

Technische Universität München

Fakultät für Informatik

Diplomarbeit

Modellbasiertes Editorenkonzept für AUTOFOCUS 2

Andreas Schweiger

Aufgabensteller: Prof. Dr. Manfred Broy

Betreuer:

Peter Braun

Tobias Hain

Dr. Bernhard Schätz

Alexander Wißpeintner

Abgabedatum: 15.04.2003

Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, im Dezember 2002

Vorwort

Ich möchte mich bei meinen Betreuern Peter Braun, Tobias Hain, Dr. Bernhard Schätz und Alexander Wißpeintner bedanken. Ihre Rückmeldungen und intensive Betreuung trugen zum Gelingen der Diplomarbeit wesentlich bei. Sie regten in konstruktiven Diskussionen neue Ideen und Lösungsansätze für Problemstellungen an. Durch das von ihnen gezeigte Interesse an der Arbeit gelang es mir, die Motivation während der Zeit der Bearbeitung auf hohem Niveau zu halten. Erwähnen möchte ich auch Heiko Lötzbeyer und Dr. Oscar Slotosch, die mich speziell bei der Einarbeitung in Quest unterstützten.

Außerdem danke ich meinen Eltern, die mir die Ausbildung an der Universität erst ermöglichten. Nicht zuletzt ist es ihnen zu verdanken, den Abschluss des Studiums zu erreichen.

Inhaltsverzeichnis

Vorwort	v
1 Einführung	1
1 Motivation	1
2 Ziele	1
3 Struktur des Dokuments	3
2 AUTOFOCUS	5
1 AUTOFOCUS-Beschreibungstechnik	5
1.1 AUTOFOCUS-Metamodell	5
1.2 AUTOFOCUS-Views und -Diagrammarten	8
2 Quest	11
2.1 Integration von Werkzeugen	12
2.2 Änderungen am Modell	12
2.3 Generierung des Metamodells	14
2.4 Zugriffsstrukturen	15
3 Prinzipien im Softwareentwurf	17
1 Ziele in der Softwareentwicklung	18
2 Rahmenwerke	19
4 Evaluierung	21
1 Anforderungen an ein Editoren-Werkzeug	21
1.1 Allgemeine Anforderungen	21
1.2 An Quest orientierte Anforderungen	23
2 Bewertungssystem	24
2.1 Ausschlusskriterien	25
2.2 Erweiterte Kriterien	25
3 Evaluierete Software	26
3.1 Diagramm-Zeichenwerkzeuge	26
3.2 Meta-CASE-Werkzeuge	27
3.3 Fertige Editoren	28
3.4 Rahmenwerke für grafische Editoren	29
3.5 Visualisierungswerkzeuge für Graphen	33
3.6 Graphen-Editoren	34
3.7 Benutzerdefinierbare Entwicklungsumgebungen	35
4 Ergebnis der Evaluierung	35

5	Anforderungsanalyse	37
1	Anforderungsklassen	37
2	Rahmenbedingungen	37
3	Anforderungsanalyse des Editors	38
3.1	Benutzerschnittstelle	38
3.2	Funktionalität	39
3.3	Dokumentation	42
4	Anforderungsanalyse der Schnittstelle	42
4.1	Benutzerschnittstelle	42
4.2	Funktionalität	43
4.3	Weitere Anforderungen	44
5	Vertikaler Prototyp	44
6	Design	47
1	Architektur	47
1.1	Architekturprinzipien	47
1.2	Organisationsformen	48
1.3	Schichtenarchitektur	48
2	Schnittstelle	50
2.1	Ziele von Schnittstellen zwischen Programmteilen	50
2.2	Informationsfluss der Schnittstelle	51
2.3	Gruppierung der Schnittstellenoperationen	51
2.4	Schnittstelle zwischen Modell und View	52
2.5	Schnittstelle zwischen View und Editor	55
3	Aufteilung der Steuerung	57
3.1	Steuerung durch Modell- und View-Ebene	57
3.2	Steuerung durch die Editorebene	58
3.3	Mischform der Steuerung	59
4	Darstellung der gewählten Kommunikation zwischen den Schichten	59
4.1	Ablaufstruktur bei View-Operationen	60
4.2	Ablaufstruktur bei Modelloperationen	60
5	Bestimmung von Entwurfsentscheidungen durch externe Vorgaben	61
7	Implementierung	63
1	Kommunikation zwischen den Ebenen Modell, View und Editor	63
1.1	Kopplung der Modell-, View- und Editorelemente	63
1.2	Anforderung und Propagierung von Änderungen zwischen den Ebenen Modell, View und Editor	64
2	Erweiterungen an den View-Elementen	65
2.1	Modelloperationen	65
2.2	View-Operationen	74
2.3	Hilfsfunktionen	77
2.4	Probleme der Ereignisverarbeitung	78
3	Verwendung der erweiterten View-Elemente	79
3.1	Modelloperationen	79

3.2	View-Operationen	80
4	Beschreibung besonderer Klassen	81
4.1	ReturnCode	81
4.2	Action	82
5	Beziehungen zwischen View-, Modell- und Editorelementen	84
5.1	Graphenmodell des GEF-Rahmenwerks	84
5.2	Editor- und Graphenmodellelemente	85
5.3	View- und Editorelemente	89
5.4	Modell- und View-Elemente	91
6	Schwierigkeiten bei der Implementierung	91
7	Einhaltung von Konsistenzbedingungen	93
8	Integration in die automatische Generierung	94
9	Bestimmung von Implementierungsentscheidungen durch externe Vorgaben	98
10	Abdeckung der Anforderungen	99
10.1	Funktionalität des Editors	99
10.2	Vergleich zwischen Anforderungen und Implementierung	99
8	Synchronisationsmechanismen	103
1	Shared Whiteboard	103
2	Manueller Sperrmechanismus	105
9	Zusammenfassung und Ausblick	107
1	Zusammenfassung	107
2	Ausblick	108
A	Eigenschaften der evaluierten Software	111
1	AbsInt aiSee Graph Visualization	111
2	daVinci Presenter	112
3	Diva	113
4	GEF	116
5	Gift	119
6	GraphEd	120
7	Graphlet	120
8	ILOG JViews Component Suite	121
9	JGraph	122
10	JHotDraw	124
11	MetaEdit+	126
12	Tom Sawyer Software	127
13	VCG	128
14	VGJ	128
15	Visual Shape	130
16	Weitere Software	132

B Erläuterungen zur Implementierung	135
1 Software-Versionen	135
2 Paketstruktur	135
3 Erweiterungen am GEF-Rahmenwerk	135
4 Erweiterungen an Quest	137
Literaturverzeichnis	139
Index	143

Einführung

1 Motivation

Die steigende Komplexität von Software erfordert den Einsatz von CASE-Werkzeugen im Entwicklungsprozess. Unterschieden werden Softwareentwicklungswerkzeuge, die den gesamten Prozess unterstützen, und solche, die lediglich Teile davon umfassen. Ganzheitliche Werkzeuge, welche die Phasen Analyse, Design, Implementierung und Test unterstützen, haben den Vorteil, dass sie im Entwicklungsprozess nur die einmalige Einarbeitung für den Anwender erfordern.

Die Basis für die vorliegende Arbeit bilden die Projekte AUTOFOCUS und Quest. AUTOFOCUS ist ein CASE-Werkzeug-Prototyp zur Modellierung verteilter, nebenläufiger Systeme und unterstützt besonders die Phasen Implementierung und Test, eingeschränkt die Phase Design und praktisch überhaupt nicht die Analyse. AUTOFOCUS basiert auf einer sichtenorientierten Beschreibungstechnik. Dabei werden Ausschnitte aus dem Systemmodell durch Editoren visualisiert und modifiziert. Quest integriert eine Reihe von Validierungs- und Verifikationswerkzeugen zur Überprüfung der Korrektheit des modellierten Systems.

Bei dem in AUTOFOCUS integrierten Systemmodell handelt es sich um ein implizites Systemmodell, welches zur Laufzeit des Programms durch seine verteilte Struktur aus Implementierungssicht nicht einheitlich navigierbar und lesbar ist. Genau diese Unzulänglichkeit geht das Projekt Quest mit einem expliziten Systemmodell an, welches in diesem Zusammenhang auch als „modellbasierter Ansatz“ bezeichnet wird. Erst hierdurch wird die Anbindung weiterer Werkzeuge ermöglicht sowie die Basis für Teile einer Entwicklungsprozessunterstützung geschaffen. AUTOFOCUS 2 wird die Editorfunktionalität von AUTOFOCUS mit dem modellbasierten Ansatz von Quest verbinden und dadurch neue Konzepte in CASE-Werkzeugen umsetzen. Dazu wird in der vorliegenden Arbeit ein modellbasiertes Editorenkonzept entwickelt.

2 Ziele

Die Editoren von AUTOFOCUS sind teilweise mittels eigenentwickelter Lösungen für die Benutzerschnittstelle implementiert und erfordern bei der Erweiterung ihrer Funktionalität zusätzlichen Implementierungsaufwand. Um den Implementierungs-, Wartungs- und Erweiterungsaufwand für die Benutzeroberfläche zu verringern, werden Standardaufgaben durch vorgefertigte Lösungen wie Klassenbibliotheken, Komponenten oder Rahmenwerke realisiert. Durch den Einsatz existierender Software soll der Aufwand für die Implementierung der Editoren für AUTOFOCUS 2 reduziert werden. Die Realisierung von Baukästen zur Realisierung von Editoren oder Benutzeroberflächen ist aus wissenschaftlicher Sicht bereits abgehandelt, die Implementierung von Visualisierungswerkzeugen auf Basis vorgefertigter Software jedoch bietet noch Raum für wissenschaftliche Untersuchungen.

Zur Auswahl einer geeigneten Software-Lösung für die Realisierung eines Editors werden unterschiedliche Ansätze zur Erstellung von Editoren unter Berücksichtigung der Anforderungen für die Implementierung von Editoren für AUTOFOCUS 2 evaluiert. Eingegrenzt werden diese Ansätze durch die geforderte Funktionalität zur Visualisierung und Modifizierung von Graphen. Die Betrachtung dieser Werkzeuge reicht aus, weil die Diagrammarten von AUTOFOCUS 2 Graphen-Struktur besitzen.

Die untersuchten Werkzeuge werden nach ihren Ansätzen zur Erstellung von Editoren nach folgenden Kategorien klassifiziert: Diagramm-Zeichenwerkzeuge ermöglichen dem Benutzer die Konstruktion von Diagrammen, welche mit benutzerdefinierten grafischen Symbolen erweitert werden können. Die Elemente der Diagramme von AUTOFOCUS 2 sind dadurch realisierbar. Meta-CASE-Werkzeuge erlauben neben der grafischen Modellierung benutzerdefinierter grafischer Elemente auch die automatische Generierung von domänenspezifischen Entwicklungswerkzeugen. Meta-CASE-Werkzeuge eignen sich als Basis für einen Editor von AUTOFOCUS 2, weil die grafischen Elemente und die Syntaxregeln der Modellierung von AUTOFOCUS 2 spezifiziert werden können. Fertige Editoren sind Werkzeuge mit Editierfunktionalität, welche die Realisierung von beliebigen grafischen Elementen erlauben. Sie können an eine beliebige Domäne angepasst werden und können deshalb den Ausgangspunkt für die Realisierung eines Editors für AUTOFOCUS 2 bilden. Rahmenwerke für Editoren bieten das Grundgerüst für die Erstellung von Editoren. Die implementierten Klassen werden so erweitert, dass sie die Anforderungen der Zielanwendung erfüllen. Sie können an die gewünschte Domäne nicht nur in ihren Anzeigeeigenschaften, sondern auch in ihrer weiteren Funktionalität angepasst werden. Anwendungsrahmenwerke besitzen Einhängpunkte zur Integration anwendungsspezifischer Funktionalität und eignen sich zur Realisierung von Editoren für AUTOFOCUS 2. Visualisierungswerkzeuge für Graphen zeigen die Struktur von Graphen an und realisieren Funktionalität zum Datenaustausch mit anderen Anwendungen. Sie unterstützen ferner die Integration von Editierfunktionalität in die Anwendung. Diese Eigenschaften zeichnen die Visualisierungswerkzeuge als Basis für Editoren für AUTOFOCUS 2 aus. Bei Graphen-Editoren handelt es sich um dedizierte Werkzeuge zur Bearbeitung von Graphen. Diagramme von AUTOFOCUS 2 besitzen Graphen-Strukturen und können daher mit Graphen-Editoren bearbeitet werden. Benutzerdefinierte Entwicklungsumgebungen können an die Bedürfnisse der Benutzer und an die Anforderungen der Domäne angepasst werden. Sie unterstützen auch die Integration grafischer Beschreibungstechniken und eignen sich daher als Basis für Editoren von AUTOFOCUS 2.

Die Anbindung von Editoren an das zentrale Systemmodell erfordert eine Schnittstelle für die Kommunikation mit dem Editor. Dazu wird in dieser Arbeit ein modellbasiertes Editorenkonzept für AUTOFOCUS 2 erarbeitet. Eine generische Schnittstelle reduziert den Implementierungsaufwand und die Fehleranfälligkeit bei der Übertragung des Konzepts auf andere Editoren. Aus den allgemeinen Prinzipien im Softwareentwurf und den Anforderungen der Schnittstelle und des eingesetzten Werkzeugs zur Realisierung von Editoren werden die funktionalen und nicht funktionalen Anforderungen abgeleitet, welche die Basis für die Entwicklung des Designs für das Editorenkonzept bilden. Weiterentwicklungen des eingesetzten Werkzeugs zur Realisierung eines Editors oder die Anbindung eines beliebigen anderen Werkzeugs sollen im Entwurf des Konzepts berücksichtigt werden, um die Anforderung der Weiterentwicklungsfähigkeit der Anwendung zu erfüllen. Ebenso sollen durch das Design Änderungen des AUTOFOCUS-Metamodells, welches das zentrale Systemmodell beschreibt, ermöglicht, ohne die Architektur der gesamten Anwendung modifizieren zu müssen.

Das Editorenkonzept bildet die Basisarchitektur für modellbasierte Editoren. Das Design des Konzepts wird in die Implementierung eines prototypischen Editors umgesetzt. Diese Implementierung validiert das erarbeitete Konzept und durchleuchtet die Funktionalität der gewählten Software, welche die Basis für den Editor bildet. Ein weiteres Ziel der Implementierung auf der Basis des gewählten Ansatzes zur Realisierung eines Editors ist die Überprüfung, ob das Werkzeug für die Erstellung weiterer Editoren für AUTOFOCUS 2 geeignet ist.

Die vorliegende Arbeit zielt auf den Entwurf eines Editorenkonzepts für AUTOFOCUS 2. Der gegenwärtige Stand der Entwicklung für dieses Werkzeug ist die Implementierung von Quest. Deshalb wird in den weiteren Ausführungen dieser Arbeit stets der Bezug zu Quest dargestellt.

3 Struktur des Dokuments

Das folgende Kapitel 2 gibt einen Überblick über die Grundlagen von AUTOFOCUS und Quest. In Kapitel 3 werden allgemeine Prinzipien der Softwareentwicklung erläutert und die Konsequenzen der Verwendung von Rahmenwerken beschrieben. Aus den allgemeinen Prinzipien im Softwareentwurf und den Anforderungen an die Modellschnittstelle von Quest resultieren die Anforderungen an ein Rahmenwerk. Diese werden in Kapitel 4 erläutert. Die Evaluierung der Softwaresysteme zur Visualisierung und Modifizierung von Graphen erfolgt nach den Aspekten der erarbeiteten Anforderungen, um sie für den Einsatz als Basis für Editoren in AUTOFOCUS 2 zu untersuchen. Dabei werden nicht nur Rahmenwerke, sondern auch fertige Produkte mit der Möglichkeit zur Anbindung an andere Programmsysteme untersucht. Die Bewertung erfolgt dabei nach den in Kapitel 4 erarbeiteten Kriterien. Die Anforderungsanalyse für die Schnittstelle zwischen Modell und Editor in Kapitel 5 ist aus den Anforderungen an den Editor-Prototyp abgeleitet. In Kapitel 6 wird die Architektur des Editorenkonzepts vorgestellt. Für die Realisierung der Schnittstelle zwischen Editor und Modell ergeben sich Varianten, die erläutert und bewertet werden. In Kapitel 7 wird die Implementierung der Schnittstelle mit ihren Mechanismen zum Austausch von Informationen zwischen Modell und Editor beschrieben. Auch hier werden Alternativen für die Implementierung von Konzepten vorgestellt und bewertet. Kapitel 8 beschreibt nötige Erweiterungen des erarbeiteten Konzepts für den Mehrprogramm- und Mehrbenutzerbetrieb. Abschließend werden in Kapitel 9 die Ergebnisse der vorliegenden Arbeit zusammengefasst und auf Weiterentwicklungen des Editorenkonzepts hingewiesen. In Anhang A sind die Ergebnisse der Evaluierung der untersuchten Softwaresysteme ausführlich beschrieben. Hinweise zur Implementierung befinden sich in Anhang B.

AUTOFOCUS

1 AUTOFOCUS-Beschreibungstechnik

AUTOFOCUS [AF02] 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. Ein Beispiel für den Einsatz des Werkzeugs im Entwicklungsprozess ist in [FH02] beschrieben. 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 besitzt ein *dokumentbasiertes Repository*, in dem die folgenden Dokumente enthalten sein können:

- Systemstrukturdiagramme (System Structure Diagram, SSD)
- Zustandsübergangsdigramme (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 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.

1.1 AUTOFOCUS-Metamodell

Die Struktur des zentralen Systemmodells wird durch ein Metamodell beschrieben, welches in Abbildung 2.1 auf der nächsten Seite dargestellt ist und in den folgenden Abschnitten näher erläutert wird.

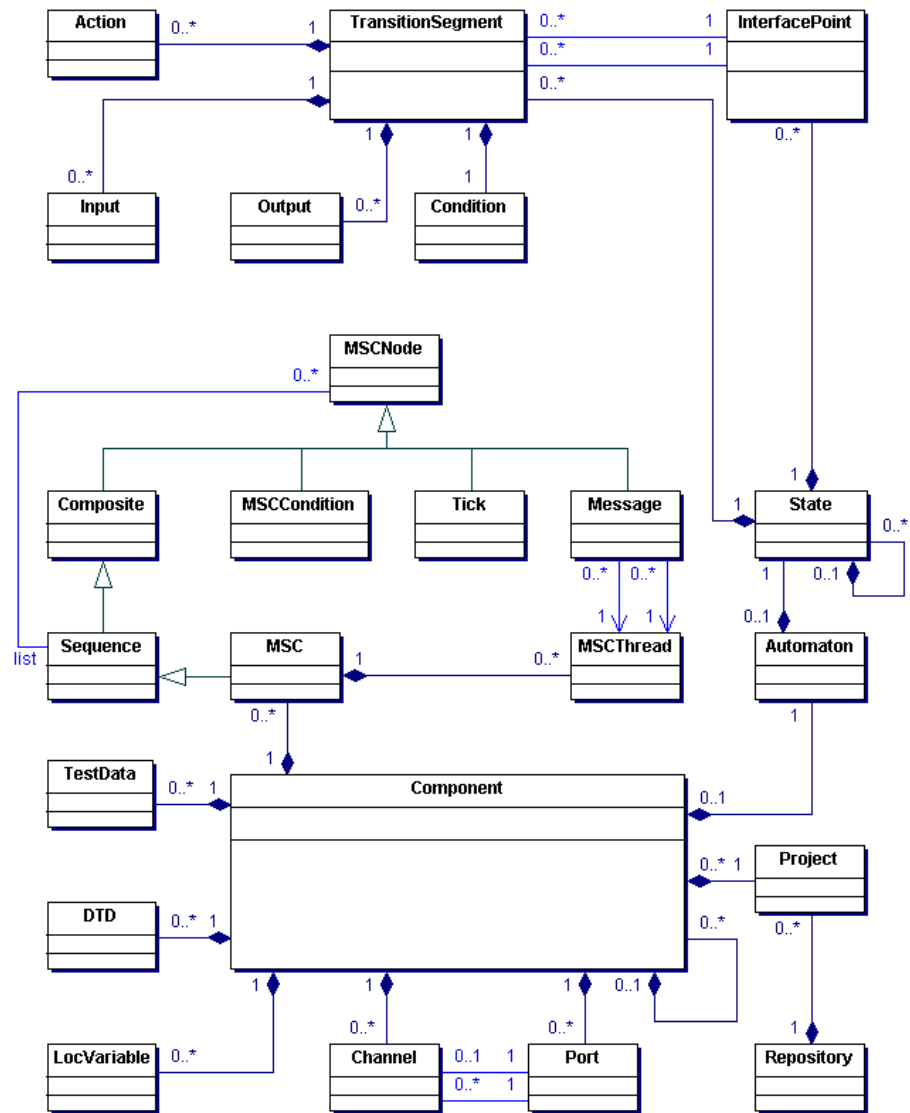


Abbildung 2.1. AUTOFOCUS-Metamodell

1.1.1 Komponenten

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 darstellen, ergibt sich die hierarchische Struktur der Modellbeschreibung.

1.1.2 Systemstruktur

Im zentralen Repository können mehrere Projekte angelegt werden. Jedes Project besitzt genau eine Komponente als zentrales Element, das durch ein Objekt vom Typ Component beschrieben wird. 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 Objekte 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. Jede Komponente kann eine Menge von lokalen Variablen vom Typ LocVariable besitzen.

Das Verhalten von Komponenten wird durch einen erweiterten endlichen Automaten spezifiziert. Im Metamodell übernimmt das Objekt Automaton diese Rolle. 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. Ein Zustand wird durch State repräsentiert. Die hierarchische Struktur findet auch in einem Automaten ihre Abbildung. So kann ein Zustand eine Menge weiterer Zustände aggregieren. Somit besteht ein Automat aus einem Basiszustand, der die eigentlichen Zustände assoziiert. Einzelne Zustände werden durch ihre Transitionsegmente vom Typ Transitionsegment miteinander verbunden. Dabei dient ein InterfacePoint als Schnittstelle zwischen einem solchen Segment und dem jeweiligen Zustand. Eine Transition aggregiert dabei weitere Objekte, die ihr Verhalten beim Übergang zwischen zwei Zuständen genauer spezifizieren. Durch Condition wird eine Vorbedingung beschrieben, die bei einem Zustandsübergang erfüllt sein muss. Die Objekte Input und Output geben die Eingabe bzw. die Ausgabe beim Übergang an. Die Aktionen, die bei einer Transition ausgeführt werden, sind durch Objekte vom Typ Action beschrieben. Aktionen ändern den Zustand einer Komponente und damit die Variablen vom Typ LocVariable. Input und Output referenzieren einen Port über MIF-Assoziationen. Das Attribut MIFPort der Ein- und Ausgabe kapselt dabei den assoziierten Port einer Komponente. Analog dazu wird über MIFLocVariable einer Aktion eine Referenz auf die Variable LocVariable einer Komponente hergestellt.

Die Datentypdefinition für eine Komponente wird durch das Objekt DTD dargestellt. Die Speicherung von Testdaten wird durch TestData realisiert.

Neben der Spezifikation des Verhaltens von Komponenten durch Automaten werden auch Sequenzdiagramme dazu verwendet, typische Abläufe und Interaktionsfolgen des Systems dar-

zustellen. Ein solches Diagramm wird durch `MSC` beschrieben. Dabei kann eine Komponente eine beliebige Anzahl an Interaktionsfolgen besitzen. Die Achsen eines Diagramms werden durch Objekte vom Typ `MSCThread` beschrieben. Eine Folge von Ereignissen vom Typ `MSCNode` bildet eine Sequenz vom Typ `Sequence`. Bei Ereignissen werden folgende Kategorien unterschieden: Ein Zeitschritt wird durch den Typ `Tick`, eine Bedingung durch `MSCCondition`, eine Nachricht durch `Message` und eine Folge von Ereignissen durch `Composite` repräsentiert.

1.1.3 Integrales Metamodell

Wie in Abschnitt 1.1.2 auf der vorherigen Seite erläutert, 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 ingetriet.

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.

1.2 AUTOFOCUS-Views und -Diagrammarten

Durch das Konzept der Views werden Daten nach Modell und Anzeigeinformationen getrennt gespeichert. Views sind Sichten auf einen Ausschnitt des Modells und werden in ihrer Struktur analog der Vorgehensweise beim Modell durch *Metaviews* beschrieben. Die Views tragen Informationen für die grafische Darstellung von Modellelementen. Beispielsweise besitzen Views Attribute für die Größe, Position oder Schriftart für die Visualisierung in einem Editor. Views projizieren also einen Modellausschnitt in ein Diagramm.

In AUTOFOCUS werden die drei Diagrammarten *SSD*, *EET* und *STD* unterschieden. Systemstrukturdiagramme (*SSD*) zeigen den Aufbau und die Schnittstelle eines Systems mit seinen Komponenten und Kanälen zum Datenaustausch. Ports dienen dabei als Verbindungspunkte zwischen einem Kanal und einer Komponente. Der Informationsfluss in den Kanälen ist gerichtet. Grafisch werden Komponenten durch Rechtecke, Ports durch Kreise und Kanäle durch rechtwinklige Kanten dargestellt. Abbildung 2.2 auf der nächsten Seite zeigt ein Beispiel für ein Systemstrukturdiagramm.

Views zeigen Modellausschnitte in Editoren oder Viewern an. Unabhängig von der Diagrammart besitzt jede Ausprägung einer View eine einheitliche Struktur. Dabei dient ein ausgezeichnetes *Wurzelobjekt* für die Speicherung aller in einem Diagramm enthaltenen View-Elemente. Das Wurzelobjekt repräsentiert einen Container, in dem alle View-Elemente gesammelt werden. Die Menge aller in einem Diagramm assoziierten Elemente wird als die *Objektmenge* bezeichnet.

Jedes View-Element referenziert ein Modellelement und repräsentiert dieses in einem Diagramm. Einem Modellelement können mehrere Views zugeordnet werden. So kann beispielsweise ein Port sowohl in einem Systemstruktur- als auch in einem Zustandsübergangsdigramm repräsentiert werden, in der Ebene des Modells ist der Port jedoch nur einmal enthalten. Damit

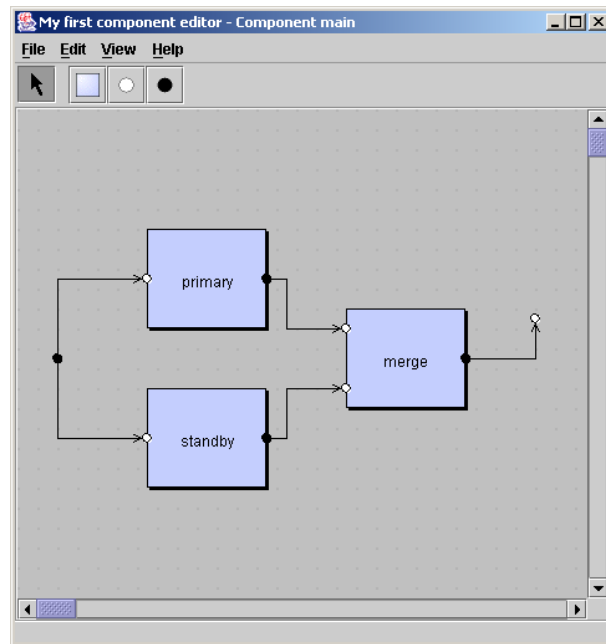


Abbildung 2.2. Beispiel für ein Systemstrukturdiagramm.

ist ein Modellelement in einem Projekt genau einmal enthalten, obwohl es in mehreren Diagrammen durch ein View-Element dargestellt ist.

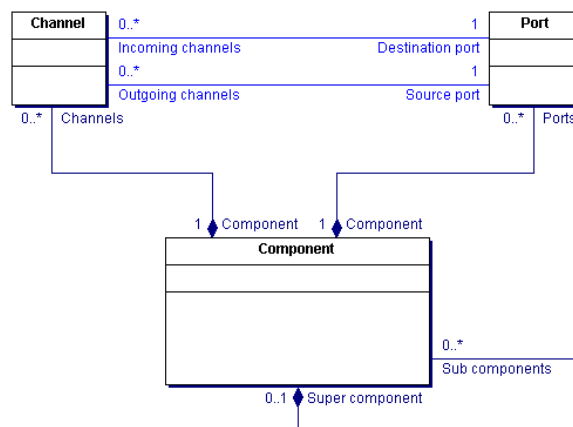


Abbildung 2.3. Ausschnitt der Modellelemente aus dem AUTOFOCUS-Metamodell, die an einem Systemstrukturdiagramm beteiligt sind.

Anhand eines Systemstrukturdiagramms wird im Folgenden der Zusammenhang zwischen Modell- und View-Elementen erläutert. Der Ausschnitt aus dem AUTOFOCUS-Metamodell für ein Systemstrukturdiagramm ist in Abbildung 2.3 dargestellt. Ein Systemstrukturdiagramm referenziert über die beteiligten View-Elemente die Modellelemente Component, Channel und Port. Die Assoziationen im Modell werden in die View-Ebene projiziert. So repräsen-

tiert ein View-Element eine Sicht auf ihr assoziiertes Modell. Die an einem Systemstrukturdiagramm beteiligten Klassen und ihre Relationen sind in in Abbildung 2.4 dargestellt. Das Wur-

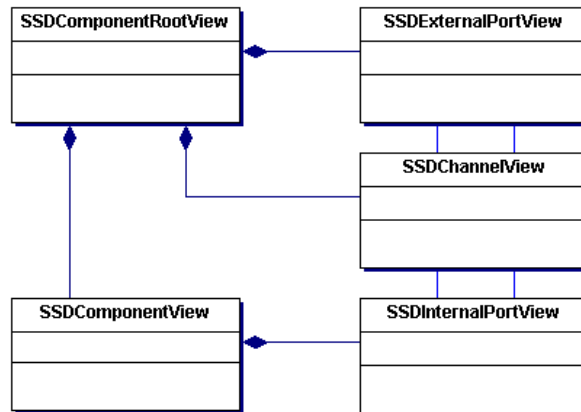


Abbildung 2.4. View-Elemente, die an einem Systemstrukturdiagramm beteiligt sind.

zelobjekt (`SSDComponentRootView`) eines Diagramms aggregiert die View-Elemente seiner Unterkomponenten, Ports und Kanäle. Die View einer Komponente wird durch die Klasse `SSDComponentView` repräsentiert. Das vom Wurzelobjekt assoziierte Modellelement, d.h. eine Komponente, liegt in der hierarchischen Struktur des Modells um eine Ebene höher als die in einem Diagramm dargestellten Komponenten. Bei Komponenten unterscheidet man eine *interne* und eine *externe Sicht*. Bei der äußeren Sicht werden Komponenten als Black-Box-Darstellung betrachtet, die innere Sicht verfeinert in der Glass-Box-Darstellung die Struktur einer Komponente. Ports treten in der externen und internen Sicht einer Komponente auf und werden abhängig von ihrer Semantik in Diagrammen unterschiedlich dargestellt. Ein Port als Modellelement ist gerichtet. Damit wird bestimmt, in welche Richtung Nachrichten gesendet werden können, also in eine Komponente hinein oder aus ihr heraus. Die Darstellung für einen Port in einem Systemstrukturdiagramm ist abhängig von der Sicht auf die Komponente. Von schwarzen Ports kann gelesen werden, auf weiße Ports können hingegen Werte geschrieben werden. Handelt es sich um einen Ausgabeport, kann in der inneren Sicht einer Komponente auf den Port geschrieben werden, weshalb dieser Port in einem Diagramm weiß dargestellt wird. In der äußeren Sicht hingegen wird der Port schwarz angezeigt, weil von ihm gelesen wird. Analog dazu wird ein Eingabeport in der inneren Sicht schwarz und in der äußeren weiß dargestellt. Damit wird ein Port abhängig von der Sicht auf eine Komponente in unterschiedlichen Farben angezeigt. Die innere Sicht auf einen Port wird in einem Systemstrukturdiagramm durch die Klasse `SSDExternalPortView` beschrieben, weil sie aus der Sicht eines Systemstrukturdiagramms einen Teil der externen Schnittstelle der Komponente darstellt. Analog dazu wird die äußere Sicht auf einen Port durch die Klasse `SSDInternalPortView` repräsentiert. Eine Zusammenfassung der unterschiedlichen Varianten und Darstellungsarten von Ports ist in Abbildung 2.5 auf der nächsten Seite dargestellt. Kanäle werden in einem Systemstrukturdiagramm durch die Klasse `SSDChannelView` repräsentiert.

Durch den Begriff *Sicht* wird also bestimmt, ob ein Modellelement aus einer abstrahierten oder verfeinerten Sichtweise betrachtet wird. Im Gegensatz dazu wird mit dem Begriff *View* ein

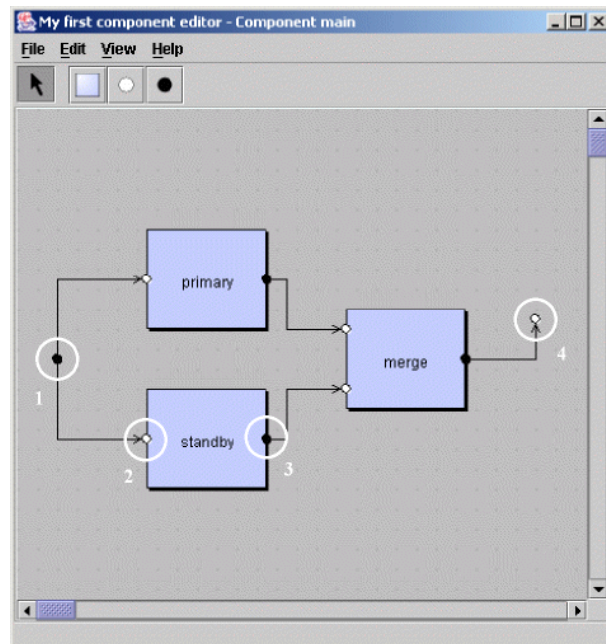


Abbildung 2.5. Die Figur 1 assoziiert ein Objekt vom Typ `SSDExternalPortView`, ist ein Teil der internen Sicht auf eine Komponente und repräsentiert einen Eingabeport. Die Figuren 2 und 3 sind jeweils ein Teil der externen Sicht, assoziieren `SSDInternalPortView` und stellen einen Eingabe- bzw. Ausgabeport dar. Figur 4 repräsentiert einen Ausgabeport, ist ein Teil der internen Sicht und assoziiert `SSDExternalPortView`.

Element bezeichnet, das ein Modellelement in einem Diagramm repräsentiert.

SSD-Diagramme beschreiben die statische Struktur des entworfenen Systems. *STD-Diagramme* beschreiben das dynamische Verhalten einer Komponente und werden durch einen erweiterten endlichen Automaten dargestellt. *EET-Diagramme* dienen zur Beschreibung von beispielhaften Systemabläufen und Interaktionsfolgen. Einer Komponente können dabei beliebig viele solcher Diagramme zugeordnet werden. Da die Spezifikationen durch STD- und EET-Diagramme für diese Arbeit weniger bedeutend sind, wird hier auf eine detaillierte Beschreibung verzichtet und dafür auf [PB02a] verwiesen.

2 Quest

Im Quest-Projekt [Que02] 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 grafischen 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.

2.1 Integration von Werkzeugen

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 [RSW97, McM92] und SATO (*Unbounded Model-Checker*) [Zha97] eingesetzt.
- *Theorembeweiser* unterstützen die Verifikation beliebiger Modelle und Eigenschaften. Verwendet wird das Werkzeug VSE II [UBR⁺99].
- *Testwerkzeuge* testen die entwickelte Software. Für die Auswahl von Testfällen wird CTE [GWG95] 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 [PB02b] realisiert. JAIG ist ausführlich in [Mar98] beschrieben. Der Überblick über die integrierten Werkzeuge und der Zusammenhang mit AUTOFOCUS sind in Abbildung 2.6 auf der nächsten Seite dargestellt.

Durch die Anbindung von AUTOFOCUS an Verifikationswerkzeuge werden formale Methoden in den Entwicklungsprozess [BS99] integriert. [BLSS00] und [BLS02] beschreiben diese Integration ausführlicher.

2.2 Änderungen am Modell

Modellbasierte Anwendungen sind vom zentralen Systemmodell abhängig. Deshalb werden erstere bei Änderungen am Modell benachrichtigt. Diese Änderungen werden durch einen Ereignis-Mechanismus propagiert und können von Klassen vom Typ `ModelChangeListener` verfolgt werden. Die Benachrichtigung der registrierten Objekte erfolgt durch Ereignisse vom Typ `ModelChangeEvent`, welche die Modelländerungen beschreiben. Dabei wird zwischen den folgenden Ereignissen unterschieden:

- Ändern eines Modellattributs
- Hinzufügen eines Modellelements, d.h. Setzen einer Assoziation im Metamodell
- Entfernen eines Modellelements, d.h. Entfernen einer Assoziation im Metamodell
- Löschen eines Modellelements, d.h. Entfernen des Elements aus dem Systemmodell
- Ein Modellelement inkonsistent setzen

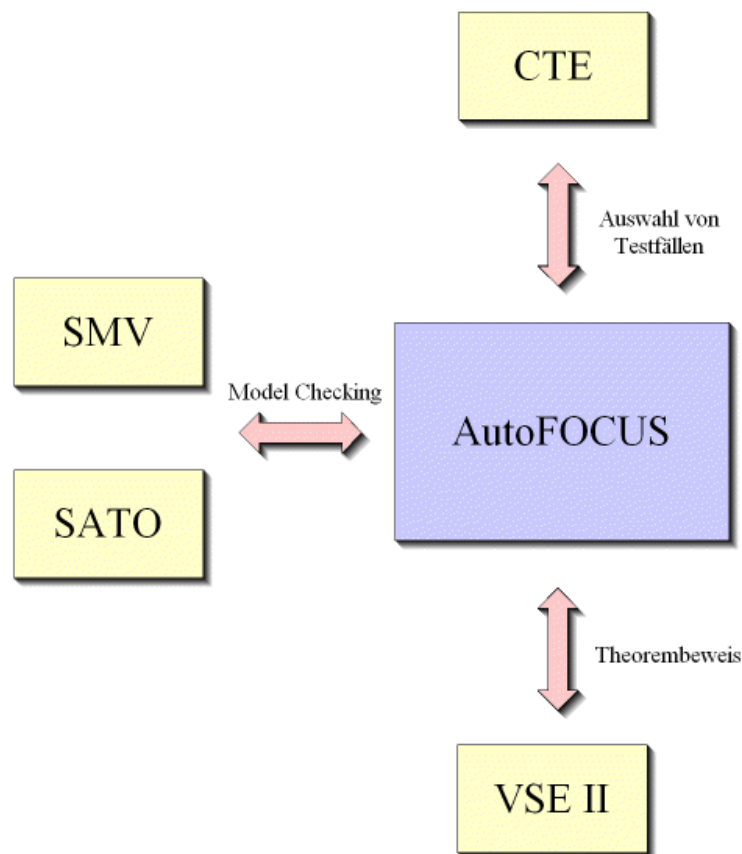


Abbildung 2.6. Integration der Werkzeuge zur Verifikation in AUTOFOCUS. Die Daten werden zwischen den beteiligten Softwaresystemen über das QML-Format ausgetauscht.

- Ein Modellelement konsistent setzen

In Abbildung 2.7 auf der nächsten Seite sind die Bestandteile der Klasse `ModelChangeEvent` beschrieben. Das Attribut `changedAttribute` beschreibt die Art der Änderung in textueller Form, also z.B. ein Modellname geändert oder eine neues Modellelement eingefügt wird. Durch das Attribut `modelElement` wird das Modellelement referenziert, das geändert wird. `targetElement` beschreibt das Element, das entfernt oder eingefügt wird.

Um die Veränderungen am Modell anzuzeigen, werden die Ereignisse in der aktuellen Quest-Implementierung manuell erzeugt und an das entsprechende Modellelement gesendet. Dazu werden auf dem Modellelement die den Veränderungen entsprechenden Methoden aus der Schnittstelle `ChangeableModel` aufgerufen. Die bei einem Modellelement registrierten Objekte werden in einer Warteschlange gesammelt und beim Auftreten von Änderungen der Reihe nach durch einen der Art des Ereignisses entsprechenden Methodenaufruf benachrichtigt. Der Überblick über die Methoden der erwähnten Schnittstellen ist in Abbildung 2.7 auf der nächsten

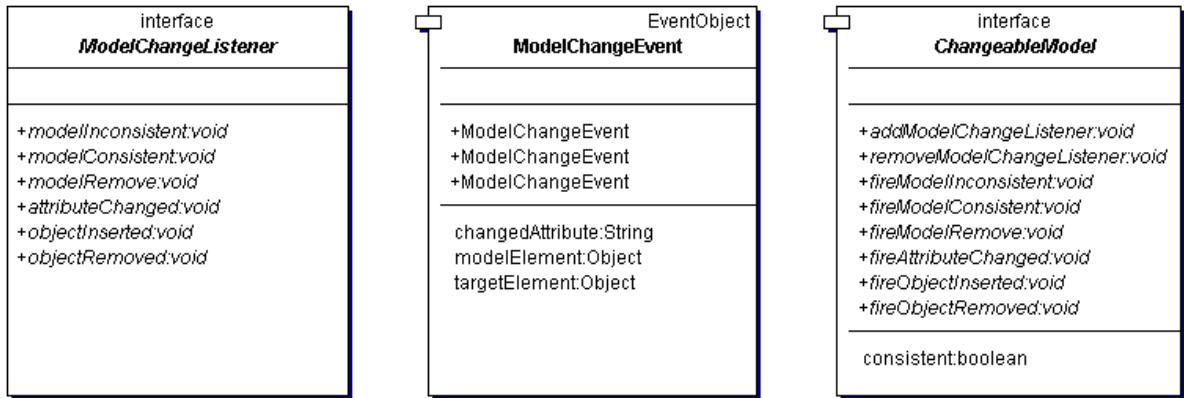


Abbildung 2.7. Objekte vom Typ `ModelChangeListener` können Änderungen bei Modellelementen verfolgen. Die Veränderungen am Modell werden durch Objekte vom Typ `ModelChangeEvent` beschrieben. Die Methoden, die auf einem Modellelement zum Anzeigen von Veränderungen aufgerufen werden, sind in der Schnittstelle `ChangeableModel` deklariert.

Seite dargestellt. Um die Bedeutung der Schnittstellen und ihrer Methoden zu verdeutlichen, wird im Folgenden kurz die Aufrufstruktur bei einer Modelländerung erläutert. Beispielsweise wird beim Einfügen einer neuen Unterkomponente ein Objekt vom Typ `ModelChangeEvent` konstruiert, dessen Attribute die Modelländerung näher beschreiben. Dieses Objekt ist der aktuelle Parameter der Methode `fireObjectInserted()` aus der Schnittstelle `ChangeableModel` beim Aufruf auf dem Modellobjekt der Oberkomponente. Dieses Modellelement besitzt eine Liste aller registrierten Listener vom Typ `ModelChangeListener`. Auf diesen Objekten wird nun jeweils die Methode `objectInserted()` aufgerufen, in der die Änderung des Modells behandelt wird. Eine detailliertere Beschreibung der Verwendung der Ereignisse einschließlich Beispielen befindet sich in [PB02b].

Der beschriebene Vorgang zur Propagierung von Änderungen am Modell entspricht nicht dem in [BCK98] erwähnten Ereignis-System. Deshalb wird dieser Mechanismus in dieser Arbeit lediglich als Ereignis-System bezeichnet. In einem Ereignis-System registriert sich ein Objekt für ausgewählte Ereignisse. In der Realisierung in Quest werden jedoch beim Auftreten von Änderungen, egal welcher Art, alle abhängigen Objekte benachrichtigt. Bei ihnen liegt die Verantwortung, das Ereignis auszuwerten und entsprechend darauf zu reagieren.

2.3 Generierung des Metamodells

Die automatische Generierung des Metamodells setzt die Beschreibung durch Klassendiagramme in Java-Programmtext um. Das Metamodell wird in der Datei `metamodel.dbi` durch das textuelle Format Database Interface Format [Mar98, PB02b] beschrieben. Eine grafische Notation in einem CASE-Werkzeug zur UML-Modellierung ist der Ausgangspunkt für die Erzeugung der dbi-Datei durch Skripte. Die konkreten Modell- und View-Klassen einschließlich ihrer Zugriffsstrukturen, Navigationsmethoden, Assoziationen und Attribute werden aus dieser Beschreibung vom Java-Programmtext-Generator *MMGen* (Metamodell-Generator) [PB02b] generiert. *MMGen* basiert auf Mustern für den Klassenkopf und die Implementierung von Asso-

ziationen. Diese Muster können für mehrere Klassen verwendet werden und enthalten Variablen, die in der Generierung durch konkrete Werte ersetzt werden. Bei der Generierung werden auch benutzerdefinierte Methoden berücksichtigt.

Die abstrakte Notation des Metamodells wird in die Implementierung übersetzt. Bei Änderungen an der Struktur des Metamodells vereinfacht sich die erforderliche Anpassung in der textuellen Beschreibung gegenüber dem Ansatz ohne Generierung. Die Änderungen werden automatisch in die erneut generierten Klassen übernommen. Aus einer fehlerfreien dbi-Datei können fehlerfreie Klassen generiert werden. Somit ist die Generierung weniger fehleranfällig als die manuelle Implementierung. Durch die Verwendung einer textuellen Beschreibungssprache besteht keine Abhängigkeit von der verwendeten Zielsprache. Durch die Anpassung des Generators kann das Metamodell in die gewünschte Sprache übersetzt werden.

2.4 Zugriffsstrukturen

Für die Kommunikation zwischen den Objekten des Modells und denen der Views implementieren Modellelemente die Schnittstelle `ModelElement` und die View-Elemente die Schnittstelle `ViewElement`. Wurzelobjekte bieten zusätzlich über die Schnittstelle `RootViewElement` weitere Zugriffsmethoden für die Navigation zwischen View- und Modellelementen an. Der Überblick über die bereitgestellten Methoden ist in Abbildung 2.8 dargestellt. Die Schnittstellen

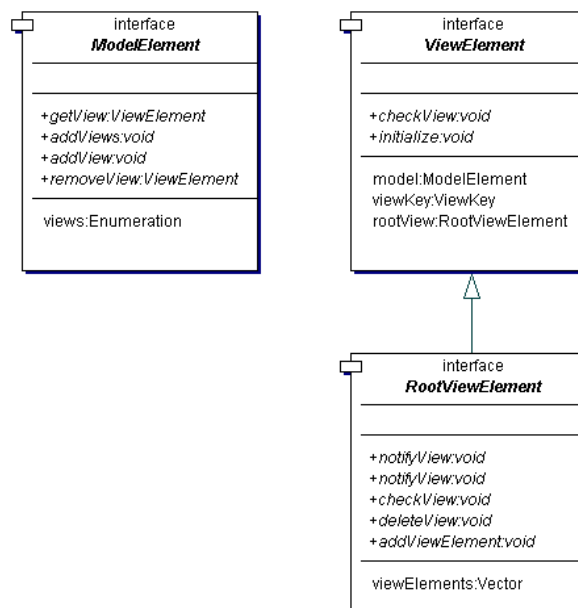


Abbildung 2.8. Schnittstellen für die Kommunikation zwischen Objekten des Modells und denen der View-Ebene.

für Modellelemente (`ModelElement`) und View-Elemente (`ViewElement`) bieten Methoden für den Zugriff auf die Ebene der Views bzw. des Modells. Das Wurzelobjekt eines Diagramms implementiert die Schnittstelle `RootViewElement` und stellt damit einen Container für alle im Diagramm enthaltenen View-Elemente dar. Die Zuweisung eines View-Elements zu einem

bestimmten Diagramm erfolgt durch die Attributierung mit einem eindeutigen *Schlüssel*. Damit werden alle View-Elemente eines Diagramms einschließlich des Wurzelobjektes durch den gleichen Wert eines Schlüssels identifiziert. Für den Zugriff auf Modellattribute über ein View-Element werden Methoden generiert, die Aufrufe auf View-Objekten an die entsprechenden Methoden des Modells delegieren. Dazu werden diese Aufrufe über die assoziierten Modellelemente in die Ebene der Modelle weitergeleitet und der Rückgabewert über das View-Objekt dem Aufrufer der Methode bereitgestellt. [Löt02] beschreibt die Zugriffsstrukturen und ihre Implementierungen im Detail.

Prinzipien im Softwareentwurf

Software zeichnet sich nicht nur im Umfang der Funktionalität, sondern vielmehr durch Qualitätsmerkmale aus. Abhängig vom Anwendungsgebiet treten gewisse Eigenschaften qualitativ hochwertiger Software besonders in der Vordergrund. Allgemein ist die Qualität der Softwareentwicklung und ihres Entwicklungsprozesses nach [SE] durch eine Reihe von Erfolgsfaktoren bestimmt:

- *Dokumentation*: Die Dokumentation in ihren verschiedenen Ausprägungen wird idealerweise parallel zum Entwicklungsprozess erstellt.
- *Modularisierung*: Das Softwaresystem wird in überschaubare Einheiten strukturiert. Dies führt zur Reduktion der Komplexität des Softwaresystems.
- *Datenabstraktion*: Soweit Details nicht von Bedeutung für das Softwaresystem sind, bleiben sie unberücksichtigt.
- *Verkapselung*: Details und Datenzustände werden beispielsweise in Klassen eingekapselt und sind nur durch Zugriffsstrukturen der Schnittstelle veränderbar oder lesbar.
- *Schnittstellen*: Der Zugriff auf Daten und Dienste erfolgt nach festgelegten Regeln.
- *Entwicklungs- und Programmierstil*: Der Programmtext zeichnet sich durch Einfachheit, Klarheit, Lesbarkeit, Änderbarkeit und Effizienz aus.
- *Wiederverwendbarkeit*: Der wiederholte Einsatz von Ergebnissen in anderen Projekten wird angestrebt. Als Folge werden Kosten reduziert und eine zufriedenstellende Qualität durch die Verwendung von erprobten Referenzlösungen erreicht.
- *Flexibilität und Änderbarkeit*: Die Software wird derart entwickelt, dass sie an zukünftige Änderungen an den Anforderungen leicht anzupassen ist. Dabei sind auch nicht vorhersehbare Entwicklungen zu berücksichtigen.
- *Benutzerschnittstellen*: Die Bedienung des Softwaresystems erfolgt in einer für den Benutzer einfachen und verständlichen Art und Weise. Dazu werden bekannte Muster der Mensch-Maschine-Schnittstelle eingesetzt. Die Klarheit der angebotenen Dienste und die Benutzerführung bzw. die Ergonomie der Software werden berücksichtigt.
- *Robustheit*: Ein robustes Softwaresystem zeichnet sich durch Fehlertoleranz aus.
- *Effizienz*: Die vorhandenen Softwarr- und Hardwareressourcen werden sinnvoll ausgenutzt.

- *Zugriffssicherheit und Datenschutz*: Die Daten des Softwaresystems sind vor dem unerlaubten Zugriff durch Dritte geschützt.

1 Ziele in der Softwareentwicklung

Ziele in der Softwareentwicklung werden in die Bereiche funktionale, nicht funktionale und technische Qualitätsmerkmale und Rahmenbedingungen eingeteilt. Diese Merkmalsgruppen werden im Folgenden nach [POM] detailliert beschrieben:

Funktionale Qualitätsmerkmale beziehen sich auf die Funktionalität der Software: Die Erfüllung der Anforderungen durch die Implementierung gibt an, in welchem Umfang die Anforderungen in der Implementierung realisiert sind. Die erarbeiteten Anforderungen sind der Problemstellung angemessen, d.h. Sie berücksichtigen die Funktionalität, die für die Lösung der gestellten Probleme erforderlich ist.

Zu den nicht *funktionalen Qualitätsmerkmalen* zählen die folgenden Eigenschaften:

- Zuverlässigkeit, Verfügbarkeit, Robustheit
- Testbarkeit
- Wartbarkeit
- Effizienz, Performanz
- Verständlichkeit, Benutzerfreundlichkeit (gute Mensch-Maschine-Schnittstelle)
- Anpassbarkeit (Wandlungsfähigkeit), Änderbarkeit, Portabilität, Weiterentwicklungs- und Wiederverwendungsfähigkeit
- Fehlertoleranz
- Einfacher und kostengünstiger Betrieb
- Adäquatheit des Umfangs der Funktionalität
- Korrektheit
- Nutzbarkeit: Die Software entspricht den Anforderungen der Nutzung bzw. des Betriebes

Die *technischen Qualitätsmerkmale* umfassen folgende Eigenschaften:

- Dokumentation
- Modularisierung, Strukturierung, transparente Architektur, einfache Ablaufstruktur
- Einfache, klar verständliche Schnittstellen, welche die Implementierung der Komponenten verbergen und die Dienste für andere Komponenten anbieten
- Funktionale Zusammengehörigkeit der Funktionalität innerhalb einer Komponente

- Verwendung von Programmier-Standards

Rahmenbedingungen grenzen die Umgebung, in der die Entwicklung realisiert wird, weiter ein. Dabei werden die beiden Gebiete unterschieden, die sich auf den Entwicklungsprozess und die technischen Voraussetzungen beziehen. Der Entwicklungsprozess wird durch Vorgaben für die Bereiche Qualität, Termine und Kosten bestimmt. Die vorgegebene Hard- und Softwarestruktur ist Teil der Umgebung, in die das neue Softwaresystem integriert wird.

Die Qualität von Software zeichnet sich u.a. durch die Wiederverwendung von Ergebnissen anderer Projekte aus. Die Verwendung existierender Software reduziert den Erweiterungs- und Wartungsaufwand des zu entwickelnden Softwaresystems. Das Zurückgreifen auf Funktionalität, die bereits in Rahmenwerken, Klassenbibliotheken oder Komponentenimplementiert ist, senkt den Aufwand für die Realisierung weiterer Software. Rahmenwerke sind ein wichtiger Vertreter der Ansätze, welche die Wiederverwendung unterstützen. Deshalb werden ihre Eigenschaften im folgenden Abschnitt detailliert betrachtet.

2 Rahmenwerke

Die objektorientierte Programmierung [Mey00] erlaubt die Einführung von *Rahmenwerken* (engl. *Framework*). Nach [PR97] ist ein Rahmenwerk wie folgt definiert:

Rahmenwerke sind Organisationsformen von Sammlungen objektorientierter Komponenten. Ein Rahmenwerk ist [...] eine Konfiguration von Klassen, die ein Lösungsschema für eine Problemstellung vergegenständlichen. Ein Anwendungsrahmenwerk (engl. *Application Framework*) ist eine spezielle Form eines Rahmenwerks, das die Grundstruktur und den Kontrollfluss einer vollständigen Anwendung enthält. Es liefert eine generische Lösung für eine Klasse von Anwendungsprogrammen; eine Lösung, die aber in den meisten Fällen noch anwendungsspezifisch durch Ableitung spezialisiert oder durch Parametrisierung ergänzt werden muss. Darüber hinaus bieten Anwendungsrahmenwerke oft eine einheitliche Benutzungsschnittstelle sowohl in Form der Präsentation als auch der Handhabung (engl. *Look and Feel*). Verglichen mit Klassenbibliotheken kommt hier der Gesichtspunkt hinzu, dass die anwendungsspezifischen Komponenten mit passenden Schnittstellen in ein Anwendungsrahmenwerk eingehängt werden. Der Kontrollfluss der Anwendung ist insgesamt bereits durch das Anwendungsrahmenwerk definiert und wird anwendungsspezifisch nur noch angepasst.

Bei einem Rahmenwerk handelt es sich also um eine halbfertige und wiederverwendbare Anwendung, die für einen bestimmten Einsatz spezialisiert wird [RJ88]. Im Gegensatz zu Klassenbibliotheken zielen Rahmenwerke auf bestimmte Geschäftsbereiche oder Anwendungsdomänen als Einsatzfeld ab. Nach [MF] ergeben sich aus der Anwendung von Rahmenwerken folgende Vorteile:

- *Modularität*: Rahmenwerke unterstützen die Modularität, indem sie Implementierungsdetails durch Schnittstellen verbergen. Dadurch werden Veränderungen im Design oder in der Implementierung lokalisiert, wodurch der Aufwand für die Wartung und die Verständlichkeit reduziert wird.

- *Wiederverwendung*: Durch Rahmenwerke werden generische Komponenten angeboten, die für die Entwicklung von Anwendungen wiederverwendet werden können. Damit müssen bei der Entwicklung von Anwendungen wiederkehrende Aufgaben nicht wiederholt gelöst werden. Mit der Anwendung von qualitativ hochwertigen Rahmenwerken werden die Entwicklungskosten reduziert.
- *Erweiterbarkeit*: Rahmenwerke verbessern die Erweiterbarkeit von Softwaresystemen, indem sie Einhängpunkte (*Hooks*) für zusätzliche Funktionalität anbieten [Pre94], durch welche die Schnittstellen erweitert werden. Dadurch werden Anwendungen für spezielle Ausprägungen angepasst.
- *Umkehrung der Programmsteuerung*: Treten Ereignisse auf, werden die Methoden der Einhängpunkte auf den vorher registrierten Objekten durch die Ereignisbehandlung des Rahmenwerks aufgerufen. Die Objekte bearbeiten die Ereignisse entsprechend den Vorgaben durch die Anwendung. Damit entscheidet das Rahmenwerk und nicht die Anwendung, welche domänenspezifischen Methoden beim Auftreten von externen Ereignissen aufgerufen werden.

Der Einsatz von Rahmenwerken verbessert die Qualitätsmerkmale und reduziert die Entwicklungszeit von Software. Dennoch resultiert der Einsatz von Rahmenwerken in einigen Nachteilen. Rahmenwerke erlauben eine geringere Wiederverwendung und Wartbarkeit als Softwarekomponenten. Die Kopplung von Anwendung und Rahmenwerk durch Implementierungsvererbung und spezielle Rahmenwerk-Schnittstellen, die von der Anwendung implementiert werden müssen, ist stark ausgeprägt. Ändert sich die Implementierung der Anwendung oder des Rahmenwerks, resultiert dies häufig in umfangreichen Änderungen des anderen Teils.

Evaluierung

Dieses Kapitel beschreibt die Evaluierung ausgesuchter Softwaresysteme für die Verwendung als Basis für einen Quest-Editor. Der Ausgangspunkt in Abschnitt 1.1 ist die Darstellung der Anforderungen an ein Editoren-Rahmenwerk. Dabei werden sowohl allgemeine Anforderungen als auch solche betrachtet, die sich speziell an Quest orientieren. Die Anforderungen sind aus den in Kapitel 3 auf Seite 17 beschriebenen Prinzipien und Zielen im Softwareentwurf abgeleitet. Im Abschnitt 2 auf Seite 24 wird das Bewertungssystem vorgestellt. Dabei wird zwischen den beiden Arten von Kriterien unterschieden, die zu einem Ausschluss der Software führen und eine detailliertere Evaluierung beschreiben. Ist eine Eigenschaft aus den Ausschlusskriterien nicht erfüllt, wird die Software nicht weiter evaluiert.

1 Anforderungen an ein Editoren-Werkzeug

1.1 Allgemeine Anforderungen

Für die Evaluierung der Softwaresysteme werden die Anforderungen nach ihrer Wichtigkeit für den Einsatz im Editorenkonzept sortiert. Anforderungen mit hoher Priorität werden vor denen mit niedrigerer Wichtigkeitsklasse beschrieben.

1.1.1 Weiterentwicklung

Die *Weiterentwicklung* für das gewählte Werkzeug ist gewährleistet. Dabei liegt der Schwerpunkt weniger auf der Erweiterung der gebotenen Funktionalität, sondern mehr auf der Anpassung an neue Plattformen. Beispielsweise wird bei der Veröffentlichung einer neuen Java-Version das Softwaresystem aktualisiert, um die neuen technischen Erweiterungen zu integrieren.

1.1.2 Entwicklungsstadium und Stabilität

Die Entwicklung ist so fortgeschritten, dass die Arbeit mit dem erstellten Editor in einem für dieses Anwendungsgebiet akzeptablen Bereich erfolgt. Für die Entwicklung einer Anwendung, die auf der Basis eines Rahmenwerks realisiert wird, ist es wichtig, eine entsprechende *Stabilität* zu fordern, um die geforderte Stabilität der Zielanwendung zu realisieren.

1.1.3 Komplexität

Der vom Rahmenwerk angebotene Umfang der Funktionalität ist überschaubar, um die Einarbeitungszeit gering zu halten. Eine Maßzahl für den Umfang ist die Anzahl der Klassen des

Rahmenwerks. Die Funktionalität ermöglicht außerdem die Realisierung der durch die Zielanwendung vorgegebenen Anforderungen.

1.1.4 Einarbeitung

Die *Einarbeitung* wird durch verschiedene Ausprägungen der Dokumentation unterstützt. Die Dokumentation umfasst dabei folgende Aspekte:

- Ein Tutorial oder Beispiele für die *Entwicklung* der Anwendung sind vorhanden.
- Die *Architektur* des Softwaresystems und die Kommunikation zwischen den einzelnen Teilen werden erläutert.
- Die *Verwendung* der Komponenten der Anwendung wird erklärt.
- Klassen, Methoden und Attribute werden durch Dokumentation in Form von *Javadoc-Kommentaren* beschrieben.

Das Werkzeug erleichtert die Einarbeitung durch seine *Strukturierung*, welche die Aufteilung des Rahmenwerkes beispielsweise nach dem Aufgabenbereich der Komponenten umfasst. Die *Einarbeitungszeit* übersteigt nicht den Zeitraum von zwei Wochen. Der Benutzer ist dann mit der Anwendung vertraut, wenn eine Anwendung gemäß den Anforderungen erstellt werden kann.

1.1.5 Lizenz

Die Verwendung des Softwaresystems ist für akademische, nicht kommerzielle Zwecke frei. Idealerweise ist auch eine spätere kommerzielle Nutzung möglich.

1.1.6 Benutzerschnittstelle

Die Mensch-Maschine-Schnittstelle zeichnet sich durch eine intuitive *Bedienung* aus und entspricht in seiner *Präsentation* dem heutigen Stand. Die *Funktionen* des Editors umfassen folgende Bereiche:

- Kopieren, Einfügen, Ausschneiden, Löschen
- Öffnen, Speichern
- Undo, Redo
- Verschieben und die Größenänderung von grafischen Objekten
- Funktionalität für die *Navigation* in einer umfangreichen Zeichnung

1.1.7 Verfügbarkeit des Programmtextes

Der *Quelltext* des Softwaresystems ist verfügbar, um die Anbindung der Anwendung an Quest zu unterstützen und die Einarbeitung zu vereinfachen. Diese Anforderung bezieht sich v.a. auf Rahmenwerke.

1.2 An Quest orientierte Anforderungen

1.2.1 Kopplung zwischen Quest und Editor

Die *Anbindung von Quest* an den Editor wird unterstützt. Dazu ist ein Mechanismus für den Datenaustausch realisiert, der idealerweise nicht auf Dateien beruht, sondern modellbasiert ist. Das Editoren-Werkzeug implementiert das *MVC-Konzept*. Außerdem werden *Ereignis-Mechanismen* unterstützt: Sind beispielsweise zwei Editoren geöffnet, von denen einer ein AUTOFOCUS-Sequenzdiagramm bearbeitet, wird der andere Editor, in dem ein AUTOFOCUS-Systemstrukturdiagramm geöffnet ist, über die Änderung am Modell benachrichtigt, um die Anzeige entsprechend zu aktualisieren.

1.2.2 Plattformen

Um die Plattformunabhängigkeit von Quest bei der Erweiterung durch das Editorenkonzept zu erhalten, ist das Editoren-Werkzeug auf folgenden *Plattformen* verfügbar: Windows, Solaris, UNIX, Linux und Macintosh. Als Implementierungssprache ist *Java* am besten geeignet, weil Quest ebenso in dieser Sprache realisiert ist.

1.2.3 Grafische Objekte

Im implementierten Editor werden nicht nur Graphen, sondern auch andere Diagrammartentypen visualisiert. Für den Einsatz in Quest ist die Darstellung von Graphen für Systemstrukturdiagramme vorausgesetzt. In Zustandsübergangsdigrammen sind Graphen mit Schlingen und Mehrfachkanten nötig. Das Editoren-Werkzeug unterstützt auch die Darstellung von Sequenzdiagrammen. Die verschiedenen Diagrammartentypen sind in Abschnitt 1.2 auf Seite 8 beschrieben. Neben den vorgegebenen grafischen Objekten für die Anzeige im Editorfenster können auch eigene Figuren definiert werden. Diese Funktionalität ist im Hinblick auf die eventuelle Erweiterung des AUTOFOCUS-Metamodells wichtig, die durch die Fähigkeiten des Editors nicht beschränkt werden darf.

1.2.4 Erweiterungs- und Anpassungsfähigkeit

Der Editor folgt den Erweiterungen und Änderungen des AUTOFOCUS-Metamodells. Wenn z.B. ein neues Element wie Tasks eingeführt wird, sind die Auswirkungen auf die gesamte Anwendung und den Editor gering. Der Aufwand für die Anpassung an die neuen Anforderungen ist gering.

1.2.5 Integration zusätzlicher Funktionalität

Das Rahmenwerk ermöglicht die Integration *benutzerdefinierter Funktionen*, welche die übliche Editorfunktionalität erweitern. Beispiele sind die Integration eines Hilfesystems oder einer Benutzerführung, welche beim Anlegen eines AUTOFOCUS-Systemstrukturdiagramms automatisch ein beispielhaftes Diagramm anzeigt, um den Benutzer bei der Konstruktion von Diagrammen zu unterstützen.

Die Integration von *Folgen von Elementaroperationen* wird unterstützt. Beispielsweise wird der Benutzer beim Einfügen einer neuen Komponente zwischen zwei bereits existierenden

Komponenten bei ihrer Erstellung angeleitet. Dazu werden die Angaben über den neuen Komponentennamen abgefragt und eine Auswahl der möglichen Verbindung von Kanälen und Ports angezeigt.

1.2.6 Integrierte Benutzerführung

Die Ausführung von bestimmten Aktionen, die den Syntaxregeln widersprechen, wird vom Editor verhindert. Dazu ist die Unterstützung der Interaktion mit dem AUTOFOCUS-Modell nötig, das Einschränkungen und Konsistenzbedingungen beschreibt.

1.2.7 Mehrbenutzerfähigkeit

Das Rahmenwerk bietet die Unterstützung für mehrere Benutzer. Dazu werden Sperrmechanismen realisiert, welche die gleichzeitige Bearbeitung von Modellelementen durch mehrere Benutzer verhindern.

1.2.8 Layout-Algorithmen

Layout-Algorithmen platzieren die Diagrammelemente auf der Zeichenfläche automatisch. Dabei wird in einem AUTOFOCUS-Systemstrukturdiagramm der Verlauf der Kanten angepasst, wenn eine Komponente verschoben wird, um die Kreuzung mit anderen grafischen Elementen zu vermeiden. Layout-Algorithmen können durch zwei verschiedene Ansätze durch das Editoren-Werkzeug unterstützt werden:

- *Einbettung* von externen *Layout-Algorithmen*: Das Editoren-Werkzeug erlaubt die Integration von weitem Layout-Verfahren. Dazu bietet es Einhängpunkte für diese Funktionalität.
- *Integrierte Layout-Algorithmen*: Das Werkzeug bietet eine integrierte Bibliothek von Layout-Verfahren.

2 Bewertungssystem

Die Evaluierung der einzelnen Softwaresysteme basiert auf der Vergabe von Punkten, welche den Grad der Erfüllung einer Anforderung ausdrücken. Dabei werden folgende Abstufungen unterschieden:

Note	Anmerkung
5	Der bewertete Aspekt ist vollständig erfüllt.
4	Die Eigenschaft ist vorhanden, wird jedoch nicht in ihrer Ganzheit erfüllt.
3	In der gewünschten Eigenschaft liegen deutliche Einschränkungen vor.
2	Die Eigenschaft ist nicht vorhanden, kann aber nachträglich realisiert werden.

Note	Anmerkung
1	Der Aspekt ist nicht implementiert und erfordert in der Realisierung einen hohen Aufwand.

Tabelle 4.1. Überblick über das Bewertungssystem

Die einzelnen Bewertungen eines Softwaresystems werden addiert und durch die Anzahl der betrachteten Aspekte dividiert. Falls nötig, werden die errechneten Werte nach den mathematischen Regeln gerundet.

In den folgenden Abschnitten ist die differenzierte Bewertung der jeweiligen Softwaresysteme nach dem oben dargestellten System beschrieben. Dabei werden die erarbeiteten Kriterien in zwei Klassen eingeteilt. Softwaresysteme, die ein Ausschlusskriterium nicht erfüllen, scheidern aus der weiteren Evaluierung aus. Werden alle diese Kriterien erfüllt, erfolgt eine detailliertere Bewertung durch die erweiterten Kriterien.

2.1 Ausschlusskriterien

Ein Softwaresystem kann im Rahmen dieser Arbeit nicht eingesetzt werden, falls eines der folgenden Kriterien nicht erfüllt wird. Diese sind ausführlich in Abschnitt 1 auf Seite 21 erläutert.

- Einarbeitung
- Lizenzkosten
- Stabilität
- Grafische Objekte
- Präsentation
- Plattformen
- Anbindung an Quest
- Weiterentwicklung

2.2 Erweiterte Kriterien

Die Softwaresysteme werden nach den Eigenschaften der erweiterten Kriterien untersucht, falls alle Merkmale nach den Ausschlusskriterien erfüllt sind. Unterschieden werden folgende Eigenschaften:

- Komplexität
- Benutzerschnittstelle: Intuitive Bedienung, angemessene Funktionalität, Navigation
- Ankopplung an Quest: MVC-Konzept und Ereignis-Mechanismen

- Erweiterungs- und Anpassungsfähigkeit
- Integration zusätzlicher Funktionalität
- Integrierte Benutzerführung
- Mehrbenutzerfähigkeit
- Layout-Algorithmen
- Verfügbarkeit des Quelltexts

In den folgenden Abschnitten werden die evaluierten Softwaresysteme kurz beschrieben und die Ergebnisse der Evaluierung dargestellt.

3 Evaluierte Software

Für die Evaluierung werden in dieser Arbeit mehrere Kategorien von Softwaresystemen betrachtet. Mit *Diagramm-Zeichenwerkzeugen* werden Diagramme erstellt. Die Anbindung an Quest erfolgt durch die Bereitstellung einer Schnittstelle zwischen diesen beiden Anwendungen. *Meta-CASE-Werkzeuge* erlauben die Spezifikation einer eigenen Modellierungssprache und die Generierung der Werkzeuge für den Entwicklungsprozess. *Vollständige grafische Editoren* sind bereits implementierte Programme, die an die anwendungsspezifischen Anforderungen angepasst werden können. Die dritte Klasse der bewerteten Softwaresysteme sind *Editoren-Rahmenwerke*. *Visualisierungswerkzeuge für Graphen* dienen ausschließlich zur Anzeige von Graphen. Die Editierfunktionalität ist dabei nicht implementiert. *Graphen-Editoren* realisieren dedizierte Editoren für die Visualisierung und Modifikation von Graphen. *Benutzerdefinierbare Entwicklungsumgebungen* ermöglichen dem Entwickler, eine Umgebung für die Erstellung von Software an die eigenen Anforderungen anzupassen.

In den folgenden Abschnitten werden Vertreter der beschriebenen Kategorien vorgestellt und die Entscheidung für oder gegen einen Einsatz als Quest-Editor begründet. Die Bewertung der in Abschnitt 1 auf Seite 21 erarbeiteten Anforderungsmerkmale der Softwaresysteme ist in Anhang A auf Seite 111 zusammengefasst.

3.1 Diagramm-Zeichenwerkzeuge

Durch Diagramm-Zeichenwerkzeuge werden aus grafischen Symbolen Diagramme konstruiert. Für den Einsatz als Basis für einen Quest-Editor ist eine Anbindung von externen Programmen erforderlich, um die Verbindung zu Quest zu ermöglichen.

Das Zeichenwerkzeug *Microsoft Visio* [Vis02] ermöglicht dem Benutzer das Erzeugen beliebiger, benutzerdefinierter grafischer Symbole. Die für Quest nötigen Diagrammbestandteile (siehe Abschnitt 1.2 auf Seite 8) werden in einem ersten Schritt entworfen. Um einen Bezug zwischen den grafischen Objekten und zugrundeliegenden Daten herzustellen, wird das AUTO-FOCUS-Modell in der Sprache Microsoft Visual Basic for Applications (VBA) implementiert und seine Bestandteile mit den grafischen Figuren assoziiert. Veränderungen am Modell werden an der Benutzerschnittstelle angezeigt. Umgekehrt werden die Änderungen an den grafischen

Objekten in das Modell propagiert. Die Realisierung von Editoren für Quest mit dem Werkzeug Visio ist aus technischer Sicht möglich. Jedoch erfordert die Implementierung eine Umsetzung des AUTOFOCUS-Modells in die Sprache VBA. Da aber in Quest auch noch weitere Werkzeuge integriert sind, würde Quest in die beiden Teile Editoren und andere Werkzeuge zerfallen. Dieser Ansatz widerspricht dem Ziel dieser Arbeit, in der ein Editorenkonzept entwickelt wird, das auf einem zentralen Systemmodell basiert. Aus diesem Grund wird Visio nicht weiter bewertet und scheidet damit als Basis für einen Quest-Editor aus.

3.2 Meta-CASE-Werkzeuge

MetaEdit+ [Met02] ist ein Werkzeug für die Systementwicklung, das die Spezifikation einer benutzerdefinierten Modellierungsmethode erlaubt. Es verfolgt den Ansatz eines Meta-CASE-Werkzeugs, mit dessen Hilfe eine Modellierungssprache spezifiziert und das eigentliche Werkzeug einschließlich seiner Hilfsprogramme generiert wird. Herkömmliche Werkzeuge unterstützen meist nur eine festgelegte Modellierungssprache, wie z.B. UML [RJB97]. Um die Entwicklungsmethode an die Anwendungsdomäne und die Bedürfnisse der Benutzer anzupassen, werden deshalb zunehmend Meta-CASE-Werkzeuge eingesetzt. In Abbildung 4.1 sind die Schritte von der Spezifikation zum Entwicklungswerkzeug dargestellt. Klassische Entwicklungswerkzeuge basieren auf einer vorgegebenen Modellierungsmethode. Der Ansatz von Meta-CASE-Werkzeugen bietet die Spezifikation der eigentlichen Modellierungssprache durch eine Metasprache. Aus der Spezifikation der Sprache wird das Werkzeug generiert.

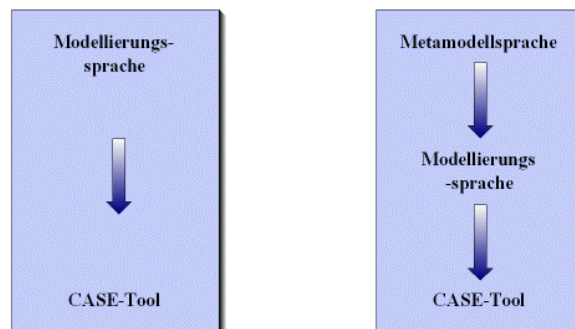


Abbildung 4.1. Klassische Entwicklungswerkzeuge basieren auf einer vorgegebenen Modellierungssprache. Meta-CASE-Werkzeuge hingegen generieren aus der Spezifikation der gewünschten Beschreibungssprache das eigentliche Werkzeug.

Meta-CASE-Werkzeuge bieten eine Reihe von Vorteilen. Sie realisieren die Unterstützung von domänenspezifischen Komponentenmodellen. Im Gegensatz zu Modellen, die sich ausschließlich am Programmtext orientieren und z.B. in UML verwendet werden, werden die Modelle mit der Domäne assoziiert, in der sie eingesetzt werden. UML hingegen bezieht sich nicht direkt auf die Anwendungsdomäne, sondern nur auf die Implementierung, d.h. diese Modellierungssprache visualisiert den Programmtext. Domänenspezifische Modelle bestehen aus Elementen, die Gegenstände aus der Welt der betreffenden Domäne und nicht aus der Implementierung repräsentieren. Damit werden die relevanten Aspekte des zu entwickelnden Produkts

erfasst und die vertraute Terminologie der Domäne verwendet.

Die Spezifikation der Modellierungssprache erfolgt durch grafische und textuelle Notationen. Dabei werden zunächst die Basiskomponenten der Anwendung (z.B. Zustände, Komponenten) spezifiziert und ihre Eigenschaften wie z.B. Name oder Datentyp festgelegt. Es folgt die Bestimmung der Beziehungen wie beispielsweise Zustandsübergänge zwischen den einzelnen Objekten. Das eigentliche Entwicklungswerkzeug wird aus dieser Spezifikation generiert. Neben dem grafischen Entwicklungswerkzeug werden weitere Hilfsprogramme zur automatischen Dokumentation, Erzeugung von Programmtext, Analyse des erstellten Systems und zur Überprüfung der definierten Konsistenzbedingungen generiert. Das modellierte System kann in benutzerdefinierten Formaten exportiert werden. Das Werkzeug arbeitet auf einem zentralen Datenspeicher, auf den mehrere Benutzer zugreifen können.

Die Diagrammbestandteile für Quest (siehe Abschnitt 1.2 auf Seite 8) und die Konsistenzbedingungen können mit dem Werkzeug spezifiziert werden. Das gesamte AUTOFOCUS-Metamodel kann durch die Spezifikation der Modellierungssprache in dieses Werkzeug übertragen werden. Die Anbindung an andere Programme erfolgt durch Exportieren in benutzerdefiniertem Format. Weil die Verbindung zu Quest auf Dateien beruht und nicht modellbasiert ist, wird MetaEdit+ als Basis für einen Quest-Editor nicht eingesetzt. Denn das Ziel ist die Ankopplung an das zentrale AUTOFOCUS-Systemmodell.

3.3 Fertige Editoren

3.3.1 Gift

Gift ist ein fertiger Editor zur Realisierung von benutzerdefinierten Editoren für beliebige grafische Notationen. Das Programm ist in Java erstellt und wird durch Programmierung und die Verwendung des Editors angepasst. Durch die Erstellung von benutzerdefinierten Paletten wird der Editor an die Anwendungsdomäne angepasst. Eine Palette ist eine Sammlung von grafischen Objekten, die durch den Editor erstellt werden. Die Speicherung von erstellten Diagrammen erfolgt durch das Format VRML oder benutzerdefiniert.

Die Anbindung an Quest erfolgt durch die Abspeicherung von Diagrammen in Form von Dateien. Der Austausch von Daten des zentralen AUTOFOCUS-Systemmodells wird in Form von Dateien realisiert. Diesem Ansatz fehlt aber die Modellbasierung, die für das in dieser Arbeit vorgestellte Editorenkonzept vorausgesetzt wird. Aus diesem Grund wird Gift als Basis für einen Quest-Editor nicht eingesetzt.

3.3.2 ILOG JViews Component Suite

Die Software umfasst Java-Komponenten zur Erstellung von Web-basierten Benutzerschnittstellen. Die Bildelemente sind vollständig an die Anwendungsdomäne anpassbar und werden durch Layout-Verfahren auf der Zeichenfläche angeordnet. Eine Integration in eine bestehende Anwendung ist möglich. Obwohl der eigentliche Einsatzzweck der Komponenten die Visualisierung von Web-basierten Benutzerschnittstellen ist, können die Design- und Editierwerkzeuge für die Erstellung der Figuren auch als grafische Modellierungswerkzeuge verwendet werden. Diese können durch die Veränderung des Quelltextes an die gewünschten Anforderungen angepasst werden. Die folgende Funktionalität wird durch die Verwendung der API an die Anforderungen angepasst:

- Verbindung zu zugrundeliegenden Daten
- Layout-Algorithmen
- Einführung neuer Arten von Kanten und Knoten
- Benutzerinteraktion

Die Software ist als Basis für einen Quest-Editor geeignet. Jedoch sind die Lizenzkosten hoch. Außerdem zeichnen diese Software bis auf die integrierten Layout-Verfahren keine weiteren Merkmale aus, über die nicht auch andere, frei erhältliche Rahmenwerke verfügen. Deshalb wird die Evaluierung nicht vollständig ausgeführt und die Software nicht als Basis für einen Quest-Editor eingesetzt.

3.4 Rahmenwerke für grafische Editoren

Rahmenwerke erlauben dem Anwender die genaue Anpassung der vorgegebenen Klassen an die Zielanwendung. Deshalb ist dieser Ansatz der Erstellung eines Editors für die Anbindung an das zentrale AUTOFOCUS-Systemmodell besonders geeignet.

3.4.1 JHotDraw

JHotDraw ist ein Rahmenwerk für Editoren für technische und strukturierte Grafiken. Das Rahmenwerk wird als Open-Source-Projekt entwickelt. Das hauptsächliche Ziel der Entwicklung ist die Demonstration der eingesetzten Entwurfsmuster. Aus technischer Sicht ist ein Einsatz dieses Rahmenwerks als Basis für einen Quest-Editor möglich. Weil in diesem Rahmenwerk die Trennung zwischen Darstellung an der Benutzeroberfläche und Modell nicht realisiert ist, wird das Editorenkonzept nicht auf der Basis von JHotDraw realisiert. Denn die strikte Trennung ermöglicht die Strukturierung und den Entwurf einer geeigneten Architektur der Anwendung und erleichtert damit die Verständlichkeit des erarbeiteten Konzepts. Eine scharfe Trennung zwischen Quest und Editor ist nicht gegeben, damit würde die Anbindung beliebiger Editoren durch das Editoren-Konzept erschwert.

3.4.2 Diva

Diva ist ein Rahmenwerk zur Visualisierung von komplexen Hard- und Softwaresystemen und ihres Verhaltens. Besondere Betonung liegt auf der dynamischen und interaktiven Visualisierung. Dazu werden unterschiedliche Komponenten miteinander verbunden, welche die Daten bereitstellen, filtern und anzeigen. Das Rahmenwerk implementiert die strikte Trennung von Daten und ihren Repräsentationen. Die Architektur umfasst folgende Teile:

- Information Spaces repräsentieren ein Modell (im Sinne der MVC-Architektur [KP88]) mit zusätzlicher Infrastruktur.
- Surfaces sind eine Erweiterung des View-Teils einer MVC-Architektur. Die Daten werden hier visualisiert und modifiziert. Die Anzeige wird aktualisiert, falls sich die Daten des assoziierten Modellelements ändern.

- Als Background wird das Dateisystem oder eine Datenbank bezeichnet.

Die Kommunikation zwischen den Ebenen View und Modell ist in zwei Schnittstellen spezifiziert. Die eine Schnittstelle deklariert Methoden zur Abfrage und Veränderung von Daten, die andere bietet Methoden zur Benachrichtigung bei einer Veränderung von Daten in der Modellebene. Die wichtigsten Klassen sind in den folgenden Paketen zusammengefasst:

- `Canvas` bietet die Anzeigefläche für beliebige, benutzerdefinierte grafische Objekte.
- Das Paket `Graph` realisiert das Erstellen, Betrachten und Editieren von Graphen. Es basiert auf dem MVC-Daten- und Benachrichtigungsmodell. Die Repräsentation und das Layout der Graphen kann angepasst werden.
- `GUI` ist ein Rahmenwerk für die Realisierung von Benutzerschnittstellen.
- `Sketch` realisiert Datenstrukturen und Algorithmen für die Erkennung von Eingabemustern durch den Benutzer.
- `Utils` bietet beispielsweise Funktionalität zur Filterung von Daten.

Diva ist aus technischer Sicht als Basis für einen Quest-Editor geeignet. In der aktuell vorliegenden Version liegt der Schwerpunkt der Implementierung auf dem Paket `Canvas`. Das für diese Arbeit bedeutendere Paket `Graph` ist nur ansatzweise realisiert. Es fehlt die Implementierung von Standardaufgaben eines Editors, z.B. Funktionalität zum Kopieren, Einfügen oder Bearbeiten von Diagrammelementen. Diva bietet zwar sehr umfangreiche Darstellungsmöglichkeiten für beliebige grafische Objekte. Diese Eigenschaft ist jedoch für das Quest-Projekt nicht entscheidend. Vielmehr ist kein Referenz-Projekt bekannt, das unter der Verwendung von Diva implementiert ist. Die Beobachtung der Entwicklung von Diva zeigt, dass eine stetige Weiterentwicklung nicht gegeben ist. Aus den erwähnten Gründen wird Diva nicht als Basis für ein Editorenkonzept für Quest eingesetzt.

3.4.3 Visual Shape

Visual Shape [Mai99] ist ein Rahmenwerk zur Erstellung von Editoren und ist speziell für Quest-Editoren entwickelt. Es ist nach dem Baukastenprinzip strukturiert, d.h. die Realisierung der verschiedenen Aufgaben des Rahmenwerks ist in Komponenten unterteilt, die durch Schnittstellen getrennt sind. Deshalb ist der Austausch einzelner Komponenten möglich, die nach den jeweiligen Bedürfnissen einer Anwendung geändert oder weiterentwickelt sind. Die Entwicklung des Rahmenwerks basiert auf der Verwendung von Entwurfsmustern.

Visual Shape ist aus technischer Sicht für den Einsatz in Quest geeignet. Die Entwicklung des Rahmenwerks ist im Zustand eines Prototyps abgeschlossen. Die Implementierung ist im Bereich der Unterstützung eines eigenen Graphenmodells mit MVC-Konzept [KP88] und der grafischen Möglichkeiten zur Darstellung von Figuren ausbaufähig. Eine Weiterentwicklung und Pflege der Software ist daher für die Erstellung eines Quest-Editors auf Basis von Visual Shape vorausgesetzt. Weil der Prototyp des modellbasierten Editors im Rahmen der vorliegenden Arbeit entwickelt werden soll, ist eine zeitliche Abhängigkeit von der Weiterentwicklung des Rahmenwerks nicht vertretbar. Deshalb dient Visual Shape nicht als Basis für die Erstellung eines Quest-Editors.

3.4.4 GEF

Das GEF-Rahmenwerk wird als Open-Source-Projekt entwickelt und basiert auf dem Einsatz von Entwurfsmustern. Das Rahmenwerk dient zur Entwicklung von Anwendungen in mehreren Domänen zum Bearbeiten von Graphen, die grafisch dargestellt werden. GEF kann zur Entwicklung einer eigenen Anwendung oder einer Benutzerschnittstelle für ein vorhandenes Programmsystem eingesetzt werden. Das Rahmenwerk zielt auf die einfache Integration von anwendungsspezifischem Programmtext. Die dafür tatsächlich vorhandenen Einhängpunkte (engl. Hooks) sind auf wenige Operationen bei Änderungen am Graphenmodell beschränkt. Anwendungsspezifische Aktionen werden beim Einfügen von Knoten und Kanten und beim Entfernen von Kanten aufgerufen. Die Architektur des Rahmenwerks ist in zwei verschiedene Ebenen unterteilt: In der Ebene des Graphenmodells werden die Beziehungen zwischen Knoten und Kanten gespeichert. Die Diagrammschicht repräsentiert die Informationen zur grafischen Darstellung der Graphen durch Figuren. Die Beziehung zwischen den beiden Ebenen ist durch das MVC-Konzept [KP88] gekennzeichnet. Dabei reflektiert die Anzeige automatisch die Veränderungen am Graphenmodell. Das Rahmenwerk ist so entwickelt, dass Erweiterungen in Form von neuen Klassen und Ableitungen von vorhandenen Basisklassen hinzugefügt werden. Damit wird die einfache Anpassung der Anwendung an neue Versionen des Rahmenwerks unterstützt. Die Benutzerschnittstelle wird durch eine Zeichenfläche, Status-, Menü- und Werkzeugleiste gebildet. Ein Beispiel eines Editorfensters ist in Abbildung 2.2 auf Seite 9 dargestellt. Die wichtigste Funktionalität des Rahmenwerks wird von den folgenden Klassen realisiert:

- `Editor` steuert andere Objekte durch die Weitergabe von Nachrichten oder Ereignissen.
- Von der Klasse `Cmd` sind alle Klassen abgeleitet, die Aktionen im Editor ausführen. Beispielsweise sind die durchzuführenden Operationen beim Kopieren, Speichern oder Drucken in Klassen implementiert, welche die Klasse `Cmd` erweitern.
- Der Editor befindet sich in einem Modus, der durch eine von `Mode` abgeleiteten Klasse spezifiziert wird. Beispielsweise sind für das Einfügen oder Selektieren von grafischen Objekten unterschiedliche Modi implementiert.
- Das Platzieren von grafischen Objekten auf Gitterlinien der Zeichenfläche wird durch Unterklassen von `Guide` unterstützt.
- Alle Figuren sind von der Klasse `Fig` abgeleitet und visualisieren ein Element aus dem Graphenmodell. Das Rahmenwerk bietet eine Standardbibliothek für grafische Objekte.
- Die Anzeigefläche besteht aus mehreren Ebenen vom Typ `Layer`. Beispielsweise wird das Gitter zum Platzieren von Figuren durch eine Klasse realisiert, die `Layer` erweitert. Die grafischen Objekte zur Repräsentierung des Graphenmodells an der Benutzeroberfläche werden in einer eigenen Ebene zusammengefasst, die durch eine von `Layer` abgeleiteten Klasse implementiert wird.
- Die Klassen mit der Basisklasse `Selection` repräsentieren die grafischen Objekte für die Selektion von Figuren.
- Die Elemente des Graphenmodells sind von der Klasse `NetPrimitive` abgeleitet.

Das Rahmenwerk dient als Basis für die Entwicklung des freien UML-Werkzeugs ArgoUML [Arg02]. Es unterstützt das UML-Metamodell (OMG, Version 1.3) [For00], jedoch sind nicht alle Diagrammart implementiert. Das Werkzeug unterstützt die Anbindung an Datenbanken. Agenten überprüfen automatisch das entworfene Design und bieten Verbesserungsvorschläge an.

Das ArgoUML-Projekt zeigt die technische Realisierung eines grafischen Entwicklungswerkzeugs auf der Basis des GEF-Rahmenwerks. Diese Referenzimplementierung aus der gleichen Anwendungsdomäne wie Quest-Editoren erhöht die Konfidenz in das Ergebnis der Evaluierung von GEF. Die erwähnte Datenbankbindung und die realisierten Agenten demonstrieren die Erweiterungsmöglichkeiten zu einer Mehrbenutzeranwendung und die Realisierung von Benutzerführung. Diese Aspekte prädestinieren GEF für den Einsatz als Basis für Quest-Editoren. Die weitere Evaluierung zeigt, dass dieses Rahmenwerk für Quest-Editoren am geeignetsten ist.

3.4.5 Arakhnê

Arakhnê ist eine freie Java-Bibliothek zum Editieren und Visualisieren von Graphen. Das Rahmenwerk kann unabhängig von der Anwendungsdomäne eingesetzt werden und ist zur Erfüllung der gestellten Anforderungen anpassbar und erweiterbar. Es unterstützt die UML-Metamodellierung [RJB97]. Einige Ideen für das Design des Rahmenwerks stammen aus ArgoUML und GEF.

Die Dokumentation umfasst lediglich eine Beschreibung der API und einen kurzen Überblick über das Rahmenwerk. Deshalb ist die Einarbeitung mit einem höheren Aufwand verbunden als bei ausführlicher dokumentierten Programmen. Die Implementierung wird gegenwärtig lediglich durch vier Programmierer fortgeführt. Eine gesicherte und stetige Weiterentwicklung ist deshalb nicht gegeben. Aus diesen Gründen und der Tatsache, dass Arakhnê keine Eigenschaften besitzt, die das Rahmenwerk über andere Rahmenwerke auszeichnen, wird die Evaluierung nicht vollständig durchgeführt. Arakhnê wird deshalb nicht als Basis für einen Quest-Editor eingesetzt.

3.4.6 JGraph

JGraph ist kein Rahmenwerk im eigentlichen Sinn, sondern eine Graphen-Komponente für die Java-Swing-Bibliothek mit Anzeige- und Editierfunktionalität. Die Komponente kann in einer Java-Anwendung eingesetzt und in eine bestehende Benutzeroberfläche als Zeichenfläche integriert werden. Mit dem Paket können beliebige, graphenähnliche Strukturen visualisiert werden. Die Anzeige basiert auf einem internen Graphenmodell mit den Elementen Kanten, Knoten und Ports, das die Beziehungen zwischen den Elementen speichert und benutzerdefinierbar ist. JGraph realisiert damit das MVC-Konzept [KP88] mit mehreren möglichen, synchronisierten Sichten auf ein Modell. Veränderungen am Modell werden atomar ausgeführt und werden in einer Historie für spätere Undo- oder Redo-Operationen gespeichert. Der Diagramm-Editor JGraphpad ist eine umfangreiche Beispielanwendung, welche die Fähigkeiten von JGraph demonstriert. In diesem Editor können eigene grafische Bibliotheken erstellt und für das Zeichnen von Diagrammen eingesetzt werden.

Aus technischer Sicht ist JGraph für den Einsatz als Basis für einen Quest-Editor geeignet. JGraph und JGraphpad sind in einer freien und kommerziellen Version verfügbar. Die kommer-

ziellen Versionen werden weiterentwickelt. Die Weiterentwicklung der freien Versionen ist von der Beteiligung der Open-Source-Entwickler abhängig. Außer JGraphpad sind zum Zeitpunkt der Evaluierung keine Referenzimplementierungen für den Einsatz von JGraph bekannt, welche die Entscheidung für die Wahl von JGraph untermauern würden. Aus den erwähnten Gründen wird JGraph nicht als Basis für einen Quest-Editor eingesetzt.

3.5 Visualisierungswerkzeuge für Graphen

3.5.1 VCG

Durch VCG werden Graphen visualisiert und auf diesen Layout-Verfahren zur Platzierung der Knoten angewendet. Das Eingabeformat für die Darstellung ist ein textuelles Beschreibungsformat. Die Software unterstützt nicht das Editieren von Graphen, sondern ausschließlich ihre Visualisierung. Die Implementierung ist in den Sprachen C und C++ realisiert. Die Anbindung an Quest würde durch Dateiaustausch realisiert. Dies widerspricht einem modellbasierten Konzept für einen Quest-Editor. Das Programm besitzt außerdem keine Editorfunktionalität. Diese Funktionalität muss ergänzt werden, wodurch der Implementierungsaufwand steigt. Aus diesen Gründen kommt VCG als Basis für einen Quest-Editor nicht zum Einsatz.

3.5.2 AbsInt aiSee Graph Visualization

Das Programmsystem basiert auf VCG und visualisiert Graphen durch ein integriertes Layout-Verfahren. Die Software ist ursprünglich für die Anzeige der internen Datenstrukturen entwickelt, die bei Programmläufen von Übersetzern auftreten. Es können jedoch auch beliebige andere Graphen angezeigt werden.

Die Anwendung visualisiert lediglich Datenstrukturen, die Editierfunktionalität muss realisiert werden. Dadurch steigt der Implementierungsaufwand. Der Informationsaustausch mit anderen Anwendungen basiert auf Dateien mit textuellem Beschreibungsformat. Ein modellbasierter Editor kann deshalb nicht mit aiSee realisiert werden. Die Evaluierung der Software wird nicht vollständig durchgeführt, weil das Ausschlusskriterium der Anbindung an Quest nicht erfüllt ist. aiSee wird nicht als Basis für einen Quest-Editor eingesetzt.

3.5.3 daVinci Presenter

daVinci Presenter visualisiert Graphen durch den Einsatz eines automatischen Layout-Verfahrens. Die Eingabeinformationen für den Graphen können in einer speziellen Datei mit textuellem Format stehen oder von einer anderen Anwendung erstellt werden. Das Programmsystem wird durch eine API in bestehende Anwendungen integriert. Die Software ist ausschließlich für die Visualisierung zuständig. Eine externe Anwendung interagiert mit daVinci Presenter über die API, sobald die Struktur des Graphen verändert wird und löst die Aktualisierung der Anzeige entsprechend der Veränderung aus. Die externe Anwendung steuert und verändert die Struktur des Graphen und wird durch die API über die Interaktionen des Benutzers an der Zeichenfläche benachrichtigt. Beispielsweise wird das Einfügen neuer Kanten oder Knoten der externen Anwendung angezeigt. Letztere reagiert darauf und kann dabei weitere Operationen ausführen. Alle Funktionen von daVinci Presenter sind über die API steuerbar. Das Programm unterstützt

die Anzeige eines Graphen in mehreren Fenstern. Die Benutzeroberfläche kann durch Menüeinträge erweitert werden, um die Funktionalität der externen Anwendung zu integrieren.

Sequenzdiagramme zeichnen sich durch ein besonderes, festgelegtes Layout aus, das in daVinci Presenter nicht implementiert ist. In der momentanen Ausbaustufe des Programms müsste dieser Algorithmus in der externen Anwendung implementiert werden und das Visualisierungswerkzeug mit Positionen für die Knoten versehen. Außerdem müssen die Knotengraphiken erweitert werden. Der Implementierungsaufwand für einen Quest-Editor erhöht sich wegen der Realisierung des Layout-Verfahrens und der Editierfunktionalität. Deshalb wird dieses Programmsystem als Basis für einen Quest-Editor nicht eingesetzt.

3.6 Graphen-Editoren

3.6.1 VGJ

VGJ ist ein Werkzeug zum Zeichnen und für das Layout von Graphen, deren Eingabe durch ein textuelles Format oder durch die Konstruktion im Editor erfolgt. Das Programm ist in der Sprache Java implementiert.

Da die Entwicklung beendet ist, ausschließlich Graphen visualisiert werden und der Austausch mit einer anderen Anwendung über Beschreibungsdateien realisiert ist, wird die Software nicht vollständig evaluiert. Die Anbindung an Quest über Dateiaustausch widerspricht dem Ziel der Entwicklung eines modellbasierten Editorenkonzepts in dieser Arbeit. Aus den genannten Gründen dient VGJ nicht als Basis für einen Quest-Editor.

3.6.2 GraphEd und Graphlet

GraphEd ist ein erweiterbarer Editor für Graphen mit integrierten Layout-Algorithmen. Die Anwendung unterstützt die Konstruktion und Modifikation von Graphen, deren Kanten und Knoten in ihrer Darstellung angepasst werden können. Die Kommunikation mit anderen Programmen erfolgt durch den Austausch von Dateien. Die Entwicklung von *GraphEd* ist eingestellt, der Nachfolger ist *Graphlet*, dessen Entwicklung in einem kontinuierlichen Projekt an der Universität Passau erfolgt.

Die Editoren *GraphEd* und *Graphlet* unterstützen ausschließlich die Konstruktion von Graphen. Für das Anwendungsfeld Quest ist jedoch auch die Konstruktion von AUTOFOCUS-Sequenzdiagrammen vorauszusetzen. Die Kommunikation mit anderen Anwendungen erfolgt über den Dateiaustausch. Dies widerspricht jedoch dem Ziel der Entwicklung eines modellbasierten Ansatzes für Quest-Editoren. Weil das Ausschlusskriterium der Anbindung an Quest damit nicht erfüllt ist, wird die Evaluierung nicht vollständig durchgeführt. Aus den genannten Gründen werden die beiden Editoren nicht als Basis für Quest-Editoren eingesetzt.

3.6.3 Tom Sawyer Software

Das Softwaresystem *Graph Editor Toolkit* besteht aus einer anpassbaren und erweiterbaren Klassenbibliothek mit einer API zur Anpassung der Darstellung von Graphen und dient zur Erstellung eines Grapheneditors. Für das Layout von Graphen stützt sich dieses Softwarepaket auf die Bibliothek *Graph Layout Toolkit*. Das Editor-Paket kann mit beliebigen Daten-Sammlungen verbunden werden und in bestehende Anwendungen integriert werden.

Die Software unterstützt ausschließlich die Konstruktion und Visualisierung von Graphen. Da für das Anwendungsfeld Quest jedoch auch andere Zeichnungen wie AUTOFOCUS-Sequenzdiagramme vorausgesetzt sind, wird die Evaluierung nicht vollständig durchgeführt. Aus dem genannten Grund kann das Programmpaket als Basis für einen Quest-Editor nicht eingesetzt werden.

3.7 Benutzerdefinierbare Entwicklungsumgebungen

Die Software *Eclipse Platform* ist für die Erstellung von benutzerdefinierbaren Entwicklungsumgebungen entwickelt. Das zusätzliche Paket GEF (engl. Graphical Editor Framework) ermöglicht die Erstellung eines grafischen Editors entsprechend eines Anwendungsmodells. Die erstellte Entwicklungsumgebung orientiert sich am Programmtext bzw. an Dokumenten. Dieser Ansatz steht im Widerspruch zur Modellbasierung von Quest. Aus diesem Grund wird die Evaluierung nicht vollständig durchgeführt. Die Software dient nicht als Basis für einen Quest-Editor.

4 Ergebnis der Evaluierung

Die Evaluierung unterschiedlicher Ansätze für die Erstellung von Editoren zeigt, dass Rahmenwerke als Basis zur Realisierung von Quest-Editoren am geeignetsten sind. Mit ihnen gelingt die enge Kopplung mit dem AUTOFOCUS-Metamodell am besten. Rahmenwerke zeichnen sich durch ihre Anpassbarkeit an die jeweiligen Anforderungen besonders gegenüber anderen Ansätzen aus. Eine Vielzahl von Editoren-Werkzeugen kann nicht eingesetzt werden, weil ihnen die Modellbasierung fehlt. Diese Eigenschaft ist auch bei Vertretern anderer Kategorien nicht gegeben. Eine weitere große Gruppe an Werkzeugen unterstützt die ausschließliche Visualisierung von Graphen. Damit können von diesen keine AUTOFOCUS-Sequenzdiagramme abgedeckt werden.

Aus technischer Sicht ist der Einsatz der Rahmenwerke Diva, JGraph, Visual Shape und GEF als Basis für einen Quest-Editor möglich. Die Gesamtergebnisse ihrer Evaluierung sind in Tabelle 4.2 zusammengefasst.

Rahmenwerk	Note
Diva	4,2
GEF	4,3
JGraph	4,5
Visual Shape	4,1

Tabelle 4.2. Ergebnisse der Evaluierung der aus technischer Sicht einsetzbaren Rahmenwerke. Je höher die Bewertung, desto geeigneter ist das Rahmenwerk für einen Einsatz als Basis für einen Quest-Editor.

Visual Shape wird nicht als Basis für einen Quest-Editor eingesetzt, weil dieses Rahmen-

werk in seiner aktuellen Implementierung einen zu geringen Umfang der Funktionalität besitzt. Für einen sinnvollen Einsatz als Ausgangspunkt für einen Quest-Editor ist eine Weiterentwicklung erforderlich. Um die Abhängigkeit der Implementierung des prototypischen Editors von der Weiterentwicklung des Rahmenwerks zu vermeiden, kommt Visual Shape nicht zum Einsatz. Um das Risiko zu minimieren, bei der Wahl eines Rahmenwerks eine Fehlentscheidung zu treffen, wird GEF für die Verwendung als Basis für einen Quest-Editor ausgewählt. Denn mit ArgoUML ist eine Referenzimplementierung aus dem Anwendungsgebiet der grafischen Modellierungswerkzeuge bekannt. Dadurch wird die Konfidenz der Evaluierung erhöht.

Das Ergebnis der Evaluierung fließt in die Analyse der Anforderungen an den Quest-Editor und die Schnittstelle im folgenden Kapitel bereits ein. In der Anforderungsanalyse für den zu realisierenden prototypischen Editor wird Funktionalität des gewählten Rahmenwerks GEF berücksichtigt.

Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die Schnittstelle zwischen Quest und einem Editor und an den Editor selbst beschrieben. Dabei werden die einzelnen Ziele nach Prioritäten klassifiziert. Diese Klassen werden im folgenden Abschnitt 1 erläutert. In Abschnitt 2 werden die Rahmenbedingungen beschrieben. Die Anforderungsanalyse des Editors in Abschnitt 3 auf der nächsten Seite wird nach den Aspekten Benutzerschnittstelle, Funktionalität und Dokumentation aufgeteilt. In Abschnitt 4 werden die Anforderungen der Schnittstelle dargestellt, die durch die Anforderungen des Editors bestimmt werden. Die Analyse erfolgt vor dem Hintergrund der Erstellung eines Editors für AUTOFOCUS-Systemstrukturdiagramme. Für die Validierung des Editorenkonzepts ist die Implementierung eines der drei AUTOFOCUS-Editoren geeignet, denn jede Art stellt einen Ausschnitt aus dem Systemmodell dar. Zustandsübergangsdigramme besitzen eine Graphenstruktur, ermöglichen die Verfeinerung der Spezifikation des Verhaltens des modellierten Systems und erlauben damit unterschiedliche Sichten auf das hierarchische Modell. Systemstrukturdiagramme zeichnen sich durch eine Graphenstruktur aus und können ebenso hierarchisch verfeinert werden. Komponenten besitzen eine innere und äußere Sicht. Sequenzdiagramme besitzen diese Eigenschaft nicht. Die Evaluierung des Rahmenwerks GEF zeigt, dass Bézier-Kurven zur Darstellung der Transitionen in Zustandsübergangsdigrammen nicht Teil der Standardbibliothek der grafischen Objekte sind. Rechtwinklige Kanten, wie sie in Systemstrukturdiagrammen erforderlich sind, werden von GEF implementiert. Aus den genannten Gründen wird als Prototyp ein Systemstrukturdiagramm-Editor realisiert.

1 Anforderungsklassen

Für die Gruppierung von Anforderungen in Klassen werden wichtige, optionale und erweiterbare Eigenschaften unterschieden. Diese Klassen sind wie folgt definiert:

- Eigenschaften der *Klasse 1* werden ohne Einschränkung implementiert.
- *Klasse 2* fasst alle Anforderungen zusammen, die optional realisiert werden.
- Die Eigenschaften in *Klasse 3* sind bei der Implementierung nicht vorgesehen, finden aber bei der Entwicklung der Architektur Berücksichtigung.

2 Rahmenbedingungen

Die Rahmenbedingungen sind durch Quest und die Entscheidung für ein Editor-Rahmenwerk aus Kapitel 4 auf Seite 21 bestimmt. Deshalb werden die folgenden Anforderungen mit der Priorität 1 bewertet.

- Die Software wird in der Programmiersprache *Java* mit der Swing-Bibliothek realisiert.
- Die Anwendung wird auf Basis des *Rahmenwerks GEF* implementiert.
- Anbindung an Quest: Der Editor arbeitet auf dem *Quest-Modell*.
- Anbindung an den *Quest-Browser*: Eine enge Integration des Editors in den Browser ist nicht nötig. Der Editor stellt eine eigenständige Anwendung dar, die vom Browser lediglich aufgerufen wird.

3 Anforderungsanalyse des Editors

Um die Anforderungen an die Schnittstelle herauszuarbeiten, werden zunächst die Eigenschaften des Editors bestimmt, aus denen Merkmale der Schnittstelle abgeleitet werden.

3.1 Benutzerschnittstelle

Die Mensch-Maschine-Schnittstelle ist so realisiert, dass sie *intuitiv bedienbar* und *ansprechend gestaltet* ist. Diese Anforderung wird durch die Verwendung der *Swing-Komponenten* unterstützt. Die folgende Übersicht beschreibt weitere Eigenschaften der Benutzerschnittstelle:

- Die *Größe der Zeichenfläche* passt sich automatisch der Größe des Diagramms an. (Klasse 1)
- *Beenden des Editors*, ggf. mit einem Hinweis auf ein noch nicht gespeichertes Dokument. (Klasse 1)
- *Verschieben* der grafischen Objekte (Klasse 1)
- Elementare *Editieroperationen* umfassen das Erstellen und Löschen von Modell- und View-Elementen. (Klasse 1)
- *Erstellen* von grafischen Objekten, die View-Elemente repräsentieren. (Klasse 1)
- Der Browser bietet die Funktionalität zum *Öffnen bzw. Erstellen eines Diagramms* in einem Editorfenster. (Klasse 1)
- Beim *Kopieren, Einfügen und Ausschneiden* werden Änderungen am Modell, den View-Elementen und an der Zeichenfläche vorgenommen. Beim Kopieren und Einfügen ist ein Konzept für die eindeutige Namensgebung der konstruierten Modellelemente erforderlich. (Klasse 2)
- Bei der Größenänderung einer Komponente folgt die Bewegung der Zeichenfläche automatisch (*Auto-Scrolling*). (Klasse 2)
- *Größenänderung* der grafischen Objekte (Klasse 2)
- *Drucken* von Diagrammen (Klasse 2)

- *Tooltips* für die grafischen Objekte zeigen Informationen der assoziierten Modellelemente an. (Klasse 3)
- *Kontextmenüs* ermöglichen beim Anwählen eines Editorelements, dieses zu kopieren, zu entfernen oder die Attribute des assoziierten Modellelements zu ändern. (Klasse 3)
- Die *Modellattribute* werden auch direkt im Editorfenster geändert, nicht nur über die Menüeinträge. Beispielsweise können die Bezeichnungen für Editor- und damit auch der Modellelemente durch einen Doppelklick editiert werden. (Klasse 3)
- Ein *Hilfesystem* unterstützt den Benutzer beispielsweise beim Anlegen einer neuen Komponente, indem die Semantik von Systemstruktur-Diagrammen erläutert, ein solches als Vorlage angezeigt und auf mögliche Folgeaktionen für den Benutzer hingewiesen wird. Dem Anwender werden beim Zeichnen von Kanälen z.B. alle noch nicht verbundenen Ports zur Auswahl angezeigt. Das Hilfesystem kann bei Bedarf deaktiviert werden. (Klasse 3)
- *Undo- und Redo-Operationen* erfolgen auf dem Quest-Modell und haben ebenso Auswirkungen auf die View- und Editorelemente. Die Protokollierung der Aktionen wird vom Modell realisiert. Eine weitere Möglichkeit für die Implementierung basiert auf dem Paket `javax.swing.undo`. Der Editor bietet lediglich die entsprechenden Menüeinträge für die Aktionen Rückgängig und Wiederherstellen und reflektiert die Änderungen. (Klasse 3)
- *Exportieren* in den Formaten EPS, JPEG und QML (Quest Model Language) [BLS02]. In einer QML-Datei werden die Modell- und View-Daten des zentralen Datenspeichers in einem textuellen Format gespeichert. Für weitere, über die Funktionalität des Prototyps hinausgehenden Anforderungen sind Erweiterungen am QML-Format nötig, weil die bisherige View-Schnittstelle für die Persistenz von Daten nicht vollständig ist. (Klasse 3)
- *Zoom* zur Navigation in komplexen Diagrammen (Klasse 3)
- *Suchfunktion* zur Suche nach den Vorkommen eines eingegebenen Musters im aktuellen Dokument (Klasse 3)

3.2 Funktionalität

3.2.1 Persistenz

Die Persistenz von Daten bezeichnet die Abspeicherung der Informationen, die beim Öffnen eines bereits bearbeiteten Diagramms wieder eingelesen werden. Dabei werden sowohl View- als auch Modelldaten berücksichtigt. Die zur Laufzeit der Anwendung nicht persistenten Daten bilden die Basis für die zu speichernden Daten. Um erstere beim Mehrbenutzerbetrieb konsistent zu halten, sind Mechanismen zur Regelung von parallelen Zugriffen auf die Daten erforderlich. Die Anforderungsanalyse für die Persistenz und den Mehrbenutzerbetrieb resultiert in folgenden Ergebnissen:

- Die Implementierung der Funktionalität zum *Laden und Speichern* von Modell- und View-Daten im QML-Format [BLS02], die für die Realisierung des prototypischen Editors erforderlich ist, ist bereits im Quest-Browser realisiert. Da der Editor ebenso auf diesen Daten arbeitet, entfällt die separate Bereitstellung dieser Funktionalität. Für weitere Anforderungen sind auch Erweiterungen am QML-Format nötig, weil die bisherige View-Schnittstelle für die Persistenz von Daten nicht vollständig ist. Die Projekte des zentralen Datenspeichers und ihre Elemente können durch den Browser bearbeitet werden. Modellausschnitte in Form eines Systemstrukturdiagramms werden im Editor bearbeitet. Beim Erstellen oder Laden eines Diagramms werden die benötigten Informationen aus dem zentralen Datenspeicher ausgelesen und im Editor angezeigt. (Klasse 1)
- Modellelemente können zu einem Zeitpunkt nur von einem Anwender schreibend bearbeitet werden. Parallele Lesezugriffe durch mehrere Anwender sind möglich. Lösungsansätze für den *Mehrbenutzerbetrieb* werden in Kapitel 8 auf Seite 103 vorgestellt. Für die vorliegende Arbeit ist die Unterstützung der Mehrbenutzerfähigkeit von untergeordneter Bedeutung. (Klasse 3)

3.2.2 Grafische Objekte

Für einen Systemstrukturdiagramm-Editor werden die folgenden grafischen Elemente benötigt. Sie können erstellt, geändert und entfernt werden.

- *Komponenten* werden durch Rechtecke repräsentiert (Klasse 1). Detailinformationen (Name der Komponente, Namen der Teilkomponenten) werden durch Tooltips oder Kontextmenüs angezeigt (Klasse 2).
- *Ports* werden durch Kreise dargestellt, abhängig von ihrer Semantik sind sie weiß oder schwarz (Klasse 1). Die unterschiedlichen Sichten auf Ports und die Auswirkung auf ihre Darstellung in einem Diagramm sind in Abschnitt 1.2 auf Seite 8 näher erläutert. Detailinformationen zu den Ports (Typ und Name) werden durch Tooltips oder Kontextmenüs angezeigt (Klasse 2). Die Ports sind abhängig von ihrer Semantik an Komponenten gebunden oder separate Bestandteile des Diagramms (Klasse 1).
- *Kanäle* werden durch gerichtete Kanten zwischen Ports dargestellt (Klasse 1). Informationen über den Typ und den Namen werden Tooltips oder Kontextmenüs entnommen (Klasse 2). Die Kanäle verlaufen rechtwinklig (Klasse 1).

3.2.3 Layout-Algorithmen

Layout-Algorithmen passen die Darstellung der grafischen Elemente einer Zeichenfläche so an, dass die vorgegebenen Eigenschaften erfüllt werden. Durch die Vorgaben wird beispielsweise beschrieben, dass sich Figuren nicht überlappen oder dass das gesamte Diagramm in einer übersichtlichen Darstellung angezeigt wird. Für einen Systemstrukturdiagramm-Editor passt das Layout-Verfahren die Darstellung der Kanäle an, wenn Komponenten verschoben werden. Ziel ist die Vermeidung der Kreuzung von Kanälen und Komponenten. Prinzipiell werden zwei Alternativen für die Realisierung unterschieden:

- Einbindung von externen Layout-Algorithmen: Das Rahmenwerk besitzt Einhängpunkte für die Integration von benutzerdefinierten Layout-Bibliotheken.
- Verwendung von Layout-Algorithmen des Rahmenwerks: Im Rahmenwerk ist eine eigene Bibliothek für Layout-Verfahren integriert.

Das Layout-Verfahren des gewählten Rahmenwerks GEF implementiert nur ein einfaches Vorgehen, das lediglich die Quelle und das Ziel einer Kante berücksichtigt, aber keine weiteren Elemente der Zeichenfläche. Diese Implementierung kann zur Überlappung von Figuren für Komponenten und Kanäle führen. Ein Verfahren, welches das gesamte Diagramm bei der Berechnung des Layouts einbezieht und auch die automatische Veränderung der Position der Ports beim Verschieben von Komponenten implementiert, erfordert die Integration eines benutzerdefinierten Layout-Verfahrens. Lösungsansätze dazu sind in [Löt99] beschrieben.

3.2.4 Modellbasierung

Im Editor wird ein Ausschnitt aus dem Quest-Modell visualisiert. Über die grafischen Objekte wird das Modell verändert. Als Anforderungen für die Modellbasierung ergeben sich folgende Aspekte:

- Die beim Editieren entstandenen Änderungen wirken sich auf das *Quest-Modell* aus. (Klasse 1)
- *Änderungen am Quest-Modell* werden durch Ereignisse vom Typ `ModelChangeEvent` propagiert. Durch die Übertragung der vorhandenen Ereignis-Implementierung auf das Editorenkonzept werden auch andere, auf dem Modell arbeitende Werkzeuge über die Modelländerungen benachrichtigt, die im Editor vorgenommen werden. Durch die Modellbasierung wird beispielsweise auch folgender Anwendungsfall unterstützt: Bei der Bearbeitung des Modells über einen Sequenzdiagrammeditor wird auch ein weiterer, geöffneter Editor für ein Systemstrukturdiagramm über die Änderung benachrichtigt, um die Anzeige entsprechend zu aktualisieren. Für die Art der Benachrichtigung stehen zwei Varianten zur Auswahl: Die Benachrichtigung kann inkrementell erfolgen. Dabei wird durch die Ereignisse angezeigt, welche Teile des Modells sich auf welche Weise ändern. Alternativ erfolgt die Mitteilung durch ein Ereignis, das die Änderung des gesamten Modells bezeichnet. Für die Realisierung werden die Ereignisse, die Modelländerungen anzeigen, in geeignete Nachrichten an den Editor umgesetzt werden. (Klasse 1)
- Die Anwendung unterstützt die Integration des in [Wil02] beschriebenen modellbasierten Hilfesystems. (Klasse 3)
- Die *Drag-and-Drop-Unterstützung* ermöglicht z.B. das Ziehen von Komponenten aus einem Systemstruktur- in ein Sequenzdiagramm. (Klasse 3)

3.2.5 Navigation in hierarchischen Komponenten

Die Navigation durch geschachtelte Komponenten erfolgt über den Quest-Browser. Die baumartige Struktur des aktuellen zentralen Datenspeichers mit seinen Projekten wird im Browser angezeigt. Die Verfeinerungen von Komponenten in Systemstrukturdiagrammen werden durch einen Eintrag in der Menüleiste des Browsers in eigenen Editorfenstern angezeigt.

3.2.6 Funktionalität für das Anwendungsfeld Quest

Die Anpassung an geänderte Anforderungen des AUTOFOCUS-Metamodells wird möglichst durch eine generische Schnittstelle unterstützt. Die Überprüfung der Einhaltung von Syntaxregeln erleichtert die Arbeit des Anwenders. Aus dem Anwendungsfeld Quest ergeben sich damit die folgenden Aspekte der Anforderungsanalyse:

- Bestimmte Aktionen, z.B. das Verbinden zweier Ports, werden nicht zugelassen, falls die *Syntaxregeln* nicht eingehalten werden. (Klasse 2)
- Das Design für den Editor bzw. die Schnittstelle ist so gestaltet, dass die Erweiterung und Anpassung des Editors an ein *geändertes AUTOFOCUS-Metamodell* möglich ist. (Klasse 1)

3.3 Dokumentation

Das erarbeitete modellbasierte Editorenkonzept dient als Basis für weitere AUTOFOCUS-Editoren. Um die Übertragung des Konzepts zu unterstützen und zu vereinfachen, ist eine ausführliche Dokumentation erforderlich, die das Design der Anwendung und die nötigen Erweiterungen an Quest-Klassen und am eingesetzten Rahmenwerk beschreiben. Bei der Dokumentation werden folgende Kategorien unterschieden:

- Das Design der Anwendung wird durch *UML* [RJB97] beschrieben. (Klasse 1)
- Einfügen von *Javadoc-Kommentaren* in den Programmtext (Klasse 1)
- *Änderungen an den Modell- und View-Klassen* von Quest für die Realisierung des Editorenkonzepts werden dokumentiert. (Klasse 1)
- *Änderungen und Erweiterungen am Rahmenwerk GEF* werden möglichst vermieden, um den problemlosen Einsatz von neuen Versionen zu gewährleisten. (Klasse 1)

4 Anforderungsanalyse der Schnittstelle

Aus der Anforderungsanalyse für den Editor resultieren die Anforderungen an die Schnittstelle zwischen Quest und Editor. In den folgenden Abschnitten wird dargestellt, wie die Anforderungen des Editors aus Abschnitt 3 auf Seite 38 durch die Schnittstelle zwischen Quest und Editor realisiert wird und welche Auswirkungen auf andere Programmteile der Anwendungen entstehen.

4.1 Benutzerschnittstelle

Die Zugriffe auf Modell- und View-Elemente erfolgt über die Editorelemente der Zeichenfläche. Dazu sind die grafischen Objekte der Benutzerschnittstelle mit View-Elementen assoziiert. Damit können die Attribute der View-Ebene verändert werden. Über die View-Elemente erfolgt die Navigation in das Quest-Modell. Anforderungen der Benutzerschnittstelle haben folgende Auswirkungen auf die Schnittstelle:

- *Elementare Editorfunktionen* (Erstellen, Löschen) werden vom Editor über die View-Elemente in das Quest-Modell propagiert.
- *Kopier-, Einfüge- und Ausschneideoperationen* haben Auswirkungen auf das Quest-Modell und die Anzeige im Editor.
- Beim *Öffnen* eines Diagramms werden die Daten des Quest-Modells und der View-Elemente geladen und angezeigt.
- Beim *Verschieben* und der *Größenänderung* der grafischen Objekte verändern sich die Attribute der View-Elemente.
- Das *Drucken* und *Exportieren* von Diagrammen wird durch das Rahmenwerk realisiert.
- Die *Zoom-Funktion* hat keine Auswirkungen auf View-Elemente, sondern verändert lediglich den Maßstab für die Anzeige.
- *Undo- und Redo-Operationen* setzen die Protokollierung aller Aktionen auf den Modell- und View-Elementen voraus.
- Beim *Suchen* ist ein Zugriff auf das Quest-Modell nötig. Dabei werden die Attribute (Name, Typ) der Modellelemente durchsucht.
- Die automatische *Größenänderung der Zeichenfläche* wird durch das Rahmenwerk realisiert und hat keine Auswirkungen auf View-Elemente.
- Beim *Auto-Scrolling* werden keine Attribute der View-Elemente verändert.
- Werden *Modellattribute* von Modellelementen verändert, werden die Eigenschaften der grafischen Objekte (Tooltips, Beschriftungen) angepasst.
- Das Anzeigen von *Kontextmenüs* und *Tooltips* wird durch das Rahmenwerk realisiert.
- Das *Beenden* des Editors hat keine Auswirkungen auf die gespeicherten Modell- und View-Daten. Die persistente Speicherung der Daten ist im Quest-Browser implementiert.

4.2 Funktionalität

4.2.1 Persistenz

Die *Mehrbenutzerfähigkeit* wird in der Realisierung nicht berücksichtigt. Mechanismen für die Synchronisation im Mehrbenutzer- und Mehrprogrammbetrieb sind in Kapitel 8 auf Seite 103 erläutert. Das *Laden* und *Speichern* von Diagramminformationen hat keine Auswirkungen auf die Schnittstelle, weil diese Funktionalität durch den Quest-Browser realisiert wird. Für weitere, über die Funktionalität des prototypischen Editors hinausgehenden Anforderungen ist eine Erweiterung des QML-Formats erforderlich, weil die bisherige View-Schnittstelle für die Persistenz von Daten nicht vollständig ist.

4.2.2 Grafische Objekte

Die Darstellung der grafischen Objekte erfolgt durch das Rahmenwerk. Die Figuren werden für die Anbindung an Quest mit den View-Elementen assoziiert. Die View-Elemente benachrichtigen den Editor über Änderungen in der View- oder Modell-Ebene über die Schnittstelle. Die grafischen Objekte des Editors reflektieren diese Änderungen im Editorfenster.

4.2.3 Funktionalität für das Anwendungsfeld Quest

Die Anpassungsfähigkeit der Anwendung an ein geändertes AUTOFOCUS-Metamodell beeinflusst den Entwurf der Schnittstelle zwischen Quest-Modell und Editor. Die Überprüfung der Einhaltung von Syntaxregeln erfordert eine entsprechende Funktionalität der Schnittstelle. Diese Aspekte resultieren in folgenden Anforderungen der Schnittstelle:

- Die Einhaltung von *Syntaxregeln* wird durch das zentrale Systemmodell gesteuert. Nur die vom Modell erlaubten Aktionen werden tatsächlich ausgeführt und die Änderungen über die View- an die Editorelemente propagiert und visualisiert.
- Der Editor bietet die Möglichkeit zur *Erweiterung* und *Anpassung* an weitere Anforderungen. Die Schnittstelle ist dazu derart gestaltet, dass eine Erweiterung oder Änderung des AUTOFOCUS-Metamodells keine Auswirkungen auf die gesamte Anwendung hat.

4.3 Weitere Anforderungen

Das *Layout-Verfahren* wird in der prototypischen Implementierung durch das Rahmenwerk realisiert. Ein erweiterter Layout-Algorithmus mit der besonderen Berücksichtigung der Anforderungen für ein Systemstrukturdiagramm ist am günstigsten nicht im Editor, sondern als Teil der Views implementiert. Dadurch ergeben sich für die Schnittstelle zusätzliche Anforderungen, welche die Funktionalität für die Kommunikation von Layout-Änderungen betreffen.

Die *Modellbasierung* der Schnittstelle ist per se gegeben, weil durch die Schnittstelle die Aufrufe von Modelloperationen aus dem Editor an das Quest-Modell weitergeleitet werden. Die *Navigation in hierarchischen Komponenten* beeinflusst die Schnittstelle nicht, da sie durch den Quest-Browser realisiert wird. Im Browser werden die Editorfenster für die Verfeinerung von Komponenten durch Systemstrukturdiagramme konstruiert. Die *Dokumentation* der Schnittstelle ist durch die in Abschnitt 3.3 auf Seite 42 bereits erwähnten Formen abgedeckt.

5 Vertikaler Prototyp

Der prototypische Editor demonstriert die Funktionsfähigkeit der Schnittstelle. Der Prototyp verifiziert die Anbindung eines beliebigen Editors an das Quest-Modell. Deswegen ist für den Prototyp die Implementierung einer vertikalen Editorfunktionalität ausreichend. Eine besondere Stellung nimmt dabei die Funktionalität ein, die Auswirkungen auf das Quest-Modell hat. Der Editor visualisiert einen Ausschnitt aus dem zentralen Systemmodell. Deshalb sind auch Änderungen am Modell im Editor anzuzeigen. Die gesamte Anwendung berücksichtigt also Kommunikationsvorgänge in zwei Richtungen, und zwar vom Editor zum Modell und vom Modell zum Editor.

Vertikaler Prototyp

Die zu implementierende Funktionalität ergibt sich aus der Einteilung der Anforderungen in diesem Kapitel in ihre Wichtigkeitsklassen: Die Anforderungen der Klasse 1 sind im Prototyp zu realisieren, die der Klasse 2 können optional implementiert werden. Die Anforderungen aus Klasse 3 sind beim Entwurf des Editorkonzepts zu berücksichtigen, um eine Implementierung zu ermöglichen.

Die in diesem Kapitel erläuterten Anforderungen des Editors und der Schnittstelle bilden die Basis für den Entwurf des Designs der gesamten Anwendung. Im folgenden Kapitel wird das Design des modellbasierten Editorenkonzepts aus den Anforderungen abgeleitet.

Design

Dieses Kapitel erläutert als Ausgangspunkt für den Entwurf des Designs in Abschnitt 1 allgemeine Architekturprinzipien. Diese Prinzipien werden auf die Aufgabenstellung des Editorenkonzepts übertragen und bilden mit den in Kapitel 5 auf Seite 37 beschriebenen Anforderungen die Basis für die Ableitung des Designs für die Anwendung. Design-Entscheidungen für die Art der Kommunikation zwischen den Programmteilen werden in Abschnitt 3 begründet. Zusammenfassend stellt Abschnitt 4 auf Seite 59 die gewählte Kommunikation zwischen den Teilen der Softwarearchitektur dar. Abschnitt 5 beschreibt, durch welche Teile der Anforderungen die Design-Entscheidungen motiviert sind.

1 Architektur

Unter der *Architektur* wird die Aufteilung eines Softwaresystems in mehrere Programmteile (meist *Komponenten* oder *Module*), ihre Schnittstellen, die Prozesse und Abhängigkeiten zwischen ihnen, sowie die benötigten Ressourcen verstanden [PR97]. Wichtige Eigenschaften einer Architektur sind die Architekturprinzipien und Organisationsformen.

1.1 Architekturprinzipien

1.1.1 Geheimnisprinzip

Durch die Kapselung oder das Verbergen (engl. *Information Hiding*) von Implementierungsentscheidungen werden Module nach dem Black-Box-Prinzip aufgebaut. Durch die Schnittstellenspezifikation eines Moduls werden seine Interaktionspunkte beschrieben. Das interne Verhalten bleibt für den Benutzer des Moduls verborgen. Damit werden nur die relevanten Informationen nach außen sichtbar, wodurch die Verständlichkeit verbessert und die Fehleranfälligkeit reduziert wird.

1.1.2 Entwurfsmuster

Die Analyse von objektorientierter Software zeigt, dass für ähnliche Aufgabenstellungen gleiche Kombinationen von Entwurfskomponenten verwendet werden. Dabei handelt es sich um Klassen, die ähnliche Dienste erbringen. Diese Konzepte sind nach [EG95] in sogenannte Entwurfsmuster (engl. *Design Patterns*) eingeteilt. Ein Entwurfsmuster liefert ein generisches Lösungsschema für wiederkehrende Aufgaben. Die Verwendung von Entwurfsmustern bringt eine Reihe von Vorteilen mit sich. Muster sind Konzepte, die sich in mehreren Implementierungen bereits bewährten. Standardisierte Muster helfen beim Verständnis eines Softwaresystems

und erleichtern die Kommunikation zwischen den Entwicklern. Die Folgen sind reduzierte Entwicklungskosten und -zeiten. Entwurfsmuster werden beispielsweise bei der Entwicklung von Rahmenwerken eingesetzt.

1.2 Organisationsformen

Ein Rahmenwerk (engl. *Framework*) als Organisationsform ist ein Programmgerüst, das aus mehreren Klassen besteht. Verwendet wird ein solches Rahmenwerk meist durch Anpassung an die jeweilige Anwendungsdomäne mittels Vererbung und durch die Implementierung von Schnittstellen. Die Struktur ist durch finale Methoden vorgegeben. Abstrakte Klassen hingegen müssen implementiert werden. Die hier vorhandenen abstrakten Methoden dienen als Einhängenpunkte (engl. *Hooks*) für die Anwendungsfunktionalität. Weitere Aspekte eines Rahmenwerks sind in Abschnitt 2 auf Seite 19 beschrieben.

1.3 Schichtenarchitektur

Die Schichtenarchitektur eines Softwaresystems zeichnet sich durch Aufteilung nach Aufgabenbereichen in mehrere Ebenen aus. Jede Ebene stellt eine Menge an Diensten bereit, die jeweils die Dienste der unter ihr liegenden Schicht verwenden und für die Dienste der über ihr liegenden Schicht verfeinern. Die einzelnen Schichten werden bei diesem Architekturprinzip voneinander entkoppelt.

Die Vorteile eines geschichteten Softwaresystems liegen in der getrennten Entwicklung der einzelnen Ebenen. Die Schichten bieten Dienste an, welche die Schnittstelle für diese Ebene bilden und von den Implementierungsdetails abstrahieren. Damit realisiert eine Schichtenarchitektur das *Geheimnisprinzip*. Dies ermöglicht den Austausch einzelner Ebenen durch andere Implementierungen, wobei die Benutzer der Dienste der ausgetauschten Schicht nicht beeinflusst werden.

Das Ziel der Schnittstelle zwischen Quest und einem Editor ist die Entkopplung zwischen diesen beiden Software-Teilen. Quest zeichnet sich durch die Bildung der Schichten Modell und View aus. Für die Bereitstellung der Funktionalität eines Editors ist eine Kapselung in einer zusätzlichen Editorschicht naheliegend, die auf die View-Ebene aufsetzt. Diese Schichtung der Ebene Modell, View und Editor kennzeichnet die Architektur des modellbasierten Editor-konzepts.

Die Schichten können unabhängig voneinander entwickelt 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.

In Abbildung 6.1 auf der nächsten Seite sind die drei Ebenen der Anwendung dargestellt. 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. Das AUTOFOCUS-Metamodell ist ausführlich in Abschnitt 1.1 auf Seite 5 beschrieben. Views ergänzen diese Daten durch Darstellungsinformationen, welche Attribute wie die Schriftart, Größe oder Position eines View-Elements beschreiben. Details über einzelne View-Klassen sind in Abschnitt 1.2 erläutert. Die Editorebene visualisiert die Informationen

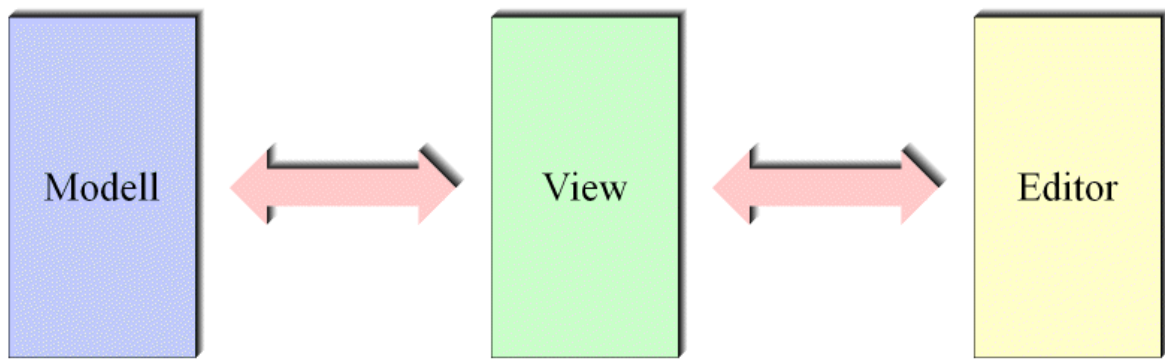


Abbildung 6.1. Überblick über die beteiligten Schichten der Anwendung.

der View-Elemente, erweitert die Anzeigeeinformation mit Editorfunktionalität und realisiert die Benutzerinteraktion. Sie nimmt beispielsweise Ereignisse an der Benutzeroberfläche entgegen und delegiert diese zu ihrer Verarbeitung an die zuständigen Teile des Rahmenwerks.

Eine Ebene ruft ausschließlich die Dienste der unter ihr liegenden Schicht auf, aber nicht umgekehrt. Deshalb erlaubt diese Schichtenarchitektur die Verwendung eines beliebigen Editors. Die View-Schicht bleibt vom eingesetzten Editor unabhängig. Änderungen am AUTO-FOCUS-Metamodell, z.B. die Einführung eines neuen 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. Die Modellebene ist damit unabhängig von der View-Schicht. Die geforderte Unabhängigkeit zwischen den einzelnen Schichten wird auch in der Wahl des Entwurfsmusters für den Kommunikationsvorgang reflektiert.

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 dadurch die Komplexität des Editors erhöht würde. Der Editor soll austauschbar sein, wenn eine neue Version des Editors zur Verfügung steht oder ein gänzlich anderer Editor angekoppelt werden soll. Außerdem enthält die View-Ebene die persistenten Daten. Aus diesen Gründen wird die Trennung zwischen View und Editor in der Architektur der Anwendung reflektiert. Die Schichtenarchitektur zeichnet sich durch die Anhängigkeit des Editors von der View- und nicht auch von der Modellebene aus. Dazu delegiert die View-Ebene die Modelloperationen an das Modell und verbirgt damit die Implementierungsdetails dieser Aktionen für den Benutzer der View-Schnittstelle.

Softwareysteme mit Benutzerinteraktion umfassen allgemein die Bereiche Präsentation, Anwendung und Dialog. Diese Aufteilung unterstützt die Portierbarkeit und die Veränderbarkeit der Anwendung. Das *Seeheim-Modell* [Pfa85] (siehe Abbildung 6.2 auf der nächsten Seite) reflektiert diese Schichtenarchitektur. Die Benutzerschnittstelle wird dabei in die Teile Präsentation, Dialogkontrolle und Anwendungsschnittstelle zerlegt. In der vorliegenden Arbeit wird

das Seeheim-Modell durch die editorspezifische View-Schnittstelle verfeinert, die ein Teil der Anwendungsschnittstelle ist.

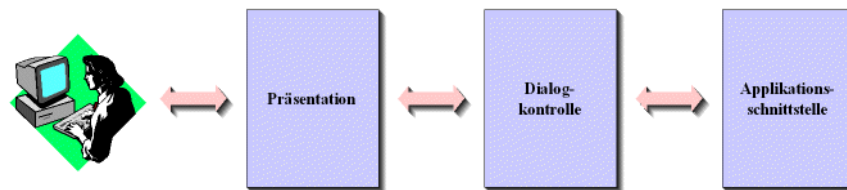


Abbildung 6.2. Die Schichten des Seeheim-Modells.

Ein weiterer Ansatz für die Aufteilung von Software mit Benutzerinteraktion in unterschiedliche Komponenten ist im Paradigma *MVC* (engl. *Model, View, Controller*) [KP88] manifestiert. Dieses Modell wurde zu *PAC* (engl. *Presentation, Application, Control*) [Cou87] weiterentwickelt. Das Seeheim-Modell diente als Basis für die Entwicklung der Modelle Arch bzw. Arch/Slinky [Wor92]. Die Entwicklungszweige des Seeheim- und MVC-Modells sind im Modell *PAC-AMODEUS* (engl. *Assimilating Models of Designers, Users and Systems*) [NC91] zusammengeführt. Alle Modelle zeichnen sich durch die Aufteilung der Architektur in mehrere Teile aus. Diese Vorgehensweise wird bei der Erarbeitung des Editorenkonzepts angewendet und die Architektur in die drei Schichten Modell, View und Editor aufgeteilt.

2 Schnittstelle

Das Ziel dieser Arbeit ist die Entwicklung einer Schnittstelle zwischen Quest und einem Editor. Die Abhängigkeiten zwischen den einzelnen Schichten der Architektur sollen möglichst gering gehalten werden. Im folgenden Abschnitt 2.1 werden die Ziele von Schnittstellen für Programmteile erläutert. Abschnitt 2.2 auf der nächsten Seite stellt den allgemeinen Informationsfluss zwischen den einzelnen Schichten dar. Die Kommunikation zwischen den einzelnen Ebenen wird in Abschnitt 2.3 auf der nächsten Seite verfeinert, um die unterschiedliche Art der Schnittstellendienste zu reflektieren. In den Abschnitten 2.4 auf Seite 52 und 2.5 auf Seite 55 wird die Kopplung zwischen Elementen der Ebenen Modell und View bzw. View und Editor dargestellt.

2.1 Ziele von Schnittstellen zwischen Programmteilen

Die Entwicklung von großen Softwaresystemen erfordert die Reduzierung der Komplexität. Dies wird durch die Strukturierung des Systems in mehrere Programmteile erreicht. Die Komponenten werden getrennt voneinander entwickelt. Die Kommunikation zwischen den Teilen erfolgt über Schnittstellen.

Die Aufgabe von Schnittstellen liegt darin, die Abhängigkeiten zwischen verschiedenen Programmteilen zu minimieren. Solche Schnittstellen führen zu Systemen, die einfacher zu verstehen sind, das *Geheimnisprinzip* realisieren und leichter zu modifizieren sind [Str99]. Schnitt-

stellen sind idealerweise so gestaltet, dass der Informationsfluss zwischen den Programmteilen möglichst zentral an einer Stelle und nicht aufgeteilt auf mehrere Komponenten erfolgt. Dies erleichtert die Wartung und Weiterentwicklung und verbessert die Verständlichkeit eines Programmsystems. Um die Unabhängigkeit der Programmteile voneinander umzusetzen, sind auch die Mechanismen für den Austausch von Daten von einer Schicht zu ihrer übergeordneten so gestaltet, dass sie von Änderungen in der Implementierung anderer Schichten unabhängig sind.

2.2 Informationsfluss der Schnittstelle

Der Informationsfluss zwischen den drei Ebenen Modell, View und Editor ist in Abbildung 6.3 dargestellt. Editoroperationen, die Modell- oder View-Elemente verändern, werden an die View-Ebene delegiert und dort verarbeitet. Dazu ruft die View-Ebene Dienste des Modells auf und wird von der Modellebene über die dortigen Änderungen benachrichtigt. Die View-Ebene wird dem veränderten Modell angepasst. Diese Änderungen der View-Ebene werden an den Editor propagiert, um die Anpassung der angezeigten Editorelemente auszulösen.

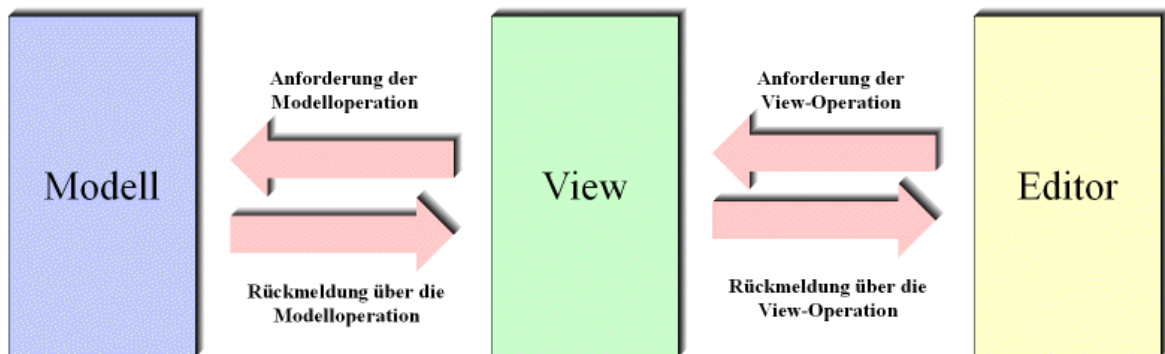


Abbildung 6.3. Der Informationsfluss zwischen den Ebenen Editor, View und Modell erfolgt nur zwischen benachbarten Schichten, und nicht über mehrere Ebenen hinweg.

2.3 Gruppierung der Schnittstellenoperationen

Die Operationen der Schnittstelle werden entsprechend ihrer Art in zwei Kategorien gruppiert. Unterschieden werden Modell- und View-Operationen. Aktionen, die lediglich die Attribute der View-Elemente verändern, werden in der View-Ebene bearbeitet und lösen die Benachrichtigung des Editors zur Aktualisierung der Anzeige aus. Operationen, welche die Eigenschaften des Modells ändern, werden von der View-Ebene an das Modell delegiert. Letzteres benachrichtigt die View-Ebene über die ausgeführten Änderungen. Das veränderte View-Element veranlasst den Editor zur Aktualisierung der Anzeige. Der Vorgang zur Anforderung und Propagierung von Änderungen an Modell- und View-Elementen ist in Abbildung 6.4 auf der nächsten Seite dargestellt.

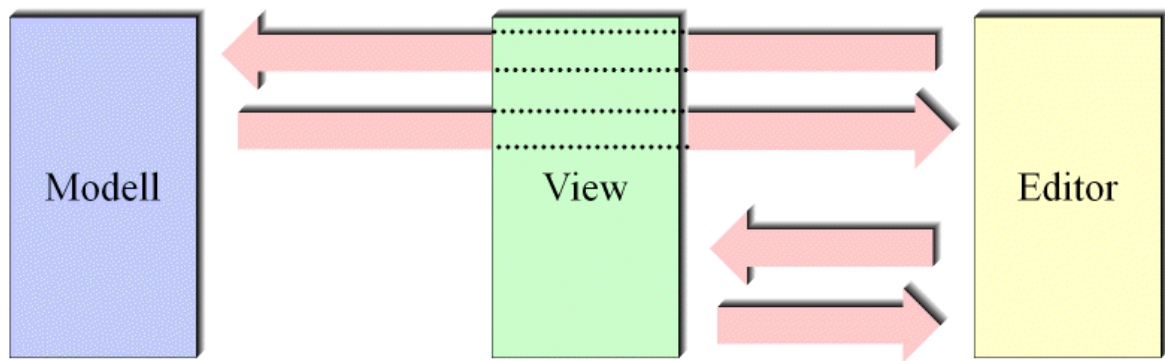


Abbildung 6.4. Der Informationsfluss erfolgt entweder vom Editor über die View-Schicht zum Modell und von dort wieder zurück zum Editor oder nur zur Ebene der View und wieder zurück zum Editor. Aktionen, die Änderungen am Modell ausführen, werden als Modelloperationen bezeichnet. Operationen, die ausschließlich Attribute der View-Ebene verändern, realisieren View-Operationen.

Die Gruppierung nach der Art der Operationen erhöht die Verständlichkeit für die gesamte Anwendung. Denn das Design für die Schnittstelle reflektiert die verschiedenen Arten von Aktionen, die entweder vom Editor bis zum Modell oder nur bis in die Ebene der Views reichen. Das Design der Schnittstelle klassifiziert also Kommunikationsvorgänge nach der Art der Operation.

2.4 Schnittstelle zwischen Modell und View

Veränderungen auf dem Modell werden in Quest durch Ereignisse propagiert. Abschnitt 2.2 auf Seite 12 beschreibt dieses Ereigniskonzept. Eine detaillierte Beschreibung der Anwendung von Ereignissen ist in [PB02b] enthalten.

Neben anderen Anwendungen wie z.B. den Validator [Val02] sind auch Views an Änderungen am Modell interessiert. Um die bestehende Integration anderer Programmsysteme beibehalten zu können, welche auf dem Quest-Modell arbeiten, wird der Ereignis-Mechanismus auch auf die View-Ebene übertragen. Weil Modellelemente nicht nur durch Editoren modifiziert werden, sondern auch durch andere Anwendungen, und diese Änderungen durch Ereignisse angezeigt werden, erfolgt die Kommunikation von Modelländerungen an die Views auch durch Ereignisse vom Typ `ModelChangeEvent`. Um Änderungen am Modell zu verfolgen, registrieren sich View-Elemente bei ihren assoziierten Modellelementen als `ModelChangeListener`. Die Daten auf der View-Ebene sind durch die Propagierung von Modelländerungen an die View-Ebene mit den Daten der Modellebene synchronisiert.

Der generische Ansatz der Weitergabe von Modelländerungen durch Ereignisse ermöglicht die Unabhängigkeit des Modells von seinen registrierten Listnern. Die Art der registrierten Anwendung entscheidet nicht über die Information, die bei der Ereignisverarbeitung an die Listener weitergeleitet wird. Der umgekehrte Informationsfluss von der View-Ebene zum Modell erfolgt über typsichere Methodenaufrufe.

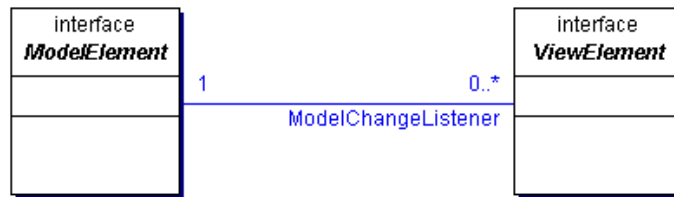


Abbildung 6.5. Ein View-Element ist bei seinem assoziierten Modellelement als Listener vom Typ `ModelChangeListener` registriert, um über Modelländerungen durch die Ereignisse `ModelChangeEvent` benachrichtigt zu werden. Der Mechanismus für die Propagierung von Modelländerungen zu anderen Anwendungen, die auf dem Modell arbeiten und von ihm abhängig sind, bleibt unverändert.

Die Kopplung zwischen Modell- und View-Elementen ist in Abbildung 6.5 dargestellt. Für die Benachrichtigung von Änderungen können sich an einem Modell mehrere View-Elemente, also z.B. mehrere Wurzelobjekte registrieren, die das Bindeglied zwischen Modell und dem Editorfenster realisieren. Ein View-Element assoziiert jedoch genau ein Modellelement.

Das Kommunikationsmuster zwischen Modell- und View-Ebene ist durch den Quest-Ereignis-Mechanismus realisiert. Jedoch können Varianten unterschieden werden, wie der Mechanismus zwischen den beiden Ebenen umgesetzt wird, d.h. welche Elemente die Schnittstelle zwischen Modell und View realisieren. Dazu werden zwei verschiedene Ansätze unterschieden, die in den folgenden beiden Abschnitten erläutert und bewertet werden. Die beiden Ansätze unterscheiden sich darin, dass eine zentrierte Schnittstelle alle Operationen zwischen Modell und View zusammenfasst und eine aufgeteilte Schnittstelle die Kommunikation zwischen den beiden Ebenen in mehrere Teile aufteilt.

2.4.1 Nach Modelloperationen zentrierte Schnittstelle

Die nach Modelloperationen zentrierte Schnittstelle vereinigt alle Modelloperationen zwischen Wurzelobjekt und Modell im Wurzelobjekt. Damit erhöht sich die Anzahl der zu realisierenden Operationen im Wurzelobjekt. Dies schränkt die Strukturierung der Aktionen nach Aufgabenbereichen ein und erschwert die Verständlichkeit der Anwendung. Zudem wird die Komplexität bei der Ereignisverarbeitung von Änderungen am Modell erhöht. Das Wurzelement ist bei allen seinen Modellelementen, d.h. Komponenten, Ports und Kanälen, und bei seiner assoziierten Komponente registriert, um über Modelländerungen benachrichtigt zu werden. Bei der Verarbeitung von Ereignissen, welche die Änderung am Modell anzeigen, muss nun nach der Herkunft des Ereignisses unterschieden werden, welche Aktion in der Editorebene durch die View-Schicht ausgelöst werden soll. Beispielsweise muss bei der Namensänderung einer Komponente erst das geänderte Modellelement betrachtet werden, um die entsprechende Aktion für die Aktualisierung des jeweiligen Editorelements auszuführen. Dazu muss unterschieden werden, ob es sich um die Komponente des Wurzelobjekts oder um eine Unterkomponente handelt. Bei ersterer wird der Editor durch eine Nachricht informiert, die die Aktualisierung der Titelleiste des Editors auslöst. Handelt es sich aber um die andere Art der Komponente, wird der Editor benachrichtigt, um die Beschriftung der Figur für die betreffende Komponente in der Zeichenfläche zu aktualisieren. Als weiteres Beispiel muss beim Einfügen von Ports un-

terschieden werden, welches Modellelement das Ereignis auslöste. Handelt es sich dabei um die Komponente des Wurzelements, muss die View-Ebene den Editor benachrichtigen, um einen Port für die interne Sicht der Komponente einzufügen. Ansonsten wird die Anzeige durch das Einfügen eines Ports für die externe Sicht aktualisiert und damit ein anderes Editorelement eingefügt, welches die Semantik des Ports reflektiert. Diese Art der Schnittstellenimplementierung erfordert Fallunterscheidungen im Wurzelobjekt nach der Herkunft der Ereignisse und erschwert damit die Ereignisverarbeitung im Wurzelement.

2.4.2 Nach Modelloperationen aufgeteilte Schnittstelle

Bei der Realisierung durch eine aufgeteilte Schnittstelle registriert sich jedes View-Element ausschließlich bei seinem assoziierten Modellelement. Dabei vereinigt eine Schnittstelle die Modelloperationen, die nach dem AUTOFOCUS-Metamodell mit dem betreffenden Modellelement verbunden sind. Der relevante Teil aus dem Metamodell ist in Abbildung 2.3 auf Seite 9 dargestellt.

Beispielsweise ist eine Unterkomponente 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 Port-Operationen der internen Sicht. Eine Oberkomponente aggregiert alle Kanäle ihrer Unterkomponenten. Deshalb werden die Operationen auf Kanälen über das Wurzelobjekt angestoßen.

Modelländerungen, welche die Assoziationen zwischen Modellelementen betreffen, werden im Wurzelobjekt bearbeitet. Änderungen an den Modellattributen werden von dem View-Element von der Modellebene angefordert, welches das betreffende Modellelement referenziert. Obwohl die Schnittstelle damit in mehrere Einheiten aufgeteilt ist, bringt diese Variante den folgenden Vorteil mit sich: Die Fallunterscheidungen wie in der ersten Variante sind bei diesem Design nicht nötig, da die Elemente aus der View-Ebene nur bei ihren assoziierten Modellelementen registriert sind.

Zusammenfassend lässt sich feststellen, dass die Realisierung durch eine zentrierte Schnittstelle mit dem Nachteil der zu realisierenden Fallunterscheidungen im Wurzelobjekt verbunden ist. Letztere sind bei der Variante der aufgeteilten Schnittstelle nicht erforderlich. Deshalb wird diese Variante für das Design der Schnittstelle gewählt, bei welcher die Modelloperationen von mehreren Elementen verarbeitet werden.

Nachdem die Varianten für die Realisierung der Schnittstelle zwischen Modell- und View-Ebene dargestellt und bewertet sind, folgt in den nächsten Abschnitten die Erarbeitung der Schnittstelle zwischen View- und Editorebene. Nach der Vorstellung der prinzipiellen Kopplung zwischen einem View- und einem Editorelement werden auch hier Variationen der Realisierung erläutert und bewertet.

2.5 Schnittstelle zwischen View und Editor

Für den Informationsfluss zwischen der View- und Editorebene ist der Einsatz des Entwurfsmusters *Observer* [EG95] sinnvoll. Dieses Muster basiert auf dem in [AG89] vorgestellten Smalltalk-80 change/update-Mechanismus. Die statische Struktur des Observer-Musters ist nach [OO] in Abbildung 6.6 dargestellt. Das Subjekt besitzt eine Liste aller registrierten Observer und die Daten, für die sich die Observer interessieren. Die Bedeutung der Methoden werden bei der Darstellung einer typischen Ablaufstruktur dargestellt. Die konkreten Observer sind von der Klasse *Observer* abgeleitet und implementieren die spezielle Funktionalität zur Erfüllung der gestellten Anforderungen.

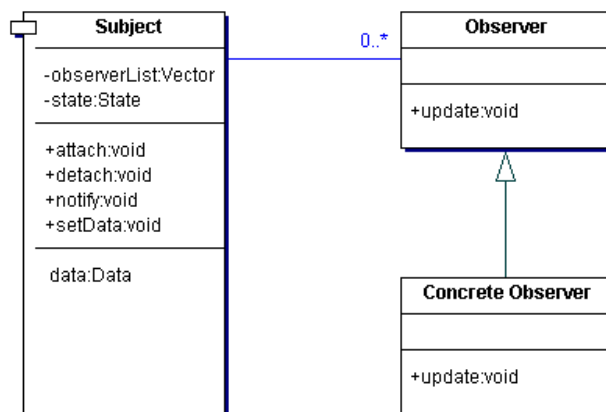


Abbildung 6.6. Statische Struktur des Observer-Entwurfsmusters.

Der Ablauf eines typischen Interaktionsmusters zwischen Observer und Subjekt ist in Abbildung 6.7 auf der nächsten Seite nach [OO] dargestellt. Ein Observer registriert sich bei einem Subjekt. Sobald der Zustand des Subjekts durch den Methodenaufruf `setData()` 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 Verfügung gestellt. Der Observer reagiert auf die geänderten Daten im Subjekt. Besteht kein Interesse an Datenänderungen mehr, erfolgt die Deregistrierung durch die Methode `detach()`.

Das Subjekt bleibt beim Einsatz dieses Entwurfsmusters von seinen registrierten Objekten unabhängig, weil letztere die Bearbeitung der Aktualisierung selbst vornehmen. Für das in der vorliegenden Arbeit beschriebene 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. Dieser Zusammenhang zwischen View- und Editorelementen ist in Abbildung 6.8 auf der nächsten Seite dargestellt. Bei der Anwendung des Observer-Entwurfsmusters bleibt die View-Ebene von der des Editors unabhängig. Änderungen am Rahmenwerk zur Realisierung der Benutzerschnittstelle haben deshalb keine Auswirkungen auf die gesamte Anwendung. Bei der Realisierung der Schnittstelle zwischen View und Editor gibt es zwei Alternativen, die in den beiden folgenden Abschnitten erläutert und bewertet werden.

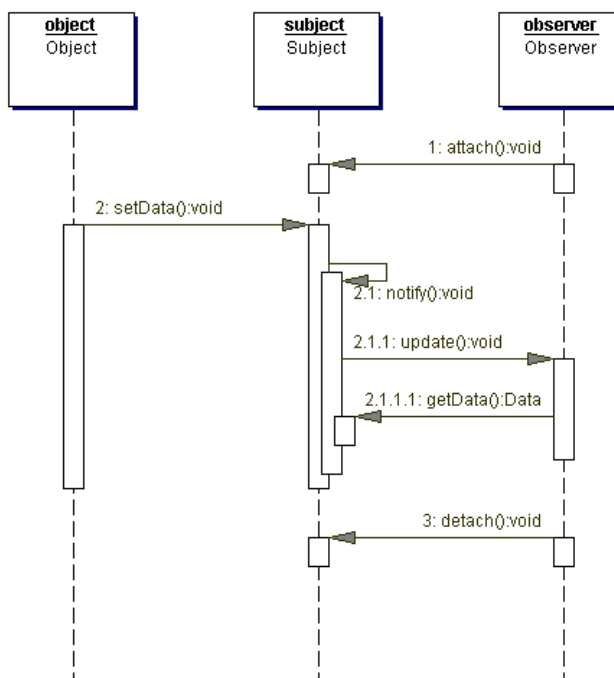


Abbildung 6.7. Eine typische Interaktionsfolge bei der Anwendung des Observer-Entwurfsmusters.

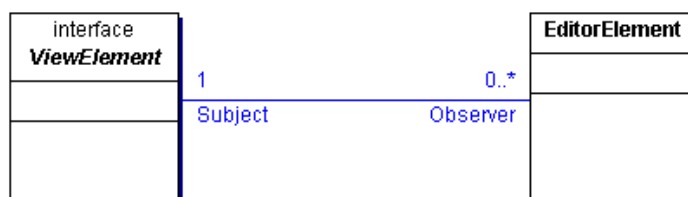


Abbildung 6.8. Ein Editorelement ist bei seinem assoziierten View-Element als Observer registriert. Das View-Element benachrichtigt bei Datenänderungen als Subjekt die bei ihm registrierten Objekte.

2.5.1 Zentrale Editorschnittstelle

Die zentrale Editorschnittstelle vereinigt die Editoroperationen aller Diagrammelemente im Observer des Wurzelobjekts. Dabei ist ein Editorfenster als Observer bei allen View-Elementen des Wurzelobjekts registriert. Deshalb erfolgt die Aktualisierung der Anzeige stets über den Aufruf der Aktualisierungsmethode im Editor. Dabei muss durch Fallunterscheidungen entschieden werden, welches Element betroffen und welche Operation darauf durchzuführen ist. Die Schnittstelle zwischen View- und Editorebene ist zwar auf ein Element zentriert, die Anzahl der Fallunterscheidungen und der Umfang der Methode für die Behandlung der Aktualisierung steigen aber an und erschweren die Verständlichkeit des Aktualisierungsvorgangs.

2.5.2 Aufgeteilte Editorschnittstelle

Elemente der Editorebene besitzen in der View-Ebene Entsprechungen. Das Editorfenster entspricht beispielsweise dem Wurzelobjekt. Alle Operationen, die in den View-Elementen implementiert sind, finden bei der Realisierung durch eine aufgeteilte Editorschnittstelle ihre Entsprechung in den Editorelementen. Dabei implementieren Editorelemente diejenigen Operationen, die durch ihre assoziierten View-Elemente realisiert werden. Die Schnittstellen in der View-Ebene finden damit ihre Entsprechungen in der Editorebene. Die Kommunikation zwischen Editor und View findet zwar nicht mehr zentriert an einer Stelle statt, sondern auf mehrere Elemente aufgeteilt, diese Aufteilung reflektiert jedoch die Unterscheidung nach View- und Modelloperationen. Dadurch wird die Verständlichkeit der gesamten Anwendung erhöht. Deshalb realisiert das erarbeitete Editorenkonzept eine auf mehrere Einheiten aufgeteilte Editorschnittstelle.

3 Aufteilung der Steuerung

Für eine Aufteilung der Steuerung der Anwendung unterscheidet man mehrere Ansätze. Ausgangspunkt sind die beiden Realisierungen, bei der entweder alle Operationen vom Modell- oder der View-Ebene gesteuert werden, und bei welcher der Editorschicht ein umfangreicherer Aktionsspielraum zugeordnet wird. Ausgehend von diesen sehr verschiedenen Ansätzen sind auch Kompromisslösungen denkbar.

3.1 Steuerung durch Modell- und View-Ebene

Die in den Abschnitten 2 auf Seite 50 und 1 auf Seite 47 beschriebene Architektur erlaubt die ausschließliche Steuerung des Editors 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 sind, werden auch im Editor reflektiert. Alle anderen Aktionen werden im Editor nicht angezeigt.

Die Steuerung durch das Modell delegiert die Aufgaben der Ausführung von Operationen und der Einhaltung von Konsistenzbedingungen an die Modellschicht. Die Bearbeitung von Situationen, bei denen 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 Kom-

ponente löschen, während ein anderer einen neuen Port einfügen möchte, können inkonsistente Zustände auftreten, falls die Aktionen nicht zentral im Modell ausgeführt werden. Mechanismen zur Synchronisation von Mehrprogramm- und Mehrbenutzerbetrieb sind in Kapitel 8 auf Seite 103 beschrieben.

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 diese Operation einen Layout-Algorithmus in der View-Ebene verhindert wird. Die Implementierung solcher Einschränkungen erfolgt zentral in der Ebene der Views. Um dies zu realisieren, muss die bereits vom Editor-Rahmenwerk angebotene Funktionalität für die Änderung von grafischen Attributen extrahiert werden und in die Gesamtanwendung integriert werden. Damit sind Änderungen an der Implementierung des Rahmenwerks nötig, welche die Funktionalität aus dem Rahmenwerk extrahiert und an die gewünschten Anforderungen anpasst.

Der Editor wird bei diesem Ansatz zu einer Visualisierungskomponente ohne semantische Information über die anderen beteiligten Elemente. Die Schichtung Modell, View und Editor gelingt bei diesem Vorgehen entsprechend der Aufgaben der einzelnen Schichten. Änderungen an einzelnen Ebenen sind auf die Anpassung der Zugriffsstrukturen der anderen Schichten beschränkt, sie haben aber keine Auswirkungen auf die gesamte Anwendung. Die Steuerung von Aktionen erfolgt zentral in den betreffenden Schichten und ist nicht auf mehrere Einheiten aufgeteilt. Bei der Aktualisierung der Zeichenfläche im Editor sind feingranulare Aktualisierungsmethoden erforderlich, welche die gezielte Veränderung nur der betreffenden grafischen Objekte entsprechend den Änderungen in der View- bzw. Modellebene ausführen.

3.2 Steuerung durch die Editorebene

Bei der Verlagerung der Steuerung von Operationen in die Editorebene erhält jeder Benutzer für ein Editorfenster ein eigenes Wurzelobjekt einschließlich eigener View-Elemente. Somit erfolgt die Synchronisation zwischen mehreren Benutzern oder Programmen lediglich über das Modell und nicht gleichzeitig über die View-Ebene. Bei diesem Ansatz liegt die Steuerung von Operationen auf dem Modell oder der View beim Editor. Die Operationen des Benutzers werden ausgeführt, und zu bestimmten Zeitpunkten mit dem Modell synchronisiert. Treten dabei Inkonsistenzen auf, werden alle seit dem letzten erfolgreichen Einbringen der Änderungen in das Modell durchgeführten Aktionen rückgängig gemacht und das gesamte Editorfenster verworfen und entsprechend der aktuellen Konstellation des Modells aktualisiert. Dazu sind im Gegensatz zu der Steuerung des Editors durch die View-Ebene Aktualisierungsmethoden für den Editor mit grober Granularität erforderlich. Die Aktualisierung der Zeichenfläche erfolgt nicht inkrementell, sondern wird in ihrer Gesamtheit entsprechend der aktuellen Zusammensetzung des Modells ausgeführt, sobald Inkonsistenzen festgestellt werden.

Bei diesem Ansatz wird die gesamte Funktionalität des Editor-Rahmenwerks verwendet. Der Einsatz des Rahmenwerkes erfolgt durch die Anpassung an die Anwendung mittels Vererbung. Die Implementierung von Aktionen auf der Modell- oder View-Ebene ist im Editor realisiert. Die Funktionalität des Editors wird dadurch umfangreich. Dies verschlechtert die Verständlichkeit der gesamten Anwendung. Bei der Erweiterung des Modells sind entsprechende, ggf. umfangreiche Anpassungen in der Editorebene notwendig. Die Schichtenarchitektur wird durch diesen Ansatz jedoch nur ansatzweise unterstützt. Denn die Ausführung von Modell- und

View-Operationen wird teilweise in die Editorebene verlagert.

3.3 Mischform der Steuerung

Bei der Aufteilung der Steuerung von Operationen zwischen der Editor- und der Modell- bzw. View-Ebene werden unterschiedliche Arten von Aktionen auf unterschiedliche Ebenen verteilt. Die Modelloperationen werden durch die Modellebene gesteuert, die Aktionen, welche die Attribute von View-Elementen verändern, werden in der Editorebene realisiert. Dabei ist die exklusive Benutzung des View-Elements durch einen Anwender vorausgesetzt, das vom Editorfenster referenziert wird. Somit werden die Änderungen an Anzeigeattributen durch das Editorenrahmenwerk realisiert und in die View-Ebene propagiert. Die Funktionalität des Rahmenwerks kann in seiner Gesamtheit für die Realisierung der Anwendung eingesetzt werden. Die Operationen auf dem Modell werden durch das Modell gesteuert und die Änderungen von dort an die anderen Ebenen propagiert. Inkonsistenzen des Systemmodells treten nicht auf, für die Aktualisierung der Zeichenfläche des Editors sind freigranulare Aktualisierungsmethoden nötig. Bei Änderungen am Modell wird die Zeichenfläche entsprechend den Änderungen angepasst.

Diese Mischform bietet also die zentrale Steuerung durch das Modell. Inkonsistenzen werden dort behandelt. Erweiterungen am Modell verursachen Anpassungen der Aufrufstrukturen zwischen den Schichten. Die Anpassungen, die dazu für die View-Ebene nötig sind, werden in den Views realisiert, und die Änderungen, welche die Benutzerinteraktion betreffen, werden in der Schicht des Editors implementiert. Aufgaben der View-Schicht werden in die Ebene des Editors verlagert.

In dieser Arbeit wird der Ansatz mit der Steuerung des Editors durch die View- bzw. Modellebene realisiert. Denn das Ziel ist die Entkopplung der Schichten voneinander. Dies wird durch eine strikte Schichtenarchitektur erreicht, welche die Ebenen nach ihrer Aufgabenstellung und den dort gespeicherten Daten aufteilt. Bei Mischformen oder der Steuerung durch den Editor ist diese Aufteilung nicht mehr so deutlich.

4 Darstellung der gewählten Kommunikation zwischen den Schichten

Für den Entwurf der Schnittstellen zwischen Modell und View bzw. View und Editor und der Art des Informationsaustausches zwischen den Schichten sind in den vorhergehenden Abschnitten unterschiedliche Lösungsansätze erläutert und bewertet. Das gewählte Design für den Austausch von Daten zwischen den Schichten wird nun durch Sequenzdiagramme detaillierter beschrieben. Dabei wird unterschieden, ob es sich um eine Modell- oder View-Operation handelt. In den beiden folgenden Abschnitten wird das Prinzip der Kommunikation zwischen den Ebenen für den Austausch von Daten für Modell- und View-Operationen dargestellt. Die Bezeichnungen für die beteiligten Klassen und Methoden reflektieren nur Ideen der Realisierung und nicht die eigentliche Implementierung.

4.1 Ablaufstruktur bei View-Operationen

In Abbildung 6.9 ist beispielhaft die Ablaufstruktur bei einer Ausführung der View-Operation dargestellt, die das Attribut der Position eines View-Elements verändert. Andere View-Operationen besitzen eine ähnliche Aufrufstruktur. Durch die Benutzerinteraktion wird die Kommunikation mit der View-Ebene angefordert. Dazu wird die Methode `moveFigure()` der Editorebene aufgerufen, um eine Veränderung der Position zu veranlassen. Die Interaktion des Benutzers wird in den Methodenaufruf `changePosition()` der View-Schnittstelle umgesetzt. In der View-Ebene erfolgt die Behandlung der Anforderung, indem das Attribut für die Position entsprechend des Benutzerwunsches durch die Methode `setPositionAttribute()` neu belegt wird. In der Methode `notifyObjects()` wird die Benachrichtigung aller registrierten Objekte angestoßen. Im betrachteten Fall wird der Editor über die Editorschnittstelle durch den Methodenaufruf `updateDisplay()` über die Änderung der Eigenschaft in der View-Ebene benachrichtigt. Die Anzeige wird im Editor so angepasst, dass sie die aktuelle Belegung des Attributs für die Position reflektiert.

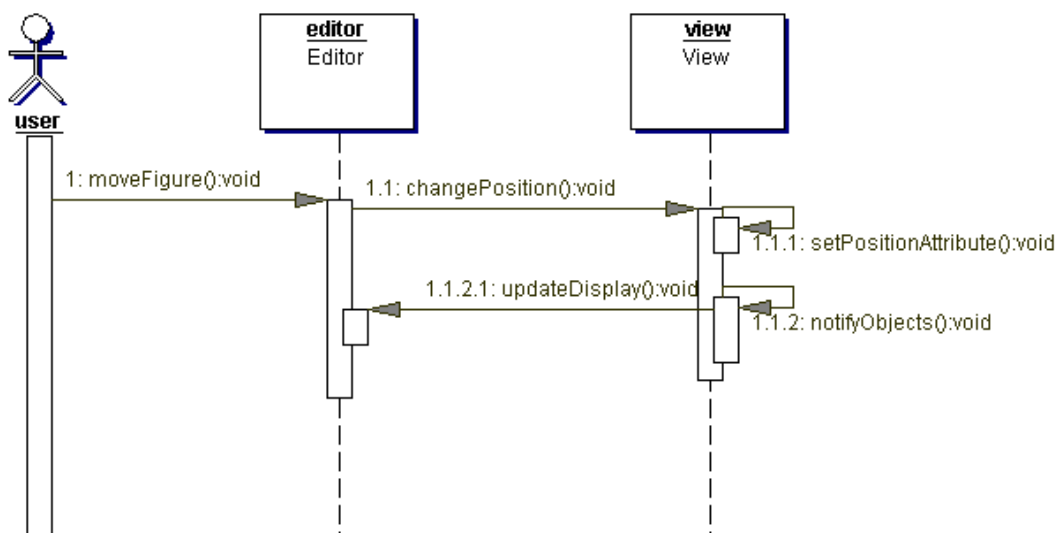


Abbildung 6.9. Bei der Ausführung von View-Operationen erfolgt die Kommunikation ausschließlich zwischen den beiden Ebenen Editor und View.

4.2 Ablaufstruktur bei Modelloperationen

Bei der Änderung von Modelleigenschaften sind die drei Ebenen Editor, View und Modell beteiligt. Die Ablaufstruktur bei der Durchführung von Modelloperationen ist in Abbildung 6.10 auf der nächsten Seite dargestellt. Betrachtet wird beispielhaft die Ausführung einer Modelloperation, die ein neues Modellelement einfügt. Andere Modelloperationen besitzen eine ähnliche Aufrufstruktur. Ausgangspunkt ist die Benutzerinteraktion, die durch die Methode `createModelElement()` in der Editorebene behandelt wird. Die Methode `insertModelElement()`

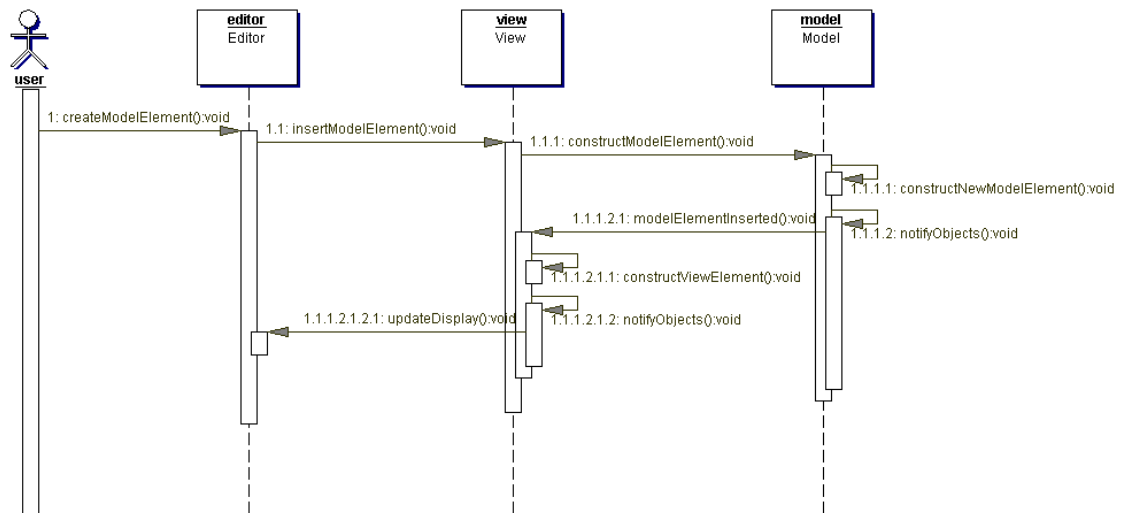


Abbildung 6.10. Bei der Ausführung von Modelloperationen werden Änderungen an Elementen der Ebenen Modell, View und Editor durchgeführt.

wird als Teil der View-Schnittstelle aufgerufen, um die gewünschte Operation anzufordern. Zur Konstruktion des Modellelements wird aus der Modellschnittstelle die Methode `constructModelElement()` aufgerufen. Die Methode `constructNewModelElement()` erzeugt ein neues Modellobjekt. In der Methode `notifyObjects()` werden alle registrierten Objekte über eine Änderung am Modell benachrichtigt. Die Methode `modelElementInserted()` wird als Teil der View-Schnittstelle aufgerufen, um die Änderung am Modell zu behandeln. In der Methode `constructViewElement()` wird für das neue Modellelement ein View-Element konstruiert. Um die Benachrichtigung der registrierten Objekte der View-Ebene auszulösen, wird die Methode `notifyObjects()` aufgerufen. Als Teil der Editorschnittstelle wird die Methode `updateDisplay()` aufgerufen, um die Anzeige entsprechend der aktuellen Zusammensetzung der View-Ebene anzupassen.

5 Bestimmung von Entwurfsentscheidungen durch externe Vorgaben

Die Vorgaben von Quest beeinflussen den Entwurf der Erweiterungen für das modellbasierte Editorenkonzept. In Quest werden die beiden Ebenen Modell und View unterschieden. Die Erweiterung der Schichtenarchitektur durch die dritte Ebene des Editors wird durch die Architektur von Quest vorgegeben. Dennoch ist diese Entwurfsentscheidung auch durch die allgemeinen Prinzipien des Softwareentwurfs getragen. Aus der Schichtenarchitektur folgt die Forderung nach Schnittstellen, welche die Kommunikation zwischen den einzelnen Ebenen realisieren.

Das gewählte Rahmenwerk beeinflusst den Entwurf der Anwendung nicht, es erleichtert jedoch die Anbindung des Editors. Die Schnittstelle zwischen View und Editor ist so realisiert, dass auch beliebige andere Editoren angekoppelt werden können. Zwischen View und Editor

werden Änderungen durch die Kommunikation nach dem Observer-Entwurfsmuster weitergegeben. Ist die Unterstützung der Kommunikation nach dem Observer-Entwurfsmuster in einem Rahmenwerk möglich, kann es als Basis für einen Editor eingesetzt werden. Das als Basis für den prototypischen Editor verwendete Rahmenwerk erlaubt die Umsetzung von Benutzerinteraktionen in Methodenaufrufe der View-Schnittstelle. Die Trennung zwischen Editor und View ist damit sehr deutlich. Dadurch wird die Ankopplung auch eines anderen Editors erleichtert.

Implementierung

Die Architektur der gesamten Anwendung zeichnet sich durch die Schichtenbildung Modell, View und Editor aus. Objekte der Modellebene repräsentieren Elemente aus dem AUTOFOCUS-Metamodell. In der vorliegenden Arbeit sind die folgenden Elemente von Bedeutung:

- Component
- Port
- Channel

Unter View-Elementen werden die Elemente der View-Ebene verstanden, welche die Modellelemente um die Anzeigeinformation erweitern. Die für diese Arbeit relevanten Elemente sind die folgenden:

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

Ein Element aus der Editorebene visualisiert ein View-Element an der Benutzerschnittstelle. In dieser Kategorie sind die grafischen Figuren und die gesamte Zeichenfläche des Editors zusammengefasst.

In den folgenden Abschnitten wird die Umsetzung des Designs in die Implementierung beschrieben. Dabei werden auch alternative Implementierungsansätze erläutert und bewertet.

1 Kommunikation zwischen den Ebenen Modell, View und Editor

1.1 Kopplung der Modell-, View- und Editorelemente

Aus den in Kapitel 6 auf Seite 47 beschriebenen Gründen werden Modell- und View-Elemente durch einen Listener vom Typ `ModelChangeListener` miteinander verknüpft. Ein Editorelement registriert sich als Observer bei einem View-Element, welches die Rolle eines Subjekts realisiert. Diese Kopplung der drei Ebenen ist in Abbildung 7.1 dargestellt.

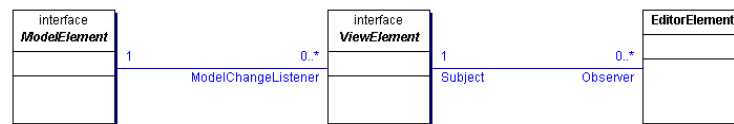


Abbildung 7.1. Die Elemente aus den Ebenen Modell, View und Editor sind als `ModelChangeListener` bzw. `Observer` miteinander verknüpft. Ein View-Element ist bei genau einem Modellelement registriert. Ein Element aus der Editorebene ist bei genau einem View-Element angemeldet. Bei einem Modell- oder View-Element können sich beliebig viele Elemente registrieren.

Ein Editorelement, d.h. ein Editorfenster oder eine Figur zur Repräsentierung von View-Elementen, ist mit seinem entsprechenden View-Element durch das Observer-Entwurfsmuster assoziiert. Sobald ein Observer bei seinem Subjekt registriert ist, meldet sich das Subjekt bei seinem assoziierten Modellelement als ein Listener vom Typ `ModelChangeListener` an. Durch diese Kopplung zwischen den Elementen der Ebenen Modell, View und Editor werden Änderungen am Modell in die View-Ebene und von dort in die Editorschicht propagiert. Die Editorelemente werden über die Änderungen an den View-Elementen benachrichtigt.

1.2 Anforderung und Propagierung von Änderungen zwischen den Ebenen Modell, View und Editor

Das zentrale Systemmodell realisiert die Konsistenzhaltung der Daten in der Modellebene. Änderungen an den Modell- oder View-Eigenschaften werden durch den Benutzer eines Editors oder eine andere Anwendung vorgenommen. Entsprechend der Unterscheidung nach der Art der Operation werden auch zwei Kommunikationswege zwischen den Ebenen Modell, View und Editor unterschieden:

Durch den Editor werden Modelloperationen über typsichere Methodenaufrufe von der Editor- an die View-Ebene weitergeleitet. Es folgt die Ausführung der Modelloperationen durch die View-Ebene. Die Änderungen am Modell werden an die registrierten Listener durch Ereignisse vom Typ `ModelChangeEvent` weitergegeben.

Werden vom Benutzer die Anzeigeeigenschaften wie die Position oder die Größe einer Komponente verändert, erfolgen diese Änderungen zunächst an den mit den grafischen Objekten des Editors assoziierten View-Elementen. Von dort aus werden die Änderungen wiederum an die Editorelemente weitergegeben.

Bei der Änderung von Eigenschaften eines Modellelements durch eine andere Anwendung werden Ereignisse ausgelöst, welche die registrierten View-Elemente über die Änderungen informieren. Die daran anschließende Ereignisverarbeitung in der View-Ebene schließt die Aktualisierung der Observer-Objekte der Editorebene ein. Die Benutzerschnittstelle wird so aktualisiert, dass sie die Änderungen am Modell reflektiert. Veränderungen an den Eigenschaften eines View-Elements durch eine andere Anwendung werden ebenso an alle Observer des betreffenden Elements weitergeleitet. Änderungen an View- oder Modelleigenschaften werden also unabhängig vom Aufrufer der Operation an den Editor propagiert.

2 Erweiterungen an den View-Elementen

Die Realisierung der Kopplung zwischen den Ebenen Modell, View und Editor ist in Java-Schnittstellen formuliert. Dies erleichtert das Auffinden der Änderungen an Quest, die für das modellbasierte Editorenkonzept ergänzt sind. Die Java-Schnittstellen sind nach Modell-, View- und zusätzlichen Hilfsfunktionen gruppiert. Methoden der Schnittstellengruppe `View2Model` zur Veränderung von Modelleigenschaften werden von der Editorebene auf Objekten der View-Schicht aufgerufen. Die Änderungen am Modell werden durch die Methoden der Schnittstelle `ModelChangeListener`, die durch die Schnittstellengruppe `View2Model` erweitert wird, vom Modell an die View-Ebene propagiert. Zur Veränderung der Belegungen von View-Attributen werden die Methoden der Schnittstellengruppe `ViewOnlyOperations` von der Editorebene aufgerufen. Die View-Schicht benachrichtigt die Editorelemente durch einen Methodenaufruf aus der Schnittstelle `View2GUI`, die lediglich die Schnittstelle `Observer` erweitert. Zusammenfassend sind in Tabelle 7.1 die Schnittstellen und ihre implementierenden Klassen der View-Ebene beschrieben.

View-Klasse	Modelloperationen	View-Operationen	Hilfsfunktionen
<code>SSDComponentRootView</code>	<code>SSDRootView2Model</code>	–	–
<code>SSDComponentView</code>	<code>SSDComponentView2Model</code>	<code>SSDComponentViewOnlyOperations</code>	<code>SSDComponentViewHelpers</code>
<code>SSDPortView</code>	<code>SSDPortView2Model</code>	<code>SSDPortViewOnlyOperations</code>	<code>SSDPortViewHelpers</code>
<code>SSDChannelView</code>	<code>SSDChannelView2Model</code>	–	–

Tabelle 7.1. Überblick über die Schnittstellen und ihre implementierenden Klassen.

2.1 Modelloperationen

Die Erweiterungen für die Ausführung von Modelloperationen und die Propagierung der Modelländerungen sind in Abbildung 7.2 auf der nächsten Seite dargestellt. Jede Schnittstelle erweitert die Schnittstelle `ModelChangeListener`. Die Methoden für Modelloperationen, die vom Editor als Teil der View-Schnittstelle aufgerufen werden, sind in Abbildung 7.2 auf der nächsten Seite dargestellt.

Die Ablaufstruktur bei Modelloperationen ist bei allen möglichen Operationen ähnlich. Deshalb werden in den folgenden Abschnitten beispielhaft einige Abläufe von typischen Modelloperationen erläutert.

2.1.1 Einfügen einer Komponente

Im Folgenden wird das Einfügen einer neuen Komponente durch die Initiierung des Benutzers des Editors beschrieben. Der Ablauf der Schritte von der Benutzeraktion bis zum Aufruf der Benachrichtigung der Modell-Listener ist in Abbildung 7.3 auf Seite 67 beschrieben. Der Benutzer lässt im Editormodus `ModePlaceComponent` für das Einfügen einer neuen Komponente die Maustaste los (`mouseReleased()`). Das Eigenschaftsfenster `ComponentDialog` für eine Komponente wird konstruiert und durch die Methode `setVisible()` angezeigt. Die Daten für die Erstellung einer neuen Komponente werden aus der Eingabemaske extrahiert (`getComment()`, `getComponentName()`). Alternativ würde statt eines Eingabefensters eine

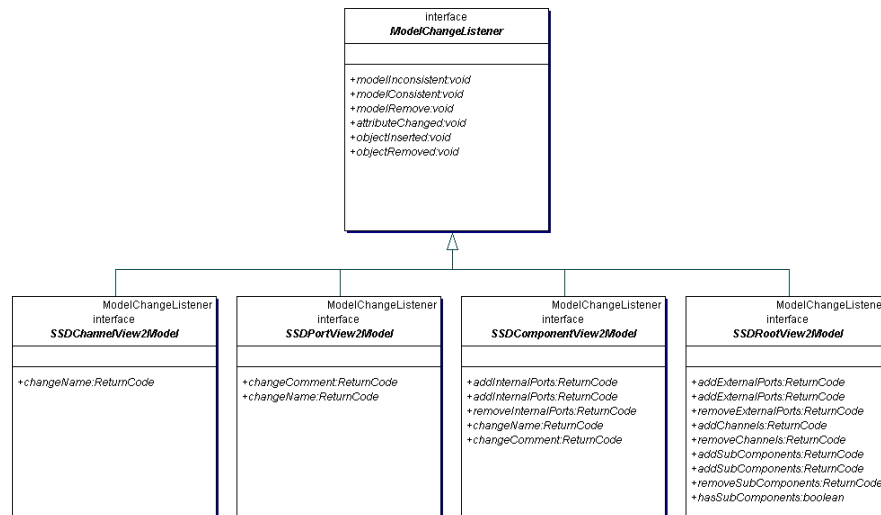


Abbildung 7.2. Die Erweiterungen an den View-Klassen für Modelloperationen sind in Schnittstellen zusammengefasst.

neue Komponente mit vorbelegten Eigenschaften gezeichnet. Die Attribute der Komponente würden vom Benutzer nachträglich angepasst. Nach den Benutzereingaben erfolgt der Aufruf der Methode `addSubComponents()` auf dem Wurzelobjekt `SSDComponentRootView` zum Einfügen einer neuen Komponente, die als Parameter die Eigenschaften der neuen Komponente erhält, also den Namen und eventuell einen zusätzlichen Kommentar. Die aufgerufene Methode ist Teil der Schnittstelle `SSDRootView2Model`. Im Wurzelobjekt wird durch die Methode `getModel()` die referenzierte Komponente `parent` bereitgestellt. Im Wurzelobjekt wird die neue Unterkomponente `component` angelegt und ihre Attribute mit den aktuellen Parametern belegt. Sobald die neue Komponente durch den Aufruf der Methode `addSubComponents()` auf ihrer Oberkomponente `parent` hinzugefügt ist, wird ein Ereignis konstruiert, das die Modelländerung anzeigt. Auf dem Modellelement der Oberkomponente erfolgt der Aufruf der Methode `fireObjectInserted()`, um die Benachrichtigung der Listener der Oberkomponente anzustoßen. Dadurch wird auch auf der Klasse `SSDComponentRootView` die Methode `objectInserted()` aufgerufen. Diese Methode ist Teil der Schnittstelle `SSDRootView2Model` und wird von der Schnittstelle `ModelChangeListener` vererbt.

Die folgenden Schritte sind in Abbildung 7.4 auf Seite 68 dargestellt. In der Ereignisbehandlung in der Methode `objectInserted()` der Klasse `SSDComponentRootView` wird ein View-Element aufgebaut, das dem Modellelement aus dem übertragenen Ereignis entspricht und dessen Attribute mit dem Aufruf der Methode `initialize()` mit Standardwerten belegt werden. Die Methode `checkView()` überprüft fehlende oder überflüssige View-Elemente des Wurzelobjekts und ergänzt oder entfernt diese entsprechend der aktuellen Konstellation des Modells. Das neu erzeugte Objekt vom Typ `Action` kapselt das neue View-Element und einen Identifikator für die durchzuführende Aktion im Editor als Observer des Wurzelobjekts. Die Klasse `Action` ist in Abschnitt 4.2 auf Seite 82 beschrieben. Die Methode `notifyObservers()` aus der Schnittstelle `Subject` ruft die `update()`-Methoden der Observer des Wurzelobjekts auf. Den Methoden werden als aktuelle Parameter die Referenzen auf das aufrufende

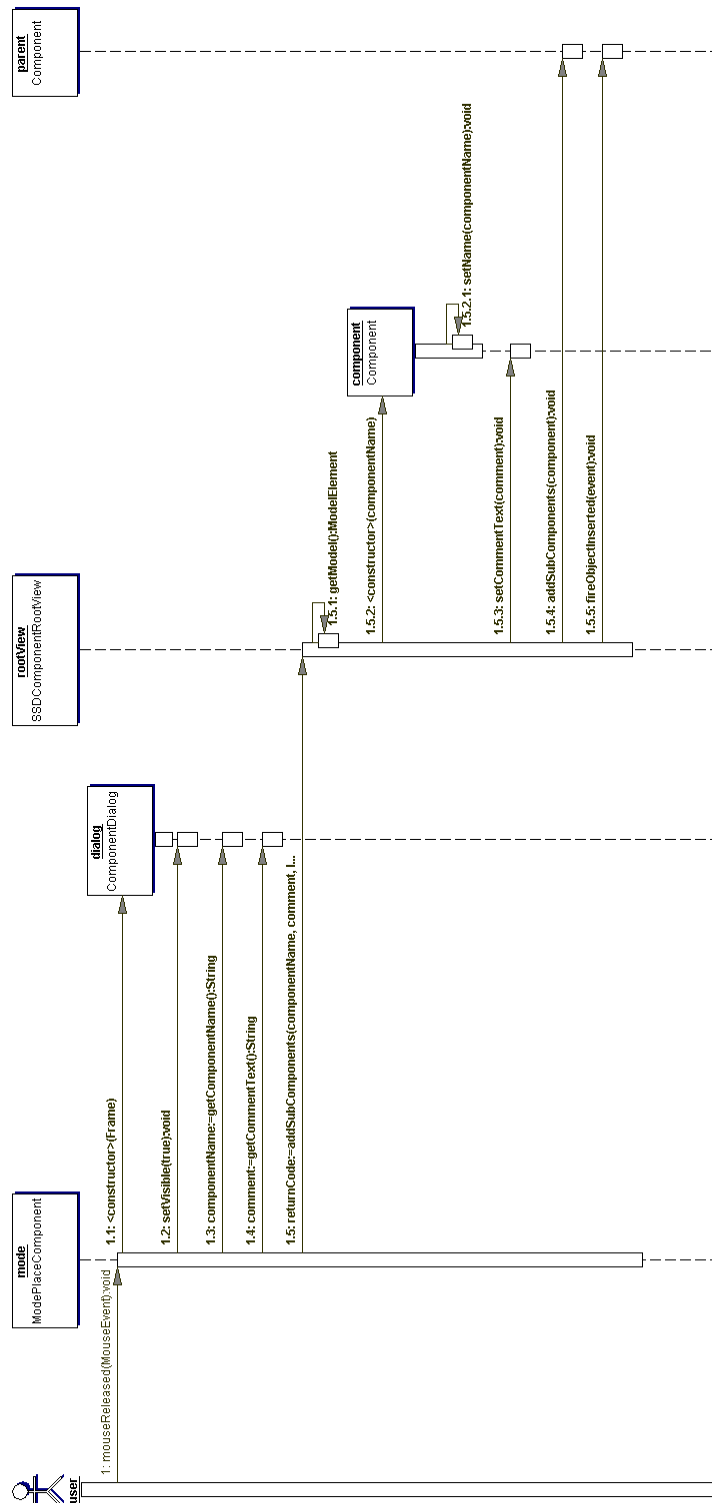


Abbildung 7.3. Der Ablauf beim Einfügen von neuen Komponente von der Benutzerinteraktion bis zum Aufruf der Benachrichtigung der Modell-Listener.

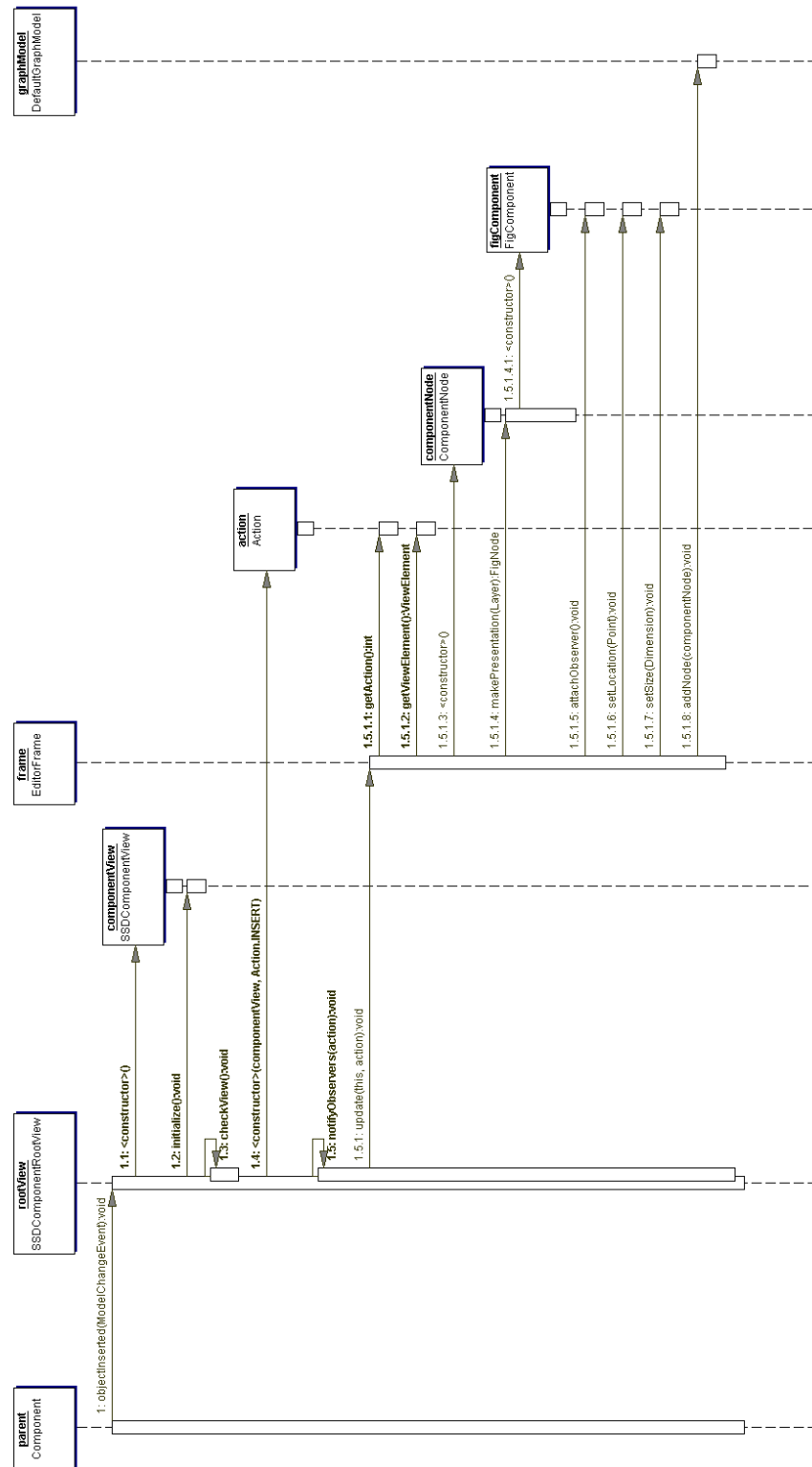


Abbildung 7.4. Der Ablauf beim Einfügen von neuen Komponente vom Aufruf zur Benachrichtigung der Modell-Listener bis zur Aktualisierung der Anzeige im Editor.

Objekt und ein Objekt vom Typ `Action` übergeben. Dabei wird auch die Methode `update()` des Editors aufgerufen, welche ein Teil der Schnittstelle `View2GUI` ist. In der `update()`-Methode der Klasse `EditorFrame` wird die Aktualisierung der Zeichenfläche durchgeführt. Dem übergebenen `Action`-Objekt wird mit den Methoden `getAction()` und `getViewElement()` die durchzuführende Operation und das betreffende View-Element entnommen. Bei einer Einfügeoperation wird die Methode `addNode()` aufgerufen. In ihr werden ein Objekt vom Typ `ComponentNode` für das Graphenmodell des Editoren-Rahmenwerks konstruiert und ein Objekt vom Typ `FigComponent` für die grafische Repräsentierung durch den Aufruf der Methode `makePresentation()` erzeugt. Das Graphenmodell des Rahmenwerks ist in Abschnitt 5.1 auf Seite 84 beschrieben. Durch den Aufruf der Methode `attachObserver()` registriert sich das grafische Objekt für die Komponente bei seinem Subjekt `SSDComponentView`. Mit dieser Registrierung erfolgt die automatische Anmeldung der Klasse `SSDComponentView` als Listener vom Typ `ModelChangeListener` beim assoziierten Modellelement `Component`. Der verkettete Registrierungsprozess ist in Abbildung 7.5 dargestellt. Vor der Registrierung bei einem Subjekt wird der Schlüssel durch die Methode `getViewKey()` bereitgestellt, der alle View-Elemente eines Wurzelobjekts eindeutig identifiziert. Die Anzeigeeigenschaften für die Klasse `FigComponent` werden dem übergebenen View-Element entnommen und die Attribute des Editorelements durch die Methoden `setLocation()` und `setSize()` dementsprechend belegt. Im abschließenden Schritt wird das Objekt dem Graphenmodell `DefaultGraphModel` des Rahmenwerks durch den Aufruf der Methode `addNode()` auf dem Graphenmodell hinzugefügt. Die Aktualisierung der Editor-Zeichenfläche erfolgt durch das im Rahmenwerk implementierte MVC-Konzept [KP88].

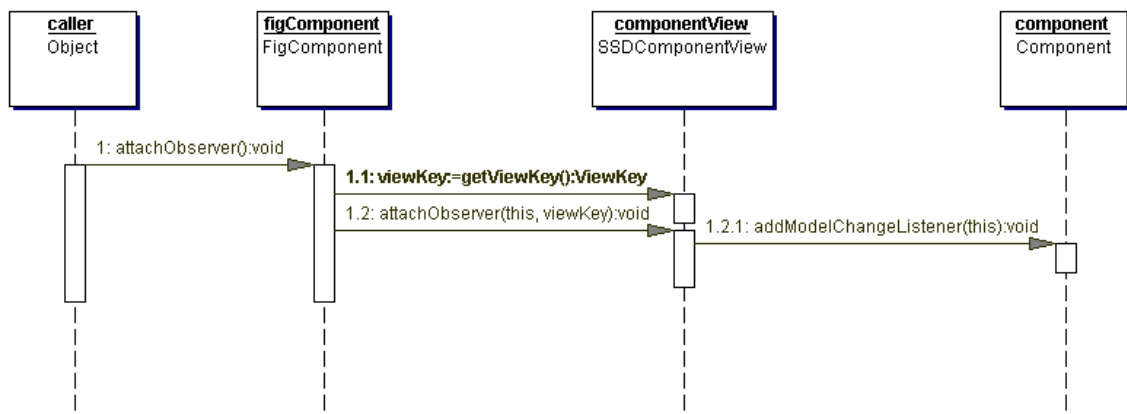


Abbildung 7.5. Die Registrierung des View-Elements beim Modell wird durch die Anmeldung von `FigComponent` bei seinem Subjekt `SSDComponentView` ausgelöst.

2.1.2 Löschen einer Komponente

Der Ablauf beim Löschen einer Komponente ähnelt in seiner Struktur dem in Abschnitt 2.1.1 auf Seite 65 beschriebenen Vorgang beim Einfügen einer Komponente. Die Schritte von der Initi-

ierung der Löschoption durch den Benutzer bis zum Aufruf der Methoden zur Weiterleitung der aufgetretenen Ereignisse ist in Abbildung 7.6 auf der nächsten Seite beschrieben. Wird die Ausführung des Befehls `CmdRemove` der Editorebene durch den Benutzer veranlasst, wird die selektierte Figur durch den Methodenaufruf `remove()` zum Löschen ausgewählt. Das mit der Figur assoziierte View-Element `SSDComponentView` der Komponente wird über die Methode `getComponentView()` bereitgestellt und als aktueller Parameter im Aufruf der Methode `removeSubComponents()` auf dem Wurzelobjekt `SSDComponentRootView` übergeben. Diese Methode ist ein Teil der Schnittstelle `SSDRootView2Model`. Im Wurzelobjekt wird über das übergebene View-Element das assoziierte Modellelement vom Typ `Component` mit der Methode `getModel()` zur Verfügung gestellt. Die Oberkomponente `parent` der zu löschenden Komponente stellt die Methode `getSuperComponent()` bereit. Der Löschvorgang in der Modellebene beginnt mit dem Auslösen eines Ereignisses, welches durch den Methodenaufruf `fireModelRemove()` an die Listener weitergeleitet wird. Mit dem Ereignis wird das Entfernen eines Modellelements angezeigt. Durch den Methodenaufruf `removeSubComponents()` auf der Oberkomponente wird die zu löschende Komponente entfernt. Das Löschen der Assoziation zwischen Komponente und seiner Oberkomponente wird durch ein weiteres Ereignis angezeigt und durch die Methode `fireObjectRemoved()` an die registrierten Objekte weitergeleitet.

Die folgenden Schritte sind in der Abbildung 7.7 auf Seite 72 dargestellt. Als Listener vom Typ `ModelChangeListener` ist das Wurzelobjekt `SSDComponentRootView` bei seinem assoziierten Modellelement `parent` registriert. Die Behandlung des aufgetretenen Ereignisses erfolgt in der Methode `objectRemoved`, die Teil der Schnittstelle `ModelChangeListener` ist, die durch die Schnittstelle `SSDRootView2Model` erweitert wird. Die Methode `notifyObserver()` erhält als aktuellen Parameter ein Objekt vom Typ `Action`. In der dieser Methode wird die Methode `update()` auf dem Editorfenster `EditorFrame` aufgerufen, welche ein Teil der Schnittstelle `View2GUI` ist. Entsprechend der gewünschten Operation führt der Editor die Methode `removeNode()` aus, die als Parameter das View-Element erhält, dessen Repräsentierung im Editor gelöscht werden soll. Die wichtigsten Schritte dieser Methode sind die Abmeldung beim Subjekt durch die Methode `detachObserver()` und das Entfernen des Objektes aus dem Graphenmodell des Rahmenwerks durch den Aufruf von `removeNode()`. Durch das MVC-Konzept des Rahmenwerks wird die Anzeige bei der Änderung des Graphenmodells automatisch aktualisiert. Das Graphenmodell ist in Abschnitt 5.1 auf Seite 84 näher beschrieben. Die Deregistrierung erfolgt wie bei der Anmeldung in einer verketteten Abfolge. Diese Schritte sind in Abbildung 7.8 auf Seite 73 dargestellt. Der Abmeldevorgang wird durch den Aufruf der Methode `detachObserver()` ausgelöst. Diese Methode stellt durch den Aufruf der Methode `getViewKey()` den Schlüssel für das zu entfernende View-Element bereit. Dieser wird zusammen mit einer Referenz auf das aufrufende Objekt `FigComponent` der Methode `detachObserver()` der Klasse `SSDComponentView` als aktueller Parameter übergeben. Das Element der View-Ebene meldet sich als Listener vom Typ `ModelChangeListener` von seinem assoziierten Modellelement `component` ab.

Nach der Ausführung der beschriebenen Aktionen ist die Anzeige im Editor mit den Daten der View- bzw. Modellebene synchron. Für einen Löschalgorithmus gibt es andere Alternativen. Die prototypische Implementierung realisiert einen Algorithmus, bei dem lediglich solche Modellelemente gelöscht werden können, die keine Assoziationen zu anderen Elementen aufweisen. Komponenten werden nur gelöscht, wenn sie keine Ports oder Unterkompo-

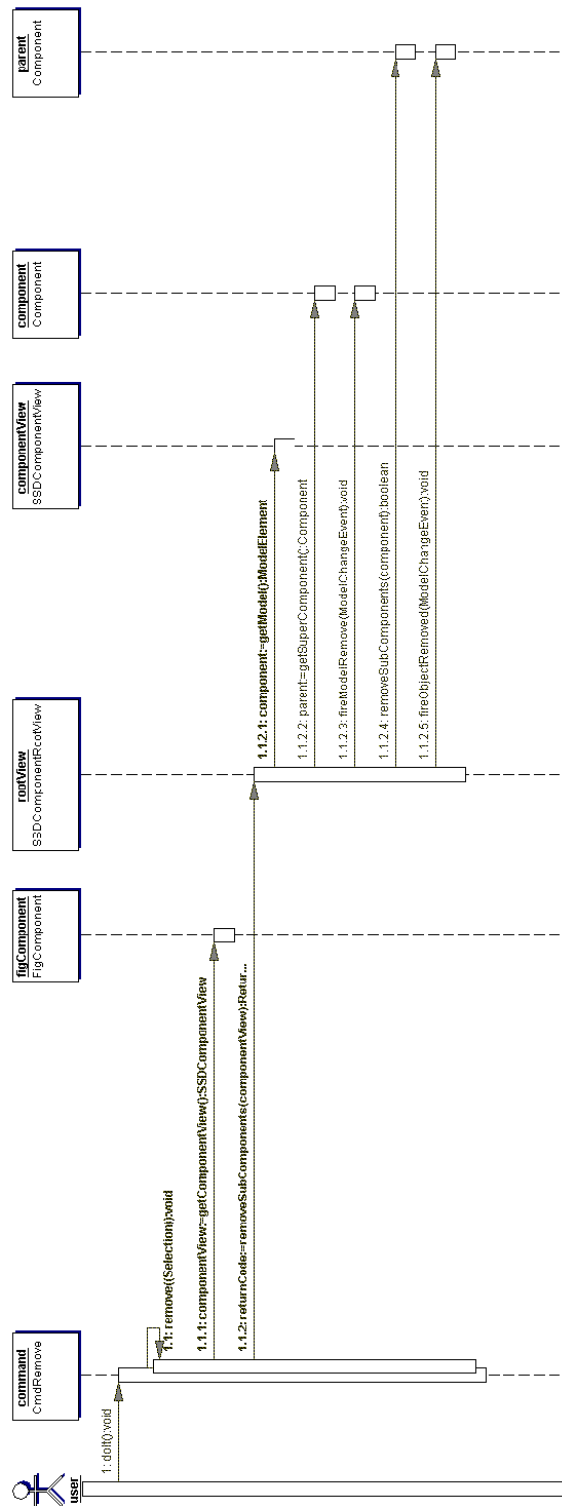


Abbildung 7.6. Der Ablauf beim Löschen einer Komponente von der Initiierung der Aktion im Editor bis zur Weiterleitung an das Modell.

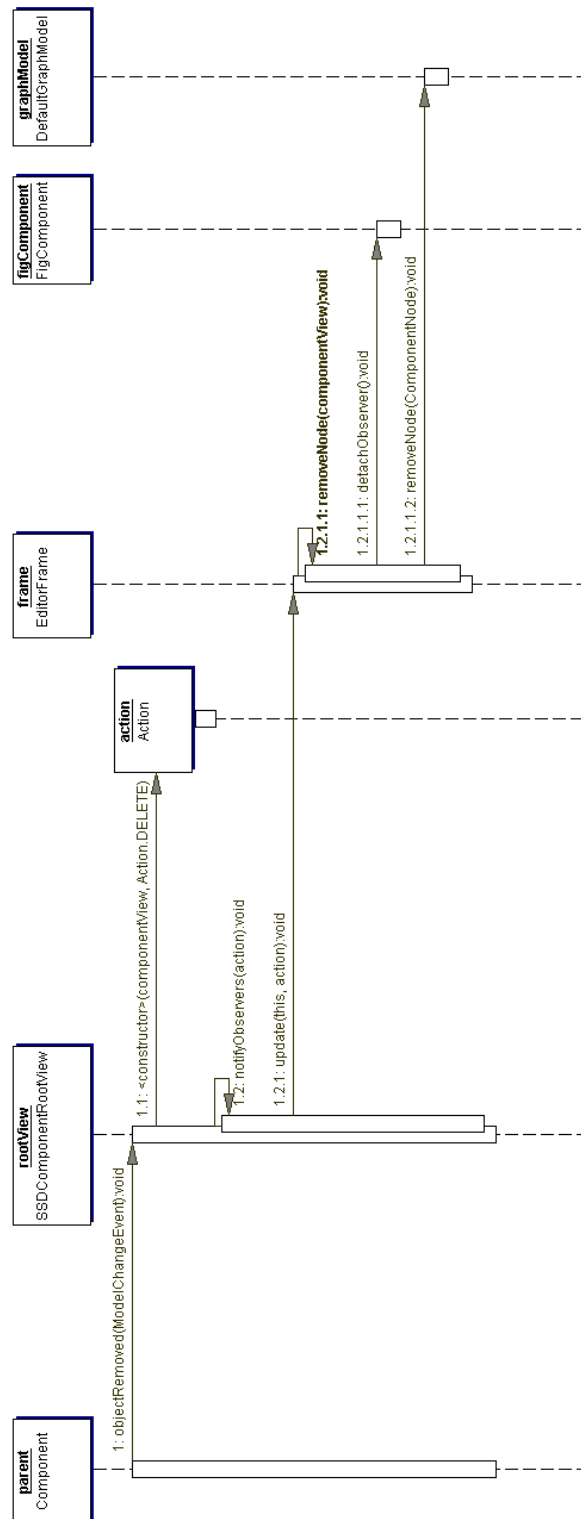


Abbildung 7.7. Der Ablauf beim Löschen einer Komponente vom Aufruf zur Benachrichtigung der Modell-Listener bis zur Aktualisierung der Anzeige im Editor.

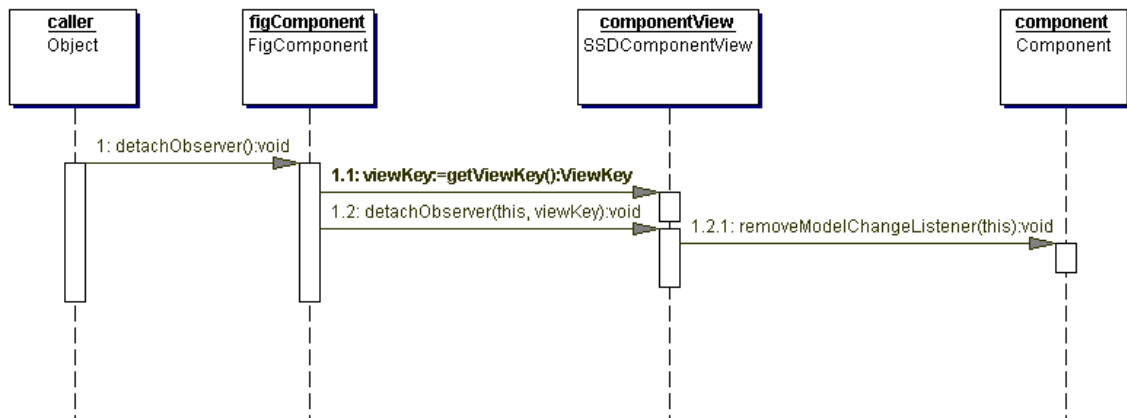


Abbildung 7.8. Meldet sich das Editorelement FigComponent bei seinem Subjekt ab, deregistriert sich auch das View-Element SSDComponentView bei seinem Modellelement Component.

nenten referenzieren. Ein Port wird entfernt, falls kein Kanal mit ihm verbunden ist. In der Implementierung wird beispielsweise in der Methode `removeSubComponents()` der Klasse `SSDComponentRootView` geprüft, ob das zu entfernende Element Referenzen auf Unterkomponenten oder Ports aufweist. Wenn eine dieser Bedingungen zutrifft, wird durch einen Rückgabewert vom Typ `ReturnCode` der Aufrufer der Methode, d.h. hier der Editor, über den Grund für die nicht mögliche Ausführung der gewünschten Aktion informiert. Die Klasse `ReturnCode` ist ausführlich in Abschnitt 4.1 auf Seite 81 beschrieben.

Alternativ ist eine Implementierung denkbar, bei der die Interaktion des Benutzers erforderlich ist. Soll eine Komponente mit mehreren Unterkomponenten entfernt werden, erfolgt die Durchführung dieser Operation erst nach dem Hinweis an den Benutzer über die Auswirkungen der Aktion und einer Bestätigung durch den Anwender. Für die Implementierung ist ein rekursiver Algorithmus nötig, der alle aggregierten Modellelemente entfernt. Beim Löschen einer Komponente werden beispielsweise alle Unterkomponenten entfernt. Bei der Realisierung dieser Vorgehensweise werden die Modellelemente gelöscht und die View-Ebene wird durch ein Ereignis benachrichtigt, das die Entfernung des in der Modellhierarchie am weitesten oben gelegenen Modells beschreibt. Das Auslösen von Ereignissen bei jedem gelöschten Modellelement würde den Umfang des Datenflusses zwischen den Ebenen erhöhen. In der Ereignisbehandlung der View-Ebene werden die betreffenden View-Elemente gelöscht und der Editor wiederum nur durch eine Aktualisierungsnachricht informiert. Weil die Realisierung ohne Benutzerinteraktion die einfachere Variante darstellt, ist diese im implementierten Prototyp realisiert.

Die in den Abschnitten 2.1.1 auf Seite 65 und 2.1.2 auf Seite 69 beschriebenen Modelloperationen verdeutlichen den Ablauf bei Änderungen am Modell. Die weiteren Operationen, beispielsweise zum Einfügen von Ports oder Ändern von Attributen wie die Bezeichnung eines Elements, werden analog zu den bereits beschriebenen Abläufen ausgeführt. Deshalb wird auf eine ausführliche Beschreibung der weiteren Modelloperationen verzichtet. Die Übersicht über alle realisierten Modelloperationen ist in Abbildung 7.2 auf Seite 66 dargestellt.

Beim Mehrprogramm- oder Mehrbenutzerbetrieb ist bei jeder Art von Modelloperation die Einhaltung von Konsistenzbedingungen auf dem Modell nötig. Bei den beschriebenen Operationen und in der Implementierung des Prototyps sind diese Bedingungen jedoch nicht berücksichtigt. Das Design der Anwendung ist so gestaltet, dass die dafür notwendigen Erweiterungen ergänzt werden können. Wie Konsistenzbedingungen eingehalten werden können, ist in Kapitel 8 auf Seite 103 beschrieben.

Wird die Funktionalität zur Steuerung der Anzeige wie in Abschnitt 3.2 auf Seite 58 beschrieben in die Ebene des Editors verlagert, werden die Änderungen am Modell im Editor vorgenommen. Die Implementierung dieser Variante erfordert jedoch einen aufwändigeren Aktualisierungsprozess für die Daten im Editor. Denn es müssen Operationen, die durch den Editor selbst ausgelöst werden, von solchen unterschieden werden, die durch andere Benutzer oder Anwendungen initiiert werden. Dadurch steigt die Komplexität der zu implementierenden Anwendungsfälle. Alternativ werden Änderungen am Modell nur zu bestimmten Zeitpunkten in das zentrale Systemmodell eingebracht. Treten dabei Inkonsistenzen auf, welche durch den Mehrprogramm- oder Mehrbenutzerbetrieb ausgelöst werden, wird im Editor das gesamte Wurzelobjekt verworfen und seine aktuelle Objektmenge der View-Elemente neu angezeigt. Bei dieser Variante ist die Unterscheidung der verschiedenen Auslöser der Änderungen am Modell nicht nötig. Weil hier keine Fallunterscheidungen erforderlich sind, ist der Implementierungsaufwand geringer. Jedoch ist die Funktionalität für Modelloperationen in die Ebene des Editors verlagert. Damit ist eine konsequente Schichtenarchitektur nicht realisiert.

2.2 View-Operationen

Änderungswünsche ausschließlich an den Attributen der Views werden vom Editor an die betreffende View weitergegeben. Die gewünschten Aktionen werden in der View-Ebene ausgeführt und der Editor über die Änderungen benachrichtigt. Alle Operationen, die diese Eigenschaften besitzen, sind in eigenen Schnittstellendefinitionen für View-Elemente zusammengefasst. Abbildung 7.9 zeigt die Deklaration dieser Methoden. Die Übersicht, welche Schnittstelle von welchem View-Element implementiert ist, ist in Tabelle 7.1 auf Seite 65 dargestellt. View-Elemente mit fehlenden Schnittstellen für View-Operationen oder Hilfsfunktionen bedürfen für die Funktionalität des Prototyps keine Deklaration von zusätzlichen Methoden als Erweiterungen für das Editorenkonzept.

Alle Operationen, welche die Eigenschaften von View-Attributen wie z.B. die Größe oder Position einer Komponente verändern, verlaufen nach dem gleichen Schema der Ablaufstruktur. Deshalb wird im folgenden Abschnitt beispielhaft der Ablauf der Schritte beim Verschieben einer Komponente erläutert, um das allgemeine Prinzip darzustellen.

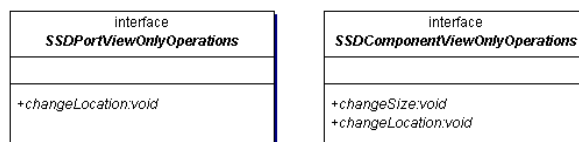


Abbildung 7.9. Schnittstellen für Operationen, die lediglich die Attribute der View-Ebene verändern.

2.2.1 Verschieben einer Komponente

Das Verschieben einer Komponente im Editor wird durch den Benutzer initiiert. Dabei werden die ausgelösten Mausereignisse vom Editor so verarbeitet, dass die Änderungen zunächst am assoziierten View-Element vorgenommen und von der View-Ebene an das grafische Objekt im Editor weitergegeben werden. In der Editorebene erfolgt die Aktualisierung der Position entsprechend der Belegung des View-Attributs für die Position. Die Elemente des Editors werden ausschließlich durch ihre assoziierten View-Elemente gesteuert. Die Durchführung einer gewünschten Änderung der Anzeigeeigenschaften erfolgt nur, wenn die betreffenden Randbedingungen erfüllt sind. Beispielsweise wird durch einen Layout-Algorithmus in der View-Ebene bestimmt, ob eine Verschiebung entsprechend des Benutzerwunsches möglich ist.

Die Schritte beim Verschieben einer Komponente im Editor sind in Abbildung 7.10 auf der nächsten Seite dargestellt. Beim Verschieben einer Figur für eine Komponente im Editor wird zur Ereignisbehandlung die Methode `mouseDragged()` der Klasse `MyModeModify` durch das Rahmenwerk aufgerufen. Das Rahmenwerk bestimmt das grafische Element, das bewegt wird, in diesem Fall ist es eine Figur vom Typ `FigComponent`. Mit dieser Figur ist das entsprechende Element aus der View-Ebene assoziiert. Durch den Aufruf der Methode `getViewElement()` wird das Objekt vom Typ `SSDComponentView` bereitgestellt. Es folgt der Aufruf der Methode `changeLocation()` auf dem Objekt `SSDComponentView`. Diese Methode ist ein Teil der Schnittstelle `SSDComponentViewOnlyOperations` und erhält als aktuellen Parameter den Wert, der die neue Position der Komponente angibt. Eine Konvertierung der Koordinaten im Editor in das Format der View-Ebene und umgekehrt ist nicht erforderlich, weil beide Schichten Koordinaten im Format (x,y) verwenden, welche die Pixelpositionen beschreiben. Durch die Methode `setLocation()` wird der Wert des View-Attributs für die Position belegt. Bei der Änderung der Position wird ein `Action`-Objekt erzeugt, das die im Editor durchzuführende Aktion und das betreffende View-Element `SSDComponentView` kapselt. Die Klasse `Action` ist in Abschnitt 4.2 auf Seite 82 beschrieben. Zur Benachrichtigung des registrierten Observers `SSDComponentView` wird die Methode `notifyObservers()` aufgerufen. Diese benachrichtigt alle registrierten Observer, u.a. auch die Figur vom Typ `FigComponent`, durch den Aufruf der Methode `update()` über eine nötige Aktualisierung der Anzeige. Dazu wird die Methode `refresh()` aufgerufen. Darin wird der aktuelle Wert des Attributs für die Position des Elements `SSDComponentView` durch die Methode `getLocation()` abgefragt und im Aufruf der Methode `setLocation()` auf die Editorfigur `FigComponent` übertragen. Als Ergebnis repräsentiert der Editor den aktuellen Wert der Position des View-Elements.

Aktionen, die ausschließlich die Attribute von View-Elementen verändern, verlaufen in ihrer Aufrufstruktur wie die gerade beschriebene Operation zum Verschieben. Deshalb wird auf eine ausführliche Erläuterung der übrigen Aktionen verzichtet. Ein Überblick über die implementierte Funktionalität für View-Operationen ist in Abbildung 7.9 auf der vorherigen Seite dargestellt.

Die beschriebene Ablaufstruktur ist auf weitere Operationen übertragbar. Beispielsweise wird zum Verbergen einer Komponente der View-Ebene in der Schnittstelle `SSDComponentViewOnlyOperations` eine neue Operation deklariert. Bei dieser Operation sind auch die assoziierten View-Elemente wie Ports und die Kanäle berücksichtigt. Um den Editor zu steuern, wird in der Klasse `Action` eine neue Konstante definiert, welche die durchzuführende Aktion im Editor beschreibt. Alle durch das Verbergen einer Komponente eingeschlossenen

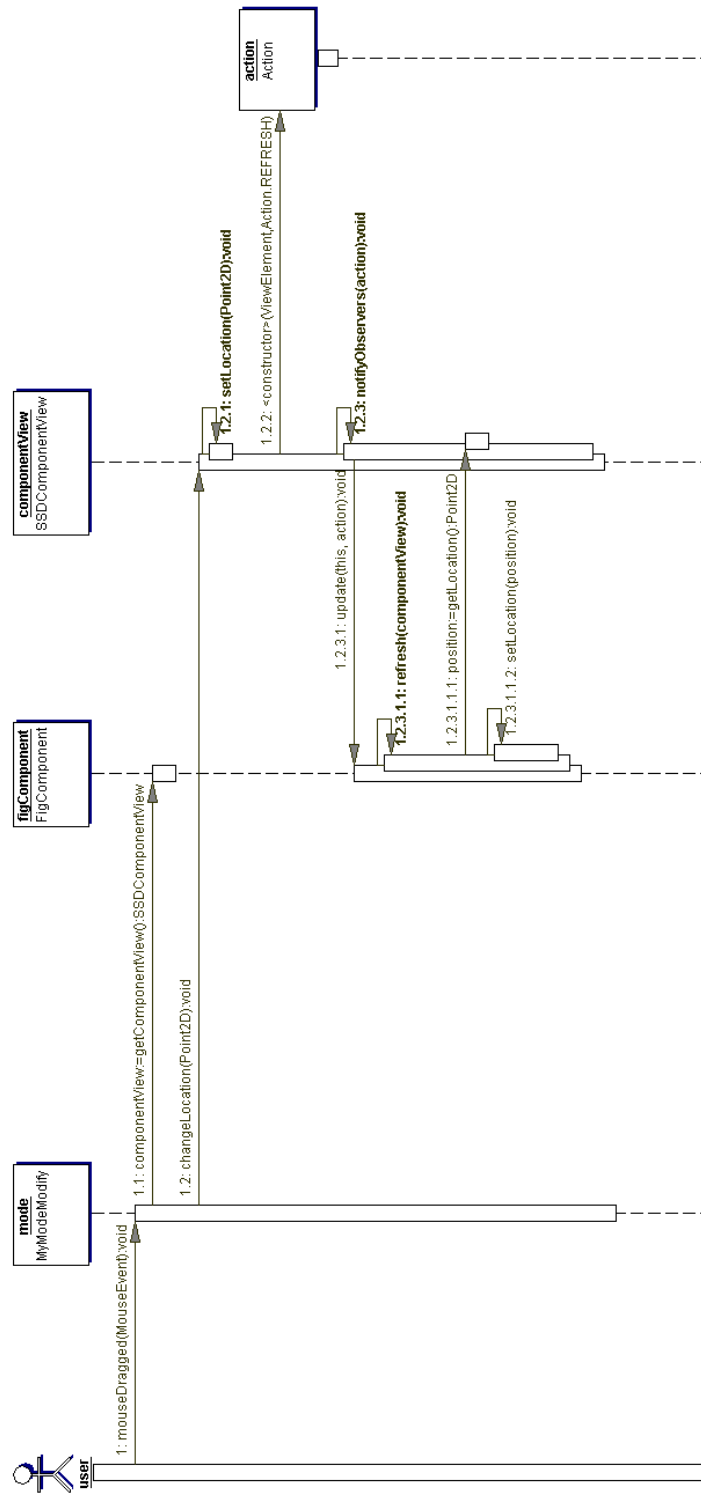


Abbildung 7.10. Schritte beim Verschieben einer Komponente. Aus Gründen der Übersichtlichkeit sind nicht alle Aufrufe beim Verschieben einer Komponente angegeben.

View-Elemente benachrichtigen als Subjekte des Observer-Entwurfsmusters ihre assoziierten grafischen Objekte durch den Aufruf der Methode `update()`. Eine weitere Methode implementiert das erneute Anzeigen von verborgenen Elementen als Teil der Schnittstelle zwischen View und Editor. In der Implementierung des Prototyps repräsentiert die Ebene des Editors die Daten aus der View-Ebene, welche die Information des Modells um Attribute für die Anzeige ergänzen. Um jedoch unterschiedliche Ausschnitte von Views zu realisieren, ist neben der beschriebenen Funktionalität die zusätzliche Erweiterung notwendig: Der Speicher- und Lademechanismus der View-Information muss so erweitert werden, dass gefilterte Views unterstützt werden. Das Filtern von View-Elementen beeinflusst also nicht nur die Editorebene, sondern auch die View-Schicht.

Ist der Editor nicht über die View-Ebene gesteuert, würden die View-Operationen vollständig in der Editorebene realisiert werden. Dabei wird die gesamte Funktionalität des Rahmenwerks eingesetzt. Beispielsweise wird das Verschieben von grafischen Elementen vom Rahmenwerk realisiert. Die Änderungen an den Anzeigeeigenschaften werden von der Editor- in die View-Ebene propagiert. Die Synchronisation mit anderen Benutzern oder Anwendungen erfolgt nur über das Modell. Jeder Anwender arbeitet exklusiv auf eigenen View-Elementen. Die Steuerung des Editors durch die View-Ebene ist bei diesem Ansatz nicht gegeben. Deshalb ist diese Variante im Prototyp nicht implementiert.

In der Realisierung des Prototyps sind keine Rückgabewerte bei View-Operationen implementiert. Dennoch treten Situationen auf, in denen ein Rückgabewert erwünscht wird. Bei einer Erweiterung der View-Klassen mit einem Layout-Algorithmus können vom Benutzer initiierte View-Operationen scheitern. Werden bei einer gewünschten Operation die durch das Layout-Verfahren bestimmten Einschränkungen verletzt, werden die Änderungen an den View-Attributen nicht vorgenommen. In diesem Fall wird der Aufrufer der Änderungsmethode durch einen Rückgabewert ähnlich wie beim Scheitern einer Modelloperation über die Ursache des Scheiterns informiert. Der Benutzer erhält eine Rückmeldung, dass die gewünschte Operation nicht durchführbar ist. Dazu ist in der Implementierung die Erweiterung erforderlich, welche die Rückgabewerte der View-Operationen mit Objekten vom Typ `ReturnCode` versieht. Der Rückgabewert wird vom Aufrufer der Methode ausgewertet, um dem Benutzer beim Scheitern der Aktion einen entsprechenden Hinweis in einem Dialogfenster anzuzeigen.

Beim gewählten Design, bei dem die Steuerung des Editors durch die View-Ebene erfolgt, ist im Mehrbenutzerbetrieb eine Synchronisierung der View-Operationen nötig, sobald mehrere Benutzer auf dem selben Wurzelobjekt arbeiten. Erweiterungen für die Synchronisation von Modelloperationen sind in Kapitel 8 auf Seite 103 beschrieben. Die View-Elemente besitzen eine ähnliche Struktur wie das `AUTOFOCUS`-Metamodell. Die Konzepte für die Synchronisation von Modelloperationen können deshalb auch auf die View-Ebene übertragen werden.

2.3 Hilfsfunktionen

In Abbildung 7.11 auf der nächsten Seite sind die Funktionen in Schnittstellen beschrieben, die weder Modell- noch View-Eigenschaften verändern, aber Hilfsfunktionalität für die View-Ebene deklarieren. Die Tabelle 7.1 auf Seite 65 gibt einen Überblick über die Schnittstellen und ihre implementierenden Klassen. Die Schnittstelle `SSDRootViewHelpers` deklariert lediglich Datenstrukturen, die das Zeichnen von Komponenten oder Ports der internen Sicht auf eine Komponente an bestimmten Positionen ermöglichen. Die Information über die Position des

Mausklicks, an welcher der Benutzer beispielsweise eine neue Komponente einfügen möchte, wird zwar beim Aufruf der Methode `addSubComponents()` auf einem Objekt der Klasse `SSDComponentRootView` übergeben, ist aber in der Nachricht des Modells an die View-Ebene nicht mehr enthalten. Denn die Veränderung von Modelleigenschaften wird durch das Auslösen von Ereignissen angezeigt, die nur Modellinformationen beinhalten. Aus diesem Grund wird die Position in der View-Ebene in Hash-Tabellen zwischengespeichert. Beim Einfügen beispielsweise einer neuen Komponente wird die gewünschte Position des Benutzers für das neue Element in der Datenstruktur `component2location` gespeichert. Bei der Benachrichtigung des View-Elements `SSDComponentRootView` über das Einfügen einer neuen Komponente wird die Position der Hash-Tabelle entnommen und das Attribut im konstruierten View-Element `SSDComponentView` dementsprechend belegt. In der Hash-Tabelle `port2location` der Schnittstelle `SSDRootViewHelpers` wird die Position für einen Port der internen Sicht einer Komponente gespeichert.

In der Schnittstelle `SSDComponentViewHelpers` ist die Datenstruktur `port2location` für die Speicherung der gewünschten Position eines Ports der externen Sicht einer Komponente deklariert. Die Methoden `contains()` und `getBorderLocation()` implementieren Hilfsfunktionalität für das Platzieren von Ports auf Komponenten. Um Ports nur auf dem Rand einer Komponente zu platzieren, reicht die Information über die Position des Mausclicks nicht aus. Aus dieser übergebenen Information wird die Position auf dem Rand der Komponente ermittelt, die der gewünschten am nächsten ist. Die Anforderungen durch die Editorebene werden durch die Einschränkungen in der View-Ebene überprüft und angepasst. Hier wird der Vorteil der Steuerung des Editors durch die View-Schicht deutlich. In den Methoden `getSubComponents()` und `hasSubComponents()` werden die Anfragen auf der View-Ebene an das Modell delegiert. Die View-Ebene verbirgt also den Zugriff auf Informationen aus der Modellebene.

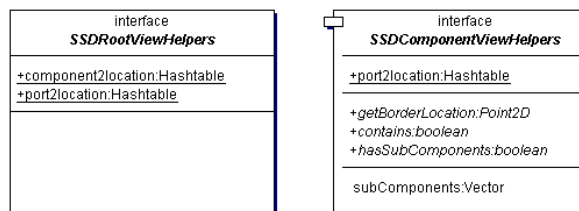


Abbildung 7.11. Erweiterungen für Hilfsfunktionen.

2.4 Probleme der Ereignisverarbeitung

Bei der Implementierung des Prototyps wird bei einer Änderung am Modell durch einen Methodenaufruf auf dem betreffenden Modellelement die Propagierung der Änderung ausgelöst. Dazu wird beispielsweise beim Einfügen einer neuen Komponente auf der Oberkomponente die Methode `fireObjectInserted()` aufgerufen. In einer Warteschlange der Oberkomponente sind alle registrierten Objekte gespeichert, die durch den Methodenaufruf `objectInserted()` über die Änderung benachrichtigt werden. Dabei kann folgendes Problem auftreten: Die Ereignisbehandlung in einem der registrierten Objekte kann sehr lange dauern. Befindet sich in der

Liste der Objekte ein View-Element, das die Änderung am Modell auslöste, weil durch den Benutzer im Editor ein neues Modellelement gewünscht ist, wird die wartende Klasse in der Editorebene blockiert, solange die Änderung des Modells an das View-Element propagiert wird. In diesem Zeitraum ist die Benutzeroberfläche des Editors aber nicht mehr benutzbar, weil der Aufruf aus dem Editor auf das Ergebnis der Operationsausführung wartet. Ein Lösungsansatz ist die Realisierung durch leichtgewichtige Prozesse, die mit unterschiedlichen Aufgaben beauftragt sind. Dazu werden zwei Arten von leichtgewichtigen Prozessen unterschieden, die für die Benutzerinteraktion und die Abarbeitung von geforderten Operationen auf der Modell- oder der View-Ebene zuständig sind. Mit dieser Lösung wird eine Blockierung der Benutzeroberfläche vermieden. Die Verwendung von mehreren leichtgewichtigen Prozessen erfordert ihre Synchronisierung. Letztere führt jedoch zu folgenden Problemstellungen:

- Zu wenig Synchronisierung kann gleichzeitigen Zugriff auf gemeinsame Daten verursachen.
- Eine unbefriedigende Leistungsfähigkeit des Systems kann durch zu viel Synchronisierung verursacht werden.
- Zu viel Synchronisierung kann zu Verklemmungen führen.

Die Einführung der Verarbeitung von Aufgaben durch mehrere leichtgewichtige Prozesse erhöht durch die erforderlichen Synchronisierungsmaßnahmen die Komplexität und damit die Fehleranfälligkeit der gesamten Anwendung. Deshalb ist eine sorgfältige Synchronisation der leichtgewichtigen Prozesse nötig.

3 Verwendung der erweiterten View-Elemente

3.1 Modelloperationen

Alle Operationen, die Änderungen am Modell auslösen, laufen nach dem gleichen Schema ab. Dabei werden folgende Schritte ausgeführt:

- Aufruf einer Methode für eine Modelloperation durch den Editor auf der View-Ebene
- Durchführung der Operation am Modell durch die View-Ebene
- Konstruktion und Auslösen eines Ereignisses, das die Veränderung des Modells anzeigt
- In der Ereignisbehandlung der View-Ebene wird entsprechend der Angaben aus dem Ereignis ein View-Element konstruiert oder angepasst .
- Benachrichtigen des Editorelements durch das Observer-Entwurfsmuster
- Behandlung der Aktualisierung in der Editor-Ebene
- Rückgabe eines Wertes von der View-Ebene an die Editor-Ebene, das dem Aufrufer das Ergebnis der Operation anzeigt

Dieses allgemeine Schema wird beispielsweise beim Einfügen einer neuen Komponente in die folgenden Schritte umgesetzt:

- Aufruf der Methode `addSubComponents()` auf `SSDComponentRootView` durch den Editormodus `ModePlaceComponent` für das Erstellen einer neuen Komponente
- Erzeugen der neuen Modellkomponente, Belegen der Attribute Name und Kommentar und Hinzufügen der neuen Komponente zu ihrer Oberkomponente
- Konstruktion und Auslösen des Ereignisses, das über das Einfügen einer neuen Komponente informiert
- In der Ereignisbehandlung in `SSDComponentRootView` wird ein neues View-Element vom Typ `SSDComponentView` erzeugt und sein Attribute belegt
- Das Editorelement `EditorFrame` wird durch den Aufruf der Methode `update()` durch `SSDComponentRootView` benachrichtigt
- In der Methode `update()` wird die Aktualisierung der Anzeige vorgenommen
- Die Methode `addSubComponents()` zeigt den Erfolg der Operation an durch die Rückgabe eines Wertes vom Typ `ReturnCode` an den Aufrufer `ModePlaceComponent`

3.2 View-Operationen

Bei View-Operationen erfolgt der Informationsaustausch ausschließlich zwischen den Ebenen View und Editor. Dabei weist die Ablaufstruktur das folgende allgemeine Schema auf:

- Aufruf einer Methode für eine View-Operation durch den Editor an der View-Schnittstelle
- Durchführung der Änderung an den Werten der View-Attribute
- Benachrichtigen des Editors durch das Observer-Entwurfsmuster
- Behandlung der Aktualisierung in der Editor-Ebene

Diese Ablaufstruktur umfasst z.B. beim Verschieben einer Komponente die folgenden Schritte:

- Im Editormodus `MyModeModify` wird die Methode `changeLocation()` auf dem View-Element `SSDComponentView` aufgerufen.
- In der Klasse `SSDComponentView` wird das Attribut für die Position des View-Elements neu belegt.
- Das Editorelement `EditorFrame` wird durch den Aufruf der Methode `update()` von einem Objekt des Typs `SSDComponentRootView` benachrichtigt.
- In der Methode `update()` wird die Aktualisierung der Anzeige vorgenommen und die Position der Figur für die Komponente entsprechend den Angaben aus dem assoziierten View-Element angepasst

4 Beschreibung besonderer Klassen

4.1 ReturnCode

Für die Erweiterungsfähigkeit der Implementierung ist der Rückgabewert von Methoden entscheidend, die Änderungen am Modell oder an View-Attributen vornehmen. Kann die geforderte Aktion in der View- oder Modell-Ebene nicht ausgeführt werden, wird dem Aufrufer der Methode durch einen Rückgabewert die Ursache für das Scheitern angezeigt. Fehler treten beispielsweise auf, falls es bei der Konstruktion von Modellelementen Namenskonflikte gibt oder Konsistenzbedingungen für das Modell nicht eingehalten werden. Eine Konsistenzbedingung ist z.B. verletzt, wenn eine Komponente gelöscht werden soll, die Unterkomponenten besitzt.

In der Implementierung des Prototyps sind lediglich Modelloperationen mit Rückgabewerten vom Typ `ReturnCode` definiert. Diese Klasse kapselt das Ergebnis der gewünschten Operation. Die in ihr definierten Konstanten beschreiben den Erfolg oder die Ursache für das Scheitern der angeforderten Aktion. Die Klasse kann durch die Definition von zusätzlichen Konstanten für andere Anforderungen der Modell- oder View-Operationen erweitert werden. Beispielsweise erfordert die Integration eines Layout-Verfahrens in die View-Ebene, welches das Verschieben von View-Elementen durch die Layout-Vorgaben einschränkt, die Einführung von weiteren Konstanten. Letztere werden in der Klasse `ReturnCode` neu definiert.

Die Klasse zur Realisierung von Rückgabewerten ist in Abbildung 7.12 dargestellt, die Bedeutung ihrer Konstanten ist in Tabelle 7.2 auf der nächsten Seite erläutert. Die Variable `code` der Klasse `ReturnCode` ist mit einer der definierten Konstanten belegt.

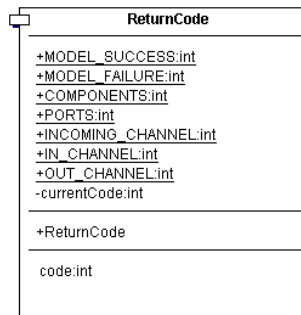


Abbildung 7.12. Belegungen des Attributs `code` beschreiben den Erfolg oder die Ursache für das Scheitern einer Aktion an.

Konstante	Bedeutung
MODEL_SUCCESS	Die Modelloperation ist erfolgreich ausgeführt.
MODEL_FAILURE	Die gewünschte Operation ist gescheitert.
COMPONENTS	Eine Komponente kann nicht gelöscht werden, weil sie Unterkomponenten besitzt.

Konstante	Bedeutung
PORTS	Eine Komponente kann nicht gelöscht werden, weil sie Ports besitzt.
INCOMING_CHANNEL	Ein Kanal kann nicht mit dem gewünschten Port verbunden werden, weil dieser bereits einen eingehenden Kanal besitzt.
IN_CHANNEL	Ein Port kann nicht gelöscht werden, weil er einen eingehenden Kanal besitzt.
OUT_CHANNEL	Ein Port kann nicht gelöscht werden, weil er einen ausgehenden Kanal besitzt.

Tabelle 7.2. Die Bedeutung der Konstanten der Klasse ReturnCode.

4.2 Action

Bei der Bearbeitung von Ereignissen in der View-Ebene, die Änderungen am Modell anzeigen, wird nach der Art der aufgetretenen Ereignisse unterschieden. Entsprechend der Art des aufgetretenen Ereignisses wird der Methode `update()` zum Benachrichtigen der Observer des View-Elements ein Objekt vom Typ `Action` übergeben, das sowohl das betreffende View-Element als auch die durchzuführende Aktion beschreibt. Diese Kapselung ist in der Implementierung des Prototyps eingeführt. Die ursprüngliche Realisierung der `update()`-Methode berücksichtigt nicht die Art der Aktion, sondern lediglich das betreffende View-Objekt. Der Observer muss nun feststellen, wie die Aktualisierung zu behandeln ist. Dazu sind zwei unterschiedliche Ansätze möglich:

- Bei einer Benachrichtigung des Editors werden zunächst alle Editorelemente gelöscht und anschließend die aktuellen View-Elemente des Wurzelobjekts wieder angezeigt.
- Der Editor entscheidet anhand der aktuellen und der vorhergehenden Zusammensetzung des Wurzelobjekts, welche Operation durchzuführen ist.

Das erste Verfahren unterstützt die vollständige Unabhängigkeit eines View-Elements von der Editorebene. Denn die View-Ebene braucht über die Art der Operation, die im Editor ausgeführt werden soll, nicht informiert zu sein. Das gesamte Neuzeichnen der Objektemenge eines Wurzelobjekts reduziert jedoch die Performanz. Da die Schnittstelle zwischen Editor und Quest für eine Weiterentwicklung entworfen ist, muss sie auch die Möglichkeit für die Erweiterung des Editors zur Benutzerschnittstelle für eine Simulation eines modellierten Systems unterstützen. Dafür ist eine gute Leistungsfähigkeit der Software entscheidend. Außerdem ist das Entfernen und Neuzeichnen von Editorelementen für den Benutzer ein ungewöhnliches Verhalten. Denn dadurch wird bei jeder Aktualisierung des Editors ein Flackern der Anzeige verursacht. Das Editorfenster als Observer des Wurzelobjekts entfernt bei einer Aktualisierung alle grafischen Figuren und zeichnet sie entsprechend der aktuellen View-Vorgaben neu. Damit

Beschreibung besonderer Klassen

wird zwischen dem Einfügen und Entfernen, aber auch dem Verschieben oder der Größenänderung von Elementen nicht unterschieden. Als Folge entfallen die Verantwortlichkeiten der grafischen Objekte für die Aktualisierung ihrer Anzeigeeigenschaften. Deshalb ist eine Implementierung aller Editorelemente als Observer ihrer View-Elemente nicht erforderlich. Damit wird die Implementierung des Aktualisierungsvorgangs im Editorfenster vereinfacht, da keine Fallunterscheidungen nach der Art der durchzuführenden Aktion nötig sind. Es gibt nur einen Kommunikationskanal zwischen der Ebene des Editors und der Schicht der Views. Dabei handelt es sich um einen schmalen Kanal, der alle Informationen zwischen den beiden Ebenen transportiert. Das Design von Schnittstellen zielt gerade auf eine solche Realisierung ab.

Bei der zweiten Variante ist es die Aufgabe des Editors zu entscheiden, welche Operation durchzuführen ist und nur das betreffende Element zu ändern. Dazu vergleicht der Editor als Observer die aktuelle Zusammensetzung des Wurzelobjekts mit der vorhergehenden. Die dabei festgestellten Änderungen werden anschließend umgesetzt. Damit ist die Aktualisierung der Anzeige ein dem Benutzer vertrauter Vorgang. Die erneute Darstellung des Inhalts des gesamten Editorfensters ist nicht erforderlich. Für den Vergleich der beiden Zustände des Wurzelobjekts ist jedoch zusätzlicher Implementierungsaufwand nötig. Mit der zusätzlichen Implementierungsarbeit ist auch eine steigende Fehleranfälligkeit verbunden.

Beide Varianten haben jedoch entscheidende Nachteile. Die erste Möglichkeit zeichnet sich durch fehlende Performanz und ein Flackern bei jeder Aktualisierung des Editors aus. Die zweite Variante erhöht den Implementierungsaufwand und die Fehleranfälligkeit. Deshalb wird bei der Implementierung des Prototyps die Einführung des neuen Objekttyps `Action` gewählt. In diesem Objekt werden das betreffende View-Element und die durchzuführende Aktion im Editor gekapselt. Diese Entscheidung bringt jedoch eine gewisse Abhängigkeit der View-Ebene von der Editorschicht mit sich. Denn die View-Ebene muss ein Objekt vom Typ `Action` konstruieren und seine Attribute entsprechend der durchzuführenden Operation im Editor belegen. Dennoch wird diese Einschränkung weniger nachteilig bewertet als die der beiden erläuterten Alternativen. In Abbildung 7.13 ist die Klasse `Action` dargestellt. Die Variable `viewElement` der Klasse `Action` referenziert das betreffende View-Element. Die Variable `action` ist mit dem Wert einer Konstanten belegt. Die Bedeutung der definierten Konstanten ist in Tabelle 7.3 auf der nächsten Seite beschrieben.

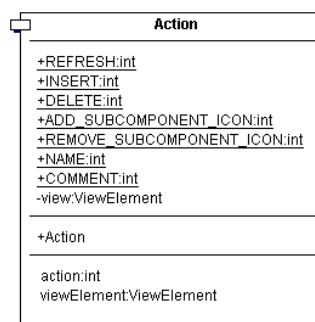


Abbildung 7.13. Die Klasse `Action` kapselt die durchzuführende Aktion und das betreffende View-Element.

Konstante	Bedeutung
REFRESH	Die Position oder die Größe eines Editorelements ist zu aktualisieren.
INSERT	Ein grafisches Objekt wird hinzugefügt.
DELETE	Ein grafisches Objekt ist zu entfernen.
ADD_SUBCOMPONENT_ICON	Das Symbol für die Dekomposition einer Komponente ist anzuzeigen.
REMOVE_SUBCOMPONENT_ICON	Das Symbol für die Dekomposition einer Komponente wird entfernt.
NAME	Die Beschriftung einer Figur wird geändert.
COMMENT	Der Kommentar für ein Element wird geändert.

Tabelle 7.3. Die Bedeutung der Konstanten der Klasse Action.

5 Beziehungen zwischen View-, Modell- und Editorelementen

5.1 Graphenmodell des GEF-Rahmenwerks

Diagramme, die in einer auf dem GEF-Rahmenwerk basierenden Anwendung angezeigt werden, sind die Visualisierung eines Graphenmodells. Das im Rahmenwerk eingesetzte Datenmodell besteht aus Knoten, die durch gerichtete Kanten verbunden sind. Die Knoten referenzieren eine Menge von Ports, welche die Verbindungspunkte zwischen einer Kante und einem Knoten darstellen. Die Elemente des Graphenmodells werden durch grafische Objekte angezeigt, sie können aber auch verborgen bleiben. Knoten, Ports und Kanten des Graphenmodells werden durch die Klassen `NetNode`, `NetPort` bzw. `NetEdge` dargestellt. Die für diese Arbeit relevanten Elemente des Quest-Modells sind `Component`, `Port` und `Channel`. Ein Port des Graphenmodells ist im Gegensatz zu einem Quest-Port eine Hilfsstruktur, welche zur Realisierung der Verbindung zwischen einem Knoten und einer Kante dient und nicht notwendigerweise ein sichtbarer Bestandteil eines Diagramms ist.

Die Struktur des in GEF verwendeten Datenmodells ist in Abbildung 7.14 auf der nächsten Seite dargestellt. Die Klasse `DefaultGraphModel` referenziert eine Liste vom Typ `NetList`, welche alle Knoten (`nodes`) und Kanten (`edges`) des Modells beinhaltet. Ein Knoten vom Typ `NetNode` referenziert alle seine Ports (`ports`) vom Typ `NetPort`. Eine Kante `NetEdge` assoziiert ihren Quell- (`sourcePort`) und Zielport (`destPort`). Alle Primitive des Graphenmodells sind von der Klasse `NetPrimitive` abgeleitet.

Für die Abbildung zwischen den Elementen des Graphenmodells und den grafischen Objekten für die Darstellung eines Systemstrukturdiagramms existieren mehrere Möglichkeiten,

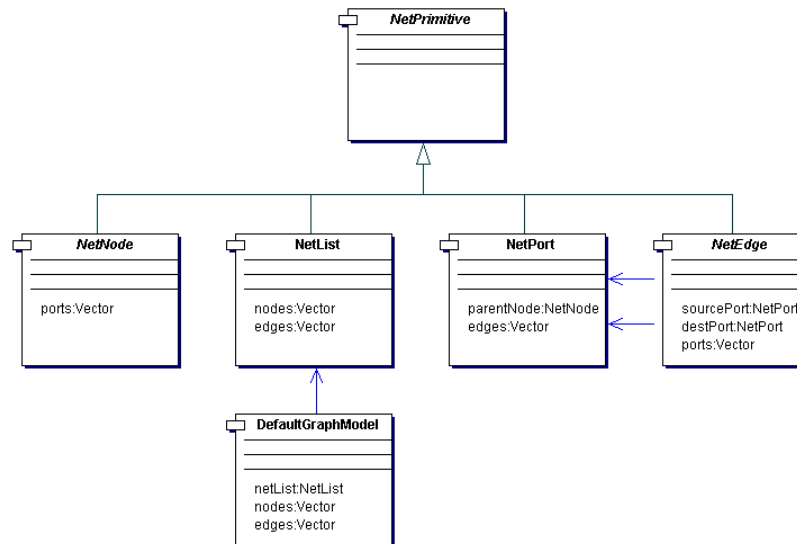


Abbildung 7.14. Im Graphenmodell von GEF werden die Primitive Knoten (NetNode), Kanten (Net-Edge) und Ports (NetPort) unterschieden.

welche im folgenden Abschnitt erläutert und bewertet werden.

5.2 Editor- und Graphenmodellelemente

Eine Möglichkeit für den Zusammenhang zwischen den Modellelementen des GEF-Graphenmodells und den grafischen Objekten eines Diagramms ist in Abbildung 7.15 auf der nächsten Seite beschrieben. Die implementierenden Klassen für die Darstellung an der Benutzerschnittstelle und im Graphenmodell sind in Tabelle 7.4 zusammengefasst. Die Figuren in der erwähnten Abbildung sind mit Nummern beschriftet, deren Bedeutungen in der zugehörigen Tabelle beschrieben sind.

Element	Klasse der Editorebene	Klasse des Graphenmodells
1	FigComponent	ComponentNode
2	FigRect für FigComponent und FigCircle für FigPort	ConnectingPort
3	FigEdgeLine	HiddenEdge
4	FigInternalOutPort	OutputPortNode
5	FigChannel	ComponentEdge
6	FigExternalInPort	InputPortNode

Tabelle 7.4. Erläuterungen zu Abbildung 7.15 auf der nächsten Seite

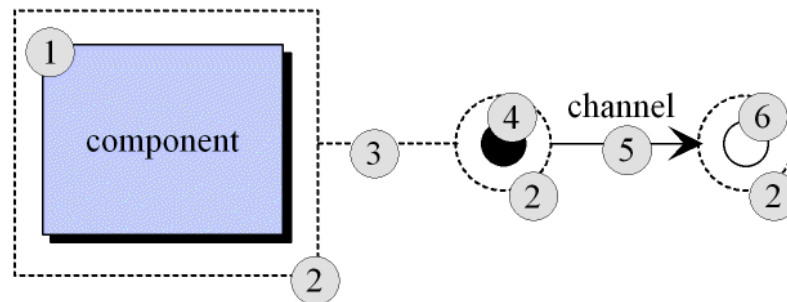


Abbildung 7.15. Der Zusammenhang zwischen dem Graphenmodell des Rahmenwerks und den Elementen in einem Diagramm. Die gestrichelten Elemente sind nicht sichtbar, sondern dienen nur zur Veranschaulichung der Konstruktion der grafischen Darstellung.

Jede grafische Darstellung im Editor referenziert ein Objekt aus dem GEF-Graphenmodell und umgekehrt. Die grafische Darstellung einer Quest-Komponente im Editor wird durch die Klasse `FigComponent` realisiert. Der entsprechende Knoten im Graphenmodell wird durch ein Objekt vom Typ `ComponentNode` dargestellt, das die GEF-Klasse `NetNode` erweitert. Die grafische Darstellung von Quest-Ports wird durch Unterklassen der Klassen `FigInternalPort` bzw. `FigExternalPort` realisiert, abhängig davon, ob es sich um einen Quest-Port vom Typ `SSDInternalPortView` oder `SSDExternalPortView` handelt. Von diesen beiden Klassen zur Darstellung von Quest-Ports gibt es noch Ausprägungen für die Repräsentation von Ein- und Ausgabeports. Im Editorausschnitt, der in Abbildung 7.15 gezeigt wird, sind dies die Klassen `FigInternalOutPort` (Figur 4) und `FigExternalInPort` (Figur 5). Die Klassen zur grafischen Darstellung von Ports sind von der Klasse `FigPort` abgeleitet. Im Graphenmodell werden diese Elemente durch die Klassen `InputPortNode` oder `OutputPortNode` repräsentiert, welche die Klasse `PortNode` erweitern. Die Klasse `NetNode` des Graphenmodells bildet die Basis für die Klasse `PortNode`. Ein Kanal wird durch die Klasse `FigChannel` an der Benutzerschnittstelle dargestellt. Das Element `ComponentEdge` im Graphenmodell ist von der GEF-Klasse `NetEdge` abgeleitet. Um die Figuren für eine Komponente und für einen Port sind jeweils Ports des Graphenmodells gelegt. Im Graphenmodell werden diese durch ein Objekt vom Typ `ConnectingPort` repräsentiert, welches die Klasse `NetPort` erweitert. An der Benutzerschnittstelle sind diese Ports für eine Komponente durch `FigRect` und für einen Quest-Port durch `FigCircle` dargestellt. Die Attribute dieser beiden Figuren sind so belegt, dass sie an der Benutzerschnittstelle durch die darüberliegenden, sichtbaren grafischen Objekte für Komponenten und Quest-Ports verdeckt sind. Diese Ports des Graphenmodells haben keine Entsprechung im Quest-Modell, sondern dienen lediglich als Verbindungspunkte für Kanten

des Graphenmodells. Zwischen einer Komponente und einem Port des Quest-Modells befindet sich jedoch keine Kante. Für die Darstellung im Graphenmodell sind jedoch Kanten für die Verbindung von Knoten erforderlich. Um die Repräsentierung im GEF-Rahmenwerk zu realisieren, wird die Kante vom Typ `HiddenEdge` eingeführt, die an der Benutzerschnittstelle durch `FigEdgeLine` dargestellt ist. Die Attribute dieses grafischen Objekts sind so belegt, dass die Kante nicht sichtbar ist.

Bei dieser Art der Abbildung zwischen Graphenmodell und den Figuren des Diagramms wird in der Editorebene nicht zwischen der Darstellung von Quest-Ports der internen oder externen Sicht unterschieden. Alle Figuren für die Darstellung von Ports sind von der Klasse `FigPort` abgeleitet. Im Graphenmodell sind Quest-Ports und Komponenten Knoten, die durch Kanten miteinander verbunden sind, unabhängig davon, ob es sich um einen Quest-Port der internen oder externen Sicht einer Komponente handelt. Damit wird der Implementierungsaufwand trotz der Einführung der Kante `HiddenEdge` reduziert. Jedoch unterscheidet der in GEF integrierte Layout-Algorithmus nicht, ob es sich bei einem Quest-Port um die Darstellung der internen oder externen Sicht auf eine Komponente handelt. Deshalb berücksichtigt das Layout-Verfahren nicht, ob eine Figur für einen Quest-Port ein Teil der Komponente ist oder nicht. Als Folge werden nur die beiden miteinander zu verbindenden Figuren betrachtet. Wird die Figur eines Quest-Ports der externen Sicht einer Komponente mit einem grafischen Objekt für die Darstellung eines Quest-Ports der internen Sicht durch einen Kanal verbunden, wird die Figur für die Komponente bei der Erstellung des Layouts außer Acht gelassen. Deshalb können Figuren für Ports unter den beschriebenen Umständen nicht durch eine Kante verbunden werden, die einen kürzesten Weg zwischen beiden Elementen darstellt, sondern die Figur für eine Komponente kreuzt. Beim Routing-Verfahren des Rahmenwerks handelt es sich um einen sehr einfachen Ansatz, der nur die Quelle und das Ziel einer Kante berücksichtigt und alle anderen Figuren des Editors außer Acht lässt. Diese Realisierung im Rahmenwerk erfordert eine erweiterte Implementierung eines geeigneten Layout-Verfahrens. Dabei kann auch das beschriebene Problem der Kreuzung von Kanälen mit Komponenten berücksichtigt werden. Dieser Layout-Algorithmus kann auch in die Ebene der Views integriert werden.

Element	Klasse der Editorebene	Klasse des Graphenmodells
1	<code>FigComponent</code>	<code>ComponentNode</code>
2	Die Darstellung des Ports ist ein Teil der Figur für die Komponente.	Im Graphenmodell muss nur die Figur für einen Port mit einem Port assoziiert werden.
3	<code>FigChannel</code>	<code>ComponentEdge</code>
4	<code>FigCircle</code>	<code>ConnectingPort</code>
5	<code>FigExternalInPort</code>	<code>InputPortNode</code>

Tabelle 7.5. Erläuterungen zu Abbildung 7.16 auf der nächsten Seite

Eine alternative Darstellung im Editor ist Abbildung 7.16 auf der nächsten Seite beschrie-

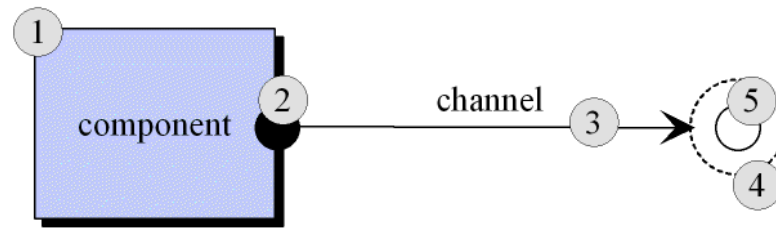


Abbildung 7.16. Alternative Umsetzung zwischen dem Graphenmodell des Rahmenwerks und den Editorelementen. Die gestrichelten Elemente sind nicht sichtbar, sondern dienen nur zur Veranschaulichung der Konstruktion der grafischen Darstellung.

ben. Die Erläuterungen zu dieser Abbildung sind in Tabelle 7.5 auf der vorherigen Seite zusammengefasst. Der Quest-Port einer Komponente fungiert nicht nur als Verbindungspunkt für eine Kante im Graphenmodell, sondern repräsentiert zugleich die Darstellung im Editor. Der Quest-Port ist fester Bestandteil des grafischen Objekts für eine Komponente. Die Zuständigkeit für die Platzierung der Ports verschiebt sich nun auf die Seite des Editorelements. Die Integration eines Layout-Verfahrens in die View-Ebene wird dadurch erschwert. Der Layout-Algorithmus des Rahmenwerks resultiert bei dieser Variante in keiner besseren Darstellung als bei der in Abbildung 7.15 auf Seite 86 beschriebenen. Im Editor muss zwischen zwei verschiedenen Figuren für Ports unterschieden werden: Solche, welche die externe Sicht auf eine Komponente darstellen und solche, die sich auf die interne Sicht beziehen. Im ersten Fall ist die Figur für einen Port Teil des grafischen Objekts für eine Komponente und damit in der Ebene des Graphenmodells kein eigener Knoten. Im zweiten Fall wird die Repräsentation des Quest-Ports im Graphenmodell durch einen eigenen Knoten realisiert. Damit steigt der Implementierungsaufwand, weil für die Realisierung zwei unterschiedliche Klassen zu erstellen sind.

Element	Klasse der Editorebene	Klasse des Graphenmodells
1	FigComponent	ComponentNode
2	Die Figuren für eine Komponente und ihre Quest-Ports werden zu einer Figurengruppe zusammengefasst.	Im Graphenmodell ist nur <i>ein</i> Knoten für diese Gruppierung erforderlich.

Element	Klasse der Editorebene	Klasse des Graphenmodells
3	FigCircle	ConnectingPort
4	FigInternalOutPort	OutputPortNode
5	FigChannel	ComponentEdge
6	FigExternalInPort	InputPortNode

Tabelle 7.6. Erläuterungen zu Abbildung 7.17 auf der nächsten Seite

Eine weitere Alternative ist in Abbildung 7.17 auf der nächsten Seite dargestellt. Die Tabelle 7.6 erläutert die in der Abbildung bezeichneten Elemente. Die Figuren für eine Komponente und ihre Quest-Ports werden zu einer Figurengruppe zusammengefasst. In der Implementierung des Rahmenwerks ist auch die in Abbildung 7.16 auf der vorherigen Seite beschriebene Variante eine Gruppierung der Elemente Komponente und Ports. Im Unterschied zu jener Implementierung wird bei dieser Variante die Realisierung vereinfacht, da für die interne und externe Sicht auf eine Komponente jeweils dieselben grafischen Objekte verwendet werden. Die Implementierung der nicht sichtbaren Kanten ist nicht nötig. Jedoch wird die Verantwortung für die Platzierung von Ports in das Editorelement für die Darstellung von Komponenten verlagert. Dadurch wird die Integration eines Layout-Verfahrens in die View-Ebene erschwert.

Bei den Alternativen, deren Darstellungen im Editor in den Abbildungen 7.16 auf der vorherigen Seite und 7.17 auf der nächsten Seite beschrieben sind, ist gleichermaßen nachteilig, dass die Gruppierung der Figuren für eine Komponente und ihre Quest-Ports zu *einem* grafischen Objekt darin resultiert, dass ein Port von einer Komponente bei einer Selektion im Editor alleine durch das Rahmenwerk nicht mehr unterschieden werden kann. Damit sind Operationen für die Änderung von Attributen oder das Entfernen von Komponenten oder Ports mit mehr Aufwand für das Erkennen der selektierten Figur verbunden. Die Funktionalität für die Platzierung von Figuren für Ports ist bei beiden Alternativen in die Ebene des Editors verlagert. Dadurch wird die Integration eines Layout-Verfahrens in die View-Ebene erschwert. Aus diesen Gründen ist die in Abbildung 7.15 auf Seite 86 dargestellte Abbildung zwischen den Elementen des Graphenmodells und den grafischen Objekten in der Implementierung des Prototyps realisiert. Dabei kann die Platzierung von Quest-Ports auf Komponenten in der Ebene der Views realisiert und die Integration eines Layout-Verfahrens in die View-Schicht ermöglicht werden.

5.3 View- und Editorelemente

Die Beziehungen zwischen den View- und Editorelementen für ein Systemstrukturdiagramm sind in Abbildung 7.18 auf Seite 91 dargestellt. In der untersten Zeile der Abbildung befinden sich die Elemente des Editors, die übrigen Zeilen stellen die Elemente der View-Ebene dar. Die Struktur der Beziehungen zwischen den View-Elementen ist durch die Meta-Views von Quest vorgegeben. Jedes View-Element hat seine Entsprechung in der Ebene des Editors. Dem Wurzelobjekt als Container für die enthaltenen View-Elemente entspricht das Editorfenster, welches bei folgenden Operationen benachrichtigt wird, um die aktuelle Zusammensetzung des

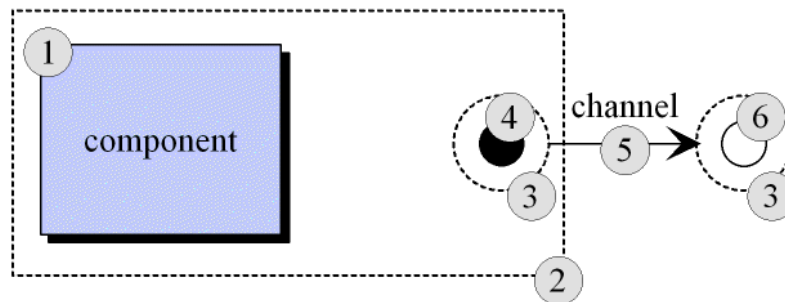


Abbildung 7.17. Eine weitere, alternative Umsetzung zwischen dem Graphenmodell des Rahmenwerks und den Editorelementen. Die gestrichelten Elemente sind nicht sichtbar, sondern dienen nur zur Veranschaulichung der Konstruktion der grafischen Darstellung.

Wurzelobjekts `SSDComponentRootView` zu reflektieren:

- Einfügen und Löschen einer Komponente
- Einfügen und Löschen eines Ports der internen Sicht einer Komponente
- Einfügen und Löschen eines Kanals zwischen zwei Ports
- Namensänderung der Komponente, die mit dem Wurzelobjekt assoziiert ist

Das grafische Objekt `FigComponent` für die Darstellung einer Komponente wird bei folgenden Operationen benachrichtigt:

- Einfügen und Löschen eines Ports der externen Sicht einer Komponente
- Namensänderung der Komponente
- Anpassung der Anzeige bei der Erstellung von Unterkomponenten. Dabei wird ein Symbol für die Dekomposition angezeigt. Dieses Symbol wird wieder entfernt, wenn alle Unterkomponenten gelöscht sind.
- Verschieben und Größenänderung einer Komponente. Dabei werden auch die Ports der Komponente benachrichtigt. Diese müssen ebenso in ihrer Position angepasst werden, wenn eine Komponente verschoben oder in ihrer Größe geändert wird.

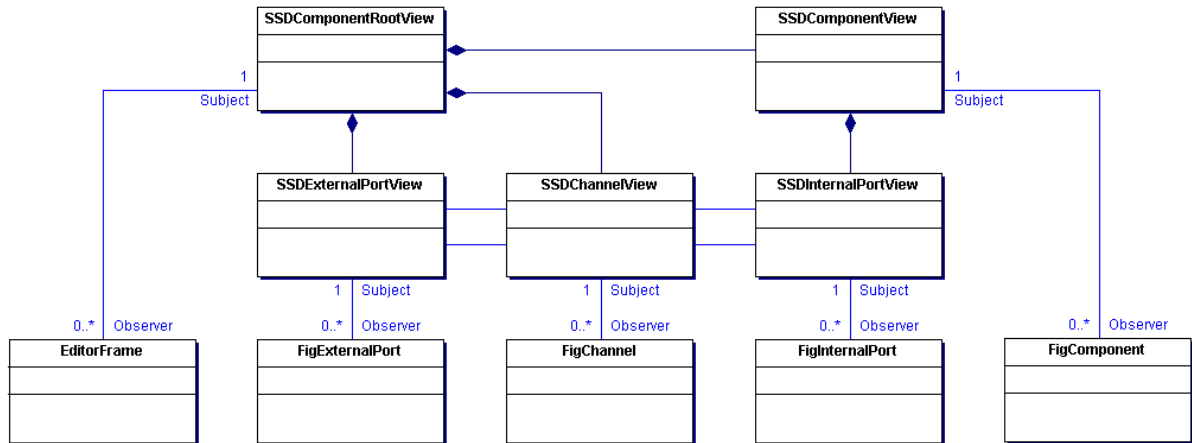


Abbildung 7.18. Die Beziehungen zwischen View- und Editorelementen für ein Systemstrukturdiagramm.

Beim View-Element für einen Kanal ist ein Objekt vom Typ `FigChannel` angemeldet, welches bei der Änderung der Modellattribute des Kanals benachrichtigt wird. Das Auslegen der Kanten ist in der Implementierung des Prototyps im Rahmenwerk realisiert.

Die beiden Klassen `FigExternalPort` und `FigInternalPort` werden jeweils durch Klassen für die Darstellung von Ein- und Ausgabeports erweitert. Sie sind aus Gründen der Übersichtlichkeit in Abbildung 7.18 weggelassen. Die Objekte für die Darstellung der Ports im Editor werden bei der Aktualisierung der Position des assoziierten View-Elements benachrichtigt.

5.4 Modell- und View-Elemente

Die Beziehungen zwischen den View- und Modellelementen ist in Abbildung 7.19 auf der nächsten Seite dargestellt. Jedes View-Element ist bei seinem Modellelement als `ModelChangeListener` angemeldet. Dadurch werden die View-Elemente über Veränderungen am Modell benachrichtigt. Beim Öffnen eines neuen SSD-Editorfensters meldet sich das Wurzelobjekt `SSDComponentRootView` bei seinem Modellelement an. Das View-Element für eine Komponente `SSDComponentView` ist dann bei seinem Modellelement registriert, wenn im Editorfenster Komponenten eingefügt werden. Bei einem Port sind sowohl `SSDInternalPortView` als auch `SSDExternalPortView` registriert, weil beide unterschiedliche Sichten auf diesen Port darstellen.

6 Schwierigkeiten bei der Implementierung

Während der Evaluierung der Software zur Visualisierung und Modifikation von Graphen zeigt sich, dass das Rahmenwerk GEF einen angemessenen Umfang besitzt, um als Basis für einen grafischen Editor eingesetzt zu werden. Jedoch lässt die Phase der Implementierung erkennen, dass die realisierte Funktionalität des Rahmenwerks geringer als erwartet ist. Beispielsweise

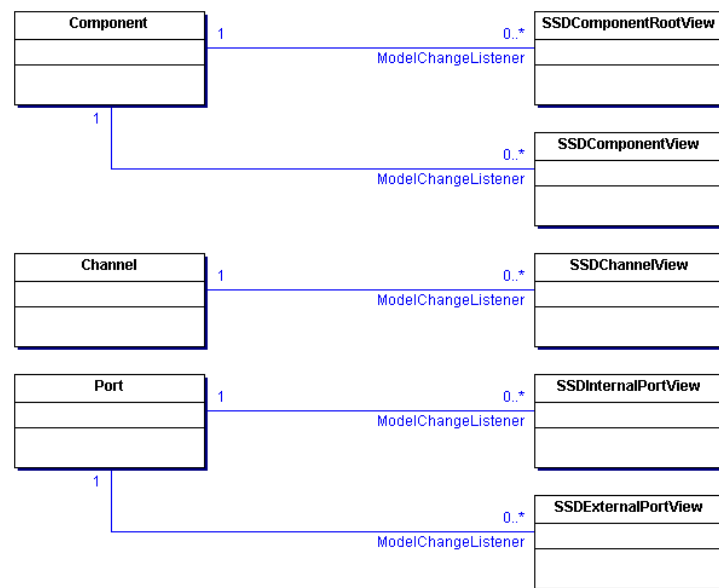


Abbildung 7.19. Die Beziehungen zwischen den View- und ihren assoziierten Modellelementen. Die Relationen zwischen den View-Elementen sind aus Gründen der Übersichtlichkeit nicht angegeben.

sind die Möglichkeiten zur Realisierung einer ansprechenden Benutzeroberfläche nicht in GEF implementiert, sondern erst im UML-Modellierungswerkzeug ArgoUML [Arg02] hinzugefügt, das auf diesem Rahmenwerk basiert. Zur Implementierung dieser Funktionalität ist ein zusätzlicher Implementierungsaufwand für die Erweiterung des Editors erforderlich.

GEF implementiert keine Funktionalität für Undo- und Redo-Operationen. Für einen Editor mit geeigneter Mensch-Maschine-Schnittstelle ist diese Funktionalität jedoch angemessen. Die Realisierung dieser Funktionalität erfordert zusätzlichen Implementierungsaufwand im Editor und für die Unterstützung durch die View- und Modellebene.

Die Implementierung des Prototyps berücksichtigt nicht die Realisierung von Sequenzdiagrammen. Ob diese Funktionalität bereits durch GEF implementiert ist, oder ein Teil der Implementierung des Werkzeugs ArgoUML darstellt, ist aus der Implementierung des prototypischen Editors nicht ersichtlich. Dies erfordert eine weitere Evaluierung des Rahmenwerks.

Rahmenwerke zeichnen sich durch Einhängpunkte (engl. *hooks*) für die Einbindung von anwendungsspezifischer Funktionalität aus. Das eingesetzte Rahmenwerk GEF bietet zu wenige Einhängpunkte. Bei folgenden Aktionen können benutzerdefinierte Methoden aufgerufen werden:

- Einfügen eines Knotens
- Einfügen oder Entfernen einer Kante
- Überprüfung, ob zwei Ports miteinander verbunden werden können

Es fehlen Einhängpunkte beim Entfernen eines Knotens und bei Aktionen, welche die Anzeigeeigenschaften verändern. Damit ist es nötig, das Rahmenwerk anders als vorgesehen zu

verwenden. Denn die bereits implementierte Funktionalität beim Entfernen von Knoten oder Ändern von Anzeigeeigenschaften wird in der prototypischen Implementierung aus dem Rahmenwerk extrahiert und Teil der Anwendung. Dadurch können Inkompatibilitäten bei der Verwendung einer neuen Version des Rahmenwerks auftreten, die umfangreichere Änderungen an den implementierten Klassen erfordern. Als Folge steigt der Implementierungsaufwand.

Der durch das Rahmenwerk vorgegebene Layout-Algorithmus ist nicht ausreichend, da nur das Ziel und die Quelle von Kanten bei der Berechnung des Layouts berücksichtigt werden. Idealerweise wird das Layout-Verfahren in die View-Ebene integriert. Denn in dieser Schicht befinden sich die Informationen über die Anzeigeeigenschaften der Modellelemente. Einschränkungen bezüglich des Layouts können in dieser Schicht unabhängig vom verwendeten Editor beschrieben werden. Bei der Integration eines Layout-Verfahrens in die View-Ebene ist die Extraktion des Layout-Verfahrens des Rahmenwerks erforderlich. Dadurch können wiederum Inkompatibilitäten bei der Verwendung einer weiterentwickelten Version des Rahmenwerks entstehen. Die Möglichkeit zur Einbindung von externen Layout-Algorithmen in das Rahmenwerk fehlt.

Insgesamt betrachtet ergeben sich bei der Verwendung des Rahmenwerks GEF also einige Implementierungsschwierigkeiten. Diese werden durch eigene Ergänzungen am Rahmenwerk gelöst, wodurch der Implementierungsaufwand steigt. Da die Kopplung zwischen dem Rahmenwerk und Quest sehr eng ist, muss an den beschriebenen Stellen auf Kompromisse eingegangen werden. Das Rahmenwerk wird dadurch zwar nicht an allen Stellen so wie vorgesehen verwendet, ermöglicht aber die Realisierung von Erweiterungen für die Anforderungen an einen Quest-Editor.

7 Einhaltung von Konsistenzbedingungen

Besonders beim Löschen von Modellelementen können Konsistenzbedingungen verletzt werden. Dazu wird in der Implementierung vor der Durchführung einer Löschoperation überprüft, ob die Aktion unter Einhaltung der geforderten Konsistenzbedingungen [HPS02, HS01] des Modells ausgeführt werden kann. Beim Löschen beispielsweise einer Komponente wird überprüft, ob diese Ausführung der Aktion zu einem konsistenten Zustand führt. Dabei wird sichergestellt, dass die Komponente weder Ports noch Unterkomponenten besitzt. Trifft eine der beiden Eigenschaften zu, kann die Komponente nicht entfernt werden. Ports können nur entfernt werden, wenn sie keine Kanäle besitzen. Beim Einfügen eines Kanals wird überprüft, ob der betreffende Eingabeport bereits einen schreibenden Kanal besitzt. Wenn dies der Fall ist, kann der Kanal nicht eingefügt werden. Die Überprüfung dieser Konsistenzbedingungen sind in der View-Ebene realisiert. Vor dem Löschen einer Komponente durch die Methode `removeSubComponents()` der Klasse `SSDComponentRootView` wird durch den Aufruf der Methoden `hasSubComponents()` und `hasPorts()` auf dem Modellelement `Component` überprüft, ob die Komponente keine Ports und Unterkomponenten besitzt und folglich gelöscht werden kann. Der Aufrufer der Methode wird durch einen Rückgabewert vom Typ `ReturnCode` über den Erfolg oder die Ursache des Scheiterns der Aktion benachrichtigt. In gleicher Weise wird beim Entfernen von Ports der internen und externen Sicht auf eine Komponente überprüft, ob mit den Ports Kanäle verbunden sind. Die Einhaltung der Konsistenzbedingungen beim Einfügen von Kanälen erfordert ebenso eine Überprüfung der durchzuführenden

Aktion.

Weil Systemstrukturdiagramme keine vom übrigen AUTOFOCUS-Metamodell isolierten Teile darstellen, können beim Löschen von Teilen dieses Diagramms auch andere Teile des AUTOFOCUS-Metamodells betroffen sein. Aus Abbildung 2.1 auf Seite 6 ist ersichtlich, dass mit dem zentralen Modellelement `Component` z.B. die Elemente `MSC` für ein Sequenzdiagramm oder `Automaton` für einen Zustandsübergangdiagramm assoziiert sind. Komponenten können nicht einfach gelöscht werden, weil bei diesem Vorgang auch andere Teile des Metamodells betroffen sind.

In der Implementierung des prototypischen Editors ist die Funktionalität für die Überprüfung von Konsistenzbedingungen wie beschrieben in der Ebene der View-Elemente realisiert. In dieser Ebene werden die Konsistenzbedingungen auf den betreffenden Modellelementen abgefragt. Je nach Entwurfsentscheidung kann die Überprüfung der Bedingungen auch in andere Schichten verlagert werden. Ist die Funktionalität der Modell- und View-Operationen in der Ebene des Editors realisiert, sind auch die Konsistenzüberprüfungen in dieser Schicht durchzuführen.

Die Überprüfung der Korrektheit von Modelloperationen in Bezug auf die Konsistenzbedingungen wird durch die Einführung der Sprache ODL (Operation Description Language) des AQUA-Rahmenwerks [Sch02] unterstützt. Ziel dieser Sprache ist die Definition von Operationen auf dem AUTOFOCUS-Metamodell, unabhängig von der tatsächlichen Implementierung des Modells. Durch diese Operationen werden Konsistenzüberprüfungen und Änderungen am Modell unterstützt.

8 Integration in die automatische Generierung

Die Erweiterungen für die Implementierung des Prototyps umfassen die Bereiche View-Klassen und Rahmenwerk. Diese für das erarbeitete Editorenkonzept nötigen Erweiterungen an den bisherigen Quest-Klassen der View-Ebene sind in Java-Schnittstellen formuliert, um die Übertragung des Konzepts auf andere Editoren zu erleichtern. Dennoch sollen die Erweiterungen langfristig fester Bestandteil der View-Klassen werden und in die automatische Generierung integriert werden.

Die Erweiterungen an den View-Klassen umfassen Attribute, Methoden, zu implementierende Schnittstellen und den Import von vollständig qualifizierten Namen, die in der automatischen Generierung berücksichtigt werden müssen. Neben der Beschreibung des AUTOFOCUS-Metamodells sind in der Datei `metamodel.dbi` auch die View-Klassen definiert. Die für die automatische Generierung erforderlichen Ergänzungen in dieser Datei werden im Folgenden erläutert.

Die View-Klassen erfordern zunächst die Importierung von vollständig qualifizierten Klassen- und Paketnamen. Alle nötigen Paket- und Klassennamen werden in der `dbi`-Datei durch die Anweisung `<import> <Paketname>` in die Definition der jeweiligen View-Klasse eingeschlossen. Der Überblick über die zu importierenden Namen sind in Tabelle 7.7 auf der nächsten Seite zusammengefasst.

View-Klasse	Importanweisungen
SSDComponentRootView	<code>quest.util.*</code> <code>quest.metamodel.views.SSDView.ext.*</code> <code>quest.metamodel.views.SSDView.ext.Action</code> <code>java.awt.Point</code>
SSDComponentView	<code>quest.metamodel.views.SSDView.ext.*</code> <code>quest.util.ChangeableModel</code> <code>quest.metamodel.views.SSDView.ext.Action</code>
SSDPortView	<code>quest.util.*</code> <code>quest.metamodel.views.SSDView.ext.*</code>
SSDExternalPortView	<code>java.awt.geom.Point2D</code> <code>quest.util.ModelChangeEvent</code> <code>quest.metamodel.views.SSDView.ext.Action</code>
SSDInternalPortView	<code>java.awt.geom.*</code> <code>quest.util.ModelChangeEvent</code> <code>quest.metamodel.views.SSDView.ext.Action</code>
SSDChannelView	<code>quest.util.ModelChangeEvent</code> <code>quest.metamodel.views.SSDView.ext.*</code> <code>quest.util.ChangeableModel</code> <code>quest.metamodel.views.SSDView.ext.Action</code>

Tabelle 7.7. Überblick über vollständig qualifizierten Namen, die von den View-Klassen importiert werden.

Neben der Integration der vollständig qualifizierten Namen ist die Implementierung von zusätzlichen Schnittstellen durch die View-Klassen erforderlich. Die Angabe dieser Schnittstellen erfolgt in der `dbi`-Datei im Kopf der Definition der View-Klasse. Die View-Klassen und die zu implementierenden Schnittstellen sind in Tabelle 7.1 auf Seite 65 dargestellt.

Die für die Erweiterungen der View-Klassen benötigten Attribute sind in den Java-Schnittstellen für die Hilfsfunktionalität der View-Klassen deklariert. Deshalb müssen die Attribute nicht erneut in der `dbi`-Datei angegeben werden. Diese Schnittstellen und die implementierenden View-Klassen sind in Tabelle 7.1 auf Seite 65 zusammengefasst.

Die Methoden `attachObserver()` und `detachObserver()` sind in den Klassen `SSDComponentRootView`, `SSDComponentView`, `SSDChannelView` und `SSDPortView` in gleicher Weise implementiert. Diese Methoden werden daher in Methoden-Dateien vom Typ `meth` definiert und in die jeweiligen View-Klassen beispielsweise durch die Anweisung `/* Methodfile Views/observers.meth */` in der `dbi`-Datei eingebunden. Tabelle 7.8 auf der nächsten Seite fasst die Implementierung gleicher Methoden zusammen.

Die Methode `notifyObservers()` ist in den Klassen `SSDChannelView`, `SSDComponentView` und `SSDPortView` bereits implementiert. Für die Erweiterung des modellbasierten Editorenkonzepts ist die Implementierung dieser Methoden durch eine andere Realisierung ausgetauscht. In der `dbi`-Datei muss daher die Änderung vorgenommen werden, welche die vorherige Implementierung dieser Methode mit der aktuellen austauscht.

Die Methode `attributeChanged()` unterscheidet sich in der Implementierung in den Klassen `SSDExternalPortView`, `SSDInternalPortView` und `SSDChannelView` nur durch den Typ des referenzierten Modellelements. Deshalb bietet sich hier die Definition dieser Methode in einer Pattern-Datei vom Typ `pat` an, in welcher der Parameter bei der Generierung durch das jeweilige konkrete Modellelement ersetzt wird. Mit der Notation `/* Patternfile <method>.pat */` in der `dbi`-Datei wird der Inhalt dieser Pattern-Datei in die jeweiligen View-Klassen eingebunden.

Die übrigen, nicht erwähnten Methoden, die in den View-Klassen für die Erweiterungen des modellbasierten Editorkonzepts erforderlich sind, unterscheiden sich in ihrer Implementierung derart, dass eine gemeinsame Integration für mehrere Klassen nicht möglich ist. Die Implementierungen dieser Methoden sind daher in Methoden-Dateien enthalten, die in die View-Klassen durch die Notation `/* Methodfile Views/<method>.meth */` in der `dbi`-Datei integriert werden.

Die in Tabelle 7.8 auf der nächsten Seite angegebenen Methoden unterscheiden sich in der gegenwärtigen Implementierung des Prototyps nicht. Bei ihnen wird der Vorteil der Einbindung von Methodendefinitionen in mehrere Klassen durch eine Methodenimplementierung in *einer* Methoden-Datei ausgenützt. Der Implementierungsaufwand wird dadurch reduziert. Die Implementierungen der Methoden `modelConsistent()`, `modelInconsistent()` und `modelRemove()` können sich jedoch in einer Ausbaustufe des Editors unterscheiden. Bei einer Änderung, die unterschiedliche Implementierungen dieser Methoden erfordert, ist der Vorteil der Reduzierung des Implementierungsaufwands bei diesen Methoden nicht mehr gegeben.

Die Ausführungen zeigen, dass in der gegenwärtigen Realisierung des Prototyps durch die Integration der Erweiterungen in die automatische Generierung von Programmtext der Implementierungsaufwand reduziert wird. Durch die Einbindung von Methoden in mehrere View-Klassen ist die gesonderte, manuelle Implementierung dieser Methoden in jeder einzelnen Klasse nicht erforderlich. Der zusätzliche Aufwand für die Integration in die automatische Generierung wird durch die Reduktion des Implementierungsaufwands kompensiert.

Methoden	View-Klassen
notifyObservers()	SSDChannelView SSDComponentRootView SSDComponentView SSDPortView
objectInserted()	SSDChannelView SSDPortView
modelRemove()	SSDChannelView SSDComponentRootView SSDComponentView SSDPortView
objectRemoved()	SSDChannelView SSDPortView
modelConsistent()	SSDChannelView SSDComponentRootView SSDComponentView SSDPortView
modelInconsistent()	SSDChannelView SSDComponentRootView SSDComponentView SSDPortView
attributeChanged()	SSDExternalPortView SSDInternalPortView

Tabelle 7.8. Überblick über Methoden, die in den angegebenen View-Klassen in gleicher Art implementiert sind.

9 Bestimmung von Implementierungsentscheidungen durch externe Vorgaben

Die Vorgaben des verwendeten Rahmenwerks und der Quest-Klassen beeinflussen die Implementierung der Anwendung. Aus dem Bereich von Quest entsteht die Forderung nach der Wiederverwendung der existierenden Mechanismen zur Kommunikation von Änderungen am Modell. Andere Anwendungen wie z.B. der Validator [Val02] kommunizieren mit dem zentralen Systemmodell. Änderungen werden durch Ereignisse an die registrierten Objekte propagiert. Dazu registrieren sich Objekte als `ModelChangeListener` bei Modellelementen. Änderungen werden durch Objekte vom Typ `ModelChangeEvent` an die registrierten Objekte gesendet. Dieser Mechanismus zur Kommunikation von Änderungen wird auf die View-Elemente übertragen, die sich als Listener vom Typ `ModelChangeListener` bei ihren assoziierten Modellelementen anmelden. Damit bleibt der bisher in Quest eingesetzte Mechanismus zur Propagierung von Modelländerungen erhalten. Die bestehenden Anwendungen werden nicht beeinflusst.

Die frühere Implementierung von Quest zeichnet die View-Klassen als Objekte vom Typ `Subject` des Observer-Entwurfsmusters aus. Dieses vorhandene Entwurfsmuster wird für das modellbasierte Editorenkonzept übernommen und angepasst. Bei einer Änderung einer Eigenschaft eines beliebigen View-Elements wird in der früheren Implementierung sein Wurzelement aufgefordert, seine Observer-Objekte zu benachrichtigen. In der Implementierung des Prototyps bestimmen die Beziehungen zwischen den View-Elementen die Kopplung zwischen den View- und Editorelementen. Aus der hierarchischen Struktur mit dem Wurzelement als Wurzel folgt die Abbildung in die Editorebene. Jedes Element der View-Ebene findet nun in der Editorebene eine Entsprechung.

In der aktuellen Implementierung von `AUTOFOCUS` werden Ports beim Einfügen neuer Kanäle automatisch eingefügt. Die Einführung von Operationen, die das explizite Einfügen neuer Ports erlauben, ist in den Erweiterungen für das modellbasierte Editorenkonzept realisiert. Diese Entscheidung verbessert die Verständlichkeit und die Einheitlichkeit der Schnittstelle, weil die angebotenen Modelloperationen nur jeweils eine Aufgabe ausführen und nicht mehrere kombinieren.

Der Einsatz eines Layout-Verfahrens ist im Rahmenwerk implementiert. Damit kann das Auslegen der Kanäle vom Rahmenwerk realisiert werden. In der View-Schnittstelle befindet sich deshalb keine Funktionalität, die sich auf das Layout bezieht. Durch die Realisierung im Rahmenwerk kann die Schnittstelle zwischen View und Editor in ihrem Umfang klein gehalten werden. Bei der Integration eines Layout-Verfahrens in die View-Ebene würde beim Verschieben oder Einfügen neuer grafischer Objekte eine Berechnung des Layouts in der View-Ebene initiiert werden und damit bei jeder Änderung der Anzeige eine Operation aus der Schnittstelle zum Editor aufgerufen werden. Außerdem würde dies den Implementierungsaufwand in der Editorebene erhöhen. Die Layout-Vorgaben durch die View-Ebene müssten in Interaktionspunkte auf den Figuren umgesetzt werden, welche die Kanäle repräsentieren. Weil das Rahmenwerk die Berechnung eines einfachen Layout-Verfahrens implementiert, ist die Erweiterung der View-Ebene für das Layout nicht erforderlich.

10 Abdeckung der Anforderungen

10.1 Funktionalität des Editors

Das Ziel der Implementierung des Prototyps ist die Validierung des erarbeiteten Konzepts für modellbasierte Editoren. Bei Operationen wird zwischen Aktionen unterschieden, die Modellattribute verändern, und solchen, die ausschließlich die Eigenschaften von View-Elementen modifizieren. Deshalb werden in der Implementierung beide Arten von Operationen berücksichtigt. Folgende implementierte Operationen verdeutlichen die Ablaufstruktur bei der Veränderung von Modelleigenschaften:

- Einfügen und Entfernen der Modellelemente Komponente, Port und Kanal
- Belegung des Attributs für den Namen der Elemente Komponente, Port und Kanal
- Belegung des Attributs für den Kommentar der Elemente Komponente und Port

Die Operationen, welche die Veränderung von View-Attributen demonstrieren, umfassen die Größenänderung und das Verschieben von Figuren, die Komponenten im Editor repräsentieren. Dabei ist auch die Anpassung der Ports berücksichtigt.

Die Implementierung realisiert neben den Modell- und View-Operationen noch weitere Aktionen. Durch das Kontextmenü einer Figur oder durch einen Menüeintrag können Modellattribute verändert werden. Die Navigation in umfangreichen Diagrammen wird durch die Zoom-Funktion unterstützt. Zur Verbesserung der Übersichtlichkeit können in der Editorebene die textuellen Beschreibungen von Figuren ein- oder ausgeblendet werden.

10.2 Vergleich zwischen Anforderungen und Implementierung

In Tabelle 7.9 auf Seite 102 werden die in den Abschnitten 2 auf Seite 37 bzw. 3 auf Seite 38 erläuterten Rahmenbedingungen und Anforderungen an den Editor mit der realisierten Funktionalität verglichen. Die in Abschnitt 4 auf Seite 42 beschriebenen Anforderungen folgen aus den Anforderungen des Editors und werden in der folgenden Tabelle nicht separat aufgeführt.

Anforderung	Realisierung
Realisierung in Java mit der Swing-Bibliothek	Der Prototyp ist in der Sprache Java realisiert. Das Rahmenwerk basiert auf der Verwendung von Swing-Komponenten.
Rahmenwerk GEF	Der Prototyp basiert auf dem Einsatz des Rahmenwerks GEF.
Quest-Modell	Durch den Editor werden Modellelemente visualisiert und modifiziert.
Quest-Browser	Der Editor ist als eigenständige Anwendung realisiert und wird durch den Browser geöffnet.
Größe der Zeichenfläche	Die Größe der Zeichenfläche passt sich automatisch dem Umfang des Diagramms an.

Anforderung	Realisierung
Beenden des Editors	Der Editor wird geschlossen, ohne jedoch den Benutzer nach der Speicherung des bearbeiteten Dokuments zu fragen.
Verschieben von grafischen Objekten	Die Figuren für die Komponenten und Ports können verschoben werden. Ports der externen Sicht auf eine Komponenten werden in ihrer Position angepasst, wenn ihre Komponente verschoben wird.
Editieroperationen	Die Operationen zum Einfügen und Entfernen von Modell- und View-Elementen für Komponenten, Ports und Kanäle sind implementiert.
Erstellen von grafischen Objekten	Die View-Elemente für Komponenten, Ports und Kanäle werden im Editor durch Figuren visualisiert.
Öffnen und Erstellen eines Diagramms	Durch den Browser wird ein neues bzw. existierendes Diagramm im Editor geöffnet.
Kopieren, Einfügen und Ausschneiden	Diese Funktionalität ist im Prototyp nicht realisiert. Die Implementierung von Operationen zum Kopieren, Einfügen und Ausschneiden von Figuren ist im Rahmenwerk GEF implementiert. Zu realisieren ist die Einbettung dieser Operationen in Quest.
Auto-Scrolling	Wird durch das Rahmenwerk implementiert.
Größenänderung der grafischen Objekte	Die Figuren für Komponenten können durch den Benutzer in ihrer Größe angepasst werden. Die Ports einer Komponente werden in der Implementierung der Größenänderung berücksichtigt.
Drucken	Diese Funktionalität ist nicht implementiert, wird aber durch das Rahmenwerk GEF realisiert. Es fehlt die Integration in den erstellten Prototyp.
Tooltips	Die Implementierung von Tooltips wird durch das Rahmenwerk realisiert. Sie zeigen in der Realisierung des Prototyps den Namen des Modellelements an.
Kontextmenüs	Kontextmenüs der grafischen Objekte erlauben die Modifikation von Eigenschaften der über View-Elemente assoziierten Modellelemente. Bei Komponenten und Ports können die Attribute für den Namen und Kommentar verändert werden, bei Kanälen kann der Name mit einem neuen Wert belegt werden. Für alle Elemente können die Beschriftungen im Editor verborgen werden. Durch das Kontextmenü können die Elemente Komponente, Port und Kanal entfernt werden. Die beschriebene Funktionalität ist auch durch Menüeinträge ausführbar. Die Implementierung von GEF erlaubt die zusätzliche Verschiebung von Figuren in den Vorder- oder Hintergrund.
Änderung der Modellattribute	Modellattribute können auch direkt im Editorfenster geändert werden. Bei einem Doppelklick auf eine Figur für einen Kanal oder eine Komponente kann das Attribut für den Namen des Modellelements direkt verändert werden.
Hilfesystem	Diese Funktionalität ist nicht realisiert. Das in [Wil02] beschriebene modellbasierte Hilfesystem ist in den prototypischen Editor zu integrieren.

Abdeckung der Anforderungen

Anforderung	Realisierung
Undo- und Redo-Operationen	Die Operationen für Undo und Redo sind nicht implementiert. Diese Funktionalität fehlt im Rahmenwerk, zusätzlich ist eine Einbettung dieser Operationen in Quest erforderlich.
Exportieren in EPS, JPEG und QML	Das Exportieren von Diagrammen ist im Rahmenwerk für die Formate EPS, PS, SVG, PGML und GIF realisiert. Es fehlt die Integration dieser Funktionalität in den Editor. Das Speichern von Diagrammen im Format QML wird durch den Browser realisiert. Die über die Funktionalität des Prototyps hinausgehenden Anforderungen erfordern Erweiterungen am QML-Format, weil die bisherige View-Schnittstelle für die Persistenz von Daten nicht vollständig ist.
Zoom	Im Editor wird die Navigation in umfangreichen Diagrammen durch eine Zoom-Funktion unterstützt.
Suchfunktion	Diese Funktionalität ist nicht realisiert.
Laden und Speichern	Diese Operationen werden durch den Browser realisiert. Weil die bisherige View-Schnittstelle für die Persistenz von Daten nicht vollständig ist, sind für weitere Anforderungen Erweiterungen am QML-Format nötig.
Mehrbenutzerfähigkeit	Die Unterstützung für mehrere Benutzer ist nicht implementiert.
Figuren für Komponenten, Kanäle und Ports	Alle geforderten Elemente werden durch Figuren im Editor visualisiert. Durch Tooltips und Kontextmenüs können Werte für die Belegungen von Modellattributen angezeigt werden. Die Figuren können erstellt, geändert und entfernt werden.
Layout-Algorithmen	Das Rahmenwerk realisiert ein sehr einfaches Verfahren zur Darstellung von Kanälen, welches für ansprechende Ergebnisse erweitert werden muss.
Auswirkungen auf das Modell	Durch den Editor werden die Änderungen am Modell ausgelöst.
Propagierung von Modelländerungen	Die Änderungen am Modell werden an alle interessierten Objekte propagiert.
Drag-and-Drop-Unterstützung	Diese Funktionalität ist nicht realisiert.
Navigation in hierarchischen Komponenten	Die Navigation in geschachtelten Komponenten erfolgt im Browser.
Einhaltung von Syntaxregeln	Ist im Prototyp nicht implementiert. Das Rahmenwerk bietet Unterstützung für den Aufruf von Methoden zur Überprüfung eines zulässigen Verbindens zweier Knoten durch eine Kante.
Verändertes Metamodell	Bei Änderungen am AUTOFOCUS-Metamodell muss nicht die gesamte Anwendung geändert werden, sondern es müssen lediglich die Zugriffsstrukturen zwischen den Schichten angepasst werden.

Anforderung	Realisierung
Dokumentation durch UML	Das Design und typische Ablaufstrukturen sind durch UML beschrieben.
Javadoc-Kommentare	Im Programmtext sind Javadoc-Kommentare integriert.
Änderungen an Modell- und View-Klassen	Die Erweiterungen für das modellbasierte Editorenkonzept betreffen ausschließlich die View-Klassen von Quest. Diese Erweiterungen sind in Abschnitt 2 auf Seite 65 beschrieben und in Java-Schnittstellen formuliert.
Änderungen und Erweiterungen an GEF	Die in Anhang 3 auf Seite 135 beschriebenen Änderungen am Rahmenwerk sind für die enge Kopplung mit Quest nötig.

Tabelle 7.9. Vergleich zwischen den gestellten Anforderungen und der realisierten Funktionalität.

Synchronisationsmechanismen

Bei Modelloperationen sind Mechanismen für die Einhaltung der Konsistenzbedingungen notwendig. Außerdem ist im Mehrbenutzer- und Mehrprogrammbetrieb die Synchronisation von Zugriffen auf Modellelementen zu gewährleisten. Im Mehrprogrammbetrieb arbeiten mehrere Programme gleichzeitig auf dem zentralen Systemmodell. Beispielsweise wird das Modell durch einen Editor bearbeitet, während ein Programm zur Verifikation lesend auf das Modell zugreift. Zur Einhaltung konsistenter Daten dürfen auf Modellelementen keine gleichzeitigen Lese- und Schreibzugriffe erlaubt sein.

Bei der Abarbeitung von Modelloperationen ist beim Mehrbenutzerbetrieb auch die Gewährleistung der Atomarität erforderlich. Beispielsweise ist zur Einhaltung der Konsistenzbedingungen beim Löschen einer Komponente zu überprüfen, dass die zu entfernende Komponente keine Unterkomponenten besitzt. In der Zeit zwischen der Überprüfung und der Ausführung der eigentlichen Operation könnte jedoch ein anderer Benutzer eine Unterkomponente anlegen. Wird die erste Operation nun fortgeführt, tritt ein inkonsistenter Zustand auf. Das Beispiel zeigt, dass Modelloperationen in atomarer Weise ausgeführt werden müssen, um die Konsistenz zu gewährleisten.

Finden parallele Zugriffe auch auf View-Elementen statt, muss die Zugriffskontrolle auf das Systemmodell auf die View-Ebene übertragen werden. In den folgenden beiden Abschnitten werden Möglichkeiten für die Realisierung von Mehrbenutzerbetrieb dargestellt. Lösungsansätze werden durch Sperr- und Transaktionskonzepte angeboten.

1 Shared Whiteboard

Bei einem *Shared Whiteboard* können mehrere Benutzer gleichzeitig auf ein Modell zugreifen und es verändern. Dabei werden alle Änderungen, die ein Benutzer durchführt, an alle anderen Benutzer propagiert. Um die Konsistenz zwischen den verschiedenen Operationen der Benutzer zu synchronisieren, sind Sperren auf den Modellobjekten nötig. Hier ist es wichtig, die *Granularität* der Sperren möglichst fein zu halten, um den Eindruck der Benutzer für eine gleichzeitige Bearbeitung zu gewährleisten.

Bei der Realisierung durch Sperren sind Zugriffe auf das gesperrte Objekt für die Dauer der Operation nur für einen Benutzer erlaubt. Dabei ist die unterschiedliche Granularität der Sperren zu berücksichtigen. Eine Sperre allein auf dem betreffenden Modellelement ist nicht ausreichend. Denn das AUTOFOCUS-Metamodell ist hierarchisch strukturiert. Dementsprechend ist ein hierarchisches Sperrsystem [Hah01] notwendig. Dabei ist das Systemmodell als Baum dargestellt, bei dem die Knoten durch Modellklassen und die Relationen zwischen den Modellelementen durch Kanten repräsentiert werden. Unterschieden werden drei Arten von Sperren:

- NL (engl. *no lock*): Der Knoten ist nicht mit einer Sperre belegt.

- X (engl. *exclusive*): Der Knoten ist von allen Objekten außer dem sperrenden nur lesbar. Mit X sind nur Objekte vom Typ `Component` sperrbar und alle weiteren Modellelemente, die in der Modellhierarchie oberhalb liegen. Außerdem muss die Bedingung erfüllt sein, dass ein Editorfenster zur Bearbeitung dieser Komponente geöffnet ist.
- IX (engl. *intention exclusive*): Der Knoten ist nicht gesperrt, in der Hierarchie weiter unten befindet sich aber mindestens ein Knoten mit der Sperre X.

Ein Knoten kann dabei auch mit mehreren Sperren versehen werden. Das Sperren und die Freigabe der Knoten erfolgt in der Hierarchie von unten nach oben:

- Bevor ein Knoten mit X gesperrt werden kann, müssen alle Knoten in der Hierarchie oberhalb mit der Sperre IX belegt werden.
- Ein Knoten kann nicht mit X gesperrt werden, falls er von einem anderen Prozess mit der Sperre IX versehen ist.
- Die Sperren werden von unten nach oben wieder freigegeben.

Um diese Anforderungen umzusetzen, werden die Modellklassen mit einem Attribut für ihre Sperre und mit Methoden zum Sperren und Freigeben erweitert. Das Konzept des Monitors bietet einen Ansatz zur Realisierung von Sperren. Eine Eigenschaft dieses Konzepts ist der ausschließliche Aufenthalt eines Prozesses in dem Block, der durch den Monitor überwacht wird. Im Monitor befindet sich die gesamte Operation zur Änderung des Modells, beispielsweise beim Einfügen einer Komponente die Methode `addSubComponents()` der Klasse `Component`. In dieser Methode werden auch die nötigen Sperren in der Modellhierarchie erworben. Durch den Einsatz von Sperren treten jedoch folgende Probleme auf:

- Durch einen Programmierfehler wird eine Sperre nicht mehr freigegeben. Damit wird der weitere Programmablauf für alle anderen Benutzer mit Zugriff auf das gesperrte Objekt beeinflusst.
- Kommt es bei der Ausführung eines Prozesses, der Sperren freigibt und belegt, zu einem Fehler, der die Ausführung abbricht, wird der Programmablauf beeinflusst, da möglicherweise gehaltene Sperren nicht mehr freigegeben werden.
- Bei den Maßnahmen zur Synchronisation kann es zu einer Verklemmung zwischen zwei Prozessen kommen.
- Ein Programm mit Fehlern, die zu einer Verklemmung führen können, ist für einen Kunden nicht akzeptabel. Die Fehlersuche in einem Programm mit Sperren ist jedoch aufwändig.

Die Programmiersprache Java bietet das Konzept des Monitors. Mit dem Schlüsselwort `synchronized` werden dazu die Methoden ausgezeichnet, die für den Zugriff durch mehrere leichtgewichtige Prozesse synchronisiert werden. Beispielsweise beim Einfügen einer Komponente wird die Methode `addSubComponents()` durch das Schlüsselwort `synchronized`

zu einem Monitor. Die exklusive Ausführung der Methode ist damit gewährleistet. Programmierfehler, die das Sperren der Methode an sich betreffen, werden durch das Monitorkonzept verhindert.

Alternativ zu den Sperren ist eine Realisierung zur Synchronisation durch einen transaktionsbasierten Ansatz möglich. Dabei werden Operationen auf Modellen durch *Transaktionen* [HR83] zu Blöcken zusammengefasst, die sich durch folgende Eigenschaften (*ACID*) auszeichnen:

- *Atomarität* (engl. *Atomicity*): Eine Transaktion ist eine Einheit, die nicht weiter zerlegt werden kann. Entweder werden alle oder keine der Operationen der Transaktion ausgeführt.
- *Konsistenz* (engl. *Consistency*): Eine Transaktion führt unabhängig vom Erfolg der Ausführung zu einem konsistenten Endzustand.
- *Isolation*: Nebenläufig ausgeführte Transaktionen beeinflussen sich gegenseitig nicht. Die Auswirkungen der Operationen sind so, als werden alle Transaktionen seriell ausgeführt.
- *Dauerhaftigkeit* (engl. *Durability*): Die Änderungen werden nach dem erfolgreichen Abschluss der Transaktion für alle anderen Benutzer sichtbar und persistent.

Zur Realisierung der Isolationseigenschaft von Transaktionen unterscheidet man optimistische und pessimistische Verfahren. Optimistische Verfahren gehen davon aus, dass bei der Mehrbenutzersynchronisation keine Konflikte auftreten. Die Bearbeitung der Aktionen erfolgt auf lokalen Kopien der Daten. Vor der Einbringung der Änderungen wird die Überprüfung vorgenommen, ob es Konflikte mit anderen Transaktionen gibt. Ist dies der Fall, werden die Aktionen der Transaktion verworfen und dem Aufrufer wird eine entsprechende Fehlermeldung zurückgegeben. Hier wird die Bedeutung des Rückgabewertes `ReturnCode` der gekapselten Modelloperationen der View-Ebene deutlich. Diese Klasse kann für die Anforderungen der Mehrbenutzersynchronisation erweitert werden. Werden keine Konflikte erkannt, können die Änderungen festgeschrieben werden. Bei der Überprüfung auf Konflikte und in der Schreibphase der Veränderungen muss das Managementsystem der Transaktionen für die Korrektheit der Synchronisation gewährleisten, dass jeweils nur ein Prozess diese Aktionen ausführt. Bei pessimistischen Verfahren werden alle benötigten Objekte gesperrt. Deshalb können Verklemmungen auftreten. Außerdem sinkt der Durchsatz bei der Transaktionsverarbeitung, wenn auf die Freigabe von Sperren gewartet werden muss.

Realisiert werden können Transaktionen durch die Umsetzung des Metamodells in eine Datenbank. Ein Datenbankmanagementsystem besitzt Transaktionskonzepte und Verfahren zur Mehrbenutzersynchronisation. Bei der Umsetzung des Modells in ein Datenbankformat tritt jedoch ein Paradigmenbruch auf. Denn die Struktur des Modells muss in die Struktur einer Datenbank übertragen werden. Ein Lösungsansatz ist in [Hah01] beschrieben.

2 Manueller Sperrmechanismus

Beim manuellen Sperrmechanismus ist es die Aufgabe des Benutzers, die Modellelemente zu sperren, die er verändern möchte. Dieser Ansatz ist jedoch recht problematisch, da sich Views

auf Modelle oft überschneiden. Deshalb zieht das Sperren eines Objektes eine Sperrung einer Reihe weiterer Objekte mit sich. Ein Sperren einer Komponente für das Verändern der Modelattribute in einem Systemstrukturdiagramm resultiert in der Sperrung der Sequenz- und Zustandsübergangsdiagramme, die ebenso mit dieser Komponente assoziiert sind. Damit ist die Arbeit der einzelnen Benutzer eingeschränkt.

Ob jedoch dem Benutzer die Entscheidung über das Sperren von Objekten überlassen werden kann, ist aus folgenden Gründen fraglich. Zum einen muss der Benutzer über die Kenntnis verfügen, welche Objekte zu sperren sind. Damit ist ein tieferes Verständnis über die Zusammenhänge des AUTOFOCUS-Metamodells erforderlich. Außerdem kommt es bei diesem Verfahren auf das vernünftige Setzen von Sperren an. Wenden einige Benutzer dieses Verfahren nicht sorgsam an, können andere in ihrer Arbeit eingeschränkt werden. Das manuelle Sperren von Modellelementen ist zudem zu zeitintensiv und reduziert die Performanz. Deshalb sollte ein automatisches dem manuellen Sperrverfahren vorgezogen werden.

Zusammenfassung und Ausblick

1 Zusammenfassung

AUTOFOCUS 2 ist ein modellbasiertes CASE-Werkzeug, das die grafische Modellierung von Software ermöglicht. Der Entwickler bearbeitet Softwaremodelle mittels unterschiedlicher grafischer Editoren. An diese Editoren ergeben sich neben der Funktionalität für den Benutzer zusätzliche Anforderungen aus der Modellbasierung sowie aus dem Einsatz existierender Software. Letztere bietet vorgefertigte Implementierungen beispielsweise für das Zeichnen von Graphen oder die Interaktion mit Eingabegeräten. Durch die Verwendung existierender Software wird der Implementierungs-, Wartungs- und Erweiterungsaufwand für die Benutzeroberfläche reduziert.

Für die Auswahl eines geeigneten Softwaresystems werden unterschiedliche Ansätze zur Realisierung von Editoren untersucht. Diagramm-Zeichenwerkzeuge und fertige Editoren können an die gewünschte Anwendungsdomäne angepasst werden. Die Verbindung zum modellbasierten Anwendungskern ist bei diesen Werkzeugen zu wenig ausgeprägt. Deshalb sind diese Werkzeuge für die Realisierung eines Editors von AUTOFOCUS 2 nicht geeignet. Meta-CASE-Editoren realisieren zwar den Datenaustausch mit externen Anwendungen, ihnen fehlt jedoch die Modellbasierung. Deshalb kommt auch diese Kategorie als Basis für einen Editor von AUTOFOCUS 2 nicht zum Einsatz. Visualisierungswerkzeuge für Graphen bieten lediglich eine Benutzerschnittstelle für Datenstrukturen. Sie erfordern eine zusätzliche Implementierung der Editierfunktionalität. Damit steigt der Implementierungsaufwand für die gesamte Anwendung. Der Vorteil durch den Einsatz von existierender Software wird dadurch eingeschränkt. Dedizierte Graphen-Editoren verfügen zwar über Editierfunktionalität, die Anbindung an den modellbasierten Anwendungskern ist ebenfalls zu wenig ausgeprägt. Deshalb eignen sich die untersuchten Graphen-Editoren nicht als Basis für Editoren von AUTOFOCUS 2. Die untersuchte benutzerdefinierbare Entwicklungsumgebung orientiert sich bei der Visualisierung am Programmtext des modellierten Systems und passt deshalb nicht zum modellbasierten Ansatz der vorliegenden Arbeit. Die Untersuchung der verschiedenen Softwarelösungen zeigt, dass eine Anbindung eines Editors an den modellbasierten Kern von AUTOFOCUS 2 beim Einsatz eines Rahmenwerks für Editoren am besten gelingt. Rahmenwerke realisieren Standardaufgaben und bieten Einhängpunkte für die Interaktion mit externen Programmen. Das Ergebnis der Evaluierung zeichnet mehrere Rahmenwerke als mögliche Basis für Editoren von AUTOFOCUS 2 aus. Das Rahmenwerk GEF erfüllt jedoch am besten die gegebenen Anforderungen. Die Konfidenz in diese Entscheidung wird durch die Referenzimplementierung eines UML-Modellierungswerkzeugs auf Basis von GEF untermauert, das auf die gleiche Anwendungsdomäne wie AUTOFOCUS 2 zielt.

Um das gewählte Rahmenwerk zu verwenden, wird eine Schichtenarchitektur entwickelt, welche die Anbindung an den modellbasierten Anwendungskern von AUTOFOCUS 2 ermög-

licht. Das Design der Anwendung ist damit durch die Anforderungen der Modellbasierung und die Vorgaben des Rahmenwerks bestimmt. Die Schichtenarchitektur unterstützt die 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 lediglich in ihren Zugriffsstrukturen an die Veränderungen angepasst werden. Die Ebenen 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.

Die Architektur des modellbasierten Editorenkonzepts wird mit einer prototypischen Implementierung eines Systemstrukturdiagramms verifiziert und bewertet. Mit der Implementierung wird zudem die Funktionalität des Rahmenwerks durchleuchtet, um es für eine Verwendung in anderen Editoren zu untersuchen. Die Architektur des Editorenkonzepts bildet die Basis für weitere Editoren. Mit der Validierung des erarbeiteten Konzepts ist auch die Übertragung auf Editoren anderer Diagrammartentypen demonstriert. Editoren visualisieren Ausschnitte aus dem integralen Systemmodell und modifizieren Modellelemente. Welche Modellelemente durch einen Editor bearbeitet werden, beeinflusst nicht den Mechanismus für den Informationsaustausch zwischen dem zentralen Systemmodell und einem Editor.

Bei der Implementierung des prototypischen Editors stellt sich heraus, dass die Funktionalität des Rahmenwerks für die sehr enge Ankopplung an den modellbasierten Anwendungskern von AUTOFOCUS 2 geringer als erwartet ist. Um die Anforderungen an die Anwendung zu erfüllen, ist die Erweiterung des Rahmenwerks durch eigene Implementierungen erforderlich. Trotz der in manchen Bereichen fehlenden Funktionalität erweist sich das verwendete Rahmenwerk als geeignete Lösung für eine Basis zur Realisierung von weiteren modellbasierten Editoren.

2 Ausblick

Zukünftige Herausforderungen betreffen die Realisierung von weiteren Quest-Editoren. Dazu wird das erarbeitete modellbasierte Konzept auf andere Editoren übertragen.

Das Editorenkonzept unterstützt keine Synchronisationsmechanismen für einen Mehrbenutzerbetrieb, sondern lediglich ein unabhängiges Arbeiten. Parallele Zugriffe auf gemeinsame Daten erfordern Mechanismen zu ihrer Synchronisation, um die Konsistenz der Daten zu gewährleisten. Die Synchronisation kann durch das Konzept der Transaktionen realisiert werden, dessen Sperrkonzept an der Struktur des hierarchischen Systemmodells orientiert ist.

Eine weitere Herausforderung bildet die Integration der Prozessunterstützung im Editor. Für Teile der Prozessunterstützung ist die Modellbasierung des Anwendungskerns unerlässlich. Diese Teile zielen beispielsweise auf die Realisierung von Konsistenzüberprüfungen oder eine integrierte Simulations- und Testumgebung ab. Die vollständige Prozessunterstützung erfordert die Integration eines zusätzlichen Produktmodells. Dieses ist in der gegenwärtigen Implementierung von AUTOFOCUS nicht realisiert. In einer weiteren Ausbaustufe des Prototyps ist die Integration der Prozessunterstützung in den Editor denkbar.

Beim Verbergen von grafischen Objekten wird die aktuelle Zusammensetzung des Modells für die Anzeige nur von Ausschnitten gefiltert. Dazu werden nur diejenigen Elemente angezeigt, die für den Benutzer interessant sind. Die übrigen Elemente im Editorfenster, die für eine

bestimmte Betrachtungsweise des modellierten Systems nicht relevant sind, werden ausgeblendet. Das Verbergen von Elementen erhöht die Verständlichkeit und die Übersichtlichkeit von umfangreichen Diagrammen. Die Realisierung dieser Funktionalität betrifft nicht nur die Anzeigefunktionalität im Editor, sondern hat auch Auswirkungen auf tiefer liegende Ebenen der Schichtenarchitektur.

Die Implementierung des Prototyps zeigt, dass der durch das Rahmenwerk vorgegebene Layout-Algorithmus nicht ausreichend ist. Für eine automatische Anordnung der grafischen Objekte im Editor ist die Integration eines geeigneten Layout-Verfahrens erforderlich.

Der prototypische Editor bietet Erweiterungsmöglichkeiten für den Bereich Editierfunktionalität wie z.B. zum Kopieren, Einfügen und Ausschneiden von Elementen. Diese Funktionalität erfordert nicht nur eine entsprechende Implementierung in der Ebene des Editors. Weil der Editor die aktuelle Zusammensetzung des Systemmodells reflektiert, hat diese Funktionalität auch Auswirkungen auf andere Schichten. Für eine Implementierung ist die Übertragung des erarbeiteten Editorenkonzepts auf weitere Editierfunktionalität erforderlich.

In [Wil02] ist ein modellbasiertes Hilfesystem für AUTOFOCUS 2 beschrieben. Die Integration dieses Konzepts in den Editor erfordert die Kombination der Aktionen des Benutzers mit dem Hilfesystem. Die Einbindung des Hilfesystems ist in dieser Arbeit jedoch nicht betrachtet.

Eigenschaften der evaluierten Software

1 AbsInt aiSee Graph Visualization

Quelle: <http://www.absint.com/aisee>

Evaluierte Version: 1.35

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–
Lizenz	Die Lizenz ist für Lehr- und Forschungszwecke an Universitäten frei.	5
Stabilität	Das kommerzielle Produkt ist in Version 1.35.	5
Grafische Objekte	Es werden nur Graphen visualisiert.	1
Präsentation	Nicht bewertet.	–
Plattform	Es stehen Versionen für Windows, Linux, Solaris, SunOS, MacOS X und NetBSD zur Verfügung.	5
Anbindung Quest	Die Anbindung an andere Anwendungen erfolgt in Form von Dateien mit textuellem Beschreibungsformat der Graphen.	1
Weiterentwicklung	Eine Einstellung der Entwicklung ist nicht zu erwarten.	5
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.1. Bewertung der Ausschlusskriterien für aiSee

2 daVinci Presenter

Quelle: <http://www.b-novative.de/products/daVinci/daVinci.html>

Evaluierte Version: 3.0.3

Kriterium	Anmerkung			Note
Einarbeitung	Entwicklung	Tutorials, die Dokumentation der API-Befehle und ein Beispielprogramm für die Anbindung einer externen Anwendung sind vorhanden.	5	5
	Architektur	Nicht bewertbar.	–	
	Strukturierung	Nicht bewertbar.	–	
	Verwendung	Die Ankopplung der Software an eine externe Anwendung ist durch die Dokumentation der API-Befehle und durch ein Beispielprogramm erläutert.	5	
	Javadoc	Nicht bewertbar.	–	
	Einarbeitungszeit	Die Einarbeitungszeit von zwei Wochen oder weniger ist ausreichend. Die Einarbeitung konzentriert sich auf die Implementierung der externen Anwendung, deren Anbindung an daVinci Presenter und die Verwendung der API-Befehle.	5	
Lizenz	Die Lizenz ist für Lehr- und Forschungszwecke an Universitäten frei.			5
Stabilität	Das kommerzielle Produkt ist in Version 3.0.3.			5
Grafische Objekte	Das Programm visualisiert gerichtete Graphen. Es können zyklische, azyklische und leere Graphen, d.h. eine Menge von Knoten ohne Kanten, angezeigt werden. Schlingen und Mehrfachkanten werden unterstützt. Der Bezug zu Graphen sollte bei den darzustellenden Daten immer vorhanden sein. Sequenzdiagramme können angezeigt werden, jedoch ist der Layout-Algorithmus durch die externe Anwendung zu realisieren, der daVinci Presenter die Positionen der Knoten mitteilt.			2
Präsentation	Die Gestaltung der Oberfläche ist etwas veraltet.			4
Plattform	Es stehen Versionen für Linux, Windows 2000 und Solaris zur Verfügung.			5
Anbindung Quest	Die Anbindung an ein externes Programm ist grundsätzlich möglich. Sie erfolgt über TCP/IP-Sockets oder UNIX-Pipes und die API-Befehle von daVinci Presenter. Die Spezifikation des Graphen erfolgt durch eine Termnotation. Das Programm dient lediglich zur Visualisierung von Graphen, nicht zu ihrer Erstellung und Modifikation.			2

Divia

Kriterium	Anmerkung	Note
Weiterentwicklung	Eine Einstellung der Entwicklung ist nicht zu erwarten.	5
Gesamtbewertung		4,1

Tabelle A.2. Bewertung der Ausschlusskriterien für daVinci Presenter

3 Divia

Quelle: <http://www.gigascale.org/divia>

Evaluierte Version: 0.3

Kriterium	Anmerkung		Note
Einarbeitung	Entwicklung	Tutorials für die Erstellung eines Editors sind vorhanden, jedoch entsprechen diese teilweise nicht dem aktuellen Stand der Implementierung des Rahmenwerks. Eine Reihe von Anwendungen ist vorhanden, welche die grafischen Möglichkeiten von Divia zeigen, jedoch nur sehr geringe Editorfunktionalität aufweisen, d.h. nur das Erstellen und Löschen von grafischen Objekten.	3
	Architektur	Eine Dokumentation der Programmteile ist vorhanden, sie fällt jedoch recht knapp aus.	3
	Strukturierung	Das Rahmenwerk ist sehr gut nach den Aufgabenbereichen strukturiert.	5
	Verwendung	Die Verwendung der Komponenten ist aus den Beispielanwendungen und der kurz gefassten Dokumentation zu entnehmen.	3
	Javadoc	Ist vorhanden.	5
	Einarbeitungszeit	Wegen der geringen Dokumentation und des recht großen Umfangs des Rahmenwerks ist die Einarbeitungszeit hoch.	3
Lizenz	Divia kann unter der GNU-Lizenz eingesetzt werden.		5
Stabilität	An manchen Stellen fehlt tatsächlich implementierte Funktionalität, beispielsweise die interaktive Beschriftung von grafischen Objekten.		3

Kriterium	Anmerkung	Note
Grafische Objekte	Beliebige Diagramme sind darstellbar. Schlingen können angezeigt werden; Mehrfachkanten sind möglich, überlappen sich jedoch gegenseitig. Bézier-Kurven für Zustandsübergangsdigramme, direkte Verbindungslinien für Sequenzdiagramme und rechtwinklige Kanten für Systemstrukturdiagramme werden realisiert.	4
Präsentation	Die Oberfläche kann entsprechend den Fähigkeiten von Java und der Swing-Bibliothek angepasst werden.	5
Plattform	Die Entwicklungssprache ist Java. Deshalb kann das Rahmenwerk auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5
Anbindung Quest	Bei Diva handelt es sich um ein Rahmenwerk mit einer MVC-Architektur.	5
Weiterentwicklung	Das letzte Snapshot Release wurde im Januar 2002 veröffentlicht.	3
Gesamtbewertung		4,2

Tabelle A.3. Bewertung der Ausschlusskriterien für Diva

Kriterium	Anmerkung	Note	
Komplexität	488 Java-Dateien.	4	
Benutzerschnittstelle	Bedienung	Ein implementierter grafischer Editor ist nicht vorhanden.	–
	Funktionen	Erstellen, Löschen, Größenänderung und Verschieben von grafischen Objekten ist implementiert. Eine Historie für die Verfolgung der Veränderung des Zustandes von Datenmodellen ist vorhanden, damit ist die Grundlage für die Realisierung von Undo- und Redo-Operationen gegeben. Die Historie kann ebenso dazu verwendet werden, kollaborative Group-Awareness zu realisieren. Weitere Funktionen: Kopieren, Einfügen, Ausschneiden, neues Dokument, Schließen, Beenden, Öffnen, Speichern und Speichern unter. Nur die Funktionalität zum Speichern und Öffnen wurde am vorhandenen Texteditors getestet. Für grafische Editoren ist die Erweiterung der Implementierung zur Realisierung der genannten Funktionalität nötig, um die grafischen Objekte zu integrieren.	3
	Navigation	Ein Navigationsfenster kann zum Navigieren in großen Zeichnungen verwendet werden.	5

Divia

Kriterium	Anmerkung			Note
Ankopplung Quest	MVC-Konzept	Die Architektur von Divia trennt Modell und Repräsentation. Das MVC-Konzept ist implementiert.	5	5
	Ereignis-Mechanismen	Die Anzeige wird aktualisiert, wenn neue Daten vorliegen.	5	
Erweiterungs- und Anpassungsfähigkeit	Änderungen am Quest-Metamodell erfordern die Anpassung der grafischen Anzeige. Divia bietet dazu die Definition von eigenen, grafischen Objekten.			5
Integration von zusätzlicher Funktionalität	Benutzerdefinierte Funktionen	Implementierung ist möglich.	5	5
	Folgen von Operationen	Implementierung ist möglich.	5	
Integrierte Benutzerführung	Es kann überprüft werden, ob ein Kantenende oder -anfang zu dem gegebenen Knoten im Modell hinzugefügt werden darf.			3
Mehrbenutzerfähigkeit	Nicht unterstützt.			1
Layout	Einbettung von Algorithmen	Erweiterungen und das Hinzufügen von Algorithmen sind möglich.	5	5
	Integrierte Algorithmen	Layout-Algorithmen zur automatischen Anordnung von Knoten sind implementiert.	5	
Quelltext	Ist verfügbar.			5
Gesamtbewertung				4,1

Tabelle A.4. Bewertung der erweiterten Kriterien für Divia

Gesamtbewertung aus Ausschluss- und erweiterten Kriterien: 4,2

4 GEF

Quelle: <http://gef.tigris.org>

Evaluierte Version: 0.96

Kriterium	Anmerkung			Note
Einarbeitung	Entwicklung	Die Vorgehensweise bei der Implementierung eines Grapheneditors ist sehr kurz beschrieben. Die Erstellung einer Anwendung auf Basis des Rahmenwerks wird durch einen implementierten Beispielditor mit Variationen in der Darstellung von Figuren erläutert.	2	3
	Architektur	Die wichtigsten Klassen werden in Form von CRC-Karten kurz beschrieben.	2	
	Strukturierung	Die Klassen sind entsprechend ihrer Aufgaben in Pakete aufgeteilt.	5	
	Verwendung	Die Verwendung der Komponenten ist durch den Beispielditor und Javadoc-Kommentare beschrieben.	2	
	Javadoc	Ist vorhanden.	5	
	Einarbeitungszeit	Das Design und die Funktionalität von GEF werden seit Version 0.6 nicht mehr dokumentiert. Mit ArgoUML existiert eine Referenzimplementierung, die für die Realisierung von AUTOFOCUS-Editoren auf der Basis von GEF relevante Funktionalität wie die Unterstützung von UML-Diagrammen und Benutzerführung besitzt.	2	
Lizenz	GEF und die Programmtexte sind unter der 4.4BSD-Lizenz frei verfügbar. Diese ermöglicht den Einsatz des Rahmenwerks auch in einem kommerziellen Produkt. ArgoUML ist unter der BSD-Lizenz verfügbar.			5
Stabilität	Die aktuelle Version von GEF ist Version 0.96. Die Entwicklung ist kontinuierlich.			4
Grafische Objekte	GEF ist hauptsächlich zur Darstellung von Graphen entwickelt. Das implementierte Modell des MVC-Konzeptes reflektiert diesen Aspekt. Schlingen und Mehrfachkanten in Graphen können dargestellt werden. Die Diagramme aus der UML-Version 1.3 [For00] können in ArgoUML auf der Basis von GEF gezeichnet werden. Direkte Verbindungslinien für Sequenzdiagramme und rechtwinklige Kanten für Systemstrukturdiagramme werden realisiert. Bézier-Kurven für Zustandsübergangsdigramme fehlen, können aber durch die Bibliothek für grafische Objekte implementiert werden.			4

Kriterium	Anmerkung	Note
Präsentation	Die Oberfläche entspricht dem heutigen Stand der Entwicklung und kann entsprechend den Fähigkeiten von Java und der Swing-Bibliothek gestaltet werden.	5
Plattform	Die Entwicklungssprache ist Java. Deshalb kann das Rahmenwerk auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5
Anbindung Quest	Die Anbindung von Quest ist realisierbar, da der Quelltext verfügbar ist und die Software als Rahmenwerk realisiert ist.	5
Weiterentwicklung	Die Entwicklung erfolgt hauptsächlich im Zusammenhang mit ArgoUML. Die letzte Version von GEF ist im Juni 2002 fertiggestellt. Die letzte Version von ArgoUML ist im Mai 2002 veröffentlicht. Die Entwicklung von ArgoUML, mit der die Pflege von GEF verbunden ist, wird von Open-Source-Entwicklern ausgeführt.	3
Gesamtbewertung		4,3

Tabelle A.5. Bewertung der Ausschlusskriterien für GEF

Kriterium	Anmerkung	Note
Komplexität	Das Rahmenwerk besteht aus 248 Java-Klassen.	5
Benutzerschnittstelle	Bedienung	Intuitiv. 5
	Funktionen	Kopieren, Einfügen, Löschen, Öffnen, Speichern, Drucken, Größenänderung und Verschieben sind im Beispielditor BasicApplication demonstriert. Eine Standardbibliothek für weitere Funktionalität ist implementiert. 5
	Navigation	Eine Klasse, welche die Zoom-Funktion implementiert, ist vorhanden. 5
Ankopplung Quest	MVC-Konzept	Das Rahmenwerk implementiert das MVC-Konzept. Damit basiert ein gezeichnetes Diagramm auf einem Modell, das entsprechend der Zielanwendung selbst definiert werden kann. 5
	Ereignis-Mechanismen	Aufgrund der Implementierung des MVC-Konzeptes ist auch die Benachrichtigung anderer Anwendungen durch Ereignisse möglich. 5
Erweiterungs- und Anpassungsfähigkeit	Das Rahmenwerk kann erweitert werden. Bei einer Änderung des Quest-Metamodells müssen Klassen für die grafische Anzeige und ein entsprechender Eintrag im Graphenmodell realisiert werden.	5

Kriterium	Anmerkung			Note
Integration von zusätzlicher Funktionalität	Benutzerdefinierte Funktionen	Eine Implementierung ist möglich. Es handelt sich bei GEF um ein Rahmenwerk, das nach den Anforderungen der Anwendung erweitert werden kann.	5	5
	Folgen von Operationen	Eine Implementierung ist möglich.	5	
Integrierte Benutzerführung	Wenn ein Knoten hinzugefügt wird, werden benutzerdefinierte Operationen ausgeführt. Nach dem Hinzufügen oder Entfernen von Kanten werden anwendungsspezifische Aktionen durchgeführt. Beim Verbinden von Ports wird überprüft, ob diese Verbindung den Syntaxregeln der Anwendung entspricht. Diese Einhängpunkte für benutzerdefinierte Methoden sind jedoch in ihrem Umfang ausbaufähig.			4
Mehrbenutzerfähigkeit	Wird nicht unterstützt.			1
Layout	Einbettung von Algorithmen	Die Einbindung von externen Layout-Verfahren wird in der Dokumentation nicht erwähnt.	–	5
	Integrierte Algorithmen	Die Kanten können direkt oder rechtwinklig verlaufen.	5	
Quelltext	Der gesamte Programmtext ist verfügbar.			5
Gesamtbewertung				4,4

Tabelle A.6. Bewertung der erweiterten Kriterien für GEF

Gesamtbewertung aus Ausschluss- und erweiterten Kriterien: 4,3

5 Gift

Quelle: <http://www.metagift.com>

Evaluierte Version: 1.8

Kriterium	Anmerkung			Note
Einarbeitung	Entwicklung	Die Erstellung des Editors erfolgt durch die grafische Benutzeroberfläche, zusätzlich können Ergänzungen programmiert werden. Die Dokumentation dafür im Benutzerhandbuch ist gut und ausführlich.	5	5
	Architektur	Nicht bewertbar.	–	
	Strukturierung	Das Rahmenwerk ist entsprechend den realisierten Aufgaben der Klassen sehr gut strukturiert.	5	
	Verwendung	Die Anpassung des Editors durch Programmierung ist im Handbuch und der API beschrieben.	5	
	Javadoc	Ist vorhanden.	5	
	Einarbeitungszeit	Die Zeit von zwei Wochen oder weniger reicht aus, da die Anpassung des Editors hauptsächlich durch die Verwendung der Anwendung erfolgt. Implementierungsarbeiten sind nicht besonders umfangreich, da sich ihr Schwerpunkt auf die Realisierung des Im- und Exports bezieht.	5	
Lizenz	Eine freie Evaluierung ist für einen Zeitraum von 90 Tagen möglich. Diese Lizenz kann für akademische Forschungszwecke immer wieder erneuert werden.			5
Stabilität	Version 1.8, dennoch handelt es sich um eine Vorversion.			5
Grafische Objekte	Es können beliebige Zeichnungen erstellt werden.			5
Präsentation	Die Oberfläche ist ansprechend gestaltet.			5
Plattform	Die Plattformen Windows, UNIX und Solaris werden unterstützt. Die Programmierung des Editors erfolgt mit der Sprache Java.			4
Anbindung Quest	Die Anbindung für den Datenaustausch wird durch die Verwendung von benutzerdefinierten Speicher- und Ladefunktionen und die Anpassung des Verhaltens der grafischen Objekte ermöglicht. Es gibt jedoch keinen Ereignis-Mechanismus, der bei Änderungen andere Komponenten benachrichtigt. Der Informationsaustausch mit dem zentralen AUTOFOCUS-Modell erfolgt nur beim Laden und Speichern.			1

Kriterium	Anmkerung	Note
Weiterentwicklung	Es ist keine weitere Entwicklung geplant.	1
Gesamtbewertung		3,9

Tabelle A.7. Bewertung der Ausschlusskriterien für Gift

6 GraphEd

Quelle: <http://www.infosun.fmi.uni-passau.de/GraphEd>

Evaluierte Version: Keine Versionsbezeichnung vorhanden

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–
Lizenz	Die Lizenz ist für nicht kommerzielle Zwecke frei.	5
Stabilität	Nicht bewertet.	–
Grafische Objekte	Es können nur gerichtete oder ungerichtete, beschriftete Graphen mit Schlingen und Mehrfachkanten gezeichnet werden.	1
Präsentation	Nicht bewertet.	–
Plattform	Nicht bewertet.	–
Anbindung Quest	Die Anbindung erfolgt nur durch einen Dateiaustausch.	1
Weiterentwicklung	Die Entwicklung ist eingestellt, der Nachfolger ist Graphlet (siehe Anhang 7).	1
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.8. Bewertung der Ausschlusskriterien für GraphEd

7 Graphlet

Quelle: <http://www.infosun.fmi.uni-passau.de/Graphlet>

Evaluierte Version: 5.0

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–

ILOG JViews Component Suite

Kriterium	Anmerkung	Note
Lizenz	Die Lizenz ist für nicht kommerzielle Zwecke frei.	5
Stabilität	Das Produkt ist bereits in Version 5.0.	5
Grafische Objekte	Es werden nur Graphen visualisiert und editiert.	1
Präsentation	Nicht bewertet.	–
Plattform	Es stehen Versionen für Windows, Linux und Solaris zur Verfügung.	5
Anbindung Quest	Die Anbindung erfolgt durch die Spezifikation von Graphen in einer Datei.	1
Weiterentwicklung	Die Weiterentwicklung erfolgt in einem kontinuierlichen Projekt an der Universität Passau.	5
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.9. Bewertung der Ausschlusskriterien für Graphlet

8 ILOG JViews Component Suite

Quelle: <http://www.ilog.com/products/jviews>

Evaluierte Version: 5.0

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–
Lizenz	Eine Lizenz für akademische Zwecke für die Pakete JViews, Views und JRules kostet 1.100 €, zehn Lizenzen 4.000 €. Eine getrennte Lizenzierung der Software ist nicht möglich.	1
Stabilität	Bei der Software handelt es sich um ein kommerzielles Produkt in der Version 5.0.	5
Grafische Objekte	Es können beliebige grafische Objekte gezeichnet werden.	5
Präsentation	Die Oberfläche entspricht dem heutigen Stand der Entwicklung.	5
Plattform	Die Entwicklungssprache ist Java. Deshalb kann die Software auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5
Anbindung Quest	Die Anbindung an eine externe Anwendung ist möglich. Der Quelltext der Software ist verfügbar.	5
Weiterentwicklung	Eine Einstellung der Entwicklung ist nicht zu erwarten.	5

Kriterium	Anmerkung	Note
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.10. Bewertung der Ausschlusskriterien für ILOG JViews

9 JGraph

Quellen: jgraph.sourceforge.net, <http://www.jgraph.com> und <http://www.jgraph.org>

Evaluierte Version von JGraph: 1.0.4

Evaluierte Version von JGraphpad: 1.1.1

Kriterium	Anmerkung	Note	
Einarbeitung	Entwicklung	Der Beispieleditor JGraphpad demonstriert die Funktionalität von JGraph. Daneben existieren zwei einfache Beispielanwendungen und zwei umfangreichere Beispiele.	5
	Architektur	Das Design und die Implementierung werden sehr ausführlich dokumentiert.	5
	Strukturierung	Die Klassen sind entsprechend ihrer Aufgaben in Pakete eingeteilt.	5
	Verwendung	Ein Tutorial beschreibt die Verwendung der Komponenten ausführlich.	5
	Javadoc	Ist vorhanden.	5
	Einarbeitungszeit	Die ausführliche Dokumentation erlaubt die schnelle Einarbeitung in die Software.	5
Lizenz	GNU Lesser General Public License	5	
Stabilität	Das Produkt ist in der Version 1.0.4.	5	
Grafische Objekte	Die Software ist hauptsächlich für die Visualisierung von Graphen entwickelt. Schlingen und Mehrfachkanten werden unterstützt. Bei der Modellierung von Sequenzdiagrammen ist ein benutzerdefiniertes Graphenmodell notwendig. JGraph unterstützt die Definition eigener Modelle. Bézier-Kurven für Zustandsübergangsdigramme und direkte Verbindungslinien für Sequenzdiagramme werden realisiert. Es fehlt die Implementierung von rechtwinkligen Kanten für Systemstrukturdiagramme.	4	
Präsentation	Die Oberfläche ist mit Java Swing-Komponenten gestaltet.	5	
Plattform	Die Entwicklungssprache ist Java. Deshalb kann die Komponente auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5	

JGraph

Kriterium	Anmerkung	Note
Anbindung Quest	Die Anbindung von JGraph an eine andere Anwendung ist möglich.	5
Weiterentwicklung	Die Weiterentwicklung der freien Version von JGraph ist von der Beteiligung der Open-Source-Entwickler abhängig. JGraphpad wird als kostenlose Version in einem Open-Source-Projekt weiterentwickelt.	4
Gesamtbewertung		4,8

Tabelle A.11. Bewertung der Ausschlusskriterien für JGraph

Kriterium	Anmerkung		Note
Komplexität	JGraphpad umfasst 17 Java-Dateien, JGraph 40 Java-Dateien mit insgesamt 16.000 Zeilen Programmtext. JGraph ist als Swing-Komponente entwickelt und für die Einbettung in eine Anwendung ausgelegt.		5
Benutzerschnittstelle	Bedienung	Intuitiv.	5
	Funktionen	Kopieren, Einfügen, Löschen, Suchen, Undo, Redo, Datei neu, Öffnen, Speichern und Drucken.	5
	Navigation	Die Navigation in umfangreichen Zeichnungen wird durch eine Zoom-Funktion realisiert.	5
Ankopplung Quest	MVC-Konzept	Die Trennung von Modell und Repräsentation ist vorhanden, das MVC-Konzept ist implementiert.	5
	Ereignis-Mechanismen	Mehrere Sichten auf ein Modell sind möglich, andere Editoren können über Änderungen am Modell benachrichtigt werden.	5
Erweiterungs- und Anpassungsfähigkeit	Die Software kann erweitert und an ein benutzerdefiniertes Metamodell für die Zielanwendung angepasst werden.		5
Integration von zusätzlicher Funktionalität	Benutzerdefinierte Funktionen	Die Integration eigener Funktionalität ist möglich.	5
	Folgen von Operationen	Zusammengesetzte Operationen können durch einen Transaktionsmechanismus bei einem benutzerdefinierten Modell implementiert werden.	5
Integrierte Benutzerführung	Die Überprüfung, ob eine Kante mit einem Port als Quelle oder Ziel verbunden werden kann, ist implementiert.		3
Mehrbenutzerfähigkeit	Die Mehrbenutzerfähigkeit ist nicht implementiert. Ein gemeinsames Modell kann durch mehrere Sichten von mehreren Benutzern betrachtet werden. Der Undo- und Redo-Mechanismus ist hier nicht eingeschränkt.		2

Kriterium	Anmerkung		Note
Layout	Einbettung von Algorithmen	Ist möglich.	5
	Integrierte Algorithmen	In der Beispielanwendung verlaufen die Kanten nur direkt. Das automatische Auslegen von Kanten wird nicht unterstützt, kann aber implementiert werden.	3
Quelltext	Die Quelltexte für JGraph und JGraphpad sind verfügbar.		5
Gesamtbewertung			4,3

Tabelle A.12. Bewertung der erweiterten Kriterien für JGraph

Gesamtbewertung aus Ausschluss- und erweiterten Kriterien: 4,5

10 JHotDraw

Quellen: <http://JHotDraw.sourceforge.net> und <http://sourceforge.net/projects/jhotdraw/>
 Evaluierte Version: 5.3

Kriterium	Anmerkung		Note
Einarbeitung	Entwicklung	Vier Editoren mit Beispielzeichnungen sind vorhanden.	5
	Architektur	Die Paketstruktur und ein Überblick über die Aufgaben der Komponenten werden nur kurz beschrieben.	2
	Strukturierung	Das Rahmenwerk ist strukturiert, bietet jedoch keine Trennung von Darstellung und Modell.	3
	Verwendung	Die Verwendung der Komponenten wird kurz erläutert.	2
	Javadoc	Ist vorhanden.	5
	Einarbeitungszeit	Trotz des überschaubaren Umfangs des Rahmenwerks ist aufgrund der wenigen Dokumentation mit einem etwas höheren Umfang der Einarbeitungszeit zu rechnen.	3
Lizenz	Das Rahmenwerk kann unter der Lesser GNU Public License (LGPL) eingesetzt werden.		5
Stabilität	Das Rahmenwerk ist bereits in Version 5.3.		5

Kriterium	Anmerkung	Note
Grafische Objekte	Graphen und beliebige Diagramme können gezeichnet werden. Schlingen und Mehrfachkanten werden unterstützt.	5
Präsentation	Die Benutzeroberfläche ist etwas veraltet.	4
Plattform	Die Entwicklungssprache ist Java. Deshalb kann das Rahmenwerk auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5
Anbindung Quest	Ein MVC-Konzept wird in der Dokumentation nicht erwähnt. Eine Anbindung an Quest ist jedoch möglich, weil der Programntext verfügbar ist.	2
Weiterentwicklung	Die letzte Version ist im Januar 2002 veröffentlicht. JHotDraw wird weiterentwickelt: Im Jahr 2002 ist die Veröffentlichung von mindestens zwei Aktualisierungen geplant.	5
Gesamtbewertung		4,3

Tabelle A.13. Bewertung der Ausschlusskriterien für JHotDraw

Kriterium	Anmerkung	Note	
Komplexität	Das Rahmenwerk für Editoren umfasst 195 Java-Dateien.	5	
Benutzerschnittstelle	Bedienung	Die Bedienung ist intuitiv.	5
	Funktionen	Kopieren, Einfügen, Ausschneiden, Löschen, Duplizieren, Verschieben, Größenänderung, Speichern, Öffnen, Drucken, Undo und Redo.	5
	Navigation	Funktionen für die Navigation in umfangreichen Zeichnungen sind nicht implementiert.	2
Ankopplung Quest	MVC-Konzept	Ein MVC-Konzept ist nicht realisiert.	1
	Ereignis-Mechanismen	Ereignis-Mechanismen sind nicht implementiert.	1
Erweiterungs- und Anpassungsfähigkeit	Die Definition von eigenen grafischen Objekten ist möglich.	5	
Integration von zusätzlicher Funktionalität	Benutzerdefinierte Funktionen	Implementierung ist möglich.	5
	Folgen von Operationen	Implementierung ist möglich.	5
Integrierte Benutzerführung	Nicht implementiert.	1	

Kriterium	Anmerkung			Note
Mehrbenutzerfähigkeit	Nicht implementiert.			1
Layout	Einbettung von Algorithmen	Nicht erwähnt.	1	3
	Integrierte Algorithmen	Beim Verschieben von Knoten werden andere Knoten durch die Kreuzung mit Kanten nicht durchzogen, da im Rahmenwerk auch rechtwinklig verlaufende Kanten realisiert sind.	5	
Quelltext	Ist verfügbar.			5
Gesamtbewertung				3,1

Tabelle A.14. Bewertung der erweiterten Kriterien für JHotDraw

Gesamtbewertung aus Ausschluss- und erweiterten Kriterien: 3,7

11 MetaEdit+

Quelle: <http://www.metacase.com>

Evaluierte Version: 3.0

Kriterium	Anmerkung			Note
Einarbeitung	Entwicklung	Tutorials für die Erstellung einer domänenspezifischen Entwicklungsumgebung sind vorhanden.	5	5
	Architektur	Die einzelnen Werkzeuge sind ausführlich dokumentiert.	5	
	Strukturierung	Nicht bewertbar.	–	
	Verwendung	Eine gute Anleitung zur Benutzung der Werkzeuge ist verfügbar.	5	
	Javadoc	Nicht bewertbar.	–	
	Einarbeitungszeit	Wegen der guten Dokumentation ist die Einarbeitung nach der angestrebten Zeit von zwei Wochen oder weniger abgeschlossen. Außerdem ist nur die Einarbeitung in die Bedienung der Software nötig.	5	
Lizenz	\$ 4.500 - \$17.500			1
Stabilität	Das kommerzielle Produkt ist in Version 3.0.			5

Tom Sawyer Software

Kriterium	Anmerkung	Note
Grafische Objekte	Beliebige, benutzerdefinierte Notationen sind möglich. In MetaEdit+ sind vordefinierte Modellierungssprachen wie z.B. UML bereits implementiert.	5
Präsentation	Die Oberfläche ist etwas veraltet.	4
Plattform	Die Software ist in Versionen für die Plattformen Windows, Linux, Solaris und HP-UX verfügbar.	4
Anbindung Quest	Das Werkzeug bietet einen zentralen Datenspeicher zur Konsistenzhaltung der Daten an. Jedoch erfolgt die Anbindung an Quest nur über exportierte Dateien. Eine Importierfunktion fehlt. Dem Werkzeug fehlt also ein modellbasierter Ansatz.	1
Weiterentwicklung	Das Produkt ist kommerziell, eine Einstellung der Weiterentwicklung ist nicht zu erwarten.	5
Gesamtbewertung		3,8

Tabelle A.15. Bewertung der Ausschlusskriterien für MetaEdit+

12 Tom Sawyer Software

Quelle: <http://www.tomsawyer.com>

Evaluierte Version: 5.0

Evaluierte Programmpakete: Graph Editor Toolkit und Graph Layout Toolkit

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–
Lizenz	Lizenzen für akademische Forschungszwecke werden nach Anfrage vergeben.	–
Stabilität	Das kommerzielle Produkt ist bereits in Version 5.0.	5
Grafische Objekte	Es werden ausschließlich gerichtete und ungerichtete Graphen mit Zyklen, Mehrfachkanten und reflexiven Kanten unterstützt.	1
Präsentation	Nicht bewertet.	–
Plattform	Die Software wird für die Plattformen UNIX, Windows, Macintosh und OS/2 angeboten. Die Pakete sind mit einer API für C++, Java und ActiveX verfügbar.	5
Anbindung Quest	Die Integration der Software in eine andere Anwendung ist möglich.	5

Kriterium	Anmerkung	Note
Weiterentwicklung	Eine Einstellung der Entwicklung ist nicht zu erwarten.	5
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.16. Bewertung der Ausschlusskriterien für Tom Sawyer Software

13 VCG

Quelle: <http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html>

Evaluierte Version: Produkt von 1995

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–
Lizenz	GNU General Public License.	5
Stabilität	Nicht bewertet.	–
Grafische Objekte	Es werden nur Graphen angezeigt.	1
Präsentation	Nicht bewertet.	–
Plattform	Die Plattformen Windows 3.1, UNIX, Solaris, Linux werden unterstützt.	4
Anbindung Quest	Die Anbindung erfolgt durch den Austausch von Dateien mit einem textuellen Beschreibungsformat der Graphen.	1
Weiterentwicklung	Nicht bewertet.	–
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.17. Bewertung der Ausschlusskriterien für VCG

14 VGJ

Quelle: http://www.eng.auturn.edu/department/cse/research/graph_drawing/graph_drawing.html

Evaluierte Version: 1.03

Kriterium	Anmerkung	Note
Einarbeitung	Nicht bewertet.	–

Kriterium	Anmerkung	Note
Lizenz	GNU General Public License, Version 2	5
Stabilität	Die aktuelle Version ist 1.03.	5
Grafische Objekte	Es werden nur Graphen visualisiert.	1
Präsentation	Nicht bewertet.	–
Plattform	Die Entwicklungssprache ist Java. Deshalb kann die Software auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.	5
Anbindung Quest	Die Anbindung erfolgt über die Spezifikation der Graphen durch ein textuelles Beschreibungsformat.	1
Weiterentwicklung	Das Projekt wird offiziell nicht mehr fortgeführt.	1
Gesamtbewertung	Die Software ist nicht vollständig evaluiert.	–

Tabelle A.18. Bewertung der Ausschlusskriterien für VGJ

15 Visual Shape

Quelle: <http://www4.in.tum.de/proj/tool/visual-shape>

Evaluierte Version: Fertig gestelltes Produkt von 1999

Kriterium	Anmerkung		Note
Einarbeitung	Entwicklung	Eine sehr kurze Anleitung zur Erstellung eines Editors ist vorhanden. Die Verwendung des Rahmenwerks zur Erstellung eines Editors wird durch zwei implementierte Beispiele für Editoren gezeigt. Ein prototypischer Systemstrukturdiagramm-Editor dient als Ausgangsbasis für die Erstellung des Editorenkonzepts.	4
	Architektur	Die Architektur und die wichtigsten Konzepte sind kurz beschrieben.	3
	Strukturierung	Die Klassen sind entsprechend ihrer Aufgaben in Pakete aufgeteilt. Das Rahmenwerk ist durch den Einsatz von Schnittstellen modularisiert, wodurch die Anpassung und das Austauschen der Implementierungen einzelner Komponenten unterstützt wird. Das Rahmenwerk ist also nach dem Baukastenprinzip entwickelt.	5
	Verwendung	Ein einfacher implementierter Editor erleichtert den Einstieg bei der Erstellung eines prototypischen Editors. Die Verwendung der Komponenten und das Vorgehen bei der Erstellung eines Editors sind kurz beschrieben.	3
	Javadoc	Ist nicht vorhanden.	1
	Einarbeitungszeit	Obwohl das Rahmenwerk nicht ausführlich dokumentiert ist, ist eine Einarbeitung innerhalb von zwei Wochen möglich, da der Umfang der Software überschaubar ist.	4
Lizenz	Eigenentwicklung am Lehrstuhl.		5
Stabilität	Das Rahmenwerk ist zwar stabil, dennoch ist eine Weiterentwicklung nötig.		3
Grafische Objekte	Beliebige Arten von Figuren können durch eine Vererbungshierarchie konstruiert werden. Die Standardbibliothek an grafischen Objekten ist jedoch nicht umfangreich.		3
Präsentation	Die Oberfläche entspricht dem heutigen Stand der Entwicklung.		5
Plattform	Die Entwicklungssprache ist Java. Deshalb kann das Rahmenwerk auf allen Plattformen mit einer JVM-Implementierung eingesetzt werden.		5

Visual Shape

Kriterium	Anmerkung	Note
Anbindung Quest	Die Anbindung an ein Dateisystem oder ein objektorientiertes Modell wird unterstützt. Ein prototypischer Systemstrukturdiagramm-Editor für Quest ist bereits implementiert. Die Quellen sind verfügbar.	5
Weiterentwicklung	Die Entwicklung von Visual Shape ist abgeschlossen. Die nötige Weiterentwicklung und Pflege des Rahmenwerks wird im Falle der Entscheidung für Visual Shape als Basis für Quest-Editoren von Lehrstuhlmitarbeitern der Technischen Universität München fortgeführt.	5
Gesamtbewertung		4,3

Tabelle A.19. Bewertung der Ausschlusskriterien für Visual Shape

Kriterium	Anmerkung	Note	
Komplexität	Das Rahmenwerk für die Implementierung eines Editors umfasst den Umfang von 75 Java-Klassen.	5	
Benutzerschnittstelle	Bedienung	Die Bedienung des prototypischen Systemstrukturdiagramm-Editors ist intuitiv.	5
	Funktionen	Das Rahmenwerk umfasst die Funktionalität zum Erstellen, Verschieben, Kopieren, Einfügen, Ausschneiden, zur Größenänderung und Selektion von grafischen Objekten. Das Drucken der Zeichnungen ist möglich. Das Öffnen, Speichern und die Beschriftung von grafischen Objekten müssen vom Entwickler implementiert werden.	4
	Navigation	Vergrößern und Verkleinern vereinfacht die Navigation in umfangreichen Zeichnungen.	5
Ankopplung Quest	MVC-Konzept	Ein implementiertes MVC-Konzept ist nicht explizit erwähnt. Außerdem bietet das Rahmenwerk kein eigenes Modell, in dem die grafischen Objekte und ihre Beziehungen strukturiert werden. Die Figuren können jedoch die Schnittstelle <code>Observer</code> aus Quest implementieren und sich bei einem View-Element anmelden. Die Anzeige wird somit aktualisiert, sobald Änderungen an den View-Elementen durchgeführt werden.	2
	Ereignis-Mechanismen	Die Anzeige kann durch den Observer-Mechanismus aktualisiert werden, auch wenn in einem anderen Editor eine Änderung an den View-Elementen durchgeführt wird. Ereignis-Mechanismen sind explizit nicht erwähnt.	2

Kriterium	Anmerkung		Note
Erweiterungs- und Anpassungsfähigkeit	Die Erweiterung des AUTOFOCUS-Metamodells um ein neues Element resultiert in einer entsprechenden Anpassung des Editors. Dabei wird ein neues grafisches Objekt aus der Vererbungshierarchie des Rahmenwerks zur Repräsentation des neuen Modellelements erstellt.		5
Integration von zusätzlicher Funktionalität	Benutzerdefinierte Funktionen	Die Implementierung eigener Funktionalität ist möglich. Das Rahmenwerk ist nach dem Baukastenprinzip entwickelt, um die Erweiterbarkeit und Flexibilität zu gewährleisten.	5
	Folgen von Operationen	Die Implementierung von Sequenzen von Elementaroperationen ist möglich.	5
Integrierte Benutzerführung	Einhängepunkte für die Funktionalität zum Überprüfen von Syntaxregeln bei der Konstruktion von Diagrammen sind nicht implementiert.		2
Mehrbenutzerfähigkeit	Wird nicht unterstützt.		1
Layout	Einbettung von Algorithmen	Die Einbindung von externen Layout-Verfahren wird durch eine universelle Layoutschnittstelle ermöglicht.	5
	Integrierte Algorithmen	Eine Standardbibliothek von Layout-Algorithmen ist vorhanden.	5
Quelltext	Ist verfügbar.		5
Gesamtbewertung			3,9

Tabelle A.20. Bewertung der erweiterten Kriterien für Visual Shape

Gesamtbewertung aus Ausschluss- und erweiterten Kriterien: 4,1

16 Weitere Software

Die folgenden Programmsysteme sind nicht detailliert nach den Ausschlusskriterien bewertet, weil die Untersuchung ihrer Eigenschaften zeigt, dass sie als Basis für Quest-Editoren nicht eingesetzt werden können. Die Merkmale, die zu einem Ausschluss aus der weiteren Evaluierung führen, sind in Abschnitt 3 auf Seite 26 beschrieben.

- Microsoft Visio

Quelle: <http://www.microsoft.com/office/visio/>

Evaluierte Version: 2002

- Rahmenwerk Arakhnê

Weitere Software

Quelle: <http://www.arakhne.org>

Evaluierte Version: 0.6

– Eclipse Platform

Quellen: <http://dev.eclipse.org> und <http://www.eclipse.org>

Evaluierte Version: 2.0

Erläuterungen zur Implementierung

1 Software-Versionen

Die Implementierung des Prototyps ist mit den Java-Versionen JDK 1.3.1 und 1.4.0 unter Linux und Windows getestet. Die Realisierung des Editors basiert auf der Version 0.96 des Rahmenwerks GEF.

2 Paketstruktur

Die Paketstruktur für den erstellten Systemstrukturdiagramm-Editor ist in Abbildung B.1 dargestellt. Im Paket `ssdeditor` befindet sich das Hauptprogramm `QuestBrowser`, aus dem die Instanzen von Editorfenstern `EditorFrame` erzeugt werden. Alle Klassen zur Verarbeitung von Editorbefehlen sind im Paket `ssdeditor.base` gesammelt. Die Klassen zur Darstellung der Benutzerschnittstelle befinden sich im Paket `ssdeditor.ui`. Die Hilfsstrukturen sind im Paket `ssdeditor.util` zusammengefasst. Die Erweiterungen für die View-Elemente sind im Paket `quest.metamodel.views.SSDView.ext` enthalten.

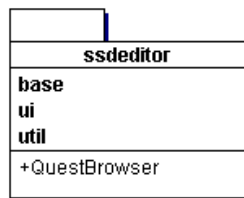


Abbildung B.1. Überblick über die Paketstruktur des prototypischen Editors.

In Abbildung B.2 auf der nächsten Seite sind die ergänzten Schnittstellen und Klassen beschrieben. In Abbildung B.3 auf der nächsten Seite sind alle Klassen und Schnittstellen des Prototyps dargestellt.

3 Erweiterungen am GEF-Rahmenwerk

Die für die Implementierung des Prototyps erforderlichen Erweiterungen am GEF-Rahmenwerk sind in Tabelle B.1 auf Seite 137 zusammengefasst. Dabei wird unterschieden, ob die Realisierung der Funktionalität durch GEF, ArgoUML [Arg02] oder durch eine eigene Implementierung erfolgt.

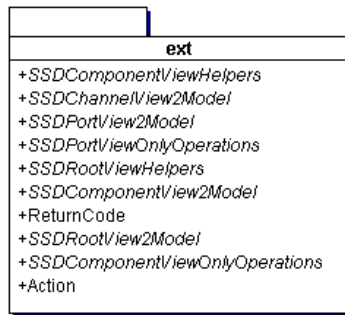


Abbildung B.2. Erweiterungen der View-Elemente.

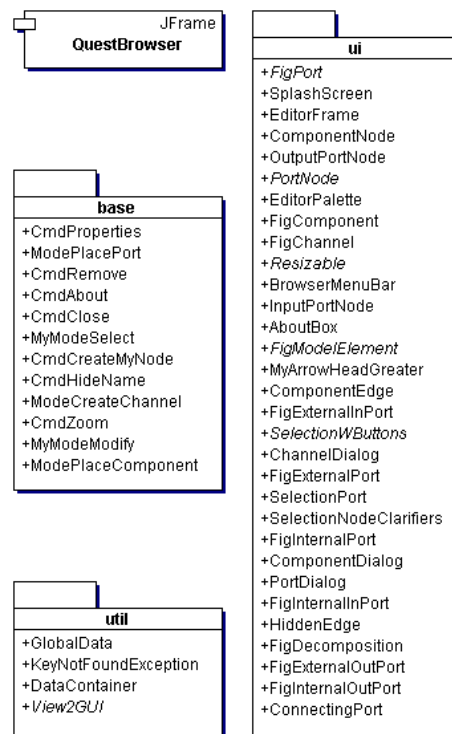


Abbildung B.3. Überblick über alle Klassen und Schnittstellen des implementierten Systemstrukturdiagramm-Editors.

Editor-Basisfunktionalität	Realisierung
Zoom	Eine Realisierung der Zoom-Funktion ist im Rahmenwerk enthalten, jedoch ist die Implementierung fehlerhaft. Deshalb ist die Klasse <code>CmdZoom</code> als Kopie der Klasse <code>ActionZoom</code> aus ArgoUML in den Editor integriert.
Einfügen eines Kanals	Aus der Klasse <code>ModeCreateEdgeAndNode</code> entstand durch eine eigene Anpassung die Klasse <code>ModeCreateChannel</code> zum Einfügen eines Kanals.
Anzeige von Informationen über den Editor	Die Klasse <code>AboutBox</code> aus ArgoUML bildet die Basis für <code>AboutBox</code> .
Figuren für Ports	Die Klassen <code>FigExternalInPort</code> , <code>FigExternalOutPort</code> , <code>FigInternalInPort</code> und <code>FigInternalOutPort</code> basieren auf der Klasse <code>FigInitialState</code> der ArgoUML-Software. Die Klasse aus ArgoUML ist durch eigene Erweiterungen an die Anforderungen angepasst.
Anzeige von Knöpfen in der Zeichenfläche zum Zeichnen von Kanälen	Für das Anzeigen von Knöpfen bei Figuren für Ports in der Zeichenfläche, die das Einfügen von Kanälen erlauben, werden einige Klassen aus ArgoUML verwendet: Die Klasse <code>SelectionNodeClarifiers</code> entspricht der ArgoUML-Klasse <code>SelectionNodeClarifiers</code> . <code>SelectionActionState</code> der ArgoUML-Software bildet die Basis für <code>SelectionPort</code> . Die Klassen aus ArgoUML sind an die Anforderungen der Anwendung durch eigene Ergänzungen angepasst. Die Klasse <code>SelectionWButtons</code> entspricht der Implementierung der Klasse <code>SelectionWButtons</code> aus ArgoUML.
Splash Screen	<code>SplashScreen</code> basiert auf der ArgoUML-Klasse <code>SplashScreen</code> . Die Klasse für den Prototyp ist dabei an die gewünschten Eigenschaften angepasst.
Selektion von Figuren	Der Modus <code>MyModeSelect</code> zum Selektieren von Figuren wird in der Klasse <code>EditorFrame</code> eingestellt. Ersterer entspricht der Implementierung aus dem Rahmenwerk GEF, mit dem Unterschied, dass der Modus zum Modifizieren von Figuren <code>MyModeModify</code> benutzerdefiniert ist und deshalb von der Klasse <code>MyModeSelect</code> eingestellt werden muss.
Verschieben und Größenänderung	Die Klasse <code>MyModeModify</code> ist an die Implementierung der Klasse <code>ModeModify</code> aus GEF angelehnt. Im Unterschied zu letzterer realisiert diese Klasse die Veränderung der Anzeigeeigenschaften nicht direkt am Editor, sondern ruft Methoden zum Modifizieren von View-Eigenschaften auf. Erst durch die Benachrichtigung des Editors durch die View-Ebene wird die Anzeige aktualisiert.

Tabelle B.1. Zusammenfassung der Erweiterungen am Rahmenwerk GEF.

4 Erweiterungen an Quest

Für die Erstellung des Editorenkonzepts werden ausschließlich an den View-Klassen Erweiterungen vorgenommen. Im Einzelnen werden dabei die folgenden Klassen um die Funktionalität ergänzt, die in den Java-Schnittstellen aus dem Paket `quest.metamodel.views.SSDView.ext` zusammengefasst ist:

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

Literaturverzeichnis

- [AF02] AUTOFOCUS, Technische Universität München, Fakultät für Informatik, <http://autofocus.in.tum.de>, Version 0.7.7, 2002. 1
- [AG89] D. Robson, A. Goldberg: *Smalltalk-80: The Language*, Addison-Wesley, 1989. 2.5
- [Arg02] ArgoUML, <http://argouml.tigris.org>, 2002. 3.4.4, 6, 3
- [BCK98] L. Bass, P. Clements R. Kazman: *Software Architecture in Practice*, Addison-Wesley, 1998. 2.2
- [BLS02] P. Braun, H. Lötzbeyer, O. Slotosch: *Quest Users Guide*, Technische Universität München, Fakultät für Informatik, 2002. 2.1, 3.1, 3.2.1
- [BLSS00] P. Braun, H. Lötzbeyer, B. Schätz, O. Slotosch: *Consistent Integration of Formal Methods*, Tools for the Analysis of Correct Systems (TACAS), 2000. 2.1
- [BS99] Dr. M. Broy, O. Slotosch, *Enriching the Software Development Process by Formal Methods*, Current Trends in Applied Formal Methods 98, LNCS 1641, 1999. 2.1
- [Cou87] J. Coutaz: *PAC, An Implementation Model for Dialog Design*, erschienen in Bullinger, H. J. Shackel (Hrsg.), INTERACT'87, North Holland, Elsevier, S. 431–436, 1987. 1.3
- [EG95] R. Johnson, J. Vlissides, E. Gamma, R. Helm: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 1.1.2, 2.5
- [FH02] B. Schätz, O. Slotosch, A. Vilbig, F. Huber, S. Molterer: *Traffic Lights – An AUTOFOCUS Case Study*, Technische Universität München, Fakultät für Informatik, 2002. 1
- [For00] OMG UML Revision Taks Force: *OMG UML 1.3*, 2000. 3.4.4, A.5
- [GWG95] M. Grochtmann, J. Wegner, K. Grimm: *Test Case Design Using Classification Trees and the Classification-Tree Editor*, erschienen in Proceedings of 8th International Quality Week, San Francisco, 1995. 2.1
- [Hah01] G. Hahn: *Konzeption einer Multi-User-fähigen Datenbankanbindung für das metamodellbasierte CASE-Tool AUTOFOCUS/Quest*, Diplomarbeit, Technische Universität München, Fakultät für Informatik, 2001. 1
- [HPS02] F. Huber, A. Pretschner, O. Slotosch et al.: *Model Based Development of Embedded Systems*, Technische Universität München, Fakultät für Informatik, 2002. 7
- [HR83] T. Härder, A. Reuter: *Principles of Transaction-Oriented Database Recovery*, erschienen in ACM Computing Surveys, Ausgabe 15(4), S. 287–317, 1983. 1

- [HS01] F. Huber, B. Schätz: *Integrated Development of Embedded Systems with AUTOFOCUS*, Technische Universität München, Fakultät für Informatik, 2001. 7
- [KP88] G. Krasner, S. Pope: *A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-8*, erschienen in *Journal of Object-Oriented Programming*, Ausgabe 1(3), S. 26–49, 1988. 3.4.2, 3.4.3, 3.4.4, 3.4.6, 1.3, 2.1.1
- [Löt99] H. Lötzbeyer: *Layoutmechanismus für das Editor-Framework, Internal Draft*, Technische Universität München, Fakultät für Informatik, 1999. 3.2.3
- [Löt02] H. Lötzbeyer: *AUTOFOCUS 2000 – Struktur und Visualisierung*, Technische Universität München, Fakultät für Informatik, 2002. 2.4
- [Mai99] Th. Maier: *Informationen zu Visual Shape*, Technische Universität München, Fakultät für Informatik, 1999. 3.4.3
- [Mar98] F. Marschall: *Konzeption und Realisierung einer generischen Schnittstelle für metamodel-basierte Werkzeuge*, Diplomarbeit, Technische Universität München, Fakultät für Informatik, 1998. 2.1, 2.3
- [McM92] K. L. McMillan: *The SMV System, Symbolic Model Checking – An Approach*, Carnegie Mellon University, 1992. 2.1
- [Met02] MetaEdit+, <http://www.metacase.com>, Version 3.0, 2002. 3.2
- [Mey00] B. Meyer: *Object-Oriented Software Construction*, Prentice-Hall, 2. Auflage, 2000. 2
- [MF] D. C. Schmidt, M. Fayad: *Special Issue on Object-Oriented Application Frameworks*, erschienen in *Communications of the ACM*, Ausgabe 40 No. 10, 1997. 2
- [NC91] L. Nigay, J. Coutaz: *Building User Interfaces: Organizing Software Agents*, ES-PRIT '91 Konferenz, Brüssel, Belgien, November 1991. 1.3
- [OO] Dr. Ch. Prehofer: *Unterlagen zur Vorlesung „Objektorientierung“ im Sommersemester 2001*, Technische Universität München, Fakultät für Informatik, 2001. 2.5, 2.5
- [PB02a] O. Slotosch, P. Braun, H. Lötzbeyer: *Project Quest, Integrated Metamodels for AUTOFOCUS*, Technische Universität München, Fakultät für Informatik, 2002. 1.2
- [PB02b] O. Slotosch, P. Braun, H. Lötzbeyer: *Quest Developers Guide*, Technische Universität München, Fakultät für Informatik, 2002. 2.1, 2.2, 2.3, 2.4
- [Pfa85] G. E. Pfaff: *User Interface Management Systems*, Springer-Verlag, 1985. 1.3
- [POM] Dr. M. Broy: *Mitschrift zur Vorlesung „Projektorganisation und Management in der Softwareentwicklung“ im Sommersemester 2000*, Technische Universität München, Fakultät für Informatik, 2000. 1

- [PR97] G. Pomberger, P. Rechenberg: *Informatik-Handbuch*, Carl Hanser Verlag München Wien, 1997. 2, 1
- [Pre94] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1994. 2
- [Que02] Quest, Technische Universität München, Fakultät für Informatik, <http://www.broy.informatik.tu-muenchen.de/proj/quest/>, 2002. 2
- [RJ88] B. Foote, R. Johnson: *Designing Reusable Classes*, erschienen in Journal of Object-Oriented Programming, S. 22–35, 1988. 2
- [RJB97] J. Rumbaugh, I. Jacobson, G. Booch: *Unified Modeling Language Reference Manual*, Addison-Wesley Longman, 1997. 3.2, 3.4.5, 3.3
- [RSW97] G. Rock, W. Stephan, A. Wolpers: *Tool Support for the Compositional Development of Distributed Systems*, erschienen in Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme, Nummer 315 GMD Studien, 1997. 2.1
- [Sch02] B. Schätz: *The ODL Operation Definition Language and the AUTOFOCUS/Quest Application Framework AQUA*, Technische Universität München, Fakultät für Informatik, 2002. 7
- [SE] Dr. M. Broy: *Mitschrift zur Vorlesung „Software Engineering: Softwaretechnik“ im Wintersemester 1999/2000*, Technische Universität München, Fakultät für Informatik, 2000. 3
- [Str99] B. Stroustrup: *The C++ Programming Language*, Addison-Wesley, 3. Auflage, 1999. 2.1
- [UBR⁺99] Ullmann, Baur, Reif, Siekmann, Scheer, Moik: *Specification Language VSE SL Version 2*, 1999. 2.1
- [Val02] Validas Model Validation AG, München: *Validation Guide for Validas Validator Version 1.4.1*, 2002. 2.4, 9
- [Vis02] Microsoft Visio, <http://www.microsoft.com/office/visio/>, 2002. 3.1
- [Wil02] S. Wildgruber: *Modellbasiertes Hilfesystem für AUTOFOCUS 2*, Systementwicklungsprojekt, Technische Universität München, Fakultät für Informatik, 2002. 3.2.4, 7.9, 2
- [Wor92] UIMS Tool Developers Workshop: *A Metamodel for the Runtime Architecture of an Interactive System*, erschienen in SIGCHI Bulletin, Ausgabe 24(1), S. 32–37, 1992. 1.3
- [Zha97] H. Zhang: *SATO: An Efficient Propositional Prover*, erschienen in Springer, W. McCune (Hrsg.), 14th International Conference on Automated Deduction, Ausgabe 1249, S. 272–275, 1997. 2.1

Index

A	
ACID	105
Arch	50
Arch/Slinky	50
Architektur	47
Atomarität	105
Atomicity	105
B	
Black Box	10, 47
C	
CASE-Werkzeug	5
Consistency	105
D	
Dauerhaftigkeit	105
Design Patterns	47
Diagramm	
EET	8, 11
SSD	8
STD	8, 11
Durability	105
E	
EET-Diagramm	11
Ereignis	12
F	
Framework	19, 48
G	
Geheimnisprinzip	48, 50
Glass-Box	10
H	
Hook	48
I	
Information Hiding	47
Isolation	105
K	
Komponente	47
M	
Konsistenz	105
M	
Meta-CASE-Werkzeug	27
Metamodell	5
integral	8
Metaview	8
Metaviews	5
MIF-Assoziation	7
MMGen	14
Model-Checker	12
Unbounded	12
Model-Checking	11
ModelChangeEvent	12
ModelChangeListener	12
Modul	47
MVC	50
O	
Objektmenge	8
Observer	55
P	
PAC	50
PAC-AMODEUS	50
Projekt	7
Q	
QML	12
Qualitätsmerkmal	
funktional	18
nicht funktional	18
technisch	18
R	
Rahmenbedingungen	19
Rahmenwerk	19, 48
Repository	7
dokumentbasiert	5
modellorientiert	5
S	
Schlüssel	16

Seeheim	49
Sicht	10
extern	10
intern	10
Simulation	5
Sperren	
Granularität	103
SSD-Diagramm	11

T

Theorembeweiser	12
Transaktion	105

V

Validierung	5, 12
Verifikation	11
Verteiltes System	5
View	10

W

Wurzelobjekt	8
--------------------	---

