Studienarbeit:
Einsatz von XML zur Repräsentation und Austausch von UML-Diagrammen im Rahmen von Softwareprozessen

Hilger Müller

Betreuer: Dr. Ralf Klischewski

Hilger Müller
Beidenfletweg 4
22149 Hamburg

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

XML ist eine Auszeichnungs-Sprache, deren Potential ich in Bezug auf Softwareprozesse untersuchen möchte. Insbesondere den Aspekt des Austauschens von Diagrammen, die in UML (**U**nified **M**odelling **L**anguage) definiert sind. Diesen Aspekt halte ich im Zusammenhang mit Softwareprozessen in verteilten und heterogenen Umgebungen für besonders wichtig.

Bei der Entwicklung komplexer Softwaresysteme trifft man immer öfter Modellierungswerkzeuge an, die UML verwenden. Sie sollen das Software-Design durch Visualisierung unterstützen. Hierbei sind besonders im Umfeld heterogener Entwicklungsumgebungen zwei Problemkomplexe besonders zu betrachten:

- zum Einen der vertikale Entwicklungsprozess: dieser umfasst die Beziehungen verschiendener Entwicklungsaktivitäten untereinander z.B. die Konsistenzhaltung von UML-Diagrammen und Softwarestruktur

- zum Anderen der horizontale Entwicklungsprozess: dieser besteht aus den Beziehungen zwischen gleichen Eintwicklungsaktivitäten und unterschiedlichen Werkzeugen z.B. die gemeinsame Nutzung von Modellen in verschiedenen UML-Werkzeugen bzw. in heterogenem Umfeld.

Die Konsistenz der Softwarestruktur und der korrespondierenden UML-Diagramme hängt stark von den Schnittstellen der verwendeten Systeme ab. Denn in der Praxis zeigt sich, je mehr manuelle Arbeit nötig ist, um Diagramme und Softwarestruktur abzugleichen, desto mehr entwickeln sich die beiden Strukturen i.d.R. auseinander. Das Problem besteht oft darin, dass Änderungen in einem System

nicht (oder zumindest nicht vollständig) an abhängige Systeme weitergeleitet werden können (sowohl vertikal als auch horizontal), z.B. können modifizierte Programme oft nicht die UML-Diagramme adäquat ändern und umgekehrt, wodurch Inkonsistenzen entstehen.

Dieses könnte vermieden werden, wenn es eine Möglichkeit gäbe, die verwendeten Systeme miteinander zu verbinden. XML (E**x**tensible **M**arkup **L**anguage) ist ein offener Standard und könnte als Repräsentation von UML-Diagrammen in verschiedenen Werkzeugen verwendet werden.

Tatsächlich bieten mittlerweile einige Hersteller die Möglichkeit UML-Diagramme als XML Dokumente zu speichern. Die Tatsache allein, dass es UML-Werkzeuge gibt, die XML unterstützen, ist jedoch noch kein hinreichender Nachweis für deren Sinnhaftigkeit. Die Motive hierfür könnten durchaus auch in Marketingüberlegungen und in anderen strategischen Erwägungen zu finden sein.

1.2 Aufbau der Arbeit

Der folgende Abschnitt beschreibt den Aufbau der Arbeit, in dem erst die Ziele, die zu beachtenden Probleme und dann die gewählte Vorgehensweise beschrieben wird.

Ziele der Arbeit

Den Einsatz von XML möchte ich dahingehend untersuchen, ob dieser zur Repräsentation und zum Austausch von UML-Diagrammen in Softwareprozessen geeignet und sinnvoll ist?

Probleme

Um zeigen zu können, ob der XML-Einsatz geeignet ist, müssen die Anforderungen für diesen Einsatz ermittelt werden. Da aber die möglichen Programmiersprachen nicht unnötig eingeschränkt werden sollen, können die Anforderungen nur allgemein gehalten sein und sich nicht auf konkrete Programm-Beispiele stützen.

Für den Austausch von UML-Diagrammen gilt Ähnliches, wenn eine automatische Transformation von Programmtexten in UML-Diagramme und umgekerhrt erfolgen soll: die Diagramme spiegeln verschiedene Sichten auf das Datenmodell wider, deren Abbildung auf die jeweilige Programmiersprache nicht eindeutig ist. Das heißt, dass die Eigenschaften die mit UML-Diagrammen dargestellt werden können auf unterschiedliche Art und Weise umgesetzt werden. Dies hat zur Folge, dass die Umsetzung von UML-Diagrammen an den entsprechenden Stellen

in den Programmtexten standardisiert sein muss, damit alle beteiligten Anwendugen die Änderungen gleich interpretieren. Das gleiche gilt für die umgekehrte Richtung, also Änderungen die in den Programmtexten erfolgen, müssen eindeutig auf Änderungen in den UML-Diagrammen abgebildet werden können. Einen solchen Standard zu entwickeln ist nicht trivial. Da es Standards gibt, die UML-Diagramme speichern und austauschen können, werde ich diese betrachten und bewerten.

Vorgehen

Im Einzelnen möchte ich den Einsatz von XML zur Repräsentation und zum Austausch von UML-Diagrammen in Softwareprozessen zeigen indem ich...

- darstelle welche Funktion UML-Diagramme in nebenläufigen Softwareprozessen haben,
- Anforderungen an den Austausch von UML-Diagrammen ermittle,
- die Wahl von XML zur Repräsentation von UML-Diagrammen motiviere und begründe,
- überprüfe, ob eine Repräsentation von UML-Diagrammen in XML existiert die sich für den gewählten Einsatzkontext eignet,
- den XML-Einsatz zur Repräsentation und zum Austausch von UML-Diagrammen bewerte.

1.3 Gliederung

Im Anschluss dieses Kapitels folgt die Klärung des Begriff des Softwareprozesses dieser Arbeit in **Kapitel 2** und beschreibt im weiteren die Abhängigkeiten der einzelnen Aktivitäten in nebenläufigen Softwareprozessen. Anschließend wird die Bedeutung der einzelnen UML-Diagramm-Typen in Softwareprozessen thematisiert.

Kapitel 3 motiviert die Verwendung von XML, denn zum einen setzt der Standard XMI zur Repräsentation von UML-Diagrammen XML ein, zum anderen soll, aus Gründen einer möglichst objektiven Betrachtung, XML und andere Ansätze (Formate, Sprachen oder Werkzeuge) gegenübergestellt und in Bezug auf die Tauglichkeit und Adäquatheit als Repräsentation von UML-Diagrammen bewertet werden. Des Weiteren soll eine Repräsentation der erwähnten Diagramme in

dem gewählten Format gefunden werden. Hierzu werden existierende Standards kurz betrachtet.

Kapitel 4 ermittelt aus den Aufgaben von UML-Diagrammen in Softwareprozessen (aus Kapitel 2) und der Auswahl einer Repräsentation von Diagrammen (aus Kapitel 3), ob XML für den Austausch von UML-Diagrammen geeignet ist.

Kapitel 5 fasst das bisher Geschriebene zusammen und beschäftigt sich mit der möglichen Zukunft der verwendeten Technologien und deren Auswirkungen auf die Softwareentwicklung.

1.4 Konventionen

In dieser Arbeit verwende ich die Neuregelung der deutschen Rechtschreibung (Rechtschreibreform).

Die ausschließliche Nennung der männlichen Schreibweise (z.B. der Programmierer), beinhaltet auch die weibliche Form, ohne dass diese explizit aufgeführt wird. Dieses geschieht aus Gründen der besseren Lesbarkeit.

Textbeispiele werden in einer **`Schreibmaschinen-Schrift`** dargestellt, um diese Bespiele besser von dem übrigen Text abgrenzen zu können.

Kapitel 2

UML-Diagramme in Softwareprozessen

Kapitel 2 klärt den Begriff des Softwareprozesses dieser Arbeit und beschreibt im weiteren die Abhängigkeiten der einzelnen Aktivitäten in nebenläufigen Softwareprozessen. Anschließend wird die Bedeutung der einzelnen UML-Diagramm-Typen in Softwareprozessen thematisiert.

2.1 Softwareprozesse

Eine besondere Herausforderung bei der professionellen Softwareentwicklung ist die Herstellung komplexer und qualitativ hochwertiger Software-Systeme.

Erschwerend kommt hinzu, dass komplexe Software meist von mehreren Personen erstellt bzw. bearbeitet wird und somit ein methodisches Vorgehen für den Erfolg großer Projekte sehr wichtig ist.

Um die Probleme bei der professionellen Softwareentwicklung anzugehen hat sich zunächst[1] eine ingeneurartige Vorgehensweise etabliert. Jedoch lässt dieses Vorgehen soziale Aspekte weitgehend aus bzw. betrachtet diese als potentielle Fehlerquellen. Es bildeten sich daraufhin zwei unterschiedliche Sichtweisen der Softwareentwicklung:

- Softwareentwicklung als Herstellung von Softwareprodukten, auf die die Prinzipien einer ingeneurmäßigen Entwicklung in ähnlicher Weise wie auf andere Produkte zutreffen. „Der Entwicklungsprozess wird als weitgehend technisch und formalisierbar aufgefasst und soll in fixierten Arbeitsschritten (sog. Phasen) ablaufen.“ [Rechenberg 99, 765]

[1] Nach der sog. Softwarekrise (ab 1965)

- Softwareentwicklung als sozialer und kommunikativer Prozess zwischen den Beteiligten. „Denn nur so können die Entwickler, die ja keine Anwendungsfachleute sind, das notwendige Wissen und das Verständnis für die Aufgaben im Anwendungsbereich und die Art und Weise ihrer Erledigung erwerben. Dies ist die Grundlage, um ein Anwendungssystem mit hoher Gebrauchsqualität entwickeln zu können. Doch das zukünftige System ist keine »eindeutige Lösung« für ein wohldefiniertes Problem. Die einzelnen Gruppen müssen sowohl die zukünftig anstehenden Aufgaben als auch deren angemessene Unterstützung im Entwicklungsprozess aushandeln. Nur einzelne Aspekte dieses Prozesses lassen sich in Dokumenten ausreichend detailliert beschreiben, so dass diese Dokumente für andere zur Arbeitsgrundlage werden können. Wesentliche Erfahrungen, Einsichten und Wertvorstellungen, die unabdingbar mit jedem Entwicklungsprojekt verbunden sind, können nicht in Dokumenten »objektiviert« werden.“ [Züllighoven 98, 267]

Der Begriff Softwareprozess beschreibt in beiden Sichtweisen insbesondere den Prozess der Softwareentwicklung.

Dieser Prozess kann durch unterschiedliche Vorgehensmodelle (Wasserfallmodell, Spiralmodell, zyklische Projektmodelle etc.) strukturiert sein, wobei zyklische Vorgehensmodelle explizit vorsehen, dass sich Anforderungen ändern können.

In der Praxis ist der lineare Ablauf des Wasserfallmodells kaum einzuhalten und das Spiralmodell berücksichtigt Versionen nur während der Herstellung und trennt Herstellung und Einsatz (bzw. Wartung) (s. [Züllighoven 98, 1103]).

Da in dieser Arbeit UML-Diagramme (s. Anhang) betrachtet werden lohnt sich ein Blick auf den **Rational Unified Process** der Firma Rational, die z. Zt. die drei Hauptentwickler von UML (G. Booch, I. Jacobsen, J. Rumbaugh) beschäftigt und das vielfach eingetzte UML-Produkt Rational Rose[2] entwickelt und vermarktet.

Der Unified Process bietet sich unter anderem auch deswegen an, weil dieser ein zyklisches Vorgehensmodell beinhaltet und explizit auf die Verwendung von UML-Diagrammen ausgerichtet ist.

[2]Ein Produkt zur visuellen Modellierung von Software-Systemen mithilfe von UML-Diagrammen.

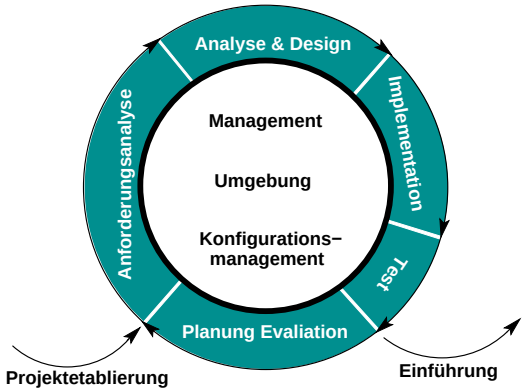

Abb. 2.1-1 Rational Unified Process [Rational 2000, 2]

Nach diesem zyklischen Vorgehensmodell wird in jedem Zyklus eine ausführbare Version des Anwendungssystems hergestellt.

In dieser Arbeit verwende ich eine leicht veränderte Darstellung des Unified Process, die folgende Unterschiede enthält:

- die Entwicklungs-Tätigkeiten Implementation und Test sind zusammengefaßt, da die Trennung für die weiteren Betrachtungen keinen Qualitativen Unterschied machen, aber die Übersicht dadurch verbessert werden kann,
- weitere Verbindungen zwischen den einzelnen Entwicklungs-Tätigkeiten wurden hinzugefügt, um die Wechselwirkungen der Aktivitäten untereinander besser darzustellen.

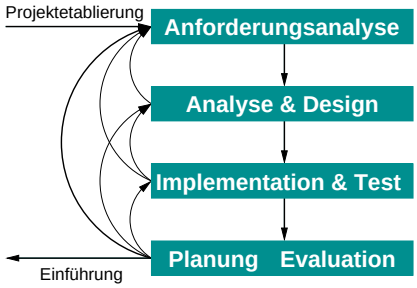

Abb. 2.1-2 Angepaßter Unified Process

Die zusätzlichen (dünnen) Pfeile geben an, dass die Aktivität von der der Pfeil ausgeht, Änderungen bei der Tätigkeit zur Folge haben kann, auf die die Pfeilspitze zeigt. Isoliert betrachtet (ohne weitere Prozesse) stellen in Abb 2.1-2 die vertikalen Pfeile verbunden mit den Entwicklungs-Tätigkeiten den vertikalen Entwicklungsprozess dar.

2.1.1 Tätigkeiten im Unified Process

Die im Unified Process verwendeten Entwicklungs-Aktivitäten, die als Tätigkeiten bezeichnet werden, sind insbesondere auf Geschäftsprozesse (Vorgänge in der Geschäftswelt) ausgerichtet (s. [Rational 98, 13]) und haben folgende Zuständigkeiten.

Projektetablierung (Initial Planning)

Die Projektetablierung enthält die Aktivitäten, die den offiziellen Start des Projektes auslösen. Es sollte ein vorläufiger Projektplan erstellt werden und eine Strategie für das Projekt. Es sind also der Projektumfang sowie die angestrebten Ziele anzugeben.

Anforderungsanalyse (Requirements)

Die Anforderungsanalyse beschreibt den Leistungsumfang des System und erlaubt den Entwicklern und den Kunden sich über diese Beschreibung zu einigen.

Durch die Analyse der Ist-Situation werden die systemspezifischen Probleme entdeckt. Die Ist-Situation soll dokumentiert und verstanden werden. Dies beinhaltet sowohl menschliche als auch automatisierte Komponenten.

Analyse und Design

Das Ziel der Analyse und des Designs ist darzustellen, wie die Anforderungen realisiert werden sollen.

Die Design-Aktivitäten befassen sich haupsächlich mit der Notation der Architektur des Systems. Die Herstellung und Überprüfung dieser Architektur ist die Hauptaufgabe in frühen Iterationen des Zykluses. Die Architektur wird repräsentiert durch eine Zahl von architektonischen Sichten. Diese Sichten enthalten die haupsächlichen strukturellen Design-Entscheidungen. Architektonische Sichten sind Abstraktionen oder Vereinfachungen des gesamten Designs, in welchem die wichtigen Charakteristika, durch Weglassen von Details, hervorgehoben werden. Hier wird die Software-Struktur weitgehend festgelegt.

Implementation und Test

Die Implementation ist die technische Umsetzung (Programmierung) der vorliegenden Modellierung (des Anwendungs-Systems) auf der Zielplattform (evtl. auch mehrere Plattformen).

Der Test des Systems hat folgende Gründe:

- Überprüfen der Interaktion zwischen Objekten,
- Überprüfen der Integration aller Softwarekomponenten,
- Überprüfen ob alle Anforderungen korrekt implementiert sind (testen kann nur die Anwesenheit von Fehler, nie aber deren Abwesenheit beweisen [Rechenberg 99, 783]).
- Identifizieren von Fehlern und Sicherstellen, dass diese Fehler vor der Auslieferung der Software behoben sind.

Testen ist eine Qualitätssicherungs-Maßname, die die Zuverlässigkeit, Funktionalität und Performanz überpüfen soll.

Planung und Evaluation

Die Evaluation bewertet die bisherigen Ergebnisse der Iteration (im zyklischem Vorgehensmodell) und beeinflussen die weitere Planung sowie die übrigen bewerteten Eigenschaften oder Vorgänge.

Die Bewertung jeder Iteration ist ein wesentlicher Bestandteil des iterativen Ansatzes.

Die Planung nimmt die Erkenntnisse der Evaluation auf und beeinflusst die Vorbereitung der nächsten Iteration.

Einführung

Einführung des Systems beim Kunden und Wartung des laufenden Systems.

Es existiert das Problem, dass bei den Übergängen der genannten Tätigkeiten zu Modellbrüchen kommen kann, denn diese Tätigkeiten stellen unterschiedliche Aspekte des Systems in den Vordergrund. Beispielsweise soll bei dem Design die ermittelten Anforderungen umgesetzt werden, wobei nicht ganz klar ist wie sich diese Anforderungen in der System-Architektur auswirken.

Der Unified Process ist offen für ein Top-Down oder eine Bottom-Up-Vorgehen. Bei der Top-Down-Vorgehensweise wird das Anwendungs-System aus der Analyse, das Anwendungs-System stufenweise (von grob bis fein) synthetisiert bzw. realisiert. Die Bottom-Up-Methode konstruiert aus identifizierten Elementen der Anwendungswelt Objekte und darauf aufbauend die Struktur des Systems.

2.1.2 Nebenläufige Softwareprozesse

Um den Zusammenhang paralleler oder verteilter Softwareprozesse (z.B. in heterogenen Netzwerken) zu verdeutlichen stelle ich zwei Prozesse exemplarisch nebeneinander dar und zeige die Abhängigkeiten der einzelnen Teilschritte untereinander, wenn sie gemeinsame Ressourcen nutzen.

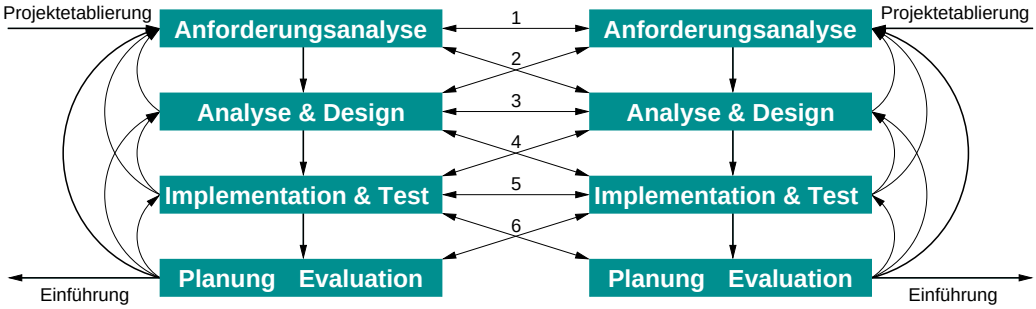

Abb. 2.1.2-1 Nebenläufige Softwareprozesse

Nach strenger Sicht gehören nur die Pfeile 1, 3 und 5 Verbindungen der Abb. 2.1.2-1 zum horizontalen Entwicklungsprozess. Die übrigen nummerierten Pfeile könnte man als zum diagonalen Entwicklungs-Prozess (Beziehung zwichen unterschiedlichen Entwicklngs-Aktivitäten und unterschiedlichen Prozessen) gehörig bezeichen.

Die in Abb. 2.1.2-1 dargestellten horizontalen bzw. diagonalen Abhängigkeiten zwischen verschieden Prozessen entstehen (korrespondierend mit der Pfeilnummerierung), wenn...

1. ein Projekt die Anforderungen eines anderen Projektes verwendet oder sich auf die fachlichen Modelle eines anderen Prozesses beziehen,
2. Anforderungen anderer Prozesse für die Analyse und das Design herangezogen werden,
3. Designelemente aus anderen Prozessen verwendet werden,
4. Designelemente anderen Prozessen implementiert werden,
5. Implementierungen aus anderen Prozessen genutzt werden,
6. Implementierungen anderer Prozesse überprüft werden.

Zwischen Tätigkeiten auf verschiedenen Ebenen findet ein Austausch von Informationen statt, der Voraussetzungen oder Ergebnisse von anderen (potentiell entfernten) Prozessen verwendet. Das bedeutet, dass Informationen der einen Ebene

in solche transformiert werden müssen, die in der anderen sinnvoll genutzt werden können, denn jede genannte Tätigkeit hat inhaltliche Auswirkungen auf die übrigen Tätigkeiten.

Diese Informationen können in den einzelnen Aktivitäten mit UML-Diagrammen partiell[3] festgehalten und visualisiert werden. Um diese Diagramme prozessübergreifend verwenden zu können, muss ein Austausch dieser Diagramme möglich sein.

2.2 Einordnung von UML-Diagrammen in Softwareprozessen

UML-Diagramme stellen verschiedene Aspekte der antizipierten Anwendung dar und können somit helfen die Übersicht bei komplexen Systemen zu behalten. UML vereint mehrere etablierte Ansätze[4] der Modellierung. Durch die beschriebene Vorgehensweise des Unified Process, der explizit UML-Diagramme einsetzt, kann mit Hilfe dieser Diagramme die Konstruktion der Anwendung unterstützt werden.

[3] Je nach Diagramm-Art stehen andere Aspekte dieser Inforamtionen im Vordergrund bzw. überhaupt zur Verfügung.

[4] OMT (Object Modelling Technique) von James Rumbaugh, Use Cases von Ivar Jacobsen und die Booch-Methode von Grady Booch

UML−Diagramme

Use−Cases beschreiben Anforderungen in Form von Akteuren (Personen bzw. funktionelle Rollen) und die ihnen zugeordneten Use−Cases (Anwendungsfälle d.h. Aufgaben bzw. Aufgabengebiete).

Klassen−Diagramme stellen die Struktur und den Aufbau der verwendeten Klassen dar.

Interaktions−Diagramme kommen in zwei Ausprägungen vor:

- **Sequenz−Diagramme** stellen dar, welche Objekte wann und womit auf Andere zugreifen.
- **Kollaborations−Diagramme** beinhalten Objekte, die Nachrichten austauschen.

Package−Diagramme fassen Diagramme oder Elemente zusammen und zeigen deren Abhängigkeiten.

Zustands−Diagramme stellen mögliche Zustände von Klassen und deren Übergänge dar.

Aktivitäts−Diagramme beschreiben Objekte eines Vorganges mittels der Aktivitäten, die sie wähtend des Ablaufes ausführen.

Implementations−Diagramme gibt es in zwei Ausführugen:

- **Komponenten−Diagramme** modellieren Abhängigkeiten zwischen Programmtext, Binärcode−Komponenten und ausführbaren Programmen.

Deployment−Diagramme visualisieren Hardware und deren Verbindungen.

Diagrammarten der UML-Spezifikation (Version 1.3) [OMG 2000b]

Da UML besonders die Modellierung unterstützt, möchte ich zunächst den in dieser Arbeit verwendeten Modellbegriff eingehen:

Modelle sind meist vereinfachte sowie idealisierte Abstraktionen komplexerer Systeme. Das Ziel ist eine Komplexitätsreduktion um das System handhabbar zu machen.

Unter Modellierung verstehe ich bei der Analyse und Entwurf von Softwaresystemen die Bildung abstrakter Modelle der Anwendung. Dies wird durch die Implementation zu einem Datenmodell konkretisiert.

Da jedes UML-Diagramm andere Aspekte der abstrakten Modelle hervorhebt, stellen diese somit auch verschiedene Sichten auf das Datenmodell dar. Daraus folgt, dass abhängig von dem Kontext die Schwerpunkte der gewählten Diagrammarten unterschiedlich sind.

Die zentrale Rolle in den meisten Softwareentwicklungs-Umgebungen mit UML-Unterstützung nehmen die Klassen-Diagramme, die die Struktur und den Aufbau der verwendeten Klassen darstellt, ein. Das hängt zum Einen damit zusammen, dass daraus direkt ein Programmtext-Gerüst erstellt werden kann bzw. aus bestehender Software (sofern unterstützt) Klassen-Diagramme gebildet werden können. Zum Anderen wird von einigen Entwicklern die Klassenhierarchie höher bewertet als andere Zusammenhänge, die mit den übrigen Diagrammen dargestellt werden können, wobei manchmal auch nur die Einarbeitung in andere Diagrammarten gescheut wird.

Anforderungsanalyse

„Anforderungen an die Software stammen aus dem Anwendungsbereich und werden aus den Erfordernissen der dort relevanten technischen Prozesse und Arbeitsprozesse abgeleitet.“ [Rechenberg 99, 775]

In Softwareprozessen werden Use-Cases sehr früh eingesetzt (bei der Anforderungsermittlung), da sie Rollen und Aufgaben festhalten und benennen ohne schon eine Software-Struktur festzulegen. Use-Cases beschreiben die Anforderungen ohne das Datenmodell zu sehr einzuschränken.

Analyse und Design (Modellierung)

Bei der Modellierung lassen sich Klassen-Diagramme direkt verwerten, denn sie spiegeln die geplante oder vorhandene Software-Struktur (möglichst korrekt) wieder, denn die Begriffshierarchie der Anwendungswelt und die Klassenhierarchie der Anwendung sollen strukturähnlich sein [Züllighoven 98, 263]. Die Forderung nach Strukturähnlichkeit bietet folgende Vorteile:

- die Klassenhierarchie (und damit auch Klassen-Diagramme) können als Diskusionsgrundlage für Anwender und Entwickler (auch gemeinsam) verwendet werden,
- neue oder geänderte Anforderungen (in der Fachsprache formuliert) können besser zu- und eingeordnet bzw. verstanden werden.

Diese Strukturähnlichkeit fördert, unter Verwendung der Fachsprache, das Verständnis der Anwendungswelt und somit des antizipierten Anwendungs-Systems.

Das Verstehen der Anwendungswelt ist ein kritischer Punkt der Anwendungsentwicklung, die durch frühe Partizipation der Anwender des angestrebten Systems

(z.B durch Prototyping) verbessert werden kann. Ohne die Anwendungswelt ausreichend verstanden zu haben, kann dafür auch kein passendes Modell erstellt werden.

Bei einer Top-Down-Vorgehensweise können recht früh beim Design Package-Diagramme eingesetzt werden, um die grobe Struktur durch Einteilung in Module oder Pakete, deren Inhalte und gegenseitige Abhängigkeiten zu definieren. Hier werden dann erst später Klassen-Diagramme verwendet um die Module oder Pakete zu konkretisieren. Nach der Bottom-Up-Methode würde man hier umgekehrt vorgehen.

Wenn bei der Entwicklung bestimmter Klassen von Objekten sich verschiedene Zustände und deren Übergänge identifizieren lassen, können Zustands-Diagramme verwandt werden. Diese benennen und zeigen graphisch Anfangs- und Endzustände, die möglichen Zustände dazwischen sowie deren Übergänge.

Implementation

Während der Implementation ist der Einsatz von Klassen-Diagrammen ebenfalls wichtig, um die Übersicht zu behalten, denn die Diagramme stellen die Software-Struktur besser dar als Programmtexte.

Sequenz-Diagramme geben die dynamische Struktur wieder. Diese Diagramme zeigen die Interaktionen der beteiligten Objekte in zeitlicher Abfolge.

Dieses hilft bestimmte Situationen oder Protokolle zu beschreiben und damit besser verstehen zu können.

Des Weiteren sind hier Implementierungs-Diagramme hilfreich, um die Hardware- und Komponenten-Struktur festzuhalten. Komponenten-Diagramme stellen die Abhägigkeiten der verwandten Typen und deren Schnittstellen dar, die bei der Implementation berücksichtigt werden müssen.

Bei der Einführung und Wartung kann ein Deployment-Diagramm, das die vorhandene Hardware-Struktur widerspiegelt, eine große Hilfe sein.

Viele Entwicklungswerkzeuge stellen ihre Dienste nur als Systemaufrufe zur Verfügung, d.h. die Dienstleistungen sind nur lokal zur verwendeten Maschine nutzbar. Dies reicht aber im Kontext verteilter Projektgruppen, die auf gemeinsame Ergebnisse zugreifen müssen, nicht mehr aus, so dass hier entfernte Anfragen (z.B. basierend auf dem Client-Server-Modell) die geforderten Leistungen (zumindest potentiell) erbringen.

2.3 UML-abhängige Übergänge in Softwareprozessen

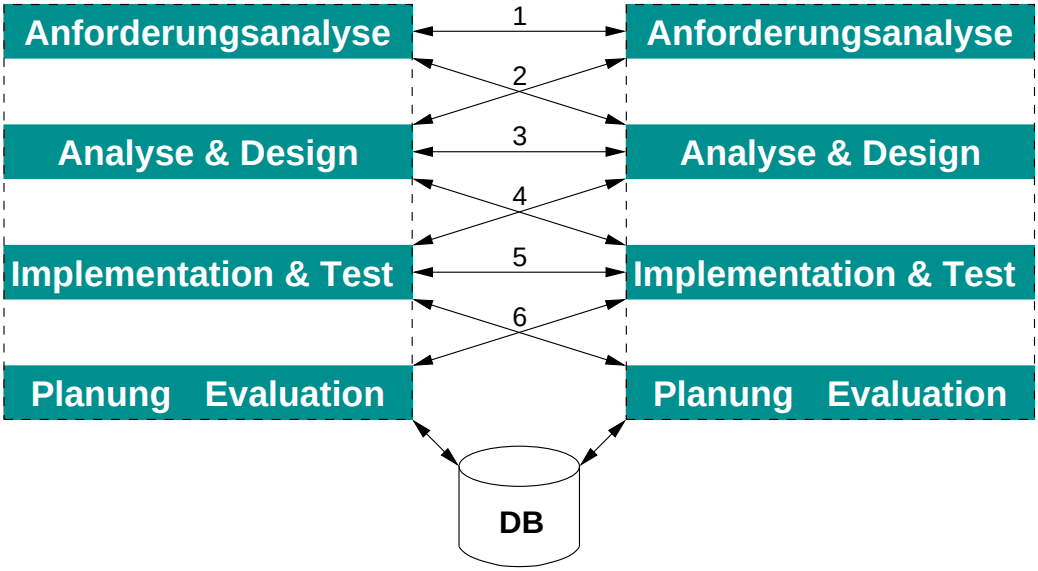

Abb. 2.3-1 Abhängigkeiten nebenläufiger Softwareprosesse

Die in Abb. 2.3-1 nummerierten Übergänge sind bei Verwendung einer zentralen Datenhaltung (DB) nicht direkt miteinander verbunden. Aus Gründen der besseren Übersicht und Zuordnung sind diese aber trotzdem als direkte Verdindungen idealisiert.

Die dargestellten Aktivitäten beziehen sich auf UML-Diagramme anderer Prozesse (korrespondierend mit der Pfeilnummerierung), wenn :

1. ein Projekt die Use-Cases eines anderen Projektes verwendet. Die Use-Cases stellen einen Teil der Anforderungen eines Prozesses dar, die von anderen Prozessen verwendet werden können,

2. die Modellierung sich auf Klassen-Diagramme eines anderen Prozesses bezieht,

3. Klassen aus Klassen-Diagrammen anderer Prozesse verwendet werden,

4. Klassen aus Klassen-Diagrammen anderer Prozesse implementiert werden,

5. Implementations-Diagramme aus anderen Prozessen genutzt werden,

6. Deployment-Diagramme anderer Prozesse verwendet werden.

Alle nummerierten Übergänge können UML-Diagramme verwenden, womit diese Übergänge den Austausch von bestimmten UML-Diagrammen unterstützen sollten.

Die verschiedenen UML-Diagramme können folglich in allen benannten Tätigkeiten eingesetzt werden.

Nebenläufige Softwareprozesse die UML-Diagramme einsetzen, müssen in der Lage sein auf die Diagramme (bzw. deren Inhalte) der anderen Prozesse zugreifen zu können. Um dies zu erreichen, muss ein Austausch von UML-Diagrammen möglich sein. Ein Austausch über Prozessgrenzen hinweg erfordert ein einheitliches Format, das von allen Beteiligten gelesen und verarbeitet werden kann.

Zwischen Tätigkeiten auf verschiedenen Ebenen findet ein Austausch von Informationen statt, der Ergebnisse bzw. Inhalte von anderen (potentiell entfernten) Prozessen verwendet. Das Bedeutet, dass Informationen der einen Ebene in solche transformiert werden müssen, die in der anderen sinnvoll genutzt werden können, denn jede der genannten Tätigkeiten hat inhaltliche Auswirkungen auf die übrigen Tätigkeiten.

Diese Informationen können in den einzelnen Aktivitäten mit UML-Diagrammen partiell festgehalten und visualisiert werden. Um diese Diagramme prozessübergreifend verwenden zu können, muss ein Austausch dieser Diagramme möglich sein.

Kapitel 3

Repräsentation von UML-Diagrammen in XML

Kapitel 3 motiviert die Verwendung von XML, denn zum einen setzt der Standard XMI zur Repräsentation von UML-Diagrammen XML ein, zum anderen soll, aus Gründen einer möglichst objektiven Betrachtung, XML und andere Ansätze (Formate, Sprachen oder Werkzeuge) gegenübergestellt und in Bezug auf die Tauglichkeit und Adäquatheit als Repräsentation von UML-Diagrammen bewertet werden. Des Weiteren soll eine Repräsentation der erwähnten Diagramme in dem gewählten Format gefunden werden. Hierzu werden existierende Standards kurz betrachtet.

3.1 Zielsetzung von XML

Um die Eignung von XML zur Repräsentation von UML-Diagrammen untersucht zu können, möchte ich die Zielsetzung von XML zunächst kurz vorstellen und um Aufbau und Ziele dieser Sprache aufzuzeigen sowie darauf basierend eine Beurteilung für den XML-Einsatz als UML-Diagramm Speicher- und Transport-Medium vornehmen zu können.

Die XML (E**x**tensible **M**arkup **L**anguage) ist eine Teilmenge von SGML (**S**tandard **G**eneral **M**arkup **L**anguage[1], ISO 8879) mit dem Ziel generisches SGML anzubieten und im Web zu verarbeiten, wie es mit HTML (**H**yper**t**ext **M**arkup **L**anguage) jetzt schon möglich ist. XML wurde entwickelt für einfache Implementierungen und für Interoperabilität zwischen SGML und HTML.

[1]Nicht zufällig besteht der nicht standardisierte Vorgänger von SGML (GML) aus den Anfangsbuchstaben der ersten Entwickler von GML bei IBM: Golfarb, Mosher und Lorie.

Zielsetzungen von XML [W3C 2000] sind:

1. XML soll einfach über das Internet benutzbar sein.
2. XML soll ein breites Spektrum an Anwendungen unterstützen.
3. XML soll mit SGML kompatibel sein.
4. Es soll einfach sein, ein Programm zu schreiben, das XML-Dokumente verarbeitet.
5. Die Anzahl an optionalen Eigenschaften in XML ist minimal, idealerweise Null.
6. XML-Dokumente sollen menschenlesbar und angemessen klar sein.
7. Die XML-Konstruktion soll schnell verfasst werden.
8. Der Aufbau von XML soll formal und präzise sein.
9. XML-Dokumente sollen leicht zu erstellen sein.
10. Kürze ist in XML-Markup von geringer Bedeutung.

Die Punkte 2, 4, 6, 8 und 9 sind für den Austausch von UML-Diagrammen als XML-Dokumente direkt von Bedeutung, wobei die übrigen Designziele dem gewünschten Einsatz nicht entgegen stehen.

Der Punkt 10 wirkt zunächst ineffizient. Wenn man jedoch bedenkt, dass effektive Komprimierungsprogramme existieren, die besonders Text-Dokumente sehr stark verkleinern können, relativiert sich dieser Punkt wieder. Denn XML-Dokumente bestehen aus Text, der mit fast jedem Text-Editor bearbeitet werden kann.

X**ML** ist wie z.B. HT**ML** oder SG**ML** eine Auszeichnungssprache[2]:

> *„Markup ist eine Methode zur Übermittlung von Metadaten, d.h. Daten über Daten. Auszeichnungssprachen benutzen Literale oder sogenannte Tags, um Metadaten zu beschreiben und vom eigentlichen Inhalt zu trennen.“* [Anderson 2000]

Im Gegensatz zu HTML liegt in XML den einzelnen Tags keine weitere Semantik zugrunde, als der Definition der Entitäten selbst. Das heißt Namen, Struktur und Daten (Inhalt der Entitäten) werden mit XML definiert, nicht jedoch deren Bedeutung oder Aussehen. Dies kann mit den im nächsten Abschnitt genannten Standards erfolgen.

[2] **M**arkup **L**anguage

3.2 XML-basierte und Metamodell Standards

Im Folgenden stelle ich kurz weitere Standards vor, die Metamodelle oder XML verwenden, einschränken, überprüfen, und die damit verbundenen Möglichkeiten vorstellen:

Um einem XML-Dokument eine bestimmte Struktur vorzugeben[3], gibt es die **D**ocument **T**ype **D**efintion (DTD). Eine DTD ist ein Text-Dokument (nicht XML-Konform) und mit bestimmten Programmen kann die Gültigkeit einer DTD für XML-Dokumente ermittelt werden. Allerdings stellen DTD's keine sehr starke Typisierung zur Verfügung.

Dieses ist erst mit XML-*Schema* der Fall. XML-Schemata (oder XML-Schemes) sind Text-Dokumente (XML-konform), die ebenfalls die Struktur von XML-Dokumenten beschreiben und mit speziellen Programmen validiert werden können. Weiterhin bieten XML-Schemes Wiederverwendungsmechanismen und Modularisierungsmöglichkeiten.

XSL (E**x**tensible **S**tyle **L**anguage) ist eine Dokumentenklasse die in XML formuliert, wie Darstellungen von XML-Dokumenten aussehen.

Aufgrund der hier entstandenen Komplexität, hat man den Bereich der Transformation (von XML in andere Formate z.B HTML) ausgelagert in XSLT (E**x**tensible **S**tyle **L**anguage **T**ransformation). XSLT wird ebenfalls in XML-Dokumenten verfasst.

Z.B. verwendet das CASE[4]-Tool INNOVATOR 7 (von MID) XML als Ausgabeformat (zur Erzeugung von HTML Dokumentationen mittels XSL-Stylesheets (XSLT)).

Mit XMI (**X**ML **M**etadata **I**nterchange) besteht ein Standard der den Austausch von Metadaten für Modellierungwerkzeuge (basierend auf UML) und Metadaten Repositories[5] in verteiltem heterogenen Umfeld ermöglicht. XMI baut auf den Industriestandards XML, UML und MOF auf.

MOF [OMG 2000a] (Meta Object Facility) stellt die Grundlage für die Definition von Metamodellen dar.

MOF besteht aus drei Hauptteilen:

[3] In einer DTD wird genau festgelegt, welche Tags verwendet werden dürfen und, welche Attribute diese Tags haben, welche anderen Tags in einem Tag vorkommen dürfen, sowie die Reihenfolge der Tags.

[4] Computer Aided Software Engeneering ist ein Ansatz, um die Softwareentwicklung mehr zu automatisieren und bietet Modellierungs-, Programmier-, Test-, und Dokumentationsmöglichkeiten, die einem bestimmten Vorgehensmodell zu Grunde liegen.

[5] Repository (engl.): Lager, in diesem Zusammenhang auch für Datenbanken, oder allgemeiner: Datenhaltung.

1. das MOF-Modell, welches das abstrakte Modell ist, das verwendet wird um alle Metadaten zu beschreiben,

2. die reflektiven Schnittstellen[6] von MOF, welche generische Schnittstellen zur Manipulation der Metadaten darstellen,

3. MOF-Mapping nach IDL, welches benutzt wird um Entwurfsmuster mit CORBA-IDL-Schnittstellen für MOF-Meta-Objekte zu spezifizieren.

Die Aufgabe von MOF ist es mittels eines standardisierten Systems Metadaten zu definieren und mittels CORBA (**C**ommon **O**bject **R**equest **B**roker **A**chritecture[7]) austauschen zu können.

3.3 Alternativen zu XML

Bevor die Tauglichkeit von XML als Repräsentation für UML-Diagramme überprüft wird, darf die Betrachtung möglicher Alternativen zu XML nicht fehlen.

Als konkurierende Ansätze habe ich ASCII (als einfaches Datenformat), HTML (als strukturierte Sprache) und Integrierte Systeme (mit evtl. proprietären Formaten oder Sprachen) ausgewählt, da diese die Möglichkeit bieten, auch semantische Aspekte von UML-Diagrammen festzuhalten.

Andere grafische Formate wie .vsd (Visio), gif, Tiff, JPEG etc. haben den Nachteil, dass nichtgrafische (sematische) Anteile von UML-Diagrammen nicht mitgespeichert werden können oder nicht standardisiert sind. Dies hätte zur Folge, dass bestimmte Namenskonventionen eingeführt werden müssten, um diese Informationen festhalten zu können. Bei rein grafischen Formaten (gif, Tiff, JPEG usw.) sind auch die Texte grafisch kodiert, so dass erst eine Texterkennung die Bezeichner extrahieren könnte. Aus diesen Gründen werden diese Formate in dieser Arbeit nicht weiter betrachtet.

Die Wahl des Formats der Repräsentation ist natürlich von mehreren Faktoren abhängig. Ein wichtiger Faktor in der Beurteilung ist der Aufwand, um auf die Daten in der konkreten Repräsentation zuzugreifen. Damit ist sowohl der programmiertechnische Aufwand, sowie der zeitliche Aufwand (für den Zugriff auf die Daten) gemeint.

[6] Dies sind Methoden, die über Klassen- und Methodeninformationen verfügen (also z.B. wieviele Methoden die Klasse hat und welche Signatur diese besitzen).

[7] CORBA ist eine Architektur, die es verteilten Objekte ermöglicht durch bestimmte Dienste (Services) sich zu organisieren und zu kommunizieren.

3.3.1 ASCII

Textdateien sind beliebt, um einfach strukturierte Daten zu speichern.

Bei komplexen Daten ist aber der Aufwand, die gewünschten Elemente zu extrahieren, dem Programmierer aufgebürdet und können umfangreich sein.

Hinzu kommt noch, dass man sich eine Syntax ausdenken muss, um den Datenelementen Werte zuzuordnen.

Die wesentlichen Schritte um den Datenaustausch mit ASCII-Dateien (American Standard Code for Information Interchange) zu realisieren sind:

- erstellen einer Syntax für die zu speichernden Daten,
- schreiben eines Parsers (evtl. mit Unterstützung eines Compiler-Compilers),
- schreiben von Programmtext für den Datenzugriff über das Datei- oder Netz-System.

Nachteilig bei diesem Vorgehen ist auch die Tatsache, dass beim Ändern der Syntax auch der Parser angepasst werden muss.

3.3.2 HTML

Durch die Omnipräsenz des World Wide Web im Geschäfts- und Privatleben vieler Menschen, ist HTML sehr weit verbreitet.

Im Gegensatz zu reinen Textdateien, muss man sich hier nicht ein eigenes Format bzw. eine eigene Syntax erarbeiten, aber dennoch muss man die zu repräsentierenden Daten an die Struktur von HTML anpassen.

Auch muss hier kein Parser selbst geschrieben werden, denn es gibt für viele Programmiersprachen kommerzielle und kostenfreie Lösungen (z.B. mittels JavaCC und dem entsprechenden HTML-Grammatikdokument).

Aber dadurch, dass es verschiedene Versionen und inkompatible Erweiterungen zu HTML gibt, muss man sich entweder für eine bestimmte Version entscheiden, oder sich auf die Schnittmenge von allen relevanten Variationen beschränken. Da aber für die Repräsentation nur eine kleine Teilmenge von HTML gebraucht wird, ist dieser Kritikpunkt von geringerer Relevanz.

3.3.3 Integrierte Systeme (z.B. CASE-Werkzeuge)

Integrierte Systeme haben bisher das Problem, Programm-Änderungen umzusetzen und an die entsprechenden Diagramme weiterzureichen und umgekehrt (dies

ist das sog. Round-Trip-Verfahren). Wodurch Probleme entstehen: Darstellungs- und Modellierungsdetails können verloren gehen. Diese Probleme werden teilweise damit umgangen, dass in dem Programmtext spezielle Kommentare eingesetzt werden, um Modellierungsdetails darzustellen (wenn z.B. ein Attribut von einer lokalen oder technischen[8] Variablen unterschieden werden muss).

Der Ansatz integrierter Systeme steht eigentlich nicht im Gegensatz zu dem bisher Genannten, denn es sagt nichts über die verwendete Basistechnologie aus. Es kann also jede der genannten (oder auch andere evtl. proprietäre) Technologien eingesetzt werden. Je nach Schwerpunkt des Systems kann eine andere Basistechnologie sinnvoll sein. Integrierte Systeme könnten erheblich von einem vereinigten Metamodell (Vergleich Kapitel 3.5.4) profitieren, durch die Verwendung von nur noch einer Datenbasis.

3.4 Eignung von XML zur Repräsentation von UML-Diagrammen

Um die Eignung von XML für die Repräsentation von UML-Diagrammen zu zeigen, ist es zunächst nötig eine passende Repräsentation zu entwickeln oder eine existierende zu finden. Mit XMI (siehe Kapitel 3.4.4) existiert ein solcher Standard, der auf MOF (nächster Abschnitt) basiert und ein Mapping zu DTD (übernächster Abschnitt) und XML-Schema (Kapitel 3.4.3) besitzt, weshalb die erwähnten Technologien, beginnend mit MOF, beschrieben werden.

3.4.1 MOF

Um UML-Diagramme zu definieren, ist es sinnvoll dieses mit Hilfe eines Metamodells (also ein Modell über Modelle) zu tun. Die OMG verwendet hierzu ein Vier-Schichten-Modell (siehe Tabelle).

[8] Mit technischen Variablen meine ich solche, die nicht direkt einen Objektzustand repräsentieren (also somit eigentlich kein Attribut darstellen) sondern programmtechnischer Natur sind, wie etwa Pufferspeicher, Indexvariablen, Zähler, Semaphore etc. .

Ebene	Beschreibung	Beispiel
Meta-Metamodell (M^3)	Die Infrastruktur für die Metamodell-Archietektur. Definiert die Spezifikations-sprache des Metamodells.	MetaKlasse, MetaAttribut, MetaOperation
Metamodell (M^2)	Eine Ausprägung bzw. „Instanziierung“ des Metamodells. Definiert die Spezifi-kationssprache für Modelle.	Klasse, Attribut, Operation, Komponente
Modell (M^1)	Eine Ausprägung des Metamodells. Defi-niert die Sprache zum Beschreiben der In-formationen des Anwendungsbereiches.	Konto, Saldo, abheben
Benutzter Ob-jekte (Benutzer Daten) (M^0)	Eine Ausprägung von Modellen. Defi-niert einen spezifischen Informations-bereich.	<0172121>, KreditLimit

Vier-Schichten Modell für Metamodelle [OMG 99, 29]

Von unten angefangen, bildet die erste Ebene (M^0) die Menge der Objekte, also die sog. „Instantiierung" (Bildung von Exemplaren aus Klassen) der darüberliegenden Ebene (M^1) von Objekt-Modellen (namentlich: Klassen). Die nächst höher liegende Ebene stellt das Metamodell (M^2) dar. Hier werden die Elemente und Konstrukte beschrieben mit denen Klassen gebildet werden können. Die oberste Ebene ist das Meta-Metamodell (M^3) und stellt die Mittel zur Verfügung, um Metamodelle zu definieren.

Die Spezifikation von UML verwendet als Meta-Metamodell (M^3) MOF (Meta Object Facility) [OMG 2000b, 40].

MOF ist ebenfalls in einem Vier-Schichten-Modell definiert, das wie folgt aussieht:

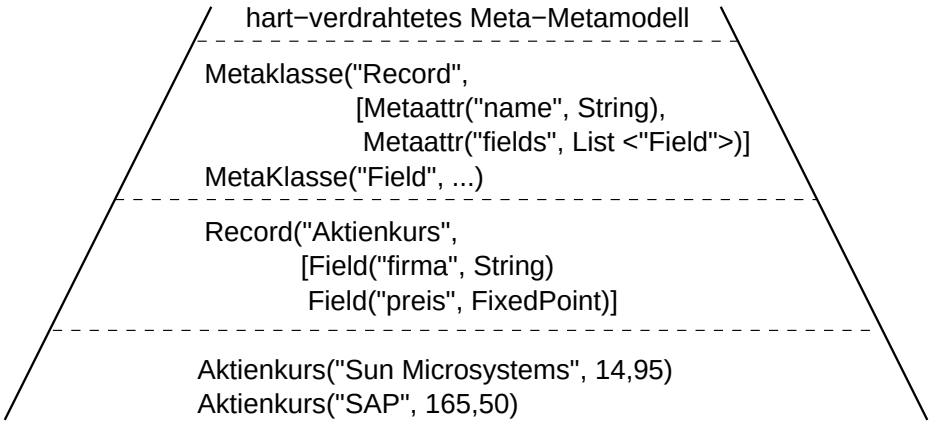

Abb. 3.3-1 Vier-Schichten MOF Metamodell

Diese vier Schichten sind konzeptionell ähnlich, jedoch sind bei MOF einige Einschränkungen in Kauf zu nehmen, um die Schnittstellen möglichst einfach zu halten (z.B. sind bei MOF nur binäre Assoziationen möglich, höhergradige Assoziationen können also nicht direkt in MOF umgesetzt werden).

3.4.2 DTD

Um XML-Dokumenten bestimmte Elemente und Strukturen vorzugeben, gibt es die **D**ocument **T**ype **D**efinition (DTD). In einer DTD wird genau festgelegt, welche Tags verwendet werden dürfen und, welche Attribute diese Tags haben, welche anderen Tags in einem Tag vorkommen dürfen, sowie die Reihenfolge der Tags. Eine DTD ist ein Text-Dokument (nicht in XML-Syntax) und mit bestimmten Programmen kann die Gültigkeit einer DTD für XML-Dokumente ermittelt werden. Allerdings stellen DTD's keine sehr starke Typisierung zur Verfügung. Beispiel für eine DTD, die den Aufbau des XML-Dokumentes eines Buches beschreibt:

```
<!DOCTYPE Buch [
  <!ELEMENT Buch (Titel, Autor, Kapitel+)>
  <!ELEMENT Titel (#PCDATA)>
  <!ELEMENT Autor (#PCDATA)>
  <!ELEMENT Kapitel (Überschrift, Absatz+)>
  <!ELEMENT Absatz (#PCDATA)>
]>
```

Die Deklaration von Elementen kann eine komplexere Grammatik enthalten, inklusive Cardinalitäten (Null bis eins "**?**", eins " ", Null oder mehr "*****" und eins oder mehr "**+**") und logischem Oder "**|**".

DTD's definieren auch die Attribute, die in einem Element unter Verwendung von **`ATTLIST`** enthalten sein können. Zum Beispiel spezifiziert die folgende DTD-Komponente, dass jedes Klassenelement einen optionales **`xmi.label`** Attribut haben kann und dass **`xmi.label`** aus Buchstaben-Daten besteht: (Die **`#IMPLIED`** Direktive zeigt an, dass das Attribut optional ist.)

```
<!ATTLIST Class xmi.label CDATA #IMPLIED>
```

3.4.3 XML-Schema

Neben der MOF Interface-Definition existiert auch eine XML-Schema Definition von Rational[9], die ein Mapping von UML zu XML definiert. Dieses Mapping setzt jedoch nicht auf DTD's, sondern auf XML-Schema auf, wodurch u.a. eine strengere Typisierung möglich ist.

Die Aufgabe von XML-Schema ist es, die abstrakte Struktur von XML-Dokumenten formal zu beschreiben. Ein XML-Schema-Dokument ist XML-Konform und beschreibt (wie auch DTD's), welche XML-Dokumente gültig sind. Es gibt Werkzeuge (z.B. xsv), die die Konformität von XML-Dokumenten zu XML-Schemata prüfen können.

Das XML-Schema kann mit unterschiedlichen Namensräumen umgehen. Namensräume werden mit xmlns (XML name space) benannt, z.B. beschreibt **`xmlns:xsd`**[10] den Namensraum xsd.

Elemente können in XML-Schema genauer als mit DTD's beschrieben werden, es wäre möglich die Anzahl der Kapitel (im Buch-Beispiel) von 2 bis 20 zu beschränken. Mit einer DTD ist dies nicht so einfach möglich.

Des Weiteren können zusätzlich zu den 44 vorhanden Datentypen in XML-Schemata eigene definiert und verwendet werden. DTD's sind auf 10 vordefinierte Typen beschränkt.

[9] Rational ist eine Firma, die u.A. das erfolgreiche UML-Werkzeug „Rational Rose“ produziert. Der Erfolg hat vermutlich auch damit zu tun, dass die drei Entwickler: Grady Booch, Ivar Jacobsen und James Rambaugh z. Zt. bei Rational tätig sind.

[10] XSD ist normalerweise die Datei-Endung (auch Extension genannt) von XML-Schema-Dokumenten

Das Beispiel auf der nächsten Seite zeigt die Umsetzung der Buch-DTD (im vorherigen Abschnitt) in XML-Schema:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
<xsd:element name="Buch">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="Title" minOccurs="1" maxOccurs="1"/>
   <xsd:element ref="Author" minOccurs="1" maxOccurs="1"/>
   <xsd:element ref="Kapitel" minOccurs="1" maxOccurs="unbound"
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
<xsd:element name="Kapitel">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="Überschrift" minOccurs="1" maxOccurs="1"
   <xsd:element ref="Absatz" minOccurs="1" maxOccurs="unbound"
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Author" type="xsd:string"/>
<xsd:element name="Kapitel" type="xsd:string"/>
<xsd:element name="Überschrift" type="xsd:string"/>
<xsd:element name="Absatz" type="xsd:string"/>
</xsd:schema>
```

3.4.4 XMI

Um Metadaten mittels XML auszutauschen ist XMI (XML Matadata Interchange, siehe Kapitel 3.2) vorgesehen. Bisher wird XMI hauptsächlich für den UML-Austausch eingesetzt, d.h. UML-Modelle werden in XMI (und damit auch in XML) repräsentiert. XMI hat (wie MOF und UML) ein Vier-Schichten-Metamodell.

Metaebene	Matadaten	XMI DTDs	XMI-Dokumente
M3	Das MOF Modell	MOF DTD	
M2	UML-Matamodell (und andere)	UML DTD (und andere)	MOF-Metamodell Dokumente
M2	UML-Modelle (und andere)		UML Modell Dokumente (und andere)
M0	Exemplare		

XMI und die OMG Metadaten-Architektur [OMG 99, 38]

XMI verwendet MOF um Metadaten zu definieren. XMI benutzt UML-Notationen um Modelle und Metamodelle darzustellen. Die Syntax von UML-Modellen in XMI-Dokumenten wird durch eine DTD beschrieben. Zu beachten ist in diesem Zusammenhang, dass diese DTD nur eine UML-Version abdeckt, die inkompatibel zu anderen Versionen ist, z.B. definiert XMI 1.2 eine UML-DTD die nur das Metamodell von UML 1.1 verwendet. Um XMI-Dokumente austauschen zu können muss sowohl die XMI Version sowie die UML Version der Tauschpartner übereinstimmen.

Ein einfaches Beispiel eines XMI-Dokumentes, das die Klasse C1 mit dem Attribut a1 enthält:

```
<XMI xmi.version="1.1" xmlns:UML="org.omg/standards/UML">
  <XMI.header>
    <XMI.metamodel name="UML" version="1.3" href="UML.xml"/>
    <XMI.model name="example" version="1" href="example.xml"/>
  </XMI.header>
  <XMI.content>
    <UML:Class name="C1">
      <UML:Classifier.feature>
        <UML:Attribute name="a1" visibility="private"/>
      </UML:Classifier.feature>
    </UML:Class>
  </XMI.content>
</XMI>
```

3.5 Ein vereinigtes Metamodell für UML und Implementierungssprachen

XMI reicht für den nahtlosen Übergang von UML zu Implementierungssprachen (z.B. für ein Round-Trip Verfahren siehe Kapitel 3.3.3) nicht aus, denn Implementierungsdetails aus der verwendeten Programmiersprache und einige Darstellungsdetails aus UML sind nicht Teil von XMI. Zum Beispiel können grafische Details nicht übermittelt werden (wie Position, Größe und Farbe von Diagramm-Elementen) und gehen somit in XMI-Dokumenten verloren.

Einen anderen Weg schlägt Tobias Baier vor:

> *„Die [...] genannten Probleme ließen sich so angehen, daß beide Metamodelle, das der Design- und das der Implementationssprache, verschmolzen werden. [...] die Konsequenz daraus ist, daß es nur noch ein Modell gibt, in dem alle Informationen des Projekts enthalten sind. Daraus folgt, daß Code, genau wie ein Diagramm, nur eine Sicht auf das Modell ist [...] . Es gibt nur noch eine Datenbasis, und Änderungen an einer Sicht auf das Modell (zum Beispiel Diagramm oder im Code) wären automatisch in allen anderen Sichten vorhanden. Die Modellinformationen sind immer konsistent, Diagramme und Code müßten nicht synchronisiert werden.“* [Baier 2000]

Dieser Ansatz könnte sich durchsetzen, wenn für jede in der Praxis bedeutende Programmiersprache ein Metamodell existiert und sich daraus ein gemeinsames Metamodell erarbeiten ließe. Besser noch wäre ein Standard, der je ein Metamodell für die verschiedenen Programmiersprachen-Paradigmen (objektorientiert, klassisch imperativ, funktional) definiert. Wenn kein gemeinsamer Standard für die Metamodelle der Programmiersprachen-Paradigmen oder zumindest der wichtigsten Programmiersprachen etabliert werden kann, so besteht natürlich das Problem, dass mehrere unterschiedliche und inkompatible Metamodelle diesen Ansatz kaum praktikabel erscheinen lässt.

Ich bin jedoch der Meinung, dass ein gemeinsames (in direkt einsetzbarer Form vorliegendes) Metamodell z.B. in XML aus folgenden Gründen sinnvoll ist:

- zum einen ist Komplexität nicht wirklich ein Gegenargument (der Umfang des Modells dürfte für Programme durchaus handhabbar sein),
- zum anderen muss nicht extra ein Compiler geschrieben oder erzeugt werden (mit Compiler Compilern wie: yacc, bison, javaCC etc.),
- auch lässt sich die Komplexität verringern, indem man nur die relevanten Konzepte in das Metamodell einfließen lässt. Also Klassenkonzept, Vererbung, Attribute usw., aber nicht unbedingt arithmetische Ausdrücke.
  Die Granularität des Metamodells sollte so grob wie möglich und nur so fein wie nötig sein,
- weiterhin ist zu beachten, dass bei Verwendung von XML das aus dem vereinigten Metamodell hervorgehende Dokument (Datei oder Stream) auch anderen Systemen zugänglich sein kann. Wenn auch nur im beschränktem Umfang, sofern nicht das gleiche Metamodell verwendet wird. Bei Verwendung von XMI als Basis für das Metamodell, sollte ein XMI-Export relativ

einfach möglich sein und somit die Verbindung mit anderen Systemen die XMI verwenden offen stehen (mit den erwähnten Einschränkungen).

Um ein vereinigtes Metamodell entwickeln zu können, müssen zunächst die einzelnen Metamodelle (für UML und Implementationssprachen) definiert werden:

Bei UML ist dies in der Spezifikation [OMG 2000b] basierend auf MOF bereits geschehen. Und mit XMI existiert sogar eine eingeschränkte Darstellungsmöglichkeit in XML.

Im Programmiersprachen-Bereich sieht das anders aus: es existieren hier hauptsächlich Definitionen von Grammatiken (z.B in BNF[11] oder EBNF[12]), aber meines Wissens bisher keine die auf Meta-Metamodellen wie MOF basieren. Es müsste hier also noch Arbeit investiert werden, um diese Lücke zu schließen.

Als Lösung (des Problems des Übergangs von UML zu Programmiersprachen) unter Berücksichtigung des Umstandes, dass ein vereinigtes Metamodell sehr umfangreich sein dürfte (z.B. sind arithmetische Ausdrücke in UML nicht spezifiziert), schlägt Herr Baier vor eine neue Programmiersprache zu entwickeln, die die nötigen Konzepte abdeckt und in verschiedene konkrete Programmiersprachen übersetzbar macht.

Java und UML

Im Java Umfeld gibt es eigene Ansätze um UML einzubinden:

- Java Metadata API (eine auf MOF basierende API für Java um Metadaten auszutauschen, die als Serialisierungsmechanismus für Metadaten XML einsetzt. Durch die MOF Basis ist der Gebrauch von XMI (und somit auch von UML) möglich)

- UML(1.3)/EJB[13](1.1) Mapping (hier wird unter Verwendung von UML-Profilen [ein Format, definiert von der OMG Object Analysis and Design Task Force (OA&DTF)] UML erweitert um Software zu modellieren, die mit Hilfe von EJB in UML implementiert wird).

[11] Backus Naur Form (John Backus, Peter Naur)

[12] Erweiterte BNF

[13] **E**nterprise **J**ava**B**eans stellen die Erweiterung des Komponentenmodells Java-Beans auf plattformunabhängige Server-Anwendungen dar; mit EJBs wird es Programmierern möglich, wiederverwendbare Server-Komponenten für geschäftliche Anwendungen zu erstellen, die auf den unterschiedlichen Systemen einer Unternehmensumgebung ausgeführt werden können.

3.6 Gegenüberstellung und Auswertung der unterschiedlichen Ansätze

Ein großer Vorteil von XML ist die Unterstützung für Programmierer durch API's (Application Programming Interface) für verschiedene Programmiersprachen und die Verfügbarkeit von Werkzeugen auf verschiedenen Plattformen, sowie das Umfeld von XML, das den Austausch, die Darstellung, die Repräsentation von UML-Modellen über standardisierte Schnittstellen ermöglicht.

Ein weiterer Vorteil von XML ist die Verwendung des Unicode zur Zeichendarstellung, welche eine größere Anzahl Schriftzeichen umfasst und u.a. auch den asiatischen Sprachraum unterstützt.

HTML kann das Umfeld von XML in dieser Form nicht bieten. Zwar gibt es auch API's und Werkzeuge, die die Programmentwicklung unterstützen, jedoch ist die Darstellung auf HTML selbst beschränkt (sofern man die Daten nicht in ein anderes Darstellungsformat transformiert). Auch ist die Repräsentation von UML-Modellen in HTML bisher nicht standardisiert, so dass eine eigene Darstellung erarbeitet werden muss.

Der Einsatz von ASCII erfordert einen ziemlich hohen Programmieraufwand, wenn die zu speichernden Daten komlex sind. Dies ist bei UML-Modellen sicherlich der Fall, so dass eine Aufwand-Nutzen-Betrachtung ein ungünstiges Verhältnis ergeben dürfte.

3.7 Auswahl einer Repräsentation

Nach Abwägung der Vor- und Nachteile überwiegen die Vorteile von XML, so dass XML als Medium für UML-Diagramme in dieser Arbeit gewählt wird.

Weiterhin besteht aber noch das Problem, dass die UML-Diagrammdaten in XML in einer geeigneten Form gespeichert werden müssen.

Durch Verwendung des vereinigten Metamodells ist jedoch noch nichts genaueres über seine Repräsentation gesagt. Es könnte mit MOF definiert und in einer XML-Konformen Syntax gespeichert werden (wie XMI). Da ein vereinigtes Metamodell auch die Repräsentation von UML-Diagrammen erfordert, ist die Integration von XMI in die Repräsentation dieses Metamodells naheliegend und gibt somit auch die Syntax-Basis XML vor (die genauer mit DTD's oder XML-Schemata beschrieben werden können).

Wenn man sich in anbetracht der Situation, dass noch kein Standard für ein vereintes Metamodell absehbar ist, favorisiere ich XMI und ein ergänzendes XML Format (z.B. PGML für die grafischen Aspekte) als Repräsentation.

Kapitel 4

Austausch von UML-Diagrammen

Kapitel 4 ermittelt aus den Aufgaben von UML-Diagrammen in Softwareprozessen (aus Kapitel 2) und der Auswahl einer Repräsentation von Diagrammen (aus Kapitel 3), ob XML für den Austausch von UML-Diagrammen geeignet ist.

4.1 Austausch von UML-Diagrammen

Beim Austausch von Daten im allgemeinen, besteht manchmal die berechtigte Befürchtung, dass Teile der übermittelten Informationen (besonders in heterogenen Netzen durch Transformation der Daten) verändert werden oder sogar verloren gehen. Um diesen möglichen Informationsverlust beim Austausch von UML-Diagrammen im besonderen, zu verhindern, gibt es verschiedene Strategien:

Zum einen kann das Single-Source-Prinzip Verwendung finden, zum anderen kann bei Peer-to-Peer Netzen eine gemeinsame Sprache oder ein einheitliches Format eingesetzt werden.

Das Single-Source-Prinzip

Bei Verwendung des Single-Source-Prinzips[1] [Rechenberg 99, 787] findet, technisch gesehen, kein Austausch von Diagrammen oder Modellen statt, denn es wird lediglich auf Teile des gemeinsamen Modells zugegriffen. Logisch betrachtet findet aber dennoch ein Austausch statt, denn die Modelle die von einem Werkzeug erstellt und geändert werden, sind von anderen Werkzeugen zugreifbar. Da dieser

[1] Das sog. Single-Source-Prinzip vermeidet unnötige Redundanzen und vereinfacht den Zugriff auf Daten dadurch, dass es für die Daten nur eine Quelle gibt. Dies impliziert eine Client-Server-Architektur, wobei die Daten auf einem Server gehalten werden und von evtl. verteilten Clients abgerufen werden können.

Zugriff auf die geänderten Daten erfolgt, könnte man hier von einem indirektem Austausch sprechen, wenn die Änderungen den abhängigen Werkzeugen signalisiert wird.

Die XML-Dokumente stellen somit eine Art zentrales Repository für die Modelldaten und der Programmtexte dar.

Somit tauschen die Clients keine Modelldaten direkt aus. Die Anbindung unterschiedlicher Clients (auf unterschiedlichen Plattformen beispielsweise) wird durch transformierbare Sprachen wie SGML oder XML begünstigt (für XML existiert der Standard XSLT für die Transformationen in andere Sprachen z.B. HTML). Um aber alle Daten der Modellierung und Implementation gemeinsam speichern zu können ist es notwendig, ein vereinigtes Metamodell zu verwenden, wenn unnötige Redundanzen vermieden werden sollen.

Peer-to-Peer Netze

Da die Verwendung des Single-Source-Prinzips nicht vorausgestzt werden kann, soll auch die konzeptionell komplementäre Peer-to-Peer[2] Topologie betrachtet werden.

Bei Verwendung einer Peer-to-Peer Architektur findet ein direkter Daten-Austausch zwischen den einzelnen Knoten statt.

Um UML-Diagramme bei Peer-toPeer Netzen austauschen zu können, muss entweder ein gemeinsames Format bzw. eine gemeinsame Sprache verwendet, oder es müssen Konvertiertungen vorgenommen werden. Im letzteren Fall müssten bei n unterschiedlichen Formaten $n \cdot (n-1)$ Konvertierungsalgorithmen vorhanden sein, sofern kein Basis-Format exisiert (von dem aus konvertiert wird). Ist ein solches Basis- oder Austausch-Format vorhanden, stellt sich natürlich die Frage, ob dieses Format nicht als Grundlage zum Austausch in Frage kommt und somit die Konvertierungen wegfallen könnten.

Um den hohen Entwicklungsaufwand von Konvertern zu vermeiden, sollte am besten ein gemeinsames Format eingesetzt werden.

Wenn das Entwicklungs-Umfeld es zulässt, sollte man der Single-Source Variante den Vorzug geben, denn die zentrale Lagerung der Daten vermeidet unnötige Redundanzen und die Zugriffe erfolgen auf aktuellen Daten. Ist ein solches Umfeld nicht möglich kann auf eine Peer-to-Peer oder ähnliche Architektur (mit den erwähnten Einschränkungen) eingesetzt werden.

[2]Peer-to-Peer (P2P) beschreibt im Gegensatz zum Client/Server-Modell eine Netzwerk-Architektur, wo Teilnemer Ressourcen des eigenen Rechners anderen zur verfügung stellen und auf Ressourcen anderer Rechner zugreifen kann. Jeder Knoten (oder Peer) ist sowohl Server als auch Client.

Austausch auf unterschiedlichen Ebenen

Auf unterschiedlichen Ebenen der Entwicklungstätigkeiten sind die Schwerpunkte der benötigten Informationen aus den verwendeten Diagrammen unterschiedlich.

Wärend z.B. Klassendiagramme bei der Implementation (neben der System-Struktur) auch die genauen Methoden-Signaturen interessant sind, ist dieses beim Design oft noch nicht der Fall. Diese asymetrischen Informations-Schwerpunkte können dazu führen, dass die Modelle (die durch die Diagramme visualisiert werden) in Teilen unterspezifiziert sein können. Diese Lücken sollten für die weitere Verwendung möglichst geschlossen werden. Nach dem Schliessen dieser Lücken sollten ebenfalls die vorgenommenen Änderungen an alle abhängigen Diagramme und Werkzeuge weitergereicht werden. Dieses Weiterreichen der Änderungs-Informationen kann, besonders bei Dokumenten-Basierter Kopplung der vorhandenen Entwicklungs-Werkzeuge, schwierig bis unmöglich sein.

Eine (wenn auch ineffiziente) Lösung dieses Problems könnte durch regelmässiges Abfragen (polling) nach Änderungen erreicht werden, wenn keine Event-Mechanismen zur Verfügung stehen.

Austausch auf gleicher Ebene

Beim Austausch von UML-Diagrammen auf der gleichen Ebene der Entwicklungstätigkeiten, entstehen die asymetrischen Informations-Schwerpunkte nicht und ist deshalb weniger problematisch. Jedoch bleibt auch hier die Problematik, des Weiterreichens von Änderungen an andere Werkzeuge.

Gemeinsames Format XMI

Alle in Kapitel 2 gennanten Entwicklungs-Tätigkeiten in Softwareprozessen können UML-Diagramme sinnvoll einsetzen (s. Kapitel 2.3). Daraus folgt, dass alle erwähnten Diagrammarten repräsentiert werden müssen. Dies ist, ohne die grafischen Anteile, mit XMI der Fall.

Die Problematik der Weiterleitung der Änderungs-Informationen ist nicht Teil von XMI und wäre auch nicht zielführend, denn Informationen über Änderungen in XMI-Dokumente zu speichern würde das Problem der Synchronisation der beteiligten Werkzeuge nicht lösen. Da die Lösung dieses Problems ausserhalb eines gemeinsamen Formats liegt, ist dieses somit kein Nachteil von XMI selbst und spricht nicht gegen den Einsatz von XMI als Austausch-Format.

4.2 Ein Use-Case als XMI-Dokument

Wie ein UML-Diagramm in XMI (und somit auch in XML) aussieht und aufgebaut ist, soll der folgende Abschnitt zeigen, um den von XMI abgedeckten Umfang des Diagrammes zu demonstrieren.

Am Beispiel einer einfach dargestellten Überweisung soll die Umsetzung in XMI gezeigt werden:

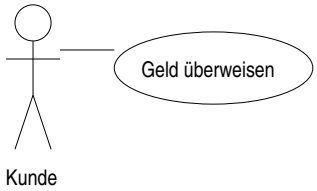

Abb. 4.2.1-1 Use-Case: Überweisung

Der Use-Case in Abb. 4.2.1-1 sieht in XMI (generiert von ArgoUML[3] Version 0.9.5), gekürzt und abschnittsweise behandelt, folgendermaßen aus:

Der XML-Vorspann und die XMI Deklaration des Datenmodells **Bank**:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  [...]
  <XMI.content>
    <Model_Management.Model xmi.id="xmi.1"
    xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff6">
      <Foundation.Core.ModelElement.name>Bank</Foundation.Core.ModelElement.name>
      [...]
```

Die Definition der enthaltenen Elemente:

```
      <Foundation.Core.Namespace.ownedElement>
```

Der Akteur: **Kunde:**

```
        <Behavioral_Elements.Use_Cases.Actor
        xmi.id="xmi.2" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff5">
          <Foundation.Core.ModelElement.name>Kunde
          </Foundation.Core.ModelElement.name>
          [...]
        </Behavioral_Elements.Use_Cases.Actor>
```

Der Use-Case: **Geld überweisen:**

```
        <Behavioral_Elements.Use_Cases.UseCase
        xmi.id="xmi.3" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff4">
          <Foundation.Core.ModelElement.name>Geld Überweisen
          </Foundation.Core.ModelElement.name>
          [...]
        </Behavioral_Elements.Use_Cases.UseCase>
```

[3] Argo UML ist ein javabasiertes OpenSource (BSD Lizenz) UML-Werkzeug.

Die 1:1 Assoziation die Akteur und Use-Case verdindet:

```
        <Foundation.Core.Association
        xmi.id="xmi.4" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff3">
          [...]
          <Foundation.Core.Association.connection>
            <Foundation.Core.AssociationEnd xmi.id="xmi.5">
              [...]
              <Foundation.Core.AssociationEnd.multiplicity>
                <Foundation.Data_Types.Multiplicity xmi.id="xmi.6">
                  <Foundation.Data_Types.Multiplicity.range>
                    <Foundation.Data_Types.MultiplicityRange xmi.id="xmi.7">
                      <Foundation.Data_Types.MultiplicityRange.lower>1
                      </Foundation.Data_Types.MultiplicityRange.lower>
                      <Foundation.Data_Types.MultiplicityRange.upper>1
                      </Foundation.Data_Types.MultiplicityRange.upper>
                    </Foundation.Data_Types.MultiplicityRange>
                  </Foundation.Data_Types.Multiplicity.range>
                </Foundation.Data_Types.Multiplicity>
              </Foundation.Core.AssociationEnd.multiplicity>
              <Foundation.Core.AssociationEnd.association>
                <Foundation.Core.Association xmi.idref="xmi.4"/>
              </Foundation.Core.AssociationEnd.association>
              <Foundation.Core.AssociationEnd.type>
                <Foundation.Core.Classifier xmi.idref="xmi.2"/>
              </Foundation.Core.AssociationEnd.type>
            </Foundation.Core.AssociationEnd>
            <Foundation.Core.AssociationEnd xmi.id="xmi.8">
              <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
              <Foundation.Core.AssociationEnd.isNavigable xmi.value="true"/>
              <Foundation.Core.AssociationEnd.multiplicity>
                <Foundation.Data_Types.Multiplicity xmi.idref="xmi.6"/>
              </Foundation.Core.AssociationEnd.multiplicity>
              <Foundation.Core.AssociationEnd.association>
                <Foundation.Core.Association xmi.idref="xmi.4"/>
              </Foundation.Core.AssociationEnd.association>
              <Foundation.Core.AssociationEnd.type>
                <Foundation.Core.Classifier xmi.idref="xmi.3"/>
              </Foundation.Core.AssociationEnd.type>
            </Foundation.Core.AssociationEnd>
          </Foundation.Core.Association.connection>
        </Foundation.Core.Association>
      </Foundation.Core.Namespace.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

Da Assoziationen in XML potentiell auch die Kardinalität n:m haben können, ist deren Definition auch komplexer. Umfangsteigernd kommt hinzu, dass sowohl n als auch m nicht nur einfache Zahlen sein müssen, sondern auch Bereiche angeben können z.B. „1..3“. Von diesen Bereichen kann es auch noch mehrere pro Assoziationsende geben.

Eine zusammenhängende und vollständige Version dieses XMI-Dokumentes befindet sich im Anhang B.

Wie aus diesem Beispiel ersichtlich ist, werden die grafischen Aspekte des UML-Diagramms nicht erfasst (Position, Grösse, Farbe etc.). Dieses ist bei der Verwendung in nicht grafischen Kontexten unkritisch, d.h. wenn Werkzeuge nur auf

Modelldaten zugreifen ohne, das die optische Ausrichtung und Anordnung eine Rolle spielt z.B ein Programmierwerkzeug, dass Klassenbezeichnung, Attribute usw. den Klassen-Diagramm-Daten entnimmt.

Werkzeuge die UML-Diagramme zumindest anzeigen, müssen die grafischen Daten entweder gesondert speichern oder ein Auto-Layout-Mechanismus muss die Darstellung übernehmen. Da automatisches Layout aber noch nicht in jedem Falle zufriedenstellende Ergebnisse liefert, ist die erste Alternative zu bevorzugen. Die grafischen Daten können ebenfalls in XML gespeichert werden z.B. in SVG[4] oder PGML[5]. ArgoUML speichert die grafischen Aspekte der UML-Diagramme in PGML-Dokumente.

4.3 Eignung von XML zur Repräsentation von UML-Diagrammen

Mit XMI existiert bereits ein Standard der XML als Repräsentation von UML-Modellen einsetzt und von einigen UML-Werkzeugen unterstützt wird (z.B ArgoUML).

In XML-Dokumenten werden lediglich Namen und Wertepaare in einer Hierarchie angeordnet. Diese Struktur ist mit ihrer hierarchischen Ordnung zwar geeignet um komplexe Daten zu speichern, wie es bei UML-Diagrammen der Fall ist, jedoch sind die Definitionen von n:m Beziehungen in Hierarchien umfangreicher. Das Beispiel im vorherigen Abschnitt hat dieses gezeigt.

XML bietet die nötige Infrastruktur um komplexe Daten zu speichern, lesen und zu verändern. XML erlaubt es die darzustellenden Daten zu strukturieren. Die Weiterverarbeitung der Daten wird durch Standards, die auf XML aufsetzen oder verwenden, unterstützt. Dieses Fördert die Verwendung der Diagramm-Informationen in Reports oder anderen Dokumentationen. Die größte Einschränkung die XML den zu speichernden Daten auferlegt, ist eine hierarchische Struktur. Bei Daten die eine Netz-Struktur aufweisen, muss diese in eine Baum-Struktur umgesetzt werden. XMI zeigt, dass es möglich ist, UML-Modelle in der hierarchischen Syntax von XML darzustellen. XML erfüllt somit die notwendige Voraussetzung um UML-Diagramme repräsentieren zu können und wird in Form von XMI in Kombination mit PGML von ArgoUML bereits eingesetzt.

[4] **S**calable **V**ector **G**raphics (SVG) beschreibt zweidimensionale Vektorgrafiken in XML und ist ein vom W3C veröffentlichter Standard.

[5] **P**recision **G**raphics **M**arkup **L**anguage (PGML) ist eine auf XML basierende Auszeichnungssprache für leichtgewichtige zweidimensionale Vektorgrafiken, die das gemeinsame Darstellungsmodell von PostScript und **P**ortable **D**ocument **F**ormat (PDF) verwendet und erweitert. PGML ist ein vom W3C publizierter Standard.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

UML spielt bei der Softwareentwicklung eine immer größere Rolle, denn die Software wird immer komplexer und dadurch auch unübersichtlicher, so dass ab einem gewissen Punkt auf Visualisierung kaum noch verzichtet werden kann. Da UML mehrere etablierte Ansätze vereint und sich als Standard etabliert hat, wird es von vielen Modellierungswerkzeugen unterstützt und eingesetzt.

UML liegt eine komplexe Struktur zugrunde, die mit XML repräsentiert werden kann. Der Standard XMI hat dieses, jedoch ohne grafische Anteile, gezeigt. Auch wird der Datenaustausch und die Weiterverarbeitung durch Standards im Umfeld von XML unterstützt.

Die betrachteten alternativen Technologien erfordern z.Zt. entweder einen höheren Programmieraufwand oder unterstützen kaum Standards die für den Austausch und die Weiterverarbeitung von UML-Modellen in verteilten bzw. unterschiedlichen Entwicklungswerkzeugen in heterogenen Netzen benötigt werden.

5.2 Ausblick

Die Aktivitäten rund um XML sind sehr rege, so dass sich immer neue Einsatzgebiete und Schnittstellen finden, die einen breiteren Einsatz dieser Technologie zulassen.

Im Rahmen einer Diplomarbeit arbeitet Stefan Müller an einem Metamodell, das auf XMI basiert und u.a. um grafische Informationen ergänzt wird, um dieses Defizit von XMI zu beseitigen.

Die UML-Entwicklung nimmt auch weiter ihren Lauf und wird in Zukunft wohl

außer Stereotypen und „tagged values“ (also benutzerdefinierte Bezeichnungen an UML-Diagramm-Elemente) durch Profile noch weiter auf die Bedürfnisse der unterschiedlichen Entwicklergruppen eingehen. Auch technisch bieten Profile Vorteile, die sich in integrierten Systemen nutzen lassen, in dem ein für diesen Einsatz spezialisiertes Profil verwendet wird.

Die Weiterentwicklung von UML und XML-basierten Standards wird ein Verschmelzen von visueller Modellierung und Implementation weiter vorantreiben und für Entwickler die Arbeit sicherlich vereinfachen.

Anhang A

UML

Die Unified Modelling Language (UML) hat es sich zur Aufgabe gemacht, möglichst den gesamten Softwareprozess mit grafischen Elementen zu unterstützen. Dieses erfolgt durch verschiedene Diagramme, die den unterschiedlichen Kontexten und Aufgaben im Softwareprozess angepasst sind.
In UML 1.3 [OMG 2000b] sind die folgenden Diagrammarten definiert:

- Use-Cases
- Klassen-Diagramme
- Interaktions-Diagramme
  - Sequenz-Diagramme
  - Kollaborations-Diagramme
- Package-Diagramme
- Zustands-Diagramme
- Aktivitäts-Diagramme
- Implementierungs-Diagramme
  - Komponenten-Diagramme
  - Deployment-Diagramme

Die einzelnen Diagramme werden in den folgenden Abschnitten kurz vorgestellt.

A.1 Use-Cases

Die Use-Cases wurden zunächst von I. Jacobson eingeführt[Jacobson 92], sie stellen das Zusammenwirken von Personen mit einem System dar. Die Personen werden als *Aktoren* bezeichnet und sind nicht nur auf Menschen beschränkt, sondern können auch Systeme darstellen. Wobei zu beachten ist, dass mit Personen meist Rollen gemeint sind.

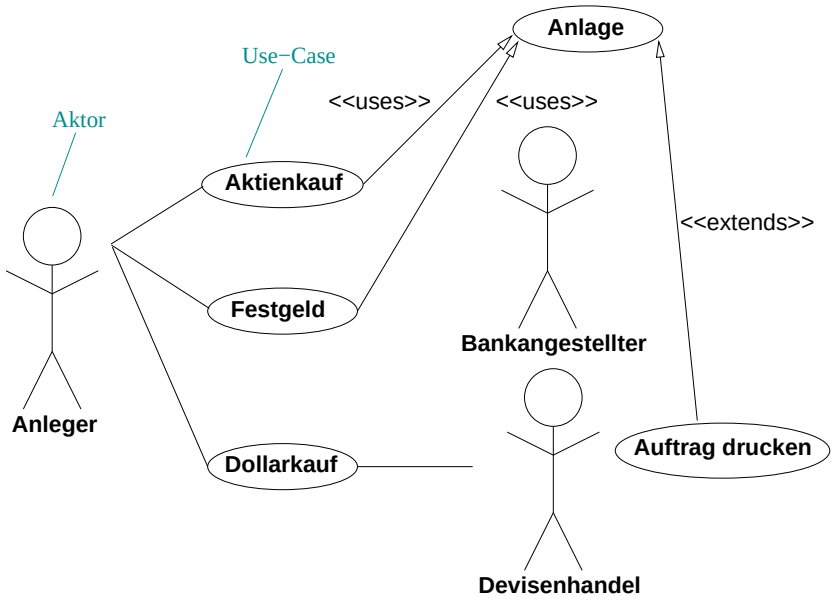

Abb. A.1-1 Use-Case: Geldanlage

Ein Use-Case ist eine Handlung eines Akteurs, wie z.B. Aktienkauf in Abb. A.1-1. Use-Cases werden als Ellipsen dargestellt. Use-Cases und Akteure werden mit Linien verbunden, um zu zeigen, welche Akteure an welchen Handlungen beteiligt sind.

In Use-Case-Diagrammen gibt es noch zwei weitere Verbindungstypen: < <uses> > und < <extends> >.

Der Verbindungstyp < <uses> > wird verwendet, wenn zwei oder mehr Use-Cases einen ähnlichen Aufbau haben und Verhalten durch kopieren wiederholt dargestellt werden muss *(Anlage* und *Festgeld* in Abb. A.1-1). Dadurch ist es möglich, Verhalten von Use-Cases in einen anderen Use-Case zu verlagern.

Der Verbindungstyp < <extends> > klammert ebenfalls Verhalten aus, jedoch mit einer anderen Zielsetzung. Bei < <extends> > wird das Verhalten erweitert. Die Pfeilrichtung zeigt an, welche Use-Cases erweitert werden. Im Beispiel oben wird *Anlage* um *AuftragDrucken* erweitert.

A.2 Klassen-Diagramme

Der zentrale Bestandteil von UML sind die Klassen-Diagramme, die Klassen, Objekte und deren statische Struktur und Beziehungen zueinander zeigen.

Objekte unterscheiden sich in dieser Darstellung von Klassen lediglich dadurch, dass die Bezeichnung unterstrichen ist.

Das Beispiel zeigt eine stark vereinfachte automatische Geldausgabe einer Bank.

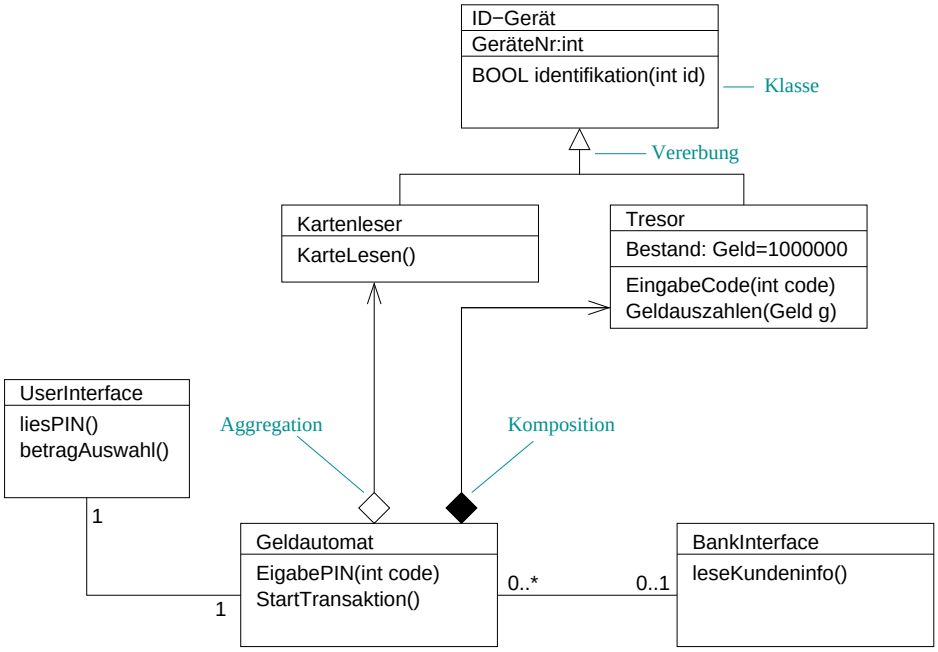

Abb. A.2-1 Klassen-Diagramm: Geldausgabe am Automaten [Wahl 98]

Eine *Klasse* wird als Rechteck dargestellt (z.B. Kartenleser). Klassen werden durch Linien miteinander verbunden, die folgende Semantik aufweist:

- *Assoziationen* (einfache Linien, siehe UserInterface und Geldautomat) stellen allgemeine Beziehungen zwischen Klassen oder Objekten dar. Es können Kardinalitäten[1] in Form von Zahlen oder Zahlenbereichen angegeben werden, die die Anzahl der in Beziehung stehenden Objekte bestimmen. Des weiteren kann ein Pfeil verwendet werden um anzuzeigen, dass Zugriffe nur in Pfeilrichtung erfolgen können (auch Navigationsfähigkeit genannt, engl. Navigability).

1 0..1	1:0,1
1 ∗	1:n≤0
n m	n:m

[1]Die Kardinalitäten werden folgendermaßen gekennzeichnet: 1: genau Eins, a..b: von a bis b wobei a und b beliebige natürliche Zahlen (inkl. Null) sein können, ∗: Null oder größer.

◇— Aggregation

- *Aggregationen* (Linien mit einer Raute, siehe Kartenleser und Geldautomat) sind Assoziationen die zusätzlich angeben, dass Klassen in anderen enthalten sein müssen (Ist-Teil-von-Beziehung).

◆— Komposition

- *Kompositionen* (Linien mit einer ausgefüllten Raute, siehe Tresor und Geldautomat) sind Aggregationen stärkerer Form, die zusätzlich das wirkliche Enthaltensein der Klasse fordern[2].

⇑ ⊥ Vererbung

- *Vererbungen* (Linien mit einer dreieckigen Pfeilspitze[3], siehe ID-Gerät und Kartenleser) stellen *Spezialisierungen* bzw. *Generalisierungen* von Klassen dar. Das heißt Kartenleser spezialisieren ID-Geräte und umgekehrt generalisieren ID-Geräte die Kartenleser. Die dreieckige Pfeilspitze gibt also die generalisierte Klasse an.

Eine Klasse besteht aus Attributen und Methoden. Die Attribute enthalten Daten und definieren damit die Eigenschaften bzw. den Zustand der Objekte. Die Methoden können den Zustand sondieren oder ändern, d.h. sie können auf die Attribute zugreifen, und stellen somit die Dienste der Klasse zur Verfügung.

Die *Sichtbarkeit* der Methoden und Attribute bestimmt, wer auf welche Daten zugeifen darf.

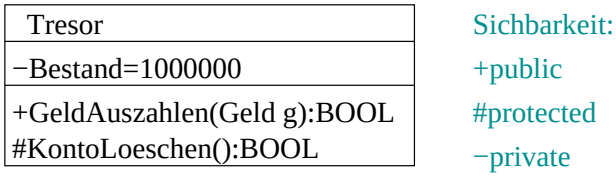

Abb A.2-2 Aufbau einer Klasse

A.3 Interaktions-Diagramme

Interaktions-Diagramme beschreiben die dynamische Struktur bzw. die zeitlichen Abläufe (Ablaufsequenzen) des Modells.

Sequenz-Diagramme und Kollaborations-Diagramme beschreiben die Menge der Interaktions-Diagramme und sollen in nächsten beiden Abschnitten kurz vorgestellt werden.

[2] Bei Aggregationen kann es sich auch um Referenzen handeln.

[3] Oder bei Mehrfachvererbungen kann ein Dreieck an der Verbindung der generalisierten Klasse an der Verzweigung zu den anderen Klassen gezeichnet werden.

A.3.1 Sequenz-Diagramme

Diese Diagrammart stellt dar, welche Objekte wann und womit auf andere zugreifen.

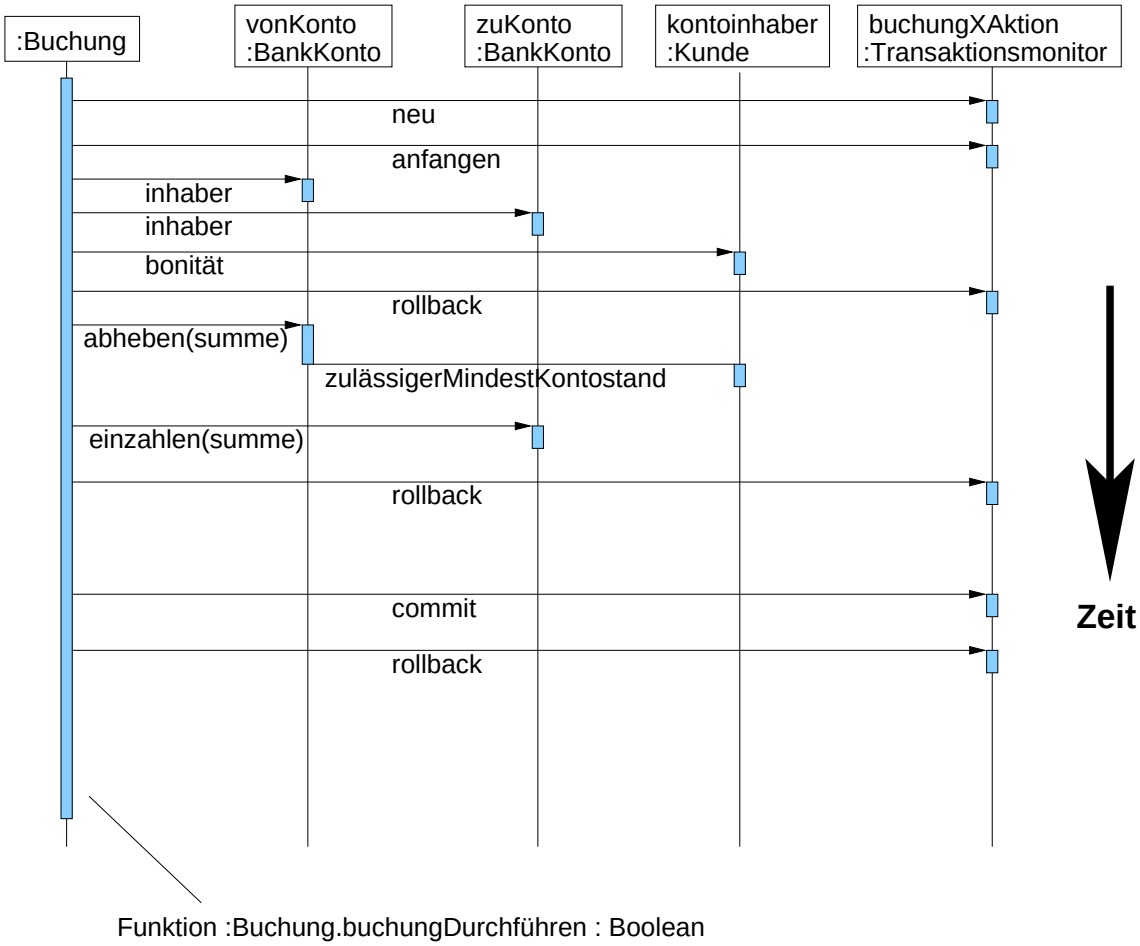

Abb. A.3.1-1 Sequenz-Diagramm: Buchung

Das Beispiel in Abb. A.3.1-1 zeigt einen Buchungsvorgang in einer Bank. Die Rechtecke stehen für Objekte. Die beschrifteten Pfeile zeigen an, welche Operation wo ausgeführt wird (die Pfeilspitze zeigt auf das ausführende und das andere Ende auf das aufrufende Objekt). Die gefüllten Rechtecke sind Operationen, deren vertikale Länge und Position proportional zu der Ausführungszeit ist. Die senkrechten Linien (Lebenslinien) geben an, wie lange das Objekt im beobachteten Zeitraum existiert.

Die Sequenz beginnt mit dem Namenlosen *Buchungs*-Objekt. Die Funktion *„buchungDurchführen“* (der lange senkrechte Balken) übernimmt den Transfer von Geld von einem Konto (*vonKonto*) zum anderen (*zuKonto*).

A.3.2 Kollaborations-Diagramme

In Kollaborations-Diagrammen werden Objekte, die Nachrichten austauschen, als Rechteck dargestellt, die den Namen des Objektes tragen.

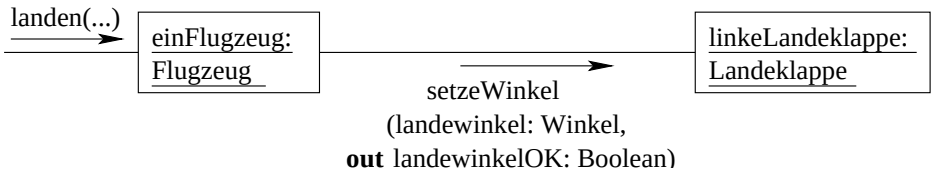

Abb. A.3.2-1 Kollaborations-Diagramm: Setze Landeklappenwinkel

Die unterstrichenen Namen geben an, dass es sich hierbei um Exemplare und nicht um die Klassen selbst handelt.

A.4 Package-Diagramme

Mit *Packages* (Pakete) werden Diagramme oder Elemente zusammengefasst. Abhängigkeiten von Packages (Kästen mit kleinen Rechtecken an der linken oberen Ecke) werden durch gestrichelte Pfeile signalisiert, wobei die Pfeilrichtung angibt, welches Package evtl. geändert oder neu übersetzt werden muss, wenn sich das Package am anderen Ende des Pfeils verändert.

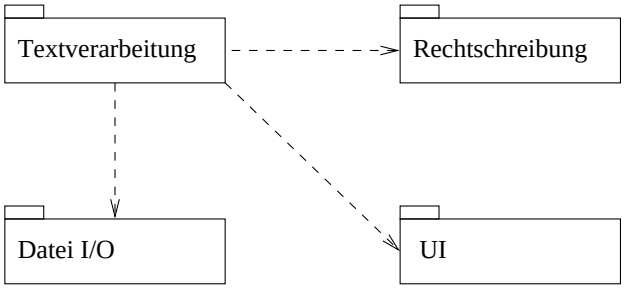

Abb. A.4-1 Package-Diagramm: Anwendungs-System

A.5 Zustands-Diagramme

Mögliche Zustände von Klassen und deren Übergänge werden in Zustandsdiagrammen dargestellt.

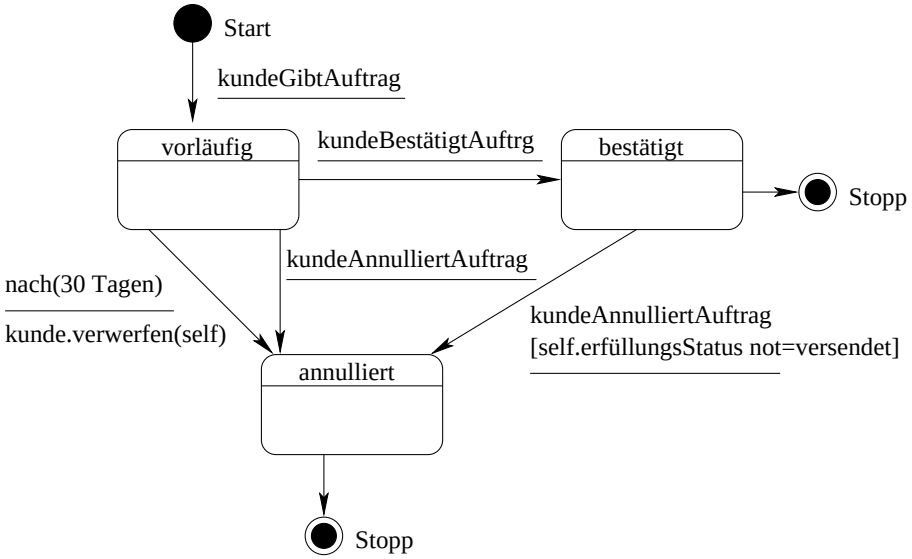

Abb. A.5-1 Zustands-Diagramm: KundenAutorisationsStatus

Die abgerundeten Kästchen sind mögliche Zustände der abgebildeten Klasse (Beispiel in Abb. A.5-1). Jeder Übergang wird mit einem Pfeil markiert, der mit einem zweiteiligen Text beschriftet ist (durch eine Linie getrennt). Oberhalb der Linie steht ein *Ereignis*, welches eine Aktion auslösen kann, die (sofern vorhanden) unterhalb der Linie steht. Ein Ereignis kann auch ein Boolsches Prädikat sein, das dann in eckigen Klammern steht.

A.6 Aktivitäts-Diagramme

Für nicht zu komplexe Problemstellungen, können Aktivitäts-Diagramme für parallele Prozesse oder zur Modellierung von Arbeitsabläufen (dynamisches Verhalten) verwendet werden.

Folgende Elemente sind in dieser Diagrammart enthalten:

- Aktivitäten
- Transitionen
- Synchronisationslinien
- „Swimlanes“ bzw. Schwimmbahnen (optional)

In Aktivitäts-Diagrammen werden die Objekte eines Vorganges mittels der Aktivitäten beschrieben, die sie während des Ablaufes ausführen.

Eine Aktivität ist ein einzelner auszuführender Schritt.

Eine Transition ist der Übergang zweier Aktivitäten. Beim Erreichen der Transition ist die vorhergehende Aktivität abgeschlossen.

Bei Synchronisationslinien wird auf alle eingehenden Transitionen gewartet bzw. alle ausgehenden werden gleichzeitig gestartet.

Die optionalen „Swimlanes“ begrenzen Klassen, d.h. alles innerhalb einer Schwimmbahn gehört zu einer Klasse, deren Name am oberen Rand steht.

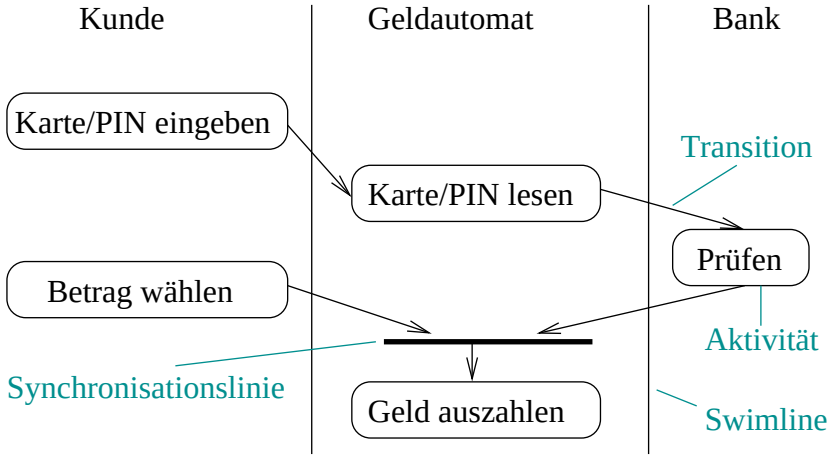

Abb. A.6-1 Aktivitäts-Diagramm: Geldausgabe am Automaten

A.7 Implementierungs-Diagramme

Aspekte der Implementierung, wie Programmtext-Struktur und Struktur des Systems zur Ausführungszeit, können durch Implementierungs-Diagramme visualisiert werden.

Es gibt zwei Arten von Implementierungs-Diagrammen:

- Komponenten-Diagramme
- Deployment-Diagramme

A.7.1 Komponenten-Diagramme

Abhängigkeiten zwischen Programmtext, Binärcodekomponenten und ausführbaren Programmen können mit Komponenten-Diagrammen modelliert werden. Die Darstellungen haben nur Typencharakter und stellen somit keine Exemplare dar.

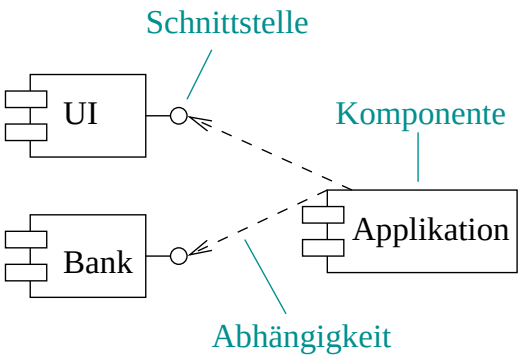

Abb. A.7.1-1 Komponenten-Diagramm: Bankanwendungs-System

A.7.2 Deployment-Diagramme

Um Hardware und deren Verbindungen zu visualisieren, können Deployment-Diagramme[4] verwendet werden, wobei zu beachten ist, dass hier Individuen bzw. Exemplare Verwendung finden und keine Klassen.

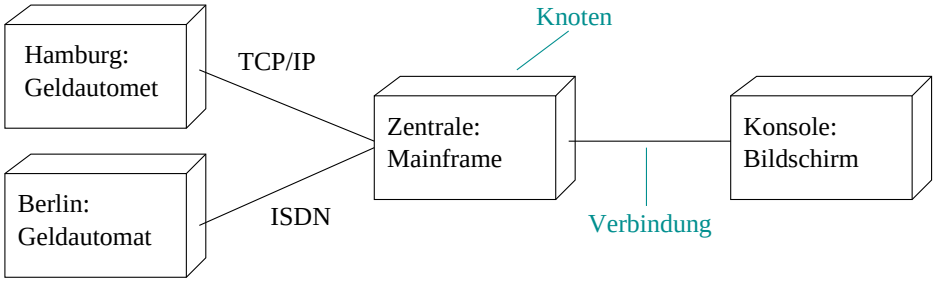

Abb. A.7.2-1 Deployment-Diagramm: Geldautomaten

Die *Knoten* stellen Hardware- oder Verarbeitungseinheiten dar. Zwischen Knoten existieren *Verbindungen*, die physikalische Kommunikationspfade repräsentieren (siehe Beispiel in Abb. A.7.2-1).

[4]Der englische Begriff deployment steht in diesem Zusammenhang für Einführung

Anhang B

Beispiel Use-Case in XMI

Vollständige Datei bank.xmi generiert von ArgoUML (Verison 0.9.5).

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Novosoft UML Library</XMI.exporter>
    <XMI.exporterVersion>0.4.19</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
  <XMI.content>
    <Model_Management.Model xmi.id="xmi.1"
    xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff6">
      <Foundation.Core.ModelElement.name>BankModel</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
      <Foundation.Core.Namespace.ownedElement>
        <Behavioral_Elements.Use_Cases.Actor
        xmi.id="xmi.2" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff5">
          <Foundation.Core.ModelElement.name>Kunde
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
        </Behavioral_Elements.Use_Cases.Actor>
        <Behavioral_Elements.Use_Cases.UseCase
        xmi.id="xmi.3" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff4">
          <Foundation.Core.ModelElement.name>Geld Überweisen
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
```

```
          </Foundation.Core.ModelElement.namespace>
        </Behavioral_Elements.Use_Cases.UseCase>
        <Foundation.Core.Association
        xmi.id="xmi.4" xmi.uuid="-64--96-0-20-e4fd7:ec4c54d5d8:-7ff3">
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Foundation.Core.Association.connection>
            <Foundation.Core.AssociationEnd xmi.id="xmi.5">
              <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
              <Foundation.Core.AssociationEnd.isNavigable xmi.value="true"/>
              <Foundation.Core.AssociationEnd.multiplicity>
                <Foundation.Data_Types.Multiplicity xmi.id="xmi.6">
                  <Foundation.Data_Types.Multiplicity.range>
                    <Foundation.Data_Types.MultiplicityRange xmi.id="xmi.7">
                      <Foundation.Data_Types.MultiplicityRange.lower>1
                      </Foundation.Data_Types.MultiplicityRange.lower>
                      <Foundation.Data_Types.MultiplicityRange.upper>1
                      </Foundation.Data_Types.MultiplicityRange.upper>
                    </Foundation.Data_Types.MultiplicityRange>
                  </Foundation.Data_Types.Multiplicity.range>
                </Foundation.Data_Types.Multiplicity>
              </Foundation.Core.AssociationEnd.multiplicity>
              <Foundation.Core.AssociationEnd.association>
                <Foundation.Core.Association xmi.idref="xmi.4"/>
              </Foundation.Core.AssociationEnd.association>
              <Foundation.Core.AssociationEnd.type>
                <Foundation.Core.Classifier xmi.idref="xmi.2"/>
              </Foundation.Core.AssociationEnd.type>
            </Foundation.Core.AssociationEnd>
            <Foundation.Core.AssociationEnd xmi.id="xmi.8">
              <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
              <Foundation.Core.AssociationEnd.isNavigable xmi.value="true"/>
              <Foundation.Core.AssociationEnd.multiplicity>
                <Foundation.Data_Types.Multiplicity xmi.idref="xmi.6"/>
              </Foundation.Core.AssociationEnd.multiplicity>
              <Foundation.Core.AssociationEnd.association>
                <Foundation.Core.Association xmi.idref="xmi.4"/>
              </Foundation.Core.AssociationEnd.association>
              <Foundation.Core.AssociationEnd.type>
                <Foundation.Core.Classifier xmi.idref="xmi.3"/>
              </Foundation.Core.AssociationEnd.type>
            </Foundation.Core.AssociationEnd>
          </Foundation.Core.Association.connection>
        </Foundation.Core.Association>
      </Foundation.Core.Namespace.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

Literaturverzeichnis

[Anderson 2000] R. Anderson et al.: *XML Professionell*. MITP-Verlag, 2000.

[Baier 2000] Tobias Baier: *Ein Metamodell für eine UML-basierte Entwicklungsumgebung für verteilte und nebenläufige Systeme.* Diplomarbeit, Universität Hamburg, 2000.

[Floyd 89] C. Floyd, F.-M. Reisin, G. Schmidt: *STEPS to software development with users*. In C. Ghezzi, J. A. McDermid (eds.): ESEC'89. Lecture Notes in Computer Science Nr. 387. Berlin: Springer-Verlag 1989.

[Gamma 98] E. Gamma et al.: *UML Handbook*, 1998.

[Goldfarb 98] Charles F. Goldfarb, Paul Prescod: *The XML Handbook*. Prentce Hall 1998.

[gs 2001] gs: *UML Modelle erhalten einheitlichen Standard*. Computer Zeitung Nr. 13 / 29. März 2001

[Jacobson 92] I. Jacobson: *Object-Oriented Software Engeneering, A Use Case driven Approach*. Addison-Wesley 1992.

[OMG 99] Object Management Group: *XML Metadata Interchange (XMI) Version 1.1*, 25. Oktober 1999.

[OMG 2000a] Object Management Group: *Meta Object Facility (MOF) Secification*, März 2000.

[OMG 2000b] Object Management Group: *Unified Modeling Language Specification Version 1.3,* März 2000.

[Page-Jones 99] Meilir Page-Jones: *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley 1999.

[Rational 98] Rational Software Corporation: *Rational Unified Process, Best Practices for Software Development Teams*. White Paper 1998.

[Rational 2000] Rational Software Corporation: *Rational Unified Process*. Broschüre 2001.

[Rechenberg 99] Peter Rechenberg und Gustav Pomberger: *Informatik-Handbuch*. Carl Hanser Verlag, 1999

[W3C 2000] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Second Edition)*, 6. Oktober 2000.

[W3C 2001a] World Wide Web Consortium: *XML Schema Part 1: Structures (W3C Recommendation*), 2. Mai 2001

[W3C 2001b] World Wide Web Consortium: *XML Schema Part 2: Datatypes (W3C Recommendation*, 2. Mai 2001

[Wahl 98] Günter Wahl: UML kompakt, OBJEKTspektrum 2/1998

[Züllighoven 98] Heinz Züllighoven: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. dpunkt-Verlag, 1998.