3 UML-2-Sprachdefinition

Im vorangegangenen Kapitel haben wir die Grundkonzepte der Sprache UML 2 kennen gelernt. Sie werden sich fragen, wie man herausfindet, welche »Wörter« Bestandteil der Sprache mit dem Namen UML ist. Als Antwort kann gelten, dass alle diese Wörter irgendwo im Dschungel der Spezifikationsdokumente zu finden sind. Aber – es gibt einen einfacheren Weg, um herauszufinden, welche Wörter man in der Sprache UML benutzen darf und wie diese Wörter zu Sätzen und letztendlich zum Text (in einem UML-Modell) zusammengesetzt werden.

Dieser Weg heißt UML-2-Metamodell. In diesem Kapitel wollen wir die »Wahrheit« über die Definition der Sprachelemente von UML 2 ergründen und diese findet sich gerade in besagtem Metamodell. Was aber ist ein Metamodell?

3.1 Meta-Modellierung

Die deutsche Sprache ist mit der deutschen Sprache selbst erklärt, der Duden regelt, welche Wörter es gibt, was sie bedeuten und wie sie zu Sätzen zusammengestellt werden dürfen. Die deutsche Sprache ist also »mit sich selbst« erklärt. Dieses Vorgehen hat sich als äußerst nützlich erwiesen – und wurde auf die Definition von Modellierungssprachen übertragen: Die Sprache UML 2 ist mit der Sprache UML 2 erklärt. Jedes Konstrukt, das man in UML benutzen darf, ist durch die Spracherklärung festgelegt. So regelt diese beispielsweise, dass es die Wörter Klasse (*Class*) und *Operation* in der Sprache gibt. Darüber hinaus legt sie fest, welche Beziehungen zwischen den Wörtern der Sprache bestehen, also beispielsweise, dass Klassen Operationen enthalten dürfen. Das Resultat der Sprachdefinition selbst ist natürlich wieder ein Modell. Da dieses Modell diejenigen Wörter definiert, die in UML-2-Modellen verwendet werden dürfen und Aussagen über die Semantik von Modellen trifft, erhält es die Bezeichnung Metamodell.

Nun gibt es aber einen Unterschied zur Erklärung natürlicher Sprachen: Es ist einfach nicht praktikabel, den gesamten Sprachumfang von UML zu verwenden, um UML zu erklären. Aus diesem Grunde schränkt man die Ausdrucksmittel im Metamodell soweit ein, dass diese gerade noch ausreichen, um die Sprache zu erklären. Die Festlegung der Teilsprache, die verwendet werden darf, um die Sprache UML zu erklären,

wird wiederum in einem Modell vorgenommen. Dieses erhält die Bezeichnung Meta-Metamodell.

Stellen Sie sich das Meta-Metamodell als Werkzeugkasten mit ganz wenigen, essentiellen Hilfsmitteln vor, sagen wir, mit einem Hammer, Feuer und Eisen. Sie können diese wenigen Werkzeuge nutzen, um sich neue Werkzeuge zu konstruieren. Um die Sprache UML zu erklären, brauchen wir nicht allzu viele Werkzeuge, es reichen uns diejenigen, die in Kapitel 2 eingeführt worden sind.

Versuchen wir, uns den Sachverhalt an einem Beispiel zu verdeutlichen. Modelle enthalten Modellelemente: Im Metamodell der Sprache UML muss also eine Konstruktion existieren, die erklärt, das in Modellen Modellelemente auftreten können. Um genau diese Aussage zu treffen, wird im Metamodell der Sprache UML die Metaklasse *Element* eingeführt. Es wird also der Begriff *Class* im Metamodell benutzt, um das Konzept *Modellelement* zu erklären. *Class* ist somit Element des eingeschränkten Sprachumfangs, der genutzt wird, um die Sprache UML zu erklären, und gehört damit in unseren Werkzeugkasten. Da es keine direkten Instanzen von *Element* in Modellen geben kann, sondern nur Instanzen spezialisierter Konzepte, ist diese Klasse entsprechend als abstrakt definiert.

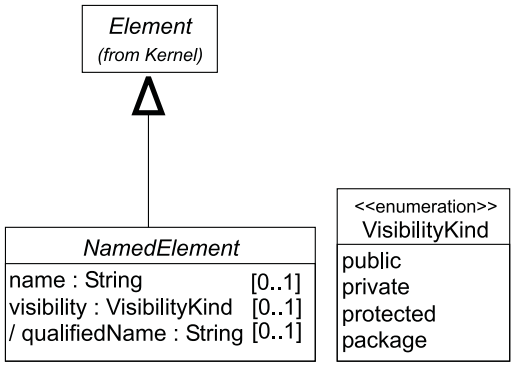

Abbildung 3.1: Metamodellauszug für Element

Diese spezialisierten Konzepte werden im Metamodell wiederum durch Metaklassen ausgedrückt, die Spezialisierung selbst durch Generalisierungsbeziehungen zwischen diesen Metaklassen. Wenn wir uns ausgehend von der Klasse *Element* entlang der Generalisierungsbeziehungen durch das Metamodell »hangeln«, so erreichen wir zunächst die abstrakte Klasse *NamedElement*. In dieser Klasse ist erklärt, dass Elemente in Modellen Namen tragen dürfen, aber nicht müssen. Der Name eines Modellelements, wie beispielsweise ein Klassenname, der Name eines Attributs oder auch der Name eines Modells ist eine Eigenschaft von Instanzen der Metaklasse *NamedElement*. Diese Metaklasse erhält dementsprechend ein Attribut mit dem Namen *name*, dem Typ

String und der Vielfachheit *[0..1]*. Des Weiteren regelt die Metaklasse *NamedElement*, dass Modellelemente Aussagen über ihre Sichtbarkeit enthalten dürfen. Die Konzepte Attribut (benutzt zur Definition von Eigenschaften der Metaklassen) und Generalisierung (eingesetzt zur Spezialisierung von Metaklassen) gehören also auch in unseren Werkzeugkasten zur Sprachdefinition.

Beim Weiterhangeln stoßen wir auf die abstrakte Metaklasse *Namespace* (Namensraum). Instanzen dieser Metaklasse enthalten Instanzen der Metaklasse *NamedElement*, alle in einem Namensraum zusammengefassten Elemente müssen anhand ihres Namens unterscheidbar sein. Die Enthaltenseinsbeziehung wird im Metamodell durch eine Assoziation zwischen *Namespace* und *NamedElement* ausgedrückt, die in Richtung *NamedElement* navigierbar ist. Das Konzept Assoziation ist – mit den in Kapitel 2 erklärten Eigenschaften – ebenfalls Element unseres Werkzeugkastens zur Sprachdefinition.

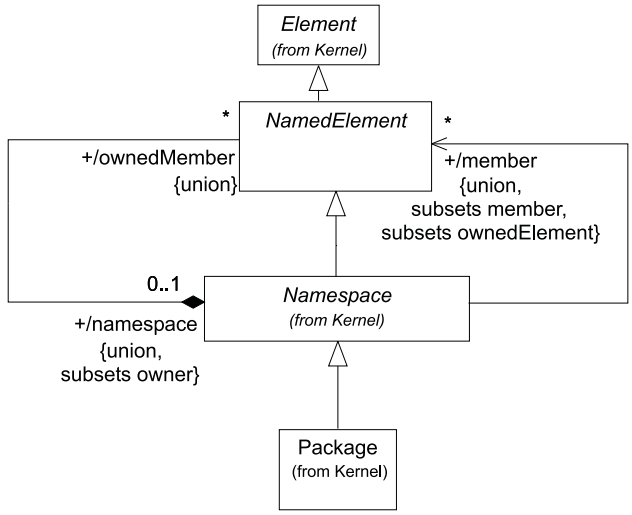

Abbildung 3.2: Metamodellauszug für Namensraum und Paket

Eine Spezialisierung der Metaklasse *Namespace* ist die Metaklasse *Package*. Instanzen dieser Metaklasse dürfen in Modellen auftreten, sie ist also nicht abstrakt. Für Instanzen von *Package* ist demzufolge auch erklärt, wie sie in einem Modell zu notieren sind – durch das in ***Kapitel 2*** eingeführte Paketsymbol.

Für alle Sprachelemente von UML ist im Metamodell definiert, in welchem Zusammenhang sie in Modellen auftreten dürfen und welche Eigenschaften und Beziehungen zu anderen Sprachelementen zulässig sind. Das Metamodell definiert die gesamte Begriffswelt der Sprache UML und regelt, wie die Begriffe zu Sätzen zusammengesetzt

werden dürfen. Für alle nicht abstrakten Metaklassen wird durch Notationsfestlegungen erklärt, wie deren Instanzen und die Eigenschaften dieser Instanzen in Diagrammen sichtbar gemacht werden können.

Dieses Vorgehen bei der Sprachdefinition entspricht dem in Kapitel 1 erläuterten Prinzip der Trennung zwischen der Begriffswelt und der Notation der Begriffe. Wird der Sprache ein Begriff hinzugefügt, so wird er zunächst in die Begriffswelt oder Konzeptraum – d.h. in das Metamodell – aufgenommen und es wird erklärt, wie er mit anderen Begriffen in Beziehung steht. Falls der Begriff in der Sprache auch darstellbar sein soll, so müssen Notationen für den Begriff und seine Beziehungen zu anderen Begriffen formuliert. Das Metamodell von UML selbst ist mit einer Untermenge der gleichen Begriffswelt festgelegt. Diese Untermenge – das Meta-Metamodell – erklärt, welche Begriffe benutzt werden können, um einen Begriff der Sprache UML festzulegen.

Ordnet man das Meta-Metamodell, das UML-Metamodell, ein beliebiges UML-Modell sowie den in diesem Modell ausgedrückten Sachverhalt in Schichten an, so ergibt sich die Vier-Schichten-Modellierungsarchitektur (*Four Layer Modeling Architecture)* wie in Abbildung 3.3 dargestellt.

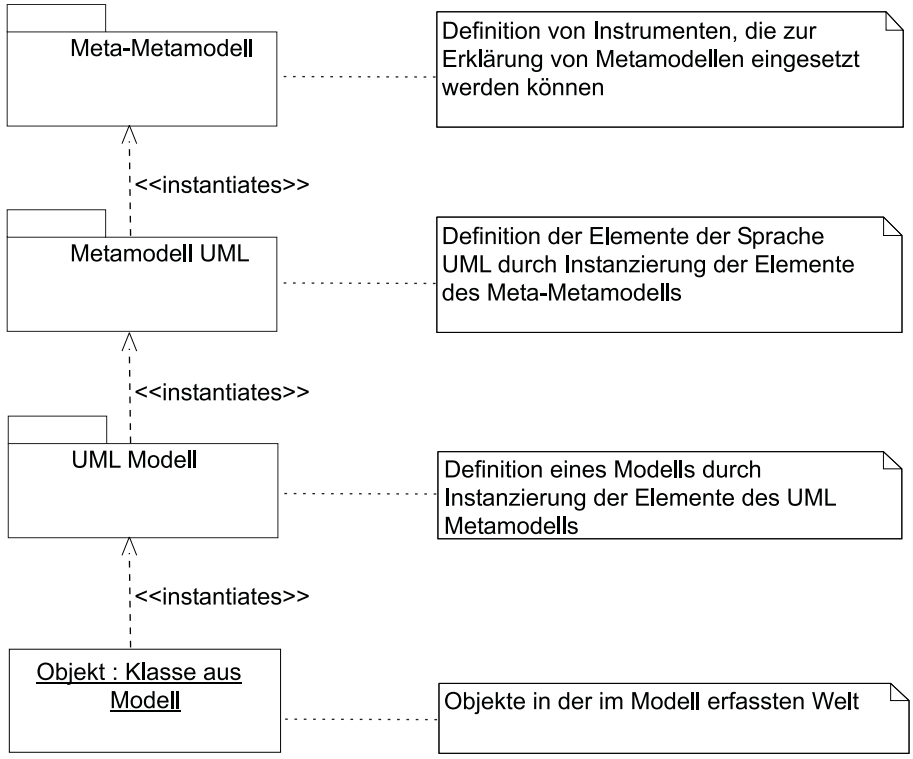

Abbildung 3.3: Four Layer Modeling Architecture

Die erste Ebene dieser Architektur bildet das Instrumentarium zur Definition von Metamodellen – das Meta-Metamodell. Dieses fasst die Grundkonzepte der Objektorientierung, die in Kapitel 2 vorgestellt wurden, zusammen. Die Tatsache, dass es sich um objektorientierte Konzepte handelt, impliziert, dass Metamodelle *objektorientiert* verfasst werden können. Dies verleiht einer mit diesen Mitteln vorgenommenen Sprachdefinition eine hohe Ausdruckskraft und erlaubt eine spätere Erweiterung und Adaption der einmal definierten Sprache mittels Spezialisierung und Re-Definition. Verglichen mit »klassischen« Mitteln der Sprachdefinition, wie man sie beispielsweise in der XML-Umfeld gebraucht, bietet die beschriebene Technologie der Meta-Modellierung einen entscheidenden Vorteil: Die Konstruktion von Sprachen erfolgt durch den Einsatz objektorientierter Mittel. Natürlich kann man sich oberhalb des Meta-Metamodells ein weiteres Modell (das Meta-Meta-Metamodell) vorstellen, dass durch das Meta-Metamodell instanziert wird. Oberhalb dieses wiederum ein weiteres usw. Nach langwierigen Diskussionen in der Gemeinde der Metamodellierer kam man jedoch zu der Schlussfolgerung, dass weitere Metaschichten keinen zusätzlichen Abstraktionsgewinn bringen und demzufolge das Meta-Metamodell die oberste Schicht in der *Four Layer Modeling Architecture* bildet.

Das UML-Metamodell als zweite Ebene ist ein Modell, das mit den im Meta-Metamodell erklärten Instrumenten konstruiert ist. Es legt alle Sprachkonstrukte sowie die erlaubten Beziehungen zwischen Instanzen dieser Sprachkonstrukte fest. Anhand des Metamodells kann also überprüft werden, ob ein Modell (statisch) korrekt ist. Um diesen Sachverhalt zu illustrieren, betrachten wir den in Kapitel 2 eingeführten Begriff der Generalisierung: Eine Klasse darf keine, eine oder beliebig viele Superklassen deklarieren. Natürlich muss diese Aussage im Metamodell verifiziert werden können (vgl. Abbildung 3.4).

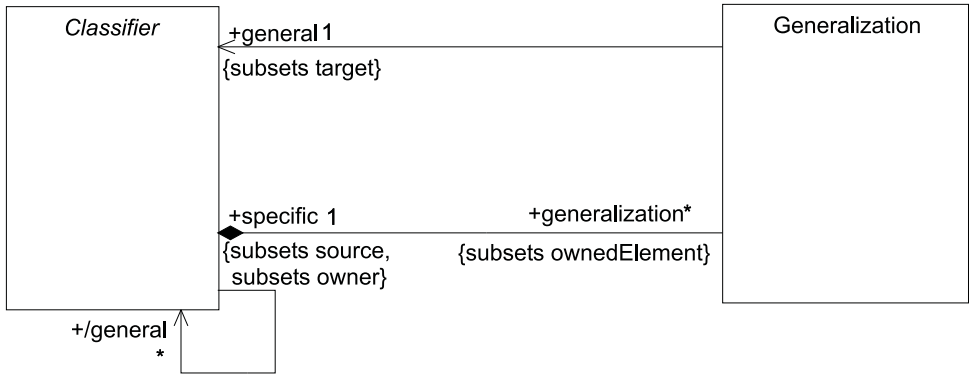

Abbildung 3.4: Generalisierung im Metamodell

Das Metamodell definiert den Begriff Generalisierung durch die Metaklasse *Generalization*, das Konzept der Klasse wird indirekt durch die Metaklasse *Classifier* ausgedrückt. Zwischen diesen beiden Metaklassen ist eine Komposition definiert, die ausdrückt, dass eine Instanz von *Classifier* an beliebig vielen Generalisierungen beteiligt sein kann, also beliebig viele Instanzen der Metaklasse *Generalization* komponiert. Die Vielfachheit * erklärt dementsprechend Mehrfachvererbung. Würde man diese Vielfachheit in *[0..1]* ändern, so wäre zwischen Klassen nur eine einfache Vererbung – wie z.B. in der Sprache Java – erlaubt. Die Metaklasse *Generalization* verfügt über ein navigierbares Assoziationsende mit dem Namen *general*. Instanzen von *Generalization* verweisen auf eine Instanz von *Classifier* – diese Instanz ist gerade die Superklasse bei der Generalisierung von Klassen. Das Metamodell erklärt auf diese Art und Weise strukturelle Regeln, die die so genannte statische Semantik definieren.

Ein UML-Modell als dritte Ebene beinhaltet ausschließlich Elemente, die Instanzen von Metamodellelementen sind. Stellen wir uns in einem Modell die Klassen *Vehicle* und *Aircraft* vor, wobei zwischen beiden eine Generalisierungsbeziehung in Richtung *Vehicle* besteht. *Vehicle* und *Aircraft* können in einem Diagramm mit dem aus Kapitel 2 bekannten Klassensymbol notiert werden. Die Darstellung der Generalisierungsbeziehung erfolgt mit dem ebenfalls bekannten Pfeil. Die Klassen *Vehicle* und *Aircraft* sind Instanzen der Metaklasse *Class*. Die Generalisierungsrelation zwischen beiden ist eine Instanz der Metaklasse *Generalization*, wobei *Aircraft* die Rolle *specific* und *Vehicle* die Rolle *general* im Kontext der Assoziationen zwischen *Generalization* und *Classifier* einnehmen (vgl. Abbildung 3.5).

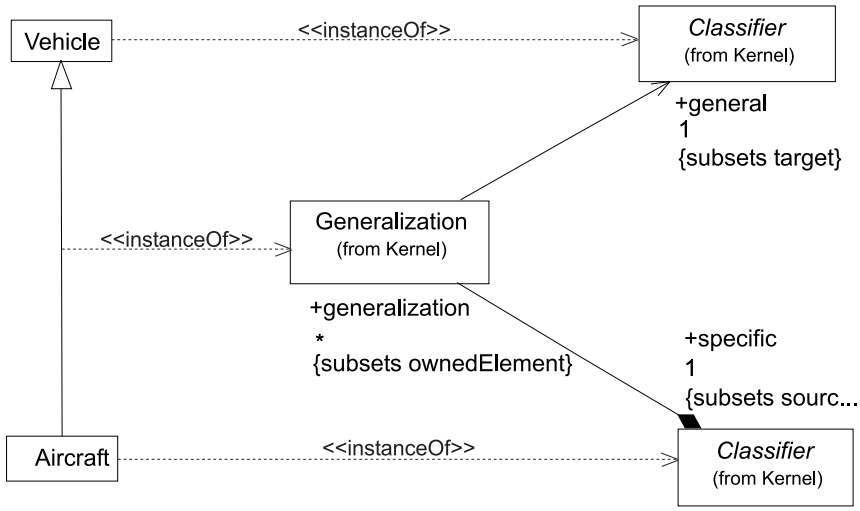

Abbildung 3.5: Aircraft und Vehicle als Instanzen von Metaklassen

Die unterste Ebene der *Four Layer Modeling Architecture* wird durch Instanzen von Elementen gebildet, die in einem Modell erfasst sind, also beispielsweise Objekte vom Typ *Aircraft*.

Vergleicht man eine mittels Meta-Modellierung vorgenommene Sprachdefinition mit der herkömmlichen Erklärung von Programmiersprachen, so findet man eine Analogie zwischen den Begriffen »Abstrakte Syntax« und »Metamodell«. Die abstrakte Syntax einer Programmiersprache erklärt den Aufbau einer Sprache, sie abstrahiert dabei von konkreten syntaktischen Konstrukten. Häufig wird bei der Angabe eine Syntaxdefinition das Prinzip der Rekursion verwendete: Ein Satz der Sprache besteht aus Teilsätzen, diese wiederum aus Teilsätzen und Wörtern usw. Ein Metamodell definiert ebenfalls den Aufbau einer Sprache – allerdings unter Nutzung objektorientierter Modellierung.

```
statement := variableDeclaration
  | ( expression ";" )
  | ( statementBlock )
  | ( ifStatement )
  |  ( doStatement )
  |  ( whileStatement )
  |  ( forStatement )
  |  ( tryStatement )
  |  ( switchStatement )
  |  ( "synchronized" "(" expression ")" statement )
  |  ( "return" [ expression ] ";" )
  |  ( "throw" expression ";" )
  |  ( identifier ":" statement )
  |  ( "break" [ identifier ] ";")
  |  ( "continue" [ identifier ] ";")
  |  (";")
```

Was kann man mit einer Sprachdefinition eigentlich anfangen? In diesem Buch erfüllt das Metamodell der UML 2 einen wesentlichen Zweck – wir studieren es, um uns über die Konzepte der Sprache und deren Zusammenhänge zu informieren. Dies ist aber natürlich nicht der einzige Zweck eines Metamodells. Wir wollen wiederum den Zusammenhang von »klassischer« Sprachdefinition und Metamodellierung heranziehen und die ähnliche Rolle beider in Bezug auf Werkzeugunterstützung analysieren.

Aus der Definition des Wortschatzes für eine Programmiersprache entsteht – bei adäquater Notation und unter Nutzung von Werkzeugen sogar automatisch – ein Programm zur lexikalischen Analyse eines Quelltextes eines Programms dieser Sprache. Aus der Syntax entsteht – ebenfalls automatisch – ein so genannter Parser, der die syntaktische Korrektheit des Quelltextes überprüft. Dieser Parser liefert als Ausgabe einen abstrakten Syntaxbaum, in dem die Blätter den syntaktischen Konstrukten des Quelltextes entsprechen.

Eine ähnliche Rolle spielt das Metamodell einer Sprache wie UML. Ein Modellierungswerkzeug erfasst die Eingaben des Modellierers – den »Quelltext«. Aus diesen Eingaben wird ein Graph erzeugt, dessen Knoten Instanzen der Elemente des Metamodells sind. Schlägt die Erzeugung des Graphen fehl, so ist die Eingabe, also das Modell, falsch. Zur Illustration wollen wir noch einmal die Generalisierungsbeziehung zwischen Klassen bemühen. Wenn wir aus dem Modell mit den Klassen *Aircraft* und *Vehicle* einen Graphen des Metamodells instanzieren wollen, so erhalten wir die in Abbildung 3.5 dargestellte Situation.

Nun wollen wir das Metamodell der UML für einen Augenblick manipulieren, indem wir die Vielfachheit des Assoziationsendes *generalization* in *[0..1]* ändern. Des Weiteren stellen wir uns vor, die Klasse *Aircraft* ist nicht nur eine Spezialisierung der Klasse *Vehicle*, sondern auch einer Klasse *Carrier*. Aus diesem Modell könnten wir keinen Graphen erzeugen, der unserem manipulierten Metamodell entspricht, das Modell ist – syntaktisch – falsch.

Man kann aus dem Metamodell der UML eine Softwarebibliothek erzeugen, die als Eingabe ein UML-Modell akzeptiert und als Ausgabe einen Graphen generiert, dessen Knoten und Kanten Instanzen der Elemente des UML-Metamodells repräsentieren. Ein UML-Werkzeug nutzt diese Softwarebibliothek zur Verifikation der statischen Semantik für ein konkretes Modell. Die Verallgemeinerung dieser Technologie, die für beliebige Metamodelle funktioniert, heißt *Meta Object Facility* [MOF 03]– und wird uns im Weiteren nochmals begegnen.

Da sich der Sprachstandard zwischen Metamodell und Notation sehr wohl unterscheidet, lassen sich nun auch die – manchmal ominösen – Präsentationsoptionen einordnen: Man kann für ein und dasselbe Konzept im Metamodell verschiedene Darstellungsformen gestatten. Erinnern Sie sich an die verschiedenen Möglichkeiten, Elemente (Instanzen der Metaklasse *PackageableElement*) eines Pakets grafisch zu formulieren. Mit all diesen Möglichkeiten wird stets das Gleiche ausgedrückt: Beziehungen zwischen einer Instanz der Metaklasse *Package* und Instanzen der Metaklasse *PackageableElement*. Bindend für die Korrektheit und den Inhalt eines Modells ist immer das Metamodell, die Notation ist zweitrangig. Es gibt in der UML sogar Sprachkonstrukte, die zwar durch das Metamodell semantisch erklärt sind, für die aber keine Notation normiert wurde (vgl. Kapitel 4.3.5)

3.2 Das UML-2-Metamodell

Wir werden im Folgenden einige ausgewählte Aspekte des UML-Metamodells etwas näher betrachten. Später werden wir – wie angekündigt – den Umstand, dass alle Sprachfeatures von UML im Metamodell erfasst sind, zur Erläuterung dieser Features nutzen, Ihnen also die Sprache UML ausgehend vom Metamodell näher bringen.

3.2.1 Metamodell-Organisation

Im vorangegangenen Abschnitt haben wir die grundlegenden Konzepte Meta-Metamodell und Metamodell diskutiert. Das Meta-Metamodell definiert das Instrumentarium zur Definition von Metamodellen, das Metamodell selbst definiert eine Sprache, wie beispielsweise die UML 2. Dieses allgemeine Prinzip spiegelt sich ebenfalls durch die Spezifikationsdokumente, die die UML 2 normieren, wider. Ein erstes separates Dokument erklärt das Meta-Metamodell, benannt als *UML Infrastructure* [UML Infra 03]. Die Konzepte, die durch das Meta-Metamodell erklärt werden, entsprechen im Wesentlichen denen, die wir in Kapitel 2 kennen gelernt haben. Ein weiteres Dokument (*UML Superstructure,* [UML Super 03]) erklärt die UML 2 selbst – und benutzt dazu die *UML Infrastructure*. Diese Nutzung findet auf zwei Arten statt:

- Alle Elemente des UML-Metamodells sind Instanzen der Elemente der *UML Infrastructure*, also des Meta-Metamodells und
- das UML-Metamodell importiert alle Elemente der *UML Infrastructure*.

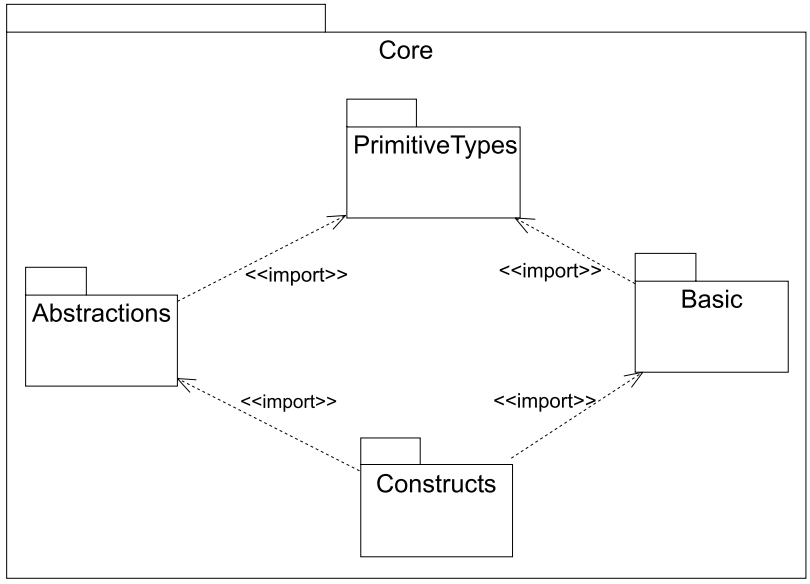

Abbildung 3.6: Core-Package der UML Infrastructure mit Sub-Packages

Die erstgenannte Form der Benutzung entspricht unserer erwarteten Instanzierung des Meta-Metamodells zur Formulierung eines Metamodells für UML. Die zweite Benutzungsform erscheint zunächst verwirrend, da sich ja die *UML Superstructure* und die *UML Infrastructure* auf unterschiedlichen Metaebenen befinden. Eine Auflösung dieses Konflikts besteht darin, dass die Konzepte der *UML Infrastructure* dem Kern der Spra-

che UML entsprechen. Da sowohl ein Meta-Metamodell als auch ein Metamodell immer auch Modelle sind, ist dieses Vorgehen sinnvoll und eben auch erlaubt.

Die *UML Infrastructure* definiert ein Paket mit dem Namen *Core*. Dieses Paket enthält die folgenden Unterpakete (vgl. Abbildung 3.6):

1. Das Paket *PrimitiveTypes* definiert die primitiven Datentypen *Integer*, *Boolean*, *String* und *UnlimitedNatural*.
2. Das Paket *Basic* erklärt die Grundbegriffe Klasse, Attribut, Operation und Paket.
3. Das Paket *Abstractions* führt weiterführende Konzepte wie Generalisierung, Instanz, Ausdruck (*Expressions*), Einschränkung (*Constraints*), Vielfachheit, Namensraum, Re-Definition, Sichtbarkeit und den allgemeinen Begriff der Relation ein.
4. Das Paket *Constructs* erweitert die Begriffswelt um das Grundkonzept Assoziation.

Die Benutzung der in den einzelnen Paketen erklärten Konzepte beruht auf Importbeziehungen zwischen diesen Paketen. Dabei wird der Mechanismus *Spezialisierung* genutzt, um einem bereits eingeführten Konzept weitere Eigenschaften hinzuzufügen.

Betrachten wir beispielsweise die Definition des Begriffs *Class* (vgl. Abbildung 3.7). Das erste Auftreten des Konzepts finden wir im Paket *Basic*. *Class* ist hier eine Spezialisierung der Klasse *Classifier* und diese wiederum eine Spezialisierung der Klasse *Namespace*. Die Klasse *Class* aggregiert die Klasse *Property*, die Attribute einer Klasse modelliert, unter dem Namen *ownedAttribute*. Das Konstrukt insgesamt erklärt, dass eine Klasse beliebig viele Eigenschaften haben kann, die unter der Bezeichnung *Attribut* geführt werden.

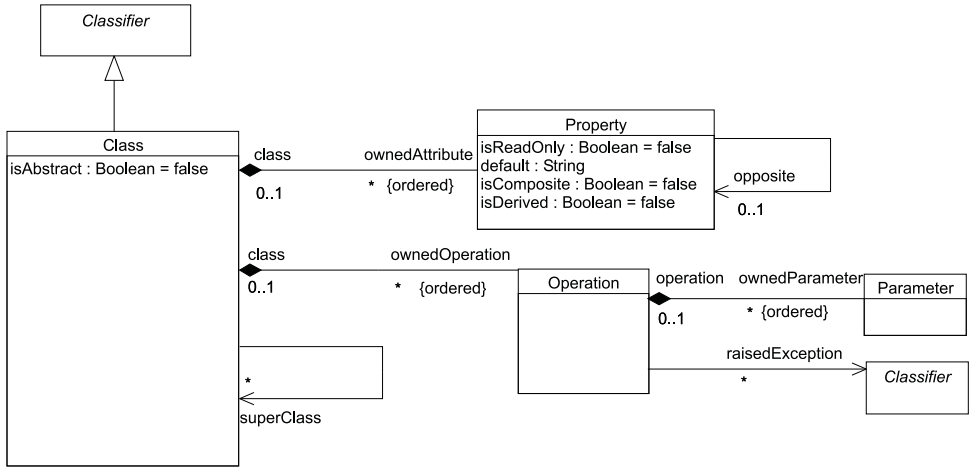

Abbildung 3.7: Class im Package Basic der UML Infrastructure

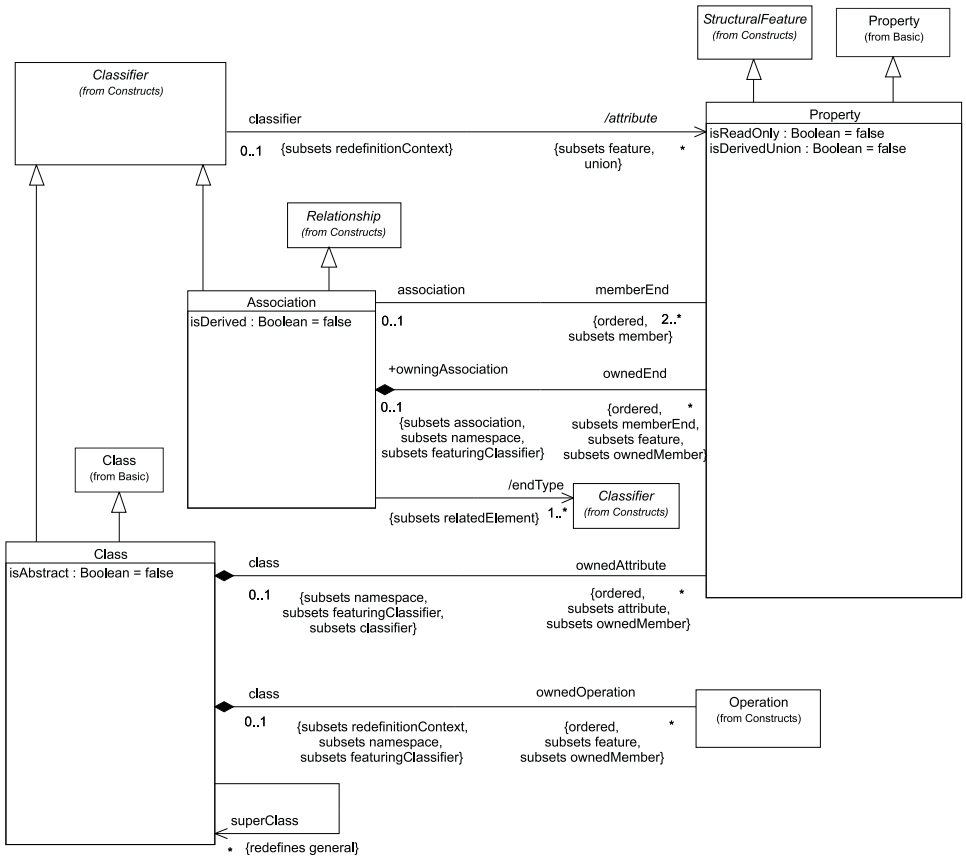

Abbildung 3.8: Metaklasse Class im Package Constructs der UML Infrastructure

Nun importiert das Paket *Constructs* das Paket *Basic*: Die Klasse *Class* ist also in *Constructs* sichtbar (vgl. Abbildung 3.8). Es ist zu erwarten, dass in Zusammenhang mit dem Konzept der Assoziation die Definition des Begriffs Klasse erweitert werden muss. Dazu wird im Kontext des Pakets *Constructs* zunächst eine weitere Klasse *Property* eingeführt, die die Klasse *Property* aus dem importierten Paket *Basic* spezialisiert. Das Konzept der Assoziation wird im Paket *Constructs* als Klasse *Association* erklärt, die *Classifier* und *Relationship* spezialisiert. Damit haben wir – neben Klasse – eine weitere Spezialisierung von *Classifier* kennen gelernt. Tatsächlich sind alle wesentlichen Eigenschaften von Klassen nicht für die Metaklasse *Class*, sondern allgemein für *Classifier* erklärt. Nun definiert die Klasse *Association* ihrerseits Assoziationen zur Klasse *Property*, die im Paket *Basic* nicht auftreten. Konkret werden die Assoziationsenden einer Assoziation als Eigenschaften der Assoziation aufgefasst, ganz ähnlich den Attributen einer Klasse. Diese Assoziationen erweitern die Klasse *Property* im Paket *Constructs* gegenüber der Klasse *Property* aus dem Paket *Basic*. Damit diese erweiterte *Property*-

Klasse auch dem Konzept der Klasse selbst bekannt ist, wird im Paket *Constructs* eine Spezialisierung der Klasse *Class* aus dem Paket *Basic* vorgenommen – ebenfalls mit dem Namen *Class*. Die Beziehung dieser Klasse zu *Property* mit dem Namen *ownedAttribute* ist eine Re-Definition der Eigenschaft *ownedAttribute* der Klasse *Class* aus dem Paket *Basic*. Die gemeinsame Metaklasse für die Begriffe Attribut und Assoziationsenden ist *Property*.

Ausgehend von der *UML Infrastructure* werden nach diesem Prinzip alle Konzepte der Sprache UML iterativ in Paketen eingeführt. Jedes dieser Pakete fasst eine bestimmte Menge von Features der Sprache zusammen. Ein Begriff – wie Klasse – tritt in einem Basispaket erstmalig auf und wird durch weitere Pakete, die dieses Paket importieren bzw. den *Merge*-Mechanismus nutzen, um spezielle in diesem Paket betrachteten Aspekte angereichert.

3.2.2 UML Superstructure

Wenn wir den Begriff Klasse durch die Pakete der *UML Superstructure* weiterverfolgen, so treffen wir auf ein Paket, das erklärt, dass mittels UML für Instanzen der Metaklasse *Class* (also Klassen eines UML-Modells) eine Verhaltensspezifikation angegeben werden kann. Das Vorgehen zur Erklärung dieses Sachverhalts im Metamodell entspricht dem bereits bekannten Muster: Es wird ein Paket *CommonBehaviors* in das Metamodell aufgenommen, in dem grundlegende Aspekte der Verhaltensmodellierung erklärt werden. In diesem Paket erfolgt eine Spezialisierung unserer Metaklasse *Class* aus *Constructs* (über mehrere Metaklassen, die alle den Namen *Class* tragen, aber aus unterschiedlichen Paketen des Metamodells stammen). Diese Spezialisierung definiert, dass Instanzen dieser Metaklasse gerade Verhaltensspezifikationen enthalten können.

In diesem Abschnitt wollen wir einige Ausflüge in die *UML Superstructure* unternehmen, um die bereits in Kapitel 2 eingeführten Sprachfeatures in der UML-Sprachdefinition aufzuspüren. Das Ziel ist es, ein Verständnis für das Metamodell zu erreichen, um in den nachfolgenden Kapiteln die Erklärung der weitergehenden Aspekte der UML 2 immer auch anhand des Metamodells nachvollziehen zu können.

Das Herzstück der *UML Superstructure* wird gebildet durch das Paket *UML::Classes::Kernel*, das das Paket *Constructs* aus der *UML Infrastructure* wiederverwendet: *Kernel* erklärt durch Paket-*Merge* die Begriffe aus *Constructs* zu Begriffen der Sprache UML. *Package Merge* nutzt einen komplexen Transformationsalgorithmus, um Elemente eines Pakets zu eigenen Elementen im Kontext eines anderen Pakets werden zu lassen. Man kann mittels *Package Merge* Konzepte, die in ihren Grundzügen in einem anderen Paket eingeführt wurden, in einem neuen Paket benutzen und erweitern, als wären sie in diesem erstmalig definiert. Die *UML 2 Superstructure* macht ausgiebig Gebrauch von dieser Technik, wie in Abbildung 3.9 illustriert.

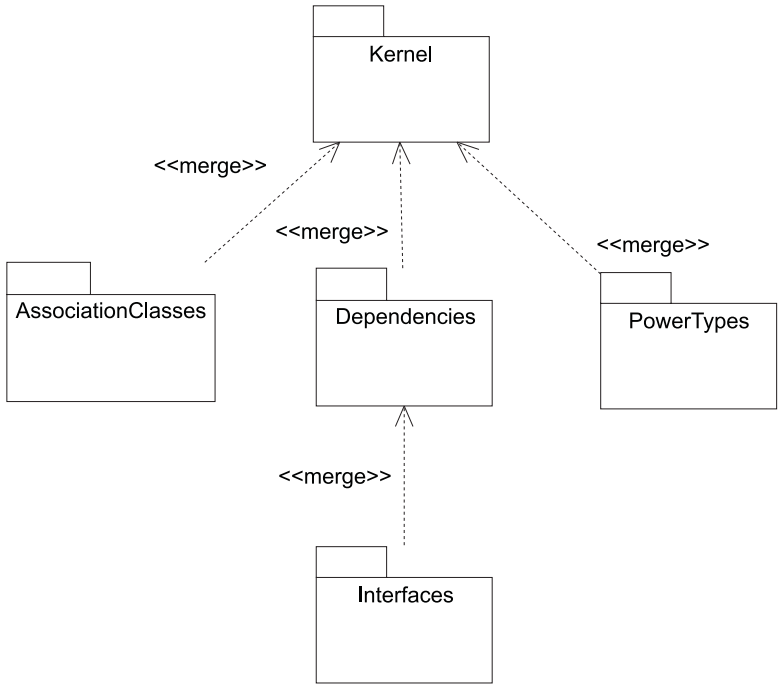

Abbildung 3.9: Nutzung von Package Merge in der UML Superstructure

Ähnlich der *UML Infrastructure* verwendet die *UML Superstructure* ein einheitliches Muster zur Erklärung von Sprachfeatures: Sie definiert immer eine Gruppe von zusammengehörigen Aspekten der Sprache in einem separaten Paket. Bei Spezialisierungen von Konzepten aus bereits existierenden Paketen wird mittels *Package Merge* das jeweilige Konzept in einem Paket für weitere Aspekte zunächst sichtbar gemacht und anschließend mittels Erweiterungen bzw. Re-Definitionen angereichert.

Ein UML-Werkzeug, das das UML-Metamodell als interne Repräsentation eines Modells einsetzt, weiß aus dem jeweiligen Kontext des Auftauchens eines Modellelements stets, welche Spezialisierung eines Konzepts gemeint ist. Wird beispielsweise in einem Modell zu einer Klasse eine Verhaltensspezifikation hinzugefügt, so ist durch das Werkzeug die Metaklasse *Class* aus dem Paket *UML::CommonBehaviors* zu instanzieren.

In den bisherigen Abschnitten ist hin und wieder das Konzept des Besitzes bzw. Enthaltenseins von Elementen im Kontext eines Elements aufgetaucht. Der Aspekt »Besitz anderer Elemente« ist elementar in UML. So können Elemente in Paketen auftreten oder Klassen können Attribute besitzen.

Eigentum im Metamodell

Wir wollen den Aspekt »Besitz« im Metamodell von UML genauer verfolgen. Zunächst stellen wir fest (vgl. Abbildung 3.10), dass die Metaklasse *Element* im Paket *UML::Classes::Kernel* Elemente des gleichen Typs komponiert (besitzt).

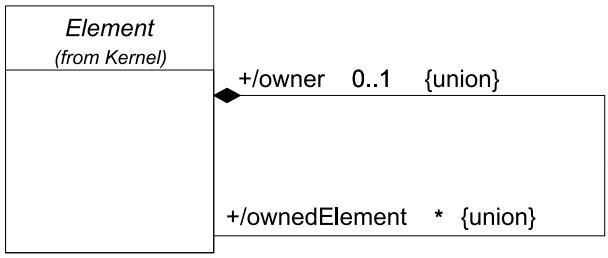

Abbildung 3.10: Metaklasse UML::Classes::Element

Die Enden der Komposition sind *owner* und *ownedElement*. Für das Ende *owner* ist eine Vielfachheit von *[0..1]* definiert. Das bedeutet, dass jedes Element Eigentum höchstens eines anderen Elements ist. Ein Element kann mittels *ownedElement* beliebig viele Elemente besitzen. *ownedElement* ist eine strikte Vereinigung (Eigenschaft *strict union*). Dies heißt, dass die Elemente, die ein Eigentümer besitzen kann, ausschließlich (*strict*) in Spezialisierungen von *Element* durch Re-Definitionen des Endes *ownedElement* mittels Angabe von Untermengen (*subsets*) bestimmt werden. Das Ende *owner* ist – wie *ownedElement* – als abgeleitet (*derived*) gekennzeichnet, geht also ebenfalls aus Re-Definitionen in Spezialisierungen von Element hervor.

Diese Konstruktion wird erstmals durch die Beziehung zwischen *Element* und *Comment* spezialisiert (vgl. Abbildung 3.11). Die Metaklasse *Comment* modelliert die Möglichkeit der Kommentierung eines Elements in einem Modell. *Comment* ist eine Spezialisierung von *Element*, ein Kommentar kann also von einer Instanz von *Element* besessen werden. Damit gehört ein Kommentar zur Menge *ownedElement* von Element. Dieser Sachverhalt wird durch eine Assoziation zwischen *Element* und *Comment* ausgedrückt, in der das Ende *owningElement* als Untermenge von *owner* und das Ende *ownedComment* als Untermenge von *ownedElement* definiert werden.

In Kapitel 2.4 wurde auf die notwendige Voraussetzung für Untermengenbeziehungen an Assoziationsenden eingegangen: Die Klasse am ursprünglichen Assoziationsende, das die Menge (*union*) definiert, und die Klasse an demjenigen Assoziationsende, das die Untermenge der Menge erklärt, müssen konform sein, d.h. in einer Generalisierungsbeziehung stehen. Diese Voraussetzung ist hier erfüllt, da *Comment* die Metaklasse *Element* spezialisiert.

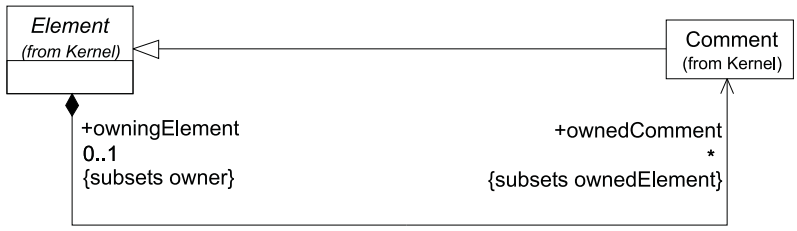

Abbildung 3.11: Ownership-Beziehung zwischen den Metaklassen Element und Comment

Einschränkungen von Metamodellaussagen

Die Aussagen, die mit den Mitteln der UML 2 getätigt werden können, reichen in Metamodellen oftmals noch nicht aus, um alle semantischen Erklärungen vorzunehmen. Beispielsweise würde man erwarten, dass ein Element sich nicht selbst besitzen darf, und zwar weder direkt noch indirekt. Es ist nicht möglich, diese Aussage mit dem Vokabular der *UML Infrastructure* zu treffen. Um Aussagen dieser Art machen zu können, definiert man in Metamodellen häufig Einschränkungen (*Constraints*) und benutzt dazu die Sprache *Object Constraint Language* (OCL, [OCL 03]). Ein *Constraint* ist ein Ausdruck, der an einen Kontext geknüpft wird und dessen Typ *Boolean* ist, d.h. der Audruckswert lässt sich zu wahr oder falsch berechnen. Um die obige Forderung des Verbots von Zyklen in Besitz-Relationen zu treffen, gibt es im Metamodell ein *Constraint*, dessen Kontext die Metaklasse *Element* ist:

```
not self.allOwnedElements()->includes(self)
```

In die natürliche Sprache übersetzt, erklärt das *Constraint* Folgendes:

1. Entnehme aus der Menge aller Instanzen des Kontexts (Klasse *Element*) eine beliebige Instanz (*self*).
2. Berechne die Menge aller Instanzen, die mittels des Endes *ownedElement* erreichbar sind (*allOwnedElements*).
3. Überprüfe, ob *self* in dieser Liste auftaucht.
4. Negiere das Ergebnis dieser Überprüfung.

Ergibt die Auswertung des *Constraints* das Ergebnis wahr, so ist das UML-Modell gültig!

allOwnedElements ist kein Standardausdruck der Sprache OCL, sondern eine Operation der Klasse *Element*:

```
Element::allOwnedElements(): Set(Element);
```

Diese Operation kann mit OCL-Sprachkonstrukten folgendermaßen beschrieben werden:

```
allOwnedElements =
      ownedElement->union
            (ownedElement->collect(e | e.allOwnedElements()))
```

Das UML-Metamodell macht regen Gebrauch von der Technik, Regeln der statischen Semantik mit der Sprache OCL zu beschreiben. Zum schnellen Verständnis sind alle derartigen Regeln in den UML-Standarddokumenten sowohl mit englischem Text (zur Beschreibung der Aussage selbst) als auch mit Mitteln der OCL erklärt.

Eigentum und Namensräume

Wenn wir die Eigentumsregelung zwischen Spezialisierungen der Metaklasse *Element* weiterverfolgen, so begegnet uns die bereits erwähnte Metaklasse *Namespace*. Diese Klasse definiert erwartungsgemäß Untermengen von *ownedElement* der indirekten Basisklasse *Element* (vgl. Abbildung 3.12).

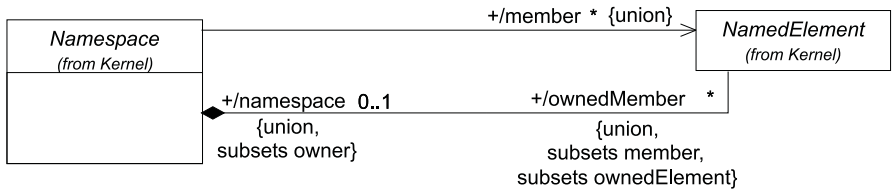

Abbildung 3.12: Metaklasse Namespace erklärt Untermengen von ownedMember

Dazu wird zunächst eine Assoziation zur Metaklasse *NamedElement* mit dem Ende *member* definiert. Instanzen von *Namespace* kennen also via *member* alle beinhalteten Elemente. *member* selbst ist wiederum eine *Derived Union* und wird in den Spezialisierungen von *Namespace* durch Definition von Untermengen erklärt. *member* ist jedoch keine Untermenge von *ownedElement*. Damit wird im Metamodell dem Umstand Rechnung getragen, dass ein Namensraum **A** Elemente eines anderen Namensraums **B** importieren kann, diese Elemente sind somit in dem Namensraum **A** sichtbar. Natürlich gehören diese Elemente aber weiterhin dem Namensraum **B**, aus dem sie importiert wurden, sind aber eben auch zu *member* von **A** gehörig.

Diejenigen Elemente, die einem Namensraum gehören, werden unter *ownedMember* zusammengefasst. *ownedMember* erklärt eine Menge, die durch Spezialisierungen von *Namespace* und *NamedElement* mittels Untermengen definiert wird. Diese Menge ist gerade eine Untermenge von *member*: Alle Elemente, die zu einem Namensraum gehören, sind Teil der Menge aller Elemente, die in einem Namensraum sichtbar sind, über diese Menge hinaus können beispielsweise Kommentare ebenfalls im Besitz des Namensraums sein.

Classifier und Re-Definition

Ein abschließender Ausflug soll das Konzept der Re-Definierbarkeit von Attributen und Operationen anhand des Metamodells verdeutlichen. Dazu wollen wir zunächst den Hinweis aufgreifen, dass die wesentlichen Eigenschaften, die in Kapitel 2 in Zusammenhang mit dem Klassenbegriff eingeführt wurden, tatsächlich für die Metaklasse *Classifier* erklärt sind. Zu diesen Eigenschaften gehört insbesondere die Spezialisierbarkeit: Instanzen von *Classifier* dürfen in Generalisierungsbeziehung zueinander stehen. Eine der Spezialisierungen der Metaklasse *Classifier* ist erwartungsgemäß die Metaklasse *Class*. Instanzen von *Class* sind die uns bekannten Klassen in einem Modell.

Welche weiteren Ausprägungen von *Classifier* gibt es? Wenn wir ausschließlich in das Metamodellpaket *UML::Classes::Kernel* schauen, so ergibt sich die Darstellung in Abbildung 3.13.

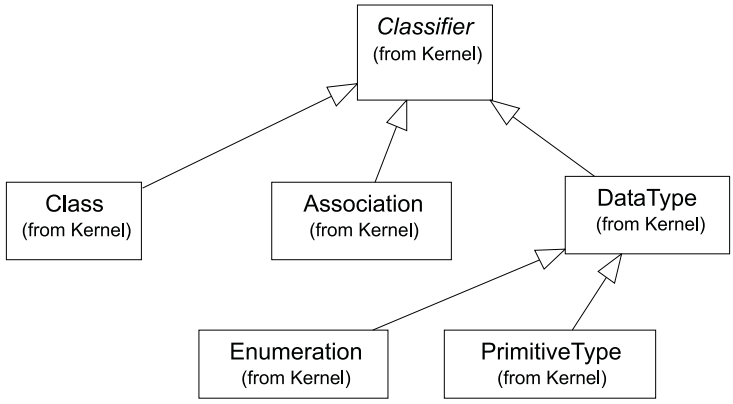

Abbildung 3.13: Classifier-Spezialisierungen im Paket UML::Classes::Kernel

Neben dem uns bekannten Konzept der Klasse sind Assoziationen und Datentypen ebenfalls Spezialisierungen von *Classifier*. Dies bedeutet, dass die Eigenschaft Spezialisierbarkeit ebenso für diese Konzepte gilt, man darf in UML-Modellen also Generalisierungsbeziehungen zwischen Assoziationen angeben! Wir werden diesen Aspekt bei der Erläuterung weiterer Aspekte der Sprache wieder aufgreifen.

Nicht alle Eigenschaften, die für *Classifier* erklärt sind, gelten uneingeschränkt für deren Spezialisierungen. Im Metamodell werden diese Einschränkungen mit Constraints im Kontext der jeweiligen Spezialisierungen beschrieben. Schaut man über das Paket *UML::Classes::Kernel* hinaus, so finden sich viele weitere Spezialisierungen von *Classifier* in anderen Paketen, die uns ebenfalls in den weiteren Kapiteln begegnen werden. Die Eigenschaft Re-Definierbarkeit ist im Metamodell durch die Klasse *RedefinableElement* erklärt, einer Spezialisierung der Klasse *NamedElement*. Instanzen

von Spezialisierungen der Klasse *RedefinableElement* treten im Kontext von *Classifiers* auf und dürfen – bei Vorliegen bestimmter Voraussetzungen – re-definiert werden. Die Kontexte, in denen ein Element definiert bzw. re-definiert wurde, sind mit *redefinitionContext* referenziert. Die Eigenschaft *isLeaf* gibt dabei an, ob ein Element weiterführend spezialisiert werden darf oder nicht: Wenn ein Modellierer entscheidet, dass ein bestimmtes Element nicht re-definiert werden soll, so wird dies durch Belegen des Attributs mit dem Wert *true* angezeigt. Das Ende *redefinedElement* verweist auf das »originale« Element, das re-definiert wird (vgl. Abbildung 3.14).

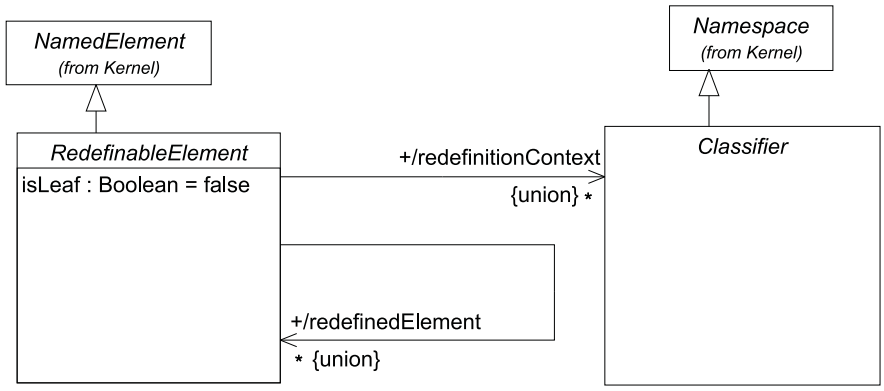

Abbildung 3.14: Metaklasse RedefinableElement

Eine Voraussetzung für Re-Definierbarkeit ist, dass sich ein re-definiertes Element im Kontext eines *Classifiers* befinden muss, der eine (direkte oder indirekte) Spezialisierung desjenigen *Classifiers* ist, der das »originale« Element definiert.

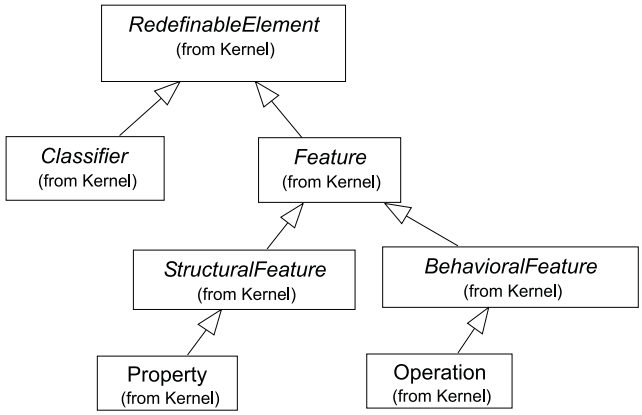

Abbildung 3.15: Spezialisierungen der Klasse RedefinableElement

Wir wollen nun diejenigen Metaklassen kennen lernen, die die abstrakte Klasse *RedefinableElement* spezialisieren, deren Instanzen in Modellen also re-definiert werden dürfen. Zunächst finden wir erwartungsgemäß die Metaklassen *Operation* und *Property* als Spezialisierungen vor. Instanzen der Klasse *Operation* treten in Modellen als Operationen von Klassen auf, Instanzen von *Property* begegnen uns beispielsweise als Attribute von Klassen. Entsprechend den Ausführungen in Kapitel 2 dürfen diese Elemente re-definiert werden (vgl. Abbildung 3.15).

Wir finden aber überraschenderweise auch die Klasse *Classifier* als Spezialisierung von *RedefinableElement*. Es gibt also Spezialisierungen von *Classifier*, die im Kontext eines *Classifiers* spezialisiert werden dürfen. Wann benutzt man eine solche Konstruktion? Um diese Frage zu beantworten, müssen wir über das Paket *UML::Classes::Kernel* hinausschauen und einen Vorgriff auf die UML-Konstrukte zur Verhaltensspezifikation wagen. Mit UML ist es möglich, dynamische Aspekte beispielsweise einer Klasse zu beschreiben. So können die Veränderung von Attributbelegungen oder das Aufrufen von Operationen an anderen Objekten für Objekte einer Klasse dargestellt sein. Eine Verhaltensspezifikation wird immer im Kontext eines *Classifiers* vorgenommen, also beispielsweise in einer Klasse. Im Sinne der Objektorientierung erwartet man, dass eine Verhaltensspezifikation eines generelleren *Classifiers* im Kontext einer Spezialisierung dieses *Classifiers* re-definiert werden kann – der Aspekt Verhalten sollte also im Metamodell in Bezug zu *RedefinableElement* stehen.

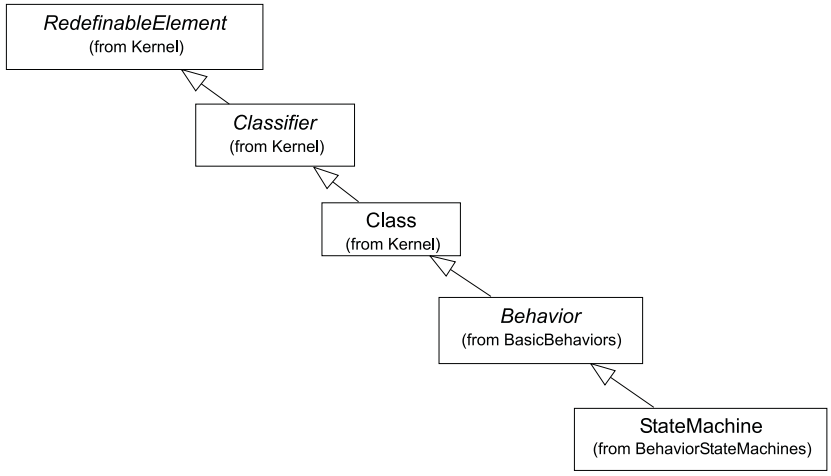

Abbildung 3.16: Metaklasse Behavior als Spezialisierung von Classifier

Verhalten wird im Metamodell (vgl. Abbildung 3.16) durch die Klasse *Behavior* repräsentiert, Spezialisierungen dieser Klasse, beispielsweise das Konzept *Zustandsmaschine* (*State Machine*), können in Modellen eingesetzt werden, um Aussagen über Verhaltensaspekte von Klassen zu formulieren. Die Metaklasse *Behavior* ist eine Spezialisierung

von *Class* und damit indirekt von *Classifier*. Durch diese Spezialisierungshierarchie (und eine Reihe von zusätzlichen Constraints) ist also festgelegt, dass eine originale Verhaltensspezifikation im Kontext eines spezialisierenden Classifiers re-definiert werden darf. Es gibt also »sinnvolle« Spezialisierungen der Klasse *RedefinableElement*, die indirekt Spezialisierungen von *Classifier* sind.

Wir möchten unsere kurze Exkursion in die Form der Sprachbeschreibung der UML an dieser Stelle beenden. In den folgenden Kapiteln werden wir die eingeführten Sprachfeatures immer wieder anhand des Metamodells verfolgen.

3.2.3 UML Infrastructure und Meta Object Facility

Warum hat man dann bei der Sprachdefinition die *UML Infrastructure* (Paket *Core*) und die *UML Superstructure* (Paket *UML*) so deutlich getrennt? Immerhin gab es zwei separate Normierungsprozesse in der Object Management Group!

Die Antwort auf obige Frage findet sich in der Geschichte der Modellierungsstandards der OMG. Gemeinsam mit der Sprache UML wurde die Norm *Meta Object Facility (MOF)* publiziert und in den Folgejahren weiterentwickelt. MOF definiert das MOF-Modell, das wiederum zur Erklärung von Sprachen eingesetzt wurde. Das MOF-Modell beschreibt also ein Meta-Metamodell. Dieses Meta-Metamodell wurde eingesetzt, um einige Sprachstandards in der OMG zu definieren, insbesondere die Komponenten- und Schnittstellenbeschreibungssprache CORBA IDL (*Common Object Request Broker Interface Definition Language*, [CORBA 02]) sowie das Metamodell der Sprache UML selbst.

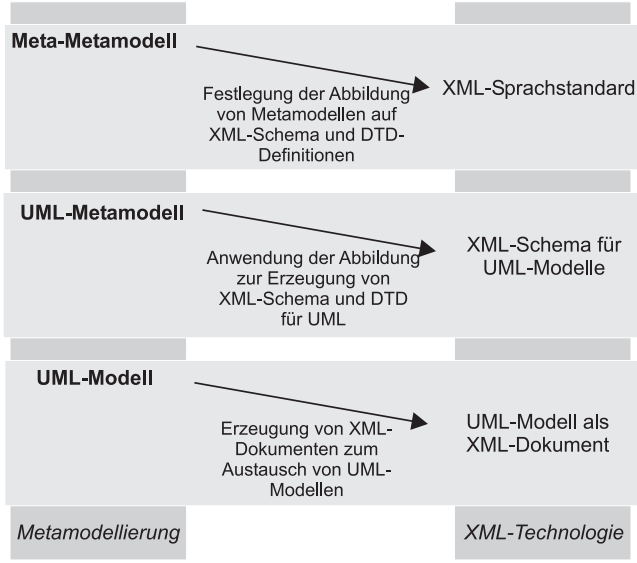

Abbildung 3.17: Metamodellierung und XML Technologie

Als Bestandteil der Norm MOF 1.x und ihrer Nachfolgeversionen sowie auch in separaten Normen – sind mehrere Technologieabbildungen definiert worden. Die bekannteste derartige Abbildung ist sicherlich *XML Metadata Interchange* (XMI, [XMI 03]). XMI erklärt, wie Instanzen der Elemente des MOF-Modells (also Elemente eines Metamodells) auf XML-Dokumenttyp- bzw. -Schemadefinitionen abgebildet werden (vgl. Abbildung 3.17). Damit wird eine Formatvorlage definiert, die man nutzen kann, um Instanzen dieses Metamodells durch XML-Dokumente auszudrücken. Wendet man die Abbildungsregeln für Elemente eines Metamodells auf das UML-Metamodell an, so erhält man eine Dokumenttypdefinition, die ein Format zum Austausch von UML-Modellen erklärt – die so genannte UML DTD. Die Technologieabbildung von MOF auf XML bildet damit die Grundlage, um UML-Modelle in standardisierter Form auszutauschen – in der Regel zwischen Werkzeugen unterschiedlicher Hersteller.

Eine weitere, nicht so stark verbreitete, aber nicht minder mächtige Gruppe derartiger Technologieabbildungen erzeugt aus MOF-Metamodellen Schnittstellenbeschreibungen von Softwarekomponenten. In diese Gruppe gehören die Abbildungsregeln zur Erzeugung von Java Interfaces (bekannt als Java Metadata Interface, JMI [JMI 02]) und von CORBA Interfaces (bekannt als MOF IDL, Teil der Norm MOF 1.x). Im Wesentlichen dienen diese Softwarekomponenten ebenfalls der Speicherung von Instanzen der Quell-Metamodelle (also der Modelle), jedoch nicht als XML-Dokumente, wie im Falle von XMI, sondern beispielsweise als Einträge in speziellen Datenbanken (so genannte *Repositories*). Wendet man diese Technologie auf das UML-Metamodell an, so erhält man die Schnittstellen eines *Repository* zur Speicherung von UML-Modellen. Stellt man MOF und CORBA in Beziehung zueinander, so ergibt sich ein der Relation zwischen MOF und XML sehr ähnliches Bild (vgl. Abbildung 3.18).

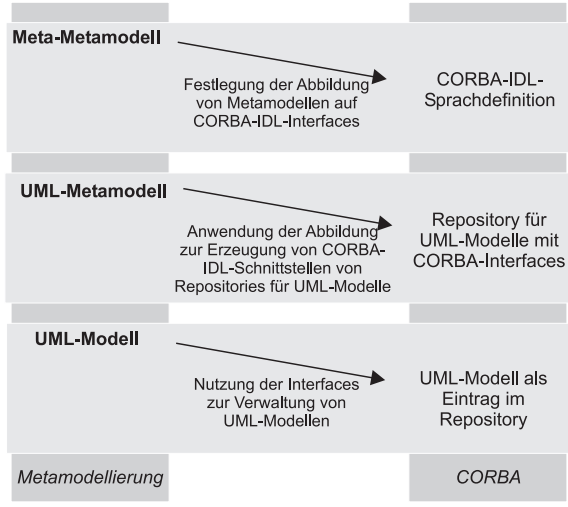

Abbildung 3.18: Metamodellierung und CORBA

Zusammengefasst führt die Verbindung zwischen MOF und UML dazu, dass man UML-Modelle mittels XMI als XML-Dokumente repräsentieren und austauschen kann und dass mittels weiterer MOF-Technologieabbildungen Schnittstellen eines UML-*Repositories* zur Verfügung stehen.

Die Schnittstellen der *Repositories* erlauben eine iterative und selektive Manipulation der UML-Modelle, man kann beispielsweise mittels Operationen neue Instanzen einer Metaklasse erzeugen, diese mit anderen Metaklasseninstanzen verknüpfen und die Korrektheit von Modellen überprüfen. Wir werden diese Technologien nochmals in Zusammenhang mit über die UML hinausgehenden Modellierungstechniken aufgreifen.

Bis auf wenige, leider jedoch essentielle Unterschiede wurde bereits zu Beginn der Normierung der Kern der Sprachversionen UML 1.x als Grundlage für das MOF-Modell genutzt. Mit der Aufnahme ders Normierungsprozesses für die Version 2.0 beider Standards wurde das Ziel verfolgt, das MOF-Modell und den Kern der UML vollständig zu vereinigen. Dies sollte auf der Grundlage einer einheitlichen Menge von Basiskonzepten der UML und des MOF-Modells geschehen, die in einem separaten Normierungsprozess zu finden war. Aus diesem Normierungsprozess ist die *UML Infrastructure* hervorgegangen, sie ist die Grundlage der UML-Sprachdefinition – und des MOF-Modells in der Version 2.0. Dieses neue MOF-Modell wurde in einem weiteren Normierungsprozess [MOF 03] festgelegt und unterteilt sich in *Essential MOF* (EMOF) und *Complete MOF* (CMOF). EMOF ist ausschließlich durch Importieren des Pakets *Core::Basic* aus der *UML Infrastructure*, CMOF durch Importieren des Pakets *Core::Constructs* und Erweiterung um MOF-spezifische Konstrukte definiert. Die Definition der Technologieabbildungen nach XML, CORBA IDL und Java ist zum gegenwärtigen Zeitpunkt noch nicht abgeschlossen.

Wir möchten an dieser Stelle unseren Exkurs in die Meta-Modellierung beenden und unsere gewonnenen Erkenntnisse über die Technologie zur Definition der Sprache UML 2 einsetzen, um weitere Sprachkonstrukte zu erforschen.