



Software Engineering

2. Semester
Wirtschaftsinformatik

Prof. Dr. Carsten Dorrhauer



Voraussetzungen und Lernziele

Voraussetzungen

- Sie haben erste Erfahrungen mit der Programmierung in einer objektorientierten Sprache.

Lernziele

- Sie kennen Gegenstand und Ziele des Software Engineering.
- Sie können wichtige Vorgehensmodelle beschreiben und beurteilen.
- Sie verstehen die Aufgaben, die bei der Entwicklung einer umfangreichen Software anfallen und mögliche Lösungen dafür.
- Sie verstehen einige wichtige aktuelle Softwaretechnologien.



Gliederung

1. Grundlagen
2. Vorgehensmodelle für Softwareentwicklungsprojekte
3. Spezifikation von Software
4. Entwurf von Software
5. Entwicklung von Software
6. Validierung von Software
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architekturen
10. Entwurfsmuster
11. Aspektorientierte Softwareentwicklung



Literaturempfehlungen

Sommerville, Ian: Software Engineering, 8. Aufl., München 2007

Hindel u.a., Basiswissen Software-Projektmanagement, 3. Aufl., Heidelberg
2009 (zu Kapitel 2)

(Abbildungen in den Folien stammen aus dem
Buch, wenn nicht anders angegeben)



Grundlagen

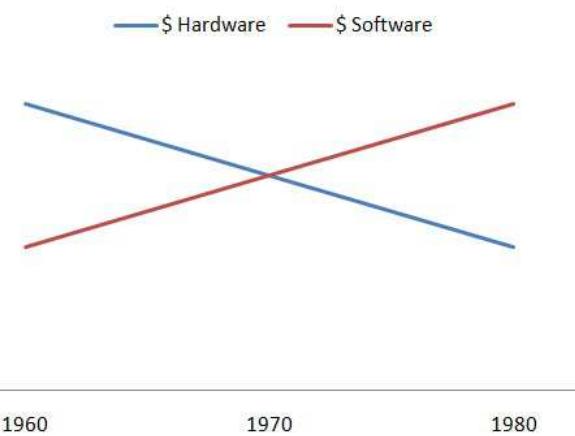
Dokument öffnen mit
wsafmutihaeekdhidbdfdes



Herkunft des Software Engineering

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Softwarekrise
 - Ende der 1960er-Jahre überstiegen die Kosten der Software in vielen Anwendungsfeldern erstmals die Kosten der Hardware.
 - Erstmals scheiterten Anwendungsprojekte an der Software.
 - Man erkannte, daß die Methoden der Softwareentwicklung nicht mit der Technik schrittgehalten hatten.





Herkunft des Software Engineering

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- 1968 wurde in Garmisch eine NATO-Konferenz zum Thema "Software Engineering" durchgeführt. Sie gilt heute als Geburtsstunde des Software Engineering.
 - Ziel: Ingenieurwissenschaftliche Methoden auf die Softwareentwicklung anwenden.
 - <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/>





- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Herkunft des Software Engineering

Brian Randell:

The idea for the first NATO Software Engineering Conference, and in particular that of adopting the then practically unknown term "software engineering" as its (deliberately provocative) title, I believe came originally from Professor Fritz Bauer... As I and other participants have since testified, a tremendously excited and enthusiastic atmosphere developed at the conference. This was as participants came to realize the degree of common concern about what some were even willing to term the "software crisis", and general agreement arose about the importance of trying to convince not just other colleagues, but also policy makers at all levels, of the seriousness of the problems that were being discussed.



Definition Software Engineering

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**



Definition von Ian Sommerville:

"Eine technische Disziplin, die sich mit allen Aspekten der Softwareherstellung beschäftigt, von den frühen Phasen der Systemspezifikation bis hin zur Wartung des Systems, nachdem sein Betrieb aufgenommen wurde"



Standardsoftware und Individualentwicklung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Aus Sicht der Entwicklung liegt der wesentliche Unterschied darin, daß der Ersteller von Standardsoftware
 - ... die Spezifikation selbst übernehmen muß
 - ... die genauen Wünsche der Anwender nicht erfragen kann.



Aktuelle Herausforderungen für das SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Heterogene Umgebungen (Bsp.: Eine Web-Applikation ist davon abhängig, daß Webserver, Applikationsserver, Datenbankserver, ggf. Backendsysteme, ggf. EAI-Middleware gleichzeitig und im 7x24-Betrieb funktionsfähig sind)
- kurze Projektzielzeiten
- Software muß immer verlässlicher sein

Welche weiteren aktuellen Herausforderungen stellen sich für das Software Engineering?



FAQ

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Frage	Antwort
Was ist Software?	Computerprogramme und zugehörige Dokumentation. Softwareprodukte können für einen bestimmten Kunden oder für den allgemeinen Markt entwickelt werden.
Was versteht man unter Software Engineering?	Das Software Engineering ist eine technische Disziplin, die sich mit allen Aspekten der Softwareherstellung beschäftigt.
Worin liegt der Unterschied zwischen dem Software Engineering und der Informatik?	Die Informatik beschäftigt sich mit der Theorie und den Grundlagen; beim Software Engineering geht es um die Entwicklung in der Praxis und die Herstellung nützlicher Software.
Worin liegt der Unterschied zwischen dem Software Engineering und der Systementwicklung?	Die Systementwicklung beschäftigt sich mit allen Aspekten computerbasierter Systementwicklung, darunter Hardware-, Software- und Verfahrensentwicklung. Das Software Engineering ist ein Teil dieses Prozesses.
Was ist ein Softwareprozess?	Ein Satz von Tätigkeiten, deren Ziel die Entwicklung oder die Weiterentwicklung von Software ist.
Was ist ein Vorgehensmodell?	Eine vereinfachte Darstellung eines Softwareprozesses, aus einer bestimmten Perspektive gesehen.
Was kostet das Software Engineering?	Etwa 60 % der Kosten sind Entwicklungskosten, 40 % sind Kosten für Tests. Bei Auftragssoftware übersteigen die Kosten für die Weiterentwicklung oft die für die eigentliche Entwicklung.
Was sind Methoden des Software Engineering?	Strukturierte Ansätze für die Softwareentwicklung, darunter Systemmodelle, Notationen, Regeln, Hinweise zum Entwurf und Anleitung zum Vorgehen.
Was ist CASE (Computer-Aided Software Engineering)?	Softwaresysteme, die für die automatisierte Unterstützung von Software-Entwicklungstätigkeiten bestimmt sind. CASE-Systeme werden oft für die Unterstützung von Methoden verwendet.
Welche Merkmale zeichnen gute Software aus?	Die Software sollte dem Benutzer die verlangten Funktionen zur Verfügung stellen sowie einfach zu warten, zuverlässig und gut bedienbar sein.
Worin liegen die größten Herausforderungen für das Software Engineering?	Der erfolgreiche Umgang mit zunehmender Vielfalt, mit Forderungen nach verkürzten Lieferzeiten und mit der Entwicklung vertrauenswürdiger Software.



Merkmale guter Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Produkteigenschaft	Beschreibung
Wartbarkeit	Software sollte so geschrieben werden, dass sie weiterentwickelt werden kann, um veränderten Kundenbedürfnissen Rechnung zu tragen. Das ist ein entscheidendes Merkmal, weil Softwareveränderungen eine unvermeidliche Konsequenz einer sich verändernden Geschäftsumgebung sind.
Zuverlässigkeit	Die Softwarezuverlässigkeit hat eine ganze Reihe von Merkmalen, darunter Verlässlichkeit, Zugriffsschutz und Betriebssicherheit. Zuverlässige Software sollte keinen körperlichen oder wirtschaftlichen Schaden verursachen, falls das System ausfällt.
Effizienz	Software sollte nicht verschwenderisch mit Systemressourcen wie Speicher und Prozessorkapazität umgehen. Effizienz umfasst somit Reaktionszeit, Verarbeitungszeit, Speichernutzung usw.
Benutzerfreundlichkeit	Software muss ohne unangemessene Anstrengungen von dem Benutzer nutzbar sein, für den sie entwickelt wurde. Das bedeutet, dass sie über eine angemessene Benutzeroberfläche und ausreichende Dokumentation verfügen sollte.

Welche weiteren Merkmale guter Software fallen Ihnen ein?



Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- a) Recherchieren Sie:
Was war Therac-25? Was genau ist dabei schiefgelaufen?
- b) Welche Maßnahmen schlagen Sie vor, um solche Fehler in zukünftigen Softwareprojekten zu verhindern?



Kritische Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Von manchen Systemen hängen Menschenleben ab oder ihr Versagen verursacht schweren wirtschaftlichen Schaden.
- Sicherheitskritische Systeme: Ausfall kann zu Schäden für Mensch und Umwelt führen. Bsp.: Steuerung für Atomkraftwerk
- Aufgabekritische Systeme: Ausfall kann zum Scheitern einer in Relation zum System selbst umfangreichen Aufgabe führen. Bsp.: Navigationssystem in der Raumfahrt
- Geschäftskritische Systeme: Ausfall kann hohe Kosten zur Folge haben. Bsp.: Informationssystem einer Börse



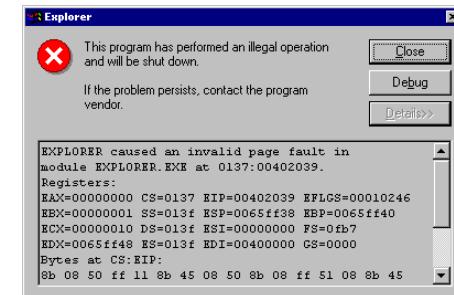


Kritische Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Die wichtigste Eigenschaft kritischer Systeme ist Ihre Zuverlässigkeit, denn:
 - Wenn ein kritisches System unzuverlässig arbeitet, wird es von den Benutzern abgelehnt (und alle anderen Systeme des selben Herstellers auch)
 - Die Kosten eines Ausfalls können so hoch sein, daß die Existenz von Unternehmen gefährdet ist.
 - Verlorene Informationen können wertvoller sein als das System selbst.

Fallen Ihnen weitere Gründe ein?





Entwicklung kritischer Systeme

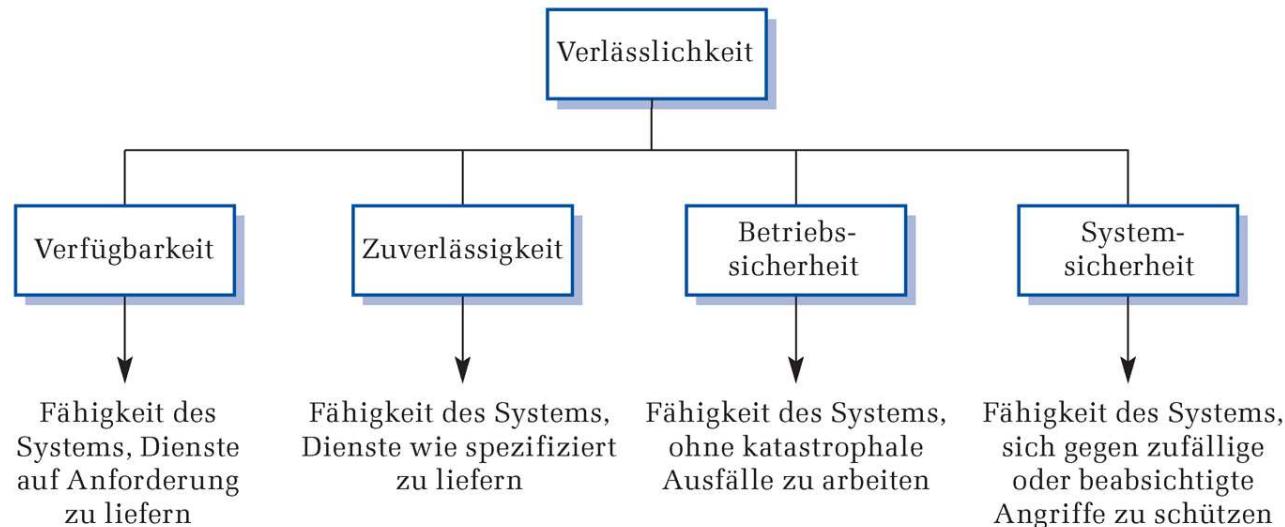
- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Lieber ausgereifte als neue Techniken verwenden
- Kosten für Tests sind relativ höher als bei unkritischen Systemen.
- Besonders aufwendige Methoden des SE lohnen sich vor allem bei kritischen Systemen.
- Hardwarefehler, Softwarefehler und Bedienungsfehler können sich gegenseitig bedingen. Wichtig ist daher die Sicht auf das Gesamtsystem.



Verlässlichkeit kritischer Systeme

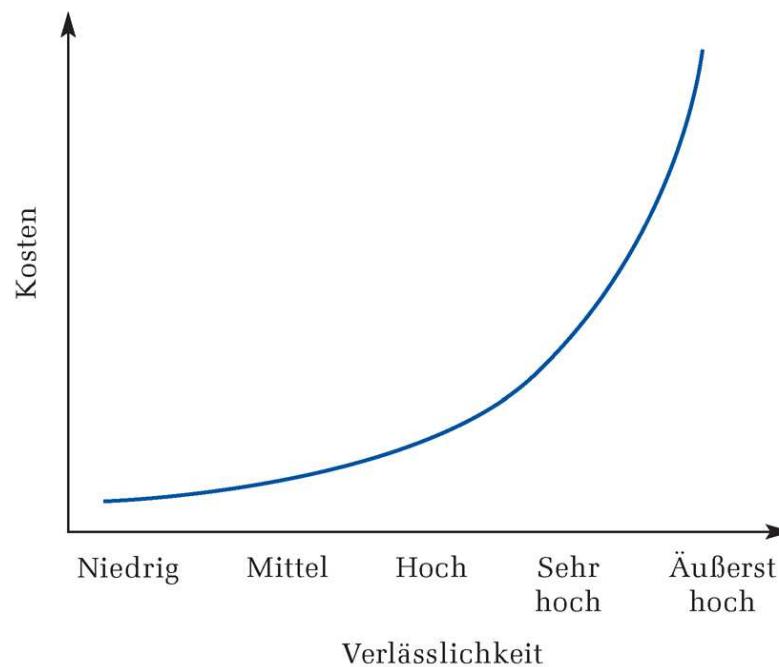
- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**





- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Kosten kritischer Systeme



Entwicklung kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Es ist möglich, daß manchen Benutzern ein System zuverlässig erscheint, anderen aber nicht.

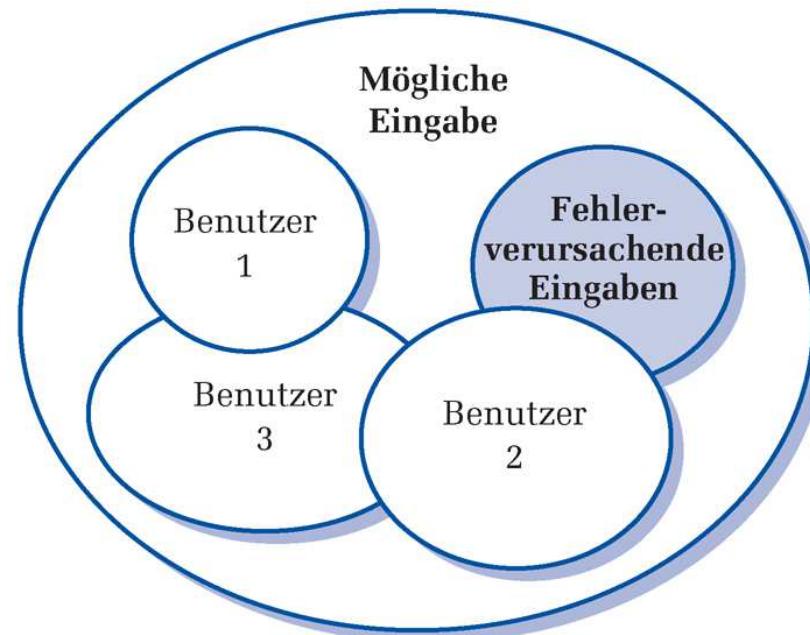




1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Entwicklung kritischer Systeme

- Der Grund: Sie nutzen immer die selbe Teilmenge der Funktionalität.





Einige Begriffe zur Systemsicherheit

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

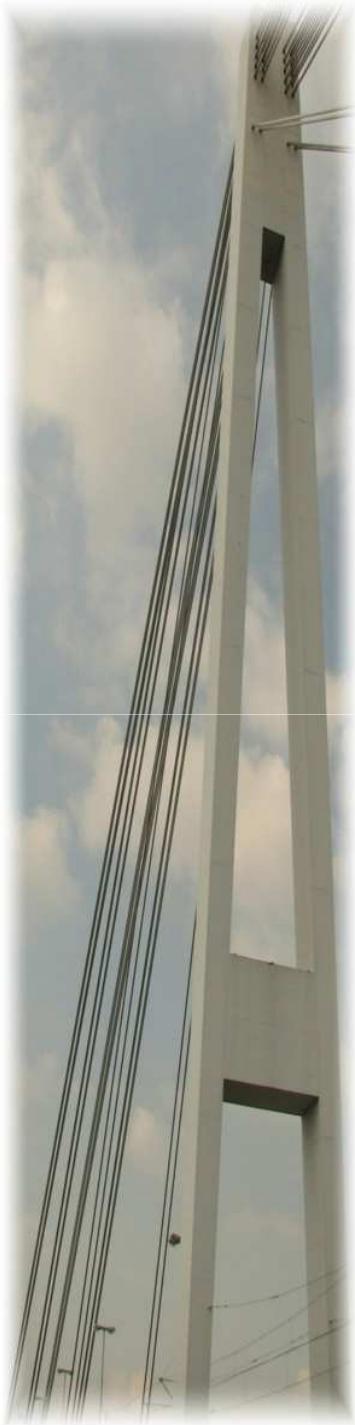
Begriff	Beschreibung
Unfall (Accident, Mishap)	Ein nicht geplantes Ereignis oder eine Folge von Ereignissen, die den Tod oder die Verletzung von Menschen, Sach- oder Umweltschäden zur Folge haben. Eine computergesteuerte Maschine, die ihren Bediener verletzt, ist ein Beispiel für einen Unfall.
Gefahr (Hazard)	Ein Umstand, der das Potenzial besitzt, einen Unfall zu verursachen oder dazu beizutragen. Der Ausfall eines Sensors, der ein Hindernis vor einer Maschine erkennt, ist ein Beispiel für eine Gefahr.
Schaden (Damage)	Ausmaß des Verlustes, der aus einem Zwischenfall resultiert. Ein Schaden kann von vielen Menschen, die durch einen Unfall getötet werden, bis zu einer unbedeutenden Verletzung oder Sachschäden reichen.
Gefahrenausmaß (Hazard Severity)	Bewertung des schlimmsten Schadens, der sich aus einer bestimmten Gefahr ergeben könnte. Das Gefahrenausmaß kann von katastrophal (Tod vieler Menschen) bis hin zu unbedeutend (kleinere Schäden) reichen.
Gefahrenwahrscheinlichkeit (Hazard Probability)	Wahrscheinlichkeit für das Auftreten von Ereignissen, die eine Gefahr implizieren. Wahrscheinlichkeitswerte sind scheinbar willkürlich, reichen aber von <i>wahrscheinlich</i> (eine Chance von 1:100, dass ein Ereignis auftritt) bis zu <i>unplausibel</i> (keine vorstellbare Situation, bei der die Gefahr auftreten könnte).
Risiko (Risk)	Dies ist ein Maß der Wahrscheinlichkeit, dass das System einen Unfall verursachen wird. Das Risiko wird bewertet, indem die Gefahrenwahrscheinlichkeit, das Gefahrenausmaß und die Wahrscheinlichkeit, dass eine Gefahr zu einem Unfall führt, in Betracht gezogen werden.



Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Diskutieren Sie, ob für die Ausübung des Berufs als Softwareentwickler eine Zulassung wie für Ärzte und Anwälte zur zwingenden Voraussetzung gemacht werden sollte.



Vorgehensmodelle



Vorgehensmodelle

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Vorgehensmodelle definieren Standards für IT-Projekte, vor allem für
 - Projektphasen (z.B. Entwicklung und Test)
 - Projektorganisation (z.B. Rollen und Aktivitäten)
 - Dokumente (z.B. Requirement Spec, Change Request)
 - Kommunikationsbeziehungen zwischen den Beteiligten
 - Methoden (z.B. Für die Aufwandsschätzung)
- Manche Vorgehensmodelle sehen ausdrücklich vor, an spezielle Anforderungen angepaßt zu werden (Tailoring).
- Manche Vorgehensmodelle sind hochspezialisiert (z.B. Raymonds Bazaar für Open Source Projekte)



Zweck von Vorgehensmodellen

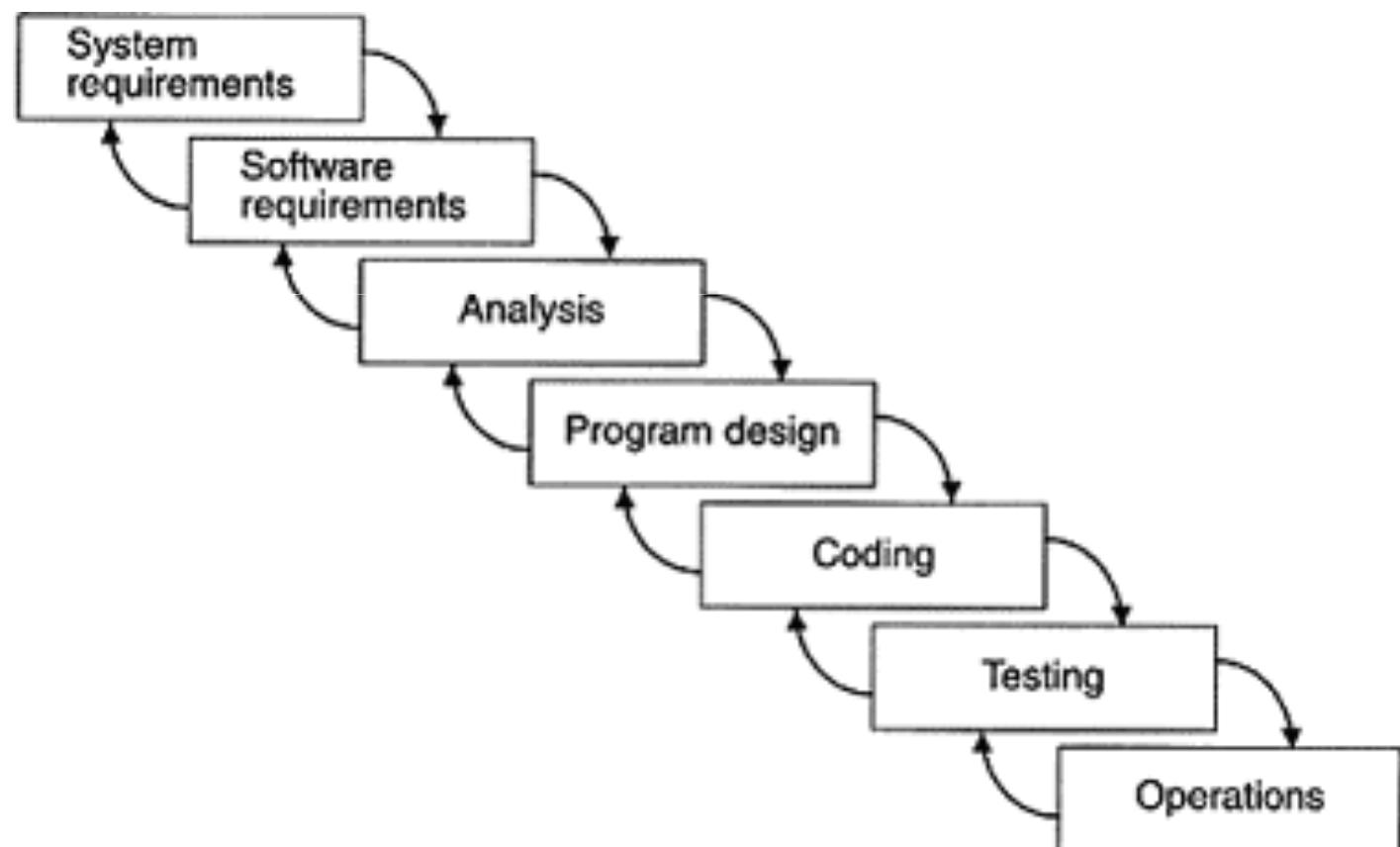
- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Sie fassen Erfahrungen aus vergangenen Projekten zusammen.
- Sie geben eine Grundstruktur für die Projektplanung vor. (Wie sollen wir vorgehen?)
- Sie helfen beim Assessment (Was ist bislang gut, was ist schlecht gelaufen?)



Wasserfallmodell

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung



Quelle: <http://www.maxwideman.com/papers/plc-models/1990s.htm>

Prof. Dr. Carsten Dorrhauer



Wasserfallmodell

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Wasserfallmodell: die Anzahl und Benennung der einzelnen Phasen variiert. Wichtiger ist:
 - Die Reihenfolge lautet: Anforderungen, Design, Implementierung, Test, Inbetriebnahme
 - Die Phasen werden sequentiell durchlaufen.
 - Am Ende jeder Phase steht ein Meilenstein.
 - Eine Rückkopplung im Falle von offensichtlichen Fehlern ist nur über eine Phase möglich, auch auf dem Rückweg wird keine Phase übersprungen.
- Vorteile
 - unmittelbar für jedermann verständlich
 - Qualitätskontrolle durch Meilensteine am Ende jeder Phase



Wasserfallmodell

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Nachteile
 - Das Wasserfallmodell unterstellt, daß nach Abschluß der Anforderungsanalyse alle Anforderungen im Detail bekannt sind und sich nicht mehr ändern.
 - Es gibt keine Prototypen.
 - Erst das fertige Softwareprodukt steht nach Abschluß der Entwicklung zum Testen zur Verfügung.
 - Erst das fertig getestete System wird den Anwendern vorgestellt.
 - Mißverständnisse bei der Anforderungsanalyse werden spät erkannt.
 - Keine Iterative Entwicklung (über mehrere Softwarereleases)



Wasserfallmodell

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Viele Experten raten grundsätzlich vom Wasserfallmodell ab. Dennoch wird das Modell bis heute häufig angewandt.
- Sinnvoll ist es nur, wenn die Anforderungen klar und unmißverständlich formulierbar sind und keine Änderungen der Anforderungen zu erwarten sind.
- Beispiel für einen sinnvollen Einsatz: Erstellung einer Datenbank für Sicherheitszwischenfälle in der Unternehmens-IT
 - Kleines, überschaubares Projekt mit kurze Laufzeit und wenigen Mitarbeitern.
 - IT-interne Anwendung, d.h. die Anwender sind selbst IT-Experten. Mißverständnisse über die Anforderungen sind unwahrscheinlich.



V-Modell des Bundes

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Vorgehensmodell für IT-Systeme der öffentlichen Hand in Deutschland
- Gepflegt und weiterentwickelt vom Beauftragten der Bundesregierung für Informationstechnik
- Aktuell: V-Modell XT 2.1
- Es ist jahrzehntelang weiterentwickelt worden, hat viele Aspekte aus anderen Vorgehensmodellen (agile, iterative) integriert.
- Es ist inzwischen sehr umfangreich.

https://www.cio.bund.de/Web/DE/Architekturen-und-Standards/V-Modell-XT/vmodell_xt_node.html





- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Inkrementell-iterative Methoden

- Initial wird die Kernfunktionalität unter Berücksichtigung der technischen Herausforderungen erstellt.
- In mehreren Iterationen wird sie ergänzt.
- Zwischenprodukte stehen zumindest als Prototyp zum Test zur Verfügung.
- Vorteil: Fehler können früh erkannt werden.
- Nachteil: Technisches Design muß eigentlich immer die "Endausbaustufe" berücksichtigen, sonst fällt Wegwerfaufwand an. (Bsp.: Web-Frontend einer Applikation als letzte Iteration)

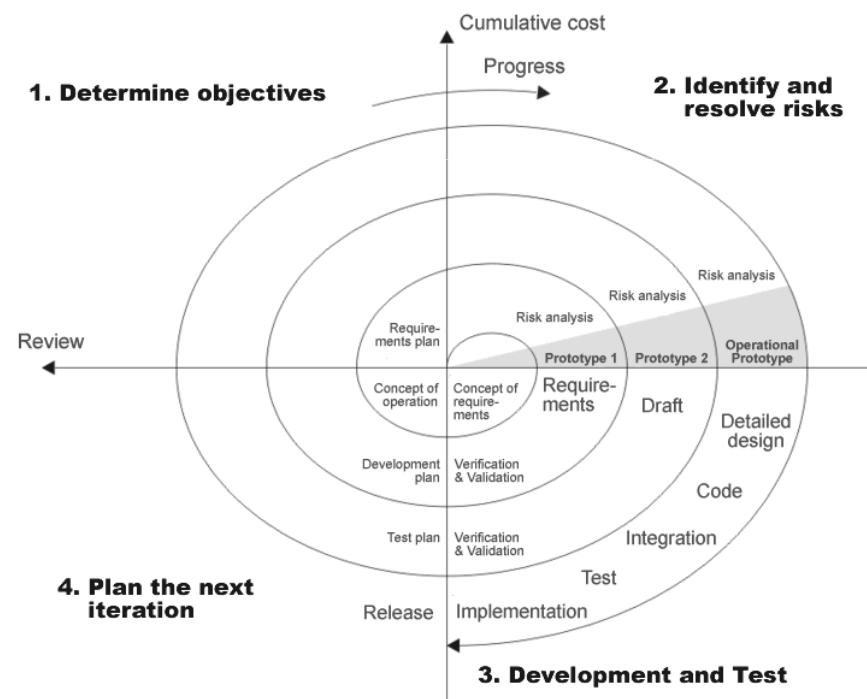




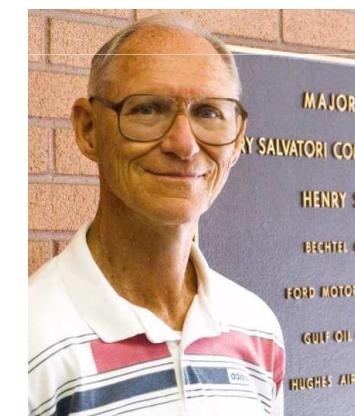
Spiralmodell nach Barry Boehm

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Inkrementell-iteratives Modell, bei dem jeder Zyklus aus 4 Quadranten besteht: Zieldefinition, Risikoabschätzung, Implementierung und Test, Planung des nächsten Zyklus



Quelle: leansoftwareengineering.com





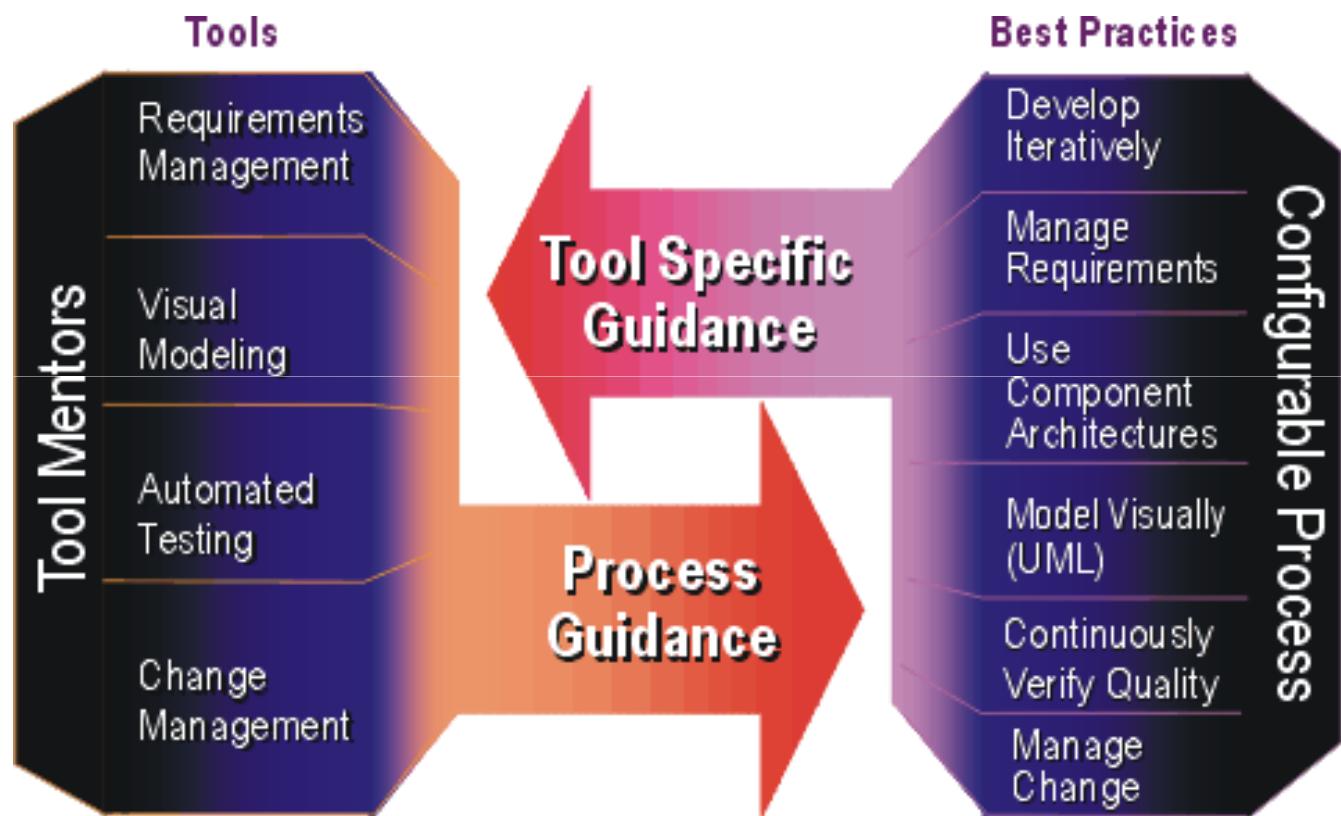
Rational Unified Process (RUP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Rational Unified Process und UML
 - Die UML enthält kein Vorgehensmodell.
 - Rational Unified Process ist ein Vorgehensmodell für Projekte der objektorientierten Softwareentwicklung, das sich auf die UML bezieht.
 - Rational macht das Vorgehensmodell frei verfügbar.
 - RUP sieht aber an vielen Stellen den Einsatz bestimmter Werkzeuge vor.
 - Rational verkauft solche Werkzeuge.
 - Rational gehört inzwischen zu IBM.

Rational Unified Process (RUP)

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung



Quelle: Evaluationsversion des RUP, frei zum Download unter <http://www-306.ibm.com/software/awdtools/rup/>



Rational Unified Process (RUP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Dreh- und Angelpunkt des RUP sind die 6 sogenannten Best practices, die von den Rational-Werkzeugen unterstützt werden.
 - Iterative Entwicklung
 - Anforderungsmanagement
 - Komponentenbasierte Architektur
 - Visuelle Modellierung, i.d.R. mittels der UML
 - Permanente Qualitätskontrolle
 - Management von Änderungen



Rational Unified Process (RUP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Vorteile des RUP
 - Iterative Entwicklung ermöglicht frühe (und damit kostengünstige) Fehlererkennung
 - Einbindung aktueller softwaretechnischer Methoden
 - Geeignet für die Entwicklung objektorientierter Software mit Komponentenarchitektur
 - Kein rein sequentielles Modell, Parallelisierung von Aktivitäten
 - Verfügbarkeit unterstützender (allerdings kostenpflichtiger) Tools.



Rational Unified Process (RUP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Nachteile des RUP
 - Komplex. Hoher initialer Einarbeitungsaufwand.
 - Speziell vorgesehen für objektorientierte Softwareentwicklungsprojekte. Deshalb ungeeignet für Hardware, für nicht-objektorientierte Legacy-Erweiterungen, für die Einführung von Standardsoftware.
 - Angepaßt auf die Tools von Rational
 - mit anderen Werkzeugen erhöhter Aufwand
 - ganz ohne entsprechende Werkzeuge nicht einsetzbar



Agile Vorgehensmodelle

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Entstanden in den letzten Jahren in fundamentaler Opposition zu herkömmlichen Methoden
- Strukturierung des Projektes, Formalisierung, Dokumentation werden als übertriebene Bürokratie empfunden. Für all dies gilt: So wenig wie möglich, so viel wie gerade noch nötig.
- Jedes Projekt ist anders, deswegen kann es allgemein gültige Regeln nicht geben. Der Mensch steht im Mittelpunkt, Prozesse und Werkzeuge sind weniger wichtig.



Agile Vorgehensmodelle

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

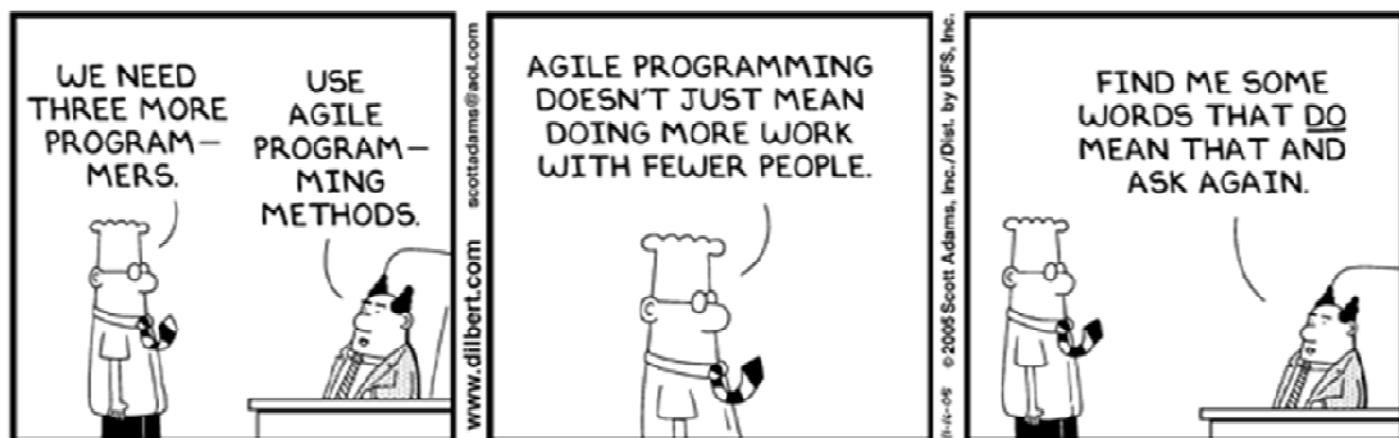
- Die durch Bürokratieabbau gesparte Zeit soll in die eigentliche Softwareentwicklung investiert werden.
- Die Anwender werden in allen Projektphasen eng in das Projekt eingebunden. Vertrauen geht vor detaillierter vertraglicher Absicherung.



Agile Vorgehensmodelle

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Auf Änderungen im Projektverlauf wird flexibel reagiert. Kundenzufriedenheit geht vor Einhaltung überholter Ziele.
- Agiles Manifest: <http://www.agilemanifesto.org/>
- Die bekanntesten agilen Methoden sind Extreme Programming und Scrum.



© Scott Adams, Inc./Dist. by UFS, Inc.



Extreme Programming (XP)

- Entwickelt u.a. von Kent Beck.
Erstmals eingesetzt bei Chrysler.
- Bekannteste agile Methode: Extreme Programming
- Entwickler und Anwender in einem Raum
- 40-Stunden-Woche, mehr Arbeit mindert die Produktivität.





Extreme Programming (XP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- "Planning Game": Die Anwender definieren die Anforderungen in "Stories", die Entwickler schätzen den Aufwand, beide verhandeln die Reihenfolge der Entwicklung.
- Ständige Wiederholung des kompletten Zyklus (Anforderung, Design, Entwicklung, Test) in kurzen Zeitabständen bei ständiger Verbesserung mit sofortige Rückmeldung der Anwender.



Extreme Programming (XP)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

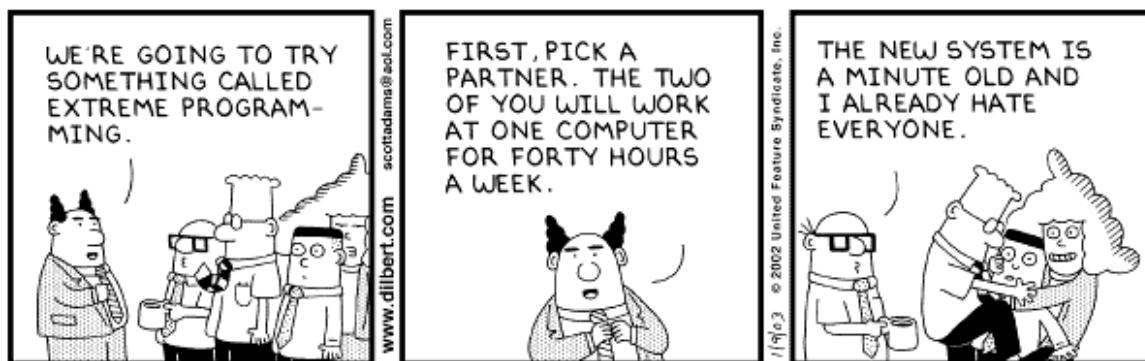
- Refactoring: Codeverbesserung ohne zusätzliche Funktionalität
- Entwicklungsstandards: Keine persönlichen Vorlieben bei der Codegestaltung
- Jeder ist verantwortlich für das Gesamtergebnis, deswegen darf auch jeder in den Programmteilen der anderen Ergänzungen vornehmen, wenn sie zu seiner Aufgabe gehören, aber aus architektonischen Gründen in den Programmteilen der anderen implementiert werden sollten.



Extreme Programming (XP)

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Von mehreren möglichen Lösungen wird nicht die raffinierteste, sondern die einfachste bevorzugt.
- Zwei Programmierer teilen sich einen Arbeitsplatz: Einer entwickelt, der andere achtet darauf, daß keine Fehler passieren.
- Ständiges Deployment (Zusammenführen der Arbeitsergebnisse der Zweiergruppen), um Unverträglichkeiten sofort zu entdecken.



Copyright © 2003 United Feature Syndicate, Inc.



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Extreme Programming (XP)

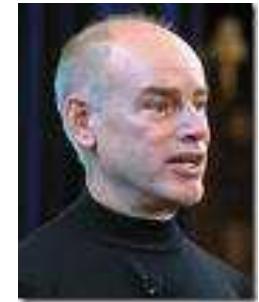
- Würdigung des XP:
 - XP ist eher für kleinere Projekte geeignet.
 - XP setzt ein großes Vertrauensverhältnis voraus und ist daher eher anwendbar, wenn Entwickler und Anwender zu einem Unternehmen gehören (interne IT-Abteilung)
 - Die Dokumentation fehlt weitgehend. Wenn das System später gepflegt werden soll, sollte immer ein Entwickler aus dem ursprünglichen Teamverfügbar sein.
 - Einige Vorschläge des XP lassen sich gewinnbringend auch in herkömmlichen Projekten umsetzen, z.B. Refactoring (Code review) und verbindliche Codierungsstandards.
 - XP besteht aus einigen wenigen Grundregeln. Die Projektdurchführung wird viel weniger detailliert vorgegeben als z.B. bei RUP oder V-Modell XT.



Scrum

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

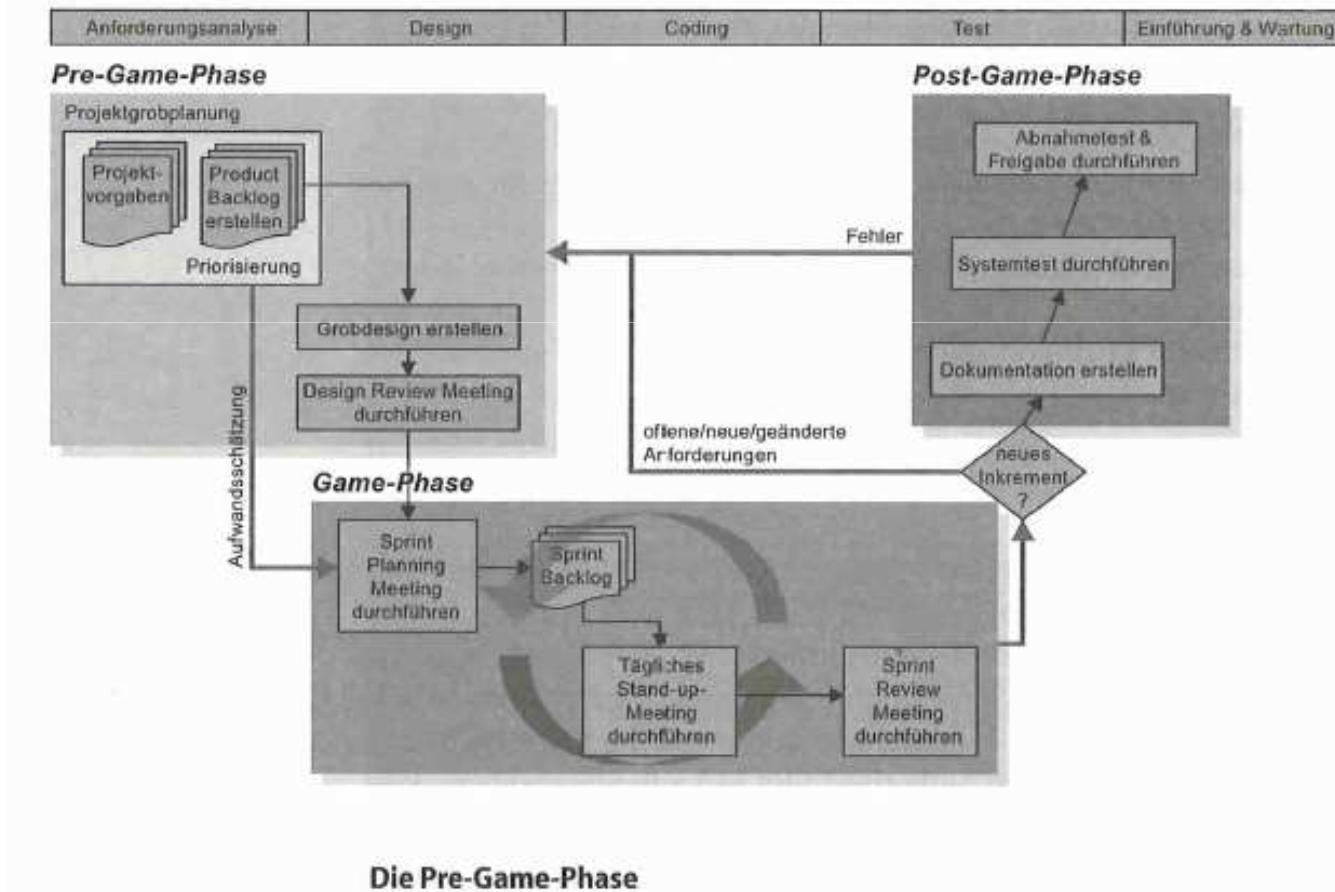
- Wurde von Ken Schwaber u.a. entwickelt.
- Idee: Die Prozesse eines Softwareprojektes sind ohnehin zu komplex, um sie detailliert beschreiben zu können.
- Scrum gibt deshalb nur einen groben Rahmen vor. Schwerpunkt sind nicht konkrete Praktiken wie bei XP, sondern das Projektmanagement.
- In diesem Rahmen können Projekte gestaltet werden, indem das Projektteam selbst sich Richtlinien gibt.
- Das Team trägt gemeinsam Verantwortung für das Ergebnis.



Scrum

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Scrum kennt drei Phasen: Pre-Game, Game, Post-Game





1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Scrum: Pre-Game-Phase

- Das Projektteam wird zusammengestellt.
- Man einigt sich auf Standards und Werkzeuge.
- Der *Product Owner* erstellt das *Product Backlog*. Darin sind alle bisher bekannten Anforderungen enthalten. Es ist nicht abgeschlossen, sondern wird ständig weiterentwickelt.
- Alternative Vorschläge zum Systemdesign werden diskutiert. Dabei wird noch nicht detailliert, sondern die Architektur bestimmt.
- Im *Design Review Meeting* wird ein Grobdesign beschlossen.





Scrum: Game-Phase

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Das Produkt wird in mehreren Abschnitten erstellt, die ein nachprüfbares Ergebnis haben, etwa einen Monat dauern und *Sprint* heißen.
- Für jeden *Sprint* gilt:
 - Im *Sprint Planning Meeting* wird zunächst von *Product Owner* und Projektteam festgelegt, was das Ziel des nächsten Sprints sein soll und wie es zu erreichen ist.
 - Das *Sprint Backlog* wird angelegt und gepflegt. Es enthält die Ziele des laufenden Sprint. Es enthält eine Auswahl jener Ziele des *Product Backlog*, die für diesen Sprint vorgesehen sind. Bezuglich dieser Ziele ist es detaillierter als der *Product Backlog*.



Scrum: Game-Phase

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Für jeden *Sprint* gilt:
 - Der *Product Backlog* kann zwar vom *Product Owner* geändert werden, nicht aber der *Sprint Backlog*. D.h. Änderungen von außen kommen frühestens im nächsten *Sprint* zum Tragen.
 - Das Team entscheidet selbst, wer welche Punkte aus dem *Sprint Backlog* als nächstes bearbeitet.
 - Jedes Teammitglied beantwortet in einem kurzen *Daily Scrum* folgende Fragen:
 - Was habe ich seit gestern erreicht?
 - Was will ich bis morgen erreichen?
 - Was blockiert meine Arbeit?
 - Täglich wird ein *Burndown Chart* fortgezeichnet, das den Restaufwand auf die verstrichene Zeit bezieht.



Scrum: Game-Phase

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Zum Abschluß des Sprint stellt in einem *Sprint Review Meeting* das Projektteam dem *Product Owner* und den *Stakeholders* seine Ergebnisse vor.
- Dann wird entschieden, ob das *Product Backlog* angepaßt und ein neuer *Sprint* begonnen oder in die Post-Game-Phase übergegangen wird.



Scrum: Post-Game-Phase

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Erstellung der Dokumentation
- Systemtest. Im Falle von Fehlern wird das Product Backlog angepaßt und ein neuer Sprint gestartet.
- User Acceptance Test



Scrum: Rollen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Der Scrum Master
 - überwacht die Einhaltung der Scrum-Regeln.
 - ist kein Teammitglied.
 - ist kein Teamleiter. Das Team organisiert sich selbst.
 - ist bei Schwierigkeiten Ansprechpartner für die Teammitglieder.
 - trägt die Blockaden, die im *Daily Scrum* von den Teammitgliedern genannt werden in den Impediment Backlog ein und ist dafür verantwortlich, sie zu beseitigen.
 - Kann einen Sprint abbrechen, wenn er die Ziele für unerreichbar oder die Hindernisse für unüberwindbar hält.



Scrum: Rollen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Der Product Owner
 - kommuniziert mit den Kunden und vertritt diese gegenüber dem Team.
 - definiert die Projektziele anhand von User Stories.
 - priorisiert die Einträge im *Product Backlog*.
 - kommuniziert direkt mit dem Team.
- Das Projektteam
 - schätzt die Aufwände für die Einträge im *Product Backlog*.
 - wählt anhand der Prioritäten des *Product Owners* aus den *Product Backlog*-Einträgen die Ziele für den nächsten *Sprint*.



Scrum: Rollen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Die Kunden/ Stakeholders
 - finanzieren das Projekt
 - bekommen im Sprint Review Meeting die Ergebnisse des letzten Sprint präsentiert.
- Die Manager
 - sind die Vorgesetzten des Teams.
 - stellen die Arbeitsumgebung zur Verfügung



Scrum

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Würdigung von Scrum
 - strukturiert, aber nicht komplex
 - es gelten die generellen Vor- und Nachteile agiler Methoden
 - Scrum mildert einige Nachteile anderer agiler Methoden ab. So darf z.B. der Product Owner, der die Kunden vertritt, während eines Sprint die Ziele nicht ändern.
 - Es wird von erfolgreichen Projekten berichtet.
 - Scrum ist recht jung. Für eine abschließende Beurteilung ist es noch früh.



Continuous Integration

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Agile Methoden beinhalten das Konzept, möglichst oft
 - neue, verbesserte Prototypen für Tester sowie
 - verbesserte Produktivversionen für Endbenutzer bereitzustellen.
- Im Extremfall: Continuous Integration
 - Mehrfach täglich neue Versionen
 - Automatisierter Build mittels geeigneter Werkzeuge
 - Automatisierter Test mittels geeigneter Werkzeuge
 - Alle Entwickler liefern mindestens täglich neuen Code zur Integration (commit)
 - Alle neuen Commits werden in einen Build integriert
- Vorteil: Fehler werden sofort erkannt.
- Nachteil: Wenn laufend neue Commits in die Queue gestellt werden, wird es schwierig, immer letzte stabile Versionen zu definieren.



DevOps

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Zielkonflikt:
 - Aus Sicht agiler Projekte ist häufiges Deployment wünschenswert, ggf. Continuous Integration
 - Aus Sicht des Systembetriebs ist spätestens nach dem Go-Live jede Änderung eine potentielle Störungsursache, deshalb sollte es nicht zu viele geben und diese ausgiebig getestet werden.
- 2009 organisiert Patrick Debois die ersten DevOpsDays in Gent
- Idee: Gemeinsame Ziele, Abläufe und Werkzeuge (Goals, Processes, Tools) für Entwicklungsprojekte und Systembetrieb



Low Res Image © Tim Deneka





DevOps

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- In vielen IT-Organisationen sind Entwicklungsprojekte und Systembetrieb traditionell voneinander getrennt.
- Nach DevOps werden Teams für eine Applikation aus Entwicklern/Testern und Operatoren/Administratoren zusammengestellt.
- DevOps ist im Grunde unabhängig vom Vorgehensmodell, ergibt aber besonders mit agilen Methoden und CI Sinn.
- Das Team ist fortlaufend zuständig für die Entwicklung neuen Codes und Systembetrieb.

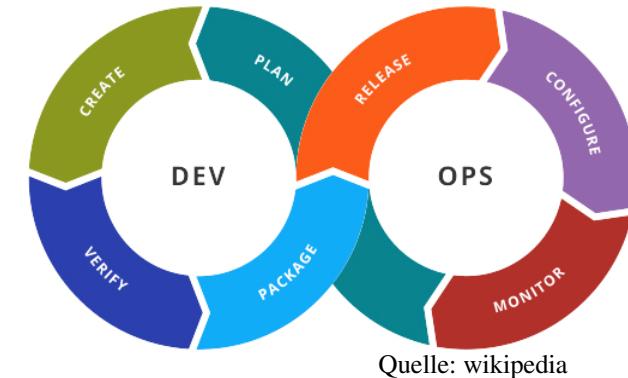


DevOps Toolchain

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Diese Kette soll von geeigneten Werkzeugen unterstützt werden:

- Create: Entwicklung und Build
- Verify: Verifikation und Validierung
- Package: Einbinden z.B. von Graphiken, Multimedia etc.
– Erstellen von z.B. JAR
- Release: Stellen und dokumentieren von RfC nach den Richtlinien der IT-Organisation
- Configure: Einrichten und Betreiben der Applikation und ihrer Infrastruktur
- Monitor: Automatisiert und nach Anwender-Feedback (Events und Incidents)
- Plan: Entwurf der nächsten Verbesserung von Produkt und Betriebsprozeß



Quelle: wikipedia



Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

In Ihrem Unternehmen soll ein E-Procurement-System eingeführt werden. Der CIO und der Leiter der Beschaffungsabteilung benennen jeweils zwei ihrer Mitarbeiter, die sich zunächst auf eine grobe Vorgehensweise für das Projekt einigen sollen.

Diskutieren Sie in Gruppen à 4 Personen (2 aus der IT, 2 aus der Beschaffung) mögliche Vorgehensmodelle und erarbeiten Sie (in Stichworten) einen Entscheidungsvorschlag für die beiden Vorgesetzten.

Unterstellen Sie dabei sinnvolle Projektparameter (Infrastruktur, Unternehmensgröße, Anzahl Anwender, Anzahl Lieferanten etc.) und leiten Sie daraus Ihre Empfehlung ab.

Es kommt nicht darauf an, für welches Vorgehensmodell Sie sich entscheiden, sondern darauf, daß Sie diese Entscheidung schlüssig aus Ihren Annahmen ableiten.



Spezifikation von Software



Softwareanforderungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Die Anforderungen an ein Softwaresystem müssen schriftlich fixiert werden, um Mißverständnisse zu vermeiden.
- Funktionale Anforderungen beschreiben, was das System leisten soll.
- Nichtfunktionale Anforderungen beschreiben Beschränkungen der vom System angebotenen Dienste.
- Domänenanforderungen betreffen nicht nur eine einzelne Applikation, sondern ihren Problembereich (Domain)

Softwareanforderungen

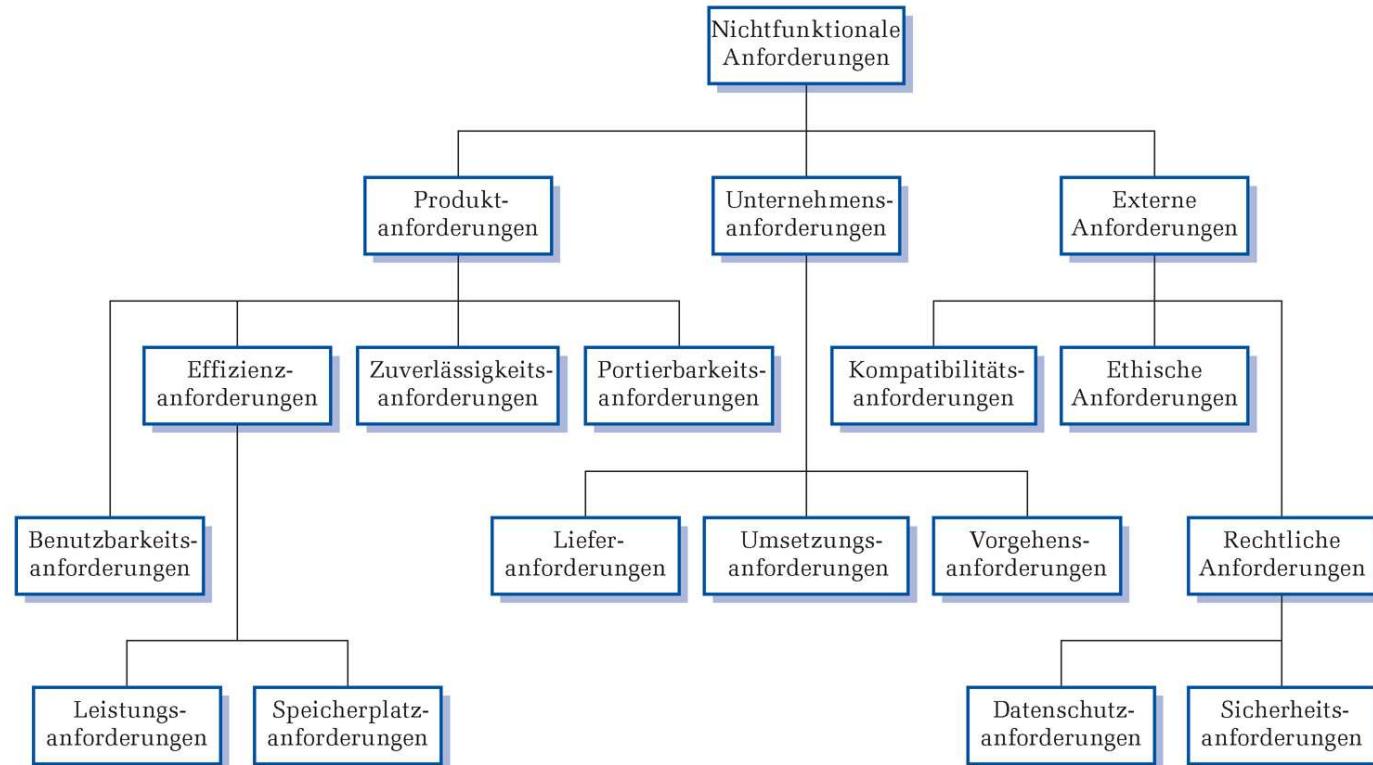
1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Funktionale Anforderungen sollen
 - vollständig sein, also alles enthalten, was der Benutzer benötigt
 - konsistent sein, also keine Widersprüche enthalten.



Softwareanforderungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**





Softwareanforderungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Nichtfunktionale Anforderungen werden oft als Systemziel formuliert. Besser sind meßbare Formulierungen.

Ein Systemziel

Das System sollte für erfahrene Benutzer einfach zu bedienen und so aufgebaut sein, dass Fehler durch den Benutzer minimiert werden.

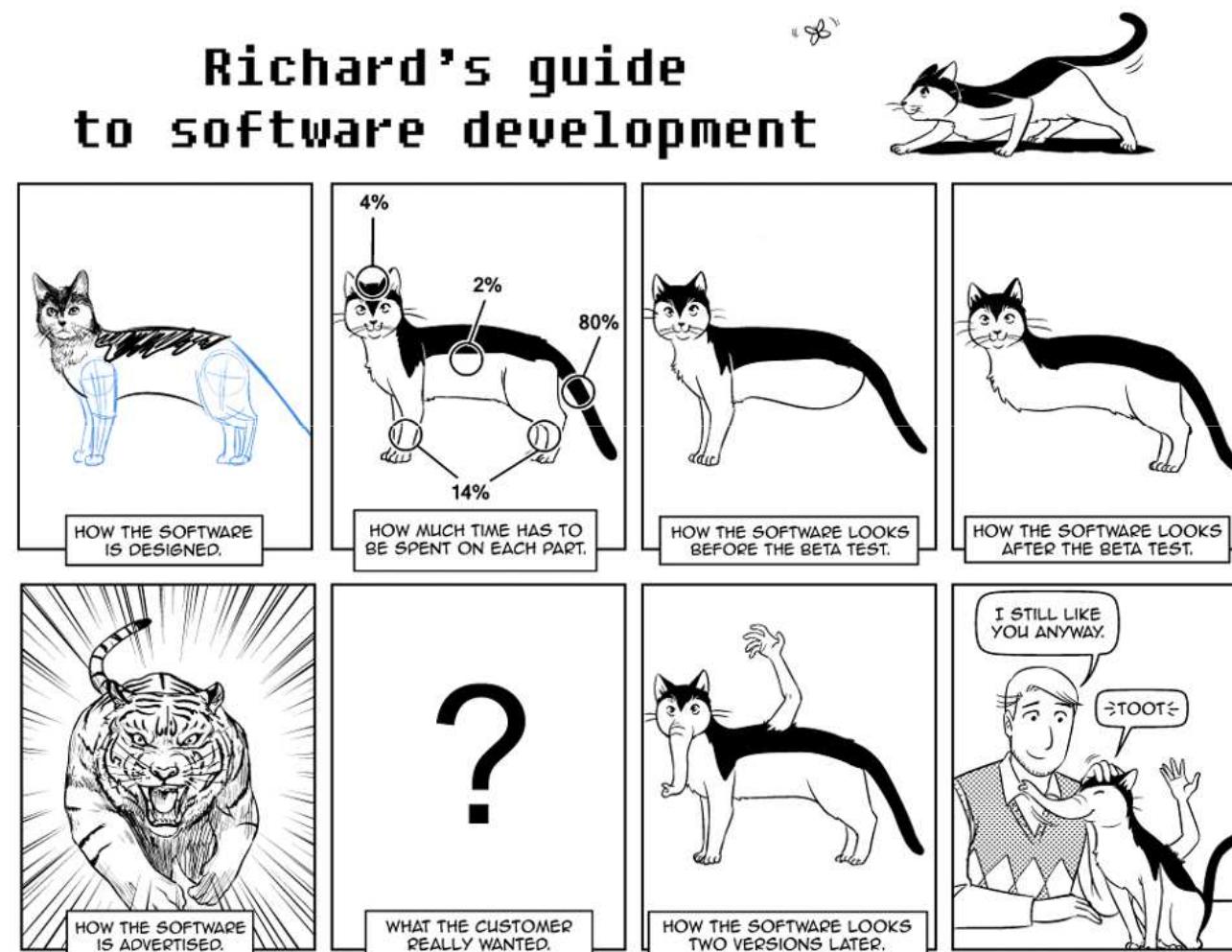
Eine verifizierbare nichtfunktionale Anforderung

Es soll erfahrenen Benutzern möglich sein, nach einer Schulung von insgesamt zwei Stunden alle Systemfunktionen zu verwenden. Nach dieser Schulung sollte der durchschnittliche Wert gemachter Fehler bei erfahrenen Benutzern nicht höher als zwei pro Tag sein.



Softwareanforderungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung



Sandra and Woo by Oliver Knörzer (writer) and Powree (artist) – www.sandraandwoo.com



Softwareanforderungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Verifizierbarkeit setzt Metriken voraus

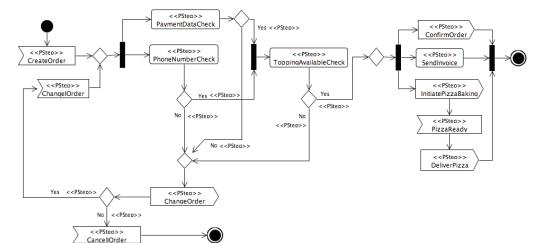
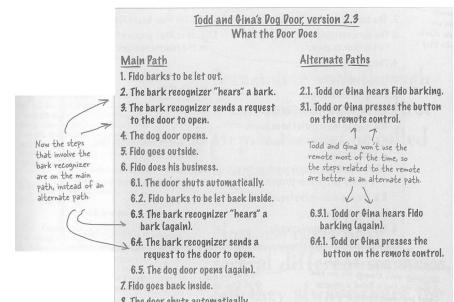
Eigenschaft	Maßeinheit
Geschwindigkeit	Ausgeführte Transaktionen/Sekunde
	Reaktionszeit auf Benutzereingabe oder Ereignis
	Bildschirmauffrischungszeit
Größe	Kilobyte
	Anzahl der Speicherbausteine
Benutzerfreundlichkeit	Schulungsdauer
	Anzahl der Hilfeseiten
	Zuverlässigkeit
Zuverlässigkeit	Durchschnittliche Zeit bis zu einer Fehlfunktion
	Wahrscheinlichkeit der Nichtverfügbarkeit
	Quote für das Auftreten von Fehlern
	Verfügbarkeit
Stabilität	Zeit bis zum Neustart nach einer Fehlfunktion
	Anteil der Ereignisse, die zu Fehlfunktionen führen
	Wahrscheinlichkeit für Datenzerstörung bei Fehlfunktion
Portierbarkeit	Anteil der plattformabhängigen Anweisungen
	Anzahl der Zielsysteme



Softwareanforderungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Anforderungen können formuliert werden:
 - in natürlicher Sprache
 - Nachteil: ungenaue Formulierungen
 - Nachteil: verwirrende Formulierungen
 - Nachteil: Gefahr der Verschmelzung von Anforderungen
 - in vorstrukturierten Formularen (Bsp.: Use-Case-Dokument)
- als graphisches Modell (Bsp.: Sequenz- und Aktivitätendiagramme aus der UML)





Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**

- 11. Aspekt-orientierte Entwicklung**

Finden Sie Mehrdeutigkeiten und Lücken in der folgenden Aufstellung von Anforderungen für einen Teil eines Fahrscheinautomaten.

Ein Fahrscheinautomat verkauft Zugfahrscheine. Benutzer wählen ihr Ziel aus und geben eine Kreditkarte und eine persönliche Geheimnummer ein. Der Fahrschein wird ausgegeben, und vom Konto der Kreditkarte werden die Kosten abgebucht. Drückt der Benutzer den Startkopf, dann wird neben einem Menü mit möglichen Zielorten eine Nachricht angezeigt, daß der Benutzer den Zielort auswählen soll. Wurde erst einmal ein Ziel ausgewählt, werden die Benutzer dazu aufgefordert, ihre Kreditkarte einzugeben. Deren Gültigkeit wird überprüft, und die Benutzer werden aufgefordert, die persönliche Geheimnummer einzugeben. Wenn die Transaktion überprüft ist, wird der Fahrschein ausgegeben.

(Sommerville: Aufgabe 6.3)



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Requirements Engineering

- Zur Sammlung der Anforderungen arbeiten die Entwickler mit Kunden und Endbenutzern zusammen.
- Mögliche Schwierigkeiten:
 - Die Beteiligten haben unklare Erwartungen.
 - Sie können Ihre Erwartungen nicht in Worte fassen.
 - Sie haben unrealistische Erwartungen.
 - Sie beschreiben ihre Erwartungen in einer unternehmensspezifischen Terminologie.
 - Die Erwartungen verschiedener Beteiligter widersprechen sich.

Aufgabe: Welche Stakeholder befragen Sie, um Anforderungen für eine Online-banking-website einer Bank zu sammeln?



Requirements Engineering

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Nachdem die Anforderungen gesammelt sind, werden sie
 - klassifiziert, strukturiert, gruppiert
 - priorisiert, verhandelt, Konflikte gelöst
 - dokumentiert



Copyright © 2003 United Feature Syndicate, Inc.



Requirements Engineering

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

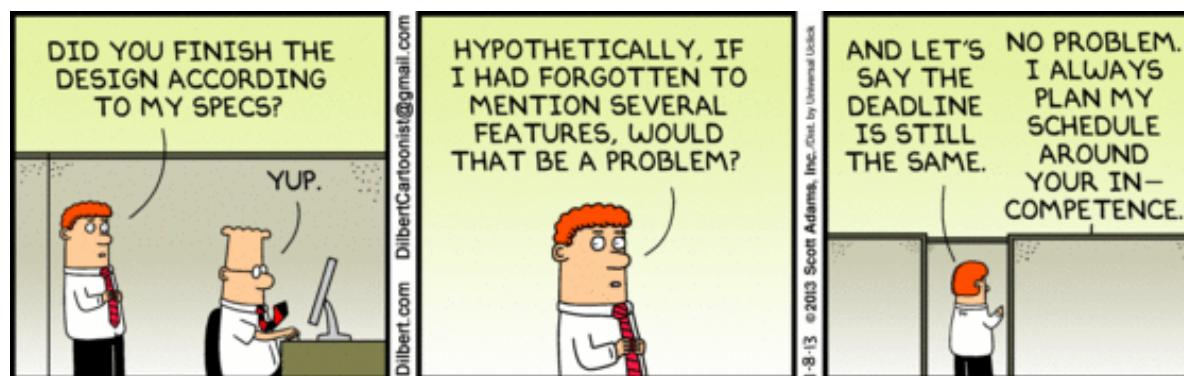
- Die Validierung der Anforderungen stellt sicher, daß sie
 - verifizierbar sind
 - verständlich sind
 - nachvollziehbar sind
 - anpassungsfähig sind, also ohne große Rückwirkungen auf andere Anforderungen geändert werden könnten.
 - widerspruchsfrei sind
- Dazu gibt es Review-Meetings mit Vertretern von Anbieter und Kunden
- Ein allererster Prototyp kann die Validierung der Anforderungen erleichtern.
- Hilfreich kann sein, bereits früh Testfälle aus den Anforderungen abzuleiten.



Pflege der Anforderungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Wenn die Anforderungen verabschiedet sind, ändern sie sich normalerweise nochmals.
- Um den Überblick nicht zu verlieren, muß es einen Prozeß geben, der beschreibt, wie über Änderungen entschieden wird und wie sie dokumentiert werden (Change Management)





Gründe für Anforderungsänderungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Art der Anforderung	Beschreibung
Veränderliche Anforderungen	Anforderungen, die sich aufgrund von Änderungen in der Umgebung verändern, in der die Organisation tätig ist. In Krankenhausssystemen könnte sich z.B. die Kostenerstattung der Krankenversicherung ändern und daher das Sammeln anderer Behandlungsinformationen erforderlich machen.
Dringende Anforderungen	Anforderungen, die zum Vorschein kommen, während sich das Verständnis des Kunden für das System bei der Systementwicklung verändert. Der Entwurfsprozess kann neue dringende Anforderungen aufdecken.
Folgeanforderungen	Anforderungen, die aus der Einführung des Computersystems resultieren. Die Einführung des Computersystems kann die Prozesse der Organisation verändern und neue Arbeitsmethoden mit sich bringen, die neue Systemanforderungen hervorrufen.
Kompatibilitätsanforderungen	Anforderungen, die von bestimmten System- oder Geschäftsprozessen innerhalb einer Organisation abhängen. Wenn diese sich ändern, können sich auch die Kompatibilitätsanforderungen der bestellten oder gelieferten Systeme entwickeln.



Spezifikation kritischer Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

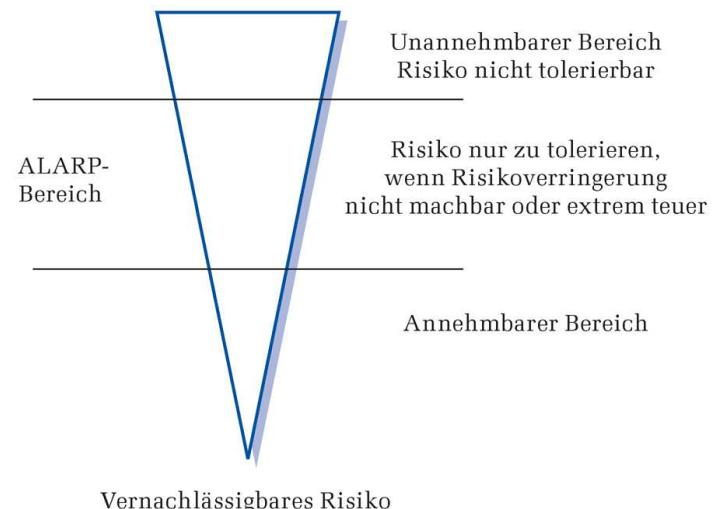
- Die besonderen Anforderungen kritischer Systeme kommen zum Ausdruck:
 - in funktionalen Anforderungen
 - Bsp.: Funktionen zur Wiederherstellung
 - in nichtfunktionalen Anforderungen, die z.B. die Verfügbarkeit beschreiben
 - darüber hinaus in abstrakten Sicherheitsanforderungen, die Sommerville als '*shall not*'-requirements bezeichnet.
 - Bsp. für ein social network: Das System soll den Benutzern nicht erlauben, mehr als 20 Profildaten anderer Benutzer pro Stunde einzusehen
 - Bsp.: für ein Kontoführungssystem einer Bank: Das System soll einem Benutzer alleine keinen Zugriff auf die Konten eigener Bankmitarbeiter gewähren (Vier-Augen-Prinzip)



Spezifikation kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Risikoklassifizierung
 - nicht tolerierbar (z.B. Ausfall eines Börsenhandelssystems für über eine Stunde)
 - as low as reasonably practical (z.B. Handelsunterbrechung für ein einzelnes Wertpapier)
 - tolerierbar (z.B. vorübergehend auf drei Sekunden erhöhte Reaktionszeit des Systems)





Spezifikation kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Maßnahmen zur Risikoreduzierung
 - Risikovermeidung: Das System wird so entworfen, daß das Risiko nicht auftreten kann
 - Risikoerkennung: Das System erkennt und beseitigt Risiken, bevor es zur Störung kommt.
 - Schadensbegrenzung: Die Folgen einer Störung werden minimiert.





Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Ein Hersteller von Fahrrädern will im Web einen Konfigurator bereitstellen, in dem seine Kunden aus vielen Optionen (Farbe, Rahmen, Lenker, Sattel, Gangschaltung, Bremsen...) wählen und dabei Ihr persönlich konfiguriertes Fahrrad zusammenstellen können. Zur Eurobike-Messe im August muß der Konfigurator online sein. Das fertige Fahrrad wird sofort als 3D-Animation dargestellt und ein Preis ermittelt. Zum Team gehören Entwickler für Java, Datenbankexperten, Vertriebsmitarbeiter, die sich mit den Märkten und Produktmanager, die sich mit den Fahrrädern auskennen. Experten für HTML5 und Javascript, Usability-Spezialisten und Web-Designer werden von einem externen Dienstleister ins Team genommen und nach Tagessatz bezahlt, da es intern keine Expertise in diesen Bereichen gibt. Die Kunden sollen mit beliebigen Endgeräten auf die Seite zugreifen können. Die Preise der Konfigurationsalternativen sollen mit einer Schnittstelle aus dem ERP-System gezogen werden, wo sie immer aktuell sind. Nach der Wahl der Konfigurationsoptionen, die für unerfahrene Surfer leicht bedienbar sein muß, soll der Anwender innerhalb von 3 Sekunden 3-D-Ansicht und Preis zu sehen bekommen. Die 3-D-Ansichten neuer Fahrradmodelle müssen in vorgegebener Auflösung und Qualität bereit stehen, bevor das neue Modell in den Konfigurator aufgenommen werden kann.

Es gibt Fachleute, die Software Engineering für Web-Applikationen für so speziell halten, daß sie unter dem Begriff *Web Engineering* ein eigenes Fachgebiet entstehen lassen wollen. Welche 10 Besonderheiten des Requirements Engineering für Web Applikationen werden in diesem Beispiel deutlich?



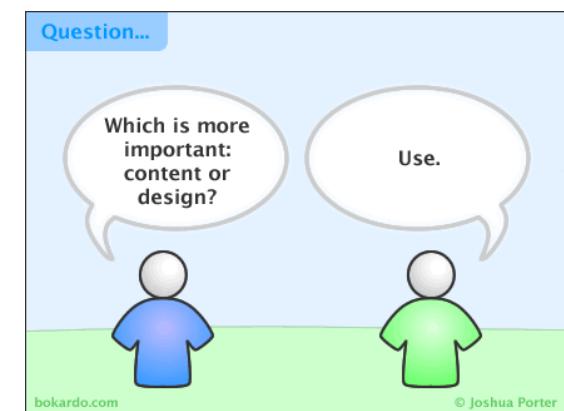
Entwurf von Software



Design von Software

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Unter dem Softwaredesign (deutsch: Entwurf) wird manchmal fälschlich ausschließlich die Gestaltung der Benutzeroberfläche verstanden.
- Die gehört zwar zum Design dazu, genauso wichtig sind aber Architektur- und Strukturentscheidungen.
- Datenstrukturen werden z.B. mit Hilfe von Klassendiagrammen und Entity-Relationship-Modellen entworfen.
- Abläufe werden mit Aktivitäten- und Sequenzdiagrammen entworfen.
- Komplexe Algorithmen kann man mit Nassi-Shneiderman-Diagrammen entwerfen, bevor man sie kodiert.





Design der Systemarchitektur

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Architekturentscheidungen bestimmen die Struktur des Systems.
- Die gewählte Systemarchitektur beeinflußt:
 - Die Performance des Systems
 - Die Angriffssicherheit (Security) des Systems
 - Die Betriebssicherheit (Safety) des Systems
 - Die Verfügbarkeit des Systems
 - Die Wartbarkeit des Systems



Architektur verteilter Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Die meisten größeren Softwaresysteme sind heute verteilte Systeme. Sie laufen auf mehreren Rechnern ab.
- Mehrere Prozesse können auf verschiedenen Rechnern gleichzeitig ablaufen.
- Software, bei der mehrere Prozesse gleichzeitig laufen, kann als verteiltes System konzipiert werden, wenn mehr als ein Prozessor vorhanden ist.





Verteilte Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Vorteil: Gut entworfene verteilte Systeme sind skalierbar.
 - Skalierbarkeit bedeutet: Sie können ohne Änderungen an der Software eine größere Anzahl an Prozessen (vor allem an Benutzern) bedienen, indem zusätzliche Hardware beschafft wird.
- Vorteil: Gut entworfene verteilte Systeme sind fehlertolerant
 - Aber: Das Netzwerk wird zum Single Point of Failure (SPOF)
- Nachteil: Verteilte Systeme sind komplexer
- Nachteil: Der Datenverkehr zwischen den beteiligten Rechnern im Netzwerk erschwert den Zugriffsschutz.



Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Ein zentrales Repository für alle beteiligten Subsysteme hat Vor- und Nachteile:
 - Keine direkte Kommunikation zwischen den Subsystemen nötig
 - Inkonsistenzen zwischen eigenen Datenbanken der Subsysteme sind ausgeschlossen.
 - Das Repository muß den Anforderungen aller Subsysteme genügen.
 - Datensicherung, Datenschutz, Datenwiederherstellung, Zugriffskontrolle sind zentralisiert.
 - wohl der wichtigste Vorteil, kann aber zum Nachteil werden, wenn die Anforderungen der Subsysteme an diese Funktionen sich stark unterscheiden.

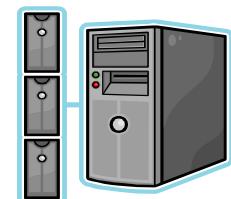




Architekturentscheidungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Client-Server-Modell:
 - Server bieten Dienste im Netzwerk an.
 - Meist zentral im Rechenzentrum
 - Clients fragen diese Dienste im Netzwerk nach.
 - Meist am Benutzerarbeitsplatz.
 - Clients kennen die Server, aber nicht die anderen Clients.
 - Mit Client und Server sind ursprünglich Prozesse der Software gemeint. Die Begriffe haben sich inzwischen auch für die Hardware durchgesetzt, auf denen die Prozesse laufen.

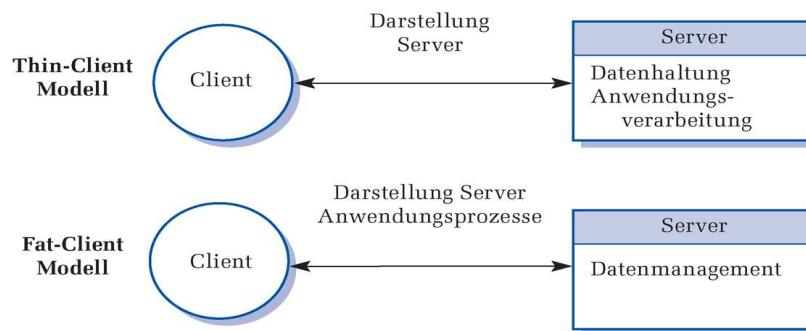




Architekturentscheidungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- 2-Tier-Client-Server-Modell: Es gibt nur zwei Stufen, Clients und Server
 - Thin-Client: Die Anwendungslogik läuft komplett auf dem Server. Der Client ist nur für Benutzereingaben und Bildschirmschirmdarstellung zuständig.
 - Fat-Client: Der Server übernimmt nur die Datenhaltung. Anwendungslogik, Benutzereingaben und Bildschirmschirmdarstellung laufen auf dem Client.

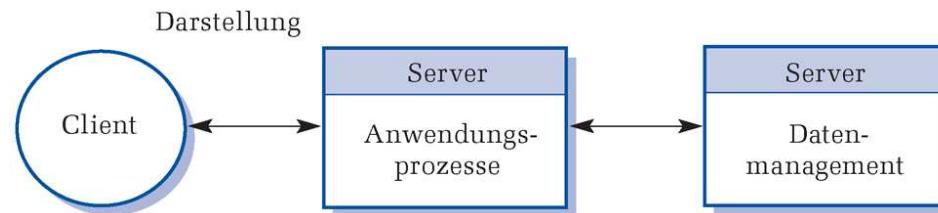




1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Architekturentscheidungen

- 3-Tier-Client-Server-Modell
 - Ein zentraler Datenbankserver sorgt für die Persistenz.
 - Ein oder mehrere Applikationsserver sind für die Applikationslogik zuständig.
 - Die Clients übernehmen Benutzereingaben und Bildschirmdarstellung.





Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Architektur	Anwendungen
Zweischichtige C/S-Architektur mit Thin-Clients	Anwendungen mit Legacy-Systemen, bei denen die Trennung von Anwendungsprozessen und Datenmanagement nicht praktikabel ist Rechenintensive Anwendungen wie zum Beispiel Compiler mit geringem oder keinem Datenmanagement Datenintensive Anwendungen (Browser oder Abfragesysteme) mit wenigen oder keinen Anwendungsprozessen
Zweischichtige C/S-Architektur mit Fat-Clients	Anwendungen, bei denen die Prozesse durch Standardanwendungen (z. B. Microsoft Excel) auf dem Client bearbeitet werden Anwendungen mit rechenintensiver Datenverarbeitung (z. B. Datenvisualisierung) Anwendungen mit relativ stabiler Endbenutzerfunktionalität, die in einer Umgebung mit gut strukturiertem Systemmanagement angeordnet sind
Drei- oder mehrschichtige C/S-Architektur	Anwendungen großen Umfangs mit Hunderten oder Tausenden von Clients Anwendungen, bei denen sowohl die Daten als auch die Anwendung selbst ständigen Veränderungen unterliegen Anwendungen, bei denen Daten aus vielfältigen Quellen verarbeitet werden



Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Webapplikationen können die Vorteile von Fat-Clients (effiziente Lastverteilung) und Thin-Clients (einfaches Systemmanagement) vereinen. Portable Code wird bei Bedarf (z.B. als Applet vom Server geladen und auf dem Client ausgeführt).
- Sie haben aber auch Nachteile: Usability und Performanz von Web-Applikationen sind tendenziell schlechter.



Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

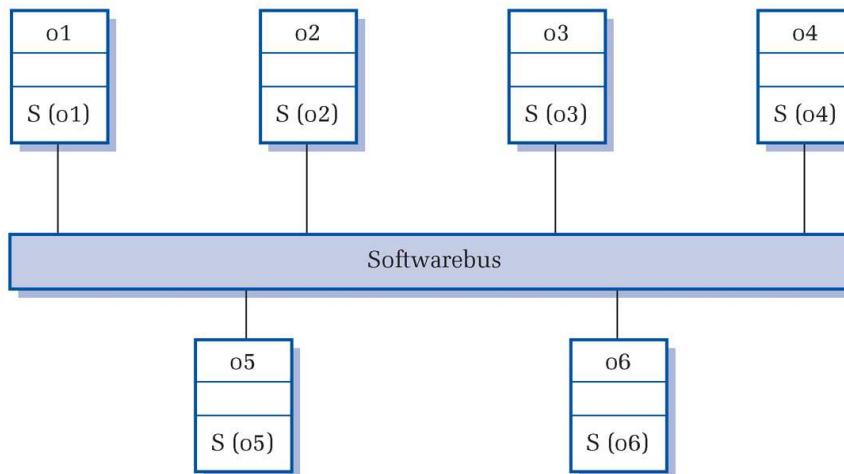
- "Distributed Objects" lösen die strikte Trennung zwischen Clients und Servern auf. Objekte bieten einander im Netzwerk Dienste an und nehmen Dienste in Anspruch.
- Vorteil: Neue Ressourcen können bei Bedarf hinzugefügt werden.
- Vorteil: Die Verteilung der Objekte im Netzwerk bleibt flexibel.
- Nachteil: Die Architektur ist durch größtmögliche Komplexität unübersichtlich und fehleranfällig.



Architekturentscheidungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Verteilte Objekte benötigen einen Object Request Broker (ORB)
- Dafür gibt es Standards wie CORBA.





Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Peer-to-peer-Systeme (P2P) sind dezentral organisiert.
Die Rechenleistung kann auf jedem Knoten im Netzwerk ausgeführt werden.
- Die Protokolle für die Kommunikation zwischen den Knoten sind in einer Anwendung implementiert, die auf jedem Knoten installiert sein muß.
- Jeder Knoten ist auch ein Switch, der Daten von einem Knoten zu einem andern weiterleitet.
- Vorteil: Hochgradig redundant, tolerant gegenüber dem Ausfall einzelner Knoten.
- Nachteil: Extrem ineffizient wegen mehrfacher Kommunikation und paralleler Suche von verschiedenen Knoten aus.

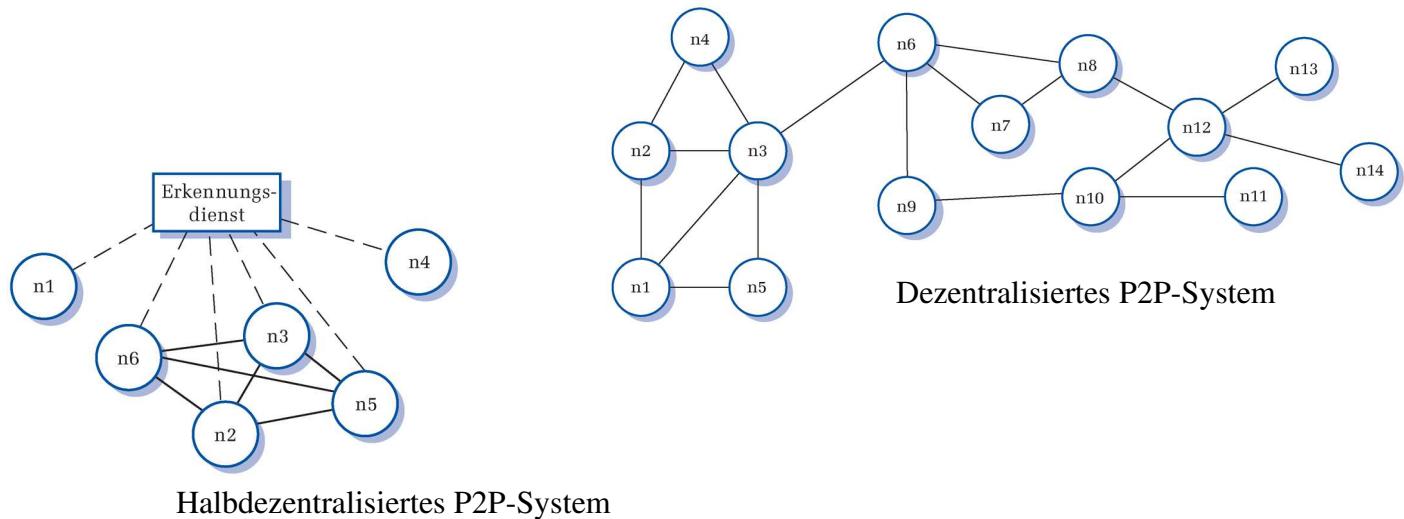


Architekturentscheidungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Halbdezentralisierte P2P-Systeme verwenden einen Server nur für die Kontaktaufnahme.
- Anschließend kommunizieren die Knoten selbständig miteinander.

Nennen Sie je zwei Vor- und Nachteile dezentralisierter und halbdezentralisierter P2P-Netze.





Architekturentscheidungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Zentrale Steuerung: Ein Subsystem kann alle anderen aufrufen, starten und stoppen.
- Ereignisbasierte Steuerung: Jedes Subsystem reagiert auf Ereignisse, die von anderen Subsystemen oder von außerhalb des Systems ausgelöst werden.





Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Welche Architektur ist besonders geeignet, wenn folgende Anforderung besonders wichtig ist:

- a) Performance,
- b) Security,
- c) Safety,
- d) Verfügbarkeit,
- e) Wartbarkeit, Erweiterbarkeit?

Welche Zielkonflikte gibt es?



Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Welche Architektur ist in folgenden Fällen besonders geeignet?

Ein System von Fahrscheinautomaten auf Bahnhöfen der Deutschen Bahn

Fahrscheinautomaten an Straßenbahnhaltestellen eines regionalen Verkehrsunternehmens

Ein Videokonferenzsystem



Typen von Softwaresystemen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Bestimmte Architekturen eignen sich besonders für bestimmte Typen von Softwaresystemen, die im folgenden unterschieden werden.

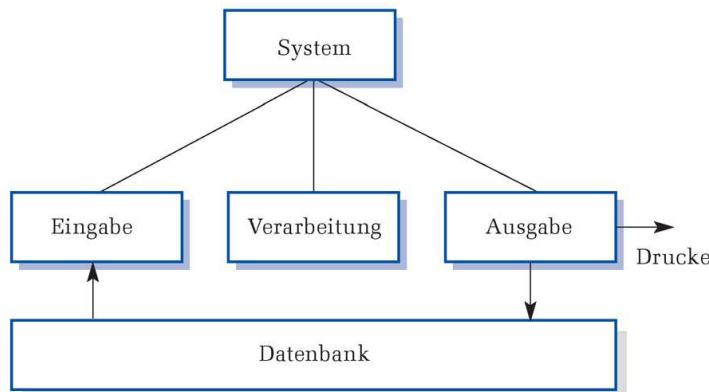
- Ereignisverarbeitende Systeme reagieren ständig und sofort auf Ereignisse wie Benutzereingaben.
 - Dazu gehören Echtzeitsysteme, aber auch Grafik- und Officeprogramme.
 - Sie sind meist Einbenutzersysteme.
 - Sie müssen schnell auf Benutzerinteraktionen reagieren.
 - Sie bearbeiten Daten längere Zeit im Hauptspeicher.
 - Architektur: Mehrere autonome Objekte stehen miteinander in Kommunikationsbeziehungen. Sie verarbeiten Ereignisse und interpretieren Befehle.



Typen von Softwaresystemen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Batchverarbeitende Systeme verarbeiten Daten am Stück ohne Eingriff des Benutzers.
 - Sie eignen sich, wenn auf eine große Menge von Daten immer ähnliche Operationen angewandt werden müssen.
 - Bsp.: Täglicher Rechnungsdruck im ERP
 - Architektur: Sie bestehen meist aus Komponenten für Eingabe, Verarbeitung und Ausgabe.

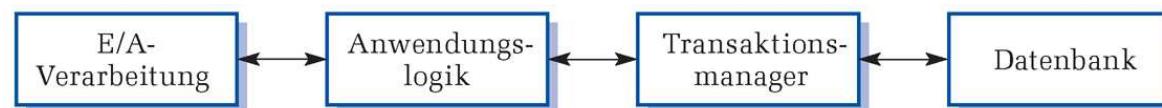




Typen von Softwaresystemen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Transaktionsverarbeitende Systeme verarbeiten Schreib- und Lesezugriffe auf einen großen Datenbestand.
 - Sie müssen sicherstellen, daß die Benutzer sich nicht gegenseitig behindern und daß jede Transaktion atomar ist (also ganz oder gar nicht, aber unter keinen Umständen teilweise durchgeführt wird)
 - Bsp.: Auftragserfassung im ERP
 - Architektur: Transaktionsverarbeitende Systeme bestehen oft aus Komponenten für Benutzerinteraktion, Anwendungslogik und Datenhaltung.

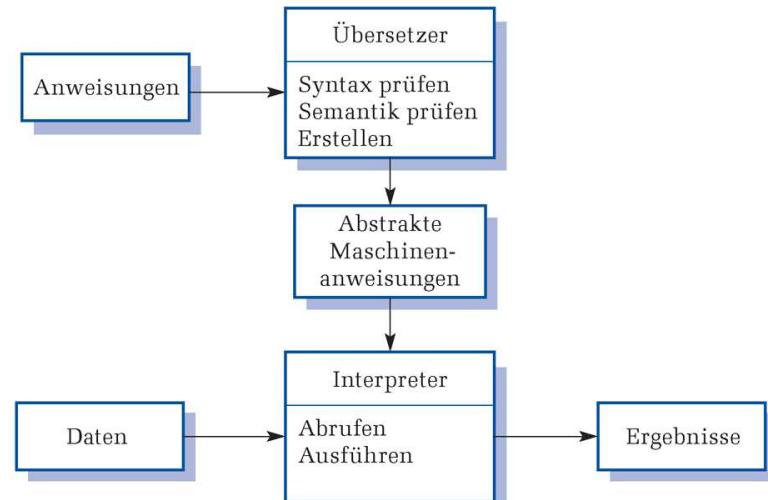




Typen von Softwaresystemen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Sprachverarbeitende Systeme arbeiten Anweisungen ab, die in einer formalen Sprache formuliert sind.
 - Dazu gehören z.B. Compiler und Skriptinterpreter für Makros.
 - Architektur: Verschiedene Komponenten übernehmen das Prüfen, Auswerten und Ausführen von Anweisungen.





Aufgabe

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Ordnen Sie die folgenden Softwaresysteme den vier vorgestellten Typen zu:

- ein Kassensystem in einem Supermarkt
- ein System, das Mahnungen über zu zahlende Zeitschriftenabonnementsgebühren druckt.
- ein Fotoalbum, das Funktionen zur Wiederherstellung alter Fotografien bietet.
- ein System, das sehbehinderten Benutzern Webseiten vorliest.
- ein interaktives Spiel, in dem Figuren sich bewegen, Hindernisse überwinden und Schätze sammeln
- ein Bestandsaufnahmesystem, das verfolgt, welche Waren auf Lager sind und automatisch Bestellungen für neue Waren ausgibt, wenn der Vorrat unter einen bestimmten Wert fällt.

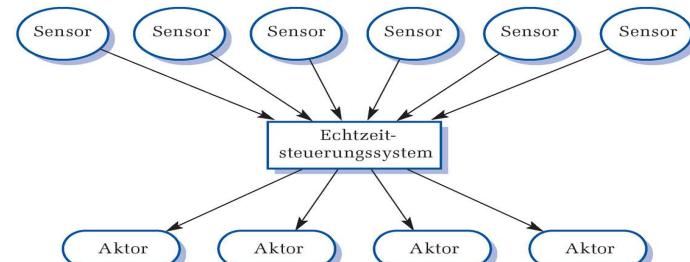


Echtzeitsysteme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

An manche Softwaresysteme werden besondere zeitliche Anforderungen gestellt. Ihr Design muß dafür geeignet sein.

- Die korrekte Funktionsweise eines Echtzeitssystems hängt nicht nur von seinen Ergebnissen ab, sondern auch von der Zeit, in der diese Ergebnisse erzeugt werden.
 - Unmittelbar nach Empfang eines Stimulus von einem Sensor muß eine geeignete Routine aufgerufen werden.
 - Dies erfolgt üblicherweise mit einer Menge nebenläufiger, miteinander kooperierender Prozesse.

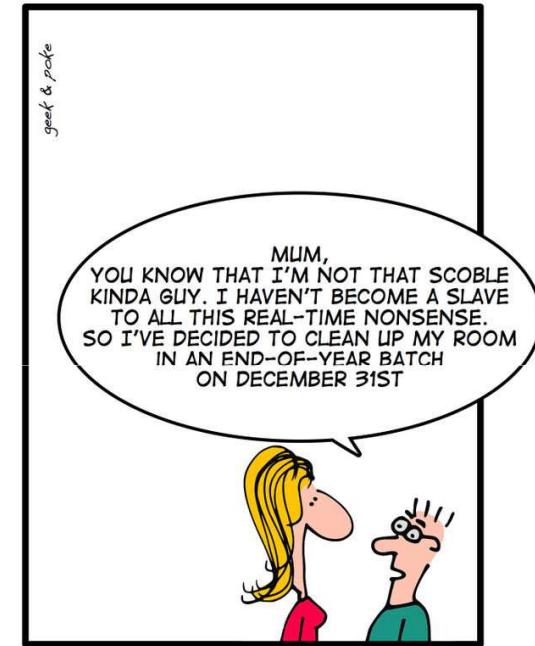




Echtzeitsysteme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- "soft real-time": Die Funktionsweise ist minderwertig, wenn die Ergebnisse nicht in der spezifizierten Zeit vorliegen.
- "hard real-time": Die Funktionsweise ist gestört, wenn die Ergebnisse nicht in der spezifizierten Zeit vorliegen.



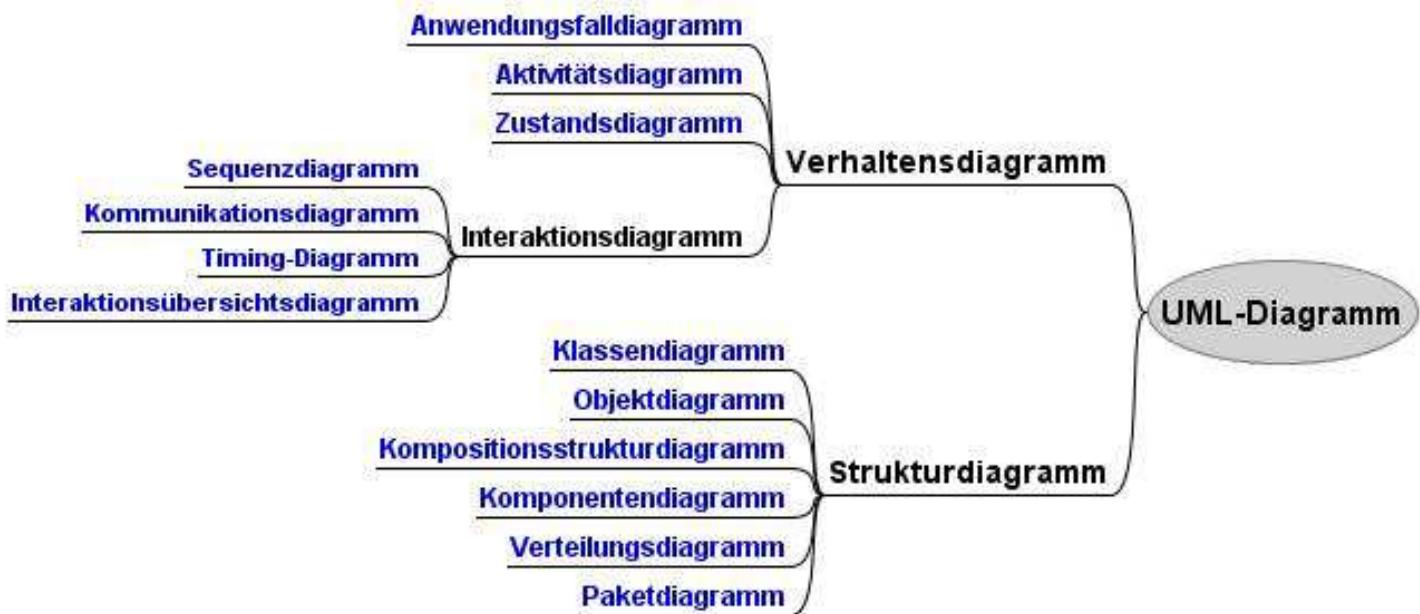
BATCH VS. REAL-TIME



Wiederholung: OOD

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Nach einer grundsätzlichen Entscheidung über die Architektur einer Anwendung erfolgt das objektorientierte Design mit Hilfe der bekannten UML-Modelle.





Benutzeroberflächen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Eine gute GUI ist entscheidend für die Zuverlässigkeit des Systems. Viele "Benutzerfehler" könnten durch eine der Zielgruppe angemessene Benutzerführung vermieden werden.
- Im Web kommt hinzu: Benutzer, die sich nicht zurechtfinden, wandern sofort zur Konkurrenz ab.
 - Bsp.: Untersuchung von Reiseportalen durch DISQ: Bei einer Suchanfrage nach einer Reise werden oft alle passenden Angebote aufgelistet – unabhängig von ihrer Verfügbarkeit. Der Kunde muß aufwendig jedes Angebot einzeln anklicken, um zu sehen, ob es überhaupt buchbar ist.



Benutzeroberflächen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Parallel zur Erstellung der Use-Case-Dokumente werden Screen Prototypen zu Papier gebracht. Sie erleichtern die Diskussion mit den Anwendern und helfen, die Anforderungen zu verstehen.
- Parallel zu Design und Implementierung entstehen immer funktionsfähigere ausführbare Prototypen der GUI.
- In der Testphase wird im Rahmen des User Acceptance Test der Testschwerpunkt auf die Usability gesetzt.



Grundregeln für den GUI-Entwurf

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Prinzip	Beschreibung
Benutzervertrautheit	Die Oberfläche sollte Bezeichnungen und Begriffe aus der Erfahrungswelt der Menschen verwenden, die am meisten vom System Gebrauch machen.
Konsistenz	Die Oberfläche sollte, wo immer möglich, konsistent sein. Das bedeutet, dass vergleichbare Operationen auf dieselbe Weise veranlasst werden sollten.
Minimale Überraschung	Benutzer sollten vom Systemverhalten nie überrascht werden.
Wiederherstellbarkeit	Die Oberfläche sollte Wiederherstellungsmechanismen für den Benutzer enthalten.
Benutzerführung	Die Oberfläche sollte beim Auftreten von Fehlern aussagekräftige Rückmeldungen und kontextsensitive Hilfsmittel für Benutzer anbieten.
Benutzervielfalt	Die Oberfläche sollte für verschiedene Arten von Systembenutzern geeignete Interaktionsmöglichkeiten bereitstellen.

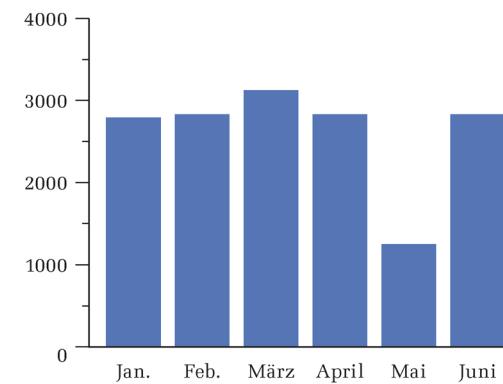


- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Grundregeln für den GUI-Entwurf

- Text benötigt weniger Bildschirmfläche als eine graphische Darstellung, kann aber nicht auf einen Blick erfaßt werden.
- Text ist vorteilhaft, wenn es auf Genauigkeit ankommt und sich die Werte nur langsam verändern. Sich ständig verändernde Textanzeigen verwirren.

	Jan.	Feb.	März	April	Mai	Juni
	2842	2851	3164	2789	1273	2835

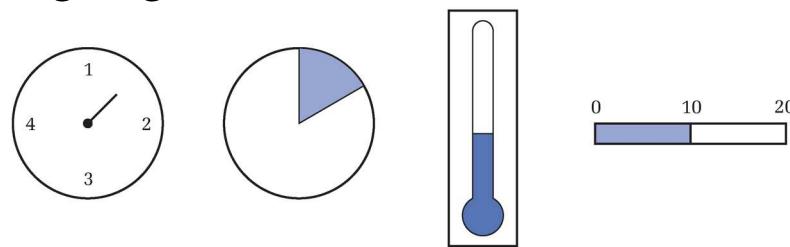




Grundregeln für den GUI-Entwurf

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Graphische Darstellungen sind ungenauer, aber übersichtlicher.
- Numerische Informationen, die sich ständig verändern, werden besser analog abgebildet.



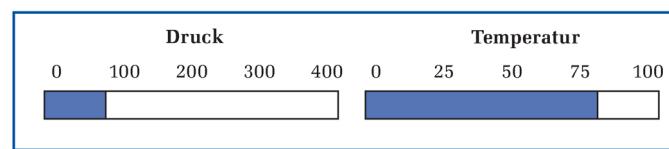
Skala mit Zeiger

Tortendiagramm

Thermometer

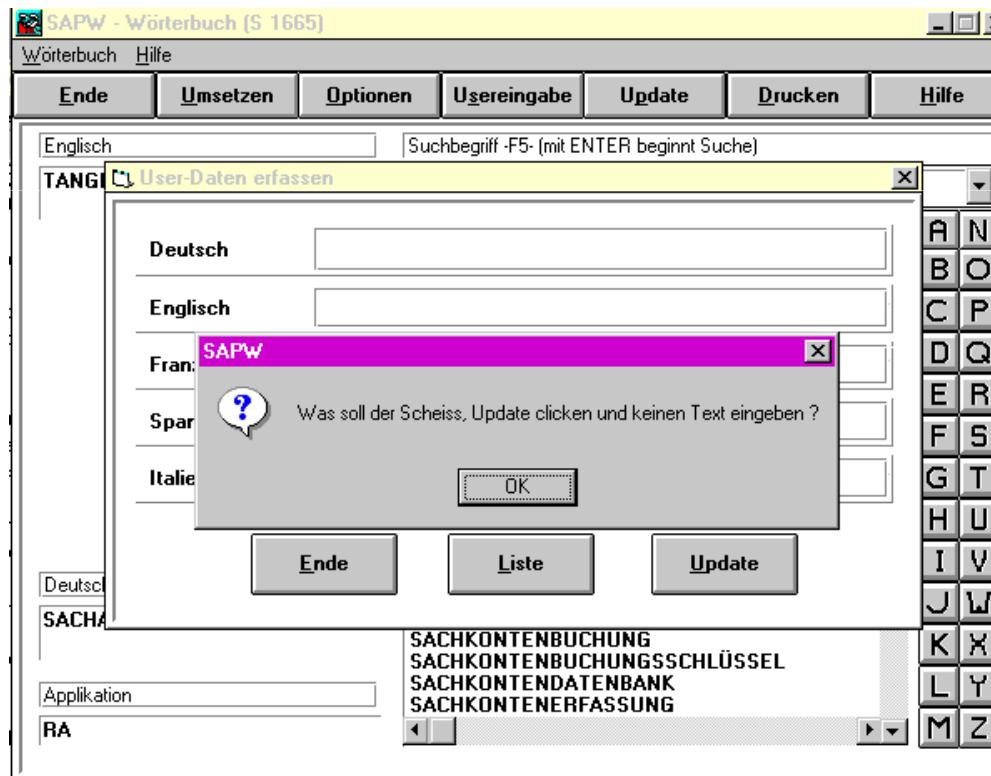
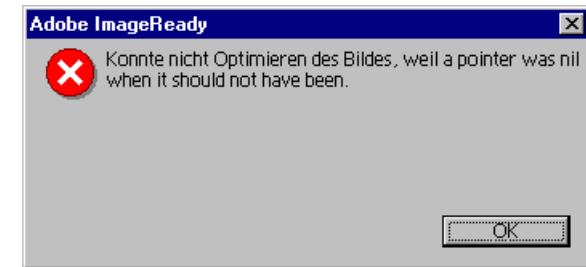
Horizontaler Balken

- Relative Informationen werden ebenfalls besser analog abgebildet.



Grundregeln für den GUI-Entwurf: Systemmeldungen

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung





Grundregeln für den GUI-Entwurf: Systemmeldungen

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

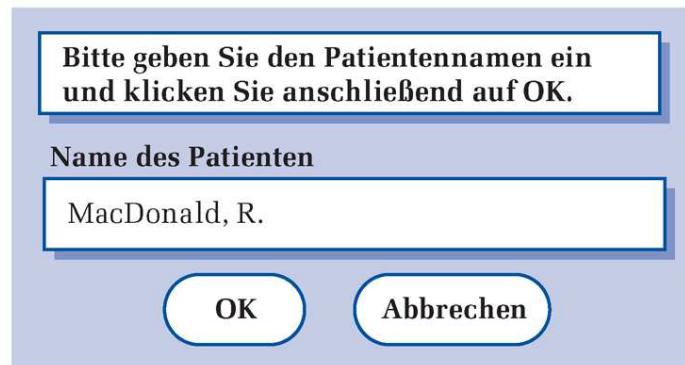
Grundsatz	Beschreibung
Kontext	Die vom System generierte Meldung sollte nach Möglichkeit den aktuellen Benutzerkontext widerspiegeln. Soweit möglich, sollte das System wissen, was der Benutzer gerade tut, und Meldungen ausgeben, die für die derzeitige Tätigkeit von Bedeutung sind.
Erfahrung	Wenn Benutzer mit einem System vertraut werden, werden sie durch lange, „bedeutungsvolle“ Meldungen irritiert. Anfänger finden es jedoch schwierig, kurze und knappe Aussagen zum Problem zu verstehen. Das Benutzerführungssystem sollte beide Meldungsarten anbieten und es dem Benutzer gestatten, die Genauigkeit von Meldungen zu steuern.
Fähigkeiten	Meldungen sollten sowohl auf die Fähigkeiten als auch auf die Erfahrung des Benutzers zugeschnitten sein. Meldungen für verschiedene Benutzerklassen sollten, abhängig von der den Lesern vertrauten Terminologie, unterschiedlich formuliert werden.
Stil	Meldungen sollten eher positiv als negativ ausgedrückt werden und lieber eine aktive Anrede als eine passive enthalten. Sie sollten nie verletzend oder komisch sein.
Kultur	Wann immer möglich, sollte der Gestalter von Meldungen mit der Kultur des Landes vertraut sein, in dem das System verkauft wird. Zwischen Europa, Asien und Amerika bestehen ausgeprägte kulturelle Unterschiede. Eine in einer Kultur angemessene Meldung kann in einer anderen unzumutbar sein.



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Grundregeln für den GUI-Entwurf: Systemmeldungen

Beispiel Arztpraxis



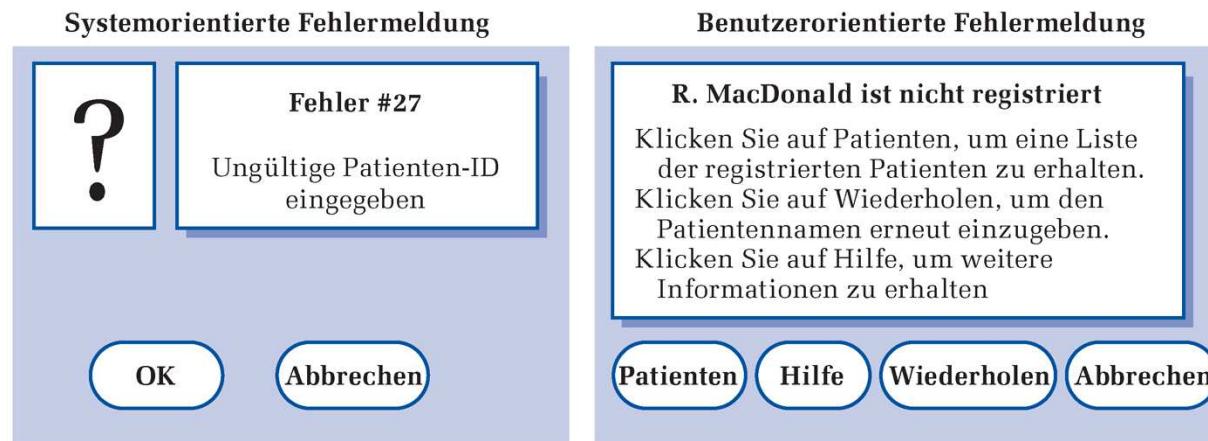


1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Grundregeln für den GUI-Entwurf: Systemmeldungen

Schlechte Fehlermeldung

- Der Benutzer wird beschuldigt, einen Fehler gemacht zu haben
- nicht auf die Fähigkeiten des Benutzers zugeschnitten
- Kein Vorschlag, wie das Problem behoben werden kann
- Systemspezifische Terminologie

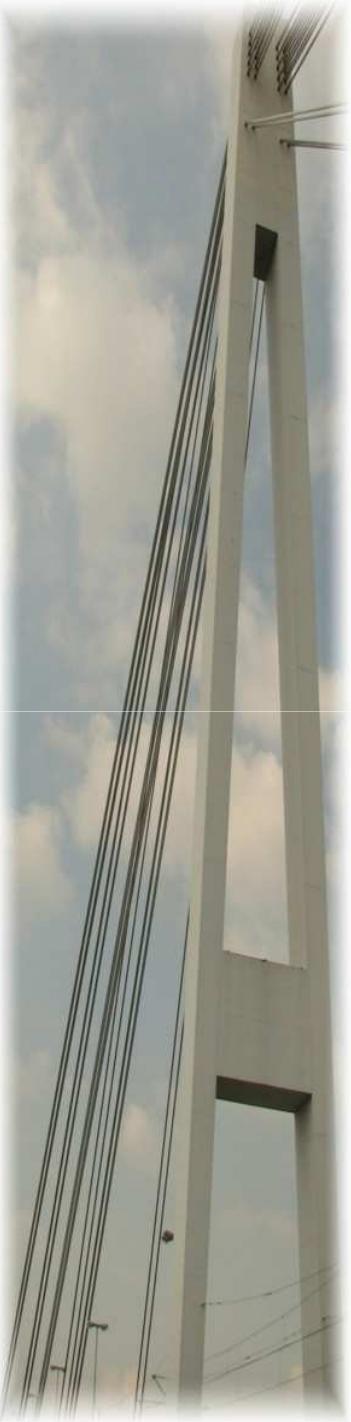




Benutzerinteraktion

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Art der Interaktion	Wichtigste Vorteile	Wichtigste Nachteile	Anwendungsbeispiele
Direkte Manipulation	Schnelle und intuitive Interaktion Leicht zu erlernen	Möglicherweise schwierig zu implementieren Nur angemessen, wenn es optische Metaphern und Aufgaben und Objekte gibt	Videospiele CAD-Systeme
Menüauswahl	Verhindert Benutzerfehler Wenig Tippen erforderlich	Für erfahrene Benutzer zu langsam Kann zu komplex werden, wenn viele Menüoptionen vorhanden sind	Die meisten allgemein eingesetzten Systeme
Ausfüllen einer Eingabemaske	Einfache Dateneingabe Leicht zu erlernen Überprüfbar	Benötigt viel Platz auf dem Bildschirm Führt zu Problemen, wenn die Benutzeroptionen nicht mit den Formularfeldern übereinstimmen	Bestandskontrolle Verarbeitung von persönlichen Krediten
Befehlssprache	Leistungsfähig und flexibel	Schwer zu erlernen Schwaches Fehlermanagement	Betriebssysteme Überwachungs- und Steuerungssysteme
Natürliche Sprache	Zugänglich für Gelegenheitsnutzer Leicht erweiterbar	Erfordert mehr Tippen Systeme, die natürliche Sprache verstehen, sind unzuverlässig	Systeme zum Abrufen von Informationen



Entwicklung von Software



Softwareentwicklung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Auf Grundlage des Entwurfs wird ein Softwaresystem in einer Programmiersprache entwickelt.
- Die Programmierung selbst ist nicht Gegenstand dieser Vorlesung.
- Im folgenden werden einige spezielle Aspekte der Softwareentwicklung behandelt:
 - Wiederverwendung von Software
 - Komponentenbasiertes Software Engineering
 - Besonderheiten bei der Entwicklung kritischer Systeme
 - Weiterentwicklung bestehender Software



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Ziele der Softwarewiederverwendung:
 - Herstellkosten verringern
 - Wartungskosten verringern
 - Projektdauer verkürzen
 - Qualität erhöhen
- Wiederverwendet werden können
 - ganze Anwendungssysteme
 - Komponenten
 - einzelne Klassen



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Vorteile	Erklärung
Höhere Zuverlässigkeit	Wiederverwendete Komponenten, die in der Praxis getestet wurden, sollten zuverlässiger als neue Komponenten sein, denn Fehler im Entwurf und der Implementierung sind bereits aufgespürt und beseitigt worden.
Geringeres Risiko	Die Kosten für bestehende Software sind bekannt, während die Kosten für die Entwicklung stets geschätzt werden müssen. Dies ist wichtig für das Projektmanagement, da Unsicherheiten bei der Projektkostenschätzung verringert werden. Das gilt besonders dann, wenn relativ große Softwarekomponenten, zum Beispiel Subsysteme, wiederverwendet werden.
Effektiver Einsatz von Spezialisten	Anwendungsspezialisten können wieder verwendbare Komponenten entwerfen, die ihr Wissen enthalten, anstatt ständig die gleiche Arbeit zu verrichten.
Übereinstimmung mit Standards	Einige Standardlösungen, beispielsweise für Bedienoberflächen, können als Menge von wieder verwendbaren Standardkomponenten implementiert werden. Wenn zum Beispiel in einer Benutzeroberfläche Menüs durch wieder verwendbare Komponenten implementiert werden, weisen alle Anwendungen dasselbe Menüformat auf. Die Verwendung standardisierter Bedienoberflächen verbessert die Zuverlässigkeit, da Benutzer bedeutend weniger Fehler machen, wenn sie mit einer vertrauten Oberfläche zu tun haben.
Beschleunigte Entwicklung	Die frühest mögliche Markteinführung eines Systems ist oftmals wichtiger als die Gesamtkosten der Entwicklung. Die Wiederverwendung von Software kann die Systemherstellung beschleunigen, da sowohl die Entwicklungszeit als auch die Zeit für die Validierung verkürzt werden.



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Problem	Erklärung
Höhere Wartungskosten	Ist der Quellcode eines wieder verwendeten Softwaresystems oder einer Komponente nicht verfügbar, erhöhen sich möglicherweise die Wartungskosten, da die wieder verwendeten Elemente in zunehmendem Maße inkompatibel mit Systemänderungen werden.
Mangel an Unterstützung durch Werkzeuge	CASE-Werkzeugsammlungen unterstützen die Entwicklung mit Wiederverwendung meist nicht. Es kann schwierig oder sogar unmöglich sein, diese Werkzeuge auf ein Komponentenbibliothekssystem abzustimmen. Der bei diesen Werkzeugen vorausgesetzte Softwareprozess schließt die Wiederverwendung nicht ein.
Das „Nicht-hier-erfunden“-Syndrom	Einige Softwareentwickler ziehen es vor, Komponenten umzuschreiben, da sie glauben, sie dadurch verbessern zu können. Dies hängt einerseits mit Vertrauen und andererseits mit der Tatsache zusammen, dass das Schreiben neuer Software als größere Herausforderung angesehen wird als die Wiederverwendung vorhandener Software.
Aufbau und Wartung einer Komponentenbibliothek	Eine Komponentenbibliothek zu füllen und sicherzustellen, dass Softwareentwickler diese Bibliothek verwenden können, kann sehr kostspielig sein. Die gegenwärtigen Techniken zur Klassifizierung, Katalogisierung und zum Abrufen von Softwarekomponenten sind unausgereift.
Auffinden, Verstehen und Anpassen wieder verwendbarer Komponenten	Softwarekomponenten müssen in einer Bibliothek gefunden, verstanden und manchmal an die Arbeit in einer neuen Umgebung angepasst werden. Die Softwareentwickler müssen ausreichend davon überzeugt sein, dass sie passende Komponenten in der Bibliothek finden, ehe sie die Suche nach Komponenten in ihren normalen Entwicklungsprozess aufnehmen.



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Kriterien für die Entscheidung über Wiederverwendung bestehender Software
 - Entwicklungszeit der Software: Bestehende Systeme integrieren geht meist schneller als neue zu schreiben
 - Nutzungsdauer der Software: Langfristig die Wartung zu garantieren ist schwierig, wenn man den Sourcecode der verwendeten Komponenten nicht hat.
 - Erfahrungen und Kenntnisse des Entwicklungsteams
 - Kompromißbereitschaft der Kunden: Werden funktionale Abweichungen von den Anforderungen in Kauf genommen, um Wiederverwendung zu ermöglichen?



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Wiederverwendung

Ansatz	Beschreibung
Entwurfsmuster	Allgemeine, anwendungsübergreifende Abstraktionen werden als Entwurfsmuster dargestellt und zeigen abstrakte und konkrete Objekte und Interaktionen.
Komponentenbasierte Entwicklung	Systeme werden durch die Zusammenstellung von Komponenten (Mengen von Objekten) entwickelt, die den Standards des Komponentenmodells entsprechen. Dies wird in Kapitel 19 behandelt.
Anwendungsrahmen	Zusammenstellungen von abstrakten und konkreten Klassen, die angepasst und erweitert werden können, um Anwendungssysteme aufzubauen.
Wrapper um Legacy-Systeme	Legacy-Systeme (siehe Kapitel 2) können in einen „Wrapper“ eingeschlossen werden, indem ein Satz von Schnittstellen definiert wird. Der Zugriff auf diese Systeme erfolgt dann über diese Schnittstellen.
Dienstorientierte Systeme	Systeme werden durch die Verknüpfung mit gemeinsam genutzten Diensten entwickelt, die von externen Anbietern bereitgestellt sein können.
Anwendungsproduktlinien	Ein Anwendungstyp wird mit Hilfe einer gemeinsamen Architektur verallgemeinert, so dass er an verschiedene Kunden angepasst werden kann.
COTS-Integration:	Systeme werden durch die Integration bestehender Anwendungssysteme entwickelt.
Konfigurierbare vertikale Anwendungen	Ein generisches System wird so entworfen, dass es an die Bedürfnisse eines Kunden angepasst werden kann.
Programmbibliotheken	Klassen- und Funktionsbibliotheken, die allgemein verwendete Abstraktionen implementieren, stehen für die Wiederverwendung zur Verfügung.
Programmgeneratoren	Ein Generatorsystem nimmt die Kenntnisse über eine bestimmte Art von Anwendung auf und erstellt Systeme oder Systemfragmente für diesen Anwendungsbereich.
Aspektorientierte Softwareentwicklung	Gemeinsam genutzte Komponenten werden bei der Kompilierung des Programms in verschiedene Stellen „eingewoben“.

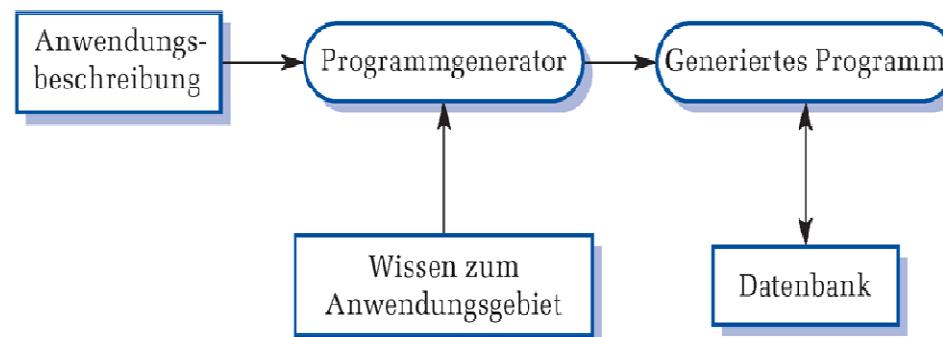


Wiederverwendung

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Beispiel Code-Generatoren:

- In CASE-Tools werden Programmfragmente anhand von Designs generiert.
- In betriebswirtschaftlichen Anwendungssystemen werden Reports anhand von Benutzereingaben generiert





Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Beispiel: Integration von Standardsoftware
- Viele Standardsoftwaresysteme haben definierte Schnittstellen (API)
- Umfangreiche Softwaresysteme lassen sich daher auch aufbauen, indem man mehrere Standardsysteme miteinander verbindet.



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgabe

Erklären Sie, warum die Kosteneinsparungen durch Wiederverwendung vorhandener Software nicht einfach proportional zur Größe der wiederverwendeten Komponenten sind.



Wiederverwendung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgabe

Nennen Sie Situationen, in denen Sie die Wiederverwendung von Software nicht empfehlen würden.



Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Prof. Dr. Carsten Dorrhauer

- Die Verwendung von Standardsoftware scheitert oft, weil die Konfiguration nicht flexibel genug auf die Anforderungen anpaßbar ist.
- Die Wiederverwendung einzelner Klassen hat sich überhaupt noch nie durchsetzen können. Die Klassen wären zwar flexibel, aber ihr Einsatz sehr komplex.
- Ein Kompromiß ist die Wiederverwendung von Komponenten mittlerer Größe.
- Sie werden unabhängig voneinander konzipiert und sollen zu einem Anwendungssystem integriert werden können.





Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Komponenten sind voneinander unabhängig. Ihre Schnittstelle ist von ihrer Implementierung getrennt.
- Ein Komponentenmodell legt die Kommunikation zwischen den Komponenten fest und wie sie ihre Schnittstellen definieren können.
 - Die wichtigsten: COM, CORBA, EJB
 - Middleware steuert die Kommunikation der Komponenten.
 - Im Entwicklungsprozeß verändert sich im Vergleich zur Erstellung reiner Individualsoftware.



Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

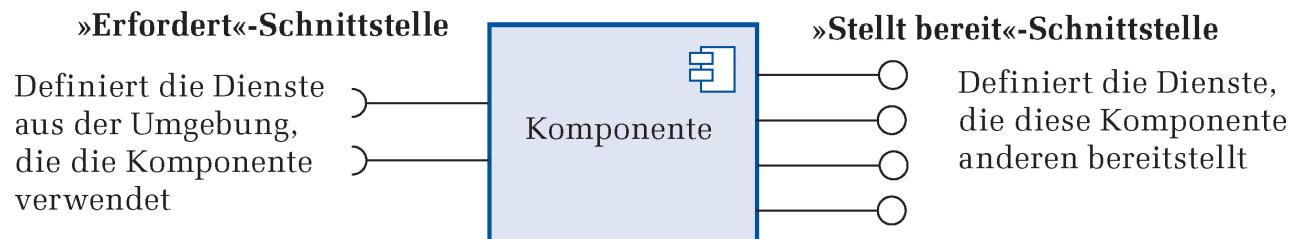
Komponenteneigenschaft	Beschreibung
Standardisiert	Standardisierung von Komponenten bedeutet, dass eine in einem komponentenbasierten Softwareprozess eingesetzte Komponente einem standardisierten Komponentenmodell entsprechen muss. Durch das Modell können Komponentenschnittstellen, Metadaten, Dokumentationen, Zusammenstellungsart sowie Bereitstellung vorgegeben werden.
Unabhängig	Eine Komponente sollte unabhängig sein – sie sollte sich zusammen- und bereitstellen lassen, ohne auf andere Komponenten angewiesen zu sein. Wenn die Komponente auf extern verfügbare Dienste angewiesen ist, sollten diese in der Spezifikation einer Benötigt-Schnittstelle ausdrücklich genannt werden.
Zusammenstellbar	Damit eine Komponente zusammenstellbar ist, müssen alle Interaktionen mit der Außenwelt über öffentlich definierte Schnittstellen ablaufen. Zusätzlich muss sie externen Zugriff auf eigene Informationen erlauben, wie z.B. über ihre Methoden und Attribute.
Einsetzbar	Um einsetzbar zu sein, muss eine Komponente in sich geschlossen und in der Lage sein, als eigenständige Entität auf einer Komponentenplattform zu funktionieren, die das Komponentenmodell implementiert. Das bedeutet im Normalfall, dass die Komponente im Binärformat vorliegt und vor der Bereitstellung nicht mehr kompiliert werden muss.
Dokumentiert	Komponenten müssen vollständig dokumentiert sein, damit potenzielle Benutzer entscheiden können, ob sie ihren Ansprüchen genügen. Die Syntax und im Idealfall auch die Semantik aller Komponentenschnittstellen müssen angegeben werden.



Komponentenbasiertes SE

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Das UML-Komponentendiagramm zeigt die von einer Komponente benötigten und die von einer Komponente angebotenen Schnittstellen.





Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Komponenten vs. Klassen

- Komponenten werden nicht im Rahmen einer Anwendung kompiliert, sondern sie liegen kompiliert vor und werden dann auf einer Komponentenplattform installiert.
- Komponenten definieren keine Objekte.
- Komponenten sind eine Black-Box. Sie sind ausschließlich über ihre Schnittstellen ansprechbar.
- Komponenten sind auf manchen Komponentenplattformen sprachunabhängig.
- Komponenten sind standardisiert, sie müssen gemäß einem Komponentenmodell entwickelt werden.
- Komponenten implementieren i.d.R. eine größere Funktionalität als einzelne Klassen.

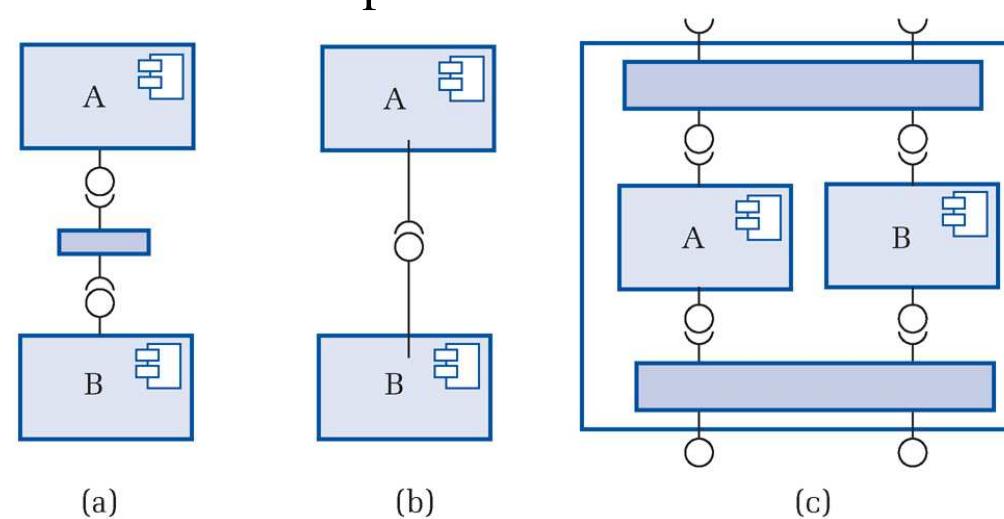


Komponentenbasiertes SE

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Zusammenstellung von Komponenten

- a) sequentiell: In der zusammengesetzten Komponente werden die Teilkomponenten nacheinander ausgeführt.
- b) hierarchisch: Eine Komponente ruft die Dienste einer anderen auf.
- c) hinzufügend: Eine neue Komponente erhält die Schnittstellen aller Teilkomponenten





Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Inkompatibilitäten zwischen Komponenten

- Parameter haben die selbe Bezeichnung, unterscheiden sich aber in Anzahl, Reihenfolge oder Datentyp.
- Operationen unterscheiden sich in Ihrer Bezeichnung.
- Operationen der einen Komponente werden nur zum Teil von der anderen angeboten oder nachgefragt.



Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Komponenten in der Praxis

- Ein freier Markt für Komponenten entwickelt sich kaum.
- Wenn es Wiederverwendung von Komponenten gibt, dann vor allem unternehmensintern.
(Bsp.: Login, Suchfunktion, Sicherheitsfunktionen, Event-Logging für alle Webapplikationen eines Unternehmens)



Komponentenbasiertes SE

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgabe

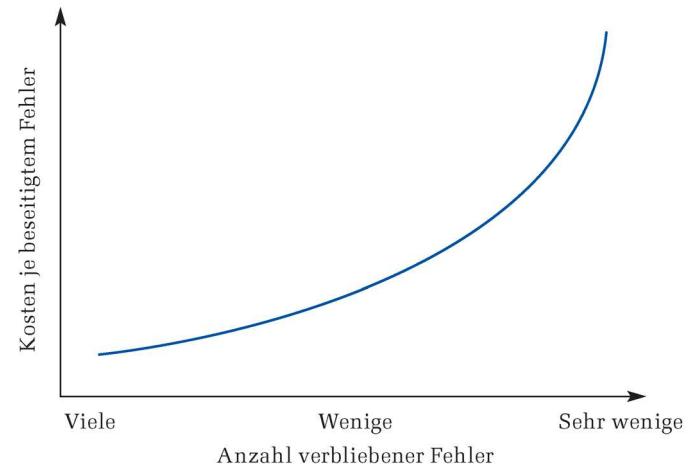
- Warum ist entscheidend, daß alle Interaktionen zwischen Komponenten über *benötigt-* und *stellt-bereit*-Schnittstellen definiert werden?



Entwicklung kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Jeder Softwareentwicklungsprozeß hat das Ziel, Fehler zu vermeiden. Fehler vollständig auszuschließen, ist aber unmöglich.
- Die Kosten der Fehlerentdeckung und Fehlervermeidung steigen dabei exponentiell mit dem Grad der Fehlerfreiheit. Es muß also immer eine Abwägung zwischen unentdeckten Fehlern und Kosten geben.
- Bei kritischen Systemen werden weit höhere Kosten in Kauf genommen.





Entwicklung kritischer Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Prinzipien verlässlicher Entwicklung

- Bei kritischen Systemen spielen gute Grundsätze objektorientierter Entwicklung natürlich eine besonders wichtige Rolle: getter/setter-Methoden, keine ungefangenen Exceptions,...
- Darüber hinaus sollte man fehleranfällige Konstrukte grundsätzlich vermeiden, wenn nicht zwingende Gründe ihre Verwendung erfordern:
 - Fließkommazahlen
 - Pointer
 - Dynamische Speicherallokation
 - Nebenläufigkeit
 - Unbeschränkte Arrays



Entwicklung kritischer Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Defensive programming:

- In den Code werden Überprüfungen aufgenommen, die eigentlich für das Funktionieren des Algorithmus nicht nötig wären.
- Fehler in der Spezifikation werden damit nicht entdeckt.





Entwicklung kritischer Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Zur Fehlervermeidung trägt *Redundanz* bei. Mehrere identische Komponenten stehen zur Verfügung. Wenn eine ausfällt, ist sind die anderen noch verfügbar.
- Weil identische Komponenten auch identische Fehler aufweisen können, ist in manchen Fällen *Diversität* vorzuziehen. Mehrere unterschiedliche Komponenten erfüllen den gleichen Zweck.
- Redundanz und Diversität erzeugen aber leider auch Komplexität und erhöhen damit wieder die Wahrscheinlichkeit zusätzlicher Fehler.

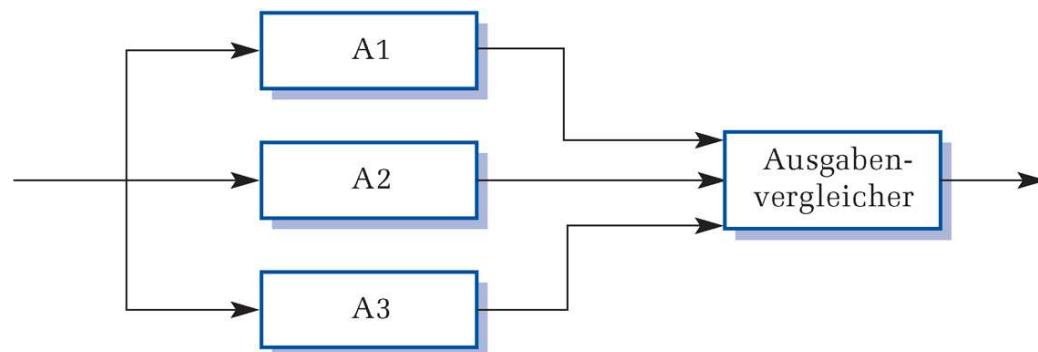


Entwicklung kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Triple modular redundancy (TMR)

- Eine Einheit wird dreifach verwendet.
- Sinnvoll, um Hardwarefehler abzufangen. Hardwarefehler betreffen selten die Gattung, sondern meist die Hardwareinstanz, kommen also unabhängig voneinander vor. Ein gleichzeitiges Versagen aller Komponenten ist unwahrscheinlich.



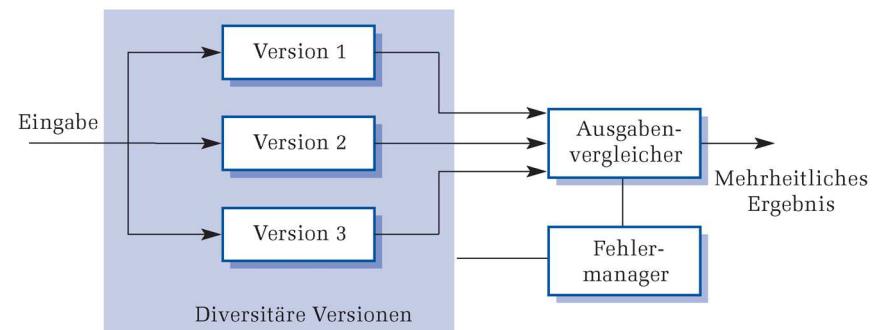


Entwicklung kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

n-Version-Programming

- Redundanz ist nicht sinnvoll, um Softwarefehler abzufangen. Softwarefehler betreffen das Design der Software, kommen also abhängig voneinander in allen Softwareinstanzen vor.
- n Verschiedene Teams implementieren anhand der gleichen Spezifikation n verschiedene Versionen der Software (diversity). Inkonsistente oder verspätete Ausgaben werden nicht berücksichtigt. Im Falle eines Fehlers stehen (n-1) konsistente Ausgaben zur Verfügung.



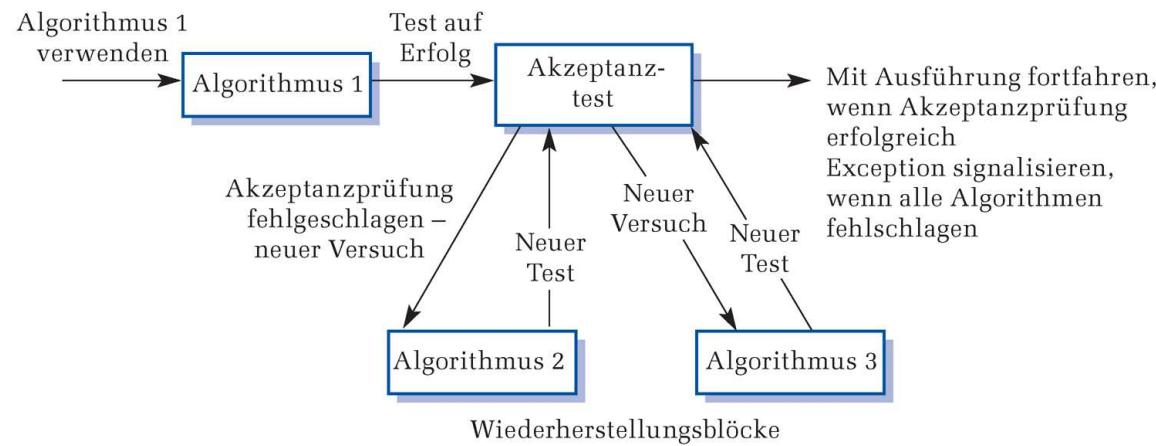


Entwicklung kritischer Systeme

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Recovery blocks

- Die diversitären Versionen werden nacheinander ausgeführt.
- Redundante Hardware ist überflüssig.





Entwicklung kritischer Systeme

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Aufgabe

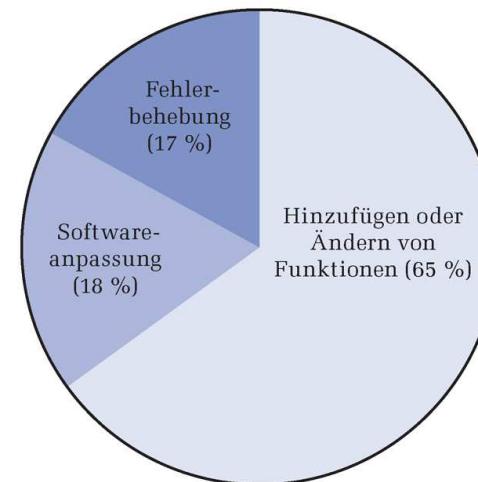
- Nennen Sie Gründe, warum alle unterschiedlichen Versionen eines diversitären Systems auf ähnliche Weise versagen können.



Weiterentwicklung von Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Auch nach dem Go-live eines Systems ergeben sich neue Anforderungen, die Wartung erforderlich machen
 - aufgrund veränderter Geschäftsprozesse, neuer oder geänderter funktionaler Anforderungen
 - zur Korrektur festgestellter Fehler
 - zur Anpassung an eine andere Plattform
 - zur Verbesserung der Performanz





Weiterentwicklung von Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Meir M. Lehman: Laws of software evolution

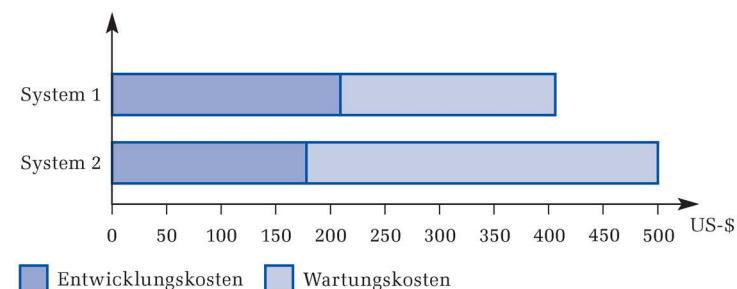
Gesetz	Beschreibung
Kontinuierliche Veränderung	Ein in einer realen Umgebung eingesetztes Programm muss sich unvermeidlich verändern oder gegenüber der Umgebung nach und nach an Nutzen verlieren.
Zunehmende Komplexität	Infolge der Änderungen an einem weiterentwickelten Programm wird dessen Struktur tendenziell immer komplexer. Zusätzliche Ressourcen müssen dem Erhalt und der Vereinfachung der Struktur gewidmet werden.
Weiterentwicklung umfangreicher Programme	Die Programmweiterentwicklung ist ein selbstregelnder Prozess. Systemeigenschaften wie Größe, Zeitabstand der Veröffentlichungen und Anzahl gemeldeter Fehler bleiben von Systemversion zu Systemversion vergleichbar groß.
Organisatorische Stabilität	Über die Lebensdauer eines Programms bleibt seine Entwicklungsraten annähernd konstant und unabhängig von den zur Systementwicklung abgestellten Ressourcen.
Bewahrung der Vertrautheit	Über die ganze Lebensdauer eines Systems bleibt der Umfang der für jede neue Version vorgenommenen Änderungen annähernd konstant.
Kontinuierliches Wachstum	Der von Systemen gebotene Funktionsumfang muss kontinuierlich wachsen, damit die Kundenzufriedenheit hoch bleibt.
Abnehmende Qualität	Die Qualität von Systemen wird scheinbar abnehmen, bis sie an Veränderungen in ihrer Betriebsumgebung angepasst worden sind.
Rückmeldungssystem	Weiterentwicklungsprozesse umfassen auch Multiagentensysteme mit mehrfach wiederholten Rückmeldungen, die Sie als Rückmeldungssysteme behandeln müssen, um eine messbare Produktverbesserung zu erreichen.



Weiterentwicklung von Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Wartungsarbeiten sind bezogen auf die entwickelte Funktionalität teuer. Es lohnt sich, während der Entwicklung in die Wartbarkeit des Systems zu investieren.
 - Die mit der Wartung betrauten Personen müssen sich in die speziellen Aspekte des Systems einarbeiten.
 - Mit zunehmender Nutzungsdauer gibt es immer weniger Experten für die dem System zugrundeliegenden Programmiersprachen, Plattformen, Schnittstellen, bei der Dokumentation verwendeten Modellierungstechniken etc.
 - Mit zunehmender Nutzungsdauer verkomplizieren die Wartungsarbeiten die Architektur und Struktur des Systems immer weiter. Zusätzliche Arbeiten werden dadurch noch aufwendiger.





Weiterentwicklung von Software

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Software Reengineering
 - Um die Wartbarkeit bestehender Systeme zu verbessern, kann es sinnvoll sein, Legacy-Systeme neu zu implementieren.
 - Dazu können gehören: Neue Dokumentation, neue Architektur, andere Programmiersprache.
 - Die Funktionalität ändert sich nicht.
 - Vorteil: Die Kosten sind geringer als bei einer vollständigen Neuentwicklung. Das Risiko ist ebenfalls geringer.
 - Meist werden aber die höheren Kosten einer Neuentwicklung in Kauf genommen, um neue Funktionen berücksichtigen zu können.





Weiterentwicklung von Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Strategien zur Weiterentwicklung von Legacy-Systemen
 - Wenn die Geschäftsprozesse sich seit der Einführung so sehr geändert haben, daß sie kaum noch von dem System abhängen, kommt in Betracht, das System komplett abzuschalten.
 - Wenn die weitere Wartung des Systems mit vertretbaren Kosten möglich ist, kann es in betrieb bleiben und weiter regelmäßig gewartet werden.
 - Wenn das System nach vielen Anpassungen und Änderungen unstrukturiert und schlecht wartbar ist, aber noch für wichtige Geschäftsprozesse gebraucht wird, kommt ein Software Reengineering in Betracht.
 - Wenn die Funktionalität eines System noch gebraucht wird, aber Funktionen in großem umfang geändert oder erweitert werden müssen oder es mit vielen neuen Schnittstellen an eine neue Infrastruktur angepaßt werden muß, sollte es ganz oder teilweise durch ein neues System ersetzt werden.



Weiterentwicklung von Software

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgabe

Was halten Sie für die entscheidenden Bedingungen für den Erfolg eines Software-Reengineering-Projektes?



Verifikation und Validierung von Software



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Verifikation: Entspricht das Produkt den Anforderungen?
- Validierung: Stellt das Produkt die Anwender in ihrer Arbeitsumgebung zufrieden?
- Zwei Verfahren der V&V
 - Softwareinspektion: Man prüft je nach Entwicklungsfortschritt die Spezifikationen, die Entwurfsmodelle oder den Quellcode auf Fehler.
 - Softwaretest: Man bringt das Programm zur Ausführung und prüft, ob es leistet, was es soll.



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standard-software**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Gezielte Fehlersuche bei der Inspektion von Quellcode

Fehlerklasse	Inspektionsüberprüfung
Datenfehler	Sind alle Programmvariablen initialisiert, bevor ihre Werte verwendet werden? Sind alle Konstanten benannt? Sollte der höchste Indexwert von Datenfeldern der Anzahl von Elementen entsprechen oder der Anzahl von Elementen minus 1? Falls Zeichenketten benutzt werden, ist ein Trennzeichen explizit zugewiesen? Besteht die Möglichkeit eines Speicherüberlaufs?
Steuerungsfehler	Sind die Bedingungen jeder bedingten Anweisung korrekt? Wird jede Programmschleife mit Sicherheit beendet? Sind die zusammengesetzten Anweisungen richtig geklammert? Sind in den Case-Anweisungen alle möglicherweise auftretenden Fälle berücksichtigt? Wenn nach jedem Case in einer Case-Anweisung ein break nötig ist, sind dann alle breaks vorhanden?
Ein-/Ausgabefehler	Werden alle Eingabeveriablen verwendet? Sind allen Ausgabeveriablen Werte zugewiesen, bevor sie ausgegeben werden? Können unerwartete Eingaben Schaden anrichten?
Schnittstellenfehler	Verfügen alle Aufrufe von Funktionen und Methoden über die richtige Anzahl von Parametern? Stimmen die formalen mit den tatsächlichen Parametertypen überein? Stimmt die Reihenfolge der Parameter? Falls Komponenten auf gemeinsam genutzten Speicher zugreifen, benutzen sie dasselbe Strukturmodell für diesen Speicher?
Speicherverwaltungsfehler	Falls eine verknüpfte Struktur verändert wurde, wurden alle Verknüpfungen korrekt neu zugewiesen? Falls eine dynamische Speicherung erfolgt, wurde der Speicherplatz korrekt zugewiesen? Wird der Speicherplatz explizit freigegeben, nachdem er nicht mehr benötigt wird?
Exception-Verwaltungsfehler	Sind alle möglichen Fehlerzustände berücksichtigt worden?



Verifikation und Validierung

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Statische Code-Analyse
 - Automatisierte Untersuchung des Quellcodes, ohne das Programm auszuführen.
 - Schwerpunkt nicht auf Laufzeitfehlern, sondern auf der Suche nach typischen, immer wiederkehrenden Fehlern.
 - einstellbar, welche Arten von Fehlern gesucht werden sollen.
 - Beispiel:

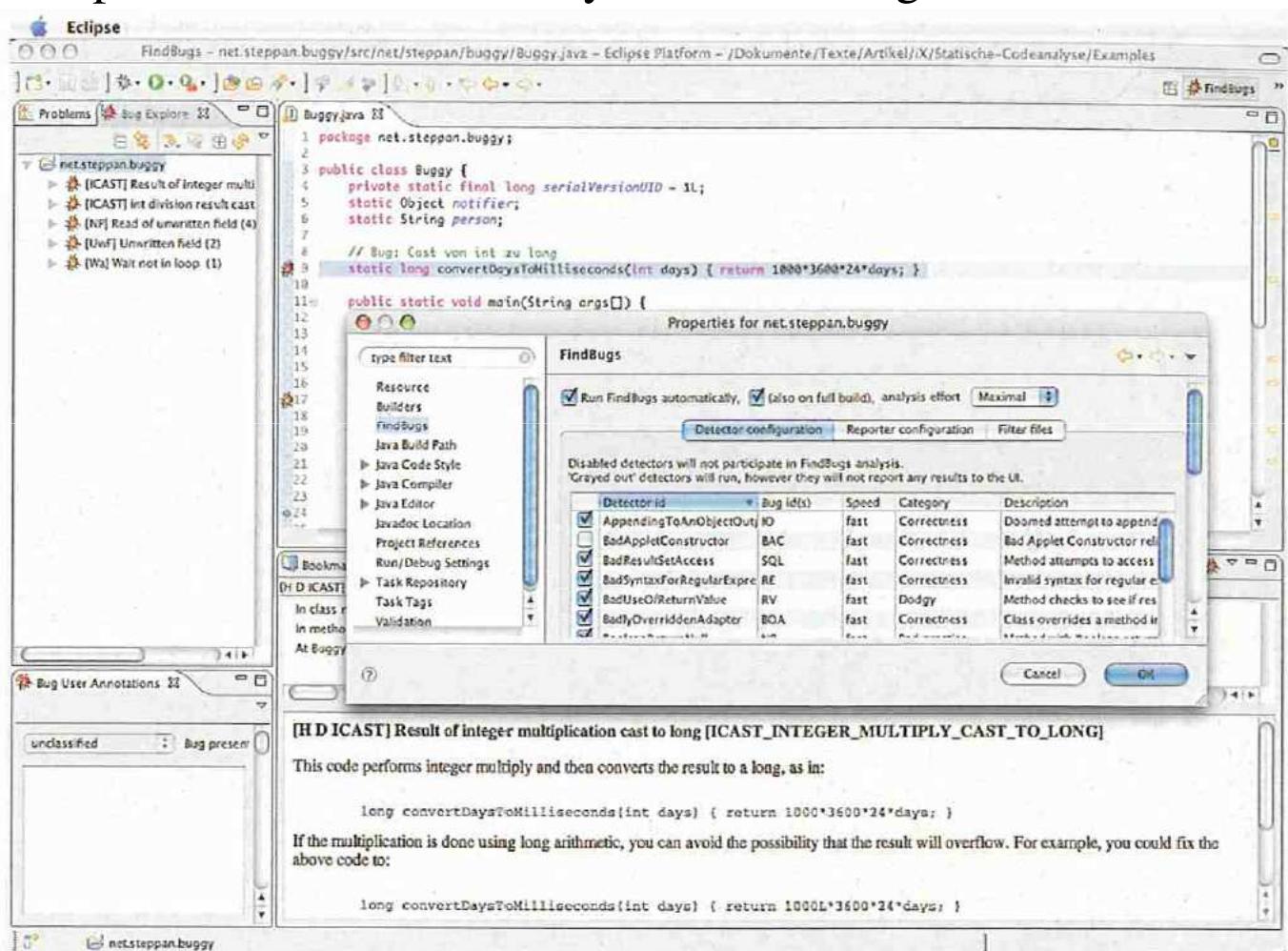
```
int x = 1;  
int y = 10;  
//Bug -> Ergebnis 0.0, statt 0.1  
double value = x / y;  
//Richtig wäre double value = x / (double)y
```

Quelle: iX 2/2009, S. 49

Verifikation und Validierung

Beispiel: Statische Codeanalyse mit FindBugs

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung



Quelle: iX 2/2009, S. 49

Prof. Dr. Carsten Dorrhauer



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgaben

- Was ist der Unterschied zwischen Verifikation und Validierung? Warum ist die Validierung besonders schwierig?
- Warum sind Quellcodeinspektionen effektiv? Welche Arten von Fehlern werden dabei aber nicht gefunden?



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgaben

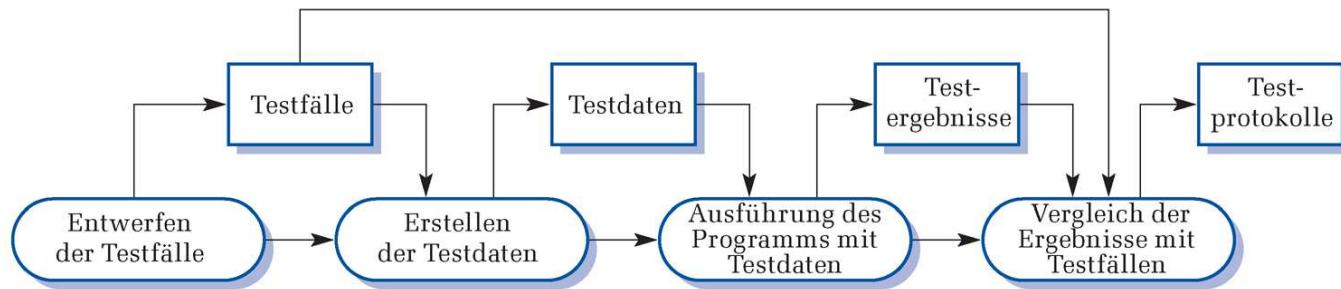
- In welchen Organisationen werden Quellcodeinspektionen problematisch sein?



Verifikation und Validierung

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Softwaretests
 - Tests können nie alle Möglichkeiten abdecken, die bei der Ausführung eines Programms vorkommen können.
 - Man entwirft Testfälle, die die zu tätigen Eingaben und die vom System erwarteten Ausgaben enthalten.

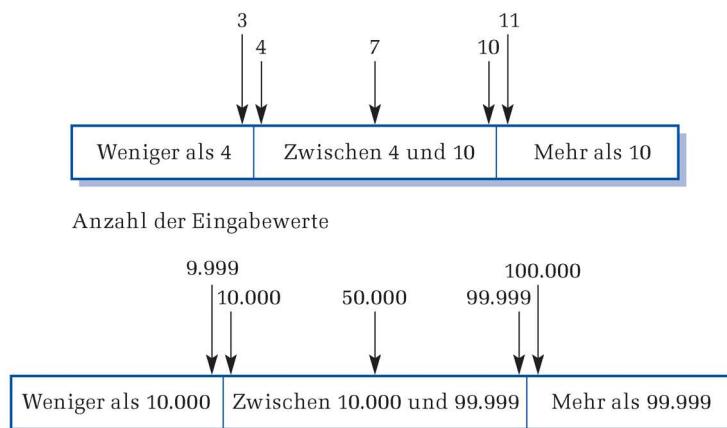




Verifikation und Validierung

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Softwaretests: Erstellen der Test Cases
 - 1. Anforderungsbasiert: Aus den Use Cases werden die Test Cases abgeleitet.
 - 2. Klassenbasiert: Ein- und Ausgabewerte werden klassifiziert nach der Art wie sie verarbeitet werden. Alle Klassen werden getestet. Beispiel: Ein Programm erhält laut Spezifikation zwischen 4 und 8 ganzzahlige Eingaben im Wertebereich zwischen 10000 und 99999





1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Verifikation und Validierung

- Softwaretests: Erstellen der Test Cases
- 3. Strukturell: Man kennt den Algorithmus und versucht, Grenzfälle abzudecken (z.B. den mittleren Wert bei einer binären Suche)
- 4. Pfadüberdeckung: Man versucht, alle Ausführungspfade eines Programms abzudecken. (z.B. den Wahr- und Falsch-Fall aller Verzweigungsbedingungen)

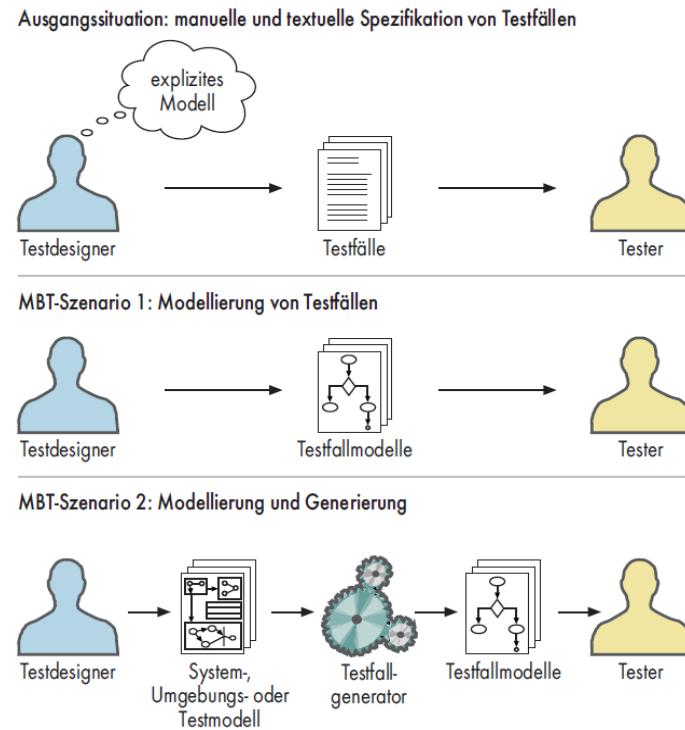




1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Verifikation und Validierung

- Model-Based Testing: Die Testfälle werden nicht textuell, sondern mit Modellen beschrieben.
- Aus Umgebungsmodellen (z.B. Use-Case-Diagrammen) und Systemmodellen (z.B. Aktivitätsdiagrammen) können Modelle für mehrere Testfälle abgeleitet oder generiert werden.



Quelle: iX 11/2009, S. 127



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Softwaretests: Unitest (Schnittstellentest)
 - Unitest werden für die einzelnen Komponenten des Gesamtsystems angewandt. Sie werden über ihre Schnittstelle angesprochen.
 - Richtlinien:
 - Grenzen der Gültigkeitsbereiche aller Parameter testen
 - Reaktion auf ungültige Parameterkombinationen testen
 - Reaktion auf Überlastung durch eine Vielzahl von Aufrufen testen,
 - zusammenwirkende Komponenten in unterschiedlicher Reihenfolge testen



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Softwaretests: Integrationstest/ Systemtest
 - Das Gesamtsystem wird getestet
 - Die Tester haben Zugriff auf den Quellcode. Sie berücksichtigen diese Kenntnisse beim Entwurf der Testcases.
 - Dazu gehören auch Performance-Tests
 - Automatisierte Testtools können zum Einsatz kommen. Ihr Einsatz ist besonders sinnvoll
 - um große Last zu erzeugen
 - wenn so oft getestet werden muß, daß der Initialaufwand zum Einrichten eines Testwerkzeuges gerechtfertigt ist.
 - Falls zunächst nur einige Komponenten und dann immer mehr integriert sind, spricht man vom Systemtest erst ab dem Zeitpunkt der vollständigen Integration aller Komponenten.



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Softwaretests: Regressionstest
 - Wenn es sich nicht um eine komplett neue Software handelt, sondern um eine neue Version, dann spricht man von einem Regressionstest.
 - Damit ist gemeint, daß auch der unveränderte Code nochmals getestet werden muß, um unerwünschte Seiteneffekte des neuen Codes zu finden.



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Softwaretests: Auslieferungstest (User-Acceptance-Test)
 - Das Gesamtsystem wird getestet
 - Die Tester haben keinen Zugriff auf den Quellcode (Black box)



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Verifikation und Validierung

- Softwaretests: Alphatest
- Hersteller von COTS-Software führen oft einen internen Test durch, der den Schwerpunkt ähnlich wie der User-Acceptance-Test auf Funktionalität und Usability setzt.

α



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Softwaretests: Betatest
 - Eine erste Version einer COTS-Software wird an einen (meist eingeschränkten) Benutzerkreis ausgegeben, um Fehlermeldungen zu sammeln.
- Danach folgen
 - Release Candidate (RC)
 - Release to Manufacturing (RTM)
 - General Availability (GA)



Verifikation und Validierung

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Verifikation und Validierung kritischer Systeme
 - Bei kritischen Systemen nehmen V&V einen noch größeren Teil von Budget und Zeit in Anspruch.
 - Grund: Die Auswirkungen eines Ausfalls sind gravierender. Investitionen in V&V mindern die Wahrscheinlichkeit eines solchen Ausfalls.
 - Grund: Kunden verlangen bei kritischen Systemen oft eine gesonderte Dokumentation von deren Verlässlichkeit.



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Verifikation und Validierung

Aufgaben

- Warum können Tests nur das Vorhandensein von Fehlern aufdecken, nicht aber die Fehlerfreiheit?



Standardsoftware



Standardsoftware

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

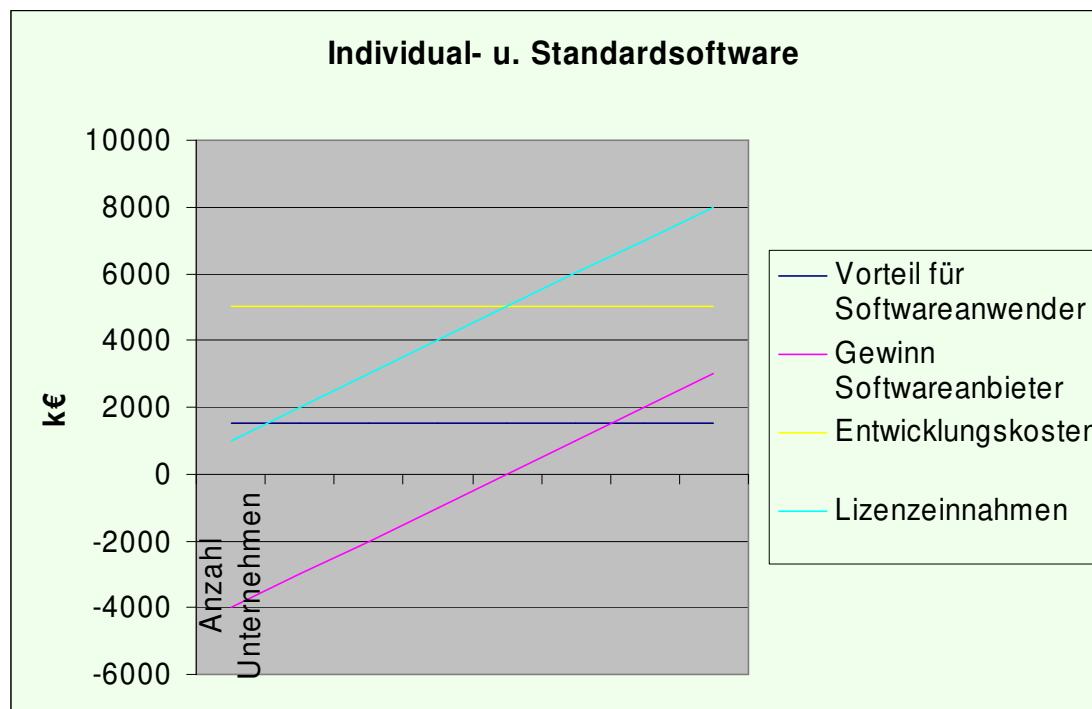
- Eine Alternative zur Entwicklung von Software ist oft (aber nicht immer) der Kauf von Standardsoftware.
- Definition Standardsoftware
 - Universell, nicht organisationsspezifisch
 - Definierte Problemlösung
 - Minimierter Anpassungsaufwand
 - Festpreise
 - d.h. nach Prozessoren, nach concurrent oder named user, nach genutzten Modulen,...
 - aber nicht: nach Entwicklungszeit oder -aufwand
- Kerngedanke:
 - Ähnliche Aufgaben und Abläufe in verschiedenen Unternehmen
 - Hoher Fixkostenanteil bei der Softwareentwicklung



Standardsoftware

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Sehr vereinfachende Annahme: Entwicklungskosten für anpaßbare Standardlösung 5000 k€, Entwicklungskosten Individuallösung 2500 k€, Lizenzkosten Standardlösung 1000 k€





Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Modellvoraussetzung: Standardnahe Anforderungen.
- In letzter Zeit zunehmender Preiskampf. Nicht mehr alles am Markt durchsetzbar.
- Wartungskosten nicht berücksichtigt.
- Wichtige Projektentscheidungen:
 - Individual- vs. Standard
 - Wahl des Anbieters bei Standard



Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Anpassung von Standardsoftware
 - Durch Parameter (wenig Aufwand, weniger stark individualisierbar, Pflege bei Releasewechsel unproblematisch)
 - Durch integrierte Entwicklungsumgebungen (Bsp.: ABAP bei SAP)
 - Durch externe Ergänzungen auf beliebigen Plattformen und Nutzung bestehender Schnittstellen (hoher Aufwand, stark individualisierbar, Pflege bei Releasewechsel problematisch)



Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

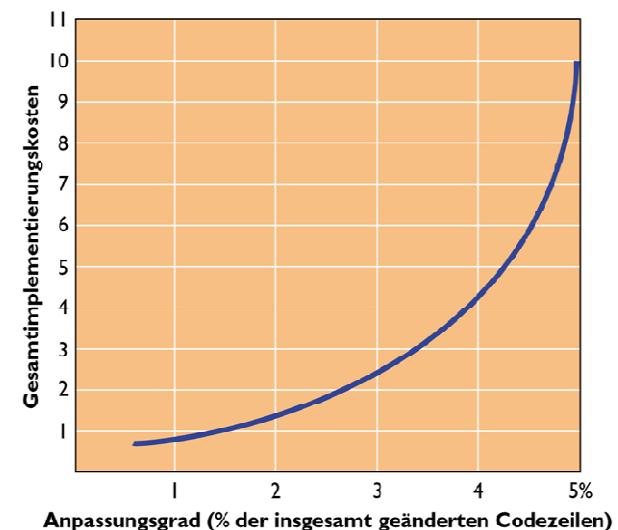
- Begriff „Customizing“: Manche Hersteller verstehen darunter Parametrisierung, andere auch oder ausschließlich Ergänzungsprogrammierung
- Standardsoftware mit hoher Anzahl von Parametern:
 - Flexibler
 - Weniger Programmierung notwendig
 - Schwieriger zu verstehen
 - Längere Einführungszeiten



Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Begriff „Customizing“: Manche Hersteller verstehen darunter Parametrisierung, andere auch oder ausschließlich Ergänzungsprogrammierung
- Standardsoftware mit hoher Anzahl von Parametern:
 - Flexibler
 - Weniger Programmierung notwendig
 - Schwieriger zu verstehen
 - Längere Einführungszeiten





Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Argumente Pro Standard
 - „State-of-the-art“-Prozesse /Zukauf von Expertise
 - Kosten besser abschätzbar
 - Wartungsaufwand abschätzbar
 - Kein kurzfristiger Personalaufwand während der Entwicklungszeit
 - Spätere Änderung durch Parameter jederzeit möglich
 - Einführung i.d.R. schneller
 - Ausgereifte Software
 - Prüfung des Systems vor der Entscheidung möglich

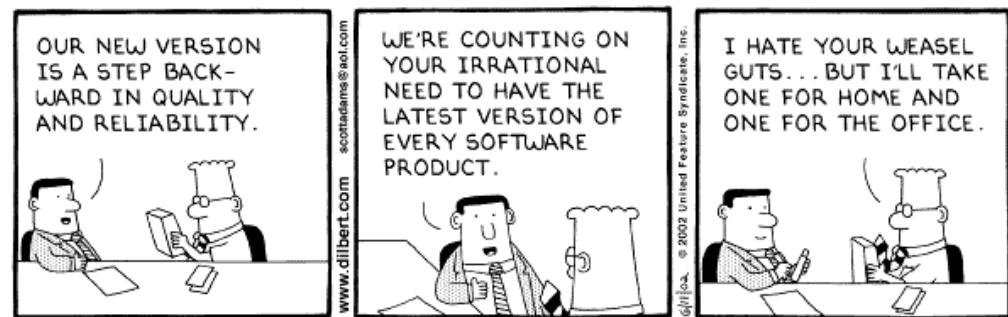




1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Standardsoftware

- Argumente Pro Standard
 - Service, Dokumentation, Schulung, Entwicklung aus einer Hand
 - Integration, einheitliche Datenhaltung, wenig Schnittstellen
 - Einheitliche GUI über alle Module, Ergonomie
 - Anpassung an zukünftige Anforderungen des Marktes durch den Hersteller
 - Experten auf dem Arbeitsmarkt verfügbar (Sowohl Anwender als auch IT-Fachkräfte)



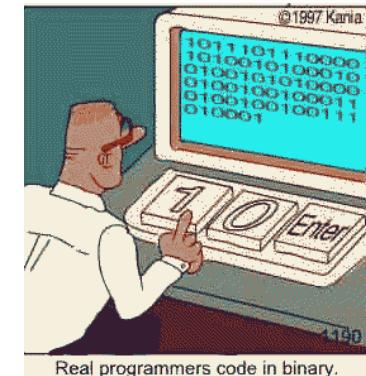
Copyright © 2002 United Feature Syndicate, Inc.



Standardsoftware

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Argumente Contra Standard/ Pro Individual
 - Überladene Standardprodukte, aufwendige Anpassung
 - Im Vorfeld keine Erfahrung mit den Standardlösungen, Auswahl schwierig
 - Wenig kritische Informationen im Vorfeld über die Standardprodukte
 - Standard weniger flexibel, keine 1:1-Abbildung bestehender Prozesse
 - Legacy-Integration mit Individuallösung i.d.R. einfacher
 - Abhängigkeit vom Hersteller
 - Individuell gestaltete und unterstützte Prozesse bringen Vorteile zum Wettbewerb





Standardsoftware

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Argumente Contra Standard/
Pro Individual
 - Performance/Hardwarebedarf:
Individuallösungen i.d.R.
technisch „schlanker“
 - Akzeptanz, Motivation („Not invented here“)
 - Standardsoftware evtl. nicht auf bestehenden
Plattformen lauffähig
 - Auswahlverfahren verlängert Projektlaufzeit
 - Standardlösungen oft nicht auf dem neusten
technischen Stand
 - Keine Standardlösung für sehr ausgewogene Aufgaben
verfügbar





Standardsoftware

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Auswahl von Standardsoftware
 - Best-of-breed
 - Zu jeder Teilaufgabe die beste Lösung
 - Keine Integration seitens der Hersteller
 - Anwendendes Unternehmen ist selbst für Schnittstellen verantwortlich
 - Wenn unklar ist, welcher Teil einen Fehler verursacht, könnten alle Hersteller die Verantwortung ablehnen
 - Alles aus einer Hand
 - Hersteller garantiert Gesamtlösung
 - Evtl. wenige oder keine Hersteller, die alle erforderlichen Komponenten anbieten



- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Standardsoftware

Auswahlkriterien für Standardsoftware

- Funktionalität
- Kosten
- Anpaßbarkeit
- Datenstruktur
- Usability
- Performance
- Skalierbarkeit
- Dokumentation
- Verfügbarkeit von Experten
- Architektur
- Internationalität
- Portabilität
- Situation des Herstellers
- Wartung, Beratung, Schulung
- Schnittstellen
- Schulungsversion
- Stabilität
- Releasewechsel: Häufigkeit, Technik



Frameworks



Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- Was ist ein Framework?
 - Eine Sammlung von Klassen kann erweitert werden, indem man Klassen hinzufügt, die Methoden und Attribute von den bestehenden Klassen erben.
 - Man definiert Callbacks: Man lässt eigene Methoden vom Framework aufrufen.





1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Frameworks

Notwendigkeit von Java EE Frameworks

- entstanden aufgrund von Defiziten herkömmlicher Web-Applikationen
- Ursprünglicher Zweck von http und html: Auslieferung und Gestaltung statischer Hypertext-Seiten





Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standard-software
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Notwendigkeit von Java EE Frameworks

- Häufige Anwendung von http und html: Auslieferung und Erstellung dynamisch erzeugter Hypertext-Seiten

The screenshot shows a web browser window with the title bar "Prüfungsamt - Fachhochschule Ludwigshafen am Rhein - Mozilla Firefox". The main content area displays a blue-themed user interface for a registration system. On the left, there is a sidebar with links: "Kontakt", "Suchen", "FH-Home", "Aktuelle Termine", "KLAPPA", "Klausurenpläne und Feedback", "KLIC", "Notenabfrage und Prüfer gesamtmeldung", "Feedback", "Impressum", "Anleitung zur Prüfungsanmeldung (NO)", "Hinweise zur Erstellung der Diplomarbeit", and "Kontakt". The right side features the logo of "Fachhochschule Ludwigshafen am Rhein Hochschule für Wirtschaft" and the heading "Prüfungsamt". Below it, there is a section titled "KLIC Notenabfrage über das Internet" with instructions about SSL encryption requirements and a password input field. Further down, there is information about the new examination regulations and a link to change the password.



Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Notwendigkeit von Java EE Frameworks

- Resultierende Schwierigkeiten für Web-Entwickler:
- Woher weiß ich, ob die Produktseite im Online-Shop von demselben Benutzer angefordert wird, der vor 2 Minuten ein anderes Produkt seinem Warenkorb hinzugefügt hat?
- Lösung: session-id
- <http://shop.firma.de/index.jsp?sessionid=665db942076a89aade16>
- Und überhaupt: In meinen Desktop-Applikationen kann ich ganz einfach einen Listener an eine GUI-Komponente hängen und ein Objekt mit den Eingabedaten füllen. Warum ist das im Web so umständlich?

...



Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Notwendigkeit von Java EE Frameworks

- Wie halte ich bei größeren Projekten die Zuständigkeiten für Web-Design und Programmierung voneinander getrennt?

Keine Lösung: Java Server Page

```
<html>
<head><title>Ihr neuer Warenkorb</title></head>
<body>
<%
    out.println("In Ihrem Warenkorb befindet sich: " + Artikel);
%>
</body>
</html>
```

Keine Lösung: Servlet

```
out.println("<html>");
out.println("<head><title>Ihr neuer Warenkorb</title></head>");
out.println("<body>");
out.println("In Ihrem Warenkorb befindet sich: " + Artikel);
out.println("</body>");
out.println("</html>");
```

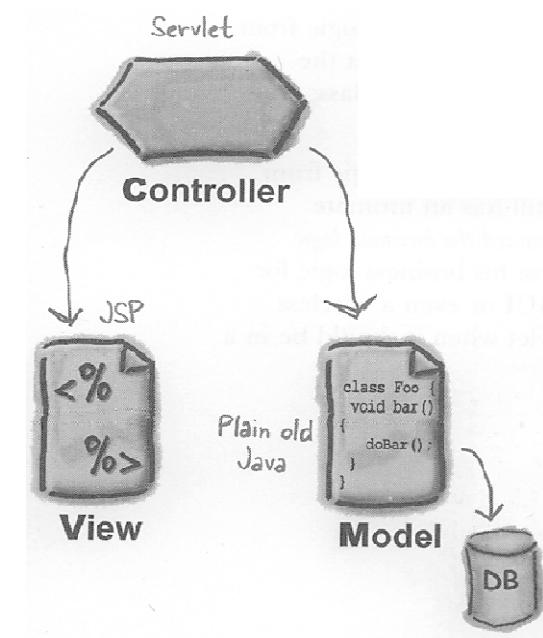


Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Notwendigkeit von Java EE Frameworks

- Wie halte ich bei größeren Projekten die Zuständigkeiten für Web-Design und Programmierung voneinander getrennt?
- Lösung: Entwurfsmuster MVC (Model-View-Controller)





Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Definition von Frameworks:

Johnson & Foote 1988*:

- Frameworks provide a way of reusing code that is resistant to more conventional reuse attempts.
- The framework ... plays the role of the main program in coordinating and sequencing application activity. This *inversion of control* gives frameworks the power to serve as extensible skeletons.



* Ralph E. Johnson / Brian Foote: Designing Reusable Classes,
in: Journal of Object-Oriented Programming 2/1988,
S. 22-35, online bei <http://www.laputan.org/drc/drc.html>



Frameworks

Definition von Frameworks:

Fayad und Schmidt 1997*:

- The primary benefits of OO application frameworks stem from the
- modularity,
- reusability,
- extensibility, and
- inversion of control.
- Frameworks can be viewed as a concrete reification of families of design patterns that are targeted for a particular application-domain.



* Mohamed Fayad/ Douglas C. Schmidt: Object-Oriented Application Frameworks, in: Communications of the ACM 10/ 1997, S.32ff. Online bei
<http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html>



Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Beispiel Struts:

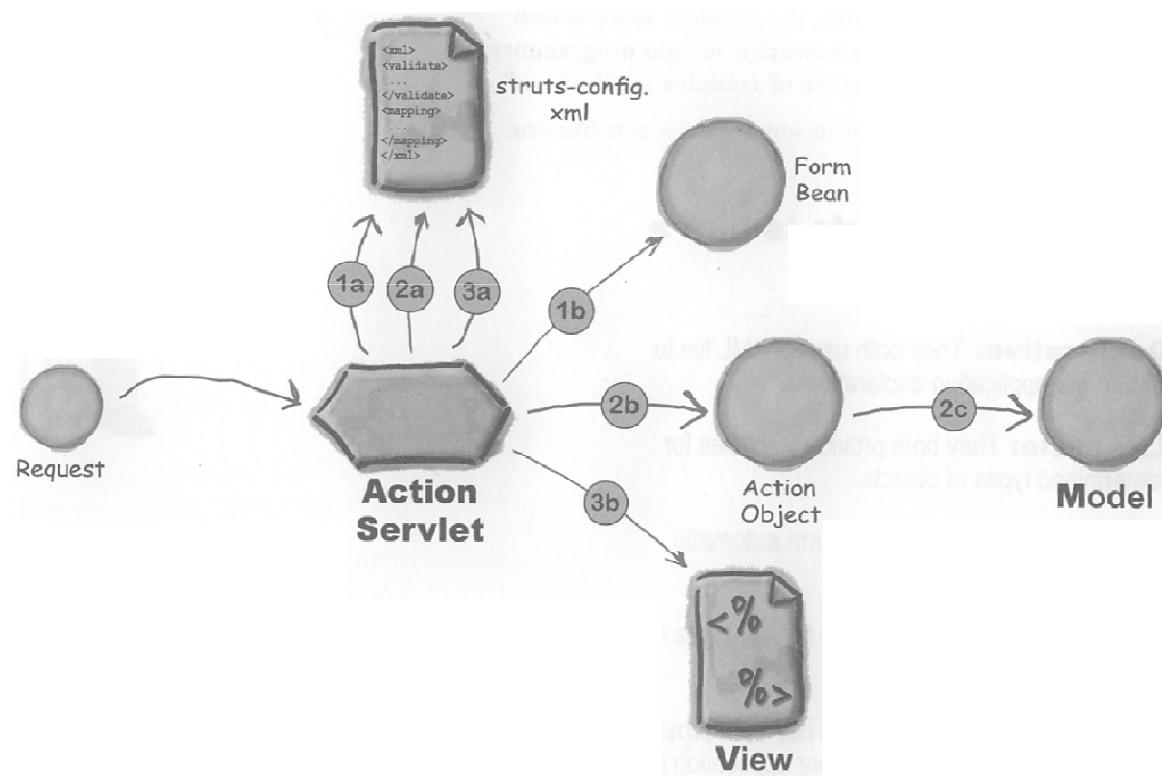
- Sun hat die Entwurfsmuster zur MVC-Umsetzung in der EE spezifiziert, aber keine Implementierung dafür mitgeliefert.
- Mit Struts liegt eine quelloffene Implementierung vor.
- Seit 2000 unter dem Dach von Apache, ausgereift
- weit verbreitet daher:
- zukunftsicher
- Entwickler für Struts-Applikationen sind gut verfügbar
- Schwerpunkt ist die Umsetzung des Model-View-Controller-Musters.
- Weitere Funktionen, z.B. Internationalisierung, Eingabevalidierung, Fehlerbehandlung



1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Frameworks

Beispiel Struts:





Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

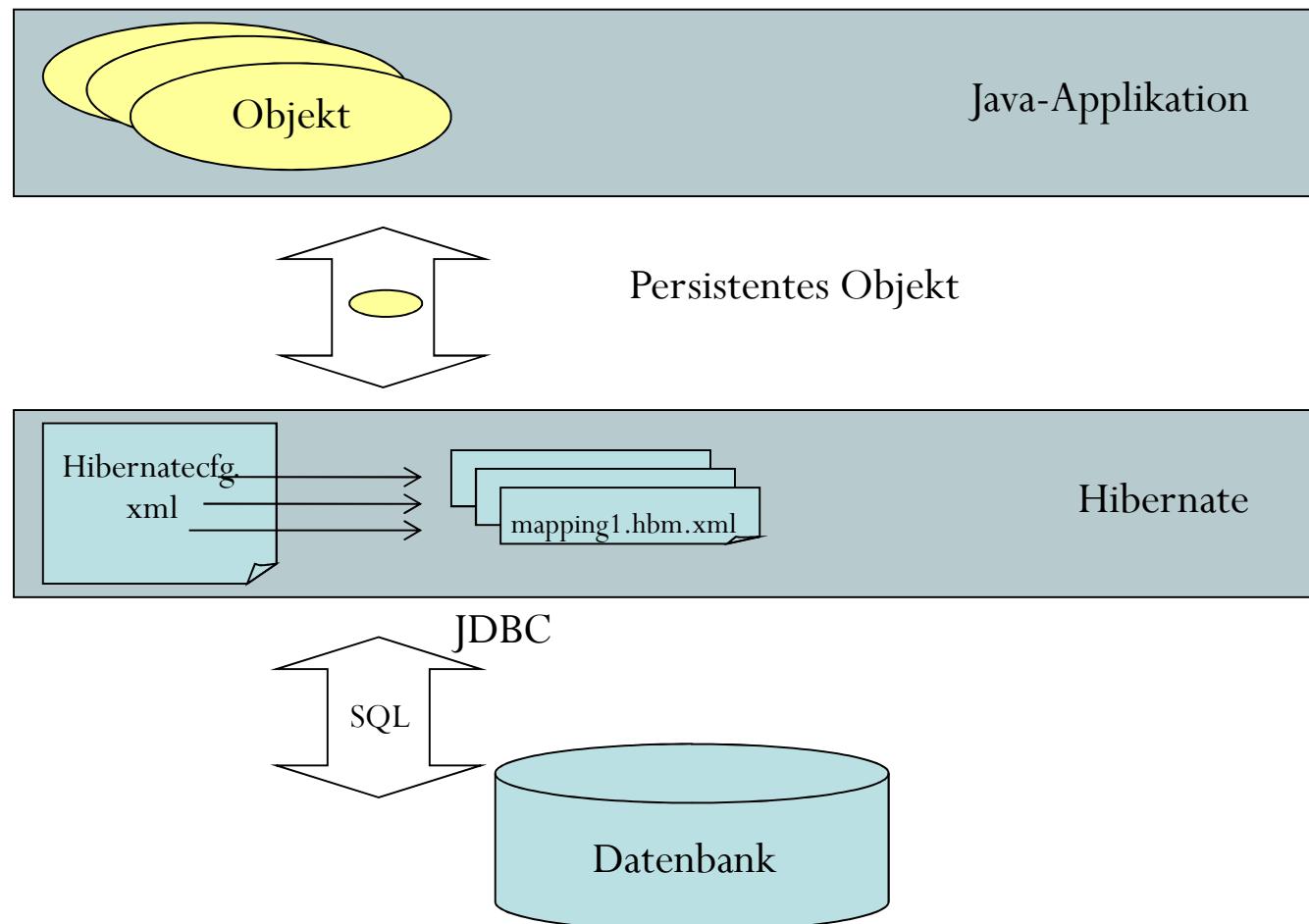
Beispiel Hibernate:

- Web-Applikationen müssen Objekte speichern (persistent machen).
- Relationale Datenbanken erwarten aber keine Objekte, sondern Einträge in Tabellen.
- Das quelloffene Hibernate bildet Objekte auf Tabellen ab.
- In Mapping-Dateien (XML-Format) werden Klassen auf Tabellen zugewiesen.
- In Java können die zugehörigen Objekte dann gespeichert werden.
- Diese Objekte können dann mit oo Kriterien (also ohne SQL) wieder in der Datenbank gesucht werden.
- Hibernate arbeitet mit den gängigen Datenbanksystemen zusammen.
- Applikationen, die ausschließlich via Hibernate auf die Datenbank zugreifen, bleiben bezüglich der Datenbanksysteme portabel.

Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Beispiel Hibernate:





Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Beispiel Spring:

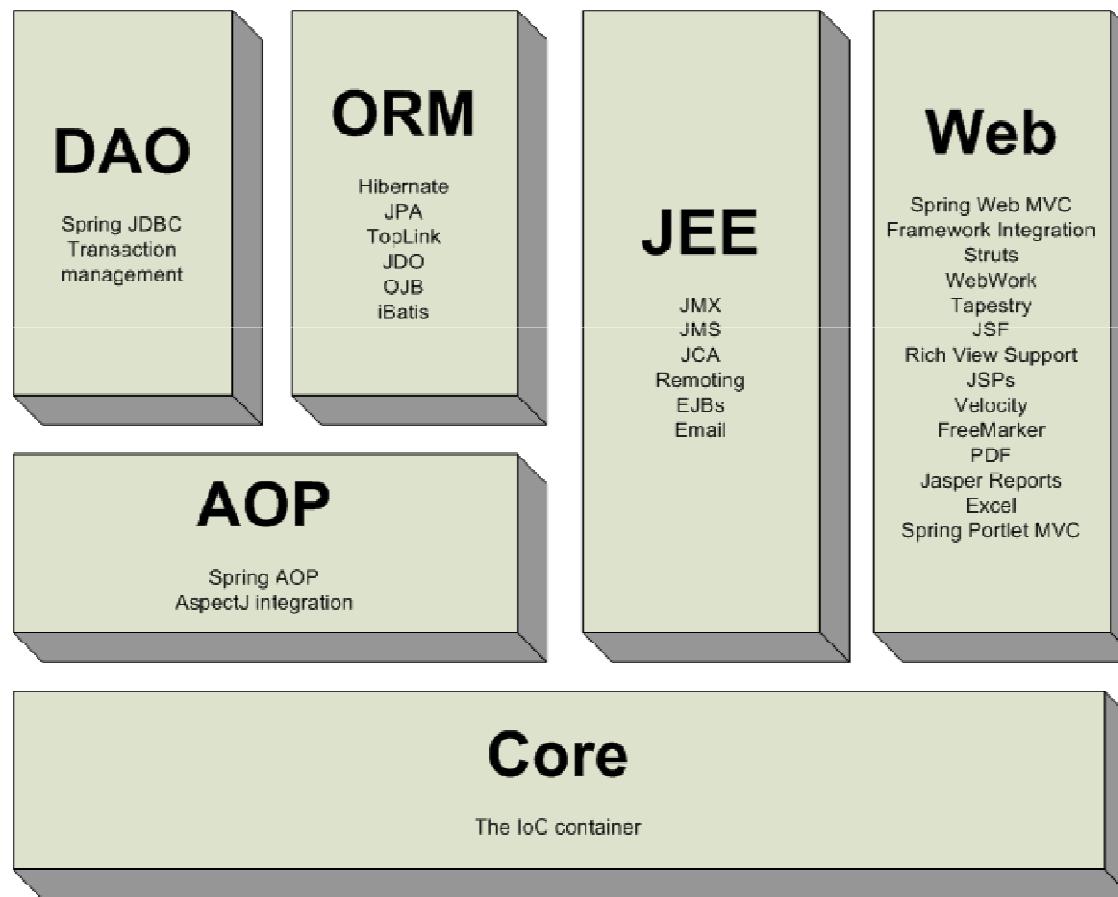
- Spring entstand als Beispielcode zu einem JavaEE-Design-Lehrbuch.
- Das quellöffentne Framework ist modular aufgebaut und unterstützt u.a. folgende Konzepte:
 - Model-View-Controller-Muster
 - Aspektorientierte Programmierung
 - Dependency Injection
- Modularität und einfache Kombinierbarkeit der Komponenten mit anderen Werkzeugen
- Enthält einige Funktionen von Enterprise Java Beans (EJB), aber keinen vollständigen EJB-Container
- Gilt funktional als das umfangreichste Java Framework



Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Beispiel Spring:





Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Beispiel Java Server Faces:

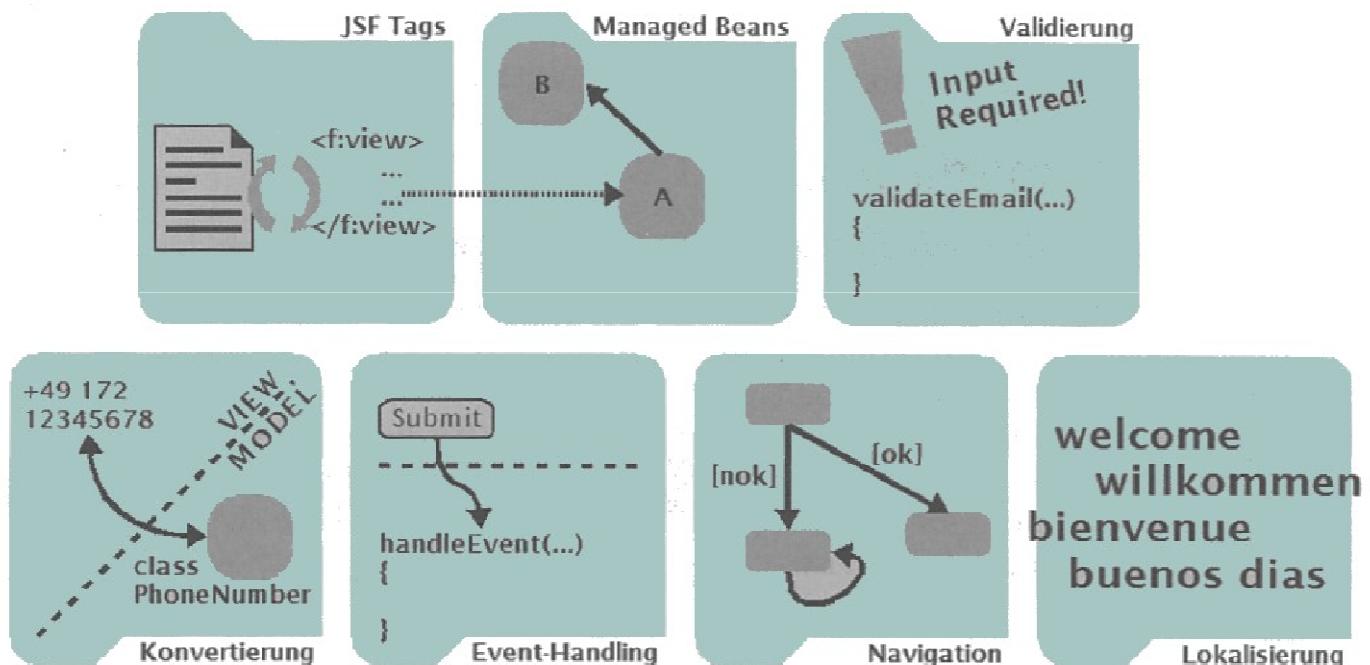
- Framework-Spezifikation mit Referenzimplementierung von Sun selbst, daher vermutlich sehr zukunftssicher
- Zweck: Einfache Entwicklung von Benutzeroberflächen für Web-Applikationen
- z.B. Techniken für Konvertierung, Validierung von Benutzereingaben, Lokalisierung von Ausgaben
- UI-Komponenten behalten ihren Status über mehrere Seitenaufrufe.
- Ereignisgesteuerte Entwicklung mit Listener ähnlich AWT
- Die Ablauflogik zwischen den Seiten wird per Konfigurationsdatei definiert.
- Häufig kombiniert mit Spring und Hibernate



Frameworks

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

Beispiel JSF:





Frameworks

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Aufgabe: Beurteilen Sie, ob Sie in den folgenden Anwendungsfällen für eine Java-Web-Applikation auf ein Framework zurückgreifen. Auf welches?

- Sie gründen nebenberuflich einen Internethandel mit Weihnachtsartikeln.
- Ein großes Versandhaus migriert sein Kundenportal auf Java EE.
- Der Vorsitzende eines Fußballclubs gibt jeden Samstag die Spielergebnisse ein, damit alle sie im Web abrufen können.
- Die Stadtbibliothek ermöglicht ihren Benutzern eine Online-Recherche. Sie legt Wert darauf, die Investition auf Jahre zu sichern, ohne die Plattform zu wechseln.
- Eine kleine deutsche Vertriebsniederlassung eines ausländischen Konzerns eröffnet einen Online-Vertrieb.



Serviceorientierte Architekturen



SOA

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Service-orientierte Architektur (SOA)

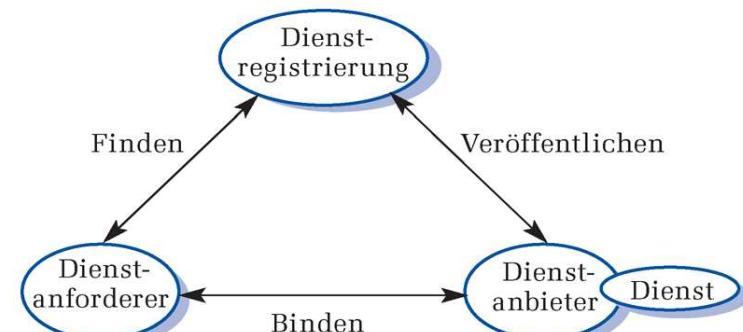
- ... sind eine spezielle Art der Wiederverwendung mit Komponenten mittlerer Größe
- Die Komponenten sind eigenständige Dienste, die im Netzwerk verteilt ausgeführt werden.
- Für die Kommunikation zwischen den Diensten und zum Aufruf der Dienste durch Applikationen werden Standardprotokolle verwendet.
- Sie sind deshalb plattform- und sprachunabhängig verwendbar. Aus Diensten unterschiedlicher Anbieter können Anwendungen zusammengestellt werden.



SOA

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- XML-basierte Standards für Service-orientierte Architektur:
 - Simple Object Access Protocol (SOAP): definiert das XML-Format der Nachrichten, die ein Service erhält und verschickt.
 - Universal Description, Discovery and Integration (UDDI): definiert das Format der Nachrichten, mit denen nach einem bestimmten Dienst gesucht werden kann und wie die Informationen über Dienste organisiert werden.





SOA

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

- XML-basierte Standards für Service-orientierte Architektur:
 - Web Services Description Language (WSDL): definiert, wie ein Service beschrieben wird.

```
<?xml version="1.0" encoding="I
<definitions name="AktienKurs">
  targetNamespace="http://loca ....
  xmlns:xsd="http://schemas.xmlsoap.or
  xmlns="http://schemas.xmlsoap.org/wsd
  <service name="AktienKurs">
    <port name="AktienSoapPort" binding
      <soap:address location="http://loc
    </port>
    <message name="Aktie.HoleWert">
      <part name="body" element="xsd:Tra
    </message>
    ...
  </service>
</definitions>
```





SOA

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

- Häufigste Form von Services: Web Services (standardisierte XML-Nachrichten werden via http oder https ausgetauscht.)
- WebServices: Dienstanbieter spezifizieren Ihre Dienste in WSDL, veröffentlichen Sie mit UDDI, und sie werden mit SOAP-Nachrichten aufgerufen.
- Ursprüngliche Idee: Öffentliche verfügbare Web Services werden nach Bedarf eingesetzt und nach Nutzung bezahlt.
- Aktuelle Marktsituation: Web Services werden zwar angewandt, aber vor allem in geschlossenen Netzwerken.



Entwurfsmuster



Entwurfsmuster

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Die Wiederverwendbarkeit von Softwarekomponenten stößt an ihre Grenzen, wenn die Entwickler der Komponente Entwurfsentscheidungen getroffen haben, die im Widerspruch zu den eigenen Anforderungen stehen.
- Entwurfsmuster (Design patterns) sind keine Softwarekomponenten, sondern abstrakte Konzepte.
- Sie beschreiben erfolgreiche Designs, die immer wieder verwendet werden können.
- In Entwurfsmuster gehen die Erfahrungen vieler Softwareprojekte ein.
- Heute sind viele Entwurfsmuster frei verfügbar.

Entwurfsmuster

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Prof. Dr. Carsten Dorrhauer

- Gang of four (GoF)
- führte 1994 das Konzept der Design Patterns in die Softwareentwicklung ein.
- Ralph Johnson, Richard Helm, Erich Gamma, John Vlissides († 2005)





Entwurfsmuster

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Ein Entwurfsmuster besteht aus folgenden Elementen

- Name des Musters, möglichst treffend
- Domäne: Wo kann man das Muster anwenden? Warum ist es von Nutzen?
- Lösungsbeschreibung, meist auch graphisch. Oft UML-Klassendiagramme
- Konsequenzen: Welche Beschränkungen hat das Mustare, was kann man damit nicht erreichen, in welchen Situationen ist es ineffizient?

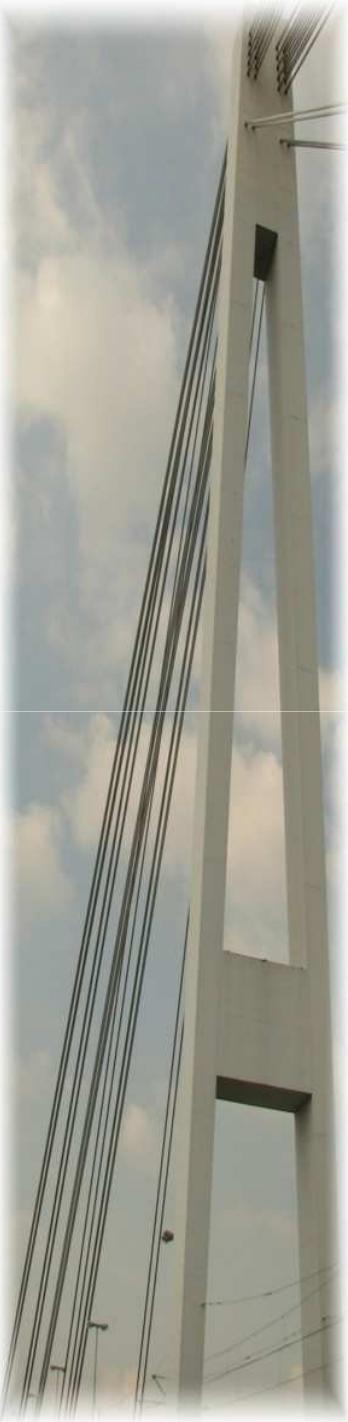


Entwurfsmuster

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

Aufgabe

Warum stellen Entwurfsmuster eine effektive Form der Wiederverwendung dar? Was sind die Nachteile dieses Wiederverwendungsansatzes?



Aspektorientierte Entwicklung



Anforderungen – Komponenten (m:n)

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

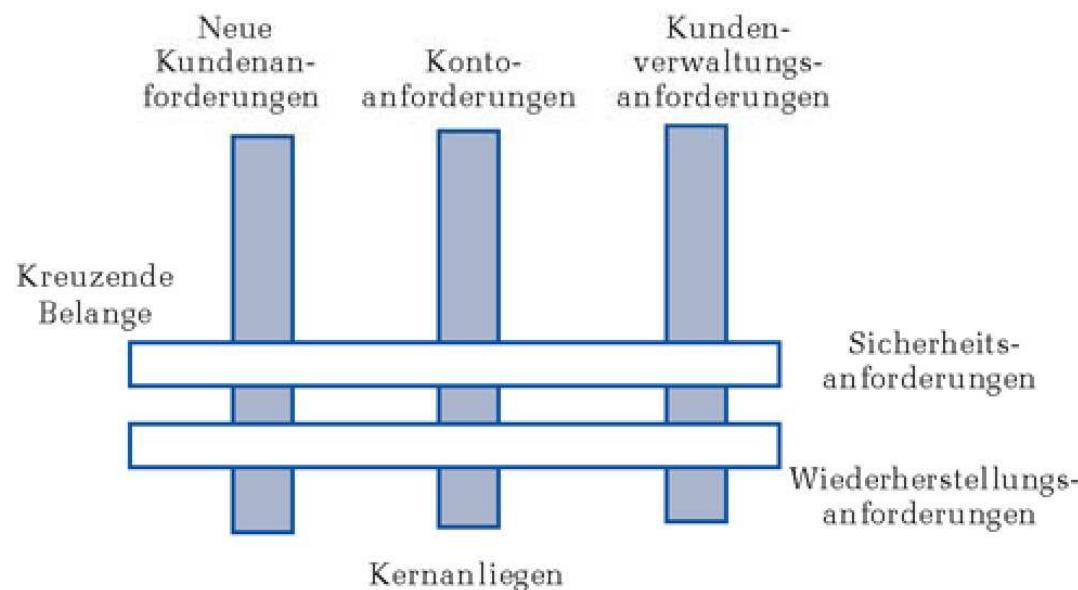
- Für die Wartung eines Softwaresystems ist es ideal, seine Aufteilung in Komponenten anhand der Anforderungen vorzunehmen.
- Dies entspricht vollständig dem Prinzip "Separation of concerns".
- Man kann versuchen, diesem Ziel so nahe wie möglich zu kommen, doch zu 100% wird es nie erreichbar sein.
- Es gibt immer Anforderungen, die sich auf mehrere Komponenten beziehen und Komponenten, die mehrere Anforderungen erfüllen müssen. (m:n-Beziehung)



Anforderungen – Komponenten (m:n)

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Der Grund dafür: Das Prinzip "Separation of concerns" (Trennung von Belangen) stößt an seine Grenzen, wenn die Belange sich überlagern. (crossing concerns)

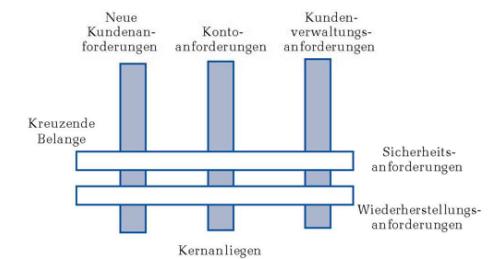




Anforderungen – Komponenten (m:n)

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Serviceorientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

- Häufige Beispiele für crossing concerns: Authentifizierung, Logging, Sicherung im Hintergrund
- Aspektorientierte Entwicklung (AOSD) führt daher das Konzept der Aspekte ein. Ein Aspekt kapselt Funktionalität, die andere Funktionalität kreuzt, also an vielen Stellen im System vorkommt.
- Im Quellcode wird spezifiziert, wie Aspekte in den Code eingewoben werden, um ein ausführbares Programm zu erzeugen.
- Dadurch können Aspekte unabhängig voneinander geändert und wiederverwendet werden.





Beispiel AOSD

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Beispielanforderung: Authentifizierung des Benutzers vor jeder Änderung kritischer Daten in einem Banksystem.
- Lösung 1: Alle Update-Methoden in allen Komponenten rufen die Methode für die Authentifizierung auf.
 - Nachteil: Vermischung von Concerns. Schreibzugriffe auf die Datenbank und Authentifizierung können nicht mehr getrennt voneinander gepflegt und wiederverwendet werden.
- Lösung 2: Vor jedem Aufruf von Update-Methoden wird die Methode für die Authentifizierung aufgerufen.
 - Nachteil: Gestreute Implementierung. Der Code (zumindest die Methodenaufrufe) steht an sehr vielen verschiedenen Stellen im ganzen System verteilt. Jeder Entwickler irgendwelcher Codeteile muß die Authentifizierungsmethoden richtig aufrufen können. Wenn Aufrufparameter sich ändern, muß das an unzähligen Stellen angepaßt werden.

Beispiel AOSD

- 
- 1. Grundlagen**
 - 2. Vorgehensmodelle**
 - 3. Spezifikation**
 - 4. Entwurf**
 - 5. Entwicklung**
 - 6. Validierung**
 - 7. Standardsoftware**
 - 8. Frameworks**
 - 9. Service-orientierte Architektur**
 - 10. Entwurfsmuster**
 - 11. Aspekt-orientierte Entwicklung**

Lösung mit AOSD: Der "crossing concern" Authentifizierung wird als Aspekt implementiert. Im Gegensatz zu Methoden enthalten Aspekte einen Hinweis darauf, wo im Programm sie aufgerufen werden sollen.

```
aspect authentication
{
    before: call (public void update*(..)) // Dies ist ein Pointcut
    {
        // Dies ist der Advice, der beim Einweben in das laufende System
        // ausgeführt werden soll
        int tries = 0 ;
        string userPassword = Password.Get (tries) ;
        while (tries < 3 && userPassword != thisUser.password ( ) )
        {
            // Erlaubt drei Versuche, das richtige Kennwort einzugeben
            tries = tries + 1 ;
            userPassword = Password.Get (tries) ;
        }
        if (userPassword != thisUser.password ( )) then
            // Ist das Kennwort falsch, wird angenommen, dass der Benutzer
            // vergessen hat, sich abzumelden
            System.Logout (thisUser.uid) ;
        }
    } // Authentifizierung
```



Aspekte

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Ein Aspekt besteht aus:
 - Advice : der Quellcode gemäß den (andere Anforderungen kreuzenden) Anforderungen.
 - Pointcut: die Definition, an welchen Join Points der Aspekt eingefügt werden soll.
- Mögliche Join Points:
 - Vor oder nach dem Aufruf bestimmter Methoden oder Konstruktoren.
 - Zugriffe auf bestimmte Attribute einer Klasse oder bestimmter Objekte
 - Initialisierung von Objekten
 - Exceptions



Begriffe des AOSD

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

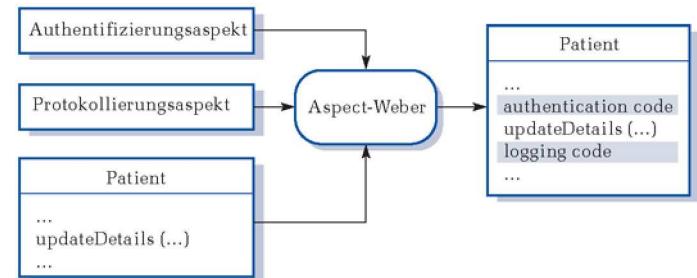
Begriff	Definition
Advice	Der Code zum Implementieren eines Belangs.
Aspekt	Eine Programmabstraktion, die einen sich kreuzenden Belang definiert. Sie enthält die Definition eines Pointcuts und den mit dem Belang verknüpften Advice.
Verbindungspunkt (Join Point)	Ein Ereignis in einem ausgeführten Programm, bei dem der mit einem Aspekt verknüpfte Advice ausgeführt werden kann.
Verbindungspunktmodell	Der Satz von Ereignissen, auf die in einem Pointcut verwiesen werden kann.
Pointcut	Eine in einem Aspekt eingeschlossene Anweisung, die die Verbindungspunkte definiert, an denen der mit dem Aspekt verknüpfte Advice ausgeführt werden soll.
Einweben	Das Einbeziehen des Advicecodes an den angegebenen Verbindungspunkten durch einen Aspect-Weber.



Einweben von Aspekten

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspekt-orientierte Entwicklung

- Möglichkeiten des Einwebens von Aspekten:
 - Quellcodevorverarbeitung: Zunächst läuft der Aspect Weaver über den Quellcode und fügt ihn (in angepaßter Form) an den passenden Stellen ein. Danach lässt sich ein Standardcompiler verwenden.
 - während des Linkens: Der Aspect-Weaver wird sozusagen in den Compiler "eingebaut".
 - zur Laufzeit: Join Points werden ständig im Hintergrund überwacht. Tritt ein definiertes Ereignis ein, so wird der zugehörige Advice ausgeführt.





Aspektorientiertes Design

- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Serviceorientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspektorientierte Entwicklung**

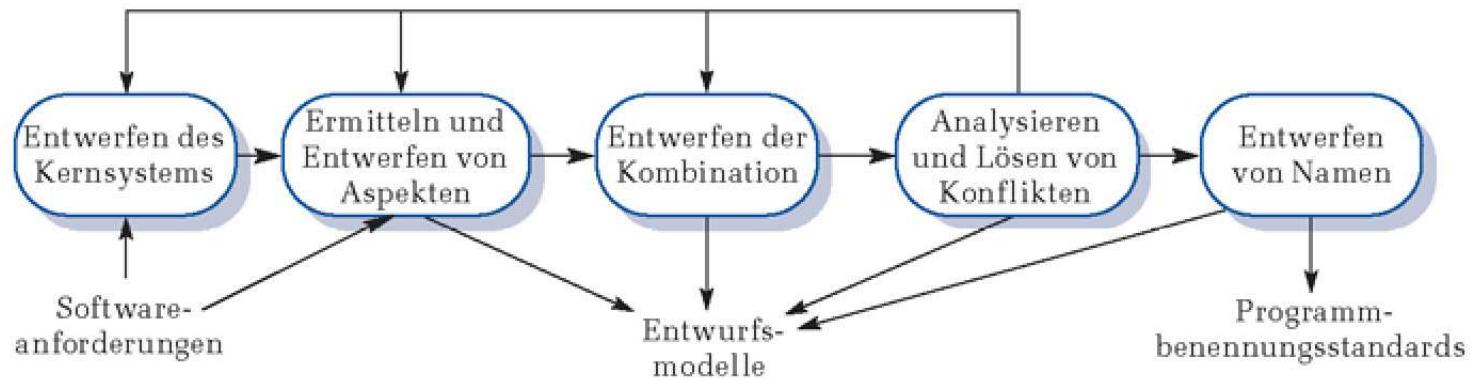
- Die Verwendung aspektorientierter Entwicklungsmethoden hat Konsequenzen für den Entwurfsprozeß.
- Bei der Use-Case-Modellierung können Aspekte z.B. als erweiternde Anwendungsfälle modelliert werden, die im Diagramm je nach Art des Join Points mit <<extends>> oder <<include>> gekennzeichnet werden.



Aspektorientiertes Design

1. Grundlagen
2. Vorgehensmodelle
3. Spezifikation
4. Entwurf
5. Entwicklung
6. Validierung
7. Standardsoftware
8. Frameworks
9. Service-orientierte Architektur
10. Entwurfsmuster
11. Aspektorientierte Entwicklung

Ein aspektorientierter Entwurfsprozeß:





- 1. Grundlagen**
- 2. Vorgehensmodelle**
- 3. Spezifikation**
- 4. Entwurf**
- 5. Entwicklung**
- 6. Validierung**
- 7. Standardsoftware**
- 8. Frameworks**
- 9. Service-orientierte Architektur**
- 10. Entwurfsmuster**
- 11. Aspekt-orientierte Entwicklung**

Verifikation und Validierung aspektorientierter Software

- Black-Box-Tests können für aspektorientierte Software durchgeführt werden wie für jede andere Software.
- White-Box-Tests und Programmminspektionen haben beim AOSD einige Besonderheiten:
 - Man sollte Tests entwerfen, die die Aspekte unabhängig von dem System testen, in das sie eingewoben werden.
 - Jeder Advice sollte einzeln testbar sein.
 - Die gegenseitige Störung von Aspekten muß ausgeschlossen werden, indem Programmabläufe erzeugt werden, bei denen die beteiligten Aspekte in allen möglichen Kombinationen und Reihenfolgen aufgerufen werden.
 - Ein Test sollte alle Join Points umfassen, die in dem System vorkommen.