf **nicht** abgeleitet werden.

- **Schnittstelle(n):** *Base* mit Einschränkungen:

  Die Funktionen *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()* und *unMeasure()* sind nicht vorhanden. Die Instanzvariable *mIsCutable* ist nicht vorhanden.

Initialisierung

- *MText(pFather(MObject), pName(Symbol), pText(String))*

  Die Funktion initialisiert eine Instanz vom Typ *MText*. Der Parameter *pText* spezifiziert den anzuzeigenden Text in Form einer ASCII-Zeichenkette.

Methoden

- *final setMaterial(pMaterial(String))* → *self*

  Das spezifizierte Material wird zugewiesen. Die ambiente Komponente des Materials wird bei der Darstellung als Farbe zugewiesen. Die Darstellung soll ohne Berücksichtigung von Beleuchtung und Schattierung erfolgen.

- *final getMaterial() → String*

  Die Funktion liefert das aktuell gültige Material der impliziten Instanz.

- *final setFont(pFont(String)) → self*

  Der spezifizierte Font wird zugewiesen. *pFont* spezifiziert den Font durch einen entsprechenden Fontnamen ohne Pfad- oder Erweiterungsangabe entsprechend Kapitel D.

- *final getFont() → String*

  Die Funktion liefert den aktuellen Font der impliziten Instanz.

- *final setText(pText(String)) → self*

  Der anzuzeigende Text wird durch die ASCII-Zeichenkette *pText* neu gesetzt.

- *final getText() → String*

  Die Funktion liefert den aktuellen Text der impliziten Instanz.

- *final setScale(pScale(Float)) → self*

  Der positive Parameter *pScale* setzt die Skalierung des Textes. Die initiale Skalierung ist 0.05.

- *final getScale() → Float*

  Die Funktion liefert die aktuelle Skalierung der impliziten Instanz.

- *final setAlignment(pAlignment(Symbol)) → self*

  Der Parameter *pAlignment* bestimmt die horizontale Ausrichtung des Textes. Die folgenden Symbole sind hierbei erlaubt:

@LEFT Der Text wird bezüglich des lokalen Bezugspunktes linksbündig ausgerichtet.

@CENTER Der Text wird bezüglich des lokalen Bezugspunktes zentriert ausgerichtet.

@RIGHT Der Text wird bezüglich des lokalen Bezugspunktes rechtsbündig ausgerichtet.

Die initiale Ausrichtung ist *@CENTER*.

- *final getAlignment() → Symbol*

  Die Funktion liefert die aktuelle Ausrichtung der impliziten Instanz.

- *final setMode(pMode(Symbol)) → self*

  Der Parameter *pMode* setzt den Darstellungsmodus des Textes. Die folgenden Symbole sind hierbei erlaubt:

@NORMAL Der Text wird normal dargestellt.

@UNDERLINE Der Text wird durch eine Unterstreichung hervorgehoben.

@BOX Der Text wird durch eine Umrahmung hervorgehoben.

Der initiale Darstellungsmodus ist *@NORMAL*.

- *final getMode() → Symbol*

  Die Funktion liefert den aktuellen Darstellungsmodus der impliziten Instanz.

Anhang G

Verwendete Notationen

G.1 Klassendiagramme nach Rumbaugh

Die in diesem Dokument verwendete Notation für Klassendiagramme stellt eine modifizierte Form der Notation von *Rumbaugh* [Rumb91] dar.

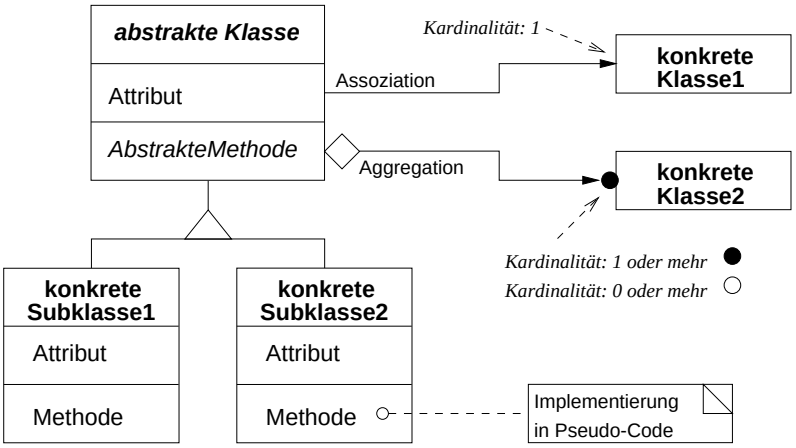

Abbildung G.1: Modifizierte Rumbaugh–Notation für Klassendiagramme

In der objektorientierten Softwareentwicklung werden Klassendiagramme zur Veranschaulichung der Eigenschaften (Attribute, Methoden) von Klassen und von Beziehungen zwischen Klassen verwendet. Prinzipiell können drei **Arten von Beziehungen** unterschieden werden:

- *Vererbung* (Inheritance).
  Eine Unterklasse (subclass) erbt die Eigenschaften ihrer Oberklasse(n).

- *Aggregation* (Consists Of).
  Eine Instanz einer Klasse (ein Objekt) enthält (besteht aus) ein oder mehrere(n) Objekte(n) einer anderen Klasse.

- *Assoziation bzw. Bekanntschaft* (Aquaintance).
  Ein Objekt einer Klasse „kennt“ ein Objekt einer anderen Klasse bzw. ist mit diesem assoziiert.

In abstrakten Modellen kann der Attribut- und/oder der Methodenteil einer Klasse weggelassen werden.

Anhang H

Kategorien

Die nachfolgend definierten Kategorien sind per Definition vordefiniert. Die Verwendung dieser Kategorien ist optional; die Anwendbarkeit und Lesbarkeit von Daten, die in OFML erfaßt wurden, erhöht sich bei Verwendung dieser Kategorien entsprechend.

H.1 Schnittstellenkategorien

Für jede OFML–Schnittstelle ist eine entsprechende Kategorie vordefiniert, deren symbolischer Bezeichner aus dem Präfix „IF_“[1] und dem Namen der Schnittstelle gebildet wird, z.B. *@IF_Article*. Auf diese Weise kann jede Instanz eines OFML–Typs mittels der Funktion *isCat()* abgefragt werden, ob sie eine spezielle Schnittstelle implementiert.

H.2 Materialkategorien

Folgende Kategorien sind vordefiniert, um die Zuordnung eines geometrischen Objekts oder eines komplexen Objekts zu einer bestimmten Materialkategorie zu kennzeichnen:

- *@FRONT* – Das Objekt gehört zur Front eines komplexen Objekts oder stellt diese dar.
- *@GRIFF* – Das Objekt gehört zum Griff eines komplexen Objekts oder stellt diesen dar.
- *@KORPUS* – Das Objekt gehört zum Korpus eines komplexen Objekts oder stellt diesen dar.
- *@KRANZ* – Das Objekt gehört zum Kranz eines komplexen Objekts oder stellt diesen dar.
- *@RUECK* – Das Objekt gehört zur Rückseite eines komplexen Objekts oder stellt diese dar.
- *@SOCKEL* – Das Objekt gehört zum Sockel eines komplexen Objekts oder stellt diesen dar.

[1] abgeleitet aus der englischen Bezeichnung für Schnittstelle: interface

- *@S_FUSS* – Das Objekt gehört zum Fuß eines Stuhles oder stellt diesen dar.
- *@S_LEHNE* – Das Objekt gehört zur Lehne eines Stuhles oder stellt diese dar.
- *@S_SITZ* – Das Objekt gehört zum Sitz eines Stuhles oder stellt diesen dar.
- *@T_FUSS* – Das Objekt gehört zum Fuß eines Tisches oder stellt diesen dar.
- *@T_GESTELL* – Das Objekt gehört zum Gestell eines Tisches oder stellt dieses dar.
- *@T_GEST_ABDECK* – Das Objekt gehört zu einer seitlichen Gestellabdeckung oder stellt diese dar.
- *@T_KANTE* – Das Objekt gehört zur Kante einer Tischplatte oder stellt diese dar.
- *@T_PLATTE* – Das Objekt gehört zu einer Tischplatte oder stellt diese dar.

H.3 Planungskategorien

Folgende Kategorien sind vordefiniert, um die Anbaumöglichkeiten eines Objekts zu kennzeichnen:

- *@CEILING_ELEM* – Das Objekt, z.B. eine Deckenleuchte, kann unterhalb eines Objekts geplant werden.
- *@TOP_ELEM* – Das Objekt, z.B. eine Schreibtischlampe, kann auf der Oberfläche eines Objekts geplant werden.
- *@WALL_ELEM* – Das Objekt, z.B. eine Steckdose, kann an der Oberfläche eines Objekts geplant werden.

Anhang I

Begriffe

- **Beleuchtungsmodell**
  - Mit einem Beleuchtungsmodell wird durch starke Vereinfachung die Beleuchtung von Körpern (3D-Objekten) simuliert.
  - Bei einem **lokalen** Beleuchtungsmodell werden nur das beschienene Objekt und die Lichtquelle (Entfernung, Materialien, etc.) betrachtet.
  - Bei einem **globalen** Beleuchtungsmodell werden zusätzlich andere Objekte der → Szene einbezogen, die Schatten oder Reflexionen verursachen.
- **Bounding-Box**
  - Eine Bounding-Box ist ein rechtwinkeliges Volumen, das einen Körper minimal einschließt.
  - Bei der Definition von Bounding-Boxes erfolgt eine Bezugnahme auf das lokale Koordinatensystem eines Objekts oder auf das gemeinsame Koordinatensystem aller Objekte (Welt- bzw. globales Koordinatensystem).
- **Clipboard**
  - Ein Clipboard ist ein Zwischenspeicher, in dem Objekte abgelegt werden können. Objekte können durch Operationen wie Ausschneiden (Cut) und Kopieren (Copy) in das Clipboard geschrieben werden. Durch die Operation Einfügen (Paste) können sie aus diesem wieder herausgelesen werden.
- **Einheiten**
  - Sofern keine andere Definition im Speziellen existiert, haben Längeneinheiten implizit die Einheit *Meter*.
  - Sofern keine andere Definition im Speziellen existiert, haben Winkeleinheiten implizit die Einheit *Radiant*.

- **Identität**
  - Die Identität einer → Instanz ergibt sich durch einen → Namen im hierarchischen Namensraum, der genau einmal existiert und die Lage in einer Instanzhierarchie eindeutig beschreibt.
- **Instanz**
  - Eine Instanz ist eine konkrete Ausprägung eines → Typs. Sie unterscheidet sich von anderen Instanzen durch eine lokale Kopie von → Attributen, insbesondere durch eine einzigartige → Identität.
  - Synonyme für Instanz sind → Objekt und Entität.
- **Koordinatensystem**
  - Ein Koordinatensystem ist ein durch drei Achsen (x, y, z) definierter orthogonaler Raum, auf den sich Positions- und Richtungsangaben beziehen.
  - Im konkreten Fall spannen z- und x-Achse eine Ebene auf, auf der die y-Achse senkrecht steht.
- **Kategorie**
  - Eine Kategorie ist eine Klassifizierung von → Typen bzw. → Instanzen, die sich durch eine gewisse Betrachtungsperspektive ergibt.
  - Kategorien stellen eine Erweiterung zum Konzept der Typen dar.
- **Name**
  - Der (absolute) Name einer → Instanz beschreibt eindeutig die topologische Lage der Instanz. Alternativ kann eine Instanz auch durch ein → Symbol bezogen auf den jeweiligen Kontext oder durch eine Variable referenziert sein.
- **Objekt**
  - Aus Sicht des Programmierers ist ein Objekt ein Synonym für die → Instanz eines → Typs. Aus Sicht des Anwenders stellt ein Objekt eine gewisse Einheit dar, die als Ganzes erzeugt, ausgewählt, modifiziert und gelöscht werden kann.
- **Programm**
  - Von einem Hersteller aus funktionalen und/oder ästhetischen Gesichtspunkten zusammengefaßte Menge von Produkten.
  - Synonyme: Kollektion, Produktlinie
- **Property (Merkmal)**
  - Eine Property ist eine Eigenschaft einer Instanz, z.B. ein geometrisches Maß oder die Kennzeichnung einer Ausführung, die interaktiv durch den Systemanwender mit Hilfe von geeigneten Dialogen (Property-Editoren) geändert werden kann.

- **Schnittstelle**
  - Eine Schnittstelle ist die Zusammenfassung einer Menge von Methoden und Membervariablen, die ein → Typ bei Schnittstellenkompatibilität definieren bzw. implementieren muß.
- **Symbol**
  - Ein Symbol ist ein String-ähnlicher Wert, der hauptsächlich zur Bezeichnung von Konstanten und Instanz- → Namen verwendet wird.
- **Szene**
  - Eine Szene ist die Zusammenfassung einer Menge von 3D-Objekten, im Kontext von OFML auch → Instanzen genannt.
- **Typ**
  - Ein Typ faßt eine Menge gleichartiger → Instanzen zusammen und definiert für diese Struktur und Verhalten.
  - Ein Typ implementiert ein oder mehrere → Schnittstellen.
  - Ein Typ besitzt höchstens einen direkten Supertyp; dessen Eigenschaften werden von ihm geerbt.
  - Ein Synonym für Typ ist Klasse.
- **Vaterobjekt**
  - Ein Vaterobjekt ist ein → Objekt, von dem Eigenschaften geerbt werden, z.B. ein Namensraum, die räumliche Modellierung, das Material, usw.
- **Wurzelobjekt**
  - Ein Wurzelobjekt ist ein → Objekt, das sich an der Wurzel einer Objekthierarchie befindet. Ein Wurzelobjekt hat somit kein → Vaterobjekt. Alle Objekte, die sich direkt in einer → Szene befinden, sind Wurzelobjekte.

Index