Software Engineering

Klaus Knopper

Stand: 28. März 2018

Organisatorisches

Vorlesungs-Inhalt, Zeitplan, Klausur.

http://knopper.net/bw/se/

Das Problem (1)

Mit immer leistungsfähigerer Hardware und immer komplexeren Anforderungen sind folgende Fakten in aktueller Software zu beobachten:

- Kostenexplosion bei der Entwicklung und dem Testen von Software
- mangelnde Termineinhaltung bei der Softwareentwicklung
- unzufriedene Anwender
- schlechte Wartung der Software
- Anforderungen werden nicht eingehalten
- viele Programmfehler

Das Problem (2)

☞ Die Softwarekrise

Definition „Software Engineering“

„zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.“
(Balzert, Lehrbuch der Software-Technik. Bd.1., S.36)

Software Engineering

...ist die Lösung aller Probleme?

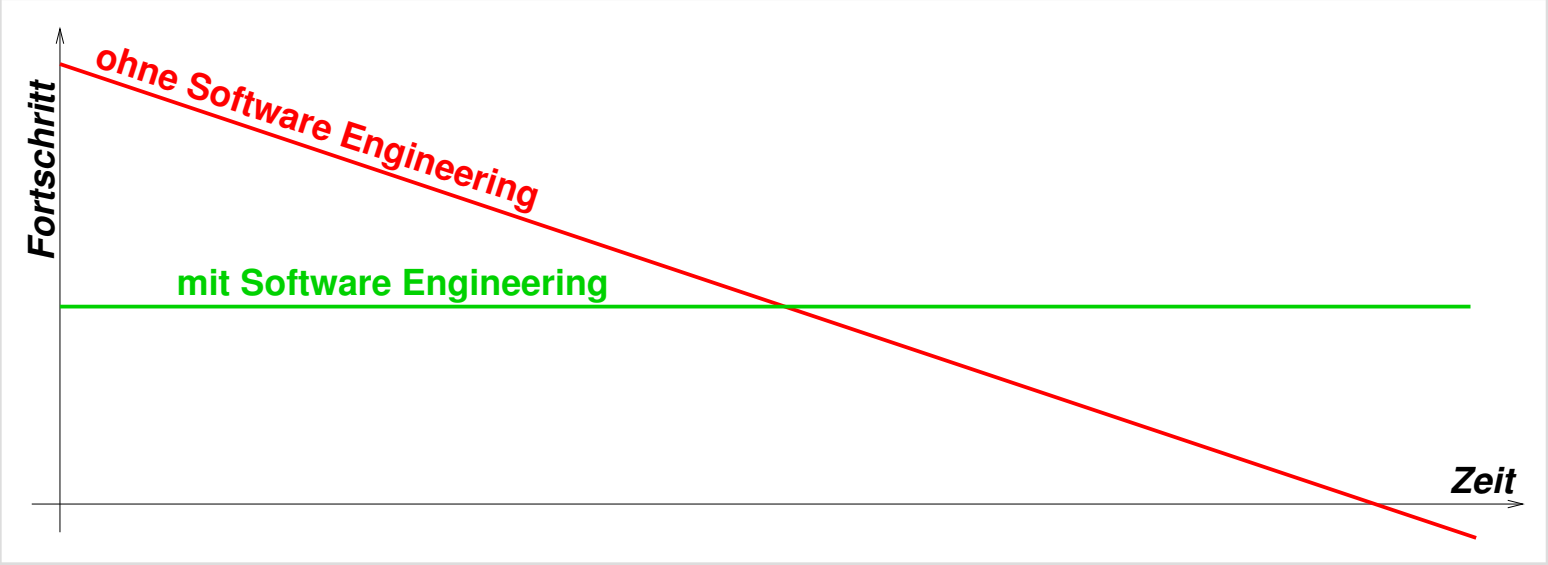
Sicher nicht, aber die Methoden des Software Engineering können helfen, komplexe Softwareanforderungen in den Griff zu bekommen, und bestimme Arten von Problemen bereits beim Entwurf zu vermeiden oder zu reduzieren.

Software Engineering vs. „einfach Programmieren“ (1)

- Kleine und wohldefinierte Aufgaben lassen sich auch durch „Drauflosprogrammieren“ lösen.
- Software Engineering ist zu Projektbeginn sehr aufwändig und „verzögert“ die Fertigstellung der Software.
- Software Engineering bedingt, dass zu Beginn die Anforderungen genau festgelegt werden.
- Bei umfangreichen Projekten hilft Software Engineering, später die Übersicht zu behalten und die Erweiterbarkeit zu sichern.

Software Engineering vs. „einfach Programmieren“ (2)

Bei der „Holzhacker-Methode“ sind die Anfangserfolge oft weit größer als beim Engineering, jedoch sinkt die Produktivität und steigt der Aufwand mit zunehmender Projektdauer und Komplexität.



Vorgriff: Aufwandsabschätzung

- Betriebskosten (laufende Kosten zur Aufrechterhaltung des Betriebes während der Entwicklung)
- Material (projektspez. Hardware, Software)
- Personal (projektspez. Arbeitsstunden)
- Sonstiges (Spesen: Reisekosten, Meetings, Schulung, Recherche...)

Die hierbei festgestellten Sachverhalte und getroffenen Festlegungen werden in einem Lasten- bzw. Pflichtenheft festgehalten.

Probleme bei der Aufwandsabschätzung

☞ Problem: Bis zum Angebots-Zuschlag (nach formal festgehaltener Aufwandsabschätzung) werden die bis dahin aufgelaufenen Kosten (Beratung, Arbeitszeit für die Kalkulation) i.d.R. nicht rückfinanziert.

☞ Problem: Unterlaufen bei der Angebotserstellung Fehler bei der Aufwandsabschätzung, so kann das Projekt schnell zum „Verlustgeschäft“ werden, da ein angenommenes Angebot als rechtskräftiger Vertrag einen gewissen finanziellen Rahmen festlegt, von dem nur geringfügig, oder in gegenseitigem Einverständnis abgewichen werden kann.

☞ Genaue Spezifikation als Grundlage der Aufwandsabschätzung, und Klarstellung der rechtlichen Rahmenbedingungen (Zuständigkeiten, Liefertermine, Leistungsumfang, Lizenzen) ist ein Muss!

Softwaretechnik

- umfasst eine Vielzahl von Teilgebieten, die in ihrer Gesamtheit die Softwareentwicklung begleiten.
- Neben dem Entwickeln ist auch das Betreiben der Software Bestandteil der Softwaretechnik.
- Da komplexe Software zu erstellen und zu warten aufwendig ist, erfolgt die Entwicklung von Software durch Softwareentwickler anhand eines strukturierten Planes.

Phasen des Software Engineering (1)

Kernprozesse:

1. Planung
   - ⇨ Projektmanagement
   - ⇨ Lastenheft (Anforderungsdefinition)
   - ⇨ Pflichtenheft (Mit technischen Ansätzen verfeinertes Lastenheft)
   - ⇨ Aufwandsabschätzung

Phasen des Software Engineering (2)

Kernprozesse:

2. Analyse
   - ⇨ Anforderungsanalyse
   - ⇨ Datenanalyse
   - ⇨ Prozessanalyse
   - ⇨ Systemanalyse
   - ⇨ Strukturierte Analyse (SA)
   - ⇨ Objektorientierte Analyse (OOA)

Phasen des Software Engineering (3)

Kernprozesse:

3. Entwurf
   - ⇨ Softwarearchitektur
   - ⇨ Strukturiertes Design (SD)
   - ⇨ Objektorientiertes Design (OOD)

Bis hier wurde keine einzige Zeile lauffähiger Code produziert!

Phasen des Software Engineering (4)

Kernprozesse:

4. Programmierung
   - ⇨ Normierte Programmierung
   - ⇨ Strukturierte Programmierung
   - ⇨ Objektorientierte Programmierung (OOP)

Phasen des Software Engineering (5)

Kernprozesse:

5. Test
   - ⇨ Modultests (Low-Level-Test)
   - ⇨ Integrationstests (Low-Level-Test)
   - ⇨ Systemtests (High-Level-Test)
   - ⇨ Akzeptanztests (High-Level-Test)

Phasen des Software Engineering (6)

Unterstützungsprozesse:

6. Projektmanagement
   - ⇨ Incident Management
   - ⇨ Problem Management
   - ⇨ Change Management
   - ⇨ Release Management
   - ⇨ Configuration Management
   - ⇨ Application Management

Phasen des Software Engineering (7)

Unterstützungsprozesse:

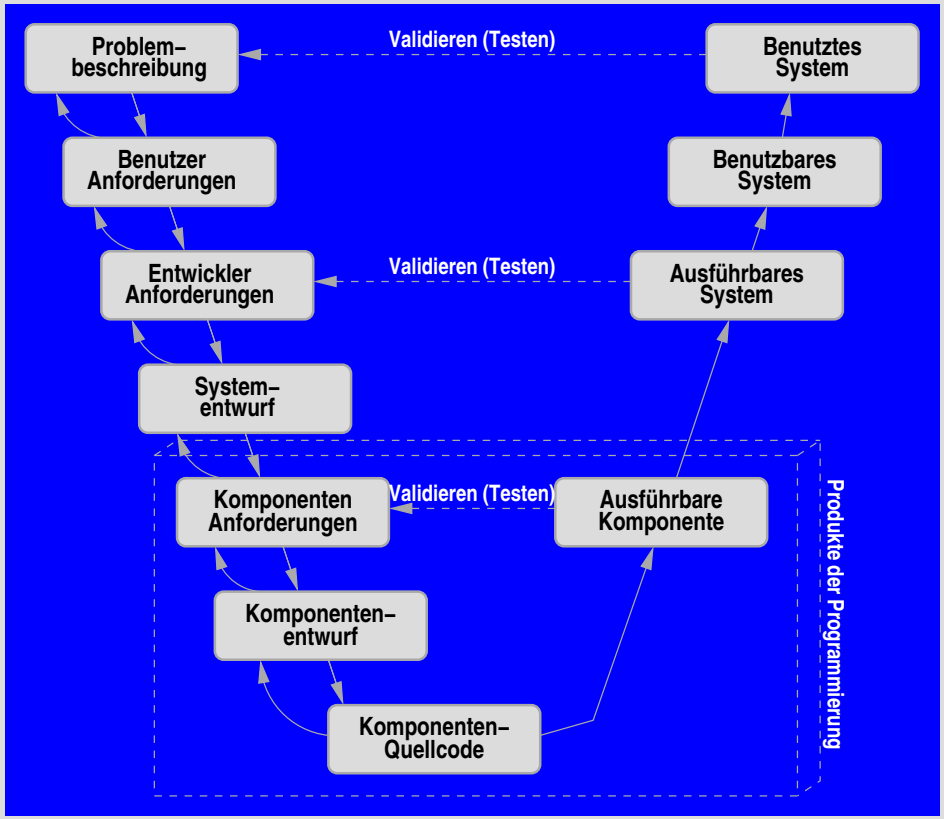
7. Qualitätsmanagement
   - ⇨ Softwareergonomie
   - ⇨ Softwaremetrik (Messung von Softwareeigenschaften)
8. Konfigurationsmanagement
   - ⇨ Versionsverwaltung
   - ⇨ Änderungsmanagement
   - ⇨ Software-Dokumentationswerkzeug

Phasen des Software Engineering (8)

Unterstützungsprozesse:

9. Dokumentation
   - ⇨ Systemdokumentation (Weiterentwicklung und Fehlerbehebung)
   - ⇨ Betriebsdokumentation (Betreiber/Service)
   - ⇨ Bedienungsanleitung (Anwender)
   - ⇨ Geschäftsprozesse (Konzeptionierung der Weiterentwicklung)
   - ⇨ Verfahrensdokumentation (Beschreibung rechtlich relevanter Softwareprozesse)

Das „V-Modell“ (s.a. ISO 12207)



Die Pfeile zwischen den einzelnen Schritten haben eine spezielle Bedeutung... ↵

Verifikation und Validierung

Verifikation: „Wird das Produkt richtig (fehlerfrei) entwickelt?“
Korrektheit - Arbeitet das Programm innerhalb der Spezifikation korrekt?

Validierung: „Wird das richtige Produkt entwickelt?“
Plausibilität - Genügt das Programm den Anforderungen?

Quiz: Was ist das/macht das?

```
"so oog"
```

Planung: Spezifikation Teil 1

- Projektmanagement (oder: Organisation in der Planungsphase)
- Lastenheft (Anforderungskatalog)
- Pflichtenheft (mit technischen Ansätzen)
- Aufwandsabschätzung

☞ Diese Punkte sind in der Praxis notwendige Voraussetzung, um eine Kostenabschätzung (Kostenvoranschlag, Angebot) überhaupt erstellen zu können.

Projektmanagement / Organisation

- Infrastrukturelle Maßnahme zur Gewährleistung eines effizienten Arbeitsablaufes im Team,
- i.d.R. hierarchisch aufgebaut: Zuständigkeiten für Teilprojekte mit Eskalationsmöglichkeit an übergeordnetes Management,
- bei kleineren Software-Entwicklungsunternehmen evtl. auch mehrere Funktionen in einer Person vereint (hohes Ausfallrisiko, kein Fallback),
- Hauptaufgabe in der Planungsphase: Mit vertretbarem Aufwand eine realistische Kostenabschätzung erstellen.

Definition Projektmanagement

Wikipedia.DE:
Die Norm DIN 69901 definiert entsprechend Projektmanagement als die *„Gesamtheit von Führungsaufgaben, -organisation, -techniken und -mitteln für die Abwicklung eines Projektes“*.

Der weltweit größte PM-Verband Project Management Institute (PMI) definiert den Projektmanagement-Begriff wie folgt: *„Project Management is the application of knowledge, skills, tools and techniques to project activities to meet project requirements“*.

Die Gesellschaft für Informatik definiert Projektmanagement so: *„Das Projekt führen, koordinieren, steuern und kontrollieren“*.

Anforderungen an den Projektmanager

Zur erfolgreichen Projektdurchführung werden vom Projektmanager die Wissensgebiete

- Projektmanagement (Methoden),
- allgem. Managementwissen und,
- produktspezifisches Wissen

benötigt.

Erwartungen an den Projektmanager

- Termine,
- Inhalt- und Umfang,
- Kosten

der zu erledigenden (Teil-)Arbeiten festlegen, kontrollieren und ggf. korrigieren.

Vorgehensweisen beim P.management

Die Wahl einer Vorgehensweise zur Durchführung eines Projekts hängt meist von folgenden Vorgaben ab:

- Vorgaben der Organisation oder des Auftraggebers,
- Größe und Komplexität des Projekts,
- Branche/Art des Projekt (Brückenbau vs. IT-Projekt).

Aufgaben im Projektmanagement

Konkrete Planungsaufgaben:

- Zeitplan mit Milestones,
- Zuständigkeiten-Hierarchie und Resourcen-Vereilung,
- Ablaufplan, Interaktionsdiagramme,
- Aktivitätsdiagramme (frühes Stadium, eher in der Analysephase notwendig).

☞ Entscheidung für ein *Vorgehensmodell* und Controlling.

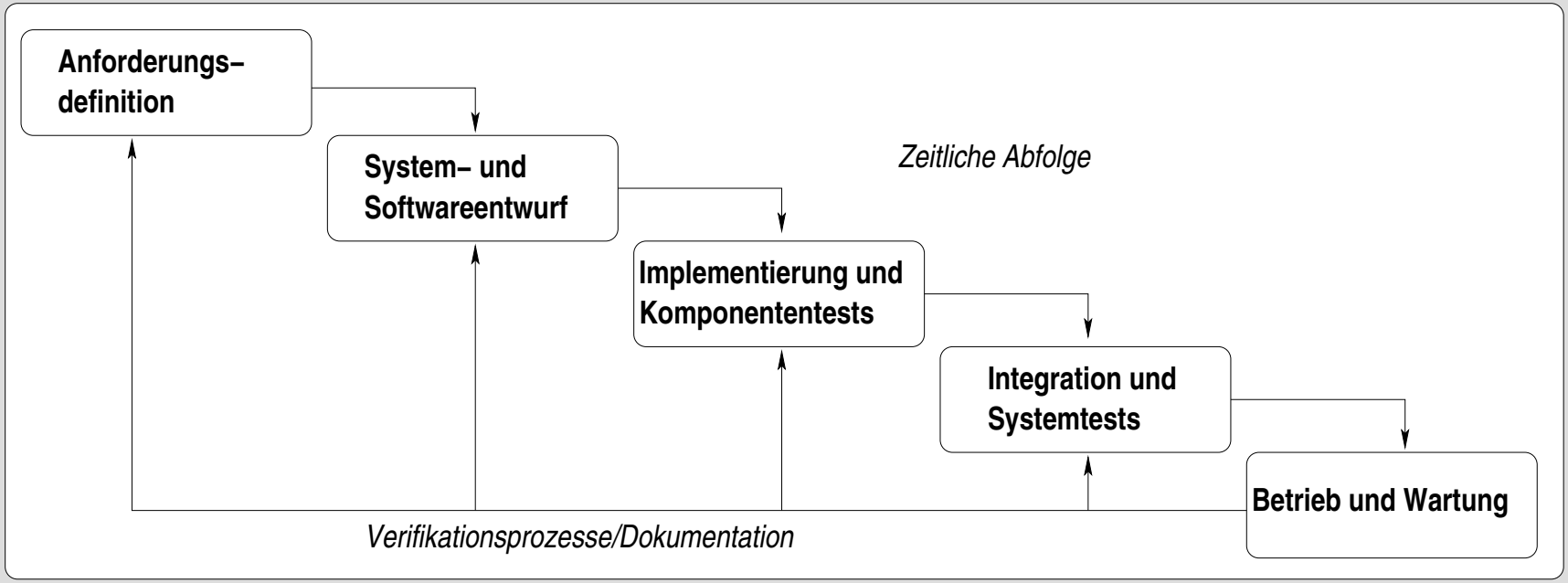
Projektmanagement: Vorgehensmodelle

Projekt-Modellierung:

- Wasserfallmodell,
- Spiralmodell,
- V-Modell (IT-Entwicklungsstandard der öffentlichen Hand in Deutschland / Modell der deutschen Bundesverwaltung),
- Extreme Programming (XP),
  - spezielle Modelle, die bei großen Softwarefirmen intern eingesetzt werden,
  - Evolutionäres Entwicklungsmodell,
  - ...

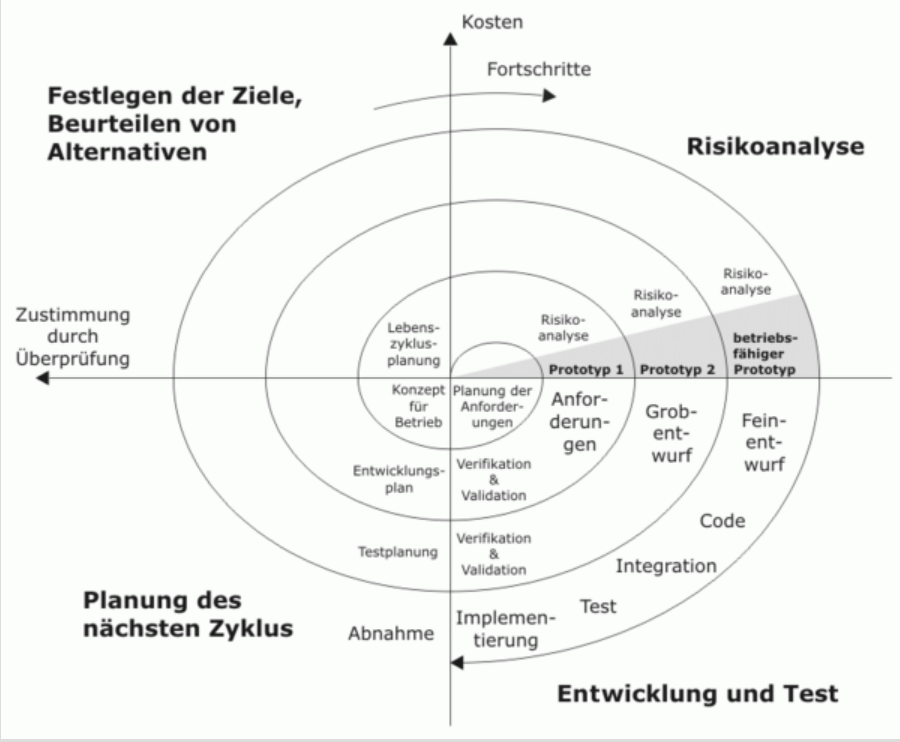
Wasserfallmodell

Das Wasserfallmodell bildet den Softwareentwicklungsprozesses in Phasen ab, wobei Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben für die nächst tiefere Phase eingehen.



Spiralmodell

Das Spiralmodell ist ein Risiko-orientiertes Entwicklungsmodell, das einzelne Phasen der Entwicklung iteriert und verfeinert, um das Ergebnis den Erwartungen/Spezifikationen schrittweise anzugleichen.



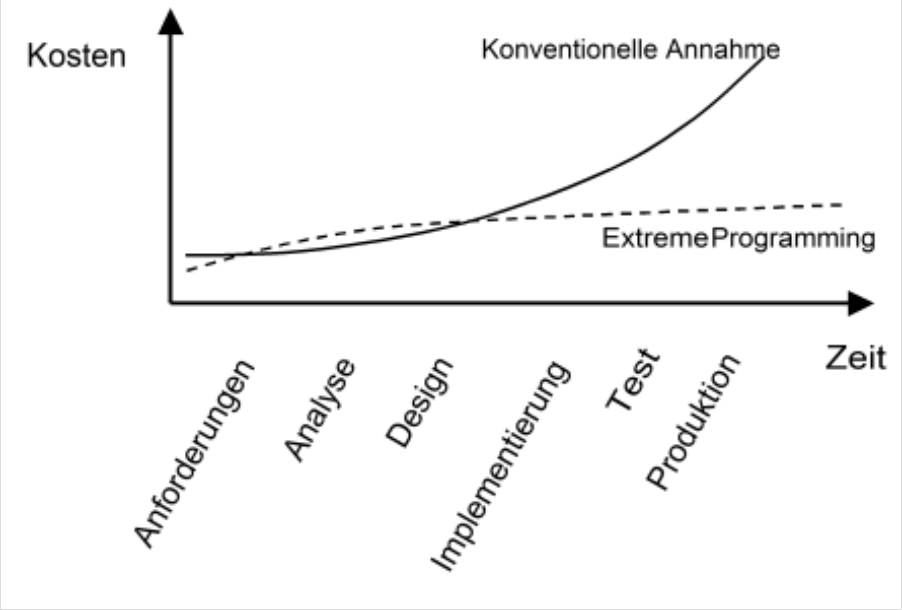
Evolutionäre Entwicklung

Dies ist eine eher „unkoordinierte“ Herangehensweise, bei der mit einem kleinen Teilprojekt begonnen, und dieses im Laufe des Softwarelebenszyklus kontinuierlich erweitert wird bis zu einem „akzeptablen“ Zustand. Dadurch ist zwar sehr schnell ein „Ergebnis“ verfügbar, die Erweiterbarkeit und Wartbarkeit leidet jedoch unter dem dynamischen Wachstum genau wie bei der Softwareentwicklung ohne „Engineering“.

☞ Nach einiger „Abnutzung“ der Software ist Re-Engineering oder sogar „Rewrite from scratch“ oft kostengünstiger als die kontinuierliche Weiterentwicklung.

Extreme Programming

Trotz des „fetzigen Namens“ geht es beim Extreme Programming eher um eine weniger festgelegte Vorgehensweise, die sogar zu Projektbeginn auf ein Pflichtenheft verzichtet, das quasi während der Implementation „mitentwickelt“ wird. D.h. die Anforderungen werden im Laufe des Projektes erst festgelegt, was für die Entwickler bezüglich des Designs eine hohe Herausforderung darstellen kann, damit nicht die gleichen Probleme wie beim Evolutionären Entwicklungsmodell entstehen.



Agile Software-Entwicklung

Dem Komplexitätsproblem wird in den neueren Konzepten der Agilen Softwareentwicklung mit zunehmender, koordinierter Flexibilität (manchmal sogar ganz ohne „festen Plan“) begegnet.
Bei Agiler Software Entwicklung werden statt eines starren Schemas (vergl. Wasserfallmodell) *Werte* definiert, die das Arbeiten im Team und die Kommunikation mit dem Auftraggeber verbessern sollen:[a]

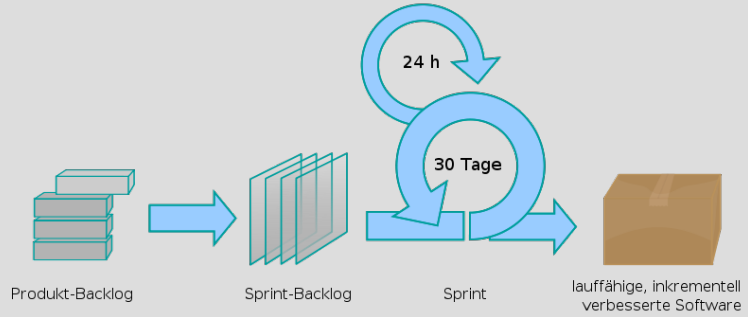
1. *Individuen und Interaktionen gelten mehr als Prozesse und Tools.*
2. *Funktionierende Programme gelten mehr als ausführliche Dokumentation.*
3. *Die stetige Zusammenarbeit mit dem Kunden steht über Verträgen.*
4. *Der Mut und die Offenheit für Änderungen steht über dem Befolgen eines festgelegten Plans.*

---

[a] *Agiles Manifest*, 2001, Ken Schwaber, Jeff Sutherland et al.

Agile Software-Entwicklung: Scrum (1)

Neben dem zuvor bereits genannten *Extreme Programming* kommt bei hochkomplexen Projekten das Vorgehensmodell Scrum[a] zum Einsatz, das auf Wiederholung bzw. Neuformulierung von Anforderungen nach jeweils einer Phase der Implementation („Sprint“) und den daraus resultierenden Erfahrungen basiert.



[a] „Gedränge“, der Terminologie beim Rugby-Spiel entlehnt

Agile Software-Entwicklung: Scrum (2)

Das Scrum-Vorgehensmodell geht davon aus, dass der tatsächliche Entwicklungsprozess kaum präzise plan- oder vorhersehbar ist, dennoch soll gelten:

> *„Das Produkt ist die bestmögliche Software unter Berücksichtigung der Kosten, der Funktionalität, der Zeit und der Qualität.“*[a].

[a] Ken Schwaber, OOPSLA 1995

Agile Software-Entwicklung: Scrum (3)

Einige Rollen und Begriffe, die in Scrum verwendet werden („Scrum Master“, „Zeremonie“, „Artefakte“ ...), lassen Scrum gelegentlich eher als Rollenspiel oder esoterische Veranstaltung erscheinen, es werden nach diesem Konzept aber tatsächlich große Projekte in der Praxis umgesetzt.

Normen

Die Aufgabenstellungen, Methoden, Instrumente und Ebenen des Projekt- und Qualitätsmanagements sind im Wesentlichen gut dokumentiert, und auch Gegenstand anderer Veranstaltungen (Vorlesungen, Kurse). Wir beschränken uns daher auf die praktischen Aspekte im IT-Bereich.

Normen: DIN 69900-1, DIN 69900-2, DIN 69901 bis 69905.

Als internationaler Leitfaden für Qualitätsmanagement in Projekten ist die Norm ISO 10006:2003 veröffentlicht worden.

Lastenheft

Aufbau:

1. Ausgangssituation und Zielsetzung,
2. Produkteinsatz
3. Produktübersicht
4. Funktionale Anforderungen, Nichtfunktionale Anforderungen (Funktionalität, Benutzbarkeit, Zuverlässigkeit, Effizienz, Änderbarkeit, Übertragbarkeit)
5. Risikoakzeptanz
6. Skizze des Lebenszyklus und der Systemarchitektur
7. Lieferumfang
8. Abnahmekriterien

Pflichtenheft (1)

Aufbau (lt. Balzert):

- Zielbestimmung (Musskriterien, Wunschkriterien, Abgrenzungskriterien)
- Produkteinsatz (Anwendungsbereiche, Zielgruppen, Betriebsbedingungen)
- Produktübersicht, Produktfunktionen (Details)
- Produktdaten (aus Benutzersicht zu speichernde)
- Produktleistungen: Anforderungen bezüglich Zeit und Genauigkeit
- Qualitätsanforderungen
- Benutzungsoberfläche: grundlegende Anforderungen, Zugriffsrechte
- Nichtfunktionale Anforderungen: Gesetze und Normen, Sicherheit, Plattform

Pflichtenheft (2)

Fortsetzung:

- Technische Produktumgebung
  - Hardware: Server + Client getrennt
  - Software: Server + Client getrennt
  - Orgware: organisatorische Rahmenbedingungen
  - Produkt-Schnittstellen
- Anforderungen an die Entwicklungsumgebung
- Gliederung in Teilprodukte

Auflösung des `"so oog"` Rätsels

(Demo)

Es handelt sich also um eine auf einen bestimmten Anwendungszweck hin optimierte Software-Lösung!

Generell kann die Wahl einer geeigneten Programmiersprache einen entscheidenden Einfluss auf den zu investierenden Programmieraufwand in der Implementierungsphase haben.

Rechtliche Rahmenbedingungen

(Kurze Wiederholung aus der Softwaretechnik-Vorlesung)

- Urheberrecht
- Wettbewerbsrecht
- Produkthaftung
- Überlassungsmodelle (Lizenzen)
  - Verkauf (selten)
  - Nutzung / Miete (entgeltlich oder unentgeltlich)
  - Open Source / Freie Software (weitgehende Übertragung der Verwertungsrechte auf den Lizenznehmer)
- Außerhalb Europas: Patentrecht (Software an sich ist nach europäischem Recht derzeit nicht patentierbar!)

Wer legt die Software-Lizenz fest?

Der Urheber.

Im Gegensatz zu den Verwertungsrechten ist das Urheberrecht nicht abtretbar.

Für wen gilt eine Lizenz?

Eine Lizenz gilt für die in der Lizenz angegebenen Personenkreise (sofern nach landesspezifischen Gesetzen zulässig).

Beispiel: Die GNU GENERAL PUBLIC LICENSE gilt für

- alle legalen EMPFÄNGER der Software, die
- die Lizenz AKZEPTIERT haben.

„Wer liest schon Lizenzen?“

- Zumindest in Deutschland bedeutet das FEHLEN eines gültigen Lizenzvertrages, dass die Software NICHT ERWORBEN und NICHT EINGESETZT werden darf.
- In Deutschland gibt es seit der letzten Änderung des Urheberrechtes keine generelle Lizenz-Befreiung mehr.
- Wurde die Lizenz nicht gelesen, oder „nicht verstanden“ (weil z.B. nicht in der Landessprache des Empfängers vorhanden), so ist die rechtliche Bindung, und daraus resultierend, die Nutzungsmöglichkeit der Software, formal nicht gegeben.

Auch als „Freeware“ deklarierte Software ist hier keine Ausnahme. Wenn keine Lizenz beiliegt, die eine bestimmte Nutzungsart ausdrücklich ERLAUBT, gilt sie als VERBOTEN.

Lizenzen: Proprietäre Software

- Der Empfänger erwirbt mit dem Kauf eine eingeschränkte, i.d.R. nicht übertragbare *Nutzungs*lizenz.
- Der Empfänger darf die Software nicht analysieren („disassemble“-Ausschlussklausel).
- Der Empfänger darf die Software nicht verändern.
- Der Empfänger darf die Software nicht weitergeben oder weiterverkaufen.

Diese Restriktionen werden im Softwarebereich so breit akzeptiert, dass man fast schon von einem „traditionellen“ Modell sprechen kann.

Lizenzen: Open Source

- Open Source stellt Software-Quelltexte als Resource zur Verfügung.
- Open Source sichert dem Anwender (Benutzer und Programmierer) bestimmte Freiheiten.
- Open Source stellt eine Basis (Lizenz) für eine Zusammenarbeit von Gruppen (oder Firmen) zur Verfügung.

Freie Software (nach der Definition der Free Software Foundation) ist ein Spezialfall von Open Source.

Definition: http://www.opensource.org/.

Beispiel: Die GNU General Public License

gibt den *Empfängern* der Software das Recht, ohne Nutzungsgebühren

- die Software für alle Zwecke einzusetzen,
- die Software (mit Hilfe der Quelltexte) zu analysieren,
- die Software (mit Hilfe der Quelltexte) zu modifizieren,
- die Software in beliebiger Anzahl zu kopieren,
- die Software im Original oder in einer modifizierten Version weiterzugeben oder zu verkaufen, auch kommerziell, wobei die neuen Empfänger der Software diese ebenfalls unter den Konditionen der GPL erhalten.

http://www.gnu.org/

Die GNU General Public License ...

- zwingt NICHT zur Veröffentlichung/Herausgabe von Programm oder Quellcode,
- zwingt NICHT zur Offenlegung ALLER Software oder Geschäftsgeheimnisse,
- verbietet NICHT die kommerzielle Nutzung oder den Verkauf der Software,
- verbietet NICHT die parallele Nutzung, oder lose Kopplung mit proprietärer Software.

Aber: Die EMPFÄNGER der Software erhalten mit der GPL die gleichen Nutzungsrechte an der Software, die die HERSTELLER/DISTRIBUTOREN haben.

Tabelle: Rechte und Lizenzmodelle

	Nutzung kostenlos	frei kopierbar	zeitlich unbegrenzt nutzbar	Quelltext wird mitgeliefert	Modifikation erlaubt	Einbau in prop. Produkte erlaubt	Derivate mit anderen Lizenzen mögl.
proprietäre Software							
Shareware	✔	✔					
Freeware	✔	✔	✔				
GPL	✔	✔	✔	✔	✔		
LGPL	✔	✔	✔	✔	✔	✔	
BSD	✔	✔	✔	✔	✔	✔	✔

Vor- und Nachteile für SW-Hersteller

Wer an der NUTZUNG der Software (pro-Kopie) verdienen möchte, wird sich eher für das proprietäre Lizenzmodell entscheiden, bei dem der Anwender keine Eigentumsrechte an der Software erwerben, und den Anbieter nicht wechseln kann.

Wer an SERVICE und SUPPORT (= Dienstleistungen) verdienen möchte, wird sich eher für das Open Source Modell entscheiden, bei dem der Kunde ein weitgehendes Eigentumsrecht an der Software erhält, und die Nutzung und Verbreitung der Software nicht eingeschränkt wird.

☞ Natürlich wissen die Kunden früher oder später auch um die für sie relevanten Vor- und Nachteile diverser Lizenzmodelle Bescheid, und werden sich ggf. mittel- bis langfristig entsprechend für einen kosteneffizienten Anbieter entscheiden, der Ihnen das passende Lizenzmodell liefern kann.

Produkthaftung

Ein genereller Haftungsausschluss, wie die folgende „empfohlene“ Zeile in der GNU GENERAL PUBLIC LICENSE:

```
This program is distributed in the hope that it will
be useful, but WITHOUT ANY WARRANTY; without even the
implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public
License for more details.
```

ist in Deutschland aufgrund des hier geltenden Produkthaftungsrechtes unwirksam. Dadurch wird nicht die komplette GNU GENERAL PUBLIC LICENSE ungültig, aber der Autor wird auch nicht von seiner Sorgfaltspflicht freigestellt, wie es wohl von den Urhebern der GPL beabsichtigt war.

Autor/Distributor haften...

- für „Geschenke“ mindestens bei GROBER FAHRLÄSSIGKEIT,
- für „Verkäufe“ bei allen vom Verkäufer/Hersteller verschuldeten Fehlern.

Vergl. RA Jürgen Siepmann ☞ „Freie Software - rechtsfreier Raum?“ (PDF) u. ☞ „Lizenz- und haftungsrechtliche Fragen bei der kommerziellen Nutzung Freier Software“ (HTML-Version)

Risikominderung

Im Pflichtenheft/Angebot können die Konditionen festgelegt werden, unter denen der Software-Empfänger bestimmte Teile der Software „abnimmt“, und somit den Hersteller teilweise entlastet.

Bei Fehlern, die der Hersteller bzw. Distributor kommerzieller Software (auch bei Open Source) zu verantworten hat, ist jedoch nach deutschem Recht eine Nachbesserung oder Wandlung auf Verlangen des Kunden möglich, je nach vertraglicher Gestaltung.

Im Extremfall kann der Hersteller bzw. Distributor auch zu Schadenersatz verpflichtet sein, wenn durch nicht nach angemessener Frist behobene Fehler Schäden oder Einnahmen-Einbußen beim Kunden entstehen.

Fazit: Preiseinfluss rechtlicher Fragen

Lizenz- und Haftungsfragen können einen erheblichen Einfluss auf die Preisgestaltung haben. Das Risiko einer „unbezahlten“ Nachbesserung muss bei komplexer Software genauso berücksichtigt werden, wie die Möglichkeit zum lizenztechnisch einwandfreien „Code Reuse“, um Entwicklungskosten einzusparen.

Auch die Kosten für die Qualitätssicherung von Software, die in unternehmenskritischen Bereichen eingesetzt werden soll, dürfen nicht vernachlässigt werden, da ein kommerzieller Softwareanbieter für Fehler nach deutschem Recht voll haftbar ist. Auch für Fehler dritter, deren Software er (mit-)verkauft.

Analysephase

1. Anforderungsanalyse
2. Datenanalyse
3. Prozessanalyse
4. Systemanalyse
5. Strukturierte Analyse (SA)
6. Objektorientierte Analyse (OOA)

Anforderungsanalyse (1)

Anforderungsanalyse (engl.: Requirements Engineering, RE) ist ein wichtiger Teil des Softwareentwicklungsprozesses.

Ziel: Die Anforderungen des Auftraggebers an das zu entwickelnde System (i.d.R. ein Anwendungsprogramm) ermitteln.

Mittel: Anforderungen werden dabei in einem Dokument (Anforderungskatalog), schriftlich fixiert.

☞ Software Requirements Specification (Std 830-1998)

Anforderungsanalyse (2)

- ... hat entscheidenden Einfluss auf den Prozess/Projektverlauf
- ... ist ein wesentlicher Bestandteil, um Qualität und Produktivität zu sichern
- ... wirkt sich auch auf die Akzeptanz und die Wartbarkeit aus, da die Anforderungen zusammen mit dem Kunden und Anwendern (idealerweise) im direkten Dialog entwickelt werden.

Anforderungsanalyse (3)

Probleme:

- Fachbegriffe aus der Softwaretechnik sind den Anwendern/Auftraggebern nicht geläufig,
- Fachbegriffe aus dem Fachgebiet des Kunden, und die damit verbundenen Prozesse, sind den Entwicklern nicht geläufig.

☞ Interpretationsfunktion (s.a. „Grundlagen der Informatik“) zur „Übersetzung“ der beiden Begriffswelten notwendig.
☞ Begriffsdefinitionen im Dokument vorsehen!

Anforderungsanalyse (4)

Phasen der Anforderungsanalyse:

1. Aufnahme und Erfassung
2. Strukturierung und Abstimmung
3. Prüfung und Bewertung

Aufnahme und Erfassung (1)

- Vollständigkeit: alle Anforderungen des Kunden müssen explizit beschrieben sein, es darf keine impliziten Annahmen des Kunden über das zu entwickelnde System geben.
- Eindeutigkeit: Definitionen und Abgrenzungen, präzise Definitionen helfen, Missverständnisse zwischen Entwickler und Auftraggeber zu vermeiden.
- Verständlichkeit: sowohl der Auftraggeber als auch der Entwickler sollen mit vertretbarem Aufwand die gesamte Anforderung lesen und verstehen können.
- Atomizität: es darf nur eine Anforderung pro Abschnitt der Anforderungs-Dokumentation beschrieben sein.

Aufnahme und Erfassung (2)

- Identifizierbarkeit: jede Anforderung muss eindeutig identifizierbar sein (z. B. über eine Kennung oder Nummer).
- Einheitlichkeit: die Anforderungen und ihre Quellen sollten nicht in unterschiedlichen Dokumenten stehen oder unterschiedliche Strukturen haben („zwei Namensräume für die gleiche Sache“ vermeiden).
- Einhaltung gesetzlicher Vorschriften
- Nachprüfbarkeit: die Anforderungen sollten mit Abnahmekriterien verknüpft werden, damit bei der Abnahme geprüft werden kann, ob die Anforderungen erfüllt wurden (Ableitung von Testfällen aus den Abnahmekriterien).

Aufnahme und Erfassung (3)

- ➪ Rück- und Vorwärtsverfolgbarkeit: Es soll erkennbar sein, ob jede Anforderung vollständig erfüllt wurde und umgekehrt für jede implementierte Funktionalität erkennbar sein, aus welcher Anforderung sie resultiert (um „überflüssige“ Zusatzfeatures zu vermeiden).
- ☞ Das Ergebnis der Anforderungsaufnahme ist das Lastenheft.

☞ Aber hatten wir das Lastenheft nicht schon unter Abschnitt 1 des Software Engineering, „Planungsphase“, erstellt?

Einwurf: Parallelität und Redundanz (1)

1. Viele der Kernprozesse im Software Engineering können parallelisiert werden, sofern die Abfolge nicht zeitlich festgelegt ist.
2. Einige der Kernprozesse sind auch nur alternative Darstellungsweisen (Views) von Methoden, die nicht unbedingt alle zum Einsatz kommen müssen.
3. Jeder Abschnitt des Software Engineering kostet Arbeit und somit Geld, daher wird versucht, einige der Aufgaben zunächst im „Überflugsverfahren“ ohne hohen Aufwand zu realisieren, und erst nach einer Bestätigung der Realisierbarkeit (Manpower, Finanzierung, Zusagen, Klärung der rechtlichen Aspekte und Formalitäten) im Detail auszuarbeiten.
4. Dadurch entsteht eine gewisse Redundanz, die jedoch auch als iterative Verfeinerung und Validierung dienen, und die Qualität des Engineering-Prozesses verbessern kann.

Einwurf: Parallelität und Redundanz (2)

Bezogen auf das Beispiel des Lastenheftes bedeutet dies, dass unser erster Entwurf (nach „vorläufigen“ Kriterien und einem Fragebogen/-Sondierungsgespräch) des Lastenheftes, erst in der ausführlicheren Analysephase seine endgültige Form erhält.

Strukturierung und Abstimmung

Nach der Erfassung muss eine Strukturierung und Klassifizierung der Anforderungen vorgenommen werden. Damit erreicht man, dass die Anforderungen übersichtlicher werden. Dies wiederum erhöht das Verständnis von den Beziehungen zwischen den Anforderungen. Kriterien sind hierbei:

- Abhängigkeiten: Anforderungen müssen daraufhin überprüft werden, ob sie sich nur gemeinsam realisieren lassen oder ob eine Anforderung die Voraussetzung für eine andere ist.
- Zusammengehörigkeit: Anforderungen, die fachlich-logisch zusammen gehören, sollten auch gruppiert realisiert werden.
- Rollenbasiertheit: jede Benutzergruppe hat ihre eigene Sicht auf die Anforderungen, die damit unterstützt werden soll. Daher sind oft verschiedene Darstellungsweisen und Ansichten auf den gleichen Sachverhalt sinnvoll (Anwendersicht, Administratorsicht, ...).

Prüfung und Bewertung

Nach der Strukturierung, zum Teil auch parallel dazu, erfolgt die Qualitätssicherung der Anforderungen nach Qualitätsmerkmalen:

- Korrektheit: die Anforderungen müssen widerspruchsfrei sein.
- Machbarkeit: die Anforderung muss, mit vertretbarem Aufwand, realisierbar sein.
- Notwendigkeit: was nicht vom Auftraggeber gefordert wird, ist keine Anforderung.
- Priorisierung: es muss erkennbar sein, welche Anforderungen am wichtigsten sind und daher bevorzugt entwickelt werden.
- Nützlichkeit, Nutzbarkeit: auch bei teilweiser Realisierung soll bereits ein produktives System entstehen.
- Benutzerfreundlichkeit.

☞ Das Ergebnis der Prüfung stellt die Basis für das verfeinerte Pflichtenheft dar.

Datenanalyse (1)

Unter Datenanalyse versteht man die Aufbereitung und Auswertung gesammelter Daten. Diese Daten können sowohl empirisch erhoben werden (beispielsweise durch Fragebögen und Umfragen), als auch - z.B. in Form von Dokumenten oder in Datenbanken - bereits gesammelt vorliegen (siehe auch Dokumentenanalyse).

Datenanalysen finden sowohl in der Marktforschung statt, als auch im Rahmen der Software-Anwendungsentwicklung, wo die Datenanalyse meist mit der Systemanalyse einher geht und eine der Grundlagen für das Pflichtenheft darstellt.

Datenanalyse (2)

Beim Software Engineering geht es weniger um das statistische Auswerten vorhandener Daten (wobei dies bei der Auswertung von Produktfragebögen durchaus auch der Fall sein kann), sondern um die technische Analyse verfügbarer projektrelevanter Datensätze, z.B. vorhandene Datenbanken oder verwendete Dokumente und Formulare, Dokumentationen und Spezifikationen, die im neuen System abgebildet werden sollen.

Prozessanalyse

Als Prozessanalyse bezeichnet man die systematische Untersuchung von Arbeitsabläufen (lat. procedere = voranschreiten) in ihren Einzelteilen, um Schwachstellen und Verbesserungspotentiale zu erkennen.

Die Prozessanalyse versucht durch das Zerlegen eines Vorgangs (vgl. Top-down-Methode) in seine Einzelschritte einen eventuell aufgetretenen Fehler oder Ungereimtheiten in dem Gesamtprozess sichtbar und Fehlerkorrekturen oder Verbesserungen möglich zu machen.

Durchführung der Prozessanalyse

Die Prozessanalyse wird in zwei Schritten durchgeführt:

1. Ist-Aufnahme der bestehenden Organisation
   Hierfür werden Organisations- und Arbeitsunterlagen ausgewertet und gegebenenfalls Mitarbeiterinterviews durchgeführt.
2. Ist-Analyse der Prozesse, z.B. durch:
   - Benchmarking
   - Schwachstellenanalyse
   - Workflowanalyse
   - Checklisten
   - Vorgangskettenanalyse

Systemanalyse

Bei der Systemanalyse konstruiert der Betrachter des Systems ein Modell aus Anwendersicht. Dabei trifft er eine Auswahl bezüglich der relevanten Elemente und Beziehungen des Systems.

Dieses Modell ist ein begrenztes, reduziertes, abstrahiertes Abbild der Wirklichkeit, mit dessen Hilfe Aussagen über vergangene und zukünftige Entwicklungen und Verhaltensweisen des Systems in bestimmten Szenarien gemacht werden sollen.

Schritte der Systemanalyse (1)

1. Festlegen der Systemgrenzen zur Unterscheidung von System und Umwelt.
2. Feststellen derjenigen Systemelemente, die für die Fragestellung als relevant betrachtet werden.
3. Feststellen derjenigen Beziehungen zwischen den Systemelementen, die für die Fragestellung als relevant betrachtet werden.
4. Feststellen der Systemeigenschaften auf der Makroebene.

Schritte der Systemanalyse (2)

5. Feststellen der Beziehungen des Systems zur Umwelt bzw. zu anderen Systemen, wenn von der Betrachtung des Systems als isoliertes oder geschlossenes System zum offenen System übergegangen wird.

6. Darstellung der Analyseergebnisse:

   - qualitativ: Flussdiagramm, Wirkungsdiagramme
   - halbquantitativ: Pfeildiagramm (je-desto-Beziehungen)
   - quantitativ: x-y-, x-t-Diagramme u. a., mathematische Gleichungssysteme

   Es werden sowohl formale Methoden und graphische Methoden für die Darstellung (Modell) eingesetzt.

Strukturierte Analyse (SA)

1. Die Strukturierte Analyse (SA) ist eine von Tom DeMarco entwickelte Methode zur Erstellung einer formalen Systembeschreibung im Rahmen der Softwareentwicklung.
2. Ergebnis der Strukturierten Analyse ist ein hierarchisch gegliedertes technisches Anforderungsdokument für das Systemverhalten.
3. Graphische Analysemethode.
4. Top-Down: komplexes System in immer einfachere Funktionen bzw. Prozesse aufteilen gleichzeitig eine Datenflussmodellierung durchführen.
5. Heute durch UML weitgehend abgelöst.

Objektorienierte Analyse (OOA)

Objektorientierte Analyse (OOA) bezeichnet die erste Phase der objektorientierten Erstellung eines Softwaresystems.

Sie ist ein Teil der objektorientierten Modellierung, welche sich in den Teil der Domänenmodellierung (Analyse) und den Teil des Systementwurfs (Design) aufgliedert.

In der OOA geht es darum, die Anforderungen zu beschreiben und zu klassifizieren, die das zu entwickelnde Softwaresystem erfüllen soll.

Objektorienierte Analyse (OOA)

Stark vereinfacht ausgedrückt sucht und sammelt man in dieser Phase alle Fakten zu einem Sachverhalt oder Prozess, stellt diese graphisch dar und überprüft sie (Lastenheft, Pflichtenheft). Das OOA-Modell ist eine fachliche Beschreibung mit objektorientierten Konzepten, fast immer mit Elementen der Unified Modeling Language (UML) notiert.

Bemerkenswert: Ein Bezug zur Informationstechnik wird in dieser Analyse-Phase ausdrücklich vermieden, sondern es wird viel mit, teils konkreten, teils abstrakten, Modellen gearbeitet.

Eine Unterteilung in ein statisches und ein dynamisches Teilmodell und eine Teildefinition von Benutzerschnittstellen ist üblich.

Ablauf der OOA (1)

Problembeschreibung des Systems:

- Pflichtenheft

Statische Analyse:

- Identifikation von Klassen, Objekten
- Identifikation von Assoziationen (Beziehungen) zwischen Objekten und Klassen
- Identifikation von Attributen (Eigenschaften) der Objekte und Klassen
- Organisation der Objektklassen z.B. mit Hilfe von Vererbungshierarchien

Ablauf der OOA (2)

Dynamische Analyse

- Szenarios entwickeln
- Beschreiben des Ereignisflusses: Interaktionsdiagramme (UML)
- Zustandsdiagramme entwickeln
- Identifikation der Geschäftsprozesse
- Erstellen eines Prototypen der Benutzeroberfläche

Unified Modeling Language

- Unified Modeling Language (UML) ist eine von der Object Management Group (OMG) entwickelte und standardisierte Sprache für die Modellierung von Software und anderen Systemen.
- Im Sinne einer formalen Sprache (s.a. Grundlagen der Informatik) definiert die UML dabei Bezeichner für die meisten Begriffe, die für die Modellierung wichtig sind, und legt mögliche Beziehungen zwischen diesen Begriffen fest.
- Die UML definiert weiter grafische Notationen für diese Begriffe und für Modelle von statischen Strukturen und von dynamischen Abläufen, die man mit diesen Begriffen formulieren kann.

UML (2)

Die UML ist heute eine der dominierenden Sprachen für die Modellierung von betrieblichen Anwendungssystemen (Softwaresystemen). Der erste Kontakt zur UML besteht häufig darin, dass Diagramme der UML im Rahmen von Softwareprojekten zu erstellen, zu verstehen oder zu beurteilen sind:

- Projektauftraggeber und Fachvertreter prüfen und bestätigen zum Beispiel Anforderungen an ein System, die Wirtschaftsanalytiker in Anwendungsfalldiagrammen (95) der UML festgehalten haben.
- Softwareentwickler realisieren Arbeitsabläufe, die Wirtschaftsanalytiker in Zusammenarbeit mit Fachvertretern in Aktivitätsdiagrammen (94) beschrieben haben.
- Systemingenieure installieren und betreiben Softwaresysteme basierend auf einem Installationsplan, der als Verteilungsdiagramm (93) vorliegt.

UML (3)

Die graphische Notation ist jedoch nur ein Aspekt, der durch die UML geregelt wird. Die UML legt in erster Linie fest, mit welchen Begriffen und welchen Beziehungen zwischen diesen Begriffen sogenannte Modelle spezifiziert werden - die Diagramme der UML zeigen dann eine graphische Sicht auf Ausschnitte dieser Modelle.

☞ Die UML schlägt ein Format vor, in dem Modelle und Diagramme zwischen Werkzeugen ausgetauscht werden können.

UML (4)

UML ist in sog. „Spracheinheiten“ (bzw. Elementen) spezifiziert, auf denen die verschiedenen Diagrammtypen aufbauen:

1. Spracheinheit Aktionen
2. Spracheinheit Aktivitäten
3. Spracheinheit Allgemeines Verhalten
4. Spracheinheit Anwendungsfälle
5. Spracheinheit Informationsflüsse
6. Spracheinheit Interaktionen
7. Spracheinheit Klassen
8. Spracheinheit Komponenten
9. Spracheinheit Kompositionsstrukturen
10. Spracheinheit Modelle
11. Spracheinheit Profile
12. Spracheinheit Schablonen
13. Spracheinheit Verteilungen
14. Spracheinheit Zustandsautomaten

UML (5)

UML 2.0 kennt die folgenden Diagrammtypen:

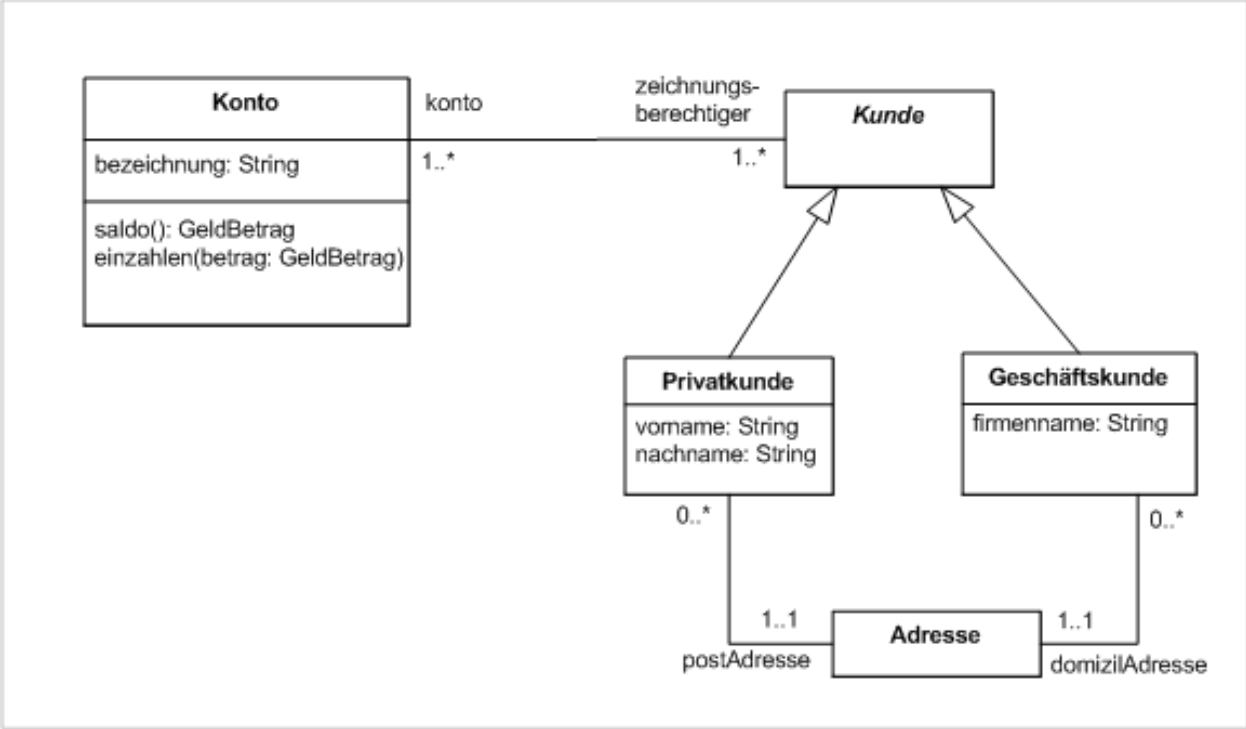
Strukturdiagramme

1. Klassendiagramm (86)
2. Komponentendiagramm (89)
3. Kompositionsstrukturdiagramm (90)
4. Objektdiagramm (91)
5. Paketdiagramm (92)
6. Verteilungsdiagramm (93)

Verhaltensdiagramme

1. Aktivitätsdiagramm (94)
2. Anwendungsfalldiagramm (95)
3. Interaktionsübersichtsdiagramm (96)
4. Kommunikationsdiagramm (97)
5. Sequenzdiagramm (98)
6. Zeitverlaufsdiagramm (99)
7. Zustandsdiagramm (100)

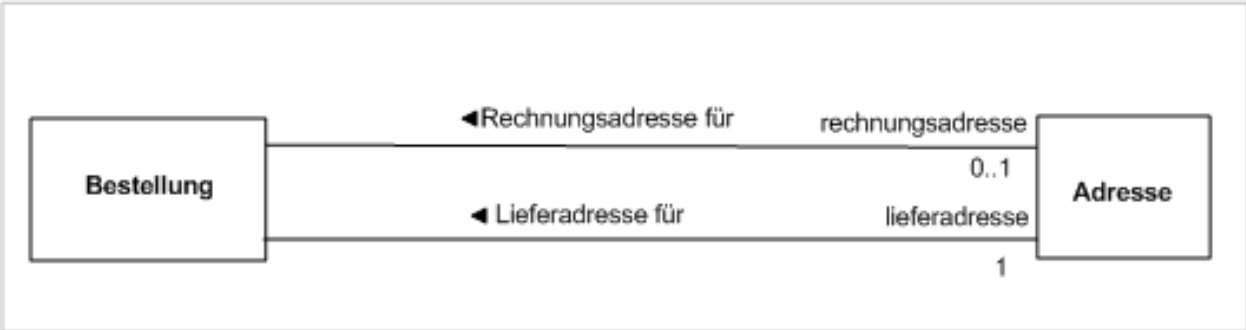
Klassendiagramm (1)



Besondere Merkmale: `Variable : Typ`, und: abgeleitete Klassen zeigen auf die Basisklassen, von denen sie Eigenschaften erben.

Klassendiagramm (2)

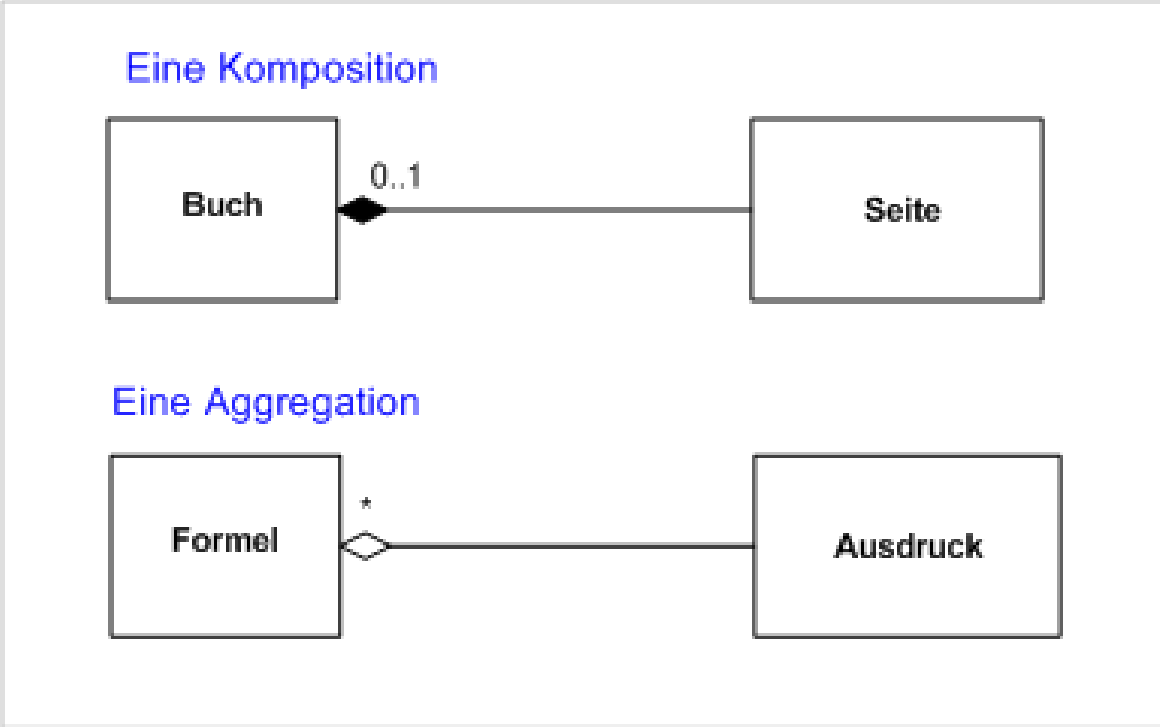
Beziehungen zwischen Objekten:



Besondere Merkmale: Die Anzahl möglicher Instanzen (von..bis) kann bei Beziehungen zwischen Objekten angegeben werden.

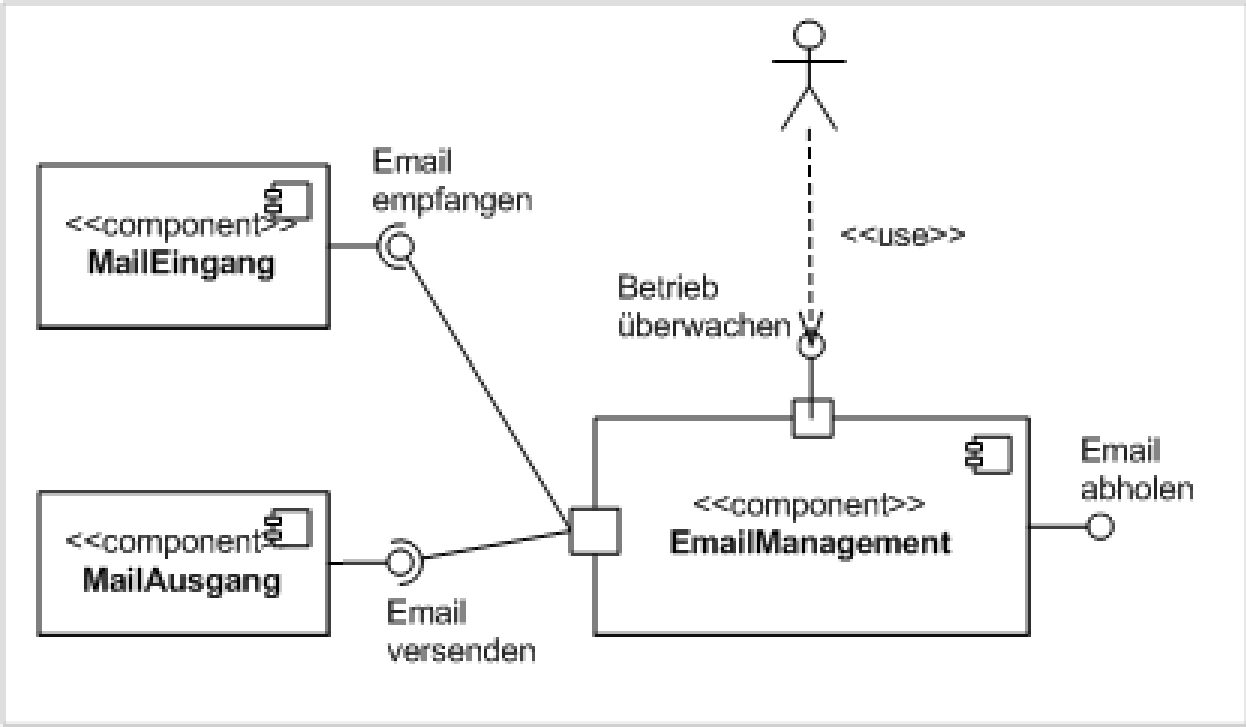
Klassendiagramm (3)

Beziehungen zwischen Objekten:



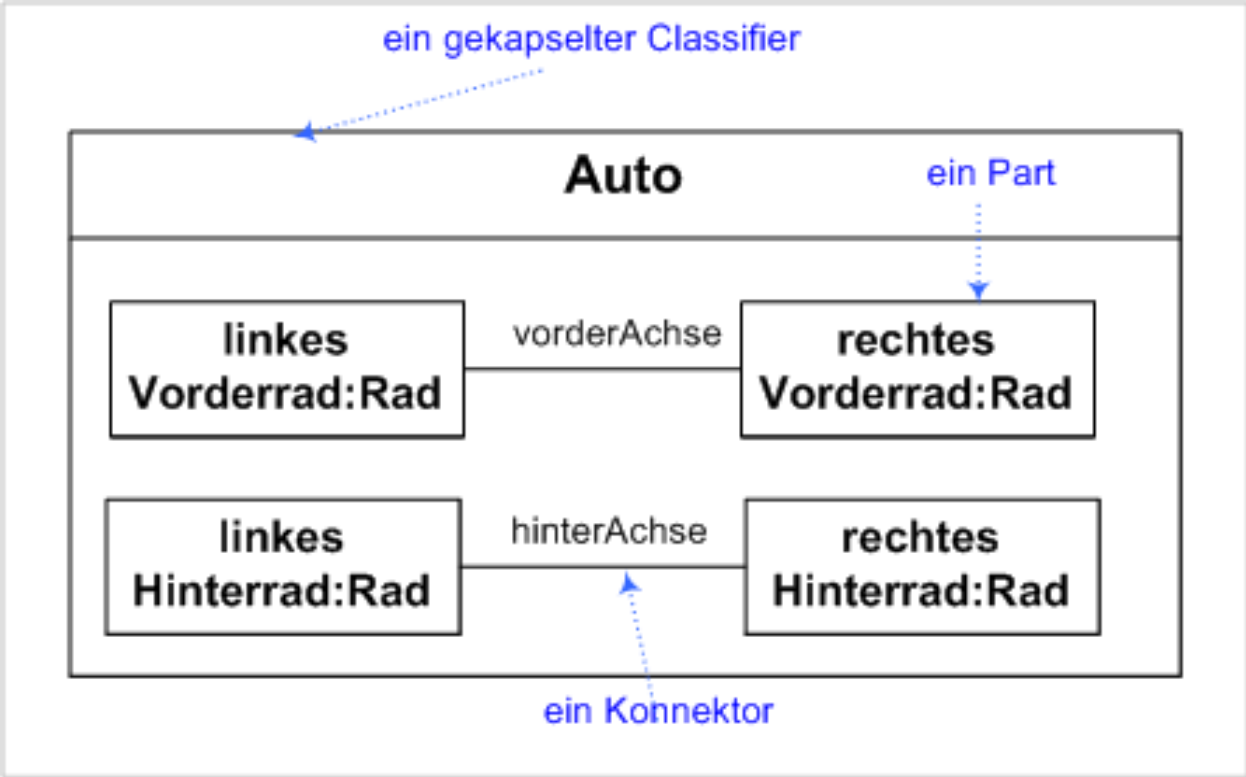
Besondere Merkmale: Die Unterscheidung zwischen ausgefüllter Raute (Komposition, „besteht aus“) und nicht-ausgefüllter Raute (Aggregation, „enthält“, beliebige Anzahl von Instanzen).

Komponentendiagramm



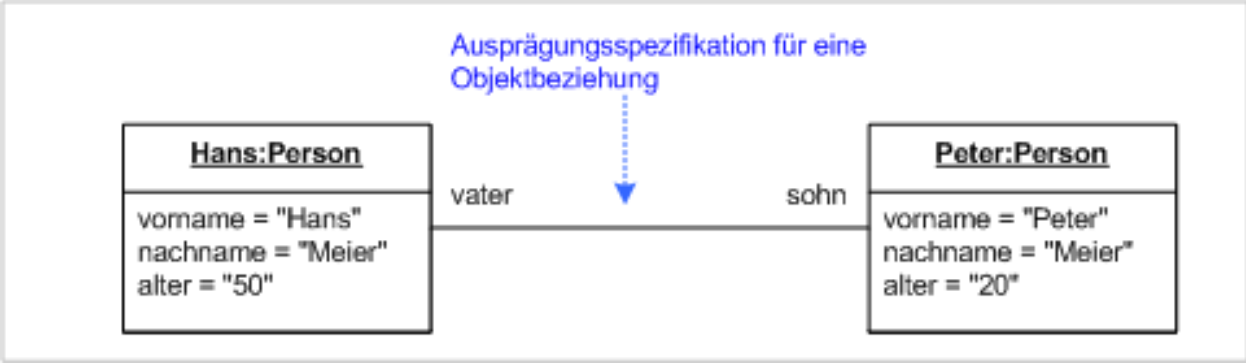
Besondere Merkmale: Zeigt (abstrakt) die Teile eines Ganzen, und die Schnittstellen und Abhängigkeiten dazwischen.

Kompositionsstrukturdiagramm



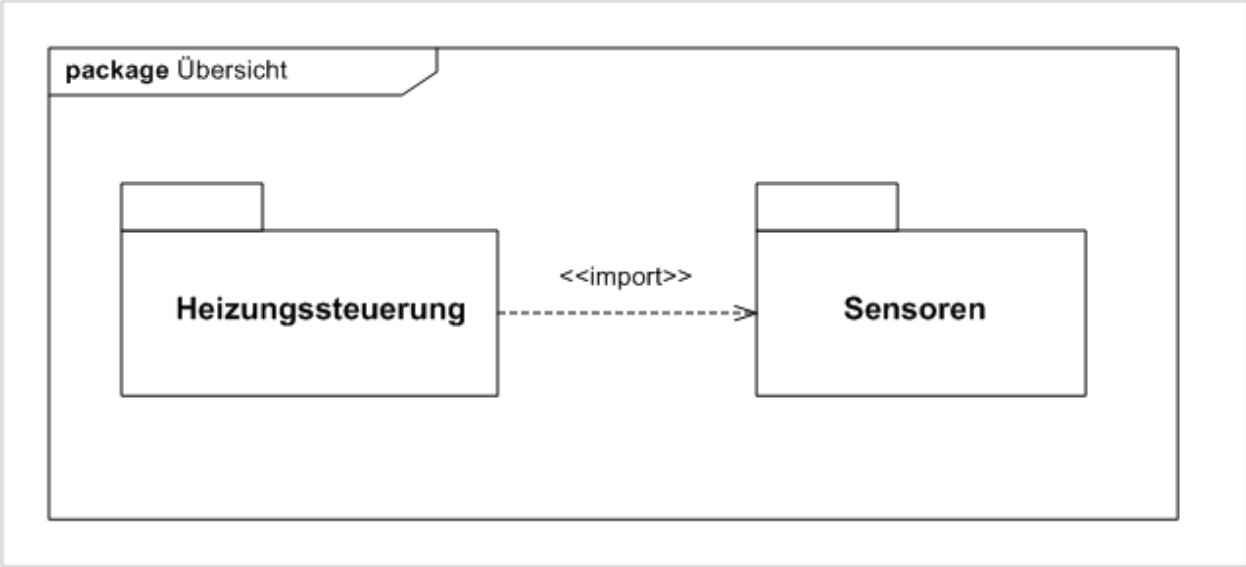
Besondere Merkmale: Zeigt das Innere einer Klasse, und ggf. die Kommunikationspfade der Teile untereinander.

Objektdiagramm



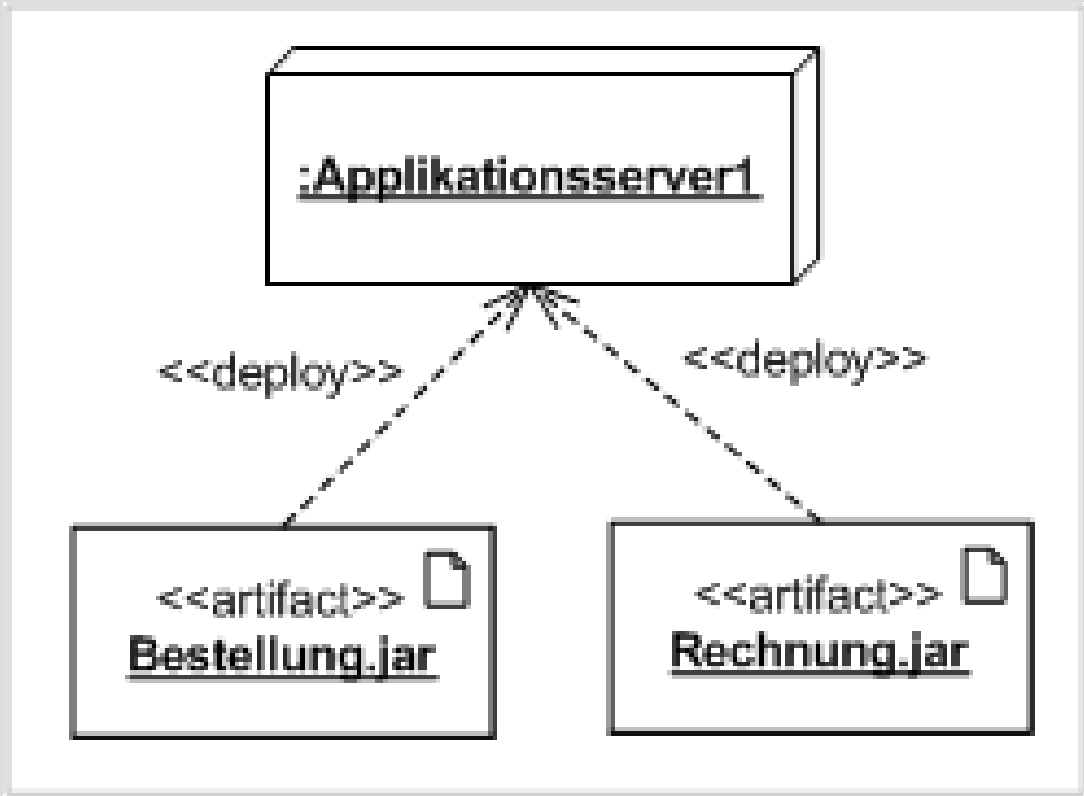
Besondere Merkmale: Zeigt, wie sich bestimmte Objekte als Instanzen einer Klasse untereinander verhalten.

Paketdiagramm



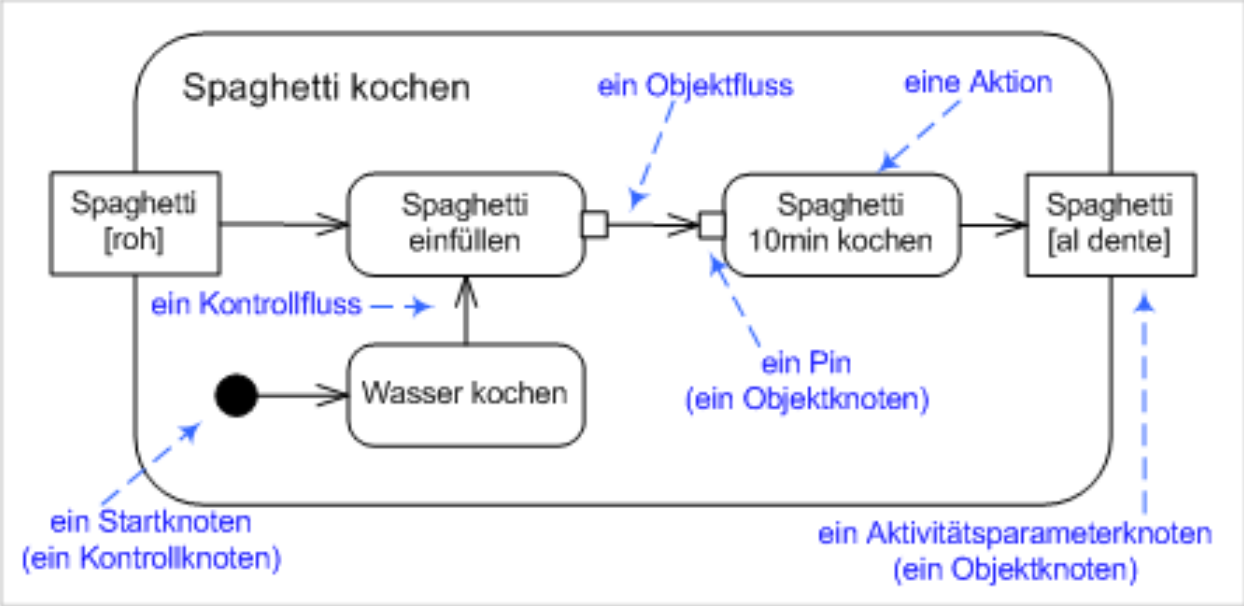
Besondere Merkmale: Stellt die Schichtung der Software, bzw. die Unterteilung der Software in Module dar. Bei Geschäftsmodellen werden Pakete oft benutzt, um fachlich zusammengehörende Modellteile zusammenzufassen, zum Beispiel zu Geschäftsfällen.

Verteilungsdiagramm



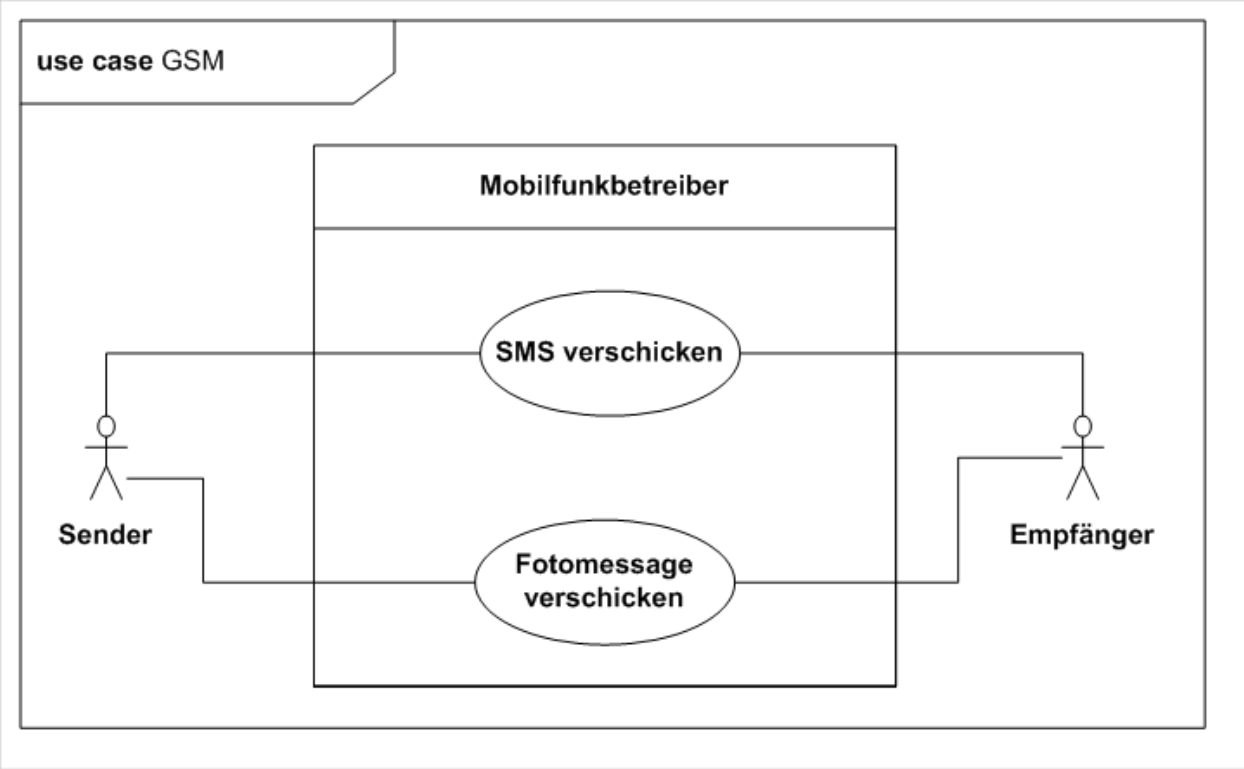
Besondere Merkmale: Hier geht es tatsächlich einfach um die Verteilung bzw. den Fluss von Information und Resourcen, was mit gestrichelten Pfeilen zwischen den beteiligten Akteuren dargestellt wird.

Aktivitätsdiagramm



Besondere Merkmale: Entspricht in der Funktionsweise genau dem Flussdiagramm, was auch heute noch gebräuchlich, aber nicht so gut normiert ist. Hier wird der Ablauf eines Prozesses dargestellt (vergl. auch Petri-Netze).

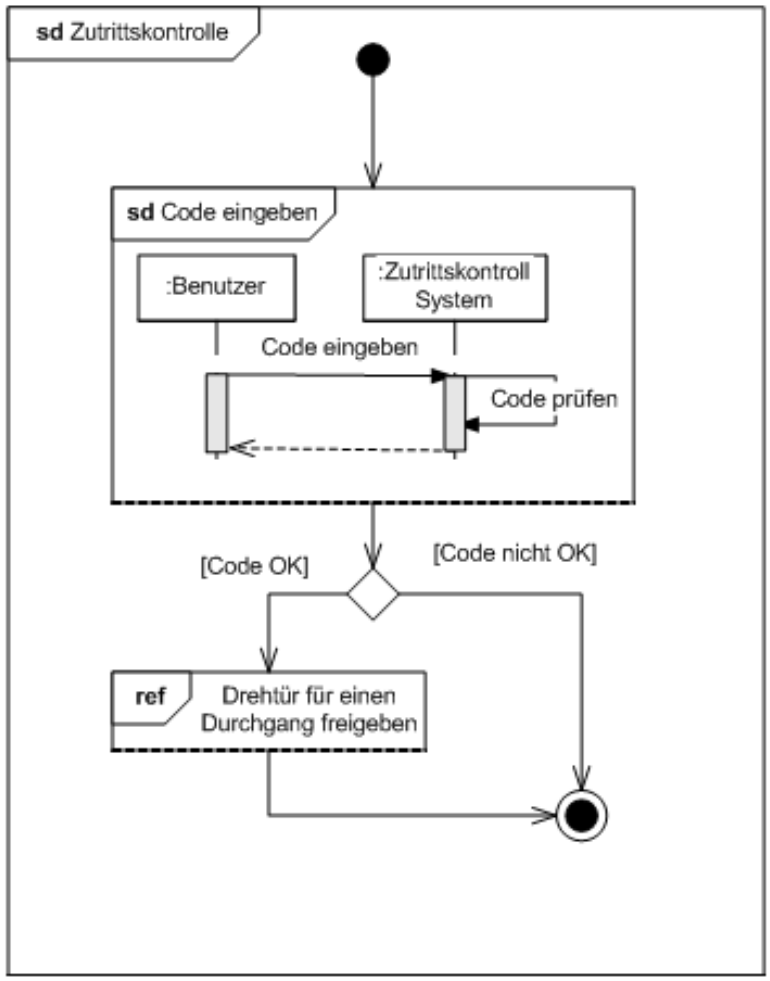
Anwendungsfalldiagramm



Besondere Merkmale: Beschreibt graphisch einen Beispielfall, in dem i.d.R. Personen als Strichmännchen dargestellt, und auch beteiligte Geräte skizziert werden.

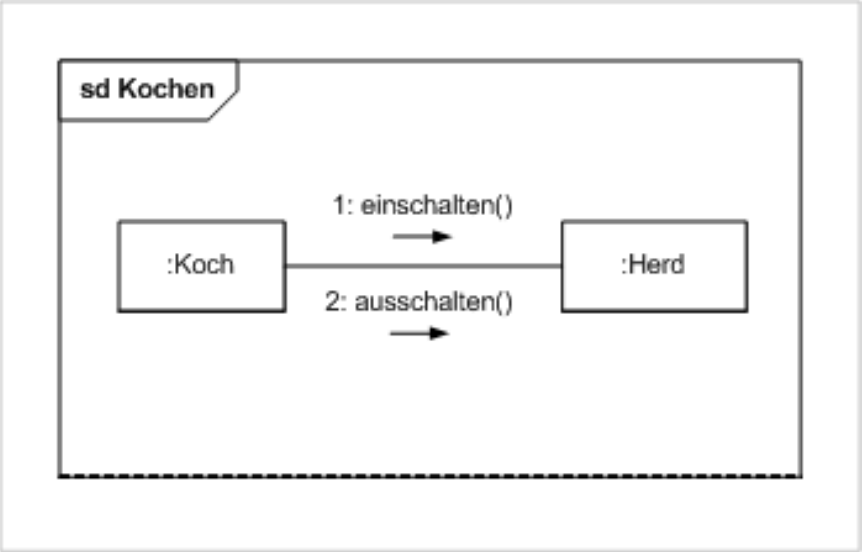
Interaktionsübersichtsdiagramm

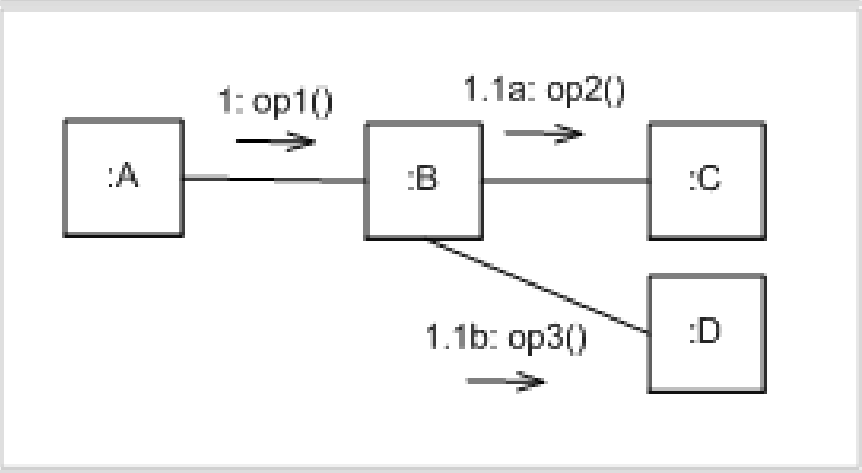
Besondere Merkmale:
Beschreibt mit graphischen Elementen, die meist denen des Aktivitätsdiagrammes sehr ähnlich sind, wie Komponenten miteinander interagieren.



Kommunikationsdiagramm

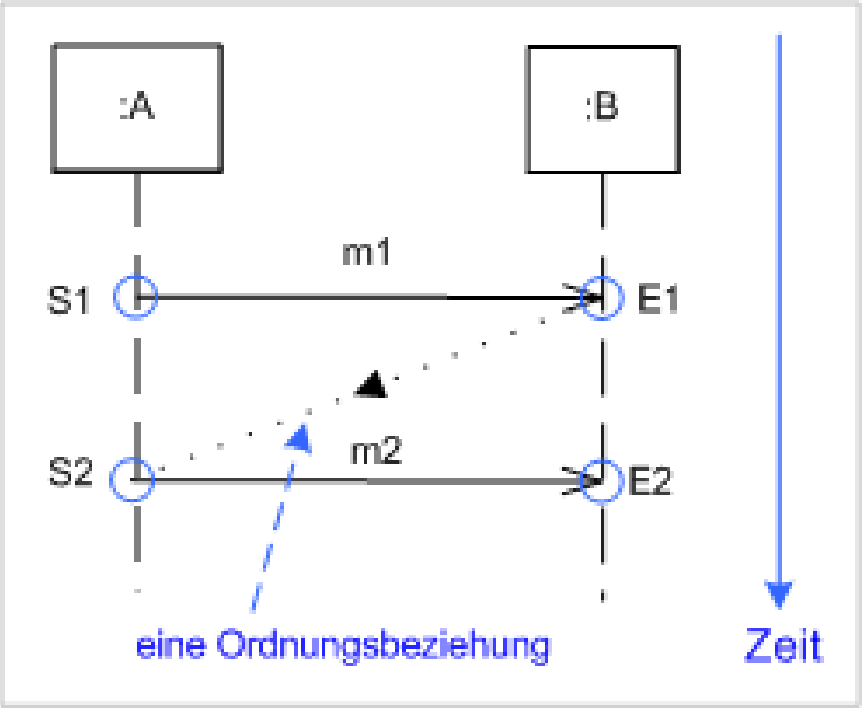
Besondere Merkmale:
Graphische Darstellung einer Interaktion, spezifiziert den Austausch von Nachrichten zwischen Instanzen, die im Diagramm als sog. „Lebenslinien“ dargestellt sind. Die Reihenfolge kann durch Nummerierung von Sequenzen, die Richtung durch Pfeile angegeben werden.



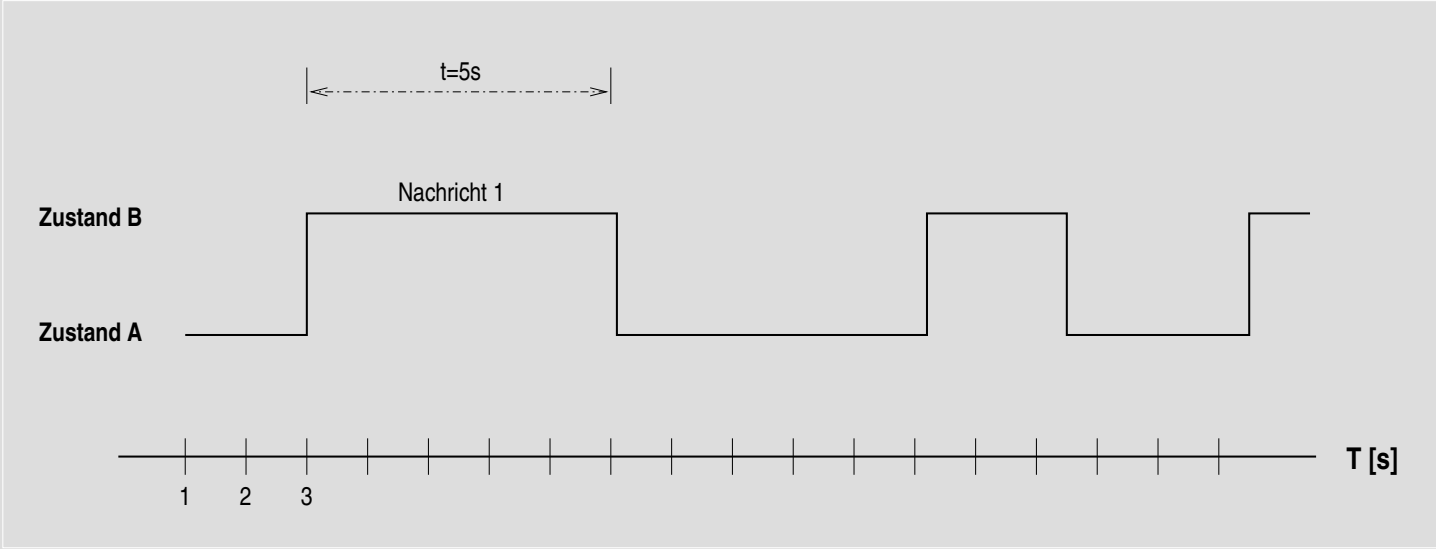


Sequenzdiagramm

Besondere Merkmale:
Zeigt den zeitlichen Ablauf beim Austausch von Information/Nachrichten.
Gefüllte Pfeilspitzen: Synchrone Kommunikation,
Einfache Pfeilspitzen: Asynchrone Kommunikation.
Ordnungsbeziehungen zeigen notwendige Bedingungen an.



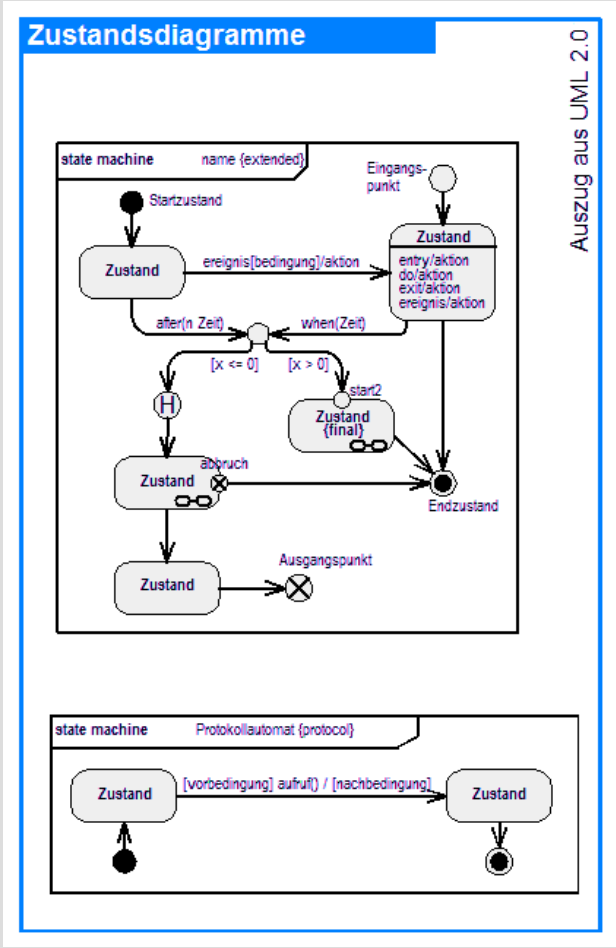
Zeitverlaufsdiagramm



Besondere Merkmale:
Zeigt das zeitliche Auftreten von Signalen oder Nachrichten, x,y-Diagramm, meist mit x als Zeit-Achse.

Zustandsdiagramm

Besondere Merkmale:
Zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann, und gibt die Ereignisse an, aufgrund welcher Zustandsänderungen stattfinden.



UML Fazit

- Teilweise (v.a. Aktivitätsdiagramm) auch ohne Informatik-Kenntnisse einfach zu verstehende, graphische Darstellung aus verschiedenen Sichtweisen.
- Normung veschiedener Darstellungsweisen.
- Auch automatische Abbildungen (Java Klassen + Klassendiagramm und umgekehrt) möglich und in diverse IDEs integriert.
- Komplexe Zusammenhänge werden strukturiert dargestellt, Abläufe untergliedert und nach Zeit oder Voraussetzungen modelliert.
- Nachteile: Rein visuell, schwer als rein beschreibende Darstellung umsetzbar, teilweise nicht eingängige Symbolik, Schwierigkeit, die „richtige“ Darstellung zu finden.

☞UML 2.0 Notationsübersicht

Entwurf(sphase)

Nach der ausgiebigen Analyse und Modellierung der in Software abzubildenden Prozesse, werden vor der eigentlichen Implementierung weitere, vorbereitende Schritte unternommen, um eine „perfekte“ Software (oder zumindest die „passende“ Software) zu erstellen.

Als Hilfsmittel zum „richtigen“ Entwurf wird eine oder mehrere der folgenden Methoden angewandt:

- Software-Architektur
- Strukturiertes Design (SD)
- Objektorientiertes Design (OOD)

Softwarearchitektur (1)

... beschreibt die grundlegenden Elemente und die Struktur eines Softwaresystems.

Helmut Balzert beschreibt den Begriff in seinem *Lehrbuch der Software-Technik* als:

> *„eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen.“*

Softwarearchitektur (2)

Bei der Systementwicklung repräsentiert die Softwarearchitektur die früheste Softwaredesign-Entscheidung (Architekturentwurf).

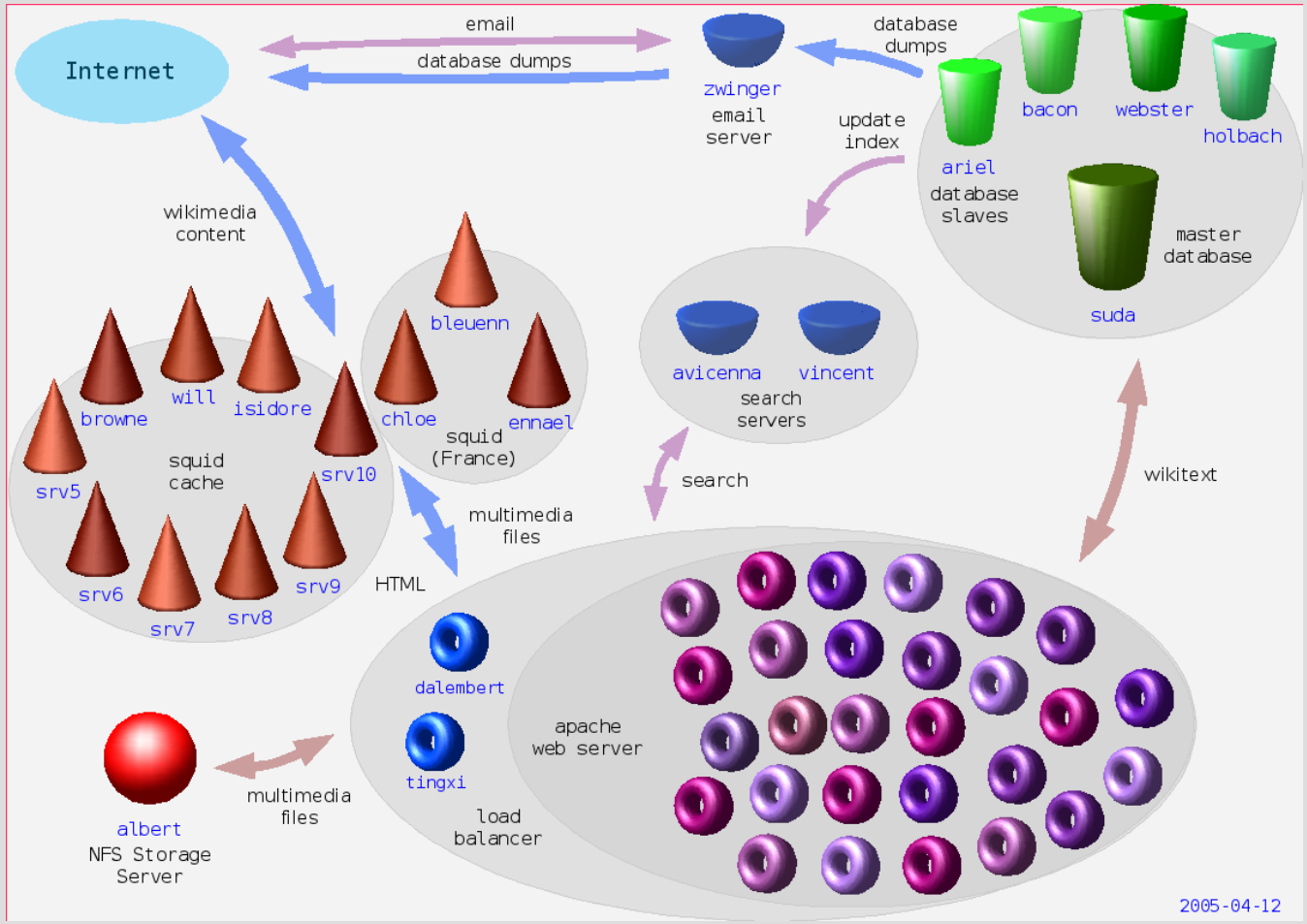
Eigenschaften wie Modifizierbarkeit, Wartbarkeit, Sicherheit oder Performanz sind von diesem Entwurf abhängig.

Eine einmal eingerichtete Softwarearchitektur ist später nur mit hohem Aufwand abänderbar. Die Entscheidung über ihr Design ist somit einer der kritischsten Punkte im Software-Entwicklungsprozess.

☞ Es besteht durchaus die Gefahr, dass ein Software-Architekt ein in der Theorie perfekt abgestimmtes Konzept entwirft, das die Programmierer später gar nicht, oder nicht optimal umsetzen können, aus technischen Gründen oder wegen nicht einkalkuliertem Zusatzaufwand.

Softwarearchitektur (3)

Beispiel für eine Architekturbeschreibung: Wikipedia



(Parade-Beispiel für die Client-Server Architektur verteilter Rechnersysteme.)

Beispiel Architektur „Echtzeitsysteme“ (1)

Anforderungen an ein Echtzeitsystem:

1. Möglichkeit, Aufgaben zeitgesteuert zu präzisen Zeitpunkten oder Intervallen abzuarbeiten,
2. definierbare Antwortzeiten,
3. Prioritäten, Unterbrechen von „weniger wichtigen“ Aufgaben durch „wichtigere“ Aufgaben muss möglich sein.

Ein Echtzeitsystem muss also nicht „unglaublich schnell“ arbeiten, sondern vielmehr die Abläufe präzise steuern können, und keine „unplanbaren“ Komponenten enthalten.

Beispiel Architektur „Echtzeitsysteme“ (2)

Generell muss das Betriebssystem die o.g. Eigenschaften unterstützen (z.B. Linux-Kernel mit aktivierter Realtime-Erweiterung). Der Software-Architekt muss auch eine Entscheidung bezüglich der geeigneten Werkzeuge und Programmiersprachen treffen.

- Geeignete Programmiersprachen: Assembler, C (systemnah, effizient, aber aufwändig wegen nicht eingebauten Nebenläufigkeits-Funktionen), evtl. spezielle Kontrollsprachen wie Ada,
- Ungeeignete Programmiersprachen: Z.B. Java (Systemverhalten und Antwortzeiten nicht planbar wg. z.B. nicht kontrollierbarer Garbage Collection).

Strukturiertes Design (1)

... ist eine Entwurfsmethode in der Softwaretechnik, welche ein modulares Design unterstützt, um neben der reinen Funktionshierarchie auch die Wechselwirkungen von übergeordneten Modulen zu beschreiben. SD wird mit der Strukturierte Analyse (SA) in der Softwaretechnik verwendet.

Das Strukturierte Design schlägt eine Brücke zwischen der technologieneutralen Analyse und der eigentlichen Implementierung. Im Strukturierten Design werden technische Randbedingungen eingebracht und die Grobstruktur des Systems aus technischer Sicht festgelegt. Es stellt damit die inhaltliche Planung der Implementierung dar.

Strukturiertes Design (2)

Die Methodik stellt mittels **Strukturdiagrammen** (s.a. UML) funktionale Module hierarchisch dar uns zeigt dadurch die einzelnen Aufrufhierarchien der Module untereinander. Ein funktionales Modul besteht aus einer oder mehreren funktionalen Abstraktionen. Diese wiederum stellt eine der ersten Abstraktionsmechanismen dar und gruppiert mehrere zusammengehörende Programmbefehle zu Einheiten (Funktionen).

Ein Beispiel für ein solches Modul wäre die Berechnung der Quadratwurzel `sqrt(x)`. Der Benutzer muss keine Details über die Implementierung wissen, sondern wendet die Funktion nur an.

```
// Signatur / Funktionsprototyp
// public static double Math.sqrt(double x);
double ergebnis = Math.sqrt(2.0);
```

Objektorientiertes Design (OOD)

Beim objektorientierten Design wird das in der objektorientierten Analyse (OOA) (77) erstellte Domänenmodell weiterentwickelt und darauf aufbauend ein objektorientierter Systementwurf erstellt.

Das Design berücksichtigt neben den fachlichen Aspekten des Auftraggebers aus der Analyse auch technische Gegebenheiten.

Beim Wasserfallmodell (30) würde als nächste Phase die objektorientierte Programmierung (OOP) folgen.

Objektorientierte Analyse, Objektorientiertes Design und Objektorientierte Programmierung sind heute (noch) die beliebtesten Methoden im Software Engineering, und werden entsprechend gut in diversen Werkzeugen (Tools) unterstützt.

Werkzeuge OOA/OOD (1)

Es gibt diverse Modellierungssoftware für die objektorientierte Analyse (OOA) und das objektorientierte Design (OOD). Im Zuge der ingenieurmäßigen (d.h. systematischen) und rechnergestützten Entwicklung sind viele dieser sog. CASE-Tools (Computer Aided Software Engineering) entstanden. Solch ein CASE-System sollte idealerweise den gesamten Softwarelebenszyklus unterstützten. Alle diese CASE-Werkzeuge basieren heute auf der grafischen Notationssprache UML (81), die bereits vorgestellt wurde.

Werkzeuge OOA/OOD (2)

Fast alle marktrelevanten Werkzeuge können aus UML-Modellen direkt Programmfunktionen in unterschiedlichen Programmiersprachen generieren. Einige sind zudem in der Lage, umgekehrt den Programmcode (Source) zu analysieren und durch Reverse Engineering entsprechende UML-Modelle zu generieren und anzupassen. Werden diese Verfahren systematisch genutzt, spricht man vom Round-Trip-Engineering.

Beispiele für CASE-Tools sind Innovator (MID), Rational Rose (IBM) und ArgoUML (Tigris.Org, Open Source).

Prototyp

... ist im Software Engineering nicht nur eine „Testversion“ eines Programms, sondern vielmehr eine in Teilbereichen lauffähige „Proof of Concept“ Implementierung eines Teils der abzubildenden Funktionalität, oder evtl. auch nur eine Oberfläche.

1. Evolutionärer Prototyp *zur Weiterentwicklung*,
2. Wegwerf-Prototyp *zur einmaligen Verwendung während der Systemspezifikation*.

Evolutionärer Prototyp

1. Ein „Fake“ bzw. eine ansatzweise lauffähig programmierte Oberfläche mit der Struktur und teilweise auch einzelnen Funktionen der gewünschten Funktionalität wird dem Auftraggeber zur Kommentierung vorgestellt.

2. Aufgrund des Feedbacks wird der Prototyp erweitert und verfeinert, bis

3. aus dem Prototyp das fertige Programm entsteht, das alle Funktionalitäten enthält.

Vorteil: Man erhält sehr schnell eine lauffähige Version mit „Wiedererkennungswert“ beim Aufraggeber, die kontinuierlich wächst.
Nachteil: Falls sich etwas an der Spezifikation ändert oder falls sich im Verlauf der Verfeinerung konzeptionelle Fehler herausstellen, ist das „Umbiegen“ des Prototypen mit hohem Aufwand verbunden.

Wegwerf-Prototyp

1. Wird nur in der Analysephase verwendet, um die Anforderungen zu fixieren,
2. dient auch dem Auffinden „vergessener“ Anforderungen im Leistungskatalog,
3. wird nach der Analyse **nicht** mehr weiterverwendet, d.h. in der Programmierphase wird neuer Code generiert.

Vorteil: Man erhält genauere Spezifikationen und nimmt weniger Fehler in die Produktimplementierung mit.
Nachteil: Fehlende Identifikation beim Auftraggeber/Anwender, da dieser den Prototyp als „erste Testversion“ auffassen könnte, was er nicht ist.

„Rapid Prototyping“

- Fokus liegt auf der **schnellen** Entwicklung eines lauffähigen Ansatzes,
- und weniger auf Performanz, Wartungsfreundlichkeit und Zuverlässigkeit.

Bei RP werden meist graphische Tools verwendet, die fertige Programmkomponenten in einer visuellen Oberfläche zusammensetzen, und automatisch Code generieren.
Vorteil: Vereinfachte, schnelle Programmentwicklung, geringere Entwicklungskosten.
Nachteil: Zuverlässigkeit und Stabilität ist stark von den verwendeten Komponenten abhängig, die exakten internen Abläufe (und mögliche Fehlerquellen) sind aber nicht einmal dem Programmierer bekannt.

Entwurfsmuster (1)

Wikipedia: Ein Entwurfsmuster (engl. design pattern) beschreibt eine bewährte Schablone für ein Entwurfsproblem.

Es stellt damit eine wiederverwendbare Vorlage zur Problemlösung dar. Entstanden ist der Ausdruck in der Architektur [NB: Häuser, nicht Software!], von wo er für die Softwareentwicklung übernommen wurde.

In den letzten Jahren hat der Ansatz der Entwurfsmuster auch zunehmendes Interesse im Bereich der Mensch-Computer-Interaktion gefunden.

Entwurfsmuster (2)

Strukturmuster

1. Klassenmuster
   - Adapter
2. Objektmuster
   - Adapter
   - Brücke
   - Kompositum
   - Dekorierer
   - Fassade
   - Fliegengewicht
   - Stellvertreter

Verhaltensmuster

1. Klassenmuster
   - Interpreter
   - Schablonenmethode
2. Objektmuster
   - Zuständigkeitskette
   - Kommando
   - Iterator
   - Vermittler
   - Memento
   - Beobachter
   - Zustand
   - Strategie
   - Besucher
   - Plugin

Implementierung am Beispiel Java

Um Entwurfsmuster, um die es in den folgenden Abschnitten geht, verstehen zu können, ist eine kurze Wiederholung (oder Einführung, falls diese Themen noch nicht behandelt wurden) einiger „Spezialitäten“ von Java als objektorientierter Programmiersprache sinnvoll.

- Vererbung (`extends`)
- abstrakte Klassen (`abstract`),
- virtuelle Methoden,
- Interfaces (`interface`, `implements`).

Vererbung

```
public class Tier {
 public String name;
}

public class Pinguin extends Tier {
 public String Gattung;
 public boolean hungrig;
}
```

Problem: Bei einer Objektvariablen wie `Tier tier = new Tier();` weiß man nie, welche Zusatzattribute das Tier noch haben könnte. Auch ist **Mehrfachvererbung**, also das Erben von Eigenschaften von **mehr als einer Basisklasse** in Java nicht möglich.

Abstrakte Klassen und Methoden

```
abstract class Tier {
 public String name;
 public void laufen() {...};
 public void hüpfen() {...};
 abstract void fliegen();
}
```

Von dieser Basisklasse können keine Objekte direkt erzeugt werden. Vielmehr kann sie nur Eigenschaften und Methoden mittels `extends Tier` vererben.

Merke: `abstract` -Klassen und -Methoden werden erst bei der Vererbung tatsächlich implementiert (`abstract`-Methoden müssen in der abgeleiteten Klasse überschrieben werden, sonst beschwert sich der Compiler). Abstrakte Klassen werden in UML-Diagrammen *kursiv* beschriftet.

Interfaces (1)

Nachteil einer `abstract class` ist die Tatsache, dass man nur von einer einzigen Basisklasse gleichzeitig mit `extends` ableiten kann. Mehrfachvererbung ist in Java nicht möglich.

Mit Hilfe von **Interfaces** können jedoch mehrere „Klassen von Methoden“ in einer Subklasse implementiert werden. Solche Interfaces enthalten nur abstrakte Methoden (ohne Implementierung) und Konstanten (`final`).

Interfaces (2)

```
interface network {
 int speed = 100;
 void connect(int ip1, int ip1);
 void disconnect();
}

public class Surfen extends Computer implements network {
 public void connect(int ip1, int ip2) { ... }
 public void disconnect() { ... }
}
```

Merke: Alle Attribute in einem `interface` sind automatisch `public`, `static` und konstant (`final`), alle Methoden automatisch `public` und virtuell (`abstract`).

Interfaces (3)

...ansonsten verhalten sich Interfaces genau wie Klassen, abgesehen davon, dass eine Kombination mit durch Komma getrennten Angaben von Interfaces hinter `implements` möglich ist. Sie können auch untereinander vererbt (`extends`) werden.

Auch bei Interfaces müssen alle virtuellen (`abstract`) Methoden in der implementierenden Klasse tatsächlich implementiert/überschrieben werden, sonst beschwert sich der Compiler!

Sichtbarkeit/Wirkungsbereiche (1)

- **private** Attribute und Methoden sind nur innerhalb eines von der Klasse instanzierten Objektes (`this`) sichtbar.
- **public** Attribute und Methoden sind nach außen sichtbar, und können z.B. als Objektmethode aufgerufen werden (`Objekt.MethodenAufruf()`). Dies ist, ohne weitere Angabe, die Standardeinstellung von Variablen in Klassen.
- **protected** Wie `private`, solche Attribute und Methoden sind aber auch in abgeleiteten Subklassen sichtbar.
- **abstract** Attribute und Methoden **müssen** in der instanzierenden Klasse implementiert/überschrieben werden, was z.B. in C einer reinen Deklaration (Prototyp einer Funktion oder Variablen, Signatur) entspricht.

Sichtbarkeit/Wirkungsbereiche (2)

- **static** Attribute und Methoden existieren unabhängig von einem instamzierten Objekt, und können direkt aus der Klasse heraus aufgerufen werden..
- **final** Attribute und Methoden können nach ihrer erstmaligen Instanzierung nicht mehr verändert werden.

Entwurfsmuster „Strategie“ (1)

Problemstellung: Von einer Basisklasse abgeleitete Subklassen sind von der tatsächlichen Implementierung her so auszulegen, dass einige Methoden der Basisklasse gar nicht, oder völlig anders als dort ggf. bereits implementiert, überschrieben werden müssten.
Zu verbessern („einfacher“ Ansatz):

```
public class Tier {
 String name;
 public void laufen();
 public void fliegen();
}

public void Pinguin extends Tier {
 public void laufen() { System.out.println("Ich laufe..."); }
 public void fliegen() { } // Geht nicht.
 ...
```

Entwurfsmuster „Strategie“ (2)

„Strategie“-Entwurf: 1. Abstrakte Basisklasse

```
public abstract class Tier {
 String name;
 LaufVerhalten laufVerhalten;
 FlugVerhalten flugVerhalten;
 public void tuLaufen() { laufVerhalten.laufen(); }
 public void tuFliegen() { flugVerhalten.fliegen(); }
}
```

Entwurfsmuster „Strategie“ (3)

„Strategie“-Entwurf: 2. Interfaces

```
public interface LaufVerhalten {
  void laufen();
}
public interface FlugVerhalten {
  void fliegen();
}
```

Entwurfsmuster „Strategie“ (4)

„Strategie“-Entwurf: 3. Interface-Klassen

```
public class Laeuft implements LaufVerhalten {
 public void laufen() { System.out.println("Ich laufe."); }
}
public class LaeuftNicht implements LaufVerhalten {
 public void laufen() { System.out.println("Ich kann nicht laufen."); }
}
public class Fliegt implements FlugVerhalten {
 public void fliegen() { System.out.println("Ich fliege."); }
}
public class FliegtNicht implements FlugVerhalten {
 public void fliegen() { System.out.println("Ich kann nicht fliegen."); }
}
public class FliegtUnterWasser implements FlugVerhalten {
 public void fliegen() { System.out.println("Ich fliege unter Wasser."); }
}
```

Entwurfsmuster „Strategie“ (5)

„Strategie“-Entwurf: 5. Abgeleitete Klassen

```
public class Fisch extends Tier {
 public Fisch() {
  name = new String("Fisch");
  flugVerhalten = new FliegtNicht();
  laufVerhalten = new LaeuftNicht();
 }
}
public class Pinguin extends Tier {
 public Pinguin() {
  name = new String("Pinguin");
  flugVerhalten = new FliegtUnterWasser();
  laufVerhalten = new Laeuft();
 }
}
```

Entwurfsmuster „Strategie“ (6)

„Strategie“-Entwurf: 6. Test

```
public class Main {
 public static void main(String[] args) {
  Tier pinguin = new Pinguin();
  Tier fisch = new Fisch();
  pinguin.tuFliegen();
  pinguin.tuLaufen();
  fisch.tuFliegen();
  fisch.tuLaufen();
 }
}
```

Fazit Entwurfsmuster „Strategie“

Das *Strategy*-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen, variieren zu lassen.

S.a. [3].

Wann „Strategie“-Entwurfsmuster?

Die Verwendung von „Strategien“ bietet sich an, wenn

- **viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden**,
- unterschiedliche, **austauschbare Varianten** eines Algorithmus benötigt werden,
- Daten innerhalb eines Algorithmus vor Klienten verborgen werden sollen oder verschiedene Verhaltensweisen innerhalb einer Klasse fest integriert (meist über Mehrfachverzweigungen) sind und die verwendeten Algorithmen wiederverwendet werden sollen oder
- die Klasse einfach **flexibler einsetzbar** gestaltet werden soll.

Entwurfsmuster „Beobachter“ (1)

Problemstellung: Ein *Subjekt* liefert nur zu gewissen Zeitpunkten (regelmäßig oder unregelmäßig) Daten (Zustandsänderungen). Diese sollen von einem oder mehreren Objekten ausgewertet bzw. dargestellt werden.
Zu verbessern („busy wait“ Ansatz):

```
public class Messung {
 public static void main(String[] args) {
  int alter_messwert = Erfassung.aktuell();
  for(;;) { // Endlosschleife
   int neuer_messwert = Erfassung.aktuell();
   if ( neuer_messwert != alter_messwert )
    System.out.println("Neuer Messwert: "
                       + neuer_messwert);
  }
 }
}
```

Nachteile: Ständige Messungen, CPU-/Resourcenverschwendung.

Entwurfsmuster „Beobachter“ (2)

In JAVA existiert in der `util`-Package eine fertige Implementierung des Observer-Musters in Form einer Basisklasse `Observable` und eines Interface `Observer`.

`Observable`-Objekte können mittels `setChanged()` und `notifyObservers()` Kontakt zu den `Observern` aufnehmen, woraufhin die Virtual Machine die Methode `update()` in den `Observer`-Objekten aufruft, die sich zuvor beim zu beobachtenden `Observable` registriert hatten.

Es handelt sich also um eine Art „Abonnement“ mit ereignisgesteuerten Methoden.

Entwurfsmuster „Beobachter“ (3)

Zunächst das Subjekt, das die Zustandsänderungen produziert:

```
import java.util.Observable;
import java.util.Observer;

public class Messung extends Observable {

 private float messwert = 0.0F; // Initialwert

 public void messung() { // Neuen Messwert holen
  float neu = Eingabe.readFloat(); // Messung simulieren
  if ( neu != messwert ) {
   messwert = neu;
   setChanged(); // Aus Observable
  }
  notifyObservers();
 }

 public float getMesswert() { return messwert; }

}
```

Entwurfsmuster „Beobachter“ (4)

Nun der Beobachter:

```
import java.util.Observable;
import java.util.Observer;

public class Beobachter implements Observer {
 Observable observable;

 public Beobachter(Observable observable) {
  this.observable = observable;
  observable.addObserver(this);
 }

 public void update(Observable observable, Object arg) {
  System.out.println(((Messung)observable).getMesswert());
 }
}
```

Entwurfsmuster „Beobachter“ (5)

„Abonnieren“ von Nachrichten:

```
Messung m = new Messung(); // extends Observable
Beobachter a = new Beobachter(m); // m.addObserver(a);
Beobachter b = new Beobachter(m); // m.addObserver(b);
```

Sobald `m` nach `setChanged()` ein `notifyObservers()` aufruft, wird die Methode `update()` in `a` und `b` aufgerufen.

Fazit Entwurfsmuster „Beobachter“

Das *Beobachter*-Muster ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte. Dieses Entwurfsmuster ist auch unter dem Namen *publish-subscribe* („veröffentlichen und abonnieren“) bekannt.

Wann „Beobachter“-Entwurfsmuster?

Die Verwendung von „Beobachtern“ bietet sich an, wenn

- **nur zu bestimmten Zeitpunkten eine Aktion erforderlich ist**,
- Aktionen **ausgelöst durch Ereignisse** initiiert werden sollen,
- Objekte **automatisch aktualisiert** werden sollen,
- eine **lose Kopplung** zwischen Objekten sinnvoll ist, und die Objekte keine Kenntnis von der Struktur ihrer „Gesprächspartner“ haben müssen.

Entwurfsmuster „Dekorierer“ (1)

Problemstellung: Es sollen Objekte gebildet werden, die gegenüber der Basisklasse um *mehrere*, *kaskadierbare* Eigenschaften *erweitert* sind. Die Zusatzeigenschaften können sich auf Instanzen, die von der Basisklasse geerbt werden, auswirken.

Zu verbessernder Ansatz:

```
public class Tee { ... }
public class TeeMitMilch extends Tee { ... }
public class TeeMitMilchUndZucker extends TeeMitMilch { ... }
public class TeeMitZucker extends Tee { ... }
```

Entwurfsmuster „Dekorierer“ (2)

Anderer Ansatz: Sammlung von Attributen in einer Klasse.

```
public class Tee {
 boolean Milch;
 boolean Zucker;
 boolean Zitrone;
 boolean Soja;
 ...
}
```

Entwurfsmuster „Dekorierer“ (3)

Eine abstrakte Basisklasse:

```
public abstract class Getraenk {
 String beschreibung = "Getränk";

 public String getBeschreibung() {
  return beschreibung;
 }

 public abstract float preis();
}
```

Entwurfsmuster „Dekorierer“ (4)

Eine Kaffee- und eine Tee-Klasse:

```
public class Kaffee extends Getraenk {
 public Kaffee() {
  beschreibung = "Kaffee";
 }

 public float preis() {
  return 1.0F;
 }
}
```

```
public class Tee extends Getraenk {
 public Tee() {
  beschreibung = "Tee";
 }

 public float preis() {
  return 0.90F;
 }
}
```

Entwurfsmuster „Dekorierer“ (5)

Die Dekorierer-Basisklasse:

```
public abstract class Option extends Getraenk {
 public abstract String getBeschreibung();
}
```

Dadurch, dass `Option` auf `extends Getraenk` basiert, ist es möglich, ein Objekt dieser Klasse einer `Getraenk`-Referenz zuzuweisen, was wir in `Main.java` sehen werden.

Entwurfsmuster „Dekorierer“ (6)

Die Zutaten (1):

```
public class Milch extends Option {
 Getraenk getraenk;

 public Milch(Getraenk getraenk) {
  this.getraenk = getraenk;
 }

 public String getBeschreibung() {
  return getraenk.getBeschreibung() + ", Milch";
 }

 public float preis() {
  return 0.20F + getraenk.preis();
 }
}
```

Entwurfsmuster „Dekorierer“ (7)

Die Zutaten (2):

```
public class Zucker extends Option {
 Getraenk getraenk;

 public Zucker(Getraenk getraenk) {
  this.getraenk = getraenk;
 }

 public String getBeschreibung() {
  return getraenk.getBeschreibung() + ", Zucker";
 }

 public float preis() {
  return 0.25F + getraenk.preis();
 }
}
```

Entwurfsmuster „Dekorierer“ (8)

Und servieren:

```
public class Main {
 public static void main(String[] args) {
  Getraenk getraenk1 = new Tee();
  getraenk1 = new Milch(getraenk1); // Tee mit Milch dekorieren

  System.out.println(getraenk1.getBeschreibung() + " kostet " +
                     getraenk1.preis() + " Euro.");

  Getraenk getraenk2 = new Kaffee();
  getraenk2 = new Zucker(getraenk2); // Kaffee mit Zucker dekorieren
  getraenk2 = new Zucker(getraenk2); // nochmal Zucker drauf
  getraenk2 = new Milch(getraenk2); // und mit Milch dekorieren.

  System.out.println(getraenk2.getBeschreibung() + " kostet " +
                     getraenk2.preis() + " Euro.");
 }
}
```

Fazit Entwurfsmuster „Dekorierer“

Das *Dekorierer*-Muster fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zwecke der Erweiterung der Funktionalität.

S.a. [3].

Wann „Dekorierer“-Entwurfsmuster?

Die Verwendung von „Dekorierern“ bietet sich an, wenn

- **mehrere Zusatzeigenschaften** einem Objekt zur Laufzeit **dynamisch hinzugefügt** werden sollen,
- **die zu dekorierende Klasse nicht unbedingt festgelegt** ist, sondern nur *deren Schnittstelle*,
- die Erweiterungsoptionen auch dynamisch austauschbar sein sollen, und
- lange und **unübersichtliche Vererbungshierarchien vermieden** werden sollen.

Entwurfsmuster „Fabrik“ (1)

Problemstellung: Es sollen Objekte gebildet werden, deren konkrete Klassen sich aber erst zur Laufzeit entscheiden. Der direkte Aufruf von `new` soll vermieden und sogar verhindert werden.

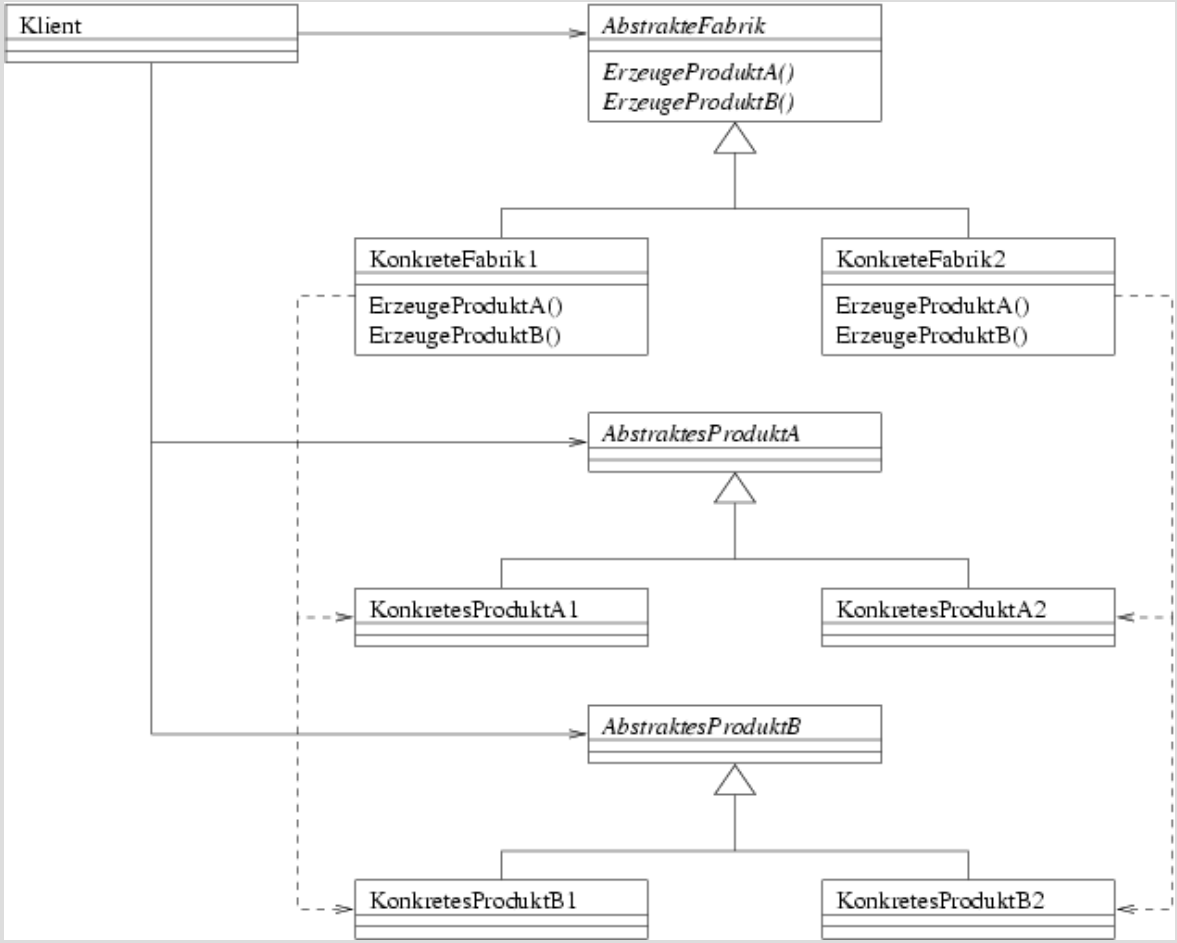
Zu verbessernder Ansatz:

```
public class WienerSchnitzel extends Schnitzel {
  ...
}

...

Schnitzel s = new WienerSchnitzel();
```

Entwurfsmuster „Fabrik“ (2)



Entwurfsmuster „Fabrik“ (3)

1. Zunächst wird eine (oder mehrere) abstrakte Klasse(n) definiert, deren Konkretisierung durchaus unterschiedlich sein kann:

```
public abstract class Spielbrett {
 int getFelder();
 ...
}

public abstract class Spielfigur {
 int getAnzahl();
 ...
}
```

Je nach Spielart (z.B. Mühle, Dame, Schach, ...) sind die konkreten Klassen sowie die davon gebildeten Spielbretter und Figuren völlig unterschiedlich!

Entwurfsmuster „Fabrik“ (4)

2. Nun wird eine abstrakte Klasse definiert, die immer noch variable Objekte erzeugt.

```
public abstract class SpieleFabrik {
 protected abstract Spielbrett erzeugeSpielbrett();
 protected abstract Spielfigur erzeugeSpielfigur(int nummer);
}
```

Die abstrakte SpieleFabrik kann zunächst selbst keine Objekte erzeugen, aber als Basisklasse für konkrete Fabriken dienen.

Entwurfsmuster „Fabrik“ (5)

3. Jetzt geht es in die konkrete Implementation.

```
public class SchachFabrik extends SpieleFabrik {
 public Spielbrett erzeugeSpielbrett() { ... };
 public Spielfigur erzeugeSpielfigur(int nummer) { ... };
}

public class HalmaFabrik extends SpieleFabrik {
 public Spielbrett erzeugeSpielbrett() { ... };
 public Spielfigur erzeugeSpielfigur(int nummer) { ... };
}
```

Wo sinnvoll, können auch für die einzelnen Figuren oder Spielbretter eigene Unterklassen mit entsprechenden Erzeugungsmethoden generiert werden. Diese werden aber normalerweise ausschließlich innerhalb der Fabriken verwendet.

Entwurfsmuster „Fabrik“ (6)

4. Aufruf im Programm

```
public class Main {
 public static void main(String[] args) {
   SpieleFabrik schachFabrik = new Schachfabrik();
   SpieleFabrik halmaFabrik  = new HalmaFabrik();
   ...
   Spielbrett schachbrett = schachFabrik.erzeugeSpielbrett();
   Spielfigur schachbauer = schachFabrik.erzeugeSpielfigur(0)
   ...
   Spielbrett halmabrett = halmaFabrik.erzeugeSpielbrett();
   Spielfigur halmafigur = halmaFabrik.erzeugeSpielfigur(1);
   ...
 }
}
```

Fazit Entwurfsmuster „Fabrik“

Das *Abstrakte Fabrik*-Muster stellt eine Schnittstelle zur Erzeugung einer ganzen **Familie von Objekten** bereit. Die konkreten Klassen der zu erzeugenden Objekte werden dabei nicht festgelegt.

Vorteil: Der Aufruf bei der Erzeugung der Objeke ist immer gleich (gleicher Methodenname bei unterschiedlicher Fabrik). Die spezifischen Erzeugungsmethoden der Objeke sind übersichtlich gekapselt.

Außer zur Erzeugung der konkreten Fabriken zu Beginn, wird `new` nicht mehr im eigentlichen Programm verwendet verwendet.

Wann „Fabrik“-Entwurfsmuster?

Die Verwendung von „Fabriken“ bietet sich an, wenn

- ein System **unabhängig von der Art der Erzeugung seiner Produkte** arbeiten soll,
- ein System mit einer oder **mehreren Produktfamilien** konfiguriert werden soll,
- eine Gruppe von Produkten erzeugt und gemeinsam genutzt werden soll oder
- wenn in einer Klassenbibliothek die Schnittstellen von Produkten ohne deren Implementation bereitgestellt werden sollen.

Entwurfsmuster „Einzelstück“ (1)

Problemstellung: Es soll von einer Klasse genau ein Objekt gebildet werden, da mehrere Objekte der gleichen Klasse die korrekte Funktion des Programms in Frage stellen würde, oder einfach implementationstechnisch nicht sinnvoll wären.

Das soll vermieden werden:

```
public class MasterControlProcess {
 public void steuereCNC(int befehlsnummer);
 ...
}
...
MasterControlProcess mcp1 = new MasterControlProcess();
MasterControlProcess mcp2 = new MasterControlProcess();
mcp1.steuereCNC(startRoboterArmNachRechtsBisSensor);
mcp2.steuereCNC(startRoboterArmNachLinksBisSensor);
// Maschine kaputt.
```

Entwurfsmuster „Einzelstück“ (2)

Die Erstellung eines Singleton ist eigentlich recht einfach: Der Konstruktor darf von außen nie aufgerufen werden, und von der Klasse selbst nur genau einmal. Beispiel 1 („Lazy creation“, wird erst bei Bedarf erzeugt):

```
public final class Singleton {
 private static Singleton instance;
 private Singleton() {} /* Konstruktor ist privat. */
 /* Statische Methode "getInstance()" liefert die einzige
    Instanz der Klasse zurück, synchronisiert und somit
    thread-sicher. */
 public synchronized static Singleton getInstance() {
  if (instance == null) { /* erster Aufruf */
   instance = new Singleton();
  }
  return instance;
 }
}
```

Entwurfsmuster „Einzelstück“ (3)

Beispiel 2 („Eager creation“, existiert von Anfang an):

```
public final class Singleton {
 /* Privates Klassenattribut, einzige Instanz der Klasse
    wird sofort erzeugt. */
 private static final Singleton instance = new Singleton();
 private Singleton() {} /* Konstruktor ist privat. */
 /* Statische Methode "getInstance()" liefert die einzige
    Instanz der Klasse zurück. */
 public static Singleton getInstance() {
  return instance;
 }
}
```

Entwurfsmuster „Einzelstück“ (4)

Aufruf für beide Beispiele:

```
...
// Erstmalige Erzeugung in Beispiel 1, bzw.
// Referenz auf bereits existierende Instanz
// in Beispiel 2

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

// s2 ist nun lediglich eine Referenz auf das gleiche
// Objekt, auf das s1 schon zeigt!
```

Fazit Entwurfsmuster „Einzelstück“

Das *Singleton*-Muster stellt sicher, dass zu einer Klasse nur genau ein Objekt erzeugt werden kann und ermöglicht einen globalen Zugriff auf dieses Objekt.

Wann „Einzelstück“-Entwurfsmuster?

Die Verwendung von „Singleton“ bietet sich an, wenn

- nur ein Objekt zu einer Klasse existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird (Beispiel: Druckerspooler, Protokollierer, Gerätetreiber) oder
- wenn das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

Bei diesem Muster sieht man besonders deutlich, dass die Entwurfsmuster sich fast immer auf Spezialfälle beziehen, die man im Programm geschickt abbilden möchte, so dass Fehler durch den Anwender der erzeugten Programmschnittstelle so gut wie ausgeschlossen sind.

Entwurfsmuster „Fliegengewicht“ (1)

Problemstellung: In einem Programm werden sehr viele Instanzen einer Klasse verwendet, wobei aber eine Unterscheidung durch individuelle Attribute nicht notwendig ist. Jede Instanz kostet jedoch Speicherplatz und Rechenzeit zur Verwaltung.

Zu optimieren:

```
public class Baum {
 private int x_koord, y_koord;
 public Baum(int x_koord, int y_coord) { ... }
 public void anzeigen(boolean ja_nein) { ... }
}
...
for(int i=0; i<1000; i++) {
 Wald.add(new Baum(i, i/10%100));
}
```

Entwurfsmuster „Fliegengewicht“ (2)

Der Wald im vorigen Beispiel besteht aus vielen gleichartigen Baum-Objeken (nur die Koordinaten sind unterschiedlich).

1. Es wird zunächst für die nicht unterscheidbaren Objekte eine „zustandslose Klasse“ erzeugt, die i.d.R. nur allgemeine Methoden zur Verfügung stellt, welche auf jedes der Objekte anwendbar sind.

```
public class Baum {
 // Zustandsfreies Einzelobjekt.
 public void zeichne(int x, int y) {...}
}
```

Entwurfsmuster „Fliegengewicht“ (3)

2. Es wird nur EIN Baum-Objekt erzeugt, und in einem Array werden die Koordinaten gespeichert, an denen dieses Objekt gefunden wird.

```
public static void main(String[] args) {
 // einziger, anonymer Baum
 Baum baumObjekt = new Baum();

 // Array mit Koordinaten
 int[2][] Bäume;

 // gewünschte Koordinaten in "Bäume" eintragen
 ...
```

Entwurfsmuster „Fliegengewicht“ (4)

3. Methoden aus `Baum` nach Bedarf anwenden.

```
for(int i=0;...)
 baumObjekt.zeichne(Bäume[0][i], Bäume[1][i]);
```

Merke: Die komplette Logik und Statusinformation steckt hier nicht im Objekt, sondern in der Umgebung. (Dies wiederspricht eigentlich der „Kapselung“, die man sonst im objektorientierten Programmierparadigma gerne erreichen möchte.)

Fazit Entwurfsmuster „Fliegengewicht“

Das *Flyweight*-Muster stellt von einer Instanz einer Klasse viele „virtuelle Instanzen“ bereit, ohne Kopien zu erzeugen.

Wann „Fliegengewicht“-Entwurfsmuster?

Die Verwendung von „Flyweight“ bietet sich an, wenn

- wenn aus Performancegründen die Anzahl von Objekten zur Laufzeit gering gehalten, und dadurch Speicher und Rechenzeit (für die Objektverwaltung) eingespart werden soll,
- wenn der Zustand von vielen „virtuellen“ Objekten an einem Ort zentral gespeichert werden soll.

Nachteil dieses Musters ist, dass eine Unterscheidung der Objekte aufgrund ihres Zustandes nicht möglich ist, inividuelle Attribute können also nicht (außer in der Umgebung(hier: im Array `Bäume`)) gespeichert und abgerufen werden.

Entwurfsmuster „Template Method“ (1)*

Problemstellung: In einem Programm tauchen (wie beim Dekorierer-Muster) unterschiedliche Objekte mit durchaus *ähnlichen*, aber jeweils *unterschiedlich zu implementierenden* Algorithmen auf. Ein genereller Ablaufplan ist jedoch für alle Objekte gleich.

Zu optimieren:

```
public class HeissesGetraenk {
 public void heissesWasser(); // Alle
 public void aufbruehen(); // Nur Kaffee
 public void ziehen_lassen(); // Nur Tee
 ...
```

Entwurfsmuster „Template Method“ (2)*

1. Finden der „immer gleichen“ Abläufe.

```
public abstract class KaffeeOderTee {
 final void zubereitungsRezept() {
  wasserKochen();
  aufGiessen();
  inTasseEinfuellen();
  zutatenHinzufuegen();
 }
// ... to be continued
```

Entwurfsmuster „Template Method“ (3)*

2. Implementieren der für alle Unterklassen gleichen Methoden in der Basisklasse, und deklarieren der unterschiedlichen Methoden als `abstract`.

```
public abstract class KaffeeOderTee {
 ...
 void wasserKochen() {
  // Zisch...
 }
 void inTasseEinfuellen() {
  // Zosch...
 }
 abstract void aufGiessen();
 abstract void zutatenHinzufuegen();
}
```

Entwurfsmuster „Template Method“ (4)*

3. Schreiben der Unterklassen, die nun lediglich die abstrakten Methoden implementieren müssen.

```
public class Tee extends KaffeeOderTee {
 void aufGießen(){
  // Teebeutel rein und Wasser drüber
 }
 void zutatenHinzufuegen() {
  // Zucker und Zitrone oder Milch
 }
 public Tee() { // Konstruktor
  zubereitungsRezept();
 }
}
```

```
public class Kaffee extends KaffeeOderTee {
 void aufGießen(){
  // Kaffee mahlen, heißes Wasser
 }
 void zutatenHinzufuegen() {
  // Zucker und/oder Milch
 }
 public Kaffee() { // Konstruktor
  zubereitungsRezept();
 }
}
```

Fazit Entwurfsmuster „Template Method“*

Das *Template Method*-Muster definiert zunächst grob die Schritte eines Algorithmus und überlässt es Unterklassen, die konkreten Implementierungen für einen oder mehrere dieser Schritte zur Verfügung zu stellen. Einige Schritte können somit in der Basisklasse für alle Erweiterungen implementiert sein.
Template Method wird im Deutschen auch oft als „Schablone“ bezeichnet.

Wann „Template Method“?*

Beim der Verwendung von „Template Method“ wird in einer abstrakten Klasse ein *Skelett* eines Algorithmus in einer Operation definiert. Die konkrete Ausformung der einzelnen Schritte des Algorithmus wird an die Unterklassen delegiert. Dies ermöglicht, einzelne Schritte eines Algorithmus zu modifizieren oder zu überschreiben, ohne dass die grundlegende Struktur des Algorithmus verändert wird. Es bietet sich an,

- wenn ein immer gleicher Ablauf sich im Detail verändern lassen soll,
- wenn Unterklassen lediglich Abweichungen von der „Schablone“ implementieren sollen.

Fazit „Entwurfsmuster“

- Entwurfsmuster sind KEINE „Muster (Beispiele) für konkrete Implementation“.
- Entwurfsmuster sollen Strategien und Ansätze zum eleganten Lösen von Engineering-Aufgaben zur Verfügung stellen.
- Entwurfsmuster sollen nie einfach „abgetippt“ werden, sondern geschickt verknüpft und mit eigenen, auf das Problem zugeschnittenen Lösungsansätzen zur Erstellung von „optimalem“ Code mitverwendet werden, wenn sie passen.

Fehler

- Syntaktische Fehler, findet der Compiler,
- Semantische Fehler, muss man selbst finden (der Compiler kann allenfalls Warnungen/Tipps geben),
- Systematische Fehler (finden durch Verifikation und Validierung) (18),
- Pragmatische Fehler (das Programm hätte vielleicht effektiver-/einfacher sein können).

Bestimmte Fehler werden oft als „bedeutungslos“ angesehen, wenn sie den beabsichtigten Einsatzzweck nicht beeinträchtigen.

Fehlersuche, Beispiel (1)

```
import java.math.*;

public class Fakultaet {
 static int fakultaet(int i) {
  if(i <= 1) return 1;
  return i * fakultaet(i-1);
 }
 public static void main(String[] args) {
  int eingabe = Integer.valueOf(args[0]).intValue();
  System.out.print(eingabe + "! = ");
  System.out.println(fakultaet(eingabe));
 }
}
```

Ein relativ überschaubares Programm, dennoch voller Fehlerquellen...

Fehlersuche, Beispiel (2)

Eine „mächtigere“ Implementierung von `fakultaet(x)` mit höherem Ausgabewertebereich durch `BigInteger`:

```
static BigInteger fakultaet(int i) {
 if (i <= 1) return BigInteger.ONE;
 return BigInteger.valueOf(i).multiply(fakultaet(i-1));
}
```

Fehlersuche, Beispiel (3)

Probleme:

- Definitionsbereich der EINGABEWERTE,
- Wertebereich der AUSGABEWERTE,
- Spezifikation des ALGORITHMUS (z.B. „Was passiert, wenn kein Eingebewert in `args[0]` vorliegt?“, oder „Sind ausreichend Resourcen für den Ablauf des Algorithmus vorhanden?“).

Software-Rollout und Tests

Nach der Implementationsphase, die je nach verwendetem Entwicklungmodell (wir erinnern uns: Wasserfall, V-Modell, Spiral-Modell, Extreme Programming, ...) mehr oder minder in Zusammenarbeit mit den Auftraggebern und Anwendern synchronisiert ist, folgt die Integration oder Migration in die Arbeitsumgebung.

Spätestens jetzt sollten evtl. in der Planungsphase aufgetretene, konzeptionelle Fehler behoben sein!

Dennoch müssen Fehler, die im „normalen Betrieb“ auftreten können, rechtzeitig erkannt werden.

Arten von Tests (Teilw. Wdh.)

- Modultests (Low-Level-Test)
- Integrationstests (Low-Level-Test)
- Systemtests (High-Level-Test)
- Akzeptanztests (High-Level-Test)

Je kritischer/riskanter der Einsatz der Software ist, desto mehr Raum müssen Tests im Software Engineering Prozess spielen.

Modultest (1)

Nach Wikipedia.DE (auch Komponententest oder engl. unit test), ist Teil eines Softwareprozesses (z.B. nach dem Vorgehensmodell des Extreme Programming).

Er dient zur Verifikation der Korrektheit von Modulen einer Software, z.B. von einzelnen Klassen bzw. Klassenmethoden. Nach jeder Änderung sollte durch Ablaufenlassen aller Testfälle nach Programmfehlern gesucht werden. Dieses Wiederholen der Tests nach Änderungen wird als „Regressionstest“ bezeichnet. Bei der testgetriebenen Entwicklung, auch „TestFirst-Programmieren“ genannt, werden die Modultests parallel zum eigentlichen Quelltext erstellt und gepflegt. Dies ermöglicht bei automatisierten, reproduzierbaren Modultests die Auswirkungen von Änderungen sofort nachzuvollziehen. Der Programmierer entdeckt dadurch leichter ungewollte Nebeneffekte oder Fehler durch seine Änderung.

Modultest (2)

Ein Komponententest ist ein ausführbares Codefragment, das das sichtbare Verhalten einer Komponente (z.B. einer Klasse) verifiziert und dem Programmierer eine unmittelbare Rückmeldung darüber gibt, ob die Komponente das geforderte Verhalten aufweist oder nicht.

Komponententests sind ein wesentlicher Bestandteil der Qualitätssicherung in der Softwareentwicklung.

Modultest (3)

Gefahr: Modultests werden auch im Automotive-Bereich an programmierbaren Steuereinheiten verwendet. Damit wird die Steuereinheit verifiziert (= ihre Übereinstimmung mit der Absicht des Entwicklers geprüft). Hier haben die Modultests auch rechtliche Bedeutung innerhalb des Vertragsdokumentes. Falls eine programmierbare Steuerung versagt kann es zu Personenschäden kommen. Bei einem solchen Test wird die Durchführung einschließlich aufgetretener Fehler protokollartig festgehalten. Bei Tests im Automotive-Bereich stehen Variablen physikalischer Werte und damit Grenzwerte im Vordergrund. So muss z.B. geprüft werden, ob das Ergebnis einer Addition von Ganzzahlen sich in jedem Fall innerhalb des Wertebereiches des Ganzzahl-Datentyps befindet.

Modultest (4)

Beispiele für Modultest-Frameworks:

- JUnit für Java, von Erich Gamma und Kent Beck.
- Cactus aus dem Apache Jakarta-Projekt für Java-Servlets.

Integrationstest (1)

Wikipedia: Der Begriff Integrationstest bezeichnet in der Softwareentwicklung eine **aufeinander abgestimmte Reihe von Einzeltests**, die dazu dienen, verschiedene **voneinander abhängige Komponenten** eines komplexen Systems **im Zusammenspiel miteinander zu testen**.

Die dabei erstmals im gemeinsamen Kontext zu testenden Komponenten sollten zuvor jeweils einen Unit-Test erfolgreich bestanden haben, und für sich isoliert fehlerfrei funktionsfähig sein.

Integrationstest (2)

Systematik:

Für jede Abhängigkeit zwischen zwei Komponenten eines Systems wird ein Testszenario definiert, durch welches nachzuweisen ist, dass nach der Zusammenführung sowohl beide Komponenten für sich wie auch der Datenaustausch über die **gemeinsame(n) Schnittstelle(n)** spezifikationsgemäss (!) funktionieren.

Schnittstellentests dienen zur Überprüfung der Daten, die zwischen den Komponenten ausgetauscht werden.

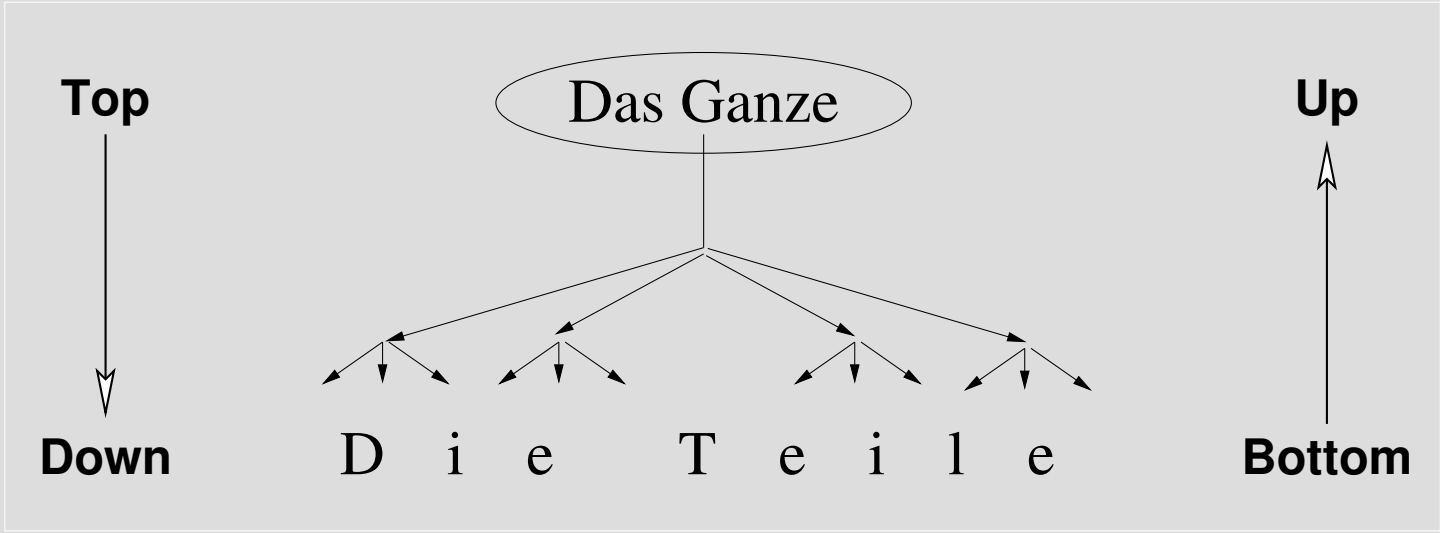
Integrationstest (3)

Umfang:

Da der zeitliche Aufwand für Integrationstests mit wachsender Komponentenanzahl überproportional ansteigt, ist es üblich, Integrationstests für einzelne, abgegrenzte Subsysteme durchzuführen und diese dann im weiteren Verlauf als eine Komponente zu betrachten (Bottom-Up-Methode).

Bei dieser Methode enden die Integrationstests erst mit den erfolgreichen Testläufen in einer mit dem späteren Produktivsystem identischen Testumgebung.

Einwurf: „Bottom-Up“ und „Top-Down“



Systemtest (1)

Der Systemtest kann definiert werden als:

1. Testphase, bei der das gesamte System gegen die Spezifikation getestet wird.
2. Test eines Gesamtsystems gegen seine Anforderungen.

Der Systemtest wird in den funktionalen und den nicht funktionalen Systemtest eingeteilt. Er wird von der entwickelnden Organisation (d.h. nicht vom Anwender selbst) durchgeführt.

Systemtest (2)

Der **funktionale Systemtest** dient zur Überprüfung eines Systems bezüglich dessen funktionalen Qualitätsmerkmalen **Korrektheit** und **Vollständigkeit**.
Der **nicht funktionale Systemtest** beinhaltet Merkmale, wie z.B. die **Sicherheit**, die **Benutzbarkeit**, die **Interoperabilität**, die Prüfung der **Dokumentation** oder die **Zuverlässigkeit eines Systems**.

Systemtest (3)

Um ein System und dessen Komponenten zu testen, bedient man sich oft des **White-Box**- oder **Black-Box**-Tests, zweier sich gegenseitig ergänzender Methoden.

White-Box-Test: Die (beabsichtigte) Funktionsweise eines Systems oder Moduls ist bekannt, es wird also direkt am Code geprüft (s.a. Open Source Software). ☞Anweisungsüberdeckung, Kantenüberdeckung, Bedingungsüberdeckung, Pfadüberdeckung

Black-Box-Test: Die konkrete Implementierung eines Systems oder Moduls ist **nicht bekannt**, es können nur Eingabewerte und Ausgabewerte beobachtet werden. Dieser Test dient dazu, die Einhaltung der Spezifikation für bestimmte Werte zu überprüfen. Ob der Code fehlerfrei programmiert oder insgesamt korrekt ist, lässt sich nicht feststellen (s.a. proprietäre Software).

Akzeptanztest

In der Software-Branche bezeichnet der Begriff Akzeptanztest einen Test der gelieferten Software durch den Anwender, der meist gleichzeitig Kunde ist (Funktionaler Akzeptanztest), sowie durch den Hersteller (Produktions-Akzeptanztest).

Oft sind Akzeptanztests Voraussetzung für die Rechnungsstellung. Bei einem solchen Test wird das Blackbox-Verfahren angewendet, d.h. der Kunde betrachtet nicht den Code der Software, sondern nur das Verhalten der Software bei spezifizierten Handlungen (Eingaben des Benutzers, Grenzwerte bei der Datenerfassung, etc.). Getestet wird anhand eines Prüfprotokolls, ob die in dem Pflichtenheft festgelegten Anforderungen erfüllt werden.

Begleitende Prozesse

- mit dem Projektmanagement hatten wir uns schon unter (11) beschäftigt.
- Incident Management (Vorfallsmanagement)
- Change Management (Veränderungsmanagement)
- Release Management
- Konfigurationsmanagement
- Application Management
- Qualitätsmanagement
- Software-Ergonomie
- Softwaremetrik
- Versionsverwaltung

Begleitende Prozesse - Dokumentation

- Systemdokumentation (Entwickler/Betreiber)
- Gebrauchsanleitung (Benutzer)
- Geschäftsprozess (Abstrakt, Verständnis)
- Verfahrensdokumentation (rechtl., Gesetze)

Dokumentation - Beispiele

- HOWTOs - kurze Bedienungsanleitungen für Programme oder Vorgehensweisen (Algorithmen)
- Man-Pages (Unix) - Nachschlagewerke für Kommandos
- RFC - Request for Comment, Definitionen von Internet-Standards in Textform
- APIs - Schnittstellen-Definitionen für Programmierer
- Handbücher - Für Benutzer
- Spezifikationen - Systemdokumentation für Betreiber und Entwickler (auch Hardware)

Versions- und Revisionsverwaltung (1)

- Revision Control System - RCS
  `ci`, `co`
- Concurrent Versions System - CVS
  `cvs update`, `cvs commit`
- Subversion - SVN
  `svn update`, `svn commit` (Wie CVS)
- GIT
  `git`

Versions- und Revisionsverwaltung (2)

`diff -u Datei1 Datei2` vergleicht 2 Dateien und erzeugt eine „Differenz-Ausgabe“.

`patch Datei1 < patch.diff` wendet eine zuvor generierte Differenz-Datei auf `Datei1` an, und verändert diese.

Versions- und Revisionsverwaltung (2)

Vorgehen aus Entwicklersicht:

1. Repository auf eigenen Rechner kopieren (CLONE, CHECK-OUT).
   (a) GIT (1): `git clone repo-adresse` (Alle Entwicklungszweige)
   (b) GIT (2): `git checkout entwicklungszweig`
   (c) SVN: `svn checkout repo-adresse`
2. Arbeiten, Änderungen am Source vornehmen...

Versions- und Revisionsverwaltung (3)

3. Aktuellen Status (außer eigene Änderungen) synchronisieren:
   (a) GIT: `git pull`
   (b) SVN: `svn update`
4. Eigene Änderungen einpflegen (MERGE, COMMIT)
   (a) GIT (1): `git merge`
   (b) GIT (2): `git commit`
   (c) GIT (3): `git push`
   (d) SVN: `svn commit`

Zitierte Literatur

Literatur

[1] DE.Wikipedia.Org

[2] *Software Engineering*, Ian Sommerville, Addison-Wesley, ISBN 3-8273-7001-9

[3] *Entwurfsmuster von Kopf bis Fuß*, Freeman et al., O'Reilly Verlag, ISBN-13 978-3-89721-421-7