Technische Universität Dresden

Fakultät Informatik

Institut für Software- und Multimediatechnik/
Softwaretechnologie

Großer Beleg

Werkzeugunterstützung für UML Profiles

Bearbeiter: Andreas Pleuß
Betreuer: Dipl.-Inform. Mike Fischer
Verantwortl. Hochschullehrer: Prof.Dr.rer.nat.habil. H. Hußmann

INHALT

1 Einleitung

Der Einsatz der Unified Modeling Language (UML) 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, so dass möglichst viele verschiedene Bereiche mit einer einheitlichen Notation modelliert werden können. Aufgrund dieser Allgemeinheit kann es Bereiche geben, in denen spezielle Eigenschaften nicht ausreichend deutlich durch UML in Modelle mit einbezogen werden können. Für solche Fälle stellt UML Erweiterungsmechanismen zur Verfügung, nämlich Stereotypes, Constraints und Tagged Values, durch die UML an spezielle Bedürfnisse angepasst werden kann. UML wird beschrieben durch ein Metamodell, das Teil einer ganzen Metamodellarchitektur ist. Durch Verwendung der in UML vorgesehenen Erweiterungsmechanismen wird erreicht, dass Erweiterungen an UML möglich sind, ohne das Metamodell selbst zu ändern.
Verschiedene einzelne Erweiterungen zur Anpassung an einen speziellen Bereich möchte man zusammenfassen können. Dieses Bedürfnis wird mit Hilfe von Profiles zu befriedigen versucht. Profiles sollen dazu dienen, zusammengehörige Erweiterungen zusammenzufassen und zu strukturieren. Die Idee des Profile-Mechanismus entstand zunächst unabhängig von UML für die gesamte Metamodellarchitektur, von der UML nur ein Teil ist. Für UML wurde diese Idee konkretisiert in Form von UML Profiles. Mittlerweile wird versucht, die Idee des UML Profiles immer vollständiger in die UML Spezifikation einzubinden. Dementsprechend wird auch die Forderung nach Werkzeugunterstützung von UML Profiles immer größer.
In dieser Arbeit soll der aktuelle Stand bezüglich UML Profiles sowie deren Unterstützung in Werkzeugen untersucht werden. Des Weiteren sollen für ein ausgewähltes CASE-Werkzeug Empfehlungen gegeben werden, wie es auf Unterstützung für UML Profiles hin zu erweitern sei.
Für Beispiele wird zunächst auf ein existierendes UML Profile, das UML Profile für CORBA [OMG00b], zurückgegriffen. Aufgrund des großen Umfangs eignet sich dieses aber, so wie auch andere existierende UML Profiles, nur bedingt. In diesen Fällen werden dann zur besseren Verständlichkeit eigene, übersichtlichere Beispiele verwendet.
In Kapitel 2 wird in UML Profiles und die damit zusammenhängenden Begriffe eingeführt. Hierbei wird zunächst von der OMG Metamodellarchitektur, die unter anderem das UML Metamodell enthält, ausgegangen und im Zusammenhang mit ihr Zweck und Motivation von Profiles verdeutlicht. Auf dieser Grundlage werden Definitionen für Profiles angegeben, zunächst eine Definition für Profiles im Allgemeinen, dann eine Definition konkretisiert und spezialisiert auf das eigentliche Thema der Arbeit, UML Profiles. Die UML Profile-Definition benennt die grundlegenden Bestandteile von UML Profiles, welche im Anschluss daran erläutert werden. Die Einführung schließt mit einer Auflistung der Anwendungsfälle von UML Profiles, wodurch auch der Nutzen von UML Profiles dem Leser nochmals verdeutlicht werden soll.
Kapitel 3 erläutert die Anforderungen, die an einen UML Profile-Mechanismus gestellt sind. Diese ergeben sich sowohl aus der allgemeinen Idee des Profiles, wie auch aus dem Ziel einer konkreten praktischen Nutzung in UML. Dabei wird unterschieden zwischen obligatorischen und optionalen Anforderungen. und weiterhin werden die Anforderungen in Gruppen geordnet.
In Kapitel 4 wird untersucht, ob und wie in den aktuellen Versionen der UML Spezifikation, Version 1.3 und 1.4 (Draft), UML Profiles eine konkrete Umsetzung gefunden haben. Die praktische Umsetzung in der neueren Version (1.4) wird dabei verglichen mit den

Anforderungen, die in Kapitel 3 zusammengestellt wurden. Die Festlegungen der Spezifikation sowohl zu Profiles selbst wie auch zu deren einzelnen Bestandteilen dienen als Basis zu Untersuchungen bezüglich der Werkzeugunterstützung.
In Kapitel 5 soll die bestehende Werkzeugunterstützung untersucht werden. Dies wird anhand eines Beispiels vorgenommen, dem CASE-Werkzeug „Objecteering“ von Softeam. Objecteering hebt sich bezüglich der Integration von UML Profiles von anderen CASE-Werkzeugen weit ab. Daher liefert es nicht nur ein gutes Beispiel für den aktuellen Stand der Entwicklung, sondern bietet auch Erkenntnisse bezüglich einer Erweiterung anderer Werkzeuge. An einem durchgehenden Beispiel wird konkret der Umgang mit UML Profiles in Objecteering untersucht und veranschaulicht. Dabei wird zunächst ein Profile in Objecteering erstellt und schließlich angewendet. Abschließend erfolgt eine zusammenfassende Bewertung.
Kapitel 6 wendet sich der Erweiterung eines CASE-Werkzeuges zu, das bisher noch kaum Unterstützung für UML Profiles bietet, dem Werkzeug „Together“ von Togethersoft. Es wird zunächst die Auswahl dieses Werkzeuges begründet und anschließend in das Werkzeug eingeführt, wobei sowohl ein Überblick über die allgemeine Funktionalität, als auch über den aktuellen Stand der Unterstützung von UML Erweiterungsmechanismen gegeben wird. Weiterhin wird gezeigt, welche grundlegenden Mechanismen Togethersoft zur Erweiterung von Together zur Verfügung stellt und wie diese zu verwenden sind. Auf dieser Grundlage werden Empfehlungen zur Erweiterung gegeben. Ein Teil dieser Empfehlungen wird in einer prototypischen Implementierung realisiert.

2 Einführung in UML Profiles

In diesem Kapitel soll in UML Profiles eingeführt werden. Dazu wird der Gesamtkontext, die OMG Vier-Schichten-Architektur, betrachtet und daran die allgemeine Motivation für Profiles begründet. Auf dieser Grundlage wird zunächst eine Definition für Profiles gegeben, die sich auf diesen Gesamtkontext bezieht.
Die Unified Modeling Language (UML) ist ein spezieller Teilbereich der Vier-Schichten-Architektur. Für diesen Teilbereich wird eine speziellere Profile-Definition angeführt, wodurch auf das eigentliche Thema dieser Arbeit, UML Profiles, hingeführt wird. Alle nachfolgenden Teile der Arbeit beziehen sich ausschließlich auf UML Profiles.
Anschließend an diese Definition werden die darin erwähnten Grundbestandteile von UML Profiles, Stereotypes, Constraints und Tagged Values, vorgestellt. Den Abschluss des Kapitels bildet eine Zusammenstellung der Anwendungsfälle für UML Profiles.

2.1 *Die Vier-Schichten-Architektur der OMG*

In diesem Abschnitt soll in den Gesamtkontext von UML Profiles, die Vier-Schichten-Architektur der OMG, eingeführt werden. Dabei handelt es sich um eine vierschichtige Metamodellarchitektur, weshalb eine Erläuterung des Begriffs des Metamodells vorangestellt ist.

2.1.1 Der Begriff des Metamodells

Als Grundlage dieses Kapitels dient [Jeck00]. Allgemein gewinnt das Konzept der Metamodellierung immer mehr an praktischer Relevanz, nicht zuletzt durch dessen Verwendung durch die OMG unter anderem auch für UML. Eine Metamodellarchitektur bietet einen Beitrag zum Verständnis einer verwandten Modellierungssprache, da zur Beschreibung der Modellierungssprache die dem Modellierer bekannten Grundprimitive verwendet werden, d.h. dass z.B. die Grundprimitive zur Syntaxbeschreibung des Modells identisch oder sehr ähnlich denen zur Diskursbeschreibung sein können. Dadurch ergibt sich ein Rationalisierungspotential bezüglich der Verständlichkeit und der Erlernbarkeit der Modellierungssprachenbeschreibung für den Modellierer. Weiterhin ergibt sich aus einer Metamodellarchitektur die Möglichkeit zu einer Validierung der Modelle, da die (semi-) formale Beschreibung durch ein Metamodell zur Prüfung von Konstrukten in der Modellierungssprache herangezogen werden kann.
Die Interpretation des Metamodellbegriffs unterliegt sehr starken Variationen. Gemein ist ihnen, was damit auch als allgemeine Aussage des Begriffs Metamodell betrachtet werden kann, dass es sich bei einem Metamodell um eine modellhafte Beschreibung eines Modells handelt. Die Unterschiede liegen in der Art und Weise dieser Beschreibung. Diese kann bezüglich unterschiedlicher Aspekte des Modells erfolgen. Dieser Aspekt der Metaisierung, das sogenannte Metaisierungsprinzip, entspricht der gewählten Abstraktionsebene über der Modellebene. Ein Beispiel dafür wäre eine dynamische Prozesssicht , die das Modell anhand eines Modellierungsprozesses beschreibt. Das Ergebnis dieses Prozesses wäre ein gültiges Modell.
Zumeist besteht in einer Metamodellarchitektur für ein Modell nur ein Metamodell, welches das Modell nach nur einem Aspekt beschreibt. Über dieses eine Metaisierungsprinzip wird in

Metamodellarchitekturen meist keine explizite Aussage getroffen. Zumeist dient als Metaisierungsprinzip die statische Struktursemantik, d.h. im Wesentlichen die Syntax der Modellierungssprache. Dies gilt auch für die Metamodellarchitektur der OMG, die Vier-Schichten-Architektur.

2.1.2 Die Vier-Schichten-Architektur der OMG

Das Kapitel orientiert sich im wesentlichen an [OMG99a].
Durch die Object Management Group (OMG, [INET01]) wurde eine Vier-Schichten-Architektur standardisiert (Abbildung 2-1). Dabei handelt es sich um eine vierschichtige Metamodellarchitektur zur strukturierten Beschreibung aller durch die OMG standardisierten Modellierungssprachen.

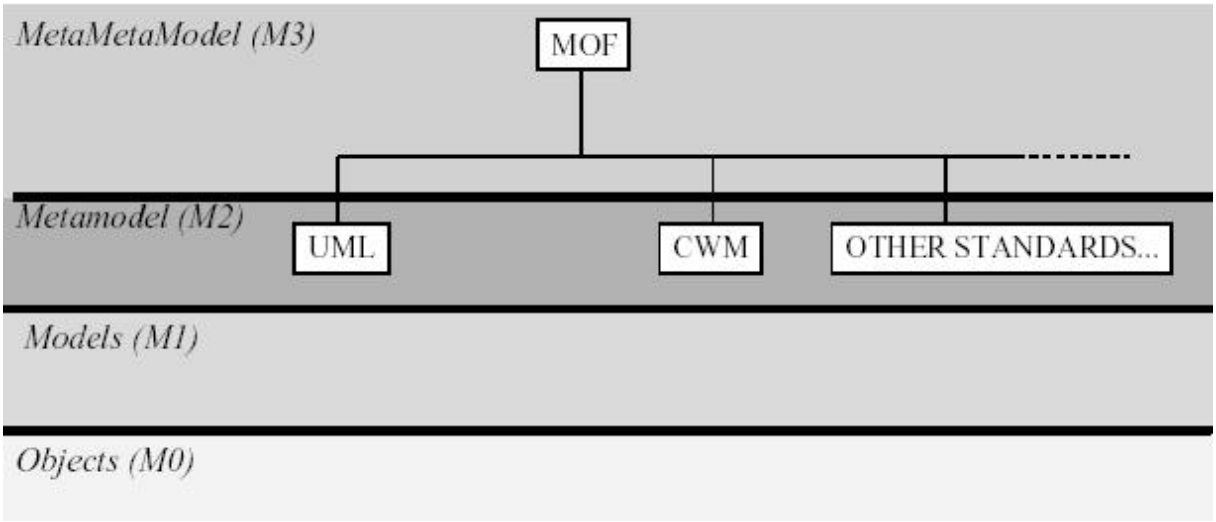

Abbildung 2-1: Die Vier-Schichten-Architektur der OMG (aus [OMG99a]).

Auf Schicht M1 befinden sich die Modelle, z.B. im Fall von UML alle UML Modelle. Darunter, auf Schicht M0, sind die Objekte angesiedelt, also konkrete Instanzen der Elemente aus den Modellen. Alle Modelle auf M1 werden beschrieben durch Metamodelle auf Schicht M2, welche von der OMG definiert und standardisiert werden. Ein Modellelement, also ein einzelnes Element auf Schicht M1, ist eine Instanz eines Metamodellelementes, also eines Elementes der Schicht M2. Metamodellelemente werden auch als „Metaklassen“ bezeichnet. Die Metamodelle wiederum werden beschrieben durch ein Meta-Metamodell, die MOF (Meta Object Facility), welche die Grundlage der Architektur bildet.
Das abgebildete Beispiel (Abbildung 2-2) zeigt ein konkretes Element aus jeder Schicht und deren Zusammenhang. Ein Element aus dem Meta-Metamodell ist *MetaClass*, welches alle Metaklassen in Metamodellen repräsentiert. Eine konkrete Instanz dessen ist z.B. *Class* im UML Metamodell. In einem UML Modell kann *Class* instanziiert werden, z.B. durch eine konkrete Klasse *Person*. Auf der Schicht der Objekte bzw. Daten wäre eine konkrete Instanz von *Person* z.B. die Person *Max Muster*.

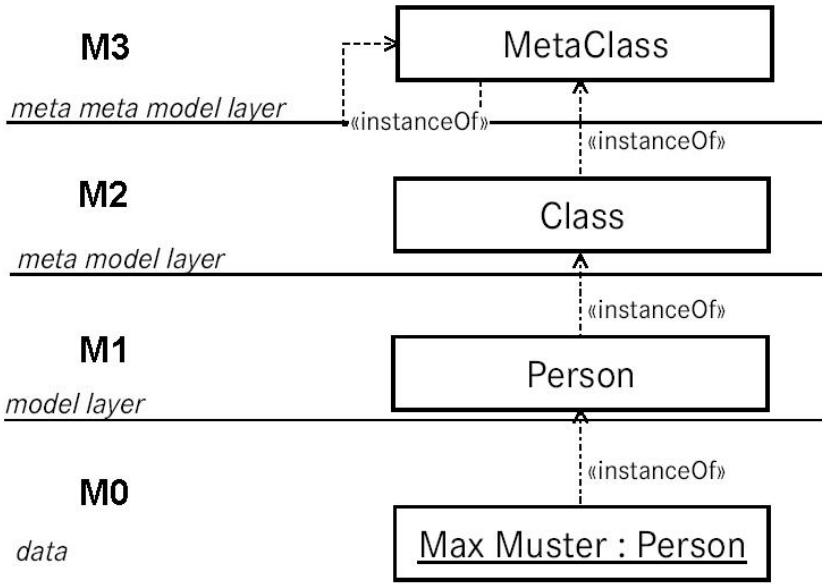

Abbildung 2-2: Beispiel konkreter Elemente der Schichten der Vier-Schichten-Architektur (aus [Jeck00])

Bezüglich UML Profiles sind besonders das UML Metamodell (auf M2) und die UML Modelle (auf M1) wesentlich.

2.2 *Erweiterung von Metamodellen*

Der Abschnitt orientiert sich an [OMG99b].
Die Metamodelle auf Schicht M2 decken meist größere Bereiche ab. Sie lassen sich in verschiedene engere Unterbereiche gliedern, welche Spezialisierungen der allgemeineren „Elternbereiche“ darstellen.
So handelt es sich im Falle von UML um eine Modellierungssprache, die ein weites Spektrum von Anwendungsfällen abdeckt und generelle Konzepte zur Modellierung zur Verfügung stellt. Unterbereiche können z.B. Modellierung von Echtzeitanwendungen (RealTime) oder EDOC sein (siehe auch „Anwendungsfälle für UML Profiles“, Kapitel 2.6)
Für diese Unterbereiche wäre es außerordentlich nützlich, Erweiterungen oder Spezialisierungen des Metamodells zu erlauben, um spezifische Konzepte oder Techniken des Unterbereichs genauer und verfeinerter wiedergeben zu können. So möchte man z.B. bei der Modellierung von Echtzeitanwendungen Zeitbedingungen im Modell ersichtlich machen können.
Für die daher benötigten Erweiterungen eines Metamodells gibt es zwei prinzipielle Möglichkeiten: zum einen die Verwendung von sogenannten „Heavyweight Extensions“, d.h. die Erweiterung des Metamodells selbst, z.B. durch das Hinzufügen von neuen Metaklassen. Im Gegensatz dazu stehen „Lightweight Extensions“, die ein Metamodell für einen bestimmten Unterbereich verfeinern bzw. spezialisieren, ohne die Semantik des Metamodells zu verändern. Dies geschieht durch die Verwendung von im Metamodell bereits vorgesehenen Erweiterungsmechanismen, im Fall von UML sind dies Stereotypes, Constraints und Tagged Values (siehe Kapitel 2.5). Dadurch, dass bei der Verwendung von Lightweight Extensions das Metamodell nicht verändert wird, können CASE-Werkzeuge erweiterte Modelle austauschen, ohne das Metamodell und das zugehörige Verhalten des Werkzeuges dynamisch ändern zu müssen. Weiterhin kann dadurch auch ein Nutzer die Semantik der Erweiterungen stets im Kontext des unverändert bleibenden Metamodells interpretieren.

Um für einen Unterbereich die einzelnen Lightweight Extensions zusammenzufassen, soll der Mechanismus der Profiles dienen. Ein Profile soll eine Einheit bilden, die die Erweiterungen für einen bestimmten Bereich repräsentiert und strukturiert und damit das Metamodell spezialisiert. Die verschiedenen Unterbereiche können selbst wiederum spezialisiert werden und somit auch die Profiles. Es ist vorgesehen, dass Profiles sowohl von der OMG standardisiert werden, als auch von Endnutzer für persönliche Bedürfnisse erstellt werden können, wobei die Standard-Profiles Bereiche, in denen sich allgemein die Notwendigkeit einer Spezialisierung gezeigt hat, betreffen dürften, während Profiles von Endanwendern auch firmen- und projektspezifische Bedürfnisse abdecken können und damit optimalen Umgang mit individuellen Techniken und Anwendungsbedürfnissen ermöglichen.
Abbildung 2-3 zeigt, wie Profiles somit in die Vier-Schichten-Architektur einzuordnen sind. Dabei ist zu beachten, dass ein durch den Endanwender erstelltes Profile nicht zwangsläufig ein anderes Profile spezialisieren muss.

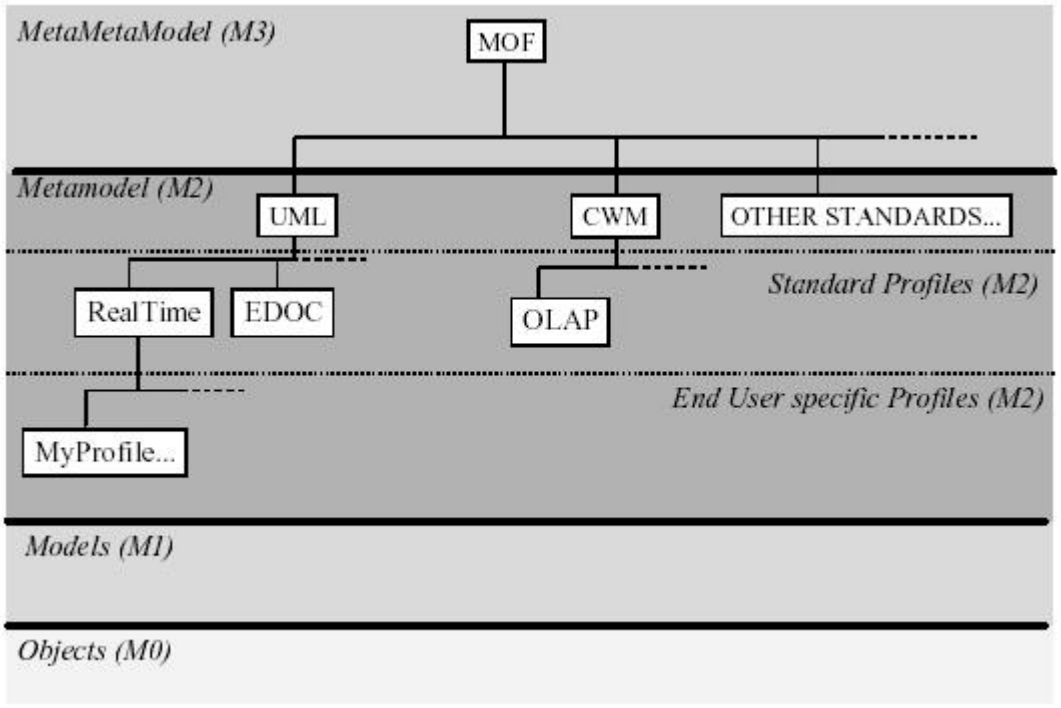

Abbildung 2-3: Profiles in der OMG Vier-Schichten-Architektur (aus [OMG99a]).

2.3 *Definition von Profiles*

Das „White Paper of the Profile mechanism“ ([OMG99a]) enthält eine allgemeine, prinzipielle Definition von Profiles, die hier in eigenen Worten widergegeben wird:

Ein Profile spezialisiert ein oder mehrere Standard-Metamodelle, d.h. Metamodelle, die von der OMG standardisiert wurden. Es bezieht sich darauf auf einen bestimmten Unterbereich des Standard-Metamodells. Im Falle des UML Modellierung sind z.B. RealTime oder Business Modeling spezielle Unterbereiche. Ein Profile für RealTime oder Business Modeling würde diesen speziellen Unterbereich dem UML Metamodell unterordnen.
Profiles können für jedes Standard-Metamodell, das gemäß der MOF definiert wurde, festgelegt werden. Da Profiles Metamodelle spezialisieren, ermöglichen sie flexible Anpassungen eines Standards, während der Standard selbst unverändert bleiben kann.

Profiles können strukturiert werden, um die einzelnen Unterbereiche, die sie repräsentieren, zu ordnen. Auch die Spezialisierung oder Kombination von Profiles ist möglich.
Ein Profile ist ein konsistenter Kontext zur Definition von Elementen wie well-formedness rules, Tagged Values, Stereotypes, Constraints, in natürlicher Sprache formulierte Semantik, Erweiterungen des Standard-Metamodells und Transformationsregeln. Die Elemente in einem Profile sind gemäß MOF bzw. dem Standard-Metamodell zu spezifizieren. Wenn z.B. Heavyweight Extensions unterstützt werden sollen, dann durch Profiles.

2.4 *Definition von UML Profiles*

Wie aus den bisherigen Ausführungen hervorgeht, sind Profiles prinzipiell für alle Metamodelle denkbar. Im Wesentlichen konzentrieren sich jedoch alle weiteren Entwicklungen bezüglich Profiles zunächst auf das UML Metamodell, das auch die größte Praxisrelevanz besitzt, und somit auf UML Profiles. UML Profiles sind auch der Gegenstand dieser Arbeit, weshalb im Folgenden die Betrachtungen immer auf UML Profiles beschränkt werden sollen.
Für die Verwendung von Profiles in UML wird eine konkretere und auf UML zugeschnittene Definition von UML Profiles als die allgemeine Definition aus Abschnitt 2.3 benötigt. Da in der UML Spezifikation 1.3 [OMG00a] UML Profiles nur rein informativ beschrieben sind (siehe Kapitel 4.1), wurde in Arbeiten zur Standardisierung konkreter Profiles, wie z.B. dem UML Profile für CORBA ([OMG00b]), eine vorläufige Arbeitsdefinition von UML Profiles verwendet. Diese wurde auch von der OMG in [OMG99b] als vorläufig allgemein gültige Arbeitsdefinition aufgenommen. Sie ist nicht als vollständig und endgültig zu betrachten, soll aber bis auf weiteres als Basis für die Entwicklung von Profiles dienen und Leitlinie für künftige (siehe auch Umsetzung von UML Profiles in UML 1.4 in 4.2) UML Profile-Definitionen sein.
Dazu sollen zwei Begriffe erläutert werden, die aus der UML Spezifikation (z.B. [OMG00a]) stammen, und auch in anderen Dokumenten Verwendung finden:
Standardelemente („standard elements“) sind standardisierte Stereotypes, Constraints und Tagged Values. In der UML Spezifikation werden Standardelemente zur Definition von Elementen verwendet, die nicht eigenständig oder komplex genug sind, um als eigenes Metamodellelement definiert zu werden.
Well-Formedness Rules sind Constraints, die erfüllt werden müssen, damit ein Modell als wohlgeformt gelten darf. Sie sind in der Object Constraint Language (OCL) der UML formuliert und tragen zur Definition von Metamodellelementen bei.

Die Arbeitsdefinition von UML Profiles wird hier in eigenen Worten widergegeben:

Ein UML Profile ist eine Spezifikation, die einen oder mehrere der folgenden Punkte erfüllt:

- Angabe einer Untermenge des UML Metamodells, d.h. einer Menge von Elementen aus dem UML Metamodell. Dies kann auch das gesamte UML Metamodell sein.
- Spezifikation von zusätzlichen Well-Formedness Rules für die angegebene Untermenge.
- Spezifikation von zusätzlichen Standardelementen für die angegebene Untermenge.
- Spezifikation von zusätzlicher Semantik für die angegebene Untermenge, ausgedrückt in natürlicher Sprache oder jeder anderen angemessenen Form.
- Spezifikation allgemeiner, wiederverwendbarer Modellelemente (d.h. Instanzen von UML Konstrukten), beschrieben mit der Terminologie des Profiles.

2.5 *Bestandteile von UML Profiles*

An dieser Stelle soll ein Überblick über die Hauptbestandteile eines UML Profiles, die Lightweight Extensions, gegeben werden. Dabei handelt es sich um die in UML bereits enthaltenen Erweiterungsmechanismen Stereotypes, Constraints und Tagged Values. Sie werden von einem UML Profile für einen bestimmten Anwendungsbereich zusammengefasst.
Eine detailliertere Beschreibung und Erläuterung von Stereotypes, Constraints und Tagged Values und ihrer versionsspezifischen Eigenschaften erfolgt in Kapitel 4, hier sollen nur die Prinzipien verdeutlicht werden.

2.5.1 Stereotypes

Stereotypes sind der wichtigste Erweiterungsmechanismus. Sie bieten die Möglichkeit, ein Modellelement zu klassifizieren oder zu markieren, und ihm damit zusätzliche Bedeutung oder Eigenschaften zuzuweisen.
Für jedes Stereotype ist festgelegt, welche Art von Modellelementen, also welche Metaklasse wie *Class* oder *Association*, mit dem Stereotype versehen werden können. Diese Metaklasse wird als „Base Class“ des Stereotypes bezeichnet.
Einem Stereotype können Constraints und Tagged Values zugeordnet sein. Diese nehmen dann Bezug auf die Base Class, wodurch den Modellelementen, die mit dem Stereotype versehen sind, zusätzliche Semantik zugewiesen werden kann. Generell darf aber durch ein Stereotype niemals die ursprüngliche Semantik des Modellelements, also die der Base Class, verletzt werden, sondern immer nur spezialisiert oder verfeinert.
Das mit dem Stereotype versehene Modellelement kann als Instanz einer „virtuellen Metaklasse“ betrachtet werden, da einerseits durch das Stereotype neue Semantik hinzugekommen ist, andererseits Stereotypes auf Modellebene (Schicht M1) instanziiert werden, wodurch das Metamodell völlig unberührt bleibt.
Ein Stereotype kann zusätzlich auch eine graphische Repräsentation für die virtuelle Metaklasse in ein Modell einführen.
Außer der Möglichkeit, durch Tagged Values und Constraints einem Modellelement zusätzliche Eigenschaften und Bedingungen zuzuordnen, können Stereotypes auch dazu verwendet werden, um einfach einen Unterschied in der Bedeutung zu kennzeichnen.
Im Beispiel in Abbildung 2-4 ist der Stereotype *CORBAInterface* einer Klasse *User* zugewiesen und zur Spezialisierung der Metaklasse *Class*. User ist somit nicht eine Instanz einer *Class*, sondern Instanz der virtuellen Metaklasse *CORBAInterface*. Das Modellelement soll also nicht als irgendeine Klasse implementiert werden, sondern als ein CORBA-Interface. Der Stereotype ist dem UML Profile für CORBA [OMG00b] entnommen.

Abbildung 2-4: Beispiel für einen Stereotype

2.5.2 Tagged Values

Bei einem Tagged Value handelt es sich um ein Paar aus einem Schlüssel, dem Tag, und einem Wert. Der Anwender kann dadurch einem Modellelement eine beliebige Eigenschaft zuweisen. Ist der Tagged Value einem Stereotype zugeordnet, wird er auf die Modellelemente angewandt, die mit dem Stereotype versehen sind. Er ist dann anzusehen als ein virtuelles Metaattribut der virtuellen Metaklasse, die das Stereotype repräsentiert.

Abbildung 2-5: Beispiel für einen Tagged Value

Abbildung 2-5 zeigt einen Tagged Value aus dem UML Profile für CORBA [OMG00b], der auf das CORBA-Interface *User* angewandt wird. Er besteht aus dem Tag *isLocal* und hat den Wert *TRUE*, wodurch in diesem Beispiel ausgesagt werden soll, dass es sich um ein lokales CORBA-Interface handelt. Der Tagged Value ist dem Stereotype zugeordnet und wird daher auf *User* angewendet, weil *User* mit dem Stereotype versehen ist. Dies ist aber aus dem Beispiel nicht zu entnehmen; theoretisch könnte der Tagged Value auch dem Modellelement *User* direkt zugeordnet worden sein.

2.5.3 Constraints

Constraints formulieren Regeln und Bedingungen, welche einem oder mehreren Modellelementen zugewiesen werden können. Sie enthalten einen Booleschen Ausdruck in beliebiger Sprache, der immer erfüllt werden muss. Analog zum Tagged Value können Constraints auch einem Stereotype zugeordnet sein und werden damit auf die Modellelemente angewandt, die mit dem Stereotype versehen sind.

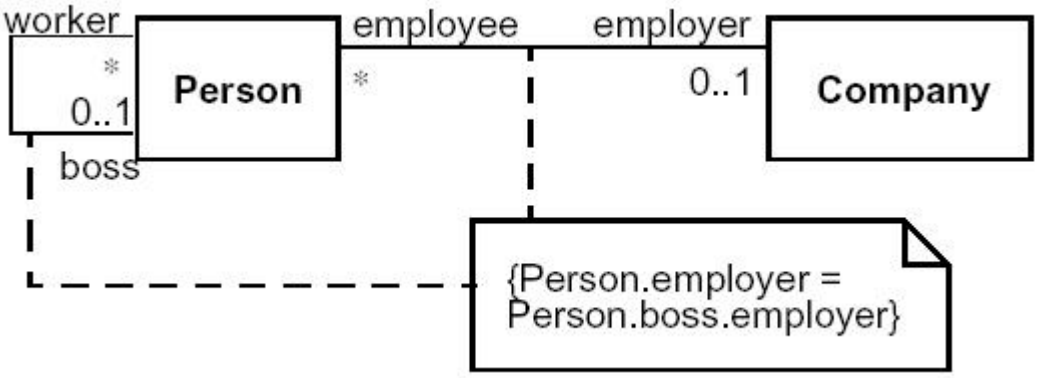

Abbildung 2-6: Beispiel für eine Constraint, die zwei Assoziationen zugeordnet wurde.

Das Beispiel in Abbildung 2-6 zeigt eine Constraint, die zwei Modellelementen, Assoziationen, zugeordnet ist. Sie legt fest, dass zu jeder Person die Werte *Person.employer* und *Person.boss.employer* (falls vorhanden) stets gleich sein müssen. Eine Constraint aus

dem UML Profile für CORBA [OMG00b] für das Stereotype *CORBAInterface* wäre beispielsweise: *In einer Klasse, die mit dem Stereotype CORBAInterface versehenen ist, haben alle Attribute die Sichtbarkeit „public“*.

2.6 *Anwendungsfälle für UML Profiles*

Nach [OMG99b] sind ohne Anspruch auf Vollständigkeit UML Profiles primär für folgende Anwendungsfälle vorgesehen:

1. Spezialisierung der UML für die technische Implementierung:

   Soll im Hinblick auf eine technische Zielumgebung, in der das Modell implementiert werden soll, modelliert werden, so bietet sich die Verwendung von UML Profiles an. Dies gilt für jegliche Art von technischen Umgebungen, wie zum Beispiel eine bestimmte Programmiersprache, Middleware oder Datenbank. Dazu zählt beispielsweise das UML Profile für CORBA [OMG00b], das bereits von der OMG standardisiert wurde, oder das UML Profile für Enterprise Java Beans, dessen Standardisierung ebenfalls angestrebt wird. Weiterhin naheliegend sind zum Beispiel UML Profiles für Java oder C++, für Relationale Datenbanken oder auch für eine spezielle Datenbank wie ORACLE.
   Durch UML Profiles können die spezifischen Notationen der jeweiligen Zielumgebung in Modelle eingeführt werden, wie zum Beispiel „virtual method“, „identifier“, „not null“. Auch bringen technische Umgebungen meist Beschränkungen mit sich, zum Beispiel ist in Java keine Mehrfachvererbung erlaubt. Diese zusätzlichen Regeln können durch ein UML Profile mit Hilfe von Constraints ebenfalls in ein UML Modell eingebracht werden.

2. Spezialisierung der UML für generelle technische Bereiche:

   Auch Bereiche mit generellen, nicht direkt implementierungsabhängigen technischen Gegebenheiten oder Anforderungen sind ein Anwendungsgebiet für UML Profiles. Dazu kann man beispielsweise Enterprise Distributed Object Computing (EDOC) oder Real Time zählen. Auch hier wird zusätzliche Semantik oder Bedingungen benötigt, zum Beispiel bei Real Time, um Zeitanforderungen in UML Modellen ausdrücken zu können.

3. Spezialisierungen der UML für spezielle Anwendungsgebiete:

   Spezielle Anwendungsbereiche, wie zum Beispiel der Finanzsektor oder Fertigungsbereiche, stellen ebenfalls bestimmte Anforderungen an UML Modelle. Oft jedoch ist hierfür jedoch eine vordefinierte Bibliothek von Modellen oder Templates ausreichend. In solchen Fällen sollen nicht UML Profiles verwendet werden. In einigen Fällen ist jedoch eine Erweiterung von UML notwendig. So benötigen z.B. Anwendungen aus dem geographischen Bereich eine Notation, um geographische Beziehungen zwischen Elementen ausdrücken zu können. In diesen Fällen sind UML Profiles zu verwenden.

4. Spezialisierungen der UML zur Modellierung von Softwareentwicklungsprozessen:

   In verschiedenen Phasen eines Softwareentwicklungsprozesses wird UML unterschiedlich verwendet. Mit Hilfe von UML Profiles kann der Kontext des Entwicklungsprozesses kenntlich gemacht werden. Beispielsweise kann durch ein UML Profile festgelegt werden, dass in der Analysephase nicht die Sichtbarkeit von Attributen anzugeben ist oder jeder Actor nur mit einem Use Case interagieren soll.
   Weiterhin kann auch der Softwareentwicklungsprozess selbst durch ein Profile spezifiziert werden, zum Beispiel „Analyse“ gefolgt von „Design“.

3 Anforderungen an UML Profiles

In diesem Kapitel werden die Anforderungen, die sich an einen Profile-Mechanismus in UML stellen, aufgeführt. Diese ergeben sich zum einen aus dem grundsätzlichen Prinzipen für Profiles und zum anderen aus den konkreten Gegebenheiten der UML. Dabei wird grundlegend unterschieden zwischen obligatorischen Anforderungen, deren Erfüllung für eine in sich abgeschlossene Realisierung eines UML Profile-Mechanismus unerlässlich sind, und optionalen Anforderungen, deren Erfüllung für einen praktischen Einsatz wünschenswert, aber nicht zwingend ist. Gruppiert werden die Anforderungen auf ihren Bezug hin, was aber nur der Übersichtlichkeit dienen soll, und keine weitere Bedeutung hat.
Die aufgeführten Anforderungen an UML Profiles orientieren sich an den Vorgaben der OMG in [OMG99b].

3.1 *Obligatorische Anforderungen*

Funktion:

1. Spezialisierung der UML Standard-Semantik
   Ein UML Profile muss Mechanismen zur Spezialisierung des UML Metamodells in der Form bieten, dass die Semantik des Standard- Metamodells nicht verletzt wird. Dazu werden typischerweise „well-formedness rules“ verwendet, die zusätzlich zu erfüllende Bedingungen einführen, welche jedoch konsistent zum Metamodell sein müssen. Es soll möglich sein, jegliche weitere spezielle Semantik zu definieren, entweder formal (z.B. in OCL) oder informell in natürlicher Sprache.

Inhalt:

2. Lightweight Extensions
   Ein Profile soll keine anderen Erweiterungsmechanismen als die in der UML vorgesehenen, d.h. Stereotypes, Constraints und Tagged Values, verwenden.

3. Anwendbare Untermenge
   Ein Profile muss die anwendbare Untermenge des UML Metamodells, d.h. die Elemente, die durch das Profile erweitert werden, definieren können.

4. Referenzen auf Modellbibliotheken
   Ein Profile muss Anwendungsbereichsspezifische UML Bibliotheken referenzieren können, in welchen manche Modellelemente vorgegeben sind. Dies trifft zu für IDL, C++, Java und andere technische Bereiche. Z.B. ein UML Profile für CORBA benötigt eine Referenz zu den vorgegebenen grundlegenden IDL Typen, die in UML als Package von Datentypen definiert werden können. Weitere Beispiele für Bibliotheken sind Packages, die „patterns“ oder wiederverwendbare „business object classes“ enthalten.

Umgebende Einheit:

5. Zusammenbinden von UML Profiles und Modellbibliotheken
   Es soll ermöglicht werden, eine UML Erweiterung zu definieren, die Profiles und Modellbibliotheken in einer einzigen logischen Einheit zusammenbindet. Innerhalb dieser Einheit soll es dabei aber möglich sein, die einzelnen Komponenten immer noch unterscheiden zu können (z.B. in separaten Packages gespeichert mit Relationen zwischen ihnen).

Anwendung:

6. Angabe gewählter UML Profiles für Packages
   Es soll ermöglicht sein, für ein gegebenes Package oder Unter-Package die gewählten Profiles anzugeben. Dies ist damit dann auch für ein gegebenes Model oder Subsystem möglich, da diese im UML Metamodell Unterklassen von Package sind. Beim Austausch von Modellen kann dadurch die importierende Umgebung das Modell korrekt interpretieren.

Austausch:

7. Verwendung bereits existierender Austauschmechanismen
   Es muss ermöglicht sein, unter Verwendung bereits existierender UML XMI (oder CORBA) Austauschmechanismen, Profiles zusammen mit Modellen zwischen Werkzeugen auszutauschen. Ein Profile muss daher also als ein austauschbares UML Modell definierbar sein; z.B. sollte es seine Elemente in Packages einbinden und Beziehungen zwischen Profiles sollen durch UML Beziehungen, wie Namensräume, Abhängigkeiten und Vererbung ausgedrückt werden.

8. Anwendung „by reference“
   Zusätzlich zum Austausch zusammen mit Modellen zwischen Werkzeugen sollen Profiles „by reference“ angewendet werden (z.B. durch „import by name“), wodurch ein Profile, das bereits im importierenden Werkzeug vorhanden ist, nicht mit übergeben werden muss.

Operationen:

9. Profile-Spezialisierung:
   Um Beziehungen zwischen UML Profiles bilden zu können, soll die Operation der Spezialisierung auf UML Profiles durchführbar sein. Ein neues UML Profile soll aus einem bestehenden UML Profile durch Spezialisierung der Semantik so abgeleitet werden können, dass die Semantik des bestehenden „Eltern-Profiles“ übernommen wird, ohne verletzt zu werden.

10. Profile-Komposition:
    Um Beziehungen zwischen UML Profiles bilden zu können, soll die Operation der Komposition auf UML Profiles durchführbar sein. Eines neue UML Profile soll durch Kombination zweier oder mehrerer gegenseitig kompatibler UML Profiles generiert werden können. Zumindest soll Konjunktion („UND“-Verknüpfung) zweier Profiles möglich sein.

Tagged Values:

11. Formalisierung der Definition von Tagged Values mit Typangabe
    Die Definition von Tagged Values soll formalisiert werden. Es sollen Typangaben gemacht werden, wobei als Typ jeder UML Datentyp, entweder einer der standardmäßig enthaltenen oder ein benutzerdefinierter, möglich sein soll.

12. Mehrwertige Tagged Values
    Es sollen sowohl ein- als auch mehrwertige Tagged Values unterstützt werden. Unter „mehrwertig“ ist zu verstehen, dass einem Tag mehrere einzelne Werte zugeordnet werden.

3.2 *Optionale Anforderungen*

Stereotypes:

1. Einem Modellelement mehrere Stereotypes zuweisbar
   Es sollen mehrere Stereotypen einem Modellelement zugewiesen werden können. Dadurch entfällt die Notwendigkeit für den Anwender, Stereotypen zu definieren, die von mehreren benötigten Stereotypen erben, um diese einem Modellelement zuzuweisen.

2. Graphische Notation für Definition von Stereotypes
   Eine Konvention der Notation für die graphische Stereotypendefinition sollte Teil eines Profiles sein.

Tagged Values:

3. Referenzieren von Metaklassen
   Tagged Values sollen Standard UML Metaklassen referenzieren können (ähnlich wie Stereotypen durch „BaseClass: Name“), wodurch das Metamodell durch gerichtete „Metarelationen“ erweitert werden kann, ohne die Standard UML Semantik zu verletzen.

Spezialisierung:

4. Generelle Spezialisierung von Metaklassen
   Ein Profile soll die Semantik von Standard UML Metamodellelementen generell spezialisieren können; z.B. soll ein Profile für Java Mehrfachvererbung verbieten können, ohne dass jeder einzelnen Klasse in einem Modell etwa ein Stereotype <<Java class>> explizit zugewiesen werden muss.

4 UML Profiles in der UML Spezifikation

In diesem Kapitel soll für die Versionen 1.3 und 1.4 der UML Spezifikation untersucht werden, in wie weit UML Profiles in die Spezifikation Eingang gefunden haben bzw. wie die Vorgaben bezüglich UML Profiles in ihnen umgesetzt wurden.

4.1 *UML Spezifikation 1.3*

UML 1.3 enthält keine weitreichende Unterstützung von Profiles. Von Belang sind hier hauptsächlich Festlegungen zu Stereotypes, Constraints und Tagged Values. Diese sind im UML Metamodell in einem Package *Extension Mechanisms* enthalten, das hier beschrieben wird. Nachfolgend kommen die hauptsächlich informativen Aussagen zu Profiles in UML 1.3.

4.1.1 Das Package *Extension Mechanisms*

Das Package *Extension Mechanisms* Abbildung 4-1 ist ein Unter-Package, das beschreibt, wie Modellelemente durch neue Semantik angepasst und erweitert werden können. Es und beschreibt die Semantik der drei in UML eingebauten Erweiterungsmechanismen Stereotype, Constraint und Tagged Value. Dazu verwendet es Elemente aus dem Package *Core*, aus dem in Abbildung 4-2 ein Ausschnitt abgebildet ist.
Diese drei Erweiterungsmechanismen erlauben neue Arten von Modellelementen, wie auch Hinzufügen von zusätzlicher Information in freier Form zu Modellelementen. Sie wirken sich auf Struktur und Semantik von Modellen aus.
Constraints, Stereotypes und Tagged Values werden auf Modellelemente angewandt, nicht auf deren Instanzen und repräsentieren Erweiterungen der Modellierungssprache selbst, nicht der Laufzeitumgebung. Sie können separat oder zusammen verwendet werden.
Zweck der Erweiterungsmechanismen:

- Hinzufügen neuer Modellelemente zur Verwendung bei der Modellerstellung.
- Zur Definition von Standardelementen, die nicht komplex genug sind, um direkt als eigene neue Metamodellklasse definiert zu werden.
- Zur Definition von prozess- oder implementierungssprachenabhängigen Erweiterungen.
- Zum Hinzufügen eigener semantischer und nicht-semantischer Information zu Modellebene.

Constraint und Tagged Values sind Strings, so dass sie auch von Werkzeugen, die ihre Semantik nicht unterstützen, editiert, gespeichert und übertragen werden können. Vorgesehen ist, dass für ihre Interpretation einige wenige Module zuständig sind, die deren Information nutzen, z.B. ein Code-Generator, der mit Hilfe von Tagged Values die Generierung von Code anpasst.

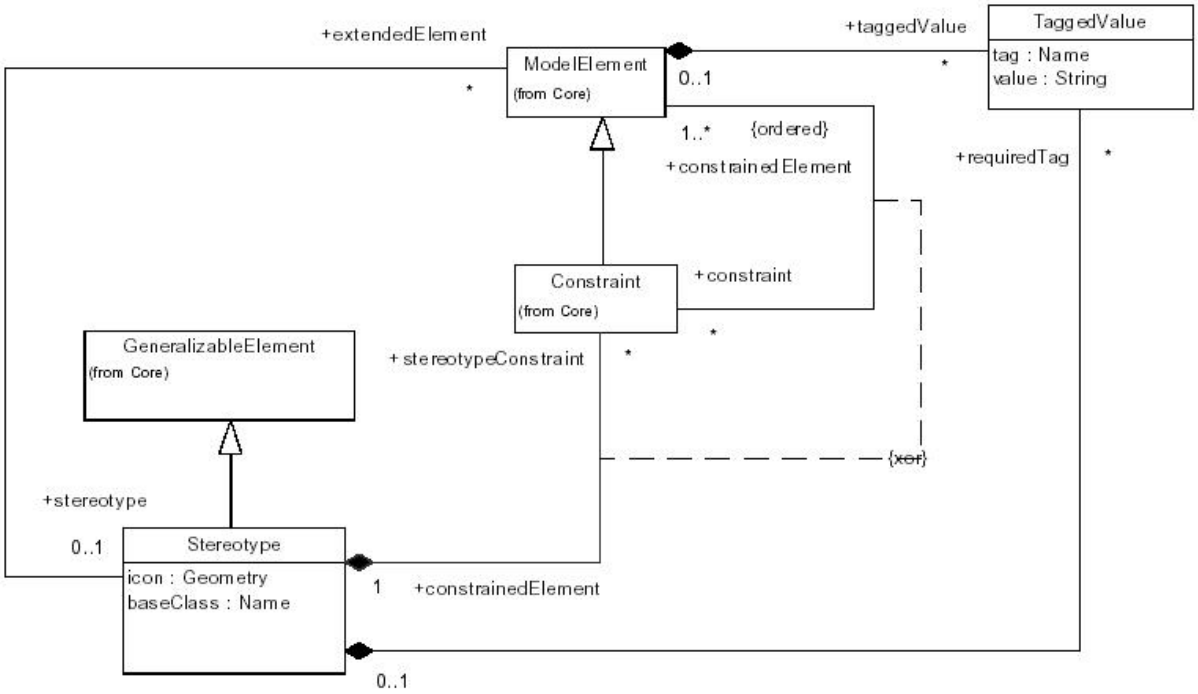

Abbildung 4-1: Das Package *Extension Mechanisms* in UML 1.3 (aus [OMG00a]).

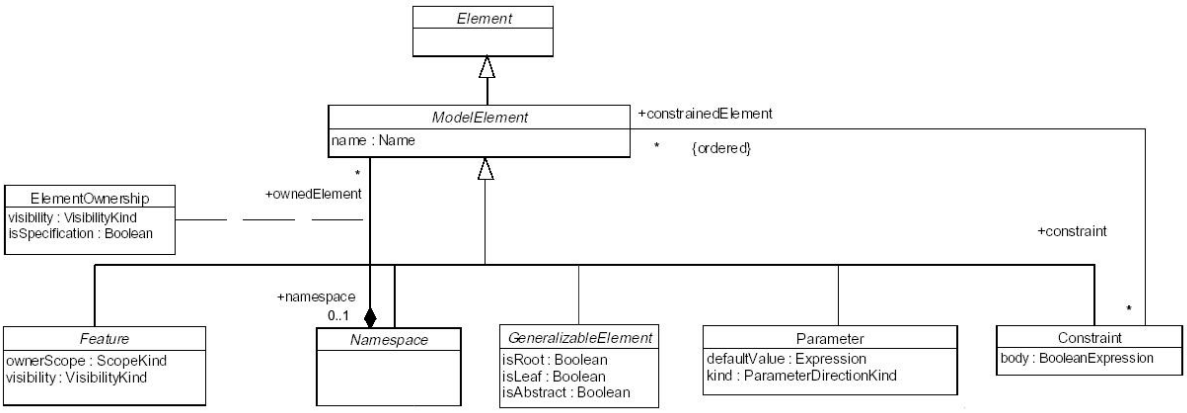

Abbildung 4-2: Ausschnitt aus dem Package *Core* (*Backbone*) in UML 1.3 (aus [OMG00a]).

4.1.2 Stereotype

Die Metaklasse *Stereotype* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-1) enthalten.
Ein Stereotype erlaubt das Markieren bzw. Klassifizieren von Elementen auf Modellebene, so dass diese sich wie Instanzen neuer, virtueller Metaklassen mit neuen Metaattributen und zusätzlicher Semantik verhalten, deren Form auf der existierenden „Base Class“ basiert.
Die Base Class ist eine Metaklasse aus dem UML Metamodell, die durch den Stereotype erweitert wird. Die Attribute, Assoziationen und Operationen werden von der Base Class übernommen, jedoch können sie zusätzliche well-formedness Constraints, zusätzliche Werte und unterschiedliche Bedeutung haben. Dazu kann ein Stereotype zusätzliche Tagged Values und zusätzliche Constraints einführen, die auf seine Instanzen angewandt werden. Daneben kann ein Stereotype auch verwendet werden, um einen Unterschied in der Bedeutung oder im

Gebrauch zweier Modellelemente gleicher Struktur anzuzeigen. Zur graphischen Unterscheidung von anderen Modellelementen kann er optional eine graphische Repräsentation einführen.

Attribute:

- *baseClass*:
  Name einer Metamodellklasse, auf die das Stereotype angewandt wird. Die Angabe einer Base Class ist obligatorisch.
- *icon*:
  Geometrische Beschreibung für ein Icon, das ein mit dem Stereotype versehenes Modellelement repräsentiert. Die Angabe eines Icons ist optional.

Assoziationen:

- *extendedElement*:
  Der *baseClass* entsprechende Modellelemente, die mit dem Stereotype versehen sind.
- *constraint*:
  Assoziation geerbt von *ModelElement*. Constraints, die auf Stereotype selbst angewandt werden.
- *requiredTag*:
  Menge von Tagged Values, deren Tags ein mit dem Stereotype versehenes Modellelement haben muss. Sind Werte spezifiziert, so sind dies Standardwerte, die dem Modellelement zugeordnet werden, solange sie von diesem nicht überschrieben werden. Ist kein Standardwert spezifiziert, so muss das Modellelement den Tagged Value explizit einen Wert zuordnen.
- *stereotypeConstraint*:
  Constraints, die auf Modellelemente, die mit dem Stereotype versehen sind, angewandt werden.

Der Name des Stereotypes darf nicht mit den Namen von Metaklassen oder Standardelementen in Konflikt geraten.

Die Metaklasse Stereotype spezialisiert *GeneralizableElement*, und damit auch *ModelElement*, und erbt alle Attribute und Assoziationen dieser Metaklassen.. Als *GeneralizableElement* sind Stereotypes spezialisierbar. Ein Stereotype kann ein oder mehrere Stereotypes spezialisieren, wobei die Eltern-Stereotypes als Base Class die gleiche Metaklasse haben müssen oder eine Unterklasse deren. Das Kind-Stereotype erbt alle Constraints und Tagged Values seiner Eltern und kann eigene Constraints und Standard Tagged Values sowie eine graphische Repräsentation hinzufügen.
Ein Modellelement darf mit maximal einem Stereotype versehen sein. Da jedoch ein Stereotype auch mehrere andere Stereotypes spezialisieren kann, können dennoch die Eigenschaften mehrerer Stereotypes einem Modellelement zugewiesen werden, da gegebenenfalls ein neues Stereotype definiert werden kann, das von mehreren Stereotypes erbt.
Da die Instanz eines Stereotypes die Struktur der Base Class übernimmt, bleibt die Behandlung des Modellelements (z.B. in Werkzeugen) für viele Fälle gleich (z.B. editieren, speichern), während bezüglich semantischer Operationen (wie z.B. der Generierung von Code) unterschieden wird zwischen einer Instanz der Base Class und einer Instanz des Stereotypes. Bezüglich der Implementierung wird vorgeschlagen, Klassen, die mit einem Stereotype versehen sind, als normale Klassen mit dem Stereotype als Eigenschaft („Property“) zu speichern.

4.1.3 Constraint

Die Metaklasse *Constraint* ist im UML Metamodell im Package *Core* (Abbildung 4-2) enthalten.
Eine Constraint spezifiziert Einschränkungen oder Bedingungen bezüglich der Instanziierung eines Modellelements und muss von jeder Instanz erfüllt werden, damit das Modell als wohlgeformt („well-formed“) gelten darf. Beschrieben wird sie in einer geeigneter Sprache, z.B. OCL, C++ oder natürlicher Sprache, die deren Interpretation spezifiziert. Es ist keine bestimmte Sprache zur Formulierung vorgegeben.
Constraints, die Stereotypes zugeordnet sind, erweitern die Modellelemente, die mit dem Stereotype versehen werden, und müssen von diesen berücksichtigt werden. Es besteht keine Notwendigkeit, sie in Verbindung mit Stereotypes zu verwenden, da sie auch direkt einem Modellelement zugewiesen werden können.
Die Auswertung von Constraints in einem System (z.B. einem CASE-Werkzeug) erfolgt immer dann, wenn das System sich im stabilen Zustand befindet, d.h. während des Wartens auf externe Eingaben nach Abschluss interner Operationen.
Im Metamodell beschreibt eine direkt einem Modellelement zugeordnete Constraint semantische Einschränkungen, die vom Modellelement beachtet werden müssen.

Attribute: (2-71)

- *body*:
  Boolescher Ausdruck in Form eines Strings. Bei Auswertung für Instanzen des *constrainedElement* während eines stabilen Zustandes des Modells muss der Ausdruck immer erfüllt sein.
- *constrainedElement*:
  Geordnete Liste von Elementen, für deren Instanzen die Constraint gilt. Ist das Element ein Stereotype, so gilt sie für die Modellelemente, die mit diesem versehen sind.

Die Metaklasse *Constraint* spezialisiert *ModelElement* und erbt damit alle deren Attribute und Assoziationen.

4.1.4 Tagged Value

Die Metaklasse *TaggedValue* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-1) enthalten.
Tagged Values sind Paare aus einem Schlüsseln („Tag“) und einem Wert, beide vom Typ String. Durch den Tag kann der mit ihm assoziierte Wert ausgewählt werden.
Häufig repräsentiert ein Tag entweder Verwaltungsinformation, z.B. Autor oder Erstellungsdatum, oder Generierungsinformation, z.B. zur Beeinflussung einer automatischen Generierung von Code oder Dokumentation.
Tags können Stereotypes zugeordnet sein und dadurch betrachtet werden als Pseudoattribute der virtuellen Metaklasse zusätzlich zu den Metaattributen der Base Class. Sie können Standardwerte definiert haben und ihre Werte können durch Constraints eingeschränkt werden. Tagged Values, die Stereotypes zugeordnet sind, erweitern die Modellelemente, die mit dem Stereotype versehen werden und müssen von diesen berücksichtigt werden. Falls der Wert eines Tagged Values, das einem Stereotype zugeordnet ist, nicht spezifiziert ist, also kein Standardwert vorgegeben ist, so muss dieser Wert in allen Modellelementen, die mit dem Stereotype versehen sind, spezifiziert werden. Es besteht jedoch kein Zwang, Tagged Values

in Verbindung mit Stereotypes zu verwenden, da sie auch direkt einem Modellelement zugewiesen werden können.
Innerhalb eines Modellelementes darf ein Tag nur einmal vorhanden sein, d.h. pro Modellelement darf es für einen gegebenen Tag-Namen nur einen Tagged Value geben.
Neben benutzerdefinierten Tagged Values gibt es auch vordefinierte in UML. Diese gehören zu den sogenannten Standardelementen, eine Bezeichnung für alle in der UML Spezifikation vorkommenden Stereotypes und Tagged Values, die in [OMG00a] in Anhang A aufgelistet sind.

Attribute:

- *tag*:
  Name, der eine erweiterte Eigenschaft beschreibt, die Modellelementen zugewiesen werden kann und als Pseudo-Attribut angesehen werden kann. Es gibt nur einen einzigen, flachen Namensraum für Tag-Namen.
- *value*:
  Wert, ausgedrückt als String. Der zulässige Wertebereich hängt von der Interpretation des zugehörigen Tags im Werkzeug ab. Die Interpretation ist außerhalb der Zuständigkeit von UML.

Assoziationen:

- *modelElement*:
  Modellelement, zu dem der Tag gehört.
- *stereotype*:
  Ein Stereotype. Der Tag gehört zu Modellelementen, die mit dem Stereotype versehen sind.

Implementierung: Vorgeschlagen wird die Implementierung als Lookup-Tabelle (qualifizierte Assoziation) von Werten (als Strings) mit Tags (als Strings) zur Auswahl der Werte. Für Tagged Values und für Attribute soll derselbe String-basierte Auswahlmechanismus verwendet werden, damit Tagged Values wie (Pseudo-) Metamodellattribute behandelt werden können.
Die Interpretation von Tagged Values liegt außerhalb von UML.

4.1.5 Modellelement

Abschließend sollen Modellelemente auf die Anwendung der Erweiterungsmechanismen hin betrachtet werden. Andere Eigenschaften sind hier nicht aufgeführt.
Die Metaklasse *ModelElement* ist im UML Metamodell im Package *Core* (Abbildung 4-2) enthalten. *ModelElement* ist die Oberklasse aller Modellelemente, wie z.B. *Class*, *Attribute*, *Association*, usw., sowie auch von *Stereotype* und *Constraint*.
Einem Modellelement können Constraints und Tagged Values zugeordnet sein (sofern diese sinnvoll für das Modellelement sinnvoll sind). Es kann mit höchstens einem Stereotype versehen werden, wobei dessen Base Class der Metaklasse des Modellelements entsprechen muss. Ein zugeordnetes Stereotype kann Constraints, die vom Modellelement erfüllt werden müssen, und Tagged Values, die im Modellelement vorhanden sein müssen, mit sich bringen. Erhält ein Modellelement Tagged Values mit unspezifizierten Werten, d.h. es ist kein Standardwert vorhanden, so muss es explizit einen Tagged Value mit dem gleichen Tag spezifizieren. Die Werte von Tagged Values, zu denen ein Standardwerte vorgegeben sind, können, müssen aber nicht, verändert werden.

Sind mehrere Constraints oder Tagged Values vorhanden, so darf es zwischen diesen keine Konflikte geben, d.h. es dürfen nicht mehrere Tags gleichen Namens oder Constraints mit widersprüchlichen Bedingungen vorhanden sein.

Attribute:

- *name:*
  Name zur Identifikation des Modellelementes.

Assoziationen:

- *constraint*:
  Constraint, die von allen Instanzen des Modellelementes in jedem stabilen Zustand (d.h. nicht während einer atomaren Operation) erfüllt werden muss.
- *stereotype*:
  Maximal ein Stereotype, das die Metaklasse des Modellelementes („Base Class“) weitergehend beschreibt. Es verändert nicht die Struktur der Base Class, aber kann zusätzliche Constraints und Tagged Values einführen, die sich auf das Modellelement auswirken. Das Stereotype fungiert als Pseudo-Metaklasse.
- *taggedValue*:
  Eine Property des Modellelementes in Form eines Tagged Value. Ist das Modellelement mit einem Stereotype versehen, so kann dieses spezifizieren, dass bestimmte Tags vorhanden sein müssen und Standardwerte zu ihnen vorgeben.

4.1.6 Profiles

UML Profiles werden nur informativ beschrieben anhand zweier (unvollständiger) Beispiele, dem „UML Profile for Software Development Processes“ und dem „UML Profile for Business Modeling“ (beide in Kapitel 4 der UML Spezifikation 1.3, [OMG00a]).
An dieser Stelle werden Profiles auch (sehr kurz) beschrieben:
Ein UML Profile ist eine vordefinierte Menge von Stereotypes, Constraints, Tagged Values und Icons für Notationen, durch die UML für einen bestimmten Bereich oder Prozess zu spezialisieren und anzupassen. Ein Profile erweitert UML nicht durch Hinzufügen neuer Grundkonzepte. Stattdessen bietet es Konventionen, um Standard-UML für einen bestimmten Teilbereich oder eine Umgebung zu spezialisieren und anzupassen.

4.2 *UML Spezifikation 1.4*

In Version 1.4 der UML Spezifikation ([OMG01a]) werden UML Profiles wesentlich besser integriert als in Version 1.3. Sie enthält sowohl eine Einbindung von UML Profiles in das Metamodell, als auch einige Weiterentwicklungen des Stereotype und des Tagged Value, die der Unterstützung von Profiles entgegenkommen.
Hier sollen nur die Änderungen zu UML 1.3 aufgeführt werden. Zum einen sind dies Änderungen im Package *Extension* Mechanismus (Abbildung 4-3). Im Falle der Constraint und des Stereotype sind diese nur geringfügig, so dass hierbei auch auf das vorangegangene Kapitel verwiesen wird, um unnötige Wiederholungen zu verweiden. Der Mechanismus des Tagged Value erfährt grundlegende Änderungen unter einer Einführung einer neuen Metaklasse *TagDefinition*. Zum Anderen ist bezüglich der Einbindung von Profiles das Package *Model Management* (Abbildung 4-4) zu betrachten.

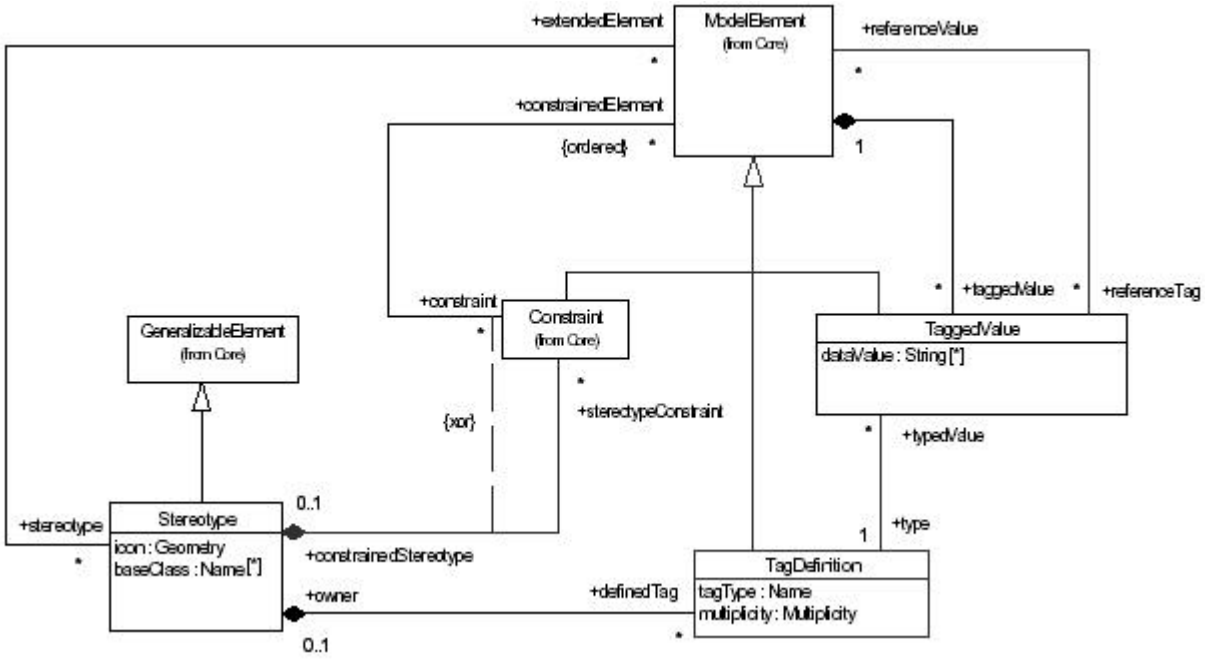

Abbildung 4-3: Das Package *Extension Mechanisms* aus dem UML Metamodell in UML 1.4 (aus [OMG01a]).

4.2.1 Stereotype

Die Metaklasse *Stereotype* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-3) enthalten. Stereotypes verhalten sich analog UML Spezifikation 1.3 (siehe 4.1.2), wobei UML 1.4 einige Weiterentwicklungen enthält.
Zum einen erfolgen Erweiterungen bezüglich Multiplizitäten in der Metaklasse *Stereotype*:
Das Attribut *baseClass* der Metaklasse *Stereotype* kann nunmehr nicht nur einen, sondern auch mehrere Namen von Metaklassen enthalten, die durch das Stereotype erweitert werden können. Weiterhin kann ein Modellelement in UML 1.4 nicht nur mit einem, sondern auch mit mehreren Stereotypes versehen werden.
Zum anderen werden Stereotypes an Änderungen in UML 1.4 angepasst:
In UML 1.4 soll die Definition von Stereotypes nur, entweder direkt oder indirekt, innerhalb eines Profiles erfolgen. Weiterhin ist die Änderung von Tagged Values in UML 1.4 zu berücksichtigen. Daher wird die Assoziation *reqiredTag* aus UML 1.3 ersetzt durch *definedTag*:

- *definedTag*:
  Menge von Tag Definitions, deren spezifizierte Tagged Values ein Modellelement haben kann, das mit dem Stereotype versehen ist.

Bezüglich der Generalisierung von Stereotypes ist in UML 1.4 verfeinert festgelegt:
Hat ein Stereotype mehrere Oberklassen, so müssen diese alle von einer gemeinsamen Oberklasse abstammen. Das Stereotype erbt dann die spezialisierteste *baseClass* oder Unterklassen davon.

4.2.2 Constraint

Die Metaklasse *Constraint* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-3) enthalten. Constraints verhalten sich analog UML Spezifikation 1.3 (siehe 4.1.3). Jedoch wird deutlicher unterschieden zwischen der Zuweisung einer Constraint zu einem Modellelement und der Zuweisung zu einem Stereotype. Im zweiten Fall ist der Wirkungsbereich der Constraint das UML Metamodell statt dem Modell, in dem die Constraint definiert wurde, d.h. die Bedingung bezieht sich auf Elemente aus dem Metamodell, wie Metaattribute oder Metaassoziationen. Dadurch können für Stereotypes „Well-Formedness Rules“, Regeln, deren Einhaltung für ein gültiges Modell Vorraussetzung ist, in gleicher Art, wie für andere Modellelemente definiert werden. Es ist dazu nicht nötig, das Metamodell explizit zu importieren. Wegen dieser Unterscheidung hat die Metaklasse *Constraint* in UML 1.4 zwei Assoziationen:

- *constrainedElement*:
  Geordnete Liste von Elementen, die Gegenstand der Constraint sind.
- c*onstrainedStereotype*:
  Stereotype, dem die Constraint zugewiesen ist.

  Eine Constraint hat genau eine Assoziation; die beiden Assoziationen schließen sich gegenseitig aus („XOR“), d.h. die Constraint kann nicht gleichzeitig sowohl einem Modellelement als auch einem Stereotype zugewiesen werden.

4.2.3 Tag Definition

Die Metaklasse *Tag Definition* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-3) enthalten. Eine Tag Definition spezifiziert die Tagged Values, die einer bestimmten Art von Modellelementen zugewiesen werden können. Sie können dazu verwendet werden, um virtuelle Metaattribute eines Stereotypes zu definieren. Ein Metaattribut kann ein Datentyp, d.h. eine Instanz der Metaklasse *DataType*, oder eine Referenz auf eine andere Metaklasse sein.
Während in UML 1.3 Tagged Values typenlos, d.h. immer vom Typ String waren, ist in UML 1.4 stets der Typ des Tagged Values in der Tag Definition anzugeben. Der Typ String bleibt nur zur Sicherung der Abwärtskompatibilität weiterhin möglich.
Weiterhin sind in UML 1.4. Tagged Values nur in Verbindung mit Stereotypes zu verwenden. Das direkte zuweisen zu einem Modellelement dient nur noch der Abwärtskompatibilität und ist zu vermeiden. Tag Definitions müssen, entweder direkt oder indirekt, in einem Profile enthalten sein. Dies ergibt sich auch daraus, da dies auch für Stereotypes gilt (siehe 4.2.1).

Attribute:

- *tagType*:
  Typ des Tagged Values. Entweder Datentyp, der den Wertebereich zugehöriger Tagged Values festlegt, oder Metaklasse, deren Instanzen durch zugehörige Tagged Values referenziert werden.
- *multiplicity*:
  Anzahl der, entsprechend *tagType*, enthaltenen Datenwerte bzw. referenzierten Modellelemente eines zugehörigen Tagged Values.

Assoziationen:

- *typedValue*:
  Zugehörige Tagged Values.
- *owner*:
  Stereotype, zu dem die Tag Definition gehört.

Die Metaklasse *TagDefinition* erweitert *ModelElement* und erbt damit alle deren Attribute und Assoziationen.

4.2.4 Tagged Value

Die Metaklasse *TaggedValue* ist im UML Metamodell im Package *Extension Mechanisms* (Abbildung 4-3) enthalten. Ein Tagged Value erlaubt die Zuweisung von Information zu einem Modellelement gemäß seiner zugehörigen Tag Definition. Er kann entsprechend der Tag Definition ein oder mehrere Datenwerte oder aber ein oder mehrere Referenzen auf Modellelemente enthalten, nicht aber beides. Die Interpretation seines Wertes ist nicht Teil der UML Semantik, sondern werkzeugspezifisch.

Attribute:

- *DataValue*:
  Menge von Werten, deren Typ dem *tagType* der zugehörigen Tag Definition und deren Anzahl der *multiplicity* der zugehörigen Tag Definition entsprechen muss.

Assoziationen:

- *Type*:
  Zugehörige Tag Definition.
- *ReferenceValue*:
  Modellelemente, die Instanzen der Metaklasse oder des Stereotypes sind, das in *tagType* der zugehörigen Tag Definition spezifiziert wird. Die Anzahl der Modellelemente entspricht *multiplicity* der zugehörigen Tag Definition.

Die Metaklasse *TaggedValue* erweitert *ModelElement*. Das geerbte Attribut *name*, der Name des Tagged Values, kann nicht frei vergeben werden, sondern muss dem Namen der zugehörigen Tag Definition entsprechen.

4.2.5 Modellelement

Ein Modellelement kann in UML 1.4 auch mit mehreren Stereotypes versehen werden.
Ansonsten verhält es sich bezüglich Erweiterungsmechanismen analog UML 1.3 (siehe 4.1.5).

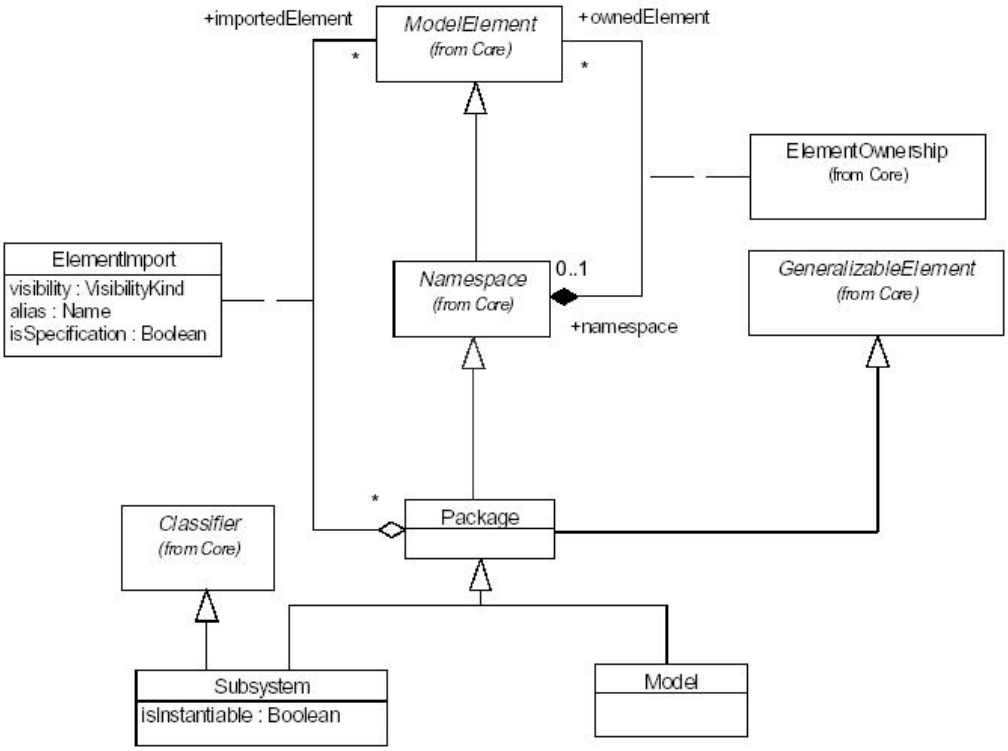

Abbildung 4-4: Das Package *ModelManagement* aus dem UML Metamodell (aus [OMG01a]).

4.2.6 Package

Die Metaklasse *Package* ist im UML Metamodell im Package *Model Management* (Abbildung 4-4) enthalten. Ein Package dient zur Organisation und Einteilung von Modellelementen in Gruppen. Es hat einen eigenen Namensraum, sonst jedoch keine weitere Semantik. Bezüglich Stereotypes ist es insofern relevant, da ein Profile ein mit dem Stereotype <<*profile*>> versehenes Package ist und eine Modellbibliothek ein Package mit dem Stereotype <<*modelLibrary*>>.
Ein Package „besitzt“ Modellelemente, d.h. dass bei Entfernen eines Packages aus einem Modell auch alle Modellelemente, die im Package enthalten sind, aus dem Modell entfernt werden.
Für enthaltene Modellelemente ist deren Sichtbarkeit (*public*, *protected* oder *private*) festlegbar. Aus dem Package heraus sind nur Elemente aus Packages in höheren Ebenen sichtbar, bis hin zum obersten Package, das durch das Stereotype <<*topLevel*>> kenntlich gemacht werden kann.
Durch eine mit dem Stereotype <<*import*>> versehene Dependency zu einem anderen Package werden dessen Elemente importiert und damit zugreifbar. Dabei müssen jedoch die Sichtbarkeiten der Elemente berücksichtigt werden. Importierte Elemente erweitern den Namensraum des importierenden Packages, wobei gegebenenfalls Aliasnamen zu verwenden sind. Die lokale Sichtbarkeit importierter Elemente ist festlegbar, der Standardwert hierbei ist *private*. Alle als *public* festgelegten importierten Elemente können von einem anderen Package weiterimportiert werden.
Analog zur <<*import*>>-Dependency kann eine <<*access*>>-Dependency verwendet werden. Der Unterschied zu <<*import*>> besteht darin, dass zugegriffene Elemente nicht in den Namensraum des Packages mit aufgenommen werden und diese auch nicht in ein anderes Package weiterimportiert werden können.

Die Metaklasse *Package* spezialisiert *Namespace* und *GeneralizabeElement*. Durch *Namespace* erhält sie den eigenen Namensraum. Durch *GeneralizableElement* ist ein Package generalisierbar. Bei der Generalisierung von Packages werden alle eigenen und importierten Elemente, die als *public* oder *protected* ausgezeichnet sind, zusammen mit deren Sichtbarkeiten vererbt, sowie alle Relationen des Packages zu anderen Modellelementen.
Spezialisiert wird *Package* durch *Subsystem* und durch *Model*.

4.2.7 Modellbibliothek

Eine Modellbibliothek ist ein Package, das mit dem Stereotype <<*modelLibrary*>> versehen ist. Es enthält Modellelemente, die zur Wiederverwendung in anderen Packages bestimmt sind. Sie kann als analog zu einer Klassenbibliothek in manchen Programmiersprachen betrachtet werden. Von einem Profile unterscheidet sich eine Modellbibliothek insofern, dass sie nicht unter Verwendung von Stereotypes und Tagged Values das Metamodell erweitert.

4.2.8 Profile

Ein Profile ist ein mit dem Stereotype <<*profile*>> versehenes Package, das Modellelemente enthält, die durch die Verwendung von Erweiterungsmechanismen für einen bestimmten Zweck oder für ein bestimmtes Gebiet angepasst sind. Es kann Lightweight Extensions, d.h. Stereotypes, Tagged Values und Constraints, enthalten und zusätzlich Datentypen, d.h. Instanzen der Metaklasse *DataType*. Es kann spezifizieren, welche Modellbibliotheken es verwendet und die Untermenge des Metamodells, die es erweitert.
Der Mechanismus des Profiles wird nicht durch die Einführung neuer Metaklassen umgesetzt, sondern mit Hilfe von Stereotypes und Tagged Values aus dem Package *ModelManagement*. Dazu gehören neben dem Stereotype <<*profile*>> für Packages und dem bereits erwähnten Stereotype <<*modelLibrary*>> für Packages (siehe 4.2.7) auch die Stereotypes <<*modelLibrary*>> für Dependencies und <<*appliedProfile*>> für Dependencies, sowie der Tagged Value *{applicableSubset}* für <<*profile*>>-Packages.
Eine Dependency, die mit dem Stereotype <<*modelLibrary*>> versehen ist, beschreibt die Abhängigkeit eines Profiles von einer Modellbibliothek. Sie besteht zwischen einem Profile und einem Package, das kein Profile ist und Modellelemente enthält, die für eine mehrfache Nutzung bestimmt sind.
Eine Dependency, die mit dem Stereotype <<*appliedProfile*>> versehen ist, besteht zwischen einem Profile und einem Package jeglicher Art, d.h. auch Unterklassen von *Package*, typischerweise *Model*. Sie beschreibt, dass das Profile auf das Package angewandt werden kann, d.h. sowohl auf das Package selbst, wie auch auf alle enthaltenen Modellelemente. Dem Package werden alle Stereotypes und Tag Definitions des Profiles zugänglich gemacht. Ein Package kann an beliebig vielen <<*appliedProfile*>>-Dependencies teilnehmen, wodurch ein Modell auf einer beliebigen Anzahl von Profiles basieren kann.
Die Tag Definition *{applicableSubset}* gehört zu dem Stereotype <<*profile*>>, weshalb ein zugehöriger Tagged Value nur in Profiles enthalten sein kann. Er beschreibt die anwendbare Untermenge, d.h. er listet die vom Profile verwendeten und erweiterten Metaelemente auf. Sein Wert ist eine Menge von Strings, die jeweils den Namen eines verwendeten Metaelementes repräsentieren. Er schließt nicht notwendigerweise die Verwendung anderer Metaelemente aus, muss aber alle Metaklassen enthalten, die vom Profile referenziert werden. Somit enthält er zumindest von jedem enthaltenen Stereotype dessen *baseClass*. Er bezieht

sich immer nur direkt auf das zugehörige Profile, auch dann, wenn das Profile andere Profiles importiert oder spezialisiert. Ist der Tagged Value in einem Profile nicht vorhanden, so bedeutet dies, dass das ganze Metamodell anwendbar ist.
Ein Profile kann Relationen zu anderen Packages, also auch Profiles, haben. Diese haben die für Packages übliche Semantik, z.B. Generalisierung, <<*import*>>-Dependency oder <<*access*>>-Dependency. Bei der Generalisierung von Profiles werden alle Stereotypes, Tag Definitions und Constraints vererbt. Zusätzliche Verfeinerungen im Kind-Profile dürfen nicht im Widerspruch zu Geerbtem stehen. Relationen zu Modellbibliotheken werden ebenso vererbt, der Tagged Value <<*applicableSubset*>> dagegen nicht.

4.2.9 Notation der Definition eines Stereotypes

Ab Version 1.4 enthält UML auch eine Notation für die Definition von Stereotypes. Alternativ stellt sie sowohl eine graphische Notation, wie auch eine Tabellennotation zur Verfügung.
Da die Definition von Stereotypes vom Anwender vorgenommen wird, indem er eine Instanz der Metaklasse *Stereotype* erzeugt, wird diese bei der Modellierung vorgenommen und geschieht damit auf Schicht M1 der OMG Metamodellarchitektur. Andererseits wird mit einem Stereotype eine neue (virtuelle) Metaklasse erzeugt, die im Modell instanziiert werden kann, weshalb die Definition konzeptionell in die Metamodellschicht, also M2 der OMG Metamodellarchitektur, gehört. Besonders bezüglich der graphischen Notation ist es daher nötig, einen Weg zu finden, der diesem Übergreifen der beiden Schichten Rechnung trägt.

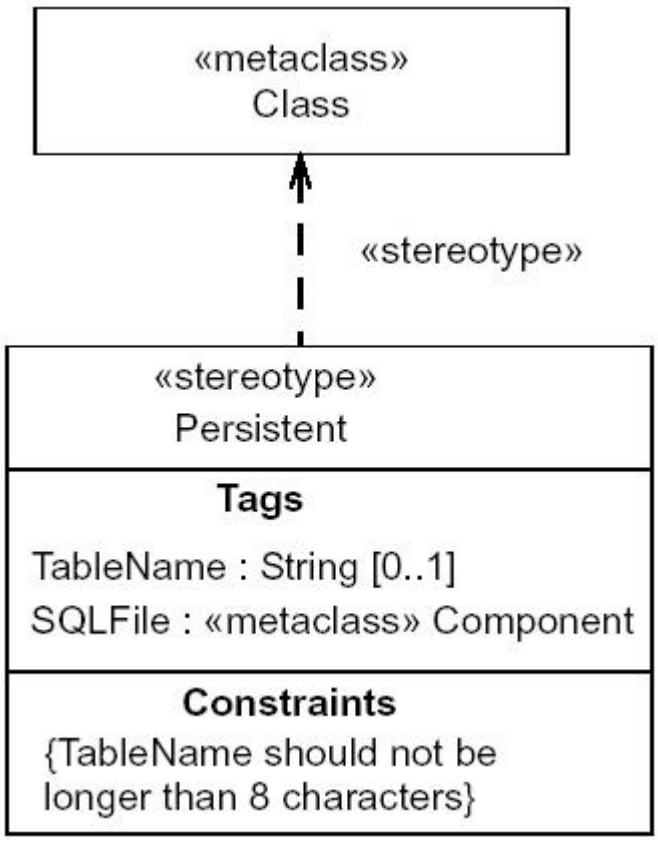

Abbildung 4-5: Graphische Notation der Definition eines Stereotypes (aus [OMG01a]).

Abbildung 4-5 zeigt ein Beispiel für die graphische Notation. Um die b*aseClass* des Stereotypes, eine Metaklasse aus der Metamodellschicht, zu kennzeichnen, wird sie mit <<*metaclass*>> markiert. Zwischen ihr und dem Stereotype, dessen *baseClass* sie ist, besteht eine Dependency, die mit <<*stereotype*>> markiert ist. Der neu definierte Stereotype selbst wird mit <<*stereotype*>> gekennzeichnet. Er enthält zugehörige Tag Definitions und Constraints. Die Tag Definitions werden durch *Name: Typ* dargestellt. Ist die Tag Definition eine Referenz auf eine Metaklassen bzw. ein Stereotype, so werden bei der Angabe des Typs, wiederum die Kennzeichnungen <<*metaclass*>> bzw. <<*stereotype*>> verwendet. Constraints werden in der in Modellen üblichen Weise, d.h. als ein in geschweiften Klammern (*{}*) gesetzter Ausdruck, notiert.
Diese Lösung einer graphischen Notation ist jedoch nicht dem UML Metamodell vereinbar. So wird Kennzeichnung der Metaklasse mit <<metaclass>> in der Notation eines Stereotypes vorgenommen, es ist aber nicht möglich, Metaklassen mit einem Stereotype zu versehen. Weiterhin wird die Beziehung zwischen Stereotype und Metaklasse in Form einer Dependency notiert, obwohl eine Instanz der Metaklasse *Dependency* nicht auf eine Metaklasse angewandt werden kann. Daher ist eine derartige graphische Stereotype-Definition nur als Konvention und pragmatische Lösung zu betrachten, die einer Grundlage im UML Metamodell entbehrt.
Bei der Tabellennotation werden die Definitionen in eine Tabelle mit entsprechend überschriebenen Spalten eingetragen, so dass keine zusätzlichen Konstrukte (wie z.B. ein Marker <<*metaclass*>>) benötigt werden. Ein Beispiel wird in Abbildung 4-6 gezeigt. Zugehörige Tag Definitions sind in einer separaten Tabelle zu notieren (Abbildung 4-7).

Stereotype	Base Class	Parent	Tags	Constraints	Description
Architectural Element	Generalizeable Element	N/A	N/A	N/A	A generic stereotype that is the parent of all other stereotypes used for architectural model-ing
Capsule	Class	Architectural Element	isDynamic	self.isActive = true	Indicates a class that is used to model the structural components of an architecture spec-ification

Abbildung 4-6: Definition eines Stereotypes in Tabellennotation (aus [OMG01a]).

Tag	Stereotype	Type	Multiplicity	Description
isDynamic	Capsule	UML::Datatypes::Boolean	1	Used to identify if the associated cap-sule class may be created and destroyed dynamically

Abbildung 4-7: Tag Definition in Tabellennotation (aus [OMG01a]).

4.3 *Anforderungen an UML Profiles vs. Umsetzung von UML Profiles in UML Spezifikation 1.4*

In Kapitel 3, „Anforderungen an UML Profiles“, wurden Anforderungen an UML Profiles formuliert. Das aktuelle Kapitel enthält die Umsetzung des Profile-Mechanismus in der UML Spezifikation, wobei in UML 1.4, im Gegensatz zur Version 1.3, bereits viele konkrete

Festlegungen bezüglich Profiles enthält. Im folgenden Abschnitt sollen die Anforderungen mit der Umsetzung in UML 1.4 verglichen werden. Die Anforderungen aus Kapitel 3 werden hierbei durch ihre Überschrift und Nummerierung aus Kapitel 3 referenziert. Zu jeder Anforderung ist die zugehörige Erklärung in Kapitel 3 zu beachten.

Erfüllte obligatorische Anforderungen:

Spezialisierung der UML Standard-Semantik (1):
Erfüllt durch die Verwendung von Lightweight Extensions. Diese sind so definiert, dass sie die Semantik des Standard-Metamodells nicht verletzten dürfen, und bieten die Möglichkeit, neue Semantik in OCL oder natürlicher Sprache hinzuzufügen.

Lightweight Extensions (2):
Erfüllt, da <<profile>>-Package nur Lightweight Extensions enthalten darf.

Anwendbare Untermenge (3):
Erfüllt durch Tagged Value *{applicableSubset}*.

Referenzen auf Modellbibliotheken (4):
Erfüllt durch Dependency <<ModelLibrary>>.

Zusammenbinden von UML Profiles und Modellbibliotheken (5):
Da Profiles und Modellbibliotheken Packages sind, können sie gemeinsam in ein Package (oder eine Unterklasse der Metaklasse *Package*) verpackt werden.

Angabe gewählter UML Profiles für Packages (6):
Erfüllt durch <<appliedProfile>>-Dependency.

Anwendung „by reference“ (8):
Modelle referenzieren Profiles durch die <<appliedProfile>>-Dependency.

Profile-Spezialisierung (9):
Definiert durch Stereotype <<profile>>.

Formalisierung der Definition von Tagged Values mit Typangabe (11):
Wurde in UML 1.4 realisiert.

Mehrwertige Tagged Values (12):
Wurde in UML 1.4 realisiert.

Nicht vollständig erfüllte obligatorische Anforderungen:

Verwendung bereits existierender Austauschmechanismen (7):
Da Profiles unter Verwendung vorhandener UML Konstrukte in das UML Metamodell eingebunden wurden, können existierende Austauschmechanismen (z.B. XMI) ohne weitere Anpassung verwendet werden. Einzig offen ist der Umgang mit Icons zur graphischen Repräsentation von Stereotypes. Hier ist noch kein Format zum Austausch des Icons festgelegt. Theoretisch sollte die Festlegung eines solchen Formats aber einfach möglich sein.

Profile-Komposition (10):

Keine Festlegungen getroffen. Die Kombination von Profiles ist zwar möglich, da ein <<profile>>-Package ein anderes mit einer <<import>>-Dependency importieren kann, jedoch bleibt das Problem der Kompatibilität. Ein allgemeiner Mechanismus zur Entscheidung, ob die Semantik, die in zwei Profiles enthalten ist, kompatibel oder inkompatibel ist, ist schwer vorstellbar. Stattdessen müsste „manuell“ diese Entscheidung getroffen werden. Denkbar wäre z.B. dass für ein Profile angegeben wird, welche anderen Profiles zu ihm kompatibel bzw. inkompatibel sind. Dies könnte eventuell. durch einen Tagged Value realisiert werden.

Erfüllte Optionale Anforderungen:

Einem Modellelement mehrere Stereotypes zuweisbar (1):
Wurde in UML 1.4 realisiert.

Referenzieren von Metaklassen (3):
Wurde in UML 1.4 realisiert.

Nicht vollständig erfüllte Optionale Anforderungen:

Graphische Notation für Definition von Stereotypes (2):
Die Lösung in UML 1.4 ist nicht vereinbar mit dem UML Metamodell (siehe 4.2.9).

Generelle Spezialisierung von Metaklassen (4):
Hierzu enthält UML 1.4 keine Festlegungen.

Zusammenfassung:

Es wird deutlich, dass UML 1.4 fast alle Anforderungen an UML Profiles erfüllt. Von den obligatorischen Anforderungen werden nur zwei nicht ganz vollständig erfüllt. Für diese sind jedoch Lösungen denkbar, so dass zu Hoffen bleibt, dass sie in zukünftigen Versionen der UML Spezifikation ebenfalls umgesetzt werden.

5 Bestehende Werkzeugunterstützung am Beispiel „Objecteering“ von Softeam

Das CASE-Werkzeug mit der am weitesten gehenden Unterstützung von UML Profiles ist „Objecteering“ von Softeam [INET02]. Ein Vertreter der Firma Softeam zeichnet sich auch für die grundsätzlichen Papiere der OMG zu Profiles ([OMG99a], [OMG99b]) mitverantwortlich. Dieses Werkzeug soll nun eingehend auf die Umsetzung von UML Profiles und den Umgang mit ihnen untersucht werden.

5.1 *Software*

Objecteering wird von der französischen Firma Softeam entwickelt. Die grundlegende Anwendung ist der UML Modeler, welche die Grundfunktionalität zur UML Modellierung zur Verfügung stellt. Die augenblicklich (September 2001) aktuelle Version ist Version 5.1, welche kostenlos verfügbar ist. Darauf aufbauend gibt es eine Reihe von kostenpflichtigen Erweiterungen der Anwendung. Bezüglich UML Profiles ist die Anwendung UML Profile Builder von belang, bei der es sich um eine eigenständige Anwendung handelt, mit der UML Profiles für den UML Modeler erstellt werden können. Im UML Modeler können erstellte UML Profiles dann genutzt werden.

5.2 *Erstellung eines Profiles mit UML Profile Builder*

Die Anwendung UML Profile Builder dient zur Erstellung von UML Profiles für Objecteering. Im folgenden soll dargestellt werden, wie dies in Objecteering gehandhabt wird und welche Bestandteile in Objecteering ein UML Profile enthalten muss. Die Reihenfolge der folgenden Darstellungen orientiert sich dabei an einer möglichen Schrittfolge zur Erstellung eines UML Profiles in Objecteering. Ausgehend von der Ansicht der Anwendung, wie sie sich nach dem Programmstart präsentiert, werden zunächst die wesentlichen Fenster und Bedienelemente erläutert. Anschließend wird die Erstellung eines neuen Profiles dargestellt und die möglichen Bestandteile des Profiles erklärt. In Objecteering bestehen UML Profiles sowohl aus Elementen, die von der OMG standardmäßig als Bestandteile eines UML Profiles vorgesehen sind, als auch aus proprietären Elementen, mit deren Hilfe einem UML Profile weitere Semantik und Funktionalität zugewiesen werden kann. Die folgende Aufzählung der Bestandteile wird nach dieser Unterscheidung gegliedert. Zuletzt wird gezeigt, welche Maßnahmen in Objecteering zur Weitergabe und zur Anwendung der Profiles nötig sind.
Zur Veranschaulichung der Prinzipien der verschiedenen Elemente soll ein durchgehendes Beispiel verwendet werden. Es soll ein Profile entstehen, mit dessen Hilfe man (sehr eingeschränkt) Java-Code generieren kann. Der Code soll ausgehend von Packages oder Klassen generiert werden, wobei für jede Klasse eine eigene Datei verwendet wird. Es werden die Attribute und Methoden der Klassen miteinbezogen, wobei vereinfachend die Methodenparameter nicht berücksichtigt werden. Methoden können durch einen Tagged Value als *synchronized* spezifiziert werden. Es soll die Möglichkeit bestehen, einer Methode

ihren Code im Modell hinzufügen zu können, sowie die erzeugten Dateien aus dem Werkzeug heraus betrachten und verwalten zu können. Zusätzlich soll das Profile beispielhaft einen Stereotype enthalten, mit dem man eine Klasse als persistent kennzeichnen kann. Um das Beispiel übersichtlich zu halten, wird der Stereotype nicht in die Codegenerierung miteinbezogen.
Bei dem beschriebenen Beispiel handelt es sich nicht um ein existierendes UML Profile, da diese weniger geeignet sind um in Verständlicher und nicht zu umfangreicher Art und Weise die Möglichkeiten von Objecteering aufzeigen zu können.

5.2.1 Überblick

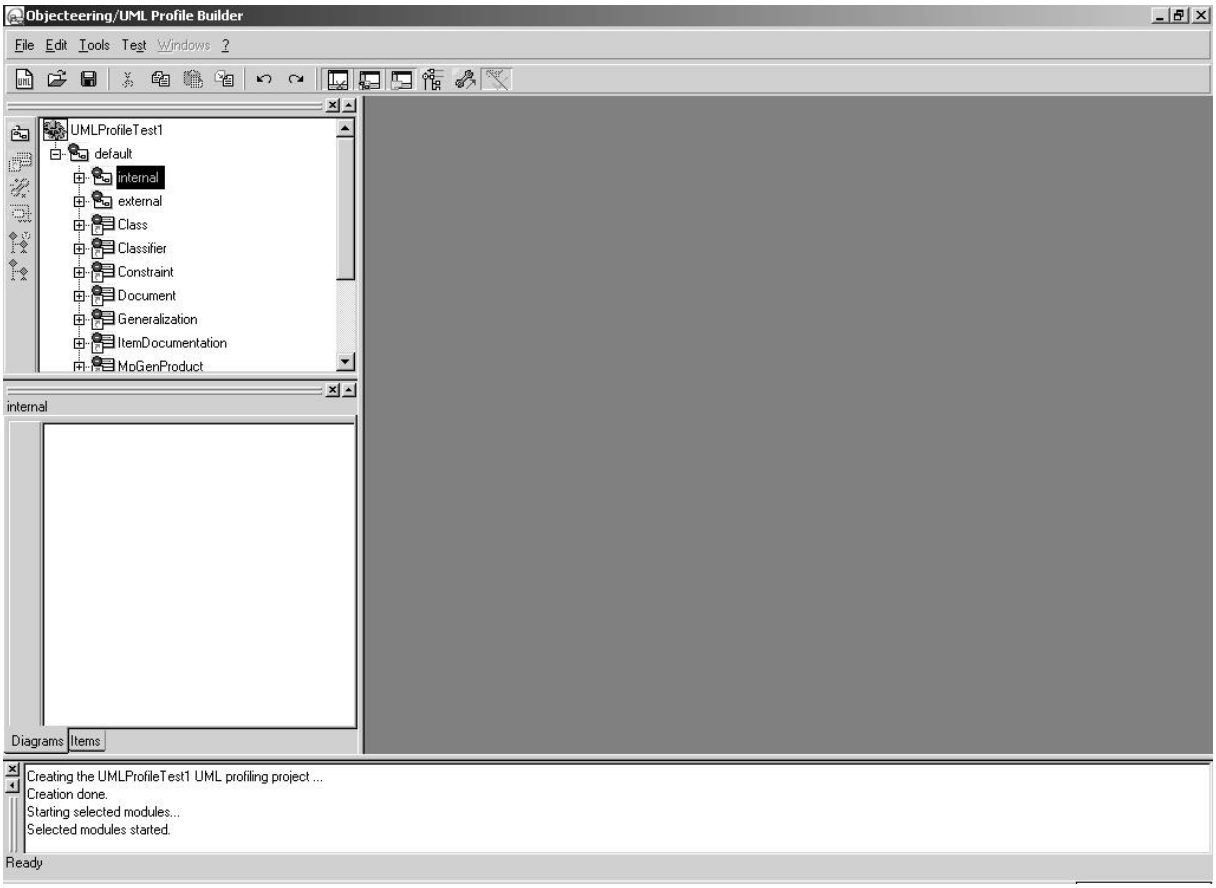

Abbildung 5-1: Der UML Profile Builder.

Abbildung 5-1 zeigt den UML Profile Builder nach dem Start und der Erstellung eines neuen Projektes. Links oben befindet sich das Explorer-Fenster, das in einer Baumstruktur alle vorhandenen Bestandteile des Projekts enthält. Darunter befindet sich das Eigenschaftsfenster, in dem Eigenschaften des aktuell markierten Objekts angezeigt werden. Das Hauptfenster rechts dient im Wesentlichen zur Erstellung eines Modells, mit dem erstellte Profiles getestet werden können. Es ist nicht notwendig, erstellte Profiles in den UML Modeler einzubinden um sie zu testen, sondern es kann direkt im UML Profile Builder ein Testprojekt erstellt werden.
Grundsätzlich enthalten im Profile Builder Dialogboxen zur Erstellung eines Elementes immer die Möglichkeit, diesem Notes (z.B. Kommentare, siehe 5.2.4.2) oder Tagged Values

zuzuordnen, die sich dann direkt auf das jeweilige Element auswirken. Diese Möglichkeit besteht grundsätzlich für alle Elemente und soll daher im Folgenden nur noch dann explizit erwähnt werden, wenn es erforderlich erscheint.

5.2.2 Erstellung eines neuen UML Profiles

In Objecteering ist in jedem neuen Projekt bereits ein hierarchischer Baum mit UML Profiles enthalten (Abbildung 5-2). Diese können weder gelöscht noch verändert werden. Die Baumstruktur repräsentiert die Vererbungshierarchien der Profiles. Ein neu zu erstellendes Profile wird in diesen Baum eingefügt und muss eines der vorhanden Profiles spezialisieren. Die Wurzel des Baumes, also das Profile, das alle anderen Profiles spezialisieren, ist das Profile *default*, welches keine weitere spezielle Semantik enthält. Soll ein neues allgemeines Profile erstellt werden, so wird man *default* als direkte Oberklasse wählen. Die darunter befindlichen Profiles enthalten bereits Semantik bezüglich Werkzeugspezifischer Eigenschaften; z.B. enthält das Profile *Code* bereits Semantik bezüglich der Codegenerierung in Objecteering. Ein Anwender, der ein Profile erstellen möchte, das die Codegenerierung beeinflusst, sollte *Code* spezialisieren, um grundlegende Eigenschaften für die Codegenerierung zu erben und nur die eigenen Änderungen oder Erweiterungen seinem Profile hinzufügen zu müssen. Das Profile i*nternal* und deren Kinder enthalten interne Regeln für Objecteering und können nicht vom Anwender spezialisiert werden. Zum Profile *H* konnten keine Informationen gefunden werden.
Bei einer Spezialisierung von Profiles werden alle seine Bestandteile, deren Sichtbarkeit nicht als private spezifiziert ist, vererbt.

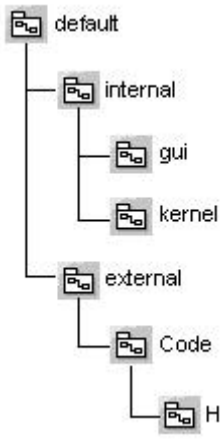

Abbildung 5-2: Baum mit bereits im Profile Builder enthaltenen Profiles.

Als Beispiel für alle weiteren Schritte wird von einem neuen Profile *TestProfile* ausgegangen. Da das Profile im Laufe des Beispieles Funktionalität zur Codegenerierung erhalten soll, erweitert es das Profile *Code*.

5.2.3 Unterstützung der Standardbestandteile eines UML Profiles

Die Standardbestandteile eines UML Profiles sind die von der OMG vorgesehenen (siehe „Obligatorische Anforderungen“, Kapitel 3.1), d.h. Stereotypes, Constraints und Tagged

Values sowie die anwendbare Untermenge und Referenzen auf Modellbibliotheken. In diesem Abschnitt sollen die Elemente in Objecteering, die sich darin einordnen lassen, untersucht werden.

5.2.3.1 Referenzen auf Metaklassen

Referenzen auf Metaklassen sind notwendig, um Metaklassen Erweiterungselemente zuzuweisen. Beispielsweise besitzt ein Stereotype das Attribut *baseClass*, das die Metaklasse spezifiziert, die von dem Stereotype erweitert wird. In Objecteering werden Metaklassenreferenzen als explizite Elemente eines Profiles behandelt und die Standarderweiterungselemente, wie Stereotypes, Constraints und Tagged Values nach ihnen gegliedert. In Abbildung 5-3 ist ersichtlich, dass in der Baumstruktur einem Profile nicht nur die Profiles, die es spezialisieren, untergeordnet sind, sondern auch Referenzen auf Metaklassen (*Attribute*, *Class*, ...), welchen dann jeweils Elemente zu deren Erweiterung zugeordnet sein können. Um einem Profile Erweiterungen bezüglich einer Metaklasse erstellen zu können, muss immer zuerst eine Referenz auf diese Metaklasse erstellt worden sein.

Eigenschaften:

- *Name*:
  Name der zu referenzierenden Metaklasse

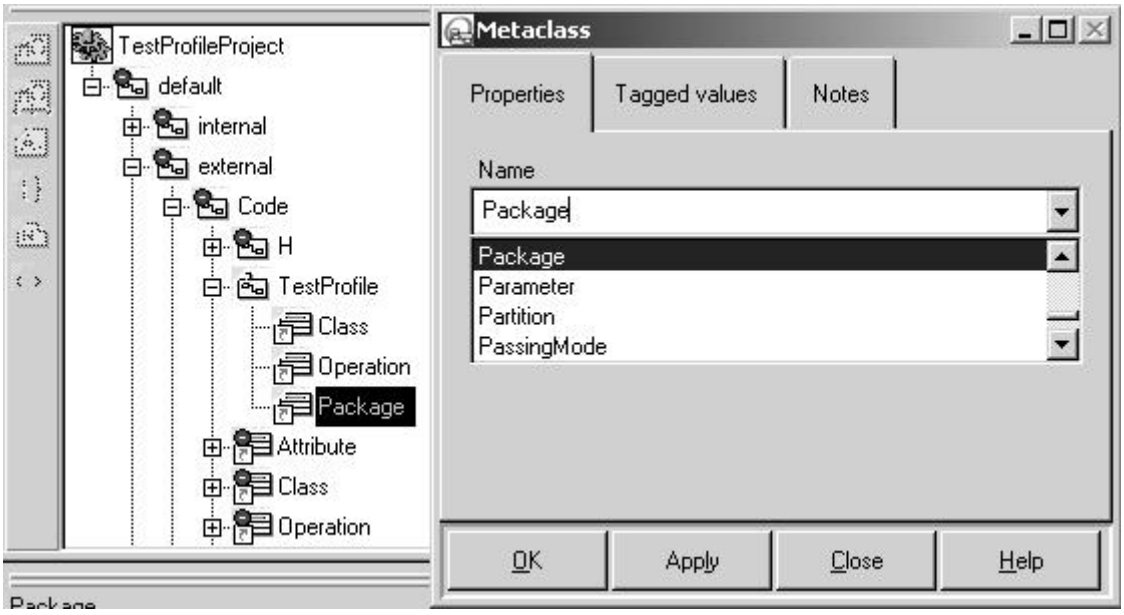

Abbildung 5-3: Dem Profile *TestProfile* wird eine Referenz auf die Metaklasse *Package* hinzugefügt.

Beispiel:
Dem Profile *TestProfile* werden Referenzen auf die Metaklassen *Class*, *Operation* und *Package* hinzugefügt. In der Abbildung sieht man das Dialogfenster zur Auswahl einer Metaklasse, hier *Package*, sowie das Explorer-Fenster.

5.2.3.2 Stereotypes

Stereotypes werden einer Referenz auf eine Metaklasse zugeordnet. Diese Metaklasse wird durch das Stereotype erweitert. Demnach entspricht die Referenz auf die Metaklasse dem Attribut *baseClass* des Stereotypes aus dem UML Metamodell. Dem Stereotype können Constraints und Tagged Values zugeordnet werden, deren Semantik sich dann auf die Metaklasse bezieht, sowie Note Types (siehe 5.2.4.2). Unabhängig davon kann, wie allen anderen Elemente im UML Profile Builder auch, das Stereotype selbst mit Tagged Values und Notes (siehe 5.2.4.2) versehen werden, welche sich dann auf das Stereotype selbst beziehen und nicht auf die erweiterte Metaklasse.

Eigenschaften:

- *Name:*
  Name des Stereotypes
- *Icon*:
  Icon im Bitmap-Format (Dateiendung *bmp*) oder GIF-Format (Dateiendung *gif*) zur graphischen Repräsentation des Stereotypes in Modellen. Das Icon ersetzt die Repräsentation der erweiterten Metaklasse, d.h. erweitert das Stereotype die Metaklasse *class*, so wird eine Instanz des Stereotypes durch *Icon* statt durch ein Rechteck (von *class*) repräsentiert. Die Angabe eines Icons ist optional.
- *Small Icon*:
  Analog *Icon*, wird jedoch der Repräsentation der erweiterten Metaklasse angefügt, d.h. erweitert das Stereotype z.B. die Metaklasse *class*, so erscheint bei einer Instanz des Stereotypes *Icon* innerhalb des Rechtecks (von *class*).
- *Explorer Icon*:
  Analog *Icon*, fungiert jedoch als Repräsentation des Stereotypes im Explorer-Fenster..

Für ein erstelltes Stereotype kann im Eigenschaftsfenster angegeben werden, dass es ein anderes Stereotype spezialisiert (Abbildung 5-4). Objecteering schlägt dann nur Stereotypes mir gleicher *baseClass* als Eltern vor. Es ist aber nicht möglich, von mehreren Stereotypes zu erben. Vererbt werden die Tag Definitions (siehe 5.2.3.3), die dem Eltern-Stereotype zugeordnet sind.

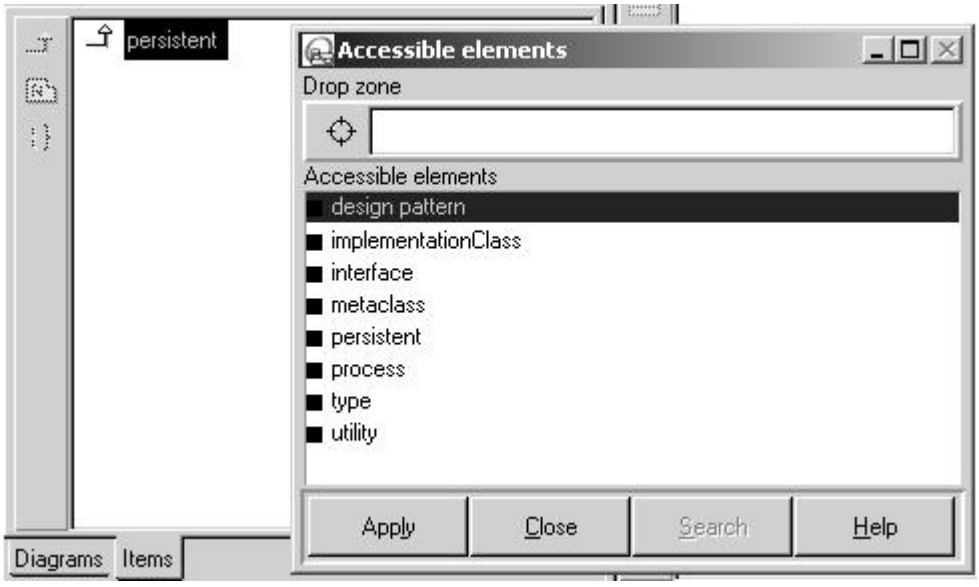

Abbildung 5-4: Ein Stereotype gibt im Eigenschaftsfenster *persistent* als Oberklasse an.

Beispiel:
Abbildung 5-5 zeigt die Erstellung eines Stereotypes *persistent*, das der Referenz auf die Metaklasse *class* zugeordnet wird. Als Icons zur Repräsentation wird das Icon  in drei verschiedenen Größen verwendet.

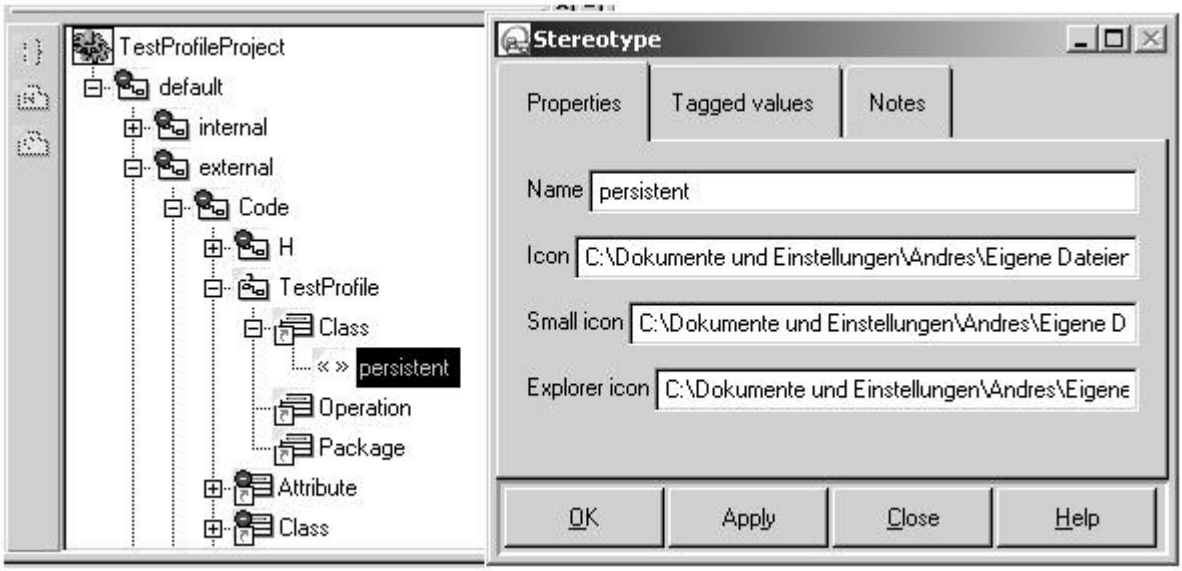

Abbildung 5-5: Erstellen eines Stereotypes *persistent* unter der Referenz auf die Metaklasse *Class*.

5.2.3.3 Tag Definitions

In Objecteering werden Tag Definitions als „Tagged Value Types“ bezeichnet. Hier soll aber weiterhin die Bezeichnung aus der UML Spezifikation, Tag Definition, verwendet werden. Sie können Stereotypes zugeordnet werden, oder auch direkt einer Referenz auf eine Metaklasse.

Eigenschaften:

- *Name*:
  Name des Tagged Values
- *Number of Parameters*:
  Anzahl der Parameter eines zugehörigen Tagged Values. Der Standardwert null steht hierbei dafür, dass der Tagged Value als Marker verwendet werden soll (wie z.B. „abstract“), dessen Vorhandensein eine bestimmte Eigenschaft markiert. (Marker könnten auch als vom Typ Boolean betrachtet werden, wobei Vorhandensein als „true“ und Nichtvorhandensein als „false“ zu werten ist, d.h. „abstract“ entspricht „isAbstract = true“).
- *Qualified*:
  Ob Tagged Value einen Qualifier besitzen soll.
- *Inclusion in the signature*:
  Ob zugehöriger Tagged Values beim Vergleich von Signaturen miteinbezogen werden soll. Dies kann in bestimmten Fällen bezüglich der Generierung von Code relevant sein, soll hier aber aufgrund der Komplexität nicht ausführlicher erläutert werden.

Die Festlegung eines Typs der Parameter ist nicht möglich. Abgesehen von der Verwendung als Marker (mit Parameteranzahl null) werden alle Werte als Strings behandelt.

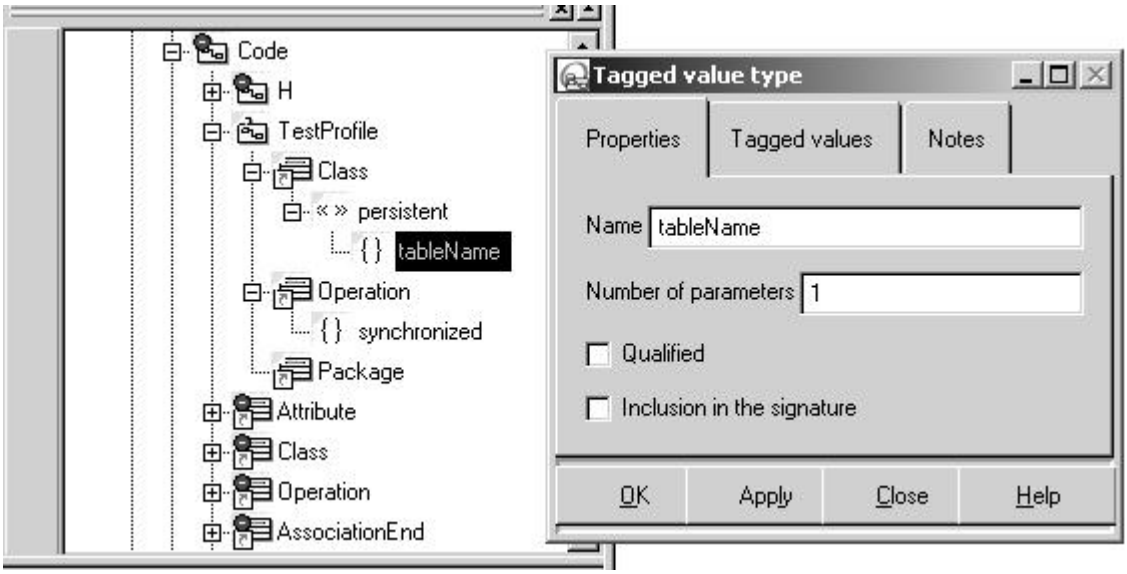

Abbildung 5-6: Eine Tag Definition *tableName* wird erstellt und dem Stereotype *persistent* zugeordnet

Beispiel:
Das Beispiel in Abbildung 5-6 zeigt eine Tag Definition *tableName*, die dem Stereotype *persistent* zugeordnet wird. Da hier der Name einer Datenbanktabelle eingegeben werden soll, ist die Parameteranzahl eins.
Zusätzlich wurde eine Tag Definition *synchronized* erstellt, die direkt der Metaklasse *Operation* zugeordnet wurde und zur Markierung von Methoden für die Codegenerierung dienen soll. Da es sich hierbei um einen „Marker“ handelt, wurde als Parameteranzahl null festgelegt.

5.2.3.4 Constraints

Constraints werden Stereotypes angefügt. Ihre Semantik bezieht sich dann auf das Modellelement, das mit dem Stereotype versehen ist.

Eigenschaften:

- *Name*:
  Name der Constraint.
- *Body*:
  Beliebiger String, der die Einschränkung oder Bedingung enthält. Er wird bei der Anwendung der Constraint im Werkzeug nicht semantisch ausgewertet.

Funktionalität kann durch Constraints nicht beeinflusst werden.

Beispiel:
Abbildung 5-7 zeigt die Erstellung einer Constraint, die dem Stereotype *persistent* hinzugefügt wird, und die Länge des Strings des Tagged Values *tableLength* auf acht Zeichen begrenzt.

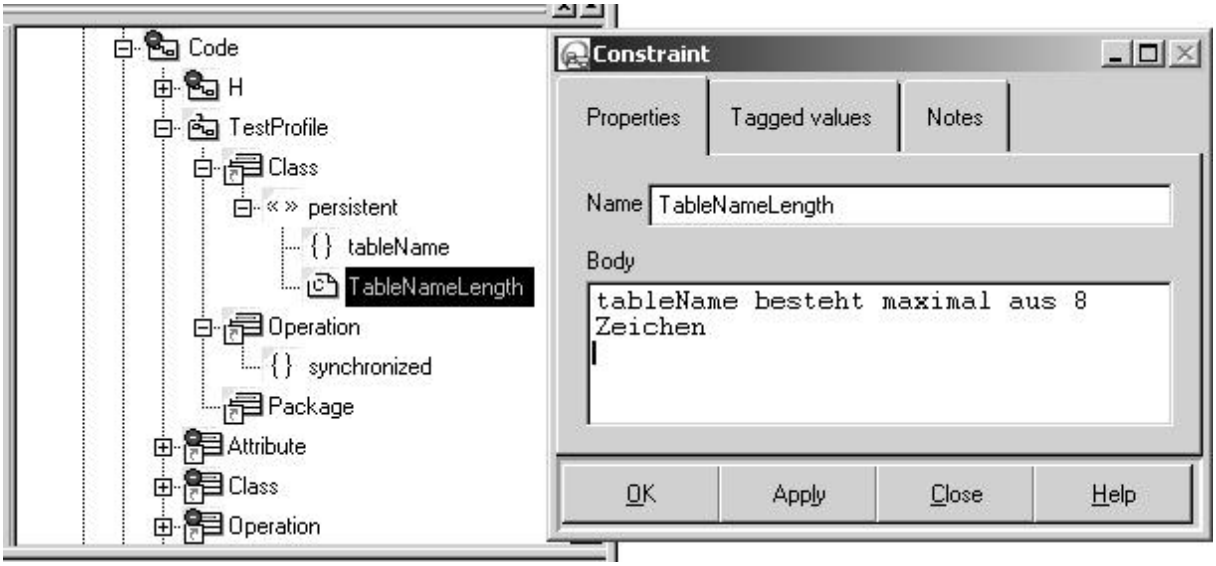

Abbildung 5-7: Eine Constraint *TableNameLength* wird erstellt und dem Stereotype *persistent* zugeordnet

5.2.4 Proprietäre Bestandteile eines UML Profiles

Zusätzlich zu den Standardbestandteilen eines UML Profiles können Profiles in Objecteering noch weitere Bestandteile enthalten. Viele Werkzeuge, so auch Objecteering, unterstützen nicht nur den reinen Modellierungsprozess, sondern auch weitergehende Schritte, die durch UML nicht standardisiert sind, wie z.B. die Generierung von Dokumentation und Code, so wie zusätzliche Unterstützung während der Modellierung und bei der Überprüfung des Modells. Bei der Anwendung von UML Profiles kann es wünschenswert sein, auch diese Schritte durch ein Profile beeinflussen zu können; so möchte man z.B. bei einem UML Profile für CORBA, dass bei dessen Anwendung auch die Generierung von Code derart beeinflusst wird, dass der erzeugte Code den CORBA-Anforderungen genügt. Um solche Semantik einem Profile hinzufügen zu können, stellt Objecteering weitere Elemente zur Verfügung, die, aufgrund der Werkzeugabhängigkeit der zu beeinflussenden Schritte, ebenfalls werkzeugabhängig sind. Kernelement zur Formulierung dieser zusätzlichen Semantik ist die sogenannte J Language, eine einfache Java-ähnliche Sprache. Weitere Elemente dienen dazu, den Profile-Entwickler zu unterstützen und Profile und zusätzliche Semantik zu strukturieren.
Diese proprietären, zusätzlichen Bestandteile sollen im folgenden dargestellt werden.

5.2.4.1 Die Sprache J

J ist eine vom UML Profile Builder zur Verfügung gestellte Sprache, mit deren Hilfe Objecteering gesteuert und parametrisiert werden kann. Sie ist eine interpretierte objektorientierte Sprache, die speziell für den Umgang mit Modellen entwickelt wurde. Ihre Syntax ist ähnlich der von Java, wobei wesentliche Vereinfachungen vorgenommen wurden.
J operiert auf Metamodellebene und geht mit Modellelementen um, die vom Anwender in Objecteering erstellt werden. Unter Verwendung von Metaklassen, Metaassoziationen, Metarollen und Metaattributen kann innerhalb von Modellen navigiert und auf Informationen zugegriffen und können diese manipuliert werden.
Alle Klassen in J spezialisieren die J Klasse *Object*. Der Programmierer erstellt jedoch keine eigenen Klassen, sondern passt nur die bereits zur Verfügung stehenden Klassen an, z.B.

durch die Definition neuer Methoden (mit Ausnahme bei der Erstellung von Work Products, da die als neue Metaklasse betrachtet werden, siehe 5.2.4.6 Work Products). Vorhanden sind einerseits Klassen für grundlegende Datentypen, wie String oder Integer, und andererseits Klassen, die Metaklassen aus dem Metamodell in Objecteering repräsentieren und die dem Programmierer Zugriff auf alle Bestandteile eines Modells bieten. Die Objekte, auf denen J bei der Ausführung arbeitet, sind die vom Endanwender bei der Modellierung erstellten Modellelemente.
Für das Zugreifen oder Speichern von Informationen stehen zur Verfügung:

- Attribute der vorgegebenen Klassen,
- vom Programmierer eingeführte Klassenvariablen,
- Methodenparameter oder
- lokale Variablen.

Eine wesentliche Funktionalität in J ist die Behandlung von Mengen. Mengen können auftreten als Variablen, Attribute oder Parameter und besonders auch bei Elementen, auf die ein Modellelement verweist, z.B. die Attribute einer Instanz der Metaklasse *class*. Neben der üblichen Navigation entlang eines Objektes mit „.“ kann mit „.<“ entlang aller Elemente einer Menge navigiert werden. Würde z.B. eine Variable *attributes* auf eine Menge von Attributen einer Klasse verweisen, so könnte mit *attributes.<getType()* die Methode *getType()* für jedes Element der Menge aufgerufen werden. In diesem Fall erspart man sich die Verwendung einer *for*-Schleife. Würde *attributes* auf kein Element verweisen, also eine leere Menge sein, würde kein Fehler erzeugt, sondern lediglich die Methode *getType()* null mal ausgeführt. Mit Hilfe sogenannter anonymer Methoden kann ein Anweisungsblock auf jedes Element einer Menge ausgeführt werden. Dazu muss lediglich der Referenz auf die Menge der auszuführende Block nachgestellt werden. So würde beispielsweise

```
getAttributes(){
    StdOut.write( name );
}
```

für jedes Element der Menge, die durch *getAttributes()* erhalten wird, den jeweiligen Namen (Attribut *name* der Metaklasse *Attribut*) auf der Standardausgabe (die Konsole) ausgeben.
Es stehen weiterhin Methoden zur Verfügung, um auf Mengen Operationen auszuführen, wie z.B. das Hinzufügen eines Elementes oder die Bestimmung der Größe der Menge. Mit Hilfe der Methode *select()* kann gemäß einem als Parameter übergebenen Booleschen Ausdruck eine Teilmenge ausgewählt werden. Ähnlich kann auf einer Menge die Methode *while()* aufgerufen werden, die solange einen nachfolgenden Anweisungsblock ausführt, wie ein als Parameter übergebener Boolescher Ausdruck den Wert *true* ergibt.
select() und *while()* können auch auf einzelne Elemente angewandt werden und dadurch die reine Funktion als Kontrollstrukturen erfüllen. Als weitere Kontrollstruktur steht lediglich die *if*-Anweisung (einschließlich *else-if* und *else*) zur Verfügung.
Weitere Erläuterungen zur Syntax und zum Umgang mit den Datentypen sind in der Hilfe zu Objecteering unter *J Language User Guide* zu finden. Attribute und Methoden der J-Klassen, die Metaklassen repräsentieren, findet man in *Metamodel User Guide*. Zusätzlich gibt es noch Bibliotheken, die weitere umfangreiche Funktionalität zur Verfügung stellen, wie z.B. den Umgang mit Dateien oder die Anpassung oder Erweiterung der Benutzeroberfläche oder von Diagrammen. Diese Bibliotheken sind im *J Libraries User Guide* der Hilfe beschrieben.
Die Anwendung und Ausführung von J erfolgt mit Hilfe sogenannter J Methods (siehe 5.2.4.3.), dort sind auch Beispiele zu J zu finden. Eingegeben wird der J Codes in Notes.

5.2.4.2 Notes und Note Types

Notes ergänzen Modelle oder einzelne Modellelemente durch beschreibenden Text, z.B. Kommentar oder einem Modellelement zugeordneter Code. Diese Beschreibungen sind in verschiedene Typen gegliedert, welche durch den sogenannten Note Type der Note festgelegt werden. Die Note Type legt die Bedeutung der Beschreibung fest, also z.B. Kommentar oder Code.
In einem Profile können neue Typen von Note Types definiert werden, z.B. um bestimmten Modellelementen zusätzlichen Text mit spezieller Bedeutung im Profile zuweisen zu können. Dabei ist festzulegen, welche Metaklassen oder welche Stereotypes mit Notes dieses Note Types versehen werden können. Dazu wird ein Note Type im UML Profile Builder entweder einer Metaklassenreferenz oder einem Stereotype zugeordnet.

Eigenschaften:

- *Name:*
  Name des Note Types.

Beispiel:
In Abbildung 5-8 wurde ein Note Type *JavaCode* der Metaklasse *Operation* hinzugefügt. In Notes dieses Typs soll der Modellierer Java-Code für eine Methode eingeben, der dann bei Generierung des Codes für die jeweilige Klasse miteinbezogen werden soll.
Als Beispiel der Zuweisung eines Note Types zu einem Stereotype wird der Note Type *SQLCode* dem Stereotype *persistent* zugeordnet. Denkbar wäre hier, dass der Modellierer SQL-Anweisungen eingeben kann, in unserem Beispiel soll aber keine weitere Verwendung der Notes dieses Typs erfolgen.

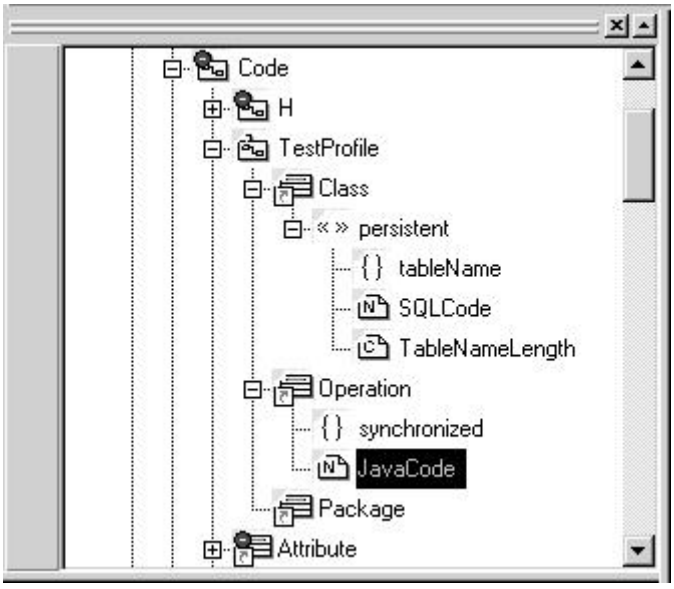

Abbildung 5-8: Ein Note Type *JavaCode* wurde der Referenz auf die Metaklasse *Operation* hinzugefügt.

5.2.4.3 J Methods

J Methods sind in der Sprache J (siehe 5.2.4.1) geschriebene Methoden, die einer Metamodellklasse zugewiesen werden können. Sie werden intern der J-Klasse zugeordnet, die die jeweilige Metamodellklasse repräsentiert.
Aufgerufen werden sie bei Anwendung des Profiles, entweder Profile-intern während der Modellierung, um Operationen über dem jeweiligen Modellelement auszuführen, oder direkt vom Modellierer durch ein zugeordnetes Kommando („Command“, siehe 5.2.4.8) im Kontextmenü des jeweiligen Modellelements. Kommandos in Kontextmenüs werden innerhalb von Modules (siehe 5.2.4.7) definiert und rufen direkt eine zugehörige J Method auf. Die assoziierte J Method muss als *public* deklariert und parameterlos sein.
Wie alle Elemente, kann auch eine J Method mit Tagged Values und Notes versehen werden, die auf die J Method selbst angewandt werden. Dies wird verwendet, um den J-Code zu beinhalten: der Code einer J Method wird beinhaltet von einer Note vom Note Type *JCode*, die der J Method zugeordnet ist.

Eigenschaften:

- *Name*:
  Name der J Method
- *Visibility*:
  Sichtbarkeit der Methode. Möglich sind *public*, *protected* und *private*.
- *Parameter*:
  Parameter der J Method mit folgenden Eigenschaften:
  - *Name*:
    Name des Parameters.
  - *Parameter passing mode*:
    Übergabemodus; möglich sind *in*, *out* und *inout*.
  - *Class*:
    Typ des Parameters. Es muss sich dabei um eine J-Klasse handeln, d.h. entweder ein einfacher Datentyp oder eine Metaklasse.
  - *Set*:
    Ob der Parameter einen einzelnen Wert oder eine Menge von Werten enthält.
- *Return parameter*:
  Rückgabewert der J Method mit folgenden Eigenschaften:
  - *Class*:
    Typ des Rückgabewertes; eine J-Klasse.
  - *Set*:
    Ob der Parameter einen einzelnen Wert oder eine Menge von Werten enthält.

Unter der Karteikarte *Note* im Dialog zur J Method kann man die bereits vorgegebene Note vom Note Type *JCode* bearbeiten und dort den eigenen Code eingeben.

Beispiel:
In Abbildung 5-9 wurden dem Metaklassen Methoden zur Generierung von Code hinzugefügt. Jedes Metaklasse, die bei der Codegenerierung berücksichtigt werden soll, erhält eine J Method *generate()*. Diese erstellen jeweils Code aus ihren Attributen (z.B. dem Namen des Elementes) und rufen die Methoden *generate()* von enthaltenen Metaklassen auf (z.B. *Package* von *Package* und *Class*). Auf diese Weise wird vom obersten Package bis zu den Attributen einer Klasse abgestiegen. Die Metaklasse *Operation* erhält zusätzlich eine J Method *getCode()* um Code einzulesen, der vom Modellierer in eine Note *JavaCode* (siehe 5.2.4.2) eingegeben wurde, *Attibute* erhält zusätzlich eine J Method *getType()*, um den Typ eines Attributs zu bestimmen.

Die Abbildung zeigt den Dialog der J Method *generate()* für die Metaklasse *Package*. Ihre Sichtbarkeit ist als *public* angegeben. Der Dialog enthält Knöpfe, durch die Dialoge zum Hinzufügen oder Ändern von Parametern und Rückgabewerten der J Method aufgerufen werden können, sowie eine Karteikarte *Notes*, in der der J Method Notes hinzugefügt werden können.

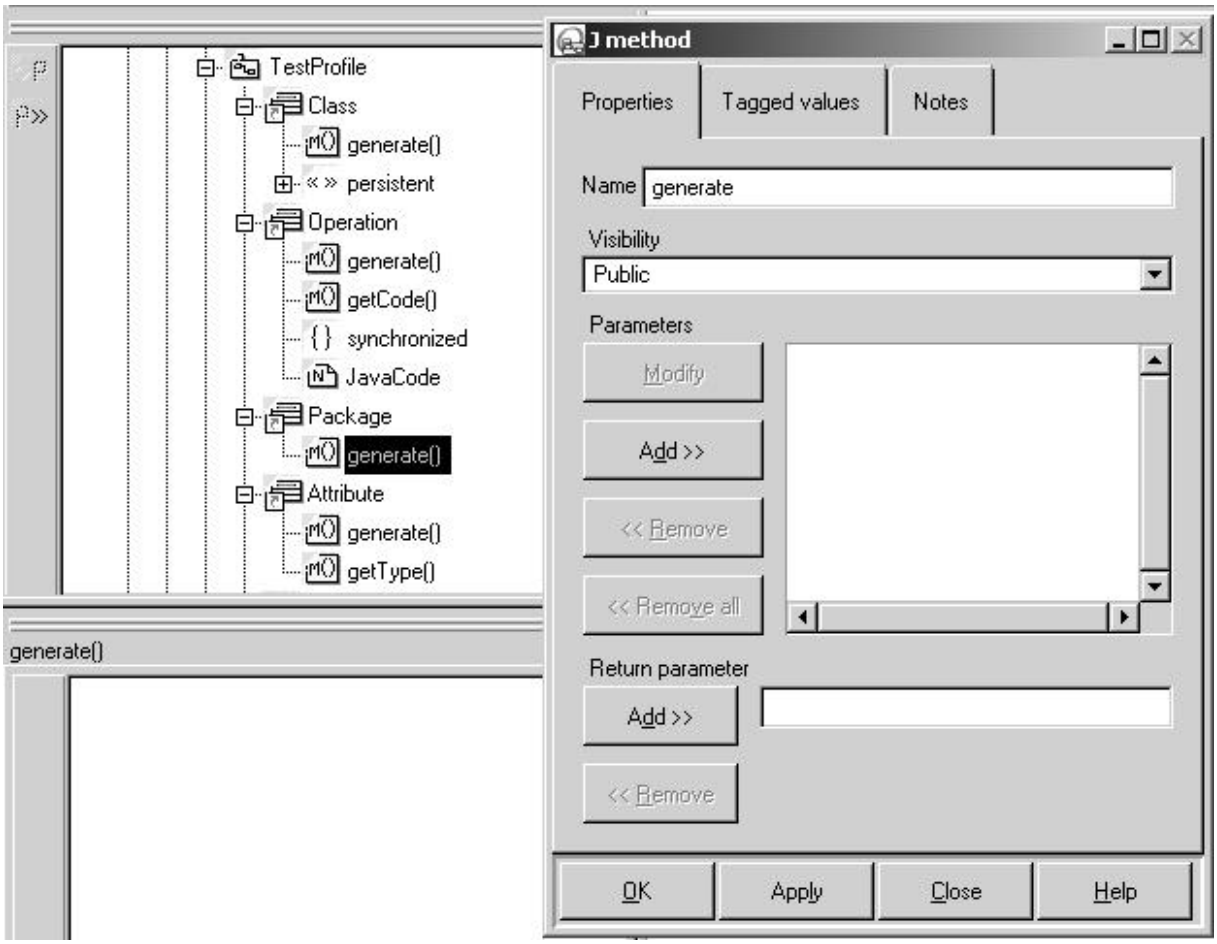

Abbildung 5-9: Erstellung einer J Method *generate()* für die Metaklasse *Package*.

In Abbildung 5-10 wird in die vorgegeben Note vom Note Type *JCode* der J Code (siehe 5.2.4.1) der J Method *generate()* der Metaklasse *Package* eingegeben. Darin wird zuerst eine Nachricht auf der Konsole ausgegeben. Sie enthält durch das Attribut *Name* den Namen des Package. Durch *NL* wird ein Zeilenvorschub bewirkt. Die Metaklasse *Package* besitzt die Attribute *OwnedElementPackage* und *OwnedElementClass*, die eine Menge alle im Package enthaltenen Packages bzw. enthaltenen Klassen bezeichnen. Durch den Operator < wird von jedem Element aus dieser Menge dessen J Method *generate()* aufgerufen. Aus dem Beispiel ist ersichtlich, wie einfach sich der Umgang mit Metaelementen durch die Sprache J gestaltet.

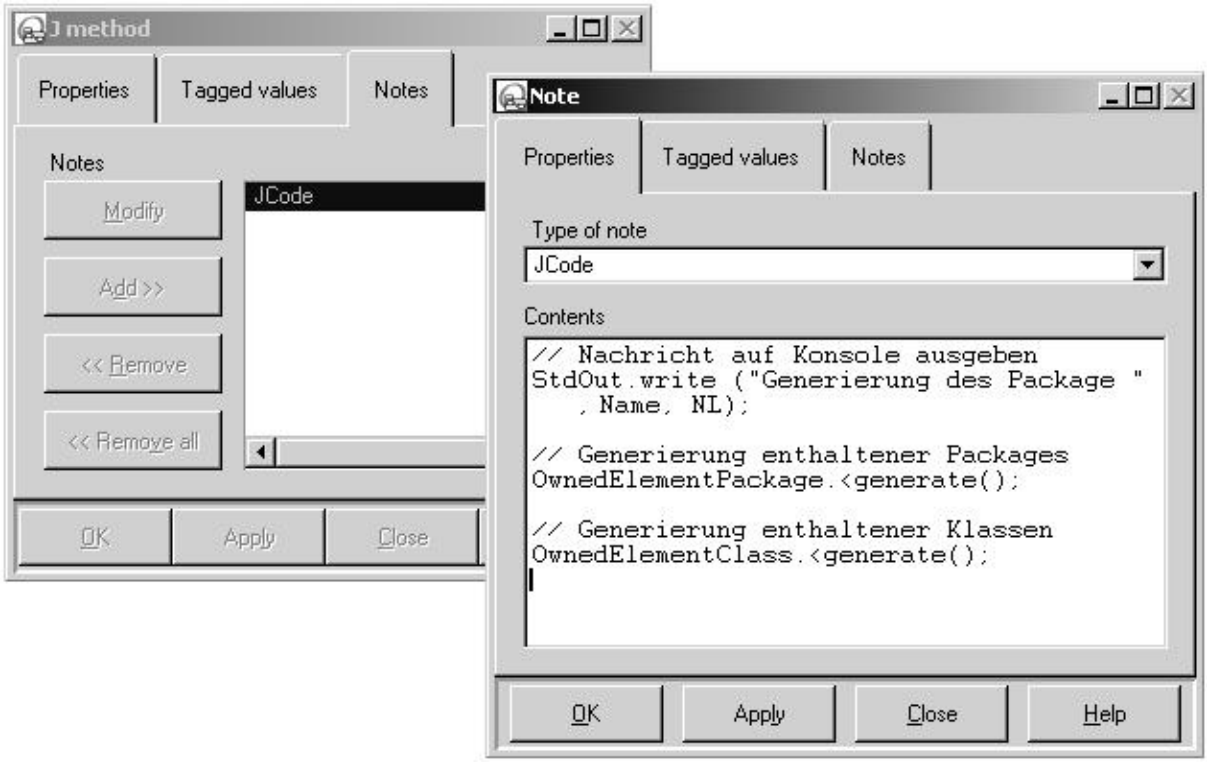

Abbildung 5-10: Der Code der J Method *generate()* zur Metaklasse *Package* wird in eine zugehörige Note des Note Types *JCode* eingegeben.

Der Dialog zur Definition von Parametern einer J Method ist in Abbildung 5-11 abgebildet. In unserem Beispiel erhält aber keine J Method Parameter.

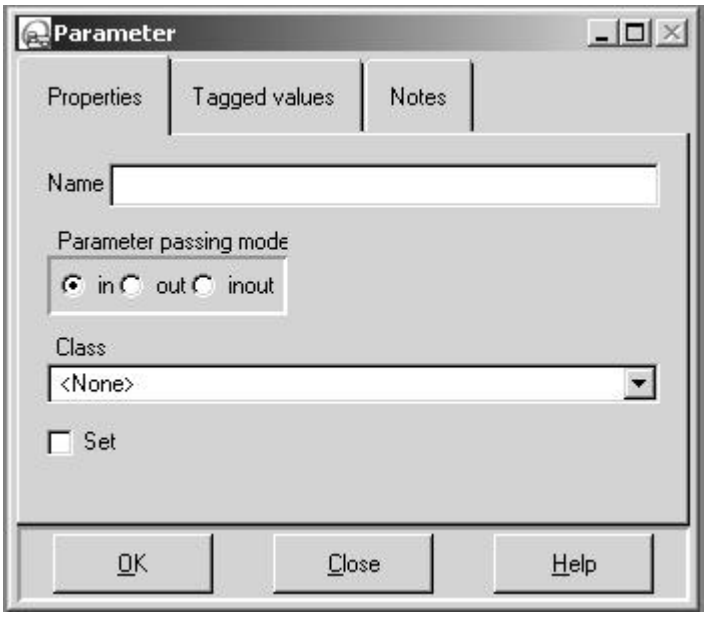

Abbildung 5-11: Dialog zur Angabe von Parametern einer J Method.

Da für jede Klasse eine Datei mit Code erstellt werden soll, wird der Code in der J Method *generate()* der Metaklasse *Class* zusammengesetzt. Die von dort aufgerufenen J Methods *generate()* der Metaklassen O*peration* und *Attribute* sollen daher einen String mit den jeweils erzeugen Codefragmenten zurückliefern. In Abbildung 5-12 wird der Rückgabewert der J Method *generate()* der Metaklasse *Attribute* definiert.

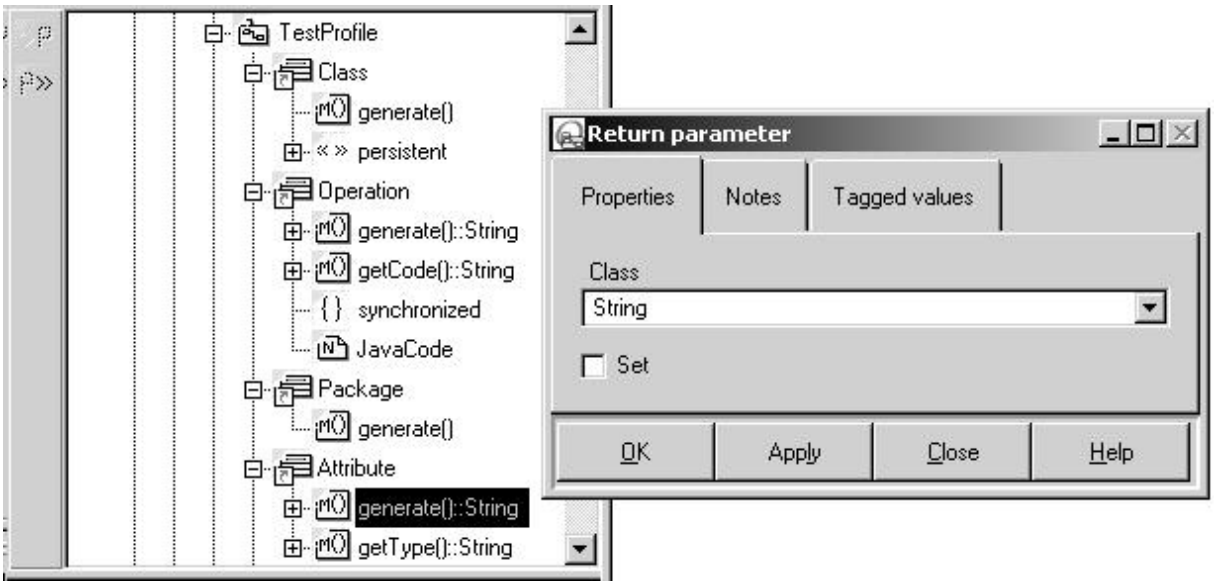

Abbildung 5-12: Hinzufügen eines Rückgabewertes vom Typ String zur J Method *getType()*.

Ein weiteres Beispiel ist der Code der J Method *generate()* der Metaklasse *Class*. Es wird eine String *content* erzeugt, der den generierten Quellcode enthält:

```
1     String content;
2
3     content.strcat ("// --------------------", NL);
4     content.strcat ("// Class ", Name, NL);
5     content.strcat ("// --------------------", NL);
6
7     content.strcat ("class ", Name, NL);
8     content.strcat ("{", NL);
9
10    PartAttribute{
11
12        content.strcat(generate());
13    }
14
15    PartOperation{
16
17        content.strcat(generate());
18    }
19
20    content.strcat ("}", NL);
```

In Zeile 1 wird eine lokale Variable *content* vom Typ String erzeugt. Über die J Method *strcat()* der (in J vorgegebenen) Klasse *String* werden diesem String einzelne Bestandteile des zu generierenden Quellcodes angehängt. In den Zeilen 3 bis 5 wird ein Kommentar für die Klasse erstellt, der über das Attribut *Name* (Zeile 4) den Klassennamen enthält. Zeile 7 und 8 fügen den Kopf der Klasse hinzu. In Zeile 10 wird das Attribut *PartAttribute* der Klasse *Class* verwendet, das die Menge aller Attribute der jeweiligen Klasse enthält. Durch Nachstellen einer anonymen Methode (10 bis 13) wird diese für jedes Element aus der erhaltenen Menge ausgeführt. Dadurch wird für jedes Attribut dessen Methode *generate()* aufgerufen und der Rückgabewert (vom Typ String, siehe Abbildung 5-12) *content* angehängt. Analog Zeile 15 bis 18 für die Methoden der Klasse. Zuletzt wird in Zeile 20 die erzeugte Java-Klasse durch *}* vervollständigt.

Der hier aufgeführte Code der J method *generate()* ist nicht vollständig, da zu einem späteren Zeitpunkt noch Anweisungen bezüglich des zu erstellenden Work Products (siehe 5.2.4.6) hinzugefügt werden müssen.

5.2.4.4 J Attributes

J Attributes sind Metaattribute für die J-Klassen, die Metamodellelemente repräsentieren. Sie werden einer Referenz auf eine Metaklasse zugeordnet und dienen der Nutzung durch J Methods. In der vorliegenden Version von Objecteering sind J Attributes stets Klassenvariablen und es besteht keine Möglichkeit, Instanzenvariablen zu erzeugen. Sie sind nicht persistent.

Eigenschaften:

- *Name*:
  Name des J Attributes.
- *Visibility*:
  Sichtbarkeit des Attributes. Möglich sind *public*, *protected* und *private*.
- *Class*:
  Typ des J Attributes. Zur Auswahl stehen alle J-Klassen, d.h. einfache Datentypen oder Metaklassen.
- *Set*:
  Ob das J Attribut ein Element seines Typs oder eine Menge seine Typs enthalten kann.
- *Initial value*:
  Wert, mit dem das Metaattribut initialisiert wird.

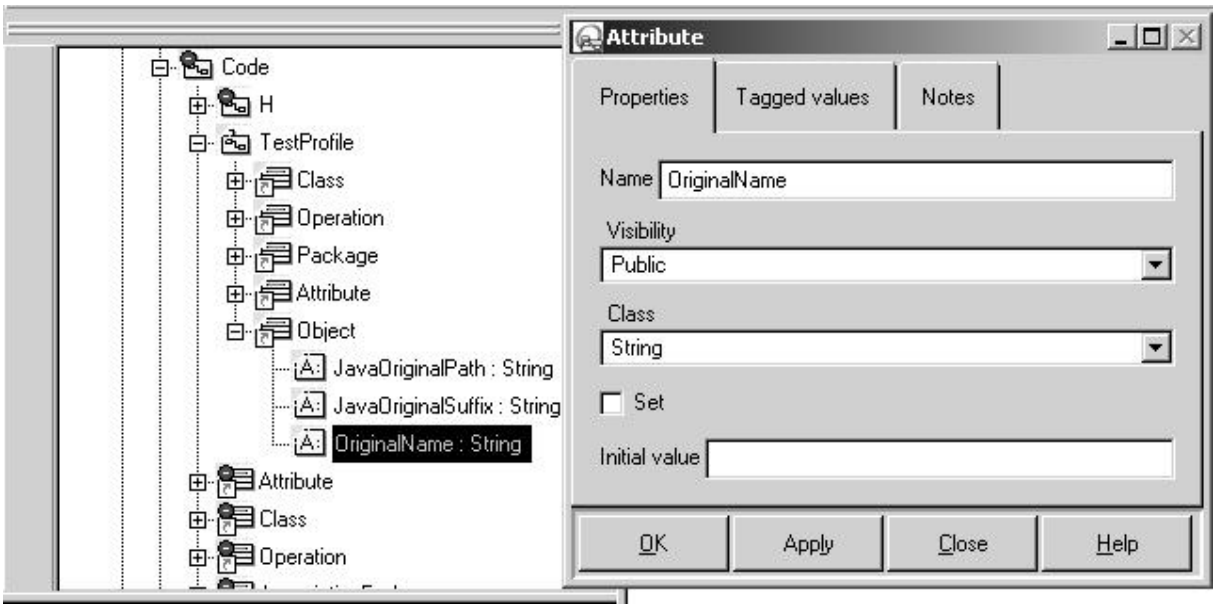

Abbildung 5-13: Erstellung eines J Attributes *OriginalName* für die Metaklasse *Object*.

Beispiel:
Für die Verwaltung generierter Dateien werden drei J Attributes benötigt. Sie speichern bei der Generierung Informationen zur Datei und sollen bei durch alle Elementen zugreifbar sein, weshalb sie der Metaklasse *Object* zugeordnet werden, da diese von jeder Klasse erweitert wird. Dazu wird eine Referenz auf die Metaklasse *Object* erstellt und dieser die J Attribute

zugewiesen (Abbildung 5-13). Das J Attribut *JavaOriginalPath* speichert den zum Zeitpunkt der Generierung aktuellen Pfad, *JavaOriginalSuffix* die aktuelle Dateiendung und *OriginalName* den aktuellen Namen des zugehörigen Modellelementes. Alle drei J Attribute sind *public* und vom Typ String.

5.2.4.5 Module Parameter

Module Parameter werden direkt einem Profile zugeordnet und sind Parameter des Profiles, die der Endnutzer bei Anwendung des Profiles (unter Verwendung eines Modules, siehe 5.2.4.7) konfigurieren kann. Sie sind in J Methods zugreifbar und machen prinzipielle Einstellungen des Endanwenders zum Profile möglich.

Eigenschaften:

- *Name*:
  Name des Module Parameters.
- *Type*:
  Typ des Parameters. Zur Wahl stehen *Boolean*, *String* und *Enumeration*. Als Strings werden auch die besonders behandelten Strings *File open*, *File save*, *Directory* und *Password* angeboten. Bei Auswahl von Enumeration ist Wertebereich des Module Parameters eine Menge von Strings, welche im Dialogfenster in der Karteikarte *Enumeration values* eingegeben werden können.
- *Group*:
  Name einer Gruppe, in die der Parameter eingeordnet wird. Parameter werden dem Anwender des Profiles nach Gruppen sortiert angezeigt. Es kann sowohl ein neuer Gruppenname angegeben werden, wie auch aus der Liste der vorhandenen Namen einer gewählt.

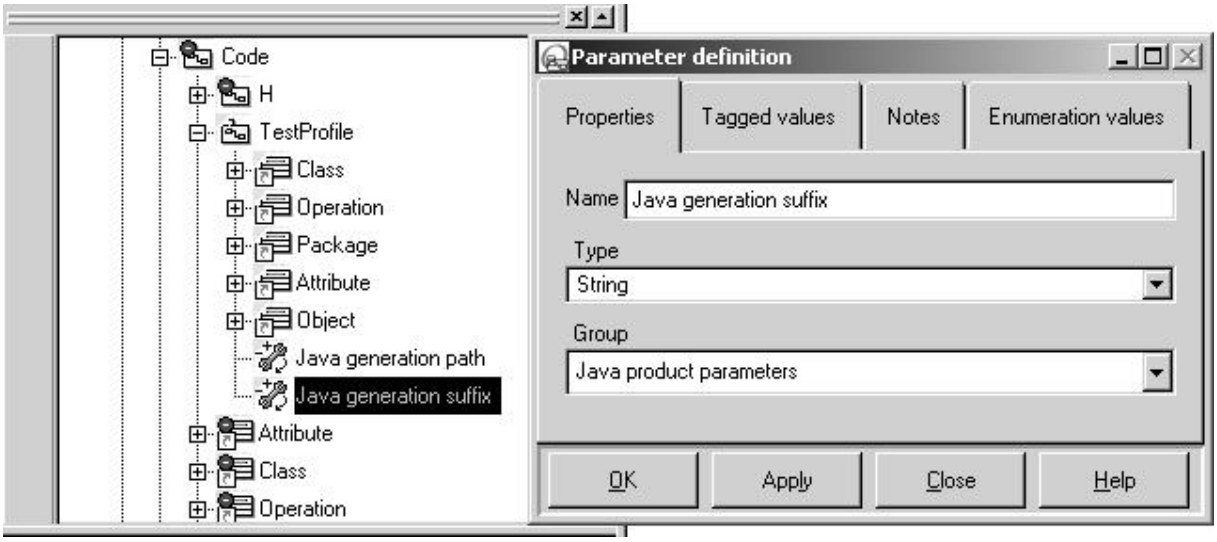

Abbildung 5-14: Hinzufügen eines Parameters *Java generation suffix* zum Profile *TestProfile*

Beispiel:
Für die Generierung der Dateien mit Java-Code soll für den Anwender des Profiles der Pfad und die Endung der Dateien konfigurierbar sein. Deshalb werden dem Profile zwei Module Parameter, *Java generation path* und *Java generation suffix*, zugeordnet (Abbildung 5-14).

5.2.4.6 Work Products

Work Products repräsentieren ein externes Ergebnis eines Generierungsprozesses, wie z.B. eine Datei mit generiertem Quellcode, ein Makefile oder generierte Dokumentation.
Im UML Modeler werden sie durch ein „Creation Icon“, d.h. einen Knopf, mit dessen Hilfe sie erstellt werden können, und ein eigenes Dialogfenster repräsentiert.
Sie sind einer Metaklasse zugeordnet, für die das Creation Icon erscheint und über der das neue Work Product erstellt werden. Im UML Profile Builder können neue Typen von Work Products definiert werden.

Eigenschaften:

- *Name*:
  Name des Work Products.
- *Label, Help Bubble, StateBar*:
  Text zur Anzeige im UML Modeler als Hilfe für den Anwender.
- *Bitmap name*:
  Name eines Icons zur Repräsentation des Work Products im UML Modeler. Das Icon dient als Knopf zur Erstellung eines Work Products dieses Typs.
- *Model element concerned*:
  Modellelemente, von denen aus das Work Product erstellt werden kann. Ist eines der Modellelemente im UML Modeler markiert, so wird das Icon (*Bitmap name*) zur Erstellung des Work Products angezeigt.

Dem Work Product können Metaattribute zugewiesen werden, welche vom Typ String oder Boolean sein müssen, wobei auch die speziellen Strings *File open*, *File save*, *Directory* und *Password* zur Auswahl stehen. Die möglichen Typen stimmen mit denen von Module Parameter überein. Es kann dem Metaattribut ein Module Parameter (siehe 5.2.4.5) zugeordnet sein, dessen Wert als Standardwert übernommen wird. Wird im UML Modeler ein neues Work Product erstellt, so erscheinen die Metaattribute automatisch im zugehörigen Dialogfenster und können dort vom Endanwender modifiziert und dem konkreten Work Product angepasst werden.
Das Work Product wird als neue Metaklasse betrachtet, wodurch eine Referenz auf diese Metaklasse erstellt werden kann und dieser weitere Elemente hinzugefügt werden können, insbesondere J Methods. Die Metaklasse erbt vordefinierte Methoden, z.B. für den Umgang mit Dateien, die überschrieben werden können. Weiterhin stehen vordefinierte Methoden zur Verfügung, die genutzt werden können um z.B. Verknüpfungen zu Dateien zu erstellen, so dass diese bei Änderung des Modells aktualisiert werden oder um das Work Product mit Generation Templates oder Documentation Templates zu verknüpfen. J Attributes, die dieser Metaklassenreferenz zugewiesen werden, sind zu unterscheiden von Metaattributen, die direkt einem Typ von Work Product zugewiesen werden (s.o.), und werden nicht im UML Modeler angezeigt, sondern sind nur für den internen Gebrauch in J Methods.
Beispiel:
Es wird im Pofile *TestProfile* ein neuer Typ Work Product erstellt, der generierte Java-Code-Dateien repräsentiert (Abbildung 5-15). Die Generierung kann von Packages oder von Klassen ausgehen. Zur Repräsentation wir ein Icon  angegeben. Dem Work Product werden zwei Metaattribute, *path* und *suffix*, zugeordnet (Abbildung 5-16), wodurch der Anwender für jede Datei Pfad und Dateiendung festlegen kann. Als Standardwerte werden die Module Parameter *Java generation path* und *Java generation suffix* (siehe 5.2.4.5) festgelegt, wodurch dem Anwender deren Werte (die er ebenfalls modifizieren kann) vorgeschlagen werden.

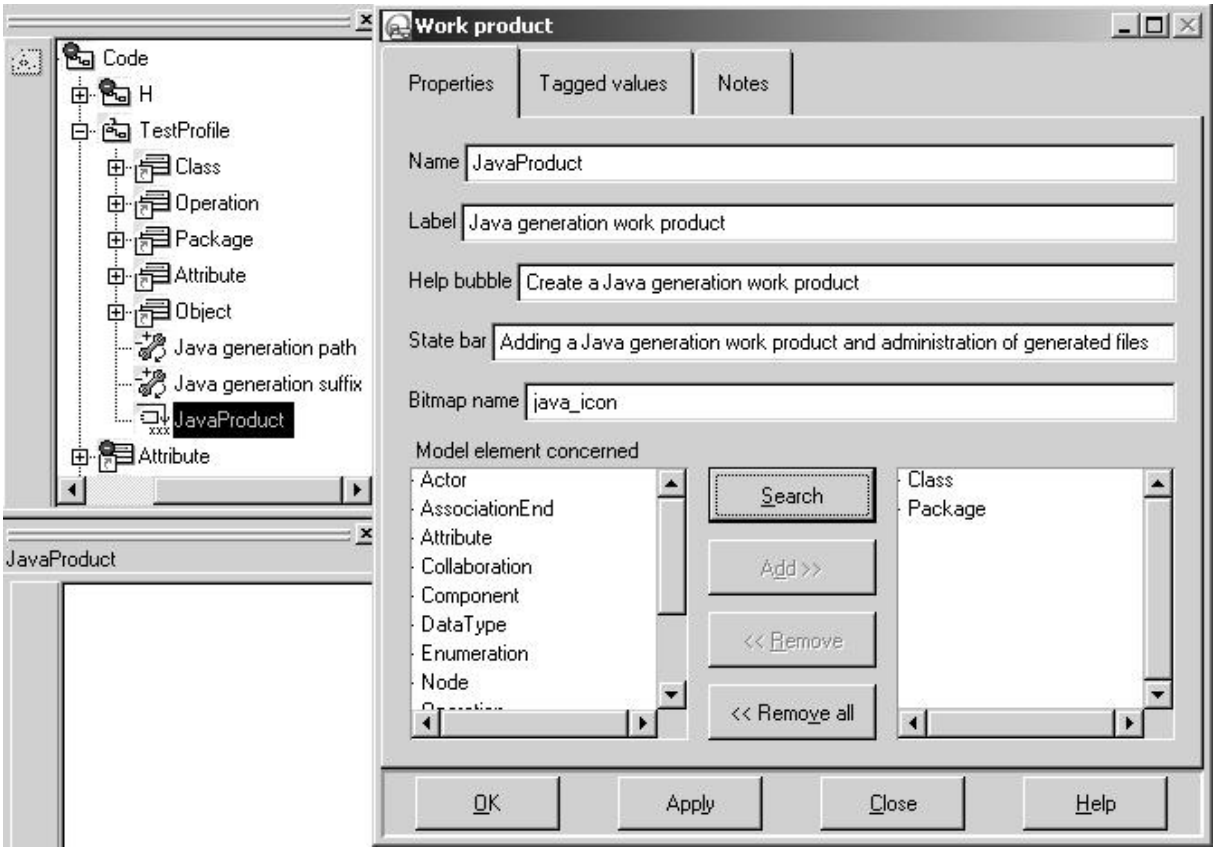

Abbildung 5-15: Erstellen eines Work Products *JavaProduct* im Profile *TestProfile*.

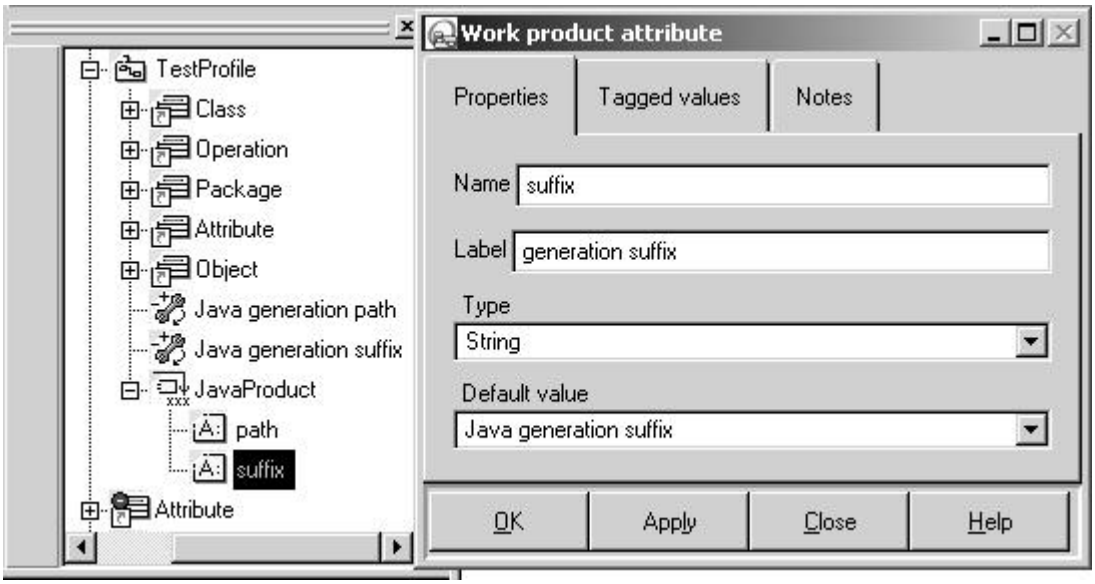

Abbildung 5-16: Erstellen des Metaattributs *suffix* für das Work Product *JavaProduct*.

Um dem Work Product J Methods zuzuordnen, muss eine Referenz auf die (neue) Metaklasse *JavaProduct* erstellt werden. Zum einen sind vorgegebene J Methods, die bei Anwendung des Work Products automatisch aufgerufen werden zu überschreiben. Dazu gehören Methoden zur Initialisierung, zur Aktualisierung und, da ein Package mehrere Klassen enthalten kann und somit mehrere Work Products gleichzeitig erstellt werden, zur Vermehrung. Zum Anderen benötigt das Work Product eine Methode, mit der die Codegenerierung gestartet wird (hier *generate()* genannt), und welche die Methoden *generate()* aus den Metaklassen

Package bzw. *Class* aufruft. Es lassen sich mit wenigen Befehlen weitere Methoden hinzufügen, um einen internen Editor aufzurufen, mit dem erstellte Dateien betrachtet werden können bzw. zum Aufruf eines externen Editors (z.B. Notepad unter Windows) zum Modifizieren der Dateien.

5.2.4.7 Modules

Modules sind funktionale Einheiten, die bei der Modellierung im UML Modeler für ein Projekt ausgewählt werden können, um für ein bestimmtes Gebiet unter Nutzung von UML Profiles Element oder Funktionalität bereitzustellen. Sie sind die umgebende Einheit für UML Profiles, in der diese ausgetauscht, in den UML Modeler eingebunden und durch den Endanwender genutzt werden. Ein Modell kann beliebig viel Modules verwenden. Es können nicht nur im UML Profile Builder neue Modules erstellt werden, sondern es stehen auch mehrere mit dem Werkzeug mitgelieferte Modules zur Verfügung, z.B. für C++-Generierung, für XMI, für Oracle und andere.
Da in Objecteering bei der Verwendung von Profiles, und damit von Modules, die Generierung von Code oder Dokumentation als Zweck im Vordergrund steht, stellt ein Module typischerweise bereit:

- Menüs und Befehle für den Generator (unter der Verwendung von Commands, siehe 5.2.4.8),
- ein System von Tagged Values, um für einzelne Modellelemente Eigenschaften im Modell spezifizieren zu können (z.B. zur Kennzeichnung virtueller Methoden für die Generierung von C++-Code),
- Note Types zum Hinzufügen verschiedener Zusatzinformationen für den Generator zu Modellelementen,
- Dokument Templates oder Generation Templates und
- die Möglichkeit zur Erstellung von Work Products.

Zusätzlich werden alle UML Erweiterungsmechanismen unterstützt.

Eigenschaften:

- *Name*:
  Name des Modules.
- *Label*:
  Bezeichnung zur Repräsentation des Modules im UML Modeler.
- *Working directory*:
  Verzeichnis zum Speichern des Modules.
- *Number of major version, number of major version, Release information*:
  Zur Versionsnummerierung des Modules.
- *Minimum binary version compatibility*:
  Niedrigste kompatible Version.
- *Mask parents*:
  Ob Eltern des Modules verborgen werden sollen.

Ein Module verwendet ein oder mehrere Profiles. Dabei stehen mehrere Möglichkeiten des Verfügung. Durch Referenzieren („reference“) eines Profiles werden alle dessen Bestandteile dem Module zugänglich. Im Unterschied dazu gibt es die Möglichkeit ein Profile zu „nutzen“ („use“), wodurch nur grundlegende Bestandteile des Profiles, nämlich Stereotypes, Tag Definitions, Notes und J Code, jedoch nicht Work Products, Generation Templates, Document

Templates und Module Parameter. Schließlich kann noch eines der referenzierten Profiles als „Installation Profile“ ausgewählt werden. Diese kann Methoden für Aktionen über dem Module spezifizieren, die ausgeführt werden bei Empfangen, Installieren, Auswählen, Abwählen oder Deinstallieren des Modules. Verwendete Profiles werden im Eigenschaftsfenster hinzugefügt.
Ein Module kann ein anderes Module spezialisieren. Dabei werden alle verwendeten Profiles und alle Module Parameter und deren Standardwerte vererbt. Ein spezialisiertes Module wird im Eigenschaftsfenster hinzugefügt.
Es können dem Module Commands hinzugefügt werden (siehe 5.2.4.8) und Standardwerte für Module Parameter festgelegt werden.
Nach Fertigstellung des Modules ist es schließlich zur Weitergabe bzw. zur Einbindung und Anwendung im UML Modeler zu packen, indem der Befehl „Package. Dabei werden alle verwendeten Ressourcen, wie z.B. Icons oder externe zusätzliche Dateien, mit eingebunden. Es kann festgelegt werden, ob enthaltener J Code verborgen werden soll und ob das gepackte Module nur gelesen oder auch modifiziert werden kann, sowie eine Versionsnummer.
Um den Parametern Standardwerte zuzuweisen, ist durch den Befehl *Configure UML Profiling Project* ein Dialog (Abbildung 5-18) aufrufbar, welcher dem Dialog im UML Modeler entspricht, in dem der Endanwender des Modules die Parameter konfigurieren kann. Den verfügbaren Modules sind hierin die Parameter nach Gruppen sortiert zugeordnet.

Beispiel:

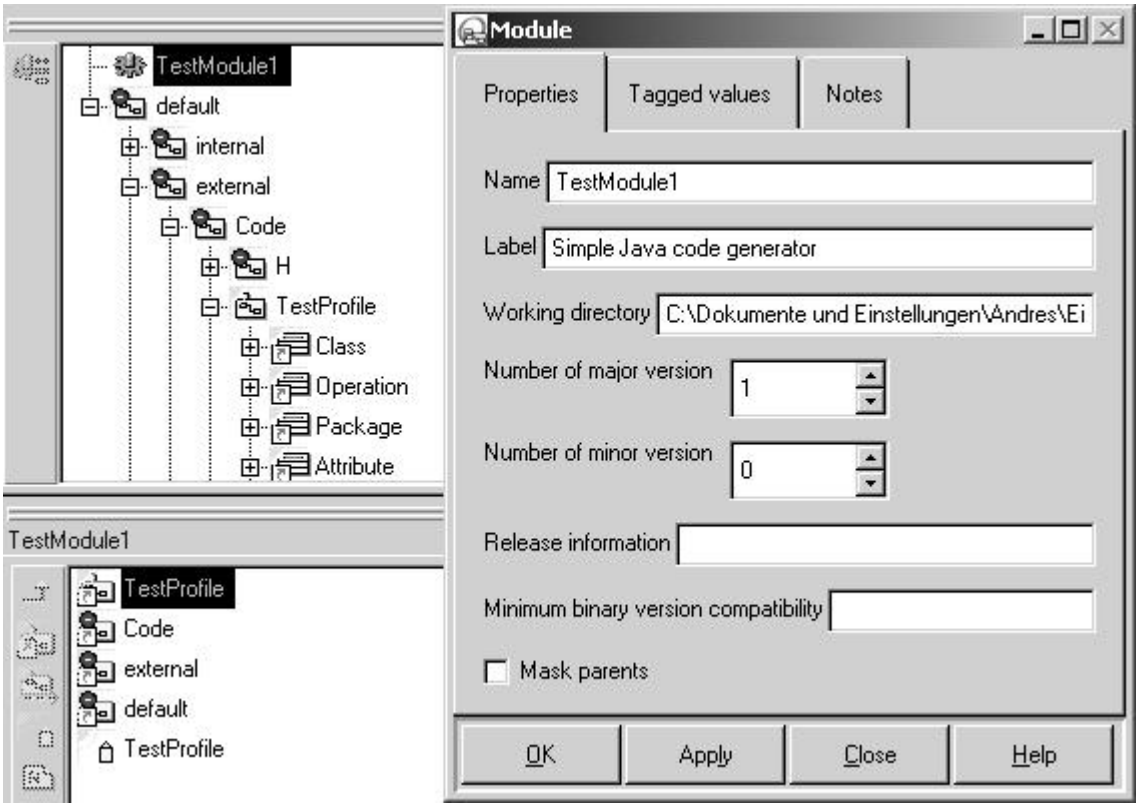

Abbildung 5-17: Dialog zur Erstellung eines Modules.

Es werden dem aktuellen Projekt TestProfileProject ein Module *TestModule1* hinzugefügt und im Eigenschaftsfenster dem Module die verwendeten Profiles zugeordnet (Abbildung 5-17). Das Module referenziert das Profile *TestProfile*, wodurch auch alle dessen Eltern referenziert werden. Zusätzlich wurde *TestProfile* auch als Installation Profile ausgewählt. Der Referenz auf die Metaklasse *Object* werden dadurch automatisch J Methods hinzugefügt,

denen Code hinzugefügt werden kann und die bei Aktionen über dem Module automatisch ausgeführt werden.
Abbildung 5-18 zeigt den Dialog zur Eingabe von Standardwerten für die Module Parameter. Der Dialog enthält die Möglichkeit zur Konfiguration aller Modules, die im aktuellen Projekt aktiviert sind, einschließlich der Einstellungen für den UML Profile Builder und den UML Modeler selbst. Das neu erstellte Module wird unter seinem Label *Simple Java code generator V1.0* angezeigt. Darunter befinden sich die enthaltenen Gruppen von Module Parameter. Die Gruppe *GroupExternalEdition* wurde vom Profile *Code* geerbt.

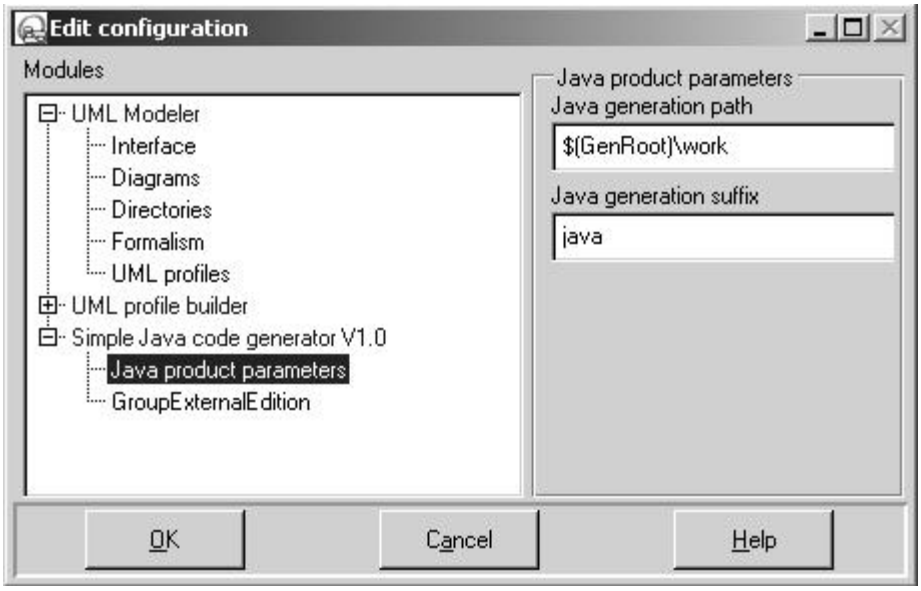

Abbildung 5-18: Dialog zur Einstellung von (Standard-)werten für Module Parameter aller aktiven Modules.

5.2.4.8 Commands

Commands werden den Modules zugeordnet. Es handelt sich dabei um Befehle, die bei Anwendung des Modules in Kontextmenüs von Modellelementen angezeigt werden und die bei Aufruf eine J Method aus einem Profile, das im Module enthalten ist, aktivieren.
Aufgerufen wird eine J Method, die einer Referenz auf eine Metaklasse zugeordnet ist. Der Befehl wird nur im Kontextmenü von Modellelementen der entsprechenden Metaklasse angezeigt. Die J Method muss parameterlos sein und von der Sichtbarkeit *public*.

Eigenschaften:

- *Name*:
  Name des Commands.
- *Label*:
  Name des zugehörigen Befehls in Kontextmenüs.
- *Can be activated in editing mode*:
  Ob das Command während der Erstellung von Modellen aktivierbar ist.
- *Can be activated in consulting mode*:
  Ob das Command während der Validierung von Modellen aktivierbar ist.
- *Can be activated on a read only object*:
  Ob das Command auf Objekten aktivierbar ist, die nur lesbar sind.

- *Running UML Profile*:
  Profile, das die zu referenzierende J Method enthält. Das Profile muss vom Module verwendet werden („reference“ oder „use“, siehe 5.2.4.7).
- *J Method*:
  Name der J Method aus dem *Running UML Profile*, die vom Command ausgeführt wird. Sie muss parameterlos und public sein.

Beispiel:
In Abbildung 5-19 wurden dem Module *TestModule* drei Commands hinzugefügt. Alle drei verweisen auf J Methods der Metaklasse *JavaProduct* im Profile *TestProfile*. Abgebildet ist das Dialogfenster zum Command *generate*, welches die J Method *generate()* aufruft und dazu dient, die Generierung von Code zu starten. Da die referenzierte J Method aus der Metaklasse *JavaProduct* stammt, ist das Command in dessen Kontextmenü zu finden und aufzurufen. Das Command *visualize* dient dazu, die generiert Datei, die durch das Work Product repräsentiert wird, in einem externen Editor einsehbar zu machen, das Command *edit* ruft einen externen Editor zum editieren der Datei auf.

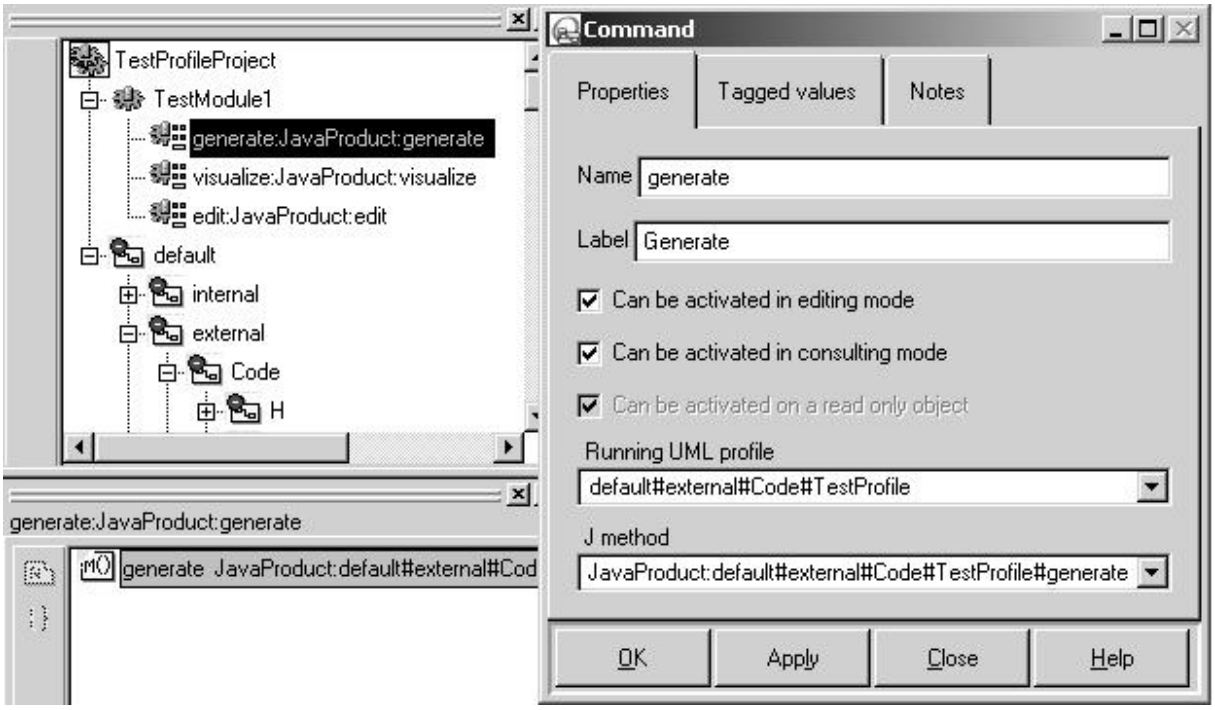

Abbildung 5-19: Erstellen eines Commands *generate* im Module *TestModule1*.

5.2.4.9 Document Templates und Generation Templates

Document Templates und Generation Templates sind Mechanismen zur Beschreibung der Struktur von textbasierten Zielobjekten, in der Regel zu generierende Dokumentation oder Code. Sie dienen dazu, durch Automatismen den Programmieraufwand für den Entwickler des Profiles zu reduzieren.
Templates bestehen aus einer Hierarchie von einzelnen Elementen, den sogenannten Document Items bzw. Generation Items. Die Items repräsentieren einzelne logische Bestandteile des zu generierenden Objektes. Durch Angabe der Metaklasse, auf die das Item Bezug nimmt, kann unter Berücksichtigung deren Eigenschaften automatisch Dokumentation

bzw. Code erstellt werden. Um Modellelementen zusätzliche Eigenschaften oder Beschreibungen zuzuweisen, werden hauptsächlich verschiedene Typen von Notes verwendet. Zum Erreichen unterschiedlicher Behandlung von einzelnen Instanzen derselben Metaklasse können Tagged Values genutzt werden. Mit Hilfe der J Language werden Regeln für ein Item festgelegt. Dabei ist es möglich, dem Item J Methods zuzuweisen, die entweder als vor, während oder nach der Generierung des Textes aus dem Item ausführbar spezifizierbar sind und die weitere Modifikation und Anpassung des erzeugten Textes erlauben, sowie J Methods zur Filterung von Eigenschaften, die nicht automatisch in den erzeugten Text mit einbezogen werden sollen.
Als Format für das Ergebnis der Erzeugung von Text ist unabhängig vom verwendeten Template zwischen den Formaten Postscript, RTF, HTML oder ASCII wählbar.

Beispiel:
Wird der oben erstellte Generator für Java-Code umfangreicher, so ist es sinnvoll, zwecks Übersichtlichkeit und Strukturierung ein Generation Template zu nutzen. Einzelne Generation Items wären z.B. Package, Klasse, ein Attribut, usw. Aufgrund des Umfangs soll hier kein explizites Beispiel demonstriert werden.

5.3 *Anwendung eines Profiles mit UML Modeler*

Der UML Modeler ist die eigentliche Anwendung zur Modellierung. UML Profiles werden in Form von Modules in den UML Modeler eingebunden. Anhand des in 5.2 erstellten Beispiels soll nun die Anwendung von UML Profiles demonstriert werden.

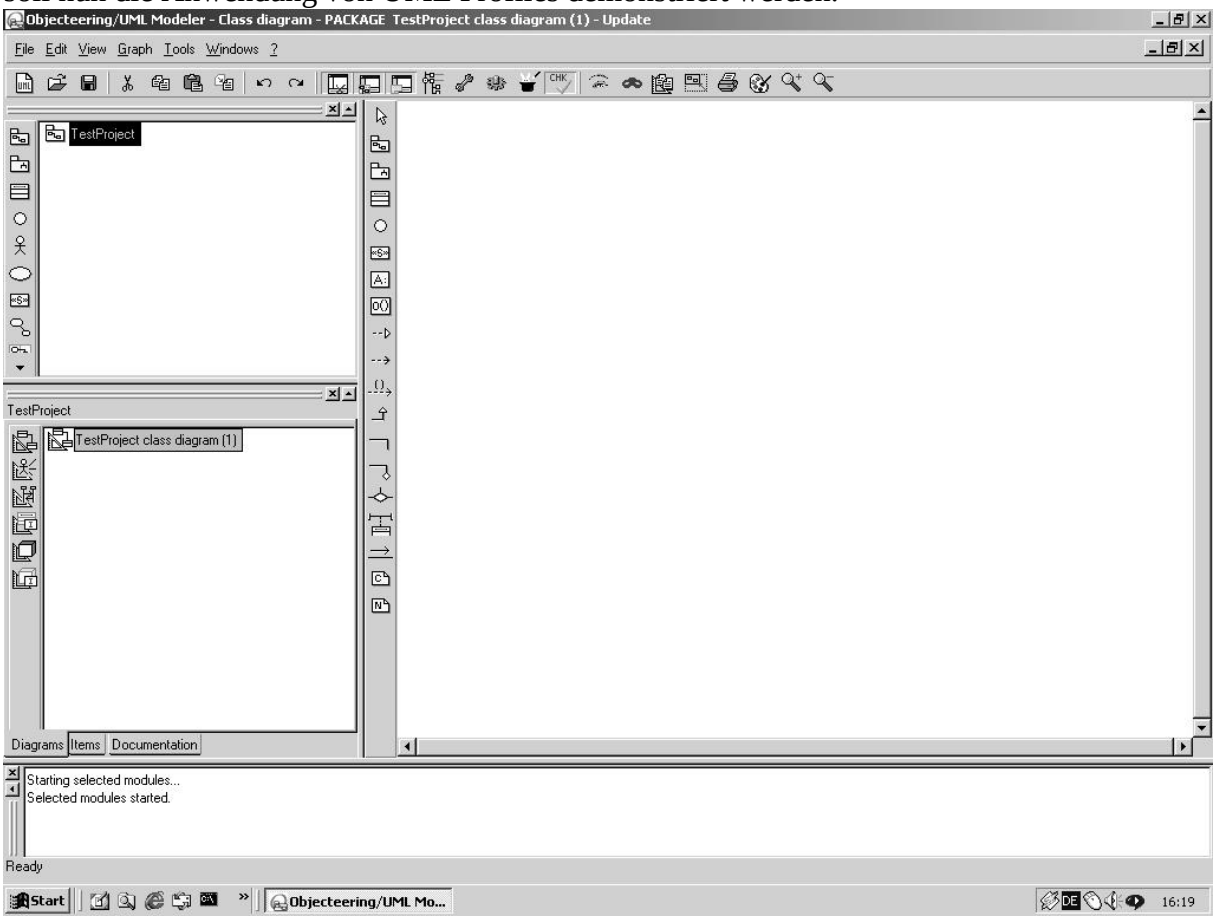

Abbildung 5-20: Der UML Modeler.

Abbildung 5-20 zeigt die Gesamtansicht auf den UML Modeler. Links oben befindet sich das Explorer-Fenster, das in einem Baum alle Modellelemente des aktuellen Projekts enthält. Darunter befindet sich das Eigenschaftsfenster, in dem zu einem markierten Modellelement verschiedene Eigenschaften und zugeordnete Objekte sichtbar sind. Das Hauptfenster enthält die erstellten Diagramme.

5.3.1 Module

Für jedes Projekt können Modules angegeben werden, die verwendet werden sollen. Dazu kann ein Dialog aufgerufen werden indem alle verfügbaren Modules, d.h. mitgelieferte und nachträglich erhaltene, zur Auswahl stehen. Als Beispiel wird, zusätzlich zu standardmäßig aktivierten Modules, das Module *MyModule* ausgewählt (Abbildung 5-21).

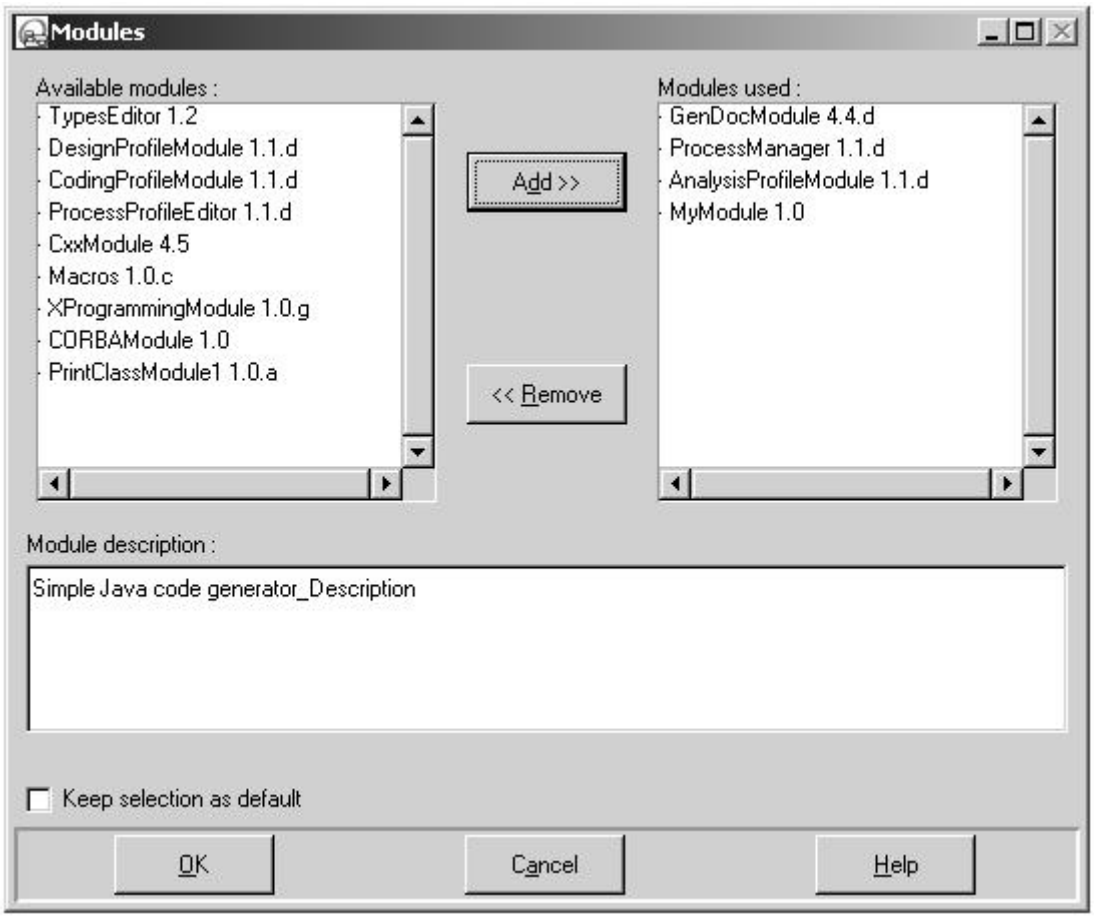

Abbildung 5-21: Auswahl von Modules für ein Projekt.

Über einen weiteren Dialog können alle ausgewählten Modules mit Hilfe deren Module Parameter (siehe 5.2.4.5) konfiguriert werden (Abbildung 5-22). Das Module *MyModule* wird angezeigt durch dessen Label (*Simple Java code generator V1.0*). Darunter befinden sich die einzelnen Parameter nach Gruppen geordnet. *Java generation path* legt einen Standardwert für den Pfad und *Java generation suffix* für die Dateiendung generierter Dateien fest. Unter der Gruppe *GroupExternalEdition* kann ein externer Editor angegeben werden.

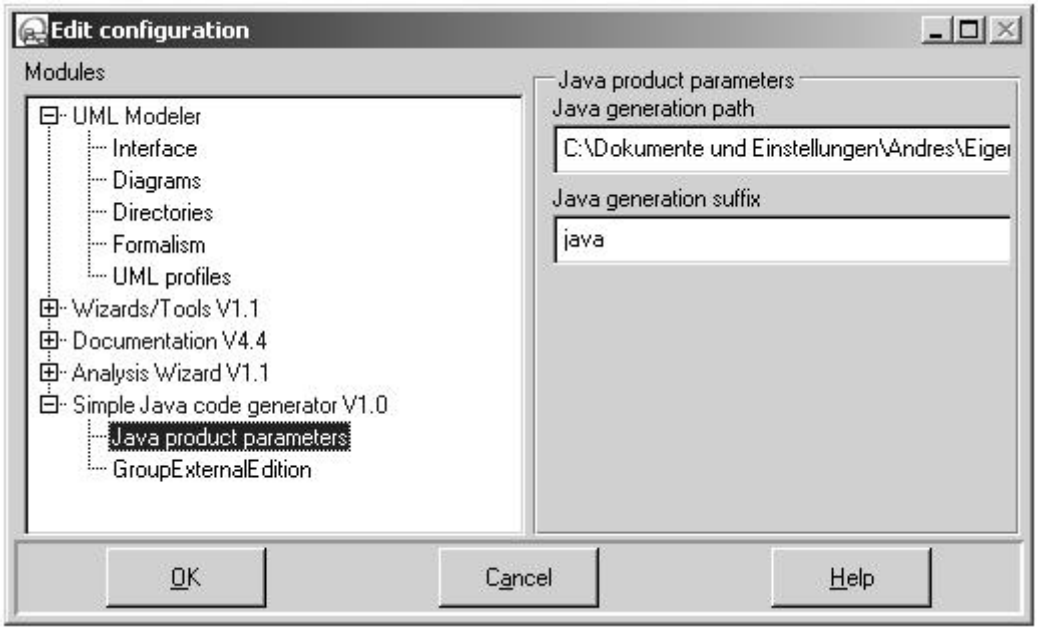

Abbildung 5-22: Konfiguration aktiver Modules durch deren Module Parameter.

Die gewählten Modules können nun während der Modellierung genutzt werden. Als Beispiel werden ein Package *Hotel* und drei Klassen, *Kunde*, *Zimmer* und *Reservierung* erstellt.

5.3.2 Stereotype

Alle Stereotypes aus den UML Profiles, die in den gewählten Modules enthalten sind, stehen bei der Modellierung zur Verfügung. Im Eigenschaftendialog zu jedem Modellelement kann ein Stereotype ausgewählt werden. Dabei ist ein Stereotype nur für die Modellelemente verfügbar, deren Metaklasse mit der Base Class des Stereotypes übereinstimmt oder eine Unterklasse deren ist. Jedes Modellelement kann nur mit maximal einem Stereotype versehen werden. Das Stereotype kann im Diagramm in der üblichen Notation angezeigt werden, d.h. entweder durch den Stereotype-Namen oder alternativ durch, falls vorhanden, das Icon des Stereotypes (siehe Beispiel in Abbildung 5-24).
Beispiel:
Als Beispiel wird eine Klasse *Kunde* im Eigenschaftsdialog mit einem Stereotype *persistent* versehen (Abbildung 5-23). Zur Auswahl stehen nur Stereotypes, die *Class* oder eine Oberklasse von *Class* als Base Class haben.
Im Diagramm kann nun das Stereotype durch ein Label oder durch sein Icon repräsentiert werden (Abbildung 5-24). Ist eine weniger detaillierte Ansicht gewünscht, so kann die Klasse auch ohne das Stereotype im Diagramm angezeigt werden.

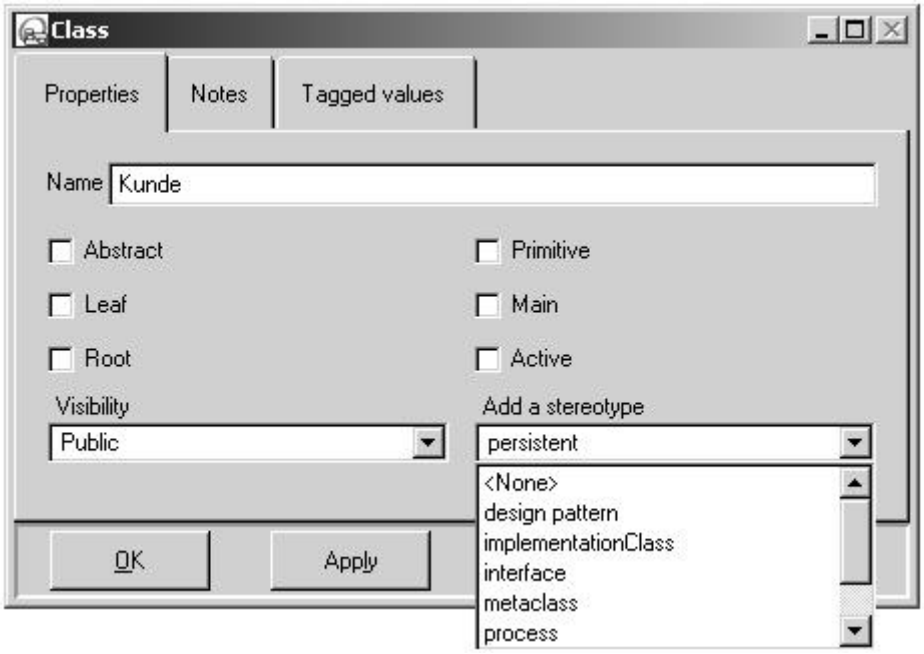

Abbildung 5-23: Im Eigenschaftsdialog der Klasse *Kunde* wird das Stereotype *persistent* ausgewählt.

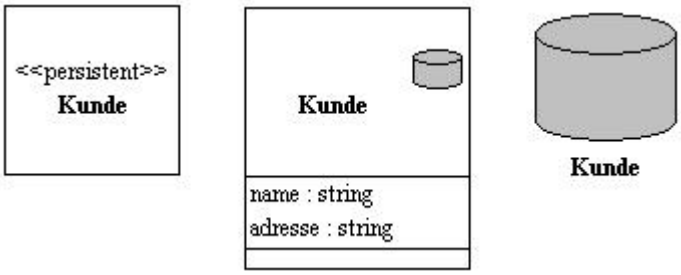

Abbildung 5-24: Alternative Darstellung eines Stereotypes.

Ist dem Stereotype ein Explorer-Icon zugeordnet, so werden Modellelemente, die mit dem Stereotype versehen sind, im Explorer durch dieses gekennzeichnet (Abbildung 5-25).

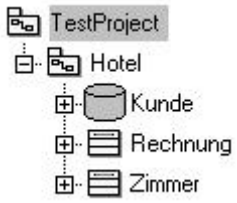

Abbildung 5-25: Kennzeichnung der Klasse *Kunde* durch das Explorer-Icon im Explorer.

5.3.3 Tagged Values

In der Karteikarte *Tagged Values* im Eigenschaftsdialog (siehe Abbildung 5-23) eines Modellelementes können diesem Tagged Values hinzugefügt werden. Dabei wird berücksichtigt, für welche Metaklassen oder Stereotypes der Tag definiert wurde.

Entsprechend der Tag Definition muss die definierte Anzahl von Parametern eingegeben werden, sowie gegebenenfalls ein Qualifier.

Beispiel:
In Abbildung 5-26 wird die für Klasse *Kunde*, die mit dem Stereotype *persistent* versehen ist, der zum Stereotype gehörende Tagged Value *tableName* ausgewählt und ihm der Wert *kunde* zugeordnet.

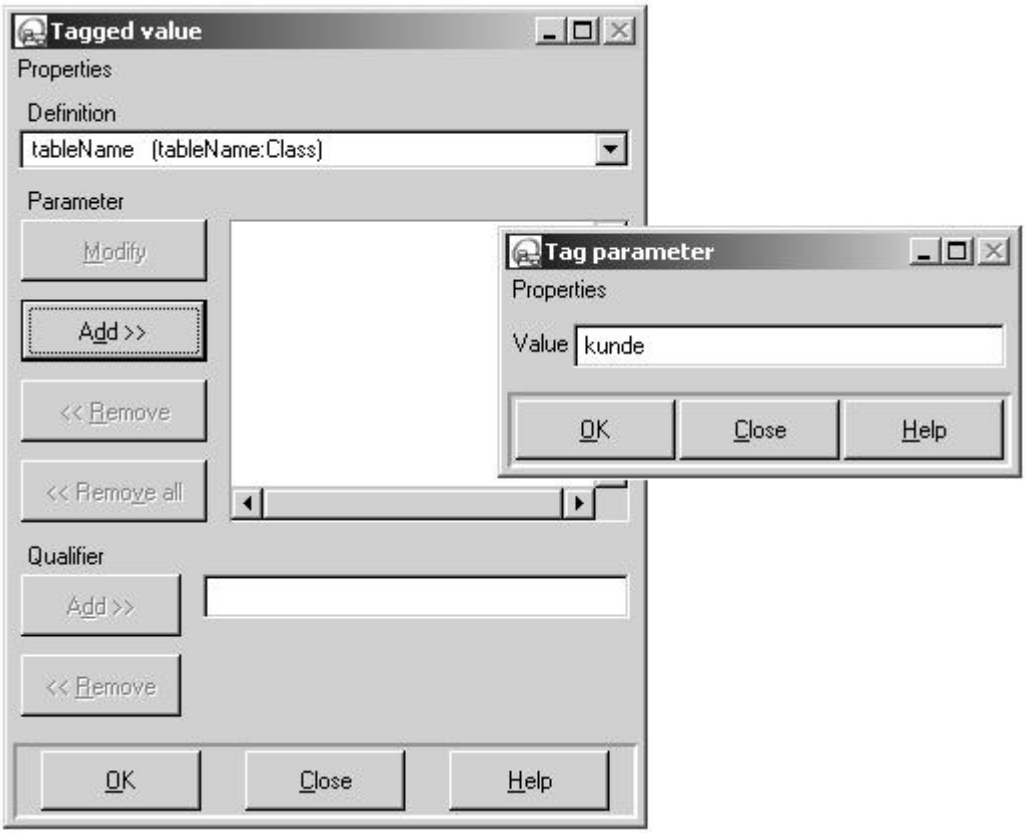

Abbildung 5-26: Hinzufügen eines Tagged Values *tableName* mit dem Wert *kunde*.

Abbildung 5-27: Die Klasse *Kunde* mit dem Tagged Value *tableName* mit dem Wert *kunde*.

5.3.4 Constraint

Wurden einem Stereotype Constraints hinzugefügt, so sollten diese für alle Modellelemente gelten, die mit dem Stereotype versehen sind. Im UML Modeler erfolgt aber keine semantische Auswertung von Constraints, noch sind sie in Diagrammen visualisierbar.

5.3.5 Note

Durch Notes kann einem Modellelement Text zugeordnet werden. Wurde im Profile ein Note Type einer Metaklasse zugeordnet, so kann eine Note dieses Typs Modellelementen der entsprechenden Metaklasse (oder einer Unterklasse) zugeordnet werden. Wurde der Note Type einem Stereotype zugeordnet, so werden Notes dieses Typs Modellelementen zugeordnet, die mit diesem Stereotype versehen sind. In Diagrammen kann die Note sichtbar gemacht werden, wobei die Notation für Kommentare in UML verwendet wird.

Beispiel:
In unserem Beispiel wurde ein Note Type *JavaCode* für die Metaklasse *Operation* definiert. Es kann daher beispielsweise in der Klasse *Rechnung* einer Methode *abschliessen()* Text in einer Note vom Typ *JavaCode* zugeordnet werden. In unserem Module wird dieser als Code der Methode interpretiert. Der Klasse *Kunde* die mit dem Stereotype *persistent* versehen wurde, wird eine Note des zum Stereotype gehörigen Note Types *SQLCode* hinzugefügt. Abbildung 5-28 zeigt die Darstellung der Notes im Diagramm.

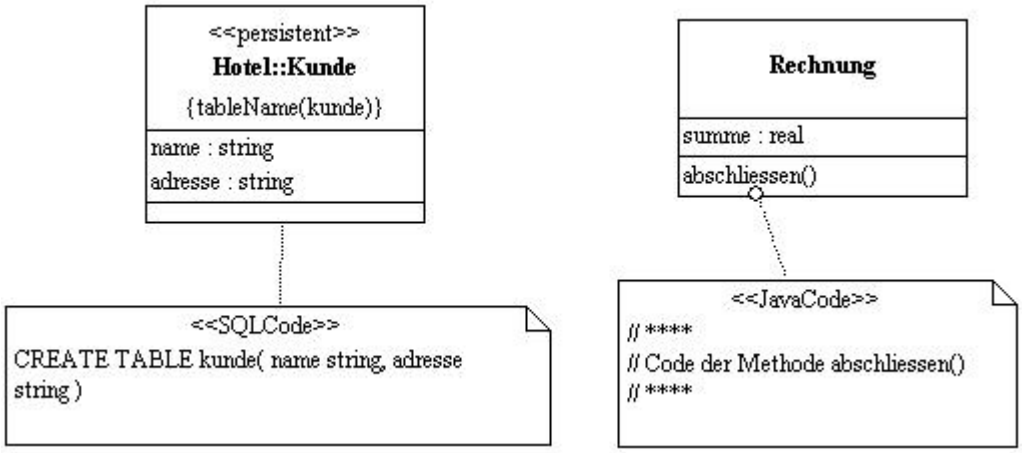

Abbildung 5-28: Darstellung der Notes *SQLCode* und *JavaCode* im Diagramm.

5.3.6 Command

Commands aus Modules erweitern die Kontextmenüs von Modellelementen. Sie rufen J Methods aus Profiles auf. Die Metaklasse, in deren Kontextmenü das Command verfügbar ist, entspricht der Metaklasse, der die aufzurufende J Method zugeordnet ist. In den Kontextmenüs ist als Menüpunkt das Label des Modules angezeigt, darunter befinden sich die einzelnen Commands als Untermenüpunkte.

Beispiel:
Im unserem Module mit dem Label *Simple Java code generator* wurden Commands für das Work Product (siehe dazu 5.3.7) definiert, die nun in dessen Kontextmenü verfügbar sind (Abbildung 5-29).

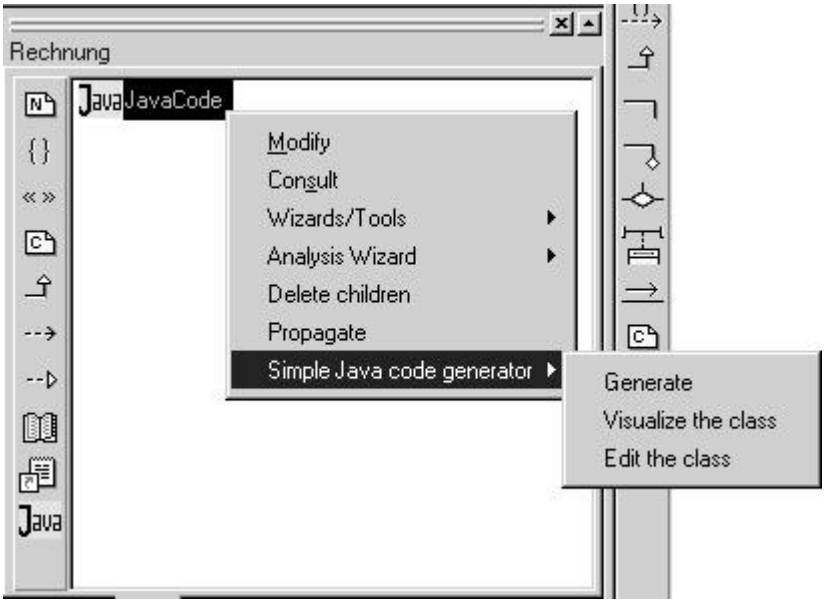

Abbildung 5-29: Im Kontextmenü des Work Products sind Commands verfügbar.

5.3.7 Work Product

Ein Work Product repräsentiert ein externes Objekt, typischerweise eine Datei mit generierter Dokumentation oder Code. Es ist bestimmten Metaklassen zugeordnet. Ist ein Modellelement der entsprechenden Metaklasse markiert, so erscheint in dessen Eigenschaftsfenster das „Creation Icon“, ein Symbol zur Erstellung eines Work Products eines bestimmten Typs. Erstellte Work Products werden auch im Explorer-Fenster angezeigt.

Beispiel:
Das in unserem Beispiel definierte Work Product *JavaProduct* kann für Modellelemente der Metaklassen *Package* oder *Class* erstellt werden. Dazu dient das Icon des Work Products, das im Eigenschaftsfenster angezeigt wird, falls das aktuell markierte Element ein Package oder eine Klasse ist. In Abbildung 5-30 wird einer Klasse *Rechnung* ein Work Product dieses Typs mit dem Namen *JavaCode* hinzugefügt. Der Dialog zur Erstellung enthält automatisch die Metaattribute des Work Products. Im Kontextmenü des Work Products sind, dem Label *Simple Java code generator* des Modules zugeordnet, Commands verfügbar, die im Module definiert wurden (siehe Abbildung 5-29). Das Command *Generate* ruft im Work Product die J Method zur Codegenerierung auf. Dabei wird auch Code in Notes vom Typ *JavaCode* mit einbezogen.
Durch die weiteren Commands des Beispieles kann mit *Visualize the class* ein interner Editor aufgerufen werden, bzw. mit *Edit the class* ein beliebiger externer. Der externe Editor kann durch einen Module Parameter festgelegt werden. Für den internen Editor wurden dem Code Markierungen hinzugefügt, die festlegen, welche Teile des Codes im Editor modifiziert werden können. Klickt man auf einen modifizierbaren Codeteil, z.B. den Methodenrumpf, so öffnet sich eine Dialogbox zu dessen Bearbeitung (Abbildung 5-31).

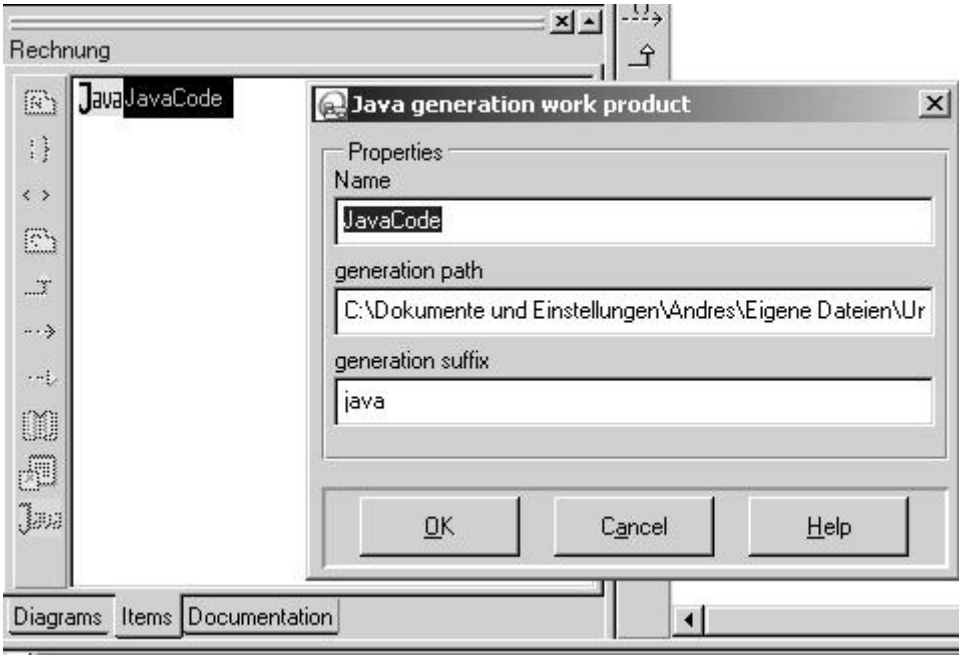

Abbildung 5-30: Erstellung eines Work Products im Eigenschaftsfenster.

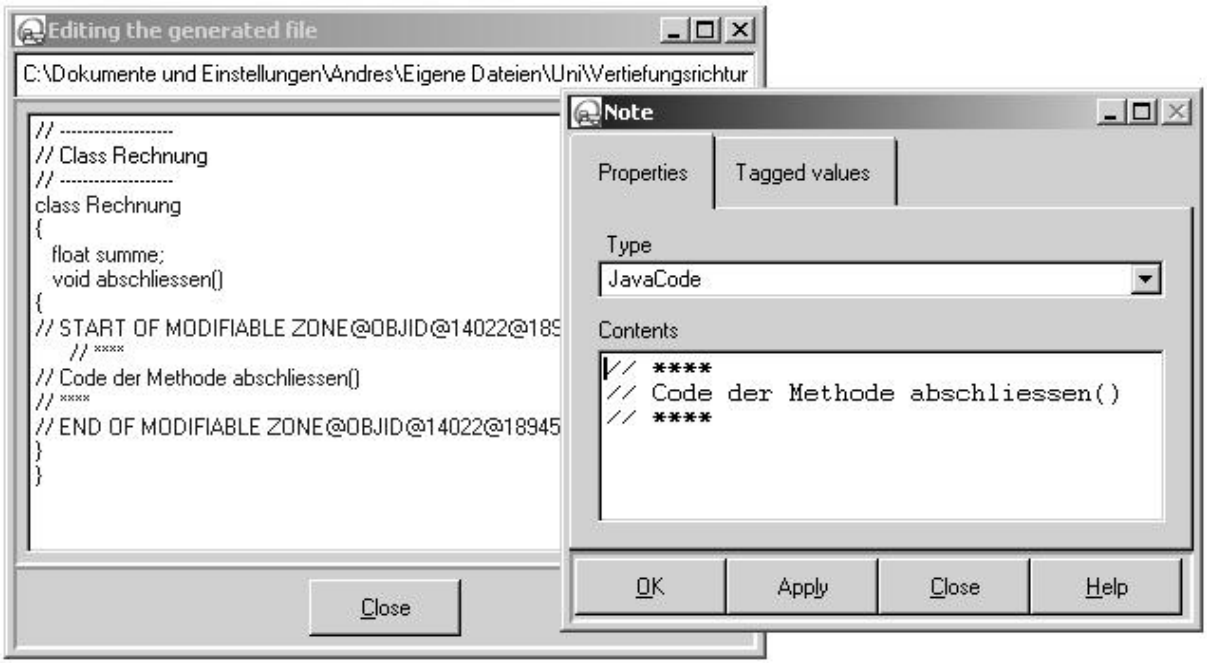

Abbildung 5-31: Ein Work Product kann mit Hilfe des internen Editors visualisiert und bearbeitet werden.

5.4 *Zusammenfassung und Bewertung*

Objecteering verwendet als Grundlage die UML Spezifikation 1.3 (siehe 4.1), was auch explizit von Softeam angegeben wird. Diese wird mit nur wenigen Einschränkungen umgesetzt. So ist bei Stereotypes keine Mehrfachvererbung möglich. Da gemäß UML 1.3 Modellelemente nicht mit mehreren Stereotypes versehen werden können, gibt es dadurch in Objecteering keine Möglichkeit, einem Modellelement die Semantik mehrerer existierender Stereotypes gleichzeitig zuzuweisen. Weiterhin können Stereotypes nicht als abstrakt definiert

werden. Sowohl Mehrfachvererbung als auch abstrakte Stereotypes werden z.B. im UML Profile für CORBA [OMG00b] verwendet. Somit ist dieses Profile in Objecteering nicht exakt umsetzbar. Bezüglich Tagged Values besteht die Einschränkung, dass diesen nicht, wie in UML 1.3 vorgesehen, ein Standardwert zugewiesen werden kann. Dies verursacht jedoch meist keine sehr großen praktischen Auswirkungen. Insgesamt werden Stereotypes, Tagged Values und Constraints gut unterstützt.
Die proprietären Bestandteile eines UML Profiles in Objecteering bieten die Möglichkeit, in flexibler und weitreichender Art eigene Semantik zu spezifizieren, wie auch das Werkzeug selbst anzupassen. Sie wurden in das intern verwendete Metamodell integriert. Dadurch präsentiert sich dem Anwender eine homogene Struktur und kann mit der Sprache J sowohl Semantik der proprietären, wie auch der Standardbestandteile beeinflusst werden. Bei der Spezifizierung eigener Semantik stehen hauptsächlich für die Generierung von Code und Dokumentation komfortable Hilfsmechanismen zur Verfügung.
Insgesamt können somit sowohl UML Profiles erstellt werden, die UML 1.3 entsprechen, als auch UML Profiles mit proprietären Bestandteilen und weitreichender Semantik. Der Schwerpunkt liegt dabei sicherlich auf Letzteren. Daher ist auch kein Format zum Austausch von UML Profiles mit anderen CASE-Tools vorgesehen.

6 Untersuchung des Werkzeuges „Together“ auf Erweiterbarkeit um Unterstützung für UML Profiles

In diesem Kapitel wird ein CASE-Werkzeug auf Erweiterbarkeit um Unterstützung von UML Profiles untersucht. Dazu wird zunächst dessen Auswahl begründet. Anschließend wird dessen Grundfunktionalität und bestehende Unterstützung von UML Erweiterungsmechanismen dargestellt, sowie die verfügbaren Mechanismen zur Erweiterung seiner Funktionalität. Letztere sollen schließlich auf eine konkrete Nutzung zur Realisierung einer Unterstützung von UML Profiles untersucht werden. Auf dieser Basis werden abschließend Empfehlungen gegeben.

6.1 *Auswahl des CASE-Werkzeuges*

In der engeren Auswahl (siehe auch Aufgabenstellung) standen zwei CASE-Werkzeuge, ArgoUML ([INET04]) und Together ([INET03]).
Bei Together gibt es als grundlegende Anwendung das Together Control Center, im Augenblick (Sommer 2001) in der Version 5.02. Es wird von der amerikanischen Firma Togethersoft hergestellt. Together Control Center wird erweitert durch „Building Blocks“, funktionale Einheiten, die mit Hilfe sogenannter Modules die Basisfunktionalität aktualisieren und erweitern. Togethersoft wirbt mit der hohen Anpassbarkeit Togethers durch Hinzufügen eigener Modules. Die Firma stellt dazu eine Programmierschnittstelle für die Sprache Java zur Verfügung, die „Open API“. Die Open API wird mit dem Together Control Center mitgeliefert, sowohl als Quellcode als auch in compilierter Form. Durch Javadoc-Dateien werden Package-, Klassen und Methoden beschrieben. Die Hilfe zu Together enthält grundlegende Erklärungen zur Nutzung der Open API. Schließlich stehen auch noch kleine Beispiele in Form eines Tutorials zur Open API zur Verfügung.
ArgoUML ist ein Projekt der Open Source-Gemeinschaft „Tigris.org“, demnach ist der gesamte Quellcode verfügbar. Im Gegensatz zu einer API sind damit Änderungen prinzipiell möglich, andererseits ist durch das Fehlen einer wohldefinierten Schnittstelle zur Erweiterung, wie z.B. einer API, meist eine Einarbeitung wesentlich schwieriger. Zum Zeitpunkt der Entscheidungsfindung war die Version 0.8 abgeschlossen und wurde an der Version 0.9 von ArgoUML gearbeitet. Aus den Webseiten ging hervor, dass Version 0.8 als überholt betrachtet und nicht für einen praktischen Einsatz empfohlen wurde. Da an Version 0.9 noch Änderungen vorgenommen wurden, hätte sie nicht ohne weiteres als gleichbleibende Basis für Erweiterungen herangezogen werden können. Auf Anfrage über die Mailing-Liste war zu erfahren, dass eventuell bereits von eigener Seite an einer Unterstützung von UML Profiles gearbeitet wird, genauere Informationen konnten jedoch nicht erhalten werden.
Da aus bezüglich ArgoUML genannte Nachteile und Unwägbarkeiten auftraten, sowie aufgrund der zeitlichen Beschränkung dieser Arbeit eine ausführlichere Voruntersuchung der Erweiterungsmöglichkeiten beider Werkzeuge nicht möglich war, wurde für Together entschieden.

6.2 *Grundfunktionalität*

Kernfunktionalität in Together ist graphische Modellierung sowie Bearbeitung und Verwaltung von parallel dazu erzeugtem Quellcode. Hier soll anhand des Grundaufbaus der Benutzeroberfläche ein Überblick über die grundsätzliche Funktionalität von Together gegeben werden. Abbildung 6-1 zeigt das Hauptfenster von Together. Neben einer Hauptmenüleiste, einer Werkzeugleiste und einer Statusleiste enthält es verschiedene Fenster unterschiedlicher Funktion, die im folgenden Abschnitt 6.2.2 vorgestellt werden. In Abschnitt 6.2.2 werden zwei Typen von Dialogen vorgestellt, die wichtig sind sowohl bezüglich grundlegender Arbeitsschritte in Together, wie auch im Hinblick auf eine Erweiterung des Werkzeugs.

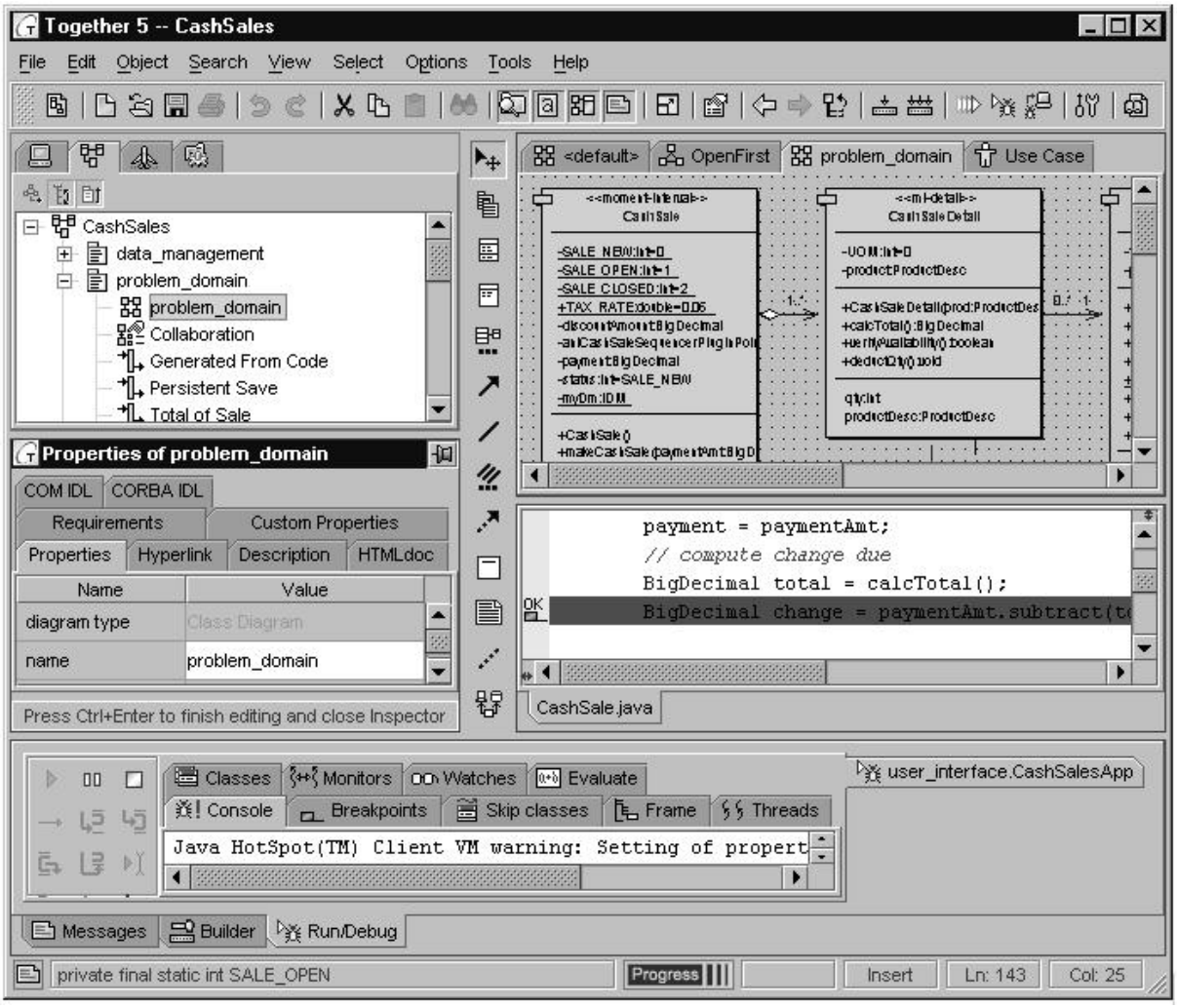

Abbildung 6-1: Das Hauptfenster von Together.

6.2.1 Verfügbare Fenster

Dem Anwender stehen im Hauptfenster bis zu vier weitere Fenster zur Verfügung, das Explorerfenster, das Diagramfenster, das Editorfenster und das Nachrichtenfenster.

Das Explorerfenster dient zur Darstellung des Projektinhalts, dem Zugang zu weiteren benötigten Dateien und zur Darstellung weiterer Informationen bezüglich der Umgebung von Together. Es enthält jeweils eine Karteikarte zur Darstellung

- des lokalen Dateisystems auf dem Rechner,
- einer logischen Sicht auf den gesamten Projektinhalt,
- aller Diagramme des aktuellen Projekts,
- eines graphischen Überblicks über das aktuell bearbeitete Diagramm,
- aller verfügbaren Modules (siehe 6.4.1) und
- verfügbarer Komponentenmodelle (sofern für das Projekt festgelegt wurde, dass Komponenten genutzt werden sollen).

Optional kann auch eine Karteikarte mit „Favoriten“, d.h. häufig benutzten Modellelementen, erstellt werden, um diese nicht vor jedem Zugriff in der Modellhierarchie suchen zu müssen. Abgesehen vom graphischen Überblick über das aktuell bearbeitete Diagramm sind alle Inhalte in einem Baum dargestellt. Über das Kontextmenü können die enthaltenen Elemente bearbeitet werden.
Das Diagrammfenster enthält Diagramme zur graphischen Modellierung der zu erstellenden Anwendung. Es kann Karteikarten für mehrere geöffnete Diagramme enthalten. Standardmäßig werden UML Diagramme und einige weitere, zum Teil Together-spezifische, Arten von Diagrammen unterstützt. Auf der linken Seite des Diagrammfensters befindet sich eine Werkzeugleiste. Diese enthält, je nach Art des aktuell sichtbaren Diagramms, Knöpfe zur Erstellung neuer Modellelemente, also in einem Klassendiagramm beispielsweise eine Klasse oder eine Assoziation. Zu allen Diagrammelementen sowie zum Diagramm selbst gibt es ein Kontextmenü mit Befehlen zur Bearbeitung und der Einstellung von Eigenschaften.
Das Editorfenster dient zur Darstellung textbasierter Dateien. Vorrangig sind dies Dateien mit Quellcode. Together erstellt während der Modellierung im Diagrammfenster zugehörigen Quellcode, so dass zu allen Änderungen im graphischen Modell direkt der entsprechende Quellcode im Editor eingesehen werden kann. Umgekehrt werden zumeist auch Änderungen des Quellcodes direkt in das Diagramm übernommen. Als Zielsprachen für die Codegenerierung stehen standardmäßig Java, C++ und CORBA IDL zur Verfügung. Sind im Editorfenster mehrere Dateien geöffnet, so befinden sich diese jeweils auf einer eigenen Karteikarte. Ein Kontextmenü zum Editor enthält Befehle zum Umgang mit den Dateien, z.B. zum Compilieren von Quellcodedateien.
Das Nachrichtenfenster enthält dynamisch ein oder mehrere Karteikarten. Stets enthält es eine Karteikarte zur Ausgabe von Nachrichten des Systems. Abhängig von den Operationen, die gerade in Together ausführt werden, enthält es weitere Karteikarten, z.B. Debug-Informationen, wenn der Together-interne Debugger gestartet wurde.

6.2.2 Grundlegende Dialoge

Zwei Typen von Dialogen zur Einstellung von Eigenschaften sind in Together für grundlegende Arbeiten wesentlich. Zum einen Konfigurationsdialoge zur Festlegung von Verhaltensweisen des Werkzeugs, zum anderen Eigenschaftsdialoge („Inspector“) zum Festlegen von Eigenschaften eines Modellelementes. Diese beiden Dialogarten sollen auch bei der Erweiterung von Together genutzt werden, und werden daher zum besseren Verständnis späterer Teile der Arbeit hier vorgestellt.
Durch Konfigurationsdialoge (Abbildung 6-2) kann das Verhalten von Together gesteuert werden. Sie können eine oder mehrere Gruppen von Eigenschaften enthalten. Jede Gruppe ist dabei auf einer eigenen Karteikarte enthalten, wobei innerhalb einer Karteikarte durch eine Baumstruktur eine weitere Gruppierung erfolgen kann. Der Dialog bezieht sich auf einen

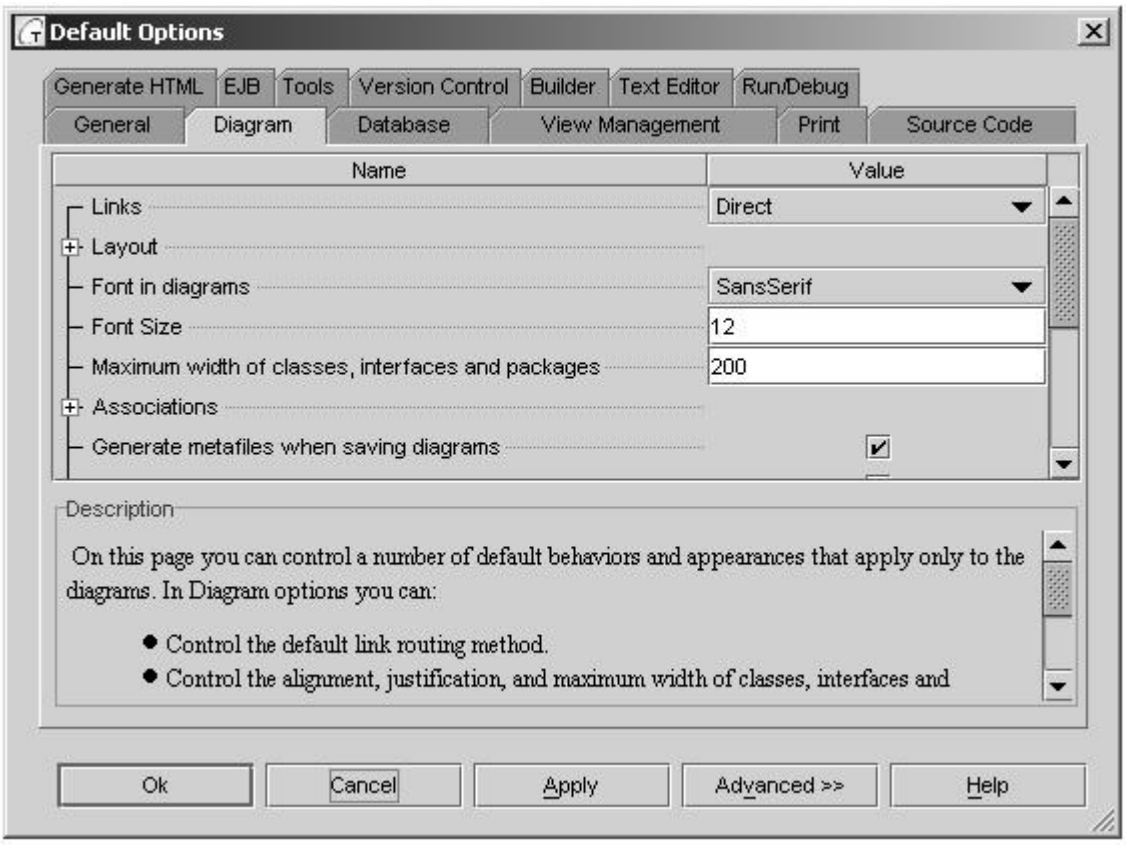

Abbildung 6-2: Konfigurationsdialog für alle standardmäßig verfügbaren Eigenschaften von Together.

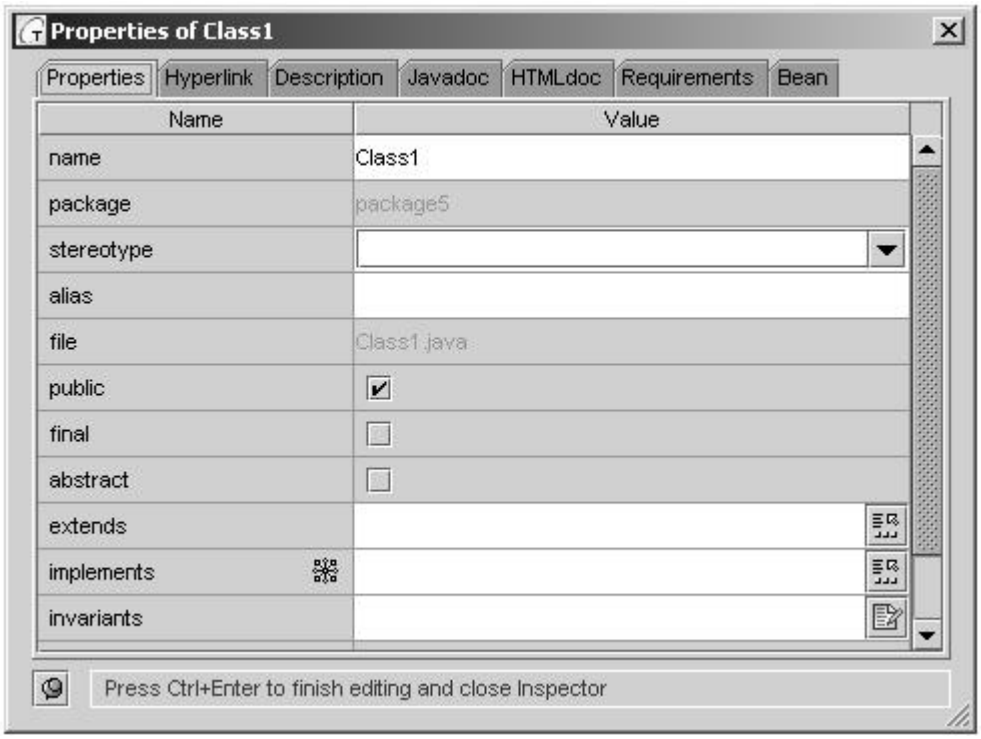

Abbildung 6-3: Inspector für eine Klasse.

bestimmten Gültigkeitsbereich der Einstellungen. Standardmäßig gibt es als Gültigkeitsbereich entweder das aktuelle Projekt oder alle Projekte. In der Hauptmenüleiste

von Together kann ein Konfigurationsdialog aufgerufen werden, der Karteikarten für alle standardmäßig verfügbaren Konfigurationseinstellungen enthält. Ansonsten werden Konfigurationsdialoge vom Kontextmenü eines konfigurierbaren Elements, z.B. dem Editorfenster, aus aufgerufen und enthalten nur dessen Eigenschaften.
Ein Inspector wird benötigt, um Einstellungen bezüglich eines Modellelementes zu treffen. Als Modellelement gelten hier alle vom Anwender erstellten Elemente, also auch z.B. Diagramme selbst, und nicht nur deren Inhalt. Abbildung 6-3 zeigt den Inspector für eine Klasse. Oft sind, ähnlich wie bei Konfigurationsdialogen, Eigenschaften auf verschiedenen Karteikarten gruppiert.

6.3 *Bestehende Unterstützung von UML Erweiterungsmechanismen*

Die bestehende Unterstützung für UML Erweiterungsmechanismen in Together ist gering.
Together bietet die Möglichkeit, einem Modellelement in dessen Inspector ein Stereotype zuzuweisen. Für einige Modellelemente, z.B. Klassen und Dependencies, sind standardmäßig Stereotypes enthalten, die im Inspector in einer Auswahlliste vorgeschlagen werden. Es kann entweder ein vorgegebenes Stereotype aus der Liste ausgewählt werden, oder ein beliebiger eigener Name eingegeben werden (Abbildung 6-4). Ein durch den Anwender eingeführter Name wird nicht der Auswahlliste hinzugefügt, d.h. er muss für jedes Modellelement neu eingegeben werden.

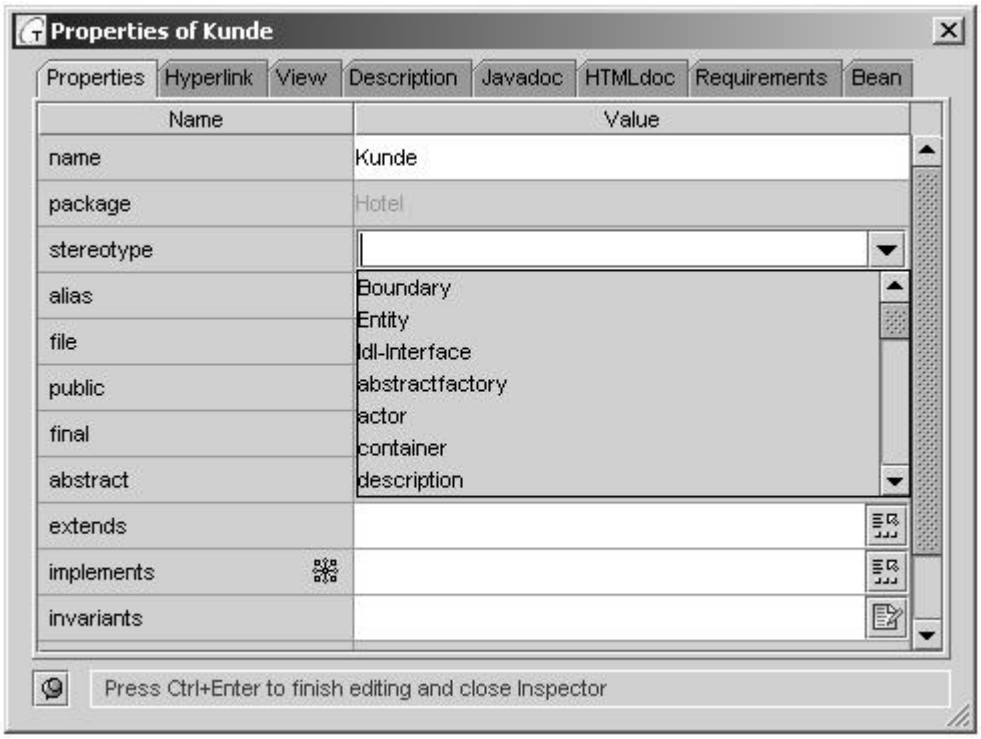

Abbildung 6-4: Der Inspector für eine Klasse enthält eine Auswahlliste von Stereotypes.

Stereotypes werden in Diagramme aufgenommen. Vom Anwender erstellte und die meisten der vordefinierten Stereotypes werden im Diagramm durch ein Label im jeweiligen Modellelement repräsentiert. Einige wenige der vordefinierten Stereotypes bringen jedoch eine eigene graphische Repräsentation mit sich, die das Label ergänzt. Je nach Typ und Inhalt

des mit dem Stereotype versehenen Modellelementes, wird dessen graphische Repräsentation zusätzlich beibehalten oder ganz durch die des Stereotypes ersetzt (siehe Abbildung 6-5). Die Art der Darstellung ist nicht durch den Anwender beeinflussbar.

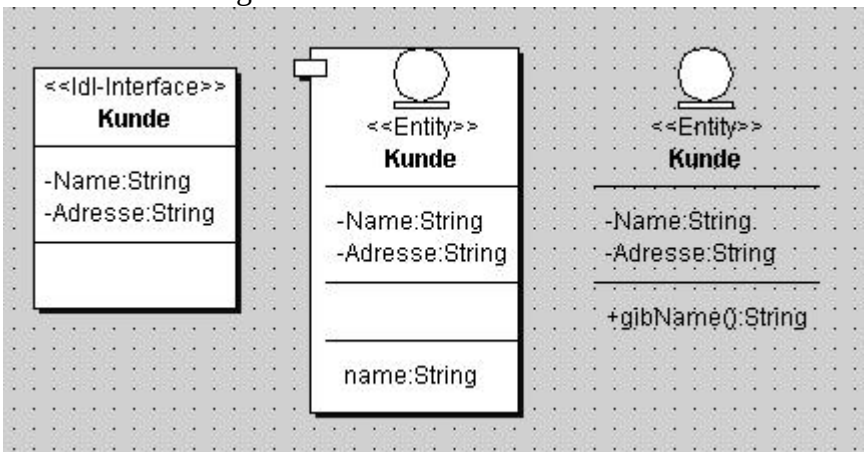

Abbildung 6-5: Darstellungen von Stereotypes im Diagramm.

Unabhängig davon, ob es sich um einen vordefinierten oder einen vom Anwender erstellten Stereotype handelt, wird der Stereotype im generierten Code als Javadoc-Kommentar eingefügt (Abbildung 6-6).

```
/* Generated by Together */

package Hotel;

/**
 * @stereotype Entity
 */
public class Kunde {
    private String Name;
    private String Adresse;
}
```

Abbildung 6-6: Aufnahme des Stereotypes „Entity“ in den generierten Code.

Eine weitergehende semantische Auswirkung von Stereotypes gibt es standardmäßig nicht. Auch werden Stereotypes beim Export eines Modells in XML nicht miteinbezogen.
Tagged Values sind in Form von Eigenschaften im Inspector zu einem Modellelement enthalten, z.B. ist zu Methoden ein Boolean-Wert *synchronized* im Inspector setzbar. Diese Eigenschaften werden semantisch ausgewertet, z.B. zur Generierung von Code. Standardmäßig können keine eigenen Tagged Values hinzugefügt werden. Together selbst verwendet den Begriff „Tagged Value“ nicht und unterscheidet auch nicht zwischen Eigenschaften, die als Tagged Values betrachtet werden können, und anderen Eigenschaften, die z.B. Metaattributen aus dem UML Metamodell entsprechen.
Eine weitere Unterstützung von UML Erweiterungsmechanismen ist in Together standardmäßig nicht gegeben.

6.4 *Die Open API*

Togethersoft bietet die Möglichkeit der Erweiterung und Anpassung von Together durch eine API (Application Programmer Interface), der „Open API“. Sie ermöglicht die Erstellung von Programmen, die mit Together interagieren und so dessen Grundfunktionalität erweitern können. Damit bildet sie die Basis für die angestrebte Erweiterung von Together um Unterstützung für UML Profiles. Dieses Kapitel führt in die Grundelemente und –prinzipien der Open API ein , und stellt Packages, Klassen und Interfaces vor, die für eine konkrete Erweiterung von Together um Profile-Unterstützung nötig sind.

6.4.1 Modules

Programme zur Erweiterung von Together werden „Modules“ genannt. Es gibt zwei Möglichkeiten der Entwicklung von Modulen:

- Java-Modules, die in Java geschrieben sind, mit einem Java-Compiler compiliert werden, und auf der selben JVM wie Together ausgeführt werden und
- TCL- oder JPython-Modules, die als Skripte von Together zur Laufzeit interpretiert werden.

6.4.1.1 Typen von Modules

Zusätzlich zur verwendeten Programmiersprache werden Modules nach dem Zeitpunkt ihres Aufrufes unterschieden. Typen von Modules sind hierbei

- „User“-Modules, die in Together durch ein Icon repräsentiert werden und zum gewünschten Zeitpunkt über ein Kontextmenü zu starten oder zu bearbeiten sind,
- „User“-Modules mit Initialisierung, die in Java geschrieben sein müssen und zusätzlich vor dem ersten Start eine Methode zur Initialisierung ausführen,
- „Startup“-Modules, die in Together nicht sichtbar sind und stets ab dem Start von Together ausgeführt werden und
- „Activatable“-Modules, die einen Mittelweg zwischen User- und Startup-Modules bilden, indem sie in einem Menü in Together aktiviert und deaktiviert werden können und auch über einen Neustart von Together hinaus aktiviert bzw. deaktiviert bleiben.

Ein Module kann auch die Eigenschaften mehrerer Typen kombinieren.
Je nach Typ muss ein Module (bzw. dessen Hauptklasse) ein oder mehrere Interfaces der API implementieren.
Es implementieren:

- User-Modules das Interface *IdeScript*, das die Methode *run()* enthält,
- Startup-Modules das Interface *IdeStartup* das die Methode *autorun()* enthält und
- Activatable-Modules das Interface *IdeActivatable*, das *IdeStartup* um eine Methode *shutdown()* erweitert.

Möglich ist auch, dass ein Startup- oder ein Activatable-Module zusätzlich *IdeScript* implementiert, um auch manuell aufzurufende Funktionalität bereitzustellen.

6.4.1.2 Anwendung in Together

Icons, die User- und Activatable-Modules repräsentieren, sind in Together im Explorer-Fenster unter der Karteikarte *Modules* enthalten (Abbildung 6-7). Dort sind sie in drei Ordner eingeteilt:

- *System* enthält Modules, die Teil von Together selbst sind,
- *Sample* enthält Beispiele für die Entwicklung von Modules und
- *EarlyAccess* enthält Modules, die noch nicht als ausreichend vollständig oder ausgereift betrachtet werden, um in *System* aufgenommen zu werden.

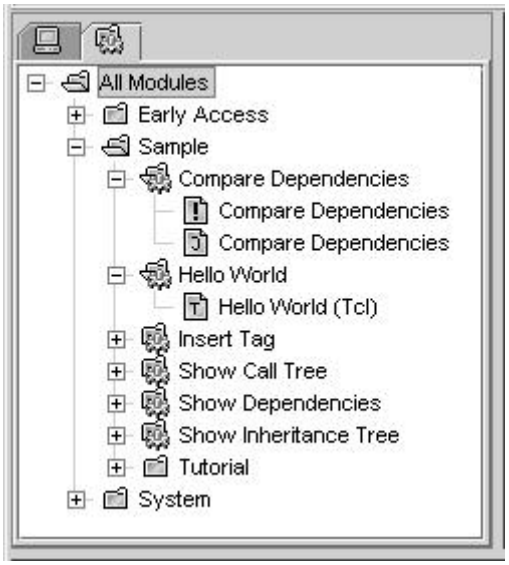

Abbildung 6-7: Baumstruktur mit Startup-Modules und Activatable-Modules.

Innerhalb eines Ordners sind die Namen der Modules aufgelistet (z.B. *Compare Dependencies* in Abbildung 6-7), darunter befinden sich Icons, die zugehörige Dateien symbolisieren. Dabei entspricht:

- einer Datei mit Java-Quellcode,
- einer compilierten Java-Datei und
- einem TCL-Skript.

Im Kontextmenü dieser Icons kann eine Datei gestartet oder bearbeitet werden, sowie, falls vom Module angeboten, eine Hilfe oder ein Konfigurationsdialog zum Module aufgerufen werden.
Activatable-Modules können zusätzlich im Hauptmenü von Together unter *Options→ActivatableModules* als aktiviert bzw. deaktiviert ausgewählt werden. Dieser Status bleibt während verschiedener Sitzungen mit Together persistent.

6.4.1.3 Manifest-Datei

Jedes Module muss ein eigenes Package unter *com.togethersoft.modules* anlegen und in einem entsprechenden Verzeichnis unter *%TogetherHome%/modules/com/togethersoft/modules/* gespeichert werden.
Ein Module benötigt eine Manifest-Datei, entweder mit der Dateiendung *.mf* in jar-Dateien oder mit der Dateiendung *.def* außerhalb einer jar-Datei. Sie ist eine Textdatei, die Informationen für Together über das Module und den gewünschten Umgang damit enthält.

Sie wird im Verzeichnis mit der Hauptklasse des Modules gespeichert und muss mit dem selben Namen wie die Hauptklasse benannt werden. Ihre Einträge haben die Syntax *Eigenschaftsname: Wert* (*.mf*-Datei), bzw. *Eigenschaftsname = Wert* (*.def*-Datei).

Folgende Eigenschaften können in der Manifest-Datei spezifiziert werden:

Obligatorische Eigenschaften:

- *Main-Class*:
  Name der Hauptklasse des Modules (nur in *.mf*-Dateien).
  In *.def*-Dateien heißt die Eigenschaft *MainClassName* und ist nicht obligatorisch.
- *Time*:
  Zeitpunkt des Aufrufes des Modules, d.h. *User*, *Startup* oder *OnDemand*. Letzteres kennzeichnet ein Module als Activatable-Module.

Optionale Eigenschaften:

- *Name*:
  Name des Modules, wie er in Together angezeigt werden soll.
- *Folder*:
  Name des Ordners, unter dem das Module im Explorer in Together angezeigt werden soll (z.B. *System/MyModule*).
- *ActivatedByDefault*:
  Nur für Activatable-Modules. Das Vorhandensein dieser Eigenschaft legt fest, dass das Module standardmäßig aktiviert sein soll.
- *Services*:
  Spezifiziert Namen für Dienste, die das Module für andere Modules bereitstellt.
- *DependsOn*:
  Spezifiziert Dienste aus anderen Modules, die verwendet werden sollen. Beim Laden des Modules wird geprüft, ob diese Dienste durch andere Modules (in der Eigenschaft *service*) auch zur Verfügung stehen, andernfalls kann das Module nicht verwendet werden.
- *Class-Path*:
  Pfad zusätzlich benötigter Klassen oder jar-Archive.
  In *.def*-Dateien: *ClassPath*.
- *HelpFile*:
  Vollständiger Pfad einer Hilfedatei. Im Kontextmenü des Modules im Explorer von Together ist die Hilfe dann aufrufbar.
- *Options*:
  Ermöglicht im Kontextmenü des Modules im Explorer von Together den Aufruf eines Konfigurationsdialogs (siehe 6.2.2). Anzugeben ist der Name der anzuzeigenden Karteikarte.
- *Hidden*:
  Hat den Wert *true* oder *false*. Legt fest, ob Module im Explorerfenster von Together angezeigt werden soll.

6.4.2 Config-Dateien

Together verwendet eine Vielzahl von Config-Dateien. Dabei handelt es sich um Textdateien zur Konfiguration mit der Dateiendung *.config*. Sie enthalten Einstellungen zu Aussehen und Verhalten von Elementen in Together. Da diese Einstellungen in den Config-Dateien änderbar sind, bzw. eigene Config-Dateien erstellt werden können, sind diese auch ein Instrument zur Anpassung bzw. Erweiterung von Together. Sie sind deshalb wichtig, weil einige Einstellungen in Config-Dateien zumindest in der aktuellen Version nicht allein über die Open API zugreifbar oder beeinflussbar sind.
Alle Config-Dateien werden beim Start von Together geladen und in Together in der Reihenfolge interpretiert, die sich durch eine alphabetische Sortierung der Dateinamen ergibt. Daher besteht die Konvention, dass Anwender dem Namen selbst erstellter Config-Dateien ein z_ voranstellen, damit Erweiterungen zuletzt ausgeführt werden und nicht von bereits vorhandenen Config-Dateien überschrieben werden.
Config-Dateien enthalten hauptsächlich Zuweisungen von Werten zu Eigenschaften in der Form *Eigenschaft = Wert*. Weiterhin sind auch Methodenaufrufe möglich, sowie die Verwendung einer *if*-Kontrollstruktur mit Syntax wie in Java. Das Zeichen # zu Zeilenanfang markiert die Zeile als Kommentar, das Zeichen / markiert, dass die Zeile die Fortsetzung der vorangegangenen Zeile bildet. Enthält eine Zeile *;*, so werden alle nachfolgenden Zeichen bis zum Zeilenende als Kommentar gedeutet.
Config-Dateien sind nur sehr wenig dokumentiert. Es steht ein Dokument „Config files HOWTO v0.6“ [Play01] zur Verfügung, das außer den hier beschriebenen allgemeinen Regeln auch ein Beispiel zur Erweiterung von Together um einen neuen Diagrammtyp mittels einer Config-Datei enthält. Ansonsten wird darauf verwiesen, sich die bei Together enthaltenen (weitgehend unkommentierten) Config-Dateien als Beispiele anzusehen. Daher sind einem Anwender nur die vordefinierten Eigenschaften und Werte bekannt, die in einer solchen Config-Dateien verwendet werden. Somit sind keine allgemeinen Aussagen über die Anwendung von Kontrollstrukturen oder Methoden möglich. Auch Eigenschaften und Werte, die nicht in einer der vorhandenen Config-Dateien gefunden werden können, bleiben dem Anwender, sofern sie nicht erschlossen oder erraten werden können, verborgen. Aus diesen Gründen sind im Rahmen dieser Arbeit keine weiteren allgemeinen Aussagen zu Config-Dateien möglich.

6.4.3 Grundaufbau der Open API

Die Open API zu Together 5.02 besteht insgesamt aus 40 Packages, die sich im Package *com.togethersoft.openapi* befinden. Sie sind in drei Gruppen eingeteilt,

- dem Package *com.togethersoft.openapi.ide* und dessen Unter-Packages,
- dem Package *com.togethersoft.openapi.rwi* und dessen Unter-Packages und
- dem Package *com.togethersoft.openapi.sci* und dessen Unter-Packages.

Diese drei Gruppen bilden zusammen ein dreischichtiges Interface zum Zugriff auf die grundlegende Infrastruktur von Together (Abbildung 6-8). Dabei bietet die oberste Schicht den am stärksten eingeschränkten und reglementierten Zugriff und die unterste Schicht den am wenigsten limitierten.
Die Schicht „IDE“ entspricht dem Package *com.togethersoft.openapi.ide* und dessen Unter-Packages. Sie ermöglicht die Anpassung der Ausgabe von Informationen aus Together und den enthaltenen Modellen. Dabei enthält sie Funktionalität sowohl bezüglich der Darstellung, als auch bezüglich der Interaktion mit dem Anwender in Together. Informationen aus dem

Modell können dabei nicht geändert, sondern nur gelesen werden. Die IDE ist durch Packages in verschiedene Bereiche gegliedert, wie z.B. Dialoge zur Anzeige der Eigenschaften von Modellelementen oder die Konfiguration von Together selbst.
Die Schicht „RWI“ ermöglicht weitergehende Eingriffe in Together, indem sie sowohl Lese- als auch Schreibzugriff auf Eigenschaften von Modellelementen bietet. Es können den Modellelementen auch weitere Eigenschaften hinzugefügt werden.
Die Schicht „SCI“ bietet uneingeschränkten Zugriff auf den Quellcode, der zu einem Modell gehört. Dabei wird, soweit möglich, von der konkret zu erzeugenden Programmiersprache abstrahiert.

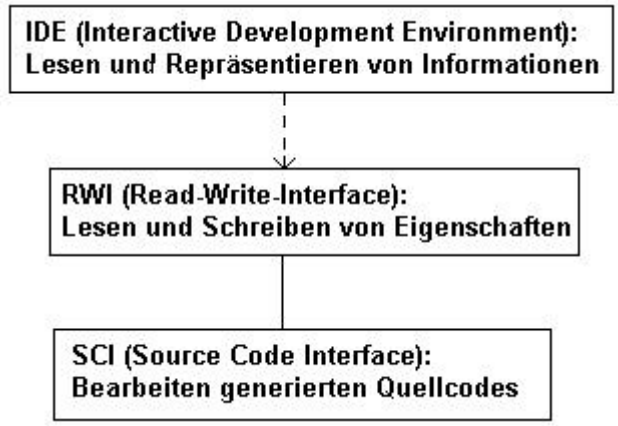

Abbildung 6-8: Grundaufbau der Open API.

RWI repräsentiert eine Sicht auf das logische Modell, das von einem Anwender in Together erstellt wurde. Durch IDE kann beeinflusst werden, wie der Modellierer sein logisches Modell erzeugen und bearbeiten kann. SCI konzentriert sich auf die Darstellung des logischen Modells als Quellcode.

6.4.4 Zugriff auf Objekte

Um aus einem Module heraus auf Objekte in Together zugreifen zu können, enthalten die meisten Packages der Open API eine zentrale Verwaltungsklasse mit dem Namen PackageName*Manager*. Zu dieser gehört eine Klasse PackageName*ManagerAccess*, welche eine statische Methode *get*PackageName*Manager()* enthält, durch die aus jedem Kontext heraus auf die Verwaltungsklasse zugegriffen werden kann. alle Verwaltungsklassen im Package *ide* sind alternativ auch über statische Methoden in der Klasse *IdeManager* erhältlich. Die Verwaltungsklassen bieten Methoden an, um auf Objekte in Together zuzugreifen und diese zu modifizieren, oder um eigene Objekte in Together zu registrieren.

6.4.5 Modellelemente und Eigenschaften der RWI-Ebene

Für den Umgang mit Eigenschaften von Modellelementen ist die Schicht RWI zu nutzen. Im folgenden sollen in deren Grundaufbau eingeführt werden. Sowohl in diesem als auch in den folgenden Kapiteln wird versucht, die Darstellung auf das Wesentliche zu reduzieren und Details, die zum grundlegenden Verständnis unnötig erscheinen, beiseite zu lassen. Daher

wurden in einigen Fällen Vereinfachungen vorgenommen, die nicht exakt den Gegebenheiten in der Open API entsprechen. Dies gilt auch für die Diagramme, die die logische Struktur innerhalb einiger Packages darstellen. Oft handelt es sich bei den zu verwendenden Elementen aus der Open API um Interfaces, deren interne Implementierung als Rückgabewert von Methoden erhältlich ist. In Diagrammen wird zur Übersichtlichkeit darauf verzichtet, die interne Implementierung dieser Interfaces darzustellen und werden zur Darstellung des Prinzips Assoziationen und Attribute dem Interface direkt zugeordnet. Ein Interface kann keine Assoziationen oder Attribute haben, jedoch zur Erläuterung der Prinzipien erscheint hier eine solche Darstellung sinnvoll.
In Together sind Eigenschaften („Properties“) die Grundbestandteile des logischen Modells und dessen Elementen. Die einzelnen Elemente und deren Eigenschaften werden durch Klassen und Interfaces im Package *com.togethersoft.openapi.rwi* repräsentiert und stehen dadurch dem Nutzer der Open API zur Verfügung.
Das logische besteht aus Modellelementen, die „RWI-Elemente“ genannt werden und durch „RWI“ am Namensanfang gekennzeichnet sind. Verbunden sind RWI-Elemente durch sogenannte „RWI-Referenzen“. Ein RWI-Element besteht aus RWI-Eigenschaften. Die RWI-Eigenschaften eines RWI-Elementes werden in einer „PropertyMap“ zusammengefasst.
Abbildung 6-9 zeigt die elementaren Interfaces des Package *rwi*.

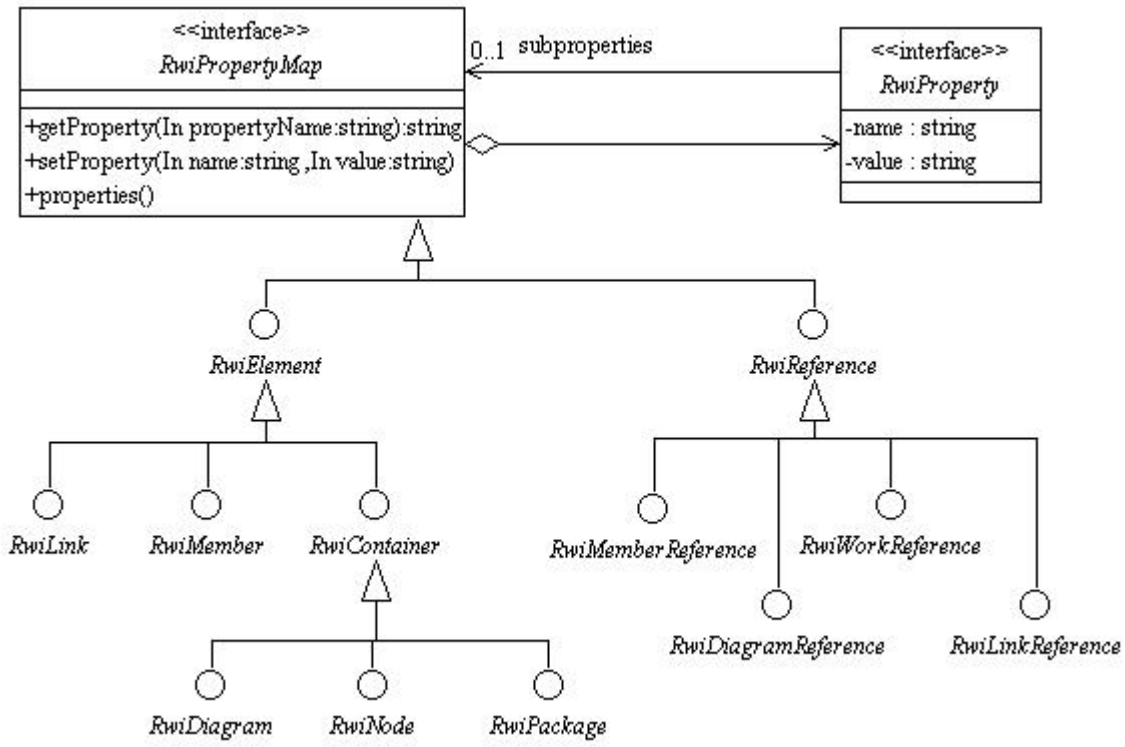

Abbildung 6-9: Package *rwi* - RWI-Elemente bestehen aus RWI-Properties (Prinzipdarstellung).

Interface RwiPropertyMap:
Eine *RwiPropertyMap* ist ein Container für Eigenschaften. Diese bestehen aus einem Namen und einem Wert. Der Wert kann vom Typ Boolean oder vom Typ String sein. Im letzteren Fall ist die Eigenschaft eine Instanz von *RwiProperty*. Für Eigenschaften und ihre Werte enthält *RwiPropertyMap* Methoden zum Hinzufügen und zum Zugriff. Im Fall einer Eigenschaft vom Typ Boolean entspricht ihr Vorhandensein dem Wert *true* und das Nichtvorhandensein dem Wert *false*.
Zusätzlich zum Typ ist noch zwischen „festkodierten“ und benutzerdefinierten Eigenschaften zu unterscheiden. Festkodierte Eigenschaften sind vorgegeben und werden von Together in

einer vordefinierten Weise interpretiert und behandelt und werden automatisch den entsprechenden Elementen zugewiesen. Die Namen festkodierter Eigenschaften sind als Konstanten in *RwiProperty* spezifiziert. Benutzerdefinierte Eigenschaften können beliebig hinzugefügt werden. Sie werden wieder entfernt, falls ihr Wert *null* im Falle eines Strings bzw. *false* im Falle von Boolean ist.
Eigenschaften, die sich auf generierten Quellcode beziehen, werden teilweise auch im Quellcode selbst gespeichert.

RwiProperty:
Eine RWIProperty repräsentiert eine Eigenschaft des logischen Modells mit einem Wert vom Typ String. Die Klasse enthält als Konstanten die Namen von „festkodierten" („hardcoded") Eigenschaften.
Von einigen dieser festkodierten Eigenschaften kann es in einem RWI-Element mehrere Instanzen geben. Dies trifft z.B. auf *RwiProperty.PARAMETER* zu, da ein Element im Modell mehrere Parameter haben kann. Dagegen kann es von *RwiProperty.NAME* nur eine Instanz pro Element geben.
Einer *RwiProperty* können weitere Eigenschaften untergeordnet sein. Dann enthält sie als Wert eine *RwiPropertyMap*, welche wiederum die untergeordneten Eigenschaften enthält. Notwendig ist dies z.B. bei *RwiProperty.PARAMETER*, da von einem Parameter sowohl der Name als auch der Typ gespeichert werden soll. Zumindest eine der beiden Eigenschaften muss dann als untergeordnete Eigenschaft gespeichert werden, da *RwiProperty* immer nur einen Wert direkt enthalten kann.
Eine *RwiProperty* kann noch weitere Eigenschaften haben, z.B. kann sie nur lesbar sein. Für alle ihre Eigenschaften, d.h. ihren Wert, etwaige untergeordnete Eigenschaften usw. enthält das Interface Methoden zum Auslesen.

6.4.6 Inspector und Eigenschaften der IDE-Ebene

Das Package *ide* enthält Elemente bezüglich der graphischen Repräsentation für den Anwender. Es enthält das Packages *ide.inspector* und dessen Unter-Packages. Durch diese kann ein Inspector angepasst werde. Bei einem Inspector handelt es sich um einen Dialog, in dem Eigenschaften eines Modellelementes aufgelistet sind und dort vom Anwender bearbeitet werden können (siehe 6.2.2).

6.4.6.1 Grundaufbau

Die Eigenschaften in einem Inspector unterscheiden sich von den Eigenschaften aus dem Package *rwi* (siehe 6.4.5) insofern, dass sie speziell für den Anwender sichtbar und modifizierbar sein sollen. Zur Unterscheidung kann von IDE- bzw. RWI-Eigenschaften gesprochen werden. So kann z.B. ein Modellelement interne RWI-Eigenschaften enthalten, die nicht in einem Inspector sichtbar sind. Andererseits kann ein Inspector zusätzliche IDE-Eigenschaften enthalten, die aus einer Aufbereitung zur Repräsentation für den Anwender resultieren und die keine direkte Entsprechung in der internen Repräsentation des Modellelementes haben. Weiterhin können IDE-Eigenschaften, die über den Inspector festgelegt werden, auch direkt im erzeugten Code (z.B. als generierter Javadoc-Kommentar) gespeichert werden, ohne dass sie intern dem Modellelement als RWI-Eigenschaft (siehe 6.4.5) hinzugefügt werden.

Ein Inspector setzt sich zusammen aus „Components“. Diese repräsentieren eine bestimmte Gruppe von Eigenschaften, z.B. allgemeine Eigenschaften, Eigenschaften bezüglich Dokumentation oder Eigenschaften, die durch ein Komponentenmodell vorgegeben sind. Graphisch wird eine Component zumeist durch eine Karteikarte im Inspector-Fenster dargestellt. Components enthalten die einzelnen Eigenschaften des Modellelements, die für den Nutzer sichtbar sein sollen. Sowohl von Components als auch von Eigenschaften gibt es verschiedene Arten. Abbildung 6-10 zeigt diesen grundsätzlichen Aufbau eines Inspectors.

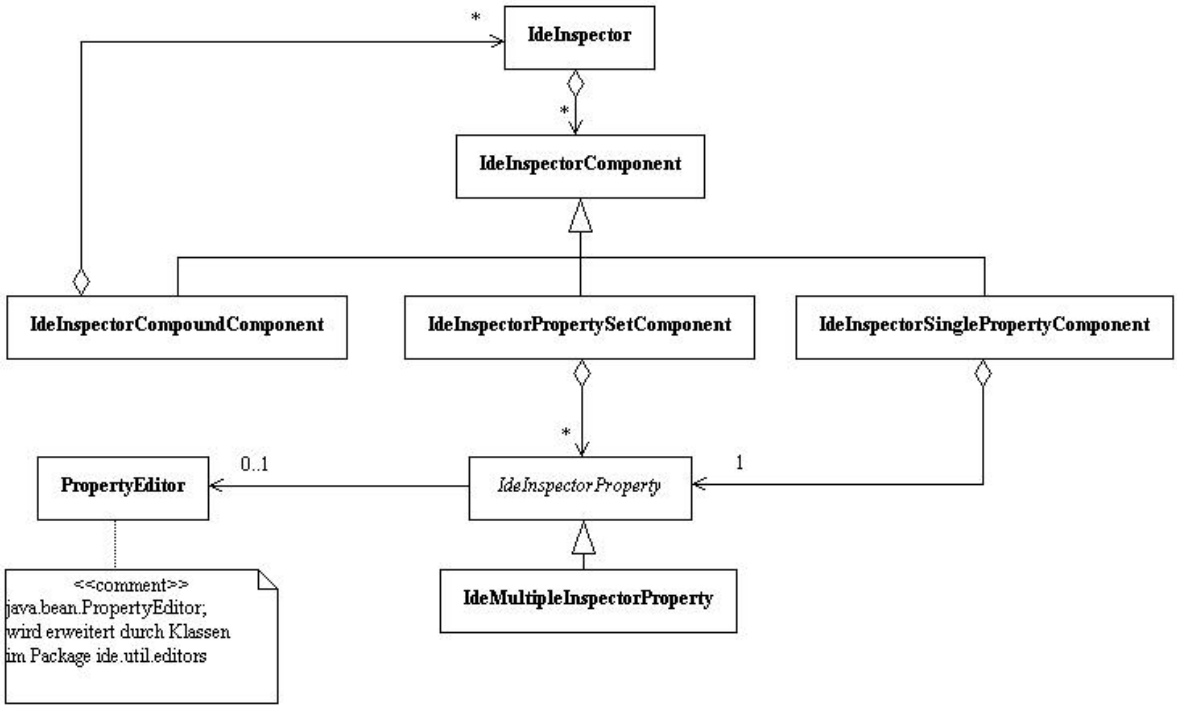

Abbildung 6-10: Package *ide.inspector* – Aufbau eines Inspectors (Prinzipdarstellung).

Die eigentliche graphische Repräsentation wird durch zusätzliche Klassen erledigt. Im Fall von Inspector und Component sind dies sogenannte Services. Als Service kann eine beliebige Klasse registriert werden. Für einen Inspector ist als Service die Klasse *ide.inspector.util.TabbedPaneInspectorUI* vordefiniert, welche einen Inspector als ein Fenster darstellt und jede enthaltene Component durch eine Karteikarte repräsentiert. Für eine Component ist die Klasse *ide.inspector.util.table.PropertyTableComponentUI* als Service vorgegeben. Sie realisiert die Anordnung der einzelnen Eigenschaften in einer Tabelle. Im Fall einer Eigenschaft in einem Inspector, d.h. einer IDE-Eigenschaft, wird die Visualisierung durch einen Editor realisiert. Dieser muss *java.bean.PropertyEditor* spezialisieren (siehe Beschreibung zu dieser Klasse).

IdeInspector:
Diese Klasse repräsentiert einen Inspector und spezialisiert *java.beans.FeatureDescriptor*. Sie enthält Methoden zum Hinzufügen von Components und zum Zugriff auf diese. Components können, abhängig von einer Bedingung, sichtbar oder unsichtbar sein. Zur Evaluation dieser Bedingung wird der Component eine Instanz des Interface *ide.Condition* hinzugefügt werden, dessen Methode *execute()* einen Boolean-Wert zurückliefert. Weiterhin kann der Component ein Double-Wert *weight* hinzugefügt werden, durch den die Reihenfolge der Components innerhalb des Inspectors spezifiziert wird. Wird z.B. jede Component im Inspector durch eine Karteikarte repräsentiert, so enthält die erste Karteikarte die Component mit dem niedrigsten Wert *weight* und die letzte Karteikarte die Component mit dem höchsten.

Außer Methoden zum Umgang mit diesen Elementen enthält Inspector noch getter- und setter-Methoden für seinen Service, sowie zum Registrieren eines *java.beans.PropertyChangeListener*.

IdeInspectorComponent:
Diese Klasse repräsentiert eine Component und spezialisiert *java.beans.FeatureDescriptor*. Sie enthält getter- und setter-Methoden für einen Service. Zusätzlich zu den Klassen in Abbildung 6-10 wird sie noch von spezifischeren Klassen spezialisiert, z.B. von der Klasse *ide.inspector.util.DescriptionComponent*, die eine Component repräsentiert, in die eine Beschreibung eines Modellelements eingegeben werden kann.

IdeInspectorCompoundComponent:
Diese Klasse spezialisiert *IdeInspectorComponent*. Sie repräsentiert eine komplexe Component, die aus weiteren Inspectors zusammengesetzt ist. Bezüglich enthaltener Inspectors enthält sie getter- und setter-Methoden. Spezialisiert wird sie durch *ide.inspector.util.TabbedCompoundComponent*.

IdeInspectorPropertySetComponent:
Diese Klasse spezialisiert *IdeInspectorComponent*. Sie enthält eine Liste von Properties. Die Behandlung von Properties erfolgt analog der Bahandlung von Components in *IdeInspector*, d.h. ebenfalls unter Einbeziehung einer Bedingung bezüglich der Sichbarkeit und eines Wertes *weight*.

IdeInspectorSinglePropertyComponent:
Diese Klasse spezialisiert *IdeInspectorComponent*. Sie repräsentiert eine Component, die eine einzige IDE-Eigenschaft enthält und enthält keine weiteren Methoden.

IdeInspectorProperty:
Diese abstrakte Klasse repräsentiert eine IDE-Eigenschaft innerhalb einer Component und spezialisiert *java.beans.FeatureDescriptor*. Neben einem Namen (aus *FeatureDescriptor*) hat sie einen Wert von einem festgelegten Typ. Als Typ kann eine beliebige Klasse festgelegt werden. Zur graphischen Repräsentation hat sie einen *PropertyEditor*. Eine Eigenschaft in einem Inspector kann auch direkt mit einer Eigenschaft in einem Modell verknüpft sein. In diesem Fall kann man mit der Methode *getModelProperty()* den Namen der Eigenschaft im Modell (auf RWI-Ebene) erhalten. Neben *IdeMultipleInspectorProperty* implementieren auch noch eine Reihe anderer Klassen *IdeInspectorProperty*. Darunter auch die *RWIInspectorProperty* aus dem Package *ide.inspector.util.property*. Diese bildet die erwähnte Verknüpfung zur RWI-Ebene, indem sie gleichzeitig auch eine Eigenschaft des Modells darstellt. *RWIInspectorProperty* hat abermals Unterklassen, die jeweils einen bestimmten Typ repräsentieren, z.B. *RWIInspectorBooleanProperty*, *RWIInspectorProperty* oder *RWIInspectorIntegerProperty*. Dass diese Eigenschaften auch Eigenschaften auf RWI-Ebene darstellen, kann, abhängig vom Typ der Eigenschaft und von der Art des Modellelementes, der sie zugewiesen sind, verschiedene Auswirkungen haben. So wird z.B. der Wert (sofern vorhanden) einer neu erstellten *RWIInspectorStringProperty*, die einer Klasse zugewiesen ist, als Javadoc-Kommentar dynamisch in den generierten Code übernommen.

PropertyEditor:
Das Interface *java.beans.PropertyEditor* repräsentiert die graphische Darstellung des Wertes einer Eigenschaft und bietet eine Möglichkeit zu dessen Modifikation. Dies kann z.B. ein einfaches Textfeld oder eine Auswahlliste mit vorgegebenen Werten sein. Durch den Editor kann beeinflusst werden, welche Werte der Anwender der Eigenschaft zuweisen kann. Das

Package *ide.inspector.util.editor* enthält eine Vielzahl von Klassen, die *PropertyEditor* implementieren. Darunter gibt es auch komplexere Editoren, z.B. ein Editor für Dateinamen, der eine Suche im Dateisystem ermöglicht.

6.4.6.2 Erweiterung eines Inspectors

Zur Erweiterung eines Inspectors wird wie in Abschnitt 6.4.4 beschrieben, eine Instanz von *IdeInspectorManager* angefordert, bei der dann eine eigene Implementierung des Interface *IdeInspectorBuilder* registriert werden kann. Immer wenn nun in Together ein Inspector geöffnet wird, wird bei allen registrierten Instanzen von *IdeInspectorBuilder* die Methode *buildInspector()* aufgerufen, wobei der jeweilige Inspector übergeben wird. Die Klasse *util.RwiElementsUtil* stellt statische Methoden zur Verfügung, mit der Information über die aktuell markierten Modellelemente bezogen werden kann. Dadurch ist feststellbar, zu welchem Typ von Modellelement der Inspector gehört, und er kann entsprechend modifiziert werden. Abbildung 6-11 zeigt ein prinzipielles Beispiel, wie in der Methode *buildInspector()* in einer vom Anwender erstellten Instanz von *IdeInspectorBuilder* ein Inspector modifiziert wird. Es wird eine neue Component, d.h. eine Karteikarte, mit einer neuen Eigenschaft vom Typ String hinzugefügt.

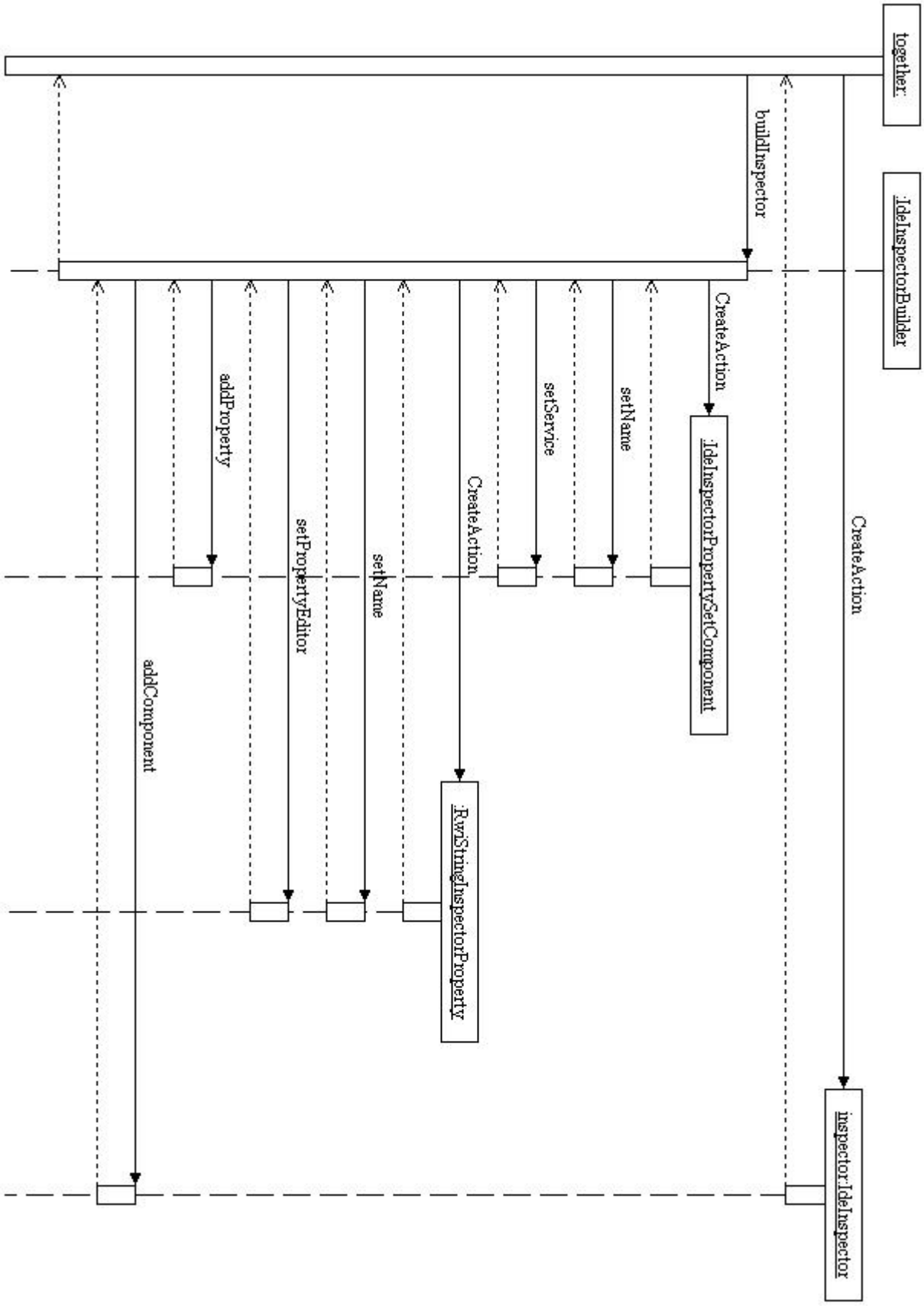

Abbildung 6-11: Hinzufügen einer neuen Component mit einer neuen Eigenschaft vom Typ String zu einem Inspector (Prinzipdarstellung).

6.4.7 Menübefehle

Um Menü- oder Werkzeugleisten in Together um eigene Befehle zu erweitern, ist das Package *ide.command* zu verwenden. Gemäß der Beschreibung in Kapitel 6.4.4 erhält man den *IdeCommandManager*. Dieser stellt Methoden zur Erzeugung eines neuen Befehls oder einer neuen Gruppe von Befehlen zur Verfügung. Eine Gruppe von Befehlen besteht aus (separat zu erzeugenden) einzelnen Befehlen, so besteht beispielsweise in der Hauptmenüleiste die Befehlsgruppe „File“ aus den einzelnen Befehlen „Open“, „Save“, usw. Parameter dieser Methoden sind:

- eine ID des neuen Befehls bzw. der neuen Befehlsgruppe,
- Eigenschaften in Form einer Instanz von *IdeCommandConstraints* und
- ein selbst erstellte Implementierung des Interface *CommandListener*.

Eigenschaften des Befehls- bzw. der Befehlsgruppe legen fest, in welche Menü- oder Werkzeugleiste der Befehl eingefügt werden soll und an welchen Ort innerhalb dieser. Dazu stellt die Klasse *IdeCommandConstraints* vordefinierte Konstanten zur Verfügung.
Mögliche Orte für einen neuen Befehl sind

- die Hauptmenüleiste,
- die Hauptwerkzeugleiste und
- das Kontextmenü zu
    - Modellelementen,
    - Dateien und
    - dem Editorfenster.

Bezüglich der Kontextmenüs ist jeweils festzulegen, ob der Befehl für jeden Typ oder nur für einen bestimmten Typ von Modellelement bzw. von Datei verfügbar sein soll.
Ein *CommandListener* enthält eine Methode *actionPerformed()*, die bei Auswahl eines Befehls aufgerufen wird. Sie enthält beliebigen Code, der bei Aufruf eines Befehls aufgerufen wird.
Erstellte Befehle bzw. Befehlsgruppen enthalten u.a. getter- und setter-Methoden für Eigenschaften wie Sichtbarkeit, ihr Label, ein Icon, eine zugehörige Tastenkombination, einen Separator vor oder hinter ihnen im Menü und ob sie aktiv sind.
Auf diese Weise ist der Umgang mit Befehlen sehr einfach und flexibel möglich. Der Zugriff auf bereits in Together enthaltene Befehle oder Befehlsgruppen ist aber nicht möglich.

6.4.8 Umgang mit Config-Dateien und Konfigurationsdialogen

Das Package *ide.config* stellt Klassen und Interfaces zum Umgang mit Config-Dateien zur Verfügung. Die wesentliche Funktionalität ist zum einen das Aufrufen eines Konfigurationsdialoges (siehe 6.2.2) und zum anderen das Laden (und wieder Entfernen) von Config-Dateien (siehe 6.4.2) in Together. Letzteres ist wichtig, weil dadurch eigene Config-Dateien nach Together geladen werden können, wodurch z.B. Diagramme oder Konfigurationsdialoge beeinflusst werden können. Geladene Config-Dateien können manuell wieder aus Together entfernt werden, was ebenfalls hilfreich sein kann, z.B. wenn ein Module deaktiviert werden soll oder Änderungen, die durch das Laden einer Config-Datei ausgelöst wurden, rückgängig gemacht werden sollen.
Der Aufbau von Konfigurationsdialogen ist über Config-Dateien festgelegt. Der Nutzer kann mit eigenen Config-Dateien Konfigurationsdialoge anpassen und eigene Karteikarten mit Eigenschaften hinzufügen. Den Eigenschaften können Standardwerte zugewiesen werden. Ändert ein Nutzer den Wert einer dieser Eigenschaften, so wird dies von Together

automatisch gespeichert. Bei Änderungen, die nur für das aktuelle Projekt gültig sind, speichert Together den neuen Eigenschaftswert automatisch in der zum Projekt gehörigen *.tpr*-Datei. Das Package *ide.config* ermöglicht auch das Auslesen sowohl projektbezogener als auch global gültiger Eigenschaftswerte.

6.4.9 Aufruf eigener Dialoge

Die Open API stellt einen flexiblen Mechanismus zum Einbinden eigener Dialoge zur Verfügung. Dazu ist das Package *ide.window* zu verwenden. Gemäß 6.4.4 erhält man eine Instanz des Interface *IdeWindowManager*. Diese stellt *create*Dialogtyp*Dialog()*-Methoden zur Verfügung, mit denen Dialoge in Together erstellt und angezeigt werden können, deren Erscheinungsbild der Oberfläche von Together angepasst ist. Es stehen sowohl vordefinierte Dialoge zur Auswahl, die in einem gewissen Rahmen konfiguriert werden können, als auch Dialoge, deren Inhalt und Funktionalität völlig unbeschränkt festlegbar ist. Durch Nutzung von Hilfsklassen können eigene Elemente den Konventionen von Together angepasst werden. Weiterhin enthält das Package ebendiese Funktionalität auch für Knöpfe.

6.5 *Empfehlungen zur Erweiterung*

Da die vorhandenen Unterstützung der UML Erweiterungsmechanismen in Together nur gering ist (siehe 6.3), sollte eine Erweiterung zunächst auf eine Unterstützung der grundlegenden Elemente von UML Profiles, d.h. Stereotypes, Tagged Values und Constraints, abzielen. Die folgenden Abschnitten beschreiben jeweils eine mögliche Stufe der Erweiterung, beginnend bei der einfachsten.

6.5.1 Lightweight Extensions als Eigenschaften vom Typ String

Die einfachste Art der Erweiterung zielt darauf ab, Modellelemente mit Stereotypes, Constraints und Tagged Values zu kennzeichnen, ohne eine semantische Auswertung mit einzubeziehen. Die Lightweight Extensions würden den Modellelementen als einfache Eigenschaften vom Typ String hinzugefügt. Dadurch würden sie zunächst nur im generierten Code als Javadoc-Kommentare eingefügt. Stereotypes wären auch in den Diagrammen als Label sichtbar.
Um dies zu erreichen, müsste lediglich der Inspector (siehe 6.4.6) modifiziert werden. Im Falle der Stereotypes müsste die Auswahlliste der Stereotypes entsprechend angepasst werden. Dies erreicht man durch Modifikation des zur IDE-Eigenschaft „Stereotype“ gehörigen Editors. Zur Unterstützung von Constraints und Tagged Values würden dem Inspector entsprechende neue RWI-Eigenschaften hinzugefügt. Dadurch würden auch diese zusammen mit einem Modellelement gespeichert.
Unter Nutzung von Menübefehlen und Dialogen können neue Stereotypes und Tagged Values vom Anwender definiert werden. Da die Open API beliebige eigene Dialoge zulässt, könnten hierbei Stereotypes auch weitere Elemente oder Eigenschaften zugeordnet werden, z.B. eine Base Class oder Tagged Values. Dementsprechend müsste im *IdeInspectorBuilder*, der zur Modifikation des Inspectors genutzt wird, abgefragt werden, welche Stereotypes und Tagged Values aktuell zur Auswahl stehen sollen. Die Open API ermöglicht dies auch dynamisch,

d.h. es ist im Inspector möglich, bei Änderung der Wahl eines Stereotypes entsprechend die Auswahl von Tagged Values zu modifizieren.
Um in dieser Form verschiedene vereinfachte Profiles mit verschiedenen, vom Anwender eingegebenen Lightweight Extensions zu handhaben, müsste sich das Module in beliebiger Form eigene Dateien anlegen und diese selbst verwalten. Die Auswahl einer dieser Dateien kann über den Konfigurationsdialog geschehen, wodurch Together übernimmt, für jedes einzelne Projekt den Namen oder Pfad der gewählten Datei zu speichern.
Die Realisierung der Erweiterungen dieser einfachen Stufe sollten mit relativ geringem Aufwand möglich sein. Für Stereotypes sind sie prototypisch realisiert (siehe 6.6).

6.5.2 Visualisierung von Tagged Values in Diagrammen

Die Modifikation der Darstellung von Modellelementen in Diagrammen ist allein durch die Open API nicht realisierbar. Stattdessen werden Diagramme über Config-Dateien erstellt bzw. beeinflusst. Da Config-Dateien im Wesentlichen undokumentiert sind (siehe auch 6.4.2) konnte im Rahmen dieser Arbeit nicht in Erfahrung gebracht werden, wie Tagged Values in Diagrammen Modellelementen hinzugefügt werden können. Als ein sehr schlichter Ersatz könnten Kommentare erzeugt werden, die den entsprechenden Modellelementen im Diagramm angefügt werden.

6.5.3 Graphische Repräsentation von Stereotypes

Bezüglich der Visualisierung von Stereotypes durch ein Icon, treten prinzipiell die gleichen Probleme wie bei 6.5.2 auf. Es ist aber möglich, die graphische Repräsentation eines Modellelements zu beeinflussen. In Together geschieht dies standardmäßig in der Config-Datei *view.config*. Es müsste daher eine eigene Config-Datei in Together geladen werden, die *view.config* kopiert und für die zu verändernden Eigenschaften entsprechend modifizierte Werte setzt. Das in 6.4.2 erwähnte „Config HOWTO“ [Play01] enthält dazu einige Erklärungen und gibt mögliche zu setzende Werte an. Folglich ist die Änderung der graphischen Repräsentation von Modellelementen möglich. Jedoch sind die vordefinierten Werte auf bestimmte vektorgraphische Formen begrenzt. So ist es beispielsweise möglich, eine Klasse statt in einem Rechteck in einem Oval oder einem Würfel anzuzeigen. Auf die Möglichkeit der Zuweisung eines Icons sind jedoch keine Hinweise zu finden. Es ist davon auszugehen, dass eine Repräsentation von Stereotypes durch ein Icon nicht möglich ist.

6.5.4 Graphische Definition von Stereotypes

Bezüglich einer graphischen Definition wäre es wohl sinnvoll, Stereotypes und damit Profiles, in einem eigenen Diagramm zu definieren. Die Erweiterung von Together um einen entsprechenden neuen Diagrammtyp wäre mit Hilfe von Config-Dateien möglich, wobei dabei das „Config HOWTO“ [Play01] Unterstützung leistet. Die Implementierung von Stereotypes als Modellelement würde auch die Nutzung von Operationen erschließen, die die Open API Modellelementen vorbehält und könnten damit einfacher und besser in Together integriert werden. Die erstellten Profiles könnten z.B. als Diagramme gespeichert werden. Zu

beachten ist, dass UML 1.4 noch keine endgültig befriedigende Festlegung zur graphischen Stereotype-Definition vorgibt (siehe 4.2.9).

6.5.5 Semantische Auswertung

Um dem Anwender zu ermöglichen, den Elementen in Profiles Semantik zuzuordnen, die sich z.B. auf die Codegenerierung beziehen kann, scheinen sehr aufwendige Schritte notwendig zu sein, da Together hierzu keine Unterstützung anbietet. Zunächst müsste eine Sprache zu Ausdruck von Semantik implementiert werden. Erschwerend kommt hinzu, dass Together sich intern nicht sehr stark am UML Metamodell orientiert, wodurch z.B. OCL-Anweisungen nur mühsam oder gar nicht auf Modellelemente angewandt werden könnten. Eine derartige Erweiterung scheint nur schwer vorstellbar.

6.5.6 Zusammenfassung

Zusammenfassend kann gesagt werden, dass einfache Erweiterungen realistisch erscheinen. Dazu gehören zumindest die Erweiterungen aus 6.5.1 und ferner eventuell aus 6.5.2 und 6.5.4. Weitergehende Unterstützung im größeren Maße scheint schwierig umsetzbar. Grund dafür sind die Grenzen der Open API sowie die Unübersichtlichkeit der undokumentierten Config-Dateien. Eine Unterstützung von UML Profiles vergleichbar mit der von Objecteering ist mit der aktuellen Version von Together sicherlich nicht denkbar.

6.6 *Prototypische Umsetzung*

Die prototypische Umsetzung orientiert sich an den Empfehlungen aus 6.5.1. Diese Empfehlungen werden beispielhaft für Stereotypes in Klassendiagrammen umgesetzt. Es wird demnach die Möglichkeit geschaffen, Dateien mit eigenen Stereotypes zu erzeugen und zur Anwendung auf Klassen auszuwählen. Konkret soll folgende Funktionalität implementiert werden:

1. Eingabe eigener Stereotypes und Speichern in einer Datei nach Wahl.
2. Bearbeiten einer Datei mit eigenen Stereotypes.
3. Auswahl einer Datei mit Stereotypes als Standardwert für alle Projekte.
4. Auswahl einer Datei mit Stereotypes für das aktuelle Projekt.
5. Anzeige der gewählten Stereotypes als Auswahlliste im Inspector.
6. Sonstige Behandlung eigener Stereotypes analog der Behandlung vordefinierter Stereotypes.

Zur Realisierung dieser Funktionalität wurde ein Module (siehe 6.4.1, „Modules“) *Manage Stereotypes* implementiert. Es ist vom Typ *Activable* (siehe 6.4.1.1, „Typen von Modules“), d.h. es wird im Explorerfenster oder in der Hauptmenüleiste unter dem Menüpunkt *Options/ActivableModules* aktiviert oder deaktiviert, wobei dieser Zustand über verschiedene Sitzungen hinweg persistent bleibt. Auf der Karteikarte *Modules* im Explorerfenster ist das Module im Ordner *Early Access* zu finden. Alle zum Module gehörigen Dateien sind im

Verzeichnis von Together unter dem Pfad *modules/com/togethersoft/modules/profile* zu finden.
Bezüglich der geforderten Funktionalität wurden Punkt 1. und 2. (siehe oben) realisiert, indem gemäß 6.4.7, „Menübefehle“, das Hauptmenü unter dem Menüpunkt *Tools* um zwei eigene Befehle *New File...* bzw. *Edit File...* erweitert wurden. Diese rufen eigene Dialoge (siehe 6.4.9, „Aufruf eigener Dialoge“, auf, die Eingabe und Löschen eigener Stereotypes ermöglichen. Zur Auswahl eines Pfads bzw. Namens für die zu erstellende oder zu ändernde Datei werden vorgegebene Dialoge (siehe ebenfalls 6.4.9) verwendet. Die Namen der eingegebenen Stereotypes werden in Textdateien gespeichert. Das Module schlägt standardmäßig eine Dateinamenerweiterung *.ste* (für „Stereotypes“) vor, die durch eine beliebige eigene ersetzt werden kann. Durch die Verwendung des Textformats können die Dateien auch beliebig mit einem Texteditor bearbeitet werden und somit auch mit dem Together-internen Editor.
Um eine Datei mit eigenen Stereotypes bei der Modellierung anwenden zu können wird sie im allgemeinen Konfigurationsdialog (siehe 6.2.2, „Grundlegende Dialoge“) ausgewählt. Der Dialog ist im Hauptmenü unter dem Menüpunkt *Options/Default* bzw. *Options/Project* aufzurufen, wobei die Auswahl der Datei dementsprechend entweder als Standardwert für alle Projekte oder nur für das aktuelle Projekt gilt. Somit wurden Punkt 3. und 4. der geforderten Funktionalität (siehe oben) realisiert. Der Konfigurationsdialog wurde gemäß 6.4.8, „Umgang mit Config-Dateien und Konfigurationsdialogen“ unter Verwendung einer Config-Datei erweitert. Zur Auswahl der Datei wurde dabei auf einen Standardmechanismus zurückgegriffen. Dieser ermöglicht das Auswählen einer beliebigen Datei durch einen Standarddialog. Eine Konfiguration diese Dialogs, so dass beispielsweise nur Dateien einer bestimmten Endung vorgeschlagen werden, konnte nicht erreicht werden. Eine ursprünglich geplante Lösung sah vor, alle Stereotype-Dateien vor dem Anwender zu verbergen und nur intern in einem Unterverzeichnis des Modules abzuspeichern, so dass der Anwender nur aus einer Liste von gültigen Stereotype-Dateinamen auszuwählen braucht. Hierbei traten Probleme im Umgang mit den kaum dokumentierten Config-Dateien auf, da keine Lösung gefunden werden konnte, um eine solche Liste mit aktuell gültigen Namen dynamisch erzeugen zu können. Prinzipiell bestünde für ein solches Problem der pragmatische Lösungsansatz, die Config-Dateien selbst dynamisch zu modifizieren, was möglich ist, da diese im Textformat gespeichert sind. Eine solche Lösung wäre jedoch unsauber und fehleranfällig. Daher wurde die oben beschriebene alternative Lösung implementiert, den Nutzer einen beliebigen Dateinamen angeben zu lassen. Der Vorteil dieser Lösung ist, dass der Anwender nach eigenen Bedürfnissen ein Verzeichnis auswählen und erstellte Dateien in verschiedene Verzeichnisse gruppieren kann.
Die standardmäßig in Together enthaltenen Stereotypes sind zum einen in der Datei *standard.ste* im Verzeichnis des Modules im Ordner *files* enthalten, und werden zum anderen auch dann verwendet, wenn kein gültiger Wert als Dateiname ausgewählt wurde.
Punkt 5. der geforderten Funktionalität wurde implementiert, indem analog zu 6.4.6.2, „Erweiterung eines Inspectors“, der Eigenschaft des Namens *stereotype* ein neuer Editor mit eigenen Werten hinzugefügt wurde. Die anzuzeigenden Stereotypes stammen aus der ausgewählten Stereotype-Datei. Da Together die Werte, die in Konfigurationsdialogen festgelegt werden, automatisch verwaltet (siehe 6.4.8), kann ist der Name der aktuell gewählten Datei mit wenigen Schritten von Together erhältlich.
Ansonsten werden eigene Stereotypes gemäß dem standardmäßigen Umgang in Together mit Stereotypes behandelt. Daraus ergibt sich, dass Stereotypes außer ihrem Namen keine weiteren Eigenschaften haben und dass sie als Javadoc-Kommentare in generierten Code eingefügt werden.

7 Zusammenfassung und Ausblick

Weiterhin wurde die Umsetzung von Profiles in der UML Spezifikation untersucht. Während in UML 1.3 Profiles nur informativ erwähnt sind, sind in der Version 1.4 bereits weitreichende Festlegungen zu Profiles getroffen. Ein expliziter Vergleich macht deutlich, dass die Anforderungen an UML Profiles durch UML 1.4 im Wesentlichen erfüllt werden. Bezüglich der offen gebliebenen Punkte sind zumindest für die obligatorischen Anforderungen Lösungen denkbar, wodurch deren vollständige Erfüllung in der nächsten Version der UML zu erhoffen ist.
An CASE-Werkzeugen mit Unterstützung von UML Profiles konnte nur Objecteering von Softeam gefunden werden. Dieses bietet eine weitreichende Unterstützung, die anhand eines Beispiels und mit Hilfe von Abbildungen möglichst anschaulich dokumentiert wurde. Als Ergebnis bleibt festzuhalten, dass sich Objecteering an der Version 1.3 der UML Spezifikation orientiert und diese auch in weiten Teilen umsetzt. Allerdings musste festgestellt werden, dass z.B. das UML Profile für CORBA nicht vollständig in Objecteering abgebildet werden kann. Festlegungen aus UML 1.4 bezüglich UML Profiles bleiben demzufolge in Objecteering unberücksichtigt. Der Schwerpunkt wird auf proprietäre Mechanismen gelegt. Diese sind sehr mächtig und erlauben in weitreichender Art die Zuweisung von Semantik an ein UML Profile. Sie könnten daher auch als Orientierung für weitere Arbeiten und eigene Ansätze dienen. Eine Möglichkeit zum Austausch erstellter UML Profiles mit anderen CASE-Werkzeugen ist aufgrund des proprietären Charakters der UML Profiles in Objecteering nicht vorgesehen.
Für die Erweiterung eines CASE-Werkzeuges um Unterstützung von UML Profiles wurde Together des Herstellers Togethersoft ausgewählt. Als Erweiterungsmechanismus wird von Togethersoft eine API zur Verfügung gestellt, jedoch war festzustellen, dass sogenannte Config-Dateien ebenso zu berücksichtigen sind. Diese Dateien werden auch von Togethersoft selbst zur Erweiterung der Basisfunktionalität Togethers verwendet und bieten Möglichkeiten die über die der API hinausgehen. Während einfache Erweiterungen relativ leicht realisierbar sind, was auch durch eine prototypische Umsetzung demonstriert wurde, gelangt man bei komplexeren Erweiterungen schnell an Grenzen. Im Fall der API liegen die Grenzen in den prinzipiellen Möglichkeiten der verfügbaren Objekte und Methoden, bei Config-Dateien zumeist in der fehlenden Dokumentation. Empfehlungen waren daher nur für einige Teilerweiterungen möglich. Eine Unterstützung von UML Profiles ähnlich der von Objecteering scheint unter den aktuellen Voraussetzungen kaum erreichbar.

Quellenverzeichnis

[Gree01] Jack Greenfield, UML Profile for EJB (draft), sun, Mai 2001, http://www.jcp.org/aboutJava/communityprocess/review/jsr026

[Jeck00] Mario Jeckle, Die vier-Schichten Metamodellarchitektur, GROOM-Workshop April 2000, http://www.jeckle.de/files/groom_20000404.pdf

[OMG99a] OMG, White paper on the Profile mechanism, Version 1.0., OMG document number: ad/99-04-07, April 1999.

[OMG99b] OMG, Requirements for UML Profiles, Version 1.0., OMG document number: ad/99-12-32, Dezember 1999

[OMG00a] OMG, UML Modeling Language Specification, Version 1.3., OMG document number: formal/00-03-01, März 2000.

[OMG00b] OMG, UML Profile for CORBA Specification, V.1.0, OMG document number: ptc/00-10-01, Oktober 2000.

[OMG01a] OMG, UML Modeling Language Specification (draft), Version 1.4, OMG document number: ad/01-02-13, Februar 2001

[Play01] Greg Playle, Config Files Explained: The Config HOWTOv0.6, Together Community, April 2001, http://www.togethercommunity.com/textual-stuff/howto/1000-config-files-explained/article.shtml

Internetquellen

[INET01] http://www.omg.com

[INET02] http://www.softeam.fr

[INET03] http://www.togethersoft.com

[INET04] http://argouml.tigris.org

Abbildungsverzeichnis