Diplomarbeit
in
Computer Networking

Entwicklung eines domänenspezifischen UML Diagramms zur Benutzeroberflächenmodellierung

Referent:Prof. Dr. Bernhard Hollunder

Korreferent:Dipl. Ing. (BA) Dirk M. Sohn

Vorgelegt am:17.03.2008

Vorgelegt von:Stefan Kuhn

Danksagung

Mein Dank gilt meinen Betreuern Prof. Dr. Hollunder und Dirk Sohn, meinem Kollegen Christoph Löwer sowie Ed Merks, Bastian Kennel und meiner Familie.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Bisher wurde bei Orientation in Objects GmbH (OiO) PowerPoint Folien zur Graphical User Interface (GUI) Beschreibung eingesetzt, oder Kunden nutzen einen GUI-Builder, um ihre Anforderungen zu spezifizieren, welche nach programmiert wurden. Hieraus entstand die Motivation auch das GUI mittels Modellen zu beschreiben, welche zukunftsorientiert auf einer Standardsprache basieren sollen, um sie mit möglichst vielen Editoren bearbeiten zu können.

Die fachliche Abstraktion domänenspezifischer Diagramme ermöglicht Domänenexperten oft erst formale Modelle zu erstellen. Die Unified Modeling Language (UML) ermöglicht zwar die Verwendung domänenspezifischer Terminologie mittels Stereotypen, ihre dreizehn Diagramme sind jedoch ein Notationsstandard [Jec04] und nicht auf die Domäne anpassbar. Die UML ist „zunächst nicht einfach genug, um effektiv mit einer graphischen Syntax arbeiten zu können“ [Sta07] Es gilt einen domänenspezifischen graphischen Editor zur GUI Modellierung zu entwickeln, welcher UML konforme Modelle erstellt.

Der Editor soll weder ein technologiespezifischer GUI-Builder, noch ein UML Werkzeug sein. Dem Ansatz am nächsten liegt das Werkzeug Argoi, welches das Metamodell von UML erweitert, um UMLi [UMLi08] Modelle zu erstellen. Jedoch kann aufgrund des verwendeten Erweiterungsmechanismus das Modell nicht von anderen UML Werkzeugen geöffnet werden. Dem Autor ist kein vergleichbarer Ansatz bekannt.

Thematisch liegt der Fokus auf Visuellen Sprachen, der UML sowie Modellgetriebene Softwareentwicklung.

Die Arbeit ist in neun weitere Kapitel gegliedert. Zunächst werden die Grundlagen und die Anforderungen beschrieben. In der Sprachdefinition werden die Elemente des Problemraums identifiziert und formal festgehalten. In Notationselemente wird die entwickelte Notation der Sprachelemente vorgestellt. Nachdem die Werkzeugwahl begründet wird fol-

gen die technischen Kapitel. Das Kapitel Generativer Entwicklungsprozess von GMF stellt einen übertragbaren, werkzeugspezifischen Entwicklungsprozess vor. Danach wird das Design des Graphischen Editors (GE) anhand von Modellen und Meta-Modellen beschrieben. Der Schwerpunkt des folgenden Kapitels Implementierung liegt auf der aufgabenspezifischen Anpassung der Transformationsdefinition. Die Arbeit schließt mit einer Kritik der verwendeten Werkzeuge und der eigenen Entwicklung ab.

2 Grundlagen

2.1 Begrifflichkeiten

Domäne

Eine Domäne ist ein „begrenztes Interessen- oder Wissensgebiet“ [Sta07] .

Meta-Modell

Eine „formalisierte Beschreibung einer Domäne nennt man Metamodell“ [Sta07] , welches die abstrakte Syntax und die statische Semantik umfasst.

Abstrakte Syntax

Meta-Modellelemente und ihre Beziehungen bezeichnet man als abstrakte Syntax.

Konkrete Syntax

„Der Begriff (abstrakte Syntax) dient zur Unterscheidung von der konkreten Syntax, die beschreibt, wie die tatsächlichen Quelltexte bzw. Diagramme aussehen“ [Sta07] .

Statische Semantik

Die statische Semantik legt Wohlgeformtheits-Bedingungen fest.

Domänenspezifische Sprache

Eine Domänenspezifische Sprache (Domain Specific Language, DSL) ist nach [Sta07] eine Programmiersprache für eine Domäne. Sie besteht aus konkreter und abstrakter Syntax sowie statischer Semantik und einer klar definierten Semantik der Sprachelemente (dynamische Semantik) [Sta07] .

2.2 Model Driven Software Development (MDSD)

„Modellgetriebene Softwareentwicklung (...) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugt“ [Sta07] . Der zu erstellende Graphische Editor (GE) wird in ei-

ner MDSD Werkzeugkette genutzt, um diese Modelle zu erstellen. Er wird auch selbst mittels Modellen erzeugt.

„Das Ziel von MDSD ist (...) domänenspezifische Abstraktionen zu finden und diese in einer formalen Modellierung zugänglich zu machen“ [Sta07] .Hierdurch ist es möglich „auf einer höheren Abstraktionsebene zu programmieren“ [Sta07] , auch ohne die Details der Implementierung zu kennen. Dadurch entsteht ein hohes Automatisierungspotential, welches in der Mehrzahl der Fälle in eine Produktivitätssteigerung mündet.

In der MDSD werden mittels eines Generators Modelle in Modelle (Model to Model, M2M) oder Modelle in Text (Model to Text, M2T) überführt. Eine Transformation wird mittels einer Transformationsdefinition beschrieben, welche konkrete Modelle bei einer Transformation als Argumente berücksichtigt. Bei M2T Transformationen ähneln die Transformationsdefinitionen Schablonen, weshalb sie „Templates“ genannt werden.

2.3 Unified Modeling Language (UML)

„Die Unified Modeling Language (UML) dient zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Softwaresysteme unabhängig von deren Fach- und Realisierungsgebiet. Sie liefert die Notationselemente (...)“ [Jec04] , sowie die formale Meta-Modell-Beschreibung. Die UML wird von der Object Management Group (OMG) standardisiert und gilt als herstellerneutral. Seit der UML Version 2.0 basiert sie vollständig auf der Meta Object Facility (MOF). Die UML wird mittels der Infrastructure [OMG08a] , welche grundlegende Sprachelemente hält, sowie der Superstructure [OMG08b] , welche Diagrammnotation und Semantik enthält, spezifiziert.

Im weiteren Verlauf dieser Arbeit ist insbesondere der Stereotyp von Bedeutung. „Ein Stereotyp eröffnet die Möglichkeit zur Definition von Erweiterungen von existierenden Klassen des Metamodells. Sie ermögli-

chen somit die Verwendung von (...) domänenspezifischer Terminologie (...) durch „Anpassung“ der UML“ [Jec04] . Mittels Stereotypen können, in der UML einzigartig, auf Benutzer-Modellebene Metaklassen modelliert werden. Von der erweiterten Klasse wird nicht geerbt, sondern sie wird referenziert. Dieses Muster ist in der MDSD auch als Modellmarkierungen [Sta07] bekannt. Stereotypen werden in einem Profil definiert, welche die einzige standardisierte UML Erweiterungsmöglichkeit ist.

2.4 Eclipse Modeling Framework (EMF)

2.4.1 Einleitung

EMF stellt die technologische Kernkomponente der zur Realisierung dieser Diplomarbeit verwendeten Technologien dar.
Die wichtigsten Konzepte von EMF werden beschrieben:

- Ecore als Kernmodell bzw. Meta-Meta-Modell mit welchem Domänenmodelle (Meta Modelle bzw. eigene Sprachen) definierbar sind, sowie sein Verhältnis zu MOF.
- EMF zur Datenintegration, mit unterschiedlichen Datenrepräsentationen in Java und XML. Das Persistieren der Daten in Resources und die Verwendung mehrere Resources in einem ResourceSet.
- Die wesentlichen Charakteristika der Java Repräsentationsklassen wie das Observer Pattern und Reflection sowie die Fähigkeit EMFs diese zur Laufzeit dynamisch zu erzeugen.
- EMF.Edit mit `ItemProvidern` die die nötigen Informationen der Modellelemente für Editoren liefern. `Commands` und der `CommandStack` als Grundlage für undo und redo Operationen. Die Kapselung eines `CommandStack` und eines `ResourceSet` in eine `EditingDomain`.

2.4.2 Hinführung

Das Eclipse Modeling Framework (EMF) ist ein Framework zur Daten Integration und Modellierung. Freigegeben wurde EMF 2002 von IBM

[Mer07a] und wurde für Eclipse entwickelt, ist jedoch auch außerhalb einsetzbar. Benutzt wird es bereits von einer Vielzahl an Produkten z.B. von IBM und Eclipse-PlugIns wie UML2, Object Constraint Language (OCL) [OCL08] und GMF [Mer07a] . EMF verbindet XML, Java und UML [Bud03] . Vorausgreifend auf die Erklärung des Kernmodells wird bei der UML Unterstützung die von „essential MOF“ (eMOF) verwendeten Notationselemente des Klassendiagramms interpretiert, also einen Teil des Klassendiagramms [Bud03] . Anzusiedeln ist EMF zwischen Programmierern und Modellieren [Bud03]

EMF verarbeitet die Struktur eines Datenmodells einer unterstützten Technologie als Struktur-Ausprägung und kann sie in eine andere Ausprägung überführen. XML-Schemata und Java-Klassen sind mögliche Ausprägungen eines Datenmodells. EMF überführt die Ausprägungen mit Hilfe eines eigenen Kernmodells ineinander. Eclipse Projekte wie Connected Data Objects (CDO) [CDO08] oder Teneo [Ten08] ermöglichen es Datenbanken als weitere Ausprägung eines Datenmodells anzusehen, indem sie das EMF Kernmodell unterstützen.

2.4.3 Ecore (Meta) Model

Das EMF Kern (engl. core) Modell, auch Ecore genannt, ermöglicht die gemeinsame, abstrakte Repräsentation des Datenmodells der unterstützen Technologien. Diese werden über ein Ecore Instanzmodell miteinander verbunden [Bud03] . Ecore Modelle sind Meta-Modelle, beschreiben also andere Modelle. Das Ecore Modell ist selbst ein EMF Modell, definiert sich selbst und ist dadurch sein eigenes Meta-Modell. Folglich ist Ecore ein Meta-Meta Modell und lässt sich durch die Selbstdefinition beliebig rekursiv fortführen, z.B. Meta-Meta-Meta-Modell. Das Ecore Modell selbst besteht aus einer Teilmenge des UML Klassendiagramms, die verwendeten Elemente liegen jedoch eine Abstraktionsebene über UML. Das von der OMG für UML2 verwendete Meta-Meta-Modell ist MOF, dessen Verhältnis zu Ecore später Beachtung findet. Ecore be-

sitzt die gleiche Elementnotation, jedoch eine deutlich geringere Elementmenge wie UML. Als Beispiel dient die Aggregation, die in Ecore nicht vorhanden ist. Die Definition der UML Klassen als Metaklassen bietet jedoch den praktischen Vorteil, bestehende UML Werkzeuge zur Erstellung von Ecore Modellen zu nutzen [Bud03] . Standardmäßig werden Ecore Modelle in XML Metadata Interchange (XMI) serialisiert. XMI ist der Standard zur Serialisierung von Metadaten, welche Ecore Modelle sind [Bud03] .

Zusammenfassend lässt sich über Ecore folgendes sagen:

- Ecore und seine XMI Serialisierung sind das Zentrum von EMF
- Ein Ecore Modell kann aus einer der drei Quellen erzeugt werden: einem UML Modell, einem XML-Schema oder aus annotierten Java Interfaces.
- Die Java Implementierung, und optional andere Ausprägungen des Modells, können aus seinem Ecore Modell generiert werden.

2.4.4 Verhältnis von Ecore und MOF

Meta Object Facility (MOF) definiert eine Teilmenge aus UML für das Konzept der Klassen Modellierung in einem Objekt Speicher (Repository). Als solches ist MOF mit Ecore vergleichbar. Der Fokus von Ecore liegt auf der Werkzeug Integration statt auf dem Verwalten von Metadaten Repositories. Hierdurch vermeidet Ecore einige Komplexität von MOF [Bud03] . In der sich noch in der Entwicklung befindenden Spezifikation [MOF08] von MOF 2.0 ist eine ähnliche Teilmenge, „essential MOF“ (eMOF) [Mer07b] seit 2003 vorgesehen. Es gibt wenige, hauptsächlich Namensunterschiede zwischen Ecore und eMOF sowie Ecores Unterstützung von Generics [Mer07b] .EMF kann eMOF Serialisierungen lesen und schreiben [EMF05] .

2.4.5 Generator Modell

Das EMF Generator Modell (.genmodel) stellt technisch das Zentrum von EMF da. Es dekoriert ein Ecore Modell und in ihm werden Informationen gespeichert die das Ecore (auch Domänen Modell genannt) mit implementierungsspezifischen Details „verschmutzen“ würden [Mer07a] . Das Ziel der Code Generierung, das Präfix der generierten Klassen, der Name der erzeugten Fabrik sowie die verwendeten Entwurfsmuster einzelner Elemente werden im EMF-Generatormodell spezifiziert. Zusätzlich gibt die Möglichkeit das Generatormodell mit dem Domänenmodell zu synchronisieren. Aus dem Generator Modell wird der Quellcode erzeugt [Bud03] . Die Generierung lässt sich mit Hilfe von Java Emitter Templates (JET) [JET08] anpassen. JET ist eine Java Server Pages (JSP) ähnliche Template Sprache [Pop04] . Hervorzuheben ist, dass Anpassungen durch Methodenspezifische „Protected Regions“ [Sta07] erfolgen und angepasste Methoden mittels @generated NOT ausgezeichnet [Bud03] werden. EMF trennt also Generat und Implementierten Code nicht.

2.4.6 Generat

EMF erzeugt ein EPackage, Fabriken sowie Java Interfaces und Implementierungsklassen. Dies gilt als gutes Design und wird benötigt um Mehrfachvererbung zu realisieren. Es werden getter und setter generiert. Die Interfaces erweitern alle, direkt oder indirekt, EObject. Die wichtigsten Methoden hiervon sind:

- eClass() - gibt das Meta-Objekt des Objects zurück
- eContainer() und eResource() - gibt den das Objekt beinhaltenden Container bzw. die beinhaltende Resource zurück.
- eSet() und eGet() - geben Zugriff auf die performante Reflection API.

EObject selbst erweitert Notifier. Ein Notifier ist das beobachtbare Objekt des Observer Patterns [Gam95] und ein Adapter ist der Beobachter [Bud03]

2.4.7 Persistenz

Die Fähigkeit Modelle zu persistieren sowie andere persistierte Modelle zu referenzieren sind Schlüsselfähigkeiten von EMF [Bud03] . Es besteht die Möglichkeit selbst einen „Serialisierer“ zu schreiben und dennoch unabhängig von der Art der Serialisierung andere Modelle zu Referenzieren und selbst referenziert zu werden. Typischerweise wird das Wurzelelement des Modells einer `Resource` übergeben wodurch alle direkt oder indirekt vom Wurzelelement beinhaltenden Elemente in ihr gespeichert werden. Ein `ResourceSet` kapselt eine Menge an Resourcen auf die zusammen zugegriffen wird und ermöglicht Resourcen übergreifende Referenzen. Die passende Serialisierungskomponente für eine Resource wird aus einem Register (Registry) erfragt.

2.4.8 Dynamisches EMF

EMF kann die oben beschriebene Generierung der Java-Klassen aus einem Domänenmodell dynamisch zur Laufzeit im Speicher vollziehen [Bud03] . Dies ist für UML Stereotype unerlässlich. Hierfür werden implizit die Standardeinstellungen des Generator Modells verwendet und Verhalten (Methoden) kann nicht implementiert werden. Dynamisches EMF resultiert in einer geringeren Performance sowie in einer kompliziertere Verwendung der Meta-Elemente und Elemente.

2.4.9 EMF.Edit

EMF.Edit ist ein Framework um Editoren für EMF Modelle zu erstellen [Bud03] . Editoren benötigen im Wesentlichen die Beschriftung (Label), die Eigenschaften (Properties), die Kinder und das Symbol (Icon) eines Elements. EMF.Edit trifft bereits Annahmen hierfür, z.B. liegen Kind-Knoten eines Elements in seinen Containment Referenzen. Diese Annahmen sind, außer bei dynamisch zur Laufzeit erzeugten Editoren, anpassbar. Des Weiteren ist für Editoren das Editieren der Modelle, im Gegensatz zum reinen Schreiben der Modelle wichtig [EMF04] . Konkret ist

hiermit das Rücksetzten (undo) und Wiederherstellen (redo) von Änderungen gemeint. Änderungen am Modell werden mittels Commands durchgeführt, die in CompoundCommands geschachtelt werden können. Die für Editoren benötigten Informationen werden von ItemProvidern bereitgestellt, welche zudem generische Commands liefern. Für Instanzen eines Meta-Elements existiert mindestens ein ItemProvider. Commands werden auf dem CommandStack gespeichert, der die undo/redo Funktionalität ermöglicht. Der CommandStack sowie ein ResourceSet werden durch eine EditingDomain gekapselt.
Die bisher beschriebene Funktionalität von EMF.Edit ist nicht an Eclipse gebunden. EMF.Edit bietet zudem die Möglichkeit auf Basis der oben genannten Funktionalität einen Eclipse JFace basierten Editor zu generieren. Für EMF.Edit bietet die Transaktionskompontente EMF Transaction eine TransactionalEditingDomain mit welcher mehrere Threads bzw. Editoren auf einer EditingDomain arbeiten können [EMF08] .

2.5 Draw2D

Draw2D ist das in dieser Arbeit verwendete Framework für graphische Komponenten. Es wurde für GEF entwickelt, basiert auf SWT und ist ohne Eclipse oder GEF einsetzbar. Es ist auf Graphische Editoren mit bewegbaren Elementen ausgelegt [Sca05] und beseitigt Unzulänglichkeiten von SWT/JFace indem beliebige Figuren definierbar sind. Draw2D wird als Leichtgewicht System bezeichnet, da es in einem SWT-Canvas beheimatet ist um die Flexibilität zu erhöhen und weniger Systemresourcen zu verbrauchen [Moo04] . Das Lightweight System besteht aus einer speziellen Wurzelfigur, einem EventDispatcher, welcher SWT Events übersetzt, und einem UpdateManager, der die Figuren zeichnet [Moo04] .
Figuren (IFigure) in Draw2D besitzen ähnlich Eigenschaften wie native Fenster: ihre Größe ist anpassbar, sie sind verschiebbar, erhalten Fokus

und besitzen ein eigenes Koordinatensystem. Sie sind die zentralen Elemente und folgen dem Composite Entwurfsmuster [Gam95] . Jede Figur kann einen eigenen Layout Manager für die Anordnung der in ihr liegender Figuren definieren und ihnen Layout Constraints zuweisen. Draw2D bietet Bildausschnitte [Sca05] , die z.B. für Bildschirm-überschreitende Figuren notwendig sind.

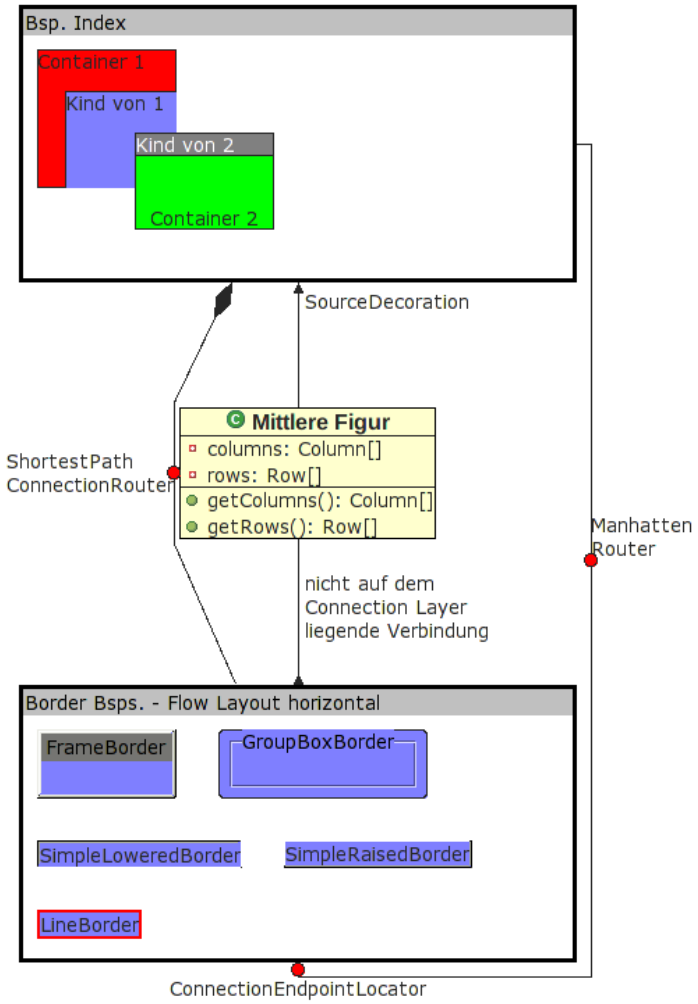

Abbildung 1 – Draw2D Beispiel

2.6 Graphical Editing Framework (GEF)

2.6.1 Einleitung

Zusammen mit der später beschriebenen GMF-Runtime liefert GEF die Laufzeitinfrastruktur für den erstellten Editors. Die GMF-Runtime ist als

Erweiterung und nicht als Abstraktion von GEF zu verstehen. Die in der Implementierung beschriebenen Anpassungen sind partiell und nicht zusammenhängend. Sie werden soweit erklärt, dass sie sich mittels der Beschreibungen von GEF und GMF-Runtime einordnen lassen. Dieser Abschnitt beschreibt unter anderem die zentralen Elemente `EditPart` und `EditPolicy` von GEF sowie ihre Kommunikation mittels `Requests` und `Commands`.

2.6.2 Hinführung

Das Modell-View-Controller [Gam95] Prinzip ist ein bekanntes Entwurfsmuster zur Rollenverteilung für Benutzerinteraktion. Das Graphical Editing Framework (GEF) wurde entwickelt, um die Controller Rolle bei Graphische Editoren zu übernehmen. Es leitet also die Interaktion zwischen dem Modell, welches die Bedeutung des Diagramms trägt, und dessen Darstellung. GEFs Aufgaben sind, die Element-Darstellung zu wählen und zu aktualisieren, dem Benutzer die Interaktion mit dem Modell zu erlauben und sich in die Arbeitsumgebung zu integrieren [Hud05] .GEF ist abhängig von Draw2D und der Eclipse Rich-Client-Platform (RCP). Als Modell in GEF kann jedes beobachtbare Java-Objekt verwendet werden, jedoch bietet sich eine Aufteilung in Geschäfts- und Ansichtsmodell an [Ani05] .

2.6.3 Überblick

Die Interaktion der GEF Elemente untereinander ist komplex. Dieser vereinfachte, typische Ablauf hilft einen ersten Überblick zu gewinnen:

1. Der Benutzer löst eine Aktion (`Action`) mittels eines Ereignisses (`Event`) aus.
2. Die Aktion schickt eine Anfrage (`Request`) an einen Controller (`EditPart`).
3. Der `EditPart` wertet die Anfrage mit Hilfe der zu ihm gehörenden `EditPolicys` aus und antwortet mit einem `Command`.

4. Der Command, welcher die Änderungen am Modell vornimmt, wird ausgeführt.
5. Die das Modell beobachtenden EditParts erkennen die Veränderung und aktualisieren die Ansicht.

Der Controller bzw. EditPart ist das zentrale Element in GEF. Bevor auf ihn eingegangen wird, werden die Komponenten betrachtet, welche seinen Einsatz erst ermöglichen.

2.6.4 Basiskomponenten

2.6.4.1 EditorPart

Eclipse Editoren sind dafür verantwortlich Eingaben entgegen zu nehmen, Viewer zu erzeugen und zu konfigurieren sowie die Eingabe zu verarbeiten und zu speichern [Moo04] . Eine Subklasse von EditPart stellt die Verbindung zwischen GEF und Eclipse her. Typischerweise hält der Editor eine EditDomain, eine ActionRegistry und einen SelectionSynchronizer [Sca05] .Die konfigurierten Viewer werden der EditDomain hinzugefügt.

2.6.4.2 EditDomain

Die Aufgaben der EditDomain und EMFs EditingDomain überschneiden sich, sie sind jedoch nicht zu verwechseln. Die EditDomain bündelt den Editor, mehrere Viewer wie z.B. die Toolbar und den CommandStack [Moo04] . Der CommandStack besitzt die gleichen Aufgaben wie der gleichnamige EMF CommandStack, sie sind jedoch ebenfalls nicht identisch.

2.6.4.3 GraphicalViewer

Der GraphicalViewer bietet eine nahtlose Integration in die Eclipse Workbench und ist überall verwendbar, wo ein SWT-Control bereitsteht. Er ist vom Editor unabhängig, und erzeugt ein LightweightSystem

auf dem Control. Folglich ist er für Ereignisbehandlung (Event Handling), unter anderem auch für „Drag & Drop“ verantwortlich, und leitet Events an die EditDomain weiter [Sca05] . Der GraphicalViewer bildet die Basis für EditParts [Sca05] und benötigt einen speziellen RootEditPart, eine EditPart Fabrik [Gam95] und Inhalt um zu funktionieren [Moo04] . Er besitzt ein EditPart-Register, um Modellelemente zu existierenden EditParts zuzuweisen, kennt die selektierten sowie fokussierten EditPart und das ausgewählte Werkzeug. Der GraphicalViewer besitzt eine spezielle IFigure, beispielsweise eine Viewport Figur.

2.6.4.4 Benutzerinteraktion

Grundsätzlich wird als Folge einer Benutzeraktion ein Request an einen EditPart geschickt, der, je nach Art der Interaktion, ihn mit einem Command beantwortet. Um einen Request zu erzeugen, kann man eine Taste drücken, eine Aktion (Action) auslösen oder mit der Werkzeugleiste (Palette) arbeiten. Das Modell wird nicht direkt von den EditParts manipuliert, sie liefern die Commands hierfür, die meist von den Werkzeugen ausgeführt werden.

2.6.4.5 Tools

Die Palette beinhaltet neben den komplexen Standard Werkzeugen [Sca05] unterschiedliche CreationTools [Moo04] , welche entweder Elemente (Knoten) oder Connections (Kanten) erzeugen. Beide werden mit einer Fabrik konfiguriert. Eine Besonderheit der CreationTools ist, dass sie für Feedback schon beim Gleiten über die Zeichenfläche fortlaufend Requests verschicken.

2.6.4.6 Requests

Requests sind die Kommunikationsobjekte in GEF [Moo04] . Sie werden verwendet, um Commands zu erhalten, und können für die Request-Ausführung relevante Informationen tragen. Beispielsweise hält ein Re-

quest zum Erstellen einer Verbindung zwischen zwei EditParts, der an die Senke gerichtet ist, den fertig konfigurierten Command zur Verbindungserstellung hinsichtlich der Quelle.

Die wichtigsten Requests sind:

- CreateRequest zum Erstellen von neuen Modellobjekten
- GroupRequest, um mehrere EditParts mit einem Request überspannen zu können. Beispielsweise löst das Löschen eines Elementes ein GroupRequest vom Typ delete aus.
- LocationRequest zum Aufspüren einer Position

2.6.4.7 EditParts

EditParts sind als Controller des MVC-Musters die zentralen Elemente in GEF und folgen zudem dem Composite Muster [Gam95] . Sie bestimmen wie Modellelemente auf visuelle Figuren abgebildet werden und wie sich Figuren in unterschiedlichen Situationen verhalten [Moo04] . Jeder Darstellung mit Verhalten ist ein EditPart zugeordnet und typischerweise gibt es für jedes Modellelement einen EditPart. Zusätzlich existieren z.B. für editierbare Labels und für nicht auf Assoziationsklassen beruhende Verbindungen (ansonsten trifft der Fall eines eigenen Modellelements zu) weitere EditParts. Die Aufgaben von EditParts sind nach [GEF08] :

- Erzeugen und erhalten einer Ansicht
- Erzeugen und erhalten von Kind EditParts und ggf. ConnectionEditParts
- Unterstützung der Modellbearbeitung

Vereinfacht lassen sich EditParts in drei Kategorien aufteilen: ConnectionEditParts für Kanten, GraphicalEditParts für Knoten und TreeEditParts für Bäume, z.B. für den Outline View.

Die EditPart-Erzeugung erfolgt durch Fabriken, welchen das Modellelement, für welches sie einen EditPart erzeugen soll, sowie ein Kontext bzw. der zukünftige Container EditPart, übergeben wird. Die EditPart-Erzeugung wird durch eine Modelländerungsbenachrichtigung angestoßen. Der folgende Ablauf betrachtet die Erzeugung eines GraphicalEditPart. Der zukünftige Container EditPart ruft refreshChildren(), welche sich über getModelChildren() eine Liste aller Kind-Elemente besorgt. Es wird geprüft, ob den Listenelementen EditParts zugewiesen sind. Ist das nicht der Fall, wird das unter Verwendung der Fabrik nachgeholt und der Kind-EditPart dem Container EditPart hinzugefügt.

2.6.4.8 EditPolicy

EditParts antworten auf Requests mittels EditPolicys, welche erst die Editierfunktionalität in die EditParts bringen [Moo04] . Sie erlauben die selektive Wiederbenutzung von Editierverhalten über mehrere EditParts [GEF08] . Erhält ein EditPart ein Request iteriert er über seine EditPolicys, welche den Request mit zusätzlichen Informationen anreichern oder einen Command zurückgeben. Im Normalfall fasst er die zurück gelieferten Commands zu einem CompoundCommand zusammen. Eine EditPolicy unterstützt einen Request, wenn sie einen Command zurückliefert, ignoriert ihn, wenn sie null zurückliefert oder unterbindet ihn, indem sie einen UnexecutableCommand zurückgibt. Beim Installieren in einen EditPart wird ihr eine in ihm einzigartige Rolle zugewiesen. Diese Rollen sind Strings und sind beim Löschen oder Austauschen von EditPolicys von Bedeutung [GEF08] .

Zusätzlich zu dem Zurückgeben von Commands übernehmen die EditPolicys auch das Benutzerfeedback. Beispielsweise erstellt eine GraphicalNodeEditPolicy beim Erstellen einer Verbindung eine „dummy“

Connection, die solange angezeigt wird, wie keine reale Verbindung existiert [Moo04] .

2.6.4.9 Command

Commands modifizieren das Modell und vereinfachten durch folgende Merkmale seine Bearbeitung [Moo04] :

- Undo und Redo
- Ausführungseinschränkungen (canExecute())
- Schachtelbarkeit

Sie gehören zwar nicht zum Modell, sind aber eng mit ihm verknüpft [GEF08] . Die bereits beschriebenen EMF-Commands übernehmen o.g. Aufgaben, jedoch ist die technische Basis unterschiedlich.

2.7 Graphical Modeling Framework (GMF)

2.7.1 Einleitung

Das Graphical Modeling Framework (GMF) entstand aus Frustration über die technischen Schwierigkeiten EMF mit GEF zu benutzen und dem Wunsch Graphische Editoren auf ähnliche Weise wie EMF-Baumeditoren zu erzeugen [Ani06] . Das aus einer Laufzeitinfrastruktur, welche GEF und EMF integriert, und einem generativen Teil bestehende GMF, wurde 2005 angekündigt und 2006 freigegeben [Sha06] . Die Laufzeit stellt zusätzliche Dienste wie Transaktionen, ein Notations-Meta-Modell, Variationspunkte für die Laufzeiterweiterung, usw. bereit. Der generative Teil umfasst eine Sprache, um Editoren zu beschreiben und passt damit in das Konzept der Software Factories [Sha06] . Zunächst wird die Laufzeitumgebung betrachtet, welche als Fortsetzung der GEF Grundlagen zu sehen ist, danach die generative Architektur.

2.7.2 Laufzeitinfrastruktur (Runtime)

Die Laufzeitinfrastruktur wurde von IBM entwickelt, welche diese für IBM Rational Produkte einsetzt [Ani06] . Ihre Eigenschaften sind:

- eine Service orientierte Struktur die den Editor erweiterbar für Drittanbieter macht
- Integration einiger EMF basierte Komponenten wie z.B. OCL und Transaction
- Definition eines erweiterbaren Notations-Meta-Modells
- GMFs Command Framework welches EMFs und GEFs Command(-stack) integriert
- die Extensible Type Registry

2.7.2.1 Notations-Meta-Modell

Das Notations-Meta-Modell speichert visuelle, nicht semantische, Informationen domänenunabhängig und ist somit unabhängig vom darunter liegendem Domänenmodell) [Ani06] . Mit seiner Hilfe werden die Positionen und Größen von Knoten, Verbindungen, etc. gespeichert. Es ist für eigene Bedürfnisse erweiterbar und das stabilste Meta-Modell in GMF. Die zentrale Klasse von ihm ist der View, er hält die element-Referenzen auf das Domänenelement und ist damit die einzige Verbindung zu dem semantischen Modell. Der View ist Teil des Modells und sollte nicht mit dem MVC Muster in Verbindung gebracht werden.

2.7.2.2 Dienste (Services)

Die GMF-Runtime wurde als Plattform für verschiedene Domäneneditoren entwickelt und benötigte deshalb robuste Erweiterungsmöglichkeiten, welche durch Eclipse Extension Point basierte GMF Service-Provider Infrastruktur [GMF08a] realisiert wurden. Diese Struktur wird über bekannte GEF Konzepte wie abstrakte „Basis“-EditParts und EditPolicys, welche an die Dienste delegieren, integriert [GMF08a] . Die unterschiedlichen Abläufe stellt [GMF08a] gegenüber. Die Services erlauben beliebige Arten der Überschreibung ohne domänenspezifische Abhängigkeiten von ihnen [GMF08a] .
Die wichtigsten Dienste sind:

- ViewService
- EditPartService
- EditPolicyService
- PaletteService

2.7.2.3 Extensible Type Registry nach [GMF08b]

Mittels der Extensible Type Registry wird ein applikationsspezifisches Klassifizierungssystem definiert. Dieses basiert auf und bietet eine Alternative zu den Meta-Klassen des Domänenmodells. Die Extensible Type Registry wird ausgiebig von GMF Services genutzt, beispielsweise beim Erfragen eines Delete Command für eine Klassifizierung oder des Icons für eine. Eine Klassifizierung ist ein Element Type (bzw. `IElementType`) und ihre konkrete Ausprägung entweder ein Meta-Modell Type oder ein Specialization Type. Der Meta-Modell Type aller Modellelemente mit gleicher `EClass` und gleichem Client-Context ist gleich. Das Editierverhalten des Meta-Modell- Type wird in seinem `EditHelper` bestimmt, welcher eine Command-Fabrik ist. Specialization Types erweitern einen Meta-Modell Type oder mehrere Specialization Types, jedoch auch indirekt nie mehr als einen Meta-Modell Type. `EditHelperAdvices`, welche die vom `EditHelper` zurückgegebenen `Commands` vor oder nach ihm dekorieren, können mehreren Element Types zugewiesen werden. Die `ElementTypeRegistry` wird von GMF verwaltet und ermöglicht das Auffinden von `ElementTypes` und `EditHelperAdvices` anhand verschiedener Kriterien. Die Angabe eines Client Context ermöglicht die Aufteilung der `ElementTypeRegistry` um Editoren auf Basis des gleichen Metamodells, wie z.B. des UML2 Metamodells, seiteneffektfrei einzusetzen.

2.7.3 Generative Architektur

Der generative Teil wird von Borland entwickelt. Er beseitigt bestehende Defizite der repetitiven Arbeit mit GEF („slow and painful“ [Ani06]) und

ermöglicht die schnelle Entwicklung von graphischen Editoren [Pla06] . Abbildung 2 zeigt dessen Aufbau, die Notation wurde in Anlehnung an das GMF Dashboard gewählt:

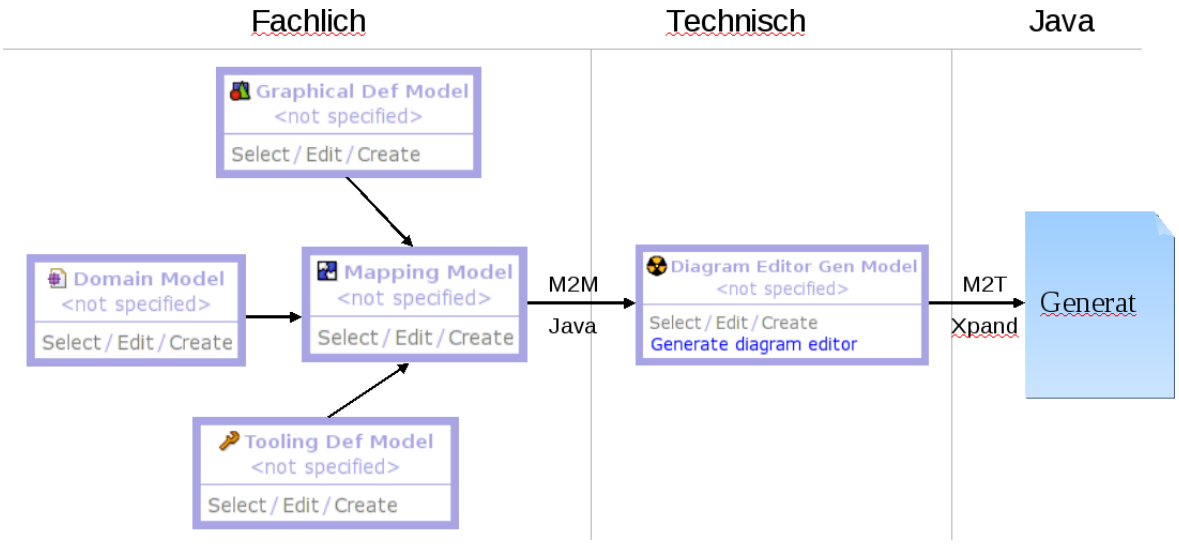

Abbildung 2 – Generative Architektur

Der Ablauf der Entwicklung eines Graphischen Editors lässt sich hieraus bereits erkennen. Ein EMF basiertes Domänenmodell z.B. UML2 wird erstellt und eingebunden. Danach wird die Notation im Graphical Definition Model sowie die Benutzerinteraktion im Tooling Definition Model festgelegt. Die drei oben genannten Modelle werden im Mapping- Modell vereint, welches per M2M Transformation in das plattformnahe Generator Modell überführt wird. Mittels der Xpand Template Sprache wird aus ihm Quellcode, welcher die GMF-Runtime nutzt, generiert.

2.7.3.1 Graphical Definition Model (gmfgraph)

Das Graphical Definition Model beschreibt die visuellen Aspekte eines Graphischen Editors anhand der Verknüpfung von Draw2D Figuren [Ani06] [Sha06] . Zudem ermöglicht es Quellcode unabhängig von anderen Modellen zu erzeugen und handgeschriebene Figuren über die Modellierung ihrer Interfaces einzubinden und ist wiederverwendbar [GMF08] .

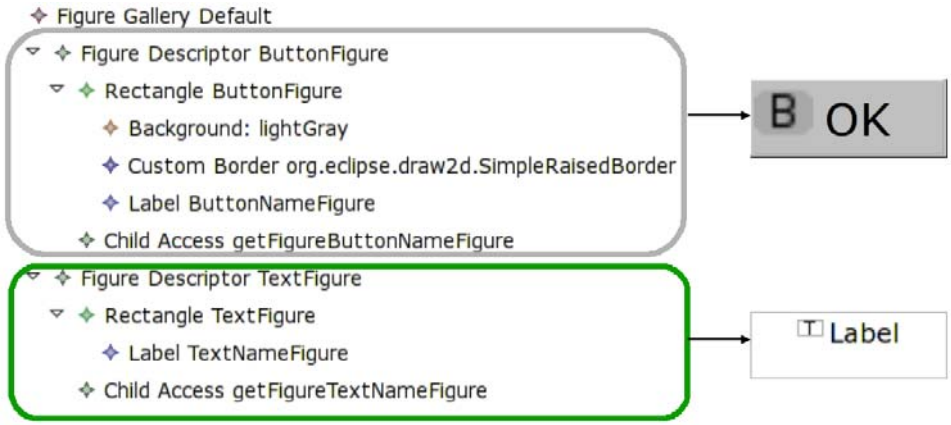

Abbildung 3 – Graphical Definition Model

2.7.3.2 Tooling Definition Model (gmftool)

Das Tooling Definition Model (Abbildung 4) wird benutzt um die Palette, Menüs, Toolbars, Popups usw. zu beschreiben [GMF08] .

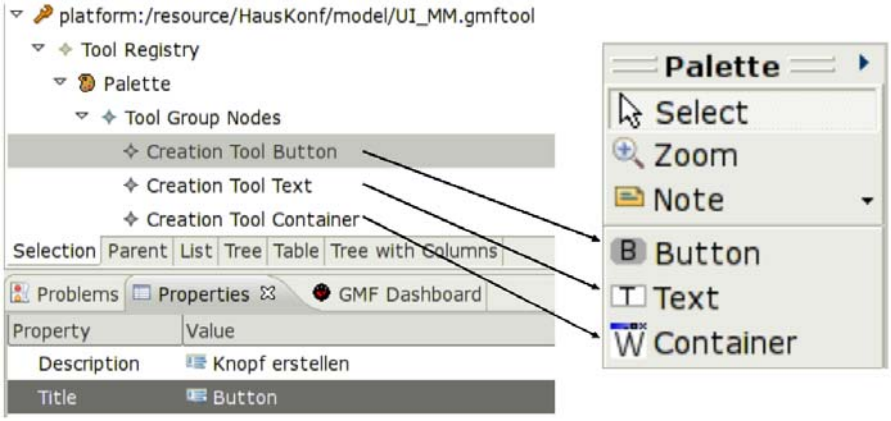

Abbildung 4 – Tooling Definition Model

2.7.3.3 Mapping Model (gmfmap)

Das Mapping Model verbindet die o.g. Modelle [Sha06] , indem es sie dekoriert. Zudem definiert es die Struktur des Editors, indem in ihm die Parts, die Kombination von Modell, EditPart und Figur, angeordnet und in Relation zueinander gesetzt werden. Beispielsweise wird ein Label mit einem Attribut des Domänenmodells verbunden und die möglichen Kind-Parts eines Parts definiert. In ihm ist es möglich Einschränkungen (Constraints) auf Parts zu definieren, Attribute initial zu belegen (Feature Initializers) und Validierungsregeln festzulegen.

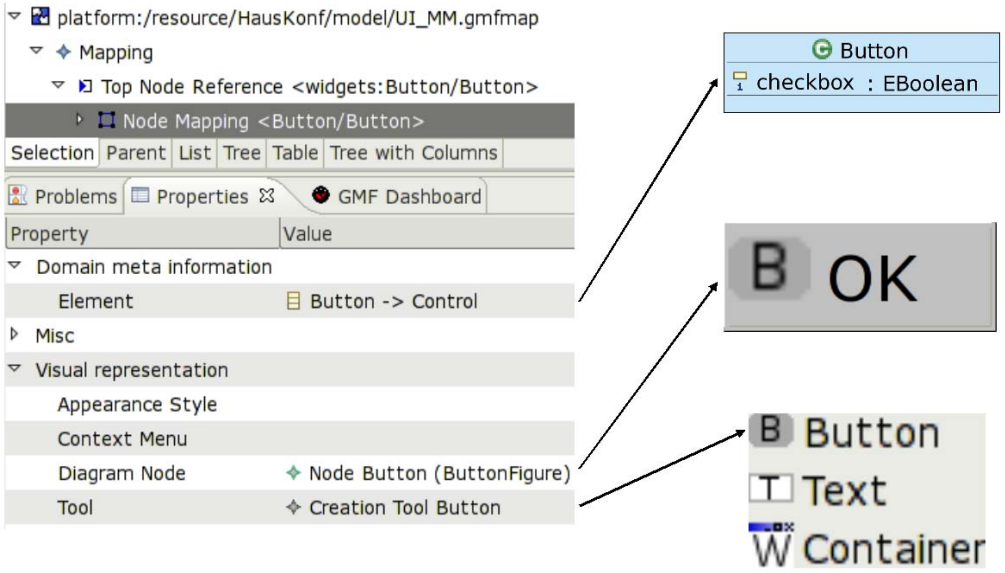

Abbildung 5 – Mapping Model

2.7.3.4 Generator Model (gmfgen)

Das Generator Modell ist das plattformnahe Modell der generativen Architektur. Es vereinfacht durch Elemente wie GenPlugin für das Eclipse Plugin und GenClass für EditParts und EditPolicies sowie deren Neuordnung die Template Programmierung. Da in ihm Implementierungsdetails konfiguriert werden können, wird es oft mit dem EMF Generator - Modell verglichen, jedoch dekoriert es kein Modell sondern wird durch eine Modell zu Modell Transformation(M2M) erzeugt und es gibt keine eins zu eins Abbildung der Elemente. Die M2M Transformation wird mit handgeschriebenen Java-Klassen erledigt, zusätzlich wird der Ergebnisquelltext der XPand Templates aus der Graphischen Definition in die zugehörigen Attribute `Class Body` geschrieben. Änderungen an gmfgen bleiben teilweise, nach nicht dokumentiertem System, erhalten. Die Anpassungsmöglichkeiten des Generator- Modells sind vielfältig, da in ihm alle Informationen der fachlichen Modelle vereint sind. Beispielsweise lässt sich ein zusätzliches Property Tab einfügen oder dynamische Templates aktiviert werden, was Anpassungen der Xpand Templates projektspezifisch erlaubt.

2.8 Xpand Template Sprache

Xpand ist Teil von openArchitectureWare (oAW) und wird von GMF seit der zweiten Version verwendet. Da GMF keine Anleitung zu seiner Version liefert, wird sie im folgendem anhand des oAW User Guides 4.2 [Eff07] erklärt, welcher sich jedoch in einigen Details von GMFs Implementierung unterscheidet.

Jede oAW Sprache kann eine gemeinsame Ausdruckssprache nutzen. Sie bietet Zugriff auf eingebaute Typen sowie zu denen registrierter Meta-Modelle und eine Art reflection layer.

Xtend ermöglicht es, neben der hier nicht behandelten M2M Transformation, Bibliotheken und nicht invasive Meta-Modell-Erweiterungen auf Basis der Ausdruckssprache oder Java Methoden zu definieren und diese für Xpand bereitzustellen.

Die Ausdruckssprache ist eine Mischung aus OCL und Java und besitzt im Wesentlichen die aus OCL bekannten Datentypen sowie Collections aus Java. Aufrufe an überladene Funktionen werden zur Laufzeit gebunden („multiple dispatch").

Drei Beispiele stellen die mengenbasierten Operationen vor:

- `{1,2,3}.select(i | i>2)` ergibt `{3}`
- `kollektion.typeSelect(meinTyp)` lieferte die Teilmenge der Instanzen des Typs `meinTyp`
- `kollektion.collect(ne | ne.name)` liefert eine Menge von Strings und ist äquivalent zu `kollektion.name`.

Xpand ist eine Template-Sprache, d.h. die Felder einer Vorlage werden mit modellabhängigen Werten gefüllt. Sie trennt Befehle mittels französischen Anführungszeichen '«' '»' vom Template. `DEFINE`-Blöcke ähneln Funktionsdefinitionen und `EXPAND`-Statements Funktionsaufrufen, welche das Ergebnis des `DEFINE`-Blocks an die aktuelle Stelle einfügen.

Xpand unterstützt Aspekte mittels `AROUND`-Blöcken, welche große Ähnlichkeit zu `DEFINE`-Blöcken besitzen. Innerhalb eines solchen Blocks ist

es möglich durch `targetDef.proceed()` den darunterliegenden DEFINE-Block auszuführen. Eine GMF Besonderheit ermöglicht „point cuts“ nur innerhalb eines Templates und schränkt Aspekte somit stark ein. Folgendes Beispiel zeigt die wichtigsten Charakteristika:

```
«IMPORT metamodel»
«EXTENSION template::MeineExtension»

«DEFINE main FOR Model»
«EXPAND javaClass FOREACH types»
«REM»'types' ist der Name der Containment-Referenz die von 'Model'
auf mehrere 'Type's zeigt. Von 'Type' erben 'Entity' & 'Datatype'«ENDREM»
«ENDDEFINE»

«DEFINE javaClass FOR Entity»
    «FILE name+".java"»«REM»FILE wird in GMF nicht verwendet«ENDREM»
        public class «name» {
            «FOREACH features AS f» «REM»FOREACH auf Collections«ENDREM»
                private «f.type.name» «f.name»;
                public «f.type.name» «f.name.getterName()»() {
                    return «f.name»;«EXPAND Kommentar("FOR ist optional")»
                }
            «ENDFOREACH»
        }
    «ENDFILE»
«ENDDEFINE»

«DEFINE Kommentar(String s) FOR Type»
//«s» «this.name» «REM»this. ist optional«ENDREM»
«ENDDEFINE»

«DEFINE javaClass FOR Datatype»
«REM»der Define Block wird Anhand
des Laufzeittyps bestimmt«ENDREM»
«ENDDEFINE»

«DEFINE javaClass FOR Type»
«REM»greift falls ueberladene javaClass DEFINEs nicht zutreffen.
this ist standardmaessig die Instanz des ersten Arguemtens«ENDREM»
«name.getterName()» «REM»Nutzt Extension«ENDREM»
«IF "Member Syntax" == (name.getterName())»
«ELSEIF "Funktions Syntax" == getterName(this.name)»
«ENDIF»
«ERROR " ungueltiges Argument"»
«REM»ERROR statt Modellvalidierung muss in GMF mangels
Anpassungsmoeglichkeiten der Validierung verwendet werden«ENDREM»
«ENDDEFINE»
```

Abbildung 6 - Xpand Beispiel

3 Anforderungen

Im Rahmen der Entwicklungsstrategie von OiO im Bereich MDSD ist ein prototypischer GE zu entwickeln. Es gilt den Domänenexperten bei der Anforderungsdefinition von GUIs zu unterstützen und ihm eine formale Beschreibung von ihnen zu ermöglichen. Es ist nicht zu erwarten, dass ein Domänenexperten eine korrekte UI Definition mit der dafür ungeeigneten UML Notation erstellen kann. Folglich muss sich der GE in Notation und Handhabung am Problemraum von GUIs orientieren. Da GUIs immer nur ein Teil der Software sind, gilt es ihre Beschreibung mit der weiteren, in UML erstellten Systembeschreibung zu integrieren. Um dies zu ermöglichen, muss der GE die UI Beschreibung möglichst verlustfrei, lesend und schreibend auf die abstrakte Syntax von UML abbilden. Da OiO keine Erfahrungen beim Entwickeln visueller Sprachen besitzt, ist zudem ein übertragbarer und reproduzierbarer Entwicklungsprozess für die gewählte Technologie zu definieren.

3.1 Funktional

3.1.1 Konformität zur abstrakten UML Syntax

Die GUI ist nur Teil einer Software, deshalb muss das Ausgabemodell mit anderen UML2 Modellen verknüpfbar und durch verbreitete UML Werkzeuge bearbeitbar sein. Dies impliziert die weitere Anforderung, dass die Ausgabe von gängigen Generatoren gelesen werden kann.

3.1.2 User Interface (UI)

Die Kernelemente der statischen Struktur einer GUI, wie Knöpfe und Eingabefelder ausgenommen ihrer Layout Informationen und Menüeinträgen, sind zu beschreiben. Datenbezogene Elemente wie Eingabefelder sind mit ihren Daten zu verknüpfen. Der Fokus liegt auf der praxistauglichen Umsetzung, nicht auf einer vollständigen Beschreibung, welche ggf. durch UML zu realisieren ist. Dynamische Aspekte (Verhalten)

können gut mittels verbreiteten Programmiersprachen, den „General Purpose Languages“ (GPL) [Sta07] , bzw. mittels UML Aktivitätsdiagrammen [Mül07] beschrieben werden und werden nicht behandelt. Die Ausnahme bildet ein in der Massendatenpflege häufig benötigtes Konzept: die „Master-Detail“ Beziehung [Sta05] . Auf Menüeinträge wird aufgrund Ihrer Nähe zu Aktionen bzw. zum Verhalten verzichtet.
Die Anforderungen werden mit denen von [Mül07] abgeglichen und relativiert. Dies geschieht, um den Problemraum für den GE auf einen Diagrammtyp zu reduzieren und einen praktikablen Ansatz zu finden.
„Alle verfügbaren Komponenten eines User Interfaces (...) müssen modellierbar sein, jedoch nur in ihrer abstrakten Beschreibung, da die Konzepte auf allen Plattformen anwendbar sein sollen“ [Mül07]
Dies wird relativiert zu: Die wesentlichen Komponenten eines UI müssen modellierbar sein, ihre Beschreibung soll sich konkret und eindeutig einem Typ zuordnen lassen.
„Bei der Modellierung von UI-Komponenten soll es möglich sein, diese für Informationsobjekte als Ganzes modellieren zu können. Es zeigte sich nämlich, dass in Datenpflegeanwendungen in den meisten Formularen alle Eigenschaften eines Informationsobjektes angezeigt werden sollen.“ [Mül07]
Die implizite Anforderung UI Komponenten mit Informationen zu verbinden gilt weiterhin. Ein Sprachelement für „Informationsobjekte als Ganzes“ ist nach Meinung des Autors nur zur reflexiven Datenabfrage an das Informationsobjekt sinnvoll, jedoch nicht für den Problemraum dieser Arbeit. Das beschriebene Sprachelement macht als Konvention eines Entwicklungsteams Sinn, falls die GUI mittels eines UML Werkzeugs definiert wird. Jedoch könnte beispielsweise ein Modellierungswerkzeug diese Verknüpfung automatisch erzeugen und bündeln und die Komplexität der Sprache würde nicht durch ein weiteres Sprachkonstrukt wachsen.

„Es müssen Abhängigkeitsbeziehungen zwischen den UI-Komponenten definiert werden können. Damit ist gemeint, dass die abhängigen UI-Komponenten die Auswahl ihrer dargestellten Informationsobjekte anpassen, sobald in einer anderen UI-Komponente ein bestimmtes Informationsobjekt ausgewählt wird. (…) Diese Problematik ist als Master-Detail-Beziehung bekannt und zeichnet sich durch eine 1:N-Beziehung (…) zwischen zwei Informationsobjektmengen aus.“ [Mül07]
Dieses Konzept ist, von den Abhängigkeiten der Typen untereinander abgesehen, umzusetzen.
Die weiteren Anforderungen von [Mül07] beziehen sich auf die Beschreibung der Daten zueinander sowie auf die Verhaltensbeschreibung. Diese sind für den zu entwickelnden Diagrammtyp nicht relevant.

3.2 Anforderungen zur UML-Abbildung

Die Sprachelemente müssen einfach, verständlich und praktikabel sein. Dies gilt in Hinblick auf den technischen Modellierer der sie in UML Notation bearbeiten muss, sowie für den Werkzeugentwickler, der andernfalls Zeit zum Verbergen von Komplexität vor dem Domänenexperten aufwenden muss. Die spätere generative Verwertung des Modells ist zu berücksichtigen. Die UML Abbildung musst standardkonform sein, ist also mit Profilen umzusetzen.

3.3 Anforderungen des Graphischen Editors

Der Editor muss direkt oder indirekt eine domänenspezifische Sicht auf ein UML Modell zur GUI Modellierung liefern. Die vorgeschrieben UML Notation ist folglich zu vermeiden und durch eine domänenspezifische zu ersetzen. Die definierte UML Abbildung ist zu berücksichtigen, nicht aus dem Problemraum stammende Sprachkonstrukte sind nicht anzuzeigen. Die Semantik der Sprachdefinition ist zu berücksichtigen. Beispielsweise darf ein Fenster nicht in einem Knopf modellierbar sein, obwohl das Meta-Modell der UML dies für stereotypisierte Klassen nicht

verbietet Konkrete Ausprägungen von Elementen sind über die Werkzeugleiste zu erstellen und ihre Eigenschaften müssen editierbar sein.

3.4 Nicht funktional

- Die Umsetzung ist mit freien Werkzeugen zu realisieren.
- Sie soll nach Möglichkeit in die von OiO eingesetzte Entwicklungsumgebung Eclipse integrierbar sein.
- Die Ergebnisse von [Mül07] zur „Front-End“ Beschreibung sind zu berücksichtigen.
- Die ausgegebenen UML Modelle sollen einfach für den Modellierer mit bestehenden UML Werkzeugen bearbeitbar sein. Zudem müssen sie sich gut für generative Entwicklung eignen. Praktikable Kompromisse sind zu finden.
- Das Aussehen (Notation) und die Handhabung des GE müssen für Domänenexperten verständlich und intuitive sein, er ist bei der korrekten Modellierung zu unterstützen.
- Der Editor muss zuverlässig „saubere“ Modelle erstellen, die Stabilität des Editors zur Laufzeit ist sekundär.
- Da die Modelle auch von UML Werkzeugen bearbeitet werden, sind Modellierungsfehler anzuzeigen. Der GE ist in diesem Hinblick auf eine gewisse Fehlertoleranz zu entwickeln.
- Die Vorgehensweise, um eine domänenspezifische Sicht auf ein UML Modell zu liefern, ist reproduzierbar OiO zur Verfügung zu stellen.
- Der GE muss leicht zu warten und zu erweitern sein.

3.5 Weiteres Vorgehen

Aus den Anforderungen ergeben sich folgende Schritte zur Erstellung des Graphischen Editors:

1. Anforderungsanalyse: Die Elemente des Problemraums, in dem der GE operiert, müssen identifiziert und ihre Bedeutung

beschrieben werden.

2. Die Sprachelemente sind auf ein UML Profil abzubilden.
3. Entwicklung einer Notation die die Elemente des Problemraums visuell beschreibt.
4. Entwicklung des GE der die Elemente des Problemraums (1.) mit der gewählten Notation (3.) anzeigt und bearbeitbar macht.
5. Entspricht das Modell des GE nicht UML, ist eine Transformation von und zu UML unter Berücksichtigung der Abbildungsvorschriften aus (2.) zu entwickeln.

Obgleich sich o.g. Schritte bei der iterativen Entwicklung vermischen, werden sie getrennt, da die Anforderungen späterer Schritte auf Ergebnissen der Vorherigen aufbauen, insbesondere im Hinblick auf die zu identifizierenden Elemente des Problemraums. Um Redundanz zu vermeiden, bzw. um Ergebnissen nicht vorweg zu nehmen und um selektives Lesen zu erleichtern, werden im weiteren Verlauf die o.g. Anforderungen verfeinert anhand der Ergebnisse der konzeptionelle Lösung vorhergehender Schritte. Die o.g. Abfolge ist speziell für die Lösung dieser Arbeit gültig, in der der vierte Schritt obsolet wird. Dies wird beim Erklären der Designentscheidungen in 8.1 deutlich. Andernfalls ist im ersten Schritt eine formale Beschreibung der abstrakten Syntax mittels eines Meta-Modells zu erstellen (anstatt 2), und die Definition der Abbildung (2) sowie die M2M Transformation auf ein UML Profil (5) können parallel zur Entwicklung des zweiten und dritten Schrittes erfolgen.

4 Sprachdefinition

4.1.1 Einleitung

Die Kernelemente des Problemraums lassen sich grob in sichtbare UI Elemente, Daten haltenden Elemente und Relationen zusammenfassen. Sie werden identifiziert, ihre Bedeutung wird geklärt und ihre Abbildung auf ein UML Profil in hinreichender Genauigkeit definiert.

4.2 Vorstellung vergleichbarer Konzepte von [Mül07] :

[Mül07] beginnt mit dem Erklären der Gruppen und bricht diese in einzelne Ausprägungen auf („Top-Down“). Dies steht im Gegensatz mit der o.g. „Bottom-Up“ Erklärung. Um ihrer Argumentationskette zu folgen, wird die Reihenfolge beim Erklären ihrer Konzepte beibehalten. Aufgrund der engen Bindung zwischen den Ausprägungen und den Typattributen werden diese, soweit notwendig, ebenfalls erklärt.
Sie beginnt bei Unterschieden zwischen einfachen und zusammengesetzten Komponenten, sowie zwischen Containern und Top-Level-Containern (S.30). Textfelder sowie Listen sind einfache Komponenten. Als Beispiel für zusammengesetzte Komponenten werden kombinierte Listen, als Kombination von Listen und Textfeldern, sowie Tabellen aufgeführt. Container folgen gleichfalls dem Composite Pattern, die Ausprägung Top-Level-Container stellt eine eigenständige Einheit der Anwendung da. Einfache und zusammengesetzte Komponenten werden (S.31) relativiert und (S.31, Abb. 19) mit dem Hinweis auf mangelnden fachlichen Mehrwert zu `UIDataComponent` zusammengefasst. Die Kombination von Attributen in `UIDataComponent` beschreibt das konkrete UI Element (S.31f). Die auf (S.33f) vorgenommene Zweiteilung dient „ein Informationsobjekt komplett“ anzuzeigen. Anhand zweier Beispiele von (S.32, Tabelle 1) wird die Beschreibung der konkreten UI Elemente erklärt:
Ein Textfeld wird beschrieben durch:

`editMode= INPUT, multiType=SINGLE`

Eine nicht editierbare Liste wird beschrieben durch:

`editMode= DISPLAY, multiType=MULTI`

Die Attribute des letzten Beispiels können zugleich auch einer aufklappbaren Liste (Picklist) zugeordnet werden und beschreiben somit das UI Element nicht eindeutig (S.32, Tab.1).

4.3 Konzeptionelle Lösung

Die konzeptionelle Lösung stellt zunächst die Kernelemente von statischen UIs vor und klärt, soweit notwendig, ihre Bedeutung. Sie werden denen von [Mül07] gegenübergestellt, und Abweichungen begründet. Danach werden die Sprachelemente vorgestellt, die Daten halten oder eine Datenabhängigkeit ausdrücken.

4.3.1 Identifikation und Einteilung statischer UI Elemente

Folgende Absätze zeigen die identifizierten statischen UI Elemente und ihre Gruppierung.

4.3.1.1 Primitive Elemente

- Beschriftungen (`Labels`), welche statische Zeichenketten anzeigen
- Textfelder (`Textfields`), welche dynamisch Zeichenketten anzeigen und ihre Eingabe ermöglichen
- Knöpfe (`Buttons`)
- Auswahlfelder (`Checkboxen`), welche entweder ausgewählt oder nicht ausgewählt sein können.

4.3.1.2 Komplexe Elemente

- Tabellen (`Table`)
- Bäume (`Trees`)
- Listen (`List`)
- Auswahllisten (`Picklist`), welche vergleichbar mit Listen sind, jedoch die einzeilig Auswahl eines Listenelements ermöglichen

4.3.1.3 Container Elemente

- Gruppen (`Groups`), welche allgemein eine Gruppierung von UI Elementen ermöglichen
- `Tabs`
- Fenster (`Views`), welche die Basis für andere UI Elemente darstellen und zudem als Basiselemente für Dialoge dienen.

4.3.1.4 Eigenschaften der Gruppen

- Primitive Elemente zeichnen sich durch eine oder keine Datenbindung aus.
- Komplexe Elemente operieren auf einer Menge von Datenelementen.
- Container Elemente folgen dem Composite Pattern [Gam95] , sind also beliebig rekursiv schachtelbar, und beinhalten beliebige statische UI Elemente.

4.3.1.5 Eigenschaften der Elemente

Zusätzlich wurden folgende Aspekte identifiziert, die an geeigneter Stelle den Kernelementen des UI verfügbar gemacht werden müssen:

- Eine Beschriftung, der GE hat diese anzuzeigen.
- Ein Kommentar zu Dokumentation
- Eine Auszeichnung wie sie editierbar bzw. ob sie verpflichtend sind. Der GE muss diesbezüglich visuelle Rückmeldung liefern.
- Eine Auszeichnung ob sie Zeichenketten mehrzeilig darstellen
- Eine Auszeichnung ob mengenbasierte Elemente Mehrfachauswahl erlauben
- Eine Auszeichnung ob Checkboxen eigenständig sind oder als Gruppe agieren und somit Radiobuttons sind.

4.3.2 Abweichungen

4.3.2.1 UI Container

Die Gruppe UI Container ist in beiden Arbeiten nahezu identisch. Sie weichen in zwei Fällen voneinander ab:

- Müller sieht (S.31) ein Layout des Containers vor, verzichtet aber auf seine Klärung und berücksichtigt keine Layout-Rahmenbedingungen in den `UIDataComponents`. Layout-Informationen sind für den GE nicht im fachlichen Modell notwendig (s. Anforderungen S.25) und es wird auf sie verzichtet.
- `UIFreeContainer` stellen für die fachliche Sicht, den Diagrammtyp, keinen Mehrwert da, auf sie wurde gleichfalls verzichtet.

4.3.2.2 Datengebundene Komponenten

`UIDataComponent` wird in primitive Komponenten, sowie in komplexe, mengenbasierte Komponenten aufgeteilt. Der Editiermodus wird als Aspekt gesehen und nicht zur Identifikation konkreter UI Elemente genutzt. Die geteilten Komponenten besitzen eigene, dem multiType ähnliche Aufzählungstypen (Enumerations), welche jedoch direkt und intuitiv ein konkretes UI Element beschreiben. Zusammensetzen von komplexen Komponenten wird im Hinblick auf die gewünschte Abstraktion sowie auf die Machbarkeit im gegebenen zeitlichen Rahmen nicht berücksichtigt.

4.3.3 Daten und Elemente zur Datenbindung

Um datengebundene Elemente mit einem Datum zu verbinden, werden drei Sprachelemente eingeführt:

- Der `UIDataProvider`, welcher Daten zum Lesen und Schreiben anbietet, und sie einer Entität ähnlich gruppiert. Er dient als Schnittstelle zu den Geschäftsdaten bzw. zu dem Controller der

Software. Bewusst wurde keine Implementierung vorgeschrieben, die ihn auf ein Value Object, Data Transfer Object oder Proxy einschränken.

- Die `Property`, welche ein einzelnes Datum im UIDataProvider darstellt. Sie muss in letzter Instanz auf eine Zeichenkette abbildbar sein.
- Die `UIDataRelation`, welche ein datengebundenes Element mit einem Datum bzw. einer `Property` verbindet.

Für Elementmengen wird das Sprachelement `UIDataSet` eingeführt. Es ist das einzige Element, welches lediglich einen Auszeichnungscharakter besitzt und sich nicht für die generative Entwicklung eignet. Es signalisiert unscharf zwei mengenorientierte Konzepte auf von ihm beinhaltende Elemente:

- Sie sind als Menge zu betrachten, was für mengenorientierte UI Elemente von Bedeutung ist.
- Sie stehen in Relation zu einander. Diese Relation ist zum Beschreiben der UI nicht von weiterem Interesse. Es stellt eine Abstraktion zum Auflösen von Daten(-sätzen) anhand eines Typs oder Schlüssels da, wie für ein Master-Detail-Verhältnis benötigt wird.

Zur statischen Verhaltensmodellierung wird das Sprachelement `UIMasterDetail` eingeführt. Mit ihm lassen sich Details zu einem Element einer Elementmenge in einem oder mehreren anderen datengebundenen UI Elementen anzeigen. Beispielsweise zeigt die Master-Liste alle Bücher einer Bibliothek, bei der Auswahl eines Buch wird in der Detail-Liste alle Exemplare sowie ihr gegenwärtiger Status angezeigt.

4.3.4 Klassifizierung

Eine weitere Klassifizierung der Gruppen bietet sich an. Die datengebundenen Elemente, also primitive und komplexe Elemente, bilden die Klasse `UIDataControl`. Weiterhin gehören alle Elemente der Benutzer-

oberfläche `UIControl` an. Die Elemente der Datenhaltung `UIDataSet` und `UIDataProvider` spezialisieren `UIAbstractData`.

4.4 UML Profilabbildung

Es gilt die identifizierten Sprachelemente in ein UML Profil zu überführen. Die Abbildung der Elemente des Problemraums wird beschrieben, diese Abbildungsbeschreibung ist für den GE bindend.

4.4.1 Klassen und Aufzählungen

Die identifizierten statischen UI Elemente sowie die Elemente der Datenhaltung lassen sich geradlinig mittels Stereotype, Attributen und Aufzählungen („Enumerations“) auf ein UML Profil abbilden. Dies ist damit zu begründen, dass sich die genannten Elemente auf Stereotype, die die Metaklasse Class erweitern, abbilden lassen. Das folgende Diagramm ähnelt einem vergleichbaren Meta-Modell für domänenspezifische Sprachen ohne Assoziationen.

Abbildung 7 zeigt alle Elemente des UML Profils, die die Metaklasse Class erweitern, sowie die dazugehörigen Aufzählungen.

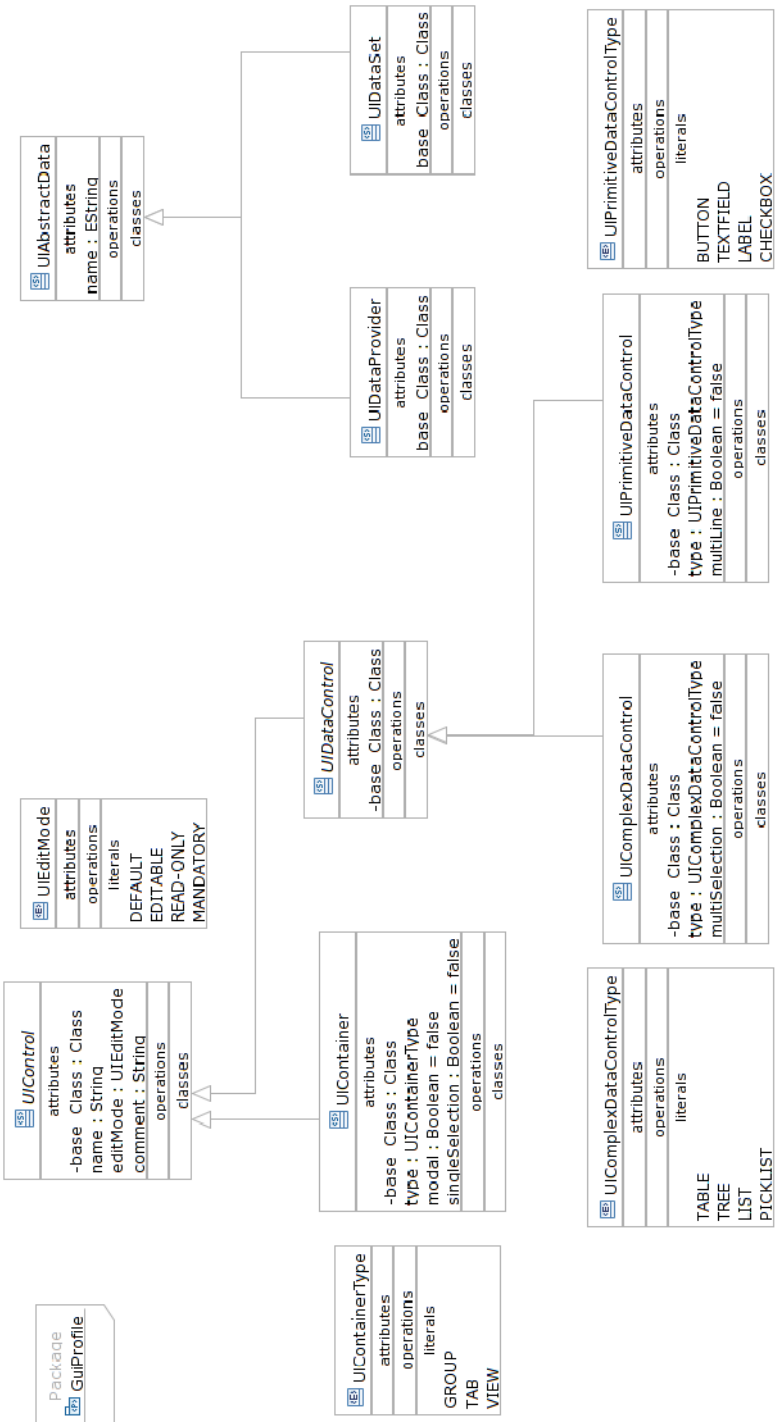

Abbildung 7 - Class erweiternde Stereotypen

Hinweis: Die Abbildung zeigt bereits die Umsetzung in Eclipse UML2. Der erste Eintrag in attributes der Stereotypen zeigt den abgeleiteten /metaclass Rollennamen des gehaltenen Assoziationsendes der Extension [OMG08b] . Die abgeleitete Form setzt sich zusammen aus base_ sowie den Namen der erweiterten Metaklasse, in diesem Fall also base_Class, im nächsten base_Association. Die erweiterte Metaklasse ist in der Abbildung nicht zu sehen. Erweiterungen („Extensions“) sind nicht mit Vererbungen zu verwechseln.

Allgemein gültig besitzt jede Klasse, die eine konkrete Ausprägung von UI Elementen repräsentiert, eine Eigenschaft type von einem zu der Klasse passendem Aufzählungstyp. Alternativ könnte beispielsweise UIContainer abstrakt sein und Group, Tab und View von ihm erben.

UIControl ist die abstrakte Basisklasse aller sichtbaren Elemente. Jede Spezialisierung besitzt einen Namen zusätzlich zum UML Klassennamen. Der name dient meist als Label bzw. zur Benamung innerhalb des UI und kann mehrfach vorkommen. Die Zweiteilung erfolgt, da der Klassenname in UML eindeutig sein soll. Weiterhin besitzt UIControl eine Zeichenkette comment zur Dokumentation. Auf den ersten Blick sicher verwunderlich ist, dass der editMode auf UIControl ausgedehnt und nicht auf UIDataControl beschränkt wurde. editMode ermöglicht die Editierbarkeit von Elementen zu setzen und wurde auf UIControl ausgedehnt, um ein bestehendes Sprachmittel auf Gruppen zu erweitern. MANDATORY besagt, dass es sich um ein Pflichtfeld handelt, DEFAULT übernimmt die Editiereinstellungen des Containers, READ-ONLY und EDITABLE sind selbsterklärend. Es ist herauszustellen, dass der editMode eines Containers nicht den seiner beinhaltenden Elemente erzwingt. Sie können nach Belieben vom DEFAULT abweichen, also gesondert ausgezeichnet werden.

UIContainer besitzt singleSelection, um Checkboxen einer Gruppe zu „Radiobuttons“ umzuwandeln. Mittels editMode=MANDATORY lässt sich diese Auswahl verpflichtend gestalten, z.B. zur Auswahl von männlich oder weiblich. modal=true verpflichtet einen Dialog bis zum Ende zu durchlaufen.

Im mengenbasierte Typ UIComplexDataControl kann mittels multiSelection=true ein Tupel ausgewählt werden.

UIPrimitiveDataControl erstreckt sich bei multiLine=true über mehrere Zeilen.

Die datenhaltenden Stereotypen UIDataProvider und UIDataSet zeichnen, vom UI Namen abgesehen, die erweiterte Klasse lediglich aus. Das ist ausreichend, um im Falle von UIDataProvider normale UML „Properties“ zum Beschreiben eines Datums zu verwenden.

4.4.2 Assoziationen und Containments

Für die Beinhaltung anderer Komponenten, für ihre Datenbindung und für Master Detail Verhältnisse wird eine Erweiterung der Metaklasse-Assoziation gewählt. Bevor die Wahl begründet und die Abbildungsvorschriften definiert werden können, ist die eindeutige Abbildung von Assoziationen zu klären. Jede Abbildung setzt eine binäre, gerichtete Assoziation voraus. Dies stellt sicher, dass die Assoziation, welche auf Meta-Ebene eine Assoziationsklasse ist, zwei „Property Ends“ besitzt, von denen nur eins navigierbar ist. Die navigierbare Property, wird von dem navigierendem Type besessen. Dies ist in UML als Punkt Notation bekannt [OMG08b] . Diese Wahl erfolgte zur Kompatibilität, da bisher, als informelle Konvention, navigierende Typen die Property beinhalteten [OMG08b] .

Abbildung 8 zeigt die Elemente des UML Profils, die eine Assoziation erweitern.

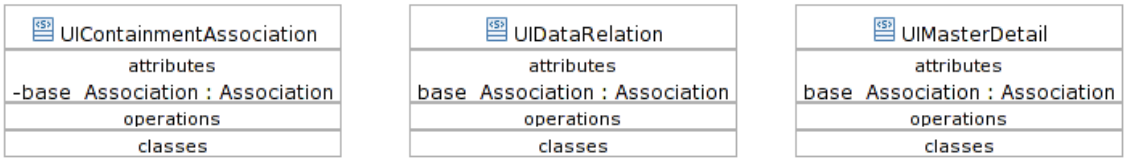

Abbildung 8 - Assoziationserweiternde UML Elemente

Die Abbildung von `UIMasterDetail` ist nun einfach: Der Master, ein mengenbasierter `UIComplexDataControl`, zeigt auf ein `UIDataControl` Detail, welches er nicht selbst ist. Es ist eine einfache Assoziation, deren Enden eine Multiplizität von eins besitzen und mit „master“ bzw. „detail“ benannt werden sollten.

Bevor die bereits vorausgegriffene und für den Leser wahrscheinlich selbstverständliche Abbildung von `UIContainmentAssociation` auf eine Komposition bzw. Aggregation erklärt wird, ist eine Alternative zu nennen und zu bewerten. „Innerhalb einer Klasse können beliebig viele weitere Klassen definiert werden. (...) Die grafische Darstellungsform eingebetteter Klassen entspricht einer gerichteten Assoziation von der umgebenden (...) zur enthaltenden Klasse (...), die mit der Multiplizität 1 versehen ist“ [Jec04] . Diese Variante ist beispielsweise für `UIContainer` vollkommen ausreichend, die zugehörigen UML Notation für den fachlichen Modellierer intuitiver, für die Generative Entwicklung ist sie mindestens gleich geeignet und für den Entwickler des GE wesentlich einfacher. Sie wurde nicht gewählt, da ihre Darstellung in bestehenden UML Werkzeugen oftmals problematisch ist. Dies zeigt, dass die Abbildung auf ein UML Profil nicht unwesentlich von bestehenden UML Werkzeugen beeinflusst wird. Die `UIContainmentAssociation` zeichnet aus, dass ein Element in einem anderen liegt. Zum Halten eines beliebigen Elements durch einen `UIContainer` wurde eine Komposition aufgrund von Existenzabhängigkeit, mit beidseitiger Multiplizität von 1 gewählt. Die Rollennamen sollten „contains“ sowie „containedBy“ sein. Beinhaltet ein `UIDataSet` einen `UIDataProvider` ist eine Aggregation zu wählen,

mit beidseitiger Multiplizität von 0..*. Die Rollen sind entsprechend angepasst und heißen „holds“ und „holdBy“. Anfang und Ende von UIContainmentAssociation dürfen nicht identisch sein, die Assoziation zeigt vom Container auf sein beinhaltendes Element.

Die UIDataRelation soll ein datengebundenes Element mit einem Datum vom Typ Property verbinden. Zum besseren Einordnen sei darauf hingewiesen, dass die Property haltende Meta-Komposition der Metaklasse Class einen zu „Owned Attribute“ abgeleiteten Namen trägt. Mit Property ist in diesem Fall also ein Attribut einer Klasse gemeint. Die Problematik wird anhand zweier Beispiele dargestellt, die beschreiben, weshalb für die Lösung eine Konvention gewählt werden musste. Im gewählten Anwendungsfall gilt es mittels eines Textfelds den Namen eines Kunden darzustellen.

Beispiel 1 – Trugschlüsse

Im ersten Beispiel wurde Textfeld und Kunde als Klassen modelliert. Kunde besitzt das Attribut name vom Typ String. Der name wurde im Anschluss aus der Klasse gezogen, dies entspricht der alternativen Darstellungsform eines Attributs in Form einer Assoziation [Jec04] . Im Anschluss wurde eine Assoziation von Textfeld zu String modelliert.

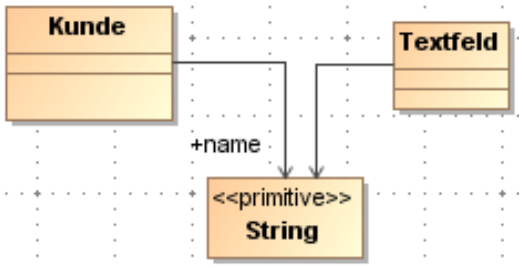

Abbildung 9 – Beispiel 1 in Magic Draw

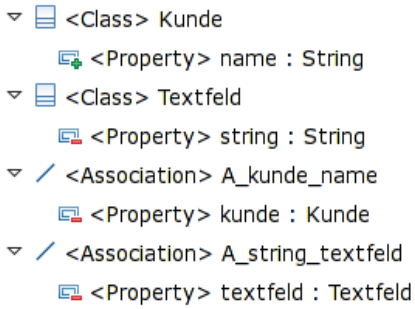

Abbildung 10 – Beispiel 1 als Eclipse UML2 Export

Abbildung 10 zeigt Beispiel 1 als Eclipse UML2 Export, von MagicDraw als „EMF UML2“ bezeichnet.

Der Blick auf die Klassen des UML Exports offenbart, dass jede Klasse für sich ein `String` Attribut besitzt. Es besagt jedoch nicht, dass sie identischen Inhalt besitzen müssen. Die UML besitzt ein einschränkendes Sprachmittel zwischen Assoziationen mittels eines Schlüsselwortes [Jec04] , im Hinblick auf den zweiten Trugschluss bleibt dem Autor unklar, ob dies für o.g. Fall überhaupt gültig ist. Die Betrachtung der Assoziationen deutet bereits hierauf hin: Sie besitzen beide nur ein Ende (`Property`). Die Assoziationsenden müssen, wie am Anfang dieses Abschnittes erklärt, zwar nicht von der Assoziation gehalten werden, ein Blick auf die „verbunden“ Elementen zeigt jedoch, dass sie ebenfalls keine Assoziationsenden halten. Die Assoziation ist somit unär, und in UML „sogar ausdrücklich verboten“ [Jec04] . Die Superstructure Definition besagt, dass die Assoziationsenden mit endType auf eine Instanz des Meta-Elements Type zeigen müssen, welches `Property` jedoch nicht spezialisiert. Die Eclipse UML2 Implementierung ist also Standard konform. Die Darstellung eines Attributs in Form einer Assoziation ist also keine Instanz des Meta-Elements `Association`. Hieraus ergibt sich, dass o.g. Verhältnis nicht über die in UML definierte Semantik, direkt über eine `Association` beschreibbar ist. Direkt meint in diesem Kontext ohne die Spezialisierungen von `Association` zu betrachten.

Beispiel 2 – Mögliche UML Darstellung

Abbildung 11 zeigt die mögliche Darstellung als Assoziationsklasse

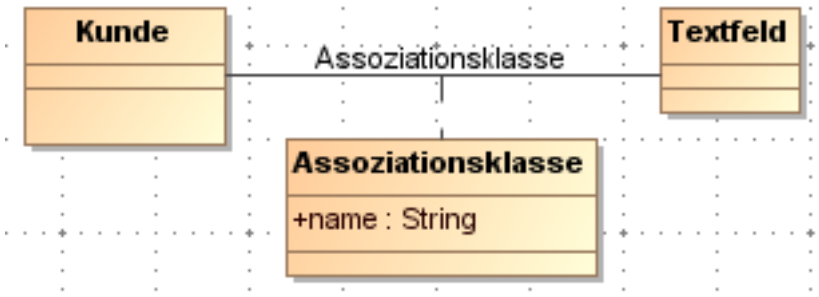

Abbildung 11 – Mögliche UML Darstellung

Im zweiten Beispiel wird das Problem einer eindeutigen Referenz mittels einer Assoziationsklasse gelöst. Das Verhältnis zwischen Textfeld und Kunde kann zwar mittels einer Assoziationsklasse beschrieben werden, dennoch dient eine Assoziationsklasse dazu „Eigenschaften näher zu beschreiben, die keinem der zur Assoziation beitragenden Classifier (...) sinnvoll zugeordnet werden können.“ [Jec04] . Ignorieren wir diese Empfehlung zunächst und bezeichnen die Anzahl der Textfelder als 'n'. Im o.g. Beispiel ist n=1. Stellen nun mehrere UI Elemente den Namen eines Kunden da, ist 'n' also eine beliebige natürliche Zahl, entstehen 'n'-Assoziationsklasse, also auch 'n' * Name. Ist man dennoch dazu geneigt diese, auch in jedem UML Werkzeug sehr komplexe Form zu wählen, ist n einschließlich 0 zu betrachten. Berücksichtigt man, dass UML keine unären Assoziationen erlaubt, fehlt dem Kunde im Falle von n=0 der Name. Weiterhin bleibt die Referenz auf Name ungelöst und es wäre eine Konvention einzuführen, dass die Assoziationsklasse nur ein Attribut besitzen darf und dieses anzuzeigen ist.

Lösung durch Konvention

Als Alternative bietet sich der Rollenname bzw. name des vom Textfeld gehaltenen Assoziationsendes an. „Rollen (...) erläutern die Semantik einer an der Assoziation teilnehmenden Klasse und die Form der Teilnahme näher. Rollen werden dabei in Form einer frei wählbaren Zei-

chenkette angetragen“ [Jec04] . Der Rollenname wird nun dazu verwendet das anzuzeigende Attribut zu bestimmen, indem der Rollenname gleich dem anzuzeigenden Attributsnamen ist. Es wurde bewusst vermieden o.g. Verwendungsweise in Form von „provides data from [Attributname]“ Rechnung zu tragen, da sich hierdurch die Fehlerwahrscheinlichkeit beim Modellieren in UML Editoren erhöhen würde und ein Generator zusätzlich den Rollenname „parsen“ müsste. Es wird erwartet, dass sich die Lösung vergleichsweise gut validieren und zur generativen Entwicklung nutzen lässt. Weiterhin stellen frei verfügbare UML Werkzeuge wie „Topcased UML“ oder „UML2 Tools“ den Rollennamen direkt im Klassendiagramm da. Abbildung 12 zeigt die Lösung.

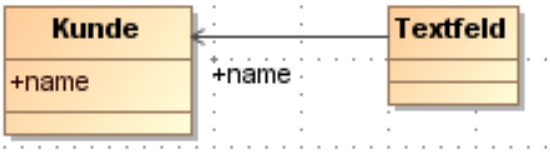

Abbildung 12 – Lösung durch Konvention

Eine Abhängigkeit eines Daten gebundenes UI Elements zu einem Attribut ist durch eine gerichtete, mit dem Stereotyp `UIDataRelation` ausgezeichnete Assoziation von dem UI Element zu dem Attribut haltendem `UIDataProvider` umzusetzen. Der Name des navigierbaren Assoziationsendes muss mit dem Name des anzuzeigenden Attributs identisch sein. Das gegenüberliegende Assoziationsende sollte mit „isPresentedBy“ benannt werden. Die Multiplizität beider Enden ist 0..*, kann aber im Falle des navigierbaren Assoziationsendes auch 0..1 sein. Dies würde sie auf einen nicht Mengen basierten Zugriff einschränken. Semantisch ist eine `UIDataRelation` mit in der referenzierten Klasse nicht existentes anzuzeigendes Attribut bedeutungslos und folglich zu löschen, sobald dieses Attribut gelöscht wird.

5 Notationselemente

Im Folgenden werden die entwickelten Notationselemente aufgeführt. Auf eine Erklärung der Notationselemente für die sichtbaren UI Elemente wird aus naheliegenden Gründen verzichtet. Auf Elemente die weiße oder durchsichtige Bereiche besitzen wird extra hingewiesen.

5.1.1 Elemente des UI

5.1.2 Primitive Elemente

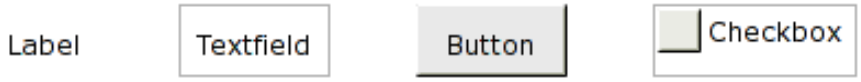

Abbildung 13 – Notationselemente der primitiven Elemente

Das Textfeld zeichnet sich durch weißen Hintergrund aus, der der Checkbox ist durchsichtig.

5.1.3 Komplexe Elemente

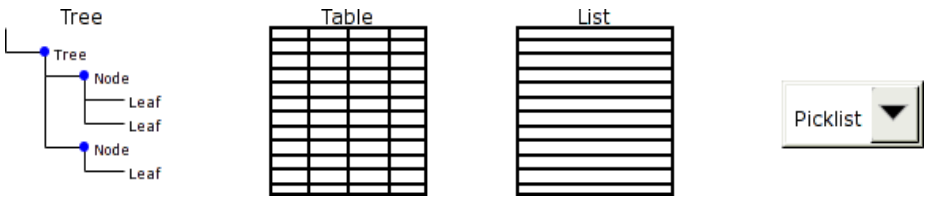

Abbildung 14 – Notationselemente der komplexen Elemente

Die `Picklist` besitzt einen weißen Hintergrund, der der restlichen Notationselemente ist durchsichtig.

5.1.4 Container Elemente

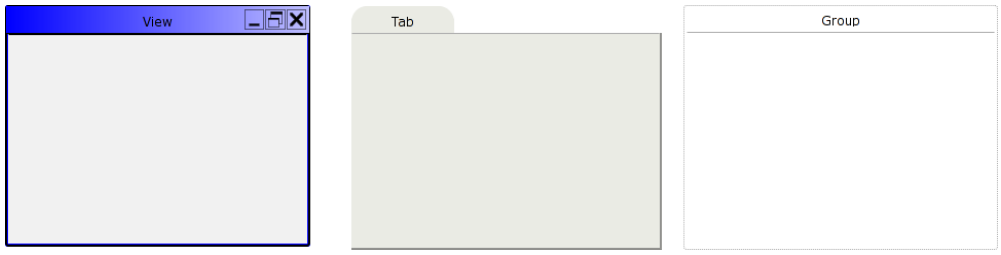

Abbildung 15 – Notationselemente der Container Elemente

Der Hintergrund von Group ist durchsichtig.

5.1.4.1.1 „editMode“ Aspekt

Die o.g. UI Elemente zeichnet der editMode Aspekt aus. DEFAULT zeichnet das Element nicht aus, da es den Normalzustand darstellt. EDITABLE wird grün signalisiert, READ-ONLY blau, und MANDATORY rot. Primitive und Containerelemente signalisieren die Farbe durch Umrandung, komplexe Elemente durch ihre Vordergrundfarbe. Abbildung 16 zeigt eine editierbare Gruppe, ein verpflichtendes Textfeld sowie eine nicht editierbare Tabelle.

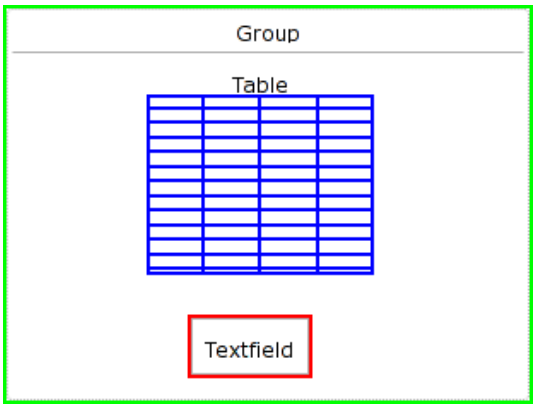

Abbildung 16 – Editierbare Gruppe

5.1.5 Datenhaltende Elemente

Abbildung 17 zeigt die datenhaltenden Notationselemente.

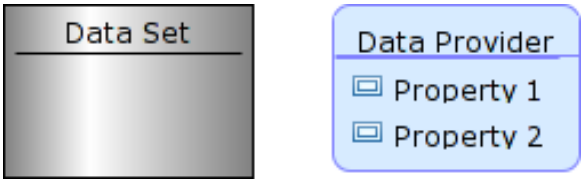

Abbildung 17 – Datenhaltende Notationselemente

Auf der linken Seite ist ein Data Set zu sehen. Da keine geeignete dezente Darstellung für eine Menge gefunden wurde, erinnert der Farbverlauf im Hintergrund entfernt an eine Tonne und hebt sich gut von dem der zu beinhaltenden Data Provider ab. Auf der rechten Seite ist ein Data Provider zu sehen, welcher zwei Properties beinhaltet. Der Data Pro-

vider ähnelt absichtlich einer graphisch ansprechenden UML Klasse ohne Methoden Bereich. Die beinhaltenden Properties sind im Gegensatz zu anderen Containern nicht frei positionierbar und werden als Liste dargestellt. Properties werden als Text mit einem Symbol dargestellt. Eine Darstellung mit Symbol wurde gewählt, um sie besser als eigenständiges Element auszuzeichnen, da jeder andere Text, vom Label abgesehen, eine Beschriftung und kein vollständiges Sprachelement darstellt.

5.1.6 Verbindungen

Abbildung 18 zeigt die Notationselemente für Verbindungen.

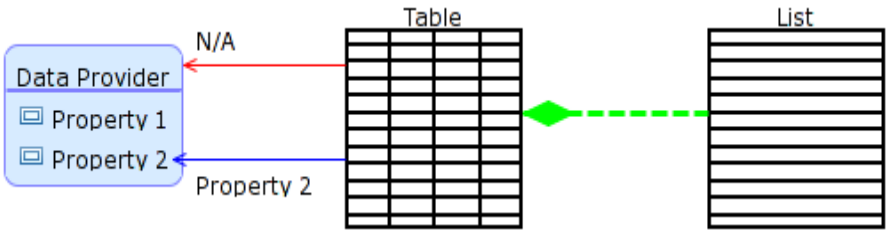

Abbildung 18 – Notationselemente für Verbindungen

Auf der rechten Seite ist die Master Detail Beziehung zwischen Table und List zu sehen. Sie wird durch eine grüne gestrichelte Linie mit einem Diamanten auf der Seite des Masters dargestellt. Die linke untere Verbindung zeigt eine Data Relation, welche als blaue Linie mit einem Pfeil auf die anzuzeigende Property dargestellt wird. Die linke obere Verbindung zeigt eine fehlerhafte Data Relation. Sie wird rot dargestellt und zeigt auf den Data Provider.

6 Werkzeugwahl

6.1 UML2

Das bei OiO eingesetzte Modellierungswerkzeug MagicDraw kann ausschließlich Eclipse UML2 Modelle importieren und exportieren. Direkt mit dem Eclipse UML2 Meta-Modell arbeiten die Modellierungswerkzeuge „Topcased UML“ oder „Rational Software Architect“, welches jedoch weitere, wahrscheinlich Layout Informationen hinzufügt. Zudem unterstützt der Generator „openArchitectureWare“ Eclipse UML2 direkt über EMF, und die Alternative AndroMDA indirekt über sog. MetaFacades. Von einer Wahl kann kaum die Rede sein, da Eclipse UML2 erfreulicherweise ein quasi Standard ist.

6.2 Framework des Graphischen Editors

Das Framework für den GE wird anhand eines Ausschlussverfahrens bestimmt. Unfreie Frameworks, wie Microsofts Software Factories oder MetaEdit+, werden gemäß den nicht funktionalen Anforderungen nicht berücksichtigt. Vier freie Frameworks zum erstellen von GE wurden identifiziert:

- Generic Modeling Environment (GME) [GME08] von der Vanderbilt Universität
- Generic Eclipse Modeling System (GEMS) [GEM08]
- GEF
- GMF

Die gewählte UML2 Implementierung basiert auf EMF. GEF und GME unterstützen EMF nicht explizit und werden nicht weiter betrachtet. Obwohl GEMS auf der GMF-Runtime basiert, sprechen drei Gründe gegen GEMS:

Es ist Teil von Eclipse „Generative Modeling Technologies“, d.h. ein Forschungsprojekt und nicht stabil.

Zu Beginn dieser Arbeit war der Dokumentationsumfang sehr gering

Im GEMS Ablauf werden automatisiert aus einem eigenen Modell EMF Modelle erzeugt [Wuc07] , was darauf hindeutet, dass ggf. kein importieren des Eclipse UML2 Metamodells möglich ist.

Für den GE wurde folglich GMF verwendet.

7 Generativer Entwicklungsprozess von GMF

7.1 Einleitung

Die Architektur von GMF bietet eine Reihe von Anpassungsmöglichkeiten. In diesem Kapitel werden die wesentlichen vorgestellt. Im Anschluss wird eine übertragbare und aufgabenstellungsunabhängige Vorgehensweise vorgestellt. Dies ist erforderlich, da ihr Kernkonzept nicht in den betrachteten Editoren oder Beispielen berücksichtigt wird und auf vier Präsentationsfolien [Tik07] dokumentiert ist. Diese Vorgehensweise wurde für den im Rahmen dieser Arbeit zu erstellenden GE verwendet. Abgewichen wurde von der konsequenten Verwendung von Xpand Aspekten aufgrund eines Fehlers [Web07] in der verwendeten GMF Version. Ist im Folgenden von einem Aspekt die Rede, ist ein Software Aspekt, also ein Aspekt im eigentlichen Sinne gemeint. Ein oAW-Aspekt qualifiziert das technische, aus Aspekt orientierter Programmierung bekannte Hilfsmittel der openArchitectureWare Template-Sprache Xpand.

7.2 Anpassungsmöglichkeiten

Die wesentlichen Anpassungsmöglichkeiten bei GMF werden kurz vorgestellt. Eine ausführliche Betrachtung mit Vorgehensweise, Wertung, Verwendungsempfehlung und verfügbaren Implementierungen ist im Anhang B – Anpassungsmöglichkeiten S. V zu finden.

7.2.1 Anpassungen am Quellcode

Das Generat wird angepasst und die veränderten Methoden mittels `@generated NOT` vor Überschreibung geschützt.

7.2.2 Anpassung durch Extensions

Die Eclipse Service-Plattform und bereitgestellte GMF Extension-Points werden verwendet um Erweiterungen hinzuzufügen. Hervorzuheben ist,

dass sich diese Erweiterungsmöglichkeit sehr gut zum Anpassen von fremden Editoren eignet.

7.2.3 Templateanpassungen anhand eines erbenden Generatormodells

Ein eigenes Meta-Modell wird erstellt, dessen Meta-Klassen von denen der gmfgen-Meta-Klassen erben. Die Templates werden auf die Spezialisierungen angepasst und Instanzen von Meta-Klassen im Generatormodell werden ersetzt.

7.2.4 Templateanpassungen anhand eines erweiternden Generatormodells

Ein eigenes Meta-Modell wird erstellt, dessen Meta-Klassen Referenzen auf gmfgen-Meta-Klassen besitzen. Dieses Konzept ist in der MDSD unter dem Begriff „Model Marking“ [Sta07] bekannt. Instanzen der gmfgen-Meta-Klassen im Generatormodell werden von Instanzen der eigenen Meta-Klassen referenziert. Anpassungen in den Templates greifen, wenn eine gmfgen-Meta-Klassen Instanz markiert ist.

7.3 Empfohlener Entwicklungsprozess

In diesem Abschnitt wird der empfohlene Entwicklungsprozess vorgestellt und näher erläutert. Das zentrale Konzept ist das erweiternde Generatormodell.

Normale initiale Schritte

Die ersten Schritte sind mit denen der normalen GMF Entwicklung identisch. Domänenmodell, Tooling-Modell, Graph-Modell und Mapping-Modell sind zu erstellen. Um spätere Verschiebungen innerhalb des Generatormodells zu reduzieren, bietet es sich im an, bereits im Mapping Modell möglichst viele Elemente, wenn auch nicht funktional, zu berücksichtigen. Im Anschluss wird ein Generatormodell erzeugt.

Namensgebung im Generatormodell

Für eine Instanz jedes Typs auf gleicher Hierarchieebene sind die zu erzeugenden Klassennamen anzupassen. Ein erneutes Erzeugen des Generatormodells zeigt welche Änderungen in der verwendeten Version erhalten bleiben, wo sie sich also lohnen. Die standardmäßige Benamung von GMF ist in nicht trivialen Situationen nicht aussagenkräftig. In Hinblick auf zu erwartenden Anpassungen an den generierten Klassen ist der Mehrwert einer aussagekräftigen Namensgebung groß.

Erste Anpassungen

Der Quellcode wird generiert und muss angepasst werden. Änderungen sind bis zum Funktionieren eines mustergültigen Einzelfalls vorzunehmen, angepasste Methoden sind mit `@generated NOT` auszuzeichnen. Wurde die Anpassung erfolgreich für den Beispielfall getestet, gilt es sie für eine höhere Abstraktionsebene verfügbar zu machen. Sie gedankliche einer Metaklasse zuzuordnen bietet sich an, um später nicht die technisch einfachste zu wählen.

Initiales erstellen eines Metamodells

Der folgende Schritt ist nur einmalig für ein Projekt notwendig. Es gilt zuerst ein eigenes Meta-Modell zum Testen zu erstellen und das korrekte Laden in das Generatormodell sicherzustellen. Hierfür wird ein normales Ecore Modell erstellt, ein Wurzel Element angelegt und das gmfgen-Meta-Modell geladen. Dem Wurzelelement wird eine Metaklasse angehängt und ihr eine Referenz auf eine Metaklasse des gmfgen-Metamodells hinzugefügt. Im Zusammenhang mit dem Laden des Erweiterungsmodells existierte ein Fehler in der verwendeten GMF Version. Dieser macht es erforderlich mittels Text- oder XML-Editor im Generatormodell die Option `xsi:schemaLocation` hinzuzufügen und korrekt zu belegen. Alternativ kann der Fehler auch mit dem Bugfix [Kuh08] behoben werden. Wird keine der beiden Varianten gewählt muss das

Meta-Modell generiert und als Eclipse PlugIn eingebunden werden. Dies behindert ein „Lebendiges Metamodell“ [Sta07] . Ist das Erweiterungsmodell geladen, sind Instanzen der zu erweiternden gmfgen-Metaklasse nicht von einer Instanz der eigenen Metaklasse referenzierbar. Obwohl die in beiden Modellen verwendeten gmfgen-Meta-Modelle ein Wurzelelement mit identischem Namensraumattribut besitzen, verwenden die beiden Modelle unterschiedliche Pfadangaben zu ihnen. Für den gmfgen-Editor sind die Meta-Modelle folglich nicht identisch und für das Erweiterungsmodell existiert keine gültige, referenzierbare Instanz der zu erweiternden Metaklasse. Die Referenz zum gmfgen-Meta-Modell muss mittels eines Text- oder XML-Editors aus dem Kopf des Generatormodells kopiert, und alle Referenzen zum gmfgen-Meta-Modell in dem Erweiterungsmodell hierdurch ersetzt werden. Wurde dieser Schritt korrekt vorgenommen, lädt der Ecore Editor im weiteren Projektverlauf das gmfgen-Meta-Modell über die „richtige“ Referenz. Später erstellte Meta-Klassen werden nun mit der im Generatormodell verwendeten Referenz gespeichert, ein späteres korrigieren dieser entfällt also. Die zum Testen erstellte Metaklasse ist bis zum Erstellen der ersten produktiv einzusetzenden Metaklasse nicht zu löschen.

Erstellen von Meta-Klassen

Im Normalfall ist eine eigene Metaklasse nach einer Anpassung des Generats zu erstellen. Die Ausnahme stellen Anpassungen da, die einem bereits vorhandenen Aspekt zugeordnet werden können und für die es bereits eine zu diesem Aspekt gehörende, den zu erweiternden Typ referenzierende Metaklasse existiert. Müssen für einen Aspekt mehrere Meta-Klassen referenziert werden, bietet es sich an die referenzierenden Klassen einer Gruppe, bzw. einem Container innerhalb des Erweiterungsmodells zuzuordnen. Da Instanzen im Erweiterungsmodell an sich nur Instanzen im Generatormodell referenzieren müssen, ist ihre Position im Erweiterungsmodell quasi frei wählbar. Sie lassen sich also später

noch gut ordnen und Referenzmultiplizitäten sind änderbar. Wurde eine Metaklasse erstellt ist zu klären, ob oder welche Eigenschaften sie besitzen muss. Besitzt sie lediglich eine Referenz auf einen gmfgen-Typ, ist sie mit den aus Java bekannten „Marker“-Interfaces wie „Serializable“ vergleichbar. Eine besessene Eigenschaft kann beispielsweise mittels eines String Attributs den Namen einer Variablen tragen oder andere Modellelemente referenzieren. Im letzten Fall ist es selbstverständlich möglich Elemente eines dritten Modells, wie z.B. Stereotype eines UML Profils zu referenzieren.

Implementierung der Templates

Der Fall einer referenzierten gmfgen-Instanz ist nun in den Templates zu berücksichtigen und die Anpassungen im Falle eines referenzierenden, bestimmten Metatyps umzusetzen. Hierbei ist die statische Struktur des GMF Template Verzeichnisses zu berücksichtigen und nicht zu ändern. Zudem sollten eigene Änderungen möglichst außerhalb eines bestehenden `DEFINE` Blocks vorgenommen werden. Sofern möglich ist dieser mittels eines oAW-Aspekts zu dekorieren oder der zu ändernde Teilbereich in einen eigenen `DEFINE` Block zu verschieben und dieser zu dekorieren. Um referenzierende Elemente zu erhalten ist eine oAW-Extension in `xpt::EMFUtils` zu verwenden. Referenzierende Objekte auf das aktuelle erhält man mittels `getReferencingObjects(this)`. Die zurückgelieferte Menge ist mittels `.typeSelect(Metaklasse)` auf Instanzen einer Metaklassen zu filtern, auf eine nicht leere Menge zu prüfen und z.B. das nullte Element der Menge auf die Metaklasse zu „casten“. Wurde das Template angepasst, wird `@generated NOT` der Methode entfernt, erneut generiert und ggf. Fehler beseitigt. Dieser Vorgang ist zu wiederholen bis alle Änderungen durch die Templates vorgenommen werden. Das Ergebnis ist nun erneut zu testen, hierbei sind insbesondere Zustände die in der exemplarischen Lösung nicht überprüft werden konnten zu berücksichtigen.

Nächste Iteration

Da die M2M Transformation vom Mapping-Modell zum Generatormodell Erweiterungsmodelle entfernt, ist vor diesem Vorgang das Generatormodell zu kopieren. Nachdem die M2M Transformation durchgeführt wurde sind beide Modelle mit einem Text- oder XML-Editor zu öffnen und der Kopf des alten Generatormodells, sowie das am Ende liegende Erweiterungsmodell in das neue Generatormodell zu kopieren.

Im weiteren Projektverlauf wird auffallen, dass das gmfgraph-Modell vergleichsweise schlecht skaliert. Es sollte seine Möglichkeit genutzt werden, lediglich die Schnittstelle von „legacy“ Draw2D Figuren zu beschreiben. Sollen mühevoll erstellte Figuren fein justiert werden, kann aus einem bestehenden gmfgraph-Modell ein Plugin welches Draw2D Figuren sowie ein gmfgraph-Modell mit ausschließlich Schnittstellendefinitionen beinhaltet, generiert werden.

8 Design des Graphischen Editors

Dieses Kapitel klärt das Softwaredesign des GE anhand der dafür von GMF entwickelten Sprache, den fachlichen Modell gmftool, gmfgraph und gmfmap sowie den Konzepten der eigens entwickelten Sprachen, den eigenen Metamodellen. Es wird den Empfehlungen gefolgt das Modell als Sprache innerhalb des Projekts zu verwenden [Sta07] . Ist der Leser bereits mit GEF vertraut, würde er an dieser Stelle eine Beschreibung des konzeptionellen Aufbaus von EditParts, EditPolicies sowie Commands und den Argumenten der Requests erwarten. Diese Konzepte sind implizit in den Modellen enthalten. Bevor das Design anhand der Modelle erklärt wird, ist zu entscheiden, ob der GE nativ auf UML Modellen aufsetzt, oder ob von und zu ihnen abgebildet bzw. transformiert wird.

8.1 Designentscheidung Metamodell

Die Entscheidung, ob der GE direkt auf einem UML Modell aufbaut oder sein eigenes Metamodell benutzt, ist fundamental. Setzt der GE nicht direkt auf UML Modellen auf, muss ein eigenes Metamodell für ihn entwickelt werden, welches mindestens die Konzepte des Problemraums beschreibt. Diese Überlegung bietet sich aufgrund folgender fünf Eigenschaften der nicht angepassten, generativen Architektur an.

1. Sie unterstützt nicht mehrere Metamodelle im gleichen Editor. Wie in den Grundlagen zu Eclipse UML2 beschrieben, erweitert ein UML Profil das UML Metamodell.
2. Dynamisches EMF wird nicht unterstützt. Wie ebenfalls in den Grundlagen zu Eclipse UML2 beschrieben, werden die Elemente der Profile mittels dynamischem EMFs erzeugt.
3. Es fehlt das Konzept der „Element Type Registry“ oder ein vergleichbares. Das gmfgen-Metamodell setzt es zwar unvollständig um, Anpassungsmöglichkeiten fehlen jedoch. Am direkten Impor-

tieren des Domänenmodells ins Mapping-Modell wird das fehlende Konzept ersichtlich. Ihm am nächsten kommen die im Mapping-Modell möglichen „Constraints“.

4. Nicht primitive Abbildungen, wie sie in den Abbildungsvorschriften zu UML Profilen definiert wurden, werden nicht unterstützt.
5. In Metamodellen definiertes Verhalten der Metaklassen, also ihre Funktionen, können nicht genutzt werden. Deshalb ist es z.B. nicht möglich eine UML Assoziation zu erstellen ohne den Quellcode anzupassen.

Wird in diesem Kapitel von einer M2M Transformation gesprochen, bleibt bewusst offen ob sie direkt im Speicher oder in einem zusätzlichen Schritt durchgeführt wird. Die unterschiedlichen Überführungsmöglichkeiten von einem eigenen Metamodell zu dem von UML mitentwickelten Profil werden vorgestellt und die Wahl begründet.

8.1.1.1 Metamodell Facade

Ein eigenes Metamodell wird erstellt und die Java Implementationsklassen generiert. Die Elementklasse sowie ihre Fabriken sind komplett neu zu implementieren. Die Implementierung leitet Änderungen an ein hinter diesem Modell liegendes UML Modell gemäß den Abbildungsvorschriften weiter. Zudem muss auf Änderungen des UML-Modells und dessen beinhaltender Elemente mittels „Listenern“ (Observer [Gam95]) reagiert werden und in ihrer Folge ggf. neue Elemente erzeugt werden.

Ausschlussgrund: Der Aufwand und die Fehleranfälligkeit einer solchen Implementierung wurden als zu hoch erachtet.

8.1.1.2 M2M Transformation auf DSL Metamodell

Ein eigenes Metamodell wird wie beim dekorierenden Metamodell erstellt und der GE auf dessen Basis entwickelt. Im Anschluss wird je eine lesende und schreibende M2M Transformation entwickelt, die die In-

stanz des eigenen DSL-Metamodells mit einem UML Modell gemäß den Abbildungsvorschriften abgleicht.

Ausschlussgrund: Bei aktualisierender Änderung des UML Modells entstehen zwangsläufig Konsistenzprobleme. Umbenennung von Elementen und überschreibende Abgleichung unter Rücksichtnahme von Referenzen, die nicht im DSL-Metamodell existieren, lassen diese Form ausscheiden.

Wird das Ziel-UML-Modell neu erzeugt, müssen Elemente, die nicht dem DSL-Metamodell bekannt sind, gesichert und beim Schreiben berücksichtigt werden. Die o.g. Abgleichungsprobleme treten nun auch hier wieder auf und wären voraussichtlich nur mit einem dritten Änderungsmodell lösbar. Auch bei dieser Variante wurde der Implementierungsaufwand als zu hoch erachtet.

8.1.1.3 M2M Transformation auf „Middleweight“ UML Metamodell

Es ist ein eigenes Metamodell zu erstellen, welches Metaklassen aus dem UML Metamodell erweitert. Hilfestellungen hierzu sind in [Hus07] [Bru07] [Bru06] zu finden. Es sei darauf hingewiesen, dass der Anwendungsfall einer der wenigen darstellt, in denen der „Middleweight“-Ansatz dem „Heavyweight“-Ansatz vorzuziehen ist. Die eigenen Metaklassen sind mit denen der Stereotype identisch, erben aber von den Metaklassen und erweitern sie nicht. Bei den folgenden M2M Transformationen bleiben alle Informationen in einem Modell erhalten.

Ausschlussgrund: Es entsteht eine direkte Abhängigkeit zu einer speziellen Version mit Anpassungsaufwand bei Versionsänderungen. Es wird erwartet, dass die M2M Transformationen vergleichsweise komplex würden und komplexes, automatisiertes Testen erforderlich wäre. Dieser Aufwand ist im gegeben zeitlichen Rahmen nicht zu bewältigen.

8.1.1.4 Entscheidung

Obwohl der GE nur prototypisch ist, würde jede der o.g. Alternativen zur direkten Verwendung von UML Modellen inakzeptable Folgen haben. Der Autor entschied sich somit trotz erkannter Probleme mit der generativen GMF Architektur für die direkte Verwendung von UML Modellen.

8.2 Tooling Model – gmftool

Das gmftool beschreibt, wie die Werkzeugleiste eines GE aussehen soll. Es ist das einfachste aller erstellten Modelle und seine Auswirkung ist leicht ersichtlich. Diese Eigenschaft wird an dieser Stelle genutzt, um einmalig den vollständigen Aufbau eines der GMF Modelle zu zeigen und seine wichtigsten Attribute zu klären. Die fachlichen GMF Modelle erscheinen auf den ersten Blick komplexer als sie sind.

Abbildung 19 zeigt das Tooling-Modell des GE und sein Resultat.

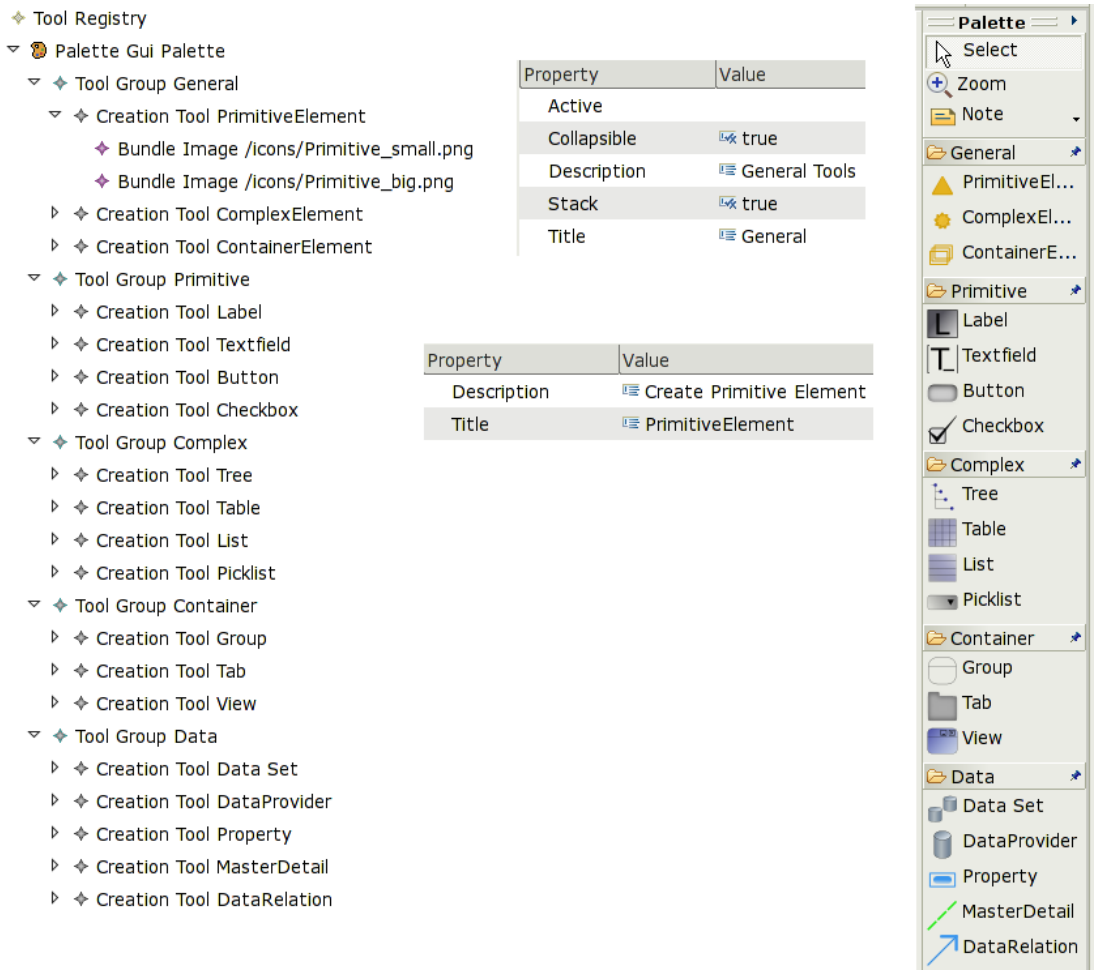

Abbildung 19 – Tooling-Model des GE

Auf der linken Seite ist das Tooling-Modell zu sehen, mittig der „Properties View“ von einer Gruppe und einem Werkzeug und rechts das Ergebnis im GE. In der Baumansicht ist die Struktur gut zu erkennen: Wurzelelemente, Gruppen, Werkzeuge und der Pfad zu ihren Symbolen. Gruppen können faltbar gemacht werden und einen Titel tragen, wie in der oberen, mittleren Abbildung zu sehen, Werkzeuge dagegen einen Titel und eine Beschreibung. Offensichtlich nehmen Gruppen Werkzeuge und Werkzeuge einen Pfad zu sowohl ihrem großen als auch ihrem kleinen Symbol auf. Ihnen wird erst im Mapping Modell eine Funktion zugewiesen. Die ersten drei Werkzeuge der Werkzeugleiste werden standardmäßig erzeugt solange selbst kein „Standard Tool“ erstellt wurde. Dies folgt dem Konzept „Configuration by Exception“ [Sta07] .

Hervorzuheben ist, dass die Gruppen `Primitive`, `Complex` und `Container` zum Erzeugen der Elemente des Problemraums nicht notwendig sind und im Tooling Modell als Werkzeuge mit initialer Wertzuweisung vorgesehen werden. Sie beinhalten also Werkzeuge, die dem Komfort dienen.

8.3 Graphical Model – gmfgraph

Das gmfgraph-Modell für den erstellten GE ist ebenfalls sehr einfach. Dies liegt darin begründet, dass, bis auf die verwendeten Verbindungen und eine zu ihr gehörende Beschriftung, nur die Schnittstellen der in Java geschriebenen Notationselemente modelliert wurden. An dieser Stelle wird deshalb nur das gmfgraph-Element der Master Detail Verbindung beschrieben.

Abbildung 20 zeigt den Modellabschnitt des Notationselements der MasterDetail Verbindung.

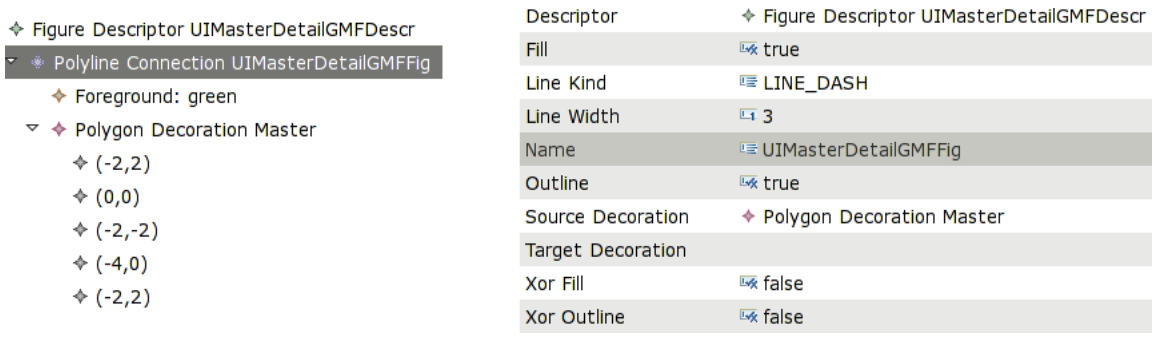

Abbildung 20 –Notationselement MasterDetail Verbindung

Die Figur von MasterDetail ist eine grüne Linie (`Polyline`), wie links zu erkennen ist. Diese ist, wie rechts abgebildet, gestrichelt (`LINE_DASH`) mit einer Breite von drei Pixeln und besitzt die Quelldekoration `Master`. Die Dekoration `Master` ist, wie links zu sehen, ein `Polygon` mit einer geschlossenen Linie, deren Koordinaten Kindelemente der Dekoration sind.

8.4 Mapping Model – gmfmap

8.4.1 Einleitung

Das Mapping-Modell ist das bedeutendste fachliche Modell, dessen Funktion bereits in Kapitel 2.7.3.3 erklärt wurde. Zuerst wird eine Übersicht gegeben, danach wird das „Mapping“ ausgewählter Elemente erklärt. Im Zusammenhang mit den abgebildeten Modellen ist zu berücksichtigen, dass die generative GMF Architektur für DSL entwickelt wurde und von unterschiedlichen Metaklassen ausgeht. Deshalb wurde es anscheinend als unnötig erachtet die Mapping-Modellelemente benennbar zu machen. Sie sind jedoch durch ihre ihnen zugewiesene Figur zu unterscheiden. Um den Einstieg in die Quellen zu erleichtern, wurde auf aussagekräftigere nicht originalgetreue Diagramme verzichtet.

8.4.2 Übersicht

Abbildung 21 zeigt die Übersicht des gmfmap-Modells. Zu beachten ist nur der Typ wie z.B. `Link Mapping` sowie die zugeordnete Figur wie z.B. `UIMasterDetail`.

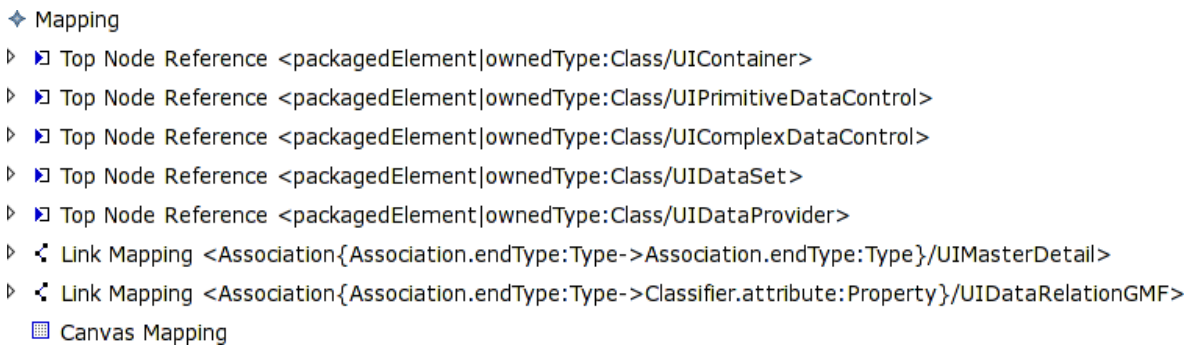

Abbildung 21 – gmfmap-Modell

Das `Canvas Mapping` zeichnet die Zeichenfläche aus, seine Metaklasse ist `Model` aus der Domäne `uml`. Direkt auf der Zeichenfläche können alle `Top Node Reference` als Figur sowie alle `Link Mapping` als Verbindungen gezeichnet werden. Die Verbindungen sollten nicht überraschen,

wohl aber, dass beispielsweise auch primitive Elemente direkt auf der Zeichenfläche erstellt werden können. Dies wurde aus Gründen der Fehlertoleranz gewählt. Durch einen externen UML Editor fehlerhaft auf der Zeichenfläche angelegte Elemente werden somit angezeigt. Es wurde jedoch festgestellt, das Validierungsregeln mittels der Object Constraint Language (OCL) beim Laden des Editors selbst validiert werden. Dies schlägt fehl, da ihm die Metamodellelemente des UML Profils zu diesem Zeitpunkt unbekannt sind. Eine Validierung erfolgt deshalb bei dem umgesetzten prototypischen GE nicht.

8.4.3 Primitives Element

Zunächst wird betrachtet, wie primitive Elemente definiert werden. Die bereits in der Übersicht erkennbare `Top Node Reference` beinhaltet zwei einzustellende Attribute, welche für alle Klassen gleich zu belegen sind.

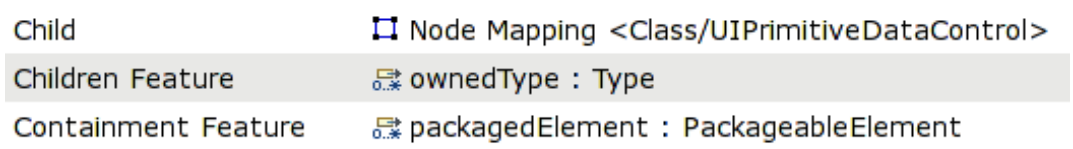

Abbildung 22 – Top Node Reference einer Klasse

Abbildung 22 zeigt den Property View der `Top Node Reference` einer Klasse.

Das erste Attribut der Abbildung wird automatisch belegt und ist nicht von Interesse. `Children Feature` ist mit dem Namen der Aggregation zu belegen, aus welcher dieses Element ermittelt wird. `Containment Feature` ist mit dem Namen der Aggregation zu belegen, in die das Element gespeichert wird.

Betrachtet wird nun die eigentliche Definition des primitiven Elements, welche übrigens fast identisch mit der des komplexen Elements ist.

Abbildung 23 zeigt die Mapping-Definition des primitiven Elements.

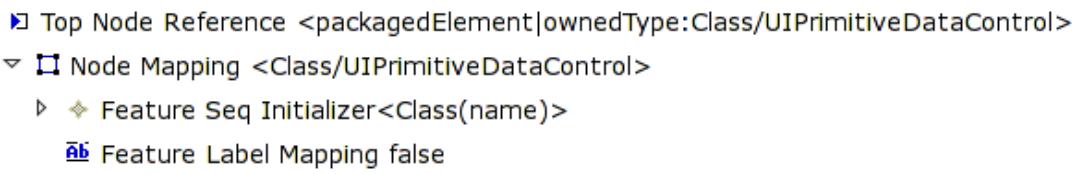

Abbildung 23 – Mapping-Definition des primitiven Elements

Im Node Mapping wird die Metaklasse des UML Modells mit einer Notation des gmfgraph-Modells und einem Werkzeug verbunden. Innerhalb des primitiven Elements liegen ein Label, welches mit einem Attribut der Klasse und einer Label-Figur verbunden wird und ein Sequenz-Initialisierer. Dieser weist der neu erstellten Klasse mittels eines nicht trivialen OCL Ausdrucks einen eindeutigen Namen zu. Im Hinblick auf die im gmftool-Modell vorgesehen „Komfortwerkzeuge“ ist anzumerken, dass sich dem „Node Mapping“ nur ein Werkzeug zuweisen lässt. Weiterhin stellt das beschriebene Label in der aktuellen Beschreibung wirklich den Namen der Klasse, und nicht, wie gefordert, den Namen des Stereotyps dar. Beides wird später mittels der zusätzlichen Metamodelle beschrieben.

8.4.4 UIContainer

Der UIContainer (Abbildung 24) beinhaltet selbst andere Notationselemente, seine Definition im Mapping-Modell ähnelt der des UIDataSet.

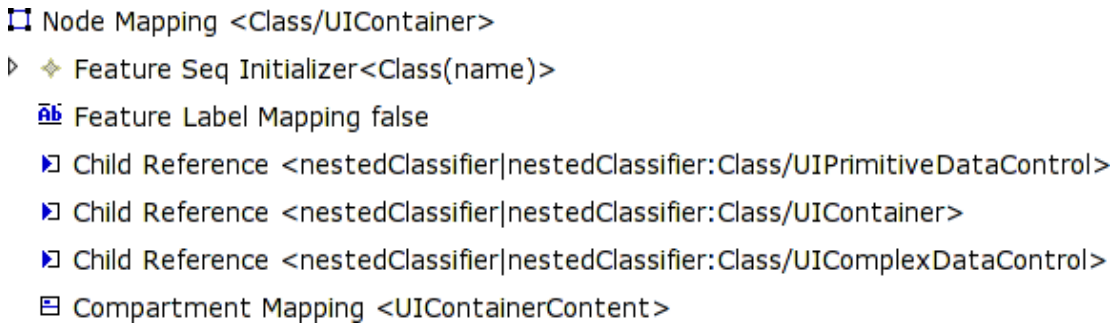

Abbildung 24 – Mapping-Definition des UIContainers

Node Mapping, Feature Seq Initializer sowie Feature Label Mapping sind mit denen des primitiven Elements nahezu identisch. Das

Compartment Mapping zeichnet einen eigenen Bereich aus, in dem Kindelemente gruppiert werden können. Jeder EditPart, also jedes Node-Mapping, besitzt zwar implizit exakt einen solchen, auch als „Content Pane“ bezeichneten Bereich, jedoch wird Compartment Mapping selbst auf einen EditPart mit eigenem Verhalten abgebildet. Dies ermöglicht „Scrollbars“ anzuzeigen, falls in ihm beinhaltete Elemente die erlaubte Zeichenfläche überschreiten und eine automatische Größenanpassung aufgrund fixierter Größe unmöglich ist. In jeder Child Reference wird an dieser Stelle ein Node Mapping, im Falle eines primitiven Elements also die bereits beschrieben Kombination aus Werkzeug, Metaklasse und Notation, referenziert. Zudem wird der Name der haltenden Aggregation von Class angegeben, die das neue Kind hält. Offensichtlich lässt nestedClassifier auf innere Klassen schließen. Das Beinhalten anderer Elemente mittels stereotypisierten Assoziationen wird später durch eigene Metamodelle ausgedrückt. Jede Child Reference wird noch dem Compartment Mapping zugeordnet. Dabei ist hervorzuheben, dass die Referenz auf eine Top Node Reference bei Child Reference für rekursiv schachtelbaren Elementen zwingend erforderlich ist.

8.4.5 UIDataProvider

Der UIDataProvider erweitert nun den Aufbau des UIContainers, da er für die beinhaltenden Instanzen von Property kein Node Mapping referenziert, sondern ein eigenes definiert.

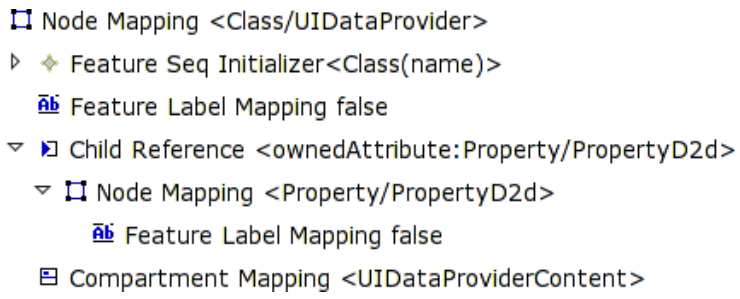

Abbildung 25 - Mapping-Definition des UIDataProviders

Das Node Mapping ist mit der bereits unter den Top Node Reference beschriebenen vergleichbar.

8.4.6 UIDataRelation

Die Data Relation Definition ist die komplexere der beiden Verbindungen und ihre Konzepte lassen sich auf die Master Detail Verbindung übertragen.

Abbildung 26 zeigt die Mapping-Definition der UIDataRelation

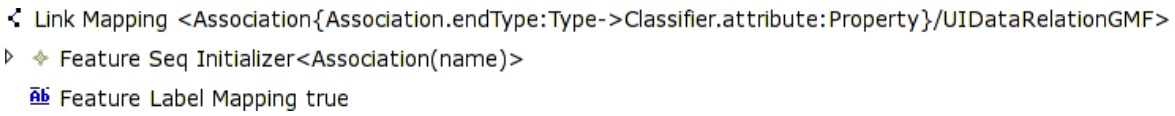

Abbildung 26 – Mapping-Definition der UIDataRelation

Die Baumansicht der Mapping-Definition ist kurz. Allerdings ist hervorzuheben, dass es sich um ein Link Mapping handelt, das eine Beschriftung beinhaltet. Der „Domain meta information“ Ausschnitt des „Property Views“ des Link Mapping bietet Klärungsbedarf.

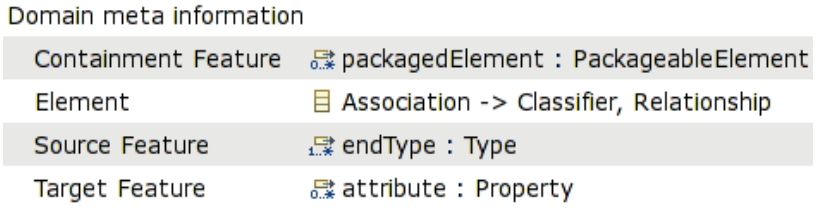

Abbildung 27 – Property View der UIDataRelation

Abbildung 27 zeigt Ausschnitt des Property View vom Link Mapping der UIDataRelation

Die Konzepte des Containment Feature und Element wurden bereits erklärt. Die Referenz Source Feature der Assoziation verweist auf ihre Quelle und Target Feature auf ihre Senke. Augenscheinlich sind beide zunächst falsch belegt. Die endType Referenz wird, wie in den Abbildungsvorschriften (s.S.38) bereits erklärt, nicht zwangsläufig von der Assoziation gehalten. Mittels attribute des Target Feature wird sogar eine Referenz der Assoziationsmetaklasse verwendet, welche keinen

Bezug zur Assoziationssenke besitzt. Sie wurden gewählt, da der Typ ihrer Referenz bereits der gewünschten Metaklasse entspricht, was die Anpassungen der Templates später geringfügig erleichtert. Die Verbindung UIDataRelation muss später durch Instanzen der eigenen Metaklassen erweitert werden.

8.4.7 Zusammenfassung

Der Aufbau des GE wurde Anhand des Mapping-Modells, welches implizit seine Architektur definiert, beschrieben. Auf spezifische Aspekte des Anwendungsfalls wurde hingewiesen, die Vorstellung ihrer Konzepte steht allerdings noch aus.

8.5 Erweiterungsmetamodelle

Die bereits erwähnten Erweiterungsmodelle werden benötigt, um auf konzeptioneller Ebene die generative Architektur von GMF anzupassen bzw. zu erweitern. Ihre Metaklassen sind folglich allein stehend zu betrachten, spiegeln keinen vollständigen Ablauf wieder und lassen sich somit nur bedingt einordnen und gruppieren. Die Metamodelle der generativen GMF Architektur werden nicht erklärt, da sie einen allgemeinen Charakter für diese besitzen. Sie sind auch nicht dokumentiert. Die Bedeutung der verwendeten Elemente wurde anhand ihrer Anwendung, also anhand von Instanzen ihrer Metaklassen, im GE erläutert. An dieser Stelle wird nun die Bedeutung der Metaklassen der Erweiterungsmodelle erklärt, da diese aufgrund ihres Designs interessant sind. Ihre Instanzen werden erst im Generatormodell, welches der Autor der Implementierung zuschreibt, verwendet. Zur Strukturierung werden zwei Metamodelle erstellt, wobei das Erste den Namensraumpräfix „guiGenModel“ und das Zweite „guiUMLGenModel“ trägt. Die eigenen Meta-Modelle referenzieren sich nicht gegenseitig. Das erste Metamodell hält Referenzen auf Metaklassen des gmfgen-Metamodells, das Zweite zudem Referenzen auf Elemente des erstellten UML Profils. Hieraus ist

nicht zu schließen, dass Elemente des ersten Metamodells nicht implizit vom Profil abhängen, lediglich die Art der Abhängigkeit ist unterschiedlich. Die Stereotypen werden im ersten Modell statisch, also fest in den Templates verwendet. Um den Stereotypen zu ändern, sind die Templates anzupassen. Im zweiten Modell werden die Stereotype des Profils referenziert. Der Stereotyp wird geändert, indem eine Referenz im Modell geändert wird. Templates für das zweite Metamodell zu schreiben bedeutet einen geringfügig höheren Aufwand zu Beginn, welcher nur bei zu erwartender, benötigter Flexibilität gerechtfertigt ist.

Bevor nun die Metamodelle anhand ihrer Diagramme erklärt werden, sind folgende Besonderheiten zu berücksichtigen. Das in jedem Metamodell vorhandene `NamedElement` und die zugehörige Vererbungsnotation werden zur besseren Übersicht nicht dargestellt. Referenzen auf Metaklassen anderer Metamodelle werden durch einen Kommentar erkenntlich gemacht. Referenzen auf das gmfgen-Meta-Modellelement wurden immer `ext` genannt, auf Profil- oder UML-Elemente mit `uml`. Die Multiplizitäten geben nicht zwangsläufig ihre Verwendung wieder. Der Grund hierfür liegt an einem unübersichtlichen Auswahldialog des EMF Baumeditor für einzelne Elemente. Folglich wird bei 0..* Multiplizitäten auf Implementierungsseite teilweise nur das erste berücksichtigt. Ob eine Metaklasse abstrakt ist, wird nicht erwähnt.

Begleitend zu der abstrakten Erklärung bietet sich ein Blick auf konkrete Instanzen der Meta-Klassen an. Sie befinden sich im Generatormodell unter `/de.oio.mdsd.tools.guiModel.guiModeler.Dev/model/GuiModeler.gmfgen`.

8.5.1 guiGenModel

Das `guiGenModel` befindet sich in der Datei „ExtendedGenModel.ecore“. Metamodell des guiGenModels.

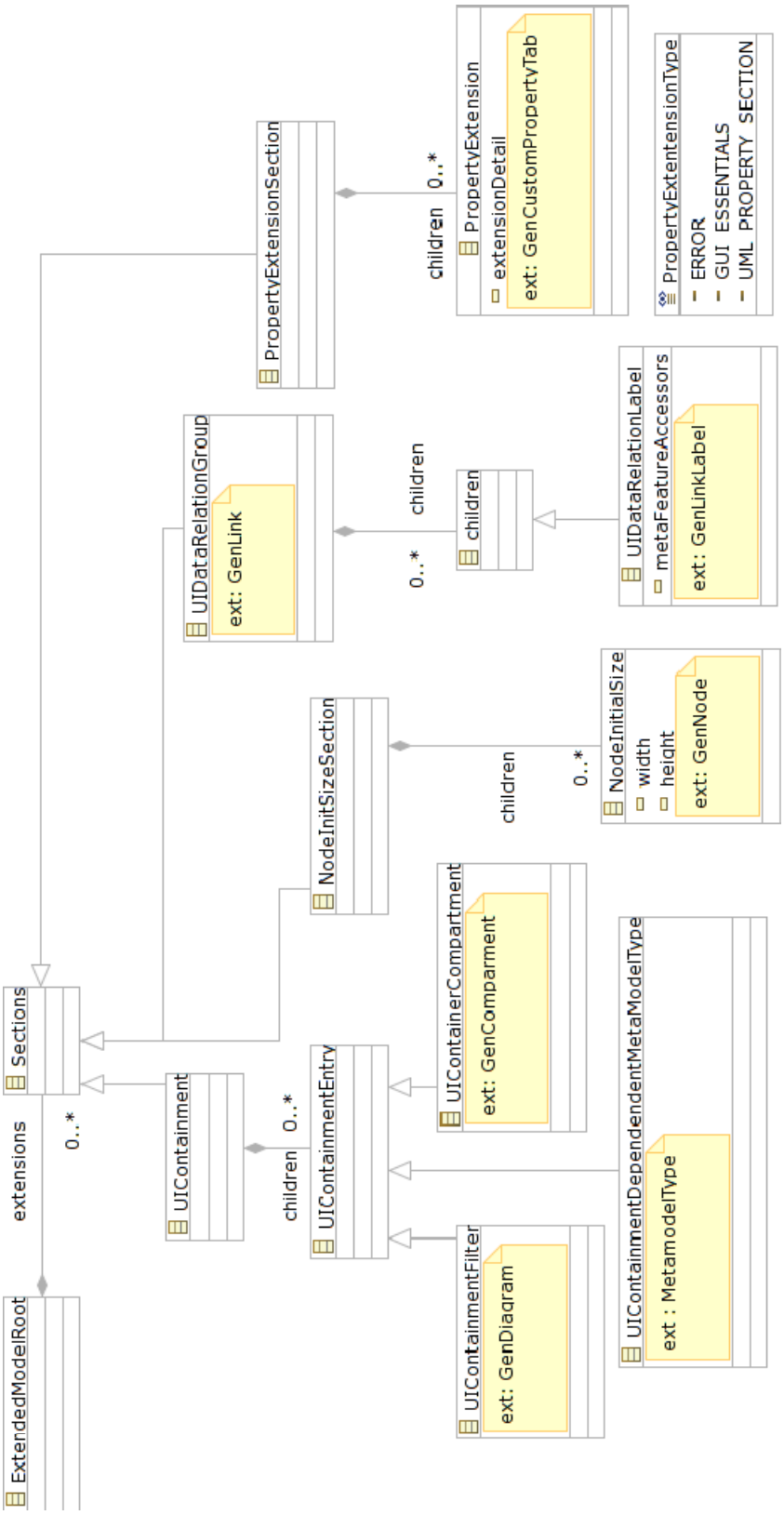

Abbildung 28 - guiGenModel

ExtendedModelRoot ist das Wurzelelement, welches in jedem Modell zu finden ist. Es beinhaltet eine beliebige Anzahl an Sektionen (Sections).

Die mittig abgebildete NodeInitSizeSection gruppiert NodeInitalSize. Mit ihren Instanzen wird ein GenNode, also ein im GE liegender Knoten, ausgezeichnet, um eine kleinere minimale initiale Größe zu ermöglichen.

PropertyExtension verweist auf ein GenCustomPropertyTab, also auf einen selbst hinzugefügten Reiter im späteren „Property View“ des Editors. Mittels einer Aufzählung kann eine Instanz den gültigen Zustand GUI_ESSENTIALS oder UML_PROPERTY_SECTION annehmen. Diese Metaklasse trägt der Anforderung Rechnung, dass die Stereotypen mittels des „Property View“ des GE editierbar sind. Der standardmäßig erzeugte „Property View“ zeigt nur die Eigenschaften der dahinter liegenden Instanz der Metaklasse. UML_PROPERTY_SECTION macht dynamisch alle Eigenschaften der gewählten Instanz verfügbar, sowie dynamisch alle Eigenschaften der auf sie angewendeten Stereotype. GUI_ESSENTIALS schränkt UML_PROPERTY_SECTION sowohl auf Stereotype des GUI UML Profils als auch auf den Namen des UML Elements ein.

UIDataRelationGroup zeichnet den GenLink aus, der für die Verbindung UIDataRelation verantwortlich ist. Das von ihm gehaltene UIDataRelationLabel ist die Beschriftung der Data Relation.

UIContainment gruppiert die Elemente, die mit dem Stereotyp UIContainmentAssociation in diesem Metamodell zusammenhängen. Gemäß Abbildungsvorschriften des UML Profils werden Kinder eines Elements durch eine stereotypisierte Assoziation beschrieben. Um diese initial zu setzen, ist eine Metaklasse des zweiten Metamodells verantwortlich. Danach müssen Elemente anhand dieser bestimmten ausgehenden Assoziationen referenzierte Elemente als Kinder anzeigen. Dieses Konzept wird von UIContainerCompartment repräsentiert. Zuletzt müssen, mit-

tels dieser Assoziation, beinhaltete Elemente aus dem eigentlich haltenden Package bzw. Model herausgefiltert werden. Dieses wird im Generatormodell mit UIContainmentFilter ausgezeichnet. Um Instanzen später korrekt zu verschieben, zeichnet UIContainmentDependendentMetaModelType ihren MetamodelType aus.

8.5.2 guiUMLGenModel

Das guiUMLGenModel befindet sich in der Datei „ExtendedUMLGenModel.ecore“. Die aus dem guiGenModel bekannten Metaklassen für das Wurzelelement und für die abstrakte Gruppe werden nicht dargestellt, um das Diagramm einfacher zu halten. Die konkreten Gruppen sind somit am Rand des Diagramms zu finden. Die Metaklasse StereotypeElementType referenziert einen Stereotyps des Profils. Durch die Metaklasse StereotypeAttribute werden seine relevanten Attribute nochmals modelliert, da eine bestimmte Property bei geladenem UML Metamodell schwer auffindbar ist. Die beiden beschrieben Metaklassen sind die zentralen Elemente des guiUMLGenModel und in folgender Abbildung in der Mitte zu finden.

Abbildung 29 zeigt das Metamodell des guiUMLGenModels.

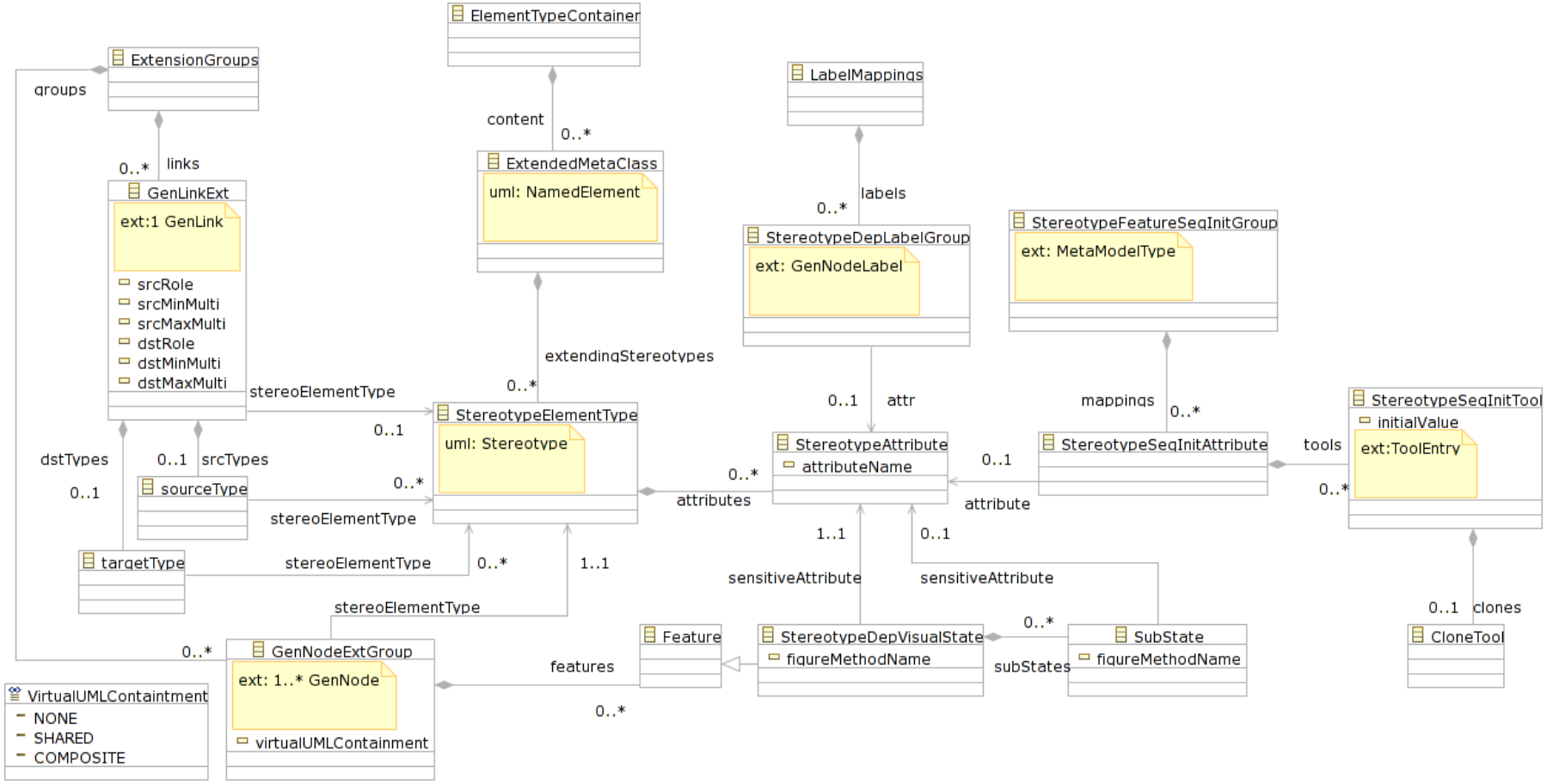

Abbildung 29 – Metamodell des guiUMLGenModels

Die zentral gelegenen Elemente ExtendedMetaClass, StereotypeElementType und StereotypeAttribute bilden die Struktur eines Stereotyps nach. In ExtendedMetaClass ist ein Verweis auf die, von ihm erweiterte, Metaklasse des UML Metamodells vorgesehen, in StereotypeElementType einer auf einen Stereotyp eines Profils und in StereotypeAttribute wird der Name eines Attributs des Stereotyps eingetragen.

GenNodeExtGroup gruppiert alle Knoten, die von einem bestimmten Stereotyp abhängen. Die Aufzählung virtualUMLContainment gibt an, ob die ihn referenzierende Assoziation von UIContainmentAssociation eine normale Assoziation, also NONE, eine Aggregation, also SHARED, oder eine Komposition, also COMPOSITE, ist. Um das Aussehen des Knotens in Abhängigkeit zu seinen Attributen zu setzen, gibt es StereotypeDepVisualState sowie SubState. StereotypeDepVisualStat“ zeichnet das Attribut höchster Ordnung, welches das Aussehen des Knotens beim Erzeugen bestimmt, aus. Mittels SubState wird die Änderung des editMode beschrieben. Beide Modellelemente halten den Name der Methode der Figur, welche bei einer Zustandsänderung mit dem neuen Zustand als Argument aufzurufen ist.

StereotypeDepLabelGroup zeichnen die Beschriftungen aus, welche nicht mehr ein Attribut der UML Metaklasse, sondern das eines Stereotyps darstellen.

StereotypeSeqInitAttribute dient als Gruppierung, um Werkzeugen initiale Wertzuweisungen eines Attributs zu ermöglichen. Es umgeht die GMF Einschränkung, ein Element nur durch ein Werkzeug erzeugen zu können. Das neue Werkzeug wird in einer Instanz von StereotypeSeqInitTool referenziert, der initiale Wert in initialValue festgelegt und das im Mapping-Modell vorgesehene Werkzeug zur Elementerzeu-

gung in CloneTool referenziert. StereotypeFeatureSeqInitGroup ist aus Implementierungsgründen notwendig.

Mittels einer GenLinkExt Instanz ist eine GMF-Verbindung, also eine UML-Assoziation, detailliert beschreibbar. Die Attribute sind selbsterklärend. Der referenzierte Stereotyp wird auf die Assoziation angewendet. sourceType Instanzen geben eine Stereotypmenge an, von denen ein Element auf die Quelle angewendet sein muss, damit die Verbindung erstellt oder zu ihm verschoben werden kann. Vergleichbares gilt bei targetType für die Senke.

9 Implementierung

9.1 Einleitung

In diesem Kapitel werden Problemlösungen auf niedrigem Abstraktionsniveau vorgestellt, auf die auf der CD befindlichen Quellen wird verwiesen und ihr Verständnis erleichtert. Der Autor empfiehlt begleitend zu diesem Kapitel die Quellen zu sichten.

9.1.1 Plattform

Folgende Software wurde zum Erstellen der Arbeit verwendet. Sie ist unter anderem von EMF und GEF abhängig, welche vom Eclipse Aktualisierungsmechanismus automatisch installiert werden. Es sollte das Standard Development Kit (SDK) von den Projekt eigenen „Update Sites“ installiert werden. Die exakte Version bietet sich insbesondere an, um die projekteigenen GMF Templates zu vergleichen.

- Eclipse Classic Version: 3.3.1.1, Build id: M20071023-1652
- GMF SDK 2.0.1
- UML2 2.1.1

Der Autor empfiehlt zudem folgende Werkzeuge beliebiger Version zu verwenden:

- einen auf Eclipse basierten UML Editor, wie den kostenpflichtigen MagicDraw, den freien GEF basierten TopcasedUML oder die experimentellen GMF basierten UML2 Tools. Der Autor verwendete letzt genannten aufgrund der technischen Nähe.
- EMFCompare um Modelle mittels TreeView vergleichen zu können

9.1.2 Struktur der Quellen

Die Quellen sind in vier verschiedenen PlugIns gegliedert.

`de.oio.mdsd.tools.guiModel.guiProfile` beinhaltet das erstellte Profil.

de.oio.mdsd.tools.guiModel.guiModeler.Dev beinhaltet die zur Entwicklung benötigten Modelle und die zugehörigen Templates.

de.oio.mdsd.tools.guiModel.guiModeler.lib beinhaltet Bibliotheken, welche vom Generat benutzt werden.

de.oio.mdsd.tools.guiModel.guiModeler beinhaltet den generierten Editor. Obwohl er das Endprodukt darstellt, wird er vollständig unter Zuhilfenahme der Bibliotheken durch Modelle und zugehörige Templates beschrieben.

9.1.3 Profil-Implementierung

Die Implementierung des Profils befindet sich in dem PlugIn de.oio.mdsd.tools.guiModel.guiProfile. Das Profil wird wie in seiner konzeptionellen Lösung abgebildet, modelliert und anschließend definiert. Im Plugin befindet sich zudem eine Hilfsklasse, welche den Uniform Resource Identifier (URI) des Profils, seinen vollqualifizierten Namen sowie direkt eine Instanz der Java Klasse Profile bereitstellt. In der plugin.xml wird eine Abbildung der URI des Profils (http://www.oio.de/guiProfile/1.0.1) auf die plattformrelative URI (platform:/.../GUI_OiO.profile.uml) vorgenommen.

9.1.4 Implementierung der Notationselemente

Die von Hand implementierten Notationselemente finden sich im Packet de.oio.mdsd.tools.guiModel.guiModeler.figures des Bibliotheks-PlugIns. Es werden drei Arten ihrer Implementierung unterschieden, ohne sich gegenseitig auszuschließen:

- Bestehenden Figuren werden verwendet und mittels Variablen konfiguriert.
- Eine bestimmte bestehende Figur wird mittels eines Bildes konfiguriert.
- Eine bestehende Figur wird erweitert, eine Methode, die ein

Graphics-Objekt als Argument trägt überschrieben und dieses zum Zeichnen verwendet.

9.1.5 Konfiguration mittels Variablen

Die erste Art stellt den Normalfall da und wird von jeder Figur verwendet. Es besteht kein weiterer Klärungsbedarf. Die Klasse `UIPrimitiveDataControlTextfield` bietet sich als einfaches, die Klasse `UIContainerView` als komplexes Beispiel an.

9.1.6 Konfiguration mittels eines Bildes

Die Klasse `ScalableImageFigure` nimmt ein Bild im Konstruktor entgegen. Pixelbasierte Bildformate wie Portable Network Graphics (PNG) eignen sich aufgrund ihrer konzeptionellen Basis schlecht zum Skalieren. Deshalb unterstützt die Klassen auch skalierbare Vektorgraphiken (SVG). Sie bieten sich an einfach Notationselemente zu erstellen. Wird ein pixelbasiertes Bildformat, wie der Screenshot eines Knopfs, mittels Inkscape in ausreichendem Detailgrad vektorisiert und als SVG im Konstruktor angegeben, ist eine hohe CPU Last zu beobachten. Vektorisierte Graphiken eignen sich also nur eingeschränkt in einem GE und wurden nicht verwendet. Um die Vektoranzahl gering zu halten, wurden eigene SVGs mittels Inkscape erstellt. Hierbei ist zu beobachten, dass die SVG Bibliothek von GMF grundlegende Funktionen wie Farbverläufe nicht unterstützt. Deshalb wurde nur in der Symbolleiste des „View“ in der Klasse `UIContainerView` eine Vektorgraphik eingesetzt.

9.1.7 Figuren mittels „Graphics“

Wie bei den vorgefertigten Figuren ist es dem Programmierer möglich das `Graphics`-Objekt, welches vergleichbar mit dem Graphical Context (GC) des Standard Widget Toolkit (SWT) ist, zu verwenden, um Linien und Füllungen selbst zu zeichnen. Dies wird verwendet um die Tabelle, die Liste, den Baum, den Picklist- Knopf und jeglichen Farbverlauf zu

zeichnen. Dies stellt ein ansprechendes Aussehen beim Skalieren sicher. Die Klasse `UIComplexDataControlTableList` bietet sich als Beispiel an. Sie implementiert die Tabelle sowie die Liste als einspaltige Tabelle. Ein komplexeres Beispiel ist der Baum in `UIComplexDataControlTree`, welcher Schriftarten passend zur aktuellen Größe erzeugt.

9.1.8 Zustandsabhängige Figuren

Die vom GE verwendeten Figuren müssen je nach Zustand des Domänenobjekts ihr Aussehen ändern. Diese Anforderung wird erklärt, soweit sie die Figuren betreffen. Für jede Gruppe wird eine Figur erstellt, diese sind `UIPrimitiveDataControl`, `UIComplexDataControl` sowie `UIContainer`. Die Figuren sind unsichtbar und beinhalten eine Beschriftung und der `UIContainer` zusätzlich ein Rechteck. Somit bleiben die Referenzen der `EditParts` für das Element, für die Beschriftung und beim `UIContainer` für das Compartment unverändert. Jede Gruppenfigur besitzt Methoden, die bei Zustandsänderungen gerufen werden und einen Schalter beinhalten. Dieser tauscht bei einer Zustandsänderung höchster Ordnung die von der unsichtbaren Figur gehaltene sichtbare Figur entsprechend aus. Der neuen sichtbaren Figur wird die Beschriftung und ggf. das Rechteck übergeben, damit sie diese korrekt positionieren kann.

9.2 Graphischer Editor

Das PlugIn des GE wird vollständig generiert. Seine Implementierung bedeutet folglich das Generatormodell sowie die Templates so anzupassen, dass das Generat ggf. unter Zuhilfenahme von Bibliotheksfunktionen den Anforderungen entspricht. Wie für die eigenen Metamodelle gilt hier, dass Anpassungen eine Abweichung vom GMF Standard in einem speziellen Punkt bedeuten. Sie müssen in der Mehrzahl der Fälle einzeln betrachtet werden. Anpassungen werden soweit beschrieben, dass der Leser sie mittels der Grundlagenkapitel EMF, GEF und GMF einordnen

kann. Da sehr spezifisch Stellen geändert werden, erfordert dieses Kapitel ein hohes Maß an Verständnis der Grundlagen. Zudem existiert keine Dokumentation zur Struktur des Standardgenerats von GMF, sie wird im Weiteren auch nur partiell für das jeweilige Problem erklärt. Dieses Kapitel kann als Problemlösung auf niedrigem Abstraktionsniveau oder als Anleitung zu den Quellen gelesen werden. Bei letztgenannten experimentiert man im Zweifelsfall zuerst mit den Quellen, nimmt die Auszeichnungen aus dem Generatormodell und kommentiert Teile der Templates aus. Den Debugger sollte man nur wohlüberlegt verwenden.

9.2.1 Anpassungen am erweiterungsmodellfreien Generatormodell

Das Generatormodell wird an zahlreichen Stellen angepasst. Herauszustellen ist, dass mit `dynamicTemplates` eigene Templates aktiviert und mittels `templateDirectory` ihr Pfad angegeben wird. Später eigens behandelt werden zwei zusätzliche `Custom Property Tab`. Die Dateiendung des Notationsmodells, der Präfix aller Pakete, das Copywrite, usw. werden geändert. Um eine Liste der über 60 Änderungen, welche größtenteils Namensänderungen der generierten Klassen sind, zu erhalten bietet sich folgendes Vorgehen an:

- Aus dem Mapping-Modell ist ein neues Generatormodell zu erstellen, um die Erweiterungsmodelle zu entfernen.
- Aus dem Mapping-Modell ist ein neues Generatormodell unter anderem Namen zu erstellen.
- Bei installiertem EMFCompare, fügt man unter Window, Preferences, General, Content Types, Model File die Endung `*.gmfgen` hinzu.
- Man vergleicht beide Modelle mittels der Eclipse „Compare With“ „Each Other“.

9.3 Templates

9.3.1 Einleitung

Die aufgeführten Templates sind in der Mehrzahl der Fälle abhängig von Auszeichnungen. Ihre Bedeutung wurde bereits bei den eigenen Metamodellen (s.S. 66) geklärt. Sie sind nicht zum Verständnis der nun aufgeführten Probleme und ihrer Lösungen notwendig und werden nicht explizit herausgestellt. Jede Anpassung wird gegliedert in Aufgabe (**A**), beteiligtes Template (**T**) und Lösung (**L**). Die Templates sind im Entwicklungsprojekt `de.oio.mdsd.tools.guiModel.guiModeler.Dev` zu finden. Die Pfadangabe A kennzeichnet einen Aspekt unter `/templates/aspects/xpt`, die Pfadangabe T ein normales Template unter `/templates/xpt`. Ggf. befindet sich unter einer Überschrift eine Erklärung, um die folgenden Probleme einzuordnen. Wird anhand der Quellen vorgegangen, ist das PlugIn `org.eclipse.gmf.codegen` zu importieren, welches die Standardtemplates enthält und diese mit denen der Quellen auf Dateien oder Ordner Basis zu vergleichen. Der Anhang C – Weitere TemplateanpassungenS. XVII vervollständigt die Beschreibung der Template-Anpassungen.

9.3.2 Fehlermeldungen

A: Fehler müssen im lib-PlugIn protokollierbar sein. Das hierfür verantwortliche `GUI_OIO_DiagramEditorPlugin` muss dem lib-PlugIn bekannt gemacht werden. Die Open Services Gateway initiative (OSGI) Struktur kann aufgrund einer sonst entstehende zyklischen Abhängigkeit nicht verwendet werden.

T: T:`/plugin/Activator.xpt`

L: `GUI_OIO_DiagramEditorPlugin` implementiert ein im lib-Plugin definiertes `LogInterface`. Beim aktivieren des PlugIns übergibt er der statischen Klasse `LogHelper` eine Referenz auf sich selbst. `LogHelper` im-

plementiert das gleiche Interface und delegiert die Aufrufe (Delegate [Gam95]).

9.3.3 Profilanwendung beim Laden

A: Wird ein neues Modell erstellt, ist das Profil direkt auf es anzuwenden

T: `T/editor/DiagramEditorUtil.xpt`

L: Direkt nach erstellen des Wurzelements wird mit Hilfe des `GuiProfileHelper` das Profil auf es angewendet.

A: Beim laden oder initialisieren eines Modells muss auf Profilanwendung geprüft werden

T: `A/editor/Editor.xpt`

L: Die Methode `initializeGraphicalViewerContents` wird überschrieben, in ihr auf Profilanwendung geprüft, ggf. ein Dialog angezeigt und bei Bestätigung das Profil angewendet.

9.3.4 Property Tab

A: Im Property Tab müssen Stereotype der Elemente bearbeitbar sein

T: `A:/propsheet/PropertySection.xpt`

L: Die generierten Property Tabs nutzen, für alle auf das Element angewendete Stereotypen, `ItemProvider` und `IItemPropertySource` der dynamisch erzeugten Stereotype um sie verfügbar zu machen. Diese Klassen sind unabhängig von JFace, aber ihre Konzepte sind ähnlich. Beispielsweise ist `IItemPropertySource` mit `IPropertySource` vergleichbar. Sie sind gut in [EMF04] bzw. [Bud03] dokumentiert. Die `GUIPropertySection` prüft zudem den qualifizierten Namen des Pakets

des Stereotyps und zeigt aus UML nur den Name des NamedElements an.

9.3.5 Stereotypabhängige Beschriftungen

Die bisher in im Mapping-Modell definierten Beschriftungen (Labels) sind von einer Eigenschaft einer UML Metaklasse, nicht von einer des Profils abhängig. Um die zugehörigen Schritte zu verstehen, besteht Klärungsbedarf. Die aus GEF bekannten und für GMF übernommenen EditParts besitzen, als Controller, eine Referenz auf ein Modellelement. Sie geben ihre statischen Kinder anhand der Kinder des Modellelements zurück, für welche später eigene EditParts erstellt werden. Um die Lage, Größe, usw. eines Elements unabhängig vom Domänenmodell zu persistieren, benutzt GMF das Notationsmetamodell. Instanzen seiner Metaklassen sind Modellelemente, die automatisch bearbeitet werden. GMF schaltet zwischen Controllern und Domänenmodell, in unserem Fall ein UML Modell, das Notationsmodell. Die EditParts besitzen also eine Referenz auf eine Instanz einer Metaklasse des Notationsmodells, welche eine Referenz auf eine Instanz eines Domänenmodellelements besitzt. Die hierfür notwendige Metaklasse des Notationsmodells heißt View. Eine Instanz von View ist also für die EditParts, die Controller, das Modell. View bezeichnet in GMF das Modell des MVC Entwurfsmusters. Folglich werden in GMF für Kinder des Views weitere EditParts erzeugt. Ein View ist nun existenzabhängig von dem referenzierenden Objekt, ein EditPart existenzabhängig vom Modell, also von einem View. Wird ein neues Domänenelement erzeugt, wird sein Typ bestimmt, anhand des Typs ein bestimmter View erzeugt, und anhand des Views und seiner Kinder die EditParts. Sind nun weitere Notationselemente mit Verhalten, also weitere EditParts, für das gleiche Domänenelement zu erzeugen, müssen beim Erzeugen des View weitere bestimmte Views als Kinder hinzugefügt werden. Ein bestimmter View

kennt seinen Typ, welcher erforderlich ist, um den passenden EditPart zu erzeugen. Ist der Kind-EditPart eine editierbare Beschriftung, besitzt er zudem einen Parser für die Eingabe. Dieser wird über einen Service bestimmt. Die Parser werden für eine bestimmte Metaklasse u.a. mit den Attributen bzw. Features konfiguriert, auf denen sie operieren. Das bisher beschriebene Verhalten wird standardmäßig generiert und ist für stereotypabhängige Beschriftungen anzupassen. Dies wird für Knotenbeschriftungen beschrieben und ist auf Kantenbeschriftungen übertragbar.

A: Nicht das gleiche Element, sondern eine Instanz des bestimmten Stereotyps ist dem EditPart des Labels zu übergeben.

T: T/diagram/views/Utils.xpt

L: Die Standard View-Erzeugung für das Kind-Label wird bei einer Auszeichnung mittels StereotypeDepLabelGroup bzw. UIDataRelationLabel durch eine eigene ersetzt. Diese holt sich vorm Erzeugen den bestimmten Stereotyp und setzt ihn statt des gleichen Domänenelements im View-Konstruktor ein.

A: Die Label-View-Erkennung ist bei unterschiedlichen Domänenelementen des Container und des Kindes sicherzustellen.

T: A/providers/ViewProvider.xpt

L: Die Prüfung ob das Domänenelement des Containers und des Kindes gleich sind wird im UMLViewProvider vorgenommen und auszeichnungsabhängig entfernt.

A: Der Parser wird mit den gewünschten Attributen des Stereotyps konfiguriert.

T: T/providers/ParserProvider.xpt

L: Der Elementtyp wird bis zur Parser-Fabrikmethode durchgereicht, welche reflexiv das Feature des dynamisch erzeugten Objects bestimmt und den Parser damit konfiguriert.

A: Korrektes funktionieren des Parsers sicherstellen

T: `T/diagram/editparts/TextAware.xpt`

L: Es zeigt sich, das der Parser-Service einen Parser einmalig für ein geladenes EditorPlugIn erstellt. Weiterhin zeigt sich, dass dynamisch erzeugte Features aus dem gleichen Metamodellelement in unterschiedlichen Laufzeit-Instanzen, also in mehreren Editor-Instanzen, nicht identisch sind. Die Anpassung verhindert das ausgezeichnete `EditParts` den Parser-Service verwenden und stellt sicher, dass die Parser nicht nur in der ersten Instanz des Editors funktionieren.

9.3.6 Komfortwerkzeuge

Unter Komfortwerkzeuge fallen Werkzeuge der Toolbar, die die gleichen Elemente wie ein anderes Werkzeug erstellen, jedoch mit unterschiedlicher initialer Wertzuweisung. Allgemein erstellt ein Werkzeug oft mehrere Elemente, jedoch nicht zum gleichen Zeitpunkt, da der Element-Typ von GMF-Tooling verwendet wird, um ein Element abhängig von seinem Container zu beschreiben. Beispielsweise besitzt der `View` eines primitiven Elements des Domänenmodells einen anderen Typ, je nachdem, ob er von dem `View` des Modells oder von dem `View` eines Container-Elements gehalten wird. Da bei Objekterzeugung durch ein Werkzeug das Werkzeug an der Cursorposition, und nicht der `DiagramUpdater`, den `View` erzeugt, muss der Typ des `Views` berücksichtigt werden, also mehrere Elemente mittels eines Wertzeugs erzeugbar sein. Um ein Komfortwerkzeug zu erstellen, muss es die gleichen erzeugbaren Elemente besitzen wie ein anderes. Werkzeuge folgen den Request-Command-Muster, schicken also zur Elementerzeugung einen `Request`

an den an aktueller Cursorposition liegenden (zukünftigen) Container, welcher diesen mit einem Command beantwortet, der von dem Werkzeug ausgeführt wird. Jeder Request besitzt eine Map, um zusätzliche Parameter aufzunehmen. Um den gewünschten initialen Wert bei der Objekterzeugung durch die zurückgelieferten Commands berücksichtbar zu machen, muss er mittels eines geeigneten Schlüssels dem Request mitgegeben werden. Der von der SemanticEditPolicy befragte EditHelper ist die dafür geeignetste Stelle diesen Schlüssel zu berücksichtigen.

A: korrekt das Werkzeug zu belegen

T: T/editor/palette/PaletteFactory.xpt

L: Ein eigenes Tool wird generiert, das die Elementtypen des referenzierte Basiswerkzeug kopiert. Aus dem vollqualifizierten Namen des Stereotyps und dem Namen des Attributs wird der Schlüssel erzeugt und der initiale Wert gesetzt.

A: der EditHelper muss die Requests berücksichtigen

T: A/diagram/edithelpers/EditHelper.xpt

L: Der EditHelper sucht nach bekannten Schlüsseln und erzeugt ggf. einen Command zur Wertzuweisung.

9.3.7 Stereotypisierte Knoten

Knoten, also Instanzen der UML Metaklasse Class, weisen im Zusammenhang mit dem GE einige Besonderheiten auf. Stereotype müssen mit dem Knoten erstellt und gelöscht werden und der Typ des Knoten wird anhand von ihnen identifiziert. Die vorgestellten Änderungen sind auf Kanten übertragbar, später werden nur ihre zusätzlichen Änderungen aufgezeigt.

A: Der bestimmte Stereotyp ist beim erstellen der Instanz auf sie anzuwenden.

T: `T/diagram/commands/CreateNodeCommand.xpt`

L: Beim ausführen des bestimmten `CreateCommands` wird der Stereotyp auf die Instanz angewendet.

A: Typunterscheidung anhand des Stereotypes.

T: `A/editor/VisualIDRegistry.xpt`

L: In die zum typisieren verantwortliche `VisualIDRegistry` werden passende Stereotyp Abfragen generiert.

A: Der Stereotyp gelöschter Elemente darf nicht in der XMI serialisierten Form auftauchen.

T: `A/diagram/editpolicies/NodeItemSemanticEditPolicy.xpt`

L: Die zugehörige `EditPolicy` fügt den auf Löschanfrage gelieferten `Command` eine Instanz der lib-Klasse `DestroyNamedElementCommand` hinzu.

9.3.8 Attributsabhängiges Figure

A: Knoten müssen zustandsabhängig ihr Aussehen anpassen

T: `A/diagram/editparts/NodeEditPart.xpt`

L: Beim Aktivieren des `EditParts` wird ein `NotificationListener` bzw. ein Observer [Gam95] am Domänenelement registriert. Anhand des geänderten Attributnamens ruft er die passende Methode der Gruppenfigur. Um die Figur beim Laden in den korrekten Zustand zu setzen, werden die relevanten Attribute abgefragt, bevor die Figur initial erstellt wird.

9.3.9 Kanten

Um die Verwendung von Kanten, also Verbindungen bzw. UML Assoziationen, zu ermöglichen, müssen weitreichende Änderungen am Standardverhalten vorgenommen werden. Zuerst ist zu prüfen, ob die Verbindung erstellt werden kann, also ob die Verbindungsenden den Typen der Anforderungen entsprechen. Die Assoziation ist danach korrekt zu erstellen und muss neu ausrichtbar sein. Zuletzt ist sicherzustellen, dass Assoziationen bei externen Änderungen aktualisiert werden.

A: Prüfen, ob die Verbindungsenden einer Assoziation den angegeben Verbindungstypen entsprechen. Referenzen, dessen Quelle und Senke identisch sind, müssen verhindert werden.

T: `T/diagram/commands/CreateLinkUtils.xpt`

L: In die `canCreateElement`-Methode wird auf die im Erweiterungsmodell angegeben gültigen Quell- und Senkestereotypen geprüft.

A: Die Assoziation muss korrekt erstellt werden

T: `A/diagram/commands/CreateLinkCommand.xpt`

L: Die Assoziation wird mittels einer UML Hilfsmethode erzeugt und unter Berücksichtigung des Erweiterungsmodells erzeugt.

A: Änderung der Assoziationsenden ermöglichen

T: `T/diagram/commands/CreateLinkUtils.xpt`

L: Die Methoden um die neue Quelle bzw. Senke zu prüfen folgen dem gleichen Muster wie bei der Assoziationserstellung. Hervorzuheben ist, dass die Methoden um die Assoziationenden zu ändern den Typ des Assoziationendes ändern und falls es nicht von der Assoziation gehalten wird es auch verschieben.

A: Nicht im Notationsmodell vorhandene Assoziationen, wie beim Initialisieren des Diagramms aus einem UML Modell, müssen erstellt werden

T: `A/diagram/updater/DiagramUpdater.xpt`

L: Um auf externe Änderungen am Modell zu reagieren, ist der `DiagramUpdater` verantwortlich. Die für Assoziationen verantwortliche Methode erkennt Quelle und Senke an dem Navigierbarkeits-Attribut der Assoziationsenden.

9.3.10 Spezielle UIDataRelation-Kante

Die `UIDataRelation` emuliert eine in der abstrakten Syntax unmögliche Verbindung zwischen einer `Property` und einer Klasse. Hierfür ist erforderlich, das der Name des Assoziationsendes synchron mit dem Name der `Property` ist und das die Assoziation gelöscht wird, sobald die `Property` gelöscht wird. Die Abweichungen zu den in Kanten beschriebenen Änderungen sind marginal und lassen sich prinzipiell auf das Setzen des Namens des Assoziationsendes und dem Auflösen des Assoziationsziels in `DiagramUpdater` reduzieren. Sie befinden sich in den bereits unter Kanten aufgelisteten Templates.

A: Der Name des Assoziationsendes und der `Property` sind synchron zu halten, wird die `Property` gelöscht muss auch die Assoziation gelöscht werden.

T: `A/diagram/editparts/LinkEditPart.xpt`

L: Es werden drei Listener erzeugt. Der erste hält die `Property` mit dem Assoziationsende synchron. Der zweite hört auf Änderungen der Klasse, um beim löschen der bestimmten `Property` die Assoziation zu löschen. Der dritte hört auf das Assoziationsende, wird es nicht von dem ersten Listener geändert, wurde die Assoziation umgesetzt und die o.g. Listener werden entfernt und neu registriert.

9.3.11 UIContainmentAssociation

Gemäß den Abbildungsvorschriften der abstrakten Syntax werden beinhaltende Elemente vom Container nicht gehalten, sondern diese Beziehung mittels der UIContainmentAssociation, ausgezeichnet. Diese Anforderung zieht tiefgreifende Änderungen nach sich. Zunächst muss jedes Kind im Package erzeugt und eine UIContainmentAssociation vom virtuellen Container zum Kind erstellt werden. Danach darf das Package diese virtuellen Kinder nicht anzeigen, jedoch muss es der virtuelle Container. Werden die Kinder korrekt dargestellt, gilt es sie korrekt zu verschieben und zyklische UIContainmentAssociation Graphen sowie eine „race condition“ zu berücksichtigen.

A: Kinder sind im Package zu erstellen und sind eine Senke einer UIContainmentAssociation

T: T/diagram/commands/CreateNodeCommand.xpt

L: s.o., beachten Sie speziell createStereotypedElement(StereotypeElementType ext) FOR gmfgen::GenChildNode

A: Virtuelle Kinder aus dem Package filtern und sie ihren virtuellen Containern zuweisen

T: A/diagram/updater/DiagramUpdater.xpt

L: Der DiagramUpdater wird generativ angepasst. Der Liste der Kindern des Package werden jene nicht hinzugefügt, welche von einer UIContainmentAssociation referenziert werden. Gleichfalls wird der Liste der Kinder der virtuellen Containern, alle der von ihm über eine UIContainmentAssociation referenzierten Elemente hinzugefügt.

A: Beim verschieben von Elementen müssen die UIContainmentAssociation angepasst, ggf. hinzugefügt oder gelöscht werden.

T: A/diagram/edithelpers/EditHelper.xpt

L: In dem für alle Klassen verwendeten bestimmten EditHelper wird bei einer Verschiebungsanfrage der lib-Command GUI_UML_ContainmentCommand zurückgegeben. Dieser prüft statisch auf UIContainmentAssociation und nimmt die erforderlichen Änderungen vor und berücksichtigt dabei neuen wie alten Container. Konzeptionell besser, sind individuelle HelperAdvice zu benutzen. Sie wurden aufgrund von Einschränkungen von GMF-Tooling in Bezug mit der ElementTypeRegistry nicht verwendet.

A: Seit dem UIContainmentAssociation verwendet wird, entstehen für jede Verknüpfung von einem Kind zu einem anderen zwei. Der Fehler ist zu beseitigen.

T: T/diagram/editparts/NodeEditPart.xpt

L: Es zeigt sich, das, vor dem Werkzeug der DiagramUpdater einen View erstellt. Es ist auf die CanonicalEditPolicy zurückzuführen, welche Notationsmodellelemente und Domänenmodellelemente synchron hält. Diese wird für die Container der an der Verbindung teilnehmenden Elemente kurzzeitig deaktiviert, indem sie eine Liste der zu deaktivierenden Elemente zurückgeben. In der Standardimplementierung von GraphicalEditPart wird der Views des Notations-Containers, nicht jedoch der View des Domänen-Containers zurückgegeben. Die EditParts werden so angepasst, dass sie zusätzlich den View ihres Package zurückgeben.

A: Entsteht durch ein externes Werkzeug ein zyklischer UIContainmentAssociation Graph sind beteiligte Elemente nicht anzuzeigen.

T: A/diagram/editparts/CompartmentEditPart.xpt sowie A/diagram/editparts/Common.xpt

L: Diagram- sowie CompartmentEditParts wird eine EditPolicy hinzugefügt, die verhindert, das Kinder die über UIContainmentAssociation ihre Eltern beinhalten angezeigt werden.

9.3.12 Bekannte Fehler

Es existieren drei bekannte Fehler des GE

9.3.12.1 Elementverschiebung

Zum aktuellen Stand ist es, fachlich korrekt, zwar nicht möglich UIDataSets in UIContainern zu erzeugen, jedoch erstellte UIDataSets in sie zu verschieben. Zur Fehlerbehebung ist eine EditPolicy zu erstellen und in der createChangeConstraintCommand-Methode die von GMF-Tooling generierten Schalter zu nutzen, um ggf. einen nicht ausführbaren Command zurückgeben. Es ist darauf hinzuweisen, dass die generierten Schalter einen View als Argument erwarten. Somit kann an vorgesehener Stelle, den EditHelpern, keine Prüfung erfolgen, da sie keine Kenntnis über das Notationsmodell besitzen.

9.3.12.2 Bei Editorwechsel verschwinden alle Elemente

Nicht reproduzierbar, d.h. ab und zu, verschwinden alle Elemente auf der Zeichenfläche, wenn z.B. vom UML2 Tools Editor zum eigenen Editor gewechselt wird. Die eigenen Fehlermeldungen lassen darauf schließen, dass kein UML Element mehr Stereotypen aufweist. Der Autor vermutet einen Fehler in den verwendeten PlugIns.

9.3.12.3 Race Condition

Im Zusammenhang mit nicht von der Assoziation gehaltenen Assoziationsenden tritt eine „race condition“ auf. GMF löscht Elemente parallel, mittels mehreren Threads. Wird eine Assoziation und das Assoziations-

ende haltende Element gleichzeitig gelöscht, versuchen beide beim Löschen das Assoziationsende zu entfernen. Es wurde davon ausgegangen, dass sobald bereits zum gleichen Zeitpunkt ein Zugriff auf das Element besteht, dieser zum Löschen dient. In Folge dessen, fängt die Klasse `GUI_UMLHelper` beim Löschen von Assoziationen `ConcurrentModificationException` und die Elemente werden korrekt gelöscht. Der Fehler tritt auf, sobald der Benutzer „undo“ anwendet. Die „fehlerhaft“ gelöschten Instanzen können nicht wieder hergestellt werden, unschöner weise jedoch die Instanzen der Notationselemente und ihre EditParts. Um den Fehler zu lösen ist zu klären ob GMF eine sequentielle `Command`-Ausführung ermöglicht. Ggf. müssen sich die `Commands` beim Ausführen auf die Assoziationsenden synchronisieren, oder die `Commands` müssen ihre Ausführung, undo und redo and einen exklusiven EMF `Command` delegieren.

10 Fazit

10.1 Kritik an den verwendeten Werkzeugen

Die folgende Kritik bezieht sich ausschließlich auf den von den Werkzeugen vorgesehenen Anwendungsbereich und nicht auf den speziellen Anwendungsfall dieser Arbeit.

10.1.1 EMF

Anlass zur Kritik bietet sich im Zusammenhang mit der Trennung zwischen Generat und Quellcode, mit dynamisch erzeugten Objekten, den Namensräumen und dem UML an.
Dynamisch erzeugte Objekte besitzen den Nachteil, dass sie, obwohl aus dem gleichen Modellelement erzeugt, nicht zwangsläufig in unterschiedlichen Laufzeit-Instanzen identisch sind. Zudem ist anhand der `toString()`-Methode schlecht ersichtlich, ob das dynamische Objekt eine Klasse oder eine Instanz ist.
Bei den Namensräumen fällt auf, dass das Meta-Modell nicht daran identifiziert wird, und somit das gleiche Meta-Modell unter einem anderen Pfad nicht als identisch erkannt wird.
Formal unvollständig wird beim UML Export der für Meta-Klassen vorgesehene Stereotyp `metaclass` nicht angewendet [Jec04] .
Insgesamt überzeugt EMF vor allem durch seine Unauffälligkeit in dieser Arbeit. Das Framework ist sehr stabil und die Basis zahlreicher Software. Beeindruckend ist zudem, dass sich ohne Aufwand ein dynamischer Baumeditor nutzen lässt um Instanzen beliebiger Ecore Meta-Klassen zu erstellen.

10.1.2 Eclipse UML2

Eclipse UML2 bietet kaum Anlass zur Kritik. Der Autor vermisste eine Methode, um Assoziationen neu zu verknüpfen, ohne dass man berücksichtigen muss, ob das assoziierte Element oder die Assoziation das As-

soziationsende hält. Zudem ist aus den Methodennamen `getStereotypeApplication` und `getAppliedStereotype` nicht zu schließen, welche den Stereotyp und welche eine Instanz des Stereotyps zurückgibt.

10.1.3 GMF-Runtime

An der GMF-Runtime ist deutlich zu sehen, dass sie historisch gewachsen ist. Der Vorteil hiervon ist, dass sie flexibel und stabil ist. Als Nachteil wird deutlich, dass sie dadurch noch komplexer wird. Die GMF-Runtime erfordert einen außerordentlich hohen Einarbeitungsaufwand.

10.1.4 GMF-Tooling

Die generative Architektur von GMF bietet die meisten Kritikpunkte.

Auffällig ist, dass sie auch im Generatormodell die erprobten Konzepte der GMF-Runtime nicht umsetzt. Der Autor vermutet, dass das auf die getrennten Entwicklerteams zurückzuführen ist.

Das Domänenmodell sollte hinter einem „Element Type Registry“ Modell dekoriert und nicht direkt in das Mapping-Modell eingebunden werden. Ggf. ließe sich dadurch auch bisher gänzlich fehlendes zustandsbasiertes Verhalten realisieren. Die gmfgen-Meta-Klasse `SpecializationType` zeigt, dass die ElementTypeRegistry unvollständig im Meta-Modell berücksichtigt wurde.

Die Entwicklung behindern die statische M2M Transformation, fehlende Auszeichnungsmöglichkeiten des Mapping-Modells, sowie Synchronisationsprobleme und fehlende Plattform Nähe des Generatormodells.

Die Auszeichnungsmöglichkeit des Generatormodells mittels eines Erweiterungsmodells ist eine der wenigen erkennbaren „Best Practices“ und sehr positiv zu werten, ebenso wie die Xpand Templatesprache. Xpand wurde jedoch angepasst und durch die Abweichungen entstehen eine Reihe von Defiziten wie unterschiedliche, undokumentierte Syntax, fehlende automatische Vervollständigung der Integrated Development

Environment (IDE), fehlende Erweiterbarkeit der M2M Transformationen, usw.

Zudem besitzt das Generat seine eigene, nicht intuitive Struktur, und es entsteht weiterer Aufwand zusätzlich zur GMF-Runtime die Struktur des Generats zu erlernen.

Es scheint unrealistisch später von einem generierten Editor zu einem handgeschrieben Editor umzusteigen.

Generell ist GMF insbesondere für kleine Meta-Modell-nahe Editoren gut geeignet. Das Generat bietet Anschauungsmaterial und anhand von Änderungen in den Modellen und ihren Auswirkungen auf das Generat lassen sich schnell Konzepte erlernen. Angesichts der aufgeführten architektonischen Defizite ist fraglich, ob diese in absehbarer Zeit bewältigt werden.

10.2 Kritische Wertung

10.2.1 Abstrakte Syntax

Es wurden zwei Defizite der abstrakten Syntax festgestellt. Zum einen ist es nicht korrekt alle primitiven Elemente wie z.B. eine Beschriftungen von `UIDataControl` erben zu lassen, da es eine Datenverbundenheit impliziert. Zum anderen sind die Gruppierungen zwar für den beschrieben Problemraum ausreichend, jedoch ist ein maßgeblicher Vorteil des Editors die Integration in UML. Wird versucht ein Element des Problemraums mit einem eines anderen Problemraums in einem UML Modell zu integrieren, werden die Defizite sichtbar. Ein konkreter Anwendungsfall könnte die Verhaltensmodellierung der UI Elemente sein. Hierfür wäre es z.B. notwendig einen Button von dem Stereotyp `Clickable` erben zu lassen. Nun wären auch Textfelder und Beschriftungen `Clickable`. Dies mittels OCL zu unterbinden, scheint dem Autor der falsche Ansatz zu sein.

10.2.2 Graphischer Editor

Der GE bietet ein domänenspezifisches UML Diagramm für GUI. Die gebotene Sicht ermöglicht Domänenexperten gültige UML Modelle zu erstellen, ohne sich bewusst zu sein, dass die UML überhaupt existiert. Durch die UML Basis gliedert er sich gut in die bestehende Werkzeugkette ein und eignet sich außerordentlich gut für die generative Entwicklung. Die UML Integration ermöglicht insbesondere stabile UML Werkzeuge für nicht domänenspezifische Aspekte zu verwenden. Aus der ausführlichen Behandlung in [Sta07] lässt sich der Schluss ziehen, dass für traditionelle domänenspezifische Sprachen oft UML ähnliche Diagrammtypen zu entwerfen sind. Der initial höherer Aufwand kann schnell amortisiert werden, wenn nicht weitere Diagrammtypen zu entwickeln sind. Durch die weite Unterstützung von UML stellt diese Basis eine Zukunftssicherheit her. Der Schluss gilt jedoch nicht für die komplette generative Architektur. Es ist zu erwarten, dass insbesondere die Generatoren den gleichen Wartungsaufwand aufweisen wie bei Domänen spezifischen Sprachen. Es ist jedoch anzunehmen, dass auch zukünftige Generatoren die UML unterstützen werden.

10.2.3 Ansatz

Der gewählte Ansatz ein domänenspezifisches UML Diagramm auf Basis von GMF zu entwickeln ist theoretisch korrekt. Praktisch ist aufgrund der Defizite der generativen GMF Architektur von ihrer Verwendung abzuraten und eine reine GMF Runtime Verwendung zu empfehlen. Die wesentlichen Probleme während dieser Arbeit lagen in den konzeptionellen Schwächen der generativen GMF Architektur und nicht in ihrer Anpassung auf UML.

10.3 Ausblicke

10.3.1 Abstrakte Syntax

Für die abstrakte Syntax gilt es die bekannten Defizite zu lösen, vorzugsweise einen Stereotyp für jedes Element zu erstellen, um ihn vor allem für Erweiterungen in bisher nicht berücksichtbarer Form flexibel zu halten.

Layout-Informationen im Modell würden das Automatisierungspotential steigern und die Akzeptanz bei Entwicklern erhöhen. Ihre Koordinaten sind zwar bereits dem GE bekannt, es ist jedoch zu erwarten, dass die Angabe von Layout-Managern einen höheren Nutzen bringt.

Zu Stereotypen sollten OCL Einschränkungen und Symbole für die UML Editoren hinzugefügt werden.

Die aktuelle Eclipse Version ermöglicht die Verwendung von statischen UML Profilen [Bru08] . Bei statischen Profilen wird aus dem Profil ein Ecore Modell erstellt und Java Repräsentationsklassen generiert. Somit sind die Meta-Klassen zur Compile-Zeit bekannt, was unter anderem für OCL Constraints und die Ausführungsgeschwindigkeit von Bedeutung ist.

10.3.2 Graphischer Editor

Eine weitere Untergliederung in Pakete, übersichtlich aufgeteilt in eigene Editor Fenstern („DiagramPartitioning“) wäre zur besseren Übersicht förderlich und wird ebenfalls von GMF unterstützt [GMF08d] .

Dass alle Elemente der Datenbindung angezeigt werden beeinträchtigt schnell die Übersichtlichkeit. Mittels zusätzlicher „Notation Styles“ ließen sich gezielt Elemente ausblenden.

Die Handhabung stark vereinfachen würden sich bei Doppelklick öffnende Dialoge (mittels `DirectEditPolicy`) und ggf. den Platz raubenden Property View optional machen.

Die Produktivität des Modellierers könnte weiter gesteigert werden, indem der GE aus einem Datenobjekte automatisch UI Elemente mit zugehörigen Datenbindungen erzeugen könnte, wie in [GMF08c] beschrieben.

Mittels des GMF-Service Framework ist auch eine Integration in den bestehenden UML2 Tools Editor möglich um beispielsweise den GE aus dem UML2 Tools Editor zu öffnen.

Literaturverzeichnis

[Ani06] Aniszczyk, C. 2006. Learn Eclipse GMF in 15 minutes. [Online] 12. 9 2006. [Zitat vom: 13. 3 2008.] http://www-128.ibm.com/developerworks/opensource/library/os-ecl-gmf/.

[Ani05] —. 2005. Using GEF with EMF. [Online] 8. 6 2005. [Zitat vom: 13. 3 2008.] http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html.

[Bru08] Bruck, J. Static Profile Definition. [Online] [Zitat vom: 17. 3 2008.] https://bugs.eclipse.org/bugs/show_bug.cgi?id=155535.

[Bru07] Bruck, J. und Hussey, K. 2007. Customizing UML: Which Technique is Right for You? [Online] 19. 7 2007. [Zitat vom: 13. 3 2008.] http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html.

[Bru06] —. 2006. Extending UML2: Creating Heavy-weight Extensions. [Online] 28. 8 2006. [Zitat vom: 11. 12 2007.] zurückgezogener Artikel.

[Bud03] Budinsky, F., et al. 2003. *eclipse Modeling Framework.* s.l. : Addison-Wesley, 2003.

[CDO08] CDO. [Online] [Zitat vom: 11. 3 2008.] http://wiki.eclipse.org/CDO.

[GMF08a] Developer Guide to Diagram Runtime Framework. [Online] [Zitat vom: 13. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Diagram%20Runtime.html.

[GMF08b] Developer's Guide to the Extensible Type Registry. [Online] [Zitat vom: 13. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.gmf.doc/prog-

guide/runtime/Developers%20Guide%20to%20the%20Extensible%20Type%20Registry/Developers%20Guide%20to%20the%20Extensible%20Type%20Registry.html.

[GMF08d] Diagram Partitioning. [Online] [Zitat vom: 16. 3 2008.] http://wiki.eclipse.org/Diagram_Partitioning.

[Eff07] Efftinge, S., et al. 2007. openArchitectureWare User Guide. [Online] 15. 9 2007. [Zitat vom: 13. 3 2008.] http://www.eclipse.org/gmt/oaw/doc/4.2/openArchitectureWare-42-reference.pdf.

[Ell08] Ellenboom, C. 2008. *Java ist auch eine Insel.* s.l. : Galileo Computing, 2008.

[EMF08] EMF-Transaction. [Online] [Zitat vom: 10. 03 2008.] http://www.eclipse.org/modeling/emf/?project=transaction.

[Gam95] Gamma, E., et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1995.

[GEF08] GEF Programmer's Guide. [Online] [Zitat vom: 13. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.gef.doc.isv/guide/guide.html.

[GEM08] GEMS Home page. [Online] [Zitat vom: 13. 3 2008.] http://www.eclipse.org/gmt/gems/.

[GMF08] GMF Tutorial. [Online] [Zitat vom: 4. 3 2008.] http://wiki.eclipse.org/GMF_Tutorial.

[GMF08c] GMF Tutorial Part 3. [Online] [Zitat vom: 3. 4 2008.] http://wiki.eclipse.org/GMF_Tutorial_Part_3.

[Gro08] Gronback, R. GMF Tutorial BPMN. [Online] [Zitat vom: 4. 3 2008.] http://wiki.eclipse.org/GMF_Tutorial_BPMN.

[Hud05] Hudson, R. und Shah, P. 2005. GEF In Depth. [Online] 28. 2 2005. [Zitat vom: 13. 3 2008.] http://www.eclipsecon.org/2005/presentations/EclipseCon2005_23_GEF_Tutorial_Final.ppt.

[Hus07] Hussey, K. 2007. What do YOU want UML to be? [Online] 6. 3 2007. [Zitat vom: 13. 3 2008.] http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3703.

[Jec04] Jeckle, M., et al. 2004. *UML 2 glasklar.* s.l. : HANSER, 2004.

[JET08] JET. [Online] [Zitat vom: 13. 3 2008.] http://www.eclipse.org/modeling/m2t/?project=jet.

[Kuh08] Kuhn, S. 2008. GMF GenModel doesn't save xsi:schemaLocation properly. [Online] 14. 1 2008. [Zitat vom: 1. 14 2008.] https://bugs.eclipse.org/bugs/show_bug.cgi?id=215282.

[Mer07b] Merks, E. und Paternostro, M. 2007. EclipseCon 2007. [Online] 6. 3 2007. [Zitat vom: 11. 3 2008.] http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3845.

[Mer07a] Merks, E., et al. 2007. Effective use of the Eclipse Modeling Framework. [Online] 2007. [Zitat vom: 2007. 3 10.] http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3619.

[Min02] Mintert, S. 2002. *XML & Co.* s.l. : Addison-Wesley, 2002.

[Moo04] Moore, B & Dean, D., et al. 2004. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* s.l. : IBM, 2004.

[Mül07] Müller, B. 2007. *Einsatz von MDA im Front-End – Entwicklung eines Metamodells als Basis für die Codegenerierung in AndroMDA.* s.l. : Universität Heidelberg, 2007.

[OCL08] OCL Overview. [Online] [Zitat vom: 15. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.ocl.doc/references/overview/OCLOverview.html.

[MOF08] OMG MOF. [Online] [Zitat vom: 2. 2 2008.] http://www.omg.org/mof/.

[OMG08a] OMG Unified Modeling Language (OMG UML),Infrastructure, V2.1.2. [Online] [Zitat vom: 13. 3 2008.] http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.

[OMG08b] OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2. [Online] [Zitat vom: 13. 3 2008.] http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.

[Pla06] Plante, F. 2006. Introducing the GMF Runtime. [Online] 16. 1 2006. [Zitat vom: 13. 3 2008.] http://www.eclipse.org/articles/Article-Introducing-GMF/article.html.

[Pop04] Popma, R. 2004. JET Tutorial Part 1 (Introduction to JET). [Online] 31. 3 2004. [Zitat vom: 13. 3 2008.] http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

[Sca05] Scarpino, M. & Holder, S. & Ng, S. & Mihalkovic, L. 2005. *SWT/JFace IN ACTION.* s.l. : MANNING, 2005.

[Sha06] Shatalin, A und Tikhomirov, A. 2006. Graphical Modeling Framework Architecture Overview. [Online] 28. 9 2006. [Zitat vom: 13. 3 2008.] http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium15_GMF.pdf.

[Sta07] Stahl, T., et al. 2007. *Modellgetriebene Softwareentwicklung.* s.l. : dpunkt.verlag, 2007.

[Sta05] Staud, J. 2005. *Datenmodellierung und Datenbankentwurf. Ein Vergleich aktueller Methoden.* s.l. : Springer, 2005.

[Ten08] Teneo. [Online] [Zitat vom: 15. 3 2008.] http://www.eclipse.org/modeling/emft/?project=teneo.

[EMF05] 2005. The Eclipse Modeling Framework (EMF) Overview. [Online] 16. 6 2005. [Zitat vom: 11. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.emf.doc/references/overview/EMF.html.

[EMF04] 2004. The EMF.Edit Framework Overview. [Online] 1. 6 2004. [Zitat vom: 13. 3 2008.] http://help.eclipse.org/help33/topic/org.eclipse.emf.doc/references/overview/EMF.Edit.html.

[GME08] The Generic Modeling Environment. [Online] [Zitat vom: 13. 3 2008.] http://www.isis.vanderbilt.edu/projects/gme/.

[UMLi08] The Unified Modeling Language for Interactive Applications. [Online] [Zitat vom: 16. 3 2008.] http://trust.utep.edu/umli/.

[Tik07] Tikhomirov, A. und Shatalin, A. 2007. GMF Best Practices. [Online] 5. 3 2007. [Zitat vom: 13. 3 2008.] http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3739.

[Web07] Weber, D. 2007. Template aspects handled incorrectly. [Online] 05. 09 2007. https://bugs.eclipse.org/bugs/show_bug.cgi?id=202257.

[Wuc07] Wuchner, E, et al. 2007. Das "Generic Eclipse Modeling System" (GEMS): skalierbare Domänenmodellierung leicht(er) gemacht. [Online] 2007. [Zitat vom: 13. 3 2008.] http://www.sigs.de/publications/os/2007/04/wuchner_white_OS_04_07.pdf.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurde, sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Ort, Datum, Unterschrift

X _____

Der Verfasser versichert, dass die beiliegende CD-ROM und alle darauf enthaltenen Bestandteile (Dateien) auf Vieren überprüft und kein schädlicher, ausführbarer Code enthalten ist.

Ort, Datum, Unterschrift

X _____

Anhang A - Beispiele

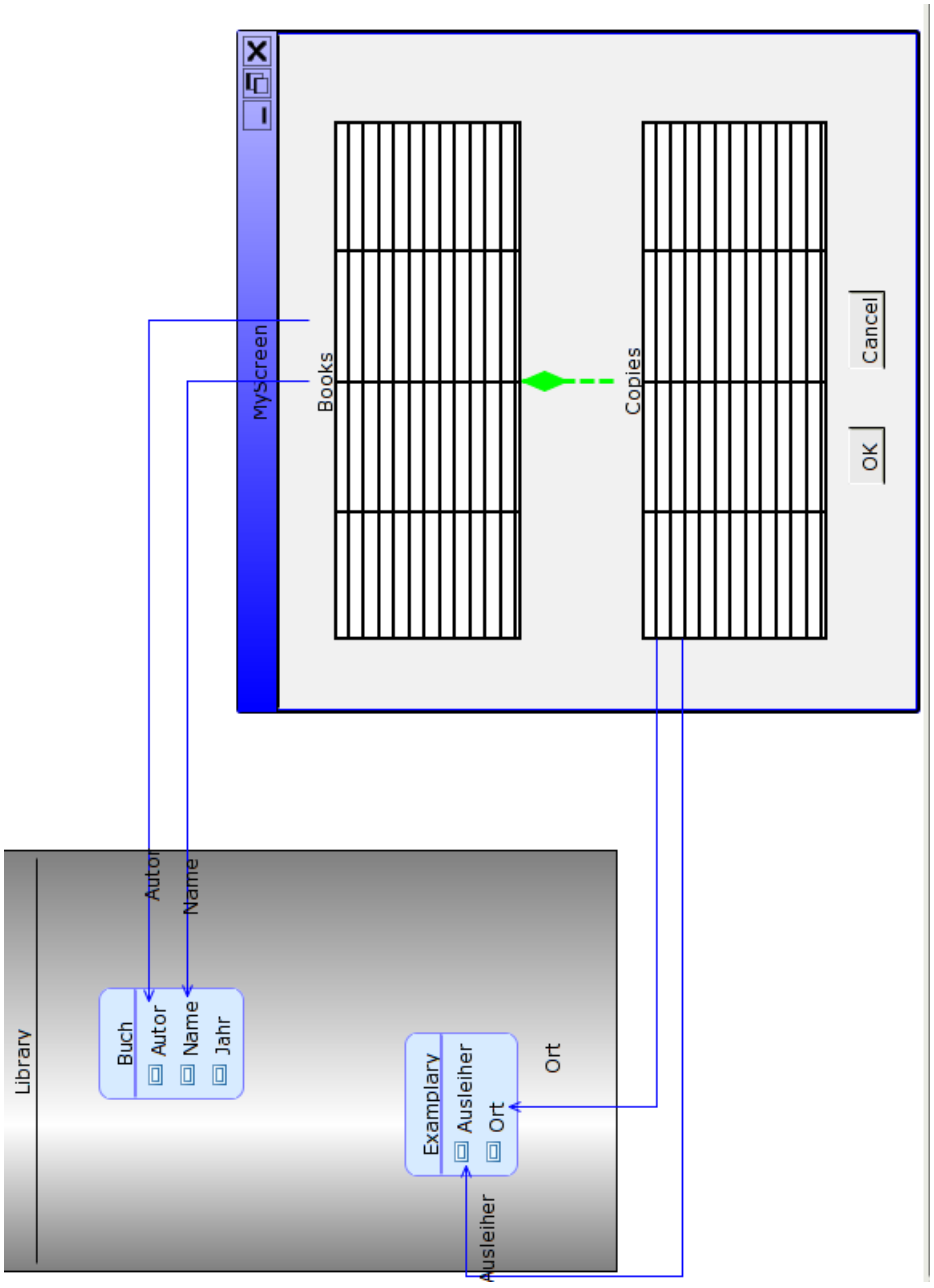

Abbildung 30 - GE Beispiel 1 / DSL

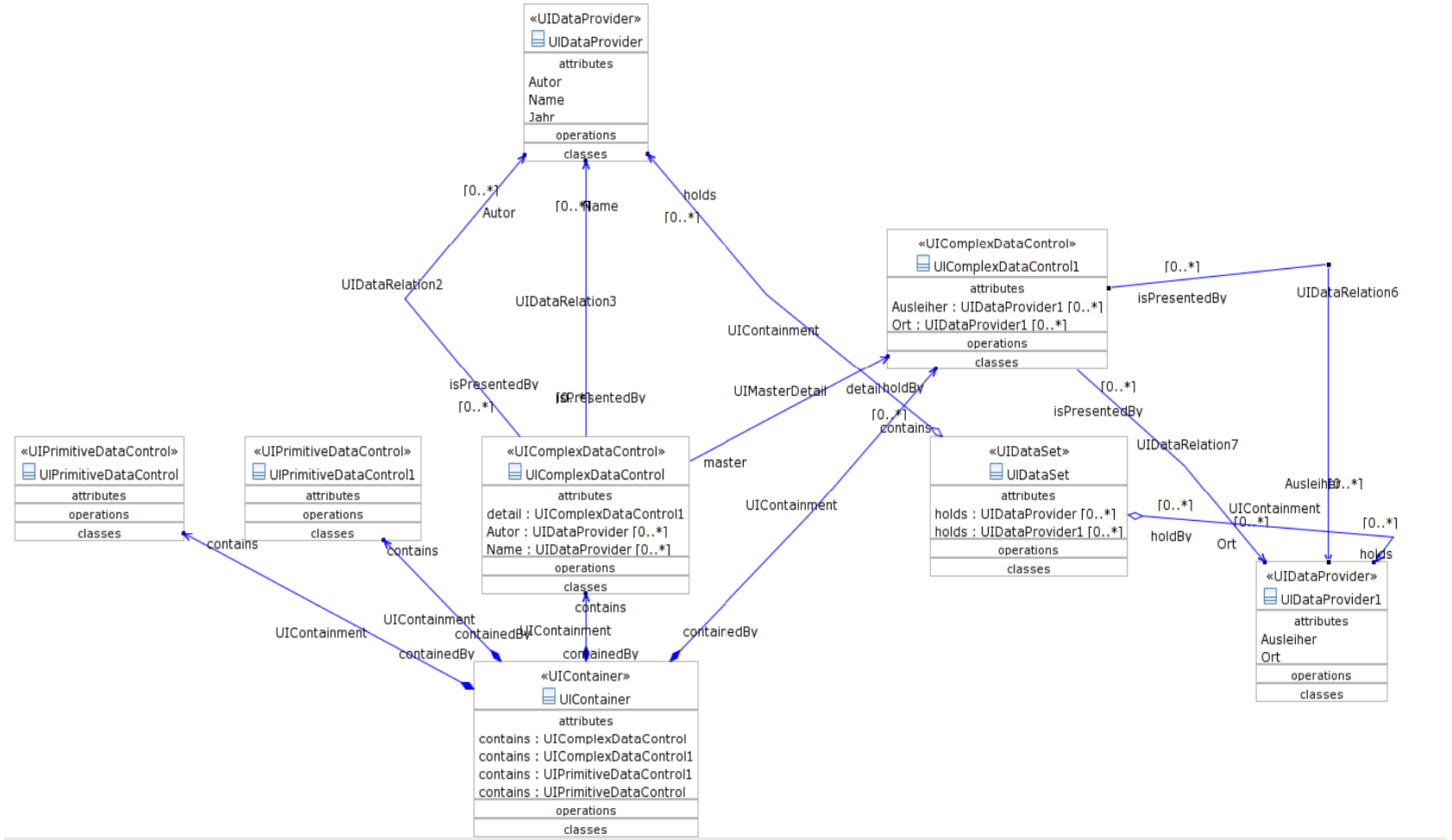

Abbildung 31 – GE Beispiel 1 / UML

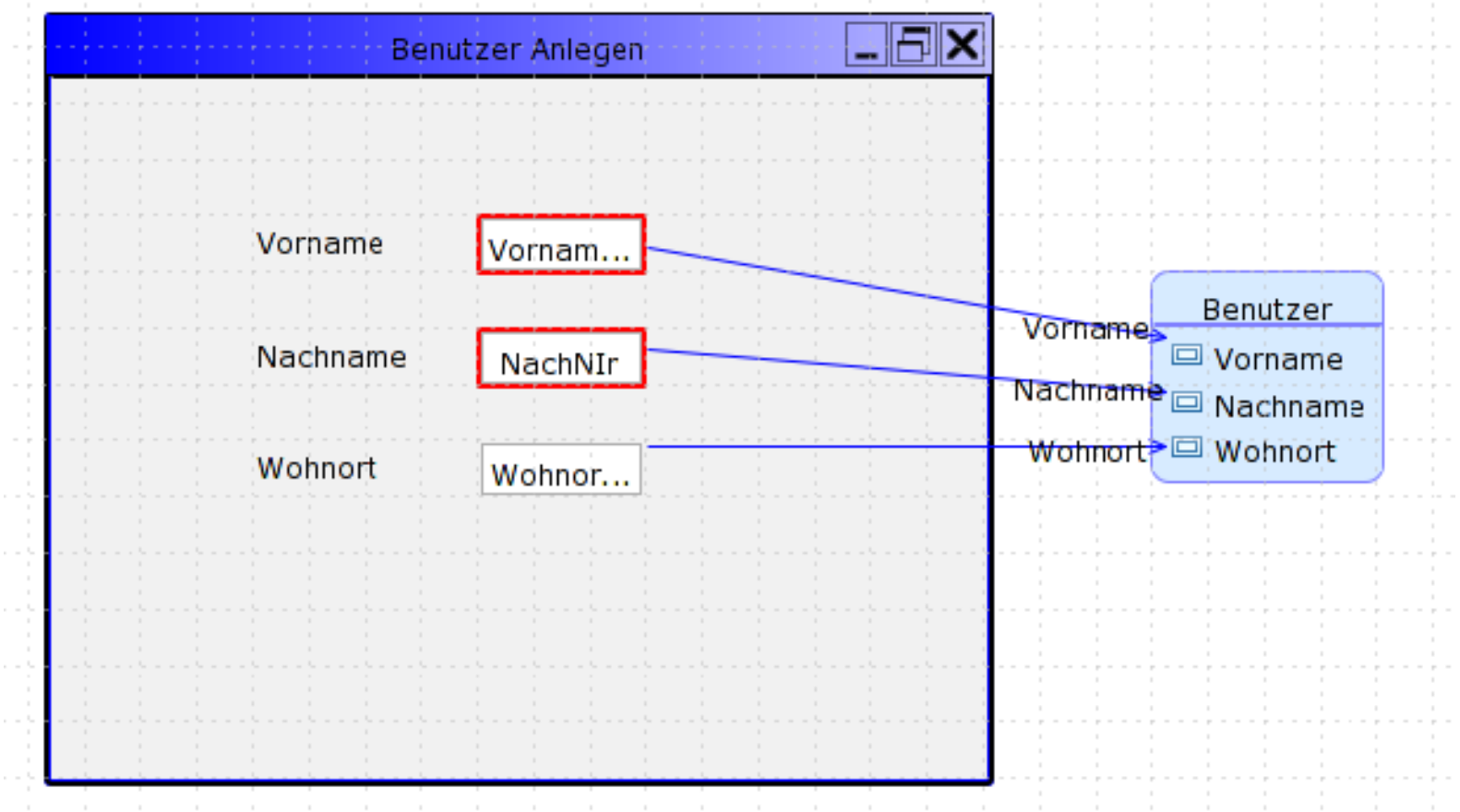

Abbildung 32 - GE Beispiel 2 / DSL

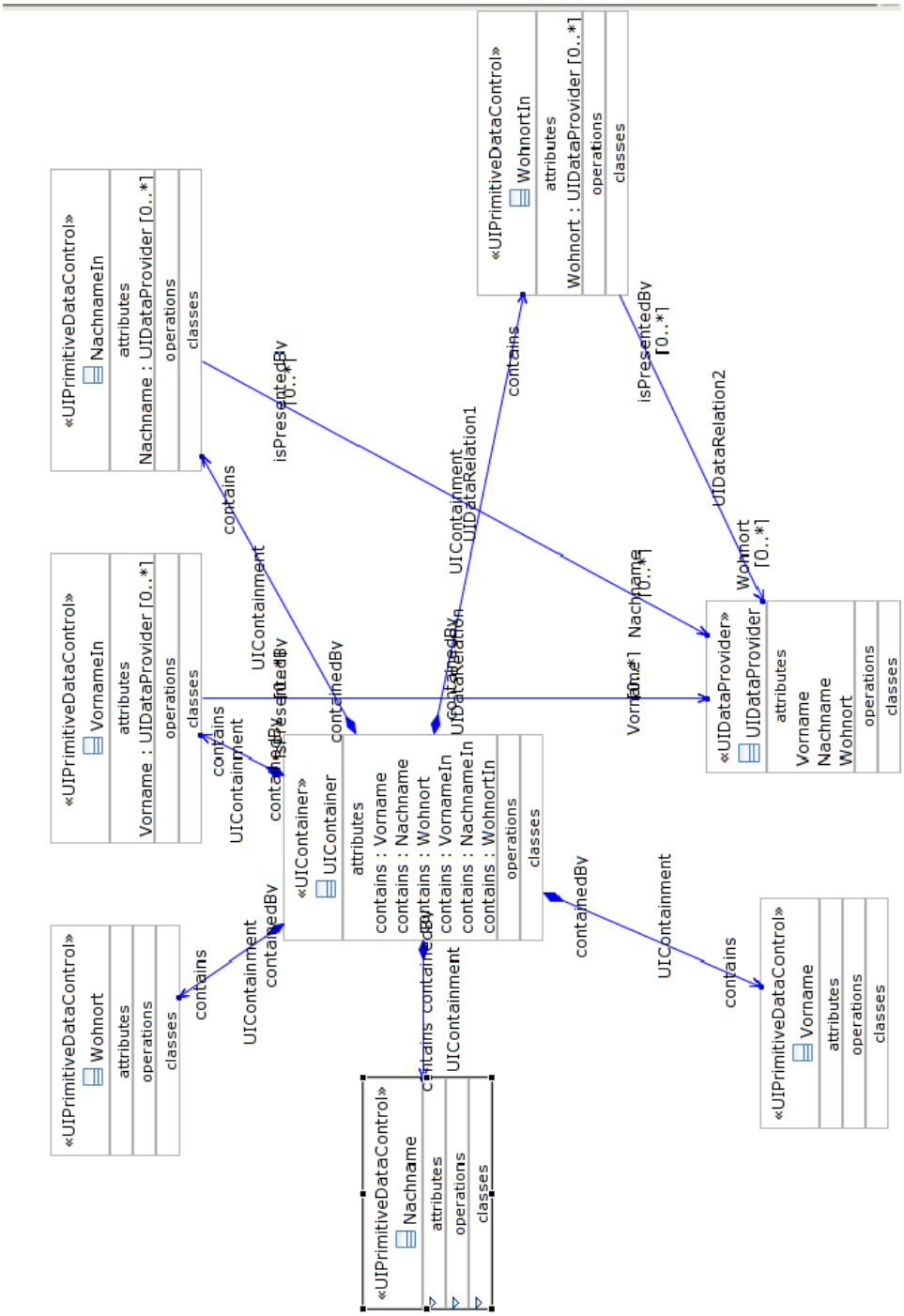

Abbildung 33 - GE Beispiel 2 / UML

Anhang B – Anpassungsmöglichkeiten

Für jede Anpassungsmöglichkeit werden folgenden Frage beantwortet:

- Eignet sie sich zum iterativen Vorgehen oder ist sie bei jeder Iteration erneut vorzunehmen?
- Wie invasiv ist sie? Greift sie in den Quellcode von GMF Tooling ein?
- Stellt sie eine Abhängigkeit zu einer bestimmten Version von GMF Tooling her?
- Ist sie skalierbar? Eignet sie sich dazu, seiteneffektfrei und schnell auf mehrere Elemente ausgedehnt zu werden?
- Entsteht zusätzlicher, rezeptiver, also fehleranfälliger Konfigurationsaufwand?
- Lässt sie sich später eindeutig einem Aspekt zuordnen und ermöglicht sie sie zu gliedern?
- Trennt sie Generat von handgeschriebenem Quellcode? Verlangt sie also das Generat zu versionieren? Hierdurch entsteht das Problem der „verwaisten“ Klassen. Darunter ist eine generierte Klasse zu verstehen, die im aktuellen Modell nicht mehr, oder unter anderem Namen, generiert werden würde, und somit nicht mehr verwendet wird. In diesem Fall wären sie manuell zu löschten.

Folgenden Implementierungen wurden betrachtet:

- GMF Ecore Editor (org.eclipse.gmf.ecore.editor)
- UML2 Tools (org.eclipse.uml2.diagram*)
- Logic Editor (org.eclipse.gmf.examples.runtime.diagram.logic*)
- Taipan Editor (org.eclipse.gmf.examples.taipan* und org.eclipse.gmf.sketch*)
- Mindmap Editor (org.eclipse.gmf.examples.mindmap.*)

- BPMN Editor [Gro08]
- oAWs GMF Adapter [Eff07]
- GEMS

Zu jeder Anpassungsmöglichkeit wird auf eine der o.g. Implementierungen verwiesen.

Die Anpassungsmöglichkeiten lassen sich grob in vier Kategorien unterteilen:

- Anpassungen ohne generative Unterstützung.
- Anpassungen am gmfgen-Modell. Obwohl mehrere Modelle anpassbar sind, nimmt das gmfgen-Modell eine Sonderstellung ein.
- Eigens definierte M2T Transformation. Die Ausprägungen dieses Bereichs sind zuerst zusammenhängend zu lesen.
- Eigens definierte M2M Transformation

Anpassungen im Quellcode

Kurzfassung: Ein bestehender Bereich im Quellcode wird als „protected Region“ ausgezeichnet.

Vorgehen: Der Quellcode der anzupassenden Methode wird umgeschrieben und mittels „@generated NOT“ gegen erneutes überschreiben geschützt.

Vorteile:

Änderungen bleiben erhalten und eignen sich zur iterativen Entwicklung. Diese greift weder in GMF-Tooling ein noch erzeugt sie eine Abhängigkeit zu ihm. Es entsteht kein Konfigurationsaufwand, sie ist die primitivste Anpassungsform.

Nachteile:

Sie ist lokal auf die Methode beschränkt, Änderungen müssen für jedes Element vorgenommen werden, hierdurch steigen der Wartungsaufwand und die Fehlerwahrscheinlichkeit. Übergreift ein Aspekt mehrere Methoden wird ist es erschwert die Änderung diesem Aspekt zuzuordnen. Ähnliches gilt für Änderungen zweier Aspekte in einer Methode. Generat und handgeschriebener Code werden vermischt und es entsteht die Gefahr „verwaister“ Klassen.

Verwendungsempfehlung:

Bei vielen folgenden Änderungen wird zuerst der Quellcode angepasst bis er korrekt funktioniert. Die o.g. Form eignet sich gut, modifizierte Bereiche auszuzeichnen und sie später iterativ in eine andere Änderungsform zu überführen bis keine so ausgezeichnet Bereiche existieren.

Implementierungen:

UML2 Tools, Taipan, Mindmap, BPMN Editor

Anpassungen durch Extensions

Kurzfassung:

Ein von der GMF Runtime bereitgestellter Extension Point wird verwendet.

Vorgehensweise:

Ein Provider wird erstellt, in der plugin.xml eingetragen und sein Verwendungsbereich eingeschränkt. Ist die Einschränkung mittels der XML Syntax nicht möglich, ist eine weitere Klasse zu erstellen. Die eigentliche, erweiternde Komponente ist nun zu erstellen und von dem Provider zurückzugeben.

Vorteile:

Extensions eigenen sich zum iterativen Vorgehen, greifen nicht in GMF Tooling ein und stellen keine Abhängigkeit hierzu her. Sie können mit

zusätzlichem Konfigurationsaufwand auf mehrere ähnliche Elemente ausgedehnt werden. Sie trennen generierten vom handgeschriebenen Quellcode und eignen sich besonders um fremde, nicht selbst erstellte Editoren anzupassen.

Nachteile:

Sie müssen vergleichsweise aufwändig in der plugin.xml konfiguriert werden. Besonders fehleranfällige ist die korrekte Verwendung ihre extension points sowie sie richtig auf die gewünschten Elemente einzuschränken. Es müssen ganze Klassen ausgetauscht werden, was in der Mehrzahl der Fälle kopieren von bestehenden, generierten Quellcodes mit marginaler Anpassung bedeutet. In einigen Erweiterungsfällen ersetzt die extension einen Bereich und ergänzt ihn nicht. Es ist nun schwer sie einem Aspekt zuzuordnen. Änderungen mittels Extensions benutzen meist eine eindeutige Kennung des anzupassenden Elements. Dies ist nachteilig, wenn sich diese, im generativen Prozess ändert.

Verwendungsempfehlung:

Sie sind zu verwenden um bestehende Editoren anzupassen.

Implementierungen:

Mindmap, Logic

Anpassungen am GMF Generatormodell

Kurzfassung:

Anpassung des gmfgen Modells wie jedes andere Modell.

Vorgehensweise:

Attribute eines Modellelements werden angepasst, bzw. neue Modellelemente hinzugefügt.

Abgrenzung:

Die mittels des gmfgen-Modells unter diesem Punkt beschrieben Anpassungen nehmen eine Sonderstellung ein. Zum einen müssen keine Templates angepasst werden, zum anderen dekoriert, also referenziert, das gmfgen Modell keine anderen Modelle sondern wird aus ihnen generiert.

Vorteile:

Sie greifen nicht in GMF-Tooling ein, erzeugen keine Abhängigkeit zu ihm und sind einfach zu handhaben. Änderungen werden auf höherer Abstraktionsebene im Modell vorgenommen, sind also skalierbar, mittelmäßig gut einem Aspekt zuzuordnen und erfordern nicht das Generat zu versionieren.

Nachteile:

Das gmfgen-Modell wird aus dem gmfmap-Modell erzeugt und für manuelle Anpassungen entstehen die aus MDA bekannten Konsistenzprobleme zwischen plattformunabhängigen und plattformspezifischen Modell [Sta07] . Problematisch ist, zu unterscheiden, ob ein im gmfgen-Modell vorhandenes Element aus dem gmfmap-Modell erzeugt wurde, bzw. wie sich Änderungen des Mapping-Modells auf das Generatormodell auswirken. Es ist weder dokumentiert, welche Änderungen erhalten bleiben, noch ist eine Systematik zu erkennen. Das Verhalten ist jedoch für Elemente eines Typs auf gleicher Tiefe reproduzierbar. Änderungen, die überschrieben werden, sind für das iterative Vorgehen nicht geeignet.

Verwendungsempfehlung:

Es gilt zuerst zu prüfen ob die Änderung beim erneuten Erzeugen des Generatormodells erhalten bleibt. Ist dies der Fall, sollte diese Form verwendet werden. Die erste Ausnahme ist das Generatormodell zu verwenden um neue Elemente hinzuzufügen, die in einer eigenen Datei resultieren. Hierdurch kann die GMF Einschränkung, welche es nicht erlaubt durch eigene Templates neue Dateien zu erzeugen, umgangen

werden. Die zweite Ausnahme gilt, falls es angemessen erscheint, elementübergreifende Änderungen nicht unter jedes Element zu ordnen, sondern einem Aspekt die zugehörigen Elemente zuzuordnen.

Implementierung:

UML2 Tools, Mindmap

Statische Anpassung der Templates

Kurzfassung:

Die Templates der M2T Transformation werden statisch, für alle Elemente eines Typs angepasst.

Vorgehensweise:

Die Standard-Templates werden kopiert und der Pfad zum Wurzelverzeichnis wird in dem Generatormodell angegeben. Sie werden entweder direkt oder durch Aspekte angepasst. Die Dateinamen sind nicht zu ändern und die oAW-Aspekte wirken nur in einer Datei.

Vorteile:

Sie eignen sich gut zum iterativen Vorgehen und greifen an einer vordefinierten Stelle in GMF-Tooling ein. Sie gelten für alle Instanzen eines Meta-Typs und wirken unmittelbar auf sie. Der Konfigurationsaufwand ist minimal und das Generierte muss nicht versioniert werden, wohl aber die Templates.

Nachteile:

Die angepassten Templates sind abhängig zu einer bestimmten GMF Version. Änderungen gelten für alle oder keine Elemente eines Typs.

Verwendungsempfehlung:

Sie eignen sich gut, um projektspezifische ganzheitliche Änderungen vorzunehmen, deren Darstellung im Generatormodell nicht vorkommt

oder wenig Sinn macht. Ein Beispiel hierfür können Bibliotheks-Plug-Ins importiert werden. oAW-Aspekte dabei zu verwenden ist, sofern möglich, immer sinnvoll.

Implementierung:

UML2 Tools

Template-Anpassungen anhand eines eindeutigen Attributs:

Kurzfassung:

Ein eindeutiges Attribut eines anzupassenden Elements wird identifiziert und eine Abfrage nach diesem in ein oder mehrere Templates eingebaut

Vorgehensweise:

s.o.

Vorteile:

Solange sich das Attribut nicht ändert eignet es sich zum iterativen Vorgehen und ermöglicht elementbasierte Anpassungen. Sie greift an vordefinierter Stelle in GMF-Tooling ein, ist auf mehrere Elemente ausweitbar und erfordert nicht das Generat zu versionieren.

Nachteile:

Wie jede Template-Anpassung müssen die Templates nun versioniert werden und sie sind von denen der verwendeten GMF Version abhängig. Ändert sich das eindeutige Attribut oder sind weitere Elemente der Gruppe anzupassender Elemente hinzuzufügen entsteht Konfigurationsaufwand. Anpassungen lassen sich leicht einem Typ zuordnen, nicht jedoch einem Aspekt.

Verwendungsempfehlung:

Diese Anpassungsart sollte unter keinen Umständen verwendet werden.

Implementierung:

Template-Anpassungen anhand eines erbenden Generatormodell

Kurzfassung:

Das gmfgen-Meta-Modell wird erweitert, Instanzen dieser Erweiterung dem Generatormodell hinzugefügt und die Templates auf sie zugeschnitten angepasst.

Vorgehensweise:

Ein eigenes Meta-Modell muss erstellt werden, die beinhaltenden Meta-Klassen müssen von denen des gmfgen-Meta-Modells erben. Folglich sind die Instanzen der gmfgen-Meta-Klassen im Generatormodell durch Instanzen des erbenden Metaklasse des eigenen Metamodells zu ersetzen und die Templates entsprechend anzupassen.

Vorteile:

Eigene Elemente lassen sich definieren und Anpassungen innerhalb der Templates auf eigene Typen und ihre Spezialisierungen zuschneiden. Generat und Quellcode werden voneinander getrennt, es entstehen keine Seiteneffekte und Änderungen lassen sich eigenen Meta-Klassen inklusive eigenen, erbenden Meta-Klassen zuordnen.

Nachteile:

Sie stellt die abhängigste Form der Erweiterungsmöglichkeiten da und ist mit dem „Middleweight“-Ansatz [Hus07] der UML vergleichbar. Meta-Modell sowie Templates sind von GMF-Tooling abhängig, welches jedoch keine Instanzen eigener Meta-Klassen erzeugt und ggf. Änderungen zu eigenen überschreibt. Ein iteratives Vorgehen wird hierdurch stark beeinträchtigt. Zudem lassen sich Änderungen nur einem Typ,

nicht jedoch einem Aspekt zuordnen. Das eigene Meta-Modell ist mit Sorgfalt zu erstellen und ist vergleichsweise unflexibel.

Verwendungsempfehlung:

Diese Form ist zu vermeiden. Empfehlenswert ist sie mit angepasster M2M Transformation, jedoch sollte geprüft werden, ob nicht ggf. aufgrund der Schwächen der GMF Domänenarchitektur (s.S. 93) eine komplett eigene vorzuziehen ist. Die hier erwähnte Anpassungsmöglichkeit ist also nur in einem begrenzten Rahmen zwischen taktischer und strategischer Entscheidung zu empfehlen.

Implementierung:

UML2 Tools

Template-Anpassungen anhand erweiternder Generatormodelle

Kurzfassung:

Elemente des Generatormodells werden, wie UML Stereotype, mittels „Model Markings“ [Sta07] ausgezeichnet und dies in den Templates berücksichtigt.

Vorgehensweise:

Ein eigenes Meta-Modell ist zu erstellen, dessen Meta-Klassen eine Referenz auf Meta-Klassen des gmfgen-Modells beinhalten. Das eigene Meta-Modell ist nun im Generatormodell zu laden um dort Instanzen des eigenen Metamodells erzeugen zu können. Die Elemente des Generatormodells werden nun von Instanzen des eigenen Metamodells referenziert. In einem angepassten Template wird für die Instanz eines gmfgen-Typs die Menge aller Instanzen die diese referenzieren auf eine Instanz einer eigenen Metaklasse gefiltert. Sie kann nun mit ihren Attributen im Template berücksichtigt werden.

Vorteile:

Die Änderungen eignen sehr gut zum iterativen Vorgehen. Das Generat muss nicht versioniert werden, in den Quellcode von GMF-Tooling wird nicht eingegriffen und sie ist vergleichsweise unabhängig vom gmfgen-Meta-Modell. Anpassungen können schnell auf mehrere Elemente ausgedehnt werden, sind eindeutig einem Aspekt zuzuordnen und die Anordnung im erweiternden Modell ist frei wählbar. Die Lösung ist sehr flexibel, da ein Element mehrfach erweitert werden kann und das eigene Meta-Modell folglich iterativ (weiter-)entwickelbar ist.

Nachteile:

EMF Modelle, also auch das Generatormodell, referenzieren standardmäßig nach Pfad vom Wurzelelement des Modells aus. Beispielsweise referenziert „/0/@diagram/@palette/@groups.3“ im ersten Modell das vierte Element von „groups“ in der ersten „palette“ im ersten „diagram“. Dies wird problematisch falls bei einer Iteration durch eine Änderung im mapping-Modell ein neues Element eingefügt wird. Zudem wird das „dekorierende“ Modell bei jeder Iteration überschrieben, jedoch nimmt dieses nur Änderungen am Anfang und am Ende der Serialisierung vor, die innerhalb von dreißig Sekunden bis einer Minute manuell übertragbar sind.

Verwendungsempfehlung:

Diese Anpassungsart ist sehr flexibel und sollte weitreichend verwendet werden. Es bleibt offen ob genannte Probleme beim einfügen von neuen Elementen ggf. durch Referenzen mittels pseudo-einzigartige IDs, wie von EMF für die UML Serialisierung verwendet wird, zu lösen ist. Werden dann für jede M2M Transformation neue IDs erzeugt, gilt zu prüfen ob die im experimentellen GMF Zweig verfügbare „.trace“-Erweiterung, welche die M2M Abbildung dokumentiert, weiterhilft.

Namensgebung:

Zwei von GMF verwendeten Namen werden im Bereich MDSD bereits abweichend verwendet. Zum einen wird o.g. Anpassungsart als „Decorate GMFGen model“ bezeichnet [Tik07] . Im Gegensatz zu EMFs „.genmodel“, welches ein dahinter liegendes Ecore Modell referenziert, setzt GMFs Generierung von „gmfgen“ nicht an den referenzierenden sondern an den referenzierten Elemente an. Zum anderen wird auf gleicher Seite im Kommentar diese Anpassungsart als „Heavy-weight advanced technique“ bezeichnet. Die XMI Serialisierung hiervon zeigt Ähnlichkeiten zu UML Stereotypen. Stereotypen werden jedoch als „Lightweight“ Erweiterung bezeichnet [Bru07] . Woran sich die GMF Entwickler bei ihrer Namensgebung orientierten ist dem Autor unklar.

Implementierung:

oAWs GMF Adapter

Anpassung der M2M Transformation in der „Generator“ Klasse

Kurzfassung:

Die in Java definierte M2M Transformation der „Generator“ Klasse ist nach eigenen Anforderungen anzupassen. Ggf. können bestehende Meta-Modelle erweitert werden.

Vorteile:

Das resultierende Generatormodell ist frei anpassbar. Durch geeignete Anpassung kann vermieden werden das Generatormodell manuell zu ändern. Es eignet sich folglich sehr gut zum iterativen Vorgehen, skaliert gut und es entsteht im Einzelfall kein zusätzlicher Konfigurationsaufwand. Zudem können bestehende Teile der „Generator“ Klasse wiederverwendet werden.

Nachteile:

Es wird weitreichend in GMF-Tooling eingegriffen an nicht explizit dafür vorgesehener Stelle. Die „Generator“ Klasse ist, verglichen mit dem anpassbaren oAW-Workflow, statisch, Java als M2M Transformationssprache vergleichsweise ungeeignet. Der Aufwand zum Anpassen steigt schnell.

Verwendungsempfehlung:

Aufgrund der hohen Abhängigkeit der Anpassungsart zu GMF-Tooling und ihrer schwerfälligen Handhabung ist sie nur sehr eingeschränkt zu empfehlen. Zudem setzt das Zielmodell, das Generatormodell, die Konzepte der Zielplattform, der GMF-Runtime, nur eingeschränkt um und lässt Flexibilität vermissen [siehe Kritische Betrachtung GMF Runtime].

Implementierung:

Taipan

Eigenständiges Generator Projekt

Kurzfassung:

Erstellung eines zusätzlichen Generatorprojekts beispielsweise mit oAW.

Vorgehensweise:

Je nach gewünschter Flexibilität. Es reicht vom „dekorieren“ von Elementen des Generatormodells mit angepassten Templates um die später durch den eigenständigen Generator erzeugte Klassen zu nutzen bis hin zu einer komplett eigenständigen generativen Architektur.

Vorteile:

Beseitigt bestehende Einschränkungen von GMF-Tooling.

Nachteile:

Entweder entsteht ein zusätzlicher Schritt in der Iteration oder hoher Aufwand zum implementieren der eigenen generativen Architektur.

Verwendungsempfehlung:

Es eignet sich als taktische Anpassung von GMF-Tooling, es zu erweitern um sich von bestehenden Einschränkungen zu lösen. Eine eigenständige generative Architektur sollte aufgrund der Komplexität als strategische Lösung betrachtet werden.

Implementierung:

taktisch: nicht bekannt

strategisch: GEMS

Anhang C – Weitere Templateanpassungen

A: Abhängigkeiten zum Profil und zum Bibliotheks-PlugIn (lib) sind zu setzen

T: T/plugin/manifest.xpt

L: Die o.g. PlugIns werden statisch in die MANIFEST.MF generiert

A: Von GMF allgemein fehlerhaft generiertes Copy and Paste wird deaktivert.

T: T/plugin/plugin.xpt

L: Die hierfür notwendige Extension in der plugin.xml wird in den Templates kommentiert.

A: Die Namensgebung anhand des modelID Attributs einer GenPlugin Instanz ist teilweise unerwünscht.

T: T/plugin/properties.xpt

L: Eine Variable wird definiert und zugewiesen, sie ersetzt statisch das modelID Attribut an den gewünschten Stellen.

A: Der Wizard um neue Modelle zu erstellen ist einer eigenen Kategorie zuzuweisen

T: T/editor/extensions.xpt

L: Eine Kategorie wird der plugin.xml hinzugefügt.

A: Die minimale initiale Größe ist für einige Elemente zu groß und ist zu verkleinern

T: A/diagram/editparts/NodeEditPart.xpt

L: Der Aspekt um „createNodePlateMethod“ setzt die initialen Werte nach einer Instanz von „NodeInitialSize“