

Vorlesungsskript

Software Engineering

Studiengänge Infotronik 3. Semester, Medientechnik , 3. Semester,

Wintersemester 2010/2011

Hochschule Deggendorf

Prof. Dr. Peter Jüttner

peter.juettner@fh.deggendorf.de

Stand 22.9.10

Vorwort

Diese Vorlesung entstand ursprünglich im Sommersemester 2008 an der Fachhochschule Regensburg für die Studiengänge Wirtschaftsinformatik und Technische Informatik in Zusammenarbeit mit meinem damaligen Kollegen Dr. Christian Flug, dem ich an dieser Stelle für die hervorragende Kooperation herzlich danken möchte.

Des Weiteren möchte ich Hr. Prof. Jürgen Mottok von der Fachhochschule Regensburg besonders danken für die freundliche Überlassung von Material zum Thema SW Entwurf und UML.

Inhaltsverzeichnis

1.	Einleitung	4
2.	Ziele und Motivation	4
2.1.	Ziele	4
2.2.	Motivation	4
	Klassifizierung	4
	Eigenschaften von Software	5
	Anforderungen an Software	5
	Motivation des Software Engineerings als eigene Disziplin.....	6
	Software Engineering – Der Weg aus der Krise	7
3.	Elemente des Software Engineering	8
3.1.	Überblick	8
3.2.	Requirments Engineering	9
3.3.	Software Design	10
3.4.	Codierung	11
3.5.	Software Test.....	11
3.6.	Projektmanagement	11
3.7.	Software Qualitätssicherung	12
3.8.	Software Konfigurationsmanagement.....	13
	Versionsmanagement Versionsmanagement hat folgende Aufgaben:.....	14
	Änderungsmanagement	14
	Build & Release Management.....	14
3.9.	Notationen	15
3.10.	Dokumente, Templates.....	15
3.11.	Methoden.....	15
3.12.	Tools.....	15
3.13.	Rollen	16
4.	SW Engineering Methodik.....	16
4.1.	Projektmanagement	16
	Definitionen.....	16
	Projektplanung	17
4.2.	Requirements Engineering	26
	Beschreibung von Requirements.....	26
	Requirements in allen Phasen der Software Entwicklung.....	26
	Funktionale und Nicht-Funktionale Requirements	26
	Stakeholder.....	27
	Methodik	28
4.3.	Software Entwurf	37
	Software Architketur	37
	Software-Feinentwurfs	47
4.4.	Objektorientierte Analyse.....	50
	Objektorientierte Methodologie	50
4.5.	UML	54
	Unterschiede UML 1 zu UML 2	56
	UML Konzeptüberblick	56
	Konzept-Katalog - Diagramme der UML 2	57
	Diagramme der UML 2 und Ihre Anwendung – Eine Übersicht	58
4.6.	Statische Modellierung in der Objektorientierten Analyse und Klassendiagramm .	60
4.7.	Dynamische Modellierung	64
4.8.	Objektorientiertes Design.....	66
	Objektorientierte Architekturmodellierung.....	66

Statische Modellierung im OOD	68
Dynamische Modellierung im Objektorientierten Design	75
4.9. Implementierung (Codierung)	80
Historische Entwicklung	80
Unterschiede C zu C++/Java	82
Unterschiede C++ zu Java	82
Codegenerierung aus UML (Vom OOD zur Implementierung)	83
Round-Trip-Engineering	89
Codierungsregeln	90
4.10. Software Test	96
Definition	96
Motivation	96
Abgrenzung Test – Korrektheit beweisen - Debuggen	97
Statischer Test – Dynamischer Test	97
Testfall	98
Testphasen	99
Test Phasen – Modultest	100
Test Phasen – Integrationstest	100
Test Phasen – Validierung	105
Abgrenzung Software Test – Systemtest	105
Testtechniken und Teststrategien	105
Test Techniken im Detail: Überblick	107
Test Techniken - Black Box	108
Test Techniken - White Box	111
Test Techniken – Weitere Begriffe	113
Test Dokumentation	114
Exkurs: Automotive Projekte und Software	115
Reduktion des Testumfangs	117
Intuitiver Ansatz	117
Exkurs: Testen objektorientierter Software	118
Exkurs: Testdatengenerierung aus UML Modellen	120
Exkurs: Modellbasiertes Tests	121
Exkurs: Testen grafischer Bedienoberflächen	121
Do's und Dont's, Erfahrungen, Probleme, Tipps und Tricks	122
5. Literatur	124

1. Einleitung

Dieses Skript entstand im Sommersemester 2009 zur Vorlesung SW Engineering für den Studiengang Technische Informatik an der FH Regensburg. Diese Vorlesung wurde in dieser Form erstmalig im Sommersemester 2008 von meinem Kollegen Hr. Dr. Christian Flug für den Studiengang Wirtschaftsinformatik und von mir für den Studiengang Technische Informatik gehalten. An dieser Stelle möchte ich Christian Flug herzlich für die Zusammenarbeit bei der Erstellung der Vorlesung danken.

2. Ziele und Motivation

2.1. Ziele

Ziele der Vorlesung sind

- die wichtigen Begriffe des industriellen Software Engineering einführen.
- die wichtigen Methoden des industriellen Software Engineering kennen lernen
- die zugehörigen Methoden im Rahmen kleinerer Aufgaben während der Vorlesung und im Rahmen der Projektarbeit üben
- speziell in der Projektarbeit Teamarbeit üben

2.2. Motivation

Der Begriff „Software“ wird auf vielfache Weise definiert, hier seien nur zwei Definitionen herausgegriffen:

- Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system. (IEEE Standard Glossary of Software Engineering)
- Computer Programme, Abläufe, Regeln, und möglicherweise zugehörige Dokumentation und Daten, die zum Ablauf eines Computersystems gehören.

Klassifizierung

Software kann auf vielfältige Weise klassifiziert werden:

- Software als eigenständiges Produkt: Software, die auf bestimmten Geräten installiert oder integriert werden kann und dort abläuft. Z.B. Office Software für einen PC , Spiel für ein Handy
- Software als Bestandteil eines Produkts, z.B. eingebettete Software in einer elektronischen Airbag- oder Motor-Steuerung im Auto oder im Flugzeug
- Generisches Produkt : Entwicklung ohne ein festes Zielsystem oder ohne einen Kundenauftrag
- Einzelanfertigung: vereinbartes Produkt im Kundenauftrag
- Echtzeitsoftware: Software, die bestimmte ggf. kritische Zeitbedingungen erfüllt, z.B. für eingebettete Systeme im Automobilumfeld
- Verteilte Software: Software, die auf verschiedenen Computern abläuft und dabei eine Gesamtfunktion erfüllt
- Monolithische Software: Software, die als ganzes auf einem Computer läuft
- Betriebssystemsoftware: meist HW-nahe Software, die anderen Programmen bestimmte Dienste leistet, ohne die diese nicht ablaufen können.
- Applikationssoftware: Software, die auf einem Betriebssystem abläuft und Anwenderfunktionen erfüllt.
- Kommunikationssoftware: Software, die Nachrichtenaustausch durchführt, z.B. in Telefonvermittlungssystemen oder für den Betrieb von Kommunikationsbussen in Fahrzeugen
- Grafische Software: Software für grafische Oberflächen

Anmerkung: Die oben aufgeführten Kategorien sind nicht disjunkt. Eine bestimmte Software kann in mehrere Kategorien fallen.

Eigenschaften von Software

- Software ist nicht fassbar, d.h. Software kann nicht berührt werden
- Software nutzt sich nicht ab, unterliegt keinem Verschleiß
- Software veraltet, obwohl sie sich nicht abnutzt
- Software wird in identischen Exemplaren ausgeliefert, z.T. in sehr großen Stückzahlen
- Software hat (theoretisch) keine Grenzen, d.h. je größer der Speicher, je leistungsfähiger der Computer in bzw. auf dem die Software abläuft, desto größer und komplexer kann die Software sein
- Software ist meist zu teuer, zumindest aus Sicht derjenigen, die die Entwicklung bezahlen müssen
- Software ist oft / meist / immer fehlerhaft, d.h. es gibt eigentlich keine fehlerfreie Software. In vielen oder den meisten Fällen sind die Fehler allerdings tolerierbar.

Anforderungen an Software

An Software werden in der Regel zahlreiche Anforderungen gestellt

- Korrekt (fehlerfrei oder wenigstens fehlerarm ?)
- Portierbar, d.h. relativ leicht in eine andere HW- oder SW-Umgebung zu bringen, verwendbar in verschiedenen Umgebungen
- Wartbar, d.h. Fehler sind leicht und schnell korrigierbar
- Erweiterbar, d.h. neue Funktionen sind mit geringem Aufwand integrierbar
 - Korrekt (fehlerfrei oder wenigstens fehlerarm ?)
 - Portierbar, d.h. verwendbar in verschiedenen Umgebungen
 - Wartbar, d.h. Fehler sind schnell korrigierbar
 - Erweiterbar, d.h. neue Funktionen sind leicht integrierbar
 - Zugriffssicher, d.h. Schutz vor Benutzung durch Nicht-Autorisierte
 - Klein, d.h. geringer Speicherplatzverbrauch
 - Schnell, d.h. kurze Reaktionszeit
 - Testbar, d.h. ausreichende Testabdeckung mit akzeptablem Aufwand erreichbar
 - Nachvollziehbar, d.h. Entwicklungsschritte dokumentiert und erkennbar
 - Zertifizierbar, d.h. Genügt bestimmten Standards (z.B. funktionale Sicherheit, Integration in bestimmte Umgebungen, Compiler). Dies kann auch den Entwicklungsprozess betreffen
 - Dokumentiert, d.h. nicht nur der Code ist vorhanden, sondern auch Dokumente, die die Anforderungen, die Lösung, den Test und das zugehörige Projektmanagement detailliert beschreiben
 - Zuverlässig, d.h. funktioniert immer (gleich in gleicher Umgebung)
 - Benutzbar
 - Software ist an die Bedürfnisse der Benutzer angepasst
 - Die Benutzerschnittstelle muss ergonomisch und selbsterklärend sein.
 - Das Benutzermanual muss in der notwendigen Detaillierung zur Verfügung stehen
 - Effizient, d.h. wirtschaftliche Nutzung der zur Verfügung stehenden Ressourcen der Umgebung.
 - Kompatibel, d.h. arbeitet mit anderer SW zusammen
 - Kostengünstig
 - Schnell verfügbar
 - Messbar
 - während der Entwicklung
 - als Produkt
 - ...

Zu beachten ist, dass diese Anforderungen sich teilweise widersprechen. Das bedeutet es sind nicht immer alle gleichzeitig erfüllbar. So ist z.B. eine Software, die möglichst klein bzgl. des Speicherplatzes und effizient bzgl. der Geschwindigkeit entwickelt wurde, eher schlechter portierbar als eine Software, bei der die Portierbarkeit vor Effizienz im Vordergrund steht. Die Erfahrung zeigt, dass Fehlerfreiheit selten erfüllbar ist. Eher hat eine Software Fehler, die beim Gebrauch akzeptiert wird. Auf der anderen Seite gibt es Anforderungen, die sich gegenseitig bedingen. So wird eine Software die detailliert dokumentiert ist, einfacher zu warten sein, als ein nicht dokumentiertes Programm.

In der Praxis müssen immer die Eigenschaften ausgewählt werden, die für eine zu entwickelnde Software im Vordergrund stehen. Bei Widersprüchen müssen geeignete Kompromisse gefunden werden.

Motivation des Software Engineerings als eigene Disziplin

Warum hat sich Software Engineering als Ingenieurdisziplin entwickelt? Schon relativ früh, in den 60er Jahren des vorigen Jahrhunderts wurde die Erfahrung gemacht, dass Software oft fehlerhaft ist und dass die auftretenden Fehler zum Teil kuriose, katastrophale und sehr oft teure Folgen haben. In der Folge davon verbreitete sich bereits damals der Begriff der **Software Krise**, die z.T. bis heute (auch aus Gründen immer leistungsfähigerer Computer und größerer Speicher immer noch anhält).

Einige Beispiele seien hier aufgeführt:

- Juli 1962, Verlust der ersten amerikanischen Venussonde Mariner 1
Entscheidender Fehler: Punkt statt Komma im Fortran Sourcecode!
- Januar 1990: 70 Millionen von 138 Millionen Ferngesprächen innerhalb USA konnten 9 Stunden lang nicht vermittelt werden.
→ Schaden ca. 75 Mio \$, bei AT&T (ohne Folgeschäden), mindestens so viel bei AT&T Kunden,
Ursache: Ein break-Befehl von C wurde falsch eingesetzt (Der Fehler wurde eine Woche später gefunden, existierte seit einer Programm-Optimierung 4 Wochen vorher)
- F-16 Jagdflugzeug, ca. 1978: Erstes Jagdflugzeug der USA mit Computer-Steuerung. Beim Überfliegen des Äquators stellt sich das Flugzeug auf den Kopf wegen eines Vorzeichenfehler im Algorithmus des Programms bei der Berücksichtigung der geographischen Breite
(Der Fehler wurde allerdings rechtzeitig beim Test gefunden und trat nicht, wie manchmal behauptet beim Flug auf)
- Nimrod-Luftüberwachungssystem (UK): Das Problem waren unklare Anforderungen. Die Entwicklungszeit war ca. Mitte 70er bis Mitte 80er Jahre mit Entwicklungskosten von mehrere hundert Million Pfund. Am Ende wurde das System durch AWACS ersetzt.
- August 1988 Flug der 1. Marssonde der UdSSR: Beim Senden eines Kommandos kurz nach dem Start wurde ein Buchstabe vergessen. Das verkürzte Kommando wurde jedoch als ein anderes (nur beim Bodentest vorgesehenes) Kommando interpretiert. Folge war eine Rotation der Raumsonde, so dass der Kontakt mit ihr abbrach. (Auch die Schwester-Sonde Phobos 2 ging später (beim Marsmond Phobos) wegen eines Software-Fehlers verloren)
- 1994: Eröffnung des Denver International Airport um 9 Monate verzögert wegen Softwareproblemen im Gepäcktransport-System
→ Schaden?
- 1996, erster Start der Ariane 5 Rakete mit Selbstzerstörung auf Grund eines SW Fehlers. Es wurde SW aus Ariane 4 in einer "unpassenden" Umgebung wiederverwendet. Eine

5mal höhere Geschwindigkeit führte zu einem Überlauf einer Variablen. Der Schaden betrug ca. 500 Mio \$

- März 1999: Fehlstart einer Titan/Centaur-Rakete wegen falscher Software-Version
- September 1999: Verlust der Sonde "Mars Climate Orbiter" wegen falscher Einheitenrechnung (metrisch / Fuss)
- US-Unternehmen verloren im Jahr 2000 insgesamt 100 Milliarden US-\$ wegen defekter Software.

1972 fasste E. W. Dijkstra das Problem folgendermaßen zusammen

„Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu nutzen.“

Software Engineering – Der Weg aus der Krise

Software Engineering (= Software Technik) definiert einen Weg aus der Software Krise. Ziel ist es sicherzustellen, dass die Anforderungen an die Software im Rahmen der Entwicklung auf systematische Weise erfüllt werden. Dazu werden im Rahmen des Software Engineering geeignete Mittel (z.B. Prozesse, Methoden) angewendet.

Es gibt zahlreiche Definitionen des Begriffs „Software Engineering“. Einige seien hier exemplarisch aufgeführt:

- Software Engineering: The establishment and use of sound engineering principles in order to obtain economically software that is reliable and runs on real machines (F.L. Bauer, NATO-Konferenz Software-Engineering 1968)
- Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them (Berry Boehm 76)
- Software-Technik ist die zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden, Konzepten, Notationen und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen. (Balzert)
- Eine *technische Disziplin*, die sich mit *allen Aspekten der Softwareherstellung* befasst, von den frühen Phasen der Systemspezifikation bis hin zur Wartung des Systems, nachdem sein Betrieb aufgenommen wurde.
Technische Disziplin
 - Suche nach Lösungen (auch, wenn keine theoretischen Grundlagen existieren) innerhalb organisatorischer und finanzieller Beschränkungen
 - Alle Aspekte der Softwareherstellung
 - Nicht nur technische Aspekte, sondern auch Projektverwaltung, Entwicklung neuer Theorien, Methoden, Werkzeuge etc.(Ian Sommerville)
- Die IEEE Computer Society definiert Software Engineering als
 - The application of a
 - systematic,
 - disciplined,
 - quantifiable

- approach to the
 - development,
 - operation, and
 - maintenance
 - of software;
- that is, the application of engineering to software.

Die verschiedenen Definitionen lassen sich folgendermaßen zusammenfassen: Software Engineering bedeutet:

- Ingenieurmäßiges Vorgehen, d.h. nach bestimmten Regeln und Verfahren
- Wissenschaftliche Erkenntnisse als Basis
- Empirische Erkenntnisse als Basis
- Wirtschaftliches Vorgehen, d.h. Budgets werden eingehalten
- Lauffähige Software als Ergebnis
- Zuverlässige Software, d.h. Fehler werden minimiert
- Unterstützung von Entwicklung, Betrieb und Wartung über den kompletten Lebenszyklus
- Die Entwicklung wird vorhersagbar(er) bezüglich Ergebnis, Kosten, Zeit, Qualität, ...
- Die Entwicklung wird messbar(er) bezüglich Ergebnis, Kosten, Zeit, Qualität, ...
- Die Entwicklung wird wiederholbar (mit dem gleichen oder ähnlichen Ergebnis)
- Unabhängigkeit von Personen, d.h. Teams mit unterschiedlichen Personen werden unter gleichen Voraussetzungen sehr ähnliche Ergebnisse liefern
- Vermittelbarkeit (in Forschung und Lehre)

Zu beachten ist, dass Software Engineering nur einen Methoden-Baukasten bereitstellt, mit dem Software entwickelt werden kann. Für ein konkretes Projekt sind die geeigneten Elemente aus dem Software Engineering Baukasten auszuwählen, d.h. der Baukasten ist richtig anzuwenden

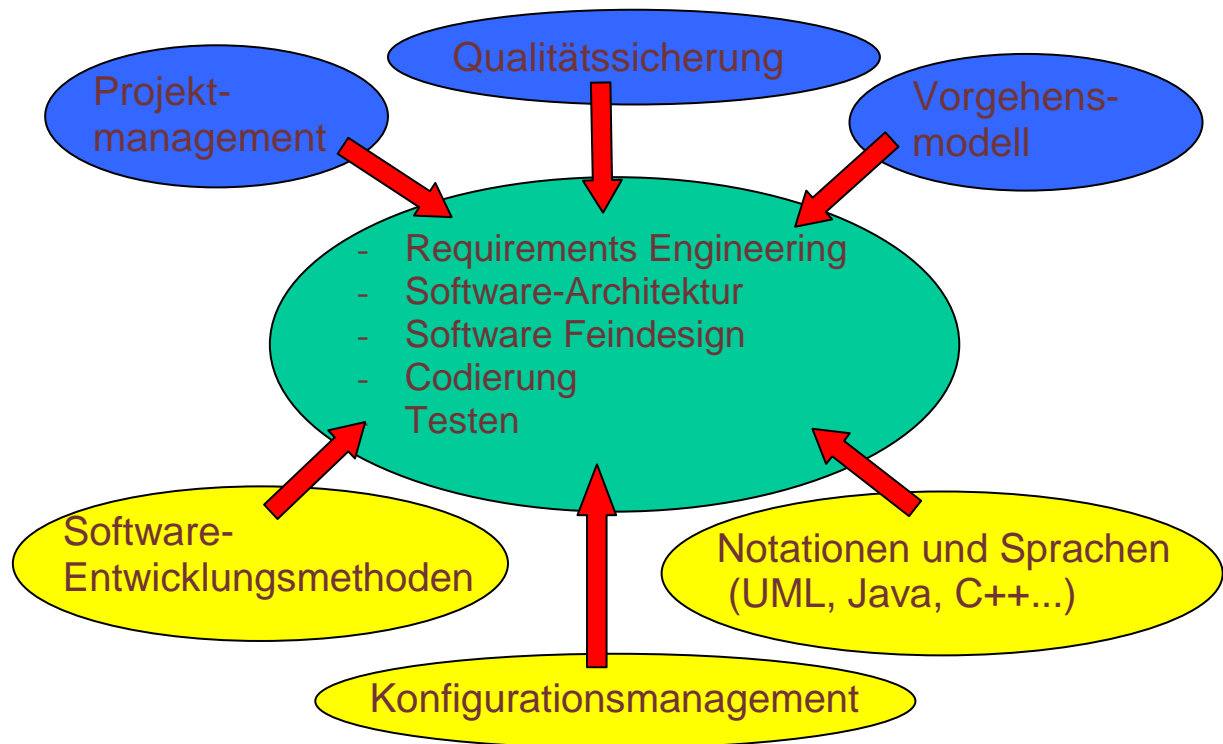
Mit der Disziplin des Software Engineering entsteht auch der Software Engineer (= Software Ingenieur, Informatiker) als neuer Beruf.

3. Elemente des Software Engineering

Dieses Kapitel soll die wesentlichen Elemente des Software Engineerings mit ihren Zielen vorstellen ohne auf die Methodik einzugehen. Die Methodik der SW Engineering Elemente ist Inhalt der darauf folgenden Kapitel.

3.1. Überblick

Die folgende Grafik gibt einen Überblick über alle Bereiche des Software Engineering:



Die „klassischen“ Engineering Bereiche Requirements Engineering, Software-Architektur, Software Feindesign, Codierung und Testen werden unterstützt durch Projektmanagement, Qualitätssicherung und Vorgehensmodelle. Entwicklungsmethoden definieren das Vorgehen beim Engineering, Notationen standardisieren, abstrahieren und vereinfachen die Entwicklung. Konfigurationsmanagement verwaltet systematisch alle Ergebnisse der kompletten Entwicklung.

Die dargestellten Elemente werden ergänzt durch Rollen und Tools. In den folgenden Kapiteln werden die einzelnen Elemente detaillierter beschrieben.

3.2. *Requirments Engineering*

Ziel des Requirements Engineering ist es, **Anforderungen** an ein System zu beschreiben im Sinne von

- Diensten, die ein Kunde von einem System erwartet und
- die Gegebenheiten, unter denen das System entwickelt wird und laufen soll

Das Requirements Engineering (auch Anforderungsanalyse) ist der Prozess des

- Herausfindens (Elicitation),
- Analysierens (Analysis),
- Dokumentierens (Specification) und
- Überprüfens (Validation & Verification)

dieser Anforderungen

Anforderungen entstehen in einem Projekt auf verschiedenen Ebenen und zu verschiedenen Zeitpunkten, d.h. nicht nur zu Beginn des Projekts, z.B.

- Anforderungen des Kunden, dokumentiert im Lastenheft
- Interne Anforderungen, dokumentiert im Pflichtenheft
- Anforderungen an einzelne SW Bausteine, Komponenten oder Module, dokumentiert in der Anforderungen an einzelne Funktionen, dokumentiert im Feindesign
- Anforderungen an den Test
- Architekturbeschreibung

3.3. *Software Design*

SW Design ist der Prozess zum Definieren

- der Architektur
 - der Komponenten,
 - der Schnittstellen und
 - anderer Charakteristika (Datenstrukturen, Algorithmen etc.)
- eines Systems oder einer Komponente sowie das Ergebnis dieses Prozesses.
(gemäß IEEE Std 610.12-1990)

Im Software Design wird basierend auf der Analyse der Anforderungen eine Beschreibung der

- internen Struktur und
- Organisation

eines Systems erstellt. Das Software Design bildet die Grundlage für die Codierung, theoretisch so, dass die Codierung von einer unabhängigen Person durchgeführt werden könnte. Dabei werden sowohl statische Aspekte der SW (z.B. Modulstruktur, Schnittstellen, Speicherstrukturen) als auch dynamische Aspekte (z.B. Datenfluss, Kontrollfluss, Prozesse, Threads, Dynamische Objekte) beschrieben.

Software Design gliedert sich in

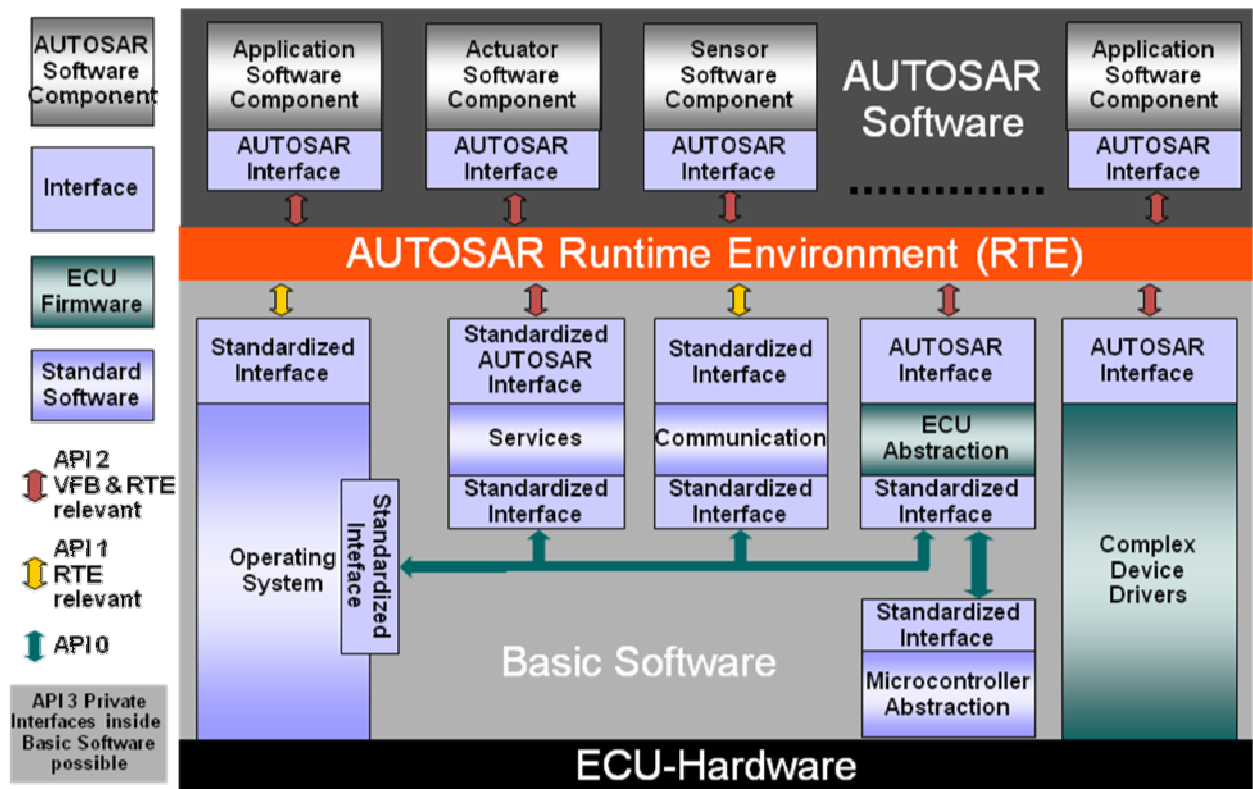
- SW Architektur (Beschreibung der Grobstruktur der Software auf Ebene größerer SW Bausteine)
- SW Feindesign (Beschreibung feinerer, detaillierter Strukturen auf Ebene kleiner SW Bausteine wie
 - Code Module
 - Schnittstellen zwischen Modulen
 - Funktionsschnittstellen

Das Ergebnis der Designaktivitäten muss in geeigneter Form dokumentiert werden. Theoretisch soweit, dass, wie schon erwähnt theoretisch die Codierung von einer anderen Person durchgeführt werden könnte.

In der Praxis sind oft Teile der Architektur und/oder des Feindesigns vorherbestimmt, da z.B.

- aus einem Vorgängerprojekt übernommen (Wiederverwendung)
- durch Anforderung vorgegeben (bei Verwendung von vorgegebenen Standard-Architekturen)
- durch Verwendung von Entwurfsmustern (Design Patterns, Wiederverwendung von Lösungen)
- durch die HW Umgebung, z.B. verteilte Systeme, die auf mehreren Controllern ablaufen.

Als Beispiel einer Standardarchitektur sei hier die AUTOSAR Architektur dargestellt. Diese ist eine in der Automobilindustrie standardisierte Software Architektur, die z.B. für Karosseriesteuergeräte oder Motorsteuerungen verwendet wird.



3.4. Codierung

Beim Codieren wird das Design in eine Programmiersprache umgesetzt und die im Design entworfenen Elemente

- unter Berücksichtigung der Requirements und
- ggf. Codierrichtlinien und
- ggf. Codemetriken

implementiert und zu einer ablauffähigen Software zusammengesetzt.

In der Praxis überlappt sich die Codierphase oft mit ersten Testphasen (siehe Modultest, statische Codeanalyse)

3.5. Software Test

Der Software Test prüft Software systematisch auf das Vorhandensein von Fehlern. Dies geschieht meist

- statisch (d.h. ohne Ausführen der Software)
- dynamisch (d.h. Software wird ausgeführt)
- über mehrere Testphasen
- basierend auf Anforderungen (aus den Requirements) und auf dem Software Design

3.6. Projektmanagement

Zunächst soll der Begriff „Projekt“ definiert werden. Ein Projekt ist ein Vorhaben, das im Wesentlichen durch die Einmaligkeit der Bedingungen in ihrer Gesamtheit gekennzeichnet ist, z.B.

- Zielvorgabe,
- zeitliche, finanzielle, personelle und andere Begrenzungen
- Abgrenzung gegenüber anderen Vorhaben, insbesondere anderen Projekten
- projektspezifische Organisation.

(gemäß DIN 69 901)

Die Grenzen oder der Umfang eines Projekts sind definiert durch

- Seine zeitliche Abgrenzung
 - Zeitplan, Termine (insbesondere ein festgelegter Endtermin),
 - Projektentstehungsphase, Nachprojekt-Phase
- Seine sachliche Abgrenzung
 - Ziele, Aufgaben, Nicht-Ziele
 - zu anderen Projekten, Tätigkeiten
- Seine soziale Abgrenzung
 - Projektteam inkl. Projektleiter, Projektauftraggeber, organisiert in einem Projektteam
 - Umwelt

Projektmanagement ist unter anderen folgendermaßen in verschiedenen Normen und Gremien definiert:

- Projektmanagement ist die Gesamtheit von Führungsaufgaben, -organisation, -techniken und -mitteln für die Abwicklung eines Projektes (gemäß DIN 69901)
- Project Management is the application of knowledge, skills, tools and techniques to project activities to meet project requirements.
Project management includes
 - identifying requirements,
 - establishing clear objectives,
 - balancing the competing demands for time, quality, and cost,
 - adapting the specifications, plans, and approach to the different concerns and expectations of the various stakeholders.(Project Management Institute, pmi.org)

Projektmanagement umfasst alle Aufgaben bei der Durchführung von Projekten hinsichtlich

- Vorbereitung und Planung des Projekts bzgl. Struktur, Personal, Inhalte, Ziele
- Kontrolle und Lenkung des Projekts während der Projektlaufzeit
- Personalführung (zumindest fachlich) durch den Projektleiter
- Projektabschluss und Dokumentation der Ergebnisse während und am Ende des Projekts
- Prozessverbesserung als Nebenziel eines Projekts
- Interaktion mit dem Auftraggeber bzw. Kunden in Form von Dokumenten, Verträgen und regelmäßiger Kommunikation
- Koordination von Zulieferern, sofern Aufgaben im Projekt an Zulieferer vergeben werden

3.7. Software Qualitätssicherung

Durch eine systematisch durchgeführte Qualitätssicherung soll sichergestellt werden, das Endprodukt mit den Anforderungen übereinstimmt. Dies geschieht auf zwei sich ergänzende Weisen:

1. Sicherstellen, dass das richtige Produkt entwickelt wird!
2. Sicherstellen, dass das Produkt richtig entwickelt wird

Der erste Punkt soll sicherstellen, dass das Endprodukt dem Projektziel entspricht. Der zweite Punkt soll dafür sorgen, dass Vorgaben des Projekts, die das Endprodukt an sich nicht betreffen, jedoch auch maßgeblich zum Projekterfolg beitragen, z.B. Budget, Termine, Dokumentation oder gesetzliche und vertragliche Bedingungen, eingehalten werden.

Auch für dieses Thema gibt es standardisierte Definitionen:

Qualitätssicherung (QS) sind alle geplanten und systematischen Tätigkeiten, die innerhalb des Qualitätsmanagement-Systems verwirklicht sind, und die wie erforderlich dargelegt werden, um angemessenes Vertrauen zu schaffen, dass eine Einheit die Qualitätsanforderung erfüllen wird. (DIN EN ISO 8402)

Dabei wird der oben verwendete Begriff des Qualitätsmanagements(QM) definiert als alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortungen festlegen sowie diese durch Mittel wie Qualitätsplanung, Qualitätssicherung und Qualitätsverbesserung im Rahmen des Qualitätsmanagementsystems verwirklichen.
(gemäß DIN EN ISO 8402)

Zu beachten ist, dass der SW Test manchmal auch als Teil der Qualitätssicherung betrachtet wird. In dieser Vorlesung ist SW Test Teil des Engineering Zyklus und nicht der Software Qualitätssicherung.

Typische Tätigkeiten der Qualitätssicherung sind z.B.

- Reviews, d.h. visuelle Inspektionen, s. Kap. XXXX
- Messen von Software (Metriken)
- SW Test (hier als eigener Bestandteil der SW Engineering, s. Kap. XXXX)
- Formale Beweise

3.8. Software Konfigurationsmanagement

Zur Motivation des Konfigurationsmanagements seien folgende Projektsituationen betrachtet:

- Mehrere Entwickler arbeiten gleichzeitig an der Entwicklung einer Software
➔ wie wird der Zugriff auf gemeinsame Daten koordiniert?
- Änderungswünsche des Auftraggebers müssen während der Entwicklung berücksichtigt werden
➔ wie wird sichergestellt, dass Änderungen definiert umgesetzt werden?
- Fehler müssen behoben werden
➔ wie wird sichergestellt, dass Fehler definiert und sicher behoben werden?
- Eine Software wird gleichzeitig oder zeitlich versetzt in verschiedenen Versionen ausgeliefert
➔ wie wird sichergestellt, dass die richtigen Features in eine bestimmte Version integriert werden?
- Alte Versionen einer Software müssen weiterhin gepflegt werden
➔ wie wird sichergestellt, dass alte Versionen rekonstruierbar sind?
- Alle Arbeitsergebnisse einer bestimmten Version sollen gemeinsam verwaltet werden
➔ wie wird sichergestellt, dass Dokumente, Code, Tests, Manuale für diese Version konsistent sind?

Das Ziel der Konfigurationsmanagements ist es, definierte Lösungen für die oben geschilderten Situationen umzusetzen.

Gemäß IEEE 828-1990 ist Konfigurationsmanagement folgendermaßen definiert:

- Identification: identify, name, and describe the documented physical and functional characteristics of the code, specifications, design, and data elements to be controlled for the project.
- Control: request, evaluate, approve or disapprove, and implement changes
- Status accounting: record and report the status of project configuration items [initial approved version. status of requested changes, implementation status of approved changes]
- Audits and reviews: determine to what extent the actual configuration item reflects the required physical and functional characteristics

Die durch das Konfigurationsmanagement zu verwaltenden Elemente (Configuration Items) sind in der Regel alle Ergebnisse des Projekts, z.B.

- Spezifikationen (Anforderungen)
- Design (Architektur und Feindesign)
- Code (Sourcecode, Objectcode und ausführbare Programme)
- Testfälle (für alle Testphasen)

- Testergebnisse
- Änderungen
- Fehler

Des Weiteren gehören auch Planungsdokumente, Ergebnisse von Qualitätssicherung oder Besprechungsprotokolle zu den Konfigurationselementen.

Konfigurationsmanagement gliedert sich in die drei sich ergänzenden und voneinander abhängigen Bereiche

- Versionsmanagement,
- Änderungsmanagement und
- Buildmanagement,

die im Folgenden kurz vorgestellt werden.

Versionsmanagement

Versionsmanagement hat folgende Aufgaben:

- Verwalten aller Konfigurationselemente (Configuration Items) in einer geeigneten Ablage (z.B. Directory, Datenbank)
- Koordinieren des Zugriffs auf Konfigurationselemente bei parallelem Zugriff durch mehrere Entwickler
 - Ein-/Auschecken (nur wer ausgecheckt hat, darf schreibend zugreifen)
 - Der lesender Zugriff ist frei möglich für alle Entwickler
 - ggf. gleichzeitiger schreibender Zugriff durch Verzweigung (Branch), d.h. es entstehen mehrere parallele Versionen eines Elements. Diese müssen u.U. wieder zu einer einzigen Version zusammengefasst werden.
- Verwalten aller Versionen eines Konfigurationselements bzgl.
 - Historie (zeitlich nacheinander entstehende Versionen)
 - Ausprägungen (zeitlich parallel entstehende Varianten)

Änderungsmanagement

Änderungsmanagement hat folgende Aufgaben:

- Verwalten aller an der Software durchgeführten bzw. noch durchzuführenden Änderungen. Dabei wird eine Änderung initiiert durch
 - Fehlerbehebung nach Auftreten eines Fehlers, z.B: beim Test
 - Änderungswunsch des Auftraggebers gegenüber der ursprünglichen Aufgabenstellung
 - Änderungswunsch aus der Entwicklung innerhalb des Projekts

Build & Release Management

Build & Release Management hat folgende Aufgaben:

- Definiertes Zusammenbauen einer Software aus ihren Bestandteilen unter Berücksichtigung der richtigen Versionen. Das Ergebnis der Zusammenbaus ist eine Software (Release), die einem bestimmten Zweck dient (z.B. Freigabe an den Test, Freigabe an den Kunden)
- Einbinden der richtigen Änderungen in ein Release. Ziel ist es sicherzustellen, dass z.B. die geplanten Fehlerbehebungen und/oder Änderungen in die Software integriert werden.
- Erstellen einer Baseline, i.e. zu einem bestimmten Zeitpunkt Festhalten eines Zwischen- oder Endergebnisses, das alle (relevanten) Konfigurationselemente, inklusive ihrer Versionen enthält.
- zu einer Baseline gehören u.a. Sourcecode, lauffähige Software, Dokumentation, Testfälle, Testergebnisse in ihren jeweiligen für die Baseline geplanten Versionen.
- eine Baseline ist rekonstruierbar, d.h. bei Bedarf zu einem späteren Zeitpunkt aus ihren Bestandteilen wieder erstellbar.
- Freigeben der Baseline, wie oben erwähnt z.B.

- zum Testen
- an den Kunden

3.9. *Notationen*

Notationen dienen der (semi-) formale Beschreibung von Arbeitsergebnissen. Dies können sein z.B.

- Programmiersprachen, z.B. C, Java
- Skriptsprachen, z.B. Perl, AWK
- Modellierungssprachen, z.B. UML
- Testfallbeschreibungssprachen, z.B. Python

3.10. *Dokumente, Templates*

Dokumente halten Arbeitsergebnisse fest in geeigneter Form mit z.B. folgenden Inhalten

- Autor(en)
- Art des Dokuments
- Titel
- Version des Dokuments
- Version des zugrunde liegenden Formatvorlage (Template)
- Änderungshistorie
- Zustand z.B. Entwurf (Draft), gereviewt, freigegeben

Templates dienen als in einer Organisation standardisierte Vorlagen zur Erstellung von Dokumenten.

3.11. *Methoden*

Methoden beschreiben die Vorgehensweise bei der Durchführung von Tätigkeiten.

Methoden

- sind Bestandteil eines Entwicklungsprozesses einer Organisation
- definieren soweit möglich genau, wie etwas zu tun ist
- beschreiben einzelne (feingranulare) Arbeitsschritte
- müssen an die Domäne (z.B. Automotive, Medizin, Luftfahrt) angepasst werden
- müssen geschult werden

Anmerkung: In Kapitel 4 werden Software Engineering Techniken beschrieben. Aus diesen Techniken können konkrete Methoden abgeleitet werden.

3.12. *Tools*

Entwicklungswerkzeuge (Tools)

- sind unabdingbar in der Software Entwicklung
- werden für alle Tätigkeiten des SW Engineering benötigt
- als kommerzielle Tools und oft als Freeware verfügbar
- sollten auf Grund von Erfahrung und Evaluierung ausgewählt werden und nicht auf Grund von Prospekten, Präsentationen und Messen
- bewirken keine Wunder, insbesondere sind Versprechungen der Hersteller mit Vorsicht zu genießen
- müssen meist systematisch geschult und eingeführt werden, insbesondere bei komplexen Werkzeugen.

➔ Einfache Tools sind in der Praxis oft ausreichend, die Anschaffung teurer und komplexer Werkzeuge oft nicht notwendig

➔ A fool with a tool is still a fool

Anmerkung: Im Folgenden werden Tools erwähnt, detaillierte Beschreibungen oder Empfehlungen werden nicht gegeben. Die Auswahl der genannten Tools erhebt keinen Anspruch auf Vollständigkeit.

3.13. Rollen

Eine Rolle im Software Engineering beschreibt

- einen bestimmten Verantwortungsbereich einer (oder mehrerer) an der Entwicklung beteiligten Person(en)
- bestimmte Tätigkeiten, die im Rahmen der Rolle auszuführen sind
- bestimmte Ergebnisse der durchzuführenden Tätigkeiten

Dabei können Rollen „ähnlich“ wie in einem Film oder Theaterstück betrachtet werden. Eine Person nimmt eine bestimmte Rolle ein und füllt sie im Rahmen der Projekts und der Entwicklung aus.

In großen Organisationen existiert oftmals eine Vielzahl von Rollen, z.B.

- Kunde / Auftraggeber
- Firmenleitung
- Abteilungsleitung
- Vertrieb
- Marketing
- Anwender
- SW Requirements Ingenieur
- SW Architekt
- SW Tester
- SW Qualitätsingenieur
- (SW) Projekt Manager
- (SW) Entwickler
- (SW) Build Manager

Es existieren auch Rollen „am Rand“ eines Projekts, d.h. an der Schnittstelle des Projekts zu seiner Umwelt, z.B. der Kunde / Auftraggeber oder auch der spätere Anwender einer Software.

Rollen sind in der Regel im SW Prozess definiert. Rollen stehen teilweise in Beziehungen zueinander und in kleinen Projekten fallen ggf. mehrere Rollen auf eine Person zusammen. Rollen können auch in Konflikt zueinander stehen, d.h. die Interessen der Rollen sind gegenläufig. Z.B. wird ein Kunde möglichst viel für sein Geld haben wollen, während die Entwicklung möglichst kostengünstig entwickeln will um den Gewinn zu maximieren. Darüber hinaus dürfen bestimmte Rollen nicht in einer Person zusammenfallen. Als Beispiel sei hier die Qualitätssicherung genannt, die unabhängig von der Entwicklung sein muss. D.h. ein Entwickler darf nicht gleichzeitig für SW Qualitätssicherung verantwortlich sein.

4. SW Engineering Methodik

In diesem Kapitel werden für alle in 3 erwähnten Bereiche des SW Engineering Methodiken beschrieben.

4.1. Projektmanagement

Dieses Kapitel soll einen kurzen Überblick über Projektmanagement mit dem Fokus auf Software Projekte geben. Die Grundprinzipien können natürlich auch auf andere Projekte, Nicht-Softwareprojekte angewendet werden.

Definitionen

Definition Projekt

Zunächst soll der Begriff „Projekt“ definiert werden.

- Ein Projekt ist ein Vorhaben mit definiertem Anfang und Abschluss
- Es hat also eine zeitliche Befristung
- Ein Projekt beinhaltet meist etwas Neues, d.h. Neuartigkeit ist auch ein Charakteristikum eines Projekts
- Ein Projekt ist in seiner Form einmalig, auch wenn es in einer Organisation (Firma, Behörde) oftmals eine Vielzahl ähnlicher Projekte gibt.
- In der Regel beinhaltet ein Projekt auch ein gewisses Maß an Komplexität. Wäre das nicht der Fall, gäbe es das Projekt nicht.
- Nicht nur das Projekt an sich ist einmalig auch der Ablauf des Projekts findet nur ein Mal statt, auch wenn sich Abläufe innerhalb des Projekts ähnlich wiederholen
- Ein wesentliches Kennzeichen von Projekten ist die Beschränkung der Ressourcen. Dies erfolgt durch Randbedingungen wie begrenztes Budget und/oder begrenzte Anzahl von Mitarbeitern und technischen Ressourcen (Computer, Tools).

Definition Projektmanagement

Projektmanagement ist definiert als die Anwendung von Methoden, Know-how, Techniken zur Planung, Organisation Führung und Steuerung von Projekten. Zu den Elementen des Projektmanagement gehören Integrations-Management (d.h. Gesamt-Änderungs-Steuerung, Projektplan-Fortschreibung), Personal-Management (d.h. Personalbereitstellung, Team-Entwicklung), Risiko-Management (d.h. Risiko-Ermittlung und Risiko-Einschätzung), Kunden-Interaktion, Koordination von Zulieferern, Kosten-Management (d.h. Abschätzungen, Budgetierung), Zeit-Management (d.h. Aufgaben-Aufwandsschätzung, Feinplanung), Kommunikations-Management (d.h. Informationsverteilung, Berichtswesen), Qualitäts-Management (Qualitäts-Planung und -Kontrolle, Prozessverbesserungen)

Projekte haben oft widersprüchliche Ziele, die zu Konflikten beim Projektmanagement führen können, z.B.

- zweiwöchentliche Lieferungen von Software
- volle Prozesseinhaltung bei der Entwicklung (passt nicht zu kurzen Lieferzyklen)
- begrenztes und niedriges Budget
- zusätzliche Anforderungen (passt nicht zum Budget)
- mangelnde Ressourcen

Projektplanung



Das obige Bild zeigt zwei gegensätzliche und durchaus berechtigte Ansichten über das Thema Planung. Die eine Ansicht „Planung heißt, den Zufall durch den Irrtum zu ersetzen!“ sagt aus, dass man nicht alles planen kann bzw. muss, da ein Projekt normalerweise Änderungen unterliegt, die zu einer teilweisen Neuplanung führen. Außerdem ist es nicht sinnvoll Tätigkeiten extrem fein zu planen (z.B. auf Basis von Stunden), da schon geringe Verzögerungen die Planung aus dem Ruder laufen lassen. Andererseits besagt die zweite Aussage „Manchmal laufen Dinge nicht nach Plan, weil es nie einen Plan gab!“, dass es ohne Planung auch nicht geht. Insgesamt bedeutet das, dass für eine gute Planung ein geeigneter Kompromiss aus Genauigkeit, Feinheit, Detaillierung gefunden werden muss.

Was muss alles geplant werden? Die folgende Liste gibt eine Übersicht über Planungsgegenstände, die in der Regel im Rahmen der Projektplanung bearbeitet werden müssen.

- Arbeitspakete (Work Packages)
Ein Arbeitspaket ist eine Planungseinheit, die direkt ausgeführt werden kann und daher nicht weiter zerlegt wird. Die Beschreibung eines Arbeitspakets enthält (mindestens) die auszuführende Tätigkeit, den dafür notwendigen Aufwand und ggf. Abhängigkeiten zu anderen Arbeitspaketen. Arbeitspakete können auch Termine enthalten. Diese können direkt definiert werden oder aus Abhängigkeiten zu anderen Arbeitspaketen und zur Personaleinsatzplanung entstehen.
- Termine
Termine des Projekts müssen geplant werden. Dazu gehören Anfangs- und Endtermine und wichtige Meilensteine während der Projektlebenszeit.
- Kundentermine
Sofern ein Projekt auf Kundentermine angewiesen ist oder ein Ergebnis direkt an einen Kunden liefert, müssen auch entsprechende Kundentermine geplant werden. Dazu gehören Zwischenstände und Lieferung, ggf. auch der Start einer Produktion
- Kosten
Kosten sind detailliert zu planen bzgl. Personalkosten, Tools, externe Zulieferungen, Material usw.
- Personaleinsatz
Im Personaleinsatz ist festzulegen, wer macht was, d.h. welche Mitarbeiter an welchen Arbeitspaketen beteiligt sind. Dies führt zusammen mit der zeitlichen Planung der Arbeitspakete zur Planung der Auslastung der Mitarbeiter.

- Material
Der Bedarf an Material (z.B. CDs für eine SW-Auslieferung) muss ermittelt und bestellt werden.
- Kommunikation
Die regelmäßige Kommunikation innerhalb der Projekts und ggf. auch nach außen (z.B. Projektbesprechungen, Kundenkontakt, Berichterstattung) muss definiert und organisiert werden (z.B. Einladungen über elektronische, vernetzte Terminkalender, Besprechungszimmer, Telefonkonferenzen)
- Equipment
Ausrüstung, die für das Projekt benötigt wird muss geplant werden (z.B. Tools, Testfahrzeuge)
- Anwendung von (Entwicklungs-) Prozessen
Welche Entwicklungsprozesse sollen für das Projekt angewendet werden?
- Qualität
Welche Qualität soll das Projektergebnis haben?

Im Rahmen der Planung wird auch die Projekt- und Systemstruktur festgelegt. Ziel ist es hier, einen Grobüberblick über das Projekt zu bekommen.

Die folgenden Schritte werden dazu ausgeführt:

- Analyse der Inputs, d.h. analysieren der relevanten Rahmenbedingungen und Voraussetzung für Projektdurchführung
- Analyse der Outputs, d.h. untersuchen, welche Zielsetzung des Kunden im Hinblick auf Nutzung des Ergebnisses hat, Abgleich mit den Vorstellungen des Kunden
- Personelle Schnittstellen abklären und dokumentieren. Dies umfasst die Organisation innerhalb und außerhalb des Projektteams
- Funktionen definieren, d.h. Festlegung der Einzelfunktionen des Geräts, keine Arbeitspakete
- Externe technischen Schnittstellen abklären zur entwickelten HW, Mechanik, Fahrzeug, Produktion

Motivation und Ziele einer Planung sind das Einhalten der Kosten, ebenso das Einhalten der Termine, das Sicherstellen der gewünschten Qualität. Umgekehrt ausgedrückt das Verhindern eines Kostenüberlaufs, einer Terminverzugs, die Vermeidung mangelhafter Qualität. Des Weiteren ermöglicht eine richtig durchgeführte Planung die korrekte Abschätzung von Aufwenden bzw. Kosten von Änderungen. Nebenziel ist es auch die Planungsgenauigkeit ständig zu verbessern.

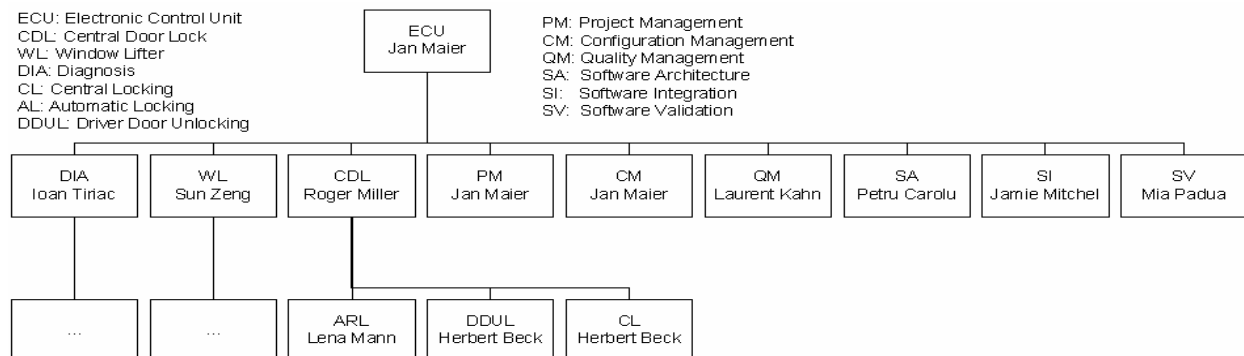
Erste Planungsaktivitäten, insbesondere Aufwandsabschätzungen werden bereits vor einer in der Angebotsphase eines Projekts vor Auftragsvergabe durchgeführt. Grobterminpläne sind dann bereits Bestandteil des Angebots an den Auftraggeber. Zum Start eines Projekts muss die gegebenenfalls vorhandene Planung bzgl. eventuell geänderter Randbedingungen überarbeitet und verfeinert werden.

Eine wesentliche Grundlage und Bestandteil der Planung ist die Work-Breakdown Struktur (Projektstrukturplan).

Ziel ist es hier das Gesamtprojekt aufzubrechen in gegebenenfalls in hierarchisch gegliederte Teilpakete, die am Ende fein genug sind für eine genaue Planung und Durchführung. Mögliche Vorgehensweisen bei der Konstruktion der Work_breakdown Struktur sind

- orientiert am Entwicklungszyklus (was muss alles in einem Zyklus durchgeführt werden) und / oder
- orientiert an den Funktionen des zu entwickelnden Systems basierend auf den Requirements und / oder
- orientiert an den verschiedenen Rollen der Projektteilnehmer („was muss ein Projektleiter tun?“, „was muss ein Codierer tun?“, „was muss ein Tester tun?“)

Das unten stehende Bild zeigt eine beispielhafte Work Breakdown Struktur. Jedes Kästchen steht für ein Arbeitspaket mit Nennung seines Verantwortlichen. Kästchen, an denen weitere Kästchen hängen, bezeichnen Arbeitsumfänge, die noch weiter strukturiert werden müssen, solange bis „atomare“ Planungseinheiten (Arbeitspakete) vorliegen.



Arbeitspakete

Die oben bereits erwähnten Arbeitspakete stellen die kleinsten Einheiten in der Work-Breakdown Struktur dar. Ein Arbeitspaket definiert eine geschlossene überschaubare Arbeitsmenge vom ca. 50 bis maximal 500 Arbeitsstunden. Jedes Arbeitspaket wird beschrieben durch

- seinen Umfang mit nachweisbarem Ergebnis
- Eingangsvoraussetzungen für den Start und die Durchführung des Arbeitspakets
- die zur Überwachung nötigen Arbeiten (Reporting), sofern dies nicht bereits auf Projektebene für alle Arbeitspakete definiert ist.
- Kostenvorgaben, die festlegen, wie teuer das Arbeitspaket sein darf (in der Regel ausgedrückt nicht durch einen Geldbetrag, sondern durch den Aufwand)
- einen Verantwortlichen für das Arbeitspaket.

Ein Arbeitspaket bezieht sich auf eine Projektphase, d.h. mit Beendigung der Projektphase muss auch das Arbeitspaket abgeschlossen sein und umgekehrt ist die Projektphase nicht abgeschlossen, solange das Arbeitspaket nicht abgeschlossen ist.

Die Summe aller Arbeitspakete ergibt alle zur Fertigstellung des Projekts notwendigen Arbeiten, das heißt alle (!) für das Projekt notwendigen Tätigkeiten werden im Rahmen von Arbeitspaketen geplant.

Für eine genaue zeitliche Planung des Projekts ist es noch notwendig, die Arbeitspakete in Abhängigkeiten zueinander zu setzen. Manche Arbeitspakete können erst begonnen werden, wenn andere beendet sind, manche müssen oder können überlappend ausgeführt werden, andere sind unabhängig voneinander.

Eine wichtige Kenngröße eines Arbeitspakets ist sein Aufwand. Da der reale Aufwand für ein Arbeitspaket vor seiner Beendigung nicht bekannt ist, muss für die Planung auf einen geschätzten Wert zurückgegriffen werden.

Schätzungen können „aus den Bauch heraus“ oder systematisch durchgeführt werden¹.

Schätzmethoden

Voraussetzung für die Durchführung von Aufwandsschätzungen ist die Zerlegung des Projekts in seine Arbeitspakete. Einige Schätzmethoden seien hier kurz vorgestellt.

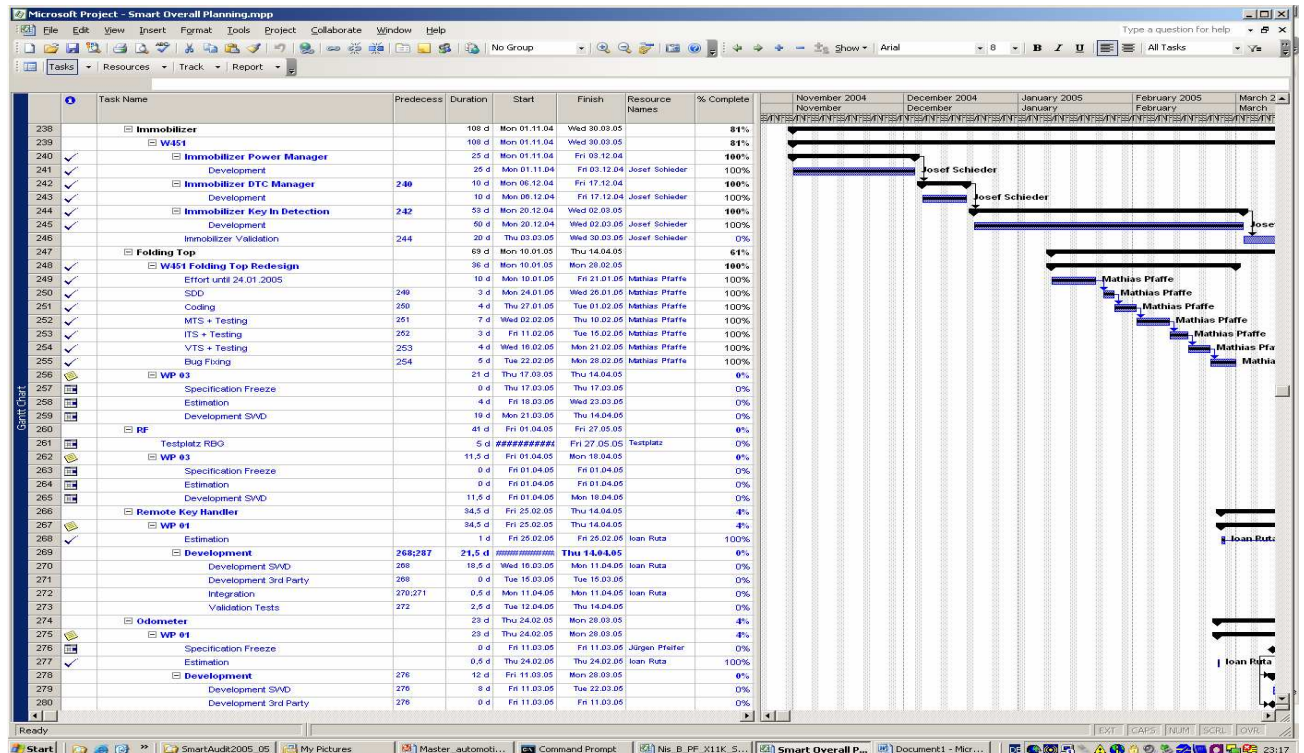
¹ Schätzungen „aus dem Bauch heraus“ müssen nicht immer ungenau und damit schlecht sein. Voraussetzung für eine gute „Aus-dem-Bauch-heraus-Schätzung“ ist eine fundierte Erfahrung.

- 4-Augen-Methode
Die Schätzung eines Arbeitspaketes wird von einer Person durchgeführt. Zur Plausibilisierung wird von einem zweiten Erfahrungsträger eine unabhängige Vergleichsschätzung durchgeführt. Der Projektleiter überprüft die Schätzungen. Alle zur Schätzung zugrunde liegenden Annahmen werden dokumentiert.
- Prozent-Methode
Basierend auf Vorgängerprojekt wird jedes Arbeitspaket als prozentualer Anteil am Gesamtprojekt abgeschätzt. Historische Daten dienen als Basis
- Schätzklausur mit Expertenrunden
Diese Methode ist die bei weitem aufwendigste aber auch genaueste. Mindestens 3 Experten schätzen hier in einem festgelegten Ablauf die Arbeitspakete unabhängig voneinander. Weichen Schätzwerte signifikant voneinander ab, werden die Abweichungen im Schätzerteam ausdiskutiert. Eine Schätzklausur und ihre Vorbereitung hat folgenden Ablauf:
 1. Die zu schätzenden Arbeitspakete werden definiert entweder durch Moderator der Schätzklausur oder den Projektleiter. Dabei sollte die Größe eines Arbeitspaketes im oben erwähnten Bereich von 50 bis 500 Stunden liegen (je kleiner die Arbeitspakete, desto genauer aber auch aufwendiger die Schätzung).
 2. Die Beschreibung der Arbeitspakete wird an die Schätzer verteilt.
 3. Jeder schätzt alle Arbeitspakete für sich ab. Der dafür notwendige Zeitaufwand kann erheblich sein, da jeder Schätzer die Inhalte der Arbeitspakete für eine exakte Schätzung genau kennen muss.
 4. Alle Schätzer kommen zu einer Besprechung (Schätzklausur) zusammen. Nacheinander werden die Arbeitspakete durchgesprochen. Jeder Schätzer gibt seine Schätzwerte an. Falls die Schätzwerte eines Arbeitspakets alle in der gleichen Größenordnung (max. Abweichung z.B. 50%) liegen, wird (ohne weitere Diskussion) der Mittelwert der Schätzungen gebildet (oder evtl. auch der Maximalwert). Weichen die Schätzwerte signifikant voneinander ab, wird diskutiert, welche Annahmen und Gründe zu den abweichenden Schätzwerten geführt haben, und (möglichst) eine Einigung über den gemeinsam definierten Wert gebildet.
 5. Alle Resultate der Schätzklausur werden dokumentiert einschließlich der Annahmen, die zu den Schätzungen geführt haben.

Terminplan

Der Terminplan ist die Synthese aus der Work-Breakdown Struktur, der Schätzung der Aufwende der Arbeitspakete unter Berücksichtigung der Abhängigkeiten der Arbeitspakete untereinander. Der Terminplan zeigt also alle Arbeitspakete mit ihren Abhängigkeiten. Zusätzlich enthält der Terminplan die für jedes Arbeitspaket definierten Anfangs- und Endtermine und nennt gegebenenfalls auch die Bearbeiter des Arbeitspakets. Wird zur Dokumentation des Terminplans ein Tool eingesetzt (meist "Microsoft-Project") können die Termine durch das Tool basierend auf Projektanfang und definierten Zwischenmeilensteinen automatisch errechnet werden. Aus dem ersten Entwurf des Terminplans sind auch Personalauslastung und gegebenenfalls kritische Pfade (terminliche Abläufe, bei denen das Ende vor oder gleichzeitig mit dem Beginn liegt) ersichtlich. Übersteigt die Personalauslastung bei einem oder mehreren Mitarbeitern eine bestimmte Grenze (z.B. 80% der Arbeitszeit) oder zeigt der Plan kritische Pfade, so ist die Planung (und damit auch das Projekt) entsprechend zu korrigieren, z.B. durch Verlegen von Terminen, zusätzlichen Mitarbeitern oder Änderungen von Inhalten. Im Verlauf des Projekts wird der Terminplan ergänzt durch Informationen, wie weit Arbeitspakete fertig gestellt sind. Bei Änderungen im Projekt wird der Terminplan entsprechend überarbeitet.

Das Bild unten zeigt exemplarisch einen realen Terminplan eines Automotive Projekts erstellt mit Microsoft Project (links textuell die Arbeitspakete mit Terminen und Aufwenden, rechts grafisch dargestellt die Abhängigkeiten und Meilensteine).



Steuerung und Kontrolle

Planung ist im Projektverlauf keine Tätigkeit, die nur einmal zu Projektbeginn durchgeführt werden muss, sondern die in der gesamten Projektlaufzeit immer wieder erledigt werden muss. Grund hierfür sind die schon oben erwähnten Änderungen, die in jedem Projekt vorkommen können. Änderungen werden verursacht, z.B. durch geänderte Kundenwünsche, geänderte Kundentermine, Änderungen einer Technologie, personelle Änderungen im Projekt, Budgetänderungen und insbesondere dadurch, dass Tätigkeiten nicht so durchgeführt werden können, wie sie geplant waren. Deshalb muss die Planung im Projektverlauf ständig gegen den aktuellen Stand des Projekts überprüft und gegebenenfalls überarbeitet werden. Bei der Überprüfung der Planung werden auch die bereits vollständig oder teilweise erledigten Arbeitspakete entsprechend gekennzeichnet. Diese ständige Planverfolgung wird als Steuerung und Kontrolle bezeichnet. Der entsprechende englische Begriff "Controlling" bedeutet also nicht allein Kontrolle, sondern auch das Reagieren auf Abweichungen bis hin zu einer Neuplanung.

Die wichtigsten Voraussetzungen für eine erfolgreiche Projektsteuerung ist das genaue Dokumentieren der Planung und das Ziel, die vorhandene Planung auch in die Tat umzusetzen. Mit dem Abarbeiten der geplanten Arbeitspakete setzt die Planverfolgung (Tracking) ein. Planverfolgung bedeutet wie schon oben erwähnt, das ständige Abgleichen des Ist-Standes mit dem geplanten Stand. Alles was geplant wurde, muss auch verfolgt werden (z.B. Arbeitspakete, Termine, Kosten, Personaleinsatz). Die Planverfolgung wird ebenso wie die Planung dokumentiert. Weicht die Realität von der Planung ab, wird die Abweichung² analysiert und bewertet bezüglich ihrer Auswirkungen auf das Projekt oder auf Teile davon. Falls eine Abweichung den Projekterfolg oder wichtige Projektziele beeinträchtigt, gefährdet oder gar unmöglich macht, müssen Gegenmaßnahmen definiert und geplant werden. Können diese Gegenmaßnahmen nicht mehr im Rahmen des geplanten Projektumfangs durchgeführt werden, z.B. zusätzliches Budget wird dafür benötigt, zusätzliche Mitarbeiter werden benötigt oder wichtige Termine müssen geändert werden, wird dazu eine Managemententscheidung (Eskalation) eingeleitet. Wichtig ist dabei, dass eine Entscheidungsvorlage gegebenfalls

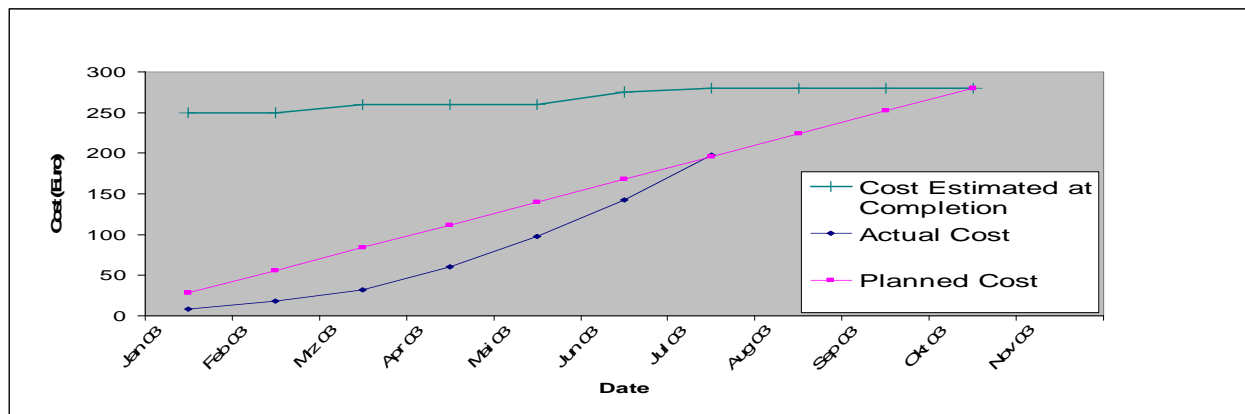
² Abweichungen können auch „positiv“ sein, z.B. schnelleres Fertigstellen eines Arbeitspakets oder Kosteneinsparung

mit Alternativen und Konsequenzen aus dem Projekt erarbeitet wird, auf dessen Grundlage das Management entscheiden kann.

Zusätzlich zur Planverfolgung im Projekt selbst, wird es häufig eine regelmäßige Berichterstattung über das Projekt (Reporting) meist in komprimierter Form an das Management und oder an den Kunden geben.

Kostenkontrolle

Ein besonderer Aspekt der Projektverfolgung ist die Kontrolle der Kosten. Wie bei Terminen müssen hier Abweichungen zwischen geplanten und tatsächlichen Kosten identifiziert werden. Bisweilen müssen Daten, die von Kostenerfassungssystemen (z.B: SAP) geliefert werden mit Vorsicht interpretiert werden, da der Kostenaufbau in einem Projekt verzögert erfolgen kann. Das Bild unten zeigt einen exemplarischen Kostenverlauf mit den zu bestimmten Zeitpunkten geplanten, den aktuellen und den für die gesamte Entwicklung angesetzten Kosten.



Risikomanagement

Jedes Projekt enthält Risiken, die den Projekterfolg und die Projektziele potenziell bedrohen können. Ein Risiko ist also eine konkrete Bedrohung des Projekts, die noch nicht zu einem Problem geworden ist. Im Rahmen des Risikomanagements werden zunächst Risiken identifiziert. Anschließend werden die Risiken bewertet bezüglich ihrer Eintrittswahrscheinlichkeit und ihrer Auswirkungen und Konsequenzen auf das Projekt. Diejenigen Risiken, die eine hohe Eintrittswahrscheinlichkeit haben und einen hohen Schaden anrichten können, werden mit entsprechenden Maßnahmen behandelt. Diese Maßnahmen zielen zum einen darauf, die Eintrittswahrscheinlichkeit des Risikos zu vermindern, zum anderen Aktionen zu definieren, die die Auswirkungen bei Eintritt des Risikos³ vermindern. Die unten stehende Grafik zeigt die Einstufung von Risiken: Risiken mit hoher Eintrittswahrscheinlichkeit und hohem Schadenspotential (rote Risiken) müssen auf jeden Fall behandelt werden. Risiken mit mittlerer Eintrittswahrscheinlichkeit oder mittlerem Schadenspotential (gelbe Risiken) sind individuell zu behandeln. Risiken mit geringer Eintrittswahrscheinlichkeit und geringen Folgekosten können unter Umständen ignoriert werden (grüne Risiken).

Risikomanagement beginnt mit der ersten Planungsphase und zieht sich im Rahmen der Projektverfolgung durch den gesamten Projektverlauf. Das heißt, dass identifizierte Risiken regelmäßig überprüft werden müssen, ebenso wie gegebenenfalls neue Risiken zu betrachten sind.

³ Ein eingetretenes Risiko ist ein Problem.

Probability	High (>75%)			Red Risk
	Medium (25%...75%)		Yellow	
	Low (<25%)	Green Risk		
		1 Low	2 Medium	3 High
		Severity (Impact)		

SW –Development Plan / Projekthandbuch

Die Ergebnisse aller Projektmanagementaktivitäten müssen geeignet dokumentiert werden. Als übergeordnetes Planungsdokument dient hier das Projekthandbuch, auch als SW-Development Plan bezeichnet. Dieses Dokument soll einen sehr kurzen Überblick mit den wichtigsten Eckdaten über das Projekt geben und alle „konstanten“ Projektmanagementinformationen enthalten. „Konstant“ sind die Informationen, die im Projektverlauf über längere Zeit stabil bleiben. Darüber hinaus kann das Projekthandbuch als internes "Vertragswerk" zwischen Organisation, SW-Projektleiter und SW-Entwicklern verstanden werden. Deshalb ist es wichtig, dass alle Projektbeteiligten mit dem Inhalt vertraut sind.

Typischerweise hat das Projekthandbuch folgenden Inhalt:

- Eine kurze Beschreibung des zu entwickelnden Systems. Diese sollte nicht nur auf die SW beschränkt sein, sofern das SW Projekt Teil eines übergeordneten Projekts, z.B. ein Elektroniksystem im Fahrzeug ist.
- Eine kurze Beschreibung des Projektumfangs, die erklärt, was entwickelt werden soll
- Festlegen der Verantwortlichkeiten innerhalb des Projektteam und auch übergeordnet, z.B. Kunde, Management
- Die wichtigsten Meilensteine des Projekts
- Die Projektziele
- Welche Arbeitsmittel werden zur Durchführung des Projekts benötigt (z.B. Testplätze, PCs)
- Welche Software wird gegebenenfalls von außerhalb des Projekts zugeliefert.
- Eine Beschreibung der (erlaubten) Abweichungen vom vorgegebenen Entwicklungsprozess im Rahmen des so genannten Tailorings
- Die Projektorganisation in Bezug auf Meetings, Dokumentablage (Konfigurationsmanagement)
- Die Eskalation, d.h. die Art und Weise, wie Probleme, die nicht innerhalb des Projekts gelöst werden können, an das Management und ggf. den Kunden weitergeleitet werden.

Zusammenarbeit über verschiedene Standorte

Eine besondere Herausforderung beim Projektmanagement ist die Zusammenarbeit über mehrere Standorte hinweg. Dabei sind verschiedene Schnittstellen zu beachten. Zum einen können funktionale Schwerpunkte nur an einzelnen Standorten verfügbar sein (technische Spezialisten). Zum anderen wird die Kommunikation größtenteils nur über E-mail, Telefon und Netmeetings ablaufen. Ein gewisser Informationsverlust ist bei dieser Art der Kommunikation, die häufig nicht in der Muttersprache der jeweiligen Partner stattfindet, zu berücksichtigen. Ein gemeinsames Kick-Off-Meeting mit dem kompletten Projektteam zum gegenseitigen Kennen lernen ist extrem hilfreich⁴. Regelmäßige Treffen (z.B. alle 4 Wochen) sind wünschenswert aber sind in der Praxis aus Kostengründen meist nicht möglich. Wegen der

⁴ Bei Kick-Off Veranstaltungen ist der informelle Teil, z.B. gemeinsam ein Bier trinken, nicht zu unterschätzen!

=> Festlegung der Arbeitspakete extrem wichtig

Abnahme der Produktentwicklung

- Systemtest hauptsächlich zur funktionellen Abnahme des Produkts durch Kunden
- Verschiedene Testebenen, z.B.
 - HW mit Simulationen von Kommunikationsbussen und Lasten
 - Fahrzeug-Simulation im Labor mit realen Steuergeräten
 - komplettes Fahrzeug
- Abgrenzung zwischen Komponentenverantwortung und Systemverantwortung wird oft vermischt
- Systematische Integration der Komponenten wird oft abgekürzt durch direktes Einbauen ins Gesamtsystem (z.B. Fahrzeug) mit nicht-kompatiblen Schnittstellen

=> Zeitaufwand muss mit eingeplant werden

=> Abgrenzung der Verantwortung wichtig

Änderungsmanagement

- Anforderungen ändern sich über die Projektlaufzeit
- Teil der Anforderungen ist zu Projektstart noch nicht definiert
- Detailliertes Angebot ermöglicht Ausweisung des Zusatzaufwandes und dessen Einforderung beim Kunden
- Kontrolliertes Einspeisen der Anforderungen zum Beginn eines V-Zyklus notwendig
- Auswirkung auf Requirements Engineering und Validierung
- Anpassung von Termin-, Kosten- und Ressourcen-Planung notwendig

3rd Party SW

- Ursachen für Notwendigkeit zum Einbau von 3rd Party SW, z.B.:
 - Automobilhersteller fordert einheitliche Module über gesamtes Fahrzeug (z.B. CAN, Diagnose, OSEK, Bootloader für Reflash)
 - Entwicklung einzelner Funktionalitäten beim Hersteller oder anderem Know-how-Träger (z.B. neue Funktionalitäten wie Reifendruck, Kryptologie)
- Formen der Zulieferung
 - Source Code
 - Object Code (+ Header File(s))
 - Modelle in Modellierungssprache zur direkten Codegenerierung (z.B. Statemate, Matlab, ASCET-SD, Rapsody)

3rd Party SW: Probleme

- Architekturinkompatibilitäten
 - statisch: z.B. Namenskonflikte, verschiedene Maßeinheit von Variablen
 - dynamisch: z.B. unterschiedliche Modulinteraktion wie Event versus Polling
- unterschiedliche funktionelle Aufteilung
- unterschiedliche Design-Konventionen
 - z.B. Präemptives und nicht präemptives Scheduling
- Entwicklungsumgebung
 - z.B. Compilerabhängigkeit
- Qualitätsprobleme: Verantwortung im Fehlerfall, beschränkte Gewährleistung

3rd Party SW: Lösungsansatz

- Integrationsdokument
 - Detaillierte SW Interface Beschreibung der 3rd-Party SW (functions, global variables, interrupts, data types, timings, access mechanisms)
 - detaillierte Beschreibung der benötigten, d. h. von einem selbst zur Verfügung zu stellenden Interfaces und deren Benutzung
 - Ressourcenverbrauch (RAM incl. Stack, ROM, EEPROM, µC) (wichtig für Embedded Systeme)

- Umweltbedingungen und Seiteneffekte (task, interrupt, μ C operation modes, μ C registers, interrupts en-/disabled)
- Konfiguration
- Entwicklungsumgebung (z. B. Compiler/Linker settings)
- Beschreibung des Integrationstests
- gemeinsame Integrationstests mit Integrations-Review
- rechtlich bindende Vereinbarung von Gewährleistung und Haftung

4.2. Requirements Engineering

Requirements oder Anforderungen beschreiben in der Regel, das **WAS** eine Software leisten soll und nicht das **WIE**. Das Wie ist Bestandteil der SW Architektur und des Feindesigns.

Beschreibung von Requirements

Requirements können in natürlicher Sprache beschrieben werden. Beispiele:

- Req. 4.1.2 Strommessung
Für die vom Batteriesensor an den Batteriemonitor übergebenen Stromwerte werden drei verschiedene Messbereiche festgelegt, die den Größenordnungen der Batterieströme bei Fahrzeug-Stillstand (Messbereich 1), im Fahrtzustand (Messbereich 2) und bei Motorstart (Messbereich 3) entsprechen.
- Req. 5.1.7 Programmiersprache
- Durch den Auftraggeber wird die Verwendung der Programmiersprache ANSI C gefordert. Wenn die Programmiersprache ANSI-C nicht eingesetzt werden kann, muss dies durch den Auftragnehmer begründet und durch den Auftraggeber genehmigt werden.

Requirements in allen Phasen der Software Entwicklung

Ausgangspunkt für Requirements sind die Anforderungen, die im Lastenheft dokumentiert sind. Das Lastenheft (oder Customer Specification) beschreibt das Produkt aus Kundensicht ohne, wie schon erwähnt, die Lösung vorweg zu nehmen. Das Lastenheft ist die Basis für Angebotserstellung und Auftragsvergabe.

Aus dem Lastenheft wird das Pflichtenheft (Product Specification, Software Specification) erstellt. Das Pflichtenheft beschreibt das Produkt aus Sicht der Entwicklung erneut ohne die Lösung vorweg zu nehmen. Das Pflichtenheft verfeinert das Lastenheft und fügt zusätzliche, z.B. entwicklungsinterne Anforderungen hinzu.

Weitere Requirements entstehen im Lauf eines Projekts während SW Designaktivitäten (SW Architektur und SW Feindesign). In der SW Architektur werden Anforderungen an SW Komponenten definiert, im Feindesign an Funktionen und Module.

Funktionale und Nicht-Funktionale Requirements

Grundsätzlich wird zwischen Funktionalen und Nicht-Funktionalen Requirements unterschieden. Funktionale Requirements sind Requirements, die das Verhalten der SW nach außen beschreiben. Nicht-Funktionale Requirements sind Requirements, die nicht die Außenwirkung der Software betreffen

Beispiele für Funktionale Requirements:

- Wenn der Schalter betätigt wird, muss das Licht angehen
- Es muss die Fakultät der eingelesenen Zahl berechnet werden
- Bei Auswahl des Menüs sollen folgende Untermenüs angezeigt werden: ...
- Der vom Sensor gelieferte Spannungswert muss über die Leitung x eingelesen werden

Beispiele für Nicht-Funktionale Requirements:

- Anforderungen an die Entwicklungsumgebung
 - Das Produkt muss in der Sprache C++ mit dem Compiler XY entwickelt werden
- Anforderungen an die Zielumgebung
 - Das Produkt muss auf den Plattformen Windows XP, Windows Vista und Linux lauffähig sein
- Anforderungen an organisatorische Randbedingungen, z.B.
 - Das Produkt muss gemäß dem V-Modell entwickelt werden“
 - Das Produkt muss am 19. Januar fertig sein
 - Das Budget von 100000€ muss eingehalten werden.

Es gibt allerdings Requirements bei denen die Zuordnung zu Funktional bzw. Nicht-Funktional nicht ganz eindeutig möglich ist, z.B.

- Anforderungen an Performance / Ressourcenverbrauch
 - Das System muss innerhalb von 10ms reagieren
 - Die Funktion darf nicht mehr als 500 Byte RAM belegen
- Anforderungen an funktionale Sicherheit
 - Das Produkt muss dem Safety Integrity Level 3 (SIL3)⁵ genügen

Eine Reaktionszeit von 10ms ist an sich keine Funktion, wird aber in der Regel an eine bestimmte Funktion gebunden sein. Ebenso ist der RAM Verbrauch nicht an eine Funktion gebunden. Andererseits kann ein höherer RAM Verbrauch dazu führen, dass die Software nicht mehr ausgeführt werden kann. SIL3 ist an sich keine Funktionale Anforderung, kann aber bedeuten, dass zusätzliche Funktionen in die Software (z.B. Redundanz) implementiert werden müssen, um SIL3 zu erfüllen.

Achtung: Nicht-funktionale Requirements haben oft einen wesentlichen Einfluss auf Entwurfsentscheidungen!

Bemerkung: Die Begriffe „Funktionale Anforderungen“ und "Nicht-funktionale Anforderungen" sind nicht eindeutig und daher „unglücklich“.

Achtung: Nicht alle Requirements werden explizit dokumentiert, da als selbstverständlich vorausgesetzt. Beispielsweise ist „Fehlerfreiheit“ meist ein implizit definiertes Requirements. Obwohl es im Allgemeinen nicht erfüllt werden kann, wird es erwartet.

Stakeholder

Ein wesentlicher Begriff des Requirements Engineering ist der sog. Stakeholder. Ein Stakeholder ist eine Person oder Institution, die ein Interesse an dem zu entwickelnden Produkt hat, also in Bezug auf die Requirements ein Interessenvertreter. D.h. von einem Stakeholder kommen in der Regel Requirements.

Typische Stakeholder in einem Projekt können sein:

- Kunde / Auftraggeber
- Nutzer
- Management
- Vertrieb /Marketing
- Entwickler
- Fertigung (bei eingebetteten Systemen)
- Qualitätssicherung
- Externe Institutionen (z.B. Autoversicherungen bei Diebstahlsicherheit)
- Gesetzgeber (z.B. Sicherheitskritische Software)

⁵ Ein Safety Integrity Level legt fest, ob und wie sicherheitskritisch ein System bzw. eine Software ist. Ein bestimmtes SIL Level hat in der Regel Anforderungen an den Entwicklungsprozess und an das Produkt selbst zur Folge.

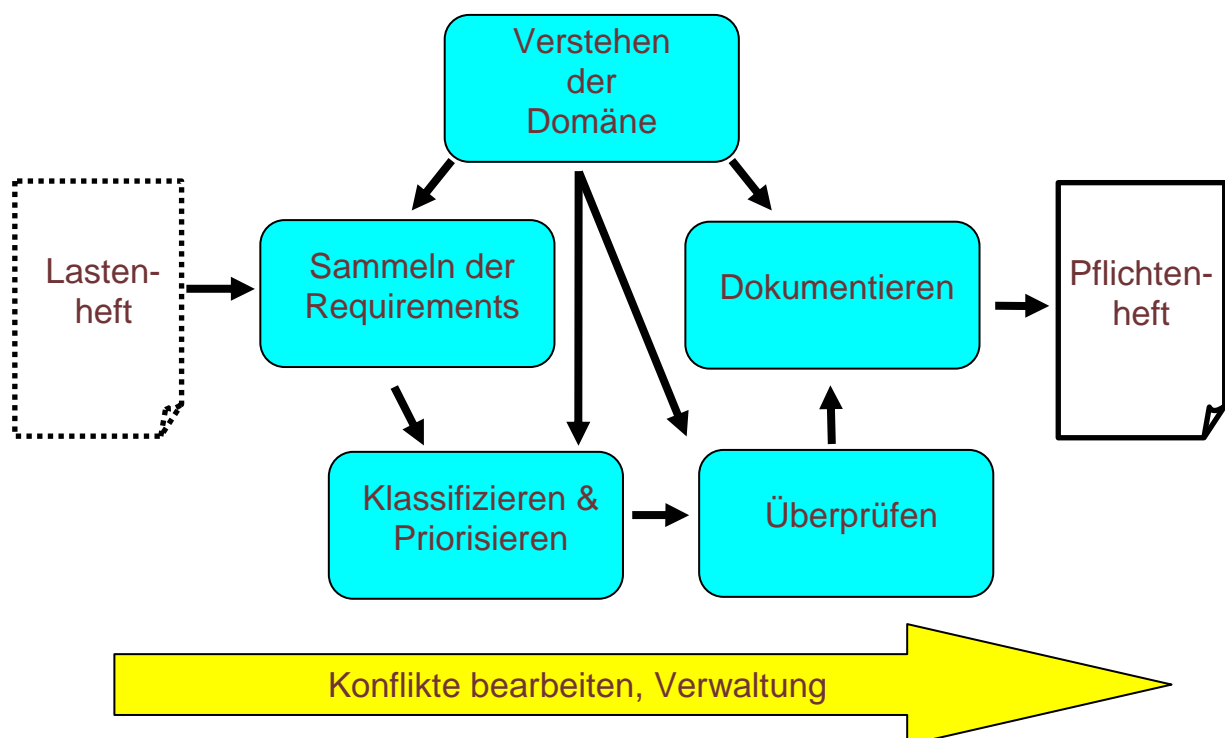
Achtung: Die Interessen verschiedener Stakeholder werden nicht immer identisch, zum Teil auch widersprüchlich sein. Dies kann dann durchaus auch zu widersprüchlichen Requirements führen, z.B.

- Der Kunde möchte möglichst wenig für das Produkt bezahlen, der Vertrieb möchte möglichst viel einnehmen.
- Der Kunde möchte das Produkt möglichst schnell. Die Qualitätssicherung möchte das Produkt in möglichst hoher Qualität

Zur Erfassung aller relevanten Requirements müssen daher alle möglichen Stakeholder identifiziert (und dokumentiert) werden. Zu beachten ist weiterhin, dass Stakeholder Interesse in verschiedenen Phasen der Entwicklung haben können. So wird eine Vertriebsabteilung zu Beginn des Projekts stärker involviert sein, als bei der Implementierung

Methodik

Das folgende Bild gibt einen Überblick über die Vorgehensweise beim Requirements Engineering beginnend mit dem Lastenheft bis zur Erstellung eines gültigen Pflichtenhefts. Dabei sollen die Abläufe in Pfeilrichtung nicht streng sequentiell sondern teilweise überlappend, parallel oder auch wiederholend nacheinander ausgeführt werden.



Verstehen der Domäne

Ein wesentlicher Punkt für das erfolgreiche Erstellen von Requirements ist das Verstehen der Domäne, für die das Produkt entwickelt wird. Mit „Domäne“ ist hier der Einsatz- und Anwendungsbereich der zu entwickelnden Software (z.B. Automotive, Medizin, Luftfahrt, Büro, Telekommunikation) gemeint.

- erleichtert bzw. ermöglicht das Verstehen der Anforderungen aus dem Lastenheft
- erleichtert die Kommunikation mit dem Auftraggeber
- erleichtert das weitere Bearbeiten der Anforderungen und das Umsetzen in ein Pflichtenheft
- erleichtert das Erstellen eines Angebots an den Auftraggeber

Sammlung der Requirements (Elicitation)

Das Sammeln der Requirements besteht aus folgenden Tätigkeiten:

- Übernahme (Kopieren) der Anforderungen aus dem Lastenheft (sofern ein Lastenheft vorhanden ist)
- Brainstorming⁶ (mit allen Stakeholdern) zur Ermittlung von Requirements
- Interviewtechnik (mit allen Stakeholdern unter Verwendung eines vordefinierten Fragenkatalogs)
- Ermittlung von Anwendungsfällen (Use Cases). Diese spielen typische Anwendungen des Gesamtprodukts beispielsweise durch.
- Übernahme der Anforderungen aus einem Vorgängerprojekt (sofern vorhanden)
- Auswerten von Rückmeldungen der Anwender eines existierenden Produkts (Kunden- / Nutzerbefragung)
- Auswerten von Testergebnissen
- Marktanalyse (Analyse von Konkurrenzprodukten)

Das Sammeln von Requirements kann je nach Projekt oder Produkt mit einzelnen, mehreren oder in allen diesen Tätigkeiten ausgeführt werden. Die Ergebnisse werden in einer ersten Version des Pflichtenhefts dokumentiert.

Klassifizieren der Requirements (Analysis)

Nach der Sammlung der Requirements werden diese analysiert bezüglich folgender Kriterien:

- Abhängigkeit zwischen Requirements, z.B.
 - welche bedingen sich?
Requirements, die sich bedingen, müssen gemeinsam umgesetzt werden, dies muss entsprechend geplant werden.
 - welche schließen sich gegenseitig aus?
Schließen sich Requirements gegenseitig aus, können nicht alle gleichzeitig umgesetzt werden, es muss eine Auswahl getroffen werden
- Zusammengehörigkeit von Requirements, z.B.
 - welche gehören zusammen?⁷ Welche machen nur gemeinsam Sinn? Auch hier sind mehrere Requirements gemeinsam umzusetzen.
 - welche lassen sich nur gemeinsam realisieren?
- Klassifizierung nach Rollen / Stakeholderbezug
 - zu welcher Rolle gehört bzw. von welchem Stakeholder kommt welches Requirement. Dies ermöglicht bei der weiteren Bearbeitung eine direkte Kommunikation mit dem Stakeholder, stellt sicher, dass alle Stakeholder berücksichtigt worden sind und unterstützt die Priorisierung der Requirements (s.unten)

Beispiele rollenbezogener Klassifizierung

Die Beispiele zeigen, dass verschiedenen Stakeholder durchaus völlig unterschiedliche Requirements haben können.

1.) Anforderungen an den Bierausschank eines Restaurants

⁶ Brainstorming Technik: Alle Teilnehmer äußern ihre Ideen und Vorschläge. Diese werden zunächst nicht diskutiert, nur erfasst und erst später besprochen und bewertet.

⁷ Requirements, die zusammengehören müssen sich nicht in jedem Fall bedingen. Bei der Zentralverriegelung in einem Fahrzeug gehören die Funktionen Verriegeln und Entriegeln natürlich zusammen. Bedingen tun sie sich allerdings nicht, denn rein theoretisch (!) könnte auch nur eine Entriegelfunktion und kein Verriegeln implementiert werden. Anders würde es aussehen, wenn eine Funktion Warnblinken gefordert ist, aber keine Funktion eine Lampe blinken zu lassen.

Requirements des Endkunden (Gast), z.B.

„Das Bier muss richtig temperiert sein“

Requirements des Lieferanten, z.B.
„Der Zugang zum Restaurant muss breit genug für Bierfässer sein“



Requirements des Wirts, z.B.
„Bier muss in ausreichender Menge vorhanden sein“

Requirements des Gesetzgebers, z.B.
„Aussschank von Bier an Personen unter 16 Jahren ist nicht gestattet“

2.) Anforderungen an die Lichtsteuerung in einem Kraftfahrzeug

Requirements des Endkunden (Autofahrer), z.B.
„Bei Betätigung des Lichtschalters muss das Fahrlicht angehen“

Requirements des Lieferanten, z.B.
„die Software der Lichtsteuerung muss in der Fertigung flashbar sein“



Requirements des Automobilherstellers, z.B.
„der Fehlerspeicher der Lichtsteuerung muss in der Werkstatt auslesbar sein“

Requirements des Gesetzgebers, z.B.
„die Stromversorgung des Fahrlichts muss redundant ausgelegt sein“
„Der Ausfall einer Stromversorgung muss erkannt und dem Fahrer per Warnlampe angezeigt werden“

Priorisieren der Requirements (Analysis)

Parallel oder nach der Klassifizierung werden Requirements priorisiert in Bezug auf folgende Kriterien:

- Bedeutung des Requirements
 - Muss-Requirements, diese Anforderungen sind unverzichtbar und müssen auf jeden fall umgesetzt werden.
 - Soll-Requirements, diese Anforderungen sind wünschenswert, allerdings nur unter vertretbaren Randbedingungen bzgl. Kosten und Termin
 - Kann-Requirements, diese Anforderungen sind „nett“, aber nicht essentiell für den Erfolg des Projekts oder Produkts
- Realisierung, d.h. welche Requirements sollen zuerst implementiert werden, welche später (sofern schon bekannt)
- Kosten, d.h. welche Requirements verursachen welche Kosten (sofern schon bekannt)

Im Bierausschank Beispiel ist das Requirement der Temperatur des Biers mit Sicherheit ein Muss Requirement. Wird dieses Requirement nicht erfüllt, wird der Verkauf des Bieres schnell zu Ende sein. Das Requirement bzgl. der Menge ist vielleicht ein Soll Requirement. Ist der Platz im Lokal zu gering kann der Wirt vielleicht auf ein nahe stehendes Gebäude auswei-

chen und dort seinen Biervorrat lagern. Das Requirement der Zugangsbreite ist möglicherweise ein Kann_Requirement, da das Bier auch aus Flaschen ausgeschenkt werden kann. Über die Bewertung der Altersbeschränkung gibt es möglicherweise unterschiedliche Ansichten.

Überprüfen der Requirements (Validation & Verification)

Hier geht es darum sicherzustellen, dass zum einen die richtigen Requirements spezifiziert sind und zum anderen dass die Requirements richtig spezifiziert sind. Beide Punkte sind wichtig, die richtigen Requirements definieren das richtige Produkt. Die richtige Art der Requirementsspezifikation stellt sicher, dass die nachfolgenden Entwicklungstätigkeiten richtig ausgeführt werden können.

Im Einzelnen wird geprüft, ob die Requirements folgende Eigenschaften besitzen

- Adäquat, d.h. der Aufgaben- und Problemstellung angemessen? Nicht adäquate Anforderungen sind entweder „unterspezifiziert“, d.h. spezifizieren „zu wenig“ oder „überdefiniert“ (s. unten), d.h. spezifizieren zu viel.
- Vollständig, d.h. es fehlen keine Anforderungen
- überdefiniert, d.h. gibt es überflüssige Anforderungen, die keinen sinnvollen Beitrag leisten.
- Verständlich, d.h. so formuliert, dass alle Beteiligten das Richtige und Gleiche darunter verstehen
- Eindeutig, d.h. es gibt keinen Interpretationsspielraum
- Konsistent d.h. widerspruchsfrei
- Testbar, d.h. kann für jede Anforderung ein entsprechender Testfall gefunden werden. Ist das nicht der Fall, kann die Umsetzung des Requirments nicht durch einen Test sichergestellt werden.
- Prüfbar, ähnlich zu testbar, d.h. gibt es eine Möglichkeit zu prüfen, ob das Requirement umgesetzt wurde.

Die Überprüfung der Requirements kann mit folgenden Methoden durchgeführt werden:

- Reviews s. Kap. XXXX
- Simulation
 - formal beschriebene Requirements mittels entsprechender Tools (z.B. Statemate, Matlab)
 - „gedanklich“, d.h. ein oder mehrere Requirement werden mittels Tafel, Flipchart, Whiteborad o.ä. durchgespielt
- Prototyping (s. unten)
- Testen
 - lässt sich zu jedem Requirement (mindestens) ein Testfall definieren?

Dokumentation

Folgende Inhalte sollten in einer Requirement Dokumentation enthalten sein:

- Funktion der Software, z.B. wie werden Eingabedaten bearbeitet, wie ist die Struktur der Daten
- Leistung der Software, z.B. Datendurchsatz, Reaktionszeit
- Kommunikation, z.B. Bedienoberfläche zum Anwender, Bussysteme, Schnittstellen zu anderer Software oder zur Hardware
- Fehlerfälle, z.B. wie reagiert die SW auf fehlerhafte bzw. unplausible Eingaben oder fehlerhafte Umgebungen
- Qualitätsaspekte
- Sonstige Randbedingungen

Die Dokumentation erfolgt meist in natürlicher Sprache. Dies hat Vor- und Nachteile.

Vorteile der natürlichsprachlichen Beschreibung:

- in der Regel verständlich für Auftraggeber und Auftragnehmer, sofern beide die gleiche Sprache sprechen
- eignet sich gut, einen Überblick über die zu entwickelnde Software zu geben

- ausdrucksstark

Nachteile der natürlichsprachlichen Beschreibung:

- erlaubt meist Interpretationen (Mehrdeutigkeiten sind nicht ausschließbar)
- nicht maschinell auf Konsistenz oder Vollständigkeit überprüfbar
- unübersichtlich, falls als alleiniges Beschreibungsmittel verwendet

Um die Nachteile zu minimieren, sollte (bzw. muss) natürliche Sprache immer in geeigneter Weise verwendet werden um Mehrdeutigkeiten zu reduzieren:

- eine geeignete Strukturierung ist zu wählen
- geeignete Formulierungen sind zu verwenden
- ein Glossar (Lexikon der verwendeten Begriffe) vermeidet Fehlinterpretationen
- Abkürzungsverzeichnis, falls Abkürzungen verwendet werden (auch bekannte Abkürzungen immer erläutern)

Des Weiteren müssen textuelle Beschreibungen soweit möglich durch weitere Beschreibungsmethoden ergänzt werden, z.B. Diagramme, Tabellen, Bilder und formale Beschreibungen.

Tipps zur richtigen textuellen Beschreibung von Requirements:

Die Formulierung sollte, wenn möglich folgendes Schema einhalten:

[Bedingung]	[Aktor]	[Aktion]	[Objekt]
Wenn ...	Fahrer ...	verarbeiten ...	die Eingabe ...
Bevor ...	das System ...	senden ...	das Signal ...
Nachdem ...	Sensor ...	empfangen ...	die Information ...
Bis ...	Komponente ...	schalten ...	den Zustand ...
Falls ...	Funktion ...	ausgeben ...	den Benutzer...
Solange ...		einlesen ...	
		wechseln	
		berechnen	

Beispiel Reifendruckkontrolle:

Während der Fahrt muss das System einen sicherheitsrelevanten Druckverlust in Reifen durch Einschalten der Warnlampe anzeigen

Dabei sollte möglichst wenig (oder kein) Interpretationsspielraum offen bleiben.

- „das System muss möglichst schnell reagieren“ → ungenau, was bedeutet „möglichst schnell“?

oder

- „das System muss in höchstens 10ms wie beschrieben reagieren“ → genaue Beschreibung der Reaktionszeit

- „das System muss möglichst alle Daten zu den Versicherten ausdrucken“ → ungenau, was bedeutet „alle Daten“?

oder

- „das System muss die Daten Name, Vorname, Geburtsdatum und Adresse zu den Versicherten ausdrucken“ → genaue Beschreibung der auszudruckenden Daten.

Des Weiteren sollten Requirements möglichst „atomar“ definiert werden, d.h. ein Requirement in einem oder ggf. mehreren Sätzen. Eine solche Beschreibung ist übersichtlicher und verbessert die Zuordnung von Testfällen zu Requirements, wie in den folgenden Beispielen erläutert:

- „die Alarmanlage muss im Fall eines Einbruchs das Blinklicht und die Sirene einschalten und über Telefonleitung die Polizei benachrichtigen und den Wachmann über Handy alarmieren“ → mehrere Requirements in einem Satz.

oder

- Req. Alarm 1: die Alarmanlage muss im Fall eines Einbruchs das Blinklicht einschalten
- Req. Alarm 2: die Alarmanlage muss im Fall eines Einbruchs die Sirene einschalten
- Req. Alarm 3: die Alarmanlage muss im Fall eines Einbruchs Telefonleitung die Polizei benachrichtigen
- Req. Alarm 4: die Alarmanlage muss im Fall eines Einbruchs den Wachmann über Handy alarmieren → jedes Requirement in einem Satz.

Weitere Anforderungen an gute Requirementsbeschreibungen sind

- Eindeutig identifizierbar, d.h. wie im obigen Beispiel wird jedem Requirement ein eindeutiger Name zugeordnet. Dies ermöglicht eine spätere Referenzierung der Requirements in Reviews oder Testfällen.
- Vermeidung von Mehrdeutigkeiten
- Verbesserte Nachverfolgbarkeit (Traceability⁸)
- aber: Sicherstellen, dass der Gesamtzusammenhang verständlich bleibt, z.B. durch Einleitungskapitel, Kommentare. Dies Verbessert maßgeblich die Lesbarkeit und Verständlichkeit von Requirementsdokumenten.
- Präzise beschreiben, d.h.
 - Substantive anstatt „man“, „es“
 - Definierte Bedeutung von „muss“, „soll“, „kann“, „darf“ Siehe dazu auch oben den Abschnitt Priorisieren von Requirements. Eine Definition dieser Begriffe ist insbesondere wichtig bei fremdsprachlichen Beschreibungen. So muss z.B. in englischsprachigen Dokumenten die Bedeutung der Worte „must“, „has to“, „shall“, „should“, „can“, „may“ usw. in der Verwendung mit Requirements definiert werden.
 - Quantifizierte Aussagen (z.B. exakte Mengen- und Zeitangaben anstatt „viel“, „wenig“, „schnell“ usw.
 - Angemessene Detaillierung (je mehr Details, desto mehr Aufwand bei der Beschreibung)

Die Gliederungsstruktur von Requirementsdokumenten für ein Projekt sollte in einer Organisation oder wenigstens für das Projekt definiert sein. Es gibt dafür auch Standards wie das folgende Beispiel „Software Requirements Specification“ angelehnt an IEEE 830 (Quelle Wikipedia) zeigt:

- Name des Softwareprodukts
- Name des Herstellers
- Versionsdatum des Dokuments und / oder der Software
- 1. Einleitung
 - 1. Zweck (des Dokuments)
 - 2. Ziel (des Softwareprodukts)
 - 3. Verweise auf sonstige Ressourcen oder Quellen
 - 4. Erläuterungen zu Begriffen und / oder Abkürzungen

⁸ Der Begriff „Traceability“ bedeutet „Nachverfolgbarkeit. Traceability stellt einen Bezug zwischen der verschiedenen Ergebnissen der Software Entwicklung her. Im Detail wird das weiter unten noch erläutert.

5. Übersicht (Wie ist das Dokument aufgebaut?)
2. Allgemeine Beschreibung (des Softwareprodukts)
 1. Produkt Perspektive (zu anderen Softwareprodukten)
 2. Produktfunktionen (eine Zusammenfassung und Übersicht)
 3. Benutzermerkmale (Informationen zu erwarteten Nutzern, z.B. Bildung, Erfahrung, Sachkenntnis)
 4. Einschränkungen (für den Entwickler)
 5. Annahmen und Abhängigkeiten (nicht Realisierbares und auf spätere Versionen verschobene Eigenschaften)
3. Spezifizierte Anforderungen (im Gegensatz zu 2.)
 1. funktionale Anforderungen (Stark abhängig von der Art des Softwareprodukts)
 2. nicht-funktionale Anforderungen
 3. externe Schnittstellen
 4. Design Constraints
 5. Anforderungen an Performance
 6. Qualitätsanforderungen
 7. Sonstige Anforderungen

Modellierung von Requirements

Neben der natürlichsprachlichen Beschreibung von Requirements besteht die Alternative oder Ergänzung durch Modellierung. Dabei wird ein Modell erstellt, das das oder die zu beschreibenden Requirements in abstrakter Weise erfüllt. Ein Modell kann eine grafische informelle Darstellung sein, eine in einer formalen Modellierungssprache erstellte Beschreibung, die soweit detailliert sein kann, dass aus dem Modell der Code direkt generiert werden kann.

Das Ziel der Modellierung ist es, Anforderungen nicht (nur) als eine Sammlung von Sätzen in natürlicher Sprache dokumentieren, sondern ein anwendungsnahes Modell der Aufgabenstellung zu erstellen. Analog zu sprachlichen Beschreibung soll keine Lösung vorweggenommen werden.

Wie schon erwähnt, wird ein Modell unter Verwendung formaler oder teilformaler grafischer Beschreibungen erstellt. Dabei steht die Darstellung funktionaler Requirements im Vordergrund.

Modelliert werden

- Statische Struktur, z.B. Daten, Klassen, Objekte
- Interaktion, z.B. Szenarien, Kommunikation, Kollaboration
- Verhalten, z.B. Zustände

des Problembereichs, nicht der Lösung

Die Modellierung von Requirements bietet Vorteile gegenüber anderen Beschreibungsverfahren:

- + problemnahe Beschreibung, insbesondere falls Domänenspezifische Modellierungssprachen verwendet werden.
- + eindeutig, formal, sofern eine formale Modellierungssprache mit definierter Syntax und Semantik verwendet wird.
- + formal überprüfbar, falls eine formale Sprache verwendet wird
- + teilweise simulierbar, z.B. Zustandsautomaten, Pseudocode
- + „sanfter“ Übergang von Requirements zum Entwurf (also zur Lösung)
- + teilweise Code aus Modellen generierbar

Zu beachten sind aber auch einige Nachteile der Modellierung von Requirements:

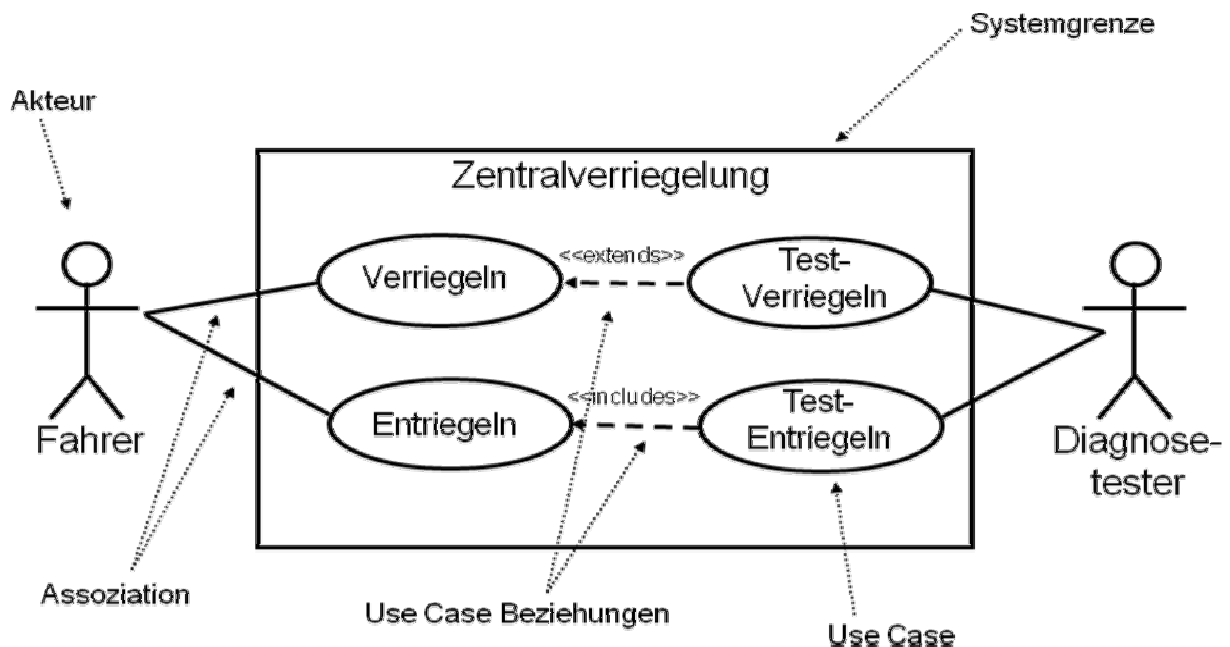
- erfordert detaillierte Kenntnisse der verwendeten Sprache und Tools
- erfordert detaillierte Erfahrung mit Modellierung
- Tools sind (teilweise) sehr teuer
- weckt zu hohe Erwartungen, die sich nicht erfüllen lassen.

Beispiele für gängige Modellierungssprachen / Tools sind

- Use Cases UML (s. Kap. XXX)
- Matlab für mess-/regelungstechnische Systeme
- Statemate für zustandsbasierte Systeme

Exkurs: UML Use Cases

Use Cases in UML bieten eine Möglichkeit der Modellierung von Requirements. Sie dienen zur Modellierung von funktionalen Anforderungen (Was soll mein System bzw. meine SW leisten?) und beschreiben das System aus Sicht der Nutzer (Akteure) in typischen, häufig vorkommenden Anwendungsfällen. Dabei muss der Akteur nicht unbedingt eine Person, sondern kann auch ein anderes System darstellen. Die Systemgrenze grenzt das zu entwickelnde System ein. Assoziationen verbinden Akteure mit Use Cases, d.h. ein Akteur spielt bei dem verbundenen Use Case eine Rolle. Use Cases können in Beziehung zueinander stehen. Ein Use Case kann einen anderen beinhalten (include) oder erweitern (extend). Ein Use Case besteht normalerweise aus dem unten gezeigten Use Case Diagramm, das ergänzt wird durch weitere Informationen wie textuelle Beschreibung, Sequenz Diagramme, Zustandsautomaten (s. a. Kap. XXX)



Verwaltung von Requirements (Requirements Management)

Anforderungen müssen verwaltet werden bzgl. Änderung (modification, change management), Nachverfolgbarkeit (traceability) und Freigabe (baselining, release management)

Die Verwaltung von Änderungen ist notwendig, da Requirements Änderungen unterliegen, z.B. durch den Auftraggeber, durch den technischen Fortschritt oder entwicklungsinterne Änderungswünsche. Dabei gilt, dass je später ein Requirement geändert wird, desto teurer ist die Umsetzung. Im Extremfall kann die Änderung von Requirements die komplette Neuentwicklung notwendig machen, falls z.B. durch die Änderung eine neue Architektur eines Systems definiert werden muss. Einerseits sollten Requirements daher für eine „ungestörte“ Entwicklung möglichst stabil sein, andererseits müssen Änderungen kontrolliert zugelassen werden.

Die Vorgehensweise bei Änderungen ist in einem Änderungsprozesses (Change Management) für Requirements zu definieren, der auch eine Versionierung von Requirements unterstützt. Zur Methodik dazu s. Kap. XXXX

Die Freigabe (baselining, release management) von Requirements schreibt die Menge der in einem Entwicklungszyklus umzusetzenden Requirements fest, die dann in einer in einer Baseline der kompletten Konfiguration festgehalten wird. S. dazu Kap. XXXX

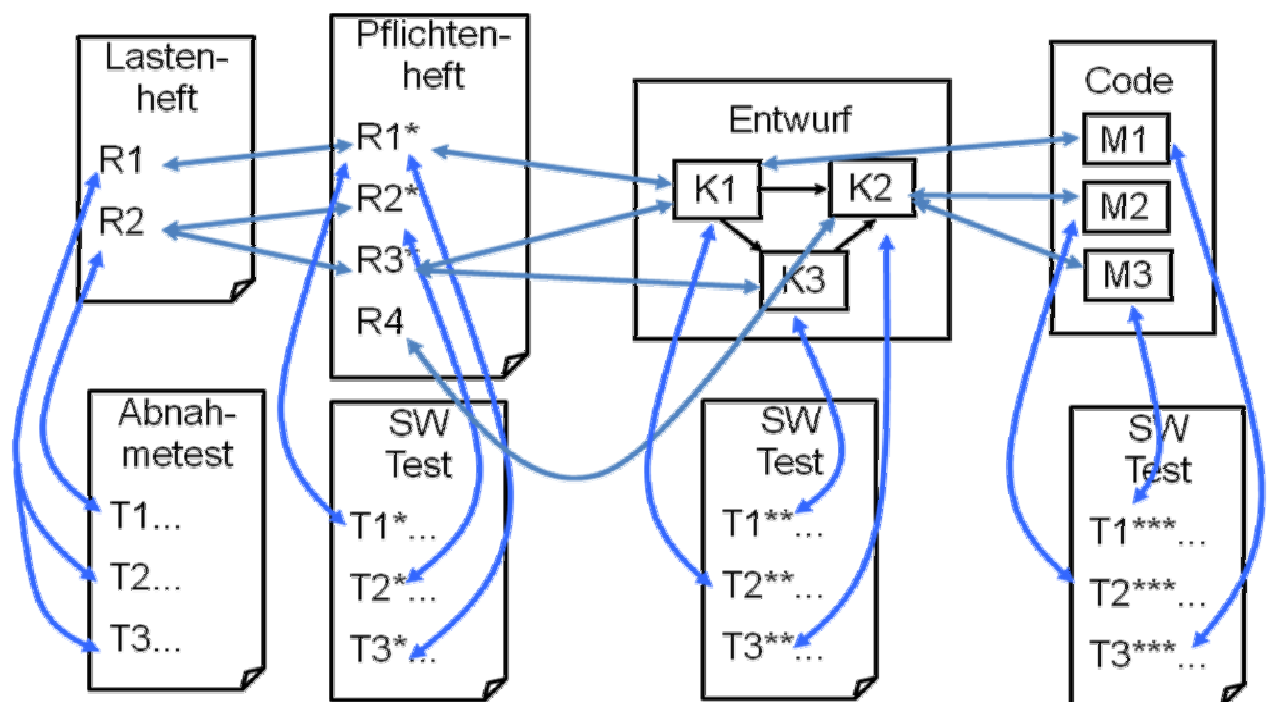
Die Nachverfolgbarkeit von Requirements (Traceability) soll sicherstellen, dass alle zu implementierenden Requirements auch implementiert wurden bzw. kein Requirement vergessen wurde, dass alle umgesetzten Requirements getestet werden und Auswirkungen von Änderungen in den Requirements verfolgt werden können.

Die Verfolgung von Requirements (Traceability) beantwortet die folgenden Fragen:

- Welche Beziehung besteht zwischen Requirements? (z.B. Lastenheft zu Pflichtenheft)
- Wo (in der Software) ist welches Requirement umgesetzt? (z.B: welche Komponente in der Software Architektur erfüllt welche Requirements)
- Durch welche Testfälle wird welches Requirement getestet? (ggf. auch durch wie viele Testfälle wird ein Requirement getestet)
- Welche Teile der Software sind von der Änderung eines Requirements betroffen?

Dabei soll eine Traceability in beiden Richtungen, d.h. von den Requirements zu Implementierung und zu den Testfällen und umgekehrt vorhanden sein.

Das folgende Bild zeigt eine „vollständige Traceability zwischen allen Entwicklungsschritten und Ergebnissen.



In der Praxis kann der Grad der erforderlichen Traceability von Projekt zu Projekt oder innerhalb verschiedener Entwicklungsstadien eines Projekts durchaus variieren

- Prototypen werden ggf. ohne jede Traceability entwickelt, Traceability spielt hier keine Rolle
- SW, die wieder verwendet werden soll oder SW, die sich häufig ändert, erfordert einen höheren Grad an Traceability
- Normen (z.B. IEC 61508 für sicherheitskritische Software) können einen sehr hohen Grad erfordern.

Zu beachten ist, dass der Aufwand zur Pflege der Traceability sehr hoch werden kann und dass „Extreme“ Traceability wie „In welcher Zeile des Codes ist das Requirement X umgesetzt?“ keinen Sinn machen!

Die Umsetzung der Traceability muss also den projektspezifischen Gegebenheiten angepasst werden, z.B:

- durch geeignete Tools, z.B. Links in DOORS. Dies kann allerdings u.U. problematisch sein, falls verschiedene Tools verwendet werden müssen, die nicht „zusammenpassen“, d.h. die keine Möglichkeit zum Austausch von Traceabilityinformation bieten.
- Traceability Tabellen können in vielen Fällen ausreichend sein.
- Kapitelweise Traceability kann innerhalb von Dokumenten ausreichen. In diesem Fall werden textuelle Hinweise auf Kapitel innerhalb des Dokuments oder in anderen Dokumenten eingefügt.

Requirements Engineering Tools

In der Praxis werden Tools zur Verwaltung und Dokumentation von Requirements notwendig sein. Für „einfache“ Anwendungen können die Office-Tools Word oder Excel durchaus verwendet werden. Für komplexe Anwendungen reichen diese Tools nicht mehr aus bzw. führen zu zusätzlichem Aufwand. Daher sind datenbank-basierte Tools mit komfortable Sortier- und Suchmöglichkeiten, Verknüpfung von Requirements (Traceability) untereinander und mit anderen Artefakten, mit Versionierung und Bearbeitungszustand notwendig. Wiki-basierte Tools sind vielleicht der zukünftige Trend.

Die folgende Liste gibt eine Übersicht über Requirements Engineering Tools ohne Anspruch auf Vollständigkeit und ohne Wertung:

- Caliber-RM (Borland)
- CARE (Sophist Group)
- DOORS (Telelogic)
- in-Step (microTOOL)
- RequisitePro (IBM / Rational)

4.3. Software Entwurf

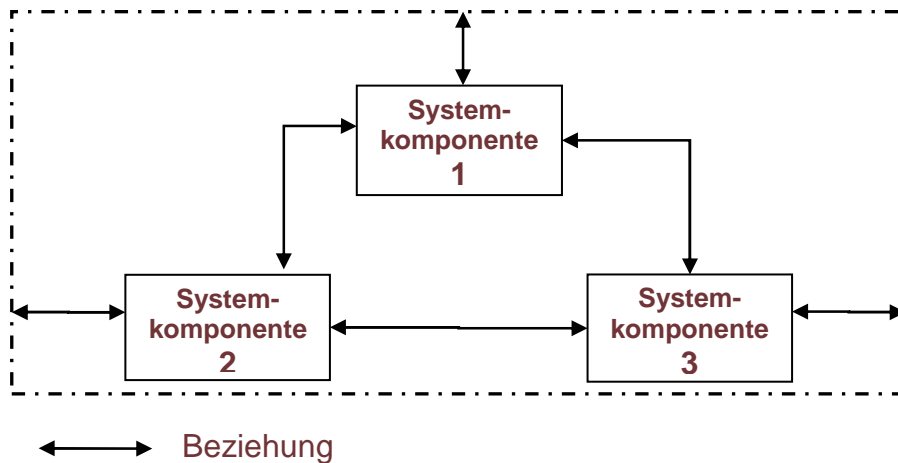
Aus dem Ziel des Software-Entwurfs, die Struktur der zu entwickelnden Software zu definieren, ergeben sich die Aufgaben der Entwurfsphase:

Entwerfen einer Software-Architektur, d.h. der Grobstruktur oder Struktur auf höherer Ebene. Diese geschieht durch

- Zerlegung des definierten Systems in Systemkomponenten, die unabhängig voneinander weiter strukturiert werden können.
- Strukturierung des Systems durch geeignete Anordnung der Systemkomponenten zueinander
- Beschreibung der Beziehungen zwischen den Systemkomponenten, d.h. welche Rolle spielt welche Komponente
- Festlegung der Schnittstellen, über die die Systemkomponenten miteinander kommunizieren
- Spezifikation des Funktions- und Leistungsumfanges der Systemkomponenten, d.h. die Requirements an die Komponente
- Detaillierung der Systemkomponenten durch Beschreibung ihrer Operationen

Software Architektur

Die Software-Architektur beschreibt also die Struktur des Software-Systems durch Systemkomponenten und ihre Beziehungen untereinander.



Eine Systemkomponente für sich ist ein abgegrenzter Bereich eines Software-Systems. Sie dient als Baustein für die physikalische und logische Struktur einer Anwendung. Beispiele für Systemkomponenten sind Funktionen/Prozeduren, abstrakte Datentypen oder Klassen.

Ziel des Software-Grobentwurfs ist es, für das zu entwerfende Produkt eine Software-Architektur zu erstellen, die die funktionalen und nicht-funktionalen Produkthanforderungen sowie allgemeine und produktspezifische Qualitätsanforderungen erfüllt und die Schnittstellen zur Umgebung versorgt.

Dabei werden verschiedene Prinzipien angewendet:

- Das Prinzip der Zerlegung, d.h. die Aufteilung des Systems in miteinander interagierende Bausteine. Um ein System übersichtlicher und verständlicher zu machen, kann es in Teilsysteme unterteilt werden, die über Relationen verbunden sind, die miteinander kommunizieren können, und somit in ihrer Gesamtheit das Gesamtsystem widerspiegeln
- Prinzip der Abstraktion⁹, d.h. Identifikation eines Teilsystems mit der funktionalen Rolle, die es im Gesamtsystem zu übernehmen hat, zunächst ohne Berücksichtigung seiner Implementierungsdetails. Das Teilsystem wird zunächst als Schwarze Kiste (Black Box) betrachtet, deren Implementierungsdetails im ersten Schritt nicht relevant sind bzw. nicht bekannt sein müssen.



Da viele zu modellierende Systeme sehr komplex und somit zunächst sehr unverständlich sind, verwendet man das Prinzip der Abstraktion dazu, um eine möglichst vereinfachte Sicht des Systems zu gewinnen.

Bei der Strukturierung eines System können verschiedene Verfahren verwendet werden: Der Top-Down- und der Bottom-Up-Entwurf

Die Top-Down Vorgehensweise ist gekennzeichnet durch

- einen schrittweise immer weiter gehende Spezialisierung, d.h. vom Gesamten zum Teil
- eine schrittweise Verfeinerung

⁹ Abstraktion ist das Gegenteil von Konkretisierung

- einer schrittweisen Zerlegung des Gesamtsystems in Teilsysteme.

Umgekehrt dazu besteht die Bottom-Up Vorgehensweise aus

- einer schrittweisen Generalisierung
- einer schrittweisen Verallgemeinerung
- einer Konstruktion des Gesamtsystems schrittweise aus Teilsystemen.

Die Top-Down Vorgehensweise wird angewendet bei komplett neuen Systemen, die Bottom-Up Vorgehensweise bei Systemen, die auf bereits bestehenden Teilen zusammengesetzt werden (Lego Prinzip, das System wird aus bestehenden Bausteinen schrittweise zusammengesetzt). Beide Vorgehensweisen sind miteinander kombinierbar und werden in der Praxis oft kombiniert, da Systeme meist zumindest zum Teil auf existierender Software aufsetzen (z.B. Betriebssysteme, Kommunikationssysteme, HW Treiber).

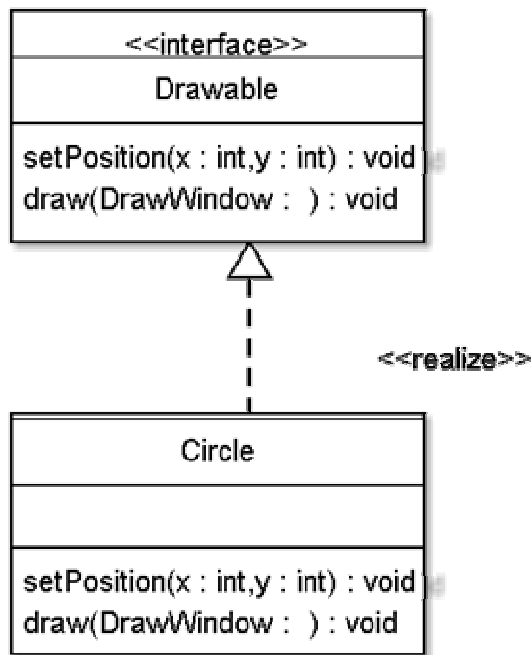
Als Ergebnis des Grobentwurfsprozesses erhält man einen Produkt-Entwurf, der aus folgenden Teilen besteht:

- Software-Architektur, d.h. die strukturierte oder hierarchische Anordnung der Systemkomponenten und ihre Beziehungen untereinander.
- Spezifikation jeder Systemkomponente, d.h. Festlegung von Schnittstellen, Funktions- und Leistungsumfang im Sinne einer Lasten oder Pflichtenheftes der Komponente. Dabei ist die Schnittstelle einer Komponente eine Spezifikation für ihr externes Verhalten, also das Verhalten, das Aufrufer bzw. Nutzer der Schnittstelle erwarten können. Die Schnittstelle spezifiziert insbesondere eine Menge von Operationen, die an der Schnittstelle verfügbar sind, gibt jedoch für diese Operationen keine Implementierung an (Abstraktion). Die Spezifikation einer Schnittstelle kann oder sollte daher mindestens enthalten:
 - die Liste von Operationen, die die Schnittstelle zur Verfügung stellt,
 - Zusicherungen an die Komponente. Zusicherungen sind logische Ausdrücke, die Eigenschaften der Komponente widerspiegeln. Diese Eigenschaften können immer gelten, unabhängig von Aufrufen der Komponente. In diesem Fall spricht man von einer Invarianten. Ist eine Eigenschaft zu erfüllen, bevor eine Operation ausgeführt werden darf, handelt es sich um eine Vorbedingung. Gilt eine Eigenschaft nach einer Operation, ist es eine Nachbedingung.Das Prinzip der Zusicherungen sei hier nochmals an der Datenstruktur stack (Stapel) erläutert. Wird auf einen Stapel ein Element angefügt (operation push(element)), gilt, dass der Stapel nicht leer ist (Nachbedingung), Wird von einem nichtleeren (Vorbedingung) Stapel ein Element heruntergenommen (Operation pop()), so gilt, dass sich die Höhe des Stapels um 1 vermindert hat (Nachbedingung) Hat der Stapel eine fest definierte maximale Größe, wäre dies eine Invariante.

Exkurs: Schnittstellen Darstellung in UML

In der UML-Notation wird eine Schnittstelle für Klassen mit dem Klassensymbol dargestellt, das zur Unterscheidung von einer gewöhnlichen Klasse mit dem Stereotyp <<Interface>> gekennzeichnet ist. Die Schnittstelle wird als Rechtecke dargestellt mit

- dem Schnittstellen-Namen
- der Deklaration der Schnittstellenoperationen
- der annotierten Bedeutung der Operationen
- evtl. mittels OCL annotierten Zusicherungen
- Eine Komponente, die die Schnittstelle realisiert, wird durch eine (gestrichelte) Realisierungskante mit der Schnittstelle verbunden



Klassische Architekturmodelle

In diesem Abschnitt werden die folgenden klassischen, d.h. weit verbreiteten Architekturmodelle vorgestellt:

- Blockdiagramm, sehr allgemeine Darstellung, geeignet als allererster Ansatz, um eine Architektur zu beschreiben.
- Datenspeichermodell (= Datenbank-Schema), geeignet speziell für den Einsatz globaler Datenbanken
- Schichtenmodell, hierarchische Anordnung zunehmend abstrahierter Sichten
- Client/Server-Modell, gemeinsamer Zugriff auf verteilte Dienste
- Verteiltes System, Verteilung des Softwaresystems über mehrere Rechner

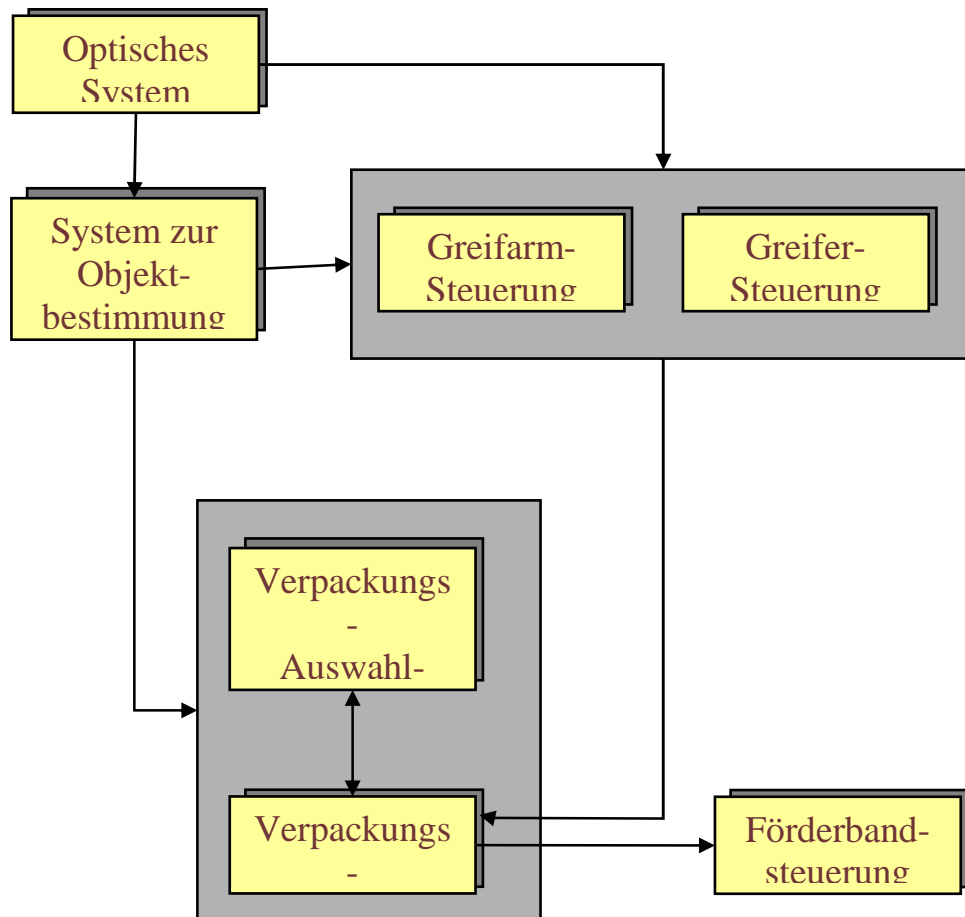
Blockdiagramm

Das Blockdiagramm ist die stärkste Vereinfachung des Architekturentwurfs. Jeder Block stellt für sich ein Subsystem dar, das weiter zerlegt werden kann. Blöcke innerhalb von Blöcken bedeuten, dass ein Subsystem selbst wiederum hierarchisch aus Subsystemen aufgebaut ist. Pfeile zwischen den Blöcken deuten an, dass Daten oder Steuerungsbefehle zwischen den Subsystemen in Richtung der Pfeile übertragen werden.

Das Blockdiagramm wird in der ersten Phase des Architekturentwurfs verwendet, um das zu entwickelnde System in eine Anzahl miteinander zusammenarbeitender Subsysteme aufzuteilen. Das Blockdiagramm gibt einen Überblick über die Systemstruktur und ist für die Entwickler, die in den Systementwicklungsprozess einbezogen sein können, einfach zu verstehen.

Beispiel für ein Blockdiagramm: Robotersystem zur automatischen Verpackung

Das Robotersystem kann verschiedenartige Objekte verpacken. Es benutzt dazu ein optisches Subsystem, um Objekte auf einem Förderband auszuwählen, den Typ des Objektes zu bestimmen und die entsprechende Verpackungsart zu bestimmen. Danach werden die Objekte vom Förderband zur Verpackungseinheit transportiert. Verpackte Objekte werden dann auf ein anderes Förderband gelegt.



→ Daten oder Steuerungsbefehle

Datenmodelle

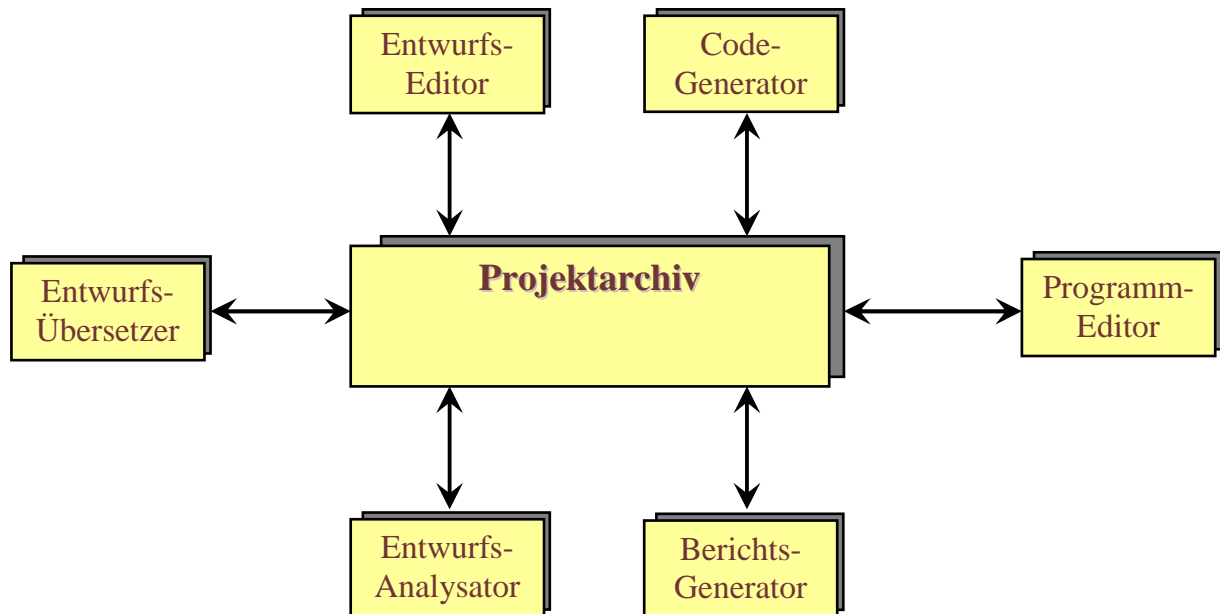
Bei Datenmodellen müssen die Subsysteme, aus denen ein Gesamtsystem besteht, Informationen austauschen, um auf effektive Weise miteinander zu arbeiten. Es gibt zwei grundlegende Methoden, wie dies geschehen kann:

1. **Datenspeichermodell (auch Repository-Modell):**
Alle gemeinsam benutzten Daten werden in einer zentralen Datenbank gespeichert, die für alle Subsysteme zugänglich ist.
2. **Dezentrales Datenmodell:**
Jedes Subsystem unterhält seine eigene Datenbank. Der Datenaustausch mit anderen Subsystemen erfolgt durch das Versenden von Nachrichten.

Das Datenspeichermodell eignet sich besonders für Systeme, die auf einer großen globalen Datenbank bzw. Repository aufgebaut sind. Dies gilt insbesondere für Anwendungen, bei denen große Datenmengen von einem Subsystem generiert und von einem anderen verarbeitet werden. Beispiel dazu sind Betriebsleitsysteme, Produktionsplanungssysteme, CAD-Systeme, CASE-Werkzeugsammlungen

Beispiel für Datenspeichermodell: Werkzeugsammlung

Die Werkzeugsammlung besteht aus einem zentralen Datenspeicher, auf den verschieden andere Komponenten wie Entwurfseditor, Codegenerator usw. zugreifen.

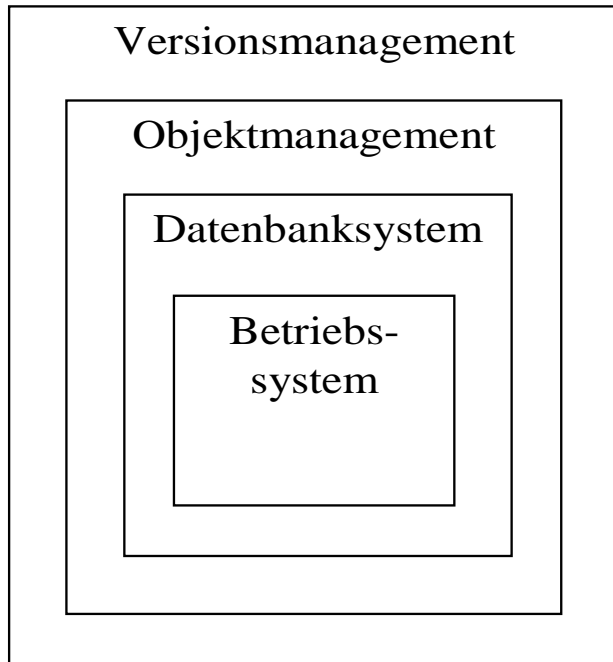


Schichtenmodell

Das Schichtenmodell ist ein Architekturmodell, das auf einer Hierarchie aufsteigender Abstraktionsebenen aufbaut und entsprechend dieser Hierarchie schichtweise aufgebaut ist. Dabei stellt jede Schicht des Modells an ihrer Schnittstelle ein ein Paket von Diensten bereit, zu deren Implementierung nur die von den darunter liegenden Schichten bereitgestellten Dienste verwendet werden (dürfen). Dienste höherer Schichten dürfen nicht in Anspruch genommen werden.

1.) Beispiel für ein Schichtenmodell: Konfigurationsmanagement

Das Konfigurationsmanagement verwendet als erste Schicht das Versionsmanagementsystem. Dieses unterstützt das Konfigurationsmanagement, indem es Versionen von Objekten verwaltet. Dazu greift es wiederum auf ein Objektmanagementsystem als zweite Schicht zurück. Dieses stellt Speicher- und Verwaltungsdienste für die Objekte bereit. Das Objektmanagementsystem verwendet seinerseits ein Datenbanksystem als nächste Schicht, das die grundlegende Speicherung der Daten und Dienste wie Transaktionsmanagement, Rückgängigmachung und Wiederherstellung sowie Zugriffskontrolle anbietet. Das Datenbanksystem benutzt schließlich zu seiner Implementierung als unterste Schicht die vom Betriebssystem angebotenen Dienste, insbesondere zur Dateispeicherung.



2. Beispiel für ein Schichtenmodell: 7 Schichten des Open System Interconnection (OSI-) Referenzmodells für Netzwerkprotokolle

Das bekannte OSI Netzwerkmodell mit seinen 7 Schichten reicht von der untersten Schicht der physikalischen Datenübertragung über verschiedenen Zwischenschichten hinauf zur Schicht über die Anwendungen logisch miteinander kommunizieren.

7	Anwendung	Bereitstellung einer Reihe von Funktionalitäten auf Anwendungsebene (z.B. Datenübertragung, E-Mail, Remote login)
6	Darstellung	Standardisierte Formatierung der Datenstrukturen zwecks einheitlicher Semantik ausgetauschter Daten (u.a. Kodierung, Kompression, Kryptographie)
5	Sitzung	Zuordnung einer Reihe diskreter Kommunikationsvorgänge zu einem kontinuierlichen benutzerspezifischen Kommunikationsprozess (u.a. organisatorische Maßnahmen zum Aufbau und Abbau der Sitzung)
4	Transport	vollständige Punkt-zu-Punkt (Sender-Empfänger, verbindungsorientierte) Kommunikation (u.a. Segmentierung von Datenpaketen und Routing-Optimierung zur Stauvermeidung)
3	Vermittlung	lokale (verbindungslose) Weitervermittlung von Datenpaketen (u.a. Verwaltung interner Routing-Tabellen)
2	Sicherheit	Gewährleistung fehlerfreier Datenübertragung, geregelter Zugriff auf das Übertragungsmedium (z.B. Fehlerkorrektur, Vermeidung und Erkennung von Zugriffskollisionen)
1	Bitübertragung	physikalische Schicht, Übertragungsmedium (z.B. Glasfaserkabel)

Als wesentlicher Vorteile des Schichtenmodells begünstigt es die schrittweise Entwicklung von Systemen. Sobald eine Schicht entwickelt worden ist, können einige der in dieser Schicht enthaltenen Dienste für die Benutzer verfügbar gemacht werden. Zudem ist ein Schichtenmodell relativ leicht veränderbar dahingehend, dass solange die Schnittstellen zwischen den Schichten nicht verändert werden, kann eine Schicht komplett durch eine andere ersetzt werden. Sollten sich die Schnittstellen einer Schicht hingegen ändern, wird davon nur die angrenzende Schicht betroffen. Die Portierbarkeit wird unterstützt, da bei Schichtensystemen die maschinenabhängigen Eigenschaften in den unteren Schichten angeordnet sind und den oberen Schichten nicht bekannt sein müssen. So kann ein

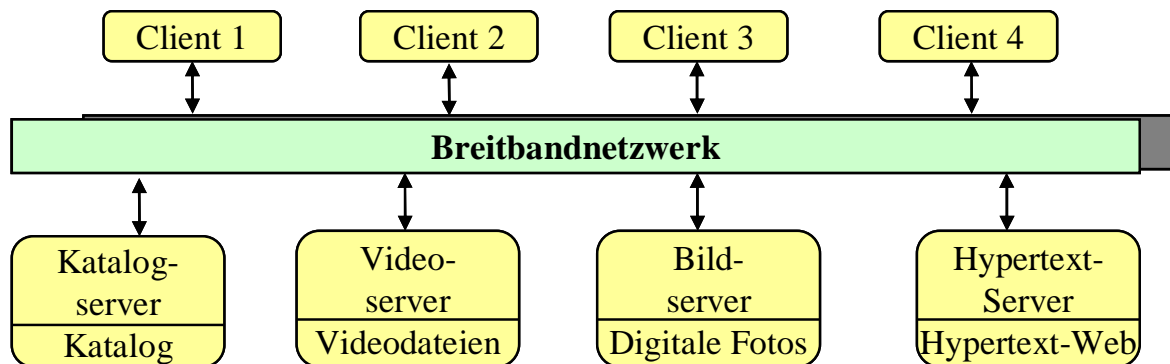
Schichtenmodell relativ leicht auf anderen Computern übertragen werden, indem man nur die inneren bzw. untersten, maschinenspezifischen Schichten neu erstellt.

Client/Server-Modell

Das Client/Server-Architekturmodell ist ein verteiltes Systemmodell, das darstellt, wie Daten und Prozesse über eine Menge von Prozessoren verteilt werden können. Die Hauptkomponenten dieses Systems sind:

- Eine Menge unabhängiger Server, welche Dienste für andere Subsysteme anbieten. z.B. Druckserver (Druckerdienste), Dateiserver (Dateiverwaltungsdienste), Compiler-Server (Übersetzungsdienste)
- Eine Menge i. A. unabhängiger Clients, welche die von den Servern angebotenen Dienste abrufen. Es kann durchaus vorkommen, dass ein Clientprogramm mehrmals gleichzeitig ausgeführt wird.
- Ein Netzwerk, über das die Clients auf die Dienste zugreifen können. (prinzipiell nicht erforderlich, falls Clients und Server auf demselben Computer laufen)

Beispiel für Client/Server-Modell: Hypertextsystem



Im Beispiel stellt das Hypertextsystem ein Client/Server System für die Verwaltung einer Film- und Fotobibliothek dar. Diverse Server zeigen die verschiedenen Medienarten an und verwalten sie. Einzelbilder von Videos erfordern eine schnelle und synchrone Übertragung, aber nicht unbedingt eine hohe Auflösung und können in einem Speicher komprimiert werden. Bei Standbildern ist dagegen eine Übertragung in hoher Auflösung erforderlich. Der Katalog muss in der Lage sein, eine Vielfalt von Anfragen zu bewältigen und Verbindungen zu dem Hypertextinformationssystem anzubieten. Bei dem Clientprogramm kann es sich hier lediglich um eine integrierte Bedienoberfläche zu diesen Diensten handeln, die nur die über das Netzwerk übertragenen Daten geeignet darstellen.

Anmerkung: Das oben dargestellte Client Server System kann auch als verteiltes System (s. unten betrachtet werden.)

Verteilte Systeme

Bei verteilten Systemen läuft die Software auf einer durch ein Netzwerk nur lose miteinander verbundenen Gruppe von Rechnern. Verteilte Systeme bestehen aus lose integrierten, unabhängigen Teilen. Bei verteilten Systemen wird die Software, die die Teile des Systems verwaltet und die plattformübergreifende Kommunikation zwischen den Systemteilen ermöglicht, als Middleware bezeichnet.

Verteilte Systeme haben Vor- und Nachteile. Zu den Vorteilen zählen

- Ressourcenteilung. Diese ermöglicht gemeinsame Nutzung von Hard- und Software seitens der verteilten Module
- Offenheit, d.h. Hard- und Software können von unterschiedlichen Herstellern stammen. Um zusammen zu arbeiten müssen Hard- und Software allerdings die für das verteilte

System definierten Schnittstellen beachten. Für verteilte Systeme, in denen Soft- und Hardware verschiedener Hersteller zusammenarbeiten, gibt es oftmals entsprechende Standardisierungsgremien, die die notwendigen Schnittstellen in Form von Normen und Standards definieren.

- Nebenläufigkeit, d.h. Module, die unabhängig voneinander sind, können parallel zueinander laufen.
- Skalierbarkeit heißt hier, dass Kapazitäten durch Hinzufügen von Ressourcen erweitert werden können.
- Fehlertoleranz bedeutet, dass Hard- oder Softwarefehler können durch mehrfaches Vorhandensein von Ressourcen oder Daten unter Umständen toleriert werden. Fehler im Netzwerk, über das die verteilten Komponenten miteinander kommunizieren, kann aber zum Ausfall des Systems führen.
- Transparenz
 - Die verteilte Struktur ist für den Benutzer unsichtbar.

Verteilte Systeme haben aber auch einige Nachteile, die zu berücksichtigen sind:

- Erhöhte Komplexität im System, das Verhalten des Gesamtsystems ist schwerer zu analysieren und / oder zu verstehen als monolithische Systeme.
- Informationssicherheit kann schlechter sichergestellt werden, da Daten über ein (ggf. öffentliche) Kommunikationsnetz transportiert werden. Schutz vor Mithören oder Verfälschen von Daten oder Angriffe gegen das System von Außen ist nur mit entsprechendem Aufwand sichergestellt.
- Die Verwaltung verteilter Systeme ist aufwendig. Unterschiedliche Plattformen, Betriebssysteme und SW-Versionen müssen berücksichtigt werden. Ggf. sind Netzwerkmanagement-Dienste erforderlich.
- Die Vorhersage der Systemperformance ist schwierig. Reaktionen des Systems sind von der Systemauslastung und Netzwerkbelastung abhängig.

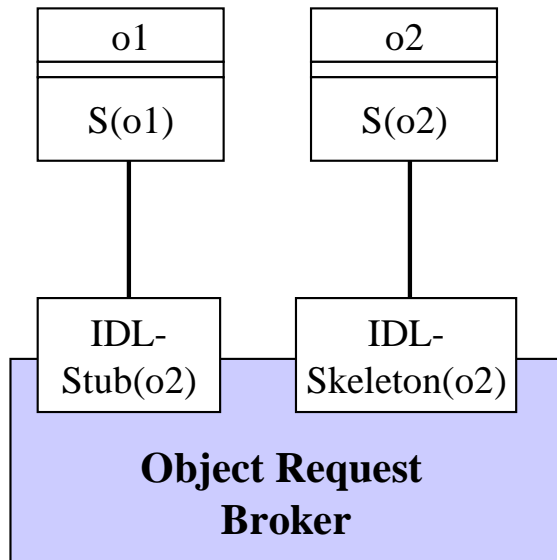
CORBA ist ein typisches und verbreitetes Beispiel einer Middleware. CORBA steht für Common Objects Request Broker Architecture und ist eine konkrete Architektur für verteilte Systeme. CORBA basiert auf einer Reihe von Standards, die von der Object Management Group (OMG) definiert wurden. Verschiedene CORBA-Implementierungen sind sowohl für Windows- als auch für UNIX-Plattformen von verschiedenen Herstellern entwickelt worden. Ein CORBA-System besteht aus einer Menge verteilter Objekte, die in unterschiedlichen Programmiersprachen implementiert sein können, jedoch die spezifizierten Schnittstellen umsetzen müssen. Die Middleware zur Kommunikation zwischen den verteilten CORBA Objekten wird ObjectRequest Broker (ORB) genannt. (DCOM ist die von Microsoft entwickelte "Konkurrenz" zum CORBA. Es gibt sogar CORBA-DCOM-Bridges)

Bestandteile von CORBA:

CORBA besteht aus den folgenden Elementen:

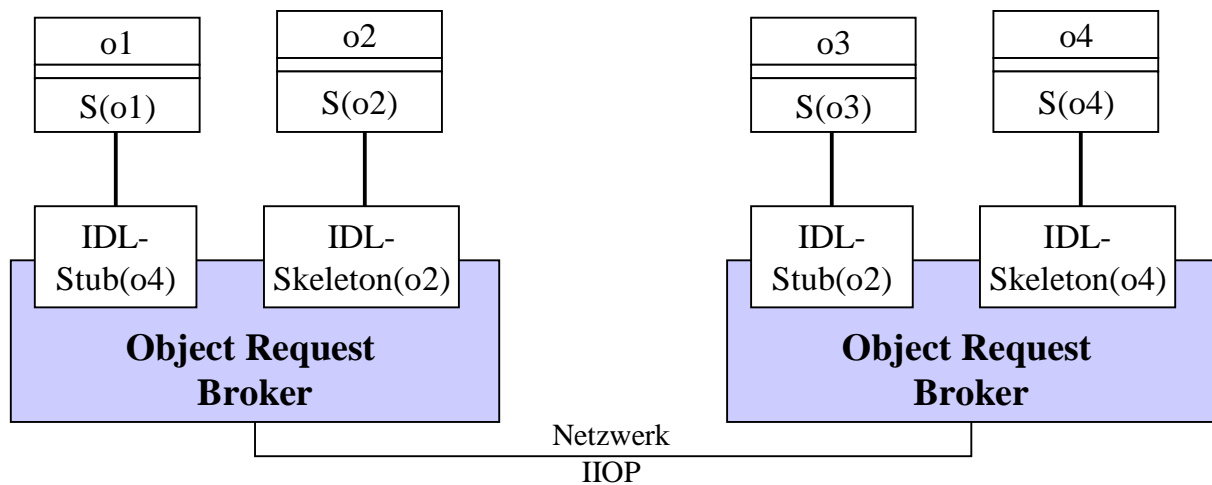
- Ein Modell für Anwendungsobjekte, umfasst u.a. die Aufrufsemantik der Objekte, sprachneutrale Beschreibung der Schnittstellen mittels der sog. Interface Definition Language (IDL)
- Der schon erwähnte Object Request Broker (ORB) vermittelt Anfragen an Objektdienste. Diese können allgemeine Objektdienste (z. B. Persistenz-Dienste zum Sichern von Objekten, Verzeichnis-Dienste die das einfache Auffinden von Objekten ermöglichen).
-

Die Objektkommunikation über den ORB wird folgendermaßen implementiert. Jedes dienst-erbringende Objekt besitzt einen IDL-Skeleton, der die Objektdienste aufruft. Zu jedem Aufrufer eines Objektes wird ein IDL-Stub erzeugt, der die Schnittstelle des dienst-erbringenden Objektes anbietet. (s. Bild unten)

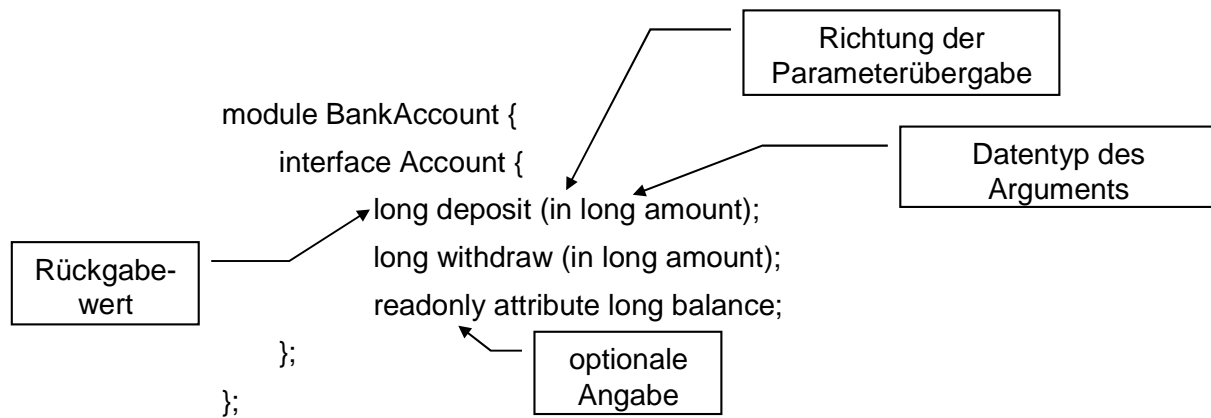


Im obigen Beispiel sei o1 sei das aufrufende Objekt. o2 biete den aufgerufenen Dienst S(o2) an. Der IDL-Stub(o2) bietet die Schnittstelle des aufgerufenen Objekts o2 an und gibt die Aufrufe von o1 über den Object Request Broker an den IDL-Skeleton(o2) weiter. Der IDL-Skeleton(o2) ruft den Dienst S(o2) auf.

Die Kommunikation zwischen den ORBs erfolgt über das TCP/IP basierte Internet Inter-ORB-Protocol (IIOP). Dies ermöglicht die Anforderung von Diensten auf anderen Rechnern im Netzwerk. Im Beispiel unten ruft o1 o4 auf und o3 ruft o2 auf.



Das Beispiel unten zeigt einen kleines Beispiel für eine Beispiel für IDL-Beschreibung in CORBA.



Es stehen 3 Arten der Parameterübergabe zur Verfügung, in-Parameter zur Übergabe vom aufrufenden Objekt (Client) zum aufgerufenen Objekt (Server), out-Parameter zur Übergabe vom Server zum Client und inout-Parameter zur Übergabe vom Client zum Server und zurück.

Das weitere "echo" Beispiel¹⁰ unten zeigt eine einfache CORBA Test-Klasse, die einen Eingabe-String wieder zurück "echoed"

```

module Test
{ interface Echo
  {
    // Echo the given string back to the client.
    string          // The returned string - which is the same as Message
    Echo_String (
      in string Message // The string which should be echoed.
    );
  };
};

```

Software-Feinentwurfs

Das Ziel des Software-Feinentwurfs ist es, die im Software-Grobentwurf erstellte Software-Architektur zu verfeinern. Der Feinentwurf beschreibt die Detailstruktur des Systems, evtl. angepasst an die Besonderheiten der Implementierungssprache und Plattform. Dazu gehört u.a. das Sprachenparadigma (funktional, objektorientiert), Speicherbeschränkungen (besonders in eingebettete Systeme) oder ein besonderer Befehlssatz des Zielprozessors.

Der Feinentwurf sollte so detailliert sein, dass (theoretisch) der Programmierer den Programmcode ohne eigene Interpretationen der Aufgabenstellung in ausführbaren Code überführen kann¹¹.

Beim Software-Feinentwurf steht die Detaillierung der Systemkomponenten im Vordergrund während beim Grobentwurf die Aufteilung des Gesamtsystems in Systemkomponenten das Ziel ist einschließlich der Beschreibung der Beziehungen der Systemkomponente untereinander. Im Software-Feinentwurf wird der Grobentwurf so detailliert, dass alle Funktionen / Operationen der Systemkomponenten auf Realisierungsebene detailliert beschrieben werden.

Betrachtung einzelner Systemkomponenten

¹⁰ Quelle: Wikipedia:

File: test-echo.idl ([http://cvs.sourceforge.net/viewcvs.py/adacl/WikiBook_Ada/Source/test-echo.idl?only_with_tag=HEAD&view=markup view])

¹¹ Dies ist eher eine theoretische Forderung, da „Hintergrundinformationen“ wie Anforderungen es dem Programmierer einfacher machen, den Feinentwurf in Code umzusetzen.

- Die Betrachtung einzelner Systemkomponenten ermöglicht eine Aufteilung der Entwurfs- und Implementierungstätigkeiten auf mehrere Mitarbeiter
- Diese Aufteilung ist speziell bei größeren Projekten unerlässlich, damit das zu entwickelnde Softwareprodukt in einem angemessenen Zeitrahmen entwickelt werden kann.

Programmiersprachenneutrale Notationen

Das Ziel programmiersprachenneutraler Notationen ist es Operationen (typischerweise Algorithmen) durch Abstrahieren, d.h. Weglassen syntaktischer Programmiersprachendetails zu beschreiben. Vorteile dieses Vorgehens sind dass die Wahl der Programmiersprache später erfolgen kann und durch das Weglassen weniger wichtiger Details (Abstraktion) die Aufmerksamkeit auf die Korrektheit des Algorithmus gelenkt werden kann (leichtere Analysierbarkeit, eventuell bessere Lesbarkeit). Beispiel dafür sind Struktogramme und Pseudocode

Struktogramme

Ein Struktogramm (Nassi-Shneiderman-Diagramm) ist eine Entwurfsmethode für die strukturierte Programmierung. Es ist genormt nach DIN 66261. Benannt wird es nach seinen Vätern Dr. Ike Nassi und Dr. Ben Shneiderman.

Mittels Struktogrammen lassen sich Algorithmen programmiersprachenneutral (bezüglich prozeduraler Sprachen) darstellen. Ansatz ist es, ein Gesamtproblem, das man mit dem gewünschten Algorithmus lösen will, in immer kleinere Teilprobleme zu zerlegen, bis schließlich nur noch elementare Grundstrukturen wie Sequenzen und Kontrollstrukturen zur Lösung des Problems übrig bleiben. Diese können dann durch ein Struktogramm visualisiert werden.

Elemente des Struktogramms

Ein Struktogramm ist eine graphische Darstellung des Kontroll- und Datenflusses eines in prozeduraler Sprache darzustellenden Algorithmus. Dabei werden die vier Elemente eines strukturierten Programms

- Befehl (einzelne Anweisung)
- Sequenz (Ausführung mehrerer Befehle)
- Verzweigung (z.B. IF-Anweisung, CASE-Anweisung)
- Schleife (mehrfacher Durchlauf durch eine Sequenz, WHILE- oder FOR-Schleifen)

durch die im Folgenden angegebenen Diagrammartentypen versinnbildlicht. Aus diesen lässt sich ein Struktogramm rekursiv herleiten.

- Sequenz

1. Anweisung
2. Anweisung
3. Anweisung
...

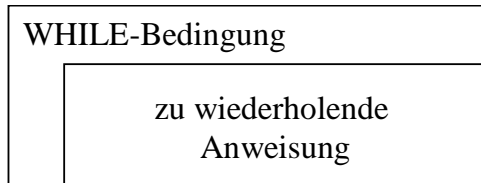
- IF-Anweisung

Bedingung	
Ja	Nein
THEN-Anweisung	ELSE-Anweisung (evtl. leer)

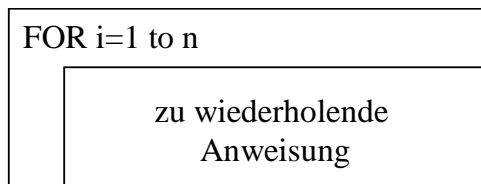
- CASE-Anweisung

Ausdruck				
Wert 1	Wert 2	Wert 3	...	Sonst
Anweisung 1	Anweisung 2	Anweisung 3	...	Anweisung n

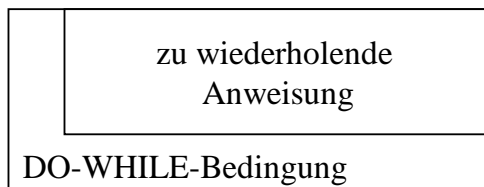
- WHILE-Anweisung



- FOR-Anweisung

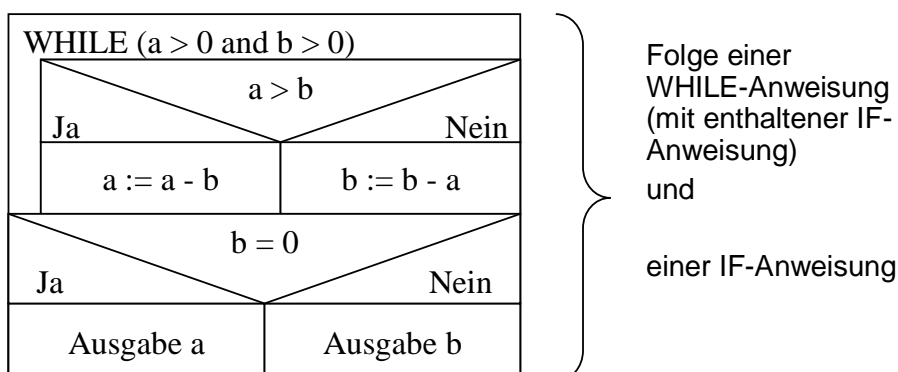


- DO-WHILE-Anweisung



Die innere Anweisung wird wiederholt, solange die Bedingung wahr ist

Als Beispiel sein hier die Berechnung des größten gemeinsamen teilers (ggT) einer Zahl aufgeführt (euklidische Algorithmus)



Pseudocode

Pseudocode ist eine Darstellung eines Algorithmus in einer intuitiv verständlichen Sprache, die an eine Programmiersprache angelehnt ist, aber noch leichter lesbar ist als ausformulierter Programmcode. Dabei werden vornehmlich natürlich sprachliche bzw. mathematische Darstellungselemente verwendet. Wie schon erwähnt, stehen syntaktische Details der Ziel-

sprache stehen nicht im Vordergrund. Das Beispiel unten zeigt einen Algorithmus der Zahlen aus einer Datei einliest und deren Summe berechnet.

```
sum := 0
read i from file
while i > 0 do
    sum := sum + i
    read i from file
end-while
```

4.4. Objektorientierte Analyse

Nachdem in den vorigen Kapiteln die Grundprinzipien der Architektur und des Feindesings erläutert wurden, soll im Weiteren speziell auf die Objektorientierte Methodik eingegangen werden.

Objektorientierte Methodologie

Die objektorientierte Methodologie wurde Ende der 80er und Anfang der 90er Jahre entwickelt. Parallel dazu wurden objektorientierte Programmiersprachen wie Eiffel (Bertrand Meyer), C++ (Bjarne Stroustrup), Objective-C (Brad Cox) und später Java und C# entwickelt. Das Hauptziel der Objektorientierung war es, eine verbesserte Wiederverwendbarkeit und damit geringere Entwicklungskosten bei der Software-Entwicklung zu erreichen.

Die Objektorientierung baut auf schon früher existierende Entwurfs- und Implementierungsmethoden auf. Dazu gehören

- das Strukturierte Programmieren unter Verwendung von Ablaufstrukturen (z.B. bedingte Anweisung und Schleifen) statt rein sequentieller Reihung von Anweisungen und Sprunganweisungen.
- die Modularisierung, d.h. Lösung einer Gesamtaufgabe durch Lösung von Teilaufgaben in abgegrenzten Modulen bei sauberer Aufgliederung nach globalen Daten (Variablen), Übergabevariablen und lokalen Variablen.
- Information Hiding, d.h. die Trennung zwischen dem Zugriff auf Information und Implementierung der Informationsverwaltung (z. B. bei relationalen Datenbanken).

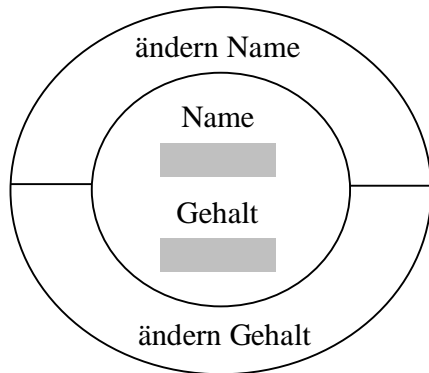
Zur Modellierung von SW werden in objektorientierte Modelle verwendet. Diese sind Abstraktionen der Realität. Dabei besteht die Kunst bei der Modell-Bildung darin, alle wesentlichen Gesichtspunkte einer Problemstellung zu erfassen *und nicht Wesentliches zu vernachlässigen* (abstrahieren, weglassen). Objektorientierte Modelle sind also Abstraktionen im Bereich der Software-Entwicklung. Sie orientieren sich an den Objekten *und* den Funktionen, die zur Manipulation dieser Objekte benötigt werden. Diese Parallelität von Daten- und Funktionsbetrachtung ist das wichtigste Merkmal der zu Grunde liegenden Konzepte und Vorgehensweisen der Objektorientierung.

Ein wesentliches Element der Objektorientierung ist die Klasse. Die Klasse abstrahiert die relevanten Eigenschaften von Objekten (s. unten) in eine gemeinsame Struktur, die aus der Zusammenfassung einer Datenstruktur und der darauf anwendbaren Operationen (sog. Methoden)

zu einer Einheit, der Klasse, besteht. Die Klasse hat den Charakter eines Datentyps, von dem es viele Ausprägungen (sog. Instanzen = Objekte einer Klasse) geben kann.

Als Beispiel sei hier die Klasse Mitarbeiter vorgestellt. Die Klasse „Mitarbeiter“ besteht aus den beiden Feldern „Name“ und „Gehalt“ und den darauf arbeitenden Methoden „ändern Name“ und „ändern Gehalt“.

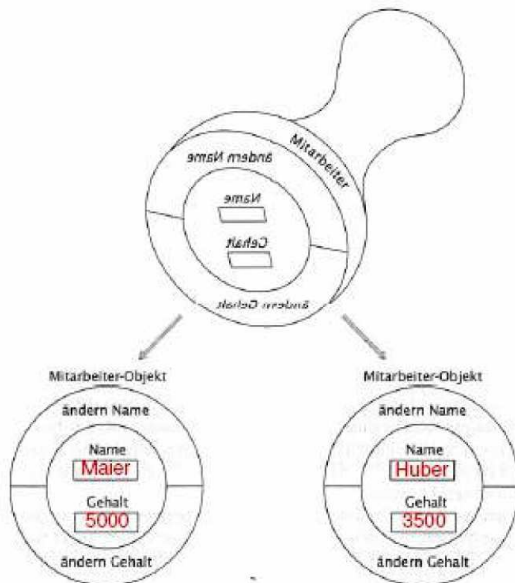
Die Klasse "Mitarbeiter"



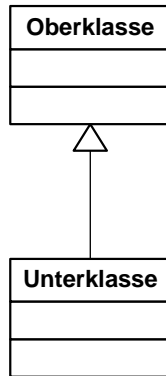
UML-Notation

Mitarbeiter
-Name -Gehalt
+ändern Name() +ändern Gehalt()

Ein Objekt ist in der objektorientierten Programmierung ein Softwaregebilde mit individuellen Merkmalen. Es definiert sich über seine Identität, seinen Zustand und sein Verhalten („Ist was, hat was, kann was“). Der Zustand eines Objekts ist durch die Werte der Instanzvariablen festgelegt. Das Verhalten eines Objekts wird durch Methoden implementiert. Die In klassischen objektorientierten Sprachen sind alle Objekte Instanzen von Klassen. Die Klasse stellt also eine Schablone dar, nach der Objekte (dieser Klasse) erzeugt werden können.



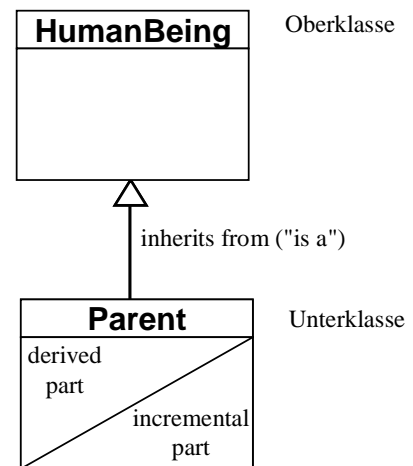
Ein weiteres wichtiges Element der Objektorientierung ist die Vererbung. Vererbung bedeutet, dass eine spezialisierte Klasse (Unterklasse, subclass, abgeleitete Klasse) über Eigenschaften einer oder mehrerer allgemeiner Klassen (Oberklassen, super classes, Basisklassen) verfügen kann. Eine Unterklasse ist vollständig konsistent mit ihrer Oberklasse bzw. ihren Oberklassen, enthält aber in der Regel zusätzliche Informationen. Ein Objekt der Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erlaubt ist, umgekehrt natürlich nicht. Durch die Vererbung entsteht eine Klassen-Hierarchie bzw. eine Vererbungsstruktur.



Als Beispiel für Vererbung sei hier die Klasse Elternteil als Unterklasse der Klasse Mensch gegeben.

```
class HumanBeing
{
    private int    age;
    private float  height;
    // public declarations of operations on HumanBeing )
} // class HumanBeing

class Parent extends HumanBeing
{
    private int    numberOfChildren;
    private String nameOfOldestChild;
    // public declarations of operations on Parent )
} // class Parent
```



Bei der Verwendung der Vererbung können Probleme auftreten. Die Änderung einer Oberklasse ist riskant, weil die Vererbung eingesetzt wurde, um die bestehende Implementierung einer Klasse wieder zu verwenden (Inheritance for Code Reuse). Die Unterklasse wird automatisch mit geändert und verändert unter Umständen ihr Verhalten. Darüber hinaus verwenden Unterklassen auch Internas (d.h. gekapselte Informationen) der Oberklasse. Somit wird die Kapselung an dieser Stelle aufgebrochen. Des Weiteren erlaubt die Vererbung die nahezu uneingeschränkte Redefinition von Operationen der Unterklasse, wodurch semantische Inkompatibilitäten zwischen redefinierter und ursprünglicher Operation entstehen können.

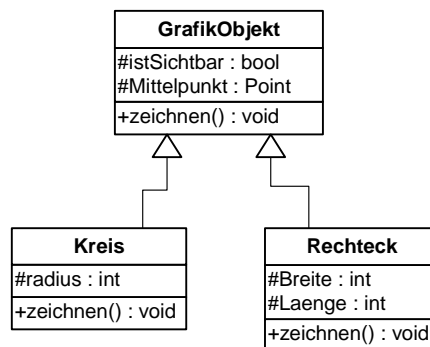
Im Zusammenhang mit der Vererbung steht der Begriff der Polymorphie (engl.: polymorphism). Polymorphie bedeutet Vielgestaltigkeit. Im Sinne der Objektorientierung ist es das Prinzip, dass sich hinter einer Objektreferenz Realisierungen unterschiedlicher Objekte bzw. Methoden innerhalb derselben Klassenhierarchie verbergen können. So kann zum Beispiel ein Pointer auf ein Oberklassenobjekt auch auf ein Unterklassenobjekt zeigen. Eine Methode kann in einer Klassenhierarchie mehrfach auftreten, sowohl in der Oberklasse als auch in der Unterklasse. das bedeutet, dass sich hinter dem gleichen Namen unterschiedliche Semantiken in verschiedenen Unterklassen verbergen können.

Zum Tragen kommt die Polymorphie insbesondere bei der späten Bindung, wenn an einer Objektreferenz einen Methode aufgerufen wird und erst zur Laufzeit entschieden wird, welche Methode ausgeführt wird. Voraussetzung ist, dass der Aufrufer wissen muss, dass das

aufgerufene Objekt die gewünschte Methode besitzt. Er muss nicht wissen, zu welcher Klasse das Objekt gehört. Erst zur Laufzeit des Programms wird bestimmt, zu welcher Klasse das aufgerufene Objekt gehört. (Man spricht daher von später oder dynamischer Bindung. Polymorphie und spätes Binden (late binding) sind untrennbar verbunden.) Dieser Mechanismus ermöglicht es, flexible und leicht änderbare Software-Systeme zu entwickeln. Nachteil der Späten Bindung ist die zusätzlich notwendige Laufzeit, um zu entscheiden, welche Methode aufgerufen werden soll.

Ähnlich ist es mit der Dynamischen Operator Bindung, bei der unterschiedliche, gleichnamige Operatoren abhängig vom Parametertyp aufgerufen werden.

Als Beispiel für Polymorphie seien die Klassen GrafikObjekt, Kreis und Rechteck mit der entsprechenden Vererbungshierarchie gegeben:



Ein Operationsaufruf grafik.zeichnen() an einer Objektreferenz grafik der Klasse GrafikObjekt kann völlig unterschiedliche Wirkungsweisen besitzen. Referenziert grafik eine Instanz der Klasse Kreis, wird die Operation Kreis.zeichnen() aktiviert. Referenziert grafik hingegen eine Instanz der Klasse Rechteck, dann wird Rechteck.zeichnen() ausgeführt. Es wird erst zur Laufzeit des Programms bestimmt, ob die Variable grafik auf ein Kreis- oder ein Rechteck-Objekt zeigt.

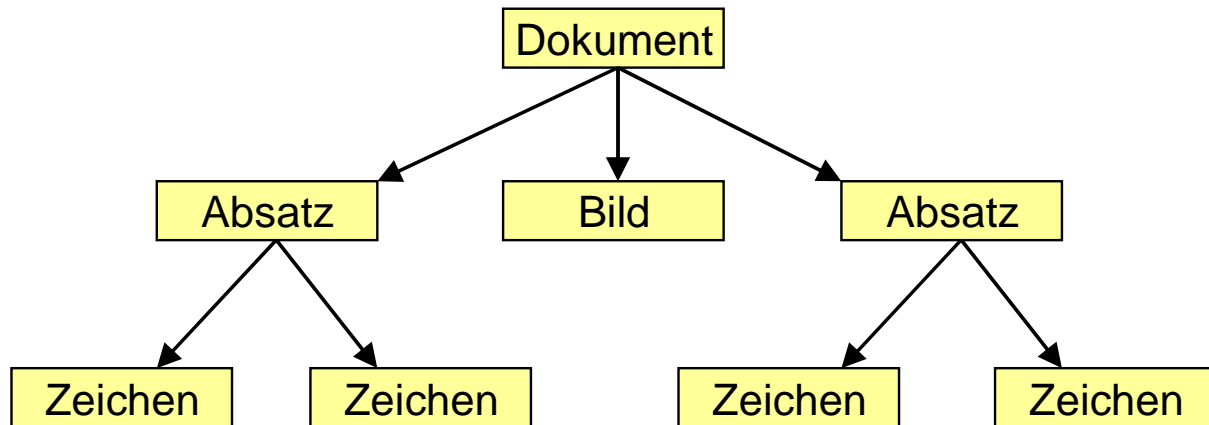
Delegation und Propagation

Delegation ist ein Mechanismus, bei dem ein bei dem ein Objekt eine Nachricht nicht vollständig selbst interpretiert, sondern an andere Objekte weiterleitet (*propagiert*), die ihrerseits einen Teil zur Ausführung der Operation beitragen. Mittels Delegation können komplexe Strukturen, z.B. Bäume oder Listen auf elegante Weise vollständig traversiert, d.h. durchlaufen und dabei ab- oder bearbeitet werden.

Typische Anwendung von Propagationsmechanismen:

Vollständiges Traversieren eines Aggregationsbaumes mit Start beim Wurzelobjekt, z.B. zur Realisierung von Operationen wie Speichern, Löschen, Kopieren oder Ausgeben des gesamten Aggregationsbaumes. Voraussetzung ist allerdings, dass alle an der Aggregation beteiligten Klassen den propagierten Operationsaufruf verstehen.

Am Speichern eines Dokumentes sei die Delegation und Propagation nochmals erläutert. Das in der folgenden Grafik dargestellte Objekt Dokument sei baumartig strukturiert und besteht aus einem Objekt der Klasse Absatz, einem Objekt der Klasse Bild und einem weiteren Objekt der Klasse Absatz. Die Absätze bestehen aus Objekten der Klasse Zeichen. Jede vorkommende Klasse verfügt über eine Methode save(), die das Objekt abspeichert.



Das gesamte Dokument wird auf folgende Weise gespeichert. Im ersten Schritt wird `save()` am Objekt Dokument aufgerufen. Das Dokument seinerseits gibt den Aufruf an seine Bestandteile weiter. Die Operation Bild speichert das Bild direkt ab, jeder Absatz delegiert das Abspeichern an jedes einzelne Zeichen. Die Zeichen werden direkt gespeichert. Anmerkung: Der Mechanismus der Propagation und Delegation ist nicht auf die Objektorientierte Programmierung beschränkt. Auch in der Funktionalen Programmierung lassen sich diese Mechanismen entsprechend anwenden.

4.5. UML

Dieses Kapitel soll einen Einblick in die Unified Modelling Language (UML) als Modellierungssprache für SW und ggf. auch Systeme geben. Zunächst soll erläutert werden, was UML ist, auch was UML nicht ist und wo UML herkommt.

Zunächst ist UML eine standardisierte Sprache ("Lingua Franca"), in der in allen Phasen des Software-Entwurfs (Architektur, Feindesign) Modelle methodisch erstellt und beschrieben werden können. UML dient zur Modellierung, Dokumentation, Spezifikation und Visualisierung komplexer Softwaresysteme. Seit ihrer Entstehung wurde UML die am meisten verbreitete Notation, um Software-Systeme zu analysieren und zu entwerfen.

Neben freiem Text sind die wesentlichen Elemente von UML Grafiken, die die Möglichkeit bieten statische und dynamische Modelle für Analyse, Architektur und Design zu erstellen. Die Wurzeln der UML liegen u.a. in der Objektorientierung, deshalb unterstützt hauptsächlich eine objektorientierte Vorgehensweise.

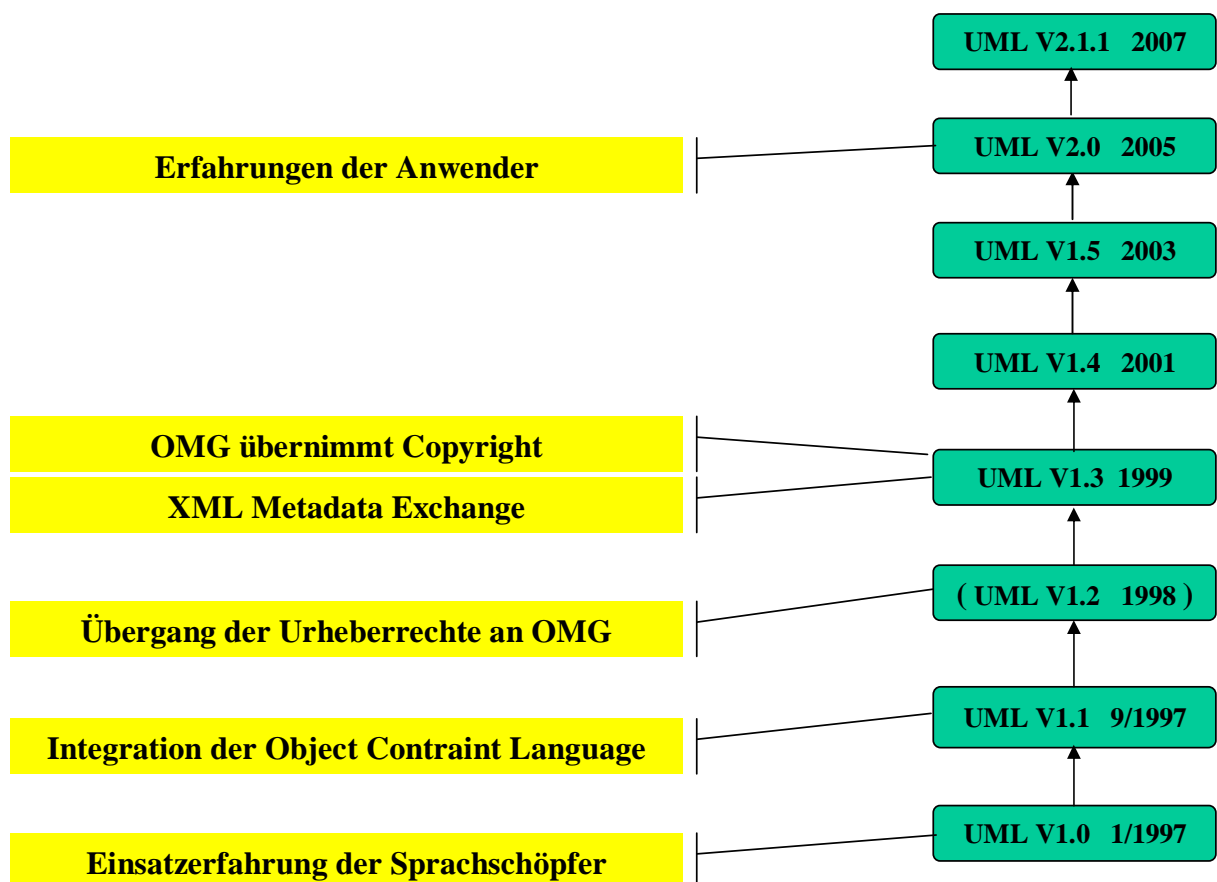
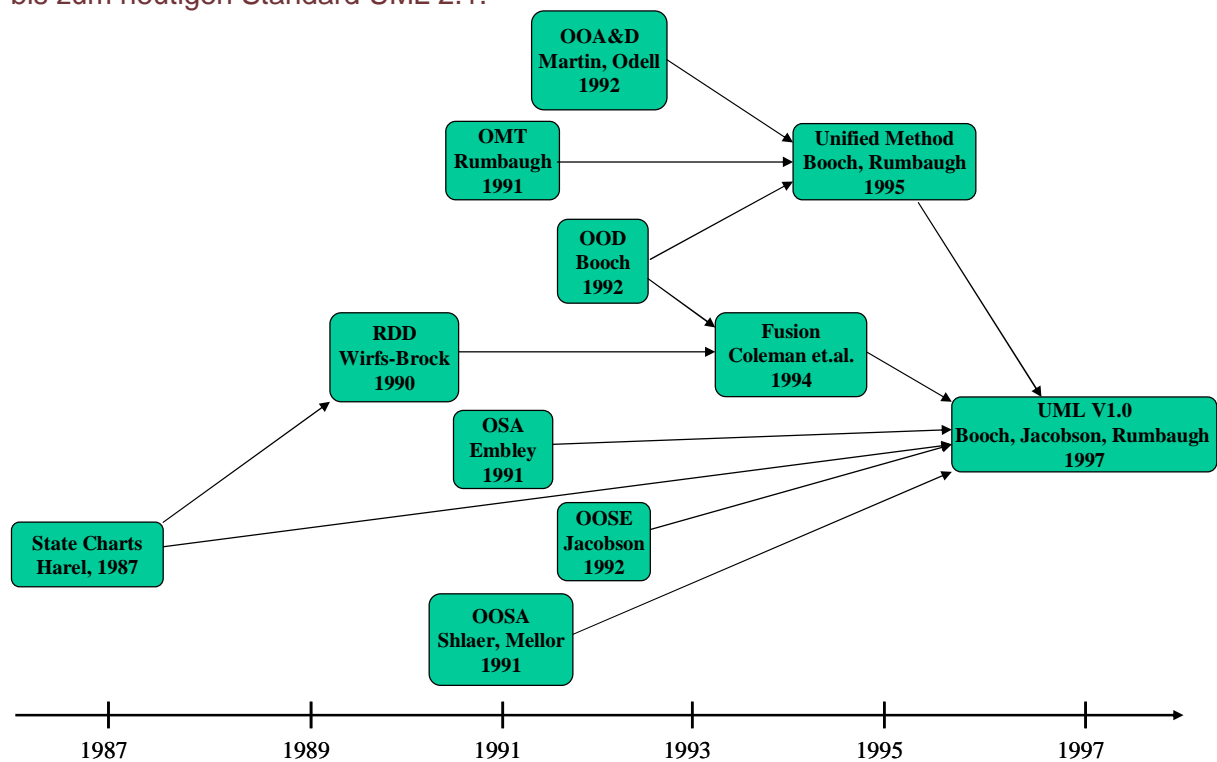
UML ist standardisiert von der Object Management Group (OMG). Die OMG existiert seit 1989 mit ca. 800 Mitgliedern¹². Die OMG erzeugt Industriestandards und ist das größte Konsortium im Software-Bereich. Wesentliche OMG-Mitglieder sind IBM, Microsoft, Oracle, HP, Daimler, Telelogix, I-Logix.

UML ist eine Modellierungssprache, die vielfältig eingesetzt werden kann. Dennoch ist UML nicht perfekt und nicht vollständig. Des Weiteren ist UML keine Programmiersprache und keine rein formale Sprache. Deshalb können UML Modelle u.a. nicht vollständig 1-zu-1 in Code übersetzt werden. UML ist nicht spezialisiert auf ein Anwendungsgebiet und die grafischen Beschreibungsmöglichkeiten ersetzen nicht 100% eine textuelle Beschreibung. UML ist auch keine Methode, d.h. UML legt keine Vorgehensweise bei der Modellierung fest.

Historisch stammt UML aus den 90er Jahren, in denen mehrere objektorientierte Analyse- und Designmethoden entstanden, die dann in der UML zusammengelaufen sind. Von diesen sind zu nennen OMT von Rumbaugh ("Object Modelling Technique"), OOD von Booch ("Object-oriented Design"), OOSE von Jacobson ("Object-oriented Software Engineering") und OOA & D von Martin/Odell ("Object-oriented Analysis and Design").

¹² die hohe Anzahl der Mitglieder belegt zum einen die Bedeutung der UML, erklärt zum anderen aber auch, dass UML umfangreich, komplex und in Teilen redundant geworden ist.

Die folgenden Bilder zeigen die historische Entwicklung von UML von den ersten Anfängen bis zum heutigen Standard UML 2.1.



Unterschiede UML 1 zu UML 2

Da sich UML 1 von der Version 2 deutlich unterscheidet soll hier noch kurz auf die hauptsächlichen Unterschiede zwischen den Versionen 1 und 2 und auf die Anforderungen an UML 2 eingegangen werden. UML 1.x war zu groß und komplex, die Modellierung von Systemen mit Echtzeitanwendungen war nicht oder nur eingeschränkt möglich. Die Komplexität führte zur Streichung von Sprachkonstrukten zur Vermeidung des "Second System Syndroms", d.h. nicht alle Elemente werden von Toolherstellern implementiert, manche Sprachelemente waren nicht in etablierten Vorgehensmodellen verwendet (z.B. parametrisierte Kollaborationen).

Manche Elemente vom UML 1 sind nur methoden- oder implementierungsspezifisch (z.B. Friend Stereotyp von C++) und einige Elemente hatten in UML 1 keine präziser Semantik (z.B. für viele Stereotypen in UML 1.0).

Neuerungen flossen in UML 2 ein bzgl. der Neuformulierung des Metamodells sowie weitestgehende Verwendung der Object Constraint Language (OCL) und Wiederverwendung von Basiskonstrukten¹³.

Des Weiteren wurde die Ausführbarkeit verbessert, insbesondere wurde eine stärkere Beziehung zwischen statischen und dynamischen Diagrammen eingeführt und erprobte Konzepte (z.B. Petri-Netze) integriert. Die Übersichtlichkeit wurde verbessert durch Verringerung der graphischen Modellkonstrukte und Basiskonzepte. Dadurch wird in einem Projekt die Kommunikation der Projektteilnehmer untereinander erleichtert.

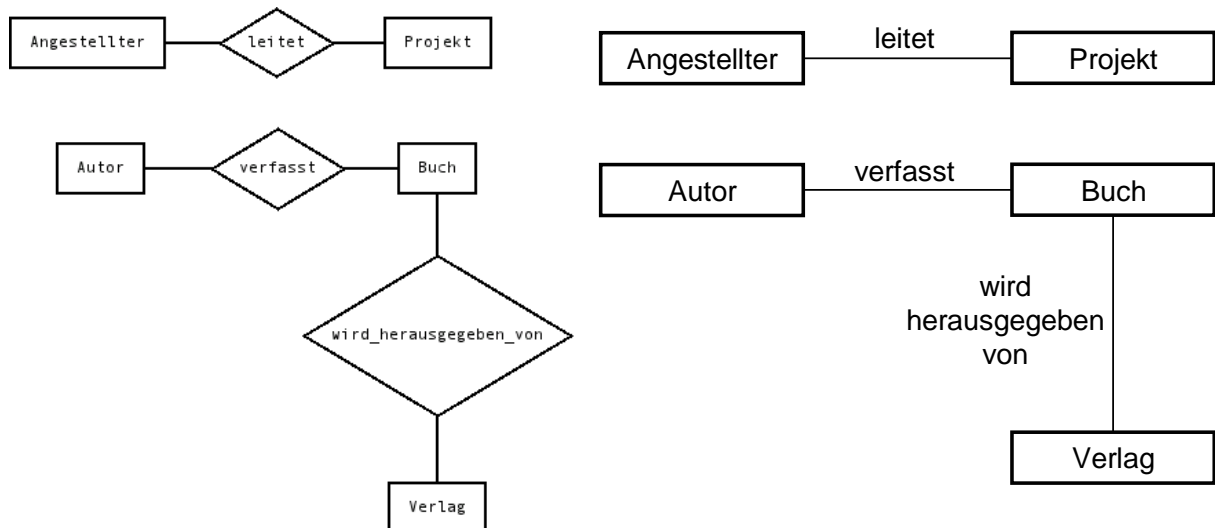
Um auch Echtzeitsysteme modellieren zu können, wurden in UML 2 geeignete Notationsmittel zur Spezifikation von Echtzeiteigenschaften hinzugefügt. Schwächen der Semantik wurden klargestellt z.B. von den Beziehungen Generalisierung, Abhängigkeit und Assoziationen. Die Kapselung und Skalierbarkeit Verhaltensmodellierung wurde verbessert, insbesondere bei Zustandsautomaten und Interaktionsdiagrammen. Einschränkungen der Aktivitätsmodellierung wurden aufgehoben. Die Möglichkeiten der hierarchischen Modellierung bzgl. der Systemzerlegung und Modellierung des "Innenlebens" von Komponenten und Funktionen wurden erweitert und die komponentenbasierte Entwicklung stärker unterstützt (z.B. Java Enterprise Edition).

UML Konzeptüberblick

Mit der UML steht nicht nur eine standardisierte Modellierungssprache zur Verfügung, sondern auch ein vollständiger Katalog von Software-Entwicklungskonzepten. Die so genannten Basiskonzepte gingen aus den Programmiersprachen hervor. Diese reichen aber nicht aus, um ein Fachmodell einer Problemstellung zu entwickeln. Es wurden deshalb Elemente der semantischen Datenmodellierung (siehe ERM) mit aufgenommen. Die Synthese zwischen der Welt der objektorientierten Programmier-Sprachen und der Welt der Datenmodellierung ermöglicht die zeitunabhängigen Aspekte eines Problemfeldes zu beschreiben (Statisches Modell). Zur Modellierung des dynamischen Verhaltens wurden weitere Konzepte mit den zugehörigen standardisierten Notationen eingeführt und ermöglichen so auch dynamisches Modelle zu entwerfen.

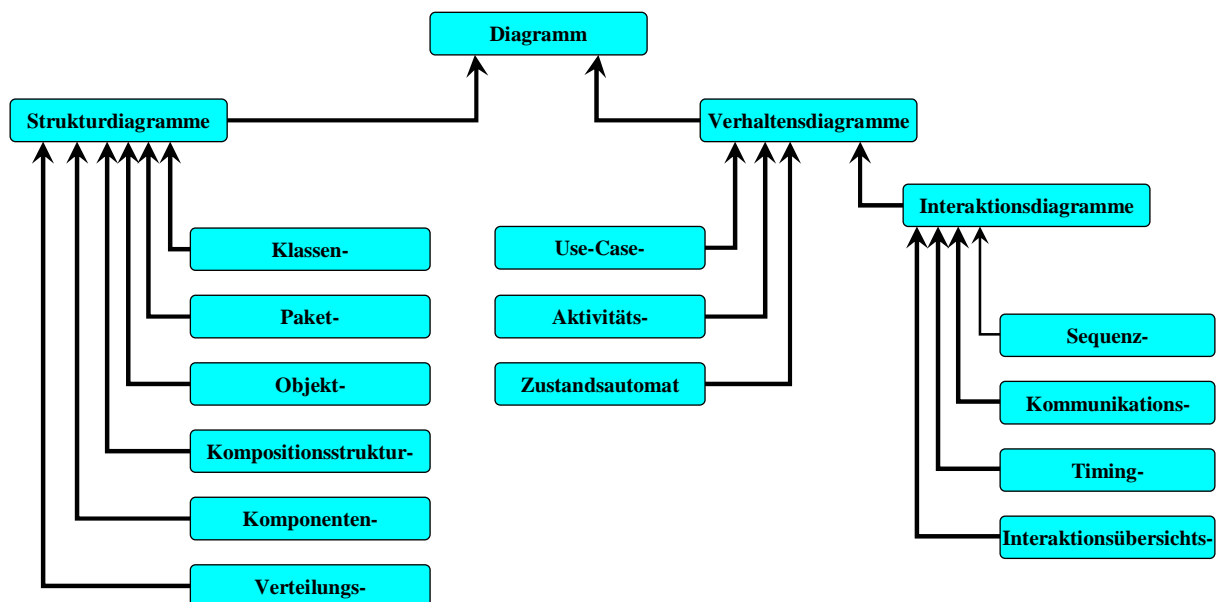
Die unten stehende Grafik zeigt beispielhaft einen Vergleich eines ERM Modells (links) zur Modellierung von Klassen und Relationen mit dem entsprechenden UML Modell (rechts).

¹³ Das Metamodell beschreibt, wie UML selbst aufgebaut ist. In der Regel ist das Metamodell für den Anwender nicht relevant.



Konzept-Katalog - Diagramme der UML 2

Die folgende Grafik gibt eine Übersicht über die Zuordnung der UML Diagramme. Es wird unterschieden in Strukturdiagramme und Verhaltensdiagrammen. Zu den Strukturdiagrammen gehören das Klassendiagramm, das Paketdiagramm, das Objektdiagramm, das Kompositionsstrukturdiagramm, das Komponentendiagramm und das Verteilungsdiagramm. Zu den Verhaltensdiagrammen gehören Use Case Diagramm, das Aktivitätsdiagramm und der Zustandsautomat (State Chart) und die verschiedenen Interaktionsdiagramme. Diese wiederum teilen sich auf in Sequenzdiagramm (Sequence Chart), Kommunikationsdiagramm, Timingdiagramm und Interaktionsübersichtsdiagramm.



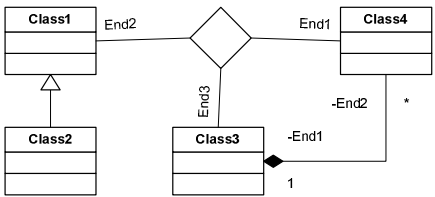
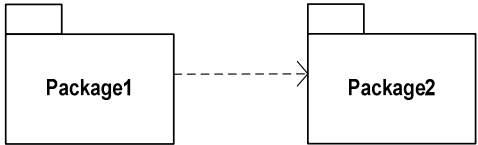
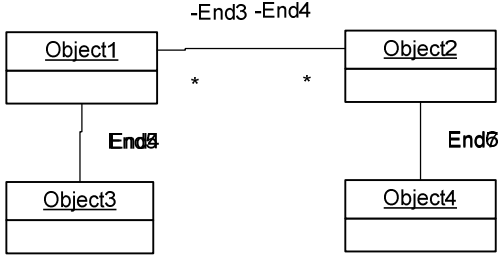
Die Strukturdiagramme beschreiben statische Sichten auf das Modell, die Verhaltensdiagramme modellieren das dynamische Verhalten.

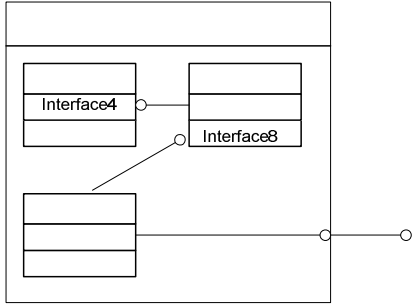
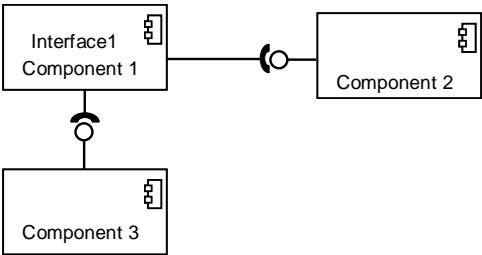
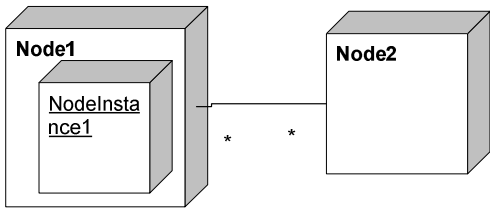
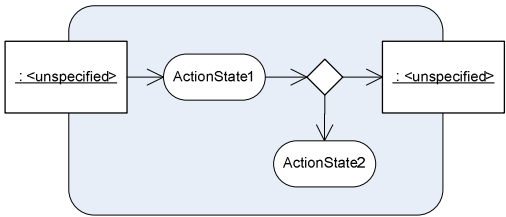
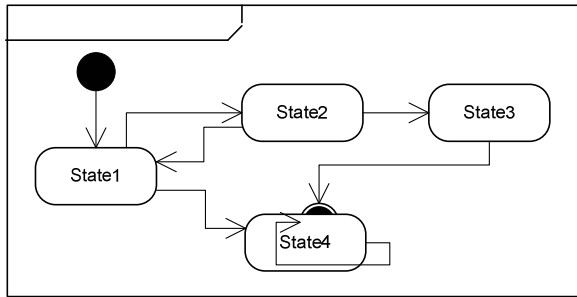
Die sowohl der statischen als auch der dynamische Modellierung zu Grunde liegenden aus der Objektorientierung bekannten Basiskonzepte sind Klasse, Objekt, Attribut, Operation, Vererbung und Polymorphismus. Darauf aufbauend werden in der UML zur Datenmodellierung Assoziationen, Aggregationen und Pakete verwendet. Dynamische

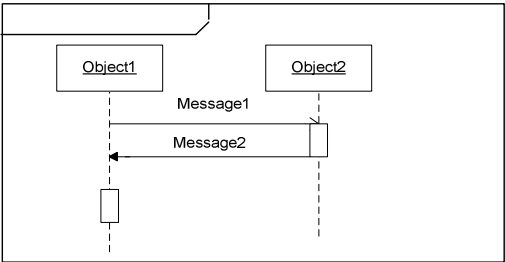
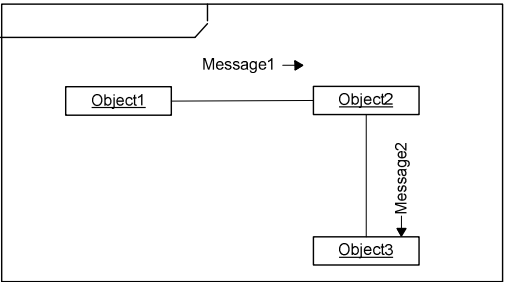
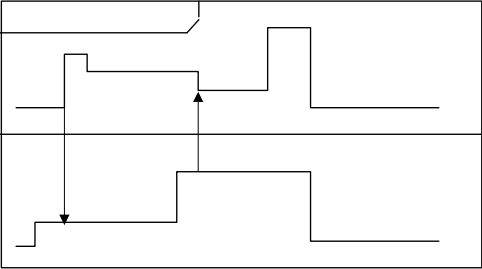
Konzepte der UML bauen auf den Use Case (Anwendungsfall), die Botschaft, Das Szenario und auf den Zustandsautomat auf.

Diagramme der UML 2 und Ihre Anwendung – Eine Übersicht

Die folgende Tabelle gibt einen ersten Überblick über die verschiedenen UML Diagrammtypen und ihre Anwendung (vgl. [6]). In den darauf folgenden Kapiteln werden die wichtigsten Diagramme im Detail näher besprochen.

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Klassendiagramm 	Aus welchen Klassen besteht mein System und wie stehen diese untereinander in Beziehungen	Beschreibt die statische Struktur des zu entwerfenden oder abzubildenden Systems. Enthält alle relevanten Strukturzusammenhänge und Datentypen. Bildet die Brücke zu den dynamischen Diagrammen
Paketdiagramm 	Wie kann ich mein Modell so schneiden, dass ich den besten Überblick bewahre?	Organisiert das Systemmodell in größeren Einheiten durch logische Zusammenfassung von Modellelementen. Modellierung von Abhängigkeiten
Objektdiagramm 	Wie sieht das Innenleben einer Klasse, einer Komponente, eines Systemteils aus ?	Ideal für die Top-down-Modellierung des Systems. Mittleres Detailniveau, zeigt Teile eines "Gesamtelements" und deren Mengenverhältnisse
Kompositionsstrukturdiagramm	Wie sieht das Innenleben einer Klasse, einer Komponente, eines Systemteils aus?	Ideal für die Top-down-Modellierung des Systems. Mittleres Detailniveau, zeigt Teile eines "Gesamtelements" und deren Mengenverhältnisse

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
		
Komponentendiagramm 	Wie werden meine Klassen zu wieder verwendbaren, verwaltbaren Komponenten zusammengefasst und wie stehen diese miteinander in Beziehung?	Zeigt Organisation und Abhängigkeiten einzelner technischer Systemkomponenten. Modellierung angebotener und benötigter Schnittstellen möglich.
Verteilungsdiagramm 	Wie sieht das Einsatzumfeld (Hardware, Server, Datenbanken, ...) des Systems aus? Wie werden die Komponenten zur Laufzeit wohin verteilt?	Zeigt das Laufzeitumfeld des Systems mit den "greifbaren" Systemteilen (meist Hardware). Hohes Abstraktionsniveau, kaum Notationselemente.
Use Case Diagramm 	Was leistet mein System für seine Umwelt (Nachbarsysteme, Stakeholder)?	Sehr detaillierte Visualisierung von Abläufen mit Bedingungen, Schleifen, Verzweigungen. Parallelisierung und Synchronisation möglich.
Zustandsautomat 	Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ... bei welchen Ereignissen annehmen?	Präzise Abbildung eines Zustandsmodells mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen. Schachtelung ist möglich.

Diagrammtyp	Diese zentrale Frage beantwortet das Diagramm	Stärken
Sequenzdiagramm 	Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?	Stellt den zeitlichen Ablauf des Informationsaustausches zwischen Kommunikationspartnern dar. Schachtelung und Flusssteuerung (Bedingungen, Schleifen, Verzweigungen) möglich
Kommunikationsdiagramm 	Wer kommuniziert mit wem? Wer "arbeitet" im System zusammen?	Stellt den Informationsaustausch zwischen Kommunikationspartnern dar. Überblick steht im Vordergrund (Details und zeitliche Abfolge weniger wichtig)
Timingdiagramm 	Wann befinden sich verschiedene Interaktionspartner in welchem Zustand?	Visualisiert das exakte zeitliche Verhalten von Klassen, Schnittstellen, ... Geeignet für Detailbetrachtungen, bei denen es sehr wichtig ist, dass ein Ereignis zum richtigen Zeitpunkt eintritt.

4.6. Statische Modellierung in der Objektorientierten Analyse und Klassendiagramm

Dieses Kapitel erläutert die wesentlichen Prinzipien der statischen Modellierung in der objektorientierten Analyse (OOA) ausgehend von den Requirements.

Ziel der objektorientierten Analyse (OOA) ist es, geeignete Klassen zur Modellierung des Problems und ihre Beziehungen untereinander zu identifizieren.

Dies erfolgt zunächst durch statische Modellierung (Klassendiagramm) und dynamische Modellierung (Sequenzdiagramm, Kommunikationsdiagramm). Die OOA entspricht im Wesentlichen der Zielsetzung der Spezifikation und zum Teil des Grobentwurfs. Das heißt die OOA beantwortet die Frage „Was tut das System?“ und definiert das Grundgerüst des Systems.

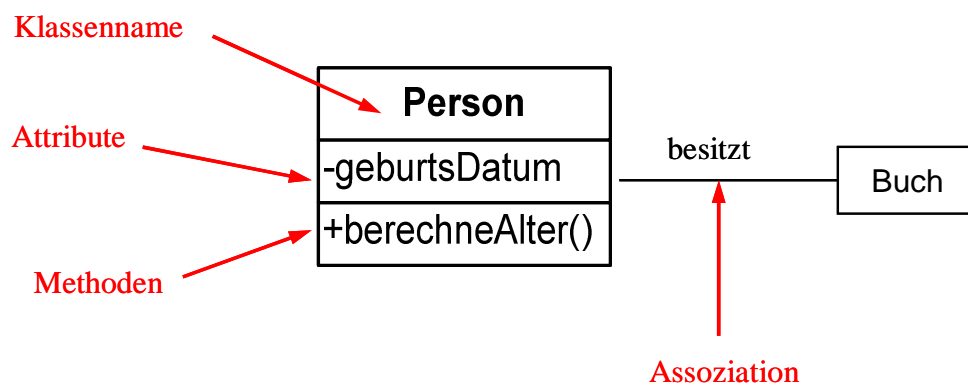
Die Realisierung steht zunächst nicht im Vordergrund, d.h. die Frage „Wie wird das Verhalten im Einzelnen realisiert?“ wird zunächst nicht beantwortet.

Zunächst beschäftigen wir uns mit dem statischen Systemmodell. Ziel ist die Modellierung der statischen Systemstruktur und Modularisieren des Systems. Dies bedeutet die Identifizierung der Klassen und ihrer Beziehungen.

tifikation der relevanten Klassen, die Beschreibung ihrer statischen Eigenschaften (d.h. die Attribute der Klassen), die Analyse der Verantwortlichkeiten der Klassen, die Beschreibung der Beziehungen (Assoziationen) zwischen Klassen, die Definition der Eigenschaften der Datenspeicherung und Datenpersistenz. Das dazu verwendete UML Beschreibungsmittel ist das Klassendiagramm. Das Klassendiagramm wird während der Analysephase iterativ erstellt. Das heißt, dass schrittweise Kandidaten für Klassen identifiziert, deren Verantwortlichkeiten festgelegt und deren Beziehungen untereinander (Assoziationen) definiert werden. Zusätzlich werden die Attribute der Klasse festgelegt.

Das Klassendiagramm wird in der nachfolgenden Phase (objektorientierter Entwurf: OOD) verfeinert und ergänzt, insbesondere durch Vervollständigung der Klassen, Beschreibung der Methoden und Präzisierung der Schnittstellen. OOD ist Inhalt des nächsten Kapitels.

Zunächst wird das Klassendiagramm eingeführt. Klassen werden durch Rechtecke dargestellt. Die Klassen sind zumindest mit dem Namen der Klasse beschriftet (Klassen ohne Namen machen eher weniger Sinn). Optional können Attribute und Methoden der Klasse nach dem Namen aufgeführt werden. Assoziationen zwischen Klassen werden durch Kanten (Linien) zwischen den Klassen, die in einer Beziehung zueinander stehen, dargestellt. Optional können Assoziationen mit Namen versehen werden. Siehe dazu auch das unten stehende Diagramm der Klasse Person mit dem Attribut Geburtsdatum und der Methode berechneAlter. Die Klasse Person steht in einer Beziehung (Assoziation) zur Klasse Buch mit dem Namen „besitzt“.



Die Syntax für Attributdeklaration erlaubt unter anderem folgende Definitionen: Sichtbarkeit (optional) Name, Typ (optional), Multiplizität (optional), einen Vorgabewert. Die Syntax für Methoden (Operationen) enthält unter anderem einen Namen, Sichtbarkeit (optional), Rückgabetyt und eine Parameterliste (optional), eine optionale Kennzeichnung, ob die Methode lesend oder schreibend auf Attribute zugreift. Ähnlich wie Attribute können auch die Parameter einer Methode definiert werden.

Die OOD Analysephase gliedert sich in statische Analyse und eine dynamische Analyse. Die statische Analyse führt in folgenden Schritten zu einem statischen Modell:

1. Klassenkandidaten identifizieren
2. Assoziationen identifizieren
3. Attribute spezifizieren

Zunächst beschäftigen wir uns mit der Identifizierung der Klassenkandidaten. Wenn man sich in einem Anwendungsbereich auskennt, wird man schnell eine ganze Reihe von Klassenkandidaten nennen können. Anders und gegebenenfalls sehr viel schwieriger wird es aber sein, exakt zu begründen, warum man gerade auf diese kommt. Folgende beispielhafte Vorgehensweise zeigt, wie man sinnvolle Klassen- bzw. Attributskandidaten erkennen kann. Erste Kandidaten für Klassen ergeben sich aus den Anforderungen, in dem man alle Substantive betrachtet. Dies soll an folgendem Beispiel erläutert werden:

Gegeben sei folgender Auszug aus einer Produktbeschreibung (Requirements):

- Eine Bibliothek besitzt Exemplare von Büchern, Zeitschriften etc.
- Die meisten Buchexemplare können ausgeliehen werden. Nicht ausgeliehen werden können so genannte Präsenzexemplare von Büchern. Zeitschriften können ebenfalls nicht ausgeliehen werden.
- Sind von einem Buch alle Exemplare ausgeliehen, so kann es vorgemerkt werden. Wird ein Exemplar eines vorgemerkten Buches zurückgegeben, so wird der erste Vorbesteller benachrichtigt. Holt er das Buch nicht binnen einer Woche ab, so verfällt die Vormerkung.
- Wird die Leihfrist überschritten, so wird der Benutzer gemahnt. Er wird solange von der Ausleihe ausgeschlossen, bis das Exemplar zurückgegeben wird.

Betrachtet werden zunächst die Substantive, die in der Beschreibung und in den Anwendungsfällen vorkommen (unten rot markiert).

- Eine Bibliothek besitzt Exemplare von Büchern, Zeitschriften etc.
- Die meisten Buchexemplare können ausgeliehen werden. Nicht ausgeliehen werden können so genannte Präsenzexemplare von Büchern. Zeitschriften können ebenfalls nicht ausgeliehen werden.
- Sind von einem Buch alle Exemplare ausgeliehen, so kann es vorgemerkt werden. Wird ein Exemplar eines vorgemerkten Buches zurückgegeben, so wird der erste Vorbesteller benachrichtigt. Holt er das Buch nicht binnen einer Woche ab, so verfällt die Vormerkung.
- Wird die Leihfrist überschritten, so wird der Benutzer gemahnt. Er wird solange von der Ausleihe ausgeschlossen, bis das Exemplar zurückgegeben wird.

Die Auswahl der Substantive gibt Anlass zu folgenden Klassenkandidaten (in alphabetischer Reihenfolge):

Ausleihe, Benutzer, Bibliothek, Buch, Buchexemplar, Exemplar, Leihfrist, Präsenzexemplar, Vorbesteller, Vormerkung, Woche, Zeitschrift

Diese Kandidaten müssen nun daraufhin untersucht werden, ob und welche Rolle sie im Anwendungsbereich spielen. Auf dieser Basis kann dann entschieden werden, ob sich Klassen aus ihnen ergeben. Dies kann im nächsten Schritt durch Ordnen nach Zusammengehörigkeit (ZG) und Kennzeichnung von Konstanten erfolgen mit folgendem Ergebnis:

ZG1: Buch, Buchexemplar, Exemplar, Präsenzexemplar, Zeitschrift

ZG2: Vormerkung

ZG3: Ausleihe, Leihfrist (Konstante), Woche (Konstante)

ZG4: Benutzer, Vorbesteller

ZG5: Bibliothek (System selbst ist keine Klasse)

Nach Entfernen der Konstanten, die nicht als Klassen modelliert werden und des Gesamtsystems, das hier ebenfalls nicht als Klasse modelliert wird, bleiben übrig:

ZG1: Buch, Buchexemplar, Exemplar, Präsenzexemplar, Zeitschrift

ZG2: Vormerkung

ZG3: Ausleihe

ZG4: Benutzer, Vorbesteller

Aufgrund der Problembeschreibung sind Zeitschriften spezielle Präsenzexemplare, die keine eigene Klasse erfordern. Des Weiteren ist Buchexemplar ein Synonym von Exemplar und Vorbesteller sind Benutzer in einer besonderen Rolle. Das bedeutet, dass diese aus der Liste ebenfalls entfernt werden können.

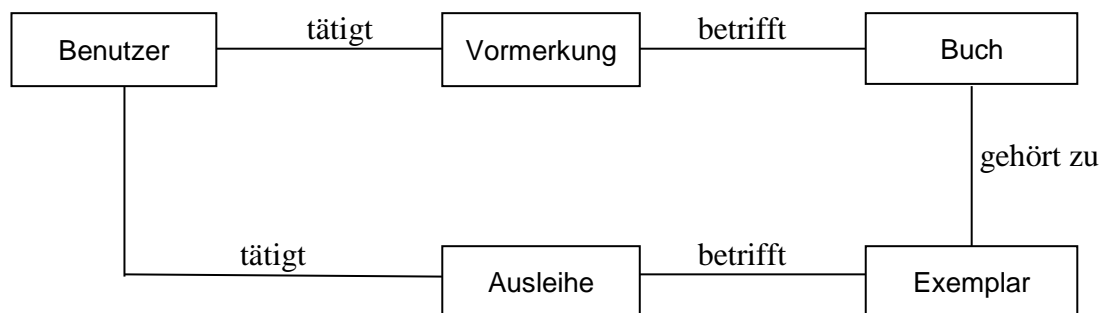
Parallel zur Klassenidentifikation erfolgt die Ermittlung der Beziehungen zwischen den bereits identifizierten Klassen. Kandidaten dafür lassen sich ebenfalls anhand der

Anforderungsszenarien herleiten, z.B. im Zusammenhang mit Aktivitäten und Datenabhängigkeiten (z.B. zwischen Buch und Exemplar: zu einem Buch gehören i.A. mehrere Exemplare, die gemeinsam auf Informationen des zugehörigen Buch-Objekts verweisen, wie etwa Angaben zu Autor und Titel). Dabei sind zunächst genauere Angaben, etwa zur Multiplizität bzw. zur Art der betrachteten Assoziation, nicht erforderlich (werden später im Objektorientierten Design ergänzt).

Bei der Herleitung von Beziehungen aus Aktivitäten wird folgendermaßen vorgegangen: Eine aus den Anforderungen zu vermutende Interaktion zwischen zwei Klassenkandidaten deutet bereits auf eine mögliche Assoziation zwischen beiden Klassen hin:

- Der Benutzer tätigt (persönlich) die Ausleihe eines Buchexemplars (also besteht eine Assoziation zwischen Benutzer und Ausleihe).
- Die Ausleihe betrifft ein Exemplar (ergibt eine Assoziation zwischen Ausleihe und Exemplar).
- Der Benutzer beantragt die Vormerkung für ein Buch (stellt eine Assoziation zwischen Benutzer und Vormerkung dar).
- Die Vormerkung betrifft ein Buch (bildet eine Assoziation zwischen Vormerkung und Buch).

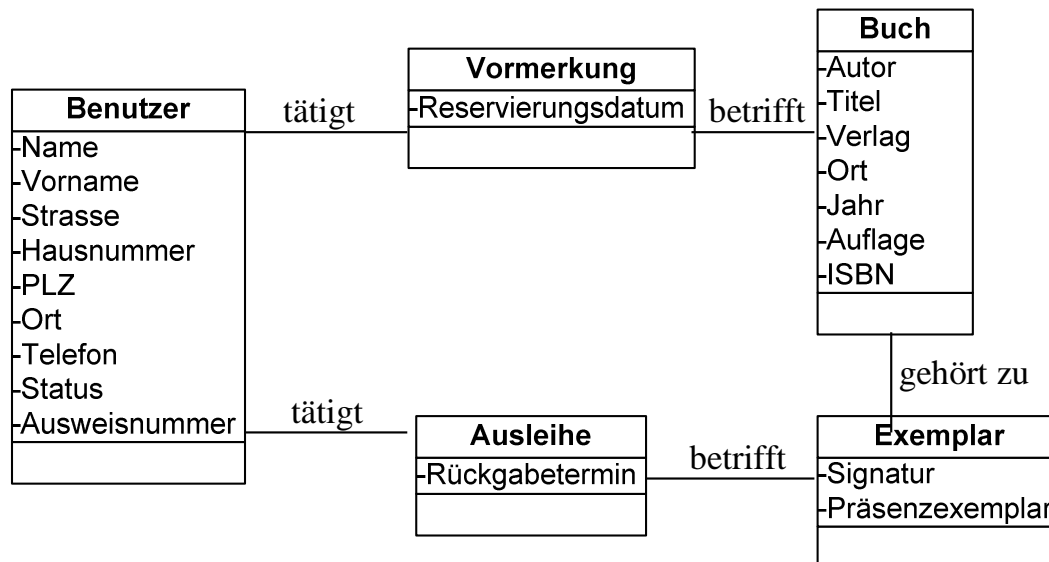
Die oben identifizierten Klassen und Beziehungen führen zu folgendem ersten Klassendiagramm:



Im nächsten Schritt werden die Attribute spezifiziert. Das heisst, zu den identifizierten Klassen können zusätzliche Eigenschaften (Attribute) festgelegt werden. Soweit zu diesem Zeitpunkt möglich und sinnvoll, werden für Attribute die folgenden Eigenschaften spezifiziert:

- Name des Attributs
- Beschreibung des Attributs
- Sichtbarkeit
- Typ des Attributs

Im Beispiel Bibliothek bietet sich etwa zur Klasse Exemplar das Attribut "Präsenzexemplar" (vom Typ Boolean) an. Die identifizierten Klassen werden über die Anforderung hinaus mit sinnvollen Zusatzinformationen ergänzt. Die Sichtbarkeit wird bei den angegebenen Attributen in aller Regel privat sein. Insgesamt führt das zum erweiterten, unten dargestellten Klassendiagramm. Damit ist die statische Modellierung zunächst beendet.



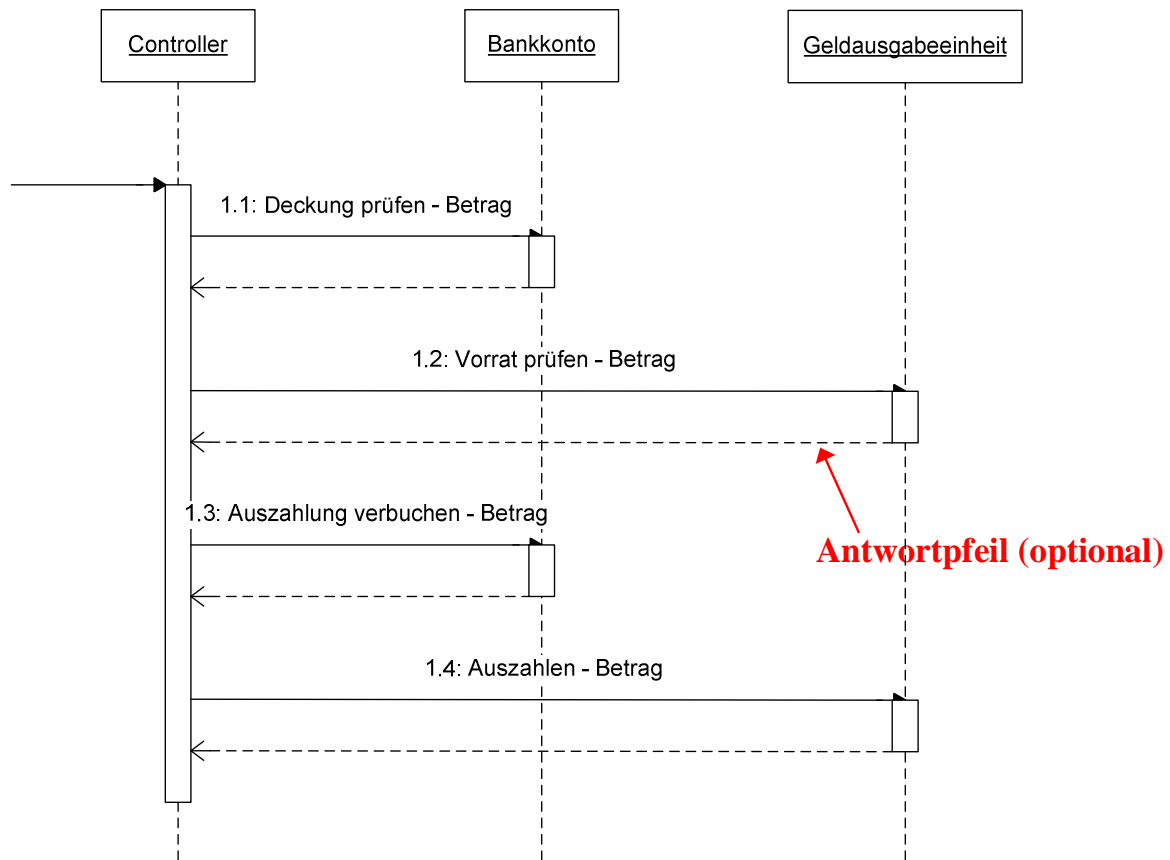
4.7. Dynamische Modellierung

Die dynamische Modellierung beschreibt die Abläufe, die in der Interaktion des zu modellierenden Systems mit seiner Umwelt stattfinden. Dazu werden externen Szenarien erstellt. Diese Szenarien sind Verfeinerungen der Anwendungsfälle (Use Cases), die in Form von Interaktionsdiagrammen dokumentiert werden. Interaktionen können ihrerseits anhand einer der beiden folgenden, äquivalenten Diagrammarten Sequenzdiagramm und Kommunikationsdiagramm dargestellt werden. Szenarien tragen im Wesentlichen dazu bei den Fluss der Botschaften durch das System zu definieren. Im Rahmen der Analysephase stehen zunächst externe Szenarien im Vordergrund, also Interaktionen zwischen dem Software-System und seiner Umgebung. Implementierungsdetails werden nicht berücksichtigt. Erst der im Feinentwurf kommen diese zum Vorschein.

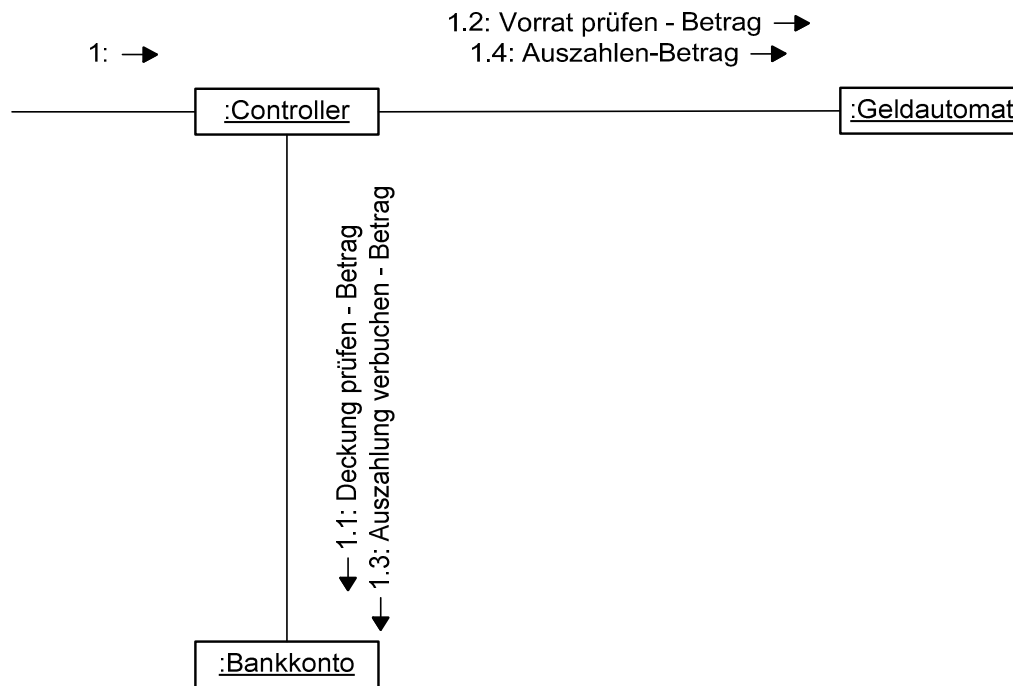
Zur Erstellung der externen Szenarien werden aus jedem Anwendungsfall oft mehrere Szenarien abgeleitet. Dabei führen Variationen von Anwendungsfällen zu unterschiedlichen Szenarien. Gegebenenfalls sind Standardausführungen und Alternativen zu berücksichtigen. Primäre Szenarien stellen die fundamentalen Funktionen des Systems dar. Sekundäre Szenarien präsentieren Variationen primärer Szenarien. Sie beschreiben Ausnahmesituationen und enthalten die weniger oft verwendeten Funktionen.

Zur Darstellung von Szenarien werden Sequenzdiagramme verwendet. Diese zeigen die miteinander in Szenarien vorkommenden und miteinander kommunizierenden Objekte. Dabei werden die an dem dargestellten Szenario beteiligten Objekte durch das übliche UML-Objektsymbol (Rechteck mit Beschriftung) repräsentiert. Zu jedem beteiligten Objekt gibt es eine gestrichelte „Lebenslinie“ (Lifeline), die vom Objektsymbol vertikal nach unten verläuft. Eine virtuelle Zeitachse bestimmt die zeitliche Ordnung der im Sequenzdiagramm dargestellten Ereignisse. Sie verläuft parallel zu den Lebenslinien der Objekte, vertikal von oben nach unten. Jede Kommunikation zwischen zwei Objekten wird durch eine Nachricht modelliert. Eine Nachricht wird dabei durch eine gerichtete Kante zwischen den Lebenslinien von Sender- und Empfängerobjekt repräsentiert. Das unten gezeigte Sequenzdiagramm zeigt den Vorgang des Abhebens aus einem Geldautomat. Die beteiligten Objekte sind ein Controller, der den Vorgang steuert, das Bankkonto des Kunden und die Geldausgabeeinheit des Automaten. Der initiale Pfeil ganz links kommt aus dem „Nichts“ und stellt den Wunsch des Bankkunden dar, einen bestimmten Betrag abzuheben. (Der Kunde könnte auch als Objekt in dem Diagramm modelliert werden). Als erste Nachricht fragt der Controller die Deckung des Kontos an. Diese wird zurückgemeldet mit der Annahme, dass genug Geld auf dem Konto vorhanden ist. Danach prüft der Controller die Verfügbarkeit der Summe in der Ausgabeeinheit. Auch dies wird positiv quittiert (hier nicht explizit dargestellt). Danach wird

der Auszahlungsbetrag vom Konto abgebogen und die Ausgabeeinheit angewiesen den Betrag auszuzahlen. Wie im Diagramm erwähnt ist der Antwortpfeil optional.



Weitgehend semantisch äquivalent zum Sequenzdiagramm ist das Kommunikationsdiagramm. Hier werden die beteiligten Objekte dargestellt, ggf. mit Rollenangaben zur Verdeutlichung der Bedeutung des Objektes innerhalb des jeweiligen Szenarios. Beziehungen zwischen den beteiligten Objekten werden durch Verknüpfungskanten (Pfeile) dargestellt. Der Nachrichtenaustausch (Interaktion) zwischen den beteiligten Objekten erfolgt entlang der Verknüpfungskanten. Die zeitliche Reihenfolge der Nachrichten wird durch ein Nummerierungsschema (z.B. 1, 2, 2.1 2.2, 3, 4) ausgedrückt. Das Diagramm unten zeigt denselben Vorgang (Abheben am Geldautomat) wie oben im Sequenzdiagramm dargestellt.



Zum Schluss dieses Abschnitts seien die wesentliche Gemeinsamkeiten und Unterschiede von Kommunikationsdiagramm und Sequenzdiagramm zusammengefasst. Das Sequenzdiagramm und das Kommunikationsdiagramm sind zwei semantisch äquivalente Arten von Interaktionsdiagrammen. Das Sequenzdiagramm betont den temporalen Ablauf und die zeitliche Reihenfolge des Nachrichtenaustauschs. Das Kommunikationsdiagramm betont die (statischen) Verknüpfungen zwischen den interagierenden Objekten.

4.8. Objektorientiertes Design

Das Ziel des Objektorientierten Designs (OOD) ist es, den bereits bestehenden Grobentwurf (Architektur gemäß dem OOA-Modell) durch Vervollständigung der Klassendiagramme (insbes. der Methoden), Vervollständigung der Interaktionsdiagramme und bei Bedarf die Modellierung des internen Verhaltens einer Klasse zum Feinentwurf zu detaillieren.

Das OOD lässt sich (ähnlich wie die OOA) in eine statische Modellierungsphase und eine dynamische Modellierungsphase unterteilen. Beide Phasen erfolgen meistens iterativ, überlappend und ergänzen sich gegenseitig. Zusätzlich wird in der Architekturmodellierungs-Phase die logische und physikalische Struktur des Systems festgelegt.

Objektorientierte Architekturmodellierung

Bisher fehlt noch eine Modularisierung des Systems, um festgelegte Aufgaben Modulen (Komponenten) zuordnen zu können, die Skalierbarkeit (durch Entfernen und Hinzufügen von Modulen) des Systems zu unterstützen und die Wartbarkeit zu verbessern (indem einzelne Module ausgetauscht bzw. geändert werden können). In der Objekt-orientierten Architekturmodellierung unterscheidet man zwischen zwei Teilaspekten:

- die Logische Systemarchitektur, d.h. welche Klassen werden zu Paketen und Komponenten zusammengefasst
- die Physikalische Systemarchitektur: wie werden die Komponenten auf mehrere Recheneinheiten verteilt und wie kommunizieren die Komponenten untereinander

Für die Logische Sicht werden Paketdiagramme und Komponentendiagramme verwendet. Paketdiagramme zeigen die Verteilung der Klassen auf Pakete, d. h. auf Mengen logisch zusammenhängender Klassen mit eindeutiger Namensgebung (Namensräume).

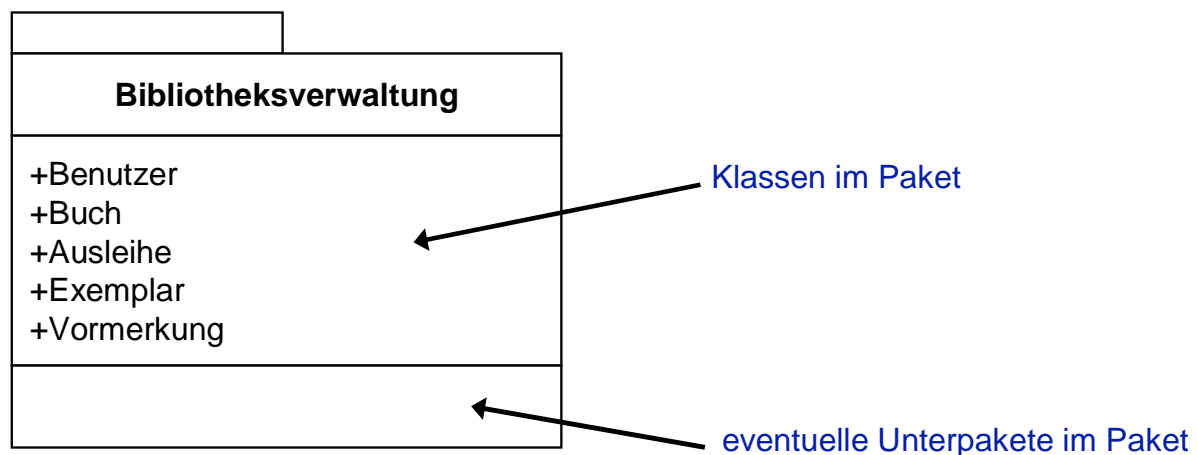
Komponentendiagramme zeigen die Verteilung der Pakete auf in sich abgeschlossene Komponenten.

Die Physikalische Sicht wird durch Einsatzdiagramme modelliert. Diese zeigen welche Komponenten auf welchen Rechnern installiert werden. Die Verwendung von Einsatzdiagrammen ist nur notwendig bzw. sinnvoll wenn eine Verteilung der Software auf verschiedene Rechner oder Controller vorgenommen wird (.z.B. bei Verteilten Systemen in Rechnernetzen oder in verteilten Embedded Systemen).

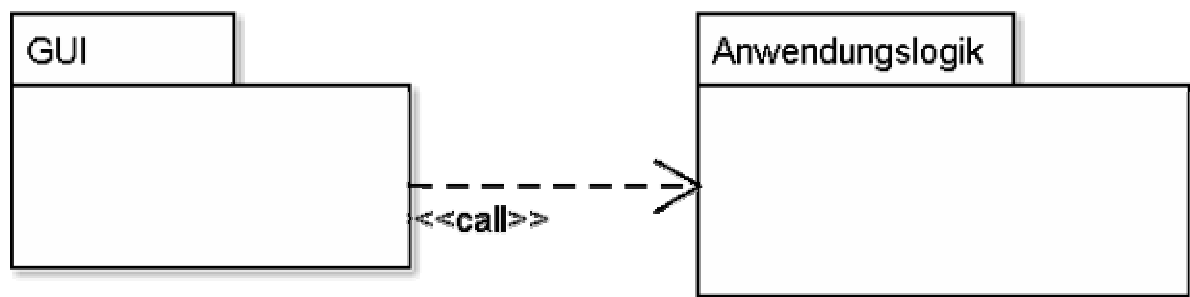
Paketdiagramm

Paketdiagramme zeigen, welche Klassen bzw. Unterpakete sich innerhalb eines Pakets befinden und Beziehung der Pakete zueinander.

Im schon oben verwendeten Beispiel Bibliotheksverwaltung könnte ein Paketdiagramm folgendermaßen aussehen:

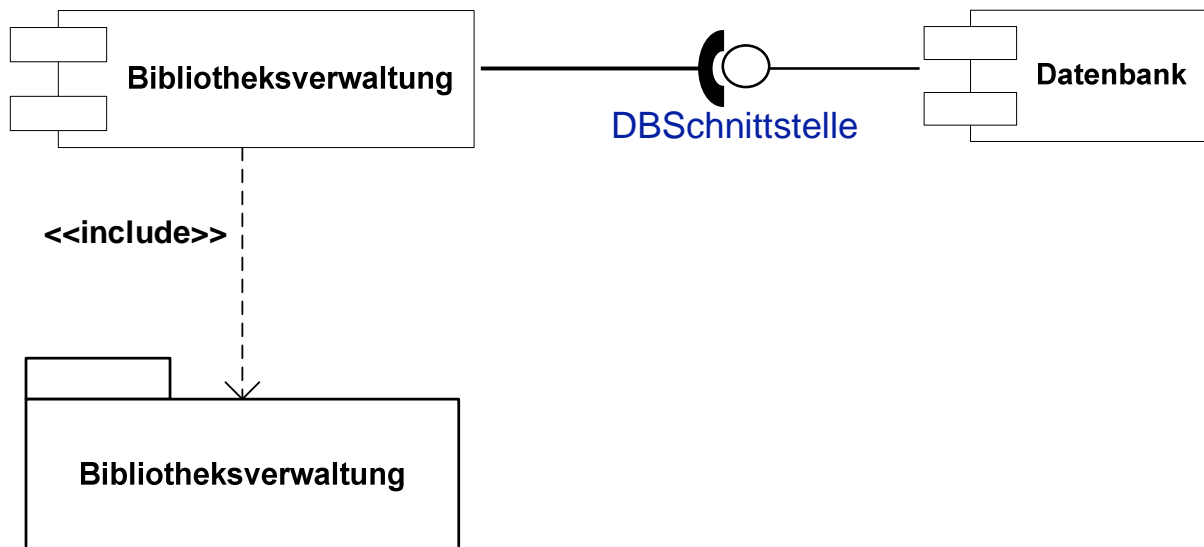


Das Paket Bibliotheksverwaltung verwendet die Klassen Benutzer, Buch, Ausleihe usw., jedoch keine Unterpakete. Die Beziehungen zwischen den Paketen werden als so genannte Abhängigkeiten (gestrichelter Pfeil) notiert. Im Beispiel unten ruft die Benutzeroberfläche (GUI) die Anwendungslogik auf.



Komponentendiagramm

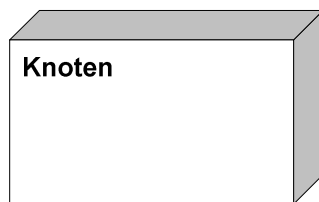
Das Komponentendiagramm zeigt Komponenten sowie deren Beziehungen untereinander. Komponenten werden durch eine oder mehrere Klassen realisiert. Komponenten benötigen Schnittstellen und bieten Schnittstellen an. Im Beispiel unten wird die Schnittstelle DBSchnittstelle von der Komponente Datenbank angeboten („Lollipop-Notation“) und von der Bibliotheksverwaltung genutzt.



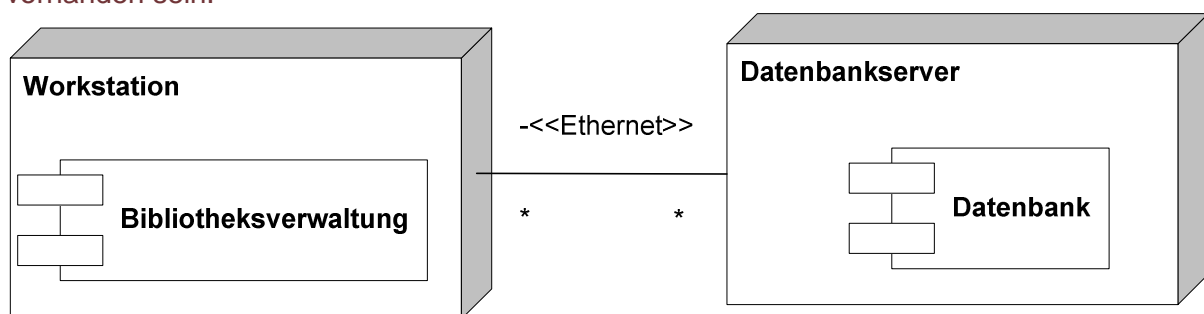
Einsatzdiagramm

Das Einsatzdiagramm zeigt die Knoten (Rechner, Controller) eines Systems, welche Komponenteninstanzen auf diesen installiert sind und die Beziehungen zwischen den Knoten. Das Einsatzdiagramm wird auch Verteilungsdiagramm (Deployment) genannt. Das Ziel ist eine statische Sicht auf das installiertes System zu beschreiben.

Ein Knoten (Instanz) wird folgendermaßen dargestellt:



Beziehungen (z. B. Kommunikationskanäle) können mit Verbindungen (Instanzen von Assoziationen) dargestellt werden. Im Bild unten läuft die Komponente Bibliotheksverwaltung auf einer Workstation und die Datenbank auf einem Datenbankserver. Die Kommunikation erfolgt über Ethernet. Dabei können beliebig viele Kommunikationspartner auf beiden Seiten vorhanden sein.



Statische Modellierung im OOD

Das Ziel der statischen Modellierung im OOD ist es, die in der OOA entwickelten UML-Klassendiagramme zu präzisieren. Dies geschieht auf folgende Weise:

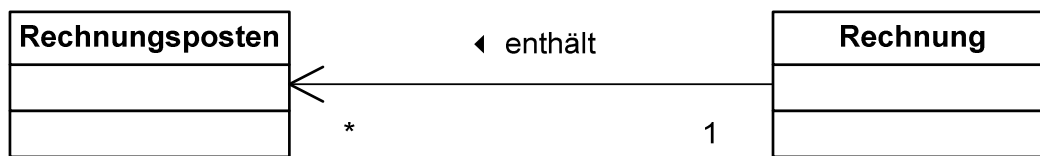
1. die bisher ausgewählten Klassen werden evtl. durch weitere Klassen ergänzt
2. die identifizierten Klassen werden um Operationen und Attribute ergänzt und die z.T. bereits ermittelten Assoziationen präzisiert.

Eigenschaften einer Assoziation

Assoziationen haben folgende Eigenschaften.

- Assoziationen haben einen eindeutigen Namen
- Die Leserichtung einer Assoziation kann durch ein schwarzes Dreieck eindeutig gemacht werden, falls sich aus dem Namen der Assoziation die Leserichtung nicht eindeutig erkennen lässt. Eine Assoziation mit Leserichtung von Klasse A nach Klasse B wird als „Klasse A <Name der Assoziation> Klasse B“ gelesen.
- Assoziationen können eine Benutzungsrichtung, dargestellt durch einen Pfeil besitzen. Eine Assoziation mit Benutzungsrichtung von Klasse A nach Klasse B bedeutet, dass Klasse A ein Attribut enthält, das eine Referenz auf ein Objekt der Klasse B darstellt.

Im Beispiel unten wird die Klasse Rechnung in eine „enthält“ Beziehung gesetzt mit der Klasse Rechnungsposten. Die Benutzungsrichtung ist hier auch die Leserichtung. Beim Verarbeiten einer Rechnung muss auf deren Rechnungsposten zugegriffen werden können. Dies wird durch die gerichtete Assoziation modelliert. Eine Rechnung kann beliebig viele Rechnungsposten enthalten¹⁴.



Assoziationen können ergänzt werden durch Rollennamen. D.h. an beiden Enden einer Assoziation kann zusätzlich die Bedeutung („Rolle“) der Klasse in Bezug auf die genannte Assoziation annotiert werden. Die Rollennamen sind für rekursive Assoziationen Pflicht, ansonsten sollten sie eingesetzt werden, um die Verständlichkeit zu erhöhen. Jede Rolle einer Assoziation hat eine Multiplizität. Diese gibt die erlaubte Anzahl an Objekten (Kardinalität) an, die sich in einer vorgegebenen Rolle an einer Assoziation beteiligen können. Die Kardinalität wird spezifiziert durch einen zugelassenen Zahlenbereich (Untergrenze .. Obergrenze), wobei das *-Symbol eine unbeschränkte Obergrenze symbolisiert. Wird keine Multiplizität explizit angegeben, beträgt diese 1..1. Abkürzend kann das *-Symbol anstelle von 0..* und 1 anstelle von 1..1 verwendet werden.

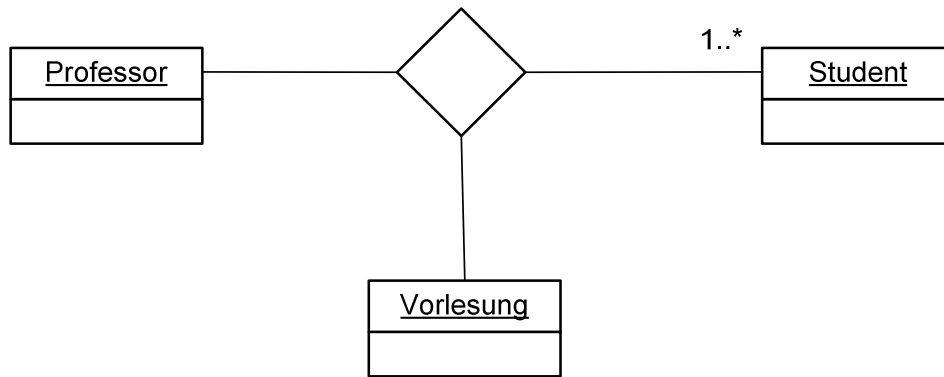
Im Beispiel unten kann ein Benutzer in zwei Rollen mit einer Teilnehmergruppe in Beziehung stehen: Als Mitglied und als Besitzer. Ein Benutzer kann dabei beliebig viele Teilnehmergruppen verwalten (dargestellt durch *). Eine Teilnehmergruppe wird von genau einem Benutzer verwaltet. (dargestellt durch die 1) Ein Benutzer kann Mitglied beliebig vieler, mindestens einer Teilnehmergruppen sein (dargestellt durch 1..*). Eine Teilnehmergruppe besteht aus mindestens einem Mitglied (dargestellt durch 1..*).



Mehrgliedrige Assoziationen

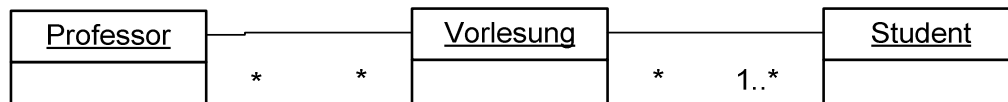
Neben Zweierbeziehungen gibt es noch mehrgliedrige Assoziationen, an denen drei oder mehr Klassen beteiligt sind. Das Beispiel unten (Studenten hören Vorlesung bei einem Professor) stellt eine ternäre (dreigliedrige oder 3-stellige) Assoziation dar.

¹⁴ es könnte auch eine 1..* Beziehung modelliert werden, falls eine Rechnung ohne Rechnungsposten keinen Sinn macht.

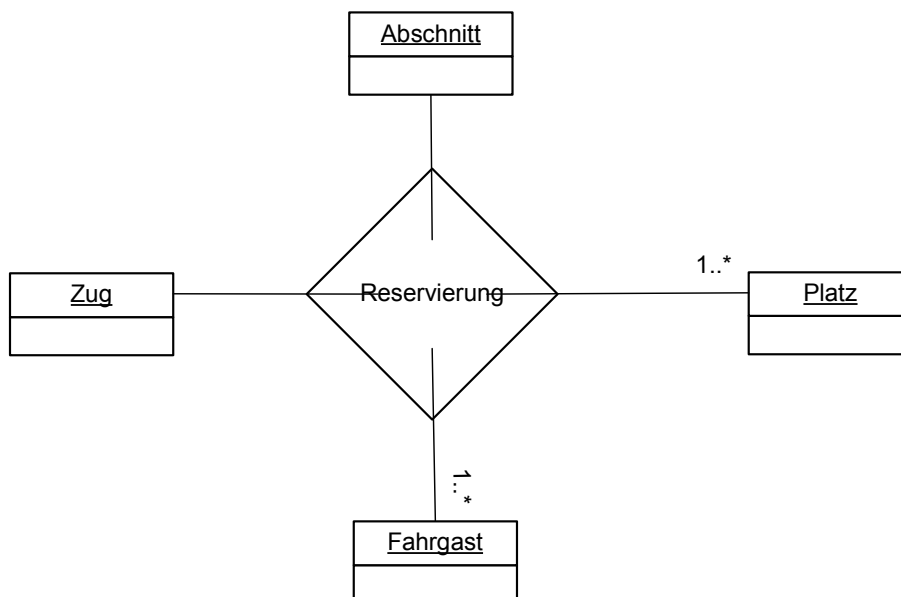


Mehrgliedrige Assoziationen können in den meisten Programmiersprachen nicht direkt realisiert werden. Es sollte daher stets geprüft werden, ob die mehrgliedrige Assoziationen ersetzt werden können durch mehrere binäre Assoziationen oder durch eine Assoziationsklasse

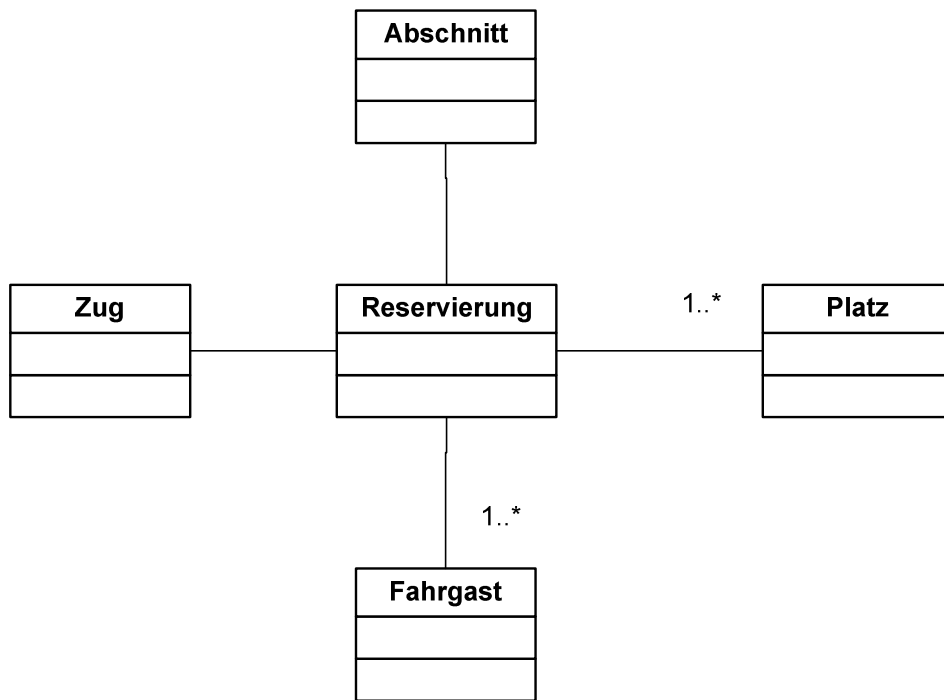
Die Assoziation oben kann auch mittels zweier binärer Assoziationen wie unten ausgedrückt werden:



Als weiteres Beispiel unten für eine mehrgliedrige (hier viergliedrige) Assoziation dient die Assoziation Reservierung..



Assoziationsklasse

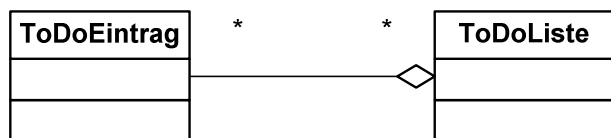


Aggregation

Die Aggregation ist ein Spezialfall der Assoziation. Es handelt sich dabei um eine asymmetrische Beziehung zwischen nichtgleichwertigen Partnern, die unabhängig voneinander existieren können, wobei ein Objekt der einen der beiden Klassen (der sog. Aggregatklasse) eine Menge von Objekten der anderen Klasse darstellt. Die Aggregation wird auch als "Teile-Ganzes-Beziehung" oder "Besteht-aus-Beziehung" bezeichnet.

Aggregationen werden in UML durch eine Raute dargestellt, die jenes Ende der Assoziationskante markiert, das zur Aggregatklasse, also "zum Ganzen" hinführt.

Beispiele für Aggregation: Listen, etc.



Eine Liste besteht aus einer beliebig großen (evtl. auch leeren) Menge von Einträgen. Die Klasse **ToDoListe** ist somit die Aggregatklasse.

Weitere Beispiele für Aggregationen sind :

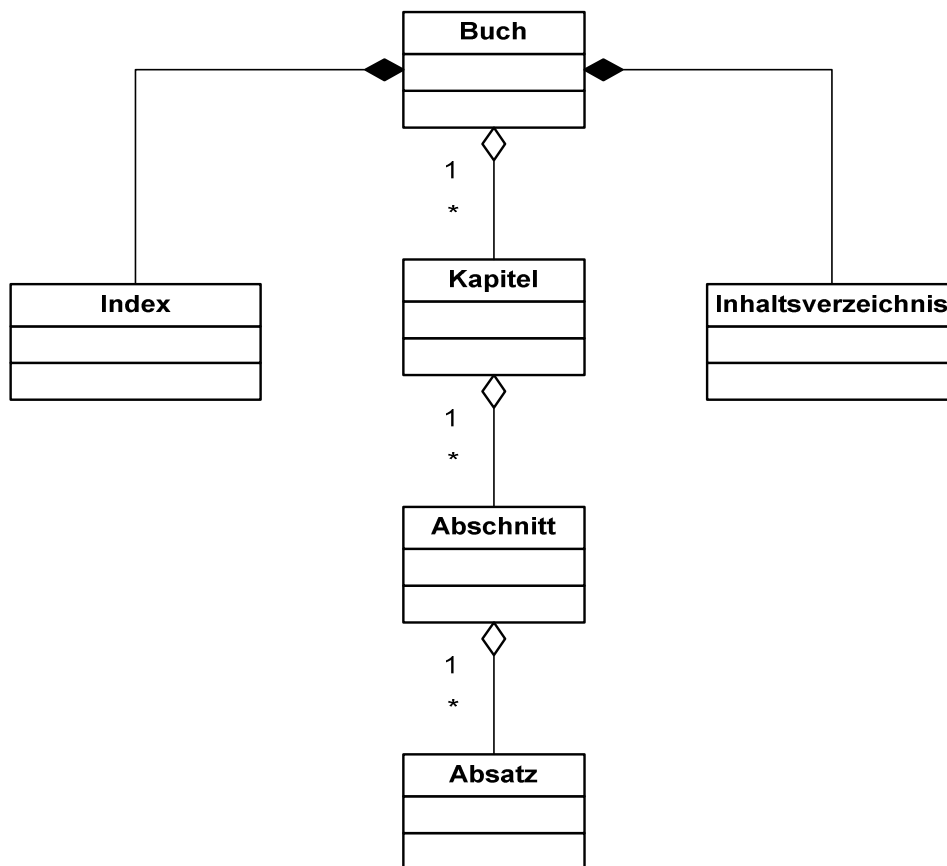
- Ein PKW hat die Teile Motor, Getriebe, Räder, Karosserie,
- Ein Kapitel besteht aus Unterkapiteln
- Ein Unternehmen besteht aus Abteilungen
- Eine Workstation besteht aus Monitor, Rechneinheit, Tastatur, Maus, ...

Komposition

Ein Sonderfall der Aggregation wiederum ist die Komposition. Eine Komposition liegt dann vor, wenn das Einzelteil vom Aggregat existenzabhängig ist. Das heißt, dass das Einzelteil nicht ohne das Aggregat existieren kann¹⁵. Ein Teil kann immer nur von *einem* Aggregat existenzabhängig sein. Kompositionen werden in UML durch eine gefüllte Raute dargestellt,

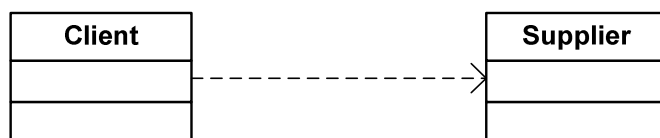
¹⁵ Die Tatsache, dass ein Teil von einem Ganzen existenzabhängig ist, ist in der Regel kontextabhängig. Ein Motor aus Sicht des Autofahrers macht möglicherweise alleine keinen Sinn, sondern eben nur im einem Fahrzeug. Aus Sicht eines Motorenentwicklers kann dies ganz anders aussehen.

die jenes Ende der Assoziationskante markiert, das zur Aggregatklasse, also "zum Ganzen" hinführt.

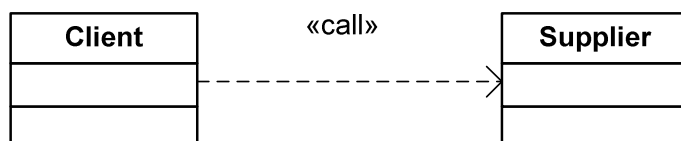


Das Inhaltsverzeichnis und der Index eines Buchs sind ohne das zugehörige Buch meist ohne Bedeutung. Ein Kapitel, Abschnitt oder Absatz hat auch für sich allein einen Inhalt, der ohne das ganze Buch verwendet werden kann, z.B. in einem Zitat.

Abhängigkeiten

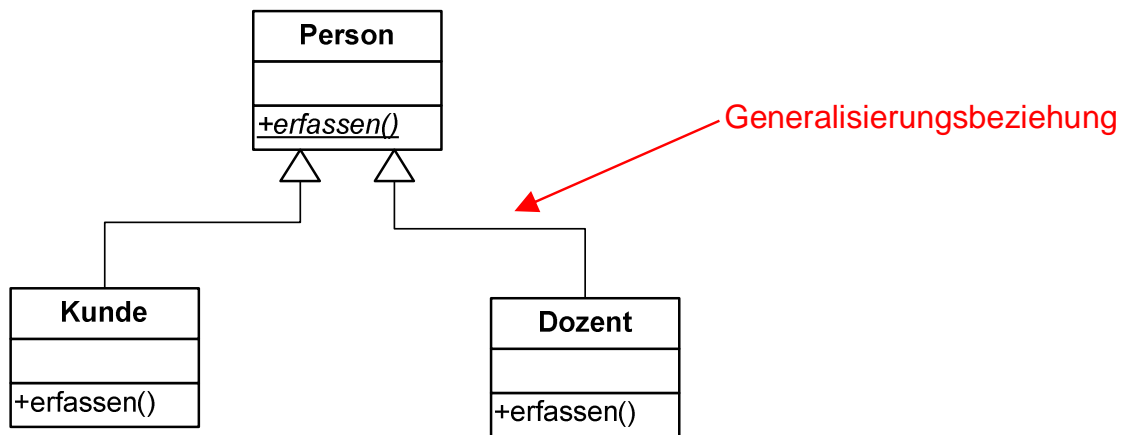


Eine Abhängigkeit zeigt an, dass eine Klasse eine andere benötigt. Dabei sind Abhängigkeiten immer gerichtet. Abhängigkeiten werden in der UML als gestrichelter Pfeil mit einer offenen Spitze dargestellt. Um die Bedeutung der Abhängigkeit näher zu spezifizieren, können so genannte Stereotypen textuell an der betroffenen Abhängigkeitskante in doppelten spitzen Klammern annotiert werden.



Abstrakte Klasse

Eine abstrakte Klasse implementiert nur einen Teil (oder keine) der Operationen, die sie deklariert. Eine abstrakte Klasse kann nicht instanziiert werden, d.h. es können keine Objekte erzeugt werden. Abstrakte Klassen und nichtimplementierte Operationen werden im Klassendiagramm kursiv dargestellt. Die nicht implementierten Operationen müssen über eine Unterklasse implementiert werden. Sobald die Unterklasse alle Methoden implementiert sind, können Objekte instanziiert werden.

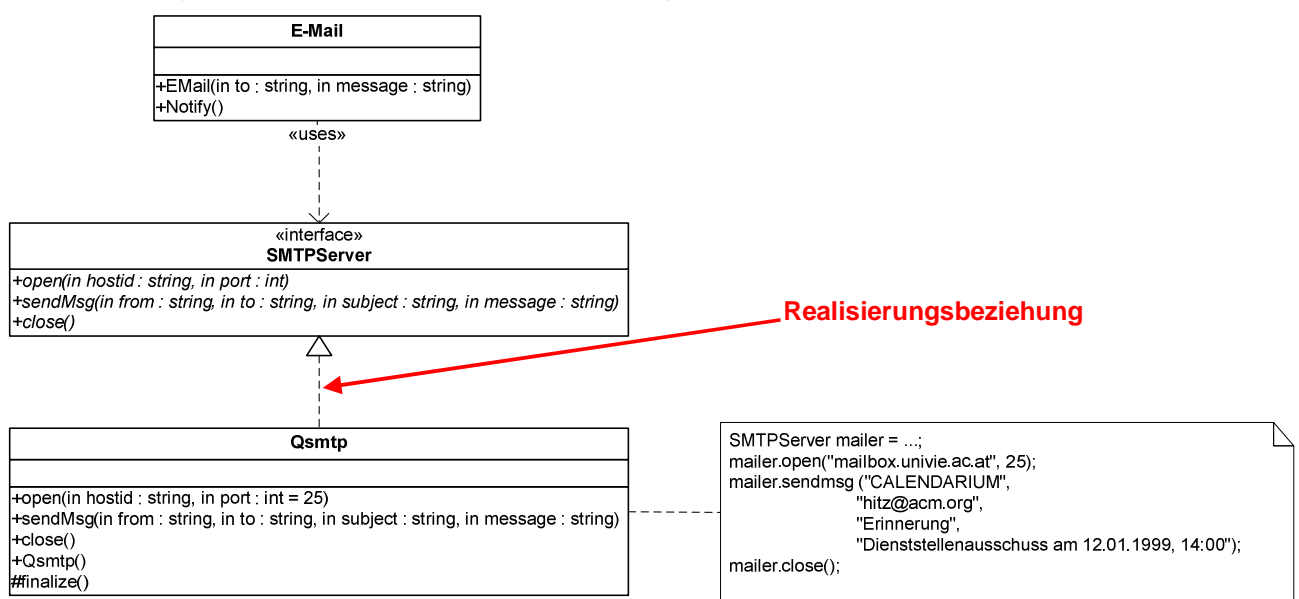


Interface

Erst im OOD sind die Klassen um noch fehlende Operationen bzw. Details bereits identifizierter Operationen ergänzt worden. Damit können nun die Schnittstellen im Detail beschrieben werden. Zur detaillierten Beschreibung von Schnittstellen werden typischerweise besondere abstrakte Klassen (sog. Interfaces) verwendet, die durch den Stereotyp «interface» gekennzeichnet werden. Ein Interface ist eine abstrakte Klasse (also nicht instanziiierbar) mit ohne implementierte Methode und ohne Attributen. Sie enthält lediglich eine Liste von Operationsdeklarationen. Diese werden als Signatur bezeichnet.

Beispiel für Interface: Versenden von E-Mails

Im Beispiel unten wird eine Interfaceklasse SMTPServer deklariert. Diese hat Operationen zum Öffnen und Schließen von Verbindungen und zum Senden einer Nachricht. Realisiert werden die Operationen in einer Unterklasse Qsmtp.



Zusicherungen

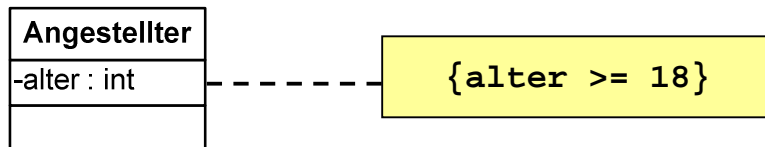
Zusicherungen sind Prädikate, die Eigenschaften von Klassen formulieren. In UML werden diese mittels der OCL (Object Constraint Language) formuliert und als Kommentare in geschweiften Klammern in den Diagrammen angegeben werden.

Im Beispiel unten wird bei der Klasse Angestellter per Zusicherung festgelegt, dass ein Angestellter immer mindestens 18 Jahre alt ist. Da diese Eigenschaft immer gilt ist es eine Invariante (ausgedrückt durch inv).

```
{context Angestellter inv:
    alter >= 18}
```

Zusicherung dienen zum einen dazu Eigenschaften von Klassen, die nicht direkt im Klassendiagramm ausgedrückt werden können, zu definieren. Die Überprüfung von Zusicherungen kann während der Laufzeit eines Programms für die entsprechende Klasse implementiert werden und ausgeführt werden. Zusätzlich können aus Zusicherungen Testfälle abgeleitet werden.

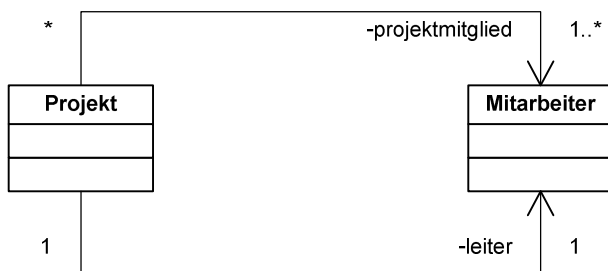
Wenn der Kontext klar ist (etwa durch Zuordnung eines Kommentarfeldes), kann die explizite Angabe des Kontextes im OCL-Ausdruck weggelassen werden:



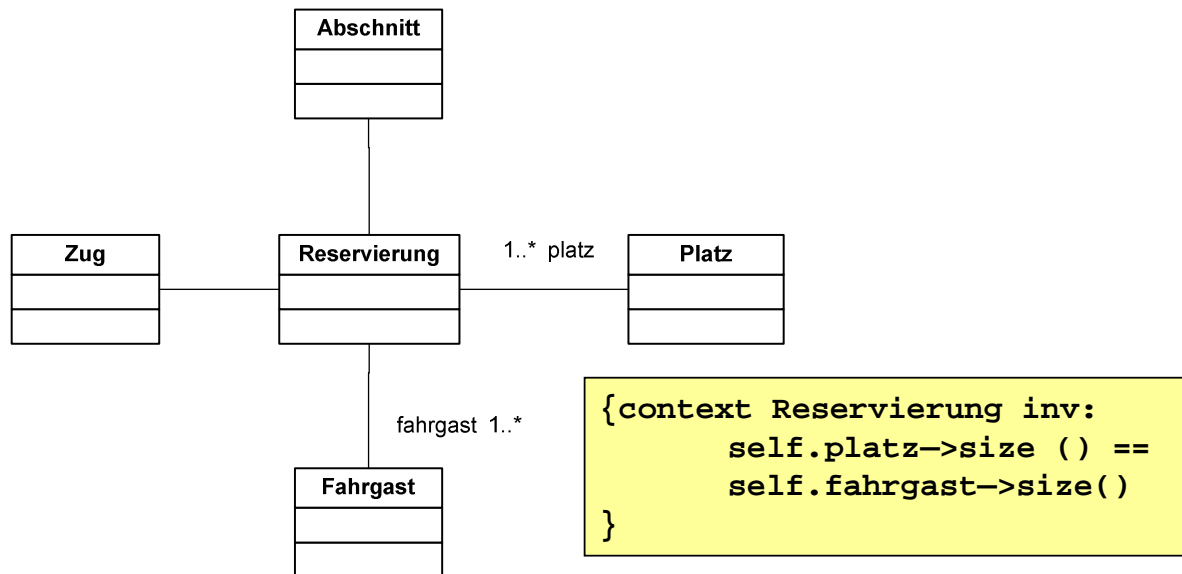
Weitere Beispiele von Zusicherungen:

Der Projektleiter muss zur Menge der Projektmitglieder gehören ausgedrückt durch folgende Zusicherung:

```
{context Projekt inv:
    projektmitglied->includes (leiter) }
```



Die Anzahl der reservierten Plätze muss mit der Anzahl der reservierenden Fahrgäste übereinstimmen

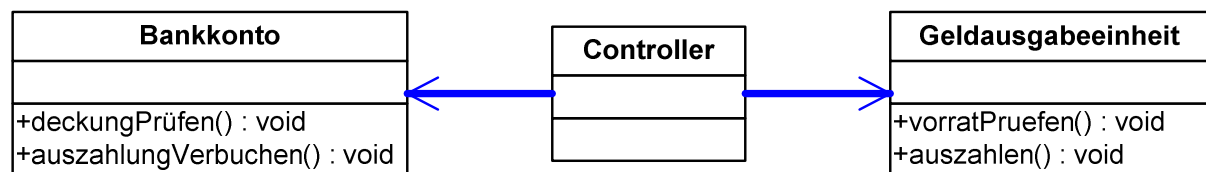


OOD: Detaillierte Klassendiagramme

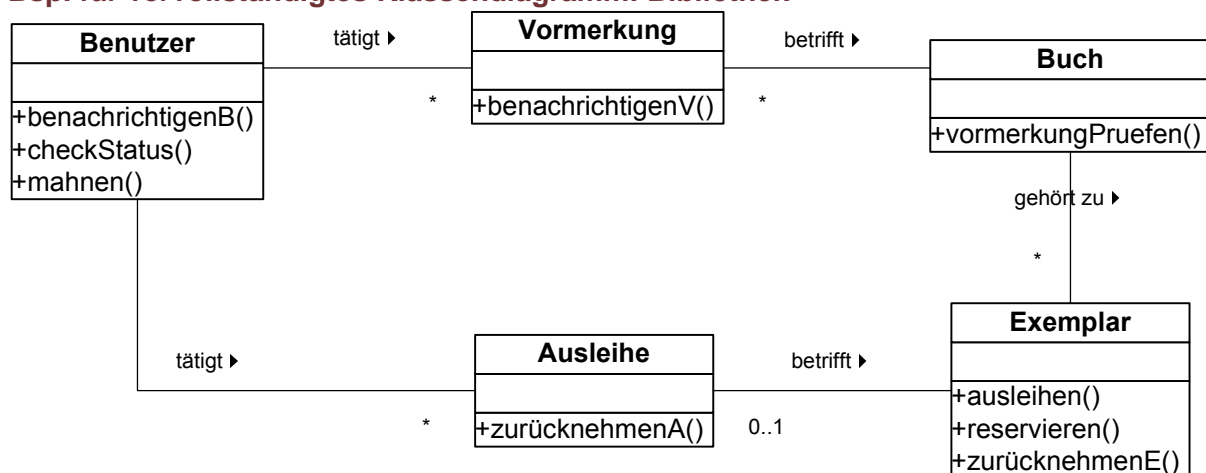
Mit den vorgestellten Notationselementen lassen sich nun die noch unvollständigen Klassendiagrammentwürfe aus der OOA verfeinern. Wo nötig, wird dieser Prozess von der dynamischen Modellierung unterstützt.

Bsp. für vervollständigtes Klassendiagramm: Geldautomat

Das in der OOA ermittelte unvollständige Klassendiagramm wird durch Angabe der Methoden vervollständigt.



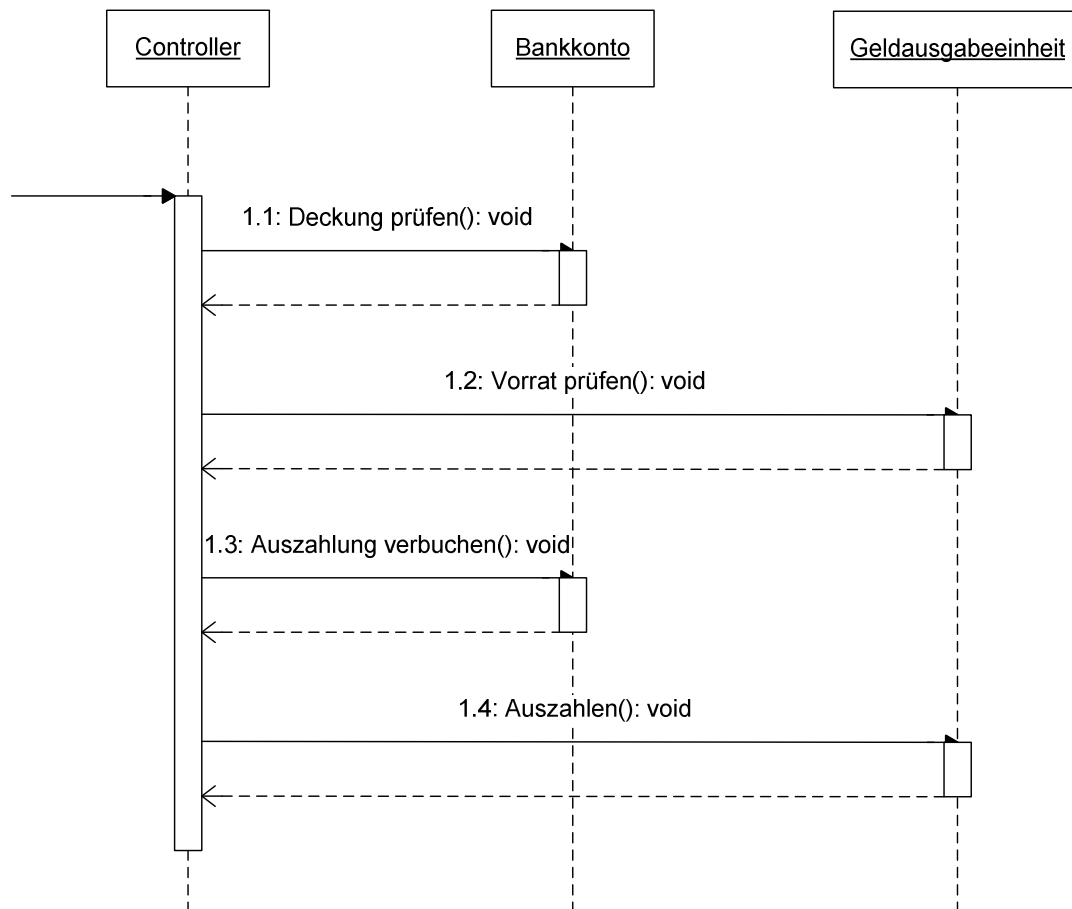
Bsp. für vervollständigtes Klassendiagramm: Bibliothek



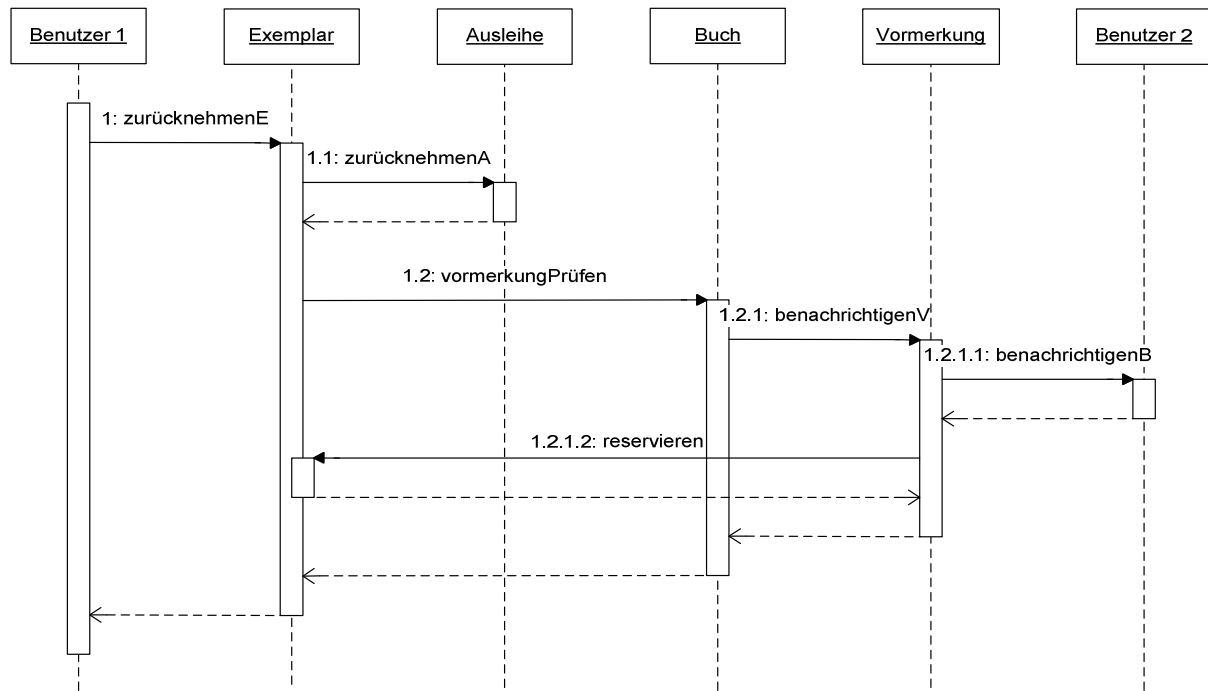
Dynamische Modellierung im Objektorientierten Design

Analog zu den statischen Diagrammen werden nun auch die bei der Objektorientierten Analyse (OOA) entstandenen, zum Teil noch unvollständigen Interaktionsdiagramme, weiter verfeinert. Die Verfeinerung der Interaktionsdiagramme führt zur Präzisierung der Operationen durch Angabe von Parametern und / oder zur Identifizierung weiterer Operationen in den

Klassen. Zu den auftretenden Interaktionen werden nun die bisherigen z.T. noch textuellen Angaben durch einzelne, konkrete Operationsaufrufe detailliert.
Als Beispiel für ein verfeinertes Sequenzdiagramm soll der schon oben in der Analyse besprochene Geldautomat dienen.



Ein weiteres Beispiel ist das oben besprochene Modell einer Bibliothek, hier mit dem modellierten Ablauf für die Rückgabe eines Buchs, das bereits von einem anderen Benutzer angefragt wurde.



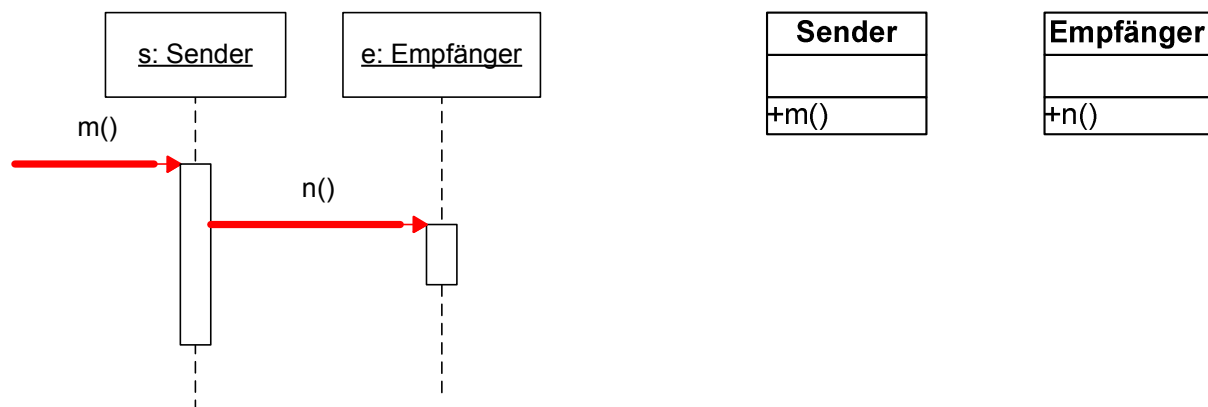
Bei der Modellierung von Nachrichten in Interaktionsdiagrammen (Sequenz- und Kollaborationsdiagramm) unterscheidet man zwischen synchronen Nachrichten, die im Folgenden durch einen Pfeil mit ausgefüllter Pfeilspitze dargestellt werden, und



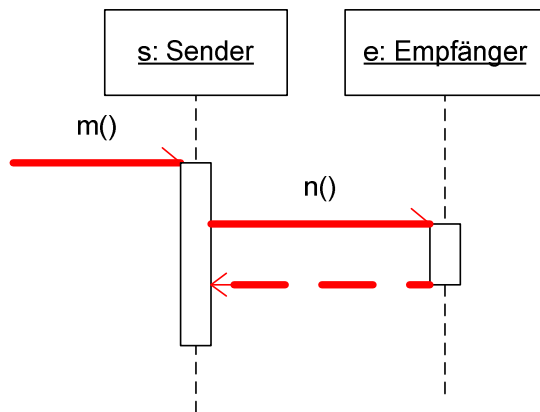
asynchronen Nachrichten, die im Folgenden durch einen Pfeil mit offener Pfeilspitze dargestellt werden.



Synchrone Nachrichten blockieren den Sender solange, bis die Nachricht vom Empfänger vollständig verarbeitet ist (und das Ergebnis vorliegt). Die Implementierung synchroner Nachrichten erfolgt durch Methodenaufrufe. Daraus ergibt sich, dass jede synchrone Nachricht eines Interaktionsdiagramms als Methode des Empfängerobjekts verfügbar sein muss. S. dazu auch das Beispiel unten.



Asynchrone Nachrichten werden verschickt, ohne auf das Ende der Verarbeitung durch den Empfänger zu warten. Die Übermittlung des Ergebnisses erfolgt ggf. wiederum durch eine asynchrone Nachricht in der Gegenrichtung. Die Implementierung kann beispielsweise über Threads oder Mechanismen der Prozesskommunikation erfolgen.



Asynchrone Nachrichten unterscheiden sich deutlich von synchronen Nachrichten. Im täglichen Leben sind Telefonanrufe oder Bestellungen in einem Restaurant meistens synchrone Nachrichten¹⁶. Asynchrone Nachrichten sind u.a. Briefe, Emails oder SMS Nachrichten. Die Synchrone Nachrichtenübermittlung ist in Programmen der Normalfall bei der Modellierung von Objekt-Interaktionen. Dabei entspricht die synchrone Nachrichtenübermittlung dem natürlichen zeitlichen Ablauf bei Methodenaufrufen.

Die Asynchrone Nachrichtenübermittlung wird nur bei besonderen Anforderungen verwendet. Mögliche Gründe sind Zeitgewinn durch Parallelisierung von Abläufen oder Irrelevanz einer Rückantwort. Ein Beispiel dafür ist eine Interaktion mit einer Hardware (etwa beim Öffnen der Lade eines CD-Spielers wird lediglich das Signal zum Öffnen versendet, nicht aber auf das Ende der Aktion gewartet). Des Weiteren können asynchrone Nachrichten zum Initiieren nebenläufiger Berechnungen verwendet werden.

Modellierung durch Zustandsdiagramm

Bei komplexem Verhalten einer Klasse, vor allem, wenn das Verhalten der Methoden vom aktuellen Zustand der Klasse abhängt, erlaubt das Zustandsdiagramm eine eindeutige Darstellung der möglichen Zustandsübergänge in Abhängigkeit von den Methodenaufrufen.

Erst mit Hilfe dieser Information lässt sich die Realisierung der einzelnen Methoden so weit verfeinern, dass sich die Codierung der Methoden direkt ableiten lässt.

Zustandsdiagramm

Ein Zustandsdiagramm ist die graphische Repräsentation eines Zustandsautomaten. Es beschreibt das dynamische Verhalten von Objekten anhand von ereignisgesteuerten Übergängen (Transitionen) zwischen diskreten Zuständen. Damit erhält man eine programmiersprachenunabhängige Verhaltensbeschreibung, die anschließend in der Implementierungsphase in die Zielsprache umzusetzen ist.

Zustände

Ein Startzustand wird als ausgefüllter Kreis dargestellt:



Ein Zustand wird als Rechteck mit abgerundeten Ecken dargestellt:

¹⁶ Die meisten Restaurantbesucher warten bis sie ihr Essen bekommen haben. Nur wenige verlassen das Lokal früher.

Zustand 1

Ein Endzustand wird als ausgefüllter Kreis mit einem weiteren äußeren Kreis dargestellt:



Transitionen

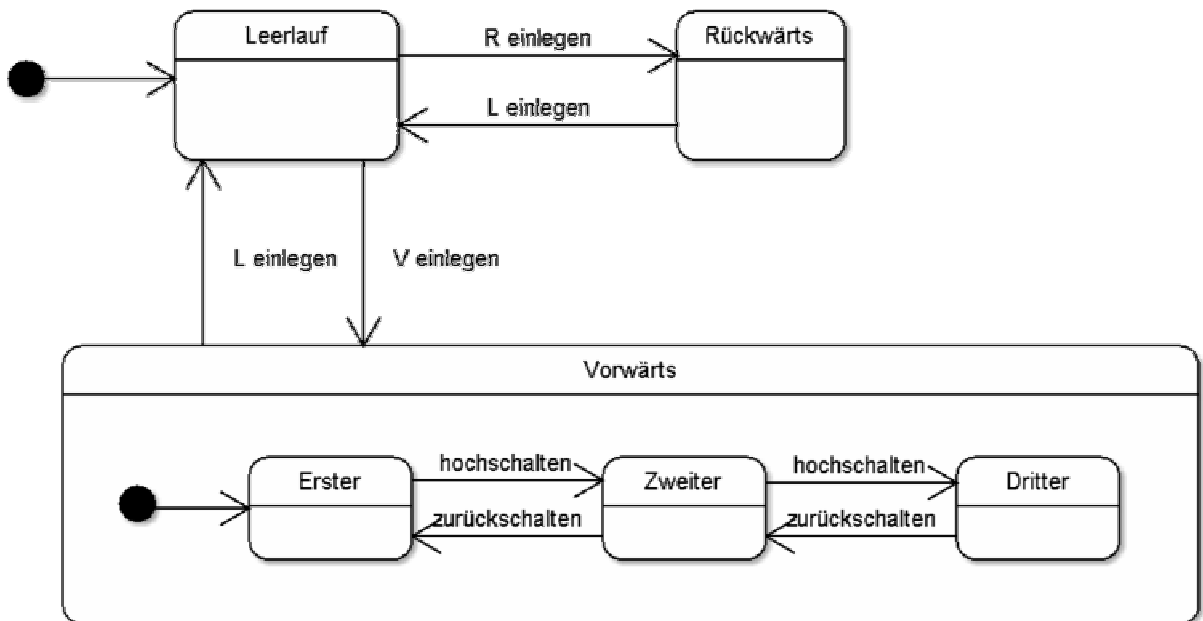
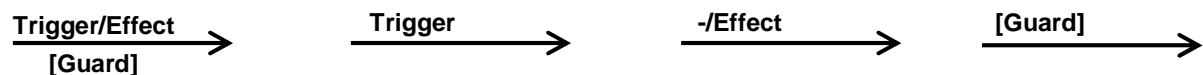
Transitionen werden als Pfeile mit offenen Pfeilspitzen dargestellt:



Die UML bietet die Möglichkeit, eine Transition mit folgenden Angaben zu versehen:

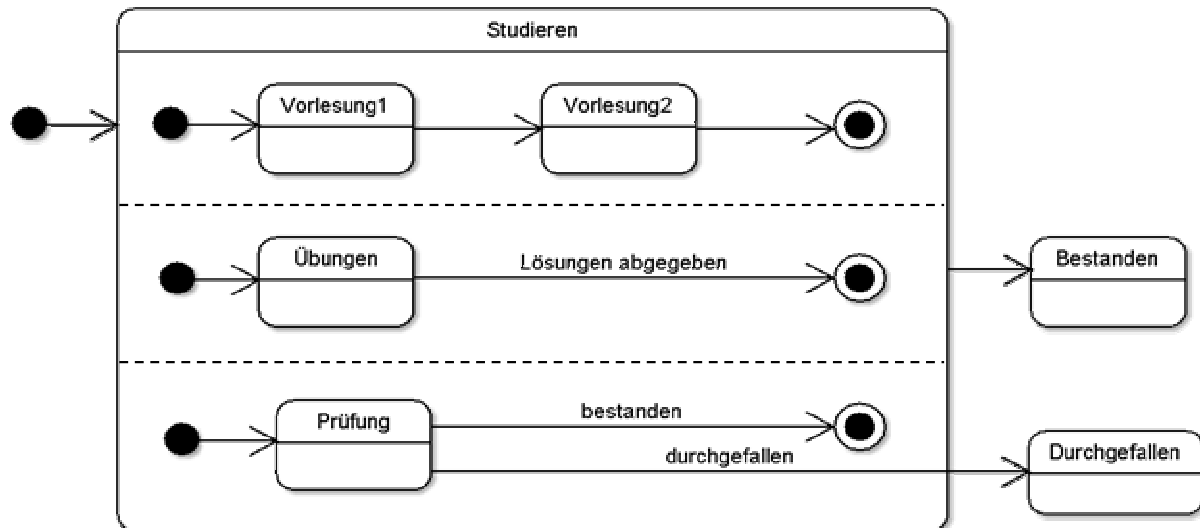
- dem auslösenden Ereignis (Trigger), der die Transition initiiert
- dem ausgelösten Ereignis (Effect), das beim Zustandsübergang stattfindet.
- einer Wächterbedingung (Guard), die erfüllt werden muss, damit die Transition schalten kann

Darstellungen:



Nebenläufige Zustandsautomaten

- Ein Zustand kann durch mehrere parallele Automaten verfeinert werden.
- Der Zustand Bestanden wird erreicht, wenn alle 3 Automaten in Studieren den Endzustand erreicht haben.
- Der Zustand Durchgefallen wird erreicht, wenn die Transition durchgefallen geschaltet wird (die anderen inneren Automaten werden dann bedeutungslos).



Ergebnis von OOA + OOD

Am Ende des objektorientierten Entwurfs sollten verfügbar sein:

1. Vollständige Klassendiagramme mit
 - sämtlichen Klassen
(incl. deren Attribute und vollständigen Operationsbeschreibungen)
 - sämtlichen Assoziationen
(incl. Kardinalitäten, Rollen usw.)
 - sämtliche Vererbungsbeziehungen
2. Vollständige Interaktionsdiagramme
 - Sequenz- bzw. Kollaborationsdiagramme
3. Bei komplexem Verhalten von Objekten
 - ausführliche Zustandsdiagramme
4. Logische und physikalische Modularisierung
 - Paketdiagramme und Komponentendiagramme
 - Einsatzdiagramme

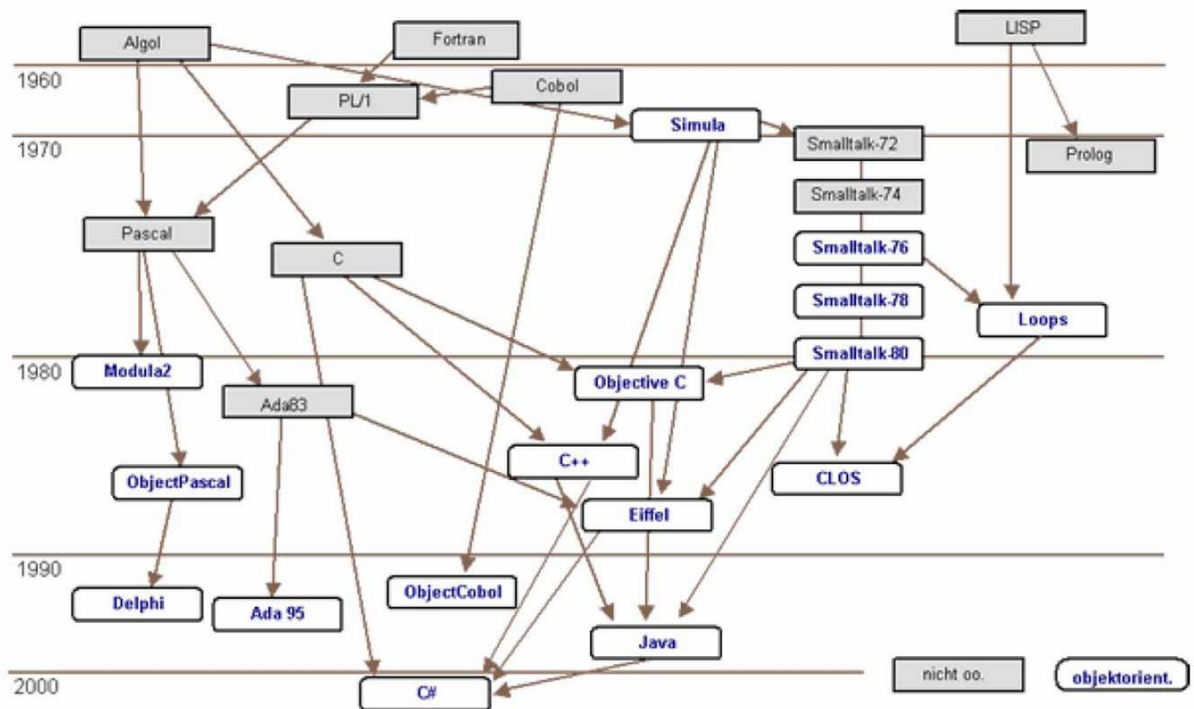
4.9. Implementierung (Codierung)

In vielen Projekten ist die Programmiersprache bereits festgelegt durch Randbedingungen wie Verfügbarkeit von Compilern auf Mikrocontrollern oder organisationsweite Festlegungen oder durch Vorgängerprojekte. darüber hinaus ist die Programmiersprache oft im Auftrag spezifiziert. Zumindest eine Klasse von Programmiersprachen (typischerweise Paradigma) sollte vorweg festgelegt werden.

Sollte die Wahl der Programmiersprache freigestellt sein, so muss eine Kosten-Nutzen-Analyse durchgeführt werden. Die Kosten und Nutzen der in Frage kommenden Programmiersprachenkandidaten sind zu untersuchen. Insbesondere kann die Notwendigkeit der Einstellung von neuem Personal oder die Weiterbildung des vorhandenen Personals zu erheblichen Kosten führen.

Historische Entwicklung

Die Entwicklung von Programmiersprachen begann bereits in den 50er Jahren des vorigen Jahrhunderts. Stammbaumartig wurden aus bekannten Programmiersprachen neue entwickelt durch Erweitern der Syntax, Erweiterung der möglichen Konstrukte und Einführung neuer Paradigmen wie die Objektorientierung. Das Bild unten gibt einen Überblick über die Entwicklung und „Abstammung“ der früher und heute verbreiteten Sprachen.



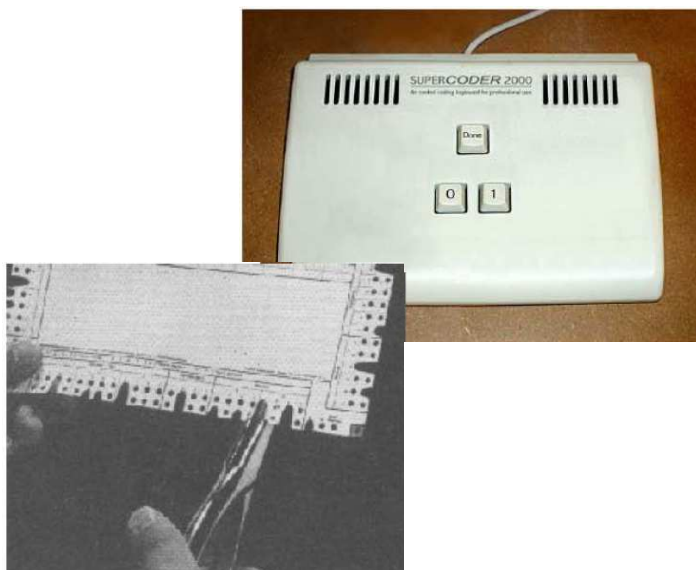
Programmiersprachen-Generationen

Die „Programmiersprache“ der 1. Generation baute auf Bitfolgen auf. „Computer“ kennen nur zwei Zustände, Strom an („1“) und Strom aus („0“). Die Programme („Kochrezepte“) waren Abfolgen binärer Befehle, meist über Schalter mechanisch einzustellen

```

:
00110101 10101101 00110101
00110011 10101101 00110101
10101101 01100110 00110101
00110101 10101101 00110101
01100110 10101101 00110101
10101101 01100110 00110101
    
```

Verwendet wurden entsprechende „Tastaturen“ oder per Hand zu erstellende Programme auf Papier.



Programmiersprachen der 2. Generation waren verdienten erstmalig diesen Namen. Die „maschinennahen“ Befehle in Binärformat wurden durch "Namen" (symbolische Notation — Assembler) ersetzt. Dabei ist nach wie vor eine 1-zu-1 Übersetzung in die binäre „Maschinensprache“ (0/1) möglich. Das Beispiel unten zeigt einen Ausschnitt aus einem Assemblerprogramm.

```
mov dx,03c4h
mov ax,0604h
out dx,ax
neq ax,0100h
out dx,ax
lda dx,03c2h
mov al,0e3h
neq dx,al
mov dx,03c4h
add ax,6500
ret
```

Die 3. Generation der Programmiersprachen („höhere“ Programmiersprachen) führte eine weitere Abstraktion ein. Das Ziel war es Befehle intuitiver im Sinne eines „Kochrezepts“ näher an der natürlichen Sprache zu bringen. Sprachen der 3. Generation benötigen komplizierte Compiler (Übersetzer), die eine 1-zu-5 oder 1-zu-10 Umsetzung von Befehlen der Programmiersprache in Assembler Maschinenbefehle durchführen. Beispiele für diese Sprachen sind COBOL, FORTRAN, Pascal,..., C++, Java.

Programmiersprachen der 4. Generation abstrahieren weiter in Richtung natürliche Sprache. Bei diesen Sprachen wird ein Befehl in 30 bis 50 Maschinenbefehle umgesetzt. Beispiel für diese Sprachen sind Focus und Natural. Ein entsprechendes Codebeispiel steht unten

```
for every surveyor
  if rating is excellent
    add 6500 to salary
```

Unterschiede C zu C++/Java

C, C++ und Java sind heute sehr weit verbreitet. Daher soll kurz auf Gemeinsamkeiten und Unterschiede eingegangen werden. C ist eine funktionale Programmiersprache, die gut geeignet ist Embedded Programmierung. In C lassen sich Laufzeit- und Speicher-effiziente Programme schreiben. Ebenso ist eine maschinennahe Programmierung, z.B. Treiber gut möglich.

Dagegen sind C++ und Java objektorientierte Programmiersprachen. Sie eignen sich gut für große Anwendungen mit komplexen Strukturen. Objektorientierte Modellierung und Programmierung sind nahtlos möglich. Bibliotheken unterstützen GUI- und Client/Server-Programmierung. Insbesondere Java erlaubt Maschinenunabhängige Programmierung.

Unterschiede C++ zu Java

Java wurde entwickelt u.a. um einige Schwächen der Programmiersprache C++ zu verbessern. Daher unterscheidet sich Java in einigen Aspekten deutlich von C++. Java ist durch den Byte-Interpreter absolut Maschinenunabhängig (sofern der Code nicht kompiliert werden soll und der Interpreter auf der Ziel verfügbar ist. Die Speicherverwaltung ist durch einen automatischen Garbage Collector einfacher. Zur Vermeidung von Problemen mit Mehrfachvererbung ist in Java nur Einfachvererbung erlaubt. Nachteilig wirkt sich bei Java der Byte-Interpreter aus, da die Performance geringer ist, als bei kompiliertem Code.

C++ dagegen ist nur mit Einschränkungen maschinenunabhängig, da Bibliotheken (z.B.

MFC-Bibliothek) oft stark „Windows“-lastig sind. Bei der Verwendung von dynamische Speicherallokierung und Verwaltung durch einen C++ Applikation können leicht Fehler gemacht werden¹⁷. Wie schon oben erwähnt kann die Nutzung der Mehrfachvererbung in C++ zu Problemen führen und in der Regel werden C++ Programme schneller sein als Java Programme.

Codegenerierung aus UML (Vom OOD zur Implementierung)

Im Übergang vom OO-Design zur Implementierung wird der Code aus den Designergebnissen abgeleitet. Am Ende des objektorientierten Entwurfs existieren u.a. Klassendiagramme, Sequenzdiagramme und ggf. Kommunikationsdiagramm. Ziel der Implementierungsphase ist es nun, mit Hilfe der in der OOD-Phase ermittelten Diagramme möglichst automatisch einen lauffähigen Code zu generieren. Anhand einer beispielhaften Anwendung soll die Vorgehensweise demonstriert werden.

Als Beispiel soll ein Ausschnitt aus einem POS-System (z.B. Kasse im Supermarkt) dienen. Das Point-Of-Sale-System (POS) ist ein System zur Aufzeichnung von Verkaufsvorgängen (inkl. Zahlungen) in einem Geschäft. Es besteht aus Software- und Hardwarekomponenten (Monitor, Computer, Barcodescanner, usw.).

Ein wichtiges Szenario ist der Prozess Verkauf mit dem Haupt-Aktor Kassierer. Ein Teil dieses Szenarioverlaufs dient als Ausgangspunkt für die beispielhafte Umsetzung des Entwurfs in Code

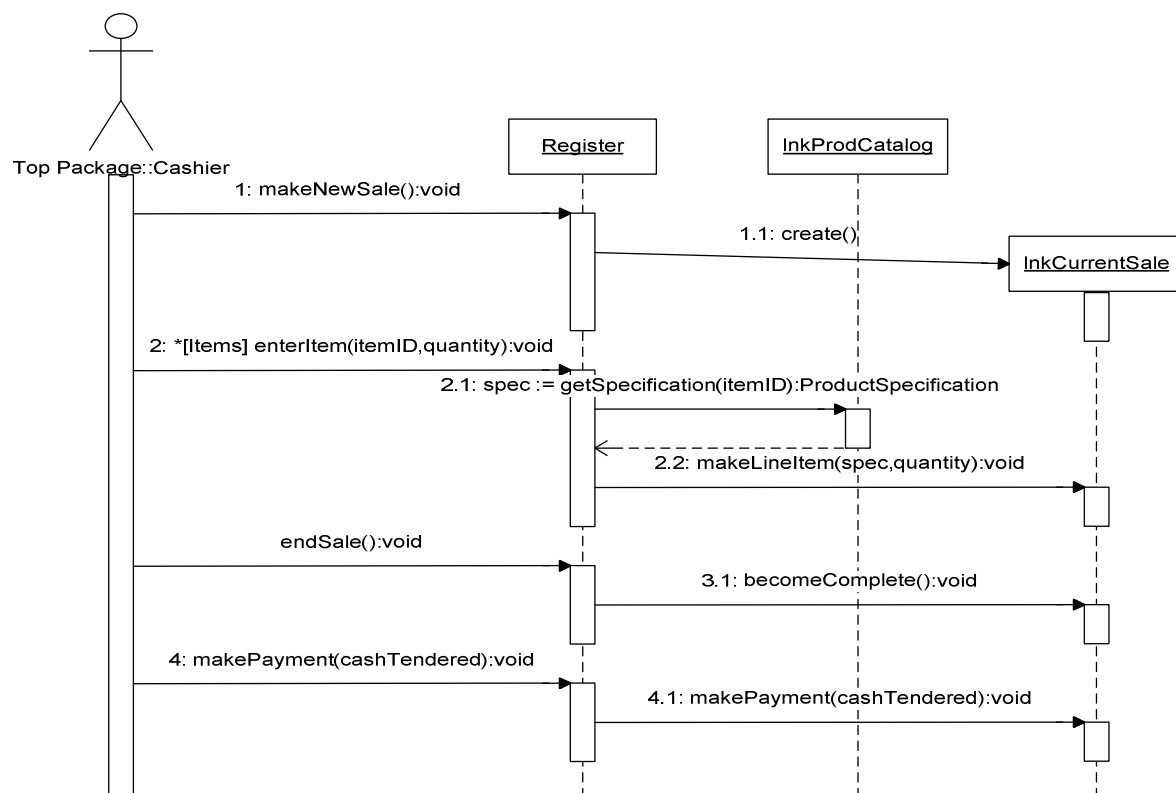
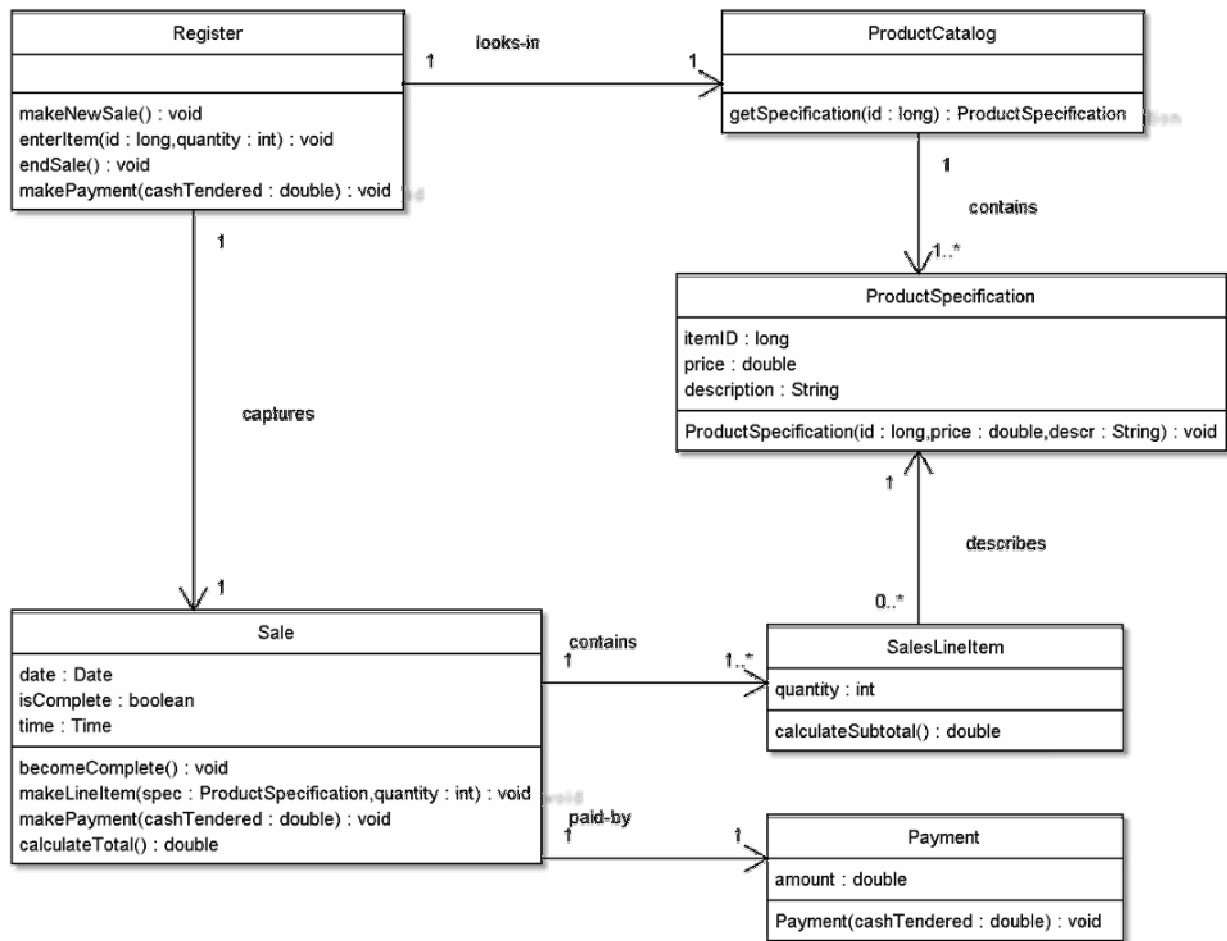
Das Hauptszenario für den Prozess „Verkauf“ stellt sich folgendermaßen dar:

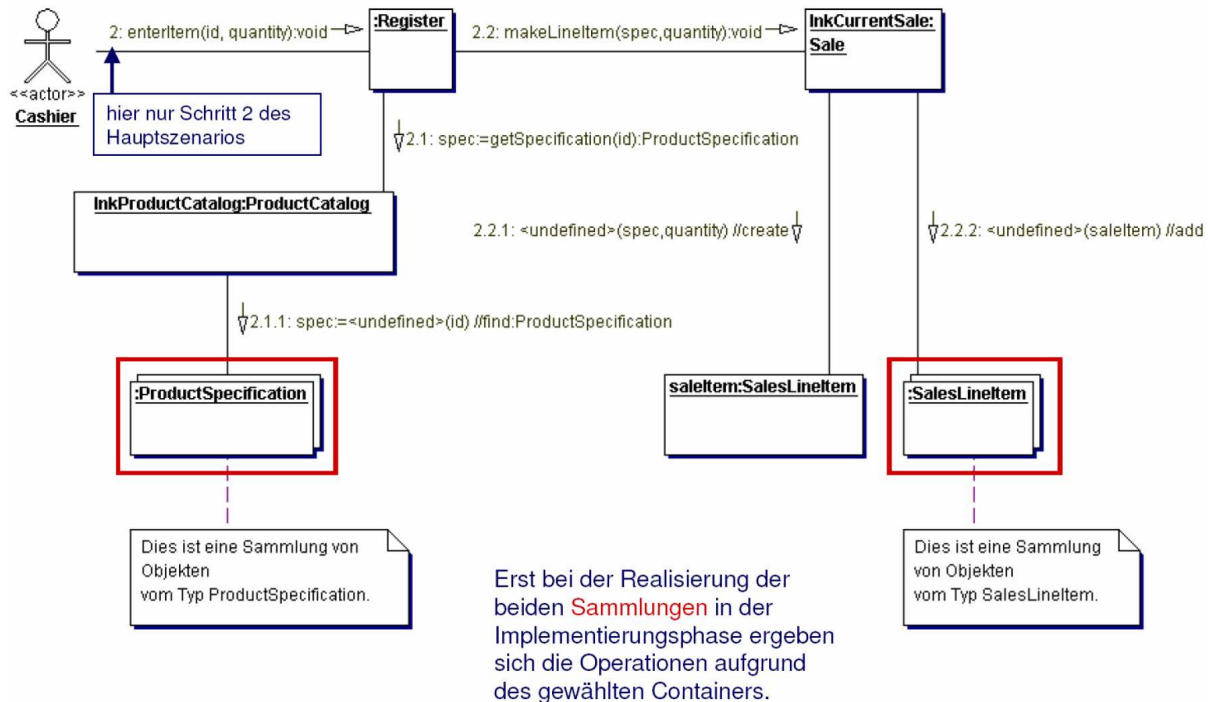
Ein Kunde kommt mit Waren beim Kassierer an.

1. Der Kassierer beginnt einen neuen Verkaufsvorgang.
2. Der Kassierer gibt die Kennung des Artikels ein.
Das System zeichnet daraufhin den Verkauf des Artikels auf und zeigt Beschreibung, Preis und Gesamtsumme an.
Der Kassierer wiederholt den Schritt 2. solange bis alle Waren, die der Kunde kaufen will, eingegeben sind.
3. Der Kassierer beendet den Verkaufsvorgang.
Das System zeigt daraufhin die Gesamtsumme inklusive der enthaltenen Mehrwertsteuer an.
4. Der Kunde bezahlt und der Kassierer gibt die Bestätigung des Vorgangs ein.
Das System speichert den kompletten Verkaufsvorgang.
Das System druckt einen Beleg aus.
Der Kunde verlässt den Kassierer mit seinen gekauften Waren.

Die Analyse und das nachfolgende Design führten zu dem folgenden Klassendiagramm, zu dem weiter unten (vereinfacht) dargestellten Frequenzdiagramm und zu dem äquivalenten Kommunikationsdiagramm:

¹⁷ dies gilt mit Einschränkung natürlich auch für C-Programme.

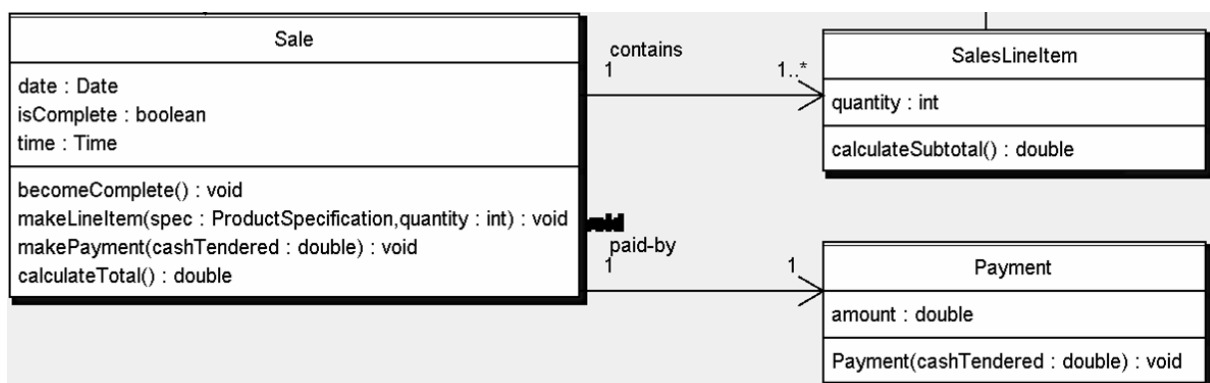




Die Allgemeine Vorgehensweise bei der Erzeugung des Codes geschieht in folgenden Schritten:

1. Die Klassendefinitionen werden auf Basis der Klassendiagramme erzeugt.
2. Typanpassung der einfachen Attribute.
3. Typanpassung der Operationen.
4. Umsetzung/Anpassung der Assoziationen zu Referenzattributen (1:1-Beziehung) bzw. Containern/Collections (1:n-Beziehung).
5. Umsetzung der Interaktionsdiagramme in den Methodendefinitionen.
6. Fertigstellung der Methodendefinitionen (z.B. lokale Variablen, Definition von Algorithmen, soweit nicht bereits modelliert).

Im 1. Schritt wird die Klasse Klasse Sale erzeugt auf Basis des Klassendiagramms der Klasse Sale und der assoziierten Klassen Payment und SalesLinItem.



Die Klasse Sale wird auf Basis des Klassendiagramms in der Zielsprache Java erzeugt. Bei Verwendung eines CASE-Tools kann die Erzeugung größtenteils automatisch erfolgen.

Sale
date : Date isComplete : boolean time : Time
becomeComplete() : void makeLineItem(spec : ProductSpecification, quantity : int) : void makePayment(cashTendered : double) : void calculateTotal() : double

```

public class Sale {
    private Date date;
    private boolean isComplete;
    private Time time;

    public SalesLineItem lnkSalesItem;
    public Payment lnkPayment;

    public void becomeComplete() { }

    public void makeLineItem
        (ProductSpecification spec,
         int quantity) { }

    public void makePayment(
        double cashTendered) { }

    public double calculateTotal()
    { return 0.0; }
}

```

Anmerkung:

Der Source code wurde mit dem Tool "ArgoUML" generiert. Java-Import-Anweisungen wurden der Übersichtlichkeit halber gelöscht.

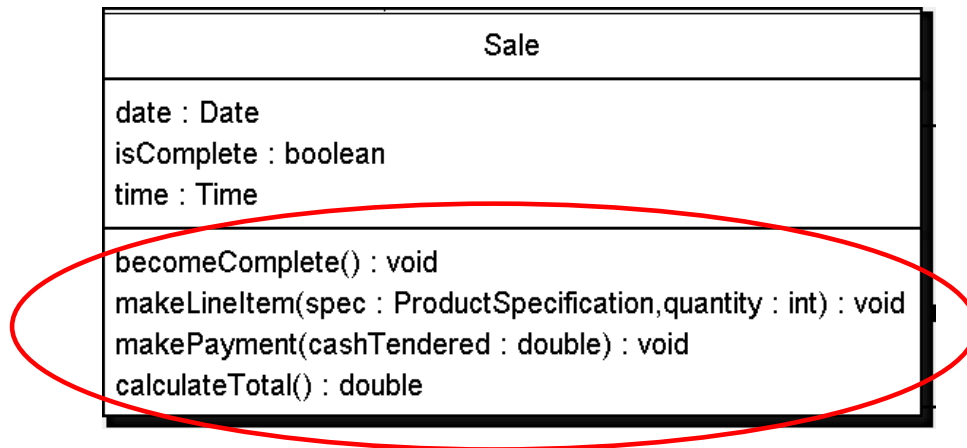
Im 2. Schritt werden die Typen der einfachen Attribute angepasst:

Eine manuelle Anpassung ist notwendig, falls die Datentypen der Attribute in der Zielsprache nicht vorhanden sind. In unserem Bsp.: in Java existiert kein Typ „Time“. Eine manuelle Anpassung ist daher notwendig: Zeiten können zusammen mit einem Datum mit Hilfe des Typs „Calendar“ verarbeitet werden. Alle anderen Attribute entsprechen einem Java-Datentyp.

Sale
date : Date isComplete : boolean time : Time
becomeComplete() : void makeLineItem(spec : ProductSpecification, quantity : int) : void makePayment(cashTendered : double) : void calculateTotal() : double

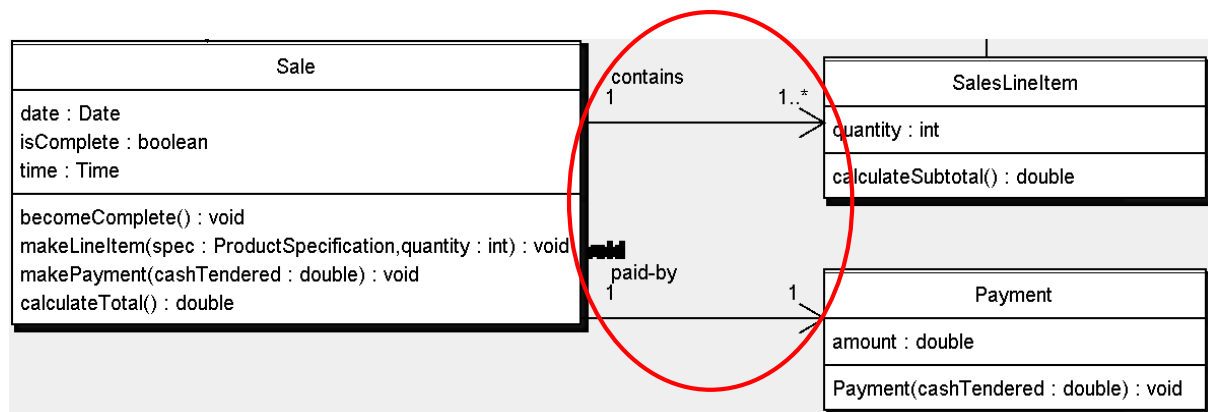
Im 3. Schritt werden die Typen der Operationen angepasst:

Analog zu den Klassen ist eine manuelle Anpassung ist notwendig, falls die Datentypen der Methoden in der Zielsprache nicht vorhanden sind. Die Methoden der Klasse Sale verwenden ausschließlich existierende Java-Datentypen bzw. selbst definierte Klassen (ProductSpecification). Daher ist in diesem Fall **keine** manuelle Anpassung notwendig.

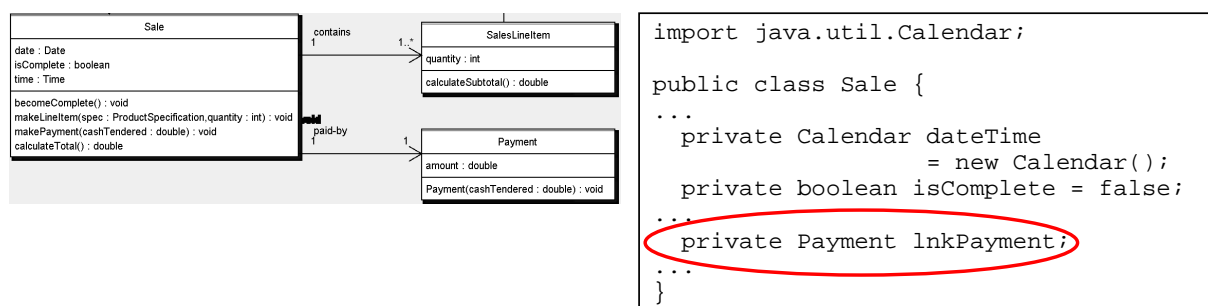


Im 4. Schritt werden die Assoziationen umgesetzt:

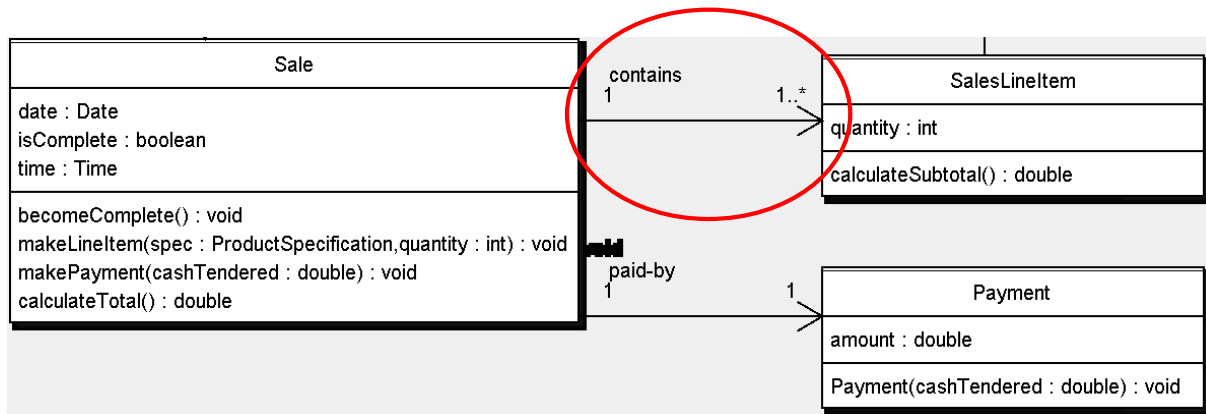
Auch hier ist wiederum eine manuelle Anpassung ist notwendig, falls die Datentypen der Referenz-Attribute (ergeben sich aufgrund der Assoziationen) in der Zielsprache nicht vorhanden sind. Die mögliche und korrekte Umsetzung von 1:n-Beziehungen durch passende Daten-Container (z.B. Listen, Mengen, Arrays) ist zu prüfen.



Eine 1:1-Beziehung besteht zwischen „Sale“ und „Payment“ (Assoziation „Paid-by“) Aufgrund der Richtung und der Multiplizität der Assoziation ergibt sich, dass jedes Objekt der Klasse „Sale“ ein Objekt der Klasse „Payment“ referenzieren muss. Dies wird vom UML-Tool ArgoUML durch Erzeugung des Attributes `lnkPayment` automatisch umgesetzt.



Eine 1:n-Beziehung existierzwischen "Sale" und "SalesLineItem" (Assoziation „contains“) Aufgrund der Richtung und der Multiplizität dieser Assoziation ergibt sich, dass jedes Objekt der Klasse "Sale" i.A. mehrere Objekte der Klasse "SalesLineItem" referenzieren muss. In diesem Fall wird die Umsetzung nicht automatisch vorgenommen, sondern muss manuell durch Festlegung einer Sammlung (hier: `ArrayList`) erfolgen.



```

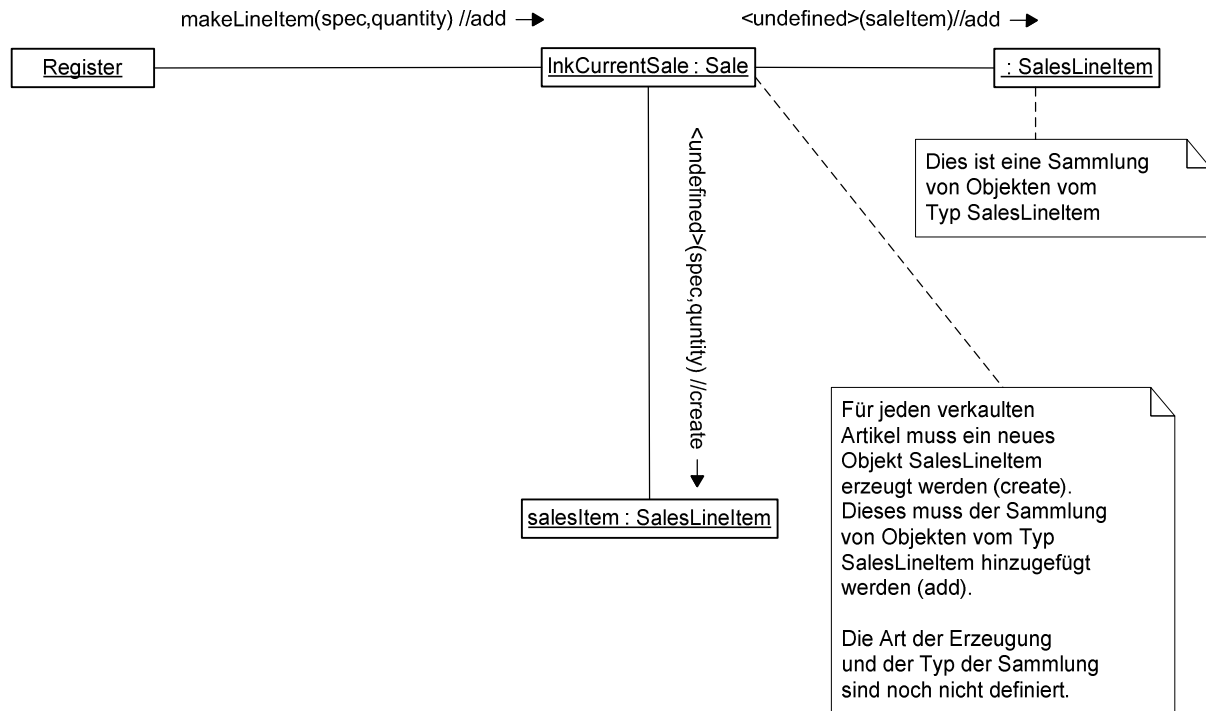
import java.util.Calendar;

public class Sale {
    ...
    private Calendar dateTime
        = new Calendar();
    private boolean isComplete = false;
    ...
    private Payment lnkPayment;
    private ArrayList<SalesLineItem> lnkSalesLineItems
        = new ArrayList<SalesLineItem> ();
    ...
}
    
```

Anmerkung:

das Tool "ArgoUML" erzeugt hier den Java-Datentyp "Vector", der allerdings nicht "von Haus aus" typenrein ist. Mit der "ArrayList"-Collection wird sichergestellt, dass nur Objekte vom Typ "SalesLineItem" in die Collection aufgenommen werden.

Nachdem die statischen Teile umgesetzt sind, erfolgt nun die Umsetzung der dynamischen Modelle in Code, beginnend mit der Umsetzung der Interaktionsdiagramme.



Die Aufrufsequenzen werden in der Regel aus den Interaktions-Diagrammen durch passende Methodenaufrufe automatisch in Code umgesetzt. Für das POS-System erkennt man beispielsweise aus dem Kommunikationsdiagramm, dass die Methode "makeLineItem()"

1. ein neues Objekt „SalesLineItem“ erzeugt (**create**) und
 2. dieses der Collection (hier ArrayList) der "SalesLineItem"-Objekte hinzufügt (**add**).
- Im Kommunikationsdiagramm ist nicht festgelegt («**undefined**»), wie die Erzeugung erfolgt und welche Art der Collection zu verwenden ist.

Aus der Definition der Klasse "Sale" ergibt sich, dass die Methode "makeLineItem" ein "SalesLineItem erzeugt und das neue Objekt der ArrayList "lnkSalesLineItems" hinzufügt

```

import java.util.Calendar;

public class Sale {
    ...
    private ArrayList<SalesLineItem> lnkSalesLineItems
        = new ArrayList<SalesLineItem> ();
    ...
    public void makeLineItem (ProductSpecification spec, int quantity )
    {
        SalesLineItem item = new SalesLineItem (spec, quantity);
        lnkSalesLineItems.add (item);
    }
    ...
}
    
```

Die weitere Programmierfähigkeit befasst sich mit der Vervollständigung des Programmcodes. In der Regel sind jetzt noch die Methodenrumpfe manuell zu programmieren. Im obigen Beispiel oben ist dies nicht notwendig, da die Methoden new und add in Java bereits definiert sind.

Round-Trip-Engineering

In der Praxis werden oftmals neue Programme aus Teilen oder Erweiterungen bestehender Programme entwickelt. Ist dies nicht der Fall, sondern wird ein Programm komplett neu entwickelt, spricht man von Forward Engineering. D.h. das fertige Softwaresystem ist

Ergebnis des Entwicklungsprozesses, wobei der Programmcode aus Entwurfsmodellen erzeugt wird. Im Gegensatz dazu steht das Reverse Engineering. Hier ist ein vorhandenes Softwaresystem Ausgangspunkt der Analyse. Das Entwurfsmodell wird aus dem Programmcode erzeugt.

Das Round-Trip Engineering ist die Kombination aus Forward Engineering und Reverse Engineering

Codierungsregeln

Jede Programmiersprache (z.B. C, C++, Ada, Java) enthält fehleranfällige Konstrukte. Ein Beispiel zur Programmiersprache C ist die Tatsache, dass gemäß dem ISO-C-Standard (Anhang G) ist die Auswertungsreihenfolge von Ausdrücken nicht spezifiziert ist.

Folgender Code sollte also nicht geschrieben werden:

```
...
int x = 1;
int ar[] = {0, 1, 2};
int y = ar[++x] + x;
...
```

Bei Auswertung von links nach rechts hat y den Wert 4, bei Auswertung von rechts nach links hat y den Wert 3

Safe Subsets

Zur Vermeidung fehleranfälliger Konstrukte, insbesondere für sicherheitskritische Software, werden Regeln definiert, die den ursprünglichen Wortschatz einschränken. z.B. in Bezug auf die Auswertungsreihenfolge von Ausdrücken:

„Verwende den Inkrement-Operator immer in einem eigenen separaten Ausdruck.“

Eine mögliche Lösung zum Problembeispiel oben ist folgender Code:

```
...
++x;
int y = ar[x] + x;
...
```

Auf diese Weise entstehen so genannte „Safe Subsets“, die in Regelwerken wie z.B. MISRA-C für C dokumentiert werden.

Allgemeine Codierungsregeln

Bevor weiter unten auf MISRA als spezifisches Regelwerk für C eingegangen wird, soll noch einige allgemeine und programmiersprachenunabhängige Regeln für das Codieren eingegangen werden. In der Regel sind Verständlichkeit und Lesbarkeit für alle Betroffenen (z.B. Codierer, Reviewer, Tester) wichtiger als ein geringerer Schreibaufwand. Daher sind gute Kommentierung, ein problemorientierter Aufbau, angemessene Datenstrukturen und ein einheitlicher Programmierstil (möglichst unternehmensspezifisch, wenigstens projektspezifisch) und eine gute Verständlichkeit meist wichtiger als Betriebsmittelbedarf (Speicherplatz, Rechenzeit). Insbesondere sollte Lesbarkeit vor Zeitersparnis, Klarheit vor Genialität (man denke an später notwendige Programmpflege und / oder Nachweisführung!) gesetzt werden. Allerdings kann Rechenzeit im Embedded-Bereich sehr wichtig sein, daher muss u.U. ein geeigneter Kompromiss gefunden werden.

Wann immer möglich sollte der Code problemorientiert aufgebaut werden, d.h. das Programm soll in Daten- und Ablaufstruktur das zu lösende Problem widerspiegeln. Es sollten Sprachelementen verwendet werden, die die Absicht des Programmierers wiedergeben.

In Großprojekten oder Projekten, die verteilt entwickelt werden, ist ein einheitlicher Stil bei Codierung und Dokumentation unerlässlich. Daher ist auf einen einheitlichen Programmierstil, einheitlichen Dokumentationsstil, einheitliches Programmlayout, z.B. Einrückungen logischer Strukturen und Zeilenaufteilung (z.B. nur eine Anweisung pro Zeile) zu achten.

Darüber hinaus sollten „Tricks“ beim Codieren möglichst vermeiden werden. Dazu gehört z.B. Mehrfachverwendung von Speicherplatz für verschiedene Zwecke, Veränderung des Wertes des Schleifenzählers im Schleifenrumpf oder Änderungen von Befehlen durch das Programm selbst.

Das folgende Ruby-Programm modifiziert sich selbst und gibt bei wiederholten Aufruf unterschiedliche Ergebnisse aus.

```
class HelloWorld
  def greet()
    print("Hallo, Welt!\n");
    def self.greet()
      print("Tschüss, Welt!\n")
    end
  end
end

hw = HelloWorld.new
hw2 = HelloWorld.new
hw.greet()
hw.greet()
hw2.greet()
```

Bei der Ausführung der Methode greet() wird diese Methode durch „def self.greet“ für das eigene Objekt neu definiert. Es werden 2 Objekte (hw, hw2) erzeugt.

Die erzeugte Ausgabe ist:

```
Hallo, Welt!
Tschüss, Welt!
Hallo, Welt!
```

Weitere allgemeine Codierungsregeln fordern eindeutige Identifizierung der Objekte über ihren Namen, leicht erkennbare Bedeutung, wobei der Name „s lang wie nötig, so kurz wie möglich“ sein sollte. Bei Deklaration und Initialisierung ist darauf zu achten, dass alle Variablen möglichst bereits bei der Deklaration initialisiert werden. Zahlen sollten direkt im Code nicht verwendet werden, stattdessen über Konstanten definiert werden. Dies erleichtert die spätere Programmpflege, sollte sich eine Konstante ändern.

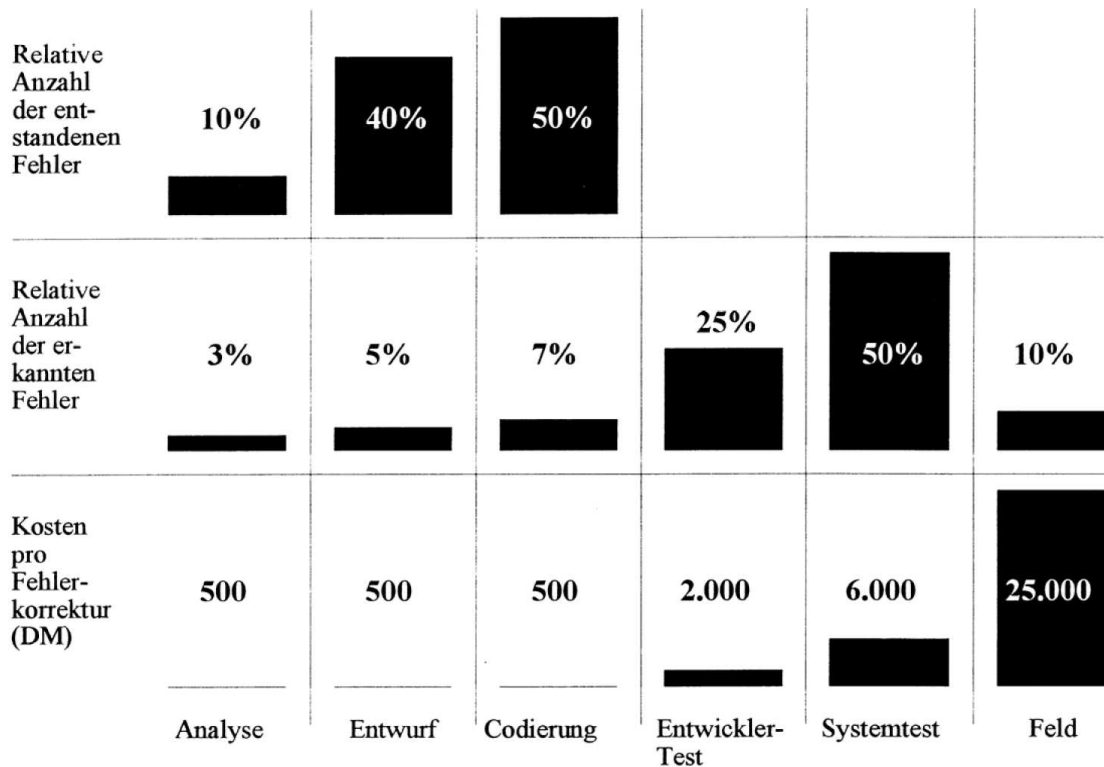
MISRA C

- Ein übersetzbares, aber nicht verstehbares C Programm:
(wird auch als "source code obfuscation" bezeichnet)

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,##/*{w+/w#cdnr/+,}{r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;q#n+,/+k#;*,/'r :d*3,}{w+K w'K:'+}e#';dq#'\
q#'+d'K#!/+k#;q#r}eKK#}w'r}eKK{nl}'/##;#q#n')}{#}w')}{nl}'/+#n';d}rw' i;# \
){nl}'/n{n#'; r{#w'r nc{nl}'/#{l,+K {rw' iK;{nl}'/w#q#n'wk nw' \
iwk{KK{nl}'/w{%l##w#' i; :{nl}'/*{q#ld;r}{nlwb!/*de}'c \
;;{nl}'-}{rw}'/+,}##*')#nc,',#nw]/+kd'+e}+;#rdq#w! nr/' )}{rl#'{n' ')# \
```

```
}'+}##(!/!/)
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{ }:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}1 ) }
```

Empirische Ergebnisse (Quelle: Liggesmeier)



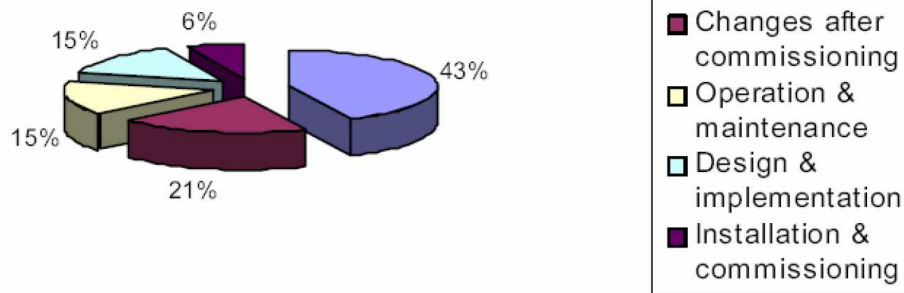
Sichereres C mit MISRA

- **MISRA:** Motor Industry Software Reliability Association:
 - Ein Zusammenschluss von Fahrzeug-Herstellern, Zulieferern und Ingenieurs-Beratungs-Firmen von Großbritannien
- Historie:
 - MISRA-C: 1998[1]: Erstveröffentlichung April 1998, Nuneaton
 - MISRA-C: 2004[2]: veröffentlicht October 2004, Nuneaton
 - MISRA, Development Guidelines for Vehicle Based Software [6], November 1994, Nuneaton
 - ...
- Siehe auch: <http://www.misra.org.uk/>

MISRA-C kann bei Reduzierung einiger Fehlertypen unterstützen

Incident Primary Cause by Phase

Source: HSE Out of Control



- ☐ Can help to reduce the 15% of defects that are introduced during design and coding
- ☐ Embodies accumulated automotive software engineering knowledge and best practice
- ☐ A suitable subset of C for use at SIL2 and above, as suggested by the MISRA Guidelines
- ☐ Must be used in conjunction with a structured development process
- ☐ Use of MISRA C in itself is no guarantee of software quality – nearly two thirds of defects arise from errors in specification or errors made during changes to specification from which MISRA C can offer little protection

Source: S. Montgomery, The Role of MISRA C in Developing Automotive Software, Ricardo UK

Hintergrund: Benutzung von C in der Industrie

- Erhöhte Wichtigkeit der C Programmiersprache (in the Automotive-Industrie)
- Potential für Portierungsfähigkeit auf große Anzahl verschiedener Micro-Prozessoren
- C ermöglicht high-level, low-level, input/output Operationen
- Erhöhte Komplexität von Applikationen zwingt zum Gebrauch einer Hochsprache gegenüber Assembler-Sprachen
- C kann kleineren und weniger Speicherplatz verbrauchenden Code erzeugen als viele andere Hochsprachen (C++, Java)
- Vermehrter Gebrauch von Code-Generatoren aus Modellierungs-Tools

Unsicherheiten in der Sprache C

Ursachen für nicht beabsichtigtes Verhalten des C-Codes:

- Der Programmierer macht Fehler
 - in Bezug auf Stil und Ausdruck
 - C-Syntax (Tippfehler wie "=" anstatt "==")
 - die Philosophie von C ist, dass der Programmierer weiß, was er tut (keine PASCAL-ähnliche Typsicherheit)
- Der Programmierer missversteht die Programmiersprache (z.B. Operator-Präzedenz)
- Der Compiler macht nicht das, was der Programmierer erwartet
- Der Compiler hat Fehler
- Laufzeit-Fehler: C macht keine Laufzeit-Checks für z.B. Overflows, gültige Address-Zeiger, Array-Out-of-bounds, ...

MISRA-C: Anpassung einer Untermenge von Regeln

- Abweichungen müssen beurteilt werden nach Kriterien der Notwendigkeit und Sicherheit (Safety).
- Der Abweichungs-Prozess sollte Teil des formellen Qualitäts-Managements ein.
- Zwei Kategorien von Abweichungen:
 - Projekt-Abweichungen:
 - erlaubte Aufweichung von Regel-Anforderungen unter speziellen Umständen

- sollte gereviewed werden als Teil des formellen Abweichungs-Prozesses
- Spezifische Abweichungen
 - Abweichung einer bestimmten Regel in einer einzelnen Datei
 - Wird gereviewed im Entwicklungs-Prozess
- Der Abweichungs-Prozess sollte nicht benutzt werden, um die MISRA-Absichten zu untergraben

MISRA-C: Kategorisierung der Regeln

Regel	Kategorie	Regel	Kategorie
1	Umgebung	12	Ausdrücke
2	Spracherweiterungen	13	Kontrollfluß-Anweisungen
3	Dokumentation	14	Kontrollfluss
4	Zeichensatz	15	Switch-Ausdrücke
5	Identifiers	16	Funktionen
6	Datentypen	17	Zeiger und Arrays
7	Konstanten	18	Strukturen und Unions
8	Deklarationen und Definitionen	19	Präprozessor-Anweisungen
9	Initialisierungen	20	Standard-Bibliotheken
10	Arithmetische Typumwandlungen	21	Laufzeit-Fehler
11	Zweitertyp-Umwandlungen		

Im Folgenden werden einige Beispiele für MISRA-Regeln aufgeführt:

- Regel 1.4 (Forderung):
Der Compiler/Linker muß garantieren, dass eine 31-Zeichen-Unterscheidung sowie Groß-/Kleinschreibung bei externen Identifiers unterstützt wird.
- Regel 1.5 (Ratschlag):
Gleitkomma-Implementierungen sollten sich nach dem Gleitkomma-Standard richten
z.B.:
`/* IEEE 754 single-precision floating point */
typedef float float_32t`
- Regel 2.1 (Forderung):
Assembler-Aufrufe sollen eingekapselt und isoliert sein.
Beispiel:
`#define NOP asm(" NOP")`
- Regel 2.2 (Forderung):
Sourcecode soll ausschließlich Kommentare der Form `/* ... */` benutzen
- Regel 2.3 (Forderung):
Die Zeichenfolge `/*` soll nicht in einem Kommentar benutzt werden
- Regel 12.1 (Ratschlag):
Es sollte möglichst wenig von den C-Regeln für Operator-Präzedenzen abhängig gemacht werden.

- Regel 12.2 (Forderung):
Das Resultat eines Ausdrucks soll immer gleich sein bei jeder möglichen Reihenfolge der Abarbeitung.
- Regel 12.3 (Forderung):
Der sizeof-Operator soll nicht in Ausdrücken benutzt werden, die Seiteneffekte beinhalten
Beispiel:

```
int32_t i; int32_t j;  
j = sizeof (i=1234);  
/* j is set to the sizeof the type of i which is an int */  
/* i is not set to 1234
```
- Regel 12.4 (Forderung):
Der rechte Operand eines logischen && oder || Operators soll keine Seiteneffekte beinhalten
- Regel 12.6 (Forderung):
Die Operanden von logischen Ausdrücken (&&, || und !) sollen Bool'sche Ausdrücke sein. Bool'sche Ausdrücke sollten nirgends sonst als Operanden benutzt werden.
- Regel 12.7 (Forderung):
 - Bit-Operationen sollen nicht auf Operanden angewandt werden, deren Datentyp vorzeichenbehaftet ist.

Tools für Überprüfung der MISRA-Regeln

Es gibt Tools, die statisch, d.h. ohne Ausführen des Codes, die Einhaltung von MISRA regeln überwacht. Als Beispiele seien hier zwei Tools (PC-Lint und QAC) erwähnt.

PCLint der Fa. Gimpel Software (www.gimpel.com) überprüft C/C++ Quellcode auf Einhaltung von MISRA Regeln.C++-Sprachumfang. Im Gegensatz zu einem Compiler, der nur ein Code Modul betrachtet, analysiert PC-Lint C/C++ Quellen modulübergreifend. PC-Lint für C/C++ kontrolliert die Einhaltung der MISRA Richtlinien, überprüft Quellcode auf typische C++- und C-Fehler und findet z.B.

- nicht initialisierte Variablen
- vererbte, nicht virtuelle Destruktoren
- Typ-Unverträglichkeiten
- falsch formulierte Macros
- unbeabsichtigtes Name-hiding
- Statische Variablen in In-line-Funktionen von Headern
- Fehler beim Kopieren einer Basisklasse oder bei der Benutzung des Copyconstructors der Basisklasse
- Gebrauch von 'throw'in einem Destruktor
- unreferenzierte 'catch'-Parameter
- virtuelle Funktionen mit einem default Parameter

PC-Lint ist ein preiswertes Tool mit guter Dokumentation, die Meldungen detailliert erklärt. Sehenswert ist auch die Webseite von Gimpel, die zahlreiche Beispiele von fehlerhaftem C/C++-Code zeigt.

Das Tool QA-C/MISRA der Firma QA Systems lokalisiert Quellcode, der nicht den MISRA-Regeln entspricht, verbindet dabei Meldungen direkt mit dem Quellcode und den entsprechenden MISRA-Regeln. QA-C ist Individuell auf jedes MISRA Subset konfigurierbar (auch nur für Teile eines Projektes). Über Querverweise (HTML) kann einfach zu Regeldefinitionen und erklärenden Beispielen gesprungen werden. Darüber hinaus generiert QA-C

Reports über die Softwarequalität (Art und Häufigkeit von Regelverletzungen etc.) und weitere Metriken (textuelle und grafische Metriken, z.B. bzgl. Testbarkeit, Pflegbarkeit, Portabilität des Codes). Qa-C unterstützt Qualitätsinitiativen wie CMM, ISO9003/EN29003, ISO 9126, IEC 61508, DO-178B, Def Stan 00-55. Eine Integrationen in Entwicklungsumgebungen wie z.B. Visual Studio oder Codewright und allen gängigen Versionskontrollsystemen ist möglich. Der hohe Preis ist ein gravierender Nachteil von QA-C

Zusammenfassung

MISRA-C definiert eine sichere Untermenge von C und erhöht damit die Qualität und Sicherheit der in C unter Beachtung der MISRA Regeln erstellten Software. Über die Software hinaus muss überlegt werden, wie man mehr Sicherheit ("Safety") in einem technischen System erreicht, dass aus Software, Hardware und Mechanik besteht.

4.10. Software Test

Definition

Auf für den Begriff Software Testen gibt es zahlreiche Definitionen. Hier sei eine IEEE Definition erwähnt:

"The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items"

[IEEE 829 Definition].

Motivation

Warum muss Software getestet werden? Die Antwort ist, wie die Erfahrung zeigt, dass Software meistens / immer Fehler enthält. Testen ist **die** Methode, Fehler in einer Software zu entdecken, nachdem die Software codiert ist. Vielfach ist Testen Qualitätsanforderung des Kunden. Gesetzliche Vorschriften wie Produkthaftung fordern intensive Tests, gestuft nach den Sicherheitsanforderungen, insbesondere auch bei sicherheitskritischen Systemen wie Kraftwerke, Flugzeuge, Fahrzeuge, oder medizinische Geräte.

Eine weitere Motivation Software zu testen, ist die Tatsache, dass die Kosten, die ein Fehler verursachen kann, extrem hoch werden können. Während in frühen Entwicklungsphasen Fehler noch „billig“ zu beheben sind, steigen die Behebungskosten exponentiell mit der Zeit an. Die Grafik unten zeigt dies im Fall einer Fahrzeugsoftware, deren Korrektur im Feld, d.h. das Fahrzeug ist schon an den Kunden ausgeliefert, ca. 250€ kostet. Bei absatzstarken Fahrzeugtypen können hier schnell mehrstellige Millionenbeträge zusammenkommen.

				~ 12 k €	>250 € pro Fzg.!	Kosten einer Fehlerkorrektur
~ 1 k€	~ 1.5 k€	~ 3 k €	~ 6 k €			
Requirements	Design	Code	SW Test	System Test	Feld	
Fehlervermeidung			Fehlerfinden			

Je früher also ein gefunden wird, desto billiger ist er. In frühen Phasen steht die Fehlervermeidung im Vordergrund, in den Testphasen das Aufspüren der noch vorhandenen Fehler. Je früher ein Fehler gefunden wird, desto höher ist auch die Wahrscheinlichkeit, dass durch die Korrektur keine neuen Fehler entstehen.

Von gesetzlicher Seite gibt es das Produkthaftungsgesetz, das unten auszugsweise aufgeführt ist:

Gesetz über die Haftung für fehlerhafte Produkte (Produkthaftungsgesetz - ProdHaftG)

1. Haftung

(1) Wird durch den Fehler eines Produkts jemand getötet, sein Körper oder seine Gesundheit verletzt oder eine Sache beschädigt, so ist der Hersteller des Produkts verpflichtet, dem Geschädigten den daraus entstehenden Schaden zu ersetzen. [..]

(2) Die Ersatzpflicht des Herstellers ist ausgeschlossen, wenn

...

5. der Fehler nach dem Stand der Wissenschaft und Technik in dem Zeitpunkt, in dem der Hersteller das Produkt in den Verkehr brachte, nicht erkannt werden konnte.

[..]

Wichtig ist der Abschnitt 5, der besagt, dass die Haftung nicht besteht, wenn das Produkt nach dem Stand der Wissenschaft und Technik entwickelt wurde. Dies umfasst heute auch einen entsprechenden Testprozess.

Abgrenzung Test – Korrektheit beweisen - Debuggen

Testen ist abzugrenzen von Methoden, Korrektheit zu beweisen und auch vom Debuggen. Das Ziel des Tests ist es Fehler zu finden und damit Vertrauen in die Qualität der SW schaffen. Testen hat immer den Charakter einer Stichprobe, d.h. es gibt eine Wahrscheinlichkeit, dass beim Testen ein Fehler gefunden wird. Testen ist immer anwendbar und bei genügend Erfahrung gut planbar bzgl. Aufwand und Dauer. Oftmals werden Tests spezifiziert, ausgeführt und ausgewertet von spezialisierten Testern (als eigenem Rolle im SW Engineering).

Mittels Testen kann (leider) kein Beweis (mathematischer) Korrektheit einer Software erbracht werden. Obwohl Verfahren dazu für bestimmte Anwendungsfälle (z.B. Zustandsautomaten) existieren, sind diese leider nicht allgemein anwendbar.

Des Weiteren ist Testen zu unterscheiden vom Debuggen. Hier ist die Existenz eines Fehlers bereits bekannt. Ziel des Debuggens ist es, Ziel: Fehlerursache und Fehlerort (z.B. im Code) zu finden. Debuggen basiert auf Design und Code. Wie die Erfahrung zeigt, ist die Planung einer Debugphase manchmal schwierig, da Fehler unter Umständen lange gesucht werden müssen. Debuggen wird meist / immer durch Entwickler durchgeführt.

Statischer Test – Dynamischer Test

Testen wird eingeteilt in Statische Tests und Dynamische Tests.

Beim Statischen Test erfolgt kein Ausführen eines Programms und es werden daher keine Testfälle benötigt. Statische Tests erfolgen durch Reviews, ggf. unterstützt durch Checklisten und automatisch durch Tools.

Ziele des statischen Tests sind u.a.

- das Finden von „echten“ Fehlern (z.B. nicht initialisierte Variable, statische Arraygrenzenverletzung)
- das Finden von potentiellen Schwachstellen (z.B. Kombination von signed/unsigned Variablen, Zuweisung von Variablen unterschiedlicher Längen)
- die Überprüfung der Einhaltung von Programmierstandards (z.B. MISRA)
- die Überprüfung der Einhaltung von Programmierkonventionen (z.B. Namensgebung von Variablen und Funktionen)
- Sicherstellen von Portierbarkeit, Wartbarkeit, Verständlichkeit des Codes.
- Statischer Test – Tools (Beispiele)

Tools für statische Tests wie PC Lint oder QAC wurden bereits im vorigen Kapitel erwähnt.

Im Gegensatz dazu steht der Dynamische Test. Hier wird die zu testende Software ausgeführt (ggf. auch Ausführen von Spezifikationen oder Modellen). Für diesen Test werden Testfälle benötigt. Typische Ziele sind u.a.

- das Finden von Fehlern in Algorithmen
- das Finden von Fehlern im Zusammenspiel verschiedener Programmteile
- das Finden von Fehlern im dynamischen Verhalten (Laufzeit, Speicherverbrauch)
- das Sicherstellen von Robustheit
- das Sicherstellen der Funktion über einen längeren Zeitraum
- das Sicherstellen eines Wiederanlaufs (Recovery)

Der Statische und der Dynamische Test ergänzen einander, denn sie sind bzgl. Fehlerfindung teilweise disjunkt. Das bedeutet, dass Fehler die bei statischen Tests einfach gefunden werden können (z.B. eine nicht initialisierte Variable) beim dynamischen Test nicht immer auffällig sind und umgekehrt. Aus diesem Grund sollten beide Testarten immer kombiniert durchgeführt werden, z.B. Codecheck mittels Tools durch den Entwickler, Review durch andere Entwickler und dynamische Tests durch (wenn möglich unabhängige(!) Tester, d.h. Designer / Codierer \neq Tester).

Testfall

Ein wichtiger Begriff des Tests ist der Testfall. Ein Testfall beschreibt einen elementaren, funktionalen Softwaretest, der der Überprüfung einer z.B. in einer Spezifikation zugesicherten Eigenschaft eines Testobjektes dient. Ein Testfall wird mittels Testmethoden (s. Teststrategie, Testtechnik) erstellt. In der Regel sollte ein Testfall einen Bezug zu einer Anforderung haben um die Nachverfolgbarkeit (Traceability) sicherzustellen¹⁸.

Wichtige Bestandteile eines Testfalls sind:

- die Vorbedingungen für die Ausführung des Testfalls
- die Eingaben/Handlungen, die zur Durchführung des Testfalls notwendig sind,
- ggf. Informationen, wie ein verwendetes Testtool zu konfigurieren oder zu steuern ist,
- die erwarteten Ausgaben/Reaktionen des Testobjektes auf die Eingaben, z.B. Resultatausgabe, Ansteuern eines externen Geräts, Schreiben des Speichers,
- die erwarteten Nachbedingungen, die als Ergebnis der Durchführung des Testfalls erzielt werden.
- die Prüfanweisungen, d.h. wie Eingaben an das Testobjekt zu übergeben sind und wie Sollwerte abzulesen sind
- die Dokumentation des Testfalls in Form einer Beschreibung (meist Bestandteil einer Testspezifikation) und das Festhalten des Ergebnis der Durchführung des Testfalls (meist Bestandteil eines Testlogs oder eines Testreports)

Weichen die Ausgaben oder Nachbedingungen während des Testlaufs/der Testdurchführung von den erwarteten Werten ab, so liegt eine Anomalie vor (das kann z.B. ein zu behebender Mangel oder Fehler sein, kann aber auch ein Fehler im Testfall sein).

Unten ist ein Beispiel für ein Requirement und einen zugehörigen Testfall angegeben:
Req.107 (Steuergerät): Falls die Spannung für mindestens 5 Sekunden über 14V steigt, soll ein Überspannungsfehler eingetragen werden.

Ein zugehöriger Testfall könnte folgendermaßen aussehen:

Vorbedingungen sind

- Fehlerspeicher gelöscht
- Steuergerät an Spannungsversorgung 12V angeschlossen

Die durchzuführende Handlung/Eingabe ist

¹⁸ Meist wird der Test eines Requirements mehrere Testfälle erfordern.

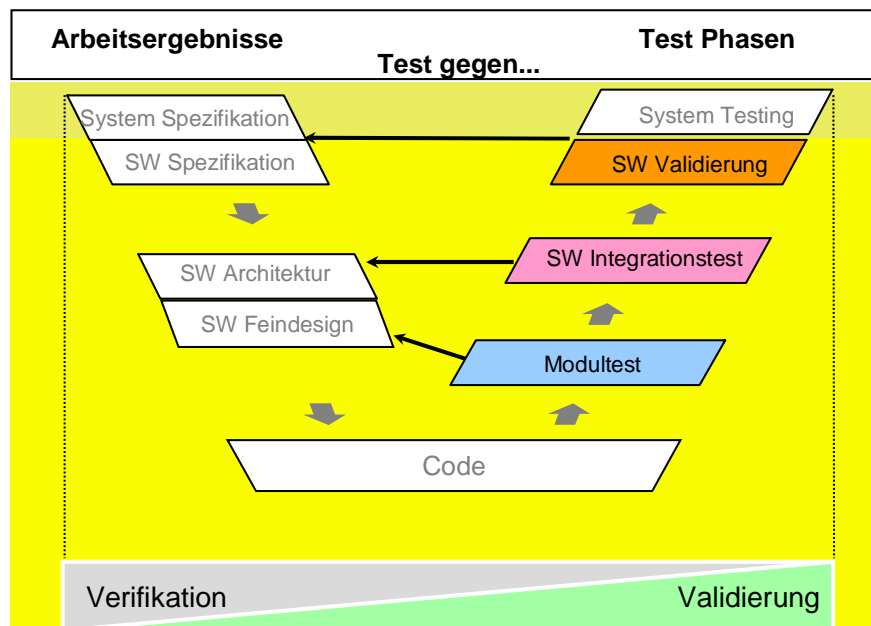
- Spannung für 5 Sekunden auf 14,1V einstellen
- Fehlerspeicher auslesen

Die erwartete Ausgabe (Sollwert) für den Testfall ist

- Überspannungsfehler ist eingetragen

Testphasen

Der Testprozess eines Projekts gliedert sich in Testphasen auf. Drei Testphasen werden unterschieden:



Die erste Phase ist der Modultest als Test einer individuellen Komponente. Die zweite Phase ist der Integrationstest mit dem Zusammenfügen und Testen bereits (modul-) getesteter SW-Komponenten. Die dritte Phase ist die SW Validation, die zeigen soll, dass die Software Anforderungen nicht erfüllt.

Zwie Begriffe Verifikation und Validierung, die im Zusammenhang mit Software Test oft verwendet, sollen hier noch erläutert werden:

Verifikation bedeutet Test einer Software gegen die Ergebnisse der vorhergehenden Phase(n). Zum Beispiel wird beim Modultest gegen das Feindesign getestet (theoretisch völlig unabhängig von der Spezifikation). Validierung heißt Test gegen die Spezifikation bzw. gegen die Erwartungen des Anwenders. In der Praxis lassen sich Verifikation und validierung nicht völlig disjunkt durchführe, In frühen Testphasen überwiegt der Verifikationsanteil, in späten Phasen der Validierungsanteil

Zum Merken und vereinfacht gesagt ist Verifikation das Sicherstellen, dass das Produkt richtig entwickelt wurde. Validierung dagegen stellt sicher, dass das richtige Produkt entwickelt wurde. Beide Aspekte sind wichtig. Wurde das falsche Produkt entwickelt, wird es keinen Abnehmer finden. Wurde das Produkt falsch entwickelt, wird es fehlerhaft sein und die Entwicklungskosten werden nicht eingehalten.

Wichtig und zu beachten ist (!) , dass sich die Testphasen in der obigen Darstellung nur (!) auf die Ausführung bzw. Auswertung des Tests („rechter“ Teil des „V“) beziehen. Die Spezifikation und Implementierung der Tests werden während Spezifikation, Design und Codierung durchgeführt („linker Teil des „V“). Wird mit der Testspezifikation erst nach der Fertigstellung des Codes begonnen, ist eine termingerechte Durchführung des Tests nicht mehr möglich. Darüber hinaus erlaubt eine frühe Spezifikation der Tests eine implizite Kon-

trolle und Überprüfung der Requirement und Designdokumente, auf deren Basis der Test definiert wird.

Des Weiteren ist zu beachten, dass Tests werden während des kompletten „V“s gemanagt werden müssen (z.B. Planung, Verfolgung, Konfigurationsmanagement)

Test Phasen – Modultest

Wie schon oben erwähnt ist der Modultest der Test einer individuellen SW Komponente (e.g. C-Funktion, Subroutine), wobei die Komponente u.U. in weitere Unterkomponenten zerlegt werden kann. Wenn möglich soll die Komponente in Isolation (!) getestet werden. D.h. die Umgebung (andere Module, die zum zu testenden Modul in einer Beziehung stehen) wird simuliert. Ein Synonym für den Modultest ist der Unitest.

Das Ziel des Modultests ist es, lokale Fehler und Abweichungen vom (Fein-) Design zu finden.

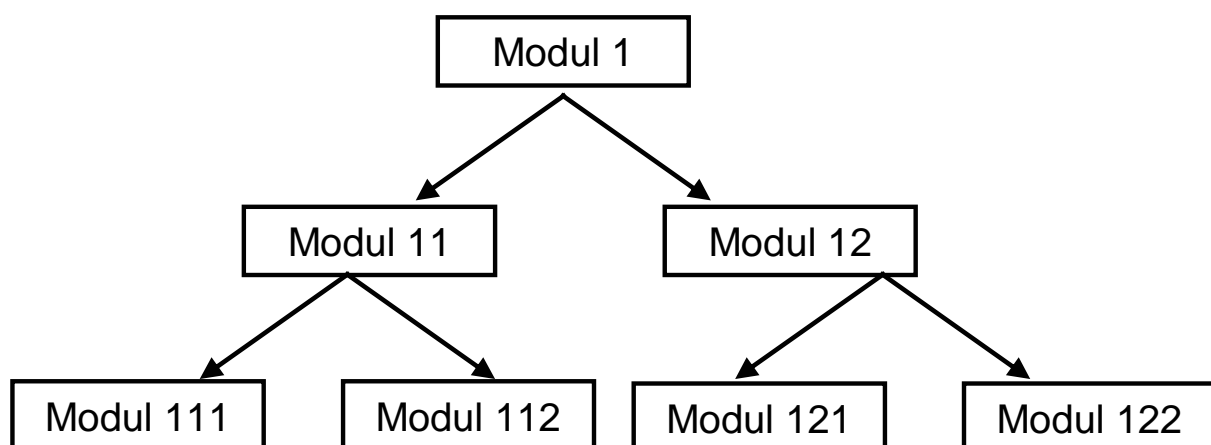
Beim Modultest werden Testtreiber benötigt, die das zu testende Modul aufrufen. Simulationen der Umgebung (z.B. aufgerufene Module) durch Stubs (Dummies) müssen durchgeführt werden, da das Modul ohne seine Umgebung getestet werden soll. Ein Stub ersetzt also das Original nur für die durchgeführten Testfälle. Stubs müssen u.U. auch getestet werden und können sehr aufwendig werden. Im Extremfall so aufwendig, wie das zu ersetzende Modul

Test Phasen – Integrationstest

Integrationstest ist der Prozess des Zusammenbauens und Testens bereits getesteter (!) Module. Die SW Architektur bildet die Basis der SW Integration, d.h. sie ist ein Test gegen die SW Architektur, d.h. bzgl. interner Schnittstellen und Abläufe. Dies können z.B. sein Parameterwerte bei Aufrufen von Funktionen, Werte globaler Variablen, bestimmte definierte Aufrufreihenfolgen und auch das Zeitverhalten. Ohne dokumentierte SW Architektur ist ein sinnvoller Integrationstest nicht möglich!

Zwie gängige Verfahren zum Integrationstest sind die Top-Down und Bottom Up Integration. Diese sollen an folgendem, einfachem Beispiel erläutert werden:

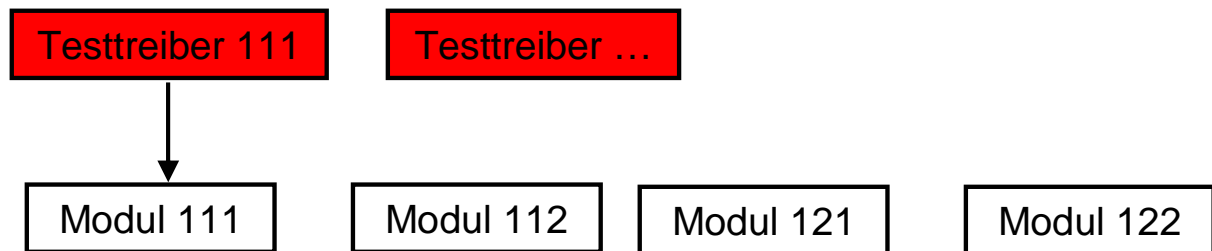
Ein Programm besteht aus 7 Modulen, die wie Baumartig voneinander abhängen. Das Modul 1 auf oberster Ebene (in C das main Module) ruft zwei weitere Module Modul 11 und Modul 12 auf. Diese wiederum rufen jeweils zwei weitere Module auf, Modul 11 ruft die Module 111 und 112 und Modul 12 ruft die Module 121 und 122 auf.



Bei der Bottom Up Integration wird mit den Modulen, die auf unterster Ebene sind (d.h. keine weiteren Module aufrufen), begonnen.

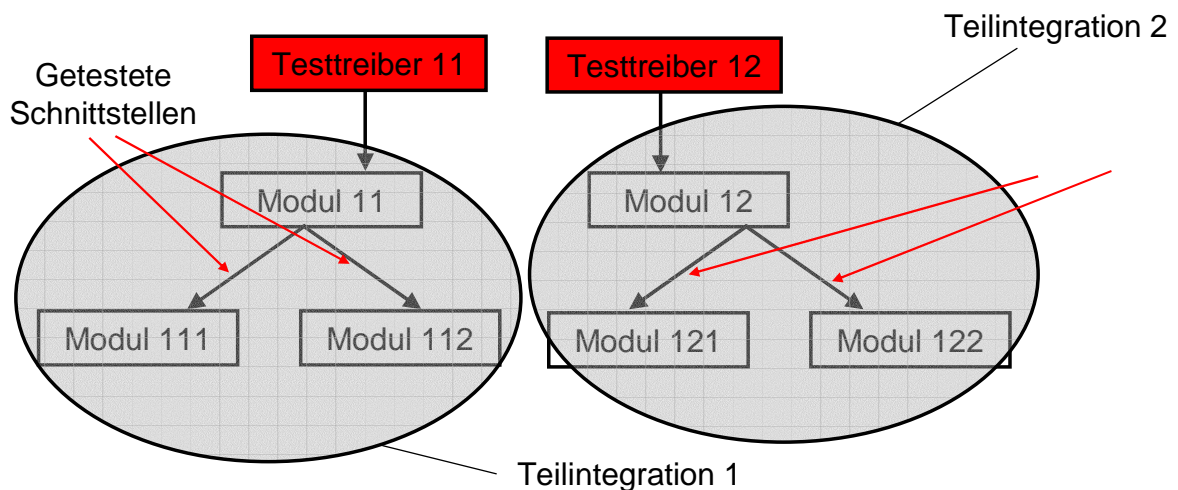
1. Schritt:

(Modul-)Test derjenigen Module, die keine weiteren aufrufen. Benötigt werden Testtreiber, die den Aufrufer und ggf. die Umgebung simulieren



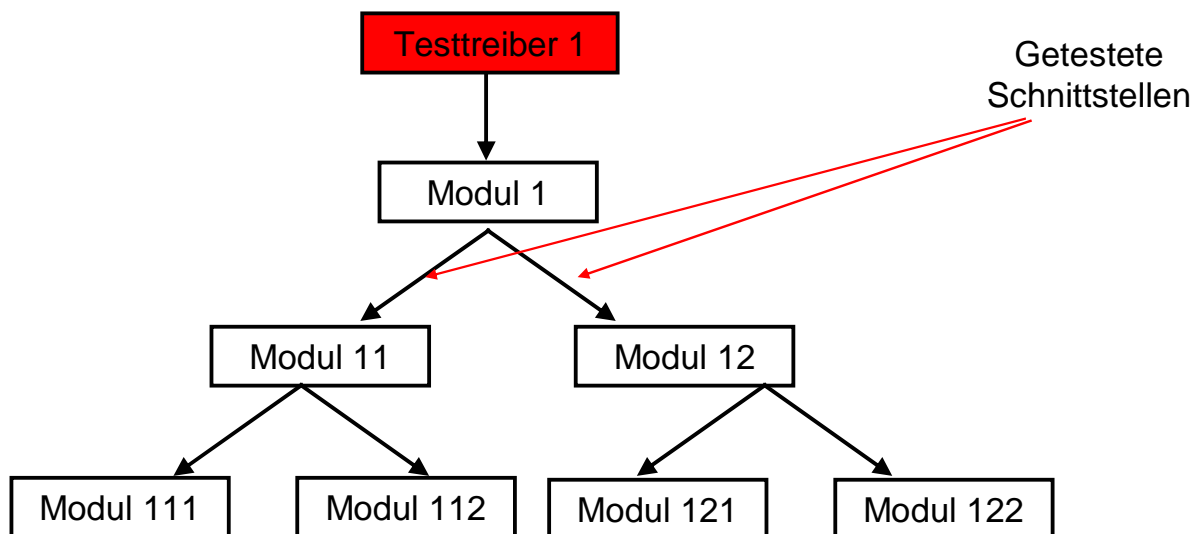
2. Schritt:

Im 2. Schritt werden jeweils das Modul 11 mit den Modulen 111 und 112 und das Modul 12 mit den Modulen 121 und 122 integriert und getestet. Benötigt werden wieder Testtreiber. Fokus des Integrationstests sind die Schnittstellen zwischen den Modulen.



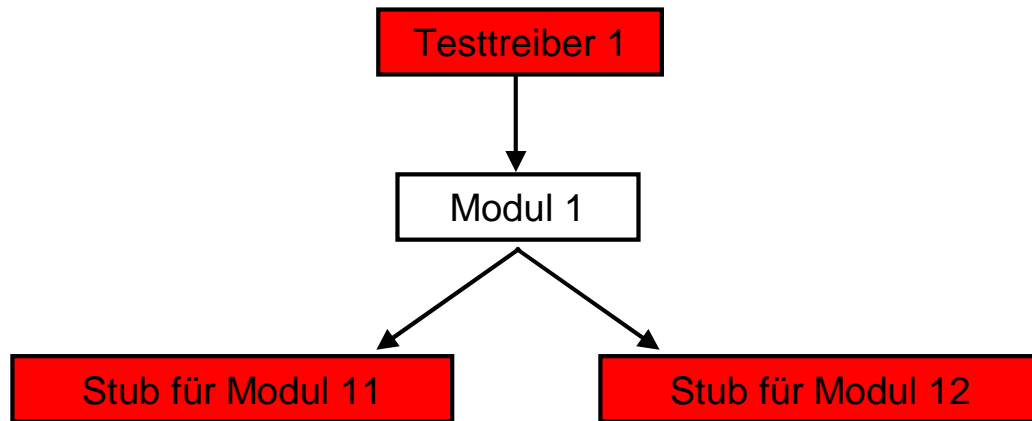
3. Schritt:

Integration der Ergebnisse des letzten Integrationsschrittes zum Gesamtprogramm. Test der Schnittstellen zwischen den Modul 1 und den M1 aufgerufenen Modulen 11 und 12.

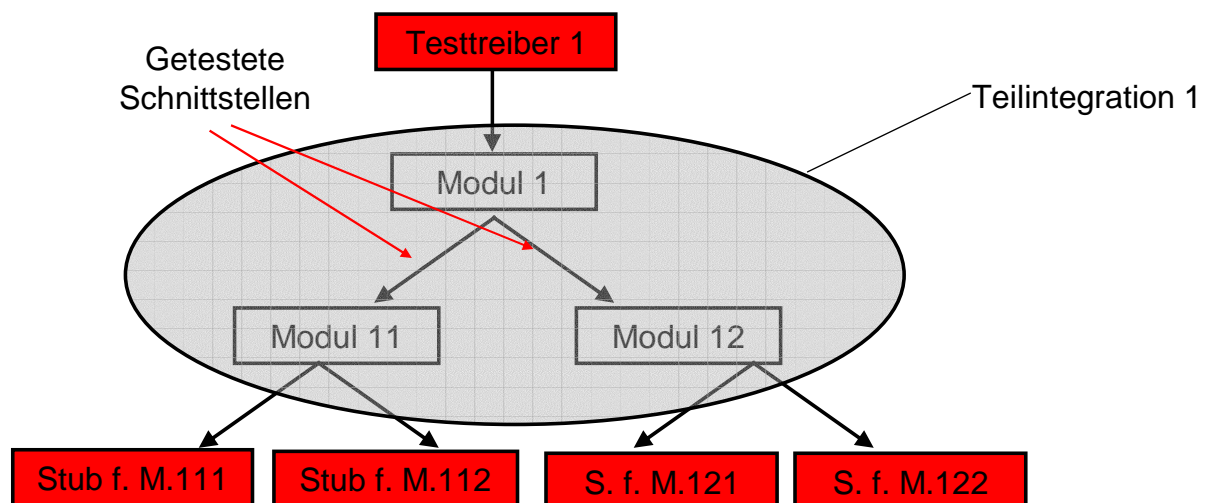


Bei der Top-Down Integration wird mit dem Modul auf höchster Ebene, d.h. mit dem Modul, das keine weiteren Aufrufer über sich hat.

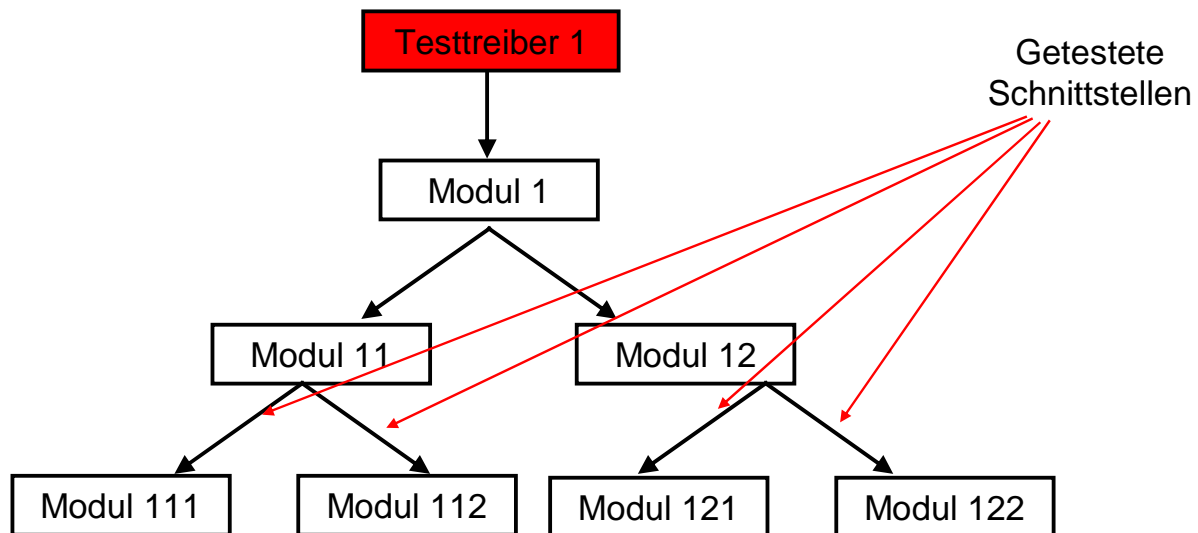
1. Schritt:
(Modul-)Test derjenigen Module, die von keinem anderen aufgerufen werden (main).
Benötigt werden Testtreiber, die den Aufrufer und Stubs, die die aufgerufenen Module 11 und 12 simulieren



2. Schritt:
Integration der Module 11 und 12, die bisher durch Stubs ersetzt wurden. Module 111, 112, 121 und 122, die von den integrierten Modulen aufgerufen werden, werden wiederum simuliert. Getestet werden die Schnittstellen zwischen Modul 1 und den von 1 gerufenen Modulen.



3. Schritt:
Integration der zuletzt „gestubten“ Module 111, 112, 121, 122 zum Gesamtprogramm.
Test der Schnittstellen zu den neu integrierten Modulen.



Jedes Integrationsverfahren hat Vor- und Nachteile.

Bei der Top-Down-Integration liegt das Augenmerk auf Kontrollfluss. Das Grundgerüst des Programms, das in den „höheren“ Modulen implementiert ist, ist frühzeitig verfügbar und damit testbar. Daneben können mit dem vorhandenen Grundgerüst frühzeitig Vorführungen des Programms, z.B. für einen Kunden durchgeführt werden. Nachteilig bei der Top-Down-Integration wirkt sich der Aufwand für Stubs und die fehlende Flexibilität (es muss immer mit dem obersten Modul begonnen werden) bei der Integration aus.

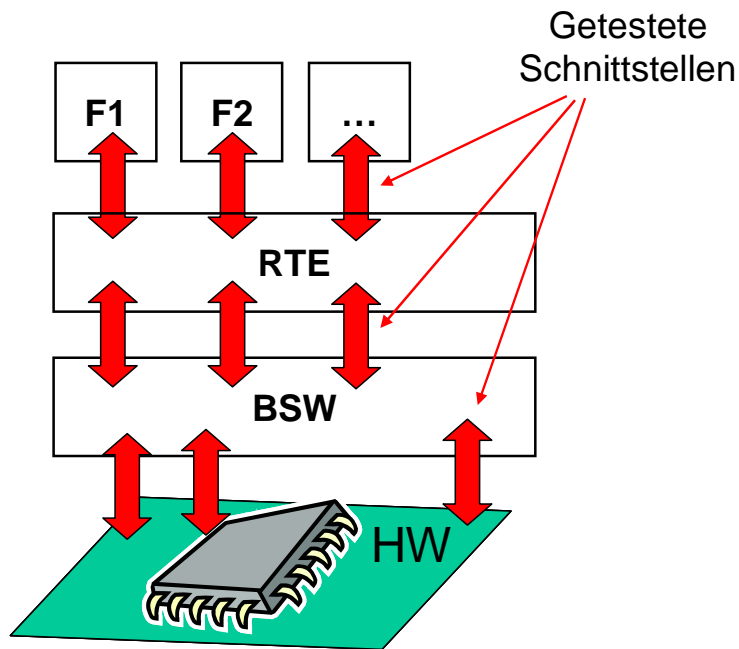
Im Gegensatz dazu liegt beim Bottom-Up-Vorgehen das Augenmerk auf Komponenten, die frühzeitig integriert und getestet werden können, z.B. bei komplexen, kritischen Treibern. Höhere Komponenten müssen simuliert werden durch Testtreiber. Ein Vorteil ist die Flexibilität im Ablauf, d.h. auf unterster Ebene kann mit der Integration der bereits vorhandenen Module begonnen werden. Nachteilig wirkt sich u.U. aus, dass der Gesamtablauf des Programms erst am Ende verfügbar und sichtbar ist.

Anmerkungen:

1. Vermeiden werden sollte die so genannte Big-Bang-Integration. Hierbei handelt es sich um einen einzigen Integrationsschritt für das komplette Programm. Problematisch ist hier eine mögliche Fehlerverdeckung, die Vielzahl der vorhandenen Schnittstellen und möglichen Abläufe.
2. Die Bottom-Up- und Top-Down-Integration lassen sich in der Praxis meist nicht vollständig durchführen. Der Grund liegt in der Vielzahl der Module, deren einzelne Integration einen zu hohen Aufwand erfordern würde und im Aufwand Stubs zu erstellen. Darüber hinaus sind Aufrufbeziehungen oft nicht so klar strukturiert wie oben gezeigt. In der Praxis sind daher Teile vorzuintegrieren, die dann zum Gesamtprogramm integriert werden können. Dies kann z.B. schichtweise erfolgen, wie im folgenden Beispiel erläutert:

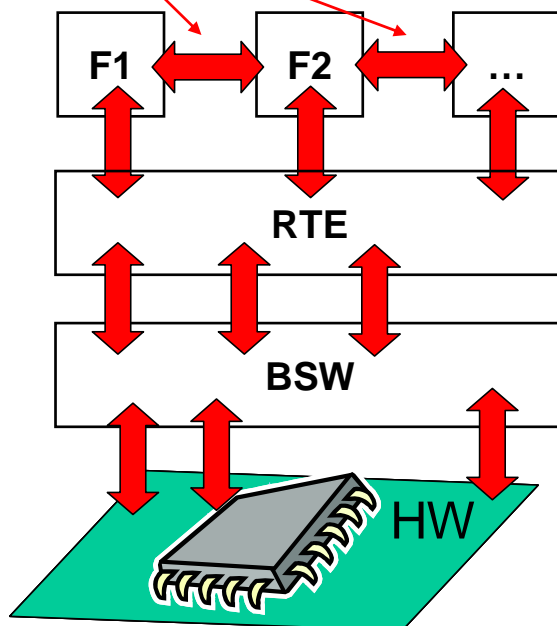
Beispiel Test Phasen – Schichtweise Integration

Im Embedded Software Umfeld (z.B. Automobiltechnik) können Integrationen schichtweise von unten (von der HW) nach oben (zur Applikation) durchgeführt werden.



Im 1. Schritt wird das Betriebssystem (BSW = HW-nahe SW) auf die Hardware integriert. Dazu gehören z.B. Scheduler, Memory Management, Input/Output-handling, Kommunikation wie CAN, LIN, Hardware-Treiber. Getestet wird das Interface zwischen Hardware und Software. Im 2. Schritt wird das Runtime Environment (RTE) integriert. Getestet werden die Schnittstellen zwischen der BSW und der RTE. Im 3. Schritt werden Applikations-SW Funktionen auf das RTE gesetzt. Diese Applikationsfunktionen wiederum können schrittweise integriert werden, wobei sowohl die Schnittstellen zwischen RTE und Applikationen und zwischen den Applikationen getestet werden.

Schrittweise Integration der Applikationsfunktionen F1, F2, ... Test der Schnittstellen zwischen den Funktionen



Test Phasen – Validierung

Die SW Validierung (Validation) ist der Test der kompletten SW gegen die Anforderungen. Der Test sollte (theoretisch) unabhängig von der Hardware (speziell im Embedded Bereich) durchgeführt werden, auch wenn der Test auf der Zielhardware läuft. In der Praxis lässt sich die Unabhängigkeit nicht immer vollständig erreichen.

Abgrenzung Software Test – Systemtest

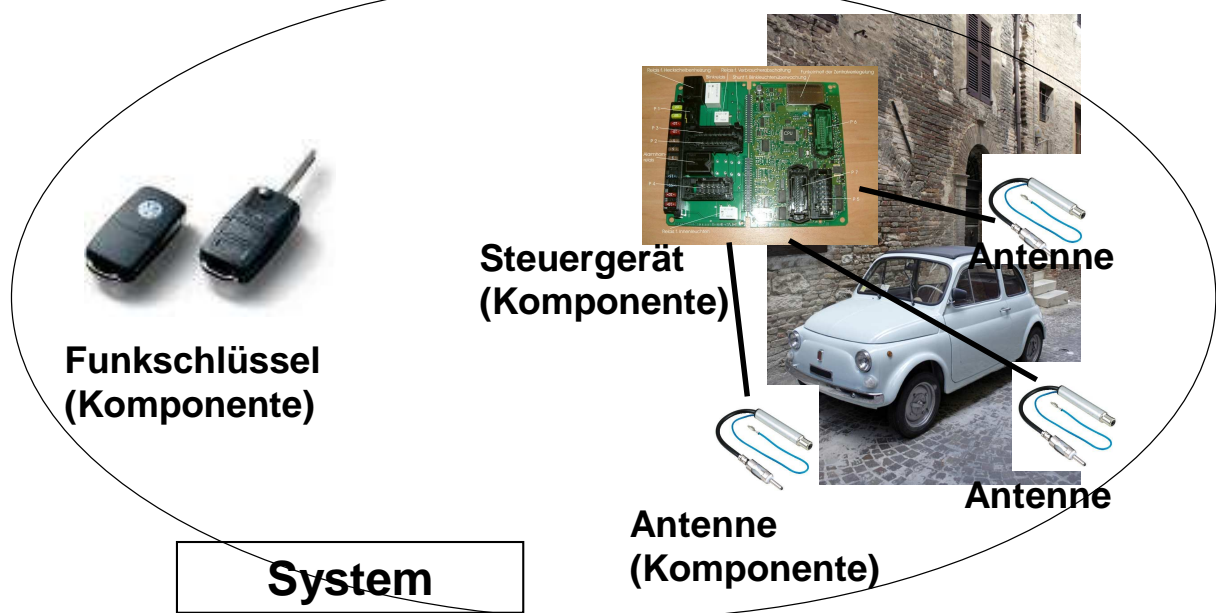
Neben bzw. nach den Software Tests gibt es oft weitere Testphasen (speziell im Embedded Bereich). Dazu gehören

- die schon erwähnte HW-SW Integration, die Integration einer Komponente, wobei das Zusammenspiel zwischen HW, SW und Mechanik getestet wird
- die Komponentenvalidierung, d.h. der Test gegen die Anforderungen der Komponente. Diese sind oft ähnlich den SW Anforderungen oder SW Anforderungen werden von Komponentenanforderungen abgeleitet.
- die Systemintegration als Test des Zusammenspiels von Systemkomponenten
- der Systemtest als Test eines kompletten Systems gegen die Systemanforderungen

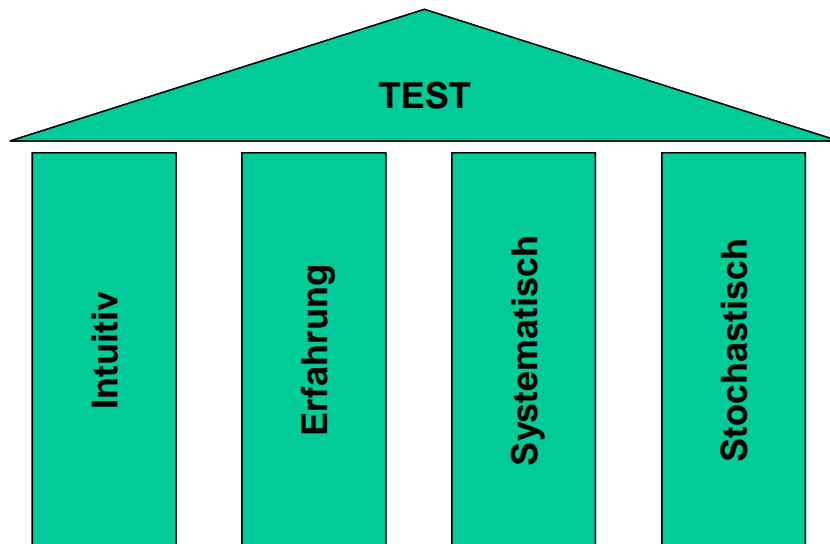
In der Praxis ist eine genaue technische und terminliche Trennung zwischen diesen verschiedenen Testphasen manchmal schwierig.

Im Beispiel unten sind Schlüssel, Antennen und Steuergerät Komponenten, die das Fernbedienungssystem bilden.

Beispiel System - Komponente



Testtechniken und Teststrategien



Testtechniken und Teststrategien stützen sich auf mehrere Säulen. dazu gehören Intuition des Testers, Erfahrung aus laufenden oder früheren Projekten, Systematik in form formaler Verfahren zum Test und ergänzend stochastische Verfahren. Diese werden im folgenden weiter erläutert.

Intuitives Testen (Error Guessing)

Die Spezifikation von Testfällen basiert hier auf einem intuitiven Ansatz, d.h. nicht nach einem formalen Verfahren. Entscheidend ist die Erfahrung und das Gespür des Testers Fehler aufzuspüren (Error Guessing). dabei können die „merkwürdigen Fehler“ gefunden werden, die mittels formaler Methoden evtl. unentdeckt bleiben.

Intuitive Tests können eingesetzt werden anstatt eines systematischen Tests z.B. für den Test eines Prototyps oder zusätzlich zu systematischen Tests. Die Wirksamkeit intuitiver Tests sollte nicht unterschätzt werden.

Wichtig ist, dass intuitiver Tests dokumentiert werden und nichts mit "Ad Hoc Testen" ohne spezifizierte Testfälle und dokumentiertes Testergebnis zu tun hat.

Erfahrung (Lessons learned)

Testen basierend auf Erfahrungen ist nicht zu verwechseln mit intuitivem Testen. Hier geht es um bereits gemachte (negative oder auch positive Erfahrungen), die formal dokumentiert wurden. Die Dokumentation kann erfolgen in form von Checklisten, die die zu beachtenden Punkte enthält oder auch in Fehlerverfolgungssystemen (Fehlertracking). Projektspezifische Themen können z.B. zusätzlich auch aus Review Ergebnissen kommen.

Stochastische Tests / Stresstests

Stress Tests versuchen, wieder der Name schon sagt, die zu testende Software extremen Bedingungen zu unterwerfen. Dies ist wichtig insbesondere bei kritischen Laufzeiten. Stress-tests werden Test mit hoher Last durchgeführt. Dies kann eine hohe Anzahl von Interrupts sein, eine hohe Kommunikationsfrequenz oder das gleichzeitiges Triggern von Inputs sein. Tests mit Überlast prüft die Robustheit der Software.

Beispiele im Bereich Automotive sind das Schalten der Zündung (KL 15) 100 mal in 10s schalten oder eine hohe Interruptlast über CAN.

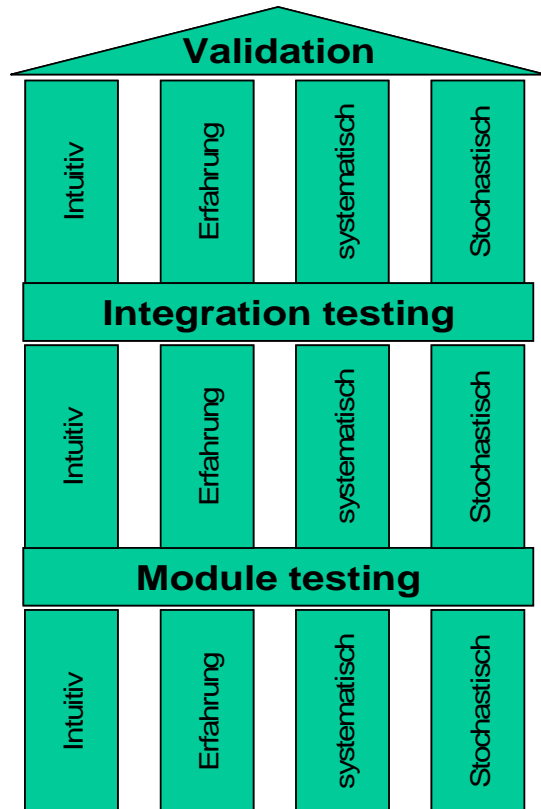
Systematische Tests

Systematische Tests verwenden bestimmte definierte systematische Verfahren um Testfälle zu definieren. Die Systematik stellt sicher, dass sich die Ermittlung der Testfälle wiederholen

lässt. Das Ziel ist hier, die “offensichtlichen” Fehler finden. Die Verfahren zur Ermittlung von Testfällen werden weiter unten im Detail erläutert.

Das Testgebäude

Alle Testtechniken über die Testphasen betrachtet bilden das „Testgebäude“ für ein Projekt



Test Techniken im Detail: Überblick

Bei den Testtechniken wird grundsätzlich unterschieden zwischen Black-Box- und White-Box-Techniken. Black.Box-Tests (oder funktionale Tests) basieren auf der Spezifikation der zu testenden Software. Die Art der Implementierung und die internen Abläufe der Software (d.h Architektur, Feindesign) spielen keine Rolle. Im Gegensatz dazu stützen sich White-Box-Tests auf interne Abläufe und Strukturen der Software (Architektur, Feindesign). Dies können, wie wir noch sehen werden, Kontrollflüsse sein aber auch Datenflüsse durch das Programm. Zu den Black.Box-Tests gehören u.a. Äquivalenzklassenanalyse, Grenzwertanalyse, Zustandsbasiertes Testen, Interface Testen, Syntax Testen. Zu den White-Box-Tests gehören u.a. Statement Testen, Zweig-/Decision Testen, Pfad-Testen, Datenflussbasiertes Testen und Real-Time Testen.

Beim Black Box Test wird noch unterschieden zwischen Positivem Test und Negativem Test. Ein Positiver Test ist ein Test unter den spezifizierten Bedingungen, während der Negative Test die SW unter nicht spezifizierten Bedingungen (z.B. falscher Input, extreme Zeitbedingungen) testet. Negative Tests sind wichtig für Robustheitstests. Die wichtigsten Testtechniken werden weiter unten erläutert. Zuvor werden noch die Begriffe Testvollständigkeitskriterium und Testendekriterium eingeführt.

Testvollständigkeitskriterium

Eine Teststrategie muss ein formales Testvollständigkeitskriterium beinhalten, das festlegt, wann der Test beendet ist. Ein Testvollständigkeitskriterium muss messbar, formal definiert

und dokumentiert sein (spätestens bei Testbeginn). „Zu wenig Zeit¹⁹“, „zu wenig Geld“, „wir benötigen die Software in zwei Stunden“ sind keine formalen Kriterien! Der Erfüllungsgrad eines Testvollständigkeitskriteriums kann als Metrik für den Testfortschritt verwendet werden.

Ein Test muss ein formales Testendekriterium beinhalten, das festlegt, wann der Test beendet ist und die Software freigegeben werden kann:

- Das oder die Testvollständigkeitskriterien der verwendeten Testtechniken sind erfüllt
- es liegen keine kritischen Fehler vor
- nur eine definierte Zahl nicht kritischer Fehler ist übrig geblieben

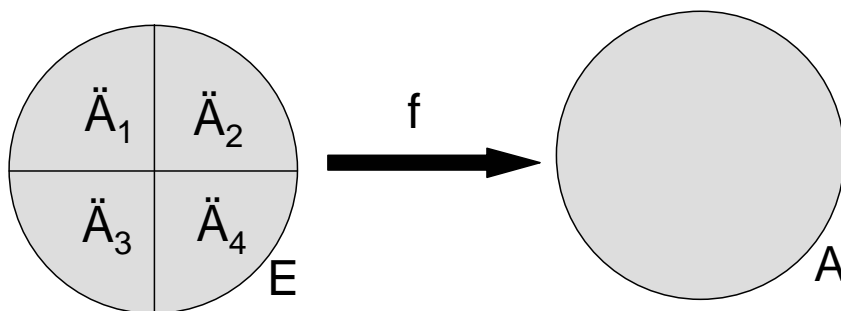
Anmerkungen: In der Praxis kann es durchaus vorkommen, dass bei einem Test geplante Testvollständigkeitskriterien und / oder geplante Testendekriterien erreicht wurden. In diesem Fall ist das Risiko für die Freigabe der getesteten Software sorgfältig abzuschätzen.

Test Techniken - Black Box

Äquivalenzklassenanalyse (Equivalence Class Analysis)

Die Äquivalenzklassenanalyse ist eine Testtechnik, die angewendet wird für funktionale Tests. Die zu testende Software realisiert eine Funktion f , die aus Eingabedaten x eine Berechnung durchführt und ein Ergebnis y bereitstellt.

D.h. $y = f(x)$. Für die Äquivalenzklassenanalyse wird zunächst der Eingabedatenraum E in Teilmengen (Äquivalenzklassen) zerlegt, so daß Elemente aus einer Äquivalenzklasse durch die Funktion F auf identische oder zumindest ähnliche (d.h. äquivalente) Art und Weise behandelt werden. Es kann auch Äquivalenzklassen geben, die illegale Werte enthalten. Bei Test wird nun die Funktion f mit Vertretern aus allen Äquivalenzklassen getestet (mindestens ein Element aus jeder Äquivalenzklasse).



Ein Testfall wird erstellt, in dem ein Vertreter v einer Äquivalenzklasse ausgewählt wird, der Sollwert ($s=f(v)$) wird ermittelt, die zu testende Software wird mit der Eingabe v ausgeführt und das Ergebnis mit dem Sollwert verglichen.

Auf Grund der Annahme, dass alle Vertreter einer Äquivalenzklasse auf identische (oder sehr ähnliche) Weise durch die Funktion f verarbeitet werden, entsteht die Vermutung oder Hoffnung, dass ein Fehler mit jedem beliebigen Vertreter einer Äquivalenzklasse gefunden wird. Anders herum ausgedrückt, wenn ein oder mehrere Vertreter einer Äquivalenzklasse beim Test korrekt verarbeitet werden, werden (hoffentlich) alle Elemente einer Äquivalenzklasse ordnungsgemäß verarbeitet.

Hat eine Funktion mehrere Parameter, so muss die Analyse für jeden einzelnen Parameter ausgeführt werden. Für die Testdurchführung sind dann geeignete Kombinationen aus allen möglich Klassen auszuwählen.

Die Äquivalenzklassenanalyse kann auch analog basierend auf dem Ausgabedatenbereich einer Funktion durchgeführt werden.

¹⁹ Zu wenig Zeit heißt hier, dass für den Test ursprünglich mehr Zeit geplant war, die inzwischen durch den Verzögerungen im Projektverlauf anderweitig verbraucht wurde.

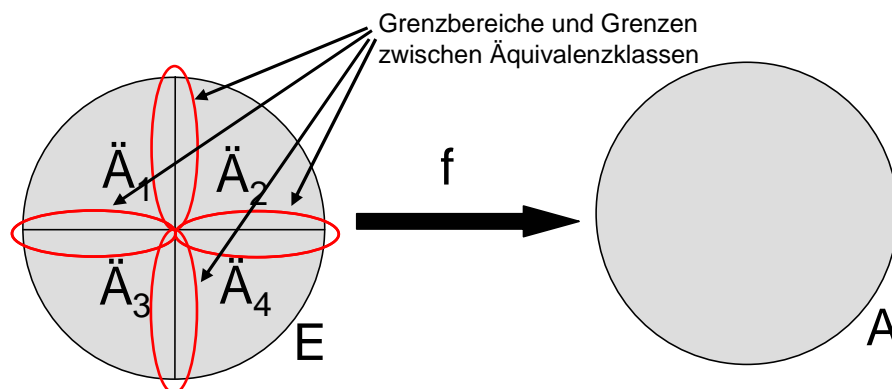
Als Testvollständigkeitskriterium wird aus jeder Äquivalenzklasse mindestens ein Vertreter getestet (in der Praxis sollte natürlich mehr als ein Vertreter ausgewählt werden).

Beispiele für Äquivalenzklassenzerlegungen:

- Gehaltsabrechnung: Unterscheidung zwischen Arbeitern, Angestellten und Leitenden
Angestellten, die bei der Gehaltsberechnung unterschiedlich behandelt werden, führt zu drei Äquivalenzklassen.
- Spannungsbereiche (Unterspannung $0 < 7V$, Normalspannung $7V \leq 14V$, Überspannung $> 14V$)
- Drehzahlbereiche
- Eingabeschalter (ein/aus)
- Temperaturbereiche

Grenzwertanalyse (Boundary value Analysis)

Die Grenzwertanalyse ergänzt die Äquivalenzklassenanalyse dahingehend, dass zusätzlich zu Testdaten aus den einzelnen Klassen gezielt Testdaten aus den Grenzbereichen der Klassen und, falls möglich, die Grenzbereiche selbst ausgewählt werden.



Grenzbereiche sind nicht immer sinnvoll identifizierbar. So gibt es zwischen Arbeitern, Angestellten und Leitenden Angestellten keine Grenzbereiche, da diese Klassen disjunkt sind und keine „stetigen Übergänge“ besitzen. Bei Zerlegungen, die Zahlenbereiche (Beispiel Betriebsspannungen) in Klassen aufteilen, lassen sich hingegen Grenzbereiche und Grenzwerte sinnvoll identifizieren.

Als Testvollständigkeitskriterium wird aus jeder Äquivalenzklasse mindestens ein Vertreter getestet (in der Praxis wird mehr als ein Vertreter ausgewählt werden) und jeweils Testdaten an den Grenzen und auf den Grenzen zwischen den Äquivalenzklassen.

Die Zerlegung in Äquivalenzklassen mit zugehörigen Grenzwerten muss nicht immer eindeutig sein, wie das folgende Beispiel zeigt, sondern bleibt gegebenenfalls dem Tester überlassen.

Es soll eine C-Funktion getestet werden, die aus einem gegebenen Datum den zugehörigen Wochentag berechnet (z.B. 16.7.2007 ist ein Montag). Zusätzlich sollen falsche Daten (30.2.2007) mit einer Fehlermeldung quittiert werden. Folgende Schnittstelle der Funktion als C-Code liegt vor:

```
enum wochentag (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag, Error)
wochentag Datum_zu_Wochentag (int Tag, int Monat, int Jahr)
```

1. Möglichkeit Äquivalenzklassenzerlegung und Grenzwertanalyse:

- Äquivalenzklassen (gültige und ungültige Bereiche) für Tag $[-\infty .. 0]$, $[1 .. 31]$, $[32 .. \infty]$

- Äquivalenzklassen (gültige und ungültige Bereiche) für Monat $[-\infty .. 0]$, $[1 .. 12]$, $[13 .. \infty]$
- Äquivalenzklasse für Jahr $[-\infty .. \infty]$
- Grenzwerte für Tag: 0, 1, 31, 32
- Grenzwerte für Monat: 0, 1, 12, 13
- Grenzwerte für Jahr: keine
- Weitere Randbedingungen z.B.: 31 Tage im Januar, März, ..., 30 Tage im April, Juni, ..., 28/29 Tage im Februar

2. Möglichkeit Äquivalenzklassenzerlegung und Grenzwertanalyse:

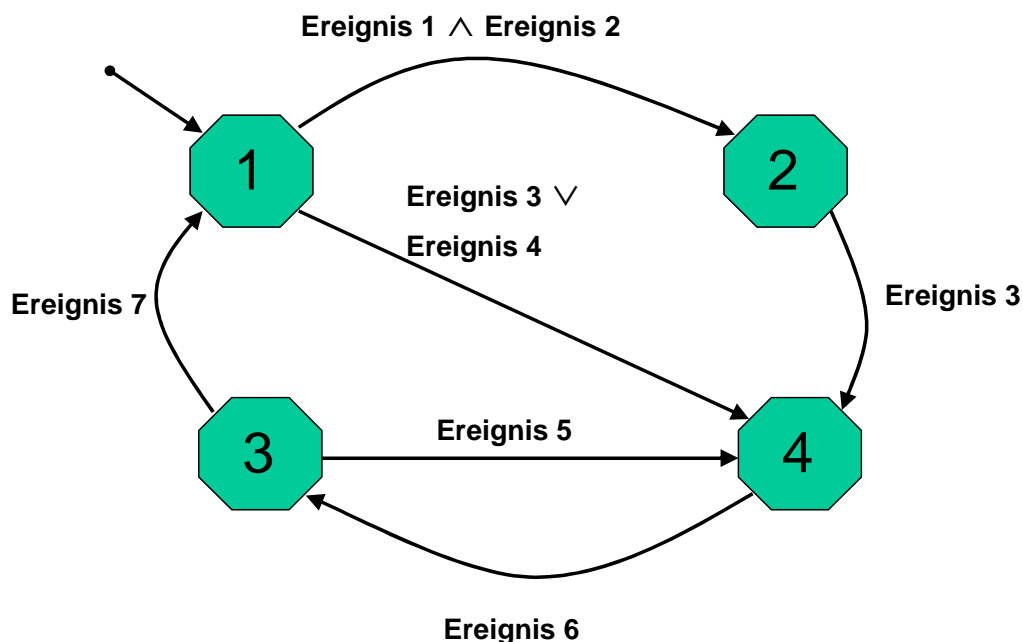
- Äquivalenzklassen für Tag: $[-\infty .. 0; 32 .. \infty]$, $[1 .. 28]$, $[29]$, $[30]$, $[31]$
- Äquivalenzklassen für Monat: $[-\infty .. 0]$, $[1 .. 12]$, $[13 .. \infty]$
- Äquivalenzklasse für Jahr: $[-\infty .. -1]$, $[0]$, $[1 .. \infty]$
- Grenzwerte für Tag : 0, 1, 28, 29, 30, 31, 32
- Grenzwerte für Monat : 0, 1, 12, 13
- Grenzwerte für Jahr : -1, 0, 1
- Weitere Randbedingungen wie oben

Zustandsbasierter Test

Basis dieses Tests ist eine Spezifikation der zu testenden Software, die in Form eines Zustandsmodells (= Zustandsautomat) vorliegt. D.h. das Zustandsmodell spezifiziert das Verhalten der Software. Getestet werden die Zustandsübergänge (Transitionen), ausgelöst durch die Ereignisse, die die Übergänge auslösen und die Aktionen, die auf Grund der Transitionen ausgeführt werden. Testfälle werden auf folgende Art definiert:

- der Ausgangszustand der zu testenden Software
- das Ereignis, das einen Zustandsübergang auslöst
- die erwartete Reaktion (z.B. Ausführen einer Aktion)
- der erwartete Folgezustand

Als Testvollständigkeitskriterium wird eine 100% Überdeckung des Zustandsautomaten (d.h. das Ausführen aller möglichen Transitionen ausgelöst durch alle möglichen Ereignisse) definiert. Als Beispiel sei der folgende Zustandsautomat gegeben:



Eine Zustandsüberdeckung kann erreicht werden z.B. durch folgende Ereignisfolge: Start der Automaten, Ereignis 1 und 2, Ereignis 3, Ereignis 6, Ereignis 7, Ereignis 3, Ereignis 6, Ereignis 5.

Damit wäre der Automat einmal "durchlaufen". Bei Übergängen, die von mehreren Ereignissen unabhängig voneinander ausgelöst werden können (oder-Verknüpfung) sollte allerdings jedes Ereignis einmal getestet werden.

Der Zustandsbasierte Test kann in allen Testphasen (Modultest, Integrationstest und Validierung) verwendet werden.

Schnittstellentest

Der Schnittstellentest wird ausschließlich angewandt beim Integrationstest. Er soll sicherstellen, dass die zu integrierenden Softwareteile an ihren Schnittstellen richtig zusammenarbeiten. Der Schnittstellentest basiert auf SW Architektur bzw. Feindesign, die die Schnittstellen zwischen SW Komponenten im Detail beschreiben (z.B. Definition, Inhalt, Ablauf, Timing).

Testfälle werden wie folgt definiert. Zunächst werden zu integrierenden Komponenten ausgewählt, die einem Schnittstellentest unterzogen werden sollen. Meist wird es nicht möglich sein, alle Komponenten auszuwählen. Das bedeutet, dass ggf. Komponenten mit kritischen, wichtigen oder komplexen Schnittstellen getestet werden. Danach erfolgt die Auswahl der zu testende(n) Schnittstelle(n) zwischen diesen Komponenten. Es sind die für die Testfälle notwendigen erwarteten Ergebnisse aus Software Architektur und Feindesign festzulegen. Dies können sein:

- Datenbereiche und Typen der Schnittstelle(n)
- Art und Anzahl der Parameter an der Schnittstelle
- Die Methode der Datenübergabe
- Die Abläufe für den Schnittstellentest

Während der Test ausgeführt wird, d.h. die zu testende Software wird zum Ablauf gebracht, werden die zu testenden Schnittstellen beobachtet. Dies kann mit Hilfe eines Werkzeugs (z.B. Debugger) geschehen oder durch zusätzlich zu implementierenden Code.

Das Testvollständigkeitskriterium beim Schnittstellentest ist formal schwer festzulegen. Theoretisch sollten alle Schnittstellen der zu integrierenden Software getestet werden. Dies ist praktisch nicht durchführbar auf Grund der Vielzahl der Schnittstellen in größeren Programmen. In der Praxis sollte daher eine (intuitive) Auswahl wichtiger, kritischer, komplexer Schnittstellen durchgeführt werden und der Test dieser Auswahl als testspezifisches Vollständigkeitskriterium definiert werden.

Anmerkung: Der Schnittstellentest basiert auf der internen Struktur der Software, insbesondere auf der Architektur. Daher ist dieser Test kein „reiner“ Black-Box-Test. Da aber die integrierten Komponenten selbst als Black-Boxes betrachtet werden, wird er hier eingeordnet.

Test Techniken - White Box

Statement Test (

Der Statement Test basiert als White-Box Test auf der Struktur und dem Code der Software. Ziel ist es, beim Test möglichst viele Statements²⁰ auszuführen. Ein Testfall besteht somit aus einer Menge von Eingabedaten, der Menge der Statements, die auszuführen sind und der erwartete Output.

Als Testvollständigkeitskriterium ist meist eine 100% Abdeckung aller Statements im Code definiert. Diese wird auch als C₀ Abdeckung bezeichnet. Eine 100% Statement Abdeckung stellt folgendes sicher:

- Es gibt kein Statement, das immer bei seiner Ausführung zu einem Fehler führt
- Jedes Statement hat mindestens einmal funktioniert
- Es gibt keinen ungetesteten Code
- Es gibt keinen toten (d.h. nicht erreichbaren) Code

²⁰ mit "Statements" sind hier Sourcecode Statements gemeint.

Der Statement Test (wie alle anderen folgenden White-Box Testtechniken) benötigen entsprechende Testwerkzeuge, die durch geeignete Instrumentierung des Codes eine Messung der Überdeckung und eine komfortable Darstellung des Testergebnisses sicherstellen. Manuell ist ein Statement Test mit vertretbarem Aufwand nicht durchzuführen. Der Statement Test wird (wie alle anderen folgenden White-Box Testtechniken) vorwiegend beim Modultest oder allenfalls in frühen Integrationsphasen durchgeführt.

Zweig/Bedingungs Test

Der Zweig oder Bedingungs Test hat zum Ziel in einer Software die Äste aller Verzweigungen (z.B. if-Bedingungen, while-Schleifen) auszuführen. Verzweigungen steuern in der Software den Ablauf abhängig von und nach Auswertung einer logischen Bedingung. Die Testfalldefinition ist ähnlich wie beim Statement Test, d.h. die Inputs der zu testenden Komponente werden festgelegt, die zu testenden Verzweigungen d.h. die auszuführenden Programmzweige und der erwartete Output.

Als Testvollständigkeitskriterium ist meist eine 100% Abdeckung aller Zweige im Code definiert. Diese wird auch als C_1 Abdeckung bezeichnet. Die Zweigüberdeckung ist eine echte Obermenge der Statement Überdeckung, wie das folgende Codebeispiel zeigt:

```
...
while (a < 10)
{ ...
  a = a - x;
  ...
}
...
```

Eine Statementüberdeckung kann erreicht werden durch einen Eintritt in die while-Schleife mit einem Wert $a < 10$. Eine Zweigüberdeckung erfordert zusätzlich einen Testfall mit einem Wert von $a \geq 10$, um um die while-Schleife "herumzulaufen".

Pfad Test

Beim Pfadtest werden bestimmte Pfade durch die zu testende Software ausgeführt. Ein Pfad ist dabei eine Folge von nacheinander ausgeführten Statements vom Start der Software bis zu ihrer Beendigung. Analog zur Statement und Zweigüberdeckung wird ein Testfall spezifiziert durch seine Inputs, die auszuführenden Pfade und den erwarteten Output.

Als Testvollständigkeitskriterium wird eine Abdeckung möglichst vieler Pfade, im Idealfall 100% = C_2 Abdeckung angestrebt. Eine 100% Pfadabdeckung wird nicht immer erreichbar sein, da eine while-Schleife mit einer geeigneten Bedingung unter Umständen zu unendlich vielen Pfaden durch eine Software führen kann. Ansonsten ist eine Pfadüberdeckung eine echte Obermenge der Zweigüberdeckung.

Weitere Bedingungsüberdeckungen.

Um die Testtiefe zu erhöhen, kann die Zweig- oder Bedingungsüberdeckung erweitert werden durch Detaillierung der Bedingungsauswertung.

Einfache Bedingungsüberdeckung

Bei der einfachen Bedingungsüberdeckung muss jede atomare Bedingung beim Test mindestens einmal den Wert falsch und einmal den Wert richtig angenommen haben. Eine atomare Bedingung ist dabei eine Bedingung, die sich nicht weiter in Teilbedingungen zerlegen lässt.

Beispiel:

If ($a < 5$) && (($b == 7$) || ($c > 10$))

Atomare Bedingungen sind hier $a < 5$, $b == 7$, $c > 10$

Die Einfache Bedingungsüberdeckung ist eine echte Obermenge der Zweigüberdeckung und erfordert daher auch mehr Testfälle.

Mehrfachbedingungsüberdeckung

Bei der Mehrfachbedingungsüberdeckung werden alle möglichen Kombinationen der atomaren Bedingungen getestet. Dies führt zu einer großen Anzahl von Testfällen (2^n bei n atomaren Bedingungen einer Verzweigung), die nur mit großen Aufwand zu erstellen sind.

Minimale Mehrfachbedingungsüberdeckung

Bei der minimalen Mehrfachbedingungsüberdeckung werden die Testfälle so gewählt, dass jede atomare Bedingung einmal den Gesamtwahrheitswert der zugehörigen Verzweigung beeinflusst.

Dies reduziert die Anzahl der Testfälle gegenüber der Mehrfachbedingungsüberdeckung.

Anmerkungen zu den Testtechniken

Es gibt weder einen wissenschaftlichen Beweis noch einen Konsens in der Industrie für eine beste Testtechnik. „Beste“ im Sinne von Finden der meisten oder gar aller Fehler mit dem geringsten Aufwand. Alle erwähnten Techniken haben vor und Nachteile, manche sind nur für bestimmte Testphasen geeignet. Die meisten Techniken erfordern die Erstellung zusätzlicher Testsoftware, z.B. Testrahmen um die zu testende Software zum Ablauf zu bringen oder zusätzlicher Code²¹, der die Überdeckung beim Whiteboxtest misst.

Die Durchführung von Tests ist auch meist durch entsprechende Testtools zu unterstützen. White-Box Tests ohne entsprechende Tools, die Überdeckungen automatisch messen, können nur mit hohem, in der Praxis zu hohem Aufwand durchgeführt werden. Entsprechende Testtools sind oft teuer und komplex zu bedienen. Daher müssen Tools vor ihrem Einsatz evaluiert werden und die Tester sollten die zugehörigen Schulungen erhalten

Insgesamt müssen also Testtechniken geeignet ausgewählt und kombiniert werden und mit geeigneten Tools unterstützt werden.

Beim Modultest können Black-Box und White-Box Testtechniken auf folgende Weise kombiniert werden:

Zunächst werden Black-Box Tests definiert und zum Ablauf gebracht, bis das Black-Box Testvollständigkeitskriterium erreicht ist. Bei der Ausführung des Black-Box Tests wird die geplante Überdeckung (z.B. Zweigüberdeckung) gemessen. Ist bereits beim Black-Box Test das White-Box Vollständigkeitskriterium erreicht, so ist der Test beendet. Falls dies nicht der Fall ist müssen weitere White-Box Testfälle durchgeführt werden.

Test Techniken – Weitere Begriffe

Regressionstest

Ein Regressionstest ist die Wiederholung eines bereits durchgeführten Tests nach einer Änderung (z.B. Fehlerbehebung, geplante Erweiterung). Der Regressionstest soll sicherstellen, dass sich die getestete Software in Bereichen, die durch die Änderung nicht betroffen sein sollten, nicht unbeabsichtigt anders verhält als vor der Änderung. Die einfachste Art des Regressionstests ist eine komplette Wiederholung des bisherigen Tests in den entsprechenden Testphasen. D.h. geänderte Module werden einem Modultest unterzogen, dann mit den nicht geänderten Modulen integriert und zum Schluss wird die komplette Software validiert. In der Praxis scheitert dies oftmals am hohen Aufwand oder an der Kürze der Zeit, in der eine Fehlerbehebung durchzuführen und freizugeben ist. In diesem Fall kann durch eine so genannte Impact Analyse durchgeführt werden. D.h. es werden basierend auf dem Sourcecode, dem Detailed Design und der Software Architektur die Bereiche des Software ermittelt, die durch die Änderungen betroffen sind. Danach werden nur, die Testfälle, die diese Bereiche betreffen ausgeführt. Weitere Möglichkeiten Testfälle für Regressionstests zu reduzieren bietet ein Vorgehen gemäß Testfallprioritäten oder ein risikobasierter Test.

Akzeptanztest (Abnahmetest)

²¹ dieser zusätzliche Code wird meist durch entsprechende Testwerkzeuge erzeugt und in den zu testenden Code eingefügt.

Der Akzeptanz oder Abnahmetest ist ein formal Test der es ermöglicht, dass ein Benutzer, ein Kunde oder eine dazu autorisierte Instanz eine SW abnimmt d.h. zur Verwendung freigibt [s. auch IEEE Std 610.12-1990].

Im Gegensatz zu anderen Tests ist es beim Akzeptanztest nicht das Ziel, Fehler zu finden, sondern denjenigen der die Software prüft, von der Qualität zu überzeugen.

Dieser Test wird normalerweise am Ende der Entwicklung durchgeführt, ggf. zusammen mit dem Kunden basierend auf einer eigenen Testspezifikation.

Test Dokumentation

Jeder Test, der durchgeführt wird, muss geeignet dokumentiert werden. Bei der Erstellung der Testdokumentation wird die Vorgehensweise beim Test implizit hinterfragt.

Testdokumentation ist notwendig, um mit anderen über den Test kommunizieren zu können und dient letztlich auch dem Festhalten von Arbeitsergebnissen. Folgender Nutzen kann aus der Testdokumentation gezogen werden:

- 1.) Sinnvoller Einsatz von Teststrategien und –Techniken, Verbesserung der Testplanung und Ausführung.
- 2.) Nachweis der Testdurchführung und des Testergebnisses gegenüber dem Kunden, Zertifizierungs- oder Abnahmeinstitutionen und im Extremfall (Produkthaftung) auch gegenüber dem Gesetz (Staatsanwalt, Richter, Sachverständiger).
- 3.) Nur dokumentierte Testfälle können geeignet wieder verwendet werden. Jeder Testfall stellt eine Investition dar, die ohne Dokumentation verloren ist. Wiederverwendung kann im eigenen Projekt stattfinden, falls Software in verschiedenen Versionen oder Varianten entwickelt wird, oder auch in parallel laufenden oder nachfolgenden ähnlichen Projekten.
- 4.) Dokumentierte Tests können mit Hilfe geeigneter Testmetriken ausgewertet werden. Metriken können hierbei sein, die Anzahl, Schwere oder Ursache der gefunden Fehler, Anzahl der ausgeführten Testfälle oder Aufwand für Testphasen und für den Test insgesamt. Einfache Metriken können dann kombiniert werden (Anzahl der Testfälle in Relation zu den gefundenen Fehlern in einer Testphase) und über mehrere Tests ggf. über mehrere Projekte zu einer Verbesserung des Testvorgehens insgesamt genutzt werden.

Teststrategie

Die Teststrategie dokumentiert die grundsätzliche Vorgehensweise beim Test bezüglich

- der Verantwortlichkeiten, d.h. wer aus dem Entwicklungsteam ist für welche Tätigkeit beim Test verantwortlich,
- der beim Test angewendeten Teststrategien und –Techniken inklusive der zu erreichenden Testvollständigkeitskriterien,
- der beim Test verwendeten Tools,
- und ggf. des beim Test durchgeführten Tailorings, d.h. Anpassen des Standardtestprozesses an die Projektgegebenheiten.

Die Teststrategie ist ein Planungsdokument, das im Rahmen der Projektplanung erstellt und ggf. dem Projektverlauf angepasst wird.

Testspezifikationen

Testspezifikationen sind testphasenspezifische Dokumentation der Testfälle. In einer Testspezifikation lässt sich jeder Testfall eindeutig identifizieren, z.B. über einen Namen oder eine Nummer. Zusätzlich sollte die Testspezifikation einen Bezug zu den Anforderungen herstellen auf die sich der Testfall bezieht. Bei Black-Box Tests sind die Anforderungen in Spezifikationen und Designdokumenten festgelegt. Über die Testphasen wird zwischen

- Modultestspezifikation(en)
- Integrationstestspezifikation(en)
- Validierungstestspezifikation(en)

unterschieden.

Testspezifikationen können (ganz oder teilweise) durch geeignete Testskripts für Testtools ersetzt werden. Voraussetzung ist, dass die Testskripts die Anforderungen an Testspezifikationen inhaltlich abdecken.

Test Log

Das Test Log enthält das detaillierte Ergebnis aller Testfälle einer Testdurchführung. Im Test Log lässt sich das Ergebnis jedes einzelnen Testfalls wieder finden. Folgende Informationen sollte das Test Log für jeden Testfall enthalten:

- Testfallname/-nummer (Zuordnung zur Testspezifikation)
- Ergebnis des Testfalls (konkretes Ergebnis für den Testfall, zusätzlich gut/schlecht)
- ggf. Testabdeckung des Testfalls bzgl. des geplanten Endekriteriums
- ggf. Dauer der Ausführung des Testfalls
- ggf. Ressourcenverbrauch

Das Test Log dient auch als Nachweise der Testdurchführung.

Test Report

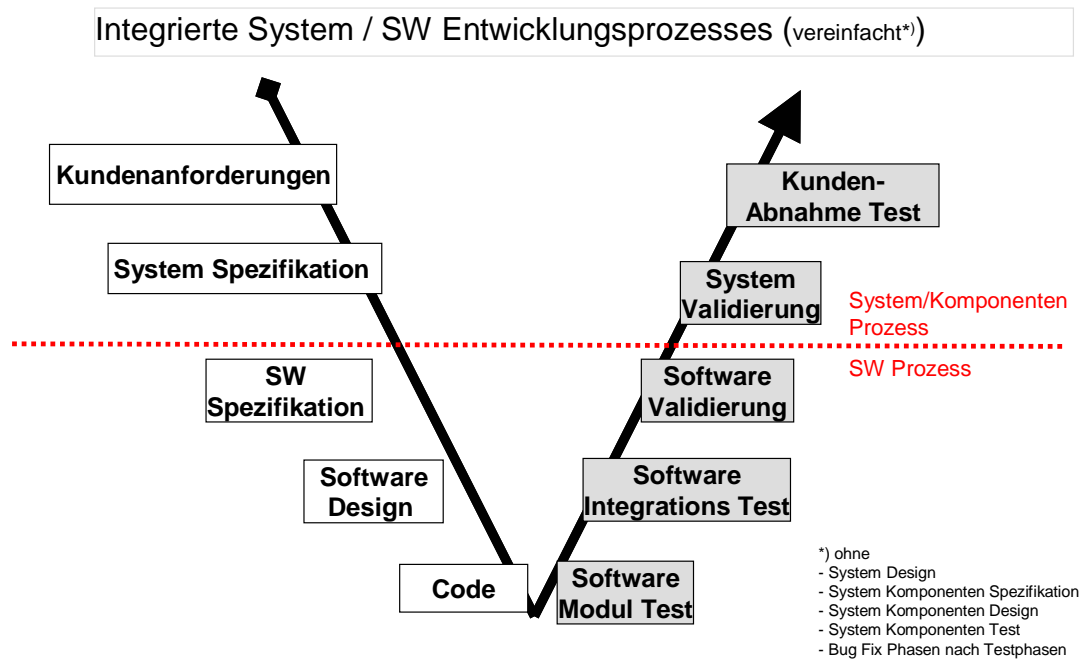
Der Test Report fasst die Testlogs einer Testdurchführung einer Testphase zusammen. Die wesentliche Information wird im Testreport dargestellt:

- Anzahl ausgeführter Testfälle
- Anzahl fehlerfrei durchgelaufener Testfälle
- Anzahl nicht durchgelaufener Testfälle (Fehler gefunden)
- ggf. Schwere der Fehler
- ggf. Abdeckung Testendekriterium
- ggf. Dauer / Ressourcenverbrauch

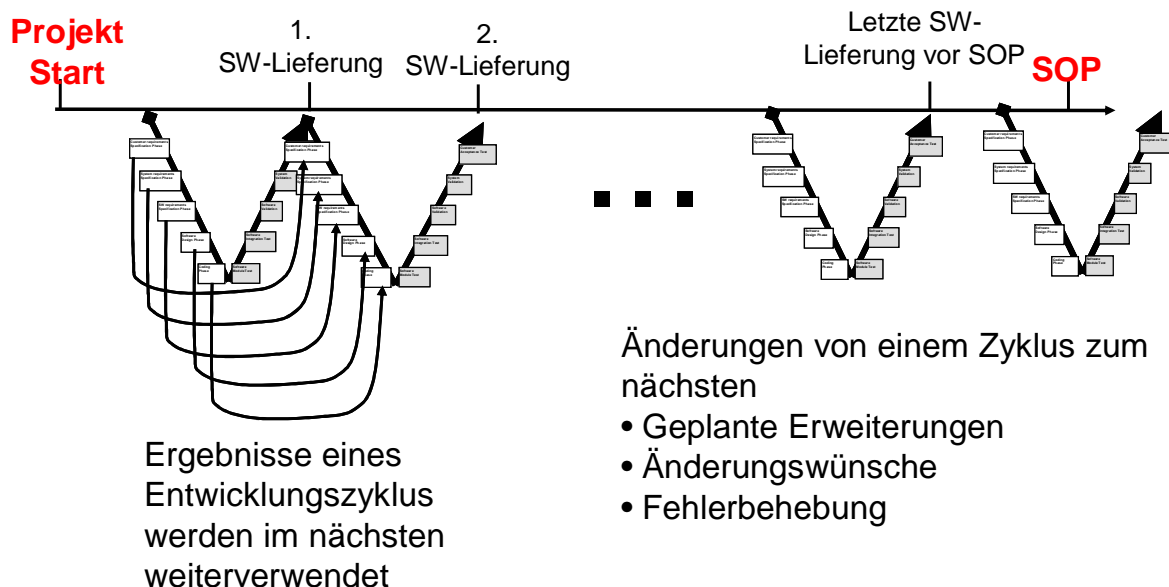
Der Testreport ermöglicht die Gesamtbewertung der Testdurchführung bzgl. Testende, Restfehler und dient auch dazu zukünftige Tests zu planen.

Exkurs: Automotive Projekte und Software

In der Automobiltechnik spielt die „reine“ Softwareentwicklung, d.h. im Rahmen eines Projekts wird nur Software entwickelt eine untergeordnete Rolle. Das entwickelte Produkt ist meist ein eingebettetes System aus Hardware, Software und Mechanik. Für die Software bedeutet das, dass das Software-Projekt eingebettet ist in ein Komponenten- oder System-Projekt und die Software-Entwicklung Schnittstellen zu Komponenten bzw. System-Spezifikationen besitzt.



Die Softwareentwicklung ist also integraler Bestandteil eines Systementwicklungsprozesses (oben dargestellt als V-Zyklus). Erschwerend kommt hinzu, dass das Produkt meist nicht in einem Schritt und oft basierend auf nicht endgültigen Requirements entwickelt wird, sondern in enger Abstimmung zwischen Automobilhersteller und Elektronikzulieferer schrittweise in aufeinander aufbauenden Musterständen entwickelt wird (unten dargestellt zwischen Projektstart und Start der Produktion (SOP)). Zusätzliche Probleme ergeben sich durch die zunehmende Integration von „Fremdsoftware“, d.h. Software, die nicht von Hersteller des Steuergeräts, sondern vom Automobilhersteller beigestellt wird. Des weiteren werden die Entwicklungszeiten für ein Produkt zunehmend kürzer, parallel zu den kürzeren Lebenszyklen neuerer Automobile.

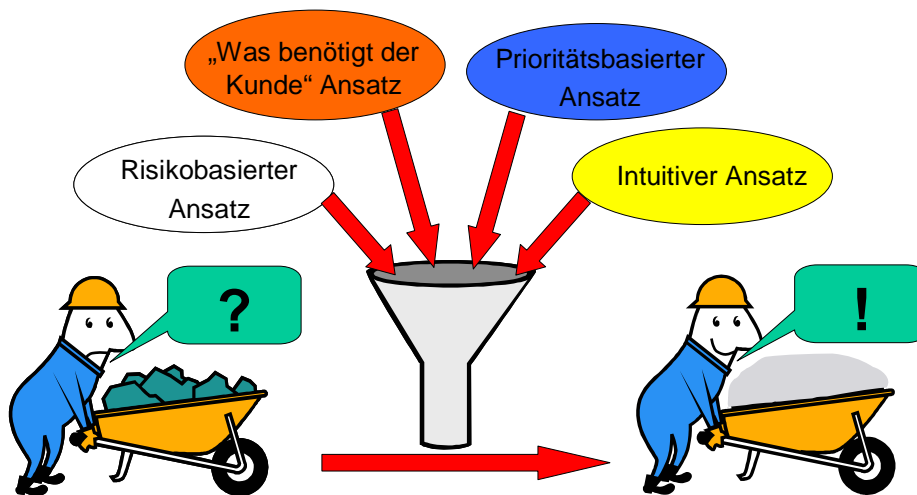


Aus diesen hier noch zum Teil vereinfacht dargestellten Sachverhalten ergeben sich folgende Konsequenzen für das Testvorgehen::

- nicht jede Softwarelieferung kann und/oder muss komplett getestet werden (s. Kapitel Reduktion des Testumfangs, unten)
- nur Integrationstest für Fremdsoftware, basierend auf (externer) Testspezifikation.

- frühe Tests auf dem PC, auf Emulator, "Hilfs-Hardware" (z.B. aus Vorgängerprojekt)
- so viele Tests wie möglich automatisieren
- nicht alle Tests lassen sich automatisieren
- Test der SW-Laufzeiten ist wichtig, inkl. Taskverhalten SW Test nicht im Detail über die ganze Laufzeit planen
- Detailplanung des Tests nur für nächste Lieferung(en), nur eine grobe Planung über Gesamtprojekt
- Gesetzliche Anforderungen bei sicherheitskritischen Systemen, beachten von entsprechenden Normen (z.B. IEC 61508)
- Labortests sind wichtig (SW-, Komponenten- und System-Tests, diese Tests gehen normalerweise über die SW-Entwicklung hinaus)
- Fahrzeugtests sind wichtig (kein SW-Test, sondern Systemtest), da sich die Fahrzeugumgebung nicht zu 100% simulieren lässt.
- HW/SW Integrationstest ist wichtig

Reduktion des Testumfangs



In der Praxis können aus vielerlei Gründen Software Tests nicht in vollem Umfang durchgeführt werden. Dies kann z.B. an kurzen Entwicklungszyklen (wie sie im Automotive Umfeld häufig vorkommen) liegen, oder durch schnelle Fehlerbehebungen. Darüber hinaus ist ein vollständiger Test auch gar nicht immer sinnvoll, z.B. falls noch Unklarheit bezüglich bestimmter Anforderungen besteht oder eine hohe Wahrscheinlichkeit für Änderungen von Anforderungen besteht. Somit ergibt sich die Notwendigkeit Software Tests geeignet zu reduzieren. Diese Anpassung kann basierend auf bereits vorhandenen Tests z.B. für einen Regressionstest oder auch für noch zu erstellende Tests durchgeführt werden. Die dafür zur Verfügung stehenden Verfahren werden in den folgenden Kapiteln kurz erläutert.

Intuitiver Ansatz

Hierbei wird der Umfang des Tests intuitiv bestimmt mit dem Risiko, eine falsche Auswahl zu treffen.

"Was benötigt der Kunde" Ansatz

Ziel ist es hier, nur das zu testen, was der Kunde (z.B. im Fall eines Zulieferers ist das der Automobilhersteller) für die auszuliefernde Software benötigt. Ist dies die Korrektur eines Fehlers sein, wird nur ein entsprechend definierter Regressionstest ausgeführt. Handelt es sich um eine frühe Lieferung, die im Wesentlichen eine Funktionalität darstellen soll, genügen vielleicht ausschließlich positive Tests der gewünschten Funktionalität. Wenn das Steuergerät auf seine Verträglichkeit in einem Bus-Netzwerk, z.B. CAN getestet werden soll, wird das Augenmerk beim Test auf die entsprechende Kommunikationssoftware, insbeson-

dere das Netzwerkmanagement gelegt.

Schwierig ist es hierbei allerdings, genau zu definieren, was der Kunde wirklich will. Eine Nachfrage führt nicht immer zur gewünschten Antwort, da die Erwartungen der Kunden höher als die eigene Einschätzung liegen können.

Prioritätsbasierter Ansatz

Beim Prioritätsbasierten Ansatz werden die Testfälle in Klassen bezüglich ihrer Bedeutung eingeteilt. Eine Einteilung könnte wie folgt aussehen:

- Priorität 1: Testfälle, die sicherheitskritische Funktionen des Produkts testen (z.B. Fensterheber).
- Priorität 2: Testfälle, die Funktionen testen, die für die Gesamtfunktion des Produkts wichtig sind (z.B. Senden und Empfangen von Funksignalen bei einem Fernbedienungsschlüssel).
- Priorität 3: Testfälle, die Einzelfunktionen testen, die aber nicht für die Gesamtfunktionalität wichtig sind (z.B. Einstellbarkeit der Instrumentenbeleuchtung).
- Priorität 4: Testfälle, die Funktionen testen, die dem Endkunden nicht sichtbar sind (z.B. Diagnosefunktionen)

Diese Prioritätseinteilung soll hier nur als Beispiel dienen. Es kann durchaus sein, dass aus Sicht des Kunden Funktionen, die dem Endkunden nicht sichtbar sind, wie Diagnose, eine sehr hohe Bedeutung haben. Die Prioritätenklassifizierung muss daher projektspezifisch geprüft werden.

Risikobasierter Ansatz

Alle zu testenden Funktionen²² in der SW werden bezüglich verschiedener Risikofaktoren bewertet. Diese können zum Beispiel sein:

- Die Erfahrung der zuständigen Entwickler mit dieser Funktion. Ist es eine Funktion, die erstmals implementiert wird oder wurde diese Funktion ähnlich schon für andere Projekte entwickelt.
- Die Komplexität der Funktion aus Sicht der Entwickler. Komplexität kann formal gemessen werden, z.B. über eine Codekomplexitätsmetrik einer bereits bestehenden ähnlichen Software, oder auch intuitiv geschätzt werden.
- Die Bedeutung der Funktion im Gesamtsystem, z.B. ist diese Funktion sicherheitsrelevant, ist sie entscheidend für die Gesamtfunktionalität oder ist sie nur eine weniger wichtige Hilfsfunktion.

Jedem Risikofaktor wird eine Bewertungsschema z.B. niedrig = 1, mittel = 2, hoch = 3 zugeordnet. Nachdem für die zu testende Funktion die Risikobewertung der einzelnen Faktoren durchgeführt ist, kann ein Gesamtrisiko aus dem Durchschnitt der Einzelrisiken $(\text{Wert Erfahrung} + \text{Wert Komplexität} + \text{Wert Bedeutung}) / 3$ zugeordnet werden²³. Dieser Gesamtwert, der ebenfalls ein α bestimmt dann den Testumfang. Z.B. Funktionen mit niedrigem Risiko werden beim Modultest nur intuitiv getestet und erst in der Validierung einem kompletten Test unterzogen.

Exkurs: Testen objektorientierter Software

Im Gegensatz zur funktionalen Programmierung (z.B. mittels C) besitzt die Objektorientierte Programmierung zusätzliche Elemente, die beim Testen berücksichtigt werden müssen. Dieses Kapitel gibt dazu einen ganz kurzen Überblick ohne auf Details einzugehen. Diese sind in der entsprechenden Literatur nachzulesen. Zu den Eigenheiten der Objektorientierung zählen unter anderem Klassen, Klassenhierarchien (Vererbung), virtueller Klassen (Klassen, bei denen nicht alle Methoden implementiert sind), Templateklassen (parametri-

²² "Funktion" kann hier auf jedem Niveau verstanden werden z.B. eine Funktion im Gesamtsystem oder auch eine C-Funktion im Sourcecode.

²³ Gegebenenfalls kann noch eine Gewichtung der Risikofaktoren durchgeführt werden, z.B. Erfahrung zählt doppelt, da ein erfahrenes Team auch komplexe und bedeutende Funktionen mit Erfolg implementieren kann.

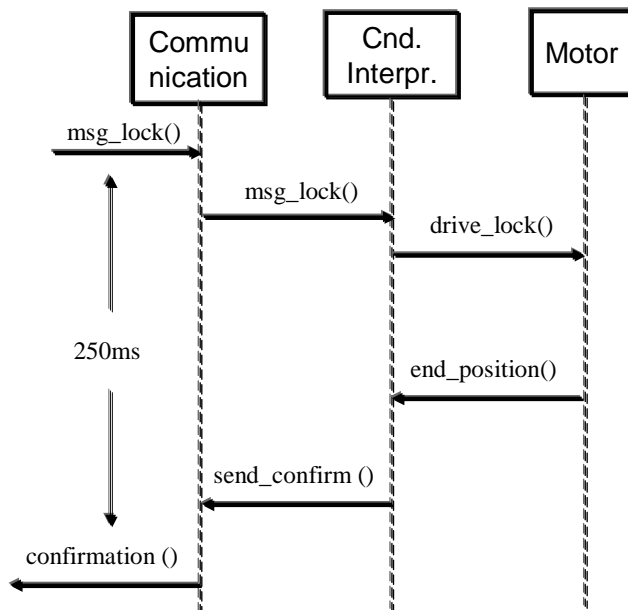
sierte Klassen) und Testen von Klassen- bzw. Objektinteraktionen.

Zusätzlich zu diesen Elementen haben gängige Objektorientierte Programmiersprachen (C++, C#, Java) ihre Eigenheiten.

Beim Test Objektorientierter Software Prinzipiell lassen sich die bekannten, „herkömmlichen“ Teststrategien und Testtechniken, wie sie oben vorgestellt wurden, auch auf Objektorientierte Software z.T. erweitert anwenden. Weitere Testtechniken sind aber für Objektorientierte Software sinnvoll bzw. notwendig:

- Beim Test von Klassen
 - Test einzelner Methoden (Member Functions)
 - Funktionaler Anteil (wie bei funktionaler Software)
 - Test der Wirkung einer Methode auf die (gekapselten) Attribute des Aufrufobjekts
 - Berücksichtigung von Konstruktoren und Destruktoren als besondere Memberfunktionen
 - Test des Zusammenspiels der Methoden einer Klasse
 - Integration der Methoden in die Klasse, d.h. die Methode muss mit den anderen Methoden wie spezifiziert zusammenarbeiten
 - Aufrufreihenfolgen testen (Konstruktor, Methode1, Methode2, ... Destruktor)
- Beim Test von Klassenhierarchien (Testen über Vererbungsbeziehungen)
 - Test des Zusammenspiels der Methoden der Unterklasse mit den Methoden der Oberklassen
 - Integration der Methoden in die Klassenhierarchie
 - Test der Wirkung einer Methode auf die Oberklassenattribute
 - Aufrufreihenfolgen testen (Konstruktor, Methode1, Methode2, ... Destruktor)
- Beim Testen virtueller Klassen (d.h. Klassen, die nicht instanzierbar sind).
 - Black-Box Testfälle definieren auf Basis der Spezifikation der virtuellen Methoden
 - Testfälle durchführen, wenn Methoden implementiert werden in einer Unterklasse
 - Test erweitern (z.B. um weitere Black Box und White Box Tests), wenn virtuelle Methoden in einer Unterklasse ausgeprägt (d.h. implementiert) werden.
- Beim Test von Template Klassen
 - Test von exemplarischen Ausprägungen der Templateklasse (Instantiierung mit einer entsprechenden Parameterklasse)
 - Wiederverwendung dieser Testfälle bei neuen Instanziierungen
 - Ergänzung um weitere Tests der Ausprägung, falls Abhängigkeiten durch die ausprägende Klasse entstehen können (z.B. durch Aufrufe von Konstruktoren)
- Testen der Objekt- bzw. Klassenkommunikation
 - Sequence Charts oder Kommunikationsdiagramme als Basis (Definition der Kommunikation zwischen Objekten bzw. Klassen)
 - Test, ob die Objekte zur Laufzeit die spezifizierte Kommunikation durchführen, ggf. mit spezifiziertem Timing
 - Test ggf. ergänzen durch negative Tests um Robustheit zu testen (Objekt verhält sich nicht so, wie spezifiziert, wie reagiert die Umwelt)

Im Beispiel unten muss getestet werden, ob die Kommunikation der Objekte, wie spezifiziert ausgeführt wird, um einen Schließvorgang z.B. einer Fahrzeughürverriegelung ausgeführt wird.



Exkurs: Testdatengenerierung aus UML Modellen

Wie schon oben erwähnt, können UML Diagramme als Ergebnis von Analyse und Designphase dazu genutzt werden, Testdaten für Tests auf verschiedenen Ebenen zu generieren. Mit Testdaten sind hier komplette Testfälle und auch Testfälle gemeint, die noch manuell mit Informationen angereichert werden müssen, um ablauffähig zu sein.

UML Diagramme werden meist verwendet, um Systeme bzw. Software objektorientiert zu modellieren, d.h. es gibt parallelen um Test Objektorientierter Software (s. oben), und UML Diagramme werden auch zur modellbasierten Entwicklung genutzt. Hier gibt es Parallelen zum modellbasierten Test (s. unten)

Bestimmte UML Diagramme (z.B. Zustandsdiagramme/State Diagrams) werden ähnlich auch in anderen Modellierungssprachen bzw. Methoden verwendet. Das bedeutet, dass Methoden für UML (ggf. angepasst) auch dort verwendet werden können.

Testdatengenerierung aus Zustandsdiagrammen / State Diagrams

Zustandsdiagramme beschreiben (vollständig) das dynamische Verhalten eines Modellelements mittels Zuständen, Zustandsübergängen und Ereignissen, die einen Übergang auslösen. Das Ziel eines Tests ist es festzustellen, ob der implementierte Zustandsautomat der Spezifikation entspricht. Testfälle der Form:

Ausgangszustand -> Ereignis -> Folgezustand
können direkt aus dem Zustandsdiagramm generiert werden.

Testdatengenerierung aus Sequenzdiagrammen / Sequence Charts

s. Testen Objektorientierter Software

Testdatengenerierung aus Kommunikationsdiagrammen / Communication Diagrams

Diese beschreiben die Kommunikation zwischen Objekten bzw. Klassen. Der Informationsgehalt ist ähnlich oder identisch zu Sequenzdiagrammen. Das Ziel ist hier zu testen, ob die Kommunikation in der Implementierung der Spezifikation entspricht. Die Testfälle können wie bei Sequenzdiagrammen abgeleitet bzw. generiert werden.

Testdatengenerierung aus Aktivitätsdiagrammen / Activity Charts

Diese Diagramme beschreiben interne Abläufe der Software. Ziel der Tests ist es, ob die Implementierung der Spezifikation aus dem Diagramm entspricht (eine Art White-Box-Test)

Exkurs: Modellbasiertes Tests

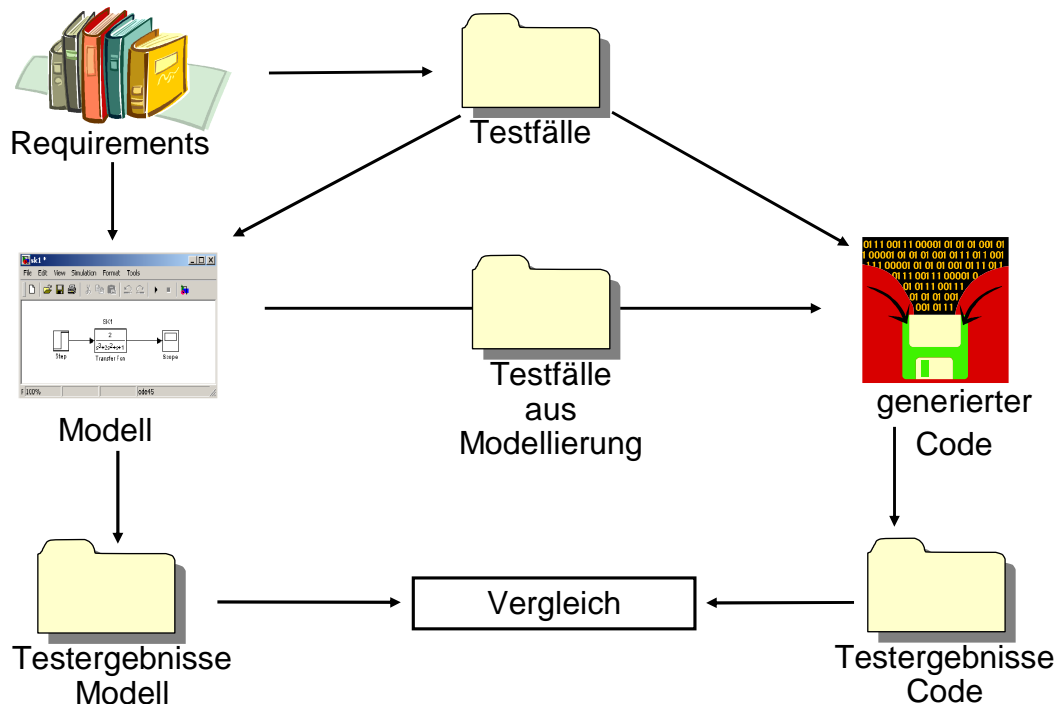
Als Basis dienen hier Tools wie Statemate/Stateflow (Zustandsbasierte Systeme), Matlab/Simulink (Regelungs-/Messtechnik), oder UML Tools

Ähnlich wie bei Code könne hier statische Tests des Modells durchgeführt werden, z.B. gegen Modellierungsrichtlinien, auf Vollständigkeit und Konsistenz. Dies kann durch Reviews geschehen oder auch durch entsprechende Toolfeatures.

Neben den statischen Tests können dynamische Testen auf Modellebene (Simulation) ablaufen. Dabei wird das Modells gegen die Requirements (Black-Box Test) geprüft (Test, ob das Modell die Requirements erfüllt), die Modellabdeckung (White-Box Test) kann gemessen werden (analog zur Codeüberdeckung, alle Modellelemente ausführen).

Dynamische Tests können Voraussetzung für Modellverfeinerungen (z.B. Fließkomma zu Festkommaarithmetik sein oder auch Voraussetzung für eine Codegenerierung oder manuelle Codierung.

Ähnlich wie bei UML Modellen könne Test-/Simulationsdaten aus Modellen generiert werden, um Tests auf Modellebene zu erstellen, um generierten oder manuell erzeugten Code zu testen. Der Code soll sich (soweit möglich) wie das Modell verhalten, unter Berücksichtigung von Verfeinerungen des Modells, verschiedenen Umgebungen (z.B. PC, Ziel-Hardware) und ggf. verschiedenen Codegeneratoren. Die Zusammenhänge sind im Bild unten dargestellt.



Exkurs: Testen grafischer Bedienoberflächen

Der „klassische“ Testansatz (Test gegen Spezifikation) ist bei grafischen Bedienoberflächen nicht 1-zu-1 übertragbar. Die verbale Spezifikation der Grafik ist schwierig, daher auch die Übertragung der Spezifikation auf Testfälle. Die Qualität einer Bedienoberfläche ist objektiv schwierig zu bewerten, subjektive Gesichtspunkte und persönlicher Geschmack der Benutzer spielen eine gewisse Rolle. Abschließend ist der Test grafischer Oberflächen nur möglich mit geeigneten Tools (z.B. WinRunner).

Das Grundprinzip „Capture Playback“ wird angewendet:

- Erstellen eines Sollergebnisses durch Aufzeichnen und Bewerten der zu testenden grafischen Abläufe durch geeignete Werkzeuge
- Aufzeichnung ggf. manuell nachbessern
- Aufzeichnung als Sollergebnis verwenden
- Aufzeichnen der Abläufe der zu testenden grafischen Bedienoberfläche und Vergleich mit dem Sollergebnis (werkzeugunterstützt)

- Testtool führt Interaktionen (z.B. Mausklicks) wie bei Aufzeichnung durch

Do's und Dont's, Erfahrungen, Probleme, Tipps und Tricks

Dieses Kapitel soll einige Erfahrungen (gute und schlechte) aus der Praxis präsentieren. Einige Punkte wurden bereits in den vorherigen Kapiteln erwähnt.

☺ **Module Test lohnt sich, auch im Embedded Umfeld.**

Das Thema Modultest bzgl. Kosten/Nutzen Verhältnis immer wieder diskutiert. Die Erfahrung zeigt, dass Modultests richtig angewendet, d.h. zum Test von Algorithmen, Fehler finden können. Module, die keine Algorithmen enthalten evtl. eher einem Code Review unterziehen.

☺ **Auch einfache Tests sollten dokumentiert werden.**

Dies ist eine grundsätzliche Regel. Tests sollten aus den oben genannten Gründen immer dokumentiert werden. Eine Testdokumentation kann auch ein handschriftliches, formloses Dokument sein, das die relevanten Informationen über den Test festhält.

☺ **Software Tests ergänzen Reviews und umgekehrt**

Software Tests und Reviews sollten immer beide im Projekt angewendet werden. Beide haben Stärken, die sich gegenseitig ergänzen und Schwächen, die sich gegenseitig aufheben.

☺ **Tester möglichst unabhängig vom Rest der Entwicklung (ggf. für alle Testphasen)**

Wenn möglich sollten die Software Tester nicht die Personen sein, die Software entworfen und codiert haben. Unabhängigen Testern fällt es leichter, eine "destruktive"²⁴ Haltung gegenüber der zu testenden Software einzunehmen.

☺ **Software Modul Test ohne dokumentiertes Software Feindesign**

Ohne dokumentiertes Feindesign der zu testenden Module gibt es keine sinnvolle Basis einen Modultest aufzusetzen. White-Box Tests allein, die nur auf eine Codeüberdeckung abzielen, sind weitgehend sinnlos.

☺ **Software Integrationstest ohne dokumentierte Software Architektur**

Ohne dokumentierte Architektur der zu testenden Software gibt es keine sinnvolle Basis einen Integrationstest aufzusetzen.

☺ **Software Validierung ohne Software Requirements**

Ohne dokumentierte Anforderungen an die zu testenden Software gibt es keine sinnvolle Basis eine Validierung aufzusetzen.

☺ **Erstellen der Testspezifikation wenn der Test beginnen soll**

Da ist es meistens zu spät! Testspezifikationen für die Validierung werden parallel zu den Requirements, für den Integrationstest parallel zur Software Architektur und für den Modultest parallel zum Software Feindesign erstellt.

☺ **Unterschätzen des Testaufwands ($\geq 50\%$ des Gesamtaufwands)**

50% des Gesamtaufwandes sollte als Daumenregel für den Minimalaufwand angesehen werden. Im Automotive Umfeld liegt der tatsächliche Aufwand manchmal auch deutlich höher. D.h. genaue Aufwandsabschätzungen sind erforderlich.

²⁴ "destruktiv" Haltung bedeutet hier: Die Software enthält Fehler, ich muss sie finden!

- ☺ **Geplante Testzeit als Zeitpuffer für andere Tätigkeiten missbraucht**
Passiert leider. In diese Fall entweder die Auslieferung verschieben. Falls nicht möglich zumindest die entstandenen Testlücken bewerten und darauf basierend eine Risikoanalyse durchführen. Ggf. den Testumfang gezielt und geplant reduzieren.
- ☺ **Überschätzen der Wirksamkeit von Test Tools**
Tools sind notwendig und nützlich, Quantensprünge bei der Reduktion von Aufwenden sind aber nicht erreichbar. Deshalb sollten Tools sorgfältig evaluiert werden und die Anwender entsprechend geschult werden.
- ☺ **Fehlende Traceability zu Requirements**
Traceability (Zuordnung und Nachverfolgbarkeit) von Testfällen zu Requirements ist eine wesentliche Anforderung aus Prozeßmodellen, die im Automotive Bereich verwendet werden (CMMI, Automotive SPICE), und aus Normen für sicherheitskritische Systeme (z.B. IEC 61508).
- ☺ **Nicht dokumentierte Tests**
Wie schon mehrfach erwähnt, haben nicht dokumentierte Tests formal nicht stattgefunden. Darüber hinaus sind die Wiederholbarkeit, der Regressionstest und die Wiederverwendung nicht möglich.
- ☺ **Unerfahrene Mitarbeiter zum Testen „versetzt“ (Testen ist Kunst und Wissenschaft!)**
Software Testen erfordert besondere Kenntnisse bezüglich Testmethodik und Testtools und viel Erfahrung und gute Intuition.
- ☺ **Testfälle von unklaren Anforderungen ableiten anstatt die Anforderungen zu klären**
Testfälle, die aus unklare Anforderungen abgeleitet werden, bergen ein hohes Risiko später geändert werden zu müssen. Auf Grund des hohen Aufwands zur Erstellung von Testfällen sollten daher unklare Requirements zuerst geklärt werden.
- ☺ **Keine saubere Wartung von Testspezifikationen**
Testspezifikationen sind "lebende" Dokumente. Gerade im bei der Entwicklung von Automotive Software sind Änderungen der Requirements häufig. Die zugehörigen Testspezifikationen müssen entsprechend angepasst werden.
- ☺ **"Ich muss auf der Zielhardware testen" (Tests auf PC-Umgebungen sind möglich)**
Softwaretests auf der Zielhardware sind natürlich notwendig. Allerdings sind diese Tests oft aufwendig durchzuführen, werden behindert durch späte Verfügbarkeit oder Qualitätsprobleme der Hardware. Deshalb sollte die Möglichkeit in PC-Umgebungen mit PC-Compilern übersetzbare oder auf Hardware-Simulatoren ablauffähige Software des Produkts zu testen, genutzt werden. Schwerpunkt hierbei ist der Test von Algorithmen, die sich auf einem PC genauso oder sehr ähnlich verhalten sollten wie auf der Zielhardware.
- ☺ **Falscher Fokus/Ziel des Tests („meine SOFTWARE ist schon korrekt, "destruktiver" Modus nicht angewendet)**
Primäres Ziel des Tests und damit auch des Testers ist es, Fehler zu finden, d.h. zu Beweisen, dass die zu testende Software nicht funktioniert.

☺ **Big Bang Integration**

Wenn möglich sollte Software schrittweise systematisch integriert und integrationsgetestet werden.

☺ **SW Qualität kann nicht durch Test allein sichergestellt werden (Qualität in das Produkt hineintesten ist unmöglich!)**

Der Software Test und die Behebung der beim Test gefundenen Fehler ist ein Baustein der Softwareentwicklung um Qualität sicherzustellen. Ist das Grundkonzept der zu testenden Software ungenügend, wird auch ein noch so ausgeklügelter Test daran nichts ändern. Daher müssen in allen Entwicklungsphasen entsprechende Qualitätssicherungsmaßnahmen durchgeführt werden.

5. Literatur

- [1] Ian Sommerville, Software Engineering, Pearson Studium, ISBN: 0321210260
- [2] Tom de Marco, Der Termin, Hanser Fachbuchverlag, Leipzig 1998, ISBN 3446194320
- [3] Helmut Balzert, Lehrbuch der Software-Technik, Band 1. 2. Auflage. Elsevier-Verlag, 2001
- [4] Edward Yourdon, Death March, Prentice Hall International (ISBN 013143635X)
- [5] Bernd Kahlbrandt, Software-Engineering mit der UML, Springer-Verlag, 2001
- [6] Chris Rupp, Stefan Queins, Barbara Zengler, UML 2 – Glasklar, Hanser, 2007 (3. Auflage)
- [7] Andreas Spillner, Tilo Linz Basiswissen Softwaretest
<http://www.dpunkt.de/certified-tester/> German Book
- [8] Beizer, Boris Black-Box Testing: Techniques for Functional Testing of Software and Systems
- [9] Peter Liggesmeyer Software-Qualität: Testen, Analysieren und Verifizieren von Software
- [10] Harry M. Sneed, Mario Winter, Testen objektorientierter Software