Deutsche Telekom
Fachhochschule Leipzig

Konzernintegrierter Kooperativer Bachelor-Studiengang

Telekommunikationsinformatik

Modul: I_STS

Softwaretechnik und Systemanalyse

Teilmodul: I_STS_SEN1

Software Engineering 1

Thema: I_STS_SEN1_02

Methoden

Autor: Dr.-Ing. Etienne N'Guessan

Redaktionsschluss: Juli 05

Erklärung des Autors

Dieser Studientext richtet sich an Studierende des Konzernintegrierten Kooperativen Bachelorstudienganges Telekommunikationsinformatik.
Der Inhalt folgt dem Curriculum der Fachhochschule Leipzig in der Fassung vom 04.03.2003.
Das Selbststudium wird von Hochschullehrern und autorisierten Tutoren unter Einsatz dieses Studientextes und in Verbindung mit Komponenten des Blended Learning (Lernplattform, Präsenzlehrveranstaltungen, Videokonferenz, Tutorien, e-Mail, usw.) betreut. Der vorliegende Studientext ist ausschließlich zur Nutzung in dieser Studienform entwickelt. Eine Verwendung anderer Art ist nicht beabsichtigt.
Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenzeichen usw. in diesem Studientext berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetze als frei zu betrachten wären und daher von jedermann benutzt werden dürften.
Der Inhalt des Studientextes wurde sorgfältig erarbeitet. Dennoch übernehmen Autor und technische Redaktion für die Korrektheit keine Haftung.

Einleitung

Seit 1965 wird in Fachkreisen von der Softwarekrise gesprochen. Selbst die objektorientierte Softwareentwicklung konnte die Kluft zwischen Bedarf und Angebot an guter Software nicht schließen. Nach wie vor ist die Softwareentwicklung ein hoch kreativer Prozess, der sich einer vollständigen Automatisierung entzieht. Der Softwareentwickler führt in jedem Software- Entwicklungsprozess einen Spagat zwischen kreativer Gestaltung und ingenieurtechnischer Entwicklung aus. Seine Aufgabe ist es, sowohl Neues zu schaffen als auch Regeln und Qualitätskriterien bei der Entwicklung der Software zu berücksichtigen.

Das Lehrgebiet Software Engineering macht Sie mit den Regeln und Gesetzmäßigkeiten der Softwareentwicklung vertraut. Es zeigt Ihnen den Weg, wie durch konzentrierte ingenieurtechnische Entwicklung Softwaresysteme erstellt werden. Sie werden auf Fehler hingewiesen, die zu einem Scheitern eines Softwareprojektes führen können. Im Lehrgebiet Software Engineering üben Sie sich in der kreativen Gestaltung von Softwaresystemen.

Die Lehrveranstaltung Software Engineering nimmt Bezug auf Ihre Vorkenntnisse aus den Fächern Hard- und Software-Architektur, Netze und Programmierung**,** Datenbanksysteme, Kommunikationsnetze und Verteilte Systeme. Absolventen des Studienganges Telekommunikationsinformatik entwickeln und verwalten/betreuen Verteilte Softwaresysteme**.** Deshalb ist das Fach Software Engineering ein wesentlicher Bestandteil des Studienganges und zeigt das Zusammenspiel der Fächer und des darin vermittelten Wissens ihres Studienganges**.** Das Fach Software Engineering gibt Ihnen das Handwerkszeug für Ihre spätere Tätigkeit.

In diesem Lehrbrief gelten alle grammatischen maskulinen Bezeichnungen für Personen weiblichen und männlichen Geschlechts.

Inhaltsverzeichnis

Literaturverzeichnis

Booch, B. 1986. „A Spiral Model of Software Development and Enhancement“. *Software Engineering Notes 11.4* (Mai 1986): 61-71

Burkhardt, R 1997. UML – *Unified Modeling Language*. Addison-Wesley

Coad, P. und Yourdon, E. 1991. *Object-Oriented Analysis 2*. Englewood Cliffs, New Jersey: Prentice Hall.

Coad, P. und Mayfield, M. 1997. *Java Design*. Upper Saddle River, New Jersey: Prentice Hall.

Doherty, D. 1997. *JavaBeans in 21 Days*, sams net

Flanagan, D. 1997 *Java in a Nutshell*, Edition 2, O'Reilly

Fowler, M. 1997. *UML konzentriert. Die neue Standard-Objektmodellierungssprache anwenden.* Addison-Wesley

Gamma, E. 1996: *Entwurfsmuster – Elemente wiederverwertbarer objektorientierter Software*, Addison-Wesley

Jacobsen, I. 1992. *Object-Oriented Software Engineering - A Use-Case Driven Approach.* Wokingham, England: Addison-Wesley.

Meyer, B. 1990. *Objektorientierte Softwareentwicklung.* Wien: Hanser.

Oestereich, B. 1998 *Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language,* Oldenbourg

Rumbaugh, J. 1991. *Object-Oriented Modelling and Design*, Englewood Cliffs, New Jersey: Prentice Hall.

Shlaer, S. und Mellor, S. 1988. *Object-oriented Systems Analysis - Modelling the World in Data.* Englewood Cliffs, New Jersey: Yourdon Press.

Vogel, A. 1997. *Java Programming with CORBA*, „Wiley Computer Publishing„

Quellenverzeichnis

http://www.omg.org
Objekt Management Group

http://www.oi.com/index.html
Object International - Education, Tools, Consulting

http://www.microgold.com/
MicroGold Software Inc
A great place to find UML OO CASE TOOLSUML CASE TOOLS

http://www.ulb.ac.be/esp/ip-Links/Java/joodcs/index.html"
Java OO Design & Coding Standards

www.ntu.edu/1/atmp/1997Courses/mc97081801.htm
OOA/OOD Using UML and Java

http://java.sun.com/nav/developer/index.html
Sun for Developers

http://www.developer.com/directories/pages/dir.java.javabean.general.html"
developer.com – Directories

http://www.parallax.co.uk/cetus/oo_uml.html
Cetus Links: 8560 Links on Object-Orientation / UML

http://www.non.com/books/Java_cc.html"
Java -- All Books

http://st-www.cs.uiuc.edu/users/patterns/patterns.html
Zu Pattern

http://st-www.cs.uiuc.edu/users(droberts/Refactory.html
Zu Redesign

1 Paradigmen

Die Welt ist ein Doppelkeks.
Christian Fatman Scherschel

Ein Paradigma vereinfacht die Welt, indem sie auf eine bestimmte Anschauung reduziert wird. In diesem Kapitel möchten wir uns verschiedene Paradigmen der Softwareentwicklung ansehen.

1.1 Das funktionsgetriebene Paradigma

Das funktionsgetriebene Paradigma konzentriert sich auf das Verhalten und die dynamischen Aspekte einer Lösung. Man spricht auch von funktionszentriert.

Vorgehensweise:

- Finden von Prozessen.
- Beschreiben von Daten, die von Prozessen manipuliert werden.
- Unterteilen der Prozesse in Unterprozesse.
- Spezifizieren der Unterprozesse.

Anwendung:

Geeignet für Anwendungen in denen die Prozesse im Vordergrund stehen z.B. Echtzeitanwendungen.

1.2 Das datengetriebene Paradigma

Hier stehen die Daten für die Lösung im Vordergrund. Man spricht auch von datenzentriert. Es interessieren die Daten und die Beziehungen zwischen den Daten.

Vorgehensweise:

- Beschreiben der Daten des Problems.
- Finden von Prozessen, die die Daten manipulieren.
- Unterteilen von Daten in untergeordnete Einheiten.
- Schreiben der Datendeklarationen.
- Schreiben der prozeduralen Bestandteile.

Anwendung:

Beschreiben von Datenbanken z.B. mit dem ER Modell.
Erstellen von Datenflussdiagrammen.

1.3 Das objektorientierte Paradigma

Das objektorientierte Paradigma konzentriert sich auf den Problembereich und berücksichtigt Funktionen und Daten. Daten und Funktionen werden in abgeschlossenen Einheiten organisiert. Diese Einheiten repräsentieren Dinge aus der realen Welt. Man spricht auch von „konzeptzentriert“.

Vorgehensweise:

- Finden von Dingen im Problembereich.
- Ausarbeiten von Beziehungen zwischen den gefundenen Dingen.
- Ausarbeiten von strukturellen und dynamischen Merkmalen der Dinge.
- Schreiben von Kode der die Dinge repräsentiert.

1.4 Vergleich

Das funktionsgetriebene Paradigma führt zu technologieorientierten Lösungen. Wiederverwendung mit Bibliotheken, die schwer abzugrenzen sind.

Das Objektorientiertes Paradigma arbeitet mit Dingen aus der realen Welt und kommt unserer Vorstellung am nächsten.

2 Objektorientierung

Die reale Welt ist sehr komplex. Mit der Objektorientierung können wir die komplexe Welt in Modellen und Programmen nachbilden.

2.1 Warum OO?

Die Sprache des Kunden

Eine konsequent am Bedürfnis des Kunden orientierte Softwareentwicklung hat sich mit dem Problem zu befassen, innerhalb kurzer Zeit mit Fachleuten aus den unterschiedlichsten Bereichen eine gemeinsame Sprache zu entwickeln, die es gestattet, deren Wünsche zu erkennen und zu dokumentieren.
Eine solche Sprache muss zum einen einfach gehalten sein, sich also möglichst an die intuitive Denkweise der Mehrheit der Menschen anpassen. Zum anderen muss sie aber gestatten, Information übersichtlich und dicht zu präsentieren, die Form weit schweifender Prosa wie in dieser Einleitung ist in jenem Zusammenhang nicht angebracht.

Ein gut ausgebildeter Entwickler war es bis in die frühen 80er Jahre gewohnt, eine Intuition dafür zu entwickeln, wie man die Probleme der Welt in die Sprache der strukturierten Softwareentwicklung übersetzen kann und aus umgangssprachlich abgefassten Problembeschreibungen Entwurfskonzepte für Software gewinnt.
Betrachtet man Softwareentwicklung nicht als „den großen Wurf“, sondern als einen zähen, iterativen Prozess, so fällt hierbei nicht nur der einmalige Übersetzungsaufwand störend ins Gewicht. Ständig sind die beteiligten Parteien gezwungen, sich ihre unterschiedlichen Denkweisen zu verdeutlichen und sicherzustellen, dass die notwendige Information auch wirklich ausgetauscht wurde.

Dieses Problem ist mit dem Ansatz der objektorientierten Programmierung ein ganzes Stück kleiner geworden. Es scheint bei Einführung objektorientierter Sprechweise sogar das Phänomen aufzutreten, dass einem von strukturierter Softwareentwicklung unbelasteten Kunden die Eingewöhnung leichter fallen kann, als einem „alten Hasen“ der strukturierten Programmierung. In jedem Fall scheint sich die Orientierung an Objekten weitaus eher mit unserem Alltagsverstand zu vertragen, als dies mit den Entwurfselementen der strukturierten Programmierung der Fall ist.

Fazit:

Der Einsatz von OO vergesellschaftlicht den Softwareentwurf.

2.2 Qualität und Kosten

Grundsätzliches Ziel des Software-Engineerings ist die Erstellung von Qualitätssoftware. Was sind die gängigsten Kriterien für Softwarequalität?

2.2.1 Qualitätskriterien

Erweiterbarkeit

Leichtigkeit, mit der Software an Spezifikationsänderungen angepasst werden kann.
Hilfsmittel:

- Einfachheit des Entwurfs
- Dezentralisierung

Wiederverwendbarkeit

Die Eigenschaft, ganz oder teilweise für neue Anwendungen wiederverwendet werden zu können.

Hilfsmittel:

- Dokumentation
- Entwicklung in Komponenten

Bereich der Wiederverwendung:

- im Projekt
- in der Abteilung
- im Unternehmen
- im Konzern
- weltweit

Kompatibilität

Leichtigkeit, mit der Software mit anderen Softwareprodukten verbunden werden kann.

Schlüssel:

- Einheitlicher Entwurf
- Vereinbarung von Standards

- **Portabilität**

Leichtigkeit, mit der Software auf verschiedene Hardware- und Software-umgebungen übertragen werden kann.

Unterscheidung in:

- Portabler Quellkode
- Portabler Bytekode

Lesbarkeit

Maß für die Leichtigkeit, mit der sich projektfremde Programmierer in vorhandene Softwarequellen einarbeiten können.

Hilfsmittel:

- Strukturierter Kode
- Selbsterklärende Variablennamen
- Dokumentation
- Module
- Gliederung

Robustheit

Die Fähigkeit von Software, auch unter Ausnahmebedingungen exakt zu funktionieren

Korrektheit

Fähigkeit von Software, ihre Aufgaben den Anforderungen und der Spezifikation gemäß exakt zu erfüllen

Effizienz

Ökonomische Nutzung der Hardware-Ressourcen

Verifizierbarkeit

Leichtigkeit, mit der Abnahmeprozeduren (Testdaten, Fehlererkennung, -verfolgung) während der Validierung und der Betriebsphase erzeugt werden können.

Sicherheit

Fähigkeit, verschiedene Komponenten gegen unberechtigte Zugriffe zu schützen

Benutzerfreundlichkeit

Leichtigkeit, mit der die Benutzung der Software erlernt werden kann. Auch Maß für die Klarheit und Effizienz in der Benutzung.

Fazit für OO:

Einen Nachteil im Rahmen dieser Qualitätskriterien kann man sich im Bereich der Effizienz der Software einhandeln. Diesen Preis bezahlt man aber im Normalfall gerne zu Gunsten von deutlich verbesserter Erweiterbarkeit, Wiederverwendbarkeit und Kompatibilität, denn in diesen Bereichen entfaltet der vergesellschaftlichte Entwurf der OO seine ganze schlichte Schönheit.

2.2.2 Wartungskosten für Software

Software muss nicht (oder kaum) gewartet werden. Hauptsächlich geht es bei diesem Begriff um die Verschleierung von sonstigen mehr (Änderungen) oder weniger vornehmen (Fehlerbehebung) Aktivitäten.

Die Hauptkosten der Softwarewartung entstehen in den Bereichen:

- Änderungen in den Benutzeranforderungen (ca. 40%)
- Änderungen in den Datenformaten (ca. 20%)

Fazit für OO:

Wiederverwendbarkeit und Erweiterbarkeit sind für die Folgekosten des Softwareeinsatzes die relevanten Kriterien und in gerade diesen Bereichen sind durch Einsatz von OO Verbesserungen zu erwarten.

2.2.3 Schlussfolgerung

Die Qualitätskriterien Verträglichkeit, Wiederverwendbarkeit und Erweiterbarkeit erfordern Entwicklungstechniken, die flexible, dezentralisierte Entwürfe produzieren, bestehend aus festen Modulen, verbunden durch wohldefinierte Schnittstellen. Diese Erfordernisse sind um so dringlicher, als der Großteil der Wartungskosten in genau diesen Bereichen entstehen. Die Relevanz der Wartungskosten dürfte spätestens im dramatisch steigenden Bedarf an Cobol-Programmierern im Rahmen der Jahr-2000-Problematik ihren Ausdruck gefunden haben: Software verschwindet nicht einfach nach ein paar Jahren, wie lange Zeit angenommen.

In einer zunehmend vernetzten Welt verschiedenster Plattformen werden darüber hinaus Verträglichkeit und Portabilität eine immer größere Rolle spielen. Wir werden uns hier hauptsächlich mit der Verbesserung der ersten drei Qualitätsfaktoren befassen. Die Portabilität bekommen wir mit der Programmiersprache Java geschenkt.

3 Begriffe der Objektorientierung

Nur Menschen, die mit Computern Erfahrung haben, finden Objektorientierung befremdlich. Für jemanden, der nicht programmiert, ist OO etwas ganz Normales.

Ein Mensch betrachtet die Welt als Menge von Objekte, die zueinander in (allerdings vielfach sehr kontextabhängigen) Beziehungen stehen und miteinander kommunizieren. Der OO-Softwareentwickler kann im Gegensatz zum Entwickler, der noch in strukturierter Weise denkt, diese Denkweise direkt übernehmen, muss allerdings hierbei den für sein Problem relevanten Kontext einer Weltsicht erkennen und isolieren.

OO-Programmierung besteht somit weniger im Ausüben einer neuen Programmiertechnik, als vielmehr im konzeptionellen Umsetzen einer eigentlich intuitiven Art zu denken und Problemanalyse zu betreiben.

3.1 Problem und Lösung

Ein Problem stellt den Ist-Zustand dar.

Die Lösung oder Solution ist der Ziel- oder Soll-Zustand den wir erreichen möchten.

? 1. Welche Anforderungen stellt das Problem an die Lösung?

? 2. Was ist die Lösung?

? 3. Wurden die Anforderungen des Problems gelöst?

3.2 Problembereich (Problem Domain) und Lösungsbereich

Der Problembereich oder die Problem Domain beinhaltet Dinge, die das Problem ausmachen. Zum Beispiel kann der Problembereich einer Reisebuchung folgende Dinge enthalten: Passagier, Flug, Hotel

? 4. Welche Dinge könnten wir im Problembereich einer Klinik finden?

Im Lösungsbereich oder in der Solution Domain modellieren wir technische Zusammenhänge. Einzelne Elemente können Programme, Prozesse, Netzwerke und Datenbanken sein.

3.3 Dragan & Alder - konkrete und abstrakte Dinge

Konkrete Dinge kann man in der realen Welt finden. Sie können einem Auto oder einem Motorrad begegnen. Wenn Sie sich ein Auto oder ein Motorrad vorstellen, können Sie sich ein konkretes Bild davon ausmalen.
Abstraktionen können wir zur Erklärung der Welt benutzen. Wir werden aber immer konkreten Dingen und keinen abstrakten begegnen. Oder sind Sie schon einmal einem Fahrzeug begegnet, das nicht konkret ausgebildet war?

? 5. Versuchen Sie sich einmal ein Fahrzeug vorzustellen, ohne an ein Auto, Motorrad oder Schiff zu denken.

? 6. Nennen Sie weitere Abstraktionen wie z. B. Geschäftsvorfall, Lebewesen etc.!

3.4 Strukturelle und Verhaltens-Merkmale

Jedem konkreten oder abstrakten Ding kann man bestimmte Merkmale zuordnen. Wir unterscheiden zwischen strukturellen Merkmalen und Merkmalen, die das Verhalten bestimmen.
Strukturelle Merkmale bestimmen einen Status, den ein Ding zu einem bestimmten Zeitpunkt besitzt z. B. eine Position oder Geschwindigkeit.
Merkmale, die das Verhalten bestimmen können Aktionen sein, die eine Sache ausführen kann z. B. beschleunigen, bremsen oder anhalten.

3.5 Abstraktion

Die Entwicklung der Softwaretechnologie ist mit einer fortschreitenden Abstraktion verbunden. Beim Abstrahieren reduzieren wir eine Sache aufs Wesentliche oder erstellen eine Repräsentation.

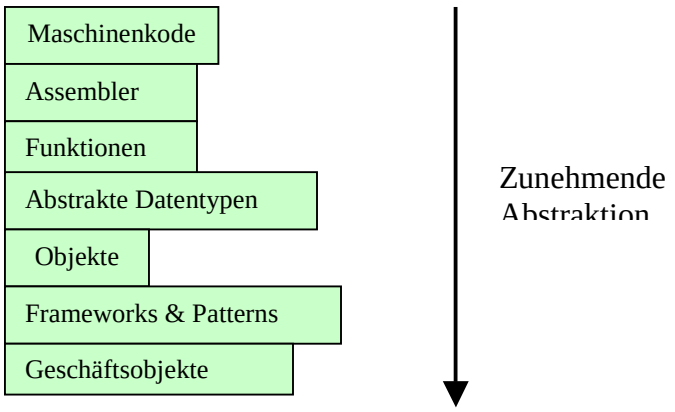


Abb. 3.1

? 7. Was ist abstrakte Kunst?

Abstraktion kann auf verschiedenen Ebenen stattfinden. Eine hohe Abstraktionsebene reduziert eine Sache aufs wesentliche, eine niedrige Abstraktionsebene enthält viele Details. Bei Gesprächen mit Leuten aus dem Fachbereich benutzt man eine hohe Abstraktionsebene während einen Programmierer das „Eingemachte“ interessiert.
Abstraktion ermöglicht Gemeinsamkeiten und Unterschiede zwischen Dingen zu erkennen. Mehrere Dinge, die Gemeinsamkeiten haben, können vielleicht in einem Ding zusammengefasst werden.
Auf Unwesentliches wird bei einem ausreichend hohem Abstraktionsniveau verzichtet. Letztendlich ermöglicht Abstraktion komplexe Zusammenhänge besser zu verstehen.

3.6 Hierarchie

Findet man mehr Komplexität vor, als in einem Erkenntnisschritt (einer Abstraktionsstufe) verständlich, so ordnet man mehrere Abstraktionsstufen an, unter anderem auch hierarchisch. Die zwei wichtigsten hierachischen Anordnungen sind Generalisierungsstrukturen („kind of“ - Spezialisierung und Generalisierung) und Aggregation („part of“ - Zusammensetzen einer hierarchisch höheren Klasse aus mehreren Teilklassen).

3.7 Kapselung

Mit Hilfe der Kapselung bekommen wir „Blackboxen“ in denen die Details verborgen sind. Kapselung unterstützt die Modularisierung. Wir können komplexe Zusammenhänge besser erfassen. Trotz Kapselung bleiben die Eigenschaften einer Sache erhalten.
Durch das Verstecken von Details unterstützt die Kapselung das Konzept der Abstraktion.

Teilung der Beschreibung einer Abstraktion in für andere Abstrakta relevante und irrelevante Bestandteile (vgl. Entscheidung ob Attribute überhaupt außenrelevante Bestandteile sein sollten - Assoziation: Die „wirkliche“ Welt ist uns in ihren Attributen nicht bekannt, nur in ihren Reaktionen auf Nachrichten).

Die äußeren Elemente können dabei noch nach ihrem Kenntnisstand des vorliegenden in Kapselungsstufen gruppiert werden.

Trennt den Benutzer eines Objekts von dem Autor.

3.8 Modularisierung

Die Modularisierung ist eine eher physikalische Gliederungsstruktur (hinsichtlich Kompilierung oder hinsichtlich Namensraum) unabhängiger, möglichst gekapselter und semantisch zusammengehöriger Programmteile. Hieraus ist eine Entwicklung von Softwaresystemen aus autonomen, in sich geschlossenen, in robusten Architekturen organisierten Teilstücken von Software möglich.

Einige Prinzipien modularer Entwicklung:

- Sprachliche Unterstützung für Modularisierung
- Wenige Schnittstellen (mit möglichst wenig anderen Moduln)
- Schmale Schnittstellen (möglichst wenig in der Schnittstelle)
- Explizite Schnittstellen (wenn, dann muss es aus dem Modultext hervorgehen)
- Geheimnisprinizp (jede Information so privat wie möglich z.B. per default modulintern)

3.9 Primitive Datentypen

Hier eine Aufzählung einiger primitiver Datentypen:

- Integer
- Float
- Double
- Character
- boolean

Fragen und Aufgaben:

? 8. Wie groß in Bytes ist ein Integer?

3.10 Abstrakter Datentyp

Menge von Datenstrukturen, die durch eine äußere Sicht beschrieben werden, nämlich durch die verfügbaren Dienste und die Eigenschaften dieser Dienste. Es geht nicht darum, was eine Datenstruktur „*ist*“, sondern was sie „*hat“*.

3.11 Klassen

In einer Klasse werden gleichartige Dinge oder Lebewesen zusammengefasst. Denken Sie an die Schulzeit zurück, dort wurden Schüler aufgrund Ihres Alters und Ihres erlernten Wissens in Klassen eingeteilt. Besitzt ein Schüler nicht mehr die Merkmale einer Klasse kann er eine Klasse überspringen oder im häufigeren Fall, eine Klasse wiederholen.
Im Tierreich kann man Lebewesen in Klassen, Arten und Gattungen einteilen. Dort entspricht eine Klasse einer Art Bauplan z. B. die Klasse der Säugetiere.
In der Objektorientierung ist eine Klasse ein Bauplan für gleichartige Objekte. Beim Backen von Weihnachtsplätzchen kann man eine Stechform benutzen mit der man viele gleiche Tannenbäume oder Sterne aussticht. Alle Plätzchen, die wir mit dem Tannenbaum ausstechen haben etwas gemeinsam, es sind Tannenbäume. Nach dem Ausstechen kann jeder Tannenbaum ein wenig individuell verändert werden. Auf den einen streuen wir Schokoladenstreusel auf einen anderen Mandeln oder Zucker. Wir können einen Tannenbaum auf

verschiedene Weise ausschmücken, trotzdem bleibt es ein Tannenbaum, da wir ihn mit der Stechform des Tannenbaums erzeugt haben.
In der Objekttechnologie entspricht der Tannenbaum einer Klasse und die ausgestochenen Plätzchen entsprechen einzelnen Objekten. Für Objekt wird auch das Wort Instanz benutzt, allerdings ist der Begriff der Instanz im deutschsprachigen Raum leider etwas anderes geprägt. Bernd Oestereich spricht in seinem Buch „Objekt orientierte Softwareentwicklung“ daher auch von einem Exemplar.
Wir können aus dem Bauplan der Klasse verschiedene Instanzen erzeugen. Man bezeichnet diesen Vorgang als Instanziierung. Nach der Instanziierung besteht zwischen einem Objekt und der zugehörigen Klasse eine instance_of Beziehung.

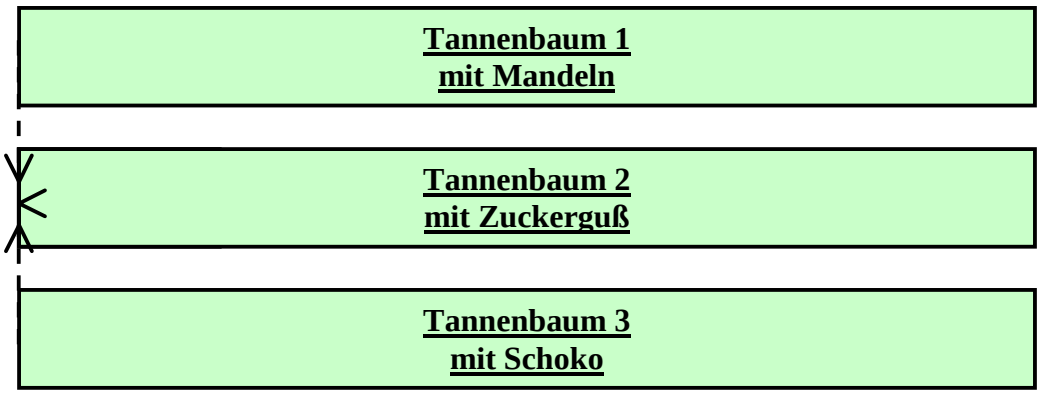


Abb.3.2: Klasse Tannenbaum mit Instanzen

3.12 Objekt

Ein Objekt ist ein konkretes Exemplar einer Klasse z.B. ein Weihnachtsplätzchen, das Sie essen können. Ein Objekt hat eine Bedeutung im Anwendungsbereich. Das Plätzchen, könnte die Rolle von Naschwerk während der Weihnachtsfeier spielen. Das Plätzchen stellt dem Benutzer ein Verhalten zur Verfügung: es lässt sich aufessen.

Jedes Objekt belegt Speicherplatz.

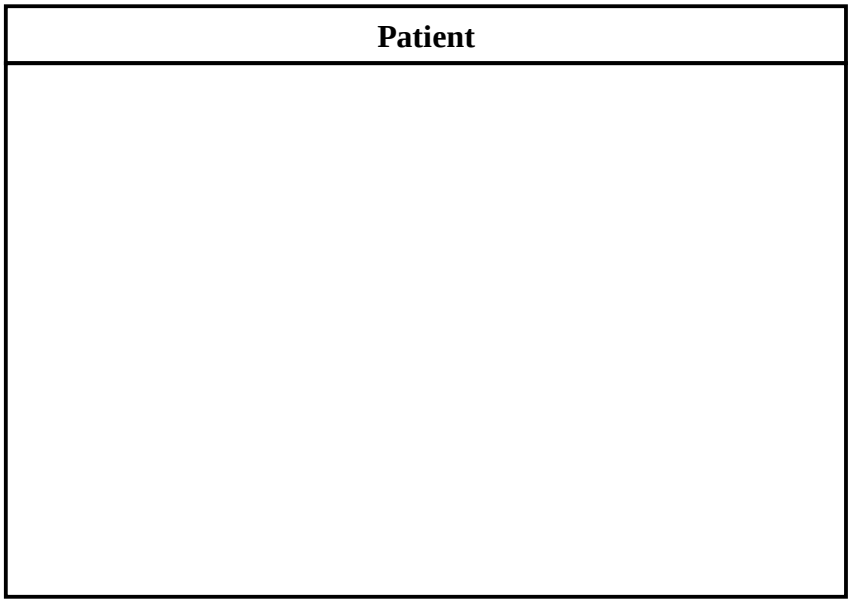
3.13 Eigenschaften

Eigenschaften beschreiben Objekte. Bei unserem Weihnachtsgebäck ist der Belag eines Plätzchens eine Eigenschaft. Der Belag kann aus Zucker, Mandeln oder Schokolade bestehen.

Anstatt Eigenschaft wird im OO Jargon der Begriff Attribut verwendet. Attribute beschreiben den Status eines Objekts zu einer bestimmten Zeit.
Das Auto von Heike hat am 18. August die Farbe rot, den Standort A6 zwischen Sinsheim und Walldorf, eine Geschwindigkeit von 10 Stundenkilometern und 23 Litern im Tank.

Übungsaufgaben

3.9 Welche Attribute kann ein Patient besitzen. Tragen Sie die Attribute ins Schaubild ein.



3.14 Operationen

Operationen oder Methoden sind einem Objekt zugeordnete Aktionen. Mit dem Auto von Heike können wir folgende Aktionen ausführen:

- Motor anlassen
- Motor abschalten
- bremsen
- beschleunigen
- tanken

Anstatt Methode wird oft auch der Begriff Nachricht verwendet. Die Methoden bestimmen das Verhalten eines Objekts.

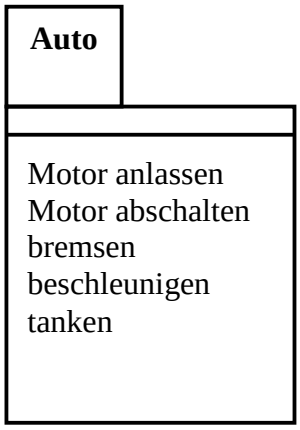


Abb. 3.3: Klasse Auto mit Operationen

3.15 Member

Einem Objekt sind Attribute und Methoden zugeordnet, man spricht auch von Mitgliedern eines Objekts. Die Attribute bestimmen den Status und die Methoden das Verhalten eines Objekts.
Die Kombination von statischen und dynamischen Aspekten in einer Einheit ist die Grundidee der Objektorientierung.

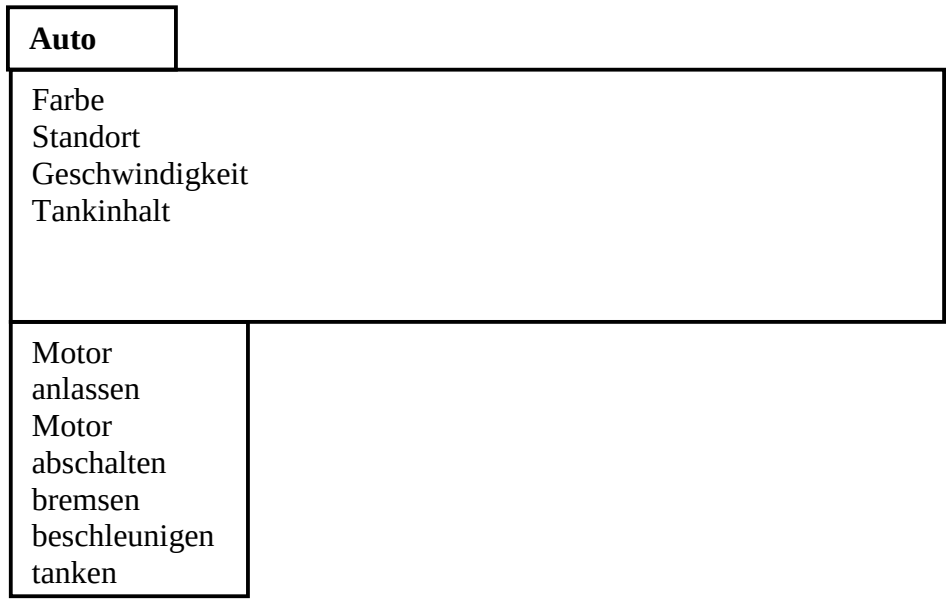


Abb. 3.4: Klasse Auto mit Attributen und Methoden

3.16 Zusicherungen

Zusicherungen sind Bedingungen, die wir an Attribute knüpfen können. Beispielsweise können wir festlegen, dass der Inhalt eines Autotanks zwischen 0 und 50 Litern liegt. Diese Zusicherung sorgt dafür, dass nie eine negative oder sehr große Anzahl von Litern sich im Tank befindet.

{ tankinhalt >= 0 und tankinhalt < 50 }

Übungsaufgaben

3.1 Welche strukturellen und Verhalts-Merkmale haben folgende Dinge?

ein Lebewesen
ein Fernseher
ein Kühlschrank
ein Flug
eine Rechnung

3.2 Finden Sie Gemeinsamkeiten in folgenden Dingen:

Hemd, Kunde, Monopoly, Kaufhaus, Angestellter, Koffer, Eis, Eisenbahn, Ameise, Verkäufer, Schuhgeschäft, Anzug, Krawatte, Lego, CD's, Weihnachtsmann, Pferd

3.3 Stellen Sie mit primitiven Datentypen einen Kundendatensatz dar.

3.4 Stellen Sie den Kundendatensatz als abstrakten Datentyp dar!

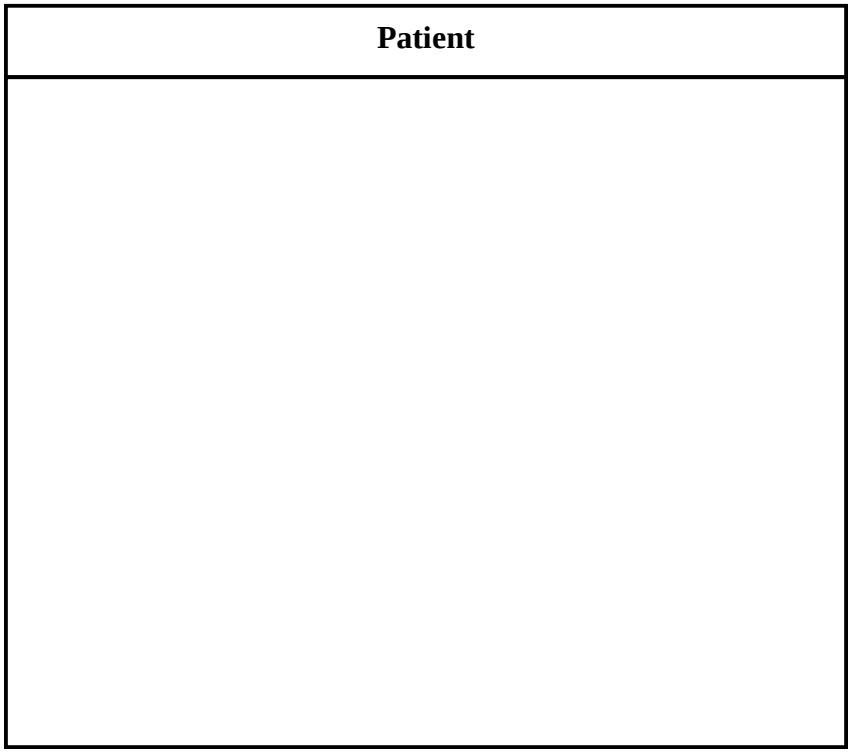
3.5 Finden Sie weitere Beispiele für abstrakte Datentypen!

3.6 Welche Attribute kann ein Patient besitzen. Tragen Sie die Attribute ins Schaubild ein.

Patient

Übungsaufgaben

3.7 Tragen Sie Operationen für den Patienten ins Schaubild ein. Auf die Darstellung der Attribute verzichten wir in dieser Abbildung.



3.8 Formulieren Sie Zusicherungen für folgende Attribute:

Auto: Geschwindigkeit, Anzahl Insassen, Anzahl Zylinder
Ferrari: Farbe
Buchhaltung: Kassenbestand

3.9 Erweitern Sie das Schaubild der Klasse Patienten um Zusicherungen.

4 Beziehungen zwischen Objekten

4.1 Vererbungsbeziehung

Mit Hilfe von Vererbung kann eine Klasse Eigenschaften d.h. Attribute und Methoden an eine abgeleitete Klasse weitergeben. In der Abbildung unten sehen wir einen ganzen Vererbungsbaum. Die Allgemeinste Klasse ist hier das Verkehrsmittel, von dem Flugzeug, Fahrzeug und Schiff abgeleitet sind. Von Fahrzeug sind wiederum Motorrad, LKW und PKW abgeleitet. Dieses Schaubild erinnert stark an einen Stammbaum, Klassen „erben“ Eigenschaften von Ihren Vorfahren.

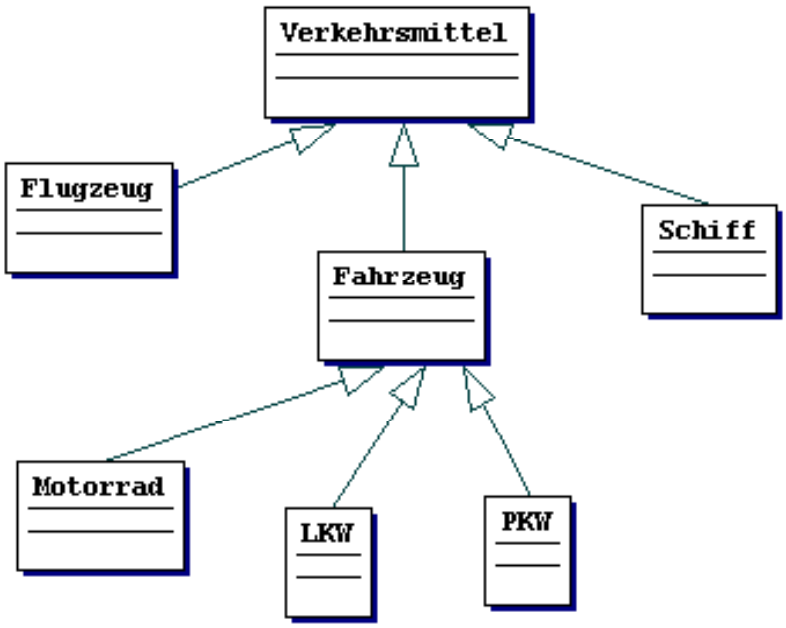


Abb.4.1: Vererbungshierarchie

In Abb. 4.1 zeigt eine abgeleitete Klasse auf Ihre „Basisklasse“. Die in der Basisklasse enthaltenen Eigenschaften werden an die Subklassen weitergegeben.
Die Vererbungsbeziehung wird als eine *ist_ein* Beziehung bezeichnet. Ein LKW ist ein Fahrzeug und ein Fahrzeug ist ein Verkehrsmittel.

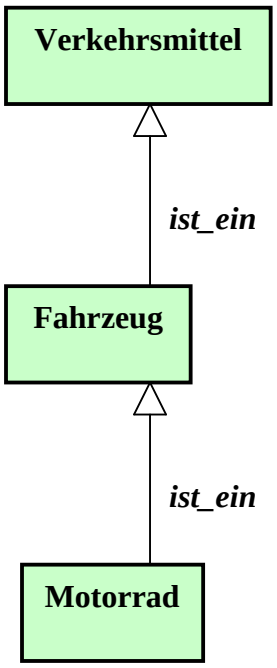


Abb 4.2: ***ist_ein*** **Beziehung**

Vererbung unterstützt die Wiederverwendbarkeit, indem bestehende Klassen erweitert werden können. Bei der Vererbung wird dem Konzept einer Klasse etwas hinzugefügt. Mit Vererbung werden keine Eigenschaften wieder entfernt. Sie besitzen durch Vererbung ja auch noch einen Blinddarm, obwohl der keinen Sinn macht.

4.1.1 Spezialisierung

Den Vorgang des Erbens bezeichnet man auch als Spezialisierung. Eine hierarchisch untergeordnete Klasse spezialisiert eine übergeordnete. Der umgekehrte Vorgang ist die Generalisierung.

4.1.2 Generalisierung

Eine Basisklasse wie z. B. Verkehrsmittel generalisiert Ihre Unterklassen Flugzeug, Fahrzeug und Schiff. Spezialisierung und Generalisierung sind vom Prinzip das Gleiche, nur in umgekehrter Richtung.

? 9. Welche Klasse enthält mehr Eigenschaften, die erbende oder die vererbende??

4.1.3 Mehrfachvererbung

Ein Mensch erbt seine Gene von Vater und Mutter. In der Objektorientierung gibt es die Mehrfachvererbung, die es zuläst, dass eine Klasse von mehreren Basisklassen erbt. Wir könnten ganz einfach ein Amphibienfahrzeug erzeugen indem wir von Fahrzeug und Schiff erben. Vom Schiff bekommt die Amphibie die Schiffsschraube und vom Fahrzeug die Räder.
Leider ist die Mehrfachvererbung eine Fehlerquelle und nicht ganz leicht anzuwenden. Einige objektorientierte Sprachen wie beispielsweise Objective C oder Java verzichten deshalb auf die Mehrfachvererbung.

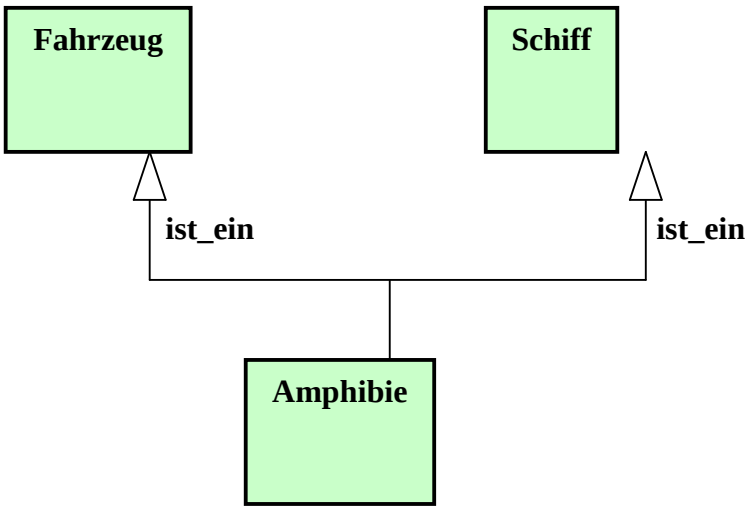


Abb. 4.3: Amphibie erbt von Flugzeug und Schiff

4.1.4 Zuordnung von Attributen und Methoden

In einer Klasse sollten nur die Attribute und Methoden platziert werden, die für diese Klasse typisch sind.

4.1.5 Überschreiben von Methoden

Mit unserem Kaufhausinformationssystem möchten wir Kunden und Angestellte auf dem Bildschirm anzeigen. Bei der Anzeige eines Kunden sollen folgende Daten dargestellt werden:

- Name und Vorname
- Umsatzvolumen

Bei der Anzeige eines Angestellten sollen folgende Daten angezeigt werden:

- Name und Vorname
- Gehalt

Wir könnten die Aufgabenstellung mit folgendermaßen realisieren:

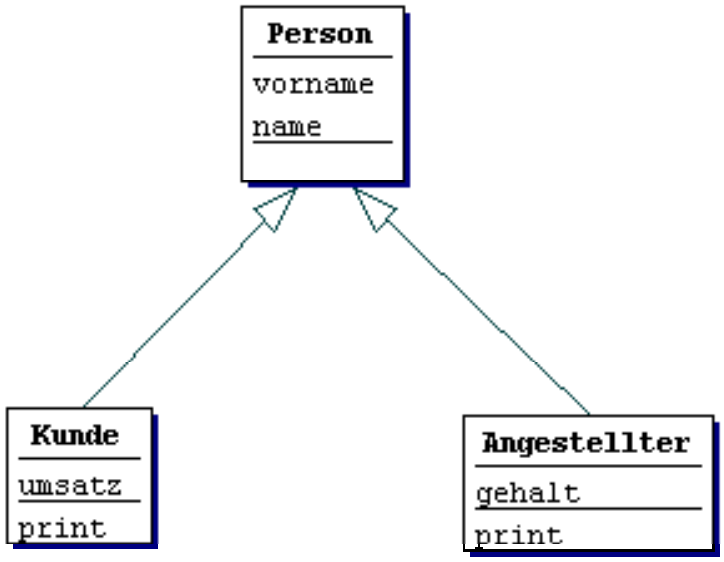


Abb. 4.4: print Methoden für Kunde und Angestellter

Diese Lösung würde funktionieren. Wir können einen konkreten Kunden anzeigen mit Umsatz, Name und Vorname. Vorname und Name sind durch Vererbung auch in der Klasse Kunde enthalten. Den Grundsatz, dass nur das in einer Klasse enthalten sein soll bzw. nur die Dinge gemacht werden sollen die für eine Klasse typisch sind ist hier verletzt. Wir geben in Kunde und Angestellter Name und Vorname aus, obwohl diese Eigenschaften zu Person gehören.
Die Technik des Überschreibens ermöglicht uns auch in Person eine print Methode einzubauen.

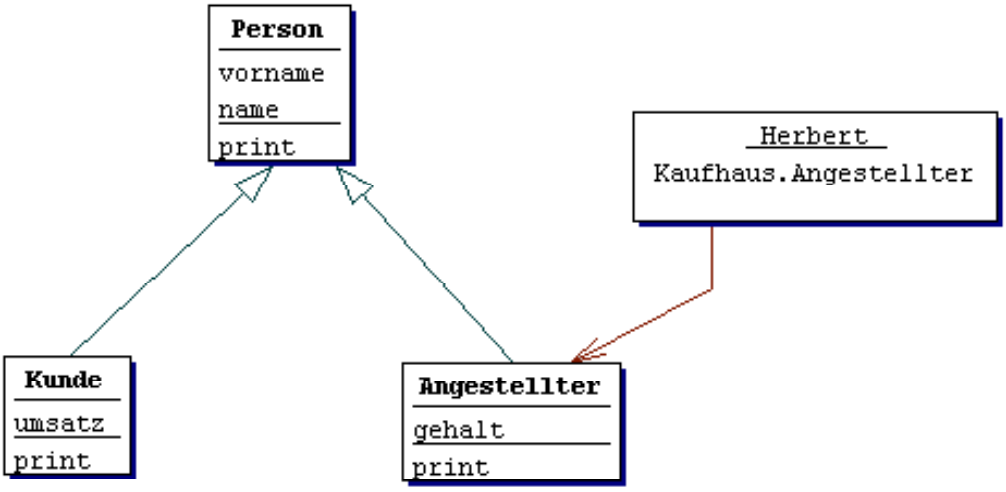


Abb. 4.5: print Methode in der Basisklasse

Die Abbildung zeigt die Vererbungshierarchie mit dem konkreten Angestellten Herbert. Wenn wir für Herbert die Methode print aufrufen, dann wird die Methode der spezialisiertesten Klasse also von Angestellter aufgerufen. Von dort aus können wir die print Methode der Basisklasse im Beispiel von Person aufrufen. Person gibt dann den Namen und Vornamen aus. Anschließend wird in die print Methode von Angestellter zurückgesprungen dort kann jetzt das Gehalt ausgeben werden.
Der Kodeauszug unten zeigt die Kodierung von Kunde und Angestellter in Pseudo-Java.

Kode: Person und Angestellter

```
class Person
{
        String name, vorname;

        print()
        {
        println(name, vorname);
        }
}

class Angestellter extends Person
{
        float gehalt;

        print()
        {
        super.print();
        println(gehalt);
        }
}
```

4.1.6 Abstrakte Klasse

Haben Sie schon mal ein Verkehrsmittel gesehen, das nur ein Verkehrsmittel ist. Es ist kein Boot, kein Auto, keine Straßenbahn, kein Fahrrad sondern nur ein Verkehrsmittel. Wir sehen in der realen Welt immer nur konkrete Ausprägungen. Ein reines Verkehrsmittel ist zu abstrakt. Trotzdem kennt unsere Sprache solche abstrakten Begriffe. Sie können die Anwendung verdeutlichen oder als Platzhalter für Konkreteres dienen. Sie können beispielsweise zum Seminar mit einem öffentlichen Verkehrsmittel anreisen. Dieser Satz enthält schon Logik ohne sich auf eine konkrete Ausprägung eines Verkehrsmittels festzulegen. Sollten Sie sich dazu entschließen ein öffentliches Verkehrsmittel zu benutzen, so müssen Sie sich für eine konkrete Ausprägung entscheiden.
Abstrakte Klassen dienen als gemeinsame Basisklasse für Klassen, die Gemeinsamkeiten besitzen.
Eine abstrakte Klasse stellen wir wie eine normale Klasse dar, schreiben aber den Namen der Klasse kursiv.

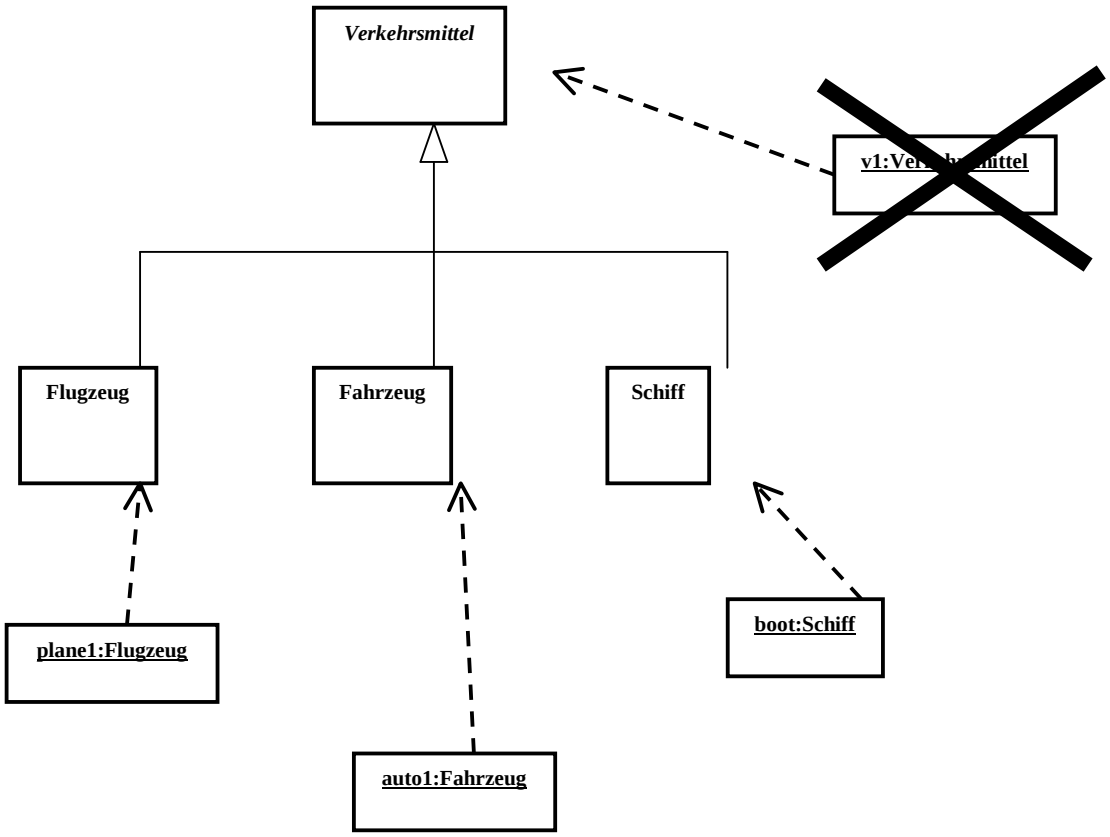


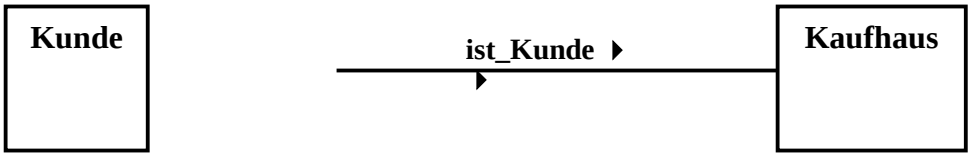
Abb. 4.6: Abstrakte und konkrete Klassen.

4.2 Assoziation

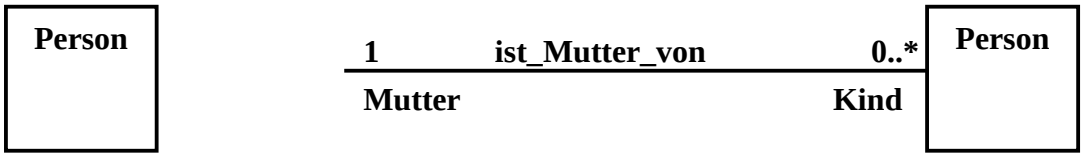
Dinge in der realen Welt können in einer Beziehung stehen. Wir listen hier mal einige Beziehungen auf:

- Ein Angestellter arbeitet in einer Abteilung.
- Ein Auto ist von einem Hersteller.
- Eine Person ist Schwester einer weiteren Person. Die gleiche Person ist auch Mutter von drei weiteren Personen und Frau einer Person.

Auch in der Objektorientierung gibt es Beziehungen zwischen Objekten. Betrachten wir die Objekte im Kaufhaus. Ein Objekt der Klasse Kunde kann bei einem Objekt der Klasse Kaufhaus Kunde sein.



Ein Objekt kann auch die gleiche Rolle bei mehreren Objekten Spielen. Betrachten Sie die „Mutter“ Rolle im Beispiel.



4.3 Aggregation

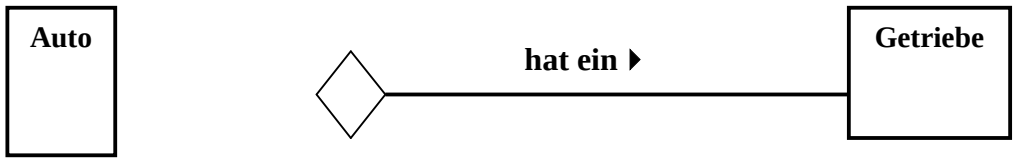
Bei der Aggregation werden mehrere einzelne Objekte zu einem Ganzen zusammengesetzt.

Beispiele:

Ein Auto besteht aus Motor, Getriebe, Batterie, Achsen, Lenkrad

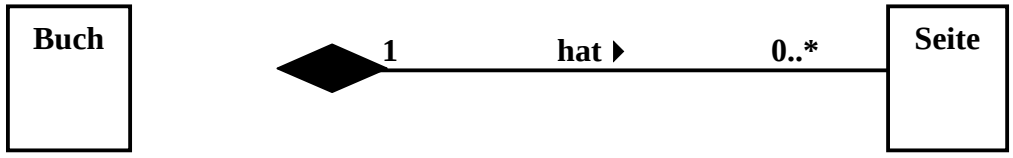
PC besteht aus CPU, Festplatte, Floppy, Harddisk

Man spricht bei der Aggregation auch von hat- oder ist Teil von- Beziehung. Man kann sogar ganze Objektbäume damit aufbauen.
Die Elemente einer Stückliste sind Teile eines Ganzen. Einzelne Elemente der Stückliste können wiederum aus weiteren Teilen aufgebaut sein. Wesentlich für die Aggregation ist
Die Aggregation wird wie die Assoziation mit einer unausgefüllten Raute auf der Seite der Aggregierenden Klasse dargestellt.



4.4 Komposition

In einer Komposition sind die Teile vom Ganzen abhängig. Wenn Sie ein Auto verschrotten, kann der Motor und das Getriebe ausgebaut und in anderen Autos wieder verwendet werden.
Bei der Komposition ist dies nicht möglich. Ein Buch ist aus einzelnen Seiten zusammengesetzt, wenn Sie das Buch verbrennen, verbrennen die einzelnen Seiten mit.
Die Komposition wird wie die Aggregation mit ausgefüllter Raute dargestellt.



4.5 Klassenkarten

Klassenkarten oder CRC-Cards für Class-Responsibilities-Collaboration sind ein beliebtes Hilfsmittel in der Analyse. Auf einer Registerkarte werden Klassenname, Verantwortlichkeiten und Beteiligte notiert. Jede Klasse separat auf einer Karte.

Übungsaufgaben

4.1 Finden Sie Spezialisierungen für Kunde, Spielzeug, Lebensmittel!

4.2 Finden Sie eine gemeinsame Basisklasse für folgende Begriffe!

1. Schauspieler, Regisseur, Kameramann, Kabelträger
2. Hund, Katze, Pferd
3. Dialog, Fenster, Listbox, Knopf
4. Meeting, Seminar, Weihnachtsfeier

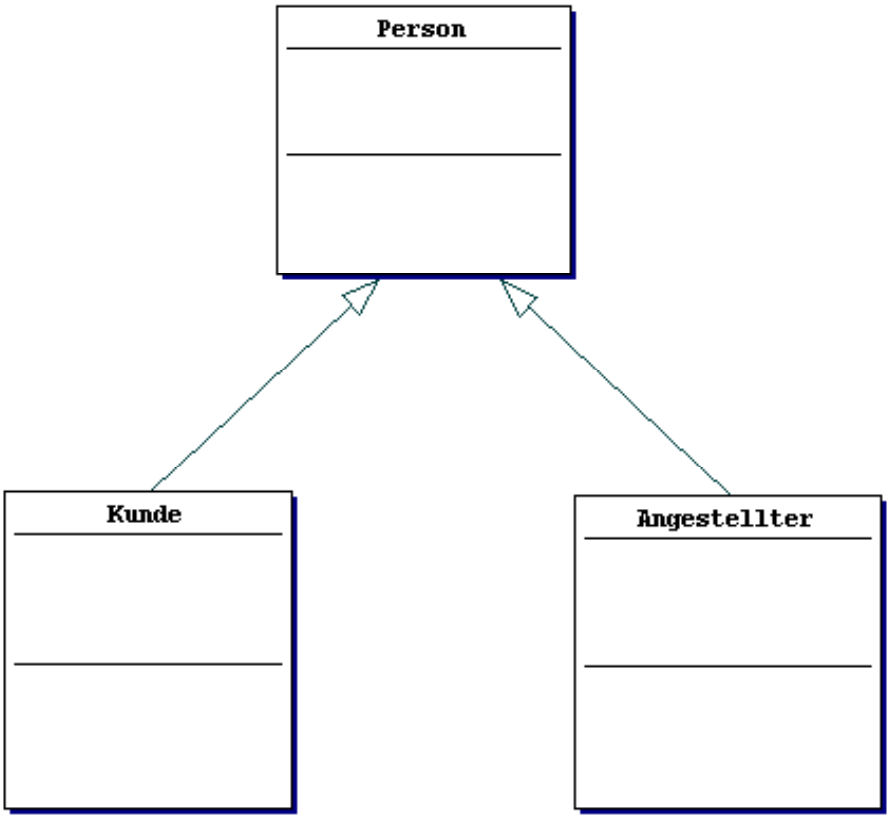
4.3 a) Tragen Sie folgende Attribute ins Klassendiagramm ein:

Name, Vorname, Gehalt, Umsatz

b) Tragen Sie folgende Methoden ins Klassendiagramm ein:

kaufen(), feuern(), holeNamen()

c) Formulieren Sie Zusicherungen für Gehalt und Umsatz!



4.4 Erstellen Sie die Klasse *Kunde* in Pseudo-Java.

4.5 Finden Sie weitere Beziehungen zwischen realen Objekten!

5 Polymorphismus

Bei der Polymorphie handelt es sich nicht um eine Korallenart sondern um einen Mechanismus, der die Objekt Technologie so mächtig macht. Polymorphie heißt auf griechisch nichts anderes als Vielgestaltigkeit. Eine Methode *beschleunigen* kann je nach Klasse ganz andere Auswirkungen haben. Betrachten Sie das Klassendiagramm in Abb. 5.1. Zwischen dem Auto, dem Motorrad, dem LKW und der abstrakten Klasse Fahrzeug und bestehen ist_ein Beziehungen. Ein Auto ist ein Fahrzeug. Ein Motorrad ist ebenfalls ein Fahrzeug. Wesentlich für ein Fahrzeug ist die Methode beschleunigen. Die Methode beschleunigen wird in Fahrzeug nur deklariert, d.h. es wurde kein Programmkode für die Methode geschrieben. Die konkreten Klassen Auto, Motorrad und LKW implementieren die Methode beschleunigen ganz unterschiedlich.

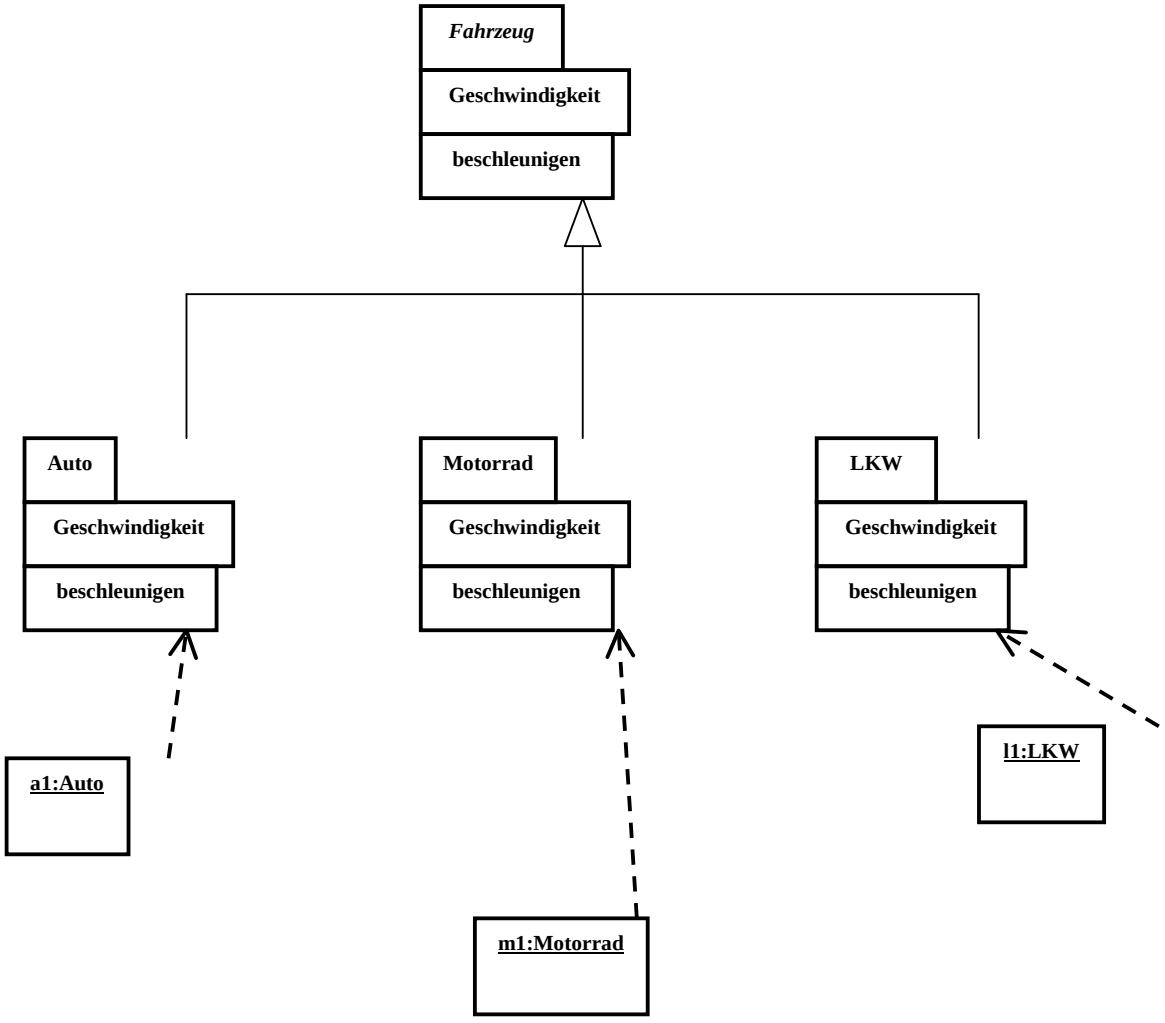


Abb. 5.1: Klassendiagramm

Implementierung von beschleunigen:

```
class Auto
{
        beschleunigen()
        {
        geschwindigkeit = geschwindigkeit + 5;
        }
}

class LKW
{
        beschleunigen()
        {
        geschwindigkeit = geschwindigkeit + 2;
        }
}

class Motorrad
{
        beschleunigen()
        {
        geschwindigkeit = geschwindigkeit + 15;
        }
}
```

Wir möchten einmal durchspielen, was an einer Ampel passiert. Zunächst erzeugen wir drei Fahrzeuge:

```
Fahrzeug f[0] = new Auto();
Fahrzeug f[1] = new Motorrad();
Fahrzeug f[2] = new LKW();
```

Die Ampel schaltet auf grün und wir beschleunigen unsere Verkehrsteilnehmer:

```
for(int i = 0; i < 3; i++)
        f[i].beschleunigen();
```

Für jedes Fahrzeug wird beschleunigen aufgerufen. Die Laufzeitumgebung in der die Anwendung läuft erkennt um welches Fahrzeug es sich handelt und ruft die passende Methode auf.
Die Zuordnung eines Methodenaufrufs zu einer Speicherstelle, an der sich die Implementierung einer Methode befindet nennt man Binden. Erfolgt die Bindung erst zur Laufzeit spricht man von dynamischem oder späten Binden.

?

10. Warum kann der Kompiler hier nicht schon den Aufruf mit der Methode verbinden?

?

11. Wenn Sie von Auto die Klasse Sportwagen ableiten und dort die Methode beschleunigen überschreiben, welche Methode wird für Objekte der Klasse Sportwagen ausgeführt?

?

12. Es wird ein Sportwagen erzeugt und einer Referenz vom Typ Auto zugewiesen. Anschließend wird die Methode beschleunigen aufgerufen. Wird die Methode von Sportwagen ausgeführt oder die in Auto?

```
Auto boxter = new Sportwagen()
boxter.beschleunigen()
```

Sieben Stufen zur objektbasierten Glückseligkeit (Meyer 1990)

1. *Objektbasierte modulare Struktur:* Systeme werden auf der Grundlage ihrer Datenstrukturen modularisiert.
2. *Datenabstraktion:* Objekte müssen als Implementierung abstrakter Datentypen beschrieben werden.
3. *Automatische Speicherplatzverwaltung:* Unbenutzte Objekte sollten ohne Programmierereingriff vom unterliegenden Sprachsystem freigegeben werden.
4. *Klassen:* Jeder nichteinfache Typ ist ein Sprachmodul, und jeder Modul höherer Ebene ein Typ. Ein Sprachkonstrukt, das die Aspekte Modul und Typ vereinigt, heißt Klasse.
5. *Vererbung:* Eine Klasse kann als Einschränkung oder als Erweiterung einer anderen definiert werden.
6. *Polymorphismus und dynamisches Binden:* Programm-Elemente dürfen sich auf Objekte aus mehr als einer Klasse beziehen, und Operationen dürfen unterschiedliche Realisierungen in unterschiedlichen Klassen haben.
7. *Mehrfaches und wiederholtes Erben:* Man kann Klassen deklarieren, die Erben von mehr als einer Klasse sind und mehr als einmal von einer Klasse erben (in Java nicht möglich).

Übungsaufgaben

5.1 Ordnen Sie die Begriffe:

Name, Vorname, Abteilungsleiter, Gehalt, Kunde, Umsatz, Person, Angestellter

in eine Klassenhierarchie mit Attributen.

a) Welche Klasse besitzt welches Attribut?
b) Welche Klasse erbt von welcher?

6 UML - Sprache und Prozess

Warum Design?

Die Verwendung einer OO-Programmiersprache reicht allein nicht aus um wiederverwendbare Software zu entwickeln. Was hinzukommen muss, ist ein weitergehendes Verständnis des OO-Konzepts, sowie eine Modellierungssprache, die es gestattet, auf dieser Basis zwischen Entwicklern und Kunden zu kommunizieren. Im Rahmen dieses Kurses wird die Unified Modelling Language (UML) vorgestellt, die verschiedene Ansätze des OO-Softwareentwurfs in sich vereint.

Eine Modellierungsmethode besteht aus

- Modellierungssprache und
- Prozessbeschreibung.

UML ist nur eine Modellierungssprache, sie macht keine Angaben darüber, wie man sie einzusetzen hat. Die drei Amigos sind gerade dabei, einen einheitlichen Prozess zu definieren, den sie *Rational Objectory Process* nennen werden.

Natürlich wird der im Einzelfall verwendete Prozess in hohem Maße von Faktoren wie Art oder Umfang der Software , die entwickelt wird, abhängen (Echtzeitsystem, Arbeitsplatzsystem, einzelner Entwickler, große Entwicklungsteams). Einen Prozess, der flexibel genug ist, derartige Bandbreiten abzudecken, nennt man auch Prozessrahmenwerk.

Hier soll erst einmal die UML als Sprache vorgestellt werden, bevor auf ihre Verwendung im Entwicklungsprozess eingegangen wird.

6.1 Die UML

Die Unified Modeling Language kann benutzt werden, um Softwareprojekte zu spezifizieren, darzustellen und dokumentieren. Zur Erstellung der UML haben sich die drei führenden Köpfe der objektorientierten Modellierung Grady Booch, Ivar Jacobson und Jim Rumbaugh zusammengetan und ihre Techniken Booch, OMT und OOSE verschmolzen. Man spricht deswegen auch von den drei Amigos.
Es gibt viele verschiedene Modellierungssprachen wie OMT, OOSE, Booch, OOAD etc. die sich oft nur in Details unterscheiden, in Kern aber das selbe ausdrücken. Damit Fachleute ihre Entwürfe vergleichen und diskutieren können ist eine gemeinsame Sprache notwendig. Die UML ist gedacht für diese einheitliche Sprache, sie vereint verschiedene Techniken die sich bei der Modellierung großer Systeme bewährt haben.

Die UML ist nicht auf eine spezielle objektorientierte Sprache zugeschnitten. Sie enthält aber einige Konstrukte und Darstellungen, die nicht in jeder Sprache zu finden sind. In JAVA gibt es beispielsweise keine Templates wie in C++ (zumindest noch nicht in JAVA 1.2). Die folgende kurze Abhandlung der UML verzichtet bewusst auf alle Elemente, die für JAVA nicht benötigt werden.

Chronologie:

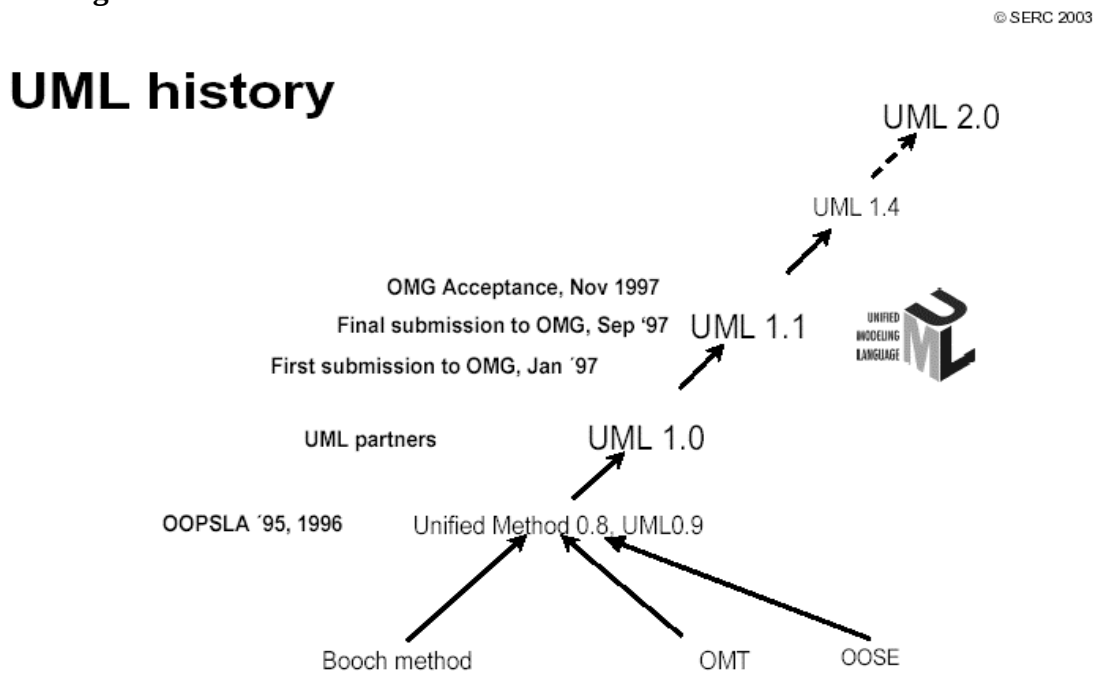


Abb. 6.1: Chronologie

6.2 Use-Cases

Ein guter Start für die objektorientierte Modellierung mit der UML sind Use-Cases. Mit dem Use-Case Diagramm können Anwendungsfälle modelliert werden. Das Use-Case Diagramm wird zusammen mit dem Anwender entwickelt.
Im Diagramm können Abhängigkeiten zwischen Anwendungsfällen und die interaktion mit Aktoren dargestellt werden.
Use-Cases können mit fachlichen Skizzen und Beschreibungen kombiniert werden. Ausgearbeitet werden Use-Cases gemeinsam mit dem Anwender aus dem Fachbereich.

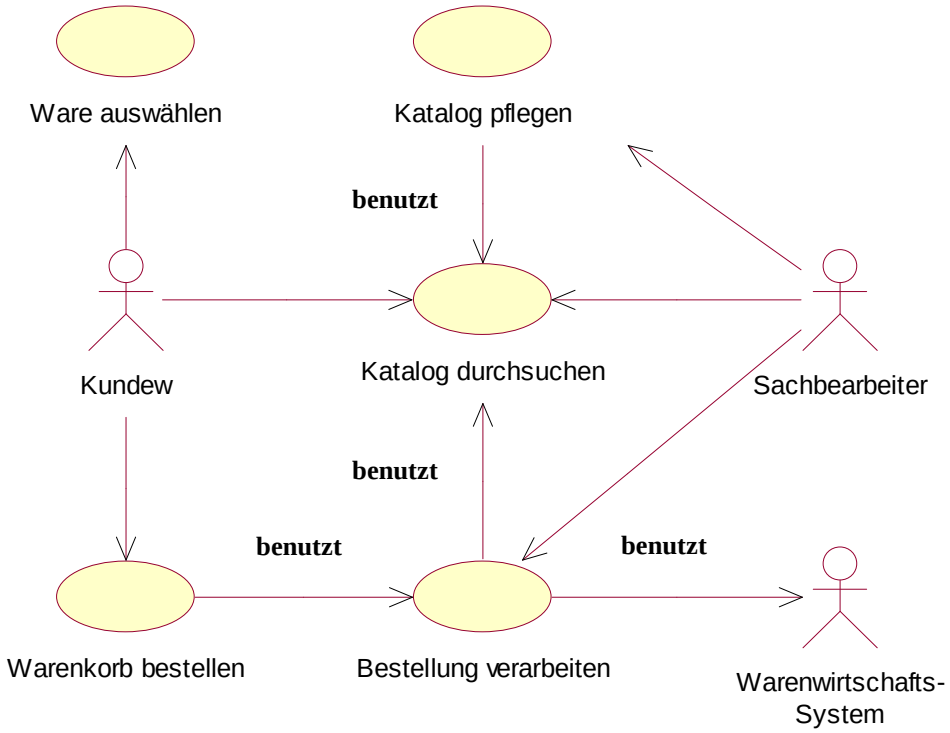


Abb. 6.2: Use-Case Online Kaufhaus

Die Abbildung 6.2 zeigt die Modellierung eines Use-Case Online Kaufhaus. Der Kunde ist ein Aktor, der sich der drei Use-Cases bedienen kann.
Die Aktoren eines Use-Case Diagramms können im Klassendiagramm als Klasse modelliert werden.

Notation:

- Ein Use-Case Diagramm besitzt einen Namen.
- Es können mehrere Use-Cases und Aktoren im Diagramm dargestellt werden.
- Zwischen Aktoren und Use-Cases kann durch einen Pfeil eine Interaktion abgebildet werden.
- Ein Use-Case kann einen weiteren benutzen oder erweitern. Diese Beziehungen werden ebenfalls durch einen Pfeil dargestellt.

6.3 Klassendiagramme

Klassendiagramme sind ein gutes Mittel statische Zusammenhänge zwischen Klassen zu modellieren. Ein besseres Wort für Klassendiagramme wäre „Statische Struktur Diagramme“, da im Klassendiagramm auch Interfaces, Beziehungen und sogar Objekte modelliert werden können.

Betrachten wir zunächst, wie eine Klasse im Klassendiagramm dargestellt wird.

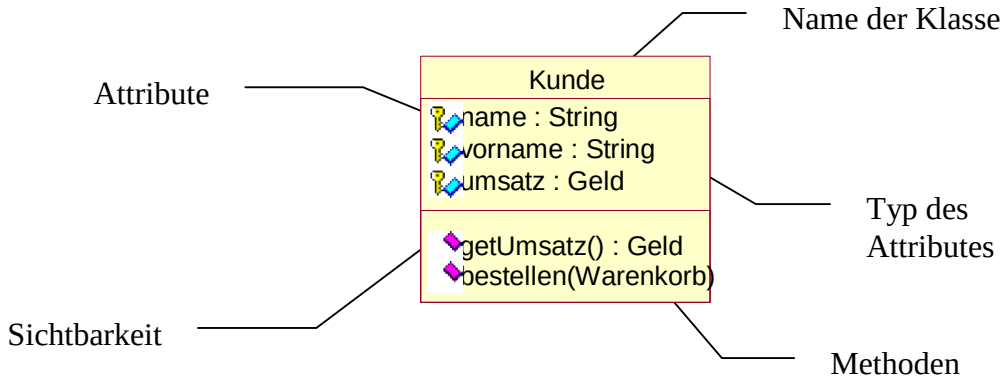


Abb. 6.3: Darstellung einer Klasse

Das Beispiel zeigt das Klassendiagramm für unser Online-Bestellsystem.

Der Kunde aus dem Use-Cases wurde ins Klassendiagramm übernommen. Auf die Darstellung des Sachbearbeiters wurde verzichtet, da er für den Problembereich nicht relevant ist.

Beziehungen zwischen den Klassen werden durch verbindende Linien dargestellt. Eine Vererbungsbeziehung haben wir zwischen einem Online-Kunde und Kunde. Online-Kunde erweitert Kunde um eine Mailadresse und die Operation sendMail(). Vererbung wird durch einen Pfeil mit leerer Spitze von der Sub- zur Superklasse dargestellt.

Außer der Vererbung finden wir im Diagramm weitere Beziehungen z.B. kann einem Kunde ein Warenkorb zugeordnet werden. Wie viele Warenkörbe ein Kunde besitzen kann und wie vielen Kunden ein Warenkorb zugeteilt wird, zeigen die Kardinalitäten. Auf der Line zwischen zwei Klassen können wir die Kardinalität angeben.

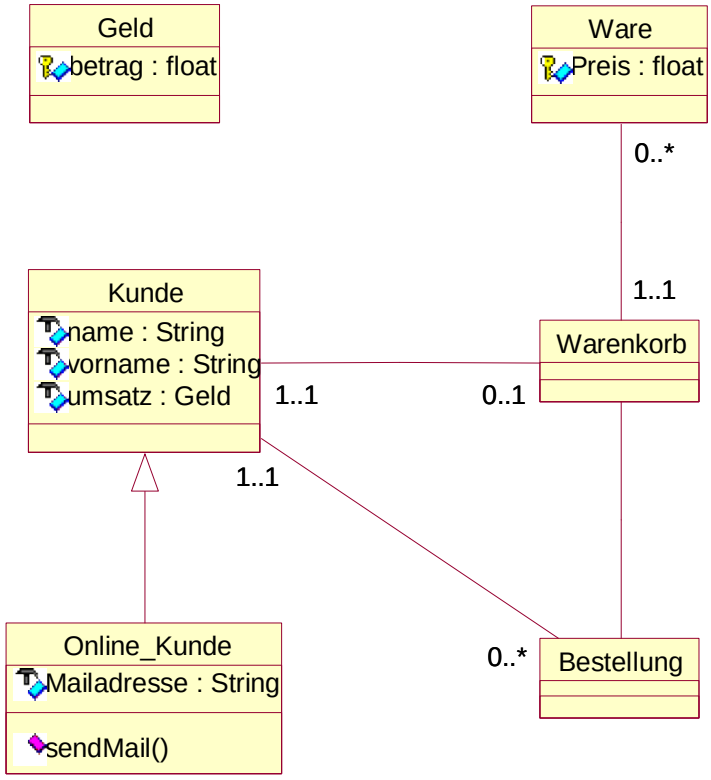


Abb. 6.4: Klassendiagramm Online Bestellsystem

Kardinalität in der UML:

Darstellung	Bedeutung
0..*	beliebig viele
1	genau eins
1..*	beliebig viele aber mindestens eins
2..4	zwei bis vier

6.3.1 Aggregation und Assoziation

Assoziationen werden im Klassendiagramm mit einer einfachen Linie dargestellt. Mit einer leeren Route können Aggregationen abgebildet werden. Bei Aggregationen spricht man auch von Teil-Ganzes-Beziehungen. Die Seite mit der Route ist in der UML die Klasse, die die andere vereinnahmt.

In JAVA werden alle Variablen außer primitive Datentypen als Referenzen realisiert. Daher kann in JAVA rein von der Sprache nicht zwischen Aggregation und Assoziation unterschieden werden. Übernimmt ein Objekt in JAVA die Verantwortung über ein anderes so kann man auch hier von einer Aggregation sprechen.

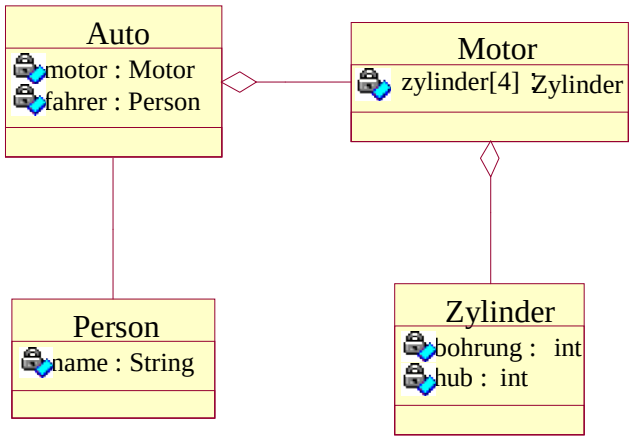


Abbildung 6.5: Aggregation und Assoziation bei Auto und Motor

6.3.1.1 Wie findet man Beziehungen zwischen Klassen?

Betrachten Sie das Sequenzdiagramm(Abb. 6.6) Wenn zwei Klassen miteinander „reden“ muss zumindest ein Objekt das andere kennen. Im Beispiel ruft der Warenkorb die Operation preisErmitteln() von Ware auf.

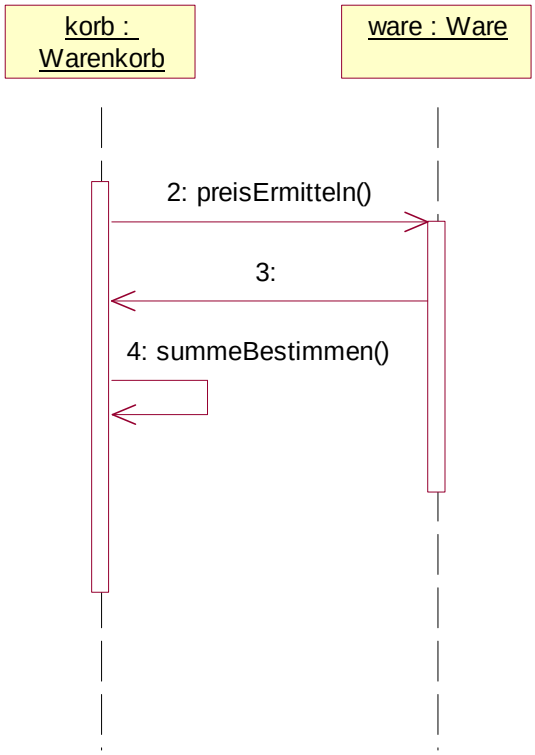


Abb. 6.6: Kommunikation zwischen Warenkorb und Ware

6.3.1.2 Navigation

Ohne weitere Angaben sind die Beziehungen Aggregation und Assoziation immer bidirektional d.h. man kann von jedem der beiden Objekte zum anderen gelangen. Die Navigation kann durch Pfeilspitzen eingeschränkt werden.

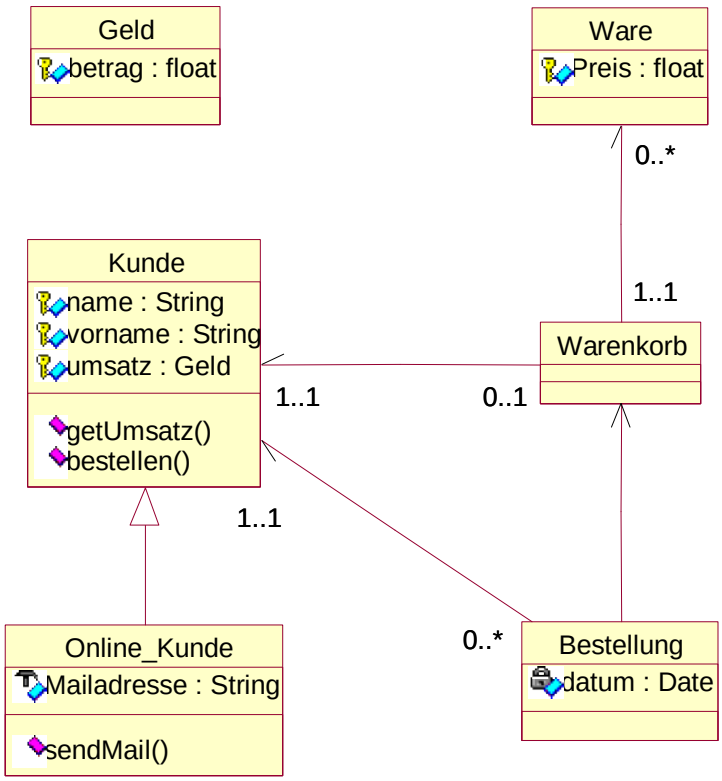


Abb. 6.7: Navigation im Klassendiagramm

6.3.2 Notation

- Eine Klasse wird durch ein Rechteck dargestellt. Dieses Rechteck wird durch zwei horizontale Linien in drei Bereiche aufgeteilt.
- Im obersten Bereich des Klassensymbols wird der Name der Klasse vermerkt.
- In die Mitte kommen Attribute und in den untersten Bereich Operationen.
- Beziehungen zwischen den Klassen können durch verbindende Linien ausgedrückt werden.
- Eine einfache Linie stellt eine Assoziation dar. An den beiden Ende der Linie können Kardinalitäten vermerkt werden.
- Vererbung wird durch einen Pfeil von der erbenden zur beerbten Klasse dargestellt. Die Pfeilspitze wird hohl gezeichnet.
- Die einzelnen Klassen können zusätzlich mit beschreibenden Texten versehen werden.

6.3.3 Analyse- und Design-Phase im Klassendiagramm

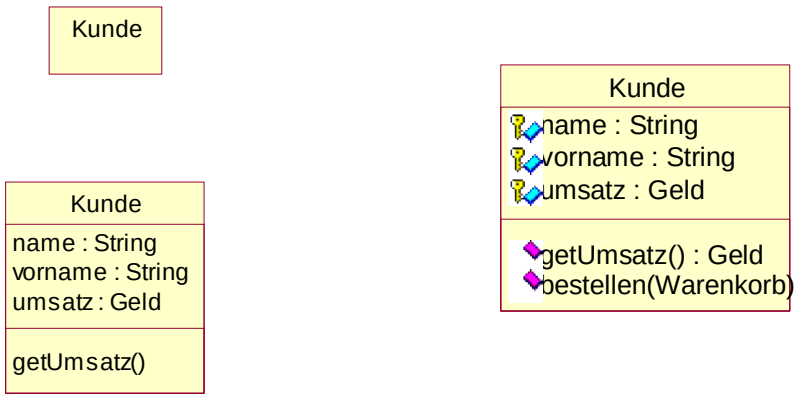
Mit dem Klassendiagramm kann der Unterschied zwischen Analyse-Phase und Design-Phase gezeigt werden.

In der Analyse-Phase wird das Problem möglichst nah am Fachbereich beschrieben. Details der Implementierung haben hier nichts verloren.

Bei der Implementierung können wir so viele Details wie möglich, wie z. B. Parameter, Rückgabetypen oder Zugriffsmodifizierer, anzeigen.

Gute Tools erlauben den Wechsel der Sichtweisen und „verstecken“ bei Bedarf die Details.

Beispiel: Darstellung derselben Klassen in den verschiedenen Phasen.



6.3.4 Sichtbarkeit

Die Sichtbarkeit eines Attributes oder einer Operation wird in der UML durch Verwendung von +,# oder – dargestellt. In JAVA gibt es aber vier Stufen der Sichtbarkeit: neben private, protected und public gibt es noch default bzw. package.
Für default Sichtbarkeit kann man z. B. einfach kein Symbol (oder ein spezielles) im Diagramm verwenden

Das Tool Rational Rose verwendet eigene Symbole für Sichtbarkeit, die Sie in den vorhergehenden Beispielen sehen konnten.

Zeichen	Sichtbarkeit
+	öffentlich(public)
#	geschützt(protected)
-	privat(private)

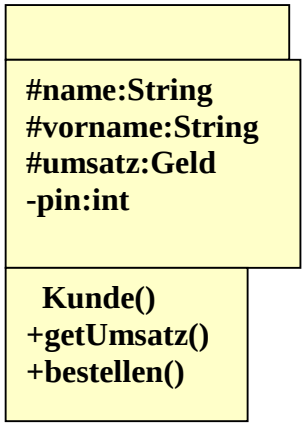


Abb. 6.8: Sichtbarkeit nach UML Notation

6.4 Sequenzdiagramme

Im Sequenzdiagramm kann der Ablauf einer Methode dargestellt werden. Für die Aufgabe des Sequenzdiagramms war in der prozeduralen Programmierung das Struktogramm und der Programmablaufplan zuständig. Im Sequenzdiagramm wird dargestellt, welche Objekte zusammenarbeiten. Schleifen werden nicht modelliert. Die Zeitachse geht im Diagramm von oben nach unten.

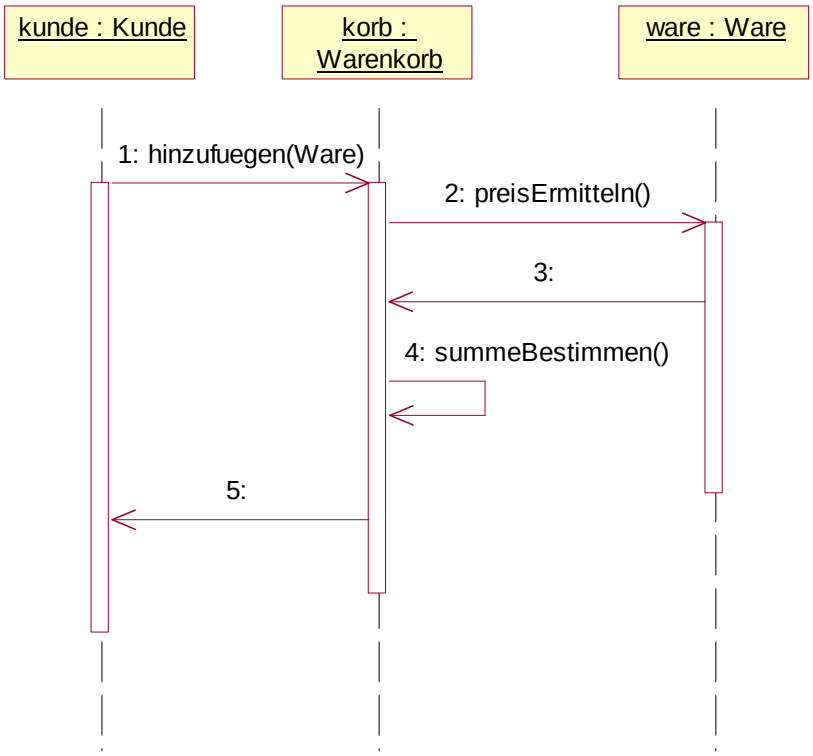


Abb. 6.9: Interaktionen der Methode einpacken(Ware) von Kunde

Die Abbildung 6.9 zeigt das Sequenzdiagramm für die Methode einpacken(Ware) von Kunde. Nach Aufruf der Methode einpacken(Ware) wird die Methode hinzufuegen(Ware) von Warenkorb aufgerufen. Diese wiederum ruft preisErmitteln() von Ware auf. Danach wird preisErmitteln beendet und zum Warenkorb zurückgesprungen. Im Schritt 4 wird eine andere Methode in der selben Klasse aufgerufen. Der Pfeil zeigt jetzt auf das aufrufende Objekt zurück. Anschließend wird im 5. Schritt die Methode hinzufuegen beendet und zur Methode einpacken von kunde zurückgekehrt.

6.5 Paketdiagramme

Gruppierungen von UML Elementen können in einem Paket zusammengefasst werden. Ein Paket kann selbst aus anderen Paketen oder aus beliebigen Elementen bestehen. Alle UML Diagramme und Schaubilder können in Pakten organisiert werden.
In Java Projekten können mit Paketdiagrammen die Abhängigkeiten von Packages dargestellt werden.

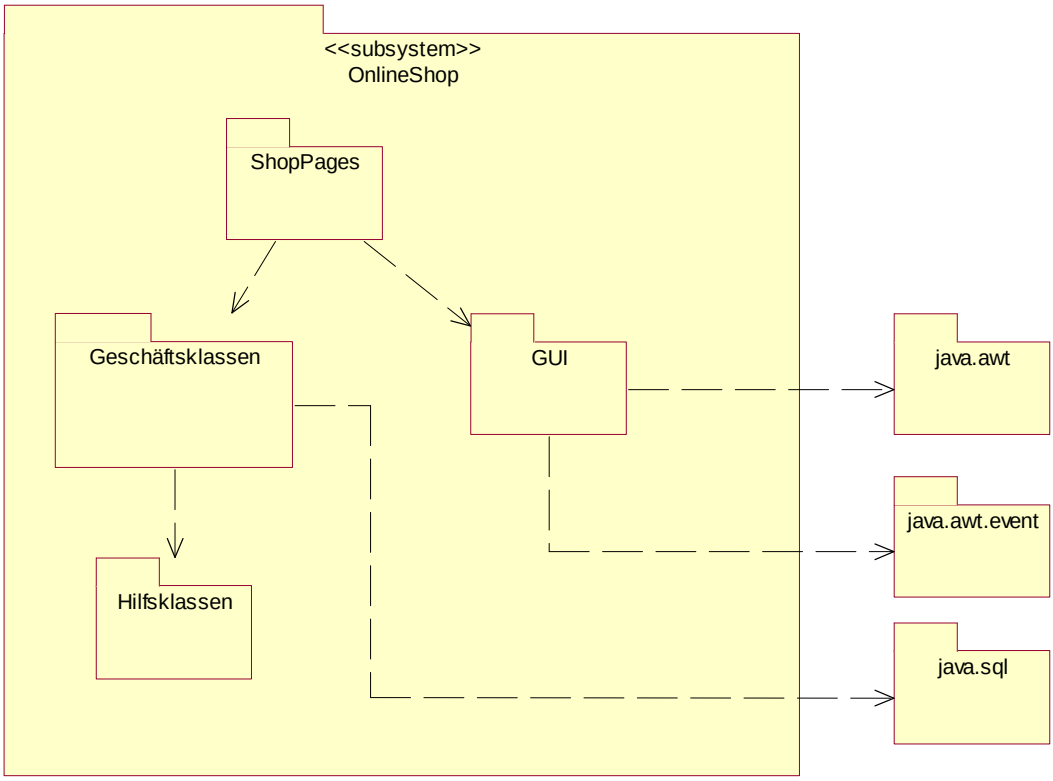


Abb. 6.10: OnlineShop und packages

6.6 Zustandsdiagramme

Mit dem Zustandsdiagramm lassen sich die verschiedenen Zustände eines Objekts oder Systems mit Ihren Übergängen darstellen. Man verwendet Zustandsdiagramme nur bei Objekten mit einem ausgeprägten dynamischen Charakter.

Die Zustandsdiagramme der UML sind von David Harel's Diagrammen abgeleitet, wurden aber leicht verändert, um der Objektorientierung gerecht zu werden.

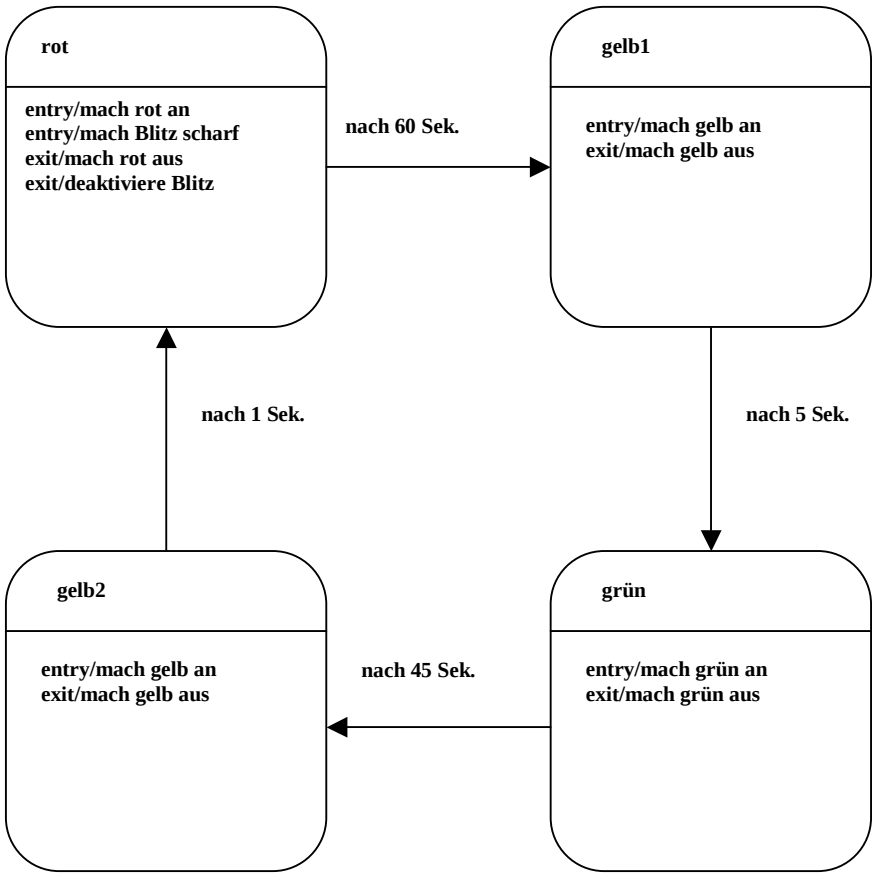


Abb. 6.11: Zustandsdiagramm einer Ampel

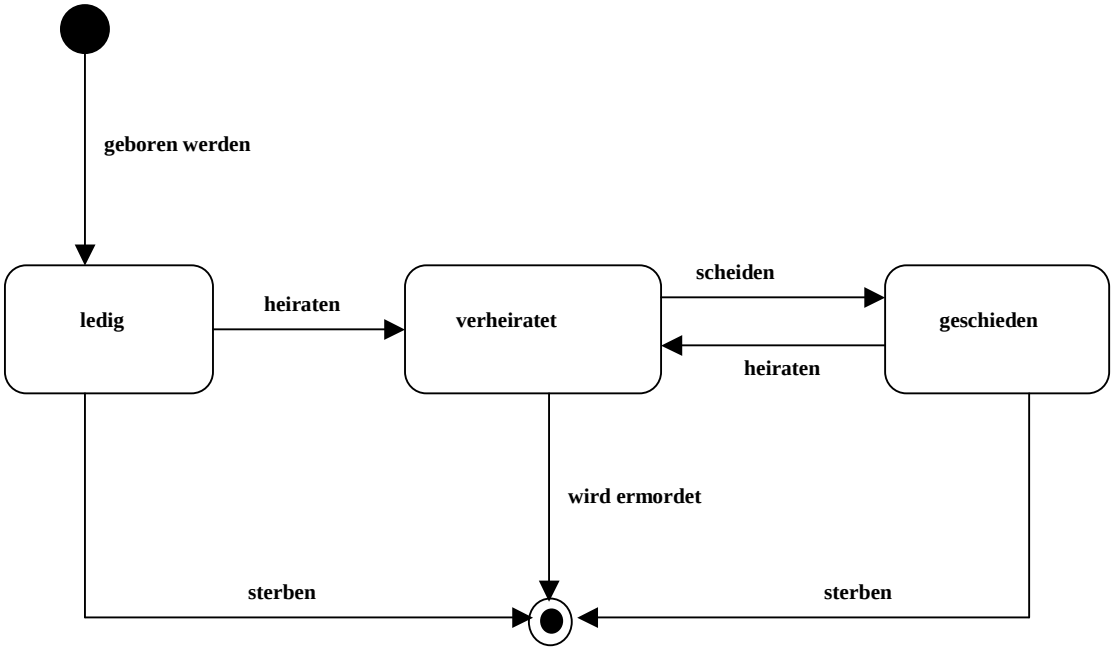


Abb. 6.12: Zustände Mensch-Objekt

6.7 Aktivitätsdiagramme

Aktivitätsdiagramme haben starke Ähnlichkeit mit Flussdiagrammen. Mit Flussdiagrammen lassen sich jedoch nur sequentielle Abläufe Modellieren, mit Aktivitätsdiagrammen lassen sich auch nebenläufige Aktivitäten darstellen.

Nachdem der UML Notation Guide, Fowler und Burkardt das Beispiel mit der Kaffeemaschine aufführen, soll es hier nicht fehlen (Eine Kaffeemaschine passt eben auch gut zur Sprache JAVA.).

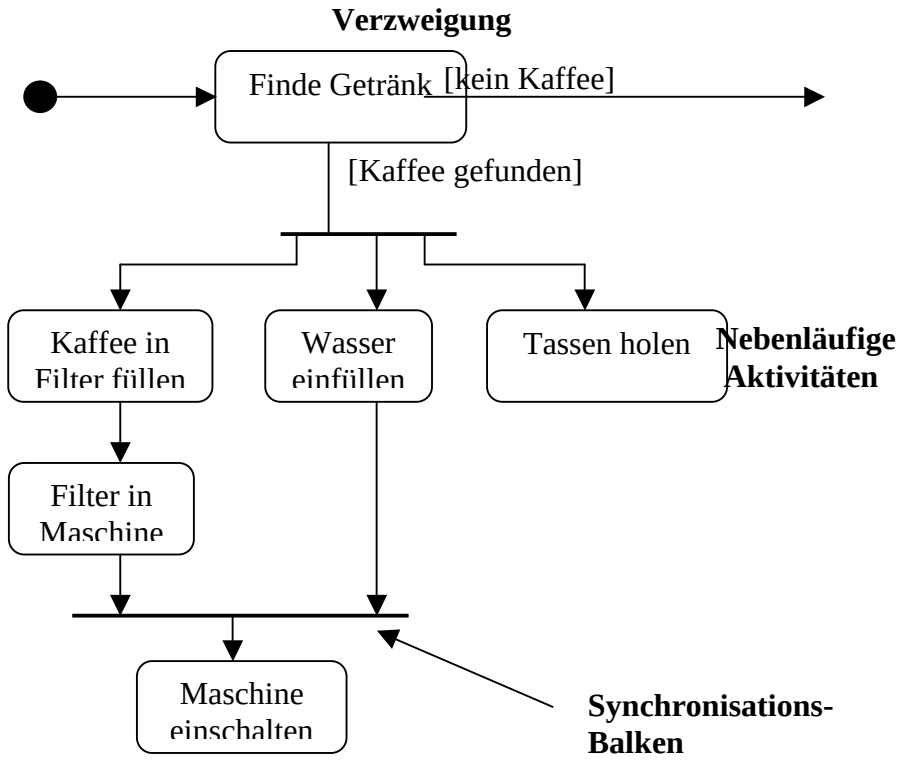


Abb.6.13: Ausschnitt aus Aktivitätsdiagramm

Die Abbildung 6.13 zeigt einen Ausschnitt aus dem Beispiel mit der Kaffeemaschine. Nebenläufigkeiten lassen sich mit einem Synchronisationsbalken synchronisieren. Nach der Aktivität finde Getränk spaltet sich die Ausführung auf die drei Pfade „Kaffee in Filter füllen“, „Wasser einfüllen“ und „Tasse holen“ auf. Mit zwei Händen oder einem „Helfer“ lassen sich diese Aktivitäten parallel verrichten.
Die Aktivität „Maschine einschalten“ setzt voraus, dass der Filter in die Maschine eingelegt und Wasser eingefüllt ist. Erst wenn beide Bedingungen erfüllt sind können wir die Kaffeemaschine einschalten.

Diese Synchronisation können wir wieder mit einem Synchronisationsbalken darstellen.

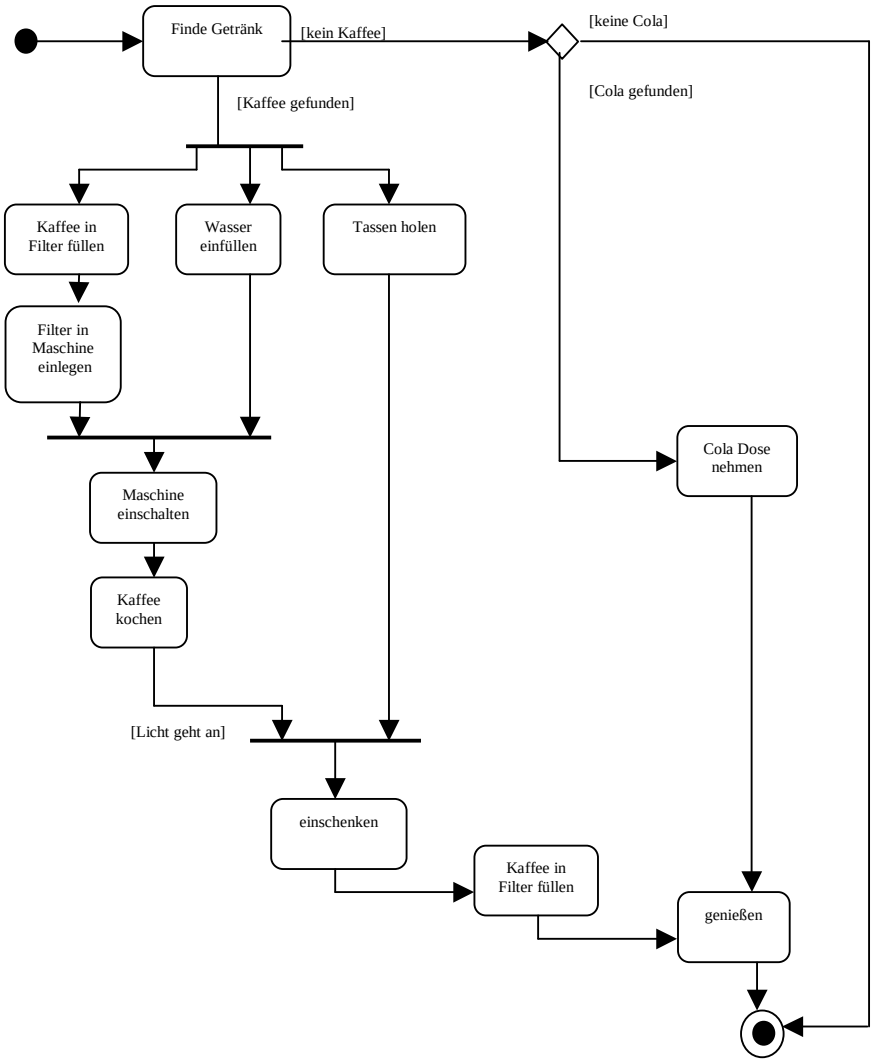

Finde Getränk
[kein Kaffee]
[keine Cola]
[Kaffee gefunden]
[Cola gefunden]
Kaffee in Filter füllen
Wasser einfüllen
Tassen holen
Filter in Maschine einlegen
Cola Dose nehmen
Maschine einschalten
Kaffee kochen
[Licht geht an]
einschenken
Kaffee in Filter füllen
genießen

Abb. 6.14

Übungsaufgaben

6.1 Entwickeln Sie die Use-Cases für ein Reisebuchungssystem!

6.2 Darstellung des AWT als Klassendiagramm

a) Benutzen Sie die konzeptionelle Sicht des Klassendiagramms. Beschränken Sie sich auf den Klassennamen und lassen Sie Attribute und Operationen weg.
b) Beschreiben Sie nur die Vererbungsbeziehungen!
c) Benutzen Sie für diese Übung die Online-Dokumentatio
d) Ihr Diagramm sollte folgende Klassen beinhalten:

Button, Liste, Checkbox, Window, Container, Object, Component, Applet, Panel, Textfield, Textcomponent und Frame.

6.3 Kodierung in JAVA: Arbeiten Sie eine Implementierung für den Warenkorb und Bestellung in JAVA aus. Beschränken Sie sich dabei auf die Attribute!

6.4 Entwickeln und diskutieren Sie gemeinsam ein Klassendiagramm für eine Fakturierung. Folgende Dinge sollen abgebildet werden:

Kunden
Verkäufer
Waren
Rechnungen

7 Objektorientierte Analyse

7.1 Ziele der OOA

Das Ziel objektorienter Analyse ist es:

- Der Entwurf einer eindeutigen, strukturierten und verständlichen Darstellung ***was*** das zu entwickelnde Software-System leisten soll, insbesondere eine Beschreibung der Anforderungen und des Nutzeffekts des Systems.

- Eine Bewertung und Auswahl aus den Möglichkeiten, ***wie*** das System EDV-technisch umgesetzt werden kann und eine Bewertung der Risiken dieser Möglichkeiten.

- Eine Einschätzung der zur Umsetzung des Projekts verfügbaren Ressourcen und sonstigen projektspezifischen Rahmenbedingungen.

7.2 Aktivitäten der OOA

Zur Erreichung der Ziele der OOA sind im Wesentlichen folgende Aktivitäten notwendig:

- Eine Dekomposition (Zerlegung) des Problem- und Anwendungsbereichs bzw. des Gesamtsystems in verständliche und überschaubare Teilbereiche bzw. Subsysteme.

- Eine Analyse und Darstellung des benötigten Systemverhaltens.

- Eine Darstellung der Anforderungen in einer Umsetzung in EDV-gerechte Abstraktionen.

- Eine Konzeption von Modellen für die Analyse, die es gestatten, die Widerspruchsfreiheit, Erfüllbarkeit und Vollständigkeit der Anforderungen zu überprüfen.

- Eine Validierung der Analysemodelle.

7.3 Input der OOA

Der Analyse geht eine gewisse Konzept- und Projektdefinitionsphase voraus. Meistens wird diese Phase durch die einem größeren Projekt vorausgehende Entscheidungsphase vorbestimmt. Diese Vorüberlegungen führen bereits zu einer Sammlung von Dokumenten, die man als Input für die eigentliche Analysephase betrachten kann, wie etwa:

- Problembeschreibung
- Beschreibung und Darstellung des Ist-Zustandes in punkto Aufgaben-Rollenverteilung, Organisation, Geschäftsprozesse, Besprechungsprotokolle, etc.
- Zielbeschreibung
- Pflichtenheft mit Benutzeranforderungen und relevanten Vorschriften.
- Darstellung der Entwicklungsbasis mit Klassenbibliotheken, Frameworks, Hardware und bestehende Anwendungen, Prototypen, Ablaufmodellen, etc.

7.4 Beteiligte der OOA

- Problembereichsexperten sowie System- und Anwendungsexperten, die sowohl das bestehende System mit seinen Geschäftsprozessen und Abläufen kennen, als auch die technischen und wirtschaftlichen Flaschenhälse des zu erstellenden Systems einschätzen können.
- System- und Anwendungsentwickler zur genaueren Einschätzung der Möglichkeiten der aktuellen EDV-Technologie zur Unterstützung von Geschäftsprozessen und Systemabläufen
- Entscheidungsträger, die bei Bedarf auch noch
- neutrale Berater und Gutachter mit ins Boot nehmen können.

7.5 Abgrenzung der Problembereiche (Separation of Concerns)

Ein Mittel zur Verringerung der Komplexität in großen Systemen ist die Abgrenzung in Problembereiche. Ein aktuelles Software-System zerfällt im Prinzip in die vier Problembereiche:

- Grafische Benutzeroberfläche
- Anwendungsmodelle
- Datenhaltung

7.6 Grafische Benutzeroberfläche

Die grundlegenden Elemente einer grafischen Benutzeroberfläche sind:

- Ein Interface-Objekt (Event-Handler) als Schnittstelle zwischen dem Benutzer und dem System
- Ein GUI-Controller zur Erstellung und Verwaltung der aktuellen View-Objekte
- Formularobjekte zur Aufnahme der Daten, die in den einzelnen Views dargestellt werden.

Zur Konzeption der Views und zur Entwicklung der entsprechenden Objekte stehen zahlreiche Tools zur Verfügung, meist mit Integration in Entwicklungsumgebungen. Deshalb kann man den GUI-spezifischen Objekten in der Analyse eine untergeordnete Rolle zukommen lassen.

7.7 Anwendungsmodelle

Die Anwendungsmodelle führen Nutzungsfälle (wie Prozesse, Transaktionen,...) gemäß den spezifizierten Szenarien aus. Sie stellen hierzu die entsprechenden Services bereit. Daten aus verschiedenen Informationsquellen werden hier z. B. über Berechnungen miteinander verknüpft. Hierzu werden benötigt:

- Interface-Objekte zur Übergabe/Übernahme von Daten an die/von der Benutzeroberfläche bzw. den Datenhaltungssystemen.

- Ein Controller-Objekt zur Prozesssteuerung, das die entsprechenden Server-Objekte startet und beendet und ihnen die Daten zuführt.

- Server- (Service) Objekte zur Verarbeitung der Daten und Dienste der Entity-Objekte.

- Passive Entitiy-Objekte (oder auch Business Objects, Fachobjekt) zur Haltung der Daten bei den Server-Objekten und Bereitstellung von entsprechenden Datendiensten. Sie repräsentieren zumeist reale Objekte des Anwendungsbereichs, verknüpft durch Assoziation oder Aggregation. Entity-Objekte sind entweder

  - **persistent,** d.h. sie werden in einem permanenten Objektverwaltungssystem gehalten, entweder als Teile einer Tabelle in einer relationalen Tabelle oder als Elemente einer Domäne einer objektorientierten Datenbank.

    oder

  - **transient**, also nicht beständig, z.B. in Form eines bei Bedarf lokal verfügbaren Entity-Repositorys.

  Die Entity-Objekte repräsentieren das Informationsmodell.

7.8 Datenverwaltung

Die Datenverwaltung ist für die Bereitstellung der Daten und die damit verbundenen Verwaltungsaufgaben zuständig. Hierbei relevante Objekte sind die Informationsträger mit ihren Diensten. Die aus diesen Diensten resultierenden Objekte werden als Entity-Objekte bezeichnet. Über spezielle Interface-Objekte wird der Zugriff auf die Datenbank geleistet. Darüber hinaus stehen Enitity-Factories zur dynamischen Erzeugung und Verwaltung der lokalen Entity-Objekte zur Verfügung. Somit besteht die Datenverwaltung aus folgenden abhängigen Bestandteilen:

- Entity-Objekte
- Entity-Factories
- Datenbank-Interfaces.

7.9 Kommunikationssystem

Das Kommunikationssystem stellt Schnittstellen und Treiber zur Übermittlung von Nachrichten und Daten zu peripheren Systemen und Geräten zur Verfügung. Die Objekte des Kommunikationssystems verkörpern darüber hinaus Geräte- oder Systemschnittstellen, oder helfen bei der Anpassung zwischen unterschiedlichen Kommunikationsstandards. Das Kommunikationssystem besteht also aus den Objekttypen:

- Interface-Objekte.
- Treiber-Objekte (Kapselung der Kommunikationsdetails)
- Vermittler-Objekte (zur Konvertierung unterschiedlicher Datenformate)
- Vertreter-Objekte (Proxy-Objekte, zur Vertretung des Senders/Empfängers auf der jeweilig anderen Seite der Kommunikation)

7.10 Der Analyseprozess

Der Analyseprozess wird sich mit verschiedenen **Analysebereichen** befassen wollen:

- Die Analyse des Problem- und Anwendungsbereiches (der Domäne)
- Eine Ist/Soll-Analyse der Vorgänge und Prozesse (auch Anwendungsanalyse)
- Eine Anforderungsanalyse mit Analyse der Nutzungsfälle (black box) der systeminternen Klassen und Objekte (white box) sowie dem Entwurf von Modellen zur Analyse des Systemverhaltens

Er zerfällt zumeist in die folgenden Phasen:

- Abgrenzung des Systembereichs, Analyse der Nutzungsfälle
- Auffinden und Abstraktion der relevanten Objekte in der relevanten Granularität.
- Analyse der statischen Aspekte
- Analyse des dynamischen Verhaltens
- Dokumentation und Überprüfung der Analyseergebnisse.

Die Phasen und ihre Einbettung in einen OO-Entwicklungsprozess zeigt das nächste Kapitel. Die folgenden Abschnitte sollen die drei Bereiche nach ihrem Auftreten im Projektablauf und den Nutzungsmöglichkeiten der UML für die einzelnen Arten darstellen.

8 Skizze eines Entwicklungsprozesses

Im Rahmen dieser Schulung soll kein bestimmter Prozess im Vordergrund stehen, vielmehr soll aufgezeigt werden, welche Faktoren bei der Definition eines Prozesses eine Rolle spielen können und wie sich die UML sinnvoll in diese Überlegungen einbinden lässt.

Hierbei soll auch nicht weiter auf die oftmals getroffene Unterscheidung zwischen Analyse und Design eingegangen werden, ein einfaches Voranschreiten durch einen möglichen Entwicklungsprozess soll genügen.

Dieses wird aber durch einen zusätzlichen Mechanismus, die Iteration, weiter verfeinert. *Iterative Entwicklung* bedeutet hier: Analyse und Design wird nicht nur zu Beginn eines Projekts, sondern auf der Basis vieler Zwischenstufen durchgeführt. „Iterative Entwicklung sollten Sie nur für Projekte vorsehen, von denen Sie wollen, dass sie erfolgreich ablaufen“ (Martin Fowler). Die Iteration gestattet ein frühzeitiges Aufdecken von Risiken und eine hervorragende Kontrolle über den Entwicklungsprozess.

Betrachten wir folgende Phaseneinteilung eines iterativen Entwicklungsprozesses:

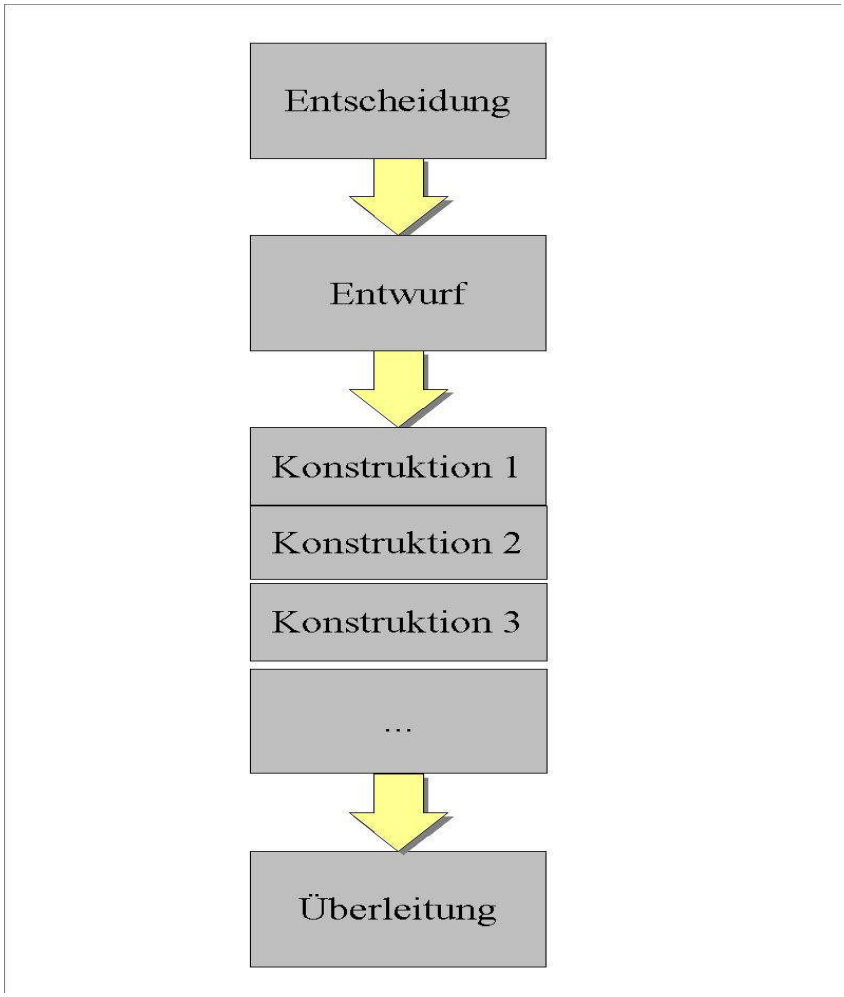


Abb. 8.1 Phaseneinteilung eines iterativen Entwicklungsprozesses

Ziel der *Entscheidungsphase ist es*, die geschäftlichen Zielsetzungen für das Projekt zu begründen und den von ihm abgedeckten Bereich zu skizzieren. Erst nach dieser Phase liegt die Bestätigung für den finanziellen Rahmen des Projekts vor, trotzdem muss gerade für eine so weit reichende Entscheidung schon ein gutes Projektverständnis in einer von Kunde und Entwickler gemeinsam gesprochenen Sprache vorliegen.

Je nach Projektumfang kann diese Phase natürlich im Umfang zwischen einem Gespräch im Aufzug und einem Gutachten im Aufwand von mehreren Mannjahren liegen.

Ziel ist in jedem Fall eine Abwägung der Kosten und des Nutzens des Projekts.

In der *Entwurfsphase* werden die Anforderungen an das Projekt weiter detailliert. Hierzu wird die Architektur des Systems spezifiziert, sowie Analyse und Design bis auf eine Ebene durchgeführt, die eine Aufteilung in Konstruktionsiterationen sinnvoll werden lässt.

Die *Konstruktionsphase* selbst läuft iterativ und inkrementell. Immer neue Teile werden in ihrem Ablauf entwickelt und freigegeben. Sie setzt sich aus einzelnen Iterationen zusammen. Nach jeder Iteration steht ein ausgetesteter und neu integrierter Stand zur Verfügung. In jeder Iteration finden die üblichen Abläufe Analyse, Entwurf, Implementierung und Test statt.

Eine *Überleitungsphase* widmet sich abschließenden Aufgaben, wie der Optimierung des Gesamtsystems oder der Fehlerbehebung. Hier sollte keine Funktionalität mehr hinzugefügt werden. Es könnte sich hierbei um die Zeit zwischen Freigabe einer Beta-Version und endgültiger Freigabe eines Produktes handeln.

Im Folgenden soll am Beispiel der Entscheidungsphase und der Entwurfsphase aufgezeigt werden, welche Techniken dem Entwickler seitens der UML oder anderen Modellierungssprachen gegeben sind, und bei welchen Aufgaben die einzelnen Techniken ihre Vorteile am besten entfalten können.

8.1 Die Entscheidungsphase

- Untersuchung und Beschreibung eines Problembereichs
- „*Was* ist zu tun?“ steht im Vordergrund
- Präzisiert Pflichtenheft oder Aufgabenstellung
- Verwendung einer präziseren Notation als der Umgangssprache
- Identifikation einiger Klassen und Objekte, erste Beschreibung von statischer Klassenstruktur und erste Modellierung der Abläufe und Interaktion von Objekten
- Führt Benutzer und Entwickler zusammen

8.2 Identifikation von Klassen

- Heuristiken (Coad 1991)
- Ableitung aus Geschäftsvorfällen (Jacobsen 1992)

- grammatikalische Untersuchung des Pflichtenheftes (Rumbaugh 1991)

  Idee: Substantive in Problembereichsbeschreibung könnten Klassen-kandidaten oder Attributskandidaten identifizieren, Verben könnten Methodenkandidaten aufzeigen.

- Ableitung aus Datenflussdiagrammen (Booch 1986)

  Ableitung aus der dritten Normalform eines ER-Modelles (Shlaer 1988)

8.2.1 Sinn und Zweck des Systems / wichtigste Features

- Anfertigen einer kompakten Beschreibung des Zweckes des Systems (< 50 Worte, warum dieses System, warum jetzt)

  - ❖ Gespräche mit Experten auf diesem Gebiet
  - ❖ Bilder, Beispiele, ...
  - ❖ Glossar - einheitliche Sprechweise der Beteiligten
  - ❖ Identifikation kritischer Bereiche der bisherigen „Welt“ des Anwenders

- Anfertigen einer Liste aller wichtigen Features des Systems

  - ❖ Anwendungsfälle (Use-Cases) können ein Hilfsmittel sein, Features zu identifizieren und in ihrer Wichtigkeit abzuschätzen
  - ❖ Bewertung der Wichtigkeit und Feststellung des Benutzers jedes Features
  - ❖ Wichtige Gruppen von Features
    - ∗ Geschäftsaufnahme (z.B. Stammdatenverwaltung)
    - ∗ Geschäftsdurchführung (z.B. Auftragsabwicklung)
    - ∗ Geschäftsanalysen
    - ∗ Kooperation mit anderen Systemen
    - ∗ Informationsspeicherung
  - ❖ Bewertung der Qualität der Features am Markt, Vergleich mit der Konkurrenz
    - ∗ Welches sind meine Highlights?
    - ∗ Welches sind meine Schwachpunkte?
    - ∗ Welche Features sind überflüssig?
  - ❖ Welches sind die zukünftigen Veränderungen, welche Features sind davon wie betroffen

8.3 Die Entwurfsphase

Jetzt erst ist der Projektrahmen eindeutig abgesteckt und das Startsignal erteilt. Zu diesem Zeitpunkt existiert typischerweise nur eine vage Vorstellung davon, welche Anforderungen an das System gestellt werden. Je nachdem, wie viel Zeit und Ruhe die Entscheidungsphase ließ, gilt es jetzt erst einmal, die genaue Architektur des Systems festzulegen.

Insbesondere gilt es, die Entscheidung über Entwicklungswerkzeuge und verwendete Technologien zu treffen, aber auch einen Verständnisgrad des Projektes zu erreichen, bei dem es möglich wird, die Konstruktionsphase in ihrer Iterationsstruktur zu erkennen. Die möglichen Risiken des Projekts werden hier als Motor für die weiteren Überlegungen dienen.

8.4 Anforderungsrisiken

Eines der Hauptrisiken für Projekte besteht nach wie vor in der simplen Tatsache, dass das falsche System entwickelt wird, eines das nicht das leisten kann, was der Kunde erwartet. Will man dieses Risiko minimieren, so findet man in den Techniken der UML eine geeignete Hilfsmittelsammlung.

- Ausgangspunkt sind ***Anwendungsfallbeschreibungen.*** Man beschreibt hierzu ein typisches Zusammenspiel von Benutzer und System, und lässt den Benutzer dabei ein bestimmtes Ziel erreichen. Man wird natürlich selten in der Lage sein, alle Anwendungsfälle eines Systems zu beschreiben. Trotzdem sollte man in der Entwurfsphase Teams aus Entwicklern und Benutzern damit beauftragen, eine möglichst große Sammlung zu erstellen. Anwendungsfälle treiben den gesamten weiteren Entwicklungsprozess.
- Zusätzlich zu den Use-Cases gilt es, ein konzeptionelles Verständnis des Problembereichs zu erarbeiten und in einem Problembereichsmodell zu dokumentieren.

Modelliert man beispielsweise die Geschäftsprozesse einer Firma, so sollte man hierzu diejenigen Mitarbeiter zu Rate ziehen, die mit den Geschäftsabläufen (und dabei insbesondere den firmeninternen Schnittstellen) vertraut sind.

Eine Klärung der zentralen Begrifflichkeit ist die Hauptaufgabe dieser Beschreibung (Was bedeuten Begriffe wie „Kunde“, „Standort“, „Dienst“ und „Rechnung“ und wie passen sie zusammen?). Diese Beschreibung legt die Grundlage für das Objektmodell des Systems.
Aus der UML sind die ***Klassendiagramme*** aus einer konzeptionellen Perspektive eingesetzt, ein großartiges Hilfsmittel, die Sprache des Problembereichs zu erfassen. Auch **Aktivitätsdiagramme** sind ein Hilfsmittel für dieses Modell. Sie gestatten es, die Arbeitsabläufe im Unternehmen zu beschreiben, insbesondere auch parallel ablaufende Aktivitäten.

- Es ist ratsam, die Sammlung von Anwendungsfällen und die Erstellung eines Problembereichsmodells möglichst von Anfang an zu synchronisieren, indem man z.B. frühzeitig zur Sammlung von Anwendungsfällen Problembereichsexperten einsetzt. Ein frühzeitiges Verständnis des Problembereichs erleichtert umgekehrt die Auffindung relevanter Anwendungsfälle enorm.

Am Ende dieser Phase sollte die in Anwendungsfällen, Klassen- und Aktivitätsdiagrammen gesammelte Information möglichst in einem einzigen großen ***konsolidierten Problembereichsmodell*** vorliegen. Ist das Projekt zu groß, lassen sich ***Paketdiagramme*** zur weiteren Unterteilung einsetzen.

Auch in dieser abschließenden Phase sollte man wieder Problembereichsexperten hinzuziehen, die sich eingehender mit der Modellierung befassen können.

Wenn es die Situation erfordert, kann man durchaus einige interessante Klassen herausgreifen und in ihrem Lebenszyklus mit einem ***Zustandsdiagramm*** aufzeigen.

Ein schnell, nicht unbedingt in der Technologie des späteren Systems, entwickelter ***Prototyp*** mit heiklen Anwendungsfällen ist außerdem ein hervorragendes Hilfsmittel zum Verständnis von dynamischen Abläufen.

8.4.1 Technologische Risiken

Auch hier sind Prototypen ein hervorragendes Hilfsmittel, in diesem Fall um Teile der geplanten Technologien zu erproben. Mögliche Vorgehensweise:

- Bereitstellung des Compilers und der Entwicklungswerkzeuge
  Umsetzen einiger Anwendungsfälle und dabei Erprobung der Werkzeuge
  Integration mit weiteren Komponeten des späteren Systems (Datenbanken, Netzwerk)

- Variation der Kombination von Werkzeugen und Komponenten

Das größte technologische Risiko besteht in der Verknüpfung der einzelnen Komponenten. Hieraus gewinnt man wichtige Erkenntnisse für architektonische Entscheidungen. **Klassendiagramme** und **Interaktionsdiagramme**

(zur Darstellung der Kommunikation der Komponenten) und **Paketdiagramme** (als Bild der Komponentenstruktur) sowie **Verteilungsdiagramme** (bei verteilten Systemen) können (sparsam eingesetzt) als Hilfsmittel dienen, diese Überlegungen zu dokumentieren.

8.4.2 Risiken bei den Fähigkeiten

Diese Überlegungen sind letztlich eine Abschätzung, ob man lieber mehr Geld in die Projektlaufzeit oder in die Ausbildung investieren möchte. Gute Trainer helfen durch Schulungen, Fehler zu vermeiden, indem sie von den eigenen berichten und durch Coaching (das Hinzuziehen eines in OO-Techniken erfahrenen Entwicklers für begrenzte Zeit) die Ausbildung in die Zeit hinein zu verlängern, in der das Projekt bereits begonnen hat. Auch eine regelmäßige Begutachtung des Projekts kann ein Mittel sein, kritische Bereiche frühzeitig zu identifizieren und Zeitpläne dadurch besser einhalten zu können. Vorhandene Eigeninitiative der Entwickler zu unterstützen, wird sich immer auszahlen. Dies kann sich, was die Ausbildung betrifft, zum Beispiel im Errichten einer Bibliothek und im Einrichten von Lesegruppen niederschlagen.

8.5 Ende der Entwurfsphase

Die Kunst des Analytikers besteht darin, bei der Erstellung des Gerippes (das ein Problembereichsmodell ja darstellen soll) zwischen Knochen und Fleisch zu unterscheiden, also das rechte Maß zwischen Detail und Überblick zu finden.

Das kleinste noch sinnvolle Team zur Erstellung eines Problembereichsmodells sollte zumindest aus einem Entwickler und einem Problembereichsexperten bestehen. Beide sollten darin ausgebildet sein, welche UML-Diagramme ihnen zum gegenseitigen Verständnis verhelfen können.

Ein harter Zieltermin für das Team hilft, das Maß in der Abwägung zwischen Knochen und Fleisch zu behalten. Die Dauer sollte über den Daumen ein Fünftel der gesamten Projektdauer nicht überschreiten.

Ergebnisse dieser Modellierungsphase sollten sein:

Die grundlegende Architektur des Systems muss geklärt sein.

Dies beinhaltet:

- Eine Liste von Anwendungsfällen, die das Pflichtenheft wiedergeben
- Ein Problembereichsmodell als Ausdruck des Verständnisses des Unternehmens und zur Dokumentation zentraler Klassen
- Eine Beschreibung der eingesetzten Technologien und Plattformen und ihrer Verknüpfungen

Es sollten alle wichtigen Risiken identifiziert und die meisten so gut verstanden sein, dass sie beherrschbar sind.

Ein Plan für die Konstruktionsphase muss vorliegen.

- Hierzu kategorisiert man die Anwendungsfälle nach Prioritätsgrad, technologischem und planerischem Risiko.
- Man lässt die Entwickler in Personenwochen abschätzen, wie viel Aufwand in den ihnen zugeordneten Anwendungsfällen steckt. Zu jeder Schätzung gehören Analyse, Entwurf, Programmierung, Modultest, Integration und Dokumentation.
- Wenn an Anwendungsfälle mit hohem zeitlichen Planungsrisiko ein hohes technologisches Risiko geknüpft ist und sie mit einem hohen zeitlichen Aufwand verbunden sind - dann ist der Plan falsch.
- Alle 4-8 Wochen sollte eine Iteration (ein Zwischenstand) durchgeführt werden. Aus der Summe der Entwickler und ihrer durchschnittlichen Effektivität sollte man zusammen mit der Aufwandsschätzung für die Anwendungsfälle und der durchschnittlichen Iterationsdauer berechnen können, wie viele Iterationen die Konstruktionsphase durchlaufen wird.
- Die Anwendungsfälle hoher Priorität und hohen Risikos wird man in frühen Iterationen abwickeln wollen.
- Schließlich kann man noch etwa 20 Prozent der Konstruktionszeit für die Überleitungsphase reservieren, einige Puffer einbauen und gelangt zu einer guten Übersicht, wie das weitere Projekt abläuft.

8.6 Die Konstruktionsphase

Jede Iteration kann als eigenständiges Projekt gesehen werden. Abschließende Tests (Modultests durch Entwickler und Systemtests durch Anwender, Testaufwand bei 50 Prozent des Gesamtaufwands, eigenständiges Testteam) sichern funktionierende Zwischenstände.

Die Iterationen sind inkrementell im Hinblick auf die Funktionalität. Jede weitere Iteration kann auf den Anwendungsfällen ihrer Vorgänger aufbauen.

Sie sind iterativ im Hinblick auf die Codebasis. Immer wieder wird man ein Umstrukturierung (*refacturing*) des Codes im Hinblick auf seine Flexibilität und Übersichtlichkeit vornehmen. Auch bei diesem Prozess profitiert man von vorher geleisteten Überlegungen, den Code erweiterbar im Sinne der OO zu entwerfen. Hierbei sollte diszipliniert zwischen Umstrukturierungs- und Erweiterungsphasen getrennt und nach jedem größeren Schritt ausgiebig getestet werden.

In der Konstruktionsphase kann man von allen Elementen der UML Gebrauch machen.

Man benutzt ***Use-Cases*** zur Abgrenzung des Arbeitsgebiets, wenn man einen bestimmten Anwendungsfall abzudecken hat.

Klassendiagramme und **Aktivitätsdiagramme** werden wie in der Entwurfsphase Verwendung finden, wobei der Detaillierungsgrad wachsen wird. Hierbei kann man auch **Klassenkarten** (nicht Element der UML) verwenden, um erst einmal das Verhalten pro Klasse zu erfassen und dieses dann im Ganzen mit Diagrammen zu dokumentieren.

Auch alle weiteren Elemente der UML halten einen großen Fundus an Anwendungsbereichen und Detaillierungsgraden bereit, ein streng vorgegebener Einsatz, der die Freiheiten des Entwicklers im Wunsch nach eigener Darstellung einschränkt, erscheint wenig sinnvoll.

8.7 Hilfsmittel bei der Erstellung der Diagramme

8.7.1 Hilfsmittel zur Auswahl der Klassen des Problembereichs

In der „wirklichen Welt“ stehen wir Objekten mit ihren Werten und ihrem Verhalten gegenüber - wie kommt man von diesen zu Klassen?

- Abstraktion - Klasse einordnen in die Beschreibung der Objektschnittstellen und -internas, der Klassenschnittstellen und die schon bestehende Klassenhierachie
- Welches sind die relevanten Objekte (Personen, Plätze, Dinge, ...)
- Betrachtung der Features:
  - ❖ Welches sind die Rollen (Kunde, Verkäufer, Angestellter, Student, Empfänger)?
    - ➢ Wer spielt die Rollen, welches sind die handelnden Objekte (Personen Organisationen, ...)?
    - ➢ In welchem Zeitraum (Zeitpunkt/Intervall)?
    - ➢ An welchem Ort?
  - ❖ In welche Behältnisse/Kataloge wird eingebettet?
  - ❖ Woher kommen die benötigten Daten?

8.8 Skizze eines UI und seiner Klassen

- Betrachtung der Features
  - ❖ Wo liegen Listen vor, welche Listenhierarchie ist erkennbar?
  - ❖ Wo sind Auswahlen zu treffen?
  - ❖ Wo gibt es Eingabefelder, wo Anzeigefelder?
  - ❖ Welche Aktionen werden ausgelöst, welche Bewertungen durchgeführt?
- Anordnung der Elemente - graphisches Layout

8.8.1 Szenarien / Use-Cases

- Ausarbeitung von Anwendungsfällen
- Mögliche Trennung in Anwendungsfällen für Geschäftsaufnahme/-durchführung
  - ❖ Erstellung von Beispielen für Use-Cases von einigen der Features aus dem Bereich der Geschäftsaufnahme
  - ❖ Erstellung von Use-Cases für alle Features aus den Bereichen Geschäftsdurchführung und –analyse
- Dabei geht man folgende Aufgaben an:
  - ❖ Identifizierung der relevanten Rollen und Rollenspieler, dies führt zu ersten Klassendefinitionen.
  - ❖ Beschreibung der Durchführung von Aktionen in einem ganzen Satz. Das Objekt (der Rollenspieler), das die relevanten anderen Objekte kennt, wird zum Ausführenden (bekommt die Methode).
- Klärung des Zusammenspiels von Objekten des Problembereichs mit Objekten des UI mittels Nachrichtenaustausch.

8.8.2 Detailliertere Objekt-Modellierung

Use-Cases liefern Klassen und Methoden. Gleichzeitig mit der Erstellung von Use-Cases sollte man damit beginnen, die Objekte des Problembereichs und des UI auszuarbeiten. Hierbei könnten allgemein folgende Arbeiten anfallen:

- Identifizierung der benötigten Attribute für Methoden
- Identifizierung der benötigten Attribute für das UI
- Aufzeigen der Verbindungen zwischen Objekten (Nachrichtenaustausch)
- Objekte des UI bekommen ihre Inhalte von Objekten des Problembereichs mittels entsprechender Signale und verwalten dann die Darstellung dem Anwender gegenüber mit zusätzlichen Elementen wie Auswahl, Darstellung,...
- Aufzeigen des Informationsflusses von/zum UI

Im Falle von UI-Objekten analysiert man zusätzlich die Verbindungen mit den Objekten des Problembereichs.

9 Abwägung diverser objektorientierter Prinzipien

9.1 Komposition statt Vererbung

Es gibt hervorragende Anwendungsmöglichkeiten für Vererbung, insbesondere im Bereich echter Generalisierung/Spezialisierung. Vorteile:

- Macht eine Abstraktion explizit: Was sind Gemeinsamkeiten, was sind Spezifikationen?
- Zeigt sich gleichermaßen im Objektmodell wie im Quellcode.

Vererbung hat einen inhärenten Hauptnachteil: In der Klassenhierarchie können keine starken Kapselungsanforderungen erfüllt sein. Ändert man eine Oberklasse, können alle Unterklassen eines großen Klassenbaums betroffen sein. Ziel der folgenden Ausführungen ist deshalb die Minimierung des Einsatzes von Vererbung auf das notwendige Maß.

Muss man nach dem Erben viele der geerbten Eigenschaften/Verantwortlichkeiten ändern oder gar löschen, so liegt der Verdacht nahe, dass ineffizient vererbt wird. Eine Lösung könnte die Aufteilung der Vorfahren in solche Klassen sein, von denen nur genau das geerbt wird, was benötigt ist. Dies ist im Falle von Multiple Inheritance einfacher. Steht diese Möglichkeit wie in Java nicht zur Verfügung, so sollte man sich mit Komposition behelfen und das Verhalten mit Interfaces angleichen.

Komposition erkennt man im Objektmodell an Verbindungen, die nicht 1-1 sind (0-1, 2, n, 1-n).

Oftmals wird Vererbung direkt dazu missbraucht, nur Verantwortlichkeit oder Arbeit zu delegieren. Diese Aufgaben lassen sich anderweitig erfüllen und rechtfertigen die daraus resultierenden Kapselungsnachteile nicht.
Auch die Einbettung von Rollen in Klassen wird oftmals mittels Vererbung realisiert. Entwirft man Rollen mittels Klassen, so hat man bei häufigen Rollenwechseln ständig Probleme im Datenaustausch bei der Verwandlung bis hin zum Verlust der Objekthistorie. Werden Rollen mit Komposition modelliert, ist der Rollenwechsel wesentlich einfacher.

Kriterien, wann Vererbung angebracht ist:

- bei „ist eine Art von“ im Gegensatz zu „spielt eine Rolle bei“
- Objekt muss sich nicht in Objekt anderer Klasse verwandeln
- Die Unterklasse stellt eher eine Erweiterung als eine Überschreibung dar
- drückt im Problembereich spezielle Rollen, Transaktionen oder Geräte aus.

Ein Beispiel für die Verwendung von Vererbung bei voliegender Komposition: Anknüpfung von Komposition kann Argument für Vererbung sein

Beispiel Applet (Object - Component, Container, Panel, Applet). Wann will man ein spezielles Applet, wann ein Applet in Komposition

9.2 Einsatz von Interfaces

Über Komposition und Vererbung stehen Objekte in Verbindung mit vielen anderen Objekte. Bei der Wiederverwendung von Klassen (in neuen Zusammenhängen/Projekten) stellt sich die Frage nach elementaren austauschbaren (wiederverwendbaren) Strukturen. Dies sind oftmals nicht einzelne Klassen, sondern ganze Klassenbäume und Klassensammlungen. Löst man die Zusammenarbeit von Objekten durch unmittelbare Codierung der direkten Verbindungen und des direkten Nachrichtenaustausches mit anderen Objekten, so wird eine Wiederverwendung von kleineren Einheiten unmöglich.

Vererbung kann teilweise eine Lösung bei Erweiterungsbedarf darstellen, erfordert jedoch unter Umständen einigen Aufwand, und ist nur im begrenzten Rahmen des sinnvollen Gebrauchs von Vererbung angeraten (vgl. 9.1). Die eigentliche Lösung liegt im Einsatz von Interfaces.

Betrachtung der Objektverbindungen und des Nachrichtenaustausches:
Liegt Objektverbindung/Nachrichtenaustausch nur zu einer bestimmten Klasse vor und bleibt dies auch in Zukunft so?
Liegt Objektverbindung/Nachrichtenaustausch zu vielen verschiedenen Klassen vor, oder könnte die derzeitige Liste in Zukunft stark erweitert werden müssen?

Bei Einsatz von Interfaces im Bereich der Komposition wiegt die Erhöhung der Wiederverwendbarkeit die Nachteile eines leicht komplexeren Design in den meisten Fällen auf.

Interfaces stellen darüber hinaus auch eine weitere Methode dar, die Kapselungsnachteile bei Vererbung zu vermeiden. Wenn der Erbe nur eine bestimmte Schnittstellenklasse erweitern soll, werden Interfaces eine Abhilfe darstellen. In diesem Fall vereinfachen Interfaces sogar den zu betrachtenden Klassenbaum deutlich. („Natürlich sind das alles Ausreden von Sprachdesignern, die sich gegen Multiple Inheritance entschieden haben“)

Man wird nicht jeden Methodenaufruf in ein Interface packen, im folgenden eine Betrachtung einiger sinnvoller Einsatzmöglichkeiten

9.2.1 Wiederholungen

Auffinden von wiederholter oder synonymer Verwendung von nahezu gleichen Methoden oder wiederholt gemeinsam auftretenden Methoden.

Mit einem Interface lässt sich die Übersichtlichkeit der Kommunikationsstruktur im Modell erhöhen.

9.2.2 Proxys

Ein Proxy ist ein Stellvertreterobjekt. Diese werden im Falle von 1-1 Verbindungen zwischen Objekten interessant (also gerade nicht im Falle von Komposition). Ein Proxy beantwortet stellvertretend Fragen aus speziellen Zusammenhängen oder dient als passendere Schnittstelle.

- Man könnte erst den entsprechenden Stellvertreter kontaktieren, um dann von ihm die gewünschten Informationen zu erhalten.
- Natürlich lässt sich dies entsprechend verbergen. Man spricht das „eigentliche“ Objekt direkt an, dieses ist dann dafür verantwortlich, intern die gewünschten Auskünfte vom Stellvertreter zu bekommen.
- Dies führt zu einem gemeinsamen Interface von eigentlichem und Stellvertreterobjekt.

9.2.3 Wiederverwendung - Denken in Analogien

Interfaces bieten neben der Vererbung einen weiteren Mechanismus zur Abstraktion. Die Aufteilung eines Systems in Kategorien verwandter Elemente wird durch den Einsatz von Interfaces um einen wesentlich offeneren Ansatz bereichert: Es geht hierbei nicht mehr um echte „Verwandtschaft“, sondern nur noch um „Kompatibilität“. In der grammatikalischen Analyse entspricht dieser Gesichtspunkt der Verwendung von Adjektiven (kaufbar, mietbar, darstellbar).

Die Mehrfachverwendung von Interfaces kann schon innerhalb einer Applikation (sowohl im Problem- als auch im UI-Bereich) von Interesse sein.
Noch viel interessanter wird es, wenn man die Wiederverwendung von in nahezu allen Applikationen (oder wenigstens in Applikationen aus einem ähnlichen Umfeld) benötigten Komponenten berücksichtigt.

9.2.4 Erweiterung - Die Zukunft

Neben der Wiederverwendung ist auch die Flexibilität des Designs eines der Hauptziele eines jeden guten Entwurfs (damit die Belohnung für gute Arbeit nicht neue Arbeit ist).
Ein Problem aus diesem Zusammenhang besteht darin, wie man die Verbindung eines Objekts mit einem anderen einer bestimmten Schnittstelle genügenden (also eine 1-1 Verbindung) nachträglich zu einer 1-n Verbindung erweitern könnte.
Hier lässt sich einige Übersichtlichkeit über vernünftigen Gebrauch von Naming Conventions im Zusammenhang mit Interfacegruppierung erreichen.

- get/set Methoden stehen für Attributveränderung.
- add/remove Methoden stehen für Objektverbindungen.

Übungsaufgaben

9.1 Kombinieren Sie beide in Kapitel 9 genannten Mechanismen!

10 Werkzeuge mit UML-Unterstützung

Wichtige Aspekte der Unterstützung sind unter anderem:

- Integration der Entwicklung in einen OO-Entwicklungsprozess. Dieser Prozess sollte möglichst variabel auf die Projektgröße und das organisatorische Umfeld zugeschnitten werden können.

- Script language zum Anpassen des Werkzeugs (mit Logbuch, Unterstützung von Modellentwicklung, Einbinden von Erweiterungen der UML, Extraktion von speziellen Informationen aus den Modellen, Hilfe bei der Integration von anderen Werkzeugen)

- Unterstützung von Reverse-Engineering

- Unterstützung des Round-trip-Engineering

- Verbindung der verschiedenen Modele mit den relevanten Dokumenten

- Integration/Anbindung einer Entwicklungsumgebung für die jeweilige Programmiersprache

Hier ein Überblick über einige Softwareprodukte, die Erzeugung, Darstellung, Analyse und Bearbeitung von Analyse- und Entwurfsmodellen auf der Grundlage der UML unterstützen. Im Wesentlichen bieten alle diese Tools Unterstützung für UML-basierte Entwicklung auf der Basis der UML 1.1 mit einigen kleineren Einschränkungen bezüglich der neuesten Standards. Unterstützung für OOA, iterative Entwicklung, Codegenerierung und Round-trip Engineering sowie Skript-Sprachen sind bei allen zu finden. Die Unterschiede liegen im Bereich der unterstützen Entwicklungsumgebungen, Dokumententechnologien, der benutzen Technologien (Repositories) bzw. der unterstützen Ablaufplattformen:

- Innovator von MID GmbH
- Object Team von Cayenne Software Inc.,
- Paradigm Plus von Platinum Technology Inc.
- Rational Rose von Rational Software Corp.
- Rhapsody von i-Logix Inc. (Stärken im Bereich Echtzeitsysteme)
- SELECT Enterprise von SELECT Software Tools Inc.
- StP/UML von Aonix Inc (Unterstützung V-Modell)
- Together/Professional von Object International Software GmbH