Studienarbeit

Entwurf und Implementierung eines Modellierungskonzepts zur Beschreibung und Simulation von Applikationslogik

Martin Girschick

Betreuerin: Dipl. Inform. Claudia Preß

TU-Darmstadt · Fachbereich Informatik · Mai 2001

Professor Dr.-Ing. José Encarnação
Institut für Informationsverwaltung
und Interaktive Systeme
Fachgebiet Graphisch-Interaktive Systeme
Fachbereich 20 (Informatik)

Rundeturmstraße 6
64283 Darmstadt
Telefon (06151) 16 3478, 155-130
Telefax (06151) 155-430
Email jle@igd.fhg.de

Aufgabenstellung für die Studienarbeit
des Herrn cand. Inform. Martin Girschick
(Matrikel Nr. 570 666)

Thema: Entwurf und Implementierung eines Modellierungskonzepts zur Beschreibung und Simulation von Applikationslogik.

Die Anforderungen an Applikationen und deren Architekturen sind in den letzten Jahren stark gewachsen - heutige Applikation müssen oftmals Internet-fähig, verteilt, sicher, skalierbar und verfügbar sein. Dies führt zu erhöhten Anforderungen im Software-Entwurf. Objektorientierte Modellierungssprachen wie UML versuchen daher den komplexen Entwicklungsproze eines Softwaresystems zu unterstützen. UML erleichtert objektorientierte Analyse und Entwurf von Softwaresystemen und ermöglicht einen unterstützten Entwicklungsproze. Dazu bietet UML dazu verschiedene Diagrammtypen wie Klassen-, Interaktions- und Aktivitätsdiagramme. Eine Unterstützung des Software-Entwurfs hinsichtlich der Visualisierung der Ablauflogik eines Softwaresystems ist jedoch nicht hinreichend gegeben.

Im Rahmen der Studienarbeit soll eine Analyse der derzeit verfügbaren objektorientierten Werkzeuge (z.B. Rational Rose, Select, TogetherJ) bezogen auf deren Fähigkeit zur Beschreibung und Simulation von Applikationslogik durchgeführt werden. Unter Berücksichtigung der gewonnenen Ergebnisse soll ein Modellierungskonzept für die Beschreibung und Simulation von Applikationslogik erstellt werden. Ziel dieses Konzeptes ist es, dem Entwickler in allen Stadien des Projektes die Verifikation und Präsentation dieser Ablauflogik zu ermöglichen.

In einer beispielhaften Implementierung soll die Praxistauglichkeit dieses Konzepts überprüft werden. Dabei soll ein Prototyp zur Visualisierung von Programmabläufen mit Repräsentationen der Modellierungselemente entworfen werden. Besondere Herausforderungen dieser Studienarbeit sind einerseits der Entwurf eines geeigneten Metadatenformats unter Berücksichtigung vorhandener Standards wie XML und andererseits die Implementierung einer anschaulichen Repräsentation der Modellierungselemente.

Darmstadt, den 15.05.2000

Betreuerin: Dipl.-Inf. Claudia Preß

Prof. Dr.-Ing. J. L. Encarnação

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Studienarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden.

Pfungstadt, Mai 2001 _____ *(Martin Girschick)*

Danksagung

Ich möchte mich an dieser Stelle bei meinen Eltern bedanken, die mir ein Studium an der TU-Darmstadt ermöglicht haben. Vielen Dank auch an meine Freunde und Kommilitonen, die mir einerseits etwas Ablenkung von der Studienarbeit geboten haben und andererseits viele Anregungen gegeben haben. Besonders erwähnt seien hier Joyce Wittur und Philipp Matthias Hahn.

Die Idee dieser Arbeit entstand nach einem Praktikum am Fachgebiet *Praktische Informatik*. Dieses Praktikum kann ich jedem Informatiker ans Herz legen, da man sich hier mit dem Gebiet der Softwareentwicklung wirklich auseinandersetzen muß. Das Praktikum hat mir besonders viel Spaß gemacht, was nicht zuletzt an dem wunderbaren Projektteam *Mind and Magic* lag. Danke an dessen Mitglieder, an Professor Henhapl und seine Mitarbeiter.

Die Studienarbeit selbst entstand am Fachgebiet *Graphisch Interaktive Systeme*. Mein Dank geht an die Mitarbeiter der Abteilung INI-SC und natürlich ganz besonders an meine Betreuerin Claudia Preß. Danke auch an Wolfgang Puchtler und Professor José Encarnação. Die Zeit am Fraunhofer IGD und mein Aufenthalt am Partnerinstitut CRCG in Providence (USA) werde ich bestimmt nicht so schnell vergessen.

Martin Girschick

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den letzten Jahrzehnten hat die Softwaretechnologie rasante Fortschritte gemacht. Am Anfang war Softwareentwicklung stark von den Entwicklungen der Hardware geprägt, im Laufe der Zeit entwickelte sich jedoch die Software zu dem dominierenden Faktor. Durch die beständige Fortentwicklung der Softwaretechnologie fanden abstraktere Konzepte Einzug in den Softwareentwicklungsprozeß. Visualisierungentechniken zur Darstellung der Funktionsweise von Programmen wurden entwickelt. Sicherlich der bekannteste Vertreter ist hier das Flußdiagramm.

Prozedurale Programmierung und schließlich die Objektorientierung ermöglichten eine bessere Modularisierung und eine klarere Trennung der verschiedenen Teile eines Softwaresystems. Eines der Hauptprobleme der Softwareentwicklung wurde dadurch aber noch nicht gelöst – Kommunikationsprobleme zwischen Softwareentwicklern und Auftraggebern führten oft dazu, daß Softwaresysteme entwickelt wurden, die nicht der Aufgabenstellung entsprachen. Was fehlte war eine allgemein anerkannte Methode zur Softwarentwicklung, die sowohl die Bedürfnisse der Programmierer als auch der Auftraggeber zu gleichen Teilen berücksichtigt. Dabei sollte sowohl ein Vokabular als auch zu gewissen Teilen eine Vorgehensweise definieren werden, die beschreibt, wie man ein System spezifiziert und entwickelt. Das Vokabular kann sich sowohl durch wohldefinierte Ausdrücke als auch durch graphische Ausdrucksmöglichkeiten definieren. Verschiedene Techniken wie Coad-Yourdon, OMT (Object Modelling Technique) oder die Booch-Methode entstanden (siehe [2]). Diese mündeten in die Entwicklung der *Unified Modelling Language*. Diese Modellierungssprache definiert eine Menge von Diagrammen, die zur Beschreibung von Software gedacht sind, aber auch in anderen Bereichen Einsatz finden.

Sicherlich das wichtigste UML-Diagramm ist das Objekt- oder auch Klassendiagramm. Es definiert die Objekte (die später durch Klassen implementiert werden) und deren Attribute und Operationen. Vererbung wird durch diesen Diagrammtyp ebenfalls dargestellt. Um die Interaktionsweise dieser Objekte untereinander zu beschreiben bietet UML Sequenz- und Kollaborationsdiagramme an. Dabei wird versucht, durch mehrere Diagramme ein Softwaresystem möglichst vollständig zu beschreiben. Diese Diagramme dienen als Basis zur Entwicklung des Systems und passen sich bei Verwendung geeigneter Modellierungswerkzeuge im Laufe der Entwicklung an.

UML wird bereits bei der Softwareentwicklung eingesetzt, jedoch nur sehr stiefmütterlich. Beispielsweise werden Sequenzdiagramme nicht oder nur selten verwendet. Meist wird versucht die wichtigsten Aspekte durch ein Klassendiagramm zu beschreiben. Bei der

Entwicklung werden diese Klassendiagrammen sehr schnell recht umfangreich, da sie alle benötigten Klassen darstellen, diese aber zum Verständnis der Software nicht notwendig sind.

Ziel dieser Arbeit ist es, ein Verfahren zu entwickeln, was das Verständnis komplexer Software verbessert. Gerade verteilten Anwendungen erreichen schnell eine Komplexität die durch einfache Diagramme nicht zu fassen ist. Dazu gehört insbesondere die Applikationslogik, die bei verteilten Komponenten sehr unübersichtlich werden kann.

Wie bereits erwähnt, ist die graphische Visualisierung von Software sehr hilfreich bei dem Verständnis von Software. Auf der Basis der Unified Modelling Language wird eine erweitere Visualisierungstechnik entwickelt. Diese soll sowohl bei der Spezifikation und dem Entwurf als auch zur Dokumentation geeignet sein. Um eine gute Integration in die Dokumentation zu ermöglichen wird ein webbasiertes System entwickelt. Um eine Integration in den Softwareentwicklungsprozeß zu ermöglichen werden geeignete Schnittstellen und Austauschformate definiert.

Zuerst wird jedoch in Kapitel 2 der aktuelle Forschungsstand und die derzeit verfügbaren Werkzeuge untersucht. Darauf aufbauend wird in Kapitel 3 ein Verfahren zur Darstellung von Applikationslogik mit Hilfe von UML-Diagrammen entwickelt.

Kapitel 4 untersucht die derzeit verfügbaren Austauschformate und Programmierschnittstellen, um die im vorhergehenden Kapitel entwickelte Technik sinnvoll umsetzen zu können. Daraus hervorgehend wird in den beiden folgenden Kapiteln ein Austauschformat für die Modelldaten und die Ablauflogik entwickelt.

Es schließt sich der Entwurf eines Prototyps an, der die Visualsierungstechnik beispielhaft umsetzt. Kapitel 8 bildet einen abschließenden Überblick über die gewonnenen Erkenntnisse.

Kapitel 2

Stand der Technik

Dieses Kapitel beschäftigt sich zuerst mit den bisherigen Forschungsergebnissen im Bereich der Sofware-Visualisierung und -Simulation. Anschließend folgt ein Überblick über die am Markt befindlichen CASE-Tools und deren Fähigkeiten, Applikationslogik graphisch darzustellen.

2.1 Forschungsergebnisse

2.1.1 Softwarevisualisierung allgemein

Der Bereich der Software-Visualisierung ist zwar noch relativ jung, dennoch gibt es bereits eine Vielfalt von Ansätzen auf diesem Gebiet. Bereits seit langem sind Flußdiagramme bekannt, die für kurze Abläufe bereits sehr übersichtlich die Verzweigungsstruktur darstellen können. Zur Visualisierung von Datenstrukturen werden oft Graphen verwendet. Die beiden Beispiele zeigen sehr gut die beiden unterschiedlichen Ansätze, die im Bereich der Softwarevisualisierung anzutreffen sind: Auf der einen Seite die Algorithmenanimation, die Abläufe in Programmen darstellt, auf der anderen Seite die Visualisierung von Datenstrukturen.

Im Bereich der Algorithmenanimation existiert eine Vielzahl von Systemen. Eines der ersten (BALSA-I) stammt aus dem Jahre 1988 und wird in dem Buch [9] von M. H. Brown vorgestellt. Später folgt BALSA-II sowie andere Systeme wie TANGO[35], XTANGO[36], POLKA und SAMBA welche allesamt an dem *Georgia Institute of Technology* (weitere Informationen siehe [34]) entwickelt wurden.

Gerade im Bereich der Lehre werden solche Systeme oft verwendet, um komplexe Algorithmen besser darstellen zu können. Nicht nur deshalb werden immer bessere Systeme mit unterschiedlichsten Zielsetzungen in diesem Gebiet entwickelt. Der GI-Workshop *Software Visualisierung 2000* beschäftigte sich deshalb zu großen Teilen mit diesem Thema. Beispielsweise das System VIVALDI 1[1], welches an der Fachhochschule Trier entwickelt wurde [26]. Dieses Framework ermöglicht es durch Instrumentierung des Sourcecodes Animationen zu erzeugen. So lassen sich beispielsweise Sequenzdiagramm-ähnliche Grafiken erstellen. Besonders viel Wert wurde auf Eingriffsmöglichkeiten während der Anima-

[1] VIsualisierung Von Ausgewählten Lehrinhalten Der Informatik

tion gelegt, ein fest vorgegebener Ablauf reicht meist nicht aus, um einen Algorithmus zu verstehen.

Eine ähnliche Zielsetzung verfolgen auch die auf dem Workshop vorgestellten Systeme GANIMAL der Universität des Saarlandes und LEDA (Universität Trier). Einen anderen Aspekt betont das System KIEL (Universität Kiel) welches sich mit der Darstellung von funktionalen Programmen beschäftigt.

Die Darstellung von Datenstrukturen ist zwar mit manchen dieser Werkzeuge möglich, jedoch existieren auch Systeme, die zur Visualisierung von Datenstrukturen besser geeignet sind. Der graphische Debugger DDD (Universität Passau), der ebenfalls in einem Vortrag auf dem Workshop vorgestellt wurde ([41]), hat bereits eine große Anwenderschaft gefunden. Er ermöglicht die Darstellung von Datenstrukturen während der Laufzeit ohne das der Sourcecode modifiziert werden muß. Andere ähnliche Systeme, die teilweise die Instrumentierung des Sourcecodes voraussetzen sind VCC, ZSTEP 95, LENS und DUEL.

Die meisten dieser Techniken wurden für prozedurale Programme und einfache Datenstrukturen entwickelt. Zur Visualisierung von objektorientierter Software sind sie deshalb nicht geeignet. Dies ist sicherlich auch ein Grund dafür, daß sie zum Entwurf von Software nicht eingesetzt werden. Als einer der Ersten beschäftigte sich John T. Stasko mit der Visualisierung objektorientierter Sofware (siehe [33]). Das daraus entstandene System GROOVE[2] enthielt bereits Visualisierungstechniken, die später auch in UML wiederzufinden waren. Auffällig an GROOVE ist jedoch die Fähigkeit, Programme in ihrem Ablauf darzustellen. Dies wurde durch Animation realisiert. Mit der statischen Visualisierung beschäftigten sich Booch ([5]) und Rumbaugh (OMT [29]), beide Ansätze mündeten letztendlich in UML [6].

2.1.2 Softwarevisualisierung unter Einbeziehung der UML

Die *Unified Modelling Language* stellt an sich schon ein sehr gutes Mittel zur Visualisierung dar. Zur Erstellung der Diagramme wurde bereits eine unüberschaubare Menge von Programmen entwickelt, auf die in Abschnitt 2.2 näher eingegangen wird. Im folgenden sollen jedoch die Forschungsarbeiten auf diesem Gebiet beleuchtet werden.

Die automatische Generierung von „schön aussehenden“ Graphen (Graph Beautifying) ist ein schon länger erforschtes Gebiet. Die dort gewonnenen Kenntnisse lassen sich nun auf das automatische Layouten von UML-Diagrammen anwenden. Ein Ansatz dazu stammt aus China [32], ein weiterer aus Deutschland [11]. Dieses, an der Universität Würzburg entwickelte Framework names SUGIBIB, basiert auf dem Sugiyama-Algorithmus. Dieser 1980 entwickelte Algorithmus war zuerst zur Visualisierung von hierarchischen Datenstrukturen gedacht. Er ließ sich jedoch vorzüglich zur Darstellung von UML-Diagrammen einsetzen. Dazu mußte der ursprünglich für azyklische Graphen gedachte Algorithmus leicht modifiziert werden. Bei dem Layout wurde vor allem auf eine übersichtliche Darstellung wert gelegt. Deshalb versucht der Algorithmus die UML-Assoziationskanten immer durch rechtwinkelige Pfade zu zeichnen. Während die Vererbungskanten direkt von Objekt zu Objekt gehen. Die Assoziationskanten verlassen Objekte, die durch Rechtecke dargestellt werden, immer an der senkrechten Kante, Vererbungskanten an der Waagerechten. Durch diese Richtlinien wird die Darstellung bereits erheblich verbessert. An den verwendeten Algorithmen sieht man jedoch auch deutlich, wie viel Aufwand getrieben werden muß, um

[2] GRaphical Object-Oriented Visualization Environment

ein ansprechendes Layout zu erreichen, welches noch immer nicht unbedingt die Semantik der einzelnen Objekte berücksichtigt. Als Eingabe verarbeitet SUGIBIB das UMLscript-Format, welches ebenfalls an der Uni-Würzburg entwickelt wurde (siehe auch 4.1.2).

Um die Übersichtlichkeit im UML-Diagrammen zu verbessern, wurde in [27] ein Verfahren entwickelt, welches unwichtige Teil des Diagramms bei Bedarf ausblendet. Die ist vor allem bei dem Verständnis von komplexen Softwaresystem sehr hilfreich. Eine ähnliche Idee kommt von der Universität Bremen [17]. Dort wird versucht, UML-Diagramme im dreidimensionalen Raum darzustellen. Unwichtigere Teile können somit in den Hintergrund wandern, wichtige Teile liegen vorne. Gerade mit Hilfe von Animation lassen sich hier viele Sachverhalte anschaulich darstellen. Beispielsweise können Nachrichten in Sequenzdiagrammen durch sich zwischen den Objekten bewegende Bälle dargestellt werden. Die Objekte selbst könnten unterschiedliche Formen und Farben annehmen, um sie besser unterscheiden zu können. Die gerade aktuelle Displaytechnologie ist jedoch noch nicht genug ausgereift und verbreitet, um diese Technik auch bei größeren Diagrammen sinnvoll nutzen zu können. Bei einer stärkeren Verbreitung von 3D-Ausgabegeräten (wie beispielsweise der „Virtual Table“[3] oder die „Cave“[4]) könnte dies jedoch in Zukunft interessant werden.

Die beiden UML-Diagrammtypen Sequenzdiagramm und Kollaborationsdiagramm wurden zur Darstellung von zeitlichen Abläufen entwickelt, um vor allem Methodenaufrufe und ihre Abfolge in den Vordergrund zu stellen. In [23] wurde eine Technik vorgestellt, die mit Hilfe dieser Diagramme Synchronisationsprobleme darstellt. Gerade bei Systemen, die mit Multithreading arbeiten hat man oft mit Problemen wie *andauernden Schlafzustand*[5] und *Verklemmungen*[6] zu kämpfen. Um diese Probleme besser zu erkennen, wurde ein Verfahren entwickelt, welches diese Abfolgebeziehungen graphisch mit Hilfe der beiden Diagramme darstellt. Zur Zeit müssen diese Diagramme noch per Hand erstellt werden, jedoch arbeitet man auch an einer automatischen Generierung, die beispielsweise mit Hilfe eines Debuggers solche Diagramme zur Laufzeit generiert.

UML bietet in der Version 1.3 noch keine Unterstützung zur nähereren Beschreibung von zeitlichen Abläufen, die gerade im Bereich der Echtzeit- und Netzwerksoftware benötigt werden, um Einschränkungen oder Anforderungen zu spezifizieren. Der Artikel [22] schlägt eine Erweiterungen der Semantik von Sequenzdiagrammen vor und erweitert diese um *Timing-Constraints*, also Beschreibungen, die das Zeitverhalten näher spezifizieren. Speziell mit den Anforderungen von Realtime-Systemen beschäftigt sich das Paper von Rambaugh und Selic ([31]). Die Semantik von UML ändert sich leicht, da Realtime-Systeme meist mit Message-Passing statt mit Methodenaufrufen arbeiten. Durch Definition von Ports zur Kommunikation der Komponenten und der Einführung neuer Stereotypen und Constraints läßt sich UML auch zur Modellierung von Realtime-Applikationen einsetzen.

Insgesamt wird hier deutlich, wie offen UML für Erweiterungen ist, und in welchen Bereichen sie sich einsetzen läßt. Einige der erarbeiteten Vorschläge werden Einfluß auf die folgenden Versionen von UML haben, sodaß ein immer größerer Bereich von Software damit modelliert werden kann.

[3] Ein Anzeigegerät, daß auch für 3D-Daten geeignet ist (Rückenprojektionssystem mit Shutterbrille)
[4] Raum, auf dessen Wände Bilddaten projiziert werden, die einen dreidimensionalen Eindruck vermitteln
[5] bezeichnet den Zustand, wenn alle Prozesse schlafen und keiner mehr da ist, einen Prozeß zu wecken
[6] engl. deadlock, der Zustand wenn (mindestens) zwei Prozesse aufeinander warten

2.2 Case-Tools

Seit dem sich UML als Modellierungssprache für Softwareentwickler als Standard etabliert hat, nimmt die Anzahl der Softwareentwicklungswerkzeuge beständig zu. Dieser Teil soll einen Überblick über die derzeit erhältlichen Tools geben. Bei der Evaluation wurde speziell auf die Fähigkeiten zur Simulation oder Visualisierung von Software gelegt. Außerdem wurden die Export- und Importmöglichkeiten untersucht, die es ermöglichen die erstellten Diagramme in anderen Programmen weiter zu verwenden.

2.2.1 Klassifizierung

Softwareentwicklungswerkzeuge zur Modellierung von UML-Diagrammen lassen sich in drei Kategorien einteilen. Die einfachsten Werkzeuge unterstützen nur die Diagrammerstellung, wie man sie von Zeichenprogrammen gewöhnt ist. Hier findet keine Überprüfung der Sinnhaftigkeit des Diagramms stat. In der nächsten Kategorie erkennen die Werkzeuge bereits die gezeichneten Diagramme und können aus ihnen Codegerüste (beispielsweise Interfaces oder Headerdateien) erzeugen. Noch weiter gehen Werkzeuge, die auch aus vorhandenem Code Diagramme generieren können. Teilweise ermöglichen diese Programme auch sogenanntes „Round-Trip“ -Modelling. Sie ermöglichen also paralleles Arbeiten an den Diagrammen und dem Code, wodurch natürlich die größte Flexibilität gegeben ist.

Wenn man nun besonderen Augenmerk auf die Simulationsfähigkeiten legt, trifft man oft auf Programme aus dem „Real-Time-Modelling“ -Bereich. Gerade hier wird oft die Fähigkeit benötigt, Programme möglichst realitätsnah zu simulieren. Da sich diese Arbeit auch mit Simulation und Animation beschäftigt, wurden auch einige Programme aus diesem Umfeld betrachtet.

2.2.2 Import- und Exportformate

Zwar ermöglicht nahezu jedes Tool das Ausdrucken der Diagramme, aber darüber hinaus schwankt die Leistungsfähigkeit doch recht stark. Am unteren Ende der Skala liegt der pixelorientierte Export, meist in GIF und JPEG. Etwas besser wird es dann beim Vektorexport, wo man oft auf Postscript oder das proprietäre WMF trifft. Einige Programme ermöglichen die automatische Generierung von Dokumentation. Sowohl HTML aber auch Microsoft Word ist hier anzutreffen.

Austauschformate, die die Struktur der Diagramme erhalten und somit zum Austausch zwischen verschiedenen UML-Werkzeugen dienen könnten, sind relativ selten anzutreffen. Vor kurzem wurde allerdings ein Standard zum Austausch von Modellen verabschiedet. Viele Hersteller ergänzen nun ihre Programme mit diesem sogenannten XMI-Export. Bisher wurde hier sehr oft das proprietäre MDL-Format von Rational Rose verwendet.

2.2.3 Unterstütze Sprachen und Plattformen

Wie nicht anders zu erwarten, wird C++ und Java von den meisten Werkzeugen unterstützt. Daneben existieren nahezu beliebig viele Speziallösungen die fast jede vorhandene Spra-

che abdecken. Lauffähig sind die meisten Programme unter Windows, manche auch unter Unix. Viele Programme basieren auf Java und sind somit auch für andere Plattformen verfügbar, sofern keine proprietären Erweiterungen benutzt wurden.

2.2.4 Besonderheiten

Um sich ausreichend von der Konkurrenz abzuheben bietet jedes der vorgestellten Programme einige Besonderheiten. Versionskontrolle und Teamfähigkeit sind oft anzutreffen, Reverse-Engineering aus Java-Klassen ist besonders nützlich, wenn kein Sourcecode vorliegt. Die Unterstützung von Design-Pattern mit vorgefertigten Codeteilen ist praktisch für den Entwurf.

Auch die Unterstützung von nicht-UML-Modellierungssprachen (Coad-Yourdon, Booch, OMT) ist zu finden. Obwohl schon seit einigen Monaten UML 1.3 propagiert wird und 2.0 sich im Entwurf befindet, ermöglichen viele Werkzeuge nur die Verwendung von UML 1.1.

2.2.5 Die Tools im Einzelnen

Insgesamt wurden etwa 30 Werkzeuge in die nähere Auswahl gezogen, bei einigen davon war es leider nicht möglich, eine Demoversion zu beziehen, weshalb sie hier nicht weiter berücksichtigt wurden. Die Auswahl reduzierte sich weiter, da einige Tools nicht das gewünschte leisteten und teilweise so fehlerbehaftet waren, sodaß man mit ihnen nicht arbeiten konnte. Übrig blieben 10 Pakete, auf die im folgenden (siehe auch 2.1 näher eingegangen wird.

Produktname	Hersteller	Preis	Exportformate	Simulation
Argo/UML	Tigris	open-source	XMI, Pixel, Vektor	◯
CASE	Elixir	$\leq$400 USD	XML	⊕
MagicDraw Pro	Nomagic	$\leq$400 USD	XMI, Vektor, Pixel	◯
Prosa/om	Prosa	$\geq$600 USD	-	⊕⊕
Rational Rose	Rational	$\geq$600 USD	XMI, MDL	⊕
Rhapsody in C++	I-Logix	$\leq$400 USD	keine	⊕⊕
Tau UML Suite	Telelogic	$\geq$600 USD	Pixel, Vektor	⊕⊕
Together Enterprise	Togethersoft	$\geq$600 USD	XMI, Pixel, Vektor	◯

Tabelle 2.1: Übersicht der getesteten Programme

Argo/UML

Die University of California unternahm als einer der ersten den Versuch, ein vollständiges UML-Tool als Open-Source zu entwickeln. Die aktuelle Version bietet nahezu alle UML-Diagrammtypen und generiert auch Sourcecode. Besonders auffällig sind die Ideen dem Benutzer Hinweise für einen möglichst guten Entwurf zu geben. Außerdem wurden typische Probleme der Softwareentwicklung berücksichtigt, beispielsweise eine Todo-Liste und Prioritäten. XMI wird direkt als Dateiformat verwendet, die graphische Repräsentation findet zur Zeit mit PGML statt, SVG ist aber als Ersatz dafür geplant. Zur Zeit ist Argo/UML als Designstudie eines Modellierungswerkzeuges zu verstehen, es sind viele interessante Ansätze enthalten, jedoch für den produktiven Einsatz ist es noch nicht geeignet.

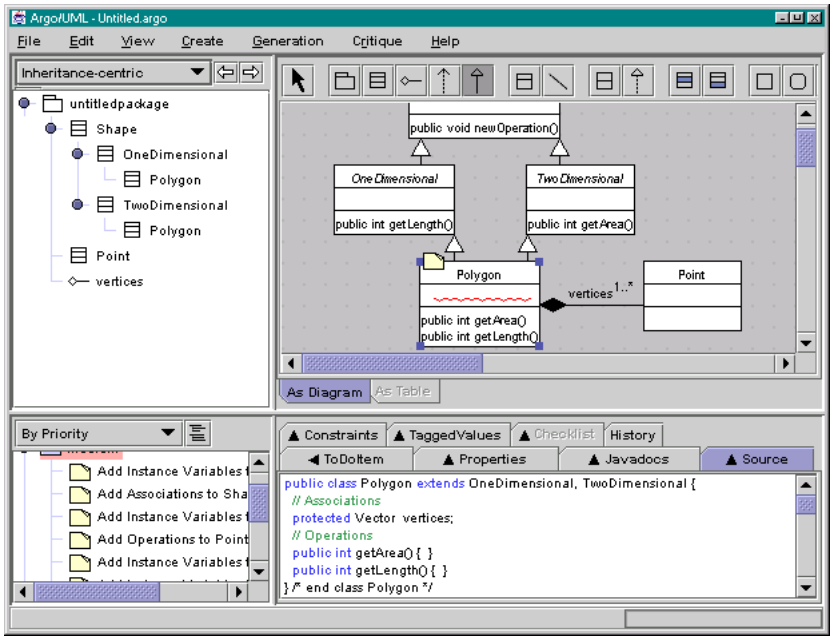

Abbildung 2.1: Argo/UML

CASE

Der Hersteller Elixir hat sich sehr viel Mühe gegeben, um eine automatische Erstellung von Sequenzdiagrammen zu ermöglichen. Mit der Maus kann man hier recht einfach ein Sequenzdiagramm zusammenklicken, indem man die einzelnen Methodenaufrufe anwählt. Die Exportfähigkeiten sind relativ eingeschränkt, ein XML-Export ist zwar vorhanden, orientiert sich aber an einer eigenen DTD und nicht XMI.

MagicDraw Pro

Auch MagicDraw verwendet XMI als Dateiformat, die Liste der weiteren Exportformate ist ebenfalls sehr umfangreich. Die Layoutwerkzeuge sind hier besonders gut gelungen, vor allem die Arbeitsgeschwindigkeit liegt über dem der Konkurrenzprodukte. Angenehm fällt auch der sehr günstige Preis auf. Es sind zwar keine herausragenden Features enthalten, jedoch sind die vorhandenen Werkzeug gut umgesetzt. Der Hersteller NoMagic war einer der ersten, der Reengineering von Java-Klassen in ein Modellierungswerkzeug integrierte, diese Funktionalität ist deshalb in MagicDraw schon recht ausgereift.

Prosa/om

Obwohl von Prosa keine Demoversion erhältlich war, sollte dieses Produkt erwähnt werden, weil es die Möglichkeit des Tracking von UML-Diagrammen bietet. Darunter versteht der Hersteller eine Funktion, mit der man eine Aufrufreihenfolge von Methoden vorgeben kann, welche dann durch eine Animation in einigen Diagrammtypen dargestellt wird. Da der Hersteller selbst nach mehrfachen Anfragen keine Demoversion zur Verfügung stellen konnte, war es jedoch nicht möglich dieser Funktion näher zu untersuchen.

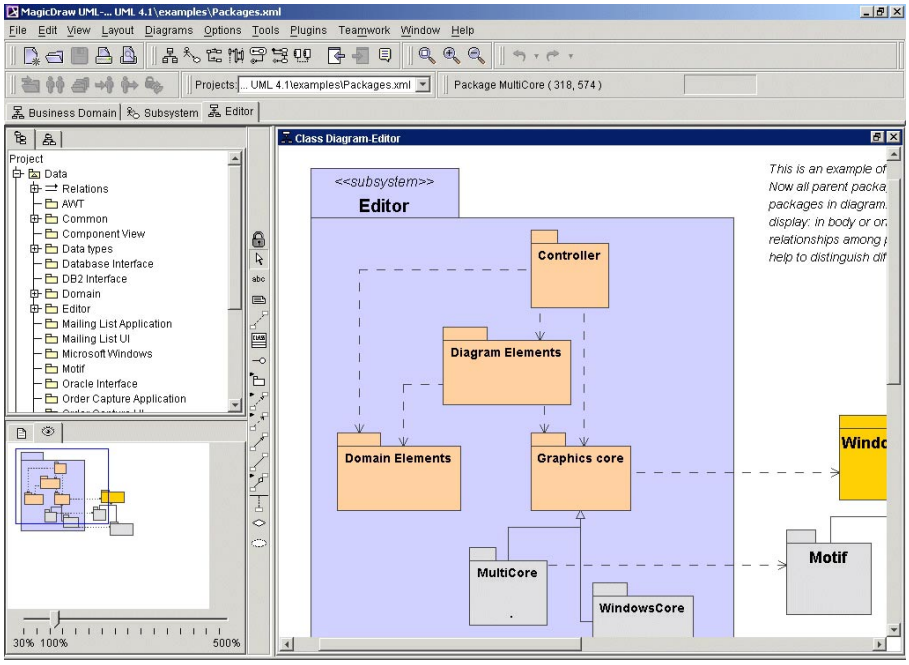

Abbildung 2.2: Magic Draw Pro

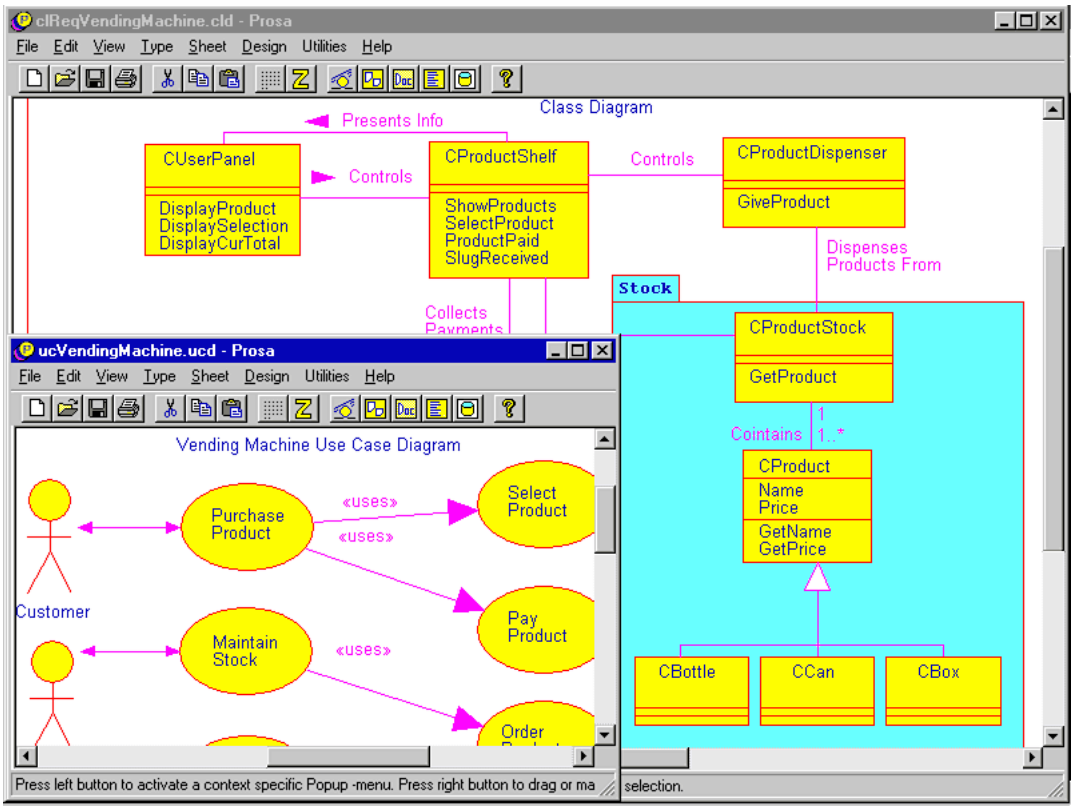

Abbildung 2.3: Prosa/om

Rational Rose

Rational Rose ist sicherlich eines der bekanntesten Modellierungswerkzeuge. Dies ist nicht weiter verwunderlich, sind doch die Entwickler auch die „Erfinder“ von UML. Rose ist ein sehr vielfältiges Werkzeug, welches versucht möglichst viele Anwendungsbereiche abzudecken. Beispielsweise ist eine Realtime-Variante verfügbar, die auch Diagrammanimation unterstützt. Die Bedienung ist an manchen Stellen nicht sehr intuitiv gelöst, einige Funktionen arbeiten nicht fehlerfrei. XMI-Export ist vorhanden.

Rhapsody in C++

Auch Rhapsody unterstützt die automatische Generierung von Sequenzdiagrammen, jedoch bietet es im Bereich der Simulation noch wesentlich mehr. Besonders erwähnenswert ist hier die Möglichkeit der Simulation mit Benutzungsoberfläche. Der Entwickler entwirft eine Prototyp, der mit Rhapsody dann simuliert wird, dies schließt eine Benutzungsoberfläche ein, d.h. man kann direkt verfolgen, welchen Effekt Eingaben auf die internen Abläufe haben. Außerdem existieren mehrere Verifikationsmöglichkeiten, die es vereinfachen den Code mit dem Modell in Einklang zu bringen.

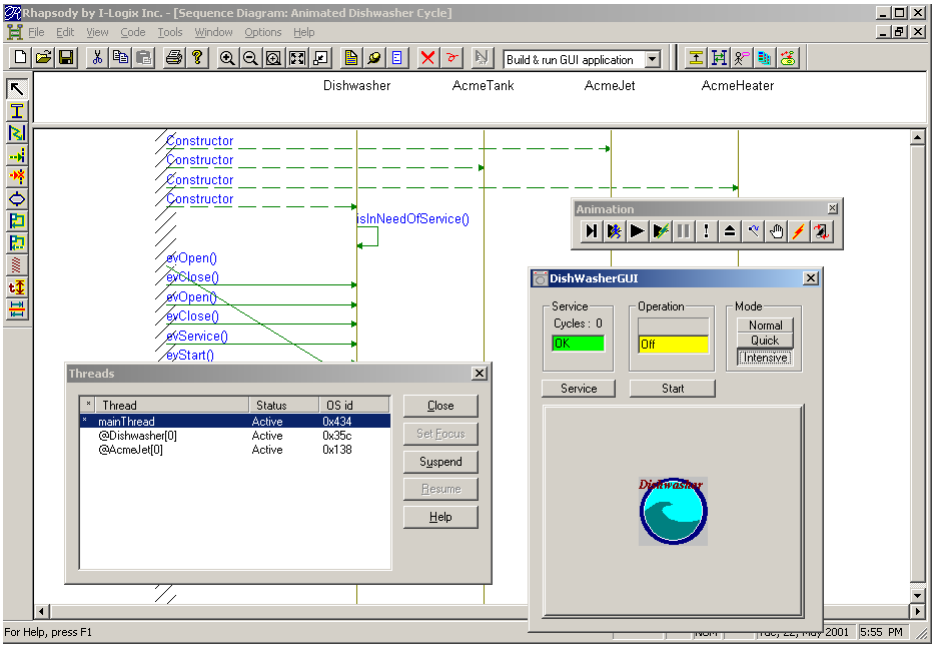

Abbildung 2.4: Rhapsody in C++

Tau UML Suite

Telelogics Produktreihe orientiert sich ebenfalls stark an den Bedürfnissen der Echzeitenwicklungen. Bereits seit einigen Jahren wird in diesem Gebiet als Austauschformat SDL genutzt. Mit Tau UML können UML-Modelle in SDL umgewandelt und dann simuliert werden. Ein XMI-Export ist ebenfalls vorhanden. Leider ist die Bedienung sehr unübersichtlich, was wohl auch an dem absolut unüberschaubaren Funktionsumfang liegt.

Together Enterprise

Together macht von den getesteten Programmen den ausgereiftesten Eindruck. Obwohl es in Java geschrieben ist, läßt sich damit flüssig arbeiten. Die Roundtrip-Funktionalität kann auch Java-Klassen reengineeren, eine Pattern-Bibliothek vereinfacht den Entwurf. Ebenfalls integriert wurde die automatische Generierung von Sequenzdiagrammen, die ihre Information aus dem Sourcecode bezieht. Sogar die Erzeugung von Codefragmenten aus Sequenzdiagrammen wurde implementiert, jedoch ist verständlich, daß diese beiden Funktionen nur für einfache Abläufe eingesetzt werden können.

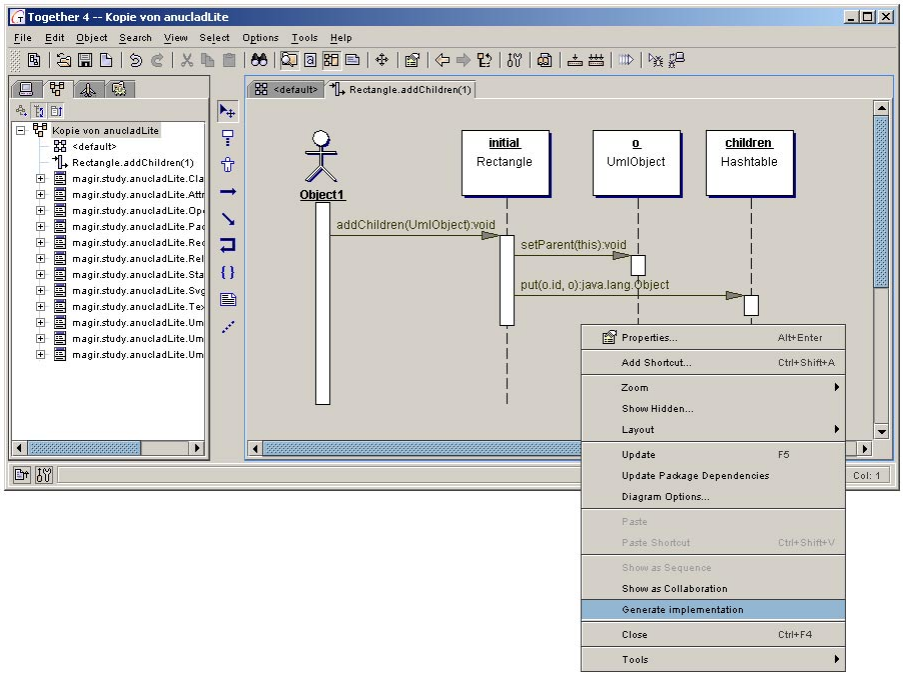

Abbildung 2.5: Together Enterprise

2.2.6 Zusammenfassung

Es ist schwer, die derzeit verfügbaren Programme einzuschätzen, da sich teilweise wöchentlich der Funktionsumfang verändert. Dennoch ist festzustellen, daß der Trend zu einer vollständigeren Implementierung des UML-Standards vorhanden ist. Ebenfalls zu begrüßen ist die immer weiter um sich greifende Unterstützung von XMI. Dadurch wird es bald möglich sein, ohne Datenverlust komplette Modelle zwischen verschiedenen Werkzeugen auszutauschen, was einerseits die Zusammenarbeit im Team vereinfacht und anderseits die Verwendung anderer Werkzeuge ermöglicht.

Doch nicht nur der XMI-Import/Export wird immer wichtiger; auch um eine vernünftige Dokumentation zu erreichen ist es notwendig, andere Dateiformate behandeln zu können. Durch die starke Verbreitung von Webbrowsern wird sich hier auf Dauer HTML oder eines der Nachfolger davon durchsetzen. Auch PDF ist hier ein Kandidat, da inzwischen eine weite Verbreitung des Adobe Acrobat Readers gewährleistet ist. Für die einzelnen Diagramme wird ein plattformübergreifendes Vektorformat benötigt, neben Postscript ist hier auch der neue W3C-Standard SVG geeignet.

Ein Großteil der Programme bieten nur Unterstützung für statische Diagramme. Simulation oder Animation von Diagrammen ist meist den Programmen aus dem Echtzeit-Umfeld vorbehalten. Hier werden vor allem die Sequenzdiagramme und Kollaborationsdiagramme verwendet. Hier ist noch der deutlich zu merken, daß diese Programme nachträglich an UML angepaßt wurden. Sie wurden vorher primär zur Simulation eingesetzt und verwendeten dabei proprietäre Darstellungstechniken. Zustandsdiagramme sind zwar auch inzwischen Bestandteil der UML, jedoch sind hier auch die Entwickler der UML gefordert, die Spezifikation weiter voranzutreiben und auch die Randbereiche der Softwareentwicklung wie der Echtzeitbereich oder auch der Embedded-Bereich ausreichend zu unterstützen.

Kapitel 3

Eine Visualisierungstechnik zur Darstellung von Applikationslogik

3.1 Darstellung von Klassendiagrammen

Wie das vorherige Kapitel gezeigt hat, ist das Gebiet der Softwarevisualisierung noch lange nicht erschöpft. Die Unified Modelling Language ermöglicht neue Ansätze auf dem Gebiet der Visualisierung objektorientierter Software. Als sehr mächtige und vor allem offene Sprache bietet sie viele Erweiterungsmöglichkeiten. Durch ihre Akzeptanz in Firmen ist außerdem eine große Anwenderschaft vorhanden, was bei propräteren Lösungen im Allgemeinen nicht gegeben ist.

Die Simulation von Software findet meist im Bereich der Modellierung von Echzeitapplikationen und in der Prozeßmodellierung anwendung. Werkzeuge zur Simulation von Objektorientierter Software sind noch nicht sehr stark verbreitet, die meisten Ansätze befinden sich noch im Forschungsstadium. Oft werden Debugger mit graphischen Ausgabemöglichkeiten angereichert, dies ermöglicht jedoch keinen Einsatz bei der Modellierung von Software.

Gerade aber bei dem Entwurf der Software sollte man bereits Abläufe innerhalb der Applikationslogik untersuchen, um Fehler zu finden und Mißverständnissen vorzubeugen. Prototypen sind ein oft eingesetztes Mittel, welche jedoch im Allgemeinen nur die Oberfläche darstellen, wodurch innere Abläufe verborgen bleiben. In der Simulation lassen sich meist auch interne Zustände darstellen, die bisherigen Anwendungen sind jedoch - wie bereits erwähnt - auf den Automations- und Prozeßtechnikbereich zugeschnitten.

Es gibt viele Möglichkeiten Simulation von Software durchzuführen. Debugger zeigen meist den Sourcecode und die Belegung von Variablen an, Realtimewerkzeuge verwenden oft Zustandsgraphen. Meist arbeiten diese Systeme auf bereits funktionierendem Programmcode. Dies hat mehrere Nachteile, beispielsweise muß der Programmcode bereits fertiggestellt sein, somit sind diese Hilfsmittel nur bedingt für den Entwurf geeignet. Da sie ihre Information meist nur aus dem Code beziehen können sie nur unzureichend komplexe Zusammenhänge darstellen. Eine vom Entwickler erweiterte Darstellung mit Kommentaren und angepassten Abläufen dürfte somit das Verständnis der Abläufe verbessern. Zudem kann sie unabhängig vom Code entstehen, wodurch sie auch für die Entwurfsphase geeignet ist.

Zum Entwurf von Software wird im zunehmenden Maße die Unified Modelling Language (UML) eingesetzt, und es ist zu erwarten, daß sie sich in den kommenden Jahren als Standard durchsetzen wird. Da die Semantik der UML-Diagramme sehr genau definiert wurde, kommt es seltener zu Mißverständnissen zwischen Auftraggebern und Programmieren. Je größer jedoch die zu entwickelnden Systeme sind, um so umfangreicher sind auch die erstellten Diagramme. Sie werden dadurch schnell unübersichtlich und gerade für Nicht-Programmierer schwer verständlich. Die Aufteilung in mehrere kleinere Diagramme gelingt nicht immer. So ist es nicht verwunderlich, wenn die erstellten Diagramme meist ausgedruckt werden, weil sie auf dem Bildschirm nicht mehr übersichtlich dargestellt werden können. Die wichtigsten Diagramme innerhalb der UML sind sicherlich die Klassendiagramme und zur Darstellung von Abläufen die Sequenz- und Kollaborationsdiagramme.

Die im Abschnitt 2.2 vorgestellten Softwarewerkzeuge verwenden meist Sequenz- und Kollaborationsdiagramme, um zeitliches Verhalten und Applikationslogik darzustellen. Einige der Werkzeuge können diese auch animieren, wobei diese Animation meist nur aus dem zeitverzögerten Hinzeichnen der Diagrammelemente besteht.

Ein Grundgedanke der UML ist durch viele "Sichten" auf ein System ein möglichst vollständig Bild zu erhalten. Ein Teil der Diagramme beschäftigt sich mit dem "Requirements capturing", welches am Anfang eines Projektes besonders wichtig ist. Der Großteil der Diagramme verwendet jedoch Objekte, die sich späteren Sourcecode durch Klassen, Interfaces und Templates realisiert werden. Während diese Objekte in mehreren Diagrammen vorkommen, so sind ihre Attribute, Operationen und Relationen zueinander nur im Klassendiagramm dargestellt. Dies führt dazu, daß man oft auf das Klassendiagramm zurückgreifen muß, um andere Diagramme zu verstehen. Somit nimmt dieser Diagrammtyp ein zentrales Element bei dem Entwurf und der Dokumentation ein.

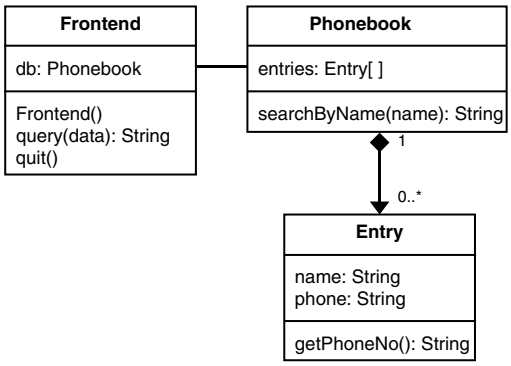

Abbildung 3.1: Beispiel eines Klassendiagramms

UML-Klassendiagramme scheinen auf den ersten Blick ungeeignet zur Simulation von Applikationslogik. Sie stellen meist zu viele Informationen dar, als zum Verständnis von einzelnen Abläufen notwendig sind. Dieses Problem läßt sich mit Hilfe von Animation und dynamisch verändernden Diagrammen mildern. Beispielsweise könnte man inaktive Objekte ausblenden oder dimmen. Durch animiertes Umgruppieren könnte man andere Zusammenhänge zwischen Objekten darstellen. Einblendeffekte könnten das Diagramm stufenweise aufbauen, damit der Betrachter nicht sofort mit dem ganzen Diagram „erschlagen" wird.

Gerade die letzte Variante findet in der Praxis schon in abgewandelter Form ihre Anwendung: Sehr oft werden UML-Diagramme skizzenartig auf Papier oder Whiteboards von Hand aufgemalt und dabei erklärt. Wenn man diesen Vorgang mit Software nachbilden

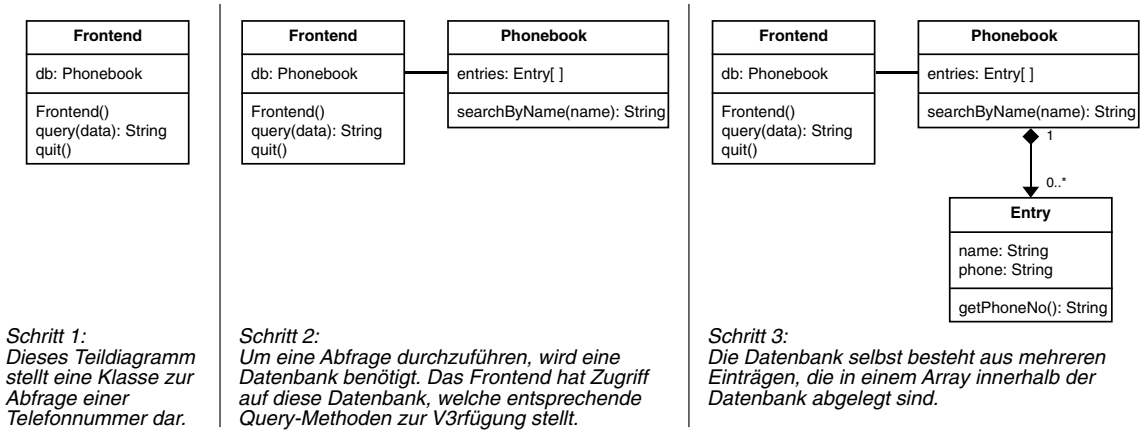

Abbildung 3.2: Klassendiagramm, stückweise dargestellt

könnte, wäre dies somit eine gute Ergänzung zur Dokumentation eines Softwarepaketes.

3.2 Animation von Klassendiagrammen

Die bisher genannten Animationstechniken erweitern noch nicht die Semantik der Klassendiagramme. Klassendiagramme beinhalten Methoden und Attribute der Klassen. Dadurch wäre die Möglichkeit gegeben, auch Methodenaufrufe in ihnen darzustellen. So ließe sich die Semantik der Sequenz- und Kollaborationsdiagramme in das Klassendiagramm integrieren. Wieder hilft uns hier Animation weiter. Die einzelnen Aufrufe könnten in ihrer zeitlichen Abfolge dargestellt werden, zusätzliche Kommentare erklären diese Aufrufe. Ein wichtiges Ausdrucksmittel ist auch der sogenannte Kontrollfokus, der anzeigt, welche Objekte gerade aktiv sind. Dies ließe sich ebenfalls mit Animation darstellen.

Auf diese Weise lassen sich Abläufe aus Sequenzdiagrammen in Klassendiagrammen integrieren. Als weiterer Schritt wäre auch eine zeitnahe Simulation eines Programmes möglich. Im Unterschied zu den vorgefertigten Sequenzdiagrammen ermöglicht die Simulation eine gewisse Interaktion mit der simulierten Software. Dazu könnte man Benutzereingaben an einem Prototyp der Oberfläche durchführen und diese im Klassendiagramm darstellen. Eine weitere Anwendung wäre das Debugging. Hierzu könnte man beispielsweise den Sourcecode mit Animationsbefehlen instrumentieren, die das Klassendiagramm entsprechend der gerade stattfindenden Abläufe verändert.

Die im vorherigen Abschnitt entwickelte Idee soll nun näher beleuchtet werden. Ziel ist ein Werkzeug zur Darstellung und Animation von UML-Klassendiagrammen. Wie bereits erwähnt sollte dieses Werkzeug sowohl für die Entwurfs- und Implementationsphase als auch zur nachträglichen Dokumentation dienen. Da eine Erweiterung bereits bestehender Werkzeuge sehr schwierig ist, soll ein separates Werkzeug entstehen. Dieses könnte die Darstellung und Animation auch in Webbrowsern ermöglichen und wäre somit gut in bestehende Dokumentationen und Arbeitsabläufe integrierbar.

Daraus ergibt sich beispielsweise folgende Anwendungstechnik: Zuerst wird der Entwurf des Klassendiagramms und eventuell einiger Sequenzdiagramme mit einem Modellierungswerkzeug durchgeführt. Diese Information wird in einem Austauschformat abgelegt und von einem Konvertierwerkzeug in ein animierbares Diagramm konvertiert. Dieses

Diagramm könnte beispielsweise zur Dokumentation hinzugefügt werden, somit benötigt man zur Darstellung der Diagrammen nicht mehr das Modellierungswerkzeug. Ein weiteres Werkzeug kümmert sich um die Animation, dazu werden idealerweise die Informationen der Sequenzdiagramme oder Programmtraces verwendet.

Eine Vielzahl der in Abschnitt 2.2 vorgestellten Programme bieten bereits einen Export der Diagramme an, jedoch ist dieser oft nur Bitmaporientiert, was einige Nachteile in sich birgt. Bitmaps lassen sich im allgemeinen nur mit Qualitätsverlusten skalieren, außerdem kann textuelle Information nicht mehr nachträglich verändert werden. Regruppierung der Elemente ist genausowenig möglich wie selektives Ausblenden. Einige der Werkzeuge bieten zusätzlich vektororientierten Export an, jedoch meist nur in proprieäteren Formaten. Somit muß ein eigenes Werkzeug entwickelt werden, welches ein vektorbasierendes Diagramm erzeugt. Viele Modellierungswerkzeuge bieten hierzu bereits Exportmöglichkeiten, die neben der grafischen Darstellung auch die Metainformationen zur semantische Bedeutung exportieren. In Abschnitt 4.5 auf Seite 34 werden diese näher beleuchtet. Dort findet sich auch ein Überblick über mögliche Vektorformate zur Darstellung.

Durch die UML-Spezifikation [25] ist das Aussehen und die Semantik der Klassendiagramme sehr genau bestimmt. Dies hat vor allem den Zweck, Mißverständnissen in der Entwurfsphase vorzubeugen. Ein Konvertierwerkzeug sollte sich möglichst genau an diese Spezifikation halten. Inzwischen ist UML bei der Version 1.3 angelangt. Entsprechend umfangreich sind die verschiedenen Ausdrucksmittel. Da jedoch ein Großteil dieser Erweiterungen nur sehr selten benötigt werden, beschränkt sich das Konvertierwerkzeug auf eine Teilmenge der Spezifikation.

Bereits im Modellierungswerkzeug wird beim Entwurf einer Software versucht eine sinnvolle Anordnung der Klassen zu finden. Die dabei zu berücksichtigen Aspekte sind dabei sehr vielfältig, sodaß eine automatische Anordnung nur als erste Näherung dienen kann. Dennoch ist dieser Bereich Ausgang reger Forschung; Arbeiten wie [11] wurden bereits in Kapitel 2 erwähnt. Die Anordnung der einzelnen Elemente sollte jedoch bereits bei dem Entwurf von dem Modellierer festgelegt werden. Eine automatische Anordnung sollte deshalb toolunterstützt ablaufen, um manuelles nacharbeiten zu ermöglichen. Die dabei entstehenden Diagramme sollten mit diesen Geometrieinformationen exportiert werden, um von dem Konvertierungswerkzeug umgesetzt werden zu können.

Ein weiteres Werkzeug dient dann zur Animation dieser Diagramme. Angelehnt an UML-Sequenzdiagramme identifiziert man folgende Nachrichtentypen:

Erstellung von Instanzen Durch Konstruktoren werden Objekte instanziiert. Graphisch wird dies in Sequenzdiagrammen durch einen Pfeil auf das Objekt dargestellt. Unterhalb des Objektes beginnt dann die sogenannte Lebenslinie.

Zerstörung von Instanzen Destruktoren werden aufgerufen, wenn einen Instanz nicht mehr benötigt wird. In Sequenzdiagrammen wird dies durch ein „X“ am Ende der Lebenslinie dargestellt.

Methodenaufrufe Das wichtigste Ausdrucksmittel in Sequenzdiagrammen sind die Pfeile, die Methodenaufrufe darstellen. Sie können sowohl zu anderen Instanzen als auch zur eigenen Instanz (Selbstdelegation) zeigen. Dargestellt werden sie durch einen Pfeil mit gefüllter Spitze.

Rückgabe Um Rückgaben von Methodenaufrufen darzustellen werden gestrichelte Pfeile verwendet.

Rekursive Aufrufe Als erweiterte Form der Selbstdelegation sind rekursive Aufrufe denkbar. Sie werden durch leicht versetzte Lebenslinien dargestellt.

Kontrollfokus In komplizierteren Diagrammen kann man das gerade aktive Objekt durch eine schmales Rechteck, welches auf der Lebenslinie liegt, darstellen. Bei Multithreading, können auch mehrere Kontrollfoki gleichzeitig existieren.

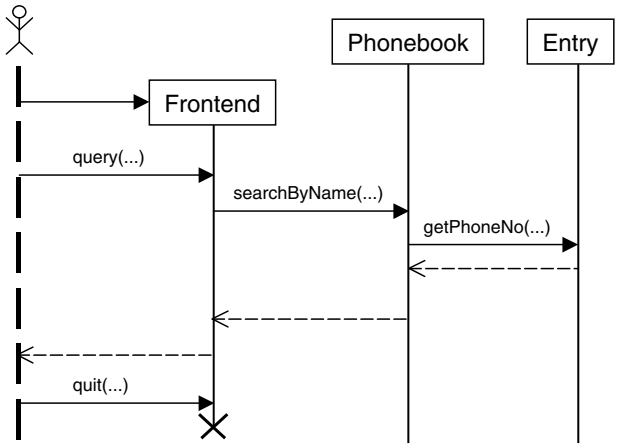

Abbildung 3.3: Beispiel eines Sequenzdiagramms

Die Zeitachse verläuft bei Sequenzdiagrammen im Allgemeinen von oben nach unten. Durch die Anordnung der Pfeile ist also die Abfolge der Nachrichten genau festgelegt. Die Pfeile selbst sind mit dem Nachrichtentyp markiert, der im allgemeinen dem Methodennamen entspricht. Die Diagramme können durch textuelle Hinweise ergänzt werden. Diese beschreiben beispielsweise Schleifen, genauere zeitliche Abfolgen oder allgemeinere Bedingungen.

Neben den Sequenzdiagrammen bietet UML noch die *Kollaborationsdiagramme*. Diese lassen eine freiere Anordnung der Objekte zu, die Reihenfolge der Nachrichten wird hier durch sogenannte Sequenznummern festgelegt. Semantisch sind diese Diagramme prinzipiell äquivalent zu Sequenzdiagrammen, wenn sie auch einigen Einschränkungen unterliegen. Je nach Anwendungsfall bietet der eine oder andere Diagrammtyp einen besseren Überblick. Kollaborationsdiagramme bieten zwar eine freiere Anordnung der Objekte, werden jedoch sehr schnell unübersichtlich, wenn Objekte mehrfach benutzt werden. Parallele Abläufe sind nur noch sehr schwer darstellbar.

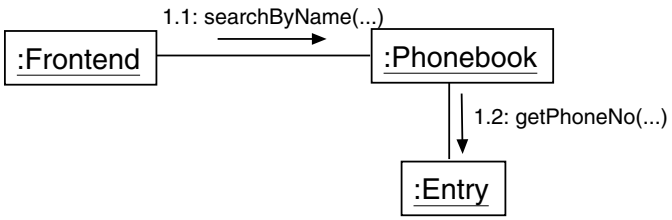

Abbildung 3.4: Beispiel eines Kollaborationsdiagramms

Nicht unerwähnt sollen die *Aktivitätsdiagramme* der UML bleiben. Sie ähneln den hinreichend bekannten Flußdiagrammen und bieten gute Darstellungsmöglichkeiten von Synchronisation, jedoch sind sie abstrakter Natur und arbeiten nicht direkt mit den Klassen-

objekten zusammen. Sie dienen daher der frühere Entwurfsphase, um Eigenschaften der zu entwickelnden Software festzulegen.

Unter Beibehaltung der Semantik sollen die in Sequenzdiagrammen verwendeten Pfeile nun zur Darstellung von Abläufen in Klassendiagrammen verwendet werden. Da man durch Animation nicht mehr an die Zeitachse gebunden ist, sondern eine Sequenz direkt in ihrem Ablauf darstellen kann, bietet sich die Verwendung des Klassendiagramms an. Diese Darstellungstechnik ähnelt den Kollaborationsdiagrammen, durch die Verwendung des Klassendiagramms und der Animation kann jedoch mehr Information untergebracht werden, die dennoch übersichtlicher dargestellt wird.

Bevor eine Animation dargestellt wird, sollten die Objektrelationen (Assoziationen und Vererbungen) etwas abgeblendet werden. Damit wird vermieden, daß zu viele Pfeile in dem Diagramm sichtbar sind, außerdem kommt es so nicht zu Verwechslungen mit den Aufrufpfeilen. Die eigentliche Animation kann beispielsweise wie folgt aussehen: Zuerst wird das Objekt hervorgehoben, welches gerade den Kontrollfokus hat. Die gerade aktive Methode wird hervorgehoben (beispielsweise durch farbige Hinterlegung). Ein Pfeil wird eingeblendet, der von dieser Methode zur aufrufenden Methode zeigt. Nach Beendigung des Aufrufs wird die Pfeilspitze an die andere Seite gesetzt, der Aufruf kehrt zurück. Zusätzlich sollte bei Rückkehrpfeilen unterschieden werden, ob es sich nur um die Rückkehr des Kontrollflußes oder auch um die Rückgabe von Werten handelt.

Um die Objekterstellung und -zerstörung besser darstellen zu können, bieten sich mehrere Möglichkeiten an. Eine Farbkodierung könnte beispielsweise die drei Zustände (statisch, instanziiert, zerstört) darstellen, jedoch kann die Farbgebung für andere Zwecke, wie logische Gruppierung besser verwendet werden. Man könnte alternativ Objekte erst einblenden, wenn sie instanziiert werden, dies zerstört jedoch den Überblick im Diagramm und ermöglicht es nicht, Aufrufe statischer Methoden darzustellen. Somit bietet sich eine nicht vollständige Ausblendung der Objekte an. Solange sie nur statisch vorhanden sind, ist ihr Hintergrund und ihre Instanzmethoden und -attribute gedimmt. Wird ein Objekt instanziiert wird es eingeblendet, bei Zerstörung wieder heruntergedimmt.

Objekte können mehrfach instanziiert werden, dies läßt sich sehr gut durch Hinterlegen eines verschobenen Rechtecks darstellen (siehe Abb. 3.5). Um die Reihenfolge der Nachrichten besser mitverfolgen zu können sollte das gerade aktive Objekt gesondert hervorgehoben werden, was beispielsweise durch einen verstärkten Rand geschehen könnte. Bei Verwendung von Multithreading können so auch mehrere, gleichzeitig aktive Objekte dargestellt werden.

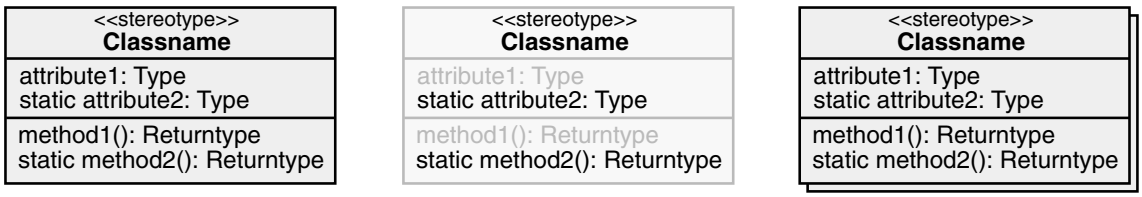

Abbildung 3.5: Darstellungsmöglichkeiten eines Klassifizierers: Instanziiert, nur statisch, mehrfach instanziiert

In Sequenz- und Kollaborationsdiagrammen sind Methoden und Attribute nicht direkt sichtbar, deshalb werden hier die Methodennamen an den entsprechenden Aufrufpfeil angehängt. In Klassendiagramme sind die Methodennamen bereits enthalten, deshalb bietet es sich an, diese in die Visualisierung einzubinden. Ähnlich wie bei dem Kontrollfo-

kus bietet sich hier eine Umrahmung oder farbliche Hinterlegung an. Um das Diagramm übersichtlich zu halten, sollte immer nur der gerade aktive Aufrufpfeil sichtbar sein. Denkbar wäre es, die ebenfalls zur aktuellen Aufrufkette gehörenden Aufrufpfeile abgeblendet darzustellen. Dadurch würde die gesammte Aufrufkette sichtbar bleiben, was Momentaufnahmen des animierten Klassendiagramms ermöglichen würde. Zur Illustration sind in Abbildung 3.6 die ersten vier Schritte einer solchen Animation dargestellt.

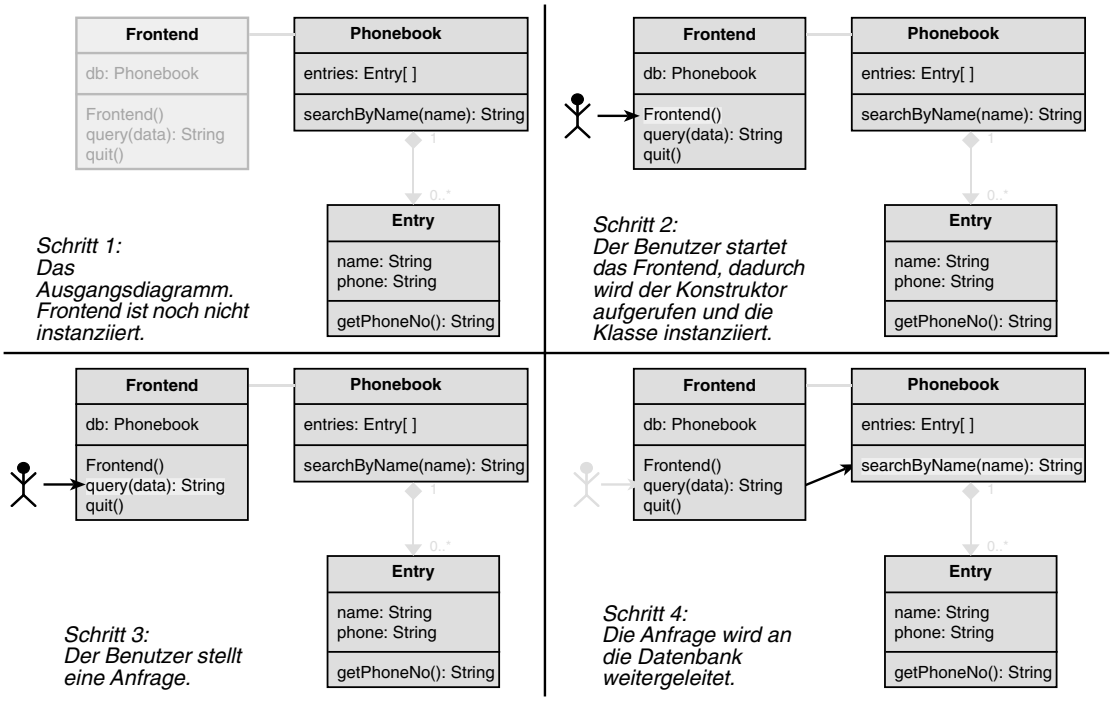

Abbildung 3.6: Vier Schritte einer Animation zur Darstellung von Applikationslogik

Sequenzdiagramme werden meist mit Notizen angereichert, die die einzelnen Schritten näher erläutern. Dies sollte in animierten Klassendiagrammen ebenfalls möglich sein. Hier stehen wieder mehrere Möglichkeiten zur Verfügung. Ähnlich wie bei Sequenz- und Klassendiagrammen könnte man kleine Notizzettel einblenden, alternativ bietet sich eine Console an, die weitere Ausgaben, wie Zeitstempel und andere Ablaufrelevante Informationen liefern könnte. Denkbar ist auch eine Sprachausgabe, die zu den gerade gezeigten Abläufen Erläuterungen gibt.

Komplexere Abläufe lassen sich meist nicht durch ein einfaches Sequenzdiagramm beschreiben, zu oft hängt der weitere Weg von verschiedenen Parametern ab. In Sequenzdiagrammen gibt es die Möglichkeit, Abläufe an gewisse Bedingungen zu koppeln, jedoch wird dadurch das Diagram meist zu kompliziert, weshalb hier oft stattdessen mehrere Diagramme gezeichnet werden. In animierten Diagrammen wäre es nun möglich, die einzelnen Alternativen zur Auswahl anzubieten. Der Betrachter wählt eine Variante aus, und läßt sich so einen Zweig des Ablaufes darstellen.

Diese gerade erwähnten Möglichkeiten der Interaktion lassen bereits erkennen, wie man das Konzept der Animation mit Interaktion erweitern kann. Einfache Beispiele wären das Anhalten der Animation sowie das Vor- und Zurückspulen.

3.3 Problemquellen und Schwächen des Konzepts

Leider hat das beschriebene Konzept auch Schwächen. UML-Klassendiagramme erreichen sehr schnell eine unhandliche Größe. UML bietet die Möglichkeit, Objekte in Pakete zu gruppieren Diese Pakete werden meist in unterschiedlichen Diagrammen dargestellt, was natürlich die Darstellung paketübergreifender Kommunikation erschwert.

Auch paketinterne Darstellung kann unübersichtlich werden, wenn zusammengehörige Klassen graphisch zu weit auseinanderliegen. In [27] wurde ein Verfahren vorgestellt, welches das Diagramm auf seine relevanten Teile beschränkt und dafür eine etwas abstraktere Darstellung in Kauf nimmt. Dieser Ansatz ist hier direkt nicht möglich, jedoch sind die dort entwickelten Verfahren auch bei *ANUCLAD* anwendbar. Beispielsweise könnte die Anordnung der Klassen aufgebrochen werden und die kommunizierenden Elemente könnten näher zueinander gerückt werden. Um dennoch dem Benutzer einen gewissen Überblick zu bieten, wäre es möglich die Umordnung und Ausblendung nicht schlagartig vorzunehmen, sondern die einzelnen Elemente langsam auszublenden oder an die neue Stelle zu bewegen.

Bei der Verwendung marktüblicher Anzeigegeräte (Monitore und Flachbildschirme bis 19 Zoll Bilddiagonale) ist es selten möglich, ein UML-Diagramm vollständig zu zeigen. Somit wäre es wünschenswert, wenn der gerade sichtbare Bildausschnitt sich zu der gerade aufgerufenen Methode bewegt. Dies könnte durch Vergrößerung und Verkleinerung des Ausschnittes unterstützt werden.

Ein größeres Problem ergibt sich aus der Vererbung und Polymorphie, die in objektorientierten Systemen oft anzutreffen ist. Wenn beispielsweise ein Objekt eine Methode von einer ihrer Superklassen ererbt, dann wird dies nicht unbedingt im Klassendiagramm dargestellt[1]. Um diese Vererbungsstruktur deutlich zu machen, bietet es sich an, den Aufruf an die entsprechende Unterklasse umzuleiten.

Die Problematik wird noch offensichtlicher, wenn man Polymorphie betrachtet. Hier kann sich das Verhalten eines Objektes ändern, wenn es auf einen anderen Objekttyp „gecastet“ wird. Darstellungstechnisch ergibt sich das Problem, daß nun vielleicht ein ganz anderes Objekt aktiv wird, welches bisher noch als inaktiv und statisch dargestellt wurde. Hier wird sehr gut deutlich, daß eine klare Trennung von Objekten und Instanzen nötig ist. Dies wird im Allgemeinen von Klassendiagrammen nicht geleistet und bisher ist auch noch kein sinnvolles Verfahren zur Darstellung dieser Dualität entwickelt worden. Es wäre zwar denkbar, diese beiden Sichtweisen klar zu trennen und sowohl ein Objektdiagramm, als auch ein Instanzdiagramm darzustellen, dies würde jedoch dem Konzept der einfachen, übersichtlichen Darstellung zuwiderlaufen.

[1] UML bietet hierzu die Möglichkeit, vor Attributen und Methoden ein „/“ zu setzen, was anzeigt, daß diese Eigenschaft ererbt wurde.

Kapitel 4

Auswahl der Technologien

4.1 Austausch von UML-Modelldaten

UML versteht sich zwar als Modellierungssprache, jedoch drückt sich diese Sprache meist nur in Diagrammen aus. Dadurch ist der Austausch der Softwaremodelle nur eingeschränkt möglich. Ein allgemein anerkannten Standard zum Austausch dieser Modelldaten ist gerade erst verabschiedet worden und ist folglich erst bei einer kleinen Anzahl von Entwicklungswerkzeugen vorhanden.

4.1.1 Rational MDL

Rational Rose war eines der ersten UML-Modellierungswerkzeuge auf dem Markt. So ist es nicht weiter verwunderlich, daß das Dateiformat von Rose sich als Quasi-Standard etabliert hat. Dieses MDL-Format ist leider sehr spezifisch auf das Rational-Produkt abgestimmt.

4.1.2 UMLscript

In 2.1.2 wurde bereits das Format UMLscript [30] erwähnt. Dieses textbasierte Format wurde anhand einer attributierten Grammatik definiert, aus dieser wurde ein Parser für Java entwickelt. Das Format selbst versucht UML 1.3 möglichst nahe abzubilden. Leider wurde neben der Semantik keine weitere Information über das Aussehen der Diagramme mit eingeflochten.

```
DIAGRAM
   CLASS CallQueue
   ATTRIBUTES queue:List, source, capacity
END DIAGRAM
```

Abbildung 4.1: Beispiel: UMLscript

Das Problem von UMLscript ist die eigene Grammatik, für die ein eigener Parser geschrie-

ben werden müßte. Wesentlich besser wäre hier eine standardisierte Beschreibungssprache. Hier ist sicherlich der bekannteste Vertreter das noch relativ junge XML.

Die eXtensable Markup Language gehört zur Familie der Markup-Sprachen, zu denen beispielsweise HTML und LaTeX zählt. Markup-Sprachen kodieren die Strukturierung und Formatierung der Daten nur mit Text, was die Lesbarkeit fördert und Erweiterungen vereinfacht. Bereits 1974 gab es erste Ansätze, Daten mit Hilfe einer Markup-Sprache zu beschreiben. Diese ersten Bestrebungen mündeten in die Spezifikation von SGML[1]. Sie wurde in den folgenden Jahren in vielen Bereichen zur Dokumentation und Datenhaltung verwendet und diente als Grundlage zur Entwicklung von HTML. 1996 begann das W3C, eine vereinfachte Variante von SGML zu entwickeln, woraus 1998 die erste Version von XML [8], als Teilmenge von SGML, entstand. Durch diese Vereinfachung sollte XML eine größere Verbreitung finden. In den folgenden Jahren entwickelten sich eine Unmenge von Applikationen und Sprachen, die auf die XML-Definition aufbauten. Bekannte Vertreter sind der HTML-Nachfolger XHTML, MathML, SVG und PList.

XML verwendet zur Strukturierung von Daten sogenannte Tags (Bsp. `<tag>`). Diese Tags gliedern die eigentliche Information (`<text>the quick brown fox</text>`). Sie können mit weiteren Eigenschaften, sogenannten Attribute angereichert werden (`<text size="10" color="green">...`). Bei HTML sind die einzelnen Tags bereits vordefiniert und erfüllen spezielle Aufgaben zur Strukturierung und Formatierung. Bei XML sind diese Tage frei definierbar, wodurch sich mit XML beliebige Daten hierarchisch strukturieren lassen.

Ein XML-Dokument muß „wohlgeformt“ sein, d.h. jeder „offnende“ Tag hat auch einen „schließenden“ Tag. Dies allein beschreibt jedoch die Datenstruktur noch nicht ausreichend. Um XML-Dokumente austauschen zu können, benötigt man eine Beschreibungssprache. Durch so eine Beschreibung könnte man dann XML-Dokumente verifizieren. Diese Beschreibungen stellen sozusagen eine Grammatik dar, die von einem Parser überprüft werden kann. Eine dieser Beschreibungssprachen, die bereits bei SGML eingesetzt wurde ist die sogenannte DTD[2]. Sie beschreibt die einzelnen Tags (und deren Attribute) und wie sie ineinander geschachtelt werden dürfen.

4.1.3 UXF

Eine solche DTD für UML wurde beispielsweise von Suzuki und Yamamoto[37] entwickelt. Die Autoren verweisen in ihrem Artikel auch auf andere Ansätze wie beispielsweise CDIF, SMIF und STEP. Auf diese, teilweise nicht XML-basierten, Formate soll hier nicht näher eingegangen werden. UXF enthält ebenfalls keine Geometrieinformationen, sodaß eine entsprechende graphische Repräsentation selbst generiert werden muß. Ein Konverter aus dem Rational-Format MDL zu UXF existiert zwar, jedoch wird UXF von keinem Modellierungswerkzeug verwendet, da es nicht offiziell als Standard anerkannt wurde.

[1] Standard Generalized Markup Language
[2] Document Type Definition

```
<?xml version="1.0"?>
<!DOCTYPE UXF SYSTEM "uml.dtd">

<UXF Version="2.0"
     xmlns:UXF="http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/">
  <ClassDiagram>
    <Name>Polygon</Name>
     <Class>
      <Name>Polygon</Name>
      <IsAbstract>true</IsAbstract>
      <Attribute>
        <Name>points</Name>
      </Attribute>
    </Class>
  </ClassDiagram>
</UXF>
```

Abbildung 4.2: Beispiel: UXF

4.1.4 XMI

Der Bedarf nach einem Standard-Austauschformat führte zu Initiative mehrere Forschungseinrichtungen und Firmen, woraus zuerst das Format MOF[3] zum Austausch von Metadaten entstand. Ziel der Entwicklung war ein Austauschformat für beliebige Datenstrukturen. Dieser Ansatz geht weit über XML hinaus. Er beschreibt neben den Datenstrukturen selbst auch deren Zugriffsmethoden und Verknüpfungen. Daten werden bei MOF in sogeannten Repositories abgelegt, was sehr leistungsfähigere Abfragemechanismen ermöglicht. MOF ist als Framework zur Entwicklung von Metadatensystemen gedacht, ein Beispiel solcher Metdaten sind UML-Modelle. Die Beschreibung solcher Metadatensysteme durch eine XML-Datei kann durch XMI[4] geschehen. Um UML-Modelle als Metadaten zu beschreiben, wurde also eine solche XMI-Spezifikation für UML entwickelt. Dieses sehr leistungsfähige Konzept zur Datenbeschreibung hat natürlich seinen Preis, der sich beispielsweise in der 800 Seiten starken Spezifikation niederschlägt. Entsprechend kompliziert sind vollständige Implementierungen, so daß nur langsam ein XMI Im- oder Export in Modellierungswerkzeugen Einzug findet. Erste Implementierungen zeigen jedoch bereits die Schwächen dieses Standards. Hier zeigt sich wieder, wie durch eine zu große Entwicklerschaft die Anzahl der Interpretation zunimmt. An der XMI-Spezifikation waren mehr als zwanzig Firmen beteiligt, was letztlich dazu geführt hat, daß es verschiedene nicht vollständig kompatible Ansätze zu XMI gibt. Beispielsweise ist das Format, welches zur Zeit von Rational Rose produziert wird nicht mit dem vereinbar, was Together produziert. Somit ist dieses Format zur Zeit noch nicht sinnvoll einsetzbar.

4.1.5 Weitere Ansätze

Eine andere SGML-DTD (UML-Xchange) stammt von Normand Rivard [28]. Sie wurde wohl nur ad hoc für ein spezielles Projekt verwendet und wurde nicht weiter entwickelt oder verwendet. Auch sie verwendet keine Geometrieinformation und unterstützt nicht alle von UML 1.3 geforderten Konstrukte.

[3] Metadata Object Facility
[4] XML Metadata Interchange

Im Rahmen eines Praktikums entstand am Northeastern University's College of Computer Science das Format UMLtext. Hier wurde eine Präprozessor verwendet, der die UML-Diagramminformation von dem Java-Sourcecode trennt, gearbeitet wird an einer gemeinsamen Datei – ein Ansatz der aus dem Literate-Programming bekannt ist.

Speziell auf den Sprachumfang von Java zugeschnitten ist das Java-Doclet Javadox [10]. Doclets sind Erweiterungen des Java-Dokumentationswerkzeuges Javadoc, welches normalerweise eine HTML-Dokumentation der Schnittstellen generiert. Javadox erweitert diese Generierung um einen XML-Export, der teilweise auch den Inhalt eines Klassendiagrammes widerspiegelt.

Abschließend läßt sich sagen, daß keines der genannten Formate mit vertretbarem Aufwand zu verwenden ist. Vor allem die fehlende Unterstützung der Geometrieinformationen verhindert eine sinnvolle Verwendung bei der Darstellung von Diagrammen sofern man sich nicht selbst um die Generierung der Diagramme kümmern will. Eine eigene Datenstruktur, die auf die hier gestellte Problematik angepaßt ist, scheint somit sinnvoller. Eine spätere Konvertierung in anderen XML-Repräsentationen ist jedoch ohne größere Probleme möglich. Beispielsweise wurde während dieser Studienarbeit ein Java-Programm entwickelt, welches aus den Ausgaben des Java-Doclets Javadox ein XML-Dokument im unten beschriebenen Format generiert.

4.2 Auswahl der zugrundeliegenden Grammatik

Die in 4.1 vorgestellten Formate basieren auf unterschiedlichen Technologien bzw. Grammatiken. UMLscript beispielsweise ist durch eine kontextfreie Grammatik definiert, die sich in der BNF-Form[5] niederschreiben läßt. Andere Ansätze orientieren sich an dem Syntax bekannterer Programmiersprachen, der größte Teil verwendet jedoch XML.

XML bietet einige Vorteile gegenüber anderen Definitionssprachen von Datenstrukturen. Durch die genaue Spezifikation des Syntax ist bereits eine Vielzahl von Parser-Frameworks zur Verarbeitung solcher Dateien verfügbar. Die Nähe zu HTML ermöglicht eine einfache Darstellung in Webbrowsern und durch die weite Verbreitung ist die Technologie bereit recht bekannt und erprobt. Aus diesen Gründen wurde der Spezifikation von UML-Klassendiagrammen hier die Grammatik von XML zugrunde gelegt.

4.3 Dokumentbeschreibungssprachen

Zur Beschreibung von Sprachen, die der XML-Grammatik folgen, wurden mehrere Ansätze entwickelt. Noch von SGML sind die sogenannten *DTDs* bekannt, die mit einer recht flachen Struktur die einzelnen Elemente und Attribute von XML beschreiben. Die DTD selbst entspricht nicht dem XML-Format, was ihre Verarbeitung erschwert. Jedoch verwenden die meisten XML-Parsers eine DTD zur Validierung, da DTDs zur Zeit noch als Standard zur Beschreibung gelten.

[5] Backus-Naur-Form

```
<!ELEMENT phonebook (entry)+>
<!ELEMENT entry EMPTY>
<!ATTLIST entry
  name     ID     #REQUIRED
  phone    CDATA  #REQUIRED
  birthday CDATA  #IMPLIED
>
```

Abbildung 4.3: Beispiel: Document Type Definition (DTD)

Einer der größten Nachteile von DTDs ist die fehlende Objektorientierung, die die Erstellung von umfangreicheren Spezifikationen erschwert. Deshalb wurden von verschiedenen Firmen neue Ansätze zu diesem Thema erarbeitet, darunter *DCD* von Microsoft und IBM welches später zu *Microsoft-Biztalk* erweitert wurde. Ein weiterer Entwurf namens *XSchema* [40] wurde von den Mitgliedern der XML-DEV-Mailingliste erstellt. Er fand jedoch keine weite Verbreitung. Die meisten dieser Ansätze entstanden aufgrund der Tatsache, daß sich das World Wide Web Consortium (W3C) nicht für einen eigenen Standard entscheiden konnte. Erst 1999 gab es einen „Working Draft“ für den neuen Standard zur XML-Strukturbeschreibung *XMLSchema*. Inzwischen ist eine „Recommendation“ [12] erreicht; erste Schema-validierende Parser für XML sind verfügbar oder zumindest in Entwicklung. XMLSchema orientiert sich wesentlich stärker an Datentypen als DTDs. Durch Definition von Attributgruppen und genauere Typdefinitionen bietet es wesentlich mehr Möglichkeiten XML-Dokumente exakt zu beschreiben.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema" elementFormDefault="qualified">
  <xsd:element name="phonebook">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="entry">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:ID" use="required"/>
            <xsd:attribute name="phone" type="xsd:CDATA" use="required"/>
            <xsd:attribute name="birthday" type="xsd:date" use="optional"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Abbildung 4.4: Beispiel: XMLSchema Definition (XSD)

Im Abbildung 4.4 sieht man eine einfache XMLSchema-Definition. Nach dem Wurzelknoten `xml:schema` folgt die Definition eines einfachen Telefonbuches, dessen Wurzelknoten `phonebook` heißt. Diesem `phonebook` wird nun ein Datentyp zugeordnet. Hier sind zwei verschiedene Typen möglich: Bei sogenannten `simpleTypes` wird direkt ein Datentyp (beispielsweise Integer, Date, Text, etc.) angegeben. Zusammengesetzte Typen bezeichnet man als `complexType`. Wie man oben sehen kann, besteht dieser `complexType` hier aus dem Hauptknoten `entry`, welcher drei Attribute besitzt. Durch den Knoten `xsd:sequence` wird definiert, daß die Unterknoten in der angegebenen Reihenfolge vorkommen müssen, was

hier bei Attributen jedoch keinen Unterschied macht. Schon interessanter ist das Attribute `maxOccurs="unbounded"` welches angezeigt, daß das Element `entry` beliebig oft vorkommen kann. Das Element `entry` ist wieder durch einen `complexType` spezifiziert, welcher hier nur die Attribute festlegt. Neben Name und Typ wird hier außerdem definiert, ob diese Attribute optional oder zwingend anzugeben sind.

Bei diesem einfachen Beispiel sind noch nicht alle Vorteile von XMLSchema ersichtlich. Beispielsweise lassen sich `simpleTypes` und `complexTypes` mit Namen versehen, wodurch sie an mehreren Stellen innerhalb der XMLSchema-Definition wiederverwendet werden können.

4.4 Programmiersprachen

Im Zusammenhang mit XML hat das W3C auch zwei standardisierte APIs zur Verarbeitung von XML-Dokumenten definiert. Die etwas einfachere Variante heißt *SAX*, was für *Simple API for XML parsing* steht. Diese API ist streamorientiert, sie liest das XML-Dokument ein und erzeugt Nachrichten wenn der Anfangs- oder Endtag erreicht wird. Diese Nachrichten können dann empfangen und verarbeitet werden. Dies bietet sich vor allem für Daten an, die nur einmalig gelesen werden müssen. Ein etwas anderer Ansatz ist *DOM* (Document Object Model), hier wird das XML-Dokument als Baumstruktur in den Speicher eingelesen und kann danach beliebig traversiert werden. XML-Parser, die diese beiden APIs anbieten sind für nahezu alle Programmiersprachen verfügbar, jedoch am weitesten entwickelt sind sie für Java. Gerade die Implementation von Sun bietet schon sehr viel zusätzliche Funktionalität. Java bietet sich auch aus weiteren Gründen an, es ist eine relativ saubere Programmiersprache, die mit einer leistungsfähigen Bibliothek ausgeliefert, die die wichtigsten Bereiche der Ein- und Ausgabe sowie Stringverarbeitung beinhaltet. Außerdem besteht die Möglichkeit durch sogenannte Java-Applets kleine Programme im Browser ablaufen zu lassen. Auch die Plattformunabhängigkeit von Java sei hier erwähnt, sie ergänzt sich hervorragend mit der Offenheit von XML.

4.5 Darstellungsformat

Wie in Abschnitt 2.2.6 erwähnt, bieten sich vektororientierte Formate zur Darstellung der Diagramme an. Pixelorientierte Formate bieten im Allgemeinen keine Möglichkeit Textinhalte zu transportieren, ohne das deren Inhalt in Textform erhalten bleibt und nachträglich geändert werden kann. Auch eine variable Darstellung wie beispielsweise Zooming ist nur bedingt anwendbar. Somit scheiden die bekannteren Formate wie GIF und BMP aus.

Im Bereich der Vektorformate ist vor allem Postscript recht stark verbreitet. Postscript ist so gesehen kein Dateiformat, sondern eher als Programmiersprache zu betrachten. Hierin liegt auch einer der Hauptnachteile, nämlich die unüberschaubare Vielfalt und Komplexität der erstellbaren Dokumente. Die Erstellung ist zumeist noch relativ einfach, jedoch nachträgliches Verändern und vor allem Animation sind unmöglich. An Postscript angelehnt entstand vor ein paar Jahren das Portable Document Format (PDF). PDF bietet bereits rudimentäre Möglichkeiten zur Animation, jedoch ist dazu weder die Spezifikation offengelegt, noch existiert eine Programmierschnittstelle zur Verwendung.

Dateiformate, die auf ein Betriebssystem beschränkt sind, bieten sich ebenfalls nicht an. In diese Kategorien fällt das Windows Metafile Format (WMF) von Microsoft oder PICT von Apple.

Schon eher den Voraussetzungen entsprechend sind die 3D-Formate VRML und Open-GL sowie Java-3D. Open-GL ist aufgrund seiner anderen Ausrichtung nicht geeignet. VRML und Java-3D würden die meisten Voraussetzungen erfüllen, da jedoch dreidimensionale Darstellung nicht benötigt wird, sollte primär ein 2D-Format eingesetzt werden.

Bereits vor einigen Jahren hatte die Firma Macromedia ein Browser-Plugin entwickelt, welches einfache vektorbasierte Graphiken und Animationen erlaubt. Die Erstellung dieser sogenannter Flash-Dateien war jedoch nur mit Produkten von Macromedia möglich. Microsoft unternahm mit VML (Vektor Markup Language) einen weiteren Vorstoß, der jedoch bisher keine weitere Verbreitung fand.

```
<svg width="5cm" height="2cm">
  <desc>anuclad</desc>
  <text x="5" y="5" fill="red">The quick brown fox jumps over the lazy dog</text>
</svg>
```

Abbildung 4.5: Beispiel: Scalable Vector Graphics (SVG)

Auf der Basis von Postscript und PDF reichte im April 1998 Adobe ein Proposal im W3C ein [1]. Daraus wurde später unter der Federführung von Adobe der W3C-Standard SVG entwickelt. SVG ist XML-basierend und paßt somit ideal in die bisher gewählte Umgebung. Animation wird unterstützt, das Dateiformat ist einfach generierbar und lehnt sich an viele andere W3C-Standards an. Je nach Implementation von SVG sind Browser-Erweiterungen, alleinstehende Applikationen und Server-Anwendungen verfügbar. Von Adobe selbst stammt das Browser-Plugin SVGViewer[6], welches während der Laufzeit mit Java und Javascript steuerbar ist und sowohl für Windows als auch Mac OS verfügbar ist.

Durch sein Mächtigkeit aber auch die weitreichenden Animationsmöglichkeiten wurde SVG als Darstellungsformat ausgewählt. Tabelle 4.1 bietet nochmals einen Überblick über alle hier erwähnten Formate.

Dateiformat	Vektororientiert	Dateiart	Browsertauglich	Animierbar	Offener Standard
GIF	nein	Binär	ja	nicht ausreichend	Patentproblematik
PNG/JPEG	nein	Binär	ja	nein	ja
Postscript	ja	Binär oder Text	nein	nein	teilweise
PDF	ja	Binär oder Text	bedingt	unzureichend	teilweise
WMF/PICT	ja	Binär	nein	nein	nein
VRML	ja	Text	Plugin	ja	ja
SVG	ja	XML	Plugin	ja	ja

Tabelle 4.1: Übersicht der Darstellungsformate

[6] Download unter `http://www.adobe.com/svg/`

Kapitel 5

Das XMLSchema für Klassendiagramme

In Abschnitt 4.1 wurde bereits festgestellt, daß keines der bisher vorhandenen Austauschformate sich direkt für den hier vorgestellten Ansatz zur Visualisierung eignet. In Anlehnung an die vorgestellten Formate wird hier nun eine eigene Sprache spezifiziert. Bei dem Entwurf des Konvertierwerkzeuges sollte jedoch Wert auf eine modularisierte Entwicklung gelegt werden, sodaß eine spätere Anpassung an andere Formate (beispielsweise XMI) möglich ist.

Zuerst ein kurzer Überblick über XML und XMLSchema. Eine XML-Datei läßt sich durch einen Baum darstellen. Wie bei Bäumen üblich existiert ein ausgezeichneten Wurzelknoten, an dem beliebig viele Unterknoten direkt oder indirekt angehängt sind. Diese Knoten können drei verschiedenen Typen angehören.

1. Attribute beschreiben Eigenschaften des Vaterknotens
   (Im Beispiel: `created="1.12001"`)
2. Textknoten stellen die eigentlichen Inhalt des Knotens dar
   (Im Beispiel: `Manfred Mustermann`)
3. Weitere Unterknoten
   (Im Beispiel: `<name>` als Unterknoten von `entry`)

```
<?xml version="1.0" encoding="UTF-8"?>
<phonebook>
  <entry created="1.1.2001">
    <name>Manfred Mustermann</name>
    <phone>08/15</phone>
    <birthday>11.11.1911</birthday>
  </entry>
</phonebook>
```

Abbildung 5.1: Beispiel einer XML-Datei

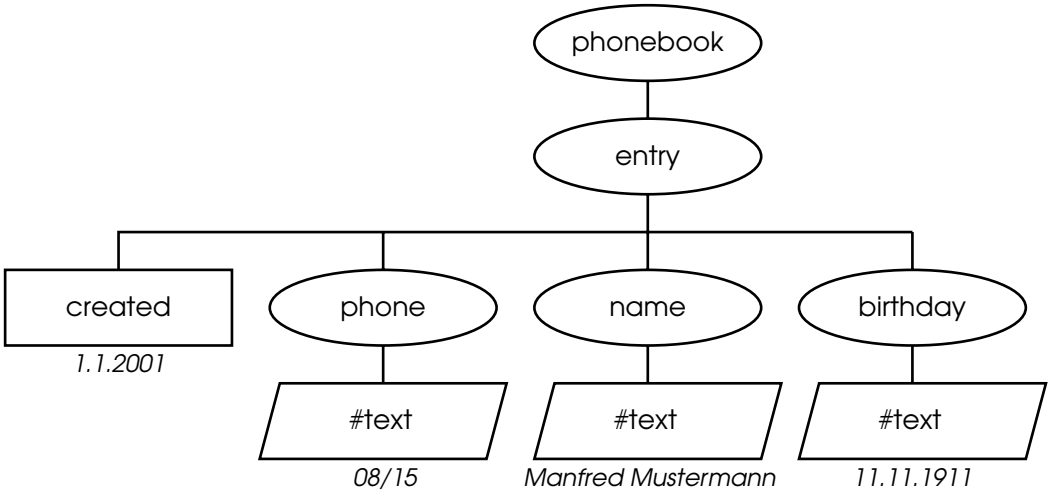

Abbildung 5.2: XML-Beispieldatei als Baum dargestellt

In 5.1 ist eine XML-Datei dargestellt. Die entsprechende Baumstruktur ist in Abbildung 5.2 zu finden. Der Wurzelknoten ist `phonebook`, darunter folgt ein Eintrag namens `entry` mit dem Attribut `created` und den drei Unterknoten `name`, `phone` und `birthday`. Um eine solche Datenstruktur zu beschreiben existieren mehrere Beschreibungssprachen. Beispiele für eine entsprechende DTD und ein XMLSchema sind in 4.3 zu finden.

Um Klassendiagramme mit XML zu beschreiben wird im folgenden eine geeignete Datenstruktur entwickelt. Bei dem Entwurf sollten folgende Punkte berücksichtigt werden:

1. XML-basierend
2. Beschreibung der Grammatik durch ein XMLSchema
3. erweiterbar für andere UML-Diagrammtypen
4. Unterstützung der wichtigsten Diagrammelemente von UML 1.3 (siehe [25])
5. Unterstützung von Geometrieinformationen (Koordinaten der graphischen Elemente)
6. Aussehen von Objekten kann optional geändert werden (Farbe, Größe)
7. Jedes Objekt kann mit einer Dokumentation versehen werden
8. URLs zum Verweis auf weitere Informationen (beispielsweise Sourcecode) können eingefügt werden
9. Eindeutige Identität für jedes Diagrammelement (nötig zur eindeutigen Referenzierung)

Der Wurzelknoten wir mit `umldiagrams` bezeichnet. Als Unterknoten können dann ein oder mehrere Diagramme beschrieben werden. Für Klassendiagramme wurde die Bezeichnung `classdiagram` gewählt. Andere Diagrammtypen sind bisher noch nicht spezifiziert.

```
<?xml version="1.0" encoding="UTF-8"?>
<umldiagrams xmlns="http://www.girschick.net/martin/study/anuclad/umldiagrams">
  <classdiagram name="..." id="..." ...>
    ...
  </classdiagram>
</umldiagrams>
```

Abbildung 5.3: Aufbau einer XML-Datei zur UML-Diagrammbeschreibung

Um die einzelnen Elemente eines Klassendiagramms beschreiben zu können benötigt man eine gewisse Grundmenge an Parametern wie Position, Größe oder Farbe. Hinzu kommen einige UML-Konstrukte, die in nahezu allen Elementen vorkommen. XMLSchema ermöglicht es, solche zusammengesetzten Datentypen zu beschreiben und den einzelnen Unterknoten zuzuweisen. In Tabelle 5.1 sind die Attribute des Datentyps `objectType` beschrieben, die Knotenstrukur wird in Abbildung 5.4 dargestellt. Das Attribut `hidden` dient zum Verstecken von Diagrammelementen, es ist implizit `false`, wenn es nicht näher spezifiziert ist.

```
<objectType name="..." id="..." visibility="..." stereotype="..." hidden="true" url="...">
  <annotation>...</annotation>
  <note>...</note>
  <appearance color="..." size="..."/>
  <properties>
    <property description="..." value="..."/>
    ...
  </properties>
</objectType>
```

Abbildung 5.4: Knotenstruktur von `objectType`

Diese Trennung zwischen Attributen und Unterknoten wurde aus mehreren Gründen vorgenommen. Die Attribute des Hauptknotens enthalten die Informationen, die direkt zur Darstellung des Objektes benötigt werden. `annotation` und `note` sind optionale Knoten, die unter Umständen längere Textpassagen enthalten und daher durch Knoten besser strukturiert werden. Der `appearance`-Knoten wurde als optionale Erweiterung ausgelagert, da er mit der Semantik des Klassendiagramm nichts zu tun hat, ähnliches gilt für die optionalen `properties`. Diese Paare von `description` und `value` wurden hinzugefügt, um beliebige Informationen ergänzen zu können. Anwendungsbereiche währen beispielsweise Autor, Erstellungsdatum, Status des Diagrams etc.

Ein weiterer zusammengesetzter Datentyp ist der Knoten `geometry`, der als Unterknoten von graphischen Elementen auftritt. Es mag zuerst verwundern, daß alle vier Attribute als optional definiert wurden, dadurch ist es jedoch möglich wahlweise nur die Position oder nur die Größe zu definieren. Die Größe der Elemente kann i.Allg. automatisch berechnet werden, dies ist, wie bereits erwähnt, für die Positionierung erheblich schwieriger. Bei Nichtangabe der Position wird versucht ein einfaches automatisches Layout zu generieren.

Eine erste Anwendung der beiden Datenstrukturen `objectType` und `geometry` findet man in dem Knoten `classdiagram`, dieser Knoten beschreibt ein vollständiges Klassendia-

Knoten	Name	Datentyp	Benutzung	Vermerk
objectType	name	CDATA	verbindlich	Name des Objekts, beispielsweise Klassenname oder Methodenname in Klassendiagrammen. Beispiel: `main(String[])`
	id	ID	verbindlich	IDs sind Namen, die innerhalb des Dokuments eindeutig sind, sie dienen zur Identifikation. Beispiel: `anuclad.Class.main`
	visibility	CDATA	optional	Dieses Attribut ist an die UML-Spezifikation angelehnt, es beschreibt die Sichtbarkeit gegenüber anderen Objekten. Beispiel: `public`
	stereotype	CDATA	optional	Ebenfalls der UML-Spezifikation entsprechend können Objekte einen Stereotyp besitzen, der ihre Eigenschaften beschreibt. Beispiel: `<<static>>`
	hidden	boolean	optional	Manche Objekte müssen nicht im Diagram dargestellt werden, sie sind jedoch für eine vollständige Beschreibung notwendig und können beispielsweise bei späterer Animation eingeblendet werden. Beispiel: `true`
	url	uriReference	optional	Dieser Verweis kann auf weitere Dokumentation zu dem entsprechenden Objekt verweisen, beispielsweise dem Sourcecode oder Javadoc. Beispiel: `Class#main(String[])`
annotation	–	CDATA	optional	Der annotation-Unterknoten beinhaltet eine Beschreibung des Elements.
note	–	CDATA	optional	UML definiert kleine „Post-it-Notes“, die Objekte des Diagrams näher beschreiben.
appearance	color	CDATA	optional	Mit dem color-Attribut kann man die Farbe verändern. Bei den meisten Elementen ist dies die Hintergrundfarbe. Beispiel: `green`
	size	int	optional	Um manche Elemente kleiner oder größer darzustellen sind hier Werte von -2 bis $+2$ erlaubt. Beispiel: 1
properties		EMPTY	optional	Dieser Knoten faßt weitere Eigenschaften des Objektes zusammen.
property	description	CDATA	verbindlich	Die Beschreibung der Eigenschaft Beispiel: `author`
	value	CDATA	verbindlich	Wert der Property. Ist dieser nicht vorhanden ist, wird der Wahrheitswert true angenommen Beispiel: `Martin Girschick`

Tabelle 5.1: Knoten und Attribute des `objectType`-Datentyps

Knoten	Name	Datentyp	Benutzung	Vermerk
geometry	x	int	optional	Position in horizontaler Richtung
	y	int	optional	Position in vertikaler Richtung
	width	int	optional	Breite des Objektes
	height	int	optional	Höhe des Objektes

Tabelle 5.2: Attribute des `geometry`-Knotens

gramm. Durch den `geometry`-Knoten werden die Gesamtabmessungen des Diagramms definiert. Als direkte Unterknoten von `classdiagram` kommen zwei Typen in Frage. Der Knoten `package` beschreibt ein Paket, welches in UML durch eine Karteikarte mit einem Reiter, in dem der Name des Paketes steht, dargestellt wird. Da es auch Klassendiagramme geben kann, die kein Paket enthalten kann alternativ zu diesem Knoten auch direkt ein Klassifizierer (engl. classifier) folgen. Auch eine gemischte Struktur mit Paketen und Klassifizierern ist möglich, wobei Pakete wiederum weitere Unterpakete enthalten können. Um Pakete im Diagramm einfacher bewegen zu können sind die Positionsangaben von Objekten innerhalb eines Paketes immer relativ zu der Position des Paketes selbst.

Zu den Klassifizierern zählen die folgenden Knoten: `class`, `interface`, `template`, `exception`. Bis auf den letzten Typ sind sie bereits durch die Unified Modelling Language definiert. `Exception` wurde hinzugenommen, da sie oft bei der Modellierung von Fehlerbehandlungen benötigt werden und aller Voraussicht nach auch in der nächsten Version von UML unterstützt werden. Java bietet beispielsweise einen expliziten Exception-Typ zur Programmierung an. Alle vier Typen haben den selben Aufbau, der durch Tabelle 5.3 dargestellt wird. Die Knoten, die durch `objectType` und `geometry` definiert werden, sind nicht näher aufgeführt. Alle Knoten sind um das Attribut `abstract` ergänzt worden, welches bei `class` und `exception` mit `false` und bei `interface` und `template` mit `true` vorbelegt ist. XMLSchema bietet zwar diese Möglichkeit, Attribute mit Standardwerten zu belegen, die von einem XMLSchema-fähigen Parser automatisch erkannt werden. Jedoch sind zur Zeit kaum XMLSchema-fähige Parser verfügbar.

Knoten	Name	Datentyp	Benutzung	Vermerk
operations	–		optional	Sammelknoten für alle Operationen
operation	–		verbindlich	Unterknoten von `operations` vom Typ `objectType`
	inherited	boolean	optional	Diese Operation wurde von einer Vaterklasse ererbt.
	static	boolean	optional	Methode ist statisch
	returntype	CDATA	optional	Falls vorhanden, gibt dieses Attribute den Typ des Rückgabewertes an. Standard ist `void`.
	abstract	boolean	optional	Diese Methode wurde hier nur abstrakt definiert
attributes	–		optional	Sammelknoten für alle Attribute (Klassenvariablen)
attribute	–		verbindlich	Unterknoten von `attributes` vom Typ `objectType`
	inherited	boolean	optional	Dieses Attribut wurde von einer Vaterklasse ererbt.
	static	boolean	optional	Attribut ist statisch (also in allen Instanzen gleich)
	type	CDATA	optional	Datentyp dieses Attributs
	derived	boolean	optional	Die Berechnung dieses Attribut hängt direkt von anderen Attributen ab.
	value	CDATA	optional	Beispielsweise bei fest-definierten Attributen (final) kann hier der Wert spezifiziert werden.
relations			optional	Abhängigkeiten zu anderen Klassifizierern werden durch Relationen angeben (siehe nächster Abschnitt).

Tabelle 5.3: Unterknoten und Attribute eines Klassifiziererknotens

Bei Operationen sind noch weitere Unterknoten vom Typ `parameter` möglich. Sie beschreiben die Eingabeparameter einer Operation. Dazu besitzen sie zwei Attribute `type` und `name`, wobei der Typ verbindlich und der Name optional ist.

Am Ende der Tabelle wurden bereits die Relationen angesprochen, sie stellen Zusammenhänge zwischen Klassen her. Die Unified Modelling Language unterscheidet hier drei Typen:

Assoziation Assoziationen stellen Beziehungen zwischen Klassen (oder allgemeiner Objekten) her. Eine einfache Assoziation wird durch einen Strich zwischen den beteiligten Objekten realisiert. Durch eine Pfeilspitze kann angezeigt werden, daß der Zugriff nur in Pfeilrichtung möglich ist. In Programmiersprachen wird hierfür oft eine Referenz verwendet. Etwas stärker als eine einfache Assoziation ist die Aggregation, sie

zeigt an, daß der Aggregat ein anderes Objekt *enthält*. Dargestellt wird dies durch eine hohle Raute auf der Aggregatseite. Noch stärker ist die Komposition, wo aus der Zerstörung des Aggregaten auch die Zerstörung der komponierten Objekte folgt. Hier wird die Raute ausgefüllt dargestellt. An beiden Seiten der Assoziation können Kardinalitäten ergänzt werden, die die Mengenverhältnisse dieser Relationen ausdrücken.

Generalisierung Sie wird auch als Vererbung bezeichnet. Graphisch drückt sie sich durch einen Pfeil in Richtung der Vaterklasse aus.

Abhängigkeit Die Abhängigkeit stellt eine programmiertechnische Eigenschaft dar. Sie zeigt an, daß Änderungen einer Klasse oder eines Paketes eventuell Änderungen in einer andere Klasse (oder Paket) nach sich ziehen. Zur Darstellung werden unterbrochene Linien verwendet, die bei Bedarf mit einer Pfeilspitze versehen werden.

Da Abhängigkeiten sehr selten verwendet werden, werden nur die ersten beiden Typen in das Dateiformat eingebunden. Alle Relationen werden durch den Hauptknoten `relations` zusammengefaßt. Dieser hat beliebig viele Unterknoten vom Typ `association` oder `generalization`. Beide Typen besitzen neben denen durch `objectType` definierten Knoten und Attributen ein Attribut `to`, welches das Zielobjekt angibt, auf welches verwiesen wird. Um eine etwas flexiblere graphische Darstellung zu ermöglichen besteht außerdem die Möglichkeit, einen weiteren Unterknoten namens `path` zu verwenden, der eine Folge von Koordinaten (`coordinates`) definiert, die den Pfad von dem einen zum andern Klassifizierer beschreiben.

Der `association`-Knoten hat noch fünf weitere Attribute, die durch Tabelle 5.4 erklärt werden. Wie man hier sieht, wurden die einzelnen Assoziationstypen nicht als Knoten sondern als boolsche Attribute realisiert. Dies hat mehrere Hintergründe. Das parsing wird dadurch vereinfacht, außerdem spiegelt es auch das typische Modellierungskonzept wieder. Eine Assoziation zwischen Klassen kann beispielsweise zu einer Komposition verstärkt werden. Dies würde dann nur das Attribut des Knotens ändern und ihn nicht durch einen anderen Knoten ersetzen.

Knoten	Name	Datentyp	Benutzung	Vermerk
association	multiplicity	CDATA	optional	Gibt die Kardinalität des Quellobjektes an.
	to_multiplicity	CDATA	optional	Die Kardinalität des Zielobjektes
	directed	boolean	optional	Wenn die Assoziation gerichtet ist
	aggregation	boolean	optional	Zeigt eine Aggregation an
	composition	boolean	optional	Zeigt eine Komposition an

Tabelle 5.4: zusätzliche Attribute des `association`-Knotens

Auf Seite 43 ist die Knotenstruktur nochmals als Diagramm dargestellt. Die Beschreibung der Attribute finden sich in den entsprechenden Tabellen. Das Diagramm wurde mit dem XML-Editor *XMLSpy* von Altova erstellt. Beispieldateien, die dieser Spezifikation entsprechen sind im Anhang A ab Seite 61 zu finden.

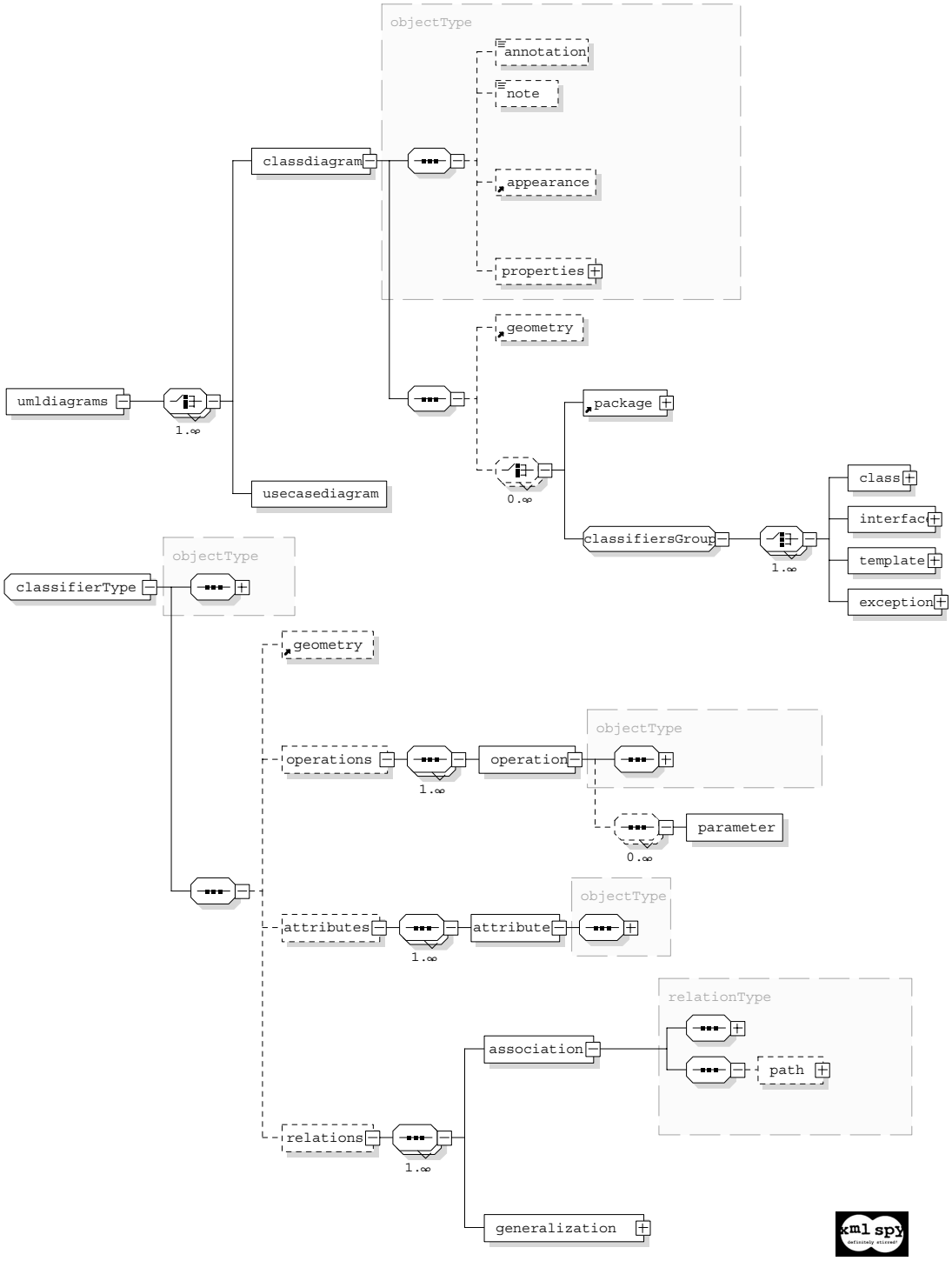

Abbildung 5.5: Knotenaufbau einer UML-Diagrammbeschreibung für Klassendiagramme

Kapitel 6

Das XMLSchema für Animation in Klassendiagrammen

Im vorigen Kapitel wurde ein Format zur Beschreibung von Klassendiagrammen entwickelt. Um nun Animationen innerhalb dieser Klassendiagramme zu beschreiben wird in diesem Kapitel ein weiteres XMLSchema entwickelt. Bei dem Entwurf wurde versucht auch allgemeine graphische Animationen zu ermöglichen, die nicht direkt zur Darstellung von Abläufen innnerhalb von Programmen gedacht sind. Diese können beispielsweise zur Erklärung der Diagramme verwendet werden, wie in Abschnitt 3.1 vorgeschlagen wurde. Die Datenstruktur orientiert sich bewußt an der graphischen Darstellung. Sie ist nicht dazu geeignet, Abläufe vollständig und semantisch korrekt zu beschreiben, da eine semantisch korrekte Darstellung wesentlich umfangreicher geworden wäre und eine entsprechende Visualisierung den Rahmen dieser Arbeit gesprengt hätte. Dennoch ist es möglich, aus Ablaufdaten entsprechende Animationsdateien zu erzeugen, als Beispiel sei hier auf Anhang A verwiesen.

Der Wurzelknoten ist `animations`, darunter befinden sich einer oder mehrere `animation`-Knoten. Jeder dieser Knoten beschreibt einen Ablauf, der auf einem bestimmten Diagramm dargestellt werden soll. Die Beschreibung der Attribute des `animation`-Knotens sind in Tabelle 6.1 aufgeführt. In Abbildung 6.1 ist die Knotenstruktur dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<animations xmlns="http://www.girschick.net/martin/study/anuclad/animations">
  <animation name="..." id="..." diagram="..." description="..." ...>
    ...
  </animation>
</animations>
```

Abbildung 6.1: Aufbau einer XML-Datei zur Beschreibung von Animationen

Innerhalb der Animationsknoten sind dann die einzelnen Animationsbefehle aufgeführt. Die Reihenfolge ist dabei nicht von Relevanz, da durch Nebenläufigkeit und Schleifen diese nicht eindeutig interpretierbar wäre. Dennoch bietet es sich aus Übersichtlichkeitsgründen an, eine annähernd korrekte Reihenfolge zu wählen. Jeder Befehl hat einen Basissatz von Attributen, der den Zeitpunkt der Ausführung und verwandte Dinge beschreibt. Diese sind in Tabelle 6.2 zusammengefaßt.

Knoten	Name	Datentyp	Benutzung	Vermerk
animation	name	CDATA	verbindlich	Name der Animation
	id	ID	optional	Um Verweise zwischen Animation zu ermöglichen
	diagram	uriReference	verbindlich	Dieser Link zeigt auf die zu animierende Datei
	dimRelations	boolean	optional	Ist dieses Attribut `true`, werden alle Relation gedimmt.
	dimClassifiers	boolean	optional	Falls alle Klassifizierer am Anfang als nicht instanziiert dargestellt werden sollen.

Tabelle 6.1: Attribute des `animation`-Knotens

Knoten	Name	Datentyp	Benutzung	Vermerk
commandAttributes	begin	int	optional	Zeitpunkt, an dem dieser Befehl ausgeführt werden soll.
	duration	int	optional	Dauer des Befehls, falls keine Angabe, wird 100 angenommen.
	end	int	optional	Alternativ zur Dauer kann auch ein Endzeitpunkt angegeben werden.
	id	ID	optional	Eindeutige Bezeichnung dieses Befehls
	description	CDATA	optional	Dieser Text wird unterstützend zur Ausführung des Befehls in der Console ausgegeben.

Tabelle 6.2: Attributgruppe `commandAttributes`

Für Zeitparameter wurde der Datentyp Integer gewählt. Dabei entspricht die Zeiteinheit 1 einer hundertstel Sekunde, dies dürfte eine ausreichend hohe Granularität für Animationen bieten. Befehle, die für `begin` den Wert 0 besitzen werden als Initialisierungsbefehle betrachtet, sie werden sofort ausgeführt.

Die einzelnen Befehlen können in drei Gruppen unterteilt werden:

1. *Kontrollbefehle* dienen zur Steuerung der Animationsfolge, sie bieten Gruppierungs- und Verzweigungsmöglichkeiten.
2. *Grafikbefehle* bewegen Objekte oder ändern deren Aussehen.
3. Die letzte Gruppe sind die *Nachrichtenbefehle*, sie ermöglichen die Darstellung von Methodenaufrufen, deren Rückkehr, sowie die Erstellung und Zerstörung von Objekten.

Jedes dieser Befehle benötigt unterschiedliche Parameter, die als XML-Attribute übergeben werden. Zur Gruppe der Kontrollbefehle gehören folgende vier Befehle. Neben den bereits oben erwähnten Attributen besitzen sie nur sehr wenige zusätzliche Parameter, weshalb hier auf eine tabellarische Darstellung verzichtet wird.

wait Dient zum Anhalten der Animation zu einem bestimmten Zeitpunkt. Die Länge der Pause kann durch den `duration`-Parameter angegeben werden. Ist dieser nicht vorhanden wird die Animation pausiert, bis der Benutzer die Wiedergabe wieder startet.

alias Dieser Befehl ermöglicht es, auf andere Befehle oder Befehlsgruppen zu verweisen. Er überschreibt dabei die dort vorhandenen Zeitparameter mit eigenen Werten. Der entsprechende Befehl wird durch das Attribute `idref` bestimmt.

group Mit `group` lassen sich mehrere Befehle zusammenfassen. Die Zeitinformationen der Befehle innerhalb der Gruppe sind relativ zur Zeitangabe des Gruppenknotens. Der Gruppenknoten selbst hat zwei zusätzliche Attribute. `type` kann die beiden Werte `sequential` und `parallel` annehmen. Diese ermöglichen es, die Zeitinformationen innerhalb der Gruppe automatisch generieren zu lassen. `sequential` führt alle Befehle der Gruppe in der Reihenfolge aus, wie sie in der XML-Datei aufgeführt

werden. Sind `duration`-Werte vorhanden, so dienen sie zur Berechnung des `begin`-Parameters des nachfolgenden Befehls. Bei `parallel` werden alle Befehle gleichzeitig ausgeführt, wobei die Zeiten durch den Gruppenknoten bestimmt werden. Das `repeat`-Attribut des `group`-Knoten bietet die Möglichkeit eine Gruppe mehrfach zu wiederholen um beispielsweise Schleifen darzustellen. Die zeitliche Länge der Schleife wird durch `duration` kontrolliert, sofern dieses Attribut mit einem Wert belegt ist.

branch Oft werden mehrere Sequenzdiagramme konstruiert, die zu großen Teilen identische Abläufe enthalten. Mit dem `branch`-Befehl ist es möglich, diese Abläufe mit einem `animation`-Knoten zu beschreiben. Der `branch`-Knoten hat einen oder mehrere Unterknoten des Types `choice`. Wird der `branch`-Knoten ausgeführt erhält der Benutzer einen Auswahldialog, der die Namen und Beschreibungen der `choice`-Knoten anzeigt. Der Benutzer wählt eine der Alternativen aus, wodurch der durch `idref` referenzierte Befehl (oder die Befehlsgruppe) wiedergegeben wird. Damit sind auch Verweise zu anderen Animationen, die in der selben Datei definiert werden, möglich.

Die zweite Gruppe enthält zwei Befehle, die beide umfangreichere Parameter benötigen. Sie werden durch Tabelle 6.3 erklärt. Das Attribut `idrefs` ist vom Typ `NMTOKENS`, da nicht innerhalb der Animationsdatei sondern in das Klassendiagramm verwiesen wird. Der Datentyp `IDREF` von XML muß jedoch ein legales Gegenstück `id` in der selben Datei besitzen.

Knoten	Name	Datentyp	Benutzung	Vermerk
move	idrefs	NMTOKENS	verbindlich	Liste der zu bewegenden Objekte
	x	int	verbindlich	vertikale Verschiebung oder Neupositionierung
	y	int	verbindlich	horizontale Verschiebung oder Neupositionierung
	relative	boolean	optional	Normalerweise findet eine absolute Neupositionierung der Objekte statt. Bei `relative="true"` wird stattdessen eine Verschiebung vorgenommen
appearance	idrefs	NMTOKENS	verbindlich	Liste der zu ändernden Objekte
	instances	int	optional	Gibt die Anzahl der Instanzen einer Klasse an
	visual	NMTOKEN	optional	Dient zum verstecken (`hide`), anzeigen (`show`) oder dimmen (`dim`) von Objekten.
	color	CDATA	optional	Bestimmt die Farbe eines Objektes (normalerweise die Hintergrundfarbe).
	hasfocus	boolean	optional	Die gewählten Objekte erhalten den Kontrollfokus.

Tabelle 6.3: Attribute der beiden Grafikbefehle

In der letzten Gruppe sind vier Nachrichtenbefehle enthalten. Sie besitzen alle die selben Attribute, die in der Attributgruppe `messageGroup` (siehe Tabelle 6.4) zusammengefaßt sind. Mit `call` werden Methodenaufrufe dargestellt, die Rückkehr der Methode wird mit Hilfe des Befehls `return` visualisiert. Erstellung und Zerstörung von Instanzen geschieht über `create` und `destroy`.

Knoten	Name	Datentyp	Benutzung	Vermerk
messageGroup	from	NMTOKEN	verbindlich	ID des Diagramms, welches die entsprechende Nachricht ausgelöst hat.
	to	NMTOKEN	verbindlich	ID des Zielobjektes.
	focus	NMTOKEN	optional	Mögliche Werte sind: `follow`, `shadow`, `stay`, `duplicate`
	label	CDATA	optional	Dieser Text wird an der entsprechenden Nachricht im Diagram positioniert.

Tabelle 6.4: Attributgruppe der Nachrichtenbefehle

Durch `from` und `to` können sowohl Klassifizierer als auch direkt Methoden referenziert werden. Es bietet sich jedoch an direkt die Methode zu referenzieren, da diese dann

auch im Diagramm entsprechend hervorgehoben werden kann. Die möglichen Werte des `focus`-Parameters haben folgende Funktion.

follow ist der Standardwert. Er bedeutet, daß der Kontrollfokus der Nachricht folgt.

shadow arbeitet ähnlich wie follow, er entfernt jedoch den Kontrollfokus nicht vollständig vom auslösenden Objekt. Dies kann beispielsweise zur Darstellung einer noch im Aufrufstapel befindlichen Methode verwendet werden.

stay beläßt den Fokus beim Ausgangsobjekt. Dies eignet sich gut für Methodenaufrufe, die keinen Rückgabewert liefern und sofort zurückkehren.

duplicate versieht sowohl Quell- als auch Zielobjekt mit dem Kontrollfokus.

In Abbildung 6.2 wird die Knotenstruktur noch einmal übersichtlich wiedergegeben. Beispieldateien, die dieser Spezifikation entsprechen, befinden sich in Anhang A.

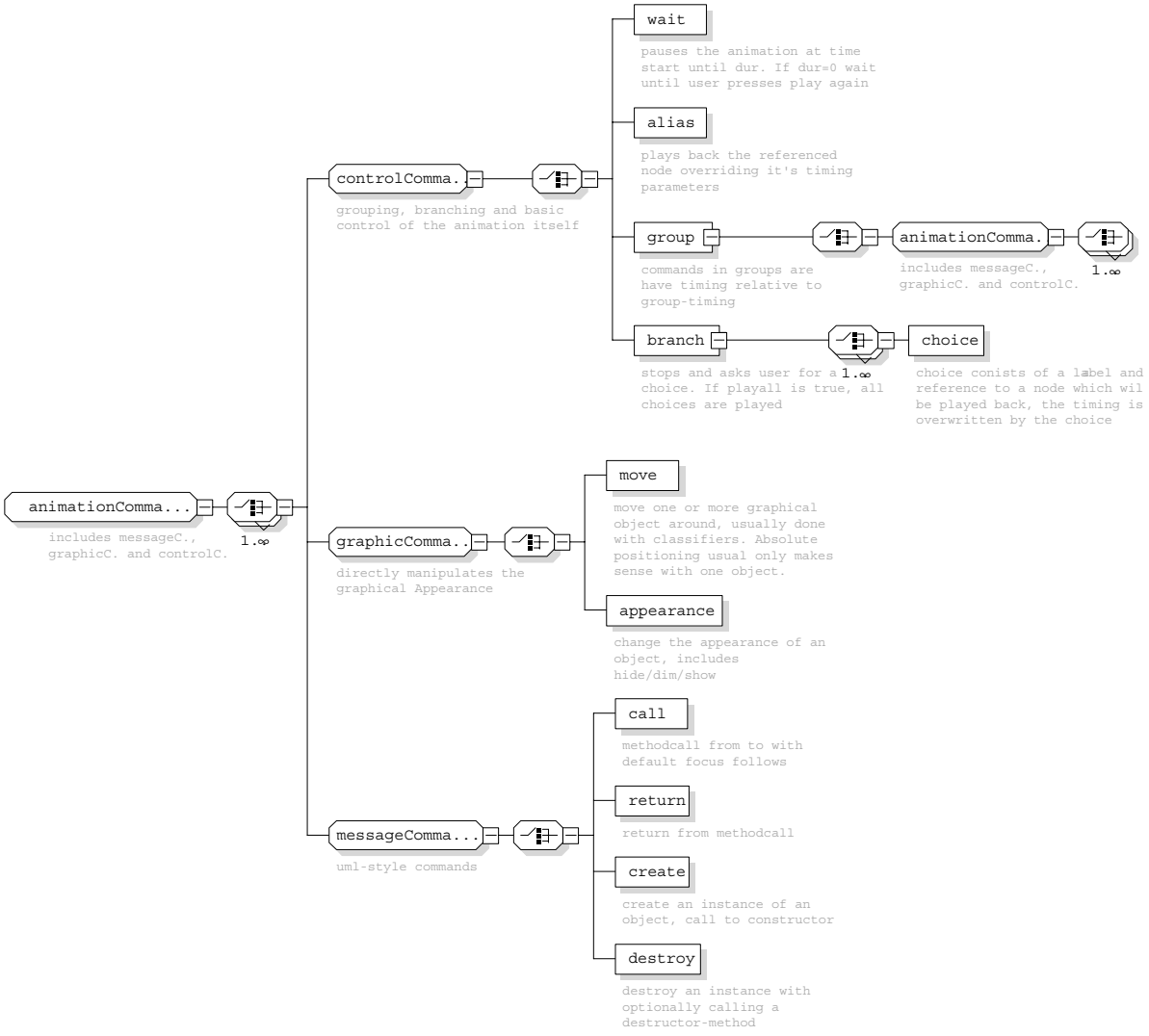

Abbildung 6.2: Knotenaufbau der Animationsdatei

Kapitel 7

Entwurf eines Prototyps

Dieses Kapitel orientiert sich an dem Aufbau eines Pflichtenheftes. Am Anfang steht eine Anforderungsspezifikation. Diese besteht aus den Zielbestimmungen, der Produktumgebung und einer Sammlung von Use-Cases. Danach folgt eine Releaseplanung und schließlich der eigentliche Entwurf.

Alle Teile enthalten Markierungen in der Form `[Buchstabe Ziffer Ziffer]`. Diese dienen zur späteren Referenzierung im Entwurf und bei der Programmierung, um zu dokumentieren, wie sich welche Bedingungen im Sourcecode wiederfinden.

7.1 Anforderungsspezifikation

7.1.1 Spezifikation und Zielsetzung

Gemäß der Beschreibung in Kapitel 3 werden zwei Werkzeuge entwickelt.

- Das erste Werkzeug (Konvertierwerkzeug) nimmt die Konvertierung der XML-Eingabedatei (die ein oder mehrere UML-Klassendiagramme beschreibt) in eine SVG-Datei vor.
- Ein weiteres Tool dient zur Animation dieses Diagramms (Animationswerkzeug). Die Elemente des Diagramms werden dabei dynamisch verändert und weitere Elemente werden hinzugefügt.

Dieser Abschnitt beschreibt die Kriterien, die die beiden Werkzeuge erfüllen müssen. Es werden dabei folgende vier Kategorien unterschieden.

Mußkriterien M Diese Eigenschaften müssen auf jeden Fall erfüllt sein.

Wunschkriterien W Diese optionalen Eigenschaften müssen nicht unbedingt erfüllt sein.

Abgrenzungskriterien A Merkmale, die außerhalb des Aufgabenbereiches.

Erweiterungsfähigkeiten E Hier werden Wünsche an die Erweiterungsfähigkeit gestellt, die als Ansätze für zukünftige Verbesserungen zu betrachten sind.

Konvertierungswerkzeug

[M01]	Eingabe als XML-Datei	Die Eingabe des Konvertierungswerkzeuges entspricht der *ANUCLAD*-Spezifikation für UML-Diagramme. Die entsprechende Dateiendung lautet AUD (*ANUCLAD*-UML-Diagram). Bei dem Entwurf der Lesekomponente sollte darauf geachtet werden, sie als separaten Teil zu implementieren, um Anpassungen an andere Formate leicht vornehmen zu können.
[M02]	Ausgabe als SVG-Datei	Die Ausgabe ist eine statische SVG-Datei, d.h. sie enthält keine Skripte und keine Animation. Hier sollte ebenfalls einen saubere Modularisierung vorgenommen werden, um nachträglich andere Formate einfach unterstützen zu können.
[M03]	Darzustellende Elemente	Das Diagramm soll alle enthaltenen Klassifizierer mit ihren Eigenschaften darstellen, Verbindungen (Assoziationen etc.) zwischen Klassen müssen ebenfalls enthalten sein. Genauere Informationen siehe 7.1.2
[M04]	Eindeutige Elementidentifikationen	Die graphischen Elemente müssen eindeutig identifizierbar sein, um eine spätere Animation zu ermöglichen.
[M05]	UML-konformes Aussehen	Die graph. Elemente müssen den Richtlinien von UML-Klassendiagrammen entsprechen.
[M06]	automatische Größenberechnung	Die Größe der Klassifizierer sollte automatisch berechnet werden.
[M07]	Konfigurierbarkeit	Das Aussehen der Elemente sollte leicht konfigurierbar sein (Farbe, Schriftart, etc.)
[W01]	einfaches Autolayout	Sind keine Koordinatenangaben vorhanden, wäre eine einfache Platzierung der Elemente wünschenswert.
[W02]	zusätzliche graph. Elemente	AUD-Dateien enthalten zusätzliche Informationen wie URLs, Notizen, etc. Diese sollten ebenfalls in das Diagramm eingeblendet werden können, bzw. Querverweise zu anderen Dokumenten sollten funktionieren.
[W03]	Gruppierung der Elemente	Graphisch zusammengehörige Elemente sollten nach Möglichkeit auch in der SVG-Datei gruppiert werden, um die Weiterverarbeitung (beispielsweise die Animation) zu vereinfachen.
[E01]	Andere UML-Diagrammtypen	Die Erweiterung auf andere UML-Diagrammtypen sollte möglich sein.
[E02]	andere Ein- und Ausgabeformate	Neben XML/AUD und SVG sollten auch andere Formate integriert werden können.
[A01]	kein vollständiges Autolayout	Es wird davon ausgegangen, daß die AUD-Eingabedatei bereits Koordinatenangaben enthält. Notfalls kann ein einfaches Autolayout diese berechnen (siehe W01), jedoch ist kein vollständiger Autolayouter vorgesehen, da diese sehr aufwändig sind.

Animationswerkzeug

[M01]	Eingabe	Die Eingabe besteht aus einer SVG-Datei und einer XML-Datei, die der *ANUCLAD*-Spezifikation für UML-Klassendiagrammanimationen (*.AUA) entspricht.
[M02]	Basissteuerungsmöglichkeiten	Auswahl einer Animation, Start, Pause, Reset
[M03]	Zoom und Fokus	Bei der Wiedergabe wird der Focus und Zoom so angepaßt, daß immer der gerade aktiv Bereich sichtbar ist
[W01]	weitere Steuerungsmöglichkeiten	Anwahl von vordefinierten Stellen (Keypoints), Vor- und Zurückspulen
[W02]	einstellbare Granularität	AUA-Dateien können mehrere Detailstufen für die Animation erhalten, die man bei während dem Ablauf der Animation verändern können sollte.
[W03]	Ablaufgeschwindigkeit	Die Geschwindigkeit der Animation sollte veränderbar sein, ein Step-by-Step-Modus sollte vorhanden sein.
[W04]	Wizard für Animationen	Durch sukzessives Anwählen von Methoden können einfache Abläufe generiert und danach animiert werden.
[E01]	Entkoppelung des Applets	Es wäre denkbar die SVG-Darstellung in das Applet zu integrieren und somit eine Standalone-Anwendung zu schaffen. Deshalb sollte die Anbindung an den Browser und die Kommunikation mit dem Plugin in einem separaten Paket untergebracht sein.
[E02]	Animation andere Diagramme	*ANUCLAD* beschäftigt sich nur mit Klassendiagrammen. Sofern möglich sollte das Programm jedoch für andere Diagramme (auch nicht-UML) erweiterbar sein.
[A01]	kein UML-Editor	Dieses Werkzeug ist explizit nicht zum entwerfen von UML-Diagrammen gedacht, es arbeit nur auf fertigen Diagrammen.

7.1.2 Graphische Elemente

Entsprechend der UML-Spezifikation bestehen UML-Diagramme aus *Klassifizierern* (classifier) die in der Ausprägung als Klassen, Interfaces, Schablonen oder Datentypen vorhanden sind. Sie werden durch rechteckige Kästen dargestellt, in denen sich nähere Beschreibungen der Klassifizierer wie Name, Attribute und Operationen befinden. Die Klassen können in *Pakete* gruppiert werden, diese werden durch karteikartenähnliche Elemente dargestellt.

Neben den Klassifizierern existieren noch verschiedene Verbindungen zwischen diesen. Dazu gehört die *Assoziation*, die die einfachste Form der Navigierbarkeit darstellt. Sie zeigt an, ob Objekte Zugriff aufeinander haben. Durch Vielfachheiten an den Klassenenden wird eine Assoziation genauer spezifiziert. Dargestellt wird sie durch einen soliden Pfad zwischen den Objekten. Eine etwas stärkere Variante ist die *Aggregation*, die mit einem hohlem Diamant auf Aggregatseite dargestellt wird (Besitzer). Ergänzend können hier auch Pfeile verwendet werden, um die Beziehung deutlicher zu machen. Noch stärker ist die *Komposition*, wo eine gefüllte Raute verwendet wird. Hier wird dargestellt, daß der Aggregat alleiniger Besitzer des Objektes ist.

Die *Verbindung* (Link) hat prinzipiell die selbe Funktion wie die Assoziation, wird jedoch bei Instanzen verwendet.

Die *Verallgemeinerung* (oder auch Vererbung genannt, englisch Generalization) wird als Pfad von Kind zu Vater mit hohlem Dreieck am Ende dargestellt.

Desweiteren gibt es noch die *Abhängigkeit*, die nicht nur zwischen Klassifizierern sondern auch direkt zwischen den darin enthaltenen Attributen und Objekten benutzt werden kann. Sie zeigt an, ob eine Berechnungsabhängigkeit zwischen den Objekten besteht. Dargestellt wird sie durch einen gestrichelten Pfad.

Alle diese Verbindungen können mit *Stereotypen* und anderen UML-spezifischen Konstrukten versehen werden, wobei jedoch nur der hier beschriebene Teil durch die *ANUCLAD*-Spezifikation unterstützt wird.

Zusätzliche graphische Elemente sind die *Notizen*, die an beliebige Elemente des Diagramms angehängt werden können. Sie enthalten zusätzliche, erläuternde Informationen, die die Semantik des Diagramms nicht ändern.

7.1.3 Systemumgebung

Konvertierungswerkzeug

Das Konvertierungswerkzeug wird in UML entworfen und in Java implementiert. Dabei wird zum Kompilieren und Ausführen das Java Development Kit von Sun in der Version 1.3 oder neuer vorausgesetzt. Java-Virtual-Machines anderer Hersteller können ebenfalls verwendet werden, jedoch wird hier keine Funktionsgarantie gegeben. Das System wird auf einer Windows-Plattform entworfen und auch dort getestet, die Funktionsfähigkeit auf anderen Windows-Plattformen dürfte dadurch gewährleistet sein, solange die selbe Java-VM eingesetzt wird. Sofern eine Java-VM vorhanden ist, sind auch andere Betriebssysteme möglich. Aus Performanzgründen wird ein Rechner mit mindestens 400 Mhz empfohlen und eine graphische Darstellung von mindestens $1024 \cdot 768$ mit 16-Bit Farbtiefe vorausge-

setzt.

Zur Darstellung der erzeugten Diagramme wird ein Darstellungswerkzeug benötigt, welches SVG gemäßt der Spezifikation 1.0 [13] verarbeitet. Cascading-Style-Sheets müssen ebenfalls unterstützt werden. Für die spätere Animation wird außerdem die Unterstützung des Event-Modells von SVG und die Verarbeitungsmöglichkeit von Animationsknoten vorausgesetzt. Es empfiehlt sich das SVG-Plugin von Adobe[1] zu verwenden, da dieses bei der Entwicklung verwendet wurde.

Animationswerkzeug

Die im vorherigen Abschnitt gemachten Angaben im Bezug auf Entwicklungs- und Anwendungsplattform treffen hier ebenfalls zu, jedoch wird die Software als Java-Applet in einem WWW-Browser ablaufen. In gängigen Browsern wird bisher nur Java 1.1.5 unterstützt, weshalb diese Version hier vorausgesetzt wird. Als Browser wird ein Netscape-Browser der vierten Generation (4.xx) unter Windows[2] verwendet. Das Werkzeug wird auf dieser Konfiguration getestet, durch die Verwendung von Java ist prinzipiell jedoch auch eine Verwendung unter anderen Browsern und Betriebssystemen möglich. Zur Darstellung wird ein SVG-Plugin benötigt, welches CSS und Javascript sowie Animationen unterstützt. Außerdem muß eine Kommunikation von Java-Applets mit Javascript und dem Plugin möglich sein[3]. Zur Zeit existiert nur ein Plugin des Herstellers Adobe, welches alle Voraussetzungen erfüllt.

7.1.4 Zielgruppe und Anwendungsbereich

Die beiden vorgestellten Werkzeuge richten sich vor allem an Softwareentwicklungsteams. Durch den großen Anwendungsbereich von UML sind sie jedoch auch für andere Zielgruppen geeignet.

Ein typischer Anwendungsbereich wäre die Erstellung von Software. Hier kommt es oft vor, daß anderen Teammitgliedern die Funktionalität und der Aufbau von Softwareteilen erklärt werden muß. Hier leisten UML-Diagramme gute Dienste, jedoch bietet es sich an, diese Erklärung mit Animationen zu unterstützen, um beispielsweise die Funktionsweise näher zu erläutern. Durch die Animationstechnik von *ANUCLAD* an Hand der UML-Klassendiagramme wird dies ermöglicht.

Ein anderer Anwendungsbereich ist die Spezifikations- und Entwurfsphase. Durch genauere Umschreibung der geplanten Softwareteile können Mißverständnisse früher aufgedeckt und beseitigt werden. Nicht zuletzt lockert die Animation die zumeist recht trockene Präsentation eines Entwurfs etwas auf.

---

[1] `http://www.adobe.com/svg/viewer/install/`

[2] Download unter `http://home.netscape.com/download/0402101/10000-en-win32---_qual.html`

[3] Netscape bezeichnet dies als „Liveconnect“, siehe [24]

7.2 Anwendungsfälle (Use-Cases)

Da die Anzahl der Anwendungsfälle beschränkt ist und keine Vererbungs- oder Erweiterungskonstrukte benötigt werden, wird hier auf eine graphische Repräsentation verzichtet und eine rein textuelle Beschreibung gegeben. Nur die notwendigen Use-Cases werden hier beschrieben, ein Teil der Wunschkriterien wird hiermit nicht abgedeckt.

Zuerst werden die Anwendungsfälle vorgestellt, die nicht direkt mit den beiden Werkzeugen realisiert werden.

7.2.1 Anwendungsfälle außerhalb des spezifizierten Systems

UML-Diagrammerstellung Mit Hilfe eines UML-Tools wird eine Software entworfen, dabei entstehen ein oder mehrere UML-Klassendiagramme, die später im Sourcecode mit Leben gefüllt werden.

UML-Diagrammgenerierung Alternativ zum Entwurf bieten einige Tools auch die Möglichkeit der automatischen Generierung von UML-Klassendiagrammen aus dem Sourcecode.

Konvertierung zu AUD Bisher unterstützt kein Tool das hier entworfene AUD-Format, jedoch bieten viele Werkzeuge den Export als XMI-Datei. Es ist geplant, ein Konvertierwerkzeug von XMI nach AUD anzubieten oder alternativ direkten XMI-Import im Konvertierwerkzeug zu ergänzen. Ebenfalls denkbar wäre ein geeigneter SVG-Export direkt aus den UML-Werkzeugen.

Direkte AUD-Generierung Werkzeuge zur automatischen Generierung von AUD-Dateien direkt aus dem Sourcecode sind denkbar aber bisher nicht geplant.

Generierung der AUA-Datei Zur Erzeugung von UML-Diagrammen sind eine Vielzahl von Werkzeugen vorhanden, jedoch existiert im Bereich der Algorithmenanimation kein Standard zur Beschreibung von Abläufen in Objektorientierten Systemen. Da das System hauptsächlich zur Erklärung von Grundfunktionalitäten gedacht ist, sind im allgemeinen keine komplexen Abläufe notwendig, welche folglich im Handbetrieb erstellt werden können. Werkzeuge zur Instrumentierung von Sourcecode wären hier sicherlich der nächste Schritt, hierzu sind bestimmt Ansätze wie Aspekt- und Subjekt-Orientiertes Programmieren sinnvoll.

7.2.2 Anwendungsfälle des Konvertierwerkzeuges

Ein typischer Konvertiervorgang könnte wie folgt ablaufen:

[U01] Auswahl der AUD-Quelldatei (diese enthält bereits eine Vielzahl der Parameter zur Erzeugung des Diagramms).

[U02] Auswahl einer Darstellungsvariante (u.U. ebenfalls durch eine Datei repräsentiert).

[U03] Eine entsprechende SVG-Datei (falls nötig mit einer HTML-Datei) wird erzeugt.

[U04] Die Datei kann mit einem SVG-fähigen Programm angezeigt werden.

Aufgrund der einfachen Struktur bietet sich ein Kommandozeilen-basiertes Werkzeug an, welches bei vorhandener graphischer Oberfläche Dateiauswahldialoge anzeigt. Eine direkte Anzeige der erzeugten SVG-Datei ist nicht vorgesehen.

7.2.3 Anwendungsfälle des Animationswerkzeuges

[U05] Auswahl der Dateien Der Benutzer wählt eine SVG-Datei und eine AUA-Datei aus.

[U06] Auswahl der Animation Sofern die AUA-Datei mehrere Abläufe enthält, wählt der Benutzer einen aus.

[U07] Start der Animation Die Animation wird mit dem in der AUA-Datei vorgegebenen Tempo wiedergegeben.

[U08] Pause der Animation Der gerade ablaufende Schritt wird beendet und die Animation angehalten, der Benutzer kann den Bildausschnitt und Zoomfaktor frei mit den durch SVG bereitgestellten Werkzeugen ändern.

[U09] Anwahl von Keypoints AUA-Dateien können markierte Stellen enthalten, durch Anwahl einer Stelle wird die gerade aktive Animation gestoppt, der Urzustand hergestellt und bis zu dieser Stelle “vorgespult”.

[U10] Ablaufgeschwindigkeit Der Benutzer kann während der Animation die Geschwindigkeit ändern oder in einen Step-by-Step-Modus schalten.

7.3 Der Entwurfsprozeß

7.3.1 Zeit- und Releaseplanung

Es sind drei Releases geplant, die ungefähr folgenden Funktionsumfang haben werden:

[R01] Prototyp des Konvertierwerkzeuges Der erste Prototyp dient nur zur Konvertierung der UML-Klassendiagrammbeschreibung in eine SVG-Datei. Animationen werden nicht berücksichtigt. Die darzustellende Elemente wurden auf ein Minimum reduziert, beispielsweise ist die Darstellung von Packages nicht möglich und Relationen werden nur durch einfache Linien ohne Pfeilspitzen und Kardinalitäten dargestellt.

[R02] Prototyp: kombiniertes Animations- und Konvertierwerkzeug Nur eine Teilmenge der Diagramm-Spezifikation wird unterstützt. Ein Konvertierwerkzeug (welches in Java programmiert ist) erstellt aus der Diagramm- und einer Animationsdatei eine SVG-Datei, die sowohl das Diagramm selbst, als auch die Animationen enthält. Diese Datei kann mit einem Browser nebst SVG-Plugin angezeigt werden, die Animation kann gestartet werden und läuft dann ohne weitere Kontrollmöglichkeiten ab.

[R03] Konvertierwerkzeug und Animationsapplet Das Konvertiertool enthält alle Mußkritierien exklusive der freien Konfigurierbarkeit des Aussehens ([M07]). Das Animationswerkzeug enthält alle Mußkriterien.

7.3.2 Prototyp des Konvertierwerkzeuges

Wie in der Releaseplanung ausgeführt besteht dieser Prototyp nur aus einem Kommandozeilenwerkzeug, welches aus einer AUD-Eingabedatei eine SVG-Datei generiert. Bei dem Entwurf der Klassenstruktur wurde versucht, bereits eine ausreichende Modularisierung einzuführen, so daß bei einer Weiterentwicklung keine umfassenden Änderungen mehr daran vorgenommen werden müssen.

Um dem Merkmal [**M01**] Rechnung zu tragen, wurde eine separate Klasse zum Einlesen der AUD-Datei entworfen (`UmlDiagramReader`). Sie entkoppelt die Datenstruktur der AUD-Datei von der Objektstruktur des Konvertierwerkzeuges. Nachdem der Einleseprozeß beendet ist wird durch die Klasse `SvgWriter(UmlObject)` eine SVG-Datei erzeugt ([**M02**]).

Neben den Hilfsklassen besteht das Konvertierwerkzeug aus Objekten, die die Elemente des Klassendiagramms modellieren. Beispielsweise gibt es die Klasse `Classifier` die einen Klassifizierer modelliert. Wieder lassen sich gemeinsame Eigenschaften dieser Objekte finden, deshalb bietet es sich an, eine Klassenhierarchie mit Vererbung aufzubauen. Diese beginnt mit der abstrakten Klasse `UmlObject`, die eine Teilmenge der durch `objectType` (siehe Seite 40) definierten Attribute beinhaltet. Diese sind `name`, `id`, `visibility`, `stereotype` und `hidden`. Um diese Klasse vernünftig in Baumstrukturen einsetzen zu können, definiert und implementiert sie Operationen für Rückwärtsverweise (`UmlObject parent`, `UmlObject getParent()`, `setParent(UmlObject)`). Die abstrakte Methode `createSvgRepresentation()` muß von den nichtabstrakten Erben implementiert werden. Sie erzeugt eine graphische Repräsentation des jeweiligen Objektes und liefert diese als `DocumentFragment`[4] zurück. Da jedes Objekt in dem zu erstellenden Diagramm eine gewisse Größe hat, wurden die beiden abstrakten Methoden `getWidth()` und `getHeight()` hinzugefügt, die zur späteren Berechnung der Gesamtabmessungen dienen.

Die Klasse `UmlObject` hat die Unterklassen `Rectangle`, `TextBlock` und `Relation`.

`Relation` dient zur Darstellung der Relationen innerhalb eines Diagrams. Sie liefert zusätzlich das Attribut `to`, welches das Zielobjekt der Relation bestimmt. Das Quellobjekt ist bereits durch `parent` definiert. Die Kardinalitäten und spezielle Assoziation werden im Prototyp noch nicht unterstützt.

Die abstrakte Klasse `TextBlock` fügt `UmlObject` die beiden Attribute `isStatic` und `isInherited` hinzu. Außerdem enthält sie Implementierungen von `getWidth()` und `getHeight()`. `TextBlock` wird von zwei Klassen implementiert. Eine ist für Operationen, die andere für Attribute zuständig. `Attribute` erweitert `TextBlock` mit dem String `type`, der den Typ dieses Attributes angibt. `Operation` beinhaltet zusätzlich `returntype`, `signature` und `isAbstract`.

Die abstrakte Klasse `Rectangle` modelliert Rechtecke. Hierzu sind die Attribute `x,y,width,height` sowie dazugehörige `set/get`-Methoden vorhanden, welche teilweise abstrakt durch `UmlObject` vorgegeben wurden. Die durch das Rechteck eingeschlossenen Elemente werden in der Hashtabelle `children` referenziert. Als Schlüssel der Hashtabelle wird die `id` verwendet, die bei allen Elementen zwingend vorgeschrieben ist ([**M04**]). Mit der Methode `calculateSize()` wird die Größte des Rechtecks aufgrund der Elemente der Hashtabelle berechnet. Erben dieser Klassse können diese Methode überschreiben und dabei auf die bereits vorhandene Implementierung zurückgreifen. Dies ist unter Umständen notwendig, wenn neben dem Inhalt eines Objektes auch das Objekt

[4] Ein `DocumentFragment` ist ein Teil einer XML-Datei, der wohlgeformt sein muß.

selbst noch zu seinen Abmessungen beiträgt und beispielsweise einen Rand hinzufügt. Ein Flag `sizeInvalid` zeigt an, ob die derzeitige Größe nicht mehr aktuell ist. Dies trägt zur Erfüllung des Merkmals **[M06]** bei. Die Methode `getObjectWithId()` liefert das entsprechende Element der Hashtabelle zurück.

Konkretisiert wird diese Klasse durch drei weitere Klassen:

UmlDiagram dient zur Beschreibung eines ganzen Diagrams. Hierzu werden keine weiteren Attribute benötigt. In der Hashtabelle können Pakete oder Klassifizierer abgelegt werden.

Package beschreibt ein Paket, auch hier sind keine weiteren Attribute zur Beschreibung notwendig. Als Kinder (`children`) kommen hier ebenfalls Pakete und Klassifizierer in Frage. Diese Klasse ist als Klassenstumpf im Prototyp bereits vorhanden, wird jedoch erst in einer späteren Release implementiert.

Classifier sind wieder etwas komplexere Objekte. Die vier Typen sind durch Integer-Konstanten bestimmt, sie werden bei Initialisierung des Objektes dem Feld `type` zugewiesen. Desweiteren existiert das Attribut `isAbstract`.

Damit sind fast alle UML-Konstrukte der AUD-Datei abgedeckt, daß Mußkriterium **[M03]** ist somit bis auf die Darstellung von Paketen erfüllt. Der Prototyp ist jedoch noch nicht voll UML-Konform, da Teile der Spezifikation außer acht gelassen wurden. **[M05]** ist also noch nicht erfüllt.

Ein Klassendiagramm findet sich in Abbildung 7.1. Eine vollständige Dokumentation in Javadoc ist unter [16] verfügbar.

7.3.3 Prototyp des kombinierten Werkzeuges

Auf Basis der im vorherigen Abschnitt vorgestellten Klassenstruktur wird nun ein kombiniertes Werkzeug entworfen, welches neben der Klassenbeschreibung auch eine Animationsbeschreibung einliest und nach Erstellung des Klassendiagramms entsprechende Animationsbefehle hinzufügt. Diese Animationsbefehle werden als sogenannte deklarative SVG-Animationen hinzugefügt. Diese benötigen kein Javascript und basieren direkt auf dem Eventmodell von SVG, wodurch sie bereits in verschiedenen SVG-Implementationen lauffähig sind. Der Sprachumfang der Animationsbeschreibung wurde auf ein Minimum reduziert. Die zeitlichen Parameter werden vollständig ignoriert, es gilt die Reihenfolge der Befehle in der Datei. Querverweise (`alias`) und Verzweigungen (`branch`) werden ebenfalls nicht verarbeitet. Der `appearance`-Befehl versteht nur die Attribute `idrefs` und `visual`, der `move`-Befehl hat keine Wirkung. Die vier `messageCommands` werden vollständig unterstützt.

Die Klassenstruktur des ersten Prototyps muß nun um einige Klassen erweitert werden. Diese kümmern sich um das Einlesen der Animationsdatei und die Erzeugung der nötigen SVG-Informationen sowie zur Speicherung der zusätzlichen Animationsinformationen für die SVG-Repräsentation.

Hierzu wird zuerst die Diagrammbeschreibung eingelesen und die interne Datenstruktur aufgebaut. Sofern notwendig wird ein einfaches Autolayout vorgenommen. Nachdem alle

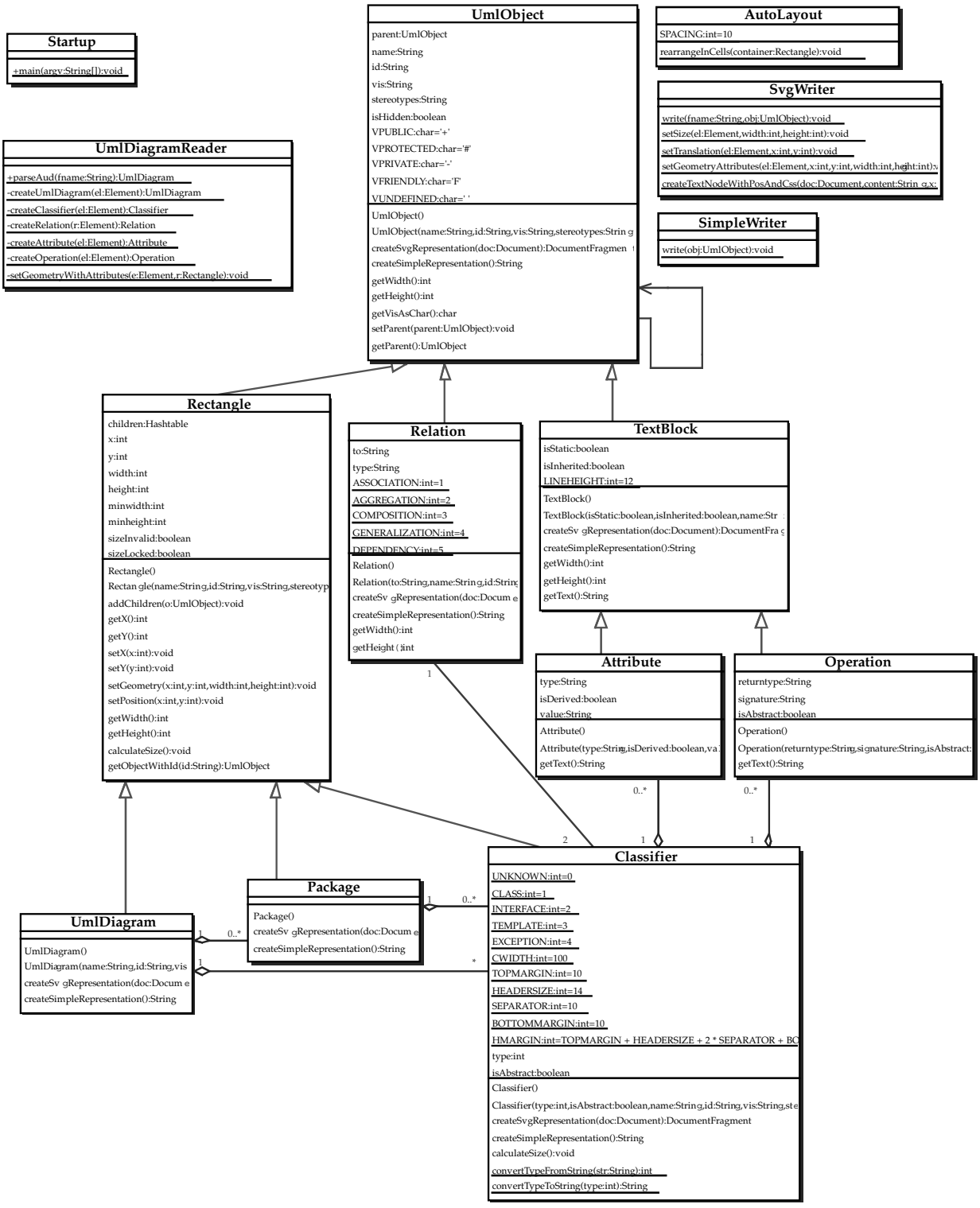

Abbildung 7.1: Klassendiagramm des ersten Prototyps

Diagrammelemente an ihrem Platz sind, werden zuerst die benötigten Kanten zur Darstellung der Methodenaufrufe hinzugefügt. Danach werden die eigentlichen Animationsbefehle generiert.

UmlAnimationCommand enthält alle Informationen um einen Methodenaufruf darzustellen, dazu gehören Anfangszeit und -dauer sowie Verweise auf die entsprechenden Diagrammelemente.

UmlAnimationLine erzeugt die Repreäsentation einer Kante.

UmlAnimationReader kümmert sich um das Einlesen der AUA-Datei.

UmlAnimations ist die Containerklasse für Animationsbefehle und -kanten.

7.3.4 Konvertierwerkzeug und Animationsapplet

Der Entwurf der dritten Release war zum Zeitpunkt der Drucklegung dieser Studienarbeit noch nicht fertiggestellt. Die Dokumentation nebst Sourcecode wird nach Fertigstellung unter `http://www.girschick.net/martin/study/anuclad/` verfügbar sein.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Studienarbeit wurde ein Konzept vorgestellt, welches das Verständnis von komplexen Softwaresystemen verbessern soll. Dieses Konzept basiert auf der weit verbreiteten Modellierungssprache UML, was eine schnelle Einarbeitung ermöglicht.

UML beschreibt Softwaresysteme durch Diagramme. Diese Diagramme werden mit dem Konzept dieser Studienarbeit zum Leben erweckt. Durch Animation von Klassendiagrammen wird das Verständnis der oft komplexen Softwaresysteme unterstützt. Abläufe können besser dargestellt werden, die Übersichtlichkeit wird verbessert.

Die entwickelten Ideen sind jedoch nicht auf Klassendiagramme beschränkt, sie sind auf andere Diagrammtypen der UML erweiterbar und auch die Anwendung in anderen Bereichen, bei denen die Visualisierung durch Diagramme möglich ist, waere denkbar.

Das Konzept wird durch einen Prototyp anschaulich umgesetzt. Durch Verwendung offener Technologien wie Java und XML konnte ein Werkzeug entwickelt werden, welche plattformübergreifend die Visualisierung von Softwaresystemen prototypisch umesetzt.

Diese Visualisierung beschränkt sich bisher auf die stückweise Darstellung von Klassendiagrammen und der einfachen Simulation von Abläufen mit Hilfe dieser Klassendiagramme. Im folgenden Abschnitt soll jedoch auf Erweiterungsmöglichkeiten eingegangen werden.

8.2 Ausblick

Die derzeitige Implementation, stellt nur einen Prototyp dar. Die Entwicklung eines komfortableren Werkzeuges wäre somit wünschenswert. Einige Punkte wurden bereits in Abschnitt 7.1 erwähnt, weitere Ideen sind im folgenden aufgeführt.

- Unterstützung andere UML-Diagrammtypen
- Integration in bestehende Modellierungswerkzeuge
- Unterstützung des XMI-Formats

- Untersuchung der Möglichkeiten zur dreidimensionalen Darstellung von Diagrammen (siehe auch [17])
- Integration eines leistungsfähigen Systems zum automatischen anordnen der Klassenobjekte.

Zur Verbesserung der Animation und des Animationswerkzeuges sind folgenden Entwicklungen denkbar.

- Selektive Ausblendung nicht benötigter Klassen und Methoden.
- Bessere Unterstützung zur Darstellung von Multithreading.
- Werkzeug zur visuellen Komposition von Abläufen (Wie es beispielsweise Elixir Case bietet, siehe 2.2).

Um eine leistungsfähige Simulationsumgebung zu schaffen, die auch zum Debugging geeignet wäre, sollen hier noch ein paar Ideen angeführt werden.

- Durch Instrumentierung des zu testenden Sourcecodes wäre eine direkte Steuerung der Visualisierung denkbar. Man könnte parallel zum Ablauf des Programmes die Methodenaufrufe an dem Klassendiagramm mitverfolgen.
- Diese Instrumentierung sollte automatisierbar sein.
- Eine Darstellung der Abläufe innerhalb einer Methode wäre wünschenswert.

Anhang A

Beispiele

Um die Funktionsweise von *ANUCLAD* zu zeigen, wurde eine kleine Demonstrationsapplikation entwickelt. Der Entwurf entstand mit dem Modellierungswerkzeug Together. Als Programmiersprache wurde Java gewählt. Die Diagramminformation wurde aus den Java-Dateien generiert. Diese wurden dazu mit Kommentaren versehen, die die Positionierung der Elemente genauer spezifizierten.

Aus dem Java-Source wurde mittels Javadoc und dem Doclet Javadox [10] eine XML-Beschreibung generiert, die mit einem selbst entwickelten Konvertierungswerkzeug namens `javadox2umldiagrams` zu einer *ANUCLAD*-Diagrammbeschreibung umgewandelt wurde.

Um die eigentliche Animation zu steuern wird zusätzlich eine *ANUCLAD*-Animationsbeschreibung benötigt. Diese wurde während eines Testlaufes der Demonstrationsapplikation erzeugt. Dazu wurde der Sourcecode mit Logging-Befehlen instrumentiert, die die Methodenaufrufe und -rückgaben protokollierten.

Das entwickelte System modelliert einen Verbund von Getränkeautomaten, die mit einem Verwaltungssystem verbunden sind. Die Automaten melden sich bei dem Verwaltungssystem an, dort kann zentral der Füllstand der Automaten abgefragt werden. An den Automaten selbst können Getränkedosen aufgefüllt und entnommen werden.

Das System besteht aus drei Paketen. Das *server*-Paket kümmert sich um die Verwaltung der Automaten. Die Automaten selbst werden durch das *dispenser*-Paket modelliert. Ein weiteres Paket *test* kümmert sich um das Testen des Systems, es stellt eine Eingabekonsole zur Verfügung, von der die einzelnen Teile gesteuert werden können.

In Abbildung A.1 ist die Java-Sourcecode-Datei einer Klasse dargestellt. Javadoc-spezifische Kommentare sind durch `/** ... **/` eingeschlossen. Die `geometry(...)`-Information wird von `javadox2umldiagrams` ausgewertet. Die `mlog`-Aufrufe protokollieren die Methodenaufrufe, um die Animation zu steuern.

Die mit Javadoc generierte HTML-Dokumentation dieser Klasse ist in Abbildung A.2 zu sehen. Ein Teil der XML-Beschreibung folgt in Abbildung A.3.

Mit Hilfe von `javadox2umldiagrams` wurde aus der XML-Beschreibung eine *ANUCLAD*-Diagrammbeschreibung generiert. Ein Teil dieser ist in Abbildung A.4 zu finden. Die Animationsbeschreibungen sind ausschnittsweise in Abbildung dargestellt.

DispenserConnector.java

```
package canDispenser.server;

import canDispenser.dispenser.ServerConnector;

/** The DispenserConnector is the facade for the server-connection. The DispenserManager
 * creates a new instance of a DispenserConnector when a new dispenser announces itself to the Manager
 * The reference to the DispenserConnector is returned to the Dispenser-Side. The DispenserConnector has a
 * reference to the ServerConnector as well. The communication between the server and the dispenser is
 * handled thru the DispenserConnector.
 *
 * geometry(x=300,y=400)
**/
public class DispenserConnector
{
    private ServerConnector sc = null;
    String name;

    /** This constructor creates a new DispenserConnector. It is called by the DispenserManager.
     * @param sc The reference of the ServerConnector
    **/
    public DispenserConnector(ServerConnector sc)
    {
    Startup.mlog(this, "DispenserConnector","-canDispenser.dispenser.ServerConnector-","stay");
        this.sc = sc;
        name = sc.name;
    }
    /** This method returns information about the Dispenser to which this DispenserConnector
     * is connected to.
     *
     * @return the information about the connected dispenser
    **/
    public String getDispenserInfo()
    {
    Startup.mlog(this, "getDispenserInfo","-","follow");
        String[] array = sc.getContentInformation();
        String str = "Dispenser: " + name + "\n";
        for (int i=0;i<array.length;i++) str+=array[i] + "\n";
    Startup.mlog(this, "getDispenserInfo","-","return");
        return str;
    }
}
```

Abbildung A.1: Java-Sourcecode der Klasse DispenserConnector

Aus diesen beiden Eingabeteilen wurde mit dem Prototyp von *ANUCLAD* eine SVG-Datei erzeugt. Das Ergebnis davon wurde mit dem Adobe-SVGViewer dargestellt (Abbildung A.6). In den Abbildungen A.7, A.8 und A.9 sind drei Animationsschritte dargestellt.

Erzeugung eines neuen Automaten Der Benutzer will einen neuen Automaten an das System anmelden. Durch Einschalten wird auf der Client-Seite (obere Hälfte der Klassen) über die `ServerConnector`-Facade der `DispenserManager` benachrichtigt.

Nachfüllen des Automaten Auf der Client-Seite wird die `refill`-Methode aufgerufen, über Selbstdelegation wird dann mittels `createItem` die einzelnen Repräsentanten der Getränke erzeugt.

Entnahme einer Getränkedose Der `ContentManager` durchsucht den Bestand nach einem Getränk, welches den selben Typ, wie der vom Benutzer gewünschte hat.

Da Animation nur schlecht durch diese statischen Bilder dargestellt werden kann, empfiehlt es sich die Originaldatei unter `http://www.girschick.net/martin/study/anuclad/` mit einem SVG-fähigen Browser anzuschauen.

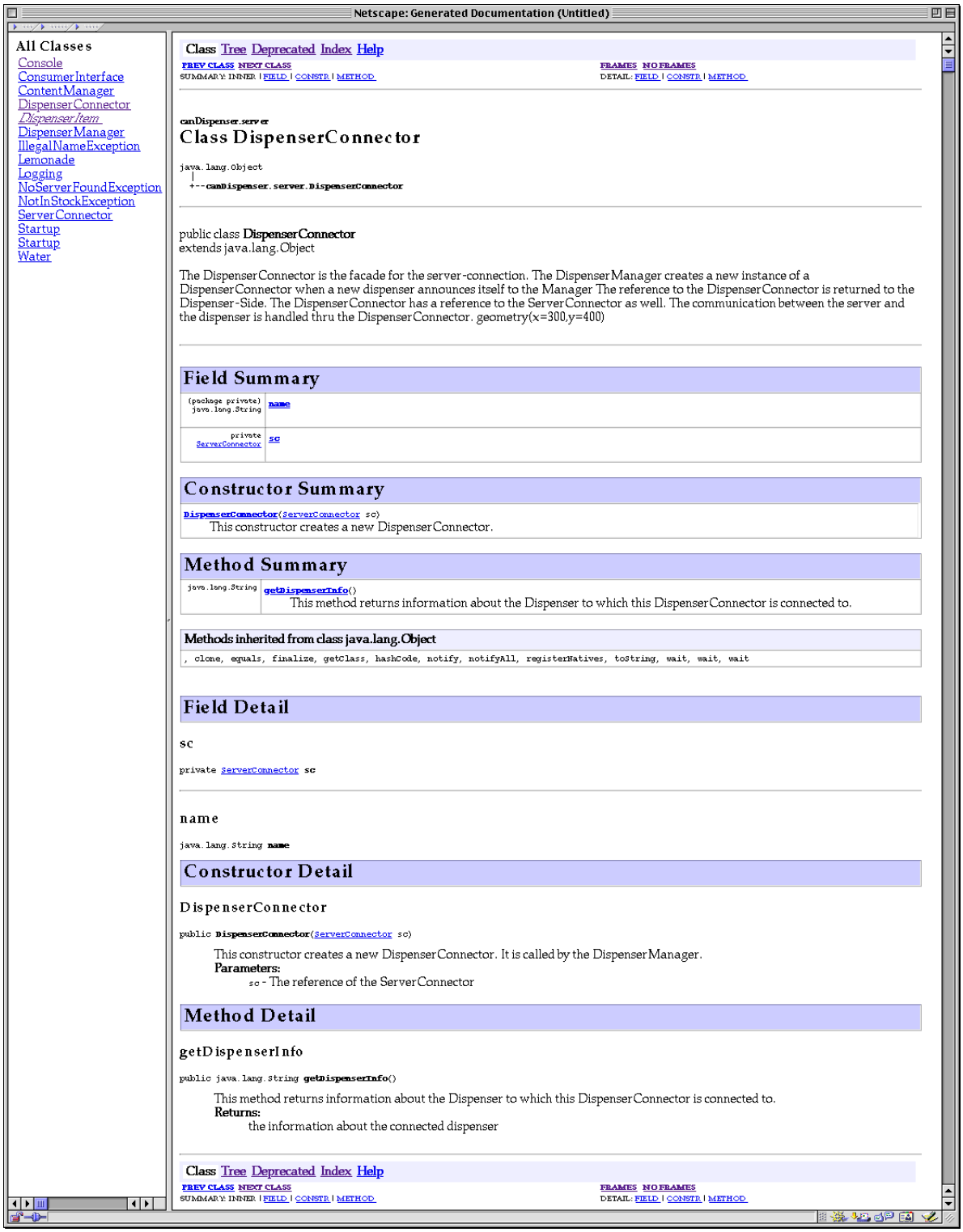

Abbildung A.2: Javadoc-Dokumentation der Klasse DispenserConnector

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
OFFICIAL COPYRIGHT NOTICE.  DO NOT REMOVE
Created by JavaDox
JavaDox Copyright 1999, Flashline.com, Inc., All Rights Reserved.
May be freely distributed.
Java, Javadoc, and JDK 1.2 are a registered trademark of Sun Microsystems, Inc.
Additional information, XSL, and public domain DTD files are available
at www.componentregistry.com/javadox
Version 1.0 (Build 31)
-->
<model version="0.8">
...
<package>
<packageName>canDispenser.server</packageName>
<classes>
  <class accessSpecifier="public" classType="class">
    <className>DispenserConnector</className>
    <classQualifiedName>canDispenser.server.DispenserConnector</classQualifiedName>
    <description>
      <briefDescription>
        <![CDATA[The DispenserConnector is the facade for the server-connection. ]]>
      </briefDescription>
      <fullDescription>
        <![CDATA[The DispenserConnector...]]></fullDescription>
    </description>
    <superClass>java.lang.Object</superClass>
    <constructor accessSpecifier="public">
      <constructorName>DispenserConnector</constructorName>
      <signature>(canDispenser.dispenser.ServerConnector sc)</signature>
      <description>
        <briefDescription><![CDATA[This constructor creates a new DispenserConnector. ]]>
        </briefDescription>
        <fullDescription><![CDATA[This constructor creates a new DispenserConnector.
        It is called by the DispenserManager.]]></fullDescription>
      </description>
      <parameter>
        <parameterType>canDispenser.dispenser.ServerConnector</parameterType>
        <parameterName>sc</parameterName>
        <description>
          <briefDescription><![CDATA[sc The reference of the ServerConnector]]></briefDescription>
          <fullDescription><![CDATA[sc The reference of the ServerConnector]]></fullDescription>
        </description>
      </parameter>
    </constructor>
    ...
</class>
</package>
...
</model>
```

Abbildung A.3: Javadoc-Ausgabe des Javadox-Doclets

```
<?xml version="1.0" encoding="UTF-8"?>
<umldiagrams>
  <classdiagram name="canDispenser_javadoc.xml.aud" id="canDispenser_javadoc.xml.aud">
    <class name="DispenserConnector" id="canDispenser.server.DispenserConnector" visibility="public">
      <annotation>
        <![CDATA[The DispenserConnector is the facade for the server-connection.
        geometry(x=300,y=400)]]>
      </annotation>
      <geometry x="300" y="400" />
      <attributes>
        <attribute name="sc" id="canDispenser.server.DispenserConnector.sc"
          visibility="private" type="ServerConnector" />
        <attribute name="name" id="canDispenser.server.DispenserConnector.name"
          visibility="friendly" type="String" />
      </attributes>
      <operations>
        <operation name="DispenserConnector" id="canDispenser.server...dispenser.ServerConnector-"
        visibility="public" signature="(canDispenser.dispenser.ServerConnector sc)" />
        <operation name="getDispenserInfo"
        id="canDispenser.server.DispenserConnector.getDispenserInfo-"
        visibility="public" signature="()" returntype="java.lang.String" />
      </operations>
    </class>
  </classdiagram>
</umldiagrams>
```

Abbildung A.4: Ausschnitt der *ANUCLAD*-Diagrammbeschreibung

```
<?xml version="1.0" encoding="UTF-8"?>
<animations>
  <animation name="canDispenserDemo">
...
    <call
   from="...DispenserManager.addDispenser-canDispenser.dispenser.ServerConnector-"
 to="...DispenserConnector.DispenserConnector-canDispenser.dispenser.ServerConnector-"
  focus="stay" begin="9" dur="1"
description="DispenserConnector"/>
    <void
   from="...DispenserConnector.DispenserConnector-canDispenser.dispenser.ServerConnector-"
 to="...DispenserManager.addDispenser-canDispenser.dispenser.ServerConnector-"
  focus="follow" begin="10" dur="1"
description="DispenserConnector"/>
    <return
   from="...DispenserManager.addDispenser-canDispenser.dispenser.ServerConnector-"
 to="...ServerConnector.ServerConnector-java.lang.String-"
  focus="follow" begin="11" dur="1"
description="addDispenser"/>
    <void
   from="...ServerConnector.ServerConnector-java.lang.String-"
 to="...Startup.getNewDispenser-java.lang.String-"
  focus="follow" begin="12" dur="1"
description="ServerConnector"/>
...
  </animation>
</animations>
```

Abbildung A.5: Ausschnitt der *ANUCLAD*-Animationsbeschreibung

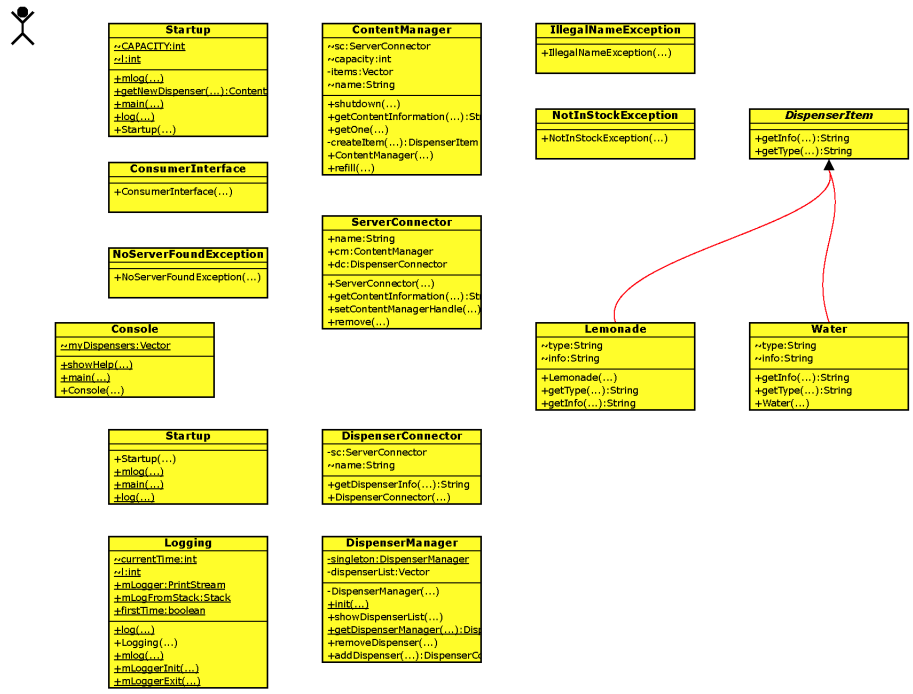

Abbildung A.6: Klassendiagramm von canDispenser, erzeugt mit *ANUCLAD*-lite

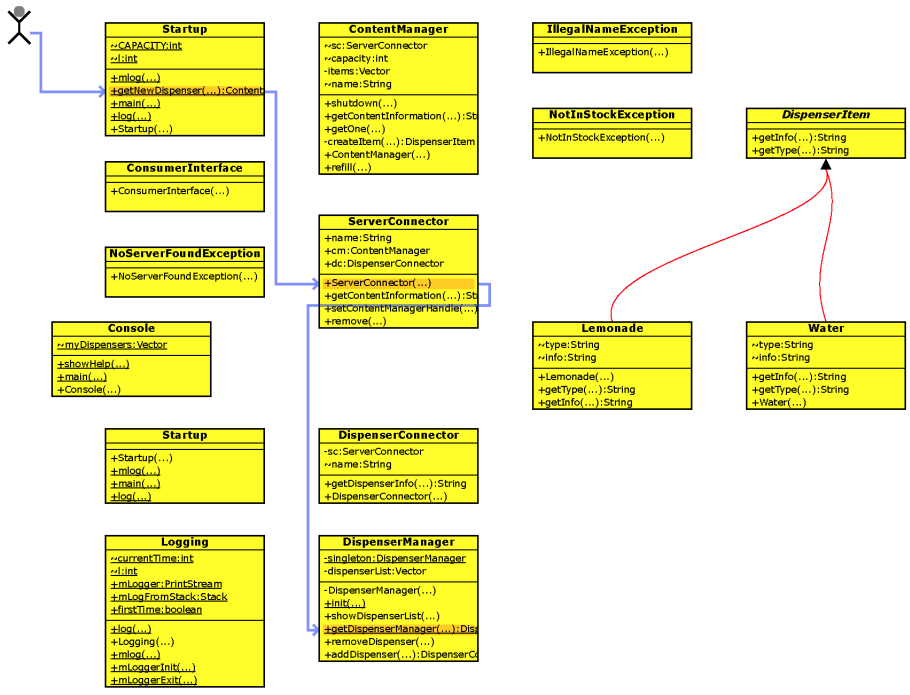

Abbildung A.7: Animation: Erzeugung eines neuen Automaten (Dispenser)

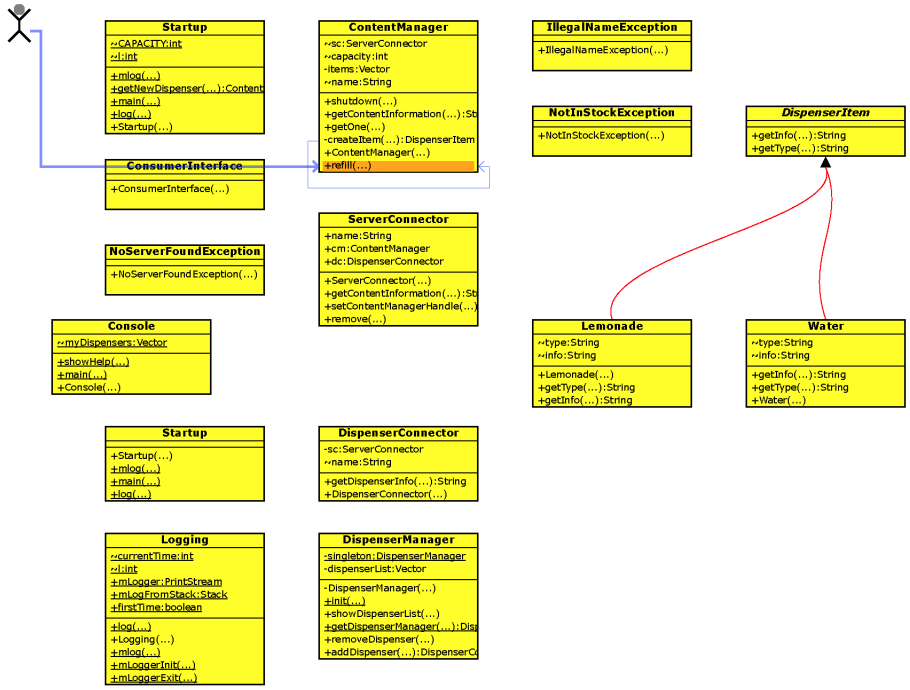

Abbildung A.8: Animation: Nachfüllen des Automaten

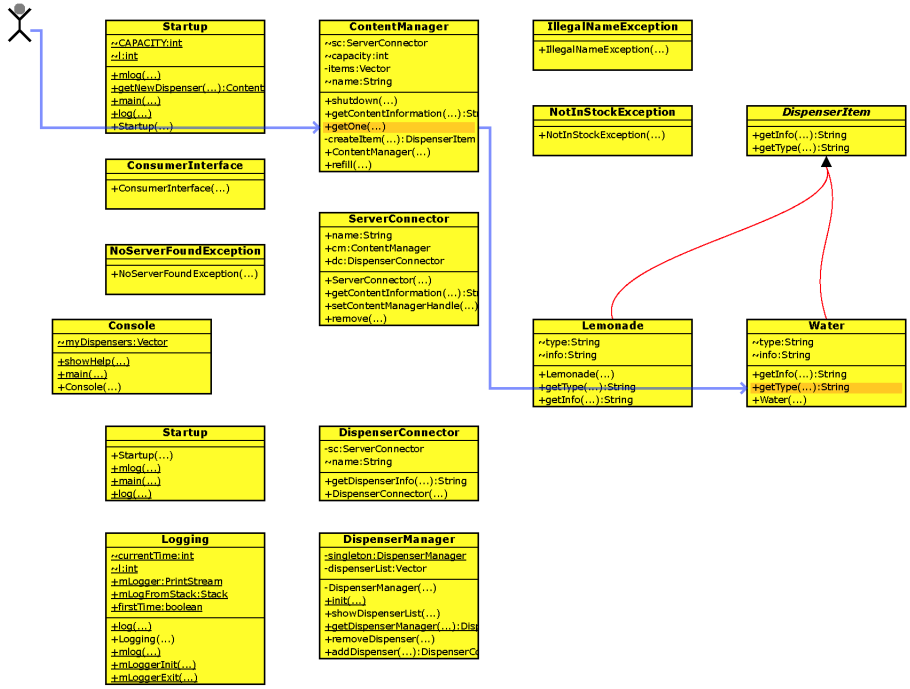

Abbildung A.9: Animation: Entnahme einer Getränkedose

Anhang B

Glossar

Aggregation UML. Eine Aggregationbeschreibt eine „enthalten in“-Verbindung. In der Implementation wird dies meist durch eine Referenz innerhalb des Aggregaten auf das aggregierte Objekt dargestellt.

API Application Programming Interface. Festgelegte Programmierschnittstelle, beispielsweise einer Bibliothek.

Assoziationen UML. Die Assoziation ist in der UML die einfachste Form einer Beziehung zwischen Objekten. Sie macht keine weiteren Aussagen über ihre Implementation.

Cascading-Style-Sheets Zur Formatierung von HTML- und XML-Dokumenten entwickelte Sprache. Durch Definition von Stilklassen kann das Aussehen von Objekten bestimmt werden, diese Klassen werden dann in HTML- oder XML-Dokumenten verwendet. Typische Formatierungsmöglichkeiten sind Schriftart und -form sowie Positionierung.

CASE-Tools Computer Aided Software Engineering. Werkzeuge zur computerunterstützten Softwareentwicklung.

Constraints UML. Um in UML-Diagrammen Einschränkungen zu modellieren, können sowohl umgangssprachliche Konstrukte, als auch die Object-Constraints-Language verwendet werden.

Design-Pattern Hilfsmittel zur strukturierten Entwicklung von Softwaresystemen. Design-Pattern versuchen oft wiederkehrende Problemstellungen und ihre Lösungsansätze möglichst abstrakt zu beschreiben. Diese abstrakten Muster können dann bei Softwareentwürfen eingesetzt werden.

DOM Document Object Model. Eine API zur Verarbeitung von Dokumenten, die eine baumartige Struktur aufweisen. Meist wird hier HTML oder XML verwendet.

DTD Document Type Definition. Beschreibungssprache für SGML- und XML-Dokumente.

ECMAScript Von Netscape entwickelte Skriptsprache (wird dort als Javascript bezeichnet), die in HTML-Dokumente eingebunden werden kann. Wird meist für Abfragen in Formularen oder zur Navigation zwischen HTML-Seiten eingesetzt.

Embedded-Solutions Soft- und Hardwarprodukte, die zur Steuerung von Maschinen eingesetzt werden (beispielsweise Waschmaschinen oder Fahrzeuge).

Exception-Handling Fehlerbehandlungsmodell. Wurde beispielsweise in Java konsequent umgesetzt. Programmteile, die Fehler produzieren können (beispielsweise bei der Datenausgabe) lösen in einer Fehlersituation sogenannte Exceptions aus. Diese enthalten dann Informationen über den Fehler.

Framework Ähnlich einer API definiert ein Framework eine Schnittstelle, jedoch bieten Frameworks zusätzliche die Möglichkeit durch eigene Objekte erweitert zu werden.

Generalisierung Auch als Vererbung bezeichnet. In objektorientierten Programmiersprachen besteht die Möglichkeit neue Funktionalität durch Nutzung bereits bekannter Objekte zu generieren. Hierzu wird die alte Funktionalität von einer Klasse geerbt und durch weitere Funktionalität zu ergänzen.

Hashtabelle In einer Hashtabelle können indizierte Daten abgelegt werden. Jedes Datum hat einen eindeutigen Schlüssel, über den es wieder in der Tabelle gefunden werden kann.

Java Von Sun entwickelte, objektorientierte Programmiersprache. Besonderheiten sind die umfangreiche API sowie die weitreichenden Sicherheitskonzepte die speziell für Netzwerkanwendungen konzeptioniert wurde.

Java-Applets Applets sind meist kleine Programme, die in einem Internet-Browser ablaufen. Sie werden über das Internet geladen und laufen auf dem lokalen Rechner ab. Durch das Sandkasten-Prinzip von Java wird gewährleistet, daß diese Anwendungen nur beschränkten Zugriff auf lokale Resourcen erhalten.

Javadoc Werkzeug zur automatische Generierung einer Schnittstellendokumentation von Java-Klassen.

Klassendiagramm UML. Ein Klassendiagramm beschreibt Objekte eines Softwaresystems sowie deren Vererbungshierarchie.

Klassifizierer UML. Übergreifender Begriff für Klassen, Interfaces und Templates.

Komposition UML. Die Komposition ist eine stärkere Form der Aggregation. Bei Zerstörung des Aggregatobjekts werden alle komponierten Elemente ebenfalls zerstört.

Multithreading Wenn ein Programm mehrere, parallele Kontrollflüsse hat spricht man im allgemeinen von Multithreading.

PDF Portable Document Format. Von Adobe entwickeltes Dateiformat zum Austausch von Dokumenten.

Realtime dt. Echtzeit. Wenn Softwaresysteme gewissen zeitlichen Bedingungen genügen müssen, spricht man von Echtzeitanwendungen.

Relationen UML. engl. Relationships. Überbegriff für Assoziationen und Generalisierungen aber auch allgemeinen Verbindungen sowie Stereotypen für Beziehungen.

SAX Simple API for Xml-parsing. API zur einfachen, streambasierten Verarbeitung von XML-Dokumenten.

Sequenzdiagramme UML. Diagrammform zur Beschreibung von Abläufen in Programmen. Die Granularität liegt meist auf Methodenebene, dargestellt werden Methodenaufrufe zwischen Objekten.

Stereotypen UML. Stereotypen erweitern die Semantik der Objekte eines UML-Diagramms. Sie können beispielsweise Klassen näher spezifizieren, in dem sie sie als Facade oder Framework bezeichnen. Die UML-Spezifikation definiert bereits einige Stereotypen, die Verwendungen eigener ist jedoch ebenfalls möglich.

Use-Cases UML. dt. Anwendungsfälle. Beschreibung der wichtigsten Abläufe, die mit dem zu entwickelnden Softwareprodukt bewältigt werden sollen.

Literaturverzeichnis

[1] ADOBE: *Precision Graphics Markup Language (PGML)*. `http://www.w3.org/TR/1998/NOTE-PGML-19980410.html`, 1998.

[2] BALZERT, HELMUT: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, Heidelberg, 1996.

[3] BANKS, IAIN M.: *Feersum Endjinn*. Orbit, London, 1994.

[4] BEHME, HENNING und STEFAN MINTERT: *XML in der Praxis – Professionelles Web-Publishing mit der Extensible Markup Language*. Addison-Wesley, Bonn, 2000.

[5] BOOCH, GRADY: *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[6] BOOCH, GRADY, JIM RUMBAUGH und IVAR JACOBSON: *The Unified Modelling Language User Guide*. Addison-Wesley, Reading, MA, 1998.

[7] BOX, DON, AARON SKONNARD und JOHN LAM: *Essential XML – Beyond Markup*. Addison-Wesley, Boston, 2000.

[8] BRAY, TIM, JEAN PAOLI, C. M. SPERBERG-MCQUEEN und EVE MALER: *Extensible Markup Language (XML) 1.0*. `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[9] BROWN, M. H.: *Algorithm Animation*. MIT Press, Cambridge, Mass., 1988.

[10] COMPONENTREGISTRY: *Java doclet for XML*. `http://www.componentregistry.com/jdox/javadox.jar`, 1999.

[11] EICHELBERGER, HOLGER: *Automatisches Zeichnen von UML-Klassendiagrammen durch den Sugiyama-Algorithmus*. Tagungsband GI-Workshop Softwarevisualisierung 2000, 2000.

[12] FALLSIDE, DAVID C.: *XML Schema Part 0: Primer - W3C Recommendation, 2 May 2001*. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`, 2000.

[13] FERRAIOLO, JON: *Scalable Vector Graphics (SVG) 1.0 Specification*. `http://www.w3.org/TR/2000/CR-SVG-20001102/`, 2000.

[14] FLANAGAN, DAVID: *Java in a Nutshell – Deutsche Ausgabe für Java 1.0*. O'Reilly, Bonn, 1996.

[15] FOWLER, MARTIN und KENDALL SCOTT: *UML konzentriert*. Addison-Wesley, Bonn, 1998.

[16] GIRSCHICK, MARTIN: *Documentation and Download of the Anuclad Prototype*. `http://www.girschick.net/martin/study/anuclad/anucladLite/`, 2001.

[17] GOGOLLA, MARTIN, OLIVER RADFELDER und MARK RICHTERS: *Towards Three-Dimensional Representation and Animation of UML Diagrams*. Proceedings of UML '99 2nd International Workshop, Fort Collins, 1999.

[18] GOSSENS, MICHAEL, FRANK MITTELBACH und ALEXANDER SAMARIN: *Der LaTeX-Begleiter*. Addison-Wesley, Bonn, 1995.

[19] JACOBSEN, IVAR, MAGNUS CHRISTERSON, PATRIK JONSSON und GUNNAR OEVERGAARD: *Object-Oriented Software Engineering*. Addison-Wesley, Essex, 1993.

[20] KOPKA, HELMUT: *LaTeX Einführung Band 1 2. Auflage*. Addison-Wesley, Bonn, 1996.

[21] LAUGHLIN, BRETT MC: *Java und XML*. O'Reilly, Bonn, 2001.

[22] LI, XUANDONG und JOHAN LILIUS: *Timing Analysis of UML Sequence Diagrams*. Proceedings of UML '99 2nd International Workshop, Fort Collins, 1999.

[23] MEHNER, KATHARINA und ANNIKA WAGNER: *Ablaufvisualisierung für nebenläufige Java-Programme mit UML Interaktionsdiagrammen*. Tagungsband GI-Workshop Softwarevisualisierung 2000, 2000.

[24] MICROSYSTEMS, SUN: *LiveConnecting Plug-ins with Java*. `http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pjava.htm`.

[25] OBJECT MANAGMENT GROUP, INC. und OTHERS: *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. OMG, Framingham, MA, 1999.

[26] OECHSLE, RAINER und DIETER JAHN: *Visualisierung von ausgewählten Lehrinhalten der Informatik - Beispielanimationen aus dem Projekt Vivaldi*. Tagungsband GI-Workshop Softwarevisualisierung 2000, 2000.

[27] RÁCZ, FERENC DÓSA und KAI KOSKIMIES: *Tool-Supported Compression of UML Class Diagrams*. Proceedings of UML '99 2nd International Workshop, Fort Collins, 1999.

[28] RIVARD, NORMAND: *UML-Xchange*. `http://www.cam.org/~nrivard/uml/umlxchng.html`.

[29] RUMBAUGH, JAMES, MICHAEL BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-Oriented Modeling and Design*. Prentice Hall, New York, NY, 1991.

[30] SEEMANN, JOCHEN und JÜRGEN WOLFF VON GUDENBERG: *UMLscript: A Programming Language for Object-Oriented Design*. `http://www2.informatik.uni-wuerzburg.de/SugiBib/Postscript/UMLscript.ps`.

[31] SELIC, BRAN und JIM RUMBAUGH: *Using UML for Modeling Complex Real-Time Systems*. Published by Objectime, 1998.

[32] SHEN, HAI-HUA, CHAO LIU, MEI-MING SHEN und WEI-MIN ZHENG: *An Algorithm for Describing Object-Oriented Software Architecture Using Graph*. TOOLS 31. Proceedings, 1999.

[33] SHILLING, JOHN J. und JOHN T. STASKO: *Using Animation to Design, Document and Trace Object-Oriented Systems*. Technical-Report GIT-GVU, 1992.

[34] STASKO, J.T.: *Information about XTango, Tango, Polka and Samba.* `http://www.cc.gatech.edu/gvu/softviz/algoanim/`.

[35] STASKO, J.T.: *Tango: A Framework and System for Algorithm Animation.* PhD Thesis, 1989.

[36] STASKO, J.T.: *Animating Algorithms with XTANGO.* SIGACT News, 1992.

[37] SUZUKI, JUNICHI und YOSHIKAZU YAMAMOTO: *Making UML models interoperable with UXF.* Lecture notes in Computer Science, 1998.

[38] WAHL, GÜNTER: *UML kompakt.* Objektspektrum, 1998.

[39] WONNEBERGER, DR. REINHARD: *Kompaktführer LaTeX.* Addison-Wesley, Bonn, 1988.

[40] XML-DEV MAILINGLISTE: *XSchema.* `http://purl.oclc.org/NET/xschema`.

[41] ZELLER, ANDREAS: *Datenstrukturen visualisieren und animieren mit DDD.* Tagungsband GI-Workshop Softwarevisualisierung 2000, 2000.

[42] ZERBE, KLAUS: *Bauplan für Objekte.* c't, 1999.