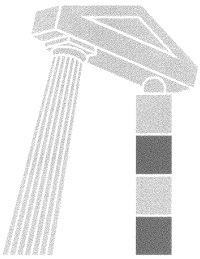
Diplomarbeit

Konzeption und Implementierung einer Sprache für Software-Entwicklungsprozesse

Boris Kötting

März 1996

Universität Kaiserslautern
Fachbereich Informatik
Arbeitsgruppe Expertensysteme
Leiter: Prof. Dr. M. M. Richter
Betreuer: Dr. Frank Maurer



KAPITEL 1

Einleitung

Die Einleitung motiviert die Vorteile der Softwareprozeßmodellierung und gibt einen kurzen Überblick über diese Diplomarbeit.

Die Entwicklung und Wartung von Software-Systemen wird ständig komplexer, da die entwickelte Software selbst immer komplexer und umfangreicher wird. Daher bietet sich zur Entlastung der Projektleiter, Projektmanager und weiterer Projektmitarbeiter eine Rechnerunterstützung der Software-Entwicklung und -wartung an. So können sie einen Überblick über den gesamten Prozeß bekommen und diesen optimieren. Eine Möglichkeit der Unterstützung liefert die Modellierung des Software-Entwicklungsprozesses. Um einen Software-Entwicklungsprozeß modellieren zu können, müssen die notwendigen Basisstrukturen identifiziert und bereitgestellt werden, was Thema dieser Arbeit ist.

Der Begriff der Software-Entwicklung wird im Rahmen dieser Arbeit definiert als die Menge aller Aktivitäten, die zur Entwicklung eines Softwareprodukts notwendig sind. Diese Aktivitäten sind unter anderem abhängig von der Art des Unternehmens und des zu entwickelnden Produkts, sowie von verschiedener Umgebungs- und Projektcharakteristika. Ein Ziel des Software-Engineerings ist die Modellierung dieses Entwicklungsprozesses sowie die Verbesserung der Modelle. Sowohl die Planung als auch die Durchführung sollen abgedeckt werden. Ausführliche Literatur zu dieser Thematik findet man in [Jal91] und [Ghe91].

Um einen Einblick in die bereitzustellenden Konzepte zu geben wird ein Szenario für einen Software-Entwicklungsprozeß vorgestellt, das aus Erfahrungen mit industriellen Software-Entwicklungsprozessen entwickelt wurde [Rom95]. Es dient der Einführung in die Problema-

tik des Entwicklungsprozesses und zeigt die Basisstrukturen, die für diesen Prozeß von Bedeutung sind und daher modellierbar sein müssen. Diese für den Software-Entwicklungsprozeß wichtigen Strukturen werden in einer Prozeßmodellierungssprache zusammengefaßt.

1.1 Der Software-Entwicklungsprozeß

Die ersten Schritte eines Software-Entwicklungsprozesses sehen in der Praxis oftmals wie folgt aus. Ein Software-Entwicklungsprojekt beginnt mit einem Kundenauftrag. Der Kunde hat eine Problembeschreibung, die noch informell gehalten ist und in die Problematik einführt sowie dabei hilft, die Anforderungen auf einer abstrakten Ebene einzuschränken. Als nächstes sind die Systemanforderungen zu definieren. Hierbei müssen späterer Benutzer und einige Mitarbeiter des Unternehmens wie zum Beispiel Software-Ingenieure zusammenarbeiten, um die Bedürfnisse des Produktes genau festzulegen. Im folgenden ist der Systementwurf durchzuführen. Hierbei werden die Systemanforderungen in Software und Hardwareprodukte aufgeteilt und die Schnittstellen zwischen diesen definiert.

Der Prozeß der Software-Entwicklung soll hier jedoch nicht detailliert besprochen werden, hierfür siehe beispielsweise [Som92].

Der oben angeschnittene erste Teil der Entwicklung läßt Rückschlüsse auf Modelle zu, die man für eine Beschreibung dieser Vorgänge benötigt. Es wurden Problembeschreibung und Systemanforderungen beschrieben. Diese Objekte werden als *Produkte* bezeichnet. Sie haben einen strukturellen Aufbau, den man beschreiben kann.

Die Problembeschreibung wird benötigt, um die Systemanforderungen zu definieren. Es bestehen also Zusammenhänge zwischen verschiedenen Produkten. Die Aktivität der Erstellung der Systemanforderungen aus der Problembeschreibung heraus bezeichnet man als einen *Prozeß*. Er beschreibt, *was* getan wird. Der Prozeß schafft eine Verbindung zwischen den beiden Produkten oder genauer gesagt, die Problembeschreibung geht in den Prozeß der Systemanforderungsentwicklung ein und die Systemanforderungen sind das Ergebnis des Prozesses.

Prozesse sind also Aktivitäten, die ausgeführt werden. Für diese Ausführung benötigt man Personen und Maschinen, die diese Aktivitäten durchführen, wie zum Beispiel Mitarbeiter und Computer. Zusätzlich benötigt man noch Werkzeuge und andere Hilfsmittel wie etwa Softwaretools. Die Sammlung der oben erwähnten Objekte kann man als *Ressourcen* bezeichnen. Sie beschreiben, *womit* beziehungsweise *von wem* die Prozesse ausgeführt werden.

Ich möchte hier erwähnen, daß es sich bei Software um eine Objekt handelt, das nicht *produziert* sondern *entwickelt* wird. Entwicklung beinhaltet Kreativität und kann nicht automatisiert werden wie etwa die Produktion von Schrauben. Daher können Maschinen nur Prozesse auf einem niedrigen Abstraktionslevel durchführen, aber nicht beispielsweise Entwurfsentscheidungen treffen.

Es ist denkbar, daß ein Prozeßziel auf verschiedenen Wegen erreicht werden kann. So möchte man erst nach Erhalt der Problembeschreibung entscheiden, ob man den Prozeß der Systemanforderungsdefinition in kleinen aufgeteilten Gruppen oder in einer großen Gruppe durchführen möchte. Da man aber nicht für jede mögliche Variante ein neues Modell erstellen möchte, läßt man *Methoden* zu, die verschiedene Alternativen der Prozeßdurchführung ermöglichen. Sie beschreiben *wie* der Prozeß ausgeführt wird. Die Methode wird erst gewählt, wenn die Durchführung des Projekts bereits begonnen hat.

Damit sind die wichtigsten Bestandteile des Software-Entwicklungsprozesses aus Sicht des Sonderforschungsbereichs 501 angesprochen. Um diese modellieren zu können, wird eine Sprache benötigt.

Warum entschieden wir uns für einen objekt-orientierten Ansatz? Es hat sich gezeigt, daß sich Funktionalitäten von Programmen oft ändern, die Datenstrukturen aber über die Lebensdauer eines Programmes relativ stabil bleiben. Aus dieser Feststellung heraus entwickelte sich der objekt-orientierte Ansatz. Nach diesem Ansatz lassen sich auch die Datenstrukturen für die Sprachelemente der Software-Entwicklungssprache „natürlich” repräsentieren.

Auf der höchsten Abstraktionsebene hat man eine Problembeschreibung, die in den Software-Entwicklungsprozeß eingeht und erhält als Ergebnis ein fertiges Softwareprodukt. Diesen Top-Level-Prozeß hat den Namen „Projekt durchführen“. Das gesamte Modell bezeichnet man als *Projektplan*.

Ein Projekt ist aus der Sicht des Software-Engineering wie folgt zu planen und durchzuführen:

- Identifiziere Charakteristika des Projekts
- Definiere (Qualitäts-) Ziele des Projekts (Metriken und Sollwerte für alle Dimensionen eines Qualitätsziels (Objekt, Blickwinkel, Qualitätsmerkmal)
- Erstelle Projektplan gemäß der Charakteristika und Ziele (Auswahl der geeigneten Modelle, Instrumentierung der Modelle entsprechen von Qualitätsmodellen, Auswahl der unterstützenden Methoden und Werkzeuge)
- Führe Projekt aus (Erfassung der Ist-Werte und Rückkopplung an das Projektteam)

Die Ausführung des Projektplans kann eine Revision der ersten drei Punkte mit sich bringen, wodurch eine Verzahnung der Planung und der Durchführung notwendig wird. Darüber hinaus handelt es sich bei einem Software-Entwicklungsprozeß nicht um einen sequentiellen Prozeß. Beispielsweise können zur selben Zeit ein Prozeß für die Implementierung von Softwarekomponenten und ein Prozeß zur Erstellung von Validierungsplänen für die Softwarekomponentenanforderungen abgewickelt werden. Diese Punkte wurden bei der Konzeption der Modellierungssprache für Software-Entwicklungsprozesse berücksichtigt.

Der Zweck der Modellierung liegt zum einen in der Unterstützung des Projektmanagements und in der Unterstützung der Abwicklungssteuerung, zum anderen in der Optimierung des Entwicklungsprozesses. Nach Beendigung des Projektes folgt die Analyse des Projektes und die Überarbeitung der Modellbasis. Das Projektmanagement kann Stärken und Schwächen lokalisieren und diese in den darauffolgenden Projekten bei der Modellierung vermeiden beziehungsweise andere Ansätze verwenden.

1.2 Die Sprache MVP-L

Die Arbeitsgruppe „Software Engineering“ von Prof. Dr. Rombach hat eine Sprache mit Namen „MVP-L“ (Multi-View Process Modeling language) entwickelt, eine Sprache zur natürlichen Repräsentation von Projektwissen, die im Rahmen des MVP-Projektes an er Universität Kaiserslautern entstanden ist. Mittels MVP-L können Produkte, Prozesse, Ressourcen und Qualitätsattribute modelliert und in einem Projektplan instantiiert werden. Dieser Projektplan ist ein ausführbares Konstrukt, in dem alle anderen Konstrukte wie etwa Prozesse und Produkte, von denen bis dahin nur die Typen vorliegen, instantiiert werden können. MVP-L unterstützt die Wiederverwendung der erstellten Modelle sowie der während des Projekts durch Qualitätsmessung erworbenen Informationen.

Zur Ausführung des mit MVP-L erstellten Projektplans existiert ein Laufzeitsystem namens MVP-S, das die im Projektplan beschriebenen Instantiierungen durchführt und die Prozeßabwicklung einschließlich Qualitätsdatenerfassung ermöglicht [Tol93]. Diese Ausführung dient nur der Kontrolle des Modells, nicht der Unterstützung des Prozesses.

Zu den Nachteilen von MVP-L zählt, daß der Projektplan bereits zu Beginn des Projektes vollständig instantiiert vorliegen muß. Alle Modelle müssen vorliegen, es ist also nicht möglich, die Planung und die Ausführung zu verzahnen, da Änderungen am Modell während der Ausführungszeit nicht mehr möglich sind.

MVP-L bietet die Möglichkeit, durch Entry- und Exit-Kriterien in Prozessen einen Kontrollfluß deklarativ festzulegen. Wenn die Entry-Kriterien erfüllt sind, kann der Prozeß begonnen werden und wenn die Exit-Kriterien erfüllt sind darf der Prozeß beendet werden.

Für genauere und umfassendere Informationen wird auf [Brö95, Rom94] verwiesen.

1.3 CoMo-Kit

Die Arbeitsgruppe „Künstliche Intelligenz“ von Prof. Dr. Richter entwickelte „CoMo-Kit“, ein Werkzeug zur schrittweisen Entwicklung von wissensbasierten Softwaresystemen [Mau93, DeMaPa95]. Es wird ein Mitarbeiterteam bei der Entwicklung von wissensbasierten Anwendungen unterstützt. Dazu muß das Wissen für alle Mitarbeiter zugänglich gemacht werden, was bei CoMo-Kit mit Hilfe eines semantischen Netzes realisiert wird. Ein semantisches Netz besteht aus Knoten und Kanten, wobei die Knoten eine Datenstruktur darstellen und die Kanten Beziehungen zwischen den Knoten wie zum Beispiel einer Spezialisierungsbeziehung oder einer Aggregationsbeziehung entsprechen.

So ist es möglich in einem Netz eine Beschreibung einer Anwendung, die in dieser Anwendung zu bearbeitenden Aufgaben, die möglichen Abwickler dieser Aufgaben, sowie die durch die Aufgaben erzeugten Daten und Datentypen zu modellieren. Analog zu MVP-L existiert auch in CoMo-Kit ein Interpreter [Del95, Sch94, Koh95], der die Ausführung eines Modells dieser Form überprüft und die Information über die Bearbeitung der Aufgaben verwaltet, wie etwa welcher Mitarbeiter welche Aufgabe bearbeitet, welcher Mitarbeiter befähigt ist eine Aufgabe zu bearbeiten und welche Aufgaben erledigt sind. Er unterstützt und validiert also die Anwendung.

Ein Vorteil von CoMo-Kit ist die mögliche Erweiterung des semantischen Netzes zu jedem Zeitpunkt der Abwicklung durch den Interpreter. Dadurch ist eine Verzahnung von Planung und Abwicklung möglich.

Durch die offene Struktur des semantischen Netzes erlaubt CoMo-Kit die Modellierung von Software-Entwicklungsprozessen. Bei diesen ist es oft der Fall, daß ein Prozeß erst beginnen kann, wenn eine anderer Prozeß beendet wird, da er auf dessen erzeugtes Produkt warten muß. Dieses Konzept läßt sich auch mit CoMo-Kit anhand der formalen Parameter realisieren. Ein CoMo-Kit Prozeß wird erst gestartet, wenn die Input-Parameter vollständig vorliegen. Doch hierin liegt ein Problem, da es auch Prozesse gibt, die nicht alle Inputparameter zum Start brauchen. So kann beispielsweise der Pro-

zeß der Validierung schon beginnen, wenn noch nicht alle Objekte erzeugt worden sind. Sowohl MVP-L als auch CoMo-Kit haben Stärken und Schwächen, die in [Gol95] behandelt werden.

1.4 Die neue Softwareprozeßmodellierungssprache

Im Rahmen des SFB501, dessen Thema die „Entwicklung großer Systeme mit generischen Methoden“ ist, stellte sich nun die Aufgabe der Entwicklung von Techniken, Methoden und Werkzeugen für integrierte, prozeßsensitive Software-Entwicklungsumgebungen. Die Konzeption und Implementierung einer Prozeßmodellierungssprache, die auf Erkenntnissen und Erfahrungen mit planbasierten KI-Techniken und der Prozeßmodellierung in MVP-L basiert, fällt in den Aufgabenbereich der Teilprojekte „Experimentell gestützte Modellierung von SE-Prozessen“ (B1) und „Wissensbasierte Planung und Steuerung von Softwareentwicklungsprozessen“ (B2). Hierbei ist die Zielsetzung, die Vorteile der beiden in den Arbeitsgruppen entwickelten Ansätze zu kombinieren und um die zusätzlich benötigte Funktionalität zu erweitern. Die entstandenen Planungs- und Abwicklungskomponenten sollen eine gute Ausdrucksmächtigkeit der Modellierung und Ausführung sowie eine vertretbare Komplexität bei Entwicklung, Spezifikation und Implementierung haben.

Diese Diplomarbeit befaßt sich mit dem Planungstool. Dieses ermöglicht die Beschreibung von Projektplänen für Software-Entwicklungsprozesse. Das Abwicklungstool wickelt diese erstellten Projektpläne ab, wobei die Abwicklung schon beginnen kann, bevor die Modellierung abgeschlossen ist. Die Entwicklung des Abwicklungstools sowie die Verzahnung mit dem Planungstool wird in zukünftigen Arbeiten geschehen.

Das Ziel dieser Diplomarbeit ist die vollständige Konzeption und teilweise Implementierung der oben angesprochenen Sprache. Hierzu gehören die Festlegung der Datenstrukturen und Zugriffsmethoden auf erzeugte Objekte sowie die Migration der Objekte in eine Datenbank.

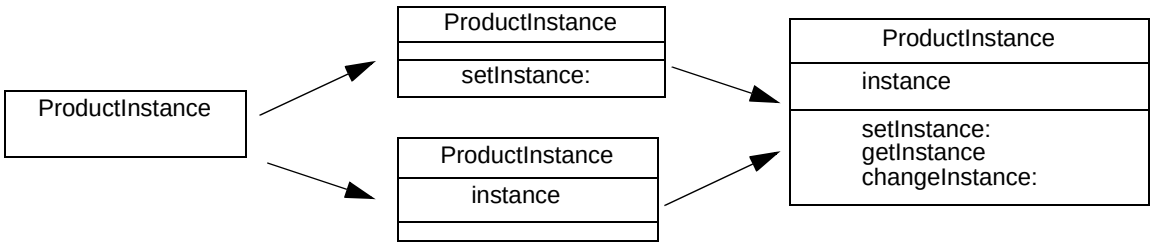
Die erzeugten Strukturen existieren in der objektorientierten Datenbank *GemStone* [Gem95]. Es kann von Smalltalk- und C++-Seite auf die Datenbank zugegriffen werden.

Die Erstellung der Schnittstellenspezifikation erfolgt in „StP-OMT“ (Software through Pictures - Object Modeling Technique), einem kommerziellen CASE-Tool, das auf der Methode von Rumbaugh aufbaut [Rum91].

1.5 Grundlegende Symbolik des Tools StP-OMT

In Abbildung 1 sind die Grundstrukturen des Werkzeugs, die für die Spezifikation der Schnittstelle zwischen Benutzer und System verwendet werden, dargestellt.

ABBILDUNG 1 StP-OMT-Darstellung für eine Klasse

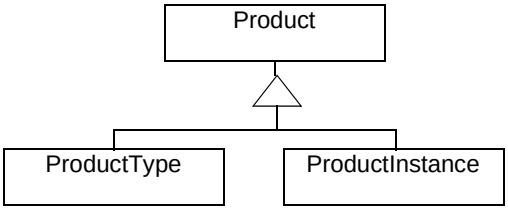


Das Symbol für eine Klasse ist ein Rechteck. Dieses Rechteck besteht aus drei Teilen. In das obere Drittel wird der Klassenname geschrieben, in das mittlere Drittel die Namen der sichtbaren Variablen und in den unteren Teil die sichtbaren Methoden. Falls Informationen zu mehr als einer Eigenschaft der Klasse benannt werden, also wenn etwa der Klassenname und eine Methode eingetragen wird, dann wird die Aufteilung des Rechtecks deutlich gemacht, so wie das in Abbildung 1 angedeutet wird. Zum einen sieht man bei der oberen Möglichkeit den Eintrag einer Methode und bei der zweiten Möglichkeit das Hinzunehmen einer Variablen.

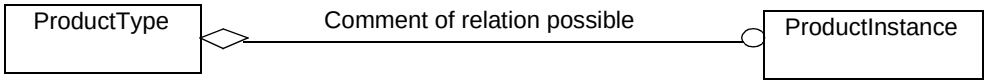
Die beiden wesentlichen Relationen, die in dieser Diplomarbeit bei der Schnittstellenspezifikation benutzt werden, sind Generalisierung und Aggregation. Beide sind in Abbildung 2 zu sehen. Eine Generalisierung stellt eine „is-a“ Beziehung zwischen einer Klasse und ihren Spezialisierungen dar. Die obere Skizze spiegelt eine Generalisierung der Klassen *ProductType* und *ProductInstance* zur Klasse *Product* wieder. Eine Aggregation beschreibt eine „part-of“ Beziehung zwischen Elementen von Klassen. Die untere Skizze beschreibt eine Aggregationsbeziehung zwischen *ProductInstance* und *ProductType*, wobei die Raute sich an der Klasse befindet, die Elemente der anderen Klasse beinhaltet. Diese Beziehung kann kommentiert werden. Das ist zum Beispiel dann sinnvoll, wenn man ausdrücken möchte, welche Variable in der aggregierenden Klasse Elemente aus der aggregierten Klasse aufnimmt.

ABBILDUNG 2 StP-OMT-Darstellung für I.) Generalisierung und II.) Aggregation

I.)

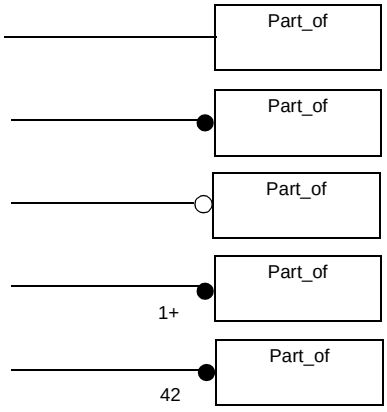


II.)



StP-OMT ermöglicht die Angabe der Anzahl der Elemente, die in die aggregierende Klasse eingeht. Hierbei existieren folgende Möglichkeiten, die in Abbildung 3 zusammengefaßt sind.

ABBILDUNG 3 StP-OMT-Darstellung für Aggrgationskardinalitäten



- **Genau ein Element**
  Bei der eingehenden Klasse ist kein Zeichen am Klassensymbol vorhanden.
- **Null oder mehr Elemente**
  Ein schwarzer ausgefüllter Kreis befindet sich am Klassensymbol.
- **Null oder ein Element**
  Ein nicht ausgefüllter Kreis befindet sich am Klassensymbol.
- **Ein oder mehr Elemente**
  Eine eins und ein „+“-Symbol befinden sich am Klassensymbol.

- **Expliziter Wert**
  Eine Zahl oder eine Zahl mit einem „+“-Symbol befindet sich am Klassensymbol. Es kann also eine minimale Anzahl oder eine feste Anzahl von Elementen angegeben werden.

Bei Abbildung 2 handelt es sich also um ein oder kein Element der Klasse *ProductInstance*, das in die Klasse *ProductType* eingeht.

Im folgenden möchte ich kurz die Gliederung dieser Diplomarbeit vorstellen.

1.6 Ziele und Gliederung der Diplomarbeit

Hauptziel dieser Diplomarbeit ist die Entwicklung von Konzepten, welche die im Software-Entwicklungsprozeß existierenden Basisstrukturen beschreiben. Es sollen Konzepte entwickelt werden, die es einem Modellierer ermöglichen, alle benötigten Basisstrukturen eines Software-Entwicklungsprozesses zu entwickeln. Hierzu soll ihm eine ausreichende Ausdrucksmächtigkeit zur Verfügung gestellt werden. Auf der anderen Seite soll die Komplexität der Konzepte auf einem vertretbaren Niveau gehalten werden.

Der Modellierer kann mit den vorhandenen Konzepten Basisstrukturen des Software-Entwicklungsprozesses erzeugen. Ihm steht eine Spezifikation zur Verfügung, in der die entwickelten Konzepte und die zur Arbeit mit diesen benötigte Funktionalität beschrieben werden. Die Spezifikation dient außerdem als Implementierungsschnittstelle zu Smalltalk. Daher wird die Spezifikation im folgenden auch als Schnittstellenspezifikation bezeichnet. Auf die Konzepte und die Schnittstellenspezifikation wird in Kapitel 2 eingegangen.

Die Konzepte wurden im Rahmen von Diskussionen erstellt, an denen MitarbeiterInnen der Teilprojekte B1 und B2 sowie StudentInnen teilnahmen. Die Ergebnisse wurden in umgangssprachlicher Form als White-Paper in [Mün95] festgehalten. Der Schritt von der informellen Konzeption zur Schnittstellenspezifikation zeigte, daß eine informelle Beschreibung viel Spielraum zur Interpretation liefert, was häufige Änderungen an obigen Papier sowie der Spezifikation zur Folge hatte. Dieser ständige Änderungsprozeß ist immer noch im Gange. Diese Arbeit zeigt den momentanen Stand, der jedoch relativ stabil erscheint. Allerdings sind noch nicht alle Fragen geklärt.

Ein Teil der Basisstrukturen, -das Produktmodell-, soll anhand der Schnittstellenspezifikation implementiert werden. Dieses Ziel der Diplomarbeit wird in Kapitel 3 beschrieben. Als Beispiel für eine Modellierung wird im 4. Kapitel ein konkretes Produkt betrachtet

und daraus ein Produkttyp entwickelt. Es handelt sich hierbei um ein Anforderungsdokument, das im Rahmen eines Praktikums in der Arbeitsgruppe von Professor Rombach entstanden ist.

Die modellierten Basisstrukturen sollen in einem verteilten System verschiedenen Personen zugänglich sein. Hierfür bietet sich die Arbeit mit einer objekt-orientierten Datenbank an. Ein weiteres Ziel dieser Diplomarbeit ist die Übertragung der erzeugten Basisstrukturen in die objekt-orientierte Datenbank GemStone. Kapitel 5 geht hierauf kurz ein und schildert die Vorgehensweise zur Übertragung von Smalltalk-Objekten in die Datenbank.

Zum Abschluß wird in Kapitel 6 die Arbeit zusammengefaßt. Es wird ein Fazit gezogen sowie ein Ausblick auf weitere, notwendigerweise auszuführende Arbeiten gegeben.

KAPITEL 2

Die Spezifikation der Prozeßmodellierungssprache

In diesem Kapitel werden die Struktur der Prozeßmodellierungssprache und die Sprachkonzepte vorgestellt.

Das Ziel der entwickelten Sprache ist es, die Arbeitsabläufe bei der Software-Entwicklung zu modellieren. Sie dient als Grundlage für die Entwicklung eines Interpreters, der die erzeugten Modelle abwikkelt. Eine gängige objekt-orientierte Sprache stellt ein Menge von Strukturen zur Verfügung, mit denen der Programmierer seine Programme erzeugen kann. Die Vorgehensweise ist bei der Prozeßmodellierungssprache dieselbe, nur wird die Menge der zur Verfügung stehenden Strukturen auf eine Menge von Konstrukten eingeschränkt, die zur Modellierung von Basisstrukturen des Software-Entwicklungsprozesses notwendig sind. Diese Konstrukte existieren in dieser Form in keiner gängigen objekt-orientierten Sprache und werden daher neu implementiert. Die zulässigen Ausdrücke der Sprache beschränken sich auf die neu implementierte Menge von Ausdrükken. Hierzu zählen auch Konstrukte, die in anderen objekt-orientierten Sprachen zu finden sind wie beispielsweise Integerzahlen.

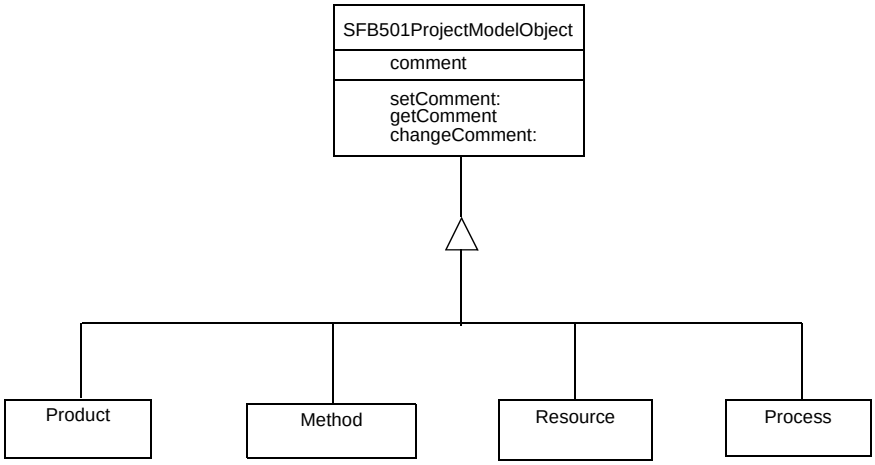
Für die Prozeßmodellierungssprache wird eine komplette Spezifikation der zur Verfügung stehenden Sprachkonstrukte angefertigt. Analog zu gängigen objekt-orientierten Sprachen kann man diese Spezifikation als vordefinierte Klassenhierarchie betrachten. Ebenso existieren bei der Prozeßmodellierungssprache die den objekt-orientierten Sprachen typische Vererbung und das Konzept der Aggregation.

Zur Veranschaulichung der im folgenden beschriebenen Strukturen sind StP-OMT-Abbildungen angegeben. Hierbei handelt es sich um Teile der Gesamtschnittstellenspezifikation. Die Klassenstrukturen

entsprechen dem Datenmodell, daß in diesem Kapitel beschrieben wird. Die in den Klassen enthaltenen Methoden und Variablen werden in diesem Kapitel nicht beschrieben, sondern erst bei der Beschreibung der Implementierung im 3. Kapitel. Sie gehören nicht zum Datenmodell.

Die Objekte der Prozeßmodellierungssprache lassen sich in die Basisstrukturen Produkte, Prozesse, Methoden und Ressourcen unterteilen. Diese Einteilung ist in Abbildung 4 zu sehen.

ABBILDUNG 4 Die Basisstrukturen der Prozeßmodellierungssprache



Produkte entsprechen Informationseinheiten, die im Laufe des Entwicklungsprozesses anfallen, von der Problembeschreibung bis zum ausführbaren Software-System. Prozesse beschreiben die Abläufe zur Erzeugung der Produkte. Methoden beschreiben, wie diese Abläufe realisiert werden. Ressourcen beschreiben die Mittel, die den gesamten Prozeß ausführen und unterstützen. Diese grobe Einteilung wird im folgenden verfeinert und die einzelnen Verfeinerungen erläutert.

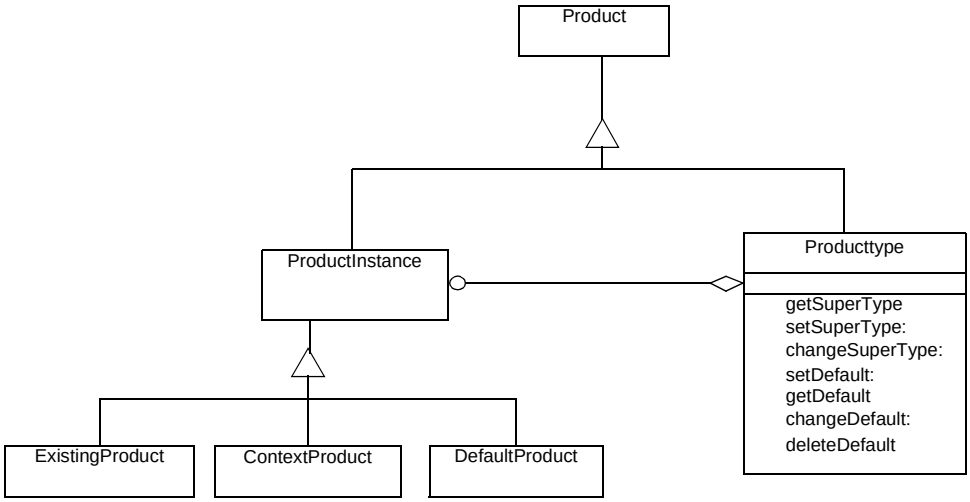
2.1 Produkt

Der Aufbau der obersten Stufe der Klassenhierarchie für Produkte ist in Abbildung 5 zu sehen. Zu Produkten zählen zum einen Produkttypen und zum anderen Produktinstanzen.

2.1.1 Produktinstanzen

Bereits zu Beginn der Modellierung können Produktinstanzen existieren. Aus dem Blickwinkel der Wiederverwendung gehören hierzu eine Software-Komponente, die auch im neuen Projekt verwendet

ABBILDUNG 5 Produkthierarchie



werden soll, oder die für die Software-Entwicklung bedeutsame Norm ISO9000. Instanzen, die für die Initialisierung von selbst modellierten Produkttypen bei der Abwicklung vorgesehen sind, werden während der Modellierung erzeugt.

Produktinstanzen gehen bei der Abwicklung in Produkte oder Prozesse ein. Es werden drei verschiedene Formen von Produktinstanzen unterschieden, die ebenfalls in dieser Form in Abbildung 5 zu finden sind:

- **Kontextprodukte**

  Unter einem Kontextprodukt versteht man Instanzen, die im Zusammenhang mit Prozessen und Produkten stehen, aber nicht ein Teil des Produktes oder Prozesses sind. Beispiele hierfür sind Handbücher (Manuals) oder ISO-Normen. Diese Produkte können von Prozessen lediglich konsumiert werden

- **Bereits existierende Produkte**

  Ein bereits existierendes Produkt geht in einen Prozeß ein und kann von diesem konsumiert oder modifiziert werden. Ein Beispiel hierfür ist eine Wiederverwendungskomponente, also etwa ein bereits existierendes Programm-Modul, dessen Funktionalität weiter genutzt werden kann. Im Gegensatz zu Defaultprodukten handelt es sich bei bereits existierenden Produkten um eine komplette Instanz einer Klasse, ein Template ohne freie Parameter.

- **Defaultprodukte**

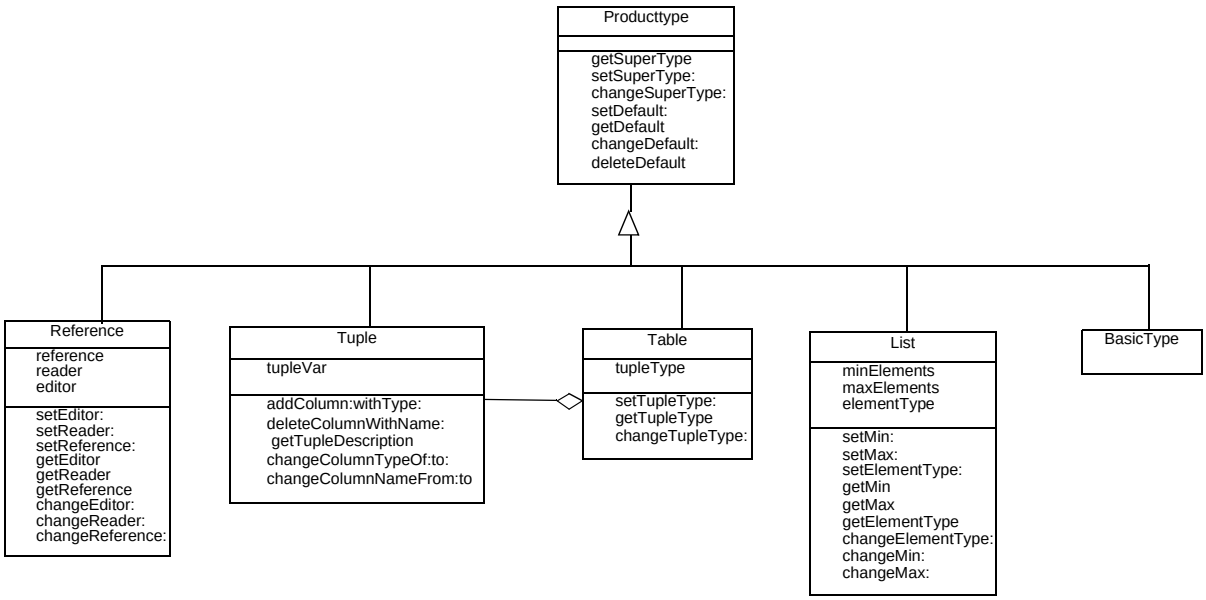
  Ein Defaultprodukt dient der Initialisierung von Instanzen eines Produkttyps. Für die Erstellung einer Produktinstanz von einem selbstdefinierten Produkttyp muß zunächst für den Produkttyp eine Klasse erzeugt werden. Danach erzeugt man eine Instanz die-

ser Klasse und füllt die Slots und Attribute entsprechend den angegebenen Defaultwerten. Danach erzeugt man eine Instanz der Klasse *DefaultProduct* (die man vom Produkttyp aus referenziert) und setzt in dieser Instanz einen Zeiger auf die vorher erzeugte Instanz. Abschließend hierzu ein Beispiel:
Der selbstdefinierte Typ Kodedokument enthält neben verschiedenen Slots, Regeln und Attributen das Attribut „Maximale Bearbeitungszeit“. Ist der maximale zeitliche Aufwand in einer Firma für die Bearbeitung eines Kodedokuments auf einhundert Stunden festgelegt worden, so kann man diesen Wert in ein Defaultprodukt schreiben. Jede nun erzeugte Instanz vom Typ Kodedokument hat bereits zum Zeitpunkt der Kreation das Attribut „Maximale Bearbeitungszeit belegt mit dem Wert einhundert. Dieser Wert kann jedoch immer noch geändert werden.

2.1.2 Produkttypen

Produkttypen ermöglichen dem Benutzer die Definition der Strukturen von Datenobjekten. Hierbei unterscheidet man zunächst zwischen vordefinierten Produkttypen und komplexen Produkttypen. Die vordefinierten Produkttypen lassen sich wiederum in Basistypen, Referenzen, Listen, Tupel und Tabellen unterteilen. Die vordefinierten Typen findet man in Abbildung 6.

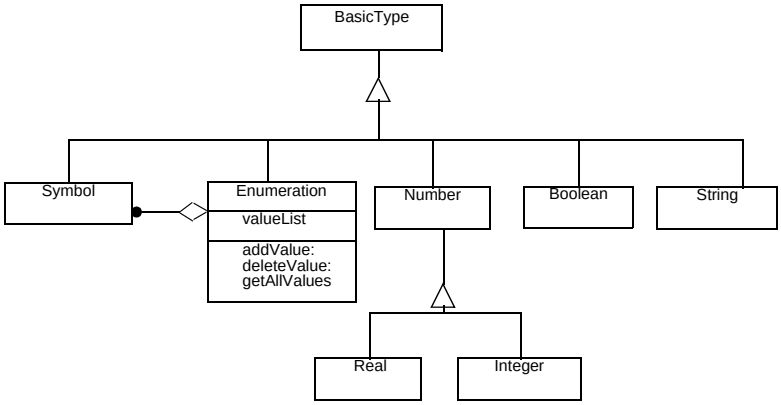
ABBILDUNG 6 Hierarchie der vordefinierten Produkttypen



Basistypen

Einen Überblick über die Basistypen verschafft Abbildung 7. Als

ABBILDUNG 7 Hierarchie der Basistypen für Produkte



Basistypen stehen *String*, *Symbol*, *Boolean*, *Enumeration* und *Number* zur Verfügung. Diese sind nach der Erfahrung der beiden Projekte MVP-L und CoMo-Kit ausreichend für die Modellierung von Software-Entwicklungsprozessen.

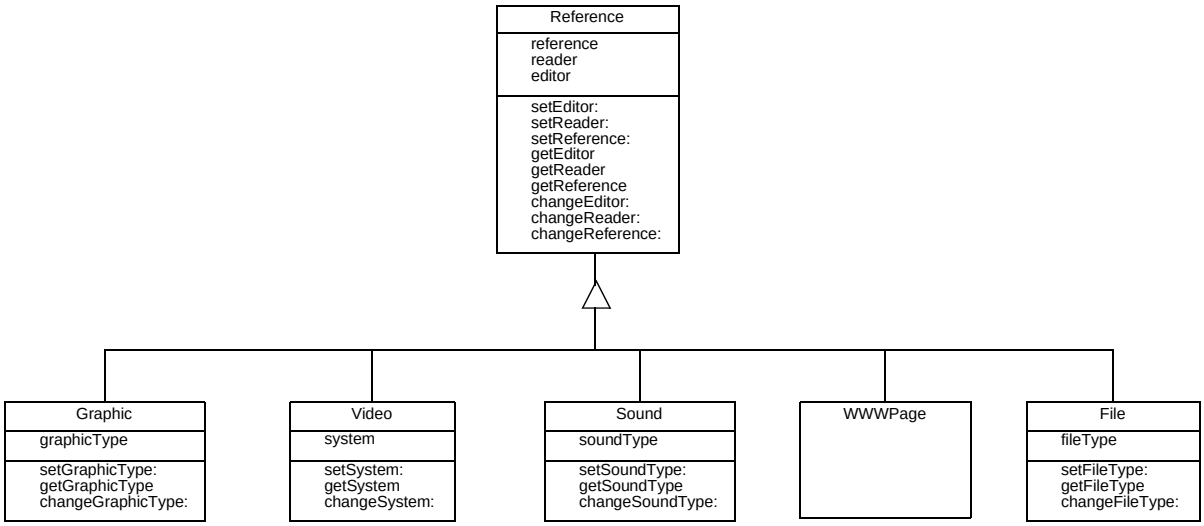
Der Unterschied zwischen einem String und einem Symbol ist der, daß ein Symbol eindeutig ist. Eine Enumeration ist eine Menge von Objekten der Klasse *Symbol*. Dieses spiegelt sich in Abbildung 7 durch eine Aggregationsbeziehung zwischen den Klassen *Enumeration* und *Symbol* wider. Diese Menge kann vom Benutzer definiert werden.

Dem Benutzer stehen zwei verschiedene Arten von Numbers zur Verfügung. Die Klasse wird spezialisiert zu *Integer* und *Real*.

Referenzen

Die Referenztypen enthalten eine Referenz auf ein konkret existierendes Objekt. Der Zweck solcher Referenzen ist die Einbindung von Objekten wie etwa Texte oder CAD-Dokumente, die mit fremden (kommerziellen) Softwarepaketen erstellt werden. Abbildung 8 zeigt die vorhandene Klassenstruktur für die vorhandenen Referenzentypen. Zu diesen Referenztypen zählen *Graphic*, *WWW-Page*, *Video*, *Sound* und *File*. Diese sind in der Grafik am linken Bildrand plaziert. Bei diesen Typen kann zusätzlich ein Editor und ein Reader angegeben werden. Beispielsweise kann man bei WWW-Seiten als Reader „Netscape“ und als Editor „asWedit“ angeben.

ABBILDUNG 8 Hierarchie für Refenztypen von Produkten



Listen

Eine Liste zeichnet sich durch folgende Charakteristika aus:

- **Kardinalität**

  Die Kardinalität einer Liste gibt an, wieviele Elemente minimal und maximal in ihr enthalten sein können. Hierbei ist auch die Angabe „unbegrenzt“ als maximaler Wert möglich. Das bedeutet aber lediglich, daß es keine obere Grenze für die Anzahl der Listenelemente gibt. Die Liste selbst ist jedoch endlich.

- **Nicht-Eindeutigkeit**

  Die Elemente der Liste sind nicht eindeutig. Das bedeutet, daß ein Element in einer Liste auch mehrfach auftreten kann.

- **Typrestriktion**

  Um die Komplexität des Datentyps Liste zu begrenzen, wurde festgelegt, daß die Elemente der Liste nicht vom Typ Liste, Tupel oder Tabelle sein dürfen. Alle Elemente der Liste müssen von einem festgelegten Produkttyp oder einem Untertyp dieses Produkttyps sein.

Tupel

Ein Tupel besteht aus einer Menge von Elementen, die jeweils wiederum einen eigenen Typ haben. Dieser Typ muß allerdings einen Produktbasistyp referenzieren. Diese Einschränkung beeinträchtigt zwar die Ausdrucksmächtigkeit, aber das ist wegen der hohen Komplexitätsverminderung akzeptabel.

Tupel kann man auch als komplexen Produkttyp definieren. Der Grund für diese Redundanz und eine explizite Aufnahme des Typs Tupel in die Sprache liegt darin begründet, daß man die Anbindung an relationale Datenbanken ermöglichen will und so die einfache Definition eines Tupeltyps während der Modellierung ermöglicht. Allerdings muß dieser Tupeltyp nicht einem physikalisch existierenden Tupeltyp einer Datenbank entsprechen, sondern kann zum Beispiel auch ein Ergebnistyp einer Join-Operation sein. Bei einer Join-Operation ist es aber häufig der Fall, daß man als Ergebnis eine Menge von Tupeln erhält. Daher liegt die Definition eines Tabellentyps nahe.

Tabellen

Konzeptionell unterscheidet sich eine Tabelle nicht von einem Tupel. Der einzige Unterschied besteht darin, daß zu einem späteren Zeitpunkt in der Ausführungsmaschine ein Objekt vom Typ Tabelle aus einer Menge von Tupeln besteht, wohingegen ein Element vom Typ Tupel tatsächlich nur ein einziges Tupel darstellt. Die Datenstruktur Tabelle referenziert daher lediglich ein bereits existierendes Element vom Typ Tupel. In Abbildung 6 wird dieser Zusammenhang wiederum über eine Aggregationsbeziehung zwischen beiden Klassen dargestellt.

Ebenso wie bei Tupeln liegt die Aufnahme des Produkttyps Tabelle in der Anbindung an relationale Datenbanken begründet. Die Redundanz der Typmodellierung wird durch deren Zweckmäßigkeit gerechtfertigt. Hierbei ermöglichen Tabellen die Aufnahme einer Menge von Tupeln. Analog zu Tupeln muß der Typ des referenzierten Tupels nicht einem physikalischen Datenbankschema entsprechen.

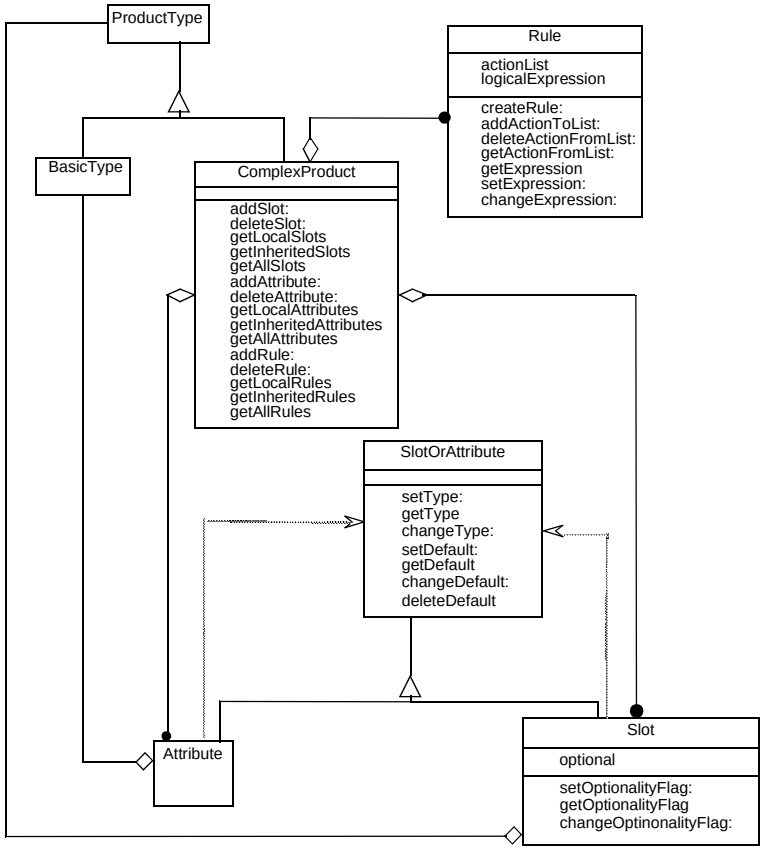
Nach Behandlung der vordefinierten Typen folgt nun eine Beschreibung der komplexen Produkttypen.

Komplexe Produkttypen

Komplexe Produkte bestehen konzeptionell aus Teilprodukten (Slots), Attributen und Regeln. In Abbildung 9 ist die Struktur der komplexen Produkte zusammengefaßt. Hierbei wurde die Klasse *ProductType* vereinfacht dargestellt. Man erkennt die Aggregationsbeziehung zwischen der Klasse *ComplexProduct* und den Klassen *Rule*, *Slot*, und *Attribute*, die Inhaltsbestandteile von Produkten darstellen.

Teilprodukte

ABBILDUNG 9 Hierarchie für komplexe Produkte



Teilprodukte sind wiederum Produkte, die Inhaltsbestandteile von Produkten darstellen, wie zum Beispiel das Inhaltsverzeichnis ein Teilprodukt eines Systemanforderungsdokumentes ist. Die Beschreibung der Teilprodukte erfolgt durch die Angabe eines Namens, eines Typs, einer Optionalitätsangabe und eines Kommentars. Jedem Teilprodukt wird genau ein Produkttyp zugeordnet. Dieses dient vor allem der Übersichtlichkeit und der Komplexitätsbeschränkung. Es ist also nicht möglich, die Typfrage offen zu lassen oder alternative Typen anzugeben. Der Teilprodukttyp ist beliebig, kann also auch wieder ein komplexer Produkttyp sein. Ist die Optionalitätsangabe erfüllt, so kann das Teilprodukt existieren, es muß aber nicht.

Attribute

Attribute entsprechen Eigenschaften von Produkten, wie zum Beispiel eine Kostenabschätzung für ein Kodedokument oder die Entwicklungskosten für ein Softwareentwicklungsprojekt. Attribute und Teilprodukte werden also bereits konzeptionell unterschieden. Der semantische Unterschied besteht darin, daß Teilprodukte Repräsentationen in der realen Welt haben, wohingegen Attribute Eigenschaften

darstellen, die Aussagen über Produkte, Teilprodukte und Prozesse machen.

Zum jetzigen Zeitpunkt unterstützt die Sprache nur Basistypen für Attribute, komplexe Typen sind aber für spätere Ausbaustufen der Sprache geplant. Die Basistypen, die hier verwendet werden können, sind die Basistypen, die auch für Produkttypen verwendet werden können, wie zum Beispiel Integer oder Boolean. Diese Beziehung wird in Abbildung 9 durch die Verbindung zwischen der Klasse *Attribute* und der Klasse *BasicType* dargestellt. Das bedeutet, daß die Typhierarchie für Attribute zur Zeit eine Teilmenge der Typhierarchie der Produkte darstellt. Es wäre nicht sinnvoll, alle vordefinierten Typen für Produkte auch für Attribute zuzulassen. Daher werden die Typhierarchien für Produkte und Attribute getrennt. So wäre ein Attribut vom Typ WWW-File (was eine Repräsentation in der realen Welt hat) nicht sinnvoll. Es wird in späteren Ausbaustufen Attributtypen geben, die für Produkte ungeeignet sind wie zum Beispiel verschiedene Qualitäts- und Komplexitätsmaße. Als Beispiel ein Qualitätsmodell für das Qualitätsmaß Produktivität sei hier das von [Boe81] vorgeschlagene COCOMO-Modell erwähnt. Dieses Modell erlaubt die Vorhersage des Erstellungsaufwandes in Monaten auf der Basis der geschätzten Produktgröße in Kodezeilen sowie weiterer fünfzehn Projektvariablen wie zum Beispiel der Art des Projektes oder der Stufe der Vorhersagbarkeit des Aufwandes.

Regeln

Zur Beschreibung von Beziehungen in und zwischen Produkten können Regeln definiert werden. Eine Regel besteht aus einer Bedingung und einer Menge von Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt ist. Bei der Bedingung handelt es sich um einen einfachen oder komplexen boolschen Ausdruck, der aus boolschen Variablen und Konstanten, sowie den boolschen Operatoren Negation, Konjunktion, Disjunktion und Exklusives Oder aufgebaut sein kann. Ein Beispiel für eine boolschen Ausdruck ist:
NOT (SystemAnforderungsdokument.Kosten LESS THAN 3000), wobei SystemAnforderungsdokument ein Produkt ist, ein Attribut dieses Produktes die Entwicklungskosten vom Typ Integer sind, LESS THAN ein Vergleichsoperator mit Ergebnistyp Boolean und NOT ein logischer Operator ebenfalls mit Ergebnistyp Boolean ist. Ist die Bedingung erfüllt, sind also die Kosten für das Dokument über einen festen Betrag gestiegen, so müssen die Aktionen im Aktionsteil ausgeführt werden. Eine typische Aktion ist eine Zuweisung. In Abbildung 10 findet man die Zuweisung (*Assignment*) als Spezialisierung der Klasse *Action*. Eine Zuweisung besteht aus einer Variable und einem auszuwertenden Ausdruck, die typkompatibel sein müssen. Bei einer Zuweisung darf es sich beim Typ der Variable lediglich um einen Basistyp handeln. Für Ausdrücke stehen die Typen *String*,

Symbol, *Boolean* und *Arithmetical* zur Verfügung. Arithmetische Ausdrücke können zum Beispiel mit den Basisoperatoren Addition, Subtraktion, Multiplikation und Division gebildet werden. Für spätere Ausbaustufen sind weitere Operatoren vorgesehen, die problemlos in die bereits existierenden Strukturen eingebaut werden können. Ein Beispiel für eine Zuweisung lautet
Systemanforderungsdokument.Kosten := Benutzeranforderungsdokument.Kosten + Entwickleranforderungsdokument.Kosten.

Bei Abbildung 10, die aus Gründen der Übersichtlichkeit nur die oberste Ebene der Regelstruktur enthält, fällt die Mehrfachvererbung auf, die in Smalltalk nicht direkt implementiert werden kann. Sie entsteht durch die beiden verschiedenen Spezialisierungsansätze „Kardinalität“ und „Ergebnistyp“ für Ausdrücke. Die Mehrfachvererbung kann aber mit Duldung von Redundanzen in eine einfache Vererbung aufgetrennt werden. Dieses sieht man in Abbildung 11 und Abbildung 12. Die unteren Ebenen der Regelklassen, die in Abbildung 10 noch nicht sichtbar sind, findet man in diesen Abbildungen.

Der Übergang von Mehrfach- zur Einfachvererbung hat Vor-und Nachteile. So erbt z.B. die Klasse *BasicTypeVariable* von den Klassen *ArithmeticalExpression*, *BooleanExpression* und *Expression*. Daher enthält sie auch eine Variable „varType“, die ihren Typ festlegt. In Abbildung 11 für boolsche Ausdrücke existiert nur noch eine Klasse *BooleanVariable*, die keine Typvariable mehr braucht, da sie eindeutig als boolsche Variable zu erkennen ist. Eine Typüberprüfung wird somit durch die festgelegte Syntax überflüssig gemacht. Der Nachteil ist, daß nach der Auftrennung der Strukturen nicht mehr eine, sondern vier Variablenklassen existieren, namentlich neben der oben bereits genannten noch *StringVariable*, *SymbolVariable* und *ArithmeticalVariable*. Ein großer Vorteil besteht in der Tatsache, daß die Strukturen so direkt in Smalltalk implementiert werden können.

In Regeln können Teilprodukt- und Attributbeziehungen auf folgenden Ebenen aufgebaut werden:

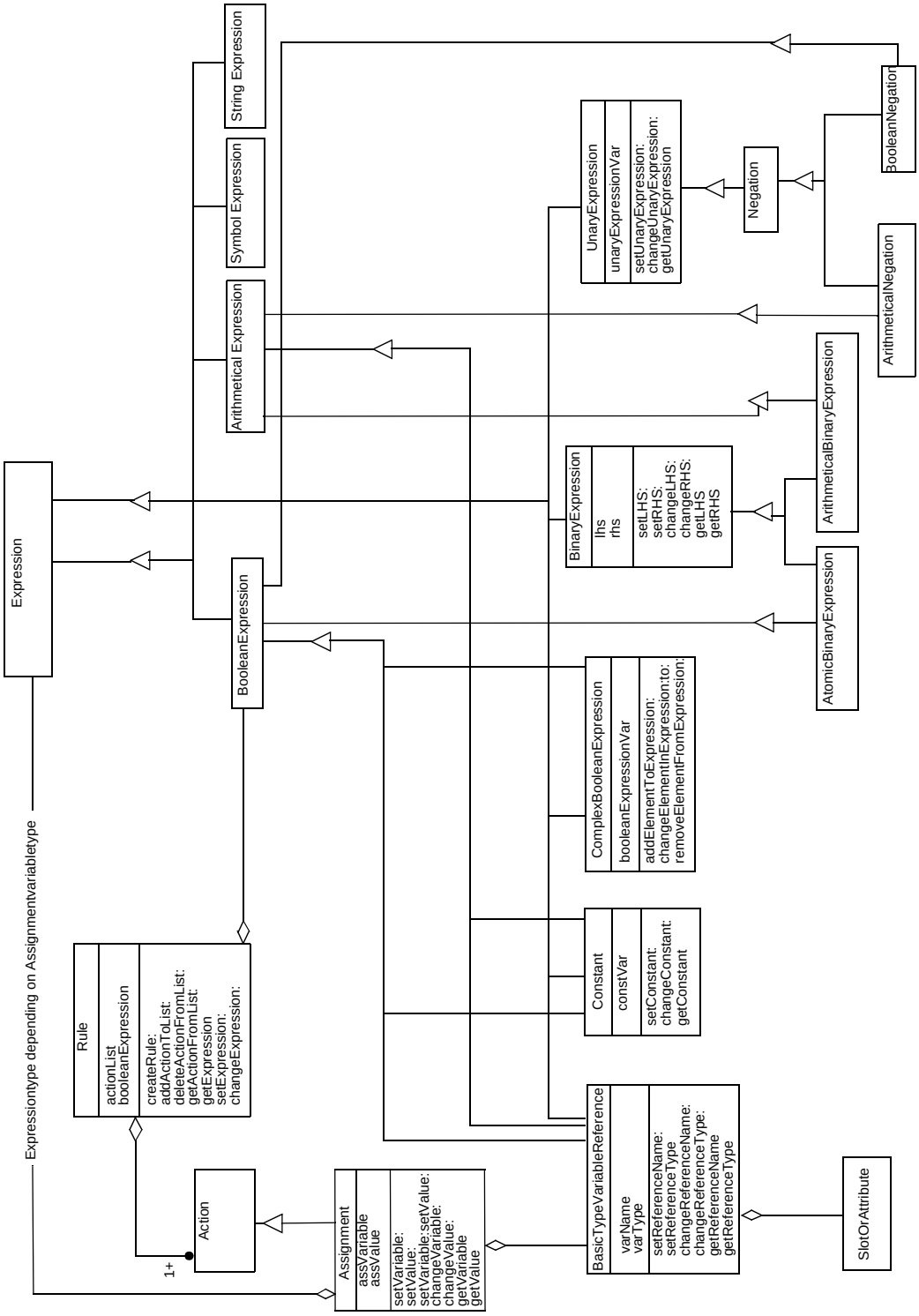
- **Attribut/Attribut**

  Beziehungen zwischen Attributen sind auf einer oder auf zwei aufeinanderfolgenden Verfeinerungsebenen möglich. Bei letzterem Fall werden die Regeln im umfassenderen, also zusammengesetzten Produkt beschrieben.
- **Attribut/Teilprodukt**

  Beziehungen zwischen Attributen und Teilprodukten sind nur auf einer Verfeinerungsebene möglich.
- **Teilprodukt/Teilprodukt**

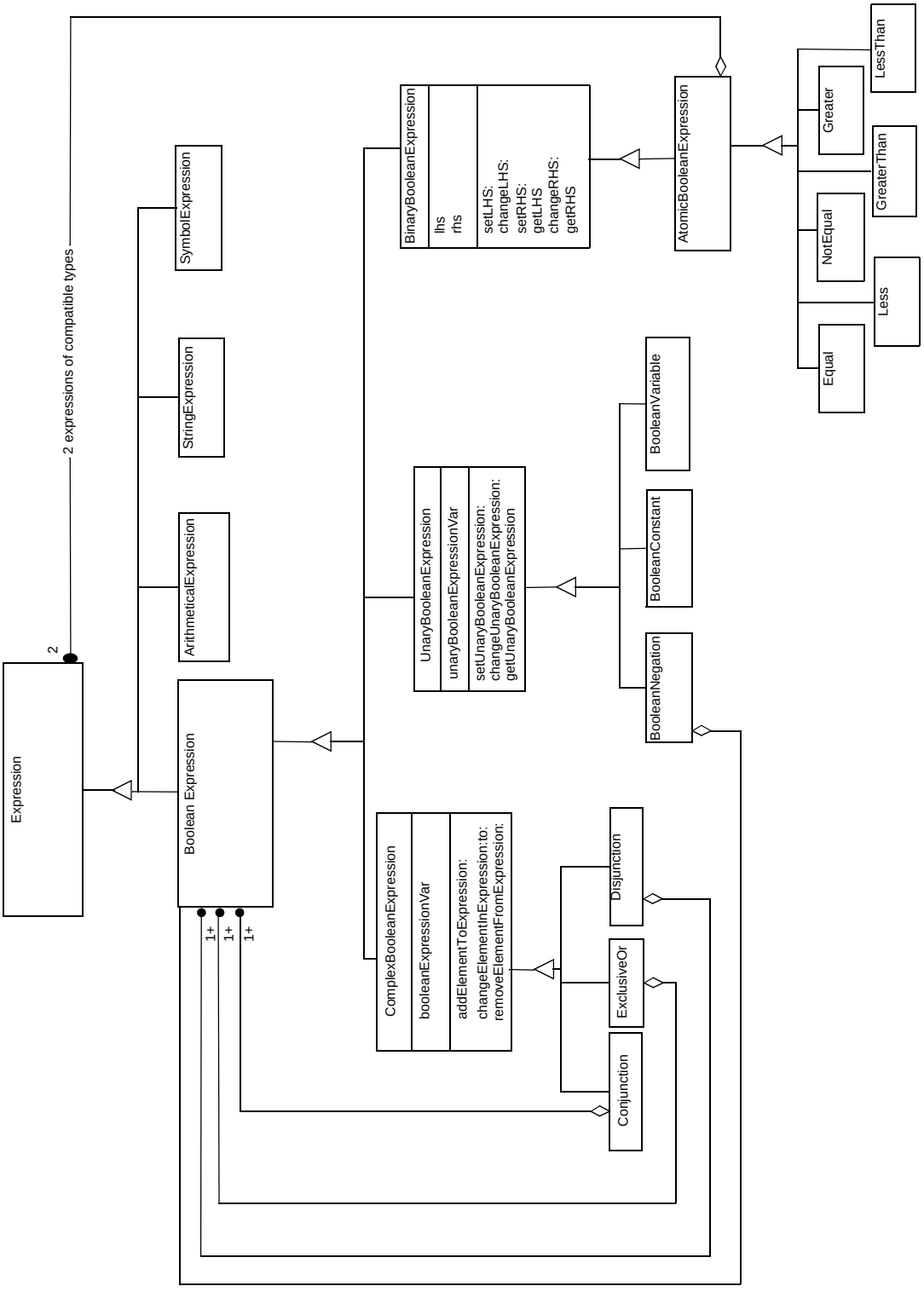
  Beziehungen zwischen Teilprodukten sind auf einer oder auf zwei Verfeinerungsebenen möglich. In letzterem Fall werden die

ABBILDUNG 10 Oberste Ebene der Regelhierarchie



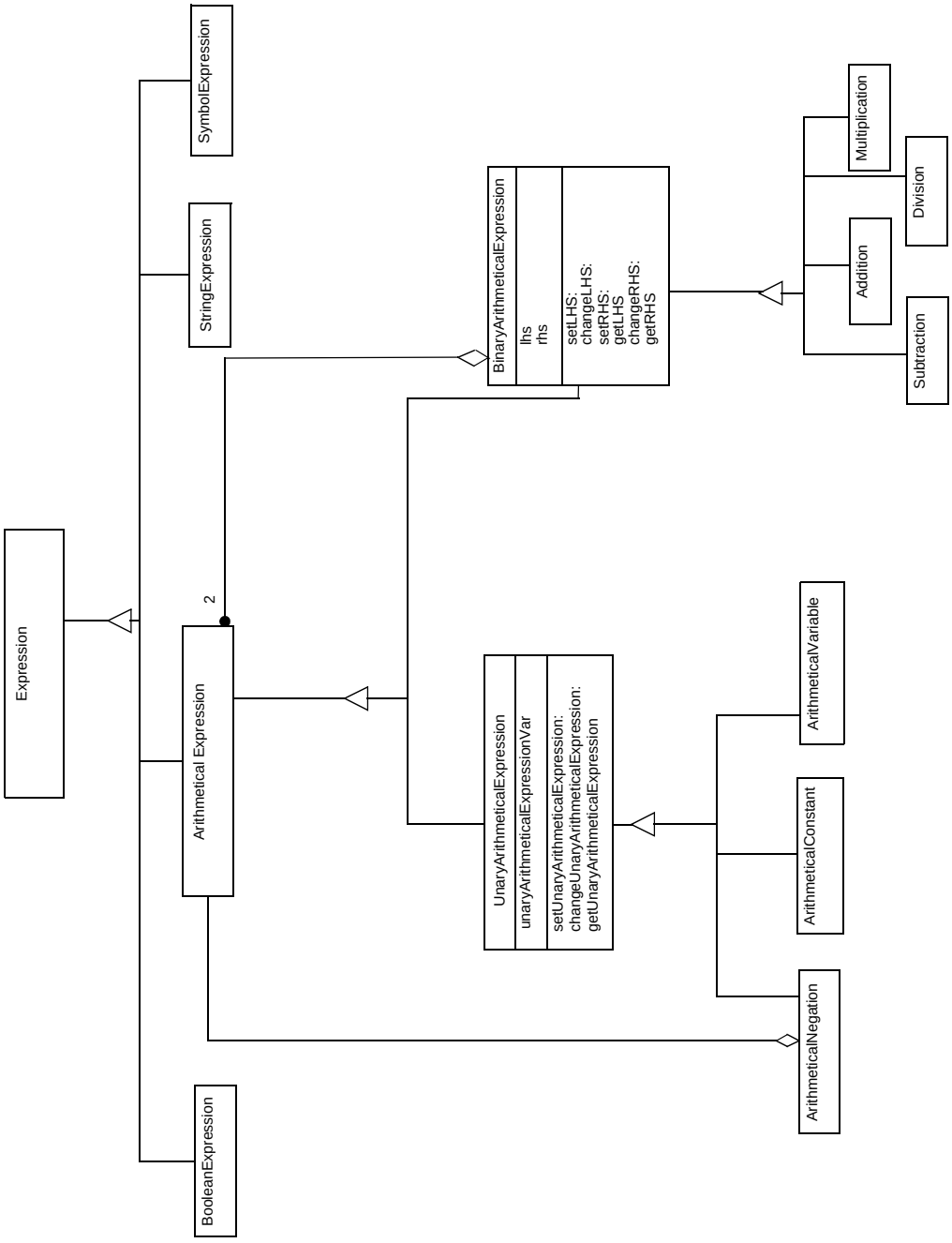
Regeln im umfassenderen, also zusammengesetzten Produkt beschrieben.

ABBILDUNG 11 Hierarchie für boolsche Ausdrücke



Zur Veranschaulichung dieser Beziehungen ein kleines Beispiel:
Der komplexe Produkttyp Softwareprogrammpaket besteht unter

ABBILDUNG 12 Hierarchie für arithmetische Ausdrücke



anderem aus dem Teilprodukt Gesamtquellcode sowie aus zwei Teilprodukten Modul 1 und Modul 2. Beide haben wiederum als Teilprodukte ihren Gesamtquellcode sowie den Definitionsquellcode und den Implementierungsquellcode. Als Attribute besitzen sie den Umfang, die Anzahl der Funktionen und die Komplexität. Zur Beziehung Attribut/Attribut auf einer Ebene kann man nun als Beispiel folgendes angeben: die Komplexität berechnet sich durch die Anzahl

der Funktionen des Moduls. Auf zwei aneinandergrenzenden Ebenen berechnet sich die Gesamtkomplexität durch die Addition der beiden Modulkomplexitäten. Eine Beziehung zwischen Produkt/Attribut ergibt sich durch den Umfang, der sich dann durch die Größe des Gesamtquellcodes des Moduls berechnet. Zwischen den Teilprodukten besteht eine Beziehung, wenn sich der Modulgesamtquellcode aus dem Definitionsquellcode und dem Implementierungsquellcode des Moduls berechnet. Somit ergibt sich auf zwei Verfeinerungsebenen noch die Berechnung des Gesamtquellcode des Programmes durch Summierung der beiden Gesamtquellcodes in das Programmpaket.

Für spätere Versionen sind auch weitere Beziehungen vorgesehen, wie zum Beispiel Constraints zwischen Attributen und Teilprodukten.

Vererbung

Komplexe Produkttypen stellen den Mechanismus der Spezialisierung zur Verfügung. Produktalternativen werden durch eine Vererbungshierarchie modelliert. Ein Obertyp vererbt gemeinsame Teilprodukte und Attribute, die zwar eingeschränkt aber nicht anderweitig verändert werden dürfen. Neue Teilprodukte und Attribute können hinzugefügt werden. Es ist nur einfache Vererbung erlaubt, das heißt es dürfen nur Teilprodukte und Attribute aus einer Oberklasse, also von einem Produkttyp, geerbt werden. Hierzu ein kleines Beispiel:

Der komplexe Produkttyp „Dokument“ besteht aus folgenden Teilprodukten:

- *Dokumenttitel* vom Typ String, nicht optional
- *Versionsnummer* vom Typ Real, nicht optional
- *Erstellungsdatum* vom Typ String, optional

Zusätzlich wird ein Produkttyp „Anforderungsdokument“ definiert. Dieser hat als Supertyp den Typ „Dokument“ und besteht aus den Teilprodukten Abteilung, Inhaltsverzeichnis und Anforderungen. Hier sieht man, wie die Vererbung die Modellierung unterstützt. Die Teilprodukte Dokumenttitel, Versionsnummer und Erstellungsdatum müssen nicht erneut definiert werden, egal um welche Art von Dokument es sich bei späterer Modellierung handelt. Ein weiterer Vorteil liegt in der Tatsache, daß man die Vererbungshierarchie als Zugriffsstruktur für eine Erfahrungsdatenbank von Softwareentwicklungsprozessen nutzen kann. In einer Erfahrungsdatenbank kann man so zunächst alle Entwurfsdokumente abfragen und anschließend alle objekt-orientierten Entwurfsdokumente oder alle objekt-orientierten Entwurfsdokumente für Gebäudesteuerungen. Der Vorteil von der

Vererbung ist hier, daß zwischen Objekten unterschiedlicher Abstraktionsniveaus ein Zusammenhang hergestellt wird.

Für die Ausführungsmaschine ist auch vorgesehen, daß Produkte spezialisiert werden können. Sie werden dann Instanzen einer Unterklasse und können zusätzliche Teilprodukte und Attribute enthalten. Hat man zum Beispiel einen Prozeß „Entwurfsdokument erstellen“, dann möchte man vielleicht erst zur Ausführungszeit festlegen, ob es sich um ein funktionales oder ein objektorientiertes Entwurfsdokument handelt. Allerdings kann man dieses auch durch eine weitere Methodendefinition für den Prozeß erledigen, indem man einfach die Methoden „Funktionalen Entwurf erstellen“ und „Objektorientierten Entwurf erstellen“ generiert. Dazu mehr bei der Definition der Prozesse und Methoden.

2.2 Prozesse

Durch Prozesse werden Aufgaben beschrieben, also *was* getan werden muß. Für diese Aufgaben werden Methoden definiert, die einen Lösungsweg beschreiben. Bei der hier beschriebenen Softwareentwicklungssprache existiert genau ein Top-Level-Prozeß, der dann verfeinert wird. Bei Prozessen unterscheidet man auf der konzeptuellen Ebene Prozeßtypen und Prozeßdefinitionen. Eine Prozeßdefinition ist im wesentlichen eine Kopie eines Prozeßtyps. Die Prozeßtypen bilden die Basis für die Planung, sie spiegeln die Erfahrung mit vorherigen Projekten wider. Die Prozeßdefinitionen passen die Typen den Projektanforderungen an. Bei Ablauf des Projektes werden die Prozeßdefinitionen instantiiert. Es ist vorgesehen, daß die Prozeßdefinitionen nach Beendigung des Projekts überprüft werden und bei Tauglichkeit in Prozeßtypen umgewandelt werden.

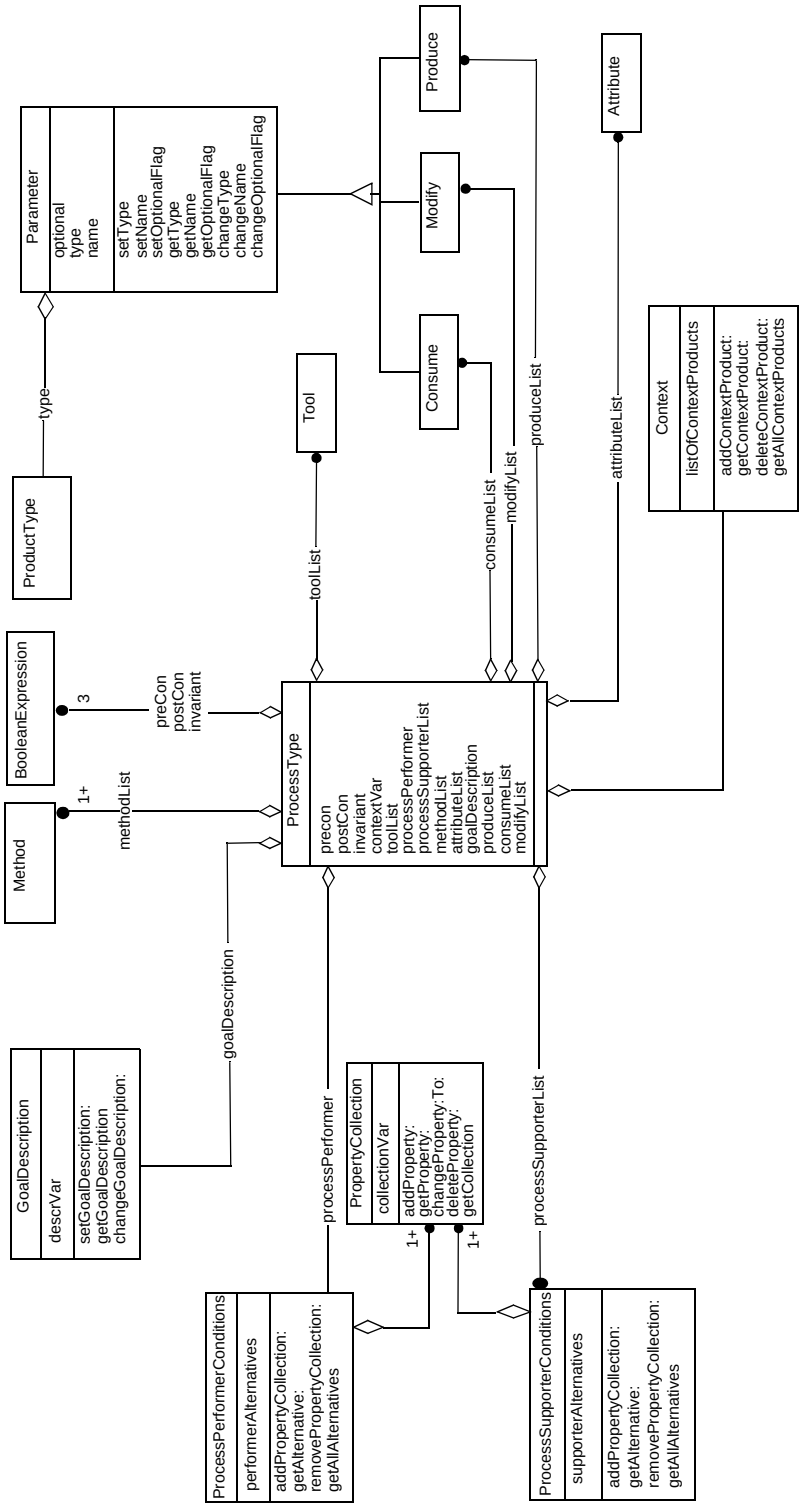
2.2.1 Prozeßtypen

Prozeßtypen bestehen neben einem Namen aus den im folgenden beschriebenen Komponenten, die in Abbildung 13 skizziert sind. Der Übersichtlichkeit halber fehlen die Methoden in der Klasse *ProcessType*, die für das Datenmodell nicht wichtig sind, und die Spezifikation der Ressourcen, Produkte und Methoden wird nur skizziert, da diese Strukturen in anderen Teilkapiteln ausführlich behandelt werden.

Zielbeschreibung

Hierbei handelt es sich um eine textuelle Beschreibung des Ziels, welches durch den Prozeß erreicht werden soll. So hat zum Beispiel der Prozeß „Testpläne entwickeln“ das Ziel Testpläne zu entwickeln, um Fehlverhalten des Programms zu entdecken und so Rückschlüsse auf die Fehlerursachen im Programm ziehen zu können.

ABBILDUNG 13 Hierarchie für Prozeßtypen



Kontextinformation

Kontextinformation geht in Form der in Kapitel 2.1.1 beschriebenen Kontextprodukte ein. Sie besteht aus einem Namen und einer Referenz auf ein Kontextprodukt.

Parameter

Prozesse haben eine Liste von konsumierten, eine Liste von produzierten und eine Liste von modifizierten Produkten. Hierbei handelt es sich während der Modellierung um Parameter der Form
Name: Produkttyp,
die dann zur Ausführungszeit mit konkreten Produkten belegt werden. Konsumierte Produkte können vom Prozeß lediglich gelesen werden. Modifizierte Produkte, können sowohl gelesen als auch geschrieben werden. Produzierte Produkte werden vom Prozeß neu erzeugt.

Konsumierte und modifizierte Produkte werden durch einen pro Prozeß eindeutigen Namen, einen Produkttyp und eine Optionalitätsangabe beschrieben. Das Optionalitätsflag gibt an, ob das in den Prozeß eingehende Produkt vorhanden sein muß bevor der Prozeß bearbeitet werden kann. Dieses ist unter anderem für die Prozeßverfeinerung von Bedeutung. Möchte man beispielsweise den Prozeß Komponententesten durchführen, so konsumiert man eine Komponentenspezifikation, den Quellkode und die ausführbare Komponente. Welche Parameter man braucht, hängt von der Prozeßverfeinerung ab. Der Produkttyp Spezifikation ist für Tests generell notwendig, muß also in den Prozeß eingehen. Die beiden anderen Parameter sind auf dieser Prozeßebene optional. Geht man Beispielsweise von den beiden Prozeßverfeinerungen „Funktionales Testen“ (Black-Box-Test) und „Strukturelles Testen“ (White-Box-Test) aus, so braucht man für die erste Variante eine ausführbare Komponente und keinen Quellkode. Bei der zweiten Verfeinerung hingegen verhält es sich genau entgegengesetzt. Die Parameter werden also auf der Verfeinerungsebene notwendig beziehungsweise entfallen.

Produzierte Produkte werden durch den Prozeß erzeugt und an seine Umgebung abgegeben. Im allgemeinen werden diese Produkte von anderen Prozessen konsumiert oder modifiziert. Sie werden beschrieben durch Namen, Typ, Optionalitätsangabe und Initialisierung. Die Optionalitätsangabe gibt analog zu den konsumierten und modifizierten Produkten an, ob das Produkt produziert werden kann oder muß. Die Initialisierung stellt die Möglichkeit zur Verfügung, daß verschiedene Slot- und Attributtypen bereits bei der Modellierung initiale Werte erhalten. Beispielsweise kann zum Zeitpunkt der Prozeßabwicklung eines Prozesses „Testen“ ein Objekt vom Typ Fehlerbeschreibungsdokument mit einer initialen Maske belegt werden.

Bedingungen

Für Prozesse können drei verschiedene Arten von Bedingungen angegeben werden, die durch logische Ausdrücke realisiert werden. Die *Vorbedingung* muß erfüllt sein, damit der Prozeß gestartet werden kann. Die *Invariante* muß während der Prozeßabwicklung erfüllt sein. Ist das nicht der Fall, so wird der Prozeß abgebrochen. Die *Nachbedingung* muß erfüllt sein, damit der Prozeß beendet werden kann. Vorbedingungen beziehen sich beispielsweise auf eingehende Produkte, die vollständig vorliegen müssen bevor der Prozeß starten kann. So ist zum Beispiel „Funktionales Testen“ nicht möglich wenn noch keine lauffähige ausführbare Komponente vorhanden ist.
Eine mögliche Invariante für einen Prozeß sieht so aus, daß der Prozeß ein bestimmtes Budget nicht überschreiten darf. Wenn der Prozeß dann abgebrochen wird, wird eine Umplanung der gesamten Projektabwicklung notwendig.
Nachbedingungen müssen erst dann erfüllt sein, wenn die notwendigen Modifikationen an allen zu bearbeitenden Produkten abgeschlossen sind.
Bei den Bedingungen wird deutlich, daß man mit ihnen die Möglichkeit der Beschreibung eines impliziten Kontrollflusses an die Hand bekommt, den man in die Prozeßabwicklung integrieren kann.

Attribute

Attribute von Prozessen entsprechen Eigenschaften, die Aussagen über Prozesse machen. Ein Beispiel für ein Prozeßattribut ist die Produktivität des Prozesses, die man über das Zeitverhalten definieren kann. Die Typhierarchie für Produkt- und Prozeßattribute ist dieselbe.

Toolbindung

Ein Tool ist ein Werkzeug, daß die Durchführung eines Prozesses unterstützt. Es wird beschrieben durch einen Namen und eine Sequenz von Anweisungen, mit denen das Tool automatisch aufgerufen werden kann, beispielsweise eine Batch-Datei.

Agentenbindung

Während der Ausführung des Projektes wickeln Agenten Prozesse ab. Hierbei gibt es einen Process Performer und eine Menge von Process Supportern. Process Performer und Process Supporter werden bei der Prozeßmodellierung durch *Bedingungen* beschrieben.

Agenten

Ein Agent ist ein menschlicher Bearbeiter, im folgenden kurz *Actor* genannt oder eine Maschine. Bei einem Actor handelt es sich um eine

Person, die an der Durchführung eines Prozesses beteiligt ist, eine Maschine hingegen dient der Abwicklung eines automatisierten Prozesses. Das Agentenkonzept wird ausführlicher beschrieben in Abschnitt 2.4.2, “Agenten”.

Process Performer und Process Supporter

Zu jeden Prozeß gehört genau ein ausgezeichneter Actor, der für die Durchführung des Prozesses verantwortlich ist, der „Process Performer“. Er wird dem Prozeß zur Abwicklungszeit zugeordnet. Seine Verantwortung bezieht sich auf folgende Bereiche:

- Einhaltung der Zeit und Kostenrestriktionen
- Befolgung von Bedingungen der Prozeßbeschreibung
- Ordnungsgemäße Erstellung der Produkte
- Bei Dekomposition des Prozesses in mehrere Unterprozesse: Zuordnen von Process Performern zu Methoden und Prozessen.

  Diese Zuordnung bezieht sich auf den Prozeß, für den der Actor Process Performer ist.

Neben den Process Performern gibt es noch Process Supporter. Bei ihnen handelt es sich um eine Menge von Agenten, die den Process Performer bei der Durchführung des Prozesses unterstützen. Auch sie werden dem Prozeß zur Abwicklungszeit zugeordnet.

Bedingungen zur Beschreibung der Performer und Supporter

Bei der Modellierung steht noch nicht fest, welcher Agent an den Prozessen als Performer oder Supporter arbeitet. Man definiert jedoch Bedingungen, die erfüllt werden müssen, damit ein Agent an einen Prozeß gebunden werden kann. Nur Agenten, die diese Bedingungen erfüllen, können zur Abwicklungszeit den Prozeß als Performer durchführen oder als Supporter unterstützen.

Bedingungen werden mit Hilfe von speziellen boolschen Ausdrücken beschrieben. Ein spezieller boolscher Ausdruck ist ein Literal oder eine Konjunktion von Literalen oder eine Disjunktion von Konjunktionen von Literalen. Ein Literal ist ein Atom oder die Negation eines Atoms. Ein Atom ist eine Eigenschaft (Property) eines Agenten wie zum Beispiel Rollen, Qualifikationen oder die Position in der Organisationsstruktur des Unternehmens. Ein Beispiel für ein negatives Atom ist zum Beispiel NOT(Abteilung(C1)), wenn man möchte, daß der Process Performer für einen Prozeß nicht aus der Abteilung C1 kommt.

Zur Ausführungszeit werden die Ausdrücke ausgewertet und nur die Agenten, welche die boolschen Ausdrücke erfüllen, kommen für die Aufgabe in Frage. Die speziellen boolschen Ausdrücke haben fol-

gende Motivation: Ein einziges Literal ist wenig einschränkend. Für ein Projekt, daß in der Arbeitsgruppe von Prof. Richter durchgeführt wird, ist die Bedingung *Mitarbeiter(Richter)* ein Literal, daß allerdings wenig einschränkend ist. Handelt es sich um eine Programmieraufgabe in Smalltalk, so kann man eine Konjunktion von Literalen erstellen, etwa
Mitarbeiter(Richter) und Programmiersprachen (Smalltalk).
Weiß man noch nicht genau, in welcher Programmiersprache programmiert werden soll, so kann man eine *Alternative* angeben, sodas man eine Disjunktion einer Konjunktion von Literalen erhält:
(Mitarbeiter(Richter) und Programmiersprachen (Smalltalk)) oder
(Mitarbeiter(Richter) und Programmiersprachen (C++)),
wobei die Bedingung wieder abgeschwächt wird. Das bedeutet, daß wieder mehr Actors aus der AG für diese Aufgabe in Frage kommen. Im Laufe der Prozeßverfeinerung werden die Bedingungen so verfeinert, daß die Anzahl der Agenten, die Process Performer werden können, immer weiter eingeschränkt wird. Die oben beschriebene Disjunktion wird wahrscheinlich schon auf der nächsten Verfeinerungsstufe aufgelöst, nämlich genau dann, wenn man sich für eine Vorgehensweise entschieden hat.

Für den Performer reicht also eine Bedingung in der oben angegebenen Form zur Beschreibung aus. Bei Supportern hingegen handelt es sich um eine Liste von Bedingungen, da verschiedene Typen von Supportern für die Aufgabe benötigt werden. Jedes Element der Liste steht dann für einen Supportertyp. Die Zahl der Supporter wird also durch die Anzahl der Supportertypdefinitionen im Modell festgelegt. Ein Problem besteht hier, wenn man beispielsweise für eine Prozeß „Komponentenkode erzeugen“ gerne zehn Process Supporter haben möchte, die C++ programmieren können. Da es bei den Bedingungen keine Möglichkeit gibt eine Kardinalität für einen Supportertyp anzugeben, müßte man die gleiche Supporterbedingung zehn mal in die Supporterliste schreiben. Nicht mehr auszudrücken ist der Bedarf von „möglichst vielen“ oder „zwischen zwei und sechs“ Programmierern. Die Einführung einer Kardinalität würde die Modellierung vereinfachen.

Bei der Abwicklung können einem Prozeß nur Einzelpersonen zugeordnet werden, da es keine Gruppentypen gibt. Man kann die Zugehörigkeit zu einer Gruppe aber als Eigenschaft definieren und diese Eigenschaft in die Bedingung mitaufnehmen, so wie das in obigen Beispiel auch angedeutet ist.

Kommentierung

Jeder Prozeß kann kommentiert werden. Sinnvoll sind hierbei vor allem die logischen Bedingungen, die ohne Kommentare bei zunehmender Komplexität schwer zu verstehen sind.

2.2.2 Prozeßdefinitionen

Prozeßdefinitionen bestehen konzeptionell aus einem Verweis auf einen Prozeßtyp und Änderungen an diesem Prozeßtyp. Hierbei muß noch festgelegt werden, welche Änderungen zulässig sind.

2.3 Methoden

Methoden beschreiben einen Lösungsweg für einen Prozeß. Für jeden Prozeß wird während der Abwicklung genau eine Methode ausgewählt. Eine komplexe Methode besteht wiederum aus Prozessen, die alle ausgeführt werden müssen, damit die Methode als vollständig bearbeitet abgeschlossen werden kann. Einen ersten Überblick über den Aufbau von Methoden liefert Abbildung 14. Jede Methode ist genau einem Prozeßtyp zugeordnet. Man kann für Methoden analog zur Vorgehensweise bei Produkten Regeln definieren, die beispielsweise Einfluß auf die Ausführung der Methode haben oder Attributwertaggregationen durchführen. Außerdem können für jede Methode die Bedingungen für den Process Performer und die Process Supporter, die im übergeordneten Prozeß definiert sind, eingeschränkt werden.

Man unterscheidet zwischen atomaren und komplexen Methoden.

2.3.1 Atomare Methoden

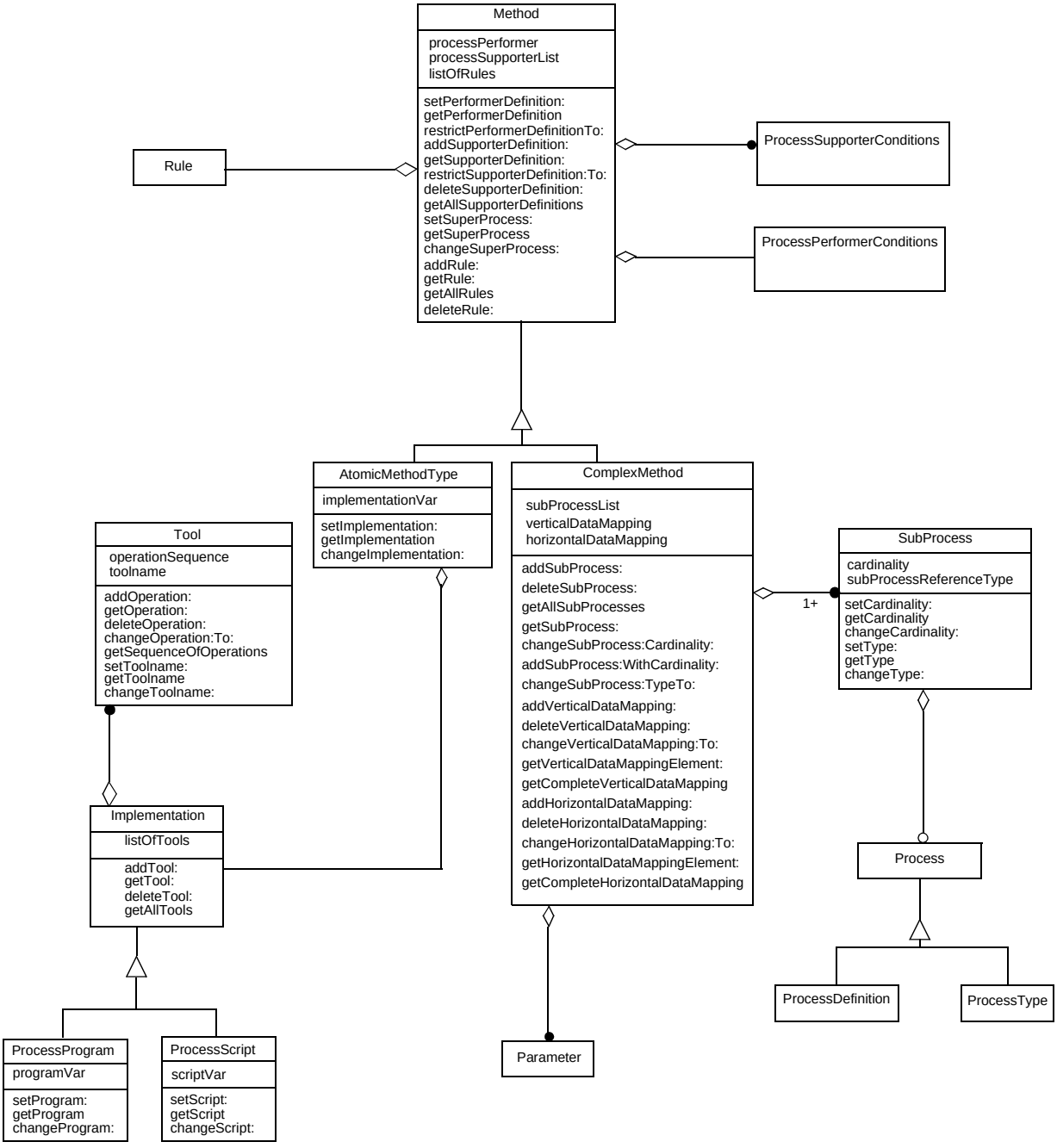
Atomare Methoden bilden die „Blätter“ des gesamten Software-Entwicklungsprozeßmodells. Sie bestehen aus einer Agentenbindung, wie sie für die Prozeßtypen bereits beschrieben wurde, und einer *Implementation*. Das bedeutet, daß für einen Actor ein *ProcessScript* und für eine Maschine ein *ProcessProgram* existiert. Das ProcessScript gibt dem Actor die Vorgehensweise bei der Methodenabwicklung vor, das ProcessProgram ist ein von einer Maschine abzuwikkelndes Programm.

2.3.2 Komplexe Methoden

Komplexe Methoden setzen sich aus einer Verfeinerung in Teilprozesse, Regeln und einer horizontalen und vertikalen Datenabbildung zusammen.

Wie oben bereits erwähnt, müssen alle Prozesse einer Methodenzerlegung abgewickelt werden. Es kann durchaus sein, daß ein Prozeß mehrfach abgewickelt werden muß. Daher wird jedem Teilprozeß eine Kardinalität mitgegeben, die mindestens eins sein muß, aber auch beliebig sein kann. Ein einfaches Beispiel hierfür liefert die Methode, nach welchem Lebenszyklusmodell man ein Softwareprojekt abwickeln möchte. Wenn man zum Beispiel das Modell des „Ite-

ABBILDUNG 14 Hierarchie für Methoden



rative Enhancement“ anwendet, so werden die Schritte des Entwicklungsprozesses n-mal angewendet.

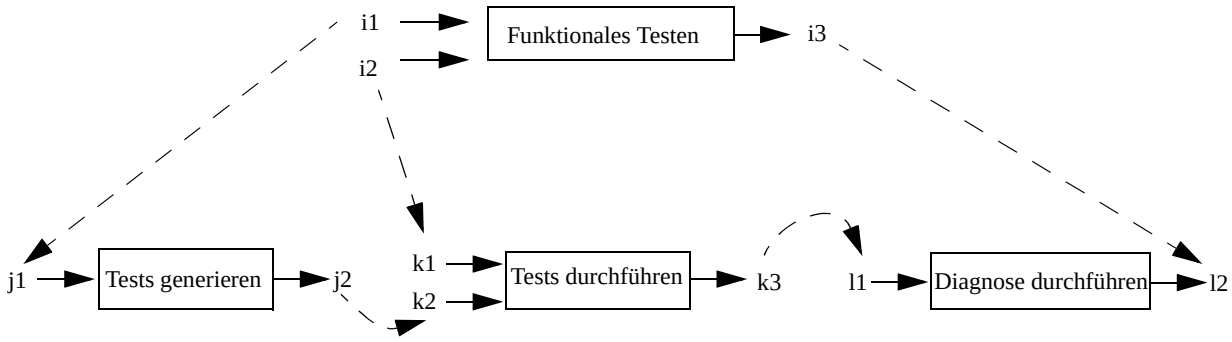
Regeln in Methoden haben denselben Aufbau wie Regeln in Produkten. Es können Regeln zwischen dem übergeordneten Prozeß und den Prozessen, durch die dieser unmittelbar verfeinert wird, definiert werden. Hierbei kann es sich beispielsweise um eine Aggregation von

Prozeßattributen, die von der Wahl einer Methode abhängt. Außerdem können Regeln innerhalb eines verfeinerten Prozesses definiert werden. Ein Beispiel hierfür ist die Abhängigkeit eines Prozeßattributs von anderen Prozeßattributen.

Wozu benötigt man Datenabbildungen? Wenn Prozesse in Teilprozesse verfeinert werden, so werden für diesen Prozeß *lokal* die Menge der konsumierten, produzierten und modifizierten Parameter definiert. Sie lassen sich somit einfach in einer Bibliothek ablegen und können wiederverwendet werden. Konsumiert ein Prozeß einen produzierten Parameter eines anderen Prozesses, so muß man diese Beziehung festlegen, also die Parameter aufeinander abbilden. Diese horizontale Datenabbildung wird in der Methode beschrieben, ebenso wie die vertikale Abbildung. Hier wird bei einer Prozeßverfeinerung angegeben, welche Parameter des Prozesses in welche Prozeßverfeinerungen eingehen.

Die horizontale und die vertikale Datenabbildung lassen sich am besten anhand Abbildung 15 zeigen. Der Prozeß „Funktionales

ABBILDUNG 15 Horizontaler und vertikaler Datenfluß



Agenda: Gestrichelte Pfeile symbolisieren Datenfluß
Produkte,Produkte::Produkttypen
i1,j1::Komponentenspezifikation
i2,k1::Ausführbare Komponente
i3,l2::Fehlerbericht
j2,k2::Tests
k3,l1::Testsverlaufsprotokoll

Testen“ wird durch die Methode „Standard Vorgehensweise“ abgearbeitet. Diese Methode beinhaltet die drei Prozesse „Tests generie-

ren“, „Tests durchführen“ und „Diagnose“. Die Produkte, die von dem Prozeß „Funktionales Testen“ konsumiert werden, gehen nun auch in die verfeinerten Prozesse ein. Dieses ist zum einen der Parameter i1 vom Typ „Komponentenspezifikation“ und zum anderen der Parameter i2 vom Typ „Ausführbare Komponente“. Bei der Verfeinerung hat man andere Parameternamen aber dieselben Parametertypen, also dieselben Produkttypen. Damit man weiß, welche Parameter auf den beiden Verfeinerungsebenen übereinstimmen, muß man eine vertikale Zuordnung der einzelnen Parameter durchführen. So wird hier beispielsweise der Parameter i1 des Prozesses „Funktionales Testen“ dem Parameter j1 des Prozesses „Tests generieren“ vertikal zugeordnet. Für Prozesse auf einer Verfeinerungsebene muß man zusätzlich angeben, welche produzierten Parameter als konsumierte Parameter für andere Prozesse benutzt werden. Auch dieses Wissen spiegelt sich in der Methode wider. Man bezeichnet diese Zuordnung als horizontale Datenabbildung. Hier wird unter anderem das produzierte Produkt j2 des Prozesses „Tests generieren“ dem konsumierten Produkt k2 des Prozesses „Tests_durchführen“ zugeordnet.
Bei obigem Beispiel kann man sich vielleicht noch denken, daß eine Zuordnung automatisch generiert werden kann.Die Notwendigkeit der manuellen Zuordnung in den Methoden wird jedoch bei mehreren Parameter desselben Produkttyps ersichtlich, da in diesem Fall bei unterschiedlichen Produktnamen keine automatische, eindeutige Zuordnung mehr durchgeführt werden kann.

2.4 Ressourcen

Ressourcen werden während eines Software-Entwicklungsprozesses benötigt, um die anfallenden Arbeitsprozesse abzuwickeln. Einen ersten Überblick verschafft Abbildung 16.

Die Ressourcen, die für einen Softwareentwicklungsprozeß zur Verfügung stehen, lassen sich unterscheiden nach *Agents* und *Tools*. Agenten sind Actors oder Maschinen wie sie auch schon bei den Prozessen kurz beschrieben wurden, Tools hingegen sind Werkzeuge, die den Entwicklungsprozeß passiv unterstützen sollen. Im folgenden werden die Charakteristika der beiden Gruppen kurz beschrieben.

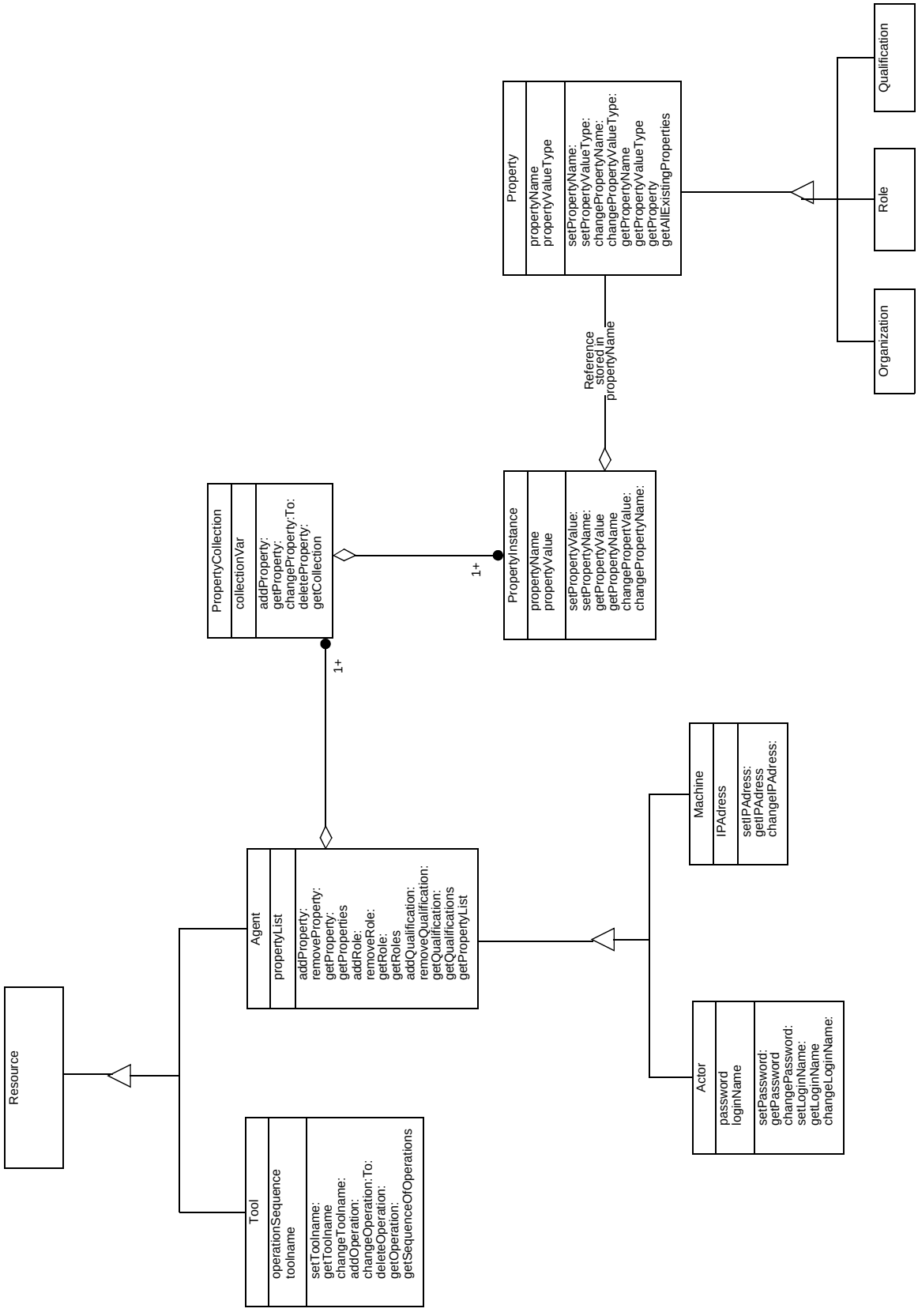
2.4.1 Tools

Tools werden charakterisiert durch einen Namen und eine Sequenz von Anweisungen, mit denen das Tool aufgerufen werden kann. Beispiele hierfür sind UNIX-Skripte.

2.4.2 Agenten

Agenten zeichnen sich durch eine Menge von Eigenschaften, im weiteren auch Properties genannt, aus. Es gibt feste und variable Eigen-

ABBILDUNG 16 Hierarchie der Ressourcen



schaften. Feste Properties sind vom System mit einer bestimmten Bedeutung versehen. So muß beispielsweise jeder Agent einen Namen haben. Variable Properties wie Programmiersprachenkenntnisse können auch während der Abwicklung noch zur Menge der Properties des Agenten hinzukommen.

Variable Properties werden einer modellierten Property-Sammlung entnommen. Die Properties in dieser Sammlung sind nach sinnvollen Kriterien gruppiert, wie etwa nach Programmiersprachen.

Die Property-Sammlung wird vor und während der Modellierung erstellt und die Agenten werden auch zu diesem Zeitpunkt mit Properties ausgestattet. Prozeß- und produktbezogene Attribute beschreiben ebenfalls Eigenschaften, aber im Gegensatz zu den Properties der Agenten werden diese erst zur Ausführungszeit belegt.

Die Process Performer- und Process Supporter-Bedingungen, die bei Prozessen und Methoden spezifiziert werden, beziehen sich auf die Property-Sammlung.

Variable Properties haben einen Wert, der die Ausprägung einer Eigenschaft beschreibt. Das ist sinnvoll, da es sich bei Properties oftmals nicht um binäre Ausprägungen handelt und so eine differenziertere Aussage gemacht werden kann. Der Typ dieses Wertes wird bei den Propertydefinition festgelegt, eine Ausprägung des Typs wird bei Zuordnung zu einem Agenten festgelegt. Falls keine Ausprägung angegeben wird, so wird ein Default-Wert eingetragen. So ist es zum Beispiel sinnvoll, C++-Programmiersprachenkenntnisse mit einer Ordinalskala von „Sehr gut“ bis „nicht vorhanden“ zu typisieren. Wenn kein Wert angegeben wird, dann wird die Ausprägung „mäßig“ als Default-Wert gesetzt.

Rollentypen können auf zwei Arten modelliert werden. Zum einem kann man sagen, daß sich eine Rolle aus einer Menge von Properties zusammensetzt. Es handelt sich dann um eine implizite Rollendefinition. Beispielsweise kann man sagen, daß ein Kodierer

- Kenntnisse über die verwendete Programmiersprache besitzt,
- die Notwendigen Kode-Dokumentationsrichtlinien kennt,
- Qualitätsanforderungen wie Zuverlässigkeit und Komplexität kennt und berücksichtigt,
- mit den benötigten beziehungsweise bereitgestellten Werkzeugen wie Compiler, Debugger und Editoren umgehen kann.

Man kann eine solche Rolle aber auch explizit als Property definieren, wobei man keine Aussage über die benötigten Properties für die Rolle macht. Wie unterscheiden sich nun die Spezialisierungen der Agenten, der Actor und die Maschine?

Actor

Ein Actor wird durch die festen Properties *Name* und *Passwort* sowie durch variable Properties der Bereiche

- Rollen
- Qualifikationen
- Stellung in der Aufbauhierarchie

beschrieben. Zu Rollen zählen beispielsweise technische und organisatorische Rollen, wie Anforderungsingenieur und Qualitätssicherer. Zu Qualifikationen zählen unter anderem erworbene Erfahrungen und Kenntnisse, wie zum Beispiel Programmiersprachen oder Meßtechniken. Die Stellung in der Aufbauhierarchie bezieht sich zum einen auf eine eingenommene Position in der Organisation und zum anderen auf die Zugehörigkeit zu Gruppen innerhalb der Organisation, etwa „Mitglied der Qualitätssicherungsgruppe Omega“.

Maschine

Eine Maschine wird durch die feste Property *Name* sowie variabler Properties der Bereiche

- Leistung
- Rollentypen
- Aufbauorganisation

beschrieben. Zur Leistung zählen hier technische Werte wie etwa „MIPS“ (Million instructions per second). Die Rollentypen beziehen sich auf maschinelle Aspekte wie Kompilierer oder Systemintegrator. Bei der Aufbauorganisation ist beispielweise die Zugehörigkeit zu einer Abteilung vorgesehen.

Damit ist die Beschreibung der Sprachelemente und deren OMT-Spezifikation beendet. Im nächsten Kapitel wird der Implementierungsteil der Diplomarbeit in Smalltalk beschrieben.

KAPITEL 3

Die Implementierung der Sprachkonstrukte zur Erzeugung von Produkttypen

Dieses Kapitel beschreibt die Implementierung der Sprachkonstrukte zur Erzeugung von Produkttypen, also die Umsetzung der in Kapitel 2 beschriebenen Schnittstellenspezifikation für Produkte in Smalltalk-Klassen.

Neben der Festlegung der Konzepte und der Spezifikation der StP-OMT-Schnittstelle zwischen System und Benutzer besteht diese Diplomarbeit in der Implementierung der Spezifikation für Produkttypen in Smalltalk [Büc93, Gol83]. Außerdem wurden die notwendigen Klassen für die Netzwerkstrukturen implementiert. Diese sind für die Verwaltung der Objekte und deren Beziehungen untereinander notwendig. In diesem Kapitel werden neben dem Datenmodell noch die Methoden beschrieben, mit denen man Elemente der erzeugten Klassen manipulieren kann. Oft handelt es sich hierbei nur um Methoden zum laden, setzen und ändern von Objekten (set, get, change), deren Implementierung einfach ist. Bei der nun folgenden Beschreibung der Implementierung wird ausführlicher auf Methoden eingegangen, die nicht diesen Zugriffsstandards entsprechen.

3.1 Implementierung der Netzwerkstrukturen

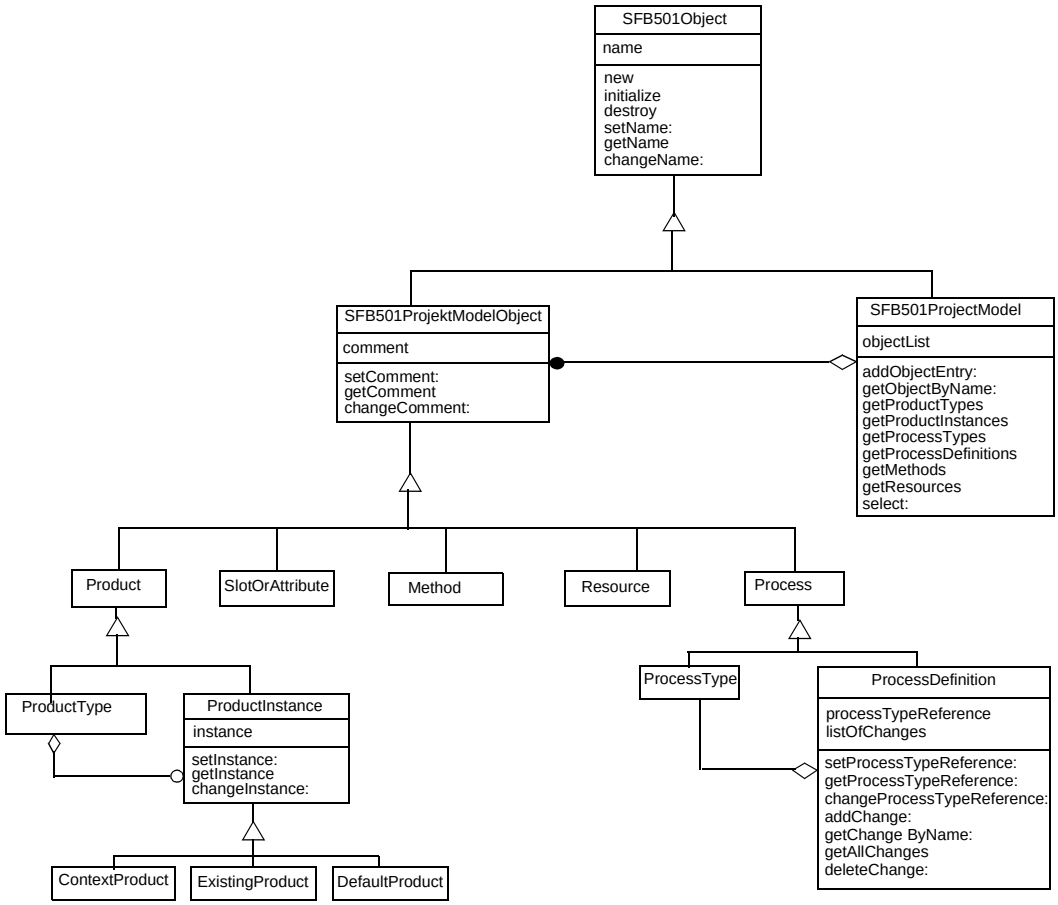
Zunächst stellte sich die Frage, ob man die Sprache komplett neu implementiert oder ob man auf bereits existierenden Strukturen aufsetzt. Die Entscheidung fiel auf die Wiederverwendung bereits existierender Strukturen, da der Implementierungsaufwand bei dieser Lösung wesentlich geringer ist als bei einer kompletten Neuimplementierung. Darüber hinaus haben sich die dort verwendeten Strukturen schon in anderen Anwendungen bewährt. Insbesondere die Verwaltung der Objekte in einem Netzwerk und die Methoden zur Manipulation der Objekte konnten zu einem großen Teil übernommen und wiederverwendet werden. Die Einarbeitung in die bereits

existierenden Strukturen erfordert zwar auch einige Zeit, ist aber im Vergleich zu einer Neuimplementierung gering.

Ein Teil der geforderten Funktionalität der Sprache wird bereits durch existierende CoMo-Kit-Strukturen abgedeckt. Es ist daher oftmals ausreichend, die existierende Klassenstruktur in ihrem Bestand zu erweitern oder verschiedene Methoden zu überschreiben.

Abbildung 17 stellt die Spezifikation der Prozeßmodellierungsspra-

ABBILDUNG 17 Hierarchie der obersten Implementierungsebene



che auf höchster Ebene dar. Die allgemeinste Klasse heißt *SFB501Object*. Hier existieren Klassenmethoden um ein Element zu kreieren (new), zu initialisieren (initialize) und zu zerstören (destroy). Diese Methoden sind allen erzeugten Objekten gemeinsam. Jede Instanz der Klasse *SFB501Object* kennt Methoden um den Namen zu setzen (set), zu ändern (change) und zu lesen (get).

Die Klasse existiert in dieser Form physikalisch nicht. In CoMo-Kit ist ihr Pendant die Klasse `HAMObject`. Sie stellt die geforderten

Eigenschaften zur Verfügung. Die Klasse *SFB501Object* hätte auch neu implementiert werden können. Da aber bereits eine Klasse existiert, die genau diese Funktionalität zur Verfügung stellt, wird auf eine Implementierung verzichtet. Sie ist auch nicht notwendig, da der Benutzer der Prozeßmodellierungssprache niemals Elemente auf dieser Ebene instantiieren wird. Wichtig ist die Vererbung der oben beschriebenen Methoden an die spezialisierten Klassen.

Die nun im folgenden beschriebenen Klassen sind Unterklassen dieser CoMo-Kit-Klasse `HAMObject`.

Spezialisierungen der Klasse *SFB501Object* führen zu den Klassen *SFB501ProjectModelObject* und *SFB501ProjectModel*. Jede Instanz der Klasse *SFB501ProjectModel* verwaltet alle Objekte eines Software-Entwicklungsprozesses. Dieses wird in Abbildung 17 durch eine Aggregationsbeziehung zwischen den Klassen *SFB501ProjectModel* und *SFB501ProjectModelObject* symbolisiert, die aus Null oder mehr aggregierten Objekten besteht. Die Klasse *SFB501ProjectModel* heißt auch in der Implementierung `SFB501-ProjectModel`.

Die zur Verfügung gestellten Instanzenmethoden ermöglichen den Zugriff auf ein bestimmtes Objekt (getObjectWithName:), die Hinzunahme eines neuen Objektes zur Liste der bereits vorhandenen Objekte (addObjectEntry:), außerdem die Extraktion von allen Elementen eines gleichen Typs. Hierzu zählen alle Produkttypen (getProductTypes), Produktinstanzen (getProductInstances), Prozeßtypen (getProcessTypes), Prozeßdefinitionen (getProcessDefinitions), Methoden (getMethods) und Ressourcen (getResources). Die Vorgehensweise bei der Implementierung der Zugriffsmethoden ist einfach. Die Instanz kennt ihre Objekte und fragt diese nach ihrer Klassenzugehörigkeit. Alle Objekte mit der angeforderten Klassenzugehörigkeit werden gesammelt und als Ergebnis zurückgegeben. Die Methode `select:` ermöglicht es dem Benutzer ein Kriterium wie bei einer Datenbankanfrage einzugeben. Die Treffer werden ermittelt gesammelt und zurückgegeben. Über Aufbau des Kriteriums muß noch diskutiert werden.

In der Implementierung ist die Klasse `SFB501ProjectModel` eine Spezialisierung der Klasse `CoMoKitNetwork`. Das hat den Vorteil, daß die Eigenschaften der CoMo-Kit-Netzwerke vollständig übernommen werden können. Beispielsweise kennt die Klasse `CoMoKitNetwork` alle Instanzen ihrer selbst. Das gilt damit auch für die Klasse `SFB501ProjectModel` und somit kann man alle existierenden Projektpläne extrahieren. Anzumerken ist hier noch, daß ein Netzwerk seine Basistypen kennt. Da ein Projekt andere Basistypen kennt als ein CoMo-Kit Netzwerk, werden die Basistypen gemäß der Spezifikation neu implementiert.

Die Klasse *SFB501ProjectModelObject* spezialisiert die Klasse *SFB501Object*, indem sie einen Kommentar und die dazugehörenden Zugriffsmethoden Setzen (setComment:), Lesen (getComment) und Ändern (changeComment) definiert. Diese Klasse existiert ebenfalls nicht. Die Gründe hierfür sind dieselben wie die für die Nichtimplementierung der Klasse *SFB501Object*. Die hier neu definierten Methoden und die Instanzenvariable befinden sich in der Klasse `HAMNetworkObject`.

Die Klasse *SFB501ProjectModelObject* wird nun spezialisiert. Es entstehen zunächst die Klassen *Product*, *Process*, *Method* und *Resource*, die den Grundobjekten der Prozeßmodellierungssprache gemäß der in Kapitel 2 beschriebenen Sprachfestlegung entsprechen.

3.2 Implementierung der Produktobjekte

3.2.1 Produktinstanzen

Es wurde die Klasse `SFBProductInstance` implementiert, die der Klasse *ProductInstance* der Schnittstellenspezifikation entspricht. Für Instanzen dieser Klasse besteht die Möglichkeit, eine Referenz auf eine konkrete Produktinstanz zu setzten (setInstance:), zu lesen (getInstance) und zu ändern (changeInstance:). Es gibt drei verschiedene Spezialisierungen der Klasse `SFBProductInstance`, nämlich `SFBExistingProduct`, `SFBContextProduct` und `SFBDefaultProduct`, die den spezifizierten Klassen *ExistingProduct*, *ContextProduct* und *DefaultProduct* entsprechen.

3.2.2 Produkttypen

Ein Produkttyp kann eine Produktinstanz referenzieren. Dieses ist vorgesehen für die Initialisierung von Produkten. Die Aggregationsbeziehung zwischen Produkttypen und Produktinstanzen wird durch einen Link realisiert. Dieser Link ist ein Zeiger auf eine konkrete Produktinstanz.

Um eine Produktinstanz zu erzeugen wird zunächst aus der modellierten Produkttypinstanz durch Compilieren eine Smalltalk-Klasse erzeugt, die dem modellierten Produkttyp entspricht. Die Instantiierung dieser Klasse ergibt ein Element des Produkttyps, in der verschiedene Slots und Attribute gefüllt sind.

Alle existierenden Produkttypen enthalten neben der oben beschriebenen Referenz auf eine Produktinstanz noch einen Link auf einen Supertyp. Das entspricht dem Vererbungskonzept, das in Kapitel 2.1.2 beschrieben wird. Es stehen Methoden zum Setzen (setSuperType:), Ändern (changeSuperType:), und Holen des abstrakteren Typs (getSuperType) zur Verfügung.

Vordefinierte Produkttypen

Vordefinierte Produkttypen bilden sozusagen den elementaren „Baukasten“ für den Modellierer. Aus diesen Typen und aus den von ihm definierten kann er später komplexere Produkttypen modellieren.

Klassen für die im folgenden beschriebenen Produkttypen existieren teilweise auch in CoMo-Kit. Auf eine Wiederverwendung der bereits vorhandenen CoMo-Kit-Strukturen wird verzichtet, da die Neuimplementierung wenig aufwendig ist und das Typkonzept in CoMo-Kit momentan neu implementiert wird.

Die Implementierung folgt der Schnittstellenspezifikation und unterteilt die vordefinierten Klassen in Basistypen und andere vordefinierte Typen.

Basistypen

Die Implementierung der Basistypen ist einfach. Es wurden lediglich die in Kapitel 2 beschriebenen Klassen erzeugt. Die Methoden zum Erzeugen der Instanzen werden geerbt. Das Erzeugen der Klassen ist wichtig, da sie für die Netzwerkinitialisierung benötigt werden. Bei CoMo-Kit-Netzwerken existieren bereits einige Basistypen. Da für die Objekte der Prozeßmodellierungssprache zusätzlich noch weitere Basistypen existieren, wird die Initialisierungsmethode überschrieben (siehe Kapitel 3.1). Einziges Problem ist der Basistyp `SFBEnumeration`, von dem keine Instanz erzeugt wird, da es nicht einen, sondern beliebige Aufzählungstypen gibt. Für die Definition des Typs steht eine Variable zur Verfügung, die alle möglichen Werte einer Aufzählung speichert. Methoden zum Hinzufügen und Löschen eines Elements, sowie zum Holen des gesamten Aufzählungstyps stehen zur Auswahl. Hierbei besteht wieder dieselbe Problematik wie bei der Implementierung der Methoden zum Setzen der Produktinstanzen. Die Elemente für die Aufzählung sind eine Teilmenge der Klasse `SFBSymbol`. Es müssen daher bereits Instanzen dieser Klasse existieren.

Referenzen

Die Klasse `SFBReference` repräsentiert die Referenzstruktur aus Kapitel 2. In ihr sind drei Instanzenvariablen für einen Editor, einen Reader und eine Referenz zu finden. Es stehen die üblichen Methoden zum Setzen, Ändern und Lesen zur Verfügung. Die Spezialisierten Klassen `SFBGraphic`, `SFBVideo`, `SFBSound`, `SFBWWWPage` und `SFBFile` können ebenfalls gemäß der Spezifikation implementiert werden. Für alle Klassen außer der Klasse `SFBWWWPage` steht noch eine Instanzenvariable mit den notwendigen Methoden zum Setzen, Ändern und Lesen zur Verfügung. Diese Variable stellt eine Eigenschaft der Klasse dar, wie etwa den Typ einer Datei.

Tupel und Tabellen

Die Klasse `SFBTuple` erfüllt die Anforderungen, die in der Srachdefinition in Kapitel 2.1.2 an die Klasse *Tuple* gestellt werden. Sie enthält eine Instanzenvariable, in der die Namen und Typen für die Tupelelemente gespeichert werden. Es gibt die Möglichkeit, einen Spaltentyp hinzuzufügen (addColumn:WithType:), einen Spaltentyp zu löschen (deleteColumn:) und einen Spaltentyp (changeColumnTypeOf:To:) sowie einen Spaltennamen (changeColumnName:To:) zu ändern. Außerdem kann man die komplette Tupelbeschreibung lesen (getTupleDescription).

Die Klasse `SFBTable` bildet die Klasse *Table* aus der Spezifikation ab. Instanzen dieser Klasse haben neben den geerbten Instanzenvariablen noch eine Variable, die eine Referenz auf ein Tupel enthält. Die üblichen Methoden zum Setzen, Ändern und Lesen des Tabellentyps wurden implementiert.

Listen

Die Klasse *List* aus der Schnittstellenspezifikation wird durch die Klasse `SFBList` in der Implementierung dargestellt. Es existieren drei Instanzenvariablen, die für die minimale Anzahl, die maximale Anzahl und den Typ der Elemente stehen. Desweiteren sind Methoden vorhanden, um diese drei Variablen zu Setzen, zu Lesen und zu Ändern.

3.2.3 Komplexe Produkttypen

Die Klasse `SFBComplexProduct` ist eine zentrale Klasse der Implementierung. Sie stellt für ihre Instanzen verschiedene Funktionalitäten zur Verfügung. Ein Slot kann eingefügt (addSlot:) und gelöscht (deleteSlot:) werden. Es können alle (getAllSlots), alle lokalen (getLocalSlots) oder alle geerbten Slots (getInheritedSlots) geholt werden, analog dazu stehen dieselben Methoden für Attribute und für Regeln zur Verfügung.

Ein komplexes Produkt enthält Slots, Attribute und Regeln. Diese werden über Links zu anderen Objekten realisiert. Ein ausgehender Link verweist auf ein anderes Objekt. Ein referenzierender Link geht von einem anderen Objekt aus und verweist auf das betrachtete Objekt. Die Klasse `SFBComplexProduct` erbt, wie alle Klassen, die in der Klassenstruktur unter der Klasse `HAMNetworkObject` liegen, eine Liste, in der alle ausgehenden Links gespeichert sind und eine Liste, die alle referenzierenden Links enthält.

Es gibt verschiedene Typen von Links, die abhängig vom Typ der referenzierten Objekte sind. Ein Objekt der Klasse `SFBComplex-`

`Product` hat neben dem geerbten Supertyp-Link noch drei andere Arten von Links:

- **Attribut-Links**

  Diese Links verweisen gemäß der Sprachdefinition auf Attribute. Diese Attribute haben wiederum einen Typ.
- **Slot-Links**

  Diese Links verweisen auf Teilprodukte. Diese Teilprodukte haben wiederum eine Typ.
- **Rule-Links**

  Diese Links verweisen auf Regeln.

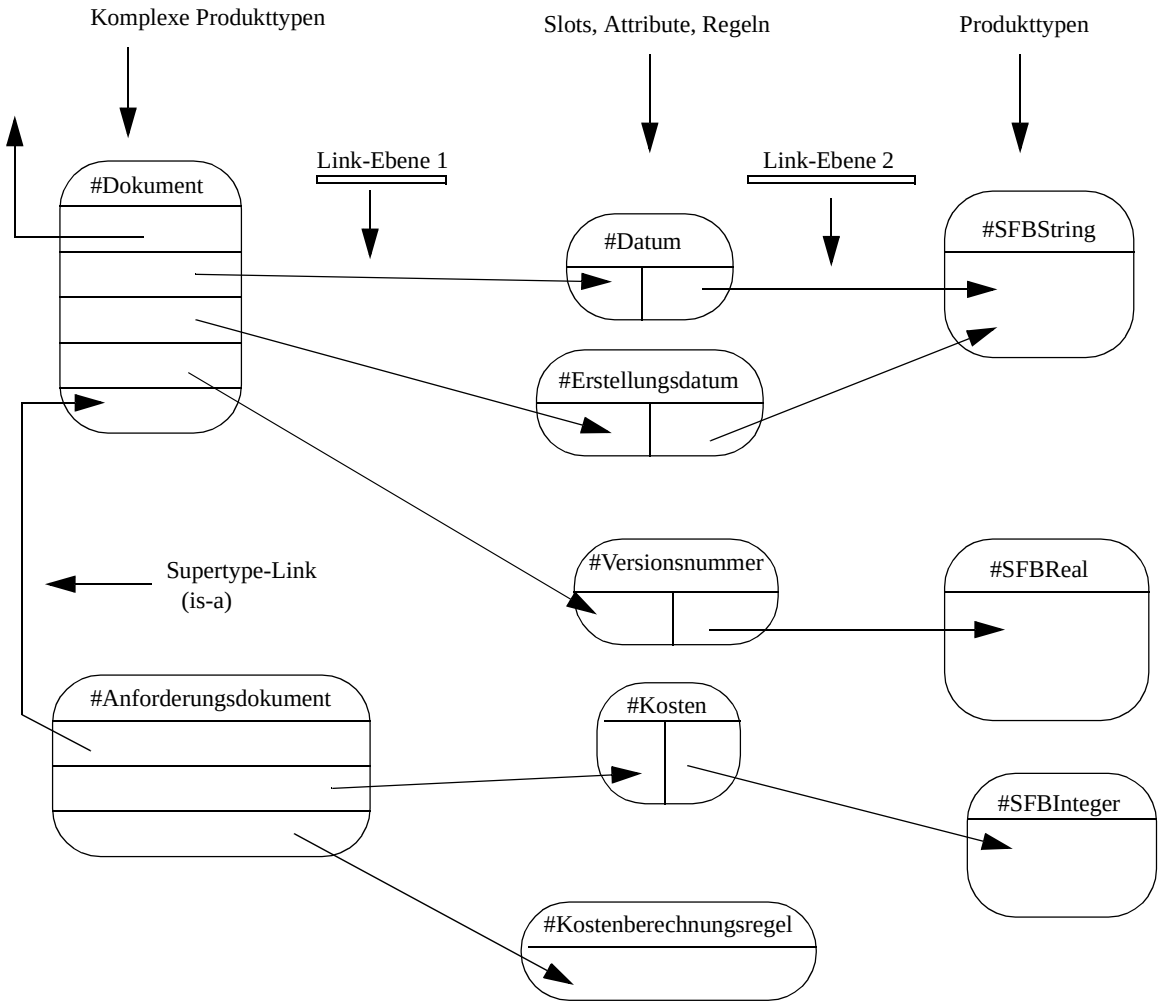
Diese unterschiedlichen Link-Typen spiegeln sich bei der Implementierung durch verschiedene Link-Klassen wider. Das vereinfacht die Selektion von Slots, Attributen und Regeln für komplexe Produkte, da man die Links lediglich auf Zugehörigkeit zu einer der drei Klassen prüfen und dann sammeln muß. Abbildung 18 veranschaulicht die Zusammenhänge. Hierbei bezieht sich die Abbildung auf ein Beispiel aus Kapitel , wo ein Dokumenttyp und ein Anforderunsdokumenttyp beschrieben wurde.

Der Einfachheit halber enthält das Anforderungsdokument lediglich eine Regel und ein Attribut. Bei allen dargestellten Objekten handelt es sich um Objektinstanzen.

Auf der linken Seite der Abbildung befinden sich die komplexen Produkttypen Anforderungsdokument und Dokument. Die Vererbungsbeziehung zwischen diesen Typen wird durch einen Supertyp-Link vom Anforderungsdokument zum Dokument dargestellt. Ein weiterer Supertyp-Link vom Dokument ausgehend deutet an, daß eine weitere Abstraktion dieses Produkttyps möglich ist. Es wird von der Implementierung sichergestellt, daß jeder Produkttyp nur einen Supertyp-Link haben kann.

Von den komplexen Produkttypen ausgehend existieren Links zu Slots, Attributen und Regeln. Sie werden hier als erste Link-Ebene bezeichnet. Bei den oberen drei Links in Abbildung 18 handelt es sich um Slot-Links, das Attribut Kosten wird über einen Attribut-Link und die Kostenberechnungsregeln über einen Rule-Link referenziert. Während es sich bei Slots und Attributen um eine Typbeschreibung handelt, zeigt der Rule-Link auf eine Regel, nicht auf einen Regeltyp. Daher wird bei Regeln keine zweite Link-Ebene mehr benötigt. Die Klasse `SFBRule` ist vorhanden, um Regeln zu entwerfen, die Klasse `SFBRuleLink` enthält die Link-Mechanismen für Regel-Links. Auf die Klasse `SFBRule` wird später noch genauer eingegangen.

ABBILDUNG 18 Realisierung von komplexen Produkten



Die Attribut- und Slot-Links zeigen auf Slots und Attribute. Um diese Instanzen zu erzeugen, müssen die Klassen `SFBSlot` und `SFBAttribute` implementiert werden. Diese Klassen bestehen aus einem Namen, einem ausgehenden und einem eingehenden Link. Die ausgehenden Links beschreiben die zweite Link-Ebene. Hierbei handelt es sich ausschließlich um Typ-Links. Produkttypinstanzen können auch von mehreren Slots referenziert werden. Das sieht man am Beispiel der Instanz der Klasse `SFBString`.

Ein Fall, der in Abbildung 18 nicht dargestellt wird ist der, daß ein Teilprodukt auf einen komplexen Produkttyp verweist. Diese Zyklen werden zugelassen, da solche Strukturen zum Beispiel für rekursive Datentypen benötigt werden. Ein Beispiel für einen solchen Zyklus ist die Definition einer Baumstruktur.

In der Spezifikation für Produkttypen befindet sich neben den Klassen *Slot* und *Attribute* noch die Klasse *SlotOrAttribute*. Sie ist die Generalisierung der beiden Klassen und drückt aus, daß es sich beide Klassen nur gering unterscheiden. Zum einen haben Slots ein Optionalitätsflag und zum anderen sind Slots zusätzliche Typwertebereiche erlaubt. Während für einen Slot alle Produkttypen zugelassen sind, darf ein Attribut nur Basistypen als Wertebereich annehmen. Für die Implementierung reichte es daher aus, lediglich die beiden oben bereits erwähnten Klassen `SFBSlot` und `SFBAttribute` zu generieren. Die Klasse SFBSlot wird als Spezialisierung der Klasse SFBAttribute kreiert, wobei zusätzliche Methoden für die neu definierte Instanzenvariable „optionalityFlag“ hinzukommen und die Methoden zum Setzen (setType:) und Ändern (changeType:) des referenzierten Typs überschrieben werden.

Die übrigen Methoden und Instanzenvariablen werden geerbt, da es sich bei Slots und Attributen um Objekte der Modellierungssprache handelt (siehe Abbildung 17).

Als alternative Implementierungsmöglichkeit zum oben beschriebenen Link-Konzept bei komplexen Produkten bot sich die Implementierung der Slots, Attribute und Regeln über drei dynamische Listen an. Die Entscheidung für diese Alternative hätte aber den Nachteil gehabt, daß die schon in CoMo-Kit vorhandene Link-Struktur nicht hätte wiederverwendet werden können. Diese Struktur hat sich schon lange bewährt und stellt die notwendige Funktionalität zur Verfügung. Um diese Funktionalität komplett zu übernehmen mußte man sich nur in die bereits implementierten Strukturen einarbeiten.

Die Regelkomponente

Die Regelkomponente ist für Produkte, Prozesse und Methoden von Bedeutung. Auch sie wurde im Rahmen dieser Diplomarbeit implementiert. Die ursprüngliche Spezifikation sieht eine Mehrfachvererbung vor. Diese muß auf eine einfache Vererbung „heruntergebrochen“ werden. Das hat Redundanzen zur Folge, da eigentlich gleiche Eigenschaften in verschiedenen Unterklassen erneut implementiert werden müssen.

Die Klasse `SFBRule` besteht aus einer Instanzenvariable für die boolsche Bedingung, einer für die Liste von Aktionen, die im Falle des Zutreffens des boolschen Ausdrucks ausgeführt werden, sowie den Methoden zur Manipulation dieser Variablen. Hierzu zählen das Einfügen einer Aktion in die Aktionsliste (addActionToList:), das Löschen einer Aktion aus der Aktionsliste (deleteActionFromList:), das Ändern einer Aktion (changeActionFromList:) und das Lesen einer Aktion (getActionFromList:) und aller Aktionen (getAllActions). Für den Ausdruck bestehen die üblichen Methoden des Set-

zens, Lesens und Änderns. Da es sich bei einer Regel auch um ein Objekt der Sprache handelt, sind die Methoden zur Erzeugung, Initialisierung und Zerstörung bekannt. Sie werden jedoch, wie bei vielen spezialisierten Klassen, verfeinert beziehungsweise überschrieben.

Elemente der Klasse `SFBAction` gehen in die Aktionsliste von Regeln ein. Die einzige bisher existierende Spezialisierung ist eine Zuweisung, die durch die Klasse `SFBAssignment` realisiert wird. Eine Instanz der Klasse `SFBAssignment` besteht aus einer Zuweisungsvariable und einem Zuweisungswert und den Methoden, um diese zu setzen (setVariable:, setValue:, setVariable:setValue:), zu ändern (changeVariable:, changeValue:) und zu lesen (getVariable, getValue). Über die Typen der Variablen und des Wertes werden in dieser Klasse keine Aussage gemacht, denn es kann sich um Strings, Symbole, Booleans und arithmetische Typen handeln. Die Typkompatibilität muß durch die Ausführungsmaschine sichergestellt werden. Für diese vier Typen muß nun jeweils eine eigene Klassenstruktur erstellt werden. Wichtig für die folgende Betrachtung der verschiedenen Ausdrücke ist die Feststellung, daß die Einteilung auf dem *Ergebnis* des Ausdrucks beruht, nicht auf den eingehenden Operanden.

Boolsche Ausdrücke

Die Klasse `SFBBooleanExpression` steht für diese Ausdrücke. Sie läßt sich in die Klassen `SFBUnaryBooleanExpression`, `SFBBinaryBooleanExpression` und `SFBComlexBooleanExpression` unterteilen.

Die unären Ausdrücke bestehen aus einer Instanzenvariable, die den Ausdruck speichert, sowie den üblichen Bearbeitungsmethoden. Die implementierten Spezialisierungen dieser Klasse heißen `SFBBooleanNegation`, `SFBBooleanConstant` und `SFBBooleanVariable`. Instanzen der letztgenannten Klasse können von Elementen der Klasse `SFBAssignment` als Assignmentvariable referenziert werden oder in komplexere boolsche Ausdrücke eingehen.

Binären Ausdrücke bestehen aus einer linken und einer rechten Seite. Die einzige Spezialisierung der Klasse `SFBBinaryBooleanExpression` besteht aus der Klasse `SFBAtomicBooleanExpression`. Beispiele für atomare Ausdrücke mit Ergebnistyp Boolean sind Vergleichsoperatoren. Diese werden als spezialisierte Klassen implementiert, wie zum Beispiel `SFBBooleanEqual`. Es können auch arithmetische Operanden in diese Ausdrücke eingehen. So kann man beispielsweise zwei Integerzahlen auf Gleichheit testen und erhält einen boolschen Ergebniswert.

Die Klasse `SFBComplexBooleanExpression` stellt eine Ausdrucksvariable in Form einer geordneten Liste und die üblichen Methoden zur Manipulation der Variablen zur Verfügung. Zu komplexen boolschen Ausdrücken zählen die Konjunktion (`SFBKonjunkion`), die Disjunktion (`SFBDisjunktion`) und das Exklusive Order (`SFBExclusiveOr`). Ein solcher komplexer Ausdruck besteht im Prinzip nur noch aus Zeigern auf andere Ausdrücke, die einen boolschen Ergebnistyp haben. Lediglich die Konstanten und Variablen sind konkrete Objekte und keine Zeiger.

Da die Implementierungsstruktur bei den vier möglichen Ausdruckstypen bis auf die unterste Ebene dieselbe ist, wird im Folgenden nur noch auf die Unterschiede zur oben beschriebenen Struktur eingegangen.

Arithmetische Ausdrücke

Die Klasse `SFBArithmeticalExpression` ist die abstrakteste Klasse der arithmetischen Ausdrücke. Die unären Ausdrücke können noch erweitert werden, man kann sich beispielsweise zusätzlich noch die Logarithmus-Funktion vorstellen. Bei den binären Ausdrücken gibt es statt Vergleichsoperatoren Operatoren auf arithmetischen Operanden wie die Multiplikation, die Division, die Addition und die Subtraktion. Eine Erweiterung ist auch hier denkbar. Komplexe Ausdrücke existieren nicht direkt, sondern können über die binären Ausdrücke erzeugt werden, da es sich wie oben bereits erwähnt, nur um Referenzen auf andere Ausdrücke handelt.

String-Ausdrücke

String-Ausdrücke lassen keine binäre und komplexen Operationen zu. Strings dürfen mit allen binären boolschen Ausdrücken verglichen werden, da eine lexikographische Ordnung auf ihnen definiert ist. Bei Wertgleichheit gelten Strings als gleich, es muß sich also nicht um ein einziges Objekt handeln. Als unäre Operation ist die Negation nicht zugelassen.

Symbol-Ausdrücke

Bei den unären Ausdrücken existiert die Symbol-Negation nicht, ebensowenig wie binäre und komplexe Symbol-Ausdrücke. Symbol-Ausdrücke erlauben nur die binären boolschen Operatoren gleich und ungleich. Die Ausführungsmaschine muß für die Einhaltung dieser Restriktionen sorgen. Symbole sind nur gleich, wenn es sich um Pointergleichheit handelt, es sich also wirklich um ein einziges Objekt handelt.

Die so modellierbaren Regeln haben für Software-Entwicklungsprozesse eine ausreichende Mächtigkeit.

Eine Wiederverwendung von bereits vorhandenen Regelstrukturen wäre denkbar gewesen. Allerdings erschien der Aufwand für die Neuimplementierung in diesem Falle geringer und wurde daher vorgezogen.

KAPITEL 4

Exemplarische Modellierung eines Produkttyps

Anhand eines konkreten Produktes wird ein Produkttyp abstrahiert und mit den zur Verfügung stehenden Sprachkonstrukten modelliert.

In Rahmen eines Software Engineering Praktikums der AG Rombach ist ein Softwareprodukt nach den Prinzipien der existierenden Software-Entwicklungsparadigmen entwickelt worden. Als Ergebnis fielen verschiedene Dokumente an wie beispielsweise Komponentenentwurfsdokumente oder Anforderungsdokumente. Hier wird das Anforderungsdokument aus dem Gesamtdokument „Projekt Haustechnik (Inkrement 1) - Klimasteuerung, System-Dokumentation“ betrachtet und die existierenden Grundstrukturen des Dokuments werden abstrahiert. Um die Vererbungsbeziehung aufzuzeigen wird ein Teil des Anforderungsdokuments, der allen im Rahmen des Gesamtdokuments erstellten Dokumenten gemein ist, in einem allgemeineren Produkttyp Dokument zusammengefaßt.

4.1 Erstellen der Strukturen für den Produkttyp Anforderungsdokument

4.1.1 Dokument

Jedes Dokument hat eine Versionsnummer, ein Erstellungsdatum und einen Titel. Über Titel und Versionsnummer kann man ein Dokument eindeutig identifizieren. Es gibt bei der Sprache keinen festen Typ für ein Datum. Daher wird das Datum als String realisiert. Das Datum ist für die Identifikation eines Dokumentes nicht wichtig. Es ist optional. Die Versionsnummer kann man als String, aber auch als Real-Zahl ausdrücken. Bei letzterer besteht allerdings dann nur noch die Möglichkeit der zweistufigen Versionsnummer. Das ist zwar für existie-

rende Werkzeuge wie das SCCS (Source Code Control System) der Unix-Umgebung ausreichend, aber insgesamt zu restriktiv. Daher wird auch hier der Typ String gewählt. Ein Titel ist ebenfalls vom Typ String. Ein Dokument hat keinen Supertyp. Somit ergibt sich folgende Struktur:

Komplexer Produkttyp Dokument

Slots:

- Name:#Titel, Typ:String, Optionalität:False
- Name:#Version, Typ:String, Optionalität:False
- Name:#Datum, Typ:String, Optionalität:True

Attribute: Keine

Supertyp: Nil (Nil ist der Supertyp für alle Typen, die keine Supertyp haben)

4.1.2 Anforderungsdokument

Das Anforderungsdokument besteht unter anderem aus den Namen der Personen, die das Dokument bearbeitet haben. Diese Namen müssen angegeben werden.

Außerdem wird eine Gruppennummer angegeben. Diese ergibt sich aber auch implizit aus den Namen der Bearbeiter (Agenten) über die Property-Liste. Sie ist daher optional.

Jedes Anforderungsdokument hat ein Inhaltsverzeichnis. Dieses ist ebenfalls ein komplexer Produkttyp und wird später definiert. Ein Inhaltsverzeichnis muß vorhanden sein.

Im Anforderungsdokument muß die Problembeschreibung angegeben sein. Diese wird durch einen komplexen Produkttyp beschrieben. Die Problembeschreibung ist nicht optional.

Die Systemanforderungen sind ebenfalls ein Teilprodukt des Anforderungsdokumentes und auch ein komplexer Produkttyp. Auch die Präsenz der Systemanforderungen ist vorgeschrieben.

Die Anforderungen der Kunden sind in einer Anforderungsliste gespeichert. Diese muß mit Hilfe des vordefinierten Typs Table definiert werden. Die Entwickleranforderungen, die nichtfunktionalen Anforderungen und die inversen Anforderungen sind ebenfalls in dieser Form vorhanden. Alle oben genannten Anforderungen sind ein notwendiger Bestandteil des Anforderungsdokuments.

Anforderungsdokumente werden geprüft. Es existieren Testpläne, die diese Validierungsvorgänge festhalten. Sie befinden sich in den Akzeptanz- und in den Systemtestplänen. Diese sind vom Typ Testfall-Liste, der im weiteren Verlauf dieses Kapitels definiert wird. Da das Testen ein wichtiger Vorgang bei der Erstellung eines Dokuments ist, sind die Testpläne notwendig und nicht optional.

Dokumente können verifiziert und validiert werden. Diese Eigenschaften werden für den Anforderungsdokumenttyp als Attribute modelliert.

Da es sich bei dem betrachteten Dokument um ein Praktikumsdokument handelt, fallen bei der Erstellung keine Bearbeitungskosten an. Dieses wäre aber bei Firmendokumenten ein wichtiges Attribut. Daher wird es mit in die Modellierung aufgenommen.

Für ein Anforderungsdokument wird nun eine Regel definiert, um auch dieses Konstrukt abzudecken. Ein einfacher Zusammenhang, der einer Integritätsbedingung gleichkommt, läßt sich zwischen dem Attribut Verifiziert des Anforderungsdokuments und dem Attribut Verifiziert des Teilproduktes Problembeschreibung herstellen. Wenn die Problembeschreibung nicht verifiziert wurde, dann kann das Anforderungsdokument nicht verifiziert werden.

Insgesamt ergibt sich folgende Struktur:

Komplexer Produkttyp Anforderungsdokument

Slots:

- Name:#Bearbeiternamen, Typ:String, Optionalität:False
- Name:#Gruppennummer, Typ:Integer, Optionalität:True
- Name:#Anforderungsdokument-Inhaltsverzeichnis, Typ:Inhaltsverzeichnis, Optionalität:False
- Name:#Anforderungsdokument-Problembeschreibung, Typ:Problembeschreibung, Optionalität:False
- Name:#Anforderungsdokument-Systemanforderungen, Typ:Systemanforderungen, Optionalität:False
- Name:#Kundenanforderungen, Typ:Anforderungstabelle Optionalität:False
- Name:#Entwickleranforderungen, Typ:Anforderungstabelle, Optionalität:False
- Name:#Nichtfunktionale Anforderungen, Typ:Anforderungstabelle, Optionalität:False
- Name:#Inverse Anforderungen, Typ:Anforderungstabelle, Optionalität:False

- Name:#Akzeptanz-Testpläne, Typ:Testfall-Liste, Optionalität:False
- Name: #System-Testpläne, Typ:Testfall-Liste, Optionalität:False

Attribute:

- Name:#Herstellungskosten, Typ:Integer, Optionalität:False
- Name:#Validiert, Typ:Boolean, Optionalität:False
- Name:#Verifiziert, Typ:Boolean, Optionalität:False

Regeln:

- Name: #Verifikationsregel
  Boolscher Ausdruck:
  (Anforderungsdokument-Problembeschreibung.Verifiziert=False)
  Liste von Aktionen:
  1.Assignment: Verifiziert:=False

Su**pertyp**: Dokument

Nun werden die zusätzlich notwendig gewordenen komplexen Typen definiert.

4.1.3 Inhaltsverzeichnis und -elemente

Ein Element eines Inhaltsverzeichnis besteht aus einer Kapitelnummer, einer Überschrift und der Seite, auf der das angegebene Kapitel beginnt. Ein Inhaltsverzeichnis setzt sich aus einer Menge von Inhaltsverzeichniselementen zusammen. Es können noch Attribute hinzukommen. Als Attribut wäre beispielsweise die Anzahl der Kapitel denkbar, dieses wird in diesem Beispiel allerdings nicht modelliert.

Es gibt mehrere Möglichkeiten, den Typ Inhaltsverzeichnis zu modellieren. Man kann den vordefinierten Datentyp Tabelle benutzen, indem man Überschrift, Kapitelnummer und Seite als Elemente eines Tupels definiert und bei einem Tabellentyp auf dieses Tupel verweist. Da es unwahrscheinlich ist, daß Inhaltsverzeichnisse in einer relationalen Datenbank gespeichert werden, wird aber eine andere Möglichkeit gewählt. Es wird ein komplexer Produkttyp Inhaltsverzeichnis-Elemente definiert und die Elemente werden in einer Liste zusammengefaßt. Somit ergibt sich folgender Aufbau:

Listen-Produkttyp Inhaltsverzeichnis

Kardinalität:

- Minimum: 1
- Maximum: unbegrenzt

Typ: Inhaltsverzeichnis-Element

Supertyp: Nil

Komplexer Produkttyp Inhaltsverzeichnis-Element

Slots:

- Name:#Kapitelnummer, Typ:String, Optionalität:False
- Name:#Überschrift, Typ:String, Optionalität:False
- Name:#Seitennummer, Typ:String, Optionalität:False

Attribute: Keine

Regeln: Keine

Supertyp: Nil

4.1.4 Problembeschreibung

Eine Problembeschreibung besteht aus verschiedenen Slots und Attributen. Da eine Problembeschreibung im allgemeinen sehr informell gehalten ist, wird für vorhandenen Slots und Attribute der Typ String verwendet.

Die Repräsentation des Problems enthält eine umgangsprachliche Beschreibung des Problems. Die Umgebungscharakteristika geben beispielsweise an, welche Rahmenbedingungen in der Firma gegeben sind. Eine informelle Systemfunktionalitätsbeschreibung verschafft einen ersten Eindruck über die erwartete Funktionalität des zu erstellenden Produktes. Der Überblick ist selbsterklärend. Ein Aktivitäten-Protokoll bietet einen Einblick über die Entwicklung der Problembeschreibung. Die Beschreibung der Validierung und Verifizierung wird in zwei Teilprodukten abgelegt.

Wie beim Typ Anforderungsdokument werden die Attribute Validiert und Verifiziert angeboten. Insgesamt ergibt sich somit für den Produkttyp Problembeschreibung folgender Aufbau:

Komplexer Produkttyp Problembeschreibung

Slots:

- Name:#Problem-Repräsentation, Typ:String, Optionalität: False
- Name:#Umgebungscharakteristika, Typ:String, Optionalität:False
- Name:#Informelle Systemfunktionalität, Typ:String, Optionalität:False
- Name:#Übersicht Typ:String, Optionalität:False
- Name:#Validierung, Typ:String, Optionalität:False
- Name:#Verifizierung, Typ:String, Optionalität:False
- Name:#Aktivitätenprotokoll, Typ:String, Optionalität:False

Attribute:

- Name:#Validiert, Typ:Boolean, Optionalität:False
- Name:#Verifiziert, Typ:Boolean, Optionalität:False

Regeln: Keine

Supertyp: Nil

4.1.5 System-Anforderungen

System-Anforderungen bestehen konzeptionell aus einem Überblick, einer Begriffsdefinition und StP-OMT Grafiken.

Der Überblick liegt in textueller Form vor und darf nicht fehlen. Die Begriffsdefinitionen könnten beispielsweise in einer relationalen Datenbank stehen. Daher wird für sie ein eigener Tabellentyp definiert. Über die StP-OMT-Grafiken werden die Systemanforderungen beschrieben. Die Anzahl der Grafiken ist variabel, daher wird für sie ein Listentyp definiert. Somit ergibt sich für System-Anforderungen folgende Struktur:

Komplexer Produkttyp System-Anforderungen

Slots:

- Name:#Überblick, Typ:String, Optionalität:False
- Name:#Begriffsdefinition, Typ:Dictionary-Tabelle, Optionalität:False
- Name:#OMT-Abbildungen Typ:OMT-Liste, Optionalität:False

Attribute: Keine

Regeln: Keine

Supertyp: Nil

Tabellen-Produkttyp Dictionary-Tabelle

Verweis auf: Tupel-Produkttyp Dictionary-Tupel

Supertyp: Nil

Tupel-Produkttyp Dictionary-Tupel

Tupelelemente:

- Name:#Begriff, Typ:String, Optionalität:False
- Name:#Erklärung, Typ:String, Optionalität:False

Supertyp: Nil

Listen-Produkttyp OMT-Liste

Kardinalität:

- Minimum: 0
- Maximum: unbegrenzt

Typ: OMT-Grafiken

Supertyp: Nil

Grafik-Produkttyp OMT-Grafiken

Editor:StP-OMT Grafikeditor
Reader:StP-OMT Grafikeditor
Referenz: - (wird erst von der Ausführungsmaschine gefüllt)
Grafiktyp:OMT-Diagramm

Supertyp: Nil

4.1.6 Anforderungstabellen

In Anforderungstabellen werden Anforderungen gespeichert. Diese Anforderungen fließen in verschiedene Produkte ein. Elemente der Tabelle bestehen aus einem Designator und einem Anforderungstext. Über den Designator kann man ein Element der Tabelle eindeutig bestimmen. Ein Beispiel für einen Designator ist Kunden-Anforderung 1.

Designator und Text liegen in Form eines Strings vor und sind nicht optional, was folgende Beschreibung widerspiegelt:

Tabellen-Produkttyp Anforderungstabelle

Verweis auf: Tupel-Produkttyp Anforderungs-Tupel

Supertyp: Nil

Tupel-Produkttyp Anforderungs-Tupel

Tupelelemente:

- Name:#Designator, Typ:String, Optionalität:False
- Name:#Anforderungstext, Typ:String, Optionalität:False

Supertyp: Nil

4.1.7 Testfall-Liste

Testfälle bestehen aus einer Testnummer, die den Testfall eindeutig identifiziert Außerdem müssen die Anforderungen, die getestet werden, angegeben werden. Eine Test-Strategie und die Startumgebung des Tests wird festgelegt. Das festgestellte Verhalten wird dokumentiert und die benutzte Testsequenz wird angegeben. Eine Testplanliste besteht aus einer Menge von Testfällen. Ein Testfall wird als komplexer Produkttyp definiert. Zusätzlich wird noch ein Listentyp für Strings definiert.

Listen-Produkttyp Testfall-Liste

Kardinalität:

- Minimum: 0
- Maximum: unbegrenzt

Typ: Testfall

Supertyp: Nil

Komplexer Produkttyp Testfall

Slots:

- Name:#Testnummer, Typ:Integer
- Name:#Anforderung, Typ:String
- Name:#Strategie, Typ:Textliste
- Name:#Startumgebung, Typ:Textliste
- Name:#Prüfungsverhalten, Typ:Textliste
- Name:#Testsequenz, Typ:Textliste

Attribute: Keine

Regeln: Keine

Supertyp: Nil

Listen-Produkttyp Textliste

Kardinalität:

- Minimum: 0
- Maximum: 20

Typ: String

Supertyp: Nil

4.2 Modellierung der Produkttypen in CoMo-Kit

Die Modellierung der Produkttypen in CoMo-Kit kann auf zwei Arten geschehen. Zum einen über Methodenaufrufe gemäß der Spezifikation und zum anderen über eine grafische Benutzeroberfläche, was einen wesentlich komfortableren Weg darstellt. Die erste Vorgehensweise ist im Anhang aufgeführt.

KAPITEL 5

Die GemStone-Anbindung

Dieses Kapitel begründet die Benutzung der objektorientierten Datenbank Gemstone und zeigt, wie man mit der Datenbank arbeiten kann.

5.1 Warum die OODB GemStone?

Software-Entwicklung ist oft ein lange andauernder Prozeß. Hierbei arbeiten viele Mitarbeiter zeitlich und räumlich verteilt an verschieden Komponenten des Produkts. Es ist daher zum einen erwünscht, die Daten dauerhaft zu speichern und zum anderen, den Zugriff auf die Daten verteilt mehreren Benutzern zur gleichen Zeit zu gestatten. Datenbanksysteme erfüllen diese Bedingungen. Sie stellen eine Möglichkeit zur persistenten Speicherung der Daten sowie ein Transaktionskonzept für schreibenden Datenzugriff zur Verfügung.

Die Verwendung einer objekt-orientierten Datenbank basiert auf der erwünschten natürlichen Darstellung der Objekte, die in einem Software-Entwicklungsprozeß verwendet werden. Die Verwendung einer relationalen Datenbank wäre prinzipiell auch möglich, verlangt aber einen wesentlich höheren Aufwand für die Umsetzung der Objektstrukturen in relationale Strukturen.

Für die im Rahmen des Sonderforschungsbereiches erzeugten Strukturen wird die objektorientierte Datenbank GemStone verwendet. Eine gute und knappe Einführung findet man in [Wol96], für genauere Information sei auf die begleitende Literatur (Handbücher) verwiesen. Die Konzepte von Datenbanksystemen (DBS), Datenbankmanagementsystemen (DBMS) und objekt-orientierten Datenbanksystemen (OODBS) werden als bekannt vorausgesetzt.

Ich möchte hier nur noch einmal auf die Konzepte eingehen, die direkt für diese Diplomarbeit von Bedeutung sind.

Zwischen der Datenbank und Visualworks-Smalltalk gibt es ein GemStone-Smalltalk-Interface (GSI). Dieses hat einen eigenen Datenraum, auf dem es arbeitet, eine lokale Sicht auf die Daten. Diese wird bei der Beendigung einer Transaktion erneuert. Bei einem Commit werden die Daten im Repository modifiziert, bei einem Abort werden die Daten im GSI durch die Daten aus dem Repository ersetzt.

Ein Connector verbindet Objekte aus dem Visualworks-Smalltalk mit Objekten aus der Datenbank beziehungsweise aus dem GSI. Wenn man einen Connector aktiviert, kann man über eine sogenannte Post Connect Action angeben, welche Seite nach dem Zustandekommen der Verbindung aktiviert wird, also Smalltalk oder GemStone. Er kann aber auch angeben, daß keine Aktion durchgeführt wird. Als letztes bleibt die Möglichkeit, statt einem Replikat einen Forwarder zu benutzen. Dann entfällt die Update-Frage.

Jeder Benutzer hat eine eigenes Benutzerprofil. Dazu gehört auch seine *SymbolList*. In dieser Liste sind die ihm bekannten Dictionaries enthalten.

Der GemStone-Smalltalk-Dialekt unterscheidet sich nur geringfügig von Visualworks-Smalltalk. Man muß jedoch immer daran denken, daß Methoden von GemStone-Smalltalk auch in GemStone ausgeführt werden, nicht in Smalltalk.

Für die objektorientierte Datenbank GemStone sprechen mehrere Gründe:

1. **Datendefinition und -manipulation**

   GemStone-Smalltalk-DB heißt die Sprache zur Definition und Manipulation der Daten. Sie ist dem zur Implementierung der Basisstrukturen verwendeten Visualworks-Smalltalk sehr ähnlich. Der Unterschied zu Datenbankmanipulationssprachen wie SQL besteht darin, daß GemStone-Smalltalk-DB nicht nur für administrative Zwecke wie Datenbankanfragen, sondern auch zur Durchführung komplexer Programme benutzt werden kann.

2. **Visualworks-Interface**

   Es existiert ein Interface zwischen der Datenbank Gemstone und Visualworks-Smalltalk. Die Übertragung von Strukturen, die man in Visualworks-Smalltalk erzeugt hat, in die Datenbank ist unkompliziert.

3. **Einarbeitungsdauer**

   Auch wenn man keine Erfahrung mit Datenbanken hat ist die Einarbeitungszeit verhältnismäßig kurz.

4. **C++-Interface**

   Es existiert eine Möglichkeit mit C++-Programmen auf die Datenbank zuzugreifen. Hiermit ergibt sich auch für Personen, die nicht Smalltalk benutzen die Möglichkeit, die in der Datenbank gespeicherten Modelle zu benutzen. Die C++-Schnittstelle stellt nicht den Komfort zur Verfügung wie die Visualworks-Smalltalk Schnittstelle. Zur Erstellung eigener Klassen in der Datenbank wird ein Registrar benötigt. Dieser arbeitet als Präprozessor auf den selbstdefinierten Klassen und speichert sie in der Datenbank. Er stellt aber noch zusätzliche Funktionalitäten zur Verfügung, wie etwa die Optimierung von Anfragenkode.

5. **Positive Erfahrungen**

   GemStone wird in der Arbeitsgruppe schon etwas länger benutzt. Vor kurzem wurde die Diplomarbeit [Wol96] beendet, die sich mit Workflow-Management in einer objektorientierten Datenbank befaßt. Die gemachten Erfahrungen mit GemStone waren sehr positiv, ebenso wie in [Hit92].

   Ein Teil der Diplomarbeit bestand in der Übertragung der erzeugten Strukturen in die Datenbank. Die Arbeit von [Wol96] befaßte sich unter anderem mit der Übertragung von CoMo-Kit-Netzwerken in die Datenbank. Da die von im Rahmen meiner Diplomarbeit erzeugten Strukturen ebenfalls als Elemente eines (SFB-)Netzwerks vorliegen, beschränkt sich die Aufgabe auf die Anpassung der Methoden zur Übertragung der Daten und die Übertragung der Klassenstrukturen.

5.2 Übertragen der Basisstrukturen nach GemStone

Bei der Diplomarbeit kann man die Arbeit mit der Datenbank in zwei Teile aufspalten. Der erste Teil behandelt die ersten Schritte zur persistenten Speicherung der Daten in der Datenbank und der zweite Teil beschreibt die Aktionen, die durchgeführt werden, wenn die Klassenstruktur bereits in der Datenbank besteht.

5.2.1 Initiale Schritte mit GemStone

- Einloggen in die Datenbank
- Erzeugen eines eigenen Dictionaries für Klassen des Sonderforschungsbereiches
- Replikation der Klassenstrukturen des Sonderforschungsbereiches in GemStone
- Bewegen der Klassen in das Dictionary des Sonderforschungsbereiches
- Erzeugen der notwendigen Connectors für oben genannte Klassen mit den zugehörigen Post Connect Actions

- Verbinden der Klassenstrukturen
- Transaktion mit Commit beenden
- Testen der obigen Schritte anhand einer Beispielstruktur
- Transaktion mit Abort beenden

Befindet man sich in der VisualWorks-Smalltalk-Umgebung, so hat man sehr bequem über den Session Browser die Möglichkeit, sich in die Datenbank einzuloggen. Man kann sich auch über einen Methodenaufruf eines GemStone-Workspaces einloggen. Der Aufruf lautet

```
GSI login.
```

Daraufhin wird man nach den Aufrufparametern gefragt, also nach einem Usernamen und gegebenenfalls einem Passwort. Beim Vorgang des Einloggens versucht das GemStone-Smalltalk-Interface die bei der letzten Session vorhandenen Verbindungen wiederaufzubauen. Falls das aus irgendwelchen Gründen nicht möglich ist, so zum Beispiel, wenn auf einer der beiden Seiten die Klassenstruktur verändert wurde, wird man darüber verständigt.

Zu Beginn der ersten Session befanden sich die Strukturen lediglich auf der Visualworks-Smalltalk Seite. Ziel war es nun, sie in die lokale Sicht des Gem-Prozesses zu bringen und anschließend mit einem Commit die Daten in das Repository zu übertragen.

Die Klasse `SFB501ProjectModel` enthält zwei Klassenvariablen, eine instanceList für lokale Netze, die nicht in die Datenbank übertragen werden sollen und eine gemstoneInstanceList für globale Netze, die auch in der Datenbank stehen beziehungsweise aus der Datenbank geladen werden. Diese beiden Variablen können initialisiert werden durch die Methodenaufrufe

```
SFB501ProjectModel initializeInstanceList.
SFB501ProjectModel initializeGemstoneInstance-
List.
```

Nun sind die Instanzenlisten leer und initialisiert. Mit den Aufrufen

```
SFB501ProjectModel browseInstances.
SFB501ProjectModel browseGemstoneInstances.
```

öffnet man die Browser über den Instanzenlisten, welche die aktuell existierenden Netzwerke enthalten und über graphische Oberflächen Manipulationen an den Netzen ermöglichen. Die Öffnung des Browsers für die Gemstone-Instanzen verläuft nur erfolgreich, wenn man in die Datenbank eingeloggt ist.

Jeder Benutzer hat eine eigene *SymbolList*, in der die Dictionaries eingetragen werden, auf die er zugreifen darf. Nun wird ein Dictionary für die Klassen erzeugt, die repliziert werden sollen. Dieses hat den Namen SFB501Dictionary. Wenn Klassen durch das GSI erzeugt werden, so findet man sie zunächst in dem Dictionary GSIGenerated-Classes.

Man kann ein Dictionary am bequemsten über den Klassenbrowser von Gemstone erstellen. Bei einem Aufruf über Methoden hat man das Problem, daß man eine Gemstone-Instanz braucht, da ja ein Dictionary in Gemstone angelegt wird. Die Aufrufabfolge sieht dann Beispielsweise so aus:

```
GSI currentSession execute: ,| newDictionary |
newDictionary := SymbolListDictionary new.
newDictionary at: #SFB501Dictionary put: new-
Dictionary.
System myUserProfile insertDictionary: newDic-
tionary at: 1.‘
```

„GSI currentSession“ liefert eine Instanz der Klasse GSSession. Diese Instanz versteht die Methode „execute“, die einen (Smalltalk-DB) String in Gemstone ausführt. In diesem String wird ein Dictionary erzeugt, daß zu Beginn als einziges Element sich selbst enthält.

Die Klasse `SFB501ProjectModel` kann nun in dieses Dictionary übertragen werden. Das kann auf zwei Arten geschehen.

1. **Menügesteuert**

   Über den Visualworks-FullClassBrowser wählt man die Klasse `SFB501ProjectModel` und dem Menüpunkt CreateInGS. Dann öffnet oder aktualisiert man einen Gemstone-Klassenbrowser und verschiebt die Klasse mit dem Menüpunkt *move* aus dem Dictionary GSIGeneratedClasses ins Dictionary SFB501Dictionary.

2. **Methodengesteuert**

   ```
   SFB501ProjectModel createGemStoneClass.
   ```

Diese Methode, führt die oben genannten Schritte aus, also auch das Übertragen in das Dictionary SFB501Dictionary. Von großem Vorteil ist dabei, daß das Dictionary erzeugt wird, falls es bis dahin noch nicht existiert.

Damit ist die Klasse repliziert und steht im vorgesehenen Dictionary. Das Dictionary muß allerdings noch für alle sichtbar gemacht werden. Möchte beispielweise ein anderer User mit denselben oder mehr Rechten diese Klassen lesen können, so muß er sie erst über einen SymbolList Browser in seine Symbolliste aufnehmen.

Die beiden Klassen mit dem Namen `SFB501ProjectModel` müssen nun verbunden werden. Das wird über einen *Connector* realisiert. Auch hierfür gibt es wieder die beiden Möglichkeiten, es über Browser oder über Methodenaufrufe zu realisieren. Es gibt einen Connector Browser, der alle Connectors anzeigt und es auch erlaubt, einen Connector hinzuzufügen. Weiter hat man die Möglichkeit, die oben erwähnte Post Connect Action anzugeben, die festlegt, ob die Visualworks-Seite oder die GemStone-Seite aktualisiert wird.

In der Klasse `SFB501ProjectModel` existiert eine Methode, die einen Connector erzeugt und die Voreinstellung der Post Connect Action übernimmt. Es soll immer die Smalltalk-Seite aktualisiert werden, da eine anderer Modellierer die Daten seit dem letzten Login in die Datenbank geändert haben könnte. Der Methodenaufruf lautet

```
(SFB501ProjectModel addConnector_InstanceList)
updateGSOnConnect.
```

Es wird hierbei ein Connector zwischen den beiden Klassenvariablen gemstoneInstanceList eingerichtet, so daß er zu Beginn der Verbindung die Datensicht der GemStone-Seite auffrischt. Weiter oben haben wir die Variable gemstoneInstanceList auf Visualworks-Smalltalk-Seite initialisiert, daher ist sie nach der Verbindung ebenfalls initialisiert. Hierzu ist noch anzumerken, daß bei späteren Verbindungen immer die Smalltalk-Seite aktualisiert werden muß, da die Instanzenlisten aus der Datenbank übernommen werden sollen. Noch sind die Variablen jedoch nicht verbunden.

Die Verbindung kann man wiederum sehr komfortabel über den ConnectorBrowser durchführen, aber auch über den Methodenaufruf

```
SFB501ProjectModel
connect_GemstoneInstance_Var_First_Time.
```

Nun werden die im letzten Kapitel beschriebenen Produkttypen in eine Instanz der Klasse `SFB501ProjectModel` geschrieben. Dieses geht zum Beispiel mit

```
| example |
example := SFB501ProjectModel new: #Productty-
pes.
example sfbDemo.
```

Das erzeugte Beispiel befindet sich noch in der Klassenvariable Instance List und wird nun auf die Klassenvariable gemstoneInstanceList übertragen. Dieses geschieht über die Klassenmethode

```
(SFB501ProjectModel instanceList networkNamed:
#Producttypes) beGemstoneNetwork.
```

Alle Klassen, für die in diesem Beispiel Instanzen erzeugt werden, findet man anschließend auch in Gemstone. Es ist allerdings sauberer, die Aufsetzpunkte für die neuen Klassen nach Gemstone zu übertragen, denn falls jemand einen Produkttyp ändern möchte, dann sollte er auch alle dafür benötigten Strukturen in der Datenbank vorfinden. Daher muß man die noch nicht vorhandenen Klassen noch in die Datenbank portieren. Da die Klassen zunächst in dem Dictionary GSIGeneratedClasses stehen, müssen sie noch in das SFB501Dictionary übertragen werden. Da es sich um Gemstone-Methoden und Gemstone-Objekte handelt, müssen die Methoden in einem GemStone-Workspace ausgeführt werden. Die Methodenaufrufe haben die Form

```
|dict|
dict := (System myUserProfile resolveSymbol:
#GSIGeneratedClasses) value.
dict  do: [ :a |(a value isKindOf: Class) ifT-
rue: [
SFB501Dictionary add: a.
dict removeKey: a key]
].
```

Die Symbole der generierten Klassen werden in der Variable Dictionary gesammelt und anschließend in das SFB501Dictionary übertragen.

5.2.2 Schritte bei der Arbeit mit GemStone

Im Gegensatz zur üblichen lokalen Modellierung kommen nun noch ein paar wenig aufwendige Punkte hinzu, die aber für den dadurch entstehenden Nutzen akzeptiert werden:

- Einloggen in die Datenbank
- Verbinden der Klassenstrukturen
- Modellieren von Strukturen des Software-Entwicklungsprozesse
- Transaktion mit Commit beenden

Möchte man die Modellierungsarbeit in GemStone speichern, so ist es notwendig, sich in die Datenbank einzuloggen.

Man muß den Connector, wenn er vorhanden ist, verbinden, ansonsten einen Neuen erzeugen und diesen verbinden. Im Gegensatz zum oben beschriebenen Modellierungsschritt muß man aber die Post-ConnectAction *updateST* wählen, sonst werden die Daten in der

Datenbank durch die wahrscheinlich veralteten Daten aus dem Visualworks-Smalltalk-Image überschrieben. Nun kann man mit der Modellierung beginnen.

Nach dem Editieren sollte man die Transaktion mit Commit beenden, damit die Änderungen aus dem GSI in das Repository des Datenbanksystems übernommen werden.

KAPITEL 6

Fazit und Ausblick

Hier wird die Arbeit bewertet sowie auf noch durchzuführende Arbeiten und zu klärende Fragen hingewiesen.

Bei der Festlegung der Sprachkonzepte blieben einige Punkte offen. Das lag daran, daß die Probleme zu komplex waren, um sie in dem gegebenen zeitlichen Rahmen zu lösen. Im folgenden werden einige der ungeklärten Fragen aufgeführt. Zunächst werden die Ergebnisse der Arbeit zusammengefaßt

6.1 Zusammenfassung

Die Ziele der Arbeit wurden erreicht. Ein wichtiges Ziel war es, die Entwicklung eines Prototyps in Smalltalk aufgrund einer *vorher* entwickelten Spezifikation durchzuführen. Aufgrund der sehr informellen Beschreibung der Daten war der Prozeß der Schnittstellendefinition recht langwierig. Es ist erstaunlich, wie viele Auslegungsmöglichkeiten eine Umgangssprache liefert. Wenn man den Vergleich zu einem Software-Entwicklungsprozeß zieht, so nehme ich die Rolle des Entwicklers ein, der in einem iterativen Verbesserungsprozeß die Benutzeranforderungen gegen die Systemanforderungen validiert. Die Änderungen an den Spezifikationen wurden noch bis vor wenigen Tagen in diese Diplomarbeit aufgenommen.

Nachdem die Schnittstellendefinition für die Produkte relativ stabil erschien, war es komfortabel, die Klassen und Methoden gemäß der Spezifikation zu implementieren. Da die Schnittstellendefinition vorlag, konnte man anhand dieser effizient implementieren. Leider war die Spezifikation nicht so stabil wie zu diesem Zeitpunkt angenom-

men. Änderungen in der Spezifikation haben auch Änderungen in der Implementierung zur Folge. Als Beispiel hierfür sei die Regelkomponente erwähnt, die sich im Verlauf der Diskussionen oft verändert hat. Die Nutzung der vorhandenen Strukturen fiel nicht schwer, nachdem man sich einen Überblick über den strukturellen Aufbau der für die Arbeit wichtigen CoMo-Kit-Klassen gemacht hatte. Mit Visualworks-Smalltalk steht in der Arbeitsgruppe eine sehr komfortable Entwicklungsumgebung zur Verfügung.

Das Tool StP-OMT war nicht so effizient zu gebrauchen wie erwartet und verhielt sich war zum Teil sehr Fehlerbehaftet. Die zeichentechnische Handhabung jedoch ist sehr angenehm. Leider entspricht eine Datei im Postscript-Format nicht immer der auf dem Bildschirm erstellten Grafik, was den „Feinschliff“ erheblich erschwerte. Wie brauchbar der vom Tool generierte C++-Code ist, konnte nicht getestet werden.

Ich denke, daß wir mit den erstellten Konzepten einen guten Ansatz für ein ausdrucksmächtiges und zugleich komfortabel zu bedienendes Werkzeug bereitgestellt haben, daß mit einem vertretbaren Ressourcenaufwand unter Einhaltung der Zeit- und Kostenrestriktionen erstellt werden kann.

6.2 Offene Punkte bei der Beschreibung der Sprachkonzepte

Zu Produkten blieb der Punkt offen, wie man Konfigurationsmanagement handhabt und wie man die Versionierung einbinden kann. Bei dem modellierten Produkttyp in Kapitel 4 wird die Versionsnummer als Teilprodukt in den Produkttyp mitaufgenommen.

Ein weiterer Punkt betrifft die Kompatibilität der Produkte mit Entwicklungstools. Wie kann man diese gewährleisten?

Schwieriger wird es bei den Prozessen. Hier ist es noch offen, was für Vererbungsregeln es gibt. Heuristisches Planungswissen wird bisher noch nicht berücksichtigt. Ein Beispiel hierfür ist, daß einer objektorientierten Analyse auch ein objektorientierter Entwurf folgen sollte. Hierbei kann man auch noch zwischen einer Vorschrift und einem Vorschlag unterscheiden, so daß man auf „weiche“ Bedingungen kommt.

Auch die Berücksichtigung von Kostenaspekten wie beispielsweise vorhandenes Budget und Personal bei den Prozeßtypen wurde nicht geklärt.

Wie kann man Attribute manipulieren? Durch Regeln, durch Benutzer oder durch Regeln *und* Benutzer? Dieser Punkt bedarf ebenfalls noch der Klärung.

Bei Prozeßdefinitionen handelt es sich um Referenzen auf einen Prozeßtyp, und einer Menge von Änderungen an diesem. Es wurde noch nicht festgelegt, welche Änderungen erlaubt sind und welche nicht.

Bei Methoden ist noch offen, ob die Datenabbildung generisch oder konkret festgelegt wird. Die generische Festlegung ist beispielsweise für den Fall von „beliebig vielen Teilprozessen“ interessant.

Bei den Ressourcen bleibt die Frage offen, ob die Aufbauorganisation explizit repräsentiert werden soll. Der von mir geschilderte Ansatz sieht noch kein explizite Repräsentation der Aufbauorganisation vor. Lediglich über die Properties wird implizit das Wissen über diese gespeichert. Im Laufe weiterer Diskussionen werden diese Fragen sicherlich beantwortet und wahrscheinlich werden bis zum Abschluß der Konzeptionsphase noch einige Fragen auftreten.

6.3 Probleme mit GemStone

Die Einarbeitungszeit in das Datenbanksystem ist relativ gering. Es gibt allerdings manchmal kleinere Probleme, bei denen die Anwesenheit eines erfahreneren GemStone-Users von großem Wert ist.

Die C++-Schnittstelle ist auf einen festen Compiler zugeschnitten. Hat man diesen Compiler nicht, so kann man die bereitgestellten Programme nicht nutzen. Zum jetzigen Zeitpunkt ist dieser Compiler nicht vorhanden. Programme für einen Compiler, der in der AG Rombach installiert ist, sind etwa ab Mitte 1996 lieferbar.

6.4 Weiterführende Arbeiten

Neben der iterativen Verbesserung der Schnittstellendefinition müssen die noch fehlenden Basisstrukturen implementiert werden. Dabei können das Prozeß- und das Methodenmodell wiederum auf bereits vorhandene CoMo-Kit-Strukturen zurückgreifen. Das Ressourcenmodell wird derzeit bereits in einer Projektarbeit nach der vorhandenen Schnittstellenspezifikation implementiert. Das Übertragen in die Datenbank dürfte keine Schwierigkeiten bereiten.

Man muß sehen, wie weit das Ausführungstool auf Strukturen des Schedulers zurückgreifen kann. Eine Wiederverwendung von bereits implementierten Schedulerklassen und -methoden würde auch hier viel Implementierungsaufwand einsparen.

KAPITEL 7

Literaturverzeichnis

[Boe81]
Boehm, B.W.: *Software Engineering Economics*, Prentice-Hall,1981

[Brö95]
Bröckers A., Lott C. M., Rombach H. D., Verlage M.: *MVP-L Language Report Version 2*, Interner Bericht, FB Informatik, Universität Kaiserslautern 265/95, Februar 1995.

[Büc93]
Bücker, M. C.: *Objectworks/Smalltalk für Anfänger*, Springer-Verlag, 1993

[Del95]
Dellen, B.: *Verwaltung von Abhängigkeiten bei der Operationalisierung konzeptueller Modelle*. Diplomarbeit, Universität Kaiserslautern, 1995.

[DeMaPa95]
Dellen B., Maurer F., Paulokat J.: *Verwaltung von Abhängigkeiten in kooperativen wissensbasierten Arbeitsabläufen*, In M.M. Richter und F. Maurer (Hrsg.), Proceedings in Artificial Intelligence 2, Expertensystem 95, 1995.

[Gem95]
GemStone-Manuals, Version 4.1, Juli 1995

[Ghe91]
Ghezzi, C. et al: *Fundamentals of Software Engineering*, Prentice-Hall, 1991

[Gol83]
Goldberg, A.: *SMALLTALK 80 - The Language and its implementation*. Addison Wesley, 1983.

[Gol95]
Goldmann, S.: *CoMo-Kit und MVP-L: Ein Vergleich zwischen zwei Werkzeugen zur Prozeßmodellierung und Ausführung*, Projektarbeit, Universität Kaiserslautern, 1995.

[Hit92]
Hitzges, A.: *Mehrbenutzerverwaltung für das Hypermedia System HyperCAKE*, Diplomarbeit, Universität Kaiserslautern, 1992

[Jal91]
Jalote, P.: *An Integrated Approach to Software Engineering*, Springer-Verlag, 1991

[Koh95]
Kohler, K.: *Design Rationale bei der Modellierung und Abwicklung von Entwurfsprozessen*. Diplomarbeit, Universität Kaiserslautern, 1995.

[Mau93]
Maurer, F.: *Hypermediabasiertes Knowledge-Engineering für verteilte wissensbasierte Systeme*. Dissertation, Universität Kaiserslautern, 1993.

[Mün95]
Münch J., Maurer F., Verlage M., Dellen B.: *Konzepte einer Prozeßmodellierungssprache, Internes Konzeptpapier, Projekt SFB501, 1996*

[Rom94]
Rombach, H.D.: *Software Engineering II*, Skript zur Vorlesung SS 94, Universität Kaiserslautern

[Rom95]
Rombach, H.D.: *Software Engineering I*, Skript zur Vorlesung WS94/95, Universität Kaiserslautern

[Rum91]
Rumbaugh, J.: *Object Oriented Modeling and Design*, Prentice Hall, New York, 1991.

[Sch94]
Schmitz, W.G.: *Multiple Aufgabenzerlegung von konzeptuellen Modellen*, Diplomarbeit, Universität Kaiserslautern, 1994.

[Som92]
Sommerville, I.: *Software Engineering*, Addison Wesley, 1992

[Tol93]
Tolzmann, E.: *Entwicklung einer Ausführungsmaschine für die Prozeßmodellierungssprache MVP-L*, Diplomarbeit, Universität Kaiserslautern, 1993

[Wol96]
Wolf, M.: *Workflow-Management mit einer Objekt-orientierten Datenbank*, Diplomarbeit, Universität Kaiserslautern, 1996

ANHANG 1

Modellierung eines Beispielprodukts mit den Methoden der Schnittstellendefinition

Die nun folgende Methode existiert in der Klasse SFB501ProjektModel. Sie kann beispielsweise durch die Sequenz

```
(SFB501ProjektModel new: #Product_Example) sfbDemo.
```

aufgerufen werden.

sfbDemo

„Beispiel für die Erstellung eines komplexen Produkttyps.“

„Letzte Änderung: 01.03.1996 B.K.“

„Definition der temporären Variablen“
| slt cp at tu ta gr li rule |

„Generelle Vorgehensweise zur Erzeugung von Objekten der Prozeßmodellierungssprache:
a) Erzeuge neues Objekt mit ‚new‘
b) Trage Objekt in Netzwerk ein
c) Führe abhängig vom Objekttyp verschiedene Aktionen aus „

„----------“

„I.) Komplexer Produkttyp Dokument“

cp := SFBComplexProduct new: #Document.
self addObjectEntry: cp.
(self getObjectByName: #Document) setSupertype: (self getObjectByName: #TYPE).

„Erzeugen der Slots für den komplexen Produkttyp Dokument“

„1. Versionsnummer des Dokuments“
slt := SFBSlot new: #Version.
self addObjectEntry: slt.
(self getObjectByName: #Version) setType: (self getObjectByName: #String).
(self getObjectByName: #Version) setOptionalityFlag: false.

```
„2. Erstellungsdatum, optional“
slt := SFBSlot new: #Date.
self addObjectEntry: slt.
(self getObjectByName: #Date ) setType: (self getObjectByName: #String).
(self getObjectByName: #Date ) setOptionalityFlag: true.

„3.Dokumenttitel“
slt := SFBSlot new: #Title.
self addObjectEntry: slt.
(self getObjectByName: #Title ) setType: (self getObjectByName: #String).
(self getObjectByName: #Title ) setOptionalityFlag: false.

„Einfügen der Slots in komplexen Produkttyp Dokument“
(self getObjectByName: #Document ) addSlot: (self getObjectByName: #Ver-
sion).
(self getObjectByName: #Document ) addSlot: (self getObjectByName: #Date).
(self getObjectByName: #Document ) addSlot: (self getObjectByName: #Title).

„----------“

„II.) Komplexer Produkttyp Anforderungsdokument“

cp := SFBComplexProduct new: #Requirements_Document.
self addObjectEntry: cp.
(self getObjectByName: #Requirements_Document) setSupertype: (self get-
ObjectByName: #Document).



„Erzeugen von Attributen und Slots fuer den komplexen Produkttyp Anforde-
rungsdokument“

„1. Name(n) der ausführenden Person(en)“
slt := SFBSlot new: #Employee_Name.
self addObjectEntry: slt.
(self getObjectByName: #Employee_Name ) setType: (self getObjectByName:
#String).
(self getObjectByName: #Employee_Name ) setOptionalityFlag: false.

„2. Gruppennummer, optional“
slt := SFBSlot new: #Group_Number.
self addObjectEntry: slt.
(self getObjectByName: #Group_Number ) setType: (self getObjectByName:
#Integer).
(self getObjectByName: #Group_Number ) setOptionalityFlag: true.

„3. Erzeugung des komplexen Produkttyps Inhaltsverzeichnis-Elemente. Das
ist der Elementtyp für den Listentyp Inhaltsverzeichnis“
cp := SFBComplexProduct new: #Table_Of_Contents_Elements.
self addObjectEntry: cp.
(self getObjectByName: #Table_Of_Contents_Elements) setSupertype: (self
getObjectByName: #TYPE).

„a) Kapitelnummer, nicht optional“
slt := SFBSlot new: #Chapter_Number.
self addObjectEntry: slt.
(self getObjectByName: #Chapter_Number ) setType: (self getObjectByName:
#String).
(self getObjectByName: #Chapter_Number ) setOptionalityFlag: false.

„b) Überschrift, nicht optional“
slt := SFBSlot new: #Headline.
```

```
self addObjectEntry: slt.
(self getObjectByName: #Headline ) setType: (self getObjectByName: #String).
(self getObjectByName: #Headline ) setOptionalityFlag: false.

„c) Seitennummer, nicht optional“
slt := SFBSlot new: #Page_Number.
self addObjectEntry: slt.
(self getObjectByName: #Page_Number ) setType: (self getObjectByName: #String).
(self getObjectByName: #Page_Number ) setOptionalityFlag: false.

„Einfügen der Slots in komplexen Produkttyp Inhaltsverzeichnis-Elemente“
(self getObjectByName: #Table_Of_Contents_Elements ) addSlot: (self getObjectByName: #Chapter_Number).
(self getObjectByName: #Table_Of_Contents_Elements ) addSlot: (self getObjectByName: #Headline).
(self getObjectByName: #Table_Of_Contents_Elements ) addSlot: (self getObjectByName: #Page_Number).

„4. Der Listen-Produkttyp Inhaltsverzeichnis. Maximum = 0 --> Anzahl der Werte ist unbegrenzt“
li := SFBList new: #Table_Of_Contents.
li setMin: 1.
li setMax: 0.
li setElementType: (self getObjectByName: #Table_Of_Contents_Elements).
self addObjectEntry: li.
(self getObjectByName: #Table_Of_Contents) setSupertype: (self getObjectByName: #TYPE).

„5. Erzeugen des Slots Anforderungs-Inhaltsverzeichnis, der später in das komplexe Produkt ‚Anforderungsdokument‘ eingebunden wird.“
slt := SFBSlot new: #Requirements_TOC.
self addObjectEntry: slt.
(self getObjectByName: #Requirements_TOC ) setType: (self getObjectByName: #Table_Of_Contents).
(self getObjectByName: #Requirements_TOC ) setOptionalityFlag: false.

„6. Der komplexe Produkttyp Problembeschreibung“
cp := SFBComplexProduct new: #Problem_Description.
self addObjectEntry: cp.
(self getObjectByName: #Problem_Description) setSupertype: (self getObjectByName: #TYPE).

„Erzeugen der Attribute und Slots fuer den komplexen Produkttyp Problembeschreibung“
„a) Problem-Repräsentation“
slt := SFBSlot new: #Problem_Representation.
self addObjectEntry: slt.
(self getObjectByName: #Problem_Representation ) setType: (self getObjectByName: #String).
(self getObjectByName: #Problem_Representation ) setOptionalityFlag: false.

„b) Umgebungscharakteristika“
slt := SFBSlot new: #Environment_Characteristics.
self addObjectEntry: slt.
(self getObjectByName: #Environment_Characteristics ) setType: (self getObjectByName: #String).
(self getObjectByName: #Environment_Characteristics ) setOptionalityFlag: false.
```

```
„c) Informelle Systemfunktionalität“
slt := SFBSlot new: #Informal_System_Funktionality.
self addObjectEntry: slt.
(self getObjectByName: #Informal_System_Funktionality ) setType: (self get-
ObjectByName: #String).
(self getObjectByName: #Informal_System_Funktionality ) setOptionalityFlag:
false.

„d) Überblick“
slt := SFBSlot new: #Synopsis.
self addObjectEntry: slt.
(self getObjectByName: #Synopsis ) setType: (self getObjectByName:
#String). (self getObjectByName: #Synopsis ) setOptionalityFlag: false.

„e) Validierung“
slt := SFBSlot new: #Validation.
self addObjectEntry: slt.
(self getObjectByName: #Validation ) setType: (self getObjectByName:
#String). (self getObjectByName: #Validation ) setOptionalityFlag: false.

„f) Verifizierung“
slt := SFBSlot new: #Verification.
self addObjectEntry: slt.
(self getObjectByName: #Verification ) setType: (self getObjectByName:
#String).
(self getObjectByName: #Verification ) setOptionalityFlag: false.

„g) Aktivitätenprotokoll“
slt := SFBSlot new: #Activity_Protocol.
self addObjectEntry: slt.
(self getObjectByName: #Activity_Protocol) setType: (self getObjectByName:
#String).
(self getObjectByName: #Activity_Protocol ) setOptionalityFlag: false.

„h) Das Attribut Validiert“
at := SFBAttribute new: #PD_Validated.
self addObjectEntry: at.
(self getObjectByName: #PD_Validated ) setType: (self getObjectByName:
#Boolean).

„i) Das Attribut Verifiziert“
at := SFBAttribute new: #PD_Verified.
self addObjectEntry: at.
(self getObjectByName: #PD_Verified ) setType: (self getObjectByName:
#Boolean).

„Einfügen der Slots und Attribute in den komplexen Produkttyp Problembe-
schreibung“
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Problem_Representation).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Environment_Characteristics).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Informal_System_Funktionality).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Synopsis).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Validation).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Verification).
(self getObjectByName: #Problem_Description) addSlot: (self getObjectBy-
Name: #Activity_Protocol).
```

(self getObjectByName: #Problem_Description) addAttribute: (self getObject-ByName: #PB_Validated).
(self getObjectByName: #Problem_Description) addAttribute: (self getObject-ByName: #PB_Verified).

„7. Erzeugen des Slots Anforderungsdokument-Problembeschreibung, der später in den komplexen Produkttyp Anforderungsdokument eingebunden wird."
slt := SFBSlot new: #Requirements_PD.
self addObjectEntry: slt.
(self getObjectByName: #Requirements_PD) setType: (self getObjectBy-Name: #Problem_Description).
(self getObjectByName: #Requirements_PD) setOptionalityFlag: false.

„8. Der komplexe Produkttyp Systemanforderungen"
cp := SFBComplexProduct new: #System_Requirements.
self addObjectEntry: cp.
(self getObjectByName: #System_Requirements) setSupertype: (self getOb-jectByName: #TYPE).

„Erzeugen der Attribute und Slots für den komplexen Produkttyp Systemanfor-derungen"
„a) Erzeugen des Slots Überblick"
slt := SFBSlot new: #Overview.
self addObjectEntry: slt.
(self getObjectByName: #Overview) setType: (self getObjectByName: #String).
(self getObjectByName: #Overview) setOptionalityFlag: false.

„b1) Erzeugen des Tupel-Produkttyps Dictionary_Tuple mit den Spalten Begriff und Erklärung"
tu := SFBTuple new: #Dictionary_Tuple.
self addObjectEntry: tu.
tu addColumn: #Notion withType: (self getObjectByName: #String).
tu addColumn: #Declaration withType: (self getObjectByName: #String).
(self getObjectByName: #Dictionary_Tuple) setSupertype: (self getObjectBy-Name: #TYPE).

„b2) Erzeugen des Tabellen-Produkttyps Dictionary_Table mit dem Verweis auf den Tupel-Produkttyp Dictionary-Tupel"
ta:= SFBTable new: #Dictionary_Table.
self addObjectEntry: ta.
ta setTupleType: (self getObjectByName: #Dictionary_Tuple).
(self getObjectByName: #Dictionary_Table) setSupertype: (self getObjectBy-Name: #TYPE).

„b) Erzeugen des Slots Begriffsdefinition für den komplexen Produkttyp Systemanforderungen"
slt := SFBSlot new: #Notion_Definition.
self addObjectEntry: slt.
(self getObjectByName: #Notion_Definition) setType: (self getObjectByName: #Dictionary_Table).

„c1) Setzen des Grafiktyps-OMT-Grafik"
gr:= SFBGraphic new: #OMT_Graphics.
self addObjectEntry: gr.
gr setGraphicType: #Diagram.
gr setEditor: #StP_OME.
gr setReader: #StP_OME. (self getObjectByName: #OMT_Graphics) setSu-pertype: (self getObjectByName: #TYPE).

„c2) Setzen des Listentyps OMT-Objektliste“
li:= SFBList new: #OMT_Object_List.
self addObjectEntry: li.
li setMin: 0.
li setMax: 0.
li setElementType: (self getObjectByName: #OMT_Graphics).
(self getObjectByName: #OMT_Object_List) setSupertype: (self getObjectBy-Name: #TYPE).

„c) Erzeugen des Slots OMT-Abbildungen für den komplexen Produkttyp Systemanforderungen“
slt := SFBSlot new: #OMT_Illustrations.
self addObjectEntry: slt.
(self getObjectByName: #OMT_Illustrations) setOptionalityFlag: true.
(self getObjectByName: #OMT_Illustrations) setType: (self getObjectByName: #OMT_Object_List.

„Einfügen der Slots in das komplexe Produkt Systembeschreibung“
(self getObjectByName: #System_Requirements) addSlot: (self getObjectBy-Name: #Overview).
(self getObjectByName: #System_Requirements) addSlot: (self getObjectBy-Name: #Notion_Definition).
(self getObjectByName: #System_Requirements) addSlot: (self getObjectBy-Name: #OMT_Illustrations).

„9. Erzeugen des Slots, der später in das komplexe Produkt ‚Anforderungsdo-kument‘ eingebunden wird.“
slt := SFBSlot new: #Requirements_SR.
self addObjectEntry: slt.
(self getObjectByName: #Requirements_SR) setType: (self getObjectByName: #System_Requirements).
(self getObjectByName: #Requirements_SR) setOptionalityFlag: false.

„10. Erzeugung einer Anforderungstabelle, dafür zunächst Erzeugung des Tupeltyps“
tu := SFBTuple new: #Requirements_Tuple.
self addObjectEntry: tu.
tu addColumn: #Designator withType: (self getObjectByName: #String).
tu addColumn: #Requirements_Text withType: (self getObjectByName: #String).
(self getObjectByName: #Requirements_Tuple) setSupertype: (self getObject-ByName: #TYPE).

ta:= SFBTable new: #Requirements_Table.
self addObjectEntry: ta.
ta setTupleType: (self getObjectByName: #Requirements_Tuple).
(self getObjectByName: #Requirements_Table) setSupertype: (self getObject-ByName: #TYPE).

„11. Erzeugung der Benutzeranforderungen für den komplexen Produkttyp Anforderungsdokument mittels der vorher erzeugten Anforderungstabelle“
slt := SFBSlot new: #Customer_Requirements.
self addObjectEntry: slt.
(self getObjectByName: #Customer_Requirements) setType: (self getObject-ByName: #Requirements_Table).
(self getObjectByName: #Customer_Requirements) setOptionalityFlag: false.

„12. Erzeugung der Entwickleranforderungen für den komplexen Produkttyp Anforderungsdokument mittels der vorher erzeugten Anforderungstabelle“
slt := SFBSlot new: #Developer_Requirements.

```
self addObjectEntry: slt.
(self getObjectByName: #Developer_Requirements ) setType: (self getObject-
ByName: #Requirements_Table).
(self getObjectByName: #Developer_Requirements ) setOptionalityFlag: false.
```

„13. Erzeugung der nichtfunktionalen Anforderungen für den komplexen Produkttyp Anforderungsdokument mittels der vorher erzeugten Anforderungstabelle“

```
slt := SFBSlot new: #Nonfunctional_Requirements.
self addObjectEntry: slt.
(self getObjectByName: #Nonfunctional_Requirements ) setType: (self getOb-
jectByName: #Requirements_Table).
(self getObjectByName: #Nonfunctional_Requirements ) setOptionalityFlag:
false.
```

„14. Erzeugung der inversen Anforderungen für den komplexen Produkttyp Anforderungsdokument mittels der vorher erzeugten Anforderungstabelle“

```
slt := SFBSlot new: #Inverse_Requirements.
self addObjectEntry: slt.
(self getObjectByName: #Inverse_Requirements ) setType: (self getObjectBy-
Name: #Requirements_Table).
(self getObjectByName: #Inverse_Requirements ) setOptionalityFlag: false.
```

„15. Erstellung der Hilfstypen für die Slots Akzeptanz- und Systemtestpläne des komplexen Produkttyps Anforderungsdokument.“

„a) Generierung einer Liste von Texten als Typ für Slots des komplexen Produkttyps Testfall“

```
li := SFBList new: #Text_List.
self addObjectEntry: li.
li setMin: 0.
li setMax: 20.
li setElementType: (self getObjectByName: #String).
(self getObjectByName: #Text_List) setSupertype: (self getObjectByName:
#TYPE).
```

„b) Erstellen von Slots fuer des komplexen Produkttyp Testfall“

„b1) Testnummer“

```
slt:= SFBSlot new: #Test_Nr.
self addObjectEntry: slt.
(self getObjectByName: #Test_Nr ) setType: (self getObjectByName: #Inte-
ger).
(self getObjectByName: #Test_Nr ) setOptionalityFlag: false.
```

„b2) Die Anforderungen, die durch den Test abgedeckt werden“

```
slt:= SFBSlot new: #For_Requirements.
self addObjectEntry: slt.
(self getObjectByName: #For_Requirements ) setType: (self getObjectBy-
Name: #String).
(self getObjectByName: #For_Requirements ) setOptionalityFlag: false.
```

„b3) Die Strategie, nach der vorgegangen wird“

```
slt:= SFBSlot new: #Strategy.
self addObjectEntry: slt.
self getObjectByName: #Strategy ) setType: (self getObjectByName:
#Text_List).
(self getObjectByName: #Strategy ) setOptionalityFlag: false.
```

„b4)Startumgebung“

```
slt:= SFBSlot new: #Starting_Environment.
self addObjectEntry: slt.
(self getObjectByName: #Starting_Environment ) setType: (self getObjectBy-
```

```
Name: #Text_List).
(self getObjectByName: #Starting_Environment ) setOptionalityFlag: false.

„b5) Das Prüfungsverhalten“
slt:= SFBSlot new: #Proved_Behaviour.
self addObjectEntry: slt.
(self getObjectByName: #Proved_Behaviour ) setType: (self getObjectBy-
Name: #Text_List).
(self getObjectByName: #Proved_Behaviour ) setOptionalityFlag: false.

„b6) Die Testsequenz“
slt:= SFBSlot new: #Test_Sequence.
self addObjectEntry: slt.
(self getObjectByName: #Test_Sequence ) setType: (self getObjectByName:
#Text_List).
(self getObjectByName: #Test_Sequence ) setOptionalityFlag: false.

„c) Erstellung des komplexen Produkttyps Testfall“
cp := SFBComplexProduct new: #Testcase.
self addObjectEntry: cp.
cp addSlot: (self getObjectByName: #Test_Nr).
cp addSlot: (self getObjectByName: #For_Requirements).
cp addSlot: (self getObjectByName: #Strategy).
cp addSlot: (self getObjectByName: #Starting_Environment).
cp addSlot: (self getObjectByName: #Proved_Behaviour).
cp addSlot: (self getObjectByName: #Test_Sequence).
cp setSupertype: (self getObjectByName: #TYPE).

„d) Modellierung der Tabelle für Testfälle, die nun als Elementtyp den oben
definierten Produkttyp Testfall verwendet“
li := SFBList new: #Testcase_List.
self addObjectEntry: li.
li setMin: 0.
li setMax: 0.
li setElementType: (self getObjectByName: #Testcase).
(self getObjectByName: #Testcase_List) setSupertype: (self getObjectBy-
Name: #TYPE).

„16. Erstellung des Slots für Akzeptanztestpläne des komplexen Produkttyps
Anforderungsdokument.“
slt:= SFBSlot new: #Acceptance_Testplans.
self addObjectEntry: slt.
(self getObjectByName: #Acceptance_Testplans ) setType: (self getObjectBy-
Name: #Testcase_List).
(self getObjectByName: #Acceptance_Testplans ) setOptionalityFlag: false.

„17. Erstellung des Slots für Systemtestpläne des komplexen Produkttyps
Anforderungsdokument.“
slt:= SFBSlot new: #System_Testplans.
self addObjectEntry: slt.
(self getObjectByName: #System_Testplans ) setType: (self getObjectBy-
Name: #Testcase_List).
(self getObjectByName: #System_Testplans ) setOptionalityFlag: false.

„18. Erstellung des Attributs Herstellungskosten für den komplexen Produkttyp
Anforderungsdokument.“
at := SFBAttribute new: #Cost_Of_Production.
self addObjectEntry: at.
self getObjectByName: #Cost_Of_Production ) setType: (self getObjectBy-
Name: #Integer).
```

"19. Das Attribut Validiert für den komplexen Produkttyp Anforderungsdokument."
at := SFBAttribute new: #RD_Validated.
self addObjectEntry: at.
(self getObjectByName: #RD_Validated) setType: (self getObjectByName: #Boolean).

"20. Das Attribut Verifiziert für den komplexen Produkttyp Anforderungsdoku ment"
at := SFBAttribute new: #RD_Verified.
self addObjectEntry: at.
(self getObjectByName: #RD_Verified) setType: (self getObjectByName: #Boolean).

„21. Erstellung der Regel Verifkationsregel für den komplexen Produkttyp Anforderungsdokument.“
rule := SFBRule new: #Verification_Rule.
self addObjectEntry: rule.

„a) Der boolsche Ausdruck, der zu prüfen ist.“
expr := SFBEqual new: #Verification_Expression.
self addObjectEntry: expr.
expr setLHS: (self getObjectByName: #PD_Verified).
expr set RHS: false.

„b) Aktionsliste (Dictionary) enthält genau ein Assignment.“
assg := SFBAssignment new: #VR_Assignment_1.
assg setVariable: (self getObjectByName: #RD_Verified).
assg setValue: false.
self adObjectEntry: assg.

„c) Setzten des Ausdrucks und Aufnahme der Zuweisung in die Aktionsliste“
rule setExpression: expr.
rule addActionToList: assg.

„----------“

„Einfügen der Slots in den komplexen Produkttyp Anforderungsdokument“
(self getObjectByName: #Requirements_Document) addSlot:(self getObject-ByName: #Employee_Name).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Group_Number).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Requirements_TOC).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Requirements_PD).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Requirements_SR).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Customer_Requirements).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Developer_Requirements).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Nonfunctional_Requirements).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Inverse_Requirements).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #Acceptance_Testplans).
(self getObjectByName: #Requirements_Document) addSlot: (self getObject-ByName: #System_Testplans).

„Einfügen der Attribute in den komplexen Produkttyp Anforderungsdokument“
(self getObjectByName: #Requirements_Document) addAttribute: (self getObjectByName: #Cost_Of_Production).
(self getObjectByName: #Requirements_Document) addAttribute: (self getObjectByName: #RD_Validated).
(self getObjectByName: #Requirements_Document) addAttribute: (self getObjectByName: #RD_Verified).

„Einfügen der Regel in den komplexen Produkttyp Anforderungsdokument“
(self getObjectByName: #Requirements_Document) addRule: (self getObjectByName: #Verification_Rule).

„----------“

ANHANG 2

StP-OMT Diagramme

Die Diagramme spiegeln den momentanen Stand der Schnittstellenspezifikation wider.

Die in Kapitel 2 der Arbeit verwendeten Diagramme stellen Teile der momentanen Schnittstellenspezifikation dar. Hier werden die vollständigen StP-OMT-Diagramme für die Oberste Klassenebene (Abbildung 19), Produkttypen (Abbildung 20), Regeln (Abbildung 21) und Prozesse (Abbildung 22)dargestellt. Die Abbildungen für Methoden und Ressourcen aus Kapitel 2 stellen bereits die vollständige Schnittstellenspezifikation dar und werden hier nicht noch einmal abgebildet.

ABBILDUNG 19 Oberste Klassenstruktur für Objekte der Prozeßmodellierungssprache

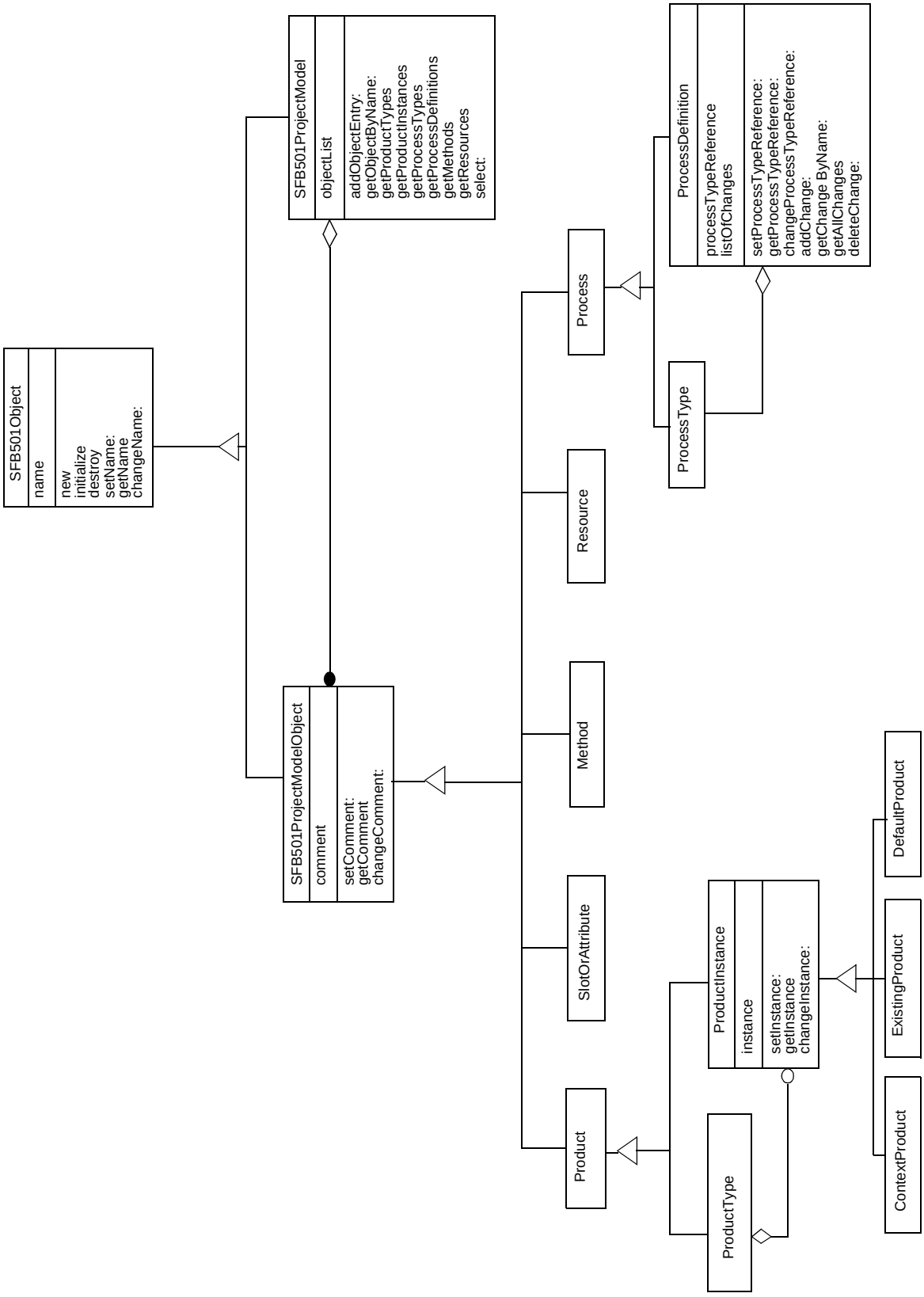


ABBILDUNG 20 Vollständige Klassenhierarchie für Produkttypen

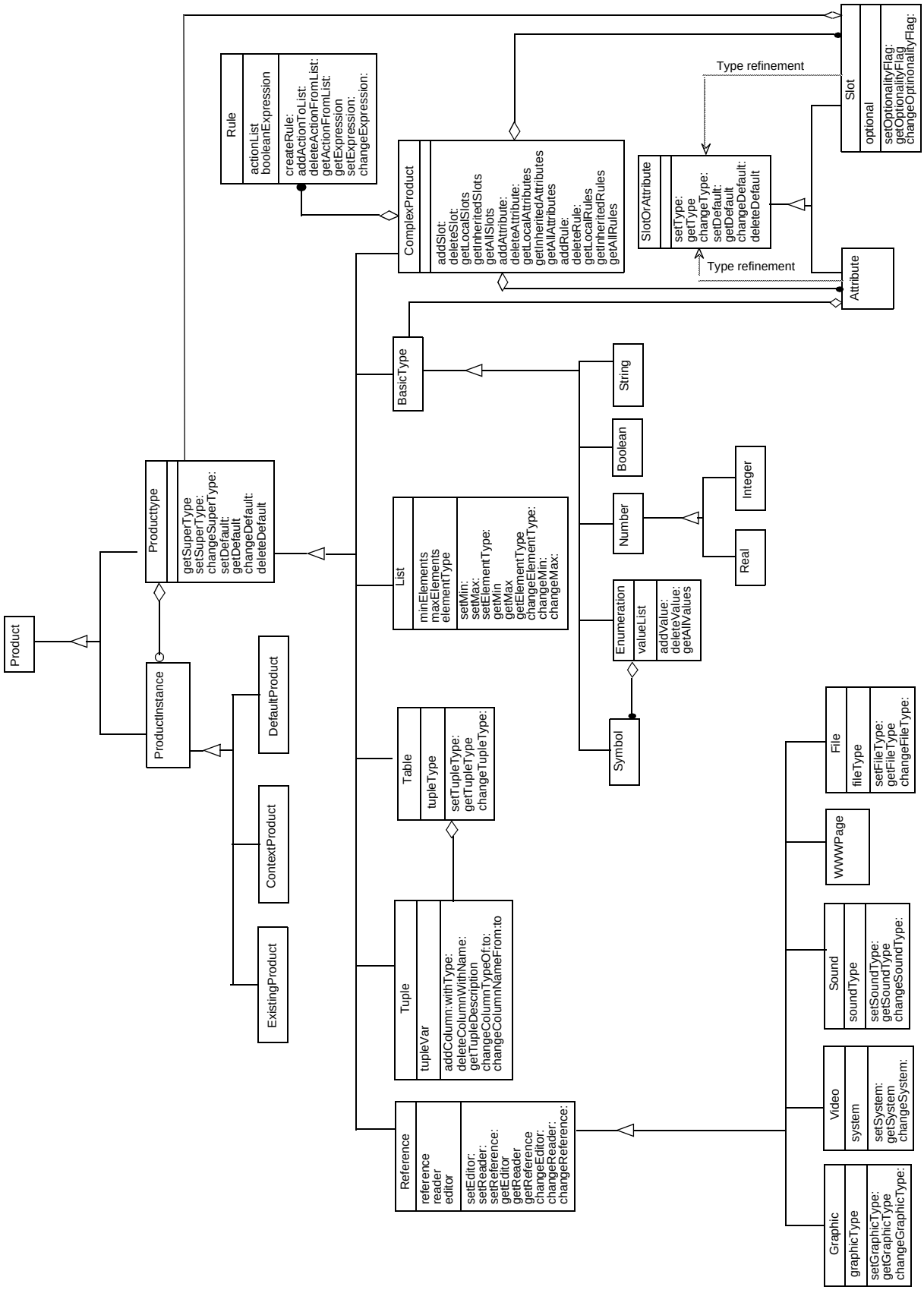


ABBILDUNG 21 Klassenstrukturen der Regeln

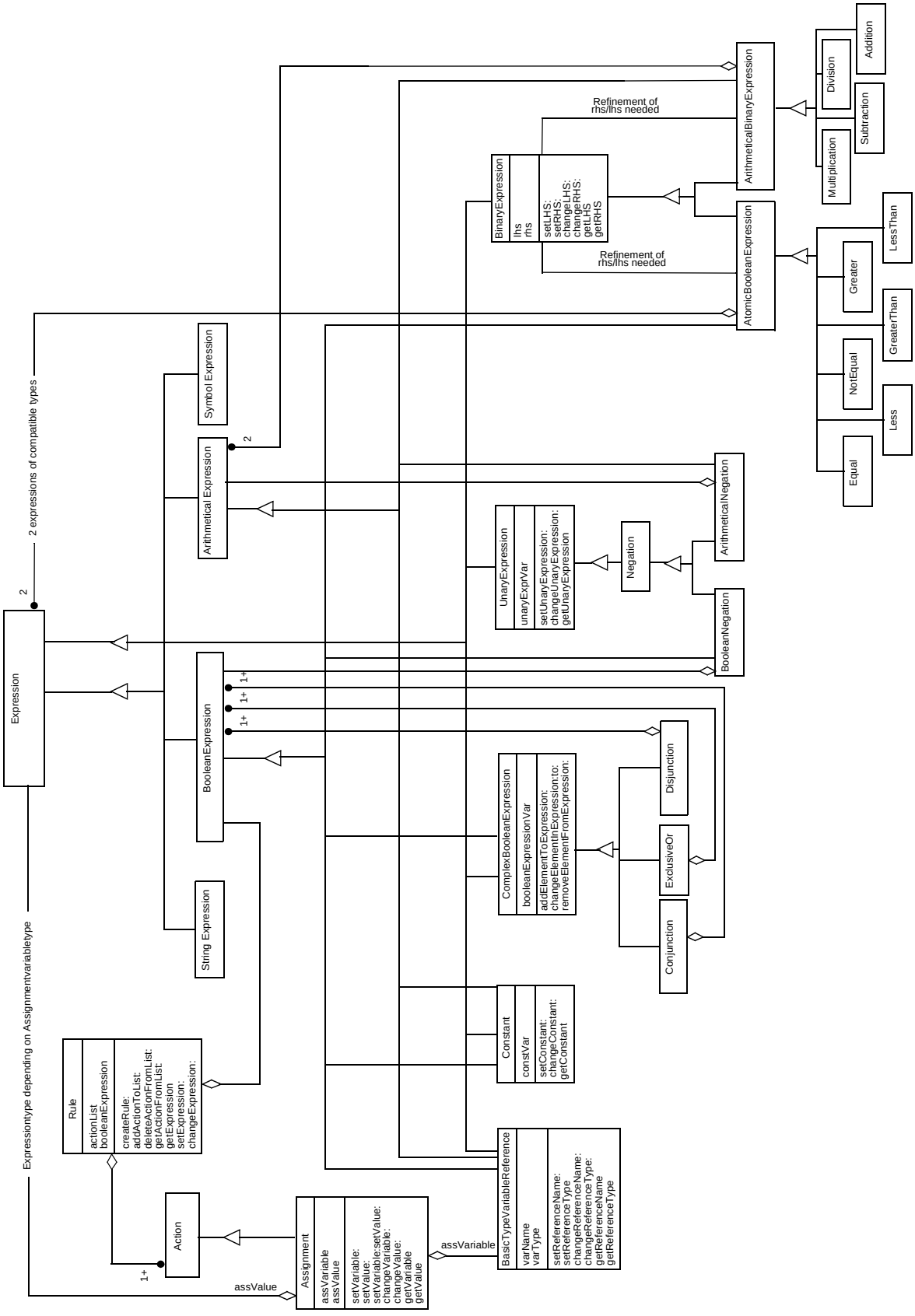


ABBILDUNG 22 Aufbau von Prozeßtypen