Diplomarbeit

Entwicklung und prototypische Realisierung eines Plug-In zur Unterstützung von UML-Profiles in Rational Rose

bearbeitet von

Andreas Pleuß

geboren am 29. Mai 1977

Technische Universität Dresden

Fakultät für Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl für Softwaretechnologie

Betreuer:	Dipl.-Inform. Mike Fischer
Externer Betreuer:	Dr. Andreas Speck
	Intershop Software Entwicklungs GmbH, Jena
Verantwortl. Hochschullehrer:	Prof.Dr.rer.nat.habil. Heinrich Hußmann

Eingereicht am 31. Oktober 2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Der Einsatz der *Unified Modeling Language* (*UML*) [Obj01d] der *Object Management Group* (*OMG*) [OMG02] als Sprache zur objektorientierten Modellierung hat weite Verbreitung gefunden. Dies beruht auch darauf, dass ein Kerngedanke bei der Entwicklung von UML war, eine möglichst allgemeine Modellierungssprache zu schaffen, um die Modellierung möglichst vieler verschiedener Bereiche mit einer einheitlichen Notation zu ermöglichen. Aufgrund dieser Allgemeinheit kann es Bereiche geben, in denen spezielle Eigenschaften nicht ausreichend eindeutig durch die UML in Modelle einbezogen werden können. Für solche Fälle stellt die UML Erweiterungsmechanismen – *Stereotypen*, *Constraints* und *Tagged Values* – zur Verfügung, durch die die UML an spezielle Bedürfnisse angepasst werden kann. Die Verwendung dieser eingebauten Erweiterungsmechanismen ermöglicht eine Anpassung der UML an spezielle Anwendungsbereiche, ohne das UML-Metamodell, durch das die UML definiert wird, selbst zu verändern.

Einzelne Erweiterungen zur Anpassung an einen speziellen Bereich möchte man zusammenfassen können. Dieses Bedürfnis wird mit Hilfe von *Profiles* zu erfüllen versucht. Profiles sollen dazu dienen, zusammengehörige Erweiterungen zusammenzufassen und zu strukturieren. Die Idee des Profile-Mechanismus entstand zunächst unabhängig von UML. In der Version 1.4 der UML-Spezifikation wurde der Mechanismus in die Spezifikation integriert. Parallel dazu wurden eine Vielzahl von UML-Profiles entwickelt. Einige davon sind bereits durch die OMG standardisiert worden, für weitere ist dies vorgesehen. Dementsprechend wird auch die Forderung nach Werkzeugunterstützung von UML-Profiles immer größer.

Für den Einsatz der UML ist eine breite Palette an Werkzeugen verfügbar [Jec02], die auch in der Industrie immer stärker genutzt werden. Da zum einen UML-Profiles in die UML-Spezifikation integriert wurden, und zum anderen die Erweiterung der UML durch UML-Profiles das UML-Metamodell nicht verändert, sollten UML-Profiles in bestehenden UML-Werkzeugen genutzt werden können. Infolgedessen sind zur Modellierung spezieller Domänen prinzipiell keine zusätzlichen Werkzeuge notwendig. Voraussetzung dafür ist, dass das jeweilige UML-Werkzeug den Profile-Mechanismus aus der UML-Spezifikation unterstützt. Dies ist bisher bei sehr wenigen Werkzeugen der Fall.

In dieser Arbeit soll daher ein Konzept zur Erweiterung eines bestehenden UML-

Werkzeuges um Profile-Unterstützung konzipiert und prototypisch realisiert werden. Dabei soll mit *Rational Rose* ein UML-Werkzeug mit hohem Verbreitungsgrad verwendet werden. Als Format zur Darstellung der Profiles soll das werkzeugunabhängige Austauschformat XMI dienen. Grundlage der Arbeit ist die aktuelle UML-Version 1.4. Die Überprüfung von UML-Modellen mittels im Profile enthaltener Constraints soll im Rahmen dieser Arbeit unberücksichtigt bleiben.

1.2 Praktische Relevanz in der Industrie

Eine konkrete praktische Relevanz der Problemstellung in der Industrie zeigt sich anhand der Anforderungen der Firma *Intershop*, einem führenden Anbieter von E-Commerce-Lösungen. Ziel ist dort eine Produktivitätssteigerung mittels systematischer Wiederverwendung. Dazu wird der Ansatz der produktlinienorientierten Softwareentwicklung verfolgt. Eine Kernaufgabe dabei ist die Modellierung von Gemeinsamkeiten und Unterschieden von Softwaresystemen. Da hierzu bis dato nur verschiedene Methoden und Vorgehensweisen für spezielle Anwendungsgebiete vorhanden waren, die keine konsistente Notationsform boten, wurde in [Cla01] ein Profile für Variabilität entwickelt. Dieses führt bestehende Ansätze unter der standardisierten und weithin akzeptierten Modellierungssprache UML zusammen und bietet darüber hinaus den Vorteil der Nutzbarkeit der zusätzlichen Möglichkeiten der UML.

Bei Intershop wird als UML-Werkzeug Rational Rose verwendet. Durch die Erweiterung von Rose um Profile-Unterstützung ergibt sich für Intershop aus der Entscheidung für UML zusätzlich zu den genannten Vorteilen der weitere wichtige Vorteil, dass die Modellierung von Variabilität mit vorhandenen Werkzeugen vorgenommen werden kann. Dies reduziert nicht nur Kosten, sondern kann auch die Integration in bestehende Strukturen und Arbeitsabläufe stark vereinfachen. Weiterhin kann dadurch die Akzeptanz neuer Entwicklungsmethoden verbessert werden.

Das Profile für Variabilität, als konkreter Anwendungsfall aus der Industrie, soll in dieser Arbeit als Leitlinie und für Beispiele dienen.

1.3 Aufbau der Arbeit

Zunächst wird in Kapitel 2 in das Aufgabengebiet eingeführt. Auf der Grundlage der Metamodellarchitektur der OMG werden UML-Profiles vorgestellt und in den Kontext der gegenwärtigen Entwicklungen eingeordnet.

Kapitel 3 untersucht die Repräsentation von UML-Profiles durch XMI. Ziel ist die Nutzung von XMI als werkzeugunabhängiges Austauschformat für UML-Profiles. Zunächst wird die allgemeine Umsetzung eines Profiles in XMI betrachtet, dann wird das UML-Profile für Variabilität als Beispiel herangezogen. Es werden jeweils auftretende Probleme und mögliche Lösungen diskutiert. Abschließend werden Schlußfolgerungen aus den identifizierten Problemen gezogen und wird die Eignung von XMI als Austauschformat für UML-Profiles bewertet.

In Kapitel 4 werden werkzeugunabhängig die grundsätzlichen Konzepte für eine Werkzeugunterstützung für UML-Profiles betrachtet. Dies beinhaltet in einem ersten Abschnitt die Anforderungen an eine Werkzeugunterstützung. Diese werden jeweils zunächst unabhängig von einer konkreten UML-Version formuliert und anschließend anhand konkreter Beispiele aus der aktuellen UML-Version – Version 1.4 – verdeutlicht. In einem zweiten Abschnitt werden unabhängig von einem konkreten Werkzeug

die grundsätzlichen Teilaufgaben einer Integration von Profile-Unterstützung in bestehende Werkzeuge diskutiert. Ein dritter Abschnitt enthält eine Zusammenstellung von Funktionalität im Kontext von UML-Profiles über den Rahmen der UML-Spezifikation hinaus.

Auf der Grundlage der vorangegangenen Kapitel wird in Kapitel 5 ein Konzept für die Integration von Profile-Unterstützung in das UML-Werkzeug *Rational Rose* vorgestellt. Zuerst wird die Ausgangslange – die bestehende Funktionalität und die Erweiterungsmöglichkeiten von Rose – beschrieben. Die beiden folgenden Abschnitte diskutieren ein Konzept zur Erstellung von Profiles sowie zur Anwendung von Profiles in Rose.

Kapitel 6 enthält die praktische Anwendung des Konzepts. Im ersten Abschnitt wird die prototypische Implementierung der Konzepte ausgeführt. Ein zweiter Abschnitt beschreibt die Anwendung des Prototyps auf das Profile für Variabilität als Nachweis der Verwendbarkeit und Grundlage der Bewertung. Zuletzt erfolgt in einem dritten Abschnitt die Bewertung der Konzepte, wobei vergleichend ein Werkzeug mit bereits integrierter Profile-Unterstützung, *Objecteering* von *Objecteering Software*, herangezogen wird.

Den Abschluß der Arbeit bilden Zusammenfassung und Ausblick in Kapitel 7.

Kapitel 2

Einführung und Stand der Forschung

Zunächst wird in den grundlegenden Problembereich, die Metamodellarchitektur der OMG, eingeführt. Auf dieser Grundlage wird anschließend der Mechanismus der UML-Profiles beschrieben. Zuletzt werden aktuelle Entwicklungen im Kontext von UML-Profiles aufgezeigt.

2.1 Die Metamodellarchitektur der OMG

Die *Unified Modeling Language* (*UML*) ist eine Sprache zur objektorientierten Modellierung. Sie wird durch die *Object Management Group* (*OMG*) [OMG02] – ein offenes Konsortium mit vielen Mitgliedern aus der IT-Industrie – standardisiert. Die UML-Spezifikation (aktuell Version 1.4 [Obj01d]) beschreibt die UML mittels eines *Metamodells*. Dieses Metamodell ist Teil einer ganzen *Metamodellarchitektur*, die ebenfalls durch die OMG standardisiert wird. Die OMG-Metamodellarchitektur besteht aus vier Schichten. Normale UML-Modelle befinden sich darin auf Schicht *M1*. Darunter, auf Schicht *M0*, befinden sich die konkreten Objekte, die Modellelemente aus einem UML-Modell instanziieren. UML-Modelle instanziieren ihrerseits (Meta-)Modellelemente aus dem Metamodell, welches sich auf Schicht *M2* befindet. Elemente aus einem Metamodell werden auch als *Metaklassen* bezeichnet. Die oberste Schicht, *M3*, enthält die *Meta Object Facility* (*MOF*), ein Meta-Metamodell, das alle Metamodelle beschreibt. Metaklassen sind somit Instanzen von Elementen der MOF. Die aktuelle Version der MOF ist die Version 1.4 [Obj02b]. UML 1.4 ist noch auf Basis von MOF 1.3 [Obj00a] definiert. Eine anschauliche Darstellung der Metamodellarchitektur befindet sich z. B. in [Jec00b]. Ausführungen zum Begriff des Metamodells und zum bei der OMG-Metamodellarchitektur verwendeten *Metaisierungsprinzip* (d. h. zur Sicht eines Metamodells auf ein Modell) finden sich in [Jec00a].

Neben der UML enthält die Metamodellarchitektur weitere durch die OMG standardisierte Metamodelle. Beispielhaft soll hier das *Common Warehouse Metamodel* (*CWM*) [Obj01a] genannt werden, das Schnittstellen für die Beschreibung und den Austausch von Daten in Warehouse-Anwendungen bietet. Prinzipiell sind beliebig viele Instanzen der MOF erlaubt.

Die Metamodelle auf Schicht M2 sind auf eine möglichst breite Verwendbarkeit ausgerichtet. So stellt die UML, als bekanntester Vertreter, generelle Konzepte zur ob-

jektorientierten Modellierung zur Verfügung und deckt daher ein großes Spektrum an Anwendungsfällen ab. Aufgrund dieser Allgemeinheit kann das Bedürfnis auftreten, die UML für die Modellierung von speziellen Anwendungsbereichen zu spezialisieren oder zu verfeinern. Beispiele für solche speziellen Anwendungsbereiche sind die Modellierung von Echtzeitsystemen oder von bestimmten Implementierungstechnologien, z. B. Komponenten-Standards wie *CORBA* oder Programmiersprachen wie *Java*. Um die UML an solche Bereiche anzupassen, muss sie erweitert werden. Dazu sind pinzipiell zwei Vorgehensweisen möglich:

- die Erweiterung der UML mittels der eingebauten Erweiterungsmechanismen und
- die Erweiterung der UML durch die Definition neuer Metaklassen.

Die erste Möglichkeit führt zu UML-Profiles und wird im folgenden Abschnitt betrachtet. Anschließend werden in einem weiteren Abschnitt UML-Profiles in den Kontext aktueller Entwicklungen eingeordnet. Dabei wird auch die zweite genannte Erweiterungsmöglichkeit aufgegriffen.

2.2 UML-Profiles

Die UML enthält eingebaute Mechanismen zu ihrer Erweiterung: *Stereotypen*, *Tagged Values* und *Constraints*. Diese erlauben, die UML zu erweitern, ohne das Metamodell der UML zu verändern.

Stereotypen sind der zentrale Erweiterungsmechanismus der UML. Sie erlauben eine UML-Metaklasse, wie z. B. `Class` oder `Association`, zu spezialisieren. Dazu werden sie Modellelementen zugeordnet, die diese Metaklasse instanziieren. Für jeden Stereotypen ist festgelegt, welche Metaklasse er spezialisieren kann. Diese wird als *Base-Class* des Stereotypen bezeichnet.

Ein Beispiel für einen Stereotypen wäre der Stereotyp `Entity` für die Base-Class `Class`, der eine Entität repräsentiert. Eine Entität ist eine spezielle Klasse, die zumeist ein Element der realen Welt repräsentiert und in einer Datenbank persistent gehalten werden soll. Der Stereotyp `Entity` beschreibt also einen Spezialfall der Metaklasse `Class`. Daher kann ein Stereotyp auch als eine (virtuelle) Metaklasse betrachtet werden und Modellelemente, die mit einem Stereotyp versehen sind, als Instanz einer virtuellen Metaklasse.

Neben der Spezialisierung einer Klasse kann ein Stereotyp auch für andere Zwecke verwendet werden, z. B. zur Unterscheidung zweier Modellelemente gleicher Struktur. Eine verfeinerte Klassifikation von Stereotypen findet sich in [BGJ99].

Für einen Stereotypen kann neben einer Base-Class optional auch ein *Icon* (d. h. eine kleine graphische Repräsentation) spezifiziert werden. Abbildung 2.1 zeigt die alternativen Möglichkeiten zur Notation von Stereotypen in Diagrammen. Die Standard-Darstellungsweise für Stereotypen ist ein Marker (*Label*), der den Namen des Stereotypen enthält. Sofern ein Icon für den Stereotypen definiert ist, kann dieses auch das Label ersetzen oder aber das gesamte Modellelement durch das Icon repräsentiert werden (*kollabierte* Darstellung). Einem Stereotypen können Tagged Values und Constraints zugeordnet sein.

Tagged Values bestehen aus einem Schlüssel und einem oder mehreren Werten. Sie dienen dazu, Modellelementen zusätzliche Eigenschaften zuzuweisen. Zu jedem Tagged Value gehört zur Definition seines Typs eine *Tag Definition*. Neben einem Typ

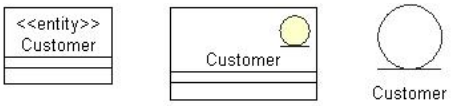

Abbildung 2.1: Die drei Alternativen der Darstellung von Sterotypen in Diagrammen.

legt diese auch fest, wieviele Werte minimal und maximal pro Tagged Value anzugeben sind. Es können beliebig viele Tagged Values durch dieselbe Tag Definition definiert sein. Als Typ kann eine Tag Definition statt einem Datentyp auch den Namen einer Metaklasse enthalten. In diesem Fall enthalten zugehörige Tagged Values anstatt Datenwerten Referenzen auf Instanzen dieser Metaklasse.

Tag Definitions sind (seit Version 1.4 der UML) an Stereotypen gebunden. Daher können Tagged Values nur Modellelementen zugeordnet werden, die mit dem jeweiligen Stereotypen versehen sind. Sie können deshalb als virtuelle Meta-Attribute bzw. -Referenzen betrachtet werden.

Constraints formulieren Einschränkungen oder Bedingungen für Modellelemente. Dazu enthalten sie einen booleschen Ausdruck in beliebiger Sprache, der zu jedem Zeitpunkt erfüllt sein muss. Mit der Object Constraint Language (OCL), die Teil der UML-Spezifikation ist, steht eine formale Sprache zur Formulierung von Constraints zur Verfügung.

Constraints können einerseits in Modellen definiert und einem oder mehreren Modellelementen zugeordnet werden. Andererseits werden sie auch Stereotypen zugeordnet, um Regeln für virtuelle Metaklassen zu spezifizieren. Im letzteren Fall bezieht sich die Constraint auf die Metamodellebene.

Um die UML für einen speziellen Anwendungsbereich zu erweitern, sind üblicherweise eine größerer Menge einzelner Stereotypen, Tag Definitions und Constraints notwendig. Diese möchte man zusammenfassen können. Dieses Bedürfnis wird durch UML-Profiles zu erfüllen versucht. Die ersten Vorschläge für ein Profile-Konzepts in [OMG02] bezogen sich nicht konkret auf die UML, sondern generell auf alle Metamodelle der OMG-Metamodellarchitektur. In [Obj99] wurden Anforderungen für Profiles definiert, die sich konkret auf die UML bezogen. Parallel dazu wurden in der UML-Spezifikation 1.3 [Obj00c] Beispiele für UML-Profiles angeführt. In der Version 1.4 [Obj01d] wurde der Profile-Mechanismus in die UML-Spezifikation integeriert. Für andere OMG-Metamodelle existiert kein gesonderter Profile-Mechanismus, so dass im Folgenden unter Profiles stets UML-Profiles zu verstehen sind.

Die Integration des Profile-Mechanismus in die UML-Spezifikation wurde mittels der UML-Erweiterungsmechanismen vorgenommen. So werden Profiles als Pakete dargestellt, die mit dem Stereotyp `profile` versehen sind. Der Profile-Mechanismus fordert, dass zu einem Profile stets die Untermenge des Metamodells angegeben wird, auf die das Profile angewendet werden kann. Dazu dient in UML 1.4 die Tag Definition `applicableSubset`, die dem Stereotypen `profile` zugeordnet ist. Um die Anwendung eines oder mehrerer Profiles in einem Modell kenntlich zu machen, steht in UML 1.4 der Stereotyp `appliedProfile` zur Verfügung. Dieser wird einer Abhängigkeit zwischen einem Profile und einem Modell (bzw. einem Paket) zugeordnet.

Die wesentlichste semantische Festlegung zu Profiles in UML 1.4 ist die Beschränkung des Inhaltes auf die UML-Erweiterungsmechanismen. Zusätzlich darf ein

Profile anwenderdefinierte Datentypen enthalten, die als Typ von Tag Definitions zu verwenden sind. Profiles dürfen spezialisiert werden und verhalten sich ansonsten wie normale Pakete auch.

Im Zusammenhang mit Profiles werden in UML 1.4 auch Modellbibliotheken eingeführt. Diese sollen wiederverwendbare Elemente enthalten; im wesentlichen Klassen und Datentypen. Eine Modellbibliothek wird als ein Paket dargestellt, das mit einem Stereotypen `modelLibrary` versehen ist. Zwischen einem Profile und einer Modellbibliothek kann eine Abhängigkeit bestehen, die ebenfalls mit einem Stereotypen `modelLibrary` versehen ist. Dies bewirkt, dass bei Anwendung des Profiles auf ein Modell die Modellbibliothek automatisch zur Verfügung steht.

Neben diesen Definitionen zur Integration von Profiles in das UML-Metamodell enthält die UML-Spezifikation 1.4 zwei Vorschläge zur Notation der Definition von Stereotypen: eine graphische Notation und eine tabellarische Notation.

Eine ausführlichere Darstellung aller beschriebenen Definitionen findet sich in [Ple01].

2.3 Aktuelle Entwicklungen

Mittlerweile wurde bereits eine Vielzahl an Profiles entwickelt. Nach [GGW00] können diese unterteilt werden in Profiles

- zur generellen Erweiterung der UML z. B. um zeitliche Aspekte, wie im *Profile for Schedulability, Performance, and Time* [Obj02e],
- für Implementierungstechnologien (d. h. Plattformen, Komponentenstandards oder Programmiersprachen), wie die Profiles für CORBA [Obj02c] oder EJB [Rat01a] und
- für bestimmte Anwendungsbereiche wie z. B. komponentenbasierte verteilte Unternehmenssoftware (*UML Profile for Enterprise Distributed Object Computing*, EDOC, [Obj02d]) oder Navigation und Präsentation in Web-Anwendungen [KBHM00].

Einige dieser Profiles wurden bereits durch die OMG standardisiert und weitere sind auf dem Weg dazu.

Viele der bestehenden Profiles werden in ihrer Spezifikation nicht mittels der UML-Erweiterungsmechanismen beschrieben, sondern als ein MOF-konformes Metamodell (Schicht M2). Die Möglichkeit der Erweiterung der UML durch Definition neuer Metaklassen wurde bereits in Abschnitt 2.1 angesprochen.

Ein virtuelles Metamodell – d. h. ein Profile konform zu UML 1.4, definiert durch die UML-Erweiterungsmechanismen – kann stets auf ein MOF-Metamodell abgebildet werden. Umgekehrt ist dies nicht immer möglich, da Stereotypen ausschließlich vorhandene Metaklassen spezialisieren dürfen. Enthält ein Metamodell ein Element, das nicht als Spezialisierung einer UML-Metaklasse dargestellt werden kann, so kann dieses Metamodell nicht als UML 1.4-konformes Profile spezifiziert werden. Die Erweiterung der UML um eigenständige Metaklassen wird als *First-Class-Erweiterung* bezeichnet.

Grundsätzlich wird ein spezieller Anwendungsbereich einfacher und präziser durch ein Metamodell beschrieben. Die Beschränkung auf die vorhandenen UML-Erweiterungsmechanismen wird deshalb kritisch diskutiert. [NDK99] schlägt verschiedene Verbesserungen des UML-Erweiterungsmechanismus vor. In [SW01] werden

grundsätzliche Ansätze zur Metamodellierung für UML untersucht und wird ein Mechanismus zur Fist-Class-Erweiterung der UML gefordert. Ein konkretes Konzept zur Integration von First-Class-Erweiterungsmechanismen in die UML findet sich in [DSB99].

Demgegenüber stehen die praktischen Vorteile der bisherigen UML-Erweiterungsmechanismen. Wichtige Vorteile sind die einfachere Handhabbarkeit sowie die Möglichkeit der Verwendung der Erweiterungen in bestehenden UML-Werkzeugen. [Des01] legt weitere Vorteile dar und plädiert für eine Verwendung von Profiles konform zu UML 1.4. In [Des00] werden die beiden Alternativen – Profiles nach UML 1.4 und Metamodellierung – gegenüber gestellt, und Vorschläge gemacht, welche Alternative in welcher Situation verwendet werden sollte.

Der Schwerpunkt der aktuellen Entwicklung bei der OMG liegt bei der *Model Driven Architecture* (*MDA*, [Obj01c], [Obj00b] und [Obj01b]). Diese bindet Profiles in einen Entwicklungsprozess ein. Basis der MDA sind bestehende Standards wie UML, MOF, CWM (siehe oben) und XMI (siehe Abschnitt 3). Kernidee ist die Trennung der spezifischen Systemfunktionalität von der konkreten Implementierungstechnologie. Dazu wird zwischen zwei Arten von Modellen unterschieden: das plattformunabhängige Modell (*platform independent model*, *PIM*) und das plattformspezifische Modell (*platform specific model*, *PSM*). Der Entwicklungsprozess sieht zunächst die Erstellung eines PIM für eine Anwendung vor. Dabei ist vorgesehen, Profiles für bestimmte Anwendungsbereiche zu verwenden, insbesondere das Profile für EDOC (siehe oben). Anschließend wird das PIM durch standardisierte Abbildungen auf ein oder mehrere PSMs abgebildet. Dadurch werden unter Wiederverwendung des PIM Anwendungen für verschiedene Implementierungstechnologien erstellt. PSMs nutzen üblicherweise Profiles für die jeweilige Implementierungtechnologie. Die Abbildung vom PIM auf PSM ist oft Teil der Spezifikation von Profiles, die für den Einsatz mit der MDA vorgesehen sind.

Kapitel 3

Repräsentation und Austausch von UML-Profiles durch XMI

Die Sprache *XMI* (*XML Model Interchange*) ist ein standardisiertes und werkzeugunabhängiges Format für die Beschreibung und den Austausch von MOF-konformen Modellen, z. B. UML-Modellen. Es wird durch die OMG [OMG02] standardisiert und basiert auf dem Standard *XML* (*Extensible Markup Language*) des W3C [W3C02]. Die aktuelle Version der XMI ist die Version 1.2 [Obj01e], der Austausch von Modellen nach UML 1.4 basiert auf XMI 1.1 [Obj00d].

Die Beschreibung eines Modells in XMI basiert auf dem Metamodell des jeweiligen Modells. Für jede Metaklasse des Metamodells und für jede Eigenschaft (Attribute und Referenzen) aller Metaklassen sind Einträge in einer XMI-Datei möglich. Für jedes Metamodell ist eine *Document Type Definition* (*DTD*) vorhanden, die alle diese möglichen Einträge definiert und zur Verifikation von XMI-Dateien dient. Da die DTD genau das Metamodell abbildet, kann sie direkt aus einem Metamodell generiert werden, sofern dieses in geeigneter (elektronisch verwertbarer) Form vorliegt. Bei der DTD [Obj01f] zum Metamodell für UML 1.4 ist dies der Fall.

Da UML-Profiles in der UML-Spezifikation in das Metamodell integriert wurden, sollte theoretisch XMI zur Darstellung von UML-Profiles ohne weiteres nutzbar sein. Dies soll in diesem Kapitel untersucht werden. Es wird bewertet, inwieweit XMI tatsächlich zur vollständigen Repräsentation von Profiles geeignet ist, wobei für mögliche Probleme Lösungsansätze vorgeschlagen werden.

Wichtig ist auch der Aspekt, dass im Zuge der Umsetzung von UML-Profiles in XMI eine genauere Überprüfung der Definition von Profiles in der UML-Spezifikation stattfindet. Zum einen erfordert die Beschreibung in XMI, dass alle Bestandteile des Profiles ausschliesslich in Konstrukten gemäß der abstrakten Syntax des UML-Metamodells dargestellt werden, da die DTD anhand dieser eine XMI-Datei validiert. Zum anderen soll bei der Umsetzung eines Profiles gemäß der abstrakten Syntax nicht die Semantik der UML-Spezifikation verletzt werden, um nicht ein ungültiges UML-Modell zu repräsentieren. Dadurch resultiert aus der schrittweisen Transformation von textuell und graphisch beschriebenen Profiles nach XMI auch eine schrittweise Überprüfung von abstrakter Syntax und Semantik. Als Ergebnis liefert dieses Kapitel auch Aussagen über die Korrektheit und Vollständigkeit der Definition von UML-Profiles in der UML-Spezifikation. Probleme werden dargestellt und Lösungsansätze diskutiert.

Die Untersuchungen im ersten Teil dieses Kapitels beziehen sich zunächst auf

UML-Profiles im Allgemeinen, so wie diese durch die UML-Spezifikation vorgesehen sind. Darauf basierend wird im zweiten Teil ein konkretes Profile, das UML-Profile für Variabilität [Cla01] in XMI repräsentiert. Dabei treten zusätzliche Probleme auf, die diskutiert werden. Den dritten Teil dieses Kapitels bildet die abschließende Bewertung, in der sowohl die Aspekte der Beschreibung von Profiles, als auch der Aspekt des Austauschs zwischen CASE-Werkzeugen berücksichtigt werden.

Die resultierende XMI-Repräsentation von Profiles dient als Basis der weiteren Arbeit, da dort Profiles und Modelle mittels XMI ausgetauscht werden sollen.

3.1 Allgemein

In diesem Abschnitt wird unabhängig von einem konkreten Profile untersucht, wie die in der UML-Spezifikation vorgegebenen einzelnen Bestandteile und Mechanismen von Profiles in XMI zu repräsentieren sind. Demnach orientiert sich dieser Abschnitt stark an den gegeben Spezifikationen. Im ersten Abschnitt wird das UML-Austausch-Metamodell betrachtet, das die Grundlage für den Austausch von UML-Modellen durch XMI bildet. Im zweiten Abschnitt werden Profile-spezifische Punkte aufgegriffen. Es werden mögliche Probleme diskutiert und Lösungsmöglichkeiten vorgeschlagen.

3.1.1 Das Austausch-Metamodell

Für den Austausch von UML-Modellen sind in der UML-Spezifikation 1.4 [Obj01d] zwei alternative Mechanismen vorgesehen: XMI in der Version 1.1 [Obj00d] und Abbildung auf CORBA IDL basierend auf der MOF-Spezifikation 1.3 [Obj00a]. Beide Mechanismen setzen ein Metamodell voraus, das konform zur MOF-Spezifikation ist: die XMI-Spezifikation 1.1 gilt für Metamodelle, die durch MOF dargestellt werden können ([Obj00d], S. 1-1), die Regeln zur Abbildung eines Metamodells auf CORBA IDL sind Bestandteil der MOF-Spezifikation selbst und setzen die Einhaltung aller MOF-Vorschriften voraus ([Obj00a], S. 5-33). Das UML-Metamodell erfüllt jedoch diese Voraussetzung nicht. Beispielsweise enthält es Assoziationsklassen (wie `ElementOwnership`, siehe [Obj01d], S. 2-188, Abb. 2-32). Diese sind kein gültiges MOF-Element.

Deshalb wird in der UML-Spezifikation zusätzlich in Kapitel 5 eine MOF-konforme Realisierung des Metamodells eingeführt, das UML-Austausch-Modell (*UML Interchange Metamodel*), oft auch bezeichnet als *physisches* (*physical*) Metamodell (im Gegensatz zum *logischen* Metamodell). Es unterscheidet sich vom logischen Metamodell nur soweit wie zur Einhaltung der MOF-Spezifikation notwendig. Die Änderungen (siehe [Obj01d], S. 5-2) lassen sich wie folgt zusammenfassen:

- MOF-konforme Namensgebung aller Metamodellelemente durch Änderung bzw. Hinzufügen von Namen und Präfixen,
- Ersetzung von Assoziationsklassen durch normale Klassen, da Assoziationsklassen in MOF-Metamodellen nicht erlaubt sind und
- Einführung von MOF-Referenzen zusätzlich zu den Assoziationen, um eine einfachere Navigation zu ermöglichen.

Im logischen Metamodell sind keine Navigationsrichtungen für Assoziationen angegeben. Die MOF-Referenzen im physischen Metamodell dagegen sind nur an Assoziationsenden vorhanden, entlang derer auch tatsächlich navigiert werden soll. Demnach werden durch die MOF-Referenzen im physischen Metamodell Navigationsrichtungen definiert.

Eine weitere Anforderung an das Austausch-Metamodell ist die Eliminierung zyklischer Abhängigkeiten zwischen Paketen des Metamodells. Diese Forderung besteht zwar nicht grundsätzlich für MOF-konforme Metamodelle, jedoch ist es eine Voraussetzung für die Abbildung eines Metamodells auf CORBA IDL ([Obj00a], S. 5-33). Das logische Metamodell enthält eine wechselseitige Abhängigkeit zwischen den Paketen `Core` und `Extension Mechanisms` ([Obj01d], S. 2-6). Ein Beispiel zeigt, dass beide dieser Abhängigkeiten benötigt werden: ein Modellelement (aus dem Paket `Core`) muss einen Tagged Value (aus dem Paket `Extension Mechanisms`), mit dem es versehen ist, referenzieren können und ein Tagged Value kann gemäß seiner Definition statt eines Datenwertes eine Referenz auf ein Modellelement beinhalten. Deshalb wird im physischen Metamodell das Paket ganz auf das Paket `Extension Mechanisms` verzichtet. Die Elemente daraus befinden sich im physischen Metamodell im Paket `Core`. Zusätzlich wird die Pakethierarchie verflacht, so dass alle Pakete ohne weitere Untergliederung im obersten Paket – dem Paket `UML` – enthalten sind ([Obj01d], S. 5-3).

Die Abbildung 3.1 zeigt einen Ausschnitt aus dem physischen Metamodell mit den Erweiterungsmechanismen (angelehnt an [Obj01d], S. 5-9). Die Abbildung enthält zusätzlich für jede Assoziation Navigationspfeile sowie zu jeder Klasse deren vollständige Attribute und Referenzen.

Aus dem physischen Metamodell wurde für den Austausch von Modellen durch XMI eine DTD [Obj01f] generiert, basierend auf XMI 1.1.

3.1.2 Probleme und Lösungsansätze

Durch das physische Metamodell (siehe Abschnitt 3.1.1) und die XMI-Spezifikation [Obj00d] sind die Beschreibung und der Austausch von UML-Modellen durch XMI spezifiziert. Da seit der UML-Spezifikation 1.4 [Obj01d] UML-Profiles als UML-Modelle beschrieben werden können (als stereotypisiertes Paket, siehe Abschnitt 2.2), sollten UML-Profiles theoretisch ohne weiteres in XMI repräsentierbar sein. Bisher sind dazu aber noch keine Beispiele oder praktischen Erfahrungen vorhanden. Auch wenn ein Profile als normales UML-Modell betrachtet werden kann, müssen dennoch die Eigenheiten eines Profiles berücksichtigt werden. Dabei zeigt sich, dass die vorhandenen Beispiele (z. B. [Obj01d], Abschnitt 5.2), die sich ausschließlich mit einfachen konventionellen UML-Modellen beschäftigen, nicht auf Profiles übertragbar sind. Weiterhin stellt sich heraus, dass die Definition von UML-Profiles in der UML-Spezifikation noch nicht alle praktischen Anforderungen an UML-Profiles abdeckt. In diesem Abschnitt sollen kritische Punkte untersucht und mögliche Lösungsansätze für auftretende Probleme diskutiert werden.

3.1.2.1 Profiles und ihre Anwendung auf Modelle

Da Profiles in mehreren Modellen verwendet werden sollen, müssen sie unabhängig von einem Modell definiert werden können. Daher sollte das Profile sich auch in einer separaten XMI-Datei befinden.

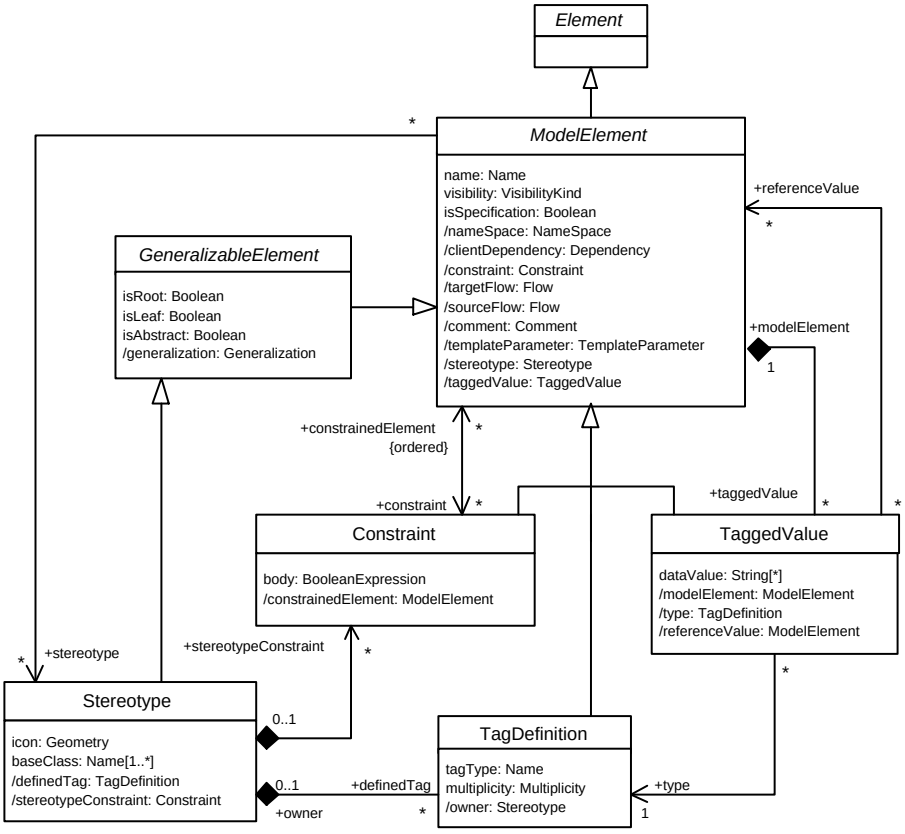

Abbildung 3.1: Ausschnitt aus dem physischen Metamodell, Paket `Core`. Zu jeder Metaklasse sind alle Attribute und Referenzen angegeben. Unidirektionale Assoziationen sind durch Navigationspfeile kenntlich gemacht.

Eine grundsätzliche Voraussetzung für die Wiederverwendbarkeit eines Paketes ist die Unabhängigkeit von Modellen, von denen es verwendet wird, d. h. es soll bei der Anwendung des wiederverwendbaren Paketes nicht notwendig sein, Referenzen von Elementen des Paketes auf Elemente des Modells zu erstellen. Hier greift, dass im physischen Metamodell Navigationsrichtungen angegeben sind (siehe Abschnitt 3.1.1): dadurch wird es möglich, dass nur eine einseitige Abhängigkeit vom Modell auf ein wiederverwendbares Paket besteht. Im Falle eines Profiles soll dieses also unabhängig sein von den Modellen, auf die es angewandt wird.

Bei Betrachtung des physischen Metamodells stellt man fest, dass alle Referenzen zwischen Elementen aus Profiles und Elementen aus Modellen so gestaltet sind, dass sie diese Bedingung erfüllen. Dies soll am Beispiel von Stereotypen verdeutlicht werden: während im logischen Metamodell eine bidirektionale Assoziation zwischen Stereotypen und Modellelementen besteht, enthält das physische Metamodell lediglich noch eine unidirektionale Referenz von Modellelementen auf Stereotypen (siehe Abbildung 3.1). Dadurch ist der Stereotyp vom Modellelement unabhängig und damit auch das Profile vom Modell.

Auch bei den anderen Bestandteilen eines Profiles werden nur auf Seiten des Modells und den darin enthaltenen Modellelementen Referenzen benötigt (siehe Abbildung 3.1):

- Modellelemente im Modell referenzieren Stereotypen aus dem Profile, mit denen sie versehen sind,
- Tagged Values im Modell referenzieren die zugehörigen Tag Definitions aus dem Profile,
- Datentypen werden von Tagged Values referenziert und
- zwischen dem Modell selbst und dem Profile besteht eine Abhängigkeit (*Dependency*), die mit dem Stereotypen `appliedProfile` versehen ist (siehe Abschnitte 2.2 und 3.1.2.2).

Bei der Abhängigkeit zwischen Modell und Profile nimmt das Modell die Rolle des Abnehmers (*client*) und das Profile die Rolle des Anbieters (*supplier*) ein. Da im physischen Metamodell nur eine wechselseitige Referenz zwischen Client und Dependency und eine einseitige Referenz von Dependency zum Supplier definiert ist, sind auch hier keine Referenzen auf Profile-Seite nötig. Die Dependency sollte somit in die XML-Datei, in der sich auch das Modell befindet, mit aufgenommen werden. Alle Referenzen werden mit dem XMI-Mechanismus des XLink bzw. XPointer [Wor01a] realisiert.

3.1.2.2 Semantik der Abhängigkeit „appliedProfile“

Die Anwendung eines Profiles auf ein Modell wird mit einer Abhängigkeit dargestellt, die mit dem Stereotyp `appliedProfile` versehen ist (siehe Abschnitt 2.2). Die Semantik dieser Abhängigkeit bezüglich des Zugriffs auf Elemente des Profiles ist in der UML-Spezifikation nicht explizit definiert.

Prinzipiell enthält die UML-Spezifikation für den Zugriff auf Elemente eines anderen Paketes zwei alternative Mechanismen ([Obj01d], S. 2-197): entweder werden die benötigten Elemente importiert, d. h. sie werden dem Namensraum des importierenden Paketes hinzugefügt (durch eine Abhängigkeit mit dem Stereotyp `import`), oder die Elemente werden über ihren vollständigen Pfadnamen referenziert (durch eine Abhängigkeit mit dem Stereotyp `access`).

Für die Abhängigkeit zwischen einem Modell und einem verwendeten Profile ist es sinnvoll, die Semantik analog zu einer Abhängigkeit mit dem Stereotyp `access` zu interpretieren. Dadurch werden die Elemente des Profiles durch ihren vollen Pfadnamen referenziert, wodurch ohne Umbenennungen Namenskonflikte zwischen Stereotypen aus mehreren Profiles vermieden werden. Für diese Interpretation spricht auch, dass die Spezifikation in diesem Zusammenhang das Wort „access“ verwendet ([Obj01d], S. 2-199).

3.1.2.3 Modellbibliotheken

Analog zu Profiles, sollen auch Modellbibliotheken wiederverwendbar und unabhängig von einem Modell sein (siehe Abschnitt 3.1.2.1). Daher sollte jede Modellbibliothek in eine eigene XMI-Datei separiert werden. Weiterhin gilt auch hier die Voraussetzung der Unabhängigkeit (analog Abschnitt 3.1.2.1), d. h. zur Verwendung von Elementen aus einer Modellbibliothek in Modellen sollen keine Referenzen auf Seiten der Modellbibliothek notwendig sein.

Laut Spezifikation sind Modellbibliotheken vergleichbar mit Klassenbibliotheken in Programmiersprachen ([Obj01d], S. 2-191) und enthalten wiederverwendbare Elemente, wie z. B. Klassen oder Datentypen ([Obj01d], S. 2-188). Weiter sind sie nicht definiert und können damit außer Klassen und Datentypen theoretisch auch beliebige andere wiederverwendbare Elemente enthalten. Datentypen können auch in Profiles enthalten sein. Ihre Unabhängigkeit bei ihrer Verwendung wurde bereits festgestellt: es bestehen nur unidirektionale Referenzen auf den Datentyp, z. B. ausgehend von einem Parameter oder einem Attribut.

Klassen werden zumeist als Teilnehmer von Beziehungen (z. B. Generalisierung, Assoziationen) verwendet. Die Metaklasse `Class` im Metamodell ist eine Unterklasse von `Classifier` und besitzt nur geerbte Referenzen. Bei Assoziationen bestehen nur unidirektionale Referenzen von den Assoziationsenden zum Classifier. Bei Generalisierungen bestehen nur unidirektionale Referenzen von der Generalisierung auf die Oberklasse und bidirektionale Referenzen zwischen der Generalisierung und der Unterklasse, d. h. eine Klasse in einer Modellbibliothek darf in Modellen spezialisiert werden. Bei einem Flow bestehen bidirektionale Referenzen sowohl zwischen Quelle und Flow als auch zwischen Ziel und Flow, d. h. ein Element aus einer Modellbibliothek kann keine solche Beziehung zu Elementen aus Modellen haben. Dies gilt auch für alle anderen Unterklassen von `Classifier`, z. B. `Interface`.

Soll eine Modellbibliothek andere Elemente enthalten, so ist jeweils für den gewünschten Anwendungszweck zu überprüfen, ob das physische Metamodell die Wiederverwendung erlaubt. Das ist z. B. bei Constraints nicht der Fall (siehe Abschnitt 3.2.1.2).

3.1.2.4 Semantik der Abhängigkeit „ModelLibrary“

Gehört zu einem Profile eine Modellbibliothek, so wird dies durch eine Abhängigkeit mit dem Stereotypen `modelLibrary` zwischen dem Profile und der Modellbibliothek kenntlich gemacht (siehe Abschnitt 2.2). Die Semantik dieser Abhängigkeit ist in der UML-Spezifikation nicht weiter definiert. Es bleibt damit offen, auf welchem Weg, formal betrachtet, ein Modell auf Elemente in der Modellbibliothek zugreifen soll.

Wird ein Profile mit einer zugehörigen Modellbibliothek auf ein Modell angewendet, so besteht zunächst noch keine Abhängigkeit zwischen der Modellbibliothek und dem Modell. Da ein Profile auch mehrere Modellbibliotheken mit sich bringen kann,

ist es sinnvoll, explizit eine Abhängigkeit zwischen Modell und Modellbibliothek zu erstellen, wenn diese vom Modell genutzt werden soll. Wie bei normalen Paketen kann diese je nach Wunsch stereotypisiert sein (also z. B. mit dem Stereotyp `access` oder `import`). Meistens wird eine Abhängigkeit mit dem Stereotyp `access` sinnvoll sein, da hier die verwendeten Elemente mit ihrem Pfadnamen referenziert werden und so sichtbar bleibt, aus welchem Paket das Element stammt. Der Zugriff auf Elemente einer Modellbibliothek erfolgt also nicht über die Abhängigkeit `modelLibrary`, sondern genauso wie bei normalen Paketen.

3.1.2.5 Standard-Elemente aus der Spezifikation

Die Erweiterungsmechanismen der UML, Stereotypen, Constraints und Tagged Values, werden auch in der UML-Spezifikation selbst verwendet, um Elemente zu spezifizieren, die nicht komplex genug sind, um als eine eigenständige Metaklasse definiert zu werden. In der Spezifikation sind sie in den Paketen der ihnen zugehörigen Metaklassen definiert und darüber hinaus in Anhang A unter dem Begriff Standard-Elemente (*standard elements*) zusammengefasst. Ein Beispiel sind die bereits erwähnten Stereotypen `import` und `access` für Abhängigkeiten zwischen Namensräumen.

Auch zur Integration des Profile-Mechanismus in UML wurden Standard-Elemente eingeführt, z. B. `profile` für die Metaklasse `Package` oder die zugehörige Tag Definition `applicableSubset`. Daher werden bei der Repräsentation von UML-Profiles in XMI Standard-Elemente benötigt. Die Standard-Elemente sind aber nicht im physischen Metamodell oder in der zugehörigen DTD enthalten; diese enthalten nur die (nicht-virtuellen) Metaklassen. Die UML-Spezifikation weist bezüglich des Austauschs mittels CORBA IDL darauf hin, dass auch die Standard-Elemente zur Verfügung gestellt werden müssen, im Zusammenhang mit XMI werden die Standard-Elemente nicht erwähnt.

Die einfachste Lösung, um dennoch Standard-Elemente in XMI zu verwenden, wäre, sie in jeder Datei, in der sie benötigt werden, neu selbst zu definieren. Dies wäre jedoch umständlich und fehleranfällig. Weiterhin wäre so nicht direkt erkennbar, dass es sich nicht um ein selbstdefiniertes Element, sondern um eines der Standard-Elemente handelt.

Deshalb ist es die günstigere Lösung, eine zentrale XMI-Datei einzuführen, die alle Standard-Elemente wiederverwendbar für die Nutzung mit XMI zur Verfügung stellt. Benötigte Elemente daraus werden per XLink und XPointer referenziert. Letztendlich ist diese Datei nichts anderes als ein „UML-Profile für UML“. Demzufolge kann als oberstes Element ein Paket mit dem Stereotype `profile` gewählt werden.

Im Zusammenhang mit den Standard-Elementen sind auch die Abschnitte 3.1.2.6 und 3.2.1.2 zu beachten.

3.1.2.6 Namenskonflikte zwischen Stereotypen

In einem Namensraum (Metaklasse `Namespace`) muss jedes Modellelement einen eindeutigen Namen haben ([Obj01d], S. 2-44). Ein Modellelement, das explizit Teil eines anderen ist (durch eine Komposition im Metamodell), hat dieses als (virtuellen) Namensraum ([Obj01d], S. 2-44). Da Tagged Values und Constraints in einem Profile Teil eines Stereotypen sind (siehe Abbildung 3.1), ist bei diesen Elementen nur auf eindeutige Namen innerhalb eines Stereotypen zu achten. Die Namen von Stereotypen dagegen müssen innerhalb des Pakets, in dem sie sich befinden, eindeutig sein. Ein Profile darf folglich keine zwei Stereotypen gleichen Namens enthalten.

Werden Stereotypen aus einem Profile in einem Modell verwendet, so könnten Namenskonflikte auftreten, wenn sie in den Namensraum des Modells importiert würden. Da die Abhängigkeit zwischen Modell und Profile so interpretiert wird, dass auf die Elemente des Profiles nur über ihren vollen Pfadnamen zugegriffen wird (siehe Abschnitt 3.1.2.2), sind Namenskonflikte vermieden, und Stereotypen gleichen Namens aus verschiedenen Profiles bleiben unterscheidbar.

Ein Problem tritt bei Stereotypen aus den Standard-Elementen auf: im Anhang A der UML-Spezifikation sind mehrere Stereotypen gleichen Namens aufgeführt. Teilweise stammen diese Stereotypen aus verschiedenen Paketen, so dass kein Namenskonflikt vorhanden ist. Dies gilt beispielsweise für den Stereotypen `destroy` für die Metaklasse `BehavioralFeature` aus dem Paket `Core` ([Obj01d], S. 2-25) und den Stereotypen `destroy` für die Metaklasse `CallEvent` aus dem Paket `State Machines` ([Obj01d], S. 2-149). Andere Stereotypen gleichen Namens sind jedoch im gleichen Paket definiert, wie z. B. die Stereotypen `implementation` für die Metaklassen `Class` ([Obj01d], S. 2-27) bzw. `Generalization` ([Obj01d], S. 2-40), jeweils im Paket `Core`. Hier liegt offensichtlich ein Mangel der UML-Spezifikation vor.

Möchte man die Standard-Elemente in einer zentralen XMI-Datei zur Verfügung stellen (siehe Abschnitt 3.1.2.5), so bleibt zunächst nur die Möglichkeit, die betreffenden Stereotypen umzubenennen oder die Konflikte zunächst zu ignorieren. Bei einer Umbenennung besteht die Schwierigkeit, dass Stereotypen anhand ihres Namens identifiziert werden und ein neuer Name nicht mehr die Bedeutung des Stereotypen erkennen lässt. Eine Möglichkeit wäre, die neuen Namen durch Erweiterung des ursprünglichen Namens zu bilden; z. B. durch Anhängen des Namens der Base-Class.

Durch das Problem drängt sich die Überlegung auf, ob zukünftig nicht als zusätzliches Attribut für Stereotypen ein Marker eingeführt werden sollte (z. B. als Metaattribut `label` vom Typ `String`), der statt des Namens im Diagramm angezeigt wird. So wären der Name zur Identifikation des Stereotypen und der Marker, der im Diagramm angezeigt wird, getrennt. Als Marker könnten dann, so wie bisher die Namen in der Spezifikation, kurze und prägnante Bezeichner vergeben werden, und als Name des Stereotypen gegebenenfalls längere und eindeutige Identifikatoren.

3.1.2.7 Top-Level-Element

Ein Top-Level-Element in einem Modell ist ein Element, das in keinem Namensraum enthalten ist, da es sich auf der obersten Ebene des Modells befindet. In Werkzeugen ist das Top-Level-Element zumeist ein Exemplar der Metaklasse `Model`, weshalb dies auch in vielen Beispielen zu XMI so dargestellt wird (z. B. [Obj01d], S. 5-23). Durch die XMI-Spezifikation werden hierzu aber keine Festlegungen getroffen und auch mehrere Top-Level-Elemente erlaubt (siehe Beispiel [Obj00d], S. C-2). Da ein Profile kein Modell im engeren Sinne ist, soll auf ein Modell als Top-Level-Element verzichtet werden.

3.1.2.8 Inhalt eines Profile-Paketes

In der UML-Spezifikation ist festgelegt, dass ein Profile nicht beliebige Elemente enthalten darf, sondern nur UML-Erweiterungselemente und in deren Kontext benötigte Metaklassen. Eine genaue Definition dazu findet sich in *Well-formedness Rule* 2.14.3.4 ([Obj01d]), S. 2-195) zu Profiles:

> A profile package can only contain tag definitions, stereotypes, constraints and data types.
>
> self.contents→forAll(e |e.oclIsKindOf(Stereotype)or
> e.oclIsKindOf(Constraint) or
> e.oclIsKindOf(TagDefinition) or
> e.oclIsKindOf (DataType))

Diese Definition scheint nicht vollständig zu sein. Da eine Generalisierung von Stereotypen vorgesehen ist ([Obj01d], S. 2-78, S. 2-83), muss ein Profile Generalisierungen enthalten können. Weiterhin wird in [Obj01d] auf S. 2-199 beschrieben, dass ein Profile – analog zu anderen Paketen – auch Profiles beinhalten kann.

Die Metaklasse `DataType` spezialisiert `Classifier`, weshalb sie `Features` enthalten kann (Abbildung 3.2). Da Datentypen keine Identität haben, sind allerdings nur Operationen erlaubt, und zwar begrenzt auf Anfragen (*Queries*, siehe [Obj01d], S. 2-60). Andere Operationen sowie Methoden oder Attribute dürfen nicht in Datentypen enthalten sein. Weiterhin dürfen Datentypen auch keine anderen Modellelemente enthalten ([Obj01d], S. 2-60), d. h. es enfällt das Verhalten als Namensraum. In Abbildung 3.2 sind Metaklassen, deren Verwendung bei Datentypen ausgeschlossen ist, in grauer Farbe dargestellt.

Da anwenderdefinierte Datentypen in einem Profile nur als Typen von Tagged Values verwendet werden können – Datentypen zur Wiederverwendung in Modellen sind in Modellbibliotheken zu geben – wird ihre Definition zumeist einfach gehalten sein. Als primitive Datentypen sollten normalerweise die bereits vorhandenen UML-Datentypen ausreichen. Bei der Definition eigener primitiver Datentypen (`Primitive Type`) oder Programmiersprachen-abhängiger Datentypen (`ProgrammingLanguageDataType`) besteht die Schwierigkeit, dass deren Semantik nicht in UML ausgedrückt werden kann ([Obj01d], S. 2-48). Deshalb sollten in Profiles Aufzählungen (`Enumeration`) ausreichen, da für diese in einfacher und für den Anwender gut handhabbarer Weise durch die Angabe von Literalen (`EnumerationLiteral`) der Wertebereich definiert werden kann. Die Definition von Queries wird für Aufzählungen kaum notwendig sein. Die Mindestanforderung an zusätzlich benötigten Metaklassen für die Definition von Datentypen besteht daher nur aus der Metaklasse `EnumerationLiteral`.

Insgesamt sind also als Bestandteil eines Profiles zusätzlich zu Stereotypen, Tag Definitions, Constraints und Datentypen mindestens auch Generalisierungen, Profiles (d. h. Pakete mit dem Stereotypen `profile`) und Aufzählungs-Literale notwendig.

3.1.2.9 Icons von Stereotypen

Bei Stereotypen besteht die Möglichkeit, ein kleines graphisches Symbol (*Icon*) anzugeben, um Modellelemente, die mit dem Stereotypen versehen sind, zu kennzeichnen. Dazu enthält die Metaklasse `Stereotype` ein Attribut `icon` vom Typ `Geometry`. Der Typ `Geometry` wird jedoch nicht weiter spezifiziert. In der XMI-Datei kann hier eine beliebige externe Ressource angegeben werden, z. B. eine Graphik-Datei. Um auch die Icons durch XMI-Dateien austauschbar zu machen, müsste ein möglichst weit verbreitetes, standardisiertes und plattformunabhängiges Graphikformat anstelle von `Geometry` vereinbart werden.

Im Augenblick verwenden Werkzeuge unterschiedliche Formate zur Angabe der Icons, wobei meist für einen Stereotypen mehrere Icons in unterschiedlicher und festgelegter Größe anzugeben sind. Tabelle 3.1 zeigt die Formate und Größen für Icons

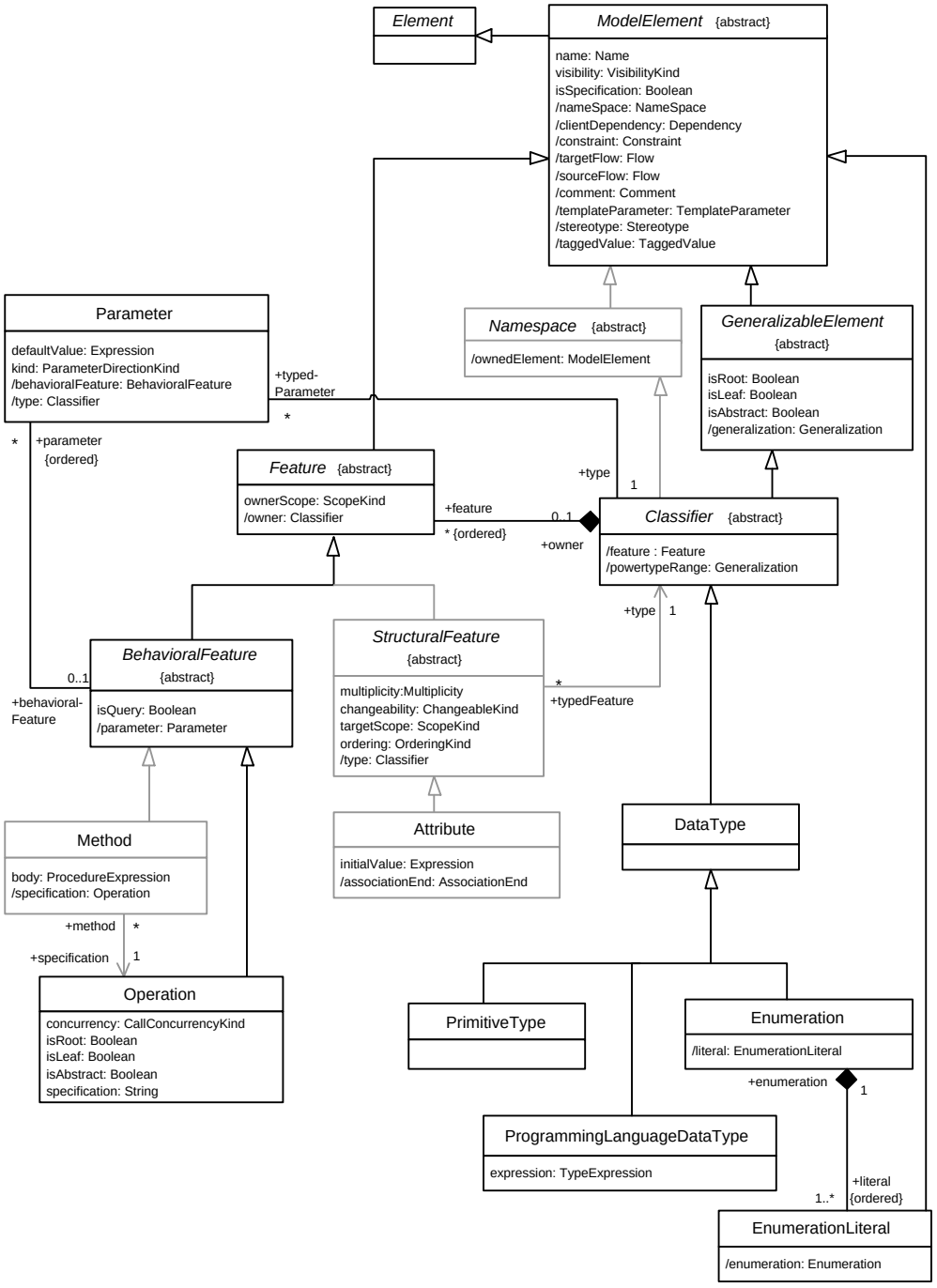

Abbildung 3.2: Datentypen und Features im physischen Metamodell. Alle Elemente stammen aus dem Paket `Core`. Elemente, die für Datentypen durch *Well-formedness Rules* ausgeschlossen sind, werden in grauer Farbe dargestellt.

in den Werkzeugen *Rose* von *Rational* [RAT02a] und *Objecteering* von *Objecteering Software* [OBJ02a]. Bei der Verwendung von Icons in Diagrammen wird die graphische Darstellung einer Metaklasse, die mit einem Stereotypen versehen ist, entweder vollständig durch ein Icon ersetzt (*kollabierte* Darstellung) oder es wird ihr nur ein kleines Icon hinzugefügt ([Obj01d], S. 3-32). In *Rational Rose* kann die Werkzeugleiste, die zur Erstellung von Modellelementen dient, auch um Stereotyp-Icons erweitert werden, wobei für die Werkzeugleiste grundsätzlich gewählt werden kann zwischen der Darstellung mit kleinen oder mit großen Icons. In der XMI-Datei sollten für das Attri-

Verwendungszweck des Icons		Rational Rose	Objecteering
Diagramm	kollabiert	Windows Meta File oder Enhanced Meta File	Windows Bitmap oder GIF
	nicht kollabiert		Windows Bitmap oder GIF
Werkzeugleiste	große Icons	Windows Bitmap (24 x 24)	-
	kleine Icons	Windows Bitmap (15 x 16)	-
Explorer-Fenster		Windows Bitmap (16 x 16)	Windows Bitmap oder GIF

Tabelle 3.1: Vergleich der verschiedenen Graphikformate und zur Angabe eines Stereotyp-Icons in den Werkzeugen „Rational Rose“ und „Objecteering“.

but `icon` die Bild-Dateien angegeben werden, die das Icon enthalten. Gegebenenfalls können auch mehrere Dateien angegeben werden.

Theoretisch könnten die verschiedenen vorhandenen Formate der verschiedenen Werkzeughersteller ignoriert und Icons nur in einem Format und einer Größe angegeben werden, indem bei Gebrauch entsprechende Transformationen vorgenommen werden. Um Qualitätsverluste bei der Transformation zu vermeiden, wäre dazu die Verwendung eines Vektor-Graphikformates notwendig. Ein geeignetes Format, besonders auch für die Verwendung in Zusammenhang mit XML, wäre *Scalable Vector Graphics* (SVG) [Wor01b]. Es wurde vom *World Wide Web Consortium* (W3C) [W3C02] standardisiert.

3.2 Beispiel: Das Variabilitäts-Profile

In diesem Abschnitt werden die Überlegungen des vorangegangenen Abschnitts beispielhaft für das UML-Profile für Variabilität umgesetzt. Durch das Heranziehen eines konkreten Beispiels werden – zusätzlich zu den theoretischen Betrachtungen aus dem vorangegangenen Abschnitt – weitere Probleme und Anforderungen aufgedeckt. Dabei ist festzustellen, dass die anhand des Beispiels auftretenden Probleme nicht domänenspezifisch sind, sondern allgemeiner Natur und damit auch auf andere Profiles übertragbar. In diesem Abschnitt werden die praktischen Probleme, die durch das Variabilitäts-Profile aufgezeigt werden, dargestellt und Lösungsansätze diskutiert. Als Resultat entsteht eine XMI-Repräsentation des UML-Profiles für Variabilität.

3.2.1 Probleme und Lösungsansätze

3.2.1.1 Name von Constraints

Bei Constraints wird meist auf die Angabe eines Namens verzichtet, so auch im Profile für Variabilität. Da die Metaklasse `Constraint` die Klasse `ModelElement` spe-

zialisiert, fordert das Metamodell für Constraints einen Namen. In den meisten Fällen werden künstliche Namen ausreichend sein.

3.2.1.2 Wiederverwendbare Constraints für die Modellebene

In UML kann zwischen zwei Arten von Constraints unterschieden werden: Constraints, die Teil eines Stereotypen sind, und Constraints, die in Modellen einem Modellelement zugeordnet sind (siehe [Obj01d], S. 2-76, Abb. 2-10). Erstere beziehen sich auf die Metamodellebene (M2) und befinden sich in Profiles, da sie zur Erweiterung des Metamodells dienen. Die andere Sorte von Constraints bezieht sich auf Modellebene (M1) auf die Modellelemente, denen sie zugeordnet ist.

Das Profile für Variabilität enthält eine Constraint `xor`, die für die Anwendung auf die Metaklasse `Relationship` (Beziehung) definiert ist ([Cla01], S. 50). Dies ist eine Constraint der zweiten Sorte, da sie in Modellen Modellelementen (hier eingeschränkt auf Beziehungen) zugeordnet werden soll. Es handelt sich also um ein wiederverwendbares Modellelement. Daher ist diese Constraint in eine Modellbibliothek aufzunehmen.

Unter den Standard-Elementen in der UML befinden sich ebenfalls solche Constraints, z. B. die Constraint `xor`, die sich zu der aus dem Variabilitäts-Profile nur dadurch unterscheidet, dass sie nur auf Assoziationen (statt auf Beziehungen im Allgemeinen) angewendet werden soll. Sollen die Standard-Elemente zur Verwendung in XMI als Profile dargestellt werden (siehe Abschnitt 3.1.2.5), müssten dementsprechend auch dort die Constraints in eine Modellbibliothek gegeben werden.

Alle wiederverwendbaren Constraints für die Modellebene haben gemeinsam, dass für jede eine Metaklasse angegeben ist, auf deren Instanzen sie angewendet werden dürfen, ähnlich dem Attribut `baseClass` bei Stereotypen. Die Semantik der Constraint ist dabei eng mit der jeweiligen Metaklasse verknüpft. Die UML-Spezifikation sieht aber keine Möglichkeit zur Angabe einer Metaklasse für Constraints vor.

Eine sehr einfache Lösung diese Problems wäre, auf die Angabe einer Metaklasse oder sogar der gesamten Constraint zu verzichten, da in der aktuellen UML-Spezifikation keine Unterstützung dafür vorgesehen ist. Der Anwender des Profiles könnte bei Bedarf die entsprechende Constraint im jeweiligen Werkzeug selbst angeben.

Ansonsten ist zur Lösung dieses Problems zu überlegen, ob eine solche Constraint nicht durch einen Stereotyp ersetzt werden kann. Der Stereotyp erhält als Namen den Namen der ursprünglichen Constraint und als Base-Class die Metaklasse, deren Instanzen die Constraint zugeordnet werden soll. In vielen Fällen wird die Ersetzung durch einen Stereotyp jedoch keine geeignete Lösung darstellen. So wird die Constraint `xor` aus dem Variabilitäts-Profile auf mehrere Beziehungen gleichzeitig angewandt, um darzustellen, welche Beziehungen sich gegenseitig ausschließen. Es können daher in einem Modell mehrere xor-Constraints vorhanden sein, die jeweils mehrere zusammengehörige Beziehungen referenzieren. Mit einem Stereotyp wäre das nicht möglich; ein Stereotyp kann zwar auch auf beliebig viele Modellelemente angewendet werden, jedoch würde dabei nicht erkennbar, welche Beziehung an welchem gegenseitigen Ausschluss beteiligt ist.

Alternativ sind Lösungsmöglichkeiten zu suchen, um für eine wiederverwendbare Constraint angeben zu können, zu welcher Metaklasse sie zugeordnet werden darf. Eine Möglichkeit wäre, der wiederverwendbaren Constraint eine Constraint zuzuordnen, die die Einschränkung auf eine bestimmte Metaklasse enthält. Das heißt, die wiederverwendbare Constraint wird als beliebiges Modellelement betrachtet, das durch

eine Constraint eingeschränkt wird. Im Falle der xor-Constraint aus dem Variabilitäts-Profile würde die Einschränkung lauten:

> Die Constraint wird nur Instanzen der Metaklasse `Relationship` zugeordnet.

bzw. in OCL:

```
self.constrainedElement ->
forAll(oclIsKindOf(Relationship))
```

Vorteilhaft an dieser Lösung ist ihre Einfachheit, sowohl bezüglich der Definiton als auch bezüglich der Verständlichkeit. Nachteilig ist die Vermischung von Metaebenen. Die wiederverwendbare Constraint ist ein Element auf Modellebene, und damit solten sich auch ihr zugeordnete Constraints auf die Modellebene beziehen. Die Einschränkung der Zuordnung auf die Metaklasse `Relationship` jedoch ist eine Einschränkung auf Metaebene. Die Vermischung dieser Ebenen ist zu vermeiden.

Eine weitere Lösung wäre die Definition eines Stereotypen für diese Art von Constraints, z. B. mit dem Namen `standardConstraint`. Dieser könnte als Tagged Value ein Attribut `applicableClass:Name` einführen, das den Namen der Metaklasse, deren Exemplaren die Constraint zugeordnet werden kann, enthält, analog dem Attribut `baseClass:Name` der Metaklasse `Stereotype`. Ein weiterer Tagged Value, `description`, kann zur Beschreibung der Constraint dienen; im Fall der xor-Constraint z. B. „Wechselseitiger Ausschluss der beteiligten Beziehungen“. Um sicher zu stellen, dass die Constraint nur auf Modellelemente angewendet wird, die Instanzen der im Attribut `applicableClass` bezeichneten Metaklasse sind, wird dem Stereotyp eine entsprechende Constraint angefügt. Abbildung 3.3 zeigt die Definition eines Stereotypen `standardConstraint` und die Anwendung auf die xor-Constraint. Der Stereotyp befindet sich dabei in einem Profile `StandardElements`, die Constraint in einer Modellbibliothek, die dem Variabilitäts-Profile zugeordnet ist.

Problematisch bleibt bei allen genannten Lösungen, wie die Constraint, wenn sie sich in einer Modellbibliothek befindet, in einem Modell angewendet werden soll. Zum einen verlangt das Metamodell zwischen einem Modellelement und einer zugehörigen Constraint eine beidseitige Referenz (siehe Abbildung 3.1), d. h. die Constraint müsste jedes Modellelelement, dem sie zugewiesen ist, referenzieren. Dies ist nicht möglich, da sie unabhängig und wiederverwendbar bleiben soll (siehe Abschnitt 3.1.2.3). Weiterhin ist notwendig, in einem Modell mehrere voneinander unabhängige gegenseitige Ausschlüsse von Beziehungen darstellen zu können, wodurch mehrere xor-Constraints in einem Modell benötigt werden. Demzufolge muss die Constraint aus der Modellbibliothek in Modelle „kopiert“ werden können. Dabei muss berücksichtigt werden, dass zwischen einer Constraint auf der Modellebene und einem Modellelement, dem sie zugeordnet ist, im Metamodell keine Komposition besteht (siehe Abbildung 3.1). Demzufolge können, wie auch bei Stereotypen, Namenskonflikte auftreten (siehe Abschnitt 3.1.2.6). Aus diesem Grund muss für jede Kopie der Constraint ein anderer Name vergeben werden, nur das Attribut `body` mit dem Wert `xor` bleibt bei jeder Kopie gleich.

Zusammenfassend muss gesagt werden, dass eine wiederverwendbare Constraint zwar unter Angabe einer Metaklasse, für die sie vorgesehen ist, in eine Modellbibliothek gegeben werden kann, dort aber nur als Vorlage dienen kann. Im weiteren Verlauf der Arbeit soll dazu die Lösung der Definition eines Stereotypen `standardConstraint` verwendet werden.

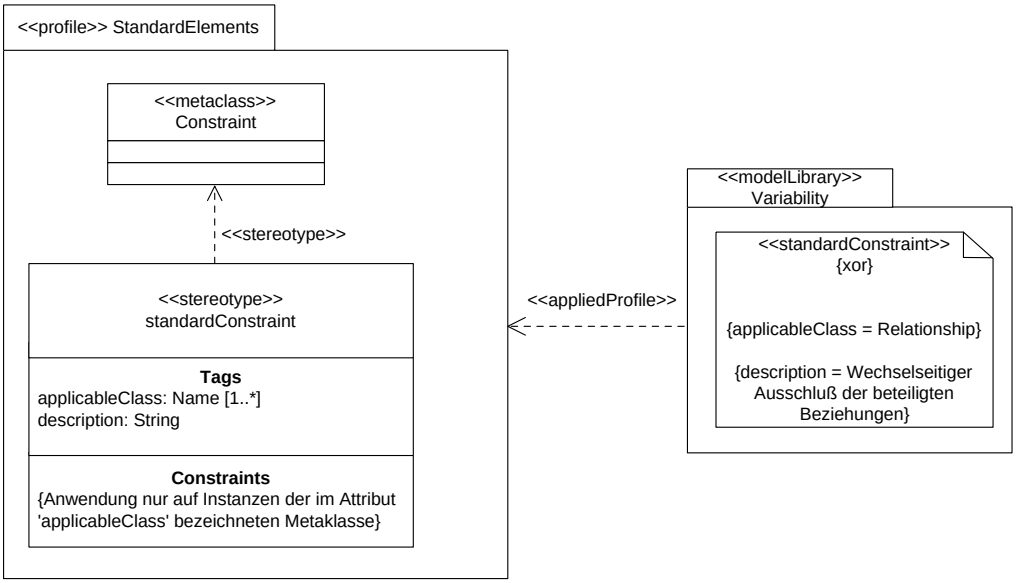

Abbildung 3.3: Definition und Verwendung eines Stereotypen 'standardConstraint' zur Definition einer wiederverwendbaren Constraint für die Modellebene. Die Constraint befindet sich in einer Modellbibliothek, der Stereotyp in einem Profile.

3.2.1.3 „Let"-Ausdrücke und Constraints

Durch das Schlüsselwort `let` können in OCL Teilausdrücke definiert werden (*Let-Ausdrücke*), die mehrfach verwendet werden sollen ([Obj01d], S. 6-8). Es erlaubt die Definition von OCL-Attributen und -Operationen, die wie Standard-OCL-Attribute bzw. -Operationen verwendet werden können.

Wird der Let-Ausdruck innerhalb einer Constraint definiert, ist er nur innerhalb dieser verfügbar. Zur Wiederverwendung in mehreren Constraints wird in der OCL ein Stereotyp `definition` eingeführt, mit dem Constraints versehen werden können. Diese „Definition-Constraints" dürfen nur einen Let-Ausdruck enthalten und werden Exemplaren der Metaklasse `Classifier` angefügt. Dadurch ist er verwendbar, wie jede andere Eigenschaft (Attribut oder Operation) des Classifiers auch.

Durch die Verwendung von Let-Ausdrücken kann der Umfang von Constraints stark reduziert werden. Constraints werden dadurch leichter verständlich und durch die Wiederverwendung von Let-Ausdrücken verringert sich die Fehlerwahrscheinlichkeit. Eine syntaktische oder semantische Notwendigkeit für die Verwendung von Let-Ausdrücken gibt es jedoch nicht, da sie jederzeit substituiert werden können.

Das Variabilitäts-Profile enthält bei der Definition der Constraints des Profiles auch Let-Ausdrücke ([Cla01], S. 50, S. 64). Ihr Verfügbarkeitsbereich ist das gesamte Profile (bzw. Teil-Profile, siehe „Untergliederung des Profiles"). Sie definieren Attribute oder Operationen für verschiedene Metaklassen, z. B. `ModelElement` oder `Relationship`; die jeweilige Metaklasse, d. h. der Kontext auf den sich der Let-Ausdruck bezieht, wird mit dem Schlüsselwort `context` spezifiziert. Es stellt sich die Frage, wie diese Let-Ausdrücke in das Profile Metamodell-konform einbezogen werden können.

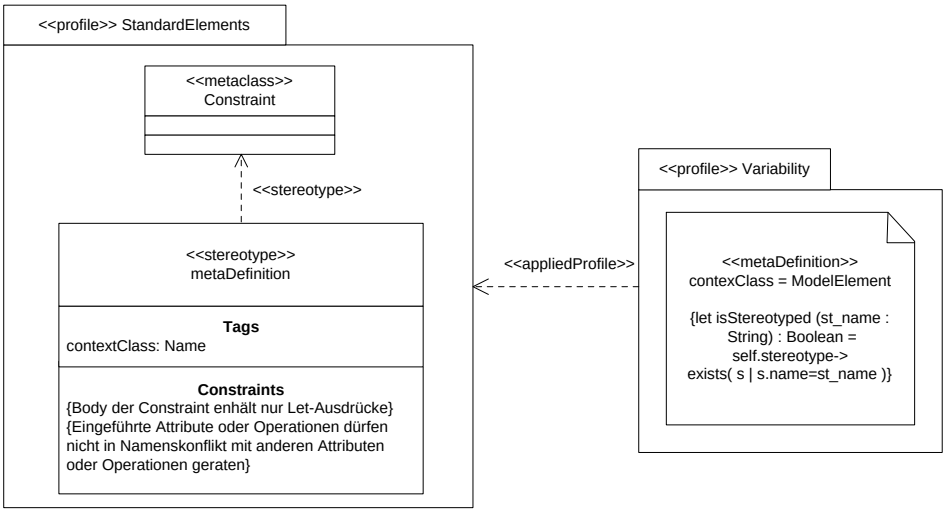

Abbildung 3.4: Definition und Verwendung eines Stereotypen 'metaDefinition' zur Verwendung von Let-Ausdrücken in Profiles. Als Anwendungsfall ist beispielhaft das Pseudo-Attribut 'isStereotyped' für die Metaklasse 'ModelElement' aus dem Variabilitäts-Profile dargestellt.

Definition-Constraints werden ausschliesslich an Classifier angefügt. Da die erlaubten Elemente eines Profiles weder ein Exemplar der Metaklasse `Classifier` sind noch Exemplare einer Unterklasse davon, sind Definition-Constraints nicht zur Verwendung in Profiles geeignet. Darüber hinaus sind Definition-Constraints den Elementen zuzuordnen, deren OCL-Eigenschaften erweitert werden sollen. Sollten sie analog in einem Profile angewendet werden, so wären sie Metaklassen zuzuordnen. Die UML-Spezifikation sieht jedoch ausschliesslich Stereotypen für eine Zuordnung direkt zu Metaklassen vor.

Die genannten Anforderungen zur Unterstützung von Let-Ausdrücken sind nur durch eine Erweiterung der Spezifikation zu erfüllen. Es könnte ein Stereotyp definiert werden, ähnlich dem Stereotypen `definiton`, z. B. mit dem Namen `metaDefinition`. Analog `definiton` muss er als Base-Class die Metaklasse `Constraint` haben, und darf nur Let-Ausdrücke enthalten. Anstatt Exemplaren der Metaklasse `Classifier` angefügt zu werden, referenziert er beliebige Metaklassen. Die UML-Spezifikation bietet keine direkte Möglichkeit zur Referenz auf Metaklassen. Analog zum Attribut `baseClass: Name` der Metaklasse `Stereotyp` könnte jedoch ein Tagged Value eingeführt werden, der den Namen der Metaklasse enthält, z. B. `contextClass: Name`. Alle in [Obj01d], S. 6-8 genannten Constraints für den Stereotypen `definiton` sollten entsprechend angepasst übernommen werden. Eine graphische Definition des Stereotypen`metaDefinition`ist in Abbildung 3.4 dargestellt.

Alternativ könnte auf Let-Ausdrücke verzichtet werden, da sie nicht zwingend notwendig sind und nicht bei der Anwendung des Profiles zur Modellierung benötigt werden. Für das Variabilitäts-Profile würde das bedeuten, dass die Let-Ausdrücke in den betreffenden Constraints substituiert werden müssen.

3.2.1.4 Untergliederung des Profiles

Das Profile für Variabilität wird in seiner Beschreibung in [Cla01] in drei Teile untergliedert: „Merkmalsmodellierung“ in Abschnitt 7.2, „Modellierung von Variationspunkten“ in Abschnitt 7.3 und „Optionale Modellelemente“ in Abschnitt 7.4., wobei die letzteren beiden Teile inhaltlich zusammenhängen. Daher stellt sich die Frage, ob eine solche Untergliederung auch bei der XMI-Repräsentation vorgenommen werden soll. Die UML-Spezifikation gibt zunächst vor, dass ein Profile nur Stereotypen, Constraints, Tag Definitions und Datentypen enthalten darf ([Obj01d], S. 2-195). Das Enthalten von anderen Profiles wird aber an anderer Stelle ausdrücklich erlaubt ([Obj01d], S. 1-199).

Eine Untergliederung des Profiles hätte den Vorteil, dass die logische Struktur erhalten bleibt und sich das Profile für den Anwender verständlicher darstellt. Ein sehr wichtiger Aspekt ist zusätzlich, dass das Profile in seinen verschiedenen Teilen Stereotypen gleichen Namens enthält: die Abschnitte zur Merkmalsmodellierung und zur Modellierung von Variationspunkten enthalten beide jeweils einen Stereotyp mit den Namen `constraint`, `requires` und `mutex`, die Abschnitte zur Merkmalsmodellierung und zu optionalen Modellelementen haben einen Stereotyp mit dem Namen `optional` gemeinsam. Durch die Untergliederung in mehrere Profile-Pakete befinden sich diese Stereotypen in verschiedenen Namensräumen, wodurch die Namenskonflikte beseitigt sind. Bei der Anwendung des gesamten Profiles müssen dann allerdings die Stereotypen anhand ihres vollständigen Pfadnamens unterschieden werden (zu den Namenskonflikten von Stereotypen siehe auch den Abschnitt 3.1.2.6).

In Absprache mit dem Entwickler des Variabilitäts-Profiles, M. Clauß, wird das Variabilitäts-Profile in zwei Unter-Profiles untergliedert. Falls notwendig – z. B. wenn eine innere Struktur von Profiles durch ein Werkzeug nicht unterstützt wird – können diese auch als zwei eigenständige Profiles betrachtet werden.

3.2.2 XMI-Repräsentation

Auf Basis der vorausgegangenen Untersuchungen wird das Profile für Variabilität in XMI formuliert. Dabei werden nur die obligatorischen Bestandteile des Variabilitäts-Profiles, die in [Cla01] in Tabellennotation aufgeführt sind, berücksichtigt. Folgende Lösungen werden gewählt:

- Profile und zugehörige Modellbibliothek befinden sich in separaten XMI-Dateien (siehe Abschnitte 3.1.2.1, bzw. 3.1.2.3),
- Abhängigkeiten befinden sich in der Datei, in der sich das Element mit der Client-Rolle der Abhängigkeit befindet,
- Standard-Elemente aus der Spezifikation werden in einer separaten XMI-Datei als ein Profile zusammengefasst (siehe Abschnitt 3.1.2.5), wobei nur die für das Variabilitäts-Profile benötigten Standard-Elemente berücksichtigt werden,
- Namenskonflikte zwischen Stereotypen in den Standard-Elementen (siehe Abschnitt 3.1.2.6) werden ignoriert,
- Top-Level-Elemente in den XMI-Dateien (siehe Abschnitt 3.1.2.7) sind Pakete, Abhängigkeiten zwischen Top-Level-Paketen und XMI-Referenzen auf Elemente aus anderen Paketen,

- Icons von Stereotypen (siehe Abschnitt 3.1.2.9) werden im Format für *Rational Rose* angegeben,
- für Constraints werden künstliche Namen vergeben (siehe Abschnitt 3.2.1.1),
- wiederverwendbare Constraints für die Modellebene (siehe Abschnitt 3.2.1.2) befinden sich als Vorlage in einer Modellbibliothek, wobei zur Angabe der Metaklasse, der sie zugeordnet werden sollen, eine Constraint verwendet wird,
- Let-Constraints (siehe Abschnitt 3.2.1.3) werden mit Hilfe eines zusätzlichen Stereotypen, `metaDefinition`, umgesetzt und
- das Profile wird in zwei Unter-Profiles geliedert (siehe Abschnitt 3.2.1.4).

Die resultierende Struktur ist in Abbildung 3.5 dargestellt. Die vollständigen XMI-Dokumente befinden sich auf der CD zu dieser Arbeit.

3.3 Bewertung

Bei der Bewertung der Verwendung von XMI zur Repräsentation von UML-Profiles sind zwei Aspekte zu berücksichtigen. Zum einen liefert die Formulierung eines UML-Profiles gemäß dem Metamodell in XMI Aufschluss darauf, inwieweit das Metamodell die Anforderungen von UML-Profiles abdeckt. Durch die Herannahme eines konkreten Profiles, dem Profile für Variabilität, offenbaren sich weitere Anforderungen an die Spezifikation eines Profile-Mechanismus. Der andere Aspekt ist die Bewertung der XMI-Repräsentation als Austauschformat für Werkzeuge. Hierbei ist zu untersuchen, inwieweit alle Informationen, die speziell für die Werkzeugunterstützung eines Profiles notwendig oder wünschenswert sind, einbezogen werden können.

Im folgenden soll daher in Abschnitt 3.3.1 festgehalten werden, welche zusätzlichen Anforderungen an den Profile-Mechanismus aus der vorangegangen Diskussion von Problemen resultieren. Abschnitt 3.3.2 bewertet bezüglich des Aspekts des Austauschs von Profiles zwischen Werkzeugen.

3.3.1 Resultierende Anforderungen an den UML-Profile-Mechanismus

Zunächst werden Anforderungen aufgeführt, die den Profile-Mechannismus im allgemeinen – unabhängig von einer konkreten UML-Version – betreffen. Anschliessend folgen Anforderungen bezüglich erforderlicher Änderungen vorhandener Definitionen aus der UML-Spezifikation 1.4.

3.3.1.1 Generelle Anforderungen

In [Ple01] wurden in Kapitel 3 bereits Anforderungen an UML-Profiles zusammengestellt. Die Untersuchung eines konkreten Beispiels, dem UML-Profile für Variabilität, zeigt das Bedürfnis nach zusätzlichen Mechanismen. Dort wurden Konstrukte verwendet, die allgemein in Profiles sinnvoll erscheinen, und die bisher noch nicht berücksichtigt wurden. Daher sind die optionalen Anforderungen an UML-Profiles um folgende Punkte zu erweitern:

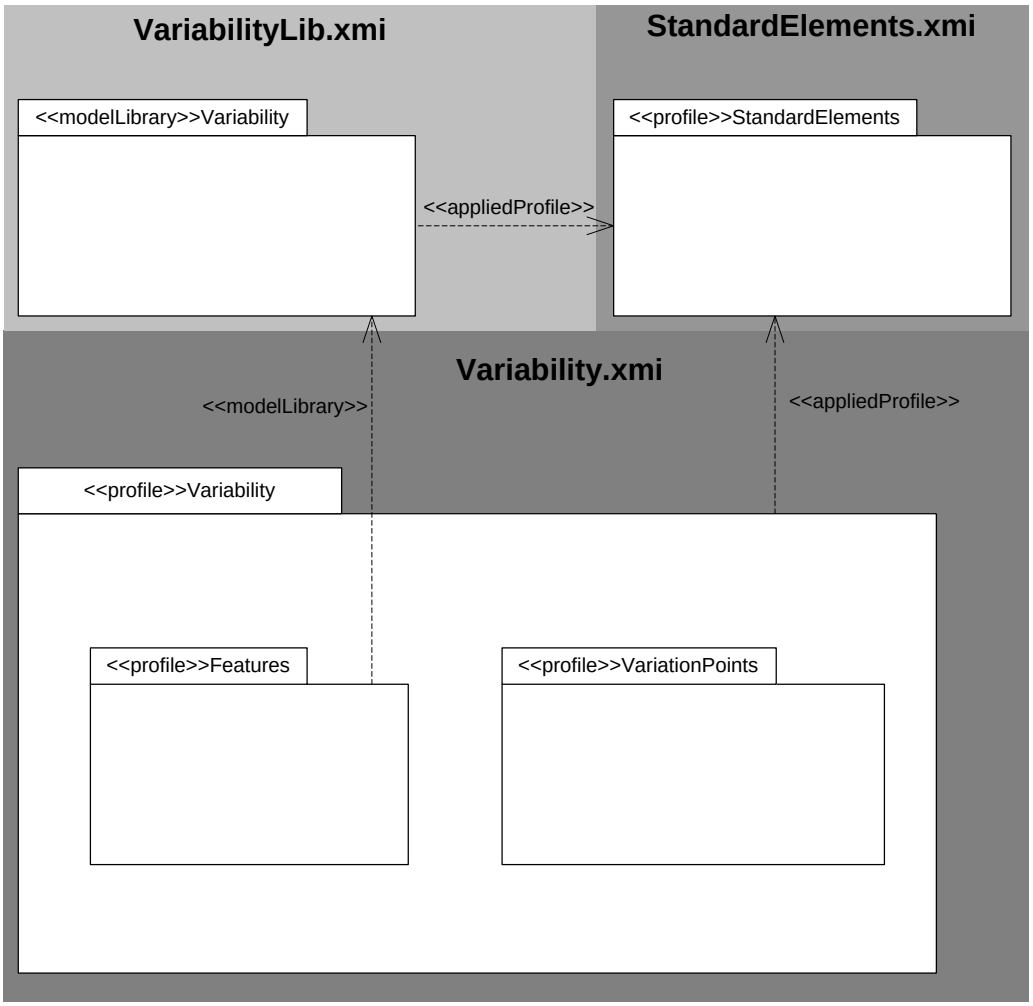

Abbildung 3.5: Struktur der XMI-Repräsentation des Variabilitäts-Profiles. Dargestellt sind die Pakete und die Abhängigkeiten zwischen ihnen sowie, farbig unterlegt, die Aufteilung in drei separate XMI-Dateien.

1. Let-Ausdrücke:
   Zur Vereinfachung und besseren Lesbarkeit von Constraints zur Spezialisierung des Metamodells sollte es möglich sein, innerhalb eines Profiles durch Let-Ausdrücke Pseudo-Attribute und -Operationen für Metaklassen zur Verwendung in OCL hinzuzufügen.

2. Wiederverwendbare Constraints für die Modellebene:
   Es sollte einen Mechanismus geben, der es ermöglicht, für einen bestimmten Anwendungsbereich wiederverwendbare Constraints zu definieren. Da diese Constraints für die Anwendung in Modellen vorgesehen sind, sollte es möglich sein, sie in Modellbibliotheken zu geben. Es soll für jede wiederverwendbare Constraint eine Metaklasse angegeben werden können, zu deren Instanzen die jeweilige Constraint zugeordnet werden kann.

3.3.1.2 Anforderungen an die UML-Spezifikation

In UML 1.4 wurde erstmals der Profile-Mechanismus integriert. Bei der Formulierung von UML-Profiles wurden Punkte gefunden, an denen die UML-Spezifikation Definitionen verbessern oder erweitern muss, um eine konfliktfreie Nutzung des in UML 1.4 vorgesehen Profile-Mechanismus zu gewährleisten:

1. Beseitigung von Namenskonflikten bei Standard-Elementen:
   Bisher enthält die Spezifikation bei den Standard-Elementen Stereotypen mit gleichem Namen (siehe Abschnitt 3.1.2.6). Da diese teilweise im gleichen Namensraum definiert sind, ergeben sich damit unerlaubte Namenskonflikte.

2. Definition der Semantik der „appliedProfile“-Abhängigkeit:
   Abhängigkeiten mit dem Stereotypen `appliedProfile` ermöglichen einem Modell Zugriff auf Elemente eines Profiles. Darüber hinaus ist die Semantik dieser Abhängigkeit nicht definiert (siehe Abschnitt 3.1.2.2). Um eine einheitliche Vorgehensweise zu gewährleisten, sollte definiert werden, wie dieser Zugriff erfolgt.

3. Definition der Semantik der „modelLibrary“-Abhängigkeit:
   Abhängigkeiten mit dem Stereotypen `modelLibrary` besteht zwischen einem Profile und einer zugehörigen Modellbibliothek. Darüber hinaus ist die Semantik dieser Abhängigkeit nicht definiert (siehe Abschnitt 3.1.2.4). Um eine einheitliche Vorgehensweise zu gewährleisten, sollte definiert werden, auf welche Weise ein Modell, das ein Profile verwendet, auf Elemente einer zugehörigen Modellbibliothek zugreifen kann.

4. Eindeutige Definition der erlaubten Bestandteile von Profiles:
   In der UML-Spezifikation ist widersprüchlich festgelegt, welche Metaklassen ein Profile enthalten darf (siehe Abschnitt 3.1.2.8). Hierzu sollte eine eindeutige Definition gegeben werden.

3.3.2 Informationsgehalt der XMI-Repräsentation

Zunächst wird der Austausch von Informationen, die konform zur UML-Spezifikation sind, bewertet und offene Anforderungen aufgeführt. Anschließend wird betrachtet, inwieweit das XMI-Format auch zum Austausch zusätzlicher Information geeignet ist.

3.3.2.1 Anforderungen an das XMI-Austauschformat

Für den Austausch mittels XMI von UML-Profiles, die gemäß der UML-Spezifikation 1.4 definiert sind, konnten folgende Notwendigkeiten erarbeitet werden:

1. Bereitstellen der Standard-Elemente für XMI:
   Die Standard-Elemente aus der UML-Spezifikation wurden in XMI bisher nicht berücksichtigt. Um Wiederverwendung zu ermöglichen, und zur Unterscheidung von anwenderdefinierten Erweiterungselementen, sollten sie in einem Standard-Dokument für XMI zur Verfügung gestellt werden.

2. Format für Icons von Stereotypen:
   Für eine graphische Repräsentation von Stereotypen durch Icons wird in der UML-Spezifikation kein Format definiert. Für einen Austausch zwischen Werkzeugen wäre ein Standard-Format notwendig.

Insgesamt können alle Teile eines UML-Profiles, die konform zu UML 1.4 sind, vollständig in XMI repräsentiert werden, wie am Beispiel des UML-Profile für Variabilität gezeigt wurde. Das XMI-Format ist deshalb grundsätzlich gut für den Austausch von Profiles geeignet.

3.3.2.2 Zusätzliche Informationen für Werkzeuge

Neben den Bestandteilen von Profiles aus der UML-Spezifikation ist denkbar, dass im Rahmen einer Werkzeugunterstützung zusätzliche Informationen zu einem Profile gespeichert werden sollen. Hier kann unterschieden werden zwischen

- Informationen für zusätzliche Funktionalität eines Profiles und
- allgemeinen zusätzlichen Informationen für Werkzeuge.

Der erste Fall tritt auf, wenn ein Profile Semantik für Funktionalität über die UML-Spezifikation hinaus beinhalten soll, beispielsweise Anpassung der Codegenerierung an einen bestimmten Anwendungsbereich. Ein Überblick über solche zusätzliche Funktionalität findet sich in Abschnitt 4.3.

Der zweite Fall steht nicht in direktem Zusammenhang mit Profiles. Es kann generell bei Werkzeugen das Bedürfnis auftreten, zusätzliche Informationen zu speichern, z. B. Verwaltungsinformation oder Informationen zur Darstellung. Im Fall von Profiles beispielsweise könnte ein Werkzeug, das die graphische Definition von Profiles in einem Diagramm unterstützt, das vom Anwender erstellte Layout der Diagramme persistent halten wollen.

Aus beiden Fällen folgt als Konsequenz, dass für ein Austauschformat eine Erweiterbarkeit um zusätzliche Informationen wünschenswert ist. Auch wenn bestimmte zusätzliche Informationen zunächst nur von einem einzigen Werkzeug behandelt werden können, kann ihre Einbeziehung in ein Austauschformat dennoch notwendig sein, da ein Dokument an verschiedenen Stellen innerhalb eines Softwareentwicklungsprozesses vom gleichen Werkzeug bearbeitet werden kann. Ein Beispiel ist die Erstellung eines UML-Modells in einem Werkzeug. Aus dem Modell wird Code erzeugt, der mittels eines Austauschformats in eine Entwicklungsumgebung exportiert wird, um den Quellcode zu bearbeiten. Um das Modell mit dem veränderten Quellcode konsistent zu halten, wird der Code zu einem späteren Zeitpunkt wieder in das Modellierungswerkzeug reimportiert (*Round-Trip-Engineering*). Unterstützt das Austauschformat die Einbeziehung werkzeugspezifischer Informationen, so kann das Modellierungswerkzeug

beim Reimport auf diese zurückgreifen. Das Beispiel zeigt, dass die Unterstützung zusätzlicher Informationen in Austauschformaten in jedem Fall notwendig ist, auch dann, wenn diese werkzeugspezifisch sind.

In XMI wird diese Forderung nach einer Möglichkeit zur Erweiterung um Zusatzinformationen berücksichtigt. Es stellt dazu die XMI-Elemente `XMI.extensions` und `XMI.extension` zur Verfügung. Beide Elemente enthalten ein Attribut, das vorgesehen ist zur Identifikation des Werkzeuges, das die Erweiterung vorgenommen hat. Durch `XMI.extensions` wird ein separater Bereich in einer XMI-Datei gekennzeichnet, der beliebige zusätzliche Information enthalten kann. Das Element `XMI.extension` wird dagegen Elementen in der XMI-Datei direkt zugeordnet. Alle XMI-konformen DTDs erlauben, jedes beliebige Element – im Fall von UML also alle Modellelemente – mit einer `XMI.extension` zu versehen. Eine `XMI.extension` enthält als Attribut zusätzlich einen Identifikator für das erweiterte Element. Weiterhin kann es beliebige Informationen beinhalten, z. B. auch Referenzen auf Informationen in `XMI.extensions`.

In der Praxis wird nicht immer der vorgesehene XMI-Erweiterungsmechanismus genutzt, sondern es werden stattdessen auch erweiterte DTDs verwendet. Ein Beispiel dafür ist die XMI-Unterstützung für das UML-Werkzeug *Rational Rose*, die von der Firma *Unisys* [UNI02a] entwickelt wurde (siehe Abschnitt 5.1.3.1). Dort kann zum Export und Import von Modellen mittels XMI alternativ entweder die standardisierte UML-DTD verwendet werden oder eine proprietäre DTD. Die proprietäre DTD dient der Einbeziehung zusätzlicher Informationen bezüglich der Darstellung von Diagrammen. Sie bildet ein dafür von *Unisys* entwickeltes Metamodell ab. Die proprietären Metaklassen, die die Diagramminformation enthalten, sind an das UML-Metamodell unter Verwendung der UML-Metaklasse `PresentationElement` ([Obj01d], S. 2-17, S. 2-48) angebunden. Diese Metaklasse ist gemäß UML-Spezifikation speziell für diesen Zweck vorgesehen. Abbildung 3.6 zeigt das Prinzip der Erweiterung des UML-Metamodells durch *Unisys*, so wie es aus der erweiterten DTD hervorgeht. Die proprietäre Erweiterung besteht aus einem Paket `Diagram`, den darin enthaltenen Metaklassen sowie Beziehungen zur Anbindung an UML-Standard-Metaklassen aus dem UML-Paket `Core`. Zur Unterscheidung von den proprietären Elementen sind in der Abbildung die Elemente aus dem Standard-UML-Metamodell in grauer Farbe dargestellt. Die Abbildung zeigt, dass jedes Modellelement einem Diagramm zugeordnet ist. Innerhalb eines Diagramms besteht für jedes Modellelement eine graphische Repräsentation `PresentationElement`, das als Attribute einen Darstellungs-Stil und die Koordinaten des Elementes im Diagramm enthält.

Die Einführung einer DTD zur Einbindung werkzeugspezifischer Informationen bringt den Vorteil, dass die zusätzlichen Informationen mittels der DTD verifiziert werden können. Nachteilig ist jedoch, dass damit erstellte Dokumente nicht mit der Standard-DTD konform sind. Folglich sollte, um eine allgemeine Austauschbarkeit zu gewährleisten, besser der in der XMI vorgesehene Erweiterungsmechanismus (`XMI.extensions` und `XMI.extension`) verwendet werden. Dieser bietet die Möglichkeit, Erweiterungen sowohl einem beliebigen XMI-Element als auch dem gesamten XMI-Dokument hinzuzufügen und diese mit einem Identifikator des verantwortlichen Werkzeugs zu versehen. Daher sollte der eingebaute XMI-Erweiterungsmechanismus für jede Art von zusätzlicher Information ausreichend sein.

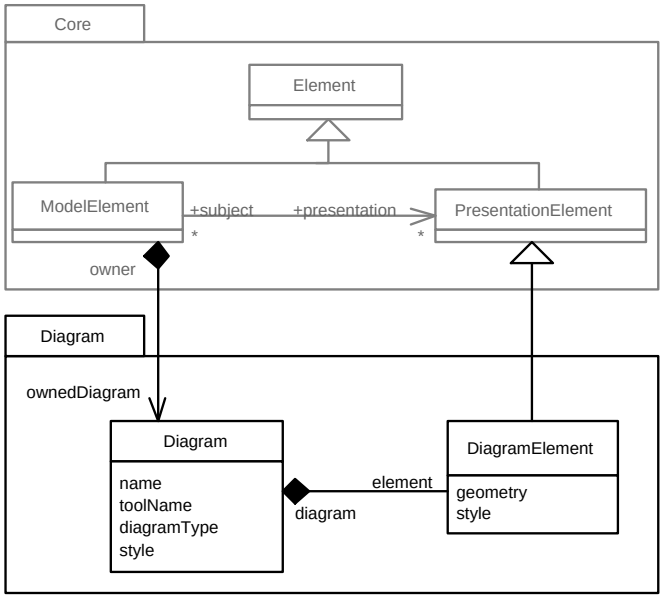

Abbildung 3.6: Beispiel für eine Erweiterung der UML-DTD: das erweiterte Metamodell von Unisys für eine DTD zur Einbeziehung von Diagramm-Informationen in XMI (Prinzipdarstellung). Der Ausschnitt aus dem Standard-UML-Metamodell ist in grauer Farbe dargestellt, die proprietäre Erweiterung in schwarzer Farbe.

Kapitel 4

Generelle Konzepte zur Werkzeugunterstützung von UML-Profiles

In diesem Abschnitt werden unabhängig von einem konkreten Werkzeug die grundlegenden Konzepte einer Werkzeugunterstützung für UML-Profiles erarbeitet. Dazu werden zunächst Anforderungen aufgestellt. Die semantische Auswertung von Constraints sowie Semantik von Profiles über den Rahmen der UML-Spezifikation hinaus sollen in dieser Arbeit nicht tiefergehend betrachtet werden und werden deshalb bei den Anforderungen nicht berücksichtigt.

Anschließend werden die grundlegenden Aufgaben einer Integration von Profile-Unterstützung in bestehende Werkzeuge beschrieben.

Abschließend erfolgt ein Überblick über eine erweiterte Verwendung von Profiles über den Rahmen der UML-Spezifikation hinaus.

4.1 Anforderungen

Beim Umgang mit Profiles ergeben sich zwei getrennte Teilaufgaben, erstens die (einmalige) Erstellung eines Profiles und zweitens die (beliebig häufige) Anwendung des Profiles bei der Modellierung. Die Anforderungen für Werkzeugunterstützung von UML-Profiles werden im folgenden Abschnitt nach diesen beiden Teilaufgaben gegliedert.

Ausgangspunkt für eine weitere Strukturierung der Anforderungen soll die Grundstruktur von Softwareentwicklungs-Werkzeugen nach [Bal98], S. 604, sein. Als grundlegende Bestandteile eines Werkzeuges ergeben sich daraus Benutzeroberfläche, Funktionalität und Persistenz. Die Funktionalität soll in Anlehnung an [Lis00] in drei Bereiche weiter untergliedert werden: Modellnotation, Werkzeug-Prüfung und -Unterstützung und Vorgehensweise und Methodik.

Unter Modellnotation ist hier die Erstellung und Bearbeitung von Dokumenten zu verstehen, die das jeweilige Modell enthalten. Dazu ist zu betrachten, welche Modelle bzw. Modellelemente erstellt und bearbeitet werden müssen, unabhängig von einer konkreten Darstellungsweise oder Benutzeroberfläche. Werkzeug-Prüfung und -Unterstützung bezeichnet die Funktionalität des Werkzeuges, um den Anwender bei

der Erstellung des Modells zusätzlich zu unterstützen und das Modell zu validieren. Weiterhin kann ein Werkzeug für die Modellnotation Vorgehensweisen und Methoden unterstützen.

Insgesamt werden die Anforderungen nach folgenden Bereichen gegliedert:

1. Modellnotation
2. Werkzeug-Prüfung und -Unterstützung
3. Vorgehensweise und Methodik
4. Benutzeroberfläche
5. Persistenz

Dabei sollen in dieser Arbeit nur Anforderungen betrachtet werden, die speziell die Unterstützung von UML-Profiles betreffen. Allgemeine Anforderungen an UML-Werkzeuge bzw. Werkzeuge generell werden vorausgesetzt und nicht explizit aufgeführt.

Alle Anforderungen werden zunächst unabhängig von einer konkreten UML-Version formuliert, so dass sie, sofern möglich, auch für zukünftige Versionen gültig sind. Alle Anforderungen sind kursiv geschrieben, numeriert und zur deutlichen Unterscheidung von Überschriften mit einem vorangestellten „A“ markiert. Allen Anforderungen, die nicht selbsterklärend sind, folgt ein konkretes Beispiel aus der aktuellen Version von UML, UML 1.4.

Es wird unterschieden zwischen optionalen Anforderungen und obligatorischen Anforderungen. Optionale Anforderungen sind explizit als solche gekennzeichnet, andernfalls ist die Anforderung obligatorisch.

4.1.1 Erstellung von Profiles

Bei der Anpassung der UML für einen bestimmten Anwendungsbereich durch UML-Profiles sind zwei Mechanismen zu unterscheiden: erstens die Erweiterung durch die eingebauten UML-Erweiterungsmechanismen in einem Profile-Paket und zweitens die Bereitstellung wiederverwendbarer Modellelemente für den Anwendungsbereich. Für letzteres sind Modellbibliotheken vorgesehen, die beliebige wiederverwendbare Elemente enthalten dürfen, hauptsächlich Klassen und Datentypen. Ihr Bezug zu UML-Profiles besteht allein darin, dass sie von einer beliebigen Anzahl von Profiles referenziert werden können. Zu ihrer Erstellung wird daher der Profile-Mechanismus nicht benötigt; der Unterschied zur Erstellung eines „normalen“ UML-Modelles besteht nur in der Beschränkung auf wiederverwendbare Modellelemente sowie in der Kennzeichnung durch den Stereotyp `modelLibrary`. Da in diesem Kapitel nur die Funktionalität untersucht werden soll, die spezifisch für UML-Profiles ist und über die Grundfunktionalität von UML-Werkzeugen hinaus geht, sollen Modellbibliotheken im Folgenden nur als Ganzes betrachtet werden. Auf ihren Inhalt und inneren Aufbau wird nicht weiter eingegangen.

4.1.1.1 Modellnotation

Zur Erstellung eines Profiles wird die Untermenge des UML-Metamodells benötigt, die zur Beschreibung eines Profiles notwendig ist. Diese enthält Metaklassen

- für die einzelnen Elemente, die Inhalt eines Profiles sind,

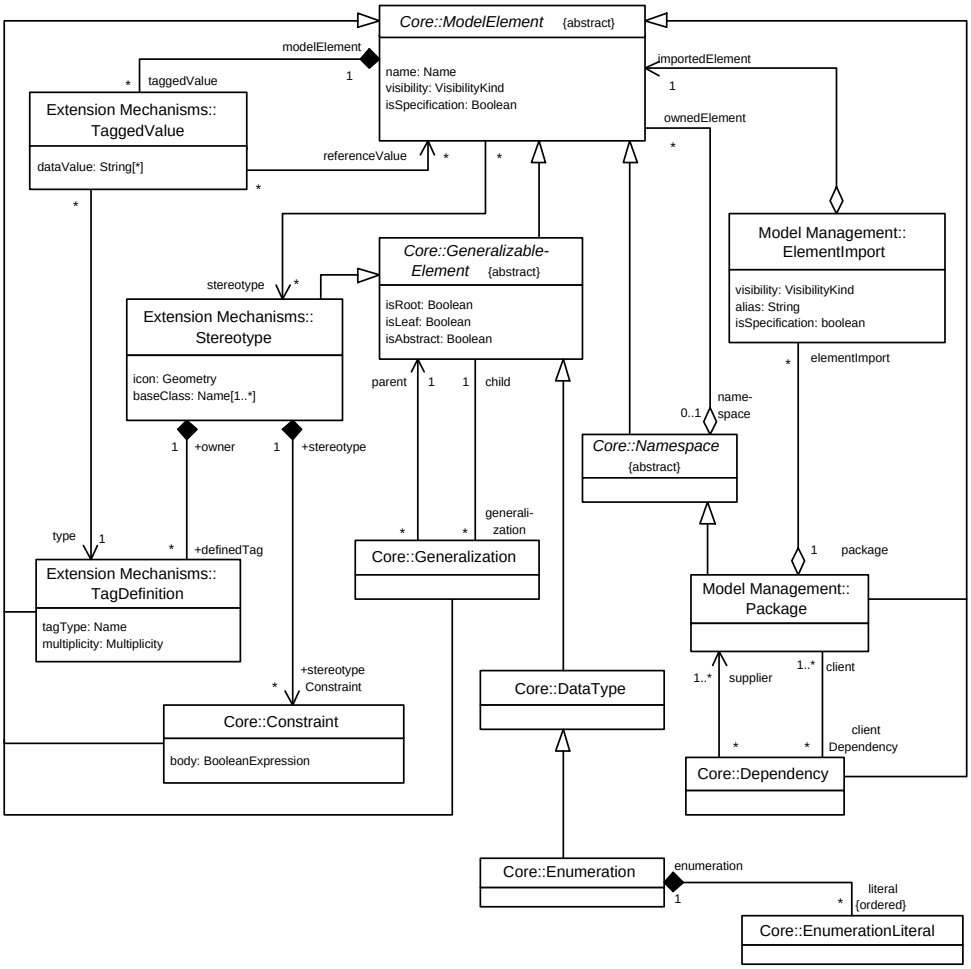

Abbildung 4.1: Die Untermenge aus dem UML-Metamodell zur Erstellung von UML-Profiles.

- als Behälter für die einzelnen Elemente und
- für Beziehungen zwischen einzelnen Elementen und zwischen Behältern.

Zusätzlich müssen zu jeder Metaklasse etwaige Eltern-Metaklassen vorhanden sein, von denen Attribute oder Referenzen geerbt werden. Abbildung 4.1 zeigt diese Untermenge für UML 1.4. Die Abbildung basiert auf dem Austausch-Metamodell (siehe 3.1.1), um den Austausch von Profiles zu ermöglichen. Zur besseren Einordnung für den Leser ist für jede Metaklasse der Name des Pakets aus dem logischen Metamodell angegeben.

Die Untermenge enthält Metaklassen zur Repräsentation des Inhalts von Profiles gemäß Abschnitt 3.1.2.8. Dabei wurde auf primitive und Programmiersprachenabhängige Datentypen sowie auf die Angabe von Operationen für Datentypen verzichtet. Als Behälter für Profiles und Modellbibliotheken werden Pakete (`Package`) benötigt, für Beziehungen zwischen diesen werden Abhängigkeiten (`Dependency`) und die Metaklasse `ElementImport` benötigt. Zusätzlich muss auch bei der Erstellung von Profiles die Anwendung von Erweiterungselementen möglich sein, da z. B. ein Profile mit dem Stereotyp `profile` und einem Tagged Value `applicableSubset` versehen werden muss, weshalb die Metaklasse `TaggedValue` und für `ModelElement` die Referenz `stereotype` in die Untermenge einbezogen werden. Erbt eine der Metaklassen Attribute oder Referenzen, so ist die Eltern-Metaklasse ebenfalls in der Untermenge enthalten. Nicht enthalten sind abstrakte Eltern-Metaklassen, die keine Attribute und keine relevanten Referenzen enthalten (`Core::Element` als Eltern-Metaklasse von `ModelElement`, `Core::Relationship` als Eltern-Metaklasse von `Generalization` und `Dependency` sowie die Generalisierung zwischen `Namespace` und `DataType`).

Ein Werkzeug zur Erstellung eines Profiles muss diese Untermenge des Metamodells instanziieren können. Daraus ergeben sich folgende Anforderungen:

A1.1 *Es müssen von allen notwendigen Metaklassen Instanzen erzeugt werden können.*

Im Fall von UML 1.4 sind dies alle Metaklassen aus Abbildung 4.1, die nicht abstrakt sind. Zur Unterstützung von Datentypen können optional die Metaklassen `PrimitiveType`, `ProgrammingLanguageDataType`, `Operation` und `Parameter` hinzugenommen werden.

A1.2 *Attribute jeder erzeugten Instanz müssen mit Werten belegt werden können.*

Die Attribute sind aus Abbildung 4.1 zu entnehmen. Dabei müssen auch geerbte Attribute berücksichtigt werden.

A1.3 *Es müssen alle notwendigen Referenzen erzeugt werden können.*

Referenzen, für die das UML-Metamodell eine Multiplizität von Null erlaubt und die im Zusammenhang mit Profiles keine Bedeutung haben, brauchen nicht berücksichtigt zu werden und sind daher nicht in der Abbildung enthalten. Dazu gehört z. B. die Referenz `templateParameter` der Metaklasse `ModelElement`.

Weiterhin müssen Elemente verwendet werden können, die in der UML-Spezifikation zur Wiederverwendung vordefiniert sind:

A1.4 *Es müssen die vordefinierten Datentypen aus der UML-Spezifikation unterstützt werden.*

Die vordefinierten Datentypen aus der UML-Spezifikation ([Obj01d], S. 2-85) werden an zwei Stellen benötigt: zum einen sind damit die Typen der Meta-Attribute definiert, zum anderen können sie bei der Festlegung eines Typs in Tag Definitions (Attribut `tagType`) verwendet werden.

A1.5 *Es müssen benötigte Standard-Elemente verfügbar sein.*

Die Standard-Elemente sind vordefinierte Stereotypen, Tag Definitions und Constraints aus der UML-Spezifikation ([Obj01d], Anhang A) und werden zur Erstellung von UML-Profiles benötigt. So wird z. B. der vordefinierte Stereotyp `profile` zur Kennzeichnung von Profiles benötigt (siehe Abschnitt 2.2).

4.1.1.2 Werkzeug-Prüfung und -Unterstützung

Bei einer werkzeuggestützten Überprüfung eines notierten Profiles sind zu prüfen:

- die Einhaltung der abstrakten Syntax, gemäß dem UML-Metamodell (siehe Abbildung 4.1) und
- die Einhaltung zugehöriger Semantik, insbesondere der *Well-formedness Rules* (siehe [Obj01d], S. 2-9) aus der UML-Spezifikation.

Neben der Überprüfung des Modells ist der Anwender bei Erstellung des Profiles und der Angabe von Eigenschaften so weit wie möglich zu unterstüzen. Dies umfasst:

- Verstecken von Elementen aus dem Metamodell, die für den Anwender keine Rolle spielen oder redundante Information enthalten,
- Vorschlagen von sinnvollen Standard-Werten (*Default-Werten*) und
- Kenntlich machen der Auswahlmöglichkeiten bei begrenzten Alternativen.

Bei der Erstellung von Metaklassen ist dies wie folgt anzuwenden:

A1.6 *Bei der Erstellung einer neuen Instanz einer Metaklasse sollen nur Metaklassen gemäß der Kompositionen im Metamodell zur Auswahl stehen.*

In UML 1.4 bedeutet dies (siehe Abbildung 4.1):

- Oberstes Element ist ein Profile, d. h. ein Paket mit dem Stereotyp `profile`,
- ein Profile enthält Stereotypen, Datentypen und andere Profiles sowie Beziehungen zwischen diesen Elementen,
- Stereotypen enthalten Constraints und Tag Definitions und
- Aufzählungen enthalten Literale.

Gemäß diesen Regeln sollen Elemente vom Anwender ausgewählt werden dürfen, wodurch der Anwender bei der Eingabe unterstützt und die Gültigkeit des erstellten Modells gewährleistet wird.

A1.7 *Optional: In Abhängigkeit von der Darstellung sollten Hilfsklassen dem Anwender gegenüber wie Datentypen behandelt werden, statt wie eigenständige Elemente.*

In Abhängigkeit von der gewählten Darstellung (siehe 4.1.1.4) sollten Metaklassen, die für den Anwender nicht unbedingt als eigenständige Elemente zu erscheinen brauchen, wie Datentypen als Attribute anderer Metaklassen dargestellt werden. Dies betrifft hauptsächlich die nicht-graphische Darstellung. Ein Beispiel ist die Metaklasse `Generalization`: bei Definition eines Profiles beispielsweise in Tabellennotation erwartet der Anwender nicht unbedingt, eine Generalisierung als einen eigenständigen Bestandteil eines Profiles zu erstellen. Stattdessen ist es komfortabler, wenn der Anwender direkt das Eltern-Element angeben kann, z. B. bei einer Generalisierung von Stereotypen direkt den Eltern-Stereotypen. Bei graphischer Notation im Diagramm dagegen soll eine Generalisierung (gemäß der UML-Spezifikation) als eigenständiges Element dargestellt werden.

Bezüglich der Wertebelegung der Meta-Attribute von erstellten Instanzen soll folgende Unterstützung erbracht werden:

A1.8 *Attribute, deren Belegung im Zusammenhang mit Profiles fest ist, sollen versteckt oder als nicht editierbar angezeigt werden.*

Ein Beispiel ist das Attribut `isSpecification` der Metaklasse `ModelElement`, das bei der Erstellung von Profiles keine Relevanz hat und stets mit dem Standard-Wert `false` belegt wird.

A1.9 *Bei Attributen, die überwiegend mit dem gleichem Wert belegt werden, soll dieser als Standard-Wert vorgeschlagen werden.*

Beispielsweise sollte für das Attribut `visibility` der Metaklasse `ModelElement` der Wert `public` als Standard-Wert vorgegeben sein.

A1.10 *Bei Attributen mit begrenztem Wertebereich soll der Anwender aus gültigen Werten auswählen.*

Dies gilt z. B. für Attribute vom Typ `boolean`, oder für das Attribut `name` der Metaklasse `Stereotyp`, das den Namen einer Metaklasse aus der UML-Spezifikation enthält.

A1.11 *Bei Attributen mit frei einzugebenden Werten soll die Eingabe auf ihren Typ hin überprüft und der Anwender über Fehler benachrichtigt werden.*

Beispielsweise muss der Name eines Modellelements daraufhin überprüft werden, ob es sich um einen gültigen Bezeichner handelt.

A1.12 *Werte von Attributen, die voneinander abhängig sind, sollen konsistent sein.*

Bei der Metaklasse `GeneralizableElement` beispielsweise bezeichnet das Attribut `isRoot`, ob die jeweilige Instanz die Wurzel in einer Vererbungshierarchie

ist (also keine Eltern hat). Die Referenz `generalization` zeigt auf etwaige Eltern. Daher darf das Attribut `isRoot` nur dann mit dem Wert `true` belegt sein, wenn keine Referenz `generalization` vorhanden ist. Abhängigkeiten können auch zwischen Attributen verschiedener Instanzen bestehen: so muss z. B. jedes Modellelement innerhalb eines Namensraumes einen eindeutigen Namen haben.

Für Referenzen ergeben sich folgende Anforderungen:

A1.13 *Bei der Erstellung einer Referenz soll nur aus gültigen Modellelementen ausgewählt werden können.*

A1.14 *Bei Umbenennung oder Löschung eines Modellelements muss die Konsistenz von Referenzen überprüft werden.*

A1.15 *Bidirektionale Referenzen sollen durch eine einmalige Eingabe konsistent zu erstellen sein.*

Eine bidirektionale Referenz besteht z. B. zwischen einer Generalisierung und dem beteiligten Kind-Element. Ordnet der Anwender einem `GeneralizableElement` durch die Referenz `child` eine Generalisierung zu, so folgt daraus automatisch die Belegung von `generalization`. Die Belegung der Referenz `generalization` sollte daher vom Werkzeug vorgenommen werden.

4.1.1.3 Vorgehensweise und Methodik

Zur Erstellung von UML-Profiles ist bisher keine Vorgehensweise vorgesehen. Ansätze dazu finden sich in [Eik02] und in [Wur02] in Form von zwei möglichen Erstellungsstrategien: dem *Top-Down-Ansatz* und dem *Bottom-Up-Ansatz*. Diese Strategien sind hauptsächlich auf Profiles zur Modellierung spezieller Implementierungstechnologien bezogen. Beim Top-Down-Ansatz wird ausgehend von einem bestimmten Anwendungsbereich versucht, diesen durch möglichst wenige Erweiterungselemente vollständig durch UML modellierbar zu machen. Im Gegensatz dazu wird beim Bottom-Up-Ansatz von den einzelnen Details eines Anwendungsbereiches ausgegangen und jedes Detail durch ein Erweiterungselement möglichst äquivalent modellierbar zu machen versucht. Dies sind aber nur erste Ansätze, die nicht für eine Umsetzung in Werkzeugen bestimmt sind.

Neben einem Vorgehen für die Definition von Profiles, soll auch angesprochen werden, ob die Profile-Erstellung selbst auch in einen Softwareentwicklungs-Prozess einzuordnen ist. Die Entwicklung eines Profiles findet jedoch in einem einmaligen, längeren und eigenständigen Vorgang statt, weshalb keine solche Einordnung vorzunehmen ist.

Zusammenfassend existieren also keine Anforderungen bezüglich einer Vorgehensweise oder Methodik. Das Werkzeug beschränkt sich darauf, die Eingabe der Bestandteile eines Profiles zu ermöglichen. Die Reihenfolge ist lediglich durch die Kompositionen im Metamodell vorgegeben.

4.1.1.4 Benutzeroberfläche

Die UML-Spezifikation enthält zwei alternative Vorschläge zur Definition von Stereotypen: eine graphische Notation und eine Notation in tabellarischer Form ([Obj01d], S. 3-57 ff.). Diese enthalten auch die Definition zugehöriger Tag Definitions, Constraints

und Datentypen, und damit die Kernelemente eines Profiles. Für eine vollständige und zweckmäßige Definition von Profiles in einem Werkzeug sind beide Notationsformen jedoch nur mit Anpassungen geeignet, weshalb ihre Verwendung nicht obligatorisch ist.

A1.16 *Optional: Die Darstellung der Definition von Profiles erfolgt gemäß den Definitionen der UML-Spezifikation zur graphischen Notation von Modellelementen.*

Die graphische Notation der Definition von Stereotypen enthält auch alle zugehörigen Tag Definitions, Constraints und Datentypen. Für die Darstellung des Profiles selbst und von Modellbibliotheken sowie von Beziehungen zwischen den Elementen würde die übliche graphische Notation verwendet werden. Nachteilig ist jedoch, dass die graphische Definition bei einem wachsendem Umfang eines Profiles schnell unübersichtlich werden kann. Dies liegt besonders daran, dass Stereotypen eine große Zahl von Tagged Values und Constraints enthalten können. Somit sollten diese aus- und eingeblendet werden können. Ein weiteres Problem der graphischen Definition von Stereotypen ist ihre Unvereinbarkeit mit dem UML-Metamodell (siehe [Ple01], S. 33).

A1.17 *Optional: Die Darstellung der Definition von Profiles erfolgt in Tabellennotation analog zur tabellarischen Definition von Stereotypen in der UML-Spezifikation.*

Die Tabellennotation aus der Spezifikation müsste um Tabellen für das Profile selbst, für Modellbibliotheken und für Beziehungen erweitert werden. Auch für die Definition von Stereotypen muss der Vorschlag aus der Spezifikation noch angepasst werden, da dort z. B. nicht die Angabe eines Icons berücksichtigt ist. Da Stereotypen oftmals mehrere Constraints mit umfangreichen OCL-Ausdrücken enthalten, sollte diesen mehr Raum als nur eine Tabellenspalte zugestanden werden, z. B. eine eigene Tabellenzeile.

A1.18 *Optional: Die Gesamtstruktur eines Profiles wird in einem „Explorer-Fenster“ dargestellt.*

Zur Darstellung der Gesamtstruktur und zur Navigation enthalten die meisten CASE-Werkzeuge zusätzlich zum Hauptfenster ein Explorer-Fenster, in dem das Modell in einer Baumstruktur darstellt wird. Dies ist auch bei einem Werkzeug zur Definition von UML-Profiles sinnvoll.

4.1.1.5 Persistenz

Bezüglich Persistenz gelten folgende Anforderungen:

A1.19 *Es muss möglich sein, ein erstelltes Profile zu speichern.*

A1.20 *Es müssen Profiles zur weiteren Bearbeitung eingelesen werden können.*

A1.21 *Es müssen Profiles, die referenziert werden sollen, eingelesen werden können.*

Das Einlesen von Profiles ist auch dann notwendig, wenn ein Profile speziali-

siert oder Elemente daraus importiert werden sollen. Da in diesem Fall das Profile nur referenziert und nicht bearbeitet werden soll, sollte nur lesender Zugriff erfolgen.

A1.22 *Es muss das Austauschformat XMI unterstützt werden.*

Zu XMI siehe Kapitel 3.

4.1.2 Anwendung von UML-Profiles

UML-Profiles werden bei der Modellierung angewendet. Daher benötigt ein Werkzeug zur Anwendung von UML-Profiles als Voraussetzung die Funktionalität eines UML-Werkzeuges. Gemäß der Einführung in 4.1 werden im Folgenden nur die Anforderungen an das Werkzeug aufgestellt, die speziell die Unterstützung von UML-Profiles betreffen.

4.1.2.1 Modellnotation

An dieser Stelle muss das gesamte UML-Metamodell unterstützt werden und instanziiert werden können. Ausgenommen sind die Metaklassen, die zur Erstellung von Profiles dienen, da das Profile bereits definiert wurde. Von den Erweiterungselementen aus dem Paket `Extension Mechanisms` ist also nur die Metaklasse `TaggedValue` zu instanziieren.

A2.1 *Es müssen von allen notwendigen Metaklassen Instanzen erzeugt werden können.*

Dies sind Tagged Values und Abhängigkeiten. Letztere werden zur Darstellung der Abhängigkeit zwischen dem Modell und dem Profile bzw. zugehörigen Modellbibliotheken benötigt.

A2.2 *Attribute jeder erzeugten Instanz müssen mit Werten belegt werden können.*

Hier sind dies ein oder mehrere Werte eines Tagged Values. Die Anzahl der Werte eines Tagged Values ist im Attribut `multiplicity` der zugehörigen Tag Definition festgelegt.

A2.3 *Es müssen alle notwendigen Referenzen erzeugt werden können.*

Zur Anwendung eines Profiles in einem Modell werden Referenzen zwischen Elementen aus dem Profile und Elementen aus dem Modell erstellt. Konkret sind dies folgende Referenzen:

- vom Modell (bzw. Paketen allgemein) über eine Abhängigkeit (mit dem Stereotyp `appliedProfile`) auf ein Profile,
- vom Modell (bzw. Paketen allgemein) über eine Abhängigkeit (bzw. `Permission`) auf eine Modellbibliothek,
- von Modellelementen auf Stereotypen,
- von Tagged Values auf Tag Definitions und

- von Modellelementen auf Tagged Values.

Dazu müssen als Voraussetzung die jeweiligen Elemente vorhanden sein:

A2.4 *Es müssen definierte Profiles verfügbar sein und auf ein Modell angewendet werden können.*

A2.5 *Ein Element aus einem Profile soll genau dann in einem Modell verfügbar sein, wenn es gemäß der Semantik von Beziehungen und Attributen bei der Anwendung des Profiles für das Modell sichtbar ist.*

Hier sind zu berücksichtigen:

- Die Sichtbarkeit von Elementen (Attribut `visibility`),
- Import- und Access-Beziehungen und
- Generalisierungen.

A2.6 *Die Untermenge, auf die das Profile anwendbar ist, muss vorhanden sein.*

Dies sollte dem Tagged Value `applicableSubset` des Profiles entsprechen. Im einzelnen sind erforderlich:

- Metaklassen, für die ein Stereotyp definiert wurde (Attribut `baseClass` der Metaklasse `Stereotype`) und
- Metaklassen, die durch einen Tagged Value referenziert werden sollen (Attribut `referenceValue` der Metaklasse `Tagged Value`).

A2.7 *Es müssen benötigte UML-Datentypen unterstützt werden.*

Datentypen werden bei Tagged Values benötigt.

4.1.2.2 Werkzeug-Prüfung und -Unterstützung

Hier gelten prinzipiell die gleichen Anforderungen aus 4.1.1.2 zur Erstellung von Metaklassen, Attributen und Referenzen, jedoch bezogen auf das Modell aus 4.1.2.1. Allerdings sind nicht alle Anforderungen aus 4.1.1.2 tatsächlich relevant, da zur Anwendung von Profiles in UML 1.4 nur wenige Metaklassen, Attribute und Referenzen notwendig sind. Es werden im folgenden nur die Anforderungen genannt, die zur Umsetzung von UML 1.4 benötigt werden.

Für Metaklassen gelten folgende Anforderungen:

A2.8 *Bei der Erstellung einer neuen Instanz einer Metaklasse sollen nur Metaklassen gemäß der Kompositionen zur Auswahl stehen.*

Dies bedeutet, dass Tagged Values zu Modellelementen hinzugefügt werden können, die mit dem entsprechenden Stereotypen versehen sind.

A2.9 *Optional: In Abhängigkeit von der Darstellung sollten Hilfsklassen dem Anwender gegenüber wie Datentypen behandelt werden, statt wie eigenständige Elemente.*

Ein Beispiel ist die Abhängigkeit zwischen einem Modell und einem verwendeten Profile. In Abhängigkeit von der gewählten Darstellung (siehe 4.1.2.4) sollte diese vor dem Anwender verborgen werden, wenn beispielsweise die verwendeten Profiles nicht graphisch im Diagramm dargestellt werden, sondern in einer Liste.

Für Meta-Attribute sind folgende Anforderungen zu beachten:

A2.10 *Attribute, deren Belegung fest ist, sollen als nicht editierbar angezeigt werden.*

Dies gilt für das Attribut `name` von Tagged Values, das dem Namen der zugehörigen Tag Definition entsprechen muss.

A2.11 *Bei Attributen mit begrenztem Wertebereich soll der Anwender aus gültigen Werten auswählen.*

Dies gilt für Tagged Values, deren Typ ein Datentyp mit begrenztem Wertebereich ist.

A2.12 *Bei Attributen mit frei einzugebenden Werten soll die Eingabe auf ihren Typ hin überprüft und der Anwender über Fehler benachrichtigt werden.*

Dies gilt für Tagged Values, deren Typ ein Datentyp mit großem Wertebereich ist.

Bezüglich Referenzen müssen folgende Anforderungen erfüllt werden:

A2.13 *Bei der Erstellung einer Referenz soll nur aus gültigen Modellelementen ausgewählt werden können.*

Zusätzlich zur abstrakten Syntax des Metamodells ist dabei besonders zu beachten:

- Tagged Values referenzieren Instanzen der Metaklasse, die im Attribut `tagType` der zum jeweiligen Tagged Value gehörenden Tag Definition angegeben ist und

- Stereotypen werden zu Instanzen der Metaklasse zugewiesen, die im Attribut `baseClass` des jeweiligen Stereotypen angegeben ist.

A2.14 *Bei Umbenennung oder Löschung eines Modellelements muss die Konsistenz von Referenzen überprüft werden.*

A2.15 *Bidirektionale Referenzen sollen durch eine einmalige Eingabe konsistent zu erstellen sein.*

Eine bidirektionale Referenz besteht z. B. zwischen einer Abhängigkeit und dem beteiligten Element mit der Rolle des *Client*. Der Anwender soll nicht separat beide Richtungen der Referenz eingeben müssen.

4.1.2.3 Vorgehensweise und Methodik

Zweck eines Profiles ist die Modellierung eines speziellen Anwendungsbereichs. Ein bestimmter Anwendungsbereich kann unter Umständen eine besondere Vorgehensweise bei der Modellierung erfordern, z. B. ist beim Profile für Variabilität ein produktlinienorientiertes Vorgehen vorgesehen. Meist nehmen Profiles Anpassungen an eine bestimmte Implementierungstechnologie vor, so dass sich die Vorgehensweise durch das Profile nicht verändert.

Grundsätzlich geht die Semantik eines bestimmten Vorgehens über den Fokus von UML hinaus (siehe [Obj01d], S. 1-8), weshalb ein Profile, das nur Bestandteile enthält, die in der UML-Spezifikation vorgesehen sind, nicht explizit (bzw. werkzeugunterstützt) die Vorgehensweise beeinflusst. Ein Überblick über eine Semantik von Profiles über den Rahmen der UML-Spezifikation hinaus befindet sich in Abschnitt 4.3.

4.1.2.4 Benutzeroberfläche

A2.16 *Elemente aus dem Profile, die in Beziehung zu Modellelementen stehen, müssen gemäß der Spezifikation graphisch in Diagrammen visualisiert werden können.*

Dazu zählen

- Stereotypen in allen drei Darstellungsvarianten,
- Tagged Values,
- Modellbibliotheken,
- Elemente aus Modellbibliotheken und
- das Profile-Paket selbst.

Die graphische Notation des Profile-Pakets ist dabei optional, da dieses semantisch nicht unmittelbar zum Modell gehört.

A2.17 *Erweiterungselemente, die in Diagrammen visualisiert werden, sollen wahlweise ein- und ausgeblendet werden können.*

In manchen Fällen sollen Stereotypen oder Tagged Values nicht im Diagramm sichtbar sein, um die Darstellung übersichtlicher zu gestalten, oder beispielsweise weil das Erweiterungselement nur für die Codegenerierung relevant ist.

A2.18 *Bei Elementen, für die mehrere Darstellungsvarianten in der Spezifikation vorgesehen sind, sollte der Anwender zwischen diesen auswählen können.*

In UML 1.4 sind für Stereotypen drei verschiedene Darstellungsvarianten definiert (siehe Abschnitt 2.2).

A2.19 *Optional: Virtuelle Metaklassen sollten wie normale Metaklassen behandelt werden können.*

Modellelemente, die mit einem Stereotypen versehen sind, werden oft als Instanz einer virtuellen Metaklasse betrachtet. Daher sollte es möglich sein, diese – analog zu Instanzen von Standard-UML-Metaklassen – über die Werkzeugleiste zu erstellen.

4.1.2.5 Persistenz

A2.20 *Profiles müssen eingelesen werden können.*

A2.21 *Beim Speichern eines Modells müssen alle Referenzen auf Profiles und deren Elemente mitgespeichert werden.*

Modelle sollen standardmäßig nicht alle verwendeten Profiles mit abspeichern, sondern es genügen Referenzen auf Profiles und deren Elemente.

A2.22 *Beim Einlesen eines Modells mit Referenzen auf Profiles müssen diese wieder im Werkzeug verfügbar sein.*

A2.23 *Verwendete Erweiterungsmechanismen aus Profiles müssen mit dem Modell gemeinsam gespeichert werden können, so dass das Modell auch ohne die Verfügbarkeit von Profiles vollständig eingelesen werden kann.*

Diese Art der Persistenz stellt eine Alternative zu A2.21 dar. Es ist notwendig, die verwendeten Elemente aus Profiles mit zu speichern, z. B. wenn das Modell weitergegeben werden soll, und der Empfänger über keine Profile-Unterstützung verfügt und nur das Modell lesen möchte.

A2.24 *Es soll das Austauschformat XMI unterstützt werden.*

Zu XMI siehe Kapitel 3.

4.2 Integration in bestehende UML-Werkzeuge

Ein wichtiger Vorteil von UML-Profiles ist die Integration von speziellen Anwendungsbereichen in UML, ohne das Metamodell zu verändern. Dadurch können bestehende Werkzeuge weiter verwendet werden. Dabei ist vorausgesetzt, dass die Werkzeuge auch den Profile-Mechanismus aus der UML-Spezifikation implementieren. Abgesehen vom Werkzeug *Objecteering* der Firma *Objecteering Software* [OBJ02a] ist dies bisher noch kaum der Fall.

Einige Werkzeuge stellen Schnittstellen zur Erweiterung ihrer Funktionalität bereit. Dadurch ist es möglich, bestehende Werkzeuge um einen Profile-Mechanismus zu erweitern. Im Detail sind die vorzunehmenden Erweiterungen abhängig vom zu erweiternden Werkzeug und von den Möglichkeiten und Grenzen der angebotenen Erweiterungsschnittstelle. Ein konkretes Beispiel findet sich dazu in Kapitel 5 anhand von *Rational Rose*. Dort wird die Erweiterungsschnittstelle vorgestellt und die Schritte zur Erweiterung um Profile-Unterstützung werden demonstriert. Ein weiteres Beispiel für Erweiterungsmöglichkeiten eines konkreten Werkzeuges findet sich mit dem Werkzeug *Together* von *Togethersoft* [TOG02] in [Ple01], Kap. 6.

In diesem Abschnitt werden werkzeugunabhängig die dazu notwendigen grundlegenden Schritte zusammengefasst. Analog zu Abschnitt 4.1 wird zwischen Erstellung und Anwendung von Profiles unterschieden, wobei die Erstellung von Modellbibliotheken separat betrachtet wird. Ausgangspunkt ist ein UML-Werkzeug, das UML-Profiles nicht unterstützt und eine Schnittstelle zur Erweiterung bietet.

4.2.1 Erstellung von UML-Profiles

Zur Erstellung von Profiles bedarf es nur einer kleinen Anzahl von Metaklassen (siehe 4.1.1.1), im wesentlichen die UML-Erweiterungselemente. Eine graphische Darstellung mittels Diagrammen ist möglich, aber nicht zwingend notwendig (siehe 4.1.1.4). Daher ist bei der Erstellung von Profiles prinzipiell keine Funktionalität aus bestehenden UML-Werkzeugen notwendig und es kann die Erstellung von Profiles werkzeugunabhängig in einem eigenständigen Werkzeug erfolgen.

Soll die Erstellung von Profiles in das zu erweiternde Werkzeug integriert werden, so sind verschiedene Abstufungen denkbar. Eine einfache Lösung wäre, das Werkzeug zur Profile-Erstellung aus dem UML-Werkzeug heraus aufrufbar zu machen, z. B. mittels eines Menüpunktes.

Alternativ können auch Mechanismen des UML-Werkzeuges zur Eingabe des Profiles genutzt werden, z. B. durch die Definition eigener Dialoge oder durch die Anpassung eines Explorer-Fensters.

Soll eine graphische Definition von Profiles in das UML-Werkzeug integriert werden, so muss dort ein neuer Diagrammtyp definiert werden. Behelfsmäßig kann auch eine bestehende Diagrammart, z. B. das Klassendiagramm, entsprechend abgewandelt werden. Ein Beispiel für Letzteres findet sich in Abschnitt 5.1.3.2.

4.2.2 Modellbibliotheken

Modellbibliotheken enthalten wiederverwendbare Elemente. Hier ist keine erweiterte Profile-spezifische Funktionalität notwendig, sondern die Grundfunktionalität von UML-Werkzeugen: die Definition und die Anwendung wiederverwendbarer Elemente, wie z. B. Klassen und Datentypen. Dazu kann die bestehende Funktionalität des UML-Werkzeuges verwendet werden. Eine Modellbibliothek wird demnach definiert, indem im vorhandenen Werkzeug ein Paket mit den wiederverwendbaren Elementen erstellt wird. Bei der Anwendung der Modellbibliothek muss dieses lediglich wieder in das Werkzeug eingelesen werden.

Der einzige formale Unterschied zwischen einem normalen Paket und einer Modellbibliothek besteht darin, dass Modellbibliotheken durch den Stereotyp `modelLibrary` gekennzeichnet werden. Das UML-Werkzeug muss gegebenenfalls um die Unterstützung dieses Stereotyps erweitert werden.

Offen bleiben zunächst Anforderungen bezüglich des Speicherns und des Einlesens von Modellbibliotheken. Dies ist zum einen die Forderung nach Unterstützung des werkzeugunabhängigen Formates XMI. Zum anderen müssen Modellbibliotheken auf sinnvolle Weise für den Anwender verfügbar sein.

Zur Unterstützung von XMI kann der Import- und Export-Mechanismus des jeweiligen Werkzeuges verwendet werden. Falls das Werkzeug XMI nicht unterstützt, müsste es entsprechend erweitert werden. Dies wäre jedoch eine recht umfangreiche Erweiterung, da Modellbibliotheken prinzipiell beliebige Elemente enthalten können. Auch bei einer Beschränkung auf Klassen und Datentypen wäre noch eine große Anzahl an Elementen zu unterstützen, da Klassen sehr viele zugehörige Modellelemente (z. B. Attribute, Methoden, Operationen, Parameter, Relationen zu anderen Klassen) referenzieren können.

Wird ein Profile verwendet, das eine Modellbibliothek referenziert, so muss diese automatisch für den Anwender verfügbar sein, so dass der Anwender Elemente aus ihr wiederverwenden kann. Demnach müsste sie Teil des Modells des Anwenders sein. Es ist aber vorstellbar, dass der Anwender nicht automatisch sein Modell um die Mo-

dellbibliothek erweitern möchte. Daher sollte eine Modellbibliothek dem Modell auf nur Wunsch des Anwenders hinzugefügt werden – analog zu verfügbaren Profiles, die ebenfalls nur auf Wunsch des Anwenders auf ein Modell angewendet werden. Folglich kann der Fall eintreten, dass ein Anwender erst während der Modellierung eine Modellbibliothek in sein Modell einbinden möchte.

Zunächst soll der Fall betrachtet werden, dass der Anwender bereits vor Beginn der Erstellung eines neuen Modells weiss, dass er eine Modellbibliothek nutzen möchte. In diesem Fall ist diese Modellbibliothek lediglich zu öffnen und steht zur Verfügung.

Schwierigkeiten treten auf, wenn der Anwender bereits Teile seines Modells erstellt hat und eine Modellbibliothek seinem Modell hinzufügen möchte. In diesem Fall muss diese Bibliothek in das Modell des Anwenders integriert werden. Möglicherweise wird dies vom UML-Werkzeug nicht unterstützt, da z. B. beim Öffnen eines Modells das bisher im Werkzeug befindliche Modell geschlossen wird. Dann muss das Werkzeug dahingehend erweitert werden. Ein möglicher werkzeugunabhängiger Ansatz wäre, das Modell ins XMI-Format zu exportieren und die Modellbibliothek in die XMI-Datei einzufügen.

4.2.3 Anwendung von Profiles

Bei der Anwendung von Profiles in bestehenden Werkzeugen sind die meisten Integrationsschritte notwendig.

Ein grundsätzlich notwendiger Schritt ist die Abbildung der UML-Metaklassen aus dem Profile auf Metaklassen aus dem internen Metamodell des zu erweiternden Werkzeuges. Bestehende UML-Werkzeuge unterstützen meist nur einen Teil des UML-Metamodells bzw. ältere UML-Versionen. Andernfalls würden sie bereits Profile-Unterstützung implementieren. In der Praxis basieren UML-Werkzeuge intern meist auf einer pragmatischen proprietären Abwandlung des UML-Metamodells. Daher muss bei der Integration von UML-konformer Profile-Unterstützung eine Abbildung von UML-Metaklassen auf entsprechende Gegenstücke aus dem Metamodell des Werkzeuges vorgenommen werden. Dies gilt insbesondere an den Stellen, wo im Profile ein beliebiger Metaklassen-Name angegeben werden kann, d. h. bei der `baseClass` von Stereotypen und bei Typen von Tag Definitions (Anforderung A2.6). Die dort bei der Erstellung des Profiles angegebenen Metaklassen werden als vorhanden vorausgesetzt. Zur Anwendung im Werkzeug muss jeder Metaklassen-Name auf sein Gegenstück aus dem internen Metamodell des Werkzeuges abgebildet werden.

Neben der Nutzung der vorhandenen Metaklassen müssen Bestandteile von Profiles dem Werkzeug hinzugefügt werden, so dass die Anforderungen aus Abschnitt 4.1.2 erfüllt werden. Oft wird dies so nicht direkt möglich sein, und die UML-Erweiterungselemente müssen z. B. den Elementen des Modells als Eigenschaften hinzugefügt werden. Diese Integrationsschritte sind abhängig von den Möglichkeiten der Schnittstelle zur Erweiterung des jeweiligen Werkzeuges und können hier nicht weiter allgemein diskutiert werden.

4.3 Erweiterte Verwendung von Profiles

Bisher wurde in dieser Arbeit die Funktionalität von UML-Profiles im Rahmen der UML-Spezifikation betrachtet. UML-Werkzeuge bieten jedoch Funktionalität über den Rahmen von UML hinaus. Darunter sind auch Funktionen, die einen wichtigen Teil der Nützlichkeit von Werkzeugen ausmachen, wie z. B. die Generierung von Code. Es

wäre wünschenswert, die Anpassung an einen bestimmten Anwendungsbereich auch auf diese Funktionen auszudehnen, so dass z. B. bei Verwendung eines Profile für EJB (siehe [Rat01a]) auch Code für EJB generiert wird. Die UML-Spezifikation grenzt sich von dieser Funktionalität explizit ab (siehe [Obj01d], 1.5.1) und überlässt diese den Herstellern von Werkzeugen.

Im folgenden wird ein Überblick über diese erweiterte Funktion von Profiles gegeben. Dazu soll auch die semantische Auswertung von Constraints gezählt werden, da diese im begrenzten Rahmen dieser Arbeit ebenfalls nicht weiter betrachtet werden soll, obwohl die UML-Spezifikation mit der standardisierten Sprache OCL die Voraussetzung dazu bietet.

Voraussetzung für die jeweilige erweiterte Funktionalität ist eine Sprache zur Formulierung zugrundeliegender Semantik in den Profiles. Soll z. B. die Codegenerierung durch ein Profile angepasst werden, so müssen die Generierungsregeln in einem Profile formuliert werden können. Funktionalität und Sprache zur Formalisierung der gewünschten Semantik stehen also im direkten Zusammenhang.

In Anlehnung an [Des00] sind drei Arten weitergehender Funktionalität von UML-Profiles zur Anpassung von Werkzeugen an einen bestimmten Anwendungsbereich zu unterscheiden:

1. Verifikation von Modellen,
2. Transformation von Modellen und
3. Anpassung der Werkzeugumgebung.

Im folgenden wird jeder Punkt kurz erläutert, wobei Beispiele für vorhandene Sprachen zur Formalisierung von Semantik vorgestellt werden.

4.3.1 Validierung von Modellen

Dieser Punkt unterscheidet sich von den nachfolgenden Punkten dadurch, dass in der UML-Spezifikation bereits eine Sprache zur Formalisierung solcher Semantik vorhanden und für die Verwendung mit Profiles vorgesehen ist.

Prinzipiell dienen zur Verifikation von Modellen Constraints, also Bedingungen, durch die *Well-Formedness Rules* formuliert werden. Dabei handelt es sich um Constraints auf Metamodellebene, die von jedem gültigen UML-Modell zu jedem Zeitpunkt erfüllt werden müssen (siehe [Obj01d], S.2-9). In der UML-Spezifikation sind dies Regeln bezüglich des Metamodells, die von den Instanzen der Metaklassen erfüllt werden müssen. Analog dazu werden in Profiles den Stereotypen Constraints zugeordnet, die Regeln bezüglich des virtuellen Metamodells spezifizieren, die von Instanzen der virtuellen Metaklassen zu erfüllen sind.

Mit der *Object Constraint Language* (OCL) [Obj01d], Kap. 6, aus der UML-Spezifikation steht eine standardisierte formale Sprache zur Formulierung dieser Regeln zur Verfügung. Die OCL kann werkzeugunterstützt ausgewertet werden, z. B. mit Hilfe des *Dresden OCL Toolkit* [Tec02], das bereits in kommerzielle Werkzeuge integriert wurde (z. B. *Poseidon* von *Gentleware* [GEN02]). Diese Ansätze sind allerdings zur Auswertung von Constraints aus Modellen vorgesehen und beziehen sich damit auf die Modellebene. Zur Auswertung von Constraints aus Profiles ist jedoch der Bezug auf die Metamodellebene notwendig. Hier sind bestehende Ansätze noch entprechend weiter zu entwickeln bzw. zu erweitern. Bisher werden alle Regeln auf Metamodellebene in UML-Werkzeugen üblicherweise „hart kodiert“.

4.3.2 Transformation von Modellen

Die Transformation von Modellen ist ein wichtiger Bestandteil der Werkzeugunterstützung von UML. Modelle sollen durch das Werkzeug zumindest teilweise automatisch in gewünschte Endprodukte transformiert werden können. Endprodukte sind z. B. Code in einer oder mehreren Programmiersprachen oder Dokumentation. Es wäre wünschenswert, dass die Transformationen durch ein UML-Profile angepasst werden können: Häufig dienen Profiles zur Modellierung einer bestimmten Implementierungstechnologie wie z. B. das UML-Profile für CORBA [Obj02c]. Soll aus damit erstellten Modellen Code generiert werden, so ist es wünschenswert, dass auch der generierte Code an die spezielle Implementierungstechnologie angepasst ist. Dazu müssten Profiles entsprechende Transformationsregeln beinhalten können.

Zur Formalisierung solcher Semantik gibt es bisher nur proprietäre Ansätze. Das Werkzeug *Objecteering* von *Objecteering Software* [OBJ02a] (ehemals *Softeam*) bietet hierzu die Java-ähnliche Sprache *J*. Dem Entwickler eines Profiles stehen alle Metaklassen als J-Klassen zur Verfügung. Weiterhin hat er die Möglichkeit, eigene Endprodukte von Transformationsprozessen, wie z. B. Code oder Dokumentation, als eigene J-Klassen zu definieren. Darüberhinaus enthält J Hilfsklassen zur Verwaltung und Anzeige der definierten Endprodukte, wie z. B. einen Editor zur Anzeige von generiertem Code. Unter Verwendung dieser Mechanismen kann der Entwickler den Bestandteilen seines Profiles J-Methoden hinzufügen, die eine Transformation in selbstdefinierte Endprodukte vornehmen. Eine ausführliche Beschreibung findet sich in [Ple01], Kap. 5.

4.3.3 Anpassung der Werkzeugumgebung

Als drittes kann die Anpassung der Werkzeugumgebung für die Modellierung eines bestimmten Anwendungsbereichs wünschenswert sein. Hier sind je nach Anwendungsbereich vielfältige Anforderungen denkbar, z. B. die Anpassung von Dialogen, die Anpassung der Vorgehensweise (siehe 4.1.1.3) oder das Ausblenden von bestimmten Details in Modellen.

Die Sprache J in Objecteering (siehe 4.3.2) ermöglicht dem Entwickler eines Profiles auch die Definition dieser Art von Semantik. Dazu werden Methoden definiert, die bei Anwendung des Profiles aufgerufen werden. Die Elemente der Werkzeugumgebung sind Teil des internen Metamodells in Objecteering und sind als J-Klassen ansprechbar. Dadurch kann der Anwender sie mittels J durch sein Profile anpassen (siehe [Ple01], Kap. 5).

4.3.4 Zusammenfassung

Zusammenfassend ist festzuhalten, dass für die Werkzeugunterstützung von UML-Profiles auch Funktionalität über den Rahmen der UML hinaus sinnvoll ist. In dieser Arbeit wird diese aber nicht weiter behandelt. Als Sprache zur Formulierung der Semantik steht nur für die Validierung von Modellen mit OCL ein Standard zur Verfügung. Für weitere Funktionalität sind bereits proprietäre Lösungen vorhanden. Eine Lösung, um Profiles mitsamt umfangreicher Semantik werkzeugunabhängig und austauschbar zu gestalten, ist bisher nicht vorgesehen.

Ein prinzipieller Ansatz zur Integration weitergehender Semantik in den Profile-Mechanismus wäre, diese in eigenständigen Einheiten zu kapseln, welche von Profiles – analog zu Modellbibilotheken – referenziert werden können. Zum einen können dann

einem Profile flexibel verschiedene Arten von Semantik hinzugefügt werden. Zum anderen kann auch eine Art von Semantik, z. B. Semantik zur Codegenerierung, mehrfach für ein Profile notwendig sein. Beispielsweise ist das Profile für Variabilität unabhängig von einer Programmiersprache, weshalb für mehrere Programmiersprachen Semantik zur Codegenerierung notwendig sein kann.

Kapitel 5

Integration von Profile-Unterstützung in Rose

Im folgenden Kapitel sollen die vorangegangenen Untersuchungen an einem konkreten UML-Werkzeug praktisch umgesetzt werden. Grundlage sind die Anforderungen aus Abschnitt 4.1, sowie die allgemeinen Schritte zu einer Integration aus Abschnitt 4.2. Als konkretes UML-Werkzeug dient *Rose* von der amerikanischen Firma *Rational Software*. Rose bietet eine umfangreiche Erweiterungsschnitstelle. Ein weiteres Argument für die Auswahl von Rose ist seine weite Verbreitung. Nach einer Untersuchung der unabhängigen Firma IDC [IDC02] ist Rose mit rund 30 Prozent Marktanteil das deutlich marktführende Werkzeug ([Rat02b]) im Bereich Analyse, Modellierung und Design (*AMD*).

Ein Anwendungsbeispiel aus der Industrie ist die Firma Intershop [ISH02], in der das Profile für Variabilität im Rahmen des Forschungsprojektes *PLIV* (*Product Line Implementation Variabilities*) verwendet wird und später in der Softwareentwicklung eingesetzt werden soll. Bei Intershop wird ebenfalls Rose verwendet, so dass dort bei einer Erweiterung von Rose um Profile-Unterstützung das Variabilitäts-Profile und zukünftige Profiles ohne zusätzliche Neuanschaffungen verwendet werden können.

Zunächst werden Rose und seine Erweiterungsmöglichkeiten vorgestellt. Anschließend werden Konzepte zur Integration von Profile-Unterstützung beschrieben. Dabei werden analog zu Abschnitt 4.1 die Erstellung von Profiles und die Anwendung von Profiles unterschieden.

5.1 Erweiterungsmöglichkeiten von Rose

Dieser Abschnitt beschreibt Rose und seine Erweiterungsmöglichkeiten. Zunächst wird der Grundzustand von Rose dargestellt. Anschließend wird die Erweiterungsschnittstelle erläutert. Abschließend werden bereits vorhandene Erweiterungen, die für die für die weitere Arbeit von Bedeutung sind, vorgestellt.

5.1.1 Ausgangszustand

Die vorliegende Arbeit bezieht sich auf die aktuelle Version, Rose 2002. Bezüglich der Betrachtungen in dieser Arbeit gibt es allerdings kaum Unterschiede zu den Vorgänger-Versionen, so dass die Ausführungen auch für ältere Versionen gültig sind.

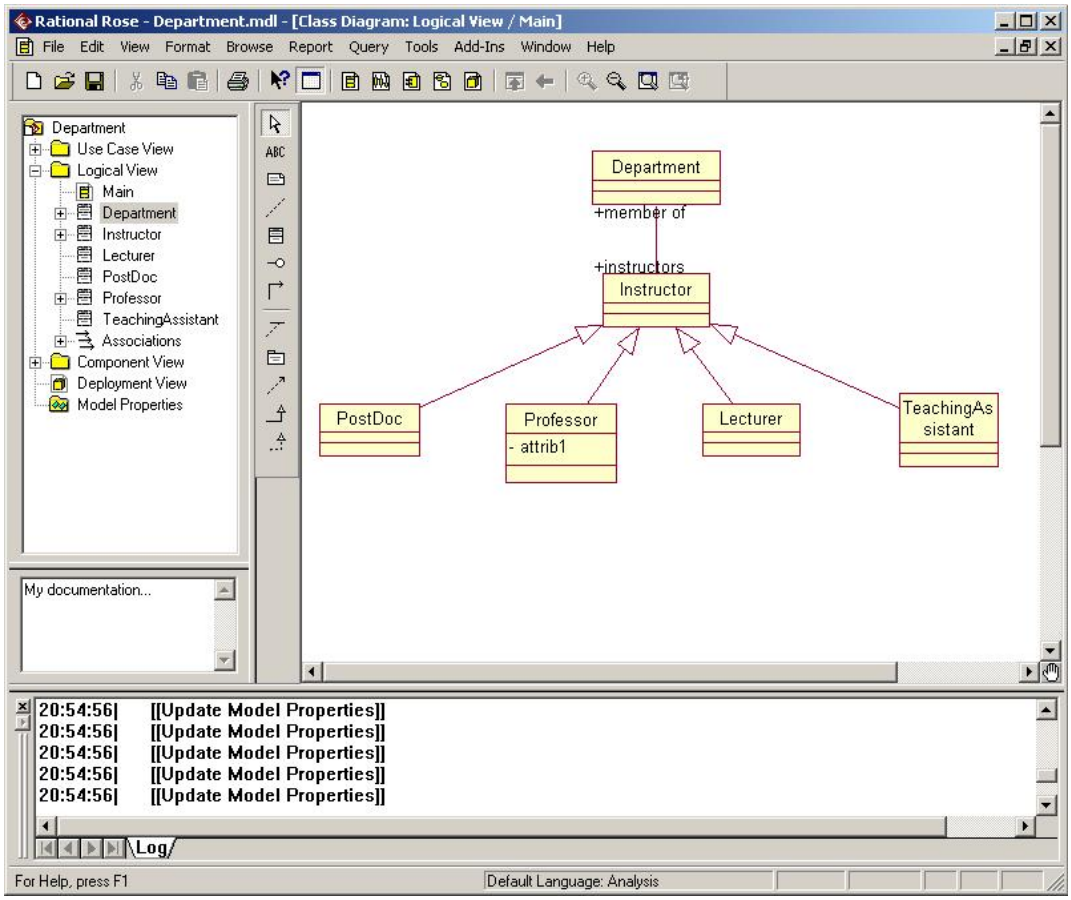

Abbildung 5.1: Rational Rose

5.1.1.1 Grundlegende Funktionalität

Abbildung 5.1 zeigt das Werkzeug im Ausgangszustand. Links oben befindet sich das *Explorer-Fenster*. Im Explorer-Fenster werden alle Sichten auf das aktuell bearbeitete Modell in einer Baumstruktur dargestellt. Es dient zur Navigation und als Überblick über die Gesamtstruktur. Darunter befindet sich das *Dokumentations-Fenster*. Es enthält beliebige textuelle Dokumentation zum aktuell markierten Modellelement oder Diagramm.

Rechts befindet sich das *Hauptfenster*. Dieses ermöglicht in enthaltenen *Diagramm-Fenstern* die Erstellung und Bearbeitung von UML-Diagrammen – in der Abbildung 5.1 beispielhaft ein Klassendiagramm. Rose unterstützt alle Arten von UML-Diagrammen.

Im unteren Bereich befindet sich das Konsolen-Fenster, das zur Ausgabe von Nachrichten und Fehlermeldungen des Programms dient. Zu jedem Modellelement lässt sich ein *Spezifikations-Fenster* (Abbildung 5.2) öffnen. Dort können die Eigenschaften des jeweiligen Modellelementes spezifiziert werden.

Rational macht keine Angaben bezüglich des von Rose unterstützten UML-

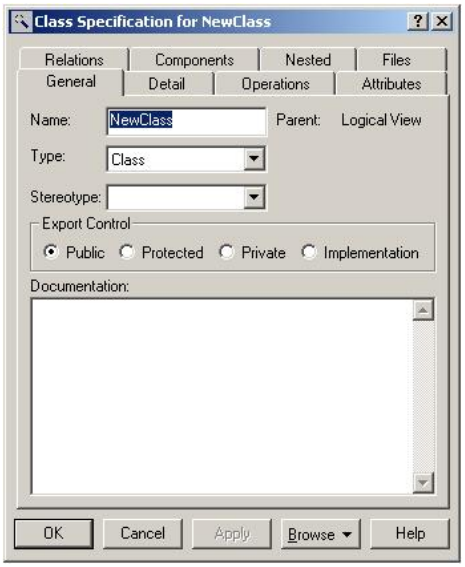

Abbildung 5.2: Das Spezifikations-Fenster zu einer Klasse `NewClass`.

Metamodells. Im Wesentlichen wird UML der Version 1.3 unterstützt.

5.1.1.2 Stereotypen

Rational Rose unterstützt Stereotypen. Diese können einem Modellelement im Spezifikations-Fenster zugewiesen werden. Dort kann der Stereotyp aus einer Liste ausgewählt werden. Die Auswahlliste ist abhängig von der Metaklasse des jeweiligen Modellelementes, d. h. Stereotypen haben in Rose eine Base-Class. Alternativ zur Auswahl aus der Liste kann auch ein beliebiger eigener Stereotypen-Name eingegeben werden. Dieser wird im aktuellen Modell der Auswahlliste der jeweiligen Metaklasse hinzugefügt, so dass innerhalb eines Modells derselbe Stereotyp für eine Metaklasse nicht mehrfach per Hand eingegeben werden muss.

Im Diagramm kann für Stereotypen zwischen allen drei Darstellungsformen, die in der UML-Spezifikation vorgesehen sind, gewählt werden. Das Ausblenden von Stereotypen ist ebenfalls möglich. Die Darstellung mittels Icon ist nur möglich bei Stereotypen aus der Auswahlliste, für die ein Icon definiert ist.

Sofern für einen Stereotypen entsprechende weitere Icons definiert sind, kann er durch ein kleines Icon im Explorer-Fenster sichtbar gemacht, sowie der Werkzeugleiste im Diagrammfenster hinzugefügt werden. Letzteres erlaubt, direkt eine Instanz der virtuellen Metaklasse zu erzeugen, d. h. eine Instanz der Base-Class des Stereotypen, die bereits mit dem Stereotyp versehen ist. Dies gilt jedoch nur für Stereotypen, deren Base-Class als eigenständiges Element in Diagrammen existieren kann, also z. B. nicht für Sterotypen für Attribute, da Attribute stets Teil eines Modellelementes sind.

Die Zuweisung mehrerer Stereotypen zu einem Modellelement ist in Rose nicht möglich. Die Karteikarte *UML* jedoch, die Rose durch die eine Erweiterung der Firma Unisys (siehe Abschnitt 5.1.3.1) hinzugefügt wird, enthält unter anderem eine Eigenschaft `secondaryStereotype` vom Typ String zur Angabe eines beliebigen zweiten Stereotyp-Namens für ein Modellelement. Auswirkungen, z. B. auf die Darstellung

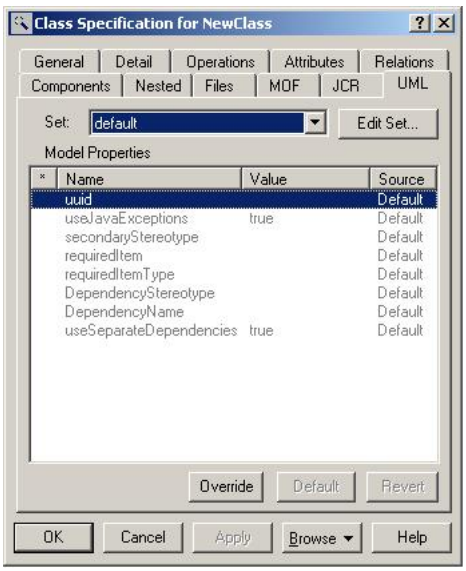

Abbildung 5.3: Karteikarte einer Erweiterung von Rose im Spezifikations-Fenster zu einer Klasse `NewClass`.

im Diagramm, hat die Angabe des zweiten Stereotypen jedoch nicht.

5.1.1.3 Eigenschaften von Modellelementen

Bei den Eigenschaften eines Modellelements macht Rose keine Unterscheidung zwischen Attributen und Assoziationen aus dem Metamodell und erweiterten Eigenschaften, also Tagged Values. Alle Eigenschaften sind im Spezifikations-Fenster enthalten. Eine graphische Darstellung von Tagged Values ist nicht möglich.

Das Spezifikations-Fenster enthält mehrere Karteikarten, die die Eigenschaften thematisch gliedern. Die Karteikarten unterscheiden sich in Standard-Karteikarten und Karteikarten, die durch Erweiterungen hinzugefügt wurden. Während erstere an ihren Inhalt angepasst sind und damit ein komplexeres Layout enthalten (siehe Abbildung 5.2), sind letztere von einer einfachen Standard-Struktur (siehe Abbildung 5.3).

Eigenschaften haben einen Typ und einen Standardwert. Unterstützte Typen sind:

- String,
- Integer,
- Float,
- Char,
- Boolean und
- Enumeration.

Abbildung 5.3 zeigt das Layout für Karteikarten, die von Erweiterungen hinzugefügt wurden. Die Tabelle *Model Properties* enthält die Eigenschaften. Die erste Spalte (*) markiert aktuell vorgenommene Änderungen der Werte. Die Spalte *Name* enthält den

Namen der jeweiligen Eigenschaft. Die Spalte *Value* enthält den zugehörigen Wert. Bei Eigenschaften vom Typ Boolean oder Enumeration werden mögliche Werte mittels einer Auswahlliste vorgeschlagen. Bei anderen Typen ist der Wert frei einzugeben. Die Spalte *Source* enthält die Information, ob die Eigenschaft mit dem Standardwert belegt ist (Wert *Default*), oder ob der Standardwert durch den Anwender überschrieben wurde (Wert *Override*).

Oberhalb der Tabelle kann eine Menge von Standardwerten (*Set*) ausgewählt werden. Mengen von Standardwerten dienen dazu, dass der Anwender den Eigenschaften häufig benötigte Wertebelegungen zuweisen kann, ohne diese für jede Eigenschaft jedesmal einzeln angeben zu müssen. Es gibt zu jeder Karteikarte mindestens eine Menge *default* und darüber hinaus beliebige weitere Standardwerte-Mengen. Der Anwender gelangt über die Schaltfläche *Edit Set* zu einem Dialog, der das Hinzufügen eigener Standardwerte-Mengen erlaubt.

Standardwerte sind nur als Vorschläge anzusehen und mit *Default* markiert. Der Anwender kann entweder den Vorschlag übernehmen, indem er ihn mit *Override* markiert, oder er ändert den Wert ab, wodurch dieser automatisch als *Override* markiert wird. *Override* markiert also Werte, die explizit vom Anwender gesetzt oder übernommen wurden.

Ein Beispiel für die Verwendung einer Standardwerte-Menge wäre eine Menge für Klassen. Ein Add-In erweitert Rose um die Eigenschaften `generateCode` und `generateStandardConstructor` vom Typ `Boolean`, um damit dem Anwender Festlegungen zur Codegenerierung zu ermöglichen. Da standardmäßig vorgeschlagen werden soll, dass für eine Klasse Code und ein Standard-Konstruktor generiert werden, enthält die Standardwerte-Menge *default* folgende Einträge:

$$default = \{generateCode = true, generateStandardConstructor = true\}$$

Akteure (UML-Metaklasse *Actor*) werden in Rose dargestellt als Klassen, die mit einem Stereotypen `Actor` versehen sind. Da standardmäßig kein Code für einen Akteur generiert werden soll, wird eine zusätzliche Standardwerte-Menge speziell für Akteure bereitgestellt:

$$actor = \{generateCode = false, generateStandardConstructor = false\}.$$

Der Anwender kann nun für jede Klasse zwischen diesen beiden Mengen auswählen. Möchte er die Vorschläge aus einer Menge übernehmen, so markiert er die Eigenschaften und betätigt die Schaltfläche *Override*.

5.1.2 Das Rose Extensibility Interface

Rose bietet eine Schnittstelle zu seiner Erweiterung, das *Rose Extensibility Interface* (*REI*). Das REI kann als ein Metamodell für Rose betrachtet werden, das Pakete, Klassen, Attribute und Methoden (beschrieben in [Rat02d]) bereitstellt, durch die Rose definiert wird und kontrolliert werden kann. Abbildung 5.4 zeigt die Nutzung der Erweiterungsschnittstelle von Rational Rose (nach [Rat01b]). Rose besteht intern aus mehreren Komponenten, deren Schnittstellen für den Zugriff von außen vom REI gekapselt werden. Die Kern-Komponenten von Rose sind auf der linken Seite der Abbildung dargestellt. Die Komponente `Rational Rose Application` kapselt die grundlegende Anwendungsfunktionalität. Die Komponente `Diagrams` bietet Zugriff auf die Sichten und Diagramme in Rational Rose. Auf die in Rose bearbeiteten Modelle und

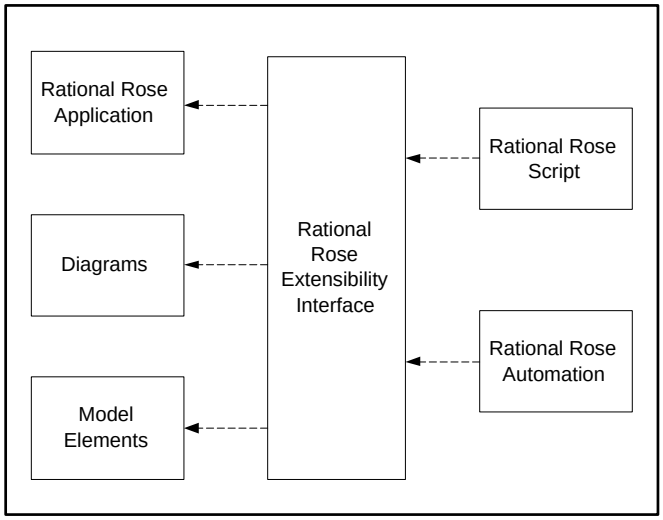

Abbildung 5.4: Die Erweiterungsschnittstelle von Rose (aus [Rat01b])

Modellelemente wird über die Komponente `Model Elements` zugegriffen. Für die Nutzung des REI durch den Anwender sind zwei Möglichkeiten vorgesehen: mittels *Rational Rose Script* und mittels *Rational Rose Automation*.

Ein Rose Script ist ein Skript in einer von Rose speziell dafür zur Verfügung gestellten Skriptsprache. Diese ist eine Erweiterung der Sprache *BasicScript* der Firma *Summit Software* [SUM02]. Sie ist objektorientiert und ihre Syntax ist der von Visual Basic ähnlich. Ein Editor zur Bearbeitung und Ausführung von Rose-Skripten ist in Rose integriert. Rose-Skripte sind zur Automatisierung von häufig benötigter Funktionalität in Rose vorgesehen und ermöglichen darüber hinaus Funktionalität, die über die graphische Benutzeroberfläche von Rose nicht erreichbar ist. Beispiel-Skripte werden mit Rose mitgeliefert.

Rose Automation dient der Verwendung von Rose als *OLE-Controller* oder *OLE-Server*. *OLE* steht für *Object Linking and Embedding* und bezeichnet einen Mechanismus des Betriebssystem *Windows* zur Kommunikation zwischen unabhängigen Programmen, der auf der Komponententechnologie *COM* basiert. Wird Rose als OLE-Controller verwendet, so können aus Rose-Skripten heraus andere OLE-fähige Anwendungen, z. B. die Textverarbeitung *Microsoft Word* oder die Entwicklungsumgebung *Microsoft Visual Studio*, aufgerufen und genutzt werden. Anwendungsbeispiele dafür wären z. B. die Nutzung von Word zur Erstellung und Anzeige von generierter Dokumentation oder die Nutzung der Entwicklungsumgebung Visual Studio zur Weiterverarbeitung von generiertem Code.

Die Verwendung von Rose als OLE-Server ermöglicht aus OLE-fähigen Anwendungen heraus die gesamte Funktionalität des REI zu nutzen. Ein Anwender kann somit in einer beliebigen OLE-fähigen Programmiersprache, wie z. B. Visual Basic oder Visual C++, Anwendungen zur Erweiterung von Rose erstellen. Der OLE-basierte Zugriff auf Rose bietet gegenüber dem Zugriff durch Rose-Skripte zusätzliche Möglichkeiten.

5.1.2.1 Add-Ins

Eine Menge von Erweiterungen von Rose für einen bestimmten Anwendungsbereich wird als *Add-In* bezeichnet. Ein Add-In erweitert Rose mittels Rose-Skripten und OLE. Zusätzlich beinhaltet es Konfigurationsdateien für Anpassungen von Rose und nimmt bei der Installation Einträge in die Windows-Registrierungsdatenbank (*Registry*) vor, um sich selbst und die einzelnen Erweiterungen bei Rose zu registrieren. Rational sieht für Add-Ins folgende Funktionalität in Rose vor:

- Hinzufügen von Menüpunkten in der Hauptmenüleiste,
- bedingtes Ein- und Ausblenden eigener Menüpunkte in Kontextmenüs zu Modellelementen,
- Anpassung von Eigenschaften (*Properties*) des Modells oder von Modellelementen,
- Hinzufügen von Datentypen zu der Auswahlliste in Spezifikations-Fenstern,
- Hinzufügen von Stereotypen,
- Bereitstellung von Online-Hilfe für Add-Ins,
- Bereitstellung von kontextsensitiver Hilfe für Add-Ins,
- Reaktion auf Ereignisse in Rose (bei Rose-Skripten nur eingeschränkt möglich),
- alle weitere über die graphische Oberfläche in Rose erreichbare Funktionalität und
- weitere beliebige eigene Funktionalität.

Bezüglich des letzten Punktes soll hervorgehoben werden, dass Add-Ins, zumindest wenn es sich um einen OLE-Server handelt, unabhängig vom REI beliebige weitere Funktionalität ausführen können, z. B. beliebige eigene Fenster anzeigen oder jegliche Arten von Dateien generieren oder nach beliebigen eigenen Kriterien eine Prüfung von aus einem Rose-Modell gelesenen Informationen durchführen. Einschränkungen bestehen dort, wo das REI genutzt wird. Diese Einschränkungen resultieren im wesentlichen daraus, dass

- auf bestimmte Eigenschaften im Rose-Metamodell nur lesender Zugriff möglich ist,
- einige Eigenschaften von Rose – wie z. B. die graphische Darstellung einer Klasse (abgesehen von Icons für Stereotypen) als Rechteck – nicht im Rose-Metamodell enthalten sind und somit überhaupt kein Zugriff darauf möglich ist und
- dass das Rose-Metamodell nicht verändert werden kann, also beispielsweise keine neuen Typen von Modellelementen oder Diagrammen erstellt werden können.

Einen strukturierten Überblick über die Erweiterungsmöglichkeiten und die damit verbundenen alternativen Abläufe gibt Abbildung 5.5. Die Möglichkeit, durch Rose-Skripte auf Ereignisse zu reagieren, ist nur für wenige Ereignisse gegeben (in der Abbildung durch eine gestrichelte Verbindung verdeutlicht). Erweiterungsmöglichkeiten,

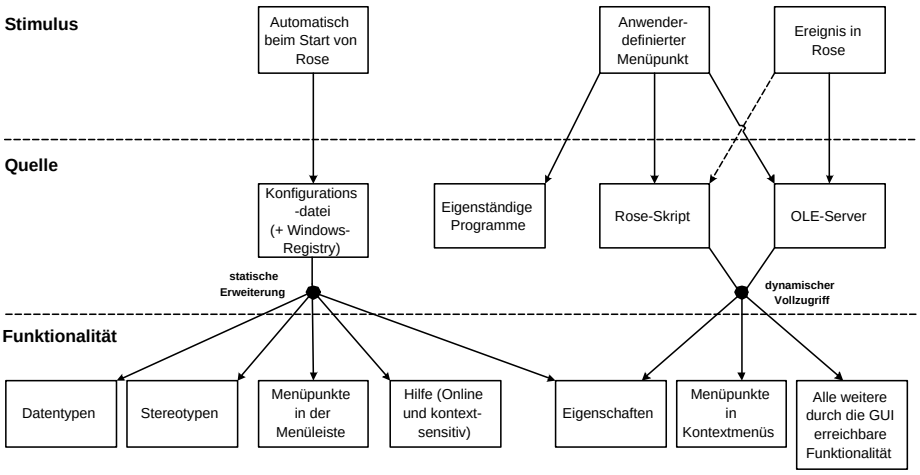

Abbildung 5.5: Abläufe bei Add-Ins (Prinzip).

die für diese Arbeit näher betrachtet werden müssen, werden in den nachfolgenden Abschnitten beschrieben.

Grundsätzlich wird zwischen zwei Arten von Add-Ins unterschieden. *Basic-Add-Ins* nehmen generell gültige Erweiterungen an Rose vor und rufen Rose-Skripte oder eigenständig ausführbare Anwendungen aus Rose heraus auf. Es können beliebig viele Basic-Add-Ins gleichzeitig aktiv sein.

Language-Add-Ins sind hauptsächlich dafür vorgesehen, Rose an eine bestimmte Programmiersprache oder Implementierungstechnologie anzupassen. Sie können die gleichen Möglichkeiten zur Erweiterung wie Basic-Add-Ins nutzen. Darüber hinaus können sie für die Generierung von Code verantwortlich sein und das Standard-Spezifikations-Fenster von Rose durch ein Spezifikations-Fenster mit beliebiger eigener Funktionalität und Oberfläche ersetzen.

Jedem Modellelement in Rose ist genau ein Language-Add-In zugeordnet, das verantwortlich ist, falls aus dem Modellelement Code erzeugt werden soll oder wenn das Spezifikations-Fenster zu dem Modellelement aufgerufen wird. Auch die meisten anderen Erweiterungen, die ein Language-Add-In bezüglich Modellelementen vornimmt, sind nur für Modellelemente verfügbar, denen das Language-Add-In zugeordnet ist.

Das Language-Add-In kann nicht für einzelne Modellelemente festgelegt werden, sondern nur für das gesamte Modell. Standardwert ist das Language-Add-In *Analysis*, welches keine Codegenerierung unterstützt und keine speziellen Spezifikations-Fenster anzeigt. Die Auswahl des Language-Add-Ins für das Modell kann zu jedem beliebigen Zeitpunkt geändert werden. Jedem Modellelement wird dasjenige Language-Add-In zugeordnet, das zum Zeitpunkt seiner Erstellung für das Modell ausgewählt ist. Die Zuordnung bleibt für alle bereits erstellten Modellelemente auch dann erhalten, wenn die Auswahl für das Modell geändert wird. Daher kann ein Modell auch Modellelemente enthalten, denen verschiedene Language-Add-Ins zugeordnet sind.

In Rose können Klassen und Schnittstellen einer Komponente zugeordnet werden. Für jede Komponente lässt sich ein Language-Add-In festlegen. Wie für das Modell kann auch für Komponenten die Auswahl des Language-Add-Ins zu jedem Zeitpunkt geändert werden. Alle Klassen und Schnittstellen, die einer Komponente zugeordnet

sind, übernehmen von dieser das Language-Add-In. Nur auf diese Weise kann die Zuordnung des Language-Add-Ins für ein Modellelement geändert werden. Da jedem Modellelement insgesamt nur ein Language-Add-In zugewiesen sein darf, können Modellelemente auch nur maximal einer Komponente zugeordnet werden.

Die Unterstützung von Implementierungstechnologien bzw. Programmiersprachen und somit die Generierung von Code ist nicht Teil der Grundfunktionalität in Rose, sondern wird ausschließlich über Language-Add-Ins realisiert. Deshalb wird Rose bereits mit einer Vielzahl von Language-Add-Ins ausgeliefert, z. B. für Java, C++ oder CORBA. Weiterhin sind auch Basic-Add-Ins im Lieferumfang enthalten, die grundlegende Erweiterungen der Funktionalität bereitstellen, wie z. B. ein Add-In zur Versionskontrolle (*Version Control*) oder zur Qualitätssicherung (*Quality Architect*).

Zur Verwaltung der Add-Ins enthält Rose den *Add-In-Manager*. Dieser erlaubt dem Anwender, Add-Ins zu aktivieren und zu deaktivieren und verwaltet die einzelnen Erweiterungen der aktivierten Add-Ins. Bei aktivierten Basic-Add-Ins sind in der Regel alle Erweiterungen aktiv. Aktivierte Language-Add-Ins können dem Modell und den Komponenten zugeordnet werden. Alle Erweiterungen aus deaktivierten Add-Ins sind nicht in Rose verfügbar.

5.1.2.2 Behandlung von Ereignissen

Add-Ins können sich bei Rose für die Benachrichtigung bei Ereignissen registrieren lassen. Je nach Art des Ereignisses ist dazu einer der beiden Schritte notwendig:

1. Registrierung über die Windows-Registrierdatenbank oder
2. Bereitstellung eines OLE-Servers mit einer entsprechenden Schnittstelle.

Bei Ereignissen, deren Behandlung einen Eintrag in die Windows-Registrierdatenbank verlangt, kann alternativ ein Rose-Skript oder ein COM-Server zur Behandlung des Ereignisses angegeben werden.

Welche der aktiven Add-Ins bei Eintritt eines Ereignisses benachrichtigt werden ist abhängig von der Art des Ereignisses. Beispielsweise gibt es Ereignisse, die nur ein Language-Add-In benachrichtigen. Detailierte Beschreibungen der Ereignisse befinden sich in [Rat02d].

Die verfügbaren Ereignisse resultieren aus einer Aktivität bezüglich

- Rose selbst: Starten der Anwendung,
- Add-Ins: Aktivieren oder Deaktivieren,
- Modellen: neu Erstellen, Öffnen, Speichern, Schliessen und Löschen,
- Modellelementen: neu Erstellen, Verändern und Löschen,
- Codegenerierung: Generieren und Navigieren sowie
- der Oberfläche: Öffnen eines Spezifikations-Fensters.

Das letztgenannte Ereignis bietet die Möglichkeit, die Spezifikations-Fenster in Rose anzupassen. Wird ein Spezifikations-Fenster geöffnet, so wird darüber nur das Add-In benachrichtigt, das dem jeweiligen Modellelement als Language-Add-In zugeordnet ist. Ist das Add-In nicht bei Rose zur Behandlung dieses Ereignisses registriert, so wird kein Add-In benachrichtigt und es wird das Standard-Spezifikations-Fenster

geöffnet. Bei Behandlung dieses Ereignisses kann durch den Rückgabewert der aufgerufenen Methode gesteuert werden, ob Rose die Standard-Anzeige des jeweiligen Spezifikations-Fensters vornehmen soll. Andernfalls kann das Add-In ein beliebiges eigenes Spezifikations-Fenster anzeigen.

5.1.2.3 Erweiterung um Stereotypen

Rose kann durch ein Add-In um Stereotypen erweitert werden. Dazu wird eine Konfigurationsdatei mit den Informationen der zusätzlichen Stereotypen erstellt und diese durch einen Eintrag in der Windows-Registry bei Rose registriert. Üblicherweise wird die Stereotyp-Datei in der Registry einem Add-In zugeordnet, wodurch die Stereotypen nur zur Verfügung stehen, falls das Add-In aktiviert ist. Ist das Add-In ein Language-Add-In, so sind die Stereotypen nur bei Modellelementen verfügbar, denen das Add-In als Language-Add-In zugeordnet ist. Ist ein Stereotyp einem Modellelement zugeordnet, so bleibt er erhalten, auch wenn das zugehörige Add-In deaktiviert wird.

In einer Stereotyp-Datei wird ein Stereotyp durch einen Namen und seine Base-Class definiert. Als Base-Class sind folgende Rose-Metaklassen möglich:

- Association,
- Attribute,
- Class,
- Component,
- Component package,
- Connection,
- Dependency,
- Device,
- Generalization,
- Logical package,
- Operation,
- Processor,
- Use case und
- Use-case package.

Optional können Icons definiert werden:

- Ein Icon zur Darstellung im Diagramm (für beide Darstellungsvarianten),
- ein Icon zur Darstellung im Explorer-Fenster und
- zwei Icons zur Darstellung in der Werkzeugleiste zur direkten Erzeugung von Instanzen der virtuellen Metaklasse.

Für die Werkzeugleiste sind zwei Icons anzugeben, da der Anwender zwischen einer Darstellung der Werkzeugleiste mit großen oder mit kleinen Icons wählen kann. Stereotypen, deren Base-Class nicht eigenständig in einem Diagramm vorhanden sein darf (wie z. B. Operationen oder Attribute), können nicht über die Werkzeugleiste instanziiert werden, weshalb auch keine solchen Icons definiert werden dürfen.

Stereotypen, die durch ein Add-In hinzugefügt wurden, können in Rose wie Standard-Stereotypen verwendet werden (siehe Abschnitt 5.1.1.2).

5.1.2.4 Erweiterung um Eigenschaften

Ein Add-In kann das Spezifikations-Fenster zu einem Modellelement um Karteikarten mit Eigenschaften erweitern. Der Add-In-Manager verwaltet die Karteikarten, die zu Add-Ins gehören. Es werden nur die Karteikarten von Add-Ins angezeigt, die für das jeweilige Modellelement aktiv sind (d. h. alle aktivierten Basic-Add-Ins, sowie das Language-Add-In, das dem Modellelement zugeordnet ist). Wird ein Wert einer Eigenschaft vom Anwender überschrieben, so bleibt dieser persistent dem jeweiligen Modellelement zugeordnet, auch wenn das Add-In nicht mehr für das Modellelement gültig ist.

Rose kann auf zwei alternativen Wegen um Eigenschaften von Modellelementen erweitert werden:

1. mittels Konfigurationsdateien oder
2. dynamisch mittels des REI aus Skripten oder Anwendungen heraus.

Durch eine Konfigurationsdatei können neue Karteikarten mit Eigenschaften zu einem Add-In erstellt werden. Dabei sind folgende Festlegungen zu treffen (vgl. Abschnitt 5.1.1.3):

- der Name der Karteikarte,
- die Namen von Mengen von Standardwerten, wobei mindestens eine Menge mit dem Namen *default* vorhanden sein muss,
- die Namen der neuen Eigenschaften,
- Typen der Eigenschaften (siehe oben),
- für jede Menge von Standardwerten ein Standardwert für jede Eigenschaft und
- den Namen der Metaklasse, deren Spezifikations-Fenster um die Karteikarte erweitert werden soll

Es können beliebig viele Karteikarten pro Metaklasse erstellt werden.

Gültige Rose-Metaklassen sind:

- Association
- Attribute
- Category
- Class
- Has

- Inherit
- Module-Spec
- Module-Body
- Operation
- Param
- Project
- Role
- Subsystem
- Uses

Durch Rose-Skripte oder mittels OLE kann auch über das REI aus Skripten bzw. Anwendungen heraus auf die Eigenschaften von Modellelementen zugegriffen werden. Es ist dabei der volle Zugriff auf alle Karteikarten, Eigenschaften und Werte möglich. Im Unterschied zu Konfigurationsdateien können änderungen auch dynamisch, d. h. bei laufendem Rose, durchgeführt werden. Änderungen gelten dabei jedoch nur für das aktuelle Modell.

Die Verwendung der Eigenschaften in Rose erfolgt wie in Abschnitt 5.1.1.3 beschrieben.

5.1.3 Vorhandene Erweiterungen

Rational bietet auf seinen Web-Seiten [Rat02c] verschiedene Add-Ins und Rose-Skripte an, die nicht im Lieferumfang enthalten sind und zusätzliche Funktionalität realisieren. Darunter befinden sich auch Erweiterungen, die im Kontext dieser Arbeit relevant sind und daher im folgenden vorgestellt werden sollen.

5.1.3.1 Unterstützung von XMI in Rose

Die Firma Unisys bietet kostenlos die *Unisys Rose XML Tools* [Uni02b] an. Dabei handelt es sich um mehrere Add-Ins, die im Wesentlichen XMI-Unterstützung für Rose realisieren. Kernfunktionalität ist der Export und der Import von Modellen von bzw. nach Rose mittels des Austauschformats XMI.

Aktuell ist die Version 1.3.4. Diese unterstützt UML 1.3 in XMI 1.0 und XMI 1.1. Neben der standardisierten DTD der OMG für UML 1.3 unterstützt das Add-In auch eine proprietäre DTD, die verwendet wird, um Diagramm-Informationen in die XMI-Datei einzubeziehen. Diese DTD wurde bereits in Abschnitt 3.3.2.2 beschrieben.

Das Add-In behandelt alle Modellelemente, die von Rose unterstützt werden. Referenzen auf Elemente in separaten XMI-Dateien werden nicht unterstützt.

Ist ein Modellelement mit einem Stereotyp versehen, so wird dieser exportiert. Dabei wird keine Unterscheidung gemacht zwischen Standard-Stereotypen, Stereotypen aus Add-Ins und Stereotypen, die vom Anwender manuell eingeben wurden. Beim Import wird jeder Stereotyp in die Auswahlliste der Stereotypen für die Metaklasse, zu der das Modellelement gehört, aufgenommen. Stereotypen und deren Attribut `baseClass` bleiben also erhalten.

Eigenschaften werden als Tagged Values exportiert, wobei nur Eigenschaften berücksichtigt werden, die vom Anwender mit einem Wert belegt wurden (markiert durch *Override*, siehe Abschnitt 5.1.1.3).

Damit ist das Add-In auch für den Export und Import von Modellen, die UML-Erweiterungsmechanismen verwenden, gut geeignet.

5.1.3.2 Der Rose-Profile-Documentor

Das Rose-Skript *Rose-Profile-Documentor* [Joh02] wurde von einem Mitarbeiter von Rational entwickelt und besteht aktuell in der Version 0.3.2. Es generiert Dokumentation in HTML aus einem UML-Profile, das in Rose gemäß der UML-Spezifikation 1.4 definiert wurde. Die generierte Dokumentation soll dem Stil für Vorschläge an die OMG (*OMG submissions*) entsprechen. Zusätzlich zu den Definitionen aus UML 1.4 kann bei der Erstellung eines Profiles durch einen Stereotypen `conceptual` ein Paket markiert werden, das Diagramme zur Darstellung des Metamodells enthält, auf dem das Profile basiert.

Weitere Dokumentation ist nicht verfügbar. Da das Skript als Quellcode vorliegt, kann die im Folgenden beschriebene Funktionalität erschlossen werden: Zur Erstellung des Profiles werden Klassendiagramme verwendet. Das Top-Level-Paket muss mit dem Stereotypen `profile` versehen sein. Stereotypen müssen gemäß der graphischen Notation in UML 1.4 definiert werden. Dazu werden Klassen verwendet, die mit dem Stereotypen `stereotype` versehen sind. Enthalten diese ein Attribut mit dem Namen `icon`, so wird dessen Wert als Name einer Graphik-Datei interpretiert, die ein Icon zur Repräsentation des Stereotyen enthält. Der Inhalt des Dokumentations-Fensters zum Attribut `icon` wird als Beschreibung des Icons interpretiert.

Die Base-Class von Stereotypen wird definiert über eine Abhängigkeit mit dem Stereotype `stereotype` zu einer Klasse mit einem Stereotypen `metaclass`. Deren Klassenname muss der Name der Base-Class sein. Generalisierung von Stereotypen sowie die Angabe von abstrakten Stereotypen ist möglich.

Die Attribute und die Assoziationen von Klassen mit dem Stereotypen `stereotype` werden als Tagged Values interpretiert, falls sie mit dem Stereotypen `taggedValue` versehen sind. Zur Angabe von Constraints wird das Dokumentations-Fenster verwendet.

Aus der Definition des Profiles wird Dokumentation in HTML generiert. Zusätzlich dazu wird auf Wunsch des Anwenders ein Add-In mit einem vom Anwender zu wählenden Namen erstellt. Dieses besteht aus Konfigurationsdateien für die Stereotypen und für Eigenschaften zur Umsetzung der Tagged Values. Die Konfigurationsdatei für Stereotypen enthält den Namen der Stereotypen, ihre Base-Class und etwaige angegebene Icons. Die Konfigurationsdatei für Eigenschaften enthält den Namen des Add-Ins als Namen für die Karteikarte, den Namen des zugehörigen Stereotypen als Name der Menge von Standardwerten und die Namen von Tagged Values als Name von Eigenschaften. Alle Eigenschaften sind vom Typ String und haben einen leeren String als Standardwert. Die Eigenschaften werden der Base-Class des Stereotypen zugeordnet. Jede Standardwerte-Menge trägt den Namen eines Stereotypen und enthält nur die Eigenschaften, deren korrespondierende Tagged Values im Profile diesem Stereotypen zugeordnet sind. Dadurch werden die Eigenschaften an Stereotypen gebunden (siehe Abschnitt 5.3.3.1).

Anstatt ein Profile zu definieren, kann der Anwender auch ein Metamodell erstellen, indem beim Top-Level-Paket der Stereotyp `profile` durch den Stereotypen `metamodel` ersetzt wird. Entsprechend wird die weitere Interpretation angepasst.

Wie aber bereits aus der Versionsnummer ersichtlich, ist das Rose-Skript noch keine umfassende Lösung. Die Lösung erscheint zunächst sehr einfach und funktionell. Jedoch muss beachtet werden, dass vom Anwender in der Praxis sehr viel Wissen über das REI verlangt wird, wenn er mit diesem Rose-Skript tatsächlich ein Add-In erzeugen will. So muss er z. B. die Rose-internen Namen von Metaklassen kennen, damit diese korrekt in die generierten Konfigurationsdateien geschrieben werden. Datentypen und Modellbibliotheken berücksichtigt das Rose-Skript nicht. Auch wenn die praktische Anwendbarkeit nur eingeschränkt gegeben ist, sind dennoch die enthaltenen Konzepte beachtenswert.

5.1.3.3 Das XMI-Toolkit

Das XMI-Toolkit ist keine Erweiterung von Rose im eigentlichen Sinne, da es weder das REI nutzt, noch seine Funktionalität auf die Anwendung zusammen mit Rose beschränkt ist. Es bietet die Möglichkeit, aus einem Rose-Modell mit Zwischenschritten über XMI Java-Code zu generieren. Das Modell und der Code können wechselseitig synchronisiert werden. Darüber hinaus bietet es unabhängig von Rose ein Framework zur Erzeugung und Behandlung von XMI in Java. Da es als Grundlage für die folgenden Konzepte dient, soll es ebenfalls (als Erweiterung von Rose im weitesten Sinne) in diesem Kapitel vorgestellt werden.

Das XMI-Toolkit [IBM02b] wird von IBM [IBM02a] entwickelt und darf kostenlos verwendet werden. Es besteht aus einem Java-Framework zum Umgang mit XMI-Dateien und Werkzeugen, die dieses nutzen. Es unterstützt XMI 1.0 und XMI 1.1.

Das XMI-Toolkit unterstützt Rational Rose. Das Hauptwerkzeug generiert Java-Code aus einem Rose-Modell. Auch die umgekehrte Abbildung ist möglich. Als Zwischenergebnisse entstehen jeweils XMI-Repräsentationen des Rose-Modells und des Java-Codes. Zusätzlich kann das Werkzeug Rose-Modelle und zugehörigen Java-Code verwalten und bei Änderungen automatisch synchonisieren (*Round-Trip-Engineering)*. Weiterhin kann auch aus dem Rose-Modell eine DTD erzeugt werden.

Das Framework bietet weitaus mehr Funktionalität als die beigefügten Werkzeuge. Während die Werkzeuge nur XMI 1.0 unterstützen, unterstützt das Framework auch XMI 1.1. Es enthält Klassen zur Repräsentation der Bestandteile einer XMI-Datei sowie der XMI-Dateien selbst. Zusätzliche Klassen unterstützen den Umgang mit XMI.

Bei der Generierung von Java-Klassen aus einem Rose-Modell kann (unter Nutzung des Frameworks mittels einer eigenen Java-Anwendung) zusätzlich eine Klasse `UserFactory` generiert werden. Diese enthält Informationen über die generierten Java-Klassen und erlaubt dadurch einen besonders einfachen Umgang mit XMI-Dateien: Instanzen dieser Klassen können dann automatisch durch das XMI-Toolkit im XMI-Format gespeichert werden. Ebenso können aus einer entsprechenden XMI-Datei durch das XMI-Toolkit automatisch Instanzen der Klassen erstellt werden. Der Anwender kann daher XMI-Dateien schreiben und lesen, ohne selbst mit XMI in Berührung zu kommen. Diese komfotable Funktionalität soll auch in dieser Arbeit für den Umgang mit XMI genutzt werden.

Eine Einschränkung beim XMI-Toolkit besteht jedoch in der Behandlung von Referenzen auf Elemente in anderen XMI-Dateien. Die dazu vorgesehenen Mechanismen, XLink und XPointer, werden vom XMI-Toolkit nicht unterstützt. Stattdessen verwendet das XMI-Toolkit für Referenzen in andere Dateien das XMI-Attribut `xmi.uuid`. Eine Möglichkeit, diese Einschränkung zu umgehen, wäre beispielsweise die Erstellung eines einfachen XSLT-Skripts [Wor99], das vor jedem Einlesen von XMI-Dateien

durch das XMI-Toolkit etwaige XLinks und XPointer auflöst und aus mehreren XMI-Dateien eine gemeinsame XMI-Datei erstellt.

5.2 Erstellung von Profiles

Gemäß Abschnitt 4.2 ist für die Erstellung von Profiles kaum Funktionalität aus bestehenden Werkzeugen notwendig. Der Hauptteil der Architektur kann daher unabhängig von Einschränkungen durch das REI entworfen werden und wird im folgenden Abschnitt vorgestellt. Anschließend werden Punkte untersucht, für die eine Integration in Rose denkbar ist.

5.2.1 Architektur

Die eigenständige Anwendung zur Erstellung von Profiles, im folgenden als *Profile-Builder* bezeichnet, muss die Anforderungen aus 4.1.1 erfüllen. Der Anwender soll ein Profile erstellen und bearbeiten können. Dieses soll als XMI-Datei weitergegeben werden können.

Aufgrund der Unabhängigkeit vom REI kann der Entwurf der Architektur unbeschränkt auf eigene Vorgaben und Ziele ausgerichtet werden. Ein wichtiges Ziel ist die möglichst hohe Wiederverwendung bestehender Implementierungen. Dies bietet sich besonders für den Umgang mit XMI (bzw. XML im Allgemeinen) an, da es sich hierbei um eine gängige Problemstellung handelt, für die bereits einige Implementierungen vorhanden sind. Hohe Wiederverwendung ist erstrebenswert, um eine möglichst einfache und effiziente Realisierung der Konzepte zu ermöglichen, damit die vorgestellten Konzepte nicht nur von theoretischer Relevanz bleiben. Weiterhin gewährleistet eine Wiederverwendung bereits erprobter Komponenten, dass diese Teile des Konzepts tatsächlich vollständig realisierbar und auch benutzbar sind.

Ein zweites wichtiges Ziel der Architektur ist eine möglichst einfache Anpassbarkeit an zukünftige Versionen der UML. Da in naher Zukunft eine weitere Versionen der UML (Version 2.0) zu erwarten ist, muss dies berücksichtigt werden, um eine längerfristige Nutzbarkeit der vorgestellten Konzepte zu sichern. Weiterhin ist zu erwarten, dass durch die Einbeziehung von Anpassungsmöglichkeiten von Anfang an auch eine hohe allgemeine Wartbarkeit begünstigt wird.

Ausgangspunkt für die Architektur des Profile-Builders ist die Grundarchitektur für Werkzeuge nach Balzert [Bal98]. Für den Umgang mit XMI bietet sich die Wiederverwendung des in Abschnitt 5.1.3.3 beschriebenen XMI-Toolkits an. Dadurch ist die Anwendung auf die plattformunabhängige Programmiersprache Java festgelegt. Aus Gründen der Einfachheit realisiert das XMI-Toolkit durch das Importieren und Exportieren der Daten von bzw. nach XMI die gesamte Datenhaltungsschicht. Eine Erweiterung um ein Repository zur Speicherung des Modells wäre jederzeit möglich.

Das Modell der Anwendung (Subsystem `Model`) entspricht der Untermenge des UML-Metamodells, die zur Erstellung von Profiles notwendig ist (siehe Abschnitt 4.1.1.1). Zur Erstellung der notwendigen Java-Klassen kann das XMI-Toolkit verwendet werden. Dieses generiert aus einem Rose-Modell entsprechende Java-Klassen. Daher muss zur Anpassung des Modells an zukünftige Versionen der UML an dieser Stelle lediglich ein neues Rose-Modell erstellt werden.

Um ein Profile zu erstellen, werden Instanzen der Modell-Klassen erzeugt, die mit Hilfe des XMI-Toolkits direkt in einer XMI-Datei gespeichert werden können. Auch das Einlesen von XMI-Dateien zur weiteren Bearbeitung von Profiles wird durch das

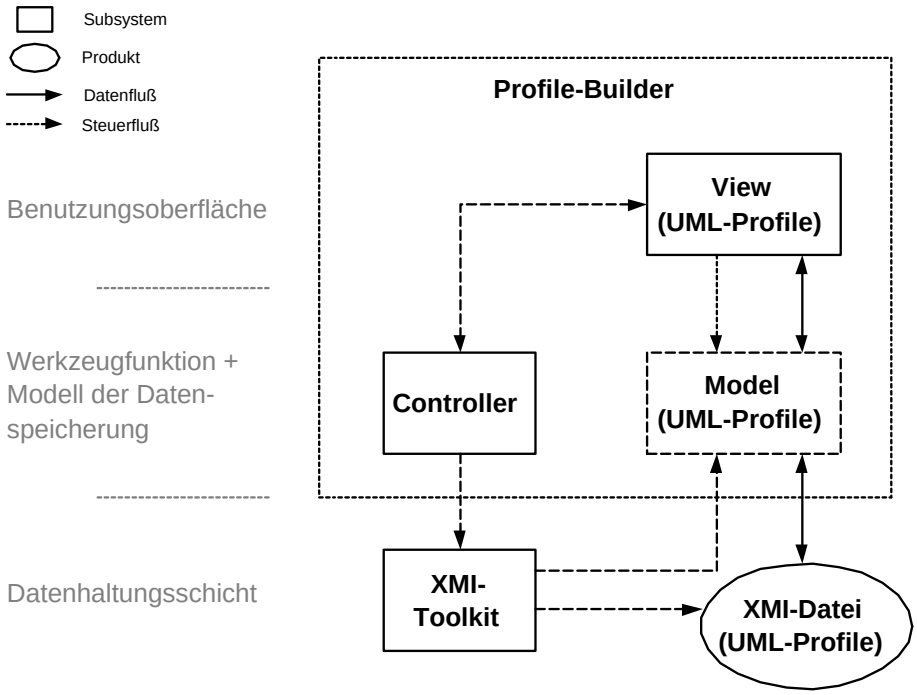

Abbildung 5.6: Die Architektur des Profile-Builders in der Struktursicht.

XMI-Toolkit unterstützt, indem aus XMI-Elementen automatisch Instanzen des Modells erzeugt werden. Dadurch werden mittels des XMI-Toolkits alle Anforderungen an die Persistenzschicht aus Abschnitt 4.1.1.5 erfüllt.

Die Benutzeroberfläche muss die Anforderungen aus Abschnitt 4.1.1.4 erfüllen. Sie bietet eine Sicht (Subsystem `View`) auf das Modell. Dabei wird hier von einer nichtgraphischen Darstellung der Profiles ausgegangen. Für jede Klasse des Modells, die nicht abstrakt ist, muss mindestens eine Sicht vorhanden sein. Diese stellt Instanzen der jeweiligen Modell-Klasse dar und erlaubt die Bearbeitung ihrer Eigenschaften. Die Anforderungen bezüglich Werkzeug-Prüfung und -Unterstützung aus Abschnitt 4.1.1.2 soll dabei aus Gründen der Einfachheit innerhalb der Sichten umgesetzt werden.

Falls nur einfache Sichten auf das Metamodell unterstützt werden sollen, könnten diese auch automatisch aus dem Modell erzeugt werden. Dafür gibt es Maskengeneratoren, wie z. B. *Janus* der Firma *Otris* [OTR02]. Für die Werkzeug-Prüfung und -Unterstützung können z. B. in Janus Regeln bezüglich der erlaubten Eingaben definiert werden. Am Ende dieses Abschnitts wird hierauf noch näher eingegangen. Wird ein Maskengenerator zur Erstellung der Sichten verwendet, so ist der gesamte Profile-Builder ohne Programmieraufwand an künftige UML-Versionen anpassbar.

Da die einzelnen Schichten des Profile-Builders generiert bzw. wiederverwendet werden, ist zusätzlich ein Subsystem zur Integration der einzelnen Schichten und zur Steuerung des Gesamtablaufs notwendig (Subsystem `Controller`). Dieses ruft die benötigten Klassen aus dem XMI-Toolkit auf und erstellt neue Instanzen der Modell-Klassen und der Sichten.

Abbildung 5.6 zeigt die Struktur der Architektur. Die neu zu implementierenden

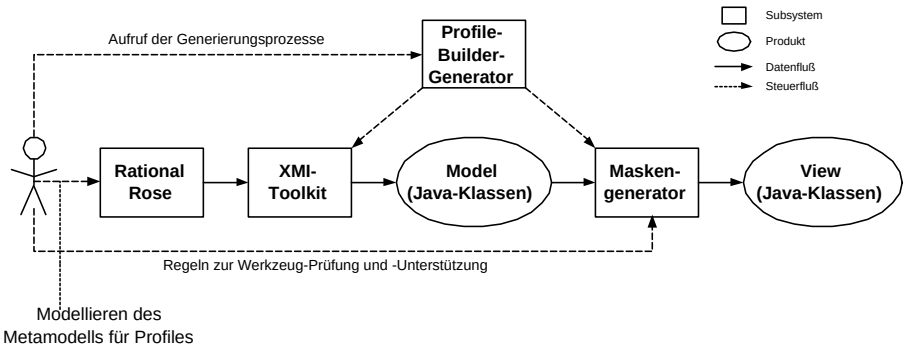

Abbildung 5.7: Ablauf einer Anpassung des Profile-Builder an ein neues UML-Metamodell.

bzw. zu generierenden Subsysteme des Profile-Builders entprechen dem Architekturmuster *Model-View-Controller*. Die Gesamtarchitektur unter Einbeziehung des XMI-Toolkits als Persistenzschicht kann als Ausprägung der Grundarchitektur für Werkzeuge nach Balzert betrachtet werden (in der Abbildung 5.6 in grauer Schrift beschrieben). Daraus ist bereits ersichtlich, dass ein Werkzeug zur Erstellung von Profiles keiner komplexen oder spezifischen Architektur bedarf. Alle Profile-spezifischen Eigenschaften des Werkzeuges sind im konkreten Modell und in den konkreten Regeln zur Werkzeug-Prüfung und -Unterstützung enthalten. Theoretisch kann das Konzept auch für andere Metamodelle als das zur Profile-Erstellung verwendet werden.

Die Anpassbarkeit des Profile-Builders an zukünftige UML-Versionen durch Generierung aller Bestandteile, die vom Metamodell abhängig sind – d. h. Model und View – soll im Folgenden näher betrachtet werden. Abbildung 5.7 zeigt den notwendigen Ablauf einer Anpassung. Der Gesamtablauf wird durch einen `ProfileBuilderGenerator` gesteuert. Dieser tätigt die nötigen Aufrufe an das XMI-Toolkit und den Maskengenerator und unterstützt den Anwender bezüglich des Gesamtprozesses.

Der Anwender muss als Ausgangspunkt das zu unterstützende Metamodell in Rose erstellen. Daraus werden mit dem XMI-Toolkit Java-Klassen erzeugt, die das Modell repräsentieren. Falls gewünscht, kann dabei auch zusätzlich eine DTD zu diesem Metamodell erzeugt werden, mit der erstellte Profile-XMI-Dateien verifiziert werden können.

Als nächster Schritt sind die zugehörigen Sichten zu erzeugen. Um diesen Schritt ebenfalls weitmöglichst zu automatisieren, ist ein Maskengenerator notwendig. Für jede Metaklasse wird eine Sicht erzeugt, die für jedes Attribut und jede Assoziation ein Eingabefeld enthält. Die Art des Eingabefeldes ist abhängig vom Typ der Eigenschaft und von deren Multiplizität. Ein Großteil der Anforderungen bezüglich Werkzeug-Prüfung und -Unterstützung kann dabei automatisch durch den Maskengenerator berücksichtigt werden, z. B. die Überprüfung des Typs bei der Eingabe von Werten für Attribute oder die Erstellung von Auswahllisten für Attribute mit begrenztem Wertebereich.

Regeln zur Werkzeug-Prüfung und -Unterstützung, die nicht aus der abstrakten Syntax des Metamodells hervorgehen, müssen dem Maskengenerator durch den Anwender mitgeteilt werden. Dies sind vor allem zusätzlich benötigte Well-formedness

Rules aus dem UML-Metamodell, z. B. die Regel, dass innerhalb eines Namensraumes alle Modellelemente einen eindeutigen Namen tragen müssen. Auch wenn bestimmte Attribute oder Metaklassen nicht angezeigt werden oder nicht editierbar sein sollen, muss dies durch den Anwender festgelegt werden. Der Maskengenerator Janus unterstützt die Eingabe solcher Regeln und Festlegungen. Im ungünstigsten Fall muss der Anwender die Sichten manuell nachbearbeiten. Bei manueller Festlegung der Regeln zur Werkzeug-Prüfung und -Unterstützung können diese auch statt den Sichten den generierten Modell-Klassen hinzugefügt werden (z. B. durch Einfügen von Code zu den Getter- und Setter-Methoden).

Zusätzlich zu den Sichten werden auch Steuerelemente benötigt, um zwischen den Sichten zu navigieren und Instanzen von Modell-Klassen zu erstellen und zu löschen. Bei der Erstellung neuer Instanzen müssen die Kompositionen im Metamodell beachtet werden. Mittels einer Komposition ist dort beispielsweise festgelegt, dass ein Tagged Values stets Teil eines Stereotypen ist. Deshalb darf im Profile-Builder eine neue Instanz eines Tagged Values ebenfalls nur als Teil eines Stereotypen erstellt werden. Ausgangspunkt bei Erstellung eines neuen Profiles muss das Top-Level-Element der Kompositions-Hierarchie – die Metaklasse für das Profile selbst – sein.

Ist kein geeigneter Maskengenerator verfügbar, müssen die Sichten von Hand erstellt werden, was ebenfalls mit vertretbarem Aufwand realisierbar ist. Das Gesamtkonzept wird dadurch nicht beeinträchtigt. Denkbar wäre auch ein Kompromiß: Es wird ein einfacher Maskengenerator implementiert, der die Anforderungen zur Werkzeug-Prüfung und -Unterstützung, die automatisch aus dem Metamodell abgeleitet werden können, umsetzt. Zusätzliche Regeln fügt der Anwender von Hand in die generierten Modell-Klassen ein.

5.2.2 Integration in Rose

Zunächst wird eine mögliche Integration des Profile-Builders in Rose betrachtet. Anschließend wird, als eine Teilaufgabe der Erstellung eines Profiles, die Erstellung von Modellbibliotheken mittels Rose untersucht.

5.2.2.1 Integration der Erstellung von Profiles

Die Erstellung von Profiles kann durch eine eigenständige Anwendung realisiert werden. Bei deren Installation kann Rose um einen Menüpunkt erweitert werden, durch den diese aufgerufen wird.

Falls die optionale Anforderung nach einer graphischen Definition von Profiles in Diagrammen (Anforderung A.1.16) erfüllt werden soll, wäre dazu ein Klassendiagramm in Rose zu verwenden. Wie beim Profile-Documentor (siehe Abschnitt 5.1.3.2) müssten Konventionen getroffen werden, wie die Elemente im Klassendiagramm als Profile-Bestandteile interpretiert werden. Die erstellten Modellelemente sind dann zu durchlaufen und entsprechend den Konventionen wird ein Modell eines Profiles daraus erstellt. Rose übernimmt demnach die Rolle der *View* im Profile-Builder.

Die Erweiterung von Rose um einen eigenständigen Diagrammtyp bzw. neuer Diagrammelemente zur Definition von Profiles ist nicht möglich.

5.2.2.2 Erstellung von Modellbibliotheken

Gemäß Abschnitt 4.2.2 ist keine zusätzliche Funktionalität notwendig, um eine Modellbibliothek in Rose zu erstellen. Ein Stereotyp `ModelLibrary` zur Kennzeichnung

von Modellbibliotheken kann notfalls von Hand eingegeben werden.

Probleme treten jedoch beim Export nach XMI auf, da durch das XMI-Add-In von Unisys (siehe Abschnitt 5.1.3.1) momentan nur XMI basierend auf der UML 1.3-DTD erzeugt werden kann. Da diese Arbeit auf UML 1.4 basiert, muss zum Speichern (und ebenso zum Einlesen) von Modellbibliotheken in XMI eine eigene Erweiterung implementiert werden. Diese würde sich vom Add-In von Unisys insofern unterscheiden, als alle Rose-Metaklassen auf Metaklassen aus UML 1.4 statt UML 1.3 abgebildet werden. Bezüglich des REI wird keine zusätzliche Funktionalität benötigt, weshalb die Machbarkeit einer solchen Erweiterung durch das Add-In von Unisys bereits nachgewiesen ist.

Alternativ kann auch versucht werden – z. B. mittels XSLT – XMI-Dateien von UML 1.3 nach UML 1.4 zu transformieren. Speziell wenn der Inhalt von Modellbibliotheken auf Klassen und Datentypen (sowie in deren Kontext benötigte Modellelemente) beschränkt wird, sollte dies mit vertretbarem Aufwand realisierbar sein. Klassen und Datentypen selbst unterscheiden sich in UML 1.3 und UML 1.4 nicht. Es muss aber berücksichtigt werden, dass die exportierte XMI-Datei weitere Modellelemente enthält. So wird beispielsweise durch das Add-In von Unisys grundsätzlich die Eigenschaft `persistence` aus Rose als Tagged Value in die XMI-Datei einbezogen. Tagged Values in UML 1.4 unterscheiden sich grundlegend von Tagged Values in UML 1.3. Eine einfache Lösung für dieses Problem wäre, die Tagged Values in der XMI-Datei zu löschen, da sie nicht zwingend notwendig sind. Das Add-In von Unisys liest prinzipiell auch XMI-Dateien nach UML 1.4 ein. Enthalten diese aber Modellelemente, die von Rose nicht unterstützt werden, so können dabei Fehler auftreten.

Einschränkungen für die Erstellung von Modellbibliotheken mittels Rose ergeben sich allerdings dann, wenn Metaklassen modelliert werden sollen, die von Rose nicht unterstützt werden. Da Rose nicht um zusätzliche Metaklassen erweitert werden kann, bleibt dieser Punkt offen. Es ist aber davon auszugehen, dass in der Praxis Modellbibliotheken in fast allen Fällen ausschliesslich gängige Elemente beinhalten, die auch von Rose unterstützt werden.

5.3 Anwendung von Profiles

Bereits in Abschnitt 4.2.3 wurde festgestellt, dass die Erweiterung bestehender Werkzeuge um Funktionalität zur Anwendung von Profiles stark abhängig von der Erweiterungsschnittstelle des jeweiligen Werkzeuges ist. Daher wird Abschnitt 5.3.1 zuerst untersucht, welche Integrationsmöglichkeiten verwendet werden sollen. Auf dieser Basis wird in Abschnitt 5.3.2 eine Architektur für die Erweiterung entwickelt. Offene Anforderungen, die zusätzlicher dynamischer Unterstützung bedürfen, werden anschließend in Abschnitt 5.3.3 behandelt.

5.3.1 Integration in Rose

In diesem Abschnitt wird untersucht, auf welchem Weg die benötigte Funktionalität am günstigsten in Rose integriert werden kann. Dazu wird zunächst ein Basisgerüst festgelegt. Anschließend wird beschrieben, auf welche Weise dieses Basisgerüst um zusätzliche Funktionalität – für offene Punkte oder zukünftige Erweiterungen der Funktionalität – erweitert werden kann.

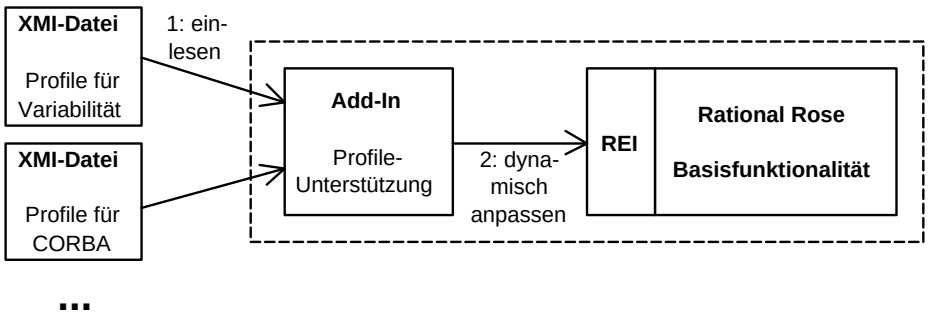

Abbildung 5.8: Ein Add-In zur Unterstützung von UML-Profiles in Rational Rose

5.3.1.1 Basisgerüst

Gemäß Abschnitt 4.2 ist zunächst nur die Anwendung des Profiles zu betrachten. Die in einem in XMI vorliegenden Profile enthaltenen Bestandteile sollen in Rose verfügbar gemacht werden. Es ist ein Add-In zu erstellen, das die vorliegenden Profiles einliest und entsprechende Anpassungen an Rose vornimmt (siehe Abbildung 5.8).

Die Möglichkeit zur Anpassung von Stereotypen ist in Abschnitt 5.1.2.3 beschrieben. Zur Unterstützung von Tagged Values müssen die Eigenschaften von Modellelementen benutzt werden. Die Erweiterung von Eigenschaften wurde in Abschnitt 5.1.2.4 vorgestellt. Benutzerdefinierte Datentypen für Tagged Values können dabei durch Verwendung von benutzerdefinierten Aufzählungen als Typen von Eigenschaften realisiert werden.

Bei der Nutzung der genannten Erweiterungsmöglichkeiten treten grundsätzliche Probleme auf. Diese beruhen darauf, dass die Erweiterung von Stereotypen und teilweise auch von Eigenschaften nur „statisch“ vorgesehen ist, d. h. diese Erweiterungen sind einmalig während der Installation des Add-Ins bei Rose zu registrieren und stehen erst nach dem nächsten Neustart von Rose zur Verfügung. Zumindest bezüglich Stereotypen gibt es keine Möglichkeit zur dynamischen Erweiterung.

Ein weiteres Problem besteht darin, dass bei einer Lösung gemäß Abbildung 5.8 nicht die Funktionalität des Add-In-Managers (siehe Abschnitt 5.1.2) genutzt werden kann. Dieser könnte lediglich das Add-In *Profile-Unterstützung* selbst verwalten, nicht aber die einzelnen Profiles, da diese nicht als eigenständige Add-Ins registriert sind. Demnach müsste das Add-In zur Profile-Unterstützung selbst einen Mechanismus bereitstellen, um Profiles aktivieren und deaktivieren zu können und um die Erweiterungen der einzelnen Profiles zu verwalten.

Aus diesen Gründen wird eine andere Lösungsalternative vorgeschlagen: es soll aus jedem Profile ein eigenständiges Add-In generiert werden (siehe Abbildung 5.9). Dies sollten Basic-Add-Ins sein, da sie keine Codegenerierung unterstützen. Die Generierung der Add-Ins kann durch eine unabhängige Anwendung vorgenommen werden. Für jedes Profile, das in Rose unterstützt werden soll, ist die Generierung eines Add-Ins einmalig vorzunehmen. Die generierten Add-Ins können bei beliebig vielen Installationen von Rose verwendet werden. Dort sind sie, wie andere Add-Ins auch, einmalig zu installieren.

Da die generierten Add-Ins wie von Rational vorgesehen bei Rose installiert werden, sind auch die Erweiterungsmechanismen von Rose wie von Rational vorgesehen

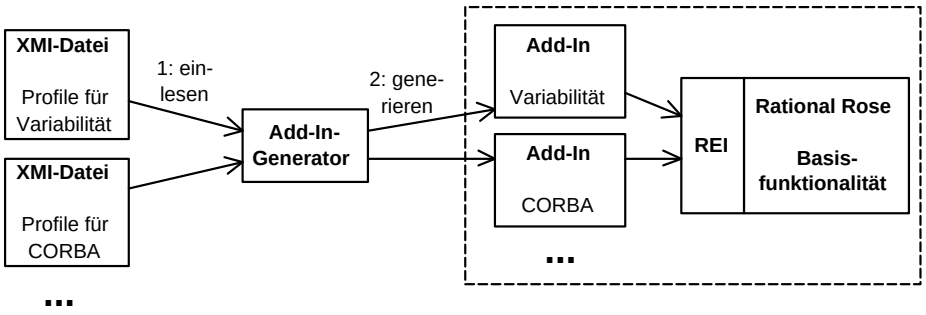

Abbildung 5.9: Verbesserte Integration einer Profile-Unterstützung in Rational Rose durch Generierung eines Add-Ins je Profile.

verwendbar. Die Erweiterung von Stereotypen und Eigenschaften wird statisch bei der Installation des Add-Ins vorgenommen. Auswahl und Deaktivierung von Profiles und die Verwaltung derer Bestandteile erfolgt durch den Add-In-Manager. Insgesamt werden die Profiles durch die Generierung von Add-Ins viel besser in Rose integriert als bei einer Lösung gemäß Abbildung 5.8.

Ein weiterer wichtiger Vorteil der generierten Add-Ins ist ihre flexible Erweiterbarkeit. Da sie als normale Add-Ins vorliegen, kann ihnen über die Funktionalität von UML-Profiles hinaus jegliche weitere Funktionalität, die von Rose unterstützt wird, hinzugefügt werden, z. B. die Generierung von Code und spezifische Funktionalität für den jeweiligen Anwendungsbereich. Bei einer Lösung gemäß Abbildung 5.8 wäre das nicht möglich, da dort Rose nur um die Funktionalität erweitert werden kann, die einerseits vom Add-In *Profile-Unterstützung* unterstützt wird und andererseits keine Informationen zusätzlich zur XMI-Repräsentation des Profiles benötigt.

Insgesamt bietet die Lösung der Generierung von Add-Ins aus vorliegenden Profiles deutliche Vorteile und wird daher als Basisgerüst für die weitere Arbeit gewählt.

5.3.1.2 Erweiterung des Basisgerüsts um dynamische Funktionalität

Im vorangegangenen Abschnitt wird das Basisgerüst festgelegt. Dadurch kann der von Rational vorgesehene Mechanismus zur Verwaltung von Add-Ins durch den Add-In-Manager verwendet werden, und sind grundlegende statische Erweiterungen möglich. Zusätzlich muss jedoch auch die dynamische Funktionalität der REI genutzt werden können: zum einen erfüllt die von Rational vorgesehene statische Erweiterung von Stereotypen und Eigenschaften noch nicht alle Anforderungen an Profile-Unterstützung aus Abschnitt 4.1. Offene Punkte werden im folgenden Abschnitt 5.3.3 diskutiert und bedürfen zur Lösung der dynamischen Funktionalität des REI. Zum anderen soll die Architektur auch für zukünftige Anforderungen erweiterbar sein und deshalb eine Möglichkeit der Nutzung dynamischer Funktionalität des REI vorsehen.

Zur Integration dynamischer Funktionalität in die Architektur des Basisgerüsts bestehen zwei Alternativen:

1. die Integration von dynamischer Funktionalität in die generierten Add-Ins selbst und

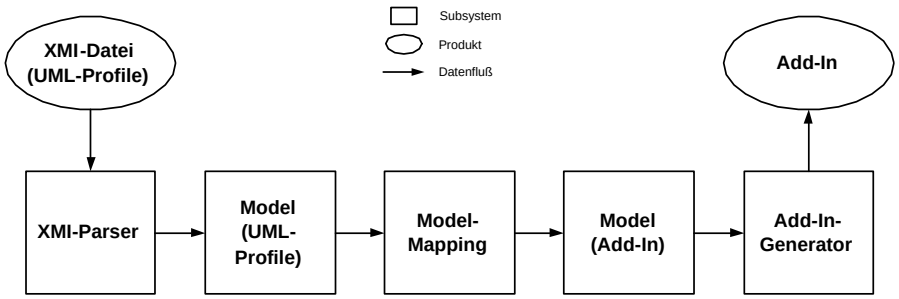

Abbildung 5.10: Die Grobstruktur des Add-In-Generators als Ausgangspunkt der Überlegungen.

2. die Erstellung eines zusätzlichen, separaten Add-Ins zur dynamischen Unterstützung von Profiles.

Die Integration dynamischer Funktionalität in jedes generierte Profile-Add-In bietet den Vorteil, dass diese eigenständig nutzbar sind. Bei einer Weitergabe eines Profiles würde der Empfänger kein zusätzliches Add-In für Profile-Unterstützung benötigen. Ein weiterer Vorteil ist der einfache Zugang zu Information aus dem Profile, falls solche für die dynamische Unterstützung benötigt wird.

Vorteile eines zusätzlichen, separaten Add-Ins wäre die bessere Wartbarkeit, da die gesamte dynamische Funktionalität redundanzfrei und zentral darin gekapselt werden könnte. Verbesserungen oder Änderungen an diesem Add-In würden sich auf alle Profiles gleichermaßen auswirken. Dadurch, dass es nur einmalig erstellt und installiert werden muss, wäre auch die gesamte Handhabung vereinfacht. Auch bei der Anwendung können sich Performance-Vorteile ergeben, indem die Funktionalität durch ein einziges Add-In erledigt wird. Dies ist insbesondere dann der Fall, wenn in Rose ein Modell nach bestimmten Modellelementen oder Eigenschaften durchsucht werden muss. Einem zentralen Add-In können benötigte Informationen aus einem Profile mittels der Windows-Registry zugänglich gemacht werden.

Insgesamt bieten beide Lösungen Vorteile. Die zweite Alternative scheint jedoch insgesamt einfacher handhabbar und wird daher in den meisten Fällen bevorzugt werden. Insgesamt wird deutlich, dass das Basisgerüst aus Abschnitt 5.3.1.1 ausreichende Möglichkeiten der Erweiterung um dynamische Funktionalität zur Verfügung stellt.

5.3.2 Architektur

In Kapitel 5.3.1 wird durch ein Grundgerüst die Schnittstelle der Erweiterung festgelegt. Demnach sollen durch eine eigenständige Anwendung aus vorliegenden Profiles Add-Ins generiert werden. Im folgenden wird die Architektur dieser Anwendung beschrieben. Dabei werden die selben Ziele wie bei der Architektur in Abschnitt 5.2.1 berücksichtigt. Dies sind eine möglichst hohe Wiederverwendung sowie möglichst einfache Anpaßbarkeit an zukünftige Versionen der UML.

Der Add-In-Generator soll ein in XMI vorliegendes Profile als Eingabe erhalten und daraus als Ausgabe Konfigurationsdateien für ein Add-In erzeugen. Dies lässt sich in folgende Schritte gliedern:

1. die XMI-Datei muss eingelesen werden und inhaltlich ausgewertet werden (*parsen*),
2. die einzelnen Bestandteile des Profiles müssen abgebildet und umstrukturiert werden, so dass sie den Bestandteilen eines Rose-Add-Ins entsprechen und
3. die Bestandteile des Rose-Add-Ins müssen in Dateien für ein Rose-Add-In geschrieben werden.

Demnach benötigt die Architektur für den Add-In-Generator folgende Bestandteile:

1. Einen Parser zum Einlesen der XMI-Datei,
2. ein internes Modell des UML-Profiles, das den ausgewerteten Inhalt der XMI-Datei enthält,
3. ein Subsystem zur Abbildung des UML-Profiles auf ein Add-In,
4. ein internes Modell des Add-Ins, auf das das Profile abgebildet werden kann und
5. ein Subsystem zum Erzeugen von Dateien aus dem Add-In-Modell.

Abbildung 5.10 stellt diese Grundstruktur dar, die im folgenden (unter Berücksichtigung der genannten Ziele Wiederverwendung und Erweiterbarkeit) verfeinert werden soll.

Zum Einlesen der XMI-Datei sollte wie beim Profile-Builder das XMI-Toolkit wiederverwendet werden. Mit dessen Hilfe werden XMI-Dateien eingelesen und auf ein Modell abgebildet.

Das Modell ist in diesem Fall ein Modell für UML-Profiles. Dies entspricht dem Modell des Profile-Builder aus Abschnitt 5.2.1 und kann daher direkt wiederverwendet werden. Das Modell wurde beim Profile-Builder durch das XMI-Toolkit aus einem Rose-Modell generiert. Es kann daher sehr einfach durch ein anderes Modell ersetzt werden, da dazu das neue Modell lediglich in Rose erstellt werden muss und kein Programmieraufwand notwendig ist. Dadurch wird für diesen Teil des Generators das Ziel der einfachen Erweiterbarkeit erfüllt.

Für die Abbildung eines Profiles auf ein Add-In (Subsystem `ModelMapping`) wäre der Versuch denkbar, dem Anwender ebenfalls eine Erweiterbarkeit für zukünftige UML-Versionen ohne Programmieraufwand zu ermöglichen. Dazu würde ein Formalismus zur Abbildung von Modellen benötigt. Die Vorstellung dabei ist, dass der Anwender in Konfigurationsdateien Abbildungsregeln und Abbildungstabellen spezifiziert. Ein Teil der Modellabbildung wäre auf diese Weise einfach realisierbar: ein Großteil der Klassen, Attribute und Referenzen aus dem Profile-Modell können direkt auf Klassen, Referenzen oder Attribute aus dem Rose-Modell abgebildet werden. Beispielsweise wird ein UML-Stereotyp auf einen Rose-Stereotypen abgebildet, der Name eines UML-Stereotypen auf den Namen eines Rose-Stereotypen, ein UML-Tagged Value auf eine Rose-Eigenschaft usw. Solche Abbildungen können durch einfache Abbildungstabellen dargestellt werden. Schwierigkeiten treten jedoch durch enthaltene Semantik auf, die vor allem in Relationen zwischen den Bestandteilen eines Profiles und teilweise auch in Attributen enthalten ist. Ein Beispiel einer solchen Relation ist die Generalisierung, die bei Stereotypen bewirkt, dass alle Tagged Values vererbt werden. Ein Beispiel für ein solches Attribut wäre ein abstrakter Stereotyp (Attribut `abstract = true`). Dieser darf nicht instanziiert werden und deshalb – da in Rose

abstrakte Stereotypen nicht unterstützt werden – nicht in ein Add-In übernommen werden.

Das Problem der Abbildung von Modellen ist unabhängig vom Anwendungsbereich der UML-Profiles und tritt beispielsweise auch bei MDA (siehe Abschnitt 2.3) auf. Ansätze zur Formalisierung von Abbildungen finden sich z. B. in [PBG01] oder [CS02].

Da Lösungen wie Abbildungstabellen demnach nicht ausreichend sind, wird entschieden, die Abbildung samt benötigter Semantik in herkömmlicher Weise als Teil der Anwendung zu implementieren, d. h. „fest zu verdrahten“. Folglich muss ein Anwender zur Anpassung des Add-In-Generator an eine andere UML-Version selbst eine entsprechende Abbildung neu implementieren. Die Architektur des Add-In-Generators sollte dies so weit wie möglich vereinfachen. Daher wird mit dem Subsystem `MappingSupport` eine Unterstützung für die Abbildung bereitgestellt. Diese soll die in Abschnitt 4.2.3 identifizierten generell notwendigen Abbildungen mittels Abbildungstabellen unterstützen. Konkret sind dies für die Erstellung von Rose-Add-Ins die Abbildungen

- von UML-Klassen auf die in Rose erlaubten Base-Classes von Stereotypen,
- von UML-Klassen auf die in Rose erlaubten Base-Classes für Tagged Values (bzw. Eigenschaften) sowie
- von UML-Datentypen auf die von Rose unterstützten Datentypen für Tagged Values (bzw. Eigenschaften).

Für jede dieser Abbildungen verwaltet `MappingSupport` eine Konfigurationsdatei. Diese enthält in einfacher Syntax (z. B. `UML:Class=Class;` `UML:Package=Category;` usw.) die einzelnen Abbildungsvorschriften. Konfigurationsdateien bieten den Vorteil, dass die umfangreiche Abbildung – es muss theoretisch für jede der weit über hundert UML-Metaklassen ein passendes Gegenstück im Rose-Metamodell identifiziert werden – eingesehen, ergänzt oder verändert werden kann, ohne Programmcode verändern zu müssen. Zusätzlich überprüft `MappingSupport` gegebene Metaklassen und Datentypen auf ihre Gültigkeit für Rose-Add-Ins.

Als weitere Maßnahme, um zukünftige Implementierungen einer Modell-Abbildung zu vereinfachen, übernimmt das Subsystem `ModelMapping` die Ablaufsteuerung des Add-In-Generators (gemäß dem Architekturmuster *Manager*). Dadurch kann ein Anwender, der ein eigenes `ModelMapping` erstellt, nach Belieben auf die vorhandenen Methoden der anderen Subsysteme zurückgreifen und beliebige weitere Funktionalität einbinden. Alle Methoden der anderen Subsysteme werden auf eine möglichst einfache und komfortable Verwendbarkeit ausgerichtet.

Das Modell eines Rose-Add-Ins könnte wie das Modell für UML-Profiles mittels des XMI-Toolkits generiert werden. Dies ist jedoch nicht notwendig, da hier keine Erweiterung für zukünftige Versionen vorgesehen ist: einerseits blieb bisher das REI aufgrund der Vielzahl der bereits vorhandenen Add-Ins stets konstant, andererseits wäre bei einer grundlegenden Änderung des REI voraussichtlich auch der gesamte Add-In-Generator zu verändern. Da kein XMI erzeugt werden soll, sondern Konfigurationsdateien für Rose, kann auch nicht das XMI-Toolkit wiederverwendet werden, sondern es ist eine eigene Implementierung zur Generierung der Konfigurationsdateien notwendig. Deshalb wird entschieden, das Rose-Modell und die Generierung der Add-Ins gemeinsam in einem Subsystem zu kapseln und die gemeinsame Schnittstelle auf eine komfortable Nutzbarkeit hin zu optimieren.

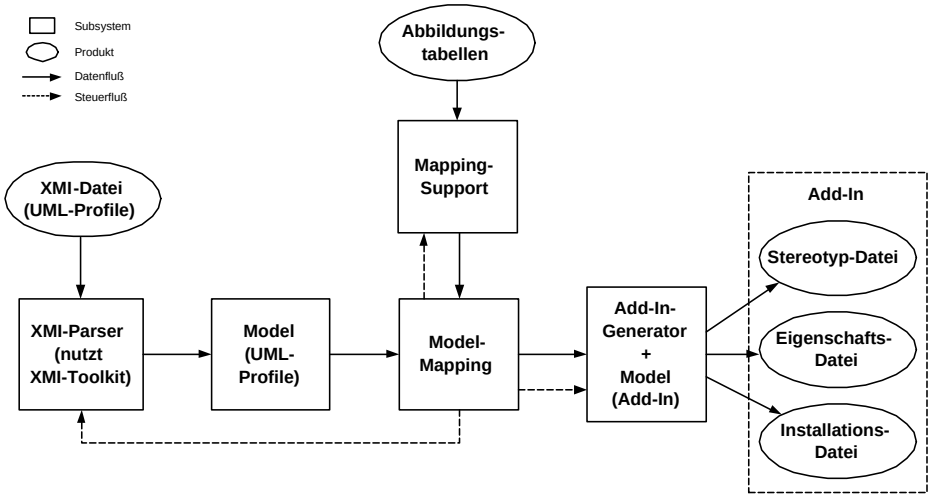

Abbildung 5.11: Die Struktur des Add-In-Generators.

Abbildung 5.11 zeigt die resultierende Struktur des Add-In-Generators. Sie entspricht einer Ausprägung des Strukturmusters *Kette*.

Zusammenfassend kann die Problemstellung beim Add-In-Generator reduziert werden auf die Abbildung eines Modells auf ein anderes Modell. Zum Einlesen des in XMI vorliegenden Modells wird das XMI-Toolkit verwendet. Dieses ermöglicht, das Modell für UML-Profiles zu generieren (bzw. die Wiederverwendung des Modells aus dem Profile-Builder) und damit einfache Anpassbarkeit ohne Programmieraufwand. Die Abbildung der Modelle selbst muss für jedes neue Profile-Metamodell neu implementiert werden. Sie wird in den Add-In-Generator integriert, indem sie die Methoden der anderen Subsysteme aufruft. Ein Teil der Abbildung wird durch Konfigurationsdateien realisiert, die durch ein Subsystem `ModelSupport` unterstützt werden. Alle Aufgaben auf Seiten des Rose-Modells werden in einem Subsystem gekapselt. Dieses ist nicht auf Anpassbarkeit an neue Rose-Modelle ausgerichtet, sondern stattdessen auf komfortable Verwendbarkeit zur Anpassung an zukünftige UML-Versionen.

5.3.3 Dynamische Unterstützung für offene Punkte

Im vorangegangenen Abschnitt wurde die Basis der Erweiterung festgelegt. Dabei werden die grundlegenden Mechanismen von Add-Ins – die Erweiterung von Rose um Stereotypen und Eigenschaften – wie von Rational vorgesehen verwendet.

Diese von Rose vorgesehenen Erweiterungen erfüllen nicht alle Anforderungen für Profile-Unterstützung aus Abschnitt 4.1. Im folgenden Abschnitt werden alle offenen Punkte beschrieben und unter Verwendung der dynamischen Funktionalität des REI Lösungsansätze vorgestellt.

Bei der Integration der Lösungsvorschläge in die Gesamtarchitektur ist jeweils zwischen den Alternativen aus Abschnitt 5.3.1.2 zu wählen.

5.3.3.1 Bindung von Tagged Values an Stereotypen

In UML sind Tagged Values zu Stereotypen zugeordnet. Deshalb sollen in Rose Eigenschaften, die Tagged Values repräsentieren, im Spezifikations-Fenster genau dann sichtbar sein, wenn dem Modellelement der zugehörige Stereotyp zugewiesen ist. In Rose können Eigenschaften aber nur pauschal einer Metaklasse zugeordnet werden; die Anwesenheit von Eigenschaften in Abhängigkeit von Stereotypen ist jedoch nicht vorgesehen. Um dies zu erreichen, muss die dynamische Funktionalität des REI verwendet werden.

Es gibt in Rose keine Möglichkeit, Eigenschaften auf einer Karteikarte ein- oder auszublenden. Sollen Eigenschaften einer Karteikarte in Abhängigkeit von einem Stereotypen erscheinen, so besteht nur die Möglichkeit, diese jedesmal aus dem gesamten Modell zu löschen bzw. diesem neu hinzuzufügen. In Rose werden Eigenschaften, deren Standard-Wert vom Anwender überschrieben wurde, intern für das jeweilige Modellelement gespeichert. Wird die Eigenschaft wieder hinzugefügt, so erhält sie den gespeicherten Wert. Insofern wäre das Hinzufügen und Löschen von Eigenschaften ein prinzipiell gangbarer Weg, da die vom Anwender zugewiesenen Werte dabei nicht verloren gehen.

Im Unterschied zu einzelnen Eigenschaften können Karteikarten sichtbar und unsichtbar gemacht werden. Deshalb besteht als zweite Lösungsmöglichkeit, für jeden Stereotypen, für den Tagged Values definiert sind, eine eigene Karteikarte mit Eigenschaften zu definieren. Im Spezifikations-Fenster eines Modellelements, das mit einem Stereotypen versehen ist, ist dann eine etwaige zugehörige Karteikarte einzublenden. Zusätzlich sollten auch für Stereotypen, die keine Tagged Values enthalten, (leere) Karteikarten vorhanden sein, damit der Anwender sich nicht unnötig lange vergewissern muss, ob tatsächlich keine Karteikarte zu dem Stereotypen existiert. Da alle Karteikarten einen eindeutigen Namen erhalten müssen, sollte dieser dem Namen des zugehörigen Stereotypen entsprechen. So bleibt auch für den Anwender trotz der wechselnden Karteikarten überschaubar, nach welcher Karte er suchen muss.

Für jedes einzelne Modellelement werden die Eigenschaften mit individuellen Werten belegt, jedoch welche Eigenschaften und Karteikarten vorhanden sind, kann nur pauschal für eine Metaklasse festgelegt werden. Deshalb ist beiden genannten Lösungen gemein, dass die Anpassungen der Eigenschaften sich stets auf alle Spezifikations-Fenster der betroffenen Metaklasse auswirken. Dies bringt mit sich, dass bei jedem Öffnen eines Spezifikations-Fensters geprüft werden muss, ob das jeweilige Modellelement mit einem Stereotypen versehen ist und entsprechend die Eigenschaften angepasst werden müssen. Das Öffnen des Spezifikations-Fensters ist ein Ereignis, für dessen Behandlung sich Add-Ins registrieren können (siehe Abschnitt 5.1.2.2).

Für die Lösung des dynamischen Erstellens und Löschens von Eigenschaften ist es problematisch, dass in Rose für beliebig viele Modellelemente gleichzeitig ein Spezifikations-Fenster geöffnet sein darf. Wird durch das Öffnen eines weiteren Spezifikations-Fensters eine Eigenschaft für eine Metaklasse gelöscht, so kann sie in keinem Spezifikations-Fenster, das zu einer Instanz dieser Metaklasse gehört, mehr editiert werden. Demnach wäre nur für das zuletzt geöffnete Spezifikations-Fenster die Verfügbarkeit der richtigen Eigenschaften gewährleistet. Beim Ein- und Ausblenden von Karteikarten dagegen gäbe es keine Probleme, da dies auf bereits geöffnete Fenster keine Auswirkung hat.

Ein weiterer Vorteil der Lösung mittels Karteikarten ist der verringerte Aufwand, da pro Stereotyp nur eine Karteikarte eingeblendet werden muss, die alle Eigenschaften enthält. Auch wird bei dieser Lösung keine Information über das Profile benötigt,

da die Zuordnung von Stereotyp zu Karteikarte durch die Namenskonvention für die Karteikarten gelöst werden kann. Für das Hinzufügen und Löschen von Eigenschaften wäre jedoch Information über alle Stereotypen und zugehörige Tagged Values notwendig.

Das Anpassen der Eigenschaften ist nicht nur beim Öffnen eines Spezifikations-Fensters notwendig, sondern auch dann, wenn der Anwender in einem offenem Spezifikations-Fenster die Auswahl des Stereotypen ändert. Dies ist aber bei beiden Lösungen nicht möglich, da sich sowohl das Ein- und Ausblenden von Karteikarten als auch das Hinzufügen und Löschen von Eigenschaften erst beim nächsten Öffnen eines Spezifikations-Fensters auswirken.

Da beide Lösungen diesen Nachteil aufweisen, soll eine dritte, unkonventionellere Lösung vorgeschlagen werden. Diese verwendet die Standardwerte-Mengen der Karteikarten. Die Standardwerte-Mengen dienen dazu, durch Auswahl einer dieser Mengen auf einer Karteikarte alle Eigenschaften gleichzeitig mit einem Wert zu belegen. Deshalb enthalten alle Standardwerte-Mengen einer Karteikarte üblicherweise die selben Eigenschaften. Intern jedoch werden Eigenschaften nicht für eine Karteikarte definiert, sondern für eine Standardwerte-Menge auf einer Karteikarte. Folglich ist es möglich, Standardwerte-Mengen zu definieren, die unterschiedliche Eigenschaften enthalten. Dies entspricht zwar nicht ihrer ursprünglich vorgesehenen Verwendung, wird aber auch im Profile-Documentor (siehe Abschnitt 5.1.3.2) angewandt und verursacht keine Fehler.

Demnach kann – analog zur oben genannten Lösung mittels Karteikarten – für jeden Stereotypen eine Standardwerte-Menge definiert werden, die zugehörige Tagged Values enthält. Für jedes generierte Add-In wird nur eine Karteikarte für die Tagged Values benötigt, welche pro Stereotyp eine Standardwerte-Menge enthält. Dabei entspricht der Name einer Standardwerte-Menge dem Namen des zugehörigen Stererotypen. Wird dem Modellelement ein Stereotyp aus dem Profile zugewiesen, so wird für das Modellelement die korrespondierende Standardwerte-Menge ausgewählt. Falls dem jeweiligen Modellelement kein Stereotyp aus dem zu unterstützenden Profile zugewiesen ist, wird die obligatorische Standardwerte-Menge *default* verwendet, die keine Eigenschaften enthält.

Diese Lösung bringt entscheidende Vorteile mit sich. Zunächst ist die Auswahl einer Standardwerte-Menge selbst eine interne Eigenschaft jedes Modellelementes, d. h. für jedes Modellelement kann individuell eine Standardwerte-Menge ausgewählt werden. Wie alle Eigenschaften bleibt die Auswahl persistent. Dadurch muss die Auswahl der Standardwerte-Menge nur dann verändert werden, wenn sich für das jeweilige Modellelement der Stereotyp ändert, und nicht bei jedem Öffnen des Spezifikations-Fensters. Das Verändern der (Standard-)Eigenschaften eines Modellelementes ist ein Ereignis, für das sich Add-Ins registrieren können. Als Parameter wird dabei übergeben, welche Eigenschaft sich geändert hat, so dass die Ereignis-Behandlung sofort beendet werden kann, falls es sich nicht um die Änderung eines Stereotypen handelt.

Weiterhin tritt bei dieser Lösung der oben genannte Nachteil aller anderen Lösungen nicht auf: auch in einem bereits offenen Spezifikations-Fenster werden Änderungen in der Auswahl der Standardwerte-Menge sofort angezeigt. Wird der Stereotyp eines Modellelementes im geöffneten Spezifikations-Fenster ausgewählt, werden demnach sofort die zugehörigen Tagged Values sichtbar.

Ein weiterer Vorteil besteht darin, dass für den Anwender immer alle Standardwerte-Mengen einsehbar sind. So sind alle Tagged Values für den Anwender transparent. Im Gegensatz zu den beiden erstgenannten Lösungen können die Tagged Values auch dann verwendet werden, wenn die zusätzliche Unterstützung zur Bindung

der Tagged Values an Stereotypen nicht verfügbar sein sollte. In diesem Fall muss der Anwender lediglich manuell die Standardwerte-Menge mit dem Namen des gewählten Stereotypen auswählen.

Abschließend soll noch bemerkt werden, dass bei dieser Lösung die ursprüngliche Funktion der Standardwerte-Mengen dennoch weiterhin verwendet werden kann. Der Anwender erstellt eigene Standardwerte-Mengen über die Benutzeroberfläche von Rose, indem er eine vorhandene Standardwerte-Menge kopiert und den enthaltenen Eigenschaften neue Standardwerte zuweist. Dies bleibt auch weiterhin für jede Menge von Tagged Values möglich.

Insgesamt bietet die dargestellte Verwendung der Standardwerte-Mengen zur Bindung von Vagged Values an Stereotypen die meisten Vorteile und ist daher zu bevorzugen. Die Definition der Standardwerte-Mengen muss bei der Generierung der Konfigurationsdatei für die Eigenschaften geschehen. Zur automatischen Auswahl der jeweils richtigen Standardwerte-Menge bei laufendem Rose ist ein OLE-Server notwendig.

5.3.3.2 Visualisierung von Tagged Values

Tagged Values sollen in Diagrammen visualisierbar sein (Anforderung A2.16). In Rose ist dies nicht vorgesehen. Das REI erlaubt auch keine Erweiterung von Diagrammen oder Diagrammelementen. Daher sind Ersatzlösungen notwendig.

Gemäß der UML-Spezifikation sollen Tagged Values unterhalb des Namens des Modellelementes, dem sie zugeordnet sind, platziert werden und deutlich als solche erkennbar sein. Bei Modellelementen, die Attribute enthalten können, wäre zunächst eine denkbare Lösung, Tagged Values in Rose als Attribute darzustellen, die durch ein Schlüsselwort `taggedValue` (anstelle eines Stereotypen) gekennzeichnet werden. Da aber dieses Schlüsselwort in Rose nicht unterstützt ist, würden die Tagged Values von Rose nicht von normalen Attributen unterschieden, was unerwünschte Seiteneffekte verursachen kann (z. B. bei der Codegenerierung). Daher ist diese Lösung zu verwerfen.

Alternativ können die Tagged Values in Kommentaren dargestellt werden. Da Kommentare beliebigen Inhalt enthalten dürfen, ist so jede gewünschte Darstellung möglich und es sind keine unerwünschten Seiteneffekte zu befürchten. Außerdem können Kommentare zu jedem Modellelement beliebig hinzugefügt werden. Hierbei sollte nicht das Dokumentations-Fenster (siehe 5.1.1.1) verwendet werden, sondern die UML-üblichen Kommentare in Diagrammen (in Rose: *Notes*), da diese in jedem Ausdruck des Diagramms enthalten sind. Die Notes werden mit dem zugehörigen Modellelement durch eine gestrichelte Linie verbunden (in Rose: *Anchor*).

Um dem Anwender Unterstützung zu bieten, sollten die Kommentare mit Tagged Values durch Rose erstellt werden. Da hierzu nur Funktionalität benötigt wird, die auch über die Benutzeroberfläche erreicht werden kann, ist dies über das REI problemlos realisierbar.

Verschiedene Alternativen sind denkbar bezüglich des Auslösers der Erstellung der Kommentare. Zu berücksichtigen ist, dass der Anwender nicht unbedingt eine Darstellung von Tagged Values in Diagrammen wünscht (Anforderung A2.17). Soll die Erzeugung der Kommentare vollautomatisch erfolgen, so muss der Anwender dies aktivieren und deaktivieren können. Nach einer Aktivierung wäre der Auslöser für die Erzeugung das Ereignis, dass Änderungen an einem Modellelement vorgenommen wurden. Das Modellelement wird dann auf eine Änderung der Standardwerte von Eigenschaften (markiert durch *Override* im Spezifikations-Fenster, siehe Abschnitt 5.1.2.4), die Tagged Values repräsentieren, untersucht, und gegebenenfalls werden Kommentare er-

zeugt. Tagged Values, die nicht vom Anwender überschrieben wurden, sollten nicht visualisiert werden.

Zusätzlich zur automatischen Unterstützung wäre es wünschenswert, dass der Anwender durch einen zusätzlichen Menüpunkt auf Wunsch nur für ausgewählte Modellelemente Kommentare generieren lassen sowie durch einen weiteren Menüpunkt die generierten Kommentare löschen kann.

Um bereits erzeugte Kommentare zu erkennen, werden sie um eine Eigenschaft erweitert. Da Notes in Rose keine Modellelemente sind, existiert für sie kein Spezifikations-Fenster, und die Eigenschaft ist nur intern für das Add-In, das die Kommentare verwalten soll, sichtbar. Wenn generierte Kommentare gelöscht werden sollen, werden diese durch die interne Eigenschaft von anderen Kommentaren unterschieden. Werden neue Kommentare erstellt, wird für jedes Modellelement zunächst geprüft, ob bereits ein Kommentar mit dieser internen Eigenschaft mit dem Modellelement über einen Anchor verbunden ist. Falls ja, wird dieser gelöscht, damit nur der aktuell erzeugte Kommentar im Diagramm vorhanden ist.

Für den Anwender ist zu beachten, dass die generierten Kommentare nicht zum Editieren vorgesehen sind. Die Werte von Tagged Values müssen stets im Spezifikations-Fenster eingegeben werden, nicht in den Kommentaren. Werden die generierten Kommentare vom Anwender um eigene Dokumentation erweitert, so besteht die Gefahr, dass diese gelöscht wird, wenn das Add-In einen Kommentar aktualisiert. Eventuell sollte bei der Löschung generierter Kommentare deren Inhalt daraufhin untersucht und gegebenenfalls unversehrt gelassen werden. Es ist in Rose nicht möglich, Kommentare als nicht für den Anwender editierbar zu definieren.

Für ein zentrales Add-In zur Visualisierung von Tagged Values ist die Information notwendig, welche Karteikarten Tagged Values beinhalten. Dazu sollten generierte Profile-Add-Ins einen Eintrag in der Windows-Registry vornehmen.

5.3.3.3 Referenzen von Tagged Values

Tagged Values können statt eines Werts eine Referenz auf ein Modellelement beinhalten. Tagged Values dieses Typs werden von Rose nicht unterstützt. Der Anwender muss den Namen des Modellelements als beliebigen String eingeben. Gemäß den Anforderungen aus Abschnitt 4.1.2.2 ist hier zusätzliche Unterstützung notwendig.

Bei Erstellung einer Referenz soll der Anwender aus gültigen Modellelementen auswählen können (Anforderung A2.13). Dies bedeutet hier, das Modellelement ist im Modell vorhanden und instanziiert die Metaklasse, die in der zum Tagged Value gehörigen Tag Definition angegeben ist.

Um diese Auswahlmöglichkeit zu realisieren, muss der Typ der Eigenschaft, die den Tagged Value in Rose repräsentiert, eine Aufzählung sein. Die Aufzählung muss die Namen aller aktuell vorhandenen Instanzen der jeweiligen Metaklasse enthalten. Es ist in Rose möglich, dynamisch den Wertebereich von Aufzählungen zu verändern. Da auch alle vorhandenen Modellelemente über das REI jederzeit abgefragt werden können, bietet das REI alle benötigten Voraussetzungen, um die Anforderung zu erfüllen.

Das Aktualisieren der Aufzählungen sollte aus Gründen der Performance von einem zentralen Add-In durchgeführt werden. Wird ein Modellelement erzeugt oder gelöscht, so aktualisiert dieses Add-In die Aufzählungen, die Instanzen dieser Metaklasse enthalten. Dafür benötigt das Add-In die Information, welche Aufzählungen zu aktualisieren sind und zu welchen Metaklassen jeweils die aufzuzählenden Modellelemente gehören. Da die Namen von Aufzählungen lediglich intern verwendet

werden und vom Anwender nicht beeinflusst werden können, können die benötigten Informationen durch Namenskonventionen übermittelt werden. Jede zu aktualisierende Aufzählung erhält daher als Namen eine vereinbarte Zeichenkette (z. B. `REF_`) und angehängt den Namen der Rose-Metaklasse, dessen Instanzen sie aufzählen soll (z. B. `REF_Class`). Dies muss bei der Generierung der Profile-Add-Ins vorgenommen werden. Die Aufzählungen enthalten zunächst nur den Standardwert `Not specified`.

Beim Start von Rose – sowie bei der nachträglichen Aktivierung weiterer Add-Ins – durchsucht das zentrale Add-In alle generierten Profile-Add-Ins nach Aufzählungen, die dieser Namenskonvention entsprechen. Generierte Profile-Add-Ins werden anhand eines speziellen Eintrages in der Windows-Registry erkannt (`isProfile = true`). Alle gefundenen Aufzählungen speichert das Add-In intern.

Wird nun ein neues Modellelement erstellt, so vergleicht das Add-In den Namen der Metaklasse des Modellelementes mit den Endungen der Namen der gespeicherten Aufzählungen. Werden Übereinstimmungen gefunden, so werden die entsprechenden Aufzählungen um den Namen des erstellten Modellelementes erweitert. Da beim Speichern eines Modells Aufzählungen mit abgespeichert werden, muss beim Öffnen eines Modells nichts weiter vorgenommen werden.

Beim Umbenennen oder Löschen eines Modellelements muss neben der Aktualisierung der Aufzählungen auch eine weitere Anforderung beachtet werden: gemäß Anforderung A2.14 muss hierbei die Konsistenz bestehender Referenzen überprüft werden. Dies sollte von dem selben zentralen Add-In vorgenommen werden. Dazu speichert dieses intern neben den Aufzählungen (siehe oben) auch alle Eigenschaften, die diese Aufzählungen verwenden (d. h. alle Eigenschaften, die Referenzen enthalten sollen).

Wird nun ein Modellelement gelöscht, so wird zuerst überprüft, ob eine der gespeicherten Eigenschaften dieses referenziert. Falls ja, wird der Anwender benachrichtigt und der Wert dieser Eigenschaft auf `Not specified` gesetzt. Danach wird der Name des gelöschten Modellelementes aus den entprechenden Aufzählungen entfernt. Wird ein Modellelement umbenannt, so wird dessen Name auch dementsprechend in den Aufzählungen und Eigenschaften geändert.

Das Öffnen, Löschen sowie das Ändern von Modellelementen sind Ereignisse, für die sich Add-Ins registrieren können (siehe Abschnitt 5.1.2.2). Dazu ist ein OLE-Server notwendig. Den Methoden zur Ereignisbehandlung wird vom REI das jeweilige Modellelement übergeben. Bei Änderung eines Modellelements wird den Methoden zur Ereignisbehandlung als Parameter übergeben, welche Änderung eingetreten ist. Dadurch kann eine Namensänderung sofort festgestellt werden.

Durch die vorgeschlagene Lösung werden mit vertretbarem Aufwand alle Anforderungen an eine Unterstützung von Referenzen erfüllt.

5.3.3.4 Multiplizitäten von Tagged Values

Gemäß der UML-Spezifikation ist für jeden Tagged Value eine Multiplizität definiert. Diese legt fest, wieviele Werte (Datenwerte oder Referenzen) der Tagged Value minimal und maximal enthalten darf. Rose unterstützt jedoch maximal einen Wert pro Eigenschaft. Daher muss Rose dahingehend erweitert werden, dass gemäß Anforderung A2.2 genau so viele Werte eingegeben werden können, wie durch die Multiplizität festgelegt ist.

Da eine Multipliztät als Obergrenze auch „unendlich“ (`unlimited`) festlegen kann, ist es nicht praktikabel, für jeden möglichen Wert eine eigene Eigenschaft zu definieren. Statt dessen müssen alle Werte innerhalb einer Eigenschaft angegeben wer-

den können. Dies ist nur möglich bei Eigenschaften vom Typ String, da dort mehrere Werte (z. B. durch Kommata voneinender getrennt) eingegeben werden können. Demnach müssen mehrwertige Tagged Values – d. h. Tagged Values, deren Multiplizität mehr als einen Wert erlaubt – durch eine Eigenschaft vom Typ String repräsentiert werden.

Diese Lösung hat jedoch den Nachteil, dass damit auch mehrwertige Tagged Values durch den Typ String dargestellt werden, die eigentlich eines anderen Typs sind. Dies hat zu Folge, dass auf Auswahllisten (für Typen `Enumeration` und `Boolean`) und auf Typüberprüfung bei frei einzugebenden Werten (für die Typen `Integer`, `Float` und `Char`) verzichtet werden muss. Dem Anwender wird nicht ersichtlich, welche Eingaben erlaubt sind.

Daher wird ein Kompromiß vorgeschlagen: bei Tagged Values, die nicht vom Typ String sind, wird der erste Wert wie gehabt durch eine Eigenschaft des ursprünglichen Typs repräsentiert. Dadurch erfährt der Benutzer, welche Eingaben für diesen Tagged Value möglich sind. Für mehrwertige Tagged Values wird eine zusätzliche Eigenschaft vom Typ String definiert, die die Eingabe des zweiten und aller weiteren Werte des Tagged Values ermöglicht. Der Name dieser zweiten Eigenschaft besteht aus einer festen Kennzeichnung (z. B. `more:`) und dem Namen des Tagged Values.

Zusätzlich sollte dem Anwender die Multiplizität eines Tagged Values sichtbar gemacht werden. Dazu kann diese einfach in Klammern hinter dem Namen der jeweiligen Eigenschaft angegeben werden.

Zur Verdeutlichung werden in Tabelle 5.1 Beispiele aus dem Profile für Variabilität ([Cla01], S. 62) angeführt.

Tagged Value			Rose-Eigenschaften	
Name	Typ	Multiplizität	Name	Typ
BindingTime	`BindingTimeEnum`	`0..*`	BindingTime (0..*) more: BindingTime	`BindingTimeEnum` `String`
VpName	`String`	`0..*`	VpName (0..*)	`String`
VariationPoint	`Integer`	`1`	VariationPoint (1)	`Integer`

Tabelle 5.1: Abbildung von Tagged Values auf Rose-Eigenschaften

Dadurch, dass der Anwender den zweiten und jeden weiteren Wert von mehrwertigen Tagged Values frei als String einträgt, werden zwei Überprüfungen notwendig: einerseits die Überprüfung der frei eingetragenen Werte auf Gültigkeit bezüglich des Typs des Tagged Values und andererseits die Überprüfung, ob die Anzahl der Werte der erlaubten Multiplizität entspricht. Letzteres ist auch für Tagged Values mit nur einem Wert notwendig, da diese vom Anwender mit einem Wert belegt werden müssen, falls als untere Schranke der Multiplizität `1` definiert ist. Diese Überprüfungen sollten von einem zentralen Add-In vorgenommen werden, damit in einem Arbeitsgang alle Tagged Values überprüft werden.

Es ist nicht möglich, durch das Add-In die Tagged Values bereits unmittelbar nach ihrer Eingabe zu überprüfen, da das REI hierzu kein behandelbares Ereignis (siehe Abschnitt 5.1.2.2) anbietet. Das REI bietet zwar ein behandelbares Ereignis an, dass bei der Änderung eines Modellelements ausgelöst wird, jedoch bezieht sich dieses nur auf die Standard-Eigenschaften von Modellelementen. Deshalb muss Rose um einen Menüpunkt erweitert werden, durch den der Anwender die Überprüfungen auslösen kann. Dies hat auch den Vorteil, dass keine zeitlichen Verzögerungen während der

Modellierung für den Anwender entstehen. Zusätzlich sollte die Überprüfungen auf Wunsch des Anwenders vor jedem Speichern eines Modells vorgenommen werden. Das Speichern eines Modells ist ein Ereignis, für das sich alle Add-Ins registrieren können. Dazu ist ein OLE-Server notwendig.

Das Add-In zur Überprüfung der Tagged Values benötigt aus der Windows-Registry lediglich die Information, welche Eigenschaften zu überprüfen sind. Dies sind alle Eigenschaften der generierten Profile-Add-Ins. Generierte Profile-Add-Ins werden anhand eines speziellen Eintrages in der Windows-Registry erkannt (`isProfile = true`). Alle weiteren benötigten Informationen sind durch die oben festgelegten Konventionen bereits in den Eigenschaften enthalten: die Multiplizitäten sind am Ende des Namens einer Eigenschaft angegeben, der Typ eines Tagged Values entspricht dem Typ der Eigenschaft, die dessen ersten Wert enthält. Mit diesen Informationen nimmt das Add-In selbständig alle Überprüfungen vor und unterrichtet den Anwender über Fehler.

Zusammenfassend soll festgehalten werden, dass die vorgeschlagene Lösung alle erforderliche Funktionalität erfüllt. Unschön ist jedoch, dass dabei mehrwertige Tagged Values durch zwei Eigenschaften repräsentiert werden müssen und weiterhin der Anwender den zweiten und alle weiteren Werte frei eintragen muss. Außerdem können Prüfungen der Eingaben des Anwenders nicht unmittelbar während oder nach der Eingabe vorgenommen werden.

5.3.3.5 Zuweisung mehrerer Stereotypen

Die UML-Spezifikation erlaubt ab Version 1.4, einem Modellelement mehrere Stereotypen zuzuweisen. Bereits in Abschnitt 5.1.1.2 wurde beschrieben, dass dies von Rose nicht unterstützt wird und deshalb das Add-In *UML* eine Eigenschaft `secondaryStereotype` zur Verfügung stellt. Dort kann ein zweiter Stereotyp eingetragen werden, was jedoch keine weitere Auswirkung im Diagramm hat.

Diese Unterstützung eines zweiten Stereotypen kann etwas erweitert werden. Dazu sollte die bestehende Eigenschaft `secondaryStereotype` genutzt werden, damit der Anwender nicht mit mehreren Eigenschaften zur Angabe des zweiten Stereotypen konfrontiert wird. Über das REI kann auch auf Eigenschaften aus anderen Add-Ins zugegriffen werden.

Der zweite Stereotyp soll möglichst ebenso vollständig unterstützt werden, wie der primäre Stereotyp. Dazu sind folgende Erweiterungen notwendig:

1. Anbieten einer Auswahlliste,
2. Anzeige im Diagramm und
3. verfügbar machen zugehöriger Tagged Values.

Das Anbieten einer Auswahlliste von Stereotypen ist zu realisieren, indem der Eigenschaft `secondaryStereotype` eine Aufzählung als Typ zugewiesen wird. Diese Aufzählung enthält alle möglichen Stereotyp-Namen. Diese müssen einmalig beim Start von Rose aus der Windows-Registry ausgelesen werden, wobei die Standard-Stereotypen und die Stereotypen aus allen aktivierten Add-Ins berücksichtigt werden müssen.

Bezüglich der Visualisierung eines zweiten Stereotypen im Diagramm gelten die gleichen Überlegungen wie für Tagged Values in Abschnitt 5.3.3.2. Da das REI keine solchen Erweiterungen erlaubt, kann ein zweiter Stereotyp nur analog zu Abschnitt 5.3.3.2 in Kommentaren angezeigt werden.

Um die zugehörigen Tagged Values eines zweiten Stereotypen zugänglich zu machen, muss analog zu Abschnitt 5.3.3.1 vorgegangen werden. Falls beide Stereotypen aus dem selben Add-In stammen, ist die Anzeige der Tagged Values des zweiten Stereotypen nicht möglich. Der Anwender muss dann selbst zwischen den beiden benötigten Standardwerte-Mengen hin und her schalten. Dieser Fall sollte jedoch selten auftreten, da die Zuweisung mehrerer Stereotypen üblicherweise nur dann notwendig wird, wenn mehrere Profiles gleichzeitig angewendet werden. Innerhalb eines Profiles werden Stereotypen normalerweise so definiert, dass einem Modellelement maximal ein Stereotyp zuzuweisen ist.

Sollen einem Modellelement mehr als zwei Stereotypen zugiewsen werden können, so muss Rose für jeden weiteren Stereotyp analog um eine zusätzliche Eigenschaft analog zur Eigenschaft `secondaryStereotype` erweitert werden. Alternativ können bei der Eigenschaft `secondaryStereotype` auch mehrere Stereotypen, durch Kommata getrennt, eingegeben werden. Dazu muss diese jedoch vom Typ String sein und es muss auf eine Auswahlliste verzichtet werden.

5.3.3.6 Anwendung von Modellbibliotheken

Gemäß Abschnitt 4.2.2 sind bei der Anwendung von Modellbibliotheken zwei Punkte zu berücksichtigen: zum einen muss eine Modellbibliothek als XMI-Datei importiert werden können, zum anderen soll der Anwender selbst bestimmen können, zu welchem Zeitpunkt eine Modellbibliothek dem Modell hinzugefügt wird.

Der erste Punkt, die Unterstützung von XMI in Rose, wurde bereits bei der Erstellung von Modellbibliotheken (siehe Abschnitt 5.2.2.2) behandelt.

Bezüglich des Zeitpunktes der Integration einer Modellbibliothek in ein Modell sind nach Abschnitt 4.2.2 zwei Fälle zu betrachten. Einerseits soll bei Erstellung eines neuen Modells vorab gewählt werden können, ob eine Modellbibliothek verwendet werden soll, andererseits soll auch nachträglich eine Modellbibliothek in ein bestehendes Modell integriert werden können.

Zur Lösung werden zwei bereits vorhandene Add-Ins verwendet. Das zu Rose mitgelieferte Add-In *Framework* erlaubt es dem Anwender, bei Erstellung eines neuen Modells eine Modellbibliothek (dort bezeichnet als *Framework*), die in das Modell einbezogen werden soll, in einem Auswahlfenster (Abbildung 5.12) auszuwählen. Eine gewählte Modellbibliothek wird dann in das neue Modell kopiert. Diese Lösung wird bereits für alle bei Rose mitgelieferten Modellbibliotheken verwendet und entspricht der UML-Spezifikation auch insofern, als Modellbibliotheken auch unabhängig von einem Profile verwendet werden können.

Modellbibliotheken werden beim Framework-Add-In registriert, indem sie als Rose-Modell-Datei (Dateiendung `mdl`) in einem speziellen Unterverzeichnis des Framework-Add-Ins abgelegt werden. Zusätzlich können noch Dateien zu ihrer Beschreibung und zu ihrer graphischen Repräsentation durch ein Icon im Auswahlfenster, aus dem der Anwender eine Bibliothek auswählen kann, hinzugefügt werden. Eine weitere Datei kann Parameter zur Darstellung der Bibliothek enthalten.

Um automatisch das Framework-Add-In zur Unterstützung von UML-Bibliotheken zu nutzen, muss eine in XMI vorliegende Modellbibliothek bei Installation eines Profile-Add-Ins in ein Rose-Modell umgewandelt und in das Unterverzeichnis des Framework-Add-Ins kopiert werden. Dies kann durch die Installationsroutine der generierten Profile-Add-Ins vorgenommen werden. Zur Umwandlung von XMI in ein Rose-Modell ist eine zusätzliche Unterstützung notwendig, da das Add-In von Unisys nur UML 1.3 unterstützt (siehe Abschnitt 5.2.2.2).

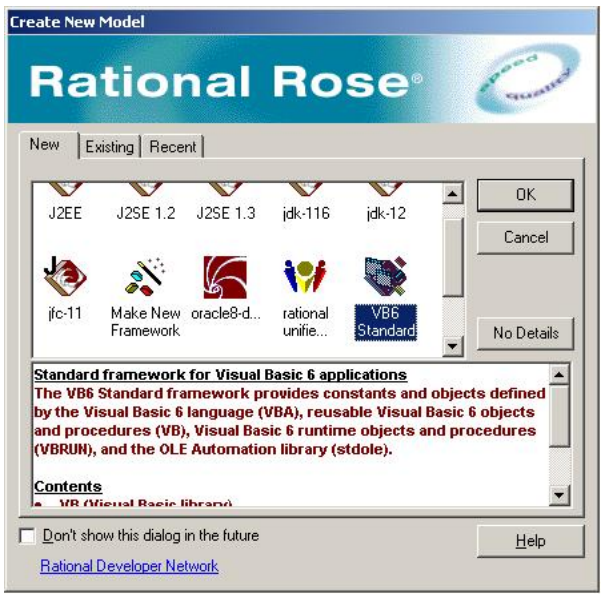

Abbildung 5.12: Das Auswahlfenster des Framework-Add-Ins zur Auswahl von Modellbibliotheken.

Zur nachträglichen Integration einer Modellbibliothek in ein bereits vorhandenes Modell wird ein zweites bereits mit Rose mitgeliefertes Add-In verwendet: das Add-In *Model-Integrator*. Dieses unterstützt die Verschmelzung zweier Rose-Modelle. Der Anwender muss dazu in Rose dieses Add-In aufrufen und die einzubindenden Rose-Modelle angeben. Auf diesem Weg können in ein bestehendes Modell ohne die Notwendigkeit einer zusätzlichen Implementierung nachträglich Modellbibliotheken eingebunden werden. Der Anwender muss lediglich informiert werden, dass die Modellbibliotheken im Verzeichnis des Framework-Add-Ins zu finden sind. Voraussetzung dafür ist auch hier, dass die Modellbibliotheken als Rose-Modell vorliegen.

Ohne die XMI-Unterstützung für UML 1.4 ist keine Unterstützung von Modellbibliotheken möglich. Behelfsweise können jedoch Modellbibliotheken genutzt werden, die entweder als XMI nach der DTD zu UML 1.3 vorliegen oder als Rose-Modell. Modellbibliotheken, die mit Hilfe von Rose erstellt wurden, können also verwendet werden. Wäre eine XMI-Unterstützung für UML 1.4 verfügbar, so würden alle Anforderungen bezüglich der Anwendung von Modellbibliotheken voll erfüllt.

Kapitel 6

Praktische Umsetzung und Bewertung

Die Konzepte aus Kapitel 5 werden praktisch umgesetzt und bewertet. Zunächst wird die prototypische Implementierung beschrieben. Dies dient auch dem Ziel, die Weiterentwicklung des Prototyps zu erleichtern. Anschließend wird der Nachweis der Verwendbarkeit der Konzepte – vorwiegend anhand von Screenshots – geführt, indem die Implementierung beispielhaft zur Anwendung des Profiles für Variabilität in Rose genutzt wird. Auf dieser Basis erfolgt abschließend die Bewertung der Konzepte.

6.1 Prototypische Implementierung

Prototypisch implementiert werden sollen vor allem die Konzepte, die auf die Erweiterungsschnittstelle von Rose angewiesen sind. Dies sind die Konzepte zur Anwendung von Profiles, d. h. der Add-In-Generator aus Abschnitt 5.3.2, sowie die dynamische Unterstützung aus Abschnitt 5.3.3. Die dynamische Unterstützung wird nur beispielhaft anhand von Rose-Skripten demonstriert, wobei auf eine Automatisierung mittels eines OLE-Servers verzichtet wird. Der Add-In-Generator, als Grundgerüst und wichtigster Bestandteil der Profile-Unterstützung, wird in allen wesentlichen Teilen realisiert.

6.1.1 Add-In-Generator

Der Add-In-Generator realisiert das Konzept aus Abschnitt 5.3.2. Da das XMI-Toolkit genutzt werden soll, wird die Programmiersprache Java verwendet. Abbildung 6.1 zeigt das Klassendiagramm der Implementierung. Die einzelnen Klassen sowie das Paket `model` werden im folgenden vorgestellt, wobei die grundlegenden algorithmischen Ideen dargestellt werden. Da die gesamte Anwendung auf externen Eingaben (XMI-Datei und Abbildungsdateien) operiert, enthalten die einzelnen Methoden der Implementierung auch zahlreiche Schritte zur Fehlerbehandlung. In diesem Detailierungsgrad soll aber nicht auf die einzelnen Methoden eingegangen werden. Es wird deshalb vorausgesetzt, dass alle Methoden erhaltene Werte zunächst prüfen und im Fehlerfall dies dem Aufrufer mittels Ausnahmen oder Rückgabewerten mitteilen, sowie aussagekräftige Fehlermeldungen ausgeben. Zusätzlich werden dem Anwender in der Konsole Nachrichten über erfolgreiche Schritte des Generierungsprozess angezeigt.

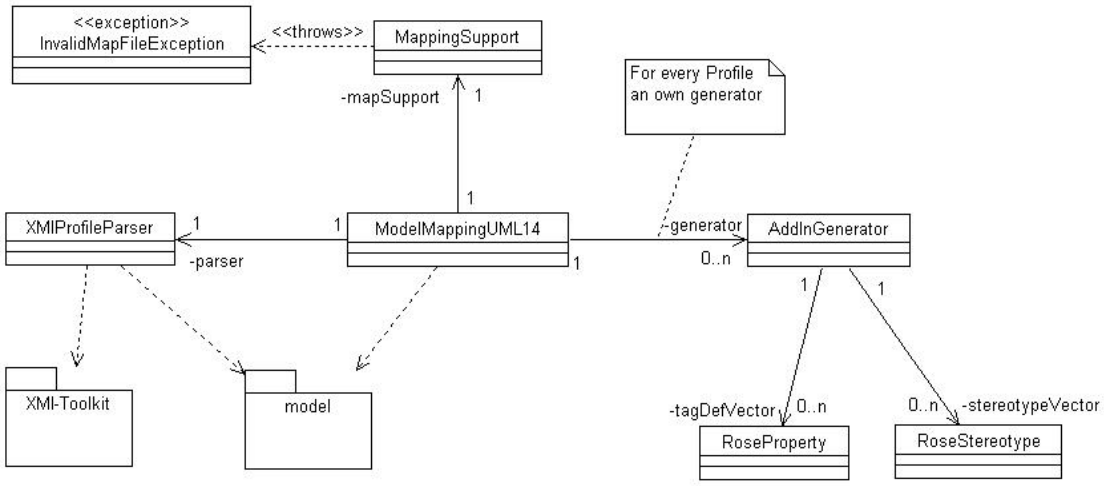

Abbildung 6.1: Klassendiagramm der Implementierung des Add-In-Generators (Paket `Add-In-Generator`).

6.1.1.1 Generierung des Modells

Das Modell für UML-Profiles wird unter Verwendung des XMI-Toolkits generiert. Das XMI-Toolkit generiert die Java-Klassen aus einem Rose-Modell. Es wurde ein Rose-Modell erstellt, das Abbildung 4.1 entspricht. Zusätzlich enthält es alle verwendeten UML-Datentypen wie `Multiplicity` oder `unlimitedInteger` als Klassen analog zu ihrer Definition in der UML-Spezifikation.

Das XMI-Toolkit kann auf verschiedene Weise verwendet werden. Es liegen Batch-Dateien und ein ausführbares Programm als einsatzbereite Werkzeuge bei. Da diese jedoch nicht alle hier benötigten Einstellungsmöglichkeiten bieten, wurden im Rahmen dieser Arbeit eigene kleine Skripte und Batch-Dateien erstellt. Durch den Aufruf der Stapelverarbeitungsdatei `makeModel` werden aus dem gespeicherten Rose-Modell Java-Klassen und eine zugehörige DTD für XMI 1.1 erstellt.

Die generierte DTD kann verwendet werden, um einzulesende Profiles zu validieren. Für UML 1.4 ist sie nicht unbedingt notwendig, da dafür die standardisierte UML 1.4 DTD besteht. Die generierte DTD hat allerdings den Vorteil, dass sie speziell für UML-Profiles erstellt ist und nicht beliebige UML-Modelle erlaubt.

Die generierten Java-Klassen befinden sich in einem Paket `model`. Sie bilden die benötigte Untermenge der UML zur Erstellung von Profiles ab. Dabei wird jedes Attribut auf eine Getter- und eine Setter-Methode abgebildet. Bei Attributen mit einer Multiplizität größer als eins sowie bei allen Referenzen liefert die Getter-Methode eine `Collection` zurück und wird die Setter-Methode durch zwei Methoden `Add` und `Remove` ersetzt. Die Methodennamen enthalten wie bei Java üblich den Namen des Attributes (z. B. `getName()`).

6.1.1.2 Die Klasse XMIProfileParser

Die Klasse `XMIProfileParser` entspricht dem Parser aus dem Konzept. Sie enthält nur eine Methode `parseXMIProfile()` zum Parsen einer XMI-Datei (Abbildung

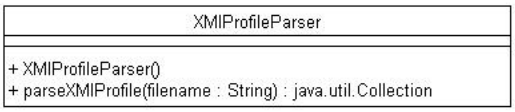

Abbildung 6.2: Die Klasse `XMIProfileParser` aus dem Paket `addInGenerator`.

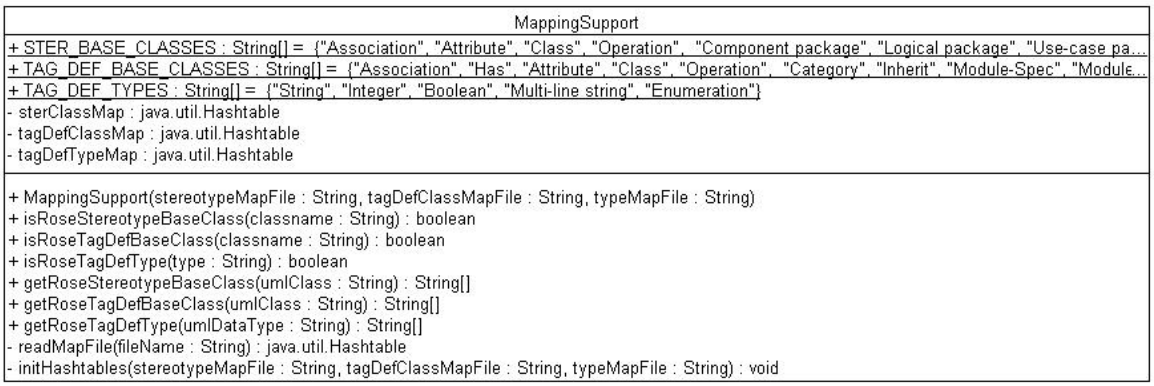

Abbildung 6.3: Die Klasse `MappingSupport` aus dem Paket `addInGenerator`.

6.2). Als Parameter wird der Methode der Name der einzulesenden XMI-Datei übergeben. Rückgabewert ist eine `Collection` der identifizierten Top-Level-Elemente in der Datei. Top-Level-Element wird üblicherweise ein Profile sein. Von den Top-Level-Elementen aus kann zum gesamten restlichen Inhalt der XMI-Datei navigiert werden. Alle Elemente sind als Instanzen der generierten Java-Klassen aus dem Paket `model` repräsentiert und sind damit sehr einfach und ohne jegliche Berührung mit XMI zu behandeln.

Da das XMI-Toolkit keine XLinks bzw. XPointer unterstützt, dürfen diese nicht vom Profile verwendet werden. Demnach müssen sich entweder alle benötigten Bestandteile des Profiles in einer gemeinsamen Datei befinden, oder die Referenzen in andere Dateien müssen mittels des XMI-Attributes `xmi.uuid` referenziert werden (siehe Abschnitt 5.1.3.3).

6.1.1.3 Die Klasse MappingSupport

Die Klasse `MappingSupport` (Abbildung 6.3) realisiert das Subsystem `MappingSupport` aus dem Konzept in Abschnitt 5.3.2. Sie dient dazu, die Abbildung der Namen der UML-Metaklassen und UML-Typen auf die Namen der Rose-Metaklassen und Rose-Typen zu unterstützen. Dazu enthält sie alle gültigen Rose-Namen und bietet die Möglichkeit, einen beliebigen Namen auf seine Gültigkeit in Rose hin zu prüfen. Weiterhin vereinfacht sie die Zuordnung der Namen, indem sie einfach zu handhabende Abbildungstabellen unterstützt.

Es sind drei Abbildungen zu unterstützen:

1. Die Abbildung von einer UML-Metaklasse auf eine oder mehrere gültige Rose-

Metaklassen als Base-Class für Stereotypen,

2. die Abbildung von einer UML-Metaklasse auf eine oder mehrere gültige Rose-Metaklassen als Base-Class für Eigenschaften und

3. die Abbildung von einem UML-Datentyp auf einen Rose-Datentyp für Eigenschaften.

Die Punkte 1. und 2. sind als getrennte Abbildungen notwendig, da sich in Rose die Menge der erlaubten Base-Classes für Stereotypen von der Menge von erlaubten Base-Classes für Eigenschaften unterscheidet. Einige UML-Metaklasse müssen auf mehrere Rose-Metaklassen abgebildet werden. So soll z. B. die UML-Metaklasse `ModelElement`, die für alle Modellelemente steht, auf sämtliche gültige Rose-Klassen abgebildet werden, da ein Stereotyp in UML, der `ModelElement` als Base-Class hat, jedem Modellelement zugewiesen werden darf. In Rose muss, da dort ein Stereotyp nur eine Base-Class haben darf, für jede als Base-Class gültige Rose-Metaklasse ein Stereotyp mit dem gleichen Namen definiert werden.

Die Klasse `MappingSupport` enthält drei Konstanten vom Typ `String[]` (Array vom Typ String): `STER_BASE_CLASSES` enthält die gültigen Base-Classes für Stereotypen, `TAG_DEF_BASE_CLASSES` die gültigen Base-Classes für Tagged Values und `TAG_DEF_TYPES` die gültigen Datentypen in Rose.

Die Methoden `isRoseStereotypeBaseClass()`, `isRoseTagDefBaseClass()` und `isRoseTagDefType()` überprüfen einen gegebenen String auf Gültigkeit, indem er mit den Strings in der zugehörigen Konstante verglichen wird.

Der Konstruktor der Klasse benötigt als Parameter die Namen von den Abbildungsdateien. Mit der privaten Methode `readMapFile()` werden diese eingelesen. Die Abbildungsdateien für Base-Classes enthalten Abbildungsvorschriften mit folgender Syntax:

$<UMLMetaklasse>$ = $<RoseKlasse_1>$, $<RoseKlasse_2>$, ..., $<RoseKlasse_n>$;

Bei Datentypen wird jeder UML-Datentyp auf genau einen Rose-Datentyp abgebildet:

$<UMLDatentyp>$ = $<RoseDatentyp>$;

Soll ein Datentyp auf eine Aufzählung abgebildet werden, so müssen die Aufzählungs-Literale definiert werden. Da eine Aufzählung mindestens zwei Literale enthalten muss, werden bei Datentypen Abbildungen auf mehrere Werte automatisch als Aufzählung interpretiert und es ergibt sich folgende Syntax:

$<UMLDatentyp>$ = $<Literal_1>$, $<Literal_2>$, ..., $<Literal_n>$;

Mit der Methode `initHashTables()` werden die einzelnen Abbildungsvorschriften in Hash-Tabellen gespeichert. Dabei werden die angegebenen Rose-Metaklassen bzw. Datentypen mittels der oben beschriebenen „is“-Methoden auf ihre Gültigkeit hin überprüft. Treten Fehler beim Parsen einer Abbildungsdatei auf oder enthält sie ungültige Rose-Metaklassen bzw. -Datentypen, so wird mit einer aussagekräftigen Fehlermeldung die selbstdefinierte Ausnahme (`InvalidMappingFileException`) ausgelöst.

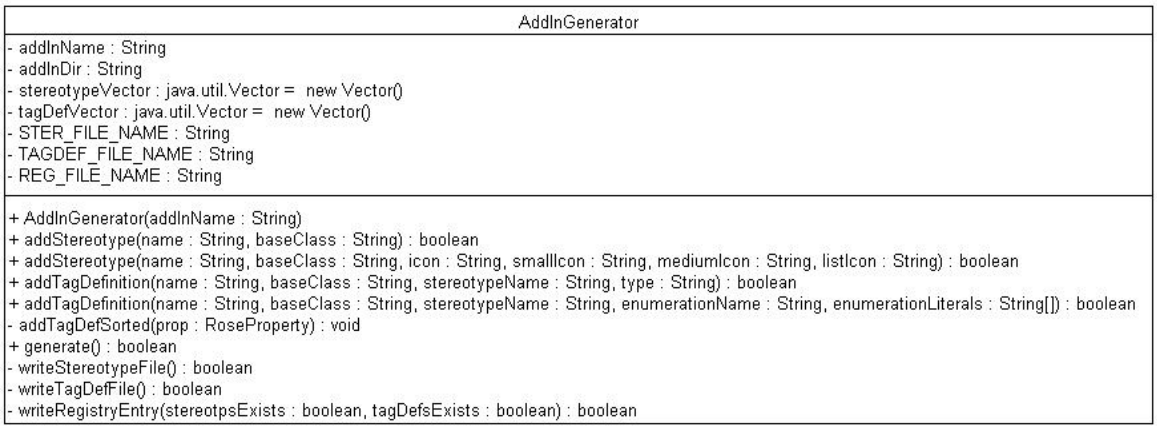

Abbildung 6.4: Die Klasse `AddInGenerator` aus dem Paket `addInGenerator`.

Schließlich können mit den Methoden `getRoseStereotypeBaseClass()`, `getRoseTagDefBaseClass` und `getRoseTagDefType()` für eine gegebene UML-Metaklasse bzw. -Datentyp der oder die korrespondierenden Metaklassen abgefragt werden. Rückgabewerte sind die Einträge aus den Hash-Tabellen. Ist für den gegebenen Bezeichner keine Abbildungsvorschrift vorhanden, wird `null` zurückgegeben.

6.1.1.4 Die Klassen Add-In-Generator, RoseStereotype und RoseProperty

Die Klasse `AddInGenerator` (Abbildung 6.4) realisiert das Subsystem `AddInGenerator` aus dem Konzept in Abschnitt 5.3.2. Sie führt den eigentlichen Generierungsprozess durch. Erzeugt werden je eine Konfigurationsdatei für Stereotypen und für Eigenschaften sowie ein Skript zur Installation des Add-Ins.

Die Klassen `RoseStereotype` und `RoseProperty` bilden das vereinfachte Modell für ein Rose-Add-In. Sie kapseln lediglich die Attribute eines Stereotypen bzw. einer Eigenschaft in einem Rose-Add-In und dienen als Hilfsklassen für `AddInGenerator`.

Dem Konstruktor der Klasse `AddInGenerator` kann der Name des Add-Ins, unter dem es in Rose sichtbar ist, die Namen der erzeugten Dateien und der Name des Verzeichnisses, indem diese abgelegt werden, übergeben werden. Dabei wird das Verzeichnis des angegeben Namens stets im Unterverzeichnis `Output` im Verzeichnis der Anwendung erstellt. Normalerweise sollte der alternative Konstruktor verwendet werden, der als Parameter nur einen Add-In-Namen benötigt und alle anderen Namen aus diesem erzeugt. Der Add-In-Name entspricht üblicherweise dem Namen des Profiles.

Mit den Methoden `addStereotype()` werden neue Stereotypen hinzugefügt. Als Parameter sind stets Name und die Base-Class des Stereotypen anzugeben, andernfalls wird kein Stereotyp erzeugt. Die Base-Class muss eine gültige Rose-Metaklasse sein und wird daraufhin überprüft. Optional können Icons für den Stereotypen angegeben werden. Ist kein entsprechendes Icon vorhanden, ist ein leerer String zu übergeben. Es wird eine Instanz der Klasse `RoseStereotype` mit entsprechenden Werten erzeugt.

Analog dazu wird mit den Methoden `addTagDefinition()` eine Tag Definition dem Add-In hinzugefügt. Als Base-Class muss eine in Rose gültige Base-Class für Eigenschaften übergeben werden. Der Parameter `stereotypeName` enthält den Namen des Stereotypen, dem die Tag Definitions zugeordnet sind, damit eine Bindung verfügbarer Tagged Values an Stereotypen in Rose realisiert werden kann. Alternativ kann entweder ein einfacher Rose-Datentyp als Typ übergeben werden oder eine Aufzählung. Im zweiten Falle sind zusätzlich mindestens zwei Aufzählungs-Literale anzugeben. Es wird eine Instanz der Klasse `RoseProperty` mit entsprechenden Werten erzeugt.

Die dem Add-In hinzugefügten Stereotypen und Tag Definitions werden jeweils in einem privaten Behälter der Klasse `java.lang.util.Vector` gespeichert. Tag Definitions werden dabei durch die private Methode `addTagDefSorted()` sortiert. Dies ist notwendig, da eine Konfigurationsdatei für Eigenschaften erzeugt werden soll, die eine Bindung an Stereotypen mittels Standardwerte-Mengen nach Abschnitt 5.3.3.1 erlaubt. Unter Berücksichtigung des erforderlichen Aufbaus von Konfigurationsdateien von Eigenschaften müssen die Eigenschaften nach Metaklassen, und innerhalb dessen nach Stereotypen, zu denen sie gehören, geordnet werden.

Sind alle Stereotypen und Tag Definitions hinzugefügt, ist die Methode `generate()` aufzurufen. Diese löst den Generierungsprozess aus, indem sie die privaten Methoden `writeStereotypeFile()`, `writeTagDefFile()` und `writeInstallFile()` benachrichtigt.

Wurden zum Add-In Stereotypen hinzugefügt, generiert die Methode `writeStereotypeFile()` dafür eine Konfigurationsdatei. Den Namen der Stereotypen wird als Pfadname der Name des Add-Ins vorangestellt. Dies vermeidet nicht nur Namenskonflikte gemäß Abschnitt 3.1.2.6, sondern ermöglicht dem Anwender auch, alle Stereotypen des Profiles auf einen Blick übersehen zu können, da diese dann in der alphabetisch sortierten Stereotypen-Auswahlliste direkt untereinander stehen.

Sind Eigenschaften vorhanden, so werden diese durch die Methode `writeTagDefFile()` in eine Konfigurationsdatei für Eigenschaften geschrieben. Für jede Metaklasse, für die Eigenschaften definiert sind, werden Standardwerte-Mengen erzeugt. Zusätzlich zu einer Standardwerte-Menge pro Stereotyp wird für jede Metaklasse eine leere Standardwerte-Menge `default` erzeugt, die benutzt werden soll, wenn einem Modellelement kein Stereotyp aus dem Profile zugeordnet ist. Aufzählungen müssen in jeder Standardwerte-Menge, in der sie verwendet werden, definiert sein, wobei durch eine Prüfung unerwünschte Mehrfachdefinitionen innerhalb einer Standardwerte-Menge vermieden werden. Jeder Eigenschaft muss ein Standardwert zugewiesen werden. Da UML 1.4 (im Gegensatz zu UML 1.3) keine Standardwerte für Tag Definitions unterstützt, können keine sinnvollen Standardwerte generiert werden. Daher wird für den Typ `String` der leere String, für den Typ `Integer` die 0 und für den Typ `Boolean` der Wert `false` verwendet. Für alle Aufzählungen wird ein zusätzliches Literal `Not specified` definiert, das als Standardwert verwendet wird.

Sofern das Add-In Stereotypen oder Eigenschaften enthält, wird mit der Methode `writeInstallFile()` ein Skript zur Installation geschrieben. In dieser prototypischen Implementierung nimmt dieses nur die notwendigen Einträge in die Windows-Registry vor. Dazu muss unter `[HKEY_LOCAL_MACINE\SOFTWARE\Rational Software\Rose\AddIns\]` ein neuer Schlüssel mit dem (internen, s. u.) Namen des Add-Ins angelegt werden, der alle Einträge enthält.

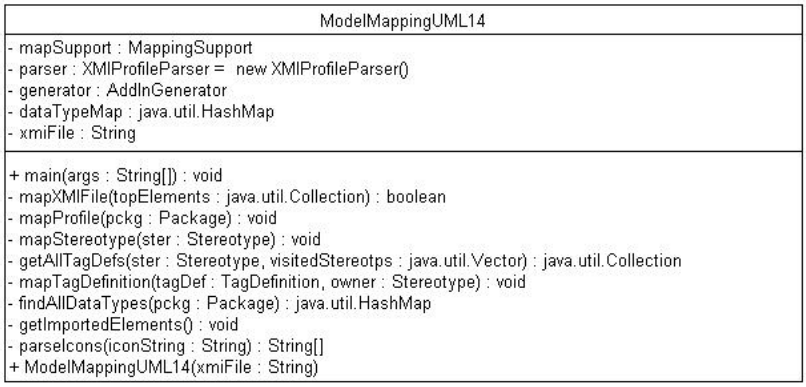

Abbildung 6.5: Die Klasse `ModelMapping14` aus dem Paket `addInGenerator`.

Das Add-In soll im Verzeichnis von Rose in einem Unterverzeichnis mit dem Namen des Add-Ins installiert werden. Der Pfad des Add-Ins muss in die Registry eingetragen werden. Da zum Zeitpunkt der Generierung noch nicht feststeht, wo sich auf dem jeweiligen Rechner das Rose-Verzeichnis befindet, liest das Skript diesen Wert zum Zeitpunkt der Installation aus der Registry und passt die vorzunehmenden Einträge des Add-Ins entsprechend an. Minimal erforderliche Einträge sind das Verzeichnis des Add-Ins, die Konfigurationsdateien für Stereotypen und Tagged Values sowie ein Eintrag, ob es sich um ein Language-Add-In handelt. Die generierten Add-Ins werden als Basic-Add-Ins eingetragen. Damit das Add-In in Rose auch tatsächlich durchgehend sichtbar bleibt, muss sich der interne Name des Add-Ins vom in Rose anzuzeigenden Namen unterscheiden (siehe [Rat02e]). Angezeigt werden soll der Name des Profiles, der der Klasse `AddInGenerator` im Konstruktor übergeben wurde. Als interner Name wird dieser um den Bezeichner `Profile` verlängert. Im Gegensatz zu den Spezifikations-Fenstern werden in Rose im Add-In-Manager die Add-Ins unter ihrem internen Namen angezeigt.

Das generierte Skript trägt den Namen `install.vbs` und ist in der Sprache *Visual Basic Script* verfasst. Es kann direkt ausgeführt werden und wird durch den im Betriebssystem *Windows* integrierten *Windows Scripting Host* interpretiert. Da es nur die Einträge in die Registry vornimmt, muss der Anwender selbstständig das generierte Add-In-Verzeichnis in das Verzeichnis von Rose kopieren. Icons zu Stereotypen müssen im Add-In-Verzeichnis in ein Unterverzeichnis `icons` kopiert werden. Das generierte Skript weist den Anwender darauf hin und teilt ihm das gefundene Rose-Verzeichnis mit.

6.1.1.5 Die Klasse ModelMappingUML14

Die Klasse `ModelMappingUML14` realisiert die Abbildung eines UML-Profiles auf ein Rose-Add-In aus dem Konzept in Abschnitt 5.3.2. Sie steuert die gesamte Anwendung unter Verwendung der anderen Klassen des Paketes. Für jedes Modell für UML-Profiles, d. h. für jede zu unterstützende UML-Version, muss eine Abbildungsklasse implementiert werden. Die vorliegende Klasse untertützt UML 1.4.

Die Klasse enthält die Methode `main()`, die in Java bei Programmstart aufgerufen wird. Sie erzeugt eine Instanz der Klasse `ModelMappingUML14` und übergibt

dieser die XMI-Datei, die das UML-Profile enthält, aus dem ein Add-In generiert werden soll. Dazu wird ein Dialog zur Auswahl einer XML- bzw. XMI-Datei angezeigt. Eine umfangreichere Benutzeroberfläche wäre sicherlich denkbar, wird aber in diesem Prototyp nicht implementiert.

Der Konstruktor der Klasse übernimmt den Namen der XMI-Datei. Es werden Instanzen der Klasse `MappingSupport` und `XMIProfileParser` erzeugt. An `MappingSupport` müssen die Namen der Abbildungsdateien übergeben werden. In diesem Prototyp sind jene festgelegt auf die Dateinamen `SterMap.map`, `TagDefClass.map` und `TagDefType.map`, jeweils im Unterverzeichnis `mappings` des Verzeichnises der Anwendung. Es wird die Methode `parseXMIProfile()` des `XMIProfileParser` aufgerufen und ihr der Name der einzulesenden XMI-Datei übergeben. Mit dem erhaltenen Rückgabewert – den Top-Level-Elementen der XMI-Datei – als Parameter wird die Methode `mapXMIFile()` aufgerufen.

Die private Methode `mapXMIFile()` durchläuft die erhaltenen Elemente und sucht darin nach Paketen, die mit dem Stereotyp `profile` versehen sind. Für jedes gefundene Profile wird die Methode `mapProfile()` aufgerufen.

Die private Methode `mapProfile()` benötigt ein abzubildendes Profile als Parameter. Für jedes Profile wird ein eigenes Add-In erstellt, weshalb eine neue Instanz von `AddInGenerator` erzeugt wird. Dieser wird als Add-In-Name der Name des Profiles übergeben. Außerdem muss für jedes Profile separat die Methode `findAllDataTypes()` aufgerufen werden. Anschließend wird für jeden enthaltenen Stereotypen die Methode `mapStereotype()` aufgerufen. Sind alle Elemente des Profiles abgebildet, so wird durch Aufruf der Methode `generate()` der zugehörigen Instanz von `AddInGenerator` ein Add-In aus dem Profile generiert.

Die private Methode `findAllDataTypes()` ist notwendig, da in einem Profile anwenderdefinierte Datentypen nur über ihren Namen referenziert werden. Das Attribut `tagType` der UML-Metaklasse `TagDefinition` ist vom Typ String. Daher kann das Attribut keine navigierbaren Referenzen auf anwenderdefinierte Datentypen enthalten. Folglich ist nicht direkt erkennbar, ob der Wert des Attributs einen anwenderdefinierten Datentyp, einen UML-Datentyp oder eine Referenz auf eine Metaklasse beinhaltet und es bleibt nur die Möglichkeit, den Wert des Attributes mit allen vorhandenen Namen von anwenderdefinierten Datentypen, UML-Datentypen und Metaklassen zu vergleichen. Aus diesem Grund muss das Profile nach allen anwenderdefinierten Datentypen durchsucht worden sein, bevor Stereotypen und ihre zugehörigen Tag Definitions abgebildet werden können.

`findAllDataTypes()` legt alle gefundenen anwenderdefinierten Datentypen in einer Hash-Tabelle ab, die dem privaten Attribut `dataTypeMap` zugewiesen wird. Die Hash-Tabelle enthält als Schlüssel die Namen des Datentyps, als zugehörigen Wert jeweils ein Array vom Typ `String`. Dieses enhält den Rose-Typ, auf den der Datentyp abgebildet werden soll. Aufzählungen in UML werden auf Aufzählungen in Rose abgebildet. Andere anwenderdefinierte Datentypen werden auf den Typ `String` abgebildet.

Die private Methode `mapStereotype()` fügt Stereotypen zum Add-In mittels der Methode `addStereotype()` der Klasse `AddInGenerator` hinzu. Zunächst wird das geerbte Attribut `isAbstract` geprüft. Ist ein Stereotyp als abstrakt definiert, wird er ignoriert. Mit der privaten Methode `parseIcons()` wird das Attribut ausgewertet. Die UML-Base-Class des Stereotypen wird mittels der Methode `getRoseStereotypeBaseClass()` der Klasse `MappingSupport` auf ein oder mehrere Rose-Base-Classes abgebildet. Für jede erhaltene Rose-Base-Class wird

die Methode `addStereotype()` je einmal aufgerufen. Anschließend werden alle Tag Definitions gesucht, die dem Stereotypen zugeordnet sind. Da dabei auch geerbte Tagged Definitions berücksichtigt werden müssen, wird dies in die Methode `getAllTagDefs()` ausgelagert. Schließlich wird jede gefundene Tag Definition der Methode `mapTagDefinition()` übergeben.

Die private Methode `parseIcons` wertet das Attribut `icons` aus. Das Attribut `icon` eines Stereotypen enthält in der XMI-Datei eine beliebige Anzahl von Referenzen auf Graphikdateien, die zum Stereotyp gehörige Icons enthalten. Daher liegt es nach dem Einlesen der XMI-Datei durch das XMI-Toolkit als ein String vor. Die einzelnen, durch Kommata getrennten Dateinamen, werden separiert. Es wird geprüft, ob Namen darunter sind, die den Konventionen in Rose zur Angabe von Icons entsprechen. Nach Konvention endet der Dateiname der Icons für die Werkzeugleiste (`smallIcon` und `mediumIcon`) auf `_s.bmp` bzw. `_m.bmp` und des Icons für das Explorerfenster (`listIcon`) auf `_l.bmp`. Sie werden daher alle im Graphikformat *Windows Bitmap* erwartet. Das Icon für Diagramme muss als *Windows Meta File* (Dateieindung `.wmf`) oder als *Enhanced Meta File* (Dateieindung `.emf`) vorliegen. Denkbar wären hier noch weitere Prüfungen, ob die Dateien tatsächlich gültig und verfügbar sind, sowie ein automatisches Kopieren der Dateien in das Add-In. In dieser Implementierung muss der Anwender selbst alle benötigten Icons in das Unterverzeichnis `icons` des Add-In-Verzeichnisses stellen.

Die private Methode `getAllTagDefs()` sucht rekursiv nach allen Tag Definitions, die einem Stereotyp direkt oder durch Vererbung zugeordnet sind. Ihr Rückgabewert sind alle gefundenen Tag Definitions. Bei der rekursiven Navigation entlang aller Generalisierungen eines Stereotypen besteht die Möglichkeit, dass der gleiche Stereotyp mehrfach als Eltern-Stereotyp gefunden wird. Dies würde beispielsweise auftreten, wenn ein Stereotyp zwei Stereotypen spezialisiert, die wiederum beide den gleichen Stereotypen spezialisieren. Um dies, sowie etwaige Endlosschleifen, auszuschließen, wird jeder besuchte Stereotyp gespeichert. Dazu wird der Methode Parameter `visitedStereotps` übergeben, der beim initialen Aufruf der Methode einen leeren `Vector` oder auch `null` enthält.

Die private Methode `mapTagDefinition()` erhält als Parameter eine Tag Definition, die dem Add-In hinzugefügt werden soll, sowie einen Stereotypen, an den diese gebunden werden soll. Die Angabe eines Stereotypen ist notwendig, da eine Tag Definition nicht nur an den Stereotypen, dem sie im UML-Profile zugeordnet ist (Attribut `owner`), gebunden sein soll, sondern auch an alle Stereotypen, die diesen spezialisieren. Diese Zuordnungen, sowie der Ausschluß abstrakter Stereotypen aus dem Add-In, wurden bereits in der Methode `mapStereotype()` vorgenommen. In `mapTagDefinition()` werden zunächst die Base-Classes der Tag Definition abgebildet. Dazu wird die Methode `getRoseTagDefBaseClass()` aus der Klasse `MappingSupport` benutzt. Dieser wird als Parameter die Base-Class des Stereotypen, an den die Tag Definition gebunden ist, übergeben. Weiterhin muss der Typ der Tag Definition abgebildet werden. Wie bereits zu `findAllDataTypes()` (siehe oben) beschrieben, lässt sich aus dem Attibut `tagType` nicht direkt ablesen ob es sich um einen anwenderdefinierten Typ, einen UML-Datentyp oder eine Referenz auf eine Metaklasse handelt. Deshalb wird zunächst die durch die Methode `findAllDataTypes()` erstellte Hash-Tabelle `dataTypeMap` nach einer Abbildungsvorschrift abgefragt. Handelt es sich nicht um einen anwenderdefinierten Typ, so wird mittels der Methode `getRoseTagDefType()` der Klasse `ModelMapping` versucht, eine der Abbildungsvorschriften für UML-Datentypen anzuwenden. Ist auch dies erfolglos, so wird, da in dieser Implementierung Referenzen auf Metaklassen

nicht weitergehend unterstützt werden, der Datentyp auf den Typ `String` abgebildet. Schließlich wird die Tag Definition mit der abgebildeten Base-Class und dem abgebildeten Typ dem Add-In mit der Methode `addTagDefinition()` der Klasse `AddInGenerator` hinzugefügt. Dies wird, falls die Base-Class auf mehrere Rose-Metaklassen abgebildet wurde, für jede von diesen je einmal vorgenommen.

Nachdem die XMI-Datei auf ein Add-In für jedes enthaltene Profile abgebildet wurde, wird die Anwendung beendet. Der Anwender kann nun, wie unter Abschnitt 6.1.1.4 beschrieben, durch Kopieren der erzeugten Verzeichnisse in das Verzeichnis von Rose und Aufruf von `install.vbs` die Add-Ins bei Rose installieren und verwenden.

6.1.2 Dynamische Unterstützung

An dynamischer Unterstützung soll nur die Machbarkeit der Rose-spezifischen Funktionlität als Nachweis der Funktionsfähigkeit der Konzepte demonstriert werden. Auf Implementierung weiterer Funktionalität, die mit allgemeinen Programmierkenntissen unabhängig von dieser Arbeit zu realisieren ist, soll verzichtet werden. Aus diesem Grund wird die dynamische Unterstützung anhand von Rose-Skripten realisiert, die im folgenden vorgestellt werden. Die verbleibenden Schritte, um die Funktionalität mittels eines OLE-Servers vollständig zu automatisieren, wird jeweils im Anschluß beschrieben.

An dynamischer Unterstützung aus Abschnitt 5.3.3 wird die Bindung von Tagged Values an Stereotypen (Abschnitt 5.3.3.1) und die Visualisierung von Tagged Values (Abschnitt 5.3.3.2) prototypisch demonstriert. Dazu werden Rose-Skripte erstellt, die über Menüpunkte aufgerufen werden können. Es wird ein selbständiges Add-In `UML Profiles` erstellt, das Rose um diese Menüpunkte erweitert.

Das Add-In `UML Profiles` besteht aus einer Konfigurationsdatei für Menüpunkte `UMLProfiles.mnu`, einer Installtionsdatei `install.vbs` und zwei Rose-Skripten, `BindTaggedValues.ebs` und `ShowTaggedValues.ebs`. Alle Dateien befinden sich in einem Verzeichnis `UML Profile`. Zur Installation des Add-Ins muss das Verzeichnis `UML Profile` in das Verzeichnis von Rose kopiert werden und die Installationsdatei `install.vbs` ausgeführt werden.

Ab dem nächsten Start von Rose ist dessen Hauptmenüleiste im Menü `Tools` um ein Untermenü `UML Profile` erweitert. Dieses enthält Menüpunkte `Bind Tagged Values` und `Show Tagged Values`, welche die zugehörigen Rose-Skripte aufrufen. Diese Erweiterung des Menüs wird durch die Konfigurationsdatei `UMLProfiles.mnu` gemäß [Rat01b] vorgenommen.

Das Rose-Skript `BindTaggedValues.ebs` wählt für markierte Modellelemente die Standardwerte-Mengen in Abhängigkeit vom gewählten Stereotyp aus (nach Abschnitt 5.3.3.1). Dazu prüft es für jedes markierte Modellelement den Namen des zugewiesenen Stereotypen. Da Stereotypen aus den generierten Add-Ins mit dem vollen Pfadnamen (d. h. voranstehend den Namen des Profiles als Paket-Namen) benannt sind, kann aus dem Namen des Stereotypen der Name des zugehörigen Profiles (bzw. Add-Ins) abgeleitet werden. Mittels des Add-In-Namens kann, sofern vorhanden, die zum Add-In gehörige Karteikarte gefunden werden, die Tagged Values enthält. Innerhalb dieser Karteikarte wird nach einer Standardwerte-Menge gesucht, die dem Namen des Stereotypen entpricht. Wird die Standardwerte-Menge gefunden, so wird diese für das Modellelement als aktuell gewählte Standardwerte-Menge gesetzt. Dadurch sind die gewünschten Tagged Values ausgewählt. Kann keine Standardwertemenge gefunden

werden – d. h. das Modellelement ist nicht mit einem Stereotypen aus einem genrierten Add-In versehen – wird die Standardwertemenge `default` gesetzt.

Das Rose-Skript `ShowTaggedValues.ebs` visualisiert Tagged Values, denen vom Anwender ein Wert zugewiesen wurde, in Kommentaren gemäß Abschnitt 5.3.3.2. Ebenso wie `BindTaggedValues.ebs` berücksichtigt es nur markierte Modellelemente. Zunächst wird wie bei `BindTaggedValues.ebs` abhängig vom Stereotyp die Standardwertemenge gesetzt, damit nur Tagged Values visualisiert werden, deren Stereotyp auch tatsächlich dem jeweiligen Modellelement zugewiesen ist. Diese Standardwertemenge wird anschließend daraufhin überprüft, ob sie Eigenschaften enthält, die als überschrieben markiert sind. Sind solche Eigenschaften vorhanden, wird ein Kommentar im Diagramm erzeugt, der diese Eigenschaften und die ihnen zugewisenen Werte enthält.

6.1.3 Schritte zur vollständigen Implementierung

Dieser Abschnitt gibt eine kurze Zusammenstellung, welche Schritte zu einer vollständigen Implementierung vorzunehmen wären.

Es ist noch folgende Funktionalität zu implementieren:

- Unterstützung von Referenzen (per XLink und XPointer) auf Elemente in anderen XMI-Dateien beim Einlesen von Profiles (siehe Abschnitt 5.1.3.3),
- Methode `getImportedElements()` in der Klasse ModelMappingUML14,
- Umsetzung der dynamischen Unterstützung in einem OLE-Server und
- die Implementierung einer XMI-Unterstützung von UML 1.4 für Rose.

6.2 Anwendung auf das Profile für Variabilität

Die im vorangegangenen Abschnitt vorgestellt Implementierung soll in diesem Abschnitt konkret angewendet werden. Dies dient zum einen als Anwenderdokumentation und um dem Leser ein deutliches Bild der Implementierung zu vermitteln. Zum anderen wird daduch die Verwendbarkeit der Konzepte nachgewiesen. Mit dem Profile für Variabilität erfolgt der Nachweis an einem Profile mit praktischer Relevanz in der Industrie.

Nach dem Start des Add-In-Generators erscheint ein Dialog zur Auswahl einer XMI-Datei. Hier muss die Datei ausgewählt werden, die das Profile enthält, zu dem ein Add-In generiert werden soll. Nach Auswahl der Datei werden die einzelnen Schritte des Generierungsprozesses in der Konsole angezeigt (Abbildung 6.6). Der Add-In-Generator erstellt ein Verzeichnis mit dem Namen des Add-Ins, das drei Dateien enthält. Durch Ausführen des Skripts `install.vbs` wird das generierte Add-In in der Windows-Registry installiert (Abbildung 6.7). Ein Dialog gibt den Anwender darüber Auskunft, wohin das generierte Verzeichnis kopiert werden muss. Nach dem nächsten Start von Rose sind dort die generierten Add-ins verfügbar. Sie werden im Add-In-Manager angezeigt und können dort aktiviert und deaktiviert werden (Abbildung 6.8). Standardmäßig sind sie aktiv.

Im Spezifikationsfenster eines Modellelementes sind alle Stereotypen, deren Base-Class dem jeweiligen Modellelement entspricht, aus dem Add-In verfügbar (Abbildung 6.9) und können dem Modellelement zugewiesen werden. Sofern für einen Stereo-

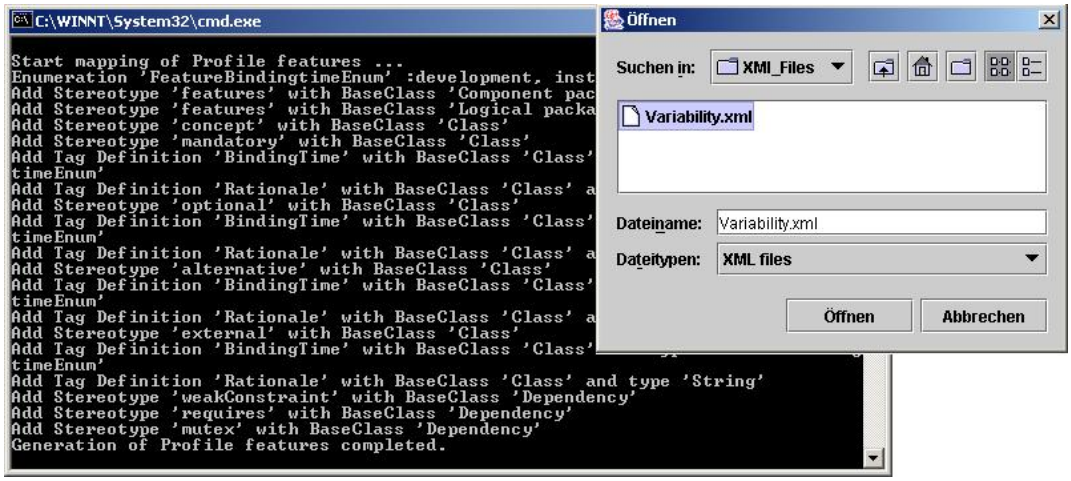

Abbildung 6.6: Anwendung des Add-In-Generators.

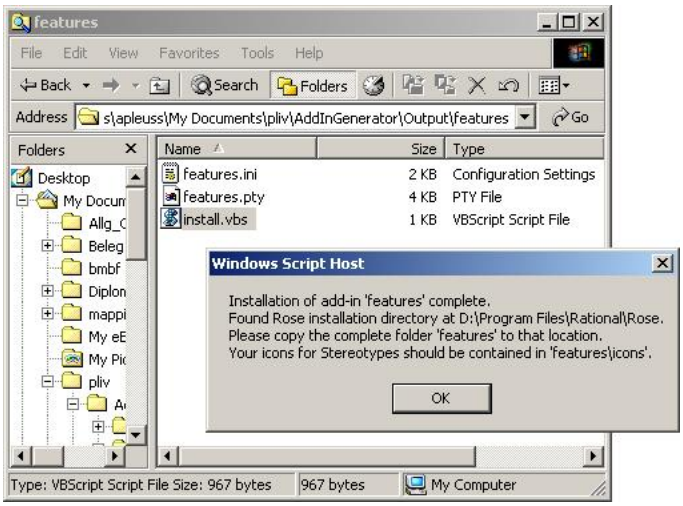

Abbildung 6.7: Installation des generierten Add-Ins `features`.

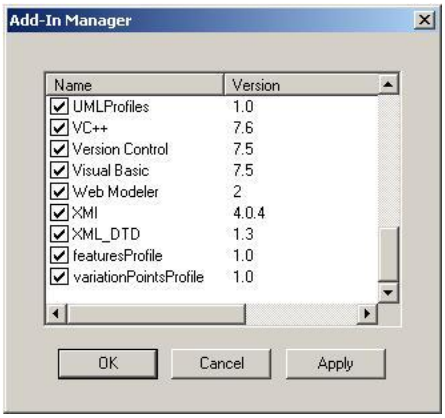

Abbildung 6.8: Die generierten Add-Ins sind im Add-In-Manager verfügbar.

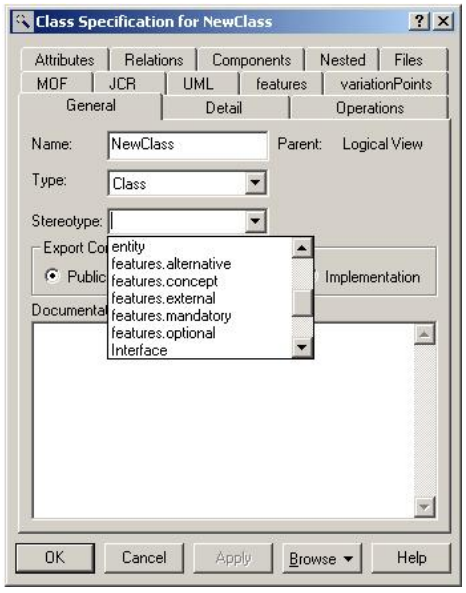

Abbildung 6.9: Stereotypen aus dem Profile sind in der Auswahlliste im Spezifikations-Fenster ihrer Base-Class verfügbar.

typen Icons definiert sind, kann er im Diagramm in allen drei Darstellungsvarianten angezeigt werden (Abbildung 6.10). Der Anwender kann Stereotypen alternativ auch ausblenden. Falls entsprechende weitere Icons definiert wurden, ist der Stereotyp im Explorer-Fenster sichtbar (Abbildung 6.11) und kann der Werkzeugleiste hinzugefügt werden . Mittels der Werkzeugleiste kann direkt eine Instanz einer virtuellen Metaklasse erstellt werden (Abbildung 6.12).

Für jedes Add-In ist eine Karteikarte mit Tagged Values im Spezifikations-Fenster vorhanden, falls für die entprechende Metaklasse Tagged Values definiert wurden. Diese enthalten für jeden Stereotypen mit entsprechender Base-Class eine Standardwerte-Menge. Die Standardwerte-Menge enthält Eigenschaften, welche die Tagged Values des jeweiligen Stereotypen repräsentieren (Abbildung 6.13). Die Datentypen der Tagged Values sind auf Rose-Datentypen abgebildet. Anwenderdefinierte Aufzählungen aus Profiles bleiben erhalten (Abbildung 6.14).

Die im Profile für Variabilität definierte wiederverwendbare Constraint `xor` kann zumindest manuell erstellt werden (Abbildung 6.15).

Zur dynamischen Unterstützung von Profiles ist ein Add-In `UML Profiles` vorhanden. Dieses dient zur Bindung von Tagged Values an Stereotypen sowie zur Generierung von Kommentaren zur Darstellung von Tagged Values, denen vom Anwender ein Wert zugewiesen wurde, im Diagramm (Abbildung 6.16).

Damit kann das Profile für Variabilität vollständig in Rose verwendet werden.

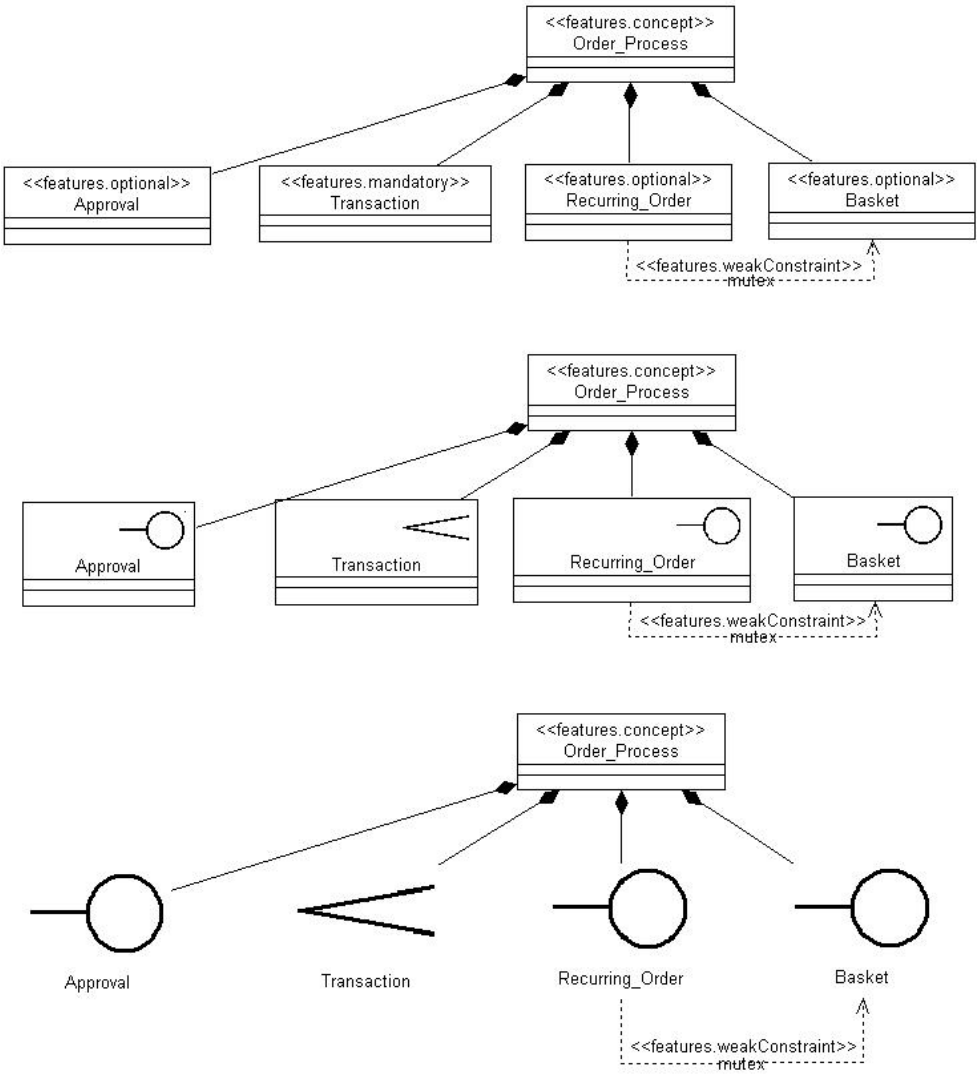

Abbildung 6.10: Sterotypen in allen drei Darstellungsvarianten.

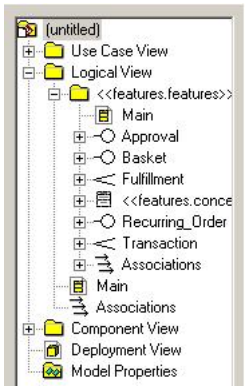

Abbildung 6.11: Die Stereotypen werden im Explorer-Fenster dargestellt.

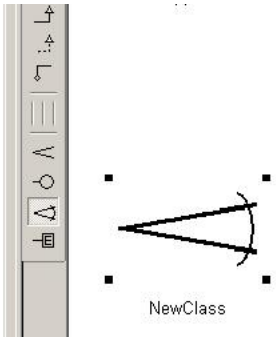

Abbildung 6.12: Die Sterotypen können der Werkzeugleiste hinzugefügt werden, wodurch direkt Instanzen erstellt werden können.

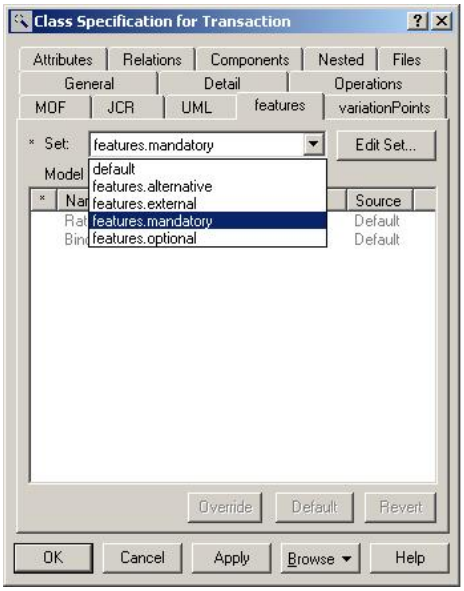

Abbildung 6.13: Für jedes Add-In ist eine Karteikarte mit Tagged Values vorhanden.

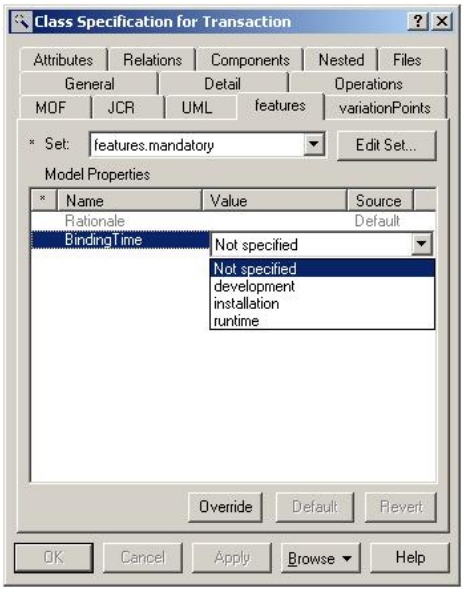

Abbildung 6.14: Aufzählungen aus dem Profile sind erhalten geblieben.

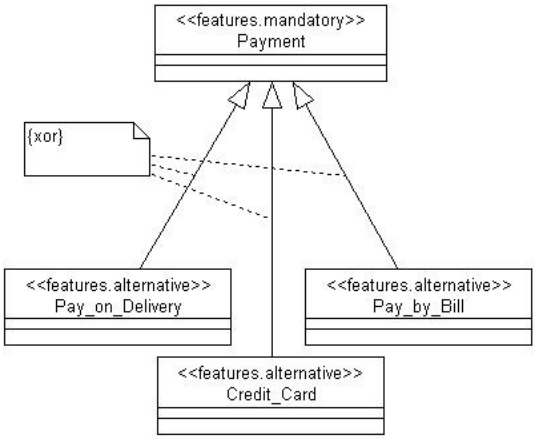

Abbildung 6.15: Constraints können manuell hinzugefügt werden.

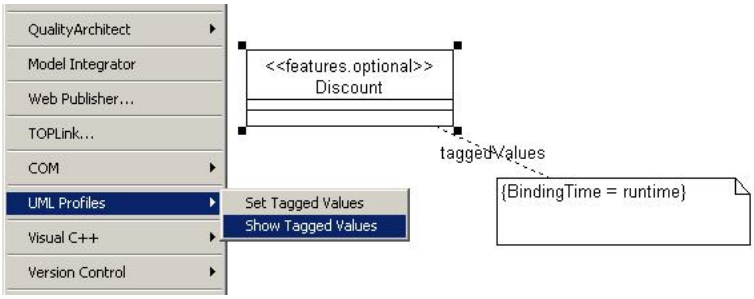

Abbildung 6.16: Dynamische Unterstützung zur Visualisierung von Tagged Values.

6.3 Bewertung

Zunächst werden die vorgestellten Konzepte anhand der Anforderungen aus Abschnitt 4.1 bewertet. Dadurch wird auf die einzelnen Details der Funktionalität eingegangen – eine Bewertung „im Kleinen“.

Ein zweiter Abschnitt bewertet die Eignung der Konzepte anhand der praktischen Anwendung auf das Profile für Variabilität.

In einem dritten Abschnitt erfolgt anhand eines Vergleichs mit der Profile-Unterstützung eines anderen UML-Werkzeugs – *Objecteering* von *Objecteering Software* [OBJ02a] – die abschließende Bewertung „im Großen“.

6.3.1 Vergleich mit den Anforderungen

Die Erstellung von Profiles ist kaum von der Funktionalität des REI abhängig, weshalb alle Anforderungen aus Abschnitt 4.1.1 erfüllt werden können. Die Anwendung von Profiles dagegen ist stark von den gegebenen Erweiterungsmöglichkeiten abhängig und bildet damit den problematischeren Teil und Schwerpunkt dieser Arbeit. Hier ist eine differenzierte Bewertung notwendig.

Das Basisgerüst aus Abschnitt 5.3.1.1 nutzt die Erweiterungsmechanismen von Rose wie vorgesehen und erfüllt dabei den größten Teil der Anforderungen. Für offen gebliebene Anforderungen wird in Abschnitt 5.3.3 versucht, mittels der dynamischen Funktionalität des REI Ersatzlösungen zu finden. Dabei konnte teilweise die geforderte Funktionalität gut in Rose integriert werden, so dass die jeweiligen Anforderungen als voll erfüllt betrachtet werden können. Dies gilt für:

- die Bindung von Tagged Values an Stereotypen (Abschnitt 5.3.3.1),
- die Unterstützung für Referenzen von Tagged Values (Abschnitt 5.3.3.3) und
- die Anwendung von Modellbibliotheken.

Für einen Teil der Anforderungen konnte mit den vorhandenen Möglichkeiten des REI keine vollständige Lösung gefunden werden. Dies gilt für

- die Visualisierung von Tagged Values (Abschnitt 5.3.3.2), da Tagged Values in Diagrammen nur in Kommentaren, nicht aber direkt in Modellelementen angezeigt werden können,
- die Unterstützung der gleichzeitigen Zuweisung mehrerer Stereotypen zu einem Modellelement (Abschnitt 5.3.3.5), vor allem da ein zweiter und weitere Stereotypen in Diagrammen nur in Kommentaren angezeigt werden können, sowie
- die Unterstützung der Multiplizität von Tagged Values (Abschnitt 5.3.3.4), da bei mehrwertigen Tagged Values der zweite und alle weiteren Werte frei und in eine zusätzliche Eigenschaft eingetragen werden müssen.

Weitere offene Anforderungen bestehen nicht.

Grundsätzliche Einschränkungen im Leistungsumfang von Rose wirken sich auf alle Anforderungen aus. Dies gilt bezüglich:

- der generell in Rose unterstützten Metaklassen,
- der Metaklassen, die in Rose mit Stereotypen versehen werden können und

- der Metaklassen, die in Rose mit Eigenschaften versehen werden können

Die Erweiterung von Rose um zusätzliche Metaklassen ist nicht möglich. Der hohe Verbreitungsgrad von Rose als Werkzeug zur Modellierung spricht jedoch dafür, dass Rose in der Regel alle im praktischen Einsatz benötigten Metaklassen abdeckt. Auch für die Anwendung von Stereotypen und Tagged Values sind nur in Einzelfällen Einschränkungen vorhanden. Ein solcher Fall sind Abhängigkeiten. Diesen können in Rose nur Stereotypen, nicht aber Tagged Values zugeordnet werden.

Zusammenfassend kann festgestellt werden, dass alle Anforderungen im wesentlichen als erfüllt betrachtet werden können. Die semantische Auswertung von Constraints wurde in dieser Arbeit von vorneherein ausgeklammert und ist daher nicht berücksichtigt.

6.3.2 Anwendung auf das Profile für Variabilität

Für alle Aspekte des Profiles für Variabilität, die durch die UML-Spezifikation abgedeckt sind, gilt die Bewertung des vorangegangenen Abschnitts 6.3.1, d. h. mit kleinen Einschränkungen kann das Profile fast vollständig verwendet werden. Alle benötigten Metaklassen – d. h. Base-Classes von Stereotypen und Tagged Values – sind in Rose vorhanden.

Nicht konform zum UML-Metamodell ist die wiederverwendbare Constraint `xor` aus dem Profile für Variabilität (siehe Abschnitt 3.2.1.2). In Abschnitt 3.2.1.2 wurde eine Lösung zur Integration wiederverwendbarer Constraints mittels eines Stereotypen vorgeschlagen. Diese, oder vergleichbare Lösungen, können in Rose jedoch nicht umgesetzt werden, da Rose prinzipiell Constraints nicht als selbständige Modellelemente unterstützt.

Bei den ersten Schritten der praktischen Anwendung des Variabilitäts-Profiles fällt auf, dass durch Anpassungen der Werkzeugumgebung (siehe Abschnitt 4.3.3) ein verbesserter Komfort bei der Anwendung des Profiles für Variabilität erreicht werden könnte. Ein Beispiel dafür sind Merkmalsdiagramme: durch Beispiele bei der Firma Intershop wird ersichtlich, dass eine sehr große Zahl an Merkmalen zu modellieren ist. In [Cla01] ist für diese Fälle eine Aufteilung des Merkmalsbaumes auf mehrere Diagramme vorgesehen. Durch die Vielzahl der erforderlichen Diagramme wäre jedoch hier für den Nutzer eine weitere Unterstützung durch das Werkzeug wünschenswert, z. B. die Möglichkeit einer Navigation in den Merkmalsbäumen.

In diesem Zusammenhang erweist sich besonders als Vorteil, dass mit Rose ein bereits weit entwickeltes Werkzeug verwendet wird. So findet sich unter der umfangreichen Funktionalität von Rose Zusatzfunktionalität, die für die Verwendung des Profiles für Variabilität nützlich ist. Ein Beispiel sind die Befehle *expandSelectedElements* und *hideSelectedElements* im Menü *Query*. Diese erlauben, ausgehend von einem Modellelement, bis zu einer vom Anwender festzulegenden Tiefe entlang Relationen zu navigieren und gefundene Modellelemente im Diagramm aus- bzw. einzublenden. Dabei kann festgelegt werden, welche Arten von Relationen berücksichtigt werden sollen. Dies erlaubt in einem Merkmalsbaum ganze Zweige ein- oder auszublenden. Es ist denkbar, mittels kleiner Rose-Skripte diese bestehende Funktionalität weiter an Merkmalsdiagramme anzupassen.

Das Konzept aus Kapitel 5, Add-Ins zu generieren, bietet auch hier Vorteile, da die generierten Add-Ins beliebig um zusätzliche Funktionalität erweitert werden können. Dadurch ist über die Möglichkeiten der UML-Spezifikation hinaus eine optimierte Anpassung – einerseits an den speziellen Anwendungsbereich und andererseits an Rose –

jederzeit möglich. Oft sind dazu lediglich kleine Rose-Skripte notwendig, die grundlegende Funktionalität aus Rose für den speziellen Anwendungsbereich optimieren. Die generierten Add-Ins erfordern keinerlei Einschränkungen im Funktionsumfang gegenüber Add-Ins, die per Hand erstellt wurden.

6.3.3 Vergleich mit Objecteering

Das UML-Werkzeug *Objecteering* von *Objecteering Software* [OBJ02a] (ehemals *Softeam*) bietet die bisher umfangreichste Werkzeugunterstützung für Profiles. In [Ple01] wurde die Version 5.02 ausführlich untersucht. Diese Untersuchung wird für einen Vergleich mit den in dieser Arbeit vorgestellten Konzepten herangezogen anhand dessen anschließend die Gesamtbewertung der aus dieser Arbeit resultierenden Profile-Unterstützung vorgenommen wird.

Objecteering unterstützt die UML-Version 1.3. Da UML 1.3 noch keine Definition von Profiles enthält (siehe Abschnitt 2.2) wurden zur Definition von Profiles auch Mechanismen aus UML 1.4 aufgegriffen. Folgende Defizite bezüglich der Umsetzung der UML-Spezifikation wurden ausgemacht:

- Stereotypen können bei bei der Erstellung von Profiles nicht als abstrakt definiert werden,
- Mehrfachvererbung von Stereotypen ist bei der Erstellung von Profiles nicht möglich,
- Modellelemente können bei der Anwendung von Profiles nicht mit mehreren Stereotypen versehen werden und
- Modellbibliotheken werden nicht unterstützt.

Die wesentlichste Auswirkung dieser Defizite ist, dass Stereotypen nur eingeschränkt strukturiert werden können. Das UML-Profile für CORBA [Obj02c] konnte daher nicht direkt in Objecteering abgebildet werden, da dieses beispielsweise Mehrfachvererbung von Stereotypen verwendet. Ansonsten sind in Objecteering alle Mechanismen aus der UML-Spezifikation bezüglich Profiles umgesetzt.

Bei der Profile-Unterstützung für Rose aus dieser Arbeit treten die aufgeführten Defizite von Objecteering nicht auf. Dafür sind andere kleinere Einschränkungen vorhanden (siehe 6.3.1). Objecteering unterstützt, ebenso wie Rose, Modelle nach UML 1.3. Sieht man von der Unterstützung von Modellbibliotheken ab, erfolgt die Umsetzung des Profile-Mechanismus aus der UML-Spezifikation in beiden Lösungen auf einem vergleichbar hohen Niveau. Dadurch, dass jedoch Modellbibliotheken in der untersuchten Version von Objecteering überhaupt nicht berücksichtigt sind, müssen insgesamt der Profile-Unterstützung von Rose Vorteile zugesprochen werden.

Ein wichtiger Aspekt von Objecteering ist die weitreichende Unterstützung von Funktionalität über die UML-Spezifikation hinaus im Zusammenhang mit Profiles (siehe Abschnitt 4.3.1). Dafür stellt Objecteering die Sprache *J* zur Verfügung sowie weitere weitreichende Strukturen und Konzepte. Solche Funktionalität berücksichtigen die Konzepte in dieser Arbeit nicht. Allerdings können die erzeugten Add-Ins jederzeit um beliebige Funktionalität erweitert werden, wodurch prinzipiell die gleichen Ergebnisse wie bei Objecteering erreichbar sind. Dadurch, dass in Rose-Add-Ins vollwertige OLE-Server in einer beliebigen OLE-fähigen Programmiersprache integriert werden können, ist prinzipiell sogar mehr Funktionalität als in Objecteering mittels J erreichbar. Für die

Implementierung der üblichen Aufgaben im Zusammenhang mit Profiles, wie z. B. Codegenerierung, reicht J jedoch völlig aus, weshalb Objecteering für derartige Aufgaben die deutlich komfortablere Unterstützung bietet.

Ein wichtiger Vorteil der in dieser Arbeit vorgestellten Erweiterung von Rose dagegen ist die vollständige Orientierung an Standards. Alle nach UML 1.4 definierten Profiles können direkt umgesetzt werden. Sowohl bei der Erstellung wie auch bei der Anwendung von Profiles wird das werkzeugunabhängige Austauschformat XMI unterstützt.

Zusammenfassend bieten beide Lösungen eine weitreichende Profile-Unterstützung, wobei jede Lösung spezielle Vorteile bietet. Bei Objecteering ist dies vor allem die komfortable und gut integrierte Unterstützung von zusätzlicher Funktionalität über die UML-Spezifikation hinaus. Das Konzept für Rose dagegen basiert vollständig auf Standards, ist anpassbar an zukünftige Standards und erlaubt prinzipiell die Erweiterung der generierten Profiles um jegliche zusätzliche Funktionalität.

Kapitel 7

Zusammenfassung und Ausblick

Zum Abschluß dieser Arbeit werden in einem ersten Abschnitt die erreichten Ergebnisse zusammengefasst. Da in absehbarer Zukunft vom Erscheinen einer neuen UML-Version – UML 2.0 – augegangen werden muss, soll in einem zweiten Abschnitt eingeordnet werden, inwieweit die Ergebnisse dieser Arbeit ihre Gültigkeit auch über UML 1.4 hinaus behalten. Schließlich wird in einem dritten Abschnitt ein Ausblick auf fortführende Aufgabenstellungen im Kontext dieser Arbeit vorgenommen.

7.1 Ergebnisse der Arbeit

In diesem Abschnitt werden die Ergebnisse, die in dieser Arbeit erreicht wurden, zusammengestellt.

In Kapitel 3 wurde die Darstellung von Profiles im Austauschformat XMI erarbeitet. Dabei werden – sowohl im allgemeinen (Abschnitt 3.1.2) als auch anhand des Profiles für Variabilität (Abschnitt 3.2.1) – eine Vielzahl von Problemen sichtbar, für die Lösungsvorschläge diskutiert werden. Resultierende Ergebnisse sind zum einen Forderungen nach Verbesserungen in UML 1.4 (Abschnitt 3.3.1.2) und zum anderen neue, zusätzliche optionale Anforderungen an den Profile-Mechanismus im Allgemeinen (Abschnitt 3.3.1.1). Ein weiteres Ergebnis ist die Feststellung der Eignung von XMI zum Austausch von Profiles, insbesondere da die in XMI integrierten Erweiterungsmöglichkeiten auch etwaige zukünftige Bedürfnisse (Zusatzinformationen für Werkzeuge in XMI-Dateien zu integrieren) voll abdecken können (Abschnitte 3.3.2.1 und 3.3.2.2). Schließlich resultiert aus diesem Kapitel auch ein vollständiges Beispiel für eine XMI-Repräsentation von Profiles (Abschnitt 3.2.2), das zukünftig als wertvolle Vorlage dienen kann, besonders da bisher keine vergleichbaren Beispiele vorhanden waren.

Auf Basis der generellen Grundlagen aus Kapitel 3 werden in Kapitel 4 Grundlagen für eine Werkzeugunterstützung für UML-Profiles geschaffen. Ergebnisse sind dabei obligatorische und optionale Anforderungen an Werkzeuge (Abschnitt 4.1). Diese sind allgemeingültig formuliert. Zusätzlich wird, soweit notwendig, jeweils die konkrete Bedeutung der Anforderung bezüglich UML 1.4 genannt. Als weiteres Resultat werden in Abschnitt 4.2 werkzeugunabhängig Schritte zur Integration einer Profile-Unterstützung in bestehende UML-Werkzeuge beschrieben. Dabei zeigt sich auch, dass einige Teilaufgaben – die Erstellung von Profiles und die Unterstützung von Modellbibliotheken – zu großen Teilen unabhängig vom bestehenden UML-Werkzeug sind.

Schließlich enthält Abschnitt 4.3 einen Überblick über die erweiterte Verwendung von Profiles.

In Kapitel 5 werden anhand des marktführenden Werkzeuges Rational Rose konkrete Konzepte zur Integration einer Profile-Unterstützung entwickelt. Ergebnis ist ein Werkzeug zur Erstellung von Profiles sowie ein Generator von Add-Ins zur Anwendung von Profiles in Rose. Die generierten Add-Ins werden zusätzlich mittels dynamischer Funktionalität in Rose unterstützt. Damit werden für alle Anforderungen und Teilaufgaben Konzepte vorgestellt werden, wobei zusätzlich eine Anpassbarkeit an zukünftige UML-Versionen erreicht wird.

Kapitel 6 stellt eine prototypische Implementierung vor. Die Beschreibung in Abschnitt 6.1 kann als ausführliche technische Dokumentation für mögliche Weiterentwicklungen dienen. In Abschnitt 6.2 wird anhand des Profiles für Variabilität die Verwendbarkeit der vorgestellten Konzepte nachgewiesen. Dieser Abschnitt dient zusätzlich als Anwenderdokumentation der Implementierung. Die abschließende Bewertung liefert als Ergebnis, dass Rose mittels der vorgschlagenen Konzepte fast alle Anforderungen an eine Profile-Unterstützung erfüllen kann. Für die wenigen Anforderungen, die nicht ganz vollständig erfüllt werden können, wird zumindest eine Ersatzlösung angeboten. Indem Add-Ins generiert werden, sind sie weiterhin prinzipiell um beliebige Funktionalität erweiterbar.

7.2 Nachhaltigkeit der Ergebnisse

In absehbarer Zukunft ist eine neue Version der UML – UML 2.0 – zu erwarten. Vor diesem Hintergrund sollen die Ergebnisse dieser Arbeit bezüglich ihrer nachhaltigen Gültigkeit, auch über UML 2.0 hinaus, eingeordnet werden.

Ein Teil der Ergebnisse dieser Arbeit sind konkret auf UML 1.4 bezogen. Dies sind

- das konkrete Beispiel für eine XMI-Repräsentation von Profiles in Abschnitt 3.2.2,
- die Anforderungen an die UML-Spezifikation aus Abschnitt 3.3.1.2 sowie
- die Beispiele zu den Anforderungen an eine Werkzeugunterstützung für Profiles in Abschnitt 4.1

Diese Ergebnisse werden für UML 2.0 nicht mehr anwendbar sein.

Als nachhaltig verwendbar werden folgende Ergebnisse eingeordnet:

- die generellen Anforderungen an den Profile-Mechanismus aus Abschnitt 3.3.1.1,
- die Bewertung von XMI als Austauschformat, insbesondere durch die integrierten Erweiterungsmöglichkeiten von XMI (Abschnitt 3.3.2),
- die Anforderungen an eine Werkzeugunterstützung von UML-Profiles (Abschnitt 4.1), da diese allgemeingültig formuliert wurden,
- die Schritte zur Integration einer Werkzeugunterstützung in bestehende Werkzeuge (Abschnitt 4.2), da diese unabhängig von einer konkreten UML-Version erarbeitet wurden,
- der generelle Überblick über erweiterte Verwendung von Profiles (Abschnitt 4.3) und

- die vorgestellten Konzepte für Rational Rose aus Kapitel 5, da diese von Beginn an auf Erweiterbarkeit für zukünftige UML-Versionen ausgelegt wurden.

Insgesamt sollte damit in der Arbeit nicht nur ein Konzept einer konkreten Profile-Unterstützung nach UML 1.4 für Rational Rose erreicht worden sein, sondern auch längerfristig nutzbare Grundlagen für zukünftige Arbeiten.

7.3 Zukünftige Aufgaben

Als Ausblick auf zukünftige Aufgaben wäre zunächst als praktische Aufgabe die Vervollständigung der in dieser Arbeit vorgestellten prototypischen Implementierung zu nennen. Die notwendigen Schritte dazu sind in Abschnitt 6.1.3 bereits vorgegeben.

Eine weitere Aufgabe wäre die semantische Auswertung von Constraints, die in dieser Arbeit explizit von vornherein ausgeklammert wurde. Dadurch, dass Constraints standardmäßig zu Stereotypen hinzugefügt werden können, sowie mit der OCL sind dafür bereits die wichtigsten Vorausetzungen erfüllt. Bezüglich Rose besteht in der Praxis die Schwierigkeit, dass das Metamodell von Rose nicht dem der UML entspricht. Um OCL-Constraints auswerten zu können, müsste demnach das UML-Metamodell auf das Rose-Metamodell abgebildet werden, wobei – in Gegensatz zu den Metaklassen-Abbildungen in dieser Arbeit – auch die einzelnen Attribute und Referenzen aller Metaklassen berücksichtigt werden müssen.

Ein großer und wichtiger Aufgabenkomplex ist die Umsetzung der weiteren erweiterten Funktionalität aus Abschnitt 4.3: die Transformation von Modellen und die Anpassung der Werkzeugumgebung. Dazu sind noch keine Vorgaben in Form von Standards vorhanden. Besondere Aufmerksamkeit gilt dabei der Generierung von Code, da diese einen wichtigen Teil der Nützlichkeit von UML-Werkzeugen ausmacht.

Ein möglicher Ansatz zur Codegenerierung könnte auf XMI basieren. Die XMI-Repräsentation eines Modells enthält – auch bei Rose – alle Stereotypen und Tagged Values, die Modellelementen hinzugefügt wurden. Diese könnten von einem separaten Codegenerator ausgewertet werden. Demnach könnte für jedes Profile ein oder mehrere Codegeneratoren – möglicherweise unter Nutzung der Sprache XSLT [Wor99] – erstellt werden, die werkzeugunabhängig aus XMI Code für den Anwendungsbereich des Profiles erzeugen. Dabei wäre zumindest im Fall von Rose auch ein Round-Trip-Engeneering möglich, wie das XMI-Toolkit (Abschnitt 5.1.3.3) bereits demonstriert. Für das Profile für EJB wurde in [Eik02] bereits ein solcher auf XMI basierender Codegenerator prototypisch implementiert. Wird für Rose eine Unterstützung für XMI 1.4 implementiert, könnte dieser ohne weitere Anpassungen im Zusammenspiel mit der Profile-Unterstützung von Rose aus dieser Arbeit genutzt werden.

Auch bei Intershop wird der Ansatz verfolgt, die XMI-Repräsentation der modellierten Variabilität als Basis für weitere Schritte im produktlinienorientierten Softwareeentwicklungsprozess zu nutzen.

Abbildungsverzeichnis

Literaturverzeichnis

[Bal98] BALZERT, H.: *Lehrbuch der Softwaretechnik (Bd. II). Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg : Spektrum Akademischer Verlag, 1998

[BGJ99] BERNER, S. ; GLINZ, M. ; JOOS, S.: A Classification of Stereotypes for Object-Oriented Modeling Languages. In: FRANCE, R. B. (Hrsg.) ; RUMPE, B. (Hrsg.): *UML'99: The Unified Modeling Language, Second International Conference, Fort Collins, CO, USA*. Berlin : Springer, Oktober 1999, S. 249–264

[Cla01] CLAUSS, M.: *Untersuchung der Modellierung von Variabilität in UML*, Technische Universität Dresden, Diplomarbeit, 2001

[CS02] CAPLAT, G. ; SOURROUILLE, J.: Model Mapping in MDA. In: J.-M. JÉZÉQUEL, S. C. (Hrsg.): *UML 2002 - The Unified Modeling Language 5th Int. Conference, Dresden, Deutschland*. Berlin : Springer, Oktober 2002

[Des00] DESFRAY, P.: *UML Profiles versus Metamodel Extensions: An ongoing debate*. OMG's First Workshop on UML In The .Com Enterprise. 2000. – www.omg.org/news/meetings/workshops/presentations/uml_presentations/5-3

[Des01] DESFRAY, P.: *Supporting the MDA Approach with UML Profiles*. OMG's Second Workshop on UML for Enterprise Applications. December 2001. – www.omg.org/news/meetings/workshops/presentations/uml2001_presentations/7-2_Desfray_UML_Profiles_for_MDA.pdf

[DSB99] D'SOUZA, D. ; SANE, A. ; BIRCHENOUGH, A.: First-Class Extensibility for UML - Packaging of Profiles, Stereotypes, Patterns. In: FRANCE, R. B. (Hrsg.) ; RUMPE, B. (Hrsg.): *UML'99: The Unified Modeling Language, Second International Conference, Fort Collins, CO, USA*. Berlin : Springer, Oktober 1999. – http://choices.cs.uiuc.edu/sane/uml-extend.pdf, S. 265–277

[Eik02] EIKENBERG, S.: *Generierung serverseitiger Komponenten basierend auf UML Profiles*, Technische Universität Dresden, Diplomarbeit, 2002

[GEN02] *Gentleware AG*. Hamburg, Deutschland, 2002. – http://www.gentleware.com

[GGW00] GIESE, H. ; GRAF, J. ; WIRTZ, G.: Präzisierung des UML-Metamodells durch ein semantisches Objektmodell. In: *GI Softwaretechnik Trends* 20 (2000), Mai, Nr. 2. – http://www.upb.de/cs/hg/archive/2000/GI-GROOM-7/gi-st-trends.pdf

[IBM02a] *IBM Corporation*. Armonk, NY, USA, 2002. – http://www.ibm.com

[IBM02b] IBM Corporation: *XMI Toolkit*. Juni 2002. – http://www.alphaworks.ibm.com/tech/xmitoolkit

[IDC02] *International Data Corporation (IDC)*. Framingham, MA, USA, 2002. – http://www.idc.com

[ISH02] *Intershop Software Entwicklungs GmbH*. Jena, Deutschland, 2002. – www.intershop.de

[Jec00a] JECKLE, M.: Konzepte der Metamodellierung - zum Begriff Metamodell. In: *Softwaretechnik Trends* 20 (2000), Mai, Nr. 2. – http://www.jeckle.de/files/swt_2000.pdf

[Jec00b] JECKLE, M.: Das vier-Schichten-Metamodell der UML. In: *GROOM Workshop 2002, Koblenz*, 2000. – http://www.jeckle.de/files/groom_20000404.pdf

[Jec02] JECKLE, M.: *UML Tools (CASE & Drawing)*. 2002. – http://www.jeckle.de/umltools.html

[Joh02] JOHNSTON, S.: *Rose Profile Documentor*. Version 0.3.2. Rational Software Corporation, Juni 2002. – http://www.rational.com/support/downloadcenter/addins/media/rose/RoseProfileDocumentor.ebs

[KBHM00] KOCH, N. ; BAUMEISTER, H. ; HENNICKER, R. ; MANDEL, L.: Extending UML to Model Navigation and Presentation in Web Applications. In: *Modeling Web Applications in the UML Workshop, UML2000, York, England* (2000). – http://www.pst.informatik.uni-muenchen.de/personen/kochn/ExtendingUML.pdf

[Lis00] LISKOWSKY, R.: *Vorlesung Softwareentwicklungswerkzeuge*. Technische Universität Dresden, Fakultät Informatik, 2000

[NDK99] N. DYKMAN, M. G. ; KESSLER, R.: Nine Suggestions for Improving UML Extensibility. In: FRANCE, R. B. (Hrsg.) ; RUMPE, B. (Hrsg.): *UML'99: The Unified Modeling Language, Second International Conference, Fort Collins, CO, USA*. Berlin : Springer, Oktober 1999, S. 236–248

[Obj99] Object Management Group: *Requirements for UML Profiles*. Version 1.0. Dezember 1999. – http://www.omg.org/cgi-bin/doc?ad/99-12-32

[Obj00a] Object Management Group: *Meta Object Facility (MOF) Specification*. Version 1.3. März 2000. – http://www.omg.org/cgi-bin/doc?formal/00-04-03

[Obj00b] Object Management Group: *Model Driven Architecture*. Draft 3.2. November 2000. – ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf

[Obj00c] Object Management Group: *OMG Unified Modeling Language Specification*. Version 1.3. März 2000. – http://www.omg.org/cgi-bin/doc?formal/00-03-01

[Obj00d] Object Management Group: *OMG XML Metadata Interchange (XMI) Specification*. Version 1.1. November 2000. – http://www.omg.org/cgi-bin/doc?formal/00-11-02

[Obj01a] Object Management Group: *Common Warehouse Metamodel 1.0*. Version 1.0. Oktober 2001. – http://www.omg.org/cgi-bin/doc?formal/01-10-01

[Obj01b] Object Management Group: *Developing in OMG's Model-Driven Architecture*. Revision 2.6. November 2001. – ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf

[Obj01c] Object Management Group: *Model Driven Architecture (MDA)*. White Paper. Juli 2001. – http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[Obj01d] Object Management Group: *OMG Unified Modeling Language Specification*. Version 1.4. September 2001. – http://www.omg.org/cgi-bin/doc?formal/01-09-67

[Obj01e] Object Management Group: *OMG XML Metadata Interchange (XMI) Specification*. Version 1.2. Januar 2001. – http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[Obj01f] Object Management Group: *UML 1.4 DTD*. Version 1.4. Februar 2001. – http://omg.org/cgi-bin/doc?ad/01-02-16

[OBJ02a] *Objecteering Software SA*. St Quentin-en-Yvelines, Frankreich, 2002. – http://www.objecteering.com

[Obj02b] Object Management Group: *Meta Object Facility (MOF) Specification*. Version 1.4. April 2002. – http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[Obj02c] Object Management Group: *UML Profile for CORBA Specification*. Version 1.0. April 2002. – http://www.omg.org/cgi-bin/doc?formal/02-04-01

[Obj02d] Object Management Group: *UML Profile for Enterprise Distributed Object Computing Specification*. Final Adopted Specification. Februar 2002. – http://www.omg.org/cgi-bin/doc?ptc/2002-02-05

[Obj02e] Object Management Group: *UML Profile for Schedulability, Performance, and Time Specification*. Final Adopted Specification. März 2002. – http://www.omg.org/cgi-bin/doc?ptc/2002-03-02

[OMG02] *Object Management Group (OMG)*. Needham, MA, USA, 2002. – www.omg.org

[OTR02] *Otris Software AG*. Dortmund, Deutschland, 2002. – http://www.otris.de

[PBG01] PELTIER, M. ; BÉZIVIN, J. ; GUILLAUME, G.: MTRANS: A general framework, based on XSLT, for model transformations. In: *WTUML: Workshop on Transformations in UML, Genova, Italien*, 2001. – http://ase.arc.nasa.gov/wtuml01/submissions/peltier-bezivin-guillaume.pdf

[Ple01] PLEUSS, A.: *Werkzeugunterstützung für UML Profiles*, Technische Universität Dresden, Großer Beleg, 2001

[Rat01a] Rational Software Corporation: *UML Profile For EJB*. Draft. Mai 2001. – http://jcp.org/aboutJava/communityprocess/review/jsr026/

[Rat01b] Rational Software Corporation: *Using the Rose Extensibility Interface*. 2001

[RAT02a] *Rational Software Corporation*, 2002. – http://www.rational.com

[Rat02b] Rational Software Corporation: *IDC Again Ranks Rational Software Number One in Multiple Segments of Software Development Market*. 2002. – http://www.rational.com/news/press/pr_view.jsp?ID=8314

[Rat02c] Rational Software Corporation: *Rational Rose Download Center*. 2002. – http://www.rational.com/support/downloadcenter/addins/rose/index.jsp

[Rat02d] Rational Software Corporation: *Rational Rose Online Help*. 2002

[Rat02e] Rational Software Corporation: *Rose Solution Nr. 18785*. 2002. – http://solutions.rational.com/solutions/display.jsp?solutionId=18785

[SUM02] *Summit Software Company*. Jamesville, NY, USA, 2002. – http://summsoft.com

[SW01] SCHLEICHER, A. ; WESTFECHTEL, B.: Beyond Stereotyping: Metamodeling Approaches for the UML. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Maui, Hawaii* Bd. 3. Los Alamitos, CA : IEEE Computer Society Press, January 2001. – http://www-i3.informatik.rwth-aachen.de/research/projects/dynamite/publications/HICSS2001New.pdf

[Tec02] Technische Universität Dresden: *Dresden OCL Toolkit*. 2002. – http://dresden-ocl.sourceforge.net/

[TOG02] *Togethersoft*. Raleigh, NC, USA, 2002. – http://www.togethersoft.com

[UNI02a] *Unisys*, 2002. – http://www.unisys.com/index.htm

[Uni02b] Unisys: *Unisys Rose XML Tools*. Version 1.3.4. Juni 2002. – http://www.rational.com/support/downloadcenter/addins/media/rose/UnisysRoseXMLTools1.3.4.zip

[W3C02] *World Wide Web Consortium (W3C)*, 2002. – http://www.w3.org

[Wor99] World Wide Web Consortium (W3C): *XSL Transformations (XSLT)*. Version 1.0. November 1999. – http://www.w3.org/TR/xslt

[Wor01a] World Wide Web Consortium: *XML Linking Language (XLink) Version 1.0*. Version 1.0 (Recommendation). Juni 2001. – http://www.w3.org/TR/2001/REC-xlink-20010627/

[Wor01b] World Wide Web Consortium (W3C): *Scalable Vector Graphics (SVG) 1.0 Specification*. Version 1.0. September 2001. – http://www.w3.org/TR/2001/REC-SVG-20010904/

[Wur02] WURBS, C.: *Modellierung .NET basierter Anwendungssysteme mittels der UML*, Technische Universität Dresden, Diplomarbeit, 2002

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

Dresden, den 31. Oktober 2002