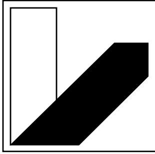

Master Thesis

Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung

Bastian Roth

Lehrstuhl für
Datenbanken und
Informationssysteme

Angewandte Informatik IV

Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung

Master Thesis im Fach Angewandte Informatik
zur Erlangung des akademischen Grades
Master of Science

vorgelegt von

Bastian Roth

geboren am 14.09.1984 in Kemnath
basti-dorrest@gmx.de

Angefertigt im Zeitraum
vom 01.05.2010 bis 13.09.2010

Lehrstuhl für Angewandte Informatik IV
Datenbanken und Informationssysteme
Universität Bayreuth

Betreuer:
Prof. Dr.-Ing. Stefan Jablonski
Bernhard Volz, M.Sc.

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Neusorg, den 13.09.2010 ________________________________

Zusammenfassung

Aufgrund starker Fluktuation der Anforderungen sind Unternehmen dazu gezwungen, ihre Geschäftsprozesse zu analysieren und zu verbessern. Nach essentiellen Änderungen dieser Anforderungen oder der Erschließung neuartiger Anwendungsdomänen ist es häufig nicht ausreichend, die zugrunde liegenden Prozessmodelle entsprechend anzupassen. Ein weiterer, tiefer greifender Ansatz ist die Adaption der verwendeten Modellierungssprache. Hiermit lassen sich komplexe Sachverhalte prägnanter als mit einer allgemeinen Sprache ausdrücken, da spezielle Konstrukte mit domänenspezifischer Semantik definiert werden können. Modellierungssprachen verfügen stets über eine abstrakte Syntax, mittels der die besagten Konstrukte angeboten werden. Zusätzlich können zu jeder abstrakten Syntax beliebig viele konkrete Syntaxen definiert werden.

Im Rahmen der vorliegenden Arbeit wird ein System entwickelt, mit dessen Hilfe zu einem frei definierbaren Meta-Modell eine konkrete grafische Syntax spezifiziert werden kann. Die Sprache zur Definition der abstrakten Syntax ist dabei identisch mit der Sprache zur Spezifikation der konkreten Syntax. Letztere repräsentiert die Grundlage eines Diagramm-Editors, der zum Erstellen und Manipulieren von Modellen in der durch das Meta-Modell deklarierten domänenspezifischen Sprache dient. Ein charakteristisches Merkmal dieses Ansatzes ist unter anderem die dynamische Anpassung der grafischen Darstellung zur Laufzeit. Obwohl als Anwendungsfall die perspektivenorientierte Prozessmodellierung zum Einsatz kommt, ist die Allgemeingültigkeit des entwickelten Modellierungswerkzeugs durch das generisch gehaltene Konzept sichergestellt.

Inhaltsverzeichnis

Kapitel 1 Einleitung 1

1.1 Problemstellung und Zielsetzung 1

1.2 Aufbau der Arbeit 2

Kapitel 2 Grundlagen 5

2.1 Meta-Modellierung 5

2.2 Open Meta Modelling Environment 5

2.2.1 Meta Object Facility 6

2.2.2 Erweiterte Modellierungsmuster 7

2.2.3 Das linguistische Meta-Modell 9

2.3 Klassifikation visueller Sprachen 11

2.4 Perspektivenorientierte Prozessmodellierung 12

Kapitel 3 Related Work 15

3.1 BPM-Systeme mit Werkzeugen zur Prozessmodellierung 15

3.2 Frameworks zur Erstellung grafischer Modellierungswerkzeuge 15

3.2.1 ConceptBase 16

3.2.2 MOFLON 16

3.2.3 Generic Modeling Environment 17

3.2.4 AToM[3] 18

3.2.5 MetaEdit+ 19

3.2.6 Graphical Modeling Framework 20

3.2.7 DSL Tools für Microsoft Visual Studio 2010 21

3.2.8 Zusammenfassender Vergleich 21

3.3 Werkzeuge zur Implementierung grafischer Editoren 23

3.3.1 SWT und Swing 23

3.3.2 GEF mit Draw2D 23

3.3.3 Zest 24

3.3.4 GMF 24

3.3.5 Fazit 25

Kapitel 4 Verwendete Technologien 27

4.1 Eclipse 27

4.2 Draw2D 27

4.3 GEF 28

Kapitel 5 Hierarchie der (Meta-)Modelle 31

5.1 Domain-Model Stack 32

5.2 Designer-Model Stack 32

Kapitel 6 Spezifikation der Meta-Modelle 35

6.1 Java-Bean-Mapping-Model 35

6.2 Graphical-Meta-Model 36

6.2.1 Inhalt des types-Package 37

6.2.2 Grundlegende Concepts 38

6.3 Editor-Meta-Model ... 41
6.3.1 Identifikation möglicher Mappings zwischen Domäne und grafischer Darstellung ... 41
6.3.2 Spezifikation des Inhalts ... 44
Kapitel 7 Anwendungsfall: Definition eines Prozess-Designers ... 49
7.1 Definition des Prozess-Diagramms ... 50
7.2 Definition des Prozess-Knotens ... 51
7.2.1 Anzeige des Prozessnamens ... 53
7.2.2 Einsatzmöglichkeit des RewriterService ... 54
7.2.3 Compartments für Eingangs- und Ausgangsdaten ... 56
7.3 Definition von Kontroll- und Datenfluss-Verbindung ... 57
7.3.1 Kontrollfluss ... 57
7.3.2 Datenfluss ... 58
7.4 Allgemeingültigkeit des Anwendungsfalls ... 60
Kapitel 8 Implementierung: Model Designer Framework ... 61
8.1 Architektur ... 61
8.1.1 Model Access Modul ... 62
8.1.2 LMM2Java Modul ... 64
8.1.3 User Interface Modul ... 65
8.1.4 Editor und Graphical Elements Modul ... 66
8.2 Benutzerinteraktion mit dem Designer ... 67
8.2.1 Erstellen und Löschen eines Knotens ... 67
8.2.2 Ändern von Größe und Position eines Knotens ... 68
8.2.3 Erstellen und Löschen einer Verbindung ... 68
8.2.4 Ersetzen von Start/Ziel einer Verbindung ... 69
8.2.5 Ändern von Properties ... 69
8.3 Stand der Entwicklung ... 70
8.4 Ausblick ... 71
Literaturverzeichnis ... 73
Anhang A Definition-Models des Prozess-Designers ... 77
A.1 Domain-Meta-Model ... 77
A.2 Graphical-Definition-Model ... 77
A.3 Editor-Definition-Model ... 82
Anhang B Inhalt der beiliegenden CD-ROM ... 85

Abbildungsverzeichnis

Abb. 2-1: Architektur der oMME-Plattform [6] ... 6

Abb. 2-2: Beispiel einer Meta-Modell-Hierarchie in Anlehnung an MOF ... 7

Abb. 2-3: Beispiel eines orthogonal klassifizierten Meta-Modells ... 8

Abb. 2-4: Beispiel für die Spezialisierung von Instanzen ... 9

Abb. 2-5: Exemplarischer Aufbau eines Modells konform zur Spezifikation des LMM in abstrakter Syntax [6] ... 9

Abb. 2-6: Graph- und Geometrie-basierte Visualisierung desselben Sachverhalts ... 11

Abb. 2-7: Verwendung dreier Prozesse zur Definition eines größeren Prozesses [6] ... 13

Abb. 2-8: Angepasste Verwendung eines Prozesses [6] ... 13

Abb. 3-1: UML-Klassendiagramm der Meta-Meta-Ebene eines GME-Modellierers [40] ... 17

Abb. 3-2: User Interface von AToM³ nach dem Starten mit leerem Modell ... 19

Abb. 3-3: Übersicht über allgemeines Meta-CASE-Tool ... 20

Abb. 3-4: Überblick über den Entwicklungsprozess eines grafischen Editors mit GMF [51] ... 20

Abb. 3-5: Tabellarischer Vergleich von Frameworks zur Erstellung domänenspezifischer, grafischer Modellierungswerkzeuge auf der Basis von Metamodellen ... 22

Abb. 4-1: Draw2D-Komponentenhierarchie (nach [75]) ... 27

Abb. 4-2: Clipping einer Figure-Hierarchie [75] ... 28

Abb. 4-3: PolylineConnection mit drei Child-Figures (nach [75]) und zugehöriger Figures-Hierarchie 28

Abb. 4-4: MVC-Architektur von GEF (nach [75]) ... 29

Abb. 4-5: Szenario für mögliche Beziehungen zwischen den einzelnen Elementen von Model, View und Controller bei GEF ... 30

Abb. 5-1: Einführung von Modellklassen analog zum MVC-Pattern von GEF ... 31

Abb. 5-2: Modell-Hierarchie zur Beschreibung eines Designers (exemplarisch mit Typ-Verwendungs-Konzept auf der Seite des Domänen-Modells) ... 32

Abb. 6-1: JavaBeanMapping-Anwendungsbeispiel ... 36

Abb. 6-2: Konzeptdiagramm des Graphical-Meta-Models ... 36

Abb. 6-3: Exemplarische Verwendung von `ShapeWithSelectableFigure` ... 40

Abb. 6-4: Visualisierung von Beziehungen in einem Prozessmodell nach der Geometrie-basierten Variante ... 42

Abb. 6-5: Visualisierung einer Attribut-basierten Beziehung in einem Prozessmodell nach der Graph-basierten Variante ... 43

Abb. 6-6: Visualisierung einer Entität-basierten Beziehung in einem Prozessmodell nach der Graph-basierten Variante 43

Abb. 6-7: Visualisiertung des Zustands zweier Prozesse 44

Abb. 6-8: Konzeptdiagramm des Editor-Meta-Models 45

Abb. 7-1: Vereinfachtes POPM-Meta-Modell 49

Abb. 7-2: Dialog zum Erzeugen des Diagramm-Modellelements und der sich anschließende leere Designer 50

Abb. 7-3: Domain-Meta-, Editor-Definition, Domain- und Editor-Usage-Model für leeren Designer 51

Abb. 7-4: Designer mit einer `Process`- und einer `ANDConnector`-Verwendung 53

Abb. 7-5: Domain-, Editor-Usage- und Graphical-Usage-Model für Designer mit einer visualisierten Prozess-Verwendung 53

Abb. 7-6: Editor-Usage- und Graphical-Usage-Model mit `EditorLabel`- bzw. `Label`-Instanz zur Anzeige des Prozessnamens 54

Abb. 7-7: Visualisierung der Verwendung von `EditorLabel` und `RewriterService` im Designer 55

Abb. 7-8: Visualisierung zweier Prozessverwendungen mit angehängten Daten im Designer 56

Abb. 7-9: Visualisierung mehrerer Kontrollflüsse im Designer zwischen vier Prozessen und einem AND-Konnektor 58

Abb. 7-10: Repräsentation eines Attribut-basierten Links in Domain-, Editor-Usage- und Graphical-Usage-Model 58

Abb. 7-11: Visualisierung von zwei Datenflüssen im Designer 59

Abb. 8-1: Übersicht über die Architektur des Model Designer Frameworks 61

Abb. 8-2: Auf die Kernbestandteile reduziertes Klassendiagram des Model Designer Frameworks 63

Abb. 8-3: Anzeige der Eigenschaften einer selektierten Prozess-Verwendung in der Properties-View von Eclipse 66

Abb. 8-4: Erforderliche Benutzerinteraktion zum Anlegen eines Knotens 67

Abb. 8-5: Aktivitätsdiagramm der einzelnen Schritte beim Erzeugen einer `Node`-Verwendung 68

Abb. 8-6: Erforderliche Benutzerinteraktion zum Anlegen einer Verbindung 68

Abb. 8-7: Aktivitätsdiagramm der einzelnen Schritte beim Erzeugen einer `Link`-Verwendung 69

Abb. B-1: Verzeichnisstruktur der CD 85

Abb. B-2: Prozess-Designer mit Beispielprozess "Frage in Forum stellen" 85

Abb. B-3: Enity-Relationship-Designer mit ER-Schema einer Personalverwaltung 86

Listings

Listing 2-1: Beispiel einer LMM-Instanz mit zwei Meta-Ebenen 10

Listing 7-1: Definition des Prozess-`Diagram` 50

Listing 7-2: Definition der Prozess-`Figure` 51

Listing 7-3: Definition des Prozess-`Nodes` inkl. eines zugehörigen `NodeEmbeddings` 52

Listing 7-4: Definition des Prozess-`Labels` als Sub-Element von `ProcessNode` 54

Listing 7-5: Definition eines `RewriterService` als Sub-Element von `ProcessNode` 55

Listing 7-6: Definition des `Compartments` für Eingangsdaten als Sub-Element von `ProcessNode` 56

Listing 7-7: Definition der Kontrollfluss-`Figure` 57

Listing 7-8: Definition des Kontrollfluss-`Links` 58

Listing 7-9: Definition des Datenfluss-`Links` 59

Abkürzungsverzeichnis

Abkürzung	Bedeutung
AToM^3	A Tool for Multi-Formalism and Meta-Modelling
BPMN	Business Process Modeling Notation
DSL	Domain-specific Language (dt.: domänenspezifische Sprache)
EMF	Eclipse Modeling Framework
FQN	voll qualifizierter Name
GEF	Graphical Editing Framework
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
i>PM	integrated Process Manager
JAR	Java-Archiv
LMM	linguistisches Meta-Modell
LWS	LightweightSystem
MDF	Model Designer Framework
MOF	Meta Object Facility
MVC	Model View Controller
OMG	Object Management Group
oMME	Open Meta-Modeling Environment
POPM	perspektivenorientierte Prozessmodellierung
SWT	Standard Widget Toolkit

Kapitel 1 Einleitung

Eine oft benutzte Floskel, die die Natur des Geschäftsprozess-Managements treffend charakterisiert, ist „The only constant is change“ (dt.: „Die einzige Konstante ist der Wandel“) [1]. Änderungen können in der Tat in allen Bereichen des Prozess-Managements auftreten, angefangen bei Prozessen in der Ausführung, über Prozess-Modelle bis hin zur verwendeten Modellierungssprache. Die Modifikation von Prozess-Modellen ist der Standardfall, der von allen Modellierungsumgebungen unterstützt wird [2]. Bei plötzlich auftretenden Änderungen innerhalb der Anwendung kann es darüber hinaus erforderlich sein, Anpassungen an laufenden Prozessinstanzen durchzuführen [3]. Der Fokus dieser Arbeit liegt jedoch auf dem zuletzt genannten Fall, der Adaption der Modellierungssprache.

Argumente, weshalb es sinnvoll sein kann derartige Modifikationen vorzunehmen, liefern Clark, Sammut und Willans in [4]. Darin beziehen sie sich zwar auf das Adaptieren von Programmiersprachen, doch lassen sich ihre Erkenntnisse auch auf die Prozessmodellierung übertragen. Als Hauptargument führen sie auf, dass die richtige Sprache es dem Entwickler ermöglicht komplexe Sachverhalte kurz und prägnant darstellen zu können, wodurch eine signifikant höhere Produktivität erreicht wird. Deshalb ist es vorteilhaft für bestimmte Anwendungsdomänen eigene domänenspezifische Sprachen (kurz: DSL) formalisieren bzw. existierende Sprachen an geänderte Anforderungen anpassen zu können. Das Erstellen eigener DSLs soll ohne tiefgehende Programmierkenntnisse möglich sein und gut durch Werkzeuge unterstützt werden. Die Autoren von [1], [4] und [5] folgen dazu dem vielversprechenden Ansatz der Meta-Modellierung, auf den wir uns ebenfalls konzentrieren.

1.1 Problemstellung und Zielsetzung

Mit Hilfe von Meta-Modellen können beliebige Anwendungsdomänen beschrieben werden, d.h. ein Meta-Modell spezifiziert eine Sprache mittels der ein Sachverhalt für die zugehörige Domäne ausgedrückt werden kann. Die genaue Syntax wird dadurch aber noch nicht festgelegt, sondern muss in einem weiteren Schritt definiert werden.

Zur Spezifikation einer konkreten grafischen Syntax für ein gegebenes Meta-Modell existieren zahlreiche Frameworks. Diese Syntax ist dabei stets die Grundlage für einen Editor, der zum Erstellen von Modellen in der durch das Meta-Modell vorgegebenen domänenspezifischen Sprache dient. Allerdings lässt sich keines der Systeme dazu verwenden, ein Prozessmodellierungswerkzeug zu definieren, das auf der perspektivenorientierte Prozessmodellierung (Abschnitt 2.4) basiert, da die Unterstützung des Typ-Verwendungs-Konzepts fehlt. Infolgedessen besteht das Ziel der vorliegenden Arbeit darin, ein Framework zu realisieren, das folgende Anforderungen erfüllt:

(a) Integration in eine sich im Aufbau befindende Meta-Modellierungsumgebung

(b) Spezifikation von Domäne und grafischem Editor durch Meta-Modelle

(c) Auswahlmöglichkeit des Modellierungsansatzes: Typ-Verwendungs-Konzept oder klassische Instanz-Erzeugung

(d) Abbildung von Modellelementen auf grafische Objekte

(e) Abbildung von Beziehungen zwischen Modellelementen auf Hierarchien grafischer Objekte

(f) Repräsentation von Beziehungen zwischen Modellelementen durch die Verbindung der entsprechenden grafischen Objekte mit Linien

(g) Visualisierung des Zustands eines Modellelements innerhalb des zugehörigen grafischen Objekts

(h) Möglichkeit zur benutzerdefinierten Spezifikation grafischer Objekte

(i) Direkte Übernahme von Änderungen an den Meta-Modellen in die jeweiligen Editor-Instanzen, ohne dass ein Neustart des Programms notwendig ist

Der Vorteil von Anforderung (b) ist, dass der Anwender durch das Spezifizieren des Domänen-Meta-Modells bereits mit den einzelnen Werkzeugen und Prozessen zur Erstellung von Meta-Modellen vertraut ist und somit die Beschreibung des Editors in analoger Weise vornehmen kann. Durch die Auswahl des Modellierungsansatzes kann sowohl das Typ-Verwendungs-Konzept als auch die herkömmliche Modellierungsweise – wie sie von allen anderen Systemen unterstützt wird – eingesetzt werden. Letztendlich kapselt der resultierende Editor eine determinierte Sicht auf das konstruierte Modell und visualisiert die einzelnen Modellelemente sowie deren Beziehungen untereinander in einer wohldefinierten Art und Weise. Demnach können zu einem vorhandenen Meta-Modell beliebig viele Sichten definiert werden, die den Inhalt in der für die jeweilige Anwendungsdomäne passenden Darstellung anzeigen. Diese Abbildung zwischen der Domäne und einer grafischen Repräsentation wird durch die Anforderungen (d) bis (g) abgedeckt. Die genaue grafische Repräsentation muss jedoch beliebig durch den Benutzer festgelegt werden können.

1.2 Aufbau der Arbeit

Nach einer kurzen Einführung im ersten Kapitel befasst sich Kapitel 2 mit den grundlegenden Begriffen und Konzepten dieser Arbeit. Außerdem wird darin das Open Meta Modelling Environment (kurz: oMME) vorgestellt, in dessen Kontext die vorliegende Master-Thesis einzuordnen ist.

Im Anschluss daran werden in Kapitel 3 verschiedene Systeme analysiert, die sich ebenso mit diesem Themenkomplex befassen. Das sind einerseits bereits verfügbare grafische Werkzeuge zur Prozessmodellierung, andererseits handelt es sich um Frameworks zur Erstellung grafischer Modellierungs-Tools auf der Basis von Meta-Modellen. Zuletzt werden Programmierbibliotheken und Frameworks der Eclipse IDE evaluiert, die sich zur Implementierung grafischer Editoren eignen.

In Kapitel 4 erfolgt die Vorstellung aller Technologien, die in dieser Arbeit zum Einsatz kommen. Das sind einerseits die Eclipse IDE (oMME baut darauf auf) und andererseits das Graphical Editing Framework (kurz: GEF) zusammen mit der Visualisierungskomponente Draw2D.

Kapitel 5 gibt einen Überblick über das Zusammenspiel der einzelnen Modelle und Meta-Modelle, die sowohl Funktionalität und Verhalten eines grafischen Editors als auch den damit visualisierten Inhalt eines Domänen-Modells beschreiben. Hierbei handelt es sich um die konzeptuelle Vernetzung der verschiedenen Modelle, die gänzlich unabhängig von einer Implementierung ist.

Daraufhin thematisiert Kapitel 6 die drei Meta-Modelle, die jeder Definition eines grafischen Editors zugrunde liegen. Zusätzlich werden hierfür mögliche Mappings identifiziert, die angeben, wie verschiedene Sachverhalte der Domäne auf eine aussagekräftige Darstellung abgebildet werden können.

Der in Kapitel 7 präsentierte Anwendungsfall „Prozessmodellierung“ demonstriert die Vorgehensweise zur Definition eines grafischen Editors. Dabei kommen alle in Kapitel 6 identifizierten Mappings zum Einsatz. Ferner wird zum Schluss die Allgemeingültigkeit dieses Anwendungsfalls dargelegt.

Im letzten Kapitel erfolgt eine detaillierte Vorstellung der als Model Designer Framework bezeichneten Implementierung, die die bereits erwähnten Modelle interpretiert und darauf basierend einen grafischen Editor zur Verfügung stellt. Abschließend wird ein Ausblick auf potentielle Erweiterungen dieses Systems gegeben, aus denen eine höhere Benutzerfreundlichkeit und ein vergrößertes Einsatzgebiet resultieren.

Metasprachlich verwendete Begriffe, d.h. bei denen es um den Begriff selbst und nicht um dessen Bedeutung geht, und solche die sich auf Elemente aus Abbildungen beziehen werden in dieser Arbeit *kursiv* gedruckt. Des Weiteren werden sämtliche Bezeichner, die Elemente von Modellen und Meta-Modellen benennen, durch `äquidistante Schrift` hervorgehoben.

Kapitel 2 Grundlagen

Dieses Kapitel vermittelt grundlegende Kenntnisse, die zum Verständnis der weiteren Ausführungen benötigt werden. Dazu zählt eine Definition des Begriffs des Meta-Modells und eine Einführung in das Open Meta-Modeling Environment [6], das die Basisplattform für die zur vorliegenden Masterarbeit gehörenden Implementierung darstellt. Hinzu kommen eine Klassifikation visueller Sprachen sowie die Vorstellung der perspektivenorientierten Prozessmodellierung [7].

2.1 Meta-Modellierung

Für den Begriff des Meta-Modells existieren in der Literatur zahlreiche Definitionen. Seidewitz definiert ihn wie folgt:

> *„A metamodel is a specification model for a class of SUS[1] where each SUS in the class is itself a valid model expressed in a certain modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language."* [8]

Demzufolge handelt es sich bei einem Meta-Modell um ein Modell, das Informationen über eine Klasse von Modellen bereitstellt. Es beschreibt somit die verwendbaren Elemente innerhalb einer Modellklasse inkl. aller möglichen Beziehungen zwischen diesen Elementen. Das Verhalten entspricht der Spezifikation einer bestimmten Modellierungssprache. Eine andere Definition von Ober und Prinz geht auf das Prinzip der Sprache etwas näher ein:

> *„A meta-model is a description of the concepts the modelling language provides without caring on the actual syntax. A particular kind of meta-model is an abstract grammar."* [9]

Die spezifizierte Modellierungssprache wird demnach lediglich in Form einer abstrakten Grammatik definiert, d.h. es gibt keine konkrete textuelle oder grafische Notation. Sie muss – falls benötigt – in einem weiteren Schritt konkretisiert und an das Meta-Modell angebunden werden.

2.2 Open Meta Modelling Environment

Das Open Meta Modelling Environment [6] ist eine Plattform zur Entwicklung, Bearbeitung und Visualisierung von Meta-Modellen. Sie basiert auf der Eclipse IDE [10], die in Abschnitt 4.1 näher vorgestellt wird, und verwendet im Zuge dessen das Eclipse Modeling Framework (kurz: *EMF*) [11]. *EMF* dient ebenfalls zur Erstellung von Meta-Modellen, wobei als Meta-Meta-Modell *Ecore* zum Einsatz kommt. *Ecore* ist zwar an den MOF-Standard [12] (Abschnitt 2.2.1) angelehnt, implementiert ihn jedoch nicht vollständig [13]. Wie Abb. 2-1 entnommen werden kann ist das linguistische Meta-Modell (kurz: *LMM*) mit Hilfe von *EMF* modelliert. Das *LMM* wird in Form einer abstrakten Syntax [4] beschrieben, d.h. es existiert lediglich die Spezifikation aller in einer *LMM*-Instanz verwendbaren Elemente sowie der möglichen Beziehungen zwischen ihnen. Die Definition einer konkreten Syntax erfolgt durch das Modul *Textual Model Editor*. Für dieses existiert eine Xtext-Grammatik [14] (konkrete Syntax), die anhand des mit *Ecore* formulierten *LMM*s spezifiziert ist und zu der Listing 2-1 ein Anwendungsbeispiel zeigt. Das linguistische Meta-Modell wird in Abschnitt 2.2.3 eingehend behandelt. Für das weitere Verständnis genügt die Information, dass eine *LMM*-Instanz zur Beschreibung beliebiger Modelle inkl. der zugehörigen Meta-Modelle dient.

[1] SUS = System under Study: System, das "untersucht" bzw. modelliert / nachgebildet werden soll

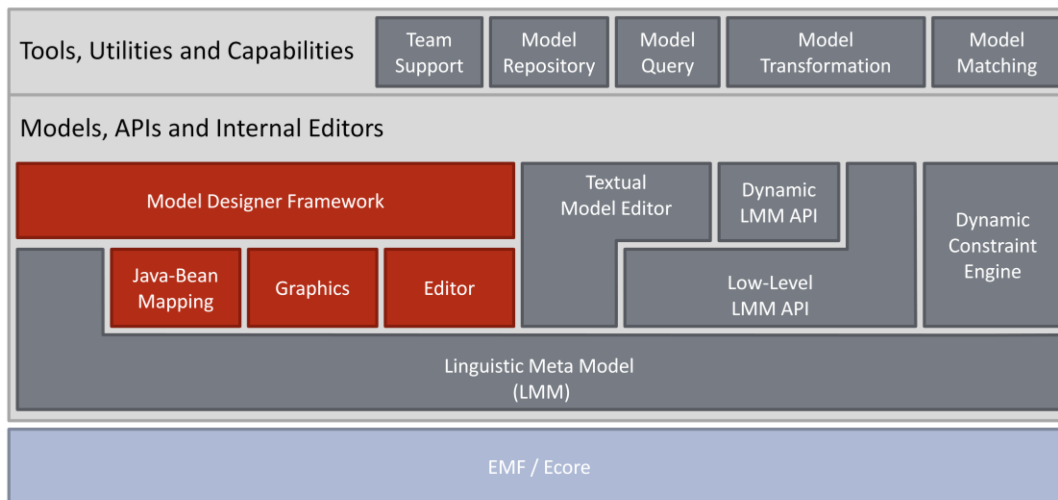

Abb. 2-1: Architektur der oMME-Plattform [6]

Im Rahmen der vorliegenden Arbeit soll die als *Model Designer Framework* (kurz: MDF) bezeichnete Komponente entwickelt werden (rot hinterlegt). Sie interpretiert – im Gegensatz zum *Textual Model Editor* – den Inhalt von mittels LMM formulierten Meta-Modellen (*Java-Bean Mapping*, *Graphics*, *Editor*), die in diesem Zusammenhang ebenfalls eine abstrakte Syntax darstellen (auch rot hinterlegt). Mit Hilfe des MDFs kann nun ein grafischer Editor definiert werden, der die zu einem Meta-Modell gehörende konkrete (grafische) Syntax kapselt. Beim Benutzen eines solchen Editors entstehen somit Modelle auf Basis des dem Editor zugrunde liegenden Meta-Modells.

2.2.1 Meta Object Facility

Nachdem das LMM durch ein an die Meta Object Facility (kurz: MOF) [12] angelehntes Meta-Meta-Modell (Ecore) beschrieben wird und zudem darin einige Modellierungsmuster von MOF Verwendung finden, erfolgt eine kurze Einführung in diesen Standard. Er wurde von der Object Management Group (kurz: OMG) [15] entwickelt und stellt einen Ansatz zur Verwaltung von Metadaten dar. Die Daten eines Systems werden als Metadaten-Architektur zusammengefasst und hierbei in Form von implizit zueinander in Beziehung stehenden Modellen wiedergegeben. Die impliziten Beziehungen zwischen zwei Modellen entstehen durch explizite *instanceOf*-Referenzen zwischen Elementen beider Modelle. In der Literatur wird ein Modell oftmals als Ebene bezeichnet. Der Begriff „Ebene" wird eingeführt, da er das Konzept einer mehrschichtigen Architektur am besten beschreibt. Im Kontext von Meta-Modellierung findet man demzufolge häufig eine geschichtete Darstellung von Modellen, bei denen Elemente einer niedrigen Ebene `A` Instanzen von Elementen der nächsthöheren Ebene `B` sind. Mit anderen Worten ausgedrückt repräsentiert der Inhalt von Ebene `B` das Meta-Modell zum durch Ebene `A` beschriebenen Modell.

Wie eingangs erwähnt wird jedes Meta-Modell von einem weiteren Meta-Modell – dem sog. Meta-Meta-Modell – definiert. Prinzipiell lässt sich dieser Prozess beliebig oft wiederholen, wodurch eine unendliche Abfolge von Meta-Ebenen entsteht. Für die oberste Ebene (auch MOF-Ebene genannt) gilt die Bedingung der Selbstbeschreibung, d.h. ihre Elemente sind gleichzeitig Instanzen von sich selbst. Zusätzlich muss eine Reihe weiterer Restriktionen erfüllt sein, die sowohl in [12] als auch in [16] nachzulesen sind. Zum Beispiel muss jedes Element einer Ebene `n` eine Instanz eines Elements der Ebene `n - 1` sein. Referenzen in umgekehrter Richtung sind nicht erlaubt. Die Benennung bzw. Nummerierung der einzelnen Ebenen geschieht absteigend, wobei die MOF-Ebene als `M3` bezeichnet wird. `M0` als unterste Ebene beinhaltet letztendlich Objekte, die eine Entsprechung in der (virtuellen) Realität aufweisen.

Ein stark vereinfachtes Beispielszenario für eine Meta-Ebenen-Hierarchie illustriert Abb. 2-2. Vereinfacht bedeutet in dem Zusammenhang, dass `M3` auf ein Element namens `Class` reduziert wurde. Zur Erfüllung der MOF-Spezifikation müssten darin jedoch alle Elemente enthalten sein, die die minimale Version des Standards – das sog. *Essential MOF* [12] – vorgibt. Die Tatsache, dass die drei unteren Ebenen frei definierbar sind, wurde in Abb. 2-2 voll ausgeschöpft.

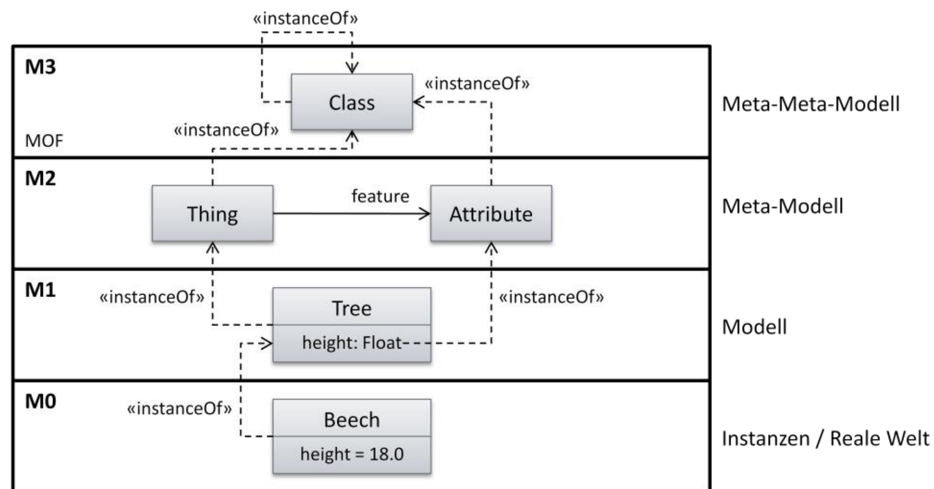

Abb. 2-2: Beispiel einer Meta-Modell-Hierarchie in Anlehnung an MOF

2.2.2 Erweiterte Modellierungsmuster

Die im Kontext dieser Arbeit benötigten erweiterten Modellierungsmuster erfahren in den nachfolgenden Ausführungen eine detaillierte Betrachtung. Sämtliche neuartige Modellierungsmuster werden darüber hinaus umfassend in [6] erklärt. Einige davon lassen sich durchaus mit Hilfe etablierter Vorgehensweisen umsetzen, was allerdings negative Auswirkungen auf die Lesbarkeit von Modellen hat. Entweder ist die Angabe entsprechend ausführlicher, formaler Constraints oder die Ergänzung informaler Beschreibungen erforderlich. Insbesondere die zuletzt genannte Strategie erschwert oder verhindert sogar die vollständige Computer-gestützte Interpretation von Modellen.

2.2.2.1 Clabjects

Das Akronym „Clabject“ setzt sich aus den beiden Begriffen „Class“ und „Object“ zusammen und drückt aus, dass ein Element mit dieser Nomenklatur sowohl eine Klasse als auch ein Objekt darstellt. Nach der Definition von Atkinson und Kühne in [17] wird der Klassen-Teil als Typ-Facette und der Objekt-Teil als Instanz-Facette bezeichnet. Innerhalb der Typ-Facette erfolgt die Deklaration von Attributen, während der Inhalt der Instanz-Facette aus der *instanceOf*-Relation des Clabjects zu seinem Typ resultiert. Letztere enthält somit Wertzuweisungen an Attribute, die den Typ-Facetten des mit *instanceOf* referenzierten Clabjects zuzüglich aller Generalisierungen entstammen. Der Grund für die Einführung dieses Modellierungsmusters beruht darauf, dass ein Element einer Ebene `N` die Instanz eines Elements aus Ebene `N + 1` ist, aber gleichzeitig der Typ eines Elements aus Ebene `N - 1`.

2.2.2.2 Orthogonale Klassifikation

Das von Atkinson und Kühne entwickelte Konzept der Orthogonalen Klassifikation [18] erlaubt die Einordnung von *instanceOf*-Referenzen in zwei orthogonal zueinander stehenden Meta-Modell Stacks [1]. Der linguistische Meta-Modell Stack beinhaltet ein Meta-Modell, das beschreibt, wie Meta-Modell-Hierarchien der Anwendungsdomäne abzuspeichern sind. Ein dazu orthogonaler logischer Meta-Modell Stack hingegen umfasst beliebig viele Modelle, die Beziehungen untereinander

ausschließlich über ihren Inhalt ausdrücken. Alle Elemente des logischen Stacks sind demnach Instanzen von Elementen des linguistischen Meta-Modells. Gleichzeitig kann ein Element des logischen Stacks eine Instanz eines anderen Elements des logischen Stacks darstellen.

Zum besseren Verständnis und in Anlehnung an das in Abschnitt 2.2.3 beschriebene linguistische Meta-Modell zeigt Abb. 2-3 ein Beispiel der Orthogonalen Klassifikation. Aus Gründen der Übersichtlichkeit wurde auf die *instanceOf*-Bezeichnung bei allen gestrichelten Pfeilen verzichtet. Das in der Grafik als `Assignment` benannte Element entspricht einer Wert-Zuweisung an ein vorher deklariertes Attribut. Analog dazu besitzt auch das `height`-Attribut einen entsprechenden Typ auf Ebene `L1` des linguistischen Meta-Modell Stacks. Das Clabject auf Ebene `M0` besitzt die zwei eingangs erwähnten Typen der *instanceOf*-Referenz, und zwar einerseits auf `Concept` und andererseits auf `Tree`. Als Fazit bleibt, dass der Inhalt des logischen Meta-Modell Stacks dank des vorgestellten Modellierungsmusters frei definiert werden kann.

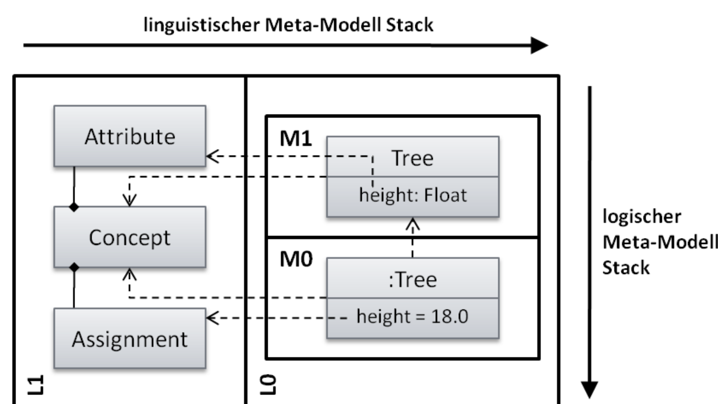

Abb. 2-3: Beispiel eines orthogonal klassifizierten Meta-Modells

2.2.2.3 Spezialisierung von Instanzen

Volz erklärt in [6] ausführlich, wieso die Möglichkeit Instanzen spezialisieren zu können, in bestimmten Fällen die Arbeit des Modellierers erleichtert. Zudem entstehen dabei leichter verständliche Modelle, als bei der Beschränkung auf herkömmliche Modellierungsmuster, da die Komplexitätssteigerung in einer Zunahme der Anzahl notwendiger Modell-Konstrukte begründet liegt. Ein konkreter Anwendungsfall ist die übersichtliche Realisierbarkeit des sog. Typ-Verwendungs-Konzepts, das in Abschnitt 2.4 detailliert behandelt wird.

Die aus objektorientierten Programmiersprachen bekannte Generalisierung / Spezialisierung beschränkt sich auf die Typfacette der beteiligten Clabjects. Analog dazu lässt sich eine Spezialisierung der Instanz-Facette definieren, die bereits zugewiesene Werte an das spezialisierende Clabject delegiert und zugleich festlegt, in welcher Weise diese Werte darin überschrieben werden dürfen. In [6] werden die möglichen Verhaltensweisen für das Überschreiben von Wertzuweisungen wie folgt definiert:

1. Die spezialisierende Instanz darf den Wert eines Attributs nicht überschreiben. Dies ist der Normalfall und entspricht der von der UML [19] her bekannten Semantik.
2. Die spezialisierende Instanz darf den innerhalb der allgemeinen Instanz gesetzten Wert überschreiben.
3. Die spezialisierende Instanz darf einen neuen Wert an den alten anhängen oder voranstellen. Dies ist nur für Attribute sinnvoll, die eine Zeichenkette aufnehmen.

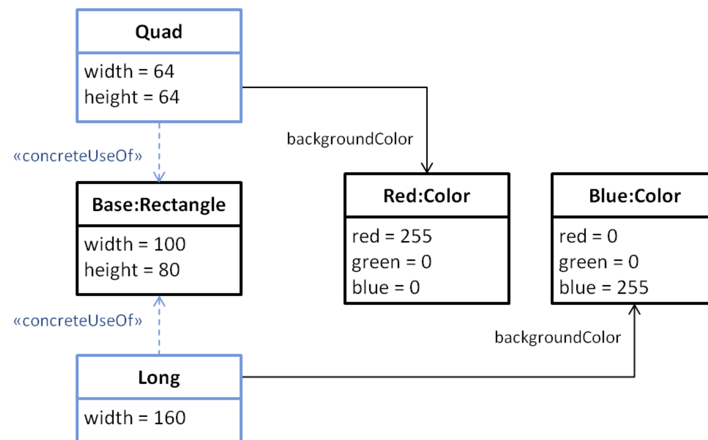

Abb. 2-4: Beispiel für die Spezialisierung von Instanzen

Abb. 2-4 illustriert ein einfaches Beispielmodell, innerhalb dessen die Instanz-Spezialisierung zum Einsatz kommt. Als Voraussetzung gilt, dass das zugehörige Meta-Modell sowohl eine Klasse `Rectangle` als auch eine Klasse `Color` zur Verfügung stellt. Das Hauptaugenmerk liegt dabei auf der erstgenannten Klasse, die zwei Integer-Attribute `width` und `height` sowie ein Referenz-Attribut auf die Klasse `Color` mit dem Namen `backgroundColor` besitzt. Außerdem dürfen alle drei Attribute im Rahmen der Instanz-Spezialisierung überschrieben werden. Im abgebildeten Modell gibt es eine direkte Instanz der `Rectangle`-Klasse (schwarz umrandet) und zwei abgeleitete Instanzen, die auf diese direkte Instanz mittels `concreteUseOf` verweisen (blau umrandet). Eine derartige Beziehung bedeutet die Spezialisierung der Instanz-Facette des referenzierten Clabjects. Von Interesse ist nun, welche Werte die einzelnen Attribute des mit `Long` betitelten Rechtecks aufweisen. `width` wird mit dem Wert 160 überschrieben und `backgroundColor` erhält eine Referenz auf `Blue`. Das Attribut `height` hingegen besitzt weiterhin den Wert 80, da dieser im `Base`-Rechteck zugewiesen und in `Long` nicht redefiniert wird.

2.2.3 Das linguistische Meta-Modell

Das ebenfalls von Volz in [6] eingeführte linguistische Meta-Modell spezifiziert den Inhalt der `L1`-Ebene des linguistischen Meta-Modell Stacks (Abb. 2-3). Ziel dieses Meta-Modells ist das Anbieten einer Sprache zur flexiblen Definition logischer Meta-Modell Stacks bei gleichzeitiger Unterstützung der erweiterten Modellierungsmuster (Abschnitt 2.2.2).

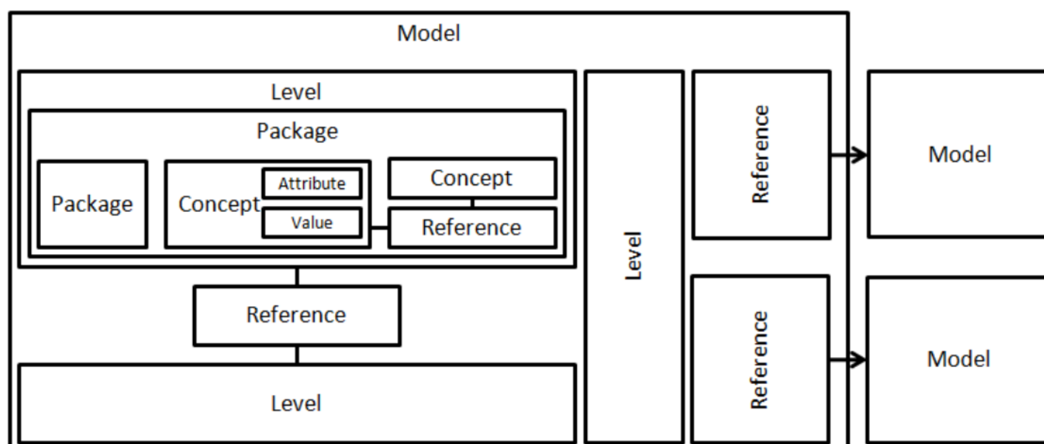

Abb. 2-5: Exemplarischer Aufbau eines Modells konform zur Spezifikation des LMM in abstrakter Syntax [6]

Der Aufbau des LMM ist in Abb. 2-5 exemplarisch skizziert und gibt die abstrakte Syntax wieder. Das Wurzelelement ist stets ein *Model*, das wiederum Referenzen zu anderen *Models* enthält. Ferner

umfasst es beliebig viele *Levels*, die sich ebenfalls gegenseitig referenzieren können. Eine derartige Referenz ist allerdings nicht zwangsweise vom Typ *instanceOf* (siehe MOF in Abschnitt 2.2.1), sondern drückt im allgemeinen Fall aus, dass Elemente des referenzierten *Levels* im Ursprungslevel verwendet werden dürfen. Zur Strukturierung des Inhalts beinhaltet ein *Level* beliebig viele *Packages*, mindestens jedoch eines. Sie können wiederum untergeordnete *Packages* enthalten. Darüber hinaus werden innerhalb eines *Packages* sog. *Concepts* definiert, die *Clabjects* repräsentieren. Deshalb verfügen sie bei Bedarf über eine Typ- (*Attribute*) und/oder Instanz-Facette (*Value*). Ein *Concept* kann Verweise auf andere *Concepts* besitzen, womit sich Beziehungen wie *instanceOf*, *extends* oder *concreteUseOf* ausdrücken lassen.

```
model Example {
     modelURI = "http://www.ai4.uni-bayreuth.de/omme/mdf/Listing1.lmm";

     level M1 {
          package P {
               concept Figure {
                    attributes {
                         concept Figure children { multiplicity = zeroOrMore; }
                    }
               }

               enum LineStyle {
                    SOLID; DASH; DOT;
               }

               concept Rectangle extends Figure {
                    attributes {
                         integer width;
                         integer height;
                         enum LineStyle style { default = Example.M1.P.LineStyle.SOLID; }
                    }
               }
          }
     }

     level M0 instanceOf M1 {
          package P {
               concept BaseRect instanceOf Example.M1.P.Rectangle {
                    values {
                         width = 100 { onUpdate = overwrite; }
                         height = 80 { onUpdate = overwrite; }
                         enum style = DASH;
                    }
               }

               concept ConcreteRect concreteUseOf BaseRect {
                    values {
                         width = 160;
                         height = 120;
                         concept children = BaseRect;
                    }
               }
          }
     }
}
```

Listing 2-1: Beispiel einer LMM-Instanz mit zwei Meta-Ebenen

Neben *Concepts* können *Packages* auch *Enumerations* enthalten, die sich aus *EnumerationsLiterals* zusammensetzen und analog zur Bedeutung in diversen Programmiersprachen als Aufzählungstyp gelten. Sie gehören gemeinsam mit den *Concepts* zu den Referenztypen, da sie in einer *Attribute*-Deklaration als Typ eingesetzt werden können. *Attributes* können außerdem auf Basis eines literalen Typs definiert sein, z.B. Integer, Double, Boolean, String, FQN oder UUID. Ein FQN kapselt einen *full qualified name*, mit dem beispielsweise ein beliebiges Modellelement adressiert werden kann. Im LMM wird er in Form mehrerer Namen angegeben, die durch Punkte voneinander getrennt sind. Der

UUID-Typ dient zur Deklaration von *Attributen*, die einen *Universally Unique Identifier* [20] aufnehmen können. Damit ist die Möglichkeit gegeben Identifikator-*Attribute* zu spezifizieren. Basierend auf dem Typ eines *Attributs* können in der Instanz-Facette bestimmte *Values* zugewiesen werden.

Zur Erstellung von LMM-Instanzen, die in `L0` des linguistischen Stacks anzusiedeln sind, existiert eine domänenspezifische Sprache definiert durch eine konkrete Grammatik. Ein Beispiel zur Veranschaulichung einer derartigen Instanz ist in Listing 2-1 abgedruckt. Es zeigt zwei Ebenen, die durch eine *instanceOf*-Beziehung miteinander verbunden sind. `M0` beinhaltet zwei Instanzen vom Typ `Rectangle`, wobei `ConcreteRect` die Instanz-Facette von `BaseRect` spezialisiert. Hierbei werden Breite und Höhe des Rechtecks neu gesetzt und `BaseRect` als Kind-`Figure` deklariert. Dem `style`-Attribut hingegen darf an der Stelle kein neuer Wert zugeteilt werden, da dies vom Standard-Verhalten der Instanz-Spezialisierung verboten wird. Zur Aufhebung dieser Restriktion ist beim *onUpdate*-Parameter der entsprechenden Zuweisung innerhalb von `BaseRect` ebenfalls *overwrite* anzugeben. Analog zu den Parametern bei der Wertzuweisung verfügen auch Attribute über verschiedene Parameter. Als wichtigster Vertreter gilt sicherlich die *Multiplicity*-Eigenschaft, die einen der vier Werte *zeroOrOne*, *one*, *zeroOrMore* oder *oneOrMore* annehmen kann. Außerdem darf für alle Attribute, mit Ausnahme derjenigen, die ein *Conzept* referenzieren, ein *Default*-Wert gesetzt werden. Weitere Attribut-Parameter werden – sofern benötigt – im späteren Verlauf dieser Arbeit vorgestellt.

2.3 Klassifikation visueller Sprachen

Eine textuelle Sprache besteht aus einer Menge von Wörtern, wohingegen eine visuelle Sprache über eine Reihe grafischer Formen definiert ist [21]. Beim Aneinanderreihen von Wörtern entstehen Sätze oder gar Texte mit semantischem Charakter. Gleiches gilt für grafische Formen, sofern man diese zueinander in Beziehung setzt. Die Klassifikation visueller Sprachen erfolgt nach der Art, wie grafische Elemente zueinander in Relation stehen können. In [22] werden nach diesem Kriterium drei Klassen identifiziert.

Die *Verbindungs-* oder *Graph-basierte* Klasse spezifiziert Beziehungen zwischen Formen durch Verbindungen zwischen diesen Elementen. Das Resultat ist demnach ein Graph mit Knoten und Kanten (Abb. 2-6, links).

Innerhalb der *Geometrie-basierten* Klasse werden Beziehungen durch die räumliche Lage der Formen zueinander ausgedrückt. Der rechte Teil von Abb. 2-6 zeigt zwei Darstellungsweisen der Geometrie-basierten Variante. Einerseits können Elemente in Relation stehen, wenn sie aneinander anhaften (*A* an *C*), oder wenn sie ineinander verschachtelt sind (*B* in *A* und *D* in *C*). Eine gerichtete Verbindung drückt im Beispiel eine Enthaltensein-Beziehung aus, wohingegen eine ungerichtete Verbindung die Aneinanderreihung der verbundenen Elemente wiedergibt.

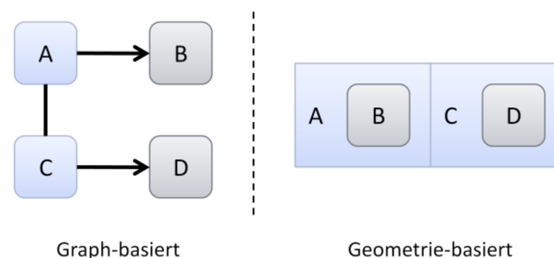

Abb. 2-6: Graph- und Geometrie-basierte Visualisierung desselben Sachverhalts

Die dritte als *Hybrid* bezeichnete Klasse ist eine Kombination der Graph-basierten und der Geometrie-basierten Variante. Sie bietet die größtmögliche Flexibilität und Ausdrucksstärke bei der Visualisierung und wird deswegen bevorzugt eingesetzt (z.B. in der UML[2] [19]). Gleichzeitig resultiert daraus eine erhöhte Komplexität der Implementierung, so dass man sich bei universitären Projekten (Abschnitte 3.2.3 und 3.2.4) häufig auf die Graph-basierte Klasse beschränkt.

2.4 Perspektivenorientierte Prozessmodellierung

Zur Erreichung einer möglichst hohen Flexibilität bei Prozessmodellen wurde die perspektivenorientierte Prozessmodellierung entwickelt. Dabei wird ein Prozess durch verschiedene Perspektiven repräsentiert, die je nach Bedarf erweiterbar sind. Im kommenden Abschnitt wird dieses in [7] eingeführte Modell näher erläutert, wobei der Großteil der Erläuterungen aus [23] stammt.

- **Funktionale Perspektive:** Die Funktionale Perspektive deklariert die funktionalen Einheiten. Zu unterscheiden gilt es zwischen elementaren Prozessen, die keine weiteren Subprozesse beinhalten, und kompositen Prozessen, die sich wiederaum aus anderen Prozessen – sowohl elementar als auch komposit – zusammensetzen.
- **Datenbezogene Perspektive:** Damit wird der Aufbau aller vom Prozess verwendeten Daten beschrieben (z.B. in Form eines ER-Diagramms). Ferner wird hier festgelegt, welche Eingangsdaten ein Prozess konsumiert und welche Ausgangsdaten er produziert. Schließlich erfolgt noch eine Konkretisierung des Datenflusses innerhalb des Prozesses, d.h. auf welchem Weg gelangen die Daten vom Eingang zum Ausgang und wie werden sie dabei verarbeitet.
- **Verhaltensbezogene Perspektive:** Die Verhaltensbezogene Perspektive legt eine mögliche Ausführungsreihenfolge der funktionalen Einheiten fest, den sog. Kontrollfluss. Denkbar sind dabei eine strikte und komplett vorgegebene Reihenfolge bis hin zu einer völlig freien Auswahl des Ablaufs. Der Kontrollfluss kann z.B. sequentiell, unabhängig oder zyklisch verlaufen.
- **Operationale Perspektive:** Sie beschreibt die Werkzeuge bzw. Programme für die Ausführung und erfolgreiche Beendigung der jeweiligen funktionalen Einheit. Das kann beispielsweise Microsoft Excel als Tabellenkalkulation oder Adobe Photoshop für die Grafikbearbeitung sein.
- **Organisatorische Perspektive:** Mit Hilfe der Organisatorischen Perspektive wird festgelegt, wer für die Ausführung eines bestimmten Prozesses verantwortlich ist. Folglich wird eine Rechteverwaltung benötigt, die einzelnen Benutzern oder Gruppen entsprechende Ausführungsrechte gewährt.

Typ-Verwendungs-Konzept

Das Typ-Verwendungs-Konzept steht zwar nicht in direktem Zusammenhang mit der perspektivenorientierten Prozessmodellierung (kurz: POPM), wird aber von dem Graph-basierten (Abschnitt 2.3) Prozessmodellierungswerkzeug i>ProcessManager (kurz: i>PM) [24] implementiert, das die POPM-Modellierungsweise umsetzt. Die von i>PM bereitgestellte Funktionalität soll auch durch ein grafisches Modellierungswerkzeug abgedeckt werden, das mittels der zur vorliegenden Arbeit gehörenden Implementierung erstellt werden kann. Deshalb ist die Anwendung des Typ-Verwendungs-Konzepts eine grundlegende Fähigkeit, die es zu erfüllen gilt.

[2] In einem Klassendiagramm können Klassen sich gegenseitig referenzieren und zugleich in verschiedenen Paketen abgelegt werden.

Das Konzept lässt sich am einfachsten anhand eines Beispielprozesses erläutern, das aus [6] stammt und in Abb. 2-7 dargestellt ist. Darin wird ein bereits definierter Prozesstyp *„Notiz anlegen / ergänzen“* verwendet, der unter Benutzung der beiden Anwendungen *„SR Office“* und *„SRO 24“* auszuführen ist. Ein anderer Prozess (Abb. 2-8) benötigt eine ähnliche Sub-Funktionalität, wobei als Anwendung lediglich *„SRO 24“* eingesetzt werden darf. Die geänderte Funktionsweise wird durch die Anpassung des Prozessnamens in *„Notiz aufnehmen“* gekennzeichnet.

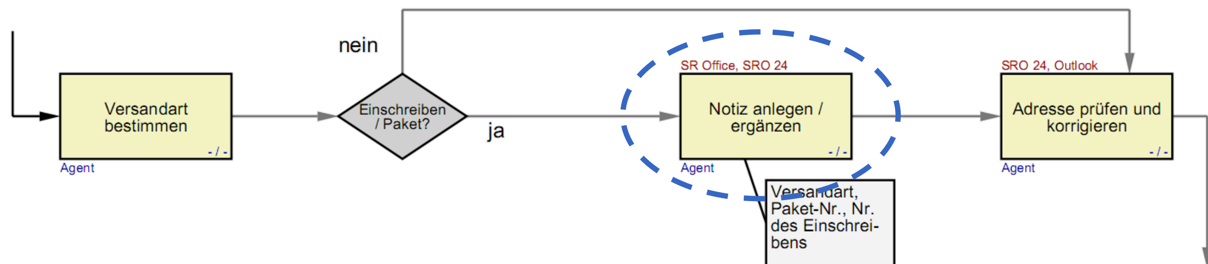

Abb. 2-7: Verwendung dreier Prozesse zur Definition eines größeren Prozesses [6]

Beim Typ-Verwendungs-Konzept wird folglich ein einmal definiertes Objekt (im obigen Fall ein Prozess) in verschiedenen Kontexten verwendet. Diese Weiterverwendung muss jedoch dynamisch geschehen, d.h. Attributwerte dürfen sich gegenüber dem zugrunde liegenden Typ ändern. Ein derartiges Verhalten lässt sich leicht mit Hilfe der vom LMM unterstützten Instanz-Spezialisierung (Abschnitt 2.2.2.3) modellieren. Dazu muss lediglich eine *concreteUseOf*-Beziehung ausgehend von der Verwendung hin zum Typ eingefügt werden.

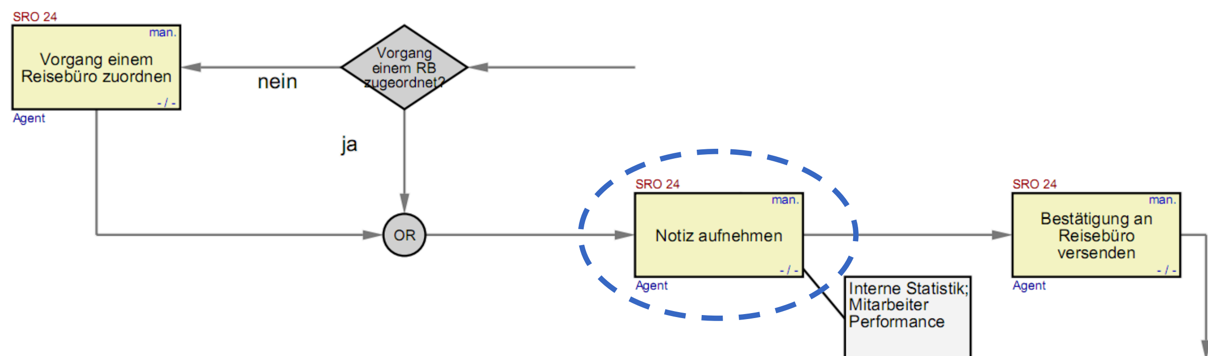

Abb. 2-8: Angepasste Verwendung eines Prozesses [6]

Kapitel 3 Related Work

In diesem Kapitel werden zunächst zwei populäre Systeme zur grafischen Modellierung im Hinblick auf ihre Flexibilität untersucht. Danach erfolgt eine Vorstellung und Evaluierung von Frameworks, die zur Erstellung grafischer Modellierungswerkzeuge dienen. Abschließend werden die verschiedenen Möglichkeiten zur Implementierung eines grafischen DSL-Editors analysiert, die die Eclipse IDE [25] zur Verfügung stellt.

3.1 BPM-Systeme mit Werkzeugen zur Prozessmodellierung

Im Umfeld von Business Process Management Systemen existieren einige Werkzeuge zur grafischen Modellierung von Geschäftsprozessen. Aufgrund der statisch vorgegebenen Modellierungssprachen sind jedoch domänenspezifische Erweiterungen des Sprachumfangs sowie der Support weiterer Sprachen nicht möglich oder mit extrem hohem Aufwand verbunden.

Als bekannter, frei für die nicht kommerzielle Nutzung verfügbarer Vertreter gilt eBPMN von Soyatec [26]. Wie der Name schon vermuten lässt, wird ausschließlich der von der OMG gepflegte Sprachstandard Business Process Modeling Notation (kurz: BPMN) [27] unterstützt. Domänenspezifische Erweiterungen sind zwar möglich, die Verwendung ist aber Tool-spezifisch. D.h. die Unterstützung einer bestimmten Erweiterung muss programmatisch in das jeweilige Modellierungswerkzeug integriert werden. Das dynamische Nachladen von Zusatzfunktionalität ist nicht vorgesehen.

Ein im SAP-Umfeld [28] kommerziell genutztes BPM-System ist die ARIS Plattform [29]. Sie bietet zwar ein umfangreicheres Repertoire an Modellierungssprachen [30] (wie z.B. BPMN [27], BPEL [31], UML [19]), ist aber darauf beschränkt und somit für gänzlich neuartige Anforderungen nicht oder nur mit starken Kompromissen einsetzbar. Bei der Benutzung von statischen kommerziellen Lösungen resultiert zudem eine deutliche Abhängigkeit vom Hersteller, da der Umstieg auf ein alternatives BPM-System im Regelfall mit hohen Kosten in Verbindung steht. Das hängt damit zusammen, dass die einzelnen Systeme unterschiedliche Schnittstellen und Formate zum Datenaustausch besitzen, zwischen denen vermittelt werden muss. Zu dem Zweck existieren zwar standardisierte Verfahren (z.B. eine Transformation mit Umweg über BPDM [32]), doch sind diese oft mit Verlusten behaftet.

3.2 Frameworks zur Erstellung grafischer Modellierungswerkzeuge

Die größtmögliche Flexibilität und Zukunftssicherheit kann nur mit Systemen erlangt werden, die es dem Benutzer erlauben vorhandene Modellierungssprachen an die eigenen Bedürfnisse anzupassen und darüber hinaus die Möglichkeit bieten, neue Sprachen zu definieren.

Analog zu Text-Grammatiken im Zusammenhang mit Programmiersprachen gibt es auch sog. Graph-Grammatiken, die zur Spezifikation von visuellen Sprachen dienen [33]. Grammatikdefinitionen basieren auf mathematischen Formalismen, die für Nicht-Informatiker – und das sind viele Domänenspezialisten – wenig intuitiv erscheinen. Aus diesem Grund ist die Akzeptanz eines Frameworks, das auf Graph-Grammatiken zur Definition grafischer Modellierungswerkzeuge setzt, bei Domänenexperten problematisch.

Ein anderer Ansatz zur Spezifikation domänenspezifischer Sprachen bzw. kompletter Modellierungsumgebungen beruht auf dem Konzept der Metamodellierung. Im Vergleich zu Grammatiken sind Modelle für Domänenspezialisten im Allgemeinen leichter verständlich. Außerdem entstand die

vorliegende Arbeit im Rahmen von oMME (Abschnitt 2.2), einer IDE zur Metamodellierung, bei der als Ausgangsvoraussetzung gilt, dass alle Informationen in Form von Modellen dargestellt werden sollen, die eine uniforme Struktur zur Repräsentation nutzen. Das wiederum gewährleistet einen einheitlichen Zugriff auf sämtliche zur Verfügung gestellten und erzeugten Daten.

In den folgenden Abschnitten werden einige Frameworks zur Erstellung grafischer Modellierungswerkzeuge auf der Basis von Metamodellen vorgestellt, evaluiert und einem abschließenden Vergleich unterzogen.

3.2.1 ConceptBase

ConceptBase ist ein an der Tilburg University (Niederlande) und der RWTH Aachen entwickelter deduktiver Objektmanger, der darüber hinaus die Fähigkeit besitzt Meta-Modellierung zu betreiben [34]. Das zugrunde liegende Meta-Modell basiert zwar nicht auf MOF, bietet aber dennoch die Möglichkeit mehr als zwei Meta-Ebenen bei der Modellierung zu verwenden. Diese Meta-Ebenen werden bei ConceptBase als *Abstraction Levels* bezeichnet. Zur Abspeicherung wird – wie auch bei LMM (Abschnitt 2.2.3) – eine einheitliche Datenstruktur verwendet, die sog. P-facts. ConceptBase implementiert O-Telos (Object-Telos), eine objektorientierte Version der Sprache Telos, die zur Repräsentation von Wissen dient [35]. O-Telos ist über Axiome [36] definiert und kann sowohl mit Hilfe textueller als auch grafischer Notation ausgedrückt werden. Allerdings verbieten gerade diese Axiome einige neuartige Modellierungsparadigmen. Beispielhaft werden die beiden folgenden Axiome betrachtet:

- **Axiom 9:** „Ein Objekt darf eine Attributdefinition in einer seiner Klassen nicht vernachlässigen.“ – Das steht im Widerspruch zu *Deep Instantiation* [6], weil unter Einsatz dieses Modellierungsmusters Attribute erst dann gesetzt werden müssen, wenn der entsprechende *Deep Instantiation Counter* den Wert 0 erreicht hat.
- **Axiom 23:** „Ein Objekt gehört genau zu einer der folgenden Kategorien: Individuum, Instanz, Spezialisierung, Attribut.“ – Das ist ein Widerspruch zur Clabject-Definition (Abschnitt 2.2.2.1), weil demnach ein Objekt sowohl eine Instanz (Instanz-Facette) als auch eine Spezialisierung (Typ-Facette) darstellen kann.

Ein weiterer negativer Aspekt ist die sehr eingeschränkte Anpassbarkeit des grafischen Editors, der ausschließlich die Graph-basierte Modellierungsweise [22] unterstützt. Mit dessen Unterstützung geht das Entwickeln von Meta-Modellen zwar leicht von der Hand, flexible domänenspezifische Editoren können aber nicht erstellt werden. Es besteht lediglich die Möglichkeit vordefinierte Formen und Verbindungen auszutauschen, die wiederum eine bestimmte Struktur aufweisen müssen [34]. So verfügt beispielsweise jede Form über genau ein Label, das das zugehörige Modellelement näher beschreibt.

3.2.2 MOFLON

MOFLON ist ein auf Fujaba basierendes Werkzeug zur grafischen Meta-Modellierung unter Einsatz von MOF [12]. Es wird an der TU Darmstadt entwickelt und kann zudem – wie bereits Fujaba – zur Generierung von Java-Code verwendet werden [37]. Prinzipiell sind mit MOFLON modellierte Meta-Modelle zwar zur Erstellung und Integration domänenspezifischer Werkzeuge geeignet, doch ist dies mit einem extrem hohen Aufwand verbunden. Der Grund dafür ist, dass MOFLON ausschließlich Code für das Modell generiert, der Editor aber manuell implementiert werden muss (z.B. unter Verwendung von GEF [38]).

Durch die strikte Festlegung auf MOF ohne die in Abschnitt 2.2.2 beschriebenen Erweiterungen folgt direkt, dass keine Unterstützung des Typ-Verwendungs-Konzepts gewährleistet ist. MOF erlaubt es hierfür nicht die Instanz-Facette zu spezialisieren. Ferner kann nur eine Meta-Ebene, nämlich `M2`, zur Definition eines Domänen-Metamodells für einen grafischen Editor verwendet werden, da `M3` selbstbeschreibend als oberste Ebene durch MOF vorgegeben ist.

3.2.3 Generic Modeling Environment

Das an der Vanderbilt University in Tennessee, USA entwickelte Generic Modeling Environment (kurz: GME) [39] ist ein konfigurierbarer Werkzeugsatz zur Erstellung domänenspezifischer Modellierungsumgebungen. Die Meta-Meta-Ebene (siehe Abb. 3-1) ist dabei fest vorgegeben und zugleich selbstbeschreibend, d.h. mit GME kann ein Modellierer gebaut werden, der die Fähigkeit besitzt sein zugrunde liegendes Meta-Modell zu definieren.

Das Herzstück dieser Meta-Meta-Ebene stellt die Klasse *FCO* (first class object) dar, von der alle visualisierbaren Modellelemente abgeleitet sind. Jedem *FCO* können bestimmte *Attributes* und *Constraints* zugewiesen werden, die es näher beschreiben bzw. Einschränkungen für die Modellierung hinsichtlich dieses Objekts festlegen. *Model*-Objekte können prinzipiell komposit sein, da sie weitere *FCOs* enthalten dürfen. Damit lassen sich hierarchische Strukturen modellieren, wodurch Potential für einen gewissen Grad an Wiederverwendung gegeben ist. Eine komposite visuelle Darstellung ist allerdings nicht möglich. Zur Anzeige des Inhalts eines *Models* ist dieses stets in Form eines separaten Diagramms zu öffnen. Des Weiteren kann für *Models* eine Vererbungshierarchie spezifiziert werden, analog zu Klassen in objektorientierten Programmiersprachen. Ein *Atom* hingegen repräsentiert ein elementares Objekt, das im späteren Domänenmodell durch ein Symbol dargestellt wird und nicht weiter verfeinerbar ist. Verschiedenen Arten von Beziehungen zwischen *FCOs* lassen sich mit Hilfe von *Connections*, *References* und *Sets* realisieren.

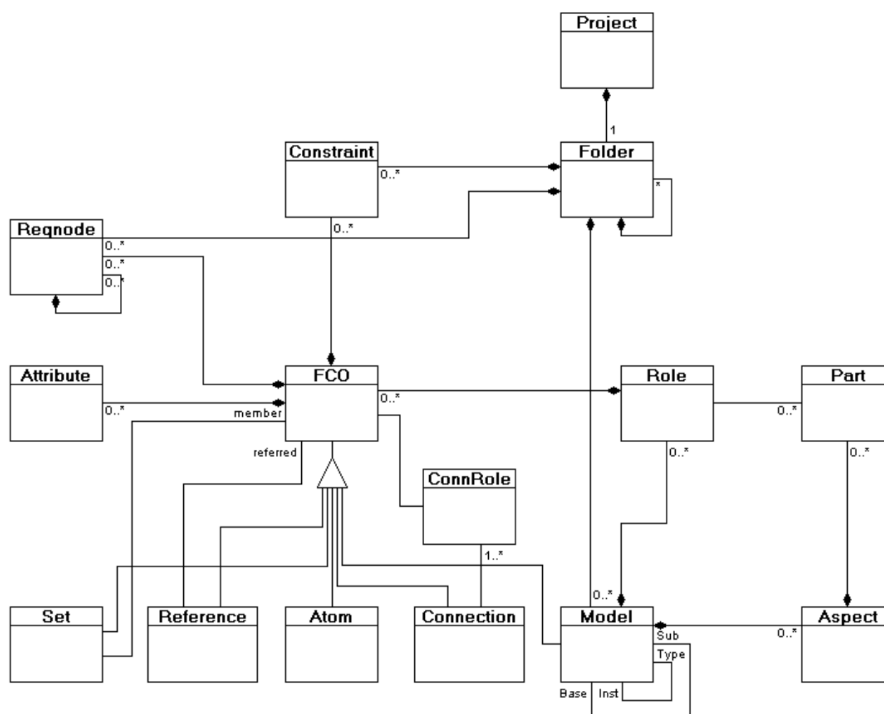

Abb. 3-1: UML-Klassendiagramm der Meta-Meta-Ebene eines GME-Modellierers [40]

Interessant im Hinblick auf die Visualisierung ist insbesondere das Konzept der *Aspects*. Vereinfacht ausgedrückt legt ein *Aspect* fest, welche *FCOs* im Diagramm dargestellt werden, d.h. er bietet eine wohldefinierte Sicht auf die *FCOs* des momentan geöffneten *Models*. Einem *Project* dürfen beliebig viele *Aspects* hinzugefügt werden, aus denen entsprechend viele Visualisierungen für die verschiedenen *Models* eines *Projects* resultieren.

Negativ zu bewerten ist jedoch, dass lediglich ein fester Formensatz für die grafische Repräsentation zur Verfügung steht und dieser ausschließlich anhand von Bitmaps und einigen wenigen grafischen Eigenschaften (z.B. Textfarbe, Schriftgröße, Hintergrundfarbe) an die Benutzerwünsche angepasst werden kann. Aufgrund des Erweiterungsmechanismus und der Modularität von GME existiert die Möglichkeit ein COM-Modul [41] [42] zu implementieren, das die Standard Visualisierung durch eine benutzerspezifische ersetzt, woraufhin beliebige grafische Objekte und Formen denkbar sind [40]. Ein Nachteil besteht allerdings weiterhin: Die Visualisierung ist fest mit den Modellelementen verbunden, weshalb keine Wiederverwendung in anderen Domänen-Modellen möglich ist.

Ein freies Wechseln der Meta-Ebenen ist zwar prinzipiell möglich, jedoch nur mit erheblichem Aufwand. Beabsichtigt der Benutzer nämlich die Meta-Meta-Ebene selbst zu definieren, so muss er zunächst einen zugehörigen Interpreter implementieren, der davon abgeleitete Meta-Modelle versteht und sie für die Verwendung als Modellierungsbasis in GME aufbereitet.

Nach der Klassifikation von [22] (Abschnitt 2.3) unterstützt GME zwar die Graph-basierte Klasse in vollem Umfang, die Geometrie-basierte Variante zur grafischen Repräsentation allerdings nur rudimentär. Eine einfache Hierarchiebildung, wie es Model-Objekte erlauben, ist nicht ausreichend, da Beziehungen zwischen Elementen aus dem Domänen-Modell häufig durch ihre räumliche Anordnung innerhalb des Diagramms visualisiert werden müssen. Als Beispiel aus der Prozessmodellierung kann hier eine Zeitleiste genannt werden, an der die einzelnen Prozessschritte ausgerichtet sind.

Abschließend gilt es noch zu untersuchen inwieweit Änderungen am Meta-Modell von den davon abgeleiteten und bereits persistierten Modellen übernommen werden. Bei Modifikationen an der Visualisierung erfolgt eine automatische Übernahme. Wird jedoch das Domänen-Meta-Modell adaptiert, so waren bei einigen Praxistests immer Änderungen an der zugrunde liegenden XML-Datei erforderlich. Der eingebaute Update-Mechanismus schaffte es nie die Domänen-Modelle automatisch zu korrigieren oder zumindest fehlerhafte Elemente als solche zu markieren.

3.2.4 AToM^3

Mit AToM^3 (A Tool for Multi-Formalism and Meta-Modelling) existiert ein weiteres Framework zur Erstellung domänenspezifischer CASE-Tools auf der Basis visuell formulierter Meta-Modelle, das an der kanadischen McGill University entwickelt wird [43]. Da es selbst ein CASE-Tool darstellt, das zur Entwicklung weiterer CASE-Tools dient, wird AToM^3 auch als meta-CASE-Tool bezeichnet (siehe Abb. 3-3). Die Spezifikation von Meta-Modellen erfolgt mit Hilfe erweiterter ER-Diagramme. Hierbei können Entitäten und Relationen zusätzlich zu Attributen auch über Actions und Constraints verfügen. Für das ER-Modell existiert selbst wieder ein Meta-Modell in Form eines ER-Modells, das beim Öffnen oder Anlegen eines ER-Modells als zugehöriges Meta-Modell (für den Editor) geladen wird. Damit ist die Möglichkeit gegeben sowohl Modell als auch Meta-Modell im selben Editor zu bearbeiten.

Eigene Meta-Modelle können zudem die Grundlage für weitere Meta-Modelle bilden. Dazu muss jedoch ein entsprechender Generator in Form von Python-Code [44] implementiert werden, d.h.

Kenntnisse in dieser Programmiersprache sind Voraussetzung [45]. Prinzipiell kann somit die Meta-Meta-Ebene frei definiert werden, was allerdings mit hohem Aufwand verbunden und nicht für jedermann umsetzbar ist.

Ein negativer Aspekt, der beim ersten Einsatz von AToM³ sofort auffällt, ist die geringe Benutzerfreundlichkeit durch eine wenig intuitive Bedienung (Abb. 3-2). Beispielsweise ist zum Hinzufügen eines neuen Modellelements eine Kombination aus linker Maustaste, STRG und rechter Maustaste notwendig. Hinzu kommen Anzeigefehler nach dem Öffnen und Bearbeiten von Modellen und außerdem häufig auftretende Exceptions, die das Programm zum Absturz bringen.

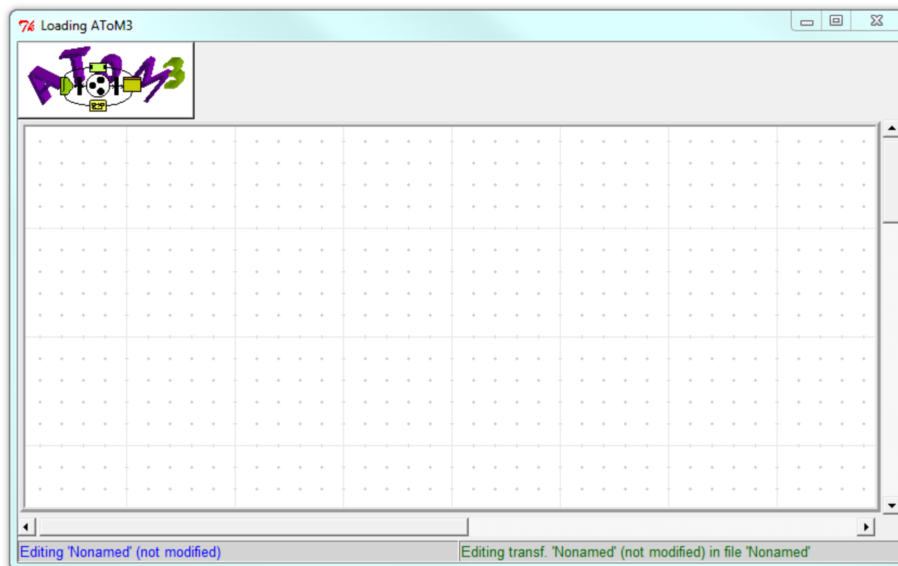

Abb. 3-2: User Interface von AToM³ nach dem Starten mit leerem Modell

Aufgrund der Tatsache, dass logisches und grafisches Meta-Modell fest miteinander verbunden sind, ist ein einfaches Umschalten der Anzeigeform nicht möglich. Für ein logisches Meta-Modell gibt es deshalb nur genau eine Visualisierung, wodurch bei großen Modellen die Übersichtlichkeit verloren geht.

Wie bereits GME (Abschnitt 3.2.3) wird auch von AToM³ die freie Definition der Meta-Meta-Ebene prinzipiell unterstützt. Dazu sind aber ebenfalls Programmierkenntnisse erforderlich, weil ein entsprechender Interpreter für diese Meta-Meta-Ebene implementiert werden muss.

3.2.5 MetaEdit+

MetaEdit+ [46] ist ein kommerzielles, von MetaCase vertriebenes Werkzeug zur Modellierung domänenspezifischer CASE-Tools. Demzufolge handelt es sich bei MetaEdit+ – wie bereits bei AToM³ – um ein sog. Meta-CASE-Tool (Abb. 3-3). Der Benutzer erhält zusätzlich diverse professionelle Werkzeuge, die das Entwickeln neuer bzw. die Erweiterung und Adaption bestehender Meta-Modelle stark erleichtern. Dazu zählen beispielsweise ein visueller Editor zur Definition grafischer Vorlagen für bestimmte Elemente im Domänenmodell [47] sowie ein Editor zum Vergleich von Modellen unterschiedlicher Versionen [48]. Des Weiteren fällt positiv auf, dass Änderungen sowohl am grafischen als auch am logischen Meta-Modell direkt nach dem Speichern übernommen und ggf. invalide Elemente als solche gekennzeichnet werden.

Zu bemängeln ist jedoch, dass logisches und grafisches Meta-Modell fest miteinander verknüpft sind. Damit ist auch mit MetaEdit+ ein Umschalten der Anzeigeform für ein gegebenes Domänenmodell nicht möglich.

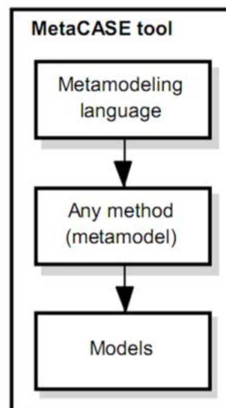

Abb. 3-3: Übersicht über allgemeines Meta-CASE-Tool

Die Meta-Meta-Ebene ist proprietär und fest vom System vorgegeben. Aus diesem Grund ist die Unterstützung der erweiterten Modellierungsmuster (Abschnitt 2.2.2) nicht gegeben. Das Typ-Verwendungs-Konzept kann daher ebenfalls nur in eingeschränkter Form realisiert werden, d.h. eine Wiederverwendung existierender Objekte ist zwar möglich, diese Verwendungen können allerdings nicht individuell manipuliert werden [49].

3.2.6 Graphical Modeling Framework

Das Graphical Modeling Framework (kurz: GMF) [50] steht in Form von Eclipse Plug-Ins für die gleichnamige IDE zur Verfügung. Es dient zur Modellierung und Generierung grafischer Diagramm-editoren auf der Basis von Ecore-Modellen. Als Ecore wird das Meta-Modell von EMF [11] bezeichnet, welches stark von MOF beeinflusst ist. Aufgrund unterschiedlicher Zielsetzungen von OMG und Eclipse bildet MOF nicht die Basis von EMF [13].

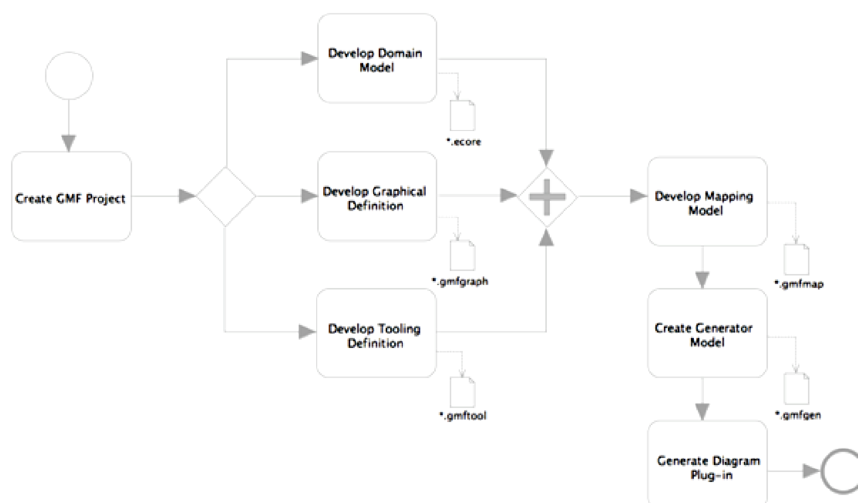

Abb. 3-4: Überblick über den Entwicklungsprozess eines grafischen Editors mit GMF [51]

Vereinfacht ausgedrückt muss bei Benutzung von GMF zunächst der gewünschte Editor anhand von Meta-Modellen spezifiziert werden, aus denen mit Hilfe des mitgelieferten Code-Generators Eclipse Plug-Ins erzeugt werden, die dem modellierten Editor entsprechen [51]. Modifikationen der Meta-Modelle, die einen Editor beschreiben, bedingen deshalb stets einer Neugenerierung des Plug-In

Codes und einem anschließenden Neustart der Eclipse IDE. Einen groben Überblick über den Entwicklungsprozess eines mit GMF erstellten Editors zeigt Abb. 3-4.

Ein Nachteil von GMF ist hingegen, dass bereits geringfügige Änderungen am Meta-Modell häufig zur Unbrauchbarkeit der darauf basierenden Domänen-Modelle führen. In einem solchen Fall kann ein Domänen-Modell nur durch Korrekturen in der zugrunde liegenden und schwer lesbaren XML-Datei gerettet werden, die vom Benutzer manuell durchzuführen sind. Als weiteren negativen Aspekt gilt es die lückenhafte Dokumentation zu erwähnen, aufgrund derer Entwickler – insbesondere in Anbetracht etwas komplexerer Anwendungsfälle – stark in ihrem Arbeitsfluss gehemmt werden; dieser Negativeffekt wird ebenfalls in der Diplomarbeit von [52] festgestellt. Außerdem fehlt sowohl der Support für das Typ-Verwendungs-Konzept als auch für alle anderen neuartigen Modellierungsparadigmen, was zum Teil daran liegt, dass mit Ecore die Meta-Meta-Ebene fest vorgegeben ist.

3.2.7 DSL Tools für Microsoft Visual Studio 2010

Mit den DSL Tools bietet Microsoft [53] für die hauseigene Entwicklungsumgebung Visual Studio 2010 ein kommerzielles Framework zur Erstellung grafischer Editoren auf der Basis domänenspezifischer Sprachen an [54]. In ihren Grundzügen ähneln die DSL Tools stark dem in Abschnitt 3.2.6 vorgestellten Graphical Modeling Framework von Eclipse: Aus einem modellierten Editor wird in der sich an die Modellierungsphase anschließende Generierungsphase Programmcode erzeugt, der in Form von Add-Ons für die Visual Studio IDE Verwendung findet und den gewünschten Editor repräsentiert.

Ein großer Vorteil gegenüber GMF ist die wesentlich umfangreichere Dokumentation und der grafische Editor zur Erstellung eigener Editoren inkl. zugehöriger DSLs. Durch den mit Visual Studio 2010 eingeführten *Model Bus* [55] lässt sich eine eingeschränkte Variante des Typ-Verwendungs-Konzepts nachbilden. Eingeschränkt bedeutet – wie bei vielen bisher untersuchten Werkzeugen – dass zwar eine Wiederverwendung existierender Elemente möglich ist, eine benutzerspezifische Anpassung dieser Elemente jedoch nicht. Der *Model Bus* erlaubt es nämlich, Referenzen auf passende Elemente aus anderen Domänen-Modellen einzubinden [56]. Außerdem stellen die DSL Tools einen neuartigen Erweiterungsmechanismus bereit, der das Konzept der „partiellen Klassen“ von C# [57] einsetzt. Dieser Mechanismus wird als *double-derived pattern* [58] bezeichnet und führt eine strikte Trennung von generiertem und vom Benutzer geschriebenem Code ein. Der Benutzer kann also eigene Erweiterungen implementieren und sie in sein DSL Tools Projekt integrieren, ohne dass die Gefahr besteht diese bei der Neugenerierung des Add-On Codes zu überschreiben.

Negativ hingegen fällt die feste Kopplung von logischem und grafischem Meta-Modell auf, wodurch pro Domänen-Metamodell ausschließlich eine Visualisierung spezifiziert werden kann. Als weiterer Nachteil, der bereits in GMF anzutreffen ist, kann das Neugenerieren des Add-On-Codes nach Änderungen am Meta-Modell genannt werden. Zuletzt kann auch bei den DSL Tools nur eine Meta-Ebene frei definiert werden, was einige erweiterte Modellierungsparadigmen verbietet.

3.2.8 Zusammenfassender Vergleich

Beim Vergleich (siehe Abb. 3-5) der oben vorgestellten Frameworks zur Erstellung domänenspezifischer, grafischer Modellierungswerkzeuge ist die größte Gemeinsamkeit, dass – mit Ausnahme von MOFLON – alle Systeme ihre eigene, oft proprietäre Meta-Meta-Ebene mitbringen und nicht auf den MOF Standard aufsetzen. Ähnlich verhält es sich bei der Umsetzung des Typ-Verwendungs-Konzepts (Abschnitt 2.4), welches lediglich von MetaEdit+ und den Microsoft DSL Tools in rudimentärer Weise unterstützt wird. Rudimentär bedeutet in diesem Zusammenhang, dass zwar eine

Wiederverwendung existierender Elemente gewährleistet ist, eine Spezialisierung derartiger Verwendungen allerdings nicht. Abgesehen von ConceptBase bietet kein Framework die direkte Unterstützung mehrerer frei definierbarer Meta-Ebenen. Mit Hilfe von GME und AToM³ kann der Benutzer aber, sofern er über entsprechende Programmierkenntnisse verfügt, weitere Meta-Meta-Ebenen programmatisch spezifizieren.

	ConceptBase	MOFLON	GME	AToM³	MetaEdit+	GMF	DSL Tools
Anzahl frei definierbarer Meta-Ebenen	beliebig viele	1 (M2)	erweiterbar auf beliebig viele (→ Impl.)	erweiterbar auf beliebig viele (→ Impl.)	1	1	1
MOF-basierend / erweiterte Modellierungsmuster	nein	ja / nein	nein, proprietäre Meta-Meta-Ebene	nein, erweitertes ER-Modell	nein	nein, aber von MOF inspiriert	nein, nur abstrakte Syntax
Unterstützt Typ-Verwendungs-Konzept	nein	nein	nein, lediglich Hierarchiebildung	nein	eingeschränkt, da keine Spezialisierung	nein, lediglich Hierarchiebildung	eingeschränkt ab VS2010 (→ ModelBus)
Aufwand zur Definition einer Visualisierung	nicht möglich	extrem hoch, da Implementierung erforderlich	gering	gering bis mittel	gering	mittel	gering bis mittel
Anpassbare Visualisierung	eingeschränkt, nur Formen austauschbar	ja	eingeschränkt (Bitmaps, fester Formensatz)	ja	ja	ja	ja
Übernahme von Änderungen am MM	keine Angabe, da statische Visualisierung	Implementierung erforderlich	nur eingeschränkt möglich (z.B. grafisch, textuell)	Theorie: Neu laden Praxis: Neustart	direkt	Eclipse neustarten	Visual Studio neustarten
Mehrere Visualisierungen pro Modell möglich	nein	Implementierung erforderlich	Definition separater Aspekte	nein, Modell und Visualisierung sind fest verbunden	nein, Modell und Visualisierung sind fest verbunden	ja, separates Plug-In erstellen	nein, Modell und Visualisierung sind fest verbunden
Mapping: grafisches MM → logisches MM	nicht möglich	Implementierung erforderlich	Vis.-Elemente verweisen auf Modellelemente	Modellelemente beinhalten Visualisierung	Modellelemente beinhalten Visualisierung	neue Mapping-Datei	Vis.-Elemente verweisen auf Modellelemente
Erweiterbarkeit im Rahmen domänen-spezifischer Editoren	beliebig, da Open Source	beliebig, da Open Source	über COM-Interface	beliebig, da Open Source	proprietäre Skriptsprache	Erweiterungspunkte & beliebig, da Open Source	Konzept der double-derived patterns
Integriert in DIE	nein	nein (basiert auf Fujaba)	nein	nein	nein	Eclipse (RCP)	Visual Studio .NET
Anwendungsfeld	universitäres Umfeld [59]	universitäres Umfeld	universitäres Umfeld	nicht für Produktiveinsatz geeignet	professionelle Vermarktung	Open Source, universitär, (nicht) kommerziell	gut für Produktiveinsatz geeignet

Abb. 3-5: Tabellarischer Vergleich von Frameworks zur Erstellung domänenspezifischer, grafischer Modellierungswerkzeuge auf der Basis von Metamodellen

Im Hinblick auf die Visualisierung schneiden ConceptBase und MOFLON am schlechtesten ab, da beide Systeme nicht die Möglichkeit offerieren, domänenspezifische grafische Editoren zu bauen; in beiden Fällen wären tiefergehende Implementierungskenntnisse erforderlich. Die übrigen Frameworks hingegen erlauben die Modellierung derartiger Editoren unter Verwendung verschiedener unterstützender Werkzeuge. Am komfortabelsten ist das grafische Design-Tool von MetaEdit+, das zur Erstellung benutzerspezifischer Formen dient. Diese können wiederum zur Visualisierung von Modellelementen innerhalb eines domänenspezifischen Editors verwendet werden. Ein weiteres positives Feature von MetaEdit+ ist die direkte Übernahme von Änderungen am zugrunde liegenden Meta-Modell. Damit ist es das einzige System, das mit dieser Problematik korrekt umgehen kann, ohne dass die Anwendung neu gestartet werden muss. Der große Vorteil von GME und GMF ist die Möglichkeit pro Meta-Modell mehrere Visualisierungen spezifizieren zu können. GME ist, was die Flexibilität der grafischen Notation anbelangt, allerdings ziemlich eingeschränkt, d.h. es können nur Bitmaps oder vorgegebene Formen verwendet werden um Modellelemente zu visualisieren.

Alle analysierten Systeme können nachträglich um zusätzliche Funktionalität erweitert werden. Allerdings beschränkt sich diese Erweiterbarkeit bei ConceptBase, MOFLON und AToM³ darauf, dass diese Frameworks Open Source sind und somit der zugrunde liegende Code adaptiert oder ergänzt werden muss. Die übrigen Systeme hingegen bieten spezielle Erweiterungspunkte an, die als Schnittstelle zur Integration eigener Modifikationen fungieren.

Die letzte Tabellenzeile mit dem Titel „Anwendungsfeld“ gibt Auskunft über die Verbreitung und mögliche Verwendbarkeit der einzelnen Frameworks. Mit Ausnahme von AToM³ werden die aus universitären Projekten stammenden Systeme auch im universitären Umfeld eingesetzt; zumindest wurden bei der Recherche nach MOFLON und GME keine Hinweise darauf gefunden, dass die beiden Werkzeuge in bestimmten Open Source bzw. Industrie-Projekten zum Einsatz kommen. AToM³ ist nach unseren Erfahrungen aufgrund seiner Instabilität und schlechten Benutzerfreundlichkeit für den Produktivbetrieb untauglich. MetaEdit+ und die DSL Tools werden professionell vermarktet und insbesondere das erstgenannte System ist durchaus für große Industrieprojekte geeignet [60]. GMF findet schließlich im universitären Umfeld, im Open Source Bereich [61] und in einigen (nicht-)kommerziellen Projekten [26] Verwendung.

3.3 Werkzeuge zur Implementierung grafischer Editoren

Da keines der in Abschnitt 3.2 analysierten Frameworks unsere Anforderungen aus Abschnitt 1.1 erfüllt, muss ein System mit entsprechender Funktionalität neu implementiert werden. Als Rahmenbedingung gilt dabei, dass diese Implementierung in Form von Eclipse Plug-Ins realisiert werden soll. Im Eclipse Umfeld kann in diesem Zusammenhang auf verschiedene Abstraktionsebenen aufgesetzt werden, d.h. Eclipse bietet diverse Werkzeuge und Hilfsmittel zur Entwicklung grafischer Editoren. Diese teilweise aufeinander aufbauenden Tools werden im Folgenden beschrieben.

3.3.1 SWT und Swing

Das Standard Widget Toolkit (kurz: SWT) [62] stammt aus dem Eclipse Projekt und bildet die Basis der GUI von Eclipse [63]. Zur Visualisierung bedient es sich der Grafikbibliothek des jeweiligen Betriebssystems, auf dem die SWT-Anwendung ausgeführt wird. Dadurch ist ein Java-Programm optisch im Regelfall nicht mehr von einer Applikation zu unterscheiden, die nativ für ein bestimmtes Betriebssystem implementiert wurde. Allerdings muss für jedes Betriebssystem, auf dem eine SWT-Anwendung ausgeführt werden soll, zunächst eine entsprechende SWT-Anbindung programmiert werden. Für die am weitesten verbreiteten Plattformen, wie Windows, Linux und Mac OS, ist eine derartige Implementierung bereits vorhanden.

Swing [64] hingegen ist die Standard Grafikbibliothek von Java zur Entwicklung grafischer Benutzeroberflächen und Bestandteil der Java Runtime Engine. Diese Bibliothek übernimmt vollständig das Zeichnen der einzelnen GUI-Elemente. Die daraus resultierende Plattformunabhängigkeit führt außerdem dazu, dass das Look-and-Feel einer Swing-Anwendung auf jedem Betriebssystem gleich ist. Unter Eclipse ist es möglich Fensterbereiche (sog. Views) zu erzeugen, die Swing-Elemente beinhalten.

Der Nachteil beider Grafikbibliotheken ist, dass sie keinen erweiterten Support zur Erstellung grafischer Editoren bieten. Sämtliche Editor-Funktionalität (Zeichnen und Selektieren benutzerdefinierter grafischer Elemente, Drag & Drop von Elementen, Routing von Verbindungen etc.) muss manuell implementiert werden, was einen enormen Aufwand bedeutet.

3.3.2 GEF mit Draw2D

Das Draw2D [65] Plug-In setzt auf SWT auf und bietet die Möglichkeit, komplexe, zusammengesetzte grafische Komponenten innerhalb eines SWT-Canvas zu rendern [66]. Es wird im Rahmen des Eclipse Graphical Editing Frameworks [67] entwickelt und bildet für dieses die Visualisierungsbasis [68]. Draw2D verfügt über Algorithmen zum Routing von Verbindungen zwischen Elementen, so dass erstere kein anderes Element außer Start und Ziel schneiden oder gar berühren. Außerdem besitzt es die Fähigkeit Labels, die zu einer Verbindung gehören, automatisch innerhalb des Diagramms zu

platzieren. Das ist keineswegs eine triviale Aufgabe, weshalb in wissenschaftlichen Veröffentlichungen diese Problematik thematisiert und untersucht wird (z.B. in [69]).

GEF trifft die Annahme, dass ein Modell existiert, welches grafisch dargestellt und bearbeitet werden soll. Zur Umsetzung verwendet GEF dabei die weit verbreitete *Model-View-Controller* Architektur (kurz: *MVC*) [70]. Die *View* basiert auf Draw2D, wohingegen das *Model* völlig frei definiert werden kann [71]. Einzige Bedingung ist, dass es in Form von Java-Objekten vorliegen muss. Der rechte Teil von Abb. 4-4 zeigt eine grobe Übersicht über die Anwendung des *MVC*-Patterns innerhalb von GEF. Daraus ist ersichtlich, dass es keine direkte Beziehung zwischen dem *Model* und der *View* gibt. Änderungen am *Model* werden also stets vom *Controller* erfasst und von ihm an die *View*(s) delegiert. Umgekehrt reagiert der Controller auf Ereignisse, die auf dem SWT-Canvas (und damit auf einer View) stattfinden, und wandelt sie in entsprechende *Model*-Änderungen um. Derartige Ereignisse sind beispielsweise einfache Click-Events, die das Anlegen eines neuen Modellelements zur Folge haben. Daraufhin wird der *Controller* erneut benachrichtigt – diesmal vom *Model* – woraufhin er alle registrierten *View*s aktualisiert. Auf diese Weise erfolgt schließlich die Visualisierung des neu erzeugten Modellelements.

GEF besitzt daneben zahlreiche weitere Features zur Erstellung grafischer Editoren. So wird das Verschieben und Vergrößern/Verkleinern von grafischen Formen unterstützt, für welche ein entsprechendes Modellelement existiert. Es besteht die Möglichkeit sowohl die Graph-basierte als auch die Geometrie-basierte Modellierungsweise (Abschnitt 2.3) zu verwenden, d.h. es können Verbindungen zwischen verschiedenen Editor-Elementen gezogen sowie Editor-Elemente ineinander verschachtelt werden. Des Weiteren ist es vorgesehen, sämtliche Modelländerungen in Form von sog. *Commands* zu implementieren. Damit wird dem Entwickler automatisch die Möglichkeit gegeben entsprechende Undo-/Redo-Funktionalität anzubieten. Zur Vereinfachung der Bedienbarkeit des entstehenden Editors kann die Snapping-Eigenschaft aktiviert werden, so dass Elemente an einem vorher definierten Raster oder an bereits vorhandenen Elementen ausgerichtet werden. Die hier beschriebenen Features sind lediglich ein kleiner Auszug der Möglichkeiten von GEF. Dank des generischen Konzepts kann die Funktionalität des Graphical Editing Frameworks beliebig erweitert werden. Als Beispiel zur Demonstration der flexiblen Erweiterbarkeit kann das auf GEF basierende GMF genannt werden, das in den Abschnitten 3.2.6 und 3.3.4 analysiert und vorgestellt wird.

Aufgrund der zahlreichen Features, die bereits im Repertoire von GEF vorhanden sind, sowie der Tatsache, dass ein beliebiges *Model* verwendet werden kann, ist dieses Framework sehr gut für unseren Anwendungsfall geeignet. Allerdings existieren für die Eclipse IDE noch zwei weitere Frameworks auf einem höheren Abstraktionsniveau.

3.3.3 Zest

Zest: The Eclipse Visualization Toolkit [72] basiert zwar auf GEF, ist aber primär zur Visualisierung von Daten und Modellen gedacht und nicht zu deren Erstellung oder Bearbeitung. Außerdem wird zum gegenwärtigen Zeitpunkt lediglich die Graph-basierte Darstellungsweise [22] unterstützt, was für unseren Anwendungsfall, der hohe Flexibilität fordert, nicht ausreichend ist.

3.3.4 GMF

GMF wird bereits ausführlich im Abschnitt 3.2.6 diskutiert. Das Framework an sich sowie der generierte Editor-Code setzen auf GEF auf. Demzufolge wird eine hohe Abstraktionsebene erreicht, was die Erstellung grafischer Editoren betrifft. Da GMF jedoch Ecore als feste Modellierungssprache zur Definition des Meta-Modells vorschreibt und deshalb die erweiterten Modellierungskonzepte von

Volz (Abschnitt 2.2.2) nicht unterstützt werden, ist dieses Framework ebenfalls für unseren Anwendungsfall ungeeignet. Hinzu kommt, dass bei geringfügigen Änderungen am Meta-Modell oder an der Editor-Definition sämtlicher Code neu generiert werden muss, woraufhin ein Neustart der Eclipse IDE erforderlich ist. Wie bereits in Abschnitt 3.2.6 erwähnt, führen derartige Änderungen des Öfteren dazu, dass bereits erstellte Modelle ungültig werden und nur mit erheblichem Aufwand repariert werden können.

3.3.5 Fazit

Zusammenfassend ist zu sagen, dass für unseren Anwendungsfall, dessen Hauptaugenmerk auf Flexibilität und Ausdrucksstärke liegt, das Graphical Editing Framework am besten geeignet ist. Beim Einsatz von SWT oder Swing würde man das sprichwörtliche Rad neu erfinden, da keinerlei Grundfunktionalität für grafische Editoren angeboten wird. Zest hingegen ist nicht für das Erstellen und Bearbeiten von Modellen ausgelegt und GMF verhält sich sehr statisch gegenüber Änderungen an Meta-Modell und Editor-Definition. Darüber hinaus ist das Meta-Meta-Modell bei GMF fest vorgegeben und kann nicht beliebig ersetzt werden.

Kapitel 4 Verwendete Technologien

Basierend auf dem Ergebnis der Evaluierung in Abschnitt 3.3 kommt für die Implementierung, die im Zusammenhang mit der vorliegenden Arbeit erstellt wird, das Graphical Editing Framework zum Einsatz. Dieses wiederum enthält Draw2D zum Zeichnen von Formen, Verbindungen und Text. Beide Technologien und die Eclipse IDE werden deshalb in den folgenden Kapiteln so detailliert vorgestellt, wie es zum Verständnis der restlichen Ausführungen notwendig ist.

4.1 Eclipse

Eclipse [10] basiert seit Version 3.0 auf der freien OSGi-Implementierung [73] Equinox [74]. OSGi wiederum stellt ein standardisiertes Java-Framework dar, das zur Entwicklung modularer Anwendungen dient [25]. Module werden bei Eclipse als Plug-Ins bezeichnet. Demnach ist Eclipse lediglich der Kern, der die einzelnen Plug-Ins lädt, die schließlich die eigentliche Funktionalität bereitstellen. Ursprünglich war dies das Entwickeln von Java-Anwendungen unter Einsatz der Java Development Tools. Inzwischen existiert jedoch eine Vielzahl von Plug-Ins, die dem Software-Entwickler während des gesamten Entwicklungsprozesses unterstützend zur Verfügung stehen. Für unsere Zwecke ist vor allem das Eclipse Modeling Project [11] von Bedeutung, dessen Aufgabe darin besteht Model-Driven Software Development zu unterstützen. Zusätzlich stellen die Plug-Ins Draw2D und GEF ein Grundgerüst zur Implementierung grafischer Editoren bereit.

Wegen des ausgereiften Plug-In Mechanismus können mit Eclipse sog. Rich Client Anwendungen entwickelt werden, die ebenfalls die Eclipse IDE als Grundlage verwenden können. Deshalb wird sie auch als *Rich Client Platform* bezeichnet. Ziel des oMME-Projekts [6] ist es eine derartige Anwendung auf Basis der Eclipse Plattform zu realisieren.

4.2 Draw2D

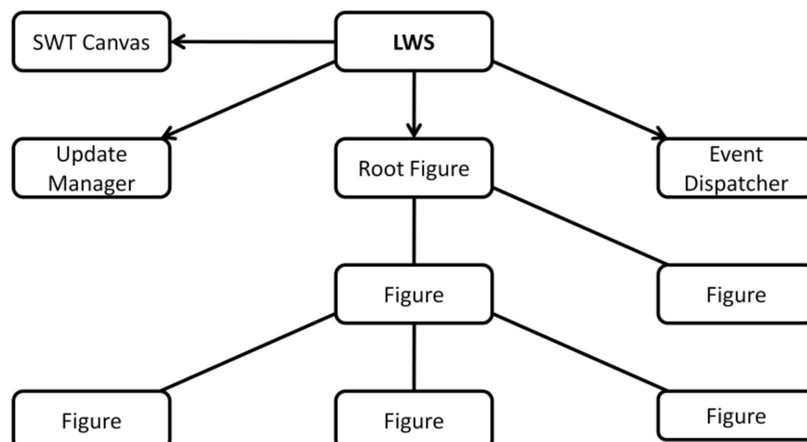

Abb. 4-1: Draw2D-Komponentenhierarchie (nach [75])

Draw2D [65] stellt einen Werkzeugsatz bereit, mit dessen Hilfe grafische Elemente komfortabel gezeichnet und gelayoutet werden können [75]. Diese grafischen Elemente, die sog. *Figures*, bilden das Herzstück von Draw2D. Sie können weitere *Figures* beinhalten, die innerhalb der *Bounds* (dt. Begrenzung) der Eltern-*Figure* gezeichnet und von einem *Layout Manager* ausgerichtet werden. Ein *LightweightSystem* (kurz: *LWS*) stellt die Wurzel der Draw2D-Komponentenhierarchie dar und legt einen *SWT Canvas* fest, der als Zeichenfläche dient. Außerdem existiert genau eine *Root Figure*, die den Inhalt der Zeichenfläche beschreibt. Das LWS schickt die meisten *SWT-Events* (z.B. *Mouse Click*

Events, Key Press Events) an den *Event Dispatcher* weiter, der sie in Ereignisse für die entsprechende *Figure* übersetzt. *Paint Events* hingegen werden an den *Update Manager* delegiert, der sich für die Koordination von Zeichnen und Layouten verantwortlich zeigt. Einen groben beispielhaften Überblick über die erwähnten Komponenten inkl. deren Zusammenhänge veranschaulicht Abb. 4-1.

Beim Zeichnen zusammengesetzter *Figures* kommt das als Clipping bezeichnete Verfahren zum Einsatz, nach dem Kind-*Figures* nur innerhalb der *Bounds* ihrer jeweiligen Eltern-*Figure* visualisiert werden. Dieser Prozess setzt sich für die gesamte *Figures*-Hierarchie kumulativ fort. Abb. 4-2 zeigt eine solche Hierarchie von drei ineinander verschachtelten *Figures* sowie das entsprechende Ergebnis nach dem Clipping.

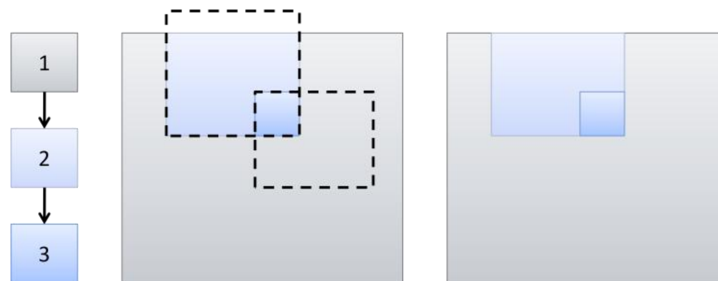

Abb. 4-2: Clipping einer Figure-Hierarchie [75]

Eine gesonderte Stellung innerhalb von Draw2D besitzen *Connections*. Die Standardimplementierung *PolylineConnection* verbindet zwei beliebige *Figures* in Form einer oder mehrerer aneinandergereihter Linien. Der Hauptunterschied zu regulären *Figures* ergibt sich aus dem für jede *Connection* zwingend erforderlichen *ConnectionRouter*, der für das Setzen der *Bounds* verantwortlich ist. Diese werden anhand der Punkte, die den Linienverlauf beschreiben, berechnet (*Bendpoints*). Start- und Endpunkt einer *Connection* werden stets an jeweils einem *ConnectionAnchor* befestigt, der zugleich die exakte Position des entsprechenden Punktes spezifiziert und genau einer *Figure* zugeordnet ist. Des Weiteren verfügt jede *Connection* standardmäßig über ein *DelegatingLayout*, das die Platzierung sämtlicher Kinder an den zugehörigen *Locator* weiterleitet. Gleichzeitig werden die *Bounds* automatisch angepasst, so dass alle Kind-*Figures* im sichtbaren Clipping-Bereich liegen. In Abb. 4-3 ist ein einfaches Beispiel einer *PolylineConnection* mit drei Kind-Elementen (zwei *Labels*, eine *PolylineDecoration*) inkl. der zugehörigen *Figures*-Hierarchie dargestellt.

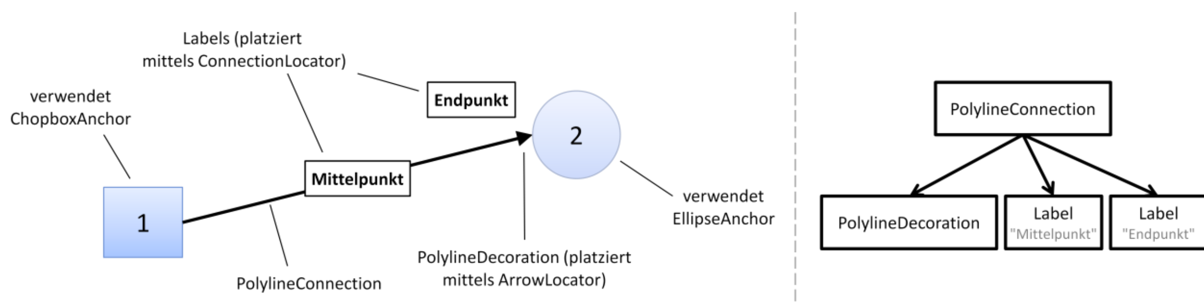

Abb. 4-3: PolylineConnection mit drei Child-Figures (nach [75]) und zugehöriger Figures-Hierarchie

4.3 GEF

Das Graphical Editing Framework [67] beruht – wie bereits in Abschnitt 3.3.2 erläutert – auf der *Model-View-Controller* Architektur. Als *Model* können hierbei beliebige Java-Objekte verwendet werden und als *View* kommt das oben vorgestellte Draw2D zum Einsatz. GEF kann dabei grob als der durch eine Ellipse abgetrennte, mittlere Bereich von Abb. 4-4 definiert werden.

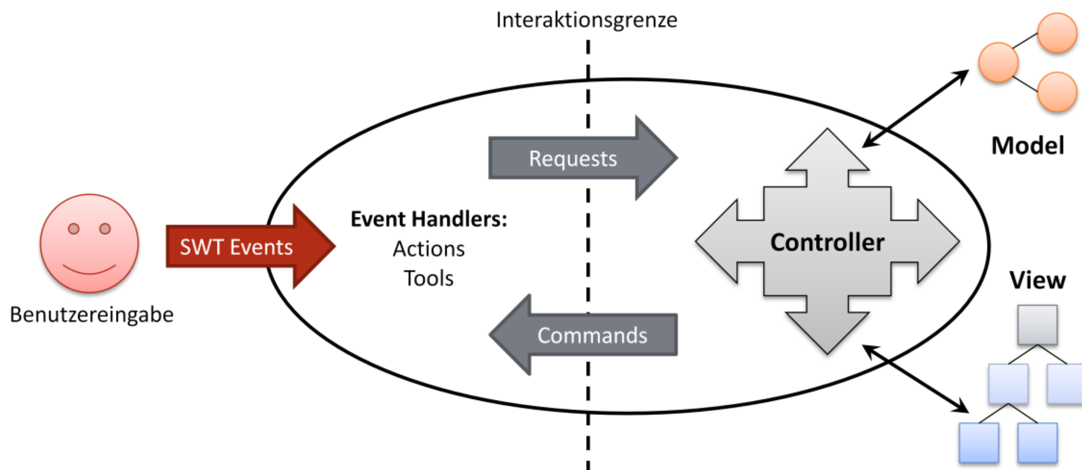

Abb. 4-4: MVC-Architektur von GEF (nach [75])

Den Kern stellt die *Controller*-Komponente dar, die als Vermittler zwischen dem *Model* und der *View* agiert und durch eine Hierarchie von *EditParts* repräsentiert wird. Jedes Modellelement, das visualisiert werden soll, muss eindeutig mit einem *EditPart* assoziiert sein. Dieser kennt zusätzlich die für die Visualisierung benötigte Draw2D *Figure*. Erfolgt nun die Modifikation eines Modellelements, so muss dieses den korrespondierenden *EditPart* darüber informieren, der anschließend seine ihm zugeordnete *Figure* entsprechend aktualisiert.

Einer Modelländerung geht im Regelfall eine Eingabe durch den Benutzer voraus. Dabei handelt es sich immer um ein *SWT Event*, das von einem *Event Handler* abgefangen und in einen passenden *Request* transformiert wird. *Event Handlers* sind entweder *Tools* oder *Actions*. *Tools* werden in der Palette des grafischen Editors angezeigt und dienen u.a. zum Hinzufügen eines neuen Modellelements, zum Selektieren eines oder mehrerer *EditParts* et cetera. *Actions* hingegen werden durch Einträge im Kontextmenü oder Buttons in der Werkzeugleiste repräsentiert, die beim Anklicken den jeweiligen *Request* absetzen. Er wird direkt an den bzw. die selektierten *EditPart*(s) gesendet und von diesen – vereinfacht ausgedrückt – in ein korrespondierendes *Command* übersetzt. Diese Transformation wird im Normalfall zwar von sog. *EditPolicies* übernommen, doch ist dieser Sachverhalt im Rahmen der vorliegenden Masterarbeit unbedeutend und kann bei Bedarf unter [75] nachgeschlagen werden. Der Vorteil des *Command*-Konzepts ist die Möglichkeit, Undo- und Redo-Funktionalität direkt anbieten zu können. Als Bedingung für die *Commands* gilt daher, dass sie ausschließlich auf dem *Model* arbeiten sollten. Das Anpassen der Visualisierung übernehmen die jeweiligen *EditParts*.

Nachdem *EditParts* die zentralen Elemente innerhalb von GEF darstellen, ist deren Struktur ebenfalls von fundamentaler Bedeutung. Jeder *EditPart* kann prinzipiell weitere *EditParts* enthalten, wobei die daraus resultierende Hierarchie im Wesentlichen mit der Hierarchie der Modellelemente übereinstimmt. Nicht jedes Modellelement muss jedoch einem *EditPart* zugeordnet sein. Ähnlich verhält es sich mit der zugehörigen *Figures*-Hierarchie. Hier kann eine *Figure* ebenfalls zusätzliche *Figures* beinhalten, die nicht von einem *EditPart* referenziert werden. Für ein bestimmtes Modellelement wird genau dann ein entsprechender *EditPart* angelegt, wenn der zugehörige Eltern-*EditPart* dieses Modellelement als eines seiner *ModelChildren* deklariert. Die Zeichenfläche wird bei GEF auch durch einen *EditPart* und somit durch ein Modellelement repräsentiert. Abb. 4-5 veranschaulicht ein mögliches Szenario der drei besagten Hierarchien und deren Beziehungen zueinander.

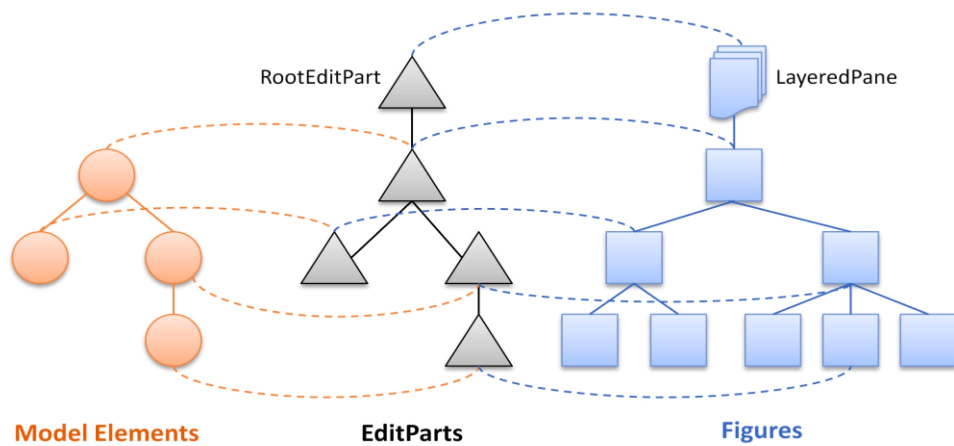

Abb. 4-5: Szenario für mögliche Beziehungen zwischen den einzelnen Elementen von Model, View und Controller bei GEF

Analog zu den Connections bei Draw2D besitzen auch die *ConnectionEditParts* bei GEF eine etwas abgewandelte Verhaltensweise, denn sowohl Ursprungs- als auch Ziel-*EditPart* sind zusammen für das Erzeugen eines *ConnectionEditParts* zuständig. Dazu müssen beide dasselbe Modellelement zurückgeben, der Ursprungs-*EditPart* als Element für eine ausgehende und der Ziel-*EditPart* als Element für eine eingehende Verbindung. Daraufhin wird schließlich der korrespondierende *ConnectionEditPart* angelegt und mit Ursprung und Ziel verbunden.

Kapitel 5 Hierarchie der (Meta-)Modelle

Ein grafischer Editor – auch Designer oder DSL-Editor genannt – soll deklarativ in Form mehrerer Modelle beschrieben werden können (Anforderung (b) in Abschnitt 1.1). Dabei gilt als Bedingung, dass die einzelnen Elemente dieser Modelle nach Möglichkeit direkt auf korrespondierende GEF-Klassen abgebildet werden sollen. Aus dem Grund ist es zweckmäßig, die Modelle entsprechend des in Abschnitt 4.3 beschriebenen MVC-Patterns zu klassifizieren (Abb. 5-1), wodurch sich folgende drei Modellklassen ergeben:

- *Graphics*: Deklariert grafische Elemente, die auf Draw2D-*Figures* abgebildet werden.
- *Model Elements*: *Concepts* der Domäne; können direkt in GEF verwendet werden, da Objekte beliebigen Typs als *Model* verwendet werden dürfen.
- *Editor*: Zuständig für die Vermittlung zwischen grafischen Elementen und Elementen der Domäne; repräsentiert durch die *EditParts* von GEF.

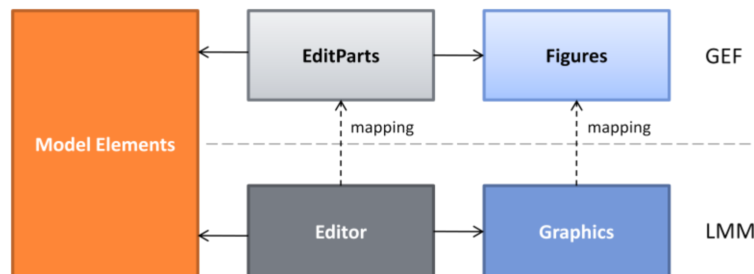

Abb. 5-1: Einführung von Modellklassen analog zum MVC-Pattern von GEF

Trotz des engen Bezugs zu GEF ist die Hierarchie der Modelle, die zur Spezifikation eines Designers benötigt werden, generisch, da die Abbildung auf eine konkrete Implementierung ausschließlich von einem Modell – dem sog. *Programming-Language-Mapping-Model* – abhängt (Abschnitt 5.2). Diese im Folgenden vorgestellte Modell-Hierarchie konkretisiert im Wesentlichen das Zusammenspiel der einzelnen Modelle und Meta-Modelle. Wie Abb. 5-2 zu entnehmen ist, lassen sie sich in eine der beiden Kategorien *Domain-Model Stack* oder *Designer-Model Stack* einordnen.

Der *Domain-Model Stack* beinhaltet alle Modelle, die zur Beschreibung der Anwendungsdomäne relevant sind. Bezogen auf das Beispiel der perspektivenorientierten Prozessmodellierung sind das einerseits diejenigen Modelle, die die einzelnen Entitäten zur Repräsentation der verschiedenen Perspektiven definieren. Andererseits gehören auch solche Modelle dazu, die den Inhalt eines konkreten Prozessmodells, d.h. Instanzen (sowohl Typen als auch Verwendungen) der durch die Perspektiven vorgegebenen Entitäten, beschreiben.

Der *Designer-Model-Stack* umfasst diejenigen Modelle, die eine bestimmte, wohl-definierte Art von grafischen Editoren spezifizieren. Sie gliedern sich erneut in zwei Unterkategorien, *Graphical* und *Editor*. Modelle der *Graphical*-Kategorie definieren verschiedene grafische Formen. Die Elemente der Editor-Kategorie bilden schließlich diese Formen auf Entitäten des *Domain-Model Stacks* ab. Beispielsweise wird auf der grafischen Seite ein blaues Rechteck definiert, das über ein Editor-Modellelement einer Prozess-Entität zugewiesen wird. Damit erscheinen alle Prozesse innerhalb eines mit dem Designer visualisierten Prozessmodells in Form blauer Kästen.

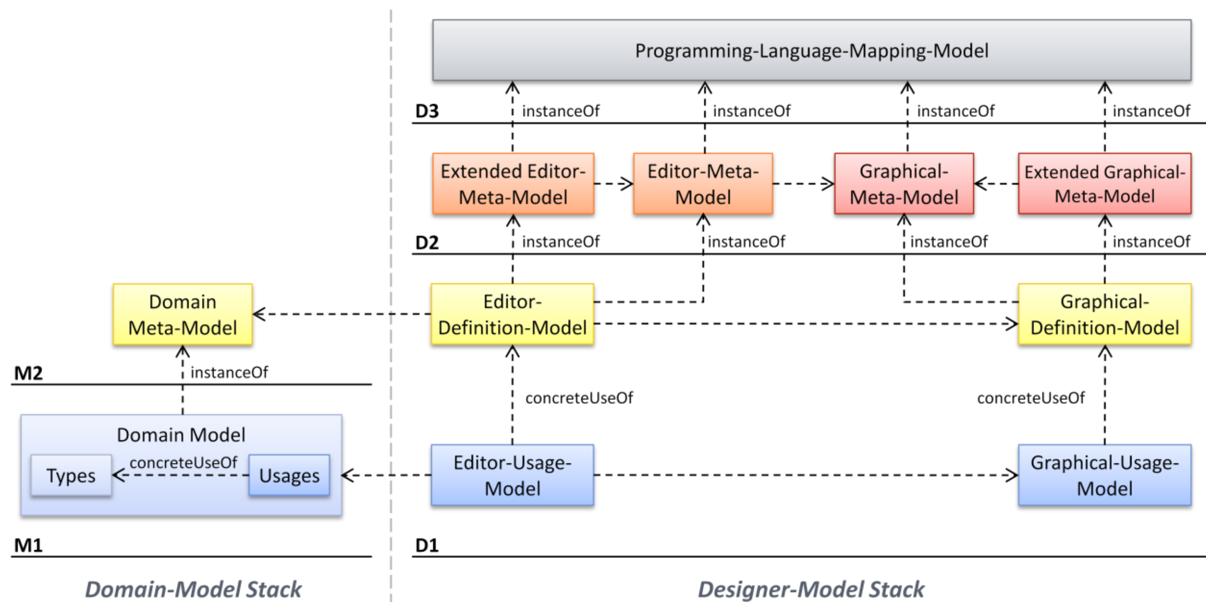

Abb. 5-2: Modell-Hierarchie zur Beschreibung eines Designers (exemplarisch mit Typ-Verwendungs-Konzept auf der Seite des Domänen-Modells)

5.1 Domain-Model Stack

Die wichtigste Anforderung, die ein Framework zur Erstellung eines Designers erfüllen muss, ist, dass das dem Editor zugrunde liegende Domänen-Meta-Modell beliebig definiert werden darf. Darin dürfen also sämtliche Modellierungsmuster Verwendung finden, die durch das LMM (Abschnitt 2.2.3) ausgedrückt werden können. Wegen der vielen Freiheitsgrade, die das LMM einräumt, sind darüber hinaus verschiedene Modellierungsansätze denkbar, mit denen auf Basis eines gegebenen Meta-Modells ein entsprechendes Domänen-Modell erzeugt werden kann. Folgende zwei Ansätze können identifiziert werden:

1. *Instance-Creation*: Zu einem gegebenen *Concept* des Domänen-Meta-Modells werden stets korrespondierende Instanz-*Concepts* im Domänen-Modell angelegt. Das entspricht der klassischen Modellierungsweise, wie sie von allen in Abschnitt 3.2 vorgestellten Systemen unterstützt wird.

2. *Type-Usage-Creation*: Zu einem gegebenen *Concept* des Domänen-Meta-Modells werden zunächst korrespondierende Typ-*Concepts* in Form von Instanzen angelegt. Von diesen Typ-*Concepts* können in einem weiteren Schritt mehrere Verwendungs-*Concepts* erzeugt werden, die jeweils eine *concreteUseOf*-Referenz auf Ersteres halten. Nähere Details dazu inkl. eines Anwendungsbeispiels können in Abschnitt 2.4 nachgelesen werden. Nachdem die Unterstützung dieses Modellierungsansatzes ein essentielles Merkmal des Model Designer Frameworks sein soll, zeigt Abb. 5-2 dessen Modell-Hierarchie mit integriertem Typ-Verwendungs-Konzept auf der Seite des *Domain-Model Stacks*. Im Folgenden wird stets dieser Ansatz verfolgt, da er im Wesentlichen eine Verallgemeinerung der *Instance-Creation* ist.

5.2 Designer-Model Stack

Die Grundlage des *Designer-Model Stacks* (Abb. 5-2) bilden die beiden obersten Ebenen `D2` und `D3`. `D2` enthält sämtliche Meta-Modelle (mit Ausnahme der Domänen-Meta-Modelle) zur Spezifikation eines grafischen Editors. Das *Graphical-Meta-Model* stellt die grundlegenden Meta-Elemente zur Verfügung, die für das Arrangement komplexer grafischer Formen benötigt werden. Im *Editor-Meta-Model* hingegen werden Mapping-*Concepts* definiert, die Elemente des *Graphical-Meta-Models* auf

Elemente des *Domain-Meta-Models* abbilden. Die *Extended*-Versionen von *Graphical*- und *Editor-Meta-Model* dienen zur Definition benutzerdefinierter Erweiterungen.

Ebene `D3` umfasst lediglich das *Programming-Language-Mapping-Model*. Dieses Meta-Meta-Modell schlägt die Brücke zwischen Modellen des LMM und einem korrespondierenden Objekt-Modell einer objektorientierten Programmiersprache. Damit lässt sich ein *Concept* auf eine beliebige Klasse dieser Programmiersprache abbilden. Weil `D2` eine *instanceOf*-Relation zu `D3` besitzt, kann zu jeder Instanz eines *Concepts* des *Graphical*- und *Editor-Meta-Modells* automatisch eine entsprechende Objekt-instanz erzeugt werden.

Auf Ebene `D1` kommt das in Abschnitt 2.4 erläuterte Typ-Verwendungs-Konzept zum Einsatz. *Graphical-Definition*- und *Editor-Definition-Model* enthalten dabei die Typen, wohingegen die entsprechend benannten *Usage-Models* die zugehörigen Verwendungen aufweisen. Die beiden *Definition-Models* werden von derjenigen Person angelegt, die einen Designer spezifiziert. In der grafischen Definition ist das Aussehen aller möglichen Formen festgelegt. Die Editor-Definition hingegen beschreibt das Mapping zwischen diesen Formen und den Elementen des Domänen-Meta-Modells. Auf der Seite der *Usage-Models* enthält die grafische Variante konkrete Verwendungen der im *Graphical-Definition-Model* spezifizierten Formen. Damit kann z.B. die Verwendung eines Rechtecks auf einfache Weise die durch den Typ zugewiesene Hintergrundfarbe überschreiben. Analog dazu besteht das *Editor-Usage-Model* aus konkreten Verwendungen seines zugehörigen *Definition-Models* und bildet konkrete grafische Formen auf Elemente des Domänen-Modells ab. Die beiden *Usage-Models* repräsentieren somit den grafischen Zustand eines durch die korrespondierenden *Definition-Models* beschriebenen Designers. Dieser Zustand wird letztendlich zusammen mit dem Inhalt des visualisierten *Domain-Models* persistiert und kann zu einem späteren Zeitpunkt vom Designer wieder gelesen und angezeigt werden. Mit anderen Worten ausgedrückt stellen die *Usage-Models* die abgespeicherten Visualisierungsdaten zu einem gegebenen Domänen-Modell dar. Der große Vorteil dieser Vorgehensweise ist, dass sowohl den Modellen zur Definition eines Designers als auch den Modellen zur Datenhaltung dieselben beiden Meta-Modelle zugrunde liegen. Demzufolge sind keine separaten Notations-Meta-Modelle zur Speicherung der Visualisierung erforderlich, wie es z.B. bei GMF der Fall ist.

Kapitel 6 Spezifikation der Meta-Modelle

Nachfolgend werden der Aufbau und die einzelnen Elemente der drei durch das Model Designer Framework vorgegebenen Meta-Modelle beschrieben. Hierbei handelt es sich um das *Programming-Language-Mapping-*, das *Graphical-Meta-* und das *Editor-Meta-Model* aus Abb. 5-2. Spätestens an der Stelle muss die endgültige Entscheidung für eine geeignete Programmiersprache getroffen werden: Die Vorgabe, dass die Eclipse IDE zum Einsatz kommen soll (Anforderung (a) in Abschnitt 1.1), impliziert die Verwendung von Java [76]. Daher wird das *Programming-Language-Mapping-Model* im Folgenden als *Java-Bean-Mapping-Model* bezeichnet. Java-Bean meint eine bestimmte Art von Java-Klassen, die die Java-Bean Spezifikation [77] erfüllen. *Graphical-Meta-* und *Editor-Meta-Model* sind prinzipiell weiterhin Plattform-unabhängig, da ausschließlich die Instanz-Facette der einzelnen *Concepts* Implementierungs-technische Details aufweist, nämlich die Abbildung auf konkrete Java-Beans. Dennoch orientiert sich der Inhalt der Modelle an der MVC-Architektur von GEF. Die Gründe für dieses Vorgehen sind pragmatischer Natur: Das Ziel der vorliegenden Masterarbeit ist ein flexibles, funktionierendes und intuitiv bedienbares Framework zur Entwicklung grafischer Editoren auf der Basis des LMMs.

6.1 Java-Bean-Mapping-Model

Das *Java-Bean-Mapping-Model* stellt das Meta-Meta-Modell des *Designer-Model Stacks* dar. Es beinhaltet lediglich ein *Concept* mit dem Namen `JavaBeanMapping`, welches drei String-Attribute besitzt:

- `beanFQN`: Voll qualifizierter Name einer Java-Bean, die instanziiert werden soll; dieses Attribut muss von einem instanziierenden Concept gesetzt werden. Wichtig ist, dass die betreffende Klasse im Class-Path der Anwendung liegt.
- `accessorFQN`: Voll qualifizierter Name einer Klasse, die das *FieldAccessor*-Interface (Abschnitt 8.1.2.1) implementiert; sie fungiert als Wrapper für die zuvor angegebene Java-Bean, so dass ein einheitlicher Zugriff auf deren Felder erfolgt. Sollte eine Klasse nicht 100-prozentig kompatibel zur Java-Bean Spezifikation sein, so kann auf diesem Weg ein individueller *FieldAccessor* eingesetzt werden. Nähere Details zu dieser Funktionalität werden in Abschnitt 8.1.2 erläutert. Wichtig ist, dass die betreffende Klasse im Class-Path der Anwendung liegt. Das Attribut ist optional, d.h. wenn es nicht gesetzt ist, wird eine Standard-Implementierung verwendet.
- `jarBundles`: Beliebige Anzahl von Dateinamen, die Java-Archive (kurz: JAR) referenzieren; dadurch kann der Class-Path für die beiden anderen Attribute kurzzeitig erweitert werden. Durch die Möglichkeit, beliebige JARs einbinden zu können, kann während der Laufzeit der oMME-Plattform neue Funktionalität eingebunden und direkt zur Verfügung gestellt werden. Das Attribut ist optional.

Abb. 6-1 demonstriert exemplarisch einen Anwendungsfall des *Java-Bean-Mapping-Models*. Voraussetzung ist stets, dass eine Java-Klasse, auf die ein *Concept* abgebildet werden soll, bereits existiert. In unserem Fall ist das die Klasse *de.ubt.ai4.omme.mdf.thesis.Dimension* mit den privaten Integer-Feldern *width* und *height*, auf die über entsprechende Getter- und Setter-Methoden zugegriffen werden kann. Dazu muss ein *Concept* (hier: `Dimension`) definiert werden, das zwei Integer-Attribute

mit demselben Namen besitzt und eine Instanz des `JavaBeanMapping`-*Concepts* darstellt. Als `beanFQN` wird der voll qualifizierte Name der Java-Klasse angegeben. Zu einer Instanz des `Dimension`-*Concepts* kann nun automatisch eine Instanz der gemappten Klasse erzeugt werden. Im Beispiel wird für das *Concept* `ExampleDimension` das Java-Objekt, auf das der *Mapping*-Pfeil verweist, angelegt. Gleichzeitig erfolgt die Übergabe aller in `ExampleDimension` zugewiesenen Werte an die korrespondierenden Felder dieses Objekts.

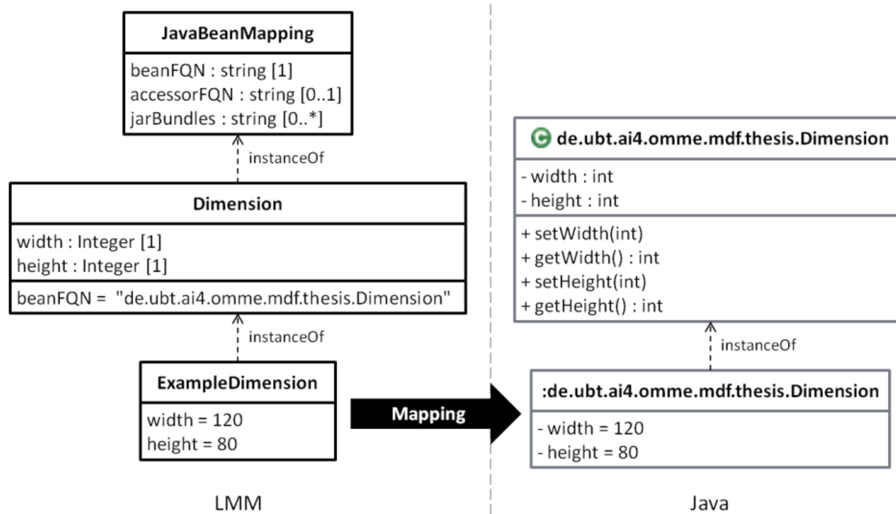

Abb. 6-1: JavaBeanMapping-Anwendungsbeispiel

6.2 Graphical-Meta-Model

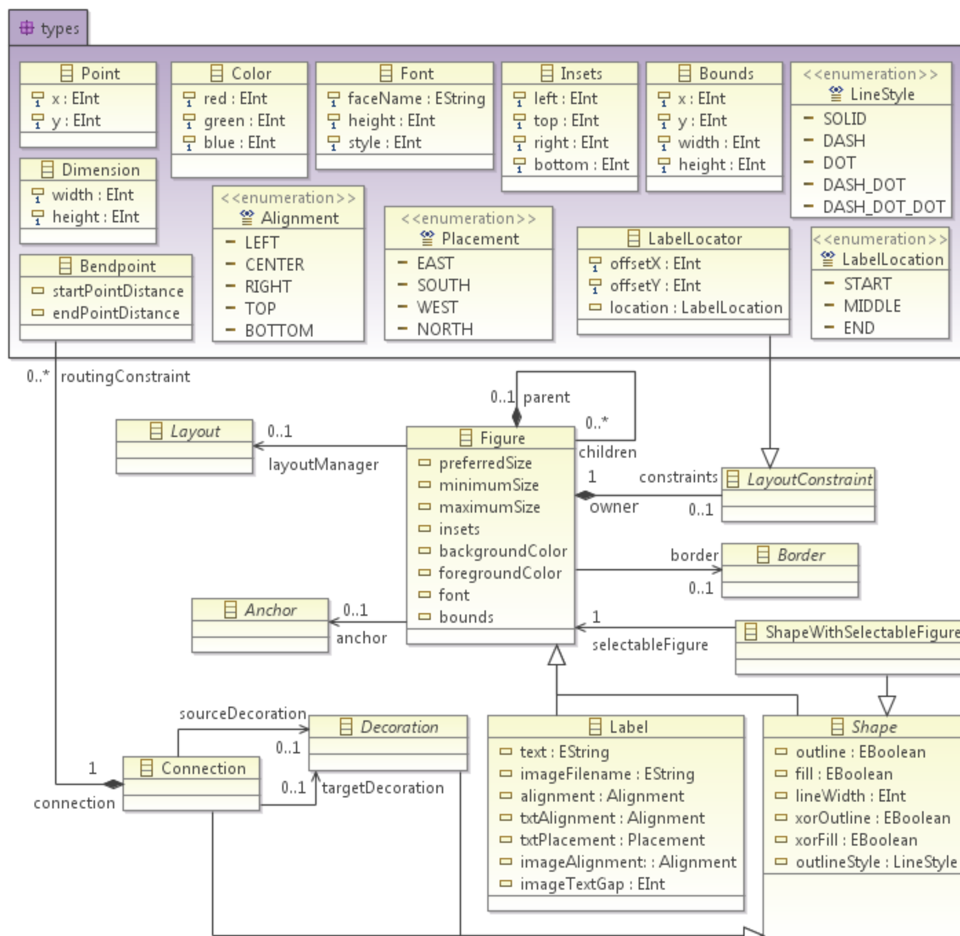

Abb. 6-2: Konzeptdiagramm des Graphical-Meta-Models

Der Inhalt des Graphical-Meta-Models besteht aus denjenigen Meta-Elementen, die zur Definition konkreter grafischer Formen instanziiert werden. Nachdem als Visualisierungs-Modul Draw2D (Abschnitt 4.2) eingesetzt wird, orientiert sich das Meta-Modell stark an der internen Struktur dieser Technologie. Das hat den Vorteil, dass die *Concepts* des *Graphical-Definition-Models* – meist ohne Implementierungsaufwand – direkt auf entsprechende Instanzen von Draw2D-Elementen abgebildet werden können.

Wie Abb. 6-2 zu entnehmen ist, lässt sich das *Graphical-Meta-Model* in zwei Einheiten unterteilen. Die erste Einheit bildet das `types`-*Package*, das eine Reihe komplexer Datentypen (in Form von *Concepts*) und *Enumerations* zur Beschreibung bestimmter Eigenschaften bereitstellt. Erstere kapseln zu diesem Zweck zusammengehörige Attribute (z.B. zur Spezifikation einer Farbe durch drei RGB-Werte bei Verwendung des `Color`-*Concepts*), wohingegen Letztere eine Menge von Auswahlmöglichkeiten bieten. Die andere Einheit beinhaltet als Kernelement das `Figure`-*Concept* sowie seine elementaren Ableitungen `Shape`, `ShapeWithSelecatbleFigure`, `Connection`, `Label` und `Decoration`. Des Weiteren gehören sämtliche Basis-*Concepts* dazu, die die Struktur und/oder das Verhalten von `Figures` beeinflussen.

Auf Basis der damit angebotenen Möglichkeiten und des Konzepts hinter dem Java-Bean-Mapping-Model kann der Benutzer beliebige grafische Formen spezifizieren. Demzufolge wird die Anforderung (h) aus Abschnitt 1.1 vollständig erfüllt.

6.2.1 Inhalt des types-Package

Im Folgenden erfolgt die Spezifikation aller Elemente des `types`-*Package* (Abb. 6-2), beginnend mit den *Enumerations*:

- `Alignment`: *Enumeration*, die zur Ausrichtung grafischer Elemente dient. Dabei handelt es sich im Regelfall um Text oder Bilder. Mögliche Werte sind `LEFT`, `CENTER`, `RIGHT`, `TOP` und `BOTTOM`.
- `Placement`: *Enumeration*, die zur Platzierung eines grafischen Elements relativ zu einem anderen dient. Mögliche Werte sind `EAST`, `SOUTH`, `WEST` und `NORTH`.
- `LabelLocation`: *Enumeration*, die zur groben Positionierung von `Labels` entlang einer `Connection` dient. Mögliche Werte sind `START`, `MIDDLE` und `END`.
- `LineStyle`: *Enumeration*, die den Zeichenstil von Linien beschreibt. Mögliche Werte sind `SOLID` (durchgezogen), `DASH` (gestrichelt), `DOT` (gepunktet), `DASH_DOT` (Strich und Punkt im Wechsel) und `DASH_DOT_DOT` (Strich und zwei Punkte im Wechsel).
- `Point`: *Concept*, das einen Punkt anhand von zwei Integer-Attributen `x` und `y` beschreibt. Beide Attribute sind obligatorisch.
- `Dimension`: *Concept*, das eine 2-dimensionale Ausdehnung anhand von zwei Integer-Attributen `width` und `height` beschreibt. Beide Attribute sind obligatorisch.
- `Bounds`: *Concept*, das einen rechteckigen Bereich innerhalb eines Koordinatensystems definiert. Er wird durch vier obligatorische Integer-Attribute `x`, `y`, `width` und `height` beschrieben.

- `Insets`: *Concept*, das das Einrückverhalten an den Rändern einer `Figure` beschreibt. Die genaue Einrückweite wird durch die vier obligatorischen Integer-Attribute `left`, `top`, `right` und `bottom` festgelegt.

- `Color`: *Concept*, das eine Farbe anhand dreier Integer-Attribute `red`, `green` und `blue` durch additive Farbmischung [78] spezifiziert. Alle drei Attribute sind obligatorisch.

- `Font`: *Concept*, das eine Betriebssystem-Schriftart anhand von drei obligatorischen Attributen beschreibt. Das String-Attribut `faceName` gibt den Name der zu verwendenden Schriftart an, das erste Integer-Attribut `height` legt die Höhe fest und das zweite Integer-Attribut `style` definiert den Stil (normal, fett, kursiv etc.).

- `LabelLocator`: *Concept*, das die genaue Position eines `Labels` entlang einer `Connection` beschreibt. Die grobe Position wird durch das `location`-Attribut vorgegeben, das als Typ die bereits vorgestellte *Enumeration* `LabelLocation` besitzt. Der Standardwert dieses Attributs ist `END`. Die beiden obligatorischen Integer-Attribute `offsetX` und `offsetY` legen die exakte Position relativ zur durch `location` vorgegebenen Position fest. Aus implementierungstechnischen Gründen ist `LabelLocator` eine Spezialisierung von `LayoutConstraint`.

- `Bendpoint`: *Concept*, das zur Angabe von Zwischenpunkten dient, die eine `Connection` passieren muss. Das `connection`-Attribut gibt die zugehörige `Connection` an. Die beiden Attribute `startPointDistance` und `endPointDistance` sind vom Typ `Dimension` und legen den relativen Abstand des `Bendpoints` zu Start- und Endpunkt der `Connection` fest.

6.2.2 Grundlegende Concepts

Das allgemeine *Concept* `Figure` bildet das Zentrum des *Graphical-Meta-Models* (Abb. 6-2) und stellt die Wurzel der Vererbungshierarchie beliebiger konkreter Figuren dar. Eine leere `Figure`, d.h. eine `Figure` ohne Kinder, ist lediglich ein unsichtbares Rechteck. Erst Typ-Spezialisierungen dieses *Concepts* legen ein genaues Aussehen fest (z.B. Rechteck, Ellipse, Linienzug). Als Vertreter derartiger Spezialisierungen gibt es `Connection`, `Label`, `Decoration`, `ShapeWithSelectedFigure` und `Shape`. Letzteres erweitert nur die Konfigurationsmöglichkeiten einer `Figure` durch Hinzufügen zusätzlicher Attribute, besitzt aber ebenfalls keine konkrete Visualisierung und ist daher als *abstrakt* gekennzeichnet. Die Attribute einer `Figure` lassen sich – abhängig von deren Typ – in zwei Kategorien einteilen: Die erste Kategorie umfasst diejenigen Attribute, die Elemente des `types`-*Package* referenzieren, da ihre Semantik bereits vollständig spezifiziert ist. In der zweiten Kategorie befinden sich die Attribute, deren referenzierte *Concepts* als *abstrakt* deklariert sind, wodurch die exakte Semantik in einem spezialisierenden *Concept* festgelegt werden muss. Die zuletzt genannten *Concepts* werden nachfolgend vorgestellt:

- `Layout`: Ein `Layout` positioniert und skaliert die Kinder einer `Figure` innerhalb deren `bounds` anhand bestimmter Regeln. Dazu kann es die jeweiligen `LayoutConstraints`, die den Kindern zugewiesen sind, auswerten.

- `LayoutConstraint`: `Layouts` können beim Layouten eines Kindes das zu diesem Kind gespeicherte `LayoutConstraint` auswerten und es in die Berechnung mit einbeziehen. Zu beachten ist, dass die `LayoutConstraints` der Kind-`Figures` zum `Layout` der Eltern-`Figure` kompatibel sein müssen. Über das `owner`-Attribut kennt jedes `LayoutConstraint` die `Figure`, der es zugewiesen ist.

- `Border`: Verzierung auf einer `Figure`, die innerhalb deren `Bounds` liegt und ihre `Insets` beeinflussen kann.
- `Anchor`: Ein `Anchor` dient als Befestigungsobjekt für `Connections`. Er definiert den genauen Start- bzw. Endpunkt einer Verbindung.

Im weiteren Verlauf werden das `Figure`-*Concept* und alle davon abgeleiteten *Concepts* erläutert. Sofern nicht anders angegeben sind sämtliche Attribute optional.

- `Figure`: Eine `Figure` beschreibt eine grafische Einheit, die zwar eine durch `bounds` bestimmte Fläche einnimmt, doch keine Visualisierung besitzt, d.h. sie ist unsichtbar. Das Aussehen wird erst von spezialisierenden *Concepts* festgelegt. Allerdings bietet das `Figure`-*Concept* einige grundlegende Attribute, die eine grafische Parametrisierung ermöglichen. Diese Parameter müssen von Spezialisierungen nicht zwingend unterstützt werden.
    - `children`: Jede `Figure` kann weitere `Figures` (sog. Kinder) enthalten, die innerhalb der `bounds` der Eltern-`Figure` gezeichnet werden. Dieses Attribut ist gegensätzlich zum `parent`-Attribut.
    - `parent`: Mit Ausnahme der Wurzel-`Figure` besitzt jede `Figure` eine Eltern-`Figure`. Dieses Attribut ist gegensätzlich zum `children`-Attribut.
    - `layoutManager`: Jede `Figure`, die Kind-Elemente aufnimmt, benötigt ein `Layout`. Dieses legt Position und Größe der Kinder fest.
    - `constraints`: Jeder `Figure`, die ein Kind einer anderen `Figure` darstellt, kann ein `LayoutConstraint` zugewiesen werden. Dieses Constraint beeinflusst Position und Größe der Kind-`Figure` beim Layouten durch das `Layout` der Eltern-`Figure`. Das `LayoutConstraint` der Kind-`Figure` muss somit kompatibel zum `Layout` der Eltern-`Figure` sein.
    - `border`: Verziert die `Figure` mit einem Rahmen (`Border`).
    - `anchor`: Legt die Befestigungsform (`Anchor`) der `Figure` für eingehende und ausgehende `Connections` fest.
    - `preferredSize`: Spezifiziert die voreingestellte Größe (`Dimension`) der `Figure`. Sie wird in der Regel dann verwendet, wenn keine `bounds` angegeben sind. Dieses Attribut ist `Layout`-spezifisch, d.h. das der `Figure` zugewiesene `Layout` entscheidet über die Interpretation von `preferredSize`.
    - `minimumSize`: Spezifiziert die minimale Größe (`Dimension`) der `Figure`. Dieses Attribut ist ebenfalls `Layout`-spezifisch.
    - `maximumSize`: Spezifiziert die maximale Größe (`Dimension`) der `Figure`. Dieses Attribut ist ebenfalls `Layout`-spezifisch.
    - `insets`: Legt das Einrückverhalten (`Insets`) an den Rändern der `Figure` fest.
    - `backgroundColor`: Legt die Hintergrundfarbe (`Color`) der `Figure` fest.
    - `foregroundColor`: Legt die Vordergrundfarbe (`Color`) der `Figure` fest.
    - `font`: Spezifiziert die Schriftart (`Font`) der `Figure`. Dieses Attribut ist nur für `Figures` relevant, die zur Anzeige von Text dienen.

- `bounds`: Legt die Begrenzung (`Bounds`), innerhalb der die `Figure` und ihre Kinder gezeichnet werden, fest. Dieses Attribut sollte nur vom Framework selbst gesetzt werden.

Die Attribute `backgroundColor`, `foregroundColor`, `font`, `border`, `constraint`, `bounds` und `children` sind als komposit definiert. Beim Löschen der `Figure`, bei der diese Attribute gesetzt sind, müssen die entsprechenden referenzierten *Concepts* ebenfalls entfernt werden.

- `Shape`: Das `Shape`-*Concept* ist eine abstrakte Spezialisierung der `Figure`. Es besitzt lediglich sechs weitere Eigenschaften, hat keine konkrete Visualisierung und dient als Grundlage zur Spezifikation von Formen, die sowohl eine Kontur als auch eine Füllung aufweisen.
  - `outline`: Das boolesche Attribut legt fest, ob das `Shape` eine Kontur besitzt.
  - `fill`: Das boolesche Attribut legt fest, ob das `Shape` eine Füllung besitzt.
  - `lineWidth`: Spezifiziert die Dicke der Kontur anhand eines Integer-Wertes.
  - `xorOutline`: Das boolesche Attribut legt fest, ob beim Zeichnen der Kontur der XOR-Operator angewendet wird.
  - `xorFill`: Das boolesche Attribut legt fest, ob beim Zeichnen der Füllung der XOR-Operator angewendet wird.
  - `outlineStyle`: Spezifiziert den `LineStyle` der beim Zeichnen der Kontur zum Einsatz kommt.
- `ShapeWithSelectableFigure`: Dieses spezialisierte `Shape` bietet die Möglichkeit, den Selektions-Rahmen auf eine enthaltene Kind-`Figure` zu beschränken. Sie wird über das obligatorische Attribut `selectableFigure` festgelegt. Ein Selektions-Rahmen wird angezeigt, sobald ein Element im Designer ausgewählt wird. Bei Verwendung dieses *Concepts* kann der Anschein erweckt werden, dass `Figures` am Rand einer anderen `Figure` – nämlich der `selectableFigure` – anhaften.

Abb. 6-3 veranschaulicht ein Beispiel für den Einsatz eines `ShapeWithSelectableFigure`. Darin ist vier Mal derselbe Prozess mit einem angehefteten *Report* als Ausgangsdatum visualisiert. *(a)* stellt die nicht selektierte Variante dar, wie sie später im Editor erscheint. *(b)* hingegen zeigt anhand eines roten Kastens die Begrenzung der gesamten Prozess-`Figure`, die auch den Container für ausgehende Daten umfasst. Bei der Verwendung eines Standard-`Shapes` würde der Selektions-Rahmen wie bei *(c)* ausfallen. Wird jedoch das `ShapeWithSelectableFigure` benutzt und als `selectableFigure` das innere mit blauem Farbverlauf versehene Rechteck angegeben, so resultiert die in *(d)* abgebildete Umrandung nach der Auswahl des Prozesses.

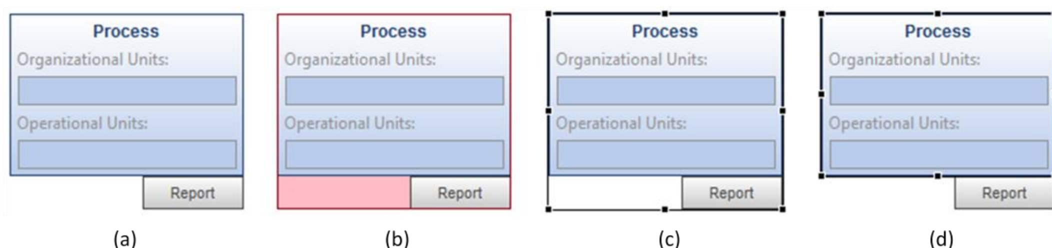

Abb. 6-3: Exemplarische Verwendung von `ShapeWithSelectableFigure`

- `Connection`: Das `Connection`-*Concept* spezialisiert `Shape` und repräsentiert eine Verbindung zwischen zwei `Figures`. Diese Verbindung wird als ein Verbund von Linien visualisiert, wobei der genaue Verlauf durch `Bendpoints` gesteuert wird. Zusätzliche Attribute sind wie folgt verfügbar:
  - `routingConstraint`: Menge von `Bendpoints`, die den genauen Linienverlauf zwischen Start- und Endpunkt der `Connection` beschreiben. Dieser etwas unglücklich gewählte Attribut-Name wird durch die Klasse *PolylineConnection* von Draw2D vorgegeben.
  - `sourceDecoration`: Optionale Verzierung (`Decoration`) am Startpunkt der `Connection` (z.B. Diamant).
  - `targetDecoration`: Optionale Verzierung (`Decoration`) am Endpunkt der `Connection` (z.B. Pfeilspitze).
- `Decoration`: Verzierung die zur Platzierung auf einer `Connection` gedacht ist. Dieses *Concept* ist *abstrakt*, d.h. erst eine konkrete Spezialisierung legt das Aussehen fest.
- `Label`: Spezialisierte `Figure`, die zur Anzeige von Text und/oder eines Bildes dient. Dazu können folgende Attribute verwendet werden:
  - `text`: String-Attribut, das den anzuzeigenden Text aufnimmt.
  - `imageFilename`: String-Attribut, das den Dateinamen einer gültigen Bilddatei enthält; die Datei muss über eine File-URI von Eclipse referenziert werden (beginnt mit „platform:/“). Deshalb ist dieses Attribut Implementierungs-abhängig.
  - `alignment`: Ausrichtung von Bild und Text innerhalb des `Labels`.
  - `txtAlignment`: Ausrichtung des Textes relativ zum Bild innerhalb des `Labels`.
  - `txtPlacement`: Platzierung des Textes relativ zum Bild innerhalb des `Labels`.
  - `imagePlacement`: Ausrichtung des Bildes innerhalb des `Labels`.
  - `imageTextGap`: Größe der Lücke zwischen Bild und Text.

6.3 Editor-Meta-Model

Das *Editor-Meta-Model* beschreibt diejenigen Elemente, die in einem grafischen Editor verwendet werden dürfen. Die Benutzerinteraktion erfolgt ausschließlich mit diesen `EditorElements` (Abb. 6-8). Außerdem bestimmen die `EditorElements` das Mapping zwischen Elementen des *Domain-Meta-Models* und Formen des *Graphical-Meta-Models*. Die Elemente des Domänen-Modells werden in den kommenden Abschnitten als Modellelemente bezeichnet. Da zur Implementierung das Graphical Editing Framework benutzt wird, sollen die *Concepts* des *Editor-Meta-Models* direkt auf Controller-Klassen (*EditParts*, Abschnitt 4.3) der MVC-Architektur von GEF abgebildet werden (Abb. 5-1). Eine gewisse Ähnlichkeit mit den Standard-*EditParts* ist somit durchaus Absicht.

6.3.1 Identifikation möglicher Mappings zwischen Domäne und grafischer Darstellung

Vor der Spezifikation der *Concepts* des *Editor-Meta-Models* sind zunächst die möglichen Mappings zwischen den Elementen des Domänen-Meta-Modells und den `Figures` des *Graphical-Meta-Models* zu identifizieren. Dabei geht es um die grafische Darstellung der Modellelemente, um die Visualisierungsmöglichkeiten von Beziehungen zwischen diesen Elementen und der Anzeige ihrer internen Zustände.

6.3.1.1 Visualisierung von Modellelementen

Modellelemente und damit *Concepts* des Domänen-Modells können prinzipiell auf drei verschiedene Weisen visualisiert werden. Der am häufigsten auftretende Fall ist sicherlich der des Knotens. Hierbei wird ein Modellelement in Form einer unabhängigen `Figure` wiedergegeben. Unabhängig bedeutet, dass sie zwar Kinder besitzen, jedoch nicht als `Connection` eingesetzt werden darf. Für jeden Typ des Domänen-Meta-Modells, dessen Instanzen angezeigt werden sollen, ist eine Abbildung auf eine bestimmte `Figure` in Form eines Knotens zu spezifizieren. Dadurch wird Anforderung (d) aus Abschnitt 1.1 erfüllt. Ein Beispiel ist das bereits oben erwähnte blaue Rechteck, das einen Prozess symbolisiert.

Die Zeichenfläche – auch Diagramm genannt – muss ebenfalls mit einem Modellelement verbunden sein. Dadurch wird der genaue Inhalt für die Visualisierung festgelegt, d.h. welche Elemente werden im Designer angezeigt. Als dritte Darstellungsform eines *Concepts* existiert die Verbindung zweier Knoten. Die beiden zuletzt genannten Mappings fallen allerdings eher in den Bereich der Visualisierung von Beziehungen und werden deshalb im nächsten Abschnitt behandelt.

6.3.1.2 Visualisierung von Beziehungen

Angelehnt an [22] können Beziehungen zwischen Elementen auf zwei verschiedene Arten visualisiert werden (Abschnitt 2.3). Die Geometrie-basierte Variante wird im Grunde von jedem Designer (zumindest in rudimentärer Weise) unterstützt, da das Diagramm als Container für `Figures` dient. Sie sind über einen Knoten einem Modellelement zugeordnet, wodurch eine hierarchische Beziehung ausdrückt wird. Demnach muss die Zeichenfläche ebenfalls durch ein Modellelement repräsentiert werden, das die Wurzel der *Concept*-Hierarchie auf der Seite des Domänen-Modells darstellt. Welche Elemente im Einzelnen hinzugefügt werden dürfen hängt in erster Linie von den zuweisbaren *ConceptTypeAttributes* ab. Gleichzeitig ist anzugeben, welchem Attribut ein neues Modellelement zugewiesen werden soll, sobald ein entsprechender Knoten dem Diagramm hinzugefügt wird. Das Konzept des Hinzufügens von Knoten zum Diagramm lässt sich verallgemeinern. Ein Knoten kann nämlich sog. Kammern besitzen, die wiederum Knoten enthalten dürfen. Auch hier ist anzugeben, welchem Attribut des Vater-Knotens ein Modellelement zugewiesen werden soll, sobald ein entsprechender Knoten der Kammer hinzugefügt wird. Somit wird Anforderung (e) ebenfalls berücksichtigt.

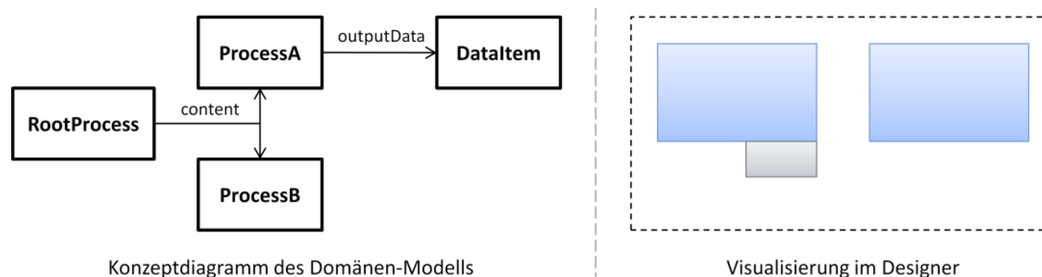

Abb. 6-4: Visualisierung von Beziehungen in einem Prozessmodell nach der Geometrie-basierten Variante

Abb. 6-4 zeigt ein Beispiel der Geometrie-basierten Variante zur Visualisierung von Beziehungen in einem Prozessmodell. Dieses Modell ist auf der linken Seite in Form eines Konzeptdiagramms abgebildet und beinhaltet die drei Prozess-Entitäten `RootProcess`, `ProcessA` und `ProcessB` sowie ein `DataItem`. `RootProcess` referenziert über das `content`-Attribut die beiden Prozesse `ProcessA` und `ProcessB`. Ferner steht `ProcessA` über `outputData` mit dem `DataItem` in Beziehung. Beide Attribute werden Geometrie-basiert visualisiert. Als Diagramm (gestrichelte Umrandung), das `ProcessA` als linkes und `ProcessB` als rechtes blaues Rechteck enthält, kommt dabei der `RootProcess` zum Einsatz. Das `DataItem` ist in Form eines grauen Kastens dargestellt, der auf-

grund der von `ProcessA` ausgehenden `outputData`-Referenz am linken blauen Rechteck anhaftet. Dieses Anhaften wird mit Hilfe einer Kammer am Rand des Prozess-Knotens realisiert.

Abb. 6-5: Visualisierung einer Attribut-basierten Beziehung in einem Prozessmodell nach der Graph-basierten Variante

Zur Erfüllung von Anforderung (f) aus Abschnitt 1.1 muss die Graph-basierte Klasse umgesetzt werden. Dabei gilt es zwischen zwei Ausdrucksmöglichkeiten auf der Seite des Domänen-Modells zu unterscheiden. Die einfachste Variante ist die Attribut-basierte Verbindung. Eine Verbindung zwischen zwei Knoten wird genau dann angelegt, wenn dem gemappten Attribut des zum Startknoten gehörenden Modellelements das Modellelement des Zielknotens zugewiesen wird. In Abb. 6-5 ist ein Beispiel dieses Sachverhalts veranschaulicht. Demgemäß erfolgt die Visualisierung eins zu eins, d.h. eine Referenz wird direkt auf eine `Connection` abgebildet. Im vorliegenden Fall resultiert aus der Zuweisung von `ProcessB` an das `successors`-Attribut von `ProcessA` der graue Pfeil zwischen den beiden blauen Rechtecken. Sie symbolisieren analog zu Abb. 6-4 die beiden Prozesse des Konzeptdiagramms.

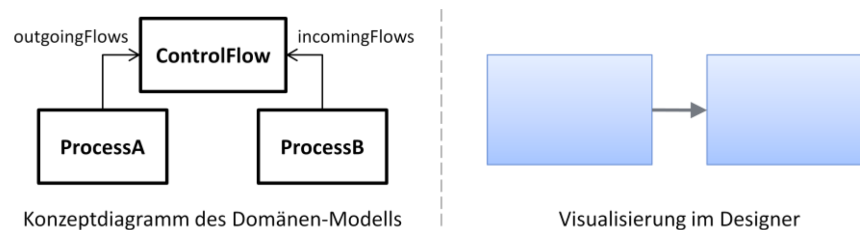

Abb. 6-6: Visualisierung einer Entität-basierten Beziehung in einem Prozessmodell nach der Graph-basierten Variante

Die Entität-basierte Verbindung ist die zweite Variante der Graph-basierten Klasse. Hierbei wird eine Verbindung durch ein konkretes Modellelement ausgedrückt, das von zwei anderen Modellelementen über jeweils eine Attributzuweisung referenziert wird. Handelt es sich bei den referenzierenden Elementen um Instanzen desselben Typs, so muss sowohl für eingehende als auch für ausgehende Verbindungen ein separates Attribut gemappt werden. Abb. 6-6 demonstriert diesen Sachverhalt exemplarisch anhand eines Prozessmodells mit zwei Prozessen, die durch einen `ControlFlow` von `ProcessA` nach `ProcessB` miteinander verbunden sind. Die Richtung dieses Flusses ist durch die beiden Attribute `outgoingFlows` und `incomingFlows` vorgegeben. Ersteres ist beim Mapping als Attribut für ausgehende Verbindungen spezifiziert, wohingegen Letzteres für eingehende Verbindungen verantwortlich ist. Das Resultat im Designer ist mit dem aus Abb. 6-5 identisch, d.h. die Beziehung wird wieder durch einen grauen Pfeil zwischen den beiden Kästen, die die Prozesse repräsentieren, dargestellt.

6.3.1.3 Visualisierung von Zuständen

Jedes Modellelement besitzt einen internen Zustand, der durch den konkreten Inhalt der Instanz-Facette gegeben ist. Dieser ist letztlich über die Wertzuweisungen an die zur Verfügung stehenden Attribute definiert. Bei der Visualisierung eines *Concepts* kann es durchaus sinnvoll sein, dessen Zustand ebenfalls auf verschiedene Arten anzeigen zu können. Damit erfolgt die Berücksichtigung von Anforderung (g) aus Abschnitt 1.1.

Die gebräuchlichste Art ist die Anzeige eines bestimmten Attribut-Wertes in Textform. Auf der Seite des *Graphical-Meta-Models* eignet sich dafür das `Label`-*Concept*. Demnach wird im *Editor-Meta-Model* ein `EditorElement` benötigt, das den Wert eines gemappten Attributs an das zugehörige `Label` delegiert. Angewendet auf die obigen Prozessbeispiele (Abb. 6-4, Abb. 6-5 und Abb. 6-6) kann damit der Name eines Prozesses – sofern ein entsprechendes Attribut vorhanden ist – auf dem jeweiligen Rechteck angezeigt werden.

Eine andere Möglichkeit zur Darstellung des internen Zustands ist die Modifikation von Parametern der korrespondierenden `Figure`. In anderen Frameworks (z.B. GME und GMF) bedient man sich zur Realisierung einer derartigen Funktionalität beim integrierten Constraint-System. Das LMM bietet bislang aber noch keine Unterstützung zur Spezifikation und Ausführung beliebiger Constraints. Deshalb sieht das *Editor-Meta-Model* lediglich vor, bestimmte grafische Eigenschaften auf der Basis von Wahrheitswerten zu beeinflussen. Diese können beispielsweise direkt aus Zuweisungen an boolesche Attribute oder indirekt aus beliebigen Berechnungen stammen. Letztere sind bei Bedarf zu implementieren. Ein Beispiel für einen solchen Anwendungsfall ist eine Markierung in Form eines kleinen Quadrats in der rechten unteren Ecke der Prozess-`Figure`, die angezeigt wird, sofern der zugehörige Prozess im Domänen-Modell komposit ist, d.h. weitere Prozesse enthält.

Abb. 6-7: Visualisiertung des Zustands zweier Prozesse

Abb. 6-7 zeigt zwei Prozesse mit den Namen „Atomic Process“ und „Composite Process“. Letzterer enthält – wie der Name vermuten lässt – weitere Sub-Prozesse und ist infolgedessen mit einem kleinen blauen Quadrat in der rechten unteren Ecke versehen.

6.3.2 Spezifikation des Inhalts

Nachdem im vorangegangenen Abschnitt alle relevanten Mappings zwischen *Domain-Meta-* und *Graphical-Meta-Model* identifiziert wurden, erfolgt im weiteren Verlauf die genaue Spezifikation der Elemente des *Editor-Meta-Models* (Abb. 6-8). Sie legen im Wesentlichen die Abbildung zwischen den Elementen der beiden anderen Meta-Modelle (*Graphical-Meta-Model* und *Domain-Meta-Model*) fest und können in zwei Kategorien eingeteilt werden. Die erste umfasst die Struktur-gebenden *Concepts*, die noch keine vollständige Abbildungsvorschrift aufweisen und ausschließlich zur Extraktion und Wiederverwendbarkeit einer bestimmten Teilsemantik dienen:

- `EditorElement`: Wurzel-Element der *Concept*-Hierarchie der *Editor-Meta-Model* Elemente.
- `DiagramElement`: Spezialisiertes `EditorElement`, das ein beliebiges Modellelement kapselt.
  - `modelElementFQN`: Voll qualifizierter Name eines *Concepts* des Domänen-Meta-Modells. Dieses Meta-*Concept* muss im *Editor-Definition-Model* gesetzt werden, wohingegen das konkrete Modellelement des Domänen-Modells im *Editor-Usage-Model* durch das MDF gesetzt wird.
  - `toolingTitle`: Legt den in der Werkzeugpalette verwendeten Anzeige-Name (String) für das mit `modelElementFQN` referenzierte *Concept* fest. Dieses Attribut ist optional.

- `toolingIcon`: Legt den Dateinamen (String) eines Icons fest, das in der Werkzeugpalette das mit `modelElementFQN` referenzierte *Concept* symbolisiert. Dieses Attribut ist optional.
- `toolingAttributeQN`: Name eines Attributs mit literalem Typ, das im durch `modelElementFQN` referenzierten *Concept* deklariert wird; dieser Parameter wird ausschließlich bei der Anwendung des Typ-Verwendungs-Konzepts interpretiert und zur Bestimmung des Namens eines Tools verwendet, das einen bereits existierenden Typ repräsentiert. Mit diesem Tool kann eine Verwendung des gekapselten Typs erzeugt werden. Das Attribut ist optional.
- `contentEUModelURI`: Darf nur durch das Model Designer Framework in einem *Editor-Usage-Model* gesetzt werden; es enthält die URI (String) eines anderen *Editor-Usage-Models*, das den Inhalt des mit `modelElementFQN` referenzierten *Concepts* visualisiert. Auf diese Weise lassen sich hierarchische Beziehungen in Form mehrerer Diagramme ausdrücken. Beispielsweise kann ein Prozess weitere Kind-Prozesse enthalten, die beim Öffnen des Eltern-Prozesses in einem separaten Diagramm (z.B. durch einen Doppelklick) angezeigt werden.
- `subElements`: Jedes `DiagramElement` kann über mehrere `SubElements` verfügen. Sie bilden den internen Zustand des zum `DiagramElement` gehörenden Modellelements auf konkrete `Figures` ab. Dieses Attribut ist optional.

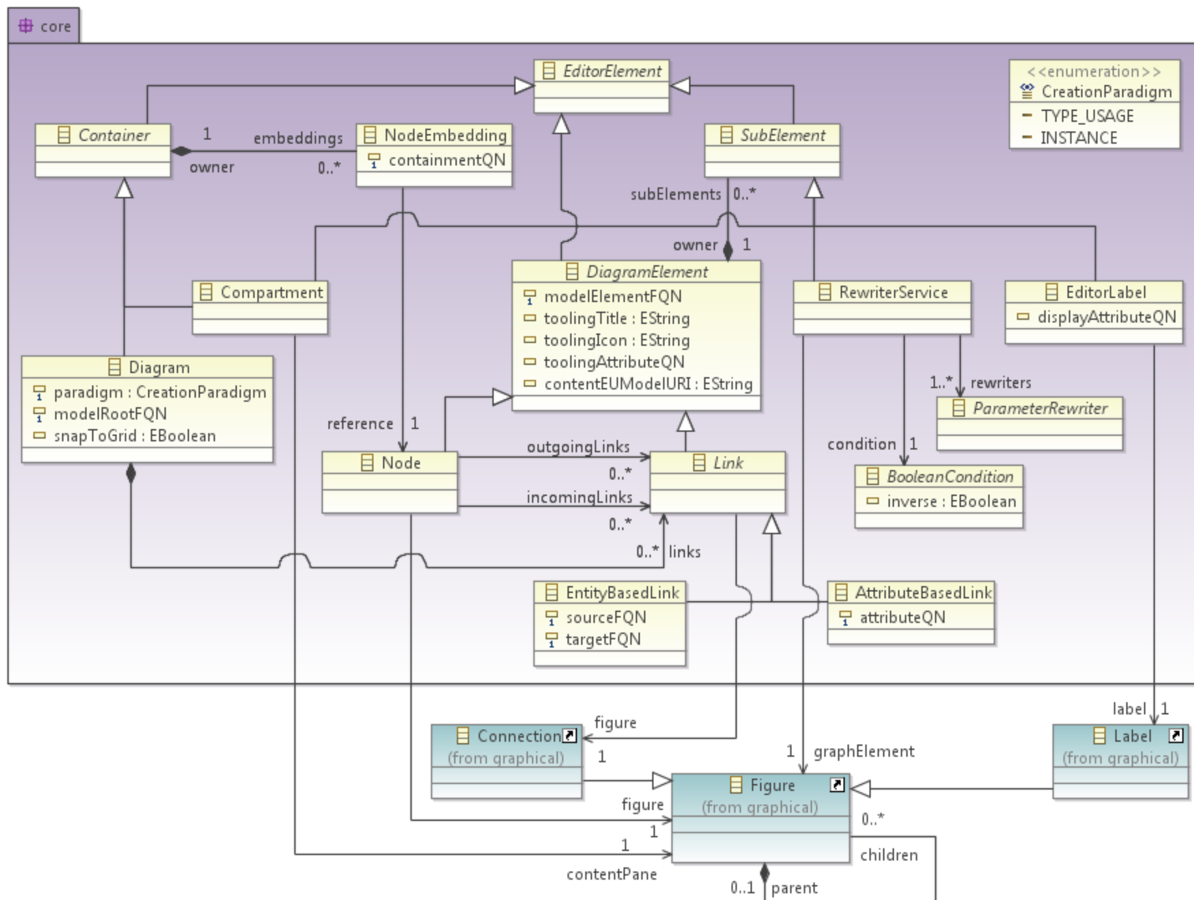

Abb. 6-8: Konzeptdiagramm des Editor-Meta-Models

- SubElement: Abbildung eines Teilaspekts des internen Zustands eines Modellelements auf eine konkrete Figure; das owner-Attribut legt das DiagramElement fest, zu dem ein SubElement gehört. Über dieses DiagramElement ist gleichzeitig ein Modellelement spezifiziert, von dessen Zustand ein Teilaspekt visualisiert wird.
- Link: Spezialisiertes DiagramElement für die Abbildung einer Beziehung zwischen zwei Modellelementen auf eine über das figure-Attribut spezifizierte Connection. Welches Beziehungs-Mapping zum Einsatz kommt wird erst durch entsprechende Spezialisierungen dieses abstrakten *Concepts* festgelegt.
- Container: Basis-*Concept*, das die Fähigkeit besitzt Nodes über NodeEmbeddings aufzunehmen. Ein Container ist stets der Eigentümer seiner über das embeddings-Attribut referenzierten NodeEmbeddings. Dieses Attribut ist optional.

In der zweiten Kategorie befinden sich diejenigen Elemente, die eine vollständige Mapping-Semantik besitzen, wie sie in Abschnitt 6.3.1 beschrieben ist:

- CreationParadigm: ***Enumeration***, die die zwei in Abschnitt 5.1 vorgestellten möglichen Modellierungsansätze auflistet. Das sind TYPE_USAGE Creation und INSTANCE Creation.
- Diagram: Spezialisierter Container, der die Zeichenfläche des Designers darstellt und das darauf abgebildete Modellelement spezifiziert. Außerdem wird damit das zu verwendende Modellierungsparadigma auf der Seite des Domänen-Modells festgelegt.
    - paradigm: CreationParadigm, das den Modellierungsansatz beim Erzeugen von Modellelementen festlegt und damit Anforderung (c) aus Abschnitt 1.1 erfüllt.
    - modelRootFQN: Voll qualifizierter Name des Modellelements, das von der Zeichenfläche repräsentiert wird.
    - links: Enthält alle Arten von Links, die in diesem Diagram verwendet werden dürfen. Dieses Attribut ist optional.
    - snapToGrid: Boolesches Attribut, das festlegt, ob EditorElements an einem Raster ausgerichtet werden. Dieses Attribut ist optional.
- Node: Spezialisiertes DiagramElement, das einem in Abschnitt 6.3.1.1 beschriebenen Knoten entspricht. Er bildet ein Modellelement auf eine beliebige Figure ab.
    - figure: Figure, durch die das zugehörige Modellelement visualisiert wird.
    - outgoingLinks: Ausgehende Links; dieses Attribut wird automatisch durch das MDF im *Editor-Usage-Model* gesetzt.
    - incomingLinks: Eingehende Links; dieses Attribut wird automatisch durch das MDF im *Editor-Usage-Model* gesetzt.
- NodeEmbedding: Legt das Attribut des Modellelements des zugehörigen Containers fest, zu dem das Modellelement des referenzierten Nodes hinzugefügt werden soll. Mit Hilfe verschiedener NodeEmbeddings kann derselbe Node-Typ innerhalb mehrerer Containers verwendet werde.
    - containmentQN: Name des Attributs, zu dem das Modellelement des referenzierten Nodes hinzugefügt wird.

- owner: Container, dessen zugehöriges Modellelement das mit containmentQN spezifizierte Attribut enthalten muss.
  - reference: Node, der das hinzuzufügende Modelelement kapselt.
- AttributeBasedLink: Hierbei handelt es sich um eine Attribut-basierte Verbindung, wie sie in Abschnitt 6.3.1.2 identifiziert worden ist. Das attributeQN-Attribut legt das Attribut des Modellelements fest, das gesetzt wird sobald eine entsprechende Verbindung angelegt wird. Die erlaubten Ziel-Modellelemente sind über den Typ des durch attributeQN referenzierten Attributs spezifiziert. Zu beachten ist, dass der voll qualifizierte Name des modelElementFQN-Attributs auf das Meta-*Concept* zeigen muss, welches das mittels attributeQN angegebene Attribut deklariert.
- EntityBasedLink: Hierbei handelt es sich um eine Entität-basierte Verbindung, wie sie in Abschnitt 6.3.1.2 identifiziert worden ist. Der modelElementFQN muss auf das Meta-*Concept* zeigen, das die Verbindungs-Entität repräsentiert. Beim Erzeugen der Verbindung wird zunächst die konkrete Verbindungs-Entität, d.h. eine Instanz des angegebenen Meta-*Concepts*, angelegt.
  - sourceFQN: Voll qualifizierter Name des Attributs, dessen deklarierendes *Concept* den Ursprungstyp für diese Verbindung darstellt; beim Erzeugen der Verbindung wird dem referenzierten Attribut im zugehörigen Modellelement die konkrete Entität zugewiesen.
  - targetFQN: Voll qualifizierter Name des Attributs, dessen deklarierendes *Concept* den Zieltyp für diese Verbindung darstellt; beim Erzeugen der Verbindung wird dem referenzierten Attribut im zugehörigen Modellelement die konkrete Entität zugewiesen.
- Compartment: Spezialisiertes SubElement und gleichzeitig spezialisierter Container; dieses Element kann weitere Nodes aufnehmen, deren Figures innerhalb der durch das contentPane-Attribut referenzierten Figure angezeigt werden. Sie muss eine direkte oder indirekte Kind-Figure der durch das zugehörige DiagramElement (über owner-Attribut festgelegt) gegebenen Figure sein.
- EditorLabel: Spezialisiertes SubElement, das den Wert eines bestimmten Attributs in Textform auf einem Label anzeigt.
  - displayAttributeQN: Name eines Attributs, das im Meta-*Concept* des zugehörigen DiagramElements (über owner-Attribut festgelegt) deklariert wird. Der Wert, der diesem Attribut im zugehörigen Modellelement zugewiesen ist, wird auf der durch label angegebenen Figure angezeigt.
  - label: Label, das zur Anzeige des durch displayAttributeQN referenzierten Attributwerts dient.
- RewriterService: Spezialisiertes SubElement, das auf Basis eines booleschen Wertes bestimmte Parameter der referenzierten Figure beeinflusst. Die Semantik wird erst durch eine spezialisierte BooleanCondition sowie einen spezialisierten ParameterRewriter festgelegt.

- condition: Spezielle BooleanCondition, die den internen Zustand des Modellelements, das vom zugehörigen DiagramElement (über owner-Attribut festgelegt) referenziert wird, auswertet und darauf basierend einen booleschen Wert zurückliefert.
  - rewriters: Spezielle ParameterRewriters, die auf Grundlage der spezifizierten BooleanCondition bestimmte Parameter von graphElement beeinflussen.
  - graphElement: Figure, deren Parameter bei Bedarf modifiziert werden sollen.
- BooleanCondition: Liefert einen booleschen Wert zurück, der auf einem Teilaspekt des internen Zustands eines Modellelements basiert; um welchen Teilaspekt es sich handelt wird erst von einer konkreten Spezialisierung definiert. Durch auf true-Setzen des inverse-Attributs wird der ermittelte boolesche Wert vor dem Zurückgeben invertiert.
- ParameterRewriter: Zuständig für das Ändern eines Figure-Parameters; der konkrete Parameter wird erst von einer Spezialisierung dieses *Concepts* festgelegt.

Instanzen dieser Elemente werden zur Erstellung von *Editor-Definition-Models* verwendet. Damit ein Modell vom MDF als *Editor-Definition-Model* interpretiert werden kann, muss genau eine Instanz des Diagram-*Concepts* definiert werden. Sie dient als Vorlage (Typ) für eine konkrete Diagram-Verwendung in einem *Editor-Usage-Model*.

Kapitel 7 Anwendungsfall: Definition eines Prozess-Designers

In diesem Kapitel wird die Vorgehensweise zur Definition eines grafischen Prozess-Designers unter Verwendung des Model Designer Frameworks erläutert, das auf den Meta-Modellen aus Kapitel 6 beruht. Diesem spezifischen Designer liegt ein an die perspektivenorientierte Prozessmodellierung (Abschnitt 2.4) angelehntes – jedoch zu Gunsten einer besseren Übersichtlichkeit stark vereinfachtes – Domänen-Meta-Modell zugrunde (Abb. 7-1). Die ursprüngliche Version des POPM-Meta-Modells entstammt der Dissertation von Volz [6].

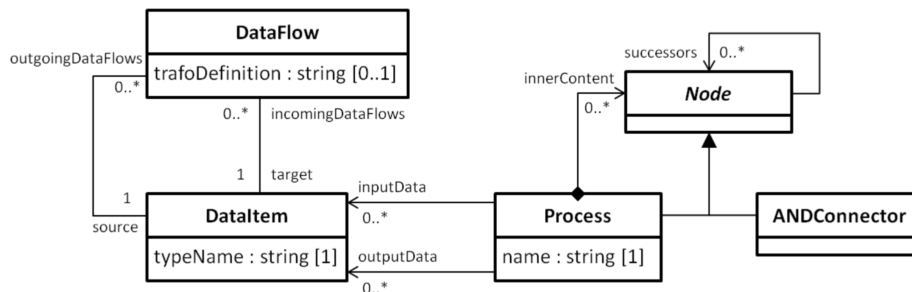

Abb. 7-1: Vereinfachtes POPM-Meta-Modell

Das zentrale *Concept* `Process` ist ein `Node` und kann sich aus mehreren solcher `Nodes` über das `innerContent`-Attribut zusammensetzen. Daraus ergibt sich die Funktionale Perspektive. Ein `Node` ist ein Element, das beliebig viele Nachfolger-`Nodes` mittels des `successors`-Attributs referenzieren kann. Dieser Sachverhalt zusammen mit dem `ANDConnector`-*Concept* repräsentiert die Verhaltens-bezogene Perspektive. Ein `Process` kann zudem über Eingangs- und Ausgangsdaten verfügen, was durch die Attribute `inputData` bzw. `outputData` ausgedrückt wird. Des Weiteren ist in seiner Typ-Facette das obligatorische String-Attribut `name` deklariert. Bei den Eingangs- und Ausgangsdaten handelt es sich stets um `DataItems`, die einen `typeName` (String-Attribut) besitzen. Jedes `DataItem` kann sowohl ein- als auch ausgehende `DataFlows` aufweisen. Beide *Concepts* zusammen stellen die Datenbezogene Perspektive dar. Ein `DataFlow` kann über eine Transformations-Definition verfügen, die über das String-Attribut `trafoDefinition` anzugeben ist, und kennt zusätzlich durch `source`- und `target`-Attribute die beiden `DataItems`, die er verbindet.

Die folgenden Abschnitte decken aufeinander aufbauend die einzelnen Schritte ab, die zur Spezifikation eines Designers notwendig sind, beginnend bei der Definition des Diagramms bis hin zur Angabe eines Mappings von Beziehungen zwischen Modellelementen. An geeigneten Stellen werden außerdem die verschiedenen Mappings durch Screenshots eines entsprechenden Designers veranschaulicht, der den Inhalt eines zum jeweiligen Szenario passenden Prozessmodells visualisiert.

Ausgangspunkt zur Definition eines von Grund auf neuen Designers ist stets das Vorhandensein eines Domänen-Meta-Modells, was bei uns durch das POPM-Meta-Modell (Abb. 7-1) gewährleistet ist. Dazu müssen zwei weitere Modelle, nämlich das *Graphical-Definition-* und das *Editor-Definition-Model* angelegt werden. Da das *Editor-Definition-Model* das Mapping zwischen den Elementen der beiden anderen Modelle spezifiziert, muss Ersteres diese beiden Modelle referenzieren. Darüber hinaus ist vorgegeben, dass sowohl das *Graphical-Definition-* als auch das *Editor-Definition-Model*

eine Ebene D1 besitzen, die die relevanten *Concepts* enthalten. Die vollständigen Modelle dieses Kapitels befinden sich im Anhang A.

7.1 Definition des Prozess-Diagramms

Wie oben erwähnt müssen alle gültigen *Editor-Definition-Models* über eine Diagram-Instanz verfügen, d.h. zunächst gilt es ein *Concept* des Domänen-Meta-Modells zu identifizieren, das durch das Diagramm dargestellt wird. Bei POPM ist jedes Prozessmodell wiederum in einem sog. Wurzel-Prozess enthalten, d.h. die Zeichenfläche repräsentiert einen konkreten Prozess. Nachdem das Typ-Verwendungs-Konzept zum Einsatz kommt, ist der Wurzel-Prozess stets ein Typ und alle darin enthaltenen Knoten (im Sinne der Domäne) sind Verwendungen bereits existierender Typen. Demnach muss als modelRootFQN der voll qualifizierte Name des Process-*Concepts* eingesetzt und dem paradigm-Attribut TYPE_USAGE zugewiesen werden (Listing 7-1). Der FQN setzt sich aus dem *Model*-, dem *Level*-, dem/den *Package*(s)- und dem *Concept*-Namen zusammen. Bezogen auf den modelRootFQN von Listing 7-1 liegt das Process-*Concept* in einem *Package* mit dem Namen POPM, dieses *Package* befindet sich im *Level* M2 und dieses *Level* ist schließlich im *Model* POPMMetaModel enthalten.

```
concept ProcessDiagram instanceOf Editor.D2.Core.Diagram {
     values {
          enum paradigm = TYPE_USAGE;
          fqn modelRootFQN = POPMMetaModel.M2.POPM.Process;
     }
}
```

Listing 7-1: Definition des Prozess-Diagram

Bei der automatischen Erzeugung eines *Editor-Usage-Models* auf Basis des gegebenen *Editor-Definition-Models* muss zunächst ein Process-Typ erstellt werden. Dazu muss der entsprechende Eintrag im Kontext-Menü des *Editor-Definition-Models* ausgewählt werden, worauf ein spezieller Dialog erscheint, der für jedes Attribut mit literalem Typ ein Eingabefeld bereitstellt (Abb. 7-2, links). Nach dem Bestätigen wird ein leerer Designer mit leerer Tool-Palette geöffnet (Abb. 7-2, rechts).

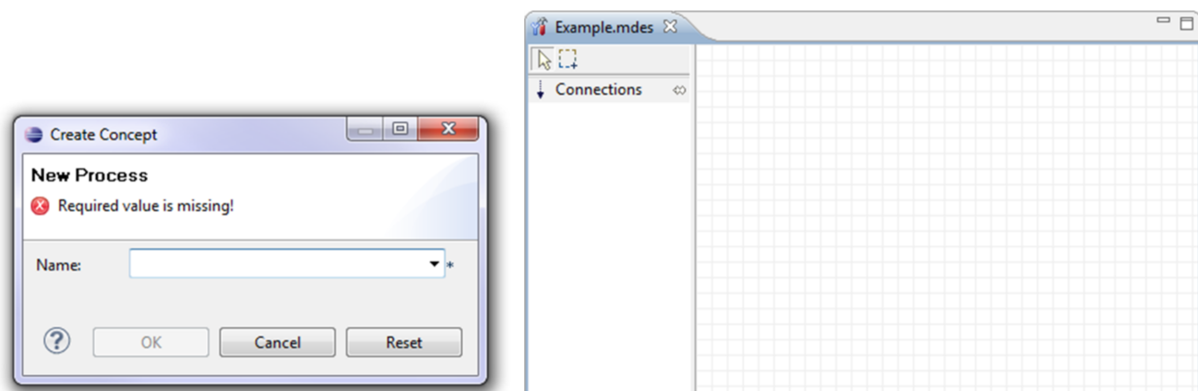

Abb. 7-2: Dialog zum Erzeugen des Diagramm-Modellelements und der sich anschließende leere Designer

Den entsprechenden Inhalt von *Domain-* und *Editor-Usage-Model* zeigt Abb. 7-3. Daraus ist ersichtlich, dass im *Editor-Usage-Model* das im zugehörigen *Editor-Definition-Model* definierte ProcessDiagram konkret verwendet wird. Ferner muss diese Verwendung den im Domänen-Model neu angelegten Process-Typ über das modelRootFQN-Attribut referenzieren und damit als Wurzel festlegen. Da es für die Benennung von anonymen Verwendungen noch keine einheitlichen Regeln gibt, wird in den folgenden Konzeptdiagrammen die Benennung von Objekten in der UML erweitert. Der vollständige Bezeichner einer Verwendung ist somit wie folgt aufgebaut:

[Verwendungsname] : [Instanzname] : [Typname]. In Anlehnung an die UML kann der konkrete Verwendungsname leer sein. Durch diese Namensgebung vereinfacht sich die Struktur der Konzeptdiagramme, weil nicht bei jeder Verwendung der entsprechende Typ mittels *concreteUseOf* referenziert werden muss. Zudem sind zu Gunsten einer größeren Übersichtlichkeit alle Verwendungen in den nachfolgenden Abbildungen mit einer blauen Umrandung versehen.

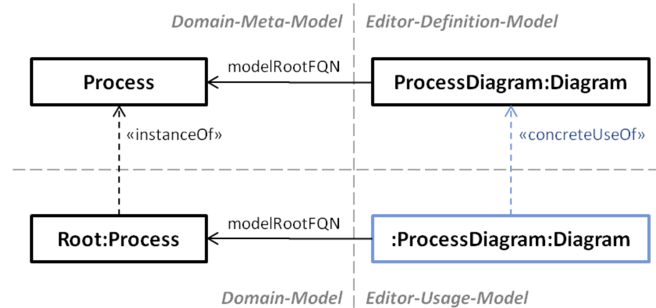

Abb. 7-3: Domain-Meta-, Editor-Definition, Domain- und Editor-Usage-Model für leeren Designer

7.2 Definition des Prozess-Knotens

Ein Prozess-Knoten ist ein `Node` des *Editor-Definition-Models*, das einen `Process` des POPM-Meta-Modells repräsentiert. Bevor ein derartiger `Node` allerdings definiert werden kann, muss vorab seine Visualisierung spezifiziert werden. Das MDF bietet dazu ein eigenes *Extended Graphical-Meta-Model* mit zahlreichen spezialisierten `Figures`, `Layouts` und passenden `LayoutConstraints` an, das als `CommonGraphics` bezeichnet wird. Darin gibt es beispielsweise ein `GradientRectangle`, was ein Rechteck mit Farbverlauf-Füllung darstellt. Überdies werden die meisten von Draw2D bekannten grafischen Elemente zur Verfügung gestellt.

```
concept ProcessFigure instanceOf Graphical.D2.Core.ShapeWithSelectableChild {
     values {
          concept backgroundColor = ProcessBackColor;
          concept foregroundColor = ProcessForeColor;
          concept layoutManager = ProcessBorderLayout;
          concept children = ProcessGraphicalDefinition.D1.POPM.Process.Center.ProcessRect;
          concept selectableSubFigure = ProcessGraphicalDefinition.D1.POPM.Process.Center.ProcessRect;
     }
}
concept ProcessBackColor instanceOf Graphical.D2.Core.Types.Color {
     values {
          integer red = 196; integer green = 226; integer blue = 242;
     }
}
concept ProcessForeColor instanceOf Graphical.D2.Core.Types.Color {
     values {
          integer red = 24; integer green = 81; integer blue = 112;
     }
}
concept ProcessBorderLayout instanceOf CommonGraphics.D2.Common.Layout.BorderLayout {}

package Center {
     concept ProcessRect instanceOf CommonGraphics.D2.Common.Figure.GradientRectangle {
          values {
               concept constraint = ProcessRectConstraint;
          }
     }
     concept ProcessRectConstraint
               instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
          values {
               integer value = 2; // center region
          }
     }
}
```

Listing 7-2: Definition der Prozess-`Figure`

Ein Prozess soll in Form eines Rechtecks mit blauem Farbverlauf angezeigt werden. Da später am Rand dieses Rechtecks DataItems anhaften sollen, wird als Wurzel der Figure-Hierarchie ein ShapeWithSelectableChild verwendet. Die GradientRectangle-Instanz, die für die eigentliche Visualisierung zuständig ist, wird dem ShapeWithSelectableChild als Kind zugewiesen und gleichzeitig als selectableSubFigure proklamiert. Listing 7-2 zeigt die komplette Definition der Prozess-Figure, wie sie momentan erforderlich ist.

Bei der Definition des Prozess-Nodes (Listing 7-3) muss die soeben spezifizierte Figure referenziert und als modelElementFQN erneut der voll qualifizierte Name des Process-*Concepts* angegeben werden. Ferner ist es sinnvoll ein Attribut des Process-*Concepts* über toolingAttributeQN festzulegen, das den Namen der Process-Typen zur Anzeige in der Tool-Palette enthält. Hierfür bietet sich das name-Attribut an. Bleibt das Attribut leer, so wird ein MDF-interner Bezeichner verwendet, der eine kryptische Endung besitzt.

Damit Prozess-Nodes dem Diagram hinzugefügt werden können, muss ein zugehöriges NodeEmbedding definiert und dem Diagram bekannt gemacht werden (Listing 7-3). Dieses legt fest, dass eine neu hinzugefügte Process-Verwendung dem innerContent-Attribut des Process-Typs, das vom ProcessDiagram repräsentiert wird, zugewiesen werden soll.

```
concept ProcessDiagram instanceOf Editor.D2.Core.Diagram {
     values {
          enum paradigm = TYPE_USAGE;
          fqn modelRootFQN = POPMMetaModel.M2.POPM.Process;
          concept embeddings = ProcessNodeEmbedding;
     }
}

concept ProcessNodeEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
     values {
          concept reference = ProcessEditorDefinition.D1.POPM.Nodes.Process.ProcessNode;
          fqn containmentQN = innerContent;
     }
}

package Nodes {
     package Process {
          concept ProcessNode instanceOf Editor.D2.Core.Node {
               values {
                    concept figure = ProcessGraphicalDefinition.D1.POPM.Process.ProcessFigure;
                    fqn modelElementFQN = POPMMetaModel.M2.POPM.Process;
                    fqn toolingAttributeQN = name;
                    string toolingTitle = "Process";
               }
          }
     }
}
```

Listing 7-3: Definition des Prozess-Nodes inkl. eines zugehörigen NodeEmbeddings

Analog zur Vorgehensweise beim Process-*Concept* lassen sich auch die Figure-Hierarchie und das Mapping für das ANDConnector-*Concept* definieren. Als Darstellungsform soll eine violette Ellipse mit dem statischen Schriftzug „AND“ verwendet werden. Nachdem ein ANDConnector ausschließlich über das successors-Attribut verfügt und keine eingebetteten Elemente aufweist, kann als Figure direkt eine GradientEllipse-Instanz referenziert werden. Der Umweg über das ShapeWithSelectableFigure ist nicht notwendig.

Abb. 7-4 zeigt einen Screenshot des Prozess-Designers, der den Inhalt des als „Root“ bezeichneten Prozesses visualisiert. Er setzt sich aus einer Verwendung des bereits erzeugten Prozess-Typs mit dem Namen „Do Something“ und einer Verwendung eines ANDConnector-Typs zusammen. Des

Weiteren kann an dem Screenshot der Aufbau der Tool-Palette beim Einsatz des Typ-Verwendungs-Konzepts gut nachvollzogen werden. Für jeden `Node` des *Editor-Definition-Models* wird eine separate Gruppe angelegt, die die zugehörigen Typen beinhaltet. Ein Typ gehört genau dann zu einer Gruppe, wenn er eine direkte Instanz des Meta-Modellelements ist, das vom `Node` gekapselt wird, aus dem die Gruppe hervorgegangen ist. `Links` werden gesondert behandelt, da sie direkt instanziiert, d.h. nicht mit der Semantik des Typ-Verwendungs-Konzepts erzeugt werden. Deshalb liegen sie in der separaten *Connections*-Gruppe.

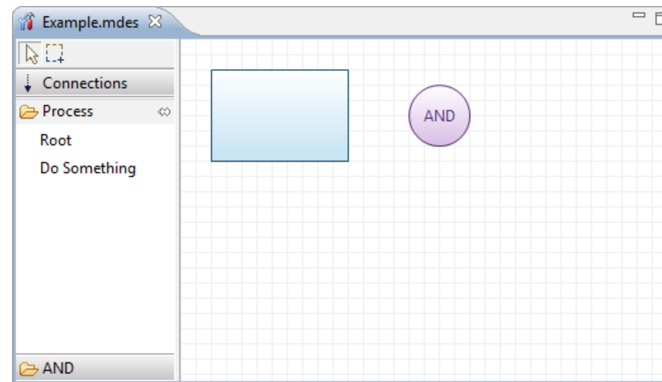

Abb. 7-4: Designer mit einer `Process`- und einer `ANDConnector`-Verwendung

Das in Abb. 7-5 dargestellte Konzeptdiagramm veranschaulicht den Inhalt von *Domain-*, *Graphical-Usage-* und *Editor-Usage-Model* des Prozess-Designers aus Abb. 7-4 unter Vernachlässigung des AND-Konnektors. Hieraus ist sofort ersichtlich, dass der Hauptaufwand für das Erzeugen relevanter *Concepts* auf der Seite von *Graphical-Usage-* und *Editor-Usage-Model* liegt. Beim Anlegen einer neuen `Process`-Verwendung müssen der korrespondierende `ProcessNode` sowie das referenzierende `ProcessNodeEmbedding` konkret verwendet werden. Für Letzteres ist ebenfalls anzugeben, zu welchem `Container` es gehört, was im Beispiel die Verwendung des `ProcessDiagram` ist. Außerdem müssen für sämtliche `Figures` der zugehörigen `Figure`-Hierarchie des *Editor-Definition-Models* entsprechende Verwendungen erzeugt werden. Dabei ist die Verschachtelung originalgetreu zu übernehmen. Das wird durch die Verwendungen von `ProcessFigure` und `ProcessRect` sowie die Beziehungen zwischen diesen beiden *Concepts* erreicht. Schließlich sind noch Position und Abmessung der Wurzel-`Figure` mit Hilfe des `bounds`-Attributs festzulegen.

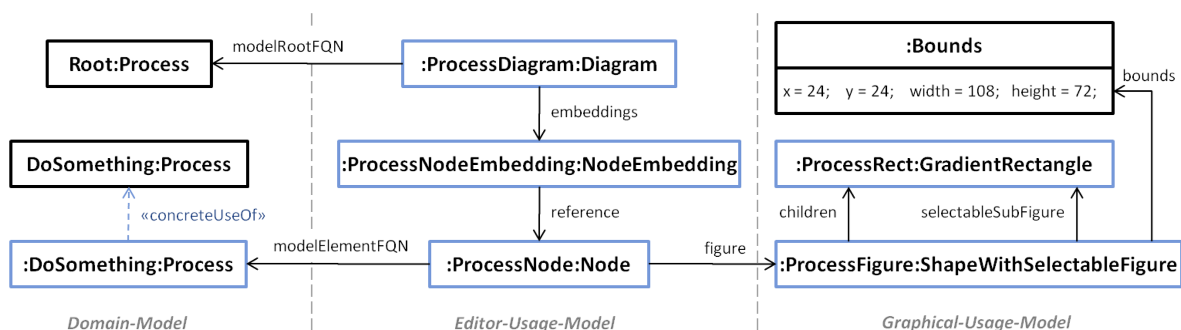

Abb. 7-5: Domain-, Editor-Usage- und Graphical-Usage-Model für Designer mit einer visualisierten Prozess-Verwendung

7.2.1 Anzeige des Prozessnamens

Die Visualisierung der Verwendung des Prozesses „Do Something" aus Abb. 7-4 ist nicht eindeutig, da nicht ersichtlich ist, zu welchem `Process`-Typ das blaue Rechteck gehört. Abhilfe kann hier die Anzeige des Prozessnamens auf der jeweiligen Prozess-`Figure` schaffen (Abb. 7-7). Dazu muss dem

ProcessRect aus Listing 7-2 eine Instanz des Label-Concepts hinzugefügt werden. Diese wird von der EditorLabel-Instanz ProcessLabel über das label-Attribut referenziert (Listing 7-4). Zusätzlich muss als displayAttributeQN das Attribut name des Process-*Concepts* angegeben werden. Hiermit wird der dem name-Attribut zugewiesene String vom Label des zugehörigen Prozessrechtecks angezeigt. Zum Schluss ist ProcessLabel dem subElements-Attribut von ProcessNode hinzuzufügen.

```
concept ProcessNode instanceOf Editor.D2.Core.Node {
     values {
          concept figure = ProcessGraphicalDefinition.D1.POPM.Process.ProcessFigure;
          fqn modelElementFQN = POPMMetaModel.M2.POPM.Process;
          fqn toolingAttributeQN = name;
          string toolingTitle = "Process";
          concept subElements = ProcessLabel;
     }
}

concept ProcessLabel instanceOf Editor.D2.Core.EditorLabel {
     values {
          concept owner = ProcessNode;
          fqn displayAttributeQN = name;
          concept label = ProcessGraphicalDefinition.D1.POPM.Process.Center.Upper.ProcessLabel;
     }
}
```

Listing 7-4: Definition des Prozess-Labels als Sub-Element von ProcessNode

Die Änderungen innerhalb von *Graphical-Usage-* und *Editor-Usage-Model* für eine visualisierte Prozess-Verwendung inkl. eines derartigen Labels fallen eher gering aus (Abb. 7-6). Auf der grafischen Seite wird lediglich die Figures-Hierarchie um die benötigte ProcessLabel-Verwendung erweitert, was automatisch beim Anlegen dieser Hierarchie erfolgt. Im *Editor-Usage-Model* geschieht dasselbe mit der gleichnamigen EditorLabel-Instanz-Verwendung, indem diese explizit dem subElements-Attribut der ProcessNode-Verwendung zugewiesen wird. Diese Vorgehensweise ist bei allen im *Editor-Definition-Model* spezifizierten SubElements identisch, d.h. es werden stets Verwendungen aller SubElements für die jeweilige Node-Instanz-Verwendung erzeugt.

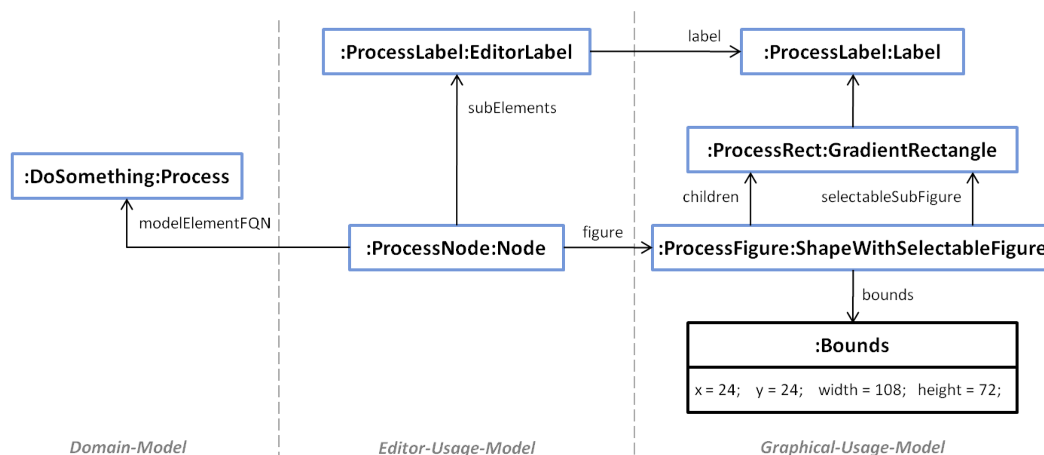

Abb. 7-6: Editor-Usage- und Graphical-Usage-Model mit EditorLabel- bzw. Label-Instanz zur Anzeige des Prozessnamens

7.2.2 Einsatzmöglichkeit des RewriterService

Um auf den ersten Blick erkennen zu können, ob sich ein Prozess in weitere Teilprozesse untergliedert, muss diese Eigenschaft auf irgendeine Weise innerhalb des Prozessrechtecks visualisiert werden. Eine Möglichkeit ist zum Beispiel, dass bei kompositen Prozessen in der rechten unteren Ecke ein kleines Quadrat eingeblendet wird. Zu diesem Zweck ist dem ProcessRect eine Rectangle-

Instanz fester Größe hinzuzufügen, deren Sichtbarkeit durch einen speziell konfigurierten `RewriterService` beeinflusst werden kann. Im als `CommonEditParts` bezeichneten *Extended Editor-Meta-Model*, das Bestandteil des MDFs ist, stehen für diesen Anwendungsfall sowohl eine spezialisierte `BooleanCondition` als auch ein spezialisierter `ParameterRewriter` zur Verfügung. Die `ContentBasedCondition` liefert für ein mittels `attributeName` spezifiziertes Attribut `true` zurück, falls diesem Attribut ein Wert zugewiesen ist; andernfalls erfolgt die Rückgabe von `false`. Der `VisibilityRewriter` ändert auf Basis eines booleschen Wertes die Sichtbarkeit der durch den zugehörigen `RewriterService` angegeben `Figure`. In Listing 7-5 ist die genaue Definition der beschriebenen `RewriterService`-Instanz abgedruckt.

```
concept ProcessNode instanceOf Editor.D2.Core.Node {
	values {
		concept figure = ProcessGraphicalDefinition.D1.POPM.Process.ProcessFigure;
		fqn modelElementFQN = POPMMetaModel.M2.POPM.Process;
		fqn toolingAttributeQN = name;
		string toolingTitle = "Process";
		concept subElements = ProcessLabel, ContentBasedRewriterService;
	}
}

concept ContentBasedVisibility instanceOf CommonEditParts.D2.Common.VisibilityRewriter { }

concept ContentBasedCondition instanceOf CommonEditParts.D2.Common.ContentBasedCondition {
	values {
		fqn attributeName = innerContent;
	}
}

concept ContentBasedRewriterService instanceOf Editor.D2.Core.RewriterService {
	values {
		concept owner = ProcessNode;
		concept graphElement =
			ProcessGraphicalDefinition.D1.POPM.Process.Center.Lower.CompositeMarker;
		concept rewriters = ContentBasedVisibility;
		concept condition = ContentBasedCondition;
	}
}
```

Listing 7-5: Definition eines `RewriterService` als Sub-Element von `ProcessNode`

Ein Beispiel ist in Abb. 7-7 dargestellt. Demzufolge ist der „Do Something“-Prozess atomar und der „Root“-Prozess komposit. Die Tatsache, dass der letztgenannte Prozess komposit ist, beruht darauf, dass er vom Diagramm repräsentiert wird. Der Inhalt des Prozesstyps „Root“ setzt sich somit aus den Prozessverwendungen „Do Something“ und „Root“ zusammen.

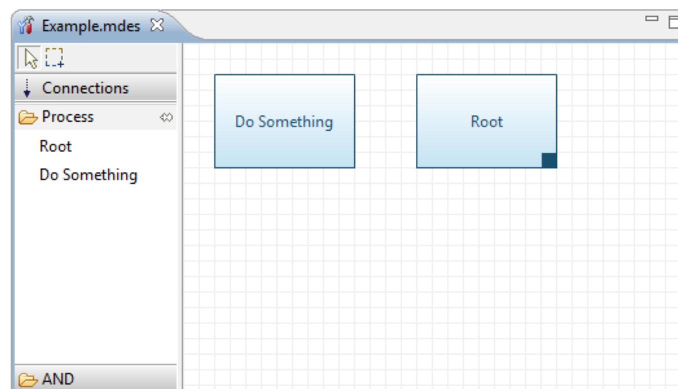

Abb. 7-7: Visualisierung der Verwendung von `EditorLabel` und `RewriterService` im Designer

7.2.3 Compartments für Eingangs- und Ausgangsdaten

Jeder Prozess kann über Eingangs- und Ausgangsdaten verfügen. Sie sollen in Form grauer Rechtecke am linken bzw. rechten unteren Rand der Prozess-`Figure` angehängt werden. Die Definition der entsprechenden Datum-`Figure` und des Datum-`Node` erfolgt analog zur Definition der beiden anderen `Figures` und `Nodes`. Eine Datum-`Figure` besitzt ein mittig ausgerichtetes `Label`, das über ein `EditorLabel`-Mapping den an das `typeName`-Attribut zugewiesenen Wert anzeigt. Aufgrund der gleichen Vorgehensweise bei der Spezifikation der beiden `Compartments` für Eingangs- und Ausgangsdaten wird im Folgenden exemplarisch nur der erste Fall betrachtet.

```
concept ProcessNode instanceOf Editor.D2.Core.Node {
     values {
          concept figure = ProcessGraphicalDefinition.D1.POPM.Process.ProcessFigure;
          fqn modelElementFQN = POPMMetaModel.M2.POPM.Process;
          fqn toolingAttributeQN = name;
          string toolingTitle = "Process";
          concept subElements = ProcessLabel, ContentBasedRewriterService, InputDataCompartment;
     }
}
concept InputDataCompartment instanceOf Editor.D2.Core.Compartment {
     values {
          concept owner = ProcessNode;
          concept contentPane = ProcessGraphicalDefinition.D1.POPM.Process.Bottom.LeftSection;
          concept embeddings = InputDataEmbedding;
     }
}
concept InputDataEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
     values {
          concept reference = ProcessEditorDefinition.D1.POPM.Nodes.DataItem.DataItemNode;
          fqn containmentQN = inputData;
     }
}
```

Listing 7-6: Definition des `Compartments` für Eingangsdaten als Sub-Element von `ProcessNode`

Grundlage ist das `InputDataCompartment`, das als `contentPane` diejenige Kind-`Figure` der Prozess-`Figure` angibt, die sich unsichtbar am linken unteren Rand befindet[3]. Es umfasst zudem das `InputDataEmbedding`, welches festlegt, dass `DataItemNodes` diesem `Compartment` hinzugefügt werden dürfen. Darüber hinaus spezifiziert das `InputDataEmbedding` über `containmentQN`, dass angehängte `DataItems` im `inputData`-Attribut des `Process`-*Concepts* abgelegt werden. Schließlich muss das `InputDataCompartment` dem `ProcessNode` mittels `subElements` bekannt gemacht werden. Die exakte Definition in konkreter LMM-Syntax ist Listing 7-6 zu entnehmen.

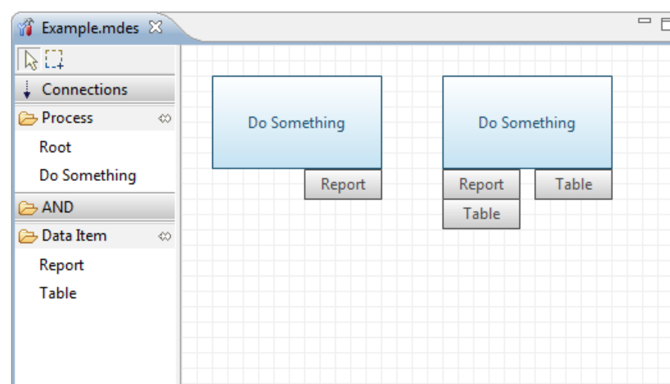

Abb. 7-8: Visualisierung zweier Prozessverwendungen mit angehängten Daten im Designer

[3] Nähere Details zur Spezifikation der kompletten Prozess-`Figure` Hierarchie sind dem *Graphical-Definition-Model* in Anhang A zu entnehmen. Dabei handelt es sich um Draw2D-spezifische Eigenheiten, die bei einer detaillierten Betrachtung den Rahmen dieser Arbeit sprengen würden.

Abb. 7-8 zeigt einen Screenshot des Prozess-Designers mit zwei Verwendungen des Prozesstyps „Do Something“. Die erste Verwendung besitzt lediglich „Report“ als Ausgangsdatum, wohingegen die andere Verwendung über zwei Eingangsdaten und ein Ausgangsdatum verfügt.

7.3 Definition von Kontroll- und Datenfluss-Verbindung

Bisher wurden Beziehungen zwischen Modellelementen ausschließlich auf Basis der Geometrie-basierten Klasse (Abschnitt 2.3) visualisiert. Einerseits existiert die Möglichkeit Prozess-Hierarchien aufzustellen und andererseits können jeder Prozess-Verwendung sowohl Ein- als auch Ausgangsdaten zugeordnet werden. In den beiden kommenden Abschnitten steht nun die Graph-basierte Variante im Mittelpunkt, die zur grafischen Darstellung von Kontroll- und Datenflüssen eingesetzt wird.

7.3.1 Kontrollfluss

Ein Kontrollfluss soll durch einen grauen Pfeil zwischen zwei `Figures`, die einen `Node` (im Sinne des Domänen-Modells) repräsentieren, ausgedrückt werden. Das Aussehen dieses Pfeils gilt es zunächst im *Graphical-Definition-Model* zu spezifizieren (Listing 7-7). Es ist darin als Instanz einer `Connection` definiert und besitzt als `targetDecoration` eine `PolygonDecoration`-Instanz mit dem Namen `FlowDecoration`. Letztere ist für das Zeichnen der ausgefüllten Pfeilspitze zuständig.

```
concept ControlFlow instanceOf Graphical.D2.Core.Connection {
     values {
          concept foregroundColor = ControlFlowColor;
          concept targetDecoration = FlowDecoration;
     }
}
concept ControlFlowColor instanceOf Graphical.D2.Core.Types.Color {
     values {
          integer red = 150; integer green = 150; integer blue = 150;
     }
}
concept FlowDecoration instanceOf CommonGraphics.D2.Common.Decoration.PolygonDecoration { }
```

Listing 7-7: Definition der Kontrollfluss-`Figure`

Als Verbindungs-Typ kommt die Attribut-basierte Verbindung zum Einsatz, da ein Kontrollfluss entsteht, sobald dem `successors`-Attribut eines `Nodes` ein anderer `Node` zugewiesen wird. Deshalb muss für das `ControlFlowLink`-*Concept* als `attributeQN` das `successors`-Attribut und als `modelElementFQN` der voll qualifizierte Name des `Node`-*Concepts* angegeben werden (Listing 7-8). Des Weiteren hält das `figure`-Attribut zur Anzeige des besagten grauen Pfeils eine Referenz auf die in Listing 7-7 spezifizierte `ControlFlowFigure`. Zuletzt muss dem `links`-Attribut des `ProcessDiagrams` ein Verweis auf den `ControlFlowLink` hinzugefügt werden.

Im Screenshot aus Abb. 7-9 sind vier Verwendungen des Prozesstyps „Do Something“ und ein AND-Konnektor durch mehrere Kontrollflüsse vernetzt. Jeweils einer dieser Flüsse gibt genau eine Nachfolger-Beziehung der verbundenen Knoten an, d.h. der AND-Konnektor ist beispielsweise ein direkter Nachfolger des im Designer links abgebildeten Prozesses.

```
concept ProcessDiagram instanceOf Editor.D2.Core.Diagram {
     values {
          enum paradigm = TYPE_USAGE;
          fqn modelRootFQN = POPMMetaModel.M2.POPM.Process;
          concept embeddings = ProcessNodeEmbedding, ANDNodeEmbedding;
          concept links = ProcessEditorDefinition.D1.POPM.Links.ControlFlowLink;
     }
}
```

```
package Links {
     concept ControlFlowLink instanceOf Editor.D2.Core.AttributeBasedLink {
          values {
               fqn modelElementFQN = POPMMetaModel.M2.POPM.Node;
               fqn attributeQN = successors;
               concept figure = ProcessGraphicalDefinition.D1.POPM.Flows.ControlFlow;
               string toolingTitle = "Control Flow";
          }
     }
}
```

Listing 7-8: Definition des Kontrollfluss-`Links`

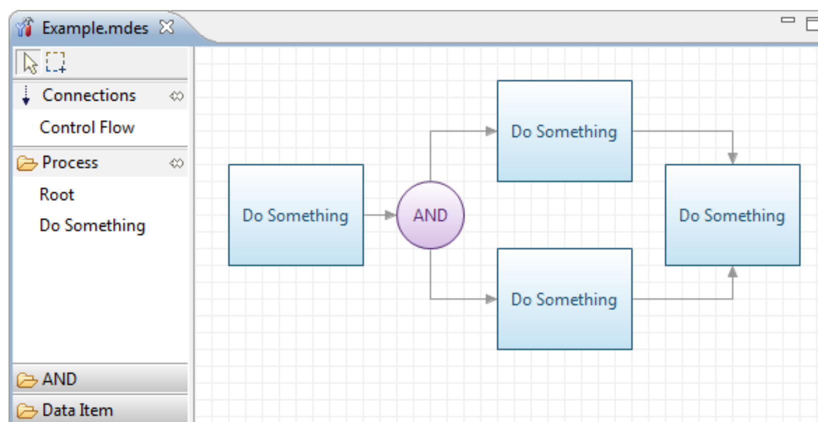

Abb. 7-9: Visualisierung mehrerer Kontrollflüsse im Designer zwischen vier Prozessen und einem AND-Konnektor

Abb. 7-10 veranschaulicht einen Ausschnitt von *Domain-*, *Graphical-Usage-* und *Editor-Usage-Model* zum in Abb. 7-9 visualisierten Prozessmodell. Der Ausschnitt beschränkt sich dabei auf den ersten Prozess, den AND-Konnektor und den dazwischen liegenden Kontrollfluss. Nachdem das Hauptaugenmerk auf der Repräsentation des Kontrollflusses liegt, ist nur die Wurzel der jeweiligen `Figure`-Hierarchie von Prozess und AND-Konnektor angegeben. Im *Editor-Usage-Model* wird der besagte Kontrollfluss durch eine konkrete Verwendung von `ControlFlowLink` ausgedrückt, der als `modelElementFQN` eine Beziehung zur ersten „Do Something"-Verwendung hält. Ferner muss die `ProcessNode`-Verwendung diese `Link`-Instanz im `outgoingLinks`-Attribut referenzieren und die `ANDNode`-Verwendung im `incomingLinks`-Attribut.

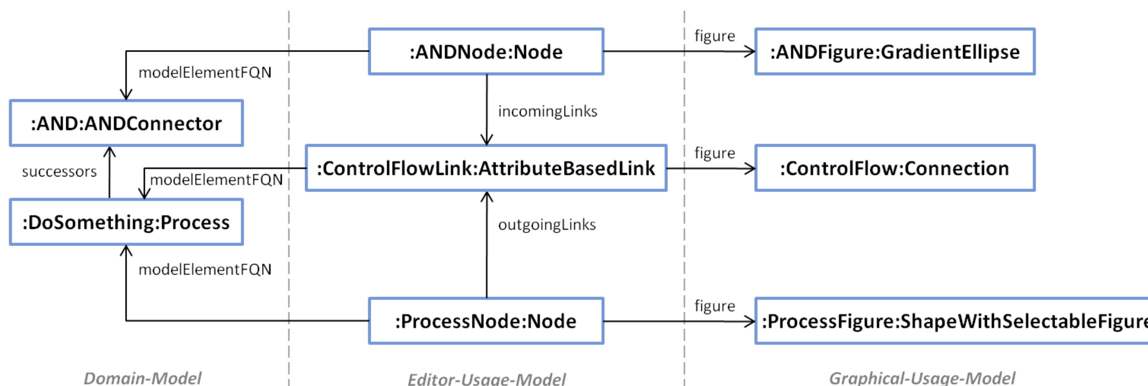

Abb. 7-10: Repräsentation eines Attribut-basierten Links in Domain-, Editor-Usage- und Graphical-Usage-Model

7.3.2 Datenfluss

Ein Datenfluss wird durch eine Instanz des `DataFlow`-*Concepts* repräsentiert und von zwei `DataItem`-Instanzen referenziert. Die Flussrichtung ist abhängig von den Attributen `outgoingDataFlows` und `incomingDataFlows` der beteiligten `DataItems`. Unter der Annahme,

dass zwei DataItems *A* und *B* sowie ein DataFlow *F* existieren, verläuft der Datenfluss von *A* nach *B*, falls in *A* das outgoingDataFlows-Attribut und in *B* das incomingDataFlows-Attribut mit *F* belegt ist.

Die Datenfluss-Figure wird analog zur Kontrollfluss-Figure definiert, da sich beide nur anhand der Farbe Schwarz unterscheiden sollen. Nachdem ein Datenfluss durch eine konkrete DataFlow-Instanz ausgedrückt wird, muss die Verbindung Entität-basiert sein. Als modelElementFQN ist für den entsprechenden DataFlowLink der voll qualifizierte Name des DataFlow-*Concepts* anzugeben (Listing 7-9). sourceFQN referenziert das outgoingDataFlows-Attribut, während targetFQN auf das incomingDataFlows-Attribut zeigt. Bei beiden Angaben muss es sich um voll qualifizierte Namen der verwendeten Attribute handeln, da auf diese Weise indirekt die Typen (Elemente des Domänen-Meta-Modells) von Start und Ziel vorgegeben sind. In unserem Beispiel sind beide Typen identisch, weil sowohl incomingDataFlows als auch outgoingDataFlows innerhalb von DataItem deklariert werden und daher dieses *Concept* den gemeinsamen Typ darstellt.

```
concept ProcessDiagram instanceOf Editor.D2.Core.Diagram {
     values {
          enum paradigm = TYPE_USAGE;
          fqn modelRootFQN = POPMMetaModel.M2.POPM.Process;
          concept embeddings = ProcessNodeEmbedding, ANDNodeEmbedding;
          concept links = ProcessEditorDefinition.D1.POPM.Links.ControlFlowLink,
               ProcessEditorDefinition.D1.POPM.Links.DataFlowLink;
     }
}

package Links {
     concept DataFlowLink instanceOf Editor.D2.Core.EntityBasedLink {
          values {
               fqn modelElementFQN = POPMMetaModel.M2.POPM.DataFlow;
               fqn sourceFQN = POPMMetaModel.M2.POPM.DataItem.outgoingDataFlows;
               fqn targetFQN = POPMMetaModel.M2.POPM.DataItem.incomingDataFlows;
               concept figure = ProcessGraphicalDefinition.D1.POPM.Flows.DataFlow;
               string toolingTitle = "Data Flow";
          }
     }
}
```

Listing 7-9: Definition des Datenfluss-Links

Der in Abb. 7-11 wiedergegebene Screenshot zeigt zwei durch schwarze Pfeile symbolisierte Datenflüsse, von denen der obere mit einem Label versehen ist. Für dieses Label ist ein Mapping definiert, das den Inhalt des trafoDefinition-Attributs der gekapselten DataFlow-Instanz visualisiert. Ein derartiges EditorLabel ist nur für Entität-basierte Verbindungen sinnvoll, da ausschließlich hier eine Verbindung durch ein separates Modellelement repräsentiert wird und dieses Element im Regelfall einen internen Zustand besitzt.

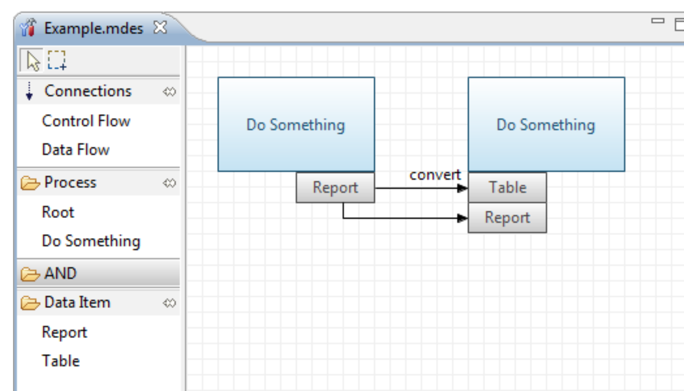

Abb. 7-11: Visualisierung von zwei Datenflüssen im Designer

7.4 Allgemeingültigkeit des Anwendungsfalls

Der auf den vorangegangenen Seiten präsentierte Anwendungsfall zur Definition eines Prozess-Designers wurde gewählt, weil er alle Mapping-Anforderungen (d) bis (g) aus Abschnitt 1.1 anwendet und zugleich das Typ-Verwendungs-Konzept zum Einsatz kommt.

- **Zu (d):** Durch die Definition des Prozess-Knoten in Abschnitt 7.2 ist die Abbildung von Modellelementen auf grafische Objekte gegeben.
- **Zu (e):** Die Geometrie-basierte Darstellung von Beziehungen zwischen Modellelementen wird zum einen durch die Definition des Prozess-Diagramms (Abschnitt 7.1) und zum anderen durch die Verwendung von `Compartments` für Eingangs- und Ausgangsdaten (Abschnitt 7.2.3) abgedeckt. Wird statt dem vereinfachten POPM-Modell die Original-Version von Volz [6] benutzt, so können analog zu diesen `Compartments` entsprechende `Compartments` für die Elemente der organisatorischen und operationalen Perspektive spezifiziert werden. Ein Beispiel, wie ein auf diese Weise erweiterter Prozess aussehen kann, ist im Anhang B abgebildet.
- **Zu (f):** In Abschnitt 7.3 sind die beiden Varianten der Graph-basierten Visualisierung von Beziehungen zwischen Modellelementen exemplarisch dargelegt. Dabei kommt für die Repräsentation von Kontrollflüssen die Attribut-basierte Variante und für die Darstellung von Datenflüssen die Entität-basierte Variante zum Einsatz.
- **Zu (g):** Die Visualisierung des Zustands eines Modellelements innerhalb der zugehörigen grafischen `Figure` erfolgt einerseits durch die Anzeige des Prozessnamens (Abschnitt 7.2.1). Andererseits wird in Abschnitt 7.2.2 der `RewriterService` dazu verwendet, um bei jedem Prozess darstellen zu können, ob dieser komposit oder atomar ist. Dies geschieht im kompositen Fall durch die Ausgabe eines kleinen ausgefüllten Rechtecks am rechten unteren Rand der jeweiligen Prozess-`Figure`.

Sämtliche Mappings können auch zur Spezifikation eines Designers verwendet werden, der auf der klassischen Modellierungsweise (Instanz-Erzeugung) beruht. Als Beispiel sei an der Stelle ein Editor genannt, der zur Erstellung von Entity-Relationship-Diagrammen [79] dient. Hierbei können *Entities*, *Attributes* und *Relationships* durch Knoten repräsentiert werden, die jeweils ein `Label` zur Anzeige des Namens besitzen. Zur Visualisierung der Zuweisung von *Attributes* an *Entities* bzw. *Relationships* bietet sich die Definition einer Verbindung an, die der Attribut-basierte Variante folgt. Beziehungen zwischen *Relationships* und *Entities* werden durch separate *Concepts*, die ein Attribut zur Angabe der Multiplizität beinhalten, ausgedrückt, weshalb die Entität-basierte Variante zur Definition von Verbindungen einzusetzen ist. Darüber hinaus ist es für den späteren Benutzer des ER-Designers hilfreich, wenn diese Verbindung über ein `Label` verfügt, das die Multiplizität anzeigt. Im Anhang B ist für einen derartigen Designer ebenfalls ein Beispiel abgebildet.

Kapitel 8 Implementierung: Model Designer Framework

Das im Rahmen dieser Arbeit implementierte und auf GEF (Abschnitt 3.3.2) aufsetzende Model Designer Framework dient zur Modell-getriebenen Entwicklung grafischer Editoren. Als Teil des Open Meta-Modeling Environments (Abschnitt 2.2) werden die zugrunde liegenden Modelle mit Hilfe der konkreten Syntax des LMM formuliert. Der Aufbau und die Spezifikation der zugehörigen Meta-Modelle ist bereits in Kapitel 5 und Kapitel 6 abgehandelt worden. Deshalb wird in diesem Kapitel erläutert, wie die Verarbeitung der verschiedenen Modelle und ihrer Elemente erfolgt. Dazu werden die Implementierungs-Architektur vorgestellt und dabei zentrale Bestandteile und Abläufe detailliert beschrieben. Zum Schluss werden eine Übersicht über die momentan implementierten Funktionen und Fähigkeiten sowie ein Ausblick auf nutzbringende Features zur Erweiterung des MDF gegeben.

8.1 Architektur

Die MDF-Architektur lässt sich – wie in Abb. 8-1 dargestellt – grobgranular in mehrere Module aufteilen. Ausgangspunkt ist das von oMME bereitgestellte *LMM*-Modul, das einen Low-Level Zugriff auf die einzelnen Modelle und Modellelemente bietet.

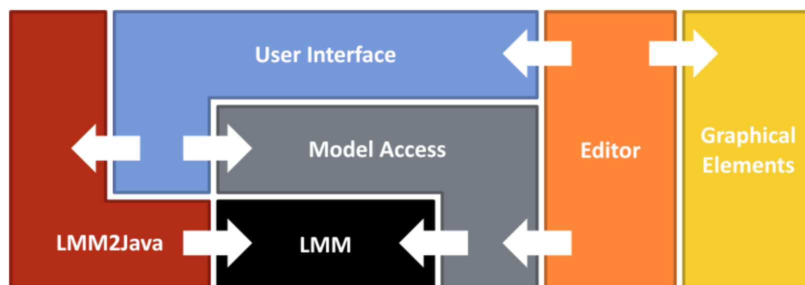

Abb. 8-1: Übersicht über die Architektur des Model Designer Frameworks

Das *LMM2Java*-Modul wurde zwar im Kontext von MDF entwickelt, kann aber aufgrund der einzig vorhandenen Abhängigkeit zu *LMM* problemlos auch in anderen oMME-Teilprojekten eingesetzt werden. Es ist für die Abbildung von *Concepts* auf entsprechende Java-Objekte verantwortlich und benötigt hierfür das in Abschnitt 6.1 spezifizierte *Java-Bean-Mapping-Model*.

Die eigentliche Basis des Model Designer Frameworks bildet das Modul *Model Access*, da es den Zugriff auf sämtliche Elemente der verschiedenen Modelle kapselt und dadurch eine High-Level API zur Verfügung stellt. Über diese API interagieren *User Interface* und *Editor* mit den Modellelementen. Zudem lassen sich darüber den einzelnen Modellen neue *Concepts* hinzufügen und bereits existierende löschen.

Im *User Interface* Modul befinden sich die Basiskomponenten der Benutzerschnittstelle, die jeder Designer benötigt. Das sind im Wesentlichen die Zeichenfläche mit zugehöriger Tool-Palette und eine Eingabemaske zur Bearbeitung von Eigenschaften des momentan selektierten `EditorElements`. Insbesondere für die Möglichkeit des Bearbeitens ist der Zugriff auf Modellelemente über *Model Access* erforderlich.

Das *Editor* Modul umfasst diejenigen Komponenten, die den Controller-Teil von GEF repräsentieren. Dabei handelt es sich in erster Linie um Klassen, für die ein korrespondierendes *Concept* im *Editor*-

Meta-Model existiert. Im *Graphical Elements* Modul hingegen sind die den Elementen des *Graphical-Meta-Models* entsprechenden Klassen enthalten. Der Großteil der gemappten Klassen stammt hierbei direkt von Draw2D.

Die vorangegangenen Ausführungen beschreiben den Aufbau des MDF nur sehr grob. Damit jedoch eigene Erweiterungen für das Framework entwickelt und anschließend integriert werden können, sind tiefergehende Kenntnisse über die Struktur und das Zusammenspiel der einzelnen Module nötig. Zu diesem Zweck zeigt Abb. 8-2 ein Klassendiagramm, dessen Inhalt sich auf die zentralen MDF-Elemente beschränkt. Die Farben der verschiedenen Pakete, Klassen und Schnittstellen geben die Zugehörigkeit zu den in Abb. 8-1 abgedruckten Modulen an.

8.1.1 Model Access Modul

Das *Model Access* Modul (graue Elemente) lässt sich weiter in die Klasse *ModelContext* sowie die Pakete *ModelWrappers, Persistence* und *ConceptCreation* unterteilen. *ModelContext* bildet die Anlaufstelle für den Zugriff auf sämtliche Modelle, die einen bestimmten Designer inkl. des visualisierten Inhalts spezifizieren. Dieser Sachverhalt ist durch die zahlreichen eingehenden Referenzen ersichtlich (Abb. 8-2). Der Zugriff auf die gekapselten Modelle erfolgt ausschließlich über die im *ModelWrappers*-Paket enthaltenen Wrapper-Klassen. Sie bieten spezielle, auf den jeweiligen Modelltyp zugeschnittene Methoden. Beispielsweise stellt die Klasse *EditorUsageModel* eine Methode bereit, die beim Aufruf eine Verwendung einer im *Editor-Definition-Model* definierten `Node`-Instanz erzeugt. Des Weiteren können beim *ModelContext* spezielle *ModelListener* registriert werden, die bei Änderungen von bestimmten Modellelementen benachrichtigt werden. Auf welche Änderungsereignisse reagiert werden soll, muss die jeweilige *ModelListener*-Implementierung festlegen.

Das *Persistence*-Paket dient zum Einlesen und Abspeichern konkreter LMM-Modelle. Mit Hilfe der abstrakten Klasse *Storage* können einerseits ein *Editor-Definition-Model* geladen und darauf basierend zugehörige modifizierbare[4] Modelle erzeugt werden. Andererseits lässt sich ein bereits vorhandenes *Editor-Usage-Models* einlesen, das zu einem späteren Zeitpunkt zusammen mit den beiden anderen modifizierbaren Modellen abgespeichert werden kann. Beim Erstellen neuer modifizierbarer Modelle werden benötigte Informationen (z.B. Modellname, Model-URI) mittels Callback-Mechanismus – realisiert durch die Klasse *AbstractCallback* – abgefragt. Sowohl beim Laden eines *Editor-Definition-Models* als auch beim Öffnen eines *Editor-Usage-Models* wird eine neue *ModelContext*-Instanz angelegt. Gleichzeitig kümmern sich entsprechende *ModelRecognizer* um die korrekte Einordnung des eingelesen Modells sowie aller referenzierten Modelle in den neu instanziierten *ModelContext*. Die eigentlichen Lade- und Speicher-Routinen sind von Klassen, die Storage erweitern, zu implementieren. So bleibt die Art der Persistierung flexibel, d.h. Modelle können z.B. in Dateiform oder unter Verwendung einer Datenbank abgelegt werden.

Im *ConceptCreation*-Paket befinden sich Klassen, die das Erzeugen neuer Instanzen im Domänen-Modell regeln. Alle diesbezüglichen Modellierungsansätze sind in Abschnitt 5.1 erläutert. Den abstrakten allgemeinen Ansatz stellt die Klasse *CreationParadigm* dar, die von *InstanceCreator* und *TypeUsageCreator* konkretisiert wird. Abhängig vom verwendeten Modellierungsansatz unterscheidet sich der Aufbau der Tool-Palette, weshalb jedes konkrete *CreationParadigm* die genaue Paletten-Struktur festlegen muss.

[4] Modifizierbar sind das *Domain-*, das *Graphical-Usage-* und das *Editor-Usage-Model*, da sie den Zustand eines Designers beschreiben und dieser Zustand verändert werden darf.

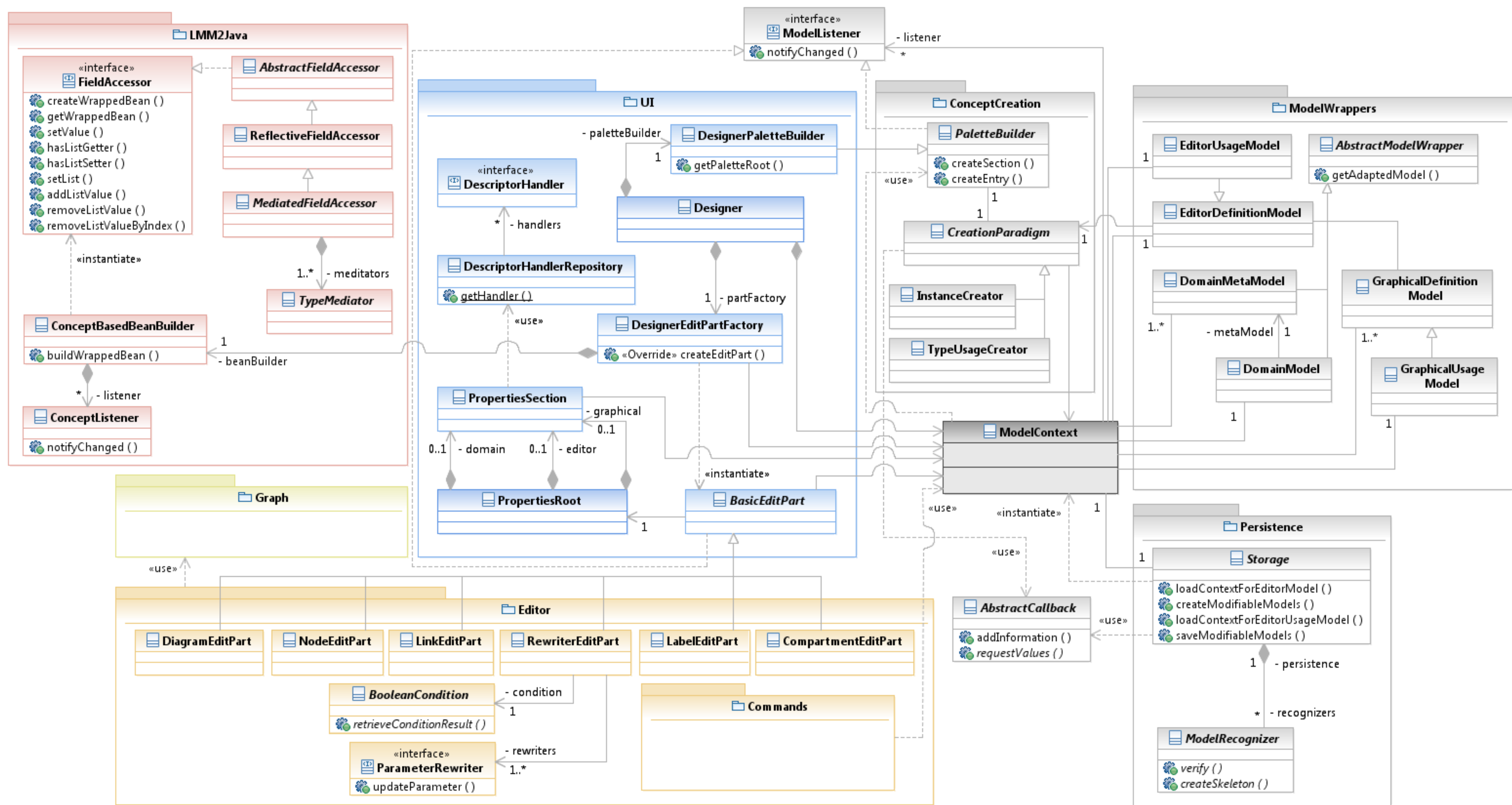

Abb. 8-2: Auf die Kernbestandteile reduziertes Klassendiagram des Model Designer Frameworks

8.1.2 LMM2Java Modul

Zur generischen Erweiterbarkeit des Model Designer Frameworks trägt in erster Linie das *LMM2Java*-Paket bei, da die bereitgestellte Funktionalität *Concepts* auf beliebige Java-Objekte abbilden kann und diese innerhalb eines Designers einsetzbar sind. Voraussetzung dafür ist, dass ein *Concept* die in Abschnitt 6.1 spezifizierte Bedingung erfüllt und somit im Rahmen von *LMM2Java* valide ist. Sie gibt vor, dass der Typ des abzubildenden *Concepts* eine Instanz des `JavaBeanMapping`-*Concepts* sein muss.

Die Erzeugung des zu einem validen *Concepts* gehörenden Java-Objekts übernimmt eine Instanz der Klasse *ConceptBasedBeanBuilder*. Für jedes verarbeitete *Concept* registriert sie einen entsprechenden *ConceptListener*, der über Änderungen innerhalb der Instanz-Facette informiert wird. Bei einer Wertzuweisung an ein Attribut dieses *Concepts* wird der zugewiesene Wert automatisch an das korrespondierende Feld des zugehörigen Objekts delegiert. Der Vorteil dieser Synchronisation ist, dass sich die Implementierung der Komponenten des *Editor*-Pakets stark vereinfacht, da Änderungen an *Concepts*, die einem Draw2D-Objekt entsprechen, direkt von diesem Objekt übernommen werden. Zugleich wird die bislang noch nicht berücksichtigte Anforderung (i) aus Abschnitt 1.1 erfüllt, da ein Neustart von Eclipse durch die direkte Weiterleitung von Modifikationen überflüssig ist.

8.1.2.1 Das FieldAccessor-Interface

Zur Vermittlung zwischen LMM- und Java-Welt muss ein einheitlicher Feld-Zugriff auf sämtliche Objekte gewährleistet sein. Dies wird durch die Kapselung jedes Java-Objekts durch einen *FieldAccessor* erreicht. Standardmäßig kommt der *ReflectiveFieldAccessor* zum Einsatz, der mittels Java Reflection API [80] auf die einzelnen Felder zugreift. Unterstützt werden dabei sowohl private Felder, die über entsprechende Setter-Methoden verfügen, als auch öffentliche Felder. Die Möglichkeit zum Zugriff auf den letztgenannten Feld-Typ existiert aus pragmatischen Gründen: Einige von Draw2D zur Verfügung gestellte Klassen besitzen ausschließlich öffentliche Felder ohne zugehörige Setter-Methoden.

Nachdem das *FieldAccessor*-Interface eine zentrale Rolle einnimmt, wird dieses nun näher vorgestellt. Jede Methode – mit Ausnahme von *createWrappedBean*() und *getWrappedBean*() – erwartet als erstes Argument den Namen des betreffenden Feldes, auf das zugegriffen werden soll. Da das *Concept*-Objekt-Mapping unidirektional ist, wird keine Methode für das Auslesen zugewiesener Werte angeboten.

Der Standardfall – also das Setzen eines Wertes auf Basis einer Zuweisung an ein einwertiges Attribut – wird von der *setValue*()-Methode abgedeckt. Der Wert (ggf. auch ein bereits gemapptes Java-Objekt) ist als zweites Argument zu übergeben.

Die Abbildung mehrwertiger Attribute auf entsprechende Felder (später auch als Liste bezeichnet) ist etwas komplexer, weil es drei Fälle zu unterscheiden gilt. Der einfachste Fall tritt auf, wenn zu einer Liste eine zugehörige Getter-Methode existiert und das Modifizieren der Liste erlaubt ist. Dann können alle *add*- und *remove*-Methoden direkt auf dieser Liste arbeiten. Ist eine Modifikation nicht gestattet, jedoch zusätzlich eine Setter-Methode vorhanden, so ist eine Kopie der Liste anzulegen, diese entsprechend zu verändern und die modifizierte Liste der Setter-Methode zu übergeben. Der letzte Fall ist durch eine Nicht-Verfügbarkeit von Getter und Setter gegeben, wodurch eine spezifische Implementierung für den Listenzugriff (*add*- und *remove*-Methoden) erforderlich wird. Als Beispiel kann die Draw2D-Klasse *Figure* genannt werden, da zum Hinzufügen und Entfernen von Kind-Elementen separate *add*- bzw. *remove*-Methoden aufzurufen sind.

8.1.2.2 Problemfall: Klassen als stationärer Verbund von Feldern

Draw2D und das zugrunde liegende SWT kommen mit zahlreichen Klassen, in denen das Setzen der deklarierten Felder ausschließlich im Konstruktor erfolgt. Im Nachhinein können die zugewiesenen Werte lediglich über entsprechende Getter ausgelesen, nicht aber ersetzt werden. Demnach handelt es sich bei derartigen Klassen um einen stationären Verbund von Feldern, deren Werte nach einmaliger Festlegung nicht mehr geändert werden dürfen. Ein Beispiel stellt die *Color*-Klasse mit den Feldern *red*, *green* und *blue* dar, auf die das gleichnamige *Concept* des *Graphical-Meta-Model* (Abschnitt 6.2) abgebildet ist. Liegen nun zwei miteinander verknüpfte Instanzen von *Color*-Klasse und `Color`-*Concept* vor, so lässt sich eine Wertzuweisung an das *blue*-Attribut nicht ohne Weiteres auf das korrespondierende Feld des Java-Objekts übertragen.

Die Problematik kann mit Hilfe einer zusätzlichen Vermittler-Klasse (*TypeMediator*), die eine auf die beschriebene Weise eingeschränkte Klasse vertritt, gelöst werden. Dazu muss der *TypeMediator* die fehlenden Setter-Methoden bereitstellen und sich darin um die benötigte Neu-Instanziierung der vermittelten Klasse kümmern. Dies impliziert, dass jede *TypeMediator*-Instanz alle Felder kennen muss, denen das vermittelte Objekt zugewiesen ist. Nur dadurch ist es möglich, dass ein neu erzeugtes Objekt im Änderungsfall an die relevanten Java-Objekte korrekt delegiert wird. Angewendet auf das obige Beispiel muss in der zum *blue*-Feld gehörenden Setter-Methode eine neue Instanz der *Color*-Klasse erzeugt werden. Dabei sind an den Konstruktor der bisherige Rot- und Grün- sowie der geänderte Blau-Wert zu übergeben. Das neu angelegte *Color*-Objekt muss schließlich allen Feldern zugewiesen werden, die das alte *Color*-Objekt referenzieren.

Damit jede *TypeMediator*-Instanz alle Java-Objekte kennt, die einen Verweis auf das gekapselte Objekt halten, wird ein spezieller *FieldAccessor* benötigt. Er muss wissen, welche Felder durch einen *TypeMediator* vertreten werden und beim Zuweisen einer entsprechenden Instanz diese über die neue hinzugekommene Referenz informieren. Umgekehrt muss eine *TypeMediator*-Instanz immer benachrichtigt werden, sobald eine Referenz entfernt wird. Diese Funktionalität bietet der abstrakte *MediatedFieldAccessor*. In einer konkreten Erweiterung dieser Klasse ist lediglich zu spezifizieren, welche Felder einen *TypeMediator* verwenden sollen. Als Beispiel einer derartigen Erweiterung kann der *FigureFieldAccessor* genannt werden, der den Zugriff auf die Felder der Klasse *Figure* managed und hierbei u.a. für die beiden Felder *backgroundColor* und *foregroundColor* angibt, dass ein *TypeMediator* zum Einsatz kommen soll.

8.1.3 User Interface Modul

Die zentrale Komponente im *UserInterface*-Paket ist die *Designer*-Klasse. Als Haupt-Schnittstelle zum Anwender ist sie für die Anzeige des Diagramms und zur Entgegennahme von Benutzerbefehlen zuständig. Des Weiteren fällt in ihren Aufgabenbereich die Bereitstellung der Tool-Palette, deren Inhalt durch den *DesignerPaletteBuilder* vorgegeben wird. Bei Verwendung der jeweiligen Tools werden neue Modellelemente inkl. der zugehörigen Elemente des *Graphical-Usage-* und des *Editor-Usage-Models* erzeugt. Auf deren Basis und unter Zuhilfenahme des *ConceptBasedBeanBuilders* instanziiert daraufhin die *DesignerEditPartFactory* die korrespondierenden GEF- und Draw2D-Klassen.

Jeder im MDF verwendete *EditPart* muss von der Klasse *BasicEditPart* abgeleitet sein, da sie eine bestimmte Grundfunktionalität zur Verfügung stellt. Hierbei handelt es sich insbesondere um das Anbieten konfigurierbarer Eigenschaften für die Standard Properties-View von Eclipse.

Struktur der Properties

Ein im Designer visualisierter Knoten wird immer durch *Concepts* aus bis zu drei verschiedenen Modellen repräsentiert. Dabei handelt es sich um das *Domain-*, das *Graphical-Usage-* und das *Editor-Usage-Model*. Für jeden Modelltyp steht eine separate *PropertiesSection* zur Verfügung, die bei Bedarf angezeigt wird. Der Inhalt der jeweiligen Sektion wird durch die im korrespondierenden *Concept* zuweisbaren Attribute festgelegt. In Abb. 8-3 ist exemplarisch ein Screenshot der Properties-View dargestellt, die zu einer im Designer selektierten Prozess-Verwendung sowohl die Eigenschaften des zugehörigen Domänen-Modellelements als auch die Eigenschaften der entsprechenden grafischen Erscheinung beinhaltet.

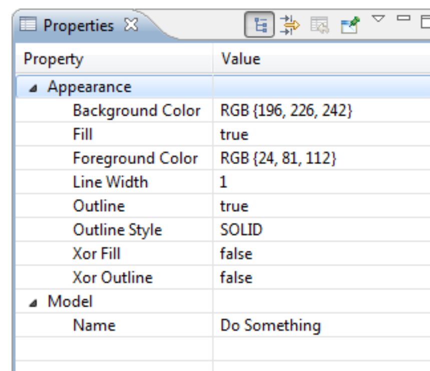

Abb. 8-3: Anzeige der Eigenschaften einer selektierten Prozess-Verwendung in der Properties-View von Eclipse

Standardmäßig werden nur *Enumeration-* und literale Attribute durch die Properties-View abgedeckt. Allerdings besteht die Möglichkeit für *Concepts* mit Typ-Facette spezielle *DescriptorHandler* zu implementieren und diese im *DescriptorHandlerRepository* verfügbar zu machen. Jeder *DescriptorHandler* muss einen separaten Dialog bieten, der als Ergebnis eine Instanz des zugeordneten *Concepts* zurückliefert. Im Idealfall kann der Dialog eine derartige, beim Öffnen übergebene Instanz auswerten und die enthaltenen Informationen entsprechend anzeigen. Demnach listet die Properties-View auch *Concept*-Attribute auf, zu deren Typ ein *DescriptorHandler* im *DescriptorHandlerRepository* abgelegt ist. Ein Beispiel dafür ist das `Color`-Concept. Bei der Bearbeitung von Instanzen dieses Typs wird ein Farbwahldialog angezeigt.

8.1.4 Editor und Graphical Elements Modul

Die Klassen des *Graph-* bzw. *Editor*-Pakets sind die Gegenstücke zu den *Concepts* des *Graphical-Meta-* bzw. des *Editor-Meta-Models* und repräsentieren den vom MDF vorgegebenen Inhalt der Module *Editor* und *Graphical Elements* (Abb. 8-1). Der überwiegende Teil der *Graphical Elements* wird durch Draw2D bestimmt, wohingegen die Klassen des *Editor*-Moduls spezifisch für das Model Designer Framework sind und auf elementarer GEF Funktionalität beruhen. Beide Module können jedoch um benutzerspezifische Klassen ergänzt werden, zu denen entsprechende *Concepts* in separaten *Extended Meta-Models* definiert werden müssen. Zusammen mit dem durch das *LMM2Java* Modul gegebenen Erweiterungsmechanismus kann auf diesem Weg der Großteil der Systemkomponenten ausgetauscht und dadurch auf neuartige Anforderungen flexibel reagiert werden. Das MDF umfasst bereits zwei derartige Modelle, und zwar sowohl auf der grafischen Seite (`CommonGraphics`) als auch auf der Seite der Editor-Elemente (`CommonEditParts`).

Ein Beispiel für eine Erweiterung stellt das *Concept* `GradientRectangle` dar, das im Modell `CommonGraphics` definiert ist. Es erweitert das `Shape`-*Concept* des *Graphical-Meta-Models* und spezifiziert ein Rechteck mit vertikalem Farbverlauf von weiß zur jeweils angegebenen Hintergrund-

farbe. Analog dazu kann auf der Editor-Seite beispielsweise eine Erweiterung des Compartment-*Concepts* definiert werden, die eine freie Platzierung (d.h. beliebige X- und Y-Koordinaten anstelle einer Index-basierten Positionierung) innenliegender Knoten erlaubt.

8.2 Benutzerinteraktion mit dem Designer

Jeder mit dem MDF spezifizierte Designer bietet dem Benutzer eine Reihe von Interaktionsmöglichkeiten, aus denen Änderungen am *Domain-*, *Graphical-Usage-* und/oder *Editor-Usage-Model* resultieren. Die konkreten Modelländerungen wurden bereits exemplarisch in Kapitel 7 erläutert. Ausgeführt werden sie durch spezielle *Commands*, die im gleichnamigen Paket innerhalb des *Editor*-Moduls liegen. In den folgenden Abschnitten werden die wichtigsten *Commands* vorgestellt, wobei lediglich der grobe Ablauf betrachtet wird.

8.2.1 Erstellen und Löschen eines Knotens

Zum Erstellen eines neuen Knotens muss ein Tool in der Tool-Palette ausgewählt werden, dem eine Node-Instanz zugrunde liegt. Im Falle des Typ-Verwendungs-Konzepts ist zuvor der gewünschte Typ auf der Seite des Domänen-Modells zu erzeugen, woraufhin die Palette entsprechend ergänzt wird. Dies funktioniert über das Kontextmenü der Tool-Palette, in welchem für jede Knotenart ein Eintrag existiert. Daraufhin kann eine Verwendung des ausgewählten Nodes einem beliebigen Container hinzugefügt werden, unter der Voraussetzung, dass ein passendes NodeEmbedding vorhanden ist und die durch das Domänen-Meta-Modell vorgegebenen Rahmenbedingungen eingehalten werden. Das Hinzufügen geschieht über einen einfachen Klick auf den jeweiligen Container. Alternativ kann beim Hinzufügen eines Knotens zum Diagram die Größe der resultierenden Figure durch Aufspannen eines rechteckigen Bereichs mit der Maus vorgegeben werden.

Abb. 8-4: Erforderliche Benutzerinteraktion zum Anlegen eines Knotens

Auf diese beiden ersten, vom Benutzer durchgeführten Schritte aus Abb. 8-4 folgt eine Reihe von Änderungen in *Domain-*, *Graphical-Usage-*, und *Editor-Usage-Model* (blau unterlegt), die in Form eines Aktivitätsdiagramms in Abb. 8-5 zusammengefasst sind. Dabei wird zunächst das zum ausgewählten Node gehörende Modellelement erzeugt und gleichzeitig die Figure-Hierarchie durch Verwendungen der ursprünglichen Hierarchie rekonstruiert. Die Verwendung der Wurzel-Figure wird mit einer Bounds-Instanz versehen, die entweder dem zuvor mit der Maus gezeichneten rechteckigen Bereich entspricht oder eine Standard-Abmessung vorgibt. Figure-Hierarchie und Modellelement werden an die zu erzeugende Node-Verwendung delegiert und dieser zugewiesen. Zudem wird zu jedem SubElement eine korrespondierende Verwendung angelegt, die eine Referenz auf ein bestimmtes Element der Figure-Hierarchie hält. Abschließend muss eine zur Node-Verwendung passende NodeEmbedding-Verwendung erzeugt werden, die dem ausgewählten Container hinzuzufügen ist. Damit kann das anfangs angelegte Modellelement dem durch NodeEmbedding spezifizierten Attribut zugewiesen werden. Die Zuweisung erfolgt innerhalb des *Concepts* des Domänen-Modells, das vom Container gekapselt wird.

Das Löschen eines bereits vorhandenen Knoten ist im Wesentlichen die inverse Operation zur Erstellung eines neuen Knoten. Hierfür muss der Benutzer den gewünschten Knoten selektieren und anschließend den *Delete*-Button betätigen. Bei der Ausführung des dadurch initiierten *Commands* werden sämtliche zur Node-Verwendung gehörenden *Concepts* aus den jeweiligen Modellen ent-

fernt. Gleichzeitig werden bei Verwendung der *Low-Level LMM API* (Abschnitt 2.2) rekursiv alle *Concepts* gelöscht, die beim Entfernen eines *Concepts* diesem über *composite* Attribute zugewiesen sind. Des Weiteren sind alle Referenzen, die von anderen *Concepts* auf die im vorherigen Schritt entfernten *Concepts* gehalten werden, ebenfalls zu löschen.

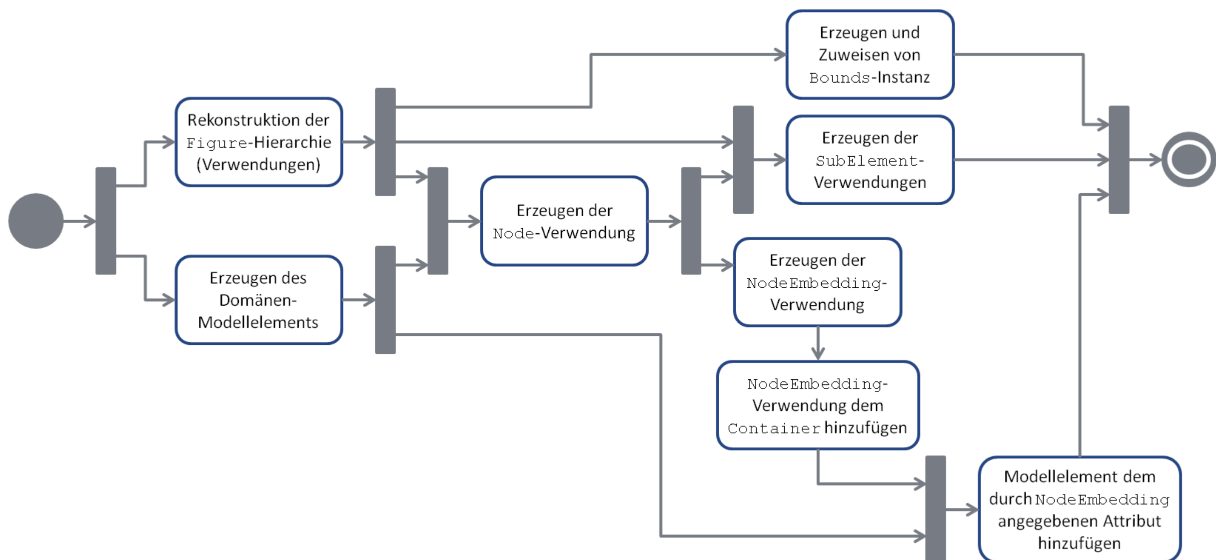

Abb. 8-5: Aktivitätsdiagramm der einzelnen Schritte beim Erzeugen einer `Node`-Verwendung

8.2.2 Ändern von Größe und Position eines Knotens

Befindet sich ein Knoten, dessen Größe und Position geändert werden soll, in einem `Container` mit der Möglichkeit zur freien Platzierung, so entspricht diese Änderungsoperation dem Anpassen der zur gekapselten `Figure` zugewiesenen `Bounds`-Instanz. Daher führt die Operation lediglich zu Modifikation im *Graphical-Usage-Model*. Die Standard-Implementierung von `Diagram` ist ein derartiger `Container`.

Eine andere Möglichkeit zur Positionsänderung bietet die Standard-Implementierung von `Compartment`. Sie ordnet die Knoten nach der Reihenfolge an, in der die zugehörigen `NodeEmbeddings` zugewiesen sind. Diese Reihenfolge ist identisch mit der Zuweisungsreihenfolge der entsprechenden Modellelemente, weshalb beide beim Verschieben eines Knotens aktualisiert werden müssen. Folglich sind Modifikationen nur innerhalb von *Domain-* und *Editor-Usage-Model* nötig.

8.2.3 Erstellen und Löschen einer Verbindung

Zum Erstellen einer neuen Verbindung muss ein Tool in der Tool-Palette ausgewählt werden, dem eine `Link`-Instanz zugrunde liegt. Danach müssen zunächst der gewünschte Start-Knoten und anschließend der gewünschte Ziel-Knoten angeklickt werden, wobei beide zur ausgewählten Verbindung kompatibel sein müssen. Das Überprüfen der Kompatibilität erfolgt durch die Auswertung der beteiligten Domänen-Modellelemente. Der Klick auf das Ziel-Element initiiert schließlich die Erzeugung einer entsprechenden `Link`-Verwendung sowie die Übermittlung dieser Verwendung an Start- und Ziel-`Node`.

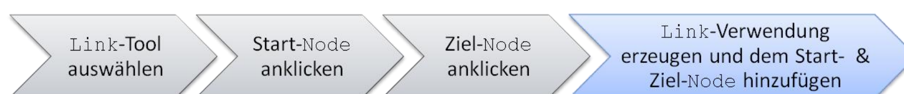

Abb. 8-6: Erforderliche Benutzerinteraktion zum Anlegen einer Verbindung

Die soeben beschriebene Benutzerinteraktion zum Anlegen einer neuen Verbindung ist in Abb. 8-6 schematisch wiedergegeben. Abb. 8-7 zeigt zusätzlich den Inhalt des letzten automatisierten Schritts zum Erzeugen einer `Link`-Verwendung in Form eines Aktivitätsdiagramms. Zu Beginn ist der Prozess weitgehend identisch mit der Erzeugung einer `Node`-Verwendung. Ab dem Anlegen der `Link`-Verwendung sind allerdings deutliche Unterschiede vorhanden. Auf diese Aktivität folgt nämlich das Hinzufügen der Verwendung zu Start- und Ziel-`Node`, wodurch ein konkreter `Link` beschrieben wird. Die rot umrandete Aktivität ist spezifisch für den jeweiligen Verbindungstyp (Attribut- oder Entität-basiert), weil damit festgelegt ist, wie eine Verbindung auf die beteiligten Elemente des Domänen-Modells abgebildet wird. Beim Erzeugen einer Verwendung eines Attribut-basierten `Links` entfällt der Schritt zum „Erzeugen des Domänen-Modellelements“, da eine Verbindung lediglich durch eine Attribut-Zuweisung repräsentiert wird.

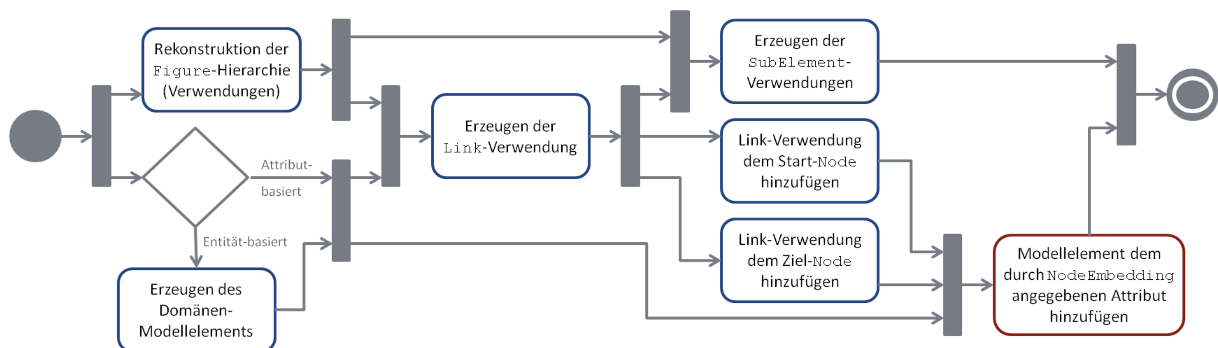

Abb. 8-7: Aktivitätsdiagramm der einzelnen Schritte beim Erzeugen einer `Link`-Verwendung

Das Löschen einer bereits vorhandenen Verbindung ist im Wesentlichen die inverse Operation zur Erstellung einer neuen Verbindung. Hierfür muss der Benutzer die gewünschte Verbindung selektieren und anschließend den *Delete*-Button betätigen. Bei der Ausführung des dadurch initiierten *Commands* werden sämtliche zur `Link`-Verwendung gehörenden *Concepts* aus den jeweiligen Modellen entfernt. Außerdem müssen alle Referenzen, die von anderen *Concepts* auf die im vorherigen Schritt entfernten *Concepts* gehalten werden, ebenfalls gelöscht werden. Bei der Attribut-basierten Variante ist auf der Seite des Domänen-Modells nur die entsprechende Zuweisung zu beseitigen.

8.2.4 Ersetzen von Start/Ziel einer Verbindung

Sowohl Start- als auch Ziel-`Node` einer Verbindung können ausgetauscht werden. Dazu muss der entsprechende Endpunkt der Verbindung selektiert und mittels Drag & Drop auf einen anderen kompatiblen Knoten gezogen werden. Dementsprechend erfolgt eine Anpassung von altem und neuem Start-/Ziel-`Node`, d.h. die `Link`-Verwendung wird vom ursprünglichen Knoten entfernt und dem neuen hinzugefügt. Außerdem werden je nach Verbindungstyp die beteiligten *Concepts* des Domänen-Modells aktualisiert.

8.2.5 Ändern von Properties

Die Möglichkeit zur Änderung von Eigenschaften des ausgewählten Editor-Elements (inkl. gekapselter `Figure` und gemappten Modellelement) mit Hilfe der Properties-View ist ein weiterer bedeutender Bestandteil der Benutzerinteraktion. Hierbei kommen die Vorzüge der automatischen Weiterleitung von Modifikationen innerhalb eines *Concepts* an das zugehörige Java-Objekt zum Vorschein, die durch das *LMM2Java* Modul (Abschnitt 8.1.2) gewährleistet sind. Bei Änderungen an Elementen des *Graphical-Usage*- oder des *Editor-Usage-Models* werden diese direkt an die entsprechenden Objekte

delegiert und von diesen visualisiert. Beispielsweise wird ein Erhöhen oder Verringern der Linienbreite einer `Shape`-Instanz sofort dargestellt.

Lediglich Modifikationen an Elementen des Domänen-Modells müssen über das in *Ecore*-Objekte integrierte Observer-Pattern [81] abgefangen werden. Dies ist z.B. der Fall, wenn der zugewiesene Wert eines bestimmten Attributs unter Verwendung eines `Labels` angezeigt werden soll. Dann muss der entsprechende `EditPart` sensitiv für diese Art von Änderung sein und bei einer Benachrichtigung den neuen Wert dem zugeordneten *Label*-Objekt übergeben.

8.3 Stand der Entwicklung

Die in den vorangegangenen Abschnitten beschriebene Funktionalität ist bereits vollständig implementiert, wobei sämtliche in Abschnitt 1.1 spezifizierten Anforderungen erfüllt werden. Darüber hinaus bietet jeder Designer einige Features, die zu einer komfortableren Benutzbarkeit beitragen. Gemeinsam mit der erwähnten Funktionalität werden diese Features in den kommenden Ausführungen zusammenfassend dargelegt.

Nachdem bei oMME zum gegenwärtigen Zeitpunkt jedes Modell als separate Datei abgelegt werden muss, ist es sinnvoll diese Dateien auf Basis des enthaltenen Modelltyps mit einer eindeutigen Dateiendung zu versehen. Als Standard-Endung wird *lmm* verwendet. Durch das MDF werden folgende Endungen den angegebenen Modelltypen zugewiesen:

- *gdef*: Dateiendung für *Graphical-Definition-Models*
- *edef*: Dateiendung für *Editor-Definition-Models*
- *guse*: Dateiendung für *Graphical-Usage-Models*
- *euse*: Dateiendung für *Editor-Usage-Models*; das Öffnen einer derartigen Datei führt standardmäßig zum Starten einer Designer-Instanz, die den zugehörigen Modellinhalt visualisiert.

Über das Kontextmenü einer *edef*-Datei lässt sich eine neue Designer-Instanz erstellen. Dazu werden im Normalfall drei Dateien für *Domain-*, *Graphical-Usage-* und *Editor-Usage-Model* angelegt. Bei Bedarf kann auch ein bereits vorhandenes *Domain-Model* ausgewählt und die neuen Modellelemente darin abgelegt werden. Individuelle Dateien für *Graphical-Usage-* und *Editor-Usage-Model* sind aber in jedem Fall erforderlich.

Nach dem Öffnen eines Designers stehen die Interaktionsmöglichkeiten des Benutzers im Vordergrund. Die elementarsten Vertreter wurden bereits in Abschnitt 8.2 näher betrachtet. Doch es existieren noch eine Reihe weiterer Interaktionsmöglichkeiten, die jeder Designer zur Verfügung stellt. Einen Überblick über alle angebotenen Interaktionsmöglichkeiten gibt nachfolgende Auflistung:

- Erstellen von `Nodes` bei gleichzeitigem Hinzufügen zu einem `Container` (inkl. Anpassung der gemappten Domänen-Modellelemente).
- Löschen von `Nodes` und entsprechende Dereferenzierung (inkl. Anpassung der gemappten Domänen-Modellelemente).
- Verschieben und Skalieren von `Nodes`.

- Erstellen von `Links` bei gleichzeitigem Verbinden mit je einem Start- und einem Ziel-`Node` (inkl. Anpassung der gemappten Domänen-Modellelemente).
- Neu-Verbinden von `Links` (inkl. Anpassung der gemappten Domänen-Modellelemente).
- Benutzerdefiniertes Routing von `Links` durch Hinzufügen, Verschieben und Löschen von `Benpoints`.
- Verschieben von `Labels`, die einem `Link` zugeordnet sind.
- Bearbeiten von Eigenschaften derjenigen *Concepts*, die das selektierte Element definieren; bisher werden alle literalen Eigenschaften sowie *Concept*-Attribute vom Typ `Color` und `Font` unterstützt.
- Undo- und Redo-Funktionalität für alle vorherigen Operationen.
- Erstellen von Typen über das Kontextmenü der Tool-Palette; nur verfügbar, wenn das Typ-Verwendungs-Konzept zum Einsatz kommt.

Während der Arbeit mit dem Designer wird der Benutzer durch einige, teils von GEF bereitgestellte Features unterstützt. Dazu zählt insbesondere die Möglichkeit sämtliche selektierbaren Elemente innerhalb des Designers an einem Raster ausrichten zu können. Des Weiteren gibt jeder Designer Feedback über die auszuführende Operation, d.h. der Mauszeiger zeigt an, ob eine bestimmte Operation an der aktuellen Position zulässig ist oder nicht.

Ein letztes Feature, das jeder Designer besitzt, ist die Möglichkeit das geöffnete Diagramm als Bilddatei zu exportieren. Hierfür existiert ein spezieller Button in der Werkzeugleiste von Eclipse. Unterstützt werden bislang die Formate PNG, JPEG und Bitmap.

8.4 Ausblick

Mit dem Model Designer Framework entworfene Designer sollen den Inhalt eines Domänenmodells sowohl vorteilhaft visualisieren als auch dem Benutzer eine gute Werkzeugunterstützung bieten. Durch die Ausrichtung von Elementen am Raster und der Möglichkeit Verbindungen mit `Bendpoints` zu versehen ist bereits ein großer Bereich dieser Anforderung abgedeckt. Denkbar sind aber durchaus noch weitere Features, wie z.B. das automatische Ausrichten des selektierten Elements an anderen Elementen, ein- und ausklappbare `Compartments` sowie sog. springende Verbindungen. Letztere werden dann verwendet, wenn sich zwei Verbindungen kreuzen. Dabei „springt“ eine dieser Verbindungen über die andere in der Form eines Halbkreises.

Zur vorteilhaften Visualisierung leisten auch die verwendeten grafischen Formen, die bestimmte Modellelemente oder Beziehungen repräsentieren, einen wesentlichen Beitrag. Deshalb kann es sinnvoll sein, benutzerdefinierte `Figures` und `Layouts` zu implementieren, die schließlich über ein weiteres *Extended Graphical-Meta-Model* dem MDF bekannt gemacht werden müssen. Dazu gilt es die Basis-Typen von Draw2D sowie die entsprechenden Elemente des *Graphical-Meta-Model* zu erweitern.

Bisher können mit dem MDF lediglich neue Domänen-Modelle erstellt bzw. der Inhalt bereits vorhandener Modelle ergänzt werden. Es ist nicht möglich Teile von existierenden Modellen zu visualisieren. Diese Funktionalität kann jedoch hilfreich sein, wenn ein Domänen-Modell nicht mit einem Designer, sondern beispielsweise mit dem *Textual Model Editor* erstellt wurde. Dann sollten automatisch zu diesem Modell und einem passenden *Editor-Definition-Model* die zugehörigen *Graphical-*

Usage- und *Editor-Usage-Models* erzeugt werden können. Hierfür sind allerdings tiefergehende Kenntnisse im Bereich des automatisierten Layoutings von Diagrammen notwendig, wie sie z.B. in [82] und [83] vermittelt werden.

Jeder Designer kann gleichzeitig sowohl die Graph- als auch die Geometrie-basierte Variante zur grafischen Repräsentation von Beziehungen anbieten (Anwendungsfall in Kapitel 7). Beziehungen lassen sich potentiell auch nicht-visuell umsetzen. Dies wird dadurch erreicht, indem bestimmte visualisierte Modellelemente (z.B. mittels Doppelklick) geöffnet werden können und demzufolge eine neue Designer-Instanz geladen wird, die den Inhalt des soeben geöffneten Modellelements anzeigt. Dadurch kann letztlich eine bessere Übersichtlichkeit erreicht werden, weil meist nur ein kleiner Ausschnitt des Domänen-Modells visualisiert wird. Ein konkretes Beispiel aus der Prozessmodellierung sind komposite Prozesse, deren Inhalt stets in einem separaten Diagramm angezeigt werden kann. Ebenso vorstellbar ist das Öffnen von Data-Items, die an verschiedenen Prozessen anhaften. Für den zweiten Anwendungsfall bietet sich jedoch die Verwendung eines speziellen Editor-Definition-Models an, das beispielsweise einen Designer zur Erstellung von Entity-Relationship-Diagrammen spezifiziert. Kommt bei der Modellierung das Typ-Verwendungs-Konzept zum Einsatz, so ist zwingend eine Art Model-Registry erforderlich. Dies liegt darin begründet, dass beim Doppelklick auf eine Verwendung der zugehörige Typ geöffnet wird und er dann in verschiedenen Diagrammen verwendet werden kann. Welches *Editor-Usage-Model* nun zu dem Typ gehört, muss an globaler Stelle abgelegt sein. Andernfalls würde für jede Verwendung in einem separaten Diagramm ein gesondertes *Edtior-Usage*- und *Graphical-Usage-Model* angelegt werden. Die Implementierung einer entsprechenden Model-Registry hätte allerdings den Rahmen der vorliegenden Arbeit gesprengt, weshalb das Feature, Typen öffnen zu können, bewusst vernachlässigt wurde. Diese Funktionalität ist jedoch bereits konzeptuell mit dem Attribut `contentEUModelURI` des `DiagramElement`-*Concepts* (Abschnitt 6.3.2) vorgesehen.

Eine letzte wichtige Ergänzung ist die Implementierung und Integration eines Fehlererkennungsmoduls, das auch die Fähigkeit besitzt, identifizierte Fehler entsprechend zu visualisieren bzw. im Optimalfall einen Vorschlag unterbreitet, wie sie aufgelöst werden können. Insbesondere sind dabei Fehler gemeint, die durch Änderungen am Domänen-Meta-Modell hervorgerufen werden, z.B. wenn das `ANDConnector`-*Concept* (Abb. 7-1) gelöscht wird. Daraufhin soll in allen Designern, die eine Instanz dieses *Concepts* darstellen, die grafische Repräsentation als fehlerhaft markiert werden. Da jeder Designer ausschließlich `EditorElement`-Instanzen und nicht direkt die *Concepts* des Domänen-Modells visualisiert, muss eine Implementierung der Fehlererkennung lediglich überprüfen, ob die gemappten Modellelemente valide sind.

Aufgrund der durch das *LMM2Java*-Modul gegebenen Flexibilität lassen sich alle voranstehenden Erweiterungen integrieren, ohne dass eine Modifikation von Kernelementen des MDF notwendig ist. Die Integration in den Kern des MDF ist aber dennoch möglich und sicherlich bei einigen der angesprochenen Features sinnvoll. Damit erfüllt das Model Designer Framework die Aufgabe eines flexiblen Systems zur Spezifikation grafischer Editoren, innerhalb deren das Typ-Verwendungs-Konzept als integraler Bestandteil zur Erzeugung von Modellelementen zum Einsatz kommt.

Literaturverzeichnis

1. Jablonski, S., Volz, B., Dornstauder, S.: On the Implementation of Tools for Domain Specific Process Modelling. ENASE 2009, Milan, Italy (2009)
2. van der Aalst, W., Jablonski, S.: Dealing with workflow change: identification of issues and solutions. Computer systems science and engineering **15** (2000) 267-276
3. Dadam, P., Reichert, M., Rinderle-Ma, S.: Prozessmanagementsysteme. Informatik-Spektrum (2010) 1-13
4. Clark, T., Sammut, P., Willans, J.: Applied metamodelling: a foundation for language driven development. CETEVA (2008)
5. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. ACM (2006) 616
6. Volz, B.: Dissertation über neuartige Modellierungsparadigmen und das LMM. Universität Bayreuth, Bayreuth (2010)
7. Jablonski, S., Bussler, C.: Workflow Management - Modelling Concepts, Architecture and Implementation. International Thomson Computer Press, London (1996)
8. Seidewitz, E.: What models mean. IEEE software **20** (2003) 26-32
9. Ober, I., Prinz, A.: What do we need metamodels for? : 4th Nordic Workshop on UML and Software Modelling. Citeseer, GRIMSTAD, NORWAY (2006) 8-28
10. The Eclipse Foundation: Eclipse.org home. http://www.eclipse.org/ (01.09.2010)
11. The Eclipse Foundation: Eclipse Modeling Project. http://www.eclipse.org/modeling/emf/ (01.09.2010)
12. OMG: Meta Object Facility (MOF) Core Specification. Version 2.0 (2006)
13. Gerber, A., Raymond, K.: MOF to EMF: there and back again. ACM (2003) 64
14. The Eclipse Foundation: Xtext. http://www.eclipse.org/Xtext/ (01.09.2010)
15. OMG Inc.: Object Management Group. http://www.omg.org/ (01.09.2010)
16. Atkinson, C.: Meta-modeling for distributed object environments. Published by the IEEE Computer Society (1997) 90
17. Atkinson, C., Kühne, T.: Meta-level independent modelling. Citeseer (2000) 12-16
18. Atkinson, C., Kühne, T.: Concepts for comparing modeling tool architectures. Model Driven Engineering Languages and Systems (2005) 398-413
19. OMG: OMG Unified Modeling Language (OMG UML). Infastructure (2003)
20. Network Working Group: A Universally Unique IDentifier (UUID) URN Namespace. Microsoft, Refactored Networks, DataPower Technology (2005)
21. Mackinlay, J.: Automating the design of graphical presentations of relational information. ACM Transactions on Graphics (TOG) **5** (1986) 141
22. Costagliola, G., De Lucia, A., Orefice, S., Polese, G.: A classification framework to support the design of visual languages. Journal of Visual Languages & Computing **13** (2002) 573-600

23. Roth, B.: Abbildung von Zyklen in graphbasierten Modellen auf blockorientierte Strukturen. Angewandte Informatik IV. Universität Bayreuth, Bayreuth (2008)

24. ProDatO GmbH: i>ProcessManager. Handbuch i>PM Integrated Process Manager, Version 2.5.0 (2005)

25. Künneth, T.: Einstieg in Eclipse 3.5. Galileo Computing, Bonn (2009)

26. Soyatec: eBPMN. http://www.soyatec.com/ebpmn/ (01.09.2010)

27. OMG: Business Process Model and Notation (BPMN). FTF Beta 1 for Version 2.0 (2009)

28. SAP AG: SAP - Business Management Software Solutions Applications and Services. http://www.sap.com/ (01.09.2010)

29. IDS Scheer AG: 'ARIS Platform' Produktbroschüre. Saarbrücken (2008)

30. ARIS: Modellierungsstandards. http://www.ids-scheer.de/de/ARIS/ARIS_Modellierungsstandards/80727.html (01.09.2010)

31. OASIS: Web Services Business Process Execution Language. Version 2.0 (2007)

32. OMG: Business Process Definition Metamodel. Object Management Group, Needham, MA (2003)

33. Andries, M., Engels, G., Rekers, J.: How to represent a visual specification. Visual Language Theory (1998) 245-260

34. Jeusfeld, M.A., Quix, C., Jarke, M.: ConceptBase User Manual. Version 7.2. Tilburg University & RWTH Aachen (2010)

35. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, K.: Telos: A language for representing knowledge about information systems (revised). Technical Report KRR-TR-89-1, Department of Computer Science, University of Toronto (1990)

36. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase - a deductive object base for meta data management. Journal of Intelligent Information Systems **4** (1995) 167-192

37. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based tool integration with moflon. ACM (2008) 807-810

38. Weisemöller, I., Wieber, M., Stuckert, D., Klar, F.: MOFLON by Example. (2009)

39. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. Vol. 17. Citeseer (2001)

40. Institute for Software Integrated Systems: GME 5 User's Manual. Vanderbilt University (2005)

41. Eddon, G., Eddon, H.: Inside distributed COM:[Entdecken Sie die Programmierung von verteilten Applikationen]. Microsoft Press Deutschland, Unterschleißheim (1998)

42. Loos, P.: Go to COM. Addison-Wesley (2000)

43. de Lara, J., Vangheluwe, H.: Using AToM3 as a Meta-CASE tool. ICEIS (2002) 642–649

44. Python Software Foundation: Python Programming Lanuage. http://www.python.org/ (01.09.2010)

45. Dubé, D.: Getting started with AToM³ Meta-Modeling. http://atom3.cs.mcgill.ca/tutorials/starting.html (01.09.2010)

46. MetaCase: ABC to MetaCase Technology. Whitepaper (2004)

47. MetaCase: Symbole zeichnen. http://www.metacase.com/de/mwb/draw_symbols.html (01.09.2010)

48. MetaCase: Integration. http://www.metacase.com/de/mwb/draw_symbols.html (01.09.2010)

49. MetaCase: Show FAQ. http://www.metacase.com/faq/showfaq.asp?cate=MWB (01.09.2010)

50. The Eclipse Foundation: Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/ (31.05.2010)

51. Gronback, R.C., Reitsma, J., Ing, D.: GMF Tutorial. http://wiki.eclipse.org/GMF_Tutorial (01.09.2010)

52. Slapeta, S.: Ein Vergleich zwischen Visual Studio 2005 und Eclipse Graphical Modeling Framework zur Unterstützung von modellgetriebener Softwareentwicklung. TU Wien, Wien (2006)

53. Microsoft: Microsoft Corporation. http://www.microsoft.com/ (01.09.2010)

54. Cook, S., Jones, G., Kent, S., Willis, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, Amsterdam (2007)

55. Microsoft: What is new in the DSL Tools for Visual Studio 2010. http://code.msdn.microsoft.com/DslTools/Wiki/Print.aspx?title=What%27s%20new&version=10 (09.07.2010)

56. Prieur, J.-M.: "Nested StateMachine" DSL Tools Sample demonstrating the usage of the ModelBus. http://code.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=DslTools&DownloadId=5981 (09.07.2010)

57. Volz, B.: Einstieg in Visual C# 2008. Galileo Computing, Bonn (2008)

58. ClariuS Consulting: Visual T4 Blog. http://blog.visualt4.com/2009_02_01_archive.html (09.07.2010)

59. Jeusfeld, M., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. (2009) 424

60. MetaCase: Success Stories. http://www.metacase.com/cases/ (01.09.2010)

61. Farail, P.: Topcased. http://gforge.enseeiht.fr/docman/view.php/52/3627/TOPCASED-presentation-2h.pdf (01.09.2010)

62. The Eclipse Foundation: SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/ (01.09.2010)

63. Northover, S., Wilson, M.: SWT: the standard widget toolkit, Vol. 1. Addison-Wesley Longman, Amsterdam (2004)

64. Loy, M., Eckstein, R., Wood, D., Elliot, J., Cole, B.: Java Swing. O'Reilly Media, Sebastopol, CA (2002)

65. The Eclipse Foundation: Draw2d Overview. http://help.eclipse.org/ganymede/topic/org.eclipse.draw2d.doc.isv/guide/overview.html (01.09.2010)

66. Lee, D.: Display a UML Diagram using Draw2D. http://eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html (01.09.2010)

67. The Eclipse Foundation: Eclipse Graphical Editing Framework. http://www.eclipse.org/gef/ (01.09.2010)

68. Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM (2004)

69. Reinhard, T., Glinz, M.: Automatic Placement of Link Labels in Diagrams. FlexiTools (2010)

70. Lahres, B., Gregor, R.: Praxisbuch Objektorientierung. Galileo Computing, Bonn (2006)

71. Aniszczyk, C., Hudson, R.: Create an Eclipse-based application using the Graphical Editing Framework. IBM (2007)

72. The Eclipse Foundation: Zest: The Eclipse Visualization Toolkit. http://www.eclipse.org/gef/zest/ (01.09.2010)

73. OSGi Alliance: OSGi Alliance. http://www.osgi.org/ (01.09.2010)

74. The Eclipse Foundation: Equinox. http://www.eclipse.org/equinox/ (01.09.2010)

75. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston, MA (2009)

76. Oracle: java.com: Java für Sie. http://www.java.com (01.09.2010)

77. Graham, H.: JavaBeans™. Sun Microsystems, Mountain View, CA (1997)

78. Foley, J., Van Dam, A., Feiner, S., Hughes, J.: Computer Graphics: Principles and Practice. Addison-Wesley Longman, Amsterdam (1995)

79. Thalheim, B.: Entity-relationship modeling: foundations of database technology. Springer Verlag (2000)

80. Oracle: Trail: The Reflection API. http://download.oracle.com/javase/tutorial/reflect/index.html (01.09.2010)

81. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-wesley Reading, MA (1995)

82. Batini, C., Nardelli, E., Tamassia, R.: A layout algorithm for data flow diagrams. IEEE Transactions on Software Engineering **12** (1986) 538-546

83. Eiglsperger, M., Kaufmann, M., Siebenhaller, M.: A topology-shape-metrics approach for the automatic layout of UML class diagrams. ACM (2003) 189

Anhang A Definition-Models des Prozess-Designers

In den drei folgenden Abschnitten sind die vollständigen Versionen von *Domain-Meta-*, *Graphical-Definition-* und *Editor-Definition-Model* des Prozess-Designers aus Kapitel 7 abgedruckt. Hierfür wird die konkrete Syntax des LMM verwendet.

A.1 Domain-Meta-Model

```
model POPMMetaModel {
  modelURI = " platform:///plugin/de.ubt.ai4.omme.mdf/models/EditorMetaModel.lmm ";

  level M2 {
    package POPM {
      abstract concept Node {
        attributes {
          concept Node successors { multiplicity = zeroOrMore; }
        }
      }

      concept Process extends Node {
        attributes {
          string name                  { multiplicity = one; }
          concept Node innerContent    { multiplicity = zeroOrMore; }
          concept DataItem inputData   { multiplicity = zeroOrMore; }
          concept DataItem outputData  { multiplicity = zeroOrMore; }
        }
      }

      concept ANDConnector extends Node {
      }

      concept DataItem {
        attributes {
          string typeName { multiplicity = one; }
          public concept DataFlow incomingDataFlows {
            multiplicity = zeroOrMore;
            opposite = target;
          }
          public concept DataFlow outgoingDataFlows {
            multiplicity = zeroOrMore;
            opposite = source;
          }
        }
      }

      concept DataFlow {
        attributes {
          concept DataItem source {
            multiplicity = zeroOrOne;
            opposite = outgoingDataFlows;
          }
          concept DataItem target {
            multiplicity = zeroOrOne;
            opposite = incomingDataFlows;
          }
          string trafoDefinition;
        }
      }
    }
  }
}
```

A.2 Graphical-Definition-Model

```
model ProcessGraphicalDefinition {
  modelURI = "platform:///resource/iPM2Light/models/ProcessGraphicalDefinition.gdef";

  include "platform:///plugin/de.ubt.ai4.omme.mdf/models/GraphicalMetaModel.lmm";
  include "platform:///plugin/de.ubt.ai4.omme.mdf.ui.graph/models/CommonGraphics.lmm";
```

```
level D1 instanceOf Graphical.D2 instanceOf CommonGraphics.D2 {
  package POPM {
    /*
     * Enthält die Spezifikation der Process-Figur
     */
    package Process {
      concept ProcessFigure instanceOf Graphical.D2.Core.ShapeWithSelectableChild {
        values {
          concept backgroundColor = ProcessBackColor;
          concept foregroundColor = ProcessForeColor;
          concept layoutManager = ProcessBorderLayout;
          concept children = ProcessGraphicalDefinition.D1.POPM.Process.Center.ProcessRect,
            ProcessGraphicalDefinition.D1.POPM.Process.Bottom.Footer;
          concept selectableSubFigure = ProcessGraphicalDefinition.D1.POPM.Process.Center.ProcessRect;
        }
      }

      concept ProcessBackColor instanceOf Graphical.D2.Core.Types.Color {
        values {
          integer red = 196;
          integer green = 226;
          integer blue = 242;
        }
      }

      concept ProcessForeColor instanceOf Graphical.D2.Core.Types.Color {
        values {
          integer red = 24;
          integer green = 81;
          integer blue = 112;
        }
      }

      concept ProcessBorderLayout instanceOf CommonGraphics.D2.Common.Layout.BorderLayout {
      }

      /*
       * Spezifikation der eigentlichen Prozess-Figur, nämlich des Prozessrechtecks (ProcessRect)
       * inkl. dessen Inhalt.
       */
      package Center {
        concept ProcessRect instanceOf CommonGraphics.D2.Common.Figure.GradientRectangle {
          values {
            concept constraint = ProcessRectConstraint;
            concept layoutManager = ProcessStackLayout;
            concept children = ProcessGraphicalDefinition.D1.POPM.Process.Center.Lower.LowerLayer,
                ProcessGraphicalDefinition.D1.POPM.Process.Center.Upper.UpperLayer;
          }
        }

        concept ProcessRectConstraint
            instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
          values {
            /*
             * Angabe, dass die mittlere Region des Border Layouts verwendet werden soll
             */
            integer value = 2;
          }
        }

        concept ProcessStackLayout instanceOf CommonGraphics.D2.Common.Layout.StackLayout {}


        /*
         * Dieses Package enthält alle Elemente zur mittigen Anzeige des Prozess-Labels.
         */
        package Upper {
          concept UpperLayer instanceOf Graphical.D2.Core.Figure {
            values {
              concept layoutManager = UpperLayerLayout;
              concept children = ProcessLabel;
            }
          }

          concept UpperLayerLayout instanceOf CommonGraphics.D2.Common.Layout.GridLayout {
            values {
              integer numColumns = 1;
              boolean makeColumnsEqualWidth = true;
            }
          }

          concept ProcessLabel instanceOf Graphical.D2.Core.Label {
            values {
              string text = "Process";
              concept constraint = ProcessLabelConstraint;
            }
          }
```

```
    concept ProcessLabelConstraint
        instanceOf CommonGraphics.D2.Common.Layout.Constraint.GridConstraint {
      values {
        boolean grabExcessHorizontalSpace = true;
        boolean grabExcessVerticalSpace = true;
        integer horizontalAlignment = 16777216; // Wert für eine mittige Ausrichtung
        integer horizontalIndent = 1;
        integer horizontalSpan = 1;
        integer verticalAlignment = 16777216; // Wert für eine mittige Ausrichtung
        integer verticalSpan = 1;
      }
    }
  }

  /*
   * Dieses Package enthält alle Elemente zur Anzeige eines kleinen Quadrats in der
   * rechten unteren Ecke des ProcessRects. Damit soll signalisiert werden, ob ein
   * Prozess komposit ist oder nicht.
   */
  package Lower {
    concept LowerLayer instanceOf Graphical.D2.Core.Figure {
      values {
        concept layoutManager = LowerLayout;
        concept children = CompositeMarker;
      }
    }

    concept LowerLayout instanceOf CommonGraphics.D2.Common.Layout.CornerLayout { }

    concept CompositeMarker instanceOf CommonGraphics.D2.Common.Figure.Rectangle {
      values {
        concept constraint = MarkerPosition;
        concept preferredSize = MarkerSize;
        integer lineWidth = 6;
      }
    }

    concept MarkerPosition instanceOf CommonGraphics.D2.Common.Layout.Constraint.CornerConstraint {
      values {
        /*
         * Angabe, dass die Figur in der rechten unteren Ecke platziert werden soll
         */
        integer position = 3;
      }
    }

    concept MarkerSize instanceOf Graphical.D2.Core.Types.Dimension {
      values {
        integer width = 12;
        integer height = 12;
      }
    }
  }
}

/*
 * Spezifiziert je einen Container für Eingags- und Ausgangsdaten
 */
package Bottom {
  concept Footer instanceOf Graphical.D2.Core.Figure {
    values {
      concept constraint = FooterConstraint;
      concept layoutManager = FooterLayout;
      concept children = LeftSection, MiddleSection, RightSection;
    }
  }

  concept FooterConstraint
      instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
    values {
      /*
       * Angabe, dass die Figur in einem BorderLayout am unteren Rand platziert wird
       */
      integer value = 32;
    }
  }

  concept FooterLayout instanceOf CommonGraphics.D2.Common.Layout.BorderLayout { }

  concept LeftSection instanceOf Graphical.D2.Core.Figure {
    values {
      concept constraint = LeftSectionConstraint;
      concept layoutManager = LeftSectionLayout;
    }
  }
```

```
    concept LeftSectionConstraint
        instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
      values {
        integer value = 1;
      }
    }

    concept LeftSectionLayout instanceOf CommonGraphics.D2.Common.Layout.FlowLayout {
      values {
        boolean stretchMinorAxis = true;
        boolean horizontal = false;
        integer minorSpacing = -1;
        integer emptyWidthExtent = 50;
        integer emptyHeightExtent = 12;
      }
    }

    concept MiddleSection instanceOf Graphical.D2.Core.Figure {
      values {
        concept constraint = MiddleSectionConstraint;
      }
    }

    concept MiddleSectionConstraint
        instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
      values {
        integer value = 2;
      }
    }

    concept RightSection instanceOf Graphical.D2.Core.Figure {
      values {
        concept layoutManager = RightSectionLayout;
        concept constraint = RightSectionConstraint;
      }
    }

    concept RightSectionLayout instanceOf CommonGraphics.D2.Common.Layout.FlowLayout {
      values {
        boolean stretchMinorAxis = true;
        boolean horizontal = false;
        integer minorSpacing = -1;
        integer emptyWidthExtent = 50;
        integer emptyHeightExtent = 12;
      }
    }

    concept RightSectionConstraint
        instanceOf CommonGraphics.D2.Common.Layout.Constraint.BorderLayoutConstraint {
      values {
        integer value = 4;
      }
    }
  }
}

/*
 * Enthält die Spezifikation der ANDConnector-Figur
 */
package Connector {
  concept ANDFigure instanceOf CommonGraphics.D2.Common.Figure.GradientEllipse {
    values {
      concept backgroundColor = ANDBackColor;
      concept foregroundColor = ANDForeColor;
      concept anchor = ConnectorAnchor;
      concept layoutManager = ANDLayout;
      concept children = ANDLabel;
    }
  }

  concept ANDLabel instanceOf Graphical.D2.Core.Label {
    values {
      string text = "AND";
      concept constraint = ANDLabelConstraint;
    }
  }
  concept ANDLayout instanceOf CommonGraphics.D2.Common.Layout.GridLayout {
    values {
      integer numColumns = 1;
      boolean makeColumnsEqualWidth = true;
    }
  }
```

```
  concept ANDLabelConstraint
      instanceOf CommonGraphics.D2.Common.Layout.Constraint.GridConstraint {
    values {
      boolean grabExcessHorizontalSpace = true;
      boolean grabExcessVerticalSpace = true;
      integer horizontalAlignment = 16777216; // Wert für eine mittige Ausrichtung
      integer horizontalIndent = 1;
      integer horizontalSpan = 1;
      integer verticalAlignment = 16777216; // Wert für eine mittige Ausrichtung
      integer verticalSpan = 1;
    }
  }

  concept ANDBackColor instanceOf Graphical.D2.Core.Types.Color {
    values {
      integer red = 215;
      integer green = 190;
      integer blue = 230;
    }
  }

  concept ANDForeColor instanceOf Graphical.D2.Core.Types.Color {
    values {
      integer red = 88;
      integer green = 45;
      integer blue = 115;
    }
  }

  concept ConnectorAnchor instanceOf CommonGraphics.D2.Common.Anchor.EllipseAnchor {
  }
}

/*
 * Enthält die Spezifikation der DataItem-Figur
 */
package DataItem {
  concept DataItemFigure instanceOf CommonGraphics.D2.Common.Figure.GradientRectangle {
    values {
      concept backgroundColor = DataItemBackColor;
      concept foregroundColor = DataItemForeColor;
      concept preferredSize = DataItemDefaultSize;
      concept layoutManager = InnerRectLayout;
      concept children = ProcessGraphicalDefinition.D1.POPM.DataItem.Label.DataItemLabel;
    }
  }

  concept DataItemBackColor instanceOf Graphical.D2.Core.Types.Color {
    values {
      integer red = 200;
      integer green = 200;
      integer blue = 200;
    }
  }

  concept DataItemForeColor instanceOf Graphical.D2.Core.Types.Color {
    values {
      integer red = 70;
      integer green = 70;
      integer blue = 70;
    }
  }

  concept DataItemDefaultSize instanceOf Graphical.D2.Core.Types.Dimension {
    values {
      integer width = 61; integer height = 24;
    }
  }

  concept InnerRectLayout instanceOf CommonGraphics.D2.Common.Layout.GridLayout {
    values {
      integer numColumns = 1;
      boolean makeColumnsEqualWidth = true;
      integer marginHeight = 0;
      integer marginWidth = 2;
    }
  }

  package Label {
    concept DataItemLabel instanceOf Graphical.D2.Core.Label {
      values {
        string text = "DataItem";
        concept constraint = DataItemLabelConstraint;
      }
    }
```

```
          concept DataItemLabelConstraint
              instanceOf CommonGraphics.D2.Common.Layout.Constraint.GridConstraint {
            values {
              boolean grabExcessHorizontalSpace = true;
              boolean grabExcessVerticalSpace = true;
              integer horizontalAlignment = 16777216; // Wert für eine mittige Ausrichtung
              integer horizontalIndent = 1;
              integer horizontalSpan = 1;
              integer verticalAlignment = 16777216; // Wert für eine mittige Ausrichtung
              integer verticalSpan = 1;
            }
          }
        }
      }

      package Flows {
        concept ControlFlow instanceOf Graphical.D2.Core.Connection {
          values {
            concept foregroundColor = ControlFlowColor;
            concept targetDecoration = FlowDecoration;
          }
        }

        concept ControlFlowColor instanceOf Graphical.D2.Core.Types.Color {
          values {
            integer red = 150; integer green = 150; integer blue = 150;
          }
        }

        concept FlowDecoration instanceOf CommonGraphics.D2.Common.Decoration.PolygonDecoration { }

        concept DataFlow instanceOf Graphical.D2.Core.Connection {
          values {
            concept foregroundColor = DataFlowColor;
            concept targetDecoration = FlowDecoration;
            concept layoutManager = DataFlowLayout;
            concept children = DataFlowLabel;
          }
        }

        concept DataFlowColor instanceOf Graphical.D2.Core.Types.Color {
          values {
            integer red = 0; integer green = 0; integer blue = 0;
          }
        }

        concept DataFlowLayout instanceOf CommonGraphics.D2.Common.Layout.DelegatingLayout {
        }

        concept DataFlowLabel instanceOf Graphical.D2.Core.Label {
          values {
            string text = "Trafo-Def.";
            concept constraint = DataFlowLabelLocator;
          }
        }

        concept DataFlowLabelLocator instanceOf Graphical.D2.Core.Types.LabelLocator {
          values {
            integer offsetX = 0; integer offsetY = 15;
          }
        }
      }
    }
  }
}
```

A.3 Editor-Definition-Model

```
model ProcessEditorDefinition {
  modelURI = "platform:///resource/iPM2Light/models/ProcessEditorDefinition.edef";

  include "platform:///plugin/de.ubt.ai4.omme.mdf/models/EditorMetaModel.lmm";
  include "platform:///resource/iPM2Light/models/ProcessGraphicalDefinition.gdef";
  include "platform:///resource/iPM2Light/models/POPMMetaModel.lmm";
  include "platform:///plugin/de.ubt.ai4.omme.mdf.ui.editor/models/CommonEditParts.lmm";

  level D1 instanceOf Editor.D2 references ProcessGraphicalDefinition.D1 {
    package POPM {
      concept ProcessDiagram instanceOf Editor.D2.Core.Diagram {
        values {
          enum paradigm = TYPE_USAGE;
          fqn modelRootFQN = POPMMetaModel.M2.POPM.Process;
          concept embeddings = ProcessNodeEmbedding, ANDNodeEmbedding;
          concept links = ProcessEditorDefinition.D1.POPM.Links.ControlFlowLink,
            ProcessEditorDefinition.D1.POPM.Links.DataFlowLink;
        }
      }
```

```
concept ProcessNodeEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
  values {
    concept reference = ProcessEditorDefinition.D1.POPM.Nodes.Process.ProcessNode;
    fqn containmentQN = innerContent;
  }
}

concept ANDNodeEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
  values {
    concept reference = ProcessEditorDefinition.D1.POPM.Nodes.ANDNode;
    fqn containmentQN = innerContent;
  }
}

package Nodes {
  package Process {
    concept ProcessNode instanceOf Editor.D2.Core.Node {
      values {
        concept figure = ProcessGraphicalDefinition.D1.POPM.Process.ProcessFigure;
        fqn modelElementFQN = POPMMetaModel.M2.POPM.Process;
        fqn toolingAttributeQN = name;
        string toolingTitle = "Process";
        concept subElements = ProcessLabel, ContentBasedRewriterService,
          InputDataCompartment, OutputDataCompartment;
      }
    }

    concept ProcessLabel instanceOf Editor.D2.Core.EditorLabel {
      values {
        concept owner = ProcessNode;
        fqn displayAttributeQN = name;
        concept label = ProcessGraphicalDefinition.D1.POPM.Process.Center.Upper.ProcessLabel;
      }
    }

    concept ContentBasedRewriterService instanceOf Editor.D2.Core.RewriterService {
      values {
        concept owner = ProcessNode;
        concept graphElement = ProcessGraphicalDefinition.D1.POPM.Process.Center.Lower.CompositeMarker;
        concept rewriters = ContentBasedVisibility;
        concept condition = ContentBasedCondition;
      }
    }

    concept ContentBasedVisibility instanceOf CommonEditParts.D2.Common.VisibilityRewriter { }

    concept ContentBasedCondition instanceOf CommonEditParts.D2.Common.ContentBasedCondition {
      values {
        fqn attributeName = innerContent;
      }
    }

    /*
     * Spezifikation des Compartments für Eingangsdaten
     */
    concept InputDataCompartment instanceOf Editor.D2.Core.Compartment {
      values {
        concept owner = ProcessNode;
        concept contentPane = ProcessGraphicalDefinition.D1.POPM.Process.Bottom.LeftSection;
        concept embeddings = InputDataEmbedding;
      }
    }



    concept InputDataEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
      values {
        concept reference = ProcessEditorDefinition.D1.POPM.Nodes.DataItem.DataItemNode;
        fqn containmentQN = inputData;
      }
    }

    /*
     * Spezifikation des Compartments für Ausgangsdaten
     */
    concept OutputDataCompartment instanceOf Editor.D2.Core.Compartment {
      values {
        concept owner = ProcessNode;
        concept contentPane = ProcessGraphicalDefinition.D1.POPM.Process.Bottom.RightSection;
        concept embeddings = OutputDataEmbedding;
      }
    }
```

```
          concept OutputDataEmbedding instanceOf Editor.D2.Core.NodeEmbedding {
            values {
              concept reference = ProcessEditorDefinition.D1.POPM.Nodes.DataItem.DataItemNode;
              fqn containmentQN = outputData;
            }
          }
        }

        concept ANDNode instanceOf Editor.D2.Core.Node {
          values {
            concept figure = ProcessGraphicalDefinition.D1.POPM.Connector.ANDFigure;
            fqn modelElementFQN = POPMMetaModel.M2.POPM.ANDConnector;
            string toolingTitle = "AND";
          }
        }

        package DataItem {
          concept DataItemNode instanceOf Editor.D2.Core.Node {
            values {
              concept figure = ProcessGraphicalDefinition.D1.POPM.DataItem.DataItemFigure;
              fqn modelElementFQN = POPMMetaModel.M2.POPM.DataItem;
              fqn toolingAttributeQN = typeName;
              string toolingTitle = "Data Item";
              concept subElements = DataItemLabel;
            }
          }

          concept DataItemLabel instanceOf Editor.D2.Core.EditorLabel {
            values {
              concept owner = DataItemNode;
              fqn displayAttributeQN = typeName;
              concept label = ProcessGraphicalDefinition.D1.POPM.DataItem.Label.DataItemLabel;
            }
          }
        }
      }

      package Links {
        concept ControlFlowLink instanceOf Editor.D2.Core.AttributeBasedLink {
          values {
            fqn modelElementFQN = POPMMetaModel.M2.POPM.Node;
            fqn attributeQN = successors;
            concept figure = ProcessGraphicalDefinition.D1.POPM.Flows.ControlFlow;
            string toolingTitle = "Control Flow";
          }
        }

        concept DataFlowLink instanceOf Editor.D2.Core.EntityBasedLink {
          values {
            fqn modelElementFQN = POPMMetaModel.M2.POPM.DataFlow;
            fqn sourceFQN = POPMMetaModel.M2.POPM.DataItem.outgoingDataFlows;
            fqn targetFQN = POPMMetaModel.M2.POPM.DataItem.incomingDataFlows;
            concept figure = ProcessGraphicalDefinition.D1.POPM.Flows.DataFlow;
            string toolingTitle = "Data Flow";
            concept subElements = DataTrafoLabel;
          }
        }

        concept DataTrafoLabel instanceOf Editor.D2.Core.EditorLabel {
          values {
            concept owner = DataFlowLink;
            fqn displayAttributeQN = trafoDefinition;
            concept label = ProcessGraphicalDefinition.D1.POPM.Flows.DataFlowLabel;
          }
        }
      }
    }
  }
}
```

Anhang B Inhalt der beiliegenden CD-ROM

Der Inhalt der beiliegenden CD-ROM umfasst das Model Designer Framework in drei verschiedenen Varianten, die durch die Wurzel-Verzeichnisse der CD repräsentiert werden (Abb. B-1). Zur Installation ist der jeweilige Ordner stets auf ein wiederbeschreibbares Medium zu kopieren.

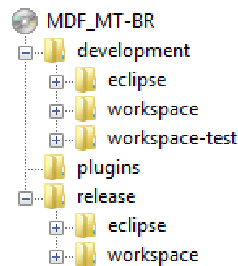

Abb. B-1: Verzeichnisstruktur der CD

Die erste, als Release bezeichnete Variante befindet sich im Verzeichnis *release/eclipse*. Dabei handelt es sich um ein voll funktionsfähiges Eclipse (Distribution für Windows 32Bit) in der Version 3.6 [10] mit installierten oMME- und MDF-Plug-Ins. Außerdem wird ein Workspace bereitgestellt (Verzeichnis *release/workspace*), der jeweils eine Definition für einen Prozess- und für einen Entity-Relationship-Designer enthält. Zu jedem Designer existiert ein Beispiel-Diagramm, das in Abb. B-2 bzw. Abb. B-3 abgedruckt ist. Damit repräsentieren die beiden, mit Hilfe des MDF definierten Designer die zweite Version von i>PM [24], sprich i>PM². Sollen auf dessen Basis neue Prozess- oder ER-Modelle erzeugt werden, so muss beim Starten von Eclipse der o.g. Workspace und anschließend das betreffende *Editor-Definition-Model* ausgewählt werden.

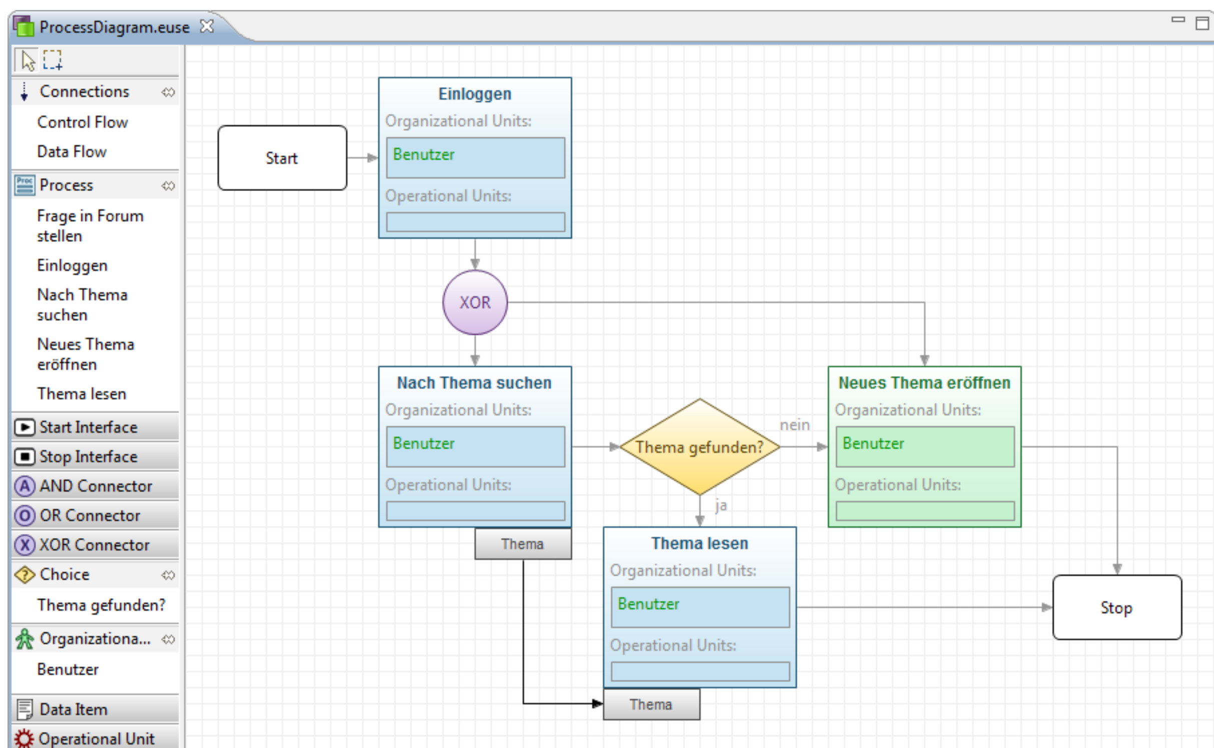

Abb. B-2: Prozess-Designer mit Beispielprozess "Frage in Forum stellen"

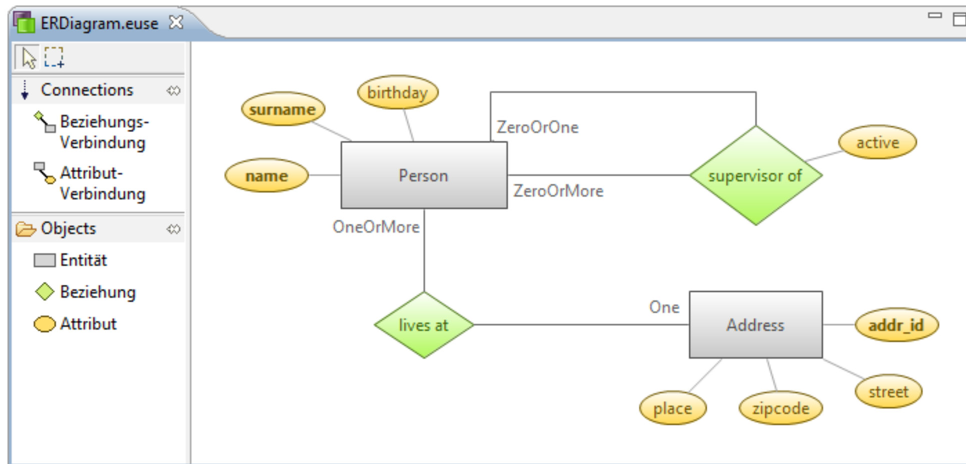

Abb. B-3: Enity-Relationship-Designer mit ER-Schema einer Personalverwaltung

Möchte man das Model Designer Framework auf einem alternativen Betriebssystem einsetzen, so muss zunächst die dafür benötigte Eclipse-Distribution (Version 3.6) zusammen mit Xtext 1.0 [14] installiert werden. Anschließend sind alle JARs aus dem *plugins*-Ordner in das gleichnamige Verzeichnis der Eclipse-Installation zu kopieren. Der Workspace aus dem *release*-Ordner kann dann in analoger Weise ausgewählt werden. Damit steht die komplette Funktionalität von i>PM² auch auf anderen Plattformen zur Verfügung.

Im *development*-Verzeichnis liegt ein voll funktionsfähiges Eclipse (Distribution für Windows 32Bit) in der Version 3.6 mit installierten oMME-Plug-Ins (ohne MDF). Die MDF-Plug-Ins befinden sich als Quellcode im *workspace*-Ordner, der nach dem Start des zugehörigen Eclipse auszuwählen ist. Auf diesem Weg kann der Kern des Model Designer Frameworks weiterentwickelt und direkt getestet werden. Dazu existiert ein zusätzlicher Workspace – nämlich *workspace-test* – mit dem in Kapitel 7 beschriebenen Prozess-Designer. Dieser Workspace muss in der zum Testen verwendeten Laufzeitumgebung von Eclipse selektiert werden.