SSD Editor auf Basis von GEF

Ferdinand Strixner und Asma Ayed

6. April 2003

Inhaltsverzeichnis

1 Einleitung **4**
1.1 Motivation . . . 4
1.2 Autofocus . . . 5
1.3 Beschreibungtechnik . . . 5
1.4 Quest . . . 7
1.5 Autofocus2 . . . 10

2 Anforderungen **11**

3 GEF **12**
3.1 Features . . . 13
3.1.1 Elementare Funktionen . . . 13
3.1.2 Zeichenfunktionen . . . 13
3.1.3 Connected-Graph-Funktionen . . . 15
3.1.4 View-Funktionen . . . 15
3.1.5 Funktionen zum Editieren . . . 16
3.1.6 Weitere Funktionen . . . 17
3.2 Architektur . . . 18
3.2.1 Präsentation der Klassen . . . 18

4 Entwurf **20**
4.1 Schichtenarchitektur . . . 20
4.2 Schnittstelle . . . 21
4.3 Konsistenz der Realisierung . . . 22
4.3.1 Schnittstelle zwischen Modell und View . . . 22
4.3.2 Schnittstelle zwischen View und Editor . . . 23
4.3.3 Steuerung durch Modell- und View-Ebene . . . 24
4.4 Änderungen gegenüber der Diplomarbeit . . . 25
4.5 Weitere Änderungen . . . 29

5 Implementierung **32**
5.1 Java-Klassen . . . 32
5.2 Implementierungsdetails im exemplarischen Durchlauf . . . 33
5.2.1 Initialisierung / Laden eines Models (QuestBrowser) . . . 33
5.2.2 Anlegen eines SSD-Editors, native und externe Ports (EditorFrame, FigPort, PortNode, SelectionPort, MyNetEdge, MyModeSelect, MyModeModify, MyModeManager) . . . 34
5.2.3 Weitergabe von Positionierungsinformationen durch Events (SSDComponentRootView / SSDComponentView) . . . 35
5.2.4 Exception-Konzept (ModeCreateChannel, SSDComponentRootView, etc.) . . . 36
5.2.5 Autorouting (CmdAutoRoute, FigChannel) . . . 36
5.2.6 automatische Bezeichner (SSDComponentRootView, ModeCreateChannel, ModePlaceComponent, ModePlacePort) . . . 37
5.2.7 Subkomponenten-Ansicht (MyModeSelect, MyModeModify, CmdDescend, QuestBrowser) . . . 37
5.2.8 schnelles Anlegen von Ports und Kanälen (ModeDirectConnect, EditorPalette) . . . 38

6 Zusammenfassung und Ausblick **39**
6.1 Zusammenfassung . . . 39
6.2 Ausblick . . . 40

7 Anhang **41**
7.1 Änderungen an GEF . . . 41

1 Einleitung

1.1 Motivation

Die Komplexität von Software steigert sich ständig und erfordert dadurch den Einsatz von CASE-Werkzeugen im Entwicklungsprozess. Es wird zwischen Softwareentwicklungswerkzeuge, die den gesamten Prozess unterstützen und solche, die nur Teile davon umfassen, unterschieden. GanzheitlicheWerkzeuge, welche die Phasen Analyse, Design, Implementierung und Test unterstützen, sind für die Anwender von Vorteil, da sie im Entwicklungsprozess nur die einmalige Einarbeitung für den Anwender erfordern. AUTOFOCUS und Quest[1] sind die Basis für diese Arbeit.

> AUTOFOCUS ist ein dokumentbasierter CASE-Werkzeug-Prototyp zur Modellierung verteilter, nebenläufiger Systeme. Es basiert auf einer sichtenorientierten Beschreibungstechnik. Dabei werden Ausschnitte aus dem Modell durch Editoren visualisiert und modifiziert. Quest integriert eine Reihe von Validierungs- und Verifikationswerkzeugen für das modellierte System.
>
> Bei dem in AUTOFOCUS integrierten Systemmodell handelt es sich um ein implizites Systemmodell, welches zur Laufzeit des Programms fragmentiert ist und sich an mehreren Stellen befindet. Dadurch ist dieses Werkzeug in Bezug auf die Verwendung im gesamten Entwicklungsprozess zu stark eingeschränkt. Die Tendenz bei der Entwicklung von Software erfordert aber ein explizites Systemmodell, welches in seiner Einheit vorliegt. Die Prozessunterstützung der Entwicklung wird durch den Einsatz eines zentralen Systemmodells erst ermöglicht. Dadurch wird die Basis für die Modellierung, die Erzeugung von Programmtext und die Verifikation bzw. Validierung des modellierten Systems gebildet. Durch ein explizites Systemmodell in AUTOFOCUS 2 sowie die Unterstützung der Sichtenorientierung durch geeignete Editoren sollte die Basis für ein Werkzeug geschaffen werden, das den gesamten Entwicklungsprozess effizient unterstützt[2].

Dazu wurde in der Diplomarbeit von Andreas Schweiger ein modellbasiertes Editorenkonzept entwickelt[3].

[1] siehe Diplomarbeit von Andreas Schweiger!
[2] Aus der Diplomarbeit von Andreas Schweiger!
[3] Aus der Diplomarbeit von Andreas Schweiger!

Dieses Konzept wird in dieser Arbeit verbessert und erweitert. Einige neue Funktionen werden für den Editor entwickelt(z.B. das Öffnen einer Subkomponente) und andere vorhandenen Funktionen werden verbessert(z.B. Externe Ports, Autorouting).

1.2 Autofocus

> AUTOFOCUS ist ein CASE-Werkzeug-Prototyp zur Modellierung verteilter und nebenläufiger Systeme. Es stellt mit seinen konzeptuell und semantisch integrierten Beschreibungstechniken ein Werkzeug für das Design eingebetteter Systeme dar. Neben der Spezifikation erlaubt das Softwaresystem ansatzweise auch die Validierung durch die Simulation eines konsistenten und ausführbaren Systemmodells. AUTOFOCUS basiert auf einer sichtenorientierten Beschreibungstechnik, d.h. Ausschnitte aus dem Modell werden durch Editoren visualisiert und modifiziert. AUTOFOCUS-Editoren arbeiten auf Dokumenten, nicht auf den Sichten eines Metamodells, welches die Struktur des Modells beschreibt. AUTOFOCUS enthält ein dokumentbasiertes Repository, in dem die folgenden Dokumente enthalten sein können:
>
> - Systemstrukturdiagramme (System Structure Diagram, SSD)
> - Zustandsübergangsdiagramme (State Transition Diagram, STD)
> - Interaktionsdiagramme (Extended Event Traces, EET)
> - Datentypdefinitionen (Data Type Definition, DTD)
>
> Das Werkzeug ist jedoch durch den dokumentbasierten Ansatz im Hinblick auf die Ausdehnung auf den gesamten Entwicklungsprozess und die umfassende Prozessunterstützung zu stark eingeschränkt. Deshalb wird dieser Ansatz in der nachfolgenden Version von AUTOFOCUS (AUTOFOCUS2) durch einen modellorientierten ersetzt. Das modellorientierte Repository enthält ein zentrales Systemmodell, das eine hierarchische Struktur aufweist und durch ein Metamodell beschrieben wird. Sichten als Ausschnitte auf das Modell werden durch Metaviews spezifiziert. Der Zugriff auf das Modell erfolgt bei der Verwendung eines Editors über die Sichten[4].

Das Modell ist in AUTOFOCUS auf mehrere Editoren verstreut, aber nicht zentral zugreifbar. Das sollte in de nachfolgenden Version (AUTOFOCUS2) nicht mehr der Fall sein.

1.3 Beschreibungtechnik

Die Struktur des Systemmodells wird durch ein Metamodell beschrieben(siehe1.1).

[4] aus der Diplomarbeit von Andreas Schweiger!

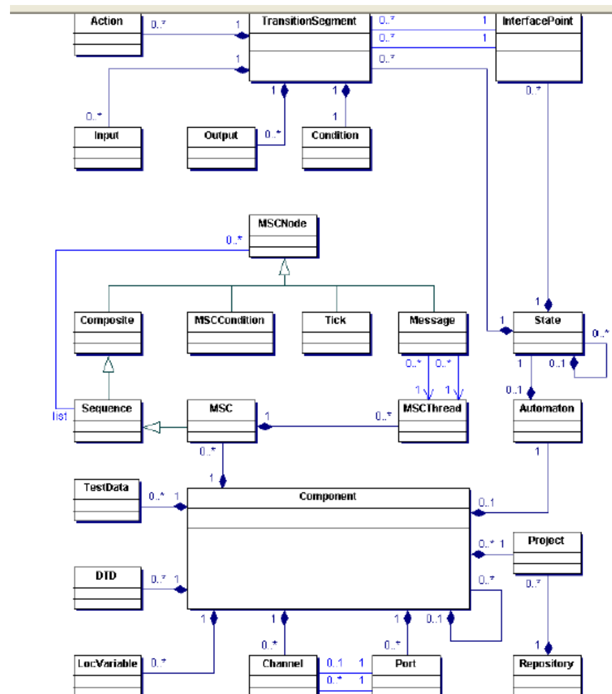

Abbildung 1.1: AUTOFOCUS-Metamodell

Die Spezifikationen der Komponenten sind die zentralen Elemente im Repository. Ihre Schnittstellen werden durch Systemstrukturdiagramme beschrieben. Eine Verfeinerung der Spezifikation wird durch zusätzliche Beschreibungen erreicht. Dies kann durch Datentypdefinitionen, Sequenz-, Zustandsübergangs- oder weitere Systemstrukturdiagramme für die Darstellung der Innensicht einer Komponente erfolgen. Mit der Verfeinerung durch Zustandsübergangsdiagramme und Systemstrukturdiagramme, welche die innere Sicht auf einen Zustand bzw. eine Komponente darstelllen, ergibt sich die hierarchische Struktur der Modellbeschreibung.

Im zentralen Repository können mehrere Projekte angelegt werden. Jedes Project besitzt genau eine Komponente als zentrales Element. Diese Komponente stellt das gesamte entworfene System dar. Eine Komponente kann durch eine beliebige Anzahl an Unterkomponenten verfeinert werden. Jede Verfeinerung stellt eine Innensicht der übergeordneten Komponente dar. Diese hierarchische Struktur ist durch die Kompositionsschlinge an der Komponente dargestellt. Die mit einer Komponente assoziierten Klassen vom Typ

> Port sind die Interaktionspunkte zwischen System und Umwelt bzw. zwischen einer Komponente und ihren Unterkomponenten. Die Verbindungen zwischen den Ports werden durch Kanäle vom Typ Channel dargestellt. Ein Kanal verbindet zwei Ports zur Kommunikation von Daten. Das Verhalten von Komponenten wird durch einen erweiterten endlichen Automaten spezifiziert. Nur atomare, d.h. nicht mehr verfeinerte Komponenten können in ihrem Verhalten durch einen Automaten spezifiziert werden. Ausnahme bildet jedoch die Beschreibung eines abstrakten Verhaltens einer verfeinerten Komponente[5].

Kurzgefasst sind die verschiedenen Spezifikationen eines Systems nicht voneinander isoliert, sondern haben in der Komponente ihr zentrales Element. Die Modelle für die Beschreibung der Struktur und des Verhaltens eines zu modellierenden Systems sind damit in ein einziges Metamodell ingetriert. Eine Komponente assoziiert u.a. einen Automaten, der ihr dynamisches Verhalten beschreibt. Umgekehrt sind die Ein- und Ausgabemuster einer Transition über MIF-Assoziationen mit den Ports der Komponente verbunden, zu welcher der Automat gehört. So wird eine wechselseitige Referenzierung zwischen der statischen Struktur einer Komponente und ihres Verhaltens realisiert und damit ein integrales Metamodell etabliert.

AUTOFOCUS hat drei Diagrammarten: SSD, EET und STD. Systemstrukturdiagramme (SSD) (siehe Abb. 1.2) zeigen den Aufbau, die Schnittstelle und den Datenaustausch eines Systems mit seinen Komponenten und Kanälen. Ports haben dabei eine Verbindungsrolle zwischen einem Kanal und einer Komponente. Der Informationsfluss in den Kanälen ist gerichtet. Die graphische Darstellung von Komponenten sind Rechtecke, von Ports Kreise und von Kanälen rechtwinklige Kanten.

1.4 Quest

> Im Quest-Projekt[6] wird AUTOFOCUS erweitert, um den Entwicklungsprozess mit grafischen Entwicklungswerkzeugen mit formalen Methoden zu verknüpfen, welche die Korrektheit des modellierten Systems gewährleisten. Die Quest-Werkzeuge übersetzen die graphischen Konzepte in andere Formalismen, die für die Verifikation, das Model-Checking und das Testen geeignet sind, und unterstützen die Erzeugung von Testfällen. Um dies zu realisieren, sind in Quest Übersetzungen für die angeschlossenen Werkzeuge implementiert. Realisiert ist dies durch die Verwendung einer API und einer textuellen Schnittstelle zu AUTOFOCUS.

[5] Andreas Schweiger: Diplomarbeit

[6] siehe Quest, 2002. 2

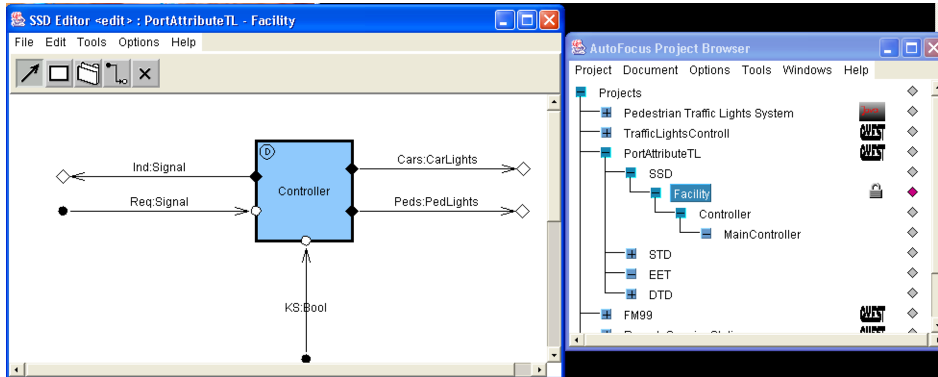

Abbildung 1.2: SSDEditor bei AUTOFOCUS1

Leider hat Quest ausser dem Modellbrowser keine Editoren also ist es schwer die Werkzeuge (z.B. Model-checker, Testfallgeneratoren, Codegeneratoren) zu modellieren.

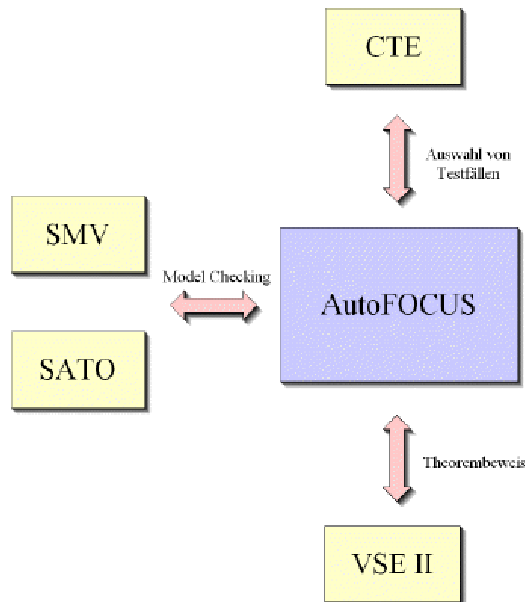

Abbildung 1.3: Integration der Werkzeuge zur Verifikation in AUTOFOCUS. Die Daten werden zwischen den beteiligten Softwaresystemen über das QML-Format ausgetauscht.

Um die Qualität von Software zu erhöhen, können mit Quest folgende

Werkzeuge in den Entwicklungsprozess integriert werden:

- Model-Checker überprüfen die temporalen Eigenschaften eines endlichen Modells automatisch. In Quest werden die Werkzeuge SMV[7] und SATO (UnboundedModel-Checker)[8] eingesetzt.
- Theorembeweiser unterstützen die Verifikation beliebiger Modelle und Eigenschaften. Verwendet wird das Werkzeug VSE II[9].
- Testwerkzeuge testen die entwickelte Software. Für die Auswahl von Testfällen wird CTE[10] eingesetzt. Ein weiteres Werkzeug generiert Testfälle aus den AUTOFOCUS-Spezifikationen. Die in AUTOFOCUS generierten Programmtexte werden gegenüber den Testfällen auf die geforderte Korrektheit überprüft(Validierung).

Im Zentrum von Quest steht AUTOFOCUS. Letzteres erlaubt dem Anwender die Spezifikation des zu modellierenden Systems durch eine grafische Notation. Die erwähnten Programmsysteme zur Verifikation und zum Testen des modellierten Systems werden über ein textuelles Austauschformat an AUTOFOCUS angebunden. Dazu ex- und importiert AUTOFOCUS das modellierte System über das QML-Format (Quest Model Language). Die Schnittstelle zwischen AUTOFOCUS und den einzelnen Werkzeugen wird durch JAIG[11] realisiert. JAIG ist ausführlich in folgendem Buch[12] beschrieben. Der Überblick über die integrierten Werkzeuge und der Zusammenhang mit AUTOFOCUS sind in Abb. 1.3 dargestellt. Durch die Anbindung von AUTOFOCUS an Verifikationswerkzeuge werden formale Methoden in den Entwicklungsprozess[13] integriert. Weiter Dokumente[14] beschreiben diese In-

[7]siehe G. Rock, W. Stephan, and A. Wolpers. Tool support for the compositional development of distributed systems. In Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme, number 315 in GMD Studien, 1997.2 und K.L. McMillan. The smv system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992. 2

[8]siehe H. Zhang. Sato: An efficient propositional prover. In William McCune, editor, Proceedings of the 14th International Conference on Automated Deduction, volume 1249 of LNAI, pages 272-275, Berlin, july 1997. Springer. 2

[9]siehe Ullmann, Baur, Reif, Siekmann, Scheer, and Moik. Specification Language VSE SL Version 2, 1999. 2

[10]siehe M. Grochtmann, J. Wegner, and K. Grimm. Test Case Design Using Classification Trees and the Classification-Tree Editor. In Proceedings of 8th International Quality Week, San Francisco, pages Paper 4-A-4, May 30-June 2 1995. 2

[11]siehe Slotosch Peter Braun, Heiko Lötzbeyer. Quest Developers Guide, 2002. 1.3, 1.4, 2, 2.4

[12]siehe Frank Marschall. Konzeption und Realisierung einer generischen Schnittstelle für metamodellbasierte Werkzeuge. Master's thesis, Technische Universität München, 1998. 1.4, 2

[13]siehe . Broy and O. Slotosch. Enriching the Software Development Process by Formal Methods. In Current Trends in Applied Formal Methods 98, LNCS 1641, 1999. 2

[14]Peter Braun, Heiko Lötzbeyer, Berhard Schätz, and Oscar Slotosch. Consistent Integration of Formal Methods. In Tools for the Analysis of Correct Systems (TACAS), 2000. 2 und Peter Braun, Heiko Lötzbeyer, and Oscar Slotosch. Quest Users Guide, 2002. 2, 3.1

tegration ausführlicher[15].

1.5 Autofocus2

AUTOFOCUS1 ist ein CASE-Werkzeug-Prototyp zur Modellierung verteilter und nebenläufiger Systeme. Quest ist eine Erweiterung von AUTOFOCUS und wurde entwickelt um weitere Werkzeuge wie z.B. Codegeneratoren, Modelchecker oder Testfallgeneratoren anzubinden. Leider hat er außer dem Modellbrowser eigentlich keine Editoren. Im AUTOFOCUS dagegen ist das Modell auf mehrere Editoren verstreut, aber nicht zentral zugreifbar. Daher kommt die Idee AUTOFOCUS und Quest zu kombinieren und davon eine neues Projekt zu erzeugen: AUTOFOCUS2.

AUTOFOCUS2 existiert noch nicht aber die Entwicklungen laufen voran, um dieses CASE-Werkzeug so bald wie möglich zu schaffen. Es soll sowohl die Funktionalitäten von AUTOFOCUS1 als auch die von Quest kombinieren. Die Idee dafür ist: die Entwicklung der in AUTOFOCUS1 vorhandenen Editoren, die auf das Kernmodell von Quest zugreifen können. Es wird weiter daran gearbeitet, den Kern aus Quest herauszuschneiden und ihn mit den Editoren (SSD, EET und STD) zu integrieren. Am Ende sollte AUTOFOCUS2 die Modellierung verteilter und nebenläufiger Systeme sowie die Anbindung von Quest-Werkzeugen (Codegeneratoren, ModelChecker, Testfallgeneratoren) erlauben.

In der Diplomarbeit von Andreas Schweiger wurden die verschiedenen Möglichkeiten zur Entwicklung eines SSDEditors für AUTOFOCUS2 betrachtet und evaluiert. Dabei wurden mehrere Rahmenwerke dafür erforscht, um festzustellen, welcher sich am besten eignet. Ausserdem wurden verschiedene Konzepte für die Umsetzung der AUTOFOCUS1-Funktionalitäten in AUTOFOCUS2 vorgeschlagen und bewertet[16]. In dieser Arbeit wird der in der Diplomarbeit gewählte Rahmenwerk GEF näher beschrieben. Ausserdem wird das Konzept der Diplomarbeit an einigen Stellen verbessert und weiter entwickelt. Die Diplomarbeit und das SEP sind die erste Schritte in der Entwicklung von AUTOFOCUS2.

[15] aus der Diplomarbeit von Andreas Schweiger!
[16] für ausführliche Informationen bitte Diplomarbeit von Andreas Schweiger nachschlagen!

2 Anforderungen

3 GEF

Das GEF-Projekt hat als Ziel die Bildung einer sogenannten Bearbeitungsbibliothek, die die Konstruktion und die Bearbeitung von hochwertigen Graphen-Applikationen ermöglicht. GEF (The Graph Editing Framework) (siehe Abb.3.1) besteht also aus mehreren Java-Klassen, die die Entwicklung neuer Applikationen einfacher gestalten. Diese Applikationen haben zwei Eigenschaften:

1. Zeichnen von Diagrammen.
2. Zeichnen von Verbindungen zwischen den Diagrammen.

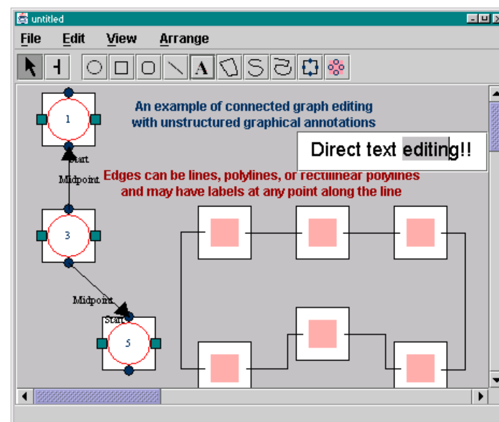

Abbildung 3.1: GEF-Screen

GEF hat ein einfaches Design, das sowohl die Benutzung als auch die Erweiterung dieses Rahmenwerks für jeden leichter macht. Ausserdem verfügt der Rahmenwerk über folgenden Figuren:

- Komponente
- Ports
- Kanäle

Anhand dieser Figuren können die meisten Graphen abgebildet werden. Das Modell-View-Controller Design von GEF basiert auf die sogenannte Swing Java UI Library und

ermöglicht also damit den Java-Programmierer die Beherrschung von GEF in kurzer Zeit. Außerdem ermöglicht GEF die Bewegung, das Selektieren, die Veränderung der Grösse, usw... der vorhandenen Figuren.

3.1 Features

3.1.1 Elementare Funktionen

Die Funktionalität von GEF ist vielfältig aber in diesem Abschnitt werden die häufig benutzen Funktionen genauer beschrieben.

- **Der Editorframe:** das Zeichen-Edidor kann mittels JGraphFrame oder JGraph panel in jeden Rahmen, Applet oder Dialog eingefügt werden.
- **Editor-in-Browser:** das Editor kann in einem Browser-Fenster geöffnet werden(siehe Abb. 3.2).
- **Plattform:** GEF ist mit allen Betriebssystemen und Entwicklung-Tools kompatibel aber der Sun-JDK wurde bis jetzt häufiger mit GEF benutzt.

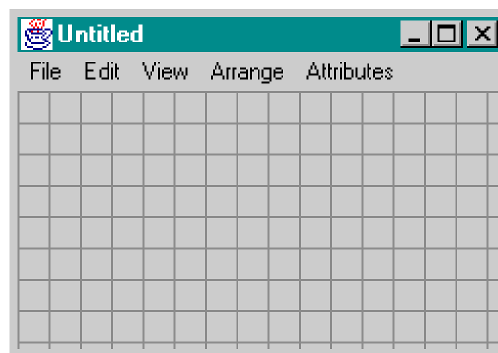

Abbildung 3.2: GEF-Editor

3.1.2 Zeichenfunktionen

- **Figuren:** ein GEF-Diagramm besteht aus mehreren Figuren(englisch figures oder figs), die mittels einem Layer geordnet werden. Diese Figuren können individuell bearbeitet werden(siehe Abb.3.3).
- **Elementare Formen:** GEF verfügt über elementare formen, die die üblichen mathematischen Formen entsprechen(z.B. Ein Rechteck, ein Kreis, eine Gerade, usw..)

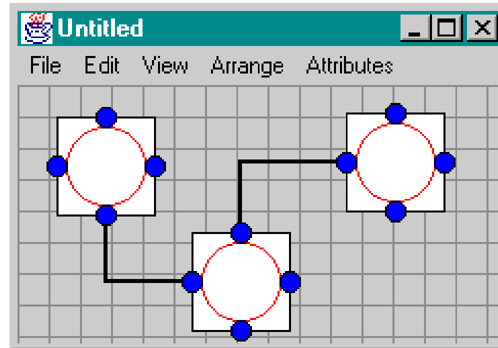

Abbildung 3.3: Beispieldiagramm

- **Gruppen:** die Klasse FigGroup implementiert diese elementare Formen
- **Graphische Eigenschaften:** Jede Figur hat ihre eigene graphische Eigenschaften(z.B. Farbe, Größe, Standort, usw..) Diese Eigenschaften könne beliebig verändert werden(siehe Abb.3.4). Neue Subklasse können ihre eigenen Eigenschaften mitbringen, die mittels get- bzw. set-Methoden definiert werden können(wie bei JavaBeans).

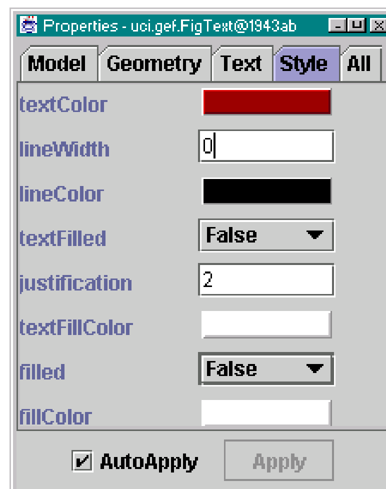

Abbildung 3.4: Eigenschaften einer Figur

- **Graph-Visualisierung:** Ein Graph kann mittels GEF-Diagramme repräsentiert werden. Die Klassen FigEdges und FigNodes implementieren die Kanten und die Knoten des Graphen. Die Klasse LayerPrespective verwaltet ein GraphModel(Interface), der den Graph entspricht. Diese LayerPrespective ermöglicht sogar die Darstellung

von einigen Teilen des Graphen.

- *Knoten-Visualisierung:* Auf dem Diagramm ist FigNodes die Visualisierung von NetNodes aus der NetList oder von den anderen Knoten in dem GraphModel. FigNodes ist eine Liste der elementaren Formen, die eine Kante haben könnte. Ein NetNode kann aus mehreren Fignode bestehen. Der GraphNodeRenderer kümmert sich um die Konstruktion von FigNodes.
- *Kanten-Visualisierung:* Auf dem Diagramm werden NetEdge(aus der NetList) oder die Kanten des GraphModel durch FigEdge visualisiert.
- *Ports-Visualisierung:* Einige formen aus FigNodes können dazu dienen, die NetPorts zu visualisieren.

3.1.3 Connected-Graph-Funktionen

- **Repräsentation von Graphen:** In GEF wird ein Connected-Graph durch Knoten, Ports und Kanälen repräsentiert. Diese Graph-Komponenten können mit eigenen Eigenschaften in Subklassen definiert werden. Der DefaulGraphModel ist eine Standard-Implementierung des GraphModel, die dem Benutzer erlaubt, seinen eigenen Graph zu definieren(so wie TreeModels und tableModels in Swing).
  - *NetList:* ist eine Liste von vorhandenen Knoten und Kanten. Die Ports sind aus den Kanten abgeleitet. Deswegen stehen sie impliziert auf der NetList.
  - *die Knoten(englisch Nodes):* NetNode ist ein Knoten im Graph und beinhaltet eine Menge von Ports.
  - *die Ports:* ein Netport ist ein Port, der mit einer Komponente assoziiert werden muss. Zu dem Port gehört eine Liste, die alle zugehörigen Kanälen dieses Ports beschreibt.
  - *die Kanäle:* ein NetEdge visualisiert eine Verbindung zwischen zwei Ports.

3.1.4 View-Funktionen

- **visuelles Update:** Beim Auftreten von Änderungen an den Eigenschaften der Komponenten können diese Änderungen entweder unmittelbar oder erst zu einem späteren Zeitpunkt erscheinen. GEF erlaubt die Programmierung von beiden Möglichkeiten.
- **Layers:** Ein Diagramm kann mehreren Layers beinhalten. Layers ermöglichen also z.B. die Visualisierung von unveränderbaren Hintergründen.
- **Mehrere Views:**Ein Diagramm kann auf mehreren Editors gleichzeitig visualisiert werden. Die Diagramm-Änderungen auf einem Editor(außer die Editor bezo-

gene Funktionen z.B. Komponente selektieren) werden auf die anderen Editoren reflektiert.

- **Minimales Update:** Bei einer Änderung an einem Teil des Diagramm wird nur dieser Teil wieder gezeichnet aber GEF bietet die Möglichkeit, diese Änderungen dynamisch anzupassen.
- **Eigenschaften einer Figur:** Die Eigenschaften der Figuren können in einem anderen Fenster gesehen oder verändert werden. Sie können auch weiter entwickelt oder erwitert werden.

3.1.5 Funktionen zum Editieren

- **Modi des Editieren:** Der GEF-Editor verfügt über zahlreiche Modi aber er kann nur ein Modi zu einem bestimmten Zeitpunkt bearbeiten. Jeder Modus entsprichst einen Event oder eine Instanz der Cmds. Mehrere Modi können gleichzeitig aktiv sein und werden auf einem Stack gelegt, der von ModeManager verwaltet wird.
  - *ModeSelect:* ist der Default-Modus und befindet sich immer im Stack.
    1. *select-by-click:* Bei einem Mausklick wird die Figur selektiert, die sich unter der Maus befindet(siehe Abb.3.5).

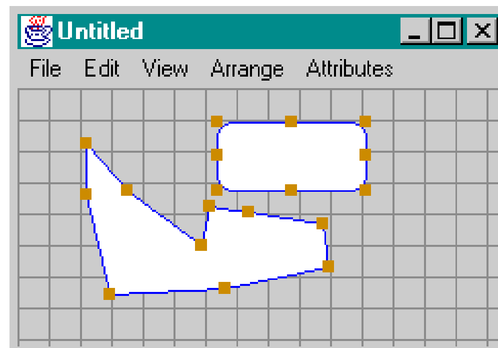

Abbildung 3.5: Selektierte Figuren

    2. *select-by-area:* Wenn die gepresste Maus bewegt wird, bildet sich ein Viereck. Alle Figuren, die sich in dem Viereck befinden, werden damit selektiert(siehe Abb.3.6).
    3. *select-all:* In dem Menü erlaubt ein Kommando das Selektieren aller Figuren eines Diagramms.
    4. *select-none:* Bei einem Mausklick im Hintergrund werden alle Figuren deselektiert.

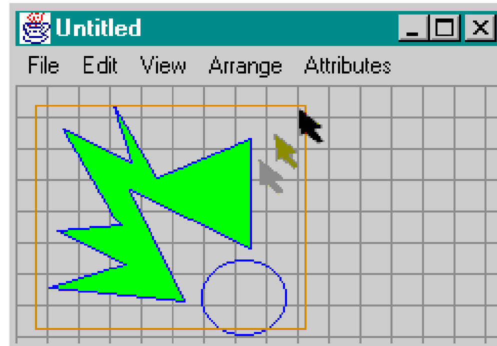

Abbildung 3.6: Selektieren

– *ModeModify:* Wenn der Benutzer die (über eine Figur) gepresste Maus bewegt, verändert sich die Figur ständig bis die Maus los gelassen wird(siehe Abb.3.7).

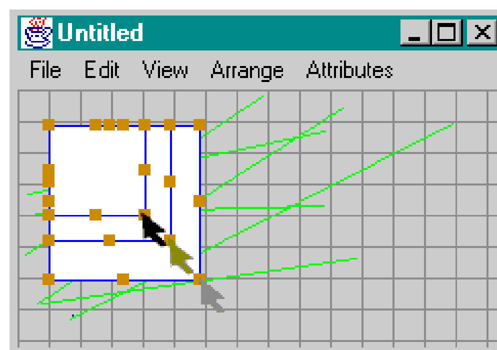

Abbildung 3.7: Veränderung einer Figur

1. *drag-object:* Eine Figur kann überall verschoben werden.
2. *drag-handle:* Die Größe einer Figur kann verändert werden.

- **Kommandos zum Editieren:** Der Editor führt die Kommandos (englisch Commands/Cmds) aus.

3.1.6 Weitere Funktionen

- **Öffnen und Speichern:** Diagramme können gespeichert oder geöffnet werden. Dies geschieht mit Hilfe der ObjectSerialization-Bibliothek des JDK(siehe Abb.3.8).

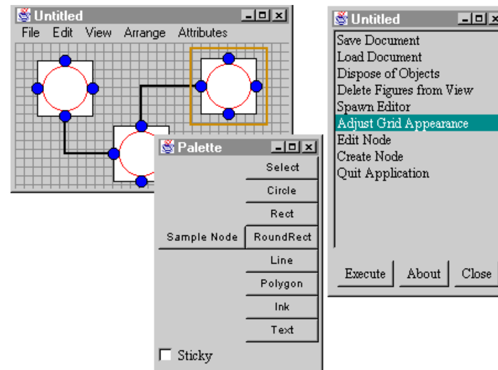

Abbildung 3.8: Save and Load

3.2 Architektur

Die Architektur von GEF ist kompliziert und besteht aus zwei Ebenen: Die Komponenten-Ebene und die Diagramm-Ebene.

- **Die Komponenten-Ebene:** besteht aus Knoten, die als logische Objekte betrachten werden. Diese Knoten können Applikationen bezogene Eigenschaften und Verhalten besitzen.
- **Die Diagramm-Ebene:** besteht aus einer Menge von Figuren(figs), die strukturierte bzw. nicht strukturierte Graphen abbilden können. Diese Figuren sind die Darstellung der Knoten der vorherigen Ebene.

GEF besteht aus mehreren Java-Klassen, die so strukturiert sind, dass eine Erweiterung möglich ist, ohne dass der vorhandene Code verändert wird. Der Graphen-Editor(in der Editor-Klasse) agiert wie ein Shell, der die Befehle an die anderen Klassen weiterleitet.

3.2.1 Präsentation der Klassen

- **Die Editor-Klasse:** ist die zentrale Klasse des Systems. Der Editor verhält sich aber wie ein Shell und führt keine Aktion aus sondern leitet sie weiter an das entsprechende Objekt der anderen Klassen.
- **Die Cmds-Klassen:** definieren doIt()Methoden, die einige Aktionen im Editor ausführen. Z.B. CmdReorder führt die Menü-Items: 'Send To Back', 'Bring To Front', usw..
- **Die Modes-Klassen:** sind für die Modi im Editor zuständig. Die elementaren Modi sind: Selektieren, Verändern und Einfügen neuer Objekte. Z.B. beim Verschieben einer Figur befindet sich das Modus in ModeModify aber beim Loslassen

springt es zu ModeSelect.

- **Die Guides-Klassen:** helfen dem Benutzer, schöne Diagramme zu zeichnen, in dem sie die Maus-Koordinaten verwalten.
- **Die Figs-Klassen:** sind Zeichen-Objekte, die im Editor verändert werden können. Sie betehen aus elementaren Elemente (z.B. eine Gerade oder ein Rechteck). Die FigGroup-Klasse ist die Menge der Figs-Klassen.
- **Die Layers-Klassen:** dienen zur Gruppierung der Figuren auf einem durchsichtigen Hintergrund sowie zur Verwaltung, zum wieder Zeichnen oder zum Finden einer Figur unter gegebenen Koordinaten der Maus.
- **Die Selections-Klassen:** sind Objekte, die der Editor aufruft, wenn eine Figur selektiert wird. Sie verwalten die Ereignisse, die zur Bewegung oder zum wieder Zeichnen eine Figur führen.

4 Entwurf

4.1 Schichtenarchitektur

GEF ist ein Programmgerüst, das aus mehreren Klassen besteht. Das Verwenden von GEF erfolgt durch Anpassung an die jeweilige Anwendungsdomäne mittels Vererbung. Die Struktur ist durch finale Methoden vorgegeben. Abstrakte Klassen müssen aber implementiert werden. Außerdem erlaubt die Architektur von GEF eine sogenannte Schichtenarchitektur(siehe Abb.4.1) durch den Modell-View-Controller.

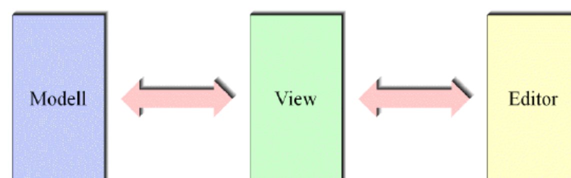

Abbildung 4.1: Die Schichtenarchitektur

Eine Schichtenarchitektur zeichnet sich durch Aufteilung nach Aufgabenbereichen in mehrere Ebenen aus. Die drei Schichten Modell, View und Editor können unabhängig voneinander implementiert werden. Durch die Schnittstellenspezifikation zwischen der View- und der Editorebene ziehen Änderungen am Modell oder auch am eingesetzten Editor-Rahmenwerk keine Änderungen am Informationsaustausch zwischen den Schichten und damit an der gesamten Anwendung nach sich. Eine Schicht verfeinert die Daten aus der unter ihr liegenden Ebene. In der Ebene des Modells sind die Informationen gespeichert, die durch das AUTOFOCUS-Metamodell bestimmt sind. In dieser Schicht sind die Modellelemente, ihre Attribute und Assoziationen des Metamodells gespeichert. Views ergänzen diese Daten durch Darstellungsinformationen, welche Attribute wie die Schriftart, Größe oder Position eines View-Elements beschreiben. Die Editorebene visualisiert die View-Elemente und erweitert die Anzeigeinformation mit Editorfunktionalität und realisiert die Benutzerinteraktion. Sie nimmt beispielsweise Ereignisse an der Benut-

zeroberfläche entgegen und delegiert diese zu ihrer Verarbeitung an die zuständigen Teile von GEF. Daneben erfolgt hier die Visualisierung der Sichten. Eine Ebene ruft nur die Dienste der unter ihr liegenden Schicht auf, aber nicht umgekehrt. Deshalb erlaubt diese Schichtenarchitektur die Verwendung eines beliebigen Editors(nämlich das von GEF). Änderungen am AUTOFOCUS-Metamodell, z.B. die Einführung eines neues Modellelements, erfordert keine Anpassung des Prinzips für die Kommunikation von Modelländerungen an die View-Ebene. Dennoch müssen die Zugriffsstrukturen in der View-Ebene an die neuen Modelleigenschaften angepasst werden. Diese Aufteilung der Architektur in Schichten wird durch folgende Überlegungen untermauert: Die View-Informationen sind nicht Teil des Modells, weil die Operationen auf dem Modell unabhängig von den Daten für die Darstellung sind. Außerdem ist das Modell ohnehin komplex, die Zusammenfassung beider Schichten würde die Komplexität erhöhen. Die View-Information ist nicht in der Editorebene enthalten, weil dies die Komplexität des Editors erhöht. Der Editor soll austauschbar sein, wenn eine neue Version des Editors zur Verfügung steht oder ein anderer Editor angekoppelt werden soll. Außerdem enthält die View-Ebene die persistenten Daten. Der Editor besitzt nur die Abhängigkeit von der View-Ebene und nicht auch noch vom Modell. Die View-Ebene delegiert die Modelloperationen an das Modell und verbirgt damit die Implementierungsdetails dieser Aktionen für den Benutzer der View-Schnittstelle [1].

4.2 Schnittstelle

Der Informationsfluss zwischen den drei Ebenen Modell, View und Editor ist in Abbildung(4.2) dargestellt. Editoroperationen, die Modell- oder View-Elemente verändern, werden an die View-Ebene weitergegeben und dort verarbeitet. Die Änderungen der View-Ebene werden an den Editor propagiert. Die View-Ebene ruft Dienste des Modells auf und wird von der Modellebene über die dortigen Änderungen benachrichtigt[2].

[1] aus der Diplomarbeit von Andreas Schweiger. Für ausführliche Informationen die Diplomarbeit nachschlagen!

[2] aus der Diplomarbeit von Andreas Schweiger! für mehr Informationen die Diplomarbeit nachschlagen.

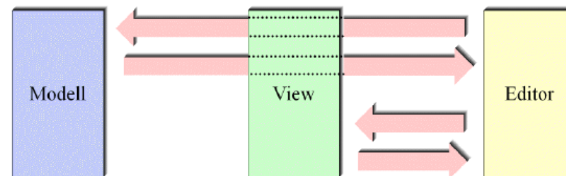

Abbildung 4.2: Informationsfluss zwischen den Ebenen

4.3 Konsistenz der Realisierung

4.3.1 Schnittstelle zwischen Modell und View

Editor und Modell kommunizieren nur über die View-Schicht. Dadurch wird es vermieden, dass wichtige Informationen (durch z.B.falsches Handeln am Editor) des Modell verloren gehen. Außerdem erleichtert das die Programmierung, da einige Eigenschaften des Modells sich in die View-Ebene spiegeln. Die Programmierung fokussiert sich also auf die Schnittstelle Zwischen View und Editor. Wenn sich am Modell was ändert, schickt es ein Ereignis(vom Typ ModelChangeEvent) an die View, die das Ereignis an den Editor weiterleitet. Die View-Elemente registrieren sich bei ihren assoziierten Modellelementen als ModelChangeListener. Die Daten aus der View-Ebene sind mit denen aus der Modell-Ebene synchronisiert. Der umgekehrte Informationsfluß erfolgt über typsichere Methodenaufrufe.

> Für die Benachrichtigung von Änderungen regestriert sich jedes View-Element ausschließlich bei seinem assoziierten Modellelement als ModelChangeListener. Dabei vereinigt eine Schnittstelle die Modelloperationen, die nach dem AUTOFOCUS-Metamodell mit dem betreffenden Modellelement verbunden sind. Eine Unterkomponente ist mit ihrer Oberkomponente assoziiert. Dementsprechend erfolgen Modelloperationen an Unterkomponenten über das Wurzelobjekt, das die Oberkomponente als ihr Modellelement assoziiert. Die Ports einer Komponente werden im Metamodell mit dieser assoziiert. In der Diagrammebene treten Ports in zwei unterschiedlichen Ausprägungen auf, nämlich als Teil der internen oder externen Sicht einer Komponente. Dementsprechend realisiert die View einer Komponente Port-Operationen auf der externen Sicht auf eine Komponente und das Wurzelobjekt diejenigen der internen Sicht. Eine Komponente aggregiert außerdem ihre Kanäle. Deshalb werden die Operationen auf diesen über das Wuzelobjekt angestoßen. Modelländerungen, die die Assoziationen zwischen Modellelementen betreffen, werden im Wurzelobjekt bearbeitet. Änderungen an den Attributen

werden vom referenzierten View-Element angestoßen[3].

4.3.2 Schnittstelle zwischen View und Editor

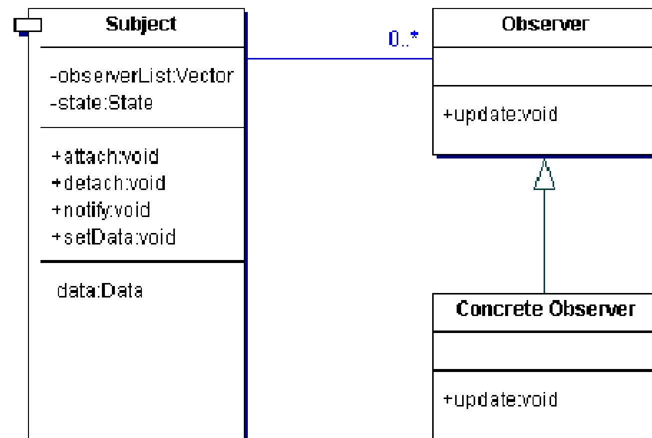

Abbildung 4.3: Statische Struktur des Observer-Entwurfsmusters.

Für den Informationsfluss zwischen der View- und Editorebene ist der Einsatz des Entwurfsmusters Observer(siehe Abb.4.3) sinnvoll.[4] Das Subjekt besitzt eine Liste aller registrierten Observer. Der Ablauf eines typischen Interaktionsmusters zwischen Observer und Subjekt ist in Abbildung (4.4) dargestellt.

Ein Observer registriert sich bei einem Subjekt. Sobald der Zustand des Subjekts verändert wird, wird über die Notify()-Methode der Aktualisierungsmechanismus beim Observer aufgerufen. Bei der Benachrichtigung des Observers mit der update()-Methode wird durch die Methode getData() der aktuelle Zustand des Subjekts dem Observer zur Verfügug gestellt. Die Reregistrierung erfolgt durch die Methode detach(). Das Subjekt bleibt von seinen registrierten Objekten unabhängig, weil diese die Bearbeitung der Aktualisierung selbst vornehmen. Für das in dieser Arbeit erarbeitete Editorenkonzept bedeutet dies die Realisierung der Editorelemente, d.h. Editorfenster, Figuren für Ports, Kanäle und Komponenten, als Observer der View-Elemente, die als Subjekte implementiert sind. Bei der Anwendung des

[3]aus der Diplomarbeit von Andreas Schweiger!

[4]Für mehr Informationen über das Konzept der Realisierung: Diplomarbeit von Andreas Schweiger nachschlagen!

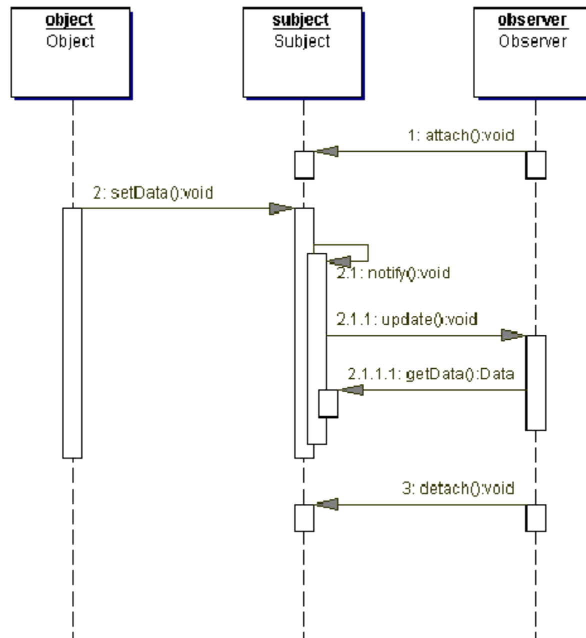

Abbildung 4.4: Eine typische Interaktionsfolge bei der Anwendung des Observer-Entwurfsmusters.

Observer-Entwurfsmusters bleibt die View-Ebene von der des Editors unabhängig. Alle Operationen, die im Wurzelobjekt implementiert sind, werden in der Editorebene durch das Editorfenster realisiert. Z.B. wird das Einfügen von neuen Komponenten durch das Editorfenster realisiert. Hingegen werden alle Änderungen an den grafischen Attributen zu ihrer Behandlung von den View-Elementen an die grafischen Elemente delegiert. Beispielsweise wird bei einer Größenänderung einer Komponente die grafische Figure benachrichtigt, die diese Komponente repräsentiert. Die Schnittstellen in der View-Ebene finden damit ihre Entsprechungen in der Editorebene[5].

4.3.3 Steuerung durch Modell- und View-Ebene

Die Steuerung des Editors erfolgt durch die Informationen in der View- oder der Modell-Ebene. Nur die Operationen, die vom Benutzer gewünscht sind und gleichzeitig von den betreffenden Klassen des Modells oder der View erlaubt werden, werden auch im Editor reflektiert. Alle anderen Aktionen

[5]aus der Diplomarbeit von Andreas schweiger!

werden im Editor nicht angezeigt. Die Steuerung durch das Modell delegiert die Aufgaben der Ausführung von Operationen und der Einhaltung von Konsistenzbedinungen an diese Schicht. Die Behandlung von Ausnahmesituationen, bei der vom Benutzer gewünschte Aktionen nicht ausgeführt werden können, erfolgt zentral im Modell. Verändern beispielsweise zwei Benutzer ein Modellelement gleichzeitig, muss dieser Anwendungsfall vom Modell geregelt werden. Möchte ein Benutzer eine Komponenten löschen, während ein anderer einen neuen Port einfügen möchte, können inkonsistente Zustände auftreten, falls die Aktionen nicht zentral im Modell realisiert sind. Die Einhaltung von Einschränkungen auf der View-Ebene werden bei diesem Ansatz auch erleichtert. Einschränkungen können auftreten, wenn das Verschieben von grafischen Elementen im Editor nicht möglich ist, weil dies durch einen Layout-Algorithmus in der View-Ebene verhindert wird. Die Implementierung solcher Einschränkungen erfolgt zentral in der Ebene der Views. Der Editor wird bei diesem Ansatz zu einer Visualisierungskomponente ohne semantische Information über die anderen beteiligten Komponenten(*)[6].

4.4 Änderungen gegenüber der Diplomarbeit

Die Diplomarbeit von Andreas Schweiger beschreibt die Entwicklung des SSD-Editors, wobei sie zum größten Teil die Entwicklungsphase(Analyse der Anforderung, mögliche Editoren, Wahl der Realisierung mit Begründung, usw..) umfasst. Das SEP ist eine Erweiterung (bzw. Verbesserung) der Diplomarbeit. Das Konzept (aus den vorherigen Abschnitte) ist das gleiche, nur die Implementierung ändert sich ein bisschen. Andere Änderung , die GEF betreffen, werden im Anhang(siehe 7.1) erklärt.

- **Die Autofocus-Ports:** Die Diplomarbeit verwendet Ports, die ursprünglich Knoten(englisch Node) sind. Die Native Ports von GEF haben den Vorteil, dass sie ihrer logisch zugehörigen Komponente direkt zugeordnet sind. Das kann man vor allem im Channel-Routing-Mechanismus ausnutzen, da nun bekannt ist, an welcher Kante der übergeordneten Komponente der Port sitzt und in welche Richtung der Channel folglich aus der Komonente austreten soll. Da die in GEF integrierten Routingmechanismen so arbeiten, kann man diese für native GEF-Ports fast unverändert übernehmen. Außerdem wird letztlich der GEF-Graph so übersichtlicher, weil man nicht für jeden Port eine Dummy-Komponente(unsichtbare Komponente) mitintegrieren muss. Native Ports haben dagegen die Eigenschaft, direkt über einer vorhandenen Komponente konstruiert zu werden.
- **Das ReturnCode-Konzept:** wurde auf Exceptions umgestellt. Dadurch ist der Quellcode der Diplomarbeit kürzer und übersichtlicher geworden. Beispiel: in der

[6] (*)aus der Diplomarbeit von Andreas Schweiger. Für mehr Informationen Diplomarbeit nachschlagen!

Methode removeSubComponents(SSDComponentView componentView) der Klasse SSDComponentRootView wird folgenden Code entspreschend verändert.

```
log.info("Going to remove subcomponent.");
Component component = (Component) componentView.getModel();
if (component.hasSubComponents()) {
return new ReturnCode(ReturnCode.COMPONENTS);
}
if (component.hasPorts()) {
return new ReturnCode(ReturnCode.PORTS);
}
if (component.hasSuperComponent()) {
Component sup = component.getSuperComponent();
```

Der veränderte Code:

```
throws ConstraintsViolatedException {_log.info("Going to remove subcomponent.");
Component component = (Component) componentView.getModel();
if (component.hasSubComponents()) {
throw new ConstraintsViolatedException("cannot remove subcomponent: subcomponent has
}
if (component.hasPorts()) {
throw new ConstraintsViolatedException("cannot remove subcomponent: subcomponent has
}
```

- **Hashtables:** einige in globalen Parametern (v.a. Hashtables) gehaltenen Informationen wurden in die Quest-Event-Struktur integriert, da globale Variablen meist der Lesbarkeit und vor allem Wartbarkeit abträglich sind. Z.B. in den Klassen ComponentRootView und ComponentView: alle Methoden, die die untergeordnete Objekte hinzufügen (addSubComponent, addExternalPort, usw..) verpacken die Positionsinformationen an ein von ModelChangeEvent abgeleitetes ModelChangePosEvent-Objekt. Früher gab es zu diesem Zweck im ext-Verzeichnis der Model-Views ein paar Klassen (SSDRootViewHelpers, SSDComponentViewHelpers, usw..) in denen in einer Hashtable die Assoziationen zwischen darzustellenden Objekten und deren zugedachten Positionen gespeichert wurden. Diese Klassen existieren jetzt nicht mehr, da sie durch die neue Klasse ModelChangePosEvent ersetzt wurden.
- **Das Autorouting:** Das Autorouting der Diplomarbeit wurde etwas verbessert, in dem zwei Komponenten berücksichtigt und sie umgeht. Außerdem wurde die Option eingefügt, dass das Autorouting ausgeschaltet wird(siehe Abb.4.5) so dass, der Benutzer per Hand routen kann.
- **Automatische Benennung:** die Benennung von Komponenten, Ports und Kanäle erfolgt in dem SEP automatisch (Bsp: Komponente1, Komponente2, usw..) In

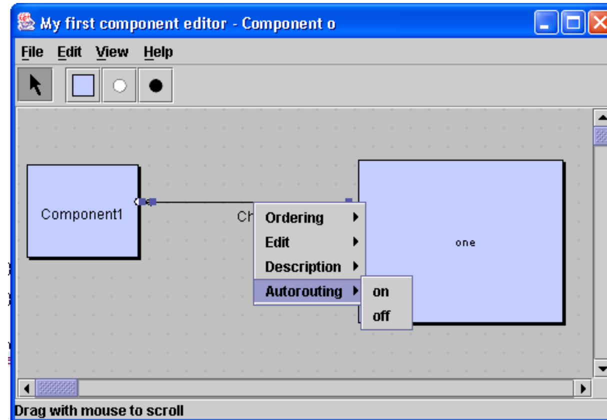

Abbildung 4.5: Autorouting kann ausgeschaltet werden.

der Diplomarbeit hingegen, erschien ein Dialog(siehe Abb.4.6), den man ausfüllen musste, bevor die Komponente, der Port oder den Kanal überhaupt gezeichnet werden konnte. Die manuelle Änderung ist aber wie in der Diplomarbeit noch erhalten.

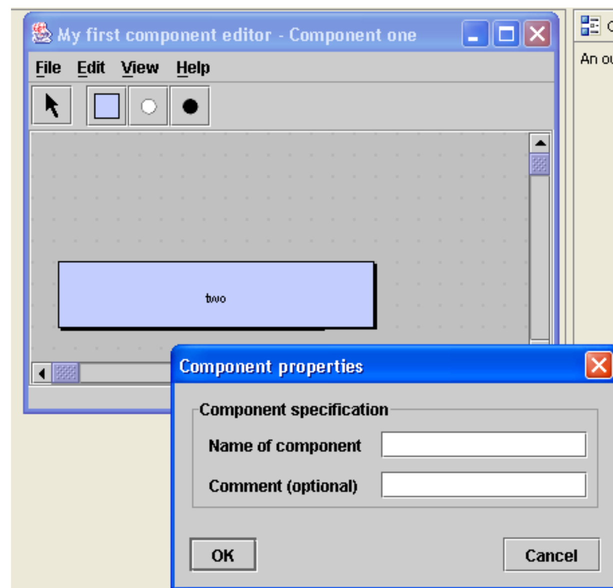

Abbildung 4.6: Dialog für Komponente

- **Die externen Ports:** durch das Umstellen von den Ports aus der Diplomar-

beit auf Native-Ports funktionnierten die externen Ports nicht mehr. Jetzt ist es wieder möglich, externe Native-Ports bei eine Subkomponente einzufügen(siehe Abb.4.7). Das ist aber bei einer Root-Komponente nicht möglich (fehlermeldung, siehe Abb.4.8) Interne Ports hängen dabei naturgemäß an der Komponente, an deren Kante sie dargestellt werden, externe Ports werden über einer, pro EditorFrame existierenden Dummy-FigComponent-Komponente konstruiert, welche selbst nicht im GEF-Graphen hängt und somit weder angezeigt noch verändert werden kann.

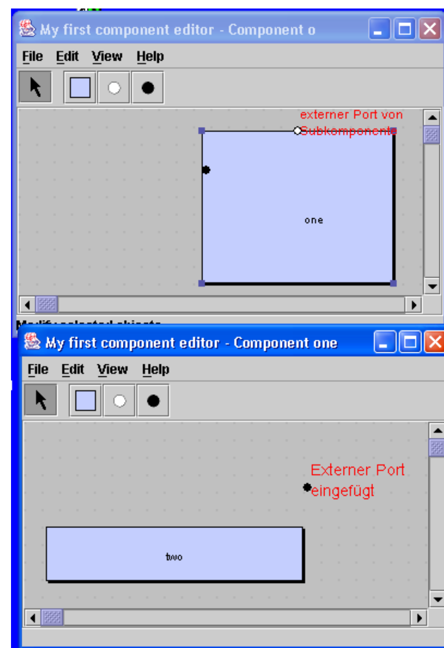

Abbildung 4.7: Einfügen eines externen Ports

Beim Anlegen eines externen Ports wird bei der übergeordneten Komponente ein interner Port zugeordnet (bzw. angezeigt) und umgekehrt.

- **Öffnen einer Subkomponente:** beim Doppelklick auf einer Komponente wird einen neuen SSD-Editor mit der Subkomponente(falls vorhanden)oder auch leer(falls keine Subkomponente vorhanden) geöffnet. Eine andere Möglichkeit wäre die Benutzung des Menü-Items *Open* (siehe Abb.4.9)
- **Shortcuts: Einfuegen von Kanaelen => Anlegen von zwei Ports und einem Kanal:** wird als erweiterter Modus eingefügt, in dem man zwei Komponenten auswählen kann und damit automatisch Input- und Outputports als auch der verbindende Channel in einem angelegt werden. Das Modus wird über eine Modifier-Taste auf den aktuellen Mode-Stack gepusht. Dafür wird einen neuen

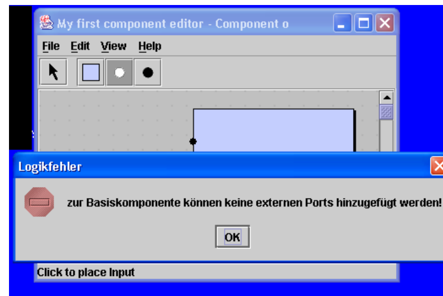

Abbildung 4.8: Es können bei eine Root-Komponente keine externe Ports eingefügt werden.

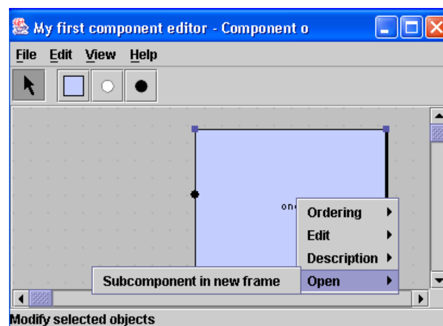

Abbildung 4.9: Öffnen eines neuen Editors für Subkomponente

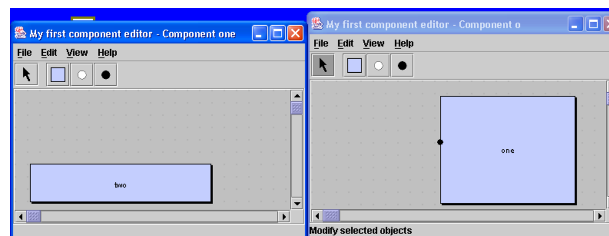

Abbildung 4.10: Subkomponente

Button im Editor eingefügt(siehe Abb.4.11).

4.5 Weitere Änderungen

Ein Großteil des eigentlichen SSDEditors (also nicht des QuestBrowser-Fensters) darstellt gewissermaßen eine Erweiterung vorhandener GEF-Klassen. Diese Erweiterungen

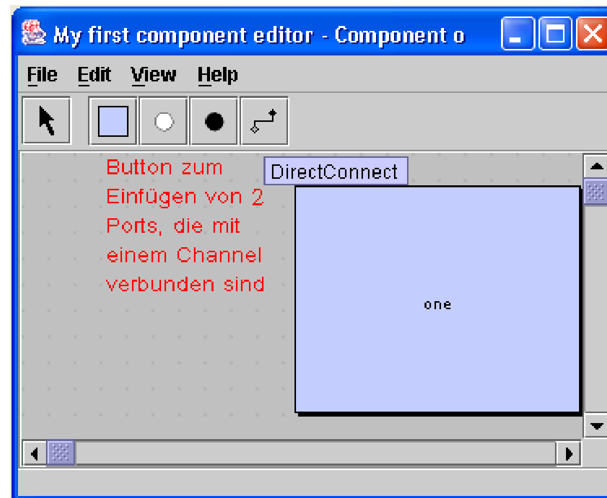

Abbildung 4.11: Shortcut:Channel

werden im Anhang(Änderungen am GEF) erläutert aber hier sind ein Paar dieser Änderungen.

- GEF bietet vorgefertigte Routing-Algorithmen, die aber vor allem keine externen Ports unterstützen. Ausserdem routen sie nicht immer optimal(*)[7].
- Handling: wenn der Benutzer einen Port anklickt, wird es nun automatisch in den Selektionsmodus(MyModeSelect) verzweigt. Um einen Channel zu ziehen, werden bei selektierten Ports zwei Konnektoren angezeigt(siehe Abb.4.12), welche auf Mausklick den CreateChannel-Modus aktivieren. Dadurch wird unterschieden, ob der Benutzer einen Port verschieben oder einen Kanal zeichnen möchte.

[7](*)mehr zu Autorouting im vorherigen Kapitel: Änderungen gegenüber Diplomarbeit

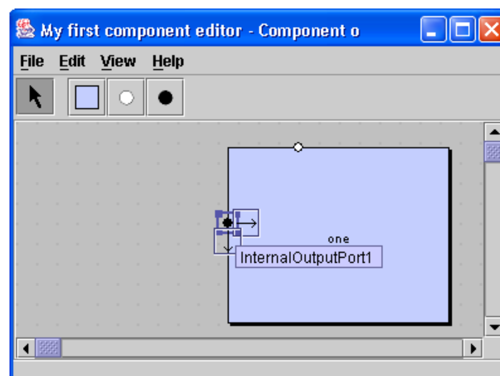

Abbildung 4.12: Die Zwei Konnektoren eines Ports

5 Implementierung

Die drei Ebenen Modell, View und Editor verfügen über Elemente. Modellelemente sind Elemente aus dem AUTOFOCUS-Metamodell. Einige der wichtigsten Elemente sind:

- `Component`
- `Port`
- `Channel`

View-Elementen sind Elemente, die die Modellelementen um die Anzeigeinformation erweitern. Hier sind einige wichtige Elemente davon:

- `SSDComponentRootView`
- `SSDComponentView`
- `SSDChannelView`
- `SSDPortView`
- `SSDExternalPortView`
- `SSDInternalPortView`

  Ein Element aus der Editorebene repräsentiert ein View-Element an der Benutzerschnittstelle. Demnach sind in dieser Kategorie die grafischen Figuren und die gesamte Zeichenfläche des Editors zusammengefasst[1].

Die Implementierung des ganzen Konzepts ist in der Diplomarbeit von Andreas Schweiger detailliert dargestellt. In diesem Kapitel wird ausschließlich die Implementierung der Erweiterungen und der Änderungen gegenüber der Diplomarbeit erläutert.

5.1 Java-Klassen

Die in dem letzten Kapitel vorstellten Konzepten der Änderungen befinden sich in den Java-Klassen des Source-Codes. Nicht alle Klassen sind aber von den Änderungen betroffen. Deshalb wird in folgender Tabelle aufgelistet, welches Konzept zu welcher Klasse gehört.

[1] Andreas Schweiger/Diplomarbeit

Konzept	Klasse
Singleton-Pattern	QuestBrowser
Native+externe Ports	EditorFrame, FigPort, PortNode, SelectionPort, MyNetEdge, MyModeSelect, MyModeModify, MyModeManager
Exceptions vs. Return-Code	ModeCreateChannel, SSDComponentRootView
Hashtables vs. Quest-Event	SSDComponentRootView, SSDComponentView
Autorouting	CmdAutoRoute, FigChannel
Automatische Benennung	SSDComponentRootView, ModeCreatChannel, ModePlaceComponent, ModePlacePort
Öffnen von Subkomponente	MyModeSelect, MyModeModify, CmdDescend, QuestBrowser
Shortcut: Kanäle	ModeDirectConnect, EditorPalette

Tabelle 5.1: Tabelle der geänderten Klassen

5.2 Implementierungsdetails im exemplarischen Durchlauf

Im Folgenden wollen wir die einzelnen implementatorischen Neuerungen gegenüber der zugrundeliegenden Diplomarbeit in einem, dem Aufrufschema entsprechenden, exemplarischen Durchlauf näher erörtern.

5.2.1 Initialisierung / Laden eines Models (QuestBrowser)

Die **main**-Methode der QuestBrowser-Klasse ist der Einstiegspunkt in das Hauptprogramm. Hier werden die externen Resourcen eingebunden und die Klasse instanziiert sich anschließend selbst. Im Gegensatz zur Diplomarbeit wurde die QuestBrowser-Klasse auf das Sigleton-Entwurfsmuster umgestellt, da ja logisch immer nur ein Exemplar dieses Browsers pro Programm-Instanz zur Verfügung stehen sollte. Die weitere Konstruktion des QuestBrowsers sowie das Laden eines auf einem Datenträger abgelegten Quest-Modells funktioniert analog zur original AUTOFOCUS-Implementierung.

Im Weiteren wird vorrausgesetzt, dass ein gültiges Repository, mit darin enthaltenem Projekt und einer Rootkomponente bereits geladen oder angelegt wurde.
Öffnet man nun für die vorhandene Rootkomponente einen neuen SSDEditor (mittels Edit→Create SSD im Menü des QuestBrowser-Fensters), so wird mittelbar über BrowserMenuBar→createSSDItem→/ActionListener/→ShowEditor (falls nicht schon vorhanden) ein zur Rootkomponente passender View erzeugt und die Methode QuestBrowser::openFrame aufgerufen, welche einen auf diesem SSDComponentRootView basierendes **EditorFrame** öffnet. (An diesem Konzept hat sich gegenüber der Diplomarbeit nichts geändert.)

5.2.2 Anlegen eines SSD-Editors, native und externe Ports (EditorFrame, FigPort, PortNode, SelectionPort, MyNetEdge, MyModeSelect, MyModeModify, MyModeManager)

Diese Klasse repräsentiert den eigentlichen SSD-Editor. Unter anderem werden hier die GEF-Darstellungen zu den Subviews des zugeordneten Rootviews erzeugt und in den GEF-Graphen eingehängt. Die dafür zuständige **addNode**-Methode musste im Hinblick auf die Kanäle an den Umstand angepasst werden, dass native GEF-Ports (siehe 4.4) mit den ihnen übergeordneten Komponenten direkt verknüpft sind, die mit den Portviews über den **DataContainer** verbundenen **FigPort**-Objekte sind also direkt den entsprechenden **NetPorts** zugeordnet. Zur Konstruktion des **FigChannel**-Objekts notwendige FigPorts lassen sich daher vergleichsweise einfach beschaffen.
Auch externe Ports sind nun durch native GEF-Ports implementiert, weshalb sie ebenfalls eine zugeordnete Elterkomponente benötigen. Da zum Rootview des aktuellen Editorframes normalerweise keine ComponentNode vorhanden ist, wird bei der Konstruktion jedes EditorFrame-Objektes eine rootNode-Instanzvariable angelegt, welche nicht in den GEF-Graphen eingehängt ist und somit auch nicht angezeigt wird. Diese rootNode-Komponente wird nun als Dummy für die Erzeugung der externen Ports für die dargestellte Rootkomponente verwendet.
Sowohl die Klassen, welche die Ports im GEF-Graphen darstellen (**PortNode** und Subklassen) als auch die graphischen Repräsentationen (**FigPort** und Subklassen) mussten nur geringfügig modifiziert werden (PortNode hat nun kein Attribut „center“ vom Typ **ConnectingPort** mehr (diese Klasse existiert nicht mehr), sondern ist selbst direkt von **NetPort** abgeleitet, FigPort packt nun den Port nícht mehr aus der NetPort-Instanz mittels getPort aus, sondern bindet sich direkt an das NetPort-Objekt. In den Subklassen von NetPort und FigPort als auch an **SelectionPort** wurden teilweise ebenfalls kleine Änderungen mit ähnlichem Ziel vorgenommen.)
Um die normalerweise fixen nativen Ports verschiebbar zu machen, mussten ausserdem die Klassen **MyModeSelect**, **MyModeModify** und **MyModeManager** angepasst werden. (Letzterer darf beim Selektieren nicht in den ModeCreateChannel sondern in

den MyModeSelect schalten. Der ModeCreateChannel dagagen wird jetzt erst beim Klick auf einen der Port-Konnektoren aktiviert. Da der ModeManager ursprünglich nicht als veränderbare Klasse geplant war, mussten eine Reihe weiterer Klassen überschrieben werden, um den neuen ModeManager in die GEF-API einzuhängen (genaueres siehe 7). Speziell für die Behandlung externer Ports wurden die Referenzen auf die beiden möglichen graphischen Repräsentationen **externalFig** und **internalFig** in die Superklasse PortNode verschoben um auf dieser Ebene eine generische **getFigure**-Methode anbieten zu können, welche in **MyNetEdge** zur unkomplizierten Darstellung der jeweils passenden Form Verwendung findet.

5.2.3 Weitergabe von Positionierungsinformationen durch Events (SSDComponentRootView / SSDComponentView)

Die Frage, welche Komponenten, Ports oder Kanäle im aktuellen Status gerade eingefügt werden können, kann letztlich erst in der Model-Schicht entschieden werden. (z.B. könnte ein Teil des Models gerade gesperrt sein, da es momentan von einer anderen Person bearbeitet wird.) Deshalb finden Einfügeoperationen nicht direkt im Editor statt, sondern es wird lediglich eine ensprechende Anfrage in Form eines Events über den View an das Model gesandt. Dort wird nun geprüft, ob das gewünschte Objekt angelegt werden kann. Ist dies der Fall, erzeugt das Model einen geeigneten Datensatz, schickt das Erzeugerevent an die Viewschicht zurück, welche ihrerseits ein entsprechendes View-Objekt anlegt und letztlich das Event an den Editor weitergibt. Dort wird nun (analog zum Laden eines gespeicherten Models) eine grafische Repräsentation für das darzustellende Objekt angelegt.
Da Positionsangaben im View abgelegt werden, der vom Benutzer gewünschte Ort zum Zeitpunkt der Erzeugung allerdings nur der Editorschicht bekannt sind, könnte man die Positionsinformation z.B. in Form einer Tabelle in der Editorschicht halten, um diese dann, falls das zu erzeugende Objekt von der Model- und der Viewschicht abgesegnet wurde, an die View-Schicht weiterzureichen bzw. um den graphische Repräsentanten an der gewünschten Stelle zu positionieren. Dieser Ansatz wurde in der Diplomarbeit realisiert. Da man hierzu potentiell irrelevante Informationen (wenn das Objekt von der Model- oder der View-Schicht verworfen wird) in designtechnisch unschönen Tabellen (z.B. Hashtables) mit großem Sichtbarkeitsbereich (im schlimmsten Fall global) benötigt, haben wir auf ein alternatives Verfahren gesetzt, das die Positionsinformationen direkt in das durchgereichte **ModelChangePosEvent** verpackt. Wird das Objekt und damit auch das Event auf einer beliebigen Ebene verworfen, so wird auch die nun nicht mehr benötigte Positionsierungsinformation entsorgt. Soll das Objekt tatsächlich dargestellt werden, wird die Information auf der anderen Seite auf dem Weg Editor→View→Model→View→Editor vom oben nach unten und wiederum nach oben

durchgereicht und ist sowohl auf der View- als auch in der Editor-Schicht bei Bedarf greifbar.

5.2.4 Exception-Konzept (ModeCreateChannel, SSDComponentRootView, etc.)

In der Diplomarbeit wurden Statusinformationen und Fehlerfälle durch Integerkonstanten, sog. **ReturnCodes** indiziert. Dieses Vorgehen hat einige entscheidende Nachteile: Da ohne weitergehende Dokumentation über die Methoden bis auf den Typ des Rückgabewertes nichts bekannt ist, wird oft nicht intuitiv klar, welche Fehlerfälle in der besagten Methode auftreten können. Dies führte bei der Diplomarbeit oft dazu, dass an diversen Stellen unnötige Abfragen in langen, per copy-paste vervielfältigten switch-case-Ketten abgefragt wurden. Ausserdem muss der Entwickler selbst dafür Sorge tragen, dass der Ort des aufgetretenen Fehlers ausreichend genau lokalisierbar ist.
Deshalb haben wir die ReturnCodes über Bord geworfen und benutzen an ihrer statt nun das von Java zur Verfügung gestellte Konzept der **Exceptions**. Hier kann man schon an der Signatur erkennen, welche Exceptions von einer bestimmten Methode geworfen werden können. (Das Fangen einer nicht geworfenen Exception führt konsequenterweise zu einem Compiler-Error.) Ausserdem lässt sich ein potentiell auftretender Fehlerfall anhand des automatisch generierten Stacktraces über beliebig verschachtelte Aufrufe hinweg zuverlässig lokalisieren.
Der Overhead des Exception-Konzeptes gegenüber der „hardcore" Integer-Fassung sollte wegen der recht effizienten Implementierung direkt in der virtuellen Maschine recht gering ausfallen und durch die Vorzüge in Sachen Übersichtlichkeit und Komfort mehr als wettgemacht werden.

5.2.5 Autorouting (CmdAutoRoute, FigChannel)

Viele der im Zuge des SEPs vorgenommenen Änderungen hatten das Ziel, das Routing-Verhalten für Kanäle zu optimieren (siehe v.a. natives Portkonzept, Abschnitt 4.4). Dabei wurden zwei verschiedene Routing-Modi vorgesehen: Ein analog zur Diplomarbeit funktionierendes, manuelles Routing und ein Autorouting-Mechanismus, welcher selbstständig versucht, aus der Lage der Komponenten und Ports einen weitgehend sinnvollen Pfad für die verbindenden Kanäle zu errechnen.
Das Umschalten der beiden Routing-Verfahren ist kanalweise bequem über das Kontextmenü (rechte Maustaste) möglich. Dazu wurde der GEF-Standard-Vorgehensweise gemäß ein neues Kommando (**CmdAutoRoute**) angelegt, welches im jeweiligen **FigChannel**-Objekt das gewünschte Attribut setzt. In FigChannel wurde nun der von der Superklasse **FigEdgeRectiline** geerbte Routing-Mechanismus dahingehend angepasst, dass der Kanal, falls sich die jeweilige Instanz im Autorouting-Modus befindet, nicht nur

beim Erzeugen, sondern bei jeder Veränderung automatisch neu ausgerichtet wird. Außerdem wurden einige Veränderungen am Routingmechanismus vorgenommen, um etwa externe Ports zu berücksichtigen. (Das Routing-Rechteck zur Bestimmung des zweiten Stützpunktes wird dort nicht mithilfe der im Fall eines externen Ports nicht vorhandenen übergeordneten Komponente, sondern manuell aus der Lage des Ports selbst ermittelt.)
Neben dem von FigEdgeRectiline zur Verfügung gestellten Routing-Algorithmus werden in GEF noch andere vorgefertigte Routing-Ansätze mitgeliefert, die man als Ausgangspunkt für eigene Experimente verwenden kann (etwa FigEdgePoly, FigEdgeLine oder FigEdgeLineDotted). Alle haben jedoch gemeinsam, dass sie jeweils nur die Ausgangs- und die Zielkomponenten berücksichtigen und teilweise suboptimale Pfade berechnen - ein Ansatzpunkt für künftige Erweiterungen.

5.2.6 automatische Bezeichner (SSDComponentRootView, ModeCreateChannel, ModePlaceComponent, ModePlacePort)

Eine Teilaufgabe des SEPs bestand darin, ein automatisches Benennungsschema für Komponenten, Ports und Kanäle einzuführen, um es dem Benutzer zu ermöglichen, zügig das Grobdesign seiner Anwendung einzurichten und sich erst später um eine aussagekräftigere, manuelle Benennung zu kümmern. (In der Diplomarbeit musste man bevor man ein Objekt anlegen konnte, erst Namen und Kommentare dazu angeben - gerade in der Entwurfsphase recht mühsam.)
Dazu wurde auf View-Ebene (genauer: im jeweiligen Rootview) ein Mechanismus (**getNext [Objekttyp]Number**) geschaffen, der über alle diesem View zugeordneten Objekte vom gewünschten Typ iteriert und eine eindeutige Identifikationsnummer berechnet, die zur Generierung eines vorübergehenden Namens verwendet werden kann. Damit ist der Namensraum für jede Komponente lokal, zwei nicht direkt ineinander enthaltene Komponenten können also Subobjekte gleichen Namens enthalten - ein durchaus gewollter Effekt.

5.2.7 Subkomponenten-Ansicht (MyModeSelect, MyModeModify, CmdDescend, QuestBrowser)

Schon in der Diplomarbeit war es möglich, direkt über den Questbrowser mehrere SSD-Editoren zu jeweils verschiedenen (auch verschachtelten) Komponenten gleichzeitig offen zu halten. Dieser Mechanismus wurde nun dahingehend vereinfacht, dass ein gewöhnlicher Doppelklick auf eine Komponente einen neuen SSD-Editor öffnet, der die angewählte Komponente als Rootview verwendet und deren direkte Subkomponenten, Channels und Ports anzeigt. (Interne Ports des vorhergehenden Editors werden dort als externe Ports mit entgegengesetzter Funktion und Farbe dargestellt.)
Dazu wurde das neues GEF-Kommando **CmdDescend** eingeführt und sowohl der Selek-

tionsmodus (**MyModeSelect**) als auch der zum Verschieben zuständige Modus (**MyModeModify**) dahingehend umgeschrieben, dass sie bei einem Doppelklick auf eine Komponente das gewünschte Kommando ausführen. Das CmdDescend-Kommando bedient sich dabei der schon vorhandenen QuestBrowser-Methode **QuestBrowser::openFrame**. (Hier kommt das Singleton-Konzept des QuestBrowsers (siehe Abschnitt 5.2.1) zum Tragen, da der SSD-Editor so keine Referenz auf das QuestBrowser-Objekt halten muss, sondern sich die aktuelle Instanz direkt von der QuestBrowser-Klasse holen kann.)

5.2.8 schnelles Anlegen von Ports und Kanälen (ModeDirectConnect, EditorPalette)

Um zwei Komponenten über einen Kanal zu verbinden, muss man zwei Ports und ein Channel-Objekt erzeugen. Da dies recht häufig passiert und jedesmal mehrere Klicks seitens des Benutzers erfordert, wurde dem User ein Werkzeug an die Hand gegeben, um das Ganze ein wenig zu beschleunigen: Er muss nun lediglich den **DirectConnect**-Button drücken, auf die Ausgangskompoenente klicken, zur Zielkomponente hinüberziehen, loslassen, fertig.
Zur Realisierung dieser Funktionalität musste ein neuer Button in der **EditorPalette** angelegt werden, welcher über das **CmdSetMode**-Kommando in den neu geschriebenen **ModeDirectConnect** umschaltet. In dessen Mouse-Eventhandler-Methoden wird nun im Prinzip die Funktionalität von **ModePlacePort** und **ModePlaceChannel** in einem Schritt zusammengeführt. Danach wird der Editor angewiesen, mit Hilfe des ModeManagers wieder in den zuvor aktiven Modus zurückzuschalten.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

AUTOFOCUS 2 ist ein modellbasiertes CASE-Werkzeug, das die graphische Modellierung von Software ermöglicht. Das GEF-Rahmenwerk ereignet sich gut, für diese graphische Modellierung, da GEF ein Programmgerüst ist, das erweitert und verändert werden kann. Die Architektur und die Funktionenen von GEF werden dabei erläutert, um ihre Umsetzung im AUTOFOCUS2 besser zu verstehen.

> Um dieses GEF-Rahmenwerk zu verwenden, wird eine Schichtenarchitektur entwickelt, welche die Anbindung an den modellbasierten Anwendungskern ermöglicht. Das Design der Anwendung ist duch die Anforderungen der Modellbasierung und die Vorgaben des Rahmenwerks bestimmt. Die Schichtenarchitektur resultiert in der Erweiterbarkeit und Wartbarkeit der Software. Verändert sich das Modell oder wird eine andere Version des Rahmenwerks oder sogar ein anderer Editor eingesetzt, hat dies keine Auswirkungen auf die gesamte Anwendung. Die Schichten müssen zwar an die Veränderungen angepasst werden, jedoch ist Realisierung der Funktionalität in den Schichten gekapselt. Die Schichten verbergen ihre Implementierungsdetails und stellen ihre Dienste in Schnittstellen zur Verfügung. Das Design ist so gewählt, dass eine Schicht nur die unter ihr liegende kennt, aber von der Ebene über ihr unabhängig ist[1].

Ausserdem werden zusätzliche Funktionen (z.B. Öffnen einer Subkomponente oder Shortcuts für Kanäle) eingefügt und einige vorhandenen Funktionen der Diplomarbeit werden verbessert(z.B. Autorouting, externe bzw. interne Ports).
Bei der Implementierung wird einen exemplarischen Durchlauf beschrieben, der die aufgrufenen Klassen und Methoden erklärt. Daraus folgt, dass einige Änderungen an GEF notwendig waren, um die Realisierung des SSDEditors durchzuführen. Diese Änderungen werden im Anhang erläutert(siehe 7.1)

[1] aus der Diplomarbeit von Andreas Schweiger!

6.2 Ausblick

Die Entwicklung eines Editors (in unserem Fall der SSDEditor) benötigt viel Überlegungen, da man immer die beste Lösung sucht. Deswegen sind Optimierungen der schon vorhanden Funktionen immer willkommen. Das SEP ist teilweise eine Verbesserung und Erweiterung der Diplomarbeit aber seine Ergebnisse sind noch nicht endgültig.
Einige zukünftige Herausforderungen wären:

- Das Autorouting berücksichtigt bis jetzt nur zwei Komponenten. Also könnte es erweitert, in dem es über mehrere Komponente routet.
- Immediate Ports werden noch nicht unterstützt und es gibt noch kein Konzept, wie sie eingefügt werden könnten.
- Die Attribut-Fenster von den Ports oder von den Komponenten könnten erweitert werden.
- Realisierung von weiteren Quest-Editoren:
  1. Zustandsübergangsdiagramme (State Transition Diagramm, STD).
  2. Interaktionsdiagramme (Extended Event Traces, EET).
  3. Datentypdefinitionen (Data Type Definition, DTD).
- Synchronisationsmechanismen für einen Mehrbenutzerbetrieb einbauen, die durch das Konzept der Transaktionen realisiert werden können.
- Integration der Prozessunterstützung im Editor. Einige Teile davon sind: Realisierung von Konsistenzüberprüfungen oder das Integrieren einer Test- bzw. Simulationsumgebung.
- Erweiterungsmöglichkeiten der Editierfunktionen: z.B. Kopieren, Einfügen und Ausaschneiden von Elementen. GEF unterstützt diese Funktionalität in der Ebene des Editors aber der Editor muss die aktuelle Zusammensetzung des Systemmodells reflektieren.

7 Anhang

7.1 Änderungen an GEF

Für die Realisierung des SEP müssten wir Änderungen am GEF unternehmen. Einige Klassen wurden aus GEF-Klassen abgeleitet. Im Volgendem werden diese Klassen näher erläutert:

- **MyCmdOpen.java:** wird aus der GEF-Klasse CmdOpen abgeleitet und hat als einzige Unterschied gegenüber der ursprüngliche Klasse, dass sie MyJGraphFrame anstatt von JGraphFrame benutzt.
- **MyJGraphFrame.java:** ist eine Vererbung der Klasse JGraphFrame. Der Unterschied zwischen beiden Klassen liegt daran, dass abgeleitete Klasse MyJGraph statt JGraph und die passenden MyCmd*-Kommandos verwendet.
- **MyCmdOpenPGML.java:** wird aus der GEF-Klasse CmdOpenPGML vererbt. Diese Klasse arbeitet mit der Klasse MyEditor statt mit dem normalen quest-Editor zusammen und verwendet MyJGraphFrame statt JGraphFrame.
- **MyEditor.java:** wird aus der GEF-Klasse Editor abgeleitet. Sie benutzt die Klasse MyModeManager anstatt von der Gef-Klasse ModeManager, da in ModeManager immer ein Kanal gezeichnet wird, wenn ein Port selektiert ist. In MyModeManager wird das Verschieben von dem selektierten Port als Dafault-Verhalten eingesetzt.
- **MyModeManager.java:** ist eine Vererbung der GEF-Klasse ModeManager. Wenn ein Port selektiert wird, springt diese Klasse zu dem Select-Modus statt zu dem Create-Edge-Modus.
- **MyCmdOpenSVG.java:** wird aus der GEF-Klasse CmdOpenSVG vererbt. Diese Klasse arbeitet mit der Klasse MyEditor statt mit dem normalen quest-Editor zusammen und verwendet MyJGraphFrame statt JGraphFrame.
- **MyCmdSpawn.java:** ist eine Vererbung der GEF-Klasse CmdSpawn und verwendet MyJGraphFrame statt JGraphFrame.

Bei der ganzen Programmierung müssten wir die Klassen von GEF ändern. Diese Änderungen sind aber in dem Kapitel Implementierung(siehe 5.2 dokumentiert. Dort wird einen exemplarischen Durchlauf beschrieben, der die verschiedenen Neuerungen gegen-

über der zugrundeliegenden Diplomarbeit näher erläutert. Die oben genannten Klassen sind diejenigen, die weder in der Diplomarbeit noch im GEF vorhanden waren. Die meisten der Klassen resultieren aus der Umstellung der in der Diplomarbeit verwendeten Ports auf GEF-Native-Ports.

Literaturverzeichnis

[Schw02] Andreas Schweiger: *Modellbasiertes Editorenkonzept für AutoFOCUS 2* Diplomarbeit Institut für Informatik TU München 23.12.2002

[GEF] GEF-Doku: *GEF Documentation* http://gef.tigris.org