Bachelorarbeit

Konzeption und Implementierung eines graphischen Editors für verschiedene Notationen im Softwareentwicklungsprozess

Fachbereich Softwaretechnik
Prof. Dr.-Ing Stefan Jähnichen
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

vorgelegt von
Peter Kolbe - 314201

Betreuer: Alexander Rein

Eingereicht am 4. Januar 2011

Zusammenfassung

Das Thema dieser Arbeit ist die Konzeption und Implementierung eines graphischen Editors für verschiedene Notationen im Softwareentwicklungsprozess. Zu diesem Zweck wird als Grundlage der anpassbare Softwareentwicklungsprozess Rational Unified Process (RUP) vorgestellt. Basierend auf diesem, wird eine spezifizierte Methodik eingeführt, die eine wesentlich reduzierte Komplexität aufweist und wie für RUP typisch, stark mit der Erstellung und Pflege visueller Modelle verbunden ist. Ein graphischer Editor kann für solch eine Methodik mehr, als nur die Darstellung von verschiedenen Notationen bieten. Die Analyse von sinnvollen Unterstützungmöglichkeiten sowie deren Umsetzung wird deswegen eine entscheidende Rolle in dieser Arbeit spielen. Um die korrekte Funktionsweise des Editors nachzuweisen, wird außerdem eine Software demonstriert, welche die Automatisierung von Tests ermöglicht.

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 4. Januar 2011

.....
Peter Kolbe

Abkürzungsverzeichnis

- RUP - Rational Unified Process
- KMG - Klassenmodell des Gegenstandsbereiches
- UCD - Use-Case-Diagramm
- SD - Sequenzdiagramm
- AD - Aktivitätsdiagramm
- SKM - Systemklassenmodell
- KD - Kommunikationsdiagramm
- EKM - Entwurfsklassenmodell
- IKM - Implementationsklassenmodell
- OMG - Object Management Group
- SPEM - Software Process Engineering Metamodel
- UMA - Unified Method Architecture
- GUI - Graphical User Interface
- RSM - Rational Software Modeler
- RSA - Rational Software Architect

Inhaltsverzeichnis

1 Einleitung

Eine wesentliche Eigenschaft der Softwareentwicklung ist ihre Komplexität. Das Problem ist im Grunde genommen eines des Projektmanagements: Ein Kunde bzw. Auftraggeber formuliert seine Vorstellungen vom gewünschten Produkt und ein Auftragnehmer versucht diese umzusetzen. Um sicher zu gehen, dass der Auftragnehmer letztlich auch ein Ergebnis abliefert, dass den Ansprüchen des Auftraggebers entspricht, wird ein Dokument, das sogenannte *Pflichtenheft* angelegt, welches alle Anforderungen an das Produkt in textueller Form beinhaltet und rechtlich gültig ist. Dabei bleibt aber trotzdem unter Umständen viel Spielraum für Missverständnisse übrig. Damit diese nicht ausufern, muss das zu entwickelnde System von allen Beteiligten verstanden werden, was mitunter große Schwierigkeiten bereiten kann. Umso komplexer das zu entwickelnde System ist, desto schwerer ist letztlich auch die Kommunikation zwischen den am Projekt beteiligten Parteien. Die Auswirkungen vom fehlendem Verständnis der Anforderungen ist in in Abbildung 1.1 recht deutlich zu erkennen.

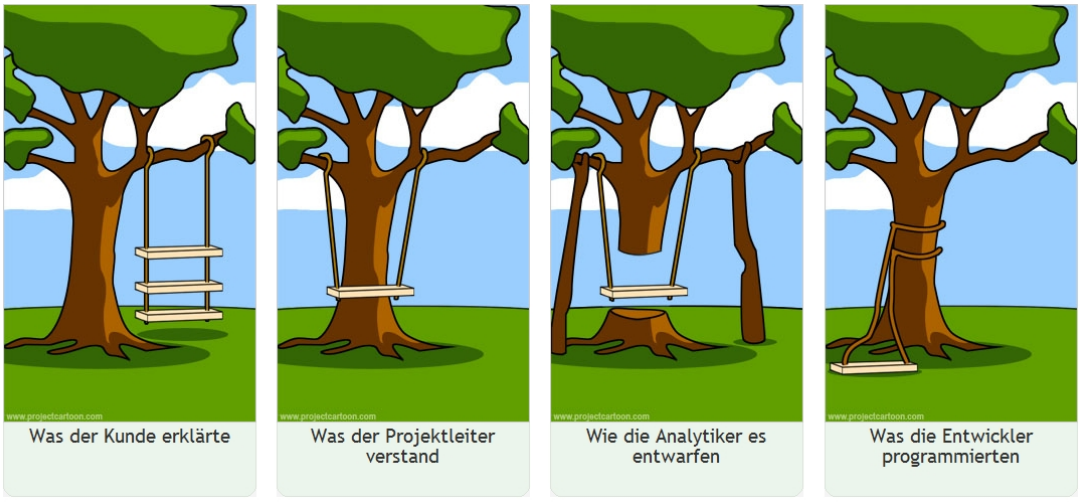

Abbildung 1.1: Das Projekt Schaukel - wenn Kommunikation versagt

Aufgrund dessen werden zur Entwicklung von Software vermehrt visuelle Modelle eingesetzt. Ein Modell ist dabei eine vereinfachte Sicht auf einen bestimmten Aspekt des zu entwickelnden Systems, welche eine Fülle an, für den Betrachter irrelevanten, Informationen vernachlässigt. So können Aufgaben und Ziele sowohl innerhalb eines Teams als auch bei der Zusammenarbeit mit anderen wesentlich einfacher, schneller und vor allem verständlicher kommuniziert werden.
Mithilfe von Software-Modellierungstools können Modelle zudem bequem angelegt, erweitert, gewartet und schließlich auch für ähnliche Projekte wiederverwendet werden. Auch die *National Aeronautics and Space Administration (NASA)* scheint diese Vorteile erkannt zu haben und beschloss daher 2007, für die Entwicklung der Software des James Webb Space Telescopes[1] visuelle Software-Modellierung als Pflicht für dieses Projekt durchzusetzen. Dafür soll das von IBM entwickelte Tool *Rational Rose* eingesetzt werden.[2]
Solche Entwicklungs-Tools gewinnen also offensichtlich immer mehr an Bedeutung. Der Nachteil bestehender Lösungen, wie *Rational Rose*, *Rational Software Modeler (RSM)* oder *Rational Software Architekt (RSA)*[3] besteht hauptsächlich darin, dass sie darauf ausgelegt sind, für möglichst viele Projektarten geeignet zu sein. Trotz diverser Optionen zur Anpassung, fehlen diesen Tools jedoch oft sinnvolle Möglichkeiten, den Benutzer zu unterstützen. Das Ziel dieser Bachelorarbeit ist vor allem zu zeigen, dass für einen graphischen Editor als solch ein Tool, sinnvolle Möglichkeiten zur Unterstützung eines Benutzers existieren, insofern eine klar vorgegebene Methodik zugrunde liegt. Dafür soll als Grundlage der *Rational Unified Process (RUP)*, als ein auf visueller Software-Modellierung basierender Softwareentwicklungsprozess, dienen.

[1] Das James Webb Space Telescope ist ein geplantes Weltrauminfrarotteleskop unter der Kooperation von NASA, ESA und der kanadischen Weltraumagentur, das 2014 in Betrieb genommen werden soll - http://www.jwst.nasa.gov
[2] IBM Presse - http://www-03.ibm.com/press/us/en/pressrelease/20901.wss
[3] IBM Rational Software - http://www-01.ibm.com/software/de/rational/

1.1 Aufbau

Der Aufbau dieser Arbeit beginnt demzufolge mit der Erklärung der Grundlagen von **RUP**, woraufhin eine Anpassung dessen, im Kapitel **Beschreibung der Methodik** erarbeitet wird. Dort werden außerdem Unterstützungsmöglichkeiten für diese spezifizierte Methodik analysiert und als Anforderungen für den zu entwickelnden graphischen Editor festgehalten. Anschließend wird im Kapitel **Implementierung** die Umsetzung des graphischen Editors erläutert, wozu auch die Beschreibung der verwendeten Technologien fällt. Daraufhin folgt letztlich die Vorstellung einer Software zum Testen des entwickelten Editors im Kapitel **Test**, wo besonders auf das Anlegen automatisierter Testfälle eingegangen wird.

2 RUP

> ,,Die Ausprägung technologisch möglicher und von der Gesellschaft geforderter Softwaresysteme gewinnt zunehmend an Größe, Komplexität, Verteilungsumfang und Wichtigkeit." [Kru99]

Aus dieser, von Grady Booch, einem Mitbegründer des RUP, verfassten Aussage, resultiert auch gleichzeitig der wachsende Anspruch an die Organisation von Software-Entwicklung. Deren Vernachlässigung hat sonst oft Probleme wie unzureichende Software-Qualität (u.a. Performanz, Nutzerfreundlichkeit, Zuverlässigkeit), schlechte Wartbarkeit und Wiederverwendbarkeit der Software oder Probleme beim Reagieren auf wechselnde Anforderungen zur Folge. Mit steigender Komplexität von Software-Entwicklungsprojekten, nimmt außerdem die Anzahl der beteiligten Mitarbeiter zu, sodass auch die Organisation ihrer Zusammenarbeit Teil eines Konzepts sein sollte, das einen Entwicklungsprozess umfassend genug beschreibt, wie es in anderen Ingenieurs-Disziplinen längst gang und gäbe ist.
In den vielen anderen Ingenieursdisziplinen wird als Methodik für das Vorgehen das sogenannte *Wasserfallmodell* angewandt. Es beschreibt einen Prozess, bei dem einzelne Schritte wie Anforderungsdefinition oder Design vollständig abgeschlossen werden müssen, bevor der nächste beginnen kann. Bei feststehenden Anforderungen, wie zum Beispiel beim Bau einer Brücke, ist dies kein Problem, bei Software-Projekten hingegen schon, wenn zum Beispiel innerhalb der Implementierung eine Änderung der Anforderungen ansteht. Außerdem ist das Testen beim Wasserfallmodell, was schließlich erst nach der Implementierung passieren kann, sehr gefährlich, da ernste oder sogar gravierende Fehler mitunter zu spät erkannt werden, um sie zu beheben.
Selbstverständlich existieren zur Lösung der ganzen Problematik mittlerweile mehrere Software-Entwicklungsprozesse, wobei neben den neueren *agilen* Vertretern *Extreme Programming* und *Scrum*, sich besonders der *Rational Unified Process* als

traditionelles, jedoch ständig weiterentwickeltes Vorgehensmodell etabliert hat. So orientiert sich RUP nicht nur an einem großem Repertoire praktischer Erfahrungen, sondern bietet auch eine sehr detaillierte Dokumentation sowie umfangreiche Software zur korrekten, Feature-gestützten Anwendung. [EM07]

2.1 Inhalte

In diesem Abschnitt werden zunächst die wesentlichen Prinzipien erläutert, nach denen sich RUP richtet. Auf diesen basierend, werden danach die drei Hauptmerkmale von RUP zusammengefasst und ihr Zusammenspiel beschrieben.

2.1.1 Best Practices

Der *Rational Unified Process* beinhaltet diverse *Best Practices*, also bewährte Prinzipien, die auf praktischen Erfahrungen beruhen. Sie repräsentieren Leitsätze und Kriterien, nach denen Projekte geführt und bewertet werden sollen. Die wichtigsten stellen dabei folgende dar:

- Prozessanpassung
- Anforderungsmanagement
- Teamübergreifende Zusammenarbeit
- Erhöhte Abstraktion
- Iterativer Entwicklungsprozess
- Verwendung komponentenbasierter Architekturen
- Ständige Prüfung der Software-Qualität

Zum besseren Verständnis wird vorher jedoch der Begriff *Stakeholder* erläutert: Stakeholder sind beliebige Personen, die ein berechtigtes Interesse am Ergebnis des Projekts haben, wie zum Beispiel Kunde, Projektmanager Entwickler, Designer etc.

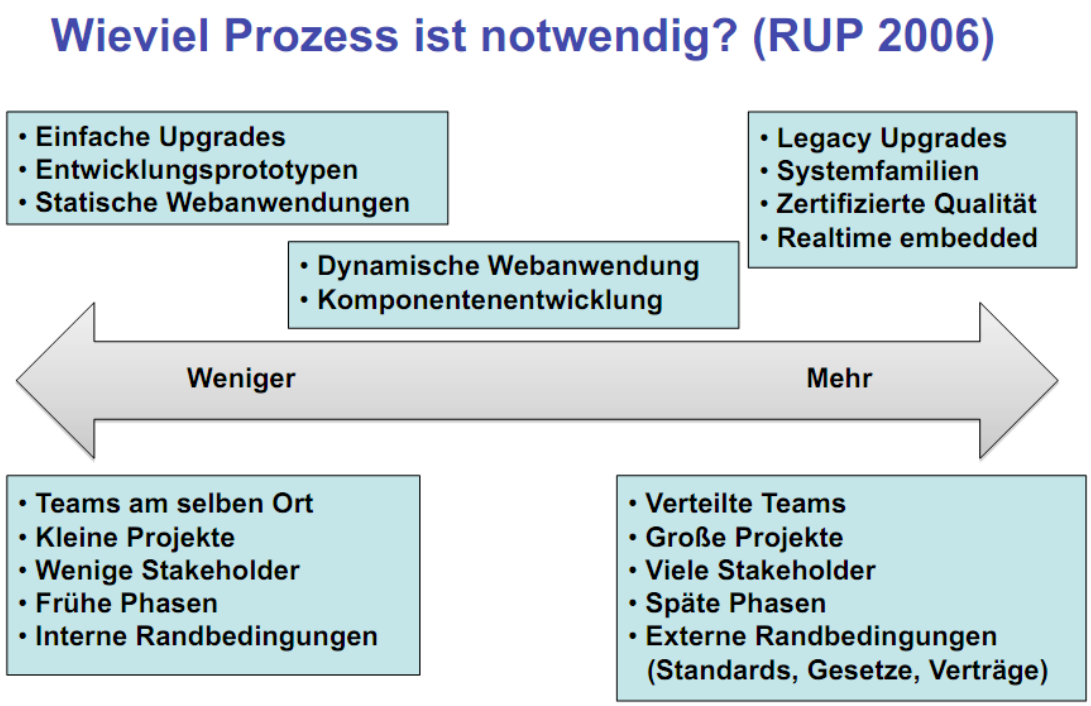

Abbildung 2.1: Anpassung von RUP an ein Projekt [DISJ09]

Prozessanpassung

Der Prozess sollte sehr genau an den Charakter eines Projekts angepasst werden, um möglichst effizient und flexibel eingesetzt zu werden. So müssen einzelne Bestandteile des Prozesses auf verschiedene Faktoren abgestimmt sein, wie zum Beispiel auf die Größe der einzelnen Teams, deren räumliche Entfernung voneinander sowie auf den Umfang der Anwendung selbst. Gerade im Projektverlauf spielen Anpassungen eine wichtige Rolle, um zur kontinuierlichen Verbesserung des Prozesses beizutragen. So sollte zum Beispiel am Anfang eines Projekts, ein Prozess weniger umfangreich sein, damit Pläne nicht zu detailliert ausfallen und somit wenig Raum für Überarbeitung bzw. Optimierung bleibt. Zusammenfassend sei gesagt, dass ein Prozess immer darauf abgestimmt werden sollte, ,,wieviel" von ihm letztlich notwendig bzw. sinnvoll für das Projekt ist, wie in Abbildung 2.1 angedeutet wird.

Anforderungsmanagement

Unter Anforderungsmanagement versteht man das systematische Erfassen, Organisieren, Dokumentieren und Verfolgen der sich ändernden Anforderungen eines Systems. Eine Anforderung kann dabei eine geforderte Funktionalität oder eine Bedingung sein. In der Praxis sind Anforderungen allerdings oft nicht gleichermaßen transparent für Auftraggeber und -nehmer, da sie sich nicht immer klar formulieren lassen.
Ein gutes Anforderungsmanagement zeichnet sich besonders dadurch aus, konkurrierende Bedürfnisse auszugleichen, also eine Übereinkunft zwischen den Interessen der Stakeholder zu finden, mit Hinblick auf die Wirtschaftlichkeit. Dabei sollten bestehende Lösungen daraufhin untersucht werden, ob sie den geforderten Bedürfnissen entsprechen und so Eigenentwicklungen eingespart werden können, samt ihrer Kosten und Risiken.

Teamübergreifende Zusammenarbeit

Mit der Entwicklung der bereits eingangs erwähnten agilen Prozesse, hat auch das Thema *Teamzusammenarbeit* Einzug in RUP erhalten. Um die Produktivität zu erhöhen und Geschäftsinteressen dauerhaft besser zu kommunizieren, steht die Motivation der Mitarbeiter im Vordergrund. Dafür muss unter anderem eine geeignete Arbeitsumgebung geschaffen werden sowie eine Struktur zur Organisation entstehender Ergebnisse. Außerdem soll die Förderung der Selbstorganisation von Teams sowie die rollenübergreifende Zusammenarbeit dieser, zu einer besseren Kooperation beitragen. Als Grundlage dafür ist jedoch auch eine klare, verständliche Beschreibung des System, einschließlich seiner einzelnen Komponenten nötig.

Erhöhte Abstraktion

Das Erhöhen des Abstraktionsgrades ist ein Mittel, um die Komplexität von Systemen zu reduzieren. Indem das Verständnis des Systems gefördert wird, kann die Produktivität gesteigert und außerdem ein erheblicher Teil der Dokumentation einspart werden. Die Verwendung abstrakter, graphischer Sprachen ist ein geeignetes Mittel, um anhand visueller Modelle die einzelnen statischen und dynamischen Aspekte eines Systems verständlich abbilden zu können. In RUP wird dieses Prinzip mithilfe der *Unified Modeling Language (UML)* umgesetzt. Sie „[...] ist

eine grafische Sprache zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation der Artefakte eines Software-Systems" [Kru99]. Wichtig ist hierbei die Unterscheidung, dass *UML* die Syntax repräsentiert, also lediglich Konstrukte zur Modellierung bereitstellt, während deren Interpretation und Anwendung durch *RUP* beschrieben wird.

Iterativer Entwicklungsprozess

Wie bereits zuvor erwähnt, besteht das Problem der linearen Entwicklung nach dem *Wasserfallmodell* darin, dass Implementation und Testen zeitlich sehr spät erfolgen, wodurch das Erkennen möglicher Probleme und somit Risiken ebenfalls hinausgezögert werden. Im Nachhinein festgestellte Fehler können so mitunter ganze Projekte scheitern lassen.
Eine Lösungsmöglichkeit dieses Problems bietet die in Abbildung 2.2 dargestellte

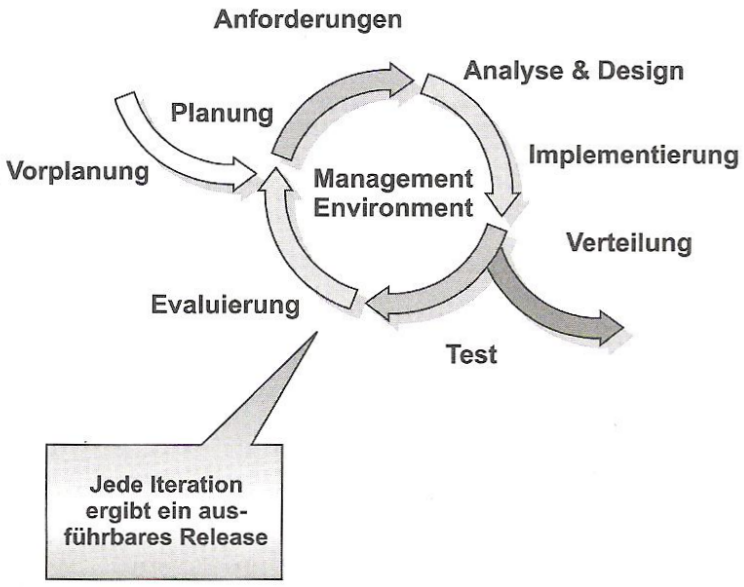

Abbildung 2.2: Eine Iteration in RUP [Kru99]

iterative und inkrementelle Entwicklung. Die einzelnen Schritte der Entwicklung werden hier mehrmals zyklisch durchlaufen (*iterativ*), sodass ein System durch die ständige Anpassung und Erweiterung seiner Artefakte entsteht (*inkrementell*). Der große Nutzen liegt darin, dass nach jeder Iteration Ergebnisse in Form eines ausführ- und auslieferbaren Releases vorliegen. So kann von Projektbetei-

ligten bereits früh Einfluss auf die Entwicklung genommen werden, um Fehler zu identifizieren und Anpassungen vorzunehmen. Diese sind dann Bestandteil der nächsten Iteration und stellen mit wachsendem Projektumfang eine Herausforderung für das Anforderungsmanagement dar.
Um Software iterativ zu entwickeln, muss bereits zu Beginn ein Prototyp der Architektur vorliegen, der schließlich immer weiterentwickelt wird, bis hin zum Endprodukt. Deshalb bietet sich eine komponentenbasierte Architektur an, welche das separate Entwickeln einzelner Module ermöglicht, die nach und nach das System erweitern. [Kru99]

Verwendung komponentenbasierter Architekturen

Architektur umfasst in RUP wesentliche Entscheidungen über die Organisation des Systems und den verwendeten Architekturstil, über die Auswahl der strukturellen Elemente und Schnittstellen, aus denen das System bestehen soll sowie über die Zusammensetzung dieser Elemente zu größeren Teilsystemen.
Komponentenbasierte Architekturen ermöglichen erst komponentenbasierte Entwicklung, deren größte Vorzüge zum einen die Wiederverwendbarkeit bestehender Komponenten und zum anderen die Möglichkeit iterativen Arbeitens sind. So können Systeme aus eigenen bzw. angepassten sowie bestehenden Teilen zusammengesetzt werden, welche dementsprechend von einander getrennt entwickelt und somit bereits früh, regelmäßig getestet werden können. [Kru99]

Ständige Prüfung der Software-Qualität

Die Vorteile früher, ständiger Qualitätsprüfungen werden relativ leicht klar, wenn Fehler eines Systems, zur Vereinfachung mit Krankheiten beim Menschen verglichen werden. Eines haben beide gemeinsam: Je länger abgewartet wird, umso größer wird das Problem und damit auch der Aufwand zu dessen Behebung. Letzteres wäre also zum Beispiel das, mit dem Fortschritt einer Krankheit eintretende, Erfordernis einer Operation wobei im schlimmsten Falle selbst diese nicht mehr ausreicht. Der Aufwand zur Behebung von Fehlern eines Systems wächst ebenfalls mit dessen Fortschritt. Zu spät entdeckte und damit kritisch gewordene Fehler können nur mit hohem Aufwand bzw. hohen Kosten behoben werden, was ebenfalls außerhalb der gegebenen Möglichkeiten liegen kann, wenn die Kosten das Budget des Projekts sprengen.

Bei der Softwareentwicklung werden folgende zwei Ausprägungen von Qualität betrachtet: Die Produktqualität sowie die Prozessqualität. Von allen Projektbeteiligten wird der kontinuierliche Fokus auf beide gefordert, um neben höherer Qualität, auch einen guten und vor allem frühen Einblick in den eigenen Prozess zu gewinnen. Außerdem kann so unter anderem vermittelt werden, wie gute Qualität erzielt wird. Wenn dieses Konzept richtig im Entwicklungsprozess verankert wird, können automatisierte Tests Stück für Stück, also *inkrementell* aufgebaut werden.

2.1.2 Hauptmerkmale

Nachdem sie bereits teilweise bei den Best Practices beschrieben wurden, werden die wichtigsten Eigenschaften von RUP hier noch einmal zusammengefasst:

- Iterativ
- Use-Case-getrieben
- Architekturzentriert

Iterativ

Das Prinzip der iterativen Entwicklung wurde ja bereits innerhalb der Best Practices weitestgehend erläutert. Zusammenfassend sei gesagt, dass es sich in erster Linie durch das flexible Verwalten von Änderungen sowie das zyklische, inkrementelle Aufbauen, Integrieren und Testen von Komponenten auszeichnet. Risiken können so früh erkannt und gebannt werden, außerdem kann neben dem Produkt auch die Prozesstruktur noch während des laufenden Projekts angepasst und optimiert werden. Ein anderer wichtiger Vorteil iterativer Entwicklung, ist das Ermöglichen komponentenweiser Entwicklung und damit das Erzielen eines höheren Wiederverwendungsgrades. [Kru99]

Use-Case-getrieben

Um funktionale Anforderungen für eine Vielzahl von Stakeholdern verständlich zu beschreiben, wird in RUP die sogenannte *Use-Case-Modellierung* eingesetzt. Dabei werden im Wesentlichen zwei Konzepte benutzt: *Akteure* und *Use-Cases*.

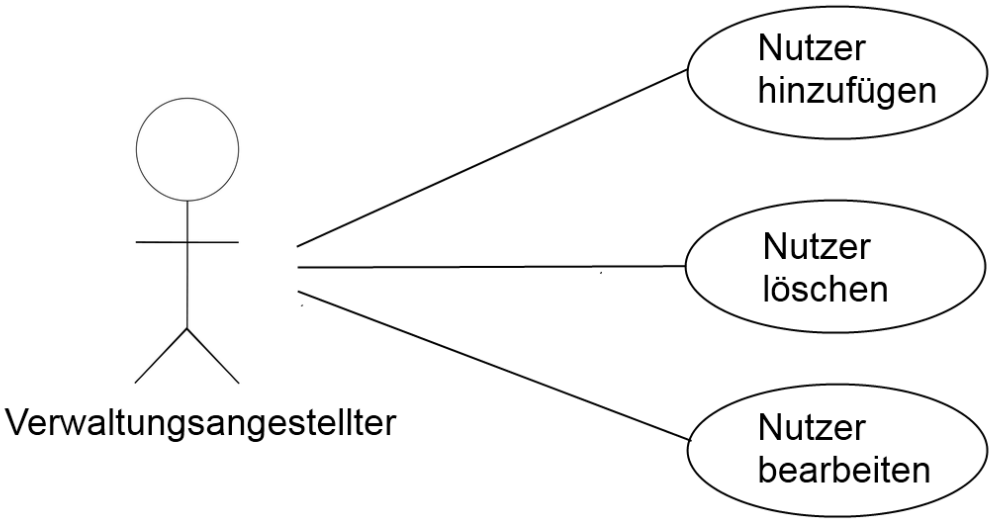

Abbildung 2.3: Use Cases für eine Nutzerverwaltung

Akteure sind außerhalb des Systems stehende Personen oder Sachen, die mit dem System interagieren. In Abbildung 2.3 wäre der Verwaltungsangestellte ein Akteur.
Use Cases (deut. Anwendungsfälle) spezifizieren diese Interaktionen. Genauer gesagt beschreibt ein Use Case eine Abfolge von Interaktionen zwischen Akteur und System, an deren Ende ein Ergebnis an den Akteur zurückgeliefert wird, das für ihn auch von Wert ist. Dabei kann ein Use Case mehrere Szenarien beinhalten, also mehrere mögliche Abfolgen. Dies könnte bei gegebenem Beispiel unter anderem beim Löschen eines Nutzers der Fall sein. Ein Szenario wäre das erfolgreiche Löschen eines in der Datenbank registrierten Nutzers, ein anderes das Nicht-Vorhandensein des zu löschenden Nutzers und die damit verbundene Fehlermeldung.
Use-Case-Modelle dienen als eine Art Vertrag zwischen Kunden und Entwicklern, da sie die gesamte Funktionalität des Systems abbilden. Außerdem bilden sie eine Grundlage für den gesamten Entwicklungsprozess, also sowohl für Analyse, Design, Implementierung, Test, als auch für die Iterationsplanung. Bei letzterer

werden Use Cases bzw. einzelne Szenarien als Inhalt eines Iterationsschrittes ausgewählt. [EM07]

Architekturzentriert

Neben strukturellen und verhaltensspezifischen Entscheidungen, die bereits im Best Practice *Verwendung komponentenbasierter Architekturen* erläutert wurden, umfasst Architektur in RUP auch Kriterien wie Benutzerfreundlichkeit (engl. Usability), Funktionalität, Performance, Wiederverwendung, Randbedingungen und Entscheidungen technischer oder wirtschaftlicher Art sowie Ästhetik. Das heißt, alle Stakeholder, sowohl Systemanalytiker, Endanwender, Projektleiter, Entwickler, etc., nutzen die Architektur als Grundlage ihrer Arbeit, sind jedoch dabei an verschiedenen Eigenschaften interessiert. Zum besseren Verständnis des Systems ist eine Darstellung notwendig, welche die Komplexität der gesamten Architektur durch die Betrachtung aus einer jeweils bestimmten Perspektive reduziert. Der von RUP angebotene Lösungsansatz für dieses Problem ist „Die 4+1 Sicht auf eine Architektur". [Kru99]
Mithilfe von Abbildung 2.4 wird kurz erläutert, welche Aspekte die jeweiligen Sichten beinhalten und für wen sie bestimmt sind:

- **Logische Sicht:** Funktionale Anforderungen in Form von Klassen und Interfaces sowie deren hierarchische Organisation in Paketen, Subsystemen und Schichten
- **Implementierungssicht:** Hierarchische Organisation der einzelnen Software-Komponenten in der Entwicklungsumgebung
- **Prozesssicht:** Gleichzeitig ablaufende Vorgänge (Aufgaben, Prozesse, Threads), deren Zusammenhang (Nebenläufigkeit, Parallelität) sowie deren Zuordnung zu Klassen
- **Verteilungssicht:** Verteilung der Software-Komponenten auf einzelne Knoten bzw. Rechner eines Netzwerks
- **Use-Case-Sicht:** Für Architektur relevante Use Cases und Szenarien, welche die Funktion der anderen Sichten illustrieren - für alle Stakeholder von Bedeutung

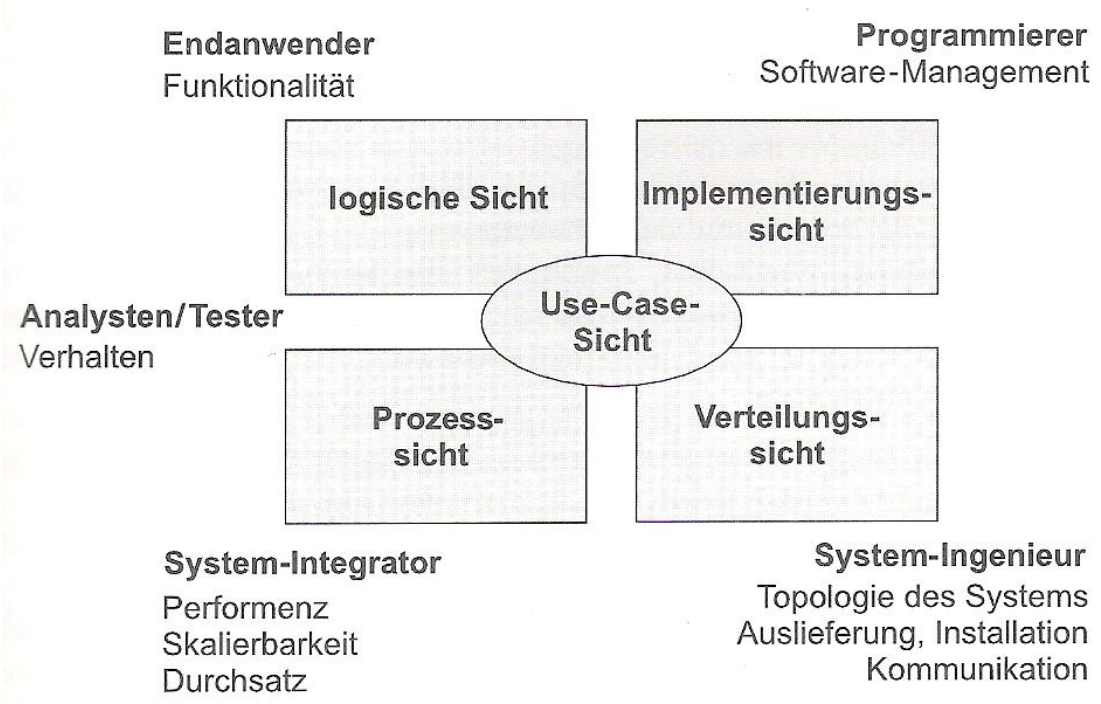

Abbildung 2.4: Die 4+1 Sicht auf eine Architektur [Kru99]

Die einzelnen Sichten der Architektur setzen sich aus Teilen der entsprechenden *UML-Modelle* zusammen, bei denen es sich, im Gegensatz zu den Sichten, um eine jeweils vollständige Darstellung des Systems handelt. Bis auf das *Designmodell*, welches die Grundlage für die logische Sicht und die Prozesssicht bildet, entsprechen die Sichten ihren gleichnamigen Modelläquivalenten. Die Sichten, die letztlich für das System von Bedeutung sind werden zusammen mit textuellen Beschreibungen im *Software Architecture Document* festgehalten.

2.2 Dynamische Struktur

Das vielleicht wichtigste Merkmal von RUP ist die iterative Vorgehensweise. Diese bringt allerdings aufgrund ihrer Struktur die Frage mit sich, wie der Fortschritt des Projekts gemessen werden kann bzw. wie letztlich die Fertigstellung des Endprodukts erzielt wird.
Dafür wird der Entwicklungsprozess eines Projekts in vier Phasen unterteilt, an deren Ende jeweils ein Meilenstein steht. Ein solcher stellt die zeitliche Begrenzung

für die Fertigstellung von Funktionalitäten, Use Cases, Tests, usw. dar. Hier wird noch einmal der Unterschied zum Wasserfallmodell klar, welches sequentiell Analyse, Entwurf, Implementierung und Test abarbeiten lässt, wohingegen in RUP diese Disziplinen mehrmals in jeder Phase ausgeführt werden, lediglich mit wechselndem Fokus. Genau dies ist auch der Inhalt des bekannten Hügel-Diagramms (Abbildung 2.5), dessen Phasen nun kurz erläutert werden. [EM07]

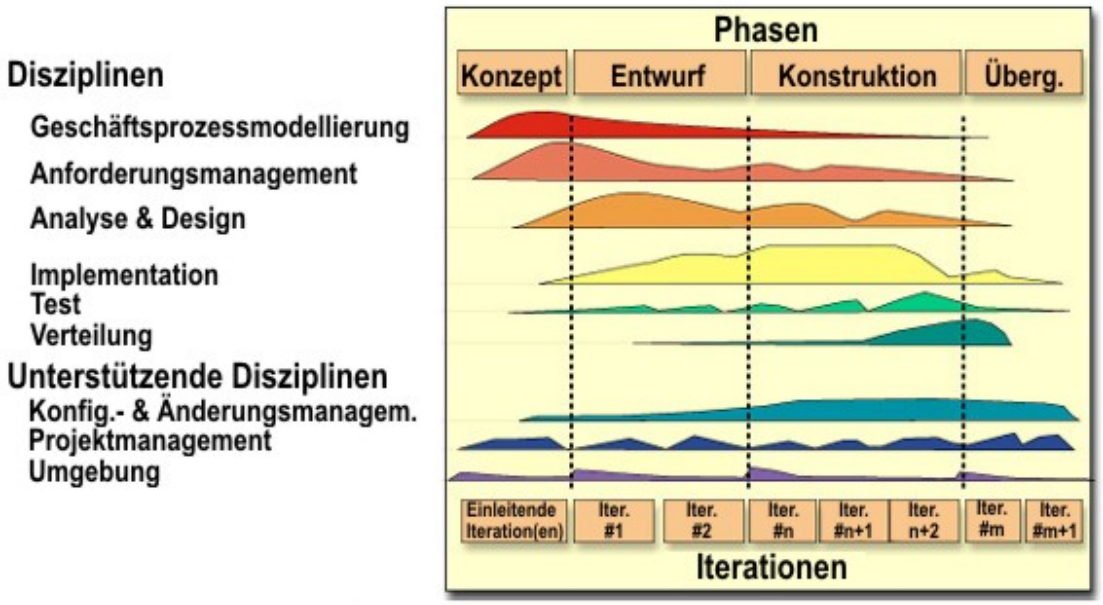

Abbildung 2.5: Hügeldiagramm (engl. Hump-Chart) des RUP

2.2.1 Konzeption (engl. Inception)

In der ersten Phase soll im Wesentlichen die *Vision* des Projekts sowie eine Risikoabschätzung für diese erarbeitet werden. Das heißt, es sollen erste grobe Entwürfe der Planung, Use Cases und Kosten-Nutzen-Analyse erstellt werden, um schon früh Umfang und Risiken des Projekts abzuschätzen zu können. Abgeschlossen wird diese Phase mit dem Meilenstein *Lifecycle Objective (LCO)*, der im Prinzip besagt, dass bis zu seinem Eintreten, die Ziele klar formuliert sein müssen.

2.2.2 Entwurf (engl. Elaboration)

In dieser Phase werden hauptsächlich die Anforderungen konkretisiert und die Architektur festgelegt. Ersteres wird durch die nahezu vollständige Ergänzung der Use-Cases und anderer Anforderungen erzielt, zweiteres durch das Erstellen eines Architektur-Prototypen, der im Laufe des Projekts weiterentwickelt wird und aus dem letztlich das System resultiert. Auf diesem gründet auch der Name des phasen-abschließenden Meilensteins *Lifefycle Architecture*.

2.2.3 Konstruktion (engl. Construction)

In der Konstruktionsphase liegt das Hauptaugenmerk auf dem Entwickeln und Testen der einzelnen Komponenten. Das System soll in dieser Phase vervollständigt werden, sodass es mit Beenden der Phase durch den Meilenstein *Initial Operational Capability* als fertiges, auslieferbares Produkt vorliegt.

2.2.4 Übergang (engl. Transition)

Die Übergangsphase umfasst im Wesentlichen die Übergabe des Produkt an den Endanwender bzw. Kunden. Darunter fallen zum Beispiel Beta-Tests beim Kunden, Schulungen zum Umgang mit dem Produkt, die Fertigstellung von Begleitmaterial sowie die Datenkonvertierung aus Altsystemen.

2.3 Statische Struktur

Nachdem die zeitliche Struktur RUPs beschrieben wurde, wird nun die funktionale betrachtet. RUP umfasst neun *Dizplinen* (ehemals Workflows), welche in grundlegende und unterstützende Disziplinen unterteilt werden. Zur einheitlichen Prozessbeschreibung wurde für RUP ein Metamodell entwickelt (später bei der *OMG* als *SPEM* etabliert), welches 2005 durch die *Unified Method Architecture (UMA)* abgelöst wurde. Eine der wesentlichen Neuerungen, war die Trennung von Methodeninhalten und ihrem zeitlichen Einsatz in konkreten Prozessabläufen. Im Rahmen dieser Bachelorarbeit, sollen jedoch nur die Methodeninhalte eine Rolle spielen. [EM07]

2.3.1 Methodeninhalte

Hier sind drei wesentliche Konzepte zu nennen: *Rollen* (engl. role, ehemals Worker), *Aufgaben* (engl. task, ehemals Aktivitäten) und *Arbeitsergebnisse* (engl. work products, ehemals Artefakte).
Eine *Rolle* ordnet einer oder mehreren Personen bestimmte Verantwortlichkeiten im Projekt zu. Dabei kann eine Person auch mehrere Rollen haben, wie z.B. Designer und Designgutachter. Vereinfacht gesagt beantworten Rollen die Frage nach dem ,,Wer?" in einem Prozess.
Jeder Rolle sind bestimmte *Aufgaben* zugewiesen. Diese werden üblicherweise in mehrere Schritte unterteilt und haben jeweils ein eindeutiges Ziel, dessen Erfüllung durch *Arbeitsergebnisse* repräsentiert wird.
Arbeitsergebnisse dienen als Grundlage für Aufgaben und werden unterteilt in Artefakte, Liefergegenstände und Resultate. Artefakte (z.B. Modelle, Quellcode, Dokumente) unterscheiden sich dadurch von Resultaten (z.B. installierter Server), dass sie für weitere Aufgaben verwendbar sind. Liefergegenstände grenzen sich von den anderen beiden ab, indem sie ausschließlich für die Auslieferung an einen Stakeholder bestimmt sind.

2.3.2 Disziplinen

Disziplinen sind letztlich eine Menge zusammengehöriger Rollen, Aufgaben und Arbeitsergebnisse. Allerdings ist für ihre praktische Umsetzung soviel Wissen nötig, dass je Disziplin mindestens ein eigenes Buch erforderlich ist. Deshalb werden die grundlegenden und anschließend die unterstützenden Disziplinen an dieser Stelle auch nur kurz umrissen, indem jeweils bloß der wesentliche Zweck beschrieben wird. [EM07]

Geschäftsprozessmodellierung (engl. Business Modeling)

Der Kern der ersten grundlegenden Disziplin besteht - wie der Name schon vermuten lässt - darin, den Geschäftsprozess, welcher vom zu entwickelnden System letztlich automatisiert werden soll, für alle Stakeholder verständlich darzustellen. Das heißt die fachlichen Abläufe eines Geschäftsprozesses werden visuell festgelegt, damit diese verstanden und als Basis für die Anforderungsanalyse verwendet werden können.

Anforderungen (engl. Requirements)

Diese Disziplin befasst sich mit dem *Anforderungsmanagement*, welches bereits bei den *Best Practices* erläutert wurde. Kurz gesagt, geht es hier nicht darum, einmalig Anforderungen aufzustellen, sondern vielmehr um das kontinuierliche Berücksichtigen von Änderungen an den Anforderungen während des gesamten Projektverlaufs.

Analyse & Design (engl. Analysis & Design)

Eines der beiden wesentliche Ziele dieser Disziplin besteht darin, die Umsetzung der fachlichen Anforderungen auf einer technischen Plattform graphisch zu modellieren (mithilfe der UML). Das heißt die einzelnen Anforderungen müssen in ein technisches Design überführt werden, welches auf einer Architektur basiert, deren Erarbeitung das zweite wichtige Ziel dieser Disziplin darstellt.

Implementierung (engl. Implementation)

Der Zweck der Implementierung besteht natürlich darin, am Ende ein lauffähiges System zu erhalten. Das bedeutet, alle Komponenten müssen fertiggestellt und nach Entwickerlertests in das System integriert werden.

Test

In dieser Disziplin wird das ausführbare System auf die Erfüllung seiner zugrunde liegenden Anforderungen geprüft und diese damit bewiesen. Ziel ist hier nicht das Verhindern bzw. Lösen von auftretenden Problemen oder Fehlern, sondern das Finden möglichst vieler. Die wichtigsten Tests sind dabei die auf Funktionalität, Performanz und Stabilität.

Verteilung (engl. Deployment)

Die letzte grundlegende Disziplin beschäftigt sich mit der reibungslosen Auslieferung des Produkts an den Kunden bzw. Benutzer. Da mit dieser hohe Produktions- und Verteilungskosten verbunden sind, liegt der Fokus nicht nur auf dem Produkt-Release, sondern auch auf dem jeweils zuvor erfolgtem Beta-Test[4].

Konfigurations- & Änderungsmanagement (engl. Configuration & Change Management)

Das Konfigurations- & Änderungsmanagement umfasst zum einen das Verwalten von Arbeitsergebnissen innerhalb eines Teams sowie das Kontrollieren von Änderungen an Arbeitsergebnissen. Damit stellt diese unterstützende Disziplin eine Art Daten- und Änderungsverwaltung für die anderen Disziplinen dar, welche so effizienter ausgeführt werden können.

Projektmanagement (engl. Project Management)

Der wesentliche Zweck des Projektmanagements besteht im Allgemeinen darin, ein Projekt, basierend auf gegebenen bzw. verhandelbaren Bedingungen und Ressourcen zu planen sowie für die Einhaltung dieser Planung zu sorgen (siehe ,,Projekt Schaukel" aus der Einleitung als negatives Beispiel). Um im Sinne der iterativen Vorgehensweise von RUP sicher planen zu können, kommt dem Risikomanagement hier eine besondere Rolle zu.

Entwicklungsumgebung (engl. Environment)

Diese Disziplin stellt im Grunde genommen die Infrastruktur für ein Projekt bereit, das heißt es wird eine Entwicklungsumgebung mit benötigten Tools und Prozessen erschaffen. Des weiteren werden Richtlinien für einzelne Bereiche erstellt, die von den Stakeholdern als Hilfestellung wahrgenommen werden sollen.

[4] ,,Ein Beta-Test ist ein Systemtest, welcher von einer ausgewählten Gruppe von Kunden vor dem offiziellen Release einer Applikation [...] angewendet wird. Während des Beta-Tests zeichnen die Kunden ihre Erfahrungen und Probleme auf und berichten diese an die Software-Entwickler." [Haa04]

3 Beschreibung der Methodik

Aufgrund der Anpassbarkeit von RUP, eignet sich dieser auch für kleine Projekte, da sein Umfang wesentlich reduziert werden kann. So ist es möglich, eine Spezifizierung von RUP zu erstellen, die zum Beispiel lediglich auf eine Disziplin ausgelegt ist.
Das Ziel dieser Arbeit ist die ,,Konzeption und Implementierung eines graphischen Editors für verschiedene Notationen im Softwareentwicklungsprozess". Solche Notationen sind zum Beispiel die Artefakte von RUP-Disziplinen. Da jedoch ein vollständiger RUP zu umfangreich für die Demonstration der Qualitäten eines eigens entworfenen graphischen Editors wäre, wird in diesem Kapitel ein angepasster RUP vorgestellt, der lediglich die Disziplin *Analyse und Design* umfasst.
Diese beinhaltet als wesentliche Artefakte das *Designmodell* sowie das *Software-Architektur-Dokument*. Letzteres enthält die Dokumentation der Entwurfsentscheidungen, welche in Form von verschiedenen UML-Diagrammen im Designmodell festgehalten werden. Jenes kann auf ,,eine[r] grobe[n] Skizze des Systems", einem *Analysemodell*, basieren, welches sich, wie das Designmodell selbst, aus Klassen-, Interaktions- und Zustandsdiagrammen zusammensetzen kann. [Kru99] Diese Artefakte werden ebenfalls angepasst, um letztlich eine Methodik zu formen, die relativ klar und einfach strukturiert ist, jedoch ein RUP bleibt und damit auch Charakteristika, wie iteratives Vorgehen aufweist.
Wie diese Methodik im Einzelnen aussieht, wird im Abschnitt *Anforderungsanalyse* erläutert. Dort werden auch die Möglichkeiten diskutiert, die ein graphischer Editor bieten kann, um die Anwendung solch einer Methodik sinnvoll zu unterstützen.
Der Unterschied zu bestehenden Tools, wie dem *RSM* oder *RSA* von IBM, soll sich vor allem dadurch auszeichnen, dass der zu entwerfende graphische Editor speziell für die, in diesem Kapitel beschriebene Methodik konzipiert wird. So kann er nicht nur allgemeine, sondern spezifische, ,,sinnvolle" Unterstützungen leisten. Um dies

relativ objektiv einschätzen zu können, wird dabei Bezug auf allgemeingültige Kriterien der Software-Ergonomie bzw. der Usability (deut. Benutzerfreundlichkeit) genommen.
Deshalb werden zunächst generelle Konzepte der Usability vorgestellt und beispielhaft anhand eines beliebigen graphischen Editors demonstriert.

3.1 Usability-Konzepte

> „Gutes Oberflächendesign wird nicht vorrangig durch Styleguides und Ästhetik geprägt (obwohl diese Aspekte durchaus eine Rolle spielen), sondern durch die Grundsätze des Informationsdesigns wie Erkennbarkeit, Lesbarkeit, Unterscheidbarkeit, Konsistenz, Prägnanz, Verständlichkeit [...]." [Akk09]

Diese eben genannten Prinzipien sind in der Norm EN ISO 9241 wiederzufinden, welche seit 2006 in Deutschland auch unter dem deutschen Titel *Ergonomie der Mensch-System-Interaktion* bekannt ist. Sie gilt als ein internationaler Standard mit dem Ziel, nicht nur gesundheitliche Schäden bei der Bildschirmarbeit zu vermeiden, sondern auch dem Benutzer die Ausführung seiner Ausgaben zu erleichtern. So legt sie auch in ihrem elften Teil fest, dass die Gebrauchstauglichkeit einer Software von ihrem Nutzungskontext abhängt. Deshalb werden drei Leitkriterien zur Bestimmung dieser festgelegt:

- Effektivität zur Lösung einer Aufgabe,
- Effizienz der Handhabung des Systems,
- Zufriedenheit der Nutzer einer Software.

Neben den oben zitierten Prinzipien des Informationsdesigns aus Teil 12 der Norm EN ISO 9241, wurden in Teil 110 noch weitere Grundsätze aufgestellt, wie Schnittstellen, insbesondere einzelne Arbeitsschritte, sogenannte Dialoge, zwischen Benutzer und Software zu gestalten und bewerten sind. Um diese besser zu verstehen, folgt auf eine kurze Erklärung jeweils ein Beispiel aus der Domäne der graphischen Editoren:

- **Aufgabenangemessenheit:**
  Ein Dialog ist *aufgabenangemessen*, wenn er die Erledigung der Arbeitsaufgaben des Benutzers unterstützt, ohne ihn durch Eigenschaften des Dialogsystems unnötig zu belasten. [FK04]

  Beispiel:
  Stellen wir uns einen graphischen Editor vor, der unter anderem eine Liste von Text-Elementen darstellt. Das vorläufige Ziel soll die Auswahl eines Text-Elements sein. Dies ist mit einem gewissen Suchaufwand verbunden, abgängig von der Menge Text-Elemente. Um Übersicht zu wahren und den Suchaufwand zu minimieren, kann durch das Bereitstellen einer Eingabezeile und einer damit verbundenen Funktion, die Menge der angezeigten Text-Elemente reduziert werden. Durch den Abgleich der Text-Elemente mit der getätigten Eingabe sollen nur noch Übereinstimmungen aufgeführt werden. So eine, auch als Filter bekannte Funktion, trägt dazu bei, den Editor für den Nutzer *aufgabenangemessener* zu gestalten.

- **Selbsterklärungsfähigkeit:**
  Ein Dialog ist *selbsterklärungsfähig*, wenn dem Benutzer auf Verlangen Einsatzzweck sowie Leistungsumfang des Dialogsystems erläutert werden können und wenn jeder einzelne Dialogschritt unmittelbar verständlich ist. [FK04]

  Beispiel:
  Ein graphischer Editor bietet meist eine Palette mit verschiedenen Icons an. Nicht nur die Wahl passender graphischer Symbole trägt zu dessen *Selbsterklärungsfähigkeit* bei, sondern auch das Hinzufügen von geeigneten Tooltips, also kleinen Beschreibungstexten als Schnell-Hilfe, die der Benutzer anfordern kann.

- **Steuerbarkeit:**
  Ein Dialog ist *steuerbar*, wenn der Benutzer die Geschwindigkeit des Ablaufs sowie die Auswahl und Reihenfolge von Arbeitsmitteln oder Art und Umfang von Ein- und Ausgaben beeinflussen kann. [FK04]

Beispiel:
In manchen Fällen kommt es vor, dass graphische Editoren lediglich die Möglichkeit bieten, jeweils nur ein Objekt ihrer Oberfläche auszuwählen. Dies wird dem Nutzer spätestens dann bewusst, wenn er Aktionen für mehrere Objekte gleichzeitig ausführen, also zum Beispiel eine ganze Gruppe von Objekten löschen will. Durch das Hinzufügen der Möglichkeit einer Auswahl von mehreren Objekten, also einer Gruppenauswahl wird der Editor *steuerbarer*.
Neben Tastaturkürzeln, als alternative Zugriffsmöglichkeit zu Schaltflächen oder Kontextmenü-Einträgen, sind für die *Steuerbarkeit* vor allem die Mechanismen von Bedeutung, die den Arbeitsprozess an sich betreffen. Dazu gehört unter anderem die Möglichkeit, sich frei in der Historie zu bewegen, also Arbeitsschritte rückgängig machen zu können oder wiederherzustellen, aber auch die Option, den aktuellen Zustand abzuspeichern und später laden zu können.

- **Erwartungskonformität:**
  Ein Dialog ist *erwartungskonform*, wenn er den Erwartungen der Benutzer entspricht, die sie aus Erfahrungen mit bisherigen Arbeitsabläufen oder aus der Benutzerschulung mitbringen, sowie den Erfahrungen, die sich während der Benutzung des Dialogsystems und im Umgang mit dem Benutzerhandbuch bilden. [FK04]

  Beispiel:
  Graphische Editoren besitzen fast immer eine Menüleiste als Bestandteil ihrer Oberfläche. Diese ist *erwartungskonform* gestaltet, wenn sich ihr Aufbau nach dem etablierten Maßstab bestehender Programme richtet.
  Ein anderes Beispiel ist die graphische Darstellung der einzelnen Elemente. Diese sollte sich nach der ihr zugrunde liegenden Vorgabe richten. Ein UML-Editor zum Beispiel sollte also die UML-Notation auch möglichst getreu abbilden.

- **Fehlerrobustheit:**
  Ein Dialog ist *fehlerrobust*, wenn trotz erkennbar fehlerhafter Eingaben das beabsichtigte Arbeitsergebnis mit minimalen oder ohne Korrekturaufwand erreicht wird. Dazu müssen dem Benutzer die Fehler zum Zwecke der Behebung verständlich gemacht werden. [FK04]

  Beispiel:
  Bei graphischen Editoren stellt zum Beispiel die „Rückgängig machen"-Funktion eine Möglichkeit dar, ohne großen Aufwand in einen vorherigen Zustand zurückzukehren. Das heißt soviel wie, nachdem der Benutzer zum Beispiel aus seiner Sicht einen Fehler bei der Modellierung gemacht hat, kann er diesen ohne weiteres mit einer Aktion rückgängig machen und weiterarbeiten. Außerdem wird der Editor *fehlerrobuster*, insofern er toleranter mit textuellen Eingaben umgeht, indem er zum Beispiel mehrere Formate für Daten akzeptiert und somit weniger limitierend auf den Benutzer wirkt.

- **Individualisierbarkeit:**
  Ein Dialog ist *individualisierbar*, wenn der Benutzer Anpassungen an seine Bedürfnisse vornehmen kann. [FK04]

  Beispiel:
  Ein einfaches Beispiel für *Individualisierbarkeit* bei graphischen Editoren ist das Anbieten der Wahl der im Editor verwendeten Sprache. Ein anderes wäre das Bereitstellen einer Zoom-, also einer Vergrößerungs- und Verkleinerungsfunktion, welche es dem Benutzer ermöglicht, den Grad der Darstellung der angezeigten Elemente *individuell* anzupassen.

[Akk09]
Neben diesen Grundsätzen ist allerdings auch die Benutzerdokumentation ein wichtiger Faktor für die Gebrauchstauglichkeit. Sie erläutert den Einsatzzweck der Software sowie deren einzelne Funktion und sollte unter anderem folgende Komponenten umfassen: *Nutzungskonzept*, *Produktbeschreibung*, *Handbuch*, *Lerntexte und -programme*, *(Online-)Hilfetexte*, und *Fehlermeldungen*.

3.2 Anforderungsanalyse

Dieser Abschnitt soll dem Verständnis der Anforderungen dienen, die dem zu entwerfenden graphischen Editor zugrunde liegen. Dafür wird zum einen die zu Beginn dieses Kapitels eingeführte Methodik beschrieben. Sie besteht im Wesentlichen aus zwei großen Teilen: *Analyse und Design*. Zur Analyse zählen die Artefakte *Klassenmodell des Gegenstandsbereichs*, *Use-Case-Diagramm*, *Sequenzdiagramme*, *Aktivitätsdiagramme* sowie *Systemklassenmodell*. Diese werden als Grundlage für die Artefakte des Designs benutzt bzw. zum Teil nur verfeinert. Zum Design zählen die Artefakte *Kommunikationsdiagramme*, *Entwurfsklassenmodell* sowie *Implementationsklassenmodell*.
Neben der Beschreibung der Methodik, sind für die Umsetzung des graphischen Editors letztlich Features von Bedeutung, die dessen Einsatz überhaupt erst rechtfertigen. Möglichkeiten für solche werden ebenfalls in diesem Abschnitt diskutiert, mit Hinblick auf bewährte Prinzipien der Usability. Besonders zwei Konzepte spielen hierbei eine wichtige Rolle: *Synchronisation* und *Konsistenzchecks*.
Da die angepasste Methodik ein angepasster RUP ist, muss vor allem das iterative Vorgehen sinnvoll unterstützt werden, was durch die *Synchronisation* bestimmter Artefakte bzw. derer Elemente erzielt werden kann. Dabei müssen allerdings die Auswirkungen genau bedacht werden, da sonst zum Beispiel, entgegen der Erwartung des Benutzers, Informationen verloren gehen können.
Auf der anderen Seite können mittels *Konsistenzchecks*, also durch das Überprüfen bestimmter, festgelegter Bedingungen, Fehlerquellen vermieden oder Fehler kenntlich gemacht werden.
Zuerst jedoch werden einige für alle Artefakte gültige Prinzipien festgehalten, die bei der Gestaltung des graphischen Editors berücksichtigt werden sollen.

3.2.1 Allgemeine Umsetzungsdiskussion

Generell gilt, dass die graphische Darstellung der Elemente von Artefakten, sich, aus Gründen der Erwartungskonformität, möglichst am jeweiligen Original der *UML* orientiert, um lange Einarbeitung und Verwirrung beim Benutzer zu vermeiden.
Weiterhin kann ein höherer Grad an Selbsterklärungsfähigkeit erreicht werden, indem Elemente, wie z.B. Buttons oder die Einträge einer Palette, mit passenden

Icons, also kleinen Grafiken versehen werden.
Außerdem sollte, neben der optimalen Unterstützung des Benutzers durch bestimmte Features, auch „[...] ein flüssiger und störungsfreier Handlungsablauf [...]" angestrebt werden, um dazu beizutragen, das sogenannte *Flow-Erleben* zu ermöglichen: „[...] Effektivität und Effizienz in perfekter Form. Usability pur [...]". [Vö02] Angewandt auf einen graphischen Editor würde dies etwa bedeuten, dass vermieden werden sollte, den User nach jeder Interaktion mit einem Dialog zu konfrontieren und ihn somit von seiner eigentlich Arbeit abzulenken.

3.2.2 Klassenmodell des Gegenstandsbereichs (KMG)

Dieses der Analysephase angehörige Artefakt nutzt die Notation des UML-Klassenmodells und modelliert den Gegenstandsbereich, das heißt, Gegenstände, Eigenschaften und Beziehungen des Problems erfassen. Außerdem bildet es die Grundlage für das im späteren Verlauf zu entwickelnde Systemklassenmodell.
Es umfasst zum einen UML-Klassen, jeweils mit einem Namen und Attributen, sowie UML-Beziehungen, also Assoziationen, Kompositionen, Aggregationen und Generalisierungen/Spezialisierungen, die jeweils unterschiedlich mit einem Namen und Multiplizitäten (Kardinalitäten) bestückt sind. Im Gegensatz zu RUP, werden bei unserer vereinfachten Methodik die Elemente Rollen, Methoden sowie Assoziationsklassen vernachlässigt (Abbildung 3.1).

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Eindeutige Klassennamen
- Beziehungen nur zwischen zwei verschiedenen Klassen
- Nur eine Beziehung zwischen zwei Klassen
- Kein Generalisierungs-/Spezialisierungs-Zyklus
- Eindeutige Beziehungsnamen
- Eindeutige Attributnamen innerhalb einer Klasse

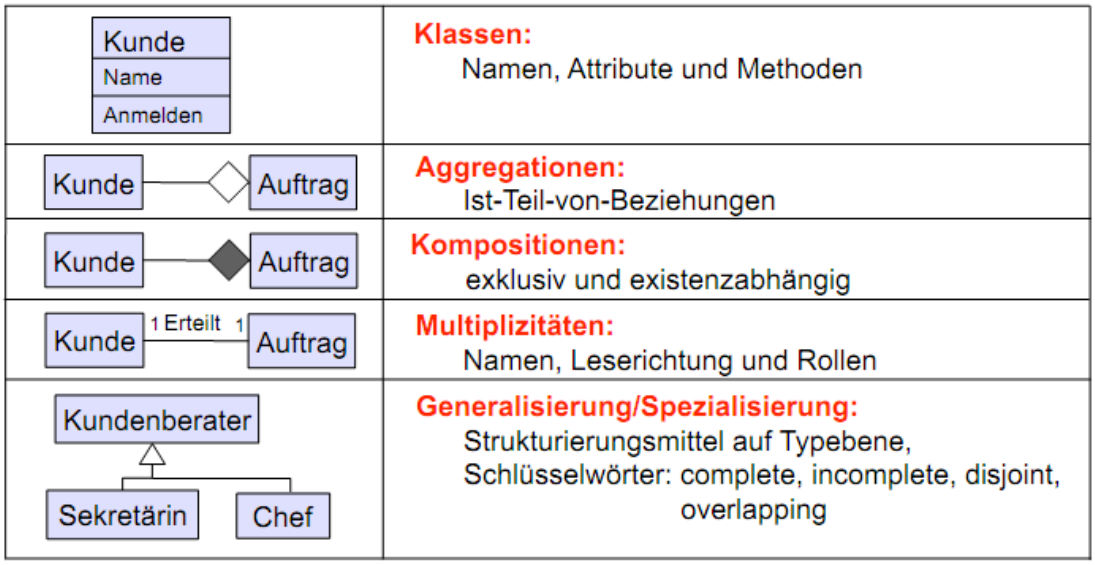

Abbildung 3.1: Notation für KMG [DISJ09]

Umsetzungsdiskussion

Bezüglich der Konsistenzen ist es sinnvoll, eindeutige Klassennamen zu fordern, um den korrekten Aufbau eines KMGs zu unterstützen und inhaltliche Verwirrungen vorzubeugen. Eindeutige Beziehungs- und Attributsnamen sind ebenfalls wünschenswert, da sie den Benutzer im korrekten Anwenden der Methodik unterstützen, allerdings ist hier abzuwägen, ob häufige Fehlermeldungen nicht das flüssige Arbeiten und damit auch die Zufriedenheit des Benutzers belasten. Eine Alternative ist die graphische Hervorhebung doppelter Namen, im Sinne der Fehlerrobustheit. Die farbliche Kennzeichnung macht dem Benutzer die Existenz eines Fehlers deutlich ohne ihn sofort zur Korrektur zu zwingen.
Wichtig ist die Realisierung der beiden Bedingungen, dass Beziehungen immer zwischen zwei verschiedenen Klassen verlaufen müssen und zwischen zwei Klassen jeweils nur eine Beziehung existieren existieren darf. Nur so können größere Fehlerquellen und Missverständnisse bezüglich der Methodik von vornherein ausgeschlossen werden.
Ein zusätzlich wünschenswertes Feature ist das Verbieten von Generalisierungs-/Spezialisierungs-Zyklen, das gerade aufgrund seines geringen Implementierungsaufwands umgesetzt werden sollte.

3.2.3 Use-Case-Diagramm (UCD)

Das Use-Case-Diagramm, oder zu deutsch Anwendungsfalldiagramm, gehört ebenfalls zur Analyse und bedient sich der Syntax des gleichnamigen Pendants der UML. Sogenannte Akteure, welche als Klasse im KMG auftreten, repräsentieren verschiedene Benutzerrollen und interagieren direkt mit dem System. Das mögliche Verhalten des Systems, wird durch verschiedene Use-Cases (Anwendungsfälle) dargestellt, die wiederum jeweils eine Menge von Szenarien umfassen. Mithilfe dieser Mittel soll eine erste grobe Systemgrenze festgelegt werden.
Die Notation umfasst also neben Akteuren, die außerhalb des Systems stehen, noch Use-Cases, welche sich innerhalb des Systems befinden. Diese können mithilfe von Kommunikationen Akteuren zugeordnet werden. Außerdem existieren für Use-Cases die Erweiterungs- (‹‹extend››), die Einbindungs- (‹‹include››) und die Generalisierungs- / Spezialisierungsbeziehung (Abbildung 3.2), um Szenarien abbilden zu können.

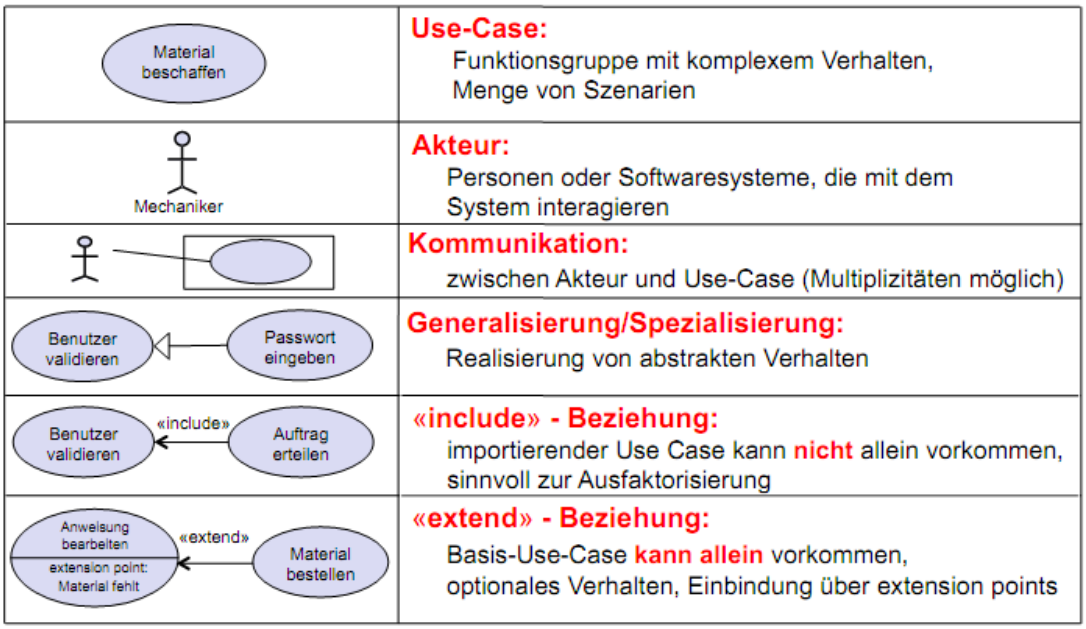

Abbildung 3.2: Notation für UCD [DISJ09]

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Akteure kommen als Klassen im KMG vor
- Keine Erweiterungs-, Einbindungs- oder Generalisierungszyklen
- Keine sukzessiven Erweiterungsbeziehungen (wenn UC1 $-\text{extend}\rightarrow$ UC2, dann darf kein UC2 $-\text{extend}\rightarrow$ UCX mehr möglich sein)

Umsetzungsdiskussion

Aufgrund der iterativen und inkrementellen Vorgehensweise, wie sie für RUP typisch ist [FK04], basieren auch Artefakte auf vorangegangenen Modellen. Das heißt unter anderem, dass das Klassenmodell des Gegenstandsbereiches die Menge der verfügbaren Klassen liefert, die als Akteur in einem Use-Case-Diagramm vorkommen können. Genau hier wäre es sinnvoll, den Punkt der Aufgabenangemessenheit ins Auge zu fassen, um den Nutzer beim Lösen seiner Aufgaben möglichst effizient zu unterstützen.
Eine Möglichkeit, diese Konsistenzbedingung umzusetzen, ist die Synchronisation des Klassenmodells mit dem Use-Case-Diagramm. Beim Erstellen von Akteuren im UCD, kann direkt der Bezug auf eine Klasse des KMGs gefordert werden, mit der er zukünftig synchronisiert werden soll. So wird die Änderung eines ,,Partners" immer auch die Änderung des Anderen nach sich ziehen. Wenn nun jedoch einer der beiden gelöscht wird, ist das übrig bleibende Element von keinem Nutzen mehr für den restlichen Entwicklungsprozess, da die betroffenen Artefakte nicht mehr auf einander abgestimmt, also untereinander inkonsistent sind. Somit muss auch ein Vorgang implementiert werden, der das Verbinden von solchen Überbleibseln ermöglicht und die Konsistenz zwischen den Artefakten wiederherstellt. Im Use-Case-Diagramm ist das die Bereitstellung einer Funktion, UCD-Akteure und KMG-Klassen ohne Partner miteinander neu zu verknüpfen. Dabei sollte der aktuelle Synchronisationsstatus eines Elements anhand seiner graphischen Darstellung erkannt werden können, zum Beispiel mithilfe von Icons. Mit diesen Eigenschaften der Synchronisation wird auch dem Prinzip der Fehlerrobustheit folge geleistet, da der Nutzer nun leicht Inkonsistenzen erkennen und korrigieren kann.

Wie beim KMG, können auch beim Use-Case-Diagramm von vornherein zyklische Erweiterungs-, Einbindungs- oder Generalisierungskonstellationen unterbunden werden, was sich als Feature besonders anbietet, da es bereits für das KMG implementiert werden soll und somit lediglich angepasst werden muss.
Das Verbieten von sukzessiven, also aufeinanderfolgenden Erweiterungsbeziehungen ist auch relativ simpel zu implementieren und erzwingt praktisch ihren sparsamen Einsatz. Das ist der Vereinfachung unserer Methodik nur zuträglich und sollte somit unbedingt seinen Einzug als Feature finden.

3.2.4 Sequenzdiagramme (SDs)

Sequenzdiagramme (SDs) tragen, wie die beiden vorhergehenden Artefakte der Analysephase, ebenfalls zur Präzisierung der Anforderungen bei. Genauer gesagt, modellieren sie nicht sämtliche, aber die wichtigsten Szenarien der vorher definierten Use-Cases. Das geschieht in Form von sequentiellen Abläufen von Nachrichten zwischen Instanzen der im UCD festgelegten Akteure und des Systems. Nachrichten werden dabei unterschieden in *Systemoperationen* (zur Systeminstanz gerichtete Nachrichten) und *Systemereignisse* (zu Akteurinstanzen gerichtete Nachrichten). Diese können spezifiziert werden, indem sie angelegten Bedingungen zugeordnet werden, den sogenannten Alternativen (Abbildung 3.3).

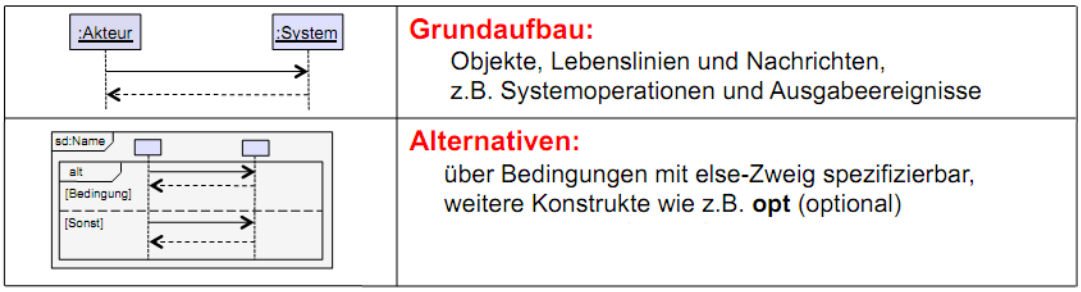

Abbildung 3.3: Notation für SD [DISJ09]

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Akteure entsprechen denen des Use-Case-Diagramms

- Nachrichten verlaufen horizontal (auf einer Ebene)
- Eindeutige Bezeichnungen für Nachrichten
- Nur eine Systemoperation pro SD

Umsetzungsdiskussion

Die Synchronisation von Sequenzdiagrammem mit dem Use-Case-Diagramm kann im Wesentlichen auf zwei Arten geschehen, die beide auf die Einhaltung der ersten Konsistenzbedingung abzielen.
Eine Möglichkeit ist, beim Erstellen eines Akteurs im SD, die Verknüpfung mit einem im UCD existierenden Akteur zu fordern. Da diese Form der Synchronisation bereits beim UCD angewandt wurde, kann hier ein Vorteil aus der Wiederverwendbarkeit gezogen werden. Der Aufwand für die Implementierung fällt somit relativ gering aus, da lediglich einige Anpassungen vorgenommen werden müssen.
Eine wesentlich komplexere Variante ist das Zuordnen eines SDs zu einem Use Case aus dem UCD. Indem im UCD ein Use Case ausgewählt wird, für den ein SD erstellt werden soll, kann eine initiale Zuordnung vorgenommen werden. Der Vorteil dieser Variante liegt darin, dass Akteure im SD automatisch anhand der Informationen aus dem UCD generiert werden können, der Benutzer also keine Akteure anlegen kann, die im UCD nicht existieren. Wenn aber nach dem Fertigstellen der SDs, das UCD verändert wird, müssen diverse Fälle beachtet werden. Einerseits müssen beim Verschieben von Kommunikationen im UCD, Akteure im SD entweder zusätzlich angelegt oder von ihrem Partner im UCD getrennt werden, wenn ein Akteur nicht mehr Teil eines Use Cases ist. Analog gilt dies natürlich auch für das nachträgliche Erstellen bzw. Löschen von Kommunikationen im UCD. Somit muss ebenfalls eine Option zum Neuverknüpfen von Akteuren des SDs mit möglichen UCD-Akteuren angeboten werden, sowie eine Lösung gefunden werden, wie mit dem Löschen von Use Cases im UCD, in Bezug auf korrespondierende SDs umgegangen werden soll.
Eindeutige Bezeichnungen für Nachrichten zu fordern ist sicherlich sinnvoll, in Hinblick auf die vollständige Korrektheit eines Diagramms, aber auch nicht unbedingt notwendig, da aufgrund der Semantik eines Sequenzdiagramms gleichnamige Nachrichten in der Regel eher selten der Fall sind, weshalb solch ein Feature als optional eingestuft werden kann.

Hingegen ist die Forderung nach der Exklusivität von Systemoperationen als ein technisch leicht umsetzbares Feature zu betrachten, das wiederum eine Fehlerquelle mehr beim Anwenden der Methodik ausschließen kann.

3.2.5 Aktivitätsdiagramme (ADs)

Aktivitätsdiagramme beschreiben das Gesamtverhalten des Systems, indem sie alle Szenarien eines Use-Cases modellieren, die von einem Akteur angestoßen werden. Im Wesentlichen bestehen sie aus Aktionen, die als Zustände zu betrachten sind und einem Kontroll- bzw. Datenfluss, also gerichteten Beziehungen zwischen diesen, die Zustandsübergänge ermöglichen. Aktionen können dabei als Ausführungsschritte eines Algorithmus, Geschäftsprozesses oder Systemablaufs verstanden werden, welche in eine Kette von untergeordneten Aktionen aufgegliedert werden können. So können Aktionen auch im Diagramm verschachtelt werden und somit die Grundlage für ein weiter spezifizierendes Unterdiagramm bilden.
Letztlich bilden Aktivitätsdiagramme die Grundlage für die Implementierung von graphischen Oberflächen (GUIs), indem sie mehrere Szenarien eines Use-Cases in Zusammenhang bringen, also die einzelnen Abläufe thematisch gruppieren.
Ihre Syntax entspricht den Aktivitätsdiagrammen der UML, umfasst neben Aktionen und dem Kontroll-/Datenfluss noch Start- und Endknoten sowie Zusammenführungs-(engl. *Merge*) und Entscheidungsknoten(engl. *Decision*) (Abbildung 3.4).

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- SD $\subseteq$ AD
- Jeweils genau ein Start- und Endknoten pro AD
- Nur von Entscheidungsknoten ausgehende Kanten besitzen Text
- eingeschränkte Beziehungsmöglichkeiten, durch RUP vorgegeben

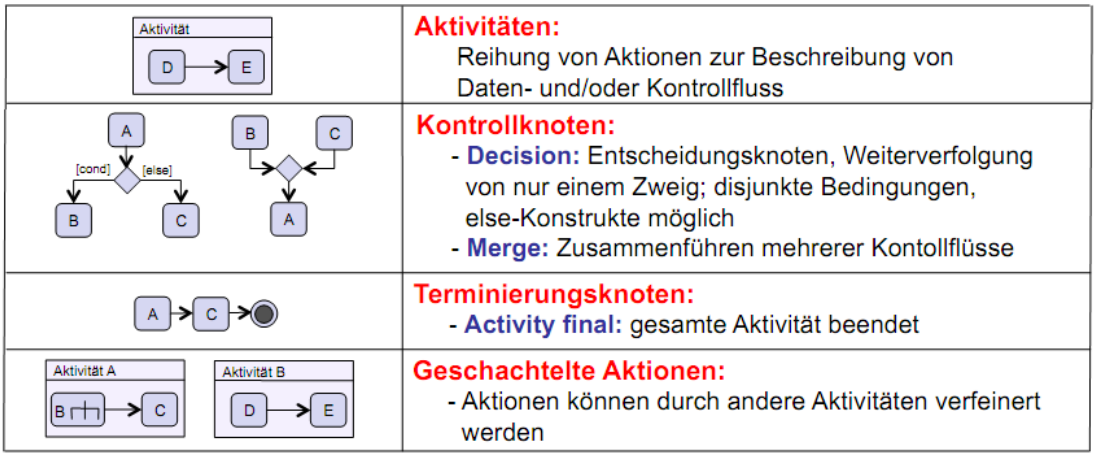

Abbildung 3.4: Notation für AD [DISJ09]

Umsetzungsdiskussion

Eine relativ einfache Möglichkeit, die erste Bedingung zu erfüllen, ist das Use-Case-Diagramm als Ausgangspunkt zu betrachten. Nach dem Erstellen einer Kommunikation im UCD, soll für das entstandene Szenario ein AD angelegt werden, benannt nach dessen Akteur und Use-Case. Nachträgliche Änderungen im Use-Case-Diagramm sollen auch Auswirkungen auf betroffene ADs haben, was im Speziellen deren Existenz und Namensgebung betrifft. Um inkonsistente Zustände zu vermeiden, also zum einen die Existenz von ADs, die nicht im UCD als Szenario vorkommen und zum anderen die Umkehrung dessen, ist es hier nicht besonders sinnvoll, eine Entkopplung der Partner und Möglichkeiten zur Neuverknüpfung anzubieten. Stattdessen ist es für den Anwender mit weniger Aufwand verbunden, dies hier in einem Schritt tun zu können, indem er einfach die Kommunikationen im UCD verschiebt, also jeweils die Endpunkte der entsprechenden Beziehung zwischen Akteur und Use-Case umsetzt.
Auch die Möglichkeit von Konsistenzchecks bietet sich hier an. Da das AD zwar, wie gerade beschrieben, Synchronisation unterstützt, aber keine Synchronisationselemente besitzt, müssen auch keine Sonderfälle durch Einwirkungen von anderen Artefakten beachtet werden. So können hier ungültige Aktionen, die der durch RUP vorgegebenen Methodik widersprechen, einfach durch situationsbezogenes Einschränken der Gestaltungsfreiheit unterbunden werden. Solche Einschränkungen betreffen dann zum Beispiel die Art, Anzahl und Existenz ein- und

ausgehender Kanten eines Knoten, abhängig von den geforderten Konsistenzbedingungen.

3.2.6 Systemklassenmodell (SKM)

Dieses Artefakt ist das letzte der Analysephase. Im Wesentlichen wird hier eine konkrete Systemgrenze festgelegt, indem Klassen des KMGs kategorisiert und neu strukturiert werden. Das Ziel ist es, die Grundlage für ein Modell zu schaffen, aus dem später Code generiert werden kann.
Die Kategorien, nach denen die Klassen unterschieden werden, entstammen der 3-Schichten-Architektur, welche im Wesentlichen die Trennung zwischen Benutzeroberfläche, Fachkonzept und Datenhaltung besagt. [Bal10] Dabei werden Akteure nicht berücksichtigt, da sie nicht Teil des Systems sind. Sie werden als außerhalb des Systems stehender Teil dargestellt, der allerdings wie die anderen Klassen auf dem Klassenmodell des Gegenstandsbereichs beruht. Somit kann eine Klasse des SKMs nun entweder Akteur (engl. *Actor*), Übergangsklasse (engl. *Boundary*), Gegenstandsklasse (engl. *Entity*) oder Steuerungsklasse (engl. *Control*) sein. (Abbildung 3.5)
Übergangsklassen bilden die Schnittstellen zwischen Akteuren und dem System, d.h. sie stellen praktisch die Präsentationsschicht in Form von GUI-Komponenten dar. Gegenstandsklassen hingegen repräsentieren die Datenhaltungsschicht, sind also für die Speicherung von Informationen bzw. die persistente Datenhaltung zuständig. Steuerungsklassen sind das Bindeglied zwischen Datenhaltungs- und Präsentationsschicht. Da sie komplexe Funktionalitäten, also die Geschäftslogik und alle anderen Abläufe modellieren, wird ihre Gesamtheit auch als Logikschicht bezeichnet. [SBD+10]

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Hat Klassenmodell des Gegenstandsbereichs als Grundlage
- Akteure werden aus dem Use-Case-Diagramm abgeleitet
- Eingeschränkte Beziehungsmöglichkeiten zwischen verschiedenen Klassentypen

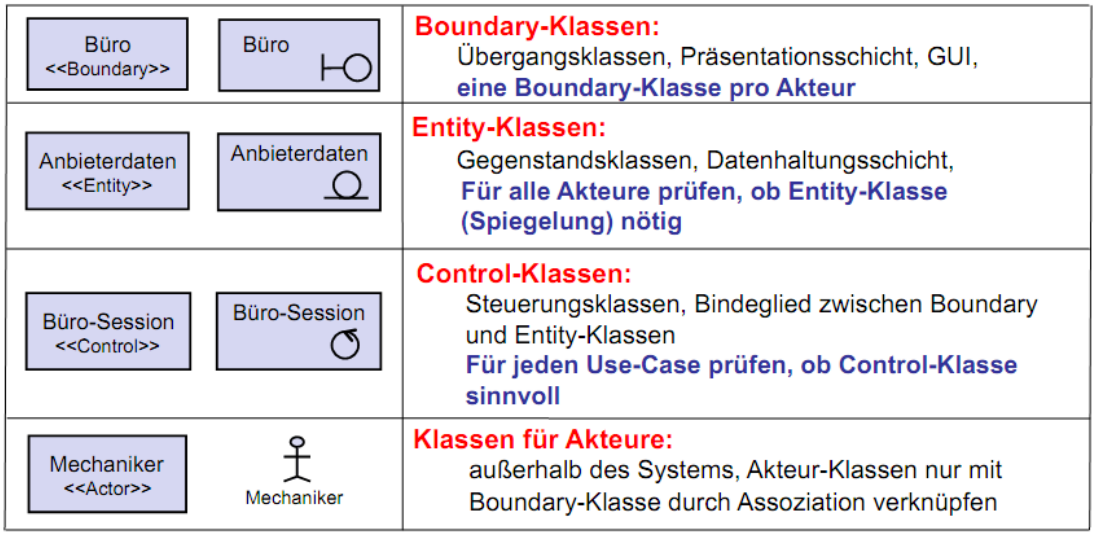

Abbildung 3.5: Notation für SKM [DISJ09]

Umsetzungsdiskussion

Anders als beim UCD, das lediglich seine Akteure mit einigen Klassen des KMGs synchronisiert, kann das SKM vorerst beim Erstellen des KMGs komplett mitgeneriert werden. Dabei sollen alle KMG-Klassen vorerst als Entity im Systemklassenmodell angelegt werden, es sei denn, die betreffende KMG-Klasse hat einen UCD-Akteur als Partner, dann muss sie im SKM natürlich auch als Akteur übernommen werden. Nun sind zwar die ersten beiden Punkte der Konsistenzbedingungen erfüllt, um allerdings diese Klassen weiterhin sinnvoll benutzen zu können, muss eine Möglichkeit geboten werden, sie in andere Klassentypen transformieren zu können, also zum Beispiel eine Entity in ein Control umzuwandeln. Durch dieses Feature wird dem User effizientes Arbeiten ermöglicht, da er zum einen die Diagramme nicht mehr per Hand übertragen muss und zum anderen einige nachträgliche Änderungen automatisch auch im Partner-Element vollzogen werden. Dabei ist allerdings wichtig zu unterscheiden, wo und wann Änderungen auch im Partnerelement übernommen werden sollen:

- **im KMG:**
  Alle hier vorgenommenen Änderungen werden auch im SKM übernommen, was die synchronisierten Elemente belangt, da das KMG schließlich die Grundlage für das SKM liefert.

- **im SKM:**
  Rückwirkende Änderungen (Auswirkungen von Änderungen im SKM auf das KMG) müssen sehr genau ausgewählt werden, da manche Spezifizierungen in vorherigen Modellen nicht notwendig sind. Dies ist zum Beispiel der Fall bei Multiplizitäten, welche im KMG noch weniger genau formuliert sein müssen und somit nicht rückwärtig (mit Auswirkung auf das KMG) synchronisiert werden sollen. Namensgebungen von Klassen und Beziehungen hingegen sollen, insofern diese synchronisiert sind, gleichermaßen verändert werden, um identische Elemente in den Artefakten überhaupt identifizieren zu können. Auch das Neuverankern von Beziehungen muss nun den Fall beinhalten, dass beim Verbinden mit einer neu angelegten, unsynchronisierten SKM-Klasse (Entity, Boundary, Control oder Akteur), die Beziehung von ihrem Partner im KMG getrennt wird und beide nun unsynchron agieren. Dies ist notwendig, wenn z.B. aufgrund der 3-Schichten-Architektur zusätzliche Klassen und Beziehungen angelegt werden müssen.

Zusätzlich kann mithilfe von Synchronisations-Status-Icons leicht deutlich gemacht werden, welche Elemente der Nutzer im SKM neu angelegt und welche er aus dem KMG übernommen hat, um einen höheren Grad an Übersichtlichkeit zu erzielen. Bei Beziehungen zum Beispiel, können solche Icons als farbige Pfeile dargestellt werden, die zusätzlich durch die Richtung des Pfeils, auch noch die Leserichtung der Beziehung angeben. Diese muss vom Nutzer änderbar sein und natürlich als Bestandteil von Beziehungen ggf. mitsynchronisiert werden, sowohl zum KMG als auch zum EKM, wo dieses Feature ebenfalls Einzug finden soll.
Aus der Verwendung der 3-Schichten-Architektur folgen die Bedingungen, dass Akteure nur mit Boundaries, Boundaries nur mit Controls oder Akteuren, Controls mit allen außer Akteuren und Entities nur mit anderen Entities oder Controls verbunden werden können. Als erster Vorschlag kommt dabei der bereits bei den Aktivitätsdiagrammen implementierte Vorgang infrage, bei dem einfach die Gestaltungsmöglichkeiten so eingeschränkt werden, dass das Anlegen ungültiger Beziehungen erst gar nicht möglich ist. Dies kollidiert beim SKM allerdings mit dem oben genannten Feature der Transformierbarkeit von Klassen, wenn zum Beispiel eine Entity in eine Boundary umgewandelt wird und bestehende Beziehungen nicht mehr die geforderten Bedingungen erfüllen. Sämtliche, der Semantik widersprechenden Beziehungen müssten gelöscht werden, was weder erwartungs-

konform wäre, noch den eingangs genannten Leitkriterien der *Norm für Ergonomie der Mensch-System-Interaktion* folgen würde.
Deshalb ist hier das Zulassen inkonsistenter Zustände, also z.B. das Existieren von nach der Methodik nicht erlaubten Beziehungen, von Vorteil, jedoch unter der Bedingung, dass solche Zustände graphisch hervorgehoben werden. So wird der Nutzer im Sinne der *Fehlerrobustheit* auf seine Fehler aufmerksam gemacht und kann diese bequem korrigieren, indem er zum Beispiel nicht erlaubte Beziehungen verlagert und so weiter benutzen kann, anstatt auf sie verzichten zu müssen.

3.2.7 Kommunikationsdiagramme (KDs)

Mit der Fertigstellung des Systemklassenmodells endet in der betrachteten, Methodik die Analyse und das Design beginnt. Hier wird beschrieben, *wie* das System die in den vorhergehenden Artefakten definierten Anforderungen erfüllen soll.
Das Design beginnt mit dem Erstellen von Kommunikationsdiagrammen für jede Systemoperation bzw. jedes Sequenzdiagramm, wobei jeweils ein KD die Realisierung einer Systemoperation auf Objektebene modelliert, indem es den Kommunikationsablauf zwischen den einzelnen Objekten beschreibt. Diese sind Instanzen von Klassen des SKMs, weshalb ihre Kommunikation ebenfalls durch dieses beschränkt ist. Das heißt, Nachrichten zwischen Objekten im KD können nur zwischen Objekten ausgetauscht werden, deren Klassen im SKM miteinander kommunizieren. Auch der Akteur, der die jeweilige Systemoperation aufruft wird abgebildet, ist jedoch wie schon zuvor als systemextern zu betrachten. Die Kommunikation zwischen Boundary und Control wird hier gar nicht abgebildet.
Kommunikationsdiagramme setzen sich prinzipiell aus einem Akteur, dessen *Controller* (empfängt Systemoperation) und einer Gruppe von *Collaborators*, also den Objekten zusammen. Diese können mithilfe von *Links* (Verbindungen) in Abhängigkeit gebracht werden. Links wiederum enthalten *Nachrichten*, die neben einem Pfeil zur Orientierung, einen Namen und eine, sich aus der Semantik des jeweiligen Diagramms ergebende, *Nummerierung* besitzen. Zusätzlich können Nachrichten mit Parametern, Bedingungen (engl. *Conditions*) und *Variablen* versehen sein. Objekte hingegen haben die Möglichkeit, *Collections* zu sein, also die Verwaltung für mehrere Elemente darzustellen, wie es unter anderem in Abbildung 3.6 gezeigt wird.

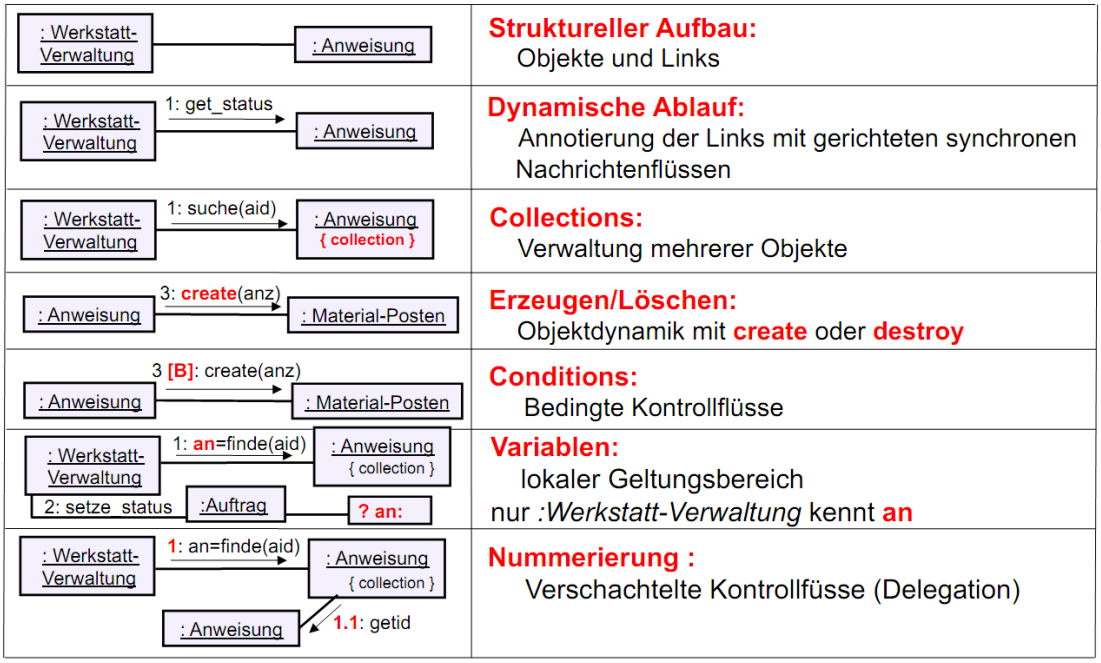

Abbildung 3.6: Notation für KD [DISJ09]

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- SKM ist Ausgangspunkt für Kommunikationsdiagramme, sowohl für Objekte als auch für Links
- Systemoperationen/-ereignisse entsprechen denen der SDs
- Spezielle Syntax für Nachrichten
- Eindeutige Sequenzierung der Nachrichten, da sie nur von aktiven Objekten ausgehen können

Umsetzungsdiskussion

Bereits auf den ersten Blick bietet sich eine Synchronisation mit dem SKM als nützliche Unterstützung an. So sollte direkt beim Erstellen von Objekten der Bezug auf eine Klasse des SKMs gefordert werden, um der Erfüllung der ersten Konsistenzbedingung nachzugehen. Dazu zählt auch die Synchronisation der

Beziehungen des SKMs mit den Links eines KDs, sodass die Kommunikation zwischen Objekten eingeschränkt wird, durch die Beziehungen ihrer korrespondierenden Klassen im SKM. Dies birgt jedoch einige Komplikationen, die noch näher erläutert werden. Bei Objekten lässt sich der Fall, dass im Nachhinein korrespondierende Klassen im SKM gelöscht oder zum Akteur konvertiert werden dadurch abdecken, dass zum Beispiel, wie beim KMG und SKM, die Möglichkeit einer Entkopplung und Neuverknüpfung angeboten wird.
Bei Beziehungen gestaltet sich das wesentlich schwieriger. Das Neuverbinden von Beziehungen im SKM, stellt das gleiche Problem dar, wie das Neuverknüpfen von Objekten, mit einer anderen Klasse des SKMs: Bestehende Links müssen neu auf Konsistenz überprüft werden, d.h. es muss verifiziert werden, ob sie laut SKM zulässig sind. Daraufhin steht wieder zur Frage, ob inkonsistente Zustände graphisch hervorgehoben (wie beim SKM) oder verboten (wie beim AD) werden sollen. Da KDs jedoch diverse Synchronisationselemente besitzen, müssten inkonsistent werdende Elemente bei letzterem Fall jeweils gelöscht werden, was aufgrund der Erwartungskonformität nicht infrage kommt. Es liegt sicher nicht im Interesse des Nutzers, beim Bearbeiten eines Artefakts, gleichzeitig an anderer Stelle Informationen zu verlieren.
Die andere Möglichkeit zur Synchronisation der Links hingegen, birgt einen sehr hohen Implementierungsaufwand, da sowohl Synchronisation, als auch Konsistenzchecks mit graphischer Hervorhebung sowie Neuverknüpfbarkeit von Links konzipiert und programmiert werden müssen.
Akteure hingehen können ebenfalls, wie Objekte synchronisiert werden. Da die Systemoperationen dem SD entsprechen, welches direkt nach dem UCD erstellt wird, ist es auch sinnvoll, die Akteure, die diese anstoßen direkt aus dem UCD zu beziehen, also Akteure des KDs mit den Akteuren des UCDs zu synchronisieren, anstatt mit denen des SKMs. Zusätzlich senkt es den Programmieraufwand, da neben dem Fall, dass dieser gelöscht wird, nicht noch der betrachtet werden muss, dass er im SKM in einem anderen Klassentyp konvertiert wurde.
Eine andere Möglichkeit der Synchronisation besteht mit dem SD. Da zu jeder Systemoperation im SD ein KD angelegt werden muss, kann hier ähnlich vorgegangen werden, wie bei der Synchronisation von UCD und ADs. Analog zu dieser Variante, wo für jede erstellte Kommunikation zwischen Akteur und Use Case, automatisch ein AD generiert wird, kann in SDs für die angelegte Systemoperati-

on auch ein KD generiert werden, dessen Name sich nach dieser richtet.
Die Syntax von Nachrichten hingegen lässt sich zum Beispiel mithilfe eines Algorithmus, der Muster abgleicht (engl. pattern matching), prüfen. So kann, unter Beachtung der Usability-Konzepte, die Konsistenz der Syntax von Nachrichten eingefordert werden. Des Weiteren können bei Fehleingaben diverse Standard-Vorschläge für mögliche, korrekte Varianten angezeigt werden, die den Benutzer in der Methodik anleiten und damit zur Selbsterklärungsfähigkeit des Editors beitragen.
Ein zusätzlich mögliches Feature ist die automatische Nummerierung von Nachrichten. Allerdings bedeutet dies, dass der Benutzer Nachrichten genau in der Reihenfolge erstellt, wie sie nummeriert werden sollen. Wenn er allerdings im Nachhinein Nachrichten hinzufügt, ist es unter Umständen nicht mehr eindeutig, welche Nummern vergeben werden müssen. Dies passiert zum Beispiel, wenn mehrere Nachrichten über den selben Link gesendet werden. Da nicht automatisch festgestellt werden kann, zu welcher Nachricht bzw. zu welcher vorangegangenen Nummer die neue Nachricht gehört, ist die Nummerierung nicht mehr eindeutig durchführbar.

3.2.8 Entwurfsklassenmodell (EKM)

Das Entwurfsklassenmodell stellt die Verfeinerung des Systemklassenmodells dar. Unter Berücksichtigung der neuen Informationen aus den Kommunikationsdiagrammen, werden nun Methoden aus den Nachrichten dieser abgeleitet und den Klassen hinzugefügt, welche inzwischen alle Attribute aufführen, wobei jedoch Referenzen nach wie vor als Assoziationen realisiert bleiben.
Zur Vereinfachung nehmen wir jedoch an, dass die Existenz einzelner Klassen, inklusive deren Attribute, bereits im SKM festgelegt wurde, und im EKM lediglich zusätzliche Assoziationen hinzugefügt sowie Änderungen bezüglich der Namensgebungen und der Multiplizitäten vorgenommen werden können.

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- SKM bildet Grundlage für EKM

- nur Assoziationen und Methoden können noch zusätzlich hinzugefügt und auch nur diese wieder entfernt werden
- Methoden leiten sich aus Nachrichten der KDs ab, dabei gelten folgende Regeln:
  - Systemoperationen treten als Methoden in Controls auf
  - Methoden werden den Klassen im EKM zugewiesen, deren Objekte Ziel der jeweiligen Kommunikation im KD waren - handelt es sich um eine Collection, wird Methode der Klasse zugeordnet, deren Objekt Ausgang der entsprechenden Kommunikation war

Umsetzungsdiskussion

Aus der ersten Bedingung geht hervor, dass die Elemente des EKMs im Prinzip komplett mit denen des SKMs synchronisiert werden können, da beide Artefakte, bis auf zusätzliche Assoziationen und Methoden im EKM, alle Elemente gemeinsam haben. Dies ist wesentlich einfacher zu implementieren, als die bisher beschriebenen Synchronisationen, weil es hier keine inkonsistenten Zustände gibt, d.h. Features, wie die Entkopplung und Neuverknüpfung von Elementen, sowie die damit verbundenen Fehlerquellen, fallen komplett weg. Dafür muss jedoch das Ändern der vollständig synchronisierten Elemente im EKM unterbunden werden, um Rückwirkungen auf das SKM zu vermeiden, da dies nicht im Sinne der Methodik wäre.
Um zu kennzeichnen, welche zusätzlichen Assoziationen der Benutzer im EKM angelegt hat, welche er also editieren kann und welche nicht, kann ähnlich vorgegangen werden, wie bei der Synchronisation vom KMG und dem SKM. So können die schon implementierten Synchronisations-Status-Icons für Beziehungen wiederverwendet werden, um sie mit leichten Anpassungen für diesen Zweck zu benutzen.
Ein anderes sich anbietendes Feature, ist die automatische Ableitung von Methoden aus den Nachrichten der Kommunikationsdiagramme. Gerade, wenn deren korrekte syntaktische Struktur gefordert wird, wie aus der vorherigen Umsetzungsdiskussion hervorgegangen sein sollte, ist mithilfe des dafür entwickelten Algorithmus auch eine Transformation der Nachrichten in Methoden für Klassen des EKMs leichter umsetzbar. So wird dem Benutzer viel unangenehme Arbeit abgenommen, da das Abschreiben von Methoden nicht nur einen Namen, sondern

ggf. auch diverse Parameter einschließt. Dabei sollen auch die Ableitungsregeln realisiert werden, sodass schließlich die Ableitung von Methoden komplett automatisiert abläuft.

3.2.9 Implementationsklassenmodell (IKM)

Das Implementationsklassenmodell stellt die Grundlage für die bevorstehende Implementierung dar und wird aus dem EKM abgeleitet. Es enthält letztlich eine vollständige Beschreibung aller Klassen. Akteure und Boundaries werden nicht mehr dargestellt, da sie nicht relevant für die Implementierung sind. Da das IKM nur noch Klassen darstellt, werden die Assoziationen des EKMs den Klassen als weitere Attribute zugeordnet.

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Klassen, Attribute und Methoden, werden aus EKM übernommen
- Assoziationen des EKMs werden den Klassen des IKMs als Attribute zugeordnet (immer der Klasse, von der Assoziation im EKM ausgegangen ist)
- Attribute und Methoden müssen vollständig beschrieben sein (Typen, Sichtbarkeiten, Multiplizitäten)

Umsetzungsdiskussion

Für das Implementationsklassenmodell bietet sich eine andere Art der Synchronisation an, nämlich die einmalige Erzeugung aus dem EKM. Da das IKM das letzte Artefakt ist, welches in der in dieser Arbeit beschriebenen Methodik bearbeitet wird, unterliegt es keinen Abhängigkeiten, außer der Konformität mit dem Entwurfsklassenmodell.
Eine fast komplett automatische Generierung des IKMs ist möglich, indem die Klassen des EKMs, inklusive deren Inhalt, in das IKM kopiert werden. Dabei können ihre Attribute alle als privat (engl. private) ausgezeichnet und gleichzeitig öffentliche (engl. public) Methoden für den Zugriff auf diese, die sogenannten *getter- und setter-Methoden*, erzeugt werden.

Um jedoch die Korrektur von eventuellen Fehlern oder das Anpassen der Darstellung von Attributen bzw. Methoden anzubieten, muss, besonders im Sinne der *Steuerbarkeit*, das Editieren generierter Elemente trotzdem ermöglicht werden.

3.2.10 Data Dictionary

Das Data Dictionary (deut. Datenkatalog) dient der Dokumentation von Artefakten. Dabei soll es für die in dieser Arbeit vorgestellte Methodik, genügen, wenn es die einzelnen Elemente der Artefakte beschreibt.

Wichtige Konsistenzbedingungen

Folgende Bedingungen werden an dieses Artefakt gestellt:

- Alle wichtigen Elemente aller Artefakte müssen eindeutig abgebildet werden

Umsetzungsdiskussion

Die Automatisierung eines Datenkatalogs ist von größerer Bedeutung, als es vielleicht im ersten Moment bzw. bei ersten, sehr kleinen Projekten den Eindruck macht. Da unter Umständen eine Fülle an ADs, SDs und KDs zu erstellen sind, neben den anderen Artefakten, kann der Dokumentationsaufwand mitunter sehr groß ausfallen, woraus sich auch ein gewisses Fehlerpotenzial ergibt.
Um dem Benutzer diese Zeit und eventuelle Fehler zu ersparen, gibt es eine relativ einfache, zweckmäßige Realisierungsmöglichkeit: Eine Iteration über die einzelnen Artefakte, genauer gesagt über die Informationen der dargestellten Elemente. Diese können nach einem bestimmten Schema in eine Datei geschrieben werden, die zum Beispiel das *Comma-Separated Values*-Format (.csv) verwendet, sodass die Daten später tabellarisch abgebildet werden können. Die wichtigsten Spaltenköpfe sind dabei: Artefakt (z.B. KMG), Elementname (z.B. MeineKlasse) und Elementtyp (z.B. Klasse), wobei für Beziehungen noch Start und Ziel zu berücksichtigen sind sowie eine generelle Spalte für Kommentare.

4 Implementierung

Nachdem die Anforderungen an den graphischen Editor beschrieben wurden, geht es nun um die konkrete Umsetzung dieser. Der graphische Editor ist als ein Plugin für *Eclipse* entwickelt worden und zwar mithilfe der Eclipse-Plugins *EMF*, GEF und *MuvitorKit*. Nachdem die verwendeten Technologien erläutert wurden, werden anschließend die Realisierungen der einzelnen Artefakte aus dem letzten Kapitel beschrieben und diskutiert, unter dem Punkt *Metamodelle*.

4.1 Eclipse

Eclipse stellt eine Plattform dar, die nicht nur eine breite Palette an bereits vorhandenen Plugins umfasst, sondern auch das Kreieren und Integrieren neuer unterstützt. Diese Plugin-Architektur ermöglicht es, mithilfe bestehender Plugins sowie den Bestandteilen der Plattform, zum Beispiel GUI-Bibliotheken, Editoren, Sichten, Wizards usw., eigene Applikationen zu entwickeln und diese innerhalb von Eclipse auszuführen. [Fou10] Außerdem ist Eclipse als ausgezeichnete Open-Source-Entwicklungsumgebung für Java bekannt, was sich ebenfalls als nützlich erweist, da der Editor letztlich in Java entwickelt wird. [DB04]
Die Plugins, die für den entwickelten graphischen Editor eine entscheidende Rolle spielen, sind das *Eclipse Modeling Framework (EMF)* sowie das *Graphical Editing Framework (GEF)*. Das Zusammenwirken dieser beiden Plugins umfasst zum einen das automatisierte Erzeugen von Quelltext aus strukturierten Modellen (EMF) sowie das relativ schnelle, jedoch manuelle Erstellen eines graphischen Editors auf der Basis eines existierenden Datenmodells (GEF). So ein Datenmodell ist zum Beispiel der von EMF erzeugte Code. Wie diese beiden Werkzeuge aufgebaut sind und funktionieren, wird im Folgenden umrissen. Das eingangs erwähnte *MuvitorKit* spielt bei der Entwicklung eher eine unterstützende Rolle, indem es Funktionalitäten von GEF erweitert, vereinfacht oder Vorlagen für diese bietet.

4.2 EMF

Das Eclipse Modeling Framework stellt eine Basistechnologie dar, mit dem Metamodelle beschrieben werden. Letztere werden unter anderem folgendermaßen definiert:

> „Metamodelle sind Modelle, die etwas über Modellierung aussagen. Genauer: Ein Metamodell beschreibt die mögliche Struktur von Modellen - es definiert damit in abstrakter Weise die Konstrukte einer Modellierungssprache, ihre Beziehungen untereinander sowie Gültigkeits- bzw. Modellierungsregeln." [SVEH07]

Das Modell, welches beschreibt, wie solche mit EMF erstellbaren Metamodelle aussehen, wird demzufolge Metametamodell genannt und trägt den Namen *Ecore*. Ecore ähnelt sehr der, von der *Object Management Group* eingeführten, *Essential Meta Object Facility (EMOF)*, welches EMF auch unterstützt. [SBPM09] Zum Erstellen von Ecore-Modellen bietet EMF entweder einen Baum-Editor als textuelle Variante oder einen graphischen Editor an (Abbildung 4.1).

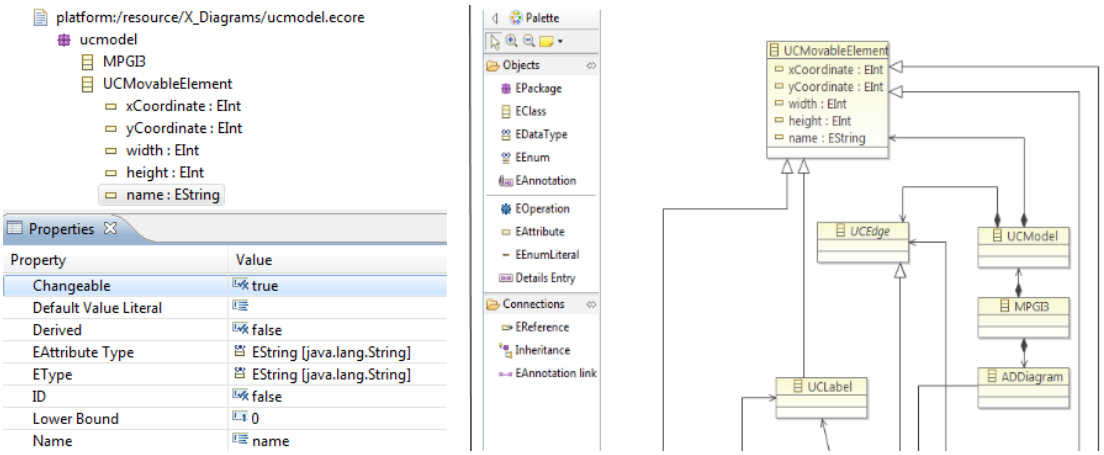

Abbildung 4.1: Textueller und Graphischer Editor zur Erstellung von Metamodellen mit EMF

Eine Ecore-Datei (*.ecore), enthält eine XML-basierte Beschreibung der gewünschten Metamodellelemente sowie deren Beziehungen zueinander. Für diese Dateien existiert, als Teil von EMF, ein Generator, der das beschriebene Metamodell in Java-Code *transformiert*, es also in eine alternative Repräsentation überführt, um

es in einem anderen Kontext zu benutzen, in unserem Fall also als Datenmodell für GEF. [SVEH07]

4.3 GEF

> „The Graphical Editing Framework allows us to easily develop graphical representations for existing models. It is possible to develop feature rich graphical editors using GEF." [BM04]

GEF ist ein Eclipse Plugin, welches eine relativ einfache Entwicklung komplexer, graphischer Editoren ermöglicht. Für die Darstellung von Elementen nutzt es das *Draw2D*-Framework, welches auf dem von Eclipse vorgegebenen Standard *SWT* basiert. Seine Architektur folgt einem bekannten Entwurfsmuster der Softwareentwicklung, dem *Model-View-Controller- (MVC) Prinzip*, welches hauptsächlich die Trennung von Daten (Model) und deren Darstellung (View) sowie deren Verwaltung (Controller) vorschreibt (Abbildung 4.2). [SVEH07]

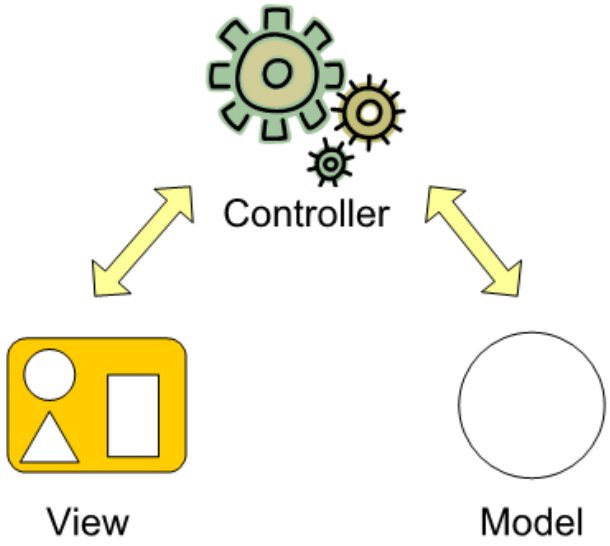

Abbildung 4.2: Model-View-Controller-Prinzip [DISJ09]

So stellt nun der von EMF generierte Code das *Model* dar, das für die persistente Datenhaltung zuständig ist. Jedes anzuzeigende Modell-Element ist indirekt mit einem Darstellungselement verknüpft, was GEF mithilfe von Draw2D-*Figures* realisiert, die den Kern vom *View* ausmachen. Für das Steuern des gesamten Editier-

vorgangs stellt GEF sogenannte *EditParts* bereit, die das Bindeglied zwischen den Model- und View-Elementen darstellen. Für jedes anzuzeigende Modell-Element muss ein *EditPart* erstellt werden, in dem diverse Funktionalitäten zur Aufbereitung der Darstellung von Modell-Elementen implementiert werden, unter anderem auch das Erstellen von korrespondierenden View-Elementen in Form von *Figures*. [BM04]
Weitere, zum Verständnis notwendige, Elemente sind:

- *Requests* - Vom Benutzer getätigte Interaktionen mit der GUI des Editors
- *EditPolicies* - Enthalten Informationen, wie auf welche Aktionen (*Requests*) reagiert wird, werden in EditParts registriert
- *Commands* - Ändern das Model
- *Adapter* - registrieren Controller bei einem Modell-Element, um den Empfang von *Notifications* zu ermöglichen (Beobachter-Entwurfsmuster[5])
- *Notifications* - Informieren Controller über Änderungen am Model

Eine einfache Aktion des Benutzers, wie etwa das Verschieben einer *Figure*, die zum Beispiel die Repräsentation einer Klasse des KMGs sein könnte, würde folgendermaßen ablaufen:

Vom View zum Model
Durch die Interaktion des Benutzers mit der *View*, generiert der Editor eine *Request* und leitet diese an den *Controller*, also den entsprechenden *EditPart* weiter. Dieser sucht in seinen registrierten *EditPolicies* die passende heraus, und übergibt die Request an diese. In der EditPolicy werden dann die nötigen *Commands* erstellt, um die Aktion letztlich auch im *Model* umzusetzen. Im Oben genannten Beispiel wäre das die Erstellung eines Commands, welches die neuen Koordinaten der Klasse im Model speichert. Bis jetzt hat sich an der Darstellung unserer Klasse nichts geändert, die View muss erst anhand der neuen Informationen im Model

[5] „Das Beobachter-Entwurfsmuster bietet ein Konzept, mit dem Ereignisquellen und Ereignissenken voneinander entkoppelt werden. Ereignisquellen sind [...] die Auslöser der Ereignisse. Ereignissenken oder Beobachter können sich bei den Ereignisquellen registrieren und werden informiert, sobald ein Ereignis eintritt" [Ari10]

aktualisiert werden.

Vom Model zurück zum View
Das von der Änderung betroffene Modell-Element sendet eine *Notification* an seinen zugehörigen *EditPart*, der sich vorher mithilfe eines *Adapters* bei diesem registriert hat. Der EditPart wiederum wertet die Notification aus und aktualisiert die von ihm angelegte *Figure*, ändert also das korrespondierende *View*-Element, was in unserem Beispiel das Verschieben der Klasse wäre.

4.4 MuvitorKit

Das MuvitorKit (Multi-View Editor Kit) ist ein an der TU-Berlin entwickeltes Framework, welches nicht nur GEF erweitert, indem es ermöglicht, mehrere graphische Sichten auf ein einzelnes Modell-Element zu haben, sondern es stellt auch so etwas, wie eine vereinfachte Vorlage für neue GEF-Projekte dar, mit diversen bereits implementierten Standard-Funktionen. Außerdem erleichtert das MuvitorKit den Zugriff auf Elemente der Eclipse-Umgebung, da es komplexe, zugrunde liegende GEF-Mechanismen kapselt. [TM09]

4.5 Metamodelle

In diesem Abschnitt geht es darum, die mit EMF erstellten Metamodelle der einzelnen Artefakte des vorherigen Kapitels zu erklären, welche alle in einem übergeordneten Paket vereint wurden, um sie in einem Editor verwenden zu können (Abbildung 4.3). Dabei spielt vor allem die bereits des Öfteren erwähnte Synchronisation eine größere Rolle, da sie durch bestimmte Metamodellelemente unterstützt wird. Genau genommen synchronisiert, in dem mit dieser Arbeit verbundenen Editor, nicht das Model die Elemente der Artefakte, sondern *Commands* erledigen dies und zwar folgendermaßen: Wenn ein Modell-Element eines Artefakts verändert wird, wurde dafür vorher von GEF ein Command erstellt. Indem das Command so implementiert wird, dass gleichzeitig auch ein anderes Modell-Element verändert wird, kann zum Beispiel eine Synchronisation realisiert werden.
Wie einzelne Metamodellelemente zur Synchronisation beitragen, folgt ggf. jeweils

auf die Erläuterung der wesentlichen Bestandteile des Metamodells eines Artefakts. Der Übersicht halber werden die Metamodelle nicht komplett abgebildet, sondern nur Ausschnitte der wichtigsten Modell-Elemente und deren Beziehungen zueinander.

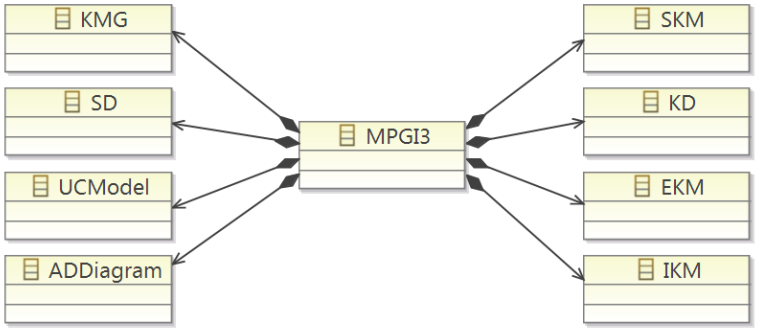

Abbildung 4.3: „MPGI3" als übergeordnetes Paket für die einzelnen Artefakte

Anmerkung: Im Folgenden werden zur besseren Unterscheidung Metamodellelemente *kursiv* geschrieben. Modell-Elemente sind als Instanzen der Metamodellelemente zu verstehen, also als konkrete, persistente Informationsträger eines Diagramms, das mit dem graphischen Editor erstellt wurde.

4.5.1 KMG

Das Root-Element des Metamodells stellt bei allen Artefakten *MPGI3* dar, darauf folgt immer das Metamodellelement des betrachteten Artefakts, hier also *KMG*. Dieses beinhaltet die zwei ausschlaggebenden Elemente eines KMGs: Klassen (*KMGNode*) und Beziehungen (*KMGEdge*). Die beiden stehen in folgendem Kontext zueinander: Klassen haben ein- und ausgehende Beziehungen, während letztere Klassen als Start und Ziel haben. Klassen haben im KMG lediglich noch Attribute (KMGAttribute), während Beziehungen in die Unterkategorien Generalisierung/Spezialisierung (*KMGGeneralization*), Aggregationen (*KMGAggregation*) und Komposition (*KMGComposition*) unterteilt werden, d.h. Assoziationen werden als nicht spezifizierte *KMGEdge* repräsentiert. Zur Darstellung einer Beziehung gehören jedoch auch textuelle Beschreibungen (*KMGLabel*), die spezialisiert

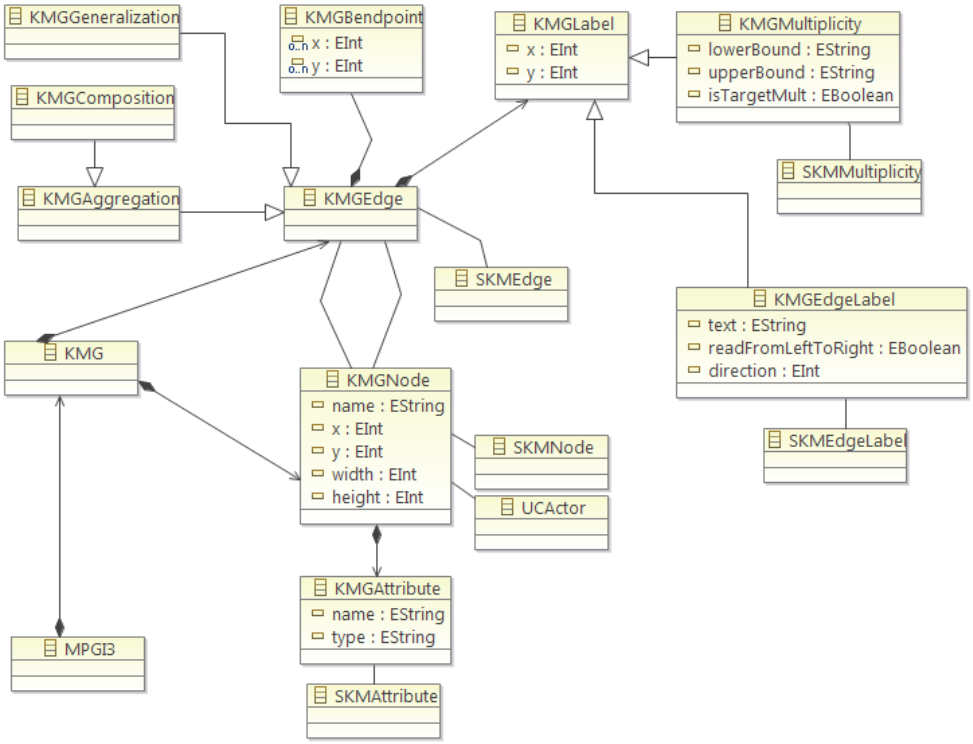

Abbildung 4.4: Metamodell des KMGs

als Name (*KMGEdgeLabel*) und Multiplizitäten (*KMGMultiplicity*) auftreten. Außerdem besitzen Beziehungen zur besseren graphischen Darstellung Bendspoints (*KMGBendpoint*), welche persistente Angelpunkte für den Verlauf von Beziehungen darstellen und somit auch als Metamodellelemente in Metamodellen anderer Artefakte wiederzufinden sind.

Realisierung der Synchronisation

Um in einem Artefakt Zugriff auf jegliche Elemente eines anderen Artefakts zu haben, müssen die Metamodellelemente der Artefakte verknüpft werden. Dies geschieht dadurch, dass die Metamodellelemente aller Artefakte *MPGI3* untergeordnet und somit über dieses erreichbar sind (Abbildung 4.3). So kann im KMG durch die Elemente der anderen Artefakte iteriert werden, um zum Beispiel später für eine Synchronisation infrage kommende Modell-Elemente anzuzeigen. Wie weiterhin in der Abbildung zu sehen ist, können Klassen, Attribute und Beziehungen sowie deren textuelle Beschreibungen mit dem SKM synchronisiert werden, da sie

jeweils eine Referenz auf ein gleichartiges Element des SKMs besitzen. Klassen haben des Weiteren die Möglichkeit, mit Akteuren des UCD zu korrespondieren, sodass zum Beispiel letztere tatsächlich auf Klassen des KMGs beruhen.

4.5.2 UCD

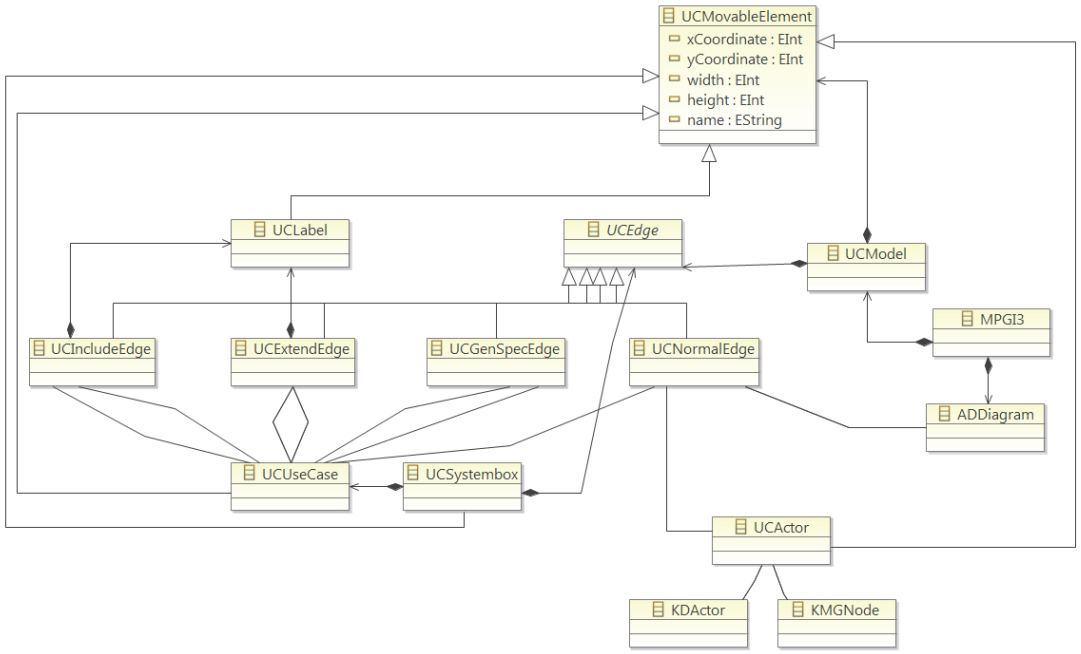

Abbildung 4.5: Metamodell des UCDs

Das Metamodellelement des UCD ist *UCModel*, welches zum einen Beziehungen (*UCEdge*) und zum anderen eine Oberklasse für frei bewegliche Metamodellelemente (*UCMovableElement*) beinhaltet, worunter die Systembox (*UCSystembox*), Akteure (*UCActor*) und schließlich die Use-Cases (*UCUseCase*) fallen. Beziehungen werden unterteilt in Einbindungs- (*UCIncludeEdge*), Erweiterungs- (*UCExtendEdge*) und Generalisierungs- / Spezialisierungsbeziehungen (*UCGenSpecEdge*), die lediglich zwischen Use-Cases hergestellt werden können, sowie in Kommunikationen (*UCNormalEdge*), also Verbindungen zwischen einem Use-Case und einem Akteur. Von diesen besitzen aber nur Einbindungs- und Erweiterungsbeziehungen zusätzliche textuelle Beschreibungen (*Label*). (Abbildung 4.5)

Realisierung der Synchronisation

Das UCD benötigt Zugriff auf jegliche Elemente des KMGs, zum Beispiel wenn ein neuer Akteur angelegt wird und dafür sämtliche verfügbaren Klassen des KMGs als dessen Basis bereitgestellt werden müssen. Dieser Zugriff kann wieder über *MPGI3* erfolgen. Wie schon beim KMG besteht die Möglichkeit, Akteure des UCDs mit Klassen des KMGs zu synchronisieren, da Referenzen zwischen ihren Metamodellelementen bestehen. Des Weiteren besitzen Kommunikationen eine Referenz auf Aktivitätsdiagramme (*ADDiagram*), was der im vorherigen Kapitel beschriebenen Synchronisation von UCD und AD dienlich ist, denn zu jeder Kommunikation im UCD soll ein korrespondierendes Aktivitätsdiagramm erstellt werden. Die Erklärung der Verknüpfung von *UCActor* und *KDActor* hingegen erfolgt erst bei den Kommunikationsdiagrammen.

4.5.3 SD

Das Metamodellelement eines Sequenzdiagramms stellt *SD* dar. SDs beinhalten im Wesentlichen Akteure, zusammen mit deren Lebenslinien, als *SDLifeline*, Nachrichten zwischen diesen als *SDMessage*, sowie Begrenzungen für Sonderfälle bzw. Alternativen als *SDBoxes*.
Das Problem bei Sequenzdiagrammen besteht darin, dass ihre Darstellung nicht auf Knoten und Beziehungen basiert, wie es bei allen anderen Artefakten der Fall ist. Da GEF keine direkte Lösung für dieses Problem anbietet, muss hier eine eigene Alternative entwickelt werden, um den Nachrichtenaustausch zwischen Lebenslinien abbilden zu können.
Die mit am wenigsten Aufwand und Fehlerpotenzial verbundene Variante ist, die Darstellung von SDs letztlich doch auf das Prinzip von Knoten und Beziehungen zurückzuführen. Dafür wird ein ,,künstliches" Knoten-Element erschaffen, die sogenannten Aktivitäten (*SDActivity*), welche als Bestandteil von Lebenslinien, auf diesen in einem einheitlichen Abstand verteilt sind. Dadurch existieren nun Verankerungen für Beziehungen, sodass Nachrichten jetzt Aktivitäten als Start bzw. Ziel haben. (Abbildung 4.6)
Weiterhin besteht ein Problem bei der Realisierung von Begrenzungen (*SDBoxes*). Letztere werden untergliedert in Begrenzungen für optionale Abläufe (*SDOpt*) sowie Begrenzungen mit verschiedenen Bedingungen (*SDAlt*), wobei die Bedin-

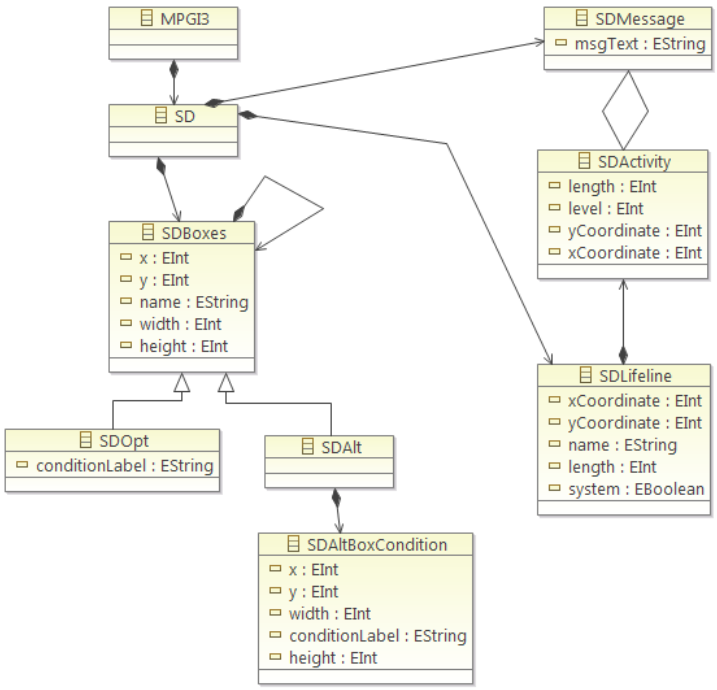

Abbildung 4.6: Metamodell des SDs

gungen ausgelagert sind in *SDAltBoxCondition*. Damit später verschiedene Begrenzungen ineinander geschachtelt werden können, hat *SDBoxes* eine Referenz auf sich selbst. Bei der Umsetzung der Verschachtelung ergibt sich allerdings ein Problem, da durch das Metamodell dafür eine klare Hierarchie definiert wird, der auch die als Boxen dargestellten Begrenzungen unterliegen. Das bedeutet, das Verändern und Verschieben solcher Boxen, ist nur eingeschränkt möglich, im Bereich ihrer jeweiligen übergeordneten Box. Um solche Boxen sinnvoll verwenden zu können, muss die Funktionalität erweitert werden, sodass einer Box zum Beispiel auch eine andere übergeordnete Box zugeordnet werden kann. Die einfachste Variante stellt dabei das Ignorieren bestehender Hierarchien beim Verschieben und Verändern von Boxen dar. So hat der Benutzer wenigstens die Möglichkeit, Begrenzungen frei einzusetzen und vor allem wieder zu verwenden.

4.5.4 AD

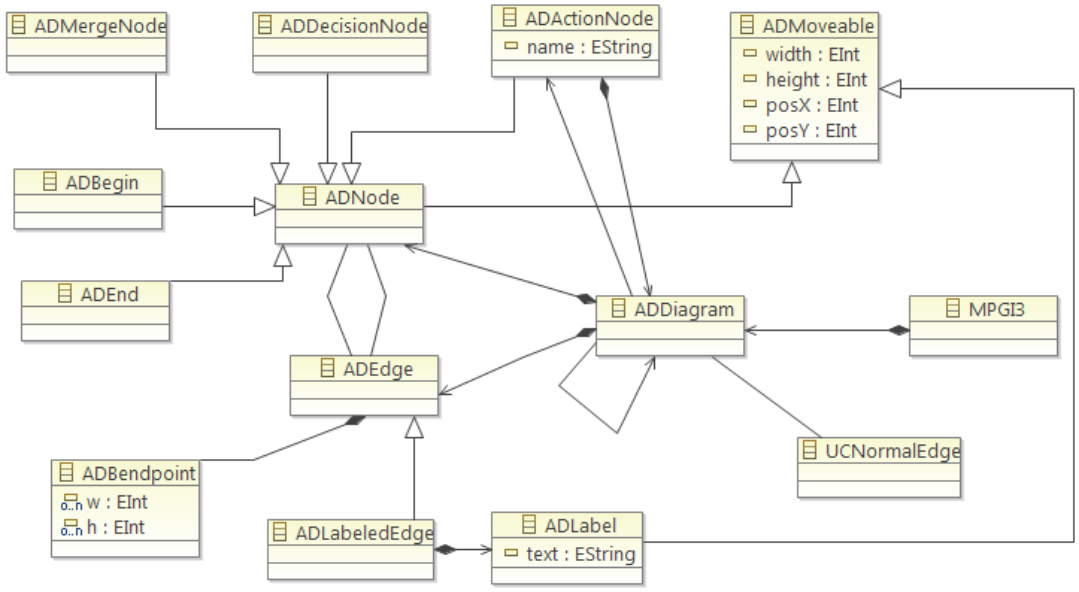

Abbildung 4.7: Metamodell des ADs

Für Aktivitätsdiagramme repräsentiert *ADDiagram* das entsprechende Metamodellelement. Dieses beinhaltet Knoten (*ADNode*) und Beziehungen (*ADEdge*), wobei für letztere eine spezialisierte, mit Text versehene Form, namens *ADLabeledEdge* existiert. Der Text wird allerdings in einem extra Metamodellelement (*ADLabel*) gespeichert, welches, ähnlich wie beim UCD, einer Oberklasse für frei bewegliche Metamodellelemente angehört: *ADMoveable*. Dieser sind somit auch Knoten (*ADNode*) zuteil, welche zusätzlich spezifiziert werden in Start- (*ADBegin*), End- (*ADEnd*), Zusammenführungs- (*ADMergeNode*) und Entscheidungsknoten (*ADDecisionNode*) sowie Aktionen (*ADActionNode*). Mithilfe gemeinsamer Oberklassen, kann allerhand Redundanz bei der Implementierung eingespart werden, da sich Unterklassen diverse Funktionalitäten teilen können, wie zum Beispiel die Implementierung eines Commands zum Ändern von gemeinsamen Attributen.
Um ADs später schachteln zu können, müssen mehrere Voraussetzungen geschaffen werden. Untergeordnete ADs sind im Prinzip nichts anderes, als genauere Beschreibungen einer Aktion in Form eines neuen ADs. Um eine Hierarchie unter ADs zu ermöglichen, kann eine Aktion (*ADActionNode*) ein AD (*ADDiagram*)

besitzen, um sein untergeordnetes AD festzulegen und Zugriff auf dieses zu haben. Des Weiteren kann ein AD eine Referenz auf eine Aktion haben, um es seiner übergeordneten Aktion zuzuordnen. Damit innerhalb eines ADs ein möglichst komfortabler Zugriff auf untergeordnete ADs möglich ist, zum Beispiel für Konsistenzüberprüfungen, die auf diese angewandt werden sollen, referenziert sich *AD-Diagram* selbst. Das bedeutet, ein AD besitzt eine Menge von untergeordneten ADs, auf die so bequem zugegriffen werden kann.

4.5.5 SKM

SKM stellt das Metamodellelement für das gleichnamige Artefakt dar. Da das SKM auf dem KMG basiert, wurde auch sein Metamodell auf Grundlage dessen entworfen, sodass sich die Metamodelle, abgesehen vom Präfix ihrer Elemente, nur in wenigen Punkten unterscheiden. Neu ist lediglich die Untergliederung von Klassen (*SKMNode*) in Übergangs- (*SKMBoundary*), Gegenstands- (*SKMEntity*) und Steuerungsklassen (*SKMControl*) sowie Akteure (*SKMActor*), wie in Abbildung 4.8 dargestellt. Die restlichen Unterschiede ergeben sich aus der Synchronisation.

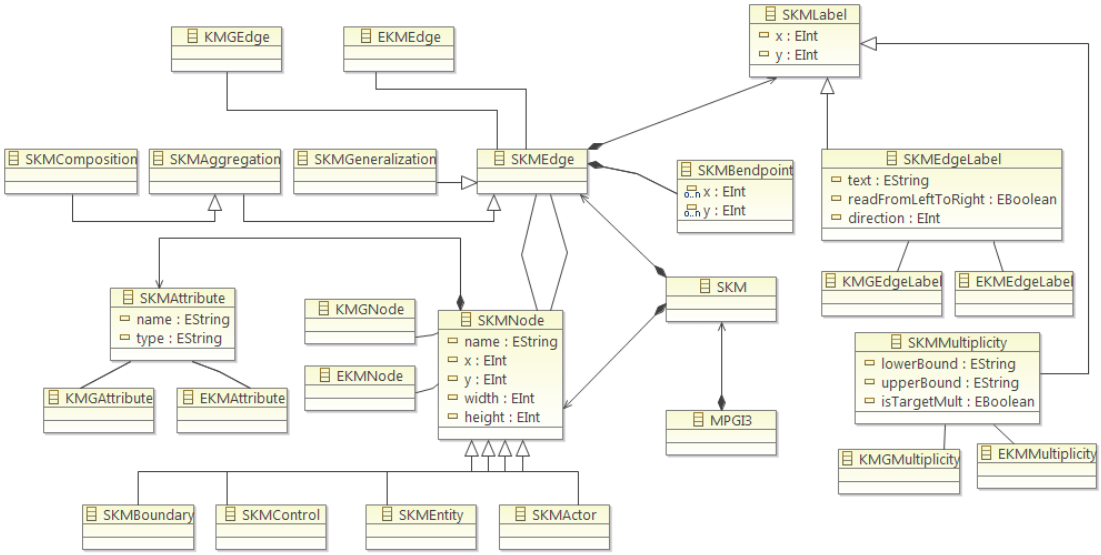

Abbildung 4.8: Metamodell des SKMs

Realisierung der Synchronisation

Auch die Unterstützung von Synchronisation ist ähnlich wie beim KMG gestaltet. Klassen (*SKMNode*), Attribute (*SKMAttribute*) und Beziehungen (*SKMEdge*) sowie deren textuelle Beschreibungen (*SKMMultiplicity* und *SKMEdgeLabel*) haben Referenzen auf gleichartige Metamodellelemente des KMGs und EKMs (jeweils nur mit anderem Präfix), um mit diesen synchronisiert werden zu können. Akteure nehmen hier eine Sonderrolle ein, da sie indirekt, über das KMG, mit dem UCD synchronisiert werden können.

4.5.6 KD

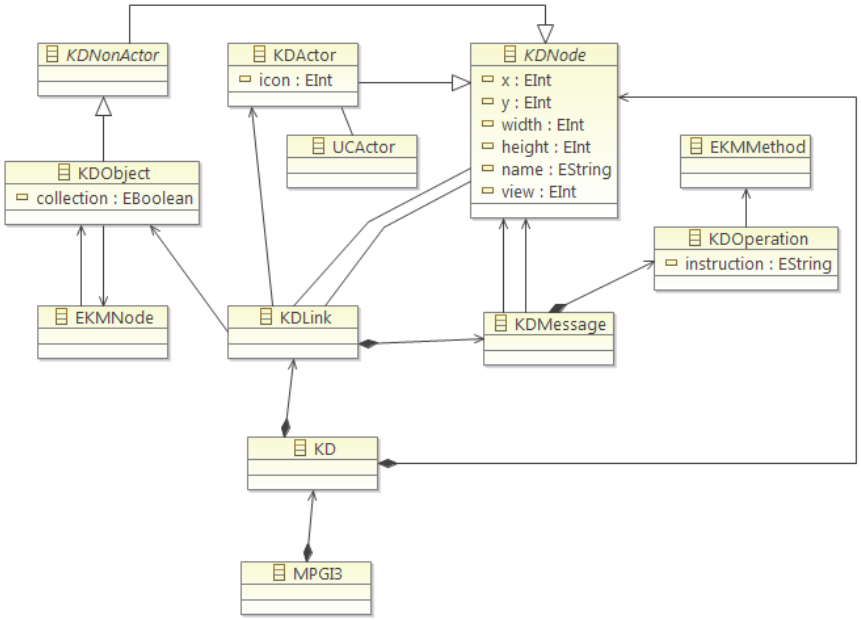

Abbildung 4.9: Metamodell des KDs

KD ist das Metamodellelement für ein Kommunikationsdiagramm. Es beinhaltet die zwei Hauptelemente Knoten (*KDNode*) und Verbindungen (*KDLink*), welche dem selben kausalen Zusammenhang unterliegen, wie zum Beispiel Klassen und Beziehungen des KMGs. Verbindungen können mehrere Nachrichten (*KDMessage*) beinhalten, welche, im Unterschied zur Methodik, nur den Pfeil einer ,,richtigen" Nachricht ausmachen. Im Metamodell sind sie als Behälter für eine

Menge von Operationen (*KDOperationen*) zu interpretieren, die den selben Start- und Endknoten und damit auch die selbe Richtung besitzen. Somit sind die Operationen die eigentlichen Informationsträger, während eine *KDMessage* lediglich Auskunft über Ursprung und Ziel einer Nachricht gibt.
Knoten hingegen werden der Übersicht halber unterteilt in Akteure (*KDActor*) und Nicht-Akteure (*KDNonActor*). Letztere treten jedoch nur in ihrer spezialisierten Form als Collaborators bzw. Objekte (*KDObject*) auf. (Abbildung 4.9)

Realisierung der Synchronisation

Die Synchronisation wird hier von zwei Seiten behandelt. Zum einen werden Akteure direkt aus dem UCD bezogen, deshalb ist *KDActor* mit *UCActor* verbunden. Andererseits besitzen Objekte eine korrespondierende Klasse im EKM (*EKMNode*). Laut Methodik sollen Objekte aus dem SKM bezogen werden, da aber aus der Umsetzungsdiskussion des EKMs resultierte, dass SKM und EKM immer die gleichen Klassen besitzen werden (SKM generiert auch EKM Klassen), können diese auch aus dem EKM bezogen werden. Dieses wird nämlich noch in einem anderen Kontext verwendet. Wie in Abbildung 4.9 zu erkennen ist, kann eine Operation eine Referenz auf eine Methode des EKMs (*EKMMethod*) besitzen. Dies ist notwendig, um später den Klassen des EKMs automatisiert Methoden zuzuweisen bzw. diese wieder zu entfernen, ausgehend von den korrespondierenden Operationen der KDs.

4.5.7 EKM

Das Metamodell des EKMs ist im Grunde genau wie das des SKMs aufgebaut, mit dem Unterschied, dass andere Artefakte Ziel von Synchronisationen sind und dass Klassen (*EKMNode*) nun, neben Attributen (*EKMAttribute*), auch Methoden (*EKMMethod*) besitzen können. (Abbildung 4.10)

Realisierung der Synchronisation

Durch diese Vereinheitlichung der Metamodelle von SKM und EKM, können fast alle ihre Modell-Elemente mit relativ geringem Aufwand synchronisiert werden. Bereits beim KD wurde die Verbindung einer Klasse des EKMs mit Objekten des

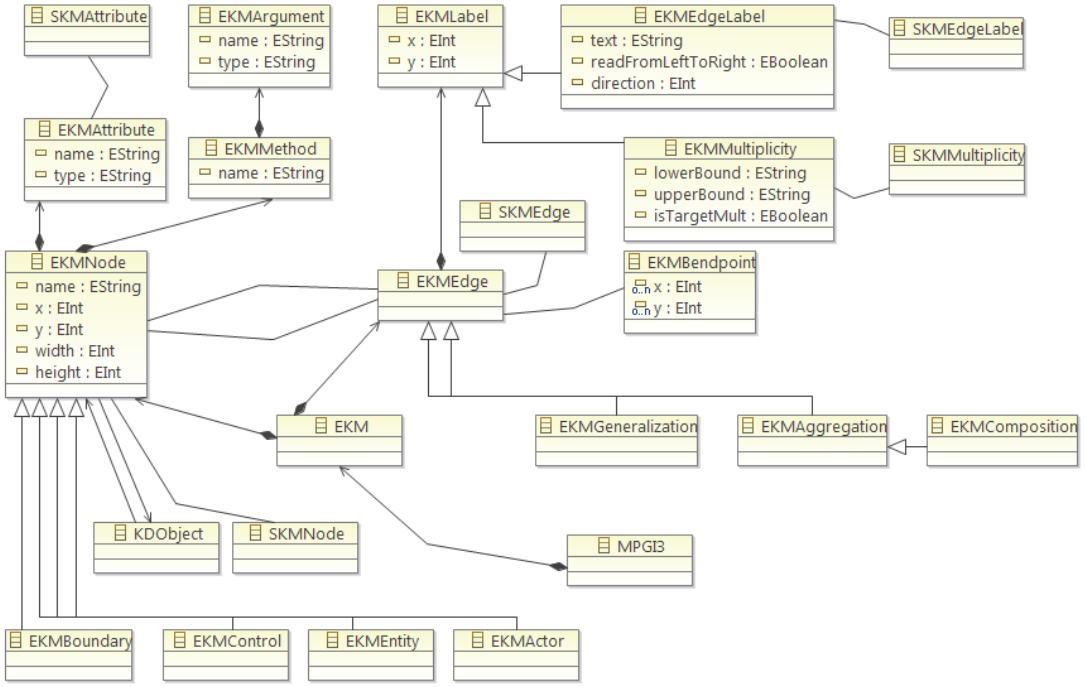

Abbildung 4.10: Metamodell des EKMs

KDs (*KDObject*) erläutert. Diese Verbindung existiert natürlich auch in entgegengesetzter Richtung, um zum Beispiel beim Entfernen einer Klasse des EKMs, auch korrespondierende Objekte der KDs darauf hinzuweisen. Ansonsten würden diese weiterhin ein nun ungültiges Element referenzieren, welches gar nicht mehr existieren dürfte. Da Methoden sich komplett der Manipulation durch den Benutzer entziehen sollen, muss der gerade erläuterte Fall bei ihnen nicht berücksichtigt werden. Sie werden komplett von den KDs gesteuert und benötigen somit keine rückwärtige Referenz.

4.5.8 IKM

Das Metamodell des IKMs beinhaltet, wie es die Methodik vorschreibt, nur noch Klassen (*IKMNode*), welche mehrere Attribute (*IKMAttribute*) und Methoden (*IKMMethod*) beinhalten können. Da es nach Umsetzungsdiskussion aus dem EKM generiert wird und gleichzeitig das letzte Artefakt darstellt, müssen keine Referenzen zu Metamodellelementen anderer Artefakte existieren. (Abbildung 4.11)

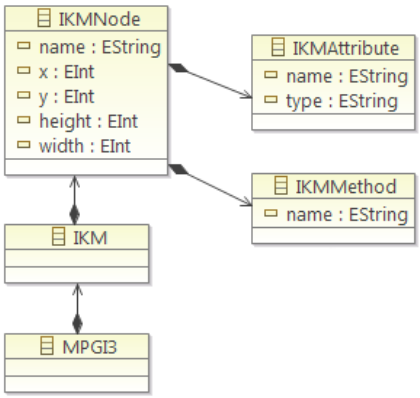

Abbildung 4.11: Metamodell des IKMs

Die Aufbau der einzelnen Klassen des IKMs, inklusive deren Inhalt, leitet sich zum einen aus den Klassen des EKMs her, wobei lediglich Entities (*EKMEntity*) und Controls (*EKMControl*) von Bedeutung sind. Dieser Teil ist automatisierbar, da die Klassen, Attribute und Methoden lediglich kopiert und höchstens kleine Anpassungen zugunsten des angezeigten Formats vorgenommen werden. Auf der anderen Seite sind aber auch die Assoziationen zwischen diesen Klassen entscheidend. Da es jedoch keinen eindeutigen Algorithmus zur Transformation dieser Assoziationen in Attribute des IKMs gibt, muss dieser Schritt weiterhin manuell geschehen.

5 Test

Nachdem bisher die *Anforderungen* für den graphischen Editor, *RUP* als deren Grundlage und die letztendliche Umsetzung im Kapitel *Implementierung* abgehandelt wurde, wird die entwickelte Software zum Abschluss *Tests* unterzogen, als eine Maßnahme zur Qualitätssicherung.
Es gibt mehrere Möglichkeiten, Testen zu definieren, da es sich, wie die Softwareentwicklung selbst, im ständigen Wandel befindet. So nahm der Umfang dieses Begriffs stetig zu: Vom anfänglichen, reinen Fehlerfinden, über den Nachweis von Funktionalität, bis hin zur Qualitätsprüfung und dem Vorbeugung von Fehlern.
Warum es letztlich nötig ist, Tests in einen Softwareentwicklungsprozess zu integrieren, illustriert unter anderem folgender Vergleich recht gut:

> ,,Ein guter Test ist wie eine Haftpflichtversicherung: Er kostet richtig Geld, lässt aber den Projektleiter und den Kunden ruhig schlafen. Zum guten Schlaf gehört auch eine gute Versicherung, die alle möglichen Risiken abdeckt. Zum Vertrauen in die Software gehört ein guter Test, der die ganze Produktionswirklichkeit abdeckt." [Sie03]

Diese Bachelorarbeit beschränkt sich aufgrund der immensen Komplexität der Thematik, lediglich auf eine Möglichkeit zum *dynamischen Testen* des erstellten graphischen Editors.

5.1 Dynamische Tests

Dynamisches Tests sind die Ausführung der zu testenden Software mit stichprobenartigen Eingaben. Die sorgfältige Auswahl einer Menge von Eingaben ist dabei Bestandteil der systematischen Erstellung von Testfällen, welche die Software möglichst umfassend überprüfen sollen. [FK04]

Die Hauptmerkmale lassen sich wie folgt zusammenfassen:
Dynamische Testverfahren ...

- benutzen konkrete Eingabewerte bei der Ausführung der Software
- sind innerhalb der realen Betriebsumgebung möglich
- haben Stichprobencharakter - Tests können nur Anwesenheit von Fehlern demonstrieren, nicht deren Abwesenheit
- können also nicht die Korrektheit der getesteten Software beweisen [Lig09]

Dynamische Tests lassen sich des Weiteren klassifizieren in *White-box-* und *Black-box-Verfahren.*

5.1.1 White-box-Verfahren

Aus dem Namen White-box-Verfahren leitet sich auch das wesentliche Merkmal dieser Unterart ab. Wenn die zu testende Applikation bildlich als Schachtel (Box) betrachtet wird, dann ist eine White-Box als transparente Schachtel zu verstehen. Der Inhalt dieser Schachtel ist also sichtbar. White-Box-Verfahren basieren darauf, das ,,Innenleben", also den Programmcode einer Applikation zu kennen, Testfälle aus diesem abzuleiten und während der Ausführung der Tests den Programmcode zu analysieren. Dabei besteht die Grundidee vor allem darin, alle Codefragmente mindestens einmal auszuführen, um alle möglichen Zustände der Applikation zu simulieren.

5.1.2 Black-box-Verfahren

Eine Black-Box kann sich bildlich als nicht durchsichtige Schachtel vorgestellt werden, deren Inhalt demzufolge unbekannt ist. Das beschreibt auch das Prinzip dieses Verfahrens, bei dem keine Kenntnisse über die inneren Abläufe einer Applikation nötig sind.
Testfälle müssen hier aus den Anforderungen bzw. der Spezifikation abgeleitet werden, wobei eine systematische Vorgehensweise zur sinnvollen Selektion von Eingabemöglichkeiten und deren Kombination notwendig ist. Diese Vorgehensweise folgt dem Prinzip, Eingabewerte zu sogenannten *Äquivalenzklassen* zusammen zu

fassen. Das heißt, für jede zu testende Eingabevariable (z.B. Methodenparameter oder Eingabefelder einer GUI), wird der Definitionsbereich ermittelt und alle darin liegenden Werte sowie alle außerhalb liegenden werden zu Mengen - den *Äquivalenzklassen* - zusammengefasst. Analog können Ausgabewerte zusammengefasst werden, sodass anschließend nicht alle Fälle einzeln, sondern jeweils nur ein Vertreter der jeweiligen Äquivalenzklassen getestet werden muss, um Redundanz zu vermeiden.
Ein weiteres wichtiges Merkmal von Black-box-Verfahren ist die *Grenzwertanalyse*. Sie beinhaltet das Prüfen von Eingabewerten an den Grenzen der Äquivalenzklassen, da an diesen Stellen oft Fehler auftreten.

5.2 QF-Test

Nachdem nun die theoretischen Grundlagen zum Testen des, mit dieser Bachelorarbeit verbundenen, graphischen Editors eingeführt wurden, wird nun die konkrete Realisierung der Tests erläutert.
Da bei einem graphischen Editor die Eingabe von Werten über eine GUI (Graphical User Interface, deut. graphische Benutzeroberfläche) erfolgt, spielt auch das Testen dieser die Hauptrolle. Der graphische Editor beruht auf GEF, welches wiederum auf SWT basiert, deshalb muss auch ein Testwerkzeug benutzt werden, welches SWT unterstützt. Nach einiger Recherche boten sich hier in erster Linie zwei Tools an: Das Open-Source-Tool *SWTBot*[6] und das kostenpflichtige *QF-Test*[7] der Firma Quality First Software (QFS), welche freundlicherweise für diese Bachelorarbeit eine befristete Lizenz bereitgestellt haben. Da das Projekt SWTBot jedoch noch in Entwicklung und nicht direkt auf GEF ausgelegt ist, war bereits die Einarbeitung und Konfiguration sehr komplex und damit eher demotivierend. Zusätzlich müssen alle Testfälle später von Hand programmiert werden, weshalb die ausführliche Betrachtung dieses Tools nicht für diese Bachelorarbeit infrage kam.
Anders hingegen verhielt sich die Erfahrung mit dem von QFS angebotenen Werkzeug. QF-Test lässt sich leicht installieren und war, zumindest für die Zwecke des zu testenden Editors, schnell zu konfigurieren. Das Programm gestaltet sich als

[6] SWTBot - http://www.eclipse.org/swtbot/
[7] QF-Test - http://www.qfs.de/de/qftest/index.html

eine komplett eigenständige Applikation, die unter anderem einen Schnellstart-Assistenten als einsteigerfreundliche Variante zum Erstellen von Testsuites bereitstellt. Nachdem das zu testende System, das *SUT* (System Under Test), festgelegt und einige wenige Einstellungen vorgenommen wurden (z.B. dass es sich um ein auf SWT basierendes SUT handelt), lassen sich bereits Testfälle anlegen.
Beim Beschreiben des Aufbaus von QF-Test werden im Folgenden bloß Elemente berücksichtigt, die für das Erstellen der Testfälle von Bedeutung sind, welche im Anschluss demonstriert werden. Für weitere Informationen stehen aber eine umfangreiche Dokumentation sowie Tutorials in Deutsch und Englisch auf der Website von QFS[8] zur Verfügung.

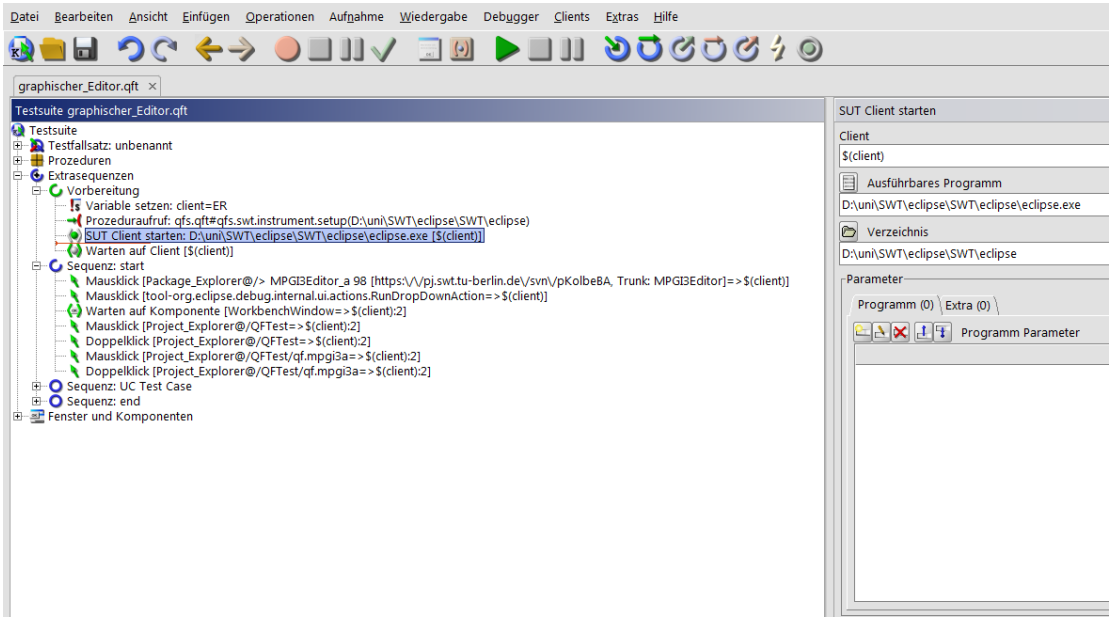

Abbildung 5.1: QF-Test-Oberfläche, markiert: Knoten, der die SUT startet

In Abbildung 5.1 ist die Oberfläche von QF-Test zu sehen, die sich hauptsächlich auf einen Baumeditor stützt, dessen verschiedene Knoten entweder Schritte eines Testfalls oder Elemente zu deren Unterstützung repräsentieren. Für die Erstellung erster Testfälle für den graphischen Editors ist im Wesentlichen nur der Knoten *Extrasequenzen* wichtig. Wie in der Abbildung zu sehen ist, wird hier das *SUT* gestartet und auch die Testfälle befinden sich hier in Form von *Sequenzen*. Eine Sequenz ist in diesem Falle die Aufzeichnung von nacheinander ablaufenden, be-

[8] Quality First Software - http://www.qfs.de/

nutzergesteuerten Ereignissen. Die Aufzeichnung einer Sequenz ist sehr einfach: Aufnahme starten, gewünschte Aktionen ausführen, die später automatisiert ablaufen sollen, Aufnahme stoppen. Die ausgeführten Aktionen werden als Ereignisse zu einer neuen Sequenz hinzugefügt, die wiederum unter den bereits erwähnten *Extrasequenzen* abgespeichert wird. Von dort aus können die Sequenzen nach Bedarf in einen *Testfallsatz* verschoben und dort verwendet werden.
In Abbildung 5.1 sind zum Beispiel die Ereignisse der *Sequenz: start* zu erkennen, in diesem Falle lediglich einige Mausklicks, die den graphischen Editor aus seiner Entwicklungsumgebung heraus starten. Die neue Eclipse-Instanz wird dabei von QF-Test als untergeordneter Prozess des SUT erkannt und kann somit wie ein eigenständiges SUT gehandhabt werden. Das heißt, es können nun automatisierte Testfälle für den graphischen Editor erstellt werden, indem diese einfach als Sequenzen aufgenommen werden. Dabei können im Nachhinein Ereignisse hinzugefügt, gelöscht, verschoben oder editiert werden. Des Weiteren können Sequenzen zur besseren Strukturierung zu einem *Testfall* zusammengefasst werden und Testfälle zu einem *Testfallsatz*. Testfallsätze können wiederum einem übergeordneten Testfallsatz zugeordnet werden. Anschließend können die aufgenommenen Tests einfach abgespielt und als automatisierte Tests verwendet werden. Zusätzlich werden für Testfallsätze die Knoten *Vorbereitung* und *Aufräumen* angeboten, welche Funktionen und Sequenzen beinhalten können, die vor bzw. nach den Testfällen ausgeführt werden. In Abbildung 5.2 sind zum Beispiel die Sequenzen *start* und *end* zu erkennen, welche dafür verantwortlich sind, vor dem Ausführen der Tests, eine Eclipse-Instanz mit dem graphischen Editor zu starten und nach den Tests, diese wieder zu schließen.
Im Folgenden wird ein konkretes Beispiel für die Automatisierung von Tests für den entwickelten graphischen Editor demonstriert. Das zu testende Objekt ist eine Klasse des KMGs, das heißt jegliche Aktionen, die mit einer KMG-Klasse verbunden sind, müssen überprüft werden. Die Auswahl der Testfälle beruht dabei vorrangig auf dem Programmcode, aber auch zum Teil auf er Spezifikation, also den gegebenen Anforderungen. Somit handelt sich um *Whitebox-Tests*, die alle möglichen Zustände einer Klasse des KMGs herstellen sollen.
Das bedeutet im Folgenden müssen diverse Tests in Bezug auf das Erstellen, Verschieben, Umbenennen und Löschen von Klassen des KMGs erstellt werden sowie Tests, welche die korrekte Synchronisation mit Klassen des SKMs und Akteuren

Abbildung 5.2: Testfallsatz für das KMG, beinhaltet z.B. verschiedene Tests für Klassen

des UCDs überprüfen. Zusätzlich können gleichzeitig für alle ausgeführten Aktionen die Funktionen *,,Rückgängig machen (Undo)"* und *,,Wiederherstellen (Redo)"* überprüft werden sowie auch die stetige Konsistenz des Modells, indem diese Funktionen nach jedem Schritt eine Tests ausgeführt werden und danach abgespeichert wird. Ein Modell wird zum Beispiel inkonsistent, wenn es Modellelemente enthält, die nicht korrekt gelöscht wurden, d.h. wenn immer noch Referenzen auf Elemente existieren, die nicht mehr im Modell vorliegen dürften. EMF überprüft dies beim Speichern und gibt ggf. Fehlermeldungen aus. Da sich die Testfälle jeweils aus Abfolgen von Ereignissen zusammensetzen, werden zum Abschluss einige für das gegebene Beispiel beschrieben (siehe Abbildung 5.2):

- Erstellen, Umbenennen, Verschieben, Löschen:

  Verschieben

  – KMG-Klasse A erstellen → KMG-Klasse A umbenennen in B → KMG-Klasse B verschieben → KMG-Klasse B löschen

  Umbenennen - Leerer Name

  – KMG-Klasse A erstellen → KMG-Klasse A umbenennen in *' '* (keine leeren Namen zulässig) → alten Namen Bestätigen für KMG-Klasse A → KMG-Klasse A löschen

Umbenennen - Doppelter Name

- KMG-Klasse A erstellen → KMG-Klasse B erstellen → KMG-Klasse A in B umbenennen (keine doppelten Namen zulässig) → alten Namen Bestätigen für KMG-Klasse A → KMG-Klassen A und B löschen

- Synchronisation mit UCD-Akteuren:

  Umbenennen des korrespondierenden UC-Akteurs

  - KMG-Klasse A erstellen → rechter Mausklick auf KMG-Klasse A → Kontextmenü-Eintrag "Create UCActor" wählen (Mit KMG-Klasse A verknüpfter Akteur wird im UCD angelegt) → UCD-Akteur A in B umbenennen (KMG-Klasse A sollte auch umbenannt werden) → UCD-Akteur B löschen → Disconnect-Dialog bestätigen (UCD-Akteur B und KMG-Klasse B sollten entkoppelt werden) → KMG-Klasse B löschen

  Umbenennen der korrespondierenden KMG-Klasse

  - KMG-Klasse A erstellen → rechter Mausklick auf KMG-Klasse A → Kontextmenu-Eintrag "Create UCActor" wählen → KMG-Klasse A in B umbenennen (UCD-Akteur A sollte auch umbenannt werden) → KMG-Klasse B löschen → Disconnect-Dialog bestätigen (UCD-Akteur B und KMG-Klasse B sollten entkoppelt werden) → UCD-Akteur B löschen

  Verbinden entkoppelter Elemente

  - KMG-Klasse A erstellen → rechter Mausklick auf KMG-Klasse A → Kontextmenü-Eintrag "Create UCActor" wählen → KMG-Klasse A löschen → Disconnect-Dialog bestätigen → KMG-Klasse A erstellen → rechter Mausklick auf KMG-Klasse A → Kontextmenü-Eintrag "Connect with disconnected UCActor" wählen → KMG-Klasse A in B umbenennen (UCD-Akteur A sollte auch umbenannt werden) → KMG-Klasse B löschen → Disconnect-Dialog bestätigen → UCD-Akteur B löschen

- Synchronisation mit SKM-Klassen:

  Umbenennen der korrespondierenden SKMEntity

  – KMG-Klasse A erstellen (sollte damit auch als SKM-Entity A erstellt werden) → SKM-Entity A umbenennen in B (KMG-Klasse A sollte ebenfalls in B umbenannt werden) → SKM-Entity B löschen (KMG-Klasse B sollte nicht gelöscht werden) → KMG-Klasse B löschen

  Doppelter Name + Verbinden entkoppelter Elemente #1

  – SKM-Entity A erstellen → KMG-Klasse A erstellen (nicht erlaubt, da A schon im SKM existiert) → B als neuen Namen der KMG-Klasse wählen (SKM-Entity B sollte auch angelegt werden) → SKM-Entity B löschen → SKM-Entity A in B umbenennen → Sychronisationsdialog bestätigen (SKM-Entity A mit KMG-Klasse B verbinden) → KMG-Klasse B löschen (auch SKM-Entity B sollte danach gelöscht sein)

  Verbinden entkoppelter Elemente #2

  – KMG-Klasse A erstellen → SKM-Entity B löschen → SKM-Entity B erstellen → rechter Mausklick auf SKM-Entity B → Kontextmenü-Eintrag "Connect with KmgNode" wählen → KMG-Klasse A auswählen (Verbindung zwischen SKM-Entity B und KMG-Klasse A sollte hergestellt, SKM-Entity B automatisch in A umbenannt werden) → KMG-Klasse A löschen (auch SKM-Entity A sollte danach gelöscht sein)

6 Schlussbetrachtung

Die Konzeption und Implementierung eines graphischen Editors für verschiedene Notationen im Softwareentwicklungsprozess als Thema dieser Bachelorarbeit birgt vor allem die Frage in sich, warum bei der Fülle an bestehenden UML-Editoren die Notwendigkeit besteht, einen weiteren zu entwerfen.
Deshalb war es besonders wichtig zu zeigen, wie ein graphischer Editor einen Softwareentwicklungsprozess über die Eigenschaften existierender Lösungen hinaus sinnvoll unterstützen kann.
Dafür wurde zunächst ein weit verbreiteter und auf visueller Modellierung gestützter Softwareentwicklungsprozess eingeführt, der Rational Unified Process. Ziel des Kapitels **RUP** war es, die Grundlagen dieses Prozesses zu vermitteln, aber auch dessen immense Komplexität anzudeuten.
Da der Umfang von RUP zu komplex ist, um für jede Anpassung sinnvolle Unterstützungsmöglichkeiten zu bieten, muss zunächst eine genaue Spezifizierung vorliegen. Das heißt es müssen klare Anforderungen festgesetzt werden und zwar in Form einer Anpassung von RUP an ein bestimmtes Projekt bzw. eine bestimmte Verfahrensweise, die mitunter auch eine in der Lehre eingesetzte Methodik sein kann.
Solch eine Anpassung wurde deshalb im Kapitel **Beschreibung der Methodik** vorgestellt sowie Unterstützungsmöglichkeiten, die unter Berücksichtigung von genormten Usability-Kriterien den Entwurf des graphischen Editors begründen.
Als wesentliche Konzepte sind dabei *Synchronisation* und *Konsistenzchecks* hervorgegangen, die zum einen Arbeit durch Automatisierungen ersparen und zum anderen gewisse Fehlerquellen beim Anwenden der Methodik ausschließen.
Aufgrund der iterativen Vorgehensweise von RUP gestaltet sich aber gerade die Umsetzung von Synchronisationen an manchen Stellen als schwierig, da genau bedacht werden muss, wie und in welche Richtungen diese wirken sollen. Dies nimmt mitunter eine derartige Komplexität an, dass mögliche Umsetzungen schwer nach-

vollziehbar und damit ungeeignet im Sinne der Benutzerfreundlichkeit sind.
Ein weiteres Problem, dass sich durch Synchronisationen ergibt, ist das Entstehen inkonsistenter Zustände innerhalb eines Modells, durch die Einwirkung von Elementen anderer Modelle. Inkonsistente Zustände dürfen also nicht immer ausgeschlossen werden, sondern müssen ggf. stattdessen für den Benutzer kenntlich gemacht werden. Außerdem müssen Möglichkeiten zur Wiederherstellung solcher Zustände angeboten werden.
Welche Synchronisationen letztlich umgesetzt wurden geht aus dem Kapitel **Implementierung** hervor. Dieses dient vor allem der Wartung und Wiederverwendbarkeit des Editors, da die Metamodelle als Grundlage der Implementierung ausführlich beschrieben wurden.
Auch das **Test**-Kapitel ist für die Wartung und Weiterentwicklung des Editors hilfreich. Die beschriebene Software von QFS ist zwar kostenpflichtig, allerdings zeichnet sie sich wahrscheinlich nicht zuletzt deshalb durch ihre leichte Bedienbarkeit sowie eine reichhaltige Dokumentation aus, im Gegensatz zur erwähnten Open-Source-Variante SWTBot.

Abschließend sei gesagt, dass der konzipierte und implementierte graphische Editor den Benutzer beim Anwenden der beschriebenen Methodik aus Kapitel 3 bereits sehr umfassend unterstützt, sodass beim Einsatz in der Lehre sogar abgewägt werden sollte, ob eine Versionierung didaktisch sinnvoll wäre. So könnte die vollständige Variante zur Erstellung von Aufgaben für den Fachbereich herausgegeben werden, während den Studenten eine reduzierte Variante zur Verfügung gestellt wird.

Literaturverzeichnis

[Akk09] Akkreditierungsstelle, DATECH Deutsche: *Leitfaden Usability*, 2009.

[Ari10] Arinir, P.Z.D.: *Java: Nebenl äufige & verteilte Programmierung*. W3l GmbH, 2010.

[Bal10] Balzert, H.: *Java: objektorientiert programmieren: Vom objektorientierten Analysemodell bis zum objektorientierten Programm*. W3L GmbH, 2010.

[BM04] Bill Moore, David Dean, Anna Gerber Gunnar Wagenknecht Philippe Vanderheyden: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, 2004.

[DB04] Dirk Bäumer, Daniel Megert, André Weinand: *ECLIPSE PLU-INS, ENTWICKELN UND PUBLIZIEREN*, 2004.

[DISJ09] Dr.-Ing. Stefan Jähnichen, Georgy Dobrev, Alexandra Mehlhase Alexander Rein: *MPGI 3 - Softwaretechnik*, 2009.

[EM07] Essigkrug, A. und T. Mey: *Rational Unified Process kompakt*. Sav Informatik. Spektrum Akademischer Verlag, 2007.

[FK04] Forbrig, P. und I.O. Kerner: *Lehr- und Übungsbuch Softwareentwicklung*. Fachbuchverl. Leipzig im Carl Hanser Verl., 2004.

[Fou10] Foundation, Eclipse: *Eclipse Documentation - Current Release (Eclipse Helios)*, 2010.

[Haa04] Haas, R.: *Usability Engineering in der E-collaboration: ein managementorientierter Ansatz f*
ür virtuelle Teams. Dt. Univ.-Verl., 2004.

[Kru99] Kruchten, P.: *Der rational Unified Process: Eine Einf*
ührung. Addison-Wesley, 1999.

[Lig09] Liggesmeyer, P.: *Software-Qualit*
ät: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, 2009.

[SBD+10] Schatten, A., S. Biffl, M. Demolsky, E. Gostischa-Franta, T. A-Streicher und D. Winkler: *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Spektrum Akademischer Verlag, 2010.

[SBPM09] Steinberg, D., F. Budinsky, M. Paternostro und E. Merks: *EMF: Eclipse Modeling Framework*. Eclipse series. Addison-Wesley, 2009.

[Sie03] Siedersleben, J.: *Softwaretechnik: Praxiswissen f*
ür Softwareingenieure. Hanser, 2003.

[SVEH07] Stahl, T., M. V
ölter, S. Efftinge und A. Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt.Verlag GmbH, 2007.

[TM09] Tony Modica, Enrico Biermann, Claudia Ermel: *An ECLIPSE Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models*, 2009.

[Vö02] Völkel, Marcus: *Kein Flow ohne Content Usability.*, 2002.

Abbildungsverzeichnis