ulm university universität

DIPLOMARBEIT

# Traceability von Requirements in einer Model Driven Architecture

## Untersuchung der Traceability und Entwicklung eines Eclipse-Prototyps für die ActiveCharts IDE


Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau

vorgelegt von

Jan Scheible


am 16. Juli 2007

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Helmuth Partsch |
| Zweitgutachter: | Prof. Dr. Franz Schweiggert |
| Betreuer: | Dipl.-Inf. Armin Bolz |

# Abstract

Die Verwendung eines MDA-Ansatzes schreibt den UML-Modellen einen anderen Stellenwert als die herkömmliche Softwareentwicklung zu. Bei der herkömmlichen Softwareentwicklung dienen die Modelle in der Design-Phase zur Dokumentation des Entwurfs der Software. Diese Modelle werden in der nächsten Phase, der Implementierungs-Phase, von Hand in Quellcode umgesetzt. Bei einer MDA hingegen findet die Umsetzung der Modelle automatisiert statt. Daher ist es bei einem MDA-Ansatz von großem Interesse, in welchen Modellen die Anforderungen umgesetzt werden, da diese gleichzeitig die Realisierung darstellen.

In dieser Arbeit wird untersucht, wie eine Traceability zwischen den Anforderungen und den Modellen bei einer Entwicklung mit der ActiveCharts IDE der Universität Ulm realisiert werden kann. Die Grundlage der Untersuchung bilden zwei Praktika, in denen die ActiveCharts IDE eingesetzt wurde.

Es wird zuerst das Vorgehen aus den Praktika in einem einfachen Wasserfallmodell formalisiert. Das dient dazu, die in der Entwicklung vorkommenden Artefakte zu identifizieren. Im nächsten Schritt wird eine Struktur für die textuellen Artefakte entwickelt. Dadurch wird erreicht, dass die einzelnen Elemente eine Semantik bekommen, obwohl natürliche Sprache verwendet wird. Diese Semantik wird bei der Untersuchung der Traceability verwendet. Denn wenn die Semantik bekannt ist, kann teilweise eine automatische Traceability erfolgen. Zur Realisierung einer übergreifenden Traceability wird eine allgemeine Traceability-Architektur entwickelt. Des Weiteren wird noch auf mögliche Konsistenzprüfungen zwischen den Anforderungen und den Modellen eingegangen.

Die erarbeiteten Ergebnisse bilden die Grundlage für einen Prototypen, welcher als zwei Eclipse-Plug-ins für die ActiveCharts IDE realisiert wurde. Das erste Plug-In enthält den Requirements Editor, der das grafische Editieren der Anforderungen erlaubt. Dabei sind die Anforderungen in der zuvor entwickelten Struktur organisiert. Zur Darstellung wurde eine SWT-Komponente entwickelt, welche eine Bedienbarkeit vergleichbar mit einer Textverarbeitung aufweist. Die Anforderungen lassen sich in dem standardisierten RIF-Format exportieren, was einen Datenaustausch mit anderen Requirements-Tools ermöglicht. Schließlich wurde ein Change Log-Plug-in entwickelt, welches die im Theorieteil konzipierte Traceability-Architektur umsetzt. Dadurch wird die Basis für eine Traceability zwischen allen Artefakten der ActiveCharts IDE gelegt.

# Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die direkt oder indirekt an dieser Arbeit mitgewirkt haben. Besonders bedanken möchte ich mich bei:

- Herrn Prof. Partsch, der mir das Thema ermöglicht hat
- Meinem Betreuer Armin Bolz für die gute Betreuung und hilfreichen Tipps
- Meiner Freundin Silke für ihr frühes Feedback, ihr Korrekturlesen und ihre große Geduld
- Meiner Mutter, die mir eine sehr große Hilfe bei den Formulierungen, der Grammatik und der Beseitigung von Tippfehlern war
- Jens Kolb für seine vielen wertvollen Tipps, obwohl er mit sehr frühen Versionen der Arbeit konfrontiert war
- Lisa Daske für ihre Grammatik- und Sprachtipps
- Meinen Eltern und meinem Großvater, der das Ende der Arbeit leider nicht mehr erleben konnte. Ohne sie wäre mein Studium nicht möglich gewesen.

# Inhaltsverzeichnis


**1 Einführung** **1**

1.1 Motivation . . . 1
1.2 Problemstellung . . . 3
1.3 Zielsetzung . . . 3
1.4 Aufbau . . . 4

**2 Grundlagen** **5**

2.1 Vorgehensmodelle . . . 5
2.2 Requirements Engineering . . . 6
2.2.1 Traceability . . . 9
2.3 Modell Driven Architecture . . . 10
2.3.1 ActiveCharts IDE . . . 12
2.4 Eclipse . . . 15
2.5 Requirements Interchange Format . . . 16
2.5.1 Castor . . . 19

**3 Theoretischer Teil** **21**

3.1 Vorgehensmodell zur Entwicklung mit der ActiveCharts IDE . . . 21
3.1.1 Entwicklungsprozess aus den Praktika . . . 21
3.1.1.1 Design-Phase . . . 22
3.1.1.2 Review- und Verbesserungs-Phase . . . 23
3.1.1.3 Implementierungs-Phase . . . 24
3.1.1.4 Test-Phase . . . 25
3.1.2 Abgeleitetes Vorgehensmodell . . . 26
3.2 Struktur der Anforderungsdokumente . . . 28
3.2.1 Anforderungen an ein Anforderungsdokument . . . 28
3.2.2 Struktur für ein Anforderungsdokument . . . 29
3.2.2.1 Stakeholder . . . 31
3.2.2.2 Anforderung . . . 31
3.2.2.3 Glossareintrag . . . 31
3.2.2.4 Use Case . . . 32
3.2.2.5 Signal . . . 32
3.2.2.6 Hardware-Schnittstelle . . . 33

3.2.2.7 Hardware-Komponente . . . 33
3.2.2.8 Testfall . . . 34
3.2.3 Erfüllung der Anforderungen . . . 34
3.3 Traceability . . . 35
3.3.1 Motivation . . . 35
3.3.2 Traceability-Architektur . . . 36
3.3.2.1 Links . . . 37
3.3.2.2 Anforderungs-Struktur . . . 38
3.3.2.3 Beliebige Artefakte . . . 38
3.3.2.4 Änderungs-Log . . . 40
3.3.2.5 Fazit . . . 40
3.3.3 Arten der Verlinkung . . . 41
3.3.3.1 Manuell erzeugte Links . . . 41
3.3.3.2 Automatisch erzeugte Links . . . 42
3.3.4 Konsistenzprüfungen . . . 45
3.3.5 Fazit . . . 46
3.3.6 Ausblick . . . 46
3.4 Toolunterstützung . . . 47
3.4.1 Requirements-Tool . . . 47
3.4.2 Traceability-Tools . . . 48
3.4.3 Fazit . . . 49

**4 Praktischer Teil** **51**
4.1 Anforderungen an das Tool . . . 52
4.2 Requirements Editor-Plug-in . . . 52
4.2.1 Anforderungs-Modell . . . 56
4.2.2 Darstellung der Elemente . . . 58
4.2.2.1 Dokumenten-Modell . . . 61
4.2.3 RIF-Import und -Export des Modells . . . 64
4.2.3.1 Transformation . . . 66
4.2.3.2 Marshalling . . . 67
4.2.3.3 Bereinigung . . . 67
4.2.3.4 Anreicherung mit Typinformationen . . . 68
4.2.3.5 Unmarshalling . . . 68
4.3 Change Log-Plug-in . . . 69
4.4 Erweiterungsmöglichkeiten . . . 70
4.4.1 Requirements Editor . . . 70
4.4.2 Änderungs-Log . . . 72
4.5 Stand der Implementierung . . . 73
4.6 Fazit . . . 74

4.6.1 Erfüllung der Anforderungen . . . . . 75
4.6.2 Bewertung . . . . . 75

**5 Zusammenfassung und Ausblick** **77**
5.1 Zusammenfassung . . . . . 77
5.2 Ausblick . . . . . 78

**A Beschreibung der GUI** **80**
A.1 Erstellung eines neuen Dokuments . . . . . 80
A.2 Elemente hinzufügen/löschen . . . . . 81
A.3 Link/LinkTarget einfügen . . . . . 81
A.4 Internen Link erstellen . . . . . 82
A.5 Externen Link erstellen . . . . . 82
A.6 Links löschen . . . . . 82
A.7 Anforderungen in das RIF-Format exportieren . . . . . 82
A.8 Ungültige Links löschen . . . . . 83

**B Bekannte Probleme** **85**
B.1 Fehlende Unterscheidung zwischen dem Löschen eines Elements und eines Drag & Drop-Vorgangs . . . . . 85
B.2 Bestehende Abhängigkeit zwischen dem Requirements Editor-Plug-in und dem Graphical Editing Framework-Plug-in . . . . . 85
B.3 Dezentrale Speicherung der Links . . . . . 86
B.4 Änderung eines Links wird nicht erkannt . . . . . 86
B.5 TextFlows können von innen gelöscht werden . . . . . 86
B.6 Markierung eines Links im Requirements Editor . . . . . 87
B.7 Interaktive Veränderung der Breite einer Tabellenspalte . . . . . 87
B.8 Layout des Requirements Editor wird manchmal nicht aktualisiert . . . . . 87
B.9 Im Requirements Editor verschobene Links werden im Anforderungs-Modell nicht mitverschoben . . . . . 88

**C Inhalt der CD** **89**
C.1 Ordnerstruktur . . . . . 89
C.2 Voraussetzung . . . . . 89
C.3 Quellcodeübersicht . . . . . 90

**Literaturverzeichnis** **91**

# 1 Einführung

## 1.1 Motivation

Softwareprojekte werden heutzutage immer komplexer und größer. So besteht z. B. die Software R/3 von SAP bereits aus mehr als 35.000.000 Zeilen Quellcode (Quelle: [38]). Sie erstrecken sich längst nicht mehr über einzelne Computer und Abteilungen, sondern über ganze virtuelle Unternehmen, die sich aus mehreren einzelnen Firmen zusammensetzen. Hierbei stellt die Integration der verschiedenen Technologien der Unternehmen eine große Herausforderung dar. Gleichzeitig wächst der Zeit- und Gelddruck auf die Softwarehersteller, dem sie sich stellen müssen, um konkurrenzfähig zu sein und zu bleiben.

Im Folgenden werden zwei Ansätze zur Handhabung der oben genannten Probleme vorgestellt. Zuerst wird die modellgetriebene Architektur (MDA; engl. Model Driven Architecture, MDA) vorgestellt, die durch eine klare Trennung von Funktionalität und Technik kürzere Entwicklungszeiten und eine größere Unabhängigkeit von der verwendeten Technologie zu erreichen versucht. Als zweites wird ein kurzer Überblick über das Requirements Engineering gegeben. Es wird versucht, durch eine ingenieurmäßige Erhebung der Anforderungen und einer Verfolgbarkeit der Anforderungen in die späteren Phasen, genau die vom Kunden gewünschte Software zu produzieren. Dies soll die Einhaltung sowohl der Budget- als auch der Zeitvorgaben ermöglichen.

### Model Driven Architecture

Technologien wie die objektorientierte Programmierung haben die Abstraktionsebene der Programmierung im Vergleich zur prozeduralen Programmierung stark erhöht. Allerdings konnten sie die Anforderungen in Bezug auf den Aufwand, der zur Programmierung nötig ist, und die Wiederverwendbarkeit von einzelnen Komponenten nicht erfüllen. Des Weiteren löst auch die Verwendung objektorientierter Techniken nicht das Problem, dass bei einem Wechsel der verwendeten Technologie ein sehr hoher Änderungsaufwand zu betreiben ist, der im schlechtesten Fall nahe an eine Neuimplementierung der Software herankommt.

Durch den Ansatz einer MDA, welcher von der Object Management Group (OMG) [22] propagiert wird, wird versucht, diese Probleme zu lösen. Dazu wird die Abstraktionsebene der Entwicklung noch höher angesetzt als bei der objektorientierten Entwicklung. Dies wird durch die Verwendung eines Modells erreicht, das den Ausgangspunkt der Softwareentwicklung dar-

stellt. Dieses Modell stellt eine abstrakte Repräsentation der zu entwickelnden Software dar. Zur Definition des Modells kann z. B. die UML [24] dienen. Durch Modelltransformationen wird dieses Modell, u. U. in mehreren Schritten, zuerst in Quellcode und schließlich in ein lauffähiges Programm umgewandelt.

Die Verwendung einer MDA kann sowohl Kosten als auch Zeit sparen, wie folgende Beispiele zeigen:

1. Die DaimlerChrysler TSS GmbH, welche ein Tochterunternehmen der DaimlerChrysler AG ist, erreichte durch die Verwendung eines MDA-Ansatzes im ersten Jahr nach der Einführung eine Steigerung der Entwicklungsproduktivität um 15%. Ein Return on Investment (ROI; dt. Kapitalrendite oder Gesamtkapitalrentabilität) von weniger als 12 Monaten wurde erreicht und eine weitere Produktivitätssteigerung auf 30% im zweiten Jahr wurde erwartet (Quelle: [8]).
2. Die Deutsche Bank Bauspar AG hat durch die Verwendung eines MDA-Ansatzes bei der Entwicklung einer webbasierten Anwendung zur Verwaltung von Spar- und Darlehenskonten 40% des Entwicklungsaufwands, im Vergleich zum geschätzten Aufwand bei einer Entwicklung ohne einen MDA-Ansatz, eingespart (Quelle: [9]).

Die ActiveCharts IDE des Instituts für Programmiermethodik und Compilerbau der Universität Ulm ist ein MDA-Ansatz, in dem die Modelle aus der Analyse-/Design-Phase mit dem Code aus der Implementierungs-Phase verknüpft werden. Zur Modellierung des Kontrollflusses werden UML-Aktivitätsdiagramme verwendet. Ein Interpreter führt diese Diagramme aus und erlaubt es, in den Aktionen Code auszuführen. Durch die Verwendung eines Interpreters wird eine Plattformunabhängigkeit erreicht. Nur der in den Aktionen aufgerufene Code muss bei einem Wechsel der Plattform angepasst werden.

### Requirements Engineering

Partsch beschreibt in [26, Seite 17] das Requirements Engineering als „eine Disziplin zur systematischen Entwicklung einer vollständigen, konsistenten und eindeutigen Spezifikation, in der beschrieben wird, was ein softwaregestütztes Produkt tun soll (aber nicht wie), und die als Grundlage für Vereinbarungen zwischen allen Betroffenen dienen kann.“

Unterstützt wird das Requirements Engineering durch die Traceability (dt. Verfolgbarkeit). Die Traceability beantwortet die Frage, woher eine Anforderung kommt und wo eine Anforderung umgesetzt wird. Sie sorgt also dafür, dass der Grund für die Existenz einer Anforderung und der Ort ihrer Realisierung bekannt sind.

Die Notwendigkeit einer gründlichen Anforderungsanalyse wird durch Abbildung 1.1 verdeutlicht. Laut [27] haben 57% aller Fehler in Softwaresystemen ihren Ursprung im mangelhaftem Requirements Engineering.

Somit besteht ein großes Verbesserungspotential durch sorgfältig durchgeführtes Require-

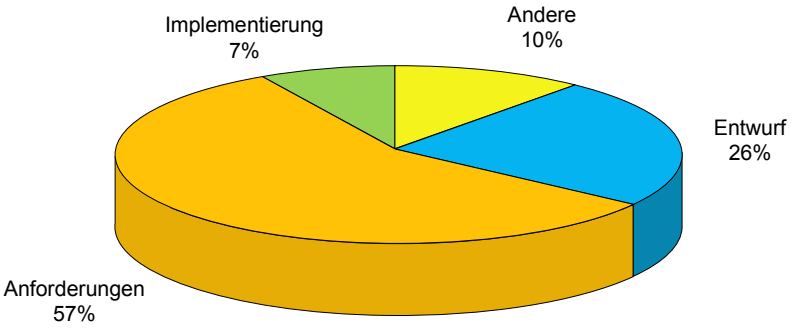



Abbildung 1.1: Ursprung von Fehlern in Softwaresystemen (Quelle: [27])

ments Engineering und Traceability, denn nur wenn weniger Fehler auftreten, ist es möglich, die Budget- und Zeitvorgaben einzuhalten.

## 1.2 Problemstellung

Die ActiveCharts IDE eignet sich sehr gut für die Modellierung einer Steuersoftware eines eingebetteten Systems. Daher wurde sie auch in dem Praktikum „Montage und Simulation von Anwendungen unter Verwendung von UML 2“ an der Universität Ulm im Sommersemester 2005 zur Modellierung der Steuerung einer Mikrowelle und im Sommersemester 2006 eines Aufzugs verwendet. Da bei der Entwicklung mit der ActiveCharts IDE die Modelle aus der Analyse-/Design-Phase auch gleichzeitig Artefakte der Implementierung sind, müssen die Anforderungen aus den Anforderungsdokumenten in den Modellen realisiert werden.

Allerdings haben weder die Anforderungen eine feste Struktur, noch besteht eine explizite Verbindung zwischen den Anforderungen und den Modellen.

## 1.3 Zielsetzung

Die Zielsetzung der Diplomarbeit ist, zuerst eine Struktur für die Anforderungen zu schaffen, mit Hilfe derer dafür gesorgt werden kann, dass die Anforderungen konsistent notiert werden können. Diese Anforderungen sollen dann mit den Modellen der ActiveCharts IDE verlinkt werden. Durch diese Links soll eine weniger fehleranfällige Entwicklung ermöglicht werden, bei der geprüft werden kann, ob alle Anforderungen realisiert wurden und ob Inkonsistenzen zwischen den Anforderungen bzw. der Spezifikation und den Modellen vorhanden sind.

Des Weiteren soll untersucht werden, welche Verlinkungen zwischen den Anforderungen und dem Modell möglich sind, an welche Elemente man welche Links hängen kann und welche automatischen Konsistenzprüfungen möglich sind, um eine Einhaltung der Anforderungen zu unterstützen.

Außerdem soll ein Prototyp eines Anforderungseditors als Eclipse-Plug-in realisiert werden, der die Verlinkung von Elementen in den Anforderungen mit Elementen in den Modellen unterstützt. Um den Datenaustausch mit anderen Requirements-Tools zu ermöglichen, soll der Anforderungseditor die Anforderungsdokumente im RIF-Format, einem standardisierten XML-Format, zum Austausch von Anforderungen exportieren können.

Dadurch werden die Vorteile einer MDA mit den Vorteilen des Requirements Engineering und einer Möglichkeit zur Traceability verknüpft. Dies erlaubt eine höhere Softwarequalität und eine höhere Entwicklungsgeschwindigkeit, da weniger Inkonsistenzen zwischen den Anforderungen und den Modellen auftreten.

## 1.4 Aufbau

In Kapitel 2 der Ausarbeitung werden die Grundlagen für diese Arbeit erläutert. Der Hauptteil der Arbeit lässt sich in einen theoretischen und praktischen Teil gliedern.

Kapitel 3 enthält den theoretischen Teil. In Abschnitt 3.1 wird das Vorgehen aus dem Praktikum in einem einfachen Vorgehensmodell formalisiert und erläutert, welche Artefakte in dem Vorgehensmodell eine Rolle spielen. Diese Artefakte sind in erster Linie Anforderungsdokumente, deren Aufbau und Inhalt in Abschnitt 3.2 beschrieben wird. Abschnitt 3.3 stellt die Untersuchung der Traceability dar. In Abschnitt 3.4 wird schließlich ein kurzer Überblick über zwei kommerzielle Tools gegeben.

Kapitel 4 enthält den praktischen Teil. In Abschnitt 4.1 werden zuerst die Anforderungen an den Prototypen definiert. Der Prototyp besteht aus zwei Eclipse-Plug-ins, welche in Abschnitt 4.2 und 4.3 vorgestellt werden. Abschnitt 4.2.3 beschreibt den Import- und Export im RIF-Format. Danach werden in Abschnitt 4.4 die Erweiterungsmöglichkeiten der Plug-ins beschrieben und schließlich wird in Abschnitt 4.5 ein Überblick über den Stand der Implementierung des Prototyps gegeben.

Die Arbeit wird schließlich in Kapitel 5 mit einer Zusammenfassung und einem Ausblick abgeschlossen.

In Anhang A wird ein Überblick über die GUI gegeben, in Anhang B werden bekannte Probleme des Prototypen aufgelistet und in Anhang C wird der Inhalt der beigelegten CD beschrieben.

# 2 Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit erklärt. Zuerst wird in Abschnitt 2.1 das Konzept des Vorgehensmodells und in Abschnitt 2.2 die Grundlagen des Requirements Engineerings vorgestellt. Dann folgt in Abschnitt 2.3 ein Überblick über MDAs im allgemeinen und der ActiveCharts IDE im speziellen. Als nächstes wird in 2.4 Eclipse und seine Plug-in-Mechanismen vorgestellt. Schließlich wird in Abschnitt 2.5 auf das RIF-Format und Castor, eine in Java geschriebene XML-Bibliothek, eingegangen.

## 2.1 Vorgehensmodelle

Ein Vorgehensmodell teilt den Entwicklungsprozess einer Software in kleinere, überschaubarere Phasen auf. Der Grund dafür ist, dass die große Komplexität der Entwicklung einer Software handhabbar gemacht werden soll. Jede Phase besteht wiederum aus einzelnen Schritten. Am Ende jeder Phase kann ein Meilenstein stehen, welcher eine Menge von Kriterien vorgibt. Ein Meilenstein ist erreicht, wenn die vorgegebenen Kriterien erfüllt wurden. Eine Phase ist fertiggestellt, wenn der jeweilige Meilenstein erreicht ist, bzw. das jeweilige Ergebnis fertiggestellt ist. Als Artefakte werden jede Art von Ergebnis oder Produkt bezeichnet, die im Rahmen des Entwicklungsprozesses entstehen.

Ein Beispiel für die Aufteilung in Phasen ist ein einfaches Wasserfallmodell (siehe Abbildung 2.1). Bei einem einfachen Wasserfallmodell gibt es keine Meilensteine, es ist aktivitätenorientiert. Aktivitätenorientiert bedeutet, dass eine Phase beendet ist, wenn die Aufgabe der jeweiligen Phase erledigt wurde. Es werden also keine Meilensteine oder zu produzierende Artefakte festgelegt. Im Folgenden werden die einzelnen Phasen kurz erklärt.

1. **Analyse-Phase**
   In der Analyse-Phase wird die Aufgabenstellung präzisiert. Das Ziel ist die exakte Ermittlung der Kundenwünsche und das Finden und Notieren der Anforderungen an das gewünschte System. In dieser Phase steht das „WAS“ im Vordergrund.
2. **Design-Phase**
   In der Design-Phase wird die grundlegende Architektur des Systems entworfen. Das System wird dann in Komponenten aufgeteilt und deren Zusammenspiel dokumentiert. Das betrifft sowohl Hardware- als auch Software-Komponenten. In dieser Phase steht

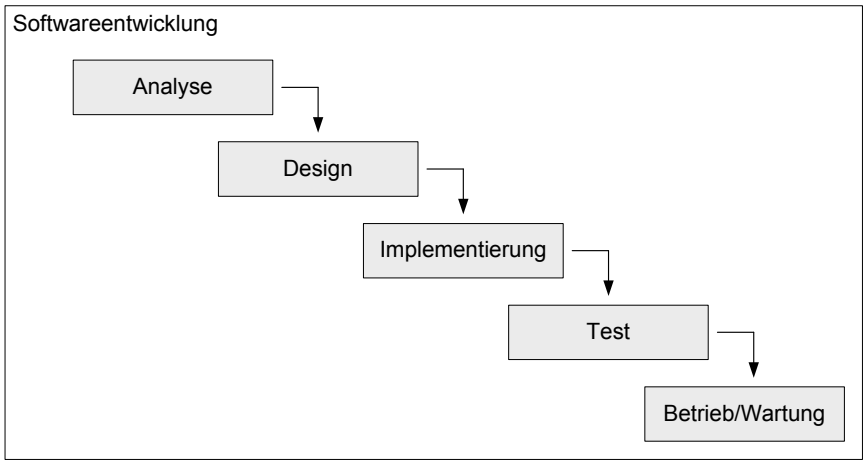



Abbildung 2.1: Einfaches Wasserfallmodell

das „WIE“ im Vordergrund.

3. **Implementierungs-Phase**
   In der Implementierungs-Phase wird der Entwurf aus der Design-Phase umgesetzt. Es entsteht ein lauffähiges Programm.

4. **Test-Phase**
   In der Test-Phase werden sowohl die einzelnen Komponenten als auch das Gesamtsystem getestet, welches sich aus den einzelnen Komponenten zusammensetzt.

5. **Betriebs-/Wartungs-Phase**
   Diese Phase findet nach dem Abschluss der eigentlichen Entwicklung der Software statt. Sie umfasst die Beseitigung auftretender Fehler und Optimierungen an dem eingesetzten System.

Das einfache Wasserfallmodell ist ein stark vereinfachtes Phasenmodell, da es idealisiert und in der beschriebenen Form nicht einsetzbar ist. So würde sich ein am Anfang in der Analyse-Phase gemachter Fehler durch alle Phase ziehen. Die Beseitigung eines solchen Fehlers wäre extrem aufwändig, da er aufgrund der hierarchischen Verfeinerung in vielen Teilen der Software zu beseitigen wäre. Daher ist eine Rückkopplung nach den einzelnen Phasen nötig oder die Prüfmaßnahmen werden explizit verankert, wie es z. B. bei dem V-Modell der Fall ist. Weitere Details zu Vorgehensmodellen siehe [26].

## 2.2 Requirements Engineering

Das Requirements Engineering stellt die erste Phase (siehe vorherigen Abschnitt) des Softwareentwicklungsprozesses dar. Es beschäftigt sich mit dem strukturierten Finden und Dokumentieren von Anforderungen und stellt die Brücke zwischen der Welt des Kunden und der Welt der Softwareentwickler dar. Dies umfasst z. B. die Erarbeitung eines gemeinsamen

Vokabulars und eines Verständnisses für die Problemstellung des Kunden. Es geht also zuerst um ein inhaltliches Verständnis der Problemstellung.

Abbildung 2.2 zeigt einen Überblick über einen generischen Requirements Engineering-Prozess. Im Fokus dieser Arbeit stehen die orange markierten Tätigkeiten. Die nicht markierten Tätigkeiten treten in den Hintergrund, da im Praktikum nur die Entwicklung des Systems im Mittelpunkt stand. Dies hat eine relativ exakte Aufgabenstellung zur Folge und es muss z. B. nicht auf die Akzeptanz des Systems bei den Anwendern geachtet werden, wie es bei einem realen Produkt der Fall wäre.

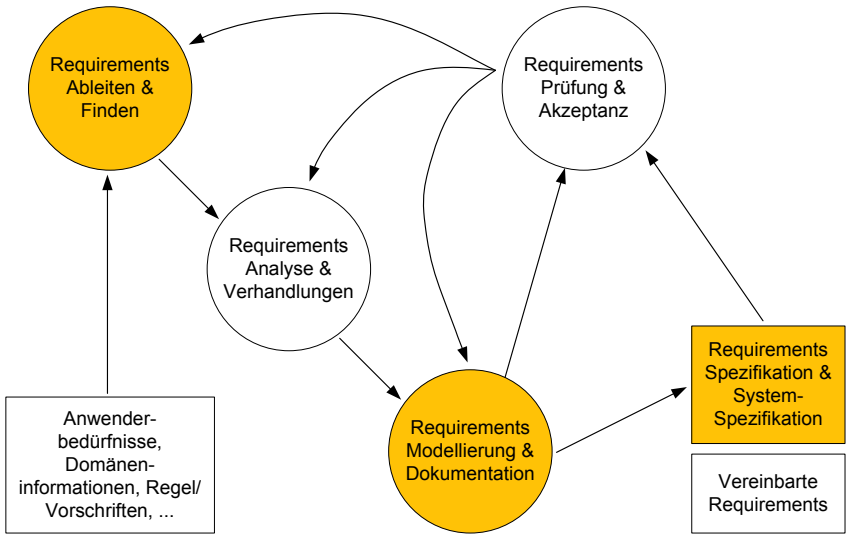



Abbildung 2.2: Überblick über einen generischen RE-Prozess (Quelle: [30, Seite 360], im Original ohne die orangen Markierungen)

Im Folgenden werden einige Definitionen für die in dieser Arbeit verwendeten zentralen Begriffe des Requirements Engineering gegeben.

**Anforderung (requirement)**

„Eine Anforderung ist eine Aussage über eine zu erfüllende Eigenschaft oder zu erbringende Leistung eines Produktes, eines Prozesses oder der am Prozess beteiligten Personen.“

*Sophist Group: Requirements-Engineering und -Management [30, Seite 135]*

„A requirement is a statement of need, something that some class of user or other stakeholders wants. “

*Ian F. Alexander; Richard Stevens: Writing Better Requirements [2, Seite 8]*

„Condition or capability needed by a user to solve a problem or achieve an objective.“

*IEEE Standard Glossary of Software Engineering Terminology [16]*

Als erster zentraler Begriff wird der Begriff der Anforderung beschrieben. Die erste Definition der Sophist Group stellt eine Anforderung als eine Funktionalität dar. Die Definitionen von Ian Alexander und der IEEE hingegen beschreiben eine Anforderung als etwas, das von einem Projektbeteiligten benötigt wird. An den Definitionen kann man erkennen, dass Anforderungen sowohl explizit als auch implizit geäußerte Wünsche des Kunden sind. Die Aufgabe des Requirements Engineering ist, alle Anforderungen explizit zu formulieren.

Zur Darstellung von Anforderungen muss man sich für einen Formalismus entscheiden. Oft wird natürliche Sprache verwendet. Dies hat einige Vorteile, wie die einfache Verständlichkeit und leichte Änderbarkeit. Allerdings gibt es auch Nachteile, wie Mehrdeutigkeit und die fehlende maschinelle Verarbeitbarkeit (vergleiche [26, Seite 39]).

Ein realisiertes System besitzt eine Menge von Funktionen, welche wie folgt definiert werden:

**Funktion (function)**

„Eine Funktion ist eine auf Grund einer Anforderung in einem System realisierte Funktionalität.“

In dieser Arbeit werden zwei grundlegende Arten von Anforderungen unterschieden: funktionale und nicht-funktionale Anforderungen.

Ein Beispiel für eine funktionale Anforderung ist, dass bei einer Textverarbeitung die Eingabe von Text möglich sein soll. Diese Anforderung wird in einer oder mehreren Funktionen des Systems realisiert. Mittels Testen kann dann geprüft werden, ob das System es ermöglicht Text einzugeben.

Eine nicht-funktionale Anforderung ist z. B., dass die Eingabe des Texts komfortabel erfolgen soll. Diese Anforderung ist schwerer zu prüfen, da nicht klar ist, wann die Eingabe komfortabel ist. Richtig formuliert muss die Anforderung lauten, dass die Eingabe die firmeneigenen Richtlinien für komfortabeles Eingeben von Text erfüllen muss. Dann ist es möglich zu testen, ob die Texteingabe die Richtlinien erfüllt.

Im Folgenden werden Definitionen für funktionale und nicht-funktionale Anforderungen gegeben:

**funktionale Anforderungen (functional requirement)**

„Eine funktionale Anforderung ist der Grund für eine oder mehrere Funktionen eines Systems.“

*Ian F. Alexander; Richard Stevens: Writing Better Requirements [2, Seite 9]*

**nicht-funktionale Anforderung (non-functional requirement, constraint)**

„Eine nicht-funktionale Anforderung ist eine Einschränkung oder Abänderung einer oder mehrerer Anforderungen bzw. deren Realisierung(en).“
*Ian F. Alexander; Richard Stevens: Writing Better Requirements [2, Seite 9]*

Schließlich soll noch das Konzept des Use Case vorgestellt werden. Use Cases sind kein Ersatz für Anforderungen, sondern ergänzen die Beschreibung des gewünschten Systems. Use Cases werden wie folgt definiert:

**Use Case (Anwendungsfall)**

„Ein Anwendungsfall ist dabei eine Beschreibung einer typischen Benutzung des Systems, die im Allgemeinen mehrere Objekte involviert. Er wird verwendet, um das Verhalten des Gesamtsystems (oder essentieller Teile davon) zu modellieren.“
*Helmuth Partsch: Requirements-Engineering systematisch [26, Seite 244]*

Requirements Engineering zu betreiben ist zeitaufwändig. Aber nur, wenn die Anforderungen das vom Kunden gewünschte System exakt beschreiben, kann das System in der vorgegeben Zeit und mit dem vorgegebenen Budget erstellt werden. Ohne exakte Anforderungen müssen erfahrungsgemäß nachträglich viele zeitaufwändige Änderungen vorgenommen werden. Der Aufwand für das Requirements Engineering kann durch geeignete Tools verringert werden.

### 2.2.1 Traceability

Die Sophist Group definiert in [30, Seite 375] Traceability als „die Nachvollziehbarkeit von Entscheidungen und Abhängigkeiten zwischen Informationen und deren Repräsentationsformen, vom Projektbeginn bis zu seinem Ende.“

Abbildung 2.3 zeigt ein Beispiel für die Traceability durch alle Phasen eines Projekts. Zwischen den Artefakten des Projekts können Traceability-Links bestehen. Diese Links können eine beliebige Abhängigkeit darstellen. So sind z. B. die Anforderungen durch einen „fordert“-Link an den Kunden gebunden und die Testfälle mit einem-„prüft“ Link an die Anforderungen. Mit Hilfe der Links in dem Beispiel können folgende Frage beantwortet werden:

- Welche Anforderung stammt von welchem Kunden?
- Wie wurde eine Anforderung umgesetzt?
- In welchen Klassen wurde eine Anforderung umgesetzt?
- Welche Testfälle prüfen welche Anforderungen?

Mit Hilfe der Traceability-Links können bei einer Änderung die anderen Artefakte, die auch

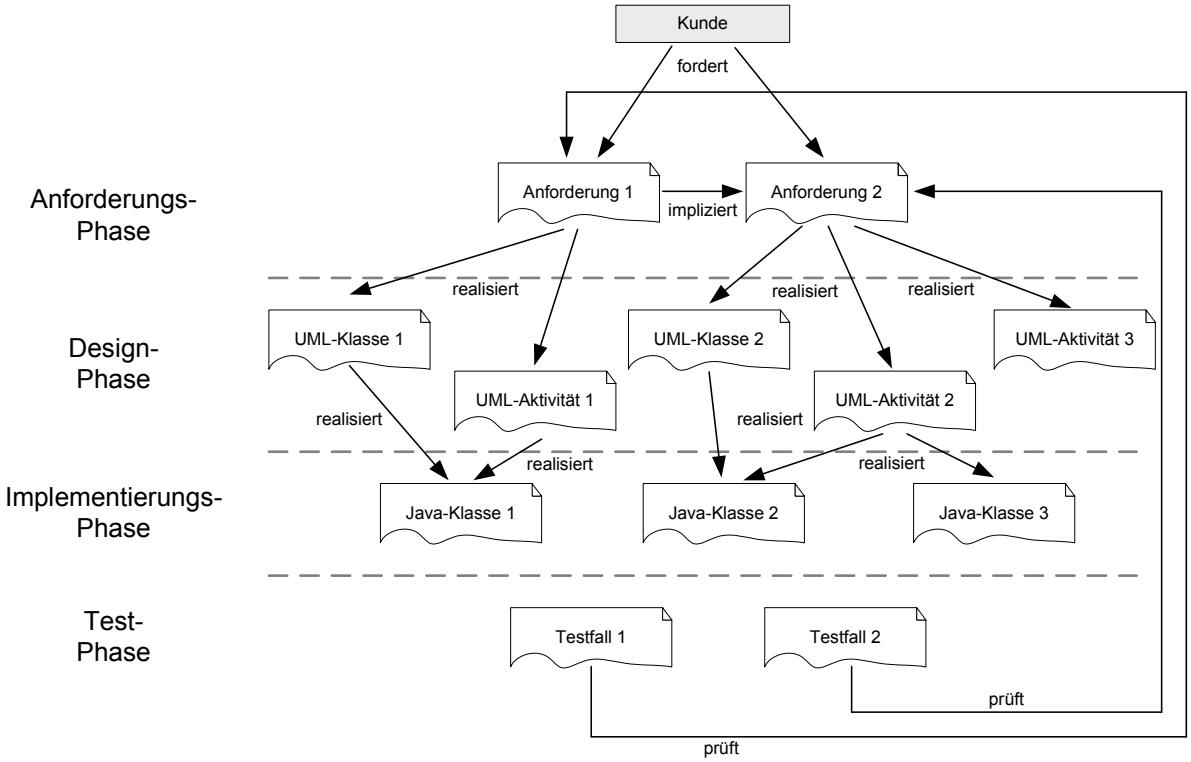



Abbildung 2.3: Beispiel für eine Traceability durch die verschiedenen Phasen

geändert werden müssen, identifiziert werden. Denn erst dann lassen sich alle notwendigen Änderungen in allen betroffenen Artefakten durchführen. Erst dies ermöglicht die Erhaltung der Konsistenz zwischen den Artefakten.

Bei Anforderungen ist der Ursprung sehr wichtig, da dadurch klar wird, warum die Anforderung vorhanden ist. Dies vermeidet die Umsetzung nicht oder nicht mehr benötigter Anforderungen und ermöglicht es, bei Unklarheit der Anforderungen die entsprechenden Kunden zu befragen.

Die Traceability ist ein sehr wichtiger Teil des Requirements Engineering, da erst die Links den Zusammenhang aller Artefakte einer Software beschreiben und verständlich machen. So kommt auch die Sophist Group [30, Seite 377] zu dem Schluss: „Trotz des Mehraufwandes, den Sie leisten müssen, ist die Traceability essenziell für Ihren Projekterfolg.“ Dieser Mehraufwand kann wiederum durch die Verwendung von geeigneten Tools verringert werden.

## 2.3 Modell Driven Architecture

Die Model Driven Architecture [22] (MDA; dt. Modellgetriebene Architektur, MDA) der OMG (Object Management Group) stellt einen offenen und herstellerunabhängigen Ansatz zur modellbasierten Anwendungsentwicklung dar. Modellbasierte Anwendungsentwicklung

bedeutet, dass das Modell der Anwendung im Mittelpunkt des Entwicklungsprozesses steht. Das Modell besitzt dabei folgende Eigenschaften:

1. Es stellt eine Sicht mit einem sehr hohen Abstraktionsgrad auf die zu entwickelnde Software dar.
2. Es enthält nur fachliche Details, aber keine Berücksichtigung der Implementierung
3. Es ist plattformunabhängig, da es unabhängig von der Implementierung und somit der verwendeten Technologie ist

Da das Modell unabhängig von der technischen Plattform ist, wird es plattformunabhängiges Modell (engl. Platform Independent Model, PIM) genannt. Bei dem MDA-Ansatz der OMG besteht das plattformunabhängige Modell aus einer Menge von UML-Modellen.

Abbildung 2.4 zeigt die einzelnen Schritte bei der Entwicklung mit einer MDA. Als Ausgangspunkt dienen die Anforderungen an die zu erstellende Software. Diese Anforderungen werden in einem plattformunabhängigen Modell modelliert. Da die Anforderungen meist als natürlichsprachlicher Text vorliegen, muss dieser Schritt meist von Hand durchgeführt werden.

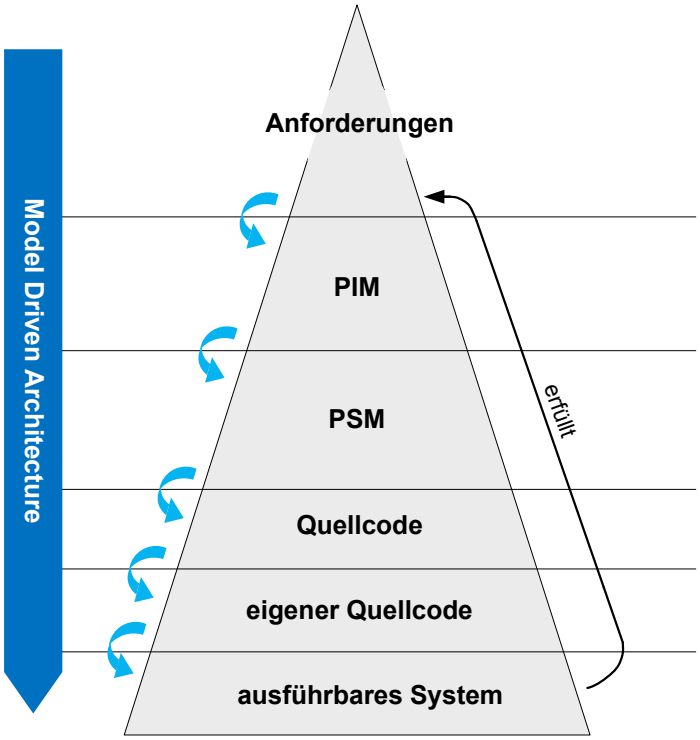



Abbildung 2.4: Schematischer Überblick über die verschiedenen Ebenen einer MDA

Das plattformunabhängige Modell wird durch eine Modelltransformation in ein plattformspezifisches Modell (engl. Platform Specific Model, PSM) umgewandelt. Diese Modelltransformation besteht aus Transformationsregeln für eine bestimmte Zielplattform und kann meist

zu großen Teilen automatisiert durchgeführt werden. Zielplattformen sind z. B. J2EE oder CORBA. Zu einem plattformunabhängigen Modell kann es beliebig viele plattformspezifische Modelle geben, da pro gewünschter Zielplattform ein plattformspezifisches Modell erstellt werden kann. Aus dem plattformspezifischen Modell wird schließlich Quellcode generiert, der unter Umständen im nächsten Schritt noch um eigenen Quellcode erweitert werden muss. Am Ende steht ein ausführbares System, das die Anforderungen erfüllt.

Wie Kleppe et al. in [19] beschreiben, können durch die Verwendung einer MDA noch andere positive Effekte erreicht werden. So wird die Produktivität erhöht, da die Entwicklung auf einer höheren Abstraktionsebene stattfindet und große Teile des Codes automatisch generiert werden können. Allerdings erfordert die Generierung des Codes passende Tools.

Außerdem steigt die Wartbarkeit und die Dokumentation der Software wird verbessert, da Änderungen an der Software nicht mehr direkt im Code gemacht werden, sondern im plattformunabhängigen oder plattformspezifischen Modell. Bei Änderungen im plattformspezifischen Modell müssen die Änderungen, entweder per Hand oder mit Hilfe eines Tools, auch im plattformunabhängigen Modell vorgenommen werden. Somit bleibt die Dokumentation in Form des plattformunabhängigen Modells immer mit dem Code konsistent.

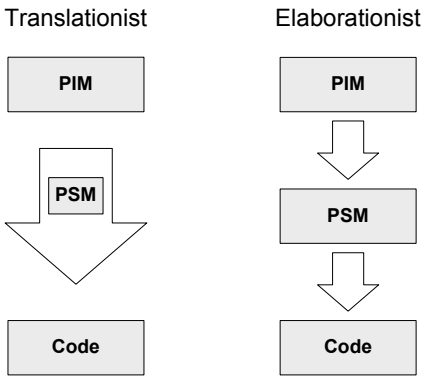



Abbildung 2.5: Zwei Hauptinterpretationen der MDA

Es existieren zwei verschiedene Ausprägungen der MDA (siehe Abbildung 2.5). Der „Elaborationist“-Ansatz entspricht dem vorgestellten Vorgehen. Bei dem „Translationist“-Ansatz gibt es nur ein plattformspezifisches Modell, welches lediglich implizit als Zwischenprodukt des Code-Generators vorliegt. Das bedeutet, es gibt nur eine feste Zielplattform.

Die im nächsten Abschnitt vorgestellte ActiveCharts IDE verfolgt den „Translationist“-Ansatz.

### 2.3.1 ActiveCharts IDE

Die an der Universität Ulm im Institut für Programmiermethodik und Compilerbau entwickelte ActiveCharts IDE verwendet zur Beschreibung des plattformunabhängigen Modells Klassen- und Aktivitätsdiagramme der UML.

Die Klassendiagramme dienen zur Beschreibung der statischen Struktur der Anwendung und der verwendeten Signale. In den Klassendiagrammen muss für jedes in den Aktivitätsdiagrammen verwendete Signal (siehe Übersicht in Abbildung 2.6) eine Klasse mit dem Namen des Signals und dem UML-Stereotypen „Signal“ definiert werden.

Mit den Aktivitätsdiagrammen hingegen wird das Verhalten der Anwendung beschrieben. Abbildung 2.6 zeigt einen Überblick über die wichtigsten Elemente eines Aktivitätsdiagramms. Für weitere Informationen siehe z. B. [25] oder die UML-Spezifikation [24].

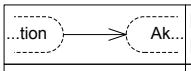
| | |
|---|---|
|  | Die Flüsse sind die Verbindungen zwischen allen Arten von Knoten. Es werden Kontrollflüsse und Datenflüsse unterschieden. |
| | Start- und Endknoten stellen den Start und das Ende einer Aktivität dar. |
| | Eine Aktion führt eine Tätigkeit aus. Dies kann eine andere Aktivität oder im Fall der ActiveCharts IDE von Hand geschriebener C#-Code sein. |
| | Sendet ein Signal mit dem angegebenen Namen (hier im Beispiel „Sender“). |
| | Wartet auf den Empfang eines Signals mit dem angegebenen Namen (hier im Beispiel „Empfänger“). |
| | Wartet einen vorgegebenen Zeitraum, hier sind es 5 Sekunden. |
| | Mit Hilfe eines Entscheidungsknotens können sowohl Entscheidungen als auch Zusammenführungen modelliert werden. |
|  | Teilt den Fluss entweder auf oder führt ihn zusammen. Im Gegensatz zu den Entscheidungsknoten müssen bei einer Zusammenführung alle eingehenden Flüsse aktiv sein. |

Abbildung 2.6: Überblick über die wichtigsten Elemente eines Aktivitätsdiagramms

In den Aktionen der Aktivitätsdiagramme kann handgeschriebener C#-Code ausgeführt werden.

Die Klassendiagramme werden von einem Codegenerator in C#-Code übersetzt und können durch eigenen Code erweitert werden. Die Aktivitätsdiagramme und der handgeschriebene C#-Code werden von einem Interpreter ausgeführt.

Die Diagramme werden in Microsoft Visio erstellt. Die ActiveCharts IDE (siehe Abbildung 2.7), welche den ActiveCharts Interpreter enthält, ist ein eigenständiges Programm und kommuniziert mit Visio, um die Ausführung der Kontrollflüsse zu visualisieren. Die ActiveCharts IDE erlaubt ein grafisches Debugging (siehe Abbildung 2.8, die gerade aktiven Elemente sind grün markiert) und das Setzen von Breakpoints. Die Ausführung kann entweder in Schritten mit Pausen dazwischen, in Einzelschritten oder in Echtzeit stattfinden.

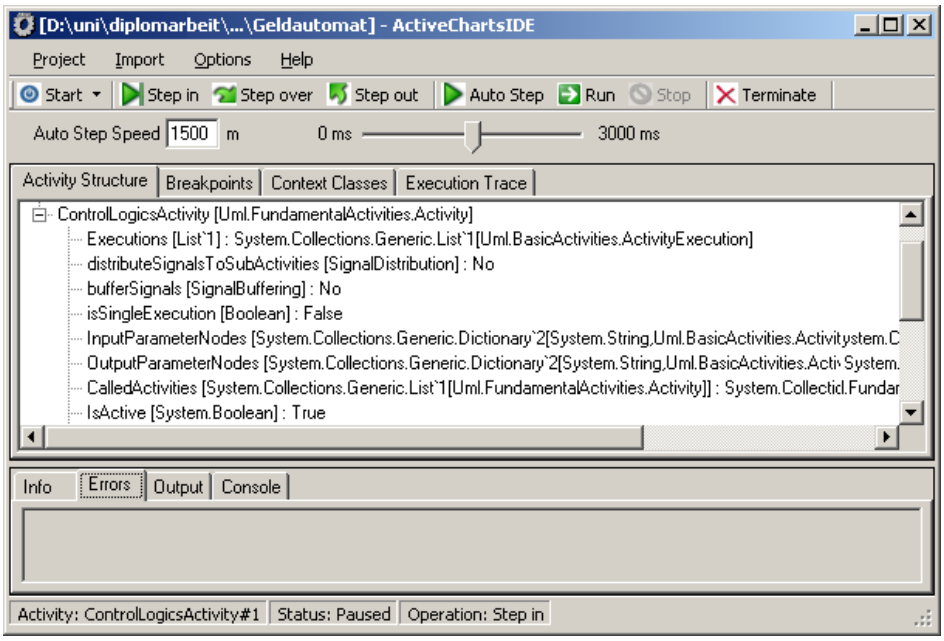



Abbildung 2.7: Screenshot der ActiveCharts IDE

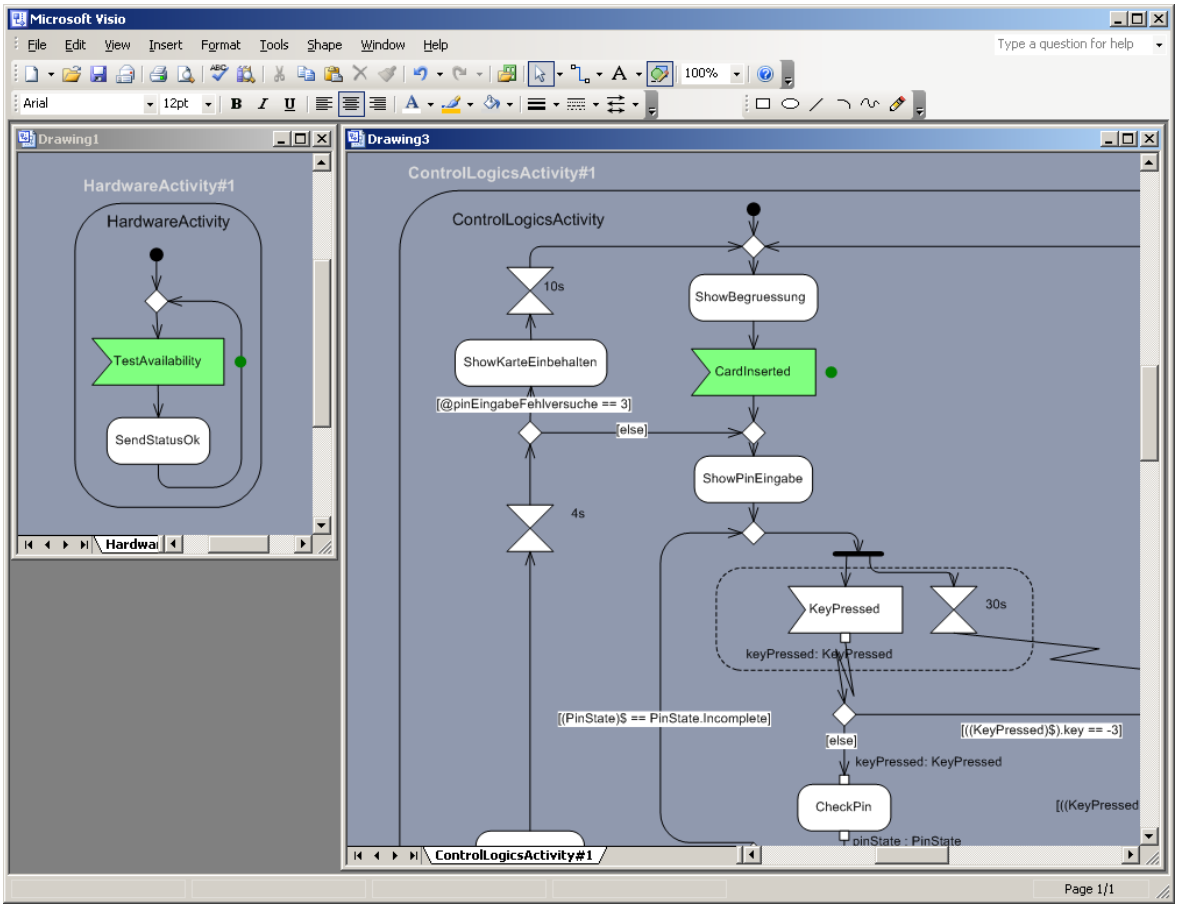



Abbildung 2.8: Screenshot der Visualisierung des Programmflusses in Visio

Visual Studio 2005 dient zum Schreiben des Codes für die Aktionen und zum Debuggen des Codes.

Momentan wird an einer Portierung der ActiveCharts IDE nach Java gearbeitet. Als Grundlage dient Eclipse (siehe Abschnitt 2.4). Eclipse wurde gewählt, da es sehr einfach mittels Plug-ins erweiterbar und Open Source ist. Denn zwei große Nachteile der ursprünglichen ActiveCharts IDE Version sind, dass sowohl Visual Studio als auch Visio kommerzielle Produkte sind und, dass die Zusammenarbeit zwischen der ActiveCharts IDE und Visio nicht immer reibungslos ist. Allerdings befindet sich diese Portierung noch in einem sehr frühen Stadium und ist zum jetzigen Zeitpunkt noch nicht lauffähig.

## 2.4 Eclipse

Eclipse 3.2 ist ein Open Source-Framework zur Entwicklung von Software jeder Art. Dabei stellt Eclipse selbst nur den Kern dar, der Plug-ins laden und verwalten kann. Das JDT-Projekt (Java Development Tools Projekt) stellt Plug-ins bereit, die Eclipse in eine Java-IDE verwandeln. Eclipse und alle Plug-ins sind in Java geschrieben. Somit ermöglicht das JDT, Eclipse als eine Entwicklungsumgebung für sich selbst zu verwenden.

Zur Darstellung verwendet Eclipse das SWT (Standard Widget Toolkit). SWT wurde entwickelt, da Swing zu langsam war und AWT (Abstract Window Toolkit) zu wenig Steuerelemente bereitstellte. AWT stellt nur die Steuerelemente bereit, welche die Schnittmenge aller verfügbarer Steuerelemente der unterstützten Plattformen bilden. SWT verwendet einen gemischten Ansatz. Es verwendet die vorhandenen nativen Steuerelemente der jeweiligen Plattform und emuliert die nicht vorhandenen Steuerelemente. Außerdem wurde darauf geachtet, dass SWT möglichst wenig Overhead besitzt. Dadurch sind SWT-Anwendungen im Gegensatz zu Swing-Anwendungen sehr performant. Da die Eclipse-GUI mit SWT geschrieben ist, müssen auch alle Plug-ins SWT verwenden. Weil SWT nur die grundlegenden Steuerelemente zur Verfügung stellt, gibt es eine Bibliothek namens JFace, welche die SWT-Steuerelemente zu komplexeren Steuerelementen zusammensetzt und z. B. die Verwendung einer MVC-Architektur (siehe z. B. [5, Seite 124]) erlaubt.

Mit Hilfe von Plug-ins kann Eclipse um beliebige Funktionalitäten erweitert werden. Die zur Erweiterung verwendeten Elemente sind Editoren, Views und Perspektiven. Editoren werden zur Darstellung und zum Editieren von Ressourcen, meist Dateien, verwendet. Sie folgen einem Open-Save-Close-Modell. Das bedeutet, dass beim Laden einer Datei der dazugehörige Editor geöffnet wird. So sind beliebige Änderungen an der Ressource möglich, die spätestens beim Schließen des Editors gespeichert oder verworfen werden. Im Gegensatz dazu werden Views zum Anzeigen von Eigenschaften der Ressourcen oder zum Navigieren in den Ressourcen verwendet. Die in den Views gemachten Änderungen werden sofort übernommen und nicht wie bei den Editoren erst beim nächsten Speichern.

Abbildung 2.9 zeigt einen Screenshot von Eclipse. Der blau umrandete Bereich rechts in der Mitte ist ein Editor für Java-Dateien. Der orange markierte Bereich an der linken Seite („Package Explorer“) und die orange umrandeten Elemente unten rechts sind Views. Zur Organisation der Views werden Perspektiven verwendet. Jede Perspektive enthält mehrere Views. Durch den Wechsel der Perspektiven können unterschiedliche Aspekte betrachtet werden. In Abbildung 2.9 sind die Java- und die Debug-Perspektive verfügbar, wobei die Java-Perspektive aktiviert ist.

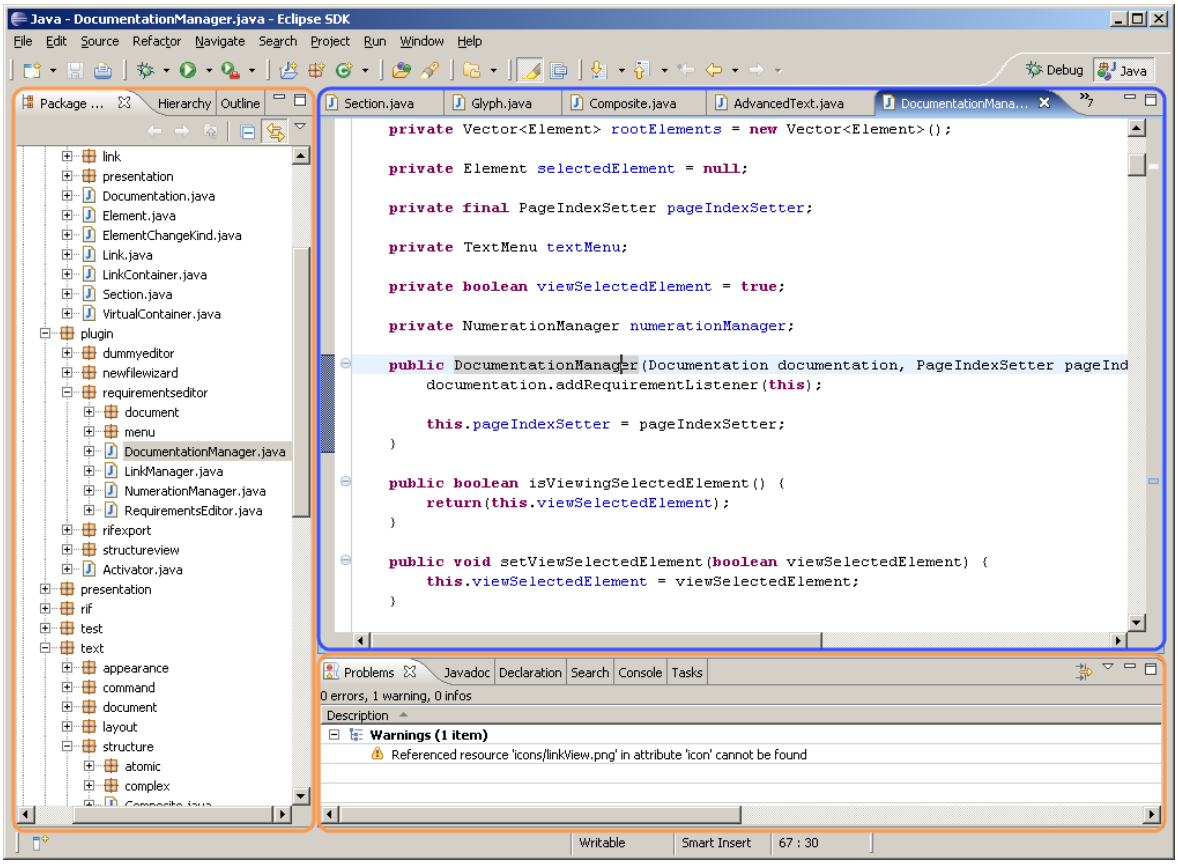



Abbildung 2.9: Screenshot der Eclipse-IDE

Zusätzlich zu den Editoren, Views und Perspektiven können Plug-ins die Eclipse-IDE um neue Menüpunkte, neue Einträge in den Pop-up-Menüs und neue Tastenkombinationen erweitern. Weitere Informationen zur Programmierung von Eclipse-Plug-ins finden sich u. a. in [6].

## 2.5 Requirements Interchange Format

Das RIF-Format (Requirements Interchange Format) ist ein standardisiertes Austauschformat für Anforderungen in der Automobilindustrie. Es ist ein Entwurf der HIS (Herstellerinitiative Software), welcher 2005 zum ersten Mal vorgestellt wurde. Mitglieder der Herstellerinitiative Software sind unter anderem die Audi AG, die BMW Gruppe, die DaimlerChrysler

AG, die Porsche AG und die Volkswagen AG. Es wurde entwickelt, weil bis zu dem damaligen Zeitpunkt kein gemeinsames Austauschformat für Requirements-Tools verschiedener Hersteller zur Verfügung stand. Durch die Unterstützung der namhaften Mitglieder der HIS und durch die Tatsache, dass es bisher kein anderes gemeinsames Austauschformat gab, hat das RIF-Format das Potential, sich als Standard zu etablieren. Das Ziel des RIF-Formats ist es, Anforderungen zwischen den verschiedenen Herstellern und Zulieferern aus der Automobilindustrie auszutauschen. Wichtige Eigenschaften sind dabei die Herstellerunabhängigkeit und der verlustfreie Austausch der Anforderungen. Abbildung 2.10 zeigt das RIF-Format als Austauschformat zwischen Requirements Management-Tools verschiedener Hersteller.

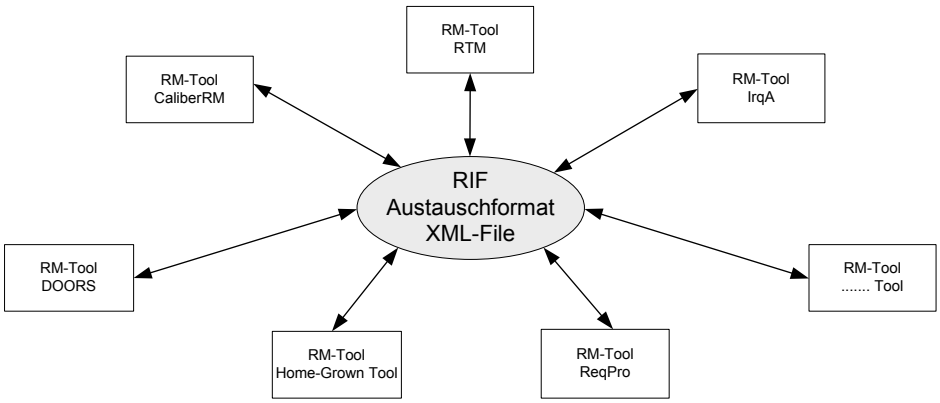



Abbildung 2.10: RIF-Format als Austauschformat zwischen verschiedenen Tools (Quelle: [14])

Allerdings muss jeder Toolhersteller eine Import- und Exportfunktion für das RIF-Format bereitstellen. Daher ist der Erfolg des RIF-Formats direkt von der Unterstützung der Hersteller abhängig. Zum jetzigen Zeitpunkt existiert bereits ein Tool namens EXERPT [12] der Firma EXTESSY. EXERPT soll als Vermittler zwischen den einzelnen Requirements-Tools der verschiedenen Hersteller dienen. Zur Anbindung eines Tools wird ein passendes Plug-in benötigt. Bis jetzt ist nur ein Plug-in für Telelogic Doors verfügbar. Ein Requirements-Tool, welches selbst einen Import und Export für das RIF-Format bereitstellt, ist IRqA [29] von QA Systems.

Da das RIF-Format herstellerunabhängig ist, wird ein eigenes Typensystem benötigt. Zur Festlegung des Typensystems wird in der RIF-Spezifikation ein RIF-Modell in Form von UML-Klassendiagrammen spezifiziert. Zum Abbilden der Objekte der Klassen auf eine XML-Datei werden Abbildungsregeln angegeben. Abbildung 2.11 zeigt die wichtigsten Klassen des Modells.

Das Äquivalent zu den Klassen in der objektorientierten Programmierung bilden die Instanzen der SpecType-Klasse. Sie stellen einen Typ dar, wie z. B. eine Anforderung oder einen Use Case. Diese Typen können Attribute besitzen, deren Datentypen durch Instanzen der Klasse DatatypeDefinition beschrieben werden. Instanzen der Typen werden durch SpecObject-Instanzen repräsentiert. Wieder übertragen auf die objektorientierte Programmierung kann

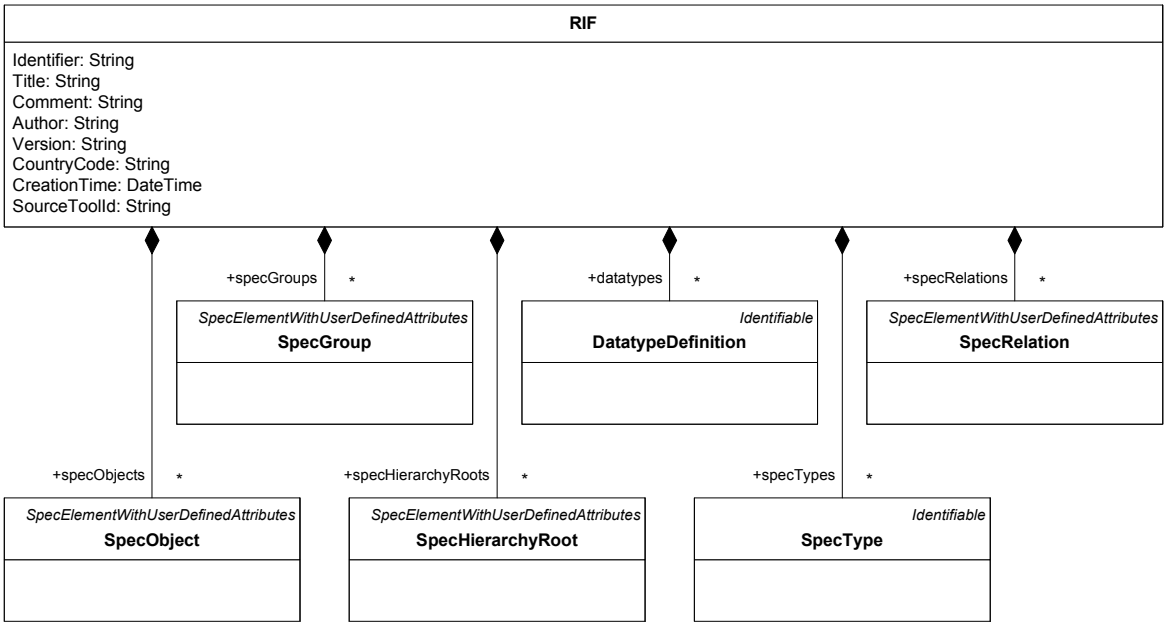



Abbildung 2.11: Das RIF-Modell als UML-Klassendiagramm

zusammengefasst werden, dass es in dem Typensystem des RIF-Formats Klassen mit Attributen und Instanzen der Klassen gibt. Daher werden sowohl die Klassendefinitionen als auch die Instanzen der Klassen in die XML-Datei geschrieben.

Zusätzlich können SpecGroup-Instanzen dazu verwendet werden, Instanzen von SpecObject zu gruppieren. Und SpecHierarchyRoot-Instanzen können zur hierarchischen Strukturierung von SpecObject-Instanzen verwendet werden.

Da im RIF-Format beliebige Typen definiert werden können, müssen Tools, die Daten über das RIF-Format austauschen, sich entweder auf die Schnittmenge der intern von den Tools verwendeten Typen oder auf eine vorgegebene Menge von Typen einigen. So kann z. B. das Tool A einen Typ für einen Use Case kennen, Tool B hingegen nicht. In diesem Fall muss Tool B den Use Case anders behandeln. Eine Möglichkeit hierzu ist die Abbildung auf eine Tabelle. Allerdings muss beim Export dafür gesorgt werden, dass der Use Case wieder in seiner ursprünglichen Form und nicht als Tabelle exportiert wird. Diese Abbildung der verschiedenen internen Typen der Tools aufeinander wird Mapping genannt.

Ein weiteres Problem stellt die große Anzahl an benötigten Mappings dar. Wie in Abbildung 2.12 gezeigt, sind bei 6 Tools 30 verschiedene Mappings nötig. Denn jedes Tool benötigt ein Mapping für die internen Typen jedes anderen Tools. Wie schon oben erwähnt, ist die Einführung einer gemeinsamen Schnittmenge an verwendeten Typen eine Möglichkeit, das Problem zu mildern. Eine andere Möglichkeit ist, dass sich die internen Typen eines bestimmten Tools, z. B. Telelogic Doors, als Quasistandard etablieren.

Weitere Informationen über das RIF-Format finden sich in der Spezifikation [14]. In [4] wird

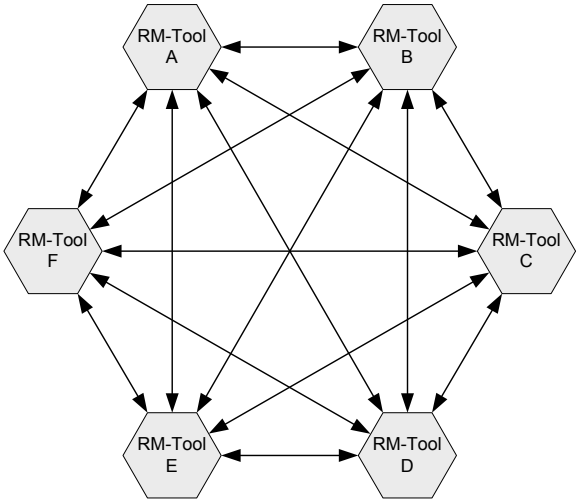



Abbildung 2.12: Beispiel für die große Anzahl an benötigten Mappings

untersucht, wie ein RIF-Export aus Telelogic Doors realisiert werden kann.

### 2.5.1 Castor

Castor ist eine Open Source Java-Bibliothek zur Abbildung von XML-Daten auf Java-Objekte. Mittels des sogenannten Marshalling-Frameworks kann Castor Java-Objekte in eine XML-Datei schreiben und wieder laden und als Java-Objekte zurückgeben.

Um die Abbildung der Java-Objekte auf die XML-Datei anzupassen, kann eine XML-Mapping-Datei verwendet werden. Die Mapping-Datei erlaubt z. B. eine freie Wahl der verwendeten Namen der XML-Tags, auf die abgebildet wird. Abbildung 2.13 zeigt schematisch die Funktionsweise des Marshalling-Frameworks von Castor.

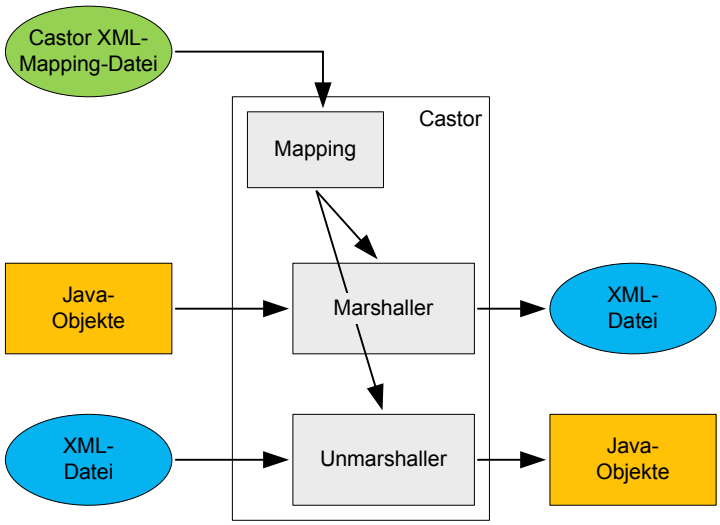



Abbildung 2.13: Schematischer Überblick über das Marshalling-Framework

Der Marshaller ist für die Umwandlung der Java-Objekte in eine XML-Datei und der Unmarshaller für die Umwandlung einer XML-Datei in Java-Objekte verantwortlich. Beide berücksichtigen dabei das übergebene Mapping, welches zur Laufzeit aus der XML-Datei geladen und mit denselben Mechanismen des Marshalling-Frameworks wie in Abbildung 2.13 in die korrespondierenden Java-Objekte umgewandelt wird. Dadurch ist es möglich, das Mapping dynamisch zur Laufzeit anzupassen.

Ein Mapping enthält Klassen- und Feld-Deskriptoren. Klassen-Deskriptoren geben an, wie Objekte einer bestimmten Java-Klasse abgebildet werden. Listing 2.1 zeigt einen Klassen-Deskriptor für die Klasse String. In der XML-Datei werden mit diesem Mapping alle String-Objekte durch einen „MY-STRING“-Tag repräsentiert. Die Angabe der Superklasse darf in einem Mapping nur gemacht werden, wenn auch für die Superklasse ein Klassen-Deskriptor existiert.

```
1 <class name="java.lang.String" extends="java.lang.Object">
2    <map-to xml="MY-STRING" />
3 </class>
```

Listing 2.1: Beispiel eines Klassen-Deskriptors

Feld-Deskriptoren geben an, wie ein Attribut einer Java-Klasse in die XML-Datei abgebildet wird. Voraussetzung hierfür ist, dass es ein Getter- und Setter-Methodenpaar in der Klasse des Attributs gibt. Listing 2.2 zeigt einen Feld-Deskriptor für ein Attribut mit dem Namen „test“. Das bedeutet, es muss eine Methode getTest() und eine Methode setTest() geben, ansonsten tritt ein Fehler zur Laufzeit auf. In der XML-Datei wird das Attribut mit dem Tag „MY-TEST“ abgebildet.

```
1 <field name="test">
2    <bind-xml name="MY-TEST" />
3 </field>
```

Listing 2.2: Beispiel eines Feld-Deskriptors

Im Gegensatz zu handgeschriebenem Code zur XML-Serialisierung und -Deserialisierung bietet Castor eine robuste Grundlage, die mit wenig Aufwand eine Umwandlung von Java-Objekten in XML-Dateien ermöglicht. Der Aufwand ist geringer, da kein neuer Code geschrieben werden muss, sondern nur eine XML-Mapping-Datei.

# 3 Theoretischer Teil

Dieses Kapitel stellt den zentralen Teil der Arbeit dar. Es wird die theoretische Grundlage für den Prototypen (siehe Kapitel 4) gelegt.

Dazu wird in Abschnitt 3.1 das Vorgehen aus dem Praktikum in einem einfachen Vorgehensmodell formalisiert und erläutert, welche Artefakte in dem Vorgehensmodell eine Rolle spielen. Diese Artefakte sind in erster Linie Anforderungsdokumente, deren Aufbau und Inhalt in Abschnitt 3.2 beschrieben wird. Abschnitt 3.3 stellt die Untersuchung der Traceability dar. In Abschnitt 3.4 wird schließlich ein kurzer Überblick über zwei kommerzielle Tools gegeben.

## 3.1 Vorgehensmodell zur Entwicklung mit der ActiveCharts IDE

In diesem Abschnitt wird der Entwicklungsprozess aus den Praktika genauer untersucht. Das erste Ziel ist, alle an dem Entwicklungsprozess beteiligten Artefakte zu identifizieren. Denn diese Artefakte werden in Abschnitt 3.2 benötigt, um die Struktur und den Inhalt der Anforderungsdokumente festzulegen. Das zweite Ziel ist, das Vorgehen in einem einfachen Vorgehensmodell zu formalisieren.

### 3.1.1 Entwicklungsprozess aus den Praktika

Im Folgenden wird das Vorgehen aus den Praktika vorgestellt. Dies beinhaltet in erster Linie die einzelnen Phasen und die in den Phasen beteiligten bzw. produzierten Artefakte. Das an dieser Stelle vorgestellte Vorgehen ist eine Mischung aus den Vorgehen beider Praktika. So musste im ersten Praktikum, der Mikrowelle, die simulierte Hardware in der Design-Phase erstellt werden, wohingegen im zweiten Praktikum, dem Aufzug, die Hardware schon vorgegeben war.

Die Entwicklung fand in mehreren Kleingruppen statt, die jeweils eine eigene Version des Systems entwickelten.

Das grobe Vorgehen wurde durch Abbildung 3.1 vorgegeben. Auf dem Zeitstrahl sind die einzelnen durchzuführenden Phasen und ihre Dauer in Wochen angegeben. Die Design-Phase fängt erst in der vierten Woche an, da vor der Entwicklung des eigentlichen eingebetteten

Systems einige Übungen zum Kennenlernen der ActiveCharts IDE durchgeführt wurden.

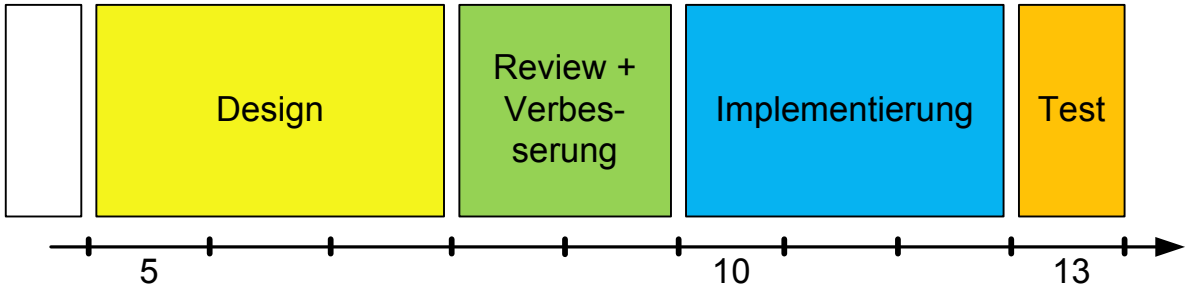



Abbildung 3.1: Vorgabe aus den Praktika (Quelle: Praktikum)

Im Folgenden werden die einzelnen Phasen und ihre durchzuführenden Aktivitäten im Detail beschrieben.

#### 3.1.1.1 Design-Phase

In der Design-Phase werden folgende Schritte durchgeführt:

1. Anforderungsanalyse
   a) Erstellung eines Anforderungsdokuments auf Basis der initialen Aufgabenbeschreibung
   b) Erstellung von Testfällen aus den Anforderungen
   c) Erstellung der Spezifikation und Simulation der Hardware (optional)
2. Modellierung
   a) Erstellung eines Domänenmodells (UML-Klassendiagramme)
   b) Modellierung der Kontrollflüsse (UML-Aktivitätsdiagramme)

Abbildung 3.2 zeigt einen Überblick über alle an der Design-Phase beteiligten Artefakte, die im Folgenden beschrieben werden.

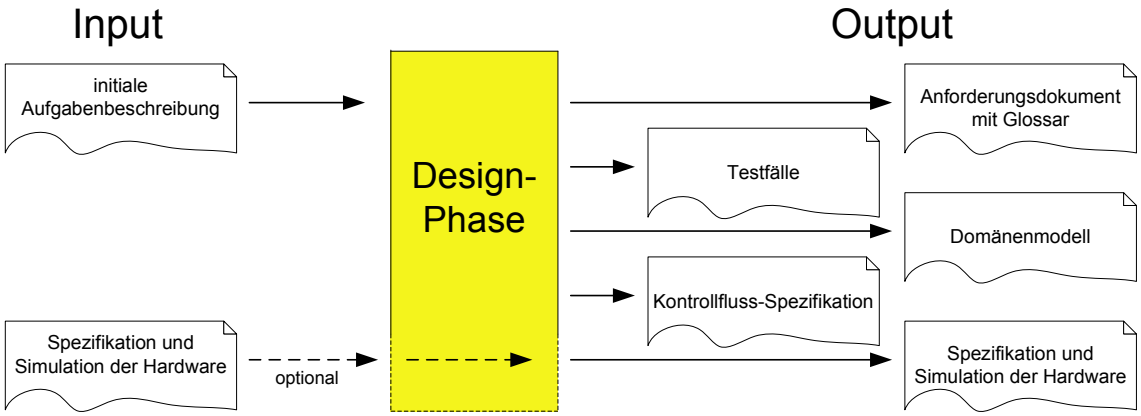



Abbildung 3.2: Ein- und Ausgabenartefakte der Design-Phase

Aus der initialen Aufgabenbeschreibung wird ein Anforderungsdokument erstellt. Die initiale Aufgabenstellung umreißt das gewünschte eingebettete System sehr grob. Es werden lediglich Informationen gegeben, um was für ein Gerät es sich handelt und welche Grundanforderungen es erfüllen soll. Da die Informationen nicht dazu ausreichen, ein vollständiges Anforderungsdokument zu erstellen, übernehmen die Praktikumsbetreuer die Rolle des Kunden. Das bedeutet, dass sie für Rückfragen zu dem zu entwickelnden System zur Verfügung stehen. Das Anforderungsdokument besteht aus einzelnen Anforderungen, die an das System gestellt werden, und zusätzlich aus Use Cases. Die Use Cases werden zur Beschreibung komplexer Eingaben oder Zustandsänderungen des Systems verwendet. Es ist nicht zwingend notwendig (und meist nicht möglich, da die Anzahl der möglichen Use Cases viel zu groß ist), dass die Funktionalität des gesamten Systems von Use Cases abgedeckt wird.

Aus den Anforderungen werden Testfälle erzeugt, welche später dazu dienen sollen, am fertigen System zu testen, ob alle Anforderungen korrekt umgesetzt wurden.

Die Spezifikation und Simulation der Hardware kann schon vor der Design-Phase vorgegeben sein. Wenn nicht, muss sie in der Design-Phase erstellt werden. Eine Simulation der Hardware ist nötig, da keine reale Hardware zur Verfügung steht, was z. B. bei einem Aufzug nur schwer möglich ist.

Das Verhalten der Hardware kann durch eine Mischung aus Programmierung und Kontrollfluss-Modellierung nachgebildet werden. Die öffentliche Schnittstelle zur Interaktion mit der Hardware muss in der Spezifikation dokumentiert werden.

Nach der Anforderungsanalyse wird zuerst ein Domänenmodell als UML-Klassendiagramm erstellt. Das Domänenmodell ermöglicht eine erste Übersicht über den statischen Aufbau der Steuerungslogik des eingebetteten Systems. Zusätzlich wird ein Glossar erstellt, das zwingend alle Klassennamen aus dem Domänenmodell und beliebige weitere Einträge enthält. Es werden die aktiven und passiven Klassen identifiziert, die Beziehungen zwischen den Klassen und deren Kardinalitäten festgelegt und der zu speichernde Zustand in Form von Variablen definiert. Die aktiven Klassen werden im nächsten Schritt mit einem Kontrollfluss in Form einer Aktivität verknüpft.

Der Kontrollfluss beschreibt auf relativ hohem Abstraktionsniveau die Steuerlogik. In dieser Phase werden zwar schon Aktionen in die Modelle eingefügt, allerdings wird noch kein Code geschrieben.

#### 3.1.1.2 Review- und Verbesserungs-Phase

In der Review- und Verbesserungs-Phase werden folgende Schritte durchgeführt:

1. Ausarbeitung eines Review-Berichts über die bisher erstellten Artefakte einer anderen Gruppe aus der Design-Phase

2. Durchführung einer Review-Sitzung
3. Verbesserung der durch das Review aufgedeckten Fehler

Abbildung 3.3 zeigt einen Überblick über alle an der Review- und Verbesserungs-Phase beteiligten Artefakte, die im Folgenden beschrieben werden.

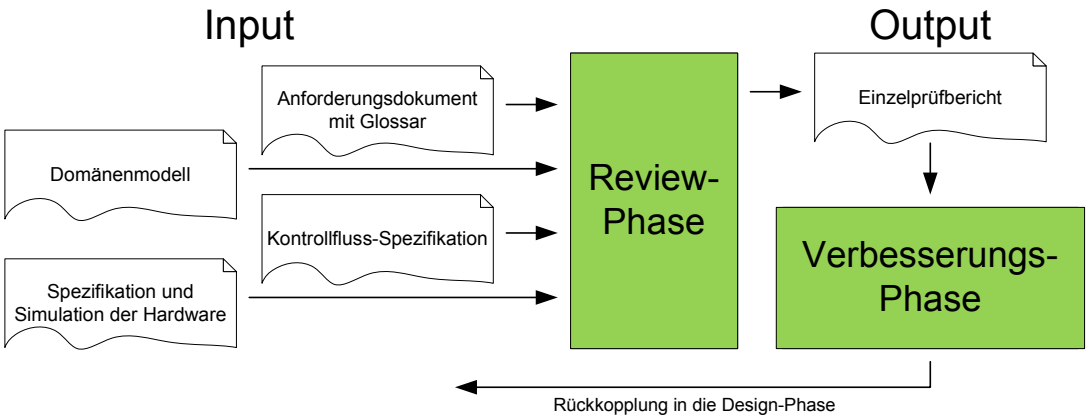



Abbildung 3.3: Ein- und Ausgabenartefakte der Review- und Verbesserungs-Phase

In der Review- und Verbesserung-Phase wird von einer anderen Gruppe ein Review der Artefakte aus der Design-Phase erstellt. Die gefundenen Fehler und Ungereimtheiten werden in einem Einzelprüfbericht festgehalten.

Dynamische Tests konzentrieren sich auf die Fehlerfindung am ausgeführten System. Reviews hingegen sind eine statische Test-Methode und sind universell einsetzbar, d. h. jedes Artefakt kann Gegenstand einer Review-Sitzung sein. Da Reviews keine lauffähige Software benötigen, können sie schon in frühen Phasen durchgeführt werden. Das hat den Vorteil, dass sich Fehler, die bereits früh im Entwicklungsprozess gefunden werden, leichter und mit weniger Arbeitsaufwand korrigieren lassen, als wenn sie in einer späteren Phase durch dynamische Tests gefunden werden.

Anschließend wird eine Review-Sitzung durchgeführt. Die Durchführung orientiert sich an dem IEEE Standard für die Durchführung von Reviews [18]. Das beinhaltet unter anderem, dass nur konstruktive Kritiken geäußert werden sollen, ein fester Zeitplan vereinbart wird und ein Moderator die Review-Sitzung leitet. Dies soll eine effiziente Durchführung der Reviews ermöglichen.

Nach der Review-Sitzung verbessern die einzelnen Gruppen die gefunden Fehler in den Dokumenten und den Modellen.

#### 3.1.1.3 Implementierungs-Phase

In der Implementierungs-Phase werden folgende Schritte durchgeführt:

1. Hardware-Simulation und modellierter Kontrollfluss werden zusammengebracht
2. Der Programmcode für die Aktionen aus den Aktivitätsdiagrammen wird implementiert

Abbildung 3.4 zeigt einen Überblick über alle an der Implementierungs-Phase beteiligten Artefakte, die im Folgenden beschrieben werden.

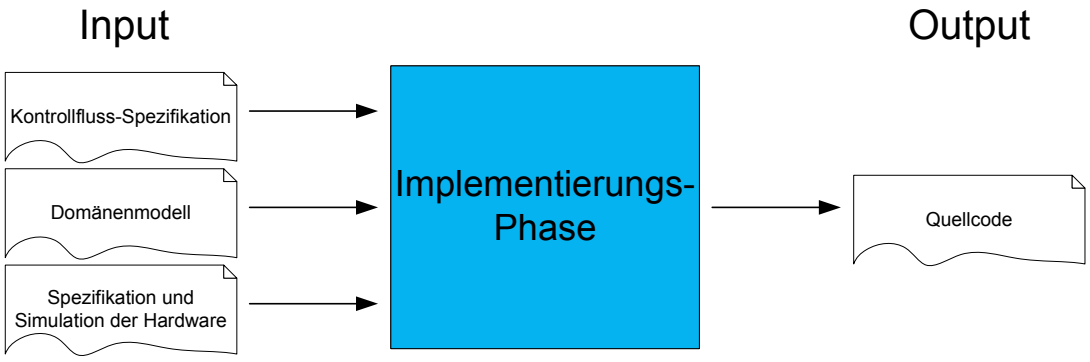



Abbildung 3.4: Ein- und Ausgabenartefakte der Implementierungs-Phase

In der Implementierungs-Phase wird der benötigte Programmcode geschrieben. Das Schreiben von Programmcode ist einerseits zur Interaktion mit der simulierten Hardware nötig, andererseits zur Realisierung von Funktionalität, die nicht oder nur sehr schwer durch Aktivitätsdiagramme abgebildet werden kann. Ein Beispiel für nicht abbildbare Funktionalität ist die Ausgabe einer Sounddatei, um ein akustisches Feedback zu geben.

Erst in dieser Phase wird das System lauffähig und erste Tests können durchgeführt werden. Beim Testen und Debuggen helfen die Step-By-Step-Ausführung und der Breakpoint-Mechanismus der ActiveCharts IDE. Die Visualisierung des Programmflusses erlaubt einen schnellen Überblick über den Zustand des Systems.

#### 3.1.1.4 Test-Phase

In der Test-Phase werden folgende Schritte durchgeführt:

1. Testen der Implementierung mittels der Testfälle

Abbildung 3.5 zeigt einen Überblick über alle an der Test-Phase beteiligten Artefakte, die im Folgenden beschrieben werden.

In der Test-Phase wird getestet, ob das entwickelte System die Anforderungen aus den Anforderungsdokumenten erfüllt. Jetzt liegt ein lauffähiges System vor. Der Test ist daher, im Gegensatz zu der Review-Phase, ein dynamischer Test. Treten Fehler auf, muss wieder in die entsprechenden Phasen zurückgesprungen werden, in denen der Fehler verursacht wurde. Dabei wird in den Praktika als einzige Phase die Review-Phase nicht wiederholt, da davon

ausgegangen wird, dass die Review-Phase zum Aufdecken von Mängeln in der Architektur dient, die in der Test-Phase nicht mehr vorkommen sollten.

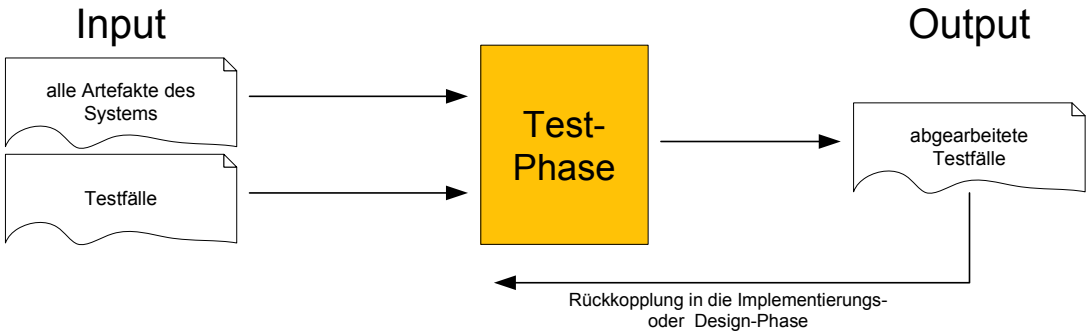



Abbildung 3.5: Ein- und Ausgabenartefakte der Test-Phase

In den Praktika wurde der Rücksprung in die vorherigen Phasen meist nicht komplett durchgeführt, da nur ein begrenzter Zeitraum zur Verfügung stand.

### 3.1.2 Abgeleitetes Vorgehensmodell

Der Entwicklungsprozess aus den Praktika soll im Folgenden als ein an ein rückgekoppeltes Wasserfallmodell angelehntes Vorgehensmodell abstrahiert werden. Das Ziel ist ein einfaches Vorgehensmodell, bei dem klar ist, wann welche Artefakte erzeugt bzw. benötigt werden.

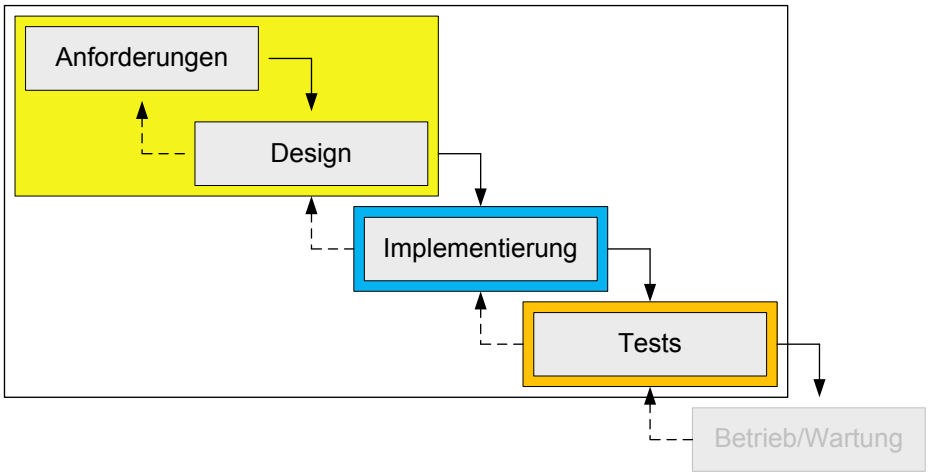



Abbildung 3.6: Verwendete Phasen eines einfachen rückgekoppelten Wasserfallmodells

Abbildung 3.6 zeigt, welche Phasen aus den Praktika als Phasen in einem Wasserfallmodell vorkommen. Die Anforderungs-Phase und die Design-Phase wurden in den Praktika zu einer einzigen Design-Phase zusammengefasst. Implementierungs- und Test-Phase sind eins zu eins übertragbar. Einzig die Betrieb-/Wartungs-Phase entfällt, da die Steuerlogik aus den Praktika nicht in einem realen System zum Einsatz kommt.

Abbildung 3.7 zeigt das abgeleitete Vorgehensmodell und alle Artefakte, die an der Erstellung des Systems beteiligt sind.

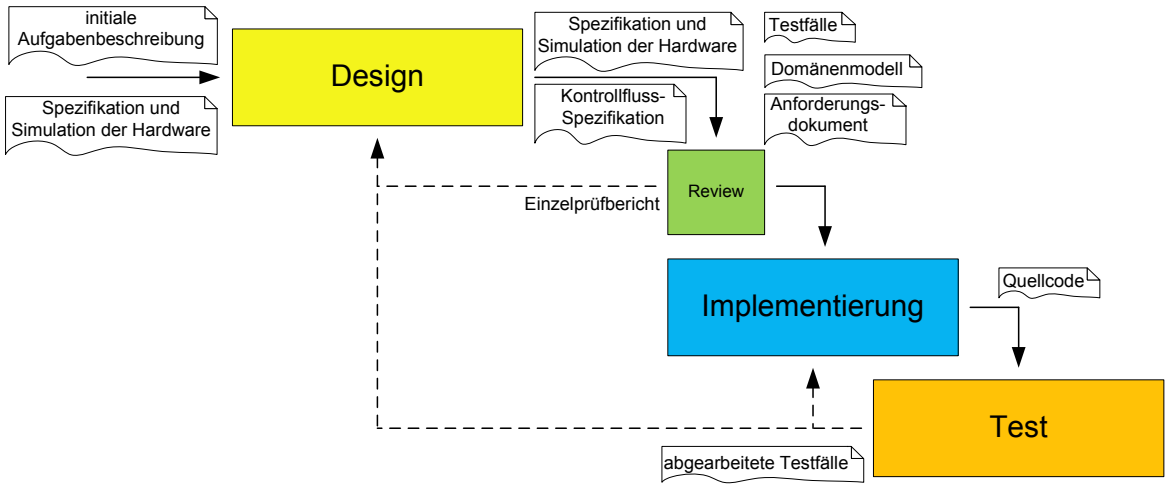



Abbildung 3.7: Abgeleiteter Prozess mit allen verwendeten Artefakten

Das resultierende Vorgehensmodell hat folgende Eigenschaften:

- Es ist ein wasserfallartiges Vorgehensmodell und stellt daher auch die einfachste Form eines Phasenmodells dar (vergleiche [26, Seite 2]).
- Enthält explizit ein Review als statischen Test, d. h. es ist festgelegt, wie entschieden wird und ob eine Rückkopplung in die Design-Phase stattfindet.
- Es ist ein dokumentengetriebenes Modell, da am Ende jeder Phase ein oder mehrere Artefakte als Ergebnis stehen.
- Es ist sequenziell, vorherige Aktivität muss beendet werden, bevor die nächste Aktivität beginnt.
- Es ist großteils Use Case-Driven, aber das Anforderungsdokument enthält auch textuelle Anforderungen, die nicht als Use Case strukturiert sind.

Allerdings besitzt das hier vorgestellte Vorgehensmodell einige Einschränkungen. Es ist exakt auf das Vorgehen aus den Praktika zugeschnitten. Daher ist es weder vollständig, dazu fehlte in den Praktika die Zeit, noch ist es allgemeingültig. Es enthält z. B. keinerlei Management- oder Budget-Aspekte und die Phasen Inbetriebnahme und Wartung sind komplett entfallen. Die Qualität ist zwar ein wichtiger Aspekt, aber durch die Zeitvorgaben, dass der Entwicklungsprozess am Ende des Semesters beendet wird und dass möglichst ein lauffähiges System vorhanden ist, wurde die Qualität weniger streng geprüft als es wünschenswert ist.

Das in Abbildung 3.7 gezeigte Vorgehensmodell ist eine erste Version des Modells. Um seine Tauglichkeit zu zeigen, muss es weiterhin in Praktika verwendet und angepasst werden. Dies kann z. B. durch eine Anwendung des CMMI [33] erreicht werden, der eine sukzessive

Prozessverbesserung durch Rückkopplung nach jeder Verwendung des Prozesses erlaubt. Erst durch die Anwendung kann etwas über die Qualität und Tauglichkeit des Prozesses ausgesagt werden.

## 3.2 Struktur der Anforderungsdokumente

In diesem Abschnitt wird eine Struktur entwickelt, welche die Anforderungen und beliebige andere textuelle Artefakte aus Abschnitt 3.1 aufnehmen können soll. Andere textuellen Artefakte sind z. B. Beschreibungen der Stakeholder, Signale und Hardware-Komponenten oder ein Glossar.

Dazu werden zuerst in 3.2.1 die Anforderungen an ein solches Anforderungsdokument beschrieben und dann in Abschnitt 3.2.2 eine Struktur vorgestellt, welche die Anforderungen erfüllt.

Die zentrale Idee ist, den einzelnen Elementen eine Semantik zu geben. Dadurch wird zwar auf der einen Seite die Allgemeingültigkeit eingeschränkt, aber auf der anderen Seite eine starke Spezialisierung an die ActiveCharts IDE erreicht. Diese Spezialisierung erlaubt eine höhere Robustheit beim Schreiben der Anforderungen und Verwendung der semantischen Zusammenhänge (siehe Abschnitt 3.3 „Traceability“).

### 3.2.1 Anforderungen an ein Anforderungsdokument

Nach IEEE [17] muss ein gutes Anforderungsdokument folgende Anforderungen erfüllen:

1. **Korrektheit**
   Ein Anforderungsdokument ist korrekt, genau dann wenn alle Anforderungen im Dokument von dem zu realisierenden System erfüllt werden sollen.
2. **Eindeutigkeit**
   Ein Anforderungsdokument ist eindeutig, genau dann wenn jede Anforderung im Dokument genau eine Interpretation zulässt.
3. **Vollständigkeit**
   Ein Anforderungsdokument ist vollständig, genau dann wenn jeder Aspekt des gewünschten Systems durch eine Anforderung beschrieben wird.
4. **Konsistenz**
   Ein Anforderungsdokument ist konsistent, genau dann wenn sich keine Anforderungen im Dokument gegenseitig widersprechen und wenn keine Anforderung sich selbst widerspricht.
5. **Sortierbarkeit**
   Die Anforderungen müssen nach Kriterien wie Realisierbarkeit oder Relevanz sortiert

werden.

6. **Prüfbarkeit**
   Ein Anforderungsdokument ist prüfbar, genau dann wenn jede Anforderung im Dokument auf Einhaltung im realisierten System geprüft werden kann.
7. **Änderbarkeit**
   Ein Anforderungsdokument ist änderbar, genau dann wenn Änderungen an den Anforderungen gemacht werden können ohne die Struktur des Anforderungsdokuments zu zerstören oder Inkonsistenzen zu erzeugen.
8. **Verfolgbarkeit**
   Ein Anforderungsdokument ist verfolgbar, genau dann wenn der Ursprung jeder Anforderung bekannt ist. Außerdem muss jede Anforderung bei Änderungen im Dokument und im weiteren Entwicklungsprozess referenzierbar bleiben.

Im nächsten Abschnitt wird eine Struktur vorgestellt, welche die oben genannten Anforderungen erfüllen soll. Eine vollständige Erfüllung der Anforderungen kann mit der Verwendung von natürlichsprachlichem Text nicht erreicht werden, da Anforderungen wie Korrektheit, Eindeutigkeit und Vollständigkeit sich nicht ohne Weiteres prüfen und somit einhalten lassen. Diese Anforderung kann nur eine formale Beschreibungssprache erfüllen (vergleiche [26, Seite 39]), welche die Einhaltung dieser Anforderungen garantieren kann. Allerdings ist das hier nicht gewünscht, da wegen der leichten Verständlichkeit die natürliche Sprache als Beschreibungssprache gewählt wird. Daher kann die Struktur, die im nächsten Abschnitt vorgestellt wird, nur eine unterstützende Funktion haben, d. h. sie kann z. B. die Eindeutigkeit unterstützen, indem ein Glossar verwaltet wird und somit die Verwendung von Synonymen vermieden wird.

### 3.2.2 Struktur für ein Anforderungsdokument

Die Struktur für das Anforderungsdokument wird in Abschnitte unterteilt, die beliebig verschachtelt werden können. Das bedeutet, dass in jedem Abschnitt wiederum entweder beliebig viele Abschnitte oder Elemente enthalten sein können.

Allerdings kann jeder Abschnitt einschränken, welche Abschnitte und welche Elemente er als seine Kinder zulässt. So kann z. B. ein Abschnitt, der ein Glossar darstellt, nur Glossareinträge als Kinder zulassen.

Jedes Element ist von einem Typ und kann über eine Kennung eindeutig identifiziert werden. Der Typ eines Elements gibt an, welche Eigenschaften und welche Semantik es besitzt. Es gibt folgende sieben Typen:

1. Stakeholder
2. Anforderung

3. Glossareintrag
4. Use Case
5. Signal
6. Hardware-Schnittstelle
7. Hardware-Komponente
8. Testfall

Abbildung 3.8 zeigt die Elemente und deren Beziehungen als UML-Klassendiagramm. In den folgenden Abschnitten werden die einzelnen Typen und deren Eigenschaften detailliert beschrieben. Da jedes Element einen Namen und eine Beschreibung hat, werden diese beiden Eigenschaften, im Folgenden Standardeigenschaften genannt, nicht für jeden Typ erneut aufgeführt. Beschreibungen sind natürlichsprachlicher Text und können beliebig viele Referenzen (siehe Abschnitt 3.3 „Traceability“) enthalten.

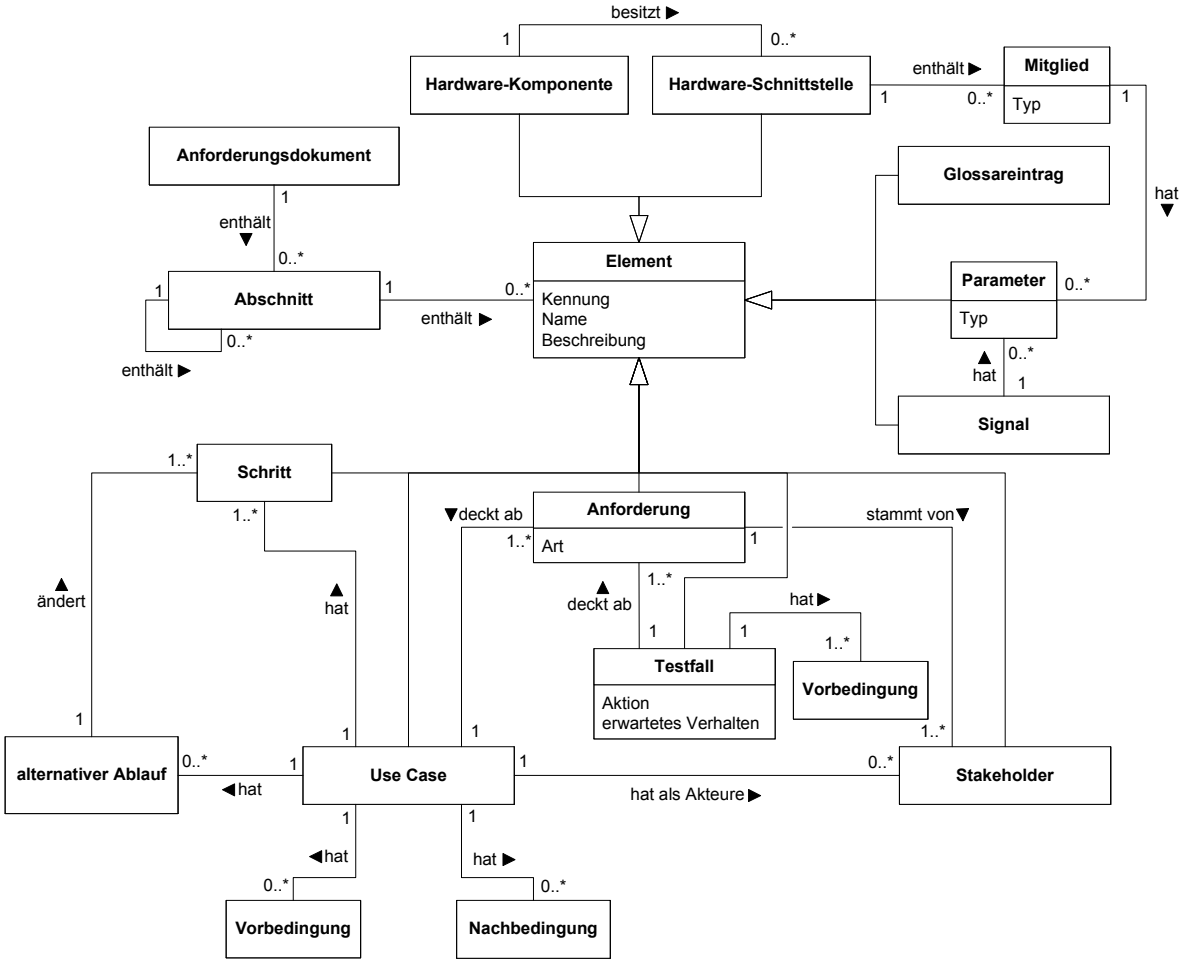



Abbildung 3.8: UML-Modell des Anforderungsdokuments

#### 3.2.2.1 Stakeholder

Unter Stakeholder werden alle in irgendeiner Weise an dem Projekt beteiligten Personen zusammengefasst. Wie Ian Alexander et al. in [2, Seite 16] erwähnen, ist es nicht einfach, alle Stakeholder zu finden. Aber der Projekterfolg ist maßgeblich abhängig von dem Auffinden aller Stakeholder, da sie mit dem System arbeiten müssen und von ihnen die Akzeptanz abhängt.

| **Stakeholder** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Der Typ Stakeholder besitzt nur die Standardeigenschaften. | |

#### 3.2.2.2 Anforderung

Dieses Element beschreibt eine Anforderung. Dadurch, dass Abschnitte und Elemente beliebig verschachtelt werden können, können die Anforderungen hierarchisch strukturiert werden. Somit kann eine Anforderung hierarchisch verfeinert werden (vergleiche [2, Seite 66]). Die Grundidee hinter dem Anforderungs-Element ist dieselbe wie bei den Snowcards der Volere Vorlagen [36], allerdings in vereinfachter Form. Bei den Volere Vorlagen enthält jede Snowcard eine Anforderung mit einer Beschreibung und zusätzlichen Informationen, wie z. B. einer Begründung für die Existenz der Anforderung und deren Beziehungen. Mögliche Beziehungen sind Konflikte mit anderen Anforderungen und Abhängigkeiten von anderen Anforderungen.

| **Anforderung** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Art | Ist entweder funktional oder nicht-funktional. |
| Quelle | Eine Liste von Referenzen auf die Stakeholder, die der Ursprung der Anforderung sind. |

#### 3.2.2.3 Glossareintrag

Ein Glossareintrag dient zur Definition eines Begriffes. Der Glossareintrag ist ein wichtiges Hilfsmittel, um Synonyme und somit Mehrdeutigkeiten in den textuellen Beschreibungen zu vermeiden. Das Glossar dient außerdem dazu, dass alle Projektbeteiligten den gleichen Wortschatz verwenden. Das ist vor allem bei Fachfremdheit in einer Domäne wichtig.

| **Glossareintrag** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Der Typ Glossareintrag besitzt nur die Standardeigenschaften. | |

#### 3.2.2.4 Use Case

Ein Use Case beschreibt eine Interaktion mit dem System. Ein Use Case ist stärker formalisiert als eine textuell formulierte Anforderung.

<table>
<tr><th colspan="2">Use Case</th></tr>
<tr><th>Eigenschaft</th><th>Beschreibung</th></tr>
<tr><td>Anforderungen</td><td>Eine Liste von Referenzen auf die Anforderungen, die durch den Use Case teilweise oder ganz abgedeckt werden.</td></tr>
<tr><td>Akteure</td><td>Eine Liste von Referenzen der an dem Use Case beteiligten Stakeholder und Hardware-Komponenten.</td></tr>
<tr><td>Vorbedingungen</td><td>Eine Liste von natürlichsprachlichen Vorbedingungen für den Use Case.</td></tr>
<tr><td>Nachbedingungen</td><td>Eine Liste von natürlichsprachlichen Nachbedingungen.</td></tr>
<tr><td>Schritte</td><td>Eine Liste von Schritten, wobei jeder Schritt folgende Eigenschaften hat:
<table>
<tr><th>Eigenschaft</th><th>Beschreibung</th></tr>
<tr><td>Akteur</td><td>Eine Referenz auf einen Stakeholder oder eine Hardware-Komponente, der die Aktion ausführt.</td></tr>
<tr><td>Reakteur</td><td>Eine Referenz auf einen Stakeholder oder eine Hardware-Komponente, mit Hilfe dessen die Aktion ausgeführt wird.</td></tr>
<tr><td>Beschreibung</td><td>Eine Beschreibung der Aktionen, die in dem Schritt durchgeführt werden.</td></tr>
</table></td></tr>
<tr><td>alternative Abläufe</td><td>Eine Liste von alternativen Abläufen, wobei jeder alternative Ablauf folgende Eigenschaften hat:
<table>
<tr><th>Eigenschaft</th><th>Beschreibung</th></tr>
<tr><td>Ändert</td><td>Eine Referenz auf einen Stakeholder. Dieser Stakeholder ist dafür verantwortlich, dass ein alternativer Ablauf zustande kommt.</td></tr>
<tr><td>Beschreibung</td><td>Eine Liste von Referenzen auf die Schritte des Use Case, die von dem alternativen Ablauf betroffen sind.</td></tr>
</table></td></tr>
</table>

#### 3.2.2.5 Signal

Beschreibt ein Signal der ActiveCharts IDE. Korrespondiert eins zu eins mit einer Klasse im Klassendiagramm mit dem Stereotyp „Signal“. Allerdings ist der Typ des Parameters

von der verwendeten Programmiersprache abhängig. So heißt der Datentyp, der einen booleschen Wert repräsentiert, in Java „boolean“ und in C# „bool“. Eine mögliche Lösung ist die Schaffung eines eigenen generischen Typensystems. Dann kann je nach verwendeter Programmiersprache dementsprechend von dem generischen Typensystem in das Typensystem der Programmiersprache konvertiert werden.

| **Signal** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Parameter | Eine Liste von Parametern, wobei jeder Parameter folgende Eigenschaften hat: |

| **Eigenschaft** | **Beschreibung** |
|---|---|
| Typ | Der Typ des Parameters. |

#### 3.2.2.6 Hardware-Schnittstelle

Hardware-Schnittstellen stellen definierte Kommunikationspunkte mit der Hardware dar. Sie werden benötigt, da die Hardware in der ActiveCharts IDE nur simuliert wird und nicht real vorhanden ist.

| **Hardware-Schnittstelle** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Mitglieder | Eine Liste von Mitgliedern, wobei jedes Mitglied folgende Eigenschaften hat: |

| **Eigenschaft** | **Beschreibung** |
|---|---|
| Typ | Ist entweder eine Methode oder ein Ereignis. |
| Parameter | Eine Liste von Parametern, wobei jeder Parameter folgende Eigenschaften hat: |

| **Eigenschaft** | **Beschreibung** |
|---|---|
| Typ | Der Typ des Parameters. |

#### 3.2.2.7 Hardware-Komponente

Stellt eine physisch vorhandene Hardware-Komponente des Systems dar. Kann als Aktor oder Reakteur in einem Use Cases dienen. Eine Hardware-Komponente hat beliebig viele Schnittstellen, über die mit ihr kommuniziert werden kann.

| **Hardware-Komponente** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Schnittstellen | Eine Liste von Referenzen auf die Hardware-Schnittstellen der Komponente. |

#### 3.2.2.8 Testfall

Ein Testfall dient dazu, ein gewünschtes Verhalten bzw. eine gewünschte Funktion des Systems zu prüfen. Im Idealfall sollte jede Anforderung durch einen oder mehrere Testfälle abgedeckt werden.

| **Testfall** | |
|---|---|
| **Eigenschaft** | **Beschreibung** |
| Vorbedingungen | Beschreibt den Zustand des Systems vor der Durchführung der Aktion. |
| Aktion | Die Aktion, welche mit dem System durchgeführt werden soll. |
| erwartetes Verhalten | Beschreibt das Verhalten, welches nach der Aktion eintreten muss wenn der Testfall erfolgreich sein soll. |
| Anforderung | Eine Referenz auf eine Anforderung, die durch den Testfall geprüft wird. |

### 3.2.3 Erfüllung der Anforderungen

Im Folgenden wird diskutiert, ob die Struktur aus 3.2.2 die Anforderungen aus 3.2.1 ausreichend unterstützt. Wie schon in 3.2.1 erwähnt, ist es nicht möglich, einen formalen Nachweis für die Erfüllung zu bringen.

- **Korrektheit**
  Die Struktur kann die Einhaltung der Korrektheit nicht erzwingen. Es kann nur der Überblick über die Anforderungen erleichtert werden, da sie hierarchisch gegliedert und somit übersichtlicher dargestellt sind.
- **Eindeutigkeit, Vollständigkeit, Konsistenz**
  Die Eindeutigkeit, Vollständigkeit und Konsistenz kann in bestimmten Fällen durch die Verlinkung von Anforderungen mit dem Modell (siehe nächster Abschnitt) erreicht werden. Aber auch hier ist höchstens eine Unterstützung möglich, kein Zwang.
- **Sortierbarkeit**
  Die freie Sortierbarkeit ist gegeben. Die Struktur kann um Sichten, welche Elemente nach Kriterien anzeigen, erweitert werden. In einer Sicht können Elemente stehen, die nicht in einem gemeinsamen Teilzweig der Struktur stehen.
- **Prüfbarkeit**
  Die Prüfbarkeit kann durch das Umwandeln der Anforderungen in Testfälle ermöglicht werden. Durch die Semantik der Elemente kann man auch Anforderungen von anderen Elementen unterscheiden, die nicht direkt prüfbar sind.
- **Änderbarkeit**
  Die Änderbarkeit wird durch die Kapselung in einzelne Elemente erleichtert. Dies stellt einen Vorteil im Vergleich zu einem Anforderungsdokument in einer Textverarbeitung

dar. Die Verlinkungen (siehe nächster Abschnitt) können helfen, Inkonsistenzen zu vermeiden.

- **Verfolgbarkeit**
  Auch diese Eigenschaft kann durch die Verlinkungen, die im nächsten Abschnitt vorgestellt werden, realisiert werden.

Wie hier aufgezeigt wird, kann die Struktur es ermöglichen, dass alle Anforderungen erfüllt werden. Die Struktur stellt einen Kompromiss zwischen einer Formalisierung (Verwendung einer festen Struktur) und der Lesbarkeit (Verwendung von natürlicher Sprache) dar. Wie in [26, Seite 50] beschrieben, muss zwischen einer formalen und einer natürlichsprachlichen Beschreibung gewählt werden. Es wird versucht, in der Struktur eine textuelle Beschreibung mit möglichst viel Semantik zu verbinden, und somit ein im Sinne der IEEE-Anforderungen aus Abschnitt 3.2.1 „besseres“ Anforderungsdokument zu ermöglichen.

## 3.3 Traceability

Dieser Abschnitt stellt die Untersuchung der Traceability dar. Es wird dabei sowohl untersucht, wie die Traceability realisiert werden kann, als auch zwischen welchen Elementen eine Traceability stattfinden kann.

### 3.3.1 Motivation

In der Softwareentwicklung entstehen in jeder Phase Artefakte, die entweder aus Artefakten einer vorherigen Phase abgeleitet werden, oder wie in der Anforderungs-Phase, von außen vorgegeben werden. Ohne Traceability besteht selbst bei kleineren Projekten das Problem, dass Inkonsistenzen zwischen den abgeleiteten Artefakten entstehen können. Außerdem sind Änderungen fehleranfällig, die mehrere Artefakte betreffen. Denn dabei können einzelne Artefakte versehentlich vergessen werden und es entstehen wiederum Inkonsistenzen.

Im folgenden Abschnitt soll eine Traceability-Architektur für den in Abschnitt 3.1 hergeleiteten Entwicklungsprozess mit der ActiveCharts IDE vorgestellt werden. Dabei liegt der Fokus auf der in Abschnitt 3.2 entwickelten Struktur zur Verwaltung der Anforderungen und den in der ActiveCharts IDE verwenden Diagrammen.

Es gibt viele verschiedene Möglichkeiten Traceability zu realisieren. [1] nennt als Beispiele Traceability-Matrizen, Datenbanken, Hyperlinks, Graphen-basierte Ansätze und formale mathematische Ansätze. In dieser Arbeit wird ein Hyperlink-basierter Ansatz gewählt, da Hyperlinks sehr intuitiv sind, sich sehr gut für moderne GUIs eignen und selbst bei vielen Artefakten übersichtlich bleiben. Im Folgenden wird statt des Begriffs Hyperlink der Begriff Link verwendet. In der hier vorgestellten Architektur gibt es sowohl automatisch als auch

manuell erzeugte Links. Die automatisch erzeugten Links ergeben sich durch die Ausnutzung von semantischen Zusammenhängen.

Im Folgenden wird der Begriff Editor verwendet. Ein Editor ist ein abstraktes Konzept für eine Bearbeitungs- und Darstellungsmöglichkeit eines Artefakt-Typs. Es gibt z. B. einen Editor für die Anforderungsstruktur, die Aktivitätsdiagramme und die Klassendiagramme. Ein weiteres Beispiel für einen konkreten Editor ist OpenOffice. Es dient dazu, Text-Artefakte jeder Art zu editieren. Außerdem sind die Editoren die GUI-Elemente, die zur Darstellung und Erstellung der Links dienen.

### 3.3.2 Traceability-Architektur

Jedes Element, das als Quelle oder Ziel eines Links fungieren kann, muss folgende Eigenschaften erfüllen:

1. **eindeutige Identifizierbarkeit**
   Der Bezeichner des Elements muss eindeutig unter allen Elementen sein. Es bietet sich z. B. ein Bezeichner nach RFC 4122: A Universally Unique IDentifier [20] an.
2. **Versionierung**
   Bei jeder Änderung an einem Element muss sich die Version erhöhen.
3. **Name**
   Dient zur Darstellung des Elements, ohne den genauen Inhalt zu kennen.
4. **Beschreibung**
   Eine optionale ausführlichere Beschreibung des Elements.

Eigenschaft 1 und 2 ermöglichen eine eindeutige Identifikation eines Elements über Raum und Zeit. Durch Eigenschaft 3 und 4 wird eine menschenlesbare Darstellung von Elementen ermöglicht, die an einem Link beteiligt sind.

Um von dem Typ der Linkquelle und des Linkziels zu abstrahieren, wird ein sogenannter Element-Deskriptor eingeführt. Daher muss jeder Link zwei Element-Deskriptoren speichern, einen als Quelle und einen als Ziel. Dadurch werden die Links unabhängig von dem Typ der verlinkten Artefakte. Die Element-Deskriptoren müssen die vier oben genannten Eigenschaften erfüllen.

Abbildung 3.9 zeigt einen Überblick über die an der Architektur beteiligten Elemente. Im Folgenden wird jeder der farbig unterlegten Teile erläutert.

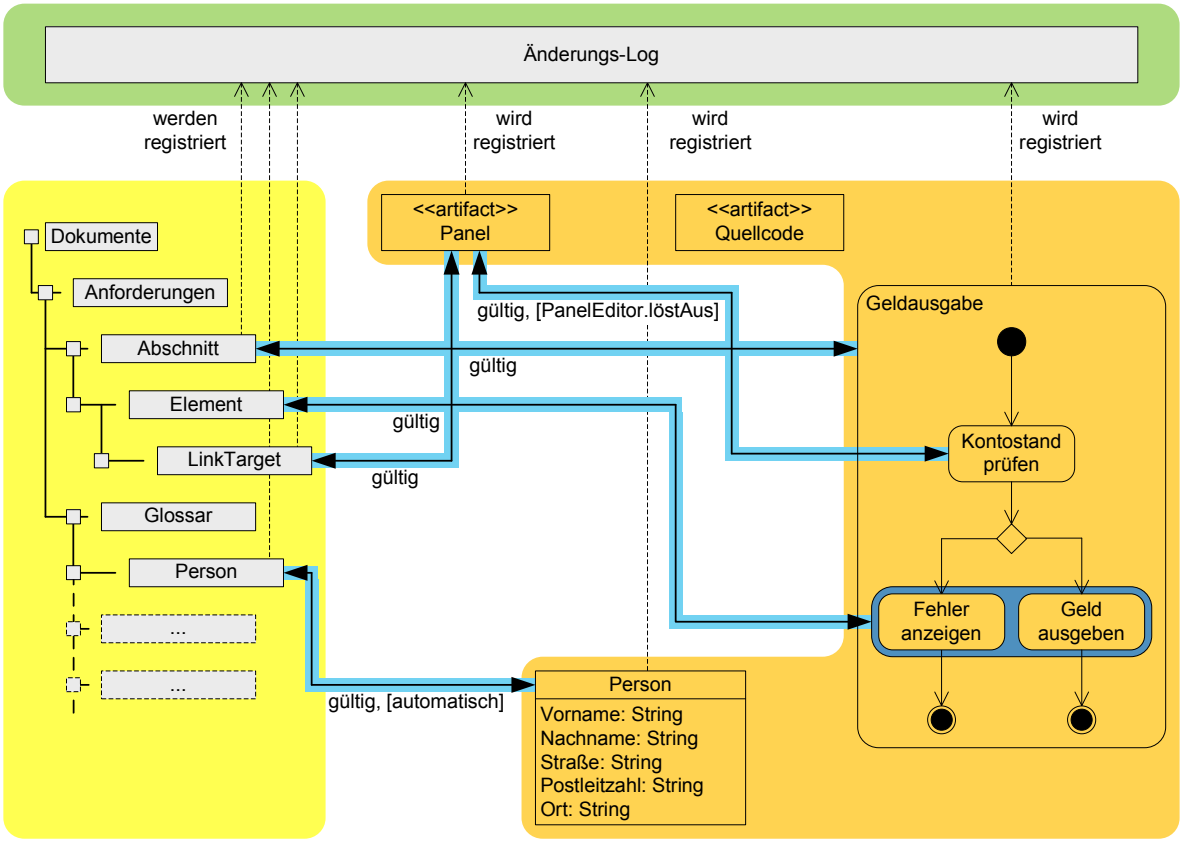



Abbildung 3.9: Überblick über die an der Traceability-Architektur beteiligten Elemente

#### 3.3.2.1 Links

Die blau hinterlegten Pfeile stellen Links dar. Alle Links sind bidirektional, d. h. es kann sowohl von der Linkquelle zum Ziel als auch andersherum navigiert werden. Ein Link hat genau einen von drei Zuständen:

1. **gültig**
   Diesen Zustand hat jeder Link, nachdem er erstellt wurde und wenn er keinen der anderen zwei Zustände hat.
2. **ungültig**
   Die Linkquelle oder das Linkziel des Links wurde gelöscht.
3. **geändert**
   Die Versionsnummer der Linkquelle oder des Linkziels hat sich verändert.

Die drei Zustände sind fest und es können keine weiteren definiert werden. Hat ein Link den Zustand ungültig oder geändert, kann sein Zustand durch das Änderungs-Log oder einen Editor entweder auf gültig gesetzt oder der Link komplett gelöscht werden. Bei Links mit dem Zustand ungültig kann das Löschen in Frage kommen, da der Link ohne die gelöschte

Linkquelle oder das gelöschte Linkziel seine Existenzberechtigung verloren hat.

Zusätzlich kann jeder Link mit einer beliebigen Anzahl an Tags versehen werden. Ein Tag besteht aus einem (eindeutigen) Namespace und einem Namen. Der Namespace dient dazu, Kollisionen von Tags mit gleichem Namen aber unterschiedlicher Herkunft zu vermeiden. Es gibt pro Editor einen Namespace. Sowohl die Tags als auch die Namespaces sind frei wählbare Zeichenketten. In 3.9 sind die Tags in eckigen Klammern bei den Links angegeben. So wurde der Link zwischen dem Element Person aus der Anforderungsstruktur und der Klasse Person in einem Klassendiagramm mit dem Tag „[automatisch]“ versehen, da er nicht von Hand erstellt wurde, sondern automatisch (siehe Abschnitt 3.3.3.2). Der Tag besitzt keinen Namespace, da der Tag „[automatisch]“ global gültig sein soll.

#### 3.3.2.2 Anforderungs-Struktur

Der gelb hinterlegte Bereich in Abbildung 3.9 stellt die Struktur aus Abschnitt 3.2 dar. Sie enthält die Dokumentation in Form von Abschnitten und Elementen. Jeder Abschnitt und jedes Element kann als Ziel oder Quelle eines Links dienen. Außerdem kann jeder Abschnitt und jedes Element beliebig viele LinkTargets enthalten. LinkTargets haben eine noch feinere Granularität als Elemente und können z. B. durch ein Wort im Text repräsentiert werden (siehe Abbildung 3.10 für ein Beispiel eines LinkTargets). Der blau markierte Text in Abbildung 3.10 in der Anforderung, die als Element repräsentiert wird, stellt ein in dem Anforderungs-Element enthaltenes LinkTarget dar. Auch LinkTargets können als Quelle oder Ziel beliebig vieler Links dienen. Im Beispiel ist das LinkTarget die Quelle eines Links auf ein Element (orange markiert) eines Aktivitätsdiagramms. Sowohl das LinkTarget als auch der Timer müssen die vier Anforderungen am Anfang des Abschnittes erfüllen, indem sie einen Element-Deskriptor zur Verfügung stellen.

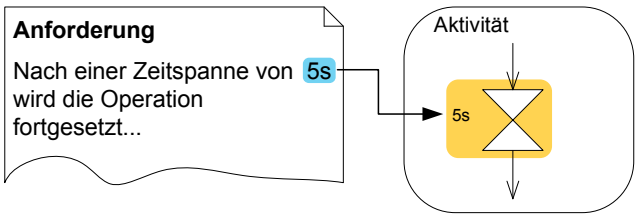



Abbildung 3.10: Beispiel für ein LinkTarget

#### 3.3.2.3 Beliebige Artefakte

Im orange hinterlegten Teil sind mehrere Artefakte abgebildet. Der Fokus liegt auf den Aktivitäts- und Klassendiagrammen, da diese beiden Artefakte die wichtigsten bei der Entwicklung mit der ActiveCharts IDE sind. Sowohl als Linkquelle als auch als Linkziel sind

einzelne Elemente oder Gruppen von Elementen möglich. Das ist z. B. bei Aktivitätsdiagrammen sehr nützlich, da eine Anforderung meist nicht nur mit einer Kante oder einem Knoten realisierbar ist, sondern von einer Gruppe von Kanten und Knoten. So wird die Anforderung aus Abbildung 3.11 durch den Entscheidungsknoten, den zwei Bedingungen an den Kanten und den zwei Aktionen erfüllt.

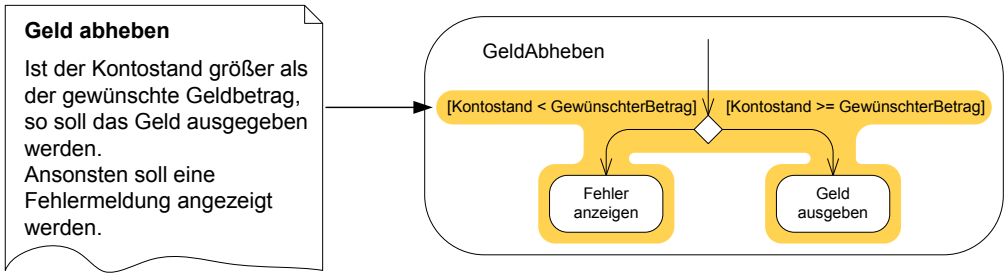



Abbildung 3.11: Beispiel für eine Gruppe als Linkziel

Hierbei wird eine Gruppe von Elementen eines Artefakts als einzelner Element-Deskriptor repräsentiert. Es ist die Aufgabe des jeweiligen Editors, intern eine Abbildung des Element-Deskriptors auf die Gruppe von Elementen zu verwalten. Eine andere Möglichkeit ist, eine Menge von Element-Deskriptoren als Linkquelle oder Linkziel zu erlauben. Es wurde aber die erste Möglichkeit gewählt, da es die Speicherung der Quelle und des Ziels im Link vereinfacht und die Gruppen nach außen transparent macht. Der Nachteil dieses Vorgehens ist, dass die Editoren intern die Gruppen verwalten müssen.

Des Weiteren ist sogar eine Verlinkung in den Quellcode denkbar. Die Aktionen in den Aktivitäten können über ihre Namen der entsprechenden Methode zugeordnet werden. Wenn an beliebige Stellen in den Quellcode verlinkt werden können soll, dann treten zwei Probleme auf. Das erste Problem ist die Markierung als Linkziel. Eine Möglichkeit ist, den Link als Kommentar in den Quellcode einzufügen. Allerdings gibt es dann Probleme, wenn der Link von Hand geändert wird oder per Copy & Paste einfach an einer anderen Stelle eingefügt wird. Eine mögliche Lösung hierfür ist die Einführung spezieller geschützter Bereiche, die nicht von Hand editiert werden können, wie sie z. B. von dem GUI-Builder von Netbeans verwendet werden. Allerdings sollte das Verlinken möglichst transparent sein, was hier nicht der Fall ist. Das zweite Problem ist die Frage der Granularität. Wie fein granular sollen die zu verlinkenden Elemente sein? Klassen und Methoden werden für die Verlinkung zwischen den Aktionen und ihrer Realisierung benötigt. Aber gibt es auch Fälle, in denen z. B. mit Variablen oder einzelnen Zeilen verlinkt werden muss?

Auch eine Verlinkung von Elementen des Panel Editors ist sinnvoll. So zeigt Abbildung 3.9 die Verlinkung einer Aktion mit einem Element in einem Panel. Außerdem wurde der Link mit dem Tag „[PanelEditor.löstAus]“ versehen um zusätzliche Informationen darüber zu speichern, in welcher Beziehung die beiden Elemente stehen. Dabei ist „PanelEditor“ der

Namespace und „löstAus“ der eigentliche Tag.

#### 3.3.2.4 Änderungs-Log

Das Änderungs-Log (vergleiche [7]) ist eine zentrale Komponente, bei der jedes Element, das als Linkquelle oder als Linkziel verwendet wird, registriert wird. Es liegt somit ein Publisher-Subscriber-Pattern vor (vergleiche [5, 338]). Die verlinkten Elemente sind die Publisher ihrer Änderungen und das Änderungs-Log ist der Subscriber.

Das Änderungs-Log hat zwei zentrale Aufgaben. Zum ersten sorgt es für die Aktualisierung des Linkzustandes. Dadurch, dass jedes Linkziel und jede Linkquelle bei ihm registriert ist, können sie eine Nachricht schicken, wenn sich ihre Versionsnummer geändert hat oder wenn sie gelöscht wurden. Die zweite Aufgabe ist die Bereitstellung eines Logs mit allen Änderungen. Dieses Log dient auf der einen Seite dazu, nötige Änderungen an Elementen darzustellen, die durch die Änderung oder Löschung des Elements auf der anderen Seite des Links nötig wurden. Und auf der anderen Seite können sich wiederum Interessenten bei dem Änderungs-Log registrieren um z. B. eine nötige Änderung eines Elements über die eigene GUI anzeigen zu können. Das bedeutet, dass zweimal eine Publisher-Subscriber Beziehung vorliegt: einmal alle Linkquellen und alle Linkziele als Publisher zu dem Änderungs-Log als Subscriber und alle am Log interessierten GUIs als Subscriber zu dem Änderungs-Log als Publisher aller Änderungen.

#### 3.3.2.5 Fazit

Die hier vorgestellte Architektur erlaubt eine umfassende Verlinkung beliebiger Elemente und Gruppen von Elementen. Die Zustände der Links sind mit drei Zuständen sehr einfach gehalten. Aber durch die frei wählbaren Tags wird dennoch eine hohe Anpassbarkeit erreicht, da unbekannte Tags einfach ignoriert werden können. Ein weiterer Vorteil ist, dass eine globale Änderungsverfolgung über alle Artefakte, die zur Entwicklung mit ActiveCharts IDE benötigt werden, möglich ist.

Allerdings gibt es auch Nachteile. So kann bei einer sehr großen Anzahl an Ereignissen in dem Log des Änderungs-Log der Überblick verloren gehen. Das Problem kann durch die Einführung von Gruppen, also einer Hierarchisierung der Ereignisse, oder von Prioritäten (vergleiche [7]) einzelner Ereignisse abgemildert werden. Des Weiteren kann die Art der Änderung an einem Artefakt konkretisiert werden. Takahashi et al. [28] führen dazu „Types of Changes“ ein. Sie identifizieren die Änderungsarten Aufteilen, Ersetzen, Hinzufügen, Entfernen, Invertieren und Portieren. Dadurch, dass die Änderung einen konkreten Typ bekommt, ist es leichter für den Entwickler, die Folgen der Änderungen auf die Links und die verlinkten Artefakte abzuschätzen.

Ein weiterer Nachteil ist, dass die Verlinkung explizit von allen Editoren unterstützt werden muss, da die Element-Deskriptoren von den Editoren bereitgestellt werden müssen. Des Weiteren muss es in jedem Editor eine Möglichkeit geben, ein verlinktes Element als verlinkt anzuzeigen und dem Link zu folgen. Außerdem müssen die Links z. B. mittels Drag & Drop erzeugt werden können.

### 3.3.3 Arten der Verlinkung

Wie schon in Abschnitt 3.3.1 erwähnt, können Links sowohl automatisch als auch manuell erzeugt werden. In diesem Abschnitt wird die Erstellung der Links genauer untersucht. Dabei werden interne und externe Links unterschieden. Interne Links sind Links, welche zwei Elemente aus der Anforderungs-Struktur miteinander verlinken. Externe Links hingegen sind Links zwischen einem Element der Anforderungs-Struktur und einem Element eines beliebigen Artefaktes.

In Abschnitt 3.3.3.1 werden zuerst die manuell erstellten Links besprochen und in 3.3.3.2 die automatisch erzeugten Links.

#### 3.3.3.1 Manuell erzeugte Links

Manuell erzeugte Links werden mittels der GUI erzeugt. Die Art und Weise ist abhängig von dem Look & Feel der GUI. Eine oft verwendete Methode zur Erstellung von Links ist Drag & Drop, da es sehr intuitiv ist und der Semantik eines Links entspricht, indem zwei Dinge miteinander verbunden werden. In Abbildung 3.12 wird eine Aktion aus einem Aktivitätsdiagramm mit einer Anforderung aus der Dokumenten-Struktur per Drag & Drop verknüpft. Eine andere Möglichkeit ist die Auswahl eines Linkziels aus einer Liste oder aus einem Baum. Bei der Verwendung einer Liste kann es vorkommen, dass die Anzahl der Linkziele sehr groß und dadurch sehr unübersichtlich wird. Daher bietet sich die Verwendung eines Baumes an.

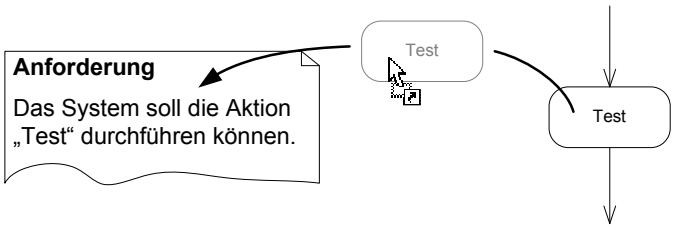



Abbildung 3.12: Schematische Darstellung eine Drag & Drop-Vorgangs

Da es sich bei der vorgestellten Traceability-Architektur um eine Lösung für alle Editoren handelt, bietet es sich an, die Verlinkung zu verallgemeinern, um auf der einen Seite eine konsistente Bedienbarkeit und auf der anderen Seite einen reduzierten Implementierungsaufwand zu erreichen.

Bei manuellen Links muss keine Unterscheidung zwischen internen und externen Links gemacht werden, da sie nicht abhängig von der Semantik sind.

Allerdings haben manuell erzeugte Links den Nachteil, dass sie explizit vom Benutzer erzeugt werden müssen. Das stellt einen nicht unbedeutenden Arbeitsaufwand dar. Es kann vorkommen, dass es mehr Traceability-Links als Anforderungen gibt.

#### 3.3.3.2 Automatisch erzeugte Links

Automatisch erzeugte Links sind sehr wichtig, da sie wie gewünscht, die Robustheit des Entwicklungsprozesses erhöhen ohne mehr Aufwand für den Entwickler zu bedeuten.

Eine automatisierte Erzeugung von Links wird in der vorgestellten Architektur durch die Tatsache erleichtert, dass es sich nicht um ein allgemeines Traceability-Framework handelt, sondern um eine explizit auf die Domäne der ActiveCharts IDE zugeschnittene Lösung. Zusätzlich sind sowohl der verwendete Entwicklungsprozess (vergleiche Abschnitt 3.1) als auch die vorkommenden Artefakte vorgegebenen (vergleiche Abschnitt 3.2.2). Denn erst wenn die Semantik der Artefakte bekannt ist, kann automatisch verlinkt werden.

Allerdings besteht die Gefahr, dass Links automatisch erzeugt werden, die nicht vom Entwickler gewünscht sind. Es gilt abzuwägen, ob in der GUI eine Bestätigung für jeden automatisch erzeugten Link eingeholt werden soll. Das würde aber die Vorteile wieder schmälern, da sehr viele Nachfragen beantwortet werden müssten. Es bietet sich ein Kompromiss an, der nur bei bestimmten automatischen Links eine Bestätigung anfordert, die sich als fehleranfällig herausgestellt haben.

Durch eine Markierung dieser Links mit einem „[automatisch]“ Tag kann auch eine automatische Synchronisation der verlinkten Inhalte realisiert werden. Diese Änderungen werden nicht mehr in das Änderungs-Log geschrieben, sondern können automatisch aktualisiert werden.

Zuerst werden im Folgenden einige Möglichkeiten für interne automatisch erzeugte Links und dann für externe automatisch erzeugte Links vorgestellt.

##### Automatisch erzeugte interne Links

Ein Beispiel für automatisch erzeugte interne Links sind Links auf Glossareinträge. Durch eine Volltextsuche kann für jedes Wort geprüft werden, ob es im Glossar vorkommt. Wenn ja, kann ein automatischer Link erzeugt werden. Probleme gibt es dabei allerdings bei Wörtern, die im Text in einer anderen Form als im Glossar vorkommen, z. B. im Text im Plural und im Glossar im Singular.

Wenn im Glossar die Möglichkeit besteht, Synonyme zu jedem Eintrag zu definieren, kann erreicht werden, dass eine Warnung ausgegeben wird, wenn ein Synonym anstatt des im Glossar definierten Hauptterms verwendet wird. Intern kann das z. B. mit einem Link realisiert wer-

den, der ein Tag „[Synonym]“ erhält.

Auch Homonyme, Worte die für verschiedene Begriffe stehen, können behandelt werden, indem mehrere Glossareinträge mit demselben Term erlaubt werden. Bei einer Verlinkung muss dann entschieden werden, welcher der Begriffe gemeint ist.

Auch Anforderungen, Use Cases und Signale können im Text automatisch verlinkt werden. Hier liegt der Vorteil vor allem in einer leichteren Navigierbarkeit und einem vereinfachten Überblick über die Anforderungen. Abbildung 3.13 zeigt ein Beispiel für eine automatische Verlinkung von Text in einer Anforderung mit einem Use Case. Allerdings besteht hier das gleiche Problem mit unterschiedlichen Wortformen wie bei den Verlinkungen mit dem Glossar.

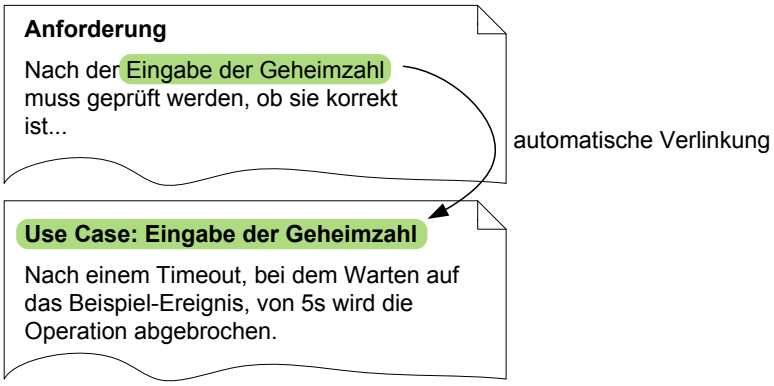



Abbildung 3.13: Beispiel einer internen automatische Verlinkung

Zusätzlich kann zu jeder Anforderung ein entsprechender Testfall, oder zu jedem Testfall eine entsprechende Anforderung, generiert und verlinkt werden.

### Automatisch erzeugte externe Links

Beispielsweise können Anforderungen oder Use Cases automatisch mit Aktivitäten mit gleichen Namen verlinkt werden.

Externe automatische Links können allerdings nur generiert werden, wenn die Semantik der Artefakte und vor allem die Semantik der Elemente in den Artefakten bekannt ist. Deshalb wird im Folgenden ein kombinierter Ansatz vorgestellt, der nicht auf die Analyse der Semantik von bestehenden Artefakten angewiesen ist, sondern mit Hilfe von Templates und Informationen aus der Anforderungs-Struktur automatisch Teile von Artefakten generiert und verlinkt.

Sobald ein Element oder Artefakt aus Tabelle 3.1 erzeugt wird, kann das dazu passende Gegenstück in der Anforderungs-Struktur oder in den Artefakten automatisch generiert und verlinkt werden.

| | Element in der Anforderungs-Struktur | Artefakt oder Teil eines Artefakts |
|---|---|---|
| 1. | Signal | Klasse mit Signal Stereotyp |
| 2. | Glossareintrag | Klasse |
| 3. | Hardware-Schnittstelle | Klasse |

Tabelle 3.1: Automatisch erzeug- und verlinkbare Elemente/Artefakte

Ein Beispiel eines automatisch generierten und verlinkten Signals zeigt Abbildung 3.14. Aus der Beschreibung des Signals in der Anforderungs-Struktur wird direkt eine Klasse in einem Klassendiagramm generiert und verlinkt. Änderungen, wie z. B. die Änderung des Namens eines Parameters in den Anforderungen, können automatisch in das Klassendiagramm übernommen werden. Das gilt auch für die entgegengesetzte Richtung, wenn z. B. im Klassendiagramm der Name geändert wird. Allerdings müssen bei der Umsetzung der Typen der Parameter die unterschiedlichen Typensysteme der Zielprogrammiersprachen beachtet werden (vergleiche Abschnitt 3.2.2.5, Beschreibung des Elements Signal der Anforderungs-Struktur).

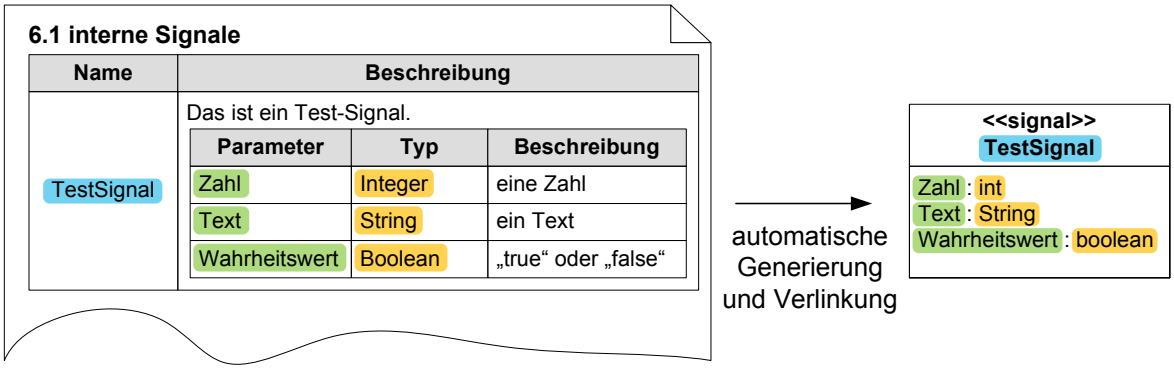



Abbildung 3.14: Automatisch generiertes und verlinktes Signal

Es können auch fest vorgegebene Formulierungen verwendet werden, ähnlich der linguistischen Analyse der Sophist Group [30, Seite 227]. In den vorgegebenen Schablonen können nur bestimmte Teile geändert werden. Die ausgefüllten Schablonen werden dann mittels eines Templates generiert und verlinkt. Dies ist für alle Artefakte möglich, die in der Anforderungs-Struktur textuell beschrieben werden. Es muss ein Repository mit Text-Schablonen und Realisierungstemplates erstellt werden.

Zwei Beispiele mit den dazugehörigen generierten Aktivitätsdiagramm-Fragmenten zeigen Abbildung 3.15 und 3.16. Änderungen sind nur an den farbig unterlegten Textteilen möglich. In Abbildung 3.15 wirken sich Änderungen an dem blau und grün unterlegten Text direkt auf die Aktivitätsdiagramme aus. Der gelb hinterlegte Text dient lediglich zur Dokumentation. In Abbildung 3.16 wirken sich ebenfalls alle farbig unterlegten Texte direkt auf das generierte Aktivitätsdiagramm aus.

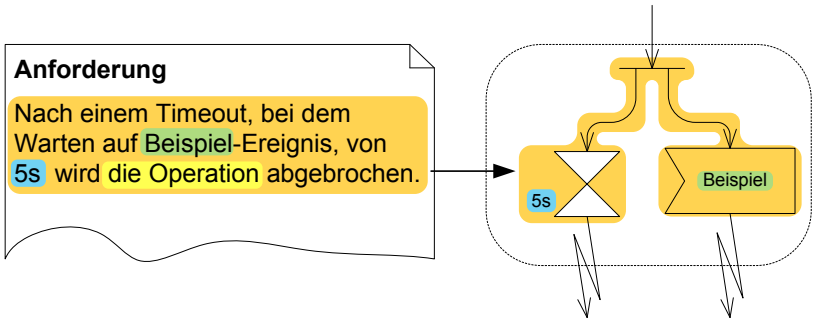



Abbildung 3.15: Beispiel eines Timeout-Templates

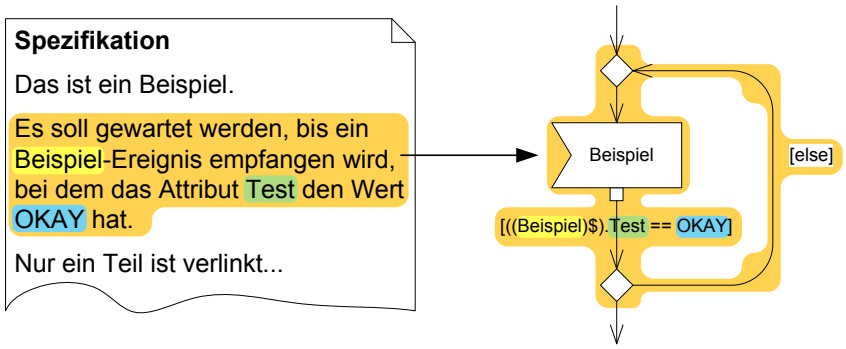



Abbildung 3.16: Beispiel eines „Warten auf Attributwert“-Templates

### 3.3.4 Konsistenzprüfungen

Mit Hilfe der Links können einige Konsistenzprüfungen durchgeführt werden. Die wichtigste Prüfung ist dabei die Impact-Analyse (vergleiche [3]). Wenn ein Artefakt geändert wird, müssen potentiell alle verlinkten Artefakte verändert werden. Dies wird in Abbildung 3.17 im oberen Teil anhand eines geänderten Use Cases und im unteren Teil anhand eines geänderten Modells gezeigt.

Durch Prüfung aller betroffenen Elemente, die durch die Impact-Analyse identifiziert wurden, können Inkonsistenzen bei Änderungen verhindert werden. Weitere Konsistenzprüfungen sind möglich. Einige Beispiele werden im Folgenden vorgestellt:

- In den Aktivitätsdiagrammen kann geprüft werden, ob überhaupt ein Signal mit dem Namen existiert, der in einem Sender oder Empfänger verwendet wurde.
- In einem Use Case in der Anforderungs-Struktur können die durch den Use Case abgedeckten Anforderungen angegeben werden. In den Aktivitätsdiagrammen müssen die Anforderungen ebenfalls in dem Bereich liegen, der den Use Case abdeckt. Dies kann getestet werden.
- Es kann geprüft werden, ob sich die Realisierung zweier Anforderungen in den Aktivitätsdiagrammen zu einem bestimmten Prozentsatz überlagern. Bei der Überschreitung

eines Schwellwertes kann eine Zusammenlegung der beiden Anforderungen mit der überlappenden Realisierung empfohlen werden.

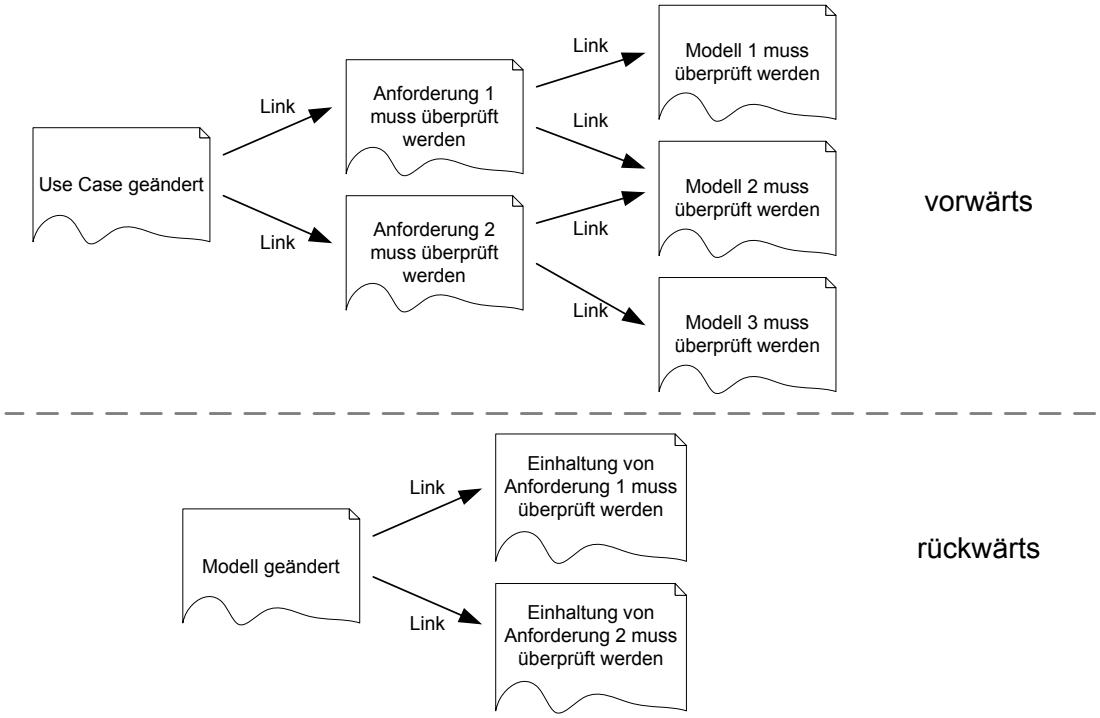



Abbildung 3.17: Impact-Analyse

### 3.3.5 Fazit

Die hier vorgestellte Architektur, die automatische Verlinkung und die Konsistenzprüfungen sind keineswegs neu. Aber sie sind höchst spezifisch und komplett an die ActiveCharts IDE angepasst. Dadurch kann Klarheit im Entwicklungsprozess mit der ActiveCharts IDE geschaffen werden. Ein Nachteil ist, dass eine explizite Unterstützung der einzelnen Editoren für die Verlinkung nötig ist. Aber dies lässt sich nicht umgehen, da es immer eine Möglichkeit geben muss, Links zu erstellen und den Links zu folgen. Das ist ohne eine GUI-Unterstützung nicht möglich.

### 3.3.6 Ausblick

Um für eine Mehrbenutzerumgebung tauglich zu sein, fehlen in der Architektur noch eine Benutzerverwaltung und Benutzerrechte. Allerdings müssen dann auch die Editoren die gleichzeitige Bearbeitung von mehreren Benutzern unterstützen. Das zentrale Änderungs-Log erfüllt aber weiterhin seinen Zweck und informiert alle Benutzer über gemachte und anstehende Änderungen.

Eine weitere Möglichkeit zum Erzeugen automatischer Links wird in [1, Seite 519] vorgestellt. Es wird durch eine Untersuchung der Änderungs-Historie festgestellt, welche anderen Artefakte bei einer Änderung an einem Artefakt auch betroffen sind. Daraus wird geschlossen, dass die Artefakte abhängig voneinander sind und die Erzeugung eines Links wird angeboten. Die vorgestellte Architektur hat durch das zentrale Änderungs-Log alle benötigten Informationen, die für eine solche Analyse nötig sind.

Für die automatische Verlinkung in den Anforderungsdokumenten können statt der einfachen Volltextsuche auch Textmining-Ansätze verwendet werden [1, Seite 518]. Dies ermöglicht nicht nur ein einfaches Pattern-Matching (wie in Abschnitt 3.3.3.2 vorgeschlagen), sondern versucht den internen Zusammenhang und bisher unbekannte Informationen über den Text zu finden. Diese Informationen könnten für das Setzen von automatischen Links verwendet werden.

## 3.4 Toolunterstützung

Dieser Abschnitt gibt einen kurzen Überblick über die Funktionalitäten des führenden Requirements Engineering-Tools am Markt und eines Integrationstools für verschiedene Requirements-Quellen.

### 3.4.1 Requirements-Tool

Abbildung 3.18 zeigt, dass im Jahr 2002 Telelogic DOORS und IBM Rational RequisitePro die Marktführerschaft im Bereich der Requirements-Tools inne hatten. Daran hat sich laut der IX-Studie Anforderungsmanagement [15] auch im Jahre 2005 nichts geändert. Im Folgenden wird exemplarisch ein kurzer Überblick über die Funktionalitäten des Marktführers gegeben, die auch für den Entwicklungsprozess mit der ActiveCharts IDE wünschenswert sind.

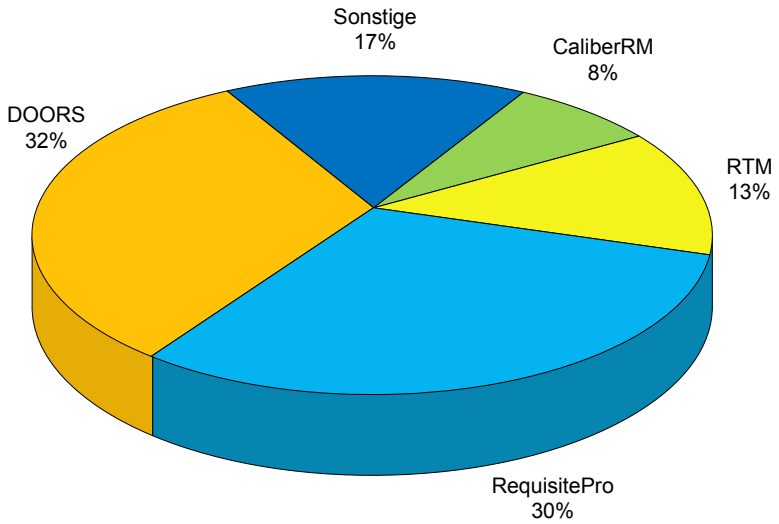



Abbildung 3.18: Marktanteil Requirements-Tools (Quelle: Standish Group (2002))

Telelogic Doors gehört zu den etablierten Produkten auf dem Markt. Es stammt aus dem Bereich der Luft- und Raumfahrt und hat sich zum Marktführer entwickelt. Doors verwaltet Anforderungen als Datensätze mit Titel, Beschreibung und beliebigen frei definierbaren Attributen. Dargestellt werden die Anforderungen in einer tabellarischen Ansicht. Die Anforderungen sind hierarchisch und werden mit natürlichsprachlichem Text beschrieben. Doors unterstützt die Erstellung eines Glossars. Die Oberfläche wirkt leicht angestaubt (siehe Abbildung 3.19) und entspricht nicht immer dem aktuellen Look & Feel. Doors verfügt über eine Versions- und Benutzerverwaltung. Außerdem werden sowohl interne Links als auch Links mit externen Artefakten unterstützt. Zur Erweiterung verfügt es über eine Programmiersprache namens DXL.

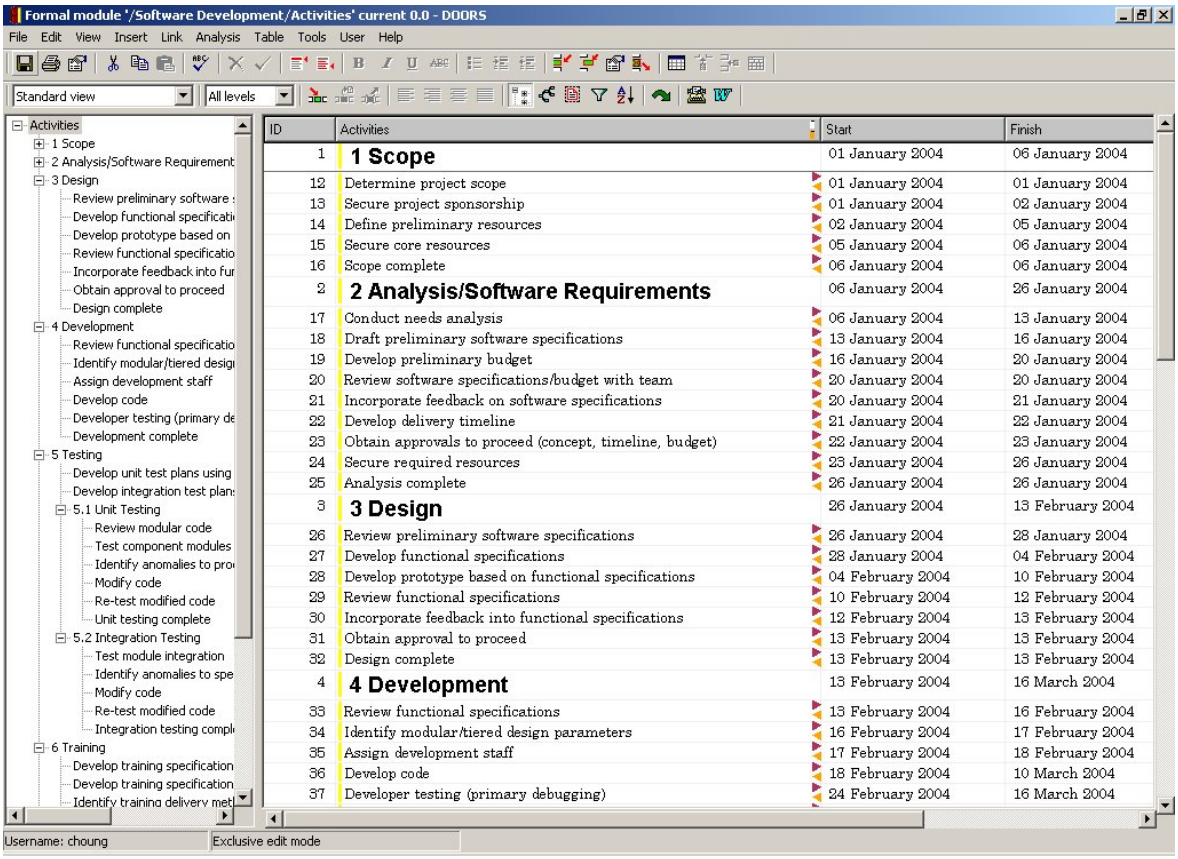



Abbildung 3.19: Doors-Screenshot (Quelle [35])

### 3.4.2 Traceability-Tools

Eine andere Tool-Kategorie stellt Reqtify von TNI-Software dar. Es dient nicht zum Schreiben, sondern zum Verwalten von Artefakten aus verschieden Quellen.

Um das zu erreichen, analysiert Reqtify die Artefakte in den externen Quellen. Reqtify kann jede externe Quelle analysieren, für die es einen Import-Filter besitzt. Nach der Analyse kann Reqtify Verknüpfungen zwischen den Artefakten der verschiedenen Quellen herstellen. Es ist

also ein Meta-Tool, das über den bestehenden Tools der einzelnen Phasen anzusiedeln ist. So werden in Abbildung 3.20 Tools aus jeder Phase der Softwareentwicklung gezeigt. Vom Requirements Engineering über die Modellierung und Implementierung bis zu den Tests und Bug-Tracking-Tools. Der Vorteil dieses Ansatzes ist, dass für jeden Bereich das gewohnte Spezialwerkzeug verwendet werden kann und (u. U. erst nachträglich) Reqtify zur Verwirklichung einer Tool-übergreifenden Traceability verwendet werden kann.

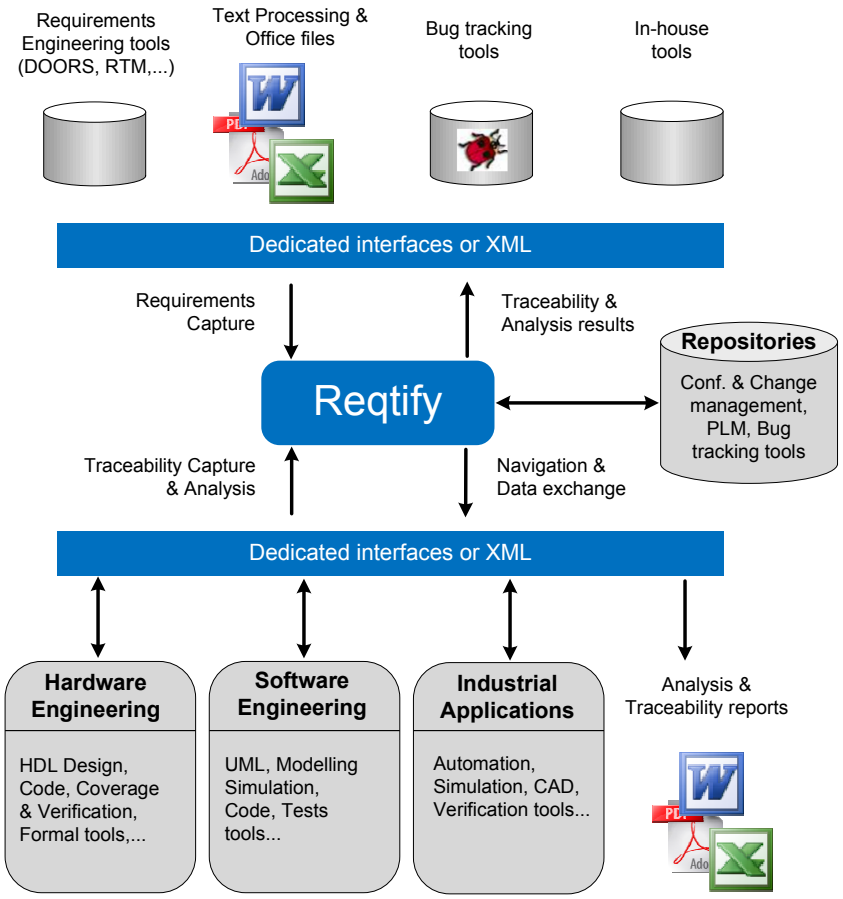



Abbildung 3.20: Übersicht über Reqtify (Quelle: [37])

### 3.4.3 Fazit

Für die Unterstützung des Entwicklungsprozesses mit der ActiveCharts IDE ist eine Kombination der beiden Tool-Arten nötig, da es sowohl möglich sein muss, Anforderungen zu verwalten als auch Elemente zu verlinken. Das Verwalten von Anforderungen umfasst unter anderem das Verfassen, Organisieren und Exportieren von Anforderungen. Die Verlinkung der Elemente erlaubt eine Traceability zwischen beliebigen Artefakten und Modellen.

Eine Kombination aus z. B. Doors und Reqtify kommt aber aus mehren Gründen nicht in Frage:

1. Für die Lizenzen der Tools entstehen sehr hohe Kosten. Eine Telelogic Doors-Lizenz kostet mit eingetragenem Benutzernamen 2.600€ und als Floating-Lizenz 6.400€ (Quelle: [15]). Die Kosten für eine Reqtify-Lizenz sind vom gewünschten Funktionsumfang abhängig und werden daher individuell bestimmt.
2. Es müssten wieder viele verschiedene Tools integriert werden. Die ActiveCharts IDE wurde aber gerade erst nach Java portiert, um ein einzelnes Programm zu erlauben (vor der Portierung: teils problematische Zusammenarbeit der ActiveCharts IDE und Visio).
3. Jeder ActiveCharts IDE-Editor würde einen Reqtify Import-Filter benötigen, das würde wieder eine Abhängigkeit von den Schnittstellen des Herstellers bedeuten.
4. Eine hohe Spezialisierung ist erwünscht, das Tool soll exakt für die ActiveCharts IDE passen.

Aus diesen Gründen wurde für diese Diplomarbeit ein Eclipse-Prototyp implementiert. Er dient als Machbarkeitstudie für ein Tool, das sowohl das Schreiben von Anforderungen als auch das Verlinken von Elementen erlaubt. Um trotzdem den Datenaustausch mit etablierten Tools zu ermöglichen, wurde ein Export in das RIF-Format für den Editor entwickelt.

# 4 Praktischer Teil

In diesem Kapitel soll der als Proof-Of-Concept entwickelte Prototyp vorgestellt werden. Dazu werden in 4.1 zuerst die an das Tool gestellten Anforderungen vorgestellt.

Das Tool wurde in Form von zwei Plug-ins realisiert. In Abschnitt 4.2 wird das erste Plug-in vorgestellt, der Requirements Editor. Dies umfasst die Architektur des Plug-ins, die Speicherung der Anforderungen, die Darstellung der Elemente und den Import und Export der Anforderungen in das RIF-Format.

Das zweite Plug-in, das Change Log, wird in Abschnitt 4.3 vorgestellt. Auch hier wird zuerst die Architektur vorgestellt und dann auf die Funktionalität eingegangen.

In Abschnitt 4.4 werden die Erweiterungsmöglichkeiten der beiden Plug-ins vorgestellt. Bei dem Requirements Editor umfasst das unter anderem das Hinzufügen neuer Elemente und das Ändern der Darstellung. Bei dem Change Log-Plug-in wird vorgestellt, wie andere Editoren an der Traceability-Architektur teilnehmen können.

Da es sich um einen Prototypen handelt, wird in Abschnitt 4.5 ein Überblick über den Stand der Implementierung gegeben.

Schließlich wird in 4.6.1 die Einhaltung der Anforderungen aus Abschnitt 4.1 diskutiert und eine kurze Bewertung des Tools gegeben.

#### Konventionen

Da im Quellcode englische Klassenbezeichner verwendet wurden, werden im folgenden Text die englischen Namen verwendet. Allerdings sind die Klassennamen, die den Namenskonventionen von Java folgen, nicht sehr angenehm zu lesen. Daher wird an manchen Stellen bei der ersten Verwendung des englischen Klassennamens eine deutsche oder sprechendere Variante des Klassennamens eingeführt, welche im Folgenden anstatt des Klassennamens verwendet wird. Des Weiteren werden Klassenbezeichner und Methoden zur besseren Lesbarkeit im Text kursiv geschrieben. Methoden werden zur Unterscheidung von Klassenbezeichnern durch „()“ nach dem Methodennamen gekennzeichnet.

Es werden nachfolgend vereinfachte, nicht vollständige, Klassendiagramme gezeigt, da diese oft sehr groß und unübersichtlich sind. Für eine Beschreibung aller Methoden wird auf die API Dokumentation verwiesen.

## 4.1 Anforderungen an das Tool

Um einen möglichst großen Nutzen für die Entwicklung mit der ActiveCharts IDE zu erreichen, werden folgende Anforderungen an das zu entwickelnde Tool gestellt:

- **Enge Integration mit der ActiveCharts IDE**
  Das Tool soll ein Teil der ActiveCharts IDE werden.
- **Anforderungs-Bearbeitung**
  Das Tool soll auf eine möglichst komfortable Art und Weise das Editieren und Verwalten von Anforderungen erlauben.
- **Anforderungs-Import und -Export**
  Das Tool soll Anforderungen mit externen Tools austauschen können.
- **Unterstützung des Vorgehens aus dem Praktikum**
  Das Tool soll das Vorgehen aus dem Praktikum unterstützen. Das beinhaltet insbesondere die Unterstützung der in Abschnitt 3.1 „Vorgehensmodell zur Entwicklung mit der ActiveCharts IDE“ identifizierten Artefakte.
- **Unterstützung der Traceability-Architektur**
  Das Tool soll die Traceability-Architektur aus Abschnitt 3.3 unterstützen.
- **Einfache Erweiterbarkeit**
  Das Tool soll einfach um neue Artefakt-Typen und Darstellungen erweitert werden können.

Inwiefern diese Anforderungen erfüllt wurden, wird am Ende dieses Kapitels in Abschnitt 4.6.1 diskutiert.

## 4.2 Requirements Editor-Plug-in

Das Requirements Editor-Plug-in stellt einen Eclipse-Editor zur Bearbeitung der textuellen Artefakte zur Verfügung. Außerdem wurde ein Eclipse-View, der einen hierarchischen Überblick über die Struktur des Dokuments gibt, implementiert.

Abbildung 4.1 zeigt die Architektur und die wichtigsten Komponenten des Plug-ins.

Die Klassen *RequirementsEditor* (Requirements Editor), *SectionRegistry* (Section Registry) und *StructureView* (Structure View) stellen die Hauptkomponenten des Plug-ins dar. Sie werden im Folgenden beschrieben:

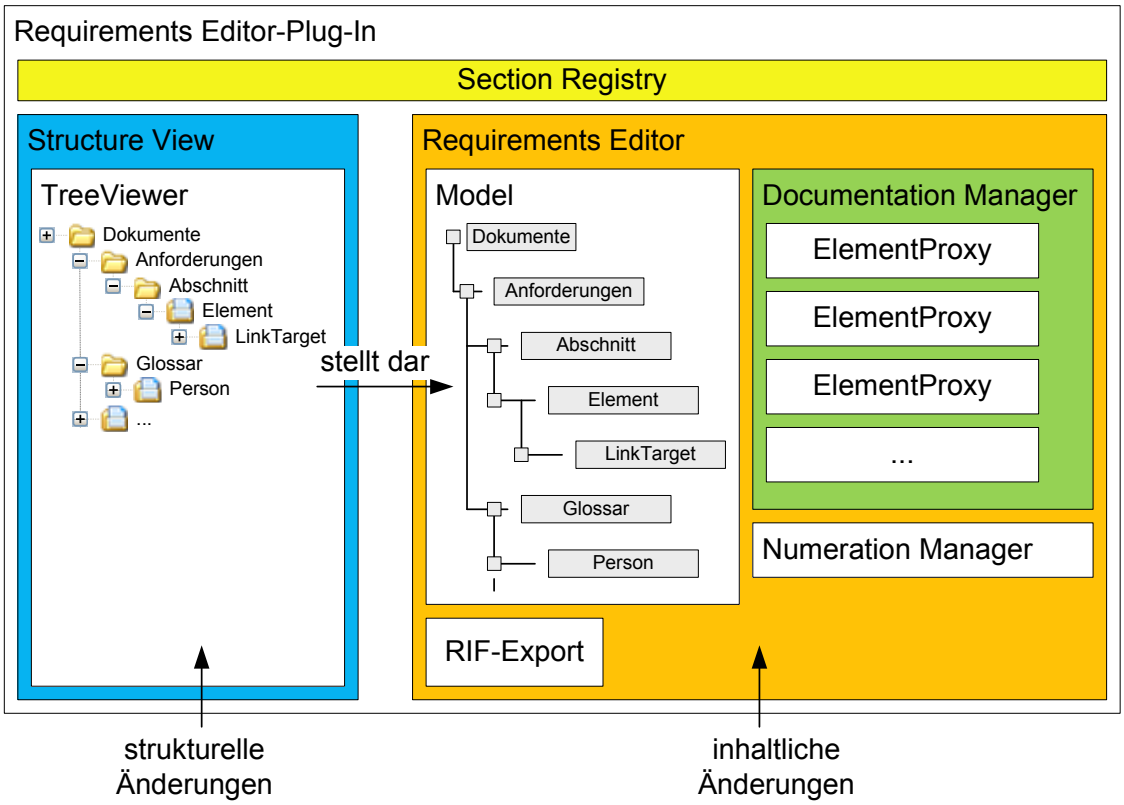



Abbildung 4.1: Schematischer Aufbau und Komponenten des Requirements Editor-Plug-ins

### Requirements Editor

Der Requirements Editor dient zur Bearbeitung des Anforderungs-Modells (siehe nächsten Abschnitt). Mit ihm können inhaltliche Änderungen an dem Anforderungs-Modell vorgenommen werden. Er enthält die Elemente des Anforderungs-Modells, die aus der zu dem Editor gehörenden Datei beim Öffnen geladen wurden. Der Editor verwendet zum Laden und Speichern die Serialisierung von Java. Wie in Abbildung 4.2 gezeigt, stellt der Requirements Editor die Elemente in einer Tabellenform dar. Die Reihenfolge der Elemente entspricht einem Preorder-Durchlauf der Baumstruktur des Anforderungs-Modells. Die einzelnen Elemente werden mit Hilfe von Klassen dargestellt, welche die Schnittstelle *ElementProxy* (siehe Abschnitt 4.4 „Erweiterungsmöglichkeiten“) implementieren. Die Zuordnung, welches Element mit welcher Klasse angezeigt wird, wird der Section Registry entnommen.

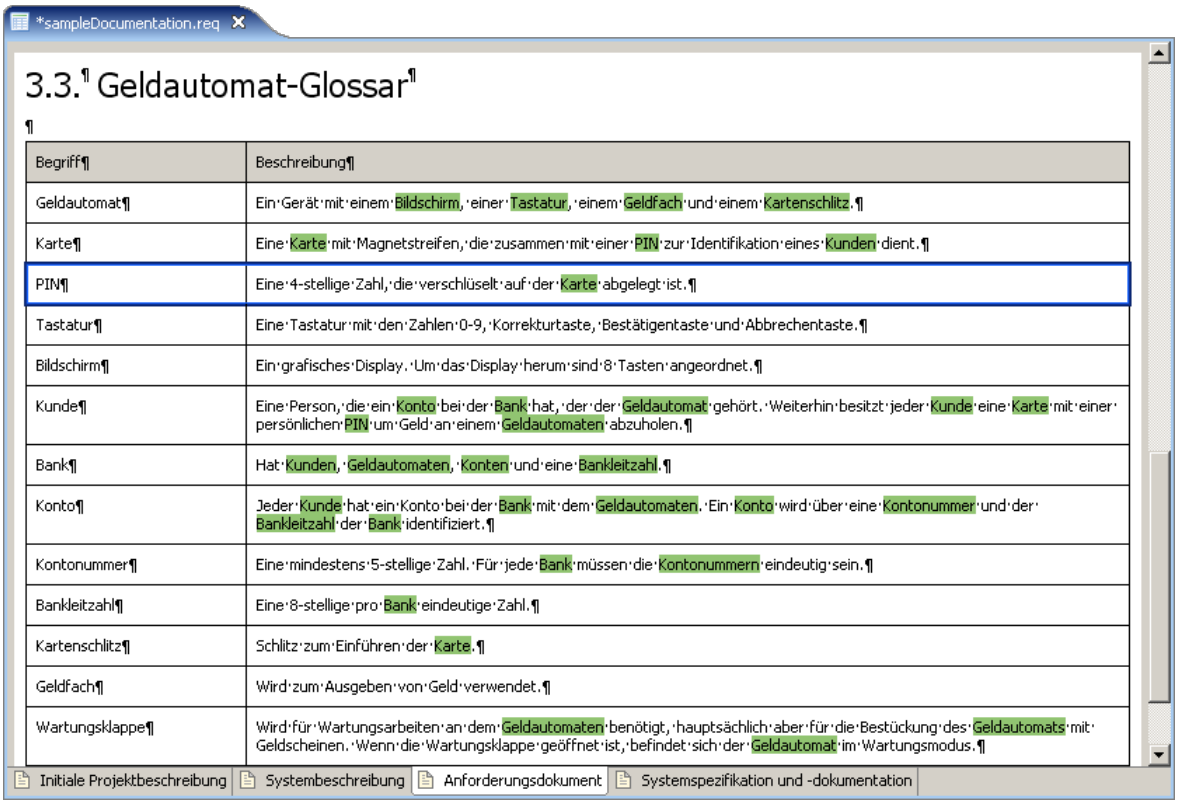



Abbildung 4.2: Screenshot des Requirements Editors

Die Erzeugung und Positionierung der *ElementProxy*-Instanzen wird von einer Instanz der *DocumentationManager*-Klasse (Documentation Manager) verwaltet (siehe Abbildung 4.3).

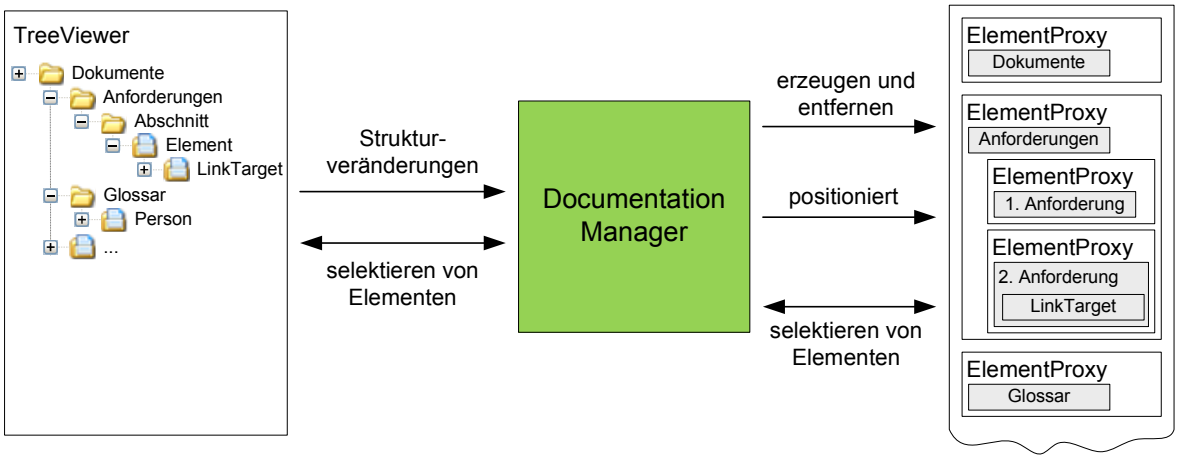



Abbildung 4.3: Schematischer Überblick über den Documentation Manager

Außerdem sorgt der Documentation Manager für die Verwaltung des aktuell markierten Elements und dem Löschen von Elementen und den dazugehörigen *ElementProxy*-Instanzen. Der Documentation Manager stellt das Bindeglied zwischen den Änderungen im Section View und

der Aktualisierung der Darstellung im Requirements Editor dar.

Der Numeration Manager sorgt für eine Nummerierung der Elemente. Er teilt jedem *ElementProxy* bei einer Änderung der Struktur des Anforderungs-Modells seine aktuelle Nummerierung mit. Das bedeutet, jede *ElementProxy*-Instanz kennt seine Nummerierung und kann sie anzeigen.

Der RIF-Export des Anforderungs-Modells wird in Abschnitt 4.2.3 beschrieben.

### Section Registry

Die Section Registry hat zwei Aufgaben. Zum ersten sorgt sie dafür, dass nicht beliebige Elemente in dem Anforderungs-Modell verschachtelt werden können. Dies wird durch eine Einschränkung der erlaubten Kinder erreicht. So können z. B. für ein Glossar nur Terme als Kinder erlaubt werden. Zum zweiten wird bei der Section Registry registriert, welche Elemente mit Instanzen welcher Klasse angezeigt werden sollen. Die Section Registry ist durch eine XML-Datei konfigurierbar und somit leicht zu ändern und zu erweitern (siehe Abschnitt 4.4 „Erweiterungsmöglichkeiten“).

### Structure View

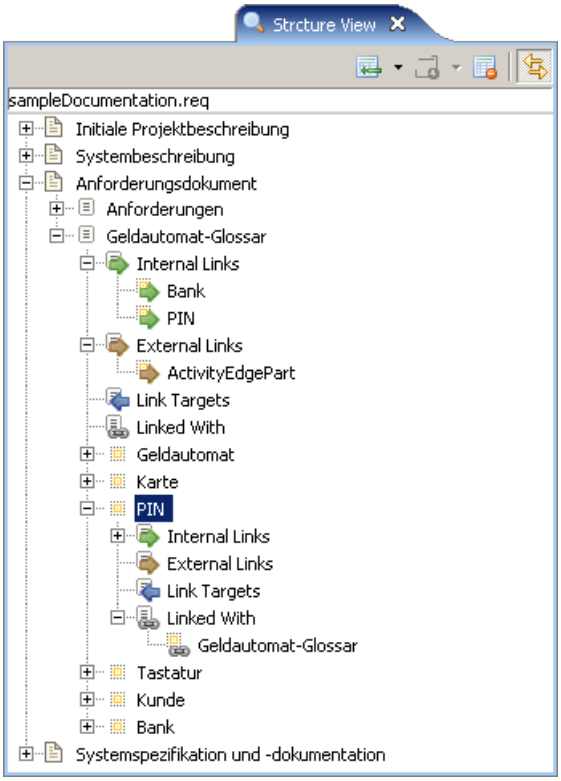



Abbildung 4.4: Screenshot des Structure View

Der Structure View stellt das Anforderungs-Modell in einer Baumstruktur dar und erlaubt strukturelle Änderungen, wie das Löschen oder Einfügen von neuen Elementen. Außerdem

können Elemente mit Hilfe von Drag & Drop verschoben werden, auch hier wird die Gültigkeit der Drop-Position mit Hilfe der Section Registry eingeschränkt. Die Darstellung der Baumstruktur wurde mittels der JFace-Komponente TreeViewer realisiert. Der Structure View selbst beinhaltet kein Anforderungs-Modell, sondern zeigt das Modell des momentan oder als letztes aktiven Requirement Editors an. Abbildung 4.4 zeigt den Structure View bei der Anzeige des Anforderungs-Modells eines Geldautomaten.

Für eine detaillierte Beschreibung der GUI siehe Anhang A.

### 4.2.1 Anforderungs-Modell

Das interne Modell zur Speicherung der Anforderungen bzw. aller textueller Artefakte wurde aus der Anforderungs-Struktur aus Abschnitt 3.2.2 „Struktur für ein Anforderungsdokument“ übernommen und um Java-spezifische Details erweitert und angepasst (siehe Abbildung 4.5). So wurden die Datentypen festgelegt und die neuen Klassen *Section* (dt. Abschnitt) und *Documentation*, die zur Verwaltung dienen, hinzugefügt. Außerdem wurden einige Elemente entfernt und nur die in Abbildung 4.5 gezeigten Elemente im Prototypen realisiert.

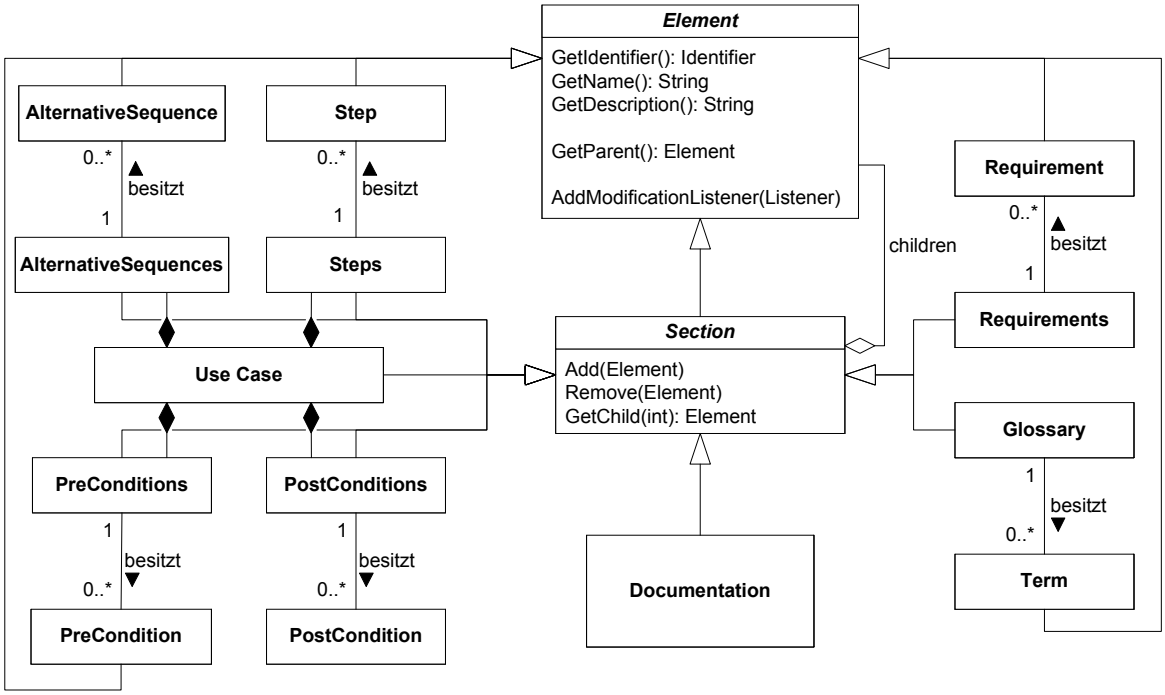



Abbildung 4.5: In Java realisierte Variante der Anforderungs-Struktur

Der Wurzelknoten eines Anforderungs-Modells ist immer eine Instanz der *Documentation*-Klasse. An jedem Element kann ein Änderungs-Listener registriert werden, mit dessen Hilfe auf das Einfügen, das Löschen und das Ändern eines Elements reagiert werden kann. Die Änderungs-Listener werden zentral in der *Documentation*-Instanz, dem Wurzelknoten, ver-

waltet. Wird ein Listener bei einem Element registriert, wird diese Registration in der Baumstruktur nach oben an die *Documentation*-Instanz weitergereicht.

Die *Section*-Klasse ist ein Teil eines Composite Pattern (siehe Gamma et al. [13, Seite 163]), welches für das Anforderungs-Modell verwendet wird. Das Composite Pattern ist ein Strukturmuster und erlaubt, sowohl die Blätter als auch die inneren Knoten in einer Baumstruktur mit der gleichen Schnittstelle anzusprechen.

Abbildung 4.6 zeigt ein Objektdiagramm eines Anforderungs-Modells. Der Wurzelknoten ist, wie bei jeder Instanz des Anforderungs-Modells, eine Instanz der *Documentation*-Klasse. Diese ist ein Container für beliebig viele *TopLevel*-Instanzen. Die Klasse *TopLevel* ist eine Subklasse der *Section*-Klasse und stellt die einzelnen Dokumente in dem Modell dar. Die einzelnen Dokumente bzw. *TopLevel*-Instanzen dienen nur zur logischen Gliederung. So wird z. B. in Abbildung 4.6 der Inhalt des Anforderungs-Modells in ein Anforderungsdokument und eine Spezifikation aufgeteilt. Die Anzahl und Benennung der *TopLevel*-Instanzen ist nicht fest, sie kann, wie in Abschnitt 4.4 erläutert, angepasst werden. Außerdem zeigt Abbildung 4.6 noch drei Anforderungen und ein Glossar.

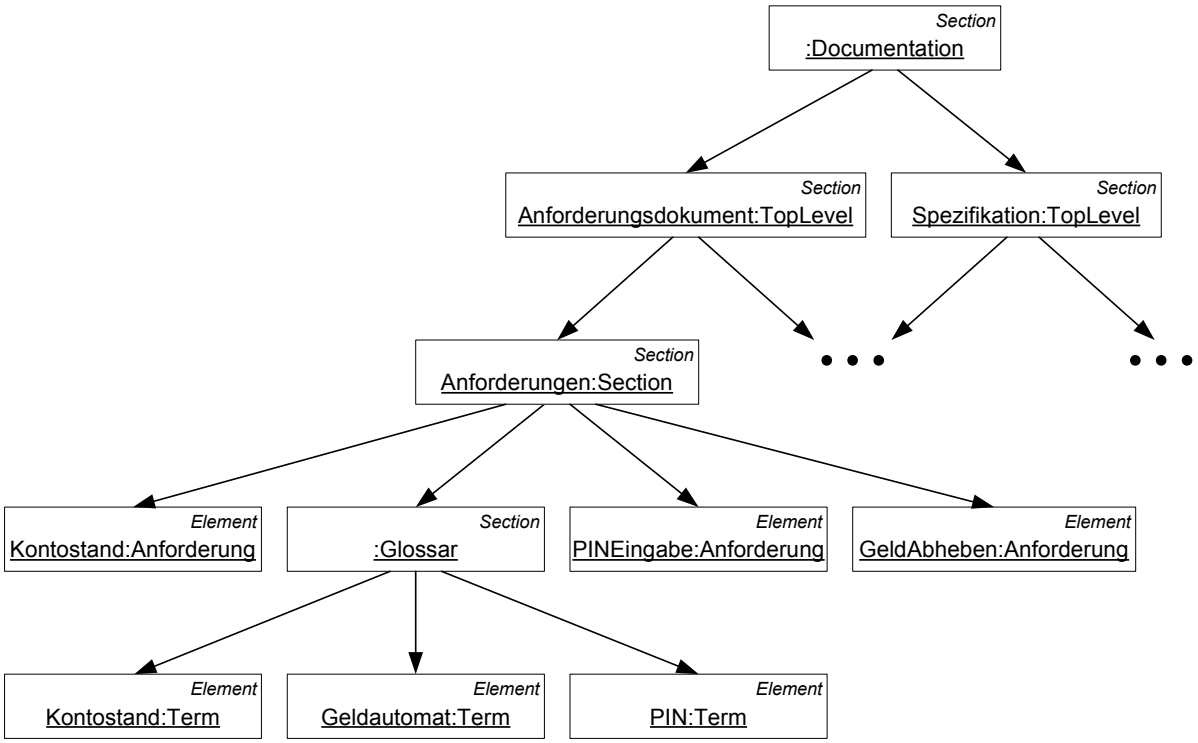



Abbildung 4.6: Baumstruktur des Anforderungs-Modells

Allerdings wurde das Composite Pattern im Vergleich zu der von Gamma et al. vorgestellten Version etwas variiert, indem keine gemeinsame Basisklasse oder Schnittstelle verwendet wird, sondern die abstrakte Klasse für innere Knoten (*Section*) erbt direkt von der abstrakten Blätter-Klasse (*Element*). Das führt zu dem in Abbildung 4.7 gezeigten Aufbau.

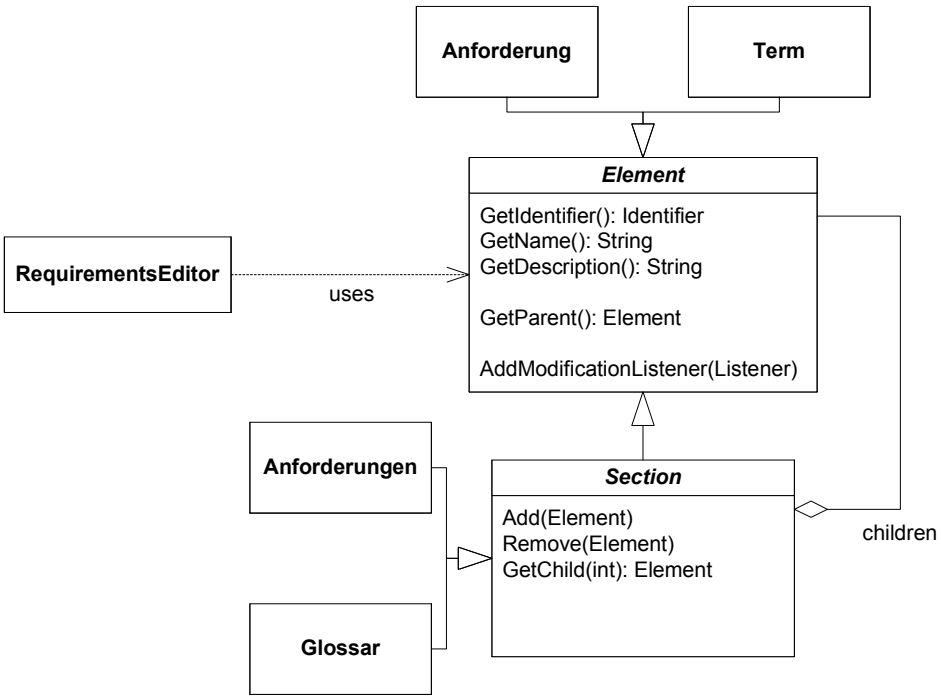



Abbildung 4.7: Abgewandeltes Composite Pattern mit vier Beispielklassen

Jedes Element hat einen eindeutigen Identifier, einen Namen und eine Beschreibung. Außerdem kann bei jedem Element (wie schon zuvor beschrieben) ein Änderungs-Listener registriert werden. Die *Section*-Klasse verfügt über Methoden zur Verwaltung der Kinder. Zusätzlich zu den schon beschriebenen Klassen sind in Abbildung 4.7 beispielhaft vier Klassen aus dem Anforderungs-Modell aus Abbildung 4.5 dargestellt.

### 4.2.2 Darstellung der Elemente

Die Darstellung der Elemente im Requirements Editor ist nicht fest vorgegeben. Jede SWT Komponente, welche die Schnittstelle *ElementProxy* (für eine detaillierte Beschreibung siehe Abschnitt 4.4 „Erweiterungsmöglichkeiten“) implementiert, kann ein Element darstellen. Da aber eine möglichst intuitive Bearbeitung gewünscht ist, muss die zur Darstellung verwendete Komponente einige Eigenschaften erfüllen. Im Folgenden werden diese Eigenschaften erläutert:

- **Anpassbarkeit**
  Unter Anpassbarkeit wird verstanden, ob die Komponente definierte Erweiterungspunkte zur Veränderung ihres Verhaltens bereitstellt.
- **Tabellen-Unterstützung**
  Die Tabellen-Unterstützung gibt an, ob die Komponente im Text eingebettete Tabellen unterstützt. Außerdem muss ein Zugriff aus dem Code heraus auf die einzelnen Zeilen

und Zellen der Tabelle möglich sein.

- **Handhabung**
  Unter Handhabung wird verstanden, ob sich der Inhalt in der Komponente wie in einer Textverarbeitung editieren lässt.
- **Einbettung beliebiger Objekte**
  Gibt an, ob beliebige benutzerdefinierte Objekte in den Textfluss eingefügt und editiert werden können. Ein Beispiel dafür stellen die LinkTargets dar (siehe Abschnitt 3.3.2.2, Abbildung 3.10)
- **native SWT-Komponente**
  Ob die Komponente eine native SWT-Komponente ist.

Um eine Komponente zu finden, welche diese Eigenschaften erfüllt, wurden die momentan für SWT verfügbaren Komponenten untersucht, welche komfortables Editieren von Texten ermöglichen. Dabei wurden nur frei verfügbare Komponenten berücksichtigt. Es wurden drei Komponenten gefunden.

Die erste ist die StyledText-Komponente [11] von SWT. Zwar ist sie die Standard-Komponente zur Anzeige von formatiertem Text in SWT, aber sie hat z. B. keinerlei Tabellen-Unterstützung.

Die zweite ist die Swing-Komponente JTextPane [34], welche über eine AWT-SWT-Bridge verwendet werden muss. In einer JTextPane-Komponente sind zwar Tabellen möglich, jedoch ist die Unterstützung für eingebettete Objekte nicht komplett. So kann z. B. ein eingefügtes Bild nicht markiert werden. Außerdem handelt es sich um eine Swing Komponente, die über einen Wrapper eingebunden werden muss.

Die letzte gefundene Komponente ist die richhtml4eclipse-Komponente [32]. Den Kern bildet der JavaScript-WYSIWYG-Editor tinyMCE. Dieser Editor wird in einer Instanz eines Browser-Fensters angezeigt. Zur Kommunikation zwischen Java-Code und dem JavaScript-Code werden Event-Listener verwendet. Der Nachteil der Komponente ist, dass nur HTML editiert werden kann und dass durch die Integration der verschiedenen Techniken ein großer Overhead entsteht.

Da keine der Komponenten die gewünschten Eigenschaften besitzt, wurde die AdvancedText-Komponente entwickelt. Eine solche Komponente hat großes Potenzial, da die Nachfrage der SWT-Entwickler in Foren und Newsgroups nach einer Komponente mit den oben geforderten Eigenschaften relativ groß ist. Eine Eigenentwicklung hat auf der einen Seite den Vorteil, dass alle gewünschten Eigenschaften erfüllt werden, auf der anderen Seite besteht aber der Nachteil eines großen Entwicklungsaufwands und ein mangelnder Reifegrad der Implementierung. Die AdvancedText-Komponente verfügt bis jetzt über rudimentäre Eingabemöglichkeiten und hat noch einige bekannte Probleme (siehe Anhang B).

In Abbildung 4.8 wird ein Vergleich der gefundenen Komponenten und der AdvancedText-Komponente gezeigt.

| | Anpassbarkeit | Tabellen-Unterstützung | Handhabung | Einbettung beliebiger Objekte | native SWT-Komponente | Stabilität |
|---|---|---|---|---|---|---|
| **SWT StyledText** | o | - | o | o | + | + |
| **Swing JTextPane** | + | o | o | o | - | + |
| **richhtml4eclipse** | - | + | + | o | - | + |
| **AdvancedText** | + | + | + | + | + | o |

Legende: + gut o zufriedenstellend - schlecht

Abbildung 4.8: Übersicht über Texteditor-Komponenten

Im Folgenden wird ein Überblick über die Architektur der AdvancedText-Komponente gegeben und Abbildung 4.9 zeigt den schematischen Aufbau.

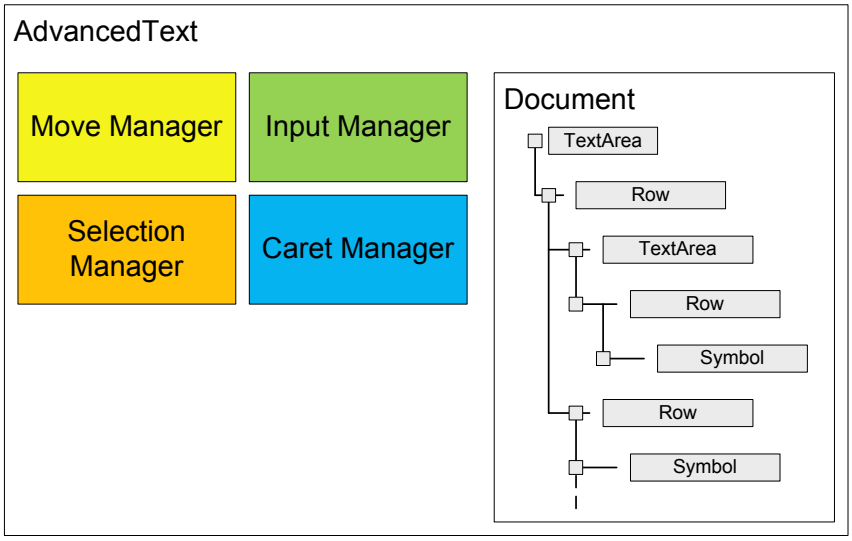



Abbildung 4.9: Architektur der AdvancedText-Komponente

Die AdvancedText-Komponente enthält eine *Document* Instanz, die in einer Baumstruktur das zu bearbeitende Dokument speichert. Für eine genaue Beschreibung siehe nächsten Abschnitt. Eine *Document* Instanz ist das Dokument und die AdvancedText-Komponente ist die grafische Oberfläche, welche eine Bearbeitung ermöglicht.

Außerdem enthält die AdvancedText-Komponente noch verschiedene Manager. Jeder dieser Manager ist für einen Aspekt der Funktionalitäten der AdvancedText-Komponente verantwortlich. Die Manager bilden die Schnittstelle zwischen der *Document*-Instanz und SWT bzw. der GUI im Allgemeinen. Im Folgenden werden die einzelnen Manager und ihre Aufgabe kurz beschrieben:

- **Input Manager**
  Wertet Tastendrücke aus und fügt neue Zeichen ein, bzw. löscht Zeichen aus der *Document*-Instanz.
- **Caret Manager**
  Ist für die Darstellung und richtige Positionierung des Textcursors verantwortlich.
- **Move Manager**
  Mit Hilfe des Move Managers können *TextFlow*-Instanzen (siehe nächster Abschnitt) im Text per Drag & Drop verschoben werden.
- **Selection Manager**
  Der Selection Manager verwaltet eine Markierung im Text und zeigt sie an.

Im nächsten Abschnitt wird die Baumstruktur des Dokumenten-Modells vorgestellt.

#### 4.2.2.1 Dokumenten-Modell

Der Inhalt eines Dokuments wird durch eine Baumstruktur aus *Glyph*-Instanzen repräsentiert. Der Aufbau der Baumstruktur wurde aus [13, Seite 35] übernommen. Die grundlegende Idee ist, dass sowohl Text als auch Bilder und sonstige Elemente in der Baumstruktur gleichberechtigt sind. Das erlaubt eine Verallgemeinerung der Schnittstelle aller Objekte im Dokument, alle sind Subklassen der Klasse *Glyph*. Des Weiteren können neue Elemente transparent hinzugefügt werden, da sie die gleiche schon bekannte Schnittstelle verwenden.

Abbildung 4.11 zeigt, wie die Baumstruktur aus Abbildung 4.10 zum Editieren dargestellt wird.

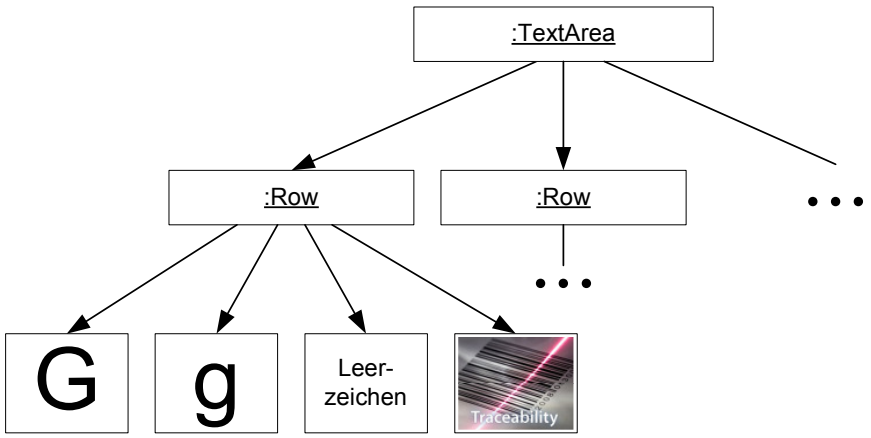



Abbildung 4.10: Ausschnitt aus einer *Glyph*-Struktur (Quelle: [13, Seite 37])

Für die Baumstruktur wurde das gleiche abgewandelte Composite Pattern wie in Abschnitt 4.2.1 für das Anforderungs-Modell verwendet, d. h. es wurde keine gemeinsame Basisklasse oder Schnittstelle verwendet, sondern alle Klassen erben direkt von der *Glyph*-Klasse.

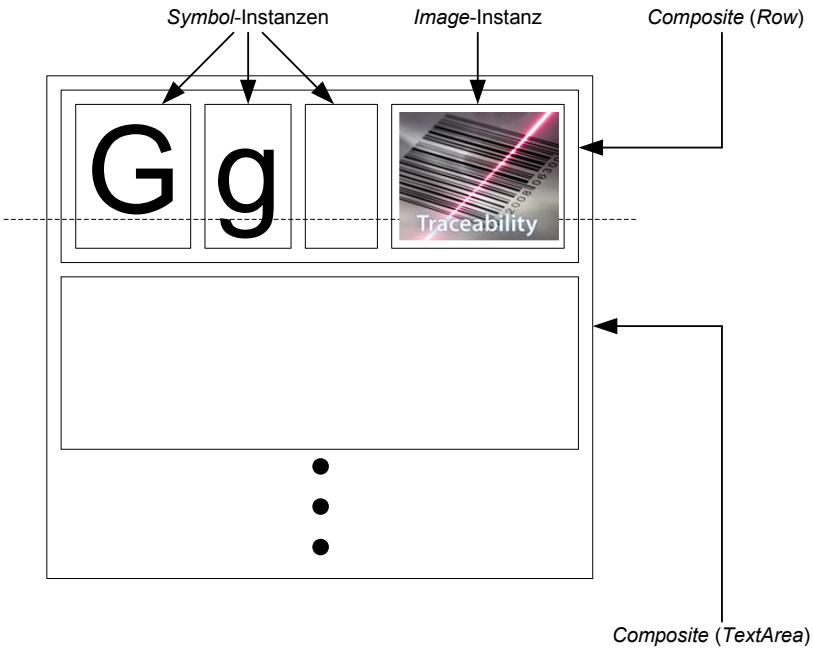



Abbildung 4.11: Darstellung der *Glyph*-Struktur (Quelle: [13, Seite 37])

Zusätzlich zu den Methoden eines Composite Patterns (siehe Gamma et al. [13, Seite 163]) muss jede Instanz einer Subklasse von *Glyph* folgende Anforderungen erfüllen:

- Sie muss sich selbst und ihre Kinder darstellen können.
- Sie muss eine Größe besitzen.
- Sie muss eine Liste von Kindern verwalten und ihren Elternknoten kennen.

In Tabelle 4.1 werden die atomaren *Glyph*s ohne Kinder und in Tabelle 4.2 die komplexen *Glyph*-Subklassen beschreiben.

| **Atomare Subklassen von *Glyph*** | |
|---|---|
| *Symbol* | Repräsentiert einen einzelnen Buchstaben in einer gewählten Schriftart und Größe. |
| *Image* | Repräsentiert ein Bild mit einer festen Größe. |

Tabelle 4.1: Beschreibung der wichtigsten atomaren *Glyph*-Subklassen

| **Komplexe Subklassen von *Glyph*** | |
|---|---|
| *TextArea* | Dient als Container für *Row*-Instanzen. Jede *Document*-Instanz besitzt eine *TextArea*-Instanz als Wurzelknoten der Glyph-Struktur. |
| *Table* | Stellt eine Tabelle mit absoluter oder relativer Breite dar. |
| *TableRow* | Stellt eine Zeile in einer Tabelle dar. |
| *TableCell* | Stellt eine Zelle in einer Tabellenzeile dar und ist eine Subklasse von *TextArea*. |

| | |
|---|---|
| *Row* | Repräsentiert eine Zeile. |
| *TextFlow* | Ein TextFlow ist ein zusammengehöriger Bereich im Text, der als Einheit behandelt wird, der aber trotzdem (siehe Beispiel weiter unten, der blau unterlegte Bereich ist ein TextFlow) umgebrochen werden kann. Die·Details·der·Eingabe· werden·in·Eingabe·der· PIN·beschrieben .¶ Aber obwohl der TextFlow umgebrochen werden kann, wird z. B. der ganze TextFlow gelöscht, wenn der letzte Teil gelöscht wird. Die TextFlows werden im Requirement Editor zur Darstellung von Links verwendet und stellen daher ein wichtiges Feature der AdvancedText-Komponente dar. Sie sind eine Subklasse von *TextArea*. |

Tabelle 4.2: Beschreibung der wichtigsten komplexen *Glyph*-Subklassen

Abbildung 4.12 zeigt einen Überblick über die Blätter-Klasse *Glyph*, die Klasse *Composite* für innere Knoten und die wichtigsten abgeleiteten Klassen.

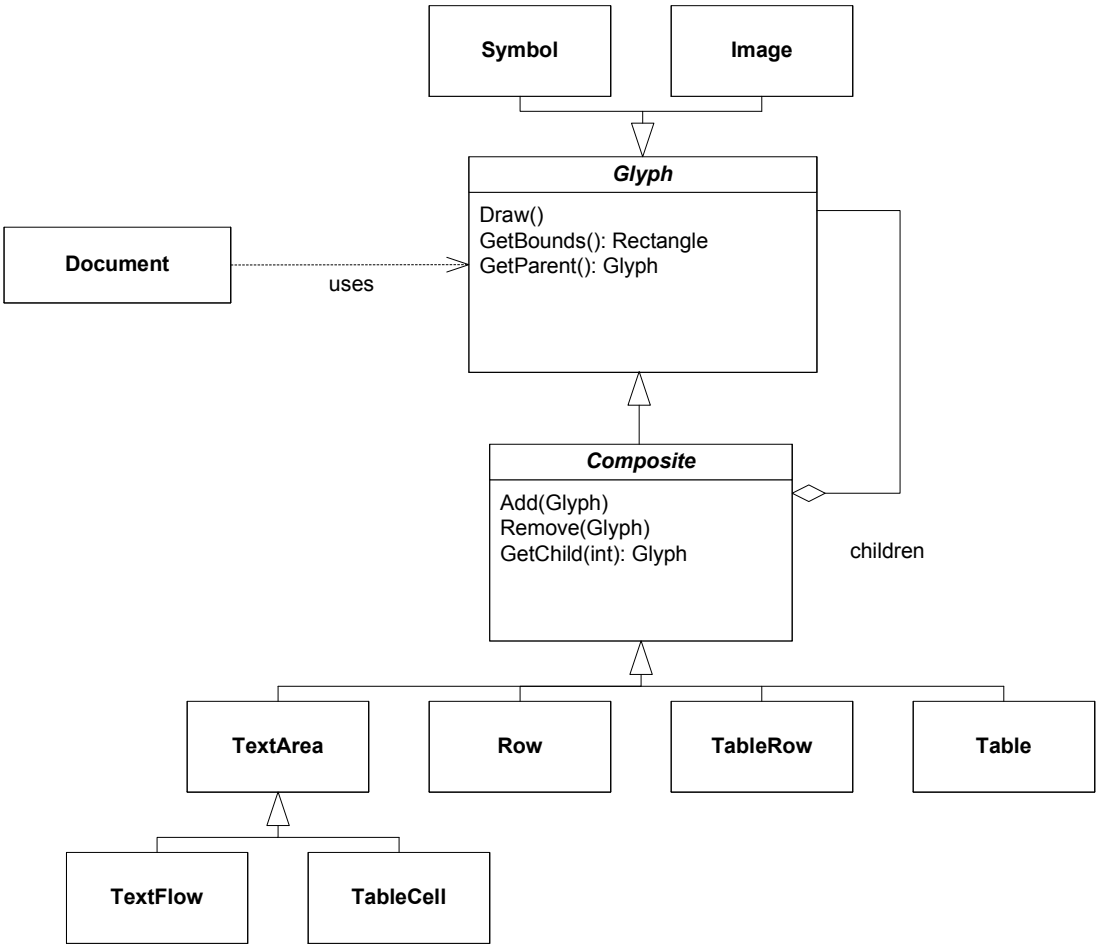



Abbildung 4.12: Abgewandeltes Composite Pattern der Glyph-Struktur

Damit bei der Speicherung einer *Document*-Instanz nicht der gesamte Glyph-Baum gespeichert werden muss, wurde eine Möglichkeit eingeführt, beliebige Glyphs als Text zu speichern. Dadurch wird es möglich, nur mit Texten umgehen zu müssen. Eine *Glyph*-Subklasse kann als Text serialisiert und wieder deserialisiert werden, wenn sie die Schnittstelle *TextSerializable* implementiert und einen Konstruktor zur Verfügung stellt, der den Inhalt des Glyphs aus dem übergebenen Text wiederherstellt.

Die Glyph-Struktur an sich ist unabhängig von dem SWT-Toolkit. Es werden entweder nur Klassen aus der Java-Laufzeitbibliothek oder implementierungsunabhängige Schnittstellen verwendet. Erst die AdvancedText-Komponente und ihre Manager sind SWT spezifisch. Sie implementieren die entsprechenden Schnittstellen und rufen SWT-Methoden auf.

Die AdvancedText-Komponente stellt zusätzlich die Schnittstelle *ActivatableGlyph* und *DropableGlyph* zur Verfügung. Sie können von Subklassen der *Glyph*-Klasse implementiert werden. Das Interface *ActivatableGlyph* ermöglicht es Glyphs, durch die GUI aktiviert zu werden. Wird ein *Glyph* aktiviert, so wird die Methode *activate()* aus dem *ActivatableGlyph*-Interface aufgerufen und das Glyph kann auf die Aktivierung reagieren. Durch das Implementieren der Schnittstelle *DropableGlyph* weisen *Glyph*-Subklassen sich als Dropziel für Element-Deskriptoren aus. Diese beiden Schnittstellen werden z. B. von den Links implementiert. Die Aktivierung wird zum Springen zu dem Linkziel und die Ausweisung als Dropquelle zum Verlinken verwendet.

### 4.2.3 RIF-Import und -Export des Modells

Das Ziel des RIF-Imports und -Exports ist es, das in Abschnitt 4.2.1 vorgestellte Anforderungs-Modell in das RIF-Format zu exportieren und wieder zu importieren. Dadurch wird ein Datenaustausch mit anderen Tools möglich.

Zur Realisierung eines RIF-Exports wurde das in Abschnitt 2.5 in Abbildung 2.11 gezeigte UML-Modell der HIS in Java-Klassen umgesetzt. Zwar verfügt Castor über eine Möglichkeit, aus einem XML-Schema automatisiert Java-Klassen zu generieren. Ein Test mit dem XML-Schema der HIS für das RIF-Format ergab aber, dass die generierten Klassen inkonsistent mit dem vorgegeben UML-Klassendiagramm sind. Daher wurden die Klassen des RIF-Modells von Hand erstellt. Die Klassen für die Beziehungen, Gruppen und Benutzerrechte wurden bisher noch nicht erstellt, da diese bisher nicht für den Export benötigt wurden.

Zusätzlich zu den erstellten Klassen wurde eine XML-Mapping-Datei zur Abbildung der Java-Klassen des RIF-Modells auf ein XML-Dokument erstellt. Wie der Ausschnitt aus der Mapping-Datei in Listing 4.1 zeigt, wird im Mapping auch die Umsetzung der Namenskonventionen vorgenommen, wie sie von der RIF-Spezifikation gefordert werden.

```
1  <class name="activecharts.re.rif.model.SpecType"
       extends="activecharts.re.rif.model.Identifiable">
2     <map-to xml="SPEC-TYPE" />
3     <field name="SpecAttributes"
4        type="activecharts.re.rif.model.AttributeDefinition"
5        collection="vector"
6        direct="false"
7        container="false">
8     <bind-xml name="SPEC-ATTRIBUTES" />
9     </field>
10 </class>
```

Listing 4.1: Ausschnitt aus der XML-Mapping-Datei für Castor

In Listing 4.1 wird die Java-Klasse *SpecType* auf einen Tag mit dem Namen „SPEC-TYPE“ abgebildet. Zusätzlich zu den von der Klasse *Identifiable* geerbten Feldern hat die Klasse *SpecType* ein Feld *SpecAttributes*. Das Feld ist eine Liste, die Instanzen einer Subklasse von *AttributeDefinition* enthält. Es wird in der XML-Datei auf einen Tag mit dem Name „SPEC-ATTRIBUTES“ abgebildet.

Der Import und Export des Anforderungs-Modells geschieht in mehreren Schritten. Abbildung 4.13 zeigt einen Überblick über die einzelnen Schritte.

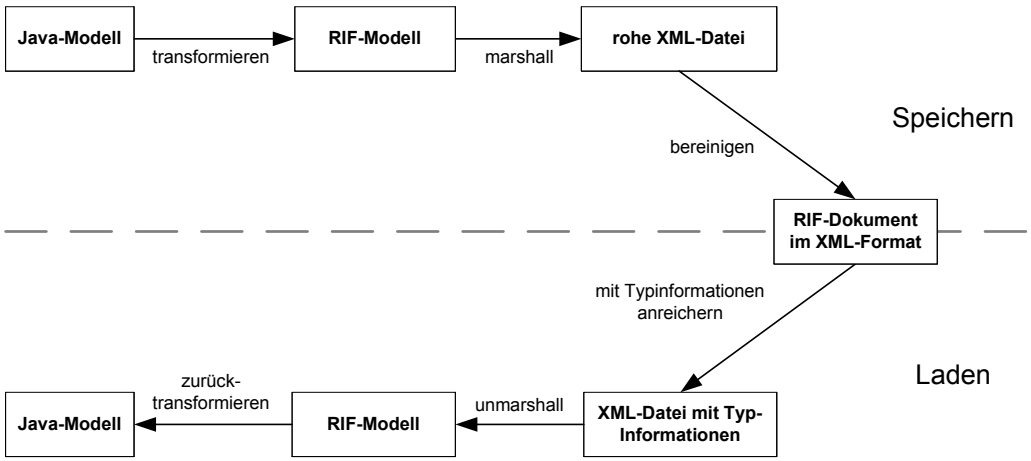



Abbildung 4.13: Überblick über den RIF-Export und -Import

Im ersten Schritt wird der Inhalt und die Struktur des Anforderungs-Modells in Instanzen des RIF-Modells transformiert (siehe Abschnitt 4.2.3.1). Anschließend wird, wie in 4.2.3.2 beschrieben, das RIF-Modell mit Hilfe der Castor-Bibliothek in eine XML-Datei geschrieben. Diese muss danach bereinigt werden, da Castor einige nicht benötigte Informationen einfügt (siehe Abschnitt 4.2.3.3). Die bis jetzt erwähnten Schritte stellen den Export dar.

Beim Importieren einer XML-Datei werden zuerst, wie in Abschnitt 4.2.3.4 beschrieben, Typinformationen in die XML-Datei eingefügt. Diese Typinformationen werden in Abschnitt 4.2.3.5 zum Laden der XML-Datei in ein RIF-Modell benötigt. Der letzte Schritt der Rücktransformation in das Anforderungs-Modell wurde noch nicht implementiert, da es für den Prototypen genügte, Anforderungen in das RIF-Format zu exportieren.

#### 4.2.3.1 Transformation

Während der Transformation wird zuerst das Anforderungs-Modell per Reflection analysiert. Es werden Listen aller verwendeten *Element*-Subklassen und aller verwendeten Rückgabewerte der Properties ermittelt. Als Properties werden Paare von Getter- und Setter-Methoden bezeichnet. So stellen z. B. die beiden Methoden *getName()* und *setName()* eine Property mit dem Name „Name“ dar. Diese Informationen werden benötigt, da die Java-Klassen, die ein Rückgabewert eines Properties sind, zu *DatatypeDefinition*-Instanzen werden. Die Java-Klassen, die eine Subklasse von *Element* sind, werden zu *SpecType*-Instanzen im RIF-Modell. Abbildung 4.14 zeigt ein Beispiel einer Term-Klasse und ihrer Umsetzung in das RIF-Modell. Bisher wurden nur die Informations-Typen des RIF-Modells definiert.

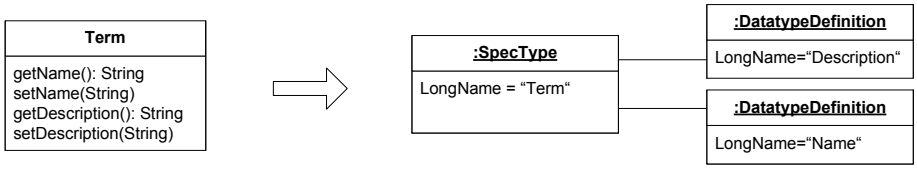



Abbildung 4.14: Umsetzung einer Klasse in eine SpecType- und mehrere DatatypeDefinition-Instanzen

Im nächsten Schritt werden alle Instanzen der *Element*-Subklassen auf *SpecObject*-Instanzen abgebildet, die *SpecObject*-Instanzen enthalten die Werte einer Instanz einer Klasse. Sie sind Informations-Elemente. Die Werte der Objekte werden wiederum mittels Reflection ausgelesen. Abbildung 4.15 zeigt die Abbildung einer Instanz der Term Klasse aus dem vorherigen Beispiel.

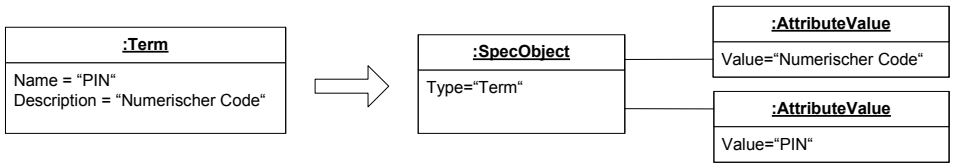



Abbildung 4.15: Umsetzung einer Instanz in eine SpecObject- und mehrere AttributeValue-Instanzen

Im letzten Schritt wird die hierarchische Struktur des Anforderungs-Modells, wie in Abbildung 4.16 gezeigt, auf eine hierarchische Struktur von *SpecHierarchy*-Instanzen abgebildet, da bisher die hierarchische Struktur der Elemente in dem Anforderungs-Modell noch nicht gespeichert wurde. Jede *SpecHierarchy*-Instanz hat einen Verweis auf das *SpecObject*, das von ihr repräsentiert wird.

Dieser Schritt stellt eine Umwandlung des Anforderungs-Modells aus dem Typsystem von Java in das Typsystem eines RIF-Dokuments dar. Es werden sowohl die Klassen als auch die Objekte transformiert.

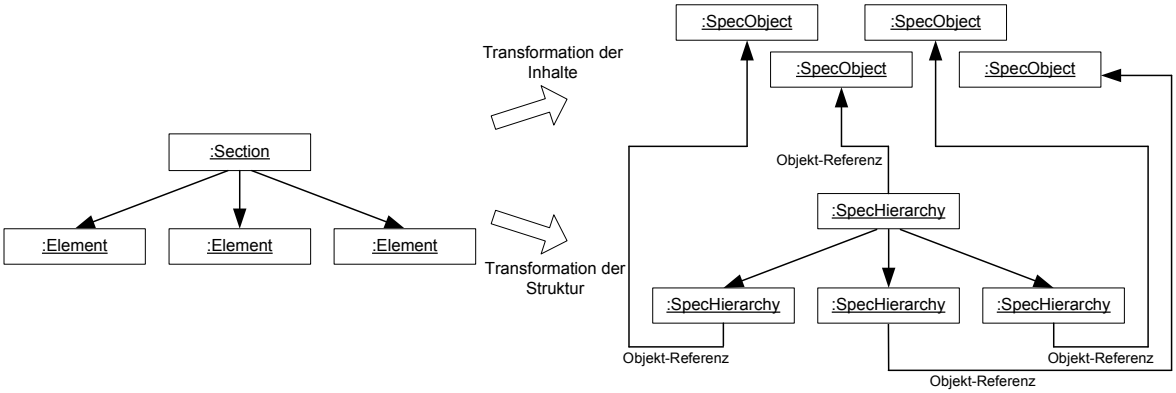



Abbildung 4.16: Umsetzung mehrerer Instanzen in mehrere SpecObject- und SpecHierarchy Instanzen

#### 4.2.3.2 Marshalling

In diesem Schritt wird das erzeugte RIF-Modell mit Hilfe von Castor in eine XML-Datei geschrieben. Dieser Vorgang wird Marshalling genannt. Ein Problem, welches beim Marshalling auftritt ist, dass z. B. unterschiedliche Datentypen auf unterschiedliche Tags abgebildet werden. So wird ein String-Wert auf das Tag „DATATYPE-DEFINITION-STRING“ und ein Integer-Wert auf das Tag „DATATYPE-DEFINITION-INTEGER“ abgebildet. Das ist ungewöhnlich, da normalerweise ein gleichbleibender Tagname und ein Attribut, das unterschiedliche Werte annehmen kann, verwendet wird. Castor sieht auch keine Einstellung vor, die diesen Fall behandelt. Daher muss ein Marshal-Listener implementiert werden, der die Klassen-Deskriptoren dynamisch modifiziert.

Dazu wird in der Klasse *RIFPersistence* in einem Marshal-Listener bei jedem zu schreibenden Objekt geprüft, ob es eine Instanz der Klasse *AttributeDefinitionSimple* oder *AttributeValueSimple* ist. Wenn ja, wird der Klassen-Deskriptor der Klasse des gefundenen Objekts so angepasst, dass der Name des Tags den verwendeten Datentyp korrekt widerspiegelt.

Dank dieser dynamischen Modifikation der Klassen-Deskriptoren kann Castor trotzdem verwendet werden.

#### 4.2.3.3 Bereinigung

Die Bereinigung ist kosmetischer Natur. So lässt Castor z. B. leere Namespace-Attribute in manchen Tags stehen, die nicht nötig sind. Nach der Bereinigung ist der Export-Vorgang abgeschlossen und die exportierten XML-Dateien entsprechen den Beispieldateien der RIF-Spezifikation.

#### 4.2.3.4 Anreicherung mit Typinformationen

Beim Importieren einer RIF-Datei tritt das Problem auf, dass in der Mapping-Datei nur die Basisklassen für einen Typ angegeben sind. So kann z. B. der Aufzählung in Zeile 3 des Listings 4.1 jede Subklasse von *AttributeDefinition* hinzugefügt werden. Um welche konkrete Subklasse es sich handelt, wird in der XML-Datei in Form der Referenzen kodiert.

Eine Referenz kann z. B. „DATATYPE-DEFINITION-STRING-REF“ lauten. Sie stellt einen Verweis auf eine Instanz der Klasse *DatatypeDefinitionString* dar, die eine Subklasse von *DatatypeDefinition* ist.

Beim Einlesen einer XML-Datei könnte zwar in dem, wie im nächsten Abschnitt beschriebenen, neu eingefügten Ereignis der Klassen-Deskriptor geändert werden (vergleiche 4.2.3.2). Allerdings wird das Ereignis zu einem zu späten Zeitpunkt ausgelöst und kann den Klassen-Deskriptor nicht mehr verändern.

Daher wird die XML-Datei vor dem Einlesen in einem Vorverarbeitungsschritt um die Typinformationen angereichert, welche in Form der Referenzen vorliegen. Die Typinformationen werden in Form von Attributen mit dem Name „xsi:type“ eingefügt, welche von Castor ausgewertet werden.

#### 4.2.3.5 Unmarshalling

In diesem Schritt wird die XML-Datei eingelesen und daraus wieder das RIF-Modell erzeugt. Diesen Vorgang nennt man Unmarshalling.

Um die XML-Dateien mit den Typinformationen wieder einlesen zu können, musste die Schnittstelle *UnmarshalListener* von Castor jedoch um ein Ereignis erweitert werden. Die Methode, die beim Eintreten des Ereignisses aufgerufen wird, wurde *elementStarting()* genannt und tritt auf, wenn ein neuer Tag in der XML-Datei bearbeitet wird. Die Modifikation der Castor-Bibliothek ist transparent, d. h. das Verhalten hat sich nach außen nicht verändert. Der modifizierte Quellcode befindet sich in dem Paket „org.exolab.castor.xml“, welches im Requirements Editor Plug-in enthalten ist. Zur besseren Auffindbarkeit wurde jede modifizierte Stelle mit dem Kommentar „#MODIFIKATION“ markiert. Mit Hilfe des neuen Ereignisses können die gleichen Modifikationen der Klassen-Deskriptoren, wie zuvor beim Schreiben der XML-Datei, vorgenommen werden. Allerdings müssen diese Änderungen beim Lesen der XML-Datei im Unmarshal-Listener vorgenommen werden.

Als letztes werden die Java-Referenzen wieder gesetzt. Diese sind verloren gegangen, da anstatt der referenzierten Elemente jetzt neue Elemente stehen, die bis auf den Identifier des referenzierten Objekts leer sind. Diese referenzierten Elemente werden in diesem Schritt gesucht und die neuen Elemente durch echte Referenzen ersetzt.

Das geladene RIF-Model muss jetzt wieder in ein Java-Modell zurück transformiert werden. Dieser letzte Schritt wurde im Prototypen noch nicht implementiert.

## 4.3 Change Log-Plug-in

Durch das Change Log-Plug-in wurde ein Änderungs-Log, wie in Abschnitt 3.3.2.4 beschrieben, implementiert. Es ist nicht von dem Requirements Editor-Plug-in abhängig. Das Plug-in enthält dabei nicht nur das Änderungs-Log, sondern auch die in Abbildung 4.17 gezeigten Klassen. Die Klassen dienen dazu, die in Abschnitt 3.3.2 vorgestellte Traceability-Architektur umzusetzen. Sie haben die gleiche Bedeutung wie in der vorgestellten Traceability-Architektur.

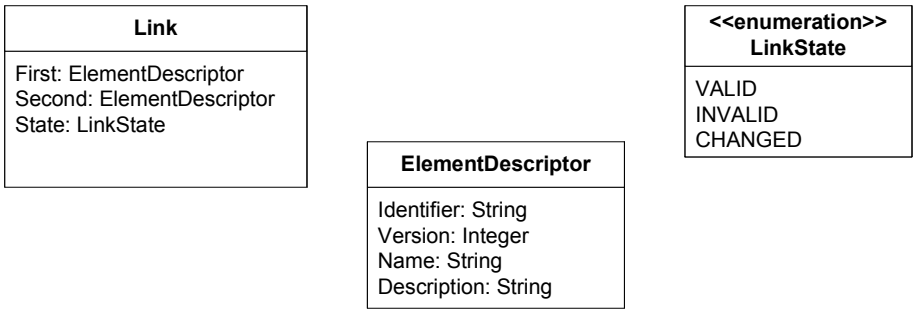



Abbildung 4.17: Überblick über die Klassen des Change Log-Plug-ins

Allerdings müssen die einzelnen Editoren die *ChangeLog*-Klasse (siehe Abbildung 4.18) verwenden, um eine editorübergreifende Traceability zu ermöglichen. So muss sich z. B. jeder Editor beim Öffnen beim Änderungs-Log registrieren und beim Schließen deregistrieren, um zu ermöglichen, dass zu einzelnen Elementen durch das Verfolgen eines Links gesprungen werden kann. Im Folgenden werden die einzelnen durch Zahlen in 4.18 markierten Blöcke von Methoden erläutert:

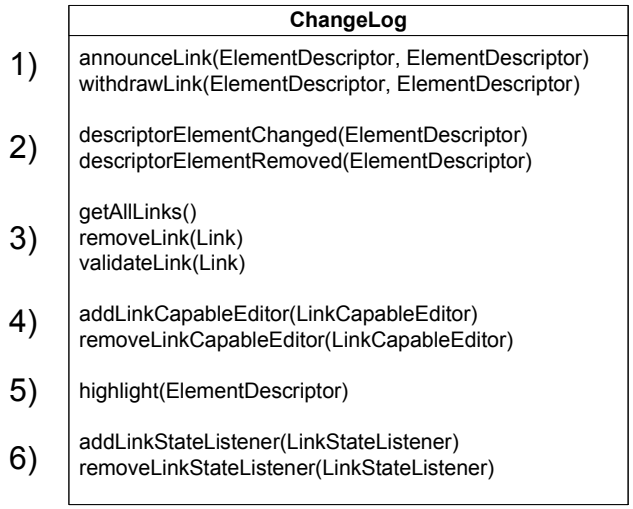



Abbildung 4.18: *ChangeLog*-Schnittstelle

1. Dem Änderungs-Log wird ein neuer Link (oder das Entfernen eines Links) zwischen den zwei angegebenen Deskriptoren gemeldet.
2. Dem Änderungs-Log wird gemeldet, dass ein Element, das durch den Deskriptor beschrieben wird, sich entweder verändert hat oder gelöscht wurde.
3. Erlaubt die Manipulation der Links von außen.
4. Erlaubt das Registrieren und Deregistrieren eines Editors, welcher die Schnittstelle *LinkCapableEditor* implementiert.
5. Mit Hilfe dieser Methode kann an alle registrierten Editoren die Aufforderung geschickt werden, das von dem Deskriptor beschriebene Element anzuzeigen. Diese Aufforderung muss an alle Editoren geschickt werden, da es nicht bekannt ist, zu welchem Editor der Deskriptor gehört. Der Editor, zu dem der Deskriptor gehört, soll in den Vordergrund gebracht werden und die zu dem Deskriptor gehörenden Elemente markieren.
6. Diese Listener erlauben es, von außen alle Statusänderungen der Links mit zu verfolgen.

Außerdem wurde noch ein Link View als Eclipse-View implementiert, der eine Liste aller Links anzeigt (siehe Abbildung 4.19). Diese Liste kann nach dem Zustand der Links gefiltert werden und erlaubt die Entfernung von ungültigen Links.

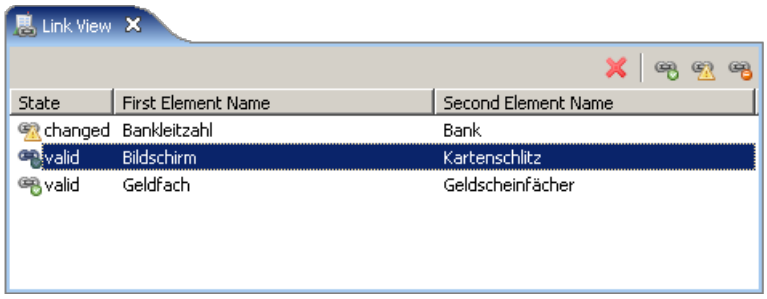



Abbildung 4.19: Link View-Screenshot mit Links

## 4.4 Erweiterungsmöglichkeiten

In diesem Abschnitt werden zuerst die Erweiterungsmöglichkeiten des Requirements Editors und danach die des Änderungs-Logs beschrieben.

### 4.4.1 Requirements Editor

Der Requirements Editor ist an mehreren Stellen erweiterbar. So können die TopLevel-Elemente einer neu erstellten *Documentation*-Instanz angepasst werden, neue *Element*-Subklassen hinzugefügt werden und die Darstellung bestehender und neuer *Element*-Subklassen

ausgetauscht bzw. festgelegt werden. Dies alles wird über eine XML-Konfigurationsdatei festgelegt. Listing 4.2 zeigt ein Beispiel.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3     <!-- A list of all top level documents, which should form the documentation. -->
4     <toplevelSections>
5        <section name="Systembeschreibung"/>
6        <section name="Anforderungsdokument"/>
7     </toplevelSections>
8     <!-- A list of mappings between an Element subclass and an ElementProxy instance.
          -->
9     <presentation>
10       <mapping elementClass="activecharts.re.documentation.complex.Glossary"
             proxyClass="activecharts.re.presentation.complex.GlossaryText"/>
11    </presentation>
12    <!-- A list of elements classes and their allowed child element classes. -->
13    <structure>
14       <element elementClass="activecharts.re.documentation.link.ExternalLinks">
15          <allowed elementClass="activecharts.re.documentation.link.ExternalLink"/>
16    </structure>
17 </configuration>
```

Listing 4.2: Beispiel einer XML-Konfigurationsdatei

### Anpassung der TopLevel-Instanzen

In dem Tag „toplevelSections“ (Zeile 4-7) können beliebig viele „section“-Tags definiert werden. Für jeden „section“-Tag wird eine *TopLevel*-Instanz mit dem als Attribut angegebenen Namen erzeugt. Eine Änderung der TopLevel-Elemente kann z. B. nötig werden, wenn sich das in Abschnitt 3.1 entwickelte Vorgehensmodell ändert.

### Hinzufügen neuer Element-Subklassen

Neue Elemente können durch Klassen hinzugefügt werden, die entweder von *Element* oder *Section* abgeleitet sind. Von welcher der beiden Klassen abgeleitet wird, ist davon abhängig, ob es sich um ein atomares Element ohne Kinder handeln soll oder nicht. In jeder Setter-Methode der abgeleiteten Klasse muss die Methode *notifyChangeUpdate()* aufgerufen werden, um eine Änderung bekannt zu geben.

Meistens wird zu einer von *Element* abgeleiteten Klasse eine von *Section* abgeleitete Container-Klasse benötigt. Allerdings kann das neue Element auch zu einer bestehenden *Section*-Subklasse im Baum hinzugefügt werden.

Damit neue Elemente überhaupt mittels des Structure Views in das Anforderungs-Modell eingefügt werden können, müssen sie in der XML-Konfigurationsdatei als erlaubte Kinder angegeben werden. Das wird mittels des „structure“-Tags (Zeile 13-18) erlaubt. Für jedes Element gibt es einen „element“-Tag. Jedes dieser „element“-Tags hat beliebig viele „allowed“-Tags. Jeder „allowed“-Tag lässt eine *Element*-Subklasse als Kinder für die in dem „element“-Tag angegebe *Section*-Subklasse zu.

#### Anpassung der Darstellung

In dem „presentation“-Tag kann angegeben werden, mit Hilfe welcher Klasse, die *ElementProxy* implementiert, eine bestimmte von *Element* abgeleitete Klasse im Requirements Editor dargestellt werden soll. Dazu wird in jedem „mapping“-Tag eine *Element*-Klasse und eine dazugehörige *ElementProxy*-Klasse angegeben. Gibt es für ein Element kein passendes „mapping“-Tag, wird zur Darstellung eine Instanz der Klasse *DefaultElementProxy* verwendet. Dieser Standard-ElementProxy zeigt lediglich den Namen und die Beschreibung des angezeigten Elements an und verwendet nur Standard-SWT-Komponenten.

Für die Darstellung der Elemente eines Abschnitts in einer Tabelle gibt es die abstrakten Klassen *GenericTextSection* und *GenericRowElement*. Eigene Klassen können von diesen beiden Klassen abgeleitet werden und von der bereitgestellten Funktionalität profitieren. So gibt es z. B. eine automatische Datenbindung zwischen einer Property eines Elements und einer Subklasse von *TextArea*.

Durch die Verwendung der *ElementProxy*-Schnittstelle wird ermöglicht, dass jede beliebige SWT-Komponente zur Darstellung verwendet werden kann.

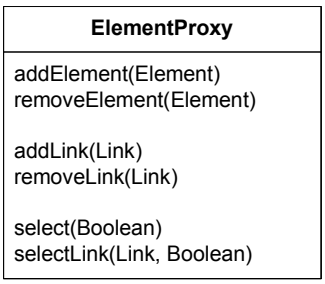



Abbildung 4.20: *ElementProxy*-Schnittstelle

Abbildung 4.20 zeigt die wichtigsten Methoden. Mit Hilfe der *addElement()* und *removeElement()* Methoden können Elemente zu einem Abschnitt hinzugefügt und entfernt werden. Bei einem Element werden die Methoden nicht benötigt, da Elemente keine Kinder haben können. Im Gegensatz dazu müssen die Methoden *addLink()* und *removeLink()* sowohl für einen Abschnitt als auch für ein Element implementiert werden. Des Weiteren gibt es Methoden, mit deren Hilfe der Documentation Manager Elemente und Links selektieren kann.

### 4.4.2 Änderungs-Log

Die *ChangeLog*-Klasse (Änderungs-Log) stellt selbst keine Erweiterungsmöglichkeiten bereit. Allerdings können Editoren das Änderungs-Log verwenden, um eine Verlinkung ihrer Elemente mit beliebigen anderen Elementen aus Editoren zu ermöglichen, welche auch das Änderungs-Log verwenden.

Um das Änderungs-Log zu verwenden, muss ein Editor folgende drei Funktionalitäten imple-

mentieren:

1. Der Editor muss sich melden, wenn ein neuer Link erstellt oder ein bestehender entfernt wird. Das ermöglicht eine zentrale Verwaltung der Links im Änderungs-Log.
2. Der Editor muss sich bei dem Änderungs-Log melden, wenn ein von einem Element-Deskriptor beschriebenes Element sich geändert hat oder gelöscht worden ist. Dies ermöglicht eine Zustandsänderung der Links, die den betroffenen Element-Deskriptor als Ziel oder Quelle haben.
3. Der Editor muss die Schnittstelle *LinkCapableEditor* implementieren und sich beim Öffnen bei dem Änderungs-Log registrieren. Wenn die Methode *highlight()* der Schnittstelle *LinkCapableEditor* implementiert wird, ist es möglich von außerhalb des Editors den Fokus auf das gewünschte Element zu setzen und es zu markieren. Das ermöglicht eine graphische Verfolgung von Links, die ihre Quelle oder ihr Ziel im Editor haben.

## 4.5 Stand der Implementierung

Die AdvancedText-Komponente verfügt bisher nur über rudimentäre Eingabemöglichkeiten und praktisch keine Komfortfunktionen. So funktionieren Tastenkürzel wie „Pos1“ und „Ende“ nicht und momentan ist es weder möglich, Text zu markieren noch Eingaben rückgängig zu machen.

Der Requirements Editor unterstützt LinkTargets, interne Links und externe Links. Die internen Links verlinken zwei Elemente bzw. LinkTargets innerhalb des Anforderungs-Modells. Externe Links verlinken ein Element bzw. LinkTarget mit einem anderen Editor, wie z. B. dem Activitydiagram Editor. Abbildung 4.21 zeigt ein Beispiel eines externen Links. Eine Anforderung aus dem Requirements Editor (orange markiert) wurde mit einer Aktion aus dem Activitydiagram Editor verlinkt (blau markiert).

Um externe Links erstellen zu können, wurde der Activitydiagram Editor aus [31] modifiziert. Mittels Drag & Drop kann ein Element aus dem Activitydiagram Editor mit einem Element bzw. LinkTarget des Anforderungs-Modells verlinkt werden. Allerdings ist die Modifikation des Activitydiagram Editors nur ein Provisorium, denn per Drag & Drop wird nicht, wie eigentlich gefordert, ein Element-Deskriptor zurückgeliefert, sondern eine GEF-spezifische Klasse. Das GEF (Graphical Editing Framework [10]) wurde zur Implementierung des Activitydiagram Editors verwendet. Somit kann der erstellte Link nicht beim Änderungs-Log registriert werden, da das Änderungs-Log zwei Element-Deskriptoren benötigt um einen Link zu verwalten. Die an dem Quellcode des Activitiydiagram Editors vorgenommenen Änderungen beschränken sich auf die Klassen *ActiveChartsEditor* und *ActiveChartDropTargetListener*. Um die Änderungen einfach auffindbar zu machen, wurde jede geänderte Stelle mit dem Kommentar „#PROVISORIUM“ markiert.

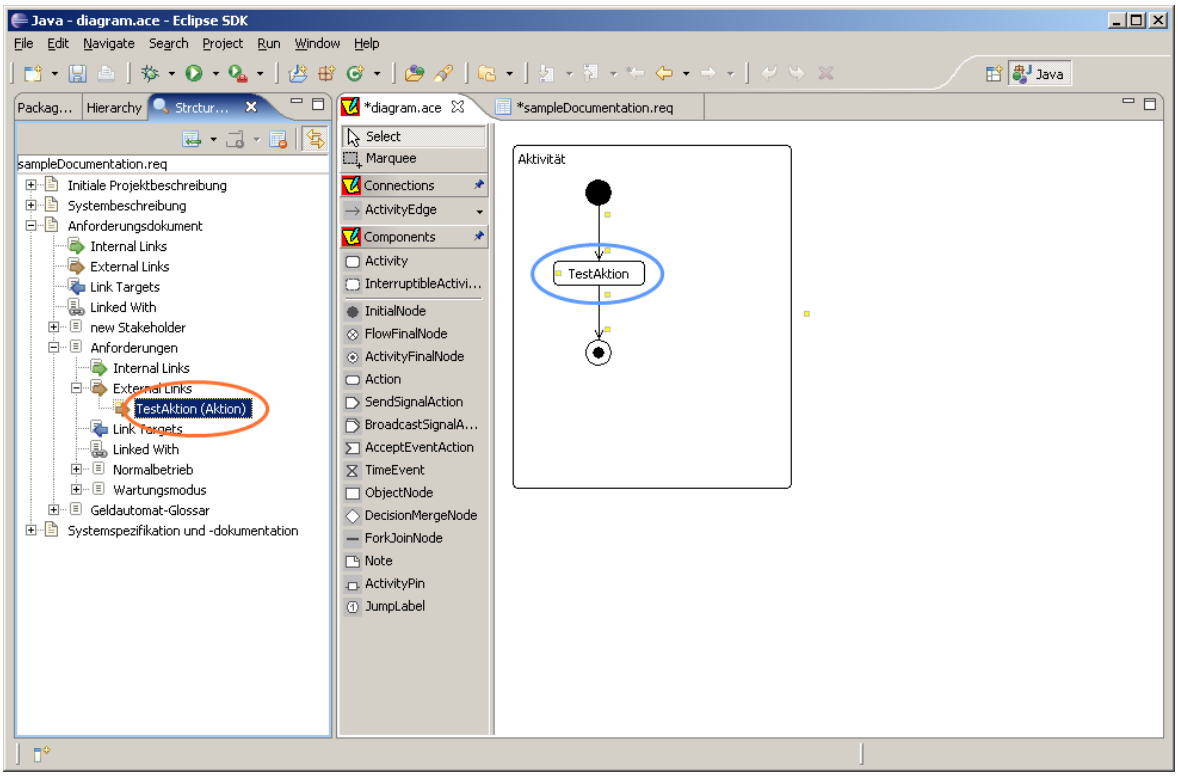



Abbildung 4.21: Beispiel: Verlinkung einer Anforderung mit einer Aktion

Des Weiteren meldet der Requirements Editor sich mit Hilfe der *LinkManager*-Klasse beim Änderungs-Log, wenn ein neuer Link erstellt oder ein bestehender entfernt wurde. Auch registriert und deregistriert er sich als Editor mit Unterstützung der *LinkCapableEditor*-Schnittstelle. Selbst die externen Links des Activitydiagram Editors werden dem Änderungs-Log gemeldet, indem von dem Requirements Editor ein Element-Deskriptor für das Element im Aktivitätsdiagramm erstellt wird. Das stellt aber wieder nur ein Provisorium dar, da dem Activitydiagram Editor der Element-Deskriptor nicht bekannt ist und er somit über keine Zuordnung zwischen dem Element im Diagramm und dem Deskriptor verfügt.

Außerdem besitzt der Requirements Editor eine Subklasse der *ElementDescriptor*-Klasse, die an ein Element des Anforderungs-Modells gebunden wird und sich bei Änderungen an dem Element selbstständig bei dem Änderungs-Log mit der Methode *descriptorElementChanged()* meldet. Dadurch kann der Zustand der Links im Änderungs-Log aktualisiert werden.

## 4.6 Fazit

In diesem Abschnitt wird zuerst die Erfüllung der Anforderungen aus Abschnitt 4.1 diskutiert und dann eine Bewertung des entwickelten Prototyps gegeben.

### 4.6.1 Erfüllung der Anforderungen

Im Folgenden soll kurz gezeigt werden, dass in dem Prototypen alle Anforderungen aus 4.1 umgesetzt werden. Dazu werden die Anforderungen an dieser Stelle nochmals stichpunktartig wiederholt:

1. Enge Integration mit der ActiveCharts IDE
2. Anforderungs-Bearbeitung
3. Anforderungs-Import und -Export
4. Unterstützung des Vorgehens aus dem Praktikum
5. Unterstützung der Traceability-Architektur
6. Einfache Erweiterbarkeit

Die erste Anforderung wurde durch die Implementierung in Form von Eclipse-Plug-ins erfüllt. Wie in Abschnitt 2.3.1 beschrieben, besteht die Neuimplementierung der ActiveCharts IDE aus einer Menge von Eclipse-Plug-ins.

Die zweite Anforderung wird durch die Verwendung der AdvancedText-Komponente erfüllt, welche eine Bearbeitung der Anforderungen ähnlich einer Textverarbeitung erlaubt. Außerdem kann die Darstellung der Elemente ausgetauscht werden, falls die Fähigkeiten der AdvancedText-Komponente nicht ausreichen.

Der RIF-Import und -Export erfüllt die dritte Anforderung. Durch das RIF-Format ist ein Import und Export der Anforderungen mittels eines offenen Austauschformats möglich.

Durch die Anpassbarkeit, die durch die XML-Konfigurationsdatei erreicht wird, werden auch die vierte Anforderung und die sechste Anforderung erfüllt.

Und schließlich erfüllt das Änderungs-Log auch die fünfte Anforderung. Somit erfüllt der Prototyp die an ihn gestellten Anforderungen.

### 4.6.2 Bewertung

Bei dem Requirements Editor-Plug-in und dem Change Log-Plug-in handelt es sich um Prototypen. Mehr hat sich in dem begrenzten Zeitraum der Diplomarbeit nicht realisieren lassen.

Positiv ist, dass gezeigt wurde, dass eine enge Integration in die ActiveCharts IDE möglich ist. Beispielhaft wurde eine Verlinkung mit dem Activitydiagram Editor gezeigt.

Des Weiteren ist das Tool durch die Verwendung der XML-Konfiguration einfach erweiterbar und kann an Prozessänderungen und neue Artefakte angepasst werden. Außerdem müssen die dargestellten Artefakte keine textuellen Artefakte sein. Denn es können beliebige Datenstrukturen in einer *Element*-Klasse gespeichert werden. So kann z. B. ein Artefakt auch

einen Graphen darstellen. Zur Darstellung werden dann z. B. keine Subklassen von *AdvancedText* verwendet, sondern ein grafischer Editor, welcher die Schnittstelle *ElementProxy* implementiert. Dies könnte z. B. zur Darstellung von SysML-Diagrammen [23] oder UML-Sequenzdiagrammen verwendet werden.

Wenn alle ActiveCharts IDE-Editoren die Schnittstelle für die Verlinkung implementieren, können praktisch alle Elemente verlinkt werden, die zur Entwicklung mit der ActiveCharts IDE nötig sind.

Dank des Exports in das RIF-Format ist ein Datenaustausch mit anderen Tools möglich. Es fallen keine Lizenzkosten für das Tool an und sein Quellcode ist offen zugänglich und kann somit an die eigenen Bedürfnisse angepasst werden.

Negativ ist der mangelnde Reifegrad der Implementierung und der beschränkte Funktionsumfang der AdvancedText-Komponente. Außerdem wurde nur ein Teil der im Abschnitt 3.3 „Traceability“ vorgestellten Ideen umgesetzt.

# 5 Zusammenfassung und Ausblick

In diesem Kapitel sollen zuerst die Ergebnisse der Diplomarbeit zusammengefasst und ein Ausblick gegeben werden, an welchen Stellen noch Erweiterungsbedarf besteht.

## 5.1 Zusammenfassung

Als Erstes wurde das Vorgehen aus dem Praktikum analysiert und in einem einfachen Wasserfallmodell formalisiert. Das wichtigste Ergebnis dieser Analyse waren die in dem Prozess vorkommenden Artefakte. Dann wurde eine Struktur erstellt, welche es ermöglicht, diese Artefakte mit einer Semantik zu versehen. Das war notwendig, da alle Artefakte entweder natürlichsprachlicher Text oder Modelle waren. Die Struktur ermöglicht eine Speicherung und Verwaltung der textuellen Artefakte, welche zuvor z. B. in einer Textverarbeitung geschrieben wurden. Sie besaßen somit keinerlei Struktur und Semantik. Um zu zeigen, dass die Struktur einen Gewinn darstellt, wurde geprüft, ob sie die Kriterien der IEEE [17] für ein gutes Anforderungsdokument erfüllt. Es war nicht möglich, eine Erfüllung formell zu beweisen. Aber es konnte dennoch gezeigt werden, dass die Struktur und die im nächsten Abschnitt vorgestellte Traceability-Architektur eine Unterstützung für die Erfüllung der Kriterien darstellt.

Es wurde vermieden, den entwickelten Prozess und die Anforderungs-Struktur als „gut“ zu bezeichnen, da ohne eine Verwendung in der Praxis keine Aussage über die Güte getroffen werden kann. Erst eine Verwendung in einem weiteren Praktikum würde zeigen, inwiefern Anpassungsbedarf besteht. Allerdings kann davon ausgegangen werden, dass durch die Strukturierung der Anforderungen und der anderen textuellen Artefakte auf jeden Fall ein Gewinn entsteht, da mehr Klarheit und Strukturiertheit als zuvor vorhanden ist.

Als Nächstes wurde eine Traceability-Architektur entwickelt, welche die gesamte ActiveCharts IDE abdeckt. Die Traceability-Architektur, deren Kernstück das Änderungs-Log ist, verbindet bestehende Ansätze zur Realisierung einer Traceability mit der ActiveCharts IDE. Die Stärke der Architektur liegt daher in der engen Verzahnung mit der ActiveCharts IDE. Aufbauend auf der vorgestellten Architektur wurde untersucht, welche Arten von Links möglich sind. Besonders lag der Fokus auf den automatisch erzeugten Links, welche einerseits innerhalb der Anforderungs-Struktur und andererseits zwischen den Anforderungen und den Modellen möglich sind. Für die automatische Erzeugung der Links wurde ein Ansatz vorgestellt, welcher Templates verwendet, um Links automatisch zu erstellen und dem Entwickler Arbeit abzunehmen. Des Weiteren wurden noch einige Konsistenzprüfungen vorgestellt, die

dafür sorgen, dass bei Änderungen die textuellen Anforderungen mit den Modellen synchron bleiben. Auch die Traceability-Architektur schafft in erster Linie mehr Klarheit. Durch die Links kann ausgedrückt werden, welche Elemente wie zusammen hängen und welche Elemente bei Änderungen betroffen sind. Die Templates sorgen für weniger Entwicklungsaufwand und durch die Verlinkung wird allgemein für mehr Robustheit in der Entwicklung gesorgt, da die zuvor impliziten Zusammenhänge zwischen den Artefakten explizit gemacht werden.

Schließlich wurde ein Prototyp erstellt, welcher die zuvor entwickelten Konzepte umsetzen sollte. Die wichtigsten Elemente, die Anforderungs-Struktur und das Änderungs-Log, wurden prototypisch umgesetzt. Dazu wurden zuerst die Anforderungen an das Tool aufgestellt und dann zwei Plug-ins entwickelt.

Das Change Log-Plug-in stellt das Änderungs-Log der Traceability-Architektur dar und stellt allgemeine Klassen zur Verwaltung von Links zur Verfügung, welche zur Realisierung von Traceability in allen ActiveCharts IDE-Plug-ins verwendet werden können.

Das zweite Plug-in, der Requirements Editor, erlaubt ein grafisches Editieren der Anforderungs-Struktur. Er kann sowohl Links zwischen den Anforderungen als auch Links mit Elementen aus externen Editoren erstellen. Zur Darstellung verwendet er die in dieser Arbeit entwickelte AdvancedText-Komponente. Sowohl die Anforderungs-Struktur als auch die Darstellung des Requirements Editors sind über eine Konfigurationsdatei anpassbar. Daher kann er sowohl an eine Änderung des Entwicklungsprozesses als auch für neue Artefakte angepasst werden. Außerdem können die textuellen Anforderungen in der Anforderungs-Struktur mit Hilfe des RIF-Formats exportiert und in anderen Requirements-Tools geladen werden.

Zwar werden längst nicht alle Funktionen im Prototyp realisiert, aber dennoch wird durch das Change Log-Plug-in der Grundstein für eine umfassende Traceabiliy in der ActiveCharts IDE gelegt. Dem Requirements Editor fehlen zwar noch Funktionen, aber auch er stellt eine Architektur zur Verfügung, welche als Basis für zukünftige Entwicklungen verwendet werden kann.

## 5.2 Ausblick

Da es sich bei dem entwickelten Tool um einen Prototypen handelt, gibt es einige interessante Punkte, an denen in weiteren Arbeiten angeknüpft werden kann. Im Folgenden werden einige Punkte vorgestellt:

#### RIF-Import und Entwicklung eines Mappings für Doors

Das zum Export verwendete RIF-Format hat Dank der Unterstützung namhafter Unternehmen der deutschen Automobilindustrie das Potential, sich als Standard zu etablieren. Das größte Problem stellt dabei die Abbildung der internen Typen der verschiedenen Tools aufeinander dar. Wie in Abschnitt 2.5 der Grundlagen beschrieben, ist für jede Kombination

zweier Tools, die Daten mittels dem RIF-Format austauschen möchten, jeweils ein Mapping nötig. Die Erstellung eines Mappings ist zwar sehr aufwändig, könnte aber dadurch gemildert werden, dass die Toolhersteller ihre internen Typen in Zukunft immer weiter angleichen. Es wird sich auf Dauer zeigen müssen, wie groß die Akzeptanz des RIF-Formats sein wird. Für den Requirements Editor würde sich ein Mapping für Doors anbieten, da Doors von Telelogic der Marktführer bei den Requirements-Tool ist (vergleiche Abschnitt 3.4.1).

Außerdem muss der letzte Schritt des RIF-Imports noch fertiggestellt werden.

#### Verbesserung der AdvancedText-Komponente

Die AdvancedText-Komponente kann noch um einige Komfortfunktionen erweitert werden. Das schließt zum einen die vollständige Unterstützung der gebräuchlichen Tastenkürzel und eine Verbesserung der Bedienbarkeit ein, z. B. um Tooltipps und eine Rechtschreibkorrektur. Zum anderen kann die AdvancedText-Komponente noch besser in Eclipse integriert werden, indem z. B. die Undo/Redo-Operationen der Eclipse-IDE unterstützt werden.

#### Anpassung anderer Editoren der ActiveCharts IDE

Die bereits für die ActiveCharts IDE existierenden Editoren können angepasst werden, um an der Traceability-Architektur teilzunehmen. Die für die Teilnahme nötigen Schritte werden in Abschnitt 4.4.2 beschrieben.

#### Änderung des Speicherorts der Links

Momentan werden die Links in den einzelnen Editoren gespeichert und beim Öffnen des Editors dem zentralen Änderungs-Log gemeldet. Das hat zwar den Vorteil, dass jeder Editor selbst für die Speicherung der Links verantwortlich ist, aber auch den Nachteil, dass die Links nur zur Verfügung stehen, wenn der entsprechende Editor geöffnet ist. Daher sollte über eine zentrale Speicherung der Links nachgedacht werden. Allerdings treten dann wieder neue Probleme auf. So ist z. B. nicht mehr sichergestellt, dass jedes Element, welches in einem Link referenziert wird, noch in den Dateien der Editoren vorhanden ist. Es könnte beispielsweise eine Bearbeitung der Datei auf einem anderen Rechner stattgefunden haben. Das Änderungs-Log kann diese Bearbeitung nicht registriert haben und es besteht die Möglichkeit, dass Links auf nicht mehr existierende Elemente verweisen. Daher muss für eine Synchronisierung gesorgt werden.

#### Template Bibliothek

Eine weitere Erweiterungsmöglichkeit ist eine zentrale Template-Bibliothek. Diese Bibliothek könnte dazu verwendet werden, beliebige Templates für beliebige Editoren zu verwalten. Diese Templates könnten dann für eine automatische Verlinkung von textuellen Artefakten und Modellen (vergleiche Abschnitt 3.3.3.2) verwendet werden.

# A Beschreibung der GUI

An dieser Stelle soll die GUI der entwickelten Plug-ins kurz vorgestellt werden. Es wird eine aufgabenorientierte Einführung gegeben. Die folgenden Abschnitte tragen jeweils die vorgestellte Aufgabe als Titel.

## A.1 Erstellung eines neuen Dokuments

Zur Erstellung eines neuen Dokuments muss im Hauptmenü „File > New > Other...“ gewählt werden. Abbildung A.1 zeigt den erscheinenden Dialog. In dem Dialog muss unter dem Knoten „ActiveCharts IDE“ entweder „Requirements Document“ oder „Requirements Document (with sample content)“ gewählt werden. Die Erstellung eines Dokuments mit Beispielinhalten ist zu Testzwecken möglich.

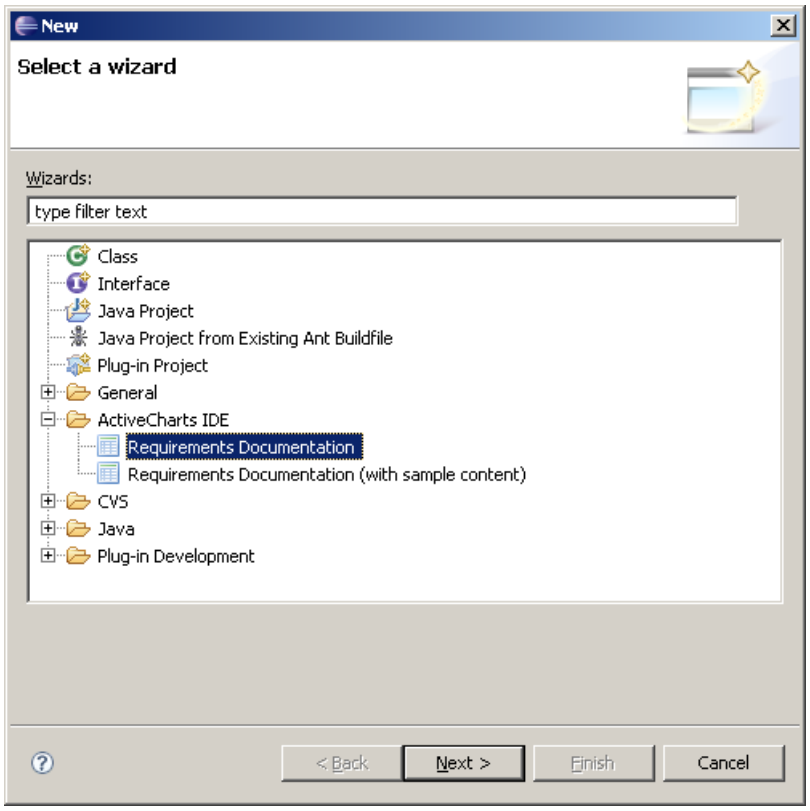



Abbildung A.1: Wizard zur Erstellung eines neuen Dokuments

## A.2 Elemente hinzufügen/löschen

Elemente können mit Hilfe des Structure Views hinzugefügt oder gelöscht werden. Die Aktionen der drei farbig markierten Buttons in Abbildung A.2 beziehen sich immer auf das gerade markierte Element in der Baumstruktur. Dieses markierte Element wird im Folgenden aktuelles Element genannt.

Die orange markierten Buttons fügen ein neues Element ein, wobei der linke Button ein neues Element vor dem aktuellen Element einfügt und der rechte Button das neue Element dem aktuellen Element als letztes Kind hinzufügt. Allerdings können jeweils nur die durch die Konfiguration (vergleiche Abschnitt 4.4) erlaubten Elemente dem aktuellen Element hinzugefügt werden.

Der blau markierte Button löscht das aktuelle Element.

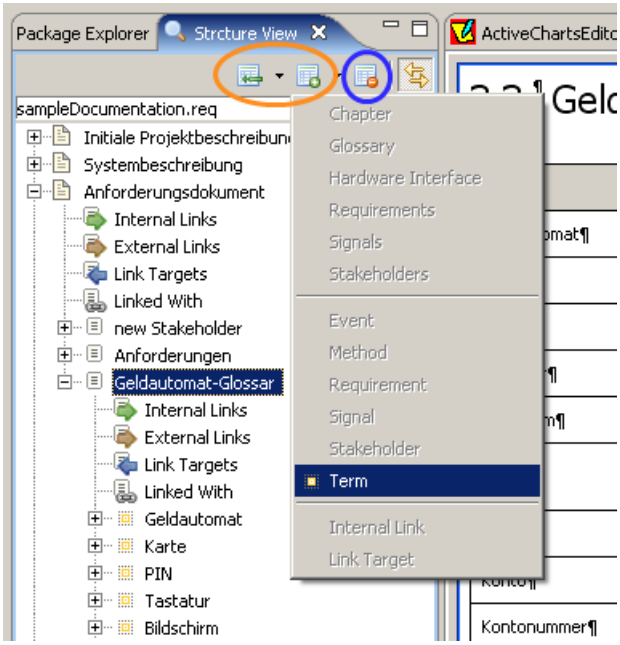



Abbildung A.2: Hinzufügen/Löschen eines Elements

## A.3 Link/LinkTarget einfügen

Abbildung A.3 zeigt das Kontextmenü, welches zum Einfügen eines neuen internen Links oder eines neuen LinkTarget verwendet wird. Das Kontextmenü wird durch einen Rechtsklick in den Text des Requirement Editors geöffnet.

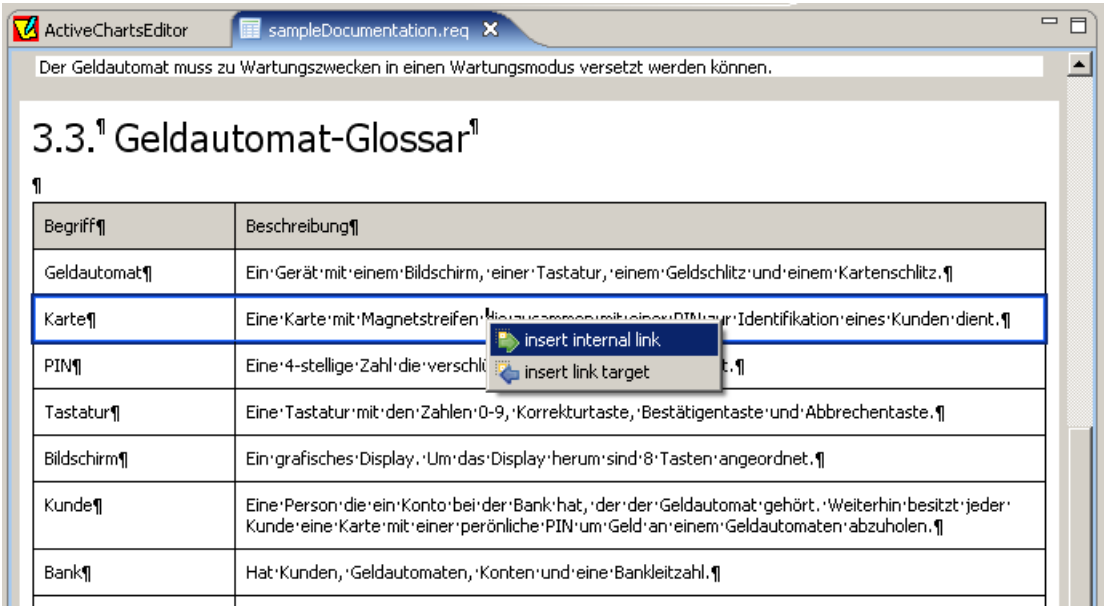



Abbildung A.3: Einfügen eines Links/LinkTargets

## A.4 Internen Link erstellen

Ein interner Link wird durch das Ziehen eines Elements oder eines LinkTargets aus dem Structure View auf einen Link im Requirements Editor erstellt.

## A.5 Externen Link erstellen

Externe Links werden durch das Ziehen eines Elements aus einem Editor, welcher bei einer Drag & Drop-Operation eine ElementDescriptor-Instanz zurückliefert, auf ein Element oder ein LinkTarget im Structure View erstellt.

## A.6 Links löschen

Links können wie normale Elemente im Structure View gelöscht werden (mit Hilfe des Buttons mit der blauen Markierung in Abbildung A.2).

## A.7 Anforderungen in das RIF-Format exportieren

Abbildung A.4 zeigt, wie ein Dokument im „Package Explorer“ exportiert wird. Das Kontextmenü wird wiederum über einen Rechtsklick geöffnet. Bevor die Datei exportiert wird,

erscheint der Dialog aus Abbildung A.5. Es kann der Dateipfad und der Name des Autors gewählt werden.

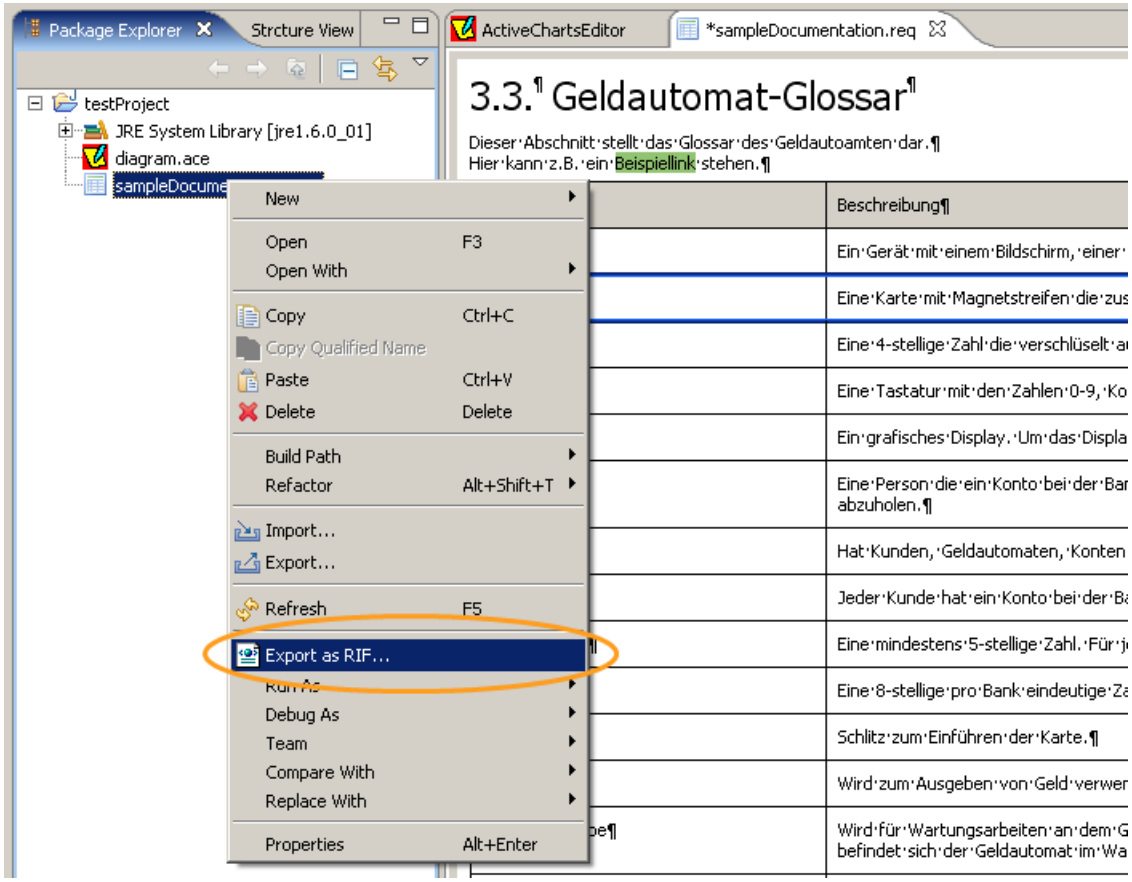



Abbildung A.4: Datei im RIF-Format exportieren

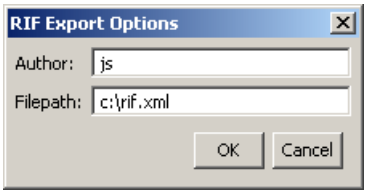



Abbildung A.5: Optionen des RIF-Exports

## A.8 Ungültige Links löschen

Im Link View können ungültige Links gelöscht werden. Der in Abbildung A.6 blau markierte Button löscht den momentan markierten Link, falls dieser den Zustand ungültig hat.

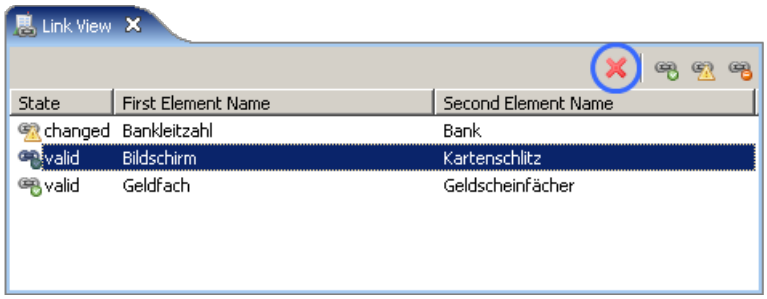



Abbildung A.6: Löschen eines ungültigen Links im Link View

# B Bekannte Probleme

Im Folgenden werden die bekannten Probleme der Implementierung aufgelistet. In vielen Fällen gibt es einen Lösungvorschlag, wie das Problem behoben werden kann. Allerdings hat sich die Realisierung des Lösungsvorschlags in dem begrenzten Zeitraum der Diplomarbeit nicht umsetzen lassen.

## B.1 Fehlende Unterscheidung zwischen dem Löschen eines Elements und eines Drag & Drop-Vorgangs

**Beschreibung:** Bisher können Elemente sich beim Change Log nur bei Änderungen melden.

**Ursache:** Drag & Drop-Operationen (lösen ein Löschen und Einfügen des Elements aus) lassen sich bisher nicht von einem richtigen Löschen unterscheiden. Der Link würde zwar gelöscht werden, aber nicht wieder mit dem Element an der neuen Position erstellt werden.

**Lösungsvorschlag:** In der Schnittstelle *RequirementListener* zusätzlich ein Ereignis Element-Moved einführen, welches angibt, dass ein Element im Dokummenten-Modell verschoben wurde.

**Bewertung:** Ein solches Ereignis wäre allgemein sehr nützlich.

**Aufwand:** Für die Einführung des neuen Ereignisses gering, allerdings für die Änderung der bisherigen Drag & Drop-Operationen groß.

## B.2 Bestehende Abhängigkeit zwischen dem Requirements Editor-Plug-in und dem Graphical Editing Framework-Plug-in

**Beschreibung:** Das Requirements Editor-Plug-in ist abhängig von dem Graphical Editing Framework-Plug-in. Das ist unnötig, da keinerlei GEF-Klassen benötigt werden.

**Ursache:** Die Abhängigkeit wird momentan noch benötigt, da der ActivityDiagram Editor bei einem Drag & Drop-Vorgang (d. h. das Verlinken eines externen Elements) keinen Element-Deskriptor, sondern eine Instanz der GEF-Klasse TemplateTransfer zurück liefert.

**Lösungsvorschlag:** Der ActivityDiagram Editor muss einen Element-Deskriptor für das zu verlinkende Element zurück liefern.

**Bewertung:** Sehr wichtig, da die Erzeugung eines Element-Deskriptors eine Voraussetzung für die Teilnahme an der Traceability-Architektur ist.

**Aufwand:** Unbekannt, da eine Einarbeitung in das GEF erforderlich ist.

## B.3 Dezentrale Speicherung der Links

**Beschreibung:** Zum jetzigen Zeitpunkt werden die Links mit den Anforderungen noch in dem Anforderungs-Modell gespeichert. Das hat zur Folge, dass die Links mit den Anforderungen nur zur Verfügung stehen, wenn der Editor geöffnet ist.

**Ursache:** Design-Entscheidung

**Lösungsvorschlag:** Schaffung einer zentralen Instanz zur Speicherung der Links.

**Bewertung:** Langfristig sicher nötig, dadurch wird auch die Speicherung der Links verallgemeinert und die einzelnen Editoren werden davon befreit.

**Aufwand:** Hoch

## B.4 Änderung eines Links wird nicht erkannt

**Beschreibung:** Wenn ein Element auf einen internen Link gezogen wird und der den internen Link umschließende Text nicht mehr geändert wird, so wird der neue Wert des Links beim nächsten Speichern nicht berücksichtigt.

**Ursache:** Der neue Wert des Links wird nicht als Änderung des Textes erkannt und somit die Datei des Editors nicht als geändert markiert.

**Lösungsvorschlag:** Der Link muss ein ElementChanged-Ereignis auslösen, wenn sein Ziel durch eine Drag & Drop-Operation verändert wird.

**Bewertung:** Bei diesem Verhalten handelt es sich um einen Fehler, der behoben werden sollte.

**Aufwand:** Mittel

## B.5 TextFlows können von innen gelöscht werden

**Beschreibung:** Als nicht entfernbar markierte TextFlows können zwar nicht von außen gelöscht werden, aber durch das Entfernen des letzten Zeichens kann der TextFlow von innen gelöscht werden. Nach dem Löschen des letzten Zeichens verschwindet er, obwohl er als nicht entfernbar markiert ist.

**Ursache:** Das Verschwinden ist das bisher korrekte Verhalten.

**Lösungsvorschlag:** Es sollte nach dem Löschen des letzten Zeichens eine Markierung stehen bleiben, welche symbolisiert, dass sich an dieser Stelle ein leerer TextFlow befindet.

**Bewertung:** Das bisherige Verhalten ist nicht intuitiv und sollte für eine bessere Benutzbarkeit verändert werden.

**Aufwand:** Hoch

## B.6 Markierung eines Links im Requirements Editor

**Beschreibung:** Bei einem Klick auf einen Link im Requirements Editor wird die Markierung im Baum nicht auf den Link gesetzt.

**Ursache:** Der Klick wird zwar abgefangen, aber der Documentation Manager hat keine Möglichkeit, einen Link zu markieren.

**Lösungsvorschlag:** Einführung einer LinkProxy-Schnittstelle.

**Bewertung:** Das hätte den Vorteil, dass sowohl die Elemente als auch die Links durch eine Proxy-Schnittstelle ansprechbar wären.

**Aufwand:** Hoch, der Documentation Manager müsste geändert werden. Es wären relativ große Architekturänderungen nötig.

## B.7 Interaktive Veränderung der Breite einer Tabellenspalte

**Beschreibung:** Wenn die Zelle einer Tabelle nicht breit genug ist, um den Inhalt vollständig anzuzeigen, kann die Zellenbreite nicht mit der Maus verändert werden.

**Ursache:** Diese Feature ist noch nicht implementiert.

**Lösungsvorschlag:** Allgemein könnte das Verschieben entweder ähnlich dem Verschieben eines *TextFlow*s realisiert werden oder es wird explizit als Operation bei Tabellen implementiert.

**Bewertung:** Es ist noch unklar, wie dieses Feature in die bestehende Architektur am transparentesten integriert werden kann.

**Aufwand:** Hoch, erfordert tiefe Eingriffe.

## B.8 Layout des Requirements Editor wird manchmal nicht aktualisiert

**Beschreibung:** Das Layout wird bei einer Größenänderung eines *ElementProxies* manchmal nicht richtig aktualisiert.

**Ursache:** Unbekannt. Unter Umständen wird in manchen Situationen vergessen, ein ContentSizeChanged-Ereignis zu schicken.

**Lösungsvorschlag:** Es muss geprüft werden, ob an allen relevanten Stellen ein ContentSizeChanged-Ereignis geschickt wird.

**Bewertung:** Erfordert nochmaliges genaues Debugging.

**Aufwand:** Mittel

## B.9 Im Requirements Editor verschobene Links werden im Anforderungs-Modell nicht mitverschoben

**Beschreibung:** Wenn ein interner Link in einem ElementProxy in ein anderes Element verschoben wird, wird das dazugehörige Link-Element im Baum nicht mitverschoben.

**Ursache:** Da es sich um einen Drag & Drop-Vorgang handelt, werden die ElementRemoved- und ElementInserted-Ereignisse umgangen. Somit wird das Anforderungs-Modell nicht über das Verschieben des Links informiert.

**Lösungsvorschlag:** Es könnte ein Ereignis wie TextFlowMoved eingeführt werden, da bis jetzt TextFlows die einzigen verschiebbaren Elemente sind. Eine allgemeine Lösung wäre es, einen Mechanismus zum Verschieben von Glyphs zu schaffen und dazu ein allgemeines Ereignis GlyphMoved einzuführen.

**Bewertung:** Ein solches Ereignis wäre allgemein sehr nützlich.

**Aufwand:** Wiederum für die Einführung des neuen Ereignisses gering, allerdings für die Änderung der bisherigen Drag & Drop-Operationen groß.

# C Inhalt der CD

Im Folgenden werden zuerst die Ordnerstruktur der beigelegten CD-ROM und die Voraussetzung für die Verwendung der entwickelten Plug-ins vorgestellt. Schließlich wird ein grober Überblick über die Organisation des Quellcodes gegeben.

## C.1 Ordnerstruktur

Abbildung C.1 zeigt den Inhalt der beigelegten CD-ROM. Der Quellcode liegt in Form eines Eclipse-Workspaces im Verzeichnis „Eclipse Workspace“ vor. Der Workspace muss zur Verwendung in Eclipse als aktueller Workspace angegeben werden (im Hauptmenü „File > Switch Workspace....“). Das Verzeichnis „JavaDoc“ enthält die aus den Javadoc-Kommentaren des Quellcodes generierte API-Dokumentation. Die Datei „Diplomarbeit.pdf“ ist die Diplomarbeit in digitaler Form.

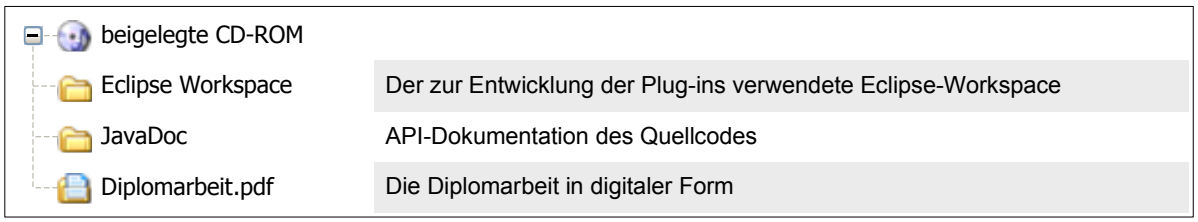



Abbildung C.1: Inhalt der beigelegten CD-ROM

## C.2 Voraussetzung

Folgende Voraussetzung sind für die Verwendung des Eclipse-Workspaces zu erfüllen:

- Eclipse 3.2.2
- Graphical Editing Framework (GEF) in der Version 3.2.2

Die in dieser Arbeit entwickelten Plug-ins hätten eigentlich keine externen Abhängigkeiten, denn die modifizierte Version der Castor-Bibliothek ist bereits in dem Eclipse-Workspace enthalten. Das GEF-Plug-in wird allerdings von dem beigelegten Activitydiagram Editor benötigt (siehe Abschnitt B.2 „Bestehende Abhängigkeit zwischen dem Requirements Editor-Plug-in und dem Graphical Editing Framework-Plug-in“).

## C.3 Quellcodeübersicht

Tabelle C.1 zeigt einen Überblick über die wichtigsten Pakete des Quellcodes. Die Liste der Pakete ist nicht vollständig und soll nur als Leitfaden für einen schnelleren Überblick über den Aufbau des Quellcodes dienen.

| Paketname | Beschreibung |
|---|---|
| activecharts.re.configuration | Enthält die Wrapper-Klasse für die Konfiguration des Requirements Editors (siehe 4.4.1). |
| activecharts.re.documentation | Enthält das Anforderungs-Modell (siehe 4.2.1) mit allen Elementen und die Section Registry (siehe 4.2). |
| activecharts.re.plugin.requirementseditor | Enthält den Requirements Editor, den Documentation Manager und den Numeration Manager (siehe 4.2). |
| activecharts.re.plugin.structureview | Enthält den Structure View (siehe 4.2). |
| activecharts.re.presentation | Enthält die ElementProxy-Klassen (siehe 4.4.1), welche im Requirements Editor zur Darstellung der Elemente aus dem Anforderungs-Modell verwendet werden. |
| activecharts.re.rif | Enthält die Klassen des RIF-Imports und -Exports. Außerdem sind die Java-Klassen des UML-Modells aus der RIF-Spezifikation enthalten (siehe 4.2.3). |
| activecharts.re.text.appearance | Enthält die implementierungsunabhängigen Schnittstellen des Dokumenten-Modells. |
| activecharts.re.text.swt | Enthält die AdvancedText-Komponente und ihre Manager (siehe 4.2.2). |
| activecharts.re.text.structure | Enthält das Dokumenten-Modell (siehe 4.2.2.1). |
| activecharts.traceability | Enthält Klassen zur Unterstützung des Änderungs-Logs (siehe 4.3). |
| activecharts.traceability.plugin.linkview | Enthält den Link View (siehe 4.3). |
| org.exolab.castor.xml | Enthält die Klassen der Castor-Bibliothek, welche bei der Einführung des neuen Ereignisses aus Abschnitt 4.2.3.5 geändert werden mussten. |
| de.uniUlm.activecharts | Enthält die modifizierte Version des Activity-diagram Editors (siehe 4.5). |

Tabelle C.1: Überblick über die wichtigsten Pakete des Quellcodes

# Literaturverzeichnis


[1] Aizenbud-Reshef, Netto ; Nolan, Brian T. ; Rubin, Julia ; Shaham-Gafni, Yael: Model traceability. In: *IBM Systems Journal* 45, Issue 3 (2006), S. 515–527

[2] Alexander, Ian F. ; Stevens, Richard: *Writing Better Requirements*. Addison-Wesley, 2002

[3] Arnold, Robert ; Bohner, Shawn: *Software Change Impact Analysis*. IEEE Computer Society Press, 1996

[4] Besstschastnich, Alex: *Dokumentenaustausch nach RIF Standard in DOORS*, Fachhochschule Münster - Fachbereich Elektrotechnik und Informatik, Diplomarbeit, 2006

[5] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-orientierte Software-Architektur*. Addison-Wesley, 2000

[6] Clayberg, Eric ; Rubel, Dan: *Eclipse. Building Commercial-Quality Plug-ins*. Addison-Wesley, 2006

[7] Cleland-Huang, Jane ; Chang, Carl K. ; Christensen, Mark: Event-Based Traceability for Managing Evolutionary Change. In: *IEEE Trans. Softw. Eng.* 29 (2003), Nr. 9, S. 796–810. – ISSN 0098–5589

[8] DaimlerChrysler TSS GmbH: *MDA Success Story ePEP erfolgreich mit Model Driven Architecture*. Stand: 3. Juli 2007. – http://www.interactive-objects.com/customers/success-stories/successstory_dc_tss_mdo_german.pdf

[9] Deutsche Bank Bauspar AG: *Deutsche Bank Bauspar AG Uses ArcStyler to Embed Existing Cobol Mainframe Application into Modern Web-Based Systems*. Stand: 3. Juli 2007. – http://www.omg.org/mda/mda_files/SuccessStory_DBB_4pages.pdf

[10] Eclipse Foundation: *Graphical Editing Framework (GEF)*. Stand: 3. Juli 2007. – http://www.eclipse.org/gef/

[11] Eclipse Foundation: *StyledText (Eclipse Platform API Specification)*. Stand: 7. Juli 2007. – http://help.eclipse.org/help32/nftopic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/custom/StyledText.html

[12] EXTESSY: *EXERPT*. Stand: 7. Juli 2007. – http://www.extessy.com/?id=7d23c1b00b73d37ee7d33ade11293cbb

[13] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Desing Patterns*. Addison-Wesley, 1995

[14] Herstellerinitiative Software: *Requirements Interchange Format (RIF) Specification*. Stand: 3. Juli 2007. – http://www.automotive-his.de/download/Docu_Requirements_Interchange_Format.pdf

[15] Hood, Colin ; Kress, Andreas ; Stevenson, Robert ; Versteegen, Gerhard ; Wiebel, Rupert. *IX-Studie Anforderungsmanagement*. Heise Verlag. 2005

[16] IEEE: Standard Glossary of Software Engineering Terminology / The Institute of Electrical and Electronics Engineers, Inc. 1990. – Forschungsbericht

[17] IEEE: Recommended Practice for Software Requirements Specifications / The Institute of Electrical and Electronics Engineers, Inc. 1998. – Forschungsbericht

[18] IEEE: Standard for Software Reviews / The Institute of Electrical and Electronics Engineers, Inc. 1998 (reapproved 2002). – Forschungsbericht

[19] Kleppe, Anneke ; Warmer, Jos ; Bast, Wim: *MDA Explained*. Addison-Wesley, 2003

[20] Leach, Paul J. ; Mealling, Michael ; Salz, Rich: *A Universally Unique IDentifier (UUID) URN Namespace*. Stand: 3. Juli 2007. – RFC 4122: http://www.ietf.org/rfc/rfc4122.txt

[21] Müller, Uwe ; Sälzer, Kerstin: *Requirements Engineering Empirie*. Masterseminar Informatik. Mai 2006. – Hochschule Mannheim

[22] Object Management Group: *Model Driven Architecture*. Stand: 3. Juli 2007. – http://www.omg.org/mda/

[23] Object Management Group: *SysML Spezifikation*. Stand: 3. Juli 2007. – http://www.omgsysml.org/#Specification

[24] Object Management Group: *UML Spezifikation*. Stand: 3. Juli 2007. – http://www.omg.org/technology/documents/formal/uml.htm

[25] Oestereich, Bernd: *Analyse und Design mit UML 2*. Oldenbourg Verlag, 2005

[26] Partsch, Helmuth: *Requirements-Engineering systematisch*. Springer Verlag, 1998

[27] Partsch, Helmuth: *Requirements Engineering*. Universität Ulm: Institut für Programmiermethodik und Compilerbau, 2006. – Folien zur Vorlesung

[28] Potts, Colin ; Takahashi, Kenji: An active hypertext model for system requirements. In: *IWSSD '93: Proceedings of the 7th international workshop on Software specification and design*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1993. – ISBN 0–8186–4360–9 (PAPER), S. 62–68

[29] QA Systems: *IRqA-Integral Requisite Analyzer*. Stand: 7. Juli 2007. – http://www.qa-systems.de/html/deutsch/produkte/irqa/irqa.php

[30] Rupp, Chris ; SOPHIST GROUP: *Requirements-Engineering und -Management*. Hanser Fachbuchverlag, 2004

[31] Schwendemann, Eva: *Umfassende Anforderungsdefinition und prototypische Implementierung eines intelligenten Editors für UML 2 Aktivitätsdiagramme*, Universität Ulm - Institut für Programmiermethodik und Compilerbau, Diplomarbeit, 2007

[32] Seidel, Tom: *rich client 2.0 - richhtml4eclipse*. Stand: 6. Juli 2007. – http://www.richclient2.eu/richhtml4eclipse/

[33] Software Engineering Institute: *CMMI (Capability Maturity Model Integration)*. Stand: 3. Juli 2007. – http://www.sei.cmu.edu/cmmi/

[34] Sun Microsystems, Inc.: *JTextPane (Java Platform SE 6))*. Stand: 7. Juli 2007. – http://java.sun.com/javase/6/docs/api/javax/swing/JTextPane.html

[35] Telelogic: *Telelogic DOORS: A market-leading solution for Requirements Management*. Stand: 4. Juli 2007. – http://www.telelogic.com/corp/products/doors/index.cfm

[36] The Atlantic Systems Guild: *Volere Requirements Resources*. Stand: 3. Juli 2007. – http://www.volere.co.uk/

[37] TNI-Software: *Reqtify Users webseite*. Stand: 3. Juli 2007. – http://users.reqtify.tni-software.com/?p=home

[38] Werth, Dirk: *Enterprise Resource Planning*. Universität Saarbrücken: Institut für Wirtschaftsinformatik, 2007. – Folien zur Vorlesung

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer, als der angegebenen, Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch in keiner anderen Prüfungsbehörde vorgelegen.

Ulm, 16. Juli 2007

_____

Jan Scheible