Konzepte graphischer Frameworks und ihre Anwendung auf einen UML Editor

Diplomarbeit

Lehrstuhl Praktische Informatik III

Fachbereich Informatik

Software Engineering

FernUniversität in Hagen

von

Jens von Pilgrim

Betreuer:

Prof. Dr. Hans-Werner Six

Dr. Henrik Behrens

Dipl. Ing. Matthias Then

Berlin, März 2005

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Wörtlich oder inhaltlich übernommene Literaturquellen wurden besonders gekennzeichnet.

Berlin, 15. März 2005

Inhaltsverzeichnis

1 Einführung

Die an der Fernuniversität Hagen angebotenen Kurse im Bereich Software Engineering enthalten Einsendeaufgaben, welche die Erstellung von Diagrammen in der „Unified Modeling Language“, kurz UML, in der Lösung erfordern. Bisher wurden diese Lösungen in schriftlicher Form von den Kursteilnehmern eingeschickt. Sollen diese Lösungen zukünftig über das Internet eingereicht werden, so müssen die UML-Diagramme in elektronischer Form vorliegen.

Zur Erstellung der Diagramme können graphische Editoren eingesetzt werden. Dabei können zwei Arten graphischer Editoren eingesetzt werden. So genannte Multi-Domänen-Editoren wie Microsoft Visio[1] oder OmniGraffle[2] ermöglichen die Erstellung von beliebigen Diagrammtypen. Domänenspezifische Editoren sind auf die Erstellung bestimmter Diagrammtypen vorbereitet. Im Bereich UML stehen dafür Modellierungstools wie IBM Rational Modeler[3] oder Gentlewares Poseidon[4] bereit.

Nachteil genannter Anwendungen ist die Bereitstellung nicht benötigter Funktionen, welche die Kursteilnehmer von der eigentlichen Aufgabe – der Erlernung von UML und Techniken des Software-Engineerings – ablenken. Auch ist gerade beim Einsatz vorhandener UML-Editoren nicht sicher gestellt, dass die Anforderungen des Kurses durch die Funktionalität der Anwendung abgedeckt ist. Beispielsweise ist die Umsetzung bestimmter Notationen in der UML nicht eindeutig geregelt oder optional.

---

[1] URL: http://www.microsoft.com/germany/office/spezielle_anwendungen/visio2003/default.mspx (Stand: 8.3.2005)

[2] URL: http://www.omnigroup.com/applications/omnigraffle/ (Stand : 8.3.2005). Mit diesem Editor wurden einige im Rahmen dieser Arbeit verwendeten Abbildungen und UML-Diagramme erstellt.

[3] URL: http://www-306.ibm.com/software/awdtools/modeler/swmodeler/index.html (Stand: 8.3.2005)

[4] URL: http://gentleware.com/ (Stand: 8.3.2005)

Aus diesen Überlegungen heraus bietet sich die Erstellung eines eigenen Editors an, der auf die Bedürfnisse der jeweiligen Kurse maßgeschneidert werden kann. Als Grundlage der mit dieser Arbeit vorgestellten Eigenentwicklung wird ein graphisches Framework eingesetzt. Da die realisierbaren Fähigkeiten und die weitere Entwicklung maßgeblich von dem eingesetzten Framework abhängen, stellt die Analyse vorhandener Frameworks einen Schwerpunkt dieser Arbeit dar.

Die Arbeit gliedert sich in drei Teile. Im ersten Teil werden graphische Editoren als Anwendungsbereich untersucht. Dabei werden die wesentlichen Konzepte und Begrifflichkeiten als Grundlage der folgenden Abschnitte entwickelt.

Im zweiten Teil werden drei bekannte Frameworks für graphische Editoren (JHotDraw, Tigris GEF und Eclipse GEF) beschrieben und analysiert. Anhand des Einsatzes und der Umsetzung von Entwurfsmustern werden die Gemeinsamkeiten und Unterschiede der Frameworks im Vergleich diskutiert.

Im dritten Teil dieser Arbeit wird die Umsetzung eines graphischen Editors für UML-Diagramme auf Basis eines der vorgestellten Frameworks (Eclipse GEF) vorgestellt. Abweichend von einem klassischen Entwurf werden hier nicht eigene Klassen modelliert, sondern bereits existierende Lösungen in Form weiterer Frameworks vorgestellt. Der eigentliche Entwurf besteht dann aus der Modellierung des Zusammenspiels aller Frameworks.

2 Konzepte graphischer Editoren

Graphische Editoren sind eine klassischer Anwendungsbereiche des Software-Engineerings. Viele heute noch relevante Konzepte sind hier bereits in 60er Jahren entworfen worden. Im Rahmen späterer Arbeiten über graphische Editoren aus den frühen 90er Jahren wurden nicht nur Konzepte für den Anwendungsbereich entwickelt, sondern auch Techniken des Software-Engineerings allgemein. So ist etwa die Entwicklung von Entwurfsmustern eng an diesen Bereich geknüpft.

Sketchpad, Unidraw und HotDraw können als Meilensteine der Entwicklung graphischer Editoren bezeichnet werden. Sketchpad, 1963 von Ivan Sutherland entwickelt ([Sutherland63]), ist einer der ersten graphischen Editoren überhaupt und somit Ausgangspunkt für die spätere Entwicklung von graphischen Editoren. Zu den entscheidenden Konzepten dieses Editors gehört es, dass hier Zeichnungen als eine Menge von Elementen und deren Beziehungen zueinander beschrieben werden. Der Benutzer von Sketchpad konnte die Zeichnung mittels eines Lichtgriffels bearbeiten (die Maus war noch nicht erfunden).

Ein wichtiges Framework in der Geschichte graphischer Editoren ist Unidraw. Es wurde 1989 entwickelt und ist in John Vlissides Dissertation von 1990 beschrieben ([Vlissides90]). In Unidraw sind viele Konzepte enthalten, die für die späteren Frameworks typisch sind. HotDraw ist eine Weiterentwicklung von Unidraw (vgl. [Brant95, 22]). Es wurde von Kent Beck und Ward Cunningham in Smalltalk entwickelt. Entscheidend an HotDraw ist, dass hier das Model-View-Controller-Muster umgesetzt und das Konzept der Graphen, also die Unterscheidung der Zeichnungselemente in Knoten und Kanten, verwendet wird.

Für den Vergleich aktueller Frameworks im dritten Abschnitt wird in den folgenden Unterabschnitten eine einheitliche Begrifflichkeit für die Diskussion von graphischen Frameworks eingeführt. Auch wenn sich die Konzepte der Frameworks alle sehr ähneln, werden doch jeweils verschiedene Begrifflichkeiten verwendet. Im Anhang ist zur Übersichtlichkeit eine Tabelle aufgeführt, welche die jeweiligen, in den unterschiedlichen Frameworks verwendeten Begriffe im Vergleich aufführt (siehe Anhang, S. ix).

2.1 Components und Constraints

Graphische Objekte, beispielsweise Kreise oder Text, werden als *Components* (*Komponenten*)[5] oder *Figure* (*Figur*) [6] bezeichnet. Diese Komponenten werden hierarchisch verwaltet. Eine Komponente kann als *Container* (Behälter) für weitere Komponenten dienen. Damit ergibt sich zunächst eine baumartige Struktur der Komponenten. Je nach Position im Baum werden die Komponenten unterschiedlich bezeichnet. Die Wurzel des Komponentenbaums ist die *Zeichnung* selbst (bei HotDraw heißt sie *Drawing*). Diese ist in der Regel in einer *Pane* (*Zeichenfläche*) enthalten, die oftmals nicht zum Komponentenbaum selbst gezählt wird. Komponenten ohne Kinder werden, wie bei Bäumen üblich, als *Blätter* bezeichnet.

Neben dieser Kompositionsbeziehung können weitere Beziehungen zwischen Komponenten definiert werden. Bei Sketchpad konnten Beziehungen zwischen den Elementen vom Benutzer in Form von *Constraints* (Beschränkungen) als Abhängigkeiten der Eigenschaftsvektoren der Elemente definiert werden. In der Begrifflichkeit der objektorientierten Programmierung würde man diese Eigenschaftsvektoren als Tupel aller Felder einer Klasse bezeichnen. Hierbei handelt es

---

[5] Englische Fachbegriffe werden mit jeweils deutscher Übersetzung in Klammern eingeführt. Sofern englisches Original und Übersetzung entweder sehr ähnlich sind (wie im Fall von „Komponent“ oder „Component“) oder die Übersetzung allgemein verwendet wird, werden beide Varianten synonym verwendet. Bei der Beschreibung von Entwurfsmustern und wenn die englischen Bezeichnungen in den Klassennamen der Implementierungen verwendet werden, wird die englische Version bevorzugt. *Fachbegriffe* graphischer Frameworks und die Namen von Teilnehmern in Entwurfsmustern werden, wenn Sie in der englischer Version verwendet werden oder es der Unterscheidbarkeit dient, kursiv gesetzt. `Bezeichner` von Klassen- oder Schnittstellen werden dagegen in nicht proportionaler kursiver Schrift angezeigt. Im Anhang ist eine Übersicht der englischen und deutschen Begriffe aufgeführt.

[6] Die Bezeichnung „Figure“ bezieht sich vor allem auf den darstellenden Aspekt, also im später verwendeten MVC-Muster auf den View-Teilnehmer. Der Begriff „Komponente“, aus UniDraw übernommen, bietet den Vorteil, das MVC-Triple als Gesamtheit bezeichnen zu können.

sich um eine weitgehende Abstraktion: Unterschiedlichste Zeichnungselemente werden auf Eigenschaftsvektoren reduziert; so werden sie vergleichbar und können damit zueinander in Beziehung gesetzt werden.

In [Sutherland63] sind *Constraints* wie folgt definiert:

> "Constraint [is] a specific storage representation of a relationship between variables which limits the freedom of the variables, i.e., reduces the number of degrees of freedom of the system." [Sutherland63, 141]

Übertragen auf den, von den diskutierten Frameworks verwendeten, objektorientierten Ansatz heißt das, dass die Werte der Attribute einer Klasse von den Attributen einer anderen Klasse abhängen.

Die folgenden Abbildungen zeigen die Beziehungen der Komponenten. Das erste Diagramm zeigt die Zeichnung, wie sie vom Benutzer erstellt wurde und für diesen sichtbar ist. Das zweite Diagramm illustriert die Komponentenhierarchie. Die Komponenten sind hier entsprechend der Baumhierarchie angeordnet – je nach Umsetzung kann diese Hierarchie natürlich etwas anders aussehen.

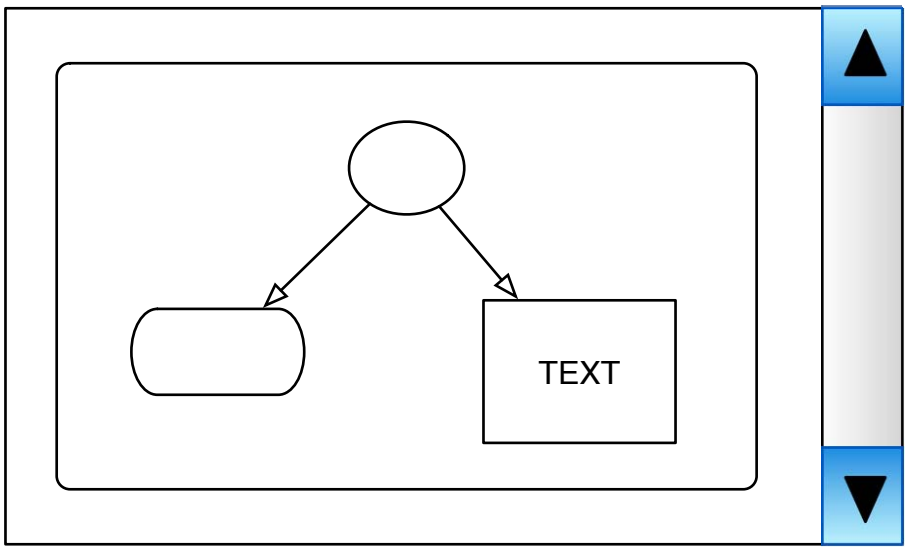

Abbildung 1: Zeichnung und Komponenten

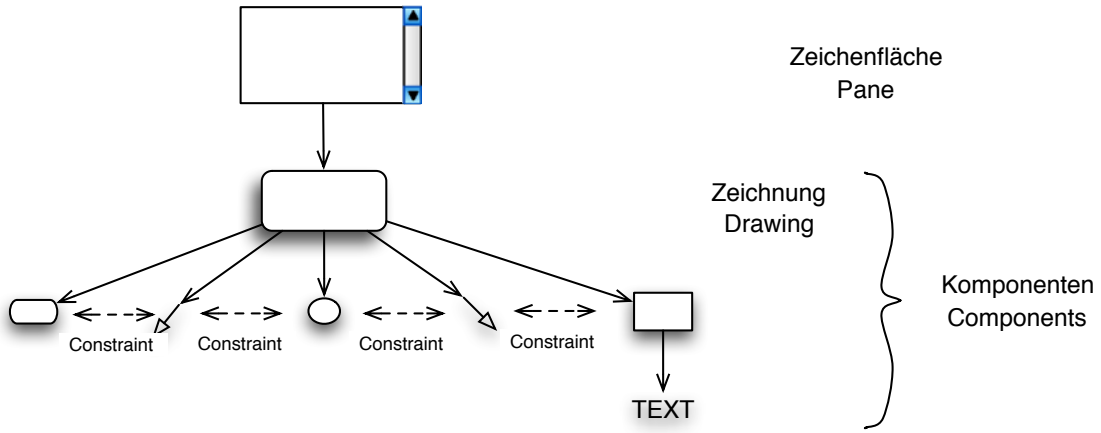

Abbildung 2: Baumhierarchie der Komponenten

Ein typisches Beispiel für *Constraints* ist die Anordnung von Komponenten. Das folgende *Constraint* würde etwa die Komponenten K_0 bis K_n untereinander anordnen:

$$K_{i+1}.top = K_i.bottom+1 \text{ für } 0 \leq i < n, \text{ top und bottom sind}$$

Eigenschaften der Komponenten

Ein anderes Beispiel aus [Helm+92, 3] ist die Verbindung von Komponenten, etwa von Knoten und Kanten: Seien A und C Knoten, repräsentiert als Rechtecke, und B eine Kante, repräsentiert als Linie, die A und C miteinander verbindet, so stellt folgendes *Constraint* sicher, dass die Linie mit beiden Rechtecken verbunden bleibt:

$$A.left \leq B.left \leq A.right,\ A.bottom \leq B.top \leq A.top,$$

$$C.left \leq B.right \leq C.right,\ C.bottom \leq B.bottom \leq C.top$$

Zur Veranschaulichung ist dieser Sachverhalt in folgender Abbildung noch einmal dargestellt.

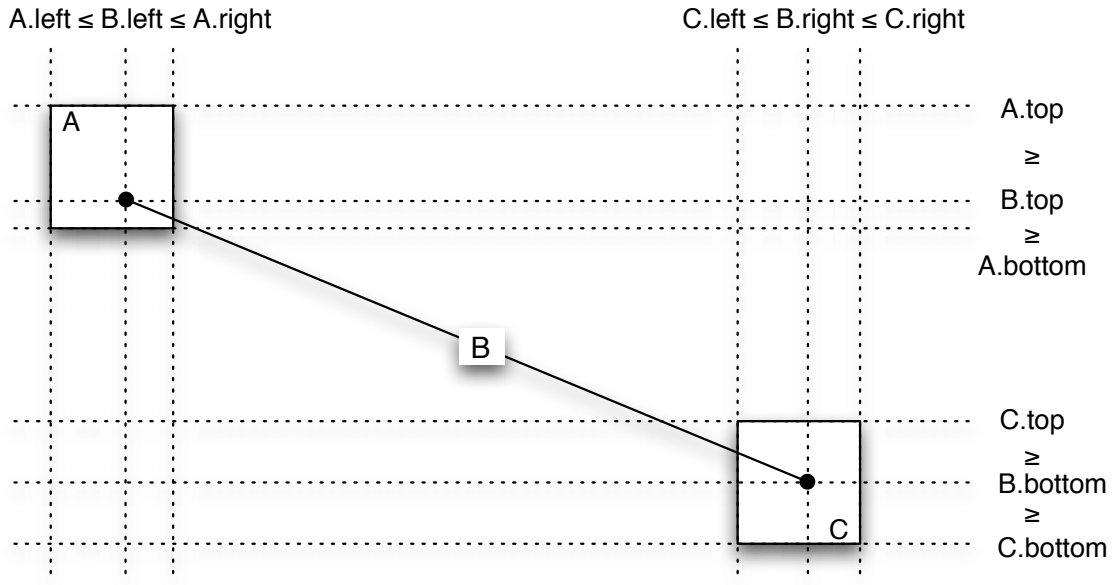

Abbildung 3: Beispiel für ein Constraint

Soll die Linie B die Rechtecke jeweils an den Rändern berühren, so müssen weitere Bedingungen hinzugefügt werden, etwa:

min(|B.left - B.right|), min(|B.top - B.bottom|)

Constraints sind also immer Beziehungen bzw. Relationen zwischen zwei oder mehr Komponenten, wobei das Verhalten mindestens einer Komponente, sprich ihre Eigenschaften, über das *Constraint* eingeschränkt wird.

In der Praxis treten allerdings Probleme beim Auflösen dieser Abhängigkeiten auf, etwa bei der Performanz. Beispielsweise wären die letzten Bedingungen im obigen Beispiel (min(..)) Optimierungsprobleme, die aufwendiger zu lösen sind. Daher sind *Constraints* in der Regel nicht in allgemeiner Form implementiert, sondern mittels eigener, Konzepte, die in den folgenden Abschnittel vorgestellt werden.

2.2 Node, Connection und Anchor

Eine bekannte Beziehung zwischen Komponenten ist die zwischen Knoten und Kanten in Graphen. Alle Frameworks kennen entsprechende Spezialisierungen der allgemeinen Komponente. Diese werden *Nodes* (*Knoten*) und *Connections* (*Verbindung*), letztere auch *Edges* (*Kanten*), genannt. Die Verbindung von *Node* und *Connection* geschieht dabei über spezielle Elemente: *Anchors* (*Anker*). *Diese* verhalten sich je nach Umsetzung leicht unterschiedlich. Sie sind quasi die Endpunkte der Kante und werden meistens vom Knoten initialisiert. Sie werden auch als *Connection Point*s, *Connectors*,

oder *Ports* bezeichnet. Außerdem können sie als Umsetzung der oben beschriebenen *Constraints* aufgefasst werden. Das *Constraint* legt die Eigenschaften der Kante fest, nämlich die Position ihrer Endpunkte in Abhängigkeit von der Lage zweier Knoten.

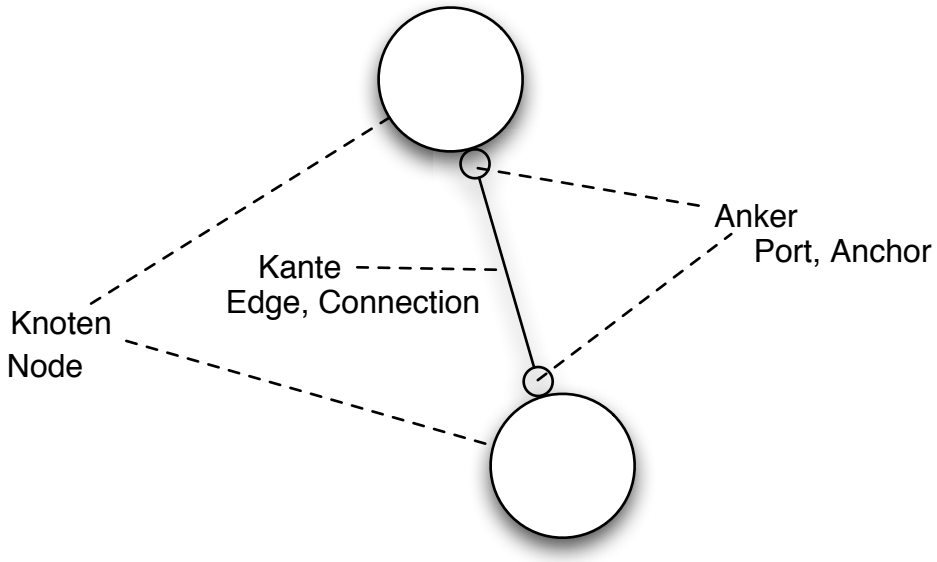

Abbildung 4: Knoten, Kante und Anker

Kanten sind hier fast immer gerichtete Kanten, das heißt sie haben einen Start- und einen Endknoten (*source node*, *target node*). Start- und Endknoten können dabei identisch sein, die Kante ist in diesem Fall eine Schleife. Falls einer der beiden Knoten entfernt wird, muss im Allgemeinen auch die Kante entfernt werden. Falls ein Knoten im Diagramm verschoben wird, muss die Lage der Kante angepasst werden. Zu beachten ist, dass Kanten ebenfalls Knoten sein können, dass also eine Kante zwei andere Kanten verbinden kann.

2.3 Locator, Layout und Grid

Eine weiteres typisches *Constraint* wird über *Locators* (*Lokalisierer*) umgesetzt. *Locators* werden beispielsweise für Beschriftungen von Kanten eingesetzt, deren Position relativ zur Lage der Kante gesetzt wird. Wird in einem solchen Fall die Kante verschoben, sorgt der *Locator* dafür, dass die Beschriftung ebenfalls verschoben wird. Ein einfacher *Locator* für die Beschriftung einer Kante würde vielleicht die Beschriftung immer 10 Punkte oberhalb des Mittelpunkts der Kante positionieren.

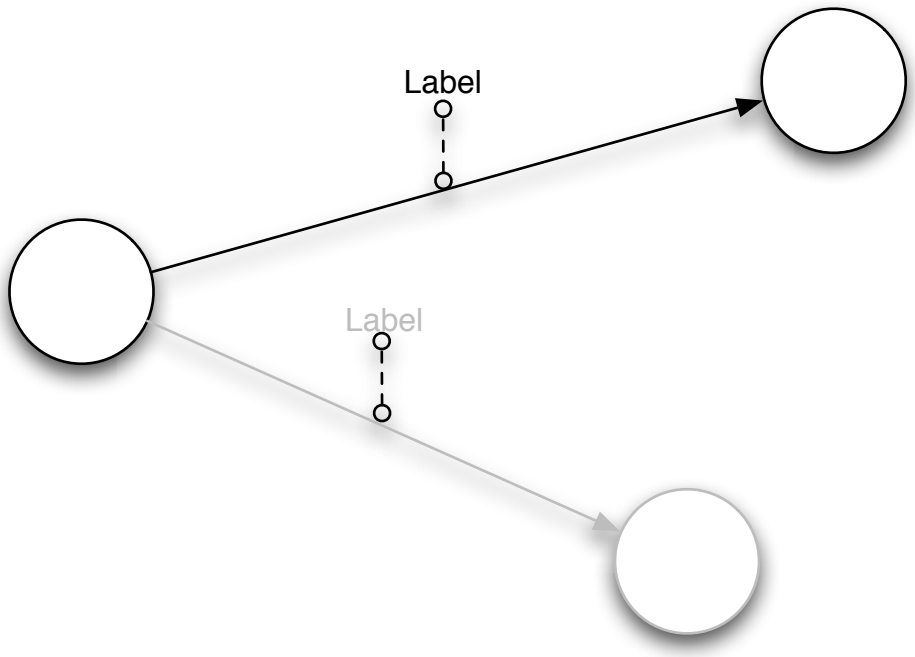

Abbildung 5: Locator

Während *Locators* allgemein die Position einer Komponente in Abhängigkeit zu einer anderen Komponente berechnen, berechnen *Layouts* die Position der Kinderelemente einer Komponente. *Layouts* sind bereits aus Bibliotheken für graphische Benutzeroberflächen wie Java-Swing bekannt. Häufig ordnen sie die Kinderkomponenten tabellarisch in Zeilen- und Spalten an. Diese *Layouts* heißen *Flow-Layout* (wenn nur eine Zeile oder Spalte verwendet wird) oder *Grid-Layout* (wenn alle Zellen der Tabelle die gleiche Größe haben). Ein bei graphischen Editoren häufig eingesetztes *Layout* ist das *XY-Layout*. Hier werden die Kinderelemente einfach an definierten X- und Y-Koordinaten relativ zur Behälterkomponente positioniert. *Layouts* benötigen zur Berechnung häufig zusätzliche Informationen, etwa die X- und Y-Koordinaten, oder Angaben zur Ausrichtung der Kinderkomponenten. Diese zusätzlichen Informationen werden, leider etwas verwirrend, ebenfalls als *Constraint* bezeichnet. *Layouts*, die diese zusätzlichen Informationen auswerten, werden (etwa bei Eclipse GEF) *Constrained Layouts* genannt.

Während *Layouts* die Position der Kinderkomponenten berechnen und selbst Eigenschaften der Behälterkomponente sind, können auch temporäre Hilfsmittel die Position der Komponenten, häufig in Kombination mit *XY-Layouts*, festlegen. Eine bekannte Art globaler *Constraints* stellen *Grids* (*Raster*) dar. Hierbei handelt es sich um sichtbare oder unsichtbare Gitter, welche die Koordinaten von Komponenten ausrichten

– beim Zeichnen rasten die Elemente in das Gitter ein („snap to grid“). Neben diesen, in jeweils festen Abständen definierten Rasterlinien können weitere Hilfslinien, allgemein *Guides*, zur Erleichterung der Positionierung von Komponenten Verwendung finden, etwa um Elemente mittig untereinander darzustellen. Oft werden diese Art der *Constraints* mit speziellen Befehlen ergänzt, die vorhandene Komponenten im nachhinein entsprechend ausrichten. Diese heißen dann „am Raster ausrichten“ oder „horizontal anordnen“. *Grids* und andere Hilfslinien dienen nur der Positionierung bei der Eingabe und werden nicht mit den Komponenten gespeichert.

2.4 Eigenschaften der Komponenten

Knoten sind die zentralen Komponenten von Diagrammen. Allgemeingültige Aussagen über Knoten sind schwer zu treffen, da ihre Eigenschaften von dem jeweiligen Diagrammtyp abhängig sind. Meistens werden sie als Rechtecke oder Ellipsen gezeichnet und enthalten weitere Figuren oder Text.

Connections können unterschiedliche Farben oder Linienstile aufweisen. Außerdem sind die Enden der Linien häufig semantisch bedeutsam, was über *Decorations* (*Dekorationen*) markiert wird. Neben diesen anwendungsspezifischen Eigenschaften haben *Connections* in Diagrammen unterschiedliche Formen: einfache Linien, *Polylinien* oder *Bezierkurven*. *Polylinien* sind aus mehreren Liniensegmenten bestehende Kanten. Die einzelnen Punkte einer *Polylinie* werden als *bendpoint* (auch *waypoint – Wegpunkte*) bezeichnet.

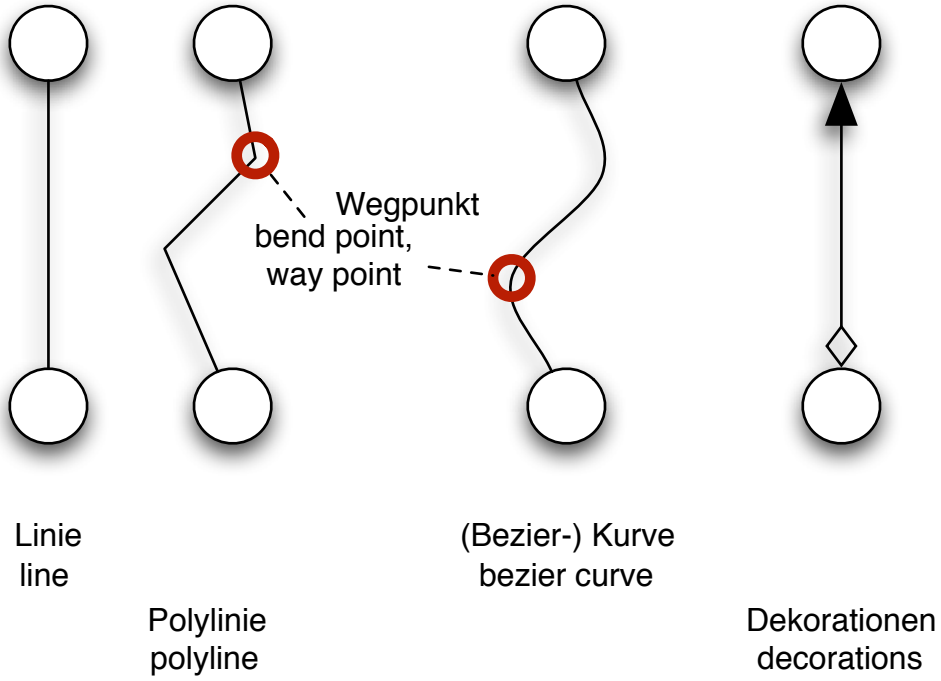

Abbildung 6: Connections

Die Anordnung der *Bendpoints* kann manuell oder automatisch erfolgen. Die automatische Berechnung der Positionen von *Bendpoints* wird über so genannte *Router* durchgeführt. Die folgende Abbildung zeigt verschiedene *Routertypen*. Der *Manhatten Router* zählt zu den gängigsten Typen. Er erzeugt eine Polylinie mit jeweils rechten Winkeln. Interessant ist auch der in Eclipse GEF enthaltene *Fanrouter*, der mehrere Polylinien zwischen zwei Knoten auffächern kann.

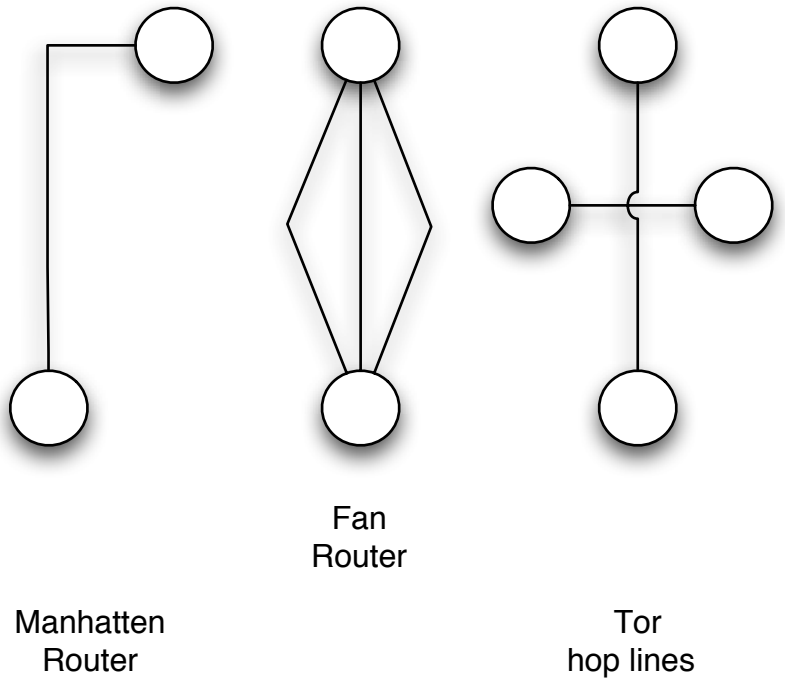

Abbildung 7: Router

Ein leider von den drei Frameworks nicht unterstütztes Feature sind *Tore* oder *hop lines*, also kleine Bögen, die Überschneidungen von Kanten visualisieren.

2.5 Benutzerinteraktion

Als graphische Benutzerschnittstellen bieten graphische Editoren die üblichen Interaktionselemente wie *Scrollbars* oder *Buttons*.

Toolbars, wie man sie heute in vielen Programmen mit graphischer Benutzerschnittstelle findet, werden in graphischen Editoren als *Palettes* (*Paletten*) bezeichnet. Auf einer *Palette* werden Symbole der angebotenen *Tools*(*Werkzeuge)* angezeigt. Diese *Tools* legen fest, wie die Benutzeraktionen, meistens Eingaben über die Maus, zu interpretieren sind. Für das Framework stellen *Tools* häufig Zustände des graphischen Editors dar. Typische *Tools* sind Selektion, Vergrößern und Verkleinern sowie Erzeugen von Knoten und Kanten.

Zur Bearbeitung der Zeichnung bzw. der Komponenten können die einzelnen Komponenten mit dem *Tool* „Selektion“ markiert werden. Anders ausgedrückt: Wenn der Editor sich im Zustand „Selektion“ befindet, werden Benutzereingaben so interpretiert, dass Komponenten entsprechend der Eingaben selektiert werden. Die Auswahl wird dem Benutzer häufig über Rechtecke aus gestrichelten Linien angezeigt. Selektierte Komponenten können über so genannte *Handles* manipuliert, also etwa in

der Größe verändert („resize“) oder verschoben werden. Dies wird optisch über passende Mauszeiger dargestellt. Intern verwenden die Frameworks zur Manipulation der Komponenten *Commands (Befehle, auch Actions - Aktionen*) analog dem Befehlsmuster.

Die folgende Abbildung fasst die typischen Elemente der Benutzerschnittstelle zusammen.

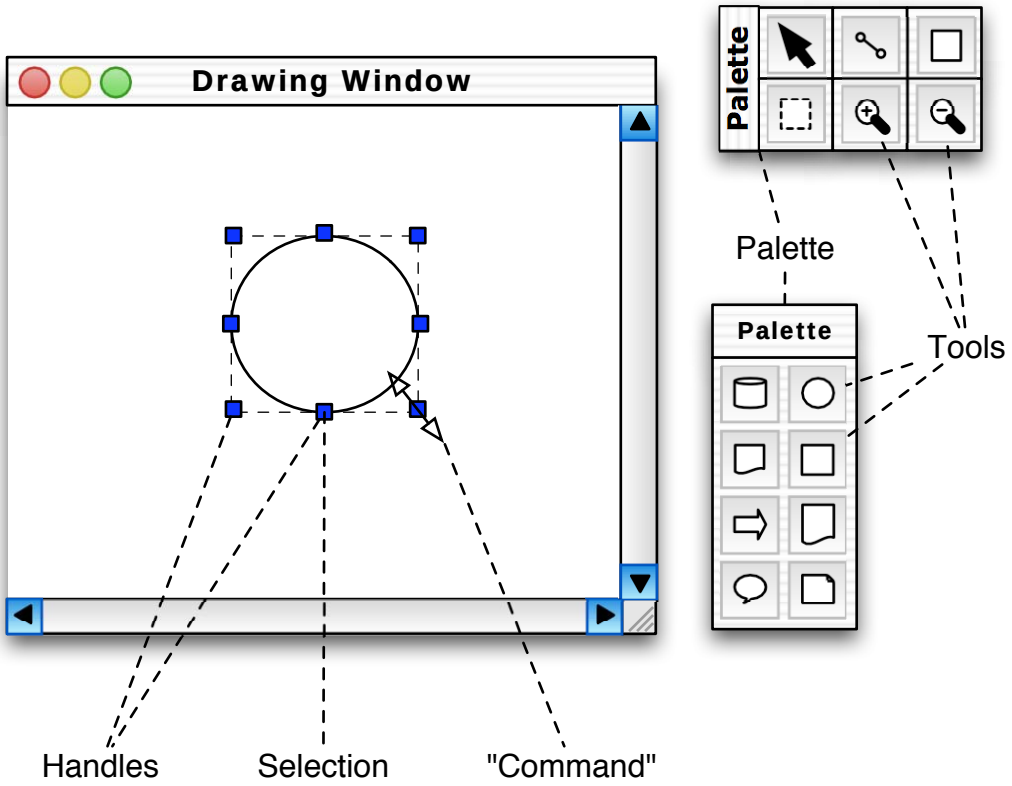

Abbildung 8: Benutzerschnittstelle graphischer Editoren

2.6 Layers

Eine besondere Komponente stellen *Layers* (*Ebenen*) dar. Unidraw kannte *Layers* noch nicht, und auch die ersten Versionen von HotDraw kamen ohne dieses Konzept der Ebenen aus. Mittlerweile sind *Layers* allerdings als fester Bestandteil in allen Frameworks enthalten. Prinzipiell können *Layers* als Komponenten interpretiert werden, die selbst im Allgemeinen nicht sichtbar sind und nur der Strukturierung der Komponenten dienen.

Layers werden übereinander gestapelt, d.h. sie besitzen eine Z-Order. Neben benutzerdefinierten *Layer*, die der Strukturierung der Zeichnung dienen, werden *Layers*

von den Frameworks auch intern eingesetzt. So werden von Tigris GEF oder Eclipse GEF die oben eingeführten *Handles* etwa in einen eigenen *Layer* gezeichnet, der über allen anderen *Layer* liegt. Für Benutzer und Entwickler sind diese technischen *Layer* sowohl in der Darstellung wie in der Programmierung transparent. Eclipse GEF verwendet einen weiteren *Layer* für *Connections*, so dass alle Kanten über den Knotenkomponenten liegen. Diese technischen *Layer* können vom Programmierer entsprechend konfiguriert werden. Praktisch sind *Layers* auch für den Ausdruck von Zeichnungen bzw. deren Export – so sind in Eclipse GEF *Printable Layers* definiert, die dann entsprechend berücksichtigt werden. Der *Layer* für die *Handles* kann auf diese Weise beim Ausdruck unterdrückt werden, da dieser nicht als *Printable Layer* gekennzeichnet ist.

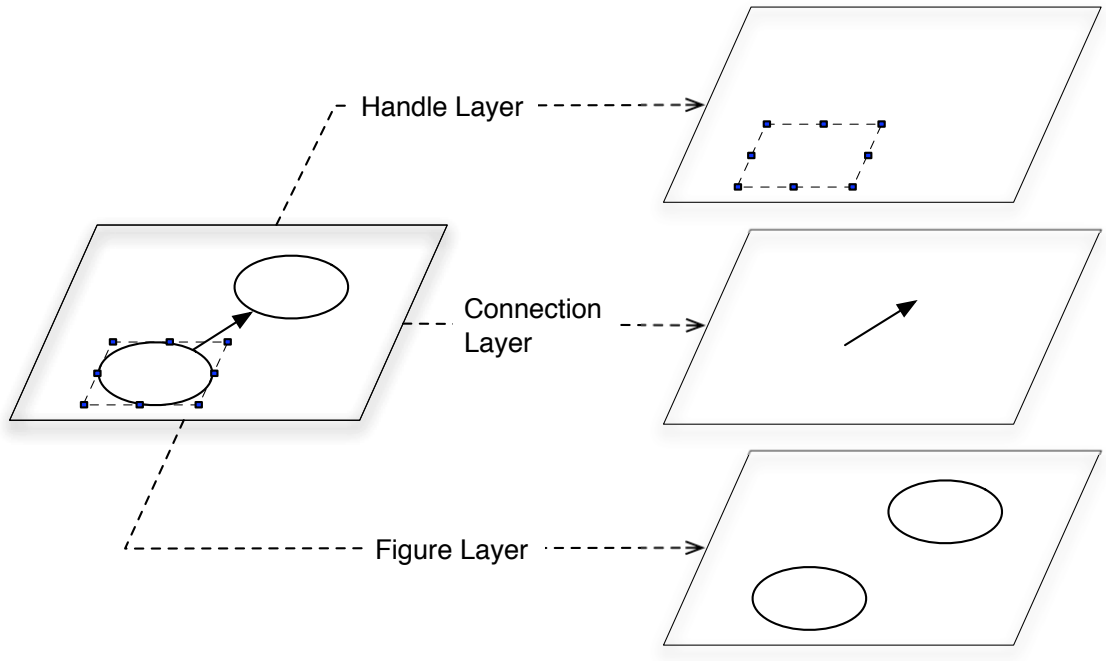

Abbildung 9: Layers

3 Vergleich graphischer Frameworks

Im Folgenden sollen die bereits erwähnten Frameworks im einzelnen vorgestellt und ihre Funktionsweise beschrieben werden. Folgende Versionen wurden dazu herangezogen:

- JHotDraw, Version 5.2 [JHotDraw]
- Tigris GEF, Version 0.10.11 [TigrisGEF]
- Eclipse GEF, Version 3.0.1 [EclipseGEF]

Alle drei Frameworks sind frei verfügbar und können von den (im Anhang) angegebenen URLs geladen werden. Entscheidend für die Vorauswahl war, dass alle Frameworks in Java implementiert sind und daher keine Unterschiede aufgrund unterschiedlicher Programmiersprachen auftreten und ein plattformübergreifender Einsatz gewährleistet ist.

JHotDraw wurde ausgewählt, um einen Vertreter der klassischen Frameworks in den Vergleich aufzunehmen. Es ist eine Java-Version dieses eigentlich in Smalltalk entwickelten HotDraws und wurde unter anderem von Erich Gamma, einem der Autoren des legendären Buches „Entwurfsmuster“ ([Gamma+96]), für die Lehre entwickelt. JHotDraw ist sehr ausführlich mittels Entwurfsmustern dokumentiert und daher nicht nur als graphisches Framework, sondern auch als Beispiel einer Dokumentation von Frameworks interessant.

Tigris GEF ist das von ArgoUML[7] verwendete graphische Framework. Bei ArgoUML handelt es sich um einen frei verfügbaren UML-Editor, der auch in einer kommerziellen Variante, Poseidon von Gentleware[8], vorliegt. Dadurch ist die Praxistauglichkeit von Tigris GEF eindrucksvoll unter Beweis gestellt, weswegen es in den Vergleich aufgenommen wurde.

Das dritte Framework, Eclipse GEF, ist ein von IBM entwickeltes Framework, das nun im Rahmen des Eclipse-Projekts weiterentwickelt wird. Es ist im Vergleich zu den

---

[7] URL: http://www.tigris.org/argo (Stand: 5.3.2005)

[8] URL: http://www.gentleware.com (Stand: 8.3.2005). Mit diesem Editor wurden die meisten im Rahmen dieser Arbeit verwendeten UML-Diagramme erstellt.

anderen beiden Frameworks das am weitesten entwickelte Framework. Eclipse GEF setzt die bereits in den anderen Frameworks verwendeten Ansätze am konsequentesten um und bietet den größten Funktionsumfang – entsprechend schwierig ist es allerdings auch zu erlernen. Eingesetzt wird Eclipse GEF beispielsweise in EclipseUML, einem UML-Editor und Eclipse Plug-In von Omondo[9]. Im Gegensatz zu JHotDraw und Tigris GEF, die beide auf der Java-Swing-Bibliothek aufsetzen, verwendet es SWT[10] als Widgetbibliothek – aus Sicht des Entwicklers macht dies allerdings keinen großen Unterschied.

3.1 Vorstellung der Frameworks

Vor der eigentlichen Analyse der Frameworks soll hier zunächst auf den Begriff des Frameworks eingegangen werden. Eine Annäherung an diesen Begriff findet sich bei [Johnson97]:

> "One common definition is 'a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact'. Another common definition is "a framework is the skeleton of an application that can be customized by an application developer" [Johnson97, 1]

Die hier angesprochenen Eigenschaften sind vor allem in Hinblick auf ihre Konsequenzen interessant. Im Gegensatz zu Klassenbibliotheken legen Frameworks die Architektur der Anwendung fest. Die Anwendung, der Client, besteht aus Client-spezifischen Klassen und Klassen des Frameworks. Die Client-spezifischen Klassen sind entweder Ableitungen von (abstrakten) Klassen bzw. Implementierungen von Interfaces des Frameworks oder sie müssen sich an konkrete Klassen des Frameworks geeignet anmelden bzw. diese intanziieren. Das Framework bestimmt also die Zerlegung der Anwendung und die Aufteilung der Komponenten, da, anders als bei Klassenbibliotheken, die Klassen des Frameworks nicht einfach vom Client verwendet werden, sondern vielmehr umgekehrt, das Framework die Klassen des Clients verwendet.

---

[9] URL: http://www.omondo.com (Stand: 8.3.2005)

[10] URL: http://www.eclipse.org/swt (Stand: 8.3.2005)

Aufgrund dieser in Frameworks enthaltenen und für die Anwendung relevanten Beziehungen zwischen den Klassen können Frameworks gut mittels Entwurfsmustern beschrieben werden:

> "Since frameworks are reusable designs, not just code, they are more abstract than most software, which makes documenting them difficult. [...] The principal audience of framework documentation is someone who wants to use the framework to solve typical problems, not someone building a software cathedral. Patterns seem to be well suited for this audience." [Johnson92, 1]

Daher werden die Frameworks im folgenden anhand von Entwurfsmustern diskutiert. Zum Teil konnte hierbei auf Verweise auf die jeweils angewandten Entwurfsmuster in der Literatur zurückgegriffen werden. Häufig mussten die Muster jedoch aus dem vorliegenden Quellcode herausgefiltert werden. Zusätzlich wurden die konkreten Umsetzungen der Entwurfsmuster mit deren Beschreibungen, etwa in [Gamma+96], verglichen. Dabei werden in dieser Arbeit Muster, die in [Gamma+96] vorgestellt werden, grundsätzlich als bekannt vorausgesetzt.

Um Entwurfsmuster in den Frameworks zu erkennen und zu beschreiben werden UML-Diagramme eingesetzt. Auch hier konnten in manchen Fällen Diagramme aus der vorhandenen Literatur als Grundlage dienen. Im Wesentlichen wurden die in dieser Arbeit erstellten Diagramme jedoch im Rahmen der eigenen Analyse angefertigt (Reverse-Engineering). Sie haben damit nicht nur rein deskriptiven Charakter, sondern sind auch als Ergebnisse der Analyse zu sehen.

Zur Beschreibung der Frameworks werden einheitlich jeweils drei UML-Diagramme verwendet: zwei Klassendiagramme und ein Sequenzdiagramm. Ein Klassendiagramm beschreibt die wichtigsten Klassen und Interfaces in vereinfachter Form. Zweck dieses ersten Diagramms ist es, einen Überblick des Frameworks zu erhalten und die jeweils besonderen Namen und Begrifflichkeiten, die sich in den Klassennamen widerspiegeln, vorzustellen.

Alle drei Frameworks sind White-Box-Frameworks: Erst durch die Instanziierung, also durch Ableitung spezifischer Klassen des Frameworks bzw. durch Implementierung entsprechender Interfaces durch Client-spezifische Klassen, wird aus dem Framework eine Anwendung. Anders ausgedrückt:

> "A framework is instantiated when a software engineer provides the concrete components that the framework expects." [Carrington+04, 185]

Zur Veranschaulichung der Instanziierung wird daher ein zweites Klassendiagramm verwendet, das sowohl Teile des ersten Diagramms als auch Client-spezifische Klassen enthält. Als konkretes Beispiel werden dabei ausschnittsweise Klassen verwendet, die zur Implementierung eines einfachen Editors für mathematische Graphen mit Knoten (*Vertices*) und Kanten (*Edges*) dienen könnten. Die Instanziierung eines Frameworks ist dabei nicht mit der Instanziierung von Klassen, also der Konstruktion von Objekten, zu verwechseln. Daher wurde auch in diesem Fall statt eines Objektdiagramms ein Klassendiagramm verwendet.

Mittels Sequenzdiagrammen wird schließlich das Zusammenspiel der einzelnen Teile des Frameworks gezeigt. Dazu wird immer der gleiche Anwendungsfall, das Zeichnen eines Knotens, verwendet. Dieser Anwendungsfall hat den Vorteil, dass er einen entscheidenden Prozess, die Erzeugung von Objekten, veranschaulicht.

Anmerkung zu den Diagrammen und zur Beschreibung der Diagramme:

Die abgebildeten Klassendiagramme verwenden UML Notation (vgl. [UML2.0S]). Da sie der Erklärung des Aufbaus und der Funktionsweise der Frameworks dienen, handelt es sich hierbei nicht um exakte Abbildungen der Java-Klassen. Zur Vereinfachung wurden Attribute und Operationen der Klassen und Interfaces zumeist ausgeblendet. Assoziationen sind so einfach wie möglich dargestellt, so sind etwa Kardinalitäten von „1“ und „0..1“ nicht extra ausgewiesen. Wenn die Rollen der Klassen in Assoziationen den Klassennamen entsprechen, werden diese ebenfalls unterdrückt.

Um eine Übersichtlichkeit der Diagramme zu gewährleisten, werden Ableitungshierarchien zum Teil verkürzt dargestellt; auch einige Assoziationen werden weggelassen, insbesondere wenn es sich um abgeleitete Eigenschaften handelt.

Zur Bezeichnung der Elemente werden die Begriffe mit ihrer Semantik aus der UML verwendet. Anstelle von *Classifier* als Oberbegriff von Klasse und Schnittstelle oder dem Ausdruck „Instanzen der Klasse oder Schnittstelle“ wird hier zumeist der Begriff Klasse verwendet, es sei denn, die Unterscheidung der Konzepte ist relevant. Die vom Entwickler eines graphischen Editors zu implementierenden Klassen und Schnittstellen

werden als Client-Klassen und -schnittstellen bezeichnet. Die Frameworks stellen aus dieser Sicht die Dienstanbieter (*Supplier*) dar.

Bei den Sequenzdiagrammen werden ebenfalls einige Vereinfachungen durchgeführt. Für das Verständnis unwesentliche Nachrichten und Objekte sind hier nicht dargestellt.

Der in den Beschriftungen der Schwimmbahnen (*swimlanes*) angegebene Objekttyp ist in manchen Fällen durch die Oberklasse anstelle der konkreten Klasse angegeben. Da der Instanzname hier zumeist unwesentlich ist, wird stattdessen entweder die Rolle der Instanz im jeweiligen Entwurfsmuster oder eine andere erklärende Benennung verwendet.

3.1.1 JHotDraw

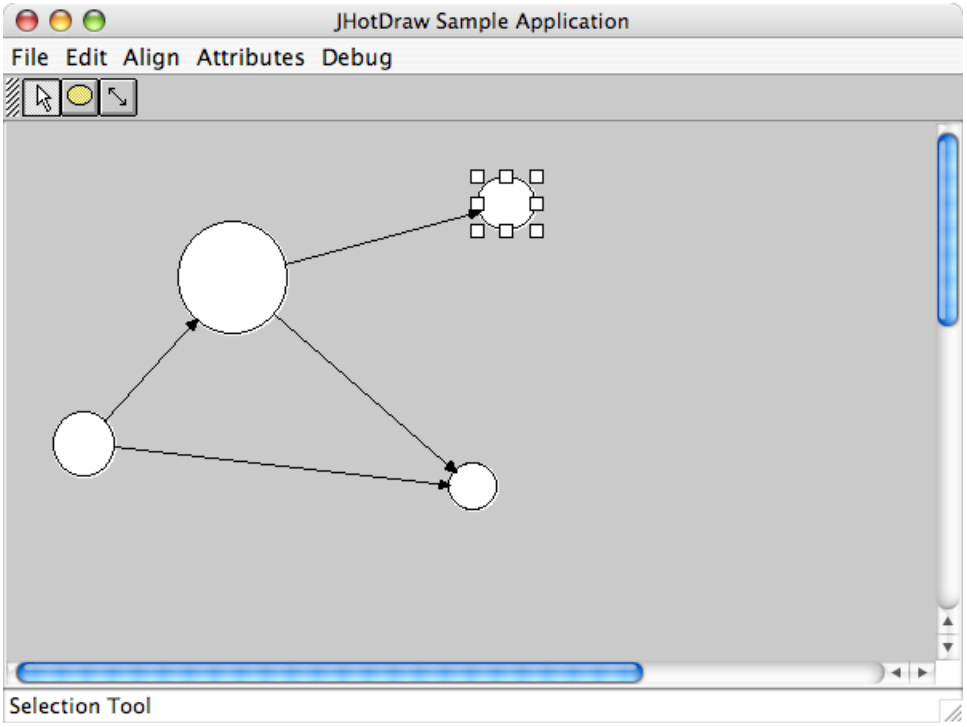

Abbildung 10: JHotDraw Beispielapplikation

JHotDraw 5.2 ist eine von Erich Gamma und Thomas Eggenschwiler entwickelte Java-Version von HotDraw. Diese Version unterscheidet sich in Teilen von der ursprünglichen Smalltalk-Version, insbesondere da hier auf den Einsatz des MVC-Patterns verzichtet wird. Ursprünglich wurde JHotDraw für die Lehre entwickelt:

> „[JHotDraw] was originally planned to be a trivial case study for an applied design patterns seminar. But then it was too much fun to just stop there...“ [JHotDraw5.2, doc/drawBackground.html).

Diese Herkunft ist dem Framework deutlich anzumerken: Fast alle Klassen sind mit Anmerkungen versehen, die erläutern, welche Rolle sie in welchem Muster einnehmen. Neben der API-Dokumentation, die Querverweise zu den umgesetzten Entwurfsmustern enthält, die sämtlichst aus [Gamma+96] stammen, behandeln weitere Artikel JHotDraw unter dem Gesichtspunkt der Entwurfsmuster, so etwa [Riehle00]. Daneben sind diverse Einführungsartikel in JHotDraw zu finden, wie [Kaiser01].

a) Struktur

Das folgende Klassendiagramm zeigt den Kern des Frameworks, der im Paket `CH.ifa.draw.framework` abgelegt ist.

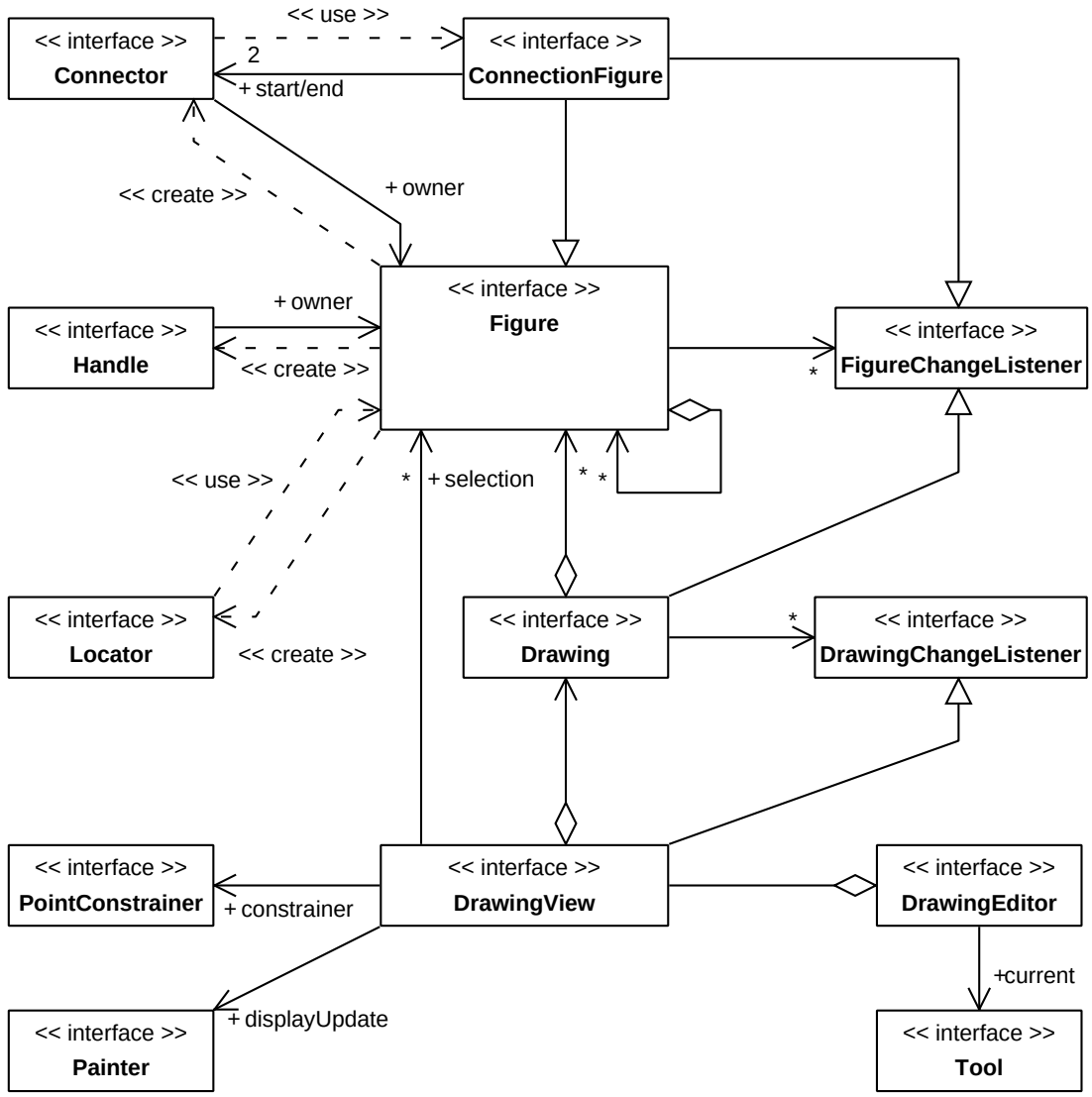

Abbildung 11: Klassendiagramm JHotDraw, vereinfachte Übersicht der JHotDraw Klassen und Interfaces

JHotDraw ist ein sehr übersichtliches Framework: Der Kern besteht lediglich aus 15 Klassen, wobei die beiden Event-Klassen für das Verständnis unwesentlich sind. Die 13 relevanten *Classifier* sind alle *Interfaces*. In der Praxis werden jeweils Standardimplementierungen verwendet, von denen im Allgemeinen auch abgeleitet wird. Allerdings ist es so möglich, über eigene Implementierungen das Framework an

Client-spezifische Klassenhierarchien anzuschließen, ohne Probleme mit Mehrfachvererbung zu bekommen.

Im Mittelpunkt steht die Klasse `Figure`, also die allgemeine Komponente, aus der die Zeichnung, hier repräsentiert durch die Klasse `Drawing`, besteht. Ein spezielles Knotenelement ist nicht definiert – alle Instanzen von `Figure` können als Knoten interpretiert werden. Kanten werden durch die von `Figure` abgeleitete Klasse `ConnectionFigure` dargestellt. Dies impliziert, dass Kanten selbst immer auch Knoten sein können. Eine Kante beobachtet dabei ihre Start- und Zielknoten, um im Falle von Änderungen entsprechende Anpassungen vornehmen zu können. Diese, im Falle von Knoten und Kanten gängige Umsetzung von *Constraints* über das Beobachtermuster wird unten (vgl. 3.2.3, S. 60) noch näher untersucht werden.

Figuren sind hier *Container* und können selbst wiederum Figuren enthalten. Die Zeichnung selbst ist in einer `DrawingView` enthalten, also der Zeichenfläche. Diese verwaltet hier auch die Auswahl (Selektion). Eine eigene Klasse hierfür ist bei JHotDraw nicht vorhanden.

Einige oben eingeführte Mechanismen zur Umsetzung von *Constraints* sind hier über das Stragegiemuster gelöst: `Connector` (Anker), `Locator` (Lokalisierer) und `PointConstrainer` (Raster). Auch der Algorithmus zum Zeichnen der Komponente auf der Zeichenfläche ist als Strategie gekapselt (`Painter`).

Obiges Diagramm unterschlägt eine, für die Programmierung von Editoren wichtige Klasse: `Command`. Diese befindet sich bei JHotDraw im `util`-Paket. Sie setzt das Befehlsmuster aus [Gamma+96] um und unterstützt kein Undo/Redo. Bei JHotDraw werden nicht alle Änderungen an den Figuren über *Commands* abgewickelt, einige Aktionen werden direkt über *Tools* ausgeführt.

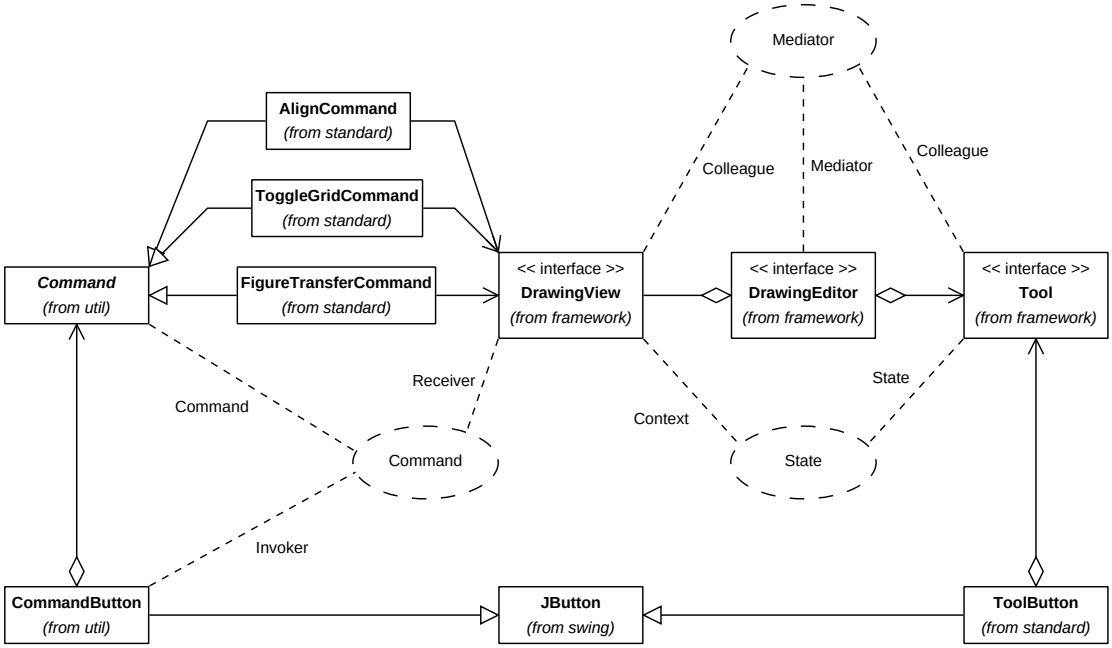

Abbildung 12: Klassendiagramm JHotDraw, Controller-Klassen

b) Instanziierung

Das folgende Diagramm zeigt eine einfache Umsetzung des Graphen-Editors. Auf eine eigene Implementierung des MVC-Musters wird in der Anwendung verzichtet. Daher finden sich hier keine speziellen Modellklassen.

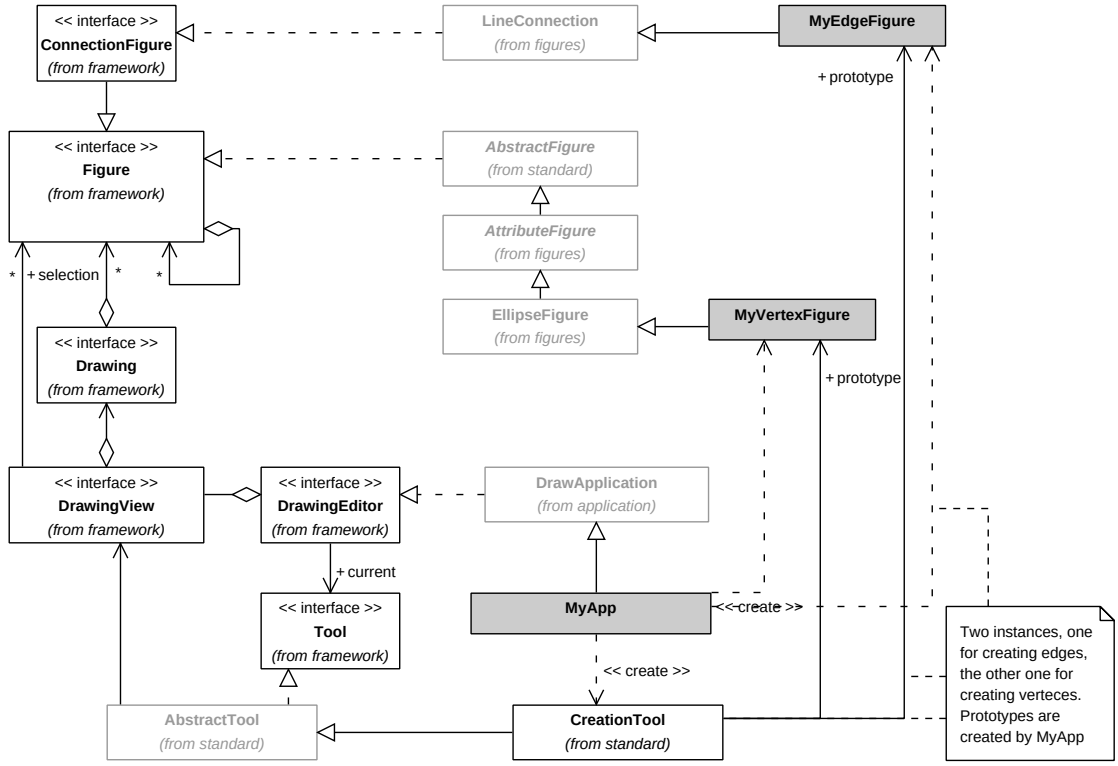

Abbildung 13: Klassendiagramm JHotDraw, Instaziierung

Bemerkenswert ist der geringe Aufwand: Es müssen lediglich drei Klassen erstellt werden. Da keine Trennung der Komponenten in *Model*, *View* und *Controller* notwendig ist, muss für die dargestellten Komponenten, hier Knoten und Kante, nur je eine Klasse implementiert werden. Daneben muss außerdem eine eigene Applikationsklasse die vorhandenen Klassen geeignet initialisieren. Die Client-spezifischen *View*-Klassen werden den Instanzen der Klasse `CreationTool` hier als Prototypen für die spätere Erzeugung übergeben. Der Code der Applikationsklasse ist so kurz, dass er hier komplett aufgeführt werden kann:

```
public class MyApp

        extends DrawApplication {
```

```
    public MyApp() {
        super("JHotDraw Sample Application");
    }

    public static void main(String[] args) {
        MyApp window = new MyApp();
        window.open();
    }

    protected void createTools(JToolBar io_palette) {
        super.createTools(io_palette);

        Tool tool;

        tool = new CreationTool(view(), new MyVertexFigure());
        io_palette.add(createToolButton(IMAGES+"ELLIPSE", "Create Vertex", tool));

        tool = new ConnectionTool(view(), new MyEdgeFigure());
        io_palette.add(createToolButton(IMAGES+"CONN", "Create Edge", tool));
    }
}
```

Codebeispiel 1: MyApp.java

c) Ereignisverarbeitung

Da das Framework in dieser Version auf der Swing-Bibliothek aufsetzt, ist die Ereignisverarbeitung wesentlich davon bestimmt. Die Ereignisse werden über das Anwendungsfenster, hier eine Instanz von *`MyApp`*, die indirekt von der Java-AWT-Klasse *`Component`* abgeleitet ist, an die *`DrawingView`* weitergeleitet. Diese reichert die Information des Ereignisses um die Koordinaten an, an denen es stattgefunden hat und leitet es dann an das aktuell aktive *Tool* weiter. Wie bereits beschrieben implementieren *Tools* das Zustandsmuster. Mit der Weiterleitung an das *Tool* wird das Ereignis also abhängig vom Zustand weiterverarbeitet.

Prototypisch wird im Folgenden die Erstellung eines neuen Knotens verfolgt:

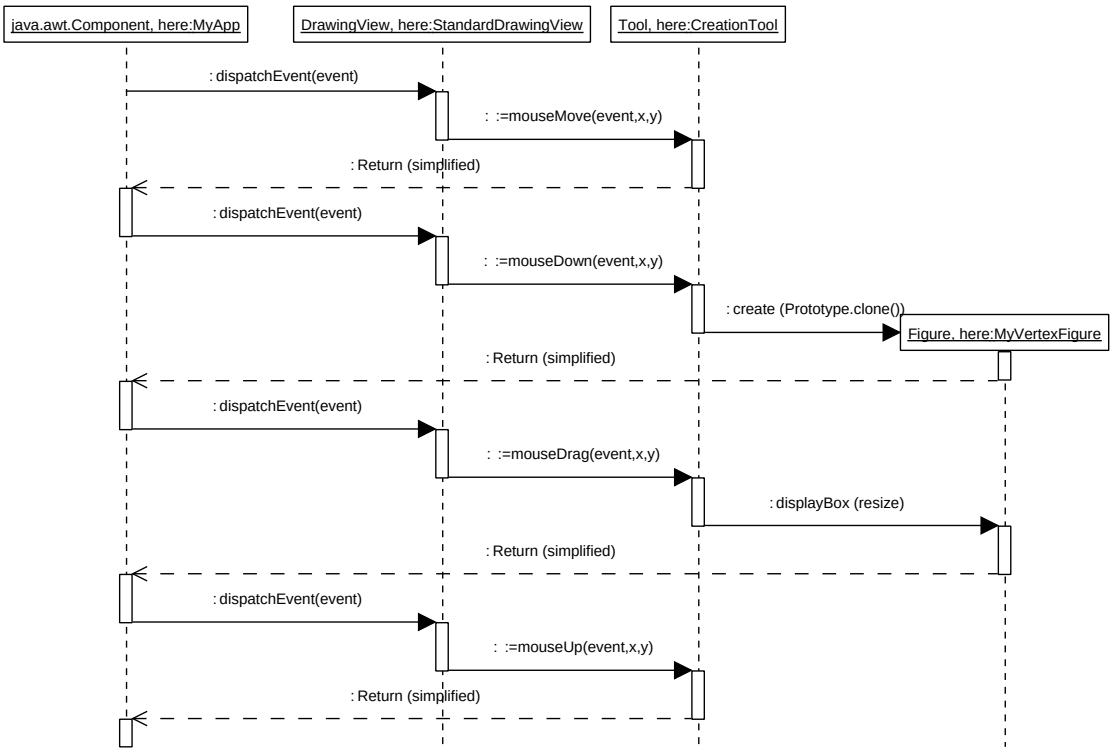

Abbildung 14: Sequenzdiagramm JHotDraw, Erstellen eines Knotens

In diesem Diagramm werden insgesamt vier Ereignisse verarbeitet, die hier untereinander in einem Diagramm aufgeführt sind. Voraussetzung für obiges Diagramm ist, dass der Editor in den richtigen Zustand (hier: Erstellung eines Knotens) versetzt worden ist. Das heißt, dass der aktuelle Zustand durch eine Instanz vom Typ `CreationTool` repräsentiert wird. Der Benutzer fährt mit der Maus über die Zeichenfläche und die Swing- bzw. AWT-Bibliothek setzt dies in entsprechende Ereignisse um. Diese Art von Ereignis führt innerhalb des Frameworks bzw. des Zustands `CreationTool` zu keinen weiteren Aktionen. Erst wenn der Benutzer den Mausknopf drückt, erstellt das `CreationTool` einen neuen Knoten. Hierbei verwendet JHotDraw das Prototypmuster (vgl. [Gamma+96, 144]): Beim Instanziieren der `CreationTool`-Instanz wird dieser ein Prototyp des zu erzeugenden Produktes, hier eine `MyVertexFigure`, übergeben. Während der Mausknopf gedrückt gehalten wird, ändert das `CreationTool` die Größe der *Figur*. Beim Loslassen des Knopfes wird dann keine weitere Aktion mehr durchgeführt.

Wie schon beim Instanziieren des Frameworks fällt hier dessen Übersichtlichkeit auf: Lediglich vier Klassen sind beteiligt.

3.1.2 Tigris GEF

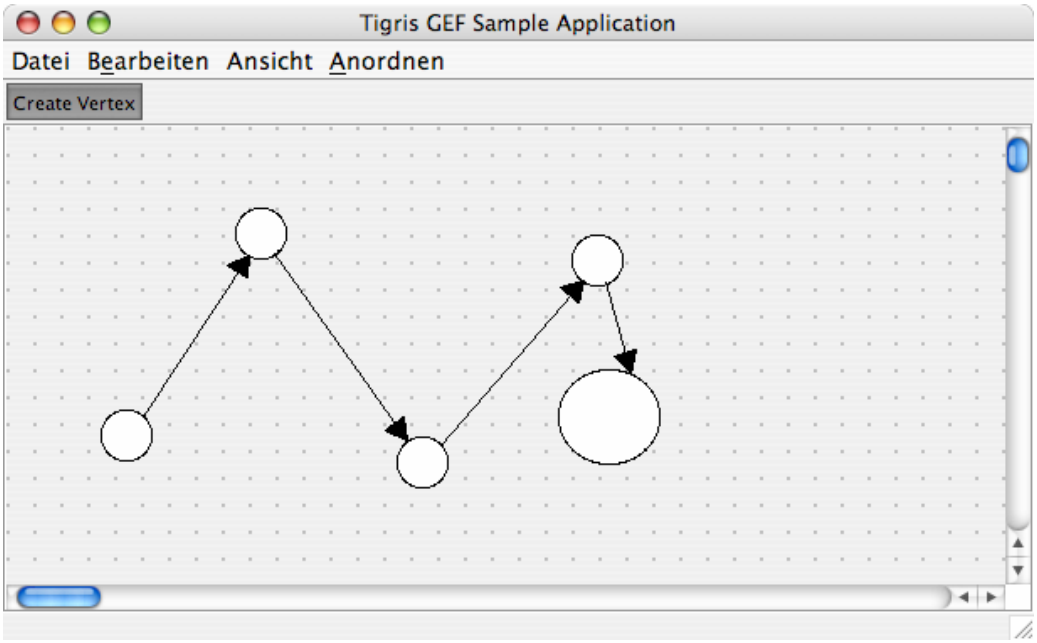

Abbildung 15: Tigris GEF Beispielapplikation

Tigris GEF wurde von Jason Robbins im Rahmen seiner Disseratation ([Robbins99]) entwickelt. Ausgehend von Unidraw und HotDraw wurde das „Graph Editing Framework“ mit folgenden Zielen entworfen:

> “GEF takes this previous work [Unidraw und HotDraw, Anm. d. Autors] into account but emphasizes extensibility, simplicity, and a high-quality user experience. HotDraw and Unidraw both achieve great extensibility by using flexible, abstract concepts. I limited the number and flexibility of GEF’s concepts to make the framework more understandable.“ [Robbins99, 186]

Als Dokumentation von Tigris GEF konnte hier diese Dissertation und die API-Dokumentation verwendet werden. In [Robbins99, 185-192] wird der grundlegende Aufbau knapp beschrieben, die folgende Beschreibung hat diesen Text zur Grundlage, basiert aber überwiegend auf der eigenen Analyse der Klassen.

a) Struktur

Die Hauptklassen von Tigris GEF sind in drei Pakete aufgeteilt, die grob die Teilnehmer Controller, View und Modell des MVC-Musters widerspiegeln:

- *`org.tigris.gef.base`*

- *org.tigris.gef.presentation*
- *org.tigris.gef.graph*

Das folgende Diagramm zeigt die wichtigsten Klassen dieser Pakete und ihre Beziehungen:

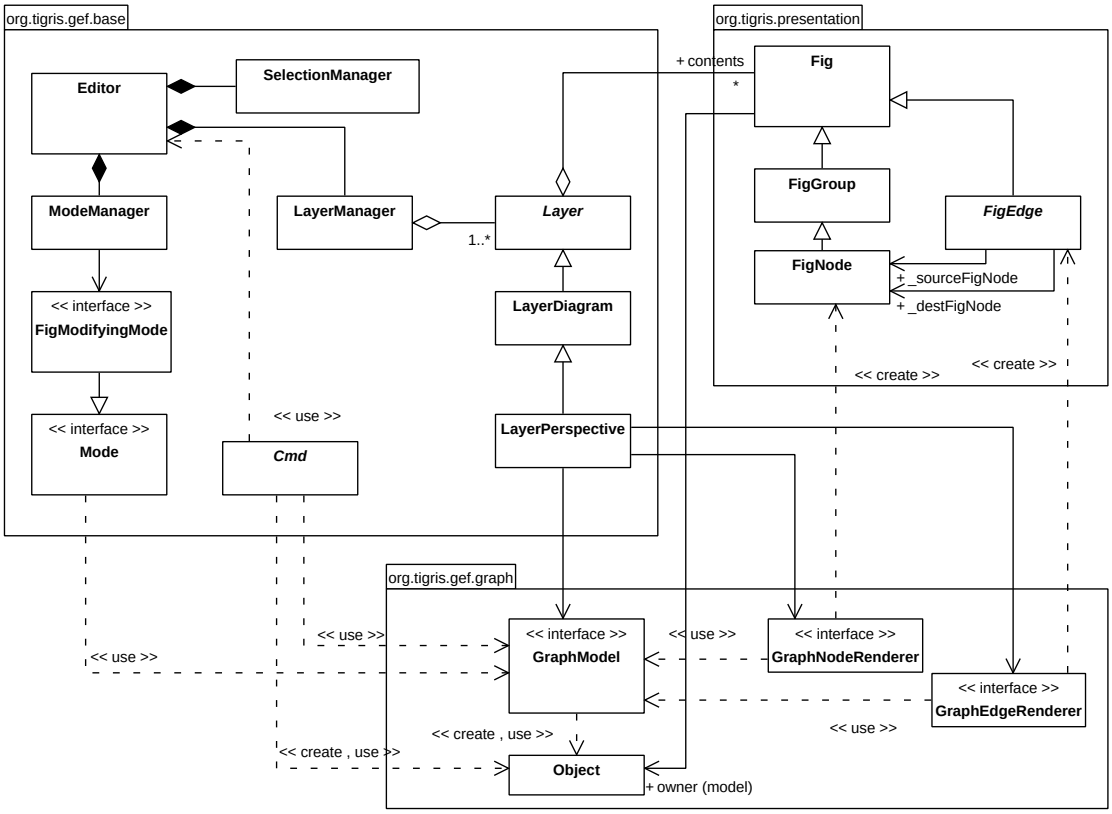

Abbildung 16: Klassendiagramm Tigris GEF, vereinfachter Überblick

Die zentrale Klasse des Frameworks ist die Klasse `Editor`. `Editor` ist ein Mediator (vgl. [Gamma+96, 385ff]), der die einzelnen Teile des *Controllers*, nämlich `ModeManager`, `LayoutManager` und `SelectionManager`, miteinander verbindet. Je nach Benutzeraktion kann ein `Mode` aktiviert werden, dieser entspricht einem Zustand des Editors bzw. *Controllers*. Abhängig vom Zustand werden Benutzeraktionen unterschiedlich interpretiert. Neben `Mode`-Klassen werden von `Cmd` abgeleitete *Befehle* zur Durchführung von Aktionen verwendet. Diese sind entsprechend dem Befehlsmuster gestaltet und können zur Implementierung der Undo-Funktionalität verwendet werden. Tigris GEF enthält die erstaunliche Anzahl von 49 Befehlsklassen, wobei hier nur `CmdSequence`, eine Art Meta-Befehl, die entsprechende `undoIt()`-

Funktion auch wirklich implementiert. Der `SelectionManager` verwaltet Listen von ausgewählten *Figuren*; auf diese Funktionalität wird hier nicht weiter eingegangen.

Das Paket `presentation` ist im Diagramm rechts oben zu sehen. Wichtigste Klasse ist dort die `Fig`. Diese bzw. ihre Unterklassen dienen der Darstellung von *Figuren* – den graphischen *Komponenten* von Tigris GEF. Für Knoten und Kanten werden spezielle Klassen bereitgestellt. Tigris GEF interpretiert wie alle vorgestellten Frameworks die Zeichnung in Form eines Graphen mit Knoten und Kanten. Zusätzlich kennt Tigris GEF so genannte *Ports*. Hierbei handelt es sich um *Figuren*, die in einer Knotenfigur, einer Ableitung von `FigNode`, als *Port* definiert werden.

Die Anbindung des Modells geschieht durch das Paket `graph`. Die Fassadenklasse `GraphModel` steuert das eigentliche Modell an (zum Fassadenmuster siehe [Gamma96+, 212]). Zum Erzeugen neuer Komponenten werden so genannte *Renderer* eingesetzt, die passend zu Modell-Objekten entsprechende *Figuren* erzeugen. Diese *Renderer* übernehmen also die Aufgabe von Fabriken nach dem Fabrikmuster (vgl. [Gamma+96, 107]).

b) Instanziierung

Im Folgenden wird wiederum die zuvor eingeführte Beispielapplikation umgesetzt. Da Tigris GEF im Unterschied zu JHotDraw den Einsatz eines Modells explizit unterstützt, wurde hier das Beispiel leicht abgewandelt: Als reine Modell-Klassen neu eingeführt werden die Klassen `MyVertex`, `MyEdge` und `MyGraph`, die über das `PropertyChangeListener`-Interface Beobachter benachrichtigen können. Diese Klassen sind also unabhängig von Tigris GEF und werden auch in der Beispielapplikation zu Eclipse GEF zum Einsatz kommen.

Hier das Klassendiagramm mit Client-spezifischen Klassen:

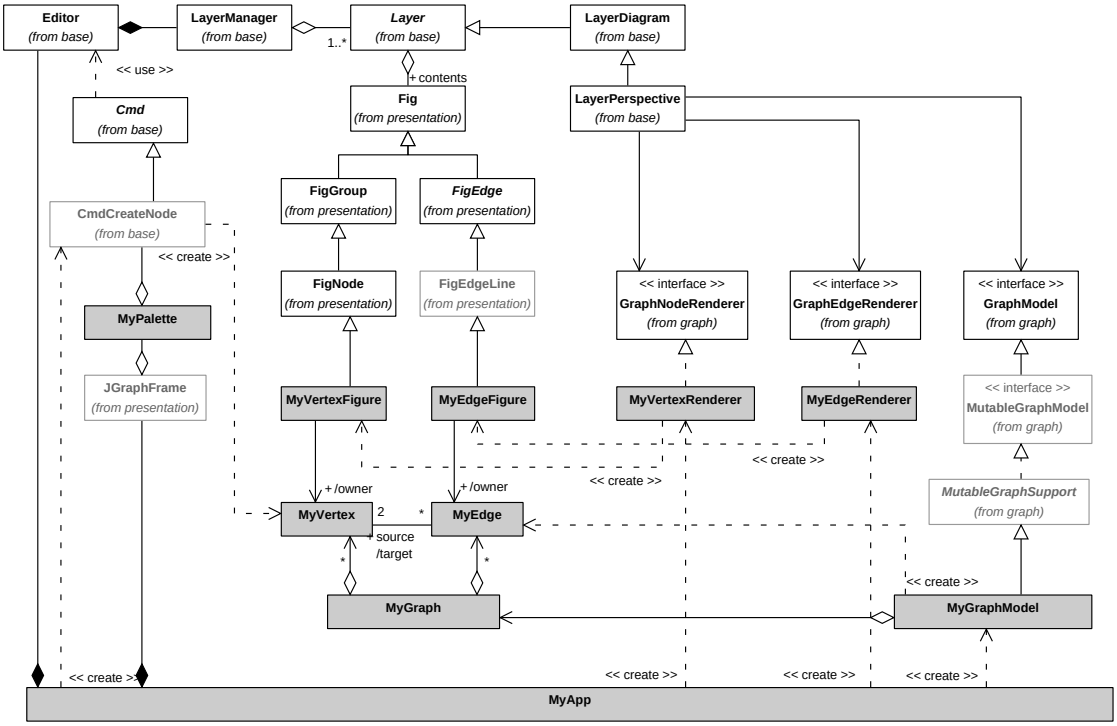

Abbildung 17: Klassendiagramm Tigris GEF, Instanziierung

Im Gegensatz zu JHotDraw ist hier eine deutlich höhere Anzahl eigener Klassen zu implementieren. Dies rührt allerdings vor allem daher, dass nun ein unabhängiges Modell eingesetzt wird und dies entsprechend eingebunden werden muss. Das Modell wird dabei über die Fassadenklasse `MyGraphModel` und die beiden Fabrikklassen `MyVertexRenderer` und `MyEdgeRenderer` sowohl den *Controller*- (`Editor`, `Cmd`) als auch *View*-Klassen (`Fig`) zugänglich gemacht.

```
    protected void initModel() {

        LayerPerspective lp = (LayerPerspective)

                    getEditor().getLayerManager().getActiveLayer();

        lp.setGraphNodeRenderer(new MyVertexRenderer());

        lp.setGraphEdgeRenderer(new MyEdgeRenderer());

        getEditor().setGraphModel(new MyGraphModel());

    }
```

Codebeispiel 2: MyApp.java, initModel()

Wie bei JHotDraw sind auch hier Figuren für die *Komponenten* (`MyVertexFigure`, `MyEgdeFigure`) sowie eine Klasse zur Steuerung der gesamten Anwendung, insbesondere zum Initialisieren aller Elemente (`MyApp`), zu erstellen.

Während in JHotDraw die benötigten *Tools* direkt in der Applikationsklasse initialisiert wurden, wird hier eine eigene *Palette* (`MyPalette`) verwendet, welche die *Befehle* als Aktionen von AWT-Buttons initialisiert. Dies ist aus Gründen der Übersichtlichkeit im Diagramm nicht dargestellt. Im Gegensatz zu JHotDraw, wo die entsprechenden Figuren als Prototypen an ein *Tool* (also ein Zustands-Objekt) übergeben wurden, werden hier die Metaklassen der Modellklassen verwendet und das Befehlsmuster eingesetzt, etwa:

```
...
 button = add(new CmdCreateNode(MyVertex.class, "Vertex"));
 ...
```

Codebeispiel 3: MyPalette.java, Auszug

Erstaunlicherweise muss hier kein *Befehl* oder *Tool* zum Erzeugen von Kanten initialisiert werden. Das entsprechende Zustandsobjekt, das dann auch die Erstellung der Kante regelt, wird von Tigris GEF standardmäßig erzeugt, wenn der Mauszeiger im Selektionsmodus über eine als *Port* ausgezeichnete *Figur* fährt. Dabei wird dann die Modell-Fassade verwendet um zwischen den verbundenen Knoten eine passende Kante zu erzeugen.

c) Ereignisverarbeitung

Wieder soll nun ein Knoten erzeugt werden. Ähnlich wie bei JHotDraw durchläuft das Ereignis (*Event*) zunächst die Java-AWT-Klassen und -Methoden zur Ereignisverarbeitung, bis schließlich der `Editor` aufgerufen wird. Wie bei JHotDraw wird auch hier das Zustandsmuster eingesetzt, allerdings wird nun ein `ModeManager` zwischengeschaltet, was für den Client unerheblich ist. Erstaunlich ist jedoch der Einsatz des Befehlsmusters an dieser Stelle: Der Befehl `CmdCreateNode` erzeugt nämlich wider Erwarten keinen Knoten, sondern führt eine Zustandstransition des *Controllers* durch, in dem der aktuelle Zustand erzeugt und – indirekt – gesetzt wird.

Das erste Sequenzdiagramm zeigt den Vorgang, wie er bei der Auswahl des *Tools* „Create Vertex“ in der Palette intern abläuft:

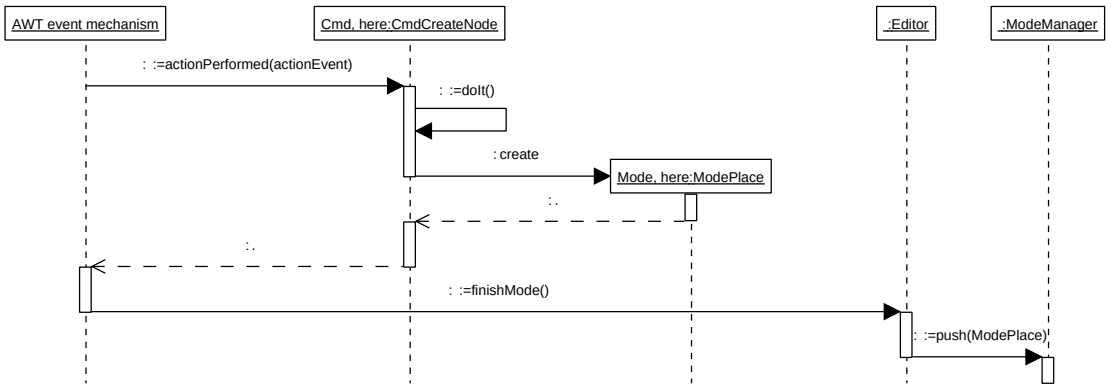

Abbildung 18: Sequenzdiagramm Tigris GEF, Erstellen des Mode-Objekts

Über den AWT-Eventmechanismus wird der Befehl ausgeführt. Der Befehl erzeugt den neuen Zustand (`ModePlace`) und legt diesen zunächst in einer globalen Hilfsvariable ab (hier nicht aufgeführt). Danach ruft der AWT-Eventmechanismus[11] den `Editor` auf, dieser holt das erzeugte Zustandsobjekt und gibt es an seinen `ModeManager` weiter. Der `ModeManager` arbeitet bei Tigris GEF mit einem Stapel: Bei jedem Ereignis wird jeweils der oberste Zustand aktiv, falls dieser das Ereignis nicht verarbeitet, kann der nächste Zustand auf dem Stapel das Ereignis verarbeiten. Der neue Zustand wird daher mit `push(Mode)` aktiviert.

Obwohl also bereits der Befehl `CmdCreateNode` mit `doIt()` aufgerufen wurde, ist noch keine Knotenkomponente erzeugt worden. Dies erfolgt in einem zweiten Schritt, nämlich wenn der Benutzer den Mausknopf innerhalb der Zeichenfläche drückt. Da der Zustand bereits aktualisiert wurde, können der Mauszeiger und die Anzeige in der Statuszeile kontextabhängig, oder besser zustandsabhängig, vom aktuellen `Mode`-

[11] Der AWT-Mechanismus wandelt einen Mausklick intern in bis zu drei Ereignisse um: „mouse pressed“, „mouse released“ und „mouse clicked“. Daher werden die Frameworkklassen, die sich für jeweils unterschiedliche Ereignisse als Beobachter bei den AWT-Komponenten angemeldet haben, in der gewünschten Reihenfolge aufgerufen. Dies spielt aber zum Verständnis des Frameworks keine Rolle.

Objekt angepasst werden. Das zweite Diagramm zeigt nun die Reaktion auf die Benutzereingabe:

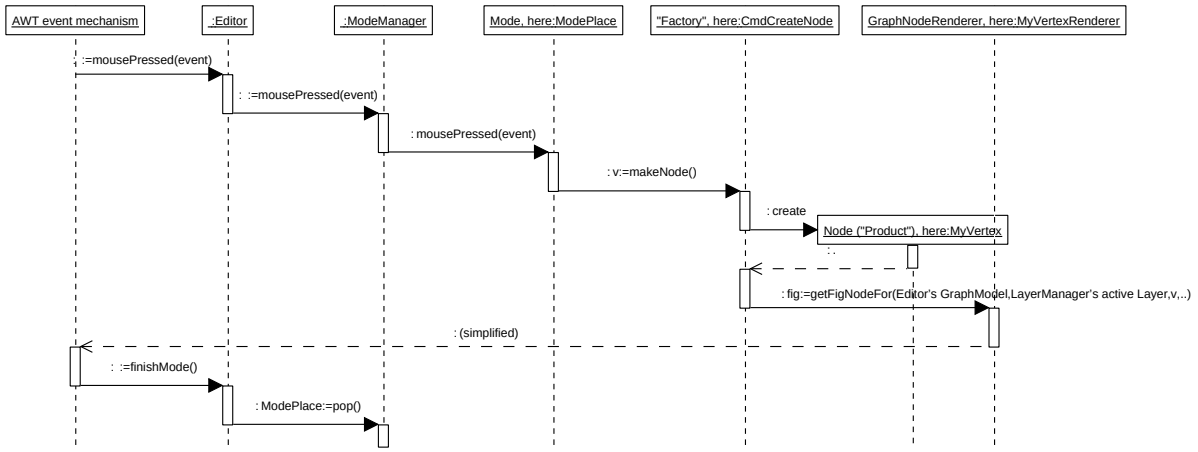

Abbildung 19: Tigris GEF, Sequenzdiagramm, Erzeugen des Knotens

Der `Editor` wird aufgerufen. Dieser reicht das Ereignis entsprechend dem Zustandsmuster an das aktuelle Zustandsobjekt weiter, was hier indirekt über den `ModeManager` erfolgt. Der *Zustand* ist nun für die Erzeugung des Knotenobjekts zuständig. Dabei wird wieder der bereits erwähnte Befehl `CmdCreateNode` aktiv, diesmal aber nicht in seiner Rolle als *Befehl* aus dem Befehlsmuster, sondern als *Fabrik* mit der Methode `makeNode()`. Dort wird dann auch ein Objekt vom Typ `MyVertex`, also das Knotenmodell, erzeugt. Der Befehl ruft zusätzlich den `GraphNodeRenderer` mit passenden Parametern auf – sowohl der *Renderer* als auch die Parameter werden sämtlichst vom `Editor` bereitgestellt. Damit ist der eigentliche Prozess der Knotenerzeugung abgeschlossen. Der AWT-Eventmechanismus ruft abschließend erneut den `Editor` auf, der den *Zustand* zum Erzeugen von Knoten vom Zustandsstapel entfernt.

Letztendlich führt also das Befehlsobjekt die Erzeugung durch, allerdings nicht in der Rolle des *Befehls*, sondern als vom *Zustand* eingesetzte *Fabrik*. Zu beachten ist, dass die Fabrikmethode `makeNode()` hier nicht Teil der Befehlsschnittstelle `Cmd`, sondern eine Besonderheit dieser speziellen Befehlklasse ist. Dies ist vielleicht auch der Grund, warum Tigris GEF so viele vorimplementierte Befehle mitbringt. Die enge Verzahnung mit dem AWT-Eventmechanismus und die Erweiterung von Befehlsklassen mit

Methoden, die nicht im Befehlsinterface definiert sind, aber vom Framework intern aufgerufen werden, erschwert die Implementierung eigener Client-spezifische Befehle.

3.1.3 Eclipse GEF

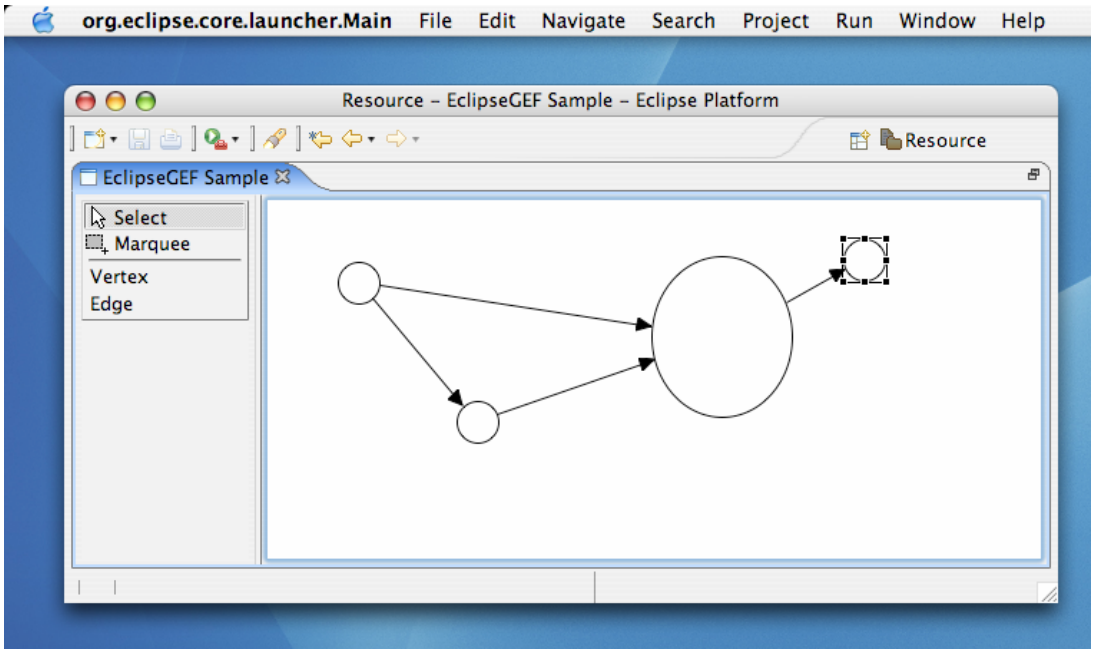

Abbildung 20: Eclipse GEF Beispielapplikation

Eclipse GEF verwendet wie Tigris GEF eine MVC-Struktur, setzt diese aber sehr viel kleinteiliger um. Ursprünglich wurde hier eine JHotDraw-Version (speziell für die Widget-Bibliothek SWT) als *View*-Komponente eingesetzt, diese wurde dann durch eine eigene Bibliothek, Draw2D, die mit Eclipse GEF mitgeliefert wird, ersetzt.[12]

Eclipse GEF ist Gegenstand mehrerer Tutorials ([Eclipse Wiki, GefDescription], [Bokowski+05]). [Moore+04, 87ff] bietet eine sehr ausführliche Einführung, die vor allem auf das Zusammenspiel von GEF und EMF eingeht. Neben den genannten Artikeln und der eigenen Codeanalyse ist bei der Arbeit mit Eclipse GEF die dazugehörige Newsgroup der Eclipse Foundation[13] eine wertvolle Informationsquelle.

---

[12] Für diesen Hinweise danke ich Randy Hudson, GEF Team Lead, IBM .

[13] URL: http://www.eclipse.org (Stand 8.3.2005)

a) Struktur

Das folgende Diagramm zeigt sehr vereinfacht die wichtigsten Klassen und Schnittstellen von Eclipse GEF.

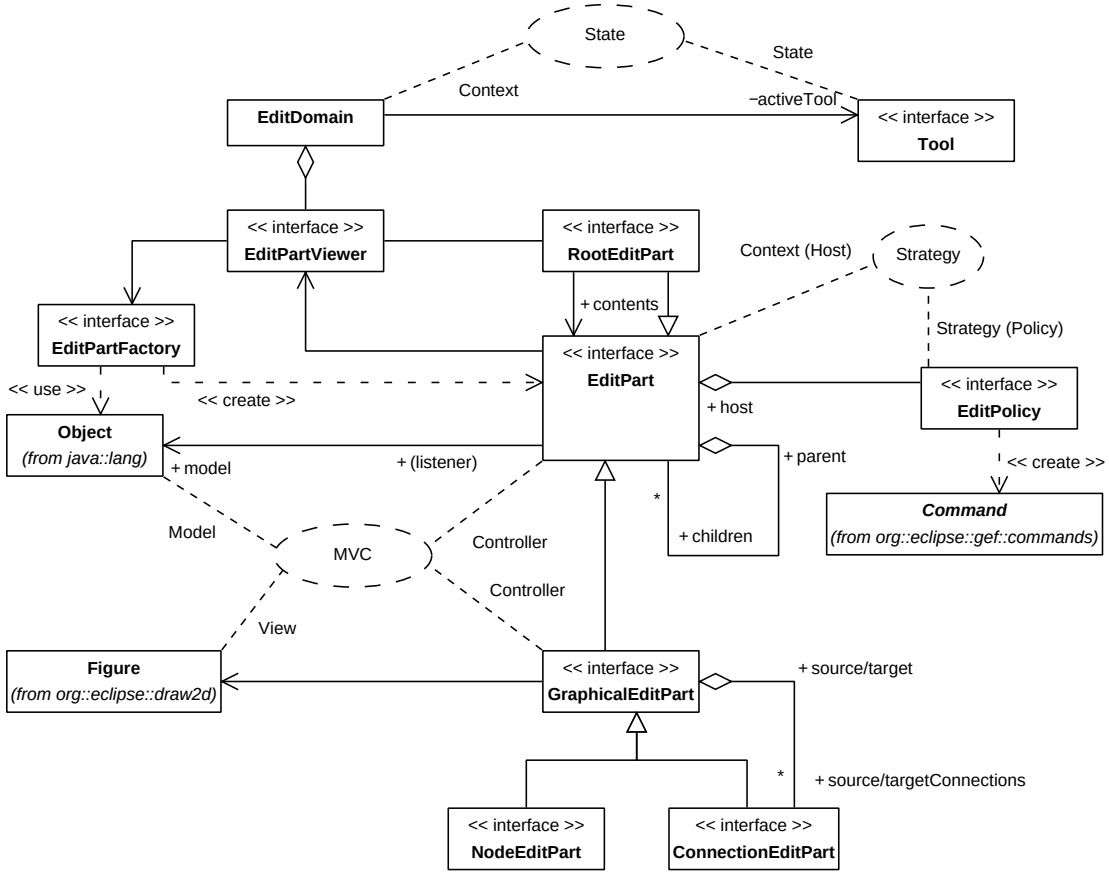

Abbildung 21: Klassendiagramm Eclipse GEF, Übersicht

Im Mittelpunkt von Eclipse GEF steht ein MVC-Tripel. Der *Controller* muss bei Eclipse GEF die Schnittstelle `EditPart` implementieren. Die MVC-Struktur von Eclipse GEF ist sehr kleinteilig, das heißt für jede *Komponente* der Zeichnung muss es ein entsprechendes MVC-Tripel geben. Daher ähnelt diese Struktur sehr der in [Krasner+88] beschriebenen. Die Kopplung der MVC-Teilnehmer ist sehr lose: Zunächst werden weder für die *Modell*- noch für die *View*-Klassen Vorgaben gemacht. Sinnvollerweise werden aber, über die Spezialisierung `GraphicalEditPart`,

zumindest Instanzen der Klasse `Figure` aus dem mitgelieferten Draw2D-Paket[14] verlangt. Diese Kombination unterstützt dann auch die gesamte Ereignisverarbeitung. Der Client des Frameworks bemerkt in diesem Fall nichts von den dahinter liegenden technischen Details und muss „nur“ seine Implementierungen passend an die Frameworkklassen anmelden.

Alle Komponenten eines Diagramms müssen Kinder einer Wurzelkomponente sein, die über ein `RootEditPart` verwaltet wird. Diese Komponente ist nicht die Zeichnung selbst, sondern ein technisches Objekt. Das einzige Kind dieser Wurzelkomponente ist dann die Zeichnung. Der `RootEditPart` wird von einem `EditPartViewer` verwaltet: der Zeichenfläche. Typischerweise wird hier eine Spezialisierung mit Scroll- und Zoomfunktionalitäten verwendet, die Eclipse GEF mitliefert. Im Gegensatz zu JHotDraw oder Tigris GEF, wo die Hierarchie der angezeigten Komponenten über die *Figuren*, also die *View*-Klassen, definiert wurde, sind bei Eclipse GEF die *Controller*-Einheiten, also die `EditParts`, dafür zuständig. Daher nimmt auch die Klasse `EditPartViewer` die Rolle eines *Controllers* ein, dessen *View* die Zeichnfläche eines SWT-Widgets ist – dies ist für den Client allerdings transparent.

Über die Schnittstelle `EditPolicy` können `EditParts` in ihrem Verhalten nach dem Strategiemuster (*Strategy* oder *Policy Pattern*) erweitert werden. Die im Strategiemuster neuralgische Auswahl der jeweils passenden *Strategie* wird bei Eclipse GEF über vordefinierte *Rollen* geregelt: Eine *Strategie*, also `EditPolicy`, wird für

---

[14] Das Draw2D-Paket mit seiner Superklasse Figure ist ein eigenes Framework zur Erstellung zweidimensionaler Zeichnungen. Es basiert auf der Widget-Bibliothek SWT. Letztere wird im Zusammenhang mit Eclipse entwickelt und ist eine Konkurrenzbibliothek zu Javas Swing-Bibliothek. Sie ist mit dieser nicht kompatibel, obwohl seit Eclipse 3.0 (bzw. SWT 3.0) Swing-Komponenten in SWT-Komponenten eingefügt werden können. Umgekehrt ist (bisher) keine Anbindung möglich. Der Vorteil von SWT gegenüber Swing ist eine plattformnähere und performantere Programmierung. Da es systemeigene Funktionen verwendet, ist es nicht plattformübergreifend, sondern kann nur auf Systemen mit entsprechender SWT-Implementierung laufen; dazu gehören aber die gängigen Betriebssysteme wie Windows, Linux oder Mac OS X.

jeweils eine definierte Rolle beim `EditPart` installiert. So stehen beispielsweise Rollen zur Erzeugung neuer Komponenten oder zur Verwaltung des Layouts zur Verfügung. Bis auf die Erzeugung der *View*-Komponente, die bei Eclipse GEF Aufgabe der *Controller*, also `EditParts`, ist, und die Umsetzung effizienter Updatemechanismen bei Änderungen des *Modells*, werden alle Aufgaben des *Controllers* durch `EditPolicy`-Klassen übernommen, die vom Client entsprechend definiert werden müssen. Die `EditPolicy`-Intanzen führen die Aufgaben jedoch nicht selbst durch, sondern erzeugen Client-spezifische Instanzen der Klasse `Command` nach dem Befehlsmuster. Die Befehle sind hier auch für Undo- und Redofunktionen verantwortlich.

Wie in den anderen Frameworks wird die Interpretation der Benutzereingaben auch bei Eclipse GEF über *Tools* definiert, die nach dem Zustandsmuster den Zustand des Editors festlegen. Der Kontext ist hier die Klasse `EditDomain`, die zugleich als *Mediator* zwischen den Zuständen und dem `EditPartViewer` fungiert und auch den Befehlsstapel für das Undo/Redo verwaltet.

Eclipse GEF stellt keine Anforderungen an das *Modell*. Prinzipiell muss jedoch jedem `EditPart` ein *Modell*-Objekt zugewiesen werden. Über eine `EditPartFactory` werden für Modellobjekte passende `EditParts` erzeugt. Daher ähnelt diese Klasse sehr den Klassen `GraphNodeRenderer` und `GraphEdgeRenderer` von Tigris GEF. Im Gegensatz zu Tigris GEF erzeugt die `EditPartFactory` jedoch *Controller*-Objekte, die dann erst die View-Objekte konstruieren.

b) Instanziierung

Diese allgemeinen Ausführungen werden in der Anwendung klarer. Folgendes Diagramm zeigt eine Umsetzung des Beispieleditors mit Eclipse GEF:

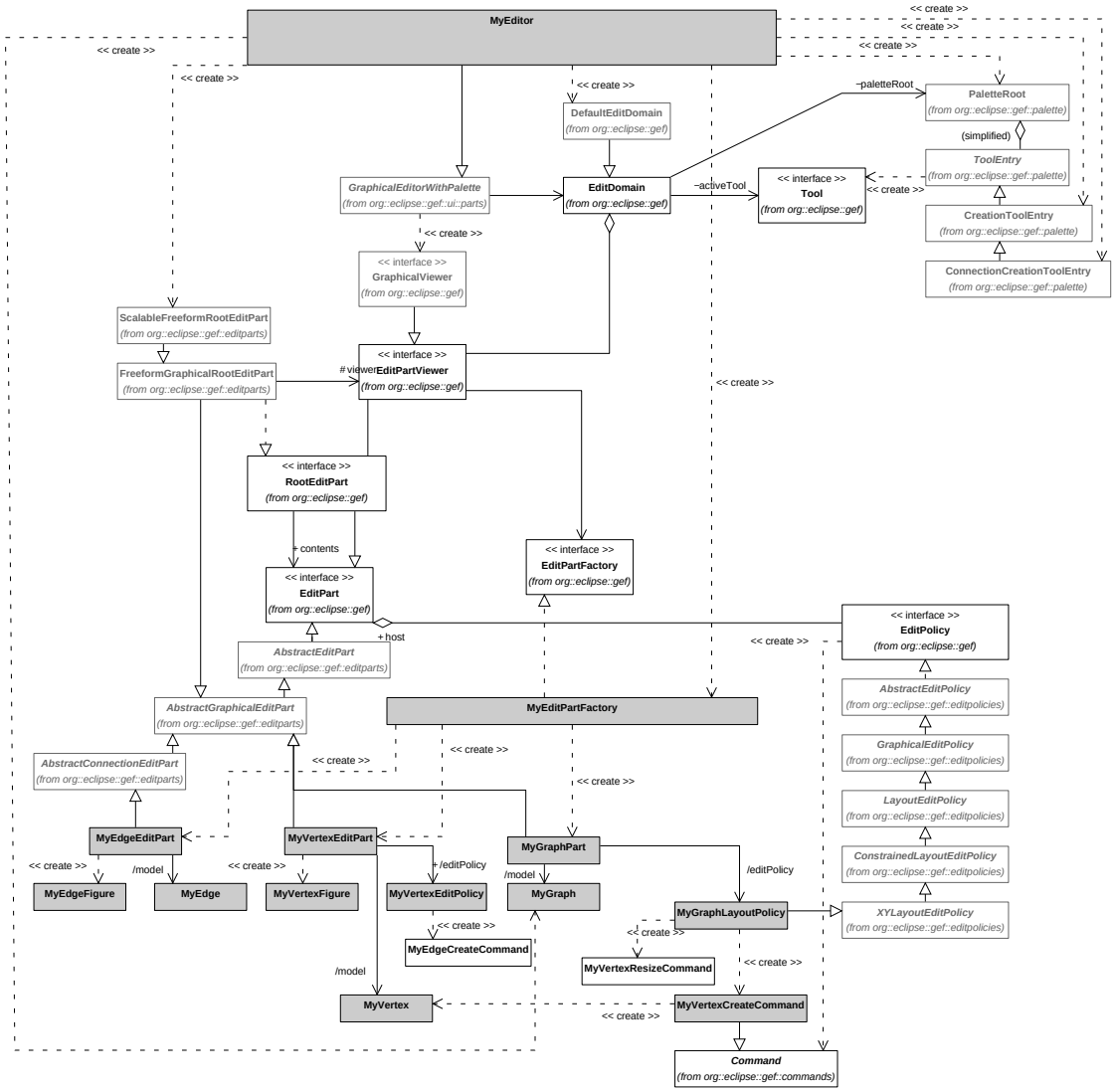

Abbildung 22: Klassendiagramm Eclipse GEF, Instanziierung

Bei Eclipse GEF müssen im Vergleich zu den anderen beiden Frameworks am meisten Klassen selbst erstellt werden. Dies liegt vor allem an der strikten Umsetzung des MVC-Musters; bei den anderen Frameworks mussten nicht für Knoten, Kante und Graph jeweils drei Klassen definiert werden (für den Graph wurde keine *View*-Klasse definiert, da hier eine vordefinierte Klasse verwendet werden konnte). Die Modellklassen sind die gleichen wie im vorherigen Beispiel bei Tigris GEF.

Die *EditParts* werden über eine Fabrikklasse erstellt, die passend zu den Modell-Objekten die entsprechenden *EditParts* erzeugt. Hier ein Ausschnitt aus der *EditPartFactory* mit der Fabrik-Methode:

```
    public EditPart createEditPart(
            EditPart i_context, Object i_model) {
        EditPart part = null;
        if (i_model instanceof MyGraph) {
            part = new MyGraphPart();
        } else if (i_model instanceof MyVertex) {
            part = new MyVertexPart();
        }
        if (part!=null) {
            part.setModel(i_model);
        }
        return part;
    }
```

Codebeispiel 4: MyEditPartFactory.java

Da das Beispiel sehr einfach gehalten ist, sind in diesem Beispiel nur zwei Strategien zu erzeugen: eine *LayoutPolicy* (*MyGraphLayoutPolicy*) für den *EditPart* des Graphen und eine *NodeEditPolicy* (*MyVertexEditPolicy*) für Knoten. Die *LayoutPolicy* verwaltet die Anordnung der Elemente des Graphen sowie deren Erzeugung. Folgendes Codefragment zeigt die Methoden dieser *Strategie* zum Anordnen und Erzeugen von Knoten:

```
...
    protected Command createChangeConstraintCommand(EditPart i_child,
            Object i_constraint) {
        Rectangle rect = (Rectangle) i_constraint;
        Object obj = i_child.getModel();
        if (obj instanceof MyVertex) {
            return new MyVertexResizeCommand((MyVertex)obj, rect);
        } else {
```

```
            return null;
        }
    }

    protected Command getCreateCommand(CreateRequest i_request) {
        Object obj = i_request.getNewObject();
        MyGraph G = (MyGraph) getHost().getModel();
        if (obj instanceof MyVertex) {
            return new MyVertexCreateCommand((MyVertex)obj,
                i_request.getLocation(), i_request.getSize(), G);
        }
        return null;
    }
...
```

Codebeispiel 5: MyGraphLayoutPolicy.java

Wie hier zu erkennen ist, erzeugt die *Strategie* Befehlsobjekte, die dann die eigentliche Aktion durchführen.

Kanten werden über *Befehle* erzeugt, die in der *Strategie* der Knoten-*Controller*, der `NodeEditPolicy`, erstellt werden. Die Befehle operieren dabei nur auf Modellebene, wie folgendes Beispiel des Befehls zum Erzeugen von Knoten zeigt:

```
...
    public MyVertexCreateCommand(MyVertex i_vertex, Point i_location,
            Dimension i_size, MyGraph i_graph) {
        super();
        m_Vertex = i_vertex;
        m_Location = i_location;
        m_Size = i_size;
        m_Graph = i_graph;
    }


    public void execute() {
        m_Vertex.setX(m_Location.x);
```

```
        m_Vertex.setY(m_Location.y);
        if (m_Size!=null) {
            m_Vertex.setWidth(m_Size.width);
            m_Vertex.setWidth(m_Size.height);
        }
        m_Graph.addVertex(m_Vertex);
    }
...
```

Codebeispiel 6: MyVertexCreateCommand.java

Wie an den teilweise recht tiefen Ableitungshierarchien und den vielen Client-Klassen zu sehen ist, ist die Verwendung von Eclipse GEF nicht trivial. Im Gegensatz zu den anderen Frameworks ist hier allerdings die erste Hürde auch die höchste – die Ergänzung um weitere Diagrammelemente oder neues Verhalten ist dann jeweils auf wenige oder zumindest analog zu erzeugende Klassen beschränkt.

Im Hinblick auf den dritten Teil dieser Arbeit, in dem Eclipse GEF zur Umsetzung eines UML-Editors eingesetzt wird, soll hier am Beispiel des *Controllers* der Graphen-*Komponente* die Umsetzung eines `EditParts` näher erläutert werden:

```
public class MyGraphPart
        extends AbstractGraphicalEditPart
        implements PropertyChangeListener {
```

Codebeispiel 7: MyGraphPart.java

Die Client-spezifischen `EditParts` werden im Allgemeinen von der Oberklasse `AbstractGraphicalEditPart` abgeleitet, welche die Basisfunktionalitäten bereitstellt. Zusätzlich müssen die `EditParts` die Beobachterschnittstelle entsprechend dem Beobachtermuster des *Modells* implementieren (hier `PropertyChangeListener`). Um den `EditPart` als Beobachter beim *Modell* an- und abmelden zu können, stehen entsprechende Schablonenmethoden (vgl. [Gamma+96, 366]) bereit, die vom Framework passend aufgerufen werden.

```
    public void activate() {
        if (!isActive())
```

```
            ((MyGraph)getModel()).addPropertyChangeListener(this);
        super.activate();
    }

    public void deactivate() {
        ((MyGraph)getModel()).removePropertyChangeListener(this);
        super.deactivate();
    }
```

Codebeispiel 8: MyGraphPart.java, activate() und deactivate()

Im Gegensatz zum „Original“-MVC-Muster aus [Krasner+88] erzeugt hier nicht die *View* den *Controller*, sondern umgekehrt der *Controller* die *View*. Dieses Verhalten wird unten noch ausführlicher diskutiert.

```
    protected IFigure createFigure() {
        Figure f = new FreeformLayer();
        f.setLayoutManager(new FreeformLayout());
        return f;
    }
```

Codebeispiel 9: MyGraphPart.java, createFigrue()

In diesem einfachen Beispiel wird nur eine *Strategie* eingesetzt. Hier ist zu sehen, wie die *Strategie* einer vordefinierten *Rolle* zugeordnet wird.

```
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new MyGraphLayoutPolicy());
    }
```

Codebeispiel 10: MyGraphPart.java, createEditPolicies()

Die Hierarchie der Komponenten spiegelt im Allgemeinen die Hierarchie der Modellobjekte wider. Allerdings können hier beliebige Abbildungsvorschriften hinzugefügt werden. Im Fall des UML-Editors werden etwa künstliche Modellkinder eingeführt, um die *Figuren* besser darstellen zu können und die Bedienbarkeit zu erhöhen.

```
    protected List getModelChildren() {
        return ((MyGraph)getModel()).getVerteces();
    }
```

Codebeispiel 11: MyGraphPart.java, getModelChildren()

Nachrichten des *Modells* müssen von den Client-spezifischen *Controllern* behandelt werden. Dabei ist es auch Aufgabe der *Controller*, entsprechende `refresh`-Methoden aufzurufen, um Änderungen des *Modells* in der *View* zu reflektieren.

```
    public void propertyChange(PropertyChangeEvent i_evt) {
        if (i_evt.getSource()==getModel()) {
            String strPropertyName = i_evt.getPropertyName();
            if (MyGraph.PROPERTY_VERTECES.equals(strPropertyName)) {
                refreshChildren();
            }
        }
    }
```

Codebeispiel 12: MyGraphPart.java, propertyChange(..)

c) Ereignisverarbeitung

Um die Zusammenarbeit der einzelnen Muster zu verdeutlichen, soll hier erneut das Beispiel der Erzeugung eines Knotens betrachtet werden. In der Ausgangssituation hat der Benutzer den entsprechenden Schalter in der Palette gewählt und fährt nun mit der Maus auf die Zeichenfläche, um den Knoten zu setzen.

Die Verarbeitung erfolgt in zwei Schritten:

1. Erzeugen eines `Request`-Objekts aus dem Ereignis
2. Erzeugen oder Ausführen eines `Command`-Objekts durch die *Strategie*

Im ersten Schritt wird das aktive *Tool*, also der aktuelle Zustand des Editors bzw. der `EditDomain`, verwendet, um aus dem Systemereignis – hier die Bewegung der Maus

(MouseMove) – ein *Request*-Objekt zu erzeugen. Das folgende Sequenzdiagramm[15] zeigt diesen ersten Schritt:

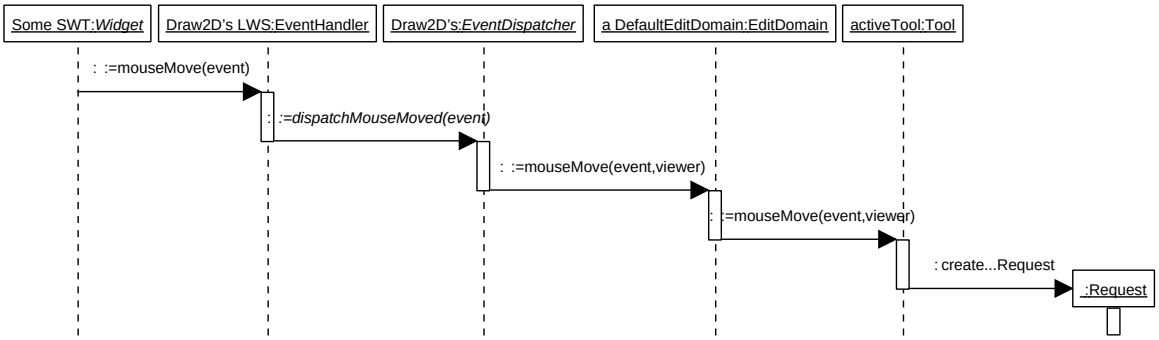

Abbildung 23: Sequenzdiagramm Eclipse GEF, Erzeugung des Request-Objekts

Um die Systemereignisse passend aufzubereiten wird hier das Statusmuster eingesetzt. Die weitere Verarbeitung des Ereignisses kann nun unabhängig vom Zustand erfolgen, dieser ist quasi im *Request*-Objekt aufgehoben. „Unabhängig" bedeutet in diesem Zusammenhang nicht nur, dass die weitere Verarbeitung einfacher wird, da der Zustand nicht explizit berücksichtigt werden muss, sondern auch, dass die Klassen, die im weiteren Verlauf involviert sind, strukturell unabhängig von den *Tool*-Klassen sind. Der *EventDispatcher* ist hier ein spezieller *DomainEventDispatcher*, der das *EditDomain*-Objekt kennt und so den Übergang vom Draw2D- zum GEF-Framework herstellt.

In einem nächsten Schritt wird aus dem *Request*-Objekt ein Befehlsobjekt (*Command*) erzeugt, wie in folgendem Diagramm zu sehen ist:

[15] Zur Erstellung der UML-Diagramme wurde Poseidon 3.0 verwendet. Die Fähigkeiten dieses Programms im Bereich Sequenzdiagramme sind eingeschränkt und auch nicht ganz UML2-konform. So sind Rücksprungnachrichten hier immer mit einer Beschriftung versehen – da diese hier nicht benötigt wurde, wurde versucht, diese möglichst kurz zu halten (die Zeichen „:." lassen sich leider nicht vermeiden – ein Beispiel für die Notwendigkeit eines eigenen Editors).

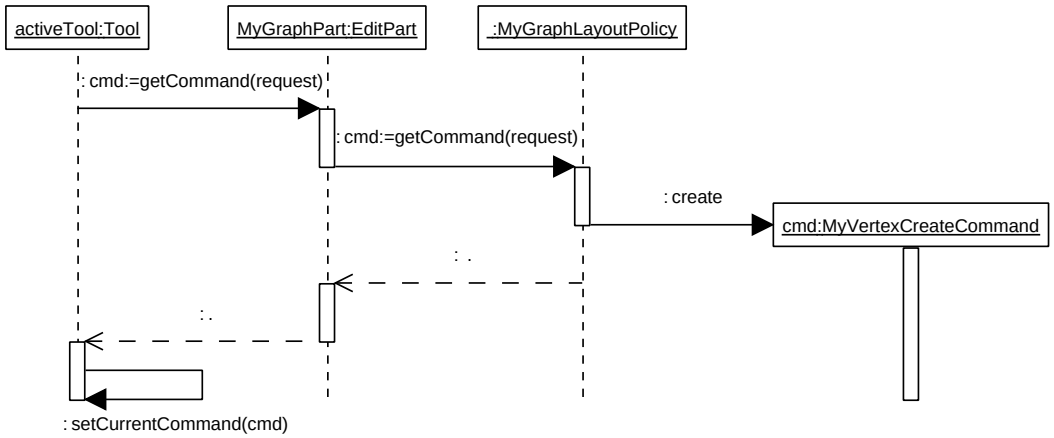

Abbildung 24: Sequenzdiagramm Eclipse GEF, Erzeugung eines Befehlsobjekts

Nachdem das aktive Tool im vorherigen Schritt das Ereignis in eine *Anfrage* (`Request`) umgewandelt hat, wird dieses Anfrageobjekt nun an den *Controller* der passenden *Komponente* übergeben, damit dieser ein Befehlsobjekt erzeugen kann. Die passende Komponente wird in diesem Fall mittels der aktuellen Mauszeigerposition auf dem `EditPartViewer` der `EditDomain` ermittelt – in anderen Fällen, das heißt *Zuständen* (*Tools*), kann die Komponente auf andere Art und Weise ermittelt werden. Der Controller der Komponente ist bei GEF eine Instanz der Klasse `EditPart`.

Die Erzeugung eines Befehlsobjekts wird über das Strategiemuster mittels einer passenden *Strategie*, hier *Policy*, durchgeführt. Im ersten Augenblick erscheint die Tatsache erstaunlich, dass ein „mouseMove“-Ereignis einen Befehl zur Erstellung eines Knotens erzeugt – in der Regel wird ein Knoten erst bei Tastendruck (typischerweise „mouseUp“) erstellt. In gewisser Weise ist dies hier auch der Fall, denn die Erzeugung des Befehlsobjekts ist ja noch nicht die Ausführung desselben! Die Ausführung des Befehls erfolgt tatsächlich erst bei Tastendruck im nächsten Schritt. Die Trennung von Erzeugung und Ausführung des Befehlsobjekts vereinfacht hier das visuelle Feedback, also beispielsweise die Anpassung des Mauszeigers. Die Bewegung des Mauszeigers über den Komponenten wird vom Framework intern im Sinne der Frage „Was würde passieren, wenn hier eine Taste gedrückt werden würde?“ interpretiert. Der erzeugte Befehl ist die Antwort auf diese Frage. Könnte an der entsprechenden Stelle keine Aktion stattfinden, so würde statt eines Befehlsobjekts „null“ zurückgegeben werden. Auch dies kann entsprechend visualisiert werden.

Ausgeführt wird der Befehl also erst bei Tastendruck. Wie beim Bewegen des Mauszeigers wird nun wieder in einem ersten Schritt aus dem Ereignis eine Anfrage generiert. Allerdings wird nun im zweiten Schritt kein weiterer Befehl erzeugt, sondern der zuvor erzeugte Befehl ausgeführt, wie folgendes Diagramm zeigt, wobei der erste Schritt hier verkürzt dargestellt ist:

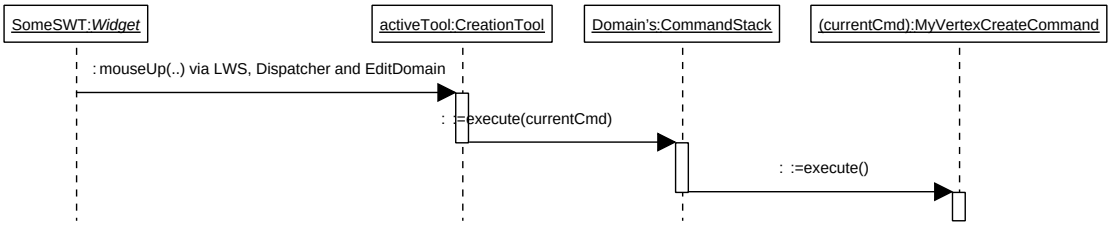

Abbildung 25: Sequenzdiagramm Eclipes GEF, Tool führt Befehl aus

Der Befehl wird nicht direkt vom Tool ausgeführt, sondern über den *`EditDomain`*-eigenen Befehlsstapel (*`CommandStack`*). In Kombination mit diesem Befehlsstapel ermöglicht das Befehlsmuster hier einfaches Undo- und Redo.

Die Ausführung des Befehls selbst führt zur Erzeugung einer neuen Komponente aus einem MVC-Tripel, wie das nächste Diagramm zeigt:

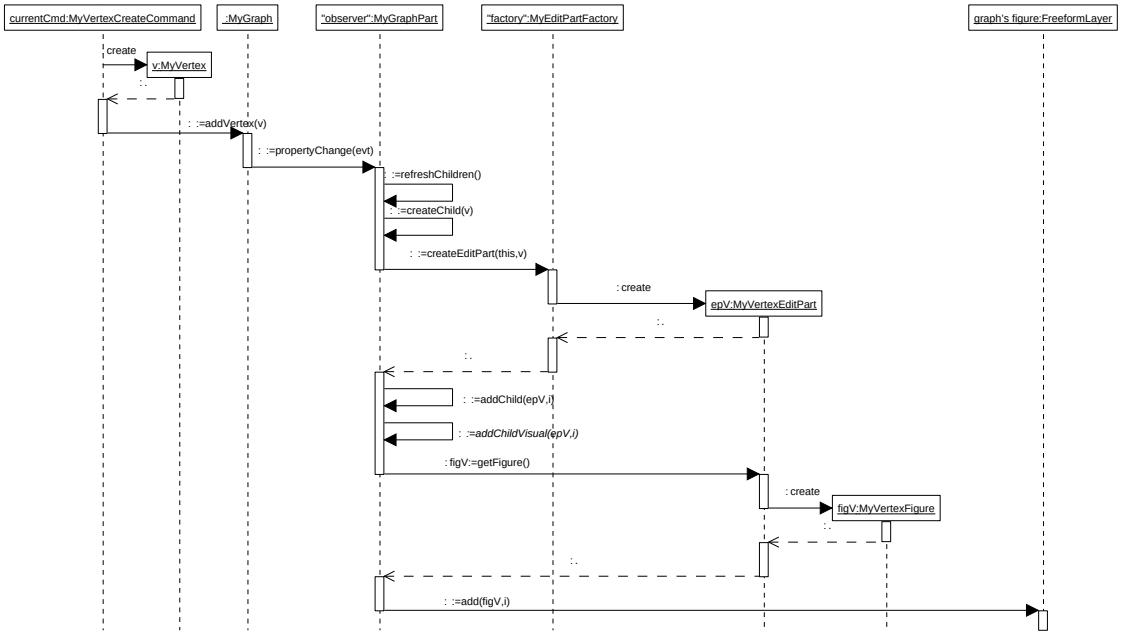

Abbildung 26: Sequenzdiagramm Eclipse GEF, Befehlsausführung

Der Befehl `MyVertexCreateCommand` erzeugt selbst einen neuen Knoten und fügt diesen dem Graphen hinzu. Dies geschieht hier zunächst nur auf Modellebene. Durch den Einsatz des Beobachtermusters wird der *Controller* des Graphen benachrichtigt. GEF bietet zwar eine Infrastruktur zur Durchführung der Befehle, die einzelnen Schritte müssen jedoch vom Client ausgeführt werden. Insbesondere die Update-Strategie muss hier korrekt umgesetzt werden (`refreshChildren()`). Die Oberklasse des Graph-*Controllers* erzeugt daraufhin intern eine neue Kindkomponente des Graphen, die dann den vom Befehl erzeugten Knoten als *Modell* erhalten wird. Die Instanziierung der Komponente beginnt also über das *Modell*. Der *Controller* des (zukünftigen) Behälters der Komponente, hier der Graph-*Controller*, veranlasst dann die Erstellung eines neuen *Controllers* (`createChild(v)`, `createEditPart(..)`) auf Basis des Modellobjekts durch die `EditPartFactory` (hier `MyEditPartFactory`) des Clients nach dem Fabrikmuster. Zusätzlich stößt der *Controller* dann die Erzeugung eines neuen *View*-Objektes für die neue Komponente an, dabei wird die entsprechende Funktion des *Controllers* der neuen *Komponente* (`getFigure()`) verwendet.

3.2 Diskussion

Im Folgenden werden die Frameworks anhand von drei zentralen Aspekten bzw. Mustern verglichen: Model-View-Controller-Muster, Ereignisverarbeitung und die Umsetzung von Constraints.

Zentral für die Architektur der Frameworks ist die Auswahl des Model-View-Controller Musters (MVC). Die Umsetzung dieses Musters ist für die Anwendung eines Frameworks von entscheidender Bedeutung, da die Modellklassen anwendungsspezifische Client-Klassen sind. Je strikter die Trennung von Darstellung und Modell durchgeführt werden kann, desto loser werden Framework- und Client-Klassen miteinander gekoppelt.

Ein für Frameworks wichtiges Prinzip ist die „*Inversion of Control*“:

> “The run-time architecture of a framework is characterized by an inversion of control.” [Fayad+97, 34]

Dieses Prinzip tritt insbesondere bei der Verarbeitung von Benutzereingaben zu Tage. Die Klassen des Frameworks rufen geeignet angemeldete und abgeleitete Client-Klasse

auf. Daher werden die Ereignisverarbeitung und die dabei zum Einsatz kommenden Muster eingehender untersucht.

Die Erweiterbarkeit und Flexibilität der Frameworks soll anhand der Umsetzung der speziellen Anforderungen graphischer Editoren, insbesondere von *Constraints*, überprüft werden.

Beim Vergleich wurden hier Entwurfsmuster verwendet. Entwurfsmuster bieten ein höheres Abstraktionsniveau als die Untersuchung auf Quellcodeebene. Ein weiteres Motiv zur Verwendung von Entwurfsmutern lag darin, dass die Dokumentation der Frameworks selbst explizit Entwurfsmuster verwendet. Der Vorteil der Verwendung von Entwurfsmustern gegenüber anderen Verfahren wie beispielsweise Metriken ist, dass sie den Aspekt der Anwendbarkeit explizit aufnehmen und die Konsequenzen der einzelnen Muster bekannt sind.

3.2.1 Model-View-Controller

Tigris GEF und Eclipes GEF verwenden laut Dokumentation (vgl. [Robbins99, 189], [Moore+04]) das Model-View-Controller-Muster (MVC). Ausführlich vorgestellt wird diese Struktur als Paradigma zur Programmierung von Benutzerschnittstellen in Smalltalk in [Krasner+88]. In [Gamma+96, 5ff] wird dieses Paradigma knapp analysiert und als Kombination der Entwurfsmuster Beobachter, Komposition und Strategie (sowie Fabrikmethode und Dekorierer) interpretiert. Mittlerweile wird MVC auch als Muster beschrieben, etwa in [Fowler03] oder unter dem Namen „Model-2“ als Architekturmodell (etwa in Zusammenhang mit Webapplikationen bei [Cavaness03]).

Das folgende Diagramm stellt das MVC-Pattern entsprechend der Analyse in [Gamma+96] dar:

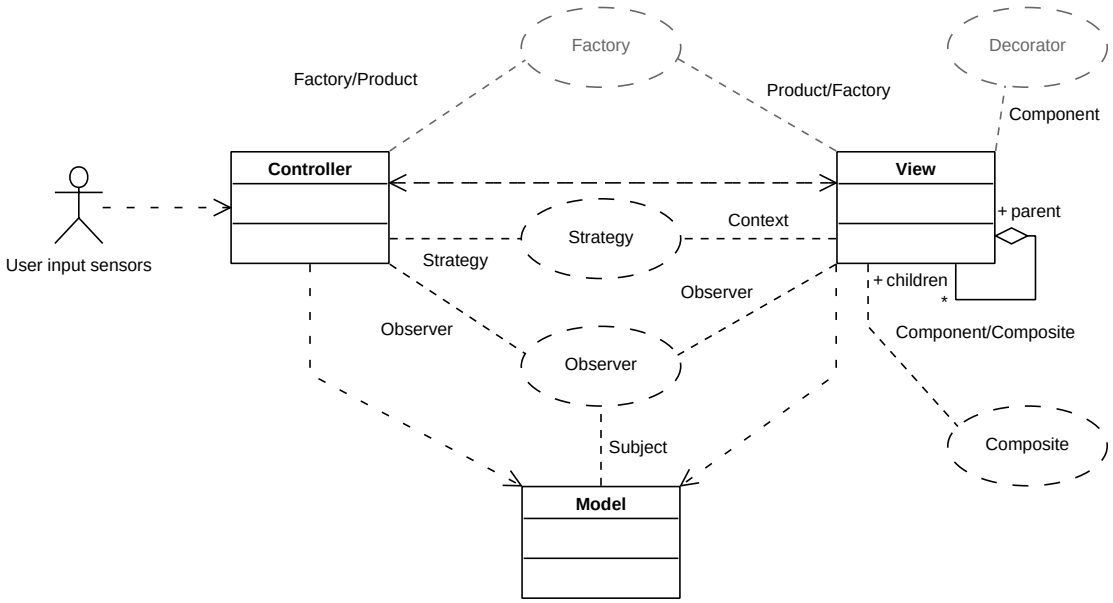

Abbildung 27: Klassendiagramm Entwurfsmuster Model-View-Controller

JHotDraw verwendet, im Gegensatz zu anderen HotDraw-Implementierungen, kein MVC-Muster. Tatsächlich trägt die Absenz des MVC-Patterns mit seinen drei Teilnehmern im Falle von JHotDraw sehr zur Übersichtlichkeit des Frameworks bei. `Figur`-Objekte sind hier sowohl für die Darstellung als auch für die Datenhaltung und die Geschäftslogik, also etwa semantische Abhängigkeiten, zuständig. Aus Sicht des MVC-Musters stellen sie das gesamte Tripel dar. Für den Fall, dass in eigenen Anwendungen eine Abtrennung zumindest des Modells erreicht werden soll, bietet JHotDraw keine Unterstützung an. Eine mögliche Lösung wäre, die Schnittstelle `Figure` als *View*-Rolle einer eigenen auf der Domänenebene angesiedelten MVC-Struktur zu interpretieren.

In der Architektur sind jedoch „Spuren“ des MVC-Musters von HotDraw erkennbar. In HotDraw sind die Teilnehmer des MVC-Musters die Klassen `DrawingEditor` (*Model*), `DrawingView` (*View*) und `DrawingController` (*Controller*). JHotDraw implementiert die Klasse `DrawingController` nicht mehr, *View* und *Controller* sind in der `DrawingView` zusammengefasst. Die Klasse `DrawingEditor` verwaltet hier auch kein *Modell*, sondern dient als *Mediator* (vgl. Vermittlermuster in [Gamma+96, 385]), an welchen evtl. ein *Modell* angemeldet werden könnte.

Der Versuch, die einzelnen Klassen des Frameworks von Tigris GEF den entsprechenden Rollen des Musters zuzuordnen, scheitert. Die Umsetzung des MVC-Musters wird in [Robbins99] so erklärt:

> "Like many user interface systems, GEF loosely follows the Model-View-Controller (MVC) design pattern to support multiple views (Krasner and Pope, 1988; Gamma et al., 1995). As with many MVC implementations, GEF sometimes combines the view and controller roles into the same object. GEF's GraphModels play the role of the model: they provide access to the semantic state of the diagram and send notification messages when that state changes. Layers and Figs act as models for the visual properties of the diagram, including coordinates, colors, and back-to-front ordering. GEF's Editors, Layers, and Figs provide most of the functionality of the view role by displaying the diagram. Other view functionality is provided by GEF's Modes and Selections, which also contribute graphics for interaction feedback. GEF's Modes and Selections primarily play the role of controller; however, GEF provides the option for Figs and model elements to perform some event handling." [Robbins99, 189]

Die folgende Tabelle ordnet die genannten Klassen den jeweiligen Rollen zu und fasst damit den zitierten Text zusammen. Zusätzlich wurden die Ergebnisse der Analyse der Ereignisverarbeitung aufgenommen, die nahelegt, dass der Editor mindestens in Teilen die Rolle eines *Controllers* einnimmt. Die Kreuze der Tabelle bedeuten, dass die Klasse die jeweilige Rolle einnimmt, ein größeres Kreuz visualisiert die primäre Anwendung.

	Model	View	Controller
`GraphModel`	X		
`Layer`, `Fig`	x	X	x
`Editor`		X	x
`Mode`, `Selection`		x	X
Client-Klassen	X		x

Tabelle 1: Klassen und ihre Rollen im MVC-Muster bei Tigris GEF

Die Tabelle zeigt, dass `Layer` und `Fig` beide sowohl *Modell* als auch *View*-Rollen einnehmen. *View*- und *Controller*-Rollen werden ebenfalls meistens kombiniert. Da die `Figs` die Modellrolle bezüglich graphischer Informationen einnehmen, kann es vorkommen, dass die Modellklassen des Client zu *Controllern* werden. Dies ist der Fall, wenn diese Klassen selbst graphische Informationen enthalten, wie im Beispiel des Graphen oben gezeigt. Die eigentlich unabhängigen Modellklassen `MyVertex` und `MyEdge` müssten im Beispiel als *Beobachter* bei den ihnen zugeordneten *Views* angemeldet werden, um die in ihnen gespeicherten graphischen Informationen (Koordinaten, Größe) konsistent zu halten. Tigris GEF stellt dafür übrigens eigene Schnittstellen zu Verfügung. Wenn die Modellklassen des Client diese implementieren, wird dies per Reflection erkannt und die Client-Klassen werden von den Frameworkklassen entsprechend aufgerufen.

Der entscheidende Punkt von MVC, die Trennung von Modell und Darstellung, scheint bei Tigris GEF nicht umgesetzt worden zu sein. Allerdings werden in obigem Zitat Hinweise auf den Kern des Problems gegeben, die den Widerspruch (MVC ohne Trennung von Modell und View) auflösen: Das Modell wird nicht mehr als kohäsive Einheit wahrgenommen, sondern in semantisches („semantic state“) und graphisches Modell („models for the visual properties of the diagram“) unterteilt. Diese Unterscheidung wird in Kapitel 4 der vorliegenden Arbeit wieder aufgenommen.

Eclipse GEF setzt das MVC-Muster konsequent um. In Abbildung 21 wurde bereits das Muster mit den Rollen als Kollaboration eingezeichnet. Allerdings weicht auch diese Umsetzung leicht vom beschriebenen Muster ab. Insbesondere die Ereignisverarbeitung und der Lifecycle unterscheiden sich von den in der Literatur (etwa [Buschmann+98, 129ff]) aufgeführten Merkmalen:

- nicht die *Views*, hier also Instanzen der Klasse `Figure`, erzeugen die zugehörigen *Controller*, sondern umgekehrt, die `Figures` werden von den `EditParts` erzeugt
- *Modell* und *View* sollten bei Eclipse GEF keine Abhängigkeiten haben, alle Änderungen sollen über den `EditPart` verwaltet werden
- der `EditPart` wird bei Eclipse GEF zur zentralen Steuerungskomponente über die Eingabeverarbeitung hinaus. So übernimmt der *Controller* auch die Abbildung der Objekthierarchie des Modells und kann hier modifizierend eingreifen

Diese Merkmale sind eher typisch für die Rolle des *Controllers* im Presentation-Abstraction-Controller-Muster, kurz PAC ([Buschmann98+, 145ff]). Idee dieses Musters ist die Aufteilung des Gesamtsystems in hierarchisch angeordnete Teilsysteme, *Agenten* genannt. Diese sind ähnlich wie bei MVC aufgeteilt, allerdings steht hier der *Controller* im Mittelpunkt des Tripels. Folgende Abbildung ist an [Buschmann98+, 153] angelehnt:

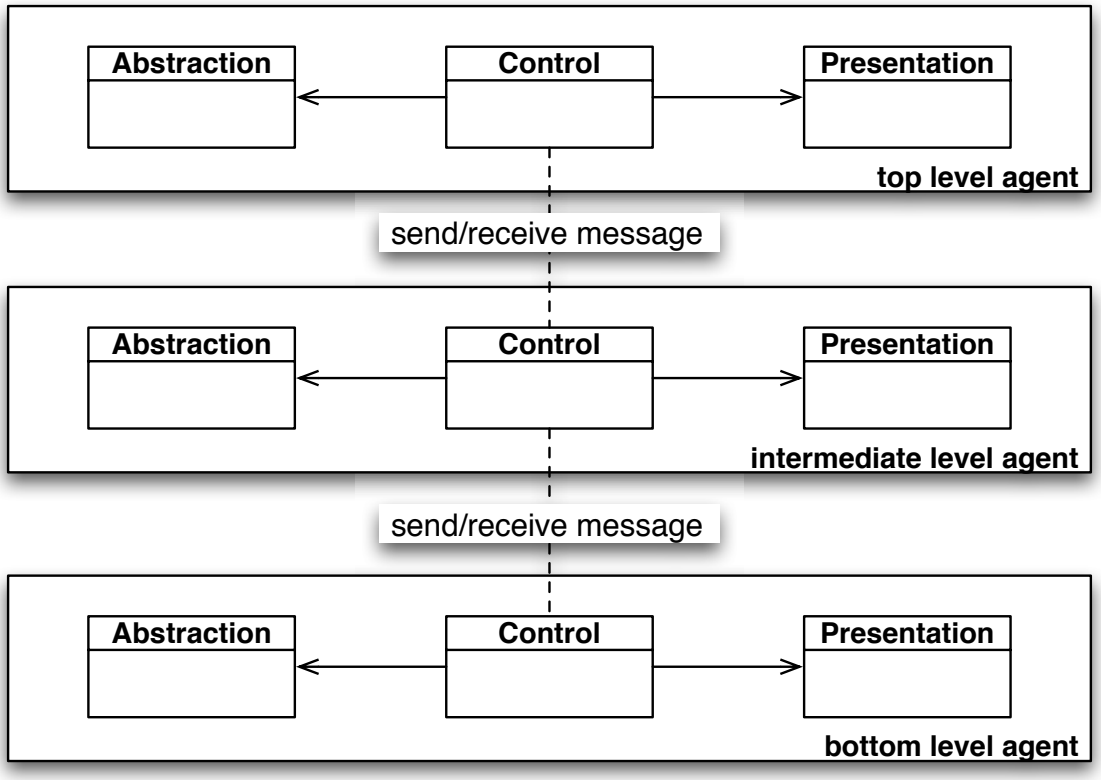

Abbildung 28: Presentation Abstraction Control

PAC ist ein Architekturmuster und daher auf einer anderen Ebene angesiedelt als MVC. Eclipse GEF orientiert sich sicherlich mehr am MVC-Muster als an PAC. Allerdings entspricht die Gewichtung der `EditParts` sehr den *Controllern* aus PAC.

Wenn hier auch keine exakte Umsetzung des PAC-Musters vorliegt, so ist folgendes Diagramm, das neben Eclipse GEF auch die „Eclipse Rich Client Platform“ (siehe auch Kapitel 4.2.1, S. 67) miteinbezieht, doch zumindest als Verständnishilfe interessant. Es versucht, die PAC-Agentenstruktur auf die Plug-In-Architektur von Eclipse und zugleich auf die MVC-Implementierung von Eclipse GEF zu übertragen:

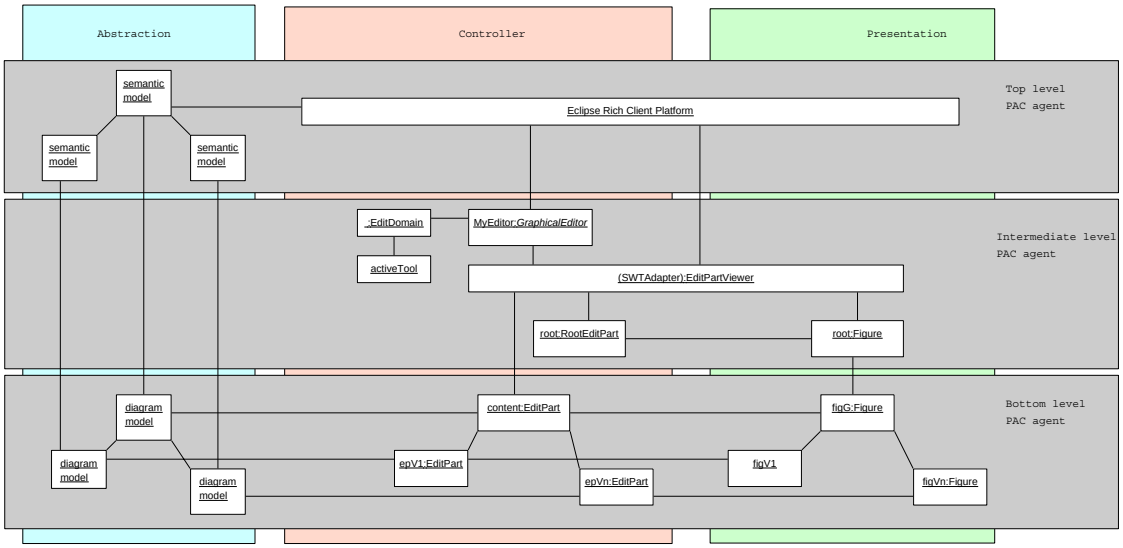

Abbildung 29: Eclipse GEF und RCP, Anwendung des PAC Musters

Nebeneinander sind hier die Rollen *Abstraction*, *Controller* und *Presentation* aufgeführt, die ungefähr den Rollen *Modell*, *Controller* und *View* des MVC-Musters entsprechen. Untereinander angeordnet sind die unterschiedlichen Agentenebenen. Die Eclipse Rich Client Platform ist hier als Top-Level Agent, die Klassen EditDomain und die Zeichenebene (RootEditPart etc.) sind als Intermediate-Level Agenten und die Komponenten der Zeichnung selbst als Bottom-Level Agenten interpretiert.

Zusammenfassend könnte eine Hierarchie der Frameworks nach Trennung von Modell und Darstellung erstellt werden: JHotDraw verzichtet auf diese Trennung. Der Vorteil ist eine einfache Klassenstruktur und ein schnell zu erlernendes Framework. Tigris GEF ähnelt in diesem Punkt JHotDraw, da es ebenfalls keine konsequente Trennung von Modell und Darstellung vornimmt. Die Erwähnung des MVC-Musters in der Literatur zu Tigris GEF ist dem Verständnis des Frameworks eher hinderlich, da dem Leser hierdurch ein falscher Eindruck vermittelt wird. Tigris GEF unterstützt ein Modell nur insoweit, als es die Struktur der Modellklassen über die Fassadenklasse `GraphModel` in die Struktur der *View*-Klassen übersetzen hilft. Eclipse GEF vollzieht eine konsequente Trennung von Darstellung und Modell. Dazu werden vor allem die *Controller*-Klassen stärker gewichtet.

3.2.2 Ereignisverarbeitung

Im Mittelpunkt der Ereignisverarbeitung finden sich bei allen Frameworks *Tools* und *Commands* (Befehle). Wie in Kapitel 2.5 (S. 12) bereits beschrieben, können *Tools* als

Zustand des gesamten Editors interpretiert werden, das heißt der Editor kann sich beispielsweise im Zustand „Markieren“ oder „Knoten erzeugen“ befinden. Tatsächlich ist dies bei allen Frameworks analog dem Zustandsmuster (vgl. [Gamma+96, 398ff]) umgesetzt. Daneben werden das Befehls- und Strategiemuster ([Gamma+96, 273ff u. 379ff)]) verwendet. Das Strategiemuster wird von JHotDraw und Eclispe GEF verwendet, allerdings nur von letzterem im Rahmen der Ereignisverarbeitung.

Das folgende Diagramm zeigt die Struktur dieser drei Muster im Vergleich:

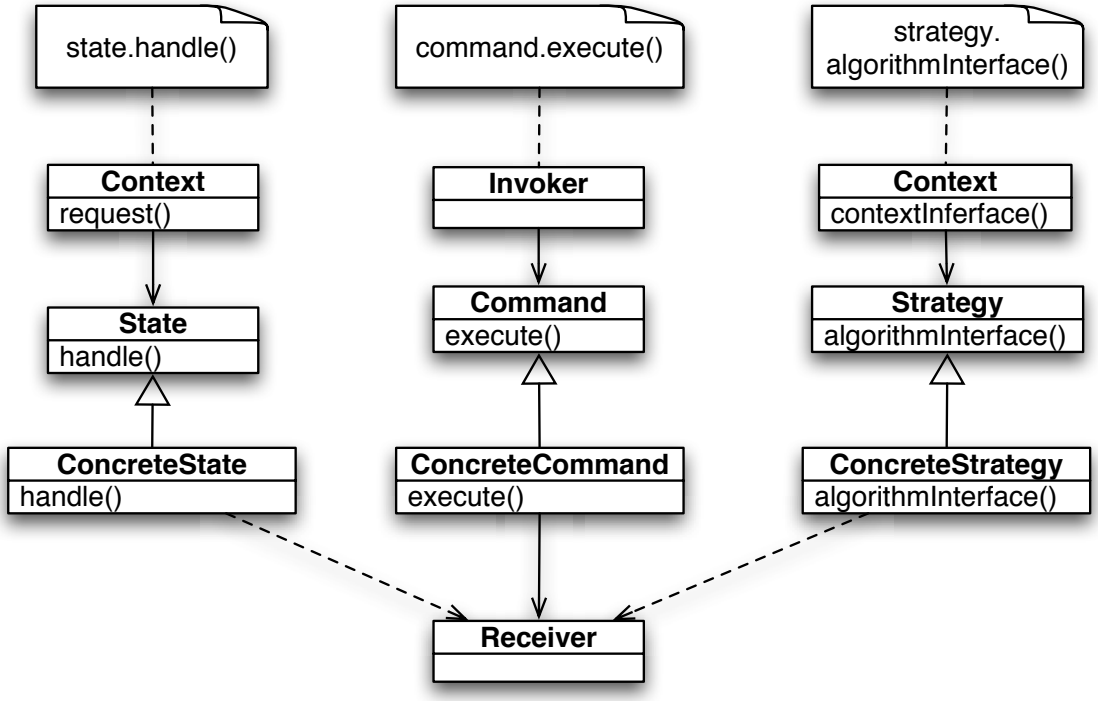

Abbildung 30: Verhaltensmuster im Vergleich

Dieses Diagramm zeigt ein Problem, welches bei der Analyse auftritt: Entwurfsmuster sind nicht eindeutig an ihrer Struktur zu erkennen. Faktisch ist nämlich die Struktur aller drei Muster gleich. Neben der Benennung der Klassen gibt das Verhalten der Frameworks Aufschluss darüber, welche Muster konkret eingesetzt wurden. Das Verhalten wurde hier in den vorhergehenden Abschnitten anhand eines einheitlichen Beispiels untersucht und in den jeweiligen Sequenzdiagrammen festgehalten. Alle drei Muster ermöglichen Verhaltensänderungen des Frameworks bei Berücksichtigung des „Inversion of Control“-Prinzips (vgl. etwa Sweet95]).

Das heißt, ein statischer Kontext oder vom Client nicht modifizierbarer Aufrufmechanismus ruft eine Client-Klasse auf, die an die jeweiligen speziellen Bedürfnisse der konkreten Anwendung angepasst ist und entsprechend dem jeweiligen Zustand der Anwendung durch Klassen mit gleichen Schnittstellen ausgetauscht werden kann. Im Ergebnis können die folgenden Klassen den jeweiligen Rollen der Muster zugeordnet werden:

	Zustandsmuster		
	Context	**State**	**Concrete State**
JHotDraw	*framework.DrawingView*	*framework.Tool*	*standard.ConnectionTool*, *standard.CreationTool*, *standard.SelectionTool* u.a., insgesamt ca. **15** *Tools*
Tigris GEF	*base.Editor* (über *base.ModeManager*)	*base.Tool*	*base.ModeCreateEdge*, *base.ModePlace*, *base.ModeSelect* u.a., insgesamt ca. **18** *Modes*
Eclipse GEF	*gef.EditDomain*	*gef.Tool*	*tools.CreationTool*, *tools.ConnectionCreationTool*, *tools.SelectionTool* u.a., insgesamt ca. **15** *Tools*

Tabelle 2: Zustandsmuster, Klassen und ihre Rollen

	Befehlsmuster		
	Invoker	**Command**	**Concrete Command**
JHotDraw	AWT-Eventmechanismus	`util.Command`	`util.CopyCommand`, `util.AlignCommand` u. a., insgesamt ca. **13** *Befehle*
Tigris GEF	`base.Editor` und AWT-Eventmechanismus	`base.Cmd`	`base.CmdCreateNode`, `base.CmdCopy` u . a., insgesamt ca. **50** *Befehle*
Eclipse GEF	`commands.CommandStack`	`commands.Command`	keine vordefinierten *Commands*[16]
	SWT-Eventmechanismus	`jface.actionAction`	`ui.actions.ZoomInAction`, `ui.actions.PrintAction` u. a., insgesamt ca. **30** *Actions*

Tabelle 3: Befehlsmuster, Klassen und ihre Rollen

	Strategiemuster		
	Context	**Strategy**	**Concrete Strategy**
Eclipse GEF	`gef.EditPart`	`gef.EditPolicy`	`editpolicies.XYLayoutEditPolicy`, `editpolicies.GraphicalNodelEditPolicy` u. a., insgesamt ca. **19** *Policies*.

Tabelle 4: Strategiemuster, Klassen und ihre Rollen

[16] Zwar gibt es noch weitere Command-Klassen, die aber nur Hilfsklassen sind, etwa CompoundCommand für zusammengesetzte Befehle – die eigentlich ausführenden Befehle sind nicht vorgegeben.

An der einfachen Summenbildung aller zur Verfügung gestellten konkreten Klassen, also der Summe der konkreten Befehls-, Zustands- und Strategieklassen, lassen sich die unterschiedlichen Funktionsumfänge der Frameworks deutlich ablesen: JHotDraw bietet nur ca. 30 Klassen an, während Tigris GEF und Eclipse GEF jeweils ca. 70 Klassen bereitstellen. Da es sich um Klassen handelt, die konkretes Verhalten modellieren, kann, was nicht allgemein gilt, indirekt auf den Funktionsumfang rückgeschlossen werden.[17]

Wie arbeiten nun die Muster und die jeweiligen Klassen in den Frameworks zusammen? Obwohl alle drei Frameworks die gleichen Muster und zum Teil sogar identische Klassennamen verwenden, unterscheiden sich die Umsetzungen gravierend.

Die Frage, wann welches Muster verwendet wird, hängt mit der Reihenfolge der Auswahl der *Receiver*-Objekte – im Fall graphischer Editoren sind das die Komponenten-repräsentierenden Klassen, zusammen. Interessanterweise wird in [Gamma+96] nur im Fall des Befehlsmusters in der Struktur ein *Receiver* aufgeführt. Natürlich benötigen im Fall der hier untersuchten Frameworks auch *State*- oder *Strategy*-Objekte Daten, auf denen die jeweiligen Operationen bzw. Algorithmen ausgeführt werden können.

JHotDraw verwendet alternativ entweder das Zustands- oder das Befehlsmuster zur Durchführung von Benutzeraktionen. Das Befehlsmuster wird hier verwendet, wenn zum Zeitpunkt der Auswahl des *Befehls* – und hier ist das identisch mit der Ausführung des *Befehls* – alle *Receiver*-Objekte bekannt sind oder die gesamte Zeichnung als *Receiver* verwendet wird. Dies ist der Fall etwa bei Copy- und Paste-Aktionen, beim Anordnen (oder Ausrichten) von Komponenten oder beim Speichern. Das Zustandsmuster wird verwendet, wenn die *Receiver*-Objekte noch nicht bekannt sind. Dies ist etwa beim Erstellen neuer Komponenten der Fall, denn das *Receiver*-Objekt ist ja dann gerade das Ergebnis der Funktion. Ähnliches gilt für die Auswahl (Selektion) der Komponenten selbst. Beide Muster werden hier also unabhängig, fast komplementär verwendet.

Bei Tigris GEF arbeiten beide Muster zusammen. Hier führen Benutzeraktionen zunächst immer zur Aktivierung eines Befehlsobjekts. Falls der Befehl auf bereits

---

[17] Die einfache Addition dieser Klassen ist nicht ganz zulässig, wie in der weiteren Analyse gezeigt wird, aber als Richtwert trotzdem brauchbar.

bekannten *Receivern* ausgeführt werden kann, entspricht das Vorgehen dem von JHotDraw. Die Anwendungsfälle sind identisch. Der *Befehl* wird jedoch auch dann ausgeführt, wenn die *Receiver*-Objekte noch nicht bekannt sind, etwa im Falle des Erstellens von Komponenten. In diesen Fällen wird der Zustand des Editors vom Befehl geändert; mit anderen Worten: Ein *Mode*-Objekt wird erzeugt. Dieses ist dann für die weitere Durchführung der Funktion wie bei JHotDraw verantwortlich.[18]

Eclipse GEF kennt, wie in der Tabelle zu sehen, zwei verschiedene Umsetzungen des Befehlsmusters. Eine Umsetzung ist die der SWT-Widgetbibliothek, diese verwendet von der Klasse `Action` abgeleitete *Befehle*. Diese *Befehle* sind ähnlich wie bei JHotDraw oder Tigris GEF auf bereits selektierte *Receiver*-Objekte angewiesen, sie werden über Menüeinträge und Schalter der Eclipse-Rahmenanwendung aktiviert. Die Eclipse GEF-spezifische Umsetzung des Befehlsmusters basiert auf Befehlen, die von der Klasse `Command` abgeleitet werden. Diese „interne“ Umsetzung arbeitet eng mit den Strategie- und Zustandsklassen zusammen. Folgendes Diagramm zeigt, wie die Muster miteinander verzahnt sind:

[18] Es sei nur kurz angemerkt, dass diese Lösung problematisch ist: Da das Befehlsobjekt keinen Context hat, kann das vom Befehl erzeugte State-Objekt einem solchen auch nicht zugeordnet werden. Das State-Objekt wird zunächst in einer globalen Hilfsvariablen gespeichert, um dann nachträglich einen Context zugewiesen zu bekommen. Auch gibt es Fälle, in denen das State-Objekt im späteren Verlauf das Befehlsobjekt aufruft, dabei allerdings keine allgemeine Schnittstelle verwendet.

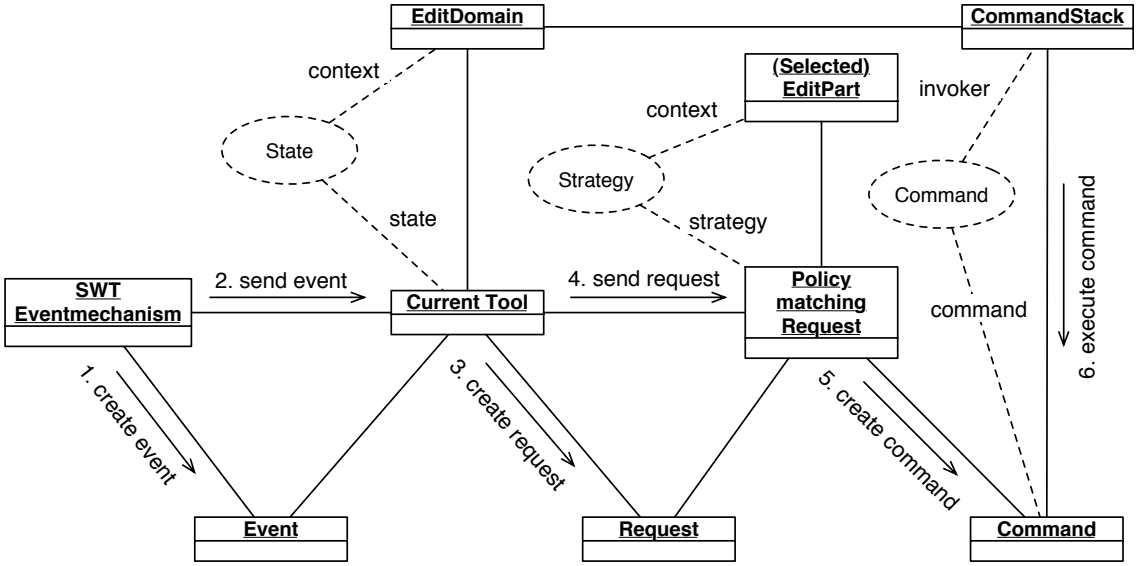

Abbildung 31: Eclipse GEF Kommunikationsdiagramm, Kombination der Verhaltensmuster in der Ereignisverarbeitung

Wie aus der Tabelle ersichtlich, werden bei Eclipse GEF keine konkreten `Commands` mitgeliefert. Diese operieren ausschließlich auf dem Modell und müssen daher vom Client implementiert werden. Die `EditParts` werden über die `EditPartFactory` passend zu den Modellobjekten erzeugt und bei Veränderungen der Modellobjekte als Beobachter benachrichtigt – eine direkte Manipulation der `EditParts` durch die `Commands` ist daher weder erwünscht noch notwendig.

3.2.3 Constraints

Constraints wurden oben bereits als spezielle Beziehungen zwischen Komponenten bzw. deren Zustandsvektoren eingeführt. Alle drei Frameworks verzichten auf allgemeine *Constraints*, setzen aber Verfahren wie *Router*, *Locator*, *Layouts* oder *Anchor* um. Dabei kommen zwei Muster zum Einsatz: das Beobachtermuster und das Strategiemuster. Bei Änderungen der Eigenschaften einer Komponente wird die von ihr abhängige Komponente über das Beobachtermuster benachrichtigt. Die notwendige Anpassung der Eigenschaften wird dann entweder über eine *Strategie* oder innerhalb der Komponenten selbst berechnet. Die Komponente nimmt dabei die Rolle des *Context* ein und das spezifische Verfahren, etwa der *Locator*, wird als *Strategie* modelliert. Die folgende Tabelle listet die jeweiligen Konzepte und die in der Umsetzung beteiligten Klassen, der jeweiligen Rolle des Strategiemusters zugeordnet, auf:

	Rolle	JHotDraw	Tigris GEF	Eclipse GEF
Router	*Context*	–	–	`ConnectionFigure`
	Strategy	–	–	`Router`
Anchor	*Context*	`ConnectionFigure`	–	`Connection`
	Strategy	`Connector`	–	`ConnectionAnchor`
Locator	*Context*	`Figure`	–	`draw2d.Figure`
	Strategy	`Locator`	–	`draw2d.Locator`
Grid	*Context*	`DrawingView`	`Editor`	`EditPart`
	Strategy	`PointConstrainer`	`Guide`, `GuideGrid`	`SnapToHelper`, `SnapToGrid`

Tabelle 5: Klassen und ihre Rollen im Strategiemuster

Wie zu erkennen ist, verzichtet Tigris GEF größtenteils auf den Einsatz von Strategien. Die Anpassung der Eigenschaften wird in den jeweiligen Klassen (die in den anderen Frameworks als *Context* auftauchen) selbst vorgenommen. Wenn Client-seitig ein anderes Verhalten gewünscht wird, müssen also bei Tigris GEF entsprechend Unterklassen gebildet werden.

Etwas verwirrend kann beim Einsatz des Strategiemusters in den Frameworks der Lifecycle der Strategieobjekte sein. Beispielsweise werden *Strategie*-Objekte nicht an die *Komponente* angemeldet, deren Verhalten sie erweitert, und dann im Bedarfsfall aufgerufen. Sondern sie werden von den Komponenten auf Anfrage spontan erzeugt, also bei jeder Anfrage neu. Der Vorteil ist, dass die Strategieobjekte damit zustandslos implementiert werden können.

3.3 Fazit

Als ein erstes Ergebnis des Vergleichs kann man die Frameworks gemäß ihrer Komplexität anordnen. JHotDraw ist aufgrund seiner wenigen Klassen das

übersichtlichste Framework, Eclipse GEF das komplexeste, Tigris GEF siedelt sich hier in der Mitte an.

Ein differenzierteres Ergebnis liefert die Betrachtung der Umsetzung und Anwendung der Entwurfsmuster. Bei Eclipse GEF ist die Umsetzung des MVC-Musters mit der Trennung von Modell und Darstellung am konsequentesten realisiert. Bei Tigris GEF ist diese nicht vollkommen umgesetzt, obwohl hier laut Dokumentation das MVC-Muster verwendet wurde. Tatsächlich ähnelt Tigris GEF in dieser Hinsicht eher JHotDraw, welches den Einsatz des MVC-Musters erst gar nicht für sich proklamiert. Sowohl Eclipse GEF als auch JHotDraw sind durch entsprechende Muster einfach erweiterbar, Tigris GEF hingegen erfordert hierzu eine intensive Einarbeitung sowie die Anpassung der Client-Klassen an das Framework.

Neben der Güte des Frameworks, die hier über das Instrument „Entwurfsmuster“ zu erfassen versucht wurde, stellt bei der Auswahl eines Frameworks die Qualität der Dokumentation ein entscheidendes Kriterium dar. Ein Problem, vor allem bei Tigris GEF, ist die knappe Dokumentation. Die in Kapitel 3.1 aufgeführten Diagramme mussten bei allen Frameworks größtenteils durch aufwendiges Reverse Engineering erstellt werden.

Für die Umsetzung des UML-Editors ist die Trennung von Darstellung und Modell sehr wichtig, da das UML-Modell sehr komplex ist, bereits eine Implementierung dieses Modells vorliegt (vgl. Kapitel 4.2.3, S. 73) und es wünschenswert ist, dieses einfach zu integrieren. Nicht zuletzt deshalb wird für die Erstellung des Editors für UML-Diagramme im folgenden Kapitel Eclipse GEF als graphisches Framework ausgewählt.

4 Modellierung des UML-Editors

Die im ersten Teil vorgestellten Konzepte und Frameworks sollen nun im Rahmen eines Prototyps auf ihre Tragfähigkeit hin geprüft werden.

Auf der Basis der in den vorhergehenden Abschnitten erarbeiteten Konzepte stellt das folgende Kapitel die Implementierung eines graphischen Editors für UML-Diagramme vor. Bei der Umsetzung des Editors, im folgenden **GUMLE** für „**G**raphischer **UML**-**E**ditor“ genannt, kommen zwei Anwendungsbereiche zusammen: graphische Editoren und UML.

Im ersten Teil wurde die Trennung von Darstellung und Modell bereits vorgestellt. Das im folgenden verwendete Framework, Eclipse GEF, setzt diese Trennung um. Es stellt selbst die Infrastruktur für den graphischen Editor bereit. Das Modell wird nun weiter ausgearbeitet, dazu wird auf weitere Frameworks zurückgegriffen.

Die Modellierung des Prototyps ist grob in die Bereiche Analyse, Entwurf und Implementierung unterteilt. Aufgrund der besonderen Bedingungen sind diese Bereiche nicht im klassischen Sinne ausgeführt und haben hier eher strukturierenden als formalen Charakter.

4.1 Graphische Notationen

Anstelle einer vollständigen Anforderungsanalyse, die den Rahmen dieser Arbeit überschreiten würde, sollen zunächst Kernfunktionen herauskristallisiert werden, die aus technischer Sicht von Interesse sind. Diese werden dann bei der Realisierung des Prototyps umgesetzt.

Als Basis für die allgemein benötigten Funktionen und Notationen werden zwei Quellen herangezogen:

1. Kurs 1793 „Software Engineering“ der Fernuniversität Hagen [Six+02]
2. UML 2.0 Spezifikation [UML2.0S] und [UML2.0I]

Im Rahmen der Umsetzung des Prototyps werden die wichtigsten Funktionen für Klassendiagramme mit folgenden Komponenten implementiert:

- Klassen mit Attributen und Operationen
- Generalisierungsbeziehung

- Assoziationsbeziehung mit fester Beschriftung
- Kommentare (Notes)

4.1.1 Beschreibung der Notation

Der folgende Abschnitt beschreibt die graphische Notation, wie sie im Prototyp realisiert ist.

Die im Klassendiagramm benötigten Elemente „Interface“ und „Klasse“ sind im Modell von UML 2.0 (im folgenden einfach UML) Spezialisierungen der allgemeinen Klasse *Classifier*. Dieser Typ ist abstrakt und kann daher weder instanziiert noch dargestellt werden. Allerdings können alle von *Classifier* abgeleiteten Klassen auf gleiche Weise (neben evtl. anderen optionalen Darstellungen) in *Classifier-Notation* notiert werden. Die Definition dieser allgemeinen Notation erfolgt hier über diese Klasse. Falls abgeleitete Klassen eine andere Darstellung verlangen, müssen die hier definierten Merkmale entsprechend geändert (*überschrieben*) werden.

Die Notation folgt der Spezifikation [UML2.0S, 64]. Bei der Umsetzung werden UML-Diagramme als Graphen interpretiert. Dabei sind alle UML-Klassen, die sich als Knoten interpretieren lassen, von der Metaklasse *Classifier* abgeleitet und werden mittels der Classifier-Notation dargestellt. Der Name ist in einem eigenen *Compartment* enthalten. Damit besteht die Notation aus einem Rechteck, das mehrere *Compartments* enthält. Attribute werden im *Attribut-Compartment* untereinander aufgeführt. Die Eigenschaften der Attribute werden in OCL-Schreibweise (Object Constraint Language, vgl. [OCL2.0]) angegeben. Wenn der Platz im *Compartment* nicht zur Darstellung aller Attribute oder Operationen ausreichend ist, werden die Attribute soweit möglich dargestellt und bei Bedarf eine Scrollbar eingeblendet. Operationen werden analog in einem Operation-Compartment in OCL-Schreibweise aufgeführt (vgl. [UML2.0S, S. 78]).

Die *Generalisierung* wird als einfache Linie mit einem leeren Dreieck an der Seite des *generellen Classifiers* dargestellt. Es werden nur *binäre Assoziationen* unterstützt. Diese werden als einfache Linien dargestellt, wobei der Name der Assoziation automatisch an der Mitte der Linie angezeigt wird. Weitere Beschriftungen sind im Prototyp nicht vorgesehen.

Die Enden der Linie können weiter ausgezeichnet werden:

- ein Pfeil markiert einseitige *Navigierbarkeit*
- eine ausgefüllte Raute (Diamant) steht für *Komposition*
- eine leere Raute kennzeichnet *Aggregation*

Alle beschriebenen Elemente (*Klassen*, *Attribute*, *Operationen*, *Generalisierungen*, *Assoziationen* etc.) können mit *Notes (Kommentaren*) verbunden werden. Die Verbindung wird über eine gestrichelte Linie angezeigt und im Folgenden als Comment-Link bezeichnet. Der *Kommentar* wird als Kasten mit abgeknickter Ecke dargestellt.

Abbildung 32 zeigt die graphischen Notationen um Überblick:

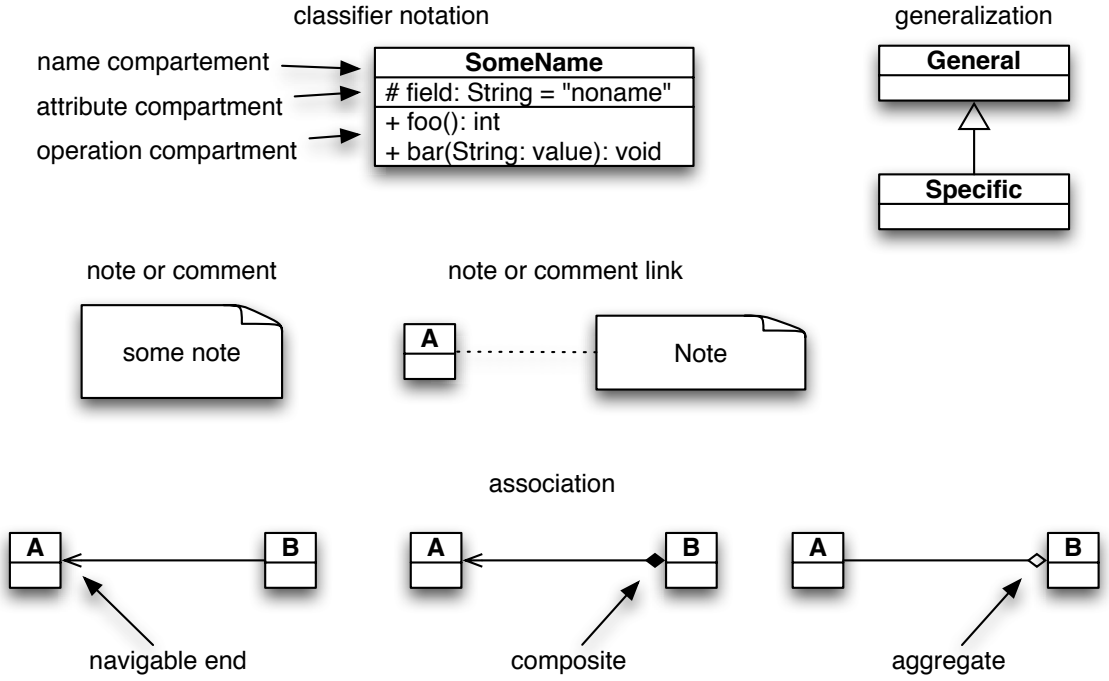

Abbildung 32: Umgesetzte Notationen der UML 2.0

4.1.2 Anforderungen an das Framework

Die oben beschriebene Notation zählt zu den typischen Anforderungen an graphische Frameworks. Da es sich bei dem zu erstellenden UML-Editor um einen Prototyp handelt, soll und muss hier keine Vollständigkeit im Sinne des Anwendungsbereichs erreicht werden. Hingegen sollen die möglichen technischen Probleme beurteilt werden können. Im Folgenden sollen die besonderen Anforderungen der ausgewählten Notationen und Elemente aus technischer Sicht erläutert werden. Die Beschreibung der

Auswahl der umzusetzenden Anforderungen enthält daher auch bereits Hinweise auf die Implementierung.

Die vorgestellten Frameworks teilen Diagramme in Knoten und Kanten auf. Mit *Klassen* und *Kommentaren* werden zwei verschiedene Knotentypen, mit *Generalisierung*, *Assoziation* und *Commentlink* drei Kantentypen implementiert. Der Benutzer muss dabei zwischen verschiedenen Typen wählen können, was über entsprechende *Tools* umgesetzt werden kann.

Die Verwendung von *Compartements* und deren Umsetzung im Editor zeigt, wie Komponentenhierarchien implementiert werden können. Interessant ist dabei die Abbildung des UML-Modells auf die Modellklassen innerhalb des MVC-Musters. Das Modell von UML definiert keine *Compartements*, was in der Abbildung der Modellhierarchie auf die Komponentenhierarchie berücksichtigt werden muss.

Die Umsetzung zweier verschiedener Relationen (*Generalisierung* und *Assoziation*) zeigt, wie unterschiedliche *Connections* umgesetzt werden. Außerdem können hier semantische Beschränkungen auftreten, die vom Editor berücksichtigt werden müssen. So sind etwa zirkuläre Vererbungsbeziehungen (Klasse A ist Oberklasse von Klasse B, Klasse B ist Oberklasse von Klasse A) nicht möglich. Die unterschiedlichen Linienenden der Assoziation zeigen beispielhaft den Umgang mit *Decorations*.

Kommentare selbst sind sehr einfache Komponenten. Interessant sind allerdings die Verbindungen der Kommentare mit den jeweils annotierten Elementen. *Kommentare* können mit allen Elementen (außer mit anderen *Kommentaren*) verbunden werden. Insbesondere können *Kommentare* auch mit anderen Kanten und Kinderelementen von Komponenten (etwa *Attributen*) verknüpft sein, was besondere Anforderungen an den Entwurf stellt.

Die implementierten Notationen können als prototypische Vorlagen für weitere, hier nicht umgesetzte Elemente dienen. So sind alle hier nicht umgesetzten Strukturdiagramme (etwa Objekt- oder Paketdiagramme) dem Klassendiagramm sehr ähnlich. Weitere, auf einem XY-Layout basierende Verhaltensdiagramme (Use-Case- oder Aktivitätsdiagramm) sind ebenso in Analogie zum Klassendiagramm zu implementieren. Lediglich Sequenzdiagramme und in der UML 2.0 neu eingeführte Timingdiagramme stellen weitere, hier nicht untersuchte, technische Anforderungen in Form von speziellen Layouts.

4.2 Entwurf

Eclipse (http://www.eclipse.org) wird häufig mit einer Java-Entwicklungsumgebung identifiziert. Tatsächlich ist Eclipse jedoch ein allgemeiner Applikationskern, der mittels Plug-Ins erweitert werden kann. Bei der Java-Entwicklungsumgebung handelt es sich „lediglich“ um ein solches Plug-In (Java Development Tools, kurz JDT). Auch der hier entwickelte Prototyp wird als Eclipse Plug-In realisiert.

Die bereits in Kapitel 3.2.1 („Model-View-Controller“, S. 48) vorgestellte Trennung von semantischem und graphischem Modell wird nun umgesetzt. Das semantische Modell enthält die Daten des Anwendungsbereichs, in diesem Fall der UML 2.0. Das graphische Modell speichert Daten, die zur Darstellung der Elemente des semantischen Modells im Diagramm notwendig sind, etwa Positionsangaben, Größe, Farbe etc. Zur Realisierung des semantischen Modells wird das „Eclipse Modeling Framework“ (im Folgenden EMF, vgl. [EMF]) bzw. die darauf aufsetzende „EMF-based UML 2.0 Metamodel Implementation“ (im Folgenden UML2, vgl. [UML2]) eingesetzt, welche ebenfalls im Rahmen des Eclipse Projekts entwickelt werden. Das graphische Modell wird über eine eigene, auf EMF basierende Implementierung der „UML Diagram Interchange“-Spezifikation [UML2.0DI], im Folgenden UML-DI, umgesetzt.

Die Verwendung des Frameworks Eclipse GEF legt die Architektur der Anwendung, bis auf das Modell, fest. Da das Modell ebenfalls mittels Einsatz eines Frameworks erstellt wird, ist auch hier die grundlegende Architektur festgelegt. Im Folgenden werden daher die Architekturen der gewählten Frameworks, die Gründe für die Auswahl der Frameworks und deren Zusammenspiel beschrieben.

4.2.1 Eclipse Rich Client Platform

Grundlegender Rahmen für die Entwicklung des Prototyps ist die „Eclipse Rich Client Platform“, kurz RCP (vgl. [RCP]). Der Kern der Eclipse Platform, der so genannte Runtime-Kernel, besteht im Wesentlichen aus einem Mechanismus zum Laden von Plug-Ins. Die RCP verwendet nicht Java-Swing als Widgetbibliothek, sondern eine eigene, ebenfalls im Rahmen des Eclipse Projekts entwickelte Bibliothek – das „Standard Widget Toolkit“, kurz SWT (vgl. [SWT]).

Ein Plug-In wird über eine Deskriptordatei „plugin.xml“ in die Platform eingebunden. Das Eclipse Plug-In-Konzept basiert auf Erweiterungspunkten (Extension Points), die

von Plug-Ins bereitgestellt oder erweitert werden können. Ein bestimmter Erweiterungspunkt bietet die Möglichkeit, eigene Editoren einzubinden, die dann in die Benutzeroberfläche integriert werden. Das Komponentendiagramm zeigt, wie GUMLE als Plug-In in die RCP eingebunden wird.

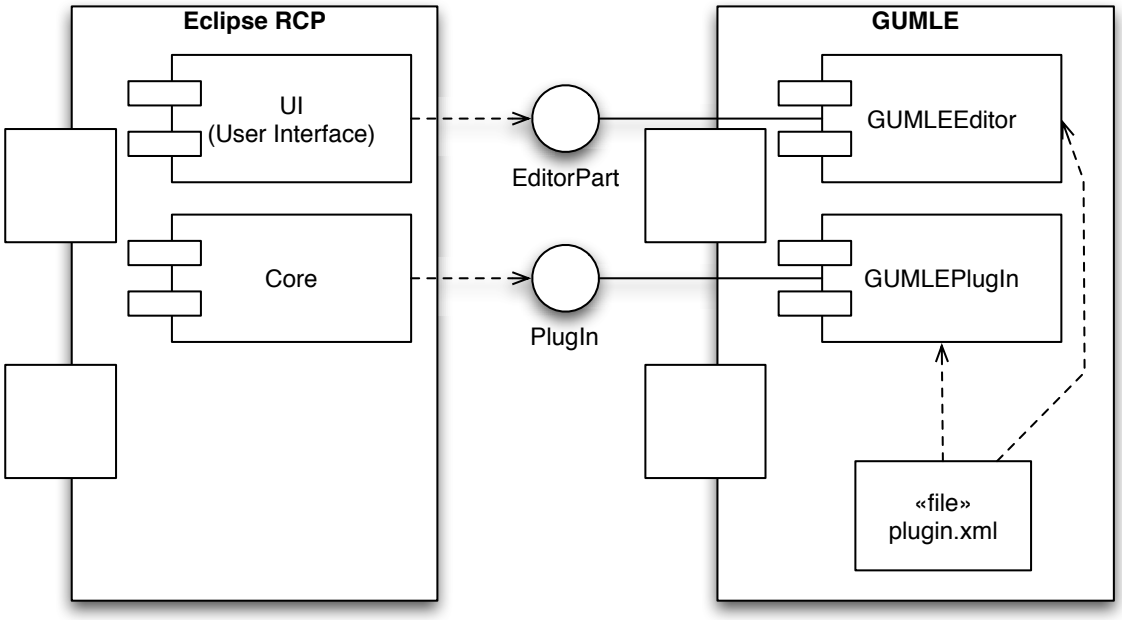

Abbildung 33: Komponentendiagramm GUML, Einbindung als Plug-In

Der Vorteil dieser Technik ist, dass die Platform die für graphische Benutzeroberflächen notwendige Infrastruktur wie Menü-, Status- und Toolbar oder das Hilfesystem bereitstellt. Neben diesen typischen Elementen graphischer Benutzeroberflächen stellt die Platform mehrere spezialisierte Ansichten und Fenster bereit, etwa eine „Navigator View“, die eine baumbasierte Ansicht der Ressourcen enthält und im Fall von GUMLE auch die Namen der angelegten Diagramme auflistet.

Der gezeigte Ausschnitt der Deskriptor-Datei für GUMLE zeigt, dass zwei „extension points“ erweitert werden, einer zum Einbinden des graphischen Editors, ein anderer, um einen *Wizard* zum Erstellen neuer Klassendiagramme bei der Platform anzumelden.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin name="Graphical UML Editor"
    ...
    <extension point="org.eclipse.ui.editors">
      <editor
            class="de.fernuni.gumle.GUMLEditor"
            ...>
      </editor>
   </extension>

    <extension point="org.eclipse.ui.newWizards">
      ...
      <wizard
            class="de.fernuni.gumle.wizards.ClassDiagramWizard"
            ...>
         ...
      </wizard>
   </extension>
</plugin>
```

Codebeispiel 13: plugin.xml, GUMLE Deskriptordatei, Ausschnitt

Eine ausführliche Beschreibung der Eclipse RCP und des Plug-In-Konzepts ist in [Beck+03] zu finden.

4.2.2 Eclipse Modeling Framework

> "EMF is a powerful, practical technology for implementing object models from formal model definitions [..]" [Budinsky+03, S. xix]

Der Begriff des *Modells* ist bereits im Zusammenhang mit dem MVC-Muster erörtert worden. Warum wird nun hier ein zusätzliches Framework zur Implementierung des *Modells* eingesetzt?

Im Rahmen des objektorientierten Designs besteht ein Modell, zumeist erstellt mit Hilfe von UML-Klassendiagrammen, aus miteinander verbundenen Klassen. Bei der Programmierung müssen diese Klassen und ihre Beziehungen mit den Mitteln der Programmiersprache, in diesem Fall Java, umgesetzt werden. Dabei müssen einige

Konstrukte der UML durch Hilfskonstruktionen in Java ersetzt werden. Ein typisches Beispiel hierfür sind bidirektionale Assoziationen oder Assoziationen höherer Kardinalität, etwa 0 zu n. Bei der Bidirektionalität sind Konsistenzbedingungen zu beachten, deren Einhaltung der Anwendungscode überwachen muss. In Java sind nur feste Kardinalitäten in Form einfacher Variablen oder Felder fester Größe möglich, variable Feldgrößen werden nicht unterstützt, hier müssen Hilfsklassen (Collections) verwendet werden. Der Grund dieser Probleme ist, dass die Metamodelle von Java und UML unterschiedliche Fähigkeiten haben. Der Begriff des Metamodells soll an dieser Stelle nicht weiter erläutert werden, hier sei auf die Literatur (etwa [UML2.0I] oder auch [Frankel03]) verwiesen. Das Metamodell von UML ist die Meta Object Facility, kurz MOF ([MOF2.0C]). Da UML zur Modellierung Client-spezifischer Modelle eingesetzt wird, wird UML auch als Metamodell beschrieben, so dass MOF auch als ein Meta-Metamodell bezeichnet werden kann.

EMF definiert ein *Metamodell* namens *ecore*, das MOF sehr ähnelt. Ein Modell in EMF ist nun eine Instanz von *ecore*. Mit anderen Worten: Ein EMF-basiertes Modell besteht aus Java-Klassen, die von der Klasse `EObject` im Packet `org.eclipse.emf.ecore` abgeleitet sind. Damit bietet EMF eine Brücke zwischen den Metamodellen von UML und Java. Dieser Zusammenhang wird in der folgenden Abbildung illustriert.

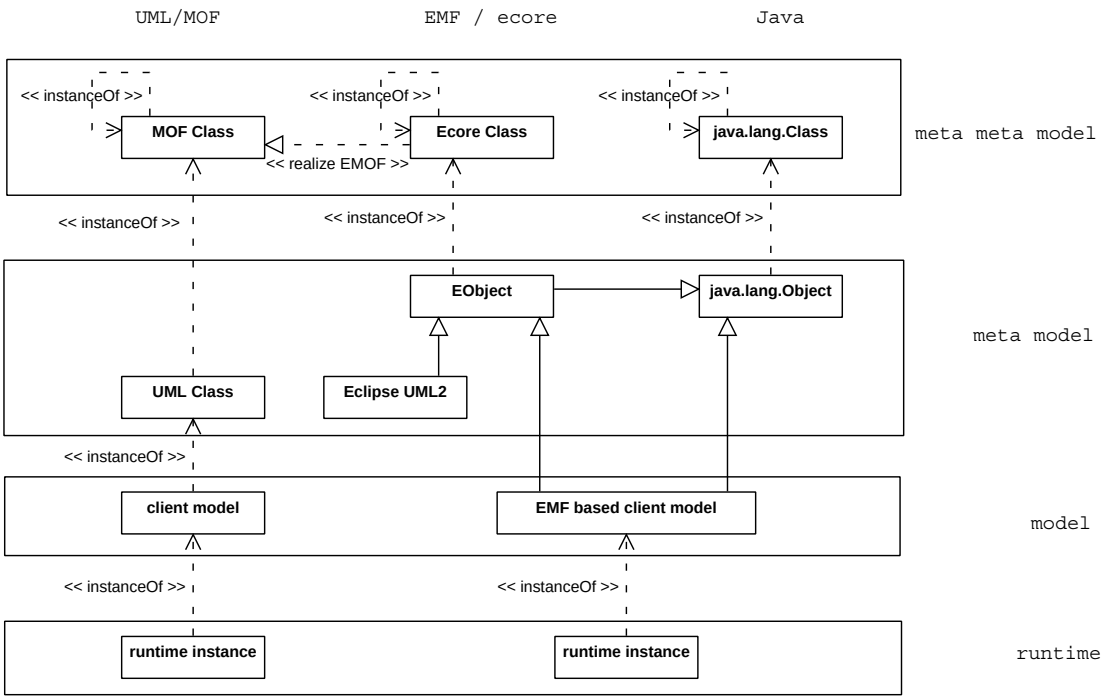

Abbildung 34: UML, MOF, EMF und Java - Modelle und Metamodelle

MOF kommt aus dem Umfeld der „*Model Driven Architecture*“ (MDA). Eines der Kernfeatures von MDA ist die automatische Codegenerierung aus Modellen heraus. EMF bietet genau diese Möglichkeit, das heißt mit EMF ist es möglich, per Mausklick aus einem vorhandenen Modell eine entsprechende Implementierung desselben zu erzeugen. EMF stellt einen einfachen baum-orientierten Editor zur Verfügung, mit dem Modelle definiert werden können. Alternativ dazu kann EMF auch Modelle aus Rational Rose einlesen oder annotierten Java-Code umwandeln; letzteres wurde hier angewendet.

Die generierte Implementierung besitzt unter anderem folgende Eigenschaften:

- Serialisierbarkeit entsprechend dem MOF-XMI-Mapping[19] [MOF-XMI]
- Implementierung des Beobachtermusters
- Zugriff auf Metadaten des Modells, etwa Typen von Elementen einer *Collection*

[19] XMI steht für „XML Metadata Interchange“, ein auf XML basierendes Datenformat zum Speichern von Objekten (vgl. [XMI2.0])

- Behandlung von beidseitig navigierbaren Assoziationen, Sicherstellung der Konsistenz

Diese Eigenschaften werden von GUMLE genutzt: So werden die Diagramme im XMI Format gespeichert, die Eclipse GEF Controllerschicht über das implementierte *Beobachtermuster* mit dem Modell verbunden und in der Eventverarbeitung die Metadaten des *Modells* genutzt (um etwa zu ermitteln, welche Eigenschaft eines Objekts geändert wurde). Dadurch vereinfachen sich die Prozeduren zur Modifikation des Modells erheblich, da die Konsistenz desselben durch die EMF-Implementierung bereits gewährleistet ist.

Die von EMF generierte Implementierung des Modells besteht aus Interfaces der definierten Modell-Klassen sowie einer Implementierung dieser Interfaces. Die Implementierungen können nur über eine ebenfalls generierte *Factory*, die als *Singleton* implementiert ist, erzeugt werden. Informationen der Metadaten, wie etwa Identifikatoren der Objekteigenschaften sind über eine Paket-Klasse erreichbar, die ebenfalls als *Singleton* implementiert ist.

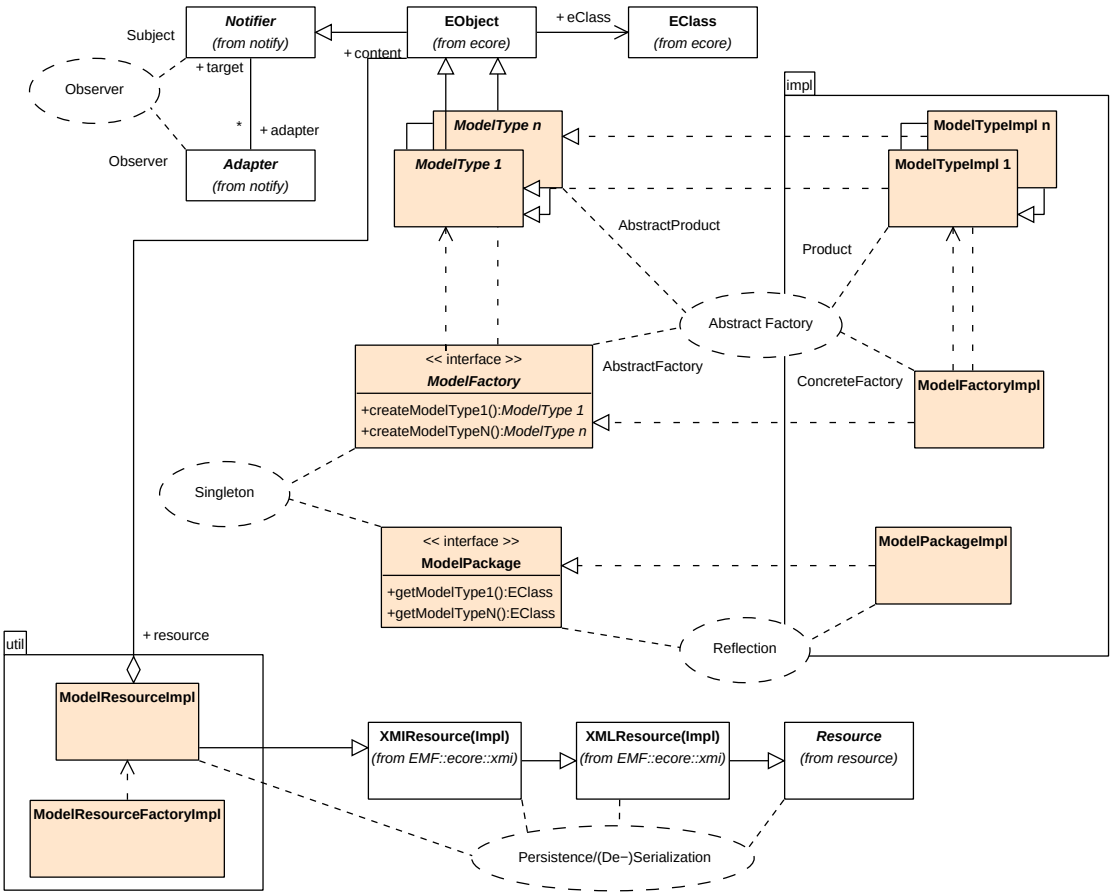

Abbildung 35: Klassendiagram EMF, API des generierten Modells [Pilgrim05, 82]

Da das Modell in GUMLE in einen semantischen und einen graphischen Teil zerlegt ist, kommt EMF hier gleich zweimal zum Einsatz. Das semantische Modell ist das Metamodell von UML 2.0, eine auf EMF basierende Umsetzung liegt bereits vor und wird im nächsten Kapitel beschrieben.

4.2.3 Eclipse UML2

Nun sollen UML-Diagramme im Editor bearbeitet werden. Das heißt die darzustellenden Elemente sind Instanzen der in der „UML 2.0 Infrastructur“ [UML2.0I] definierten Klassen. Das semantische Modell entspricht also gerade dem Modell von UML 2.0. Mit Eclipse UML2 ([UML2]) existiert bereits eine EMF-basierte Implementierung des UML-Modells. Diese setzt die über 200 Klassen des UML-Modells wie oben dargestellt als EMF-Implementierung um. Für den Prototyp werden davon nur die für das Klassendiagramm relevanten Klassen benötigt, eine sehr vereinfachte Übersicht ist nachfolgend abgebildet.

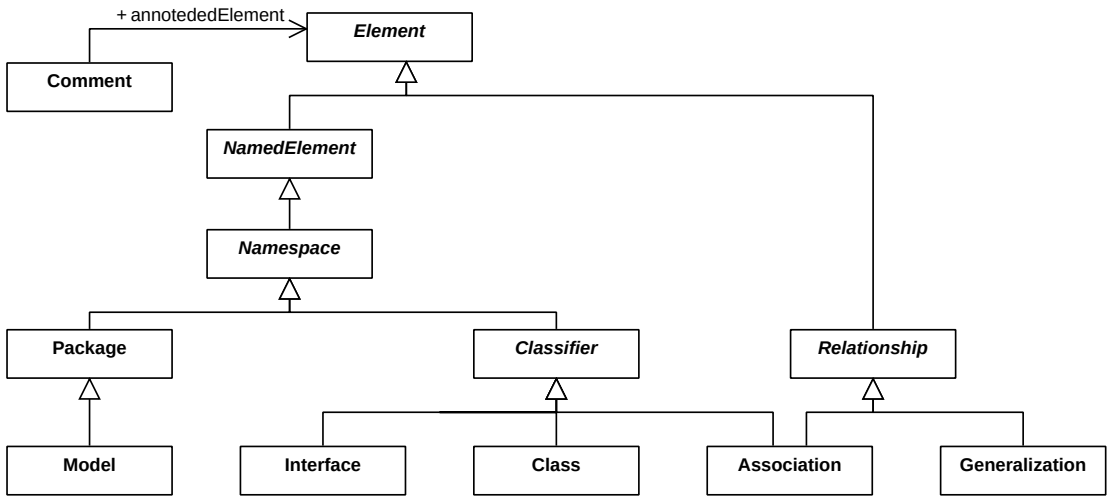

Abbildung 36: Klassendiagramm UML2, stark vereinfacht

UML2 implementiert neben der Struktur der Klassen des UML-Modells auch Beschränkungen, wie das bereits erwähnte Verbot zirkulärer Vererbungshierarchien. Damit ist bereits der semantische Anteil des Modells abgedeckt. Die in UML2 eingebaute Konsistenzprüfung sorgt dafür, dass nur semantisch gültige Modelle erzeugt werden können. Daher muss keine weitere Prüfung in den Editor eingebaut werden.

Da UML2 eine EMF-Modell-Implementierung darstellt, ist die API wie üblich aufgebaut. Die Factory ist unter `org.eclipse.uml2.UML2Factory`, die Paketklasse unter `org.eclipse.uml2.UML2Package` zu finden.

Das Modell von UML 2.0 enthält keine graphischen Informationen – schließlich handelt es sich bei UML um eine Sprache für Modelle und nicht für Diagramme. Daher müssen die graphischen Informationen, wie Position und Größe der in einem Diagramm abgebildeten Klassen, zusätzlich gespeichert werden.

4.2.4 UML Diagram Interchange

Prinzipiell gibt es zwei Möglichkeiten, die graphischen Informationen zum semantischen Modell hinzuzufügen:

- Ableitung: Generierung eines neuen Modells, das sowohl die semantischen als auch die graphischen Informationen enthält
- Komposition: Verwendung zweier Modelle: eines für die semantischen und eines für die graphischen Informationen

Wie bereits erwähnt definieren graphische Frameworks zwei Haupttypen graphischer Komponenten: Knoten und Kanten. Demgegenüber enthält das Model von UML 2.0 über 200 Klassen. Die graphischen Informationen über Ableitung dem semantischen Modell hinzuzufügen wäre daher äußerst aufwendig und höchstens mit Mitteln der automatischen Codegenerierung möglich. Aus diesem Grund wird hier der zweite Ansatz, die Komposition von Objekten, verfolgt.

Grundsätzlich müssen hierfür zwei graphische Modellklassen – je eine für Knoten und Kanten – erzeugt werden, die jeweils eine Referenz auf das Objekt der UML 2.0, also das semantische Modell, enthalten. Das Problem der Speicherung der graphischen Informationen tritt bei allen Editoren für UML-Diagramme auf. UML Diagram Interchange, kurz UML-DI, ist eine Erweiterung der UML 2.0 und definiert ein Modell zur Speicherung von Diagrammen ([UML2.0DI]). Es ist sehr allgemein gehalten und ließe sich daher auch für andere semantische Modelle verwenden.

Wie die besprochenen graphischen Frameworks interpretiert auch UML-DI ein Diagramm als eine Art Graphen mit Knoten und Kanten. Das folgende Klassendiagramm zeigt vereinfacht die Struktur von UML-DI:

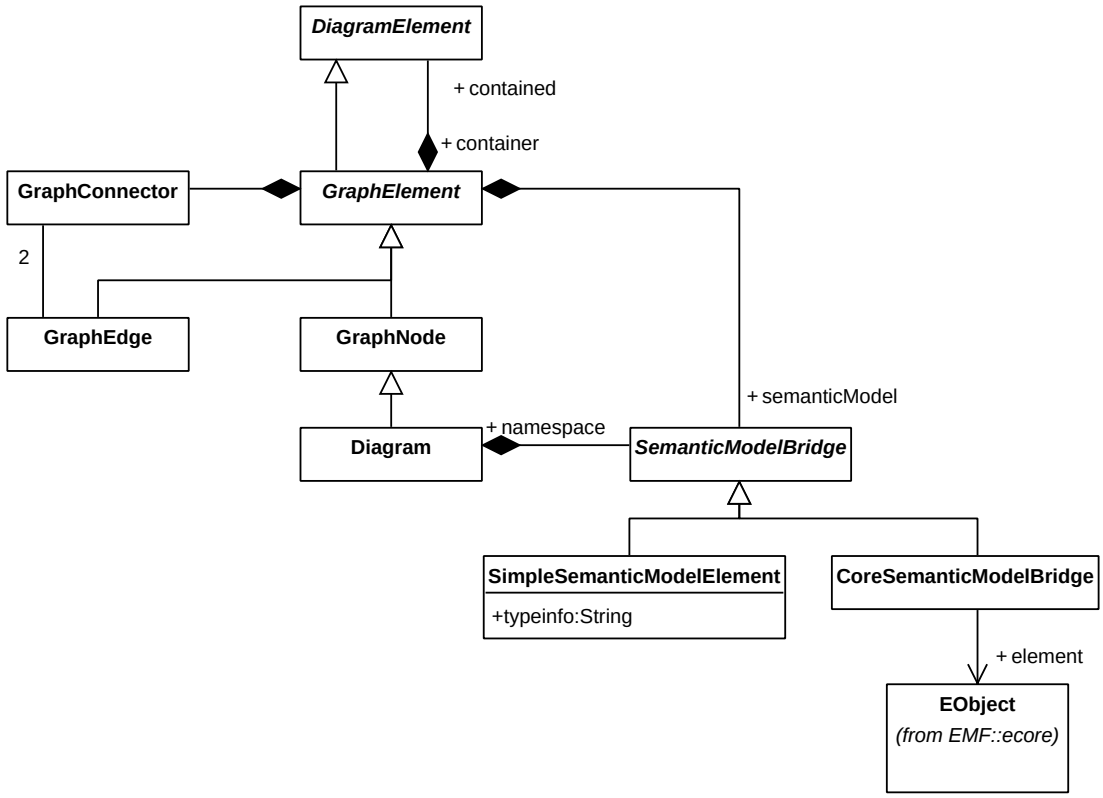

Abbildung 37 Klassendiagramm UML-DI, vereinfachte Übersicht

Knoten (`GraphNodes`) und Kanten (`GraphEdges`) sind von einer allgemeinen Klasse `GraphElement` abgeleitet, an die mittels des Brückenmusters (vgl. [Gamma+96, 186ff]) das semantische Modell, hier UML2-Elemente, angeschlossen werden kann. Auch hier werden Kanten über Anker (`GraphConnector`) mit den Knoten verbunden. UML-DI stellt also damit die auch in den Frameworks zu findenden Elemente der Zeichnung bereit.

Dieses UML-DI-Modell ist als EMF-Modell nachgebildet. Mittels des Codegenerators von EMF wurde eine entsprechende Implementierung des Modells erstellt. Wie UML2 hat auch diese Implementierung die typischen EMF-Eigenschaften. Insbesondere ist die *Serialisierung* von und nach XMI möglich. Da UML2 ebenfalls diese Eigenschaft besitzt, kann so das Modell im standardisierten XMI-Format gespeichert werden.[20]

[20] UML-DI wurde primär als Format zum Austausch der Diagramminformationen zwischen Tools verschiedener Hersteller entwickelt. Entsprechend müssen die Daten serialisiert werden, um als Austauschformat verwendet werden zu können. Dabei wird

Die Spezifikation von UML-DI beschreibt nicht nur das Modell, sondern auch die Zuordnung von UML-DI-Elementen und UML-Elementen [UML2.0DI, S. 25, 26]. Dort wird definiert, welche UML-Elemente als Knoten und welche als Kanten abgebildet werden. Das Mapping definiert allerdings keine 1 zu 1 Beziehung – es gibt UML 2.0 Elemente, die aus mehreren graphischen Elementen bestehen.[21]

Ein Beispiel hierfür sind die Verweise von Kommentaren zu den Elementen, die sie kommentieren. Diese Beziehung ist, im Gegensatz zu anderen Relationen, im UML-Modell nicht als eigenständige Klasse definiert. Mittels der speziellen Brückenklasse `SimpleSemanticModelElement` kann dieses Problem jedoch gelöst werden: Der `EditPart` verweist wie üblich auf ein Modell-Objekt vom Typ `GraphEdge`, dieses enthält in diesem Fall dann ein `SimpleSemanticModelElement`, das

---

auf den Standard XMI zurückgegriffen, d.h. UML-DI Instanzen werden als XML-Dateien in XMI-MOF gespeichert. Diese Dateien können dann nicht nur die graphischen Informationen enthalten, sondern zugleich auch das UML-Modell, welches dazu ebenfalls mit XMI serialisiert werden muss bzw. kann. Da sowohl UML2 als auch die eigene UML-DI Implementierung auf EMF beruhen, wird dieses Format automatisch unterstützt. Damit sind die Diagramme und, wenn separat gespeichert, auch die Modelle von GUMLE potentiell mit anderen Tools austauschbar, etwa mit Poseidon (URL: www.gentleware.com) oder den Entwicklungstools von IBM (etwa Rational Rose XDE Modeler, URL: http://www.ibm.com/software/info/ecatalog/de_DE/products/W107428N46756Z97.html). Da die neuen Versionen der IBM-Tools ebenfalls EMF und UML2 verwenden, dürfte hier der Austausch relativ unproblematisch sein.

[21] Die Spezifikation von UML-DI ist noch nicht abgeschlossen und die angegebene Tabelle ist weder vollständig (es fehlt etwa der Typ „CommentLink“) noch in allen Belangen plausibel. Zur Klärung von nicht eindeutigen oder fehlenden Angaben wurden mit Poseidon UML erstellte Beispieldokumente analysiert. Poseidon UML von Gentleware ist aktuell das einzige Produkt, das einen Export von Diagrammen mittels XMI und UML-DI unterstützt, Gentleware ist auch bei der Definition des Standards UML-DI beteiligt, wie etwa aus den Copyright-Vermerken in [UML2.0DI, iv] hervorgeht.

entsprechend als „CommentLink“ gekennzeichnet wird. Die Struktur ist somit konsistent, jedes graphische Modell-Element referenziert ein semantisches Modell-Element.

4.2.5 MVC mit zwei Modellen

Nachdem nun sowohl semantisches als auch graphisches Modell vorliegt, müssen diese beiden Modelle miteinander verbunden werden. Dies geschieht, wie bereits beschrieben, über Komposition. Implementiert ist diese Verbindung über die Assoziation `semanticModel` der UML-DI-Klasse `GraphElement`. Die folgende Graphik verdeutlicht, wie diese Komposition in das MVC-Muster eingearbeitet ist.

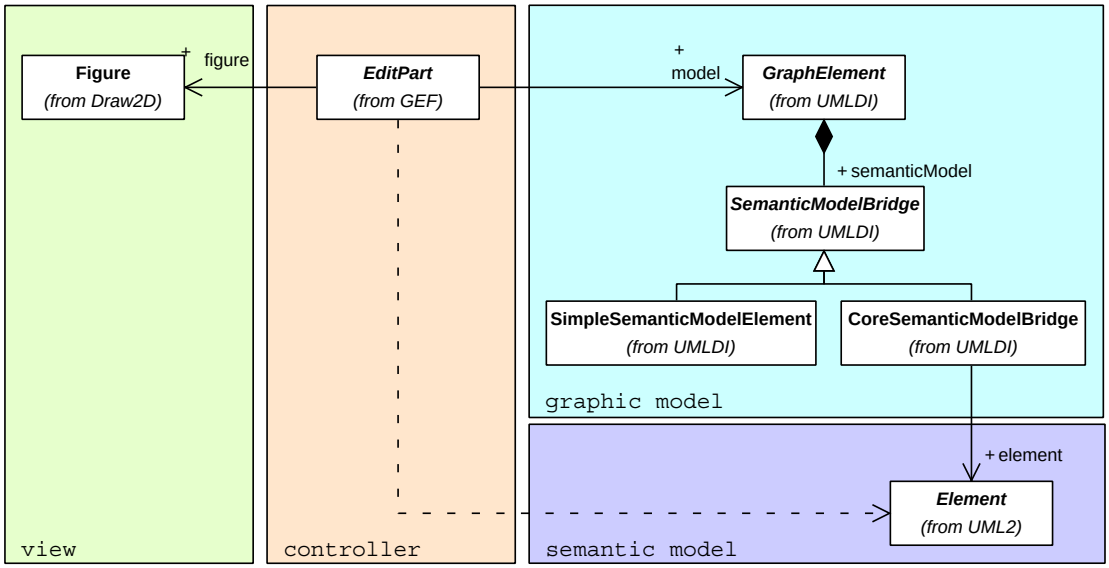

Abbildung 38: Klassendiagramm GUMLE, MVC mit UML2 und UML-DI

Aus Sicht des `EditParts` ist also das graphische Modell das primäre Modell. Das semantische Modell wird, in diesem Diagramm über die Abhängigkeitsbeziehung dargestellt, ebenfalls direkt vom `EditPart` angesprochen. Dies ist nötig, da einige Attribute in der Anzeige direkt aus dem semantischen Modell stammen, wie beispielsweise Namen von Klassen. Der `EditPart` wird bei beiden Modellen als Beobachter angemeldet.

Wie bereits in Kapitel 3.2.1 (S. 48) im Rahmen der Interpretation von Eclipse GEF als PAC-Agent gezeigt (Abbildung 29, S. 54), können semantisches und graphisches Modell verschiedenen Ebenen (Agenten-Level) zugeordnet werden. Im Sinne des

MVC-Musters kann aus Sicht des semantischen Modells das graphische Modell als View-Teilnehmer gelten. Grund dieser Verteilung auf verschiedene Ebenen oder Teilnehmer der beiden Modelle ist der klassische Fall, mehrere Sichten auf dieselben Daten zu benötigen. So können etwa mehrere Diagramme identische UML-Elemente referenzieren.

4.2.6 Hierarchie der EditParts und Aufbau der MVC-Tripel

Der Editor ist semantisch orientiert, das heißt aus Sicht des Benutzers werden nicht „Knoten“ mit „Kanten“, sondern „Klassen“ mit „Assozationen“ verbunden. Daher wird die Struktur des UML2-Modells für die primäre Strukturierung der GEF-*Controller* und *Views* (d.h. `EditParts` und `Figures`) übernommen. Das folgende Diagramm zeigt ausschnittsweise, wie die Hierarchie der GEF-*Controller* (`EditParts`) die Hierarchie der UML2-Klassen widerspiegelt:

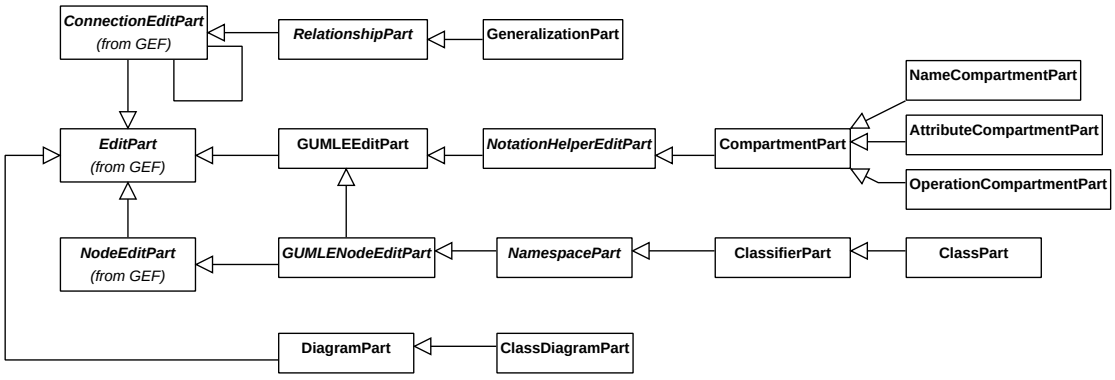

Abbildung 39: Klassendiagramm GUMLE, Hierarchie der EditParts

Jedes graphische Element, das der Benutzer im Editor später bearbeiten können, das heißt etwa mittels Drag-and-Drop verschieben oder über Mausklicks erzeugen soll, muss in GEF einen eigenen `EditPart, also` *Controller,* besitzen. `EditPart`-Objekte können in GEF hierarchisch angeordnet werden, wobei ein `EditPart` jeweils eine geordnete Liste von Kindern führen kann. Da die `EditParts` jeweils Teil eines *MVC-Tripels* sind, muss die Objekt-Hierarchie der *Views* (`Figures`) und des *Modells* (UML-DI) dieselbe Hierarchie besitzen.

Diese Beschränkung auf eine einzige sortierte Liste hat starke Auswirkungen auf die Implementierung der *Views*: Da nur die Ordnung der Liste der Kinder die Objekt-

Hierarchie der *Views* bestimmt, muss die Anordnung der Kinder innerhalb der *Parent View* geschickt gewählt werden. Einerseits können spezielle Layouts verwendet werden, um die Kinder entsprechend anzuordnen. Andererseits ist hier die Zuordnung der UML-DI-Elemente zu den UML-Elementen von Nutzen. Das folgende Beispiel zeigt zwei *Klassen*, die mit einer *Generalisierung* verbunden sind. Dabei ist im Fall der ersten *Klasse* auch die innere Struktur eingezeichnet (bei der zweiten *Klasse* wurde diese zur Vereinfachung unterdrückt).

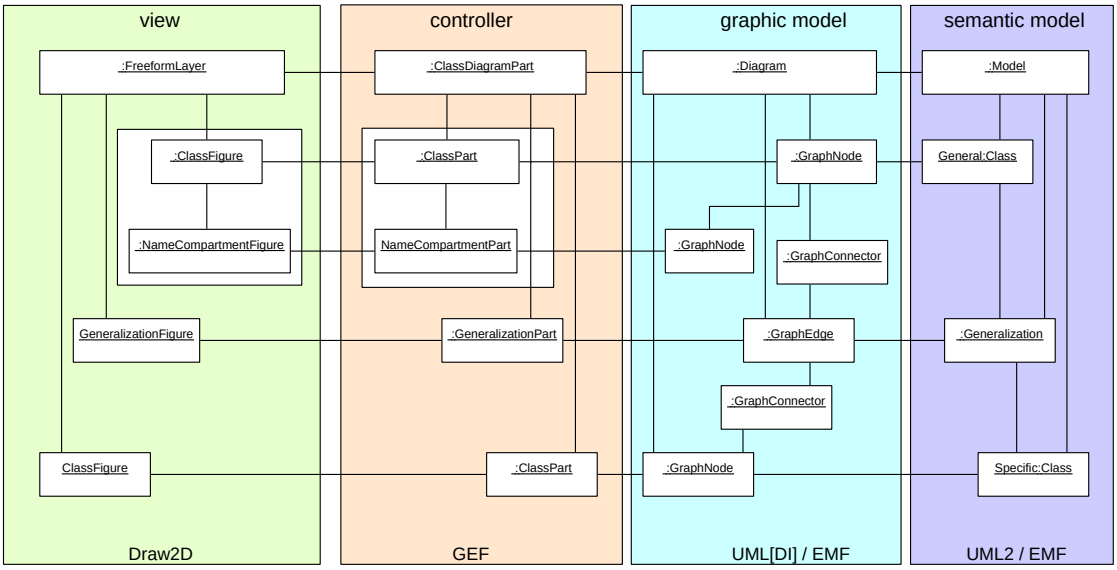

Abbildung 40: Objektdiagramm GUMLE, Hierarchie der Komponenten

Die vier Balken zeigen die Teilnehmer des MVC-Musters (mit graphischem und semantischem Modell). Untereinander ist die Hierarchie der *Komponenten* dargestellt: Die *Komponenten* selbst können wiederum *Komponenten* enthalten. Die Diagramm-Komponente enthält hier drei Komponenten: zwei *Klassen* und eine *Generalisierung*. Beispielhaft ist hier ein Kind der oberen Klasse, das *NameCompartement*, eingezeichnet.[22] Über den geschickten Einsatz graphischer Elemente ohne korrespondierende UML-Elemente, also `GraphElements` mit einem

[22] Zur Übersichtlichkeit ist hier nur diese eine Kinderkomponente aufgeführt, tatsächlich enthalten beide Klasse je drei Kinderkomponenten in Form des Name-, Attribute- und Operation-Compartments.

`SimpleSemanticModelElement` als *Bridge*, können hier Strukturen unabhängig vom semantischen Modell aufgebaut werden.

4.3 Implementierung

Die folgenden Abschnitte beleuchten einige Implementierungsdetails des Prototyps. Die ausgewählten Details sind vor allem für Nachfolgeimplementierungen auf Basis des Prototyps von Interesse. So werden einige Probleme und ihre Lösungen oder Lösungsansätze vorgestellt, die im Rahmen der Realisierung des Prototyps aufgetreten sind.

4.3.1 Implementierung des Beobachtermusters

Das *MVC-Muster* ist eine Erweiterung des *Beobachtermusters*: Der *Controller* ist ein *Observer* des *Modells*. Da UML2 und UML-DI Implementierungen auf EMF-Basis sind, wird die EMF-Implementierung des Beobachtermusters eingesetzt. Das observierte *Subjekt* ist hier vom Typ `Notifier` (Unterklassen Element oder DiagramElement). Der *Beobachter*, in Java häufig `Listener`, heißt im Fall von EMF `Adapter`, da er zusätzlich die Eigenschaften eines *Adapters* des *Adaptermusters* [Gamma+96] oder einer *Extension* im *Extension Object-Pattern* [Gamma98] besitzt.

Leider können die GEF `EditParts` nicht direkt das EMF-Interface `Adapter` implementieren, da im speziellen Fall des `ConnectionEditParts` ein Namenskonflikt auftritt: Beide Interfaces definieren eine Methode `getTarget(),` allerdings mit unterschiedlicher Semantik und unterschiedlichen Rückgabetypen. Aus diesem Grund wird ein Delegationsobjekt (`EditPartAdapter`) verwendet, wie im folgenden Diagramm dargestellt ist:

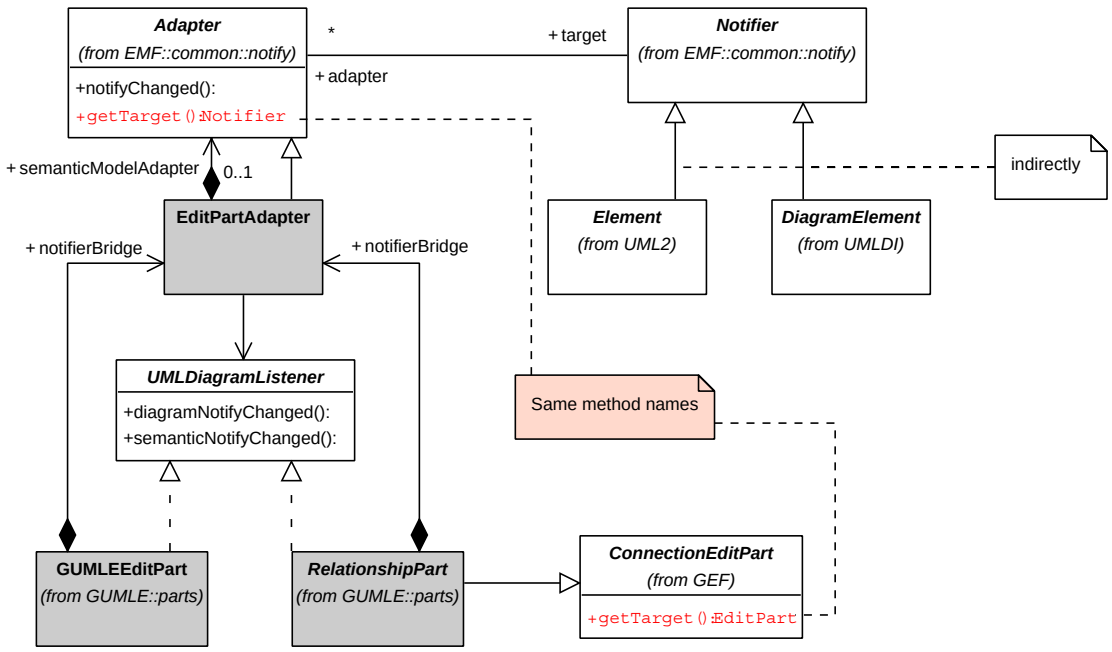

Abbildung 41: Klassendiagramm GUMLE, Adapter-Delegator

Dieses Delegationsobjekt wird gleichzeitig genutzt, um sowohl graphisches als auch semantisches Modell zu beobachten. Um einfach zwischen Nachrichten der beiden Modelle unterscheiden zu können, beobachtet das Delegationsobjekt zunächst nur ein Modell und verwendet intern ein weiteres Delegationsobjekt, um das andere Modell zu beobachten. Die Nachrichten werden dann an Instanzen von `UmlDiagramListener` geordnet weitergeleitet. Dieses Interface wird von allen verwendeten `EditPart`-Ableitungen implementiert.

Über diesen Mechanismus ist es auch geschachtelten `EditParts` mit graphischen Modellen ohne UML2-Modell (also Modellen vom Typ `SimpleSemanticElement`) möglich, direkt das semantische Modell eines Vorgängers zu beobachten. Aus Sicht der Programmierung dieser geschachtelten `EditParts` entspricht der Zugriff auf das semantische Modell dem von `EditParts`, deren Modelle das semantische UML2-Objekt selbst enthalten.

4.3.2 Implementierung der Befehle

Die Verwendung von graphischem und semantischem Modell birgt neue interessante Aspekte, die im Rahmen dieser Arbeit nicht behandelt werden können. Ein Aspekt wirkt sich auf die Gestaltung der Befehlsobjekte bzw. deren Funktionalität aus. Durch die strikte Trennung von Modell und Darstellung (sowie den geschickte Einsatz des

Fabrikmusters) muss bei Einsatz von Eclipse GEF im Befehl nur das Modell manipuliert werden – *Controller* -und *View*-Änderungen werden dann davon abgeleitet.

Durch die Aufteilung des Modells stellt sich nun die Frage, welches Modell – das semantische, das graphische oder beide – innerhalb der Befehlsausführung manipuliert werden soll. Da zudem EMF-basierte Modellimplementierungen eine eigene Umsetzung des Befehlsmusters mitbringen, sind mehrere Möglichkeiten denkbar, unter anderem:

- Manipulation nur des semantischen Modells, wie *View* und *Controller* können Änderungen im graphischen Modell davon automatisch (etwa im *Controller*) abgeleitet werden
- Manipulationen nur des graphischen Modells und Delegation der Änderungen am semantischen Modell an Befehle der EMF-Implementierung
- Manipulation beider Modelle

Im Prototyp wurde die dritte Variante gewählt. Dabei wurden teilweise die von der EMF-Implementierung der Modelle bereitgestellten Befehle verwendet. Die konkreten Implementierungen der Methoden zur Befehlsausführung (`Command.execute()`) sind relativ umfangreich und kompliziert. Das Hauptproblem dabei ist, dass Änderungen am Modell zu Benachrichtigungen der Controller führen – was an sich ja gewünscht ist. Allerdings ist dabei die Synchronisation der beiden Modell problematisch.

Am Beispiel der `execute()`-Methode der abstrakten Klasse `CreateGraphEdgeCommand`, von der alle Befehle zum Erzeugen von Kanten abgeleitet sind und entsprechend diese `execute()`-Methode verwenden (und dabei die aufgerufenen Templatemethoden `createConnectionModel()` und `createEdge()` überschreiben), wird die Reihenfolge, die eingehalten werden muss, deutlich:

```
public void execute() {
    try {
        sortEnds(); // 1
        createConnectionModel(); // 2
```

```
            GraphEdge edge = createEdge(); // 3
            setEdge(edge);
            createWaypoints(); // 4
            createConnectors(); // 5
            registerEdge(); // 6
        } catch (RuntimeException ex) {
            ...
        }
    }
```

Codebeispiel 14: CreateGraphEdgeCommand.java, execute()

Bei (1) werden die Enden der *Connection* sortiert – dies ist ein eher technisches Problem. In Schritt (2) wird das semantische Modell manipuliert, hier also eine Assoziation oder Generalisierung erzeugt. Die Anbindung des Modells an den *Controller* geschieht erst in Schritt (3) zusammen mit der Erzeugung des graphischen Modellobjekts. Das Erzeugen der Wegpunkte der Linie und der *Connectoren* ((4) und (5)) sind nun unproblematisch. Interessant ist, dass erst am Ende das graphische Modellobjekt, die `GraphEdge`, als Kind an ihren *Behälter* angemeldet werden darf (6). Würde dies vorher geschehen, so würde der bereits den *Behälter* beobachtende `EditPart` benachrichtigt und versuchen, seine „Kinder“ zu aktualisieren, was ohne (4) und (5) zu unerwünschten Ergebnissen führt (die Linie könnte nicht wie erwartet gezeichnet werden).

Generell ist die hier skizzierte Lösung brauchbar, die konkreten Kanten-erzeugenden Befehle sind in der Folge sehr einfach umzusetzen. Für eine Ausarbeitung dieses Problems sollten weitere Aspekte berücksichtigt werden: Wenn semantisches Modell und graphisches Modell unterschiedlichen Agenten zugeordnet werden (vgl. PAC-Muster), kann es vorkommen, dass andere Sichten nur das semantische Modell verändern. Hier ist zu klären, wie und in welcher Form diese Änderungen im Diagramm reflektiert werden können und sollen, und welchen Einfluss derartige Ereignisse auf die Umsetzung der Befehlsobjekte haben. Beispielsweise ist es durchaus denkbar, dass ein- und dasselbe Modellobjekt, etwa eine Instanz einer UML-Klasse, sowohl in einem Sequenz- als auch in einem Klassendiagramm dargestellt werden soll. Wenn nun durch die Definition einer Nachricht im Sequenzdiagramm die Klasse zugleich eine neue Abhängigkeitsbeziehung, quasi „implizit“, zugeordnet bekommt, stellt sich die Frage,

ob diese Abhängigkeit überhaupt im Klassendiagramm dargestellt werden soll und – falls die Änderung reflektiert werden soll – wie die entsprechende Kante im Klassendiagramm zu erzeugen ist.

Folgend wird ein Beispiel für eine solche andere Sicht vorgestellt und skizziert, welche Probleme hier konkret auftreten können.

4.3.3 Property-Sheets und Command-Stack

Die Modelle basieren auf Generaten des EMF. Neben der Generierung einer robusten Modell-API kann EMF auch einen baumbasierten Editor in Form eines Eclipse-Plugins generieren. Dieser Editor verwendet einen Baum zur Darstellung der Hierarchie der Elemente und so genannte *PropertySheets* zur Bearbeitung der Eigenschaften der einzelnen Elemente. Letztere sind in der Eclipse-Umgebung im Menü unter Window > Show View > Other und im folgenden Dialogfenster unter Base > Properties zu aktivieren. Auch wenn der baumbasierte Editor von GUMLE nicht verwendet wird, können diese *PropertySheets* zur Bearbeitung der UML-Elemente verwendet werden, wie folgende Abbildung zeigt:

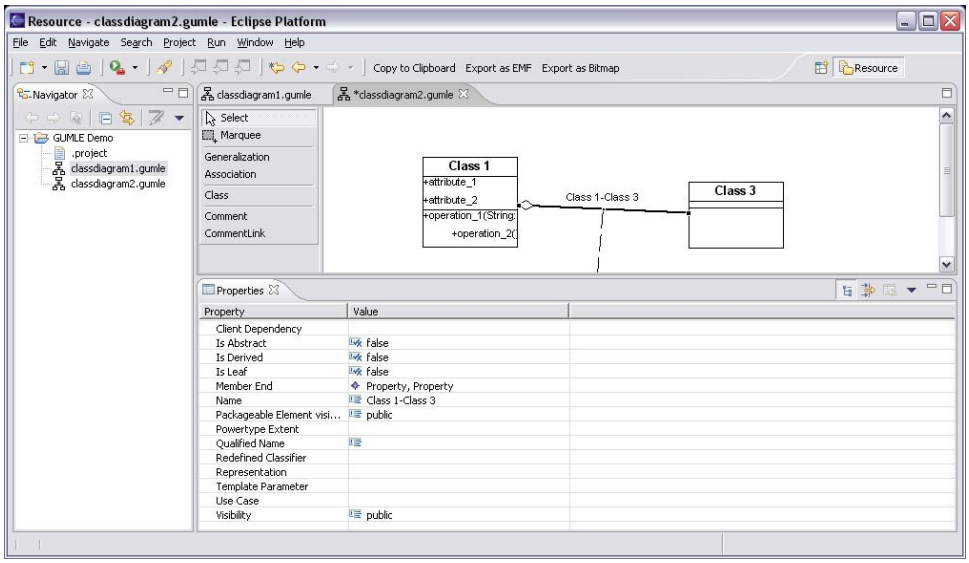

Abbildung 42: Screenshot GUMLE, Properties

Dieser „Nebeneffekt“ von EMF kann verwendet werden, um alle Eigenschaften der Elemente, sofern sie nicht direkt in der Zeichnung bearbeitet werden können, zu editieren.

Das Problem an dieser Stelle ist, dass EMF eine eigene Implementierung des Befehlsmusters mit eigenem Befehlsstapel für Undo- und Redo-Funktionalität generiert. Die Verbindung der Befehlsmusters von Eclipse GEF und EMF ist im Prototyp, wie oben beschrieben, nicht sauber umgesetzt (da es ja hier vorwiegend um die Demonstration des graphischen Editors geht). Daher ist die Verwendung des *PropertySheets* in GUMLE nur unter Vorbehalt möglich, das Verhalten von GUMLE ist hier nicht exakt definiert.[23]

4.3.4 OCL Miniparser

Graphische Editoren und Benutzerschnittstellen stoßen in bestimmten Bereichen an ihre Grenzen. Klassische Beispiele hierfür sind die visuelle Programmierung von Schleifen oder die Überlegenheit von TeX gegenüber graphischen Formeleditoren – auch wenn es hier vielfach um Fragen des „Geschmacks“ geht.

Im Falle des UML-Editors kann so eine Grenze im Bereich der Notation von Attributen und Operationen in OCL-Syntax gesehen werden. Die Implementierung einer graphischen Benutzerschnittstelle, die Sichtbarkeit, Rückgabewerte, Parameterliste und weitere Attribute einer Operation über Kontextmenüs editierbar macht, ist sowohl in der Programmierung als auch in der Benutzung sehr aufwendig. Die Definition einer Operation der Art

+createWindow (location: Coordinates, container: Container [0..1]): Window

---

[23] Das Problem der unterschiedlichen Befehlsstapel und auch Befehlsklassen ist von der Eclipse Community erkannt worden. Eine Lösung ist es, den Befehlsstapel weder auf Ebene von Eclipse GEF noch EMF zu implementieren, sondern auf Ebene der Platform, also der Applikation. Dieses Problem steht in engem Zusammenhang mit der Zuordnung von semantischem und graphischem Modell – und in diesem Fall auch von den verschiedenen Befehlsstapeln – zu unterschiedlichen Ebenen innerhalb der Applikation (vgl. Kapitel 4.2.5, S. 78). Die Erstellung eines applikationsweiten Befehlsstapels bzw. das Fehlen eines solchen in der aktuellen Version ist als Fehler im Bugtracking-System von Eclipse vermerkt (https://bugs.eclipse.org/bugs/show_bug.cgi?id=37716) und wird dort kontrovers diskutiert.

(aus [UML2.0S, 79]) ist schnell geschrieben, über Kontextmenüs und spezielle Dialoge wären jedoch etliche Interaktionen notwendig.

Um trotzdem eine semantische Prüfung der Eingabe zu ermöglichen und die entsprechenden UML-Elemente korrekt zu initialisieren, wurde hier ein Mini-OCL-Parser für Attribut- und Operationssignaturen mit Hilfe des Tools JavaCC [JavaCC] implementiert.

Bei JavaCC handelt es sich um einen Parser- und Scannergenerator für Java. Dieses Tool liest eine Sprachbeschreibung in EBNF (Erweiterte Backus-Naur-Form) ein und generiert daraus Java-Klassen. Die Sprachbeschreibung enthält neben den Produktionsregeln Java-Rümpfe, welche auf die in den Produktionsregeln definierten Tokens und Symbole zugreifen können.

Der folgende Auszug zeigt Teile der Definition zum Parsen einer Operationssignatur:

```
/**
* OperationSignature :=
*       [Visibility] Name [ "(" [ParameterList] ")" ] [ ":" Type ] [{[PropertyList]}]
*/
void OperationSignature(Signature s):
{}
{
       [ Visibility(s) ] Name(s) [ "(" [ParameterList(s)] ")" ] [ ":" Type(s) ]
        ["{" [PropertyList(s)] "}" ]
}
```

Codebeispiel 15: SignatureParser.jj, Auszug (Paket de.fernuni.gumle.signatureparser)

Im Java-Kommentar ist die Produktionsregel ohne JavaCC-Elemente, darunter, im Methodenrumpf, in JavaCC-Form angegeben. Die Klasse *`Signatur`* enthält später die Informationen der geparsten Eingabe.

4.3.5 Code-Konvention und Logging

Die Benennung von Variablen verwendet eine angepasste ungarische Notation:

```
Name    := [Scope ,_'] [,a'][Typ] Beschreibung

Scope   := ,m' | ,s' | ,i' | ,io' | ,o'      wobei  m-Member,
                                             s-Static                    Member,
                                             i,io,o: vgl UML „IN“, „INOUT“ und „OUT“

Typ     := ,str' | ,i' | ,b' | ,f' | ,d' | ,list' ...

                                             wobei  die  Abkürzungen  für  String,  int,
boolean,
                                             float, double oder List stehen

Beschreibung := gültige Zeichen für Variablennamen
```

Konstanten werden entsprechend der Java Konvention groß und ohne Präfixe geschrieben.

Variablen ohne Scope-Präfix sind damit, außer im Fall von Konstanten, lokale Variablen. Das Präfix „*m_*“ ersetzt so die ansonsten häufige Konvention, Member-Fields immer über „*this.*“ anzusprechen.

Zum Logging wird die „Jakarta Commons Logging API“ (JCL) mit darunter liegendem JDK-Logger verwendet. Ein speziell angepasster „Formatter“ (*de.fernuni.gumle.utils.EclipseConsoleFomatter*) erzeugt die Ausgabe derart, dass innerhalb der Eclipse-Umgebung die Stelle, an der die Log-Meldung erzeugt wurde, über die „Console“ direkt angesprungen werden kann.

4.4 Installation und Anwendung des Prototyps

GUMLE liegt als Eclipse-Plug fertig übersetzt in Form eines Archivs auf beiliegender CD vor. Zur Installation muss dieses Archiv entpackt und in das Installationsverzeichnis von Eclipse installiert werden. Dabei sind folgende Systemanforderungen zu berücksichtigen:

- Eclipse, Version 3.0.1, getestet wurde im Rahmen dieser Arbeit eine Version unter Windows XP
- Eclipse GEF, Version 3.0.1
- Eclipse EMF, Version 2.0.1
- Eclipse UML2, Version 1.0.1

- optional (nur Windows): Tritos Eclipse Plug-Ins[24]

Zur Vereinfachung liegt eine vollständig konfigurierte Version von Eclipse für Windows inklusive aller benötigter Plug-Ins bei.

In einem beliebigen vorhandenen oder einem neu zu erstellenden Projekt (File > New > Simple Project) kann nun über den GUMLE *Wizard* (File > New > Other > GUMLE > Classdiagram) ein neues Klassendiagramm erstellt werden.

Abbildung 43: Screenshot GUMLE, Wizard

Dieser erstellt ein leeres Klassendiagramm, der GUMLE Editor wird automatisch geöffnet.

[24] Dieses Plug-In wird unter Windows verwendet um das Diagramm im „Enhanced Meta Format“ zu speichern. Die auf der CD beiliegende Version behebt einen Fehler in diesem Plug-In und wurde direkt vom Autor Boris Bokowski zu Verfügung gestellt, die öffentlich verfügbare Version (vom 12.11.2004, URL: http://sourceforge.net/projects/tritos/) enthält diesen Fehler noch.

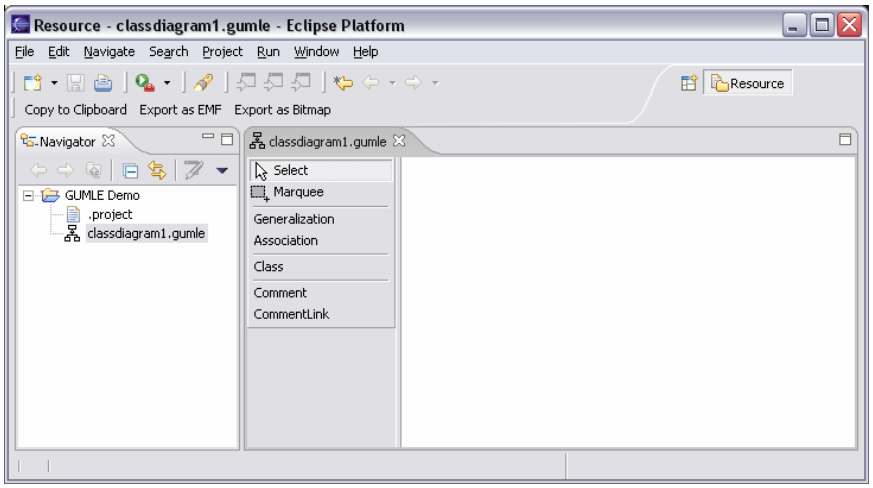

Abbildung 44: Screenshot GUMLE, leeres Klassendiagramm

Auf der linken Seite des Screenshots ist die Navigatoransicht zu sehen. Diagramme werden dort als Dateien mit der Endung „.gumle“ und entsprechendem Symbol angezeigt. Auf der rechten Seite ist der Editor zu sehen, dieser enthält eine Palette mit Tools. Unter der Toolbar sind drei Schalter eingefügt, über die das Diagramm als Bitmap (JPG-Format) gespeichert werden kann. In der Windows-Version kann das Diagramm zusätzlich im „Enhanced Meta Format“ als Vektorgrafik gespeichert oder ins Windows-Clipboard kopiert werden.

Die folgende Abbildung zeigt ein Beispieldiagramm mit den umgesetzten Funktionen.

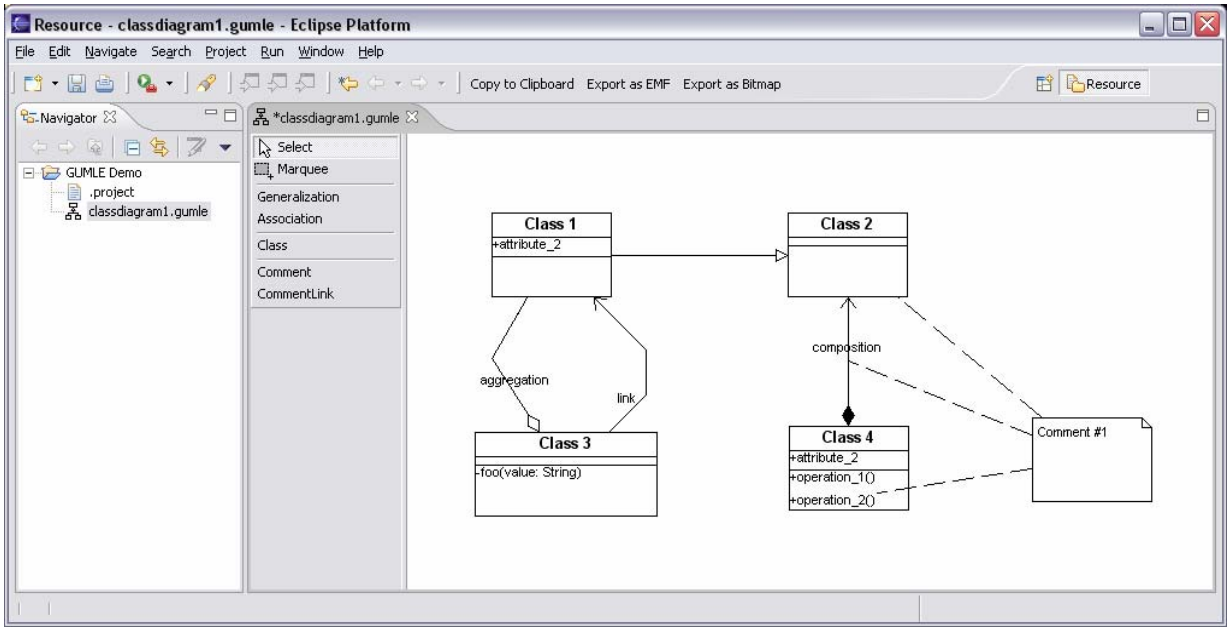

Abbildung 45: Screenshot GUMLE, Beispieldiagramm

Die angezeigten Komponenten werden erzeugt, indem in der Palette das entsprechende Tool gewählt und das gewählte Element auf der Zeichenebene eingefügt wird. Selektierte Komponenten können verschoben und in der Größe geändert werden, bei *Connections* können dabei auch neue *Bendpoints* eingefügt werden. Vorhandene Texte können durch Selektion und einen weiteren Mausklick direkt im Diagramm editiert werden, Operationen und Attribute müssen dabei in gültiger OCL-Notation eingegeben werden.

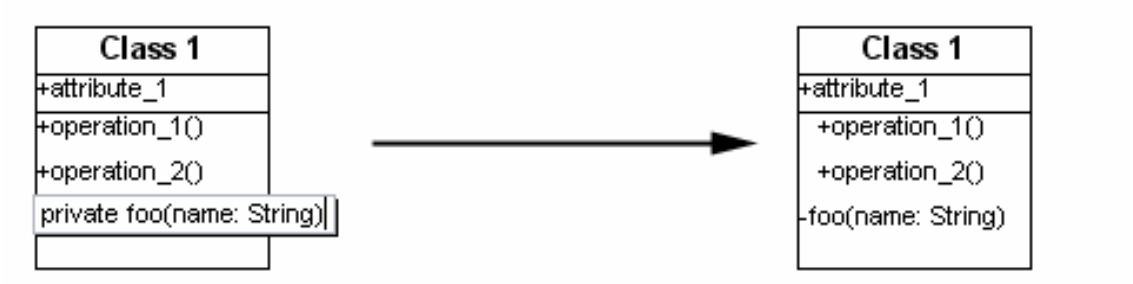

Abbildung 46: Screenshot GUMLE, OCL Parser erkennt Signatur der Operation

Operationen und Attribute sowie die Kardinalitäten und Arten der Assoziationsenden können über Kontextmenüs eingefügt und manipuliert werden.

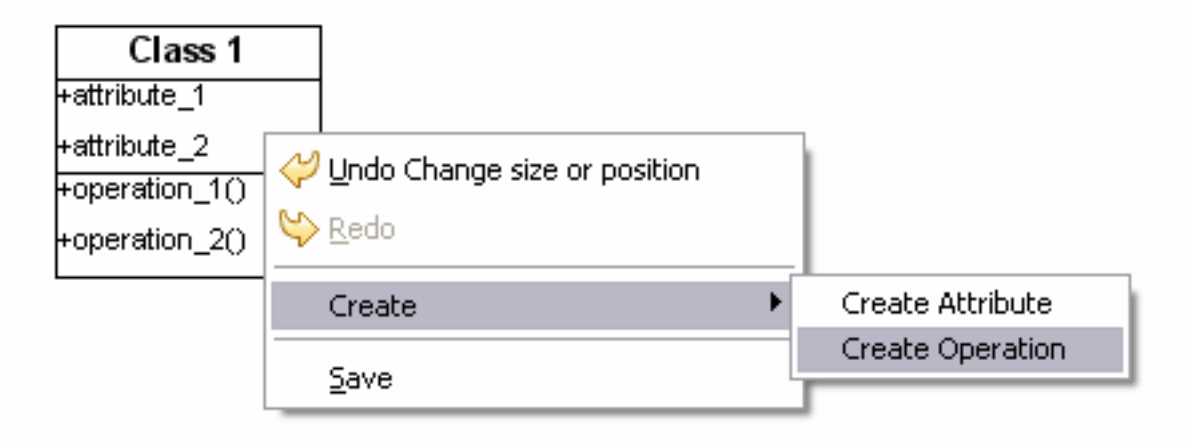

Abbildung 47: Screenshot GUMLE, Kontextmenü

Hier sind auch aktivierte Einträge für Undo (und Redo) zu sehen, die der Prototyp ebenfalls beherrscht. Komponenten können durch Selektion und die „Delete“-Taste gelöscht werden.

5 Schluss

Anhand der im Rahmen dieser Arbeit erfolgten Umsetzung des Prototyps zeigt sich, dass die Implementierung eines vollständigen UML-Editors unter Einsatz eines graphischen Frameworks mit akzeptablem Aufwand möglich ist. Interessanterweise rücken die spezifischen Probleme graphischer Editoren hierbei in den Hintergrund. Offensichtlich sind die beschriebenen Konzepte graphischer Editoren ausgereift und in den Frameworks praxistauglich umgesetzt.

Die Gesamtkomplexität eines graphischen Editors wird, wie allgemein bei Anwendungen mit graphischer Benutzeroberfläche, durch die Aufteilung in Darstellung und Modell handhabbarer. Da die Frameworks sowohl die Probleme der Darstellung als auch das Interaktionsmanagement gut lösen, ergeben sich Probleme bei der Anbindung und in dem Bereich des Modells. Die hier vorgenommene Trennung von semantischem und graphischem Modell bietet hier einen Lösungsansatz, der, etwa in Hinblick auf die Probleme der Befehlsimplementierung, weiter ausgebaut werden kann. Die Zuordnung der verschiedenen Modellaspekte auf unterschiedliche Ebenen der Applikationen (vgl. PAC-Muster) könnte dabei eine mögliche Lösung darstellen.

Unabhängig von graphischen Editoren stellten sich im Verlauf dieser Arbeit zunächst nicht vermutete Probleme bei der Beschreibung der Frameworks heraus. Die von den Frameworks mitgelieferte bzw. in der Literatur vorhandene Dokumentation ist sehr uneinheitlich. Eine standardisierte Dokumentation, ähnlich der von Entwurfsmustern, ist hier nicht zu finden – dies mag vielleicht auch daran liegen, dass alle Frameworks Open-Source-Projekte sind. Zumindest sind alle drei Frameworks über Entwurfsmuster dokumentiert, und auch hier wurden diese als Methode für die Beschreibung und den Vergleich herangezogen.

Ein spezielles Problem der Verwendung von Entwurfsmustern im Rahmen der Dokumentation ist, dass die Umsetzung von den bekannten Beschreibungen der Muster abweichen können. So unterscheiden sich die Umsetzungen im Fall des MVC-Musters so stark, dass es schwer fällt, überhaupt noch von ein- und demselben Muster zu

sprechen; gerade das MVC-Muster scheint als ein „Do the right thing pattern“[25] in allen interaktiven Systeme vorhanden.

Wenn, wie im Fall des Modells von UML, sehr viele Modellklassen vorliegen, stellt sich die Frage, wie der Aufwand zur Realisierung der Komponenten gering gehalten werden kann, da ja eine Modellklasse jeweils mindestens eine View- und eine Kontrollerklasse benötigt. Hier könnten für eine vollständige Umsetzung eines UML-Editors Verfahren aus der MDA[26] und allgemein Codegenerierung helfen. Interessant wäre es an dieser Stelle, zu untersuchen, inwiefern aus, evtl. annotierten, Modellklassen direkt graphische Editoren ableitbar sind. Das freie MDA-Tool openArchitectureWare[27] bietet etwa die Möglichkeit, automatisch Eclipse GEF-basierte graphische Editoren zu generieren.

---

[25] vgl. Joseph Bergin, URL: http://csis.pace.edu/~bergin/patterns/dotherightthing.html (Stand: 10.3.2005)

[26] MDA (Model Driven Architecture) ist ein von der OMG eingetragenes Warenzeichen. Um rechtliche Probleme zu umgehen werden auch häufig synonyme Begriffe wie MDSD (Model Driven Software Development) verwendet.

[27] URL: http://architecturware.sourceforge.net (Stand: 10.3.2005)

Anhang

Literatur

[Beck94] Kent Beck, Ralph Johnson (1994): Patterns Generate Architectures. In: Proceedings of the ECOOP'94. Berlin; Springer Verlag, Seiten 139-149.

[Berg04] Klaus Berg (2004): Das Presentation Abstraction Control Pattern in der Praxis. In: Javamagazin 09/2004, S. 18-26.

[Bokiwski+05] Boris Bokowski, Frank Gerhardt (2005): GEF: Maßgeschneiderte grafische Editoren selbst erstellen. In: Eclipse Magazin, 2/2005, S. 85-90.

[Brant95] John Brant (1992): HotDraw. Thesis University of Illinois. URL: ftp://st.cs.uiuc.edu/pub/papers/HotDraw/HotDraw.ps.gz (Stand: 10.3.2005)

[Budinsky+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose (2003): Eclipse Modeling Framework: A Developer's Guide. Boston u. a.; Addison-Wesley.

[Buschmann+98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal (1998): Pattern-orientierte Software-Architektur: Ein Pattern-System. Bonn u. a.; Addison-Wesley-Longman.

[Carrington+04] David Carrington, Leesa Murray, Paul Stropper (2004): An approach to specifying software frameworks. In: Proceedings of the 27th conference on Australasian computer science, Volume 26, Dunedin, New Zealand; ACM International Conference Proceeding Series, Seiten 185-192.

[Cavaness03] Chuck Cavaness (2003): Programming Jakarta Struts. Sabastopol; O'Reilly & Associates.

[Eclipse Wiki] Anonymous: Eclipse Wiki, URL: http://eclipse-wiki.info/ (Stand: 9.7.2004)

[Fayad+97] Mohamed E. Fayad, Douglas C. Schmidt (1997): Object-Oriented Application Frameworks. In: Communications of the ACM, Volume 40, Issue 10 (October 1997), Seiten 32 - 38.

[Fowler03] Martin Fowler (2003): Patterns of Enterprise Application Architecture. Boston; Person Education.

[Frankel03] David S. Frankel (2003): Model Driven Architecture. Applying MDS to Enterprise Computing. Indianapolis; Wiley Publishing.

[Beck+03] Kent Beck, Erich Gamma (2003): Contributing to Eclipse: principles, patterns, and plug-ins. Boston u.a.; Addison-Wesley.

[Gamma+96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1996): Entwurfsmuter: Elemente wieder verwendbarer objektorientierter Software. Bonn; Addison-Wesley-Longman.

[Gamma98] Erich Gamma (1998): Extension Object. In: Robert C. Martin, Dirk Riehle, Frank Buschmann (Hrsg.) (1998): Pattern languages of program design 3. Reading u. a.; Addison-Wesley. Seiten 79-88.

[Gokhale+04] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt (2004): Frameworks: Why They Are Important and How to Apply Them Effectively. URL: http://citeseer.ist.psu.edu/649481.html (Stand: 10.3.2005)

[Helm+92] Richard Helm, Tien Huynh, Kim Marriott, John Vlissides (1992): An Object-Oriented Architecture for Constraint-Based Graphical Editing. In: Proceedings of the Third Eurographics Workshop in Object-Oriented Graphics, Champéry, Seite 1-22,.

[Hudson03] Randy Hudson (2003): Create an Eclipse-based applicatoin using the Graphical Editing Framework. IBM developerWorks, URL: http://www-106.ibm.com/developerworks/opensource/library/os-gef/ (Stand: 9.7.2004)

[Johnson92] Ralph E. Johnson (1992): Documenting Frameworks using Patterns. In: Andreas Paepcke (Hrsg.) (1992): Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices, Vol. 27, No. 10, New York; ACM Press, S. 63-72.

[Johnson97] Ralph E. Johnson (1997): Components, Frameworks, Patterns. In: 5C9 Symposium on Software Reusability archive. Proceedings of the 1997 symposium on Software reusability table of contents. Boston, Seiten 10-17

[Kaiser01] Wolfram Kaiser (2001): Become a programming Picasso with JHotDraw. JavaWorld, Feb. 2001. URL: http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw_p.html (Stand: 20.2.2005)

[Krasner+88] Glenn Krasner, Stephen Pope (1988): A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: Journal of Object-Oriented Programming, Aug./Sept. 1988, Seiten 26-49,. URL: http://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf (Stand: 7.2.2005)

[Lieberherr+89] Karl J. Lieberherr, Ian Holland (1998): Assuring Good Style for Object-Oriented Programs. In: IEEE Software, Vol. 6, 5/1989, Seiten 38-48. URL: http://citeseer.ist.psu.edu/lieberherr89assuring.html

[Martin00] Robert C. Martin (2000): Design Principles and Design Patterns. URL: http://www.objectmentor.com (Stand: 8.3.2005)

[MOF2.0C] Object Management Group (2003): Meta Object Facilitiy (MOF) 2.0 Core Specification. OMG Adopted Specification, Version 2.0, 4.10.2003. URL: http://www.omg.org/docs/ptc/03-10-04.pdf (Stand: 16.7.2004).

[MOF-XMI] Object Management Group (2003): MOF-XMI Final Adopted Specification. OMG Adopted Specification, Version 4.11.2004. URL: http://www.omg.org/docs/ptc/03-11-04.pdf (Stand: 16.7.2004).

[Moore+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden (2004): Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM Redbook, URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf (Stand: 7.7.2004)

[OCL2.0] Object Management Group (2003): OCL 2.0. OMG Final Adpoted Specification, Version 14.10.2003. URL: http://www.omg.org/docs/ptc/03-10-14.pdf (Stand: 16.7.2004)

[Pilgrim05] Jens von Pilgrim (2005): Agile MDA mit EMF: dargestellt am Beispiel einer PHP-Website. In: Eclipse Magazin, 2/2005, S. 78-84.

[Riehle00] Dirk Riehle (2000) Framework Design. A Role Modeling Approach. Dissertation ETH Zürich. URL: http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf (Stand: 20.2.2005)

[Robbins99] Jason E. Robbins (1999): Cognitive Support Features for Software Development Tools. Dissertation University of California. URL: http://argouml.tigris.org/docs/ (Stand 28.7.2004)

[Six+02] Hans-Werner Six, Mario Winter (2002): Software-Engineering I. Grundkonzepte der objektorientierten Softwareentwicklung. Kurs 1793, FernUniversität in Hagen.

[Sutherland63] Ivan E. Sutherland (1963): Sketchpad: A man-machine graphical communication system. PhD thesis, Massachusetts Institute of Technology. Hier: Technical Report No. 574, University of Cambridge, 2003. URL: http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-574.pdf (Stand: 8.12.2004)

[Sweet85] Richard E. Sweet (1985): The MESA Programming Environment. In: Proceedings of the ACM SIGPLAN Symposium of Programming Languages and Programming Environments. Vol. 20(7), Seiten 216-229. URL: http://www.digibarn.com/friends/curbow/star/XDEPaper.pdf (Stand: 7.2.2005)

[UML2.0DI] Object Management Group (2003): UML 2.0 Diagram Interchange. OMG Adopted Specification, 1.9.2003. URL: http://www.omg.org/docs/ptc/03-09-01.pdf (Stand: 16.7.2004)

[UML2.0I] Object Management Group (2003): UML 2.0 Infrastructure Specification. OMG Adpoted Specification, 15.9.2003. URL: http://www.omg.org/docs/ptc/03-09-15.pdf (Stand: 16.7.2004)

[UML2.0S] Object Management Group (2003): UML 2.0 Superstructure Specification. OMG Adopted Specification, 2.8.2003. URL: http://www.omg.org/docs/ptc/03-08-02.pdf (Stand: 16.7.2004)

[Vlissides89] John Vlissides, Mark Linton (1989): Unidraw – A Framework for Building Domain-Specific Graphical Editors. In: Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology.Williamsburg, Seiten 58 - 167.

[Vlissides90] John Vlissides (1990): Generalized Graphical Object Editing. PhD thesis, Stanford University, URL: http://citeseer.ist.psu.edu/vlissides90generalized.html (Stand: 02.03.2005)

[XMI2.0] Object Management Group (2003): XML Metadata Interchange (XMI) Specification. OMG Adopted Specification, Version 2.0, 2.5.2003. URL: http://www.omg.org/docs/formal/03-05-02.pdf (Stand: 16.7.2004)

Projektseiten im Internet der besprochenen Frameworks:

[JHotDraw] Java HotDraw, Version 5.1, URL: http://www.jhotdraw.org (Stand 7.3..2005)

[TigrisGEF] Java Graph Editing Framework, URL: http://gef.tigris.org (Stand 7.3.2005)

[EclipseGEF] Graphical Editing Framework, URL: http://www.eclipse.org/gef (Stand 7.3.2005)

[EMF] Eclipse Modeling Framework, URL: http://www.eclipse.org/emf (Stand 7.3.2005)

[UML2] EMF-based UML 2.0 Metamodel Implementation, URL: http://www.eclipse.org/uml2 (Stand 7.3.2005)

[RCP] Eclipse Rich Client Platform, URL: http://www.eclipse.org/rcp (Stand 7.3.2005)

[SWT] SWT Standard Widget Toolkit, URL: http://www.eclipse.org/swt (Stand 7.3.2005)

[JavaCC] Java Compiler Compiler, URL: https://javacc.dev.java.net/ (Stand 7.3.2005)

[JCL] Jakarta Commons Logging, URL: http://jakarta.apache.org/commons/logging/ (Stand 7.3.2005)

Abbildungsverzeichnis

Inhalt der CD-ROM

Pfad	Beschreibung
Frameworks	
Graphical Frameworks	
Eclipse-GEF-SDK-3.0.1.zip	Eclipse GEF 3.0.1
JHotDraw5.2.zip	JHotDraw 5.2
Tigris GEF-0.10.11-bin.zip	Tigris GEF 0.10.11
Tigris GEF-0.10.11-src.zip	
Modeling Frameworks	
emf-sdo-xsd-SDK-2.0.1.zip	EMF 2.0.1
uml2-SDK-1.0.1.zip	UML2 1.0.1
Other Tools	
commons-logging-1.0.4.zip	Logging API
javacc-3.2.zip	Java Parser und Scanner Generator
net.sf.tritos.emfactionsforgef.win32_1.1.0.zip	Eclipse Plug-In für Windows zum Erzeugen von Vektorgrafiken
net.sf.tritos.emfdevice.win32_1.1.0.zip	
GUMLE	
Eclipse-GUMLE Win 32	Vorkonfigurierte Eclipse-Umgebung für Windows
gumle.zip	Der Prototyp als Eclipse Plug-In inklusive des Quelltextes
Samples	
Eclipse PlugIn	Beispiel Eclipse-Plug-In
EclipseGEFSample.zip	Beispiel aus Kapitel 3.1.3
Executable Jars	Ausführbare Java-Archive, können direkt von CD gestartet werden
JHotDrawSample.jar	Beispiel aus Kapitel 3.1.1
TigrisGEFSample.jar	Beispiel aus Kapitel 3.1.2
lib	Bibliotheken für JAR-Archive
src	
de.jevopi.eclipsegef	Beispiel aus Kapitel 3.1.3, Quellen
de.jevopi.graphmodel	Modellklassen zu den Beispielen aus Kapitel 3.1.2 und 3.1.3
de.jevopi.hotdraw	Beispiel aus Kapitel 3.1.1, Quellen
de.jevopi.tigrisgef	Beispiel aus Kapitel 3.1.2, Quellen

Die vorkonfigurierte Eclipse-Umgebung kann unter Windows direkt von CD (eclipse.exe) gestartet werden. Der Prototyp sowie das Beispiel aus Kapitel 3.1.3 sind dort bereits als Plug-Ins installiert. Klassendiagramme können, wie im Text beschrieben, über den Wizard erstellt werden. Der Beispieleditor aus Kapitel 3.1.3 kann dort aktiviert werden, indem einfache Dateien (Simple Files) mit der Endung „.graph“ angelegt werden.

Begriffstabelle

	Unidraw	**HotDraw, JHotDraw**	**Tigris GEF**	**Eclipse GEF**	**UMLDI**
Zeichnung	Root Component	Drawing	Diagram	Root	Diagram
Komponente	Component	Figure	Fig	Figure, EditPart	GraphElement
Knoten		Figure	Node (FigNode)	Node	GraphNode
Kante		Connection	Edge (FigEdge)	Connection	GraphEdge
Dekoration		Decoration	Arrow Head	Decoration	
Wegpunkte		Points	Points	Bendpoint	Waypoint
Werkzeug	Tool	Tool	Mode	Tool	
Handle	Handle	Handel	Handle	Handle	
Router				Router	
Befehl	Command	Command	Action (Cmd)	Command, Action	
Anker	Connector	Connector	Port	Connection-Point	Anchor, Connector
Palette	Catalog	Palette	Palette	Palette	
Ebene			Layer	Layer	
Auswahl	Selection	Selection	Selection	Selection	

Index