Entwicklung eines graphischen Editors zur Unterstützung eines modellgetriebenen Softwareentwicklungsprozesses

– Diplomarbeit –

Enrico Schnepel

Eingereicht an der

Fachhochschule für
Technik und Wirtschaft Berlin

Fachbereich IV
Angewandte Informatik

am
11.08.2008

von
Enrico Schnepel
Matrikelnummer s0511445

betreut durch
Prof. Herrman Heßling (FHTW Berlin)
und
Dipl.Inf. Stefan Hänsgen (Gentleware AG)

Abstract

„base“ ist eine bei der „b+m Informatik GmbH Berlin“ intern verwendete modellgetriebenen Softwareentwicklungsumgebung. Für die Modellbearbeitung in „base“ wurden bisher Baum-basierte Editoren genutzt, die auf dem Eclipse Modelling Framework (EMF) aufbauen. Insbesondere für das `dataflow`-Modell ist eine Baumdarstellung nicht benutzerfreundlich, da das Modell konzeptuell einen Graphen im mathematischen Sinn repräsentiert. Ziel dieser Diplomarbeit war es den Entwicklungsprozess in „base“ wesentlich zu optimieren. Deshalb wurde für die Bearbeitung von `dataflow`-Modelle ein graphischer Editor entwickelt, der auf dem Graphical Modelling Framework (GMF) basiert. Das den `dataflow`-Modellen zugrunde liegende Metamodell enthält 20 Elementtypen, die direkt auf der Zeichenfläche des Editors platziert werden können. Die Komplexität der im GMF genutzten Modelle steigt jedoch mit den verwendeten Elementtypen überproportional an und ist damit nur noch schwer zu handhaben. Deshalb wurde das „GenGMF“ entwickelt, statt einen graphischen Editor direkt mit dem GMF zu implementieren. „GenGMF“ ist eine Domain Specific Language (DSL) mit einer dazu passenden generischen Modell-zu-Modell-Transformation zur automatischen Erstellung der GMF-Modelle. Mit diesem neu entwickelten Ansatz zur Erstellung graphischer Editoren wurde anschließend der `dataflow`-Editor modelliert. Durch die Nutzung von „GenGMF“ für die Entwicklung des `dataflow`-Editors ermöglichte die Erstellung eines „GenGMF“-Modells mit 72% weniger Elementen als die generierten GMF-Modelle enthalten. Zusätzlich entfällt die manuelle Bearbeitung der GMF-Modelle. Die Darstellung im entwickelten graphischen `dataflow`-Editor erfolgt nun einer benutzerfreundlichen Art und Weise.

„base“ is a model-driven software development environment for internal use by „b+m Informatik GmbH Berlin“. For editing models in „base“ a tree-based editor has been used in the past, which builds on the Eclipse Modelling Framework (EMF). A tree representation in particular is difficult to manage for the dataflow model, because conceptionally the model represents a graph in the mathematical sense of the word. The objective of this thesis was essentially optimizing the developmental process in „base“. To this end, a graphic editor for `dataflow` models was developed on the basis of the Graphical Modelling Framework (GMF). The meta-model, on which the `dataflow` models are based, contains 20 element types, which can be placed directly in the editor's drawing area. The complexity of the models used in GMF increases over proportionally though with the number of element types and is consequently difficult to administrate. This was the reason for developing „GenGMF“ instead of implementing a graphic editor with GMF itself. „GenGMF“ is a Domain Specific Language with an appropriate generic model-to-model transformation for the automatic construction of GMF models. The dataflow editor was modeled using this new development for the construction of graphic editors. Using „GenGMF“ for development of the `dataflow` editor allowed construction of a „GenGMF“ model requiring 72% less model elements than generated GMF models. In addition, it did not have to be manually adjusted. The result is ease of use for presentation in the graphic `dataflow` editor.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

In dieser Arbeit steht die Entwicklung und Implementierung eines graphischen Editors im Mittelpunkt. Die Ausgangsbasis bilden Baum-basierte Editoren auf der Basis des Eclipse Modelling Framework (EMF). Sie werden bereits in einem internen Projekt „base“ in der „b+m Informatik GmbH Berlin“ verwendet, um modellgetrieben Software vollständig zu generieren. In der Praxis hat sich herausgestellt, dass im Allgemeinen Baum-basierte Editoren für Modelle auf der Basis eines Graphen im mathematischen Sinn ungeeignet sind.

Es wird angenommen, dass die Verwendung eines graphischen Editors für den komplexen Modelltyp `dataflow` folgende Vorteile hat: Zum Einen wird die Verständlichkeit eines Modells wesentlich verbessert, wenn Objekte auch durch adäquate Elemente in der graphischen Oberfläche repräsentiert werden. Andererseits erleichtert ein für den Entwickler verständliches Modell die Bearbeitung desselben sowie die Kommunikation im Team, da eine gemeinsame Vorstellung über das Modell gebildet werden kann. Im Ergebnis wird nicht nur die Entwicklungszeit für ein Modell reduziert, sondern auch der gesamte Softwareentwicklungsprozess beschleunigt. Dies bringt Vorteile in der modernen Softwareentwicklung, da Projekte oft mit sehr knappen Zeitvorgaben effizient bewältigt werden müssen.

Erfahrungen im Bereich Model Driven Software Development (MDSD) (de.: modellgetriebener Softwareentwicklung) sammelte der Verfasser während seines Praktikums im Wintersemester 2007 bei der „b+m Informatik GmbH Berlin“ im Projekt „base“. Die Aufgabe bestand u. a. in der Entwicklung von für „base“ verwendeten Generatoren auf der Basis der Generatorspache Xpand der Software openArchitectureWare (oAW).

Fachlich wird diese Diplomarbeit von der „b+m Informatik GmbH Berlin“ sowie technisch von der „Gentleware AG“ in Hamburg betreut. Dies sind zwei von fünf Tochtergesellschaften des Konzerns „b+m Informatik AG“ in Melsdorf bei Kiel, dessen Organisationsstruktur im Anhang A 1 beschrieben ist. Eines der Geschäftsfelder des Konzerns ist die MDSD. Die „Gentleware AG“ hat sehr viel Erfahrung im Bau von graphischen Editoren auf der Basis des Eclipse-basierten Graphical Modelling Framework (GMF). Auf universitärer Seite wird die Diplomarbeit von Prof. Heßling im Fachbereich IV (Wirtschaftswissenschaften II) an der Fachhochschule für Technik und Wirtschaft (FHTW) betreut.

Die fachliche Auseinandersetzung mit den Grundlagen der modellgetriebenen Softwareentwicklung sowie den darauf aufbauenden Standards und Produkte erfolgt im ersten Teil der Arbeit. In den beiden darauf folgenden Teilen GMF und „base“ werden die technischen Grundlagen für Entwicklung des `dataflow`-Editors erläutert. Eine Analyse des Baum-basierten `dataflow`-Editors sowie ein daraus abgeleitetes Pflichtenheft für den neuen Editor enthält das Kapitel 5. Anschließend wird das im Rahmen dieser Diplomarbeit entwickelte Framework „GenGMF“ beschrieben, mit dem graphische Editoren auf der Basis des GMF erstellt werden können. Am Ende der Arbeit wird die Implementierung des `dataflow`-Editors auf der Basis des „GenGMF“ vorgestellt sowie das Erreichte zusammengefasst und ein Ausblick auf zukünftige Entwicklungen gegeben.

Zur besseren Verständlichkeit der komplexen Materie der MDSD werden Analogien zu fachfremden Themen sowie Beispiele und Zitate gesondert mit einem grauen Kasten gekennzeichnet.

KAPITEL 2

Grundlagen

Nicht zuletzt seit dem Aufkommen der Unified Modelling Language (UML) werden immer häufiger abstrakte Modelle in der Softwareentwicklung genutzt. Zum Beispiel kann mit den UML-Modellen in einer vereinheitlichten Art und Weise sowie unabhängig von anderen Bestandteilen ein eventuell Problem-spezifischer Aspekt eines komplexen Systems strukturiert und vollständig dokumentiert, visualisiert oder bearbeitet werden.

Erst in letzter Zeit wird der Fokus immer mehr auf die vollständige Generierung von Software aus Modellen gesetzt. Werden Modelle verwendet, um aus ihnen automatisch und wiederholt Artefakte zu generieren, welche die modellierten Bestandteile einer Software nach definierten Regeln abdecken, wird von der modellgetriebenen Softwareentwicklung gesprochen. Sie ist letztendlich nur eine weitere Abstraktion in der Entwicklung von Programmiersprachen. Smith und Stotts schreiben hierzu:

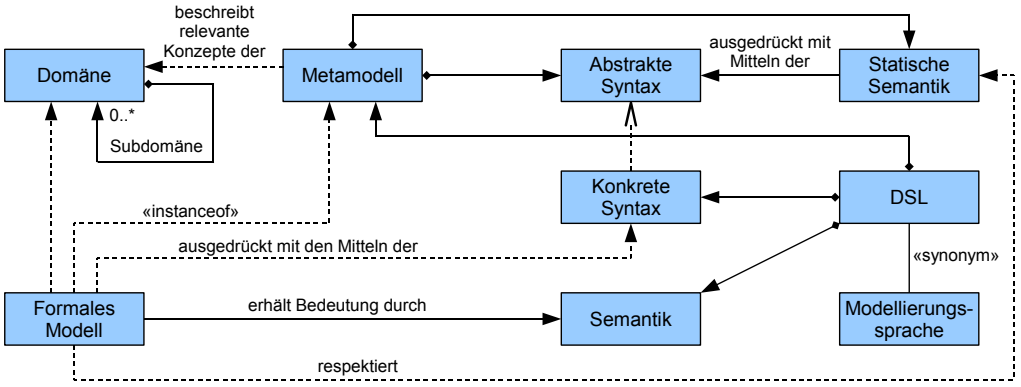
> **Zitat** (aus dem Englischen)**:** Abstraktion
> Die Geschichte der Programmierung ist ein Beispiel der hierarchischen Abstraktion. In jeder Generation produzieren die Entwickler von Sprachen Konstrukte für Dinge, die sie aus der vorherigen Generation gelernt haben. Architekten benutzen diese Konstrukte, um komplexere und leistungsfähigere Abstraktionen zu entwickeln.
>  [Smi02, S. 2]

Das Kapitel 2 unterteilt sich in vier Abschnitte. Zunächst werden häufig in der MDSD verwendete Begriffe definiert, um danach auf für diese Arbeit relevante Standards und Produkte einzugehen. Der Abschnitt 2.3 stellt Lösungen vor, mit denen graphische Editoren für Eclipse entwickelt werden können. Abschließend wird auf die modellgetriebene Softwareentwicklung eingegangen.

2.1 Begriffsdefinitionen

In diesem Abschnitt werden die für das Verständnis dieser Arbeit wichtigen Begriffe definiert und erläutert. Ausgehend von einer Definition für den Begriff Instanz wird über den Begriff Modell das Abstraktionsnivau bis hin zum Begriff Meta-Metamodell erhöht. Anschließend wird auf den Begriff Domäne sowie weitere Begriffe aus dessen Umfeld eingegangen. Zum Schluss werden verschiedene Varianten von Transformationen sowie die Aspekt-orientierte Programmierung definiert.

Viele der Begriffe sowie deren Zusammenhänge zueinander sind in der Abbildung 2.1 dargestellt.



([Sta07, S. 28])

Abbildung 2.1: Zusammenhänge der verwendeten Begrifflichkeiten

2.1.1 Instanz und formales Modell

Eine Instanz wird oft durch ein real (z. B. Kunde) oder virtuell (z. B. Konto) existierendes Objekt repräsentiert.

Analogie: Instanz
In dieser Diplomarbeit werden Analogien zwischen Themen aus der Bauwirtschaft und der modellgetriebenen Softwareentwicklung gezogen. Hier entspricht das Haus einer Instanz.

In einem Modell werden Informationen gespeichert. Modelle können beliebig komplex sein, sie entsprechen jedoch immer einer formal fest definierten Struktur. Im Modell existieren Angaben, welche Informationen die Instanzen enthalten, wie z. B. Name und Schuhgröße oder Kontonummer und Saldo. „Formal“ bedeutet, dass das Modell den modellierten Sachverhalt vollständig beschreibt. Es müssen klare Regeln definiert sein, was in einem Modell enthalten sein soll und was nicht.

Modelle können u. a. in präskriptive und deskriptive Modelle klassifiziert werden. Ein deskriptives Modell ist vom Original (der Instanz) abhängig, d.h. das Original wird nicht vom Modell beeinflusst. Sobald das Modell das Original beeinflusst, handelt es sich i. d. R. um ein präskriptives Modell. (vgl.: [Lud02, S. 12])

Analogie: Modell
Das präskriptive Modell eines Hauses entspricht – im Kontext der modellgetriebenen Softwareentwicklung – der Bauzeichnung. Ein weiteres Modell – auf einer anderen Abstraktionsebene – ist die Teileliste eines Fertigbauhauses.
Ein deskriptives Modell ist z. B. das Foto eines Hauses.
Die von einem Architekten gefertigte Miniaturvariante eines Hauses, im Bauwesen als Modell bezeichnet, ist hier eine weitere Instanz, bei deren Erstellung weniger und andere Materialen verwendet werden.

2.1.2 Metamodell

Metamodelle beschreiben die formale Struktur von Modellen. Damit sind die Modelle Instanzen von Metamodellen. In den Modellen werden die Informationen in den im Metamodell definierten

Attributen und Referenzen gespeichert.

Die in einem Metamodell definierten Namen der Strukturelemente haben meist ihren fachlichen Ursprung in dem zu lösenden Problem. Aus technischen Gründen sind zusätzliche Elemente notwendig, um mit dem Metamodell Modelle erstellen zu können. So wird z. B. ein Wurzelelement benötigt, das alle anderen Elemente des Modells enthält.

Analogie: Metamodell
Das Metamodell einer Bauzeichnung entspricht einer DIN-Vorschrift zum Anfertigen von Bauzeichnungen. (z. B. DIN 5, 201, 919, 1356 und 1986 sowie ISO 2594, 4157 und 7518)

2.1.3 Meta-Metamodell

Ein Metamodell ist ebenfalls ein Modell und somit folgt auch die Definition eines Metamodells einer festgelegten Struktur. Meta-Metamodelle definieren, aus welchen syntaktischen Elementen die Metamodelle aufgebaut sind.

Analogie: Meta-Metamodell
Theoretisch würde das Meta-Metamodell einer DIN-Vorschrift zum Anfertigen von DIN-Vorschriften entsprechen.

2.1.4 Problemdomäne und Sprache

Die Problemdomäne definiert sich durch das fachliche Umfeld und ist die Gruppierung ähnlicher Probleme. Die in der Domäne feststehenden fachlichen Begriffe bilden die Grundlage jeglicher fachlichen Kommunikation.

Analogie: Problemdomäne
Eine mögliche Problemdomäne beim Hausbau ist der Bereich der Statik. Die in der Domäne gesprochene Sprache enthält z. B. die Begriffe Wandstärke und Material.

2.1.5 Domain Specific Language (DSL)

In der Domänen-spezischen Modellierung (engl. Domain Specific Modelling (DSM)) werden Modelle in einer Domänen-spezischen Sprache (engl. Domain Specific Language (DSL)) geschrieben. Diese Sprache bedient sich den Begriffen aus der Problemdomäne, wodurch die Modelle von Fachleuten intuitiv verstanden werden können. Venkatesan schreibt hierzu:

Zitat (aus dem Englischen)**:** Domain Specific Modelling und Domain Specific Language
In der Domänen-spezifischen Modellierung repräsentieren die Domänenelemente Dinge der Domänenwelt und nicht der Quellcode-Welt. Die Modellierungssprache, welche der Abstraktion und Semantik der Domäne folgt, gibt den Modellierenden das Gefühl sie würden direkt mit den Konzepten der Domäne arbeiten.

vgl.: [Ven06, S. 1]

Bei Verwendung von DSLs können durch die Konzentration auf das Wesentliche viele Fehler bei der Modellierung vermieden werden. Wird dagegen keine Domänen-spezifische Sprache, sondern z. B. eine generische also unspezifische Sprache benutzt, können Fehler durch die falsche oder missbräuchliche Verwendung von Sprachkonstrukten gemacht werden. Dies beruht auf der Möglichkeit, in der Sprache auch andere Begriffe (Modellelemente) zu verwenden. Ähnlich liegt der Fall bei der

Verwendung einer DSL, die für eine möglicherweise nur verwandte Problemdomäne entwickelt worden ist.

2.1.6 Syntax

Eine Syntax ist im Allgemeinen eine Vorschrift, die definiert, wie etwas zu schreiben ist. Im Rahmen der DSL werden die Begriffe „Abstrakte Syntax“ und „Konkrete Syntax“ verwendet. Die abstrakte Syntax definiert die internen Zusammenhänge in der DSL, d.h. wie wird das Modell verwaltet und wie kann auf das Modell zugegriffen werden. Eine konkrete Syntax ist die externe Sicht auf das Modell und wird von Spezialisten verwendet, um ein Modell zu erstellen. Zu einer DSL können mehrere konkrete Syntaxen definiert werden, mit denen jeweils andere Aspekte modelliert werden können. In einer konkreten Syntax können z. B. Elemente ausgeblendet werden, die im Modell vorhanden sind.

2.1.7 Editor

Fachleute einer Domäne benutzen Editoren, um Modelle in einer konkreten Syntax einer DSL zu erstellen. Die Editoren müssen das Modell entsprechend den im Metamodell definierten formalen Regeln bearbeiten können.

Um eine DSL zu erstellen, wird oft ein Metamodell entworfen, welches die Konzepte der Domäne als abstraktes Domänenmodell definiert. Auf der Grundlage des Metamodells ist es im Rahmen von Metamodellierungstools möglich, automatisch einen Editor zu generieren, der die Modelle konform zum Metamodell mit einer konkreten Syntax bearbeiten kann. Andere Ansätze gehen von einer konkreten Syntax aus und leiten hiervon das Metamodell ab.

Terfloth et al. erläutern die verschiedenen Möglichkeiten zur Erstellung von Editoren für DSLs. Die Autoren klassifizieren Editoren in die vier Bereiche der graphischen, textuellen, baumbasierten/hierarchischen und formularbasierten Notation. ([Ter07, S. 1]) Für jede Domäne muss geprüft werden, welche Art der Notation für das Modellieren am besten geeignet ist. Hier kann es auch sinnvoll sein, das Modell zu teilen und die Teile in unterschiedlichen Notationen zu bearbeiten.

Analogie: Editor
Der graphische Editor für das Modell, also die Bauzeichnung eines Hauses, ermöglicht die Platzierung von graphischen Objekte wie Wänden, Fenstern und Türen auf einer Zeichenfläche.

2.1.8 Modell-zu-Modell-Transformation (M2M-Transformation)

Die Modell-zu-Modell-Transformation (M2M-Transformation) ist ein integraler Bestandteil der modellgetriebenen Softwareentwicklung. Transformatoren überführen hierbei ein Modell in ein beliebiges anderes Modell. Beide Modelle sind jeweils Instanzen des ihnen zugeordneten Metamodells.

Eine Modell-zu-Text-Transformation (M2T-Transformation) ist letztendlich eine sehr spezielle Form der M2M-Transformation, da der generierte Text i. d. R. auch durch einen Abstract Syntax Tree (AST) beschrieben werden kann und bestimmten formalen Regeln entsprechen muss. Dies können z. B. eine DTD oder die Java-Quellcode-Spezifikation sein. Das Ergebnis ist bei der M2M-Transformation oft abstrakter als bei der M2T-Transformation, da bei letzterer meistens Plattformspezifische Details im Transformator, welcher hier Generator genannt wird, enthalten sind.

Ein Beispiel für eine M2M-Transformation ist die in das EMF integrierte Transformation zur Erstellung von Ecore-Modellen aus UML-Modellen. Mit dieser Transformation kann unter Einschränkungen ein UML-Klassenmodell in ein Ecore-Modell transformiert werden, um mit den generierten Editoren Modelle zu bearbeiten, die dem ursprünglichen UML-Metamodell genügen.

Eigenschaften, wie z. B. union und subsets, welche nicht direkt in EMF abgebildet werden, sind in [Ala05] näher erläutert. (vgl.: [Sta07, S. 66ff.])

2.1.9 Templates für Generatoren

Für die M2T-Transformation werden Templates eingesetzt, um eine Template-Engine zu parametrieren. Ein Generator setzt sich aus den Templates sowie der für die Ausführung der Templates notwendigen Template-Engine zusammen. Stahl definiert Templates wie folgt:

> **Zitat:** Templates
> Ein Template besteht aus beliebigem Text, der an bestimmten Stellen mit sog. Tags versehen ist. Tags enthalten Ausdrücke, die von der Template-Engine während des Generierungsvorganges an einem Eingabemodell evaluiert werden.
> [Sta07, S. 146]

2.1.10 Aktive und passive Generatoren

Neben der Einteilung von Transformatoren nach dem erstellten Ergebnis (Modell/Text) wird von Kriha eine Einteilung nach der Art bzw. der Häufigkeit des Aufrufes in aktive und passive Generatoren vorgenommen:

> **Zitat** (aus dem Englischen)**:** Passiver Generator
> Ein passiver Generator (Wizard) kann nur einmal zum Erzeugen eines Artefaktes benutzt werden. Sobald das Generat erstellt wurde, übernimmt der Entwickler das Ergebnis und verbessert oder vervollständigt es. [. . .] Normalerweise gibt es für passive Generatoren kein zugrunde liegendes Modell.
>  vgl.: [Kri05]
>
> **Zitat** (aus dem Englischen)**:** Aktiver Generator
> Ein aktiver Generator ist z. B. ein Kompiler. Als ein ständiger Teil des Entwicklungsprozesses wird ein Modell in ein anderes Artefakt transformiert. Sobald sich das Modell ändert, muss der Kompiler erneut gestartet werden. Typisch für aktive Generatoren ist, dass der generierte Code von den Entwicklern nicht modifiziert wird. Alternativ kann ein ausgereiftes System generierten und manuell veränderten Code anhand von Checksummen oder `@generated`-Tags (in EMF) im Code erkennen.
>  vgl.: [Kri05]

Für die Effizienz eines MDSD-basierten Entwicklungsprozesses ist es sehr wichtig, dass der von einem aktiven Generator generierte Quellcode nicht modifiziert werden muss. Das Generat kann so einfach gelöscht und anschließend wieder automatisch generiert werden. (vgl.: [Sta07, S. 159])

2.1.11 Aspektorientierte Programmierung (AOP)

Im Rahmen der MDSD gewinnt die Aspektorientierte Programmierung (AOP) immer mehr an Bedeutung. So kann generierter Quellcode von manuell erstellten Artefakten sauber getrennt werden. Die Aspekte greifen hier an definierten Punkten (Pointcuts) in die generierte Anwendung ein, um die manuelle Implementierung in das Generat einzuweben.

2.2 Standards und Produkte

Die in dieser Diplomarbeit verwendeten und dazu verwandte Standards sowie die genutzten Produkte werden in den folgenden Abschnitten beschrieben.

2.2.1 Meta Object Facility (MOF)

Ein Standard der Object Management Group (OMG) zur Definition von Metamodellen ist die Meta Object Facility (MOF), welche selbst ein Meta-Metamodell ist. Die wichtigsten Bestandteile hiervon werden in der Essential-MOF (EMOF) zusammengefasst. Es enthält z. B. keine Assoziationen, die als eigenständiges Objekt in MOF modelliert werden können. Stattdessen können nur Referenzen eingesetzt werden, in denen nur das Zielobjekt angegeben wird.

MOF definiert u. a., wie Klassen, Attribute, Referenzen und Datentypen in Namensräumen beschrieben werden. Eine Besonderheit von MOF und anderen rekursiven Metamodellen ist, dass es sich selbst definiert und der eigenen Definition entspricht. Damit ist es eine Instanz von sich selbst (vgl.: [Atk01, S. 7]). Seidewitz schreibt in diesem Zusammenhang über „Reflexive Metamodelle“:

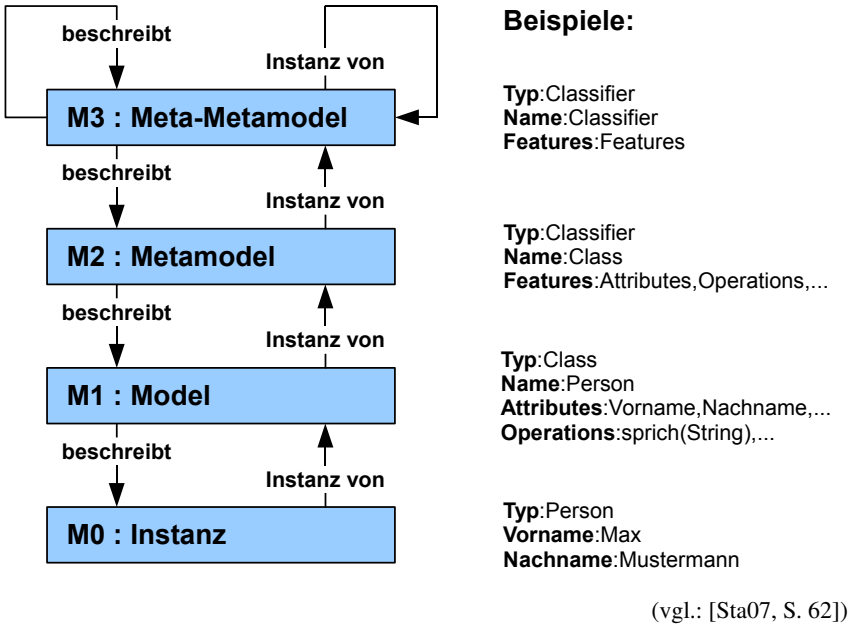
Zitat (aus dem Englischen)**:** Reflexives Metamodell
Da ein „Reflexives Metamodell“ in derselben Modellierungssprache ausgedrückt wird, die es selbst beschreibt, liefert die Interpretation ein Mapping der Modellierungssprache auf sich selbst. Es wird von der kompletten Modellierungssprache auf eine Teilmenge gemappt. Dies kann iterativ fortgeführt werden und produziert bei jedem Schritt eine kleinere Teilmenge, solange bis das „Minimale reflexive Metamodell“ erreicht wird, welches vollständig auf sich selbst, statt auf eine Teilmenge mappt.

vgl.: [Sei03, S. 6]

Die von der OMG verwendeten vier verschiedenen Ebenen (Meta-Metamodell, Metamodell, Modell und Instanz) in der Modellhierarchie sind in der Abbildung 2.2 dargestellt.

Zur Datenspeicherung der Modelle der verschiedenen Ebenen wird meistens XML Metadata Interchange (XMI), ein Extensible Markup Language (XML)-Format der OMG zum Austausch von Metadaten, genutzt. Da die MOF für die meisten Angelegenheiten zu umfangreich ist, wurde mit EMOF eine „abgespeckte“ Variante eingeführt, welche nur die wichtigsten Elemente wie „Package“, „Class“, „Attribute“ und „Reference“ berücksichtigt.

Analogie: Meta Object Facility
Die MOF würde der allgemeinen Vorschrift zum Anfertigen von Vorschriften entsprechen. Diese Vorschrift müsste demnach nach den selbst definierten Vorgaben geschrieben worden sein.



(vgl.: [Sta07, S. 62])

Abbildung 2.2: Meta-Ebenen der OMG

2.2.2 Unified Modelling Language (UML)

Die auf der MOF aufbauende Unified Modelling Language (UML) wurde als „alleskönnende Modellierungssprache“ (engl. „general purpose modelling language“; vgl.: [Exe04, S. 1]) konzipiert. Sie wird oft genutzt, um Dokumentationen von Programmen in einer vereinheitlichten Art und Weise zu beschreiben oder Teile, z. B. Methoden- und Klassenrümpfe, zu generieren und später zu implementieren.

Der im UML-Metamodell definierte und für den allumfassenden Ansatz notwendige Sprachumfang ist für die Modelle oft zu umfangreich. So muss, um z. B. Verhalten in einem UML-Modell auszudrücken, oft eine umständliche Beschreibung gewählt werden.

Mit den Profilen der UML 2 hat sich dieser Zustand ein wenig gebessert, da nun Typinformationen zu den einzelnen Modellelementen hinterlegt und somit an die Domäne angepasst werden können. In den mit Profilen erweiterten Modellen ist es jedoch weiterhin möglich, Domänen-fremde Aspekte zu modellieren - es steht die volle Funktionsvielfalt von UML zur Verfügung und lässt sich nicht einschränken. Das Programm „Magic Draw“ geht an dieser Stelle einen Schritt weiter und ermöglicht es, an das Profil angepasste Diagrammtypen zu verwenden. Mit ihnen werden die Möglichkeiten auf die vom Profil erlaubten reduziert. Trotzdem lässt sich hier „einfach“ das Profil umgehen, indem z. B. ein Klassendiagramm erstellt wird.

Ein weiteres Problem besteht in der Definition von Constraints, welche bei der UML in der Object Constraint Language (OCL) definiert werden, denn für die Erstellung der Regelsätze gibt es kaum Unterstützung seitens der UML-Tools. Aufgrund der Komplexität sind die Sprachen UML sowie OCL schwer zu erlernen bzw. zu verstehen. (vgl.: [Sta07, S. 64ff.])

Analogie: Unified Modelling Language
Ein Haus soll gebaut werden und es wird ein Modell erstellt, welches die Zuordnungen der in dem Haus wohnenden Personen zu den einzelnen Zimmern enthält. In der UML müsste nun ein Profil erstellt werden, mit welchem die Beziehungen zwischen Zimmern und Personen erstellt werden können. Jedoch ist es auch mit diesem Profil möglich, ein Klassendiagramm zu entwerfen, was jedoch für die Zimmer-Personen-Zuordnung nicht vorgesehen ist.

Für Eclipse existieren diverse Plugins, um UML-Diagramme zu erstellen. Neben dem UML2-Projekt ([UrlEclf]) von Eclipse sind hierzu vor allem Borland Together ([UrlBor]), Rational Rose ([UrlIBM]) sowie Omondo EclipseUML ([UrlOmo]) zu nennen.

2.2.3 Eclipse Modelling Framework (EMF)

Mit dem zu den Metamodellierungstools zählenden Eclipse Modelling Framework (EMF) ist es möglich, ein eigenes Metamodell sowie den dazu passenden Editor zu erstellen. Anschließend können mit dem Editor eigene Modelle bearbeitet werden.

Das EMF definiert hierzu mit Ecore ein eigenes Meta-Metamodell, was eine sehr große Ähnlichkeit mit der EMOF der OMG. Dies liegt an dem ähnlichen, modellierten Sachverhalt. Um für eine bestimmte Problemdomäne ein eigenes Metamodell zu erstellen, wird aufbauend auf dem sich selbst definierenden Ecore-Meta-Metamodell ein Ecore-Metamodell erstellt. Mit den Mitteln des EMF kann nun aufbauend auf dem Ecore-Metamodell ein reflektiver und dynamischer Editor gestartet werden. Alternativ kann auch ein eigenständiger EMF-basierter Editor generiert und in einer separaten Eclipse-Runtime-Workbench gestartet werden. (vgl.: [Sta07, S. 71f.])

Mit beiden Editoren ist es möglich, Instanzen des zuvor erstellten Metamodells also Modelle, zu erstellen und zu bearbeiten. Alle Modelle werden in der Regel im XMI-Format gespeichert. Der resultierende Editor stellt das Modell als Baum dar und wird daher auch als Baumeditor oder Baum-basierter Editor bezeichnet. (vgl.: [UrlEclb])

Das EMF generiert diese Editoren aus jedem Paket eines „EMF-Model“ oder auch „GenModel“ genannten Generator-spezifischen Modells, welches wiederum aus dem Ecore-Modell und den dort enthaltenen Paketen erstellt wird. Das „GenModel“ kann auch aus anderen Metamodell-Sprachen wie z. B. einem XML-Schema oder auch mit Annotationen versehenen Java-Schnittstellen erstellt werden.

Um ein neues Modell anlegen zu können, muss der Typ des Wurzelelementes angegeben werden. Für jeden im Metamodell über eine `Containment`-Referenz verknüpften Typ kann nun im Editor eine Instanz als Kindelement angelegt werden. Die aus dem Ecore-Modell automatisch erstellten Baumeditoren sind schwer zu handhaben, wenn das dargestellte Modell Relationen enthält, die in einem Baum nicht gut nachzuvollziehen sind. Dieser Nachteil lässt sich unter Umständen mit der Anpassung des bestehenden Editors beheben. Mit der im Vergleich dazu komplexen Entwicklung eines graphischen Editors können Relationen jedoch auch als solche dargestellt werden.

2.2.4 openArchitectureWare (oAW)

oAW wurde ursprünglich von der „b+m Informatik AG“ unter dem Namen „b+m Generator Framework“ entwickelt und im Jahr 2003 als Open Source veröffentlicht, seitdem es kontinuierlich weiterentwickelt wird.

Die Software umfasst mit der Workflow-Engine eine integrierte Umgebung, um andere Komponenten in einer definierten Konfiguration ablaufen zu lassen. Als Komponenten werden z. B. weitere,

geschachtelte Sub-Workflows, die Templatesprache Xpand für Modell-zu-Text- und die funktionale Programmiersprache Xtend für Modell-zu-Modell-Transformationen sowie das Check-Framework zum Prüfen von Modellen verstanden. Zusätzlich können mit dem Xtext-Framework textuelle Domäneneditoren erstellt werden. (vgl.: [UrlEcle])

Die genannten Komponenten werden in folgenden Abschnitten genauer erklärt.

Workflow-Engine

Die Workflow-Engine ermöglicht es, mit einer einfachen XML-basierten Sprache einen modellgetriebenen Ablauf zusammenzustellen. Dies wird auch als „Orchestration“ bezeichnet.

Xpand

Xpand ist die im oAW-Umfeld verwendete Sprache zum Generieren von beliebigen Artefakten. Um die Integration von generiertem und manuell geschriebenem Quellcode zu vereinfachen, können geschützte Bereiche („Protected Regions“) genutzt werden.

Die AOP ist in der Entwicklung von Anwendungen bereits etabliert und z. B. mit AspectJ auch gut in die Entwicklungsumgebung Eclipse integriert (vgl.: [UrlEcla]). Für Generatoren ist es jedoch eine relativ neue Entwicklung, mit der es ermöglicht wird, bereits definierte Templates zu überschreiben. Diese Möglichkeit wird von oAW seit der Version 4.1 unterstützt und von Völter in [Vö06] näher beschrieben.

Xtend

Die von den verschiedenen oAW-Sprachen verwendete Ausdruck-Sprache (Expression) ähnelt der OCL und wird u. a. benutzt, um über die Modelle zu navigieren. Vor allem in Xtend werden die Ausdrücke genutzt, um z. B. eine M2M-Transformation zu implementieren.

Ähnlich wie in Xpand gibt es auch in Xtend Aspekte, mit denen Funktionen überschrieben werden können.

Check

Eine Transformation im Rahmen einer modellgetriebenen Softwareentwicklung entspricht dem Übersetzungs- oder Interpretationsvorgang bei der herkömmlichen Softwareentwicklung. Dementsprechend sollte vor einer Transformation auch geprüft werden, ob das Eingangsmodell bestimmten Regelsätzen entspricht. Diese Regelsätze sollten ähnlich fein granular sein, wie sie auch ein Kompiler bei der Übersetzung ausgibt. Das Finden eines Modellierungsfehlers verursacht erhebliche Zeitaufwände, wenn der Fehler erst zur Übersetzungszeit der generierten Artefakte auftritt. Mit dem Check-Framework können diese Regelsätze erstellt und mit Fehlerausgaben hinterlegt werden.

Xtext

Mit dem Xtext-Framework können textuelle Modelleditoren erstellt werden. Hierzu wird die Sprache in einer Extended Backus–Naur Form (EBNF)-ähnlichen Syntax definiert. Xtext generiert nun aus der Syntax Modellklassen und, um das Modell laden und speichern zu können, einen Parser sowie einen Serializer. Der im selben Schritt generierte und dynamisch durch Xtend-Artefakte erweiterbare Texteditor unterstützt Syntaxhervorhebung, Textvervollständigung und eine Outline-Ansicht.

2.3 Graphische Editoren

Für die Entwicklung einer graphischen DSL gibt es verschiedene Möglichkeiten. Von diesen stellen die folgenden zwei Abschnitte die Eclipse-basierten Möglichkeiten vor.

2.3.1 Graphical Editing Framework (GEF)

Das Graphical Editing Framework (GEF) stellt viele grundlegenden Funktionen zur Erstellung von graphischen Editoren bereit und besteht aus der Darstellungs- und Rendering-Komponente Draw2d sowie dem Teil von GEF, der für die Bearbeitung von graphischen Elementen verantwortlich ist. Um einen GEF-basierten Editor zu erstellen, müssen komplexe Java-Klassenstrukturen nach dem Model-View-Control (MVC)-Paradigma erstellt werden. (vgl.: [UrlEcld])

2.3.2 Graphical Modelling Framework (GMF) und TOPCASED

Mit dem GMF als auch dem TOPCASED können graphische Editoren anhand von Modellen beschrieben werden. Beide Frameworks nutzen das EMF sowie das GEF intern. Mit dem EMF wird auf das Modell hinter der graphischen Repräsentation zugegriffen. Das GEF wird verwendet, um die einzelnen Elemente des Modells mit der entsprechenden graphischen Repräsentation darzustellen. Um einen graphischen Editor zu erstellen, wird aufbauend auf dem Ecore-Metamodell eine formalisierte Beschreibung für den Editor angelegt, aus welcher anschließend der eigentliche Editor generiert wird.

Prof. Zimmermann hat für die Nordakademie die beiden Frameworks anhand eines Kriterienkataloges hinsichtlich der funktionalen Abdeckung, des Investitionschutzes, der Anpassbarkeit sowie des Softwaremanagements untersucht und verglichen. „Dabei stellt sich heraus, dass TOPCASED zwar einen einfacheren Einstieg erlaubt, GMF aber das ausgereiftere Werkzeug mit umfassenderer Funktionalität ist.“ [Zim07, S. 1]

Die Erstellung eines GMF-basierten Editors wird ausführlich im Abschnitt 3 beschrieben.

2.4 Modellgetriebene Softwareentwicklung

Das für diese Diplomarbeit sehr wichtige Konzept der Model Driven Software Development (MDSD) wird von Stahl folgendermaßen definiert:

> **Definition:** MDSD
> Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.
> [Sta07, S. 11]

Hierbei werden Modelle mit den generierten Artefakten gleichgesetzt bzw. ersetzen diese und sind damit als höherwertiges Gut zu bewerten. In der Praxis bedeutet dies, dass die generierten Artefakten im Gegensatz zum Modell und dem Generator nicht unter Versionskontrolle stehen, da sie jederzeit neu generiert werden können.

Die Model Driven Architecture (MDA), eine spezielle Form der MDSD, wurde von der OMG definiert und verwendet MOF als einziges Meta-Metamodell. Damit können auch nur MOF-basierte DSLs für die Definition der Modelle benutzt werden. In der Praxis wird die Verwendung von UML-Profilen als konkrete Syntax empfohlen. (vgl.: [Sta07, S. 36])

Wenn bei der UML keine Profile eingesetzt werden, kann die mögliche Abstraktion vom Code aufgrund der verwendeten Modellkonstrukte (Klassen, Attribute, Funktionen) nicht so hoch sein, wie es bei der Verwendung von einer DSL möglich ist (vgl.: [Tol06, S. 1]). Es ist jedoch zu beachten, dass beim Einsatz von Profilen Fehler in der Modellierung auftreten können, da die sehr universell gehaltene Domäne in Ihrer Ausdruckskraft nicht beschränkt werden kann. (s. a. Kapitel 2.2.2).

Mit dem Einsatz von oAW können komplexe MDSD-Szenarien aufgebaut werden. Hierbei können die Frameworks EMF und GMF von Eclipse sowie Xtext von oAW benutzt werden, um mit den erstellten DSL-Editoren die im MDSD-Prozess verwendeten Modelle zu bearbeiten. oAW ermöglicht es auch, mit anderen Produkten erstellte Modelle zu verarbeiten.

Als weiterführende Literatur wird an dieser Stelle auf die in [Sta07, S. 11-52] genannten Definitionen, Entwicklungsansätze und Techniken verwiesen.

Die modellgetriebene Softwareentwicklungsumgebung „base“ mit dem verwendeten Entwicklungsprozess und den Funktionen der einzelnen verwendeten Modelltypen ist im Kapitel 4 ausführlich beschrieben.

KAPITEL 3

Graphical Modelling Framework

Im Rahmen der Entwicklung von graphischen Editoren mit dem GMF gibt es viele Möglichkeiten, ein bestimmtes gewünschtes Ergebnis zu erreichen. So enthält das GMF verschiedene Wizards, um aus einem vorhandenen Domänenmodell einen rudimentären graphischen Editor zu erstellen. Die von den Wizards erstellten Modelle sind einfach strukturiert und können als Basis für manuelle Änderungen genutzt werden, um den Editor weiter zu entwickeln. Neben den Modelländerungen besteht die Möglichkeit, Eclispe-Extension-Points zu nutzen, mit denen das Aussehen sowie das Verhalten genauer definiert werden können.

In den folgenden Abschnitten werden neben dem Entwicklungsprozess verschiedene Varianten der Erweiterung erläutert. Soweit nicht anders angegeben, beschreiben die in den Abbildungen dieses Kapitels dargestellten Strukturen die Ergebnisse der entsprechenden Wizards. Die in dieser Diplomarbeit angegebenen Bezeichnungen für Modellelemente des GMF sind mit Icons ausgezeichnet (z. B. `NodeMapping` ( )), die den GMF-Plugins entnommen wurden.

Dieses Kapitel sowie die darauf folgenden enthalten Abbildungen, die ein Modell schematisch darstellen. Hier wird eine Baumstruktur mit geschlossenen Pfeilspitzen (⟶) verwendet, um eine inhaltliche Beziehung (`Containment`; „Ein Haus hat eine Haustür.“) zueinander zu repräsentieren. Im Gegensatz dazu werden offene Pfeilspitzen (⟶) genutzt, um die Verwendung von Referenzen („Diese Tür führt zur Terasse.“) anzuzeigen. Diese genannten Abbildungen entsprechen dem in der Abbildung A.2 dargestellten Farbschema (siehe Anhang A 2).

Eine komplette Klassenhierarchie sowie die komplette Inhaltshierarchie ist einer dieser Arbeit beigefügten Dokumentation der Metamodelle des GMF („tooldef“, „gmfgraph“ und „mappings“) in digitaler Form als kreuzverlinkte HTML-Seiten zu entnehmen ([Sch08a]). Diese Dokumentation wurde automatisch aus den Metamodell-Beschreibungen mit dem Eclipse-Plugin MetamodelDoc (entwickelt vom Verfasser dieser Diplomarbeit, siehe [UrlSch07]) erzeugt. Eine einfache Einführung in die Konzepte des GMF ist bei Venkatesan zu finden (vgl.: [Ven06]).

3.1 Entwicklungsprozess

Das GMF unterstützt die modellgetriebene Entwicklung von graphischen Editoren. Der dazugehörige Entwicklungsprozess ist in der Abbildung 3.1 dargestellt. Ausgehend von der Problemanalyse wird zunächst das Metamodell als EMF „Domain Model“ erstellt. Mit Hilfe der GMF-Wizards können auf der Grundlage des Metamodells die verschiedenen GMF-Modelle generiert werden. Diese bilden die Basis für die Generierung des GMF-Editors. Der fertige Editor kann von den Anwendern genutzt werden, um Modelle konform zum Metamodell zu bearbeiten.

Die Abhängigkeiten der Modelle bzw. Plugins untereinander werden in der Abbildung 3.2 gezeigt. Das „Mapping Def Model“ enthält Referenzen auf die beiden anderen GMF-Modelle sowie das

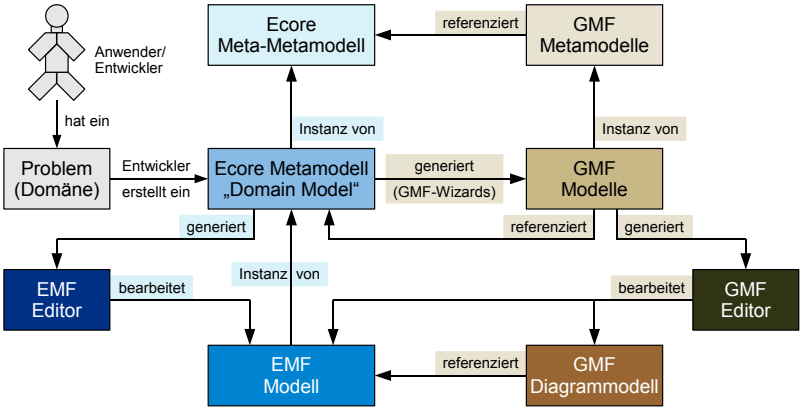


Abbildung 3.1: Vereinfachte Darstellung des Entwicklungsprozesses für GMF

„Domain Model“ und bildet damit Einheiten aus dem Tool, der graphischer Repräsentation sowie dem Metamodellelement. Nach einer Transformation in das „Diagram Gen Model“ wird aus diesem Modell der GMF-Editor generiert. Das hier linksseitig grau dargestellte EMF-Editor-Plugin wird nur für den EMF-Baumeditor verwendet und ist eine Alternative zum GMF-basierten Editor.

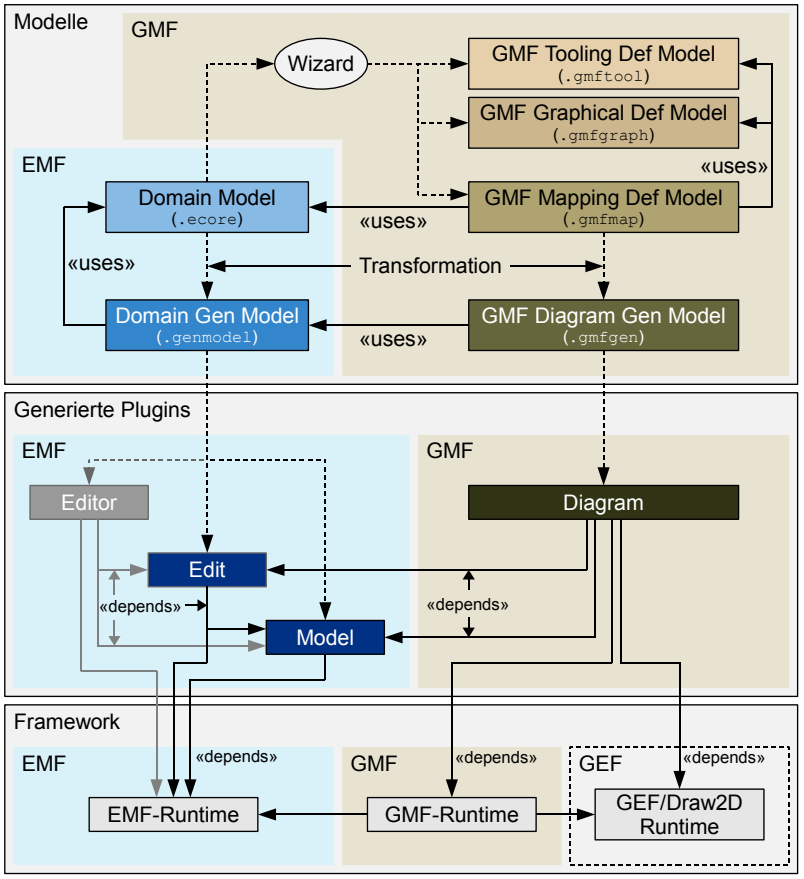
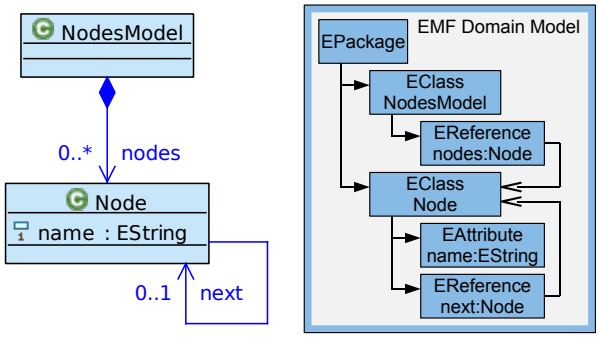


Abbildung 3.2: Entwicklungsprozess für GMF

3.2 EMF „Domain Model“ und „Domain Gen Model“

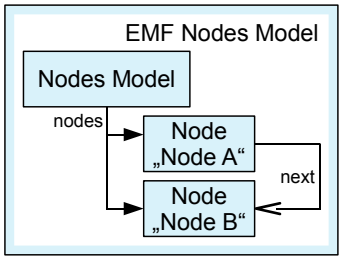
Die Modelle „Domain Model“ (bzw. „`ecore`“-Modell) und „Domain Gen Model“ (bzw. „EMF-Model“ oder „`genmodel`“-Modell), welche mit dem EMF bearbeitet werden können, dienen als Grundlage für die Erstellung von graphischen Editoren mit dem GMF. Sie müssen demzufolge auch als Erstes erstellt werden. Bei der Erstellung kann es leicht zu Fehlern im Metamodell kommen, wenn das `containment`-Attribut einer Referenz falsch gesetzt wird. Mit dem durch das EMF generierten Baumeditor kann auf einfache Weise geprüft werden, ob im Metamodell die Syntax korrekt modelliert wurde. Die im „Domain Model“ verfügbaren Informationen werden von den Wizards des GMF genutzt, um dem Entwickler Vorschläge für die Erstellung der GMF-Modelle zu unterbreiten.

Das in der Abbildung 3.3 dargestellte EMF „Domain Model“ ist ein einfaches Domänenmodell. Hier wird in (a) ein UML-ähnliches Ecore Diagramm und in (b) die in EMF notwendige Baumstruktur gezeigt. Dieses Domänenmodell wird als Grundlage für alle in diesem Kapitel gezeigten Diagramme verwendet. Eine mögliche Instanz ist in der Abbildung 3.4 abgebildet.



(a) Ecore Diagramm **(b)** Ecore Modell

Abbildung 3.3: EMF „Domain Model“



(Dieses Modell ist eine Instanz des EMF „Domain Model“ aus der Abbildung 3.3)

Abbildung 3.4: EMF „Nodes Model“

3.3 GMF „Tooling Def Model“

Das „Tooling Def Model“ („`gmftool`“) aus dem GMF beschreibt u. a. die in GEF-Editoren übliche Palette, welche die für den Editor verfügbaren Funktionalitäten in einer Toolbox darstellt. Dies sind in der Regel neben den Tools zur Erstellung der Elemente des Metamodells die Select-, Zoom- sowie Note-Tools. Letzteres dient dazu, Elemente des Diagramms mit Anmerkungen zu versehen. Eine schematische Darstellung einer einfachen Palettendefinition ist in der Abbildung 3.5 dargestellt. Die `CreationTools` (✦) sind in diesem Beispiel direkt innerhalb der `Palette` (?) angeordnet.

Für komplexere Editoren mit vielen verschiedenen Tools ist es sinnvoll, `CreationTools` (✦) in `ToolGroups` (✦) thematisch zusammenzufassen. Das Modell muss diesbezüglich manuell editiert werden, da der im GMF integrierte Wizard hierfür keine Unterstützung bietet. Bei einem erneuten Aufruf des Wizards werden die alten Einträge sowie die Änderungen übernommen. Eine vollständige Dokumentation inklusive einer Klassen- und einer Inhaltshierarchie ist dem Paket „tooldef“ aus [Sch08a] zu entnehmen.

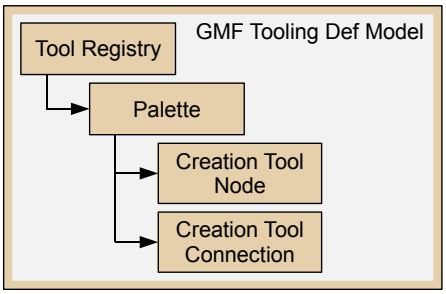


Abbildung 3.5: GMF „Tooling Def Model“

3.4 GMF „Graphical Def Model“

Das „Graphical Def Model“ („`gmfgraph`“) des GMF repräsentiert die visuellen Eigenschaften der Diagrammelemente. Hierzu zählen neben der Form der Objekte (z. B. Rechteck, Polygon, Ellipse), der Hintergrund- und Linienfarbe, den Schrifteffekten sowie Größen- und Positionsangaben auch Objekthierarchien für die Figur und Definitionen für den Zugriff auf die einzelnen Objekte der Graphik.

Eine schematische Darstellung einer typischen Konfiguration ist in der Abbildung 3.6 zu sehen. Die `FigureGallery` (✦) enthält alle in einem Diagramm sichtbaren Elemente, welche durch `FigureDescriptors` (✦) beschrieben werden. Sie werden durch eine weitere Abstraktion (im Diagramm als „Externe Sicht“ gekennzeichnet) gekapselt. Kindelemente, wie z. B. `Labels` (✦), müssen über `ChildAccess`-Elemente (✦) aus der Abstraktionsschicht (in diesem Fall `DiagramLabels` (✦)) angesprochen werden. Dies ermöglicht eine Wiederverwendung auf der Ebene der Figurbeschreibun-

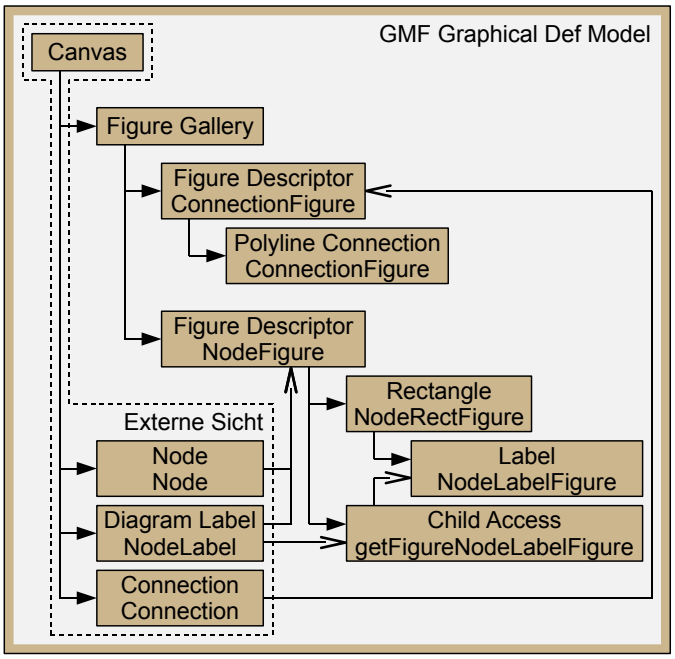


Abbildung 3.6: GMF „Graphical Def Model“

gen, da z. B. ein `FigureDescriptor` (✦) von mehreren `Node`-Elementen (✦) referenziert werden kann. Dies geht über die Möglichkeiten des Wizards hinaus. In der GMF-Dokumentation [Sch08a] ist die vollständige Dokumentation inklusive einer Klassen- und einer Inhaltshierarchie im Paket „gmfgraph“ enthalten.

3.5 GMF „Mapping Def Model“

Das GMF „Mapping Def Model“ („`gmfmap`“) verknüpft die in den Abschnitten 3.2 bis 3.4 beschriebenen Modelle („Domain Model“, „Tooling Def Model“ und „Graphical Def Model“). Beispielsweise wird hierzu die `Children`- bzw. `Containment Feature` Referenz des `TopNodeReference`-Elements (▹) aus dem Mapping-Modell auf die Referenzbeziehung des Diagramm-Wurzelelementes aus dem „Domain Model“ gesetzt. Zusätzlich verweist ein Kindelement vom Typ `NodeMapping` (⁞) auf die zu der Referenzbeziehung zuweisungskompatiblen Elementklasse, die definierte abstrahierte Figurbeschreibungen `Node` (✦) (siehe Kapitel 3.4) aus dem „Graphical Def Model“ sowie auf das `Tool` aus dem „Tooling Def Model“. Nach diesem Schema muss für jede Kombination aus Referenzbeziehung, Elementklasse, Figurbeschreibung und Tool ein `TopNodeReference` (▹)-Eintrag mit `NodeMapping` (⁞)-Kindelement erstellt werden.

In der Abbildung 3.7 wird eine schematische Darstellung einer solchen Mapping-Definition gezeigt. Wie auch im „Graphical Def Model“ kann hier eine Wiederverwendung bei den referenzierten abstrahierten Figurbeschreibungen stattfinden. Beim Aufrufen des Wizards kommt es bei den neu erstellten Mappings oft zu falschen Zuordnungen. Aus diesem Grund sollte ein „Mapping Def Model“ von Hand erstellt werden.

Der folgende Abschnitt stellt zwei Varianten zum Abbilden von Verbindungen vor und in den darauf folgenden vier Abschnitten werden manuelle Änderungen am „Mapping Def Model“ beschrieben, die in dieser Form nicht von den im GMF enthaltenen Wizards unterstützt werden. Durch die Änderungen werden Knoten so definiert, dass die im Modell enthaltene Komplexität in verschiedenen Varianten dargestellt werden kann.

3.5.1 Verbindungen

Es gibt zwei Möglichkeiten, eine Verbindung für einen graphischen Editor zu definieren. Im zugrunde liegenden „Domain Model“ kann eine Referenz auf eine Metamodellklasse definiert werden. Alternativ kann eine eigene Klasse für die Verbindung definiert werden, in der neben den zwei Referenzen auf den Verbindungsursprung („from“) sowie das -ziel („to“) weitere Attribute enthalten sein können. Für die Referenz-basierte Verbindung können keine weiteren Attribute angegeben werden.

Im „Graphical Def Model“ kann nun ein `Connection`-Element (✦) mit der passenden Figurbeschreibung (z. B. mit einer `PolylineConnection` (✦)) hinterlegt werden. Ein im „Mapping Def Model“ anzulegendes `LinkMapping`-Element (⁖) muss das `Connection`-Element (✦) und ein Tool referenzieren. Für die Referenz-basierte Verbindung muss das Referenzbeziehungselement aus dem Metamodell als Ziel angegeben werden. Die zusätzlichen Attribute „Containment Feature“, „Element“ und „Source Feature“, die ebenfalls Elemente aus dem Metamodell referenzieren können, müssen jedoch leer gelassen werden. Wenn der zweite Verbindungstyp benutzt wird, müssen diese Attribute gefüllt werden.

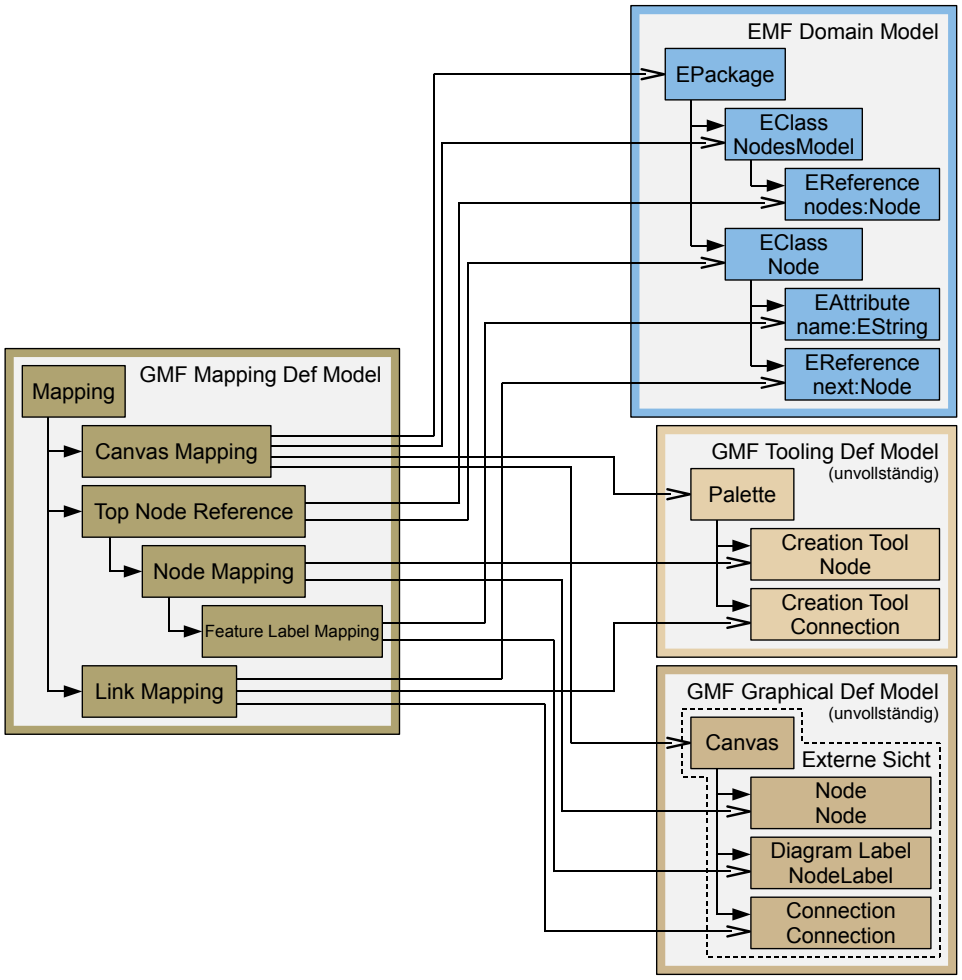
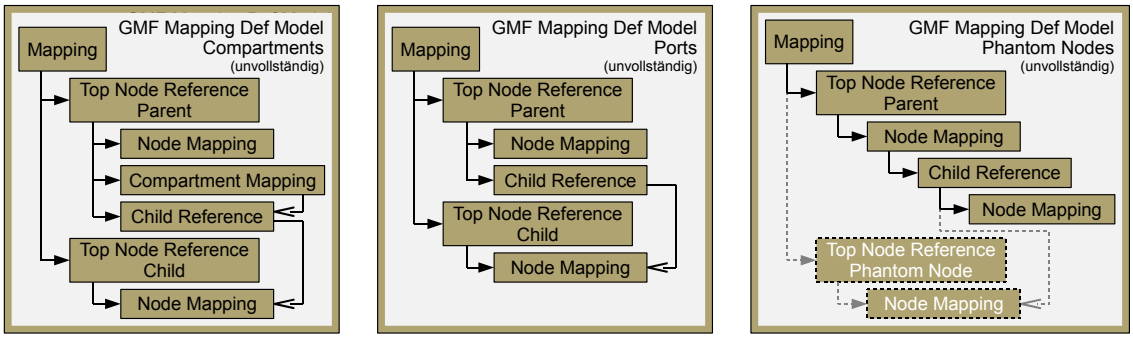


Abbildung 3.7: GMF „Mapping Def Model“



(a) Compartments **(b)** Ports **(c)** Phantom Nodes

Abbildung 3.8: Manuelle Änderungen am GMF „Mapping Def Model“

3.5.2 Compartments

Bei komplexeren Modellen ist es sinnvoll, Knoten in einem anderen Knoten darzustellen, d.h. zu schachteln. Da dies von den Wizards nicht unterstützt wird, muss hierfür im „Graphical Def Model“ ein Rechteck innerhalb der Eltern-Figur manuell definiert werden, welches die Kinder aufnehmen soll. Dieses wird – ähnlich einem Label – über ein `ChildAccess` referenziert und als `Compartment` definiert. Im `NodeMapping` wird ein `CompartmentMapping` sowie für jedes mögliche Kindelement eine `ChildReference` angelegt. Die `ChildReferences` referenzieren die `NodeMappings` der Typen der möglichen Kinder als `ReferencedChild` und das `CompartmentMapping` als `Compartment`. Analog zu der `TopNodeReference` werden in den `ChildReferences` die Referenzbeziehung und die Elementklasse aus dem EMF „Domain Model“ verknüpft. Eine schematische Darstellung hierzu ist – ohne externe Referenzen – in der Abbildung 3.8a zu finden.

3.5.3 Ports

Um ein Kindelement eines Knoten nicht innerhalb, sondern außerhalb eines Knotens darzustellen (ähnlich den Ports (⊐C, •⊐) in einem UML-Komponentendiagramm), muss aus der Liste der `ReferencedChilds` des `CompartmentMapping`-Elementes im „Mapping Def Model“ die `ChildReference` entfernt werden. Die Konfiguration von Ports wird durch die Abbildung 3.8b verdeutlicht.

3.5.4 Phantom Nodes

Wenn eine `TopNodeReference` mit keiner Referenzbeziehung aus dem Wurzelelement verknüpft ist und damit nicht direkt auf der Zeichnungsfläche des Diagramms platziert werden kann, wird die `TopNodeReference` als „Phantom Node“ bezeichnet. GMF gibt in diesen Fällen eine Warnung bei der Transformation in das „Diagram Gen Model“ aus. Um dies zu verhindern und das Modell syntaktisch korrekt aufzubauen, muss – wie in der Abbildung 3.8c gezeigt – innerhalb der `ChildReference` ein weiteres `NodeMapping` eingefügt werden. Dieses `NodeMapping` referenziert das zum Kind gehörende Tool und die Figurbeschreibung. Zum Aufbau rekursiver `Compartment`-Strukturen ist es jedoch teilweise notwendig, „Phantom Nodes“ einzusetzen (in der Abbildung 3.8c grau dargestellt).

3.5.5 Diagram Partitioning

Die einem Knoten zugehörigen Informationen können so komplex sein, dass sie nicht in dem Knoten selbst dargestellt werden können. Hier ist es sinnvoll mit der Diagrammpartitionierung (engl. „Diagram Partitioning“) einen zweiten Editor zu öffnen, der die Aspekte des komplexen Knotens enthält. Das „UML2“-Projekt z. B. nutzt diese Möglichkeit, um in einem Klassendiagramm in ein Unterpaket zu springen (siehe [UrlEclf]). Für die Umsetzung dieser Funktialität müssen neben Änderungen am „Mapping Def Model“ diverse Einstellungen im „Diagram Gen Model“ manuell angepasst werden. (vgl.: [Rei07])

3.6 GMF „Diagram Gen Model“

Das GMF „Diagram Gen Model“ („gmfgen“) ist wie das EMF „Domain Gen Model“ ein Generator-spezifisches Modell und wird durch eine M2M-Transformation aus den in den Kapiteln 3.2 bis 3.5 genannten Modellen (EMF „Domain Model“ sowie GMF „Tooling Def Model“, „Graphical Def Model“ und „Mapping Def Model“) erstellt. Einige Einstellungen für den GMF-Diagramm-Editor können nur hier durchgeführt werden. Viele dieser Einstellungen werden bei wiederholten Transformationen aus dem alten Modell übernommen und gehen damit nicht verloren. Hierzu gehören z. B. die Attribute `Dynamic Templates`/ `Template Directory` des `Gen Editor Generator`-Elementes sowie `Validation Decorators`/ `Validation Enabled` und `Synchronized` des `Gen Diagram`-Elementes.

3.7 Änderungen am generierten Editor

In den bisherigen Kapiteln wurde beschrieben, wie entweder durch das Ändern von Modellen oder durch das Ändern von Templates der generierte Editor verändert werden kann. Im Rahmen der Entwicklung von graphischen Editoren kommt es jedoch oft vor, dass sich die gewünschten Funktionalitäten nicht mit Änderungen an den Modellen abbilden lassen. In der Regel sollten Veränderungen an generiertem Code möglichst vermieden werden. (vgl.: [Sta07, S. 159]) Um die gewünschten Funktionalitäten ohne Änderungen am Code umzusetzen, bestehen im Rahmen des GMF-Frameworks verschiedene Möglichkeiten. Diese sind in den folgenden Abschnitten beschrieben.

3.7.1 Fragmente

Eclipse bietet mit Fragmenten eine Möglichkeit, bereits bestehende Plugins zu erweitern und einzelne Komponenten in diesen zu ersetzen. Für den GMF-basierten Diagramm-Editor können so z. B. weitere Tools zur Palette oder neue Menüeinträge im Kontextmenü hinzugefügt und mit Aktionen belegt werden. Des Weiteren können in den GMF-Modellen z. B. ausschließlich Attribute desselben Elementes für die Anzeige von Labels in den Knoten und an den Verbindungen verwendet werden. Sobald eine Referenz verfolgt werden soll, um einen Text anzuzeigen, muss manuell ein entsprechender Provider entwickelt und über einen Extension-Point registriert werden.

3.7.2 Aspekte

Mit der AOP und dem AspectJ-Framework ist es wie auch mit Fragmenten möglich, generierten Code zu erweitern, ohne ihn zu verändern. Hierzu muss im generierten Code ein passender Pointcut gefunden werden, um dem System in einem Aspect den manuell entwickelten Code hinzuzufügen. Auf diese Weise kann z. B. die Ausrichtung des zu einem `Label` (✦) gehörenden Icons einfach gesetzt werden.

3.7.3 Dynamic Templates

Da GMF seit der Version 2 für den Generierungsprozess auf das oAW Xpand-Framework (siehe Kapitel 2.2.4) aufbaut, stehen hier die Möglichkeiten der AOP für die Templates zur Verfügung. Um diese Funktionalität nutzen zu können, muss im GMF „Diagram Gen Model“ im Element `GenEditorGenerator` der Wert für `DynamicTemplates` auf „`true`“ gesetzt werden. Des

Weiteren muss im selben Element das Attribut `TemplateDirectory` entsprechend dem Schema „`/[Plugin-Name]/templates`“ angepasst und in dem angegebenen Verzeichnis ein weiteres Verzeichnis mit dem Namen „`aspects`“ angelegt werden. In dem „`aspects`“-Verzeichnis muss für die zu verändernden Templates die originale Verzeichnis- und Dateistruktur aus dem GMF-Generator-Plugin abgebildet werden.

3.7.4 „Diagram Gen Model“-Extensions

Der „Diagram Gen Model“-Editor kann mit der Angabe eines eigenständigen Ecore-Modells dynamisch um zusätzliche Modellelementtypen erweitert werden. Mit dem erweiterten Metamodell können im Editor zusätzliche Elementbäume angelegt werden. Die zusätzlichen Informationen können von dafür vorbereiteten „Dynamic Templates“ ausgewertet werden, um den generierten Editor anzupassen.

Zum Beispiel ist im Rahmen der oAW-Software ein Erweiterungsplugin „GMF2-Adapter“ verfügbar, welches eine einfache Validierung von Modellen mit dem Check-Framework ermöglicht. Das hierfür benötigte Template wird ebenfalls im Rahmen des Plugins zur Verfügung gestellt. Die Abbildung 3.9 stellt die Zusammenhänge zwischen den „Dynamic Templates“ und den Erweiterungsplugins am Beispiel des hier vorgestellten oAW-Plugins dar.

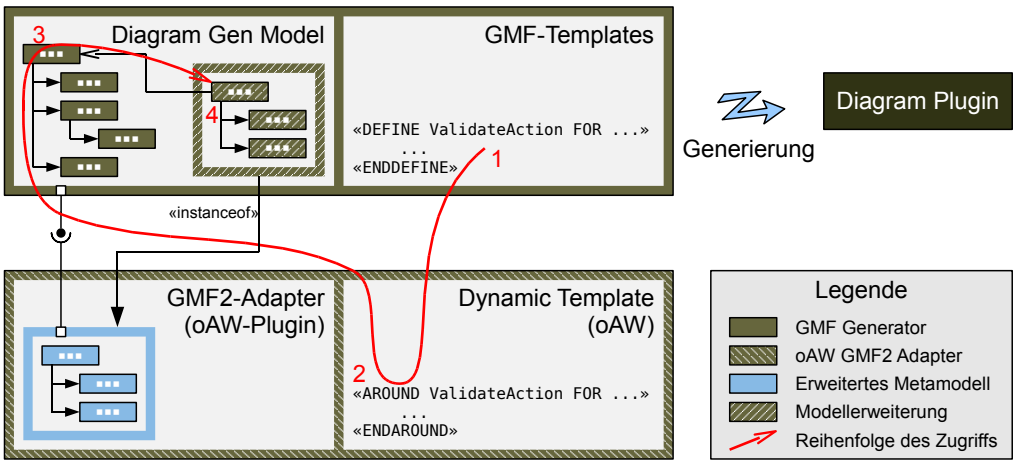


Abbildung 3.9: Die Verwendung des oAW „GMF2-Adapters“ mit den „Dynamic Templates“

Sobald das Template in die Generierung eingebunden ist, wird eine veränderte `ValidateAction` generiert (1, 2). Das Template greift über den Wurzelknoten (3) des „Diagram Gen Model“ und die darauf zeigende Rückrefernz auf die angegebenen Check-Dateien (4) zu. Mit dieser Information kann die `ValidateAction` entsprechend generiert werden, um das Modell zu prüfen. Eine ausführliche Beschreibung der Vorgehensweise ist in [ope07] nachzulesen.

KAPITEL 4

Die Softwareentwicklungsumgebung „base“

Während des Praktikums bei der „b+m Informatik GmbH Berlin“ hat der Verfasser dieser Diplomarbeit an einer Anwendungsplattform „base“ und einem darauf abgestimmten modellgetriebenen Entwicklungsprozess mitentwickelt. In diesem Entwicklungsprozess werden Modelle erstellt, welche eine Anwendung beschreiben. Die Anwendung wird anschließend aus den Modellen generiert. In der Regel handelt es sich bei den mit „base“ entwickelten Anwendungen um eine im Auftrag eines Kunden entwickelte Individualsoftware. Es kann jedoch auch aus bereits vorhandenen Modulen ein Produkt im Sinne einer Standardsoftware zusammengestellt werden. (vgl.: [Sch07])

Da im Rahmen dieser Diplomarbeit ein Editor für in „base“ verwendete `dataflow`-Modelle entwickelt wird, soll an dieser Stelle ein Überblick über den Entwicklungsprozess in „base“ gegeben werden. Anschließend werden das `dataflow`-Modell ausführlich sowie die weiteren Modelle kurz erläutert.

4.1 Entwicklungsprozess

Der Entwicklungsprozess für die Anwendungsplattform „base“ gliedert sich in die Bereiche „Analyse der Problemdomäne“, „Erstellen der Masken“ und „Erstellen der Modelle “ sowie das „Generieren“, „Anpassen“, „Testen“ und „Ausliefern der Anwendung“. Der Ablauf im Entwicklungsprozess ist in der Abbildung 4.1 dargestellt. Da im Rahmen dieser Diplomarbeit die Modelle und nicht der Entwicklungsprozess selbst im Vordergrund stehen, soll im Folgenden nur ein Überblick über den Prozess vermittelt werden.

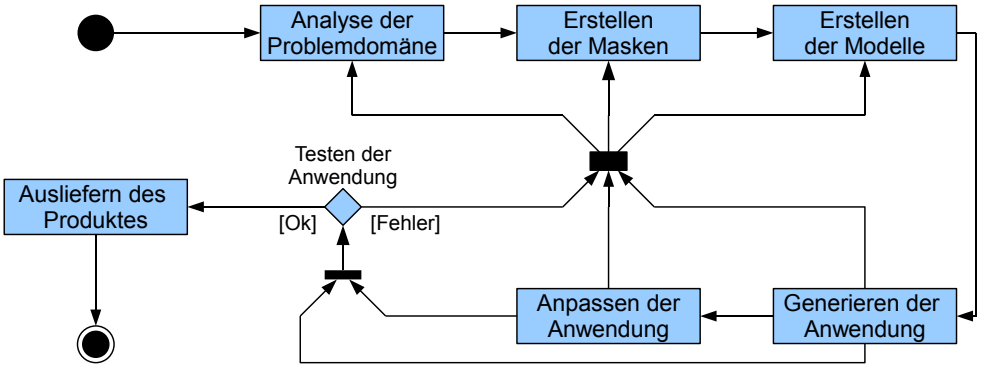


Abbildung 4.1: Der Entwicklungsprozess für die Anwendungsplattform „base“

4.1.1 Grundlagen

Vom Ansatz her ist die modellgetriebene Entwicklungsplattform „base“ verwandt mit den „Software Factories“, welche in „Microsoft Visual Studio“ Verwendung finden. (vgl.: [Sta07, S. 40]) Im Rahmen der „Software Factories“ werden dem Entwickler Kontext-sensitiv Assistenten zur Ausführung angeboten. So kann der Entwickler z. B. für eine Web-Anwendung den Workflow innerhalb der Anwendung oder für einen Web-Service die Schnittstelle zur Außenwelt modellgetrieben definieren. Diese Hilfestellung durch Assistenten wird durch die Entwicklungsplattform „base“ noch nicht unterstützt. Des Weiteren stehen nur EMF-Baumeditoren zum Bearbeiten der Modelle zur Verfügung.

Wie auch die „Software Factories“ wurde die Plattform „base“ dafür entwickelt, dem Entwickler einen Großteil der Arbeit abzunehmen und die Entwicklungszeit von Anwendungen – aufbauend auf einer fest definierten Plattform – zu verkürzen. Hierzu wurde „base“ soweit optimiert, dass für einzelne Anwendungen der gesamte Quellcode generiert werden kann. Erreicht wurde dies u. a. durch die mögliche Auslagerung von Berechnungen und anderer Programmlogik in die Definition von Ein- und Ausgabemasken. Hierzu werden Excel-Dateien verwendet, welche durch die Einführung von benannten Zellen und aktiven Steuerelementen für die Verwendung in „base“ vorbereitet werden. Auf dieser Basis können auch bereits vorhandene Excel-Dateien eines Auftraggebers wiederverwendet werden. Durch das Formatieren von Zellen kann der Excel-Datei das Aussehen eines normalen Programmes verliehen werden (z. B. schwarze Beschriftung auf grauem Hintergrund und weiße Eingabefelder mit schwarzem Rahmen). Funktionen wie z. B. die Summenbildung u. a. Formeln oder das Darstellen von Diagrammen aus den eingegebenen Daten werden direkt in Excel realisiert.

4.1.2 Modelle, Editoren und Generatoren

Um eine Anwendung zu erstellen, müssen neben den Masken in Form von Excel-Dateien diverse Modelle erstellt werden. Die im Rahmen dieses Entwicklungsprozesses verwendeten (Meta-)Modelle bauen auf dem EMF auf. Von der „b+m Informatik GmbH Berlin“ wurden die durch das EMF aus dem Metamodell generierten Baumeditoren (s. a. Kapitel 2.2.3) modifiziert, was sowohl das Verhalten als auch die visuelle Repräsentation betrifft. Alle Modifikationen wurden mit Hilfe von AspectJ und oAW realisiert. Das EMF generiert z. B. für jeden Typen im Metamodell eine `ItemProvider`-Klasse. Hier werden die Methoden `getText()` und `getIcon()` mit einem „Around-Advice“ überschrieben und über zwei Utility-Klassen `IconFinder` und `OAWFacade` mit einer zusätzlichen Funktionalität ausgestattet. In der Abbildung 4.2 werden die vorgenommenen Modifikationen am Beispiel der `getText()`-Methode dargestellt. Weitere Modifikationen betreffen die Auswahlmöglichkeiten bei Referenzen auf andere Modellelemente sowie Initialisierungen von neuen hinzugefügten Elementen mit Attributen und notwendigen Kindelementen. Durch die Verwendung der AOP konnte auch vermieden werden, dass der durch das EMF-Framework generierte Quellcode des Baumeditors verändert werden muss, was oft problematisch ist. (vgl.: [Sta07, S. 45])

Während der Generierung der Anwendung werden diverse Projekte für Eclipse angelegt, wobei zwischen dem Applikationsprojekt sowie den Modul- und den Persistenzprojekten unterschieden wird. Die einzelnen generierten Artefakte und die zugrundeliegende Laufzeitbibliothek sind hierbei auf einander abgestimmt.

In der Regel muss zum Erstellen einer Anwendung kein Sourcecode manuell entwickelt werden. Die vorhandenen Möglichkeiten zum Ausdrücken der Programmlogik decken die meisten Fälle ab.

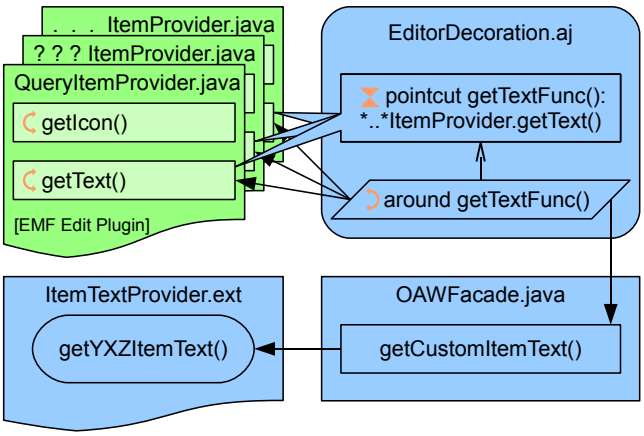


Abbildung 4.2: Schematische Darstellung eines Aspektes für die EMF-Baumeditoren

Sowohl die Laufzeitbibliothek als auch der Generator, der die Artefakte generiert, wurden seit dem Ende des Praktikums weiterentwickelt. Zum Beispiel wurde eine Umstellung der Benutzerschnittstelle von Eclipse mit dem Standard Widget Toolkit (SWT) auf reines Java mit Swing vorgenommen. Die Ursache war hierbei, dass SWT und die verwendete COM-Schnittstelle „jacoZoom“ (s. a. [UrlZos04]) nicht miteinander harmonierten. Ein weiterer Vorteil der reinen Java/Swing-Lösung ist, dass für eine Auslieferung der Anwendung zum Kunden keine Abhängigkeiten zu SWT und dem Eclipse-Projekt mehr bestehen.

4.2 Struktur der Metamodell-Pakete

In der modellgetriebenen Softwareentwicklung bilden Modelle die Grundlage jeder Entwicklung. Für „base“ wurde dieser Ansatz so weit entwickelt, dass es nur noch in Ausnahmefällen notwendig ist Sourcecode von Hand zu erstellen. Das in „base“ verwendete Metamodell mit insgesamt 113 Klassen teilt sich in 14 Pakete und Unterpakete auf. Diese Pakete werden kurz in der Tabelle 4.1 erläutert. Hier wird in der Spalte „Wurzelelement/ Modelltyp“ in der jeweils ersten Zeile das Element angegeben, welches für die Erstellung eines Modells dieses Metamodell-Paketes notwendig ist (siehe Kapitel 2.2.3). Die Pakete ohne angegebenes „Wurzelelement“ (dargestellt als „[–]“) sind zur Vereinfachung der gesamten Paketestruktur und nicht für die Verwendung als eigenständiger Editor erstellt worden. Die zweite Zeile derselben Spalte gibt an, ob es sich bei diesem Modell um ein präskriptives, d.h. vorschreibendes oder deskriptives, d.h. beschreibendes Modell handelt (siehe Kapitel 2.1.1). Die dargestellten Icons für die Elemente sind dem Baumeditor entnommen.

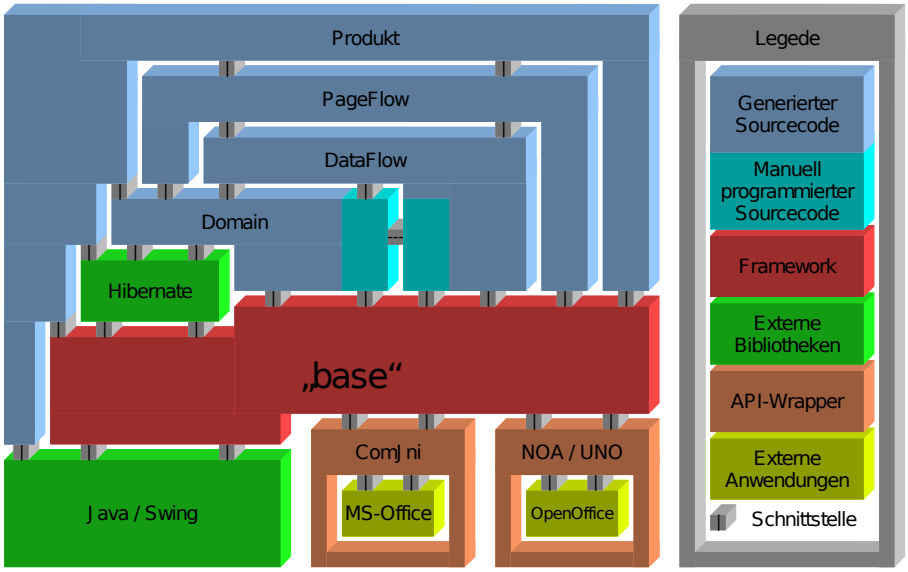
Alle Klassen des zugrunde liegenden Metamodells sowie deren Attribute, Referenzen und Abhängigkeiten sind in einer eigenen Referenzdokumentation [b+m08b] beschrieben. Diese Dokumentation wurde aus dem vorhandenen Metamodell mit dem, vom Verfasser dieser Diplomarbeit entwickelten, Eclipse-Plugin „MetamodelDoc“ [UrlSch07] erstellt.

Tabelle 4.1: Paketstruktur des Metamodells des Anwendungsframeworks „base“

Paket	Klassen/ Datentypen	Wurzelelement Modelltyp	Verwendungszweck
base	8 / 1	[–]	Container für alle Pakete sowie für einige, in verschiedenen Paketen benötigte, abstrakte Basisklassen und Schnittstellen.
basetype	6 / 0	`BaseTypes` (✦) deskriptiv	Definition von Java- (`Integer`, `Double`, `String`) und Base- (`Day`, `Month`, `Year`) spezifischen Datentypen.
domain	22 / 1	`Domain` (✦) prä-/deskriptiv	Definition von Entitäten der Domäne sowie von (parametrierbaren) Abfragen.
page	11 / 1	`Page` (✦) deskriptiv	Beschreibung von Maskenelementen. Das Modell wird automatisch aus den in Excel-Dateien enthaltenen Metainformationen erstellt.
dataflow	8 / 1	`DataFlow` (✦) präskriptiv	Datenfluss in einer Maske. Aufteilung in verschiedene Unterpakete für die verschiedenen Knotentypen.
nodes	5 / 0	[–]	Verarbeitung von Daten (z. B. `Filter` (✦), `Selector` (👁), `Setter` (⬇), `Forwarder` (↗)).
persistence	3 / 0	[–]	Abfrage einer im `domain`-Modell definierten Datenquelle (`Query` (🔍))
time	5 / 3	[–]	Verarbeitung von Zeit-, Datums- und Zeitraumangaben.
view	12 / 0	[–]	Ein- und Ausgabe für den Anwender (siehe `gui`-Modell)
model	14 / 1	[–]	Hilfsmodellelemente für das Mapping von Ein- (▫) und Ausgabeparametern (▫) mit benötigten Informationen für die Ein- und Ausgabe.
module	8 / 0	`Module` (✦) präskriptiv	Workflow innerhalb eines Moduls sowie Zugriffsberechtigungen auf die einzelnen Masken.
app	5 / 0	`Configuration` (✦) prä-/deskriptiv	Bündelt verschiedene Module zu einer Produktkonfiguration.
gui	2 / 1	`GuiElements` (✦) präskriptiv	Definition von zusätzlichen Elementen für die Benutzerführung (Toolbar-Buttons, Menüeinträge)
resources	4 / 0	`Resources` (✦) deskriptiv	Ermöglicht die Definitionen von Modellreferenzen auf Objekte des Dateisystems (Graphiken, Excel-Dateien).

[–] Nicht zutreffend

In der Abbildung 4.3 wird die Struktur einer fertigen Anwendung schematisch dargestellt. Zu erkennen sind hier die einzelnen generierten Komponenten der Anwendung „Product“, „PageFlow“, „DataFlow“ und „Domain“ (welche durch die Modelle `app`, `module`, `dataflow` und `domain` repräsentiert werden) sowie die Abhängigkeiten zwischen ihnen. Die evtl. anders liegenden Abhängigkeiten zwischen den Paketen sind in der Abbildung 4.4 dargestellt.



(Quelle: vgl.: Enrico Schnepel, Praktikumsbericht [Sch07, S. 5])

Abbildung 4.3: Der strukturelle Aufbau der Anwendungsplattform „base“

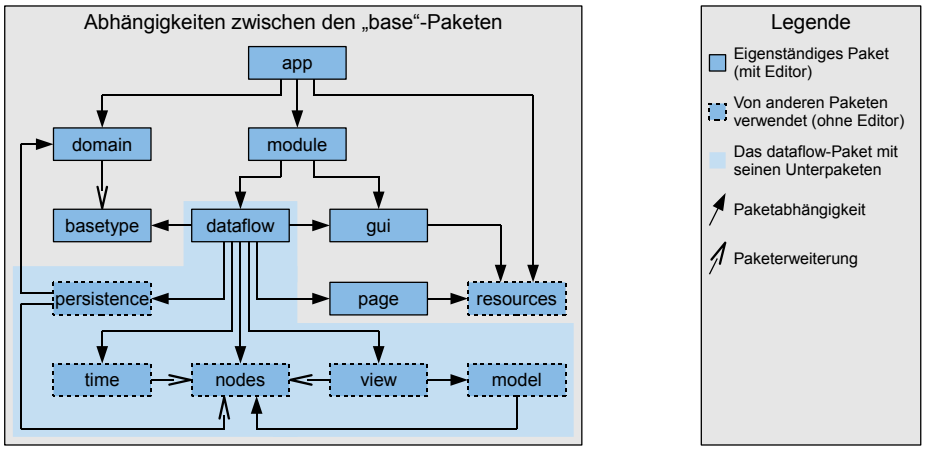


Abbildung 4.4: Die Abhängigkeiten zwischen den „base“-Modellen

In den folgenden Abschnitten werden zunächst die Pakete `base`, `basetype`, `domain` sowie `page` erläutert. Sie bilden die Grundlage für das Erstellen von `dataflow`-Modellen. Anschließends wird das `dataflow`-Paket mit den einzelnen Metamodelltypen ausführlicher erklärt, da im Rahmen dieser Arbeit der Fokus auf der Erstellung eines graphischen Editors für `dataflow`-Modelle liegt. Zum Schluss werden die restlichen Pakete `module`, `app`, `gui` sowie `resources` vorgestellt.

4.2.1 base

Das Hauptpaket des „base“-Metamodells `base` beinhaltet neben den Paketen für die einzelnen Modell-Editoren die Klassen `Named`, `Type`, `Typed` und `Identifier`, welche in den verschiedenen Unterpaketen Verwendung finden. `Named` enthält ein Plichtattribut „`name`“ vom Typ `String`. In den `Named`-Abwandlungen `NamedID` bzw. `NamedOptional` ist der Name modellweit ein-eindeutig bzw. optional. `Type` erbt von `Named`, während `Typed` eine „`type`“-Referenz auf den Typ `Type` sowie ein Attribut „`cardinality`“ mit der Aufzählung `Cardinality` (`ONE`, `MANY`) definiert. Der Typ `Identifier` vereint die beiden Typen `Named` und `Typed` mit Hilfe der in EMF erlaubten Mehrfachvererbung.

4.2.2 basetype

Viele Java-spezifischen Datentypen wie z. B. `String`, `Integer` und `Date` sind in einem Anwendungs-übergreifenden `basetype`-Modell definiert. Einige Plattform-spezifischen Typen wie `Day`, `Month` und `Year` werden ebenfalls auf die Implementierungsklassen abgebildet. Im Rahmen einer Anwendung kann ein eigenständiges `basetype`-Modell erstellt werden, welches die Anwendungsspezifischen Typen enthält.

4.2.3 domain

Das Paket `domain` definiert alle Modellelemente, die notwendig sind, um eine Datenbankdomäne zu beschreiben. Unterstützt werden Entitäten (`Entity`) und Beziehungen zwischen diesen (`Association`). Neben den Standarddatentypen aus dem `basetype`-Modell können Aufzählungen (`Enumeration`) modelliert werden. Außer dem Typ `Entity` existieren zwei weitere Sonderformen zum Abbilden eines Zeitpunktes (`ScheduledEntity`) sowie eines Zeitraumes (`EnduringEntity`). Alle verwendeten Typen der Attribute der verschiedenen Entitäten müssen entweder in einem `basetype`-Modell oder in Form einer `Enumeration` definiert sein.

Für jede Entität werden zusätzlich mögliche Abfragen auf den Datenbestand definiert. Standardmäßig wird für jede Entität eine `FindAllQuery` angelegt, welche alle Einträge zurückliefert. Für parametrierte Abfragen kann entweder eine `FindByMatchingAttributesQuery` oder eine `CustomHibernateQuery` verwendet werden. Die Parameter für die Abfrage werden bei ersterer über ein Attribut-Mapping modelliert. Für die `CustomHibernateQuery` müssen zusätzlich ein Hibernate-Query-String sowie alle in diesem verwendeten Parameter angegeben werden.

Da die meisten Referenzen im „base“-Metamodell für die Angabe eines Typs auf das Metamodell-Element `Type` verweisen und die drei Entitätentypen – wie auch `BaseType` – hiervon ableiten, lassen sich die Domänenelemente genauso wie native Datentypen in den Modellen verwenden.

4.2.4 page

Im Gegensatz zu den meisten anderen Modellen wird das `page`-Modell automatisch aus einer Excel-Datei erstellt. Hierbei werden die Namen der benannten Zellen und der Dialogelemente ausgewertet und nach einem festgelegten Namensschema bestimmten Klassen von Oberflächenelementen zugeordnet. Die Zuordnungsregeln sind in der Tabelle 4.2 dargestellt. Die einzelnen Namen sind mit einem Unterstrich in mehrere Bereiche getrennt. Hierbei wird der Teil bis zum ersten Unterstrich als Elementtyp bezeichnet. Nach dem ersten Unterstrich kommt der ElementName.

Tabelle 4.2: Zuordnungsregeln für das automatische Erstellen von `page`-Modellen

Typ	Startzeichenkette	resultierendes page-Modellelement
Zelle	`FIELD_`	`TextField`
Zelle	`TABLE_`	`Table`
Zelle	`MATRIX_`	`Matrix`
Dialogelement	`BUTTON_`	`Button`
Dialogelement	`COMBO_`	`ComboBox`
Dialogelement	`LISTFIELD_`	`ListField`
Zelle bzw. Dialogelement	`COMP_`[Typ]`_`	`Composite` Ein `Composite` Element fasst mehrere `TextField`s und `ComboBox`en zu einer logischen Einheit zusammen. Damit ist es möglich, in einem `dataflow`-Modell die Elemente als eine Einheit anzusprechen. Z. B. können so die Stammdaten eines Kunden in übersichtlicher Form dargestellt werden, ohne jedes Feld einzeln mit einem Attribut zu verknüpfen. Der Name des Objektes wird nach der veränderten Regel „`COMP_`[Typ]_[CompositeName]_[ElementName]“ gebildet.
Zelle	`COMP_FIELD_`	`TextField` in einem `Composite`
Dialogelement	`COMP_COMBO_`	`ComboBox` in einem `Composite`

4.2.5 `dataflow`

Dieses Kapitel geht auf das Modell ein, welches den Datenfluss innerhalb einer Excel-Datei beschreibt. Hierzu werden im Modell Elemete erstellt, die die einzelnen Verarbeitungsschritte spezifizieren. Zu den in der Tabelle 4.3 beschriebenen Elementen gehören neben den Knoten die interaktiven Oberflächenelemente. Die Knotentypen können zum einen nach der Art der in ihnen enthaltenen Informationen sowie nach der Funktion (A, B und C) gruppiert werden. Vorerst werden jedoch die für die Knoten obligatorischen Ein- (▫) und Ausgabeparameter (▫) näher erläutert.

Ein- (▫) und Ausgabeparameter (▫)

Grundsätzlich kann jeder Knoten beliebig viele benannte und typisierte Ein- (▫) und Ausgabeparameter (▫) (`InputParameter` (▫) bzw. `OutputParameter` (▫)) enthalten. Für die meisten Knotentypen ist der Typ und/oder die Zahl der Ein- (▫) und Ausgabeparameter (▫) sehr genau festgelegt. Elemente vom Typ `NodeConnection` (➔) verbinden jeweils einen Ein- (▫) mit einem Ausgabeparameter (▫), wobei der Typ und die Kardinalität der beiden Parameter genau übereinstimmen müssen. Mit einem Eingabeparameter (▫) kann jedoch nur eine einzige Verbindung verknüpft sein. Sollen Ausgabeparameter (▫) von mehreren Knoten mit einem Eingabeparameter (▫) verknüpft werden, muss ein `Forwarder` ( )-Knoten verwendet werden. Dies wird z. B. benötigt, wenn es irrelevant ist, ob der Benutzer einen Kunden über den Namen sucht oder die Kundennummer direkt eingibt.

Tabelle 4.3: Übersicht über die einzelnen Elemente des Metamodell-Paketes `dataflow`

Elementtyp	Eingabe-parameterzahl	Eingabe-typ	Ausgabe-parameterzahl	Ausgabe-typ	Weitere Bedingungen	Funktionsbeschreibung
(A) Knotentypen zur Verarbeitung beliebiger Datentypen Die folgenden Knoten sind, soweit nicht anders angegeben, im Metamodell-Paket `base.dataflow.nodes` definiert.						
`Creator`	0		1	*1		Erstellt ein Objekt vom Typ des Ausgabeparameters (▫).
? `Custom`	?		?			Benutzerdefinierte Verarbeitung in einer Implementierungsklasse
`Forwarder`	1...n		1		*3, *4	Bündelt den Datenfluss; es werden immer die „aktuellsten“ Daten verwendet
`Selector`	1	*2	1	*2		Selektiert eine Spalte aus den Daten
`Setter`	1		1		*3, *4	Daten des angegebenen Attributes im ersten Parameter setzen
	1	*1	[–]		Attributtyp	
`Query` (Definiert im Metamodell-Paket `base.dataflow.nodes.persistence`)	0...n		1			Datenbankabfrage – Eingabeparameter (▫)-Mapping im `QueryNodeMappingModel` (⇄)
(B) Knotentypen zur Verarbeitung von Datumsangaben und Zeiträumen Die folgenden Knoten sind im Metamodell-Paket `base.dataflow.nodes.time` definiert.						
`CurrentDate`	0		1	Date*1		Aktuelles Datum
`DateToDate`	1	Date	1	Date	*3, *4	einfache Datumsoperationen
`DateToRange`	1	Date*1	2	Date*1		Zeitraum eines Datums ermitteln
`RangeToDates`	2	Date*1	1	Date*2		Zeitraum aufteilen
`DateToDates`	1	Date*1	1	Date*2	*4	`DateToRange` (DR) ⇉ `RangeToDates` (RD)
(C) Knotentypen zum Anzeigen und Bearbeiten von Daten Die folgenden Knoten sind im Metamodell-Paket `base.dataflow.nodes.view` definiert und enthalten jeweils eine `widgetRef`-Referenz zu einem gleichnamigen Element (Ausnahme: ✦ `ClassifierTable` ⇒ `Table`) aus einem `page`-Modell.						
`TextField`	1		1			Einfaches Eingabefeld
`ComboBox`	1	*2	1	*1	*4	Drop-Down-Liste
✦ `ListField` *5	1	all *2	[–]			Auswahlliste – Mehrfachauswahl je nach Cardinalität des Ausgabeparameters (▫)
	1	sel *2	1	out		
`Table`	1	*2	1	*2	*4	Tabellarische Listendarstellung – Spaltendefinition im `TableModel` (✦), Setzen von Referenzen für neue Elemente mit dem `ViewNodeMappingModel` (⇄)
✦ `ClassifierTable`	1	*2	1	*2	*4	wie `Table` (T), zusätzlich mit einer festgelegten Gruppierung auf einer Achse im `ClassificationModel` (📷)
`Matrix` *6	2-3	*2	0			ähnlich `ClassifierTable` (✦), jedoch erfolgt die Klassifikationen auf beiden Achsen im `MatrixClassificationModel` (✦), Setzen von Referenzen für neue Elemente mit dem `ViewNodeMappingModel` (⇄)
✦ `Composite`	1	*1	1	*1	*4	Darstellen der Attribute eines Datensatzes in verschiedenen Feldern – Festlegen der darzustellenden Daten im `CompositeModel` (✦), Setzen von Referenzen für neue Elemente mit dem `ViewNodeMappingModel` (⇄)
Interaktive Oberflächenelemente Die folgenden Elemente sind im Gegensatz zu den bisherigen Knoten vom Typ `Event` abgeleitet und ebenfalls im Metamodell-Paket `base.dataflow.nodes.view` definiert. Durch eine `NodeEventConnection` (⚡) ermöglichen sie es, die Verarbeitung eines Knotens interaktiv zu steuern.						
`Button`	[–]		[–]			Button in der Maske
✦ `ToolbarButton`	[–]		[–]			Button in der Toolbar

*1 Der Parameter darf keine Liste enthalten. („Cardinality = ONE“) — *2 Der Parameter muss eine Liste enthalten. („Cardinality = MANY“) — *3 Gleichheit für die Kardinalität der Ein- (▫) und Ausgabeparameter (▫) — *4 Typengleichheit der Ein- (▫) und Ausgabeparameter (▫) — *5 `ListField` (✦) enthält zwei Eingabeparameter (▫); „all“ definiert die darzustellenden Einträge, „sel“ die ausgewählten; Der Ausgabeparameter (▫) „out“ enthält die veränderte Auswahl; Alle Ein- (▫) und Ausgabeparameter (▫) müssen den gleichen Typ haben — *6 `Matrix` (M) enthält zwei oder drei Eingabeparameter (▫); Die Klassifikation der Daten in Spalten und Zeilen erfolgt im `MatrixClassificationModel` (✦) — [–] nicht zutreffend

Gruppierung nach der Art der in den Knoten enthaltenen Informationen

Einige Knotentypen können neben den Ein- ( ) und Ausgabeparametern ( ) weitere `Model`-Kindelemente enthalten, die das Verhalten der „base“-Anwendung im Kontext des Knotens spezifizieren. Es können auch Attribute für die Spezifikation des Verhaltens angegeben werden. Diese beiden Varianten sind kreuzweise miteinander kombinierbar, wodurch vier Gruppen entstehen.

Es existieren insgesamt fünf Knotentypen, die mindestens ein zusätzliches Kindelement aufweisen können. Bei den Kindelementen wird oft ein Mapping (eine Zuordnung) modelliert. Der `Query`-Knoten ( ) kann z. B. ein `QueryNodeMappingModel` ( ) enthalten, in dem angegeben wird, welcher `QueryParameter` ( ) aus welchem Eingabeparameter ( ) befüllt wird.

Mit dem `Selector`-Knoten ( ) können Attribute aus einem Eingabeparameter ( ) ermittelt werden, um sie später in anderen Knoten weiterverarbeiten zu können. Der `Selector` ( ) benötigt hierfür eine Referenz auf das zu selektierende Attribut des Eingabeparameters ( ).

Gruppierung nach der Funktion der Knoten

Das dieser Arbeit zugrunde liegende `dataflow`-Metamodell-Paket enthält insgesamt:

(A) sechs Knotentypen zur Verarbeitung beliebiger Datentypen,
(B) fünf Knotentypen zur Verarbeitung von Datumsangaben und Zeiträumen sowie
(C) sieben Knotentypen zum Anzeigen und Bearbeiten von Daten.

Die verschiedenen Knotentypen sind auf unterschiedliche Unterpakete des `dataflow`-Metamodells verteilt.

(A) Knotentypen zur Verarbeitung beliebiger Datentypen

Viele der in einer Anwendung benötigten internen Verarbeitungsschritte lassen sich auf einige wenige elementare Funktionen reduzieren. Hierzu stehen die im Metamodell-Paket `dataflow.nodes` enthaltenen Knotentypen `Creator` ( ), `Custom` (?), `Forwarder` ( ) sowie `Selector` ( ) und `Setter` ( ) zur Verfügung. Durch das Kombinieren von Knoten können die für die Anwendung benötigten komplexen Funktionalitäten zusammengestellt werden. Im ersten Abschnitt der Tabelle 4.3 sind die genannten Knotentypen sowie zusätzlich der Knotentyp `Query` ( ) aus dem Metamodell-Paket `dataflow.nodes.persistence` aufgeführt.

Soll z. B. ein Kunde anhand einer Kundennummer oder eines Namens in der Datenbank gefunden werden können, müssen die beiden Eingabedaten an zwei separate parametrierte `Query`-Knoten ( ) weitergegeben werden. In einem `Forwarder`-Knoten ( ) können die beiden zurückgegebenen Listen aus den Abfragen so kombiniert werden, dass die zuletzt eingegangene Liste und damit auch die letzte Datenbankabfrage an seinen Pflicht-Ausgabeparameter ( ) weitergegeben wird. Zum Beispiel kann nun in der darauf folgenden Verarbeitung in einer `ComboBox` ( ) die Liste angezeigt, und falls notwendig der gewünschte Kunde ausgewählt werden.

(B) Knotentypen zur Verarbeitung von Datumsangaben und Zeiträumen

Die Verarbeitung von Datumsangaben wird im Rahmen der Softwareentwicklungsumgebung „base“ durch das `dataflow`-Modell explizit unterstützt (s. a. die Elemente `ScheduledEntity` ( ) und `EnduringEntity` ( ) des `domain`-Modells im Kapitel 4.2.3). Mit den fünf zur Verfügung stehenden Modellelementen können viele Operationen definiert werden, die in der Praxis benötigt werden. Ein Datum wird hierzu in verschiedene virtuelle Felder (Enumeration `DateField`: Tag

= DAY, Monat = MONTH, Quartal = QUARTER und Jahr = YEAR) unterteilt, auf denen Operationen durchgeführt werden können.

Das einfachste Element CurrentDate dient zum Ermitteln des aktuellen Datums. Mit dem Element DateToDate können Operationen auf dem durch den Eingabeparameter gelieferten Datum durchgeführt werden. Hierbei wird mit dem Attribut operation grundsätzlich zwischen dem Setzen (SET) eines Feldes des Datums (Attribut field) auf den durch das Attribut parameter angegebenen Wert und dem Addieren (ADD) unterschieden. Durch die Angabe des DateOperationMode im Attribut mode wird angegeben, ob hierbei kleinere Felder beibehalten (KEEP_LOWER_FIELDS), minimiert (MINIMIZE_LOWER_FIELDS) oder maximiert (MAXIMIZE_LOWER_FIELDS) werden sollen.

Beispiel: DateToDate
Für das aktuelle Tagesdatum (durch einen CurrentDate-Knoten geliefert) soll der erste Tag des nächsten Quartals ermittelt werden. Ein durch eine NodeConnection verknüpfter DateToDate-Knoten muss hierzu die Angaben operation = ADD, field = QUARTER, parameter = 1 und mode = MINIMIZE_LOWER_FIELDS enthalten.

Die Elemente DateToRange, RangeToDates sowie DateToDates sind für die Verarbeitung von Zeiträumen (Range) gedacht. Zeiträume werden hierbei immer durch ein Paar von Ein- oder Ausgabeparametern mit dem Typ Date angegeben. Mit dem Typ DateToRange kann ein Datum durch die Angabe eines DateField sowie Werten für die Attribute from und to definiert werden, welchen Umfang der Zeitraum beinhalten soll. Im Gegensatz dazu erlaubt es RangeToDates, aus einem Zeitraum mehrere Starttermine in den im Attribut field als DateField angegebenen Abständen zu ermitteln. Der Typ DateToDates kombiniert die Typen DateToRange und RangeToDates, wobei die Angabe der Ein- und Ausgabeparameter sowie der NodeConnections zwischen den Typen entfällt.

(C) Knotentypen zum Anzeigen und Bearbeiten von Daten

Zu dieser Kategorie von Knotentypen zählen neben TextField, ComboBox, ListField und Composite die für tabellarische Darstellungen verwendeten Knotentypen Table, ClassifierTable und Matrix. Jeder Knoten muss hierbei ein passendes Element aus dem page-Modell referenzieren. Hierbei enthalten das Composite-Element sowie die für Tabellen zuständigen Knoten Informationen in zusätzlichen Kindelementen.

Ein für die Table und ClassifierTable verwendetes Kindelement ist das TableModel, in dem mit weiteren ColumnModel-Kindern die anzuzeigenden Spalten angegeben werden. Die Reihenfolge für die Anzeige in der Tabelle definiert sich hierbei über die Reihenfolge im Modell.

Interaktive Oberflächenelemente

Interaktive Oberflächenelemente (ToolbarButton oder Button) steuern die Verarbeitung in einem dataflow-Modell aktiv. Sie implementieren das Metamodell-Interface Event und sind damit technisch gesehen keine Knoten. Aus diesem Grund enthalten sie auch keine Ein- oder Ausgabeparameter.

Um eine Verarbeitung eines Knotens aktiv zu steuern, muss eine NodeEventConnection ein Event-Objekt mit dem Knoten verbinden. Durch diese Verbindung wird der Knoten beim Klicken des Button aktiviert und verarbeitet die Daten der Eingabeparameter. Liegen an einem Knoten

keine `NodeEventConnections` ( ) an, werden die Daten verarbeitet, sobald sich ein Eingabeparameter ( ) ändert.

Die aktive Steuerung kann dazu verwendet werden, um z. B. eine Kundensuche nach einem Namen zu implementieren. Hierbei muss eine mit dem Namen parametrierte `Query` ( ) aktiv gesteuert werden.

4.2.6 `module`

Mehrere `dataflow`-Modelle werden im `module`-Modell zu einem Modul zusammengefasst. Modelliert werden hier Übergänge zwischen den einzelnen darzustellenden Seiten ähnlich einer „Statemachine“. Das Modell wird angereichert mit benötigten Rechten, die ein Benutzer haben muss, um eine bestimmte Seite aufzurufen. Zur Laufzeit werden diese Rechte automatisch in das evtl. schon vorhandene Rechtesysten in der Datenbank integriert.

4.2.7 `app`

Das `app`-Modell fasst mehrere `module`-Modelle zusammen und definiert somit eine Konfiguration für die Anwendung. Es lassen sich auf diese Weise einfach – entsprechend den Wünschen des Kunden – weitere Module hinzufügen. Des Weiteren ist eine Datenbankkonfiguration (`PersistenceContext` ( )) hinterlegt.

4.2.8 `gui`

Mit dem `gui`-Modell können zusätzliche Bedienelemente eines Modules beschrieben werden. Ein `GuiElement` ( ) muss dabei zur Anzeige im Menü (`Menu`), in der Toolbar (`Button`) oder beidem (`MenuAndButton`) konfiguriert werden. Der Eintrag kann z. B. von einem `ToolbarButton` ( ) eines `dataflow`-Modells referenziert werden. Von der generierten Anwendung werden entsprechende Menü- und ToolbarButton-Einträge erstellt und mit einer entsprechenden Aktion belegt.

4.2.9 `resources`

Ähnlich dem `page`-Modell wird auch das `resources`-Modell automatisch erzeugt. Es enthält die Bilder und Graphiken eines Verzeichnisses. So ist es dann z. B. möglich, aus den `gui`- oder `module`-Modellen heraus Icons für bestimmte Oberflächenelemente zu referenzieren. Durch diesen modellgetriebenen „Umweg“ werden Tippfehler bei der Eingabe des Bild-Dateinamens vermieden und es können nur vorhandene Bilder ausgewählt werden.

KAPITEL 5

Analyse und Pflichtenheft

Das Ziel dieser Diplomarbeit ist die Ablösung des bestehenden `dataflow`-Editors für das Projekt „base“. Zunächst steht eine Problemanalyse des alten Editors im Vordergrund. Zusätzlich werden die für in modellgetriebenen Softwareentwicklungprozessen eingesetzte DSL-Editoren vorteilhafte Funktionalitäten beschrieben. Hierauf aufbauend enthält der Abschnitt 5.2 das Pflichtenheft mit den für den Editor wichtigen Funktions- und Abgrenzungskriterien.

5.1 Analyse

Im Rahmen der Softwareentwicklungsumgebung „base“ werden zur Zeit manuell angepasste Baumeditoren verwendet, die mit dem EMF aus dem zugrunde liegenden Metamodell generiert wurden. (siehe Kapitel 4.1.2) Die Modelle `domain`, `dataflow` sowie `page` enthalten einen gerichteten, zyklischen Graphen im mathematischen Sinn. Eine Baumstruktur kann jedoch einen Graphen nicht in einer adäquaten Art und Weise darstellen. Haller schreibt hierzu in seiner Diplomarbeit über „Mappingverfahren zur Wissensorganisation“:

> **Zitat:**
> Ein gutes System zur Unterstützung von Lernen und Wissensorganisation sollte seine Inhalte auf eine Weise darstellen, die der mentalen Repräsentation dieser Inhalte entgegenkommt.
> [Hal02, S. 7]

5.1.1 Der `dataflow`-Baumeditor

Im `dataflow`-Modell fällt die Bearbeitung des Modells in einem Baumeditor besonders schwer, da die Anzahl der Objekte und Verbindungen im Modell relativ zu den anderen beiden Modellen (`domain` und `page`) höher ist. Beim Modellieren einer Anwendung im Rahmen einer modellgetriebenen Softwareentwicklungsumgebung muss der Entwickler jedoch das Modell verstehen, um die Anwendung entwickeln zu können. Ein Baumeditor unterstützt diesen Prozess nur ungenügend, denn in der Darstellung als Baum werden die Verbindungen zwischen den einzelnen Knoten ebenfalls als eigene Modellelemente und auf der gleichen Ebene dargestellt. Zur Verdeutlichung sind in der Abbildung 5.1 die Knoten mit einer roten und die Kanten mit einer blauen Umrandung markiert. Um das Modell verstehen zu können, muss der Entwickler die ausschließlich textuell dargestellten Verknüpfungen interpretieren und die Knoten anhand der Namen identifizieren. Dieser manuell textuelle Vergleich ist sehr zeitaufwändig und fehleranfällig. Daher soll der baumbasierte Editor im Rahmen dieser Diplomarbeit ersetzt werden.

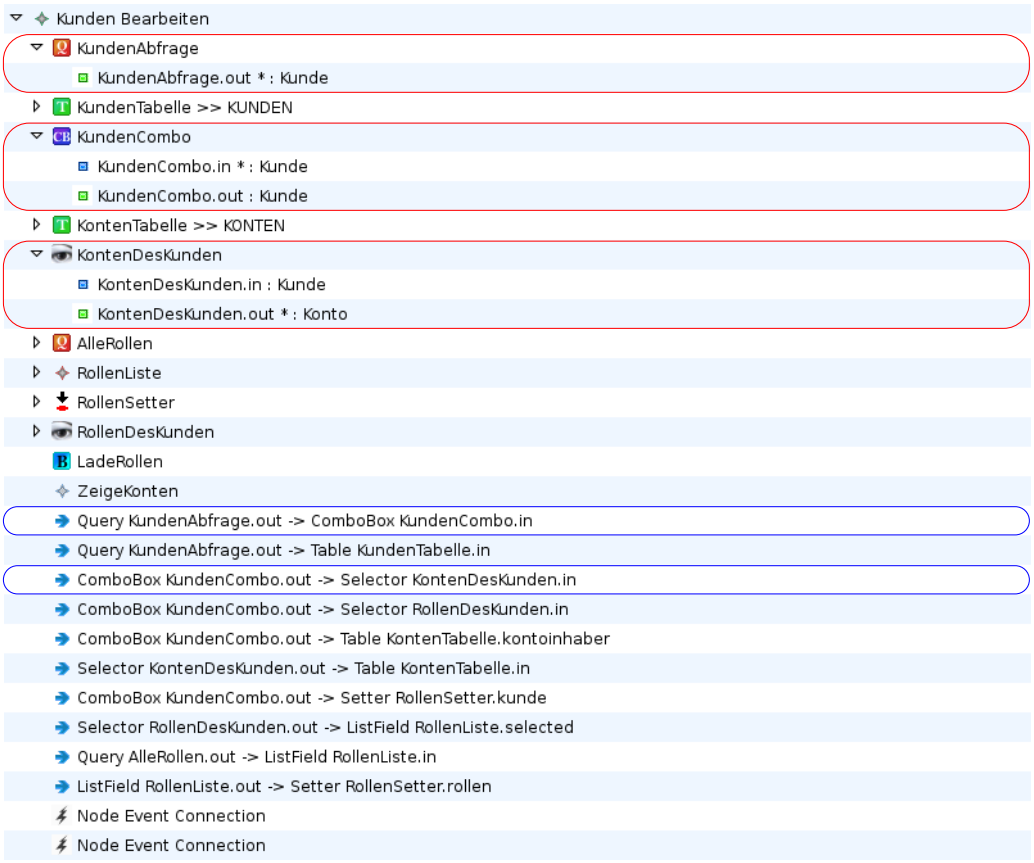


Abbildung 5.1: Der `dataflow`-Editor mit der Baumdarstellung des Modells

5.1.2 Editoren für Domänen-spezifische Sprachen

In den Abschnitten 2.1.5 bis 2.1.7 wurde bereits kurz auf die verschiedenen Möglichkeiten zur Erstellung von DSLs eingegangen. In der Tabelle 5.1 werden die verschiedenen Alternativen zur bisher verwendeten baumbasierten Notation des `dataflow`-Modells verglichen.

Grundsätzlich kann das Modell eines Graphen im mathematischen Sinne mit seinen Knoten und Verbindungen nur von einem graphischen Editor optimal modelliert werden.

5.1.3 Graphische Editoren

Die eigenständigen Programme „Microsoft Visual Studio 2005“, „MetaCase+“ sowie „MagicDraw“ werden für die Entwicklung des Editors nicht in Betracht gezogen, da die bereits existierende Softwareentwicklungsumgebung „base“ auf Eclipse und dem EMF aufbaut. Der Medienbruch, der mit einem eigenständigen Tool eingeführt werden würde, soll dadurch vermieden werden. Von den Eclipse-basierten Frameworks ist die anvisierte Entwicklung eines Editors mit dem GEF zu aufwändig, da der gesamte Editor manuell implementiert werden muss. Die beiden modellgetriebenen Ansätze für die Editor-Entwicklung, GMF und TOPCASED bieten fast die vollständige Funktionsvielfalt des GEF und kombinieren diese mit einer einfachen Definition der für graphische Editoren benötigten Konstrukte. In der Gegenüberstellung hat sich GMF als das ausgereiftere Werkzeug (vgl.: [Zim07, S. 1]) herausgestellt. Da außerdem kompetente Ansprechpartner seitens der betreuenden Firma verfügbar sind, wird für den zu entwickelnden Editor daher eine Lösung mit dem GMF angestrebt. Die Konzepte des GMF wurden bereits im Kapitel 3 näher erläutert.

Tabelle 5.1: Vergleich der bisher verwendeten baumbasierten Notation des `dataflow`-Modells mit den verschiedenen Alternativen

Notation Umsetzung mit	Vorteile	Nachteile
baumbasiert EMF	• Keine Entwicklungsaufwände für den Editor, da er bereits vorhanden ist.	• Referenzen müssen kognitiv interpretiert werden. • Einige Informationen sind nur im Eigenschaftsfenster verfügbar.
textuell Xtext-Editoren	• Hierarchische Strukturen können einfach abgebildet werden. • Das Modell wird direkt editiert. • Es werden alle Informationen gleichzeitig angezeigt.	• Referenzen müssen kognitiv interpretiert werden. • Ein gezieltes Ausblenden von Informationen ist nicht möglich (außer Folding).
formularbasiert Eclipse-Forms (s. a. [Glo05] und [Ecl, S. 4f.])	• Der Entwickler wird durch den Modellierungsprozess geführt. • Es wird nur der kontextuell interressante Ausschnitt des Modells dargestellt.	• Die gleichzeitig dargestellte Informationsmenge ist begrenzt. • Es gibt keinen Überblick über das gesamte Modell.
graphisch GEF/GMF oder TOPCASED	• Es gibt einen Überblick über das gesamte Modell. • Referenzen können als Verbindungen zwischen den Elementen dargestellt werden.	• Die Editorentwicklung ist sehr aufwändig. • Einige Informationen sind nur im Eigenschaftsfenster verfügbar.

5.1.4 Modellvalidierung

Die Benutzung von Modelleditoren in der modellgetriebenen Softwareentwicklung ist eng an die Möglichkeiten der integrierten Modellvalidierung geknüpft. Der Generator der Entwicklungsumgebung „base“ kann als eine Art Kompiler angesehen werden, der z. B. ein `dataflow`-Modell in ein anderes Modell (z. B. Java-Quellcode und andere Artefakte) überführt. Da dem Benutzer nicht zugemutet werden kann, anhand von Fehlermeldungen des Java-Kompilers und Laufzeitfehlern auf Entwicklungsfehler im Modell zurückzuschließen, muss der Anwender rechtzeitig auf Fehler im Modell hingewiesen werden. Mit Regelsätzen können syntaktische, semantische und logische Fehler in den Modellen gefunden werden (vgl.: [Sta07, S. 60f.]). In der Tabelle 5.2 sind für die genannten Fehlerkategorien Beispiele und Möglichkeiten zur Erkennung aufgeführt.

Tabelle 5.2: Mögliche Fehlerarten bei der Entwicklung von Modellen mit Beispielen aus dem Kontext des `dataflow`-Modells

Fehlerart	Beispiel	Erkennung
syntaktisch (Struktur)	• Es wird ein Knoten als Kindelement eines anderen Knotens erstellt. • Ein `NodeConnection`-Element ( ) ist über die `from`-Referenz mit mehreren Ausgabeparametern ( ) verknüpft.	Durch die Verwendung des EMF können struktuelle syntaktische Fehler nicht auftreten. Die strukturelle Integrität des Modells wird bereits durch das Framework sichergestellt.
syntaktisch (Pflichtangaben)	• Ein `NodeConnection`-Element ( ) ist über die `from`-Referenz mit keinem Ausgabeparameter ( ) verknüpft.	Das Metamodell enthält Mindestanzahlen für Referenzen und Attribute. Bei der Generierung der Plugins für den Baumeditor werden Tests mitgeneriert, welche die Pflichtangaben abprüfen. Durch das Aufrufen des „Validate“-Eintrages aus dem Kontextmenü werden die Tests durchlaufen und das Ergebnis dem Entwickler präsentiert.
semantisch (Kompilierungsfehler)	• Ein `Creator` ( )-Knoten besitzt keinen Ausgabeparameter ( ). Da jeder Knoten 0...n Ausgabeparameter ( ) enthalten kann, wäre dieser `Creator` ( )-Knoten syntaktisch korrekt. Für den `Creator` ( )-Knoten ist jedoch genau ein Ausgabeparameter ( ) vorgeschrieben (siehe Kapitel 4.3). Hierdurch wird der Typ des zu erstellenden Elementes festgelegt.	Semantische Fehler können durch Tests, z. B. duch die Verwendung des Check-Frameworks, gefunden und dem Entwickler an geeigneter Stelle präsentiert werden. Für die Erstellung der Tests ist eine sehr gute Kenntnis des zu testenden Typs notwendig.
logisch (Modellierungsrichtlinien, Laufzeitfehler)	• Der Ausgabeparameter ( ) eines `Creator` ( )-Knotens hat keine verknüpfte `NodeConnection` ( ) zu einem Eingabeparameter ( ) eines anderen Knotens. In diesem Fall würde ein Element erstellt, dann jedoch nicht weiterverarbeitet werden.	Wie auch für die semantischen können für logische Fehler Regelsätze (z. B. mit dem Check-Framework) erstellt werden. Aufgrund der Komplexität des Gesamtsystems und der Tatsache, dass das `dataflow`-Modell Anwendungsverhalten und Algorithmen beschreibt, kann dieses Regelwerk jedoch nie vollständig sein. (vgl.: [Rot01, S. 47])

5.2 Pflichtenheft

Bisher wurden mit dem Baumeditor Verknüpfungen zwischen zwei Knoten für den Benutzer nicht adäquat dargestellt. Es ist daher eine Mindestanforderung, dass der zu entwickelnde graphische Editor Knoten und Kanten in einer kognitiv besser verständlichen Form dem Benutzer präsentiert.

Um den Entwickler bei der Arbeit mit den verschiedenen Modellen bestmöglich zu unterstützen, soll der Editor den folgenden vom Verfasser der Arbeit definierten Pflicht-, Kann-, Wunsch- und Abgrenzungskriterien genügen.

5.2.1 Pflichtkriterien

R1 Alle Knoten sollen als Rechteck dargestellt werden.

R2 Die Hintergrundfarbe soll für jeweils alle Knoten einer Gruppe (siehe Tabellen 4.3 und 5.3) gleich sein.

R3 Die Ein- ( ) und Ausgabeparameter ( ) sollen in der Darstellung den Knoten direkt zuzuordnen sein. Dies soll über die Verwendung von Ports geschehen, wodurch die Parameter ganz oder teilweise außerhalb des Knotens dargestellt werden.

R4 Die beiden Verknüpfungstypen (`NodeConnection` ( ) und `NodeEventConnection` ( )) sollen in ihrer Darstellung unterscheidbar sein.

R5 Icons, die bereits im EMF-Baumeditor angezeigt werden, sollen auch im graphischen Editor innerhalb der Knoten dargestellt werden.

R6 In den Knoten soll der Typname – ähnlich den Stereotypen in der UML – in Guillemots („«`...`»“) sowie der vom Entwickler vergebene Name dargestellt werden.

R7 Neben dem Namen des Ein- ( ) oder Ausgabeparameters ( ) sollen die Kardinalität sowie der Typ angezeigt werden.

R8 Der graphische Editor soll das einfache Bearbeiten des gesamten Modells (Knoten und deren Kindelementbäume sowie die Verbindungen; siehe Tabelle 4.3) ermöglichen.

R9 Der Editor muss mit einer integrierten Modellvalidierung ausgestattet werden.

R10 Die Modellvalidierung (R9) soll alle durch das EMF zur Verfügung gestellten Tests für das Modell aufrufen.

R11 Die Tests, welche bereits im Rahmen der Modellvalidierung für den Baumeditor verwendet werden, sollen auch für den graphischen Editor genutzt (R9) werden.

R12 Es sollen zusätzliche Tests für die in der Tabelle 4.3 definierten Bedingungen entwickelt und in die Modellvalidierung (R9) integriert werden.

5.2.2 Kannkriterien

R13 Die Ein- ( ) und Ausgabeparameter ( ) sollen sich relativ zum Knoten nur auf der oberen/linken bzw. rechten/unteren Seite platzieren lassen, um den Verarbeitungsfluss für die Daten besser zu verdeutlichen. Dies soll auch das bessere Verständnis für das Modell fördern, da es der im Deutschen üblichen Leserichtung entspricht.

R14 Die Modellvalidierung (R9) soll in Echtzeit erfolgen, also während der Entwickler das Modell bearbeitet, und ohne einen expliziten Aufruf.

5.2.3 Wunschkriterien

R15 Für die Elemente des Metamodells, denen bisher nur ein vom EMF generiertes Icon in Form einer Raute zugeordnet ist, sollen Icons erstellt werden. Hierdurch soll der Wiedererkennungswert der Elemente im Editor erhöht werden. (s. a. R5)

R16 Für die einzelnen Knotentypen sollen zusätzlich zum Namen und dem Typ (R6) die in der Tabelle 5.3 aufgelisteten Inhalte gezeigt werden. Beispielhaft ist dies für den Typ `Query` ( ) in der Abbildung 5.2 schematisch dargestellt.

5.2.4 Abgrenzungskriterien

X1 Frameworks zur Entwicklung graphischer Editoren, die nicht auf Eclipse basieren, werden nicht in Betracht gezogen, da die vorhandene Softwareentwicklungsumgebung auf Eclipse aufbaut und kein Medienbruch eingeführt werden soll.

X2 Die Entwicklung des Editors findet ausschließlich auf einem Referenzsystem (Linux; Kubuntu 8.04) mit Java 1.5 und Eclipse (Version 3.3.X; Europa) statt.

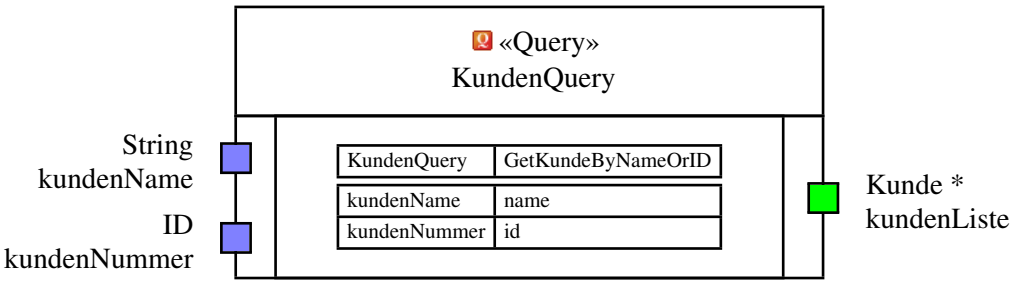
X3 Für die Entwicklung des graphischen Editors auf der Basis von Eclipse wird nur das GMF in Betracht gezogen, da sich das GMF als das stabilere Werkzeug herausgestellt hat und kompetente Ansprechpartner zur Verfügung stehen (siehe Kapitel 5.1.3).

X4 Der graphische Editor wird für die Verwendung als Eclipse-Plugin entwickelt. Die Verwendung des Editors als eigenständige Rich Client Platform (RCP)-Anwendung wird nicht in Betracht gezogen, auch wenn dies durch das GMF unterstützt wird.

X5 Auch wenn die Softwareentwicklungsumgebung „base“ noch keinen produktiven Entwicklungsstand erreicht hat, soll im Rahmen dieser Diplomarbeit nur der graphische Editor entwickelt werden.

X6 Von der in X5 definierten Regel soll nur dann abgewichen werden, wenn es für die Entwicklung des graphischen Editors als vorteilhaft erscheint.

In der Abbildung 5.2 ist eine schematische Darstellung eines `Query`-Knotens ( ) zu sehen. Die Implementierung für den graphischen Editor sollte sich an den genannten Kriterien (R1 bis R16 sowie X1 bis X6) und der gezeigten Figur orientieren.



* Das Sternchen drückt die Kardinalität `MANY` aus und soll auch so angezeigt werden.

Abbildung 5.2: Darstellungsvariante für einen Datenverarbeitungsknoten

Tabelle 5.3: Übersicht über die einzelnen Knotentypen

Elementtyp	Inhalt der Figur
Knotentypen zur Verarbeitung beliebiger Datentypen	
`Creator`	Typ des Ausgabeparameters (▫)
`Custom`	[–]
`Forwarder`	Typ und Kardinalität des Ausgabeparameters (▫)
`Selector`	verwendetes Attribut
`Setter`	Typ und Kardinalität des Ausgabeparameters (▫), Typ und Name des verwendeten Attributes
`Query`	Name der verwendeten `FindAllQuery` (Q) aus der `Entity` (E)-Typdefinition des Ausgabeparameters (▫), Mapping zwischen den Eingabe- (▫) und Queryparametern (▫) (s. a. Abbildung 5.2)
Knotentypen zur Verarbeitung von Datumsangaben und Zeiträumen Die Inhalte für die Knoten sind in einer „Quasi"-Syntax angegeben. Die einzelnen Elemente und Attribute sind im Kapitel 4.2.5/„Knotentypen zur Verarbeitung von Datumsangaben und Zeiträumen" näher erläutert.	
`CurrentDate`	[–]
`DateToDate`	[DateOperation (add/ set)] [parameter (Integer)] to [DateField (Year/ Quarter/ Month/ Day)] / [DateOperationMode (Keep/ Minimize/ Maximize lower Fields]
`DateToRange`	[DateField (Year/ Quarter/ Month/ Day)] [from]. . . [to]
`RangeToDates`	[DateField (Year/ Quarter/ Month/ Day)]
`DateToDates`	`DateToRange` (DR) ⇉ `RangeToDates` (RD) [DateField (Year/ Quarter/ Month/ Day)] [from]. . . [to]
Knotentypen zum Anzeigen und Bearbeiten von Daten Es wird immer der Name des referenzierten Elementes aus dem page-Model dargestellt. Zusätzliche Angaben sind in der Spalte „Inhalt der Figur" aufgeführt.	
`TextField`	verwendetes Attribut des Eingabeparametertyps, veränderbar ja/nein
`ComboBox`	verwendetes Attribut für die Darstellung
`ListField`	verwendetes Attribut für die Darstellung
`Table`	Auflistung der Spalten mit den darzustellenden Attributen
`ClassifierTable`	Auflistung der Spalten mit den darzustellenden Attributen, Attribute, nach denen klassifiziert werden soll
`Matrix`	Attribute, nach denen X-/Y-klassifiziert werden soll, Ergebnisfunktion
`Composite`	Mapping zwischen den Attributen und den im Composite verwendeten Eingabefeldern
Oberflächenelemente zur Interaktion	
`Button`	Name des referenzierten Buttons aus dem `page`-Modell
`ToolbarButton`	Name des referenzierten Buttons aus dem `gui`-Modell

[–] Nicht zutreffend

5.2.5 Technische Anforderungen

In der Tabelle 5.4 sind die verwendeten Programm- und Pluginversionen der Entwicklungsumgebung Eclipse sowie von Java dargestellt. Der Editor wird auf der Grundlage dieser Versionen entwickelt. Für die Funktionsfähigkeit des erstellten Editors mit anderen Versionen wird nicht garantiert. Vor allem in Bezug auf die kommenden Versionen des EMF (2.4) und des GMF (2.1) sind Inkompatiblitäten zu erwarten.

Tabelle 5.4: Mindestanforderungen an die Entwicklungsumgebung

Programm / Plugins	Name	Version
Programm	Java	1.5
Programm	Eclipse	3.3.3
Plugins	Eclipse Modelling Framework (EMF)	2.3.2
Plugins	Graphical Editing Framework (GEF)	3.3.2
Plugins	Graphical Modelling Framework (GMF)	2.0.2
Plugins	openArchitectureWare (oAW)	4.2

KAPITEL 6

Generieren von Modellen für das Graphical Modelling Framework

Im Rahmen der Diplomarbeit soll ein graphischer Editor für `dataflow`-Modelle entwickelt werden. Das `base`-Metamodell enthält 20 Knotentypen, die hierfür relevant sind. Wenn mit den Mitteln von GMF ein graphischer Editor erstellt wird, muss für jeden Knoten eine `TopNodeReference` () im „Mapping Def Model“ mit den entsprechenden Kindern angelegt werden. In Kapitel 5.2 wurde bereits beschreiben, dass der Name des Knotentyps Inhalt der Figur sein soll. Für die Umsetzung muss im „Graphical Def Model“ für jeden Knoten eine eigene, jedoch jeweils minimal unterschiedliche, Definition der visuellen Repräsentation erstellt werden. Eine Wiederverwendung ist hier nicht mehr möglich. Für die Ein- () und Ausgabeparameter () müssen im „Mapping Def Model“ jeweils `ChildReference`s () in den `NodeMapping`s () der Knoten angelegt werden (s. a. Abschnitt „Ports“ im Kapitel 3.5.3). Im Gegensatz zur Definiton der graphischen Elemente der Knotentypen ist für die Parameter eine Wiederverwendung möglich.

GMF-Modelle, in denen viele verschiedene Knotentypen vorkommen, erreichen schnell eine nur schwer zu handhabende Komplexität, wenn die Knoten zudem jeweils ein nur leicht unterschiedliches visuelles Erscheinungsbild haben sollen. Bei der Umsetzung der in Kapitel 5 beschriebenen Anforderungen wurde sehr schnell klar, dass die manuelle Bearbeitung der GMF-Modelle für den graphischen `dataflow`-Editor ungeeignet ist.

Um die Entwicklungszeit des Editors zu verkürzen und eine Wiederverwendung für die GMF-Modelle „Graphical Def Model“ und „Mapping Def Model“ einzuführen, hat der Verfasser mit „GenGMF“ ein Metamodell mit einer passenden M2M-Transformation entwickelt, damit die GMF-Modelle modellgetrieben generiert werden können.

In den folgenden beiden Abschnitten wird zuerst eine Einführung in die verwendeten Konzepte sowie den verwendeten Editor gegeben. Anschließend werden mit den Templates und den Deskriptoren die grundlegenden Elemente des „GenGMF“-Modells näher erläutert. In Kapitel 6.5 werden die einzelnen Schritte der M2M-Transformation in die GMF-Modelle beschrieben. Der darauf folgende Abschnitt erklärt, wie Variabilität für die Elementtypen umgesetzt werden kann. Abschließend wird das Erläuterte in einem Beispiel angewendet.

6.1 „GenGMF“ – Ein Generator für GMF-Modelle

Der Begriff „GenGMF“ ist einerseits eine Wortschöpfung aus den Bestandteilen „Gen“ für Generator und der bereits eingeführten Abkürzung für das Graphical Modelling Framework (GMF) und andererseits ein Wortspiel mit Bezug auf das in GMF genutzte „`gmfgen`“-Modell („Diagram Gen Model“) aus dem Kapitel 3.6. Im GMF wird aus dem „`gmfgen`“-Modell der Editor generiert, während „GenGMF“ dazu dient, die GMF-Modelle zu generieren.

„GenGMF“ baut auf der Idee auf, Gemeinsamkeiten (engl. „commonality“) und Unterschiede (engl. „variability“) an unterschiedlichen Stellen zu erfassen bzw. zu modellieren, um die Komplexität insgesamt zu reduzieren. Für das „GenGMF“-Metamodell wurde – wie auch schon für das „base“-Metamodell – eine Dokumentation (siehe [Sch08b]) mit dem Tool MetamodelDoc ([UrlSch07]) erstellt.

In den Abbildungen 6.1 und 6.2 ist der im Vergleich zum Kapitel 3.1 und den Abbildungen 3.1 und 3.2 auf der Seite 14 veränderte Entwicklungsprozess für graphische Editoren dargestellt. Hierbei wird weiterhin der eigentliche Editor von den GMF-Generatoren anhand der in den Modellen hinterlegten Spezifikation generiert. Für die Erstellung der Modelle „Graphical Def Model“ und „Mapping Def Model“ wird, statt des in GMF enthaltenen Wizards oder der manuellen Bearbeitung, eine „GenGMF“-spezifische M2M-Transformation benutzt.

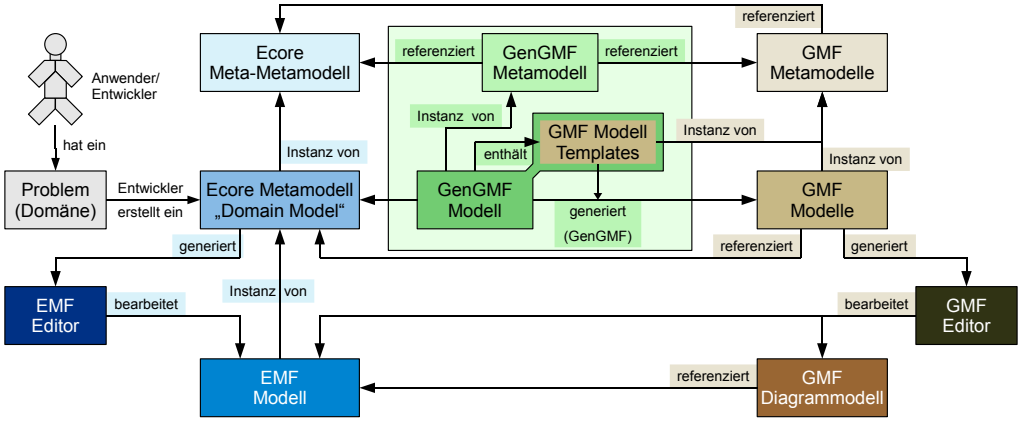


Abbildung 6.1: Vereinfachte Darstellung des modifizierten Entwicklungsprozesses für „GenGMF“ (im Vergleich zum GMF (s. a. Abbildung 3.1))

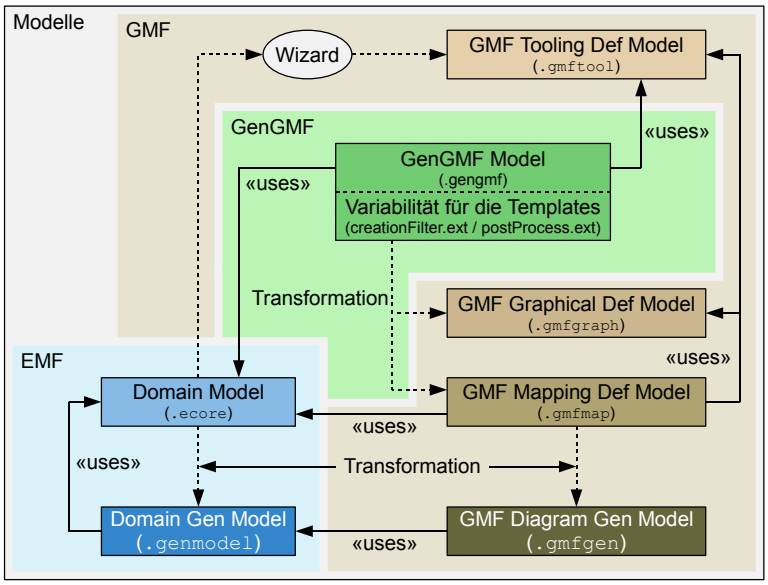


Abbildung 6.2: Der modifizierte Entwicklungsprozess für „GenGMF“ (im Vergleich zum GMF (s. a. Abbildung 3.2))

6.2 Anpassungen für den „GenGMF“-Editor

Das „GenGMF“-Metamodell baut auf dem EMF auf. Für die Erstellung von „GenGMF“-Modellen wird ein vom EMF aus dem Metamodell generierter und dann manuell modifizierter Editor verwendet. „Manuell modifiziert“ heißt hier, dass für die wichtigsten Metamodellelemente ein Satz an Icons erstellt wurde, um die Funktionalitäten auch als Piktogramm darzustellen.

Die Namen der Elemente des Metamodells sind aus jeweils zwei Bestandteilen (links / rechts) aufgebaut. Für jeden Namensbestandteil wurde ein Piktogrammbestandteil erstellt, wobei die Piktogrammbestandteile auf der gleichen Seite wie die zugeordneten Namensbestandteile dargestellt werden. Eine Zuordnung der Piktogrammbestandteile zu den Namensbestandteilen ist der Tabelle 6.1a zu entnehmen. Die beiden Namensbestandteile werden in verschiedenen Kombinationen zusammengesetzt. Mögliche Kombinationen der Namensbestandteile mit den ebenfalls kombinierten Piktogrammen sind in der Tabelle 6.1b dargestellt.

Tabelle 6.1: Aufbau der Piktogramme sowie der Namen der Elemente des „GenGMF“-Editors

(a) Zuordnung der Piktogramm- zu den Namensbestandteilen

Piktogramm- & Namensbestandteile	
linker Bestandteil	
/	Edge
↗	ReferenceEdge
▫	Node
▣	Compartment
·	CompartmentChild
a	Label
rechter Bestandteil	
!	Desc
?	Template
⋮	Graph
↕	Map

(b) Mögliche Kombinationen der einzelnen Bestandteile aus der Tabelle (a)

Piktogramm & Name des Elementes	
Kanten (/ , ↗)	
/!	EdgeDesc
↗!	ReferenceEdgeDesc
/?	EdgeTemplate
/⋮	EdgeGraph
/↕	EdgeMap
Knoten (▫ , ▣)	
▫!	NodeDesc
▫?	NodeTemplate
▫⋮	NodeGraph
▫↕	NodeMap
▣!	CompartmentDesc
▣?	CompartmentTemplate
▣⋮	CompartmentGraph
Mögliche Kindelemente der Deskriptoren	
·!	CompartmentChildDesc
a!	LabelDesc

Die Icons für den „GenGMF“-Editor sind nach demselben Prinzip integriert worden, wie die Icons für die „base“-Editoren (vgl.: Abbildung 4.2). Die hierfür notwendigen Pointcuts `getImage()` und `getText()` des Aspektes `EditorDecoration` sind im Listing 6.1 dargestellt. Die angezeigten Texte im Baum des Editors werden durch den `getText()`-Pointcut mit in oAW-Xtend geschriebenen Funktionen berechnet. Exemplarisch sind die, für die Berechnung des anzuzeigenden Textes von `LabelDesc`-Elementen (a!), relevanten Funktionen im Listing 6.2 aufgeführt.

Listing 6.1: Der Aspekt „EditorDecoration“ für die Funktionen `getText()` und `getImage()`

```
5  /**
6   * customizes the ItemProvider getImage() and getText() functions to support
7   * self made images and dynamic label strings (using oAW) without the risk of
8   * overgenerating the customized implemetation
9   */
10 public aspect EditorDecoration {
14   /**
15    * customize item icon
16    * points to all ItemProvider classes which have a getImage(Object) function
17    */
18   pointcut getImage(Object item) :
19     execution(public Object *..*ItemProvider.getImage(Object)) && args(item);
20
21   /**
22    * the around advice for customizing the icon
23    */
24   Object around(Object item) : getImage(item) {
25     Object imageURL = IconFinder.getImageURL(item);
26     if (imageURL != null)
27       return imageURL;
28     else
29       return proceed(item);
30   }

32   /**
33    * customize item text
34    * points to all ItemProvider classes which have a getText(Object) function
35    */
36   pointcut getText(Object item) :
37     execution(public String *..*ItemProvider.getText(Object)) && args(item);
38
39   /**
40    * the around advice for customizing the displayed text
41    */
42   String around(Object item): getText(item) {
43     String customItemText = oawFacade.getCustomItemText(item);
44     if (customItemText != null)
45       return customItemText;
46     else
47       return proceed(item);
48   }
49 }
```

Ebenfalls durch Aspekte manuell verändert wurde, wie auch in „base“, die Auswahlliste der möglichen Einträge für Referenzen im Eigenschaftsfenster. Hierzu werden alle Aufrufe der Methode `protected org.eclipse.emf.edit.provider.ItemProviderAdapter.createItemPropertyDescriptor(/*elf Parameter*/)` mit einem Pointcut und einem `around` Advice abgefangen. Anstatt des `ItemPropertyDescriptors` aus dem EMF wird die eigene Implementierung `ItemPropertyDescriptorReferenceFilter` verwendet, um die angezeigten Elemente mit einer oAW-Xtend-Funktion zu filtern. Sobald ein Dropdown-Feld mit Referenzen im Eigenschaftsfenster geöffnet wird, werden alle von der ursprünglichen Implementierung gelieferten Referenzen gefiltert. Exemplarisch sind im Listing 6.3 die beiden Filterfunktionen für `LabelDesc`-Elemente (a!) angegeben.

Der jeweils erste Parameter gibt das Element an, welches die Referenz (dritter Parameter) enthält. Ob ein im zweiten Parameter angegebenes Element in der Liste auftauchen soll, kann über den Rückgabewert gesteuert werden.

Listing 6.2: Funktionen in „ItemTextProvider.ext“ zur Berechnung des `LabelDesc`-Textes (a!)

```
58 // description text for the label descriptor
59 String getItemText(LabelDesc d)
60 : d.metaName("gengmf::desc::")
61 + " "
62 + d.labelFeature.getShortItemText()
63 + " -> "
64 + d.labelMapping.getItemText()
65 ;

89 // label text for mappings::LabelMapping
90 String getItemText(LabelMapping m)
91 : m.metaName("mappings::")
92 + " "
93 + m.diagramLabel.name
94 ;

112 // short description for ecore::EAttribute and ecore::EReference
113 private String getShortItemText(EStructuralFeature f)
114 : f.eClass().name
115 + " "
116 + f.eContainingClass.name
117 + "."
118 + f.name
119 ;

121 // removes the package part from the metaType name of an object
122 private cached String metaName(Object o, String stripName)
123 : o.metaType.name.replaceFirst(stripName, "")
124 ;
```

Listing 6.3: Filterfunktionen für den Typen `LabelDesc` (a!) in ReferenceFilter.ext

```
90 /**
91  * the displayed EAttribute should be from the EClass which is referenced by
92  * the parent FigureDesc
93  */
94 Boolean isValidReference(LabelDesc desc, EAttribute attr, String feature)
95 : desc.figureDesc.getClass().eAllAttributes.contains(attr)
96 ;
97 /**
98  * the FeatureLabelMapping must be contained in the mapping description of the
99  * referenced template
100  */
101 Boolean isValidReference(LabelDesc desc, FeatureLabelMapping m, String feature)
102 : desc.figureDesc.getTemplate().getMap().eAllContents.contains(m)
103 ;
```

6.3 Templatestrukturen

„GenGMF“ nutzt Templatestrukturen, um Gemeinsamkeiten in den GMF-Definitionen zu vereinheitlichen. Diese Templates setzen sich aus den bereits im GMF verwendeten Objekten, wie z. B. `NodeMapping` (), `FigureDescriptor` () und `ChildAccess` () zusammen (siehe Kapitel 3). Hierzu wurden im „GenGMF“-Metamodell `Containment`-Referenzen auf Elemente der GMF-Graph- und GMF-Map-Metamodelle erstellt. Zwei Templates – ein Knoten (`NodeTemplate` (?)) mit einem Label (`FeatureLabelMapping` ()) sowie eine Verbindung (`EdgeTemplate` (/?)) – werden schematisch in der Abbildung 6.3 dargestellt. Im Vergleich zur Abbildung 3.6 (S. 16) bleibt

das „Graphical Def Model“ fast vollständig erhalten. Nicht übernommen wurden die Elemente `Canvas` sowie `FigureGallery`. Sie werden während der Transformation dynamisch erzeugt. Bei dem „Mapping Def Model“ (vgl.: Abbildung 3.7, S. 18) werden – für das GMF untypisch – aus dem `NodeMapping`-Element keine Referenzen auf Elemente aus dem „Tooling Def Model“ gesetzt. Das Element `TopNodeReference` wird automatisch durch die Transformation erstellt und ist in den Templates nicht enthalten. Damit entfallen auch die in der Abbildung 3.7 vorhandenen Referenzen auf die Elemente des „Domain Model“.

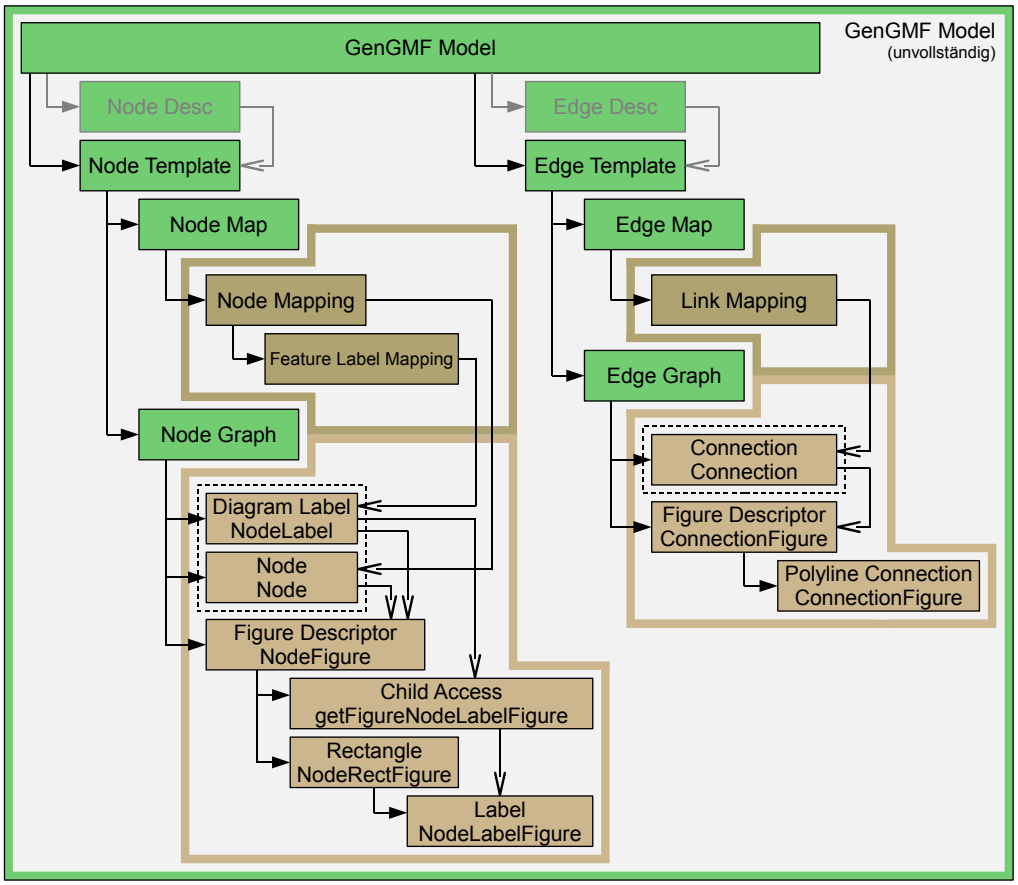


Abbildung 6.3: Schematische Darstellung der in „GenGMF“ verwendeten GMF-Template-Strukturen

Neben den in der Abbildung 6.3 gezeigten Elementen `Node-` und `EdgeTemplate` existiert noch ein weiteres, mit ersterem verwandtes Element `CompartmentTemplate`. Es wird benötigt, um die Compartments und Ports zu modellieren. Zusätzlich zu den im `NodeGraph` möglichen Einstellungen kann ein `Compartment`-Element (s. a. Kapitel 3.5.2) im `CompartmentGraph` angelegt werden. (s. a. [Sch08b])

Während der Verarbeitung der Templates wird in jeder Zeichenkette, die in den Templatestrukturen enthalten ist, die Teilzeichenkette „`__CLASSNAME__`“ durch den Namen des in den Deskriptoren verknüpften Knotentyps ersetzt. Hierdurch ist es z. B. möglich, den Namen eines Knotentypen als Label im Knoten darzustellen, ohne für jeden Knotentypen eine eigene graphische Repräsentation zu erstellen.

6.4 Deskriptoren

Deskriptoren bilden einen Teil der Variabilität für die Templates ab. Sie verknüpfen die Templates mit den für das Mapping benötigten Referenzen auf die Modellelemente des „Domain Model“ bzw. des „Tooling Def Model“.

In der Abbildung 6.4 werden die Typen `NodeDesc` und `ReferenceEdgeDesc` gezeigt, welche zum Abbilden von kinderlosen Knoten bzw. Verbindungen dienen. Daneben existieren die Elementtypen `CompartmentDesc`, zum Definieren von Knoten mit Kindern als Compartments oder als Ports, sowie der Typ `EdgeDesc`. Letzter dient dazu, Verbindungen mit eigenem Elementtyp zu definieren – im Gegensatz zum `ReferenceEdgeDesc`, welcher direkte Referenzen zwischen Elementtypen abbildet.

6.4.1 Labels

Wenn der Inhalt eines Attributes in einem Knoten oder als Beschriftung einer Verbindung angezeigt werden soll, muss ein `LabelDesc` Element innerhalb des entsprechenden Elterndescriptors angelegt werden. Das `LabelDesc`-Element enthält jeweils eine Referenz auf das entsprechende `FeatureLabelMapping`-Element des Templates sowie auf das anzuzeigende Attribut des Elementtypen des Elterndescriptors.

6.4.2 Compartments

Um die im Kapitel 3.5.2 besprochenen und in der Abbildung 3.8a gezeigten Compartments mit „GenGMF“ zu definieren, müssen innerhalb des Elementes `CompartmentDesc` Kinder vom Typ `CompartmentChildDesc` angelegt werden. Hier kann die verwendete `containment`-Referenz mit der Kind-Knotenbeschreibung (`NodeDesc` oder `CompartmentDesc`) und dem `Compartment`-Element aus dem `CompartmentGraph` des verknüpften `CompartmentTemplates` miteinander kombiniert werden. Eine schematische Darstellung ist in der Abbildung 6.5a zu finden.

6.4.3 Ports

Die im Kapitel 3.5.3 beschriebenen Ports werden auch mit „GenGMF“ fast genauso definiert wie Compartments. Der einzige Unterschied besteht darin, die Referenz auf das `Compartment` leer zu lassen. Dies ist in der Abbildung 6.5b verdeutlicht.

6.4.4 Phantom Nodes

Bei GMF-Modellen müssen „Phantom Nodes“ immer dann eingesetzt werden, wenn rekursive Compartment- oder Portstrukturen abgebildet werden sollen (siehe Kapitel 3.5.3 und Abbildung 3.8c). Im Gegensatz dazu werden bei den „GenGMF“-Modellen alle Deskriptoren parallel abgelegt. Auf der Grundlage der evtl. rekursiven Strukturdefinitionen der Deskriptoren wird algorithmisch entschieden, ob ein `NodeMapping` während der Transformation als Kind eines `TopNodeReference`- oder `ChildReference`-Elementes angelegt wird.

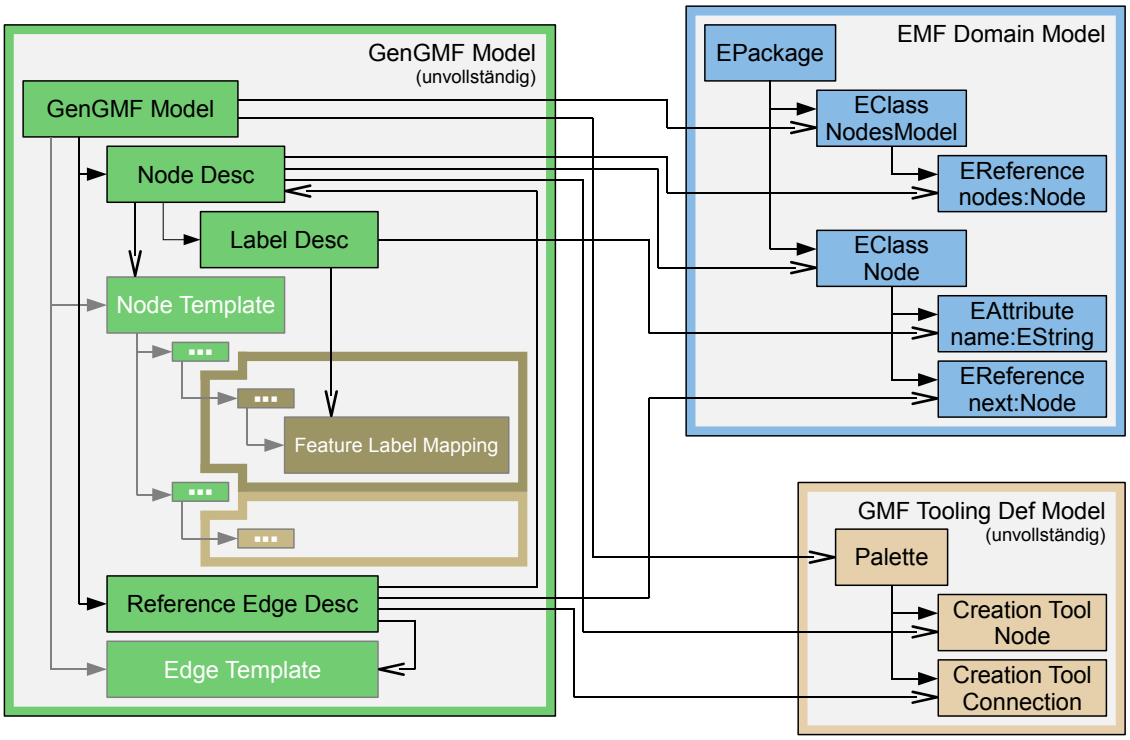
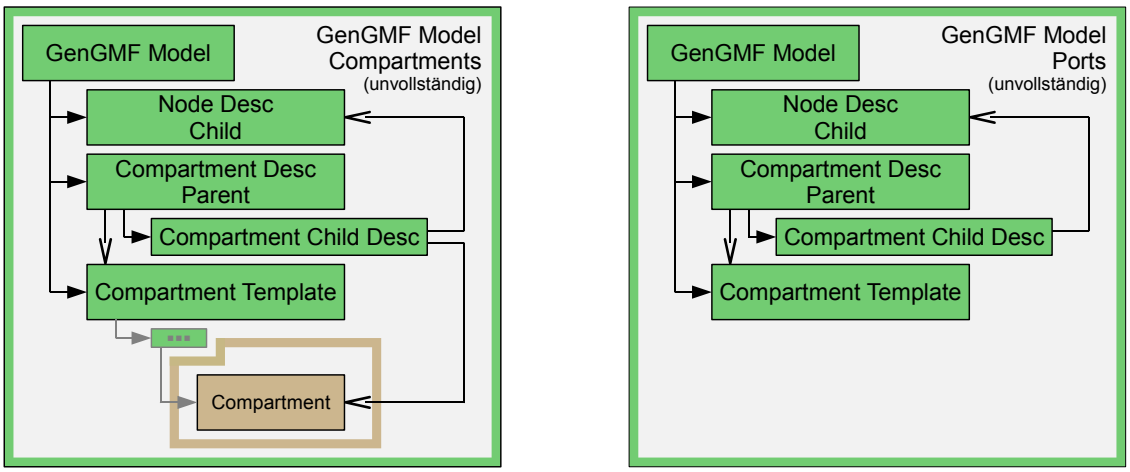


Abbildung 6.4: Schematische Darstellung der in „GenGMF“ verwendeten Deskriptoren mit den Referenzen auf die Modelle „Domain Model“ und „Tooling Def Model“ (vgl.: Abbildung 3.7)



(a) Compartments (s. a. Abbildung 3.8a)

(b) Ports (s. a. Abbildung 3.8b)

Abbildung 6.5: Schematische Darstellung der für „GenGMF“ benötigten Modellstrukturen für die Modellierung von Kindelementen

6.4.5 Diagram Partitioning

Das im Kapitel 3.5.5 beschriebene „Diagram Partitioning“ wird von „GenGMF“ nicht unterstützt. Dies liegt vor allem daran, dass einige Einstellungen in den GMF-Modellen nur im „Diagram Gen Model“ vorgenommen werden können, was direkt von GMF bearbeitet wird und von „GenGMF“ nicht generiert werden kann.

6.4.6 Verbindungen

Analog zu den für die Knoten benötigten `NodeTemplate`- (▫?) und `NodeDesc`-Elementen (▫!) müssen auch für Verbindungen Template- und Deskriptorstrukturen aufgebaut werden. Im GMF wird zwischen den Element- und Referenz-basierten Verbindungen nur auf der Basis der belegten Attribute im `LinkMapping` (⁖), also der Semantik des Modellelementes, unterschieden. Für „GenGMF“ wurden dagegen die syntaktisch eigenständigen Elemente `EdgeDesc` (/!) sowie `ReferenceEdgeDesc` (↗!) eingeführt. Ähnlich dem GMF (siehe Kapitel 3.5.1) müssen für die Element-basierte Verbindung (`EdgeDesc` (/!)) die Referenzen auf den Ursprung und das Ziel der Verbindung angegeben werden. `ReferenceEdgeDesc`-Elemente (↗!) enthalten im Gegensatz dazu nur die Angabe der Referenz, welche die Verbindung an sich ausmacht.

6.5 Transformation

Dieser Abschnitt beschreibt den Ablauf der Transformation des „GenGMF“-Modells in die beiden GMF-Modelle „Graphical Def Model“ sowie „Mapping Def Model“. Angestoßen wir die Transformation durch den Eintrag „Generate GMF-Graph and -Map model“ im Kontextmenü der `.gengmf`-Datei des Eclipse „Package Explorers“ Dieser Vorgang lässt sich grob in die drei Schritte „Validieren des ‚GenGMF‘-Modells “, „Generieren des GMF-Graph-Modells“ sowie „Generieren des GMF-Map-Modells“ unterteilen.

Die Verarbeitung erfolgt durch einen oAW-Workflow (siehe Kapitel 2.2.4) und wird im Vordergrund gestartet. Durch die Verwendung eines `ProgressMonitorDialog`es wird es dem Anwender ermöglicht, den Fortschritt zu verfolgen bzw. die Verarbeitung im Hintergrund ablaufen zu lassen.

Die Erläuterung der einzelnen oben genannten Transformationsschritte erfolgt erst nachdem die für die Generierung sehr wichtige Klasse „`EMFCloner`“ vorgestellt wurde.

6.5.1 Die Klasse „`EMFCloner`“

Im Rahmen von „GenGMF“ müssen die in den Templates definierten Objektbäume im Kontext des Deskriptors kopiert werden. Die Klasse `EcoreUtil.Copier` aus dem EMF stellt hierfür die notwendige Basisfunktionalität bereit. Hier können durch das mehrfache Aufrufen der Methode `public EObject copy(EObject eObject)` verschiedene Objektbäume sowie – mit der Methode `public void copyReferences()` – die Relationen zwischen ihnen kopiert werden (s. a. [Ecl06]). Die für die Verwendung von Templates sehr wichtige Semantik des ursprünglichen Modells wird so mit kopiert.

Für die Generierung der GMF-Modelle wurde mit `EMFCloner` die Klasse `EcoreUtil.Copier` um einen „kontextsensitiven Cache“ erweitert und bildet damit einen essentiellen Teil des in „GenGMF“ verwendeten Algorithmus ab.

Während der Transformation wird für jeden Kontext (Deskriptor) ein `Copier` instanziert und über eine Map dem Kontext zugeordnet. Die für das kontextsensitive Kopieren von Objekten relaventen Funktionen sind im Listing 6.4 enthalten.

Listing 6.4: Utility-Klasse „EMFCloner“

```
18 /**
19  * Allowes to copy EObjects on a context sensitive basis. The copies are cached
20  * per context between the calls.
21  */
22 public class EmfCloner {
54   /** the global context cache */
55   private static Map<EObject, Copier> globalCache =
56     new HashMap<EObject, Copier>();

58   /** the current context */
59   private EObject context;
60   /** the cache for the current context */
61   private Copier copier;

68   /**
69    * initialization for the context sensitive cache
71    * @param context          the context for being sensitive
73    */
74   public EmfCloner(EObject context) {
75     this.context = context;
84     if (context != null) {
85       copier = globalCache.get(context);
86       if (null == copier) {
87         copier = new Copier();
88         globalCache.put(context, copier);
89       }
90     } else {
91       copier = new Copier();
92     }
93   }

95   /**
96    * statically clones an EObject obj in the context of the EObject context
99    * @param obj              the EObject to clone
100   * @param context          the context for being sensitive
102   * @return                 the cloned EObject
103   * @throws                 CloneNotSupportedException
104   */
105  public static EObject clone(EObject obj, EObject context)
106      throws CloneNotSupportedException {
107    EmfCloner emfCloner = new EmfCloner(context);
108    return emfCloner.clone(obj);
109  }

118  /**
119   * performs the cloning
121   * @param obj  the EObject to clone
122   * @return     the copied object
123   */
124  private EObject clone(EObject obj) {
125    /* the copier object is already context specific */
126    if (copier.containsKey(obj)) {
127      return copier.get(obj);
128    }
129    EObject copy = copier.copy(obj);
130    copier.copyReferences();
131    return copy;
132  }
173 }
```

Während der Transformation von Deskriptoren kann es notwendig sein, auf Objektbäume anderer Deskriptoren zuzugreifen. Hierzu wird eine Anfrage an den Objektcache des `Copier` des zweiten Deskriptors gestellt. Entweder wird die im Cache gespeicherte Kopie oder, wenn bisher keine Kopie erstellt worden ist, eine dynamisch und „on demand“ erstellte Kopie zurückgeliefert.

In den kopierten Bäumen wird anschließend jedes Vorkommen der Zeichenkette „`__CLASSNAME__`“ durch den Namen der im Deskriptor verknüpften Domänenklasse ersetzt. Die rekursive Ersetzung im Baum erfolgt mit der Methode `public static EObject replaceStrings(EObject, String, String)` der Klasse `EMFCloner` aus dem Listing 6.5.

Beispiel: Objektbaumkopien
Wenn in einem Template ein `Node`-Element (✦) mit dem Namen „`__CLASSNAME__Node`“ definiert und in einem das Template referenzierenden Deskriptors die Klasse „`MyClass`“ verknüpft ist, dann entsteht durch den Kopiervorgang und die Nachverarbeitung der Zeichenketten ein fast identisches Objekt mit dem Namen „`MyClassNode`“.
Ein `NodeMapping`-Element (⌗) im selben Template mit einer Referenz auf das ursprüngliche `Node`-Element (✦) „`__CLASSNAME__Node`“ würde genauso kopiert werden. Die Kopie enthält dann jedoch eine Referenz auf das kopierte Element („`MyClassNode`“).
Für ein in einem anderen `CompartmentDesc` (▫!) definierten `CompartmentChildDesc` (·!), welcher auf den „`MyClass`“-Deskriptor verweist, muss nun auf die kopierten Objektbäume des zweiten Deskriptors zugegriffen werden, um entsprechende Referenzen in einem `ChildReference`-Element (▷) (vgl.: Kapitel 3.5.2) setzen zu können.

Listing 6.5: Funktion `replaceStrings` in der Utility-Klasse „EMFCloner“

```
134  /**
135   * recursive post processing for all string attributes in an EObject
141   */
142  public static EObject replaceStrings(EObject obj, String search,
143      String replace) {
144    EClass c = obj.eClass();
145    EList<EAttribute> allAttributes = c.getEAllAttributes();
146    for (EAttribute attribute : allAttributes) {
147      if (obj.eIsSet(attribute)) {
148        Object attr = obj.eGet(attribute);
149        if (attr instanceof String) {
150          obj.eSet(attribute, ((String) attr)
151              .replaceAll(search, replace));
152        }
153      }
154    }
155
156    EList<EReference> allContainments = c.getEAllContainments();
157    for (EReference containment : allContainments) {
158      if (obj.eIsSet(containment)) {
159        Object con = obj.eGet(containment);
160        if (containment.isMany()) {
162          EList conList = (EList) con;
163          for (Object object : conList) {
164            replaceStrings((EObject) object, search, replace);
165          }
166        } else {
167          replaceStrings((EObject) con, search, replace);
168        }
169      }
170    }
171    return obj;
172  }
```

6.5.2 Validieren des „GenGMF“-Modells

„GenGMF“ generiert GMF-Modelle, welche durch das GMF bei der Transformation in das „Diagram Gen Model“ validiert werden. Da es dem Anwender von „GenGMF“ nicht zugemutet werden kann, Fehler im „GenGMF“ aufgrund von Fehlermeldungen aus dem GMF-Validator zu finden, müssen die „GenGMF“-Modelle vor dem Erstellen der GMF-Modelle geprüft werden. Die in oAW-Check (siehe Kapitel 2.2.4) implementierte Validierung in „GenGMF“ prüft zur Zeit nur wahrscheinliche Fehler im Modell ab, wie z. B. die Zuweisungskompatibilität des Domänenelementes zur `containment`-Referenz in den Deskriptoren.

6.5.3 Generieren des „Graphical Def Model“

Für alle in einem „GenGMF“-Modell enthaltenen `Node`- (▫!), `Compartment`- (▫!), `Edge`- (/!) und `ReferenceEdgeDesc`-Elemente (?!) werden die in den Graphikdefinitionen ( ) enthaltenen Templatestrukturen mit Hilfe der Klasse „EMFCloner“ kopiert. Hierzu sind die Templates so organisiert, dass die dort enthaltenen `containment`-Strukturen mit denen im Zielmodell korrespondieren.

Durch die Funktion `create Canvas generate(Model m)` des Listings 6.6 werden die gewonnenen Objekte, der Syntax des „Graphical Def Model“ entsprechend, nach der `containment`-Referenz

Listing 6.6: „GenGMF“-Transformationsregeln zum Erstellen des „Graphical Def Model“

```
15 /** Generating the gmfgraph model */
16 create Canvas generate(Model m)
17 :   let fg = new FigureGallery
18 :   let cn = m.getAllNodes()
19 :   let cne = m.getAllFigures()
20 :   newTask("generating GMFGraph model",
21             3 * cne.size + m.compartments.size, 1000)
22 ->  setName(m.name)
23 ->  setFigures({fg})
24 ->  fg.setName('Default')
25 ->  cne.collect(
26         e|  {e.createDescriptorFromTemplate()}
27             .collect(
28                 e2| fg.descriptors.add(e2)
30     )       )
31 ->  cn.collect(
32         e|  e.createNodesFromTemplate()
33             .collect(
34                 e2| this.nodes.add(e2)
36     )       )
37 ->  m.compartments.collect(
38         e|  e.createCompartmentsFromTemplate()
39             .collect(
40                 e2| this.compartments.add(e2)
42     )       )
43 ->  m.edges.collect(
44         e|  {e.createEdgeFromTemplate()}
45             .collect(
46                 e2| this.connections.add(e2)
48     )       )
49 ->  cne.collect(
50         e|  e.createLabelsFromTemplate()
51             .collect(
52                 e2| this.labels.add(e2)
54     )       )
74 ;
```

gruppiert, der sie hinzugefügt werden müssen. Exemplarisch zeigt die im letzten Listing aufgerufene Funktion `List[DiagramLabel] createLabelsFromTemplate(FigureDesc d, Model m)` (dargestellt im Listing 6.7) die Transformationsregel für `LabelDesc`-Elemente (a!). Das fertige Modell wird anschließend durch den Workflow gespeichert.

Die aufgerufenen Hilfsfunktionen `cached Object cloneAndFilter(Object o, FigureDesc ctx)` und andere sind im Listing 6.8 dargestellt.

Listing 6.7: „GenGMF“-Transformationsregel für Elemente vom Typ `LabelDesc` (a!) für das „Graphical Def Model“

```
111 /** cloning the labels from the figure template */
112 List[DiagramLabel] createLabelsFromTemplate(FigureDesc d)
113 :   stepStart("Label for " + d.getMonitorString())
114 ->  stepStop()
115 ->  d
116     .getTemplate()
117     .getGraph()
118     .labels
119     .collect(e|e.cloneAndFilter(d))
120 ;
```

Listing 6.8: Die Hilfsfunktion `cloneAndFilter` und Weitere

```
1  /** Java functions */
2  Object clone(Object o, Object _context, String creationFilter)
3  :JAVA EmfCloner.clone(o.e.emf.ecore.EObject,o.e.emf.ecore.EObject,java.lang.String);
4  Object filterNames(Object o, String search, String replace)
5  :JAVA EmfCloner.replaceStrings(o.e.emf.ecore.EObject,java.lang.String,java.lang.String);
6
7  /** helper functions */
8  cached Object clone(Object o, Object _context, Model m)
9  : clone(o, _context, m.creationFilter);
10 cached Object clone(Object o, FigureDesc _context)
11 : clone(o, _context, _context.getModel());
12
13 /** cloneAndFilter */
14 cached Object cloneAndFilter(Object o, FigureDesc ctx)
15 : clone(o, ctx)
16   .filterNames('__CLASSNAME__',
17                null == ctx.getClass() ? "Default" : ctx.getClass().name);
```

6.5.4 Generieren des „Mapping Def Model“

Die relativ einfachen Transformationsregeln für das „Graphical Def Model“ bilden eine flache Objektstruktur ab. Im Vergleich dazu sind die Transformationsregeln für das „Mapping Def Model“ aufgrund der möglichen verschachtelten Struktur der `NodeMapping`-Elemente (⌑) bei Verwendung von Compartments wesentlich komplexer. Ähnlich wie für das „Graphical Def Model“ existiert auch hier eine `generate`-Funktion (siehe Listing 6.9), um die einzelnen Teilschritte der Transformation zusammenzufassen.

Das Listing 6.10 zeigt die von der `generate`-Funktion zur Verarbeitung von `NodeDesc`-Elementen (▫!) verschachtelt aufgerufenen Funktionen `createNodeFromTemplate` sowie `createAbstractNodeFromTemplate`. In Letzterer wird das aus dem Map-Template (⇕) kopierte `NodeMapping`-Element (⌑) initialisiert. Dies geschieht u. a. mit der Funktion `setupMappingEntry` aus dem Listing 6.11. Die Initialisierung der `FeatureLabelMappings` (ab) eines `NodeMappings` (⌑) wird an die Funktion `setupMappingEntryLM` delegiert.

Listing 6.9: Die Hauptfunktion `generate` zum Generieren des „Mapping Def Model“

```
/** Generating the gmfmap model */
create Mapping generate(Model m, Canvas c)
:   let cn =
        m.getAllNodes()
        .select(
            e|  e.isTopNodeReference())
:   newTask("generating GMFMap model", cn.size + m.edges.size, 1000)
->  setDiagram(createCanvasMapping(m,c))
->  cn.collect(
        e|  {e.createNodeFromTemplate()}
            .collect(
                e2| this.nodes.add(e2)
    )       )
->  m.edges.collect(
        e|  {e.createLinkFromTemplate()}
            .collect(
                e2| this.links.add(e2)
    )       )
;
```

Listing 6.10: Die Funktionen `createNodeFromTemplate` und `createAbstractNodeFromTemplate`

```
/**
 * creates a top node reference for node descriptors,
 * delegates to {@link #createAbstractNodeFromTemplate(AbstractNodeDesc)}
 */
TopNodeReference createNodeFromTemplate(NodeDesc d)
:   stepStart("TopNodeReference for " + d.getMonitorString())
->  stepStop()
->  d.createAbstractNodeFromTemplate()
;

/** creates a top node reference for node and compartment descriptors */
create TopNodeReference createAbstractNodeFromTemplate(AbstractNodeDesc d)
:   setOwnedChild((NodeMapping)
        ((AbstractNodeTemplate)
            d.getTemplate()
        )
        .map
        .node
        .cloneAndFilter(d))
->  setContainmentFeature(d.containmentFeature)
->  ownedChild.setupMappingEntry(d)
->  ownedChild.setupTool(d)
;
```

Listing 6.11: Die Funktionen `setupMappingEntry` und `setupMappingEntryLM`

```
/** setup for mapping entry */
MappingEntry setupMappingEntry(MappingEntry me, ClassFigureDesc d)
:   me.setDomainMetaElement(d.getClass())
->  me.setupMappingEntryLM(d)
;

/** setup for mapping entry's label mapping */
MappingEntry setupMappingEntryLM(MappingEntry me, FigureDesc d)
:   d.labelMappings.isEmpty
    ?   null
    :   d.labelMappings
        .collect(
            e|  ((FeatureLabelMapping)
                    e.labelMapping.clone(d)
                ).features
                .add(e.labelFeature)
        )
/* removing feature label mappings without a feature set */
->  me.setLabelMappings(
        me.labelMappings.reject(
            e|  FeatureLabelMapping.isInstance(e)
            &&  0 == ((FeatureLabelMapping)e).features.size))
->  me
;
```

6.6 Skripte zum Abbilden von Variabilität

Die bisher in „GenGMF“ beschriebenen Funktionalitäten decken die für die Generierung benötigte Flexibilität nur unzureichend ab. So kann zwar z. B. die Hintergrundfarbe in den Templates angegeben werden, jedoch ist sie dann für alle Knoten gleich. Im Rahmen von „GenGMF“ kann mit Skripten an zwei verschiedenen Stellen mit Hilfe von „Filtern“ in die Transformation eingegriffen werden. Die beiden Filterarten werden entweder bei der Erstellung von Objekten (`creationFilter`) oder bei der Nachverarbeitung (`postFilter`) eingesetzt.

Um die Möglichkeiten der dynamischen Modellveränderung nutzen zu können, muss eine oAW-Xtend-Datei in einem Source-Ordner im selben Eclipse-Projekt wie das „GenGMF“-Modell angelegt werden. Der Ressourcen-Pfad der Datei ohne die Endung „`.ext`“ sowie mit zwei Doppelpunkten als Pfadtrenner wird im Attribut „Creation Filter“ oder „Post Proc Filter“ des Wurzelelementes `Model` () hinterlegt (z. B. „`gengmf::filter::createProc`“ für eine Datei mit dem Pfad „[Source-Ordner]`/gengmf/filter/createProc.ext`“).

Die in den Skripten hinterlegten Filterfunktionen müssen einem definierten Schema entsprechen, das im folgenden Abschnitt erklärt wird. Anschließend wird die Verwendung der `creationFilter` sowie `postFilter` erläutert.

6.6.1 Ermitteln der Funktionsnamen

Die Namen der in den Xtend-Dateien hinterlegten Funktionen müssen ein festgelegtes Schema einhalten, um während der „GenGMF“-Transformation aufgerufen werden zu können. Abhängig von dem zu verarbeitenden Element des „GenGMF“-Modells werden die Namen nach dem in der Tabelle 6.2 dargestellten Schema erstellt. Für jeden generierten Namen wird versucht die Funktion aufzurufen. Sobald ein Aufruf erfolgreich abgeschlossen wurde, ist die Verarbeitung abgeschlossen.

Tabelle 6.2: Aufbau der möglichen Funktionsnamen für die „GenGMF“-Elementtypen

`NodeDesc` (▫!), `CompartmentDesc` (▣!) und `EdgeDesc` (/!)	
„`filter`“ + [Domänenelementname]	+ [GMF-Elementname]
„`filter`“ + [Domänenelementname]	
„`filter`“	+ [GMF-Elementname]
„`filter`“	
`ReferenceEdgeDesc` (↗!)	
„`filter`“ + [Domänenelementname] + [Referenzname]	+ [GMF-Elementname]
„`filter`“ + [Domänenelementname] + [Referenzname]	
„`filter`“ + [Domänenelementname]	+ [GMF-Elementname]
„`filter`“ + [Domänenelementname]	
„`filter`“	+ [GMF-Elementname]
„`filter`“	

Durch die Möglichkeit, einen allgemeingültigeren Funktionsnamen zu wählen, wird die Implementierung von Funktionen vereinfacht, die verschiedene Fälle abdecken. So kann z. B. eine Hintergrundfarbe für alle Metamodellelemente eines Paketes gesetzt werden, wenn das Paket innerhalb der Funktion entsprechend geprüft wird. Die Funktion müsste sonst für jedes Element implementiert werden, nur weil der Metamodellelementname im Funktionsnamen enthalten ist. Die Funktion wird immer mit zwei Parametern aufgerufen. Als erster Parameter wird das kopierte Element aus dem Template übergeben. Der zweite Parameter ist der Deskriptor (`!`).

6.6.2 `creationFilter`

`creationFilter` werden eingesetzt, wenn die in einem Template (`?`) benutzten Elementtypen nicht 100-prozentig passen, denn oft kann die gewünschte veränderte Darstellung nur mit dem Einsatz von artverwandten Typen erreicht werden.

Sollen z. B. zwei Verbindungen unterschiedlich dargestellt werden, so bietet sich eine Differenzierung durch eine offene (⟶) bzw. geschlossene (⟶) Pfeilspitze an. Die Art der Darstellung wird im „Graphical Def Model“ durch die Verwendung einer `PolylineDecoration` ( ) bzw. durch eine `PolygonDecoration` ( ) und das Attribut `fill = true` angegeben.

Bisher wurde von der Klasse `EMFCloner` die `EcoreUtil.Copier` Klasse verwendet, um die in den Templates (`?`) enthaltenen Strukturen für die Deskriptoren (`!`) zu kopieren. Die Klasse `EcoreUtil.Copier` stellt verschiedene als `protected` markierte Funktionen zur Verfügung, mit denen – durch Überladen – die Funktionalität des `Copier`s modifiziert werden kann. In der Klasse `EMFCloner` wurde hierzu eine innere Klasse `Copier` eingeführt (siehe Listing 6.13) und die Methode `protected EObject createCopy(EObject object)` überschrieben. Laut Spezifikation erstellt `createCopy` ein neues Element vom gleichen Typ wie des übergebenen Elementes. Wenn ein Element eines anderen zur `containment`-Referenz zuweisungskompatiblen Typs übergeben wird, werden anschließend in den darauf folgenden Schritten im `EcoreUtil.Copier` nur die in den gemeinsamen Elternklassen definierten Attribute und Referenzen kopiert.

Im Listing 6.12 ist eine Filterfunktion zu sehen, in der eine Pfeilspitze verändert wird. Mit dieser Funktionalität ist es z. B. auch möglich, in einem `NodeGraph`-Template ( ) statt eines normalen Rechtecks (`Rectangle` ( )) eines mit abgerundeten Ecken (`RoundedRectangle` ( )) zu verwenden.

Listing 6.12: Filterfunktionen, um die Pfeilspitze einer Verbindung zu ändern

```
1 /**
2  * Using a PolylineDecoration instead of a PolygonDecoration for the
3  * next reference of a Node
4  */
5 Figure filterNodeNextPolygonDecoration(PolygonDecoration pd, EdgeDesc ed)
6 : new PolylineDecoration
7 ;
```

Listing 6.13: Die innere Klasse `Copier` sowie deren Initialisierung in `EMFCloner`

```
23   /**
24    * Allowes to create a copy of an EObject. Instead of coping the type
25    * information one-to-one -- all newly creates EObjects are passed through
26    * the {@link #creationFilter} Xtend functions provided in the parent class.
27    */
29   private class Copier extends EcoreUtil.Copier {
31     protected EObject createCopy(EObject object) {
32       EObject copy = super.createCopy(object);
33       if (null != creationFilter) {
34         Object[] params = new Object[] { copy, context };
36         for (String filterName : GenGMFHelper.generateFunctionNames(
37             "filter", object, context)) {
38           try {
39             copy = (EObject) creationFilter
40                 .call(filterName, params);
41             System.out.println("CreationFilter succeeded for "
42                 + copy.toString() + " with filter "
43                 + filterName + "!");
44             break;
45           } catch (IllegalArgumentException e) {
46             // function not found
47           }
48         }
49       }
50       return copy;
51     }
52   }

63   /** the XtendFacade is also cached to gain some performance */
64   private static XtendFacade creationFilter = null;
65   /** the XtendFacade will be changed if the pointer to the function changes */
66   private static String creationFilterString = null;

68   /**
69    * initialization for the context sensitive cache
71    * @param context          the context for being sensitive
72    * @param creationFilter   the .ext-file with oAW-Xtend functions
73    */
74   public EmfCloner(EObject context, String creationFilter) {
75     this.context = context;
76     if (creationFilter != creationFilterString) {
77       EmfCloner.creationFilterString = creationFilter;
78       if (null != creationFilter && creationFilter.length() > 0) {
79         EmfCloner.creationFilter = XtendFacade.create(creationFilter);
80         EmfCloner.creationFilter
81             .registerMetaModel(new EmfRegistryMetaModel());
82       }
83     }
84     if (context != null) { /* siehe Listing 5.4 */ }
93   }

95   /**
96    * statically clones an EObject obj in the context of the EObject context
97    * using oAW creationFilter functions
99    * @param obj              the EObject to clone
100   * @param context          the context for being sensitive
101   * @param creationFilter   the .ext-file with oAW-Xtend functions
102   * @return                 the cloned EObject
103   * @throws                 CloneNotSupportedException
104   */
105  public static EObject clone(EObject obj, EObject context, String creationFilter)
106      throws CloneNotSupportedException {
107    EmfCloner emfCloner = new EmfCloner(context, creationFilter);
108    return emfCloner.clone(obj);
109  }
173 }
```

6.6.3 postFilter

Um eine für jeden Knoten unterschiedliche Hintergrundfarbe zu ermöglichen, wurde eine Schnittstelle entwickelt, mit der solche Aspekte – ähnlich den `creationFiltern` – in oAW-Xtend geschrieben werden können.

Die für die Nachverarbeitung veränderte Funktion `create Canvas generate(Model m)` ist im Listing 6.14 dargestellt (s. a. Listing 6.6). Jedes kopierte Objekt wird zuerst im `PostProcessor` (Listing 6.15) registriert (`e2.registerPostProcess(e)`), um anschließend – wenn alle Objekte kopiert und initialisiert wurden – die Nachverarbeitung für alle registrierten Objekte über die Funktion `doPostProcess(m.postProcFilter)` anzustoßen. Diese Nachverarbeitung muss in zwei Schritten ablaufen, da Referenzen in einem kopierten Objekt auf ein anderes, jedoch später kopiertes Objekt erst mit dem Kopieren des Zweiten gesetzt werden.

Listing 6.14: „GenGMF“-Transformationsregeln zum Erstellen des „Graphical Def Model“

```
15 /** Generating the gmfgraph model */
16 create Canvas generate(Model m)
17 :   let fg = new FigureGallery
18 :   let cn = m.getAllNodes()
19 :   let cne = m.getAllFigures()
20 :   newTask("generating GMFGraph model",
21             3 * cne.size + m.compartments.size, 1000)
22 ->  setName(m.name)
23 ->  setFigures({fg})
24 ->  fg.setName('Default')
25 ->  cne.collect(
26         e|  {e.createDescriptorFromTemplate()}
27             .collect(
28                 e2| fg.descriptors.add(e2)
29                 ->  e2.registerPostProcess(e)
30     )       )
31 ->  cn.collect(
32         e|  e.createNodesFromTemplate()
33             .collect(
34                 e2| this.nodes.add(e2)
35                 ->  e2.registerPostProcess(e)
36     )       )
37 ->  m.compartments.collect(
38         e|  e.createCompartmentsFromTemplate()
39             .collect(
40                 e2| this.compartments.add(e2)
41                 ->  e2.registerPostProcess(e)
42     )       )
43 ->  m.edges.collect(
44         e|  {e.createEdgeFromTemplate()}
45             .collect(
46                 e2| this.connections.add(e2)
47                 ->  e2.registerPostProcess(e)
48     )       )
49 ->  cne.collect(
50         e|  e.createLabelsFromTemplate()
51             .collect(
52                 e2| this.labels.add(e2)
53                 ->  e2.registerPostProcess(e)
54     )       )
73 ->  doPostProcess(m.postProcFilter)
74 ;
```

Listing 6.15: Die Klasse `PostProcessor` für die Nachverarbeitung von Elementen

```
24 /**
25  * allows GenGMF to pass objects through some post processing oAW functions
26  */
27 public class PostProcessor {
28   /** the wrapper for the Xtend functions */
29   private static XtendFacade postProcFilter;
30
31   /** map for all used contexts and their cloned objects */
32   private static Map<FigureDesc, List<Object>> postProcObjects =
33     new HashMap<FigureDesc, List<Object>>();
34
35   /**
36    * to be called for each object which needs to be processed later on
38    * @param obj      the cloned object
39    * @param context  the context under which it was copied
40    * @return         the obj
41    */
42   public static Object registerPostProcess(Object obj, FigureDesc context) {
43     List<Object> objects = postProcObjects.get(context);
44     if (null == objects) {
45       objects = new ArrayList<Object>();
46       postProcObjects.put(context, objects);
47     }
48     objects.add(obj);
49     return obj;
50   }
51
52   /**
53    * processes all registered objects with the Xtend functions found in
54    * postProcFilterString
56    * @param postProcFilterString
57    *            an .ext-file containing the oAW-Xtend functions
58    */
59   public static void doPostProcess(String postProcFilterString) {
60     if ((null != postProcFilterString)
61         && (0 != postProcFilterString.length())) {
62       postProcFilter = XtendFacade.create(postProcFilterString);
63       postProcFilter.registerMetaModel(new EmfRegistryMetaModel());
64       for (Entry<FigureDesc, List<Object>> entry : postProcObjects.entrySet()) {
65         FigureDesc key = entry.getKey();
66         for (Object object : entry.getValue()) {
67           postProcess(object, key);
68         }
69       }
70     }
71     postProcObjects.clear();
72     postProcFilter = null;
73   }
74
75   /**
76    * the post processing for a specific object
78    * @param obj      the Object to be processed
79    * @param context  the context under which it will be processed
80    */
81   private static void postProcess(Object obj, FigureDesc context) {
82     try {
83       EObject eobj = (EObject) obj;
84       Object[] params = new Object[] { obj, context };
85       for (String filterName : GenGMFHelper.generateFunctionNames(
86           "filter", eobj, context)) {
87         try {
88           obj = postProcFilter.call(filterName, params);
89           break; // success
90         } catch (IllegalArgumentException e) { /* function not found */ }
91       }
92     } catch (Exception e) {
93       System.err.println(e.getMessage());
94     }
95   }
96 }
```

6.7 Beispiel

Anhand eines kurzen Beispiels sollen im Folgenden die Funktionsweise verdeutlicht sowie die verschiedenen Möglichkeiten von „GenGMF“ demonstriert werden.

6.7.1 Das Metamodell und die Definition für den Editor

Hierfür wird das aus der Abbildung 3.3 bekannte EMF „Domain Model“ erweitert. Das ursprüngliche Metamodell enthielt nur das Hauptelement `NodesModel` sowie dessen Kindelement `Node`, welches mit der `next`-Referenz auf ein anderes `Node`-Element desselben Modells verweisen konnte. In der Abbildung 6.6 ist das hierzu veränderte Metamodell dargestellt.

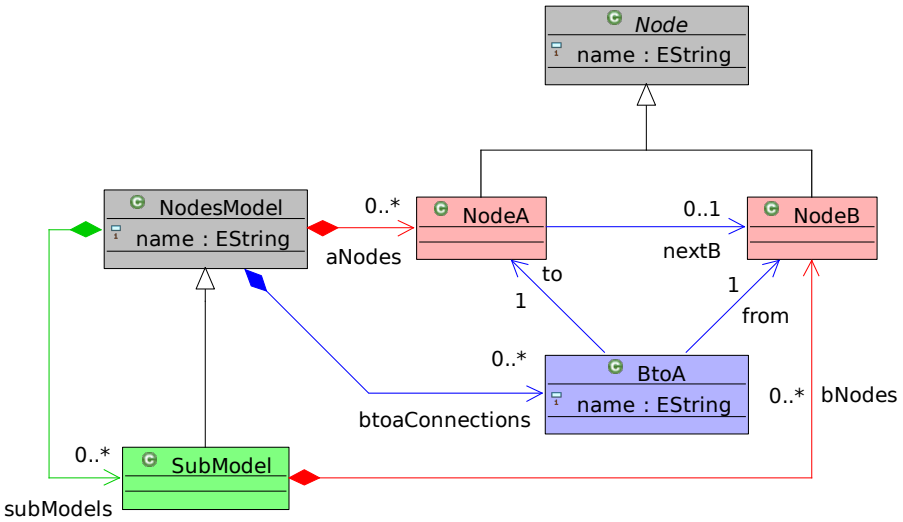


Abbildung 6.6: Das im Vergleich zur Abbildung 3.3a modifizierte EMF „Domain Model“

Das neue Metamodell enthält neben der Referenz-basierten Verknüpfung `nextB` auch die Element-basierte `BtoA`-Verknüpfung. Die für die Referenzen verwendeten Metamodellelemente sind im Diagramm blau eingefärbt. Des Weiteren wird der Einsatz von rekursiven `SubModel`-Compartments demonstriert. Als zusätzliches Kindelement für das Compartment ist der in der Abbildung rot eingefärbte `NodeA`-Knoten erlaubt. Das Element `NodeB` kann im Gegensatz zu `NodeA` kein direktes Kindelement des Hauptelementes `NodesModel` sein. So kann im Rahmen dieses Beispiels auch der Einsatz von Phantom Knoten gezeigt werden (s. a. Kapitel 3.5.2 und 3.5.4). Das `NodeB`-Element, was laut Metamodell ebenfalls ein Kind des Elementes `SubModel` ist, soll als Port neben dem elterlichen Knoten dargestellt werden.

In den folgenden Abbildungen werden die einer Kategorie („Referenzen“, „Knoten“ und „Compartments“) zugeordneten Farben (blau, rot bzw. grün) weiter verwendet, um die Beziehungen zwischen den Modellen zu verdeutlichen. In der Abbildung 6.7 ist das zu dem oben dargestellten Metamodell passende „GenGMF“-Modell abgebildet. Im oberen Bereich sind die Deskriptoren dargestellt. Durch Modell-interne Referenzen wird auf die passenden Templatestrukturen im unteren Bereich verwiesen. Mit den durch die Deskriptoren parametrierten Transformationsregeln entstehen aus den Templatestrukturen die in den Abbildungen 6.8 und 6.9 dargestellten GMF-Modelle. Hier sind die aus dem „GenGMF“-Modell kommenden Strukturelemente farblich gekennzeichnet.

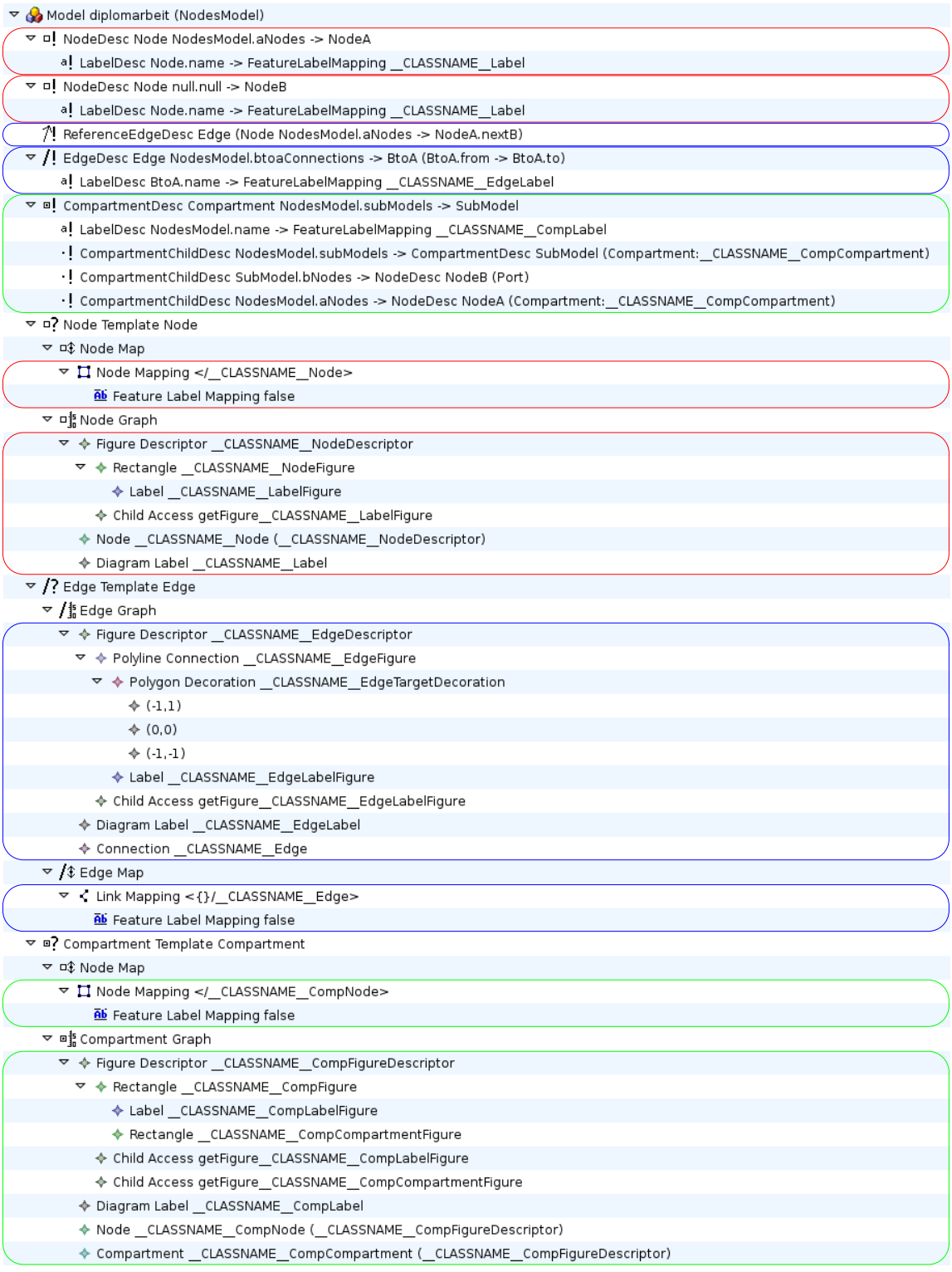


Abbildung 6.7: Das für dieses Beispiel erstellte „GenGMF“-Modell *

* Die für die Abbildung 6.6 verwendeten Farben für die Metamodellelemente und den Referenzen zwischen ihnen wurden auf die Abbildung 6.7 übertragen, um die Verwendung der „GenGMF“-Modellelemente zu verdeutlichen. Die Bedeutung der Farben ist dem Text zu entnehmen.

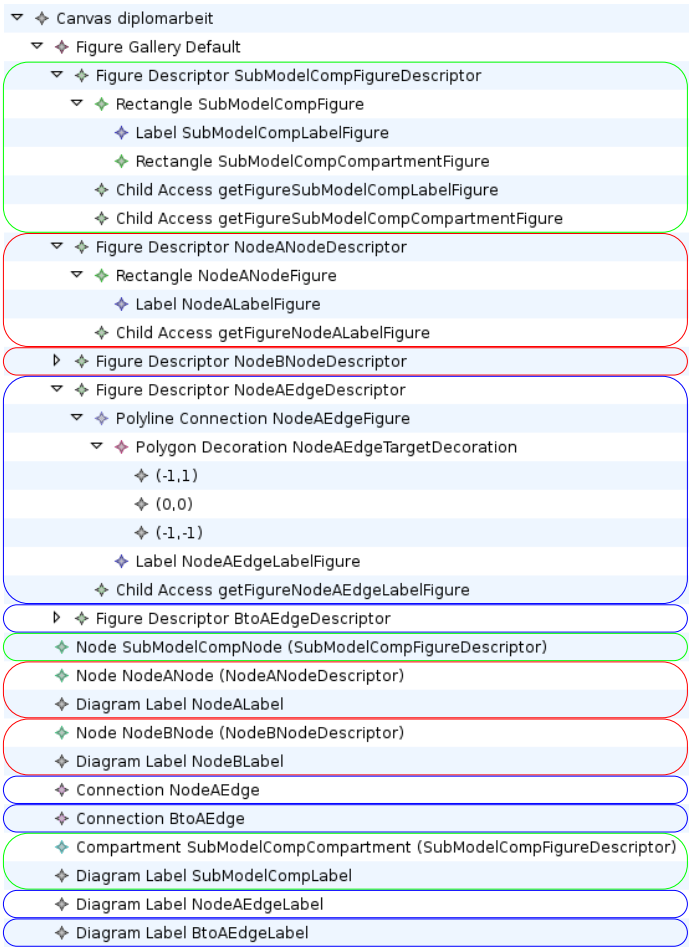


Abbildung 6.8: Das aus dem „GenGMF“-Modell generierte „Graphical Def Model“ *

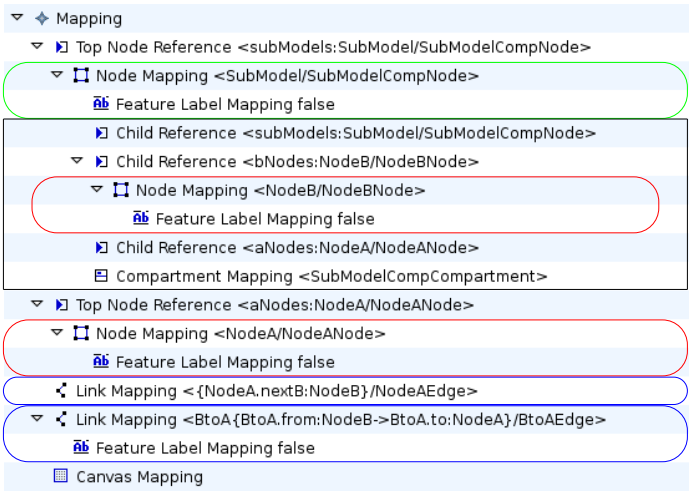


Abbildung 6.9: Das aus dem „GenGMF“-Modell generierte „Mapping Def Model“ *

* Die aus dem „GenGMF“-Modell kopierten Strukturen, die in den beiden Abbildungen 6.8 und 6.9 gezeigt werden, sind entsprechend zu den in den Abbildungen 6.6 und 6.7 verwendeten Farben gekennzeichnet. Die Bedeutung der Farben ist dem Text zu entnehmen.

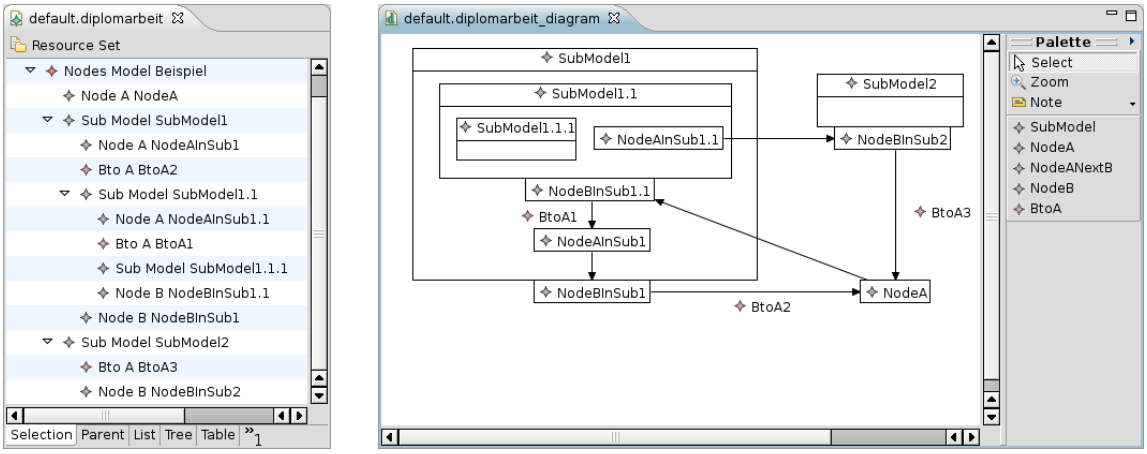
Für das in der Abbildung 6.9 dargestellte „SubModel“-TopNodeReference-Element ( ) werden durch die Transformation die mit einem schwarzen Rahmen markierten und für Compartments notwendigen Elemente dynamisch erstellt (s. a. Kapitel 3.5.2 und 6.4.2).

Zu beachten ist, dass im „Mapping Def Model“ für den Phantom Knoten NodeB automatisch das NodeMapping-Element ( ) als Kind der ChildReference ( ) angelegt wurde, anstatt – wie für NodeA geschehen – eine TopNodeReference ( ) anzulegen.

Des Weiteren fällt auf, dass das nextB-LinkMapping ( ) im Gegensatz zur BtoA-Variante kein FeatureLabelMapping ( ) enthält, obwohl im Template des „GenGMF“-Modells im EdgeMap-Zweig ( ) ein entsprechendes Element vorhanden ist. Hierfür verantwortlich zeigen sich die Zeilen 246 bis 249 des Listings 6.11. Es werden alle FeatureLabelMappings ( ) entfernt, denen durch die vorausgehenden Transformationsschritte kein darzustellendes Attribut zugewiesen wurde.

6.7.2 Der Editor

Die beiden aus den verschiedenen Modellen generierten Editoren sind in der Abbildung 6.10 zu sehen. Hierbei ist in (a) der baumbasierte EMF-Editor abgebildet. Der aus den automatisch erstellten GMF-Modellen generierte grapische Editor ist in (b) dargestellt.



(a) Der EMF-Editor **(b)** Der GMF-Editor

Abbildung 6.10: Die aus dem EMF- und dem „GenGMF“-Modell generierten EMF- und GMF-Editoren

6.7.3 Erweitertes Beispiel

Der erstellte Editor soll nun graphisch etwas ansprechender gestaltet werden. Aus diesem Grund werden für die verschiedenen Knotentypen `NodeA`, `NodeB` und `SubModel` unterschiedliche Hintergrundfarben definiert. Nach dem Setzen der in Kapitel 6.6.3 genannten Einstellungen für die Nachververarbeitung und unter Verwendung der Funktionen aus dem Listing 6.16 werden die Knoten der Typen `NodeA`, `NodeB` und `SubModel` rot, grün bzw. blau dargestellt. Nach einer erneuten Transformation in die GMF-Modelle muss der Editor neu generiert werden. In der Abbildung 6.11 des neuen Editors sind die Knoten nun farbig dargestellt.

Durch die Einführung von eigenen Icons könnte nun die Benutzerfreundlichkeit des Editors weiter gesteigert werden. Hiervon wurde für das die Möglichkeiten von „GenGMF“ demonstrierende Beispiel abgesehen.

Listing 6.16: Filterfunktionen, um die Hintergrundfarbe der Knoten zu ändern

```
1  /**
2   * A NodeA element should be displayed in red
3   */
4  FigureDescriptor filterNodeAFigureDescriptor
5    (FigureDescriptor fd, NodeDesc cd)
6  : fd.actualFigure.setBackgroundColor(255,128,128);
7
8  /**
9   * A NodeB element should be displayed in green
10  */
11 FigureDescriptor filterNodeBFigureDescriptor
12   (FigureDescriptor fd, NodeDesc cd)
13 : fd.actualFigure.setBackgroundColor(128,255,128);
14
15 /**
16  * A SubModel element should be displayed in blue
17  */
18 FigureDescriptor filterSubModelFigureDescriptor
19   (FigureDescriptor fd, CompartmentDesc cd)
20 : fd.actualFigure.setBackgroundColor(128,128,255);
```

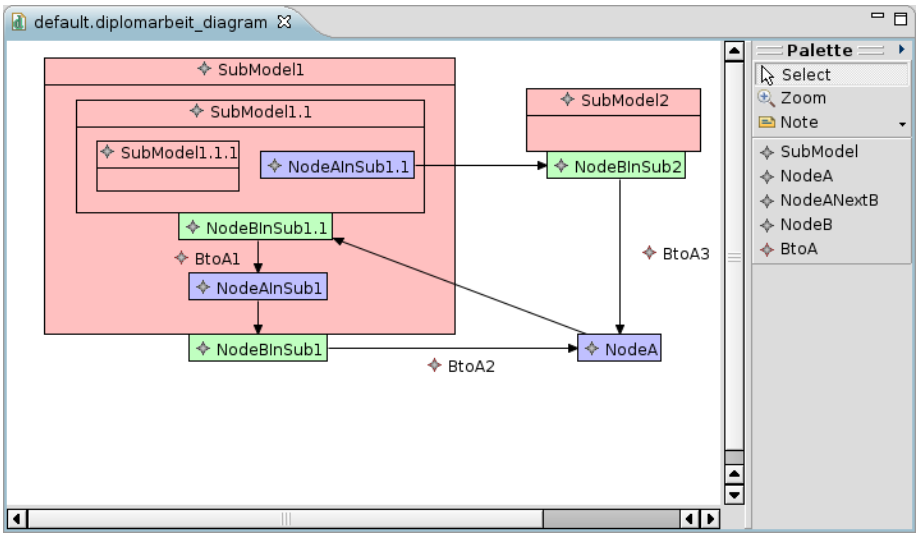


Abbildung 6.11: Der Beispieleditor mit den eingefärbten Knoten

KAPITEL 7

Implementierung / Der graphische `dataflow`-Editor

Wie bereits im letzten Kapitel angesprochen, verwendet der `dataflow`-Editor der „base“-Softwareentwicklungsumgebung 20 verschiedene Knotentypen, damit der Datenfluss in einer Maske modelliert werden kann. Da die Komplexität der GMF-Modelle für einen graphischen Editor überproportional mit der Anzahl der Knotentypen steigt, wurde mit „GenGMF“ eine Entwicklungsumgebung geschaffen, mit deren Hilfe die Komplexität generisch reduziert werden kann (s. a. Kapitel 6). Die folgenden Abschnitte beschreiben die Implementierung des auf dem „GenGMF“ aufbauenden graphischen `dataflow`-Editors.

7.1 Möglichkeiten zur Darstellung von Inhalten in den Knoten

Viele der im `dataflow`-Editor verwendeten Knotentypen enthalten zusätzliche Kindelemente, die das Verhalten dieses `dataflow`-Verarbeitungsschrittes genauer spezifizieren. So enthält z. B. der Typ `Query` (Q) ein `QueryNodeMappingModel` (⇆). Hier können über `QueryParameterMapping`-Kinder (↦) Eingabeparameter (▫) mit Queryparametern (▫) verbunden werden. Das in der Abbildung 7.1 gezeigte `dataflow`-Modell enthält ein solches Mapping mit den Referenzen auf die Entität aus dem `domain`-Modell.

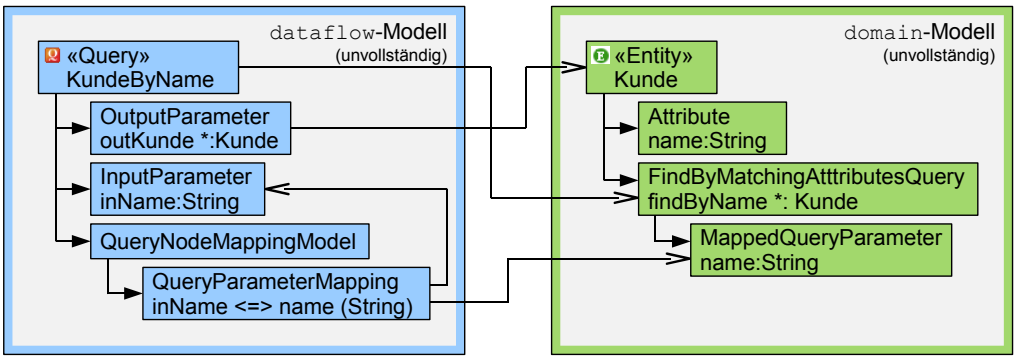


Abbildung 7.1: Schematische Abbildung des `QueryParameterMapping`s (↦) sowie der verknüpften Entität aus dem `domain`-Modell

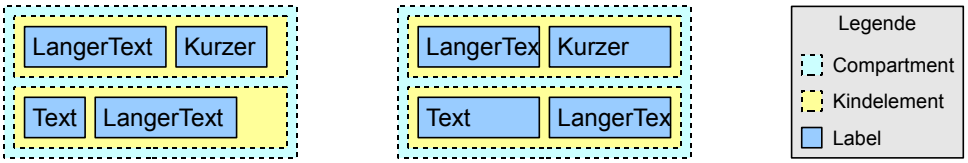
Auch in anderen Knoten wird das Verhalten der „base“-Anwendung durch die Attribute und Kindelemente des Knotens definiert. Ursprünglich sollten, wie in der Anforderung R16 geschrieben, die Eigenschaften des Knotens auch im Knoten angezeigt und bearbeitet werden können. Die Idee des Verfassers war es, die Eigenschaften des Knotens auch im Knoten tabellarisch anzuzeigen und dort bearbeiten zu können. Für die oben genannten Mappingstrukturen bietet sich eine Darstellung in Form einer Tabelle an, in der die gemappten Elemente gegenübergestellt werden. Dies wurde bereits im Rahmen des Pflichtenheftes in der Abbildung 5.2 für den Elementtyp `Query` (Q) gezeigt.

Da sich tabellarische Strukturen nicht mit GMF abbilden lassen, wurden Alternativen gesucht, mit denen sich die Attribute und Kindelemente der Knoten in einer konsistenten Art und Weise darstellen lassen. Die folgenden Abschnitte stellen neben den tabellarischen Strukturen drei weitere Ansätze („Diagrammpartitionierung“, „DummyModel“ und „Outline View“) vor, mit denen Inhalte der Knoten angezeigt und bearbeitet werden können. Mit der „Outline View“ wurde eine Lösung gefunden, mit der das Pflichtkriterium R8 erfüllt werden kann. Die Darstellung der Inhalte „in den Knoten“ aus dem Wunschkriterium R16 kann nicht erfüllt werden.

7.1.1 Tabellarische Darstellung

Optimal für die Darstellung von Mappingstrukturen wäre eine Tabelle. Das GMF stellt jedoch keine Möglichkeit bereit, Tabellenstrukturen anzuzeigen. Um eine Struktur aufzubauen, die einer Tabelle ähnlich sieht, muss in der elterlichen Knotenbeschreibung ein `CompartmentMapping` angelegt werden, das die Knoten von `QueryParameterMapping`-Elementen als Kinder aufnehmen kann. In der Beschreibung des Kindelementes müssen nun zwei `Labels` angelegt werden, die mit Rahmen dargestellt werden, um ein tabellarisches Aussehen zu imitieren.

Die Abbildung 7.2 verdeutlicht die Probleme, die bei einer Realisierung einer Tabellen-ähnlichen Struktur auftreten können. Wenn keine Positions- und Größenangaben für das `Label` gesetzt werden, kann eine Darstellung ähnlich der in der Abbildung (a) gezeigten entstehen, da der von GMF für das Kind des Compartments verwendete Layoutmanager keine Kenntnis über die Labels der anderen Kinder des Compartments hat. Werden dagegen die Positions- und Größenangaben pixelgenau gesetzt, kann es passieren, dass die dargestellten Texte abgeschnitten werden. Dieses Verhalten wird in der Abbildung (b) gezeigt.



(a) Keine Tabellenstruktur mit variablen Angaben

(b) Abgeschnittene Texte mit festen Angaben

Abbildung 7.2: Probleme bei der Implementierung einer Tabellen-ähnlichen Struktur mit GMF

Um trotzdem eine tabellarische Darstellung zu erreichen, müsste für das Compartment und für die Kindelemente jeweils ein eigener Layoutmanager implementiert werden. Zusammen könnte die gewünschte Darstellung ermöglicht werden. Der hierfür notwendige Aufwand wurde – für die Umsetzung im Rahmen dieser Diplomarbeit – als zu hoch eingestuft.

7.1.2 Diagrammpartitionierung

Das Partitionieren von Diagrammen gestattet es dem Entwickler des Editors, ein komplexes hierarchisches Modell so aufzuteilen, dass bei einem Doppelklick auf einen Knoten ein separates Diagramm für den Knoten geöffnet wird (s. a. Kapitel 3.5.5). Für den dataflow-Editor wäre dies für einzelne Knotentypen ein denkbares Vorgehen. Bei konsequenter Verwendung dieses Ansatzes müsste für jedes der sechs möglichen Kindmodelle (`QueryParameterMapping`, `TableModel`, `ClassificationModel`, `MatrixClassificationModel`, `CompositeModel` sowie `ViewNodeMappingModel`) ein eigener Diagrammeditor entwickelt werden. Diese Editoren müssten aus den fünf Knotentypen (`Query`, `Table`, `ClassifierTable`, `Matrix`

sowie `Composite` (✧)), in insgesamt zehn Kombinationen (s. a. Tabelle 4.3), aufgerufen werden. Aufgrund des Umfanges wurde dieser Ansatz jedoch nicht weiter verfolgt.

7.1.3 DummyModel

Einige Knotentypen des `dataflow`-Metamodells enthalten Knoten-spezifische Eigenschaften direkt im Knoten und nicht in einem eigenständigem Metamodellelement, das als Kind in den Knoten eingefügt werden könnte. Um eine konsistente Darstellung der Knoten zu erreichen, müssten diese Eigenschaften so dargestellt werden, als ob die Inhalte in einem Kindelement enthalten sind. Dies kann mit den Mitteln des GMF jedoch nicht realisiert werden.

Aus diesem Grund sollte für Knotentypen, die kein Modellelement als Kind enthalten, ein „DummyModel“-Element in einer transienten `Containment`-Referenz eingeführt werden, um die in der Hauptklasse enthaltenen Attribute in einem Kind anzuzeigen. Hiermit sollte erreicht werden, dass alle Knotentypen ein Kindelement besitzen, in dem die wichtigsten Informationen gebündelt angezeigt werden. Damit würden alle Knotentypen konsistent mit einem Compartment angezeigt werden, was es wiederum dem Benutzer des Editors vereinfacht, die wichtigsten Informationen eines Knotens zu finden und zu bearbeiten.

Die Abbildung 7.3 zeigt ein „DummyModel“, das für zwei Knotentypen implementiert wurde. Da sich der Aufwand für die Implementierung eines Kindelementes inklusive dem Reflektieren der Attribute des elterlichen Elementes und dem Darstellen mit den Mitteln von GMF als zu umfangreich herausgestellt hat, wurde diese Variante verworfen.

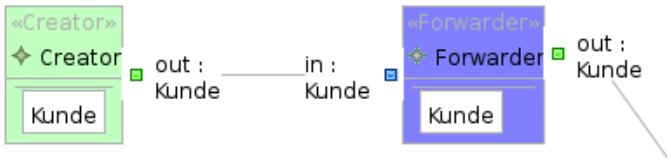


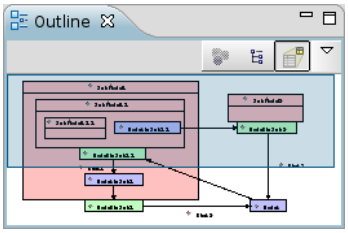
Abbildung 7.3: Darstellung von Knoten mit einem „DummyModel“ in einem Compartment

7.1.4 Outline View

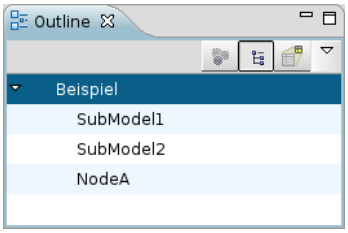
Das GMF stellt für die generierten Editoren eine Outline View bereit, die das gesamte Diagramm als Navigationsansicht aus der Vogelperspektive darstellt. Hierbei wird der aktuell im Editor sichtbare Bereich durch einen halbtransparenten Rahmen hervorgehoben. Zusätzlich besteht die Möglichkeit, alle direkt auf der Diagrammfläche platzierten Elemente in einer Baumansicht anzuzeigen. Beide Ansichten sind mit dem graphischen Editor synchronisiert, wodurch der Editor im Falle eines Klicks in der Outline View den sichtbaren Ausschnitt entsprechend verschiebt, um z. B. das angeklickte Element anzuzeigen. Für den im Kapitel 6.7 erstellten graphischen Editor sind die beiden Ansichten in der Abbildung 7.4a/b dargestellt.

In der Abbildung 7.4b sind nicht alle Elemente des Modells aus der Abbildung 6.10a dargestellt. Des Weiteren fehlen in der generierten Implementierung für die Outline-View nicht nur die Icons, die in den EMF- und GMF-Editoren Verwendung finden, sondern auch jegliche Editiermöglichkeit, so wie sie in der Outline View des EMF-Editors vorhanden ist.

Jacques Lescot, ein Entwickler des EcoreTools-Projektes ([UrlEclc]), hat mit der im Eclipse Bugzilla Eintrag 206778 ([Les08]) publizierten Modifikation des GMF-Generators eine Möglichkeit geschaffen, die EMF-Baumdarstellung für GMF-basierte Editoren in die Outline View zu integrieren.



(a) Navigationsansicht



(b) Baumansicht

Abbildung 7.4: Von GMF generierte Ansichten für die Outline View

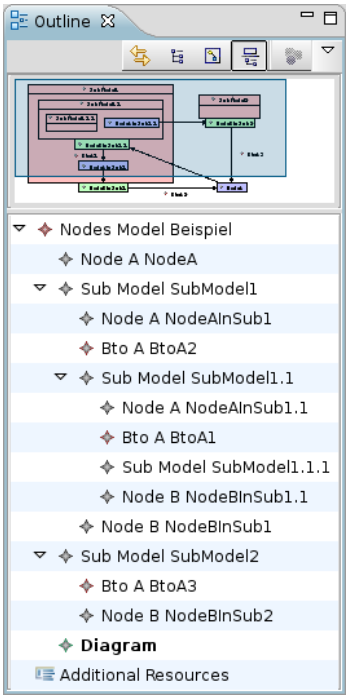


Abbildung 7.5: Modell-orientierte Outline View von Jacques Lescot

Die neue Outline View ist in der Abbildung 7.5 dargestellt. Mit dem integrierten Synchronisationsmodus ( ) wird immer dasselbe Element in beiden Ansichten (graphisch und im Baum) angezeigt. Im Rahmen dieser Diplomarbeit wurden darauf aufbauend diverse Editiermöglichkeiten wie die Standardfunktionen Kopieren, Einfügen und Löschen sowie „Drag’n’Drop“ hinzugefügt.

7.2 Struktur des für den Editor verwendeten „GenGMF“-Modells

Der Aufbau des für den `dataflow`-Editor notwendigen „GenGMF“-Modells wird in diesem Abschnitt auf der Basis der in den Kapiteln 6.3 und 6.4 beschriebenen Definitionen erläutert. Für den `dataflow`-Editor müssen die drei grundlegend verschiedenen Strukturen der Knoten, deren Parameter sowie der Kanten modelliert werden. Für jede dieser Strukturen wurde ein Template angelegt, das von den Deskriptoren referenziert werden kann (s. a. Kapitel 6.3 und 6.4). Die Knoten haben in diesem Fall ein `CompartmentTemplate` ( ) erhalten. Nur so können die durch die Anforderung R3 verlangten Portstrukturen abgebildet werden (s. a. Kapitel 6.4.3). Für die Ein- ( ) und Ausgabeparameter ( ) wurde ein `NodeTemplate` ( ) und für die Kanten ein `EdgeTemplate` ( ) erstellt. In den folgenden Kapiteln werden die Templates ausführlich diskutiert.

7.2.1 Darstellung von Knoten

Die in R1 geforderten Rechtecke lassen sich in GMF mit dem `Rectangle`-Element ( ) abbilden. In der Abbildung 7.6 ist das Element in der Zeile sieben dargestellt. Das Setzen der Hintergrundfarbe aus dem Pflichtkriterium R2 für die Knoten wurde mit der in den Kapiteln 6.6.3 und 6.7.3 beschriebenen Nachverarbeitung von Knoten implementiert.

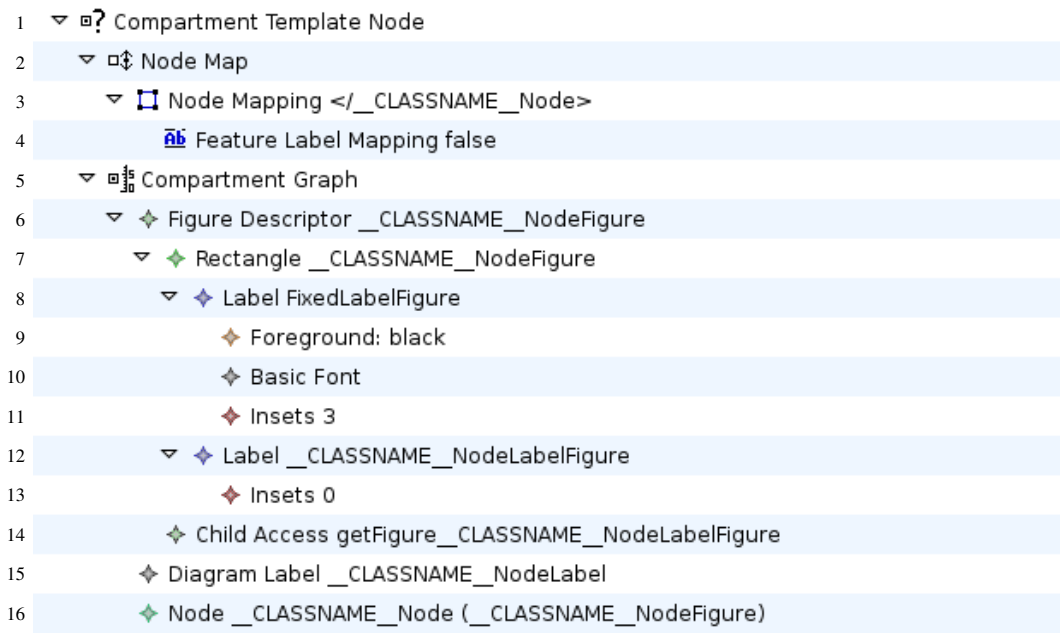


Abbildung 7.6: Das `CompartmentTemplate` (▫?) des „GenGMF“-Modells für den `dataflow`-Editor

7.2.2 Inhalte der Knoten

In den im Pflichtenheft (Kapitel 5.2) genannten Kriterien werden verschiedene Inhalte für die Darstellung benannt. Diese lassen sich in die drei Kategorien „Icons“, „Textuelle Inhalte“ sowie „Zusätzliche Inhalte“ unterteilen.

Icons

Das GMF stellt von sich aus das zu einem Metamodellelement gehörende Icon auf der linken Seite eines Labels dar, wenn das Element `DiagramLabel` (✧) über ein `FeatureLabelMapping` (Ab) mit einem Attribut eines Metamodellelementes verbunden wurde. Aus diesem Grund musste für die in R5 geforderte Darstellung von Icons nichts verändert werden. Im Gegensatz zu der in der Abbildung 5.2 dargestellten Positionierung des Icons befindet es sich nun allerdings neben dem Namen des Elementes und nicht neben dem Elementtyp.

Für einige Metamodellelemente wurden neue Icons in den Editor eingefügt (R15). Dies betrifft die Elementtypen `Creator` (), `Custom` (?), `Forwarder` (), `Selector` (), `Setter` () sowie `ClassificationModel` ().

Textuelle Inhalte

Um die in Kapitel 5 besprochenen Namen der Knotentypen in die Templates zu integrieren (R6), musste die Zeichenkette „«`__CLASSNAME__`»“ als Text für das Label „FixedLabelFigure“ (Zeile 8 in der Abbildung 7.6) eingetragen werden. Im Rahmen der „GenGMF“-Transformation (s. a. Kapitel 6.5.1) wird „`__CLASSNAME__`“ mit dem Namen des Metamodellelementes ersetzt. Damit ist hierfür kein zusätzlicher Aufwand erforderlich. In der Newsgroup „eclipse.modeling.gmf“ ist diese Funktionalität bereits unter dem Titel „Metadata Feature Label“ ([Boz08]) benannt worden.

Da der Zeichensatz „UTF8“ im GMF nicht konsistent verwendet wird, mussten die Guillimotzeichen „«“ und „»“ durch die entsprechenden Unicoderepräsentationen „\u00AB“ und „\u00BB“ ersetzt werden. Deshalb wurde letztendlich die Zeichenkette „\u00AB__CLASSNAME__\u00BB“ im „GenGMF“-Modell verwendet.

Um den vom Entwickler vergebenen Namen im Knoten anzuzeigen und auch bearbeiten zu lassen, muss eine Referenzkette aus `Label`- (✦), `ChildAccess`- (✦) und `DiagramLabel`-Element (✦) aufgebaut werden (s. a. Kapitel 3.4).

Die textuellen Inhalte wurden vom umgebenden Rahmen abgesetzt, indem in die beiden `Label`-Objekte (✦) jeweils ein `Insets`-Element (✦) eingefügt wurde (s. a. Zeilen 11 und 13 der Abbildung 7.6). Hier wird der Abstand nach außen festgelegt. Ein Abstand von 3 Pixeln wurde hier als ansprechend empfunden und entsprechend gesetzt. Der Abstand zwischen den beiden Labels wurde auf 0 reduziert. In der Abbildung 7.7 werden die Einstellungen dargestellt.

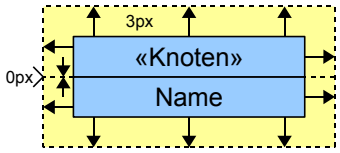


Abbildung 7.7: Die gesetzten Insets für die Labels der Knoten

Zusätzliche Inhalte

Die für das Wunschkriterium R16 darzustellenden Inhalte wurden nicht implementiert, da der Aufwand hierfür zu hoch ist (s. a. Kapitel 7.1). Stattdessen wird es, mit Hilfe der erweiterten Outline View, ermöglicht, das gesamte Modell zu bearbeiten (Pflichtkriterium R8). Zusätzlich wird dadurch die Übersichtlichkeit des Diagramms gewährleistet, da erweiterte Informationen in den Eclipse Views „Properties“ und „Outline“ angezeigt werden.

7.2.3 Darstellung von Ein- (▫) und Ausgabeparametern (▫)

Die Templates für Knoten und für die Parameter unterscheiden sich vor allem in zwei Punkten. Erstens hat der Parameter nur ein Label und zweitens wird das Rechteck des Parameters ohne Rahmen dargestellt. Letzteres wird erreicht, indem die Eigenschaft „Outline“ für das Rechteck auf „false“ gesetzt wird. Die Definition des Templates ist in der Abbildung 7.8 zu sehen.

Das Label stellt eine zusammengesetzte Zeichenkette dar, die den Namen und den Namen des Typs enthält. Da GMF selbst keine Labels unterstützt, mit denen Attribute in referenzierten Elementen angezeigt werden können, wird mit einem Aspekt die Methode überschrieben, welche die anzuzeigende Zeichenkette zusammensetzt. In dem Aspekt wird, wie auch schon aus den Projekten „base“ und „GenGMF“ bekannt, eine oAW Xtend Funktion aufgerufen, die eine fertig formatierte Zeichenkette zurückgibt (s. a. Abbildung 4.2 sowie die Listings 6.2 und 6.1). Im Fall der Ein- (▫) und Ausgabeparameter (▫) ist diese mit einem Zeilenumbruch versehen, um das Seitenverhältnis des Parameters zu reduzieren. Hierbei werden in der ersten Zeile der Name des Parameters sowie optional ein Sternchen („*“) für die `MANY`-Kardinalität und in der zweiten Zeile wird der Name des Typs ausgegeben (R7).

GMF unterstützt in den Modellen nur die Anzeige von Icons auf der linken Seite eines Labels. Die Ein- (▫) und Ausgabeparameter (▫) sollen jedoch an unterschiedlichen Seiten platziert werden

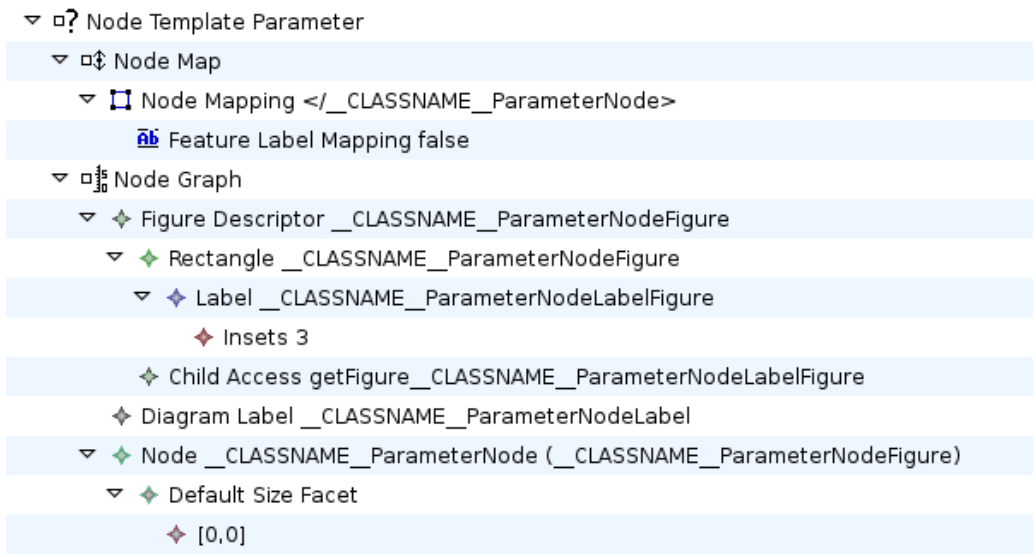


Abbildung 7.8: Das `NodeTemplate` (▫?) des „GenGMF“-Modells für die Ein- (▪) und Ausgabeparameter (▪)

(siehe Kriterium R13). Um das Icon sowohl für die Eingabeparameter (▪) als auch für die Ausgabeparameter (▪) in der Nähe des Elternknotens darzustellen, muss das Eingabeparameter-Icon (▪) auf der rechten Seite platziert werden. Die Umsetzung erfolgte mit dem im Listing 7.1 dargestellten Aspekt, der für das `WrapLabel` eines Eingabeparameters (▪) die Positionen des Textes und des Icons vertauscht.

Listing 7.1: Vertauschen des Icons mit dem Text in der Darstellung von Eingabeparametern (▪)

```
1  public aspect SwapInputParameterImage {
2      /**
3       * Captures the new call for a WrapLabel if called
4       * inside a InputParameterParameterNodeFigure class.
5       */
6      pointcut swapInputParameterImage():
7          call(public WrapLabel.new())
8      &&  within(*..InputParameterParameterNodeFigure)
9      ;
10
11     /**
12      * Exchanges the position if the text and the icon and
13      * aligns the text to the right side.
14      */
15     after() returning(WrapLabel l): swapInputParameterImage() {
16         l.setTextPlacement(PositionConstants.WEST);
17         l.setTextAlignment(PositionConstants.EAST);
18     }
19 }
```

Um das gezeigte `NodeTemplate` (▫?) (Abbildung 7.8) mit den Ein- (▪) und Ausgabeparametern (▪) zu verknüpfen, müssen zwei `NodeDesc`-Elemente (▫!) definiert werden. Sie sind in der Abbildung 7.9 dargestellt.

In der Anforderung R3 steht, dass die Parameter außerhalb der Knoten als Port darzustellen sind. Die hierfür notwendigen `CompartmentDesc`-Elemente (▫!) sind in der Abbildung 7.10 darge-

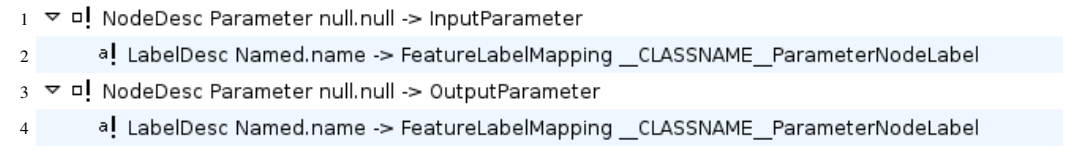


Abbildung 7.9: Die `NodeDesc`-Elemente (▫!) für das Mapping der Ein- (▪) und Ausgabeparameter (▪)

stellt. Exemplarisch ist hier nur der Baum für den `Creator`-Knoten (◔) aufgeklappt. Die beiden `NodeDesc`-Elemente (▫!) für die Ein- (▪) und Ausgabeparameter (▪) aus der Abbildung 7.9 sind in den Zeilen drei und vier der Abbildung 7.10 durch die Verwendung von `CompartmentChildDesc`-Elementen (·!) als Portstrukturen definiert.

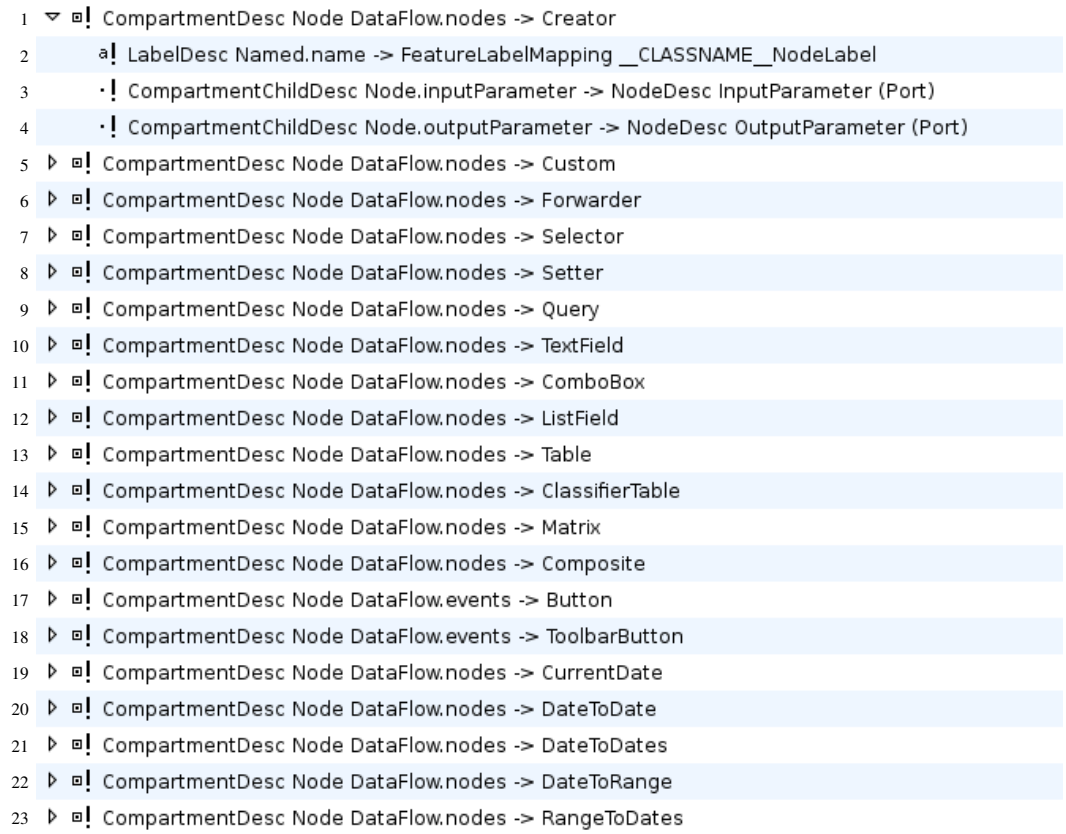


Abbildung 7.10: Die `CompartmentDesc`-Elemente (▫!) für das Mapping der Knoten

7.2.4 Darstellung von Kanten

Im `dataflow`-Metamodell existieren zwei verschiedene Arten von Verbindungen. Ein Ausgabeparameter (▪) wird durch die `NodeConnection` (➔) mit einem Eingabeparameter (▪) verbunden. Objekte, die Ereignisse „senden“ können (`Button` (B) und `ToolbarButton` (✦)), werden durch eine `NodeEventConnection` (⚡) mit einem Knoten verbunden (s. a. Abschnitt „Interaktive Oberflächenelemente“ im Kapitel 4.2.5). Das für die beiden Arten verwendete `EdgeTemplate` (/?) ist in der Abbildung 7.11 gezeigt. Hierbei wird eine geschlossene schwarze Linie mit einer geschlossenen Pfeilspize (`PolygonDecoration` (✦)) definiert.

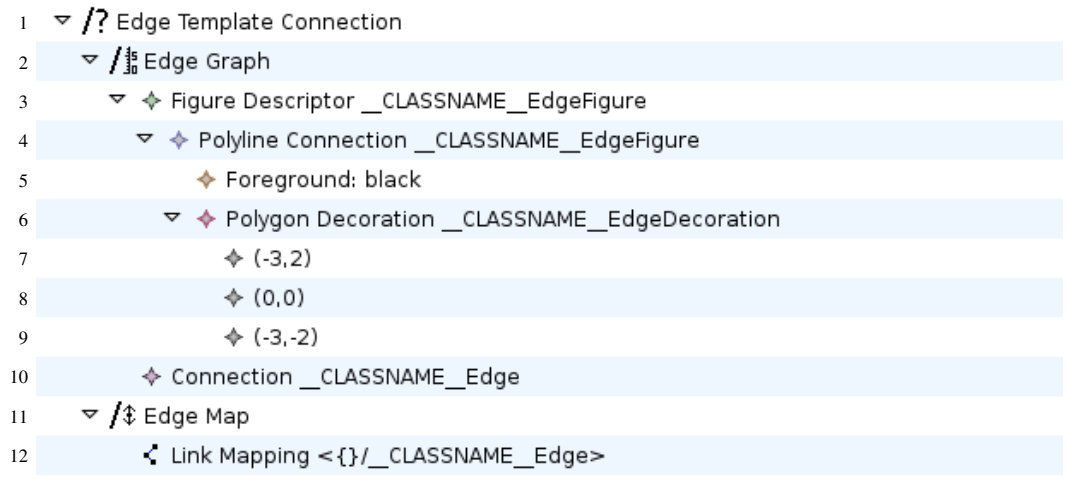


Abbildung 7.11: Das `EdgeTemplate` (/?) des „GenGMF“-Modells für Verbindungen

Um die in der Anforderung R4 gewünschte optische Differenzierbarkeit zu implementieren, wird die Pfeilspitze der `NodeEventConnection` (⚡) geöffnet dargestellt. Dies geschieht durch die Verwendung einer `PolylineDecoration` (✦). Der hierfür notwendige creation-Filter wurde bereits im Kapitel 6.6.2 erklärt. Zusätzlich wird die Verbindungslinie gestrichelt gezeichnet, was mit dem im Listing 7.2 dargestellten zusätzlichen Nachverarbeitungsfilter erreicht wird (s. a. Kapitel 6.6.3).

Listing 7.2: Funktion zum Setzen der gestrichelten Linienart für die `NodeEventConnection` (⚡)

```
1 FigureDescriptor filterNodeEventConnectionFigureDescriptor
2         (FigureDescriptor fd, EdgeDesc ed)
3 :   ((Polyline)fd.actualFigure).setLineKind(LineKind::LINE_DASH)
4 ->  ((Polyline)fd.actualFigure).targetDecoration.setLineWidth(2)
5 ;
```

7.3 Verhalten des Editors

Grundsätzlich bieten die von GMF generierten Editoren eine einfache Benutzerinteraktion. So können z. B. Verbindungen nur zwischen den im Metamodell vorgegebenen Endpunkten für die Verbindung gezogen und Parameter nur als Ports neben den Knoten platziert werden.

7.3.1 Platzierung der Ein- (▫) und Ausgabeparameter (▫) an den Seiten der Knoten

Die in den generierten Editoren verwendete Klasse des GMF „`BorderItemLocator`“ implementiert den Algorithmus für die Positionierung von Ports. Leider wirkt sich die in den GMF-Modellen mögliche Angabe der „Affixed Parent Side“ nur auf die initiale Position relativ zum Elternknoten aus und der Port lässt sich trotzdem auf eine andere Seite verschieben. Genau dieses Verhalten ist jedoch nach der Anforderung R13 nicht gewünscht. Der originale Quellcode enthält einige für den Algorithmus wichtige Methoden, von denen eine jedoch als `private` markiert ist. Aus diesem Grund konnte die Funktionalität nicht durch Ableitung modifiziert werden und der Quellcode musste in eine eigene Klasse „BorderItemLocatorEx“ kopiert und angepasst werden. Die Listings 7.4 und 7.5 zeigen einige wichtige Bestandteile für den neuen Algorithmus. Im Listing 7.5 sind die bereits

im originalen Algorithmus vorhandenen und gleichartigen `if-else` Blöcke nur in einer verkürzten Variante dargestellt. Um den neuen Algorithmus zu verwenden, wird mit einem Aspekt (siehe Listing 7.3) die Methode `addFixedChild` in den EditPart-Klassen des generierten Editors verändert.

Listing 7.3: Der Aspekt zur Integration des „`BorderItemLocatorEx`“ in die Anwendung

```
9  public aspect ParameterSideConstraint {
10   /** intecepts the addFixedChild method in the EditParts */
11   pointcut addFixedChildNewBIL(AbstractBorderedShapeEditPart myThis,
12       EditPart childEditPart) :
13     execution( protected boolean *..*EditPart.addFixedChild(EditPart))
14     && args( childEditPart ) && target(myThis);

16   /**
17    * If the classname for the childEditPart parameter matches
18    * "{In|Out}putParameter*EditPart" a BorderItemLocatorEx object
19    * will be initilialized for the child.
20    */
21   boolean around(AbstractBorderedShapeEditPart myThis, EditPart childEditPart):
22       addFixedChildNewBIL( myThis, childEditPart ) {
23     String partClassName = childEditPart.getClass().getSimpleName();
24     // {In|Out}putParameter...EditPart
25     if (partClassName.endsWith("EditPart")) {
26       int constraint = 0;
27       if (partClassName.startsWith("InputParameter")) {
28         constraint = PositionConstants.NORTH_WEST;
29       } else if (partClassName.startsWith("OutputParameter")) {
30         constraint = PositionConstants.SOUTH_EAST;
31       }
32       if (constraint!=0){
33         IBorderItemLocator locator = new BorderItemLocatorEx(
34             myThis.getMainFigure(), constraint);
35         myThis.getBorderedFigure().getBorderItemContainer().add(
36             ((AbstractBorderItemEditPart) childEditPart).getFigure(), locator);
37         return true;
38       }
39     }
40     return proceed(myThis, childEditPart);
41   }
42 }
```

Listing 7.4: Der in „`BorderItemLocatorEx`“ verwendete Algorithmus (Teil 1)

```
57  /** the constraints on where to place the border item */
58  private int constraintSides;

69  /**
70   * The constraint side determines placement of figure.
71   *
72   * @param parentFigure
73   * @param constraintSide
74   *            the constraint side of the parent figure on which to place
75   *            this border item as defined in {@link PositionConstants}
76   */
77  public BorderItemLocatorEx(IFigure parentFigure, int constraintSide) {
79    this.parentFigure = parentFigure;
80    this.constraintSides = toClockwiseBitField(constraintSide);
81  }

432 /**
433  * determines the closest side according to the constraints.
434  *
435  * @param closestSide
436  */
437 private int restrictToSideConstraints(int closestSide) {
438   closestSide = toClockwiseBitField(closestSide);
439   if ((constraintSides & closestSide) != closestSide) {
440     // 4 bit roll right by 1
441     int closestSide2 = (closestSide >> 1) | ((closestSide & 1) << 3);
442     if ((constraintSides & closestSide2) != closestSide2) {
443       closestSide = closestSide2;
444     } else {
445       // 4 bit roll left by 1
446       closestSide2 = (closestSide >> 3) | ((closestSide & 7) << 1);
447       if ((constraintSides & closestSide2) != closestSide2) {
448         closestSide = closestSide2;
449       } else {
450         // the opposite
451         closestSide2 = (closestSide >> 2)
452             | ((closestSide & 3) << 2);
453         if ((constraintSides & closestSide2) != closestSide2) {
454           closestSide = closestSide2;
455         } else {
456           throw new IllegalArgumentException();
457         }
458       }
459     }
460   }
461   closestSide = toPositionConstantsBitField(closestSide);
462   return closestSide;
463 }

465 /**
466  * converting to a {@link PositionConstants} NORTH(1), SOUTH(4), WEST(8),
467  * EAST(16) bit field
468  *
469  * @param sides
470  * @return
471  */
472 private static int toPositionConstantsBitField(int sides) {
473   // Moving to bit 2
474   return (sides & 13) | ((sides << 3) & PositionConstants.EAST);
475 }

477 /**
478  * converting to a clockwise NORTH(1), EAST(2), SOUTH(4), WEST(8) bit field
479  *
480  * @param sides
481  * @return
482  */
483 private static int toClockwiseBitField(int sides) {
484   // Moving from bit 2
485   return (sides & 13) | ((sides & PositionConstants.EAST) >> 3);
486 }
```

Listing 7.5: Der in „`BorderItemLocatorEx`“ verwendete Algorithmus (Teil 2)

```
266   /**
267    * The preferred side takes precendence.
268    *
269    * @param suggestedLocation
270    * @param suggestedSide
271    * @param circuitCount
272    *            recursion count to avoid an infinite loop
273    * @return point
274    */
275   private Point locateOnBorder(Point suggestedLocation, int suggestedSide,
276       int circuitCount, IFigure borderItem) {
277     Point recommendedLocation = locateOnParent(suggestedLocation,
278         suggestedSide, borderItem);
279     int vertical_gap = MapModeUtil.getMapMode(getParentFigure()).DPtoLP(8);
280     int horizontal_gap = MapModeUtil.getMapMode(getParentFigure())
281         .DPtoLP(8);
282     Dimension borderItemSize = getSize(borderItem);
283     boolean forceNextSide = suggestedSide != restrictToSideConstraints(suggestedSide)
284         && circuitCount < 4;
285     if (forceNextSide || (circuitCount < 4)
286         && conflicts(recommendedLocation, borderItem)) {
287       if (suggestedSide == PositionConstants.WEST) {
288         if (!forceNextSide)
289           do {
290             recommendedLocation.y += borderItemSize.height
291                 + vertical_gap;
292           } while (conflicts(recommendedLocation, borderItem));
293         if (forceNextSide
294             || recommendedLocation.y > getParentBorder()
295                 .getBottomLeft().y
296                 - borderItemSize.height) { // off the bottom,
297           // wrap south
298           return locateOnBorder(recommendedLocation,
299               PositionConstants.SOUTH, circuitCount + 1,
300               borderItem);
301         }
302       } else if (suggestedSide == PositionConstants.SOUTH) {
303 
318       } else if (suggestedSide == PositionConstants.EAST) {
319 
332       } else { // NORTH
333 
344       }
345     }
346     return recommendedLocation;
347   }
348 
377   /**
378    * Find the closest side when x,y is inside parent.
379    *
380    * @param proposedLocation
381    * @param parentBorder
382    * @return draw constant
383    */
384   public int findClosestSideOfParent(Rectangle proposedLocation,
385       Rectangle parentBorder) {
387     Point parentCenter = parentBorder.getCenter();
388     Point childCenter = proposedLocation.getCenter();
389     int closestSide;
390     if (childCenter.x < parentCenter.x) // West, North or South.
392       if (childCenter.y < parentCenter.y) // west or north
394         // closer to west or north?
395         Point parentTopLeft = parentBorder.getTopLeft();
396         if ((childCenter.x - parentTopLeft.x) <= (childCenter.y - parentTopLeft.y)) {
397           closestSide = PositionConstants.WEST;
398         } else {
399           closestSide = PositionConstants.NORTH;
400         }
401       } else { // west or south
402 
408       }
409     } else { // EAST, NORTH or SOUTH
410 
426     }
428     closestSide = restrictToSideConstraints(closestSide);
429     return closestSide;
430   }
```

Für die gewünschte Funktionalität wurde ein Parameter `constraintSides` eingeführt, mit dem die Positionierung von Ports auf ein oder mehrere Seiten beschränkt werden kann. In der von Eclipse zur Verfügung gestellten Klasse `PositionConstants` existieren für jede Himmelsrichtung eine Konstante (`NORTH` = 1, `SOUTH` = 4, `WEST` = 8 und `EAST` = 16) sowie durch ein Bit-weises „Oder“ kombinierte Bitmasken (z. B. `NORTH_WEST` = `NORTH` | `WEST`). Da der originale Algorithmus auf der Verwendung dieser Konstanten aufbaut, wurden diese ebenfalls für die neuen Parameter verwendet.

Bei der Verwendung als Bitmaske jedoch fällt auf, dass die Wertigkeit 2 unbelegt ist und außerdem die Werte nicht im oder gegen den Uhrzeigersinn verteilt sind. Würde dagegen die Konstante `EAST` mit dem Wert 2 belegt sein, könnte mit einer einfachen 4-Bit-Rotation die Himmelsrichtung um 90° geändert werden. Um den Algorithmus einfacher zu gestalten, wurden hier die zwei Methoden `toClockwiseBitField` und `toPositionConstantsBitField` eingeführt, die zwischen den zwei Bitmasken umrechnen.

In der neuen Funktion `restrictToSideConstraints` wird zu einer gegebenen Seite die Seite gesucht, die zu den in der Variablen `constraintSides` angegebenen Seiten passt. Hierzu wird ausgehend von der aktuellen Position nach links, rechts und zur gegenüberliegenden Seite „gerollt“, um die Seitenbeschränkung zu prüfen. Als Ergebnis wird die am „nächsten“ zur übergebenen Seite Liegende zurückgegeben.

Die beiden bereits in der originalen Version enthaltenen Methoden `locateOnBorder` sowie `findClosestSideOfParent` (siehe Listing 7.5) wurden angepasst, um die oben besprochene `restrictToSideConstraints`-Funktion aufzurufen. Für Erstere wurde zusätzlich der Algorithmus leicht modifiziert, um die Suche nach einem Platz für den Port zu vereinfachen.

Die Abbildung 7.12 zeigt, wie die Ein- (▫) und Ausgabeparameter (▫) nun neben den Knoten platziert werden können.

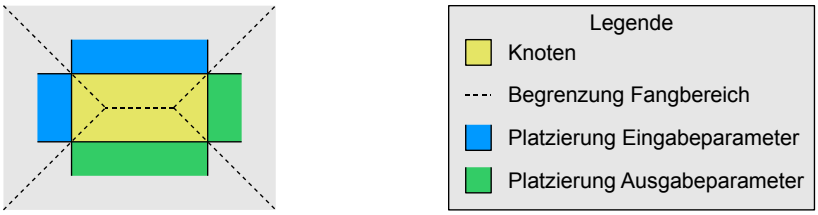


Abbildung 7.12: Platzierung der Ein- (▫) und Ausgabeparameter (▫) anhand von Fangbereichen

Behebung von Fehlern im Layoutalgorithmus von GMF

Während der Benutzung des Editors ist aufgefallen, dass die Ports nicht an den Null-Positionen (links bzw. oben) in den Positionierungsbereichen platziert werden konnten.

Die Analyse ergab, dass dies ein Fehlverhalten des oben besprochenen originalen Algorithmus ist. Der orginale Algorithmus initialisiert die Position mit Null-Werten (0) und nimmt gleichzeitig an, dass dies eine ungültige Position ist. In diesem Fall wurde eine Neuberechnung der Position durchgeführt.

Mit der Umstellung der Initialisierung auf den Objektwert `null` und entsprechenden Prüfungen beim Zugriff auf die Objektreferenz konnte dieser Fehler behoben werden.

7.3.2 Ziehen von Verbindungen

Das `dataflow`-Metamodell gestattet für Eingabeparameter ( ) nur ein `NodeConnection` ( )-Objekt als eingehemde Verbindung. Der generierte Editor gestattet jedoch das Ziehen einer Verbindung auch dann, wenn bereits eine andere Verbindung für den Eingabeparameter ( ) existiert. Hierbei wird im Modell die ursprüngliche Verbindung in „die Luft gesetzt“, d.h. es wird die Referenz auf das Ziel gelöscht. Eine Validierung des Modells würde nun einen Fehler melden, da das Verbindungsobjekt noch im Modell vorhanden ist, jedoch keinen Eingabeparameter ( ) mehr referenziert.

Um das Fehlverhalten zu korrigieren, wurde ein weiterer Aspekt eingeführt, der die Methode boolean `boolean canExecute()` in der Klasse `NodeConnectionCreateCommand` sowie in der Klasse `NodeConnectionReorientCommand` die beiden Methoden `boolean canReorientTarget()` und `boolean canReorientSource()` abfängt. In den Methoden wird entschieden, ob das Anlegen einer Verbindung bzw. Umhängen eines Verbindungsendes erlaubt ist oder nicht. Die Kombination aus der Verbindung sowie den neuen und alten Ein- ( ) und Ausgabeparametern ( ) wird durch die Methode `boolean canExecute(NodeConnection, OutputParameter, InputParameter)` einer weiteren Prüfung unterzogen. Exemplarisch sind die für die Methode `boolean canReorientTarget()` verantwortlichen Teile des Aspektes im Listing 7.6 zu sehen.

Listing 7.6: Der Aspekt zur Verhinderung des Ziehens von `NodeConnections` ( ) in bestimmten Fällen

```
9   public privileged aspect CreateOrReorientLinkConstraint {
39      /**
40       * intercepts the canReorientTarget() method in
41       * NodeConnectionReorientCommand
43       * @param cmd the NodeConnectionReorientCommand (this)
44       */
45      pointcut canExecuteForReorientTargetLink(NodeConnectionReorientCommand cmd) :
46          this(cmd)
47      &&  execution( boolean *..*.canReorientTarget())
48      ;

50      /**
51       * Around advice for the canReorientTarget() method in
52       * NodeConnectionReorientCommand. If the original algorithm succeeds an
53       * additional test is performed by
54       * {@link #canExecute(NodeConnection, OutputParameter, InputParameter)}
58       */
59      boolean around(NodeConnectionReorientCommand cmd):
60          canExecuteForReorientTargetLink(cmd){
61          boolean ret = proceed(cmd);
62          if (ret) {
63              NodeConnection conn = cmd.getLink();
64              OutputParameter out = conn.getFrom();
65              InputParameter in = cmd.getNewTarget();
66              ret = ret && canExecute(conn, out, in);
67          }
68          return ret;
69      }

102     /**
103      * Tests if the combination of out and in parameters is allowed for the
104      * current parent connection object. For new connections - conn should be
105      * null.
110      */
111     private boolean canExecute(NodeConnection conn, OutputParameter out,
112             InputParameter in) {
113         return in == null
114                 || (in.getIncomingLink() == conn
115                     && (out == null || out.getNode() != in.getNode()));
116     }
117 }
```

7.4 Modellvalidierung im Editor

Der graphische Editor soll das Modell auf diverse Modellierungsfehler hin prüfen. Hierfür eignet sich die im Abschnitt 3.7.4 beschriebene Erweiterung des „Diagram Gen Model“. Von der Firma Gentleware wurde dem Verfasser ein verändertes „Dynamic Template“ zur Realisierung der Anforderung R9 zur Verfügung gestellt. Es baut auf dem originalen Template auf und behebt ein Speicherloch im generierten Quellcode des originalen Templates. Da das Template nur eine Check-Datei unterstützt, wurden für den `dataflow`-Editor Änderungen am Template vorgenommen, um mehrere Dateien zu unterstützen. Durch weitere Änderungen wird das Modell nur dann geprüft, wenn es verändert wurde, was die Geschwindigkeit erhöht.

Die im Kriterium R14 geforderte Validierung in Echtzeit konnte mit den gegebenen Mitteln nur teilweise umgesetzt werden. Das sich im Editor in Bearbeitung befindliche Modell wird regelmäßig und damit „zeitnah“ überprüft. Hierbei wurde für den `dataflow`-Editor ein Zeitintervall von fünf Sekunden gewählt. Ein kürzeres Intervall behindert die Benutzerinteraktion, da das System mehr damit beschäftig ist, das Modell zu prüfen.

7.4.1 Regeln zur Prüfung des `dataflow`-Modells

Um die im Metamodell definierten und durch das EMF generierten Regeln auch mit dem oAW-Check basierten Ansatz nutzen zu können (s. a. Anforderung R10), wurde eine M2T-Transformation entwickelt, die die Regeln aus dem Metamodell in Check-Regeln umwandelt. Die generierten Dateien konnten dann in die GMF-Erweiterung eingetragen werden.

Da die in den Baumeditoren verwendete Modellvalidierung ebenfalls auf oAW-Check basiert, mussten die „alten“ Checks nur in das erweiterte „Graphical Def Model“ eingetragen werden, um die Anforderung R11 umzusetzen. Viele der in den „base“-Check-Dateien definierten Regeln waren als Ersatz für die in EMF definierten Regeln, jedoch nicht durchgängig und mit inkonsistenten Fehlermeldungen, angelegt worden. Da die EMF-Regeln nun auch als eigenständige Check-Dateien generiert werden, konnten die doppelten Regeln aus den alten Check-Dateien entfernt werden.

Im Rahmen der Anforderung R12 wurden bestehende Regeln verändert sowie weitere entwickelt. Hierzu zählen vor allem die Einführung von Prüfungen für die in der Tabelle 4.3 besprochenen Kardinalitäten. Es ist nicht möglich, alle während der Programmierung einer Anwendung mit der „base“ Softwareentwicklungsumgebung auftretenden Eventualitäten mit entsprechenden Regeln zu prüfen. Dies wird von Rotmans und Asselt ausführlich in [Rot01, S. 47] diskutiert. Das Ziel ist nur, häufige Fehler zu verhindern.

7.5 Test

Während der Entwicklung der „GenGMF“-Plugins und des graphischen Editors wurden Entwicklertests durchgeführt. Für ein aktuelles Problem wurde jeweils geprüft, ob sich die Software (weiterhin) korrekt verhält, nachdem Modifikationen an den Modellen oder den Quellen vorgenommen wurden.

In unregelmäßigen Abständen wurden zusätzlich die diversen Artefakte des Gesamtsystems getestet. So funktioniert der Editor z. B. grundsätzlich auch dann, wenn die verwendeten Aspekte nicht aktiv sind. Die Labels für die Ein- (▫) und Ausgabeparameter (▫) werden in diesem Fall nicht korrekt dargestellt. Der Fehler kann behoben werden, indem die kompilierten Artefakte der verschiedenen Plugins gelöscht und anschließend neu kompiliert werden. Das Auftreten dieses Fehlers hat mit der Benutzung von Eclipse als Runtime-Workbench zu tun, in der dieselben Projekte eingebunden sind, wie in der ursprünglichen Installation.

Von dem Aufbau einer Test-Umgebung z. B. auf der Basis des JUnit-Frameworks wurde abgesehen, da dies den Rahmen dieser Diplomarbeit überstiegen hätte.

Nicht durchgeführt wurden Tests für verschiedene Betriebssysteme bzw. Laufzeitumgebungen. Dies wurde in den Abgrenzungskriterien X2 und X4 ausgeschlossen.

7.6 Der graphische dataflow-Editor

Mit dem im Kapitel 7.2 erstellten Modell und den in den Kapiteln 7.3 und 7.4 beschriebenen manuellen Änderungen konnte der graphische Editor den Anforderungen entsprechend erstellt werden. Er ist, zusammen mit der Outline- und Properties View, in der Abbildung 7.13 dargestellt. Die graphische Darstellung des Modells enthält dieselben Modellelemente wie die Baumdarstellung in der Abbildung 5.1. Die Outline View auf der rechten Seite stellt dasselbe Element wie im graphischen Editor dar. Dies geschieht durch den in der Outline View integrierten Synchronisationsmodus ( ).

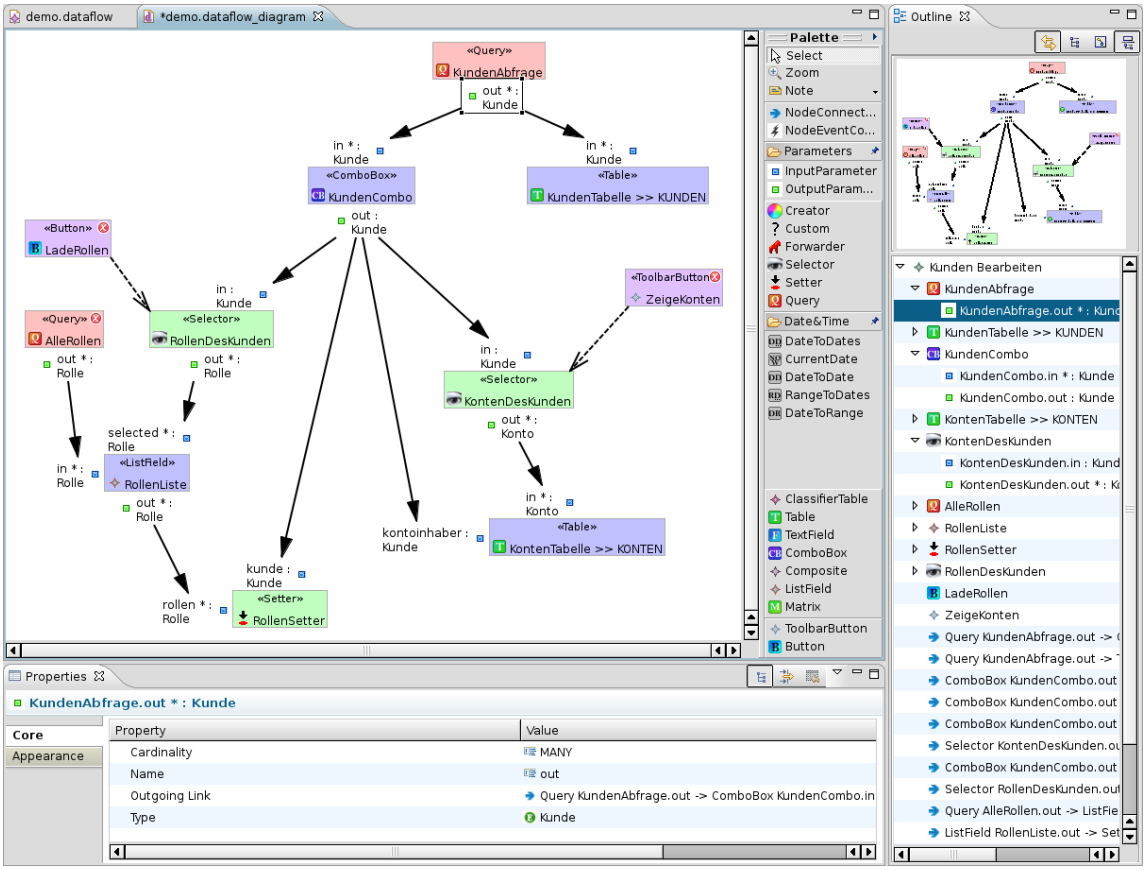


Abbildung 7.13: Der graphische dataflow-Editor

7.7 Sonstiges

Während der Umsetzung des Editors ist aufgefallen, dass im Metamodellpaket `dataflow` zwei Knotentypen existieren, die von der Generierung nicht verwendet werden. Dies betrifft die Typen `Filter` (✦) sowie `Save` (S). Der `Filter`-Knoten (✦) enthielt keine Attribute und es wurde hierfür keine Funktionalität im Quellcode generiert. Durch eine vor dem Beginn der Diplomarbeit vorgenommene Umstellung der Semantik der Datenhaltung in der „base“-Laufzeitumgebung wurde der `Save`-Knoten (S) nicht mehr benötigt, da inzwischen automatisch bei Änderungen gespeichert wird. Im Rahmen dieser Diplomarbeit wurden die beiden Knotentypen aus dem Metamodell entfernt, was den Aufwand für die Implementierung des `dataflow`-Editors reduzierte. Diese Änderung fällt unter das Abgrenzungskriterium X6.

KAPITEL 8

Ergebnisdiskussion und Ausblick

Dieses Kapitel dient der Diskussion der erreichten Ergebnisse sowie dem Ausblick auf zukünftige Entwicklungen.

8.1 Ergebnisdiskussion

Das Ziel der Diplomarbeit, die Entwicklung eines graphischen Editors zum Bearbeiten von `dataflow`-Modellen, wurde im Gegensatz zur ursprünglichen Planung nicht direkt mit dem Graphical Modelling Framework (GMF) erreicht. Dies lag an der hohen Anzahl von 20 für den Editor relevanten Knotentypen und des damit verbundenen sehr hohen Aufwandes für die Erstellung der GMF-Modelle. Stattdessen wurde zuerst „GenGMF“ entwickelt, um diese Aufgabe bewältigen zu können (Kapitel 6).

„GenGMF“ setzt sich aus einer auf die generische Entwicklung von GMF-Editoren spezialisierten Domain Specific Language für Eclipse und einer dazu passenden Modell-zu-Modell-Transformation zusammen. Es werden Templatestrukturen auf der Basis des GMF genutzt (Abschnitt 6.3). Deskriptoren verknüpfen die Templates mit den Knotentypen (Abschnitt 6.4). Durch die Transformation werden die Templatestrukturen für jeden Deskriptor kopiert und um zusätzliche Informationen aus dem Deskriptor angereichert (Abschnitt 6.5). In einer Skriptsprache kann die Nachbearbeitung der generierten Modelle erfolgen, um jedem Knoten ein individuelles Aussehen zu geben (Abschnitt 6.6). Auf diesem Weg können zwei der drei GMF-Modelle vollautomatisch generiert werden. Das dritte Modell ist im Vergleich zu den anderen Modellen relativ einfach und kann weiterhin mit dem in GMF enthaltenen Wizard generiert werden.

Dabei ist „GenGMF“ nicht auf „base“ oder `dataflow` beschränkt, vielmehr können Editoren für beliebige Metamodelle erstellt werden. Einen Vorteil durch die Nutzung von „GenGMF“ stellt sich jedoch erst ab einer gewissen Anzahl von für den Editor genutzten Metamodellelementen ein. Wo diese Grenze liegt, hängt neben der Einarbeitungszeit in das „GenGMF“ auch von der Komplexität der GMF-Modellstrukturen ab. So ist z. B. die Komplexität der GMF-Modelle bei der Verwendung von Compartment- oder Portstrukturen wesentlich höher als ohne.

Für den Entwicklungsprozess „base“ konnte der `dataflow`-Editor auf der Basis des „GenGMF“ mit weniger Aufwand entwickelt werden, als für eine Realisierung mit dem GMF notwendig gewesen wäre. Das folgende Beispiel soll dies verdeutlichen:

Beispiel: Reduktion der Komplexität mit „GenGMF“
Ein Maß für die Komplexität von Modellen ist die Anzahl der enthaltenen Knoten. Das „GenGMF“-Modell für den `dataflow`-Editor enthält 124 Knoten. Die aus dem „GenGMF“-Modell generierten GMF-Modelle „Graphical Def Model“ und „Mapping Def Model“ enthalten 275 bzw. 171 Knoten. In Summe hätten damit 446 Knoten manuell erstellt werden müssen, die so mit „GenGMF“ generiert wurden. Dies entspricht einer Ersparnis von 72%.
Damit hätte, um diese Strukturen direkt mit GMF-Modellen abzubilden, ein erheblich höherer Aufwand betrieben werden müssen, als dies mit den „GenGMF“-Modellen getan wurde.

In „base“ werden zur Zeit Editoren verwendet, die auf dem Eclipse Modelling Framework (EMF) aufbauen. EMF basierende Editoren, wie z. B. für auch aktuell das `dataflow`-Modell verwendet, sind ungeeignet für Modelle, welche das Konzept eines Graphen im mathematischen Sinne abbilden. Der entwickelte graphische Editor stellt die für einen Graphen wichtigen Bestandteile Knoten und Kanten auch als solche dar.

Vor allem für die Suche nach einer Lösung für die Anforderung R8 (Möglichkeit zur Bearbeitung des gesamten Modells), wurde viel Zeit verwendet (Kapitel 7.1). Letztendlich realisiert wurde eine ursprünglich durch Jacques Lescot entworfene „Outline-View“, die im Rahmen dieser Diplomarbeit um zusätzliche Funktionalitäten erweitert wurde (Abschnitt 7.1.4). Der Vorteil dieser Lösung ist, dass die im graphischen Editor dargestellten Informationen nicht zu detailiert sind und damit der „große Überblick“ über das gesamte Modell nicht eingeschränkt wird. Die Details lassen sich dennoch in der „Outline-View“ bearbeiten.

Der Verfasser hat während der Entwicklung des Editors den subjektiven Eindruck gewonnen, dass die Bearbeitung von `dataflow`-Modellen mit dem neuen graphischen Editor erleichert wird. Nur mit dem Einsatz in größeren und praxisnahen Projekten kann die Auswirkung auf die Effizienz im Entwicklungsprozess „base“ ermittelt werden.

8.2 Ausblick auf die weitere Entwicklung von „base“

Für das Projekt „base“ wurde ein graphischer Editor erstellt, mit dem die Entwicklungszeit einer „base“-Anwendung wesentlich verkürzt werden kann. Durch die Repräsentation von Elementen des `dataflow`-Modells durch adäquate Objekte in der Oberfläche sowie durch den Einsatz der „Outline View“ sollte sich die Benutzbarkeit des Modell-Editors insgesamt verbessern.

Während der Implementierung fiel auf, dass für die `dataflow`-Modelle einzelne Funktionalitäten fehlen. Eine Umsetzung ist jedoch das Ziel dieser Diplomarbeit gewesen (s. a. Kapitel 5.2.4 / Abgrenzungskriterien X5 und X6). Es wird aus diesem Grund vorgeschlagen, die folgenden Funktionalitäten im Metamodell, im Editor und im Generator hinzuzufügen:

Zusätzlicher Knotentyp `Union`

Ein `Union`-Knoten würde zum Kombinieren von Datenflüssen dienen. Einer Liste von Kunden könnte z. B. vor der Anzeige in einer `ComboBox` () ein zusätzlicher „[Bitte auswählen. . .]“-Eintrag hinzugefügt werden.

Zusätzlicher Knotentyp `DatesToRange`
: Aktuell besteht die Möglichkeit, aus einem Zeitraum (`Range`) mehrere Datumsangaben in einem festgelegtem Intervall zu generieren. Der umgekehrte Weg, aus einer Liste von Datumsangaben das minimale und maximale Datum zu ermitteln und als Zeitraum auszugeben, könnte von einem `DatesToRange`-Knoten ermöglicht werden.

Neuer Eintrag `WEEK` *für die Aufzählung* `DateField`
: Zusätzlich zu den Einträgen `DAY`, `MONTH`, `QUARTER` und `YEAR` sollte der Aufzählung `DateField` der Eintrag `WEEK` hinzugefügt werden, um Intervalle auch als Wochen angeben zu können.

Angabe einer Schrittweite für das Element `RangeToDates` *(RD)*
: Das Element `RangeToDates` (RD) unterstützt zur Zeit nur Intervalle, die das durch `DateField` angegebene Datumsfeld jeweils um 1 inkrementieren. Hier sollte es ermöglicht werden, durch die Angabe einer Schrittweite ein Intervall von z. B. 2 Monaten anzugeben.

Relative Rechenoperationen in `DateToRange` *(DR)*
: Für das `DateToRange`-Element (DR) sollen neben der Möglichkeit, feste Werte für die Modifikation des Datumsfeldes einzutragen, auch relative Werte eingegeben werden können.

8.3 Ausblick auf die Entwicklung des Frameworks „GenGMF“

Für das bereits unter [UrlSch08] veröffentlichte Tool „GenGMF“ hat der Verfasser dieser Arbeit vor, sich folgender Themen anzunehmen:

Vollständige Generierung des GMF „Mapping Def Model“
: Es soll evaluiert werden, inwiefern die in den Templates hinterlegten Mappingelemente ( ) generiert werden können. Ziel ist die Reduktion der Templates auf die Graphikdefinitionen ( ), was die Erstellung der „GenGMF“-Modelle vereinfachen soll.

Einbinden vorgefertigter Templatestrukturen
: Für jedes „GenGMF“-Modell müssen zur Zeit Templatestrukturen neu erstellt werden. Eine Wiederverwendung zwischen den Modellen für unterschiedliche Editoren ist nicht vorgesehen. Es sollen Möglichkeiten geprüft werden, eine Wiederverwendung zu ermöglichen bzw. vorgefertigte und durch „GenGMF“ zur Verfügung gestellte Templates in den Editor zu integrieren.

Abbilden der Variabilität
: Die aktuelle Implementierung verwendet Skripte, um Variabilität für den „GenGMF“-Editor flexibel handhaben zu können. Hier soll geprüft werden, ob die momentan verwendeten Skripte vereinfacht oder ersetzt werden können.

Update auf die neuesten stabilen Versionen verwendeter Software
: Für diese Diplomarbeit wurde „GenGMF“ mit den zu Begin aktuellen Versionen der Produkte GMF (2.0.2), EMF (2.3.2), Eclipse (3.3.3) und openArchitectureWare (4.2) erstellt. Zwischenzeitlich wurden jedoch für alle vier Produkte neue Versionen veröffentlicht. Hier ist zu prüfen, inwiefern „GenGMF“ und mit den neuen Versionen kompatibel ist. Sollten Probleme auftreten, ist geplant, Veränderungen an „GenGMF“ vorzunehmen, um die neuen Versionen ebenfalls zu unterstützen.

Integration in Eclipse
: Da „GenGMF“ sehr stark mit dem GMF verknüpft ist, wird langfristig die Integration von „GenGMF“ als eigenständiges Projekt in Eclipse oder in das GMF angestrebt.

KAPITEL 9

Zusammenfassung

Das Ziel der Diplomarbeit war die Entwicklung eines graphischen Editors zum Bearbeiten von `dataflow`-Modellen in der Softwareentwicklungsumgebung „base“. In „base“ werden zur Zeit Baumeditoren auf der Grundlage des Eclipse Modelling Framework (EMF) verwendet. EMF basierende Editoren, wie z. B. auch aktuell für das `dataflow`-Modell verwendet, sind ungeeignet für Modelle, welche das Konzept eines Graphen im mathematischen Sinn abbilden.

Als Framework für die Erstellung des Editors wurde das Graphical Modelling Framework (GMF) gewählt. Grundlage dieser Entscheidung war eine Literaturrecherche mit dem Ergebnis, dass das GMF im Vergleich zu TOPCASED das ausgereiftere Framework darstellt. Zusätzlich stehen in der betreuenden Firma „Gentleware AG“ technisch kompetente Ansprechpartner für Fragen zum Thema GMF zur Verfügung.

Es hat sich herausgestellt, dass das GMF grundsätzlich für die Erstellung von graphischen Editoren sehr gut geeignet ist, da die meisten gewünschten Funktionalitäten abgebildet werden können.

Das dem `dataflow`-Modell zugrunde liegende Metamodell enthält 20 verschiedene Knotentypen, die auch im graphischen Editor Verwendung finden sollen. Wenn das Metamodell jedoch viele für den Editor relevante Knotentypen enthält, steigt die Komplexität der GMF-Modelle so stark an, dass sie nur noch schwer handzuhaben sind.

Im Rahmen dieser Diplomarbeit wurde mit „GenGMF“ ein universelles Framework für Eclipse geschaffen, mit dem graphische Editoren auf der Basis des GMF entwickelt werden können. Es bietet vor allem dann Vorteile, wenn das dem graphischen Editor zugrunde liegende Metamodell viele verschiedene Klassen enthält, die als darzustellende Knoten oder Verbindungen in Frage kommen. Aus mit „GenGMF“ entwickelten Modellen werden die für die GMF-Editoren notwendigen Modelle „Graphical Def Model“ und „Mapping Def Model“ durch eine „GenGMF“-spezifische Modell-zu-Modell-Transformation generiert.

Die für den realisierten `dataflow`-Editor erstellten „GenGMF“-Modelle sind wesentlich kleiner als die daraus mit „GenGMF“ generierten GMF-Modelle. Es konnte eine Ersparnis von 72% erreicht werden.

Die im Pflichtkriterium R8 beschriebene Anforderung, dass das gesamte Modell bearbeitet werden kann, konnte aufgrund von Restriktionen im GMF nicht so umgesetzt werden, wie ursprünglich geplant (Darstellung von Inhalten in den Knoten; Wunschkriterium R16). Vier Varianten zur Darstellung von Details wurden evaluiert, von denen drei nicht weiter betrachtet wurden, da sie entweder zu umfangreich waren oder mit dem existierenden Metamodell nur mit umfangreichen Änderungen umsetzbar gewesen wären.

Realisiert wurde eine Möglichkeit zur Darstellung und Bearbeitung der Inhalte in der seperaten Ansicht „Outline View“ in Eclipse. Die Anforderung, dass das gesamte Modell bearbeitet werden kann (R8), ist damit erfüllt, auch wenn der eigentliche Editor nur Teile des Modells darstellt. Ein weiterer Vorteil ist, dass durch die Auslagerung von Inhalten in eine weitere Ansicht die Übersicht im Editor gewährleistet bleibt.

Sowohl „GenGMF“ als auch der `dataflow`-Editor steigern die Entwicklungseffizienz von graphischen Editoren bzw. von auf „base“ basierender Anwendungen. In „GenGMF“ wurde dies mit der getrennten Modellierung von Gemeinsamkeiten („commonality“) und Unterschieden („variability“) erreicht. Bei „base“ geschieht dies durch die Verwendung des graphischen `dataflow`-Editors.

Das „GenGMF“ Framework bildet die Basis für die Entwicklung von Editoren, z. B. im Rahmen einer weiteren wissenschaftlichen Arbeit. Zur Förderung dieses Ansatzes wurde das Framework bereits in [UrlSch08] unter der „Eclipse Public License“ als Open Source veröffentlicht. Hier sind die für die Benutzung notwendigen Eclipseplugins sowie das in Kapitel 6.7 vorgestellte Beispielprojekt verfügbar. Die mit dieser Arbeit angestoßene Entwicklung wird vom Verfasser mit Interesse verfolgt.

Auch die Softwareentwicklungsumgebung „base“ hat das Potential für weitere wissenschaftliche Arbeiten. So beinhalten die `domain`- und `module`-Modelle konzeptuell einen Graphen im mathematischen Sinn. Hier bietet sich die Verwendung von graphischen Editoren an, um die Effizienz in der Softwareentwicklung mit „base“ weiter zu steigern. Die Entwicklung der oben genannten Editoren könnte der Inhalt weiterer Arbeiten sein.

Literaturverzeichnis

[Ala05] Alanen, Marcus und Porres, Ivan: Subset and Union Properties in Modeling Languages. *TUCS Technical Report* (2005), (731), URL http://www.tucs.fi/publications/attachment.php?fname=TR731.pdf [10.08.2008]

[Atk01] Atkinson, Colin und Kühne, Thomas: Processes and Products in a Multi-Level Metamodeling Architecture. *International Journal of Software Engineering and Knowledge Engineering* (2001), Bd. 11(6):S. 761–783, URL http://www.mcs.vuw.ac.nz/~tk/publications/papers/processes-products.pdf [10.08.2008]

[b+m08a] b+m Informatik AG: Geschäftsbericht 2007 (2008)

[b+m08b] b+m Informatik GmbH Berlin: Referenzdokumentation des Metamodells „base“ (2008), liegt der Diplomarbeit in digitaler Form bei.

[Boz08] Bozec, Matthieu: Metadata Feature Label ? (2008), URL http://dev.eclipse.org/newslists/news.eclipse.modeling.gmf/msg11220.html [10.08.2008]

[Ecl] Eclipse Foundation Inc.: EMF/Recipes: Create an Eclipse Forms editor with widgets for your properties, URL http://wiki.eclipse.org/EMF/Recipes#Recipe:_Create_an_Eclipse_Forms_editor_with_widgets_for_your_properties [10.08.2008]

[Ecl06] Eclipse Foundation Inc.: EcoreUtil.Copier, JavaDoc (2006), URL http://help.eclipse.org/help33/topic/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/util/EcoreUtil.Copier.html [10.08.2008]

[Exe04] Exertier, Daniel; Langlois, Benoit und Roux, Xavier Le: PIM Definition and Description, in: *First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications*, URL http://modeldrivenarchitecture.esi.es/pdf/paper2-1.pdf [10.08.2008]

[Glo05] Glozic, Dejan (IBM Canada Ltd.): Eclipse Forms: Rich UI for the Rich Client. *Eclipse Corner* (2005), URL http://www.eclipse.org/articles/Article-Forms/article.html [10.08.2008]

[Hal02] Haller, Steffen Heiko Matthias: *Mappingverfahren zur Wissensorganisation*, Diplomarbeit, Freie Universität Berlin (2002), URL http://www.knowledgeboard.com/download/1672/pdf-filename-mapping_wissorg_haller.pdf [10.08.2008]

[Kri05] Kriha, Walter: Introduction to Generative Computing – Active vs. passive Generators (2005), URL http://www.kriha.de/krihaorg/docs/lectures/generativecomputing/genintro/foil15.html [10.08.2008]

[Les08] Lescot, Jacques; Zeidler, Urs; Shatalin, Alex; Hunter, Anthony; Schnepel, Enrico; Gronback, Richard; Tikhomirov, Artem und Merks, Ed: Bug 206778 – Provide a model oriented Outline View (2008), URL https://bugs.eclipse.org/bugs/show_bug.cgi?id=206778 [10.08.2008]

[Lud02] Ludewig, Jochen: Modelle im Software Engineering - eine Einführung und Kritik, in: Martin Glinz und Günther Müller-Luschnat (Herausgeber) *Modellierung 2002*, Bd. 12 von *LNI*, GI, S. 7–22, URL http://www.informatik.uni-stuttgart.de/iste/se/publications/download/Modelle.pdf [10.08.2008]

[ope07] openArchitectureWare Team: GMF2 Adapter (2007), URL http://www.eclipse.org/gmt/oaw/doc/4.2/html/contents/r51.html [10.08.2008]

[Rei07] Reitsma, Jaap; Gross, Johannes und Kuhn, Stefan: Diagram Partitioning (2007), URL http://wiki.eclipse.org/Diagram_Partitioning [10.08.2008]

[Rot01] Rotmans, Jan und van Asselt, Marjolein B.A.: Uncertainty in integrated assessment modelling: A labyrinthic path. *Integrated Assessment* (2001), Bd. 2(2):S. 43–55, URL http://journals.sfu.ca/int_assess/index.php/iaj/article/viewFile/211/162 [10.08.2008]

[Sch07] Schnepel, Enrico: Praktikumsbericht (2007)

[Sch08a] Schnepel, Enrico: Referenzdokumentation der Metamodelle des „GMF“-Projektes (2008), liegt der Diplomarbeit in digitaler Form bei.

[Sch08b] Schnepel, Enrico: Referenzdokumentation des Metamodells des „GenGMF“-Projektes (2008), liegt der Diplomarbeit in digitaler Form bei.

[Sei03] Seidewitz, Ed (InteliData Technologies Corporation): What do models mean? (2003), URL http://www.semanticcore.org/Docs/WhatDoModelMean.pdf [10.08.2008]

[Smi02] Smith, Jason McC. und Stotts, David: Elemental Design Patterns – A Link Between Architecture and Object Semantics, in: *Proceedings of OOPSLA 2002*, URL ftp://ftp.cs.unc.edu/pub/techreports/02-011.pdf [10.08.2008]

[Sta07] Stahl, Thomas; Völter, Markus; Efftinge, Sven und Haase, Arno: *Modellgetriebene Softwareentwicklung*, 2 Aufl. (2007)

[Ter07] Terfloth, Axel; Wendler, Svenja und Habiger, Marc: Domänenspezifische Editoren für die Entwicklung von Embedded Systems (2007), URL http://www.itemis.de/binary.ashx?id=3387 [10.08.2008]

[Tol06] Tolvanen, Juha-Pekka: Domänenspezifische Modellierung für vollständige Code-Generierung. *Java Spektrum* (2006), URL http://www.sigs.de/publications/js/2006/01/tolvanen_JS_01_06.pdf [10.08.2008]

[Ven06] Venkatesan, Madanagopal Doraiswamy: *Generation of Diagram editors, taking the Enterprise Application Integration Patterns as Case study*, Diplomarbeit, Technische Universität Hamburg-Harburg (2006), URL http://www.sts.tu-harburg.de/pw-and-m-theses/2006/dorai06.pdf [10.08.2008]

[Vö06] Völter, Markus: OpenArchitectureWare 4.1: Using AOP in templates (2006), URL http://www.eclipse.org/gmt/oaw/doc/4.1/35_templateAOP.pdf [10.08.2008]

[Zim07] Zimmermann, Prof. Dr. Frank: Vergleich von TOPCASED und GMF mit der Atos Origin Methode QSOS im Rahmen von modellgetriebener Softwareentwicklung. *Arbeitspapiere der Nordakademie* (2007), URL http://www.nordakademie.de/fileadmin/downloads/Arbeitspapiere/AP_2007_04.pdf [10.08.2008]

Weiterführende Internetquellen

[Urlb+m] b+m Informatik AG. Internetseite der „b+m Informatik AG“. URL http://www.bmiag.de.

[UrlBor] Borland. Internetseite des UML-Tools Together von Borland. URL http://www.borland.com/de/products/together/index.html.

[UrlEcla] Eclipse Foundation Inc. Internetseite des Projektes „AspectJ“. URL http://www.eclipse.org/aspectj/.

[UrlEclb] Eclipse Foundation Inc. Internetseite des Projektes „Eclipse Modelling Framework“. URL http://www.eclipse.org/modeling/emf/.

[UrlEclc] Eclipse Foundation Inc. Internetseite des Projektes „EcoreTools“. URL http://www.eclipse.org/modeling/emft/?project=ecoretools.

[UrlEcld] Eclipse Foundation Inc. Internetseite des Projektes „Graphical Editing Framework“. URL http://www.eclipse.org/gef/.

[UrlEcle] Eclipse Foundation Inc. Internetseite des Projektes „openArchitectureWare“. URL http://www.eclipse.org/gmt/oaw/.

[UrlEclf] Eclipse Foundation Inc. Internetseite des Projektes „UML2“. URL http://www.eclipse.org/uml2/.

[UrlGen] Gentleware AG. Internetseite der „Gentleware AG“. URL http://www.gentleware.com.

[UrlIBM] IBM. Internetseite der IBM Rational Rose Produktlinie. URL http://www-306.ibm.com/software/awdtools/developer/rose/index.html.

[UrlOmo] Internetseite des UML-Tools Omondo. URL http://www.omondo.de/.

[UrlSch07] Enrico Schnepel. Projektseite des Dokumentationsgenerators für Metamodelle („MetamodelDoc“), 2007. URL http://metamodeldoc.randomice.net/.

[UrlSch08] Enrico Schnepel. Projektseite des Generators für GMF-Modelle („GenGMF“), 2008. URL http://gengmf.randomice.net/.

[UrlZos04] Stephan Zoschke. jacoZoom, 2004. URL http://www.infozoom.de/de_jacoZoom.shtml.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

AOP	Aspektorientierte Programmierung
AST	Abstract Syntax Tree
DSL	Domain Specific Language
DSM	Domain Specific Modelling
EBNF	Extended Backus–Naur Form
EMF	Eclipse Modelling Framework
EMOF	Essential-MOF
FHTW	Fachhochschule für Technik und Wirtschaft
GEF	Graphical Editing Framework
GMF	Graphical Modelling Framework
M2M-Transformation	Modell-zu-Modell-Transformation
M2T-Transformation	Modell-zu-Text-Transformation
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MOF	Meta Object Facility
MVC	Model-View-Control
oAW	openArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
RCP	Rich Client Platform
SWT	Standard Widget Toolkit
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

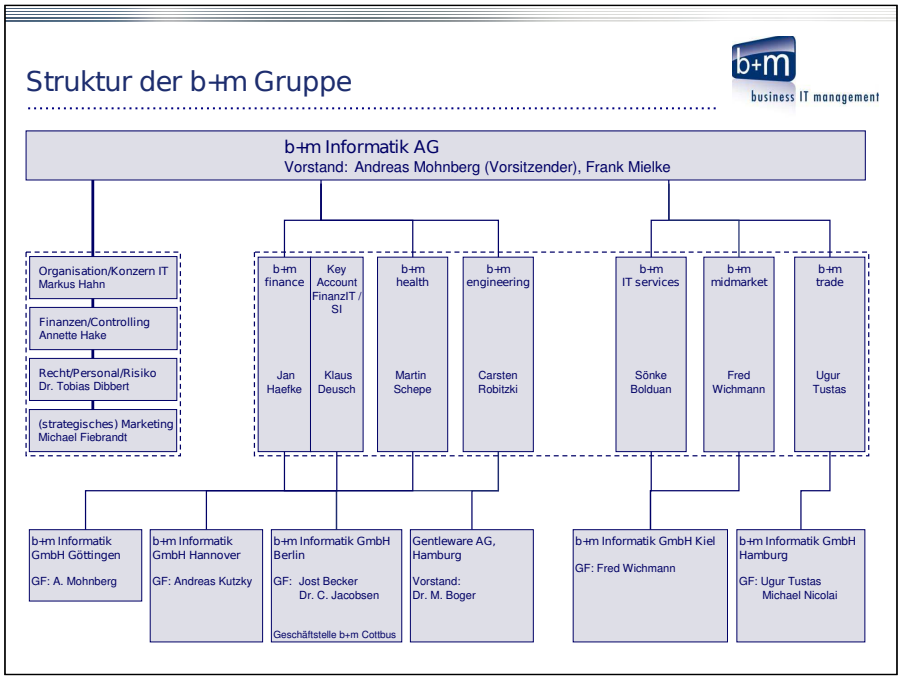
Index

ANHANG A

Anhang

A 1 Organisationsstruktur der „b+m Informatik AG“

Der Konzern „b+m Informatik AG“ in Melsdorf bei Kiel wurde als „becker&mohnberg Softwaregesellschaft mbH“ im Jahre 1994 gegründet und im Jahre 2000 in eine Aktiengesellschaft umgewandelt. Der Namensbestandteil „b+m“ entstammt den Anfangsbuchstaben der Nachnamen der Gründungsmitglieder Kai Becker und Andreas Mohnberg. Mit inzwischen 225 (Stand 31.12.2007) Mitarbeitern und mehreren auf ganz Deutschland verteilten Tochterfirmen werden diverse Projekte und Produkte – vor allem im Bankenumfeld – betreut und entwickelt. Die Berliner GmbH wurde 2000 als Tochterfirma gegründet und hat inzwischen 25 Mitarbeiter (Stand 31.12.2007). Ebenfalls im Jahre 2000 wurde Gentleware von Marko Boger als AG gegründet. Anfang 2007 beteiligte sich die „b+m Informatik AG“ mehrheitlich. Die „Gentleware AG“ hat zum 31.12.2007 elf Mitarbeiter beschäftigt. Die Organisationsstrukturen des Konzerns mit den einzelnen Tochterfirmen sind der Abbildung A.1 zu entnehmen. (vgl.: [b+m08a], [Urlb+m] und (vgl.: [UrlGen]))



(Quelle: b+m, Stand: Juni 2008)

Abbildung A.1: Organisationsstruktur der „b+m Informatik AG“

A 2 Farbschema für Abbildungen

Die in dieser Diplomarbeit dargestellten Modell-Abbildungen, also Abbildungen, in denen Modelle dargestellt werden, sind nach dem in der Abbildung A.2 gezeigten Farbschema erstellt worden.

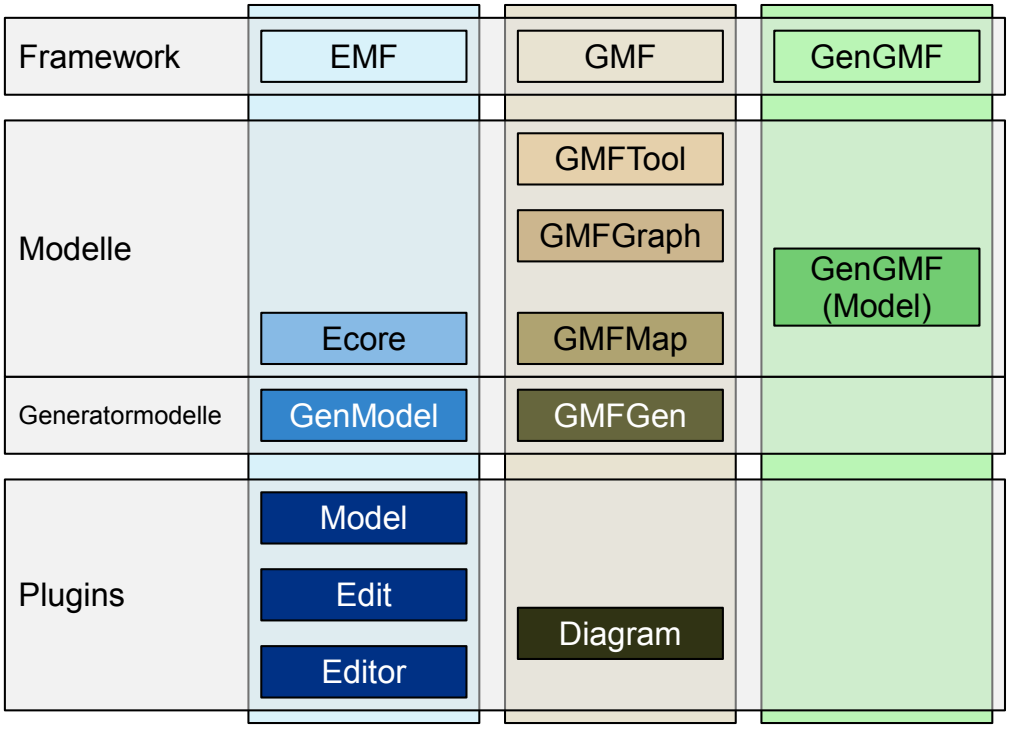


Abbildung A.2: Farbschema für die in dieser Arbeit dargestellten Abbildungen

Danksagung

Mein besonderer Dank gilt an dieser Stelle Herrn Prof. Heßling vom Fachbereich IV, Angewandte Informatik an der Fachhochschule für Technik und Wirtschaft Berlin sowie Herrn Stefan Hänsgen von der Firma „Gentleware AG“ für die Betreuung und Unterstützung bei der Anfertigung dieser Diplomarbeit.

Außerdem bedanke ich mich herzlich bei meinen Arbeitskollegen Dr. Thomas Valenthin und Jost Becker von der „b+m Informatik GmbH Berlin“ für die fachliche Betreuung zum Thema „base“ sowie für die zahlreichen Hinweise zum strukturellen Aufbau der Arbeit.

Bei Herrn Alexander Thomas möchte ich mich vor allem für die kompetenten Ratschläge zum Thema Farbgestaltung und Layout bedanken.

Für die grammatikalische und orthographische Durchsicht sowie für die moralische Unterstützung bedanke ich mich sehr herzlich bei meiner Familie und inbesondere bei meiner Lebensgefährtin Sabine Enigk sowie bei den vielen hier nicht genannten Personen für die mir gegebenen konstruktiven Vorschläge auf Fragen meinerseits.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 11.08.2008

Enrico Schnepel