



UML

UNIFIED MODELING LANGUAGE

INHALTSVERZEICHNIS

INHALTSVERZEICHNIS	2
1. ALLGEMEINES	4
1.1. Historische Entwicklung von UML:.....	4
1.2. Notation und Methodik.....	4
2. VORGEHENSWEISE OD. DER UMRIS EINES ENTWICKLUNGSPROZESSES	5
2.1. Vorstudie, Anforderungsanalyse und Definition.....	5
2.2. Grobdesign- Entwurf und Komponentenbildung	6
2.3. Iterative inkrementelle Entwicklung; Implementierung	6
2.4. Systemtest und Einführung	6
3. UML- DIE DIAGRAMMTYPEN IM ÜBERBLICK.....	7
4. USE CASE – OD. ANWENDUNGSFALLDIAGRAMM.....	8
4.1. Was ist ein Anwendungsfall od. ein <i>use case</i> ?	8
4.2. Akteure od. <i>actor</i>	8
4.3. Notation.....	8
4.4. Anwendungsfalldiagramm	9
5. KLASSENDIAGRAMM.....	11
5.1. Notation Klasse und Objekt	11
5.2. Elemente eines Klassendiagramms (Beziehungselemente)	12
5.3. Beispiel.....	14
6. ZUSTANDSDIAGRAMM	15
6.1. Beschreibung	15
6.2. Notation und Beispiel:.....	15

7. AKTIVITÄTSDIAGRAMM.....	16
7.1. Beschreibung	16
7.2. Notation und Beispiel	16
8. VERHALTENS OD. INTERAKTIONSDIAGRAMME	17
8.1. Sequenzdiagramm.....	18
8.2. Kollaborationsdiagramm	19
9. KOMPONENTEN- UND VERTEILUNGSDIAGRAMM	20
9.1. Komponentendiagramm	20
9.2. Verteilungs od. Deployment- Diagramm.....	21
Literaturverzeichnis	21
Anhang A: CRC- Karten	22
Anhang B: Java Klassen	23

1. ALLGEMEINES

1.1. Historische Entwicklung von UML:

Die Idee der Objektorientierung ist schon 30 Jahre alt und hatte mit objektorientierten Programmiersprachen wie Simula und Smalltalk ihren Anfang.

Anfang der 90er Jahre kam eine Welle von objektorientierten Analyse und Designmethoden auf. Die Methoden von Grady Booch (Booch 91), James Rumbaugh (OMT) und Jacobsen mit OOSE haben sich als die beliebtesten herauskristallisiert. Booch und Rumbaugh begannen ihre Methoden zusammenzufügen und entwickelten eine neue Notation- die UM od. *Unified Method*. Kurze Zeit später gesellte sich auch Ivar Jacobson dazu und brachte seine Use Cases od. Anwendungsfälle in die UM ein.

Sie nannten sich die drei Amigos und entwickelten mit ihrer *Unified Modeling Language* UML einen Quasi Standard, der 1997 bei der Object Management Group (OMG) zur Standardisierung eingereicht wurde.

UML 1.1 wurde akzeptiert- die derzeit aktuelle Version ist 1.2 – die jedoch nur wenige Änderungen zur Version 1.1 aufweist. Die Weiterentwicklung von UML wird derzeit durch die OMG betrieben.

1.2. Notation und Methodik

„Die *Unified Modeling Language* (UML) ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme. Die UML berücksichtigt die gestiegenen Anforderungen bezüglich der Komplexität heutiger Systeme, deckt ein breites Spektrum von Anwendungsgebieten ab und eignet sich für konkurrierende, verteilte, zeitkritische, sozial eingebettete Systeme uvm“¹

UML ist in erster Linie die Beschreibung einer einheitlichen Notation zur Modellierung, sie ist jedoch bewußt keine Methode. UML kann Basis für versch. Methoden sein, aber grundsätzlich muß eine Methode die individuellen Rahmenbedingungen und das Umfeld usw. berücksichtigen.

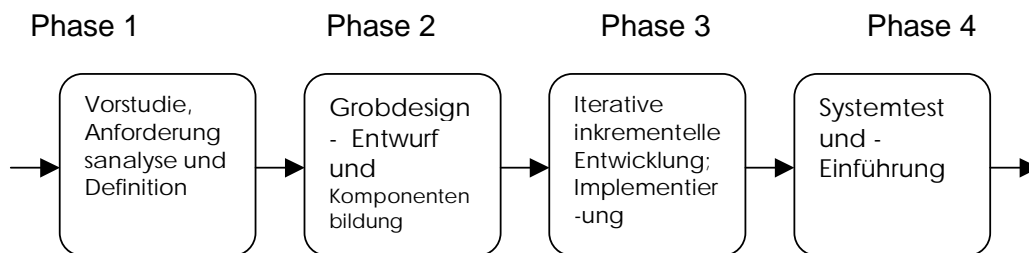
Die *Unified Modeling Language* ist eine Modellierungssprache und eignet sich deswegen hervorragend zur Übersetzung in eine objektorientierte Programmiersprache.

¹ aus [Oestereich'98]

2. VORGEHENSWEISE OD. DER UMRISß EINES ENTWICKLUNGSPROZESSES

Wie oben erwähnt ist die UML eine Modellierungssprache, jedoch keine Methode- sie enthält keine Prozeßbeschreibung, die einen wichtigen Teil einer Methode ausmacht.

Ich gehe nur grob auf einen Entwicklungsprozeß ein, und beschreibe kurz die einzelnen Phasen der Entwicklung. In der Vorlesung wurde dieser Punkt als Software Engineering mit dem klassischen Phasenmodell erläutert.



2.1. Vorstudie, Anforderungsanalyse und Definition

Der wichtigste Teil hierbei ist es, mit den zukünftigen AnwenderInnen zu kommunizieren und damit den Anwendungsbereich aus der Praxis kennenzulernen. Es müssen die genauen Anforderungen an die zu erstellende Software definiert werden. Hier stellt sich hauptsächlich die Frage: **Was** soll das Produkt können, und nicht **wie** wird das Ergebnis erreicht. Natürlich fällt in diesen Bereich auch eine Kosten/Nutzen Analyse.

Das Ergebnis der Analyse und Definition ist die Produktdefinition, die den Leistungsumfang des Produkts festlegt.

Die Produktdefinition muß laut Vorlesung aus Software Engineering WS98/99 folgende Festlegungen enthalten:

- Funktionsumfang
- Benutzeroberfläche
- Schnittstellen zur Systemumgebung (Hardware und Software)
- Hardware und Softwarebasis
- Leistungsverhalten
- Dokumentation
- Terminplanung

2.2. Grobdesign- Entwurf und Komponentenbildung

Ein wichtiges Ergebnis beim Entwurf ist eine grundlegende Architektur für das System. Diese Architektur ist die Grundlage für die Entwicklung; sie dient als Bauplan für spätere Phasen. Es soll zur Bewältigung der Komplexität das Gesamtprodukt in kleine, beherrschbare Einheiten zerlegt werden. Das Ergebnis ist hier eine klare Software- Spezifikation.

2.3. Iterative inkrementelle Entwicklung; Implementierung

Bei der Implementierung des Entwurfs erstellt man das System in einer Folge von Iterationen. Jede Iteration ist ein Miniprojekt, das programmiert, getestet und integriert wird. Sie besteht jeweils aus Analyse, Design, Realisierung und Test einer Menge von Teilfunktionalitäten. Das Testen ist hier besonders wichtig, um am Ende ein korrektes System zu erhalten. Das Ergebnis ist eine Sammlung von Quelltexten die am Schluß zusammengefügt werden.

2.4. Systemtest und Einführung

Hier gibt es kein Hinzufügen von Funktionalität. Es werden nur noch Fehlerkorrekturen vorgenommen. (von der Beta bis zur Endversion) Integrationstest, Effizienzstest, Installation und Abnahmetest sind die wichtigsten Punkte bei der letzten Phase eine Entwicklungsprozesses.

3. UML- DIE DIAGRAMMTYPEN IM ÜBERBLICK

Die Notation der UML umfaßt Diagramme für die Darstellung der verschiedenen Ansichten auf das System, vergleichbar mit Bauplänen für Häuser. Auch hier gibt es z. B. einen Grundriß, einen Lageplan, verschiedene Außenansichten und Werkpläne für die Handwerker. Für eine spezielle Aufgabe ist meist eine Diagrammart besser geeignet als andere.

Insgesamt umfaßt die UML folgende Diagrammtypen:

- **Anwendungsfalldiagramm (Use Case Diagramm):** zeigt Akteure, Anwendungsfälle und ihre Beziehungen. Einsatz in Phase: 1, 2, 3 und 4
- **Klassendiagramm:** Zeigt Klassen und ihre Beziehungen untereinander. Einsatz in Phase: 2 und 3. Das Klassendiagramm ist das wichtigste Diagramm der UML.
- **Verhaltensdiagramme (behavior diagrams):** Zeigen den Nachrichtenfluß und damit die Zusammenarbeit der Objekte im zeitlichen Ablauf. Einsatz in Phase: 1, 2, 3 und 4
 - **Aktivitätsdiagramm:** zeigt Aktivitäten, Objektzustände, Zustände, Zustandsübergänge und Ereignisse. Einsatz in Phase: 2 und 3
 - **Kollaborationsdiagramm (collaboration diagram):** zeigt Objekte und ihre Beziehungen inklusive ihres räumlich geordneten Nachrichtenaustausches.
 - **Sequenzdiagramm:** zeigt Objekte und ihre Beziehungen inklusive ihres zeitlich geordneten Nachrichtenaustausches.
 - **Zustandsdiagramm:** zeigt Zustände, Zustandsübergänge und Ereignisse also das dynamische Verhalten. Einsatz in Phase: 1, 2, 3 und 4.
- **Implementierungsdiagramme:** Besonders für die Darstellung von verteilten Anwendungen und Komponenten; allgemein: Darstellung von Implementierungsaspekten (Übersetzungseinheiten, ausführbare Programme, Hardwarestruktur) Einsatz in Phase: 2, 3 und 4
 - **Komponentendiagramm:** zeigt Komponenten und ihre Beziehungen also die Zusammenhänge der Software.
 - **Verteilungsdiagramm (deployment diagram):** zeigt Komponenten, Knoten und ihre Beziehungen. (Hardwareaufbau)

4. USE CASE – OD. ANWENDUNGSFALLDIAGRAMM

4.1. Was ist ein Anwendungsfall od. ein use case ?

Definition laut [Oestereich'98]: „ Ein Anwendungsfall beschreibt eine Menge von Aktivitäten eines Systems aus der Sicht seiner Akteure, die für die Akteure zu einem wahrnehmbaren Ergebnis führen. Ein Anwendungsfall wird stets durch einen Akteur initiiert. Ein Anwendungsfall ist ansonsten eine komplette, unteilbare Beschreibung „

Ein wenig klarer Ausgedrückt ist ein Anwendungsfall eine typische Interaktion zwischen einem Benutzer und einem Computersystem. Es werden Anforderungen wie : was das System leisten muß gestellt; jedoch nicht wie es das System leisten muß.

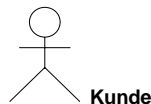
Daraus folgt, daß ein Anwendungsfall also einen typischen Arbeitsablauf beschreibt.

4.2. Akteure od. actor

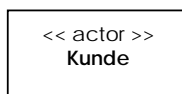
Akteure sind die vom Anwender – in Bezug auf das System – eingenommenen Rollen (wie z.B. Kunde, Verkäufer ...). Meist gibt es viele Kunden, aus der Sicht des Systems haben aber alle Kunden dieselbe Rolle. Wenn nicht, ist ein neuer Akteure nötig.

4.3. Notation

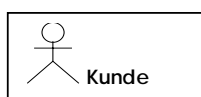
Akteure können in versch. Weise dargestellt werden: als textuelles Stereotyp, als visuelles Stereotyp od. in einer gemischten Form.



Akteur; am Anwendungsfall beteiligte Person (visuelles Stereotyp)



textuelles Stereotyp



textuelles und visuelles Stereotyp

Einschub: Was ist ein Stereotyp ?

Stereotypen sind Mechanismen, die es erlauben, die UML zu erweitern. Sie haben in der UML die Bedeutung „so ähnlich wie etwa“. Stereotypen können eigene Piktogramme (Icons) haben und werden in eckigen Klammern dargestellt, z.B.:

`<<uses>>`. Sie erlauben eine weitere Einteilung der Klassen, Abhängigkeiten, Assoziationen etc. So ist z.B. eine Einteilung der Klassen in Schnittstellen-, Kontroll- und Entitätenobjekte (irgendwelche Dinge) möglich.

Ein Modellierungselement kann mit beliebig vielen Stereotypen klassifiziert werden. Dadurch werden die Semantik und visuelle Darstellung des Elements beeinflusst.

Beispiele von Stereotypen:

visuelle Stereotypen: siehe oben 

textuelle Stereotypen:

`<<präsentation>>`, `<<vorgang>>`, `<<fachklasse>>` (hier werden die Bedeutungen einer Klasse in der Anwendungsarchitektur angegeben)

`<<model>>`, `<<view>>`, `<<controller>>`, `<<exception>>`, `<<uses>>`, `<<extends>>`
`<<interface>>` (mit interface wird eine abstrakte Klasse gekennzeichnet, die nur abstrakte Operationen beinhaltet)

`<<implements>>` deklariert zwischen zwei Klassen eine Verfeinerungsbeziehung

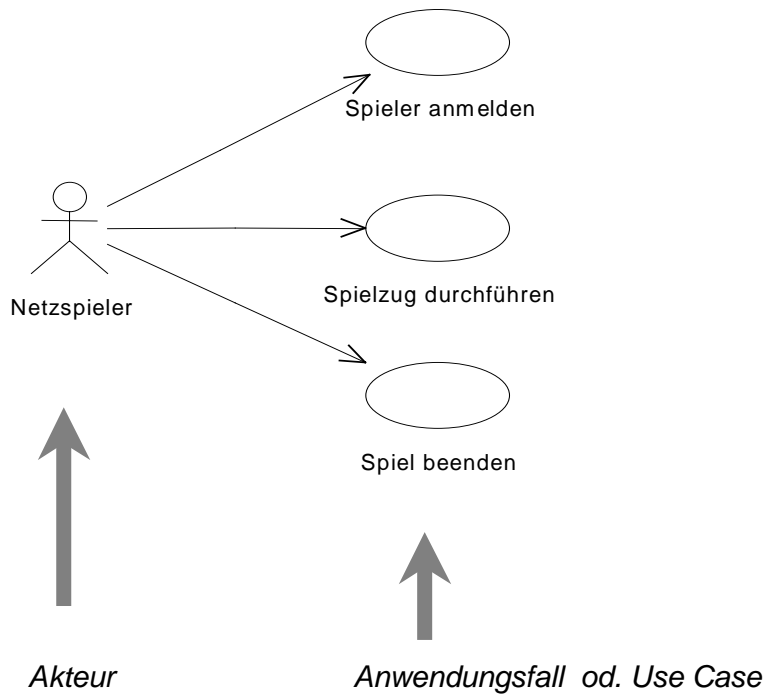
4.4. Anwendungsfalldiagramm

Ein Anwendungsfalldiagramm beschreibt die Zusammenhänge zwischen einer Menge von Anwendungsfällen und den daran beteiligten Akteuren. Es bildet somit den Kontext und eine Gliederung für die Beschreibung, wie mit einem Geschäftsvorfall umgegangen wird.

Anwendungsfälle beschreiben gewöhnlich nur die Aktivitäten, die durch die zu entwickelnde Software unterstützt werden sollen, und deren Berührungspunkte zum Umfeld dieser Software.

Ein Anwendungsfalldiagramm enthält eine Menge von Anwendungsfällen, die durch einzelne Ellipsen dargestellt werden und eine Menge von Akteuren und Ereignissen, die daran beteiligt sind. Sie können außerdem hierarchisch verschachtelt werden (siehe unten)

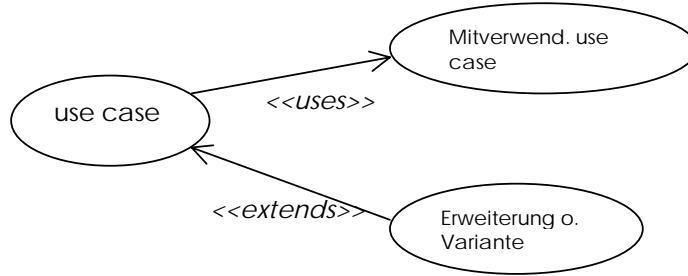
Wobei der Anwendungsfall „Spielzug durchführen“ ein eigenes Anwendungsfalldiagramm darstellt:



Verbindungstypen:

Untenstehende Stereotypen sind nützliche, aber entbehrliche Modellkonstrukte, manche ModelliererInnen verzichten darauf- andere wiederum schwören darauf. (vgl [Fowler'97] od. [Oestereich'98]) Ab der Version UML 1.3 gibt es eine 3. Stereotyp: Die Generalisierungsbeziehung, in der Subanwendungsfälle von den Super-Anwendungsfällen Verhalten und Bedeutung erben kann (analog zur Generalisierungsbeziehung zwischen Klassen)

- `<<uses>>`, `<<benutzt>>` (ab UML 1.3 umbenannt in `<<include>>`)
wird verwendet, wenn zwei od. mehr Use Cases einen ähnlichen Aufbau haben und Verhalten durch Kopieren wiederholt dargestellt werden muß. Also wenn die gleiche Use Case Beschreibung in verschiedenen Use Cases vorkommt. Um dies zu vermeiden, wird der entsprechende Teil separiert und mit einer `<<uses>>` Beziehung wieder in die andere Anwendungsfallbeschreibung eingebunden. Dadurch ist es möglich, Verhalten von den Use-Cases in einen separaten Use-Case zu verlagern.
- `<<extends>>`, `<<erweitert>>`
Dieser Verbindungstyp klammert auch Verhalten aus, jedoch ist die Zielsetzung eine andere. Bei `<<extends>>` wird das Verhalten erweitert. (z.B. für spezielle Abweichungen oder Erweiterungen)



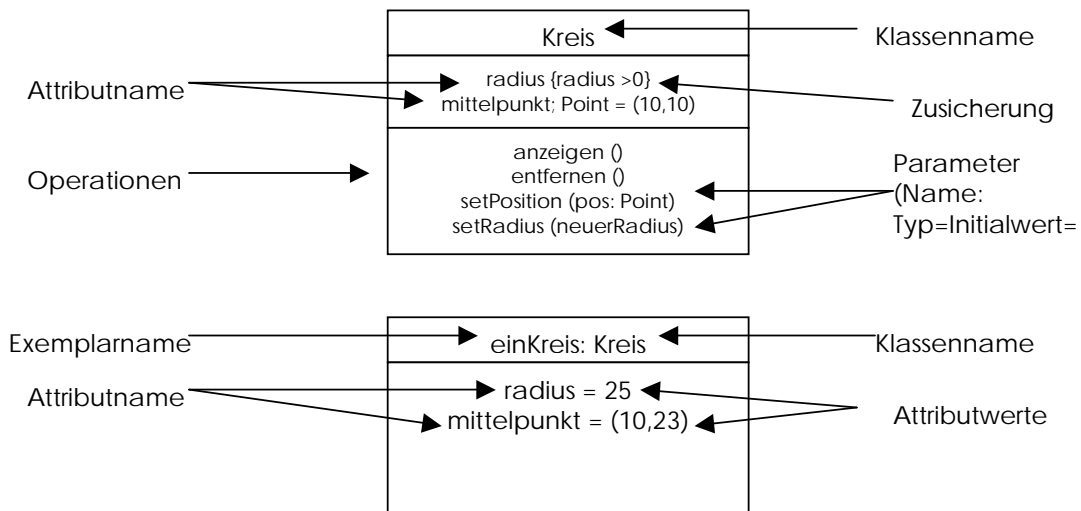
5. KLASSENDIAGRAMM

Klassendiagramme sind der zentrale Bestandteil der UML und auch zahlreicher objektorientierter Methoden. Wie die Klassen ermittelt werden, darüber gibt die UML keine Auskunft; hierfür gibt es andere Techniken, z.B.: CRC-Karten (Abkürzung für „Class, Responsibility and Collaboration“) auf die ich in Punkt 10 - Anhang - auf Seite 21 später kurz eingehen möchte.

Die UML beschreibt lediglich die Notation und die Semantik:

5.1. Notation Klasse und Objekt

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse (fett) oder zusätzlich auch Attribute und Operationen tragen. Klassennamen beginnen mit einem Großbuchstaben. Zum besseren Verständnis hier ein kleines Beispiel: Eine Klasse Kreis würde beispielsweise die Attribute *radius* und *position* sowie die Operationen *anzeigen()*, *entfernen()*, *setPosition(pos)* und *setRadius(neuerRadius)* beinhalten. Das darunterliegende Objekt mit dem Namen: einKreis, welches Exemplar der Klasse ist. Objektname werden unterstrichen.



Metaklasse: Klassen für die Klassenobjekte werden Metaklassen genannt und ähnlich wie eine normale Klasse mit dem Stereotyp <<metaclass>> notiert. In Smalltalk sind Klassen grundsätzlich Instanzen ihrer Metaklassen. Die Metaklassen sind selbst Instanzen der Klasse *MetaClass*, die ist allerdings wieder Instanz der Klasse *MetaClassClass*. Klassenoperationen müssen in der UML nicht innerhalb der Metaklasse notiert werden, sie können auch in der Klasse selbst enthalten sein, wobei sie dann unterstrichen werden, um sie von normalen Operationen unterscheiden zu können.

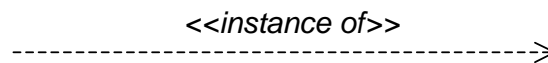
Parametrisierbare Klasse: Bei einer parametrisierbaren Klasse wird keine konkrete Klasse definiert, sondern lediglich eine Schablone (template) zur Erzeugung von Klassen. Bei diesen Schablonen handelt es sich meist um einfache Makros, die hauptsächlich Textersetzung durchführen.

Abstrakte Klasse: Eine abstrakte Klasse ist eine Oberklasse, die selbst keine Objekte instanzieren kann. So kann Vehikel ein abstrakte Klasse sein, von der man selbst kein Objekt haben kann, und Auto, Flugzeug, usw sind konkrete Klassen. Eine abstrakte Klasse ist also eine Verallgemeinerung od. Abstraktion.

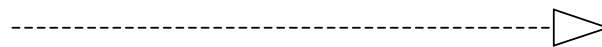
5.2. Elemente eines Klassendiagramms (Beziehungselemente)

- Abhängigkeitsbeziehungen: werden dargestellt durch einen gestrichelten Pfeil, wobei der Pfeil von einem abhängigen auf das unabhängige Element zeigt:

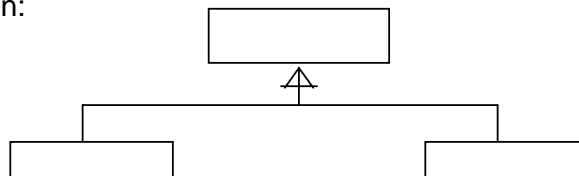
Klassen – Objekt Beziehungen werden so dargestellt, daß das Objekt auf seine Klasse zeigt.



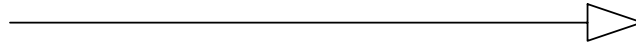
- Verfeinerungsbeziehungen: werden dargestellt als gestrichelter Generalisierungspfeil in Richtung auf das „Hauptelement“:



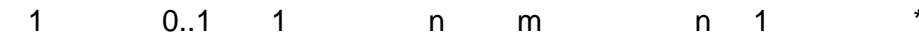
- Vererbung: Bei einer Vererbung werden die Eigenschaften und Operationen der Oberklasse an die Unterklassen weitervererbt.
Notation:



- Generalisierung- Spezialisierung: Bei der Generalisierung – Spezialisierung werden Eigenschaften hierarchisch gegliedert, das bedeutet, das allgemeine Eigenschaften Oberklassen zugeordnet werden, und spezielle werden Unterklassenzugeteilt. Somit erben die Unterklassen die allgemeinen Eigenschaften der Oberklasse. Der Pfeil zeigt von der Unterklasse zur Oberklasse:



- Assoziation: Werden durch eine Linie zwischen den Beteiligten Klassen dargestellt. Sie stellt eine allgemeine Beziehung zwischen 2 Klassen dar und sind notwendig, damit Objekte miteinander kommunizieren können.
Multiplizität: Die Multiplizität einer Assoziatin gibt an, mit wievielen Objekten der anderen Klasse ein Objekt assoziiert sein kann. Man spricht auch von Kardinalität. Liegt das Minimum bei 0, bedeutet das, daß die Beziehung optional ist. Ein * bedeutet „unbestimmt – od. Joker“- man kann alles dafür einsetzen:



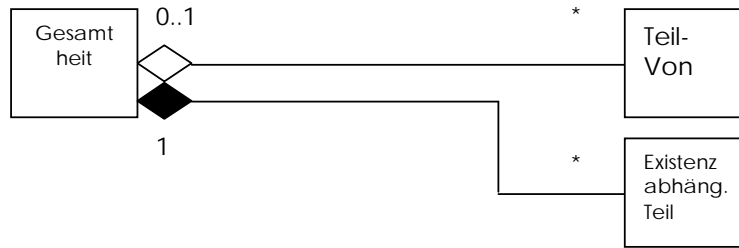
Es gibt viele weitere Arten der Assoziation wie z.B. Rekursive Assoziation, Attributierte Assoziation, Assoziationszusicherung, Qualifizierte Assoziation, Abgeleitete Assoziation, Mehrgliedrige Assoziation, Gerichtete Assoziation. Im Rahmen dieser Arbeit gehe ich nicht näher auf die einzelnen Arten ein, es würde den Rahmen sprengen. Eine gute Beschreibung gibt es in **[Oestereich'98]**- ab Seite 259.

- Aggregation: Unter einer Aggregation versteht man die Zusammensetzung eines Objektes aus einer Menge von Einzelteilen. Sie gibt an, daß eine Klasse „Teil-Von“ in einer Klasse „Gesamtheit“ enthalten ist. (IST-TEIL-VON-Beziehung). Eine Aggregation wird durch eine Linie zwischen 2 Klassen dargestellt, bei der zusätzlich auf der Seite des Aggregats, als des Ganzen (mit der Führungsrolle), eine kleine Raute steht:



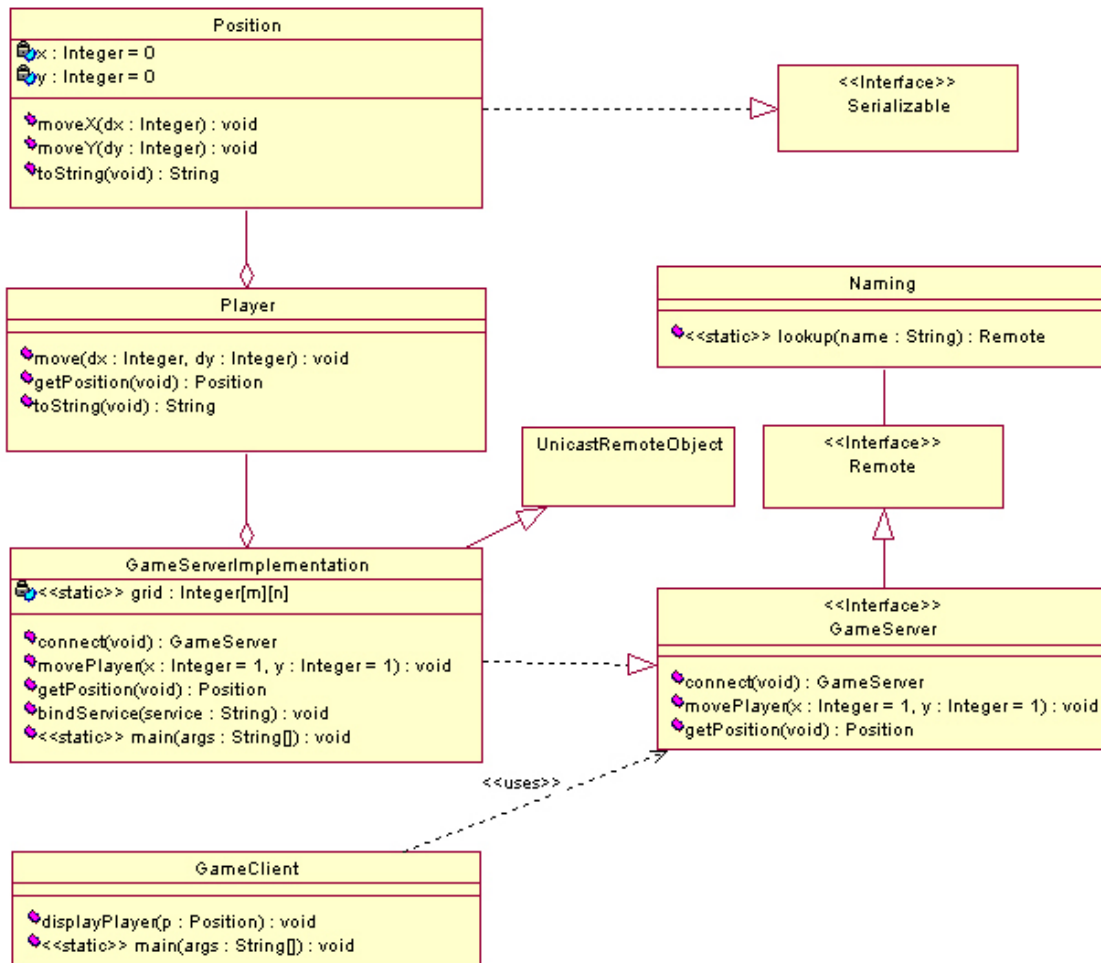
- Komposition: Eine Komposition ist eine stärkere Form der Aggregation, für sie gelten die gleichen Regeln. Das bedeutet, daß die Kardinalität

auf der Seite des Aggregats nur 1 sein kann (nicht wie oben 0..1). Der Teil des Aggregats ist existenzabhängig und hat also die gleiche Lebensdauer wie das Ganze selbst. Notation:



5.3. Beispiel

Die zu diesem Klassendiagramm in Java programmierten ausprogrammierten Klassen sind im Anhang zu finden.



6. ZUSTANDSDIAGRAMM

6.1. Beschreibung

Aus den Interaktionsdiagrammen können Zustandsdiagramme entwickelt werden. Sie beschreiben das Verhalten eines Systems. Alle Szenarios definieren ein bestimmtes Zustandsdiagramm genauer. Die Szenarios sind so zu wählen, daß das Zustandsdiagramm einer Klasse vollständig definiert wird.

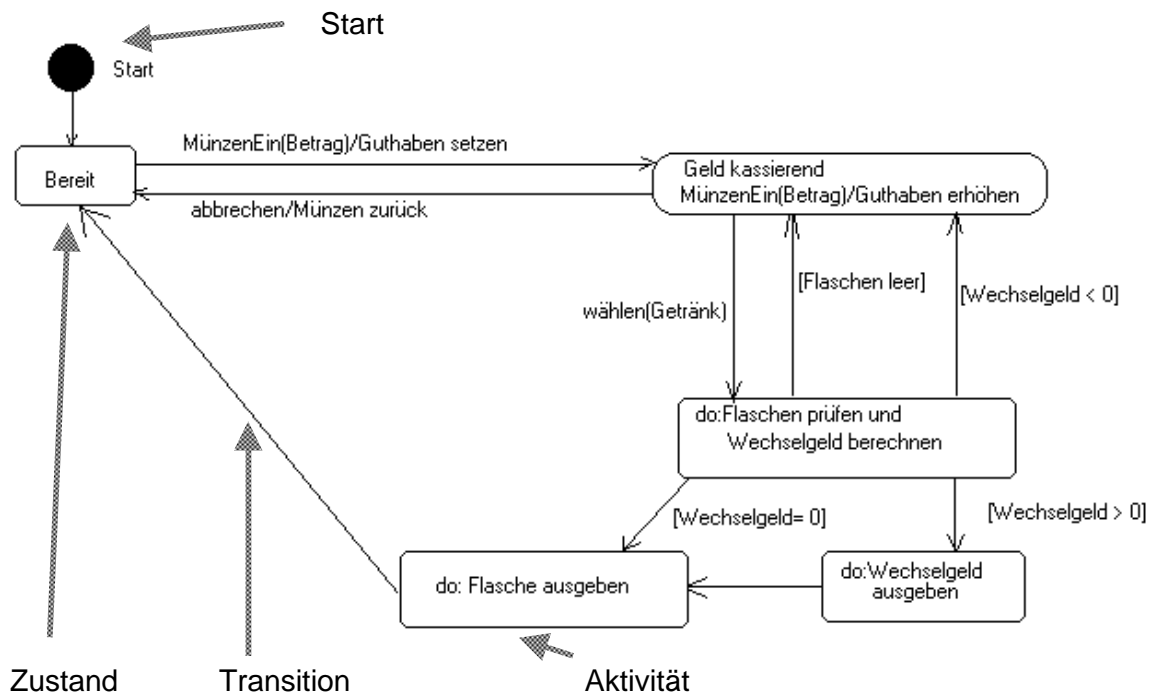
Zustandsdiagramme zeigen einfach eine Folge von Zuständen, die ein Objekt im Laufe seines Bestehens einnehmen kann. Zustandsdiagramme beschreiben endliche Automaten.

6.2. Notation und Beispiel:

Zustände werden durch abgerundete Rechtecke dargestellt. Sie können einen Namen und (durch einen horizontalen Strich getrennt) Zustandsvariablen besitzen.

Am besten wird die Funktion eines Zustandsdiagrammes Anhand eines Beispiels erklärt:

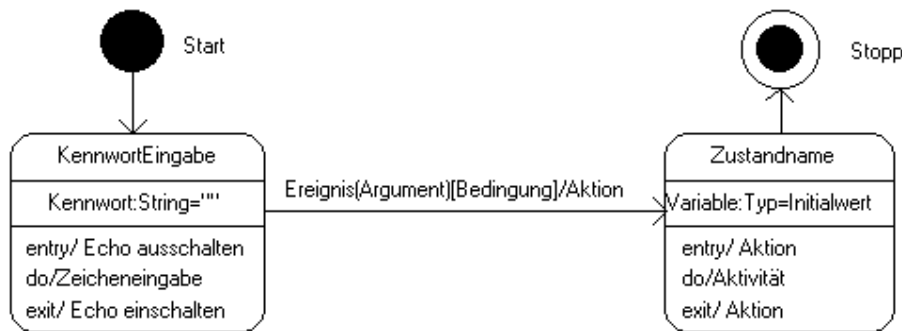
Zustandsdiagramm für einen Getränkeautomaten:



Durch das Eintreffen von Ereignissen, kann ein anderer Zustand erreicht werden (was durch die Pfeile symbolisiert wird)

Eine genaue Notation siehe unten:

entry, do und exit sind als Wörter reserviert, und können nicht als Bezeichnung für einen Zustand verwendet werden. (entry gibt an was zu tun ist, wenn man in einen Zustand kommt, do ruft die Aktivität auf, und exit kennzeichnet das Verlassen eines Zustandes)



7. AKTIVITÄTSDIAGRAMM

7.1. Beschreibung

Ein Aktivitätsdiagramm ist eine spezielle Form des Zustandsdiagramms, das überwiegend oder ausschließlich Aktivitäten enthält. Eine Aktivität ist ein einzelner Schritt in einem Ablauf. Sie ist ein Zustand, in der ein Vorgang abläuft.

Aktivitätsdiagramme sind ähnlich den Flussdiagrammen, jedoch sind alle Aktivitäten eindeutig Objekten zugeordnet.

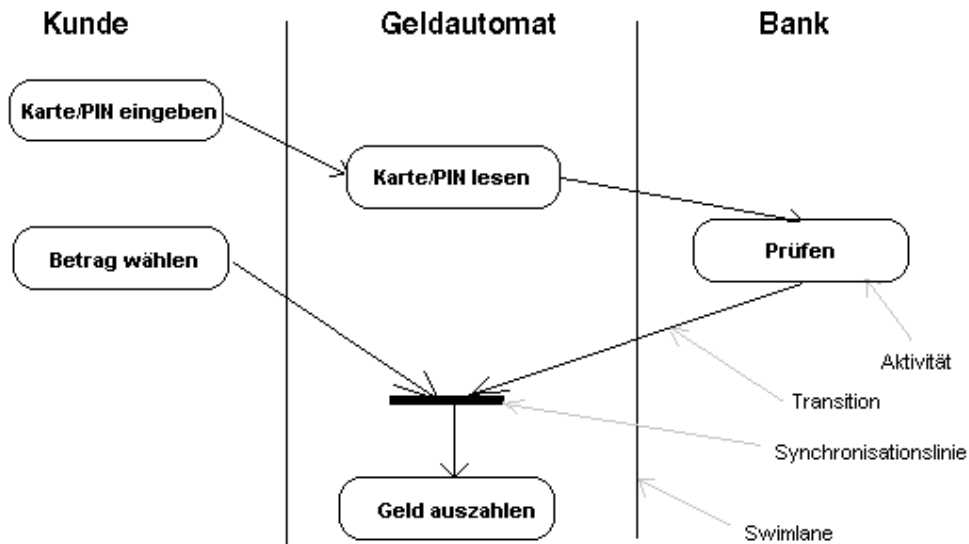
7.2. Notation und Beispiel

Grundelemente dieser Diagramme sind

- Aktivitäten: Zustände in denen Vorgänge ablaufen
- Transitionen: erfolgen automatisch am Ende der Aktivitäten; sie werden durch Pfeile dargestellt
- Synchronisationslinien: werden durch Striche dargestellt und schalten, wenn alle Engangstransitionen vorhanden sind.

- Swimlanes od. Verantwortlichkeitsbereiche: teilen ein Aktivitätsdiagramm so ein, daß die Bereiche, die sie abgrenzen, einzelnen Klassen zugeordnet werden können.

Beispiel: Geldbehebung via Bankomat.



8. VERHALTENS OD. INTERAKTIONSDIAGRAMME

Es gibt 2 Arten von Verhaltensdiagrammen:

- Sequenzdiagramme und
- Kollaborationsdiagramme

Beide beschreiben die zeitlichen Abläufe, das heißt die Aufrufsequenzen. Im Grunde beschreiben sie exakt das selbe, die Darstellungsart ist jedoch verschieden.

Beim Erstellen dieser Diagramme beschränkt man sich auf die wichtigsten Szenarios (oft werden sie auch Szenariodiagramme genannt)- später dann werden Sonderfälle miteinbezogen.

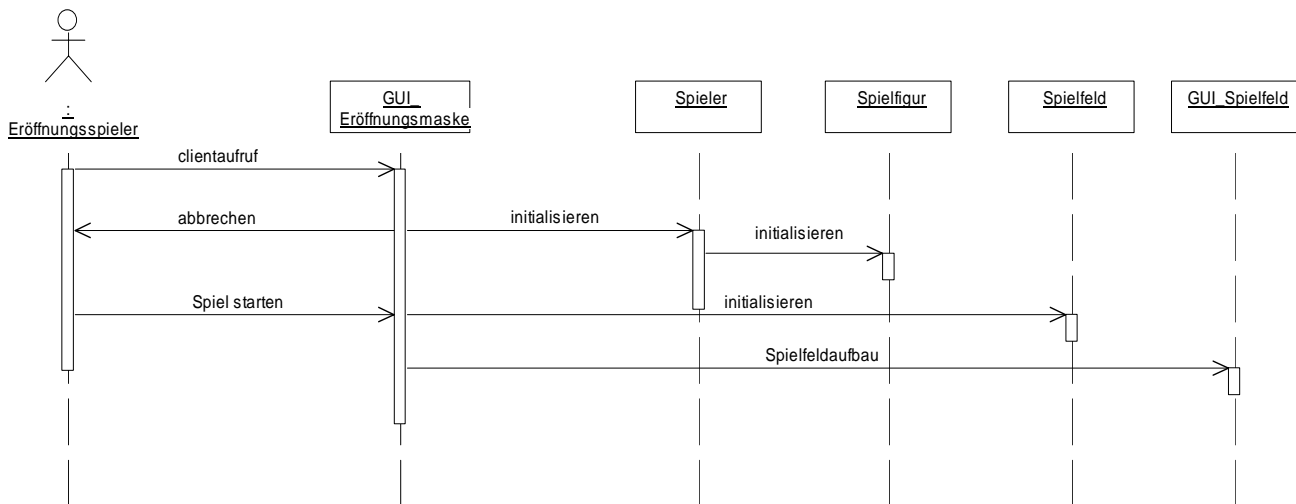
8.1. Sequenzdiagramm

Die Darstellung der Objekte erfolgt durch gestrichelte senkrechte Linien („Lebenslinien“). Oben über der Linie steht der Name bzw. das Objektsymbol. Die Nachrichten werden als Pfeile dargestellt, auf denen der Name notiert ist. Nachrichten können auch Bedingungen haben, die dann in eckigen Klammern angegeben wird.

Die Balken, die vertikal über den Lebenslinien liegen, kennzeichnen den Steuerungsfokus, d.h. welches Objekt gerade aktiv ist. (Dieser Balken wird auch „Focus of Control“ genannt). Das Löschen eines Objektes wird durch ein X am Ende des Steuerungsfokus-balkens gekennzeichnet.

Die Zeitachse verläuft von oben nach unten.

Hier ein Beispiel aus der Anmeldesequenz beim Netzwerkspiel: Mensch-ärgere-dich-nicht.



Ein Eröffnungsspieler startet über einen Internetbrowser ein Java-Applet (GUI-Eingabemaske-Clientseitig). Beim nächsten Schritt werden seine Daten in der Klasse Spieler gespeichert sowie die Spielfiguren initialisiert. Nachdem sich auch andere Spieler angemeldet haben, kann der Eröffnungsspieler das Spiel starten(über die GUI-Eingabemaske). Danach wird die Klasse Spielfeld initialisiert und die grafische Oberfläche für das Spielen aufgebaut

8.2. Kollaborationsdiagramm

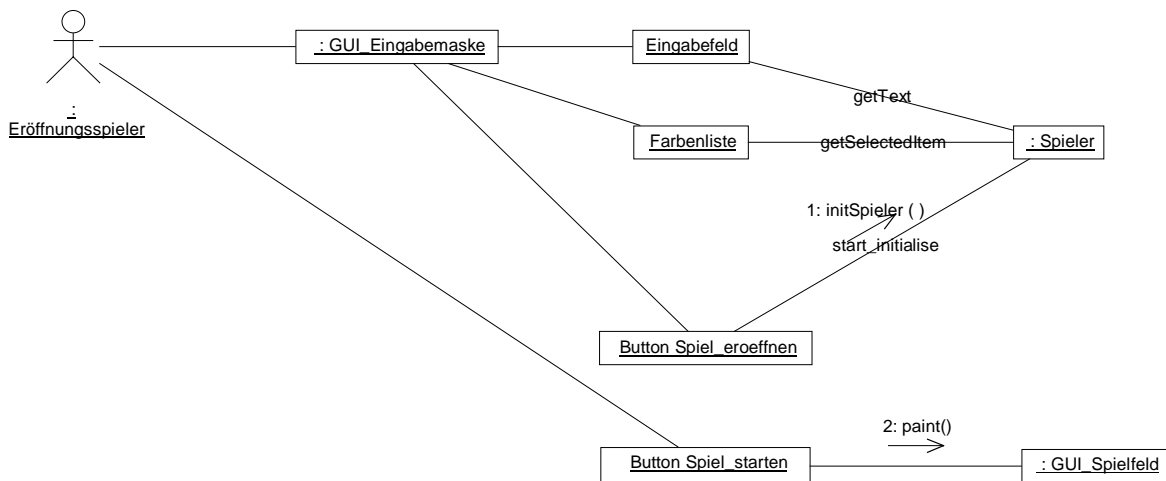
Das Kollaborationsdiagramm und das Sequenzdiagramm beinhalten die gleichen Informationen und unterscheiden sich lediglich in der Darstellung. Die automatische Umwandlung eines Sequenzdiagramms in ein Kollaborationsdiagramm und umgekehrt ist möglich. Bei vielen Klassen und wenigen Nachrichten sind Kollaborationsdiagramme übersichtlicher als Sequenzdiagramme. Sind wenige Klassen und viele Nachrichten vorhanden, so ist das Sequenzdiagramm besser geeignet.

Beim Kollaborationsdiagramm stehen die Objekte und ihre Zusammenarbeit (Kollaboration) untereinander im Vordergrund; zwischen ihnen werden bestimmte Nachrichten dargestellt.

Notation:

Zwischen den Objekten werden Assoziationslinien gezeichnet, auf denen dann die Nachrichten notiert werden.

Beispiel- Mensch-ärgere-dich-nicht:



Ein Eröffnungsspieler kann seine Daten über die GUI_Eröffnungsmaske eingeben. Dazu zur Verfügung hat er ein TextField Eingabefeld und eine List Farbenliste. Nachdem er seine Auswahl getroffen hat, kann er sein Spiel durch Drücken des

Button Spiel_eroeffnen eben eröffnen. Durch dieses Drücken wird ein Event-Prozedur ausgelöst, in der die Daten im Remote Object Spieler gespeichert werden. Nach Warten auf die anderen Spieler kann das Spiel durch Drücken des Button Spiel_starten gestartet werden und die Methode paint der Klasse GUI_Spielfeld wird aufgerufen und das Spielfeld aufgebaut

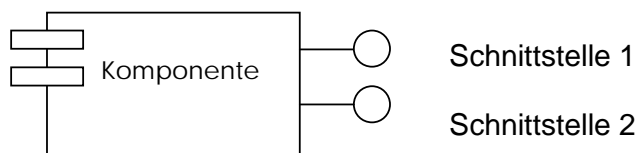
9. KOMPONENTEN- UND VERTEILUNGSDIAGRAMM

9.1. Komponentendiagramm

Das Komponentendiagramm zeigt die Abhängigkeiten unter den Softwarekomponenten, genauer die Abhängigkeiten zwischen Quellcode, Binärcodekomponenten und ausführbaren Programmen. Einige dieser Komponenten existieren nur während des Übersetzungsvorgangs, einige nur während des Linkens, andere zur Ausführungszeit und wieder andere die ganze Zeit über. Im Komponentendiagramm haben die Darstellungen nur Typencharakter, im Gegensatz zu den Deployment oder Verteilungsdiagramm, wo sie zu Instanzen werden (d. h. die Bezeichnungen werden unterstrichen).

Die Komponenten werden als drei ineinander verschachtelte Rechtecke gezeichnet; ihre Schnittstellen sind Striche mit Kreisen am Ende. Dadurch können die verschiedenen Schnittstellen der Komponenten dargestellt werden. Das Diagramm enthält ferner Abhängigkeiten in Form von gestrichelten Pfeilen.

Beispiel:

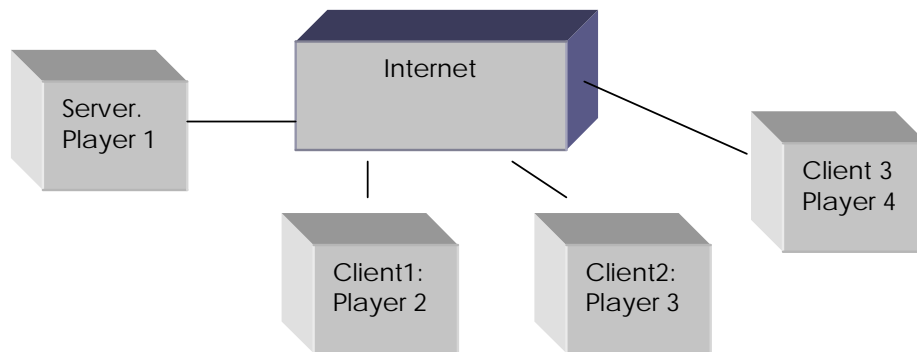


9.2. Verteilungs od. Deployment- Diagramm

Zur Darstellung der Hardware werden Verteilungs-Diagramme verwendet. Sie zeigen, welche Komponenten und Objekte auf welchen Knoten (die eine Verarbeitungs- oder Hardwareeinheit darstellen) laufen. Knoten werden als Quader gezeichnet. Unter den Knoten existieren Verbindungen: Dabei handelt es sich um die physikalischen Kommunikationspfade, die als Linien eingezeichnet werden

Häufig werden diese Diagramme mit normalen Zeichenprogrammen erstellt und für die Quader werden Clip-Arts verwendet.

Verteilungsdiagramm für Mensch-ärgere-dich-nicht Spiel



Literaturverzeichnis

[Oestereich'98]..... Bernd Oestereich, Objektorientierte Softwareentwicklung- Analyse und Design mit der Unified Modeling Language. 4. aktuell. Auflage. Oldenbourg Verlag 1998. ISBN 3-486-24787-5

[Fowler'97]..... Martin Fowler, Kendall Scott, UML konzentriert- Die neue Standard- Objektmodellierungssprache anwenden. 1. Auflage. Addison Wesley Longman Verlag GmbH 1998. ISBN 3-8273-1329-5

Anhang A: CRC- Karten

CRC- Karten (Class-Responsibility-Collaboration) wurden in den späten achtziger Jahren von Cunningham und Beck entwickelt (eine gute Einführung in CRC- Karten gibt die Website des Erfinders: <http://c2.com/doc/oopsla89/paper.html>)

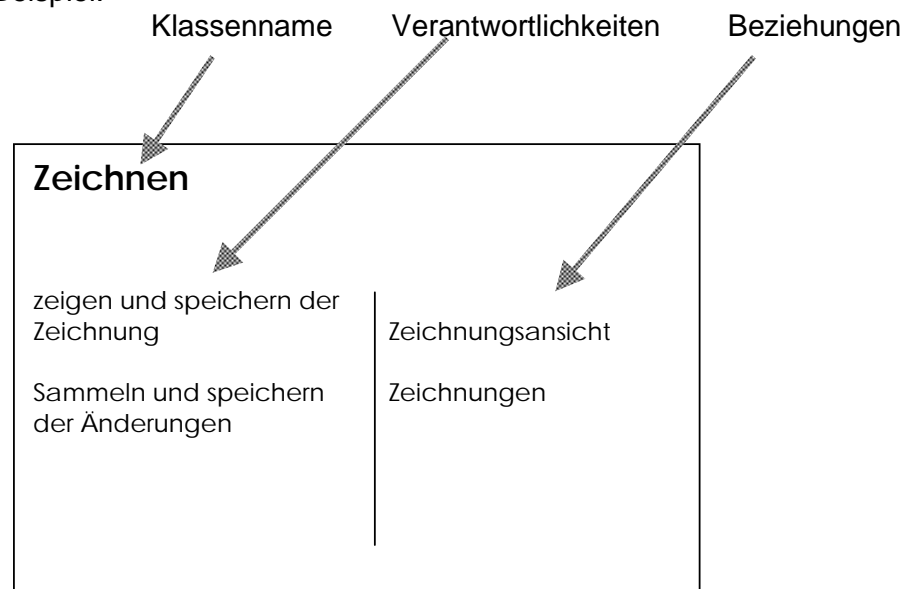
Anstatt Diagramme zur Modellentwicklung zu benutzen, verwendet man zu Präsentation der Klassen kleine Karteikarten.

Anstatt Attribute und Methoden schreibt man Verantwortlichkeiten auf. Was sind nun Verantwortlichkeiten: Es handelt sich um eine Beschreibung des Zwecks der Klasse (losgelöst von Bits und Daten). Wichtig: Es ist nicht erlaubt mehr aufzuschreiben, als auf die Karte paßt.

Als zweites schreibt man nun zu jeder Verantwortlichkeit die Zusammenhänge zu den anderen Klassen auf. So erhält man eine Verbindung zu den anderen Klassen.

Besonders von Vorteil sind CRC Karten beim durchgehen eines Anwendungsfalles: Man nimmt sich die Karten, wie die Klassen in dem Anwendungsfall gerade zusammenwirken. Wie sich die Verantwortlichkeiten dabei gerade bilden, kann man sich auf den Karten notieren.

Beispiel:



Wichtig ist, daß man die Karte nicht mit Verantwortlichkeiten „überlädt“. Man würde zu sehr in die Tiefe gehen, und verliert die Übersicht.

Anhang B: Java Klassen

GameServer.java

```
// Network game example, (C) Bernd Wender, 1998

import java.rmi.*;

public interface GameServer extends Remote {
    public GameServer connect() throws RemoteException;
    public void movePlayer(int dx, int dy) throws RemoteException;
    public Position getPosition() throws RemoteException;
}
```

GameServerImplementation.java

```
// Network game example, (C) Bernd Wender, 1998

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.security.*;
import java.util.*;
import java.net.*;

// This is the game server. An instance of this class is created for
// each player
public class GameServerImplementation extends UnicastRemoteObject
    implements GameServer {

    // the grid on which players are moving around (for future use)
    static int grid[][];

    // represents a player
    private Player player;

    // constructor
    public GameServerImplementation() throws RemoteException {
        player = new Player();
    }
}
```

```
// main method
public static void main(String args[]) {

    try {
        GameServerImplementation gameServer =
            new GameServerImplementation();
        gameServer.bindService("Game");
    }
    catch (Exception x) { x.printStackTrace(); }
}

// binds the "Game" service to the RMI registry
private void bindService(String service) {

    try {
        Registry registry = LocateRegistry.getRegistry();
        registry.bind(service, this);
    }
    catch (Exception e) { e.printStackTrace(); }
}

// creates new instance of itself (GameServerImplementation) and returns
// its remote interface to the client
public GameServer connect() throws RemoteException {
    return new GameServerImplementation();
}

// moves its player by [dx, dy]
public void movePlayer(int dx, int dy) throws RemoteException {
    player.move(dx, dy);
}

// gets the position of its player
public Position getPosition() throws RemoteException {
    Position p = player.getPosition();
    return(p);
}
}
```

Player.java

```
// Network game example, (C) Bernd Wender, 1998

import java.io.*;

// represents a player (at the server side)
public class Player implements Serializable{

    // the player's position
    Position position;
```

```
// constructor
public Player() {
    position = new Position();
}

// move the player by [dx, dy]
public void move(int dx, int dy) {
    position.moveX(dx);
    position.moveY(dy);
}

// get the player's position
public Position getPosition() {
    return position;
}

public String toString() {
    return new String("I am the player");
}
}
```

Position.java

```
// Network game example, (C) Bernd Wender, 1998
```

```
import java.io.*;

// represents a player (at the server side)
public class Player implements Serializable{

// the player's position
Position position;

// constructor
public Player() {
    position = new Position();
}

// move the player by [dx, dy]
public void move(int dx, int dy) {
    position.moveX(dx);
    position.moveY(dy);
}

// get the player's position
public Position getPosition() {
    return position;
}

public String toString() {
    return new String("I am the player");
}
}
```

GameClient.java

```
import java.rmi.*;

// This class represents a player on the client side.
public class GameClient {

    // this is the connection to the GameServer interface.
    private GameServer gameServer;

    // constructor: connects the GameClient to the GameServer
    GameClient(String url) {
        try {
            // look up the GameServer interface
            gameServer = (GameServer) Naming.lookup(url);

            // now connect your client. This is, let the server construct
            // a new game server thread for your client exclusively.
            gameServer = gameServer.connect();
        }
        // Oops, something went wrong.
        catch (Exception x) { x.printStackTrace(); }
    }

    // displays the position of a player.
    public void displayPlayer(Position p) {
        System.out.println("Position is " + p);
    }

    // create new client and make it connect to the server. Display
    // player's position, move the player and display its position again.
    public static void main(String args[]) {
        String machine, url;
        try { machine = new String(args[0]); }
        catch (Exception x) { machine = new String("esther"); }
        url = new String("//" + machine + "/Game");
        GameClient gameClient = new GameClient(url);

        try {
            gameClient.displayPlayer(gameClient.gameServer.getPosition());
            gameClient.gameServer.movePlayer(2, 3);
            gameClient.displayPlayer(gameClient.gameServer.getPosition());
        }
        catch (RemoteException rx) { rx.printStackTrace(); }}}
}
```