Softwaretechnik

Softwaretechnik wird am FB Informatik der Fachhochschule Darmstadt seit Mitte der achtziger Jahre gelehrt. Softwaretechnik war im auslaufenden Diplomstudiengang eine "4+2" Veranstaltung, d. h. sie setzt sich aus 4 Semesterwochenstunden (SWS) Vorlesung und 2 SWS Praktikum zusammen. Die Diplom-Prüfungsordnung des FB Informatik sah Softwaretechnik für das dritte Semester vor.

Im Bachelor-Studiengang findet Softwaretechnik 1 im 2. Semester mit 2 SWS statt. Softwaretechnik 2 ist mit 3+1 SWS im 3. Semester positioniert. Es sind Pflichtveranstaltungen für alle Studierenden des FB Informatik.

Die Lehrinhalte von Softwaretechnik führen, gemeinsam mit den Lehrinhalten von Datenbanken, zu der Vordiplom-Prüfungsleistung Software Engineering.

In Softwaretechnik wird versucht die Studierenden - nach einer Einführung in Grundbegriffe wie Software-Qualitätsmerkmale, Phasenmodelle usw. - mit relevanten Methoden der Software-Entwicklung vertraut zu machen. Die gelehrten Methoden werden im begleitenden Praktikum unter Einsatz von CASE-Tools praktisch angewendet.

Hinweise zum SWT-Praktikum

Die folgenden Seiten bieten einen kurzen, kompakten Überblick der Grundlagen, die zur Durchführung des SWT-Praktikums benötigt werden. Sie führen in die im Praktikum eingesetzte Notation, die UML (Unified Modeling Language) ein und beschreiben die Diagrammtypen der UML, die im Praktikum eingesetzt werden.
Die UML umfaßt eine Vielzahl von Diagrammen, die zur Beschreibung der unterschiedlichen Sichten (Abstraktionsebenen) auf ein System verwendet werden können. Die im Praktikum eingesetzten Diagrammtypen sind ausführlich beschrieben und zum leichteren Verständnis mit einem kleinen, kommentierten Beispiel versehen.
Für die Diagrammtypen, die im Praktikum wahrscheinlich nicht verwendet werden, ist lediglich eine kurze Beschreibung des Diagrammzwecks eingefügt.

Im Praktikum wird für die konstruktiven Phasen der Software-Entwicklung das CASE-Tool **Innovator** der Firma MID, Nürnberg eingesetzt. Zur Qualitätssicherung, Bewertung der Software-Qualität und Test-Unterstützung wird das Tool **Logiscope** der Firma Telelogic (früher Verilog) verwendet. Die praktische Benutzung der eingesetzten Tools ist ebenfalls beschrieben, so daß diese Seiten den Studierenden als theoretische und praktische Unterstützung während des SWT-Praktikums dienen können.

Softwaretechnik - Einleitung

Warum ist es so schwierig, Software zu erstellen?

Antworten (vgl. E. Denert:Software-Engineering (Springer 1991)) :

1. Softwaresysteme gehören zu den komplexesten Gebilden, die je von Menschen geschaffen wurden. Sie bestehen aus einer riesigen Zahl von Bauelementen

(Maschinenbefehlen und Datenspeichern), die in vielfältiger Art zusammenwirken. und eine zusätzliche Schwierigkeit: Software ist immateriell, eigentlich nur eine Idee, die nur im Geiste von Menschen existiert.
2. Software wird in Teams von Menschen hergestellt, deren Qualifikation oft zu wünschen übrig läßt.
3. Es existieren nur wenige geeignete Werkzeuge für die Entwicklung großer Softwaresysteme.

Was ist Softwaretechnik - Software-Engineering ?

Softwaretechnik ist die Anwendung wissenschaftlicher Erkenntnisse mit dem Ziel, Computer mittels Programmen, Verfahren und den zugehörigen Dokumenten dem Menschen nutzbar zu machen. Softwaretechnik besteht aus "Brainware" und "Toolware".

Brainware	Toolware
Prinzipien Methoden Verfahren Techniken	Entwicklungssystem Werkzeuge Hilfsmittel

Brainware und Toolware werden üblicherweise in Projekten von mehreren Personen angewendet.

Was ist ein guter Software-Ingenieur?

Er besitzt die Fähigkeit, gute Ideen zu fassen und sie anderen mitzuteilen. Er ist in der Lage, gute Konzepte zu finden und durchzusetzen, und er beherrscht die Werkzeuge, um diese Ideen in die formale Darstellung umzusetzen, die der Computer versteht.

Das Problem:
Firmen sind von Information abhängig und sind auf ständige Versorgung mit neuester Information angewiesen, sonst verlieren sie ihre Wettbewerbsfähigkeit. Jedoch wächst der Umfang der Information schneller als die Fähigkeit, diese Information sinnvoll zu nutzen. Bildlich gesprochen: Die Firmen ertrinken in den Daten. Deshalb brauchen sie Software zur Bewältigung der Information und zur dringend erforderlichen Verarbeitung und Aufbereitung der Daten.

Aber das Problem entsteht aufgrund folgender Tatsachen:

- Die Softwarekosten steigen - während die Hardwarekosten sinken.
- Die Softwareentwicklungszeiten und die Wartungszeiten steigen, während zur gleichen Zeit die Hardware Entwicklungszeiten fallen und die Hardwareentwicklungskosten sinken.
- Es gibt Massen von Softwarefehlern, während die Hardware nahezu fehlerfrei ist.
- Software wird nach einem sehr streng strukturierten Schema entwickelt, das sehr unflexibel ist.

Dies bestätigen Studien:

Entwicklungsphase	% Entwicklungs-kosten	% Der enststehen-den Fehler	% Der gefundenen Fehler	Relative Kosten der Fehler-korrektur
Anforderungs-definition	3	55	18	1.0
Entwurf	8	30	10	1.0-1.5
Realisierung	7	10		1.0-5.0
Test	15		50	1.0-5.0
Wartung	67	5	22	10-100

Quelle: Amerikanisches Verteidigungsministerium (aus Lee,Tepfenhart:UML and C++ (Prentice Hall, 1997))

Die Tabelle zeigt, daß 85% der Fehler dort gemacht werden, wo die Behebung sehr billig ist, nämlich bei Definition und Entwurf, daß aber viele unentdeckt bleiben und erst in späteren Phasen mit erheblichem Kostenaufwand entdeckt und beseitigt werden. Diese beiden Entwicklungsphasen zu verbessern, ist danach der kosteneffektivste Weg, um die Qualität der Software anzuheben.

Ein anderes Problem:
Ein großer Teil der Software ist überflüssig schon vor ihrer Auslieferung, weil sie nicht mehr den geänderten Bedingungen entspricht und Studien haben gezeigt, daß nur ca. 25% der Softwareprojekte zu Ergebnissen führen, die dann in einem System auch arbeiten. Das kann eigentlich nichts anderes bedeuten, als daß die Computersysteme bisher nicht die Anwender zufriedenstellen. Um dieses Problem zu lösen, müssen die Softwareentwickler zunächst die Bedürfnisse der Anwender verstehen lernen.

Was will der Anwender?

Er / sie will ein System, das ...

- die funktionalen Anforderungen erfüllt
- die ständigen Veränderungen, die im Einsatzgebiet stattfinden, bewältigen kann
- das die Zeit- und Speicherplatzbedingungen einhält.

Der Anwender will Software ...

- die leicht zu handhaben ist
- die ressourcenschonend ("effektiv") arbeitet
- die entworfen ist, mit Langlebigkeit als Konstruktionsziel

Die bisher verwendeten Methoden, Strukturierte Analyse und Design ("SA"), Datenmodellierung ("Entity-Relationship Modell") haben nicht die Erwartungen der Anwender erfüllt, und es werden heute große Erwartungen in ein neues Paradigma, die **Objektorientierte Systementwicklung,** gesetzt. Insbesondere die OO-Methodik unter Verwendung von UML ("Unified Modeling Language") wird von großen Erwartungen bei den Anwendern begleitet.

Warum ist das OO-Paradigma für die Softwareentwickler so wichtig?

Die **SA-Methode** hat zweifellos zu einer Systematisierung der Softwareentwicklung und zu besserer Dokumentation ihres funktionalen Aufbaus geführt, aber in vielen Anwendersystemen war gerade der funktionale Aufbau ständigen Änderungen unterworfen, so daß ein großer Wartungsaufwand bei den meisten so entwickelten Systemen entstand.

Der Ansatz von Peter Chen, die **Entity-Relationship-Methode** stellte deshalb den Begriff des Entities (deutsch:"Objekt", genauer "Datenobjekt") und seine Beschreibung als Relationstyp einer relationalen Datenbank, in den Mittelpunkt der Softwareentwicklung. Aber seine "Objekte" waren statische Datengebilde. Mit seiner Methode, der Entity-Relationship-Diagramme konnten die funktionalen Aspekte der Softwaresysteme nicht erfaßt werden.

Die **OO-Methode** unterstützt **Datenmodellierung und funktionale Gliederung** des so entwickelten Systems in gleicher Weise.
Mittels der OO-Methode wird das Softwaresystem als dynamisches Netzwerk kooperierender Objekte konstruiert. Die Entwickler können mit ihr die Komplexität des Problembereichs reduzieren und damit besser bewältigen. Dadurch können Entwickler mit der OO-Methode flexiblere und besser wartbare Software herstellen.

OO- und SA-Methode im Vergleich

Die Objektorientierte Softwareentwicklung

Die Kernaufgaben des **Objektorientierten Systementwurfs:**

- Auffinden der geeigneten Objektklassen
- Festlegen der Attribute und Methoden dieser Klassen
- Aufdecken der speziellen Beziehungen untereinander:
    - Vererbung (Beziehung: "ist-ein(e)")
    - Aggregation (Beziehung: "ist-Teil-von")
- Aufdecken von anderen Beziehungen:
    - Assoziation

Ein **Objektorientierter Systementwurf** besteht aus einem System von Objekten, deren Aufbau und Verhalten durch ihre Klassenbeschreibung festgelegt werden, und der Beschreibung der Kommunikationsstruktur zwischen den Objekten. Jede Klasse besitzt bestimmte Verantwortlichkeiten, diese ergeben sich aus dem Wissen, das die Objekte dieser Klasse zu verwalten haben. Das Wissen einer Klasse wird durch die **Attribute** (d.h. die verwalteten Daten), die Verantwortlichkeiten werden durch die **Verarbeitungsmethoden** für diese Daten ausgedrückt.

Die Methode der achtziger Jahre:

"Strukturierte" Analyse und Design

Ein strukturierter Systementwurf besteht aus Datenflußdarstellungen. Das System wird spezifiziert durch Daten, die zwischen bestimmten Funktionen herumkreisen, oder auf Speichern abgelagert werden.

Die strukturierte Methode ist auch heute noch sehr populär, weil es eine Vielzahl von "tools", also Softwareentwicklungswerkzeugen gibt, die mit dieser Methode arbeiten. Außerdem entspricht die strukturierte Methode, zusammen mit der Verwendung von Entity-Relationship-Diagrammen, der Vorgehensweise bei datenintensiven Verarbeitungen, Daten in Datenbanken zu führen und ihre Bearbeitung in davon getrennt entwickelten Prozeduren vorzunehmen.

Die objektorientierte Methode, die Daten und die sie bearbeitenden Funktionen in Objekten zusammenfaßt, entspricht einerseits besser der realen Welt und verringert andererseits, durch die Zusammenfassung von Daten und Bearbeitungsmethoden für diese Daten, die Komplexität der Softwaresysteme. Insbesondere ist die Verwendung und die Überprüfung eines Objekts, wegen der stark vereinfachten Schnittstelle viel einfacher als die Verwendung und Überprüfung einer Komponente aus einem "strukturierten" Entwurf.

Hinweis:
Bei sehr datenintensiven Verarbeitungen müßten allerdings zur konsequenten Umsetzung der OO-Entwürfe dann auch OO-Datenbanksysteme zur Verfügung stehen, die neben den Daten auch die Methoden abspeichern, was heute sehr oft nicht der Fall ist, da Daten immer noch überwiegend in Relationalen Datenbanken abgespeichert werden.

Objektorientierter Ansatz

Die Objektorientierte Softwareentwicklung unterteilt sich in vier Abschnitte (Rumbaugh u.a.: OO Modellieren und Entwerfen (Hanser, 1993, S.5)) , die verschiedene Entwicklungsschritte beinhalten:

a) Analyse: Das Analysemodell ist eine kompakte, präzise Abstraktion dessen, **was** das System leisten muß, ohne Aussage darüber, **wie** das zu geschehen hat.

Das Analysemodell enthält:
Das **Anforderungsmodell** (Use-Case-Diagramme (UC), Sequenzdiagramme (SQ)) Das **Objektmodell** (OM-Diagramm (OM)) **Dynamisches Modell** (Statetransition-Diagramme (ST), Event-Flow-Diagramme (EF), weitere SQ-Diagramme) Funktionales Modell:(SA-Diagramme) (Rumbaugh u.a. empfehlen dies, jedoch sind SA-Diagramme nicht in UML enthalten)

b) Systementwurf:
Der Systementwurf legt die Architektur des Systems fest.

c) Objektentwurf:
Festlegung der Implementierungsdetails der Objektklassen.

d) Implementierung:
Programmierung des Systems in einer Programmiersprache.

In der **Analysephase** müssen die handelnden Objekte aufgefunden, ihre Beziehungen untereinander festgelegt, ihr Zusammenspiel aufgedeckt und ihr Aufbau geklärt werden.

Es empfehlen sich eine Reihe von Schritten zum Auffinden der geeigneten Objekte:

Ausgegangen wird mindestens von einer umgangssprachlich formulierten Anforderungsdefinition, die eine möglichst präzise Problembeschreibung beinhaltet. Diese Anforderungsdefinition ist von Spezialisten des Problembereichs angefertigt worden.

1. Problembeschreibung genau lesen und verstehen.
   Ausdenken von verschiedenen "Szenarien" ("use-cases") anhand der Problembeschreibung.
2. Extrahiere Hauptworte aus der Problembeschreibung und anhand der verschiedenen Szenarien und benenne Klassenkandidaten:
   - Physische Objekte
   - Konzeptuelle Begriffe
   - Bilde Kategorien von Objekten.
   Stelle für jeden ausgesuchten Objektkandidaten die Frage: Ist dieses Objekt der realen Welt wichtig für die Anforderung an das zu schaffende System und besitzt es eine klar definierte Abgrenzung zu anderen Objekten? Diskutiere möglichst mit Experten aus dem Anwendungsbereich des Systems, ob die ausgesuchten Objektkandidaten in diesem Bereich von Bedeutung sind.
3. Finde gemeinsame Oberklassen durch Gruppierung von Klassen nach gemeinsamen Attributen und Methoden.

Auffinden und Zuordnen von Attributen an Objekte:

Die Adjektive in der Anforderungsdefinition sind potentielle Kandidaten für Attribute.

Die grundsätzliche Frage nach Attributen:
Für welche Daten ist dieses Objekt verantwortlich, d.h. welche Daten besitzt es und welche muß es wissen?

Unterfragen:

1. Wie wird das Objekt normalerweise beschrieben?
2. Welche Teile der allgemeinen Beschreibung sind für die Problemlösung notwendig?
3. Welche Teile der Objektbeschreibung braucht man mindestens für die angestrebte Problemlösung?

Jedes Attribut sollte ein "atomares" Konzept beschreiben also einen Einzelwert oder eine eng zusammengehörige Gruppe von Werten umfassen.

Falsche Attributkandidaten für ein Objekt:

Objekte:
Wenn die Eigenschaft unabhängig vom zugehörigen Objekt bearbeitet werden soll, ist diese kein Attribut, sondern selbst ein Objekt, zu dem eine Beziehung existiert.
Beziehungsattribute:
Wenn der Wert eines Attributes von ganz speziellen Umständen abhängt, dann ist dieses meistens besser als Attribut einer Beziehung zu modellieren. **Überflüssige Details:**
Eigenschaften des Objekts, die in keiner Weise die Methoden des Objekts beeinflussen sind überflüssig und sollten nicht als Attribute im Modell auftauchen **Auffinden und Zuordnen von Methoden:**

Definition: **Verantwortlichkeit**
Die Verantwortlichkeiten eines Objekts beinhalten: - Das Wissen, das ein Objekt besitzt und - Die Aktionen, die ein Objekt ausführen kann. Grundsätzliche Vorgehensweise zum Auffinden der Methoden: **Auswählen der Verben in der Anforderungsdefinition als Kandidaten für Methoden.**

Weitere Schritte:

1. Finde die Verantwortlichkeiten aufgrund der Systemaufgaben und weise jeweils die Verantwortlichkeit der zuständigen (im Zweifelsfall der am meisten zuständigen) Klasse zu. Bei Vererbungshierarchien lokalisiere die Verantwortlichkeiten so hoch wie möglich.
2. Betrachte die Benutzerszenarios in den SQ-Diagrammen
3. Auf welche Ereignisse muß das System reagieren?
4. Lege die Ein- und Ausgabeparameter der Methoden fest.
5. Die Methode wird so dicht wie möglich bei den zu beachtenden Informationen plaziert.

Aufdecken bzw. Aufbau von Vererbungsstrukturen:

1. Ist eine Klasse Spezialfall einer anderen?
   - Subtyping: Die Unterklasse fügt neue Attribute und Methoden dazu.
   - Specialization: Die Unterklasse redefiniert Attribute und Methoden.
2. Haben unterschiedliche Klassen gleiche Verantwortlichkeiten, dann bilde eine (abstrakte d. i. eine Klasse, von der keine Objekte angelegt werden. Sie ist lediglich eine Entwurfsidee.) Klasse dieser gemeinsamen Verantwortlichkeiten, von der die Klassen diese Verantwortlichkeiten erben.
3. Kontrolle für eine gute Vererbungsstruktur:
   - Die Unterklasse benötigt alle ererbten Attribute und Methoden.
   - Das Modell entspricht der natürlichen Struktur der realen Welt

Auffinden und Gestalten von Assoziationen:

1. Suche Verben in der Problembeschreibung und prüfe, ob die dadurch ausgedrückten Beziehungen problemrelevant sind.
2. Bei Assoziationen sind die beteiligten Klassen gleichrangig.
3. Meist bestehen Assoziationen unter Klassen aufgrund von Kommunikationswegen.
4. Schnappschuß oder Historie.

Für die einzelnen Schritte der Analyse gibt J. Rumbaugh (Rumbaugh, S. 317 ff) folgende Kurz-Anleitung:

1. Beschaffe (oder erstelle) eine Problembeschreibung.
2. Entwickle ein Objektmodell:
   - Identifiziere Objektklassen
   - Beginne mit dem Aufbau eines Data Dictionary, mit der Beschreibung der Klassen, Attribute, Assoziationen und Aggregationen
   - Füge die Beziehungen zu dem Objektmodell hinzu
   - Füge Attribute zu den Klassen und Beziehungen hinzu
   - Beachte Vererbung im Objektmodell

    - Teste Zugriffspfade mit Szenarien und wiederhole gegebenenfalls obige Schritte
    - Gruppiere geeignete Klassen zu Moduln
3. Entwickle ein dynamisches Modell:
    - Bereite Szenarien mit typischen Interaktionssequenzen vor
    - Identifiziere Ereignisse zwischen den Objekten und zeichne ein SQ-Diagramm (SQ) für jedes Szenario
    - Zeichne ein Ereignisfluß-Diagramm (EF) für das System
    - Zeichne ein Zustands-Diagramm (ST) für jede Klasse, die ein wichtiges dynamisches Verhalten besitzt
    - Prüfe Konsistenz und Vollständigkeit der Ereignisse, die sich auf mehrere Zustandsdiagramme beziehen

Die Unified Modeling Language - UML

Enstehung der UML

Zum besseren Verständnis der Unified Modeling Language sei hier eine kurze Zusammenfassung ihrer Entstehung gegeben.

Im Zeitraum Ende der achtziger bis Anfang der neunziger Jahre entstanden mehrere Notationen zur Modellierung von OO-Systemen. Die Grundlage der Notationen waren objektorientierte Programmiersprachen, wie Smalltalk und C++. Einige der wichtigsten Notationen waren:

- Booch (Grady Booch)
- OMT (James Rumbaugh)
- OOSE (Ivar Jacobson)
- Shlaer/Mellor
- Coad/Yourdon

Seit 1994/1995 arbeiteten die "drei Amigos" Grady Booch, James Rumbaugh und Ivar Jaboson in ihrer gemeinsamen Firma Rational Rose an einer Methode, die die Vorteile der einzelnen Notationen vereinen und die ihre Nachteile eliminieren sollte. Es entstand, in Zusammenarbeit mit vielen Firmen aus der Software-Industrie, die **Unified Method 0.8,** die bis 1997 zur **Unified Modeling Language 1.1** weiterentwickelt wurde. Die Notation der UML 1.1 war so umfangreich, daß der Anspruch eine Methode (konkrete Handlungsanweisung zum Erreichen eines Ziels) zu definieren auf die Definition einer Modellierungssprache reduziert wurde.

Die UML 1.1 wurde 1997 bei der Object Management Group (OMG) eingereicht und wurde ein weltweiter Standard im Bereich der Objekt-Orientierten Entwicklung. Zur Zeit (WS 02/03) ist die UML 1.4 aktuell. Die Version 2.0 wird entwickelt, ist aber noch nicht freigegeben.

Was ist die UML?

Die UML ist eine Modellierungssprache, also eine Sprache zur Beschreibung von Software-Systemen. Sie bietet eine einheitliche Notation, die für viele Anwendungsgebiete nutzbar ist.

Sie enthält Diagramme und Prosa-Beschreibungsformen. Mit Hilfe der UML können statische, dynamische und Implementierungsaspekte von Softwaresystemen beschrieben werden. "Die UML ist damit zur Zeit die umfassendste Sprache und Notation zur Spezifikation, Konstruktion Visualisierung und Dokumentation von Modellen für die Softwareentwicklung." (Gabriele Bannert - Objektorientierter Softwareentwurf mit UML). Eine Einführung in die Elemente, die im SWT-Praktikum benötigt werden finden Sie weiter unten in diesem Dokument.

UML und Vorgehensmodelle

Vorgehensmodelle beschreiben wie eine Notation (z. B. die UML) angewendet werden muß, um ein bestimmtes Ziel (Software) zu erreichen. Beispiele für Vorgehensmodelle sind das Wasserfallmodell, das evolutionäre Modell, das V-Modell. Vorgehensmodelle schreiben also die Art und Weise der Anwendung einer Notation vor. Vorgehensmodelle sind von vielen Faktoren abhängig, wie z. B. Projektgröße, vorhandene Infrastruktur, Know How der Entwickler, Branche, für die entwickelt wird.

Da die UML mit dem Anspruch entwickelt wurde eine universell gültige Notation zur Modellierung von Software-Systemen zur Verfügung zu stellen, beinhaltet sie kein Vorgehensmodell. Sie definiert keine Regeln zur Anwendung der verschiedenen Beschreibungselemente. Sie bildet mit ihren Beschreibungselementen die Basis verschiedener Vorgehensmodelle.

Entwickler müssen sich aus der umfangreichen Notation der UML, die für ihre speziellen Zwecke nötigen Beschreibungs- und Modellierungsformen (Diagramme, Beschreibung und Zusammenhänge) aussuchen und ihre eigene Methode mit Hilfe der UML-Notation definieren. Die UML stellt ein flexibles Rahmenwerk zur Verfügung, das an die Anforderungen der Benutzer (Firmen) angepaßt werden muß.

Das im Praktikum eingesetzte CASE-Tool **Innovator** enthält das V-Modell.

Vorgehensmodelle in der Praxis oder Wie verläuft ein Projekt in der Realität?

Zur praxisnahen Erläuterung eines UML-Vorgehensmodells sei hier ein Beispiel aus der Literatur zitiert. Es stammt aus der Zeitschrift OBJEKTspectrum 5/98. Die Autoren sind Johannes Bellert, Jörg Noack und Thomas Zang. Sie beschreiben wie sie in einem realen Software-Projekt die Diagrammtypen der UML - aufeinander aufbauend - eingesetzt haben, um das Projekt, vom Projektstart bis zum fertigen Produkt, mit Hilfe der UML zu modellieren und zu konstruieren.

Projekt starten

- Projektrahmen abstecken
- Werkzeuge definieren
- Qualitätsplan erstellen

Wiederhole Produktzyklus

- **Anforderungen aufnehmen**
- **Use Cases benennen**

Akteure identifiziern

Kommunikationsbeziehungen eintragen

Qualität sichern

Szenarien definieren

Objekte identifiziern und anlegen (CRC-Karten)

Nachrichten eintragen

Methoden definieren

Objekte klassifizieren

Qualität sichern

Logische Architektur erstellen

Klassenstrukturmodell erstellen

Attribute definieren

(weitere) Methoden definieren

Vererbungsstruktur festlegen

"Global Services" und Verwalterobjekte definieren

Testfälle definieren

Qualität sichern

Komponentenarchitektur und Komponentenmodell festlegen

Systemkonzept und Architektur definieren (Architekturmuster)

Komponenten bilden

Persistenzmerkmale definieren

Persistenzschicht einziehen

Wiederhole Realisierungzyklus

(weitere) benötigte Dienste identifizieren
Klassen/Parts anlegen

(weitere) Attribute definieren
(weitere) Methoden definieren und realisieren
Benutzeroberflzeptäche und Anwendungsfenster entwerfen
Datenzugriffsschicht entwickeln (Cobol)
Host-DB2-Tabellen anfordern
Qualität sichern und Entwicklertest durchführen
so lange, bis Ausbaustufe realisiert

Ausbaustufe fertigstellen (Laufzeit-Image, Softwareverteilung usw.

Ausbaustufe prTD>

Projekt verwalten

so lange, bis das Produkt fertig
Projekt abschließen

UML und Geschäftsprozeßmodellierung

Scheer, ganzheitlicher Ansatz, Software wird eingebettet in betriebliche Abläufe.

UML und Modellierung von Embedded Systems

reaktive technische Systeme, z. B. MI, PR, Dominanz der dynamischen Diagrammtypen

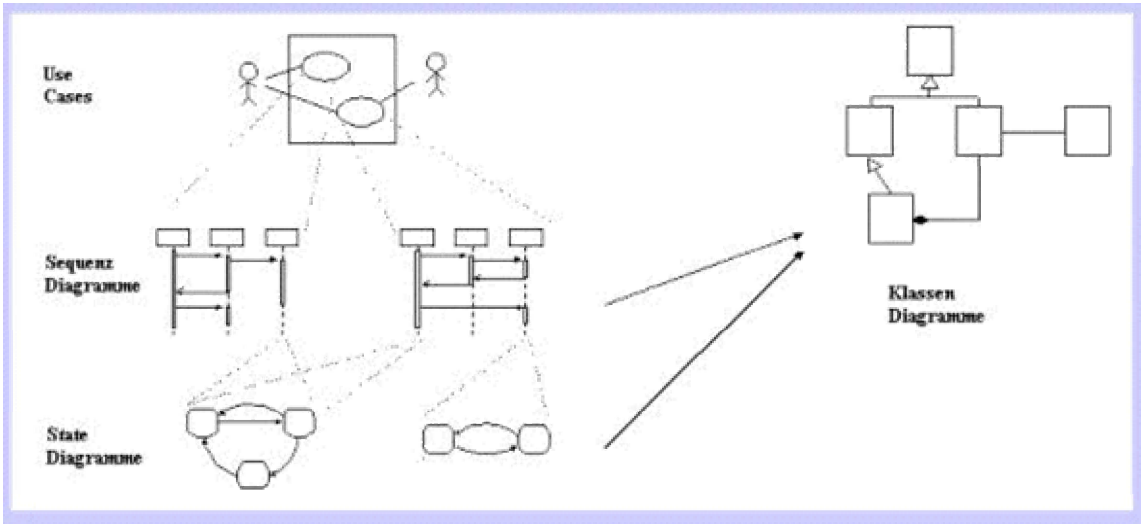
UML und Client/Server bzw. Komponentenbasierten Umgebungen

Zwei-Schicht, Drei-Schicht-Architekturen, große Systeme, UML-Einsatz unverzichtbar

UML und Modellierung von Software

ist Kern der SWT in FHD-FBI

Übersicht der UML-Diagrammtypen



Diese Grafik gibt eine Übersicht der UML-Diagrammtypen, die im SWT-Praktikum eingesetzt werden. Die Zusammenhänge seien stichwortartig erläutert:
In Use-Case-Diagrammen werden die Anwendungsfälle (Verwendungszwecke, Use Cases) der geplanten Software beschrieben. Diese stehen als Oval in dem Rechteck, das das geplante System gegen den Rest der Welt abgrenzt. Die Strichmännchen, die sogenannten Akteure, sind systemfremde Elemente, die mit dem System kommunizieren.
Für jeden Use Case können ein oder mehrere Sequenzdiagramme erstellt werden. Sequenzdiagramme beschreiben die Interaktion der an einem Use Case beteiligten Klassen. Die senkrechten Balken stellen die Objekte dar; die Pfeile zwischen den Objekten stellen die Nachrichten dar über die die Objekte miteinander kommunizieren.
Collaboration Diagramme sind eine alternative Darstellung der Sequenzdiagramme. Die Rechtecke repräsentieren Objekte, die Linien Kommunikationsbeziehungen und die Pfeile bezeichnen die Nachrichten.
Die Klassen eines Systems werden in Klassendiagrammen zusammengefaßt. Die Klassen werden als Rechteck dargestellt, die Linien zwischen den Klassen bezeichnen Beziehungen zwischen den Klassen.
Das interne Verhalten der Klassen (ihre Dynamik) wird in State-Transition-Diagrammen dargestellt. Ein State-Transition-Diagramm bezieht sich auf eine Klasse. Die abgerundeten

Rechtecke stellen Zustände der Klasse dar; die Pfeile beschreiben die Zustandsübergänge.

Die UML enthält nachfolgende Diagrammtypen. Die Diagramme, die im SWT-Praktikum eingesetzt werden sind in diesem Dokument näher beschrieben.

- Strukturdiagramme (statische Aspekte)
  - Klassendiagramm (class diagram)
  - Objektdiagramm (object diagram)
  - Komponentendiagramm (component diagram)
  - Verteilungsdiagramm (deployment diagram)
  - Paket Diagramm (package diagram)
- Verhaltensdiagramme (dynamische Aspekte)
  - Use-Case-Diagramm, Anwendungsfalldiagramm (use case diagram)
  - Sequenzdiagramm (sequence diagram)
  - Aktivitätsdiagramm
  - Kollaborationsdiagramm (collaboration diagram)
  - Zustandsdiagramm (statechart diagram)

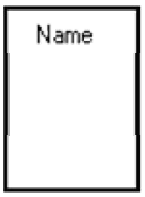
Ein Glossar der Begriffe der UML findet sich auf dem Server von Bernd Oestereich http://www.oose.de/glossar . Unter http://www.system-bauhaus.de/uml/ haben mehrere Autoren eine englisch-deutsche Übersetzung der UML-Begriffe veröffentlicht, die zur Vermeidung von Mißverständnissen bei der Übertragung der englischen Begriffe in die deutsche Sprache hilfreich ist.

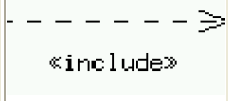
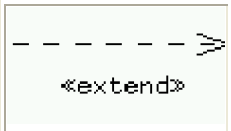
Use Case Diagramme

Zweck

In einem Use Case Diagramm wird das geplante System, die Verwendung des geplanten Systems und seine Interaktion mit externen Objekten beschrieben.

Diagrammelemente

	**System** Das Rechteck stellt das geplante System dar. Der Name gibt den Namen des Systems an. Ein Use Case Diagramm kann auch mehrere Systeme enthalten. Dadurch kann ein System in Teilsysteme gegliedert werden.
	**Use Case** Ein Oval symbolisiert einen Anwendungsfall des Systems. Ein Anwendungsfall ist ein bestimmter Zweck für den das System eingesetzt werden soll. Er beschreibt aus Sicht des Anwenders eines System (Fachabteilung für die das System realisiert wird) welche Leistungen das System für den Anwender zur Verfügung stellt. Ein Use Case stellt somit einen Teil der Funktionalität des Systems dar.

	**Akteur** Das Strichmännchen steht für ein externes, nicht zum System gehörendes Objekt. Es kann einen Benutzer des Systems symbolisieren, eine Person, die am Terminal sitzt und das System über die Benutzeroberfläche bedient. Das Strichmännchen kann aber auch ein fremdes (Software- oder Hardware-) System sein mit dem das geplante System kommuniziert. Aus der Sicht des geplanten Systems ist die Unterscheidung dieser zwei Rollen, die das externe Objekt im realen Leben spielen kann, unerheblich. Das geplante System muß auf die Ereignisse, die ein Akteur verursacht reagieren.
	**Assoziation** Eine Linie stellt eine Assoziation dar. Eine Assoziation zwischen einem **Akteur** und einem **Use Case** beschreibt die Kommunikation des Akteurs mit der Funktionalität, die das System in diesem Use Case zur Verfügung stellt. Assoziationen sind auch **zwischen zwei Use Cases** möglich.
	**include-Assoziation** Bei der **include** Beziehung (früher uses) verwendet ein Use Case die Funktionalität, die der andere Use Case zur Verfügung stellt. Dies zielt auf die Wiederverwendung der Funktionalität, die sich hinter einem Use Case verbirgt.
	**extend-Assoziation** Die **extend** Beziehung beschreibt die Erweiterung der Funktionalität eines Use Cases durch einen anderen Use Case. Man kann dadurch optionales Verhalten beschreiben, bzw. Funktionen modellieren, die nur unter bestimmten Bedingungen ausgeführt werden.

Anwendungsbereich

Use Case Diagramme werden zur Festlegung der Anforderungen an ein Softwaresystem und bei der Modellierung der Geschäftsprozesse eingesetzt. Sie werden in den frühen Phasen eines Projektes (Analyse) eingesetzt. Die Funktionalität wird aus Sicht der zukünftigen Benutzer des geplanten Systems analysiert oder definiert. Implementierungsdetails, wie z. B. die zu verwendende Programmiersprache, Systemarchitektur (Client-Server, ...) usw. sind in diesen Phasen noch nicht wichtig und deshalb zu vernachlässigen.

In der Systemanalyse wird, z. B. in Meetings zwischen Systemanalytikern und Anwendern des künftigen Systems definiert was das System aus der Sicht der Anwender leisten soll. Die einzelnen Fälle, die das System abdecken soll, werden als Use Case in die Diagramme eingezeichnet und beschrieben.
Die Systemgrenze wird festgelegt. Es wird definiert welche Leistungen das System bringen soll und welche Leistungen nicht im zukünftigen System enthalten sein sollen. Die Akteure sind systemfremde Elemente, die mit dem System über die Beziehungen zu den Use Cases mit dem System kommunizieren.
Ein Use Case Diagramm stellt also eine grobe Skizze des Systems dar das den Zweck des geplanten Systems angibt, seine Grenzen und Schnittstellen und die Fremdsysteme oder Benutzer, die auf es einwirken.

Zusammenhang

Für jeden Use Case können ein oder mehrere Sequenzdiagramme erstellt werden, die das Verhalten des Systems in diesem Fall modellieren.

Hinweise zu Use Case Diagrammen

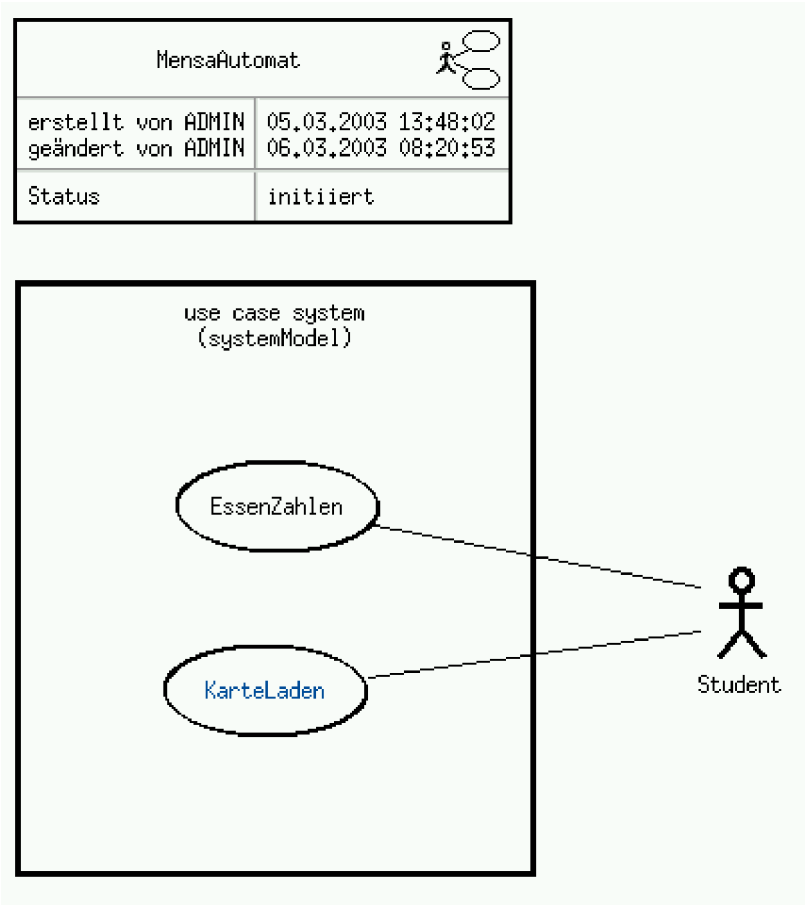
Identifizieren Sie das System und den Zweck, den es erfüllen soll. Beantworten Sie die Frage, "Was soll das System tun?" und "Was soll das geplante System **nicht** tun?". Dokumentieren Sie Ihre Antworten auf die beiden Fragen in der Systemspezifikation (Beschreibung).

Identifizieren Sie die Personen, die das System nutzen sollen und Fremdsysteme mit denen das System kommunizieren soll. Beschreiben Sie auch diese Akteure, damit Sie im nächsten Praktikum (in zwei Wochen) bzw. in der nächsten Projektbesprechung (wann wird die sein?) noch wissen warum Sie bestimmte Elemente eingeführt haben.

Identifizieren Sie die Use Cases, indem Sie überlegen wofür das System eingesetzt werden soll. Beantworten Sie die Frage welche verschiedenen Handlungen das System für die Benutzer (Akteure) erbringen soll und dokumentieren Sie diese in den Use-Case-Spezifikationen.

Welche Akteure stoßen welche Use Cases an? Welche Use Cases liefern Ergebnisse an die Akteure? Die Antwort auf diese beiden Fragen hilft Ihnen die richtigen Assoziationen zu finden.

Beispiel



Sequenzdiagramm

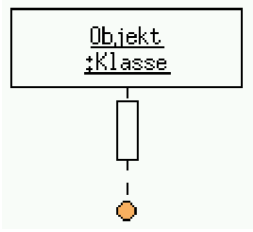
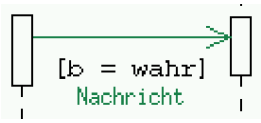
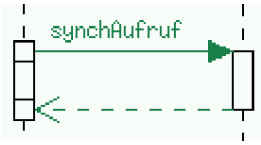
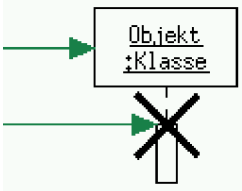
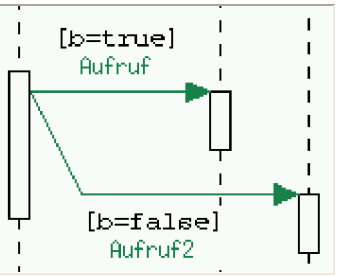
Synonym:

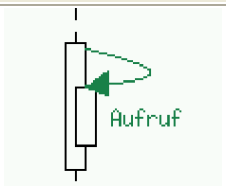
Interaktionsdiagramm, Szenariodiagramm

Zweck

Sequenzdiagramme beschreiben die Kommunikation zwischen Objekten in einer bestimmten Szene. Es wird beschrieben wie die an einer Szene beteiligten Objekte zusammenarbeiten müssen, damit das System die Leistung erbringt, die der Benutzer fordert. Sequenzdiagramme spezifizieren also die Interaktion der an einer Szene beteiligten Objekte.
Sequenzdiagramme enthalten eine implizite Zeitachse. Die Zeit schreitet von oben nach unten fort.

Diagrammelemente

	**Klasse, Objekt** Eine senkrechte, gestrichelte Linie stellt die Lebenslinie (lifeline) eines Objekts dar. Im Bereich der Lebenslinie existieren Objekte der Klasse. In dem Rechteck über der Lebenslinie steht der Klassenname und/oder der Objektname. Das schmale Rechteck auf der gestrichelten Linie stellt eine Aktivierung dar. Nur im Bereich der Aktivierung kann ein Objekt Nachrichten empfangen oder versenden. Eine Aktikvierung ist der Bereich, in dem eine Methode aktiv ist. Auf einer Lebenslinie können mehrere aktive Bereiche enthalten sein.
	**Nachricht** Objekte kommunizieren über Nachrichten. Nachrichten werden als Pfeile zwischen den Objekten eingezeichnet. Der Name der Nachricht steht an dem Pfeil. Die Angabe einer Bedingung ist optional. Bedingungen werden in eckigen Klammern dargestellt.
	**Synchroner Aufruf** Der Pfeil mit der ausgefüllten Spitze bezeichnet einen synchronen Methodenaufruf. Der Aufruf erfolgt von der Quelle zum Ziel, d. h. die Zielklasse muß eine entsprechende Methode implementieren. Die Quelle wartet mit der Verarbeitung bis die Zielklasse ihre Verarbeitung beendet hat und setzt die Verarbeitung dann fort. Deshalb werden asynchrone Aufrufe mit einem Return abgeschlossen. Die Bezeichnung des Return mit einem Namen ist optional.
	**Asynchroner Aufruf** Mit einer halben Pfeilspitze werden asynchrone Aufrufe gekennzeichnet. Der Aufruf erfolgt von der Quelle zum Ziel. Die Quelle wartet mit der Verarbeitung **nicht** auf die Zielklasse. Asynchrone Aufrufe werden verwendet, um parallele Verarbeitung zu modellieren.
	**Konstruktor, Destruktor** Endet eine Nachricht oder ein Aufruf im Namensfeld eines Objekts, so wird dadurch das Erzeugen einer Instanz der Klasse dargestellt. Das Vernichten einer Instanz wird durch ein Kreuz auf der Lebenslinie dargestellt.
	**Verzweigung** In Sequenzdiagrammen kann durch Angabe von Bedingungen zu verschiedenen Zielen verzweigt werden. Ist die Bedingung b=true erfüllt, wird Aufruf gesendet. Ist die Bedingung b=false, wird Aufruf2 gesendet.

	**Rekursion** wird wie im nebenstehenden Bild dargestellt. Bei rekursiven Nachrichten ruft ein Objekt eine eigene Methode auf.

Anwendungsbereich

Sequenzdiagramme werden zur Modellierung der Dynamik des Systems eingesetzt. In Sequenzdiagrammen werden einzlene Szenarien des Systems modelliert. Ein Szenario ist eine Szene, die bei der Anwendung des Systems vorkommen kann. In Sequenzdiagrammen wird festgelegt welche Objekte an einer Szene beteiligt sind, welche Nachrichten die Objekte sich zusenden und in welcher Reihenfolge die Nachrichten gesendet werden. Die Nachrichten werden in der zeitlichen Reihenfolge in der sie auftreten müssen von oben nach unten in einem Diagramm eingezeichnet.
Jedes Sequenzdiagramm beschreibt das Verhalten des Systems in einer Szene und stellt somit einen Ausschnitt des Gesamtsystems dar.
Ein System wird in der Regel nicht vollständig durch Sequenzdiagramme spezifiziert. Es werden nur die Szenen modelliert, die häufig vorkommen oder besonders wichtig sind.

Zusammenhang

Jedes Sequenzdiagramm kann sich auf einen Use Case beziehen. Da Use Case Diagramme aber nicht in jedem Projekt verwendet werden, können Sequenzdiagramme auch unabhängig von Use Case Diagrammen existieren.
Die Klassen der Sequenzdiagramme müssen in den Klassendiagrammen enthalten sein. Ereignisse und Nachrichten erfordern entsprechende Methoden bei den Zielklassen.

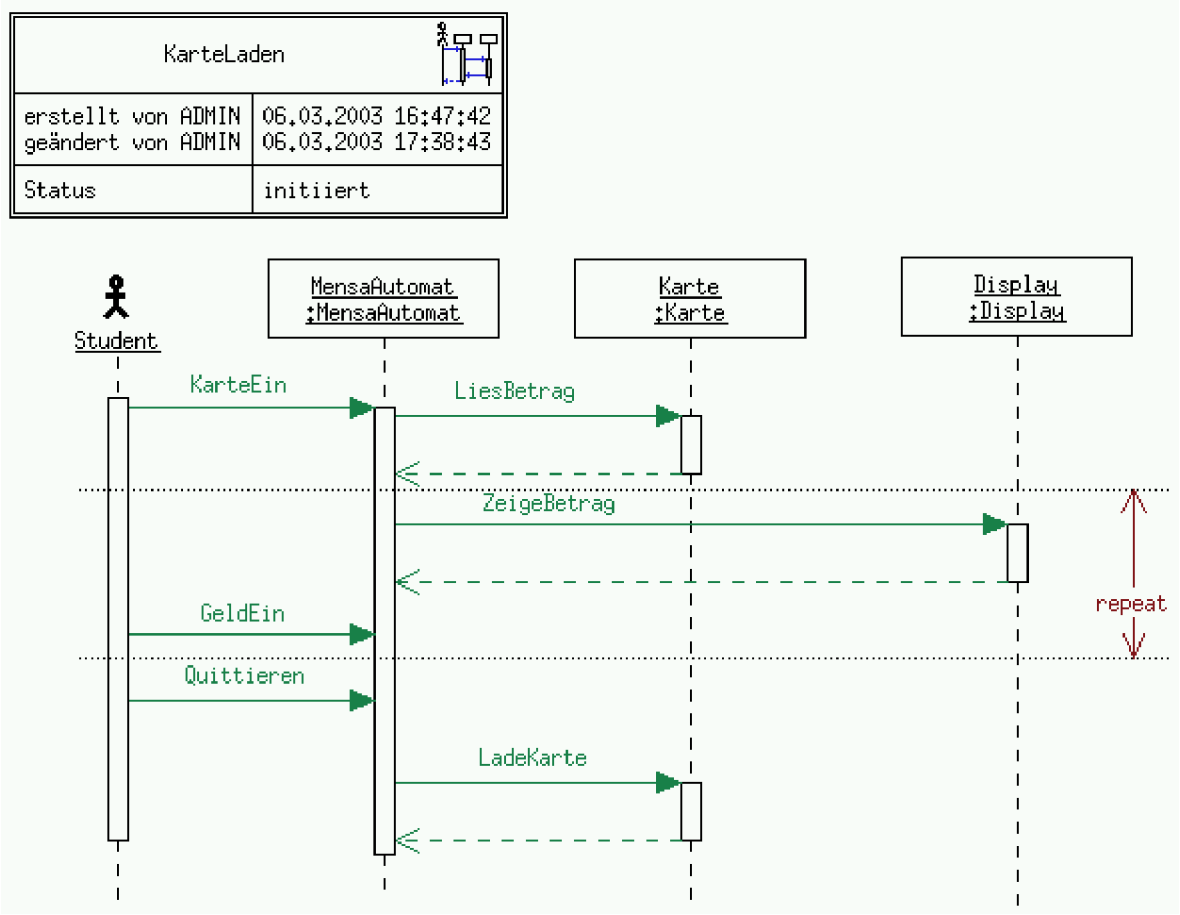
Hinweise

In Sequenzdiagrammen wird die Dynamik, der an einem Use Case beteiligten Klassen modelliert. Bevor Sie Sequenzdiagramme modellieren können, müssen Sie eine Vorstellung der Klassen des Systems haben. Der Prozeß der Identifikation der Klassen des Systems muß also parallel zur Modellierung der Sequenzdiagramme oder zeitlich vorher erfolgen.

Konzentrieren Sie sich zuerst auf die wichtigen Fälle, modellieren Sie die Sonderfälle später.

Daß eine Beschreibung der Sequenzdiagramme vorteilhaft ist, dürfte jetzt schon selbstverständlich sein.

Beispiel



Da die Instanzen, der in diesem Diagramm modellierten Klassen, noch nicht festgelegt sind, wurden für die Klassennamen und Objektnamen identische Begriffe angegeben. Es wurde davon ausgegangen, daß alle Instanzen der beteiligten Klassen sich gleich verhalten. Die punktierten, waagrechten Linien, die mit dem senkrechten "repeat"-Pfeil verbunden sind, geben an, daß die eingeschlossenen Nachrichten mehrfach wiederholt werden.

Collaboration Diagramm

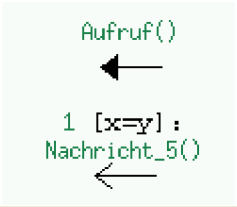
Synonym:

Kollaborationsdiagramme, Kooperationsdiagramme, Event Flow Diagramme

Zweck

Auch in Collaboration Diagrammen wird die Kommunikation zwischen Objekten in einzelnen Szenen modelliert. Sie können alternativ zu Sequenzdiagrammen verwendet werden. In Sequenzdiagrammen werden Szenen dargestellt an denen wenige Klassen beteiligt sind, die viele Nachrichten austauschen. Collaboration Diagramme eignen sich zur Darstellung von Szenen mit vielen Klassen, die aber wenige Nachrichten austauschen. Die Darstellung der zeitlichen Reihenfolge der Nachrichten kann durch Nummerierung vorgenommen werden.

Diagrammelemente

	**Objekt** Ein Rechteck beschreibt ein Objekt des Systems. Es kann der Objektname und der Klassenname angegeben werden.
	Beziehung zwischen Objekten Die Linie kennzeichnet eine Beziehung zwischen zwei Objekten.
	**Nachricht zwischen Objekten** Die Pfeile an den Kommunikationsbeziehungen geben die Nachrichten an, die über die Beziehung übertragen werden und ihre Richtung. Diese können vom selben Type sein, wie im SQ-Diagramm, synchroner Aufruf, asynchroner Aufruf oder unspezifische Nachricht.

Anwendung

Collaboration Diagramme werden zur Modellierung der Dynamik eines Systems in einzelnen Szenen vewendet. Es wird festgelegt welche Objekte an der Szene beteiligt sind, welche Beziehungen zwischen den Objekten existieren und welche Nachrichten übertragen werden.

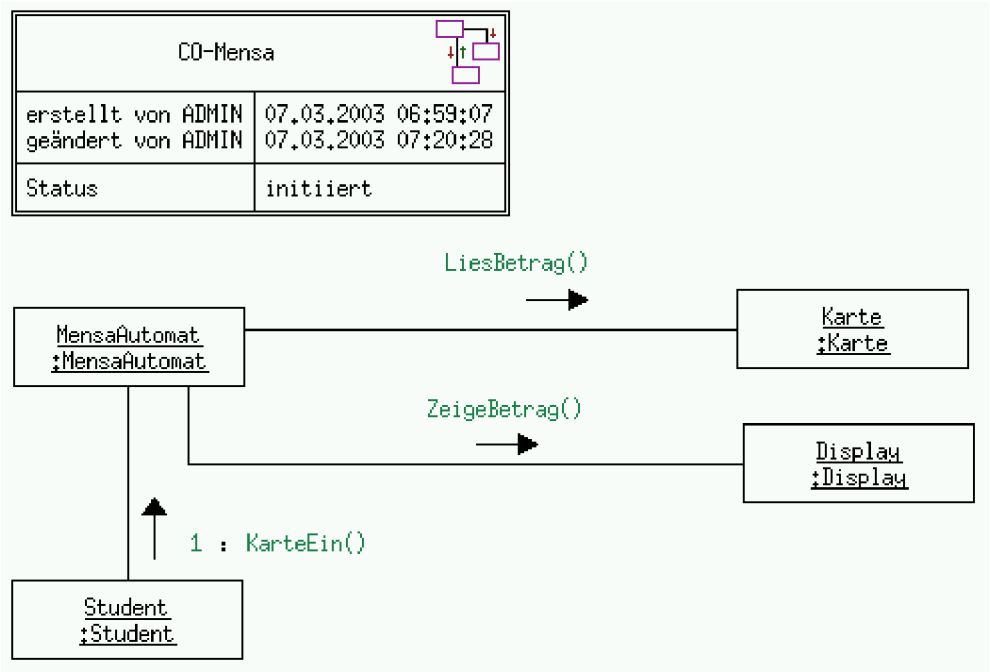
Zusammenhang

Da die Collaboration Diagramme alternativ zu Sequenz Diagrammen verwendet werden sind die selben Zusammenhänge zu berücksichtigen, die unter Sequenzdiagrammen beschrieben sind. Collaboration Diagramme werden manchmal zusätzlich zu Sequenzdiagrammen eingesetzt um unabhängig von der zeitlichen Reihenfolge der Nachrichten die Kommunikationsbeziehungen der an einer Szene beteiligten Objekte übersichtlich darzustellen.

Hinweise

Collaboration Diagramme werden dann als Alternative zu Sequenz Diagrammen verwendet, wenn an einer Szene viele Objekte beteiligt sind, die wenige Nachrichten austauschen. Sind wenige Objekte an einer Szene beteiligt, die viele Nachrichten austauschen, sind Sequenz Diagramme wegen der übersichtlicheren Darstellung vorzuziehen.

Beispiel



Klassendiagramm

Synonym:

(Objektdiagramm)

Zweck

Klassendiagramme werden verwendet um die Klassen und Objekte eines Bereiches, die Eigenschaften (Attribute) und das Verhalten (Operationen) der Klassen und die Beziehungen zwischen den Klassen zu modellieren.

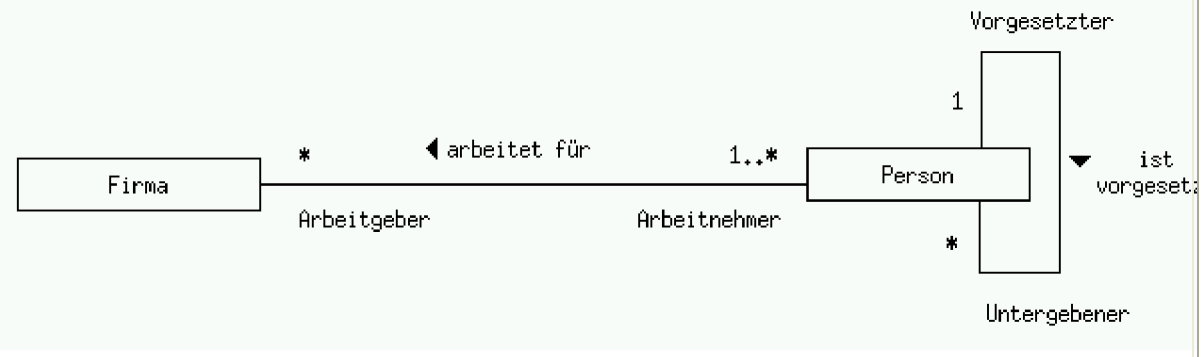
Klassendiagramme beschreiben die statische Sicht auf eine System. Sie sind der zentrale Diagrammtyp der UML.

Diagrammelemente

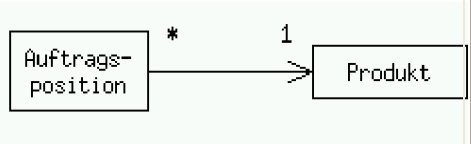
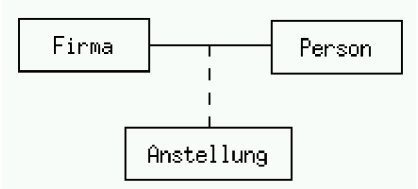
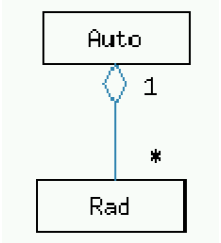
	**Klasse** Eine Klasse beschreibt die Struktur und das Verhalten ihrer Objekte. Sie wird als dreigeteiltes Rechteck dargestellt. Im oberen Bereich steht der Klassenname, in der Mitte die Attribute der Klasse und im unteren Drittel stehen die Operationen (Methoden) der Klasse. **UML-Syntax für Attribute** attributname [: attributtyp [= initialwert]] **UML-Syntax für Methoden** methodenname [(parametername [: parametertyp] [,

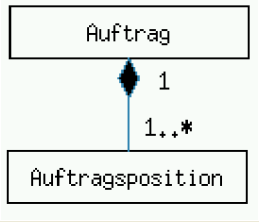
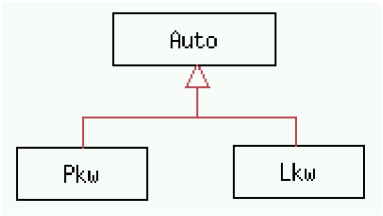
	...])] [: rückgabetyp] **Sichtbarkeit der Attribute und Methoden** Die Sichtbarkeit von Methoden und Attributen wird in C++ ähnlicher Weise gekennzeichnet. + öffentlich (public) # geschützt (protected) - privat (private)

Assoziation



Die Assoziation stellt eine Beziehung zwischen Klassen dar. Sie kann einen Beziehungsnamen ("arbeitet_für") tragen. Der Pfeil am Namen gibt an in welcher Richtung der Beziehungsname gelesen werden muß. Die Rollen der an einer Assoziation beteiligten Klassen können mit Hilfe von Rollennamen ("Arbeitgeber", "Arbeitnehmer") beschrieben werden. Die Multiplizität gibt an wieviel Objekte ("*, 1..*") an einer Beziehung beteiligt sein können. Ungerichtete Assoziationen sind in beiden Richtungen navigierbar.

	**Gerichtete Assoziation** Soll eine Assoziation nur in einer Richtung navigierbar sein (Objekt Auftragsposition kennt das Objekt Produkt, das es enthält, aber das Produkt weiß nicht welchen Auftragspositionen es zugeordnet ist) so wird die Assoziation als Pfeil gezeichnet. Die Pfeilspitze gibt die Navigationsrichtung an.
	**Assoziationsklasse** Sind die Objekte einer Klasse vom Vorhandensein einer Beziehung abhängig, so kann diese Klasse als Assoziationsklasse modelliert werden.
	**Aggregation** Eine Aggregation ist eine Teil-von- oder besteht-aus-Beziehung und sagt aus, daß ein Objekt Bestandteil eines anderen ist. Ein Auto besteht z. B. aus Fahrgestell, Motor, Rädern usw.. Die Einzelteile können aber unabhängig vom Ganzen als Objekte existieren.

	**Komposition** Auch eine Komposition ist eine Teil-von-Beziehung. Hier sind jedoch die Einzelteile vom Ganzen existenzabhängig. Eine Auftragsposition gehört immer zu einem Auftrag. Es gibt keine Auftragspositionen, wenn nicht vorher ein Auftrag erzeugt wurde.
	**Vererbung** wird auch Generalisierung oder Spezialisierung genannt und als Dreieck, das mit seiner Spitze auf die generelle, vererbende Klasse zeigt, dargestellt. (Ein Auto ist ein Pkw oder ein Lkw. Ein Pkw erbt alle Eigenschaften der Klasse Auto.)

Anwendungsbereich

Klassendiagramme sind der zentrale Diagrammtyp der UML und vieler anderer OO-Methoden. Klassendiagramme beschreiben die Klassen eines Systems, ihre Eigenschaften, Operationen und dieBeziehungen zwischen den Klassen. Bei der Modellierung von Klassendiagrammen muß zwischen dem Analyse- und dem Designmodell unterschieden werden.

Im Analysemodell werden alle in der Systemanalyse identifizierten, problemrelevanten Klassen und Beziehungen dargestellt. Das Analysemodell bildet das System aus der Sicht eines Anwenders ab. Es ist die Diskussionsbasis zwischen Analytiker und Anwender, die dafür verwendet wird um zu dokumentieren und zu definieren was der Anwender von dem zukünftigen System erwartet. Für den Anwender (den Kunden des Informatikers) beschreibt das Modell was die zukünftige Software aus fachlicher Sicht leisten soll. Für die Informatikabteilung, die das geplante Produkt entwickeln soll, stellt das Analysemodell die Basis der zukünftigen Entwicklungsarbeit dar. Da Fehler in der Analyse sehr teuer sind muß das Analysemodell sehr sorgfältig modelliert werden.

Im Designüberführt der Informatiker das Analysemodell in ein Designmodell. Er ergänzt die in der Systemanalyse erkannten und dokumentierten Zusammenhänge um die Informationen, die nötig sind um das fachliche Modell zu implementieren.

Im Design wird die Implementierungssprache festgelegt. Es wird definiert wie Assoziationen zu implementieren sind; die Art und Weise wie die Instanzen der Klassen verwaltet werden wird festgelegt. Die Persistenzschicht wird modelliert, d. h. es wird definiert wie die Klassen auf eine Datenbank abgebildet werden; die Datenbank wird ausgewählt. In der Regel werden bei der Implementierung eines Modells Klassenbibliotheken verwendet, die im Klassenmodell dokumentiert werden müssen. Die Benutzeroberfläche (Windows, Motif, Swing) muß eingeführt werden. Eventuell muß das Analysemodell in eine Client/Server-Architektur überführt werden.

In der Realität werden immer mehrere Sichten auf das Klassenmodell existieren; die Analysesicht und die Design- oder Implementierungssicht. Diese Sichten auf das Klassenmodell können durch den Einsatz moderner Entwicklungswerkzeuge, wie z. B. Case-Tools konsistent gehalten werden.

Zusammenhang

Das Verhalten einzelner Klassen kann in State Diagrammen modelliert werden. Wurden Use Case Diagramme eingesetzt können die Klassen des Klassenmodells aus den Use Case Spezifikationen abgeleitet werden.

Wurden Sequenz Diagramme eingesetzt, so sind alle dort modellierten Klassen in den Klassendiagrammen zu integrieren.

Hinweise

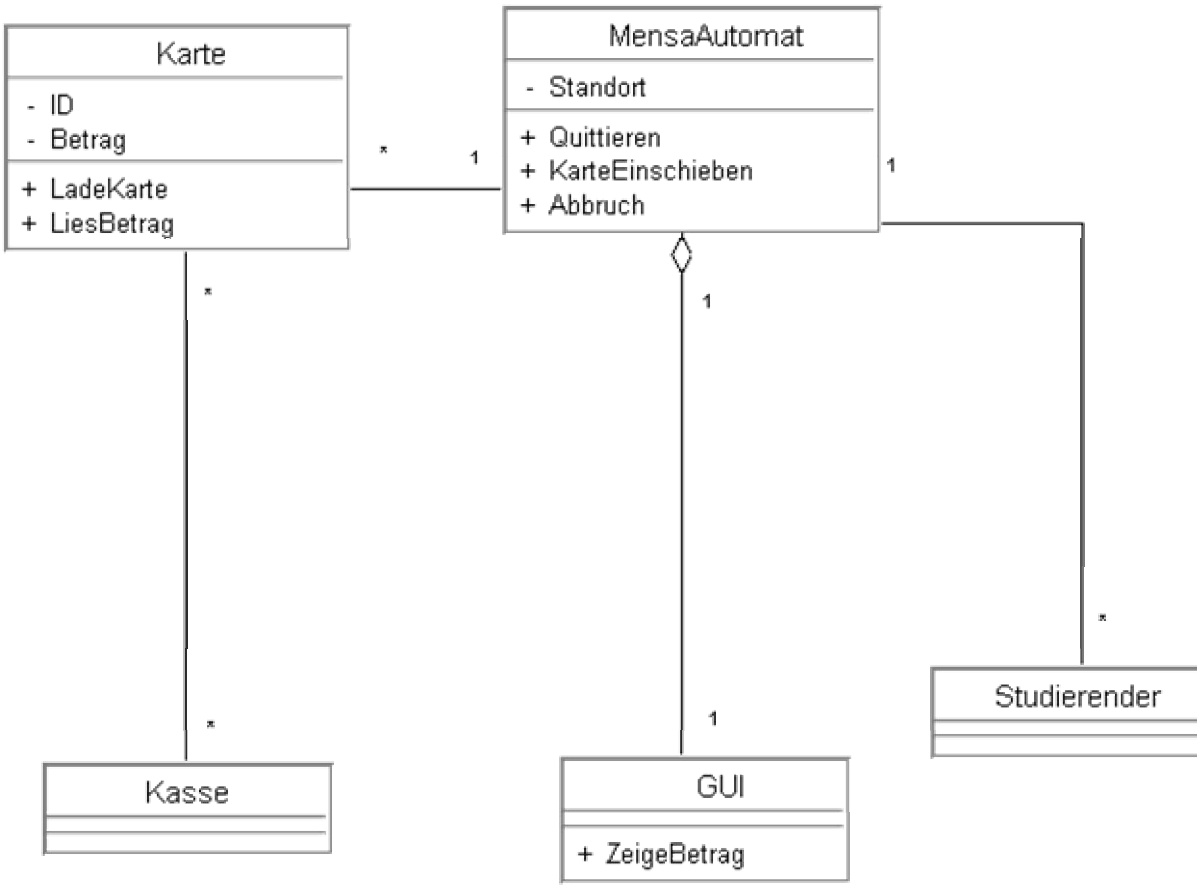
Die Klassen der Klassendiagramme werden auch in Sequenzdiagrammen bzw. Collaboration Diagrammen verwendet. Die beiden Diagrammtypen müssen bei Änderungen abgeglichen werden.

Ändern sich die Systemanforderungen, die in den Use Case Diagrammen oder in Sequenz Diagrammen modelliert wurden, so sind die Auswirkungen auf das Klassenmodell zu überprüfen. *Wie findet man Klassen?*

Diese Frage wird von der UML nicht beantwortet. Dafür gibt es verschiedene Techniken, wie z. B. **CRC-Karten-Technik** (Class, Responsibility, Collaboration). Jede **Klasse** stellt dem Gesamtsystem einen bestimmten "Service" zur Verfügung, der vom Gesamtsystem durch Aufruf der entsprechenden Methoden genutzt werden kann. Jede Klasse ist für das Wissen ihrer Objekte (Attribute) und deren Verhalten (Methoden) verantwortlich (**responsibility**). Zur Erfüllung ihrer Aufgaben müssen Klassen über Beziehungen mit anderen Klassen zusammenarbeiten (**collaboration**). Die Verantwortlichkeiten und die Beziehungen von Klassen werden bei Einsatz der CRC-Technik in der Analyse notiert um so die Klassen eines Problembereiches zu identifizieren.

Rumbaugh empfiehlt die Substantiv-Methode zur Identifikation von Klassen. Aus allen vorhandenen Beschreibungen eines geplanten Systems werden die Substantive als Klassenkandidatenextrahiert. Diese werden durch kritische Analyse so weit eliminiert, bis nur noch die zur Lösung eines Problems notwendigen Klassen übrigbleiben.

Beispiel



Zustandsdiagramm

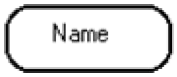
Synonym:

State Diagramm, State Transition Diagramm, Zustandsautomat, endlicher Automat, Zustandsübergangsdiagramm.

Zweck

Das dynamische Verhalten von Klassen kann in Zustandsdiagrammen beschrieben werden. In Zustandsdiagrammen wird der Lebenszyklus einer Klasse (bzw. ihrer Objekte) dargestellt. Jedes Objekt kann im Laufe seiner Existenz verschiedene Zustände annehmen. Nachrichten von anderen Objekten veranlassen es zum Zustandswechsel (Transition). In jedem Zustand kann ein Objekt nur auf bestimmte Ereignisse (Nachrichten) reagieren.

Diagrammelemente

	**Zustand (State)** Eine Zustand wird als Rechteck dargestellt, dessen Ecken abgerundet sind. Im Zustandssymbol steht der Name des Zustands.

	**Zustandsübergang (Transition)** Ein Zustandsübergang wird als Pfeil dargestellt. Der Pfeil liegt immer zwischen zwei Zustandssymbolen. Das nebenstehende Symbol zeigt die vollständige Beschriftung eines Zustansübergangspfeils. Es hat folgende Bedeutung: Der Zustansübergang wird von einem Ereignis (Nachricht an die Klasse) verursacht. Der Zustandübergang kann zusätzlich zum Eintreffen einer Nachricht von einer Bedingung abhängig sein. Das modellierte Objekt kann beim Zustandsübergang eine Aktion ausführen und es kann eine Nachricht (Ausgangsereignis) an eine (andere) Zielklasse senden. Die Beschriftungselemente sind optional und können beliebig miteinander kombiniert oder auch weggelassen werden.
	**Anfangszustand** Ein Anfangszustand symbolisiert den Beginn des Lebenszyklus eines Objektes. Er steht somit für das Erzeugen und Initialisieren eines Objektes.
	Endzustand Ein Endzustand symbolisiert das Lebensende eines Objektes und somit das Vernichten (Löschen) einer Instanz.
	**Zustand (vollständige Notation)** Das nebenstehende Bild stellt die vollständige Notation zur Beschreibung eines Zustandes dar. (Auch diese Elemente können verwendet werden, sie müssen nicht für jeden Zustand vollständig angegeben werden.) Sie haben folgende Bedeutung: Im mittleren Bereich können die Zustandsattribute angegeben werden. Ein Zustandsattribut ist ein Attribut, das in einem Zudtand verändert wird. Im unteren Bereich werden die Aktionen, die das Objekt ausführt angegeben. Die Worte "entry", "do" und "exit" sind reserviert und dürfen nicht für Nachrichten oder Ereignisse verwendet werden. "Entry" bezeichnet eine Aktivität, die beim Eintritt in den Zustand ausgeführt wird. "Do" bezeichnet eine komplexe Aktivität, die lange andauern kann. Sie kann in einem weiteren Zustandsdiagramm "verfeinert" werden. Damit können verschachtelte Zustandsdiagramme aufgebaut werden. "Exit" bezeichnet eine Aktivität, die beim Verlassen des Zustandes ausgeführt wird. **Beispiel:** Das Beispiel beschreibt den Zustand "PasswortEingabe" einer GUI-Klasse, die zur Eingabe von Passwörtern dient. In diesem Zustand wird das Zustandsattribut (Variable) "Passwort" verändert. Beim Eintritt in den Zustand führt das Objekt die Aktivität "EchoAusschalten" aus. Die Aktivität "ZeichenEingabe" nimmt das Passwort entgegen, und die Aktivität "EchoEinschalten" wird beim Verlassen des Zustandes ausgeführt.

Anwendungsbereich

Zustandsdiagramme werden meistens im Design eingesetzt. Sie werden nur für Klassen mit nichttrivialem (interessantem) Lebenszyklus modelliert. Ein trivialer Lebenszyklus ist dann vorhanden, wenn ein Zustandsdiagramm nicht zum besseren Verständnis des Verhaltens des Objektes beiträgt. Nichttriviale Lebenszykeln von Objekten liegen dann vor, wenn die Objekte "viele" Zustände haben und in den einzelnen Zuständen nur auf eine bestimmte Art und Weise auf eingehende Nachrichten reagieren können.

Zusammenhang

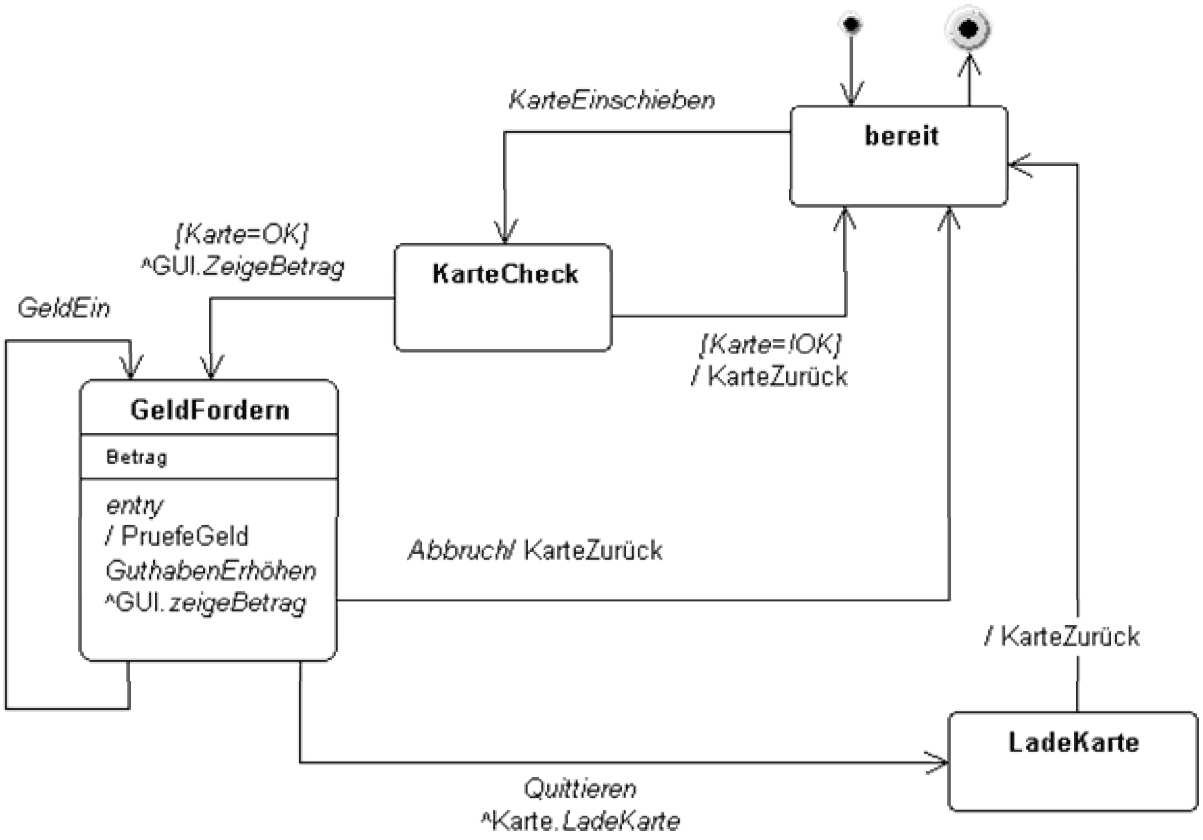
Ein Zustandsdiagramm bezieht sich immer auf eine Klasse des Klassendiagramms. Die Nachrichten auf die eine Klasse mit einem Zustandsübergang reagiert, werden von anderen Klassen gesendet. Alle Nachrichten die zu Zustandsübergängen führen müssen folglich in den Sequenzdiagrammen des UML-Modells an die Klasse gesendet werden, für die ein Zustandsdiagramm modelliert wird. Ein Zustandsdiagramm ist also auch eine Zusammenfassung aller Sequenzdiagramme, in denen eine Klasse auftaucht.

Hinweise

Die Unterscheidung zwischen Aktionen und Aktivitäten wurde aus der Methode OMT von Rumbaugh übernommen. Rumbaugh unterscheidet dort ganz akademisch zwischen Aktivitäten (Operationen die Zeit zur Durchführung benötigen) und Aktionen (Operationen, die keine Zeit zur Ausführung benötigen). Da selbstverständlich die Ausführung jeder Operation Zeit benötigt ist diese Unterscheidung nicht absolut zu sehen, sondern relativ zu dem Zeitraster das für ein zu lösendes Problem relevant ist.

Salopp ausgedrückt werden Operationen als Aktionen modelliert, wenn ihre Ausführung "schnell" geht, z. B das Inkrementieren eines Zählers. Alle "länger" dauernden Operationen werden als Aktivitäten modelliert.

Beispiel



Dieses State Diagramm beschreibt das Verhalten des Mensa-Karten-Automaten in der FH-Mensa. Der Automat dient dem Aufladen von Karten mit Geld. Mit den Karten kann dann an der Kasse bezahlt werden. Mensa Kunden schieben eine Plastikkarte in den Automaten. Danach geben sie Geldscheine ein. Der Gesamtbetrag wird auf der Karte gespeichert und beim Bezahlen an der Kasse von der Karte abgebucht.

Wird der Automat nicht benutzt, so verharrt er im Zustand "bereit". Die Nachricht "KarteEinschieben" führt zum Zustandsübergang in den Zustand "KarteCheck". Hier wird die Karte geprüft. Ist sie gültig (Bedingung [Karte=OK] ist wahr) dann sendet er bei dem Zustandswechsel in den Zustand "GeldFordern" an das Objekt GUI die Nachricht ZeigeBetrag. In dem Zustand "GeldForder" führt er die Eingangsaktivität PruefeGeld aus. Über GuthabenErhöhen merkt er sich die bezahlte Summe. Er bleibt so lange in diesem Zustand bis er vom Benutzer die Nachricht Abbruch oder Quittieren erhält.
Das vom Benutzer initiierte Ereignis "Abbruch" führt zur Aktion "KarteZurück". Der Automat wechselt in den Zustand "bereit".
Durch Drücken des Quittieren-Knopfes löst der Benutzer das Ereignis "Quittieren" aus, der Automat wechselt in den Zustand "LadeKarte" und sendet der Karte die Nachricht "LadeKarte". Danach wirft er die Karte aus. Diese letzte Transition wird im Gegensatz zu den übrigen Transitionen in diesem Diagramm von dem Objekt "MensaAutomat" selbst ausgelöst. Die anderen Transitionen werden von dem externen Objekt Benutzer verursacht.

Packagediagramme

Unable to read strip image!

[01.12.2004] m.guist@fbi.fh- darmstadt.de