

Objektorientierte Analyse und Design

Inhalt

1	Die Planung und Dokumentation von Software	1-1
1.1	Wieso planen wir Software?	1-1
1.2	Wie planen wir Software?	1-2
1.3	Die Methodik der Planung	1-3
1.3.1	Prozeduraler versus objektorientierter Softwareentwurf	1-3
1.3.1.1	Daten- und prozedurale Systeme	1-3
1.3.1.2	Objektorientierte Systeme	1-4
1.3.2	UML - Unified Modeling Language	1-5
1.4	Das Vorgehen	1-6
1.4.1	Das klassische Vorgehensmodell (Wasserfall)	1-6
1.4.2	Das neuzeitliche Modell (Spirale)	1-7
1.4.3	The Rational Unified Process RUP	1-8
1.4.3.1	Begriffe und Definitionen für RUP	1-8
1.4.3.2	Eigenschaften von RUP	1-9
1.4.3.3	Übersicht über RUP	1-12
1.4.3.4	Analyse und Design in RUP	1-16
1.4.3.5	Conclusion	1-19
2	Die grundlegenden OO-Konzepte	2-1
2.1	Klasse und Objekt (Class, Object)	2-1
2.2	Objektdefinition	2-2
2.2.1	Objektstatus (Object-Status)	2-2
2.2.2	Objektverhalten (Object-Behaviour)	2-2
2.2.3	Objektidentität (Object-Identity)	2-3
2.3	Abstraktion (Abstraction)	2-4
2.4	Kapselung (Encapsulation, Hiding)	2-5
2.5	Modularisierung (Modularity)	2-6
2.6	Hierarchie (Hierarchy)	2-7
2.6.1	Allgemeine Definition	2-7
2.6.2	Vererbung (Inheritance)	2-8
2.7	Typenprüfung (Typing)	2-9
2.8	Gleichzeitigkeit (Concurrency)	2-10
2.9	Objektspeicherung (Persistence)	2-11
2.10	Mischung der Konzepte	2-12
3	UML Basics	3-1
3.1	Übersicht	3-1
3.2	Dinge (Things)	3-2
3.2.1	Strukturen (Structures)	3-2

3.2.1.1	Klassen (Classes)	3-2
3.2.1.2	Objekte (Objects)	3-4
3.2.1.3	Schnittstelle (Interface)	3-5
3.2.1.4	Kollaboration (Collaboration)	3-5
3.2.1.5	Anwendungsfall (Use Case)	3-5
3.2.1.6	Aktive Klasse (Active Class)	3-6
3.2.1.7	Komponente (Component)	3-6
3.2.1.8	Knoten (Node)	3-6
3.2.2	Verhalten (Behaviour)	3-7
3.2.2.1	Mitteilung (Message)	3-7
3.2.2.2	Zustand (State)	3-7
3.2.3	Gruppierung (Package)	3-7
3.2.4	Anmerkung (Note, Annotation)	3-8
3.3	Beziehungen (Relations)	3-8
3.3.1	Generalisierung, Spezialisierung (Generalization, Spezialisization)	3-8
3.3.2	Assoziationen (Associations)	3-11
3.3.2.1	Aggregationen (Aggregation)	3-16
3.3.2.2	Komposition (Composition)	3-16
3.3.3	Abhängigkeiten (Dependency)	3-17
3.3.4	Realisierung (Realisation)	3-17
3.4	Diagramme (Diagrams)	3-17
3.4.1	Klassendiagramm (Class Diagram)	3-18
3.4.2	Objektdiagramm (Object Diagram)	3-20
3.4.3	Anwendungsfalldiagramm (Use-Case Diagram)	3-21
3.4.4	Aktivitätendiagramm (Activity Diagram)	3-23
3.4.5	Zustandsdiagramm (State Diagram)	3-25
3.4.6	Sequenzdiagramm (Sequence Diagram)	3-26
3.4.7	Kollaborationsdiagramm (Colaboration Diagram)	3-28
3.4.8	Komponentendiagramm (Component Diagram)	3-29
3.4.9	Verteilungsdiagramm (Deployment Diagram)	3-29
4	Vorgehensschritte des OOA und OOD	4-1
4.1	Positionierung im Gesamtprozess	4-1
4.2	Teilprozesse OOA	4-3
4.2.1	Anwendungsfallanalyse	4-3
4.2.2	Architektur und SEU Evaluation	4-3
4.2.3	Geschäftsklassenanalyse	4-4
4.3	Teilprozesse OOD	4-5
4.3.1	Aktivitätsmodellierung	4-5
4.3.2	Sequenz- oder Kollaborationsmodellierung	4-5
4.3.3	Fachklassenmodellierung	4-6
4.3.4	Komponentenbildung	4-6
4.3.5	Komponentenentwurf	4-7
4.3.6	GUI Design	4-7
5	Analyse	5-1
5.1	Anwendungsfälle (Use case)	5-1
5.1.1	Zusammenhänge	5-1
5.1.2	Vorgehen	5-1
5.1.3	Notation	5-2
5.1.4	Beispiel	5-3
5.1.5	Fallen	5-4
5.1.6	Aufwand	5-4
5.2	Akteure	5-5
5.2.1	Vorgehen	5-5

5.2.2	Notation und Beispiel	5-5
5.2.3	Fallen	5-5
5.2.4	Aufwand	5-5
5.3	Architektur und SEU Evaluation.....	5-6
5.3.1	Mögliche Architekturen	5-7
5.3.1.1	Standalone Application - SDI	5-7
5.3.1.2	Standalone Application - MDI	5-7
5.3.1.3	Client - Server	5-8
5.3.2	Notation und Beispiel	5-9
5.3.3	Fallen	5-9
5.3.4	Aufwand	5-9
5.4	Klassenmodell der Analyse	5-9
5.4.1	Zusammenhänge	5-9
5.4.2	Vorgehen	5-10
5.4.3	Notation und Beispiel	5-10
5.4.4	Fallen	5-10
5.4.5	Aufwand	5-10
6	Das Design	6-1
6.1	Aktivitätenmodell.....	6-1
6.1.1	Vorgehen	6-1
6.1.2	Notation und Beispiel	6-1
6.1.3	Fallen	6-1
6.1.4	Aufwand	6-1
6.2	Klassenmodell des Designs	6-2
6.2.1	Zusammenhänge	6-2
6.2.2	Klassenmerkmale	6-2
6.2.3	Wie identifiziere ich Klassen	6-2
6.2.3.1	Die Textanalyse	6-3
6.2.3.2	Die Realitätsbeobachtung	6-3
6.2.3.3	Benutzersichtanalyse	6-3
6.2.3.4	CRC Methode	6-3
6.2.4	Modellieren oder wie erkenne ich Ordnungsstrukturen	6-4
6.2.4.1	Generalisierung und Spezialisierung	6-4
6.2.4.2	Assoziation.....	6-8
6.2.4.3	Aggregation und Kompositionen	6-9
6.2.5	Notation.....	6-11
6.2.6	Beispiel	6-12
6.2.7	Fallen der Klassenmodellierung.....	6-12
6.2.8	Aufwand	6-12
6.3	Sequenzmodelle	6-13
6.3.1	Vorgehen	6-13
6.3.2	Notation und Beispiel	6-13
6.3.3	Fallen	6-13
6.3.4	Aufwand	6-13
6.4	Kollaborationsmodelle.....	6-14
6.4.1	Vorgehen	6-14
6.4.2	Notation und Beispiel	6-14
6.4.3	Fallen	6-14
6.4.4	Aufwand	6-14
6.5	Komponentenbildung	6-15
6.5.1	Vorgehen	6-15
6.5.2	Notation und Beispiel	6-15
6.5.3	Fallen	6-15
6.5.4	Aufwand	6-15

7	Die Verwendung von Design-Patterns	7-1
7.1	Was sind Design-Patterns	7-1
7.1.1	Architekturmuster	7-2
7.1.2	Entwurfsmuster	7-2
7.1.3	Idiome	7-2
7.1.4	Eine nicht komplette Liste	7-3
7.2	Singleton Idiom	7-4
7.2.1	Ziel	7-4
7.2.2	Anwendung	7-4
7.2.3	Struktur	7-4
7.2.4	Konsequenzen	7-4
7.2.5	Implementation	7-5
7.3	Composite Pattern	7-6
7.3.1	Ziel	7-6
7.3.2	Anwendung	7-6
7.3.3	Struktur	7-6
7.3.4	Konsequenzen	7-7
7.3.5	Implementation	7-7
7.4	Model-View-Controller Pattern	7-12
7.4.1	Ziel	7-12
7.4.2	Anwendung	7-12
7.4.3	Elemente	7-12
7.4.3.1	Model	7-12
7.4.3.2	View	7-13
7.4.3.3	Controller	7-13
7.4.4	Konsequenzen	7-13
7.4.5	Bekannte Variante	7-13
8	Anhang	8-1
8.1	CRC Karten	8-2
8.2	Beschreibung eines Anwendungsfalles	8-3
8.3	Klassenspezifikation / Generalisierungstabelle	8-4
8.4	Begriffe und Abkürzungen	8-5
8.5	Stichwortverzeichnis	8-11

1 Die Planung und Dokumentation von Software

Das oberste Ziel einer Firma die Software herstellt ist es, den richtigen Nutzen für den Kunden zu erzeugen. Die Dokumentation und Planung der Software ist zweitrangig. Leider gibt es immer noch Entwickler, die die Begriffe zweitrangig und überflüssig verwechseln.

1.1 Wieso planen wir Software?

Beim Erstellen von einfachen Exceltabellen wird man kaum jemanden finden, der eine Planung erstellt und diese auch dokumentiert. Entsprechend wird in diesem Fall eine Tabelle als Grundversion erstellt, mit dieser wird ein paar Tage oder Wochen gearbeitet und wenn notwendig werden Anpassungen vorgenommen. Ich habe oft erlebt wie bei diesem Prozess die Endbenutzer, die in diesem Fall zum Entwickler wurden, mit der Zeit die Übersicht verloren haben und die Rechentabellen lieber nicht mehr weiterentwickeln, da sonst die Gefahr drohte, dass das Modell nicht mehr so rechnet wie es sollte.

Dieses einfache Beispiel zeigt, dass es möglich ist, Software ohne Planung zu erstellen. Es zeigt aber auch, dass es gefährlich ist, ohne Planung zu arbeiten, da diese immer auch eine Dokumentation darstellt. Die Dokumentation wird für die Weiterentwicklungen als Grundlage herangezogen.

Wenn Sie nun nicht eine einfache Rechentabelle zusammenstellen sondern die Steuerung einer Kraftwerksanlage programmieren sollen, ist natürlich bereits die Anforderung, alle Notwendigkeiten der künftigen Software zu beschreiben, ein Planungsschritt. Letztendlich unterscheidet sich das Softwareengineering nicht von anderen Ingenieurdisziplinen. Nehmen wir zum Beispiel einen Architekten. Haben Sie schon einmal einen Architekten erlebt, der ein Haus ohne Bauplan baut? Haben Sie schon eine Gebäudesanierung oder einen grossen Umbau eines Geschäftsgebäudes erlebt ohne Planung? Sind Ingenieure "planungswütige" Denker? Natürlich nicht. Es geht bei der Planung im wesentlichen darum ...

- ... eine gemeinsame Sprache aller Beteiligten zu haben und ...
- ... komplexe Verhältnisse einfach darzustellen, sowie ...
- ... die Arbeitsschritte, deren Reihenfolge, Kosten und Benötigte Ressourcen zu erfassen.

Moderne Software wird immer grösser und komplexer. Client / Server Anwendungen, verteilte Komponenten (DCOM und CORBA), eng und weit vernetzte Anwendungen auf verschiedenen Architekturen im Bereich Internet und Intranet. Obwohl Computer heute teilweise um Faktoren von 10^3 schneller sind und ebensoviel mehr Speicher besitzen als noch vor 15 Jahren, sind die Anwendungen nicht schneller geworden sondern oftmals „nur“ umfangreicher.

Das Internet stellt dabei ganz neue Anforderungen an IT Systeme: Jederzeit und an jedem Ort höchste Verfügbarkeit der Daten und Applikationen. Die Erfahrung lehrt uns jedoch häufig: neue Software Releases haben neue Features statt alte Fehler korrigiert, denn die Fehler sind oft nicht die direkte Folge von einfachen Programmierfehlern, sondern entstehen aus unglücklich gewählten Architekturen der Software Systeme. Fehler solcherart sind jedoch nur äusserst schwer zu korrigieren und entstehen in der Frühphase eines Projektes.

Gleichzeitig wird die „Time to market“, also die Zeit welche ein Produkt braucht um von der frühesten Planungsphase bis zur Markteinführung zu kommen, immer kürzer. Dadurch steigt der Druck wieder verwendbare Software-Komponenten zu benutzen. Solche Komponenten entstehen jedoch nicht durch planloses Entwickeln, sondern können nur aufgrund genauester Planung erstellt werden.

Noch heute wird fast immer ausschliesslich, wenn überhaupt methodisch, im gleichen Stil wie vor 15 Jahren entwickelt. Der Einsatz der Methodik hat mit der rasanten Entwicklung im IT Bereich nicht Schritt gehalten. In vielen Firmen gibt es „Hausregeln“ für die Softwareentwicklung. Diese Hausregeln sind fast immer historisch innerhalb der Firma gewachsen und zu nichts kompatibel als zu sich selber. Das Einarbeiten neuer Mitarbeiter ist so sehr aufwendig und damit teuer.

Das Erweitern bestehender Software-Systeme ist oftmals unmöglich oder mit riesigem Aufwand verbunden. Die ursprünglichen Entwickler arbeiten oftmals nicht mehr in der Firma und neue, fremde Entwickler verstehen die

erstellten Dokumentationen, so sie denn überhaupt existieren und dem realen Produkt ähnlich sehen, nicht. In solchen Situationen werde Systeme dann nicht mehr erweitert sondern aufwendig neu erstellt.

Verschiedene Entwickler-Teams, zum Teil sogar innerhalb der gleichen Firma, sprechen nicht die gleiche „Sprache“ und benutzen nicht die gleichen Hilfsmittel und können sich somit nicht verstehen.

Da der Softwareentwickler selten den fachspezifischen Background des potentiellen Benutzers hat, entstehen oft Lösungen welche zwar funktionieren, aber nicht den Benutzeranforderungen gerecht werden. Ein Software-System existiert jedoch nur aus einem Grund: um den Benutzer dienlich zu sein, Zeit und Personalressourcen einzusparen und damit die Entwicklungsaufwände kurz bis mittelfristig wieder einzuspielen.

1.2 Wie planen wir Software?

Das erste Ziel ist, überhaupt Software zu planen. Wie dies geschieht ist eigentlich nicht speziell wichtig. Egal wie gross ein Softwareprojekt ist, es hat noch keinem Entwickler oder Entwicklerteam geschadet, wenn er oder es eine Planung erstellt. Das praktische an der Planung ist, dass automatisch auch eine Dokumentation entsteht. Nehmen wir an, dass eine Software genauso umgesetzt wird, wie sie geplant wurde. Dann ist eigentlich die Dokumentation bereits gemacht bevor die Software steht (ausser vielleicht die Dokumentation für den Endbenutzer).

Der engere Sinn der Planung ist aber natürlich nicht die Dokumentation, sondern folgende zentrale Fragen zu beantworten:

- WAS soll die Software können (Zielplanung)
- WIE soll die Software das Ziel erreichen (Umsetzungsplanung)

In den zahlreichen Projekten, die ich in meiner Praxis bis jetzt miterleben durfte, konnte ich immer wieder erleben, dass der Endbenutzer bei der Frage "Was soll die Software können" zuerst zusammenzuckt, dann einen regelrechten Redeschwall in Richtung Projekt loslässt, und letztendlich doch eigentlich nicht so genau weiss, was er nun eigentlich sagen soll. Bei Projekten für den Eigengebrauch konnte ich das selbe Phänomen bei mir und meinen Entwicklern feststellen.

Diese Erkenntnis der scheinbaren fachlichen Unsicherheit hat etwas zu tun mit Präzision der Ausdrucksfähigkeit und der Fähigkeit komplexe Abläufe zu erkennen und auszudrücken.

Aus obigen, eigentlich sehr natürlichen Fragestellungen und Verhaltensregeln ergibt sich, dass wir Software in der Regel von allgemeinen Fragestellungen mit Verfeinerungstechniken immer präziser beschreiben. Auch hier gilt: Es ist grundsätzlich alles erlaubt, was zum Ziel führt: Grafiken, verbale Beschreibungen, tabellarische Darstellungen, Prototypen und vieles mehr.

Besser ist es aber natürlich für die Planung von IT Entwicklungen eine standardisierte Methode einzusetzen damit auch neue Entwickler in einem Team mit den bereits erarbeiteten Resultaten effektiv umgehen können. Es gibt sehr viele verschiedene Methoden um den Entwicklungsprozess zu planen und zu steuern. Wir werden hier den „Rational Unified Process“ (kurz RUP) benutzen.

1.3 Die Methodik der Planung

Den Darstellungen für die Planung ist allen gemeinsam, dass sie Teile des Endzustandes beschreiben. Daraus können wir ableiten, dass es in nur äusserst einfachen Projekten möglich ist, eine ganzheitliche Darstellung der Lösung zu kreieren. Damit wir grosse Projekte auf verschiedene Mitarbeiter oder Teams aufteilen können, ist es von Vorteil, wenn alle Beteiligten mit der selben Darstellungsart planen. Dies garantiert zwar noch keine eindeutige Schnittstellenbeschreibung zwischen den verschiedenen Teams, erleichtert jedoch die Definition der Schnittstellen und den Umgang miteinander wesentlich, weil auf der selben Basis kommuniziert wird.

Es ist Sache der Projektleitung, die verwendeten Methoden und Darstellungsformen für ein Projekt zu normieren und so Effizienz für das Projekt zu garantieren.

1.3.1 Prozeduraler versus objektorientierter Softwareentwurf

Im Verlauf der Evolution der Softwareentwicklung haben sich auf dem Markt verschiedenste Methoden für Entwicklung von Systemen etabliert. Alle bekannten Methoden neigen dazu, in Ihren Grundzügen als sogenannte

prozedurale oder objektorientierte Methoden definiert zu werden. Währenddem bis in die 80-er Jahre fast ausschliesslich prozedural entwickelt wurde, ermöglichte in jüngster Zeit die Entwicklung von neuen Sprachansätzen (Programmiersprachen) die objektorientierte Systementwicklung.

1.3.1.1 Daten- und prozedurale Systeme

Die daten- oder prozeduralen Entwicklungsmethoden führen zur Trennung der daten- und funktionsorientierten Sicht von Systemen. Zu diesen Methoden gehören:

- Jackson
- Strukturierte Analyse und Design
- Structure Chart
- ...

In Perfektion werden die Modelle nach obenstehenden Methoden soweit herausgebildet, dass reine Daten- und Funktionsmodelle entstehen. Je nach Grösse und Art des Systems werden zum Beispiel für Datenmodelle ERD's (Entity Relationship Diagram) oder Datenflussdiagramme nach Yourdon eingesetzt. Funktionen werden in Form von Datenflussdiagrammen mit Kontrollflüssen, PSPEC's (Programspecification) oder Zustandsdiagrammen herausgeschält.

Das oberste Ziel beim prozessorientierten Softwareentwurf bleibt letztendlich die Wiederverwendbarkeit von Datenaufbau und Funktionalität. Dieses Streben führt in aller Konsequenz zu Funktionsbibliotheken und verallgemeinerten Speicherabbildungen von Daten. In den meisten Fällen bleibt dabei jedoch die allgemeingültige Interaktion zwischen den Daten und den Funktionen auf der Strecke. Diesem Umstand wird in Realität durch Funktionsgruppierung oder allenfalls Parametrisierung begegnet.

Betrachten wir zum Beispiel ein Zeichnungsprogramm. Für die Implementation der verschiedenen geometrischen Figuren erstellen wir eine Funktionsbibliothek. Diese enthält für jede Figur eine Funktion:

- Ellipse(...)
- Rectangle(...)
- Polygon(...)
- Bezier(...)
- ...

Im weiteren werden wir für jede geometrische Figur eine Datenstruktur definieren. Diese könnte wie folgt aussehen:

- struct rect { ... };
- struct ellipse { ... };
- struct polygon { ... };
- ...

Die Anwendung der Funktionen für einzelne Daten könnte nach obigen Definitionen nach dem Muster geschehen:

- Ellipse(struct ellipse, int color); // Zeichnen einer Ellipse
- Rectangle(struct ellipse, int color); // Zeichnen eines Rechteckes
- ...

Zusammengefasst können wir den prozeduralen Softwareentwurf wie folgt definieren:

Systementwicklung durch Verallgemeinerung der Lösungsansätze für Algorithmen und Datenstrukturen.

1.3.1.2 Objektorientierte Systeme

Der objektorientierte Softwareentwurf verfolgt primär die "datentechnische und funktionelle Zusammenführung" der einzelnen Komponenten. Dies äussert sich in einer natürlichen und ganzheitlichen Denkweise des Entwicklers. Der Begriff "natürlich" steht in diesem Zusammenhang für "die real vorkommenden Gegebenheiten", und der Begriff "ganzheitlich" für "die Gegebenheit als Ganzes zu analysieren, die Lösung zu skizzieren und in sich geschlossen umzusetzen".

Für den objektorientierten Softwareentwurf wurden seit Beginn der 80-er Jahre zahlreiche Methoden entwickelt. Dazu gehören:

- Softwareentwicklung nach Booch

- Object Modeling Technique OMT
- Object oriented Software Engineering (OOSE)
- Unified Modeling Language UML

Je nach Methode (Methodiker) wird in den verschiedenen Entwurfsformen eher eine aktionsorientierte oder eine zustandsorientierte Sicht auf die natürliche Gegebenheit bevorzugt. Den Methoden jedoch gemeinsam ist der Versuch der Umsetzung der objektorientierten Konzepte (siehe auch nächstes Kapitel).

Unser Zeichnungsprogramm könnte zum Beispiel den Ansatz pflegen, dass jede geometrische Figur auf einer Zeichnung ein Objekt ist, das selbständig seine Informationen verwaltet und selbständig agieren kann.

In diesem Fall würden wir Typen für Objekte definieren:

- Typ Ellipse{ ... }
- Typ Rectangle{ ... }
- Typ Polygon{ ... }
- Typ Bezier{ ... }

Um mit den Objekten in Interaktion zu treten, müssten wir einzelne Funktionen der Objekte ansprechen können. Da jedoch das Objekt selbständig ist, brauchen wir viel weniger parametrisierte Funktionen:

- Ellipse.Draw();
- Rectangle.Draw()
- ...

Zusammengefasst können wir den objektorientierten Softwareentwurf wie folgt definieren:

Systementwicklung unter Berücksichtigung natürlicher und ganzheitlicher Denkweise für eine optimal abgestimmte Interaktion zwischen den verschiedenen Typen (Klassen) und Objekten.

1.3.2 UML - Unified Modeling Language

Die UML ist eine moderne, normierte Sprache für objektorientierte Entwicklung oder genauer ausgedrückt, für Definition, Konstruktion und Dokumentation. Sie kann in allen Bereichen der Softwareentwicklung (auch realtime) angewendet werden. Neben der eigentlichen Fokussierung der Notation auf objektorientierte Softwareentwicklung ist die Anwendung von UML aber auch für die Beschreibung von Abläufen oder andern Sachverhalten in jedwelcher Richtung möglich (Organisation, Maschinenbau, Elektronik ...).

Die Notation ist nicht nur auf die reine Symbolbeschreibung behaftet. Vielmehr sind auch inhaltliche Beschreibungen vorgegeben, was letztendlich die Präzision und das gegenseitige Verständnis bei Anwendung der UML fördert. Die Beschreibung der UML bildet den eigentlichen Inhalt dieses Kurses und wird deshalb nicht an dieser Stelle eingeführt.

Die Entwicklung der UML geschah in einem evolutionären Prozess. Massgebend beteiligt waren Grady Booch, James Rumbaugh und Ivar Jacobson. Die drei Personen haben im Rahmen ihrer Tätigkeiten eigene objektorientierte Methoden mit jeweils verschiedenen Fokussierungen entwickelt. Ihr Bestreben nach einer allgemeingültigen und im Markt anerkannten Notation hat sie zusammengeführt. Die drei "Amigos" entwickeln seit 1995 die UML 1.0 welche im Januar 1997 der Object Management Group (OMG) zur Standardisierung eingereicht wurde. Im Rahmen der Standardisierung haben alle führenden Hersteller ihre Mitarbeit angekündigt und so wurde es möglich, dass im Sommer 1997 die OMG erstmals eine überarbeitete Version 1.1 der UML als Standard veröffentlichen konnte (siehe auch [1]). Weitere Ergänzungen und Verfeinerungen führten zur heute aktuellen Version 1.3, die im Juni 1999 freigegeben wurde.

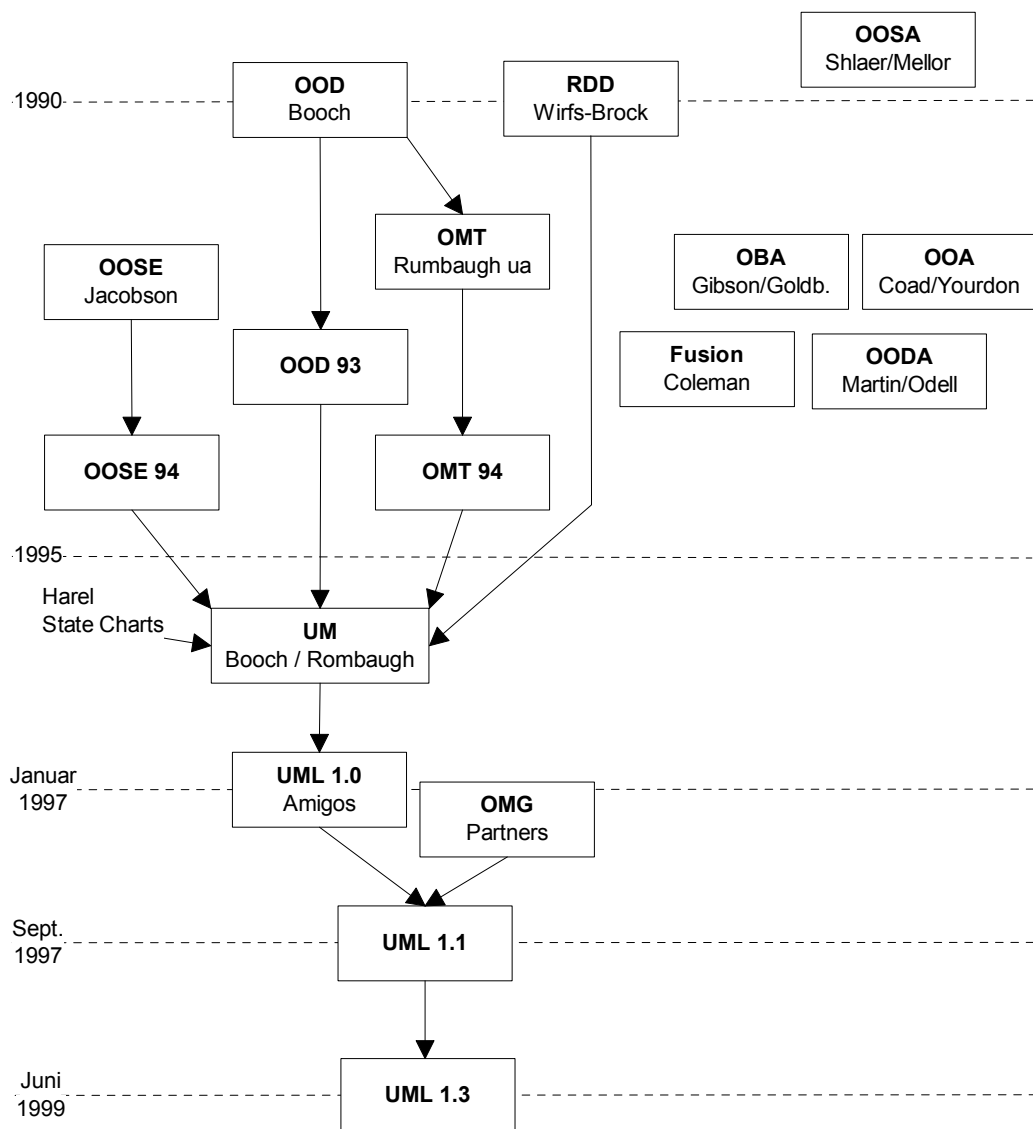


Figure 1-1 Historie von UML

1.4 Das Vorgehen

1.4.1 Das klassische Vorgehensmodell (Wasserfall)

So alt wie die Software Planungsmethodik ist auch die Methodik für das Projektmanagement. Anfänglich und heute noch weit verbreitet ist das Vorgehen nach dem klassischen Projektvorgehen der Ingenieurwissenschaften. Das Grundmuster des sogenannten Wasserfall Vorgehensmodell ist das Denken vom groben Konzept in Richtung verfeinerte Spezifikation. Nach determinativer Spezifikation wird die Software konstruiert, getestet und in Betrieb genommen.

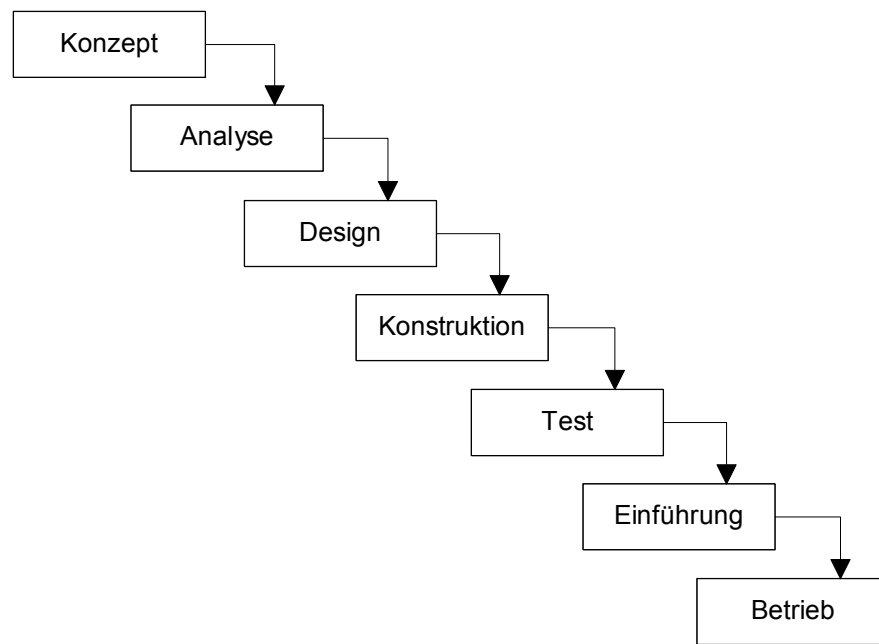


Figure 1-2 Klassisches Vorgehen nach dem Wasserfallmodell

1.4.2 Das neuzeitliche Modell (Spirale)

Das neuzeitliche Modell beginnt ebenfalls mit der Konzeption der Lösung. Innerhalb der Konzeption kann es bereits zu ersten Realisierungen in Form von Prototypen kommen. Die Verfeinerung der Lösung führt über die eigentlichen Analyse- und Designblöcke in die iterative Erstellung der definitiven Lösung.

Die Aufteilung der Betrachtung vom Groben in Richtung der feingegliederten Endlösung geschieht in diesem Modell pro Iterationsschritt und nicht linear top down.

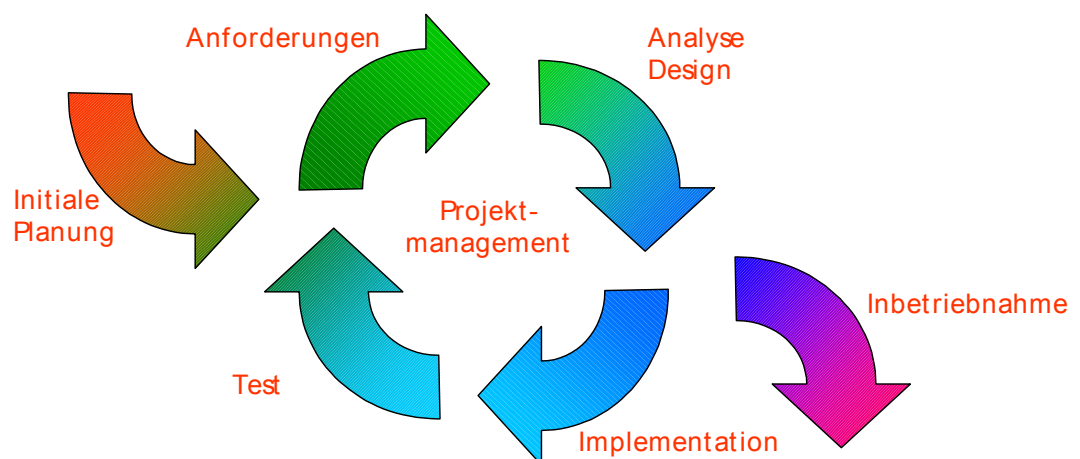


Figure 1-3 Neuzeitliches Vorgehen nach dem Spiralmodell

1.4.3 The Rational Unified Process RUP

Neben der Definition der ausdrückenden Sprache einer Softwareplanung (UML), ist zur Zeit die Entwicklung des übergreifenden Prozesses (eben dem unified process) im Gange. Auch hier sind die Amigos die treibende Kraft und bringen ihre Ideen über die OMG in den Markt.

Der Prozess basiert auf den grundlegenden Gedanken des iterativen Vorgehensmodells, und integriert das Projektmanagement mit dem eigentlichen Vorgehensmodell.

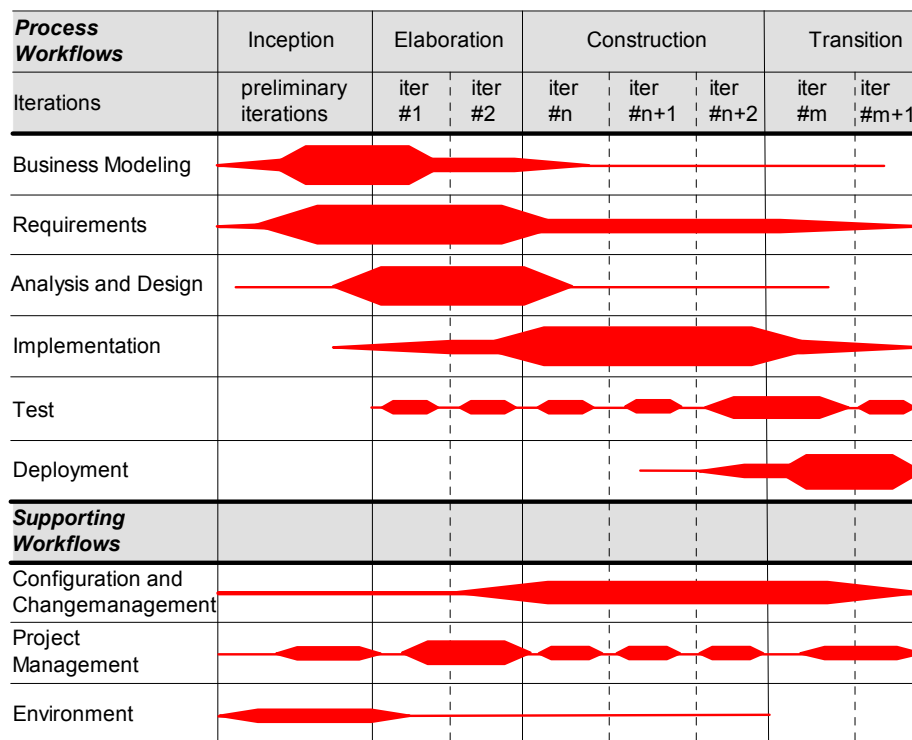


Figure 1-4 Übersicht des Rational Unified Process

Das eigentliche Modellieren der Software geschieht aufgrund der Beschreibung des unified process in der Phase Elaboration (deutsch Ausarbeitung) und zieht sich bis in die Phase Konstruktion.

Die Analyse und das Design sind die Schritte in der Planung, die die Planung am meisten konkretisieren und entsprechend auch die UML anwenden. In dieser Phase entstehen somit die eigentlichen Baupläne für die Software.

1.4.3.1 Begriffe und Definitionen für RUP

Wie viele abstrakte und methodische Prozessbeschreibungen benutzt RUP eine Vielzahl von Begriffen. Die wichtigsten sind:

DEFINITION: Ein **Anwendungsfall** (oder **Use Case**) ist die Beschreibung einer Menge von Aktionen, einschliesslich Varianten, welche ein System ausführt um zu einem beobachtbaren und für einen spezifischen Akteur nützlichen (angestrebten) Resultat zu kommen.

BEMERKUNG 1: ein Anwendungsfall besteht normalerweise wieder aus weiteren Anwendungsfällen → ständige Verfeinerung bis zu einem „vernünftigen“ Level.

BEMERKUNG 2: ein Anwendungsfall beschreibt eine funktionelle Anforderung an ein System aus der Sicht eines spezifischen Akteurs.

BEISPIEL: ein Anwendungsfall kann sein: das Prüfen einer Kreditkarte, das Reservieren eines Sitzplatzes, das Bestellen eines Artikels, ...

DEFINITION: Ein **Akteur** ist jemand oder etwas, welches mit dem System interagiert.

BEISPIEL: ein Akteur kann ein Benutzer, ein anderes Software-System, ein Teil eines Computers, ein Drucker, sein.

DEFINITION: Ein **Artifakt** ist ein Begriff für jede Art von Information erstellt, verändert oder benutzt von den Entwicklern eines Systems.

BEISPIEL: UML Diagramme und dazugehörige Textbeschreibungen, Zeichnungen von GUIs, Geschäftsmodelle, Projektplanung und Prototypen sind Beispiele für Artefakte.

DEFINITION: Ein **Modell** ist eine Abstraktion des Systems aus der Sicht eines spezifischen Projektteilnehmers (Benutzer, Projektmanager, Entwickler, Tester, ...)

BEMERKUNG: Modelle sind Abstraktionen des Systems, welches die Architekten und Entwickler erstellen. Zum Beispiel kümmert sich ein Use Case Modell, welches die funktionalen Ansprüche an das System definiert, nicht um die innere Struktur des Systems.

DEFINITION: Ein **Geschäftsprozess** (oder **Business Process**) ist die Gesamtmenge an Aktivitäten, welche benötigt werden, um ein bestimmtes, für einen individuellen Teilnehmer wertvolles Resultat zu erreichen.

BEMERKUNG: Geschäftsprozesse und Anwendungsfälle sind eng miteinander verknüpft. Ein Geschäftsprozess wird aus seinen Anwendungsfällen konstituiert.

1.4.3.2 Eigenschaften von RUP

Der Rational Unified Process definiert ein Use Case zentrierten Zugang zur Softwareentwicklung. Im Zentrum der Aktivitäten der am Projekt beteiligten Entwickler und Benutzer stehen somit die funktionalen Anforderungen an das System. RUP ist ein Leitfaden, welcher festlegt wer im Team wann was machen soll:

- RUP definiert und steuert die Aktivitäten der einzelnen Entwickler, wie auch des gesamten Teams
- RUP definiert die zu erarbeitenden Artefakte
- RUP definiert Kriterien für das Überwachen und Messen der Projekt Aktivitäten und Resultate

RUP ist ein Framework für den gesamten Softwareentwicklungs-Prozess.

RUP besteht aus einer Menge von definierten Aktivitäten, welche benötigt werden um die Benutzeranforderungen in ein Software-System umzusetzen. Der Prozess arbeitet komponentenbasiert und damit erstellte Software-Systeme bestehen aus Komponenten (= Klassen) mit genau definierten Schnittstellen.

Die Sprache von RUP ist UML. Mit ihr werden die Artefakte erstellt und diese bilden einerseits die Grundlage für die Bildung (Konstruktion) des Systems, andererseits sind sie dessen genaue Dokumentation.

RUP als Prozess für die Durchführung von IT Projekten, bietet sich hauptsächlich für grössere und grosse Projekte an. UML als Methodik und Sprache von RUP dient jedoch auch in kleinen und kleinsten Projekten für die Erstellung von Dokumentationen und Artefakten. Sie ist gerade auch in Projektsitzungen mit Benutzerbeteiligung ein wertvolles Hilfsmittel um handschriftliche Aufzeichnungen anzufertigen.

Durch die strenge Methodik werden Kosten gesenkt und der Entwicklungsprozess beschleunigt. Dabei steigt die Qualität des Produktes und die zukünftige Erweiterbarkeit kann erreicht werden.

RUP kann mit drei Schlagwörtern umrissen werden:

- Anwendungsfall gesteuert
- Architektur zentriert
- iterativ und inkrementell

1.4.3.2.1 RUP ist Anwendungsfall gesteuert

Alle Anwendungsfälle zusammen bilden das sogenannte „Use Case Modell“, welches die kompletten funktionalen Anforderungen an das System beschreibt. Das Use Case Modell sagt somit aus WAS das System machen muss aber nicht WIE es das tut.

Die modellierten Use Cases definieren nicht nur die funktionalen Anforderungen an das System, sondern beeinflussen darüberhinaus auch sehr stark das Design, die Implementierung und die zu bestehenden Tests des Systems.

Use Cases treiben den Entwicklungsprozess an.

Basierend auf den Anwendungsfällen erzeugen die Entwickler eine Serie von Designs und Implementierungs-Modellen welche die Use Cases realisieren sollen. Die Entwickler überprüfen jedes sukzessive Modell auf Konformität zu den Use Cases. Die Tester überprüfen die Implementierung auf die vollständige Erfüllung der Use Cases.

Use Cases sind im Softwareentwicklungsprozess nicht isoliert. Sie werden gemeinsam mit der Architektur der Lösung entwickelt. Use Cases und Architektur beeinflussen sich gegenseitig.

1.4.3.2.2 RUP ist Architektur zentriert

Die Rolle der Architektur in einem IT System ist in ihrem Zweck analog der Rolle, welche die Architektur beim Bau eines Gebäudes spielt. Ausgehend von verschiedenen Perspektiven wird ein Gebäude betrachtet: Struktur, Heizung, Wasserversorgung, Stromversorgung, etc...

Die Architektur erlaubt die Betrachtung eines Gesamtbildes des Gebäudes, bevor die eigentliche Konstruktion beginnt. Demzufolge beinhaltet auch die Software Architektur eines Systems die wichtigen statischen und dynamischen Aspekte des Gesamtgebildes. Die Architektur entsteht dabei aus den Bedürfnissen der Akteure, festgehalten und modelliert in den Use Cases.

Die Architektur ist zusätzlichen Einflüssen unterworfen, welche sich aus der eingesetzten IT Plattform ergeben (Hardware, Betriebssystem, Legacy Systeme, Datenbanken, Netzwerk Protokolle, ...) sowie bestehenden Software Komponenten (Klassenbibliotheken), Performance- und Zuverlässigkeits-Anforderungen, wie auch Konzepten für die Verteilung und Installation des Systems.

Die Architektur ist die technische Sicht auf das Ganze.

Die Architektur fokussiert den Blick auf die signifikanten Aspekte des Designs unter Weglassung der weniger wichtigen Details. Leider ist die Feststellung, welche Aspekte eines Systems signifikant sind oder nicht, eine Frage der Urteilsfähigkeit des Betrachters. Diese Urteilsfähigkeit ergibt sich aus der Erfahrung der am Entwicklungsprozess beteiligten Personen. RUP hilft dem Entwickler dabei, seine Sicht auf die wichtigen Ziele zu lenken: Verständlichkeit, Zukunftsfähigkeit, Erweiterbarkeit, Wiederverwendbarkeit.

Wie sieht die Relation zwischen Use Cases und Architektur aus? Jedes Produkt besitzt Form und Funktion. Nur einer der beiden Aspekte alleine ist nicht genug. Ein gutes Produkt besitzt eine ausbalancierte Beziehung zwischen Form und Funktion: „Form follows Function“.

Im Fall von RUP entspricht die Form der Architektur und die Funktion den Use Cases.

Dabei entsteht oftmals ein klassisches „Huhn und Ei“ Problem. Auf der einen Seite müssen die Use Cases, wenn implementiert, in die Architektur passen, auf der anderen Seite muss die Architektur Raum für die Realisierung aller jetzigen und zukünftigen (sic !) Use Cases lassen.

In der Realität des Prozesses entwickelt man die beiden parallel. Der Softwareentwickler ist auch ein Softwarearchitekt. Die Architektur gibt dem System die Form. Diese Form, die Architektur, muss so beschaffen sein, dass sich das System in die Zukunft entwickeln kann.

Um diese Form zu entwickeln, muss der Entwickler die zentralen Use Cases des Systems finden. Oftmals sind nur 5% - 10% aller Use Cases von zentraler Bedeutung, sie definieren jedoch die Kernfunktionalität des Systems.

1.4.3.2.3 RUP ist iterativ und inkrementell

Die Entwicklung moderner Software Systeme zieht sich oft über viele Monate oder sogar Jahre hinweg. Zugleich ist die IT Branche aber äusserst innovativ und schnelllebig.

Das einzig Konstante ist die ständige Veränderung.

Dies führt zwangsweise zur Strategie des „Divide and Conquer“. Es ist oft zweckmässig grosse Projekte in kleinere Teile und Teilprojekte zu zerlegen. Jeder Teil ist eine Iteration und führt zu einem Inkrement innerhalb des Produktes.

Iterationen sind Schritte im Workflow von RUP.

Inkmente sind Ergänzungen und Verbesserungen des Produktes.

Um möglichst effektiv zu sein, müssen die Iterationen kontrolliert und koordiniert ablaufen.

Das Projektmanagement fällt die Entscheidung WAS zu einem gegebenen Zeitpunkt implementiert werden soll, hauptsächlich aufgrund zweier Kriterien:

- die nächste Iteration befasst sich mit einer Gruppe von Use Cases, welche die Benutzbarkeit respektive Nützlichkeit (Usability) des bisher entwickelten Produktes erweitert und verbessert
- die nächste Iteration befasst sich mit dem wichtigsten Projektrisiko

Sukzessive Iterationen bauen dabei auf die Artefakte, im Zustand ihres definitiven Verbleibens, nach dem Abschluss der vorherigen Iteration. Iterationen sind nicht zwangsweise additiv. Besonders in der Frühphase eines Projektes ersetzen Entwickler ein überflüssiger Design durch ein neues, verbessertes Design. In späteren Phasen sind Iterationen jedoch praktisch immer additiv.

In jeder Iteration bestimmt der Entwickler die relevanten Use Cases, erzeugt ein Design auf der Basis der gewählten Architektur als Leitlinie, implementiert das Design in Komponenten und überprüft die Erfüllung der Use Cases durch die Komponenten.

Erfüllt eine Iteration ihre Ziele, bestimmt man den Inhalt der nächsten Iteration und führt die Entwicklung damit weiter. Erfüllt eine Iteration hingegen ihre Ziele nicht, muss der Entwickler die getroffenen Entscheidungen überdenken und einen neuen Ansatz wählen.

Es ist die Aufgabe der Projektleitung die Iterationen in einer logisch sinnvollen Sequenz auszuführen um das Projektziel mit möglichst kleinem zeitlichen und finanziellen Aufwand erreichen zu können.

Ein kontrollierter, iterativer Prozess hat verschiedene Vorteile:

- Kontrollierte Iteration reduziert das Risiko zeitliche, personelle und finanzielle Ressourcen auf eine einzelne Iteration zu verschwenden. Das Projekt verliert in einem solchen Fall nur die Aufwendungen für diese Iteration, jedoch nicht die Aufwendungen des gesamten Projektes.
- Kontrollierte Iteration reduziert das Risiko, ein Projektziel nicht innerhalb der notwendigen Time to Market zu erreichen. Wenn Projektrisiken früh erkannt werden, können Sie behoben werden, solange das Projektteam noch nicht unter Verzug leidet und sozusagen „frisch“ ist. In klassischen Projekten werden solche Risiken oftmals erst spät im Projektlauf bei den Systemtests erkannt und der zeitliche Aufwand sie zu beheben ist so gross (da oft Architektur und/oder Design Änderungen notwendig sind), dass praktisch immer eine Verzögerung der Produkteinführung eintritt.
- Kontrollierte Iteration erhöht das Tempo der gesamten Entwicklung während der Projektlaufzeit, weil Entwickler effizienter arbeiten in einem klaren und übersichtlichen Terminplan, als in einem ständig sich verändernden Ablauf aufgrund von Verzögerungen.
- Kontrollierte Iteration berücksichtigt eine oftmals ignorierte Tatsache: Benutzerwünsche und die damit zusammenhängenden Anforderungen an ein System sind nicht von Anfang an klar

definierbar. Sie verändern sich typischerweise während der Laufzeit eines Projektes im gleichen Masse, wie die Benutzervertreter von den Entwicklern lernen und Einblicke in die Möglichkeiten der IT erfahren. Durch die Iterationen können die sich verändernden und anpassenden Anforderungen einfacher abgedeckt und in das System implementiert werden.

1.4.3.3 Übersicht über RUP

RUP läuft in einer Serie von Zyklen während der „Lebenszeit“ eines Systems ab. Jeder Zyklus wird mit einem Produkte Release abgeschlossen.

Ein Zyklus in RUP besteht aus vier Phasen: Inception, Elaboration, Construction und Transition.

Jede Phase eines Zyklus unterteilt sich in Iterationen.

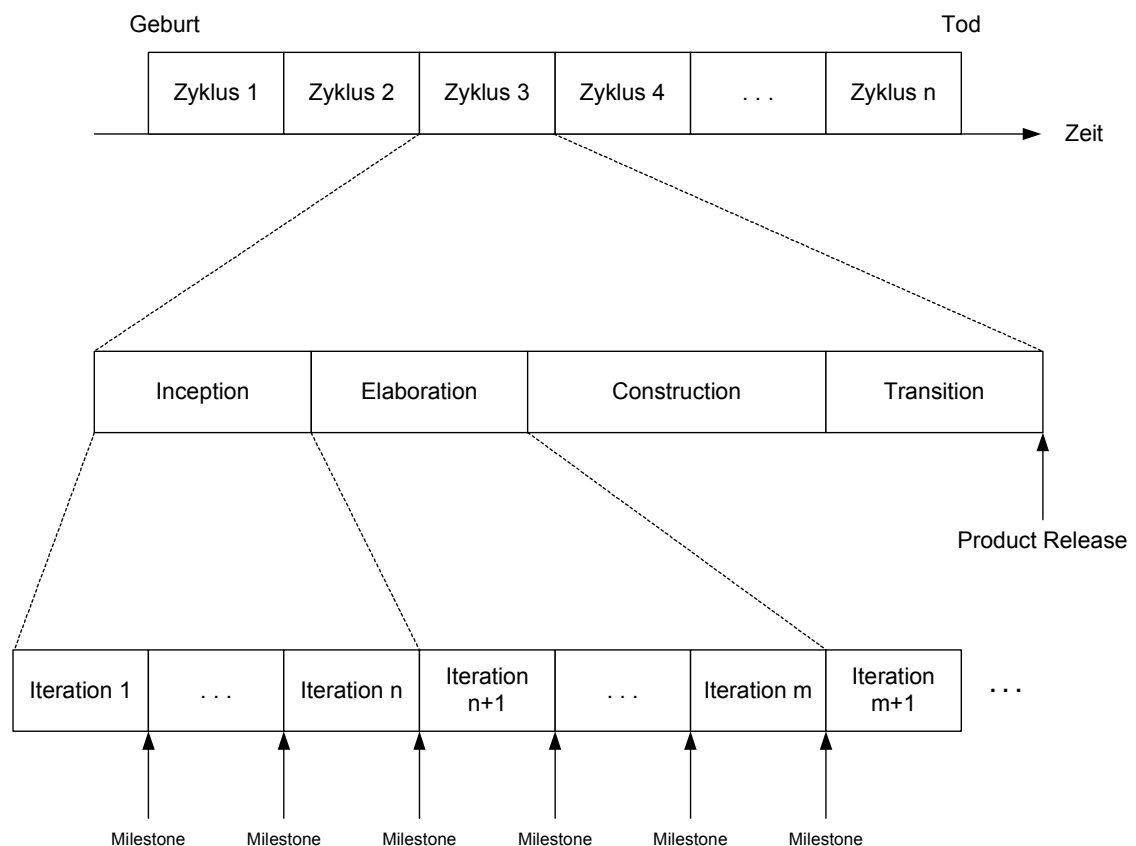


Figure 1-5 Zyklen, Phasen und Iterationen in RUP

Jeder Produkte Release am Ende eines Zyklus umfasst Source Code für die Komponenten welche in ausführbaren Code kompiliert werden können, plus Handbücher und Zusätze wie Installationsprogramme, Datenübernahme und so weiter.

Das „Produkt“ umfasst in der Regel aber mehr als nur ausführbaren Maschinen-Code es beinhaltet auch Einführungsabläufe, Fallback Szenarien, Rahmenorganisationsdokumente und Strukturen.

Auch wenn aus der Sicht des Benutzers (und häufig leider auch aus der Sicht der Entwickler) der ausführbare Code das wichtigste an einem Produkte Release ist, genügt dies alleine nicht. Da sich das gesamte Umfeld eines IT Produktes ständig verändert und entwickelt, verändern sich auch Anforderungen und Möglichkeiten. Neue Hardware, Betriebssysteme, Datenbank- und Umsysteme führen zu ständigen Veränderungen im Umfeld.

Um den nächsten Zyklus in diesem hoch volatilen Umfeld effizient durchlaufen zu können, benötigt der Entwickler alle Artefakte des Produktes aus dem vorherigen Zyklus:

- ein **Anwendungsfall Modell** mit allen Use Cases und deren Relationen zu den Akteuren
- ein **Analyse Modell**, welches einerseits erlaubt die Use Cases zu vertiefen und andererseits das Verhalten des Systems mit einer Menge von Objekten, welche das Verhalten bereitstellen, zu verknüpfen
- ein **Design Modell**, welches einerseits die statische Struktur des Systems in Form von Subsystemen, Klassen und Interfaces definiert, andererseits die Use Cases in Form von Kollaborationen zwischen den Subsystemen, Klassen und Interfaces festlegt
- ein **Implementierungs Modell**, welches die Komponenten umfasst sowie die Klassen auf die Komponenten abbildet
- ein **Deployment Modell**, welches die physikalischen Knoten des Netzwerkes sowie die Abbildung der Komponenten auf die Knoten definiert
- und schliesslich ein **Test Modell**, welches die Test Szenarien beschreibt, um die Use Cases zu überprüfen

All diese Modelle stehen in enger Beziehung zueinander und definieren als Ganzes den aktuellen Stand des Systems.

Process Workflows	Inception	Elaboration		Construction			Transition	
Iterations	preliminary iterations	iter #1	iter #2	iter #n	iter #n+1	iter #n+2	iter #m	iter #m+1
Business Modeling								
Requirements								
Analysis and Design								
Implementation								
Test								
Deployment								
Supporting Workflows								
Configuration and Changemanagement								
Project Management								
Environment								

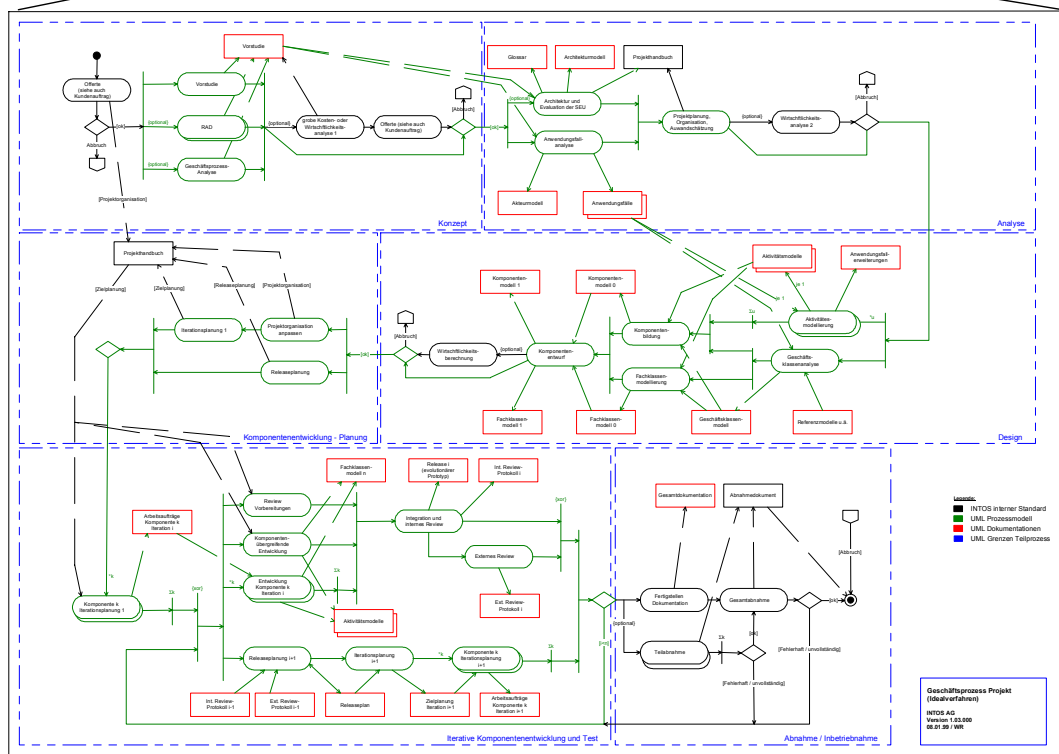


Figure 1-6 Während einer Iteration treten immer wieder gleichartige Aktivitäten auf

Jede Phase resp. jede Iteration, falls eine Phase aus mehr als einer Iteration besteht, endet mit einem Milestone. Jeder Milestone definieren wir durch die Verfügbarkeit einer spezifischen Menge an Artefakten (also bestimmte Modelle und Dokumente sind in einen vordefinierten Zustand gebracht worden).

Ein Milestone dient verschiedenen Zwecken:

- die Projektmanager können aufgrund des Erreichens eines Milestones entscheiden, ob das Projekt in die nächste Phase übertritt oder eine weitere Iteration innerhalb der Phase notwendig ist
- die Projektmitarbeiter wie auch die Projektmanager können anhand der Milestones das Fortschreiten der Arbeiten überwachen

- durch die Beobachtung des geleisteten zeitlichen und personellen Aufwandes für das Erreichen eines Milestones, können Rückschlüsse gemacht werden, welche für andere und zukünftige Projekte sehr wertvoll sind

1.4.3.3.1 Phase Inception

Während der Inception Phase wird eine Vision des Endproduktes erarbeitet. Basis für diese Vision ist das Erstellen und Erarbeiten der Business Cases (Geschäftsfälle).

Die Inception Phase beantwortet vorallem die Fragen:

- Was wird das System primär für jeden der Haupt-Akteurgruppen leisten ?
- Wie kann eine Architektur für das System aussehen ?
- Wie sieht der Overall-Plan für die Entwicklung des Produktes aus ?
- Was für personelle, zeitliche und finanzielle Ressourcen werden notwendig sein, um das Produkt zu erstellen ?

Ein vereinfachtes Use Case Modell beschreibt die primären und wichtigsten Use Cases für die Haupt-Akteure und beantwortet die erste Frage.

In dieser Phase ist die Architektur erst grob und provisorisch erkennbar. Normalerweise bestehen erst Erkenntnisse über die wichtigsten Subsysteme.

Es ist von essentieller Wichtigkeit in dieser Phase die hauptsächlichen Risiken für das Projekt zu erkennen und abzuschätzen. Diese Risiken sind klar zu priorisieren und in ihren Konsequenzen auf den Projektlauf zu taxieren.

Die Elaboration Phase wird im Detail vorbereitet und eine Abschätzung des Gesamtprojektes wird erstellt.

1.4.3.3.2 Phase Elaboration

Während der Elaboration Phase werden die Use Cases im Detail spezifiziert. Das Projektmanagement und die Benutzer einigen sich auf die Use Cases, welche in diesem Zyklus auf jeden Fall implementiert werden müssen.

Das Design der System Architektur wird erstellt., dabei ist die Beziehung zwischen der Architektur des Systems und dem Gesamtsystem besonders wichtig.

Die Architektur stellt sich als spezifische Sicht auf alle erarbeiteten Modelle dar, welche gemeinsam das System definieren. Dies impliziert die Existenz von architekturenspezifischen Ansichten des Use Case, Analyse, Design, Implementierungs und Deployment Modells.

Die Realisierbarkeit der Architektur muss in der Phase Elaboration bewiesen werden.

Die erkannten, kritischen Use Cases werden implementiert und eine Rahmenorganisation für den Betrieb beim Kunden wird erarbeitet.

Am Ende der Elaboration Phase ist der Projektmanager in der Lage die weiteren Aktivitäten zu planen und die benötigten Ressourcen, um das Projekt zu beenden, abzuschätzen.

Die zentrale Frage ist: sind die erkannten Use Cases und die Architektur bereits soweit stabil, und sind die Projekt Risiken erkannt und taxiert so dass verbindliche Zusagen für den restlichen Projektablauf gewährt werden können ?

1.4.3.3.3 Phase Construction

Während der Construction Phase wird das Produkt erstellt. Sinnbildlich wird nun das Fleisch an die Knochen (= Architektur) getan. Die Architektur wird verfeinert und fertiggestellt. Das Projekt benötigt nun den Hauptteil der personellen Ressourcen.

Die Architektur des Systems ist stabil, kleinere Anpassungen und Änderungen sind jedoch noch möglich.

Am Ende dieser Phase deckt das Produkt alle Use Cases ab, welche für diesen Zyklus implementiert werden müssen.

Noch ist das Produkt möglicherweise nicht frei von Fehlern. Während der nächsten Phase werden sicherlich noch weitere Fehler entdeckt. Es dürfen sich jedoch nicht mehr um grundlegende Fehler in der Architektur handeln.

Die zentrale Frage für den Milestone ist: deckt das Produkt die benötigten Use Cases genügend ab ?

1.4.3.3.4 Phase Transition

In dieser Phase reift das Produkt für den Release. Ein Beta Release wird von ausgewählten, erfahrenen (!) Benutzern getestet und Fehler sowie fehlende Funktionalität rapportiert.

Die Fehler und Unterlassungen werden in zwei Kategorien eingeteilt:

- in diesem Zyklus beheben
- im nächsten Zyklus beheben

Die Entwickler beheben die rapportierten Fehler und ergänzen noch fehlende Funktionalität für dieses Release.

In der Transition Phase werden auch die Aktivitäten:

- Herstellung CD Master, Handbücher, ...
- Training der Benutzer
- Training der Mitarbeiter der Rahmenorganisation
- Training der Call Center Mitarbeiter

ausgeführt.

1.4.3.4 Analyse und Design in RUP

Im Rahmen dieser Vorlesung konzentrieren wir uns auf die Aktivitäten im Rahmen der Analyse und dem Design eines IT Systems. In RUP sind diese Aktivitäten hauptsächlich während den Iterationen der Elaboration Phase vorgesehen. Erste Teile der Analyse und abschliessende Arbeiten am Design sind auch bereits während den letzten Iterationen der Inception Phase, respektive den ersten Iterationen der Construction Phase angezeigt.

1.4.3.4.1 Analyse in RUP

In der Analyse untersuchen wir die an das System gestellten Anforderungen. Das WAS wird genauestens umrissen und beschrieben. Diese Beschreibung des WAS hilft bei der Bestimmung der Struktur des ganzen Systems insbesondere seiner Architektur.

Die Analyse untersucht WAS das System leisten soll und hilft bei der Bestimmung von dessen Architektur.

Die Analyse bestimmt die fundierte und stabile Architektur des Systems und trägt zu einem tiefen Verständnis der funktionalen Anforderungen bei.

Als **Input in die Analyse** benötigen wir:

- eine Beschreibung der Geschäftsprozesse
- das Pflichtenheft mit der möglichst exakten Beschreibung der nicht funktionalen Anforderungen an das System (IT Plattform, Performance, Zuverlässigkeit, ...)
- beschreibende Dokumentationen wie: benutzerspezifische Fachausdrücke, Vorgehensweisungen, Diensthandbücher, Geschäftsregeln (Business Rules), und so weiter...
- und vor allem: sehr viele Gespräche mit den Benutzern für das Erfassen der funktionalen Anforderungen (WAS muss das System zu WELCHEM Zeitpunkt mit WELCHEM Resultat erledigen)

Das **Resultat der Analyse** ist das Analyse Modell, welches primär aus den folgenden Elementen besteht:

- die konsolidierten und vollständigen Use Cases und damit die konsolidierten und vollständigen funktionalen Anforderungen für diesen Zyklus (Release)
- optional eine Sammlung von Use Cases, welche erst mit dem nächsten Zyklus implementiert werden; dies ist ein Entscheid der gesamten Projektleitung (IT und Benutzer Seite zusammen)
- eine Beschreibung der generellen Architektur der Lösung sowie der Nachweis von deren Realisierbarkeit (zum Beispiel mit Prototyp Code)

- ein erstes vollständiges, jedoch noch grobes Klassenmodell (sog. Geschäftsklassenmodell) welches vor allem die beteiligten, gemeinsamen Ressourcen im System umschreibt
- die Wahl der SEU (Software Entwicklungsumgebung) inklusive der benutzten Klassenbibliotheken

Die folgenden **Regeln der Analyse** sind für die Erstellung von Use Cases zu beachten:

- Use Cases müssen voneinander soweit wie möglich unabhängig sein
- jeder Use Case soll eine vollständige, in sich geschlossene und möglichst intuitive funktionale Anforderung an das System darstellen
- während der Erhebung der Anforderungen in Form der wichtigsten Use Cases während der Inception Phase, ist es von besonderer Wichtigkeit die „Sprache“ der Benutzer zu verwenden; nun in der Analyse wo sämtliche Use Cases erstellt werden, kann immer mehr die „Sprache“ und Denkweise des Entwicklers benutzt werden

Workers und Artefakte während Analyse:

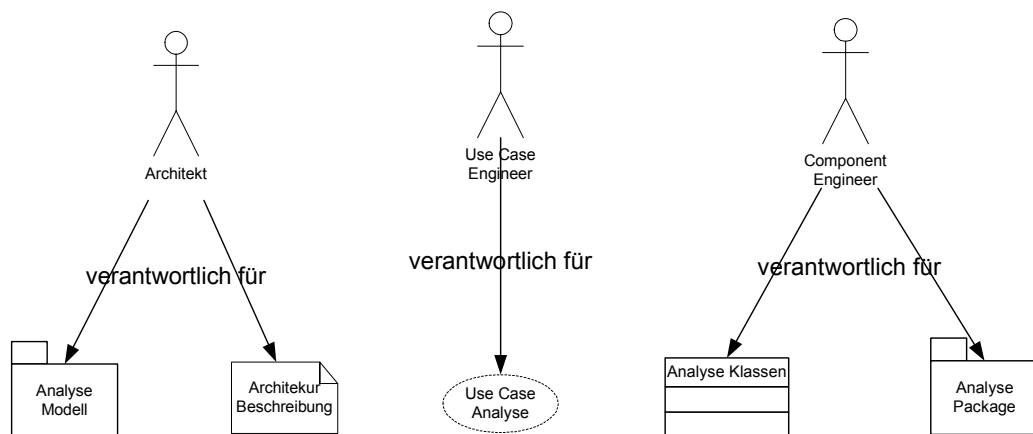


Figure 1-7 Workers und Artefakte während der Analyse

1.4.3.4.2 Design in RUP

Im Design bereiten wir die Konstruktion und Implementierung des Systems in der Construction Phase vor. Es wird genau beschrieben WIE die notwendige Funktionalität, als Resultat aus der Analyse, realisiert wird.

Im Design bestimmen wir WIE die festgestellten Anforderungen aus der Analyse in das funktionelle System umgesetzt werden.

Im Rahmen des Designs erwirbt sich der Entwickler ein tiefes Verständnis der nicht funktionalen Anforderungen und Einschränkungen in Bezug auf:

- die gewählte SEU und der Programmiersprache
- die zu erstellenden Komponenten
- dem System zugrundeliegende Betriebssysteme
- physische und geografische Verteilung der Komponenten zur Laufzeit
- Concurrency (gleichzeitiger Betrieb von Komponenten)
- Interprozess- und Netzwerk-Kommunikation
- benutzte Datenbanktechnologien
- eingesetzter User Interfaces
- Transaktions Management
- und so weiter...

Das Design Modell, als **Resultat des Designs**, liefert gewissermassen den Bauplan für die Konstruktion und besteht aus den Elementen:

- Aktivitätsdiagramme: jeder Use Case wird in einem Aktivitätsdiagramm verfeinert. Im Gegensatz zum Use Case zeigt das Aktivitätsdiagramm dabei Abhängigkeiten, allfällige Nebenläufigkeiten sowie Entscheidungs- und Verzweigungspunkte.
- Sequenzdiagramme und / oder Kollaborationsdiagramme präzisieren die Aktivitätsdiagramme, indem die einzelnen Aktivitäten (Aktionen und Reaktionen) der Objekte in einem Diagramm chronologisch dargestellt (Sequenz) oder die Aktivitäten zwischen den Objekten in der richtigen Reihenfolge bezeichnet (Kollaboration) werden
- Komponentenbildung: die Lösung wird in verschiedene Teile (Komponenten) aufgegliedert. Die Komponenten sind die Einheiten die später gesamtheitlich implementiert, getestet und betrieben werden.
- Fachklassenmodell: die Geschäftsklassen aus der Analyse werden aufgrund der Erkenntnisse aus den anderen Modellen verfeinert, in Beziehung gesetzt und modelliert. Wenn das Projekt mit einer Datenbank arbeitet, soll in dieser Phase die Abbildung in der Datenbank ebenfalls definiert werden. Dabei kann es durchaus zu einer Überführung des objektorientierten Modells in eine relationale Darstellung kommen.
- Deployment Modell: die gefundenen Komponenten werden auf die Knoten im Netzwerk abgebildet. Das Modell beschreibt die geografisch, physische Verteilung des Systems zu Beginn, während der Laufzeit und am Ende eines System-Laufes.
- eventuell GUI Design: besitzt das System ein GUI, so muss dieses jetzt definitiv festgelegt sein. Oftmals wird das GUI Design durch evolutionäre Prototypen (RAD) in der Phase Inception und zu Beginn von Elaboration verfeinert.
- eventuell DB Design (ER Diagramme, DDL Statements, Stored Procedures, ...): benutzt das System für die Persistenz eine Datenbank, so muss zu diesem Zeitpunkt das Design der Datenbank definitiv festgelegt sein. Wird keine OO DBMS eingesetzt, sondern ein relationales DBMS, so muss eine Abbildung der Fachklassen in das DB Design erfolgen.

Workers und Artefakte beim Design:

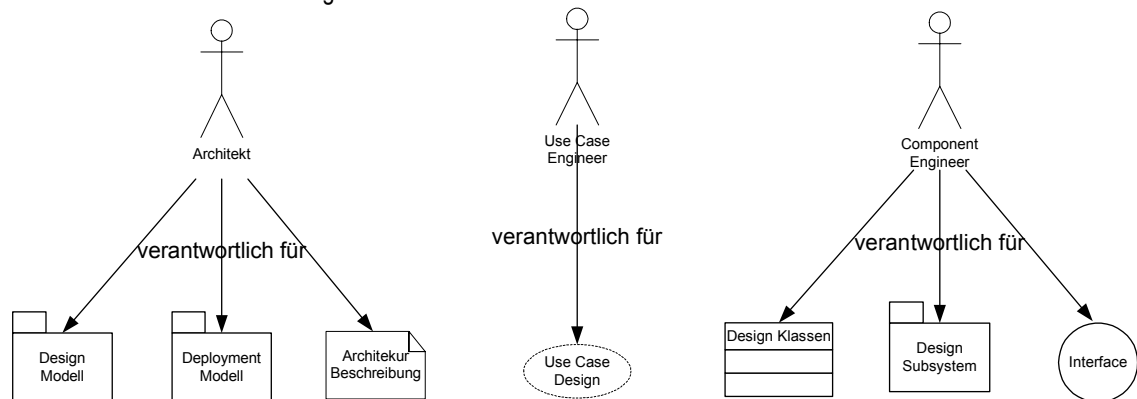


Figure 1-8 Workers und Artefakte beim Design

1.4.3.5 Conclusion

Im Ablauf der Softwareentwicklung nach dem Rational Unified Process RUP in Zyklen mit den Phasen Inception, Elaboration, Construction und Transition, laufen die Aktivitäten der OOA & OOD hauptsächlich in der Phase Elaboration ab.

Dabei ist UML das Werkzeug um die Artefakte zu erzeugen, welche RUP verlangt.

Während der Analyse und dem Design sind insbesondere die Entwickler und Architekten des IT Systems involviert, dies jedoch in engster Zusammenarbeit mit dem Benutzer (Use Cases, GUI Design).

Die Softwareentwicklung ist keine exakte Wissenschaft. RUP und UML stellen eine Framework und eine Methodik zur Verfügung, um die Entwicklung moderner, komplexer IT Systeme zu erleichtern. Automatisieren können und wollen sie den Prozess jedoch nicht.

2 Die grundlegenden OO-Konzepte

Die objektorientierte Technologie basiert auf eigenen Konzepten. Dieses Kapitel führt in diese grundlegenden Konzepte ein und definiert die entsprechenden Begriffe.

2.1 Klasse und Objekt (Class, Object)

Der objektorientierte Entwickler unterscheidet zwischen Klasse und Objekt. Die Klasse definiert dabei die Abstraktion¹ eines Gegenstandes, einer Beziehung oder einer sonstigen Sache ohne reale Manifestation. Das Denken in Klassen erfordert vom Entwickler entsprechendes Abstraktions- und allgemeines Denkvermögen.

Der Begriff „Klasse“ ist dem Begriff „Typ“ ähnlich. Da der Begriff „Typ“ jedoch in der Programmierung bereits vorgelegt ist, und damit einfach die Beschreibung eines Datenformats gemeint wird, ist es besser, den Begriff in der OO-Technik nicht noch einmal zu überlagern.

Die Klasse definiert im Gegensatz zum Typ nicht nur Informationsinhalte sondern auch Funktionalität.



Abbildung 2-1 Eine Abstraktion (Klasse) kann letztendlich nur in Form eines Objektes wiedergegeben werden.

¹ Siehe auch Kapitel „Abstraktion“

2.2 Objektdefinition

Eine Klasse kann detailliert anhand eines realen Objektes derselben studiert werden. Für die Betrachtung von Objekten einer bestimmten Klasse unterscheiden wir deren statisches und dynamisches Verhalten, sowie ihre Identifikation.

2.2.1 Objektstatus (Object-Status)

Das statische Verhalten beschreibt den zum Zeitpunkt der Betrachtung vorhandene Dateninhalt. Dabei kann der Dateninhalt aus beliebigen Datenstrukturen bestehen. In der Literatur wird dazu auch oft der Begriff „Objektstatus“ verwendet. Die Informationen eines Objektes werden mit sogenannten Attributen ausgedrückt.

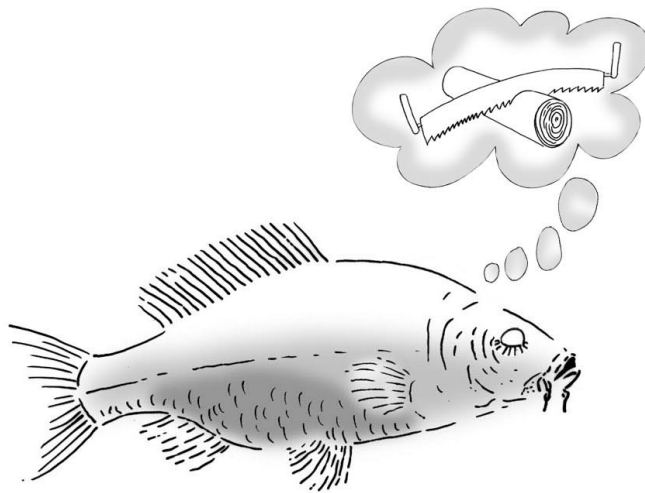


Abbildung 2-2 Ein Objekt verfügt zu jedem Zeitpunkt seines Lebenszyklus über einen Status. Der Status beschreibt den Informationsinhalt des Objektes.

2.2.2 Objektverhalten (Object-Behaviour)

Das dynamische Verhalten beschreibt die vorhandene Funktionalität eines Objektes. Ein Objekt im Sinn der Informatik verfügt im Regelfall über verschiedenste Funktionalitäten, von denen aber zu einem bestimmten Zeitpunkt nur eine aktiv ist (Singletasking). Objekte sind viel schwieriger zu beherrschen, wenn zu einem bestimmten Zeitpunkt gleichzeitig mehrere Funktionen aktiv sind (multitasking)².

Da sich reale Objekte in aller Regel genau umgekehrt verhalten und erst noch viel komplizierter aufgebaut sind, erkennen wir, dass die Informatik einmal mehr nur eine kümmerliche Annäherung dessen ist, was sich in der realen Welt tatsächlich abspielt.

² Siehe auch Kapitel „Gleichzeitigkeit“

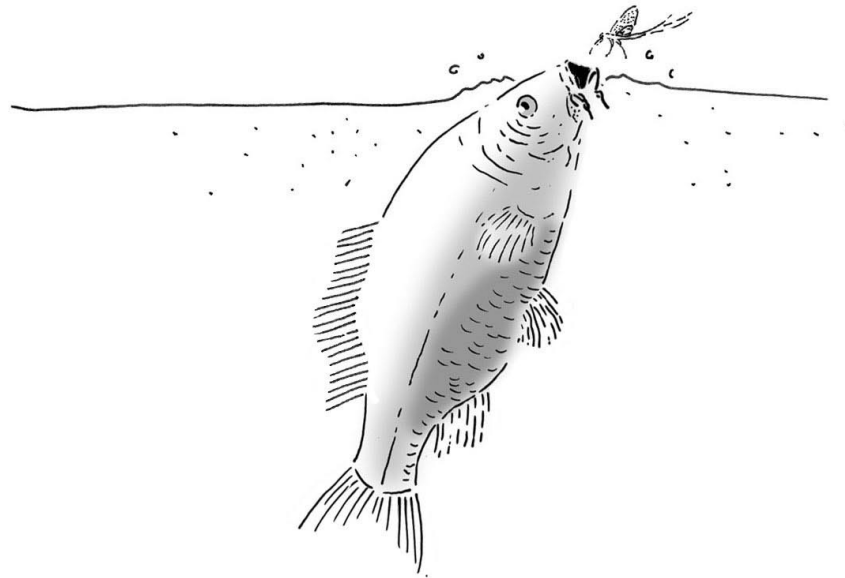


Abbildung 2-3

Ein Objekt führt zu jedem bestimmten Zeitpunkt eine oder mehrere Funktionen aus. Die Summe aller Funktionen beschreibt das Verhalten des Objektes.

2.2.3 Objektidentität (Object-Identity)

Es ist durchaus möglich, dass mehrere Objekte der gleichen Klasse mit dem selben Status existieren. Jedoch selbst dann muss es möglich sein, die einzelnen Objekte voneinander zu unterscheiden. Zu diesem Zweck benutzen wir die Identität eines Objektes. Die einzelnen Attribute der gleichen Objekte können den selben Inhalt haben und trotzdem sind die Objekte zu unterscheiden. In der Informatik führen wir zu diesem Zweck sehr oft eindeutige Numerierungen. Diese entkoppeln die Erkennung der Identität von normalen beschreibenden Elementen.

In einer Datenbank für Personen verwenden wir deshalb kaum den Namen mit dem Vornamen zur Objektkennung, da es möglich ist, dass mehrere Personen mit dem selben Namen und Vornamen existieren.

Beim Design einer Klasse ist der Überlegung nach der Möglichkeit der exakten Identifikation eines Objektes, grösste Beachtung zu schenken. Es ist möglich, dass eine Klasse ohne Attribut für die Objektkennung definiert werden kann. Sollte sich jedoch später herausstellen, dass es trotzdem notwendig sein sollte, die besagte Klasse mit einer Objektkennung zu versehen, kann sich die Unterlassung zum Zeitpunkt des Design durch unverhältnismässig grossen Korrekturaufwand rächen.

Eine eindeutige Objektidentität kann nicht von einem Objekt auf ein anderes übertragen werden. Selbst wenn ein Objekt mit einer Identität X gelöscht wird und anschliessend ein neues Objekt mit der Identität X hergestellt würde, wäre die wahre Identität des neuen Objektes eine andere. Ob diese theoretische Betrachtung jedoch im Informatikalltag eine Rolle spielt, ist der Beurteilung des Entwicklerteams überlassen.

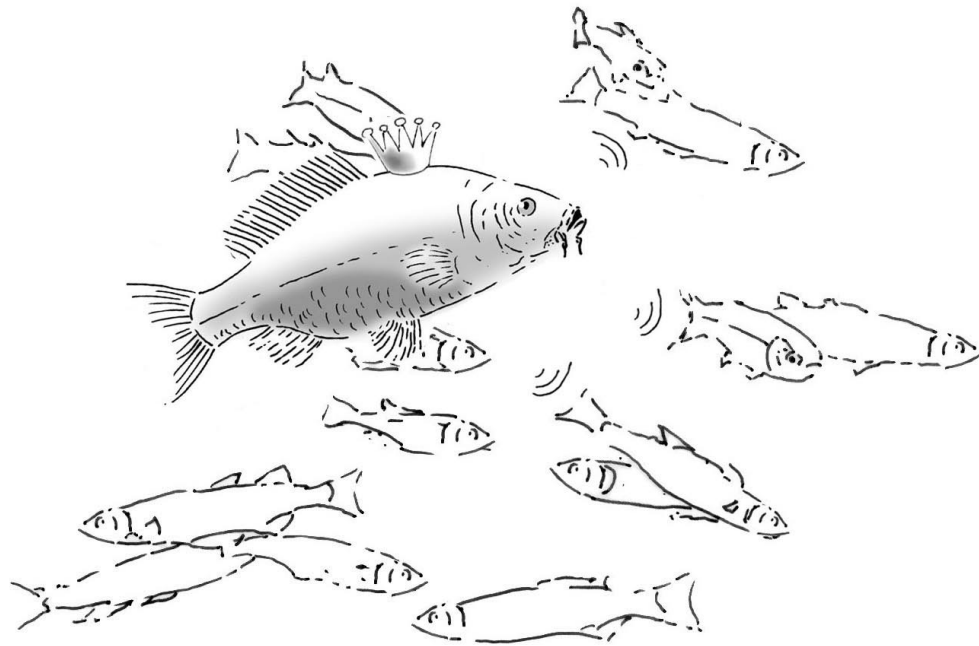


Abbildung 2-4 Ein Objekt verfügt über eine eindeutige Identität. Es ist in Realität nicht möglich die Identität eines Objektes zu wechseln.

2.3 Abstraktion (Abstraction)

Wie wir bei der Begriffsdefinition im Kapitel „Klasse und Objekt“ bereits definiert haben, bildet die Abstraktion (engl. Abstraction) eine Abbildung der Realität, relativ zur Perspektive des Betrachters. Das heisst, dass eine Abstraktion durchaus verschieden interpretiert werden kann (und in Realität auch wird).

Die unterschiedliche Interpretation von Abstraktionen durch verschiedene Personen führt in einem Informatikprojekt immer zu Schwierigkeiten innerhalb des Projektteams. Diese Schwierigkeiten äussern sich in Form von nicht erwünschten Resultaten, gegenseitigem Unverständnis, Gesprächskonflikten oder gar nicht kompatibler Software. Es ist somit wichtig, dass die Abstraktion sehr genau definiert wird. Dazu gehört die Definition der statischen Informationen (Attribute) und des dynamischen Verhaltens (Funktionen oder Methoden).

Die richtige Abstraktion finden wir, indem wir immer wieder die Abstraktion mit den real abzubildenden Sachen vergleichen. Das heisst, dass beispielhafte Objekte von der gefundenen Abstraktion abgeleitet werden und betreffend ihrem Status und Verhalten geprüft werden.

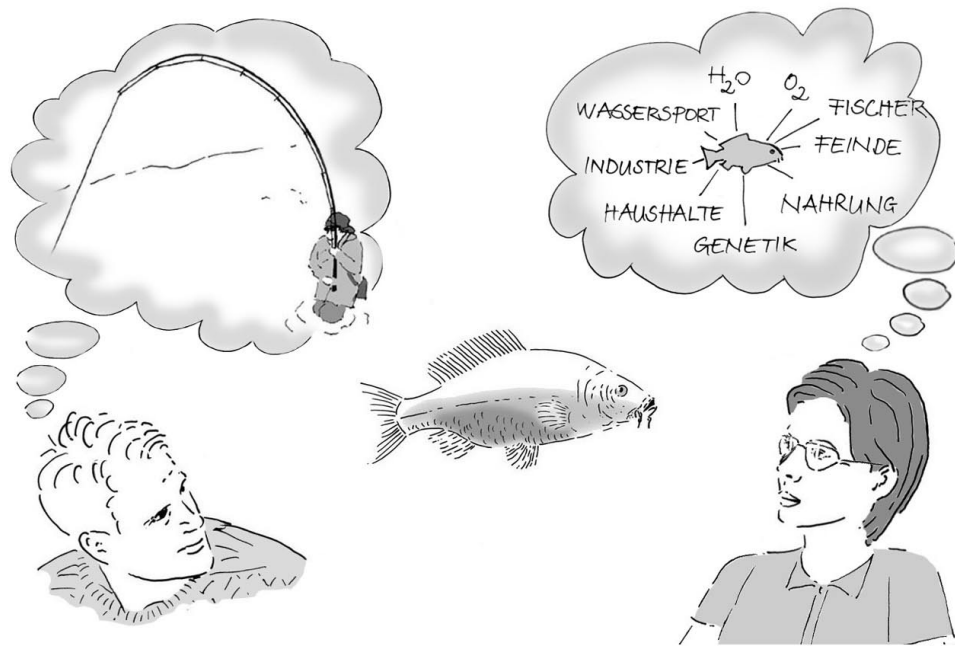


Abbildung 2-5 Eine Abstraktion beschreibt die Charakteristika einer realen Sache relativ zur Perspektive des Betrachters.

2.4 Kapselung (Encapsulation, Hiding)

Ein Objekt kann intern viel komplizierter aufgebaut sein, als sich dies von aussen erkennen lässt. Diesen Umstand nennen wir Kapselung.

Mit der Kapselung (engl. Encapsulation) wird es möglich, interne Details gegen aussen zu schützen, respektive komplizierte Gebilde einfach erscheinen zu lassen. Der Hersteller des Objektes (eigentlich der Klasse) ist mit Vorteil bestrebt, die Kapselung vielfältig und feingliedrig auszugestalten. Dies fördert die vielzitierte Wiederverwendung des Codes, indem die Schnittstellen (die Ansicht von Aussen) nicht geändert werden muss, jedoch die Objekte über ganz andere Funktionsmöglichkeiten verfügen können.

Die Kapselung kann soweit gehen, dass selbst einfachste Attribute nur von internen Funktionen angesprochen, resp. diese Attribute über sogenannte Schnittstellenfunktionen beeinflusst oder abgefragt werden können.

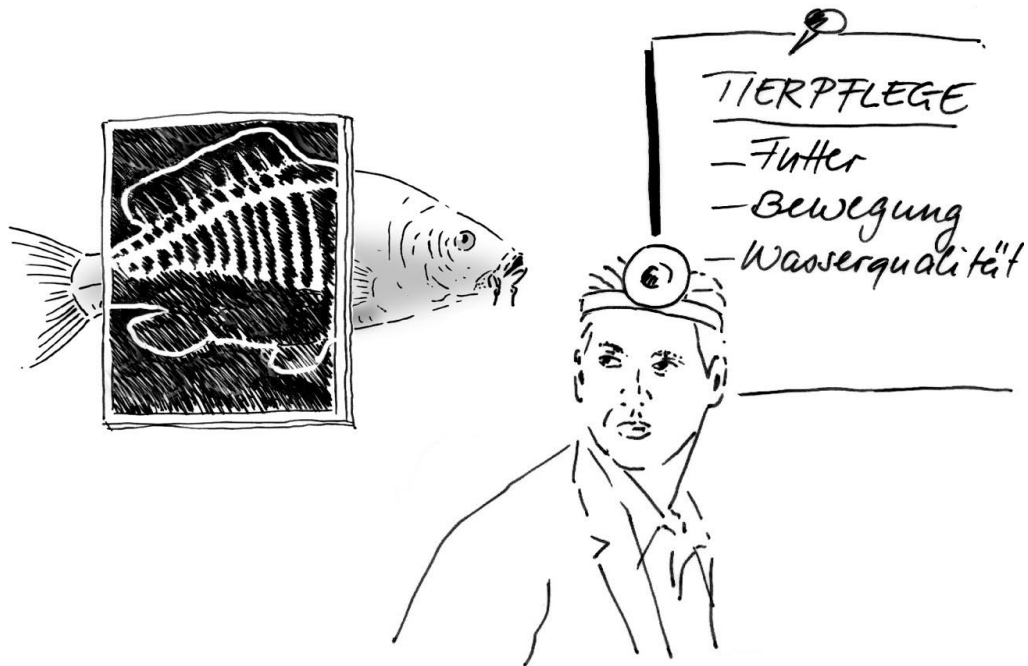


Abbildung 2-6 Die Kapselung beschreibt den inneren Aufbau der Objekte.

2.5 Modularisierung (Modularity)

Die Modularisierung (engl. Modularity) ist in der Informatikindustrie durchaus kein neues Konzept. Sie dient auch in der OOT zur Beschreibung der Aufteilung einzel verwendbarer Bauteile. In Anlehnung an die Hardwareindustrie sprechen wir in Bezug auf Modularisierung auch von der Herstellung von Software-IC's. Im Gegensatz zur prozeduralen Softwaretechnik, welche die Modularisierung auf Funktionsstufe durchführt, definieren wir in der OOT die Modularisierung auf Klassen- resp. Objektstufe.

Die Modularisierung kennt verschiedene Arten der Implementation. Die einfachste Art ist die Modularisierung von Sourcecode. Hierbei geht es darum, dass die Wiederverwendung der einzelnen Module auf Sourcecodelevel statisch herbeigeführt wird. Diese Anwendung ist banal und alltäglich.

Die heute modernste Art der Modularisierung finden wir in der Komponententechnik, welche die Modularisierung soweit treibt, dass die einzelnen Module (Komponenten) unabhängig von einander fertig entwickelt werden, und zur Laufzeit über Grenzen eines Computersystems hinweg miteinander zusammenarbeiten können (Componentware). Dabei ist es sogar möglich, dass die einzelnen involvierten Systeme auf unterschiedlichen Rechner- und Betriebssystemarchitekturen basieren.

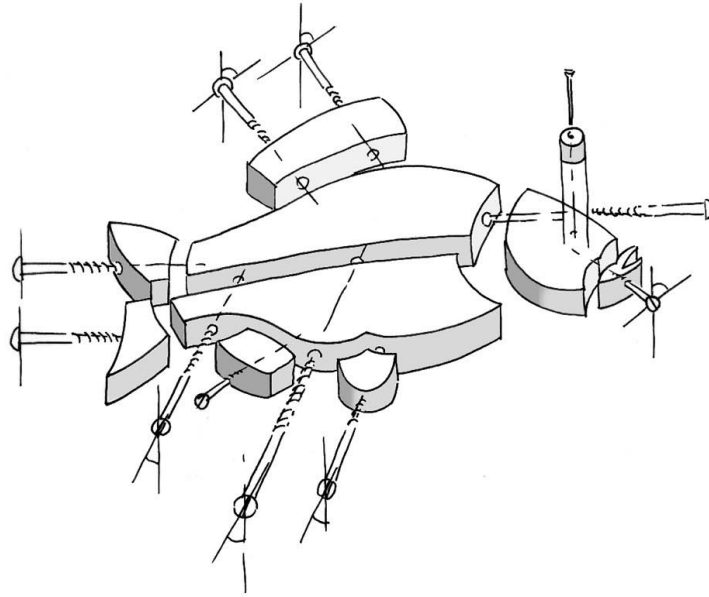


Abbildung 2-7 Die Modularisierung beschreibt die Aufteilung in einzel verwendbare Bauteile.

2.6 Hierarchie (Hierarchy)

2.6.1 Allgemeine Definition

Die Hierarchie (engl. Hierarchy) beschreibt die Abhängigkeit der Klassen untereinander. Gemeinsamkeiten verschiedener Unterklassen werden in sogenannten Oberklassen zusammengefasst. Den Prozess der Begriffsbildung für eine Oberklasse zu einer oder mehreren Unterklassen nennen wir Generalisierung. Der umgekehrte Prozess der Begriffsbildung von Unterklassen zu einer bestimmten Oberklasse nennen wir Spezialisierung.

In der Regel verfügt eine Unterklasse über mehr Informationen und Funktionen als eine Oberklasse.

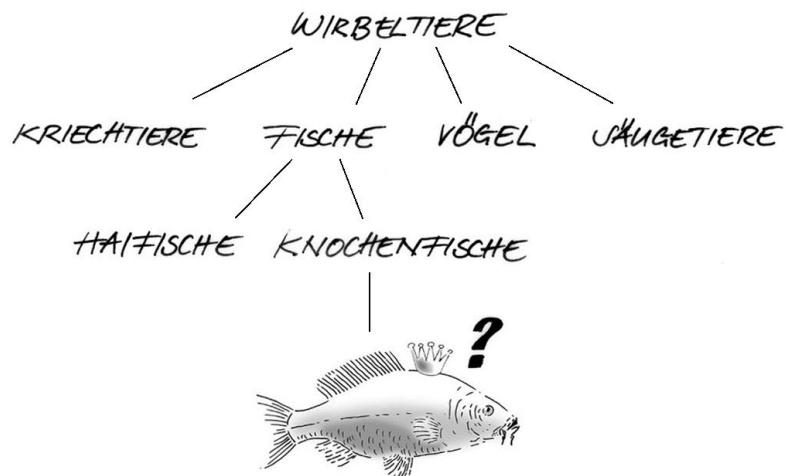


Abbildung 2-8 Die Hierarchie beschreibt die Abhängigkeit der Klassen untereinander.

2.6.2 Vererbung (Inheritance)

Die Bildung von Hierarchien hat in der OOT die Vererbung (engl. Inheritance) zur Folge. Beim Prozess der Vererbung werden Informationen und Funktionen von der Oberklasse an die Unterklasse weitergegeben. Je nach Sprachimplementierung ist es dabei möglich, die genauen Modalitäten der Vererbung zu beeinflussen. So ist es in C++ zum Beispiel möglich, neue Klassen durch Vererbung von einer oder mehreren Oberklassen herbeizuführen, oder die Weitergabe von Attributen und Funktionen an die Unterklasse zu beeinflussen.

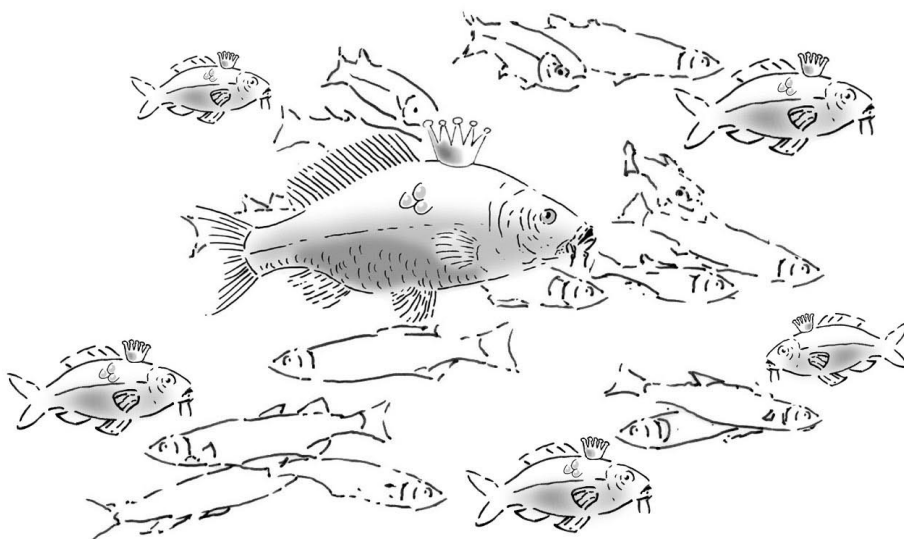


Abbildung 2-9 Die Vererbung beschreibt die Weitergabe von Status und Verhalten von Oberklasse(n) zu Unterklasse(n).

2.7 Typenprüfung (Typing)

Die Typenprüfung (engl. Typing) verhindert die unbewusste Überführung eines Objektes einer Klasse in ein Objekt einer anderen Klasse. Der Ausspruch aus dem Volksmund „ein Fisch ist kein Vogel“ zeigt sehr typisch was hier gemeint ist.

Die unbewusste Typenüberführung birgt die Gefahr der falschen Behandlung des neuen Objektes (zum Beispiel Abfrage von Informationen die es nicht besitzt), und kann so zu echten betrieblichen Problemen führen (Softwareabstürze).

Was im Bereich der Typenüberführung erlaubt ist und was nicht, muss den jeweiligen Sprachimplementationen entnommen werden. In aller Regel sind Möglichkeiten der Objektüberführung innerhalb einer gültigen, definierten Hierarchie in Richtung Oberklasse möglich, jedoch nicht umgekehrt. Die Überführung der diskreten Informationstypen (Integer, Float ...) unterliegt meistens besonderen Regeln.

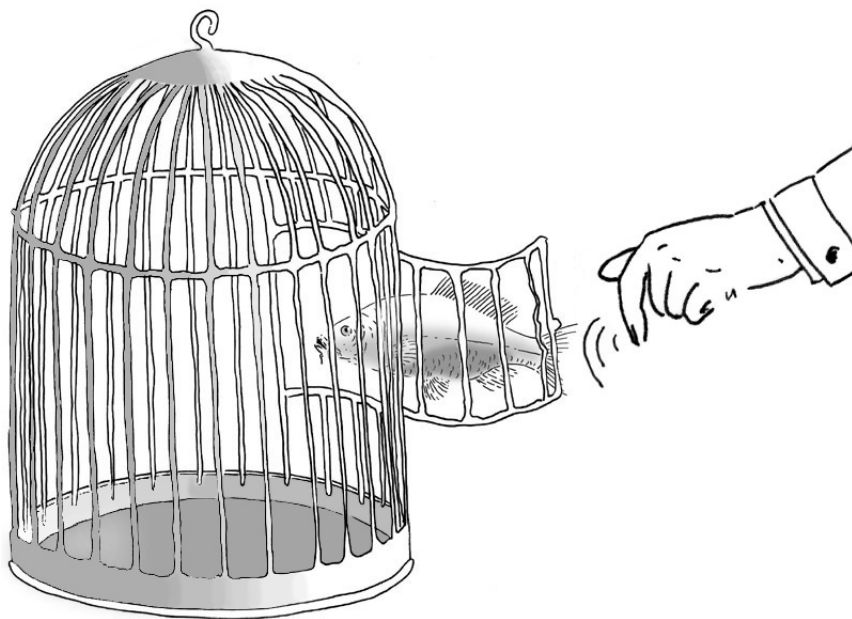


Abbildung 2-10 Die Typenprüfung verhindert die unbewusste Überführung einer Information vom einen Typ in den andern.

2.8 Gleichzeitigkeit (Concurrency)

Die Gleichzeitigkeit (engl. Concurrency) definiert die Möglichkeit der Verrichtung mehrerer Aktionen zu einem bestimmten Zeitpunkt. Die OOT beinhaltet Möglichkeiten zum Entwurf von Gleichzeitigkeit. Die Möglichkeiten der Implementation der Gleichzeitigkeit ist jedoch wiederum abhängig von der verwendeten Sprache und dem verwendeten Betriebssystem.

Die Gleichzeitigkeit ist wohl das Gebiet der Informatikindustrie, welches die „kümmerlichste“ Umsetzung der Realität und gleichzeitig die grösste Herausforderung darstellt. Obschon die heutigen Computersysteme sehr ausgeklügelt sind und vorallem sehr schnell rechnen, findet die Implementierung der Gleichzeitigkeit immer wieder ihre Grenzen an der Kapazität und Komplexität der realen Welt.

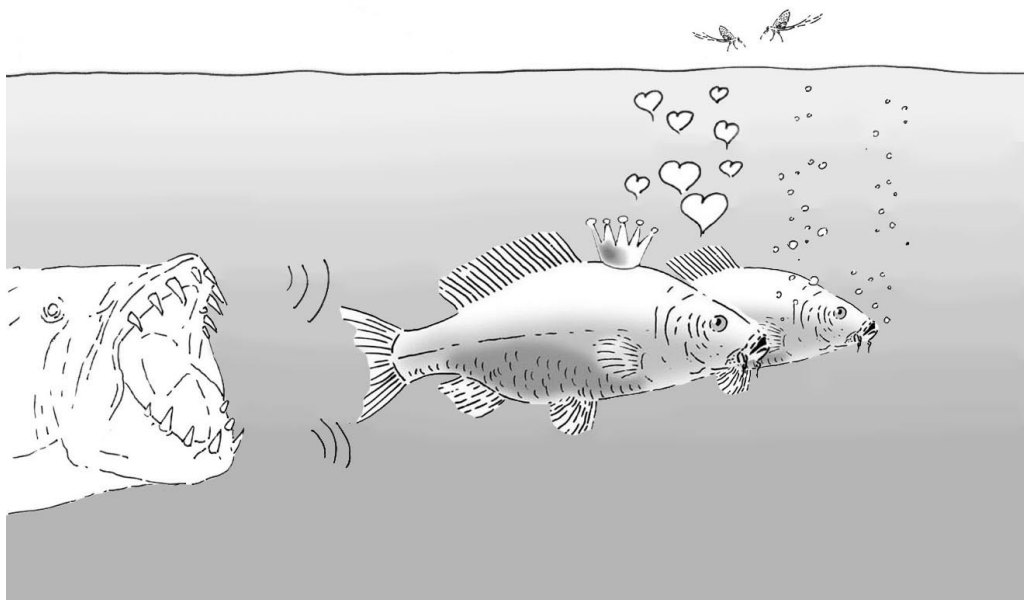


Abbildung 2-11

Die Gleichzeitigkeit beschreibt die Möglichkeit der Verrichtung von mehreren Aktionen zu einem bestimmten Zeitpunkt

2.9 Objektspeicherung (Persistence)

Die Objektspeicherung (engl. Persistenz) beschäftigt sich mit dem Umstand der Lagerung der Objekteigenschaften (Status und Identität) und der Beziehungen zwischen einzelnen Objekten.

Bemerkenswert ist in diesem Gebiet der OOT, insbesondere die Umsetzung der verschiedenen Algorithmen für die Speicherung von Beziehungen zwischen Objekten. So können zum Beispiel die Verknüpfungen von Objekten innerhalb des Arbeitsspeichers eines Rechners über Zeiger führen, währenddem die Verknüpfungen auf einem externen Datenträger über die Objektidentifikation geführt werden.

Ebenfalls wichtig ist die Erkenntnis, dass ein Objekt einmal auf einen externen Datenträger übertragen, als Objekt einer bestimmten Klasse wieder erkannt werden muss, da zur Laufzeit die Verbindung der Information auf dem Datenträger mit der Funktionalität über die Klassenangaben führen.

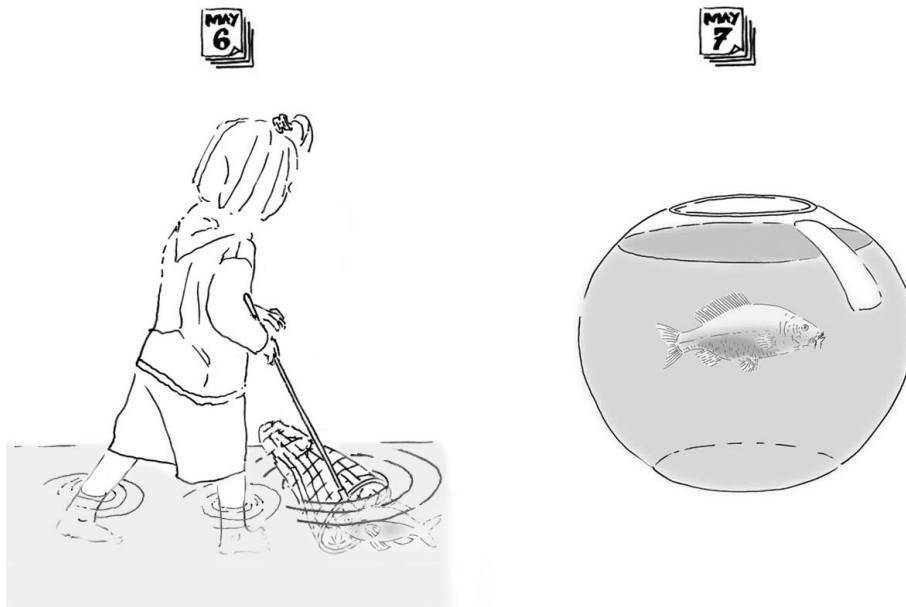


Abbildung 2-12 Die Persistenz beschäftigt sich mit der Erhaltung der Eigenschaften und Beziehungen der Objekte über Raum und Zeit.

2.10 Mischung der Konzepte

Die in diesem Kapitel dargestellten grundlegenden Konzepte werden in der Praxis natürlich nicht in sich geschlossen sondern nur in sich vernetzt angewendet. Dabei spielen einige Konzepte eine sogenannte Primärrolle, das heisst, dass sie in jedem OOT Projekt mit Sicherheit angewendet werden.

Dazu gehören:

- Abstraktion
- Kapselung
- Modularisierung
- Hierarchie

Die weitergehenden Konzepte ...

- ... Typenprüfung
- ... Gleichzeitigkeit und
- ... Objektspeicherung

sind ergänzende Konzepte, die ein reales Projekt abrunden, jedoch nicht überall vorhanden sind.

Die Kunst der objektorientierten Analyse und des objektorientierten Design besteht nun darin, obige Konzepte auf eine reale Problemstellung anzuwenden und diese in einer „künstlichen“ Ausdruckswelt zu beschreiben, so dass der Programmierer anschliessend die Lösung erstellen kann.

Zu diesem Zweck bedient sich der Analyse- und Designspezialist einer speziellen Sprache (UML), die im Rahmen dieses Kurses eingeführt wird.

3 UML Basics

3.1 Übersicht

Die UML ist eine stark normierte auf wenigen Grundelementen aufbauende Sprache. Die Normierung wurde bewusst streng gewählt, damit eine optimale Toolunterstützung für Engineering als auch reverse Engineering erreicht werden kann¹.

Die Unterteilung der Sprache in ihre verschiedenen Elemente lässt sich wie folgt darstellen:

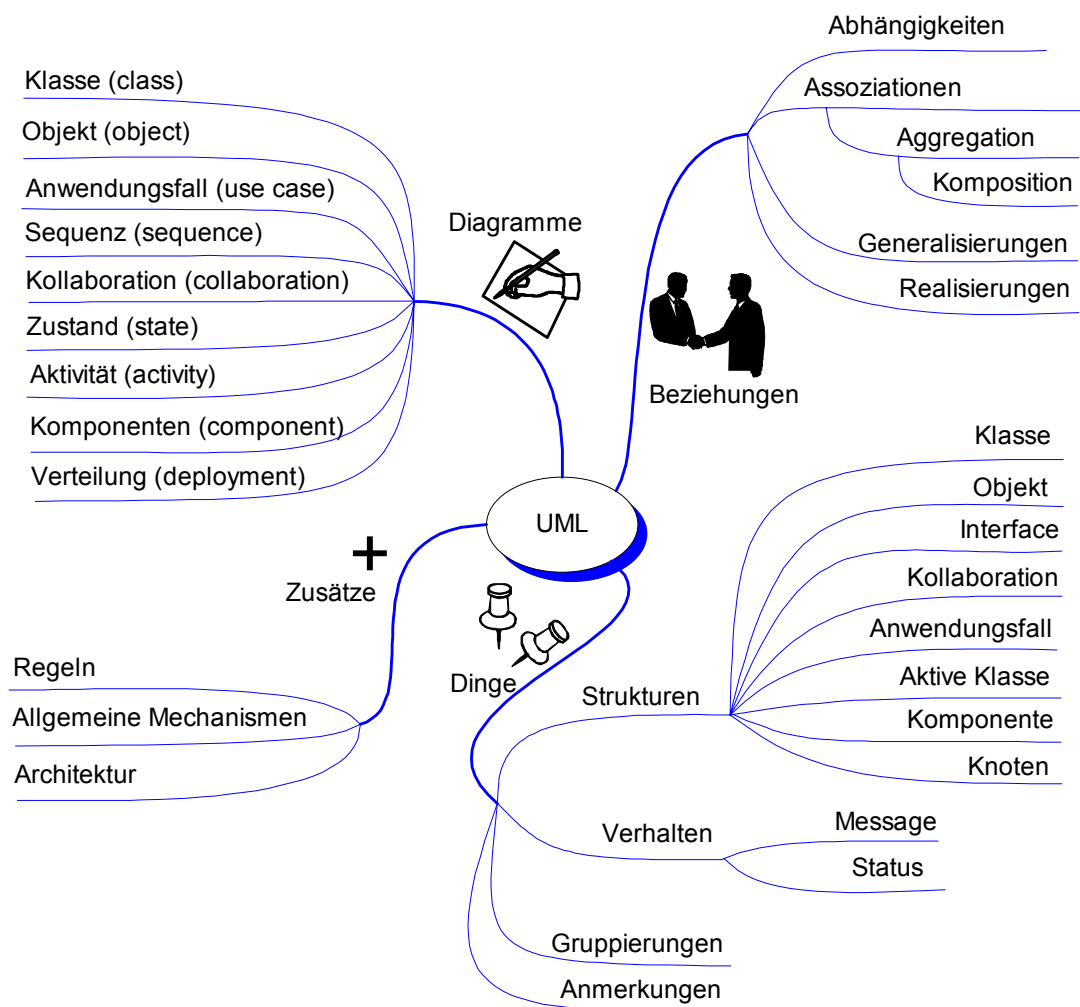


Abbildung 3-1 Übersicht UML

¹ UML ist mittels Stereotypen erweiterbar und anpassbar.

3.2 Dinge (Things)

Dinge² bilden die Hauptelemente der Modelle in der UML. Sie verkörpern mit ihrer symbolischen Präsenz im Modell real existente Sachen mit ihren Stati und Verhalten.

3.2.1 Strukturen (Structures)

Strukturelemente verkörpern die statischen Dinge in einem Modell. Je nach Element werden dabei logische oder physische Gegebenheiten modelliert. Wir unterscheiden folgende Strukturelemente:

- die Klasse
- das Interface
- die Kollaboration
- der Anwendungsfall
- die aktive Klasse
- die Komponente
- der Knoten

3.2.1.1 Klassen (Classes)

Das wohl bekannteste Strukturelement ist die Klasse, welche Objekte gleicher "Bauart", das heisst mit gleichen Attributen, Operationen und Beziehungen verkörpert. Das Grundsymbol für die Darstellung einer Klasse ist ein Rechteck. Je nach Kontext und Darstellungstiefe können innerhalb des Rechtecks mehr oder weniger Informationen dargestellt werden.

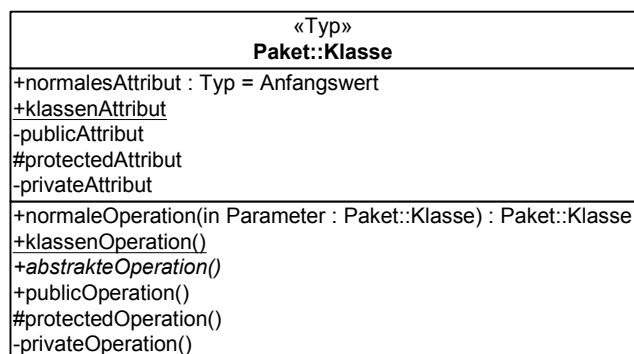


Abbildung 3-2 Allgemeine Darstellung einer Klasse.

² Der Begriff "Dinge" wurde bewusst aus [1] wörtlich übersetzt übernommen. Andere Autoren bezeichnen die Dinge als Elemente, Hauptelemente oder Beschreibungselemente.

In der oben abgebildeten Definition einer Klasse sind folgende detaillierten Informationen einsetzbar:

Information	Bedeutung
Stereotyp	Der Stereotyp ist ein Name für einen neuen Typ eines Modellelementes, der einem vordefinierten Element eine spezielle Semantik zuordnet. In der UML sind bestimmte Stereotypen bereits definiert. Beispiel: Akteur.
Paket	Gruppe zu der die Klasse gehört.
Klasse	Eigentlicher Name des "Dinges".
Eigenschaftswerte	Allgemeine Eigenschaften der Klasse. Beispiel abstract.
Attribute	Informationen der Klasse.
Operationen	Funktionalität der Klasse (auch Methoden).

Tabelle 3-1 Elemente der Klassendefinition.

Sowohl bei den Attributen als auch bei den Operationen kann die Sichtbarkeit "Visibility" definiert werden. Es kommen dabei folgende Zeichen (Sichtbarkeitssymbole) zur Anwendung:

Symbol	Bedeutung
\	optionales Symbol zur Kennzeichnung einer Abhängigkeit von andern Elementen der Klasse (innere Abhängigkeit)
+	optionales Symbol zur Kennzeichnung des öffentlichen Zugriffs (von Aussen). Wenn kein Symbol definiert ist, wird diese Sichtbarkeit als standard angenommen (public).
#	Symbol zur Kennzeichnung des geschützten Zugriffs über Vererbung (protected).
-	Symbol zur Kennzeichnung des rein inneren Zugriffs (privat)

Tabelle 3-2 Sichtbarkeitssymbole

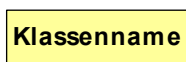


Abbildung 3-3 Die einfachste Form der Darstellung einer Klasse.

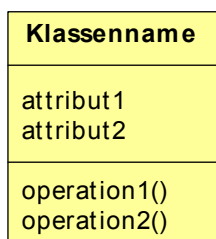


Abbildung 3-4 Darstellung einer Klasse mit Angabe von Attributen und Operationen.

Für die Beschreibung von Klassenelementen gelten folgende Regeln³:

Der Klassenname wird in Fettschrift zentriert geschrieben.

Der Klassenname beginnt mit einem Grossbuchstaben.

Allfällige Namen des Stereotyps werden in "<< >>" Zeichen oberhalb des Klassennamens geschrieben.

Attribute und Operationen werden linksbündig in normaler Schrift geschrieben.

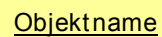
Namen von Attributen und Operationen werden in Kleinbuchstaben begonnen.

Namen von abstrakten Klassen oder von abstrakten Operationen werden in Schrägschrift geschrieben.

Die Attribute und Operationen werden nur angezeigt wenn nötig.

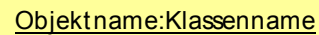
3.2.1.2 Objekte (Objects)

Das Objekt ist die reale Manifestation einer Klasse. Entsprechend wird für das Objekt die selbe Symbolik wie für die Klasse verwendet. Zur Unterscheidung einer Klasse und Ihrer Objekte wird der Name unterstrichen dargestellt.



Objektname

Abbildung 3-5 Darstellung eines benannten Objektes.



Objektname.Klassenname

Abbildung 3-6 Darstellung eines benannten Objektes mit Angabe der Abstammung (Klasse).

Objekte können selbstverständlich auch mit weiteren Elementen wie Attributen versehen werden. Dies ist jedoch selten erforderlich.

³ Nach [1] handelt es sich um Empfehlungen.

3.2.1.3 Schnittstelle (Interface)

Das Interface stellt eine Sammlung von Funktionen zur Verfügung, über die eine Klasse angesprochen werden kann. Im Normalfall spricht man von Interface im Zusammenhang mit Serverklassen. Das heisst Klassen die als Serverobjekt auftreten und eine bestimmte Funktionalität zur "Aussenwelt" aufweisen. Ein Interface wird grafisch durch einen Kreis repräsentiert. Dabei kann das Interface selbständig dargestellt werden. Im allgemeinen jedoch ist ein Interface mit der Klasse die das Interface zur Verfügung stellt zusammen dargestellt.

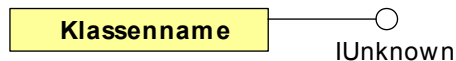


Abbildung 3-7 Darstellung eines Interface an einer Klasse.

3.2.1.4 Kollaboration (Collaboration)

Eine Kollaboration definiert die Interaktionen und die Gemeinsamkeit von Rollen aller beteiligten Elemente in einem bestimmten Kontext. Die Kollaboration selber bildet dabei eigentlich nur die abgekürzte Form der Wiedergabe in einem Modell. Die detaillierte Darstellung der Kollaboration erfolgt im sog. Kollaborationsdiagramm. Die grafische Repräsentation einer Kollaboration wird durch ein Oval mit gestrichelter Umrandung dargestellt. Im Oval wird der Name der Kollaboration wiedergegeben.

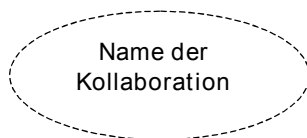


Abbildung 3-8 Darstellung einer Kollaboration.

3.2.1.5 Anwendungsfall (Use Case)

Ein Anwendungsfall beschreibt eine Kette von Aktionen mit dem Ziel der Realisierung eines bestimmten Resultates aus Sicht eines Akteurs. Die Anwendungsfälle bilden in einer OO Analyse mit UML das Rückgrat für die Methode. Die Anwendungsfälle werden grafisch durch ein Oval mit ausgezogener Linie repräsentiert. Im Oval wird der Name des Anwendungsfalles dargestellt.

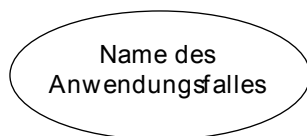


Abbildung 3-9 Darstellung eines Anwendungsfalles.

3.2.1.6 Aktive Klasse (Active Class)

Eine aktive Klasse beschreibt Objekte, die einen oder mehr Prozesse oder Threads besitzen. Dementsprechend können solche Objekte gleichzeitig zu andern Objekten in Aktion treten. Grafisch unterscheidet sich eine aktive Klasse von einer normalen Klasse nur durch eine dickere Linie des Rechteckes.

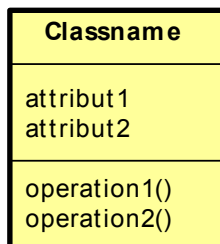


Abbildung 3-10 Darstellung einer aktiven Klasse.

3.2.1.7 Komponente (Component)

Ein System ist normalerweise aus mehreren Komponenten zusammengesetzt. Diese bilden die physischen Einheiten des ausführbaren Codes. Komponenten können in verschiedensten Ausprägungen wie COM+ oder Java Beans o.A. vorkommen. Komponenten werden mit einem Rechteck mit seitlichen Tab-Rechtecken angeordnet. Innerhalb des umfassenden Rechtecks wird der Name der Komponenten eingetragen.

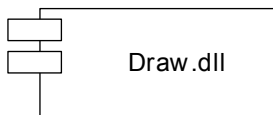


Abbildung 3-11 Darstellung einer Komponente.

3.2.1.8 Knoten (Node)

Ein Knoten wird benutzt um eine physische Ressource darzustellen. Damit ist meistens ein Prozessor oder ein Rechner gemeint, der in sich in der Lage ist ein Programm auszuführen. Grafisch wird ein Node mit einem Kubus dargestellt. In der Frontseite wird der Name des Nodes eingetragen.

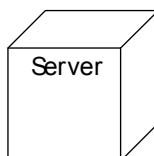


Abbildung 3-12 Darstellung eines Knoten.

3.2.2 Verhalten (Behaviour)

Die Dinge die das Verhalten in einem Modell repräsentieren, zeigen die dynamischen Verhältnisse zwischen Elementen auf. Dazu gehören zwei Dinge:

- die Mitteilung (Message)
- der Zustand (State)

3.2.2.1 Mitteilung (Message)

Mitteilungen dienen zur Darstellung einer Interaktion zwischen zwei verschiedenen strukturellen Dingen. Wir können auch sagen, es handelt sich um die modellhafte Darstellung eines Aufrufes einer Funktion oder eines Operators. Grafisch wird eine Mitteilung mit einem Pfeil mit ausgezogener Linie und ausgefüllter Pfeilspitze dargestellt.



Abbildung 3-13 Darstellung einer Mitteilung.

3.2.2.2 Zustand (State)

Für die Modellierung von Zustandsmaschinen, welche den Lebenszyklus resp. die Veränderungen des Zustandes einer Klasse (oder Objektes) während eines Lebenszyklus zeigt, benötigen wir ein Symbol für die Darstellung eines bestimmten Zustandes. Dieser wird mit einem Rechteck mit abgerundeten Ecken repräsentiert. Innerhalb des Rechteckes wird der Name des Zustands definiert, allfällige Attributwerte oder Aktionen, die beim Eintreten von weiteren Ereignissen ausgelöst werden.

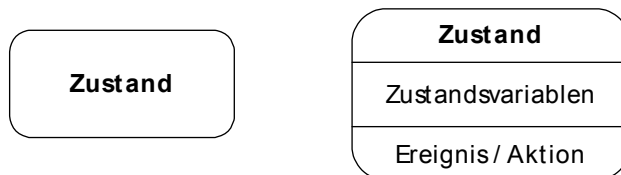


Abbildung 3-14 Darstellung eines einfachen und eines vollständigen Zustandes.

3.2.3 Gruppierung (Package)

Die Gruppierung kann in beliebigem Kontext Elemente organisieren. Dabei wird die Gruppierung insbesondere für die Bildung von konzeptionellen Gruppen verwendet. Dies unterscheidet sie von der Komponente, welche die physische Struktur zeigt. Am häufigsten wird die Gruppierung im Zusammenhang mit der Paketbildung (engl. Package) verwendet. Das Package zeigt eine zusammengehörende Gruppe von Dingen, die während der Entwicklung eines Produktes zusammengehören. Eine Gruppe wird grafisch mit einer Karteikarte repräsentiert und beinhaltet den Namen der Gruppe.

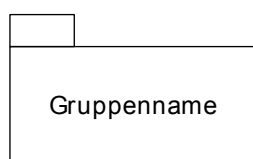


Abbildung 3-15 Darstellung einer Gruppe.

3.2.4 Anmerkung (Note, Annotation)

Die Anmerkung kann als Element an beliebiger Stelle in beliebiger Anzahl in einem Modell platziert werden. Sie dient zur einfachen textuellen Ergänzung, zur Kommentierung, zur Klärung einer allenfalls sonst falsch verständlichen Modellkomponente oder ähnlichen. Die Anmerkung wird mit einem Rechteck mit einem "Eselsohr" dargestellt. In das Rechteck werden die Bemerkungen eingefügt.

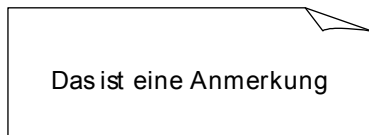


Abbildung 3-16 Darstellung einer Gruppe.

3.3 Beziehungen (Relations)

Strukturelle Elemente in einem Modell zeigen einen wichtigen Teil der realen Welt. Wie das Gewürz in der Speise sind jedoch die Beziehungen zwischen den strukturellen Elementen die interessante, und oft nicht einfach "abzuschmeckende" Ergänzung.

3.3.1 Generalisierung, Spezialisierung (Generalization, Speziation)

Die Generalisierung drückt eine taxonomische Beziehung zwischen (in der Regel) mehreren speziellen und einem allgemeinen Element aus. Das allgemeine Element fasst dabei Eigenschaften und Semantik zusammen, so dass sich alle Spezialisierungen zum generalisierten Element kompatibel verhalten können.

Die Spezialisierung ist der Umkehrprozess der Generalisierung. Dabei werden dem allgemeinen Element Eigenschaften hinzugefügt und die Semantik wird erweitert. Das Verhalten des speziellen Elementes bleibt aber kompatibel zum allgemeinen Element. Insbesondere in Bezug auf die Spezialisierung spricht man von der sogenannten Vererbung (engl. Inheritance).

Grafisch wird eine Vererbung mit einem grossen, nicht ausgefüllten Pfeil dargestellt. Dieser zeigt von der Unterklasse (Unterklasse = spezialisiertes Element) in Richtung der Oberklasse (Oberklasse = allgemeines Element). Sind mehrere Unterklassen allenfalls in mehreren Stufen involviert, können die Pfeile für die Vererbung und die Anordnung der Klassen frei gewählt werden. In der Praxis haben sich allerdings unten aufgezeigte Varianten "eingebürgert".

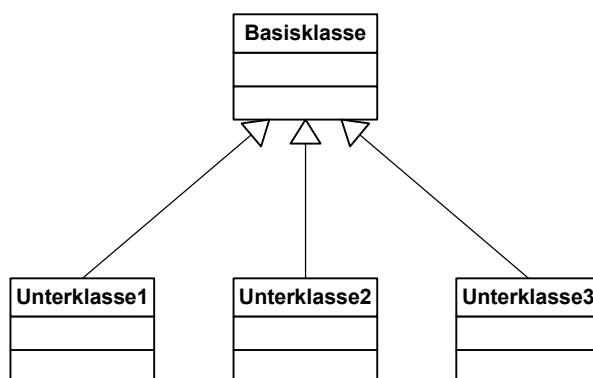


Abbildung 3-17 Vererbung in der Fächerdarstellung.

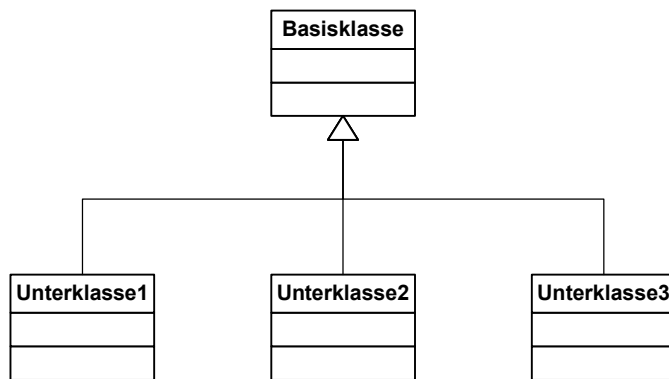


Abbildung 3-18 Vererbung in der Organigrammdarstellung.



Abbildung 3-19 Vererbung in der Bibliotheksdarstellung.

Spezialisierungen werden aufgrund eines bestimmten Kriteriums gestaltet. Dieses Unterscheidungskriterium wird in der UML als Diskriminator bezeichnet. Der Diskriminator kann in der Vererbungsbeziehung eingezeichnet werden. Gilt er für mehrere Beziehungen, kann seine Gültigkeit mittels einer gestrichelten Linie ausgedrückt werden. Andernfalls wird er einzeln zu jeder Beziehung geschrieben.

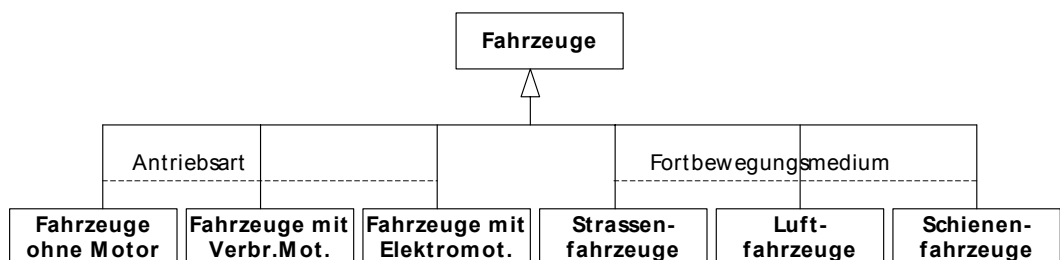


Abbildung 3-20 Darstellung des Diskriminators.

Generalisierungsbezeichnungen können mit Constraints versehen werden. Folgende Standarddefinitionen sind in UML vorgesehen:

- overlapping Ist speziell für die Mehrfachvererbung gedacht und drückt aus, dass die gemeinsamen Eigenschaften singular in ein spezialisiertes Objekt einfließen.
- disjoint Inverse Zusicherung zu overlapping. Die Zusicherung disjoint ist die Standardzusicherung für den Fall, dass keine spezielle Angabe gemacht wird.
- complete Drückt aus, dass alle Unterklassen spezifiziert wurden.
- incomplete Drückt aus, dass noch nicht alle Unterklassen spezifiziert wurden.

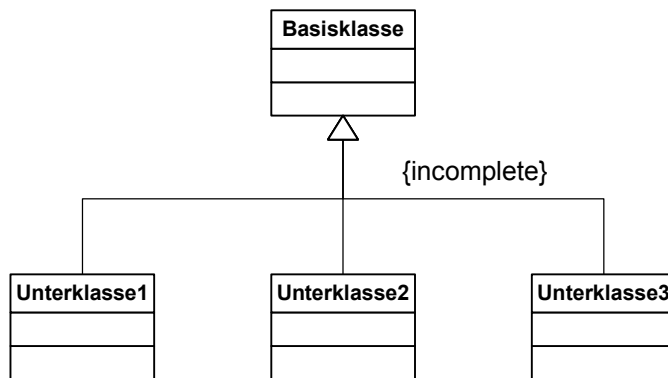


Abbildung 3-21 Darstellung einer Zusicherung.

Mehrfachvererbung (Multiple Inheritance)

Spezialisierungen bei denen die Unterklasse von genau einer Oberklasse abgeleitet ist heissen Einfachvererbungen. Es ist somit logisch, dass der Begriff Mehrfachvererbung eine Ableitung von einer Unterklasse von mehr als einer Oberklasse definiert. Getreu dem Gesetz der Vererbung, erbt bei der Mehrfachvererbung die Unterklasse ihre Eigenschaften von allen Oberklassen von der sie abstammt.

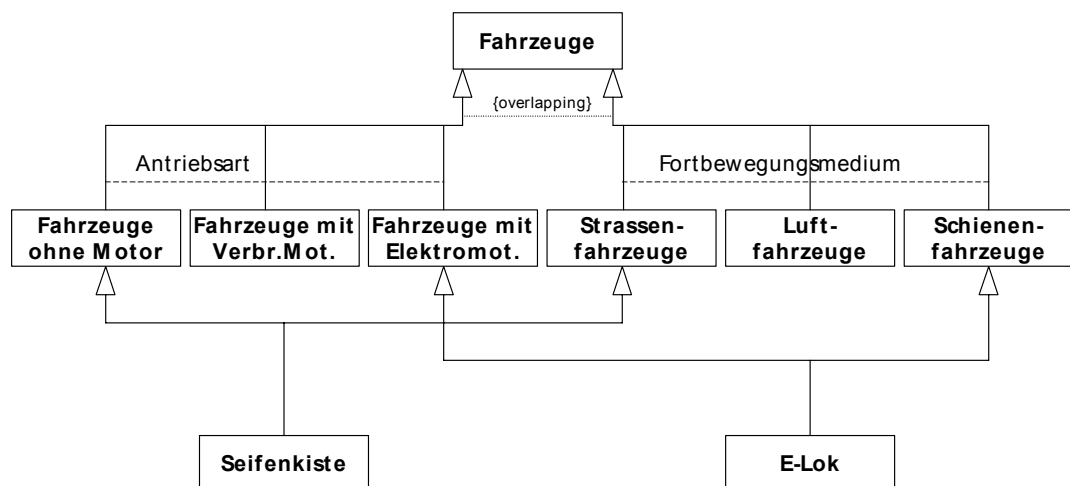


Abbildung 3-22 Darstellung einer Mehrfachvererbung.

Beachten Sie die Zusicherung der ersten Vererbungsstufe zwischen den Vererbungswegen der verschiedenen Diskriminatoren. Wird diese nicht gemacht gilt eine "disjunkte" Verbindung, was zur Folge hätte, dass die Eigenschaften von Fahrzeug über zwei Wege nach "Seifenkiste" oder "E-Lok" vererbt würden.

3.3.2 Assoziationen (Associations)

Eine Assoziation (dt. Vereinigung) stellt eine freie Beziehung zwischen zwei Elementen dar. Dabei wird impliziert, dass die Beziehung in beide Richtungen definiert ist (bidirektional). Das heisst, dass jeweiligen das eine mit dem andern Objekt eine Beziehung unterhält, als auch umgekehrt. Die grafische Darstellung erfolgt dabei mit einer ausge-

zogenen Verbindungslinie. Eine Assoziationslinie steht dabei für beide Leserichtungen die eine Verbindung besitzt. Die Bezeichnung der Assoziation kann für beide Leserichtungen separat erfolgen. Im Zweifelsfall ist dabei die Bezeichnung mit einem ausgefüllten Dreieck anzugeben dessen Spitze in Leserichtung zeigt. Im weiteren können die Bezeichnungen der Relationen mit Stereotypen und Zusicherungen versehen werden, respektive mit Sichtbarkeiten oder andern Attributen präfixiert werden⁴.

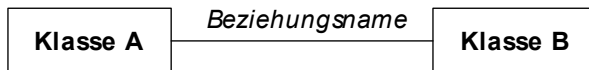


Abbildung 3-23 Einfache Darstellung einer Assoziation.

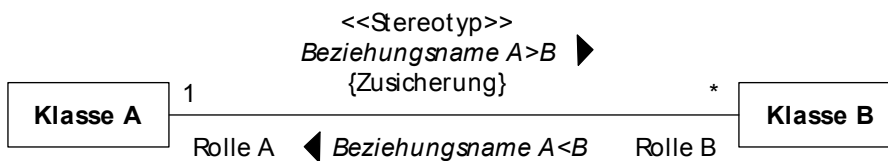


Abbildung 3-24 Darstellung einer Assoziation mit allen Elementen.

Die Mengenangabe der Beziehung wird auf der Assoziationslinie bei den jeweiligen Elementen angegeben. Dabei wird zwischen Multiplizität und Kardinalität unterschieden. Die Kardinalität definiert die Anzahl der Elemente in der Beziehung, währenddem die Multiplizität den Bereich erlaubter Kardinalitäten definiert. Die Multiplizitätsangaben sind implizit oder explizit möglich. Bei expliziter Angabe können einzelne Werte, Listen (Trennung durch Komma), Bereiche (Untere Grenze .. Obere Grenze) oder Kombinationen davon angegeben werden.

Beispiele von Multiplizitätsangaben:

Angabe	Bedeutung
1	genau eins
0,1	null oder eins
0..4	zwischen null und vier
3, 7	genau drei oder genau sieben
0 .. *	größer oder gleich null (diese Definition ist Standard, wenn keine explizite Angabe erfolgt)
*	Abgekürzte Schreibweise von 0 .. *
1 .. *	größer oder gleich eins
0 .. 3, 7, 9 .. *	größer null bis genau drei oder genau sieben oder größer neun

Rekursive Assoziation

Eine spezielle Anwendung erfährt die Assoziation bei der Definition einer Beziehung einer Sache auf sich selbst. In diesem Fall sprechen wir von einer rekursiven Assoziation.

⁴ Sichtbarkeit analog wie Attribute bei Klassen. do. für Ableitung (siehe auch abgeleitete Assoziation).

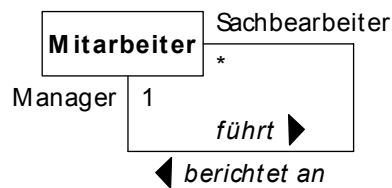


Abbildung 3-25 Darstellung einer rekursiven Assoziation.

Attributierte Assoziation (Assoziationsklasse)

Eine weitere spezielle Anwendung der Assoziation ist die attributierte Beziehung, mit welcher die Existenz von Attributen innerhalb einer Beziehung ausgedrückt werden kann. Man beachte, dass die attributierte Assoziation in den meisten Fällen (abhängig von der Multiplizität) zur Bildung von Klassen führt. Entsprechend wird in der UML die Angabe von attribuierten Beziehungen die eigentliche Beziehung mit einem Klassensymbol ergänzt. In der Literatur findet man dazu auch den Begriff Assoziationsklasse.

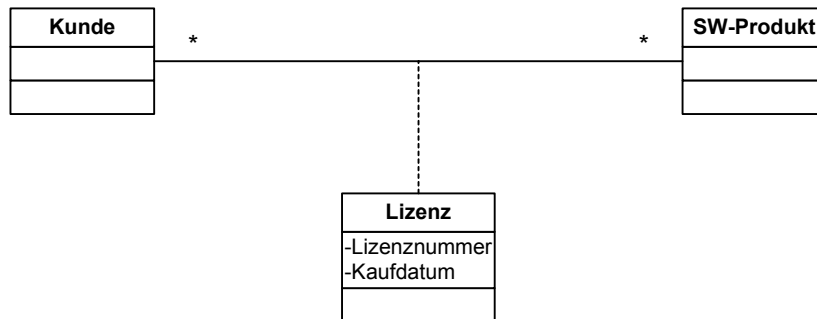


Abbildung 3-26 Darstellung einer Assoziationsklasse.

Qualifizierte Assoziation

Eine weitere spezielle Form der Assoziation bildet die qualifizierte Beziehung, welche die referenzierte Menge der Objekte durch qualifizierte Attribute in sogenannte Partitionen unterteilt (Gruppierung). Die entstehenden Partitionen müssen dabei aus Sicht des Ausgangsobjektes eindeutig sein. Grafisch wird die Angabe des Qualifizierers in einem Rechteck innerhalb der Assoziation, angrenzend an die qualifizierte Klasse dargestellt. Innerhalb des Rechteckes wird das qualifizierende Attribut (oder mehrere), welches Bestandteil der Beziehung ist, angegeben.

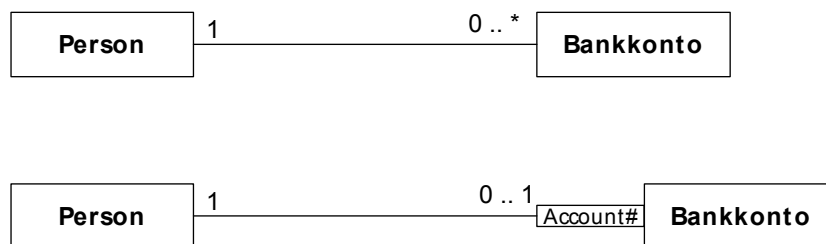


Abbildung 3-27 Darstellung einer qualifizierten Assoziation.

Abgeleitete Assoziation

In der abgeleiteten Assoziation finden wir eine weitere Spezialform der Beziehung zwischen zwei Klassen. Die abgeleitete Beziehung entsteht wenn eine konkrete Objektbeziehung jederzeit aus den Werten einer andern Beziehung nachvollzogen werden kann (Zirkularbezug). In einem Modell wird die abgeleitete Assoziation als normale Assoziation eingezeichnet. Einzig ihr Name wird mit einem "/" Zeichen eingeleitet⁵.

⁵ Man beachte, dass abgeleitete Attribute in Klassen ebenfalls mit einem führenden "/" Zeichen geschrieben werden (allgemeingültig).

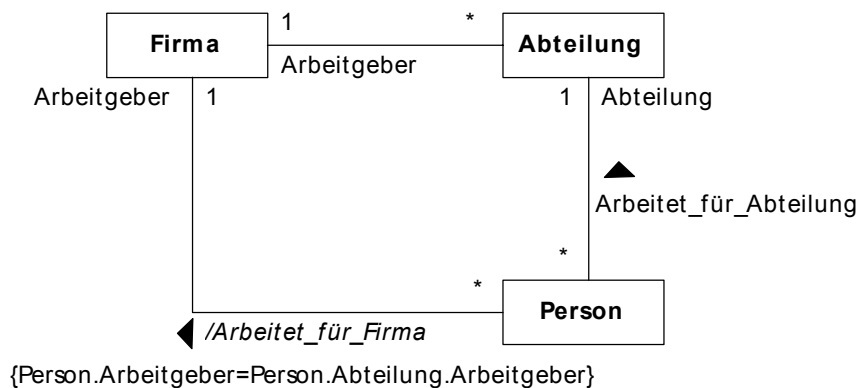


Abbildung 3-28 Darstellung einer abgeleiteten Assoziation.

Gerichtete Assoziation

Die UML impliziert die Assoziation als bidirektional. Das heisst, dass eine Beziehung vom einen zum andern Objekt und umgekehrt verwaltet wird. Dies impliziert aber auch, dass die entsprechenden Klassen über die notwendigen Attribute oder Operationen verfügen müssen. In vielen Fällen ist diese bidirekte Beziehung jedoch schwierig zu implementieren oder sogar unerwünscht. Für diesen Fall kann auf eine einseitige, sogenannt gerichtete Assoziation zurückgegriffen werden. Grafisch wird das Symbol der normalen Assoziation durch eine offene Pfeilspitze in Richtung der Klasse (oder des Objekts) gezeichnet, auf die von der Ausgangsklasse (oder Objekt) her zugegriffen werden kann aber nicht umgekehrt.

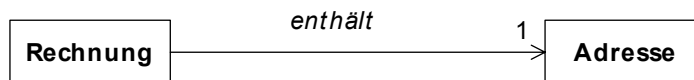


Abbildung 3-29 Darstellung einer gerichteten Assoziation.

Die gerichtete Assoziation impliziert in der Praxis, dass vom Objekt auf das navigiert wird (rechts in der Abbildung) nicht auf das Objekt das navigiert (links in der Abbildung) zurückgeschlossen werden kann.

Mehrgliedrige Assoziation

Letztendlich bleibt noch die mehrgliedrige Assoziation⁶ zu erwähnen. Bei dieser sind in der Beziehung mehr als zwei Rollen (in der Regel Klassen) beteiligt. Grafisch werden die verschiedenen Beziehungen von den Klassen zu einer Raute zusammengeführt. Diese Raute bildet die eigentliche Beziehung und wird benannt, oder aber es wird eine attributierte Assoziation zur Raute hinzugefügt. In letzterem Fall ergibt sich der Name der mehrgliedrigen Beziehung aus der attribuierten Assoziation (Klasse).

⁶ Auch ternäre Assoziation oder n-ary association genannt.

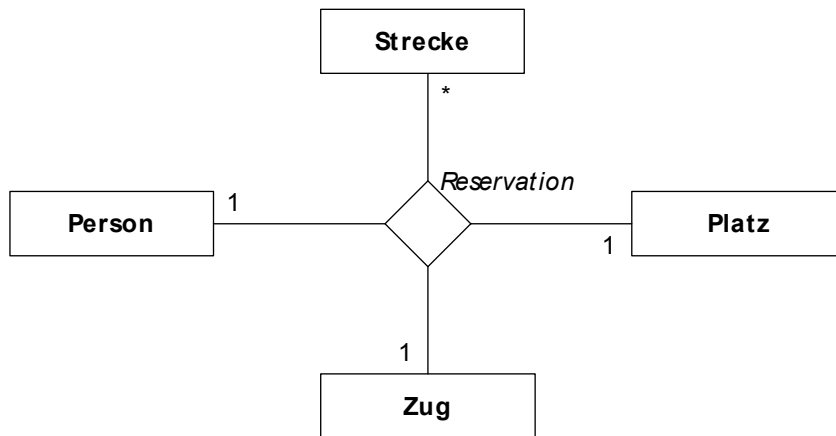


Abbildung 3-30 Darstellung einer mehrgliedrigen Assoziation.

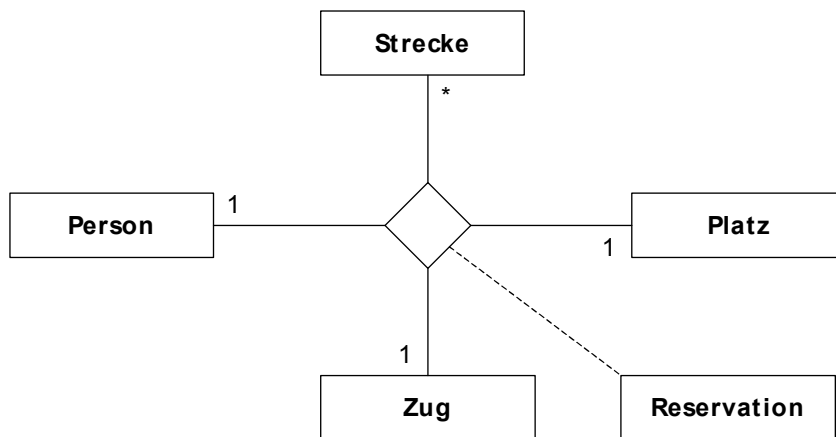


Abbildung 3-31 Darstellung einer mehrgliedrigen Assoziation durch Angabe einer attribuierten Klasse.

Eine mehrgliedrige Assoziation kann in aller Regel mit einer attribuierten Klasse mit entsprechend vielen normalen Assoziationen überführt werden.

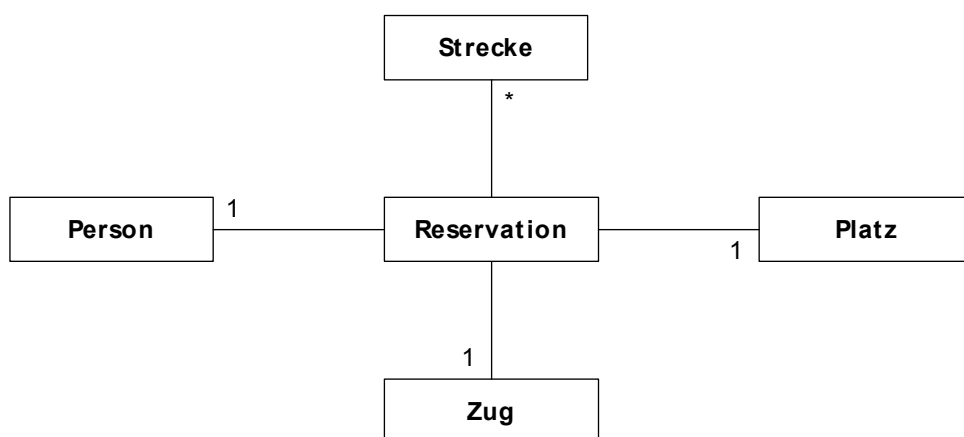


Abbildung 3-32 Aufgelöste mehrgliedrige Assoziation.

3.3.2.1 Aggregationen (Aggregation)

Die Aggregation ist eine spezielle Form der Assoziation und definiert eine sogenannte Teil-Ganzes Beziehung. Eine Aggregation besteht dabei aus mindestens zwei verschiedenen Teilen:

- Das Aggregat: auch als Ganzes bezeichnet, kommt einmalig in der Beziehung vor. Es übernimmt innerhalb der Aggregation stellvertretend für alle Einzelteile spezielle Aufgaben.
- Das Einzelteil: auch als Teil bezeichnet, kommt in Form einer oder mehrerer Klassen vor.

Die Aggregation beschreibt im Gegensatz zu der Assoziation keine gleichberechtigte Beziehung, da dem Aggregat quasi eine Führungsrolle zugeordnet ist. Die Führungsrolle umfasst vor allem die Kontrolle der Aggregation. Dazu gehört zum Beispiel das Weiterleiten von Operationen an die Teile oder das Festhalten der Anzahl der Teile in der Aggregation.

Grafisch wird eine Aggregation wie eine Assoziation festgehalten. Zur Unterscheidung beider wird bei der Aggregation auf der Seite des Aggregates eine kleine Raute in die Verbindungslinie gezeichnet. In der Regel besteht eine Aggregation aus einem Aggregat und mehreren Teilen. Aufgrund dessen kann das Fehlen der Multiplizitätsangabe auf Seite des Aggregates als 1 interpretiert werden.

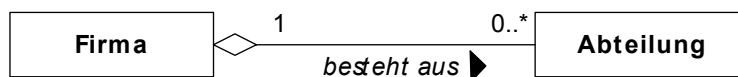


Abbildung 3-33 Einfachste Aggregation.

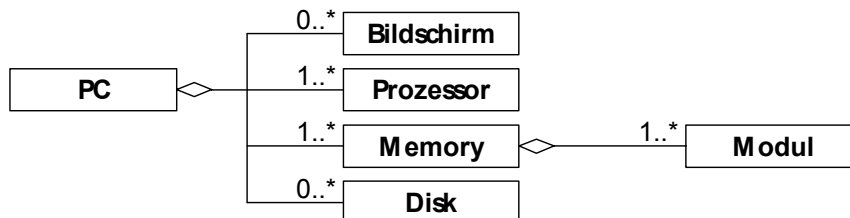


Abbildung 3-34 Mehrgliedrige Aggregation mit Kaskade.

3.3.2.2 Komposition (Composition)

Die Komposition ist eine weitere Spezialform der Aggregation und somit auch der Assoziation. Als Spezialität der Komposition beschreibt diese eine Beziehung, bei der das Teil innerhalb der Komposition existenzabhängig vom Aggregat ist. Das heißt, dass das Teil mit dem Aggregat (oder später) instanziiert wird und spätestens mit dem Zerstören des Aggregates auch wieder zerstört wird. Aufgrund dessen ist die Kardinalität bei einer Komposition auf der Seite des Aggregates immer 1 (ansonsten entsteht spätestens beim Löschvorgang des einen Aggregates ein Widerspruch).

Grafisch wird die Komposition analog der Aggregation gezeichnet. Zur Unterscheidung wird die Raute ausgefüllt dargestellt.

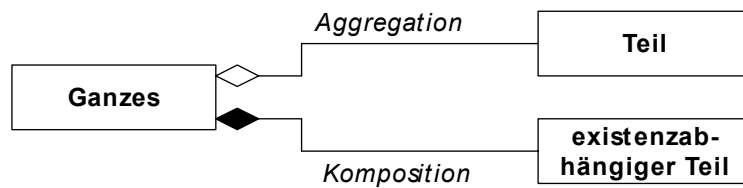


Abbildung 3-35 Grafischer Unterschied zwischen Aggregation und Komposition.

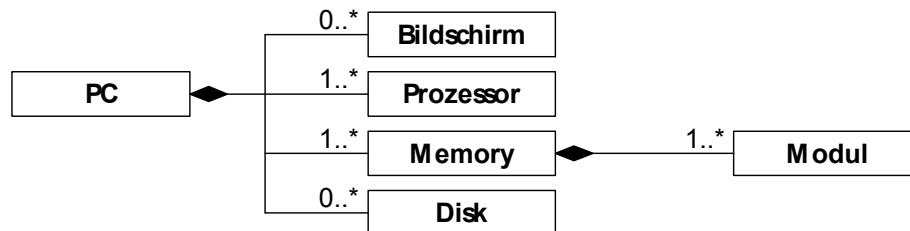


Abbildung 3-36 Mehrgliedrige Komposition mit Kaskade.

3.3.3 Abhängigkeiten (Dependency)

Abhängigkeiten zeigen das Verhältnis zwischen selbständigen und unselbständigen Elementen. Grafisch wird eine Abhängigkeit mit einem Pfeil mit gestrichelter Linie gezeichnet. Die Pfeilspitze zeigt dabei auf das unabhängige Element.



Abbildung 3-37 Darstellung einer Abhängigkeit.

3.3.4 Realisierung (Realisation)

Die Realisierung dient zum Aufzeigen in welcher Komponente Interfaces oder Klassen realisiert sind und in welchem Anwendungsfall eine Kollaboration realisiert wird. Grafisch wird eine Realisierung mit einem nicht ausgefüllten Pfeil, dessen Linie gestrichelt gezeichnet ist, dargestellt.

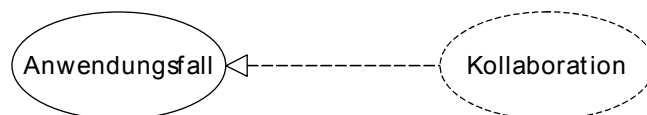


Abbildung 3-38 Darstellung einer Realisierung.

3.4 Diagramme (Diagrams)

Die Diagramme sind das eigentliche Ausdrucksmittel der UML. Mittels den Diagrammen werden Elemente der Problemlösung herausgeschält und modelliert. Wichtig ist beim letzten Satz das Wort modellieren. Es drückt aus, dass der Prozess der Softwareentwicklung keine reine mathematische Angelegenheit ist, sondern - ja, ich neige zu sagen - etwas künstlerisches an sich hat. Dementsprechend gibt es auch oft nicht nur eine Lösung eines Problems. Viel-

mehr ist die Lösung abhängig von den Umsystemen, den bereits vorhandenen Teillösungen, von den Gewohnheiten der User und auch von der Bereitschaft der User Neues aufzunehmen.

Keines der in diesem Kapitel technisch beschriebenen Diagramms ist in der Lage das gesamte zu modellierende System wiederzugeben. Ja meistens ist es so, dass nicht einmal die Gesamtheit aller Diagramme ermöglichen, ein System in seiner Gesamtheit zu beschreiben. Dies kommt davon, dass die Systeme die heute gebaut werden schlicht komplexer und umfassender sind, als das es mit einer Sprache möglich ist diese zu beschreiben.

Aus der Not wird beim Modellieren eine Tugend gemacht! Das heisst, dass in den Diagrammen gerade soviel Information wiedergegeben wird, wie zum eindeutigen Verständnis des Diagramms notwendig ist (im Zweifelsfall lieber klar).

3.4.1 Klassendiagramm (Class Diagram)

Das Klassendiagramm ist wohl die bekannteste und auch am meisten verwendete Ausdrucksform für die Modellierung von objektorientierten Systemen. Es zeigt die verschiedenen Klassen, deren Attribute Operationen sowie die Beziehungen (Vererbungen, Assoziationen ...) zwischen den Klassen.

Je nach Grösse und Art eines Systems können dabei alle Klassen und Beziehungen im gleichen Diagramm (eher selten), nach dem Kontext oder nach der Art der Beziehung geordnet werden. Werden die Diagramme nach Beziehungsart auseinander genommen, spricht man von sogenannten Vererbungs-, Containment- (Aggregation und Komposition) und Relationsdiagrammen (Assoziation). Egal welche Aufteilungsart gewählt wird. Allen Arten gemeinsam ist der Zweck der Aufteilung: Schaffen von Übersicht. Das gleiche gilt auch für die Darstellung von Attributen und Operationen. Innerhalb einer Klasse werden gerade die Attribute oder Operationen dargestellt, die für das Verständnis des Diagramms notwendig sind.

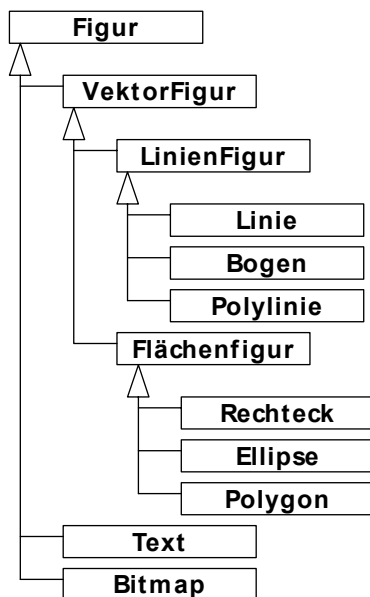


Abbildung 3-39 Darstellung eines reinen Vererbungsdiagramms.

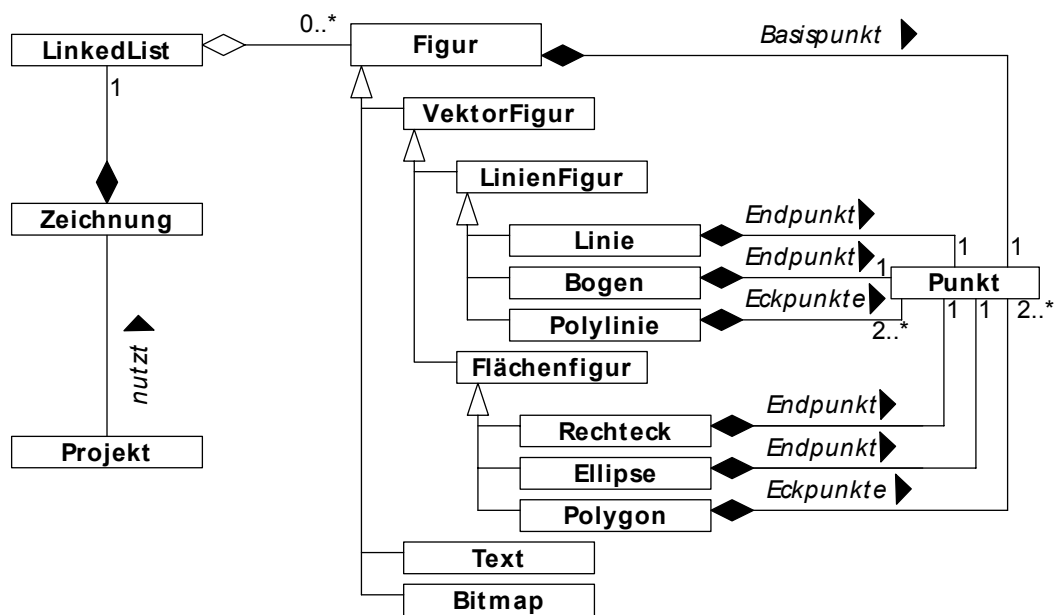


Abbildung 3-40 Darstellung eines gemischten Klassendiagramms.

Je nachdem ob ein Tool für OO Analyse oder Design verwendet wird, können die verschiedenen Daten der Klassen im Tool verwaltet werden oder nicht. Für den Fall, dass kein Tool zur Verfügung steht, sollte neben der grafischen Repräsentation auch noch eine verbale Beschreibung der Klassen bestehen. Dabei könnte folgende Aufstellung behilflich sein:

Verbale Beschreibung der Klassen

Name der Klasse:	<<Name der Klasse selber>>
Basisklasse:	<<Name der Basisklasse>>
Verwendung:	<<Beschreiben Sie hier den Verwendungszweck der Klasse>>
Abstrakt:	Ja / Nein

Attribute:

Name	Verwendung	Typ	S ⁷	M ⁸	Plausiregeln / Bemerkungen
...					
...					

Operationen:

Signatur ⁹	S	Regeln / Bemerkung
...		
...		

⁷ Die Spalte "S" (für Sichtbarkeit) definiert das Zugriffsrecht für das Attribut

⁸ Die Spalte "M" (für Mandatory) definiert ob das Attribut mit einem Wert versehen werden muss oder nicht (in Bezug auf DB-Projekte wichtig)

⁹ inkl. Rückgabtyp

3.4.2 Objektdiagramm (Object Diagram)

Ein Objektdiagramm zeigt einen bestimmten Zustand von Instanzierungen aus dem Klassenmodell zu einem bestimmten Zeitpunkt. Das Objektdiagramm wird oft zur Veranschaulichung eines komplexen Sachverhalts (Hinterfragen des Design) oder beim Reverse-Engineering eingesetzt.

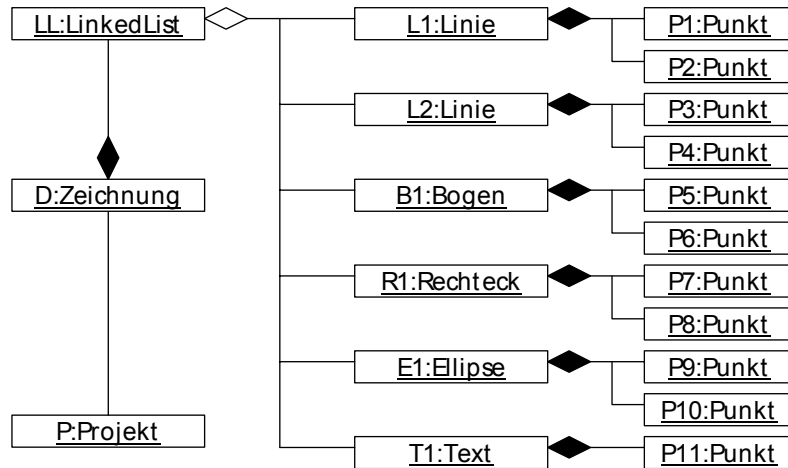


Abbildung 3-41 Darstellung eines Objektdiagramms.

3.4.3 Anwendungsfalldiagramm (Use-Case Diagram)

Das Anwendungsfalldiagramm (engl. Use case diagram) zeigt einen oder mehrere Anwendungsfälle und Akteure und deren Zusammenhang. Das Anwendungsfalldiagramm beantwortet im speziellen die Frage wer macht was.

Die Akteure sind "verkappte" Klassen, die meist in Form von Icons in einem Diagramm gezeichnet werden. Das sog. Strichmännchen ist die in der UML bevorzugte Form. Dieses Icon kann stellvertretend für einen beliebigen Akteur (ein Mensch aber auch ein anderes System oder Subsystem ausserhalb des zu untersuchenden Systems) sein. Akteure sind eigentlich normale Klassen, die untereinander auch in Generalisierungsbeziehungen treten können. Dieses Verdeutlichung macht dort Sinn, wo der Anwender verschiedene Akteure in Beziehung zu einem Anwendungsfall bringt, jedoch letztlich in der Analyse erkannt wird, dass die verschiedenen Akteure keine unterschiedliche Wirkung auf den Anwendungsfall ausüben.

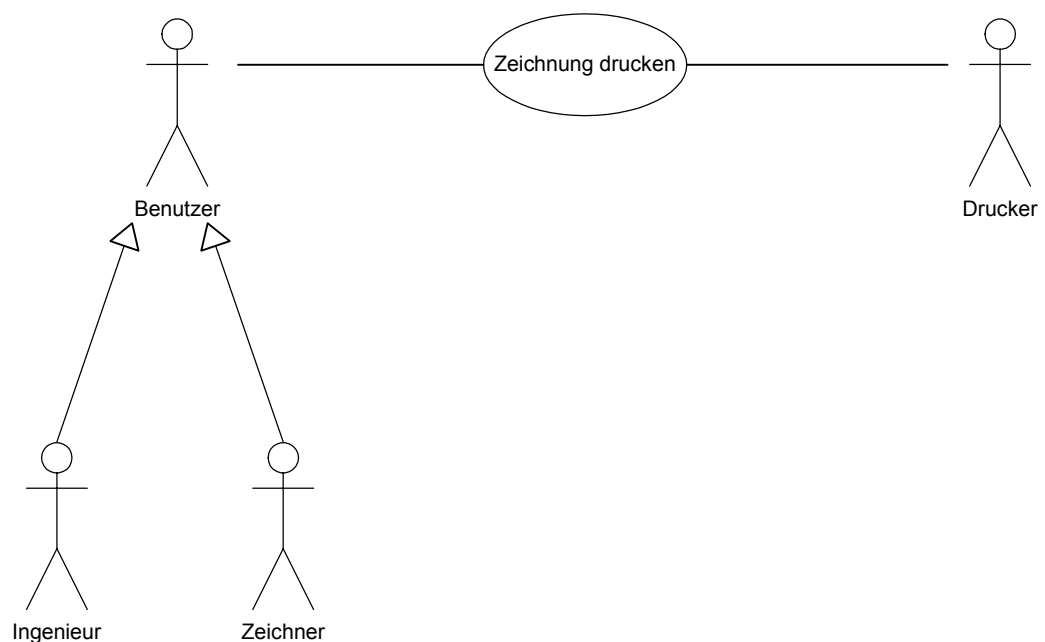


Abbildung 3-42 Darstellung eines Anwendungsfall Diagramms.

Das Anwendungsfalldiagramm wird ergänzt mit der Beschreibung der enthaltenen Anwendungsfälle. Die Beschreibung wird mit Vorteil strukturiert. Damit kann in einem Entwicklerteam, das aus mehreren Personen besteht, erreicht werden, dass die Beschreibung von verschiedenen Anwendungsfällen und Personen in etwa gleich granuliert wird. Eine Struktur für die Beschreibung könnte z.B. wie folgt aussehen (In UML existiert z.Z. noch keine Vorgabe):

Beschreibung eines Anwendungsfalles

Name:	Drucken einer Zeichnung
Nummer:	1
Kurzbeschreibung:	Die Zeichnung wird aus dem laufenden Programm auf dem angeschlossenen Ausgabegerät ausgegeben (print oder plot).
Beteiligte Akteure:	Benutzer und Ausgabegerät
Auslöser / Vorbedingung:	Zeichnung ist im System geladen
Ergebnisse / Nachbedingung:	Zeichnung ist auf Papier übertragen

Ablauf:

Nr.	Wer	Was
1	Benutzer	Einstellungen Drucker vornehmen
1.1	Benutzer	Drucken auslösen
1.2	System	Plausibilisierung der Einstellungen (Ränder, Farben...)
1.3	System	Ausnahme: Daten nicht i.O
1.4	System	Berechnungen für Ausdruck (Anzahl Seiten)
2	System	Effektive Ausgabe
2.1	System	Ausgabe der aktuellen Seite

Ausnahmen, Varianten:

Nr.	Wer	Was
1.3		Daten nicht i.O
1.3.1	System	Fehlermeldung mit Angabe der nicht korrekten Daten, Abbruch des Vorganges
1.3.2	Benutzer	Korrektur der Daten
1.3.3	Benutzer	Erneutes Aufsetzen des Vorganges

Akteurmodell

Wenn zwischen den Akteuren nicht einfache Beziehungen herrschen, kann es sinnvoll sein, ein separates Akteurmodell zu entwerfen. Das Modell zeigt die verschiedenen Verhältnisse zwischen den Akteuren auf und erklärt dabei die einzelnen Akteure in Form von Attributen und Operationen analog einem Klassendiagramm. Das Akteurmodell kann insbesondere für die Betrachtung der Sicherheitsaspekte eine grosse Rolle spielen.

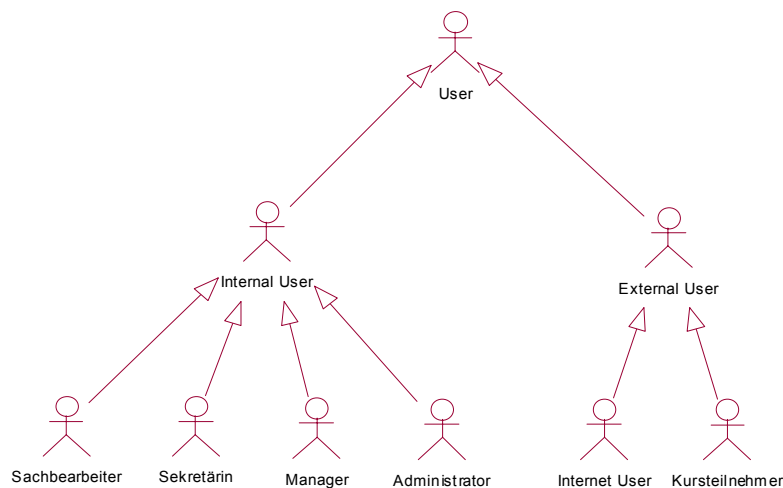


Abbildung 3-43 Darstellung eines Akteurmodells mit Icons.

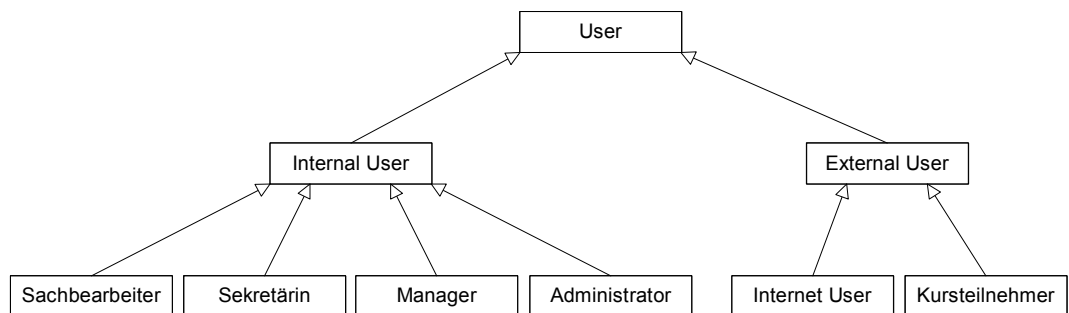


Abbildung 3-44 Darstellung eines Akteurmodells mit Klassensymbolen.

3.4.4 Aktivitätendiagramm (Activity Diagram)

Das Aktivitätendiagramm beschreibt den Ablauf oder einen Teilablauf des Systems in Bezug auf

- eine Klasse
- eine Operation
- einen Anwendungsfall

Sie sind somit eine mögliche Art der feineren Beschreibung des Modells. Die Aktivität bildet dabei einen einzelnen Schritt in der Verarbeitung. Der Übergang von der einen Aktivität in die nächste wird Transition genannt. Eine Aktivität kann mehrere ausgehende Transitionen besitzen. Dies setzt jedoch voraus, dass die verschiedenen ausgehenden Transitionen über Bedingungen unterscheidbar sind.



Abbildung 3-45 Zwei Aktivitäten mit einer Transition.

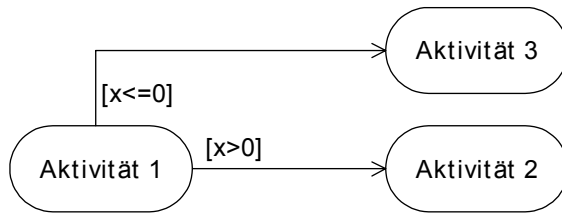


Abbildung 3-46 Eine Aktivität mit zwei ausgehenden Transitionen.

Eine weitere Möglichkeit der Beschreibung der Aktivitäten bietet UML durch die Möglichkeit Transitionen zu synchronisieren oder zu teilen. Dies wird grafisch mit einem dicken Balken festgehalten, auf dem die betroffenen Transitionen enden resp. von dem aus die Transitionen beginnen.

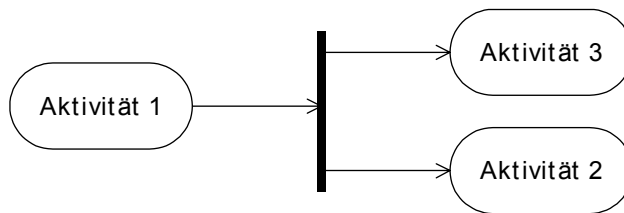


Abbildung 3-47 Teilen einer Transition.

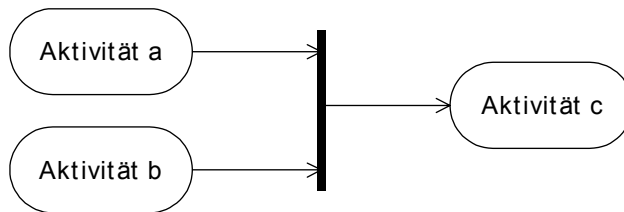


Abbildung 3-48 Synchronisation zweier Transitionen.

Innerhalb des Aktivitätendiagramms können zusätzlich Objektzustände dargestellt werden. Diese weitere Möglichkeit der Verfeinerung kann weitere Klärung der Abläufe mit sich bringen.

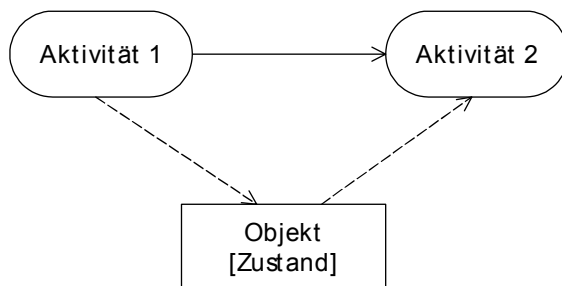


Abbildung 3-49 Darstellung eines Objektzustandes.

Für die Bezeichnung des Beginns und des Endes eines Aktivitätendiagramms sind die beiden speziellen Symbole Punkt (Beginn) und Punkt in einem Kreis (Ende) definiert.

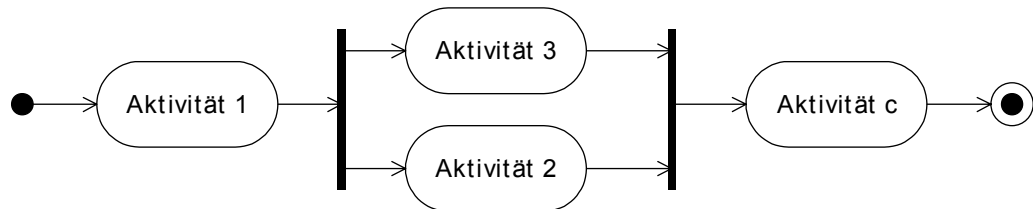


Abbildung 3-50 Aktivitätsdiagramm mit definiertem Anfang und Ende.

3.4.5 Zustandsdiagramm (State Diagram)

Das Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Verlaufe seines Lebenszyklus einnehmen kann. Das Diagramm wird nach der State-Event Technik als endlicher Automat dargestellt, der sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet.

Ein Zustand (siehe auch Kapitel "Zustand") ist als Menge bestimmter Attributwerte definiert. Der Übergang von einem Zustand in den andern erfolgt aufgrund von Ereignissen (auch Zustandsübergang genannt).

Ereignisse können Aktionen innerhalb eines bestimmten Zustandes auslösen. Drei spezielle Ereignisse sind in der UML vordefiniert:

- entry löst die Aktion automatisch beim Eintritt in einen Zustand aus
- exit löst die Aktion automatisch beim Verlassen eines Zustandes aus
- do löst die Aktion immer wieder aus, solange das der Zustand aktiv ist

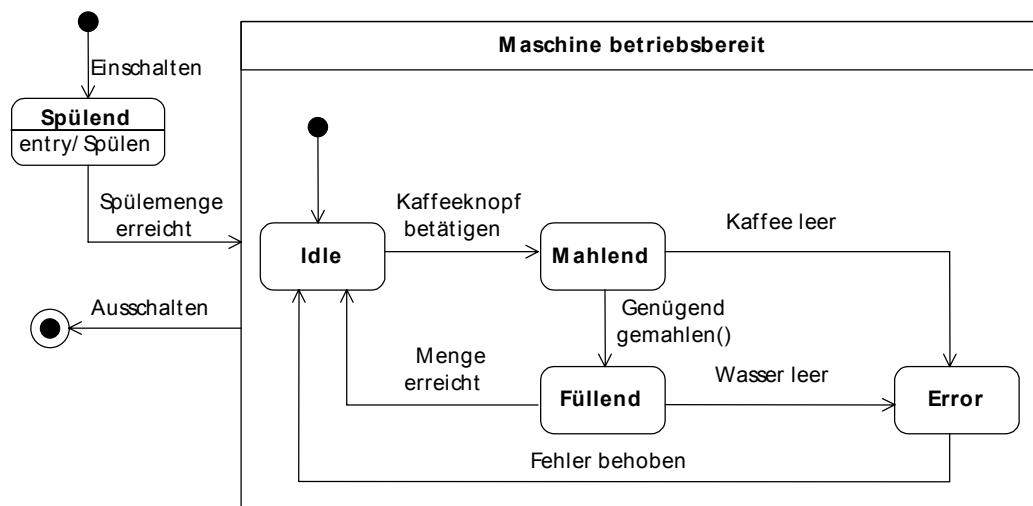


Abbildung 3-51 Darstellung Zustandsdiagramm Kaffeemaschine.

3.4.6 Sequenzdiagramm (Sequence Diagram)

Das Sequenzdiagramm zeigt die zeitliche Abfolge des Nachrichtenaustausches zwischen einer bestimmten Menge von Objekten. Im Grunde genommen zeigt ein Sequenzdiagramm den selben Sachverhalt wie ein Kollaborationsdiagramm. Allerdings ist bei letzterem der zeitliche Verlauf lediglich in Form einer Numerierung vorhanden, wogegen beim Sequenzdiagramm die Zeit eine feste Grösse in Form der vertikal angeordneten Achse einnimmt.

Die beteiligten Objekte aus dem Modell werden im Diagramm in der Horizontalen aufgereiht und mit einer senkrechten Lebenslinie ergänzt. Die jeweilige Aktivität eines Objektes wird auf der Lebenslinie mit einem dickeren Balken angedeutet.

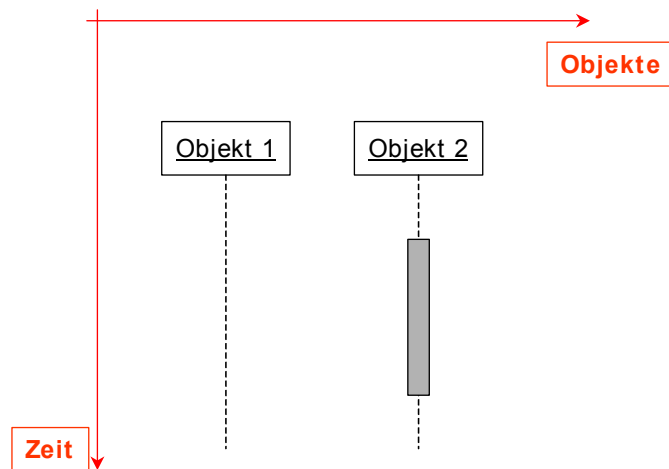


Abbildung 3-52 Achsen eines Sequenzdiagramms .

Die Aktivität eines Objektes ist abhängig von einer vorgängigen Aktivität eines andern Objektes. Die Abhängigkeit entsteht durch "Abfeuern" einer Nachricht vom einen zum andern Objekt.

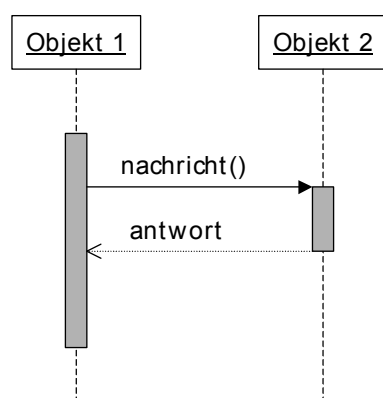


Abbildung 3-53 Abfeuern von Nachrichten und Antwort (Functioncall).

Ein Objekt kann sich auch selber aufrufen (Rekursion). Für die grafische Darstellung einer Rekursion wird eine Aktivität auf einer bestehenden Aktivität ein und desselben Objektes eingezeichnet.

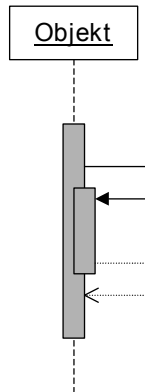


Abbildung 3-54 Darstellung einer Rekursion.

Je nach Zustand eines Objektes können unterschiedliche Funktionen eines andern Objektes angesprochen werden. Dieser Umstand kann im Sequenzdiagramm mit Bedingung und verschiedenen Aktionen dargestellt werden. Selbst Zusicherungen für das Zeitverhalten können eingezeichnet werden.

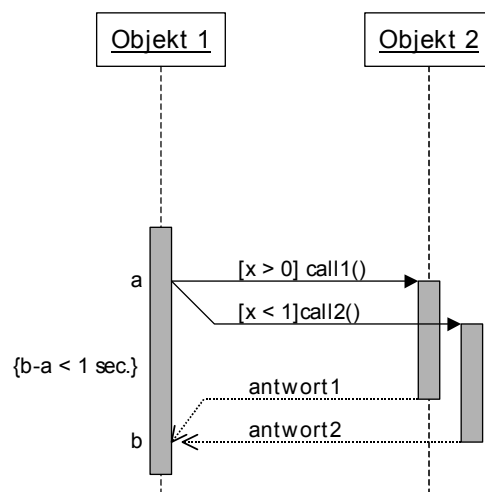


Abbildung 3-55 Darstellung einer Bedingung.

Wie bereits erwähnt, werden die betroffenen Objekte für eine Darstellung in der horizontalen Achse angeordnet. Insofern die Objekte dynamisch erzeugt werden, kann dies dadurch festgehalten werden, dass die erste Nachricht `<<create>>` direkt auf das Objekt zeigt (Objekt wird in der Zeitachse verschoben). Das dynamische Zerstören des Objektes wird in der Regel mit einer `<<destroy>>` Nachricht angedeutet und mit einem kreuzenden Durchstreichen der Aktion des betroffenen Objektes beendet.

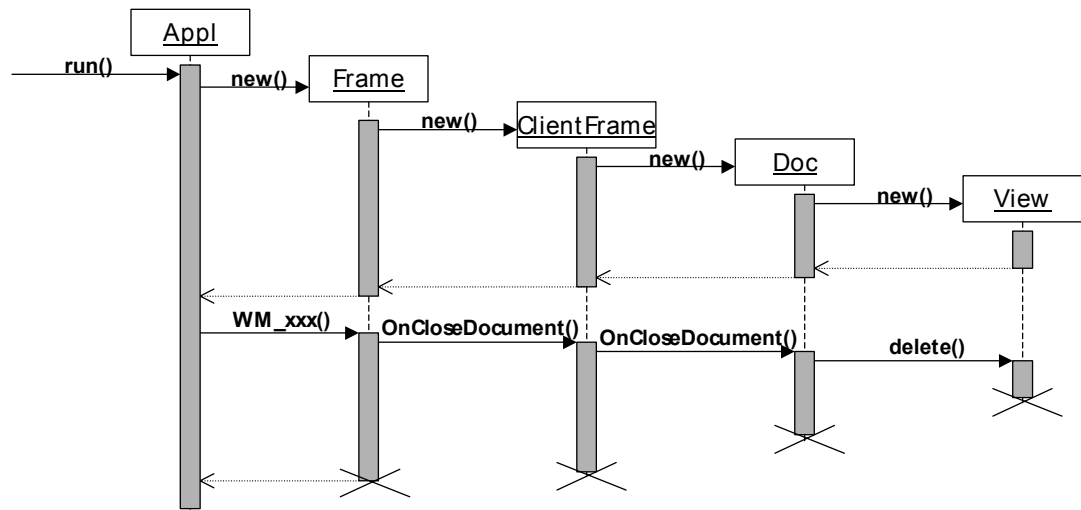


Abbildung 3-56 Darstellung einer ganzen Sequenz mit dynamisch erzeugten Objekten.

3.4.7 Kollaborationsdiagramm (Colaboration Diagram)

Das Kollaborationsdiagramm zeigt die Interaktion eines bestimmten Objektes zu den andern Objekten in Bezug auf einen bestimmten Kontext. Im Grunde genommen zeigt das Diagramm das selbe wie ein Sequenzdiagramm. Der Faktor Zeit spielt jedoch eine nebensächliche Rolle. Der Nachrichtenaustausch wird in seiner Reihenfolge nummeriert.

Die Beziehungen zwischen den Objekten bilden die Grundlage für den Nachrichtenaustausch. Die Beziehung wird in Form einer Assoziationslinie zwischen den Objekten eingezeichnet und kann mit folgenden Stereotypen (Sichtbarkeitsangaben) noch verfeinert definiert werden:

- <<asoziation>> Der Beziehung liegt eine Assoziation zugrunde.
Diese Definition gilt als Standard und kann weggelassen werden.
- <<global>> Das empfangende Objekt ist global
- <<local>> Das empfangende Objekt ist lokal in der sendenden Operation
- <<parameter>> Das mit diesem Stereotyp bezeichnete Objekt ist ein Parameter aus der empfangenen Nachricht.
- <<self>> Das empfangene Objekt ist das sendende Objekt.

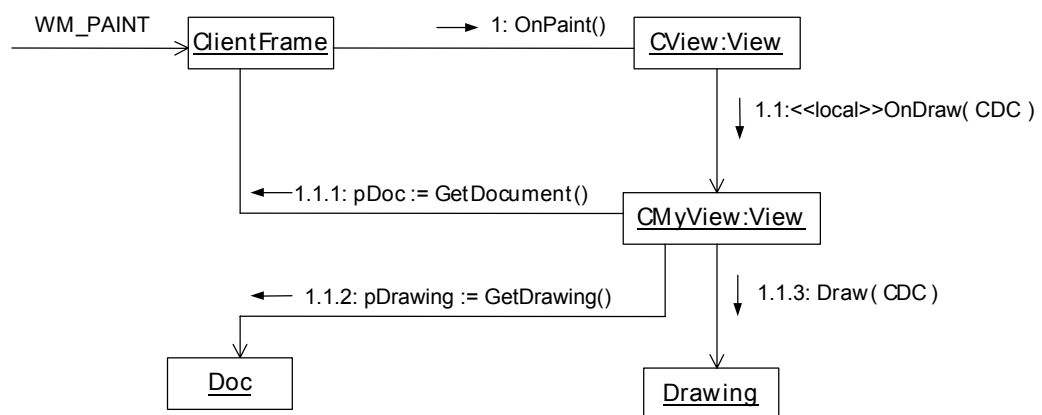


Abbildung 3-57 Kollaboration mit der MFC für das Zeichnen.

3.4.8 Komponentendiagramm (Component Diagram)

Das Komponentendiagramm ist ein Ordnungsdiagramm für das Aufzeigen der Zusammenhänge der Implementation. Die Anwendung ist abhängig von der gewählten Sprache für die Implementation. Insbesondere bei Sprachen die das Modulkonzept unterstützen ist das Komponentendiagramm praktisch für das Aufzeigen der Verhältnisse der verschiedenen Programmmodule.

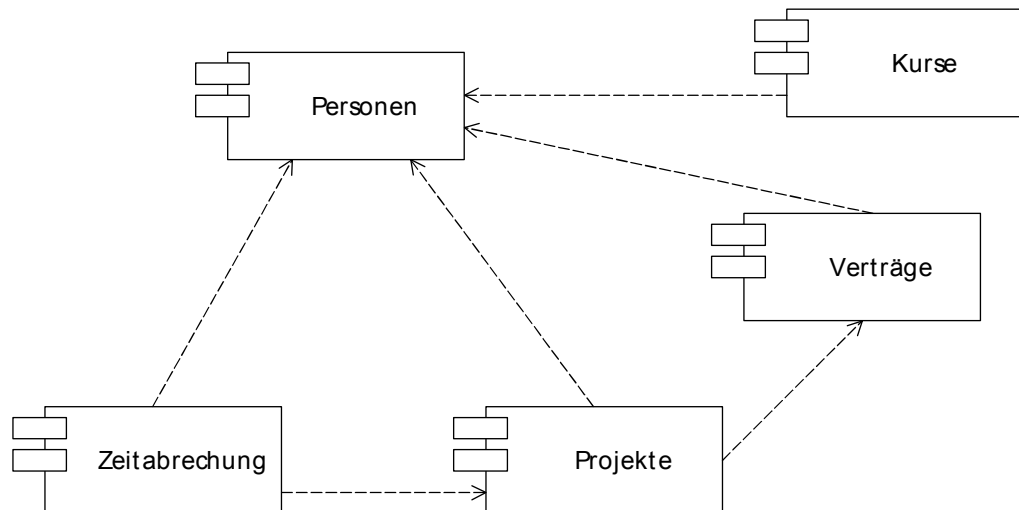


Abbildung 3-58 Darstellung eines Komponentendiagramms.

3.4.9 Verteilungsdiagramm (Deployment Diagram)

Das Verteilungsdiagramm zeigt die Grundkonfiguration der Lösung in Bezug auf die Maschinen. Die Methode UML verwendet zu diesem Zweck Quader. In der Praxis sind jedoch bessere Illustrationsmöglichkeiten gegeben, welche auch rege genutzt werden.

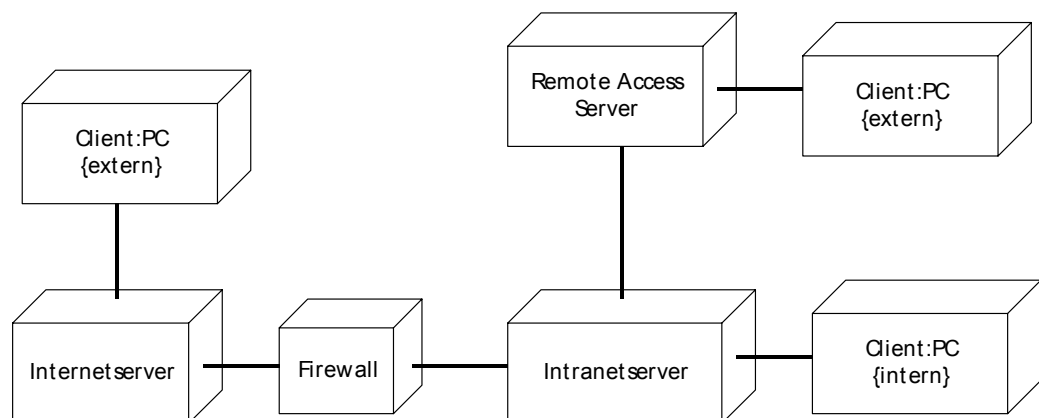


Abbildung 3-59 Darstellung eines Verteilungsdiagramms mit UML.

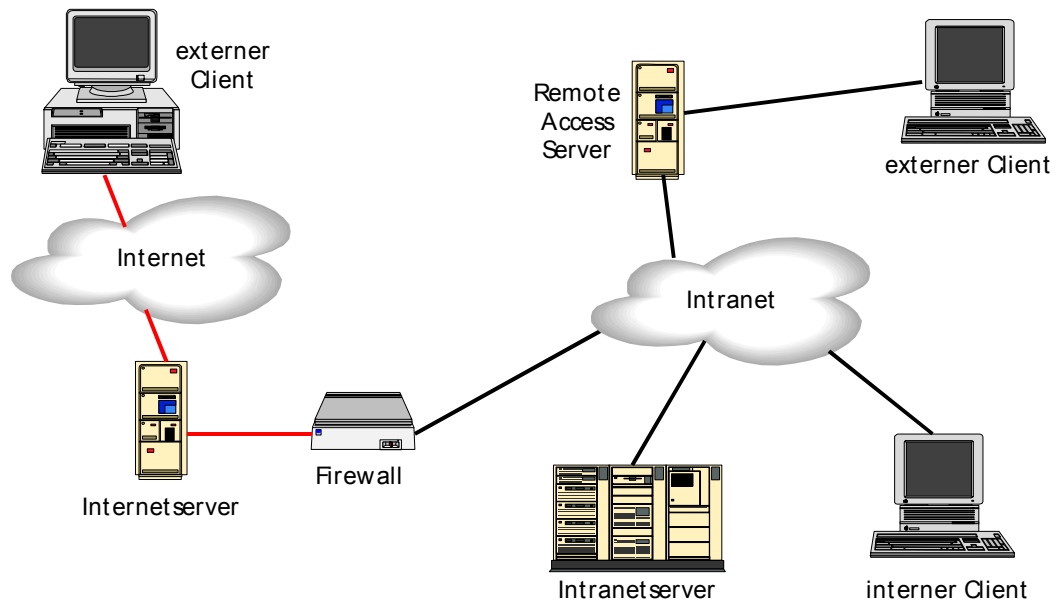


Abbildung 3-60 Darstellung eines Verteilungsdiagramms mit andern gebräuchlichen Illustrationsmitteln.

4 Vorgehensschritte des OOA und OOD

4.1 Positionierung im Gesamtprozess

Die Beschreibung des Gesamtprozesses für die Softwareplanung hat mit dem "Rational Unified Process" (kurz RUP (siehe [5])) eine noch nie dagewesene Dimension erreicht. Bis anhin waren in der Planung und Produktion zwei eigentliche "Lager" von ideologischen Vertretern auszumachen. Die Projektleiter als Vertreter des Lagers Projektmanagement hatten das Gefühl, dass mit einer allgemeinen Ausbildung jedes Projekt nach den gleichen Regeln abläuft, und somit ohne Fachwissen der Projektleitung zum erfolgreichen Abschluss gebracht werden kann. Daneben waren die Systemanalytiker und Designer als Vertreter der Softwareentwickler, welche für Teile des Prozesses Softwareplanung immer ausgeklügeltere Modelle, Notationen und Methoden für die korrekte Beschreibung des Ziels entwickelten.

Eigentlich gibt es noch das dritte Lager an Interessenvertretern, das ist das Management der Linienorganisation. Doch diese Vertreter bilden in unseren Betrachtungen als eigentliche externe Beteiligte eine "additive Konstante", die hier nicht weiter beschrieben werden soll.

In Tat und Wahrheit haben wohl die Projekte den grössten Erfolg, wo alle Beteiligten ein bisschen von ihrem puristischen Leben absahen, und gemeinsam ein Gesamtverfahren als Auswahl von verschiedensten Notationen und Methoden innerhalb des Projektes benutzt wurde.

Der Gesamtprozess wie er im Prozessmodell nach der RUP beschrieben ist, löst zwar die schwelenden Glaubenskriege nicht ganz ab (vermutlich entstehen noch neue), doch handelt es sich um einen sehr interessanten, ganzheitlichen Ansatz, der zum ersten Mal den beiden Kontrahenten Projektmanager und Systementwickler eine wirklich gemeinsame Basis vorgibt. Zwar muss jeder Vertreter im Projekt im Detail immer noch die althergebrachten Techniken beherrschen, weil das sogenannte "Handwerkszeug" nicht essentiell geändert hat, aber die verschiedenen Schritte können nun koordiniert ablaufen.

Auch die RUP ist natürlich kein Allerheilmittel für missratene Projekte. Der Prozess erfordert sehr viel Sachverstand und muss auf jedes Projekt in der Praxis sinnvoll angewendet werden.

Im Rahmen dieser Kursunterlage interessieren nun im wesentlichen die Phasen in welchen die Analyse und der Design versteckt sind. Zur einfacheren Lokation dieser Phasen habe ich in der nächsten Abbildung eine Zusammenstellung zwischen RUP, den allgemein konventionellen Begriffen, den entsprechenden Aktivitäten nach RUP und den einzelnen beteiligten Personen (Rollen) in den jeweiligen Phasen vorgenommen.

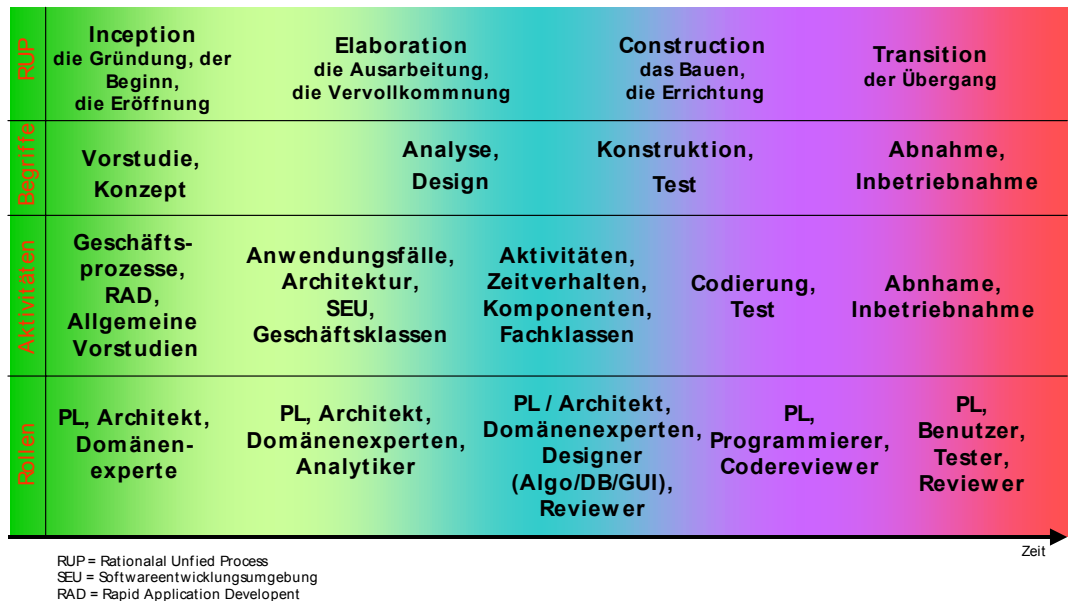


Abbildung 4-1 Ordnung von RUP, konventionellen Begriffen, Aktivitäten und Rollen.

Aus dieser Abbildung leite ich ab, dass folgende Aktivitäten in der Analyse zu erledigen sind:

- Konsolidierung der Anforderungen
- Erhebung der Anwendungsfälle
- Beschreibung der generellen Architektur der Lösung
- Definition eines ersten groben Klassenmodells (sog. Geschäftsklassen)

Die Analyseresultate (Modelle und Dokumentationen) überführen wir im Design mittels nachstehenden Aktivitäten in den Design über:

- Aktivitätenmodelle
- Definition der zeitlichen Reihenfolgen
- Komponentenbildung
- Klassenmodelle (Stufe Design)

Gemeinsam verfolgen Analyse und Design das Ziel der Beschreibung der zu konstruierenden Software in genügendem Detaillierungsgrad für die Konstruktion. Je nach Projekt gehören dazu:

- Klassenmodell
- Komponentenmodell
- DB-Modell
- GUI Design

Die Detaillierungstiefe der Analyse und des Designs lässt sich nicht determinativ bestimmen. Vielmehr ist das abhängig von den Kenntnissen der beteiligten Personen. Als Grundsatz kann jedoch gesagt werden, dass nur das festgehalten werden soll, was nicht für jeden Beteiligten selbstverständlich ist.

4.2 Teilprozesse OOA

Der Teilprozess Analyse beschreibt den Übergang vom Konzept (Vorstudie) und dient der Vorbereitung des Design. Das **WAS** bildet die zentrale Frage der Analyse. Im RUP ist das Analysemittel die sogenannte Anwendungsfallanalyse (engl. Use case).

4.2.1 Anwendungsfallanalyse

Zweck

Die Anwendungsfallanalyse beschreibt unter Einbezug der einzelnen Akteure, die Hauptfunktionen des Systems.

Input

Vorstudie, Konzept oder mindestens Pflichtenheft (bei kleinen Projekten) ist vorhanden

Output

Liste der Anwendungsfälle, pro Anwendungsfall ein Diagramm und eine Anwendungsfallbeschreibung sind erstellt.

Beteiligte Personen

Domänenexperte, Benutzer, Analytiker

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Anwendungsfälle identifizieren
- Erstellen einer Liste der Anwendungsfälle
- Reihenfolge der Verfeinerung definieren
- Erstellen der Anwendungsdiagramme
- Erstellen der Beschreibungen pro Anwendungsfall, Glossar führen
- Akteure in Akteurmodell überführen

4.2.2 Architektur und SEU Evaluation

Zweck

Beschreiben der künftigen Architektur der Lösung. Werden innerhalb des Projektes neue technologische Lösungsansätze verfolgt, kann in dieser Phase ein technischer Prototyp zur Erfahrung der angestrebten Technologie gefahren werden. Ist mit dem Einführen der neuen Technologie ein neues Entwicklungswerkzeug erforderlich, kann dieses im Rahmen des technischen Prototyps evaluiert werden (SEU = Softwareentwicklungsumgebung).

Input

Vorstudie, Konzept oder mindestens Pflichtenheft (bei kleinen Projekten) ist vorhanden.

Output

Architektur der Lösung ist beschrieben und SEU ist bekannt.

Beteiligte Personen

Systemarchitekt

Vorgehen

Das Festlegen der Architektur erfordert viel Erfahrung. Neben den rein technologischen Aspekten (wie könnte man es machen, wie würden das Problem wohl andere lösen), ist dabei unbedingt auch die Fähigkeit des Personals im Umgang mit der ausgewählten Technik zu beurteilen.

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Anforderungen an Architektur beschreiben
 - Applikationsarchitektur
 - Datentechnik
 - Systemtechnik
 - Kommunikationstechnik
 - Mengengerüst
- Mögliche Architekturen skizzieren
- Evaluation der Architektur
- Architekturreview (allenfalls mit externen Experten)
- Lösungsansatz fertig dokumentieren
 - Applikationsarchitektur
 - Datentechnik
 - Systemtechnik
 - Kommunikationstechnik
 - Mengengerüst
 - Auswirkungen auf die Rahmenorganisation

4.2.3 Geschäftsklassenanalyse

Zweck

Die Geschäftsklassenanalyse beschreibt die zur Lösung gehörenden Klassen auf einem groben Niveau. Die Granulierung ist so zu wählen, dass der Manager oder der Benutzer die Semantik noch versteht. Eine Geschäftsklasse wird später in Fachklassen weiter verfeinert und ist somit eigentlich ein virtuelles Gebilde (1 Geschäftsklasse kapselt 3..20 Fachklassen).

Input

Anwendungsfallanalyse und Architektur ist vorhanden.

Output

Erste Ordnung für das Klassenmodell ist vorhanden.

Beteiligte Personen

Domänenexperte, Benutzer, Analytiker (ev. Architekt)

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Klassen identifizieren
- Modell erstellen / beschreiben
- Modellreview

4.3 Teilprozesse OOD

Der Teilprozess Design beschreibt den Übergang von der Analyse und dient der Vorbereitung der Konstruktion. Das **WIE** bildet die zentrale Frage des Design.

4.3.1 Aktivitätsmodellierung

Zweck

Jeder Anwendungsfall wird in einem Aktivitätsdiagramm verfeinert. Im Gegensatz zum Anwendungsfall zeigt das Aktivitätsdiagramm dabei Abhängigkeiten, allfällige Nebenläufigkeiten sowie Entscheidungs- und Verzweigungspunkte.

Input

Anwendungsfallanalyse und Architektur ist vorhanden.

Output

Jeder Anwendungsfall ist verfeinert und sein Ablauf ist klar erkennbar.

Beteiligte Personen

Domänenexperte, Analyse- oder Designspezialist (evtl. Benutzer und Architekt)

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Erstellen eines ersten Diagramms aufgrund der Aktionsbeschreibungen im Anwendungsfall
- Evtl. Zuständigkeitsbereiche einzeichnen und Aktivitäten zuordnen
- Modellreview

4.3.2 Sequenz- oder Kollaborationsmodellierung

Zweck

Bei komplizierten Sachverhalten kann ein Sequenz- oder Kollaborationsdiagramm die Präzision eines Aktivitätendiagramms steigern. Dabei werden die einzelnen Aktivitäten (Aktionen und Reaktionen) der Objekte in einem Zeitdiagramm chronologisch dargestellt (Sequenz) oder die Aktivitäten zwischen den Objekten werden in der richtigen Reihenfolge bezeichnet (Kollaboration).

Input

Aktivitätendiagramm des Anwendungsfalles ist vorhanden

Output

Das Aktivitätendiagramm ist verfeinert dargestellt.

Beteiligte Personen

Analyse- oder Designer (evtl. Architekt)

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Erstellen eines ersten Diagramms aufgrund der Aktivitäten
- Modellreview

4.3.3 Fachklassenmodellierung

Zweck

Die Geschäftsklassen werden aufgrund der Erkenntnisse aus den anderen Modellen verfeinert, in Beziehung gesetzt und modelliert.

Wenn das Projekt mit einer Datenbank arbeitet, soll in dieser Phase die Abbildung in der Datenbank ebenfalls definiert werden. Dabei kann es durchaus zu einer Überführung des objektorientierten Modells in eine relationale Darstellung¹ kommen.

Input

Geschäftsklassen-, Anwendungsfallanalyse und Architektur sind vorhanden.

Output

Erste Ordnung für die Implementation der Klassen ist vorhanden.

Beteiligte Personen

Domänenexperte, Analytiker oder Designer (evtl. Architekt).

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Klassen identifizieren
- Klassen ordnen (Abhängigkeiten feststellen)
- Modell erstellen / beschreiben
- Modellreview

4.3.4 Komponentenbildung

Zweck

Die Aufgabe wird in verschiedene Teile (Komponenten) aufgegliedert. Die Komponenten sind die Einheiten die später gesamtheitlich codiert, getestet und betrieben werden sollen.

Input

Geschäftsklassenanalyse und Aktivitätsmodelle sind vorhanden.

Output

Die Komponenten sind vor allem funktionell definiert.

Beteiligte Personen

Domänenexperte, Designer (evtl. Architekt).

Vorgehen

¹ Wird für die Implementation in einer relationalen Datenbank verwendet.

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Fachlich zusammengehörende Teile identifizieren
- Modell erstellen / beschreiben
- Modellreview

4.3.5 Komponentenentwurf

Zweck

Währenddem die Komponentenbildung sich mehrheitlich an den funktionalen Aspekten der Lösung orientiert, berücksichtigt der Komponentenentwurf auch die statischen Sichten (d.h. Fachklassendiagramm) der Lösung. Durch dieses Zusammenführen kann es zu Korrekturen der Komponenten- und das der Fachklassenbildung kommen.

Input

Komponenten- und Fachklassenmodell 0² sind vorhanden.

Output

Komponentenbeschreibung und Fachklassenmodell 1.

Beteiligte Personen

Domänenexperte, Designer (evtl. Architekt).

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Klassen den Aktivitäten zuordnen
- Modell korrigieren
- Modellreview

4.3.6 GUI Design

Zweck

Der Design des GUI ist oft durch Prototyping aus vorangehenden Phasen geprägt. Das Prototyping des GUI hat den Vorteil, dass der Benutzer sich die Lösung besser Vorstellen kann und entsprechend auch seine Anforderungen genauer eruiert werden können.

Input

Anwendungsfallanalyse und Architektur ist vorhanden.

Output

GUI Grundsätze und wichtigste Masken sind definiert.

² Das erste Fachklassenmodell wird Modell 0 genannt. Als Resultat der Korrektur aus der Komponentenentwicklung entsteht das Fachklassenmodell 1.

Beteiligte Personen

Domänenexperte, Benutzer, GUI Designer (evtl. Architekt).

Vorgehen

Untenstehende Aufzählung für das Vorgehen ist nach Reihenfolge der Aktivitäten geordnet. Unabhängig davon gilt in sich natürlich auch hier das Prinzip der Iteration.

- Zusammentragen und ordnen allfälliger Prototyparbeiten
- Generelle GUI Richtlinien festlegen
- Grafische Definition komplizierter Dialoge und Verhalten
- evtl. State-Event Diagramme für Dialogverhalten aufzeichnen
- Review

5 Analyse

5.1 Anwendungsfälle (Use case)

5.1.1 Zusammenhänge

Die Begriffe Anwendungsfall, Akteur und Anwendungsfalldiagramm werden in der Analyse zusammen verwendet. Es bedeuten:

Der Anwendungsfall beschreibt eine Menge von Aktivitäten eines Systems aus Sicht eines oder mehreren Akteuren. Der Anwendungsfall führt den Akteur zu einem beobachtbaren und nützlichen Ergebnis. Zudem ist der Anwendungsfall zwar hierarchisch weiter verfeinerbar, jedoch hinsichtlich seiner Funktionen homogen.

Der Akteur ist eine Klasse, die ausserhalb des Systems liegt (Person, System, Hardware...). Sie ist an einer Interaktion, die durch einen Anwendungsfall beschrieben ist, beteiligt.

Das Anwendungsfalldiagramm fasst eine Menge von Anwendungsfällen und Akteuren zu einer übergeordneten funktionalen Einheit zusammen. Es zeigt die Beziehungen zwischen Anwendungsfällen und Akteuren wie sie zur Erledigung eines Geschäftsprozesses entstehen.

Das bedeutet, dass der Anwendungsfall eine endliche Menge von Aktivitäten aufdecken soll (determinatives Verhalten).

5.1.2 Vorgehen

Aus den Vorgaben wie Beschreibung der Geschäftsprozesse, RAD Beschreibung oder Pflichtenheft wird versucht, die eigentliche Funktionalität der Lösung zu erkennen. Die hauptsächliche Stütze dieses Prozesses bildet das Denken in Szenarien. Für das Definieren der Szenarien ist natürlich der Benutzer von grösster Wichtigkeit, denn nur er kann das System und sein Soll-Verhalten definieren.

Ich habe die Erfahrung gemacht, dass der Benutzer in der Situation jedoch oft überfordert ist. Deshalb muss der Entwickler hier mit geschickter Fragestellung und grossem Einfühlungsvermögen agieren. Fragestellungen wie

- Was geschieht wenn das und dies eintrifft?
- Was geschieht nach der Eingabe der Daten?
- Welche Regeln gelten bei der Eingabe der Daten?
- Wozu werden diese Daten benötigt?
- Wieso werden diese Daten benötigt?
- Wie entstehen die Daten, woher stammen die Daten, die eingegeben werden?
- Wer benötigt die eingegebenen Daten?

können helfen das Eis zwischen Benutzer und Entwickler zu brechen und die Aktionen, mit denen der Benutzer konfrontiert ist, herauszufinden.

Erstaunlicherweise habe ich übrigens auch festgestellt, dass ein versierter Informatiker genauso in Bedrängnis gerät, einen Sollzustand zu beschreiben für ein System das er für sich selber entwickeln will. Es kann also durchaus hilfreich sein, einen der Materie fremden Entwickler für die Definition beizuziehen.

Aus den Notizen der Beschreibungen durch den Benutzer formt der Entwickler erste Listen von Anwendungsfällen. Die Fälle werden nun in einzelne funktionell logisch zusammenhängenden Diagramme geordnet und dann in Form von Beschreibungen der Anwendungsfälle laufend detaillierter beschrieben.

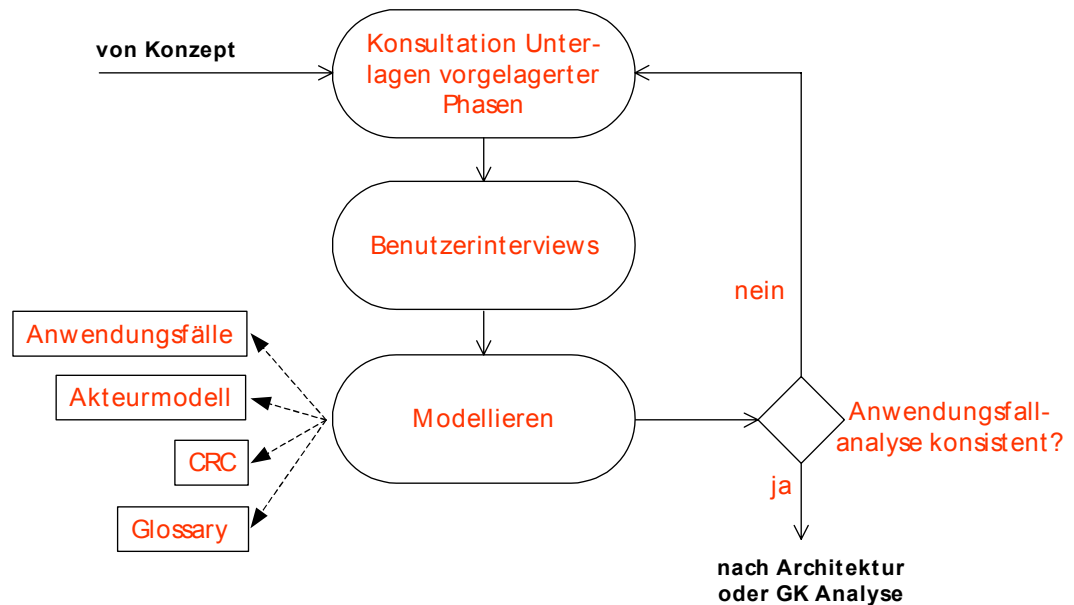


Abbildung 5-1 Tätigkeiten in der Analyse der Anwendungsfälle

Wichtig sind auch die scheinbaren Nebenprodukte, die bei diesen Arbeiten entstehen. Dazu gehört das Festhalten von benutzerorientierten Fachbegriffen. Sie zu verstehen bedeutet für den Entwickler einen grossen Teil der Lösung zu verstehen. Im weiteren entstehen hier erste Eindrücke für mögliche Kandidaten von Klassen. Diese Gedanken sollten auch festgehalten werden. Als mögliches Mittel eignen sich CRC Karten geradezu vorzüglich.

5.1.3 Notation

Siehe Kapitel 3 UML Basics.

5.1.4 Beispiel

Liste der Anwendungsfälle für das Zeichnungsprogramm

Das Zeichnungsprogramm beinhaltet folgende Anwendungsfälle

- Eine neue Zeichnung eröffnen
- Zeichnungsdaten verändern
- Persistenz
- Drucken

Das Anwendungsfalldiagramm: Neue Zeichnung eröffnen

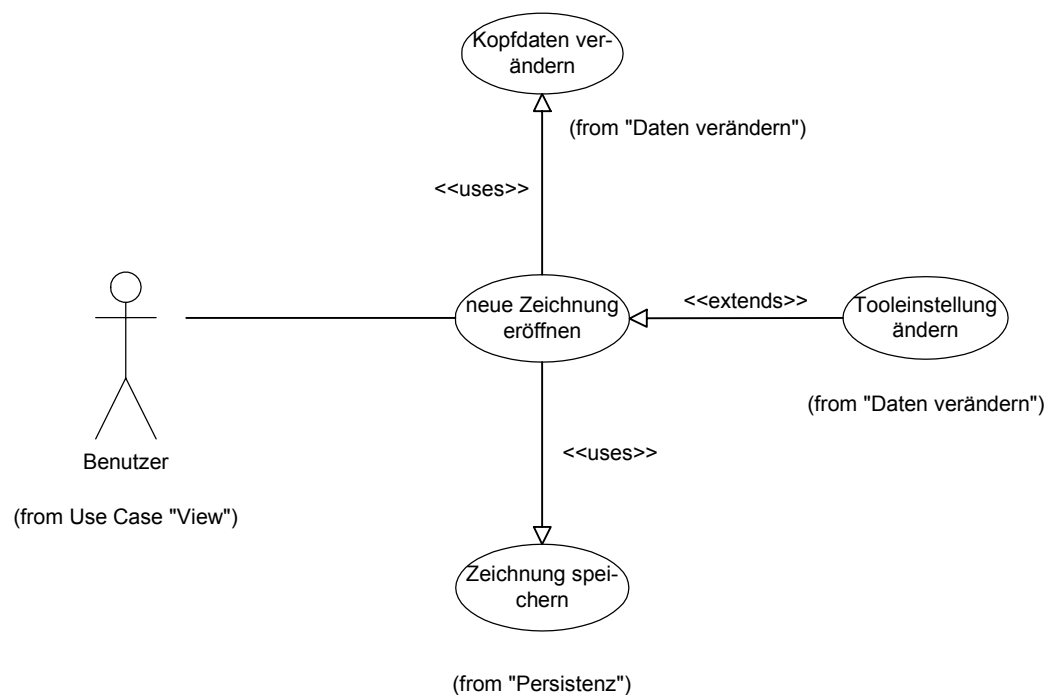


Abbildung 5-2 Anwendungsfalldiagramm erstellt mit Visio 2000.

Die Anwendungsfallbeschreibung: Neue Zeichnung eröffnen

Name:	Neue Zeichnung eröffnen
Nummer:	1
Kurzbeschreibung:	Eine neue Zeichnung wird eröffnet, die Grunddefinitionen werden eingegeben und auf dem Disk gespeichert.
Beteiligte Akteure:	Benutzer (alle Typen)
Auslöser / Vorbedingung:	Keine speziellen
Ergebnisse / Nachbedingung:	Die Zeichnung ist bearbeitbar und leer gespeichert

Ablauf:

Nr.	Wer	Was
1.1	Benutzer	Erstellen neue Zeichnung über Menü "New"
1.2	Benutzer	Eröffnen Dialog für Kopfdaten
1.3	Benutzer	Eingabe der Kopfdaten
1.4	Benutzer	Dialog beenden
1.5	System	Daten nach Regeln prüfen
1.6	Benutzer	Daten "Save" auslösen
1.7	System	Dialog "Save as..." anzeigen
1.8	Benutzer	Eingabe Filename und speichern auslösen
1.9	System	Filenames plausibilisieren
1.10	System	Datei erstellen

Ausnahmen, Varianten:

Nr.	Wer	Was
1.5	System	Bei falschen oder fehlenden Eingaben Korrektur verlangen
1.6	Benutzer	Daten "Save as..." auslösen
1.9	System	Bei fehlerhaftem Dateinamen im Dialog verweilen und Benutzer zur Korrektur auffordern.

5.1.5 Fallen

Die Beschreibung von Anwendungsfällen birgt in sich folgende Fallen:

- Der Anwendungsfall beschreibt nicht wie etwas gemacht wird, sondern was das System oder mit Hilfe des Systems erreicht werden soll. Keine detaillierte funktionale Zerlegung.
- Anwendungsfälle, die nicht von einem Akteur angestoßen oder zu keinem Resultat für einen Akteur führen, sind oft falsch oder unvollständig.
- Anwendungsfälle, die Aktivitäten ausserhalb der Systemgrenzen enthalten, sind nicht korrekt organisiert.

5.1.6 Aufwand

Es ist sehr schwierig eine zeitliche Limite für das Beschreiben eines Anwendungsfalls zu definieren. Werte zwischen ½ ..3 h pro Anwendungsfall dürfte aber dennoch eine würdige Annahme sein.

5.2 Akteure

5.2.1 Vorgehen

Die Akteure werden im Rahmen der Definition der Anwendungsfälle und der Anwendungsfalldiagramme entstehen. Bei Systemen mit mehreren Akteuren kann es wertvoll sein, die Beziehungen zwischen den Akteuren zu studieren. Da die Akteure dabei wie Klassen behandelt werden, kann nach dem Verfahren für die Definition von Klassen vorgegangen werden (siehe Kapitel Fachklassen).

Folgende grundsätzlichen Fragestellungen können den Einstieg in diesen Teil der Analyse erleichtern:

- Ist der Akteur eine Maschine, ein Prozess, ein Mensch?
- Welche Beziehungen zwischen Akteuren gleichen Typs arbeiten zusammen (Organisations- oder geographische Strukturen)?
- Wie ist die Verfügbarkeit maschineller Akteure?

Wenn es sich bei den Akteuren um Menschen handelt, können zudem folgende Fragestellungen von Bedeutung sein:

- Haben alle Akteure die gleichen Rechte für den Zugriff auf die Objekte?
- Sind zwischen den verschiedenen Akteuren spezielle Verhältnisse wie two phase commit o.ä.¹
- Sind zwischen den verschiedenen Akteuren Stellvertretungen definiert.

5.2.2 Notation und Beispiel

Siehe Kapitel 3 UML Basics.

5.2.3 Fallen

Die Beschreibung von Akteurdiagrammen birgt in sich folgende Fallen:

- Als Akteure werden aufgrund des UML Symbols nur menschliche Wesen verstanden. Dies ist falsch. Ein Akteur kann ein Mensch, eine Maschine oder ein Prozess sein.

5.2.4 Aufwand

Zwischen 2..16 h für die gesamte Beschreibung der Akteure dürfte eine würdige Annahme sein.

¹ Ein two phase commit ist eine spezielle Art der Zusicherung eines Einverständnisses. Alle Beteiligten in einem Entscheidungsprozess prüfen zuerst die Sache, sichern sich dann gegenseitig das Einverständnis zu, und wenn alle Einverständnisse vorliegen wird die Aktion ausgelöst (Bsp. Standesbeamte und Braut und Bräutigam)

5.3 Architektur und SEU Evaluation

Das Bestimmen der Architektur einer Applikation gehört zu einer der schwierigsten Aufgaben in einem Softwareentwicklungsprozess. Dazu gehört nicht nur viel Erfahrung, sondern auch Kenntnisse über Markttrends und Produkte.

Insofern das Projekt in einem grösseren Umfeld² realisiert werden soll, kommen dazu auch noch die Kenntnisse über die Firmenstrategie (Teilstrategie Informatik und Organisation) und vor allem auch die Kenntnisse über das Wissen des eigenen Personals. Insbesondere der letzte Punkt wird bei der Beurteilung einer Architektur leider oft zuwenig berücksichtigt. Dies führt jedoch unweigerlich zu gestressten Mitarbeiter/Innen, nicht erfüllbaren Projektplänen und oft sogar zum Projektabsturz.

Die Applikationsarchitektur beschreibt den grundsätzlichen Aufbau des Zusammenspiels zwischen Systemen (Hardware), den Betriebssystemen mit ihren Kommunikationen und natürlich der verschiedenen Komponenten der eigentlichen Lösung.

Neben der Bestimmung der eigentlichen Architektur gehört in das gleiche Thema die Evaluation der Entwicklungsumgebung. Es können nicht mit allen Entwicklungsumgebungen (EU) alle Architekturen abgebildet werden (oder mindestens nicht gleich gut). Dies hat zur Folge, dass mit dem Entscheid, welche Architektur benutzt werden soll, eine eventuell existente EU auf ihre Tauglichkeit hin geprüft werden muss. Ist noch keine EU vorhanden, kann eine Marktstudie mit anschliessendem Evaluationsprozess helfen die Umgebung zu definieren (Ausbildung der Entwickler bitte nicht vergessen).

Welches ist nun aber die richtige Architektur für die Lösung meines Problems? Als bodenständiger, schweizerischer Softwareentwickler ist darauf meine Frage natürlich: Es kommt darauf an! Worauf kommt es an. Ganz einfach - auf folgende Kriterien:

- Arbeitet nur ein Benutzer mit der Anwendung?
- Arbeiten 10, 100, 1000, 10000 oder mehr Benutzer mit dem System?
- Ist ein Internetanschluss erforderlich?
- Handelt es sich um eine technische oder eine kommerzielle Applikation?
- Sind Spezialgeräte anzusteuern?
- Entstehen die Daten zentral oder dezentral?
- Handelt es sich um eine datenorientierte oder eine grafikorientierte Applikation?
- Ist es eine neue Applikation, oder löst Sie ein bestehendes System ab?
- Sind "legacy-Systeme"³ einzubinden?
- Welche Kommunikationsstandards sind einzuhalten (Strategie)?
- Auf welchen Basissystemen muss die Lösung funktionstüchtig sein (OS, DB)?
- Sind zeitkritische Berechnungen (Laufzeit) durchzuführen?
- Sind mengenmässig kritische Berechnungen (Datenmenge) durchzuführen?
- Wie gross ist die geforderte Verfügbarkeit der Lösung (24h Betrieb, Max. Ausfalldauer, Max. Ausfallhäufigkeit)?
- Betreiben wir die Lösung selber, oder wird die Lösung nach der Erstellung von Dritten betrieben?
- Ist Transaktionsverhalten nötig?
- Wie gross sind die Sicherheitsanforderungen?
- Bestehen zyklische Verarbeitungsschritte (Tagesendverarbeitung, Wochenendverarbeitung...)?
- Wie ist die Einstellung meiner Entwickler?
- Wie ist das KnowHow meiner Entwickler?
- Wie ist die Einstellung des Management?
- In welchem Umfeld findet das Projekt statt?
- Machen wir alles selber, oder soll möglichst viel zugekauft werden?
- Wie gross ist die Lebensdauer der Daten?
- Ist die Menge der Daten pro Zeiteinheit konstant?

² Grossunternehmung oder Entwicklung für den Markt

³ engl. the legacy das Vermächtnis, die Erbschaft. Gemeint sind bestehende Systeme, die weiter betrieben werden müssen und zu denen vom neuen System aus eine Schnittstelle besteht.

So, einige wichtige Kriterien für grössere System kennen Sie nun. Es fehlt nun nur noch die Tabelle, in welcher die Kriterien mit den verschiedenen Architekturen in Verbindung bringt und schon ist alles klar. Leider ist das nicht so einfach, da in den Anforderungen einige widersprüchliche Gegebenheiten existieren können (in der Praxis auch tatsächlich vorkommen), lassen sich keine eindeutigen Aussagen machen. Nun, ich sagte es bereits: Es kommt eben darauf an!

5.3.1 Mögliche Architekturen

Ich will trotzdem versuchen, ein paar Architekturen aufzuzeigen, und deren Stärken und Schwächen zu nennen.

5.3.1.1 Standalone Application - SDI

Definition

Eine Applikation die auf einem Rechner losgelöst von andern Applikationen abläuft. Mit der Applikation wird zu einem Zeitpunkt eine Information geladen und bearbeitet.

Verwendung

PC oder workstationbasierte Programme für spezielle Aufgaben im technischen oder kommerziellen Bereich. In der Regel kaum kritische Verfügbarkeit gefordert und auch keine datenbankbasierte Anwendungen. Heutzutage oft mit netzwerkfähigem Datenzugriff auf Server ausgerüstet, das typische Modell für individuelle Anwendungen. Tool Charakter.

Stärken

- Einfach programmierbar
- Von vielen Entwicklungsumgebungen unterstützt
- Sehr oft volle grafische Unterstützung
- Für verschiedenste Plattformen baubar
- Fein verbreitetes Know How bei den Entwicklern und Usern
- Wenig technische Voraussetzungen für Umsetzung nötig

Schwächen

- Nicht automatisch netzwerkfähig
- Kein Multiuserbetrieb
- Datensicherung muss organisiert werden (Wenn Daten im LAN, einfache Lösung)
- Verfügbarkeit entsprechend der verwendeten Umgebung
- Datenzugriffssicherheit

5.3.1.2 Standalone Application - MDI

Wie Standalone Application - SDI, jedoch erlaubt diese Variante die Bearbeitung von mehreren Informationen zu einem bestimmten Zeitpunkt.

5.3.1.3 Client - Server

Definition

Anwendung mit funktioneller Aufteilung der Verarbeitung auf mehrere Komponenten. Je nachdem wo der Schnitt für die Aufteilung zwischen den Komponenten gemacht wird, entsteht ein mehr oder weniger kompliziertes Modell.

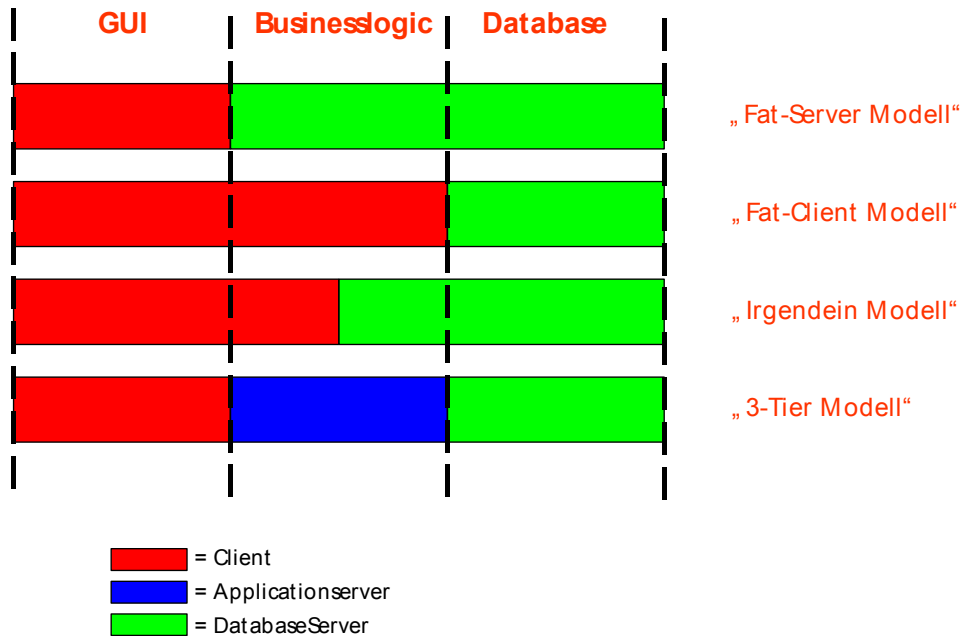


Abbildung 5-3 Client Server Architekturen Prinzipübersicht.

Verwendung

PC, Workstation oder auf zentralen Rechnern basierte Programme für Aufgaben im Datenbankbereich. Oft kritische Verfügbarkeit gefordert. Netzwerkfähiger Datenzugriff auf Server standard, in der Regel transaktionsorientiert. Das typische Modell für Datenverarbeitung. Anwendungscharakter.

Stärken

- Automatisch netzwerkfähig
- Multiuserbetrieb
- Von vielen Entwicklungsumgebungen unterstützt
- Dialogorientierte Systeme sind standard
- Für verschiedenste Plattformen baubar
- Datensicherung in der Regel systembedingt organisiert
- Datenzugriffssicherheit

Schwächen

- Nicht einfach programmierbar
- Verfügbarkeit entsprechend der verwendeten Umgebung
- Viele technische Voraussetzungen für Umsetzung nötig

5.3.2 Notation und Beispiel

Als Notation für die Architektur wird in der Regel ein Komponenten- oder Verteilungsdiagramm verwendet (siehe Kapitel 3 UML Basics). Das Diagramm ist mit entsprechenden technischen Beschreibungen zu ergänzen.

Bei der Definition von Schnittstellen ist darauf zu achten, dass diese nicht nur erwähnt, sondern dass sie beschrieben werden und vor allem auch, dass die Zuständigkeit für die Schnittstellen bekannt ist.

5.3.3 Fallen

Die Definition der Architektur birgt in sich folgende Fallen:

- Die ungenaue Definition der Architektur, führt später zu unliebsamen Überraschungen
- Die Wahl einer ehrgeizigen (top modernen, noch nie dagewesenen) Architektur kann das vorhandene Personal überfordern. Zeitverlust, Demotivation kann die Folge sein.
- Das Management wird falsch eingeschätzt. Modernste Architektur mit konservativ eingestelltem Management verträgt sich nicht (Hosti contra PC-Freak) oder umgekehrt!
- Alles mit dem PC machen wollen ist nicht gut (PC = Personal Computer). Eine Maschine ist in der Grösse richtig nivelliert, wenn Sie das erste Betriebsjahr ohne Ausbau übersteht.
- Falsch eingeschätzte Datenmengen führen später zu Performanceproblemen.
- Falsch eingeschätzte Zugriffssicherheit führt später vor den Kadi.

5.3.4 Aufwand

Wie bei den Anwendungsfällen ist es sehr schwierig, eine zeitliche Limite für das Beschreiben einer Architektur zu definieren. Werte zwischen 1..5 Arbeitstage scheinen mir jedoch angemessen.

5.4 Klassenmodell der Analyse

5.4.1 Zusammenhänge

Das Klassenmodell der Analyse definiert eine erste Näherung der künftigen Lösung. Die eingezeichneten Klassen werden später selten implementiert. Vielmehr handelt es sich um Klassen in einem erhöhten Abstraktionsniveau. Klassen, die von Benutzern und Managern gleichermassen verstanden werden und in Ihrer Menge noch überschaubar sind⁴.

⁴ In der Literatur und einigen Entwicklungswerkzeugen spricht man beim Klassenmodell der Analyse auch vom sog. Businessmodel (Geschäftsklassen).

5.4.2 Vorgehen

Aufgrund der Anwendungsfälle, allfällig bis dahin geführten CRC, Prototypen oder Beschreibungen des alten Systems, werden Klassen identifiziert. Zuerst wird dabei nicht auf das Abstraktionsniveau geachtet, sondern es werden alle Tatsachen zusammengestellt. In einem weiteren Schritt werden die gefunden Klassen meist nach funktionellen Aspekten gruppiert. Für die Gruppen werden entsprechende Oberbegriffe gesucht und diese werden als Geschäftsklassen definiert. Nun werden erste Beziehungen (Assoziationen), meist ohne genaue Definition, eingezeichnet. Das Klassenmodell soll alle Klassen in etwa niveaugleich aufzeigen.

5.4.3 Notation und Beispiel

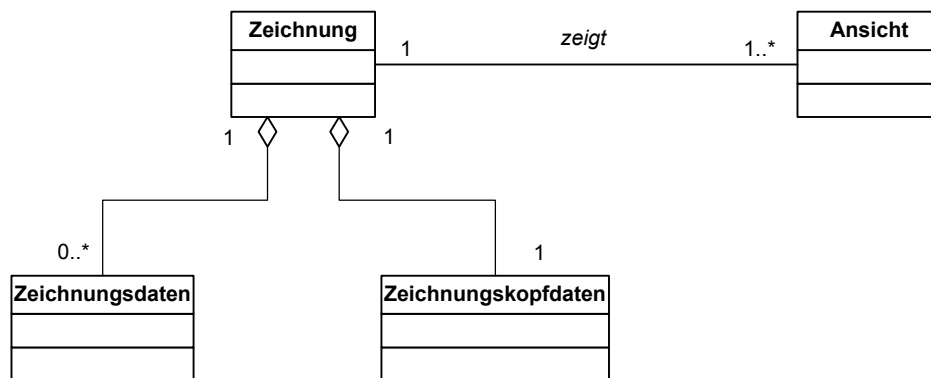


Abbildung 5-4 Klassendiagramm der Analyse gezeichnet mit Visio 2000.

5.4.4 Fallen

Die Beschreibung von Klassen in der Analyse birgt in sich folgende Fallen:

- Ungleiche, zu oberflächlich oder zu tiefe Detaillierung. Dies wirft beim Leser der Analyse meist unnötige Fragen auf.
- Unsachgemässe Begriffswahl der Klassen. Keine technischen sondern benutzerorientierte Namen für die Klassen verwenden.
- Zu detaillierte Definition der Assoziationen. Diese sind in aller Regel noch nicht genau definierbar (vergebene Mühe).

5.4.5 Aufwand

Zwischen 2..16 h für die gesamte Beschreibung des Modells dürfte eine würdige Annahme sein.

6 Das Design

6.1 Aktivitätenmodell

6.1.1 Vorgehen

Das Erarbeiten der Aktivitätenmodelle bildet den nächsten Schritt der Verfeinerung des Projektes. Hier entstehen detaillierte Abläufe wie Daten in der Interaktion zwischen dem System und den Akteuren bearbeitet werden.

Es wird jeweils pro Anwendungsfall ein Aktivitätendiagramm erstellt. Dabei können zunächst die Teilschritte aus der Beschreibung des Anwendungsfalles in Aktivitäten umgesetzt werden. Anschliessend können die Aktivitäten verfeinert, gruppiert, serialisiert oder parallelisiert werden. Die Notation der Aktivitätendiagramme erlaubt diese Möglichkeiten.

Der Schritt der Modellierung in den Aktivitätendiagrammen ist sehr wichtig. Dient er doch gleichzeitig der Erkennung von gleichartigen Aktivitäten über mehrere Anwendungsfälle und somit der Reduktion von Redundanzen.

6.1.2 Notation und Beispiel

Siehe Kapitel 3 UML Basics.

6.1.3 Fallen

Die Beschreibung von Aktivitäten birgt in sich folgende Fallen:

- Zu feine Definition der einzelnen Aktivität. Man achte darauf, dass eine Aktivität einem Arbeitsschritt entspricht und nicht die einzelnen Computeranweisungen repräsentiert.
- Aktivitäten mit unklaren Bezeichnern versehen. Prägnanten Namen bestehend aus Substantiv und Verb fördern die Verständlichkeit.
- Unbeschriftete Verzweigungen. Verzweigungen sollen mit Bedingungen dekoriert werden. Dies hilft Missverständnissen vorzubeugen.

6.1.4 Aufwand

Der Aufwand für das Erarbeiten von Aktivitätendiagrammen beläuft sich auf 2..8 Stunden pro Diagramm.

6.2 Klassenmodell des Designs

6.2.1 Zusammenhänge

Das Erarbeiten der Klassenmodelle stellt einen Höhepunkt im Design einer Lösung dar. Hier werden alle bisher erarbeiteten Resultate zusammengeführt und die eigentlichen "Arbeitstierchen" modelliert.

Der erste Lösungsansatz wird Klassenmodell 0 getauft. Aus diesem Ansatz werden die Komponenten definiert (Berücksichtigung der Architektur) und anschliessend wird das Modell zweiter Näherung das sog. Klassenmodell 1 definiert. Bei der Verwendung von relationalen Datenbanken wird in dieser Phase auch der Übergang in das Entitätenmodell notwendig sein.

6.2.2 Klassenmerkmale

6.2.3 Wie identifiziere ich Klassen

Im Unterschied zu den Geschäftsklassen, die sehr oft intuitiv entstehen, nicht implementiert werden und deshalb auch nicht unbedingt vollkommen sein müssen, stellen sich bei den eigentlichen Klassen der Lösung folgende Fragen:

- Welche Klassen brauche ich?
- Welches sind die richtigen Klassen?
- Welches sind die richtigen Beziehungen zwischen den Klassen?

Es gibt leider keine Kochbuchregeln die garantieren, dass ein Klassendesign problemlos umsetzbar und betreibbar ist. Jedoch kann man obigen Fragestellungen mit Methodik begegnen, so dass mit Wille und Fleiss gute Lösungen entstehen.

In jedem Fall aber sollte man die untenstehenden Bemessungsregeln für Klassen benutzen:

Eine Klasse repräsentiert das Muster für die Kreierung von Objekten gleichen Typs. Zum Muster gehören Variablen (Attribute) sowie Methoden (Operationen). Variablen bestimmen den Status und Methoden das Verhalten von Objekten.

Konkrete Klassen übertragen das Muster auf die instanzierten Objekte.

Abstrakte Klassen übertragen das Muster auf konkrete, vererbte Klassen.

Ein Objekt gehört zu genau einer Klasse.

In der Regel werden für das Finden der Klassen die besten Resultate erzielt, indem mehrere der nachfolgend beschriebenen Methoden angewendet werden.

6.2.3.1 Die Textanalyse

Die Textanalyse ist eine einfache aber recht wirkungsvolle Methode. Sie kann aufgrund von geschriebenen Texten, bei erfahrenen Entwicklern aber auch aufgrund von gesprochenen Texten durchgeführt werden.

Der Entwickler durchsucht den Text systematisch nach Klassenkandidaten, indem er die Substantive lokalisiert. Die dazugehörigen Adjektive deuten auf Attribute und die Verben auf Operationen der Klasse.

Diese Art von Klassenfindung ist für alle Anwendungsgebiete geeignet.

6.2.3.2 Die Realitätsbeobachtung

Die Methode der Realitätsbeobachtung basiert auf dem Aufbau von Szenarien für real existente Objekte und deren Beziehung untereinander. Anstelle der schriftlichen Beschreibung der Objekte, werden direkt Objektdiagramme gezeichnet, welche wiederum in die allgemeingültige Form als Klassendiagramm transformiert werden.

Diese Art von Klassenfindung ist für alle Anwendungsgebiete geeignet, die ein spontanes, objektorientiertes Denken des Benutzers ermöglichen.

6.2.3.3 Benutzersichtanalyse

Bei der Benutzersichtanalyse versucht der Entwickler in die Rolle des künftigen Benutzers zu schlüpfen. Er analysiert dabei Formulare, bestehende Bildschirmmasken, Belege, Listen usw. Das Erkennen von Klassen aus den Daten geschieht durch sinnvolle Gruppierung von zusammengehörenden Informationen. Die einzelnen Informationen auf den Benutzerformularen können meist als Attribute identifiziert werden. Auch kann der textuellen Darstellung auf Formularen oft eine Gruppierung der Informationen (Hinweise auf Klassen) entnommen werden.

Die Benutzersichtanalyse finden wir vornehmlich im Bereich von verwaltungstechnischen Problemstellungen (Wirtschaftsinformatik).

6.2.3.4 CRC Methode

Die CRC Methode (CRC steht für "Classes, Responsibilities, Collaborations") ist ein Hilfsmittel zum kontinuierlichen Festhalten von Informationen über potentielle Klassenkandidaten. Auf den CRC Karten sind neben dem Verwendungszweck der potentiellen Klassen auch ihre Verantwortungen (engl. Responsibilities = Attribute und Operationen) und die Zusammenarbeiten mit anderen Objekten (oder Klassen) aufgeführt.

Wenn die CRC Karten nicht mehr spontan mit neuen Erkenntnissen ergänzt werden können, werden Sie nach logischen Kriterien geordnet. Aus den verschiedenen Ordnungen können oft mehrere Modelle abgeleitet werden. Das Designteam wird sich für die einfachste und am besten passende Lösung entscheiden, und sie systematisch zum Modell 0 entwickeln.

Workingname: Line	
Possible Names: Linie, Line, Strich, Gerade	
Effective Name: CSWS_Line	
Purpose: Kapselt eine Gerade (geometrisch definiert).	
Responsibilities	Collaborations
• kennt ihre Position • kennt ihre Farbe • kennt ihre Strichstärke • kennt ihre Linienart • kann sich selber darstellen • ist persistent • kann verschoben werden • kann um einen beliebigen Zentrumspunkt rotiert werden • kennt die mit sich verknüpften Punkte	• benutzt den Devicekontext beim Zeichnen • benutzt die Persistenz der Klassenbibliothek XY • wird mit Maus auf dem Bildschirm interaktiv erzeugt • ist mit der Maus auf dem Bildschirm aktivierbar • ist eine von vielen möglichen Figuren • gehört zu einer Zeichnung • kann mit einer weiteren Figur verknüpft werden

Abbildung 6-1 CRC Karte wie sie während der Analyse entstehen könnte.

6.2.4 Modellieren oder wie erkenne ich Ordnungsstrukturen

Nach dem Vorgang der Klassenidentifikation ist eine erste Auswahl von Klassen mit mehr oder weniger Beziehungen und Strukturen vorhanden. Nun setzt die eigentliche Modellierung der Klassen ein. Darunter versteht man die gezielte Eliminierung von Redundanzen und die klare Definition von Beziehungen.

6.2.4.1 Generalisierung und Spezialisierung

Allgemeine Betrachtung

Beim Vorgang der Generalisierung werden gleichartige Klassen nach Gemeinsamkeiten untersucht. Werden solche erkannt, werden die Klassen zunächst isoliert und anschließend wird ein gültiger Oberbegriff der Klassen mit den Gemeinsamkeiten definiert. Dieser Oberbegriff wiederum bildet eine Oberklasse zu den isolierten Klassen. Die Gemeinsamkeiten können nun in die Oberklasse überführt und in den betroffenen Unterklassen eliminiert werden.

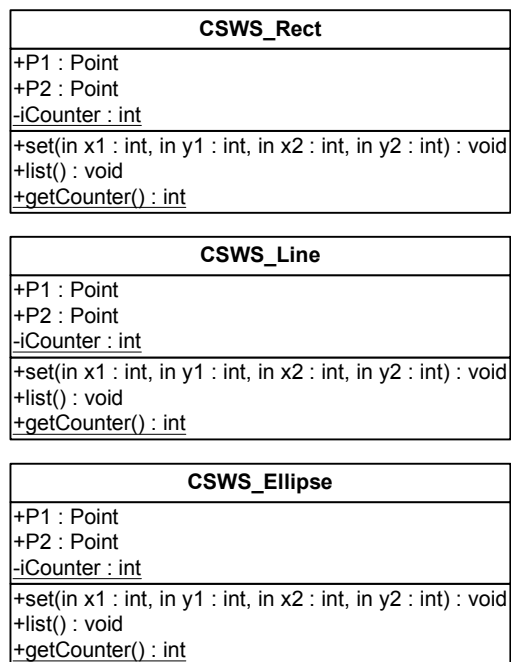


Abbildung 6-2 Mehrere zusammengehörende Klassen mit gemeinsamen Eigenschaften.

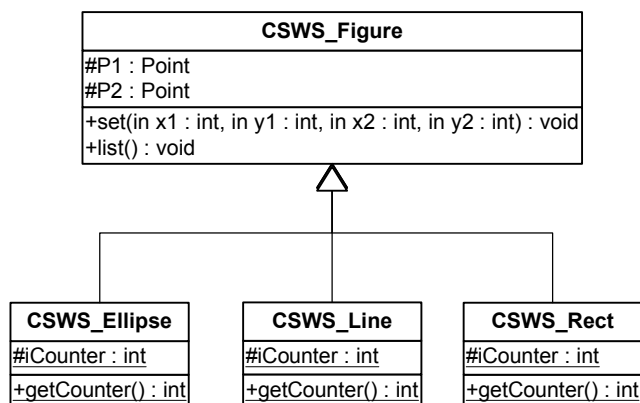


Abbildung 6-3 Neu eingeführte Oberklasse, welche die gemeinsamen Eigenschaften nun den Unterklassen zur Verfügung stellt (vererbt).

Beim Vorgang der Spezialisierung wird überlegt, ob eine Klasse in detailliertere Unterklassen zerlegt werden muss oder kann. Die Entscheidung wird im wesentlichen durch folgende Frage beeinflusst: Gibt es Objekte der betreffenden Oberklasse, die mit den definierten Informationen und Funktionen nicht determinativ beschrieben werden können?

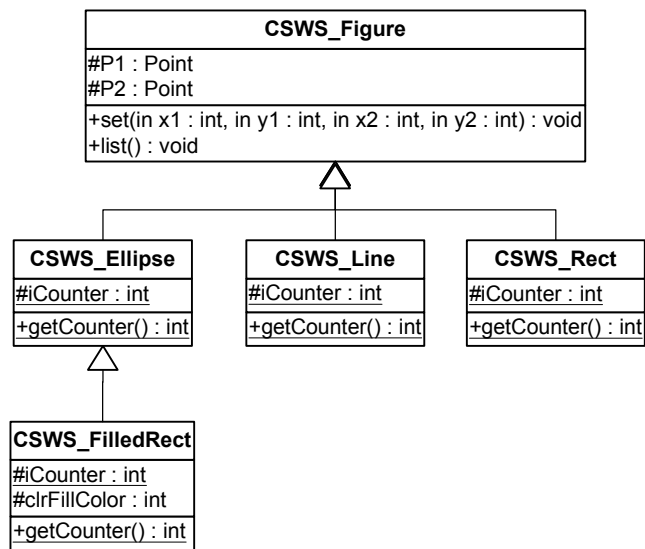


Abbildung 6-4 Neu eingeführte Unterklasse, durch Verfeinerung der vorhandenen Überlegungen.

Die Generalisierung und Spezialisierung definiert die Gemeinsamkeiten in Bezug auf Status und Verhalten der Klassen. Im Rahmen dieser Betrachtungen gibt es zwei Spezialfälle für das Verhalten von Klassen:

- statische Polymorphie
- dynamische Polymorphie

Statische Polymorphie

Bei der statischen Polymorphie definiert eine Klasse in sich mehrere gleichnamige Methoden, oder eine Subklasse gegenüber der Basisklasse neue gleichnamige Methoden (Gleichnamig aber nicht mit gleicher Signatur).¹

Die statische Polymorphie benutzen wir für die Bildung von Gruppen mit Funktionen, die in Bezug auf ein Objekt den selben Effekt aber unterschiedliche Anfangsbedingungen aufweisen.

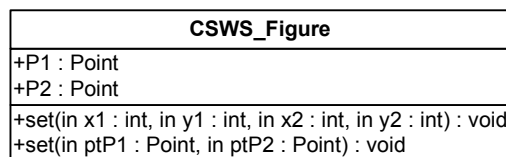


Abbildung 6-5 Funktionen mit Namen "set" aber unterschiedlicher Signatur, bilden eine statische Polymorphie.

¹ Die Implementation der statischen Polymorphie sieht in C++ eine Unterscheidung der Funktionssignatur vor, das heisst, dass sich die Parameter der gleichnamigen Funktionen unterscheiden.

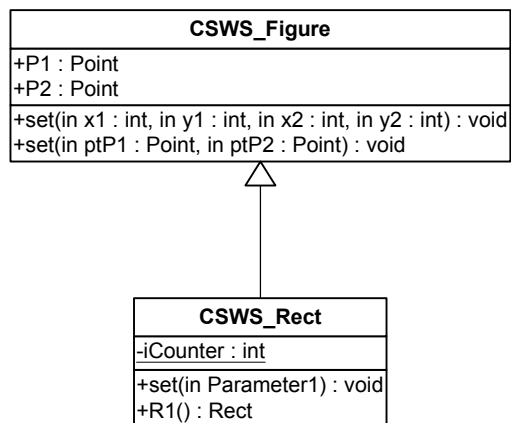


Abbildung 6-6 Funktionen mit Namen "set" aber unterschiedlicher Signatur über eine Vererbung, bilden ebenfalls eine statische Polymorphie.

Dynamische Polymorphie

Bei der dynamischen Polymorphie definieren mehrere Subklassen gegenüber einer Basisklasse eine identische Methode (mit identischer Funktionssignatur). Dabei wird die Absicht verfolgt, dass zur Laufzeit dynamisch aufgrund des Objektes (resp. seiner Abstammung) entschieden wird, welche Methode genau zur Anwendung kommt.²

Die dynamische Polymorphie wird bei Massenverarbeitungen von Objekten eingesetzt, bei welchen zum Zeitpunkt des Design die Objektreihenfolge resp. die Typen der Objekte nicht bekannt sind.

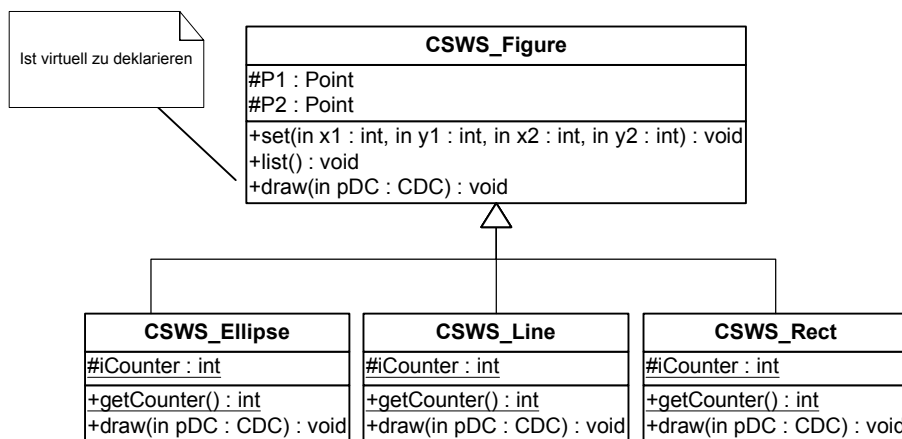


Abbildung 6-7 Funktionen mit Namen identischer Signatur über mehrere Vererbungsstufen bilden eine dynamische Polymorphie insofern die Funktion in der Basisklasse virtuell sind.

Mit der Vererbung sind die inneren Zusammengehörigkeiten (quasi der Stammbaum) der verschiedenen Klassen definiert. Nun ist das Klassenmodell mit allfällig weitergehenden Beziehungen zwischen den allenfalls artfremden Klassen untereinander zu untersuchen und zu definieren.

² Die Implementation der dynamischen Polymorphie sieht in C++ vor, dass die Methode in der Basisklasse virtuell deklariert wird.

6.2.4.2 Assoziation

Assoziationen verknüpfen unabhängige Objekte miteinander. Die Beziehungen bilden dabei ein kontextbezogenes Verhältnis ab. Die Menge der möglichen Beziehungen vom einen zum andern Objekt wird auch mit der Multiplizität umschrieben.

Im Verlauf des Designs werden Assoziationen zunächst ohne nähere Spezifikation der genauen Verhältnisse als Verbindungslinie zwischen verschiedenen Objekten festgehalten, allenfalls mit einem Beziehungsnamen versehen. Die Erkenntnisse stammen aus der umgangssprachlichen Beschreibung, dem gemeinsamen Auftreten in Listen, Masken oder Dialogen und dem Verständnis für die Sache an sich. Aussagen wie untenstehend aufgeführt, deuten auf Assoziationen hin:

- Objekt B kennt Objekt Q
- Alle Objekte D die dem Objekt T zugeordnet sind
- Objekt X besteht aus einem Objekt Y und zwei Objekten Z
- Alle Objekte Q des Typs N
- Alle Objekte W der Gruppe A (oder gehört zur Gruppe A)



Abbildung 6-8 Assoziationen vorerst festhalten durch die Assoziationslinie.

Im Rahmen weitergehender Modellierungsschritte werden dann die Assoziationen näher untersucht und Schritt für Schritt verfeinert. Dabei geht es darum, herauszufinden, ob die Beziehung symmetrisch oder asymmetrisch ist. Symmetrische Beziehungen werden als Assoziation weitergeführt und beschrieben, währenddem asymmetrische Beziehungen auf Aggregationen oder Kompositionen (Spezialformen der Assoziation) hindeuten (siehe Kapitel Aggregation und Komposition).

Folgende weitere Punkte müssen Schritt für Schritt untersucht werden:

- Welcher Name soll der Beziehung gegeben werden?
- Besteht zwischen den Objekten (Klassen) nur eine mögliche Beziehung?
- Wenn nein, interessieren für die Problemlösung die andern Beziehungen?
- Wie sind die Multiplizitäten der Beziehungen?
- Welche Rollen nehmen die Objekte in Bezug auf eine Beziehung war?
- Sind die Beziehungen einseitig oder wechselseitig?

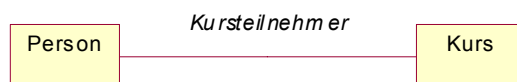


Abbildung 6-9 Assoziationen weiter verfeinern indem der Beziehungsname eingesetzt wird.

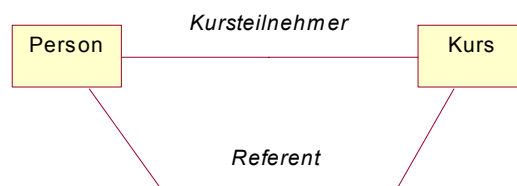


Abbildung 6-10 Mehr als eine Beziehung wird erkannt.

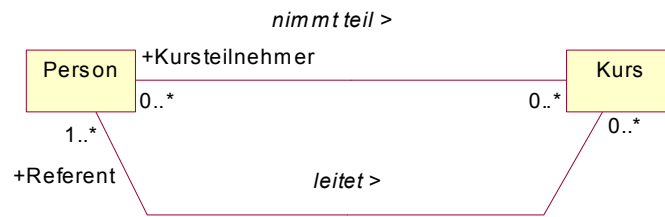


Abbildung 6-11 Die Beziehungen werden mit Multiplizitäten definiert, allfällige Rollen werden zugeteilt.

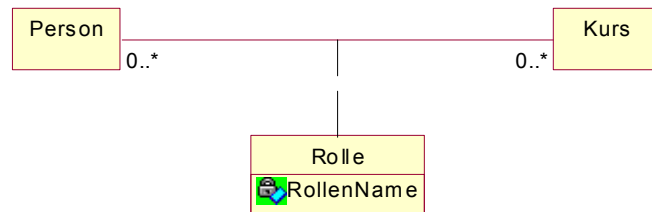


Abbildung 6-12 Varianten der gleichen Aussage studieren.

6.2.4.3 Aggregation und Kompositionen

Wie wir bereits im Kapitel 3 - UML Basics gesehen haben, beschreibt eine Aggregation eine Ganzes-Teil Beziehung (engl. Whole-Part) und ist in sich eine Spezialform einer Assoziation. Im Rahmen der Klassenmodellierung fokussieren wir uns nun auf die Frage, wie erkenne ich eine solche Beziehung?

Wir orientieren uns zur Beantwortung der Frage einmal mehr am normalen Sprachgebrauch über die zu beschreibenden Objekte und Klassen. Aggregationen erkennt man typischerweise an den Fragestellungen / Beschreibungen:

- Objekt X besteht aus einem Objekt Y und zwei Objekten Z
- Objekt F hat drei Objekt D und zwei Objekt G
- Objekt Q entsteht automatisch wenn Objekt K entsteht
- Objekt P lebt nur solange wie Objekt R

Die Existenzabhängigkeit

Die beiden letzten Fragen in obenstehender Liste beinhalten Angaben zur Lebensdauer eines Objektes. Für den Fall, dass das Teilobjekt einer Aggregation existenzabhängig vom Aggregat ist, können wir eine Komposition bilden.

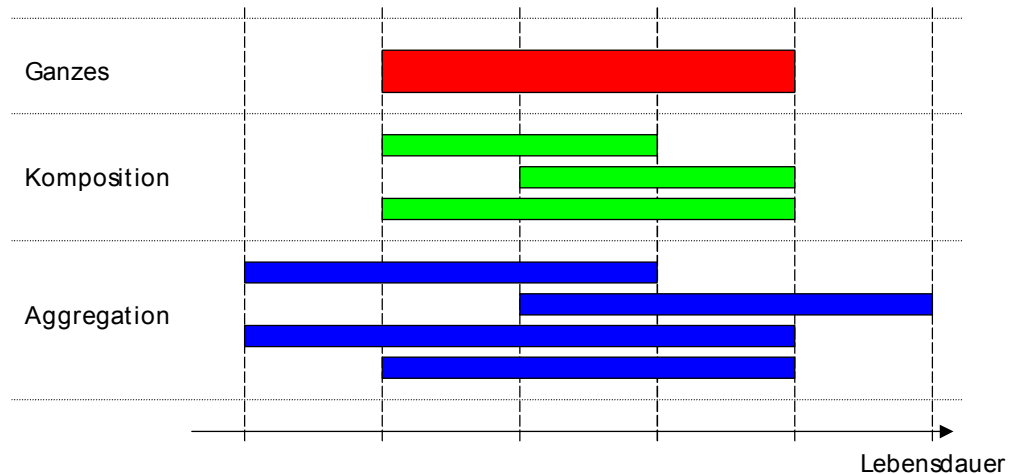


Abbildung 6-13 Darstellung der möglichen Lebensdauer der Teilobjekte für Aggregationen und Kompositionen.

Klassen sind Kompositionen

Genau betrachtet können wir eine normale Klasse als Komposition bezeichnen. Dabei übernimmt die Klasse die Funktion des Ganzen und die Attribute bilden die Teile. Diese Darstellung einer Klasse ist zwar nicht üblich, zeigt jedoch eigentlich die Verhältnisse in einer Komposition sehr schön auf.

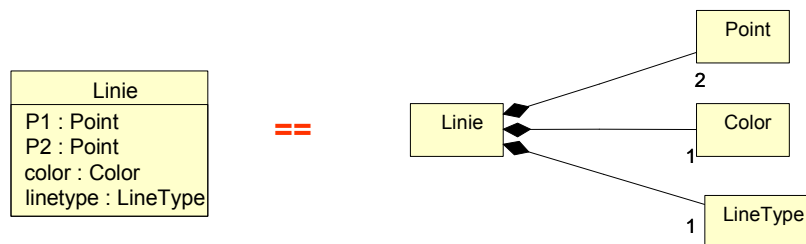


Abbildung 6-14 Gedankenmodell: Darstellung einer Klasse als Komposition.

Die Verantwortung der Aggregatklasse

Aggregationen werden auch als nicht symmetrische Beziehungen bezeichnet. Dies stammt davon, dass sich in der Aggregatsklasse Funktionalitäten implementieren lassen, die zur Kontrolle des Aggregates dienen. Das heisst, dass die Aggregatsklasse Operationen auf alle Teile anwenden kann (Propagieren von Operationen).

Zur Dokumentation dieser Möglichkeit wenden wir uns einmal einem andern Beispiel zu:



Abbildung 6-15 Die Rechnung berechnet den Gesamtpreis durch Abfrage des Preises aller enthaltenen Artikel.

Die Multiplizität der Aggregation und Komposition

Einen weiteren Unterschied zwischen Aggregation und Komposition finden wir in den möglichen Multiplizitäten. Aufgrund der Definition der Lebensdauer des Teilobjektes in einer Aggregation ist es grundsätzlich möglich (wenn auch selten), dass ein Teil zu einem Zeitpunkt in mehreren Aggregationen vorkommt.

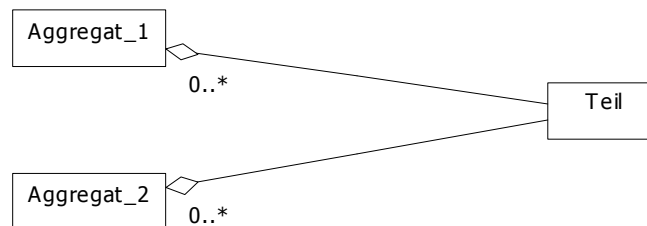


Abbildung 6-16 Der Teil partizipiert in mehreren Aggregationen.

Im Gegensatz dazu kann ein Teil nicht gleichzeitig zu mehreren Kompositionen gehören. Zwar könnte man ein solches Gebilde zum Leben erwecken, da zuerst die Aggregatobjekte der Komposition erstellt werden könnten und anschliessend die Teile hinzugefügt werden könnten. Aufgrund der Regel, dass mit dem Zerstören des Aggregates auch die Teile zerstört werden, impliziert dies eine Beeinflussung der andern Komposition, was ein unschöner Nebeneffekt wäre.

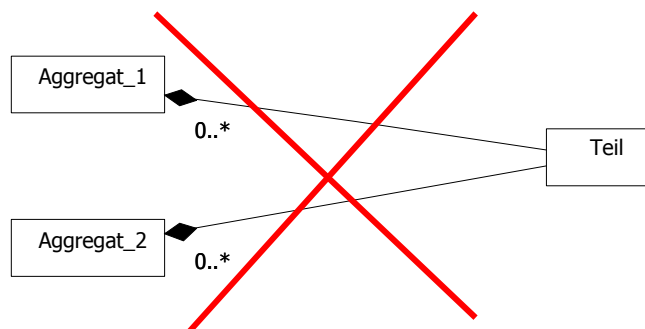


Abbildung 6-17 Der Teil kann nicht gleichzeitig Bestandteil mehrerer Kompositionen sein.

6.2.5 Notation

Siehe Kapitel 3 UML Basics.

6.2.6 Beispiel

Das Beispiel stützt sich auf die Übungsserie des Kurses und zeigt ein Übersichtsdiagramm für die Zeichnung, der ihr zugeordneten View und des Zeichnungsinhaltes.

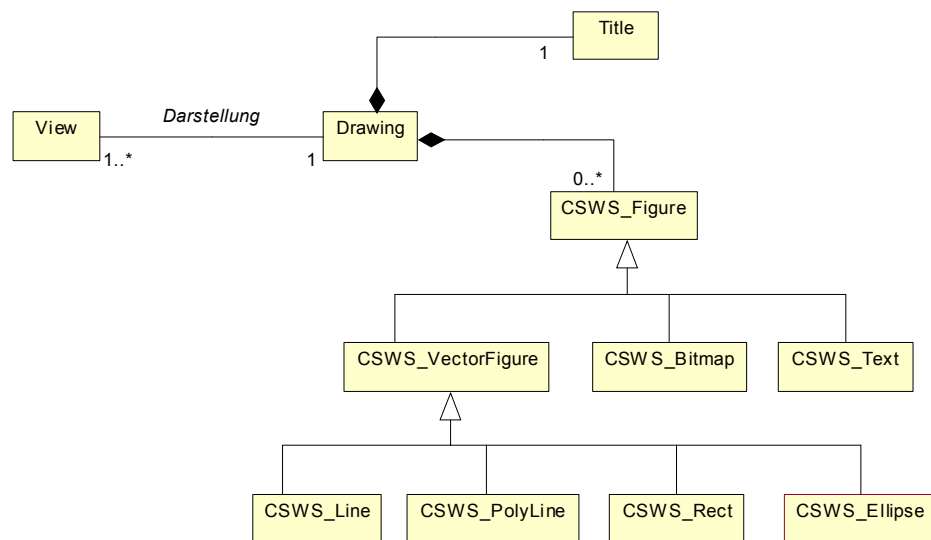


Abbildung 6-18 Mögliches Klassendiagramm im Kursbeispiel.

6.2.7 Fallen der Klassenmodellierung

- Aggregation anstelle einer Vererbung: Eine Vererbung wird nicht richtig erkannt, und in Form einer Aggregation versucht darzustellen oder umgekehrt.
- Komposition anstelle Aggregation: Eine Komposition ist eine Existenzabhängige Aggregation. Je nach Fokus sind Teile oft nicht zwingend Existenzabhängig zu definieren.
- Falsche Multiplizitäten: Es werden nicht alle Multiplizitäten geprüft. Multiplizität mit 1 oder grösser als Minimalwert implizieren existenzabhängigkeit, was wiederum entsprechende Auswirkungen auf die Programmierlogik hat.

6.2.8 Aufwand

Die Aufwandschätzung für ein Klassenmodell ist kaum mehr möglich allgemein auszudrücken. Als Bemessungsgrundlage soll aber folgende Faustregel genannt sein: Pro Klasse oder Beziehung 1 ... 4 Stunden Zeit einplanen.

6.3 Sequenzmodelle

6.3.1 Vorgehen

Sequenzmodelle werden zum Studium und zur Definition von Interaktionen zwischen einer definierten Menge von Objekten in einer bestimmten Zeit benötigt. Es gibt in der Methode UML keinen Zwang für die Erstellung von solchen Diagrammen. Ich empfehle deren Anwendung ganz einfach dann, wenn die Verhältnisse zwischen den Objekten schwer verständlich sind, mehrere Lösungen für die Folge einer Interaktion möglich sind, oder zur Klarstellung zwischen verschiedenen Entwicklern die zeichnerische Darstellung die Diskussionen verdeutlicht.

Für die Erstellung eines Sequenzmodells (-diagramms) werden die beteiligten Objekte oder Klassen in der horizontalen oben in einem Diagramm angeordnet. Die Reihenfolge spielt grundsätzlich keine Rolle, jedoch lässt sich aus der Literatur erkennen, dass allgemein gebräuchlich das Objekt (resp. die Klasse) von dem aus die Sequenz startet links eingefüllt wird.

Anschliessend werden in der Reihenfolge der zeitlichen Geschehnisse die Interaktionen (Nachrichten) zwischen den beteiligten Objekten eingezeichnet.

6.3.2 Notation und Beispiel

Siehe Kapitel 3 UML Basics.

6.3.3 Fallen

Die Beschreibung von Sequenzen birgt in sich folgende Fallen:

- Zu umfassende Beschreibung der Verhältnisse (z.B. alle Objekte einer Applikation) oder zu detaillierter Nachrichtenaustausch führt zu grossen, nur schwer verständlichen Diagrammen.
- Ungleiche Detailtreue innerhalb eines Diagrammes in Bezug auf Nachrichtenaustausch führt ebenfalls zu schwer verständlichen Diagrammen.
- In der horizontalen Anordnung sollen alle Objekte oder alle Klassen dargestellt werden. Eine Mischung von Klassen und Objekten reduziert die Verständlichkeit.

6.3.4 Aufwand

Der Aufwand für das Erarbeiten von Sequenzdiagrammen ist sehr unterschiedlich. Pro Diagramm sind schnell 2..3 Stunden zu gebrauchen. Die Menge von Sequenzdiagrammen kann nicht vorhergesagt werden, weil sie projektabhängig ist.

6.4 Kollaborationsmodelle

6.4.1 Vorgehen

Das Kollaborationsdiagramm dient ähnlich dem Sequenzdiagramm eine Reihenfolge für die Interaktion zwischen den Objekten zu dokumentieren. Im Gegensatz zum Sequenzdiagramm benutzt jedoch das Kollaborationsdiagramm die Beziehungen (Assoziationen) zwischen den Objekten (resp. den Klassen) als Grundlage für die grafische Darstellung.

Für die Erstellung eines Kollaborationsdiagramms bedienen wir uns am einfachsten eines Objektdiagramms (oder allenfalls eines Teildiagrammes). Dieses zeigt als Grundlage die Assoziationen zwischen den Objekten. Aufbauend auf den Assoziationslinien fügt man nun in der Grafik die einzelnen Nachrichten ein. Die Numerierung der Nachrichten definiert dabei die zeitliche Abfolge.

Kollaborationsdiagramme sind praktische Hilfsmittel im Teamumgang. Sie benötigen wenig zeichnerisches Geschick und lassen sich insbesondere auf dem Flipchart oder dem Hellraumprojektor händisch, während Diskussionen und Designvorgängen im Team darstellen.

6.4.2 Notation und Beispiel

Siehe Kapitel 3 UML Basics.

6.4.3 Fallen

Die Beschreibung von Kollaborationen birgt in sich folgende Fallen:

- Zu umfassende Beschreibung der Verhältnisse (z.B. alle Objekte einer Applikation) oder zu detaillierter Nachrichtenaustausch führt zu grossen, nur schwer verständlichen Diagrammen.
- Ungleiche Detailtreue innerhalb eines Diagrammes in Bezug auf Nachrichtenaustausch führt ebenfalls zu schwer verständlichen Diagrammen.
- In der horizontalen Anordnung sollen alle Objekte oder alle Klassen dargestellt werden. Eine Mischung von Klassen und Objekten reduziert die Verständlichkeit.
- Zu detaillierte Angaben für die Nachrichten sind schwer verständlich.

6.4.4 Aufwand

Der Aufwand für das Erarbeiten von Kollaborationsdiagrammen ist sehr unterschiedlich. Da die Diagramme insbesondere für die händische Projektarbeit geeignet sind, verzichte ich hier auf eine weitere Schätzung.

6.5 Komponentenbildung

6.5.1 Vorgehen

Die Komponentenbildung ist insbesondere in grossen Projekten ein sehr wichtiges Unterfangen, welches die logische Aufteilung des Codes in mehr oder weniger selbständige Teile beschreibt. Ein Teil kann dabei ein DLL, ein EXE oder eine andere Codeeinheit beschreiben. Die Komponentenbildung ist abhängig von der gewählten Systemarchitektur.

Komponenten können mit folgenden Kriterien gefunden werden:

- Gruppierung nach Geschäftslogik
- Gruppierung nach Funktionslogik
- Gruppierung nach Verarbeitungslogik (Client-Server)
- Kombinationen von obigen Punkten

Nachdem die Gruppierung abgeschlossen ist, sollte diese nach Komponenten durchsucht werden, die sich allenfalls verallgemeinern lassen würde. Allgemeingültige Komponenten benötigen zwar im konkreten Projekt etwas mehr Zeit für die Realisierung, bilden jedoch gleichzeitig eine Investition für eine künftig schnellere Entwicklung.

Neben der Gruppierung von Funktionen und der Beschreibung von Schnittstellen werden in den Komponentendiagrammen auch Abhängigkeitslinien eingezeichnet. Anhand der Abhängigkeitslinien lässt sich die Realisierungsreihenfolge bestimmen (Unabhängige Komponenten zuerst realisieren).

6.5.2 Notation und Beispiel

Siehe Kapitel 3 UML Basics.

6.5.3 Fallen

Die Komponentenbildung birgt in sich folgende Fallen:

- Aufgabenstellungen werden nicht als allgemeingültig erkannt. Dies führt zu mehrfacher Implementation von Funktionalität über verschiedene Projekte hinweg.
- Bei der Aufteilung von Funktionen in Komponenten werden die vorhandenen Kommunikationen missachtet. Intensiver Nachrichtenaustausch zwischen Komponenten über langsame Leitungen kann eine katastrophale Performance der Applikation zur Folge haben.

6.5.4 Aufwand

Keine Schätzung.

7 Die Verwendung von Design-Patterns

7.1 Was sind Design-Patterns

Design Pattern (zu deutsch Muster) sind Standardansätze für Problemlösungen im Bereich des Softwareengineering. Sie dienen uns im Entwickleralltag als altbewährte Denkmuster, mit dem Ziel möglichst effizient den richtigen Lösungsansatz für eine Problemstellung zu erreichen.

Viele werden sich nun vielleicht an die Algorithmen und Datenstrukturen erinnern, und sich dabei fragen: "Ist das Pattern nicht einfach ein neuer Begriff für Algodata" ? Nein bei weiten nicht. Währenddem die Algorithmen und Datenstrukturen dazu dienen Berechnungsprobleme und Datenhandhabungen zu lösen, verfolgen die Softwarepattern eine "höhere Zielsetzung". Die Patterntechnik versucht die Normierung der Softwarearchitektur zu erreichen. Natürlich brauchen wir beide Elemente um Software erfolgreich zu schreiben. In Realität wird sich ein Pattern sehr oft einem oder verschiedenen Algorithmen bedienen um erfolgreich implementiert zu werden.

Für sehr viele Problemstellungen existieren Design Patterns. Obschon im Nachhinein oft schwierig zu sagen ist, ob zuerst das Pattern oder ein Produkt vorhanden war, finden wir heute in der Literatur Pattern mit bedeutungsvollen Namen, die nicht selten an bekannte Techniken erinnern:

- Whole - Part
- Master - Slave
- Proxy
- Command Handler
- Forwarder - Receiver
- Client - Dispatcher - Server
- Publisher - Subscriber

Ein Studium der heute definierten Muster zeigt, dass letztere auf sehr unterschiedlichen Abstraktionsebenen definiert sind. Einige Muster helfen ein System in Subsysteme zu strukturieren, andere Muster sind eher geeignet Subsysteme in Komponenten zu teilen und wieder andere Muster sind wenig abstrakt, sondern helfen bestimmte Aspekte eines Entwurfes in einer bestimmten Programmiersprache auszudrücken.

In der Literatur werden Entwurfsmuster in drei Kategorien unterteilt:

- Architekturmuster
- Entwurfsmuster
- Idiome

Jeder Kategorie werden Muster zugeteilt, die in etwa den gleichen Abstraktionslevel enthalten.

7.1.1 Architekturmuster

Ein Architekturmuster spiegelt ein grundsätzliches Strukturierungsprinzip von Softwaresystemen wieder. Es beschreibt eine Menge vordefinierter Subsysteme, spezifiziert deren jeweiligen Zuständigkeitsbereiche und enthält Regeln zur Organisation der Beziehung zwischen den Subsystemen.

Das Architekturmuster bildet die abstrakteste Ebene der Entwurfsmuster. Sie werden sehr früh dem Design zu Grunde gelegt und beeinflussen so die Lösung markant.

Das Model-View-Controller Pattern ist ein Architekturmuster (siehe weiter unten).

7.1.2 Entwurfsmuster

Ein Entwurfsmuster beschreibt ein Schema zur Verfeinerung von Subsystemen oder Komponenten eines Softwaresystemes oder den Beziehungen zwischen ihnen. Es beschreibt eine häufig auftretende Struktur von miteinander kommunizierenden Komponenten, die ein allgemeines Entwurfsproblem in einem speziellen Kontext löst.

Entwurfsmuster bilden die mittlere Ebene der Abstraktion in einem Entwurf und beeinflussen nicht das gesamte System sondern nur Teile davon. Die Muster sind in der Regel noch sprachunabhängig.

Das Composite Pattern ist ein Entwurfsmuster (siehe weiter unten).

7.1.3 Idiome

Ein Idiom ist ein für eine bestimmte Programmiersprache spezifisches Muster auf einer niedrigen Abstraktionsebene. Es beschreibt, wie man spezielle Aspekte von Komponenten oder deren Beziehungen zwischen ihnen mit den Mitteln der Programmiersprache implementieren kann.

Die meisten Idiome sind sprachspezifisch und spiegeln Verfahren für die Implementierung wider.

7.1.4 Eine nicht komplette Liste

Thema	Architekturmuster	Entwurfsmuster	Idiome
Vom Chaos zur Struktur	• Layers • Pipes-and-Filters • Blackboard		
Verteilte Systeme	• Broker • Pipes-and-Filters • Microkernel		
Interaktive Systeme	• Model-View-Controller • Presentation-Abstraction-Controller		
Erzeugung		• Abstract-Factory • Prototype Builder	• Singleton • Factory-Method
Strukturelle Zerlegung		• Whole-Part • Composite	
Organisation von Arbeit		• Master-Slave • Chain of Responsibilities • Command Mediator	
Zugriffskontrolle		• Proxy • Facade	
Management		• Command Processor • View Handler	
Kommunikation		• Publisher-Subscriber • Forwarder-Receiver • Client-Dispatcher-Server	
Ressourcenverwaltung		• Flyweight	• Counted Pointer

In der Folge werden drei oft verwendete Pattern ausgesucht und stellvertretend für eine enorme Menge an heute vorhandenen Mustern näher beschrieben:

- Singleton
- Composite
- Modell - View - Controller (MVC)

7.2 Singleton Idiom

7.2.1 Ziel

Das Muster verfolgt die Zielsetzung, dass von einer Klasse garantiert nur eine Instanz erzeugt wird. Zudem soll gewährleistet werden, dass ein globaler Zugriffspunkt existiert.

7.2.2 Anwendung

Das Pattern findet überall dort Anwendung, wo eine singuläre Instanz für die Erledigung einer Aufgabe verantwortlich ist. Als Beispiel können wir hier ein Filesystem in einem Betriebssystem oder den Window Manager in einem GUI nennen.

7.2.3 Struktur

Die Struktur eines Singleton ist äusserst einfach:

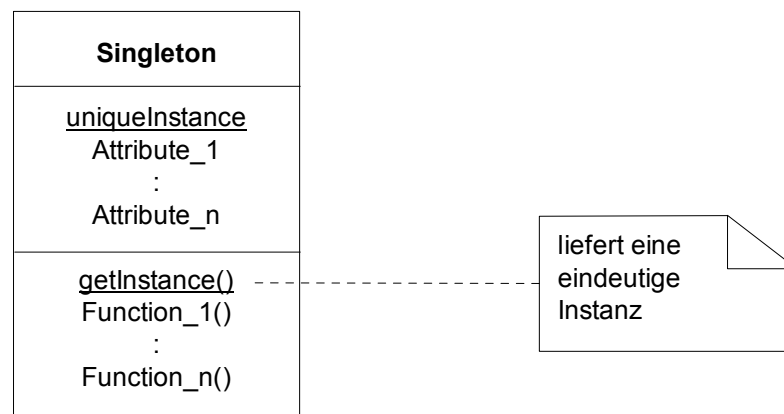


Abbildung 7-1 Struktur Singleton

Die Instanzierungsfunktion des Singleton ist verantwortlich, dass nur eine Instanz aus der Klasse erzeugt wird.

7.2.4 Konsequenzen

Folgende Vorteile können bei der Nutzung des Singleton Design gewonnen werden:

- Die Kontrolle zur Instanzierung wird beim Singleton Muster auf die Singleton Klasse gelegt. Dadurch wird die Instanzierung unabhängig vom Client.
- In einem Client werden für den Zugriff auf das Singleton keine globalen Variablen benötigt. Dies vereinfacht die Handhabung der Gültigkeiten und Namensräume.
- Durch Vererbung kann die Funktionalität auf Datenklassen weitergegeben werden.
- Das Pattern kann einfach dahingehend erweitert werden, dass eine bestimmte Menge von Instanzen erzeugt werden kann. Es ist sogar eine allgemeine Implementation denkbar, bei der die Anzahl der zu erzeugenden Instanzen von aussen konfiguriert werden kann.

7.2.5 Implementation

Um garantiert sicherzustellen, dass nur eine Instanz einer Singleton Klasse erzeugt wird, muss die Klasse selber ihre Instanz erzeugen. Dies impliziert eine Klassenfunktion, die die Instanzierung in sich kapselt.

```
1 class Singleton {
2     private:
3         static Singleton *pInstance;
4     protected:
5         Singleton();
6     public:
7         static Singleton *getInstance();
8 };
```

Beachte, dass der Konstruktor in obigem Code `protected` deklariert ist. Dies schützt die Klasse vor einer unerlaubten, direkten Instanzierung durch einen Client. Ferner sind die Deklarationen von `getInstance()` und `pInstance` statisch. Damit kann sichergestellt werden, dass die Funktion mit Scope Klasse (d.h. ohne Vorhandensein eines Objektes) aufgerufen werden kann.

Die Implementierung der Funktion `getInstance()` könnte wie folgt aussehen:

```
9 // implement the static member to hold the pointer to the instance
10 Singleton *Singleton::pInstance = NULL;
11
12 // implementation of class method to instantiate the object
13 Singleton *Singleton::getInstance()
14 {
15     if ( pInstance == NULL ) {
16         pInstance = new Singleton;
17     }
18     return pInstance;
19 }
```

Clients greifen auf das Objekt ausschliesslich über den erhaltenen Zeiger aus der Klassenfunktion `Singleton::getInstance()` zu. Der obenstehende Ansatz für die garantierte singuläre Instanzierung eines Objektes ist innerhalb eines Prozesses mit statischer Bindung schon ganz nett. Wie sieht es aber aus, wenn ein solches Konstrukt über die Grenzen eines Prozesses hinaus singulär bleiben soll. Für diesen Fall empfiehlt es sich die Funktion für die Instanzierung zusätzlich mit einem Mechanismus für die Interprozesskommunikation auszustatten, respektive die Klasse so auszulegen, dass sie in einem eigenen Prozessbereich ablaufen kann.

7.3 Composite Pattern

7.3.1 Ziel

Das Muster verfolgt die Zielsetzung einer allgemein gültigen Implementation einer Teil-Ganzes Hierarchie, die so gehalten ist, dass sie auch rekursive Strukturen abbilden kann. Ein Client der eine Composite Struktur nutzt, soll sich dabei nicht um den Unterschied zwischen den effektiven Nutzobjekten und den Containerobjekten kümmern, die eine solche Struktur aufweist.

7.3.2 Anwendung

Das Composite Pattern kann überall dort angewendet werden, wo die Aggregation von Teilen mit Unterstrukturen von den gleichartigen Teilen wieder vorkommen kann. Diese Problematik ist in allen Anforderungsbereichen von Stücklisten anzutreffen.

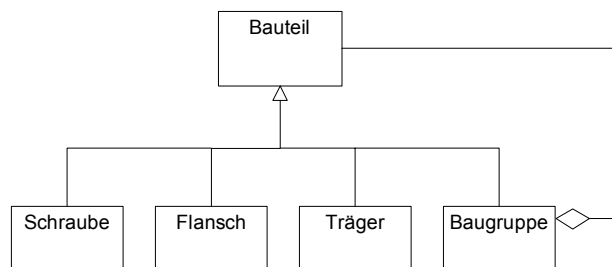


Abbildung 7-2 Beispielstruktur des Composite Pattern

7.3.3 Struktur

Die allgemeingültige Struktur des Pattern ist wie folgt definiert:

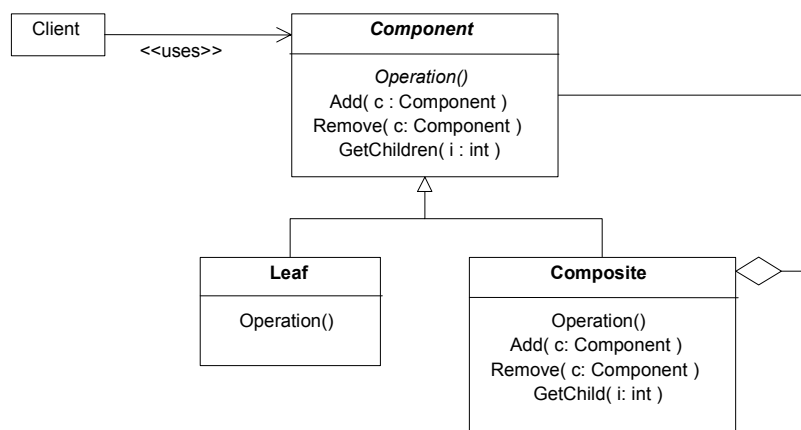


Abbildung 7-3 Beispielstruktur des Composite Pattern

7.3.4 Konsequenzen

Mit Einsatz des Composite Pattern können folgende Effekte erzielt werden:

- Möglicher hierarchischer Aufbau von einfachen Objekten über Vererbungsstrukturen mit möglicher Kombination von Ganzes-Teil Strukturen.
- Einfacher Aufbau des Nutzers (Client) und lose Kopplung zwischen Client und Composite Informationsstruktur. Der Client kann konsequent das Composite Objekt nutzen um die eigentlichen Nutzdaten zu managen (new, update, delete) und deren Inhalte zu erfragen.
- Die Erweiterung der Vererbungsstrukturen der Nutzdatenklassen (new leafs) durch neue Klassen ist nachträglich möglich. Neu hinzugefügte Klassen arbeiten automatisch mit der gesamten Struktur.
- Die mögliche, allgemeingültige Auslegung des Musters hat jedoch auch den Nachteil, dass unterschiedliche Multiplizitäten des Aggregates nur schwer erfasst werden können. Diese Anforderung kann jedoch mit einer konfigurierbaren Mengenbeschränkung die "at runtime" die Multiplizität prüft, implementiert werden.

7.3.5 Implementation

Das hier aufgezeigte Implementationsbeispiel stützt sich auf eine "natürliche Komposition" eines zusammengestellten PCs. Dabei spielt die Klasse "Equipment" die grundlegende Komponente.

```
1  // -----
2  // composite.cpp
3  //
4  // composite pattern demo programm
5  // -----
6
7  // -----
8  // includes
9  // -----
10 #include "stdafx.h"
11 #include <iostream>
12 #include <list>
13 using namespace std;
14
15 // -----
16 // base class Equipment
17 // -----
18 class Equipment {
19 // -----
20 private:
21     const char *pszName;
22
23 public:
24     virtual ~Equipment() {};
25
26     virtual int  NetPrice( void ) = 0;
27     virtual void Add( Equipment * ) {};
28     virtual void List( void ) { cout << pszName << endl; }
29
30 protected:
31     Equipment(const char * pName) { pszName = pName; }
32 };
```

Die Basisklasse stellt virtuelle Funktionen für die Zusammenstellung von Informationen der gesamten Komposition zur Verfügung. Diese Funktionen werden in den Subklassen implementiert, die die eigentlichen, konkreten Teile beschreiben (Hier am Beispiel einer Klasse für externe Speicher und einer Klasse für Bildschirme).

```

33 // -----
34 // class for a drive
35 // -----
36 class Drive : public Equipment {
37 // -----
38 public:
39     Drive(const char * pName) : Equipment(pName) {}
40     virtual ~Drive() {}
41
42     virtual int NetPrice( void );
43 };
44
45 // -----
46 int Drive::NetPrice( void )
47 // -----
48 {
49     // the price should be stored in database !
50     return 56;
51 }
52
53 // -----
54 // class for a display
55 // -----
56 class Display : public Equipment {
57 // -----
58 public:
59     Display(const char * pName) : Equipment(pName) {}
60     virtual ~Display() {}
61
62     virtual int NetPrice( void );
63 };
64
65 // -----
66 int Display::NetPrice( void )
67 // -----
68 {
69     // the price should be stored in database !
70     return 380;
71 }

```

Für die komplette Implementation brauchen wir nun noch den Vertreter der Composite Klasse. Diese wird als Container wirken um weitere Baugruppen zu definieren, sowie die Methoden implementieren, die auf alle enthaltenen Komponenten wirken:

```

72 // -----
73 // class CompositeEquipment
74 // -----
75 class CompositeEquipment : public Equipment {
76 // -----
77 private:
78     list< Equipment *> listEquipment;
79
80 protected:
81     CompositeEquipment( const char *pName ) : Equipment(pName) {}
82
83 public:
84     virtual ~CompositeEquipment() {}
85
86     virtual int NetPrice();
87     virtual void Add( Equipment * );
88     virtual void List( void );
89 };
90
91 // -----
92 int CompositeEquipment::NetPrice()
93 // -----
94 {
95     // local iterator object and sum object
96     list< Equipment *>::iterator i;
97     int iSum = 0;
98
99     // iteration over list to calculate price of components
100    for (i=listEquipment.begin(); i!=listEquipment.end(); i++ ) {
101        iSum += ((Equipment *)*i)->NetPrice();
102    }
103
104    // return sum
105    return iSum;
106 };
107
108 // -----
109 void CompositeEquipment::Add( Equipment *pPart )
110 // -----
111 {
112     // insert element into list
113     listEquipment.insert( listEquipment.end(), pPart );
114 }
115
116 // -----
117 void CompositeEquipment::List( void )
118 // -----
119 {
120     // create iterator
121     list< Equipment *>::iterator i;
122
123     // out the text for object himself
124     Equipment::List();
125
126     // out text for included equipment
127     for (i=listEquipment.begin(); i!=listEquipment.end(); i++ ) {
128         ((Equipment *)*i)->List();
129     }
130 }

```

Selbstverständlich kann nun eine spezialisierte Klasse zur Handhabung des "Hauptobjektes" der Problemlösung definiert werden. Dieses Hauptobjekt wollen wir Computer nennen. Zudem definieren wir eine Klasse "Gehäuse" (Tower), die die Innereien aufnehmen wird.

```
131 // -----
132 // class Computer
133 // -----
134 class Computer : public CompositeEquipment {
135 // -----
136 public:
137     Computer(const char * pName) : CompositeEquipment(pName) {}
138     virtual ~Computer() {}
139
140 };
141
142 // -----
143 // class Tower
144 // -----
145 class Tower : public CompositeEquipment {
146 // -----
147 public:
148     Tower(const char * pName) : CompositeEquipment(pName) {}
149     virtual ~Tower() {}
150     virtual int NetPrice();
151 };
152
153 // -----
154 int Tower::NetPrice( void )
155 // -----
156 {
157     // Price for component is price for cabinet plus the
158     // included components
159     return 50 + CompositeEquipment::NetPrice();
160 }
```

Ein Client könnte die in obenstehenden Codefragmenten aufgezeigten Mechanismen nun wie folgt nutzen:

```
161 // -----
162 // client to use the pattern
163 // -----
164 int main(int argc, char* argv[])
165 {
166     // create a computer
167     Computer *pComputer = new Computer("Mein Rechner enthaelt..");
168
169     // add a display to the computer
170     pComputer->Add( new Display( "17\" Color Display" ));
171
172     // create a container
173     Tower    *pTower = new Tower( "Das Gehaeuse enthaelt.." );
174
175     // add the container to the computer
176     pComputer->Add( pTower );
177
178     // add componentes to the container
179     pTower->Add( new Drive( "3 1/2\" Floppy" ));
180     pTower->Add( new Drive( "5 1/4\" Floppy" ));
181     pTower->Add( new Drive( "SCSI Disk 8 GB" ));
182
183     // show the whole
184     pComputer->List();
185
186     // show the costs
187     cout << "Kosten Total = " << pComputer->NetPrice() << endl;
188
189     // end client
190     return 0;
191 }
```

Obiger Client erzeugt folgende Ausgabe:

```
Mein Rechner enthaelt ...
17" Color Display
Das Gehaeuse enthaelt ...
3 1/2" Floppy
5 1/4" Floppy
SCSI Disk 8 GB
Kosten Total = 598
```

7.4 Model-View-Controller Pattern

7.4.1 Ziel

Das Model-View-Controller Pattern (kurz MVC) verfolgt die Trennung der Eingabe, Verarbeitung und Ausgabe von Informationen.

7.4.2 Anwendung

Das Muster kann bei der Entwicklung von interaktiven Systemen in Erwägung gezogen werden. Es trägt folgenden Umständen rechnung:

- Einfache Änderbarkeit der Benutzeroberfläche des Systems
- Portierbarkeit der Lösung über mehrere Systeme hinweg
- Erweiterbarkeit der Kernfunktionalität des Systems

Wie dem Namen des Pattern zu entnehmen ist, besteht es aus drei verschiedenen Hauptkomponenten:

- Model (steht für verarbeitende Komponente, Verarbeitung)
- View (steht für ausgebende Komponente, Ausgabe)
- Controller (steht für eingebende Komponente, Eingabe)

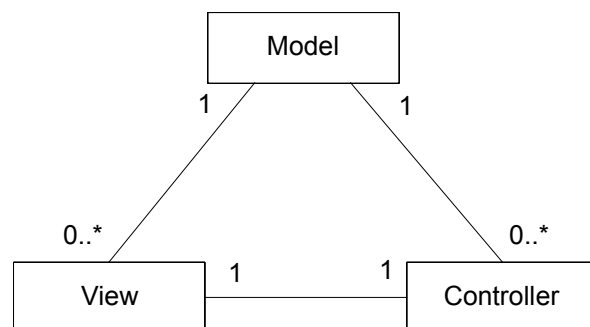


Abbildung 7-4 Das MVC Pattern

7.4.3 Elemente

7.4.3.1 Model

Die verarbeitende Komponente (Modell, Model) kapselt die Daten und die Kernfunktionalitäten. Sie ist unabhängig von speziellen, interaktiven Ausgabeformaten oder Eingabemöglichkeiten ausgelegt. Die interaktive Eingabe von Informationen erfolgt über die Controller-Komponente, die interaktive Ausgabe über die Ansicht.

Die Persistenz kann ebenfalls als Ein- und Ausgabe-Operation betrachtet werden. Diese Art der I/O ist im Zusammenhang mit dem MVC aber nicht relevant für die Aufgabenteilung zwischen den Komponenten. Die Persistenz wird in der Regel ebenfalls im Modell realisiert.

7.4.3.2 View

Die ausgebende Komponente (Ansicht, View) zeigt dem Anwender die Informationen. Zu diesem Zweck bezieht die Ansicht die Daten vom Modell. Ein Modell kann von mehreren Ansichten gleichzeitig dargestellt werden. Zudem können verschiedene Arten von Ansichten für ein Modell existieren. Beispiel Tabellenkalkulation: Die Zahlen werden zum einen in einer Tabelle, und zum andern in einer Grafik wieder gegeben.

7.4.3.3 Controller

Für jede Art von Ansicht existiert eine eigene Eingabekomponente (Controller). Die Eingaben geschehen üblicherweise durch den Bediener über Maus- oder Tastaturbedienung. Der Controller übersetzt die Eingaben der Benutzer in Serviceanforderungen an das Modell, welches auf diesem Weg neue Informationen, resp. geänderte Informationen erhält.

7.4.4 Konsequenzen

- + Mehrere Ansichten desselben Modells
- + Synchronisierte Ansichten
- + Potential für Frameworks
- Grössere Komplexität
- Potential für übermässige Anzahl von Aktualisierungen
- Enge Kopplung zwischen Ansicht und Controller
- Ineffizienter Datenzugriff innerhalb von Ansichten
- Schwierige Verwendung von MVC mit modernen Werkzeugen

7.4.5 Bekannte Variante

Das Document-View Pattern bildet wohl die wichtigste Variante des MVC-Pattern. Bei der Document-View Variante wird die Trennung zwischen der Ansicht und dem Controller im aufgehoben. Das Pattern besteht somit nur noch aus Model (Document) und View.

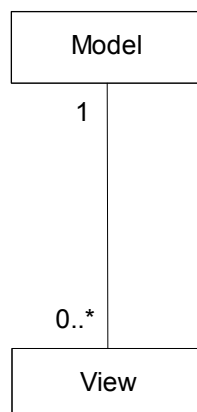


Abbildung 7-5 Das Modell-View Pattern (auch Document-View)

8 Anhang

8.1 CRC Karten

[illegible]

Legende:

- Working Name: Projekt interner Name der im Rahmen der Analyse und des Designs vom Team verwendet wird.
- Possible Names: Mögliche Namen für definitive Implementation
- Effective Name: Definitiv gewählter Name der Klasse
- Purpose: Beschreibung des Verwendungszweckes
- Responsibilities: Verantwortlichkeiten, die die Klasse wahrnimmt (Attribute, Zustände, Verhalten)
- Collaborations: Kollaborationen, die die Klasse eingeht (Client, Server ...)

8.2 Beschreibung eines Anwendungsfalles

Name:	
Nummer:	
Kurzbeschreibung:	
Beteiligt Akteure:	
Auslöser / Vorbedingung:	
Ergebnisse / Nachbedingung:	

Ablauf:

[illegible]

Ausnahmen, Varianten:

[illegible]

8.3 Klassenspezifikation / Generalisierungstabelle

[illegible]

8.4 Begriffe und Abkürzungen

Begriff / Kürzel	Bedeutung
Abstraktion	Eine Abstraktion beschreibt die grundlegende Charakteristika eines Objektes, relativ zur Perspektive des Anwenders.
abgeleitete Assoziation	Eine abgeleitete Assoziation besteht, wenn die Beziehung über einen zweiten Assoziationspfad jederzeit eindeutig redundant definiert werden kann.
Aggregat	Das ganze Teil einer Aggregation ($\Rightarrow$ Aggregation).
Aggregation	Eine spezielle Form der $\Rightarrow$ Assoziation. Die Aggregation definiert eine enthaltende Beziehung (X besteht aus Y und Z).
Assoziation	Eine Assoziation beschreibt eine Beziehung zwischen zwei Elementen. Beispiel Person A beschäftigt Person B. In der UML impliziert dies eine bidirekte relationale Beziehung.
assoziatives Array	$\Rightarrow$ Assoziation
Attribut	Eigenschaft oder Merkmal einer Sache (bei OOT einer Klasse resp. eines Objektes).
Attributwert	Inhalt eines Attributes.
berechnete Assoziation	$\Rightarrow$ Assoziation
bidirekte Assoziation	$\Rightarrow$ Assoziation
Concurrency	$\Rightarrow$ Gleichzeitigkeit
CORBA	Common Object Request Broker Architecture. Ein Konzept für verteilte Software Komponenten.
CRC	Classes, Responsibilities and Collaborations. Möglichkeit der Darstellung von Klassen in der Analysephase.
DBMS	Database Management System
DCOM	Distributed Common Object Model. Ein Konzept für verteilte Software Komponenten.

DDL	Data Description Language
derived association	⇒ abgeleitete Assoziation
Dependency	Abhängigkeitsbeziehung
Dictionary	⇒ qualifizierte Assoziation
Diskriminator	Unterscheidungsmerkmal für Generalisierung / Spezialisierung.
Encapsulation	⇒ Kapselung
ERD	Entity Relationship Diagram
Event	⇒ Zustandsübergang
Funktionen	⇒ Methoden
Generalisierung	Taxonomische Beziehung zwischen (in der Regel) mehreren speziellen und einem allgemeinen Element. Das allgemeine Element fasst in der Beziehung Eigenschaften zusammen, so dass alle Spezialisierungen sich kompatibel verhalten können.
gerichtete Assoziation	Eine gerichtete Assoziation ist eine einseitige relationale Beziehung.
Gleichzeitigkeit	Die Gleichzeitigkeit beschreibt den Ablauf von verschiedenen Aktionen (Funktionen) zu ein und dem selben Zeitpunkt.
GUI	Graphical User Interface
Hierarchie	Eine Hierarchie stellt die Rangordnung der Abstraktionen dar.
IBEL	Integrated Business Engineering Language. Formale Datenbeschreibungssprache der IBM.
Inheritance	⇒ Vererbung
Interaktionsdiagramm	⇒ Sequenzdiagramm
IT	Information Technology

Kardinalität	Die Kardinalität beschreibt die Anzahl der effektiven Beziehungen von einem Element ausgehend.
Kapselung	Als Kapselung wird der Prozess der Aufteilung der Elemente einer Abstraktion bezeichnet. Kapselungen dienen dazu, die Implementierung und die Schnittstelle einer Abstraktion voneinander zu trennen.
Klasse	Eine Klasse ist die abstrakte Beschreibung einer real existenten Sache.
kohärent	zusammenhängend
kohäsiv	zusammenhaltend
Komposition	Eine Komposition ist eine spezielle Form der $\Rightarrow$ Aggregation. Eine Komposition entsteht, wenn das enthaltene Element existenzabhängig vom enthaltenden Element ist.
mehrgliedrige Assoziation	Eine mehrgliedrige Beziehung ist eine Relation an der mehr als zwei Assoziationsrollen beteiligt sind.
Methoden	Die Methoden einer Klasse beschreiben die Funktionalität. Eine einzelne Methode definiert in der Regel eine in sich abgeschlossene Funktion.
Modularisierung	Die Herbeiführung der Modularität oder Der Designvorgang für das Erreichen der Modularisierung.
Modularity	$\Rightarrow$ Modularität
Modularität	Modularität ist die Eigenschaft eines Systems, das in eine Menge von kohäsiven und lose gekoppelten Modulen zerlegt wird.
Multiplizität	Die Multiplizität beschreibt die Menge der erlaubten $\Rightarrow$ Kardinalitäten.
n-ary association	$\Rightarrow$ mehrgliedrige Assoziation
n-gliedrige Assoziation	$\Rightarrow$ mehrgliedrige Assoziation
Nachrichtendiagramm	$\Rightarrow$ Sequenzdiagramm
Objekt	Ein Objekt ist die physische Manifestation einer Klasse. Das Objekt beschreibt eine real existente Sache. Es ist definiert über seinen Namen oder seine Identifikation. Ein Objekt hat einen bestimmten Zustand.

OCL	Object Constraint Language. Einfache formale Sprache, mit der UML-Modelle weitere Semantik zugeführt werden kann. OCL ist aus der $\Rightarrow$ IBEL hervorgegangen.
OMG	Object Management Group. The Object Management Group (OMG) was founded in May 1989 by eight companies: 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems and Unisys Corporation. In October 1989, OMG began independent operations as a non-profit corporation. Through the OMG's commitment to developing technically excellent, commercially viable and vendor independent specifications for the software industry, the consortium now includes over 800 members. As OMG moves forward in establishing CORBA as the "Middleware that's Everywhere" through its worldwide standard specifications: CORBA/IIOP, Object Services, Internet Facilities and Domain Interface specifications. OMG is headquartered in Framingham, Massachusetts, USA, with international marketing partners in the UK, Germany, Japan, India and Australia. OMG defines object management as software development that models the real world through representation of "objects." These objects are the encapsulation of the attributes, relationships and methods of software identifiable program components. A key benefit of an object-oriented system is its ability to expand in functionality by extending existing components and adding new objects to the system. Object management results in faster application development, easier maintenance, enormous scalability and reusable software. (siehe auch www.omg.org)
OOA	Objektorientierte Analyse
OOD	Objektorientierter Design
OOT	Objektorientierte Technik
partitionierte Assoziation	$\Rightarrow$ qualifizierte Assoziation
Pattern	Standardisierter Lösungsansatz für spezifische Problemstellungen
Persistenz	Die Persistenz ist die Eigenschaft eines Objektes, mit deren Hilfe es über Raum und Zeit hinweg existieren kann (z.B. Speicherung in einem anderen Medium).
Polymorphie	Vieldeutigkeit. Die Polymorphie beschreibt die verschiedenen Verhalten eines Objektes (oder einer Klasse) relativ zu seinem Typ, respektive zur Semantik der Kommunikation.
qualifizierte Assoziation	Eine qualifizierte Assoziation ist eine Assoziation, die die referenzierte Menge der Objekte durch qualifizierende Attribute in Partitionen (Gruppen) unterteilt.
RUP	Rational Unified Process
Scenario	$\Rightarrow$ Sequenzdiagramm. Manchmal auch als Objektdiagramm verstanden (alt).

Sequenzdiagramm	Das Sequenzdiagramm zeigt die zeitliche Abfolge des Nachrichtenaustausches zwischen einer bestimmten Menge von Objekten.
SEU	Software Entwicklungsumgebung
Spezialisierung	Taxonomische Beziehung zwischen einem allgemeinen und einem speziellen Element. Das spezielle Element fügt in der Beziehung weitere Eigenschaften hinzu, erweitert die Semantik aber verhält sich zum allgemeinen Element kompatibel.
State	⇒ Zustand
State-Event	⇒ Zustandsdiagramm
taxonomisch	systematisch, Einordnung eines Teils im Ganzen
Ternäre Assoziation	⇒ mehrgliedrige Assoziation
Typing	⇒ Typenprüfung
Typenprüfung	Die Typenprüfung verhindert die unbewusste Überführung einer Information vom einen Typ in den andern.
UML	Unified Modeling Language.
unidirekte Assoziation	⇒ gerichtete Assoziation
Vererbung	Die Vererbung definiert die Spezialisierung- respektive Generalisierung von real unterschiedlichen Dingen.
Zustand	Ein Zustand stellt eine Menge von definierten Atributwerten in einem bestimmten Objekt dar.
Zustandsdiagramm	Diagramme oder Modellierormen, die Zustände von Objekten und den Übergang vom einen Zustand in den andern beschreiben.
Zustandsübergang	Der Zustandsübergang ist das Ereignis das ein Objekt von einem Zustand in einen andern Zustand überführt.

8.5 Stichwortverzeichnis

A

Abhängigkeit · 3-17
Abstraktion · 2-4
Active Class · 3-6
Aggregat · 3-16
Aggregation · 3-16
Aggregatklasse · 6-10
Aggregat · 6-9
Akteur · 3-5, 3-21, 5-1
Akteure · 5-5
Akteurmodell · 3-22
Aktive Klasse · 3-6
Aktivität · 3-23
Aktivitätendiagramm · 3-23
Aktivitätenmodell · 6-1
Aktivitätsmodellierung · 4-5
Amigos · 1-5
Analyse / Design · 1-8
Analyseresultate · 4-2
Anmerkung · 3-8
Annotation · 3-8
Anwendungsfall · 5-1
Anwendungsfall · 3-5
 Beschreibung · 3-22
Anwendungsfallanalyse · 4-3
Anwendungsfalldiagramm · 3-21, 5-1
Architektur · 4-3, 5-6
 MDI · 5-7
 SDI · 5-7
 standalone · 5-7
Architekturmuster · 7-2
Assoziation · 3-11, 6-8
 abgeleitete · 3-13
 attributierte · 3-12
 gerichtete · 3-14
 Kardinalität · 3-12
 mehrgliedrige · 3-14
 Multiplizität · 3-12
 qualifizierte · 3-13
 rekursive · 3-12
 Sichtbarkeit · 3-11
Assoziationsklasse · 3-12
Attributwert · 3-25

B

Behaviour · siehe Verhalten
Benutzersichtanalyse · 6-3
Beziehungen · 3-8
Booch · 1-4, 1-5
Business Modeling · 1-8

C

Changemanagement · 1-8

Classes · 3-2
Client - Server · 5-8
Collaboration · 3-5
Component · 3-6
Composite Pattern · 7-6
Composition · 3-16
Concurrency · siehe Gleichzeitigkeit
Configuration and Changemanagement · 1-8
Constraints
 Generalisierung · 3-10
Containmentdiagramm · 3-18
CRC-Methode · 6-3

D

Datenflussdiagramm · 1-3
Dependency · 3-17
Deployment · 1-8
Design-Pattern · 7-1
Diagramme
 Aktivitäten · 3-23
 Anwendungsfall · 3-21
 generell · 3-17
 Klassen · 3-18
 Kollaboration · 3-28
 Komponenten · 3-29
 Objekt · 3-20
 Sequenz · 3-26
 Verteilung · 3-29
 Zustand · 3-25
Diskriminator · 3-9
Document-View Pattern · 7-13
dynamische Polymorphie · 6-7

E

Einfachvererbung · 3-8, 3-10
Einzelteil · 3-16
Encapsulation · 2-5
Entity Relationship Diagram · 1-3
Entwickler
 Rollen · 4-1
Entwurfsmuster · 7-2
Environment · 1-8
ERD · 1-3
Ereignis · 3-25
EU · 5-6
Existenzabhängigkeit · 6-9

F

Fachklassenmodellierung · 4-6
Funktionsbibliothek · 1-3

G

ganzheitliche Denkweise · 1-4
Generalisierung · 3-8, 6-4
Gesamtprozess · 4-1
Geschäftsklassenanalyse · 4-4
Geschäftsprozess · 5-1
Gleichzeitigkeit · 2-10
Grady Booch · siehe Booch
Grundkonzepte · 2-1
Gruppierung · 3-7
GUI Design · 4-7

H

Hierarchie · 2-7

I

Identifizieren von Klassen · 6-2
Idiome · 7-2
Implementation · 1-8
Inheritance · 3-8. siehe Vererbung
Interaktion · 3-7
Interface · 3-5
Iterations · 1-8
Ivar Jacobson · siehe Jacobson

J

Jackson · 1-3
Jacobson · 1-5
James Rumbaugh · siehe Rumbaugh

K

Kapselung · 2-5
Kardinalität · 3-11
Klasse · 2-1, 3-2
 aktive · 3-6
Klassen
 Verbale Beschreibung · 3-19
Klassendiagramm · 3-18
Klassenmodell
 Analyse · 5-9
 Design · 6-2
Klassenmodellierung
 Benutzersichtanalyse · 6-3
 CRC-Methode · 6-3
 Realitätsbeobachtung · 6-3
 Textanalyse · 6-3
Knoten · 3-6
Kollaboration · 3-5
 Stereotypen · 3-28
Kollaborationsdiagramm · 3-28
Kollaborationsmodell · 6-14
Kollaborationsmodellierung · 4-5
Komponente · 3-6
Komponenten · 3-29
Komponentenbildung · 4-6

Komponentenentwurf · 4-7
Komponentenmodell · 6-15
Komposition · 3-16, 6-9
Konzepte
 grundlegende · 2-1

M

Mehrfachvererbung · 3-10
Message · siehe Mitteilung
Mitteilung · 3-7
Modell
 Aktivitäten · 6-1
 Iterativ · siehe Spiralmodell
 Kollaboration · 6-14
 Komponenten · 6-15
 Sequenz · 6-13
 Spiral · 1-7
 Wasserfall · 1-6
Modell View Controller Pattern · 7-12
Modularisierung · 2-6
Multiple Inheritance · 3-10
Multiplizität · 3-11, 6-11
Muster · 7-1
MVC Pattern · 7-12

N

Nachricht · 3-26
natürliche Denkweise · 1-4
Node · 3-6
Notice · Anmerkung

O

Oberklasse · 3-8
Object Managment Group · 1-5
Object Modeling Technique · 1-4
Object oriented Software Engineering · 1-4
Objects · 3-4
Objekt · 2-1, 3-4
 Definition · 2-2
 Identität · 2-3
 Status · 2-2
 Verhalten · 2-2
Objektdiagramm · 3-20
objektorientierter Softwareentwurf · 1-4
Objektspeicherung · 2-11
Objektzustand · 3-24, 3-25
OMG · 1-5
OMT · 1-4
OOA
 Teilprozess · 4-3
 Vorgehen · 4-1
OOA und OOD Zielsetzung · 4-2
OOD
 Teilprozess · 4-5
 Vorgehen · 4-1
OOSE · 1-4
Ordnungsstrukturen · 6-4

P

Package · siehe auch Gruppierung
Paket · 3-3
Partition · 3-13
Pattern · 7-1
 Architecture · 7-2
 Composite · 7-6
 Design · 7-2
 Document-View · 7-13
 Idiome · 7-2
 Modell View Controller · 7-12
 Singleton · 7-4
Persistenz · siehe Objektspeicherung
Planen
 Wie · 1-2
 Wieso · 1-1
Planung · 1-1, 4-1
 Umsetzungsplanung · 1-2
 Zielplanung · 1-2
Polymorphie
 dynamische · 6-7
 statische · 6-6
Programspecification · 1-3
Projectmanagement · 1-8
Prozessmodell · 4-1
PSPEC · 1-3

R

RAD · 5-1
Rational Unified Process · 4-1
Realisierung · 3-17
Realitätsbeobachtung · 6-3
Relations · 3-8
Relationsdiagramm · 3-18
Requirements · 1-8
Rumbaugh · 1-5
RUP · 4-1

S

Sequenzdiagramm · 3-26
Sequenzmodell · 6-13
Sequenzmodellierung · 4-5
SEU · 5-6
SEU Analyse · 4-3
Sichbarkeit · 3-3
Singleton Pattern · 7-4
Software-Pattern · 7-1
Spezialisierung · 3-8, 6-4
Spiralmodell · 1-7
State · siehe Status
statische Polymorphie · 6-6
Status · 3-7
Stereotyp · 3-3
Structures · 3-2
Struktur · 3-2
Strukture Chart · 1-3
Strukturelement · siehe Struktur
Strukturierte Analyse und Design · 1-3

T

Teilprozess OOD · 4-5
Test · 1-8
Textanalyse · 6-3
The · 1-8
Transition · 3-23
 Synchronisieren · 3-24
 Teilen · 3-24
Typenprüfung · 2-9
Typing · siehe Typenprüfung

U

UML · 1-4, 1-5
 Darstellung
 Abhängigkeit · 3-17
 Aggregation · 3-16
 Aktive Klasse · 3-6
 Anmerkung · 3-8
 Anwendungsfall · 3-5
 Assoziation · 3-11
 Generalisierung · 3-8
 Gruppierung · 3-7
 Interface · 3-5
 Klasse · 3-2
 Knoten · 3-6
 Kollaboration · 3-5
 Komponente · 3-6
 Komposition · 3-16
 Mitteilung · 3-7
 Objekt · 3-4
 Spezialisierung · 3-8
 Status · 3-7
 Übersicht · 3-1
Unified modeling language · 1-5
Unified modeling Language · 1-4
Unterklasse · 3-8
Use case · 5-1
Use Case · siehe Anwendungsfall

V

Vererbung · 2-8, 3-8
Vererbungsdiagramm · 3-18
Verhalten · 3-7
Verteilung · 3-29
Visibility · 3-3
Vorgehensschritte OOA und OOD · 4-1

W

Wasserfallmodell · 1-6
Wie planen · 1-2
Wieso planen · 1-1
Workflow
 Process · 1-8
 Supporting · 1-8

Z

Zielsetzung · 4-2

Zustand · 3-25
Zustandsdiagramm · 1-3, 3-25
Zustandsübergang · 3-25

