

Softwaretechnik

Einführung



Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Basierend auf dem Material von Oscar Nierstrasz, Sven Apel, Janet Siegmund

Organisatorisches: Übersicht

- Durchführung:
 - Präsenzlehre: Prof. Dr.-Ing. Norbert Siegmund
 - Montags, 09:15 – 10:45 (HS1)
 - Dienstags, 09:15 – 10:45 (HS1)
 - Übung: M.Sc. Stefan Jahns; B. Sc. Annemarie Wittig
 - Mittwochs A, 11:15 – 12:45 (SG 1-27)
 - Mittwochs B, 13:15 – 14:45 (SG 1-27)
- Kursmaterial (gesamte Kommunikation über Moodle!):
 - Moodle: „Softwaretechnik WS24/26“ <https://moodle2.uni-leipzig.de/course/view.php?id=51684>
 - Folien, Podcasts, Diskussionen, Übungsaufgaben, Umfragen



Organisatorisches: Einordnung

- Einordnung:
 - Vorlesung: Theoretische Grundlagen, kleine Beispiele, Diskussionen, Live-Coding
 - Übung: Praktische Beispiele, Stoff orientiert sich an Erfahrungen und Problemen bei der Bearbeitung eines typischen, realen Softwareprojekts
 - **Keine** Pflichtabgaben in der Übung
- Stoff:
 - Ca. zwei Themen pro Woche bis Mitte Dezember
- Klausur:
 - 75 Minuten, 5 CLP

Vorlesung mit praktischer Relevanz!

Wissens- und Anwendungsvermittlung von Technologien, Prozessen und Konzepten, die direkt in der Praxis vorkommen

Industrievorlesungen zu ausgewählten Themen

Agile und Scrum in der Praxis



André Köhler
Professor für IT-Management
IU International Hochschule

Videovorlesung:
CI/CD + DevOPs +
Automatisierung



Jonas Hecht
Solutions Architect
Codecentric



KI in der
Softwareentwicklung
IT SONIX
Danny Hucke
AI Lead & SW Engineer
IT Sonix

Podcast (in Moodle) mit Industrievertretern zu
Themen in der Vorlesung + Aufzeichnungen vergangener
Industrievorlesungen

Ablauf einer Vorlesung

- (Optional)Java Programmierung: Quizz / Programmieraktivität
- Vorlesung
- Pause (5min... strikt)
- Vorlesung

Interaktion:

- Unbedingt mitmachen! ☺
- Fragen, wenn was nicht klar ist, undeutlich geschrieben steht, zu leise gesprochen wird, etc.



Lernziele für Heute

Verständnis über das Themengebiet Software Engineering erlangen

Bewusstsein über die Wichtigkeit des Software Engineerings verschaffen

Java-Wissen auffrischen

Erster Ausblick über ChatGPT & Co. und deren Einfluss auf die Softwareentwicklung erhalten



Was ist Software Engineering?



Software Engineering Mythen und falsche Vorstellungen

Alleine
arbeiten

SW-Entwicklung
ist nur coden...

Richtig ist

Teamwork und Kommunikation sind
essentiell für den Erfolg

Verständnis der Bedürfnisse der
Nutzerinnen ist unumgänglich

Abstraktion und Code-Qualität
gehen über Schnelligkeit und Hacks

Anpassbarkeit, Wartbarkeit und
Modularität sind Schlüsselfaktoren
für Erfolg



Softwaretechnikprojekt

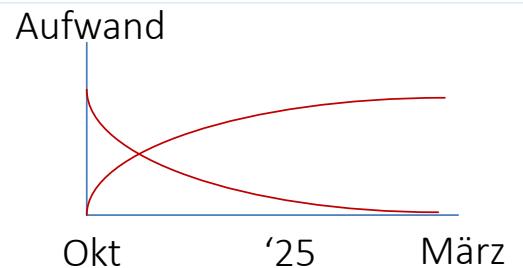


"Dieses Foto" von Unbekannter Autor ist lizenziert gemäß [CC BY](#)

Softwaretechnik + Praktikum

Softwaretechnik

Grundlagen der Softwareentwicklung
Phasen des Softwarelebenszyklus
Softwareentwicklungsprozess
Requirements Engineering
Architektur und Design
Softwareautomatisierung
Softwarequalität
Testing



Praktikum

Durchführung eines Softwareprojekts
Agile Softwareentwicklung im Team
Continuous Integration&Delivery
(Remote Softwareentwicklung)
Teamrollen und Softskills
Entwicklung in Sprints
[Systementwurf (C4)]



Industrievorträge

Thema: Digitalisierung der Essensbestellung und -ausgabe für Sozial-Arbeiten-Wohnen Borna gGmbH

- Kontext:

Sozial-Arbeiten-Wohnen Borna gGmbH unterstützt Menschen mit Behinderungen und deren Angehörigen. Ihr Ziel ist es, Menschen mit Behinderungen eine möglichst selbstständige und inklusive Teilhabe am gesellschaftlichen Leben zu ermöglichen. Dabei werden drei Werkstätten für Menschen mit Behinderungen an verschiedenen Standorten betrieben.
- Ziel des Projekts:
 - Digitalisierung des Bestellprozesses des Essens; Kontrolle und vereinfachte Ausgabe des Essens und dadurch eine Verringerung des Verwaltungsaufwandes, sowie Fehler im Zuge dieses Prozesses
- Lernziele:
 - Softwareprojekt mit echten Kunden, echten Anforderungen, echten Menschen durchlaufen
 - Reales Team- und Projektmanagement mit Abgabefristen
 - Echte SoftwareexpertInnen an Ihrer Seite zur Beratung
 - In der Praxis eingesetzte Frameworks und Technologien angewandt einsetzen



Echte Industrieexpertise!

\SSIST

Converneo: D I G I T A L



adesso

IT SONIX

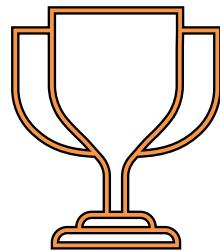
Die Teams werden 3-4x Konsultationen mit echten SW-EntwicklerInnen zum Projekt haben und so ein direktes Feedback aus Industriesicht erhalten.

Auszeichnung für die drei besten Teams



<https://www.optimax-energy.de/>

Erstmalig eine Auszeichnung für die 3 besten (vom Kunden) ausgewählten Lösungen.
Jedes Teammitglied der Siegerteams erhält einen Gutschein in Höhe von 50€.



Lernziele: Technologisch

- Kennenlernen von agiler Softwareentwicklung (Arbeiten im Team, direkt mit Kunden, mittels kleinen Sprints)
- Verwendung von Standardwerkzeugen der Softwareentwicklung (Versionsverwaltung, Issue Tracking System, CI/CD Pipeline)
- Erlernen, wie man sich in neue Frameworks und Sprachen einarbeitet
- Client-Server Architektur implementieren
- Kommunikation zwischen Microservices und REST-API in kleinen Rahmen herstellen
- Frontend(Webseiten)-entwicklung mit Serverkommunikation

Organisatorisches

- Industrievorlesungen, Kundengespräche, Präsentationen, Tutorials:
 - Freitag: 13:15 – 14:45 HS 6
 - Tipp: Geeignet auch für Gruppentreffen davor oder danach
 - Diese Woche: Erstes Kundengespräch (unbedingt wahrnehmen, damit Sie wissen, worum es geht)
- Kurs: <https://moodle2.uni-leipzig.de/course/view.php?id=45790>

Warum ist Software Engineering ~~wichtig~~
unabdingbar

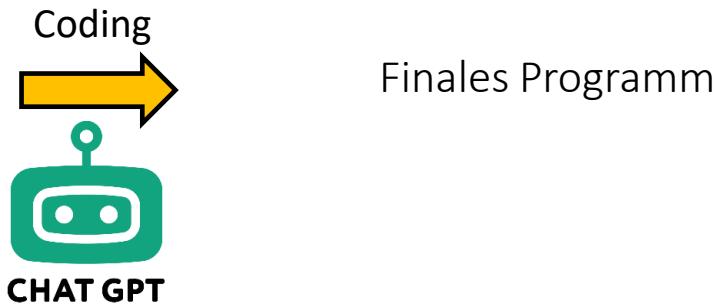
Software Engineering vs. Informatik

- Informatik beschäftigt sich mit der Theorie und den Grundlagen von Computersystemen
- Software Engineering beschäftigt sich mit praktischen Themen der Entwicklung und Auslieferung *guter* Software

Warum Software Engineering?

Naive Sicht:

Problemspezifikation



Aber:

- Woher kommt die **Spezifikation**?
- Wie korrespondiert die Spezifikation zu den **Nutzeranforderungen**?
- Wie entscheidet man, wie das Programm **strukturiert** wird?
- Wie weiß man, dass das Programm **tatsächlich den Spezifikationen entspricht**?
- Wie weiß man, dass das Programm immer **korrekt arbeitet**?
- Was macht man, wenn die Nutzeranforderungen **sich ändern**?
- Wie **teilt man Aufgaben auf**, falls man mehr als ein 1-Personen Team hat?

Warum scheitern Softwareprojekte?



Beispiele gescheiterte Projekte: Ariane 5

- Ariane 5 Flight 501: 4. Juni, 1996
- Selbstzerstörung nach 39 Sekunden
- ~ 500 million US-\$ Schaden
- Grund: Wiederverwendung von Referenzimplementierung der Ariane 4 ohne Anpassung auf neue Raketenspezifikation



Quelle(FU): <https://www.ima.umn.edu/~arnold/disasters/ariane.html>

Therac-25

- Medizinische Strahlentherapie
- Durch SW-Fehler zu hohe Strahlendosis, 3 Tote
- Grund: Ein einziger Entwickler schrieb Software und benutzte vorhandene Komponenten mit wenigen / keinen Tests

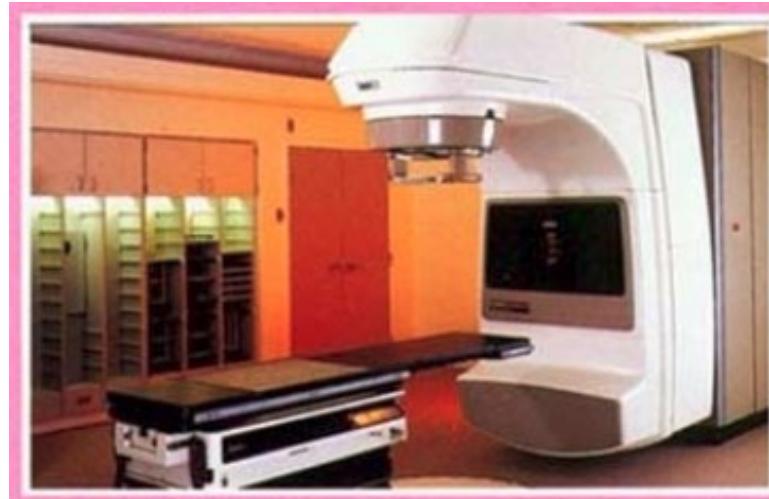


Image source (FU): <http://cr4.globalspec.com/blogentry/19025/Failure-of-the-Therac-25-Medical-Linear-Accelerator>

Apollo 11 PGNCS

- Während der Mondlandung:
 - Unerwartete “executive overflow” Alarms
 - 13% der Computerressourcen wurden durch einen Fehler im Radar verbraucht (5x Alarm + Neustart der Software)
 - Grund: Gleichzeitige Ermmittlung von Daten durch das Landeradar und des Rendezvousradars



Image source (PD): https://en.wikipedia.org/wiki/Apollo_11#Lunar_descent



“Morris Worm”

- Erster “Computervirus” – per Zufall
- Entworfen als ein Forschungswerkzeug zum Zählen der Computer im Internet (1989)
- Grund: Durch Bug verbreitete er sich selber
 - 10% des Internetverkehrs
 - Ein PC konnte mehrmals infiziert werden
- 10k-10.000k\$ Schaden

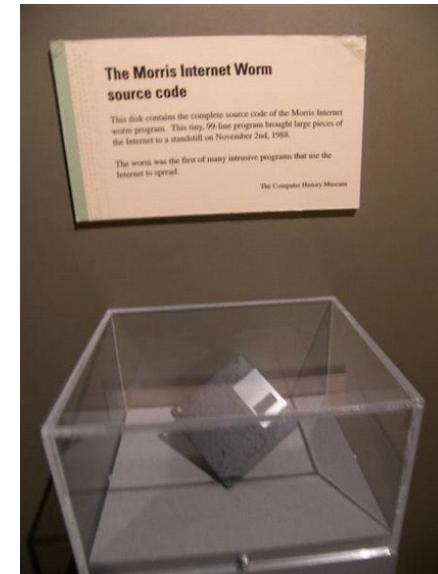


Image source (CC): https://en.wikipedia.org/wiki/Morris_worm#/media/File:Morris_Worm.jpg

Mars Climate Orbiter

- Verloren 1999 beim Anflug auf den Mars
- ~ 330 Millionen-US-\$ Schaden
- Grund: Einheitenfehler im Navi (metrisch vs. US-Skala)

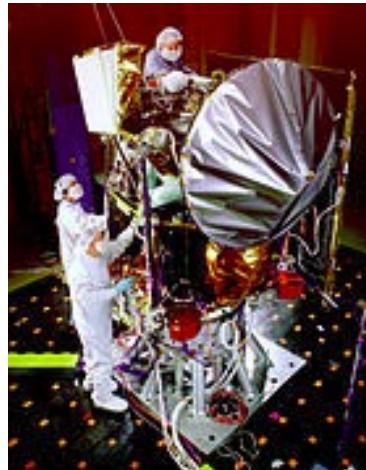


Image source (PD): https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

Mars Polar Lander

- Verloren bei der Marslandung 1999
- ~ 330 Millionen-US-\$ Schaden
- Grund: Fehler in der Software interpretierte Vibrationen bei der Landung als tatsächliche Landung und schaltete Triebwerke zu früh ab



Image source (PD): https://en.wikipedia.org/wiki/Mars_Polar_Lander

Patriot Missile Disaster

- Fehler im Zielsuchsystem führte zum Verfehlten der abzufangenden Rakete und somit zum Tod von 28 Menschen während des Golfkriegs 1991
- Grund: Fehlerhafte Zeitberechnung durch ungenaue arithmetische Berechnungen (Kommastellen wurden nach 24Bit abgeschnitten, was zu Ungenauigkeiten führte)



Image source (CC): https://en.wikipedia.org/wiki/MIM-104_Patriot

Corrupted Blood Incident

- Unbeabsichtigte “Seuche” in WoW, 2005, 1 Woche
- Wurde später bei Epidemiologen verwendet, um die Ausbreitung von Seuchen zu studieren
- Grund: Keine Tests bzw. Konzept für lokal abgeschirmte Bereiche von Effekten



Image source (FU): https://en.wikipedia.org/wiki/File:WoW_Corrupted_Blood_Plague.jpg

Hamburg-Altona Railway Switch

- Computer “upgrade” des Switch-Systems
- Ergebnis: Gesamter Bahnhof musste für 2 Tage schließen,
100 000s Passagiere betroffen
- Grund: Stackoverflow + Fehler im Code zum Behandeln des Overflows



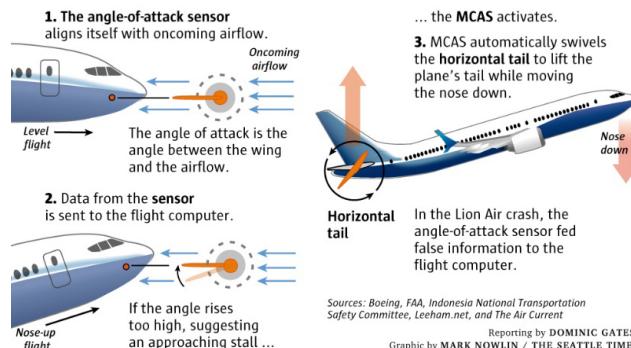
Image source (CC): https://en.wikipedia.org/.../media/File:Bahnhof_Hamburg_Altona.JPG

Boing 737 Max

- <https://www.seattletimes.com/business/boeing-aerospace/failed-certification-faa-missed-safety-issues-in-the-737-max-system-implicated-in-the-lion-air-crash/>
- Softwaresystem MCAS (Maneuvering Characteristics Augmentation System) hatte Sicherheitsprobleme und war abhängig von einem einzelnen Sensor
- 346 Menschen starben; >32\$ Mrd. Aktienverlust, >8\$ Mrd. Kosten



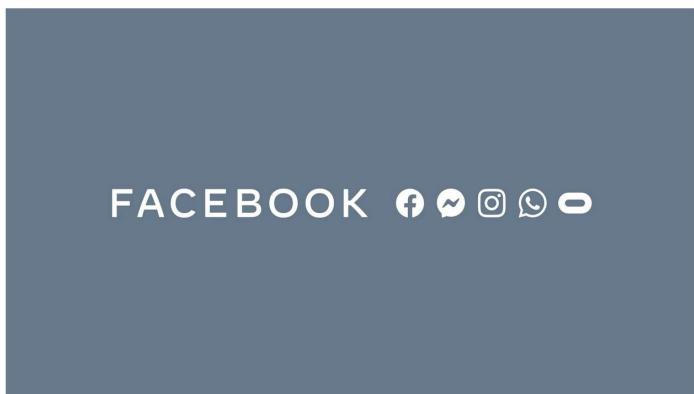
How the MCAS (Maneuvering Characteristics Augmentation System) works on the 737 MAX



Facebook Outage

POSTED ON OCTOBER 4, 2021 TO NETWORKING & TRAFFIC

Update about the October 4th outage



By Santosh Janardhan



To all the people and businesses around the world who depend on us, we are sorry for the inconvenience caused by today's outage across our platforms. We've been working as hard as we can to restore access, and our systems are now back up and running. The underlying cause of this outage also impacted many of the internal tools and systems we use in our day-to-day operations, complicating our attempts to quickly diagnose and resolve the problem.

Our engineering teams have learned that configuration changes on the backbone routers that coordinate network traffic between our data centers caused issues that interrupted this communication. This disruption to network traffic had a cascading effect on the way our data centers communicate, bringing our services to a halt.

- Facebooks Border Gateway Protocol Server erhielten ein Update mit einem Konfigurationsfehler, welcher dafür sorgte, dass alle Verbindungen von allen Datencenter von Facebook unterbrochen wurden.
- Dies führte zu weiteren Effekten, welche praktisch alle Facebook Dienste vom Rest des Internets abtrennten

So what happened to Facebook?

It turns out that BGP played a part in Facebook's issues but wasn't the root cause. In its [detailed explanation](#), released on Tuesday, the company says that a command issued as part of routine maintenance accidentally disconnected all of Facebook's data centers (oops!). When the company's DNS servers saw that the network backbone was no longer talking to the internet, they stopped sending out BGP advertisements because it was clear that something had gone wrong.

To the wider internet, this looked like Facebook telling everyone to take its servers off their maps. Cloudflare's CTO reported that the service saw a ton of BGP updates from Facebook (most of which were route withdrawals or erasing lines on the map leading to Facebook) right before it went dark. One of Fastly's tech leads tweeted that [Facebook stopped providing routes to Fastly](#) when it went offline, and [KrebsOnSecurity backed up the idea](#) that it was some update to Facebook's BGP that knocked out its services.

Crowdstrike Bug 2024

- Ein fehlerhaftes Update der *Sicherheitsfirma* Crowdstrike sorgte dafür, dass 8,5 Millionen Windowsrechner in einer Endlosschleife von BlueScreen und Reboot unbrauchbar waren
- Chaos in Reisebranche und Gesundheitssystem; 1,5 Mrd\$ Cyberversicherungsschaden; 5,4 Mrd\$ Verluste für Fortune-500-Unternehmen
- Update von Antivirus-Signatur wurde **ungetestet** auf der ganzen Welt ausgerollt



Warum scheitern Softwareprojekte?

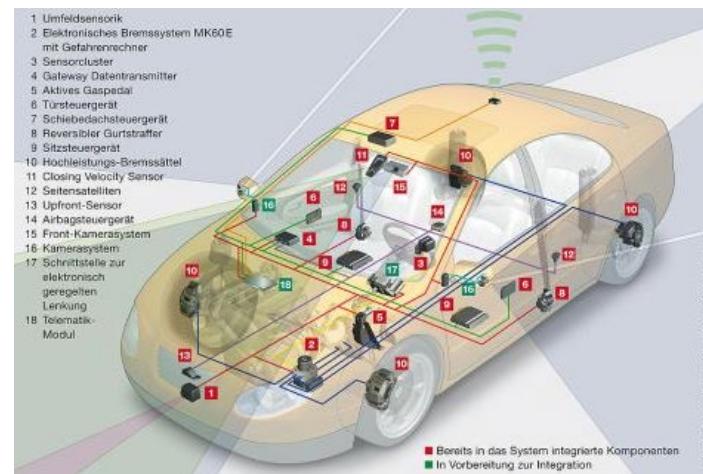
Budget zu klein oder aufgebraucht
Zeit aufgebraucht
Ineffizienz von Software
Schlechte Qualität von Software
Software erfüllt Anforderungen nicht
Projekte waren nicht mehr zu managen
Quelltext schwierig zu warten
Software wurde nie ausgeliefert



Warum scheitern SW-Projekte nicht?

Technologie unklar
Fehlende Programmierfähigkeiten
Kein Code produziert
10X ProgrammiererInnen fehlen
UML-Diagramm ist falsch

Lösung: Software Engineering



Begriff: Software Engineering

- Begriff wurde auf der NATO-Konferenz (1968) geprägt
- Idee: Softwareentwicklung in Anlehnung an Ingenieursdisziplin (Engineering)
- Im Folgenden: Definitionen
 - Geben einen Einblick über verschiedene Sichtweisen auf diese Disziplin

Definition: Software Engineering

„state of the art of developing quality software on time and within budget“

Aktuellster Stand der Technik entscheidet Community und erfordert Lebenslanges lernen
Ressourcenbeschränkung

„multi-person construction of multi-version software“ -- Parnas

Teamwork

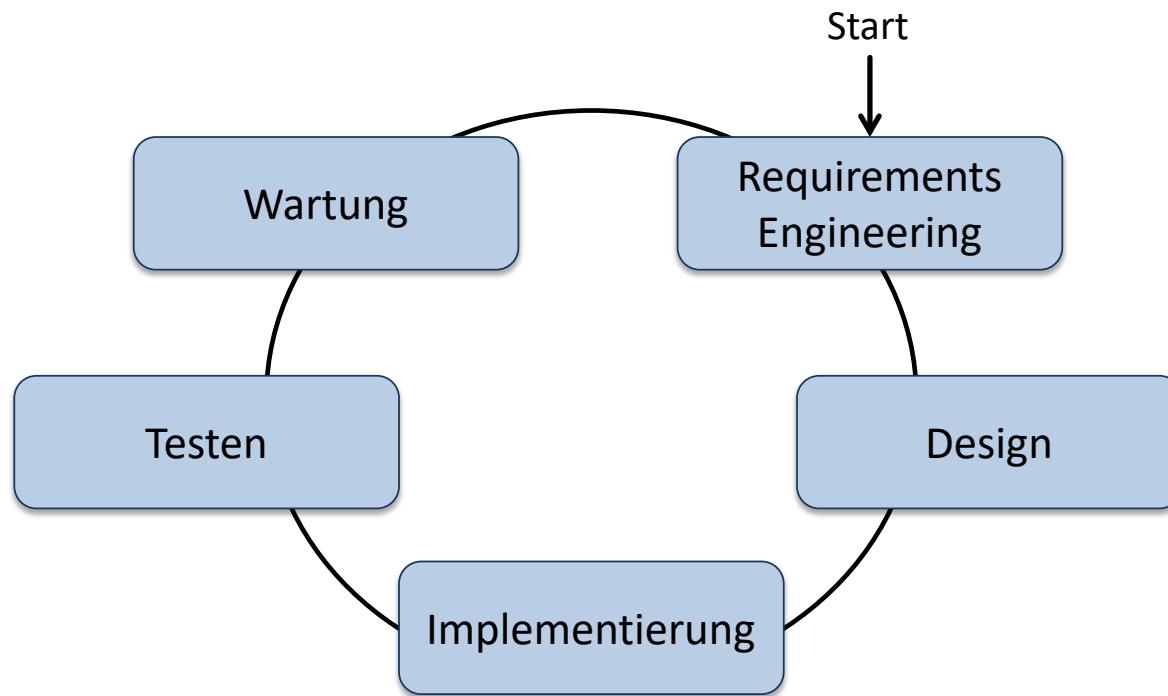
Erfolgreiche Software-Systeme müssen sich weiterentwickeln: Änderung ist der Normalfall, nicht die Ausnahme

„software engineering is an engineering discipline that is concerned with all aspects of software production“ -- Sommerville

Software ist ein Produkt, das für einen bestimmtem Kunden entwickelt wird
Nicht nur Programmierung, sondern alle Aspekte des Softwarelebenszyklus



Softwarelebenszyklus



Programmierung ist nur ein kleiner Teil von Software-Entwicklung

Large Language Models (LLMs): ChatGPT & Co.

Was sind Ihre Gedanken hierzu?

„Gute“ Software

- Maintainable (Wartbar)
- Dependable / Reliable (Verfügbar, Zuverlässig)
- Efficient (Effizient)
- Usable (Nutzbar, Anwendbar)

Die Grenzen von ChatGPT & Co.: Fundamentale Prinzipien und Konzepte der Softwareentwicklung

Prinzipien

Divide and conquer
Simplicity
Rigorousness

Konzepte

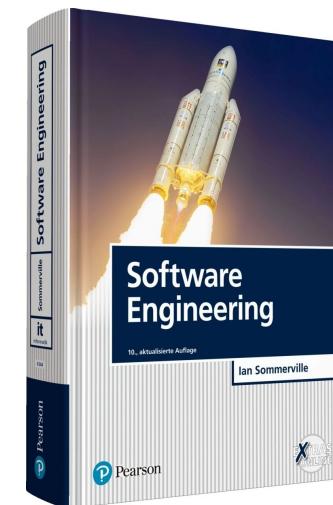
Modularität und Struktur
Abstraktion und Generalisierung
Design for change
Separation of concerns
Stepwise refinement
Information hiding

Zusammenfassung

- Grundlegende Vorstellung von Software Engineering haben
- Notwendigkeit des Software Engineerings erkennen
- Einordnung des Anteils der Programmierung an der Entwicklung von Software

Literatur

- *Software Engineering*. Ian Sommerville. Addison-Wesley Pub Co; ISBN: 020139815X, 10th edition, 2017
- *Software Engineering: A Practitioner's Approach*. Roger S. Pressman. McGraw Hill Text; ISBN: 0072496681; 5th edition, 2001
- *Using UML: Software Engineering with Objects and Components*. Perdita Stevens and Rob J. Pooley. Addison-Wesley Pub Co; ISBN: 0201648601; 1st edition, 1999
- *Designing Object-Oriented Software*. Rebecca Wirfs-Brock and Brian Wilkerson and Lauren Wiener. Prentice Hall PTR; ISBN: 0136298257; 1990
- Aber:
 - Kein einzelnes Buch für den gesamten Stoff der Vorlesung
 - Zu jedem Thema gibt es Empfehlungen
 - Vieles auch im Internet gut nachzulesen



Was Sie mitgenommen haben sollten

- Was ist Software Engineering?
- Nennen/Erklären Sie X mögliche Gründe für das Scheitern von Software-Projekten
- Was ist die Softwarekrise? Warum trat sie auf?
- Nennen/Erklären Sie die Prozesse im Software-Lebenszyklus.
- Was sind die Gemeinsamkeiten/Unterschiede von Software Engineering zur klassischen Ingenieursdiziplin?

Softwaretechnik

Requirements Engineering

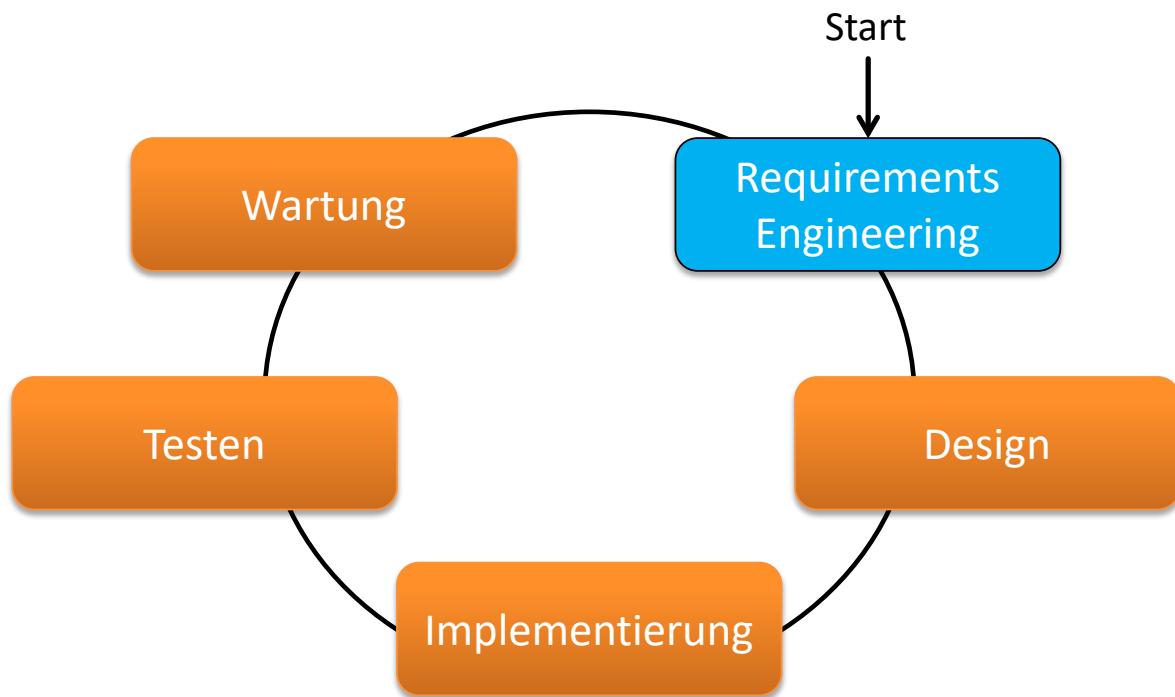


Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Einordnung



Lernziele

- Notwendigkeit von Requirements Engineering verstehen
- Typische Probleme bei der Anforderungsanalyse kennen
- Vorgehen für systematisches Finden von Anforderungen verstehen
- Anforderungen beschreiben können

Warum Requirements Engineering?



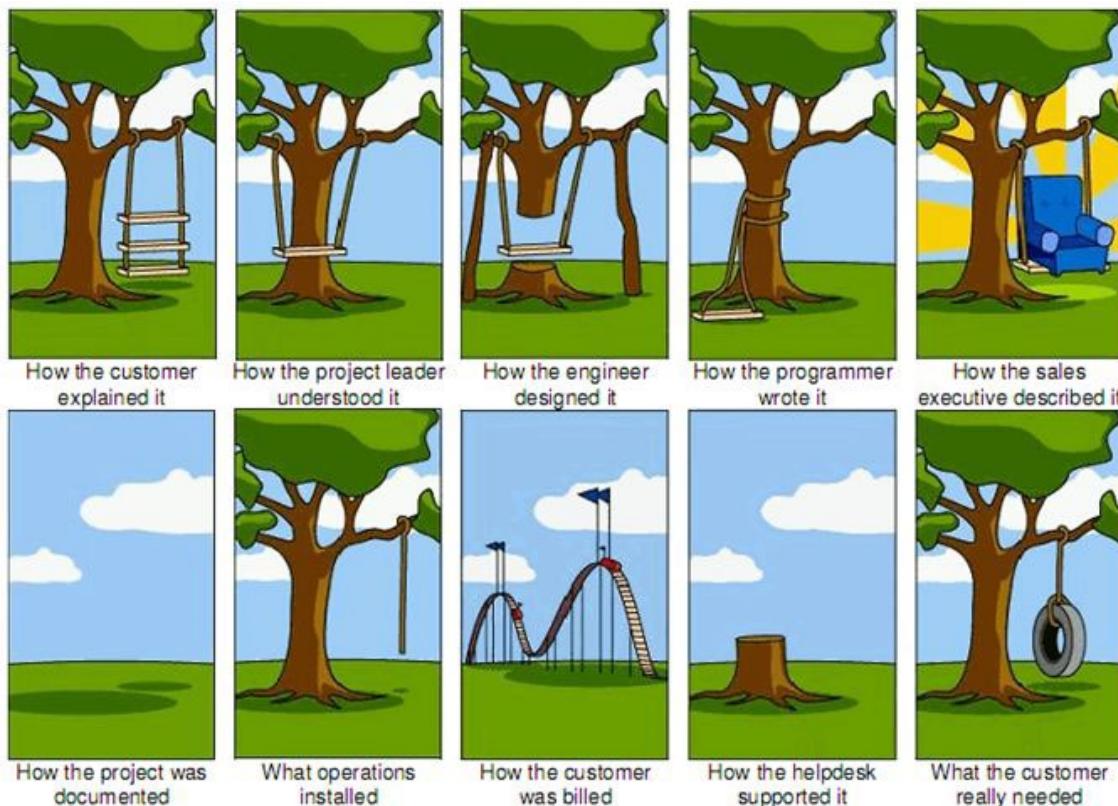
Was sind Requirements?

- Bedingung oder Eigenschaft, die ein System benötigt,
 - um ein Problem zu lösen
oder
 - um ein Ziel zu erreichen
oder
 - um einem Vertrag, Standard oder Ähnlichem zu genügen
- [Pohl. Requirements Engineering]

Was sind Requirements?

- Werden an ein Programm gestellt
- Spezifizieren:
 - Eigenschaften
 - Funktionalität
 - Einsatzsszenario
 - Qualität
- Werden von Stakeholdern bestimmt

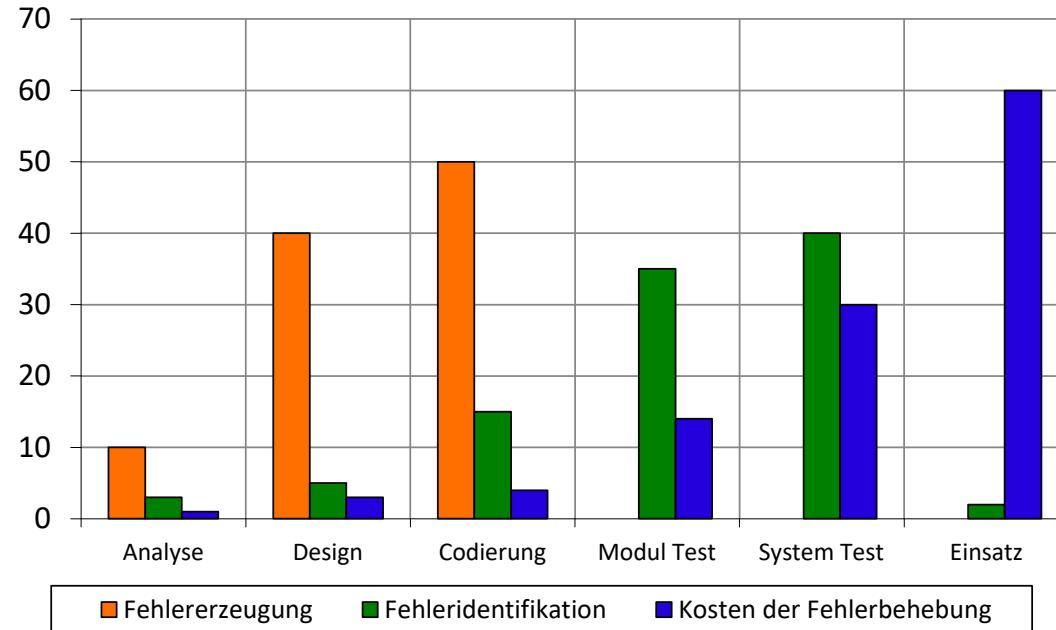
Warum Requirements Engineering?



[Quelle: <http://www.interface-gmbh.de/RequirementsEngineering.htm>]

Warum Requirements Engineering?

- Bauen wir das Richtige?



Überprüfbarkeit

- Szenario: Neue Firma, 3 Mitarbeiter (Gehalt 45.000 EUR)
- Bekommen Auftrag von Firma \$BigCooperation
 - Geschätzter Aufwand: 9 Personenjahre
 - Festpreis 500.000 EUR, 250.000 EUR sofort, Rest nach Abnahme
- Fertigstellung nach 3 Jahren
- Firma verweigert Abnahme, fordert Nacharbeiten

Probleme



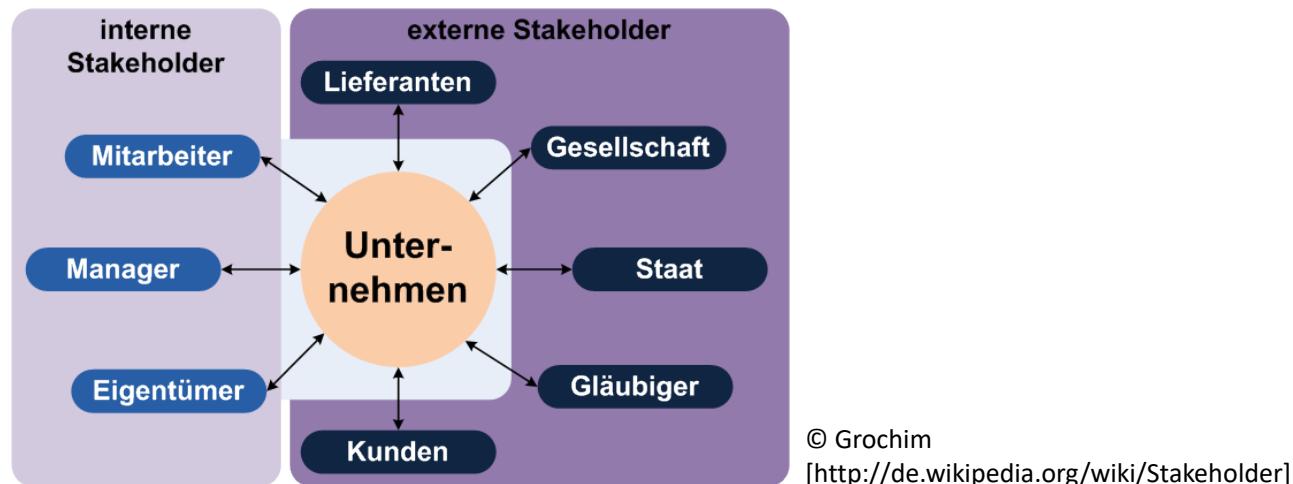
© Scott Adams, Inc./Dist. by UFS, Inc.

Probleme (2) -- Kunden

- Kunden wissen nicht, was sie wirklich wollen
 - Und sie können es oftmals auch noch nicht wissen, weil sie keine IT-Expertinnen sind
- Kunden benutzen ihre eigene Fachsprache
 - Abkürzungen, Fachbegriffe, Gesetzeslagen, implizites Wissen; all das ist relevant für die zu entwickelnde Software
- Politische und organisatorische Faktoren können Anforderungen beeinflussen
 - Datenschutzrichtlinien, Fairness und Ethik, Organisationsbereiche mit unterschiedlichen Befugnissen; all das ist relevant für die zu entwickelnde Software

Probleme (3) -- Widersprüche

- Verschiedene Stakeholder können widersprüchliche Anforderungen haben
- Neue Stakeholder mischen sich ein



© Grochim
[<http://de.wikipedia.org/wiki/Stakeholder>]

Problem (4) -- Evolution

- “Requirements *always evolve*”
 - Durch besseres Verständnis darüber was der Kunde wirklich braucht
 - Durch Änderungen in den Zielen der Organisation
- Daher wichtig: “*plan for change*” in den Requirements



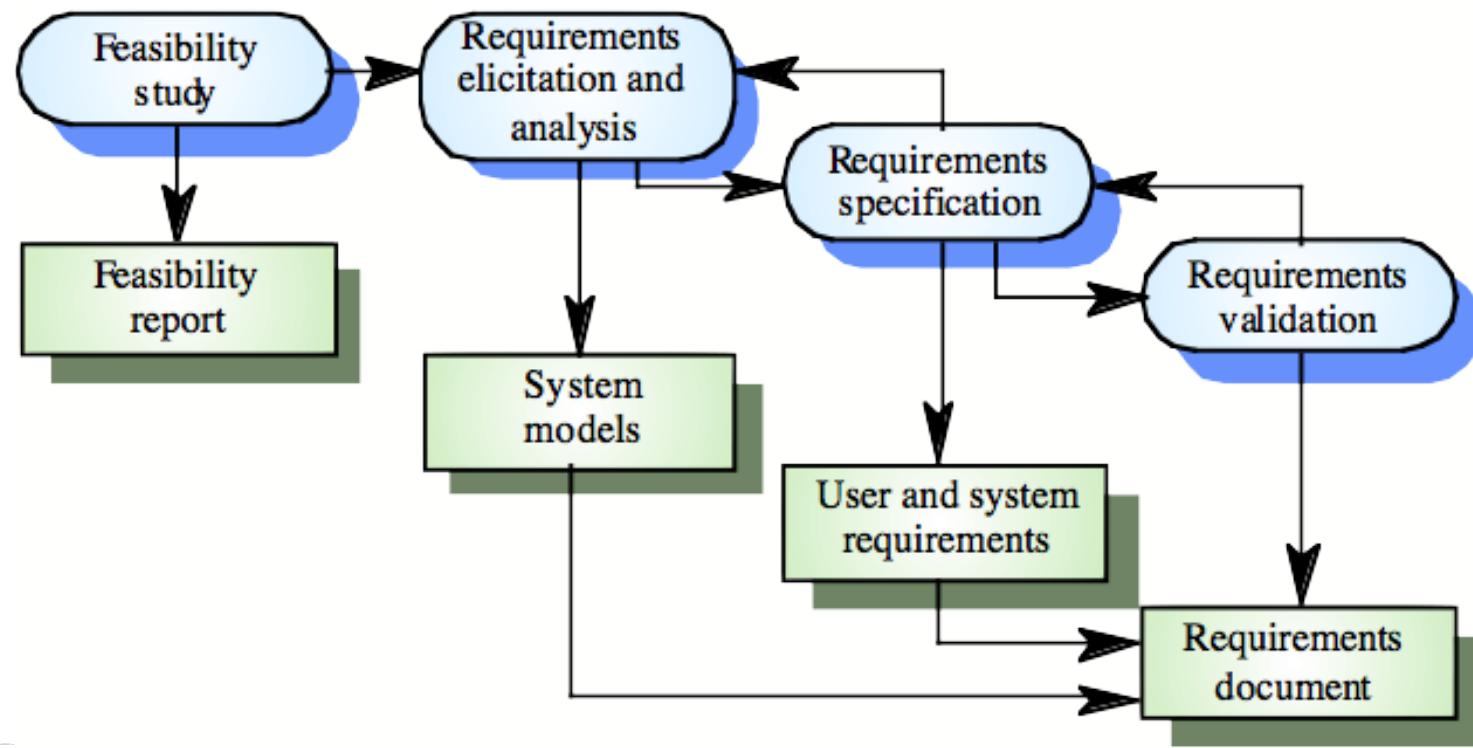
Source: <https://www.cse.msu.edu/~zhangy72/netflix/>

Wie funktioniert Requirements Engineering?



Requirements-Engineering Prozess

- 4 Schritte mit jeweils einer Aktivität



4 Aktivitäten

Feasibility study	Bestimmen ob <i>Nutzeranforderungen erfüllt</i> werden können mit der <i>verfügbaren Technologie</i> und <i>Budget</i> .
Requirements elicitation & analysis	Herausfinden <i>was Kunden</i> vom System <i>benötigen</i> .
Requirements specification	<i>Spezifizierte die Anforderungen</i> in einer Form die der Kunde versteht und als eine Art Vertrag zwischen Kunden und Anbieter.
Requirements validation	<i>Überprüfen der Anforderungen</i> auf Realisierbarkeit, Konsistenz und Vollständigkeit.

“Requirements sind für Kunden; Spezifikationen sind für Analysten und Entwickler”

4 Schritte

1. Anforderungsermittlung
 - Sammeln von Anforderungen
 - z.B. durch Anwendergespräche, Dokumenten-Studium
2. Anforderungsanalyse
 - Klassifizierung, Bewertung, Vergleich und Prüfung
 - z.B. Kosten-/Nutzen-Aspekte, Konsistenz, Vollständigkeit
3. Anforderungsbeschreibung
 - Beschreibung in einheitlicher Form (z.B. als Anwendungsfälle)
4. Anforderungsrevision
 - Erneute Prüfung/Änderung von Anforderungen
 - ggf. nur in formellem Änderungsverfahren



1. Anforderungsermittlung

- Stakeholder-basiert:
 - Identifikation von Stakeholdern
 - Gespräche mit allen Stakeholdern
- Szenario-basiert:
 - Szenario: konkreter, fiktiver (Arbeits-) Ablauf eines Systems
 - Erstellen von allen relevanten Szenarien
 - (Bekannt aus der Modellierungsvorlesung)

Beispiel: NoMoreWaiting (NMW)

Die Studierenden der Universität Leipzig sind mit dem Problem konfrontiert, dass sie zur Mittagszeit (12:00) keinen Tisch in der Mensa finden. Dies liegt daran, dass zu bestimmten Stoßzeiten besonders viele Studierende Essen gehen, die Mensa jedoch nur eine beschränkte Zahl von Sitzplätzen aufweist.

Findige Studierende der Lehrveranstaltung „Softwaretechnik“ möchten aus diesem Grund eine App entwickeln, mit der Studierende einen Sitzplatz zu einer bestimmten Zeit für sich reservieren können. Damit dies auch in der Praxis funktioniert, werden in der Mensa Tische ausgewiesen, die ausschließlich von Nutzern der Mensa-App genutzt werden dürfen. Die Anzahl der ausgewiesenen Tische kann vom Mensapersonal entsprechend der Reservierungen vorab angepasst werden.

Die App kann kostenlos aus dem Market heruntergeladen werden. Für die Nutzung der App ist es erforderlich, dass sich jeder Nutzer einen Account mit seiner Matrikelnummer anlegt. Ob die Matrikelnummer auch wirklich existiert, wird über das zentrale Hochschulverwaltungssystem der Universität geprüft.

Sobald die Registrierung abgeschlossen wurde, können Nutzer damit beginnen, Plätze zu reservieren. Plätze können am Tag der Nutzung ab 09:00 reserviert werden. Dabei kann der Nutzer über eine Karte, die die Anordnung der Tische in der Mensa zeigt, wählen, welchen Platz er gern hätte. Reservierungen sind immer nur für 20 Minuten gültig und sind aufgeteilt in Slots. In einer Stunde kann ein Platz also für drei Slots vergeben werden.

In der Mensa müssen Nutzer auf ihrem Platz „einchecken“. Dafür scannen sie mit ihrem Smartphone einen am Platz fixierten QR-Code. Über das Einchecken werden Statistikdaten gesammelt. So wird für jeden Nutzer ermittelt, wie oft er einen Platz reserviert, diesen dann aber nicht in Anspruch nimmt. Nutzer, die reservierte Plätze häufig nicht in Anspruch nehmen, werden über ein Credit-System bestraft. Entsprechend der Credit-Zahl der Nutzer müssen diejenigen Nutzer mit wenigen Credits bei der Reservierung einen oder gar mehrere Tage aussetzen.

Stakeholder für NMW

- Studierende, die App benutzen
- Studierende, die App nicht benutzen
- Mensa-Angestellte
- Datenschutzbeauftragte
- Mitarbeiter

Szenarien für NMW

- "Ich mache mich JETZT mit meinen X Freunden auf den Weg. Das System soll einfach und schnell Plätze finden und reservieren."
- „Es soll möglich sein, Plätze sehr einfach wieder freizugeben.“

So, what about AI?



<https://chat.openai.com/share/52cf2fe2-4e92-46d7-9813-18f1d2708abb>

Aufgabe

Was sind wesentliche Anforderungen für NoMoreWaiting?

Anforderungen für NoMoreWaiting

- Stakeholder-basiert
 - Schnell
 - Einfach
 - Übersichtlich
 - Schnelle Abmeldung
 - Rückmeldung über Credits und Reservierungsstatus
 - Mitarbeiter der Mensa sollen es einfach handeln können
 - Nichtbenutzer sollen nicht eingeschränkt werden
 - Mehrfachbuchen soll nicht möglich sein
 - Datenschutz (anonym)
 - Student soll Platz einfordern können, wenn andere reservierten Platz benutzen
 - Gruppenreservierung
- Szenario-basiert
 - Reservieren:
 - Kostenlos
 - Freigeben
 - System muss wissen, dass Platz frei ist
 - Registrierung
 - Ohne Matrikelnummer?
 - Fake-Reservierung?
 - Schnittstelle zur Univerwaltung zum Validieren von MatNr.
 - Sammeln
 - Verfälschung von Statistiken bei Gruppenbenutzung verhindern
 - Feedback
 - Andere müssen Gruppenreservierung annehmen
 - Verschlüsselung der Daten
 - Max. Platzbeschränkung bei Reservierung

Stakeholder- oder szenariobasiert?

- Stakeholder-basiert:
 - Vorteil: Anforderungen werden nach Personen gebündelt aufgenommen
 - Nachteil: Nutzen/Sinn/Ziel des Systems tritt in den Hintergrund
- Szenario-basiert:
 - Vorteil: Anforderungen orientieren sich an den „normalen“ Abläufen des Systems
 - Nachteil: Seltene Abläufe oder indirekt betroffene Institutionen werden leichter vergessen

Use Cases und Szenarien

- Ein use case ist die *Spezifikation* von einer *Sequenz von Aktionen*, einschließlich *Varianten*, dass ein System, *interagierend mit Aktoren* des Systems, ausführen kann".
 - Beispiel: kaufen einer DVD im Internet
- Ein Szenario ist ein *bestimmter Pfad von auftretenden Aktionen*, startend von einem bekannten, initialen Zustand.
 - Beispiel: Verbinde zu myDVD.com, Gehe zur "search" page; Gebe eine "...", ...

Unified Modeling Language

UML ist der Industriestandard für die Dokumentierung Objekt-orientierter Modelle

<i>Class Diagrams</i>	visualize <i>logical structure</i> of system in terms of <i>classes, objects, and relationships</i>
<i>Use Case Diagrams</i>	show external <i>actors and use cases</i> they participate in
<i>Sequence Diagrams</i>	visualize <i>temporal message ordering</i> of a <i>concrete scenario</i> of a use case
<i>Collaboration (Communication) Diagrams</i>	visualize <i>relationships</i> of objects exchanging messages in a <i>concrete scenario</i>
<i>State Diagrams</i>	specify the <i>abstract states</i> of an object and the <i>transitions</i> between the states

Use Case Diagramme

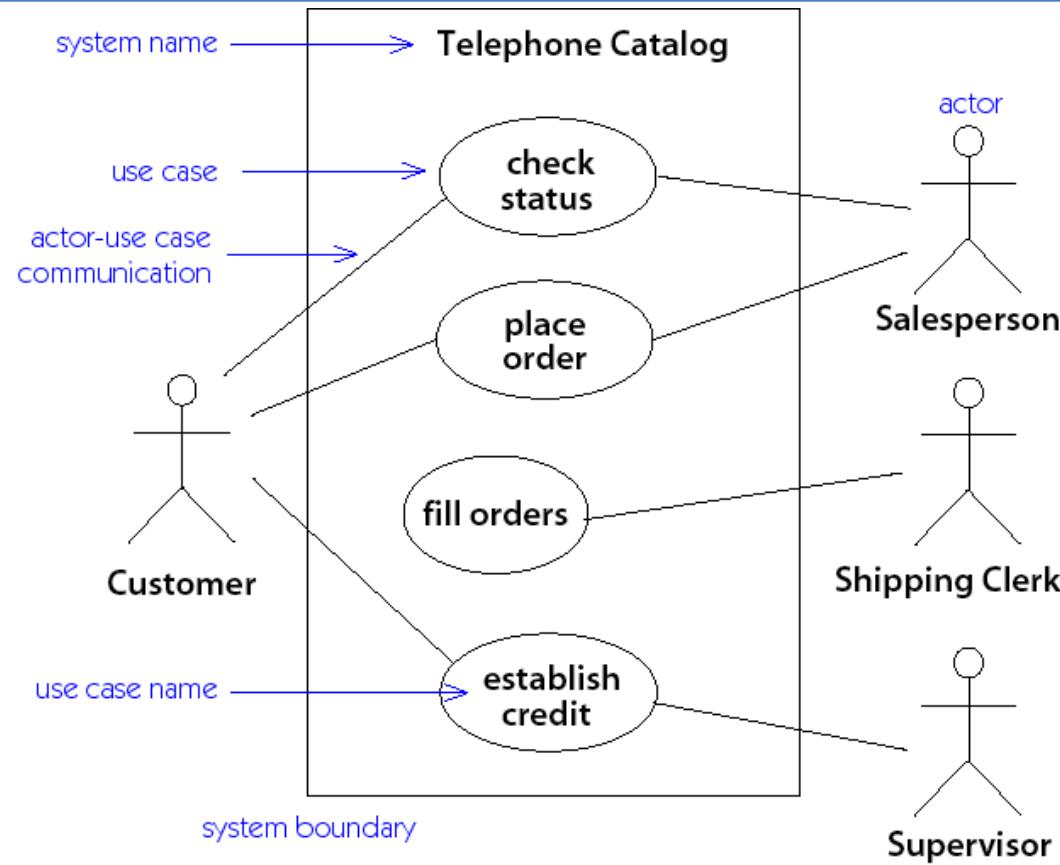


Figure 5-1. Use case diagram

Softwaretechnik – Prof. Dr.-Ing. Norbert Siegmund

Sequenzdiagramme

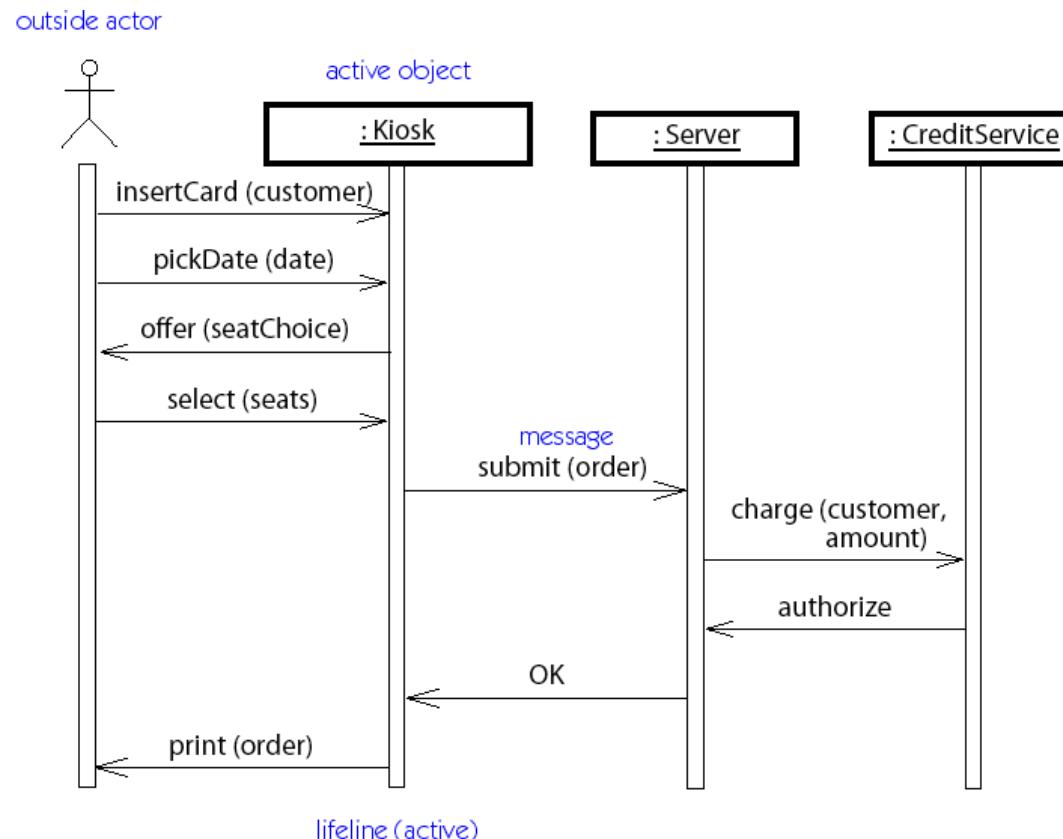


Figure 8-1. Sequence diagram

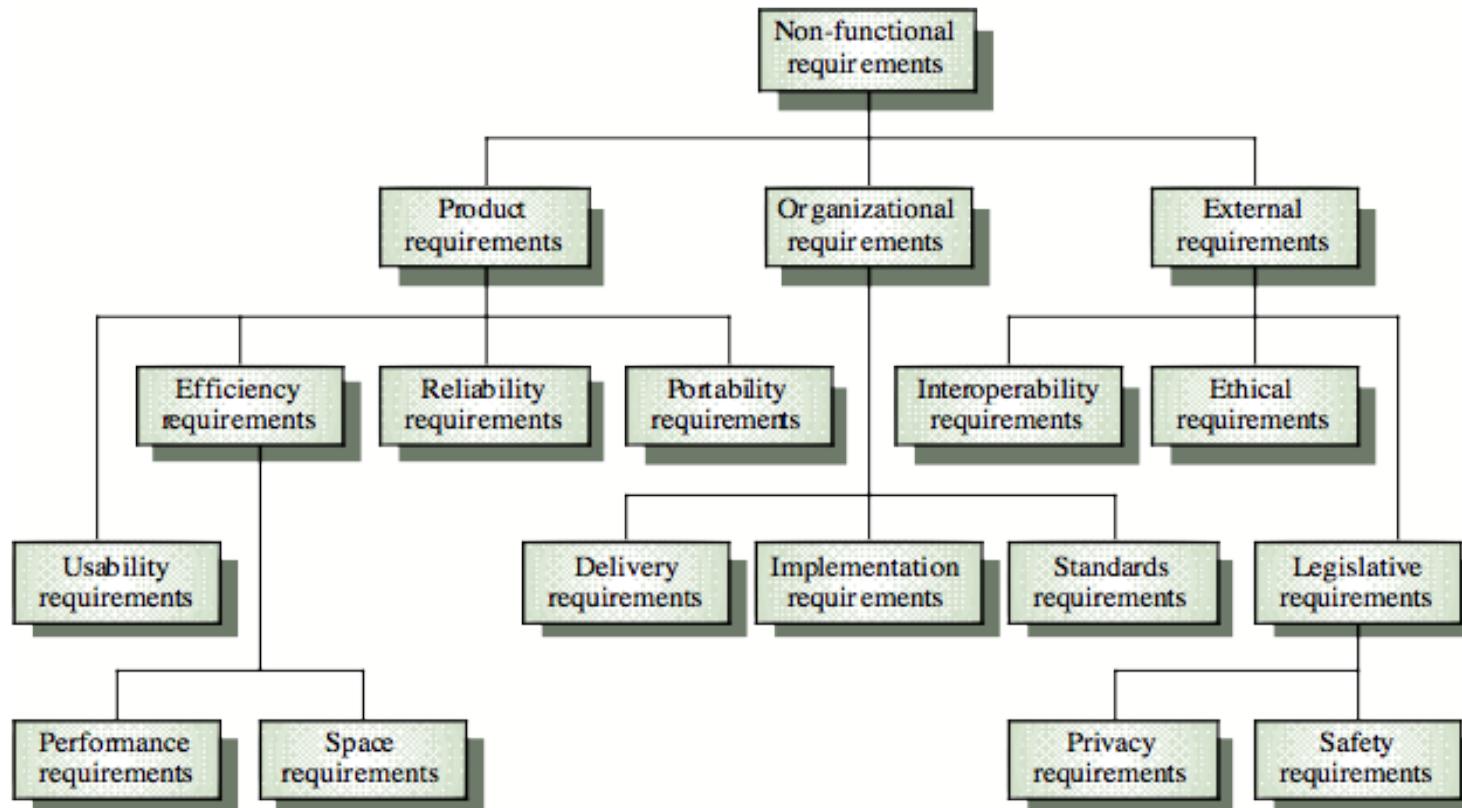
Softwaretechnik – Prof. Dr.-Ing. Norbert Siegmund

Pause

2. Anforderungsanalyse

- Funktionale Anforderungen:
 - *Was* soll das System leisten?
 - Welche Dienste soll es anbieten?
 - Eingaben, Verarbeitungen, Ausgaben
 - Verhalten in bestimmten Situationen, ggf. was soll es explizit nicht tun
- Nicht-funktionale Anforderungen:
 - *Wie* soll das System/einzelne Funktionen arbeiten?
 - Qualitätsanforderungen wie Performance und Zuverlässigkeit
 - Anforderungen an die Benutzbarkeit des Systems

Arten von Nicht-Funkt. Anforderungen



2. Anforderungsanalyse

- Stakeholder- oder szenario-basiert?
 - Funktionale Anforderungen: Szenario-basiert
 - Nicht-funktionale Anforderungen: Stakeholder-basiert
- Widersprechen sich Anforderungen?
- Sind Anforderungen konsistent?
- Sind Anforderungen umsetzbar?

Erfüllbarkeit von Anforderungen

- Generell: Anforderungen sollten so formuliert werden, dass sie objektive verifiziert werden können
- Unpräzise (Negativbeispiel):
 - Das System sollte von erfahrenen Kontrolleuren *einfach zu benutzen* sein und sollte so organisiert sein, dass *Benutzerfehler minimiert* werden.
 - Ausdrücke wie „einfach zu benutzen“ sind sinnlos
- Verifizierbar (Positivbeispiel):
 - Erfahrene Kontrolleure sollten in der Lage seien alle Funktionen des Systems *nach einem 2-stündigen Training* nutzen zu können. Die *mittl. Anzahl* von getätigten Fehlern sollte nach dem Training *nicht höher als 2 pro Tag* sein.

Präzise Metriken

<i>Eigenschaft</i>	<i>Metrik</i>
<i>Performanz</i>	Processed transactions/second User/Event response time Screen refresh time
<i>Größe</i>	K Bytes; Number of RAM chips
<i>Handhabbarkeit</i>	Training time Rate of errors made by trained users Number of help frames
<i>Zuverlässigkeit</i>	Mean time to failure Probability of unavailability Rate of failure occurrence
<i>Robustheit</i>	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
<i>Portabilität</i>	Percentage of target dependent statements Number of target systems

Beispiele

- Das System soll zügig reagieren
 - 80 % aller Anfragen sollen unter 0.1 Sekunde beantwortet werden, 99.99 % aller Anfragen unter 2 Sekunden
 - Test-Hardware spezifizieren!
- Das System soll auch im Mehrbenutzerbetrieb schnell reagieren
 - Auf XY-Server Verarbeitung von 200 Anfragen pro Sekunde von 10 unterschiedlichen Systemen mit 100mbit Ethernet-Anbindung
- Das System soll stabil und ausfallsicher sein
 - Verfügbarkeit von 99.99 %, Neustart innerhalb von 30 Sekunden, das System darf nicht aufgrund falscher Eingaben abstürzen

Aufgabe

- Welche der gefundenen Anforderungen sind funktional, welche nicht-funktional?
- Sind alle Anforderungen umsetzbar, gibt es Widersprüche?

Stakeholder-basiert

- Schnell
- Einfach
- Übersichtlich
- Schnelle Abmeldung
- Rückmeldung über Credits und Reservierungsstatus
- Mitarbeiter der Mensa sollen es einfach handeln können
- Nichtbenutzer sollen nicht eingeschränkt werden
- Mehrfachbuchen soll nicht möglich sein
- Datenschutz (anonym)
- Student soll Platz einfordern können, wenn andere reservierten Platz benutzen
- Gruppenreservierung

Szenario-basiert

- Reservieren:
 - Kostenlos
- Freigeben
 - System muss wissen, dass Platz frei ist
- Registrierung
 - Ohne Matrikelnummer?
 - Fake-Reservierung?
 - Schnittstelle zur Univerwaltung zum Validieren von MatNr.
- Sammeln
 - Verfälschung von Statistiken bei Gruppenbenutzung verhindern
- Feedback
 - Andere müssen Gruppenreservierung annehmen
- Verschlüsselung der Daten
- Max. Platzbeschränkung bei Reservierung

Anforderungen für NoMoreWaiting

- Stakeholder-basiert
 - Schnell (nf)
 - Einfach (nf)
 - Übersichtlich (nf)
 - Schnelle Abmeldung (beides)
 - Rückmeldung über Credits und Reservierungsstatus (eher f)
 - Mitarbeiter der Mensa sollen es einfach handeln können (nf)
 - Nichtbenutzer sollen nicht eingeschränkt werden (f)
 - Mehrfachbuchen soll nicht möglich sein (f)
 - Datenschutz (anonym) (nf)
 - Student soll Platz einfordern können, wenn andere reservierten Platz benutzen (f)
 - Gruppenreservierung (f)
 - Internetreservierung, nicht nur als App (f)
- Szenario-basiert
 - Reservieren:
 - Kostenlos (nf)
 - Freigeben
 - System muss wissen, dass Platz frei ist
 - Registrierung
 - Ohne Matrikelnummer?
 - Fake-Reservierung?
 - Schnittstelle zur Univerwaltung zum Validieren von MatNr.
 - Sammeln
 - Verfälschung von Statistiken bei Gruppenbenutzung verhindern
 - Feedback
 - Andere müssen Gruppenreservierung annehmen
 - Verschlüsselung der Daten
 - Max. Platzbeschränkung bei Reservierung

3. Anforderungsbeschreibung

- Anforderungen müssen systematisch und einheitlich beschrieben werden
 - Gleiches Vorgehen (nichts vergessen)
 - Überprüfbar
 - Gleiches Schema für alle Anforderungen anwenden
- Beispiele:
 - Volere-Template (Snow card)
 - IEEE Std 830-1998
 - FURPS/FURPS+

Volere

- 5 Hauptkategorien, in denen Anforderungen genau beschrieben werden
 - Project Drivers, Project Constraints, Functional Requirements, Non-Functional Requirements, Project Issues
- Eingeteilt in Subkategorien

Volere: Project Drivers

- Warum machen wir das Projekt?
 - Zweck des Projekts
 - Stakeholder

Volere: Project Constraints

- Welche (gesetzlichen) Rahmenbedingungen müssen eingehalten werden?
 - Einschränkungen
 - Namenskonventionen und Terminologie
 - Relevante Fakten und Annahmen
- Beispiele:
 - Datenschutz und Regulierungen
 - Einheitensystem
 - Ort von Rechenzentren

Volere: Functional Requirements

- Was ist der Sinn des Systems?
 - Rahmen der Arbeit
 - Datenmodell und Data-dictionary
 - Rahmen des Produkts
 - Funktionelle Anforderungen und Anforderungen an Daten

Volere: Non-functional Requirements

- Was sind (selbstverständliche) Erwartungen an das System?
 - Look and feel
 - Usability and humanity
 - Performance
 - Wartbarkeit- und Support
 - Sicherheit
 - Kulturell und politisch
 - Gesetzliche

Volere: Project Issues

- Sonstige Eigenschaften:
 - Offene Probleme
 - Off-the-Shelf Lösungen
 - Neue Probleme
 - Aufgaben
 - Migration auf neues Produkt
 - Risiken
 - Kosten
 - Nutzerdokumentation und –training
 - Ideen für Lösungen

Volere: Snow Card

Req-ID:	Eindeutige ID	Kategorie	Use cases/events, die diese Anforderung benötigen
Req-Type:		Events/UCs:	
Description:	Informelle Beschreibung		
Rationale:	Begründung, warum diese Anforderung wichtig ist		
Originator:	Stakeholder, der die Anforderung stellt		
Fit Criterion:	Wie kann man die Erfüllung der Anforderung messen/testen? Wie wichtig bzw. wie kritisch ist die Anforderung		
Customer Satisfaction:	Customer Dissatisfaction:	Priority:	
Supporting Material:	Verweise auf Dokumente, die diese Anforderung ausführlich beschreiben		
Conflicts:	In Konflikt/Konkurrenz stehende Anforderungen		
History:	Wann erstellt, welche Änderungen, letzte Bearbeiter,...		

Aufgabe: Bewertungskriterien

- Welche Kriterien sollten gute Anforderungen erfüllen?
- Sind die gefundenen Anforderungen gute Anforderungen?

Bewertungskriterien

- Korrekt
- Eindeutig
- Vollständig
- Konsistent
- Gewichtet nach Wichtigkeit und Stabilität
- Überprüfbar
- Modifizierbar
- Nachvollziehbar
- Quelle: IEEE Std 830-1998

In der Praxis

- Anforderungsdefinition bestehen gewöhnlich aus *natürlicher Sprache* mit zusätzlichen *Diagrammen und Tabellen* (z.B. UML)
- 3 Arten von Probleme:
 - Lack of clarity: Schwierig *präzise und gleichzeitig leicht-verständliche* Dokumente zu schreiben
 - Requirements confusion: *Funktionale und nicht-funktionale* Anforderungen sind oft *vermischt*
 - Requirements amalgamation: *Mehrere unterschiedliche* Anforderungen werden *zusammen ausgedrückt*.

4. Anforderungsrevision

- Erneute Prüfung und ggf. Anpassung der Anforderungen

Validität	Does the system provide the functions <i>which best support</i> the customer's needs?
Konsistenz	Are there any <i>requirements conflicts</i> ?
Vollständigkeit	Are <i>all functions</i> required by the customer included?
Realisierbarkeit	Can the requirements be implemented given <i>available budget and technology</i> ?

Checkliste kann helfen

http://wwwis.win.tue.nl/2M390/rev_req.html

- Does the (software) product have a *succinct name*, and a *clearly described purpose*?
- Are the *characteristics of users* and of *typical usage* mentioned?
(No user categories missing.)
- Are all *external interfaces* of the software explicitly mentioned?
(No interfaces missing.)
- Does each specific requirement have a *unique identifier* ?
- Is each requirement *atomic* and *simply formulated* ?
(Typically a single sentence. Composite requirements must be split.)
- Are requirements organized into *coherent groups* ?
(If necessary, hierarchical; not more than about ten per group.)
- Is each requirement *prioritized* ?
(Is the meaning of the priority levels clear?)
- Are all *unstable requirements* marked as such?
(TBC='To Be Confirmed', TBD='To Be Defined')



Zusammenfassung

- Notwendigkeit von Requirements Engineering verstehen
- Typische Probleme bei der Anforderungsanalyse kennen
- Vorgehen für systematisches Finden von Anforderungen verstehen
- Anforderungen beschreiben können

Was Sie mitgenommen haben sollten

- Warum brauchen wir Requirements Engineering?
- [Beschreibung von Anwendung X]
 - Nennen Sie X Stakeholder. Erklären Sie Ihre Auswahl.
 - Beschreiben Sie X Szenarien. Erklären Sie Ihre Auswahl.
 - Nennen und beschreiben Sie X funktionale und X nicht-funktionale Eigenschaften. Erklären Sie Ihre Entscheidung.
 - Sind Ihre Anforderungen gute Anforderungen? Warum?
- Würden Sie den stakeholderbasierten oder szenariobasierten Ansatz zum systematischen Finden von Requirements empfehlen?

Literatur

- Pohl. Requirements Engineering: Grundlagen, Prinzipien, Techniken. 2008
- Sommerville. Software Engineering. Kapitel 6-10.

Softwaretechnik

Responsibility-Driven Design



**SOFTWARE
SYSTEME**

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



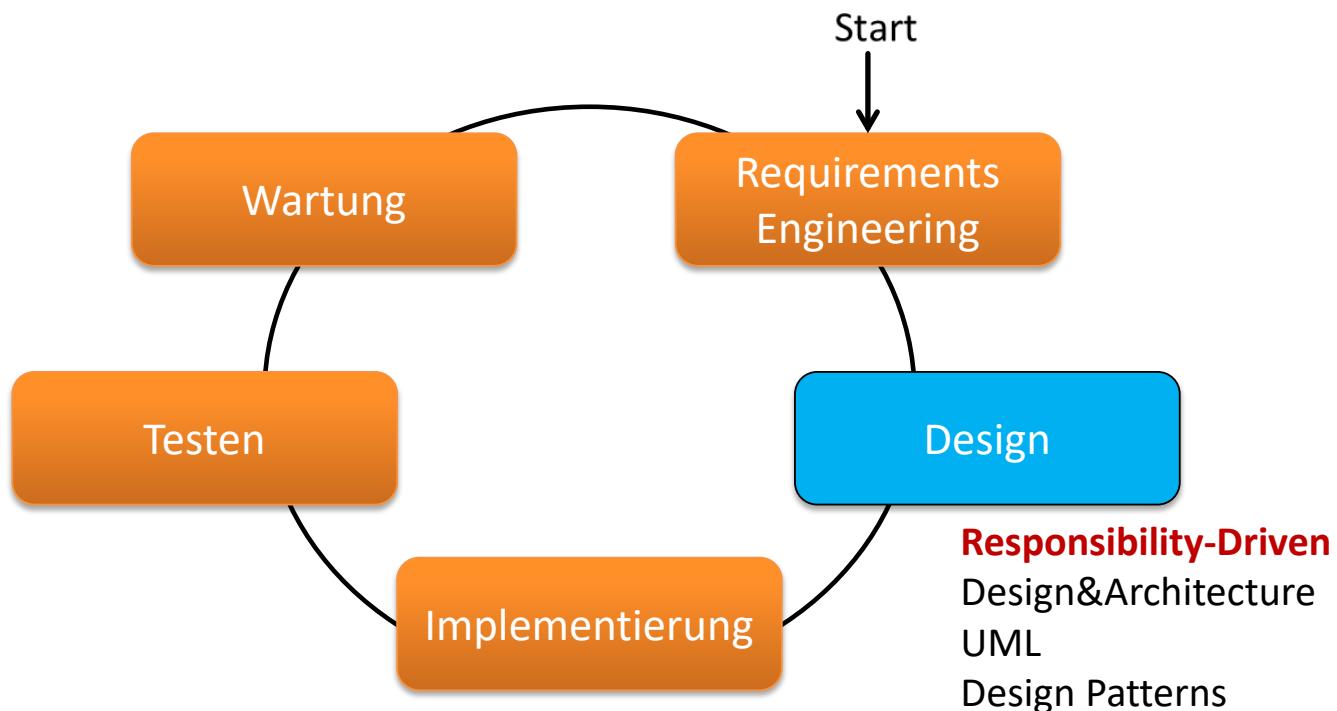
UNIVERSITÄT
LEIPZIG

Wiederholung: Entwicklungsprozesse

- Herangehensweisen und Prozesse für die Entwicklung von Softwaresystemen
- **Sequentiell:** Stricke Abfolge der Phasen im Lebenszyklus mit Dokumentationen am Ende einer Phase (gut für Projekte mit hohen Sicherheitsanforderungen, klaren und nicht änderbaren Anforderungen)
- **Iterative:** Kurze Entwicklungszyklen, zur Realisierung bestimmter Features in ein auslieferbares Produkt (gut für sich ändernde und unklare Anforderungen, schnelles Feedback, kleinere, agile Teams)
- **Scrum:** Managementframework, welches in festen Sprints Ergebnisse erzielt (gut für neue Produktentwicklung, passt zu agiler Entwicklung; kombinierbar mit Praktiken des XP)
- **Lean:** ähnlich zu Scrum, aber flexibler und darauf ausgelegt, die Arbeit an angefangenen Teilen zu minimieren (gut für Systeme in der Wartung; schnelle Bearbeitung kleinerer Funktionseinheiten)



Einordnung



Systementwurf und initiales Design

Wie komme ich von Anforderungen / Spezifikationen / User Stories zu einem (objekt-orientierten) Entwurf eines Systems?

Story-driven Design

Entwurfsmethodik mit nicht-IT'lern

Model-getriebenes Design

Eingebettete Systeme

Responsibility-driven Design

Client-Server Sichtweise im Entwurf / OOP

Data-driven Design

Daten-lastige Systeme

Objektorientierte Analyse und Design

Anwendungen implementiert mit OOP

Feature-orientierte Analyse und Design

Softwarereproduktlinien, konfigurierbare Softwaresysteme

Domain-driven Design

Verteilte, multi-purpose Systeme,
Standard für Entwurf von Microservice-Architekturen

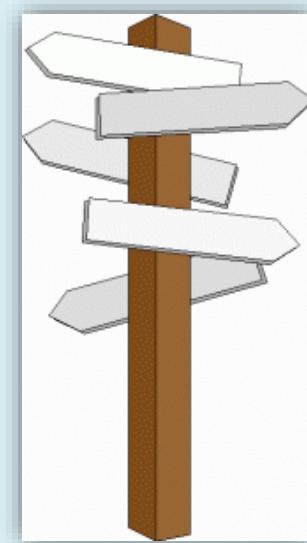


Iteration in Objekt-Orientierten Designs

- Ergebnis des Design-Prozesses ist *kein finales Produkt*:
 - Design-Entscheidungen werden evtl. *überdacht*, selbst nach deren Implementierung
 - Design ist nicht linear, sondern *iterativ*
- Der Design-Prozess ist *nicht algorithmisch*:
 - Eine Design-Methode bietet daher nur *Richtlinien* und keine festen Regeln
 - “a good *sense of style* often helps produce clean, elegant designs — designs that make a lot of sense from the engineering standpoint”

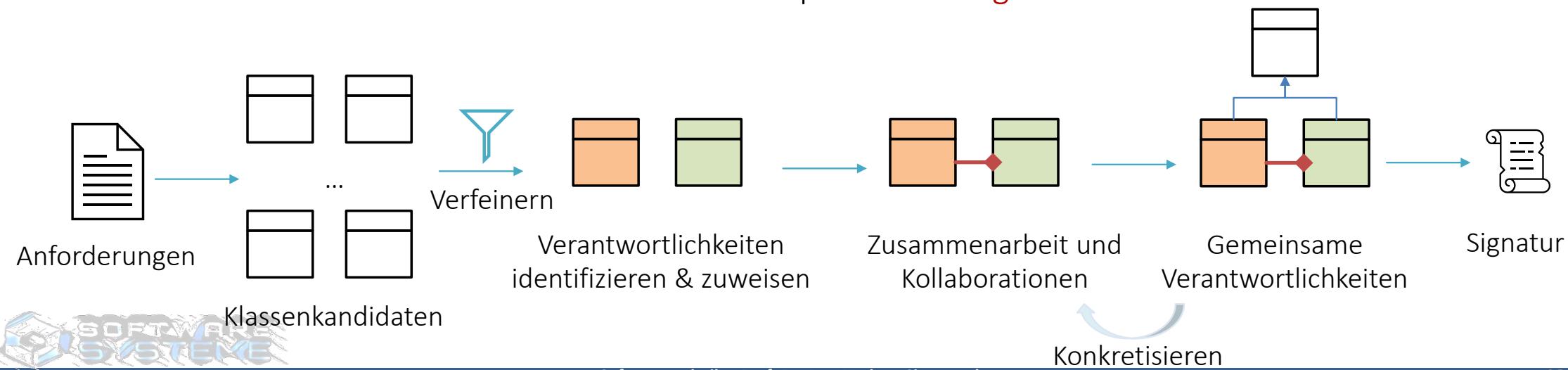
Responsibility-Driven Design ist eine (Analyse- und) Design-Technik, die gut in Kombination mit verschiedenen Methoden und Notationen arbeitet.

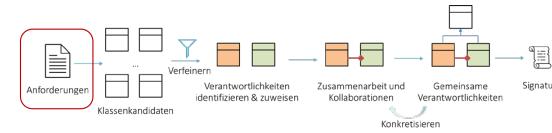
Wie komme ich zu Klassen?



Responsibility-Driven Design: Ablauf

1. Finde die *Klassen* in deinem System (von Kandidaten zu konkreten Klassen)
2. Bestimme die *Verantwortlichkeiten* jeder Klasse
3. Bestimme wie Objekte *zusammenarbeiten*, um ihre Verantwortlichkeiten zu erfüllen
4. Bestimme *gemeinsame* Verantwortlichkeiten zur Bildung von Klassenhierarchien
5. Konkretisiere *Kollaborationen* zwischen Objekte
6. Wandle Klassenverantwortlichkeiten in voll spezifizierte *Signaturen* um



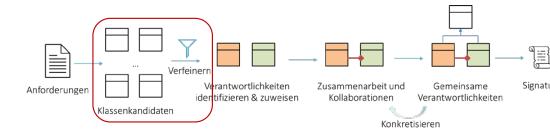


Finden der Klassen

Start mit der Requirements-Spezifikation:

Was sind die Ziele, der erwartete Input und Output des Systems, welches entworfen wird?

1. Suche nach *Nomen Phrasen*:
 - Sepriere in offensichtliche Klassen, Kandidaten für Klassen und keine Klassen



Finden der Klassen...

2. Verfeinere die Liste von *Kandidaten*.

Mögliche Hinweise:

- Modelliere *physikalische Objekte* — z.B. Festplatte, Drucker
- Modelliere *konzeptuelle Entitäten* — z.B. Windows, Dateien
- Wähle *ein Wort für ein Konzept* — Was bedeutet das Konzept innerhalb des Systems?
- Vorsicht bei *Adjektiven* — Ist es wirkliche eine eigenständige Klasse?
- Modelliere *Kategorien von Klassen* — Verschiebe noch Modellierung von Vererbungen
- Modelliere *Interfaces* zum System — z.B., Nutzerinterface, Programminterface
- Modelliere Attribut*werte*, nicht Attribute — z.B., Punkt vs. Center

Beispiel: Drawing Editor Requirements Specification

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button.

The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

Drawing Editor: Nomen Phrasen

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

...

Drawing Editor: Nomen Phrasen

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

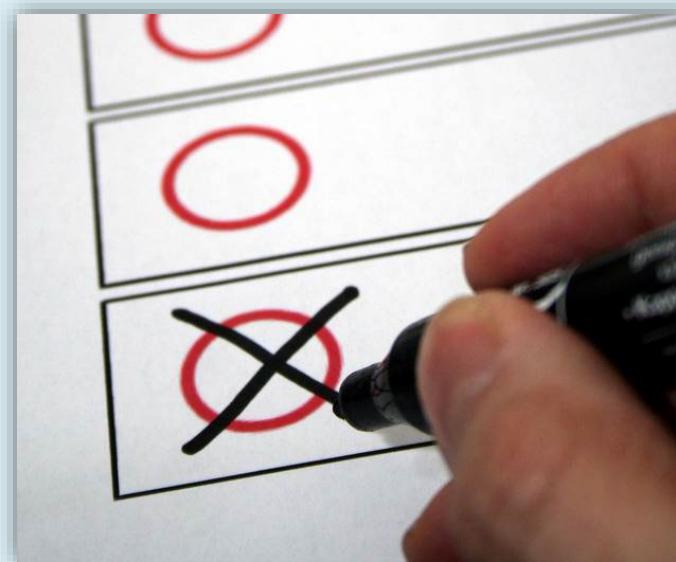
The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

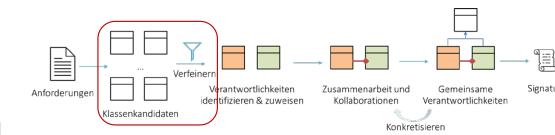
The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

Wie wähle ich Klassen aus?

- Physikalische Objekte
- Konzeptuelle Entitäten
- Ein Wort für ein Konzept
- Vorsicht bei Adjektiven
- Kategorien von Klassen
- Interface zum System
- Modelliere Attributwerte

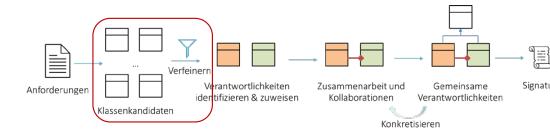




Prinzipien der Klassenauswahl

Modelliere *physikalische Objekte*:

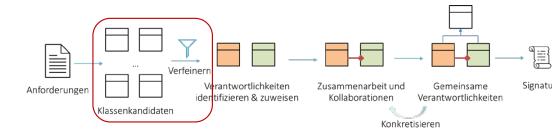
- ~~mouse button~~ [event or attribute]
- Modelliere *konzeptuelle Entitäten*:
 - ellipse, line, rectangle
 - Drawing, Drawing Element
 - Tool, Creation Tool, Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
 - text, Character
 - Current Selection



Prinzipien der Klassenauswahl ...

Wähle *ein Wort für ein Konzept*:

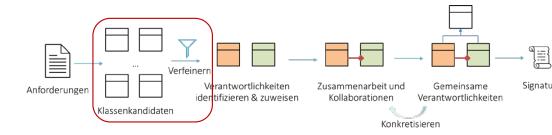
- Drawing Editor
⇒ ~~editor, interactive graphics editor~~
- Drawing Element
⇒ ~~element~~
- Text Element
⇒ ~~text~~
- Ellipse Element, Line Element, Rectangle Element
⇒ ~~ellipse, line, rectangle~~



Prinzipien der Klassenauswahl ...

Vorsicht bei *Adjektiven (in dt. auch zusammengesetzte Nomen)*:

- Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
 - Alle haben unterschiedliche Anforderungen
- Rectangle \Rightarrow ~~bounding rectangle, rectangle, region~~
 - Gleiche Bedeutung, aber unterscheiden sich von Rectangle Element
- Point \Rightarrow ~~end point, start point, stop point~~
- Control Point
 - Mehr als nur eine einfache Koordinate
- Corner \Rightarrow ~~associated corner, diagonally opposite corner~~
 - Kein neues Verhalten



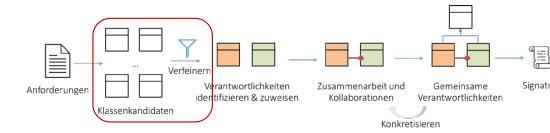
Prinzipen der Klassenauswahl ...

Modelliere *Kategorien von Klassen*:

- Tool, Creation Tool

Modelliere *Interfaces* zum System: *keine guten Kandidaten hier...*

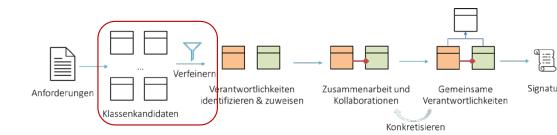
- ~~user~~ – *don't need to model user explicitly*
- ~~cursor~~ – *cursor motion handled by operating system*



Prinzipien der Klassenauswahl ...

Modelliere Attribut*werte*, nicht Attribute:

- ~~height of the rectangle, width of the rectangle~~
- ~~major radius, minor radius~~
- ~~position~~ – *des ersten Textzeichens; vermutlich Point-Attribut*
- ~~mode of operation~~ – *Attribut von Drawing Editor*
- ~~shape of the cursor, I-beam, crosshair~~ – *Attribute von Cursor*
- ~~corner~~ – *Attribut von Rectangle*
- ~~time~~ – *Ein implizites Attribut des Systems*



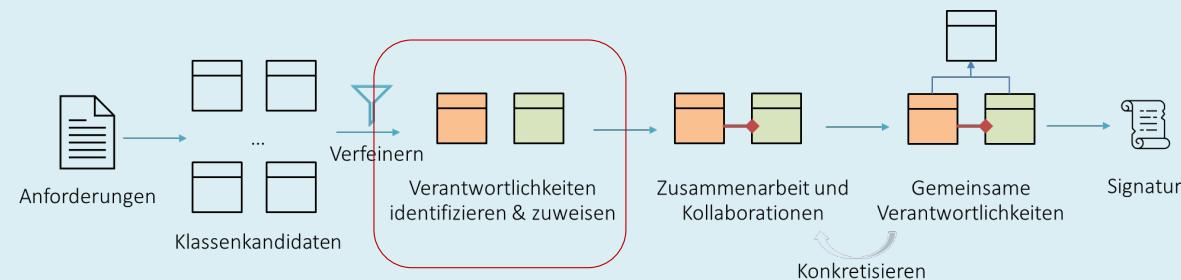
Kandidaten für Klassen

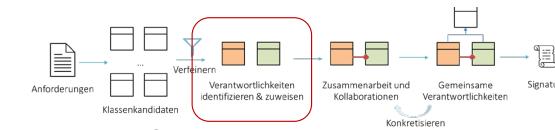
Initiale Analyse ergibt die folgenden Kandidaten:

Character	Line Element
Control Point	Point
Creation Tool	Rectangle
Current Selection	Rectangle Creation Tool
Drawing	Rectangle Element
Drawing Editor	Selection Tool
Drawing Element	Text Creation Tool
Ellipse Creation Tool	Text Element
Ellipse Element	Tool
Line Creation Tool	

Erwartet, dass die Liste sich weiterentwickelt, während Ihr beim Design voranschreitet

Wie identifizierte ich die Verantwortlichkeiten?





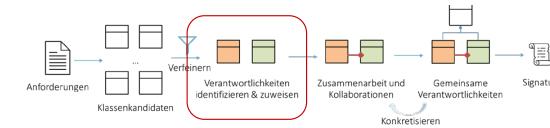
Verantwortlichkeiten (Responsibilities)

Was sind Verantwortlichkeiten?

- Das **Wissen**, welches ein Objekt verwaltet und anbietet
- Die **Aktionen**, die es ausführen kann

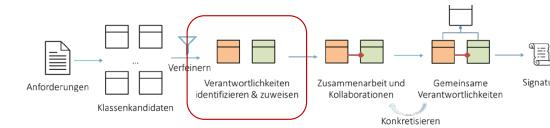
Verantwortlichkeiten repräsentieren die *öffentlichen Leistungen*, die ein Objekt seinen Klienten anbietet (aber nicht die Art, wie diese Leistungen realisiert werden können)

- Spezifizierte *was* ein Objekt tut, nicht *wie* es dies tut
- Beschreibe noch nicht das Interface, sondern nur die *konzeptuellen Verantwortlichkeiten*



Identifizieren der Verantwortlichkeiten

- Studiere die Requirements-Spezifikationen:
 - Hebe **Verben** hervor und bestimme, welche Verantwortlichkeiten diese repräsentieren
 - Mache einen **walk-through** vom System
 - Exploriere so viele Szenarien wie möglich
 - Identifizierte Aktionen, welche durch Eingaben an das System resultieren
- Studiere die Kandidatenklassen:
 - Klassennamen \Rightarrow Rollen \Rightarrow Verantwortlichkeiten
 - Aufgenommene „Sinnhaftigkeiten“ von Klassen \Rightarrow Verantwortlichkeiten

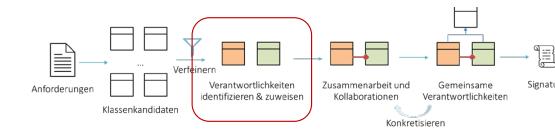


Wie weise ich Verantwortlichkeiten zu?

Pelrine's Laws:

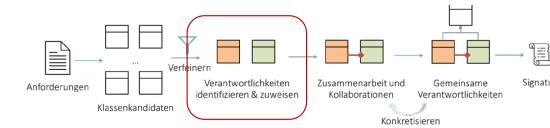
1. “Don't do anything you can push off to someone else.”
2. “Don't let anyone else play with you.”

- Beispiel: Klasse Buch und Klasse Bibliothek
 - Szenario: Such nach Text in einem Buch
 - Law 1: Bibliothek sollte nicht nach Text in einem bestimmten Buch suchen, obwohl die Klasse eine Liste von Büchern hat.
 - Law 2: Der Text eines Buches gehört zum Buch, also lass niemanden mit deinem Text arbeiten, sondern nur die Klasse Buch bestimmt alle erlaubten Aktionen dafür



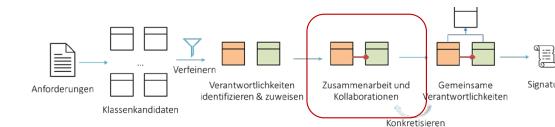
Zuweisen von Verantwortlichkeiten

- *Verteile gleichmäßig* die System-Intelligenz
 - Verhindere prozedural zentralisierte Verantwortlichkeiten
 - Halte Verantwortlichkeiten nah an den Objekten und nicht an deren Nutzern
- Definiere Verantwortlichkeiten so *generell* wie möglich
 - “draw yourself” vs. “draw a line/rectangle etc.”
 - Führt zum Teilen
- Halte *Verhalten* zusammen mit jedweder *relevanten Information*
 - Prinzip der Kapselung



Zuweisen von Verantwortlichkeiten...

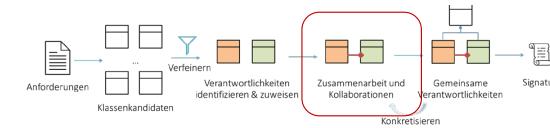
- Behalte Informationen über eine Sache an *einem Ort*
 - Falls mehrere Objekte Zugriffe auf die gleiche Information benötigen:
 1. Ein neues Objekte kann eingeführt werden, welches die Information verwaltet, oder
 2. Eines der vorhandenen Objekte kann die Information verwalten, oder
 3. die mehrfachen Objekte können in ein einzelnes Objekt überführt werden
- *Teile* Verantwortlichkeiten zwischen ähnlichen Objekten
 - Breche komplexe Verantwortlichkeiten auf



Beziehungen zwischen Klassen

Zusätzliche Verantwortlichkeiten können entdeckt werden, indem wir die Beziehungen zwischen Klassen untersuchen:

- Die “Is-Kind-Of” Beziehung:
 - Klassen, die *gemeinsame Attribute* teilen, teilen oft eine *gemeinsame Oberklasse*
 - Gemeinsame Oberklassen weisen auf *gemeinsame Verantwortlichkeiten* hin
 - z.B., um ein neues Drawing Element zu kreieren, muss das Creation Tool folgendes tun:
 1. accept user input
 2. determine location to place it
 3. instantiate the element*implemented in subclass*
generic
implemented in subclass

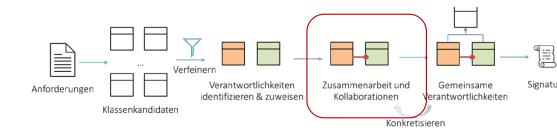


Beziehungen zwischen Klassen ...

- Die “Is-Analogous-To” Beziehung:
 - *Ähnlichkeiten* zwischen Klassen weisen auf eine zur Zeit noch unentdeckte Oberklasse hin
- Die “Is-Part-Of” Beziehung :
 - *Unterscheide* Verantwortlichkeiten zwischen eines *Teils* und des *Ganzen*

Schwierigkeiten bei der Zuweisung von Verantwortlichkeiten weisen auf:

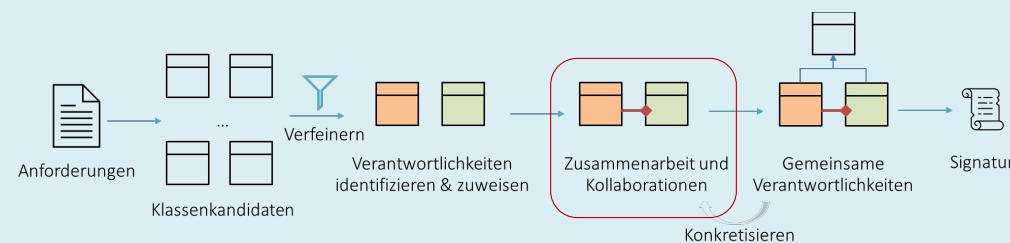
- *Fehlende Klassen* im Design (z.B., Group Element), oder
- *Freie Auswahl* zwischen mehreren Klassen hin

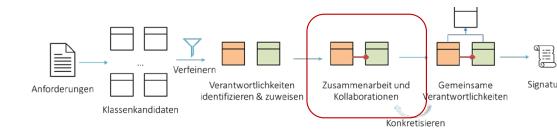


Beispiel Beziehungen

- Drawing element *is-part-of* Drawing
- Drawing Element *has-knowledge-of* Control Points
- Rectangle Tool *is-kind-of* Creation Tool

Wie finde ich Kollaborationen?

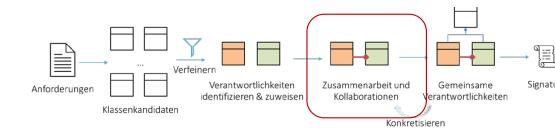




Kollaborationen

Was sind Kollaborationen?

- Kollaborationen sind *Benutzerinanfragen (client requests)* an Dienste, die benötigt werden, um Verantwortlichkeiten zu erfüllen
- Kollaborationen enthüllen *Kontroll- und Informationsflüsse* und, ultimativ, Subsysteme
- Kollaborationen können *fehlenden Verantwortlichkeiten* offenbaren
- Analysen von Kommunikationsmustern können *fehlerhaft zugewiesene* Verantwortlichkeiten offenbaren



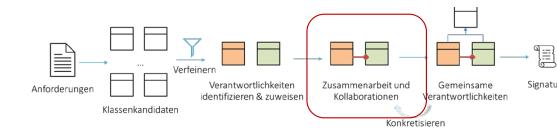
Finden von Kollaborationen

Für jede Verantwortlichkeit:

1. Kann die Klasse die Verantwortlichkeit *selbstständig erfüllen*?
2. Falls nicht, *was benötigt es*, und von welcher anderen Klasse kann es dies erhalten?

Für jede Klasse:

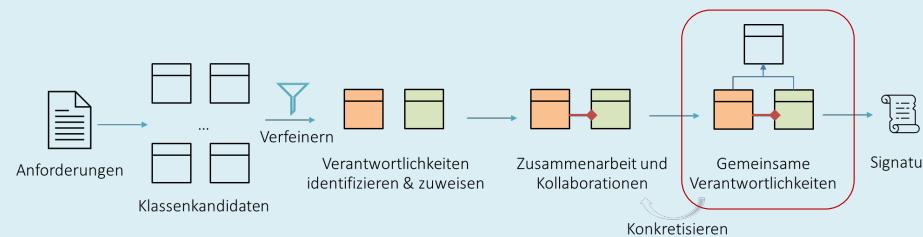
1. Was *weiß* diese Klasse?
2. Welche *anderen Klassen* benötigen ihre Informationen oder Ergebnisse? Prüfe auf Kollaborationen.
3. Klassen, die *nicht* mit anderen *interagieren*, sollten *aussortiert* werden. (sorgfältig prüfen!)

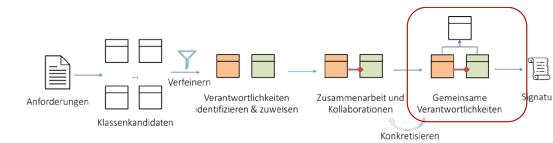


Auflistung der Kollaborationen

Drawing	
Weiß, welche Elemente es verwaltet	
Verwaltet Reihenfolge der Elemente	Drawing Element

Wie finde ich Vererbungshierarchien?

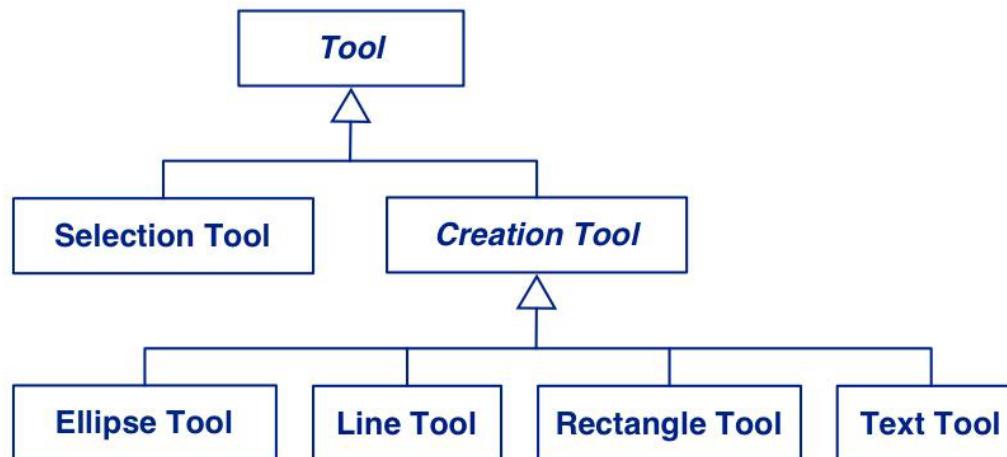


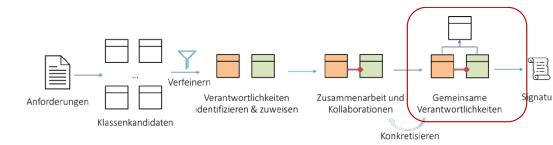


Finden von Abstrakten Klassen

Abstrakte Klassen fassen gemeinsame Verantwortlichkeiten aus anderen Klassen zusammen

- Gruppiere verwandte Klassen mit gemeinsamen Attributen
- Führe abstrakte Oberklassen ein, die diese Gruppe repräsentieren
- “Kategorien” sind gute Kandidaten für abstrakte Klassen





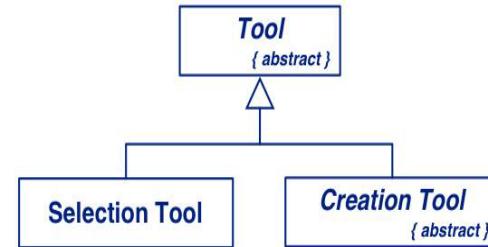
Teilen von Verantwortlichkeiten

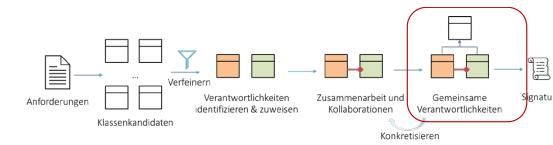
Konkrete Klassen kann man instanziieren und von ihnen erben.

Von abstrakten Klassen kann man nur erben.

Notiere Abstraktheit in Klassendiagrammen.

Venn Diagramme können für die Visualisierung von geteilten Verantwortlichkeiten verwendet werden.



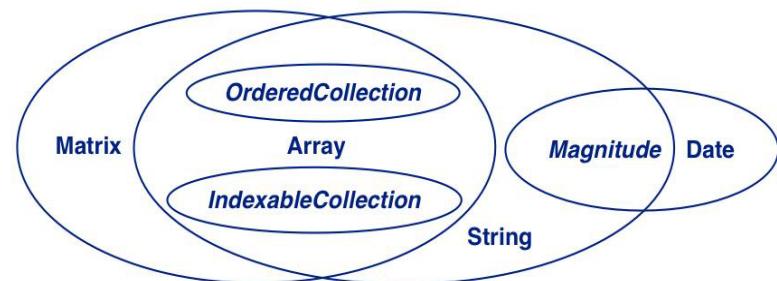
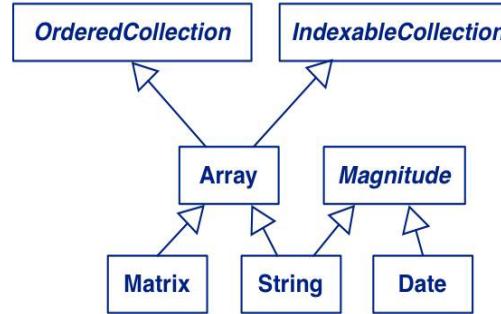


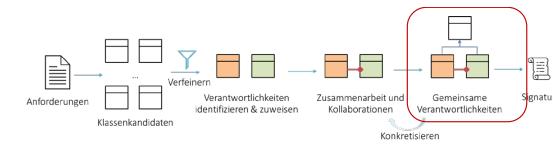
Mehrfachvererbung

Bestimme, ob eine Klasse *instanziert* wird, um zu entscheiden, ob sie *abstrakt* oder *konkret* ist.

Verantwortlichkeiten von Subklassen sind *größer* als diese von *Oberklassen*.

Überschneidungen repräsentieren *gemeinsame Oberklassen*.





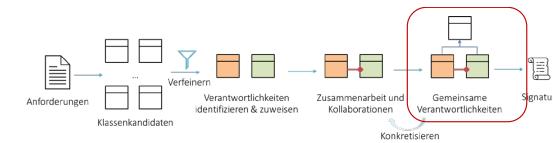
Entwerfen von Guten Hierarchien

Modelliere eine “kind-of” Hierarchie:

- Unterklassen sollten *alle geerbten Verantwortlichkeiten unterstützen*, und eher noch mehr

Schiebe gemeinsame Verantwortlichkeiten so hoch wie möglich:

- Klassen, die *gemeinsame Verantwortlichkeiten teilen* sollten *von einer gemeinsamen abstrakten Superklasse erben*; führe fehlende Superklassen ein



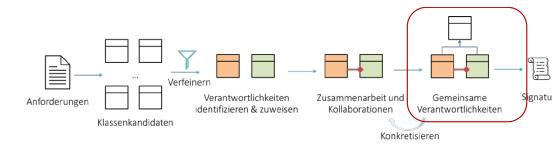
Entwerfen von Guten Hierarchien ...

Stelle sicher, dass abstrakte Klassen nicht von konkreten Klassen erben:

- Eliminiere dies durch die Einführung weiterer *gemeinsamer abstrakter Superklassen*: abstrakte Klassen sollten Verantwortlichkeiten in einem implementierungsunabhängigen Weg unterstützen

Eliminiere Klassen, die keine neue Funktionalität hinzufügen:

- Klassen sollten entweder neue Verantwortlichkeiten oder eine bestimmte Implementierung von vererbten Verantwortlichkeiten hinzufügen

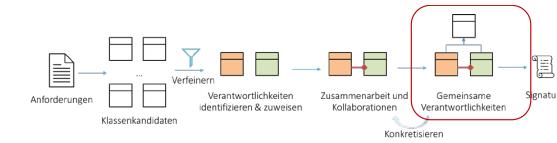


Entwerfen von Kind-Of Hierarchien

Korrekt gebildete Verantwortlichkeiten von Unterklassen:



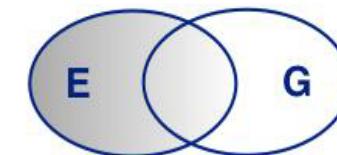
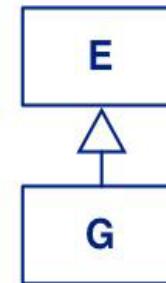
C übernimmt *alle*
Verantwortlichkeiten
von A und B



Entwerfen von Kind-Of Hierarchien ...

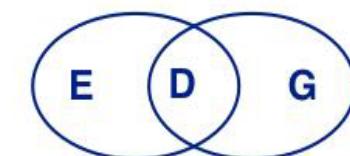
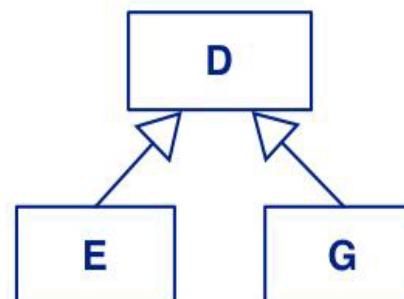
Falsche Unter-Oberklassen-Beziehung

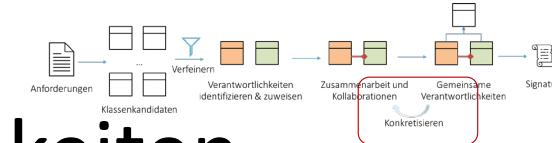
- G übernimmt nur *einige* der Verantwortlichkeiten, welche von E geerbt wurden



Verfeinerte Vererbungsbeziehung

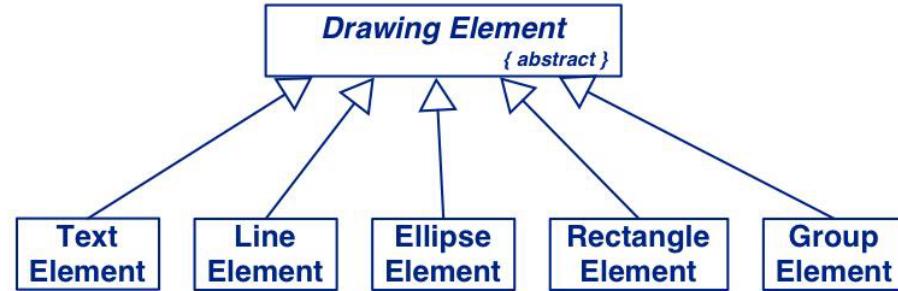
- Führe eine *abstrakte Oberklasse* ein, welche die gemeinsamen Verantwortlichkeiten kapselt



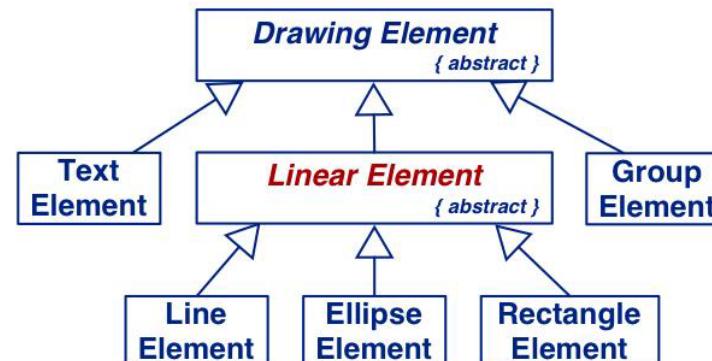


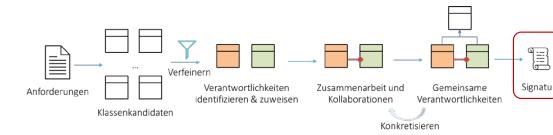
Refaktorisierung von Verantwortlichkeiten

Lines, Ellipses und Rectangles
sind verantwortlich, um die
Breite und Farbe ihrer Linien zu
speichern.



Dies weist auf eine
gemeinsame Superklasse hin.





Protokolle (Interfaces)

Ein Protokoll ist eine *Menge von Signaturen* (d.h., ein *Interface*), die zu einer Klasse gehören.

- Protokolle sind für *öffentliche Verantwortlichkeiten* spezifiziert.
- Protokolle für *private* Verantwortlichkeiten sollten spezifiziert werden, falls sie in ihren *Sub(unter)-klassen* benutzt oder implementiert werden (protected Sichtbarkeit)

1. Entwerfe eine Protokoll für jede Klasse
2. Schreibe eine Design-Spezifikation für jede Klasse und Subsystem
3. Schreibe eine Design-Spezifikation für jeden Kontrakt

Was Ihr mitgenommen haben solltet!

- Welche Kriterien gibt es mit denen ich potentielle Klassen identifizieren kann?
- Was sind Verantwortlichkeiten von Klassen und wie kann ich sie identifizieren?
- Wie kann das Identifizieren von Verantwortlichkeiten beim Identifizieren von Klassen helfen?
- Was sind Kollaborationen und wie stehen sie in Beziehung zu Verantwortlichkeiten?
- Wie kann ich abstrakte Klassen identifizieren?
- Welche Kriterien gibt es, um gute Klassenhierarchien zu entwerfen?
- Wie kann das Refaktorisieren von Verantwortlichkeiten Hierarchien verbessern?

Literatur

- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.

Softwaretechnik

Modeling Behavior



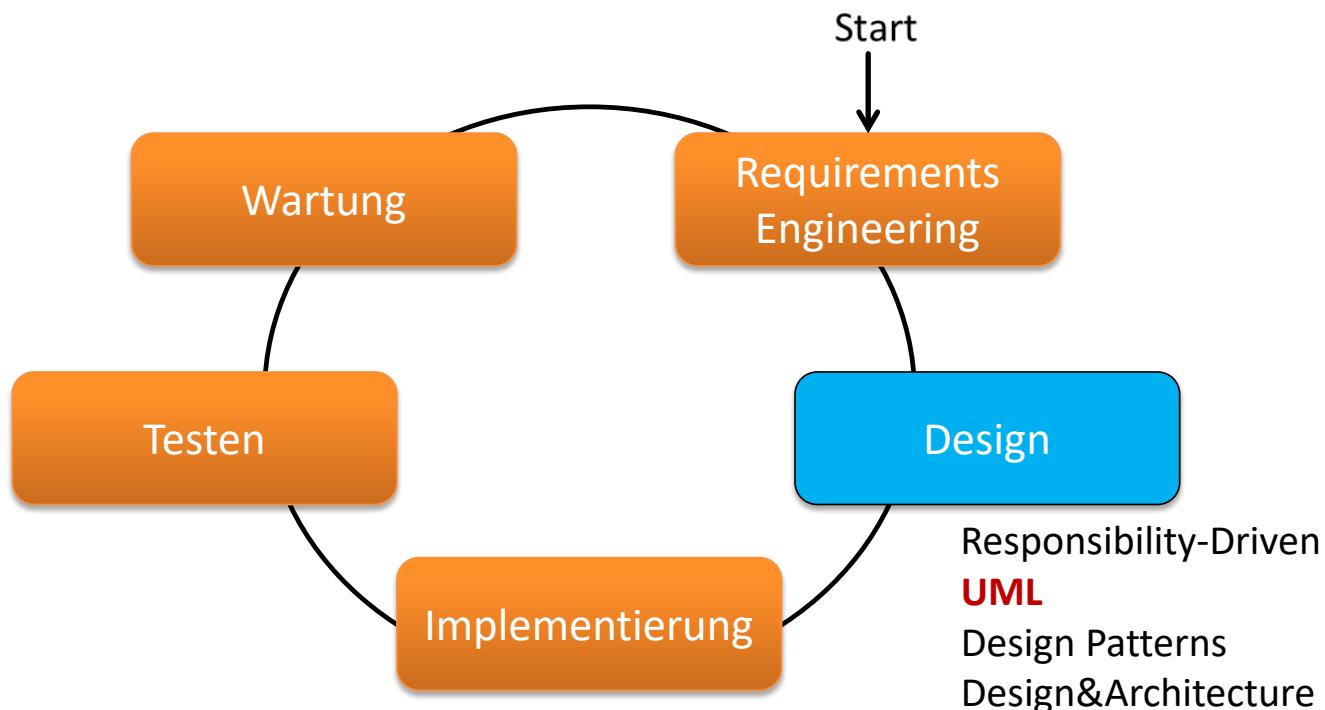
SOFTWARE
SYSTEME

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Einordnung



Use-Case Diagramme

Use-Case Diagramme

Ein use case ist eine *generische Beschreibung einer gesamten Transaktion* welche eine oder mehrere Akteure involviert.

Ein use-case Diagramm präsentiert eine *Menge von use cases* (Ellipsen) und deren externe Akteure, die mit dem System interagieren.

Abhängigkeiten und *Assoziationen* zwischen use cases können dargestellt werden.

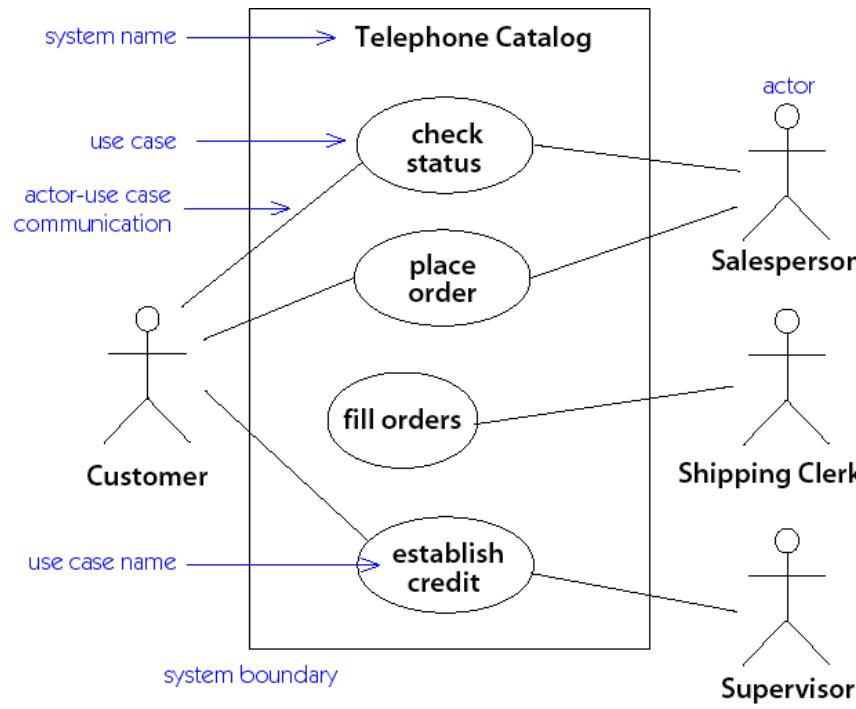


Figure 5-1. Use case diagram

Anwendung

- Aufzeigen der Ziele der User-System Interaktionen
- Definition und Organisation der funktionalen Anforderungen
- Spezifikation des Kontext des Systems
- Modellierung der grundlegenden Abläufe eines Use Cases



Identifikation von Aktoren

- Wer verwendet das System?
- Wer installiert das System?
- Wer wartet das System?
- Wer administriert das System?
- Wer beendet das System?
- Mit wem oder was kommuniziert das System?
- Wer erhält Informationen vom System?
- Welche anderen Systemen verwenden dieses System?

Was kann ein Aktor sein?

- Person
- Organisation
- Anderes System
- Externes Gerät

Identifikation von Use Cases

- Falls das System Informationen speichert: Welche Aktoren werden diese Informationen kreieren, updaten, verwalten, löschen, lesen?
- Welche Funktionen möchten die einzelnen Aktoren vom System verwenden?
- Gibt es externe Events, die das System betreffen und wie wird das System darüber informiert?
- Informiert das System Aktoren über einen geänderten Zustand innerhalb des Systems?

Was ist ein Use Case?
• Aktion, die eine bestimmte Aufgabe im System erfüllt



Beziehungen (Relationships)

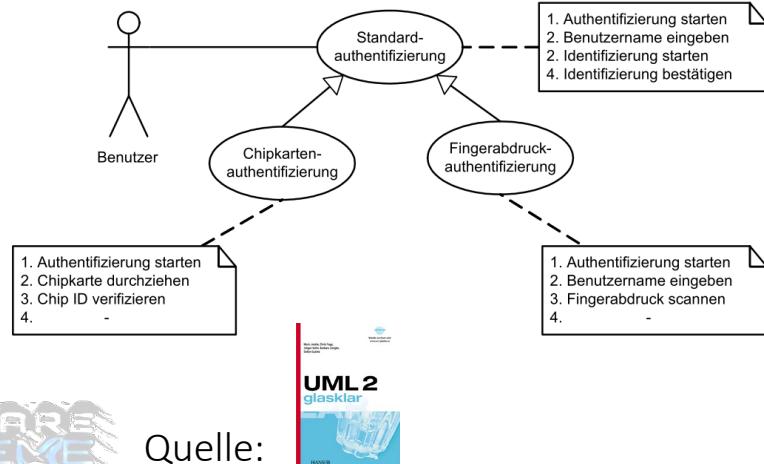
- Association: Kommunikation und Interaktion zw. Aktor und Use Case
- <<Include>>: Repräsentiert eine Abhängigkeit von einer Use Case zu einem „included“ Use Case; keine Akteure initiieren diese Interaktion, sondern sie wird von einem anderen Use Case gestartet;
Beispiel: (Log In) ---<<include>>---> (Verify Password)
- <<Extend>>: Wenn der Basis Use Case ausgeführt wird, wird *unter Umständen* (Kriterium erforderlich) der Extended Use Case ebenfalls ausgeführt;
Beispiel: (Log In) <---<<extend>>--- (Display failed log in)
- Generalization: Ähnlich zu Vererbung: Erweiterung der Basisfunktionalität;
Beispiel: (Make Payment) ← (Pay with Credit Card)

Verwendung: Use-Case Diagramm

“A use case is a *snapshot of one aspect* of your system. The sum of all use cases is *the external picture* of your system ...”

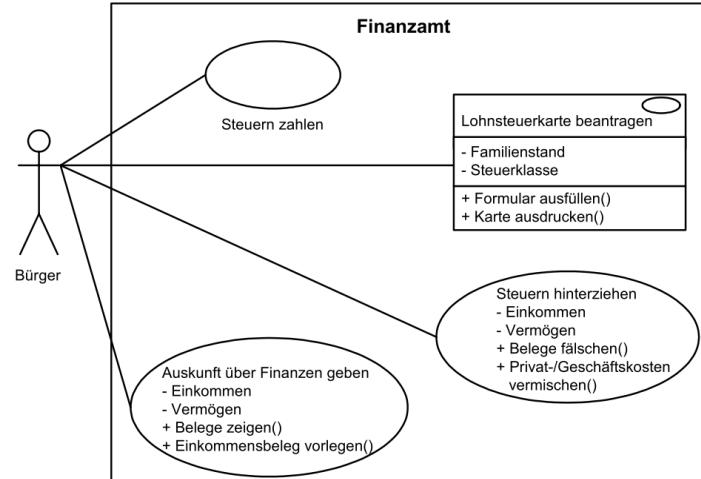
—UML Distilled

Generalisierung und Kommentare



Quelle:

Auch Attribute und Operationen möglich



Sequenz Diagramme

Szenarien

Ein Szenario ist eine *Instanz* von einem use case, dass ein *typisches Beispiel* einer Ausführung zeigt.

Szenarien können durch UML repräsentiert werden, entweder durch *sequence diagrams* oder *collaboration diagrams*.

Beachtet: Ein Szenario beschreibt nur ein Beispiel eines use cases, so dass Besonderheiten oder Bedingungen nicht ausgedrückt werden können!

Sequenzdiagramme

Ein sequence diagram beschreibt ein Szenario durch das Zeigen von Interaktionen zwischen einer Menge von Objekten in einer *zeitlichen Abfolge*.

Objekte (keine Klassen!) werden als *vertikale Balken* gezeichnet. *Events* oder Nachrichtensendungen werden als horizontale (oder schräge) *Pfeile* vom Sender zum Empfänger gezeichnet.

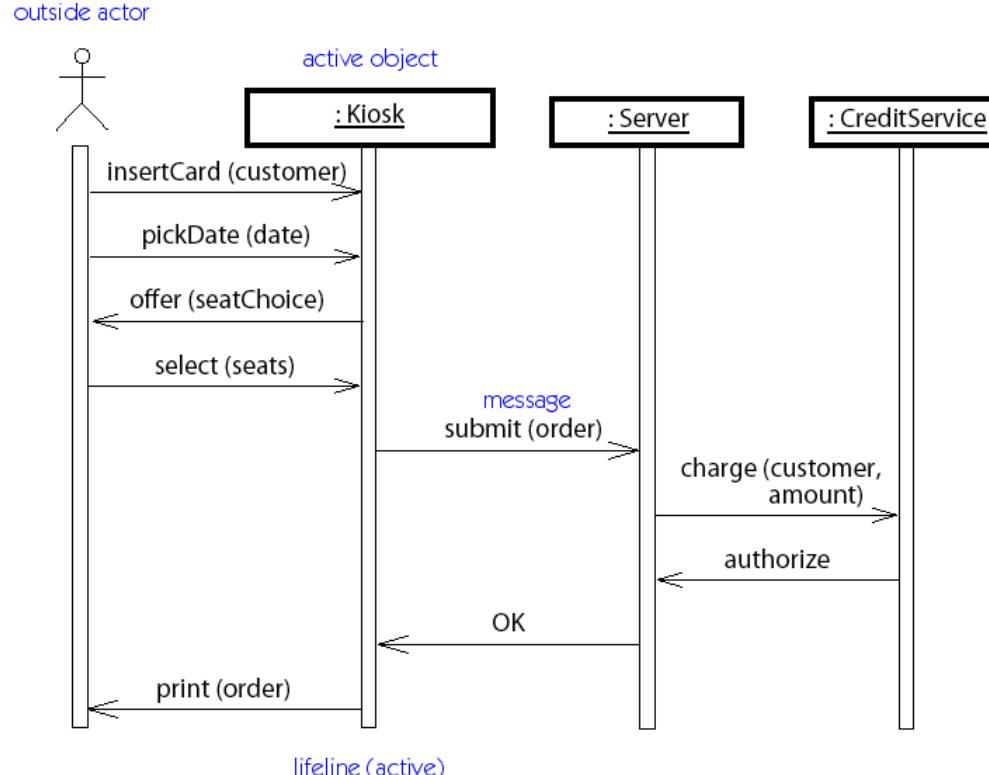


Figure 8-1. Sequence diagram Szenario: Sitzplatz im Kino reservieren

Aktivierungen

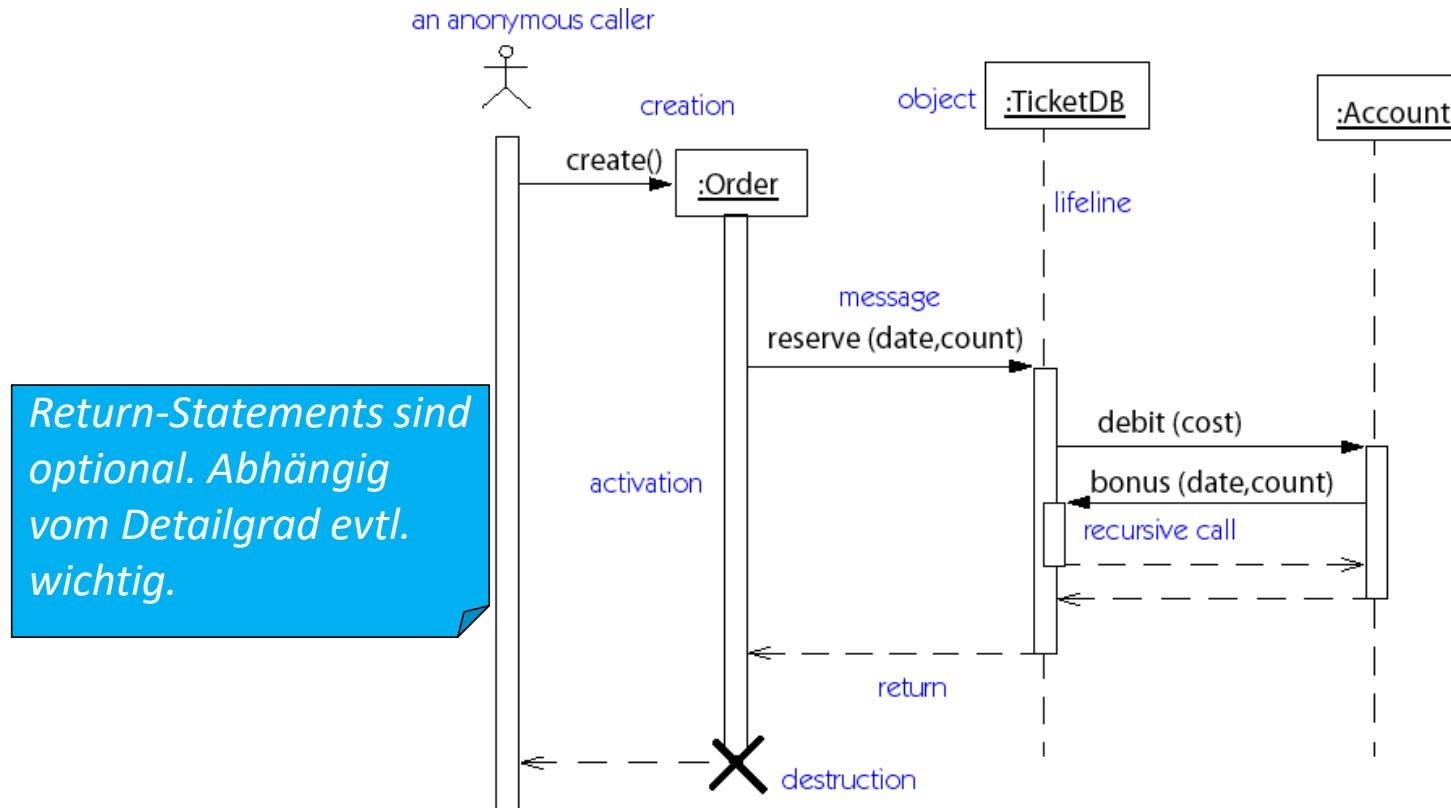


Figure 8-2. Sequence diagram with activations

Asynchronität und Bedingungen

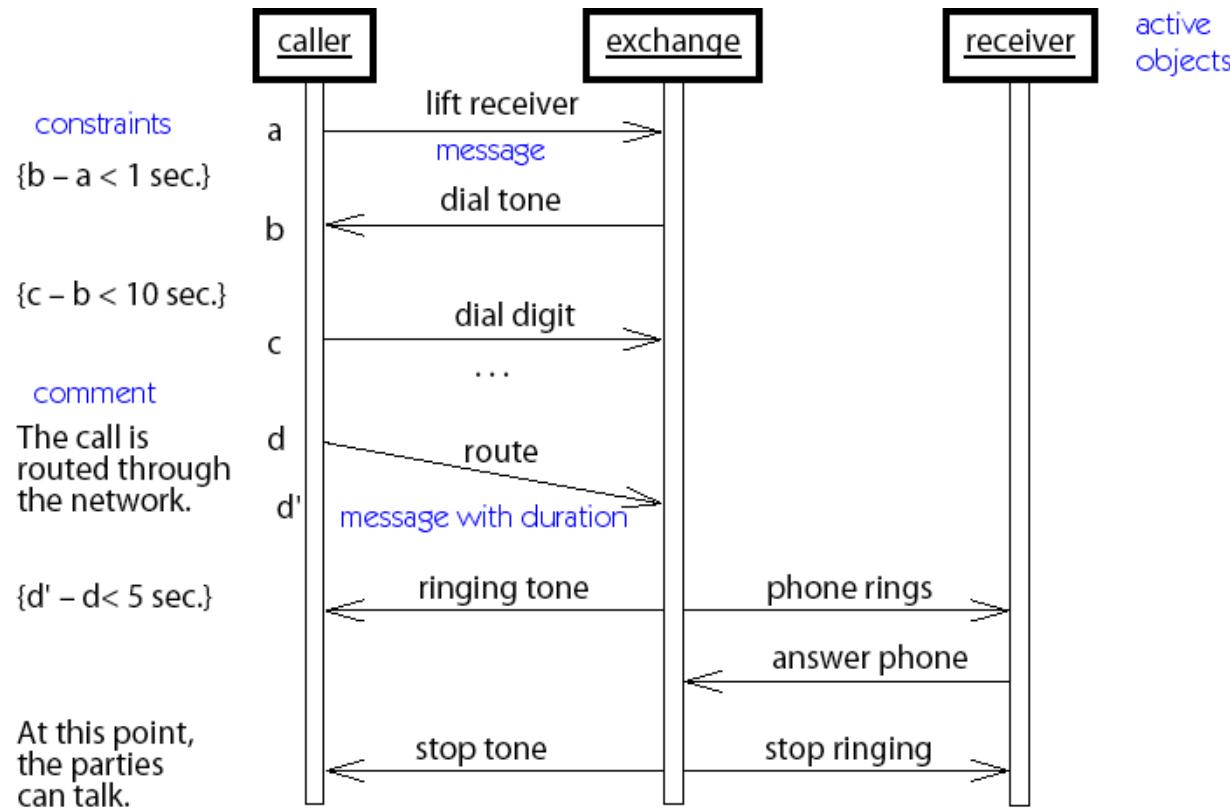
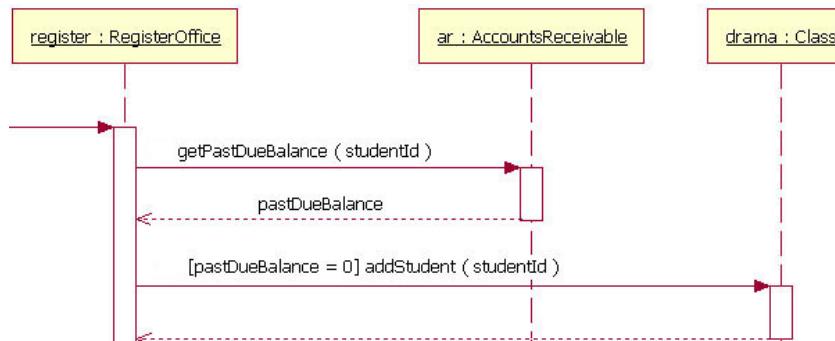


Figure 13-161. Sequence diagram with asynchronous control

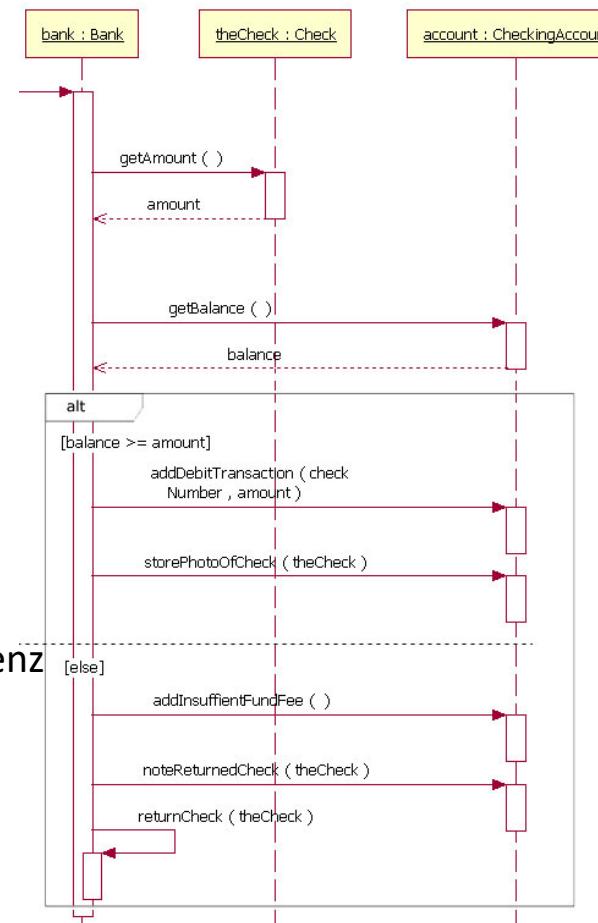
Alternativen und Guards

Guard: Bedingung muss erfüllt sein, bevor eine Nachricht verschickt wird.



Syntax: [Boolean Test]

Alternative Sequenz



Statechart (Zustands-)Diagramme

Definition

Ein Statechart Diagram beschreibt die *zeitliche Evolution* eines Objektes von einer gegebenen Klasse in Abhängigkeit von *Interaktionen* mit anderen Objekten innerhalb und außerhalb des Systems.

Ein Event ist eine one-way (asynchrone) Kommunikation von einem Objekt zu einem Anderen:

- *atomar* (nicht unterbrechbar)
- Beinhaltet *Hardware* und Realwelt-Objekte, z.B., Nachrichteneingang, input Ereignis, Zeitüberschreitung, ...
- Notation: *eventName(parameter: type, ...)*
- Kann das Objekt zu einer *Transition* zwischen Zuständen veranlassen

Definition...

Ein Zustand ist eine Zeitperiode, bei der ein Objekt auf ein Ereignis **wartet**:

- Dargestellt als **abgerundete Box** mit (bis zu) drei Sektionen:
 - *name* — optional
 - *state variables* — name: type = value (valid only for that state)
 - *triggered operations* — internal transitions and ongoing operations
- Kann **geschachtelt** sein

Beispiel

- Zustand eines Objektes

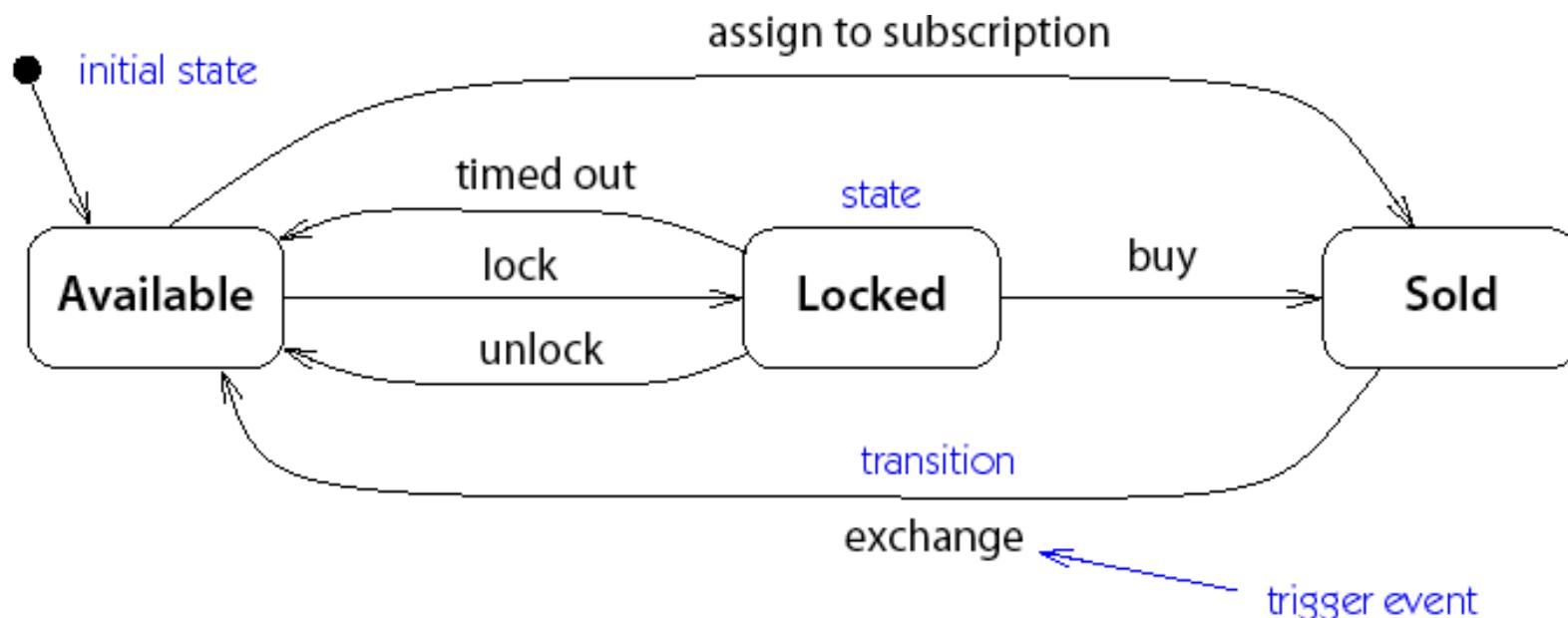


Figure 3-5. Statechart diagram

Statusbox mit Regionen

Das **Eingangs-Event** tritt auf, wann immer eine Transition zu diesem Zustand getätigigt wird.

Das **Ausgangs-Event** tritt auf, wenn eine Transition aus diesem Zustand hinaus führt.

Die **Hilfs-** und **Zeichenereignisse** lösen interne Transitionen aus ohne den Zustand zu ändern, so dass keine Eingangs- oder Ausgangsoperation durchgeführt wird.

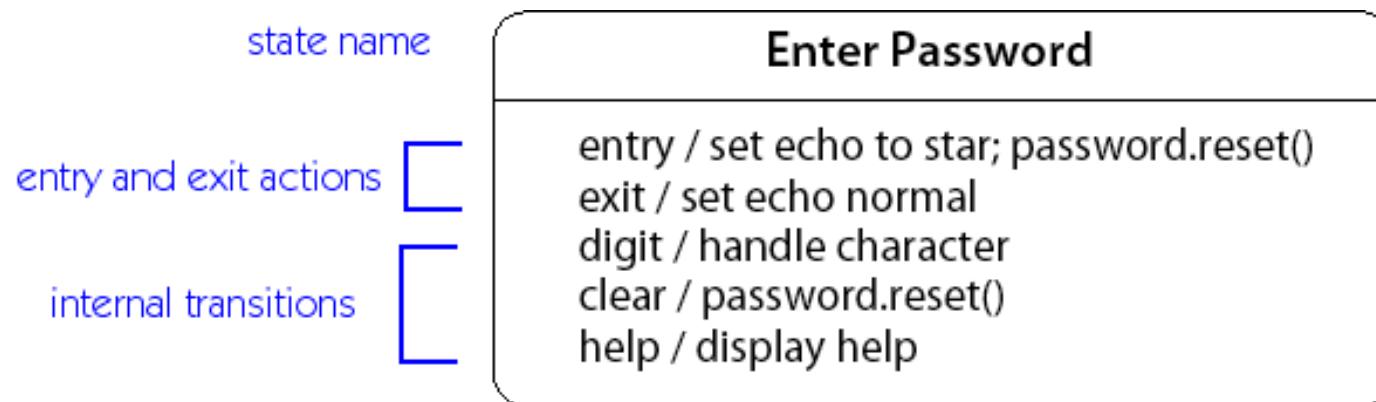


Figure 6-4. Internal transitions, and entry and exit actions

Transitionen

Eine Transition ist eine *Antwort auf ein externes Ereignis* welches das Objekt in einem **bestimmten Zustand** erhalten hat

- Kann zur *Ausführung* einer Operation und zum Wechsel des Zustands des Objekts führen
- Kann ein Ereignis zu einem anderen externen Objekten *senden*
- Transitionssyntax (jeder Teil ist optional):
event(arguments) [condition]
/ target.sendEvent operation(arguments)
- *Externe Transitionen* markieren Kreisbögen zwischen Zuständen
- *Interne Transitionen* sind Teil der ausgelösten Operationen eines Zustandes

Operationen und Aktivitäten

Eine Operation ist eine *atomare Aktion* angestoßen von einer Transition

- *Eingangs- und Ausgangsoperationen* können mit Zuständen assoziiert werden

Eine Aktivität ist eine *laufende Operation* die dann läuft, während ein Objekt in einem bestimmten Zustand ist

- Modelliert als “interne Transitionen” markiert mit dem pseudo-event **do**

Schachtelung: Nested Statecharts

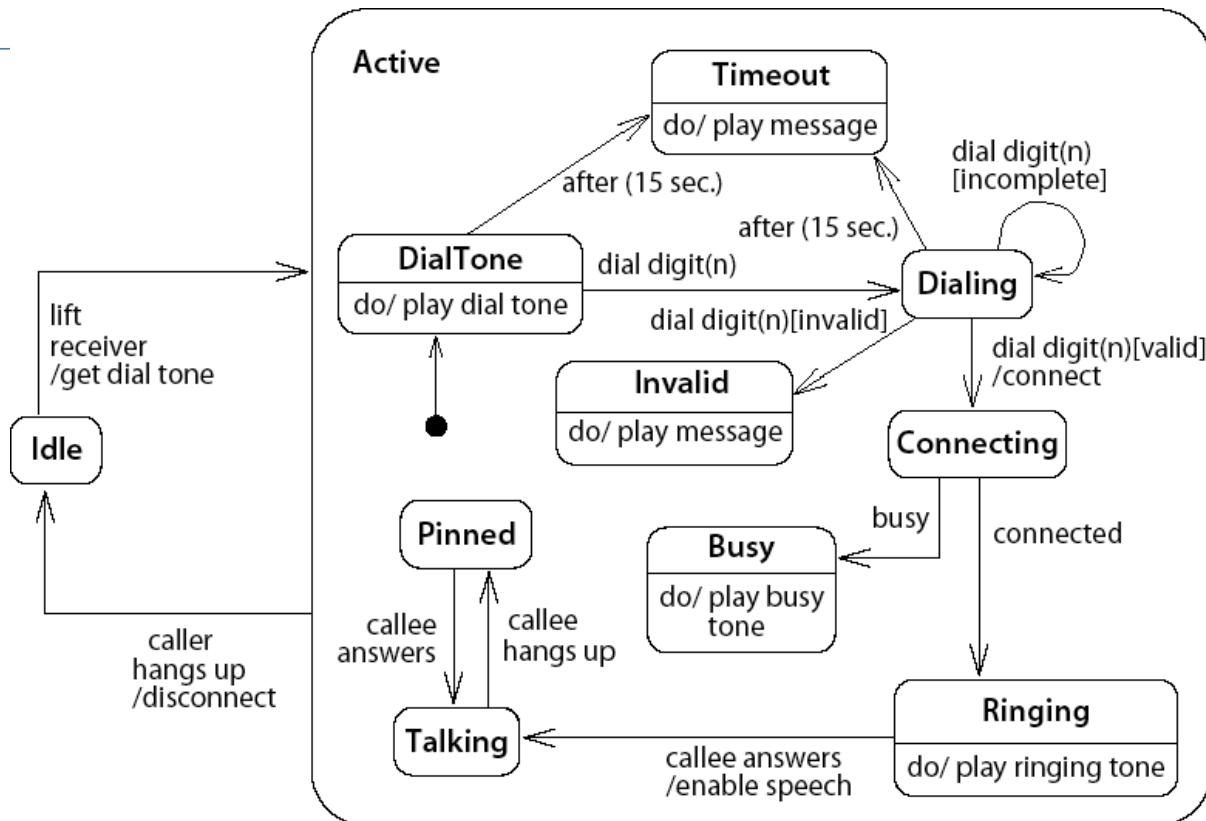


Figure 13-169. State diagram

Nested Statecharts

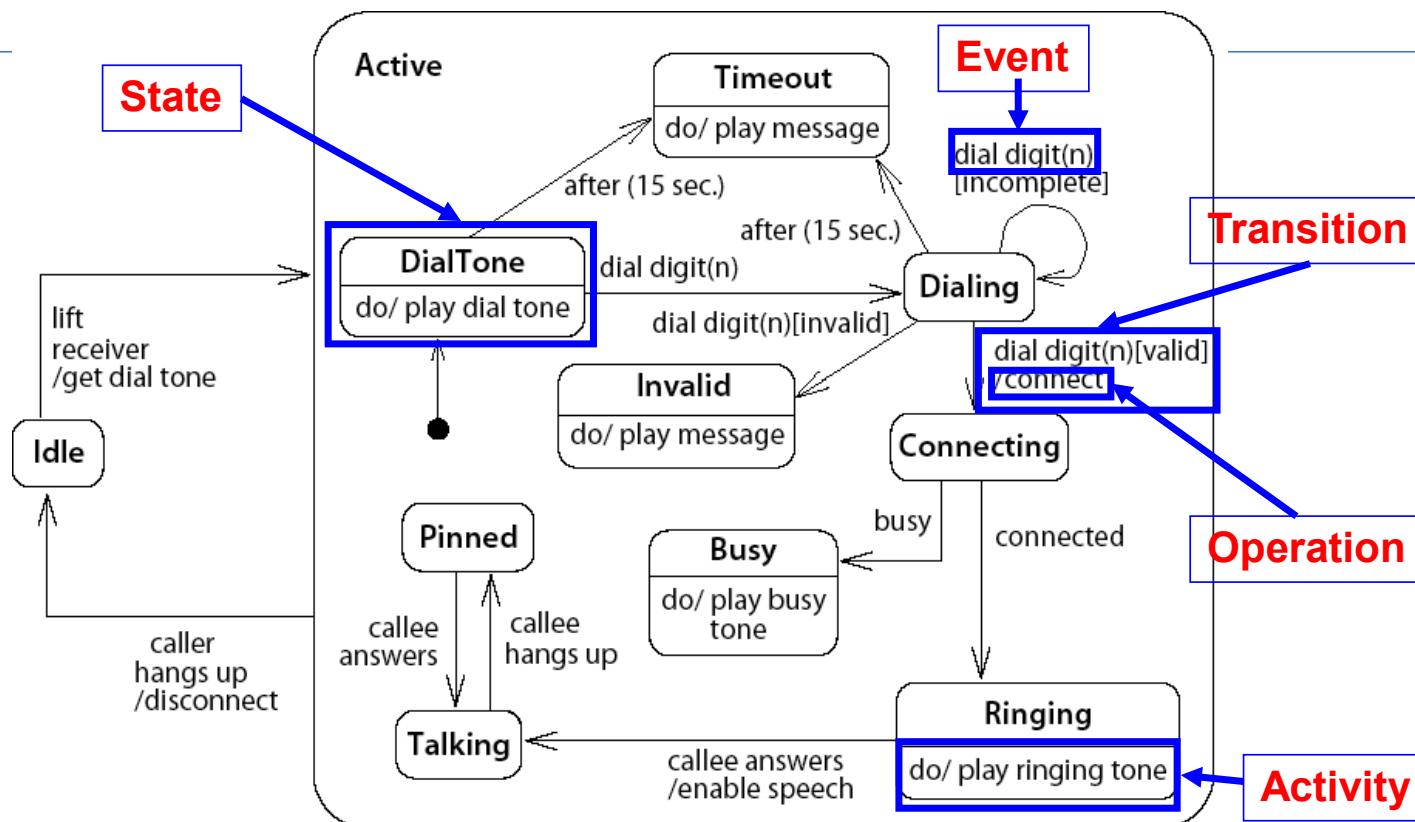


Figure 13-169. State diagram

Aufgabe

- Modellieren Sie ein Flugzeug-Objekt, welches den Zustand des Flugzeuges bzgl. der Platzreservierung wiedergibt. Definieren Sie geeignete Zustandsübergänge und evtl. Bedingungen dafür.

UML Benutzung: Perspektiven

Perspektiven

Drei Perspektiven beim Erstellen von UML Diagrammen:

1. *Konzeptionell*

- Repräsentieren Domänenkonzepte
 - Ignoriere Software Belange

2. *Spezifikation*

- Fokus auf sichtbare Interfaces und Verhalten
 - Ignoriere interne Implementierung

3. *Implementierung*

- Dokumentiere Implementierungsentscheidungen
 - Häufigste, aber am wenigsten nützliche Perspektive (!)

—UML Distilled



Was Ihr mitgenommen haben solltet

- Was ist der Zweck von use case Diagrammen?
- Warum beschreiben Szenarien Objekte und nicht Klassen?
- Wie können zeitliche Bedingungen in Szenarien beschrieben werden?
- Wie spezifiziert und interpretiert man Nachrichten-Labels in einem Szenario?
- Wie benutzt man genestete Zustandsdiagramme, um Objektverhalten zu modellieren?
- Was ist der Unterschied zwischen “externen” und “internen” Transitionen?

Literatur

- *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson and Grady Booch, Addison Wesley, 1999.

Softwaretechnik

Design Patterns (Entwurfsmuster)



SOFTWARE
SYSTEME

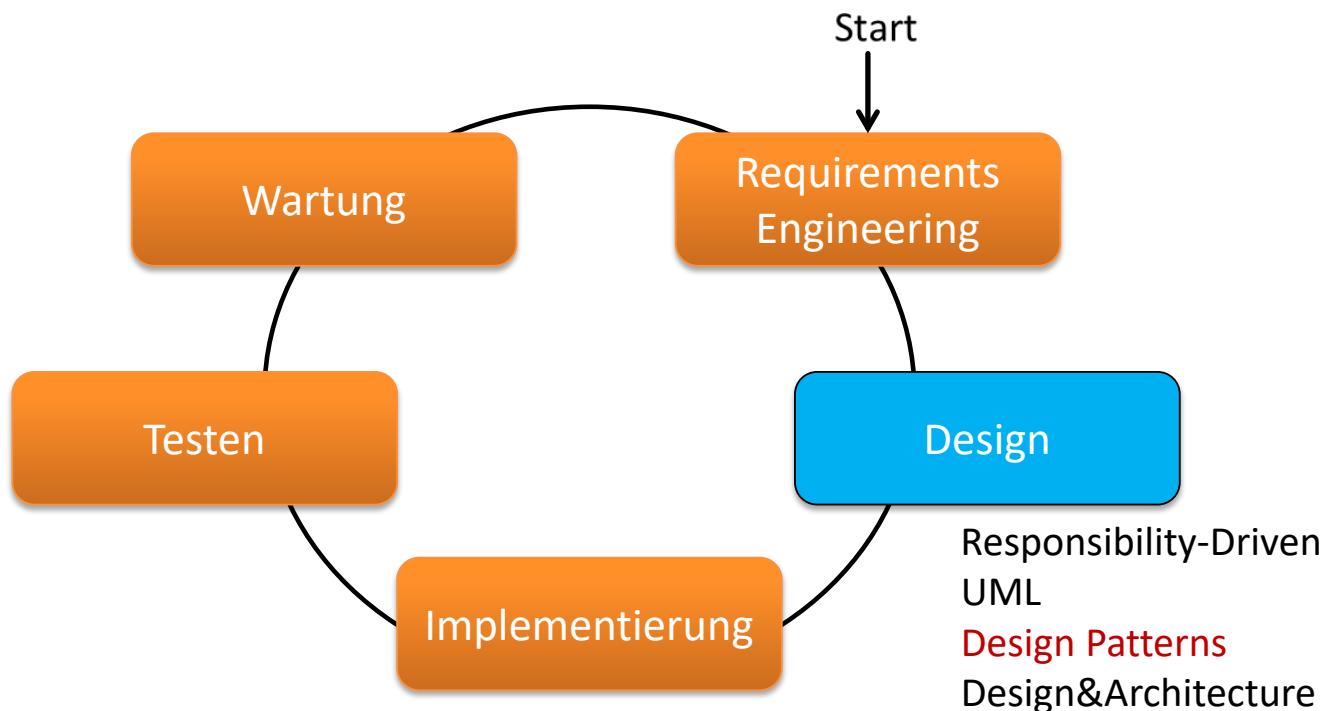
Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

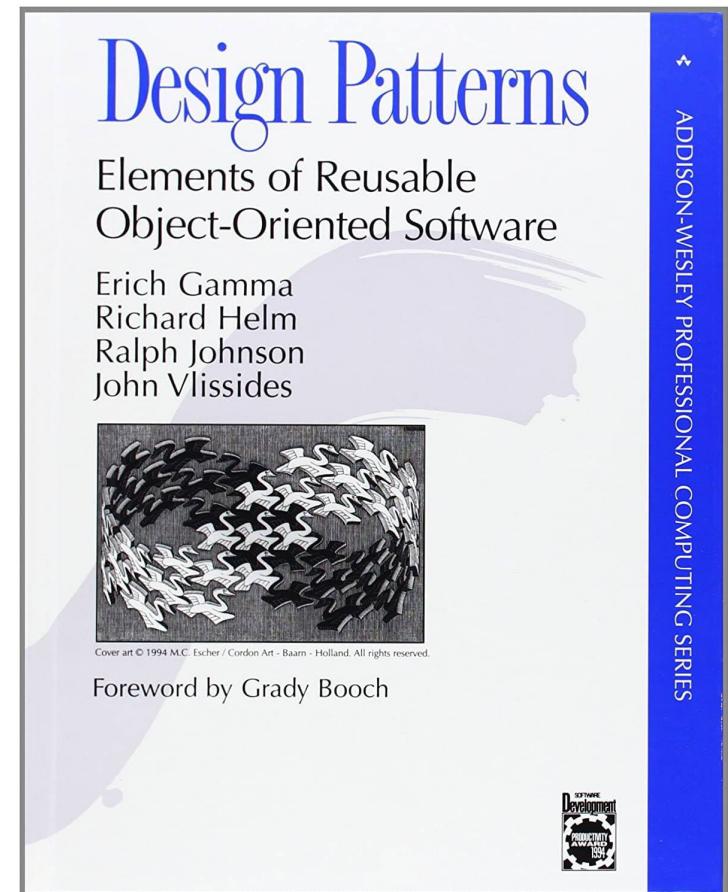
"Designing object-oriented software is hard and designing reusable object-oriented software is even harder."
—Erich Gamma

Einordnung



Inhalt

- Design Patterns
 - Warum notwendig?
 - Klassifizierung von Patterns
 - Beschreibung von Patterns
 - Kennenlernen von Patterns
 - Adapter
 - Iterator
 - Decorator
 - Observer
 - Visitor
 - Auch wichtig (Singleton, Factory Method, Strategy, State, Lazy loading, active record, DAO, Composite, Mediator, Facade)
- Bonus: Anti-Patterns



Lernziele

- In der Lage sein:
 - Die Nützlichkeit von Design Patterns zu erklären und dabei deren Unzulänglichkeiten zu kennen
 - Klar und präzise Beispiele für Situationen in der Softwareentwicklung zu benennen, bei denen Design Patterns hilfreich sind
 - Wichtig Patterns benennen und implementieren können
 - Situationen kennen, um Design Patterns anwenden zu können
- Wichtiges „Handwerkszeug“ von Software-Entwicklern kennen
- In Zukunft bei der Implementierung Wiederverwendung im Hinterkopf haben (loose coupling, Interfaces, Vererbung, etc.)

Warum Design Pattern?



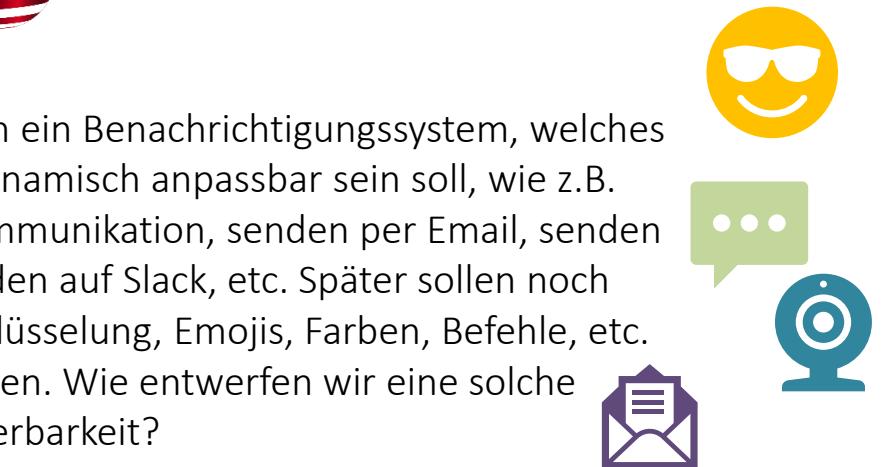
The central word in the cloud is **Design**, which is also the title of the slide. Surrounding **Design** are various design pattern names and associated concepts:

- Behavioral**: Iterator, Command, Mediator, Factory, Refactoring, Classes.
- Structural**: Chain, Code Analysis, Singleton, Composite, Abstract Factory.
- Creational**: Factory, Refactoring, Classes, Dependency Injection, Adapter, Decorator, Observer, Façade.
- Proxy**: Strategy, Agile Method, Template Method.
- Software Engineering**: Driver Development.

Entwurf einer Autoverkaufssoftware, die Grundinformationen (wie z.B. Preis und Geschwindigkeit) zu Autos liefert. Unsere Software basiert auf EU-Daten (Preis in Euro, Abmaße in metrischen System). Wir wollen nun in den US-Markt expandieren, doch deren Daten sind inkompatibel mit unseren. Wie entwerfen wir das System, so dass wir trotzdem die EU-spezifischen Daten für die USA verwenden können?



Wir implementieren ein Benachrichtigungssystem, welches von den Nutzern dynamisch anpassbar sein soll, wie z.B. einfache TCP/IP Kommunikation, senden per Email, senden auf Mastodon, senden auf Slack, etc. Später sollen noch Features für Verschlüsselung, Emojis, Farben, Befehle, etc. dazu kommen können. Wie entwerfen wir eine solche dynamische Erweiterbarkeit?



Bei der Erstellung eines Labyrinths für ein Spiel können wir unterschiedliche Settings haben (z.B. Untergrund, Natur, geheimnisvoll, herausfordernd, etc.). Ein Labyrinth besteht dabei aus Räumen, die entsprechend des Settings instanziert werden müssten. Wie entwerfen wir die Erstellung, so dass der Code nicht für jedes Setting neu geschrieben werden muss?



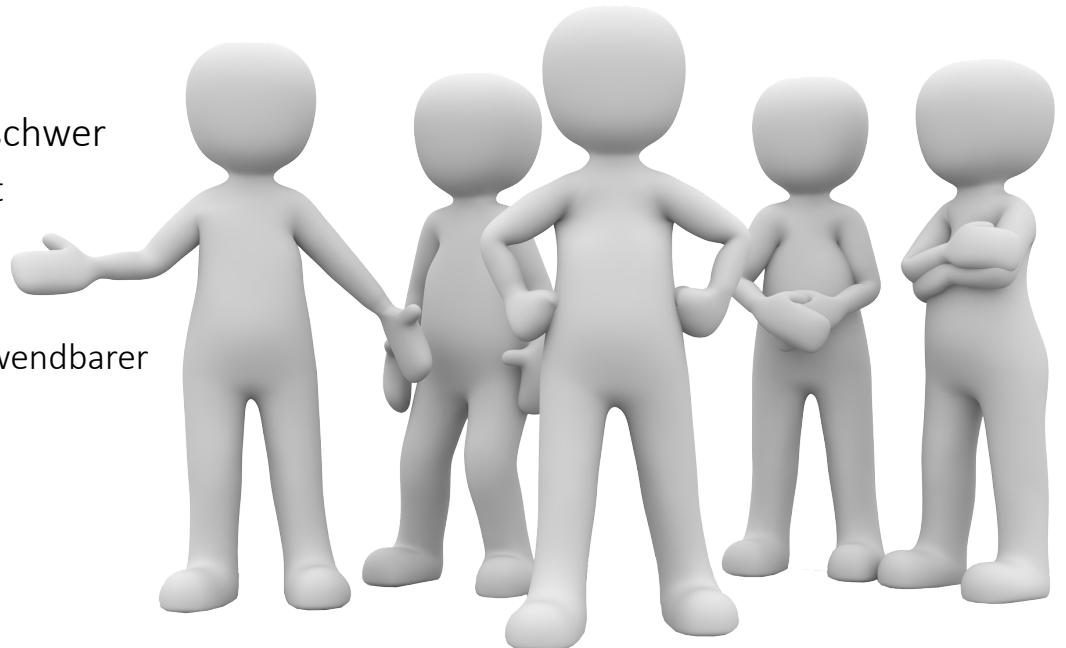
Was sind Patterns (Muster)?

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
 - Muster für ein wiederkehrendes Problem im Design (Entwurf) eines Softwaresystems.
-
- Jedes Entwurfsmuster ist ein Triple, welches für einen bestimmten **Kontext** ein **Problem** beschreibt und dessen **Lösung**.



Warum Patterns benutzen?

- Gemeinsame Sprache für Entwicklerinnen
 - Verbessert Kommunikation
 - Beugt Missverständnisse vor
- Lernen aus Erfahrung
 - Eine gute Designerin / Entwicklerin zu werden, ist schwer
 - Gute Designs kennen / verstehen ist der erste Schritt
 - Erprobte Lösungen für wiederkehrende Probleme
 - Durch Wiederverwendung wird man produktiver
 - Eigene Software wird selbst flexibler und wiederverwendbarer



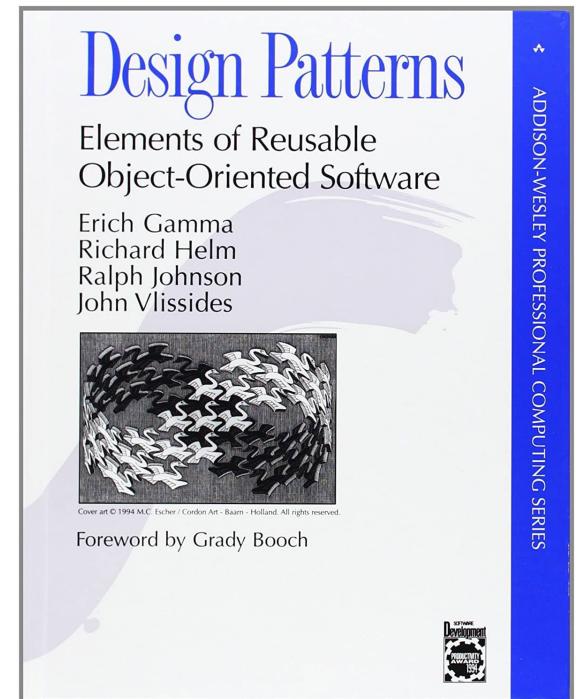
"Dieses Foto" von Unbekannter Autor ist lizenziert gemäß CC BY-NC

Zusammenhang Patterns und OOP

- Fundamentale OOP-Design-Prinzipien:
 - Patterns folgen Design-Zielen
 - Modularität, Explizite Interfaces, Information Hiding, ...
 - Patterns entstehen aus OOP Design-Prinzipien
 - Design nach Schnittstellen
 - Favorisiere Komposition über Vererbung
 - Finde Variabilität und kapsele sie
 - Patterns werden entdeckt und nicht erfunden
 - „Best Practice“ von erfahrenen Entwickelnden

Gang of Four (GoF) Design Patterns

- “The Gang of Four”: Erich Gamma, Richard Helm, Ralph Johnson, und John Vlissides
 - “A design pattern names, abstracts, and identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.”
- Klassifikation über Zweck und Anwendungsbereich



Klassifikation I

- Zweck
 - Creational Patterns
 - Helfen bei der Objekterstellung
 - Structural Patterns
 - Helfen bei der Komposition von Klassen und Objekten
 - Behavioral Patterns
 - Helfen bei der Interaktion von Klassen und Objekten (Verhalten kapseln)

Klassifikation II

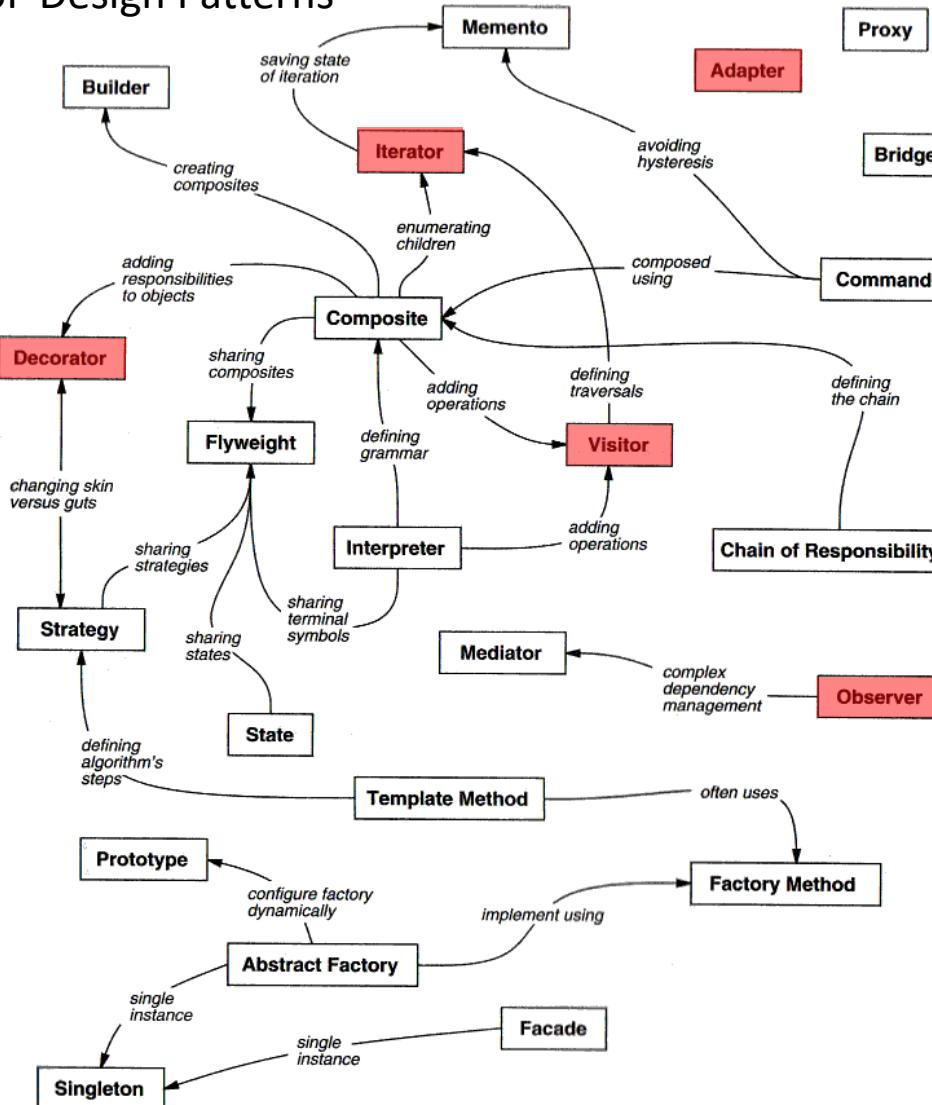
- Anwendungsbereich
 - Class Patterns
 - Fokussieren auf die Beziehung zwischen Klassen und ihren Subklassen (Wiederverwendung mittels Vererbung)
 - Object Patterns
 - Fokussieren auf die Beziehung zwischen Objekten (Wiederverwendung mittels Komposition)

Beschreibung eines Patterns

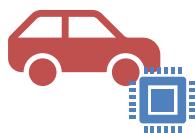
Beschreibung	Erläuterung	Beschreibung	Erläuterung
Pattern Name und Klassifikation	Präziser Name des Patterns	Konsequenzen	Vor- und Nachteile des Patterns
Zweck	Was das Pattern bewirkt	Implementierung	Hinweise und Techniken zur Implementierung
Auch bekannt als	Alternativer Name	Beispiel Code	Code Fragmente
Motivation	Szenario, wo das Pattern sinnvoll ist	Bekannte Verwendungen	Beispiele in realen Systemen
Anwendbarkeit	Situationen, wann das Pattern angewendet werden kann	Verwandte Pattern	Auflistung und Beschreibung der Verwandten Pattern
Struktur	Grafische Repräsentation		
Teilnehmer	Involvierten Klassen und Objekte		
Kollaborationen	Wie arbeiten die Teilnehmer zusammen		

Wichtige Design Patterns...

Beziehungen der GoF Design Patterns



Source: Design Patterns. Elements of Reusable Object-Oriented Software.



Entwurf einer Autoverkaufsssoftware, die Grundinformationen (wie z.B. Preis und Geschwindigkeit) zu Autos liefert. Unsere Software basiert auf EU-Daten (Preis in Euro, Abmaße in metrischen System). Wir wollen nun in den US-Markt expandieren, doch deren Daten sind inkompatibel mit unseren. Wie entwerfen wir das System, so dass wir trotzdem die EU-spezifischen Daten für die USA verwenden können?

Adapter – Structural Pattern I



Beschreibung	Inhalt
Pattern Name und Klassifikation	Adapter – Structural Pattern
Zweck	Konvertiert das Interface einer existierenden Klasse, so dass es zu dem Interface eines Klienten (Client) passt. Ermöglicht, dass Klassen miteinander interagieren können, was sonst nicht möglich wäre aufgrund der Unterschiede im Interface.
Auch bekannt als	Wrapper Pattern oder Wrapper
Motivation	Eine existierende Klasse bietet eine benötigte Funktionalität an, aber implementiert ein Interface, was nicht den Erwartungen eines Clients entspricht.



Adapter – Structural Pattern II

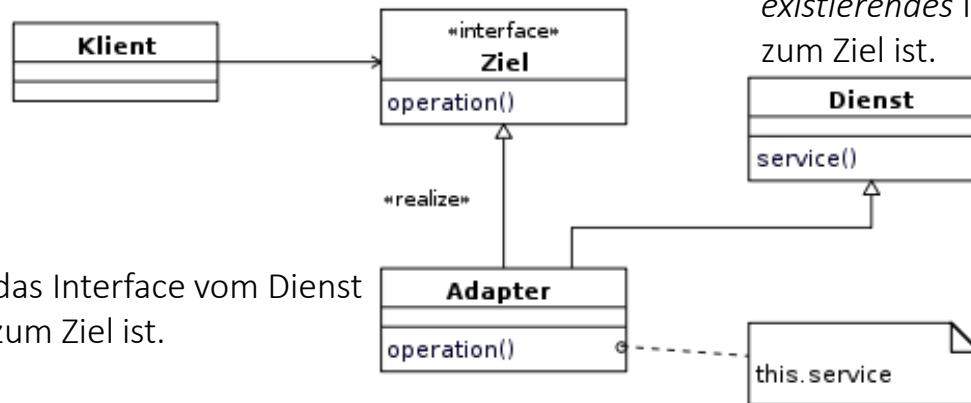
Beschreibung	Inhalt
Anwendbarkeit	<ul style="list-style-type: none">- Verwende eine ansonsten nicht wiederverwendbare (durch Interface-Inkompatibilität) Klasse wieder: Adaptiere das Interface durch das Ändern der Methodensignaturen.- Existierende Klasse bietet nicht die benötigte Funktionalität an: Implementiere die benötigte Funktion in Adapterklasse durch neue Methoden, die zum Interface passen
Struktur	Siehe nächste Folien
Teilnehmer	Siehe nächste Folien
Kollaborationen	Klienten rufen Methoden des Adapters auf, die die Anfragen an die adaptierte Klasse (Dienst) weiterleiten.

Adapter – Structural Pattern III

- Struktur Variante 1:
 - Adapter nutzt multiple Vererbung um ein Interface auf ein anderes passend zu machen
 - Adapter erbt von Ziel und Dienst (zu adaptierende Klasse)
 - Ziel muss Interface-Definition sein bei Sprachen ohne Mehrfachvererbung (wie Java)

Klient -> Interagiert mit Objekten, die das Zielinterface implementieren.

Ziel -> Definiert das domänen spezifische Interface, welches der Klient nutzt

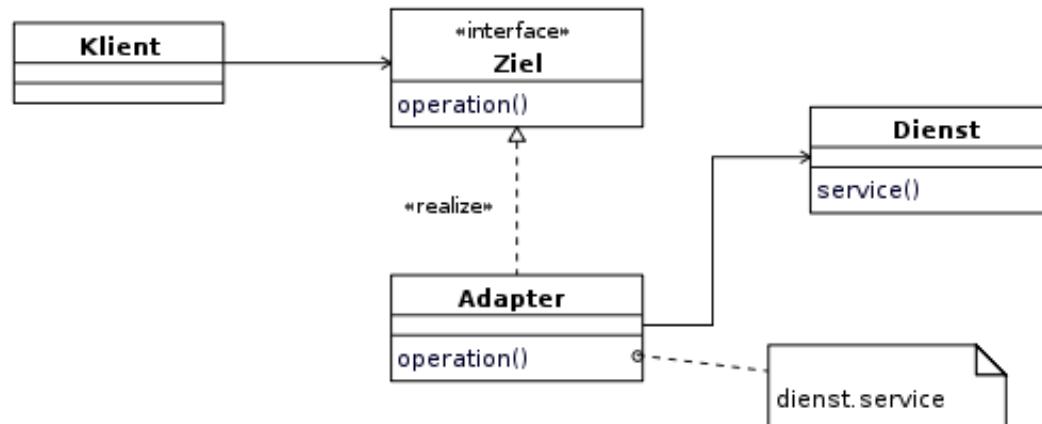


Adapter -> Adaptiert das Interface vom Dienst damit es kompatibel zum Ziel ist.

Dienst (zu adaptierende Klasse) -> Repräsentiert existierendes Interface, welches nicht kompatibel zum Ziel ist.

Adapter – Structural Pattern IV

- Struktur Variante 2:
 - Adapter nutzt Delegation, um Aufrufe weiterzuleiten
 - Adapter hält eine Referenz auf Dienst (zu adaptierende Klasse)
 - Methodenaufrufe werden vom Adapter zum Ziel weitergeleitet



Adapter – Structural Pattern VI

Beschreibung	Erläuterung
Konsequenzen	Klasse Adapter – Überschreibung der Methoden der Superklasse (Dienst) ist möglich Objekt Adapter – Dienst muss vererbbar sein, um Methoden überschreiben zu können Rate der Anpassbarkeit hängt vom Unterschied der Interfaces zwischen Ziel und Dienst ab
Implementierung	Siehe nächste Folien
Beispiel Code	Siehe nächste Folien
Bekannte Verwendungen	GUI Frameworks verwenden existierende Klassenhierarchien, müssen aber adaptiert werden
Verwandte Pattern	Decorator -> Reichert Objekt um Funktionalität an, ohne das Interface zu ändern Bridge -> separiert Interface und Implementierung, so dass unterschiedliche Implementierungen leicht austauschbar sind

Aufgabe

- Wie sieht der Beispielcode für folgendes Szenario aus:
 - Ziel: Stack
 - Dienst: List
- Implementieren Sie einen Stack mittels des Adapter Patterns bei denen Sie bereits implementierte Funktionen einer generischen Liste wiederverwenden können

Adapter – Structural Pattern VII

Ziel

```
interface Stack<T> {  
    public void push (T t);  
    public T pop ();  
    public T top ();  
}
```

Adapter

```
class DListImpStack<T> extends DList<T> implements Stack<T> {  
    public void push (T t) {  
        insertTail (t);  
    }  
    public T pop () {  
        return removeTail ();  
    }  
    public T top () {  
        return getTail ();  
    }  
}
```

Dienst (adaptierte Klasse)

```
class DList<T> {  
    public void insert (DNode pos, T t);  
    public void remove (DNode pos, T t);  
    public void insertHead (T t);  
    public void insertTail (T t);  
    public T removeHead ();  
    public T removeTail ();  
    public T getHead ();  
    public T getTail ();  
}
```

Adapter – Structural Pattern VII

Ziel

```
interface Stack<T> {
    public void push (T t);
    public T pop ();
    public T top ();
}
```

Dienst (adaptierte Klasse)

```
class DList<T> {
    public void insert (DNode pos, T t);
    public void remove (DNode pos, T t);
    public void insertHead (T t);
    public void insertTail (T t);
    public T removeHead ();
    public T removeTail ();
    public T getHead ();
    public T getTail ();
}
```

Adapter

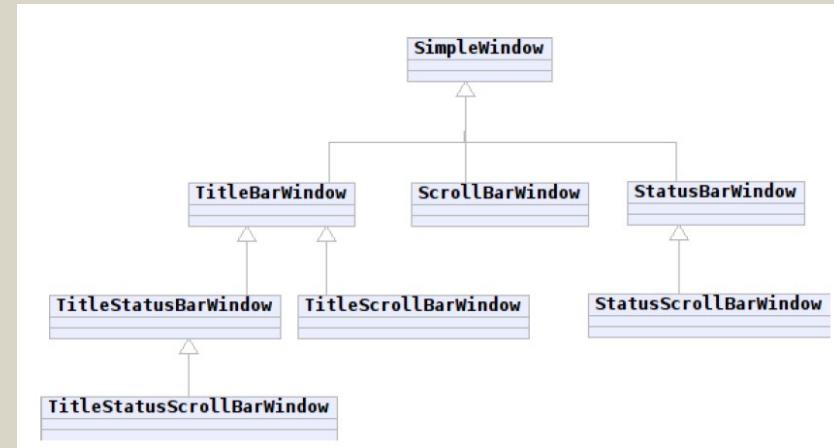
```
class DListToStackAdapter<T> implements Stack<T> {
    private DList<T> m_List;
    public DListToStackAdapter(DList<T> m_List) {
        this.m_List = m_List;
    }
    public void push (T t) {
        m_List.insertTail (t);
    }
    public T pop () {
        return m_List.removeTail ();
    }
    public T top () {
        return m_List.getTail ();
    }
}
```

Aufgabe

- Modellieren Sie folgenden Sachverhalt als UML-Klassendiagramm:
 - Ein Dialogsystem soll die Verwendung folgender Fenstertypen erlauben:
 - einfache Fenster, ohne Zusatzfunktionalität
 - Fenster die eine Titelleiste haben
 - Fenster die eine Statusleiste haben
 - Fenster die horizontal und vertikal “scrollbar” sind
 - alle daraus konstruierbaren “Featurekombinationen”, wie z.B. ein Fenster mit Titelleiste das horizontal und vertikal “scrollbar” ist

Lösung 1: Vererbung

- Probleme:
 - Explosion der Klassenhierarchie
 - TitleStatusScrollBarWindow versus ScrollStatusTitleBarWindow
 - Was passiert, wenn weitere Features (z.B. ColoredTitleBars, 3DScrollBars,...) berücksichtigt werden müssen?
 - Zur Laufzeit nicht änderbar

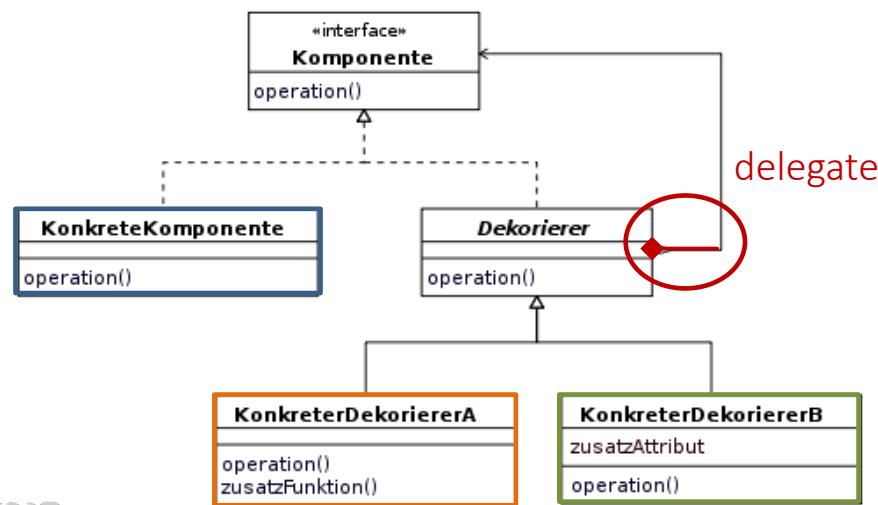


Decorator – Structural Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Decorator – Structural Pattern
Zweck	Fügt dynamisch Funktionalität zu bereits bestehenden Klassen hinzu.
Motivation	Wir benötigen flexible Implementierungen einer Klasse, die je nach Kontext unterschiedlich ausfallen.
Anwendbarkeit	Funktionale Erweiterungen sind optional. Anwendbar, wenn Erweiterungen mittels Vererbung unpraktisch ist.
Konsequenzen	Flexibler als statische Vererbung. Problem der Objektschizophrenie (ein Objekt ist zusammengesetzt aus mehreren Objekten). Viele kleine Objekte.

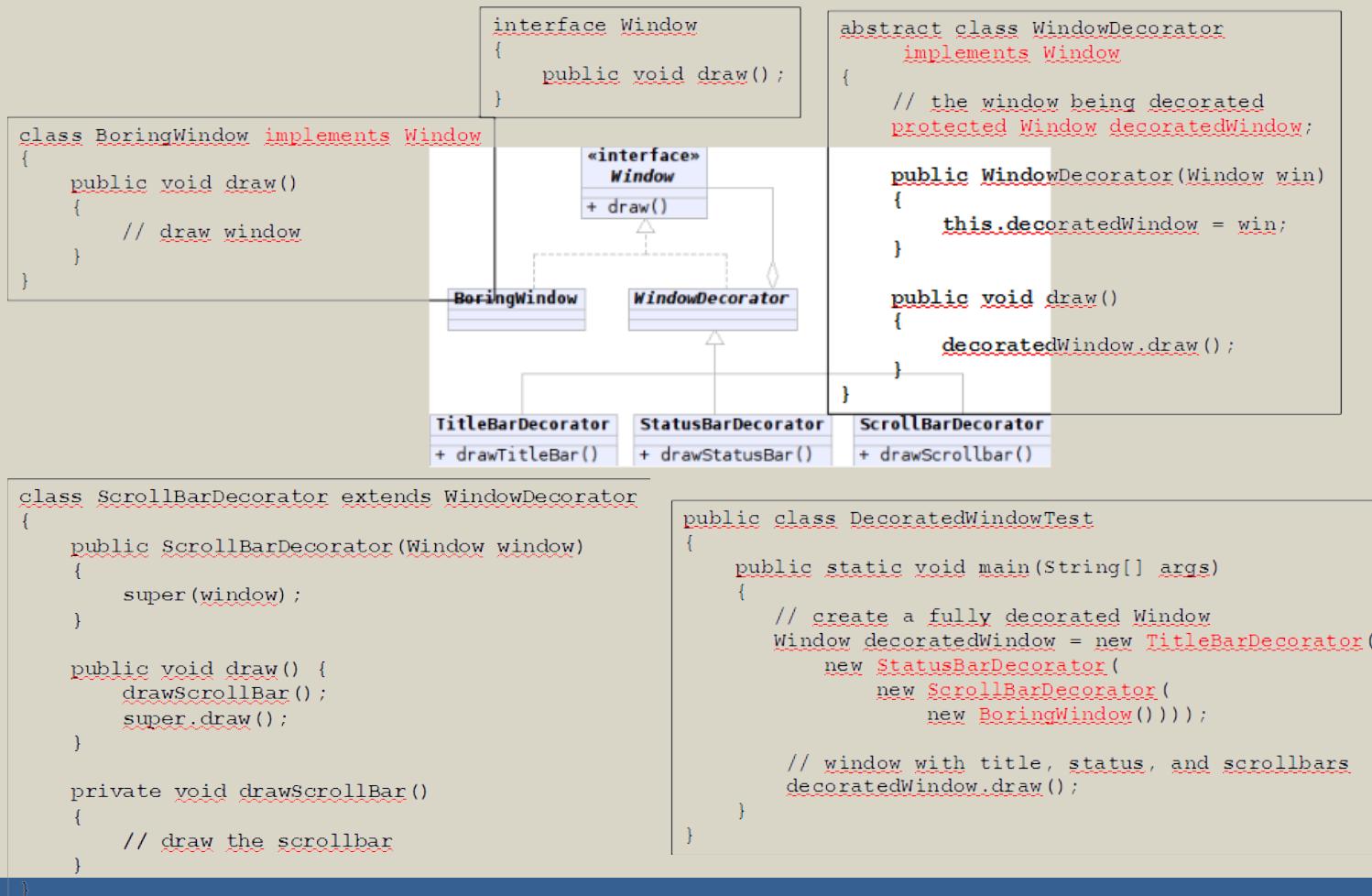
Decorator – Structural Pattern II

- Struktur:
 - Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet -> Funktionalität des Dekorierers wird zuerst ausgeführt
 - Dekorierer hat gleiche Schnittstelle wie zu dekorierende Klasse
 - Aufrufe werden weitergeleitet oder komplett selbst verarbeitet



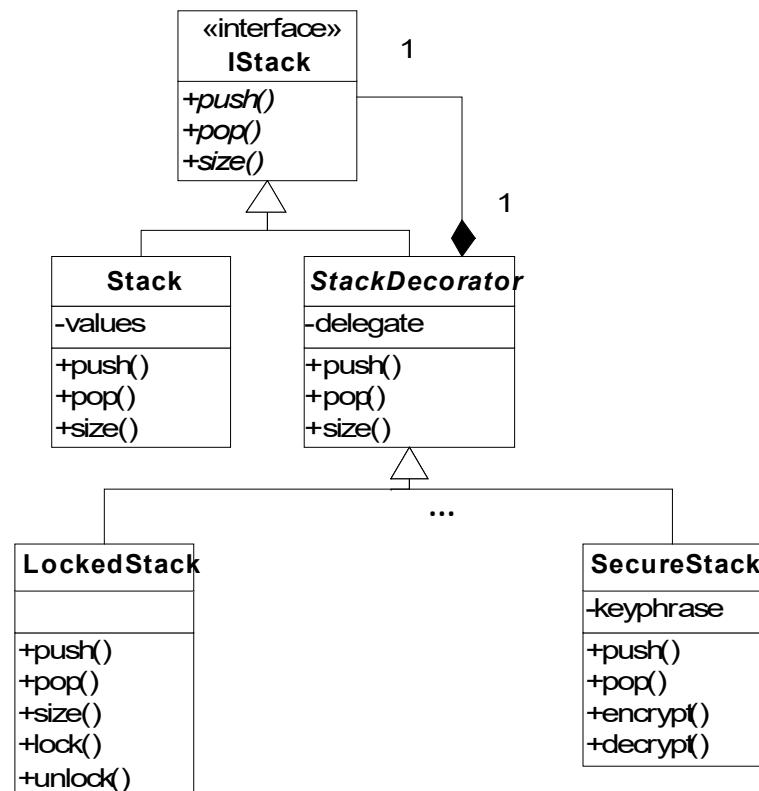
Klient ruft
`operation()` auf und
wird durch alle
Objekte delegiert

Bessere Lösung: Decorator Pattern



Decorator – Structural Pattern IV

- Struktur des Beispiels



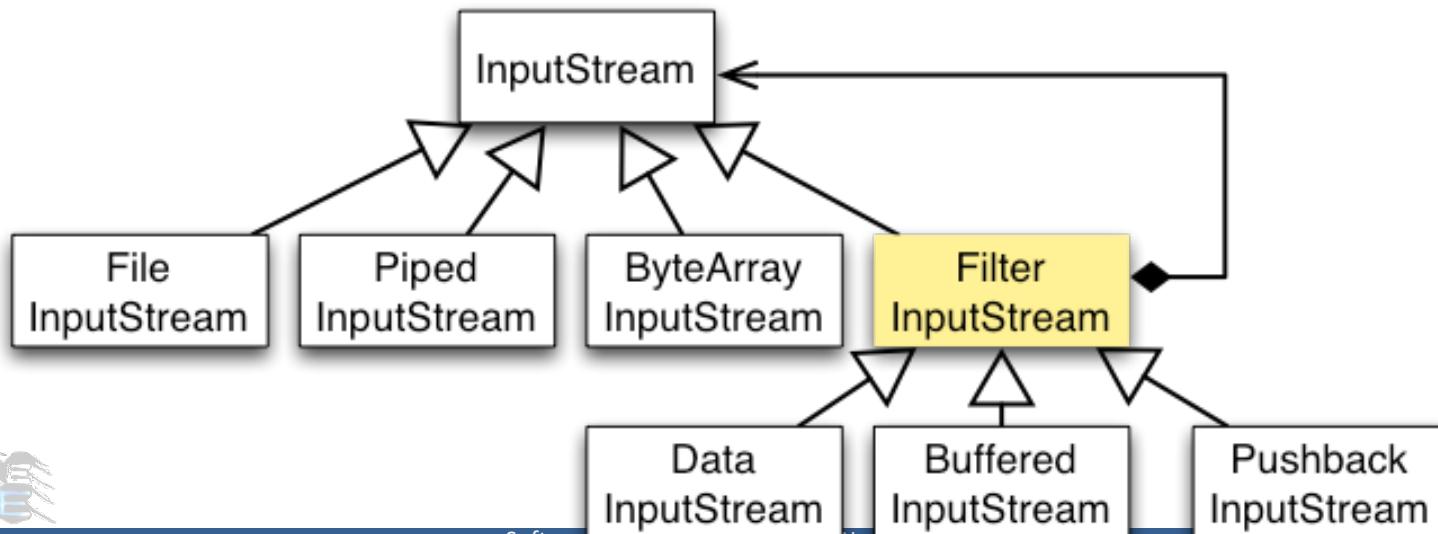
Decorator in Java

- java.io enthält verschiedene Funktionen zur Ein- und Ausgabe:
 - Programme **operieren auf Stream-Objekten** ...
 - **Unabhängig** von der Datenquelle/-ziel und der Art der Daten

```
FileInputStream fis = new FileInputStream("my.txt");
```

```
BufferedInputStream bis = new BufferedInputStream(fis);
```

```
GziplnputStream gis = new GziplnputStream(new ObjectlnputStream(bis));
```



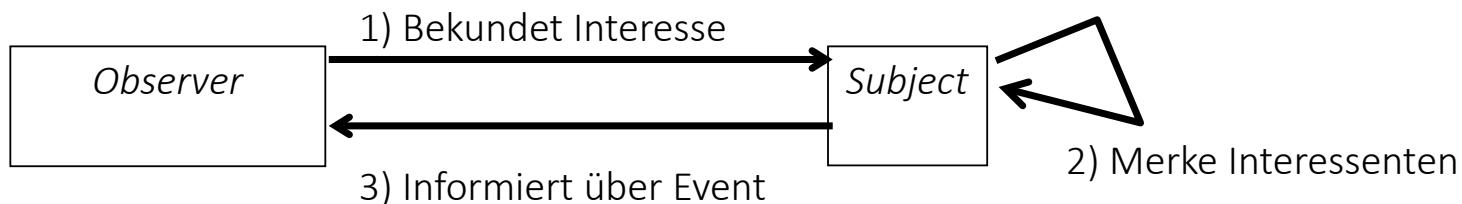


Observer – Behavioral Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Observer – Behavioral Pattern
Zweck	Objekt verwaltet Liste von abhängigen Objekten und teilt diesen Änderungen mit
Auch bekannt als	Publish-Subscribe
Motivation	Implementiert verteilte Ereignisbehandlung (bei einem Ereignis müssen viele Objekte informiert werden). Schlüsselfunktion beim Model-View-Controller (MVC) Architektur-Pattern
Anwendbarkeit	Wenn eine Änderung an einem Objekt die Änderung an anderen Objekten erfordert und man weiß nicht, wie viele abhängige Objekte es gibt. Wenn ein Objekt andere Objekte benachrichtigen will, ohne dass es die Anderen kennt.

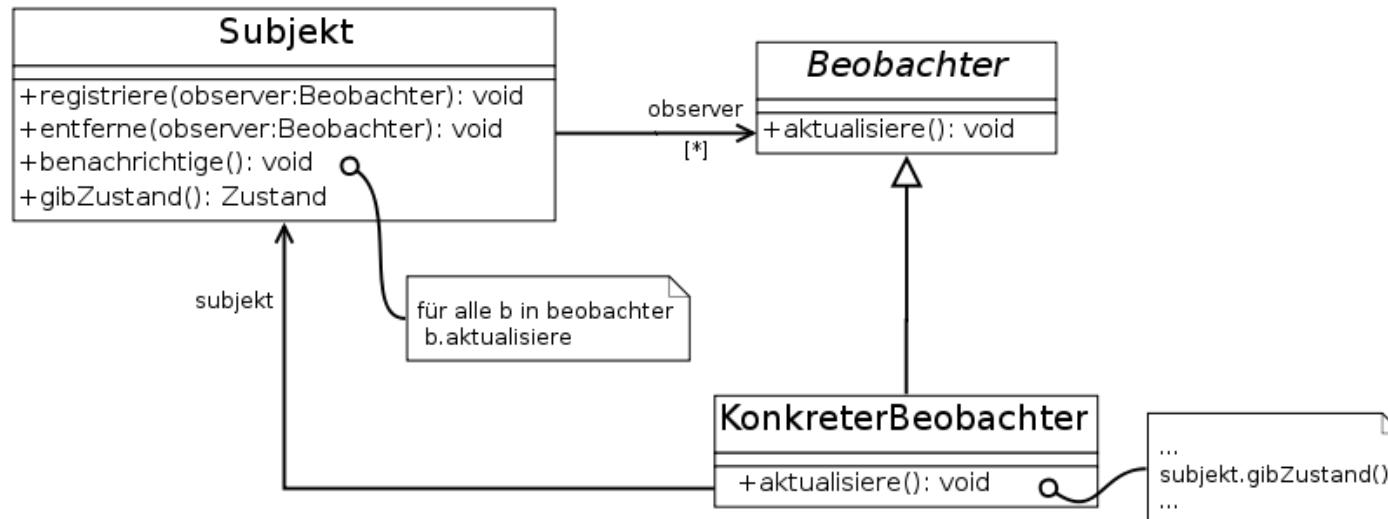
Observer – Behavioral Pattern II

Beschreibung	Inhalt
Konsequenzen	<p>Loose coupling (Lose Kopplung) von Objekten verbessert Wiederverwendung.</p> <p>Unterstützt „Broadcast“ Kommunikation (eine Nachricht an alle Teilnehmer verschicken).</p> <p>Mitteilungen können zu weiteren Mitteilungen führen und sich somit aufschaukeln.</p>



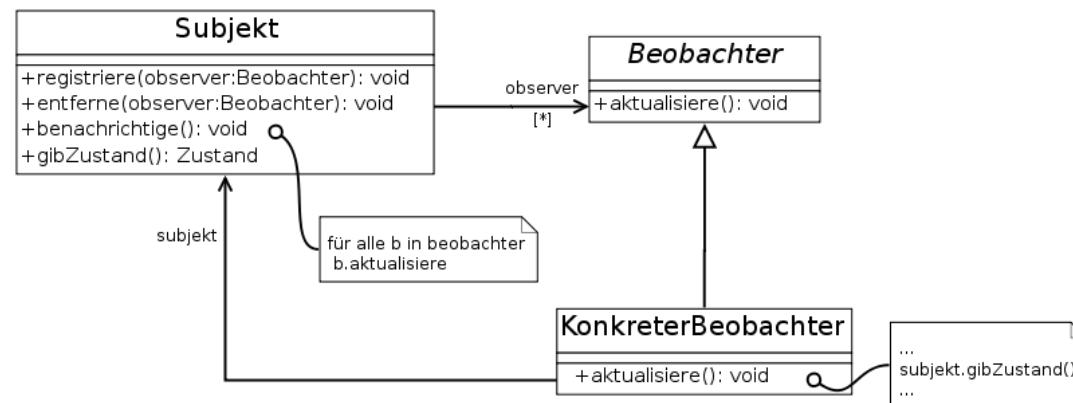
Observer – Behavioral Pattern III

- Struktur



Observer – Behavioral Pattern IV

- Struktur
- Code



```
interface IObserver {  
    public void update(String message);  
}  
interface ISubject {  
    public void registerObserver(Observer observer);  
    public void removeObserver(Observer observer);  
    public void notifyObservers();  
}
```

Aufgabe

- Wie sieht die Realisierung des Observer Patterns aus für eine Client-Server Kommunikation?
 - Mehrere unterschiedliche Clients verbinden sich mit Server und wollen über Änderungen informiert werden

Observer – Behavioral Pattern V

```
class Server implements Subject {  
    private ArrayList<Observer> observers  
        = new ArrayList<Observer>();  
    String message;  
    public void postMessage(String message) {  
        this.message = message;  
        notifyObservers();  
    }  
    @Override  
    public void registerObserver(Observer observer) {  
        observers.add(observer);  
    }  
    @Override  
    public void removeObserver(Observer observer) {  
        observers.remove(observer);  
    }  
    @Override  
    public void notifyObservers() {  
        for (Observer ob : observers) {  
            ob.update(this.message);  
        }  
    }  
}
```

```
class WebClient implements Observer {  
    @Override  
    public void update(String message) {  
        postOnWebPage(message);  
    }  
    ...  
}  
class Messenger implements Observer {  
    @Override  
    public void update(String message) {  
        System.out.println("Receiving message: " + message);  
    }  
}  
class Main {  
    public static void main(String [] args){  
        Server s = new Server();  
        WebClient wc = new WebClient();  
        s.registerObserver(wc);  
        Messenger m = new Messenger();  
        s.registerObserver(m);  
        s.postMessage("Hello World!");  
    }  
}
```

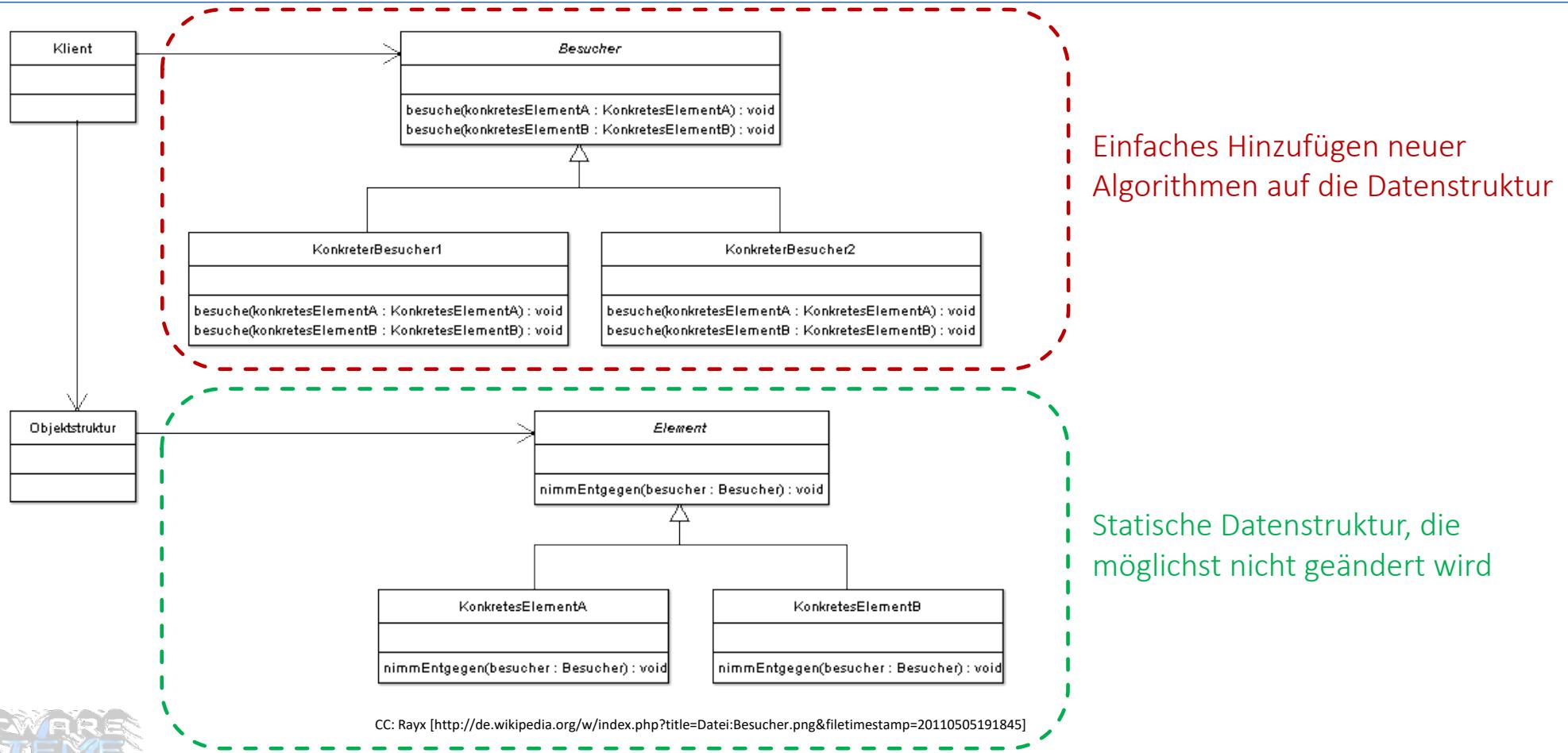
Visitor – Behavioral Pattern

Beschreibung	Inhalt
Pattern Name und Klassifikation	Visitor – Behavioral Pattern
Zweck	Trennung von Algorithmus und Daten auf denen der Algorithmus angewendet wird
Motivation	Durch die Trennung können neue Algorithmen / Funktionen auf existierenden Objekt(-strukturen) angewendet werden, ohne diese Objekte/Strukturen ändern zu müssen.
Anwendbarkeit	Struktur mit vielen Klassen vorhanden. Man möchte Funktionen anwenden, die abhängig von der jeweiligen Klasse sind. Menge der Klassen ist stabil. Man möchte neue Operationen hinzufügen.

Visitor – Behavioral Pattern II

Beschreibung	Inhalt
Konsequenzen	Einfach neue Operationen hinzufügen. Gruppiert verwandte Operationen in einem Visitor. Neue Elemente hinzufügen ist schwierig. Visitor kann Zustand speichern. Elemente müssen ein Interface bereitstellen / impl.

Visitor – Behavioral Pattern III



Aufgabe

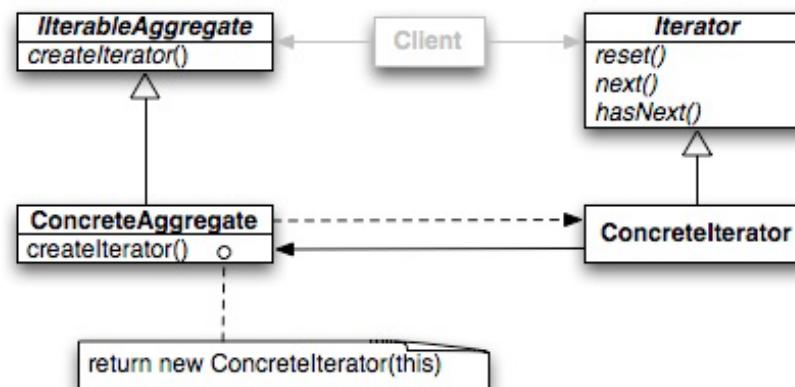
- Warum ist das Hinzufügen neuer Elemente schwierig?

Iterator – Behavioral Pattern I

Beschreibung	Inhalt
Pattern Name	Iterator – Behavioral Pattern
Zweck	Sequentieller Zugriff auf Elemente einer aggregierten Struktur, ohne diese zu kennen.
Auch bekannt als	Cursor
Motivation	Objekte werden häufige in einer Sammlung (z.B. Liste) zusammengefasst. Auf Elemente dieser Sammlung soll möglichst generisch und ohne Kenntnis von Implementierungsdetails zugegriffen werden können.
Anwendbarkeit	Elemente in Sammlung zusammengefasst. Zugriff auf Elemente unabhängig von der Implementierung der Sammlung.
Konsequenzen	Implementierung der Sammlung unsichtbar. Wenn Sammlung sich während Iteration ändert, kann es zu Fehlern kommen.

Iterator – Behavioral Pattern II

- Struktur
 - Aggregate = Sammlung von Elementen
 - IterableAggregate = Interface um Sammlung zu iterieren (spezifiziert nicht, welche Datenstruktur als Sammlung genutzt werden soll)
 - ConcreteAggregate = Implementierung dieses Interfaces
 - createliterator() gibt Objekt vom Typ Concreteliterator zurück
 - Concreteliterator implementiert Iterator Interface



Iterator – Behavioral Pattern III

```
public interface IterableAggregate {
    public Iterator createIterator();
}

public interface Iterator{
    public void remove(); //Löscht zuletzt zurück gegebenes Element aus Sammlung
    public Object next(); //Gibt nächstes Objekt zurück und setzt den Zeiger weiter in der Sammlung
    public boolean hasNext(); //Gibt true zurück, falls es noch weitere Elemente in der Sammlung gibt
}

public class SimpleList<T> implements IterableAggregate {
    private ArrayList<T> sammlung = new ArrayList<T>();
    @Override
    public Iterator createIterator() {
        return new SimpleListIterator(sammlung); }
    ... // Add, delete, etc.
}

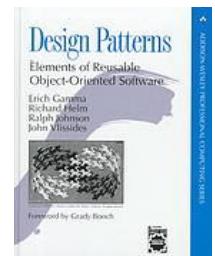
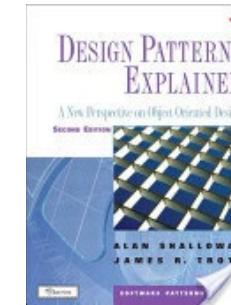
public class SimpleListIterator<T> implements Iterator{
    int index = -1;    ArrayList<T> sammlung; // Im Konstruktor initialisieren
    public Object next() {
        index++;
        return sammlung[index]; }
    public boolean hasNext() {
        return (index < (sammlung.length -1)); }
        //remove() nicht hier gezeigt, muss aber implementiert werden
}
```

Was Sie mitgenommen haben sollten

- Warum sind Design Patterns wichtig?
- Was sind folgende Pattern und wie realisiere ich sie?
 - Adapter
 - Iterator
 - Decorator
 - Observer
 - Visitor
- Was verbessern Design Patterns und wie sind sie klassifiziert?

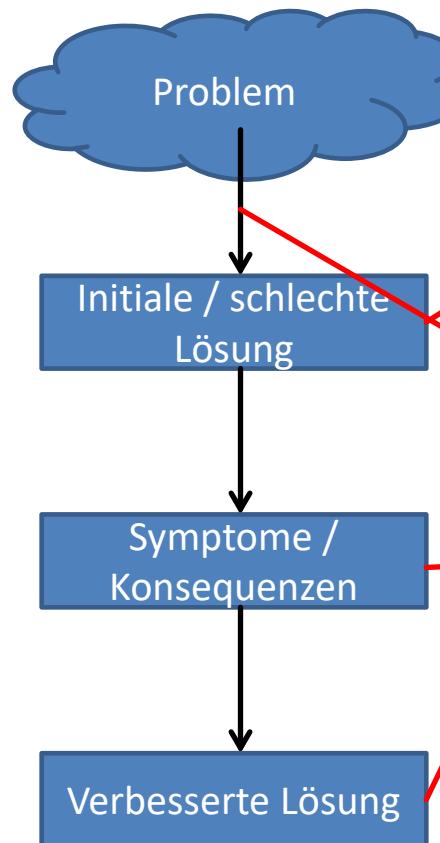
Literatur

- *Design Patterns. Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995.
- *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Alan Shalloway, James R. Trott, 2004.
 - Benutzt Java
 - Viele Beispiele -> Lesenswert!
- *Head First Design Patterns / Entwurfsmuster von Kopf bis Fuß*.
Eric Freeman, Elisabeth Robson, O'Reilly, 2021.
- Quelle Diagramme: Wikipedia



Anti-Patterns...

Anti-Patterns



- Anti-Patterns bestehen aus:
 - Nicht-optimalen Lösung
 - Optimale / verbesserte Lösung
- Beschreibung WIE und WARUM es zur nicht-optimalen Lösung kam
 - Ursächliche Wirkketten
 - Nach außen sichtbare Symptome

Beispiele für Symptome

- “Wozu ist diese Klasse eigentlich da?”
- Designdokumente und Code sind bestenfalls entfernte Verwandte
- Fehlerrate steigt mit jeder neuen Version an
- „Wenn die Liste mehr als 100 Einträge hat, sinkt die Performance in den Keller.“

The Blob

„Diese Klasse ist das Herzstück unserer Architektur.“

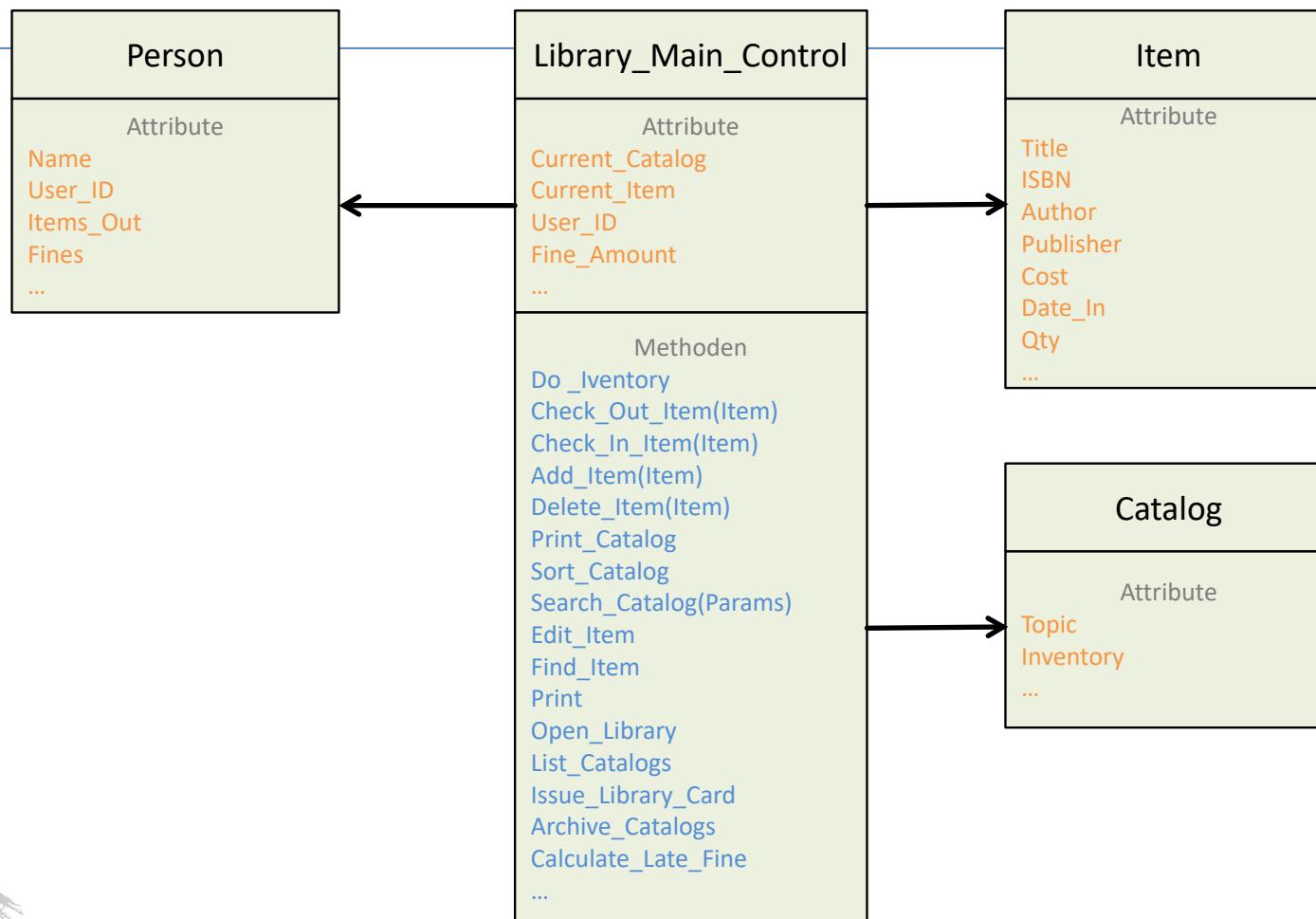


The Blob I



Beschreibung	Inhalt
Pattern Name	The Blob – Anti Pattern
Grund	Eile, Faulheit
Auch bekannt als	Winnebago and The God Class
Symptome	Klasse mit sehr vielen Methoden. Methoden und Klassen mit sehr unterschiedlichen Funktionen. Verbindung mit sehr vielen anderen Klassen, die jeweils wenige Methoden haben. Klasse zu komplex für Testen und Wiederverwendung.
Lösung	Refaktorisierung der Klasse anhand Verantwortlichkeiten. Ähnliche Attribute/Methoden identifizieren und kapseln. Methoden ggfs. verlagern in bereits existierende Klassen.
Konsequenzen	Performanceeinbußen. Schlechte Wartbarkeit. Kaum Wiederverwendbarkeit der Funktionen.

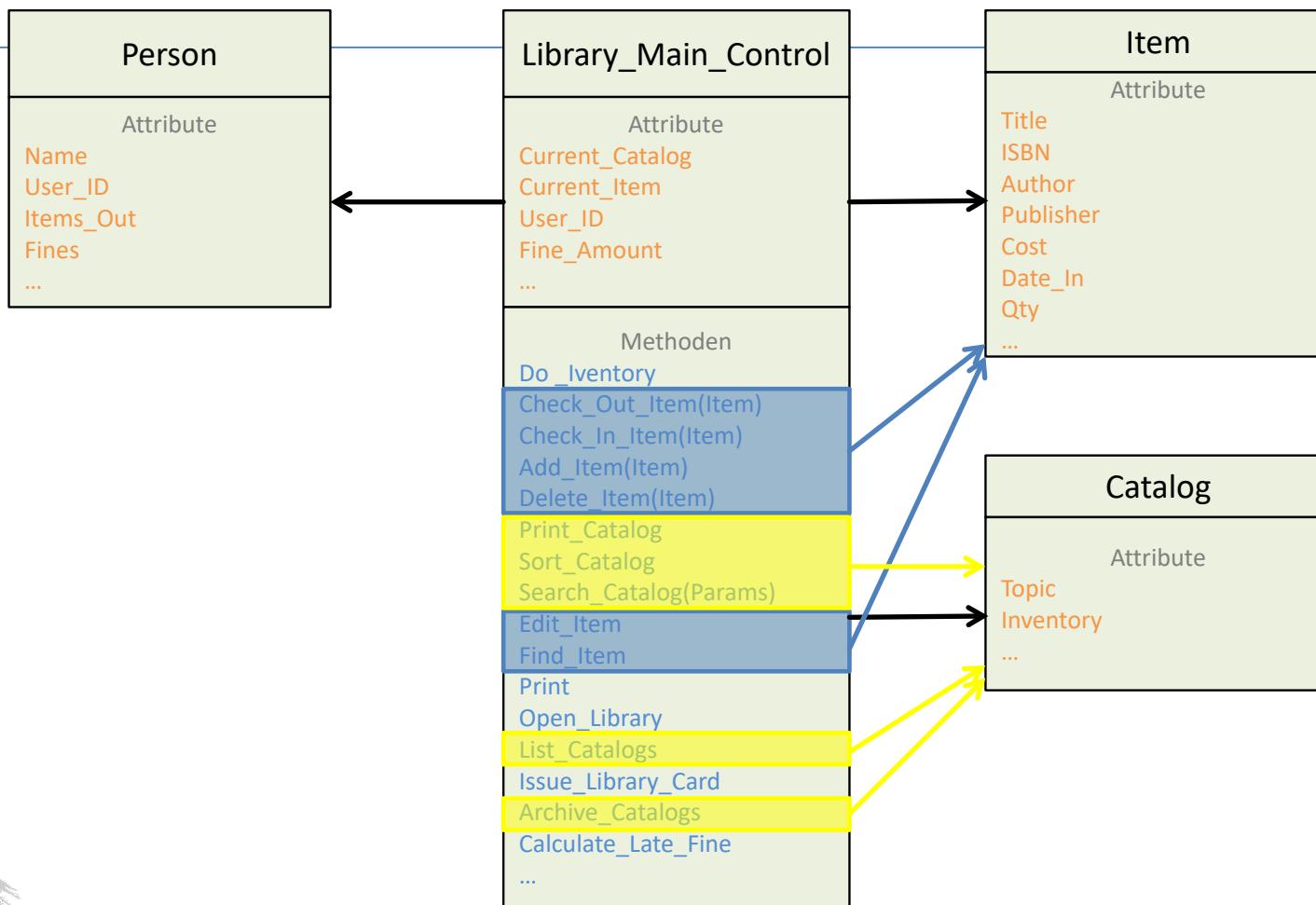
The Blob II



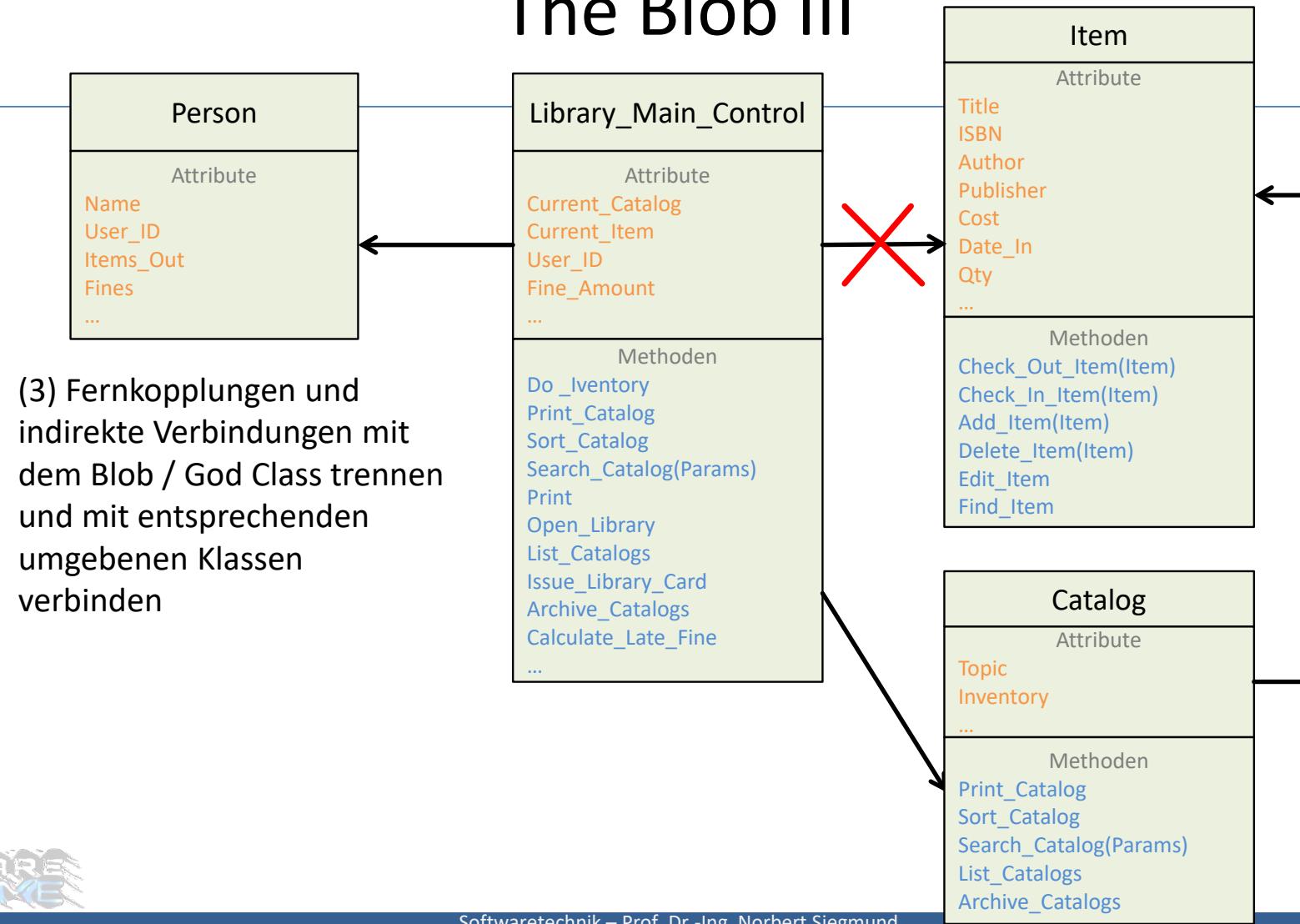
Aufgabe

- (1) Finde zusammenhängende Attribute und Methoden im Blop / God Class und gruppiere diese
- (2) Lagere die Gruppen in passende umgebene Klassen aus

The Blob III



The Blob III



Weitere Anti-Pattern

- Lava Flow (Dead Code)



```
// This class was written by someone earlier (Alex?) to manage the indexing  
// or something (maybe). It's probably important. Don't delete. I don't  
// think it's used anywhere - at least not in the new MacroIndexer module which  
// may actually replace whatever this was used for.  
class IndexFrame extends Frame  
{  
    // IndexFrame constructor  
    //-----  
    public IndexFrame(String index_parameter_1)  
    {  
        // Note: need to add additional stuff here.  
        super (str);  
    }  
    //-----
```

- Spaghetti Code

Literatur

AntiPatterns: The Survival Guide

Only available as online PDF:
<http://sourcemaking.com/antipatterns-book>

Find more in the web

<http://www.antipatterns.com/>



Softwaretechnik

Softwarearchitektur



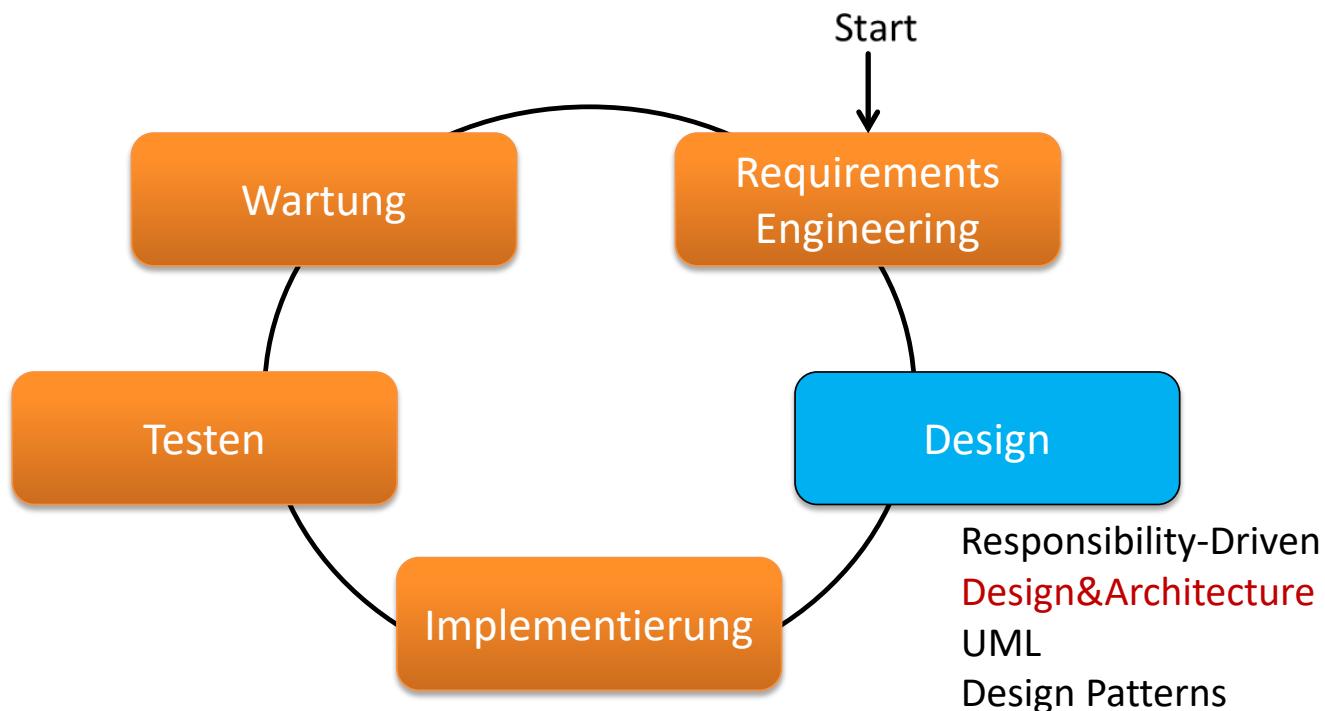
**SOFTWARE
SYSTEME**

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Einordnung



Lernziele

- Arten von Software-Architekturen kennen
- Zusammenhang zwischen Design und Architektur einordnen
- Konkrete Architekturmuster wissen

Was ist Software Architektur?



Was ist Software Architektur?

A neat-looking drawing of some boxes, circles, and lines, laid out nicely in Powerpoint or Word, does not constitute an architecture.

— D'Souza & Wills



Was ist (wirklich) Software Architektur?

Die Architektur eines Systems besteht aus:

- der *Struktur(en) ihrer Teile*
 - einschließlich Design-, Test und Laufzeit Hardware und Software Teile
- den *extern sichtbaren Eigenschaften* dieser Teile
 - Module mit Interfaces, Hardware-Einheiten und Objekte
- den *Beziehungen und Bedingungen* zwischen ihnen

In anderen Worten:

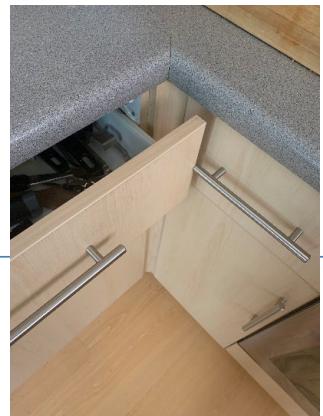
Eine Menge von Design-Entscheidungen über ein (Sub)System, die die Entwicklerinnen davon abhält sinnlos herumzuexperimentieren.

Design vs. Architektur

- Design beschreibt Aufbau von Subsystemen und Komponenten (fein granular)
 - Welche Klassen gibt es (in Modul X) und wie interagieren sie?
 - Siehe Responsibility-Driven Design / objektorientiertes Design
- Architektur beschreibt den groben Aufbau eines Systems (welche Komponenten gibt es?)
 - Welche Komponenten / Subsysteme / Module gibt es und wie interagieren sie?

Sub-systeme, Module und Komponenten

- Ein Sub-system ist selbst ein System, dessen Operation *unabhängig* von den Leistungen und Funktionen anderer Sub-systeme ist.
- Ein Modul ist eine Systemkomponente, die *Dienstleistungen / Funktionen anbietet*, welche andere Komponenten benötigen, aber welche nicht als komplett separates System angesehen werden.
- Eine Komponente ist eine *unabhängig auslieferbare Einheit* von Software, die ihr Design und Implementierung eingeschlossen hat (hiding) und ihr Interface zur Außenwelt anbietet, so dass sie mit anderen Komponenten zusammengefasst werden kann, um ein größeres System zu bilden.



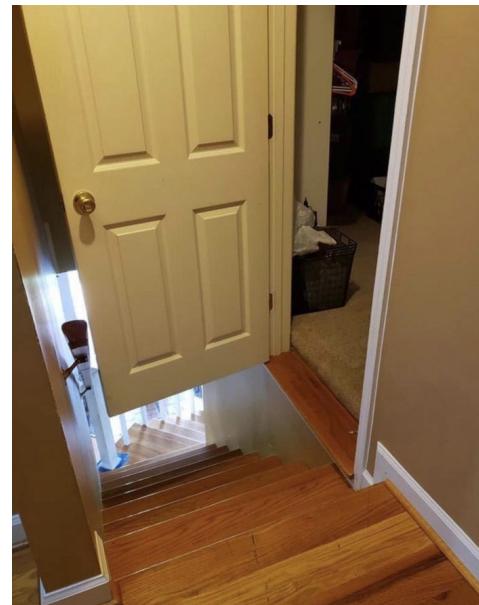
Zentralisierte Komponenten



Fehlende Abstimmung
zwischen Teams

Architekturprobleme

Fehlende gesamtheitliche
Modellierung



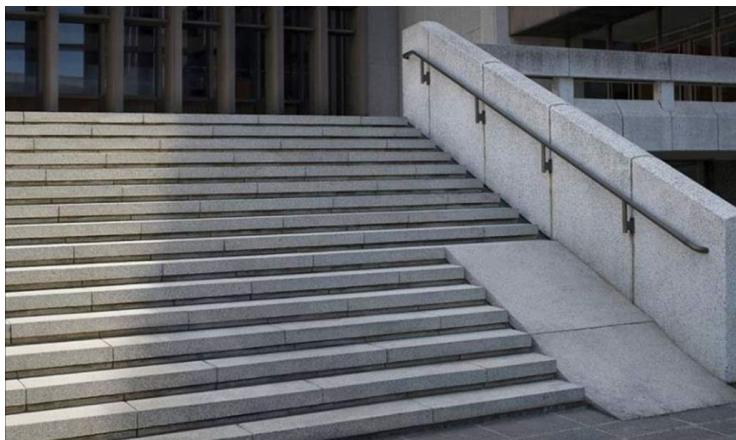
Workarounds für
schnelle Lösungen



Zu viele kleine Komponenten (zu
viel Kommunikation erforderlich)



Modul skaliert
nicht mit Rest des
Systems

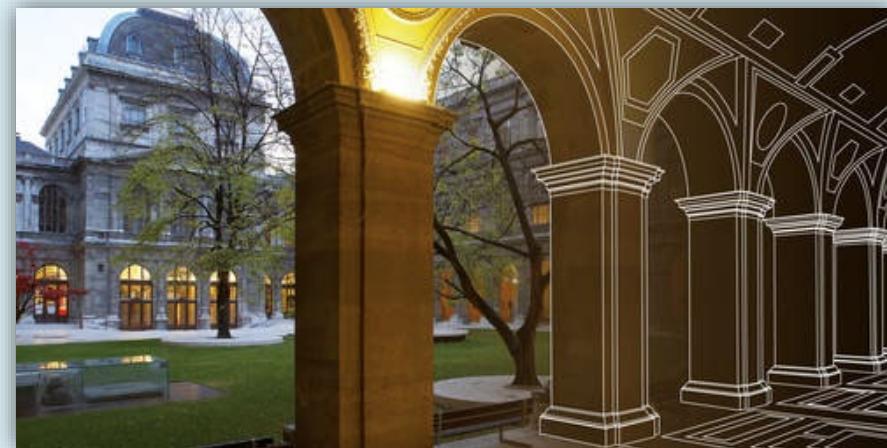


Interfaces passen nicht zusammen
(falsche Annahmen und fehlende Validierung)



Falsche / unklare Komposition-
bzw. Ausführungsreihenfolge

Arten von SW Architektur



Parallelen zur (echten) Architektur

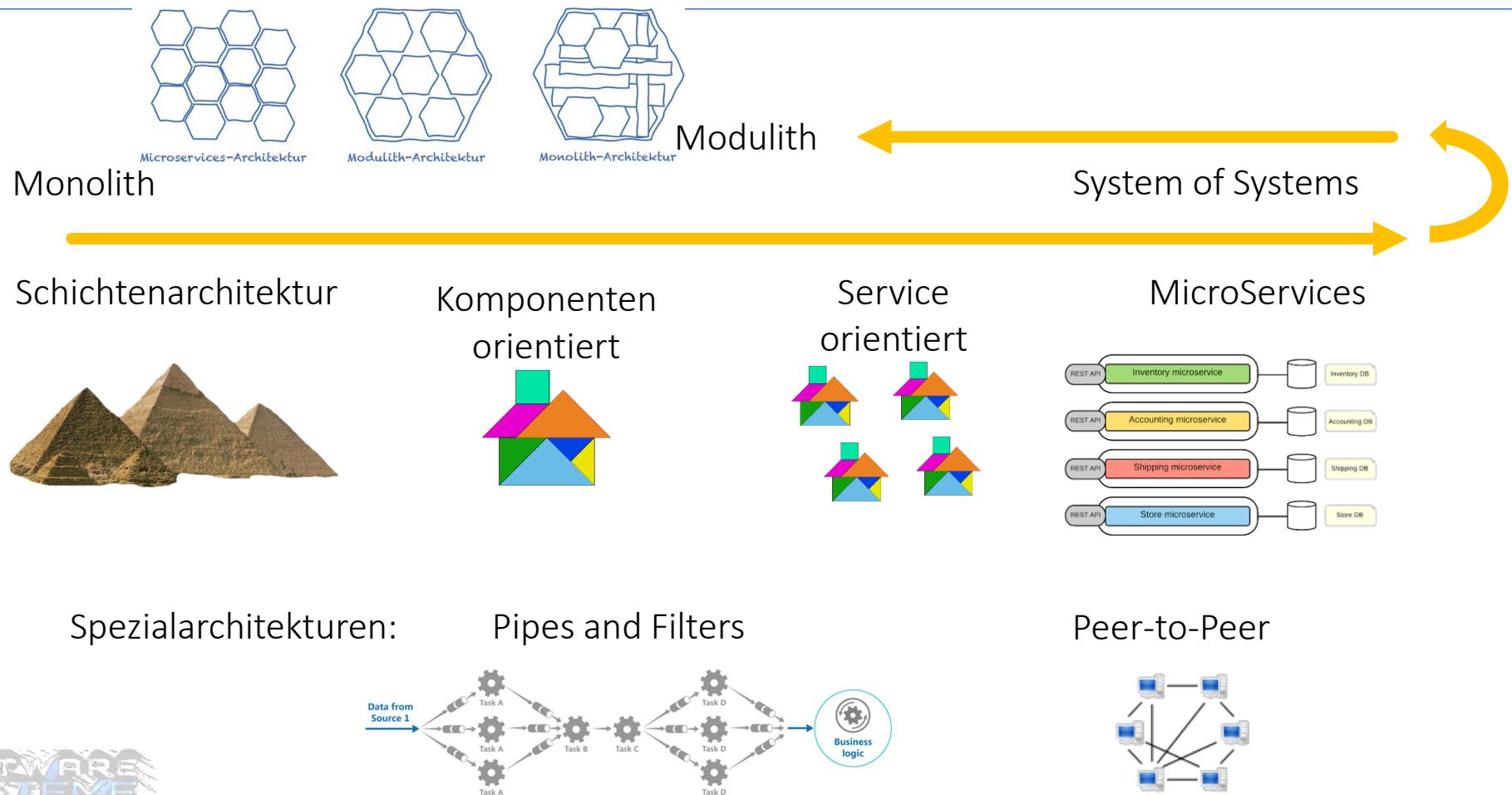
- Architekten sind die *Schnittstelle* zwischen den Kunden und den Auftragnehmern, die das System/Gebäude bauen
- Eine schlechte Architektur für ein Gebäude *kann nicht mehr durch gute Konstruktion gerettet werden* – gleiches gilt für Software
- Es gibt *spezielle Typen* von Gebäuden und Software-Architekten.
- Es gibt *Schulen oder Styles* des Bauens und der Software –Architektur.

Architectural Styles

An architectural style defines a *family of systems* in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of *components* and *connector* types, and a set of *constraints* on how they can be combined.

— Shaw and Garlan

SW-Architekturen für große Softwaresysteme

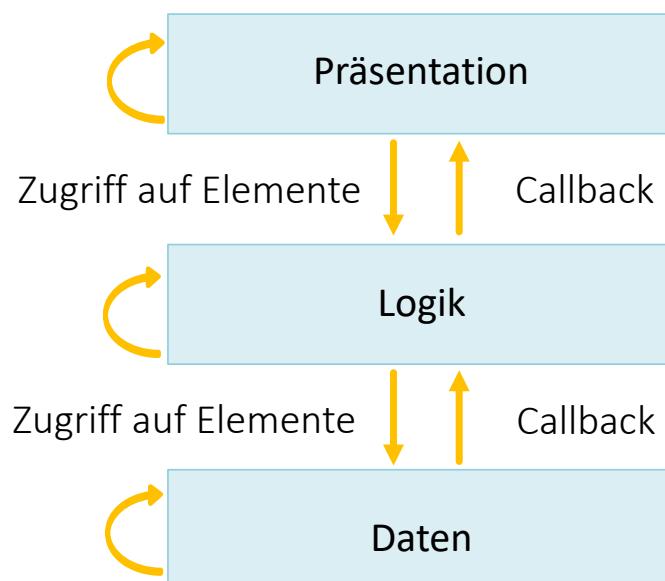


Welche Architektur?

The screenshot shows an Amazon product page for a "AmazonBasics Tasche für Laptop / Tablet mit Bildschirmdiagonale 15,6 Zoll / 39,6 cm". The page includes a navigation bar with links like "Koffer, Rucksäcke & Taschen", "Prime", "Über 2 Millionen Songs. Ohne Werbung.", and "Einkaufswagen". The main content features a large image of the black laptop bag, its dimensions (15,6 Zoll / 39,6 cm), and a price of EUR 19,49 Prime. It also displays customer reviews (4.5 stars from 616 reviews) and a "Bestseller Nr. 1". The page highlights "Auf Lager" (In stock) and offers free shipping ("KOSTENLOSER RÜCKVERSAND"). On the right, there's a sidebar for purchasing options, including "In den Einkaufswagen" and "Jetzt mit 1-Click® kaufen", along with delivery information ("Samstag, 21 Jan") and a "Lieferort" dropdown.

Schichtenarchitekturen

- Eine Schichtenarchitektur organisiert ein System in eine Menge von Schichten, wobei jede Schicht eine Menge von Leistungen / Funktionen für die Schicht “darüber” anbietet.



Vorteile:

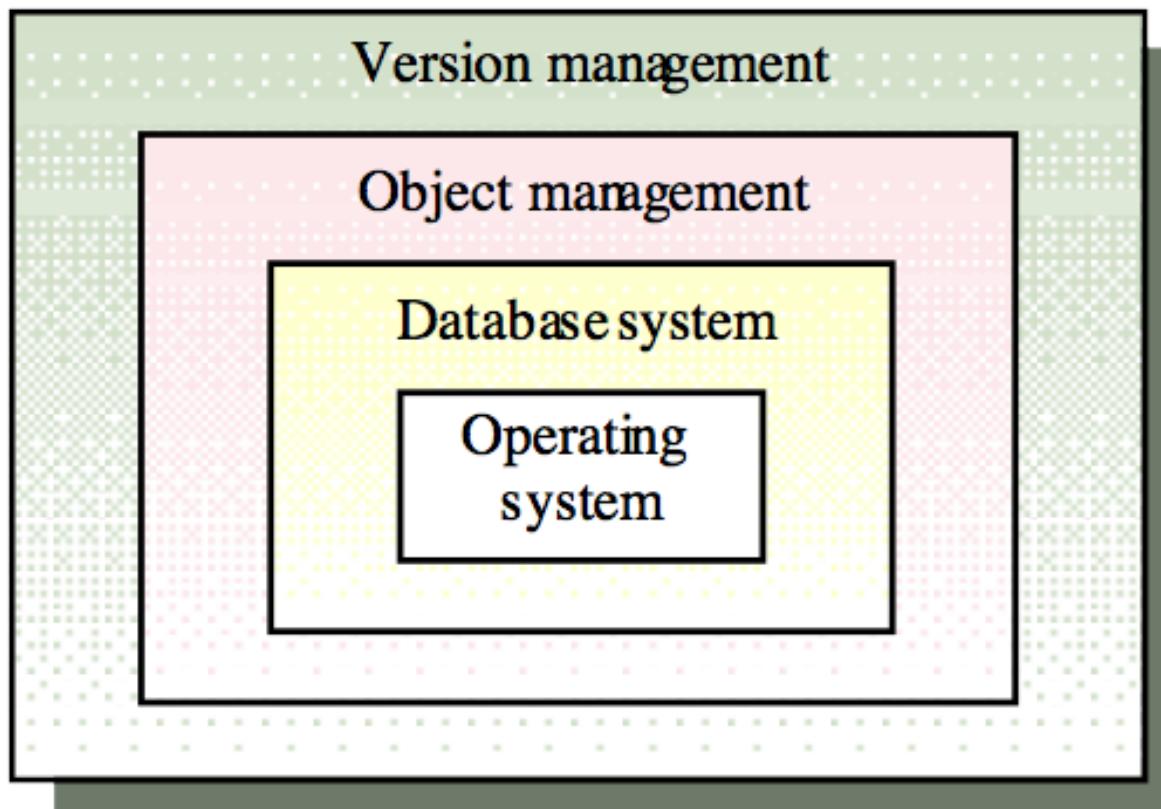
- Inkrementelle Entwicklung von Subsystemen
- Wenn ein Interface einer Schicht sich ändert, *sind nur benachbarte Schichten betroffen*
- Modularität, Kohäsion

Layered Architectures

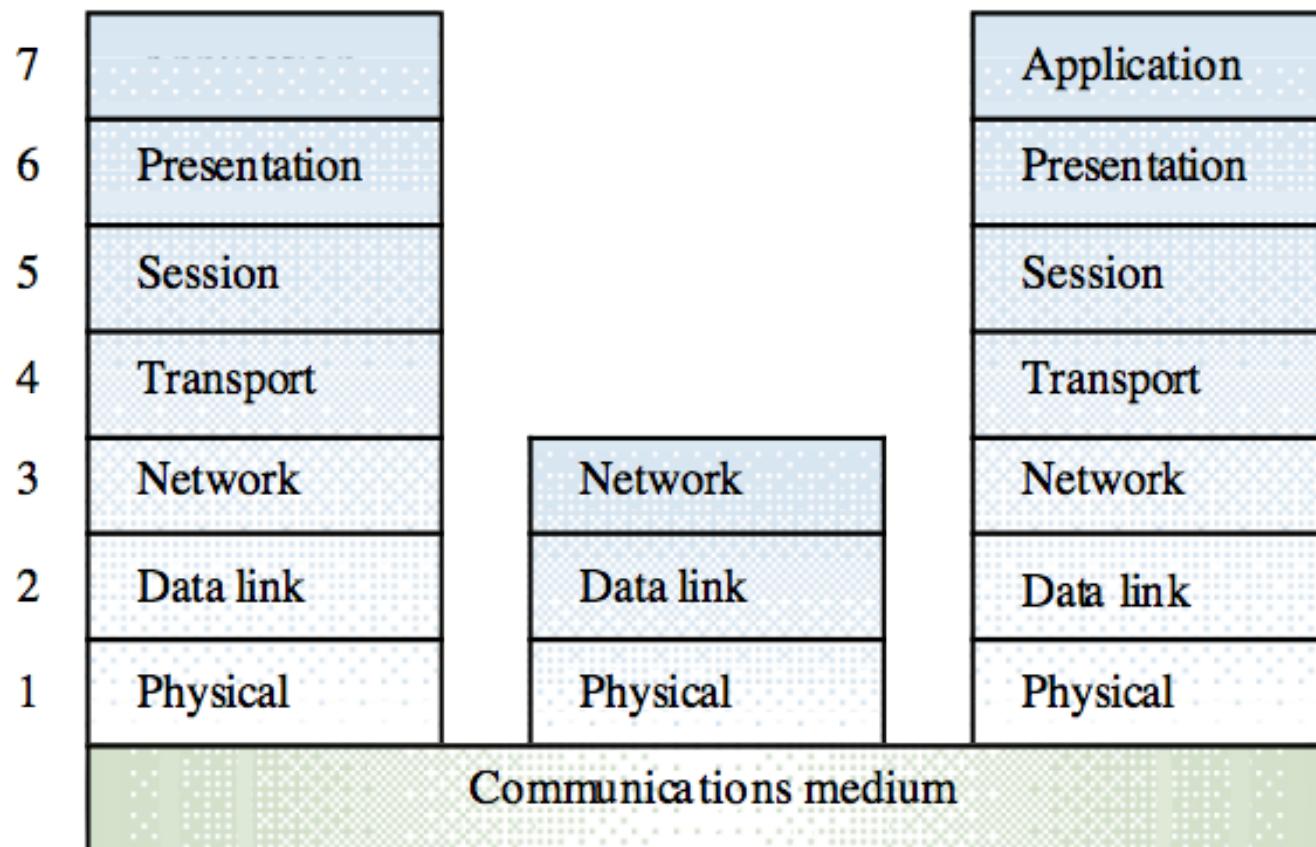
Eine Schichtenarchitektur organisiert ein System in eine Menge von Schichten, wobei jede Schicht eine Menge von Leistungen / Funktionen für die Schicht “darüber” anbietet.

- Schichten sind normalerweise *beschränkt*, so dass Elemente nur
 - Andere Element in der gleichen Schicht oder
 - Elemente von der Schicht darunter sehen können
- *Callbacks* können verwendet werden, um mit höheren Schichten zu kommunizieren
- Unterstützt die *inkrementelle Entwicklung* von Sub-systemen in unterschiedlichen Schichten
 - Wenn ein Interface einer Schicht sich ändert, *sind nur benachbarte Schichten betroffen*.

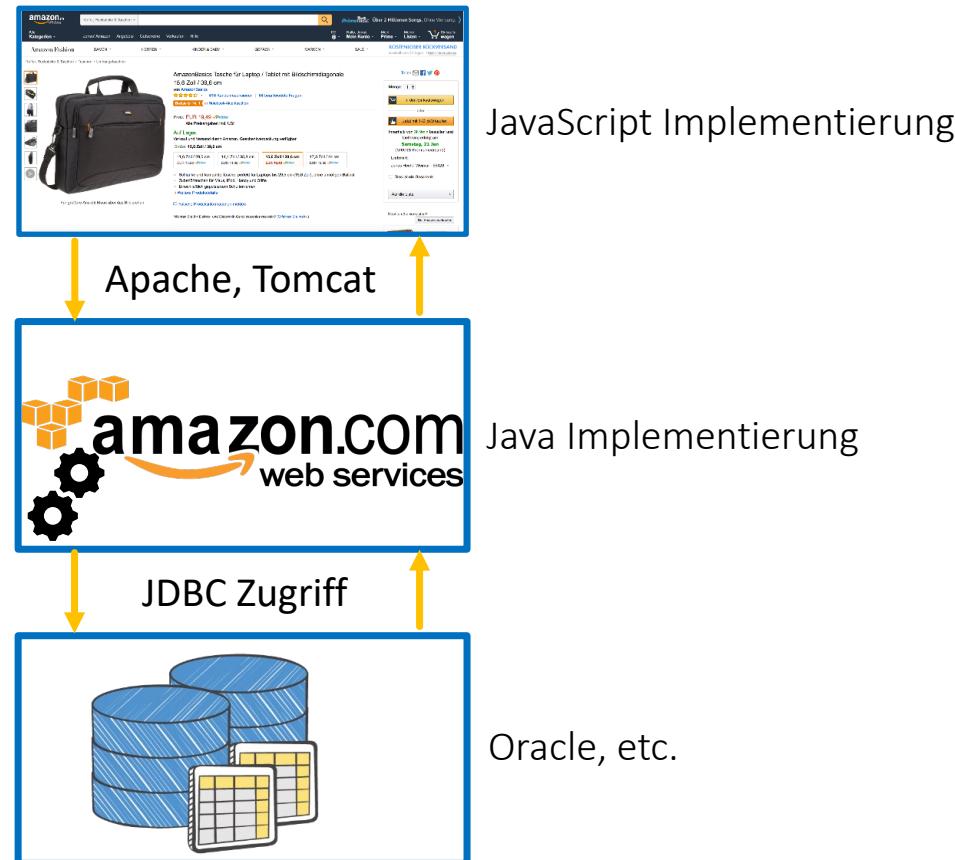
Version Management System



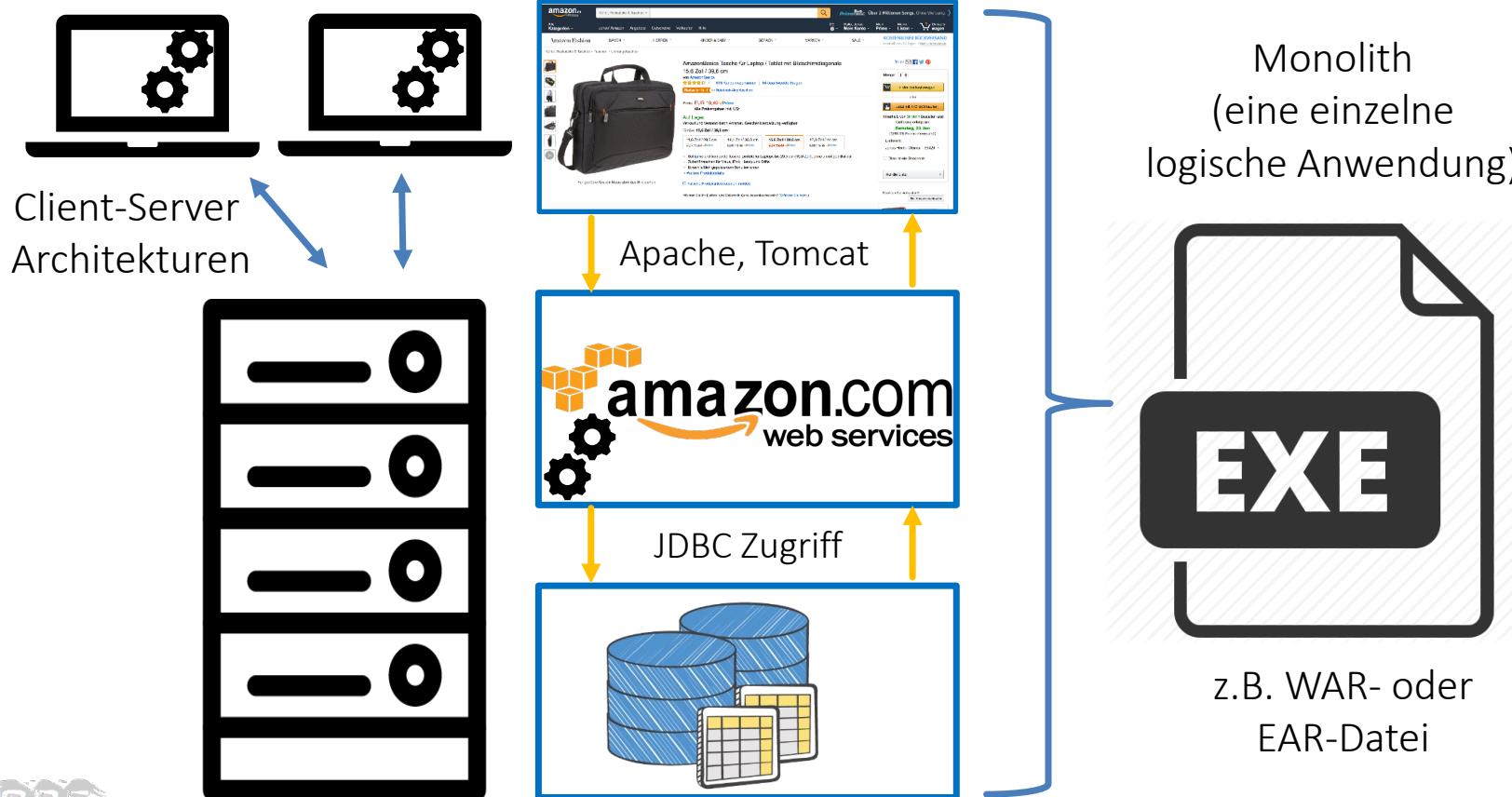
OSI Reference Model



Beispiel: 3-Layer Architektur



Client-Server & Monolith



Client-Server Architektur

Eine Client-Server Architektur *verteilt Applikationslogik und Funktionalität* zu einer Anzahl von Klienten (clients) und Server-Subsystemen, wobei jede potentiell auf einer unterschiedlichen Maschine läuft und über das Netzwerk kommuniziert.

Vorteile:

- Einfache *Datenverteilung*
- Effektive *Hardwareauslastung*
- Einfaches *Hinzufügen* neuer Server

Nachteile:

- Kein *geteiltes Datenmodell*
- *Redundante* Verwaltung
- Evtl. *zentrale Registrierung* erforderlich
(welcher Server stellt welche Dienstleistung zur Verfügung?)



Und ohne Web?

Wie würde Sie die Architektur entwerfen, wenn wir eine Desktop-Anwendung schreiben würden?

Kommunikation / Controller

Apache, Tomcat, JavaScript

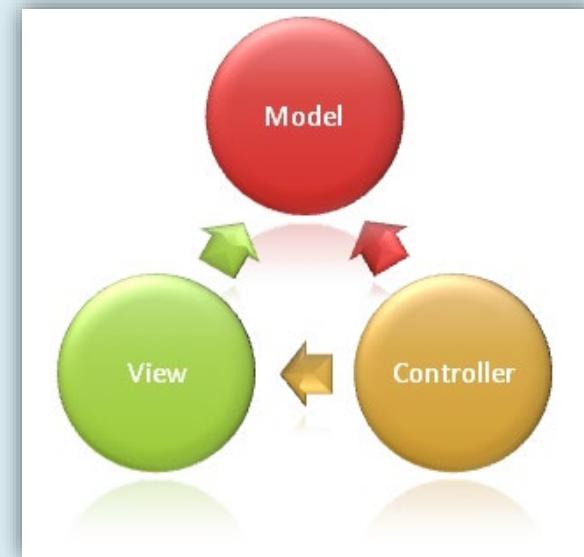
Sicht / View

JavaScript Frontend

Java Implementierung, DB, ...

Model / Business-Logik / Daten

Model-View-Controller

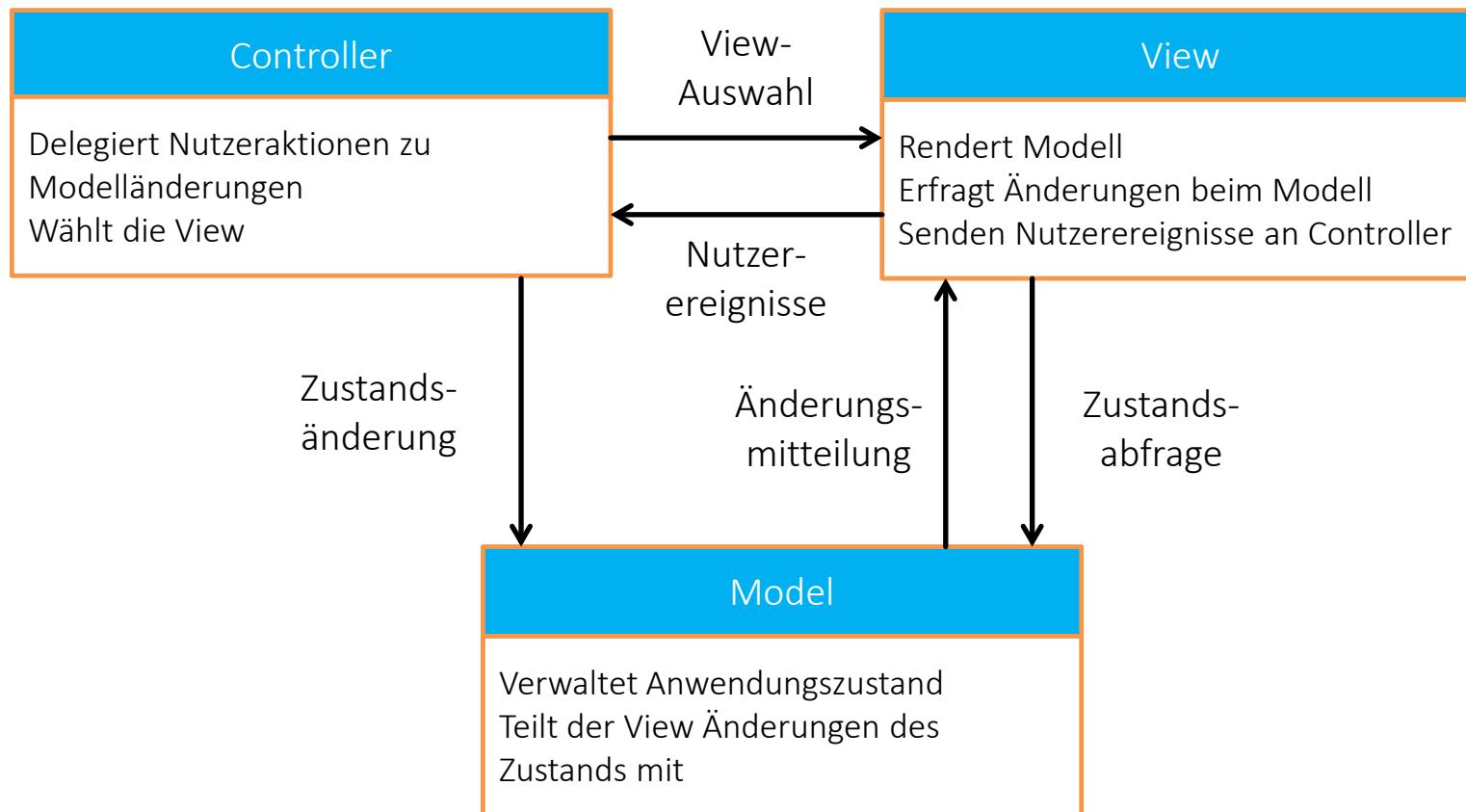


Model-View-Controller (MVC) Architektur

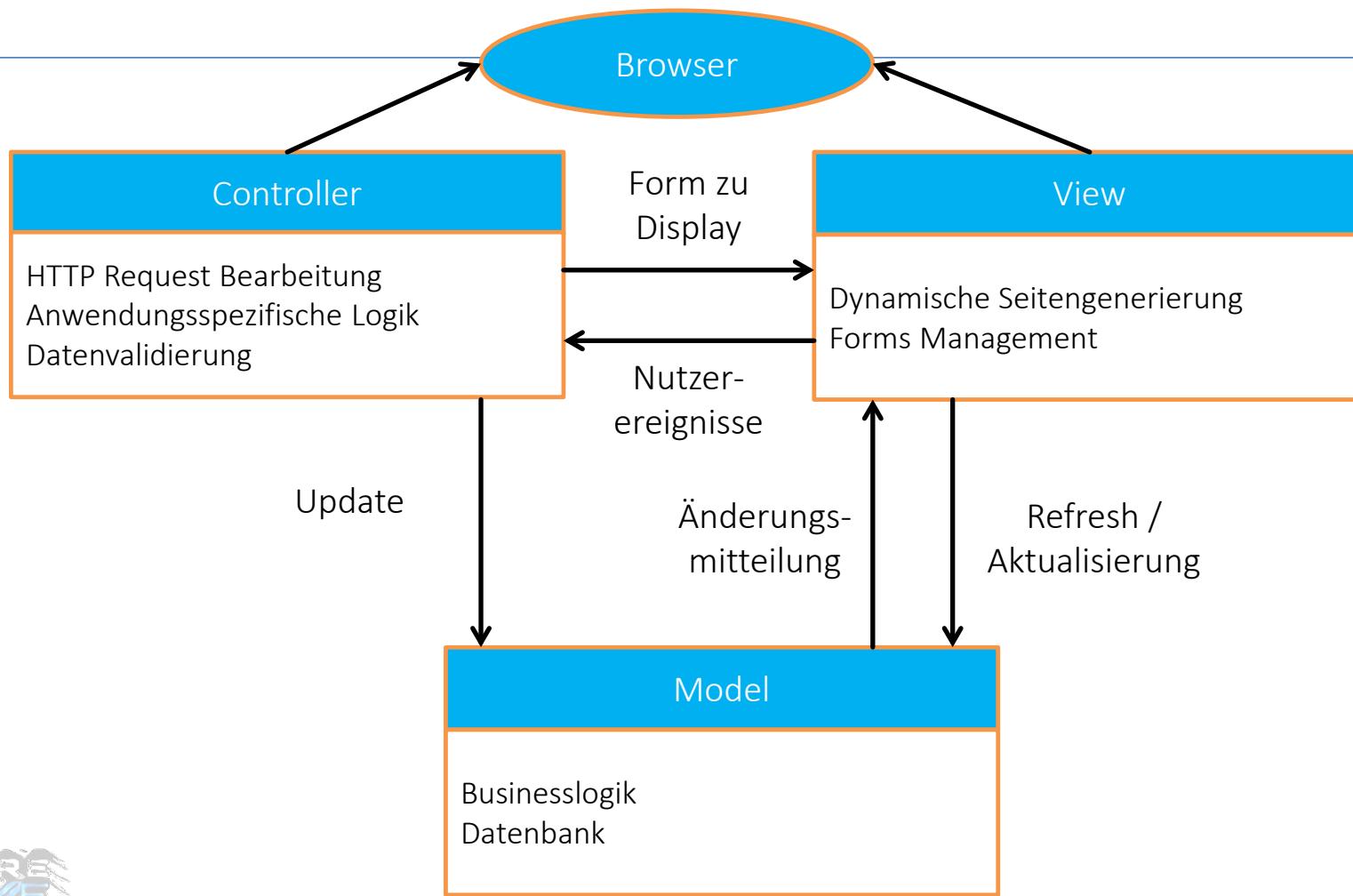
Idee: Sepriere *Präsentation* und *Interaktion* von den *Daten* des Systems

- Das System ist strukturiert in drei Komponenten:
 - *Model*: verwaltet Systemdaten und Operationen auf den Daten
 - *View*: Präsentiert die Daten zum Nutzer
 - *Controller*: händelt Nutzerinteraktion; schickt Informationen zur View und zum Model
- Nützlich, wenn es mehrere Wege gibt auf Daten zuzugreifen
- Ermöglicht das Ändern der Daten unabhängig von deren Repräsentation
- Unterstützt unterschiedliche Präsentationen der selben Daten

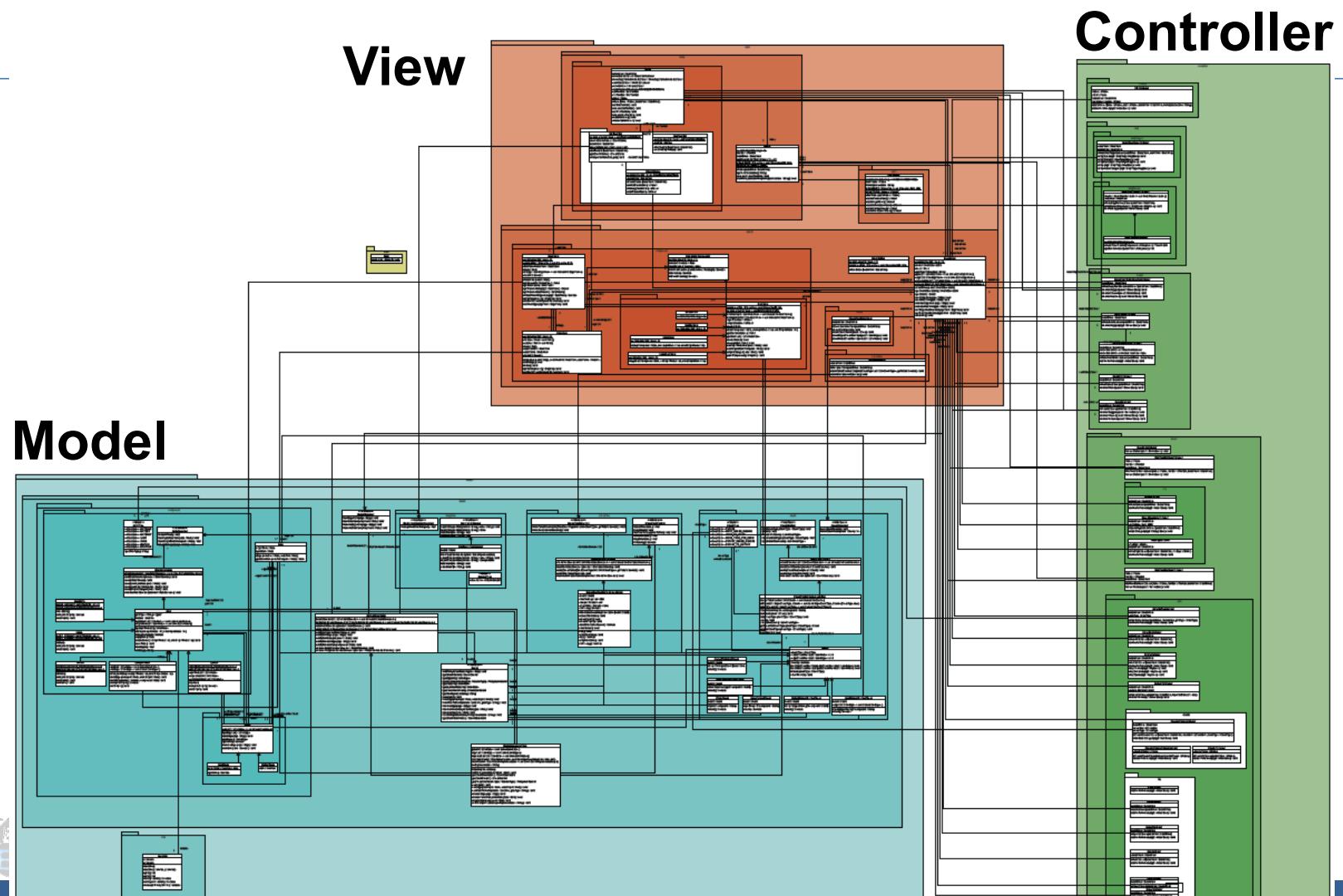
MVC Übersicht



MVC Beispiel



MVC in Action (Circuit Simulation)



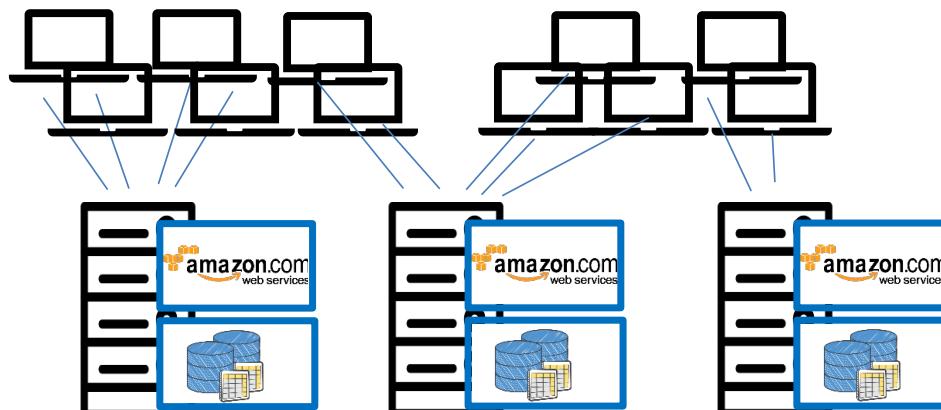
Probleme der Architekturen

- Welche Probleme erwarten Sie, wenn wir diese Architekturen in der Praxis für unser Beispiel einsetzen?
- Hinweise:
 - Großes Softwaresystem
 - Hohes Nutzeraufkommen



Probleme traditioneller Architekturmuster für große Softwaresysteme

- Häufige Änderungen
 - Monolithisches System muss komplett neu gebaut werden
 - Abhängigkeiten zwischen Subsystemen erschweren und verzögern Änderungen
- Skalierbarkeit der Hardware
 - Alle Architekturen sind schwer skalierbar, selbst Client-Server



Neue Probleme:

- Verteilte, *replizierte* Daten
- *Konsistenz* und Synchronität?
- *Gesamtes* System repliziert, obwohl nur Subsysteme ausgelastet sind

Probleme traditioneller Architekturmuster für große Softwaresysteme

- Was passiert bei Ausfällen und Fehlern von Subsystemen?
 - Oft zu starke Kopplung der Subsysteme und kaum Möglichkeit des „Fail-overs“
 - Alternative Views bei MVC möglich, aber alternative Modelle?
- Wie wird das laufende System geupdatet?
 - Herunterfahren der Server ist keine Option
- Wie kann die Entwicklung von Subsystemen parallelisiert werden?
 - Schichtenarchitektur und MVC oft zu grob-granular
 - Komponenten und Services nicht gut bei querschneidenden Belangen und erfordern zu viel Glue-Code / Kommunikation
- Wie vereinfache ich das Testen? Usw.

Gewünschte Eigenschaften

- Schneller Austausch von Subsystemen ohne, dass das gesamte System betroffen ist (lose Kopplung + hohe Modularität)
- Skalierbarkeit bei großen Lasten von einzelnen Subsystemen (Beispiel: Black Friday)
- Weniger Abstimmungsaufwand innerhalb der Unternehmensorganisation (bei Entwicklerinnen und Sachverständigen)
- Parallelisierung in der Entwicklung sowie Anwendbarkeit von agilen Methoden der Softwareentwicklung
- Einfache Testbarkeit von Subsystemen

Welche Architektur?

The screenshot shows an Amazon product page for a "AmazonBasics Tasche für Laptop / Tablet mit Bildschirmdiagonale 15,6 Zoll / 39,6 cm". The page includes a sidebar with categories like "Amazon Fashion", "DAMEN", "HERREN", etc., and a search bar at the top. A green circle highlights the main product image of a black laptop bag. A blue box highlights the product title and details. A red box highlights the "Teilen" (Share) and "Bestellen" (Buy) sections on the right.

amazon.de Prime

Koffer, Rucksäcke & Taschen

Alle Kategorien Jonas' Amazon Angebote Gutscheine Verkaufen Hilfe DE Hallo, Jonas Mein Konto Mein Prime Meine Listen Einkaufswagen

Über 2 Millionen Songs. Ohne Werbung.

Amazon Fashion DAMEN HERREN KINDER & BABY GEPÄCK MARKEN SALE

KOSTENLOSER RÜCKVERSAND Innerhalb von 30 Tagen > Mehr Informationen

Koffer, Rucksäcke & Taschen > Taschen > Umhängetaschen

AmazonBasics Tasche für Laptop / Tablet mit Bildschirmdiagonale 15,6 Zoll / 39,6 cm von AmazonBasics

5★ 616 Kundenrezensionen | 64 beantwortete Fragen

Bestseller Nr. 1 in Notebook-Akketaschen

Preis: EUR 19,49 ✓Prime Alle Preisangaben inkl. USt.

Auf Lager. Verkauf und Versand durch Amazon. Geschenkverpackung verfügbar.

Größe: 15,6 Zoll / 39,6 cm

11,6 Zoll / 29,5 cm EUR 13,99 ✓Prime 14,1 Zoll / 35,8 cm EUR 14,99 ✓Prime 15,6 Zoll / 39,6 cm EUR 19,49 ✓Prime 17,3 Zoll / 44 cm EUR 19,99 ✓Prime

- Schlanke und kompakte Tasche, perfekt für Laptops bis 29,5 cm (15,6 Zoll), ohne unnötigen Ballast
- Zubehörtaschen für Maus, iPod, Handy und Stifte
- Einschließlich gepolstertem Schulterriemen

Weitere Produktdetails

Falsche Produktinformationen melden

Möchten Sie Ihr Elektro- und Elektronik-Gerät kostenlos recyceln? (Erfahren Sie mehr.)

Teilen Menge: 1 In den Einkaufswagen oder Jetzt mit 1-Click® kaufen

Innerhalb von 3h 9min bestellen und Lieferung erfolgt am: Samstag, 21 Jan (GRATIS Premiumversand)

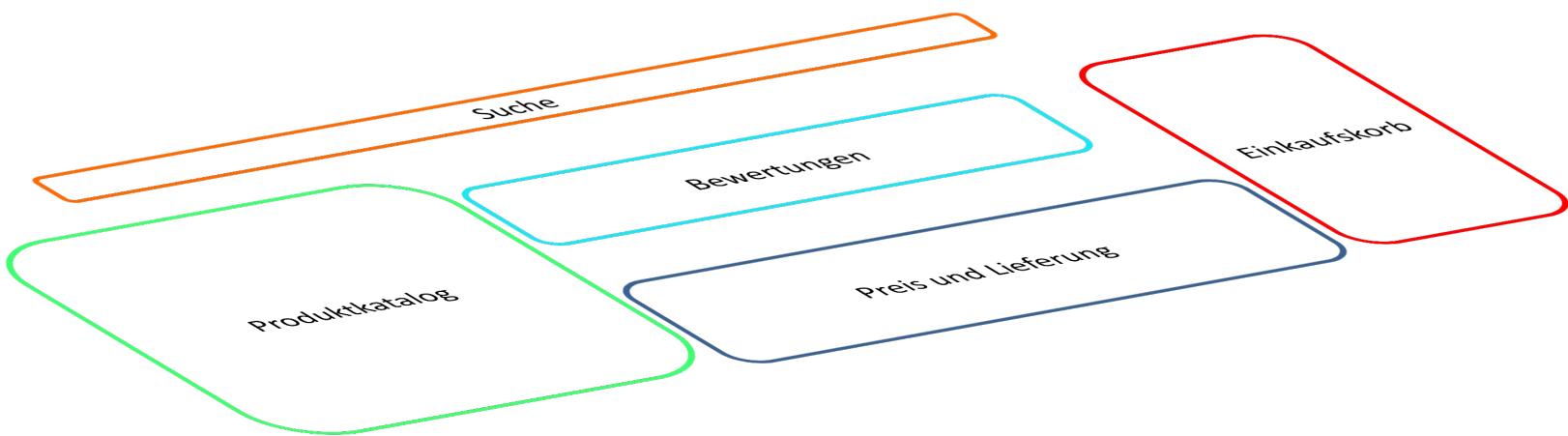
Lieferort: Jonas Hecht- Weimar - 99423 Dies ist ein Geschenk Auf die Liste

Möchten Sie verkaufen? Bei Amazon verkaufen

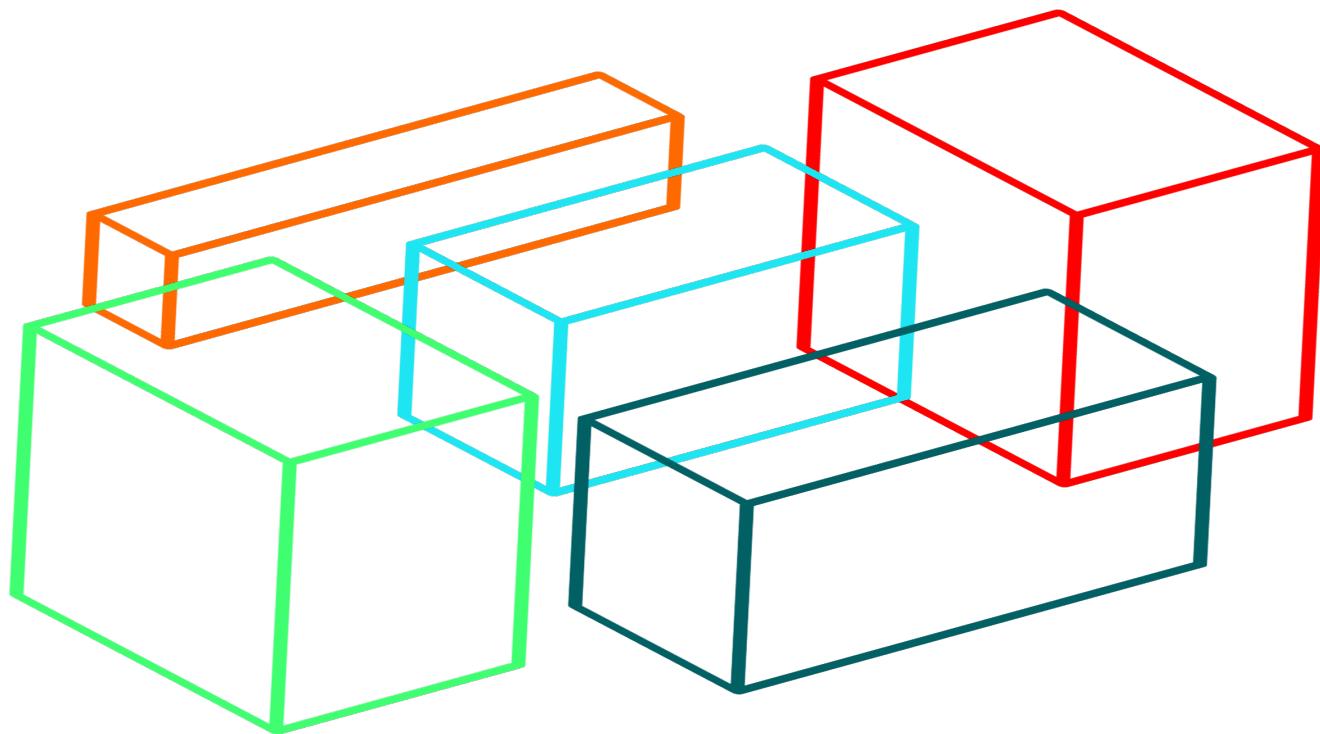
Zerlegung des Systems...



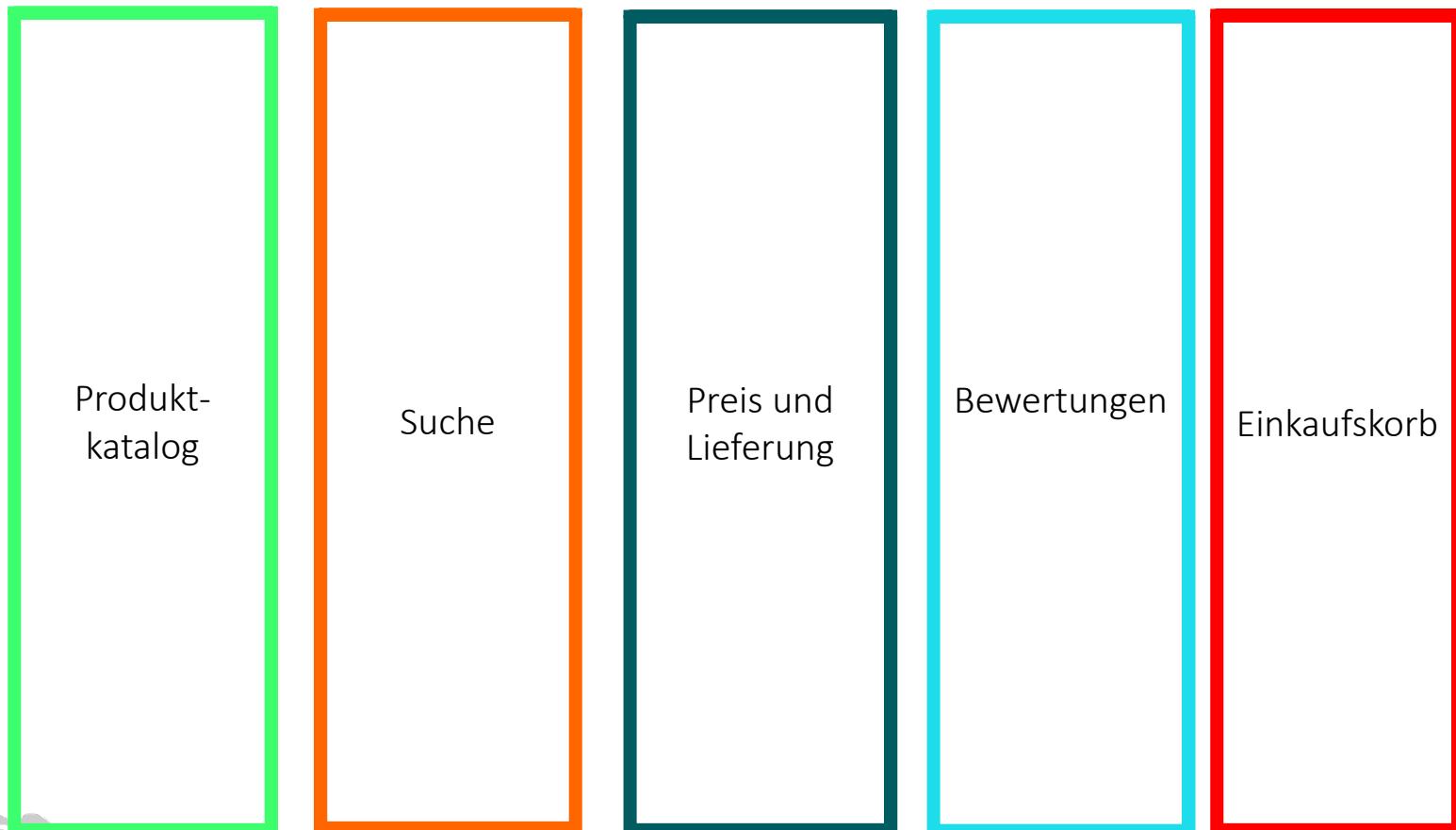
Zerlegung des Systems...



Zerlegung des Systems...



... nach fachlichen Funktionen





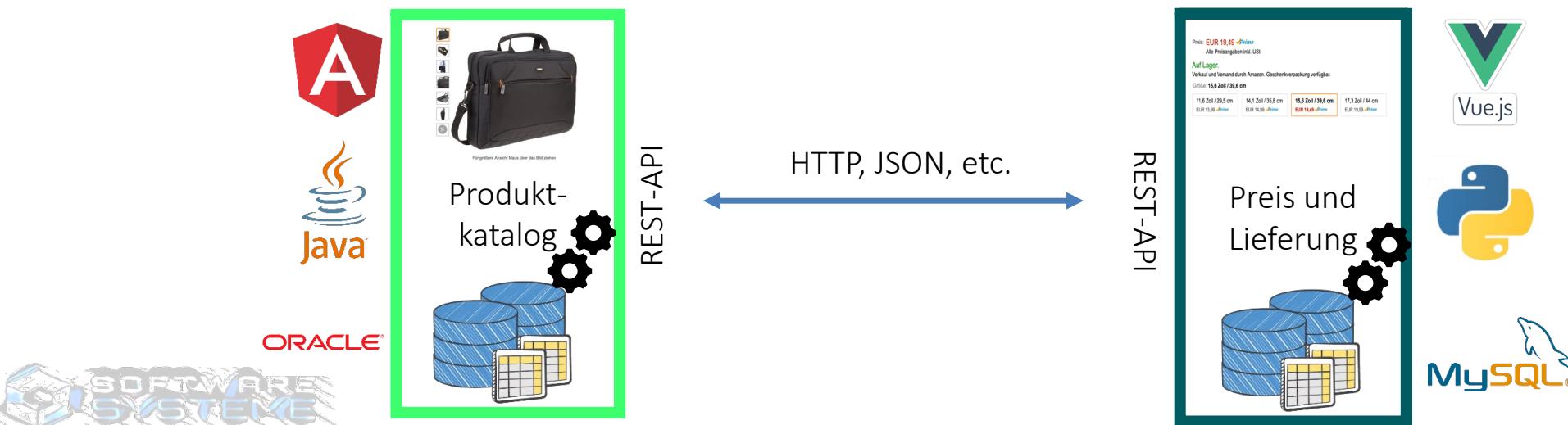
MicroServices: Idee

- Jede *fachliche* Funktion in einen *autonomen, unabhängigen Subsystem* modularisieren
- Jeder Service bietet die *vollständige* Funktionalität für die jeweilige fachliche Aufgabe an
 - Alle *Daten*, die hierfür notwendig sind
 - Gesamte *Business-Logik* für die Aufgabe
 - Alle *Sichten* und *Interaktionsmöglichkeiten*
- Teamzusammensetzung ideal für agile Entwicklung
 - Fachexpertinnen, Entwicklerinnen, Tester, DevOps

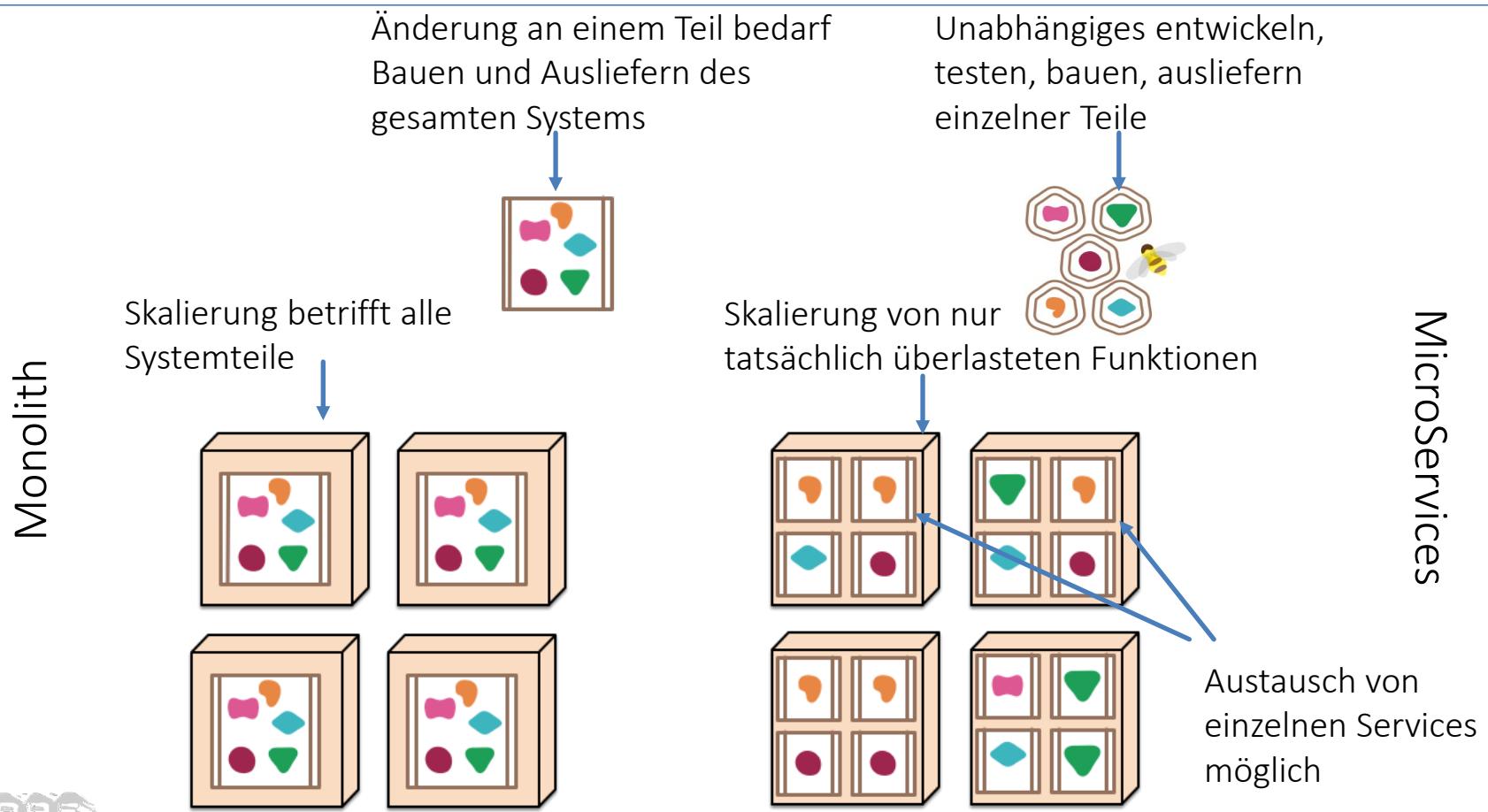
Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. -- Melvyn Conway, 1967

Von MicroService zum Softwaresystem

- Anwendung besteht aus einer Menge an MicroServices
 - Jeder hat eigene Prozesse und Daten
 - Leichtgewichtige Kommunikation (meist REST-API)
 - Unabhängig voneinander auslieferbar, testbar, entwickelbar
- Maximale Unabhängigkeit der MicroServices

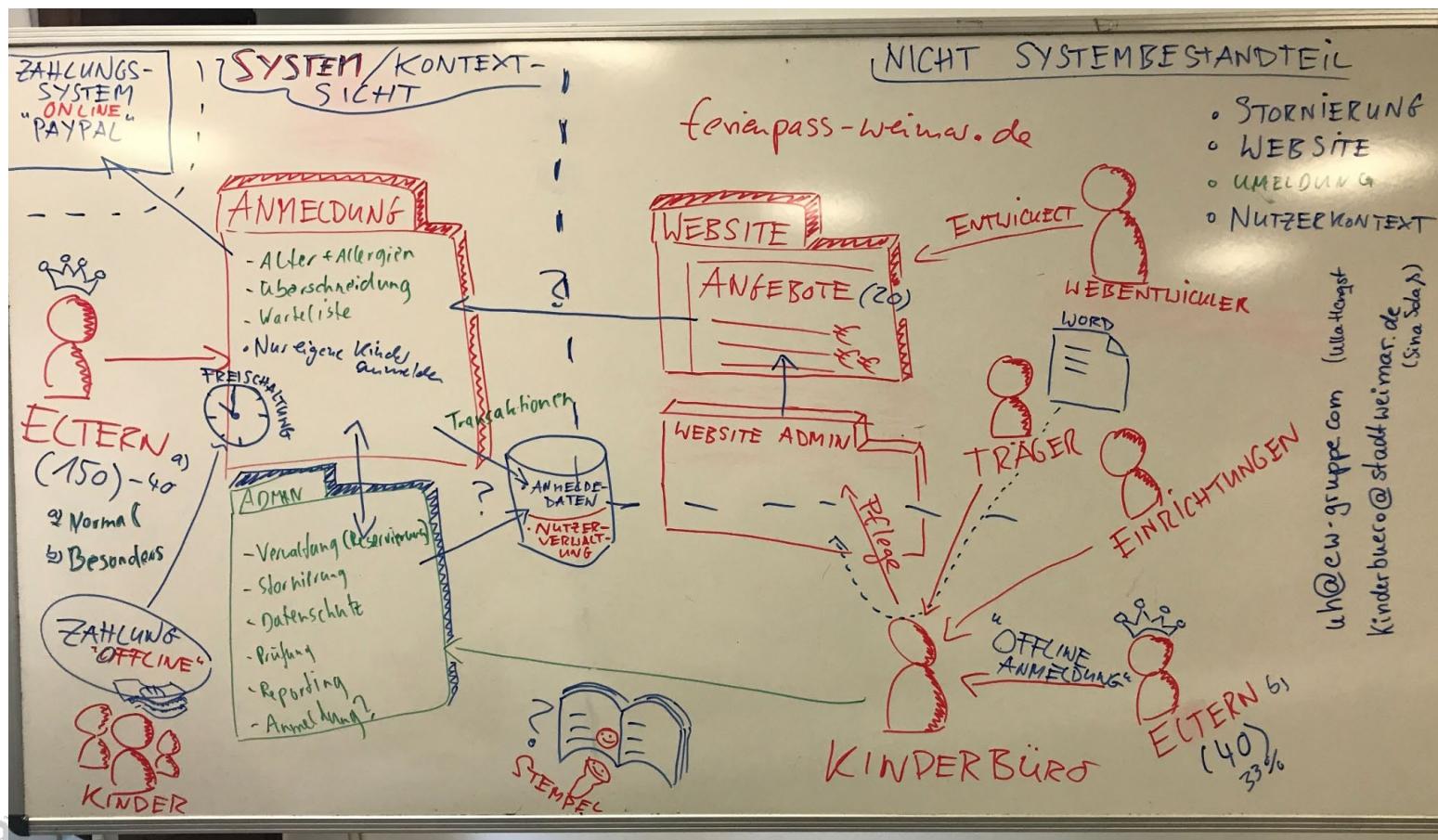


Monolith vs. MicroServices



by James Lewis & Martin Fowler

Erfahrungen Ihrer VorgängerInnen



Und in der Praxis?



NETFLIX



UBER



ebay

Kritische Diskussion

- Was sind mögliche Nachteile einer MicroService Architektur?
- Welche Besonderheiten und Voraussetzungen sollten beim Einsatz dieser Architektur betrachtet werden?
 - MicroServices für Word oder Virenscanner?

Nachteile von MicroServices

- Benötigt richtigen „Mindset“ der Entwicklerinnen und klare Definition von eines MicroServices (z.B. Mittels Domain-Driven Design Entwurfsmethodik)
- Immer noch hoher Kommunikationsaufwand zwischen Teams
- Moderne DevOps Pipeline erforderlich, Stichwort: Continuous Integration and Delivery
- Technologien entwickeln sich schnell weiter
- Architektur resultiert in ein verteiltes System mit all seinen Vor- und Nachteilen

Kriterien zur Auswahl von Architekturnmustern

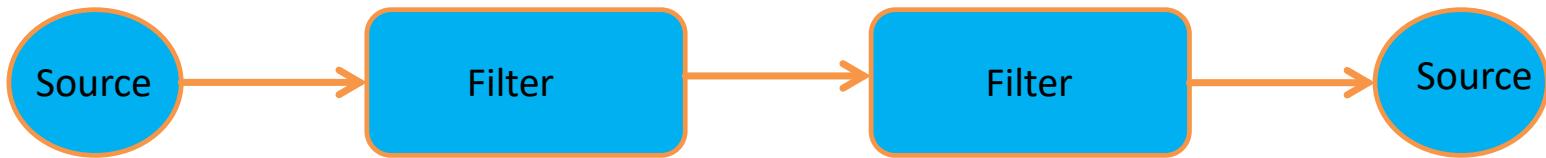
- Welche Struktur des Softwaresystems?
 - Komponenten-orientiert, monolithisch, Schichten, Pipes and Filters
- Wie kommunizieren die Sub-Systeme?
 - Ereignis-basiert, Publish-Subscribe , asynchrone Nachrichten
- Wie sind die Sub-Systeme verteilt?
 - Client-Server, shared nothing, Peer2Peer, service-orientiert, Cloud

Pipes and Filters



Aufbau

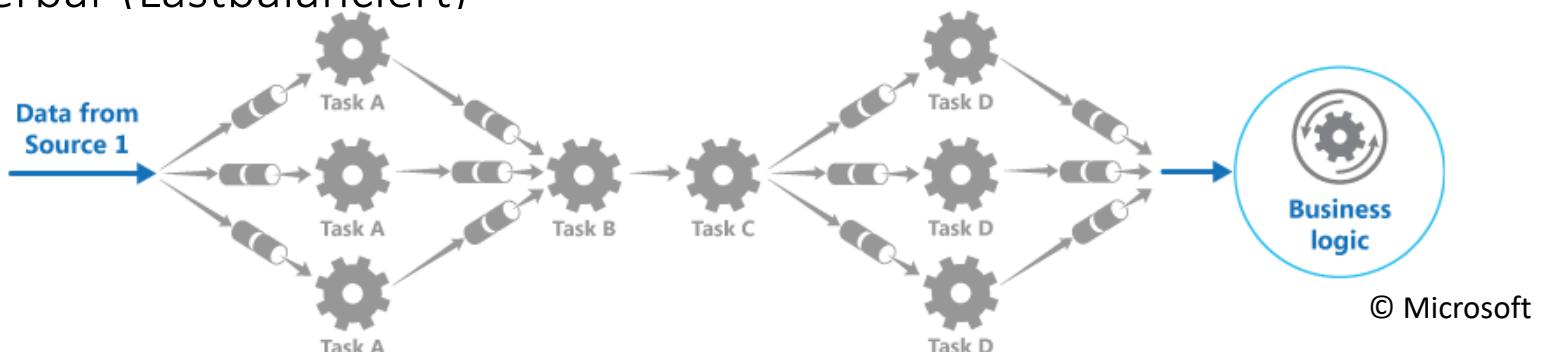
- Pipe: Verbindungsglied, welches Daten von einem Filter zu einem anderen weiterleitet
- Filter: Transformiert Daten, die es durch eine Pipe bekommen hat



- Vorteile:
 - Filter können einfach hinzugefügt und herausgenommen werden
 - Robust, performant, skalierbar, gute Wartbarkeit!

Vor- und Nachteile

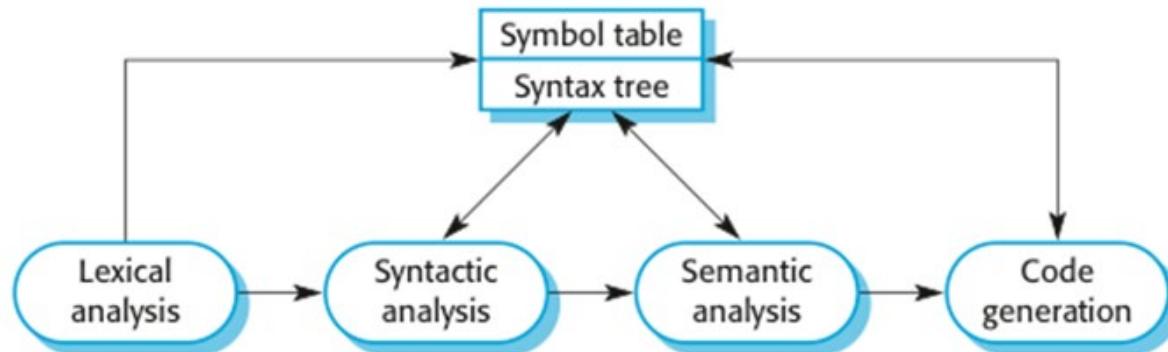
- Parallelisierbar (Lastbalanciert)



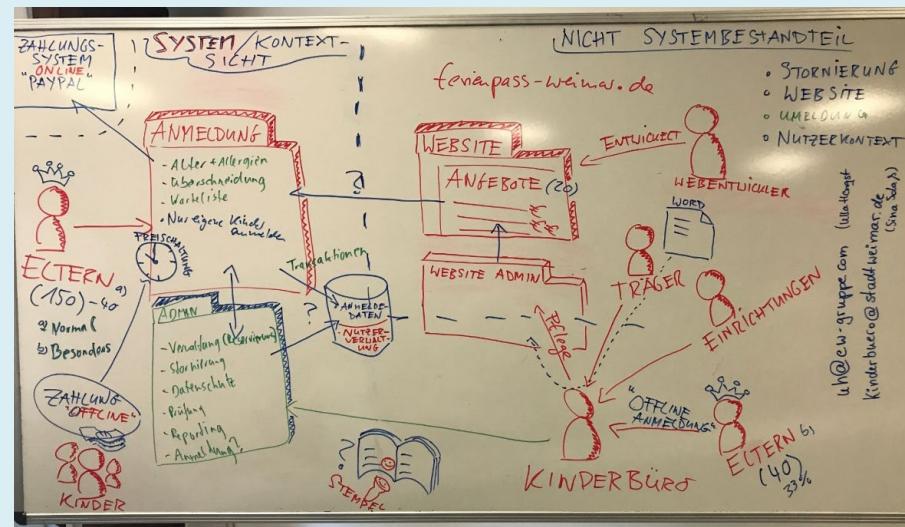
- Probleme:
 - Komplexität steigt
 - Bufferoverflow möglich
- Wann anwendbar?
 - Anwendung kann in mehrere unabhängige Teile zerlegt werden
 - Wenn viele Transformationen auf Daten nötig sind
 - Wenn Flexibilität notwendig ist (z.B. in der Cloud)

Beispiele

Domain	Data source	Filter	Data sink
Unix	<code>tar cf - .</code>	<code>gzip -9</code>	<code>rsh picasso dd</code>
CGI	HTML Form	CGI Script	generated HTML page

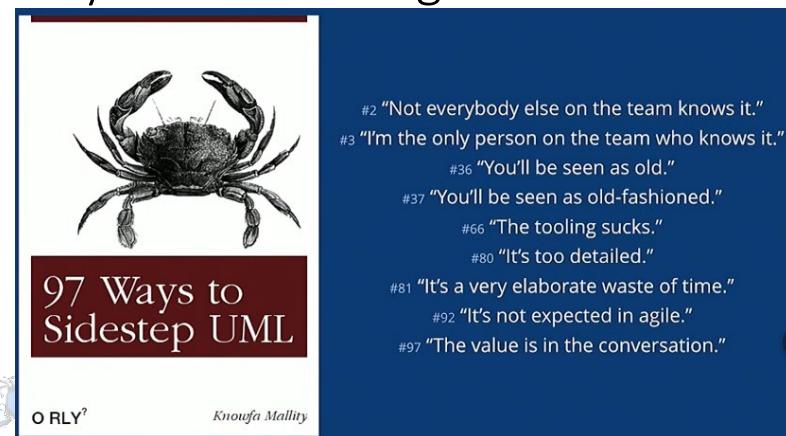


Modellierung von Softwarearchitektur

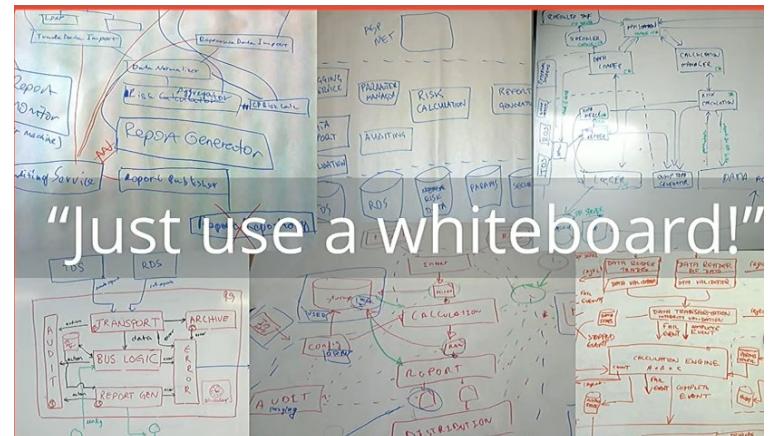


Unterschiedliche Sichten auf Softwarearchitektur

- Ziel: Modellierung des Verhaltens, der Struktur, der Logik und der Infrastruktur eines Systems, so dass alle Anforderungen erfüllt und keine Bedingungen verletzt werden.
- Architektur überbrückt Anforderungen zur Implementierung
 - Aber: Wie machen?
- Verwende eine sinnvolle Modellierungssprache für die unterschiedlichen Sichten auf das System und integriere sie zu einem gesamtheitlichen Bild

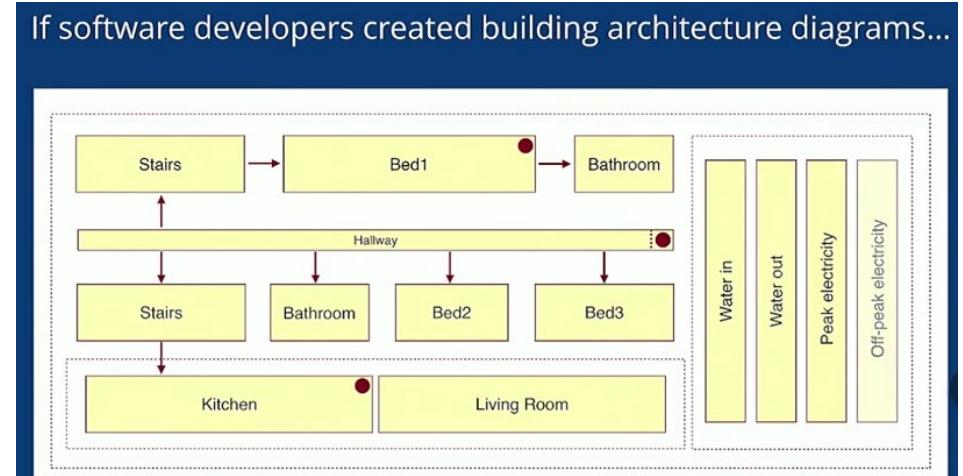


Beide Extreme funktionieren nicht allgemein in der Praxis.
Aber was dann machen?



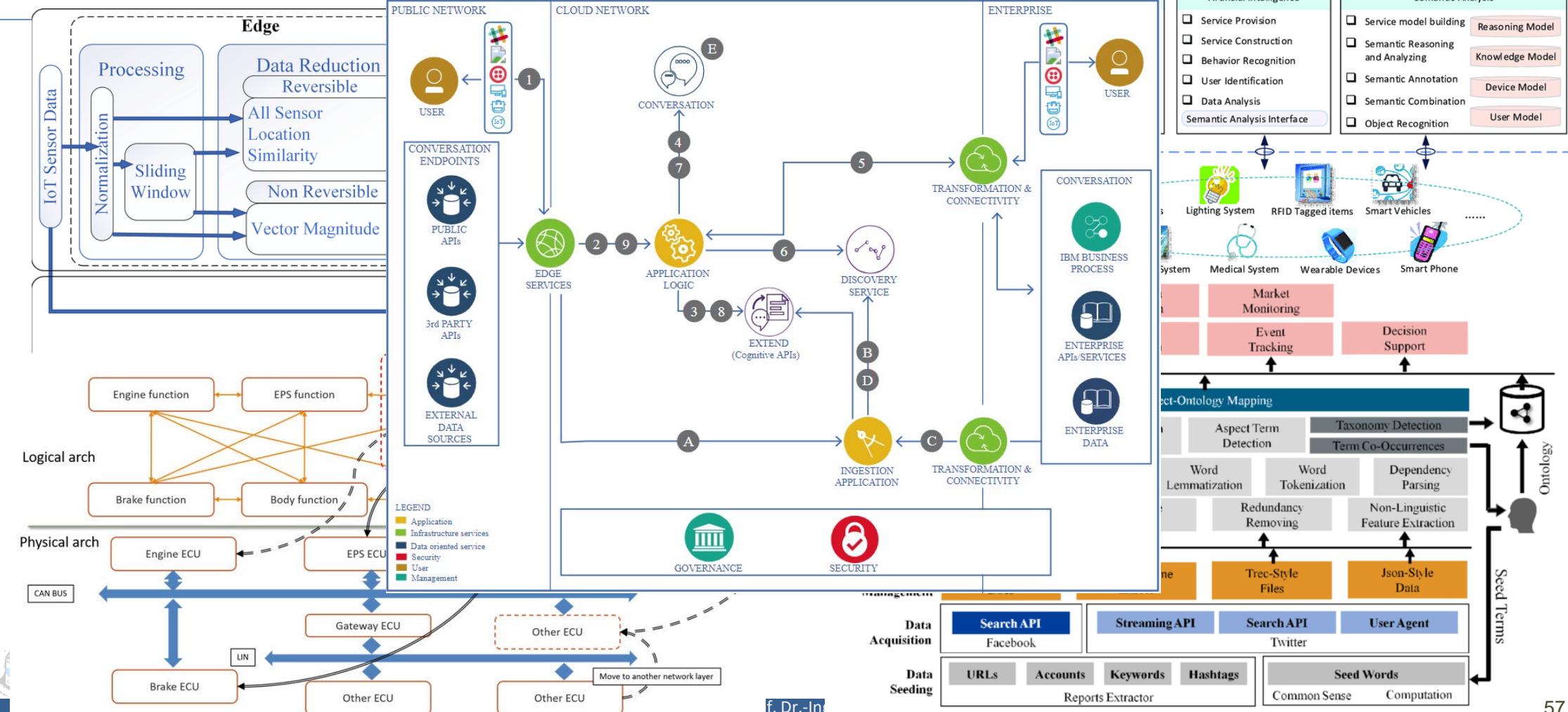
Probleme von Architekturdiagrammen

- Boxen: Unterschiedliche Formen, Größen, Farben, Kanten, Rahmen, etc.
- Linien: Formen, Pfeile, Farben, Muster, Größe, Dicke, Verbindungen
- Symbole und Formen: Unterschiedliche Bedeutungen, unklare Bilder, Akronyme
- Fehlende Elemente: Unverbundene Boxen, fehlende Akteure, Legende?
- Arrangement: Verschachtelung, Layout



Source: Simon Brown

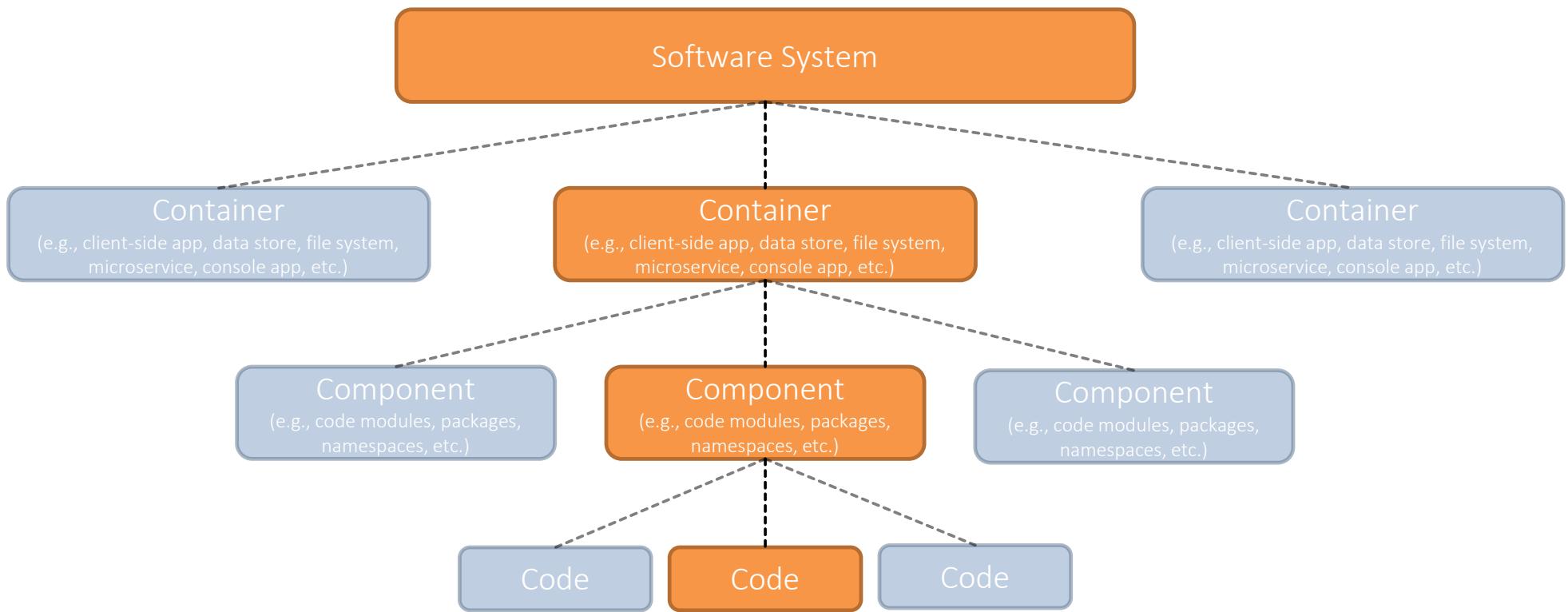
Beispiel: Architekturdiagramme



Einsicht: Abstraktionen elementar; Notationen zweitrangig

- Notationen sind komplex und können unterschiedliche Dinge für unterschiedliche Personen bedeuten
- Notationen sind oft fehlerhaft verwendet, falsch interpretiert, ad-hoc erfunden oder vergessen
- Abstraktionen sind das Schlüsselement eines jeden Systems
- Egal was für eine Notation man verwendet, Abstraktionen sind vorhanden

Modellierung der Architektur mit C4



C4 Model Overview

- Idee: Beschreibung des Softwaresystems in unterschiedlichen Detaillevel.
- Kein Zwang zur Modellierungsreihenfolge (button-up oder top-down).
- Kein Zwang alle Levels zu modellieren.
- C4 = Context, Containers, Components, Code

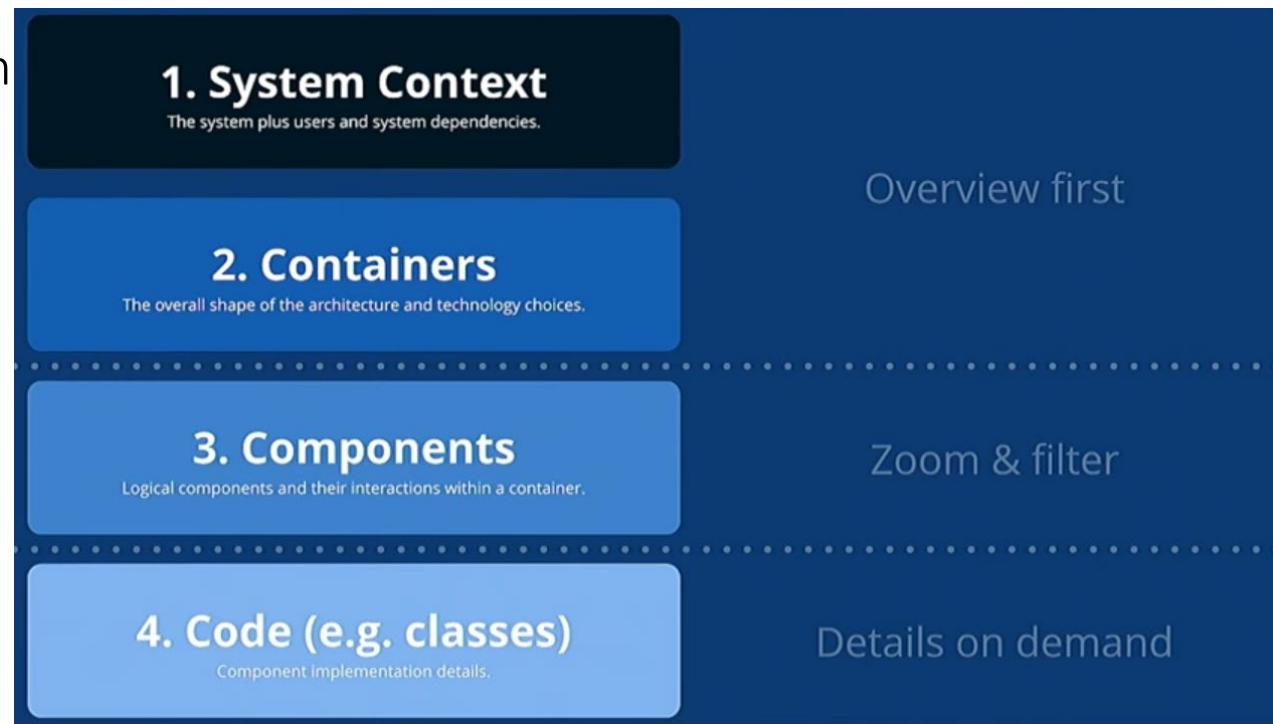


Diagramm == Karte für Entwicklerinnen

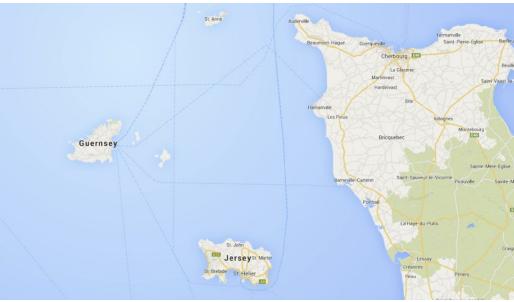
Detailliert; Realwelt
Aussehen, aber wo bin ich?



Zooming out; Wir sind auf einer Insel, aber auf welcher?



Wir sind auf Jersey, neben einer Insel / Kontinent?

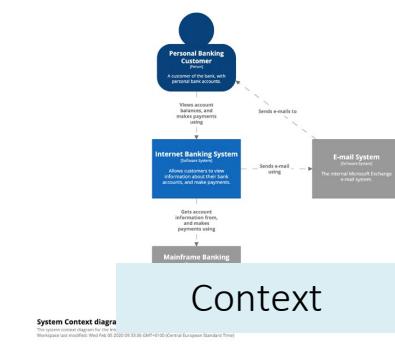
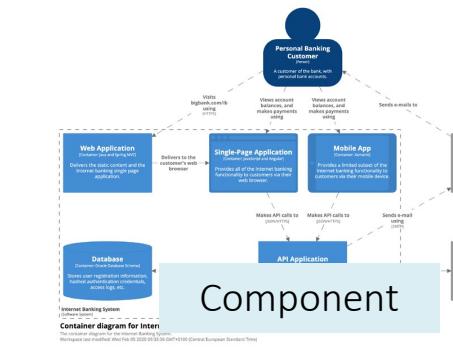
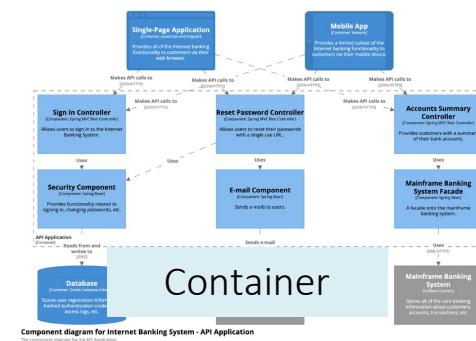
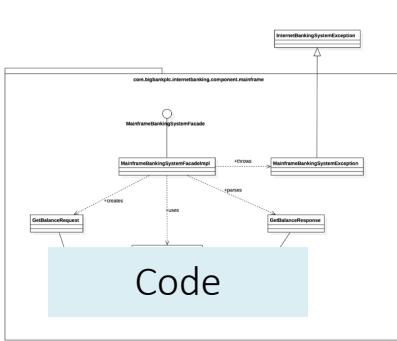


Wir sind nah an Frankreich!

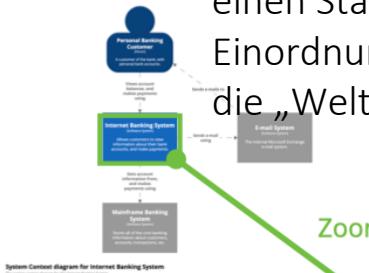


Zoom-Levels können unterschiedliche Geschichten erzählen und verschiedene Aspekte zu unterschiedlichen Adressaten des Systems hervorheben.

In der SW Entwicklung haben wir unterschiedliche Adressaten (Nutzende, Kunden, ArchitektInnen, Testende, Data Scientists, etc.). Also, sollten wir auch unterschiedliche Aspekte beleuchten.



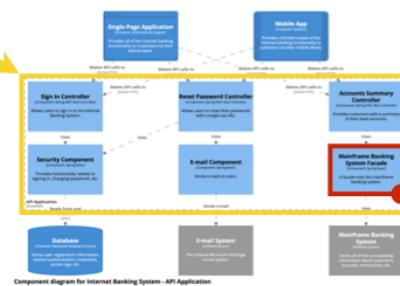
Das **System Context-Diagramm** stellt einen Startpunkt her, der die Einordnung des Softwaresystems in die „Welt“ darstellt.



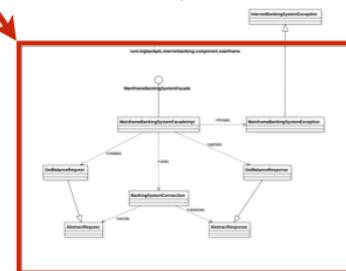
Ein **Container-Diagramm** zeigt die abstrakten Bausteine des Systems.



Ein **Component-Diagramm** zeigt die Komponenten / Module eines Containers.



Ein **Code** (z.B. UML Klassen-) Diagramm zoomt in eine individuelle Komponente, um deren Implementierung darzustellen.



Level 1
Context

Level 2
Containers

Level 3
Components

Level 4
Code

Abstraktionen: Elementare Elemente

Person: Nutzer eines Systems (Aktoren, Rolen, etc.)

Softwaresystem: Software, die Wert für den Kunden liefert (schließt nicht-Menschen mit ein, wie z.B. andere Systeme die darauf angewiesen sind bzw. vice versa).

Container: Anwendungen oder Datenhaltung, die ausgeführt werden muss, so dass das gesamte System operieren kann. Auslieferbare/Ausführbare Einheit oder Laufzeitumgebung.

Component: Gruppe von verwandter Funktionalität hinter einem wohl-definierten Interface (Schnittstelle). Komponenten im selben Container werden im selben Prozessraum ausgeführt.

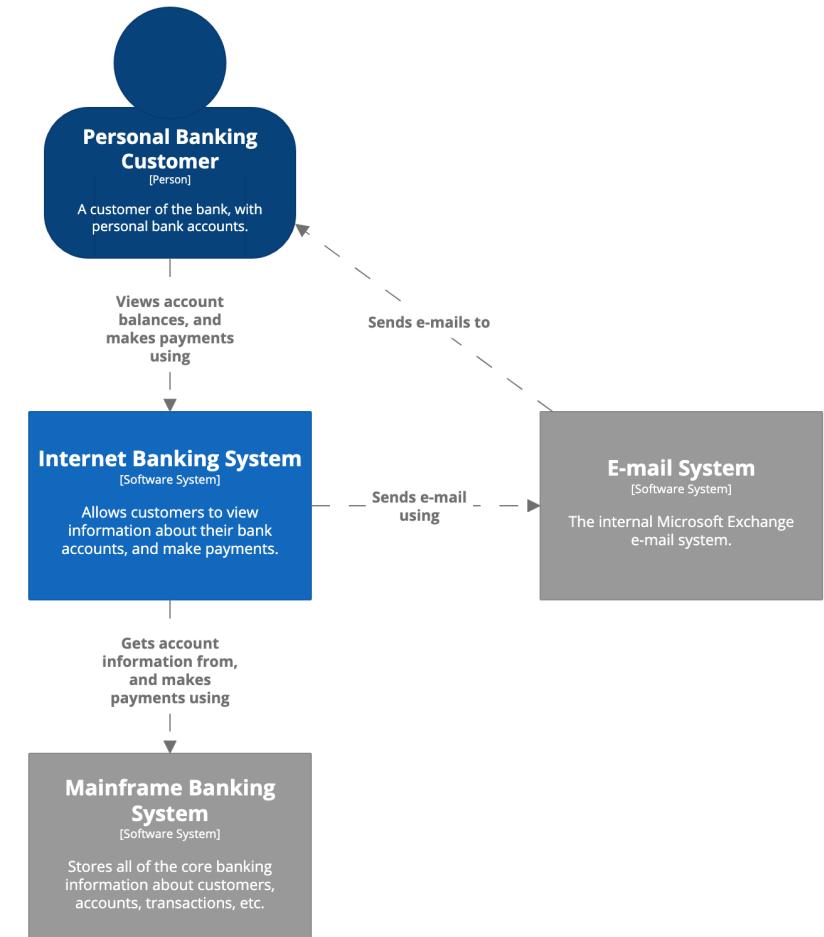


C1: Context-Diagramm

Globale Sicht auf das System und wie es in der aktuellen Systemlandschaft, einschließlich möglichen Nutzern und anderen Systemen sich integriert.

Ideale Sicht, um das System innerhalb anderer Datenverarbeitenden bzw. –produzierenden Systemen zu platzieren und deren Interaktion zu definieren.

Keine Angabe von Technologien, Protokollen und Systemdetails benötigt. Hauptsächlich zur Kommunikation mit Nicht-technischen Personen verwendet.

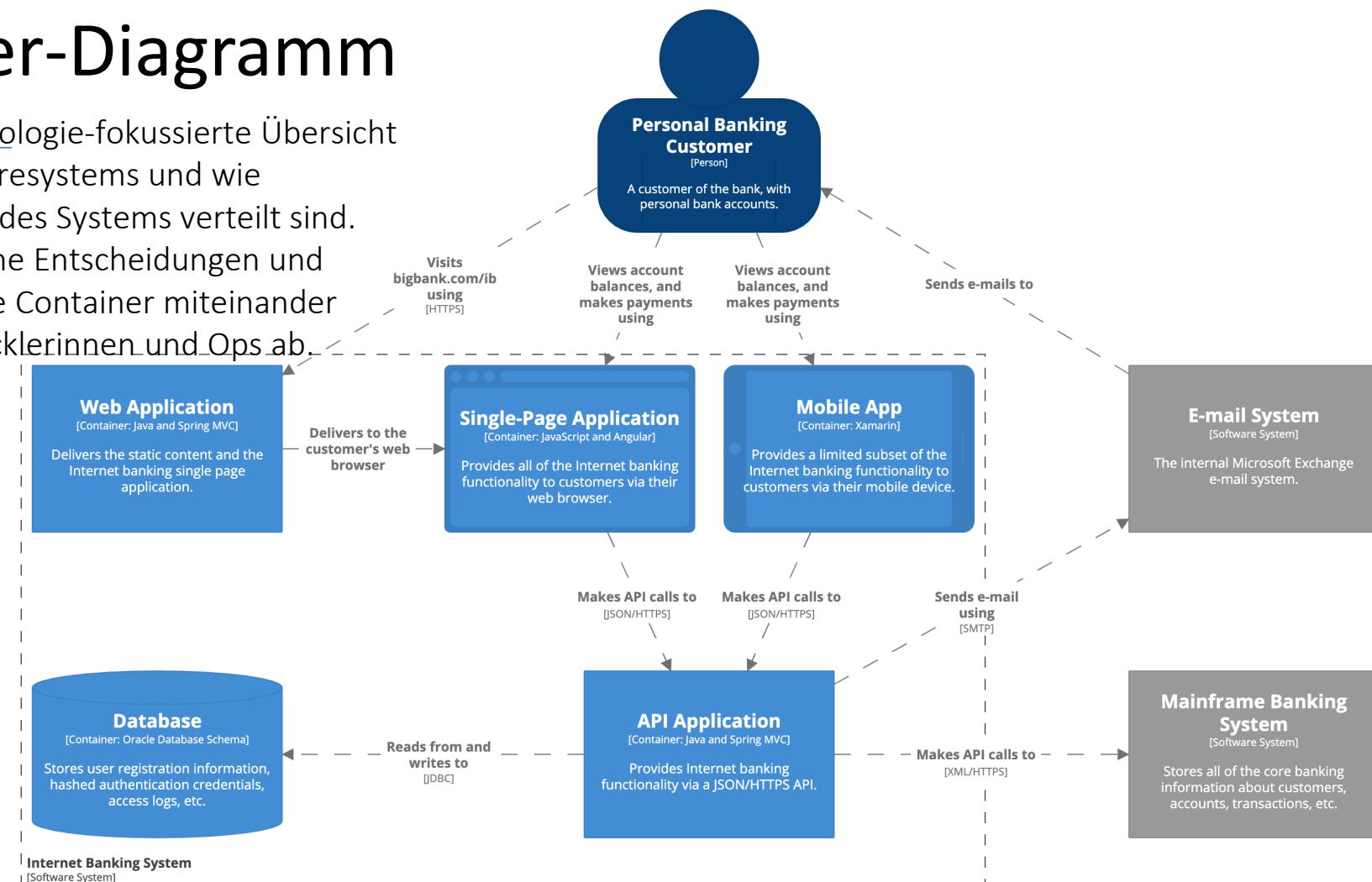


System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

C2: Container-Diagramm

Repräsentiert eine grobe, technologie-fokussierte Übersicht über die Architektur des Softwaresystems und wie Verantwortlichkeiten innerhalb des Systems verteilt sind. Beinhaltet wichtig technologische Entscheidungen und beschreibt, wie unterschiedliche Container miteinander kommunizieren. Zielt auf Entwicklerinnen und Ops ab.



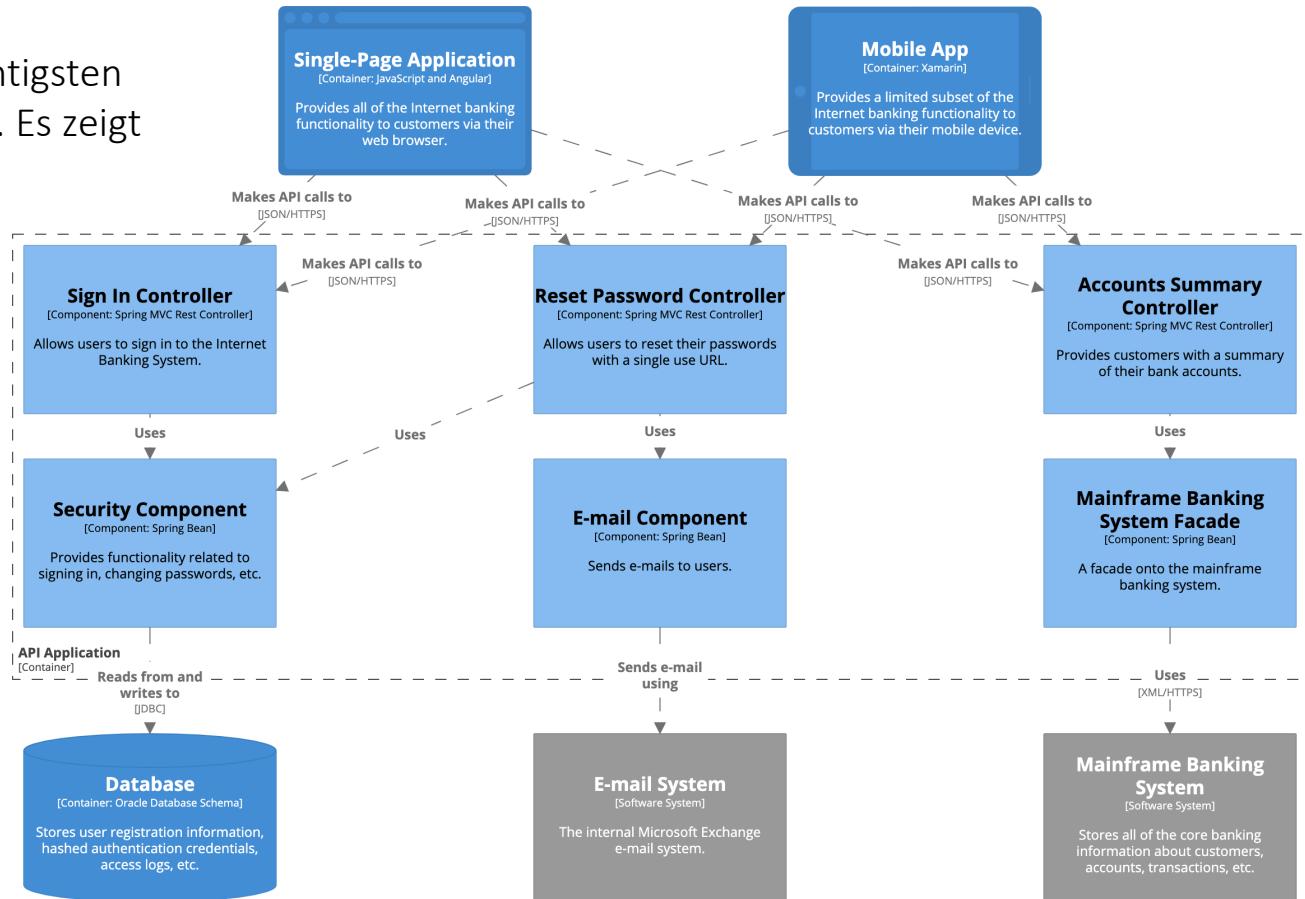
Container diagram for Internet Banking System

The container diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

C3: Component-Diagramm

Das Komponentendiagramm visualisiert die wichtigsten Bausteine (also Komponenten) eines Containers. Es zeigt deren Verantwortlichkeiten, Technologien und Implementierungsdetails.

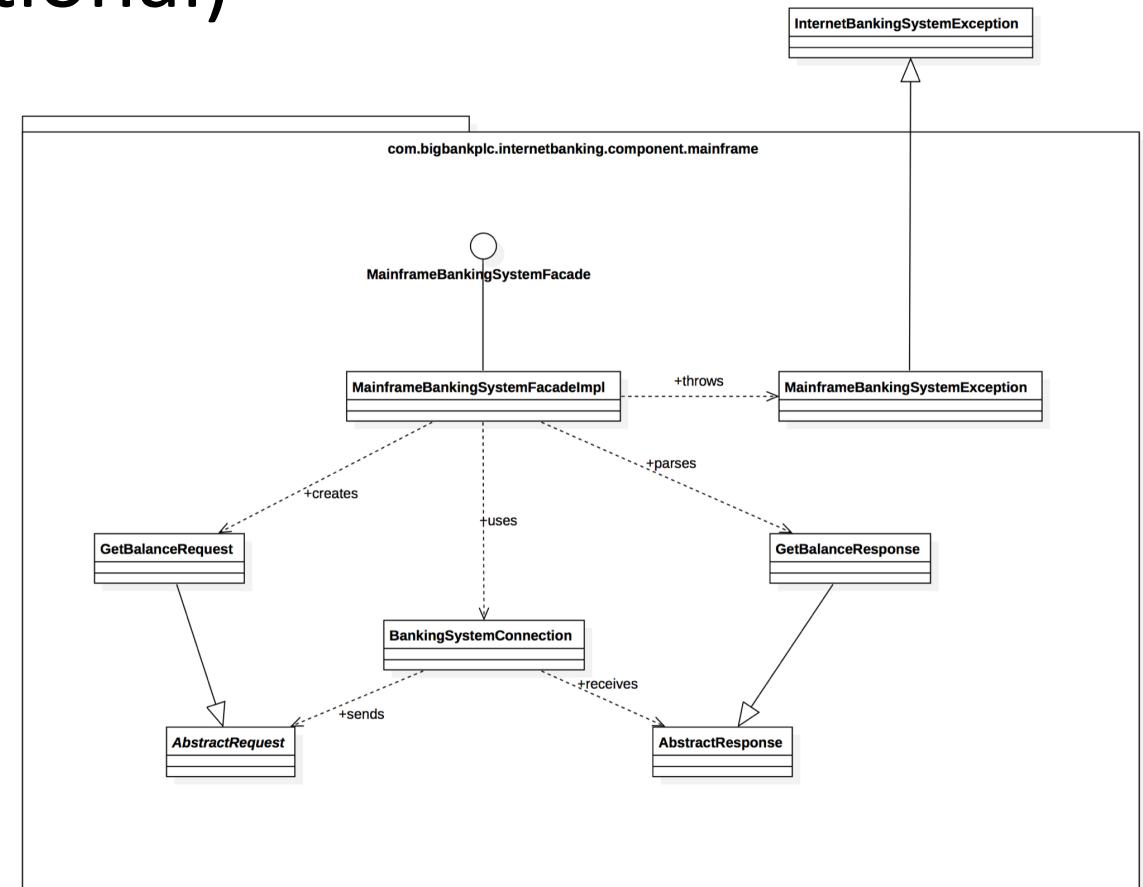
Ziel auf Entwicklerinnen ab.



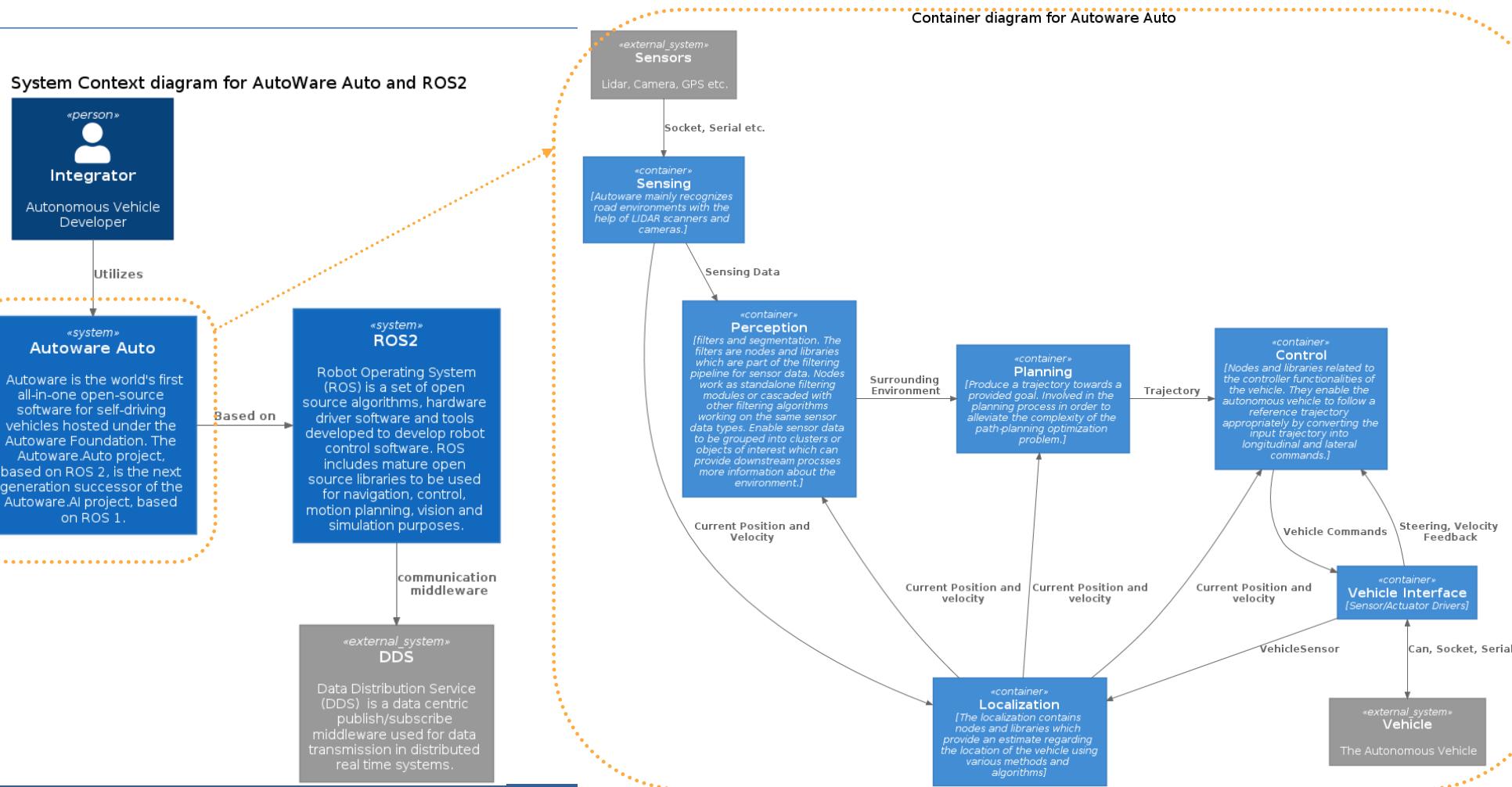
C4: Code-Diagramm (optional)

Das Code-Diagramm ist nicht spezifiziert und hängt vom Anwendungsfall ab. Oftmals werden UML-Klassendiagramme, ER-Diagramme oder Zustands- und Sequenzdiagramme verwendet.

Ein solches Diagramm könnte auch von einer IDE generiert werden. Nur für sehr komplexe Komponenten empfohlen.

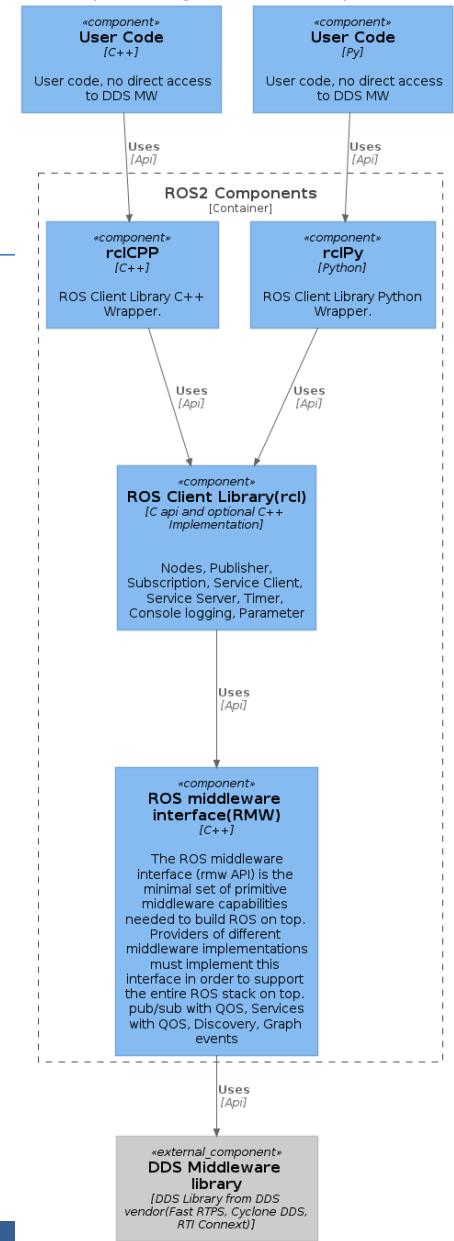


Bespiel: Autonomer Roboter mit C4

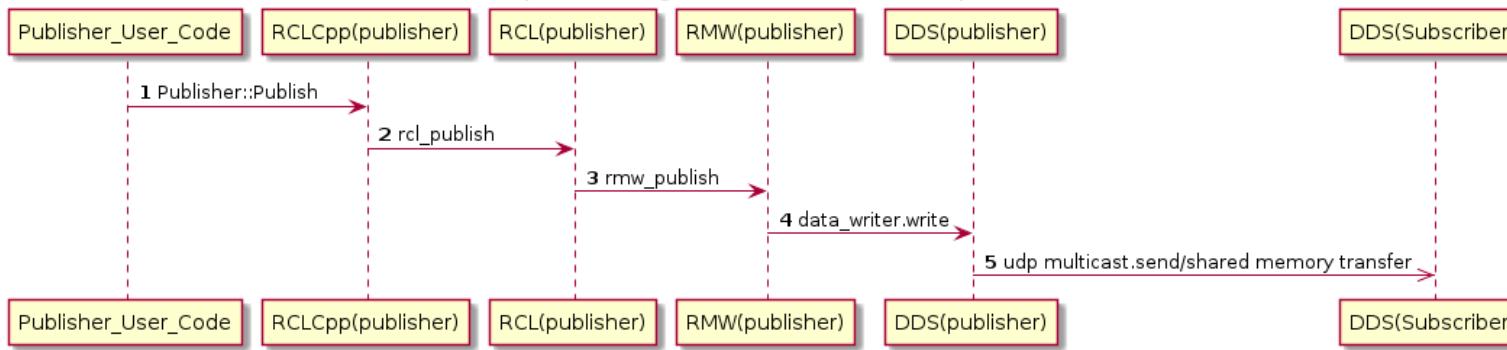


Beispiel: Autonomer Roboter mit C4

Component Diagram for ROS2 components



Sequence Diagram for Ros2 Mw components



Gute Architektur == Team Aufwand

- Vermeide Inselwissen und nehme unterschiedliche Perspektiven ein (Entwicklerin, Data Scientists, Nutzende, etc.) indem das Team sich für eine Architektur entscheidet
- Erfahrene Expertinnen kennen die Werkzeuge und Technologien für die sie Entscheidungen treffen müssen
- Domänenexpertinnen kennen die Schnittstellen und Konsequenzen mit der Umgebung sowie die Anwendungsszenarien
- Managers kennen organisatorische Randbedingungen (Komitees, Firmenpolitik, Ressourcen)



Was Sie mitgenommen haben sollten:

- Inwiefern wirkt sich die Architektur auf das Softwaresystem aus?
- Wie kann die Auswahl einer geeigneten Architektur den Entwurf / Implementierung vereinfachen?
- Was bedeutet Kopplung und Kohäsion auf Architekturebene?
- Was ist ein Architektur-Stil?
- Für welche Szenarien sind MVC oder Pipes and Filters sinnvoll?
- Was sind Limitierungen von monolithischen Systemen?
- Was sollten Schichten nicht auf Elemente in Schichten darüber Zugriff haben?
- Wann macht der Einsatz von MicroServices Sinn und wann nicht?
- Welche Kriterien für den Einsatz bestimmter Architekturmuster gilt es zu beachten?

Was Sie mitgenommen haben sollten:

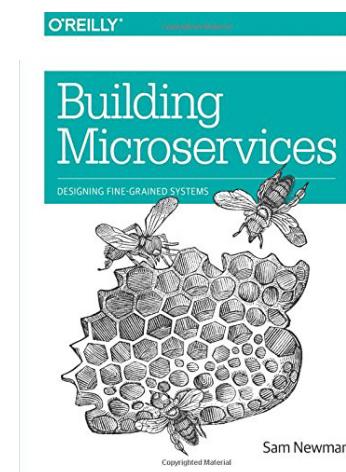
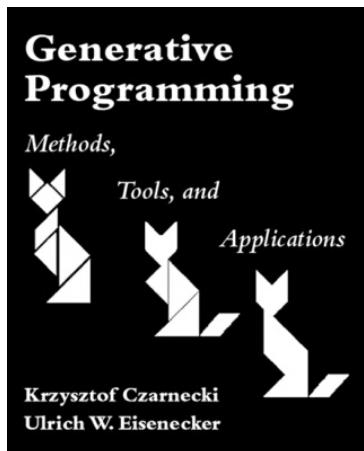
- How does software architecture constrain a system?
- How does choosing an architecture simplify design?
- What are coupling and cohesion?
- What is an architectural style?
- For which application scenarios is MVC beneficial? For which is pipes and filters?
- Why shouldn't elements in a software layer “see” the layer above?
- What is the role of programming in model-driven architecture?

Literatur

- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997 [Chapter 3, 4]
- *Software Engineering*, I. Sommerville, 7th Edn., 2004.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996

Literatur

- Service-Oriented Architecture (SOA) vs. Component Based Architecture. Helmut Petritsch (http://petritsch.co.at/download/SOA_vs_component_based.pdf)
- Microservices. James Lewis und Martin Fowler.
<https://martinfowler.com/articles/microservices.html>



Softwaretechnik

Qualität von Software und Design



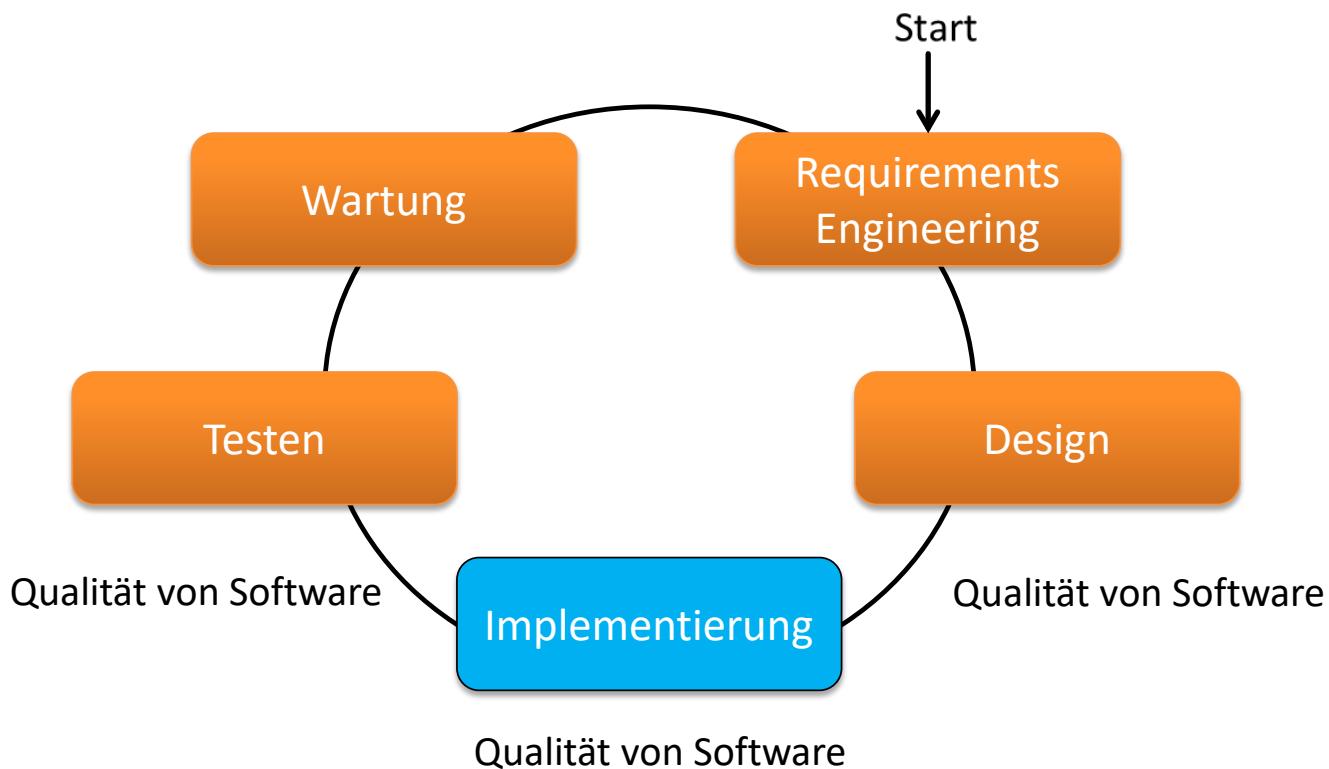
**SOFTWARE
SYSTEME**

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Einordnung



Lernziele

- Wichtige Kriterien für das Design von Software kennenlernen
- In der Lage sein, gutes von schlechten Design zu unterscheiden
- Prinzipien von guter Softwarequalität kennen und anwenden können
- Qualität von Software-Design bewerten können

Was bedeutet gutes Software Design?

- Für jedes Verhalten gibt es unendlich viele Programme
 - Wie unterscheiden sich diese Varianten?
 - Welche Variante hat die beste Qualität?
- Wie soll Variante entworfen werden, damit sie gewünschte Eigenschaften von Qualität besitzt?

Qualität von Software-Design



Was ist Qualität?

- **Interne Qualität**

- Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit
 - Robust gegenüber Änderungen
 - Coupling und Cohesion (was ist das? Wird im Folgenden erklärt)
 - Wiederverwendbarkeit
- Zusammengefasst typischerweise beschrieben als Modularität

- **Externe Qualität**

- Korrektheit: Erfüllung der Anforderungen
- Einfachheit in der Benutzung
- Ressourcenverbrauch
- Legale und politische Beschränkungen

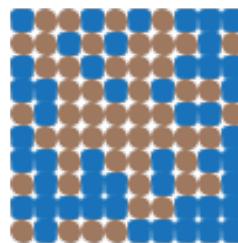
Ist gute Qualität teuer?

- Gibt es einen Trade-off zwischen SW Qualität und Funktionalität?

SW Qualität = Günstigere Entwicklung

Schlechte interne Qualität ~ hohe technische Schulden -> langsamere Weiterentwicklung

If we compare one system with a lot of cruft...

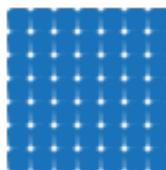


the cruft means new features take longer to build



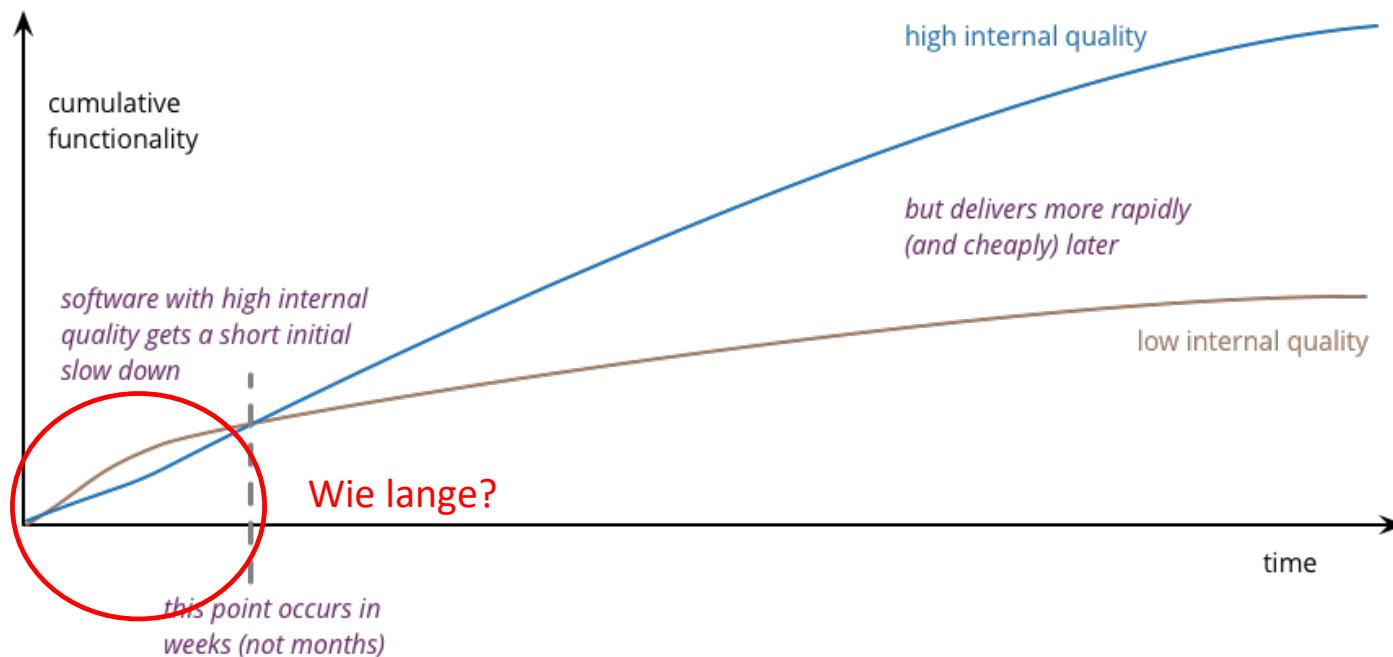
this extra time and effort is the cost of the cruft, paid with each new feature

...to an equivalent one without



free of cruft, features can be added more quickly

SW Qualität zahl sich aus



Bisher: Design

- Nach der Modellierung werden Methoden definiert und zu Klassen zugeordnet sowie die Kommunikation zwischen den Objekten festgelegt, um die spezifizierten Anforderungen zu erfüllen.

Jetzt: Wie gutes Design?

- Wie genau sollte das Design durchgeführt werden, um eine hohe interne Qualität zu erreichen?
 - Welche Methode kommt wohin? Was hat das für Auswirkungen?
 - Wie sollen die Objekte interagieren? Was hat das für Auswirkungen?
 - Wichtige, kritische, nicht-triviale Fragestellung!

Fünf Kriterien für gutes Design



Ziel: Modularität

- Beschreibt, inwieweit man ein Softwaresystem aus autonomen Elementen bauen kann, die mit einer kohärenten, einfachen Struktur miteinander verbunden sind
- Dabei helfen 5 Kriterien und 5 Regeln

Exkurs: Coupling / Kopplung

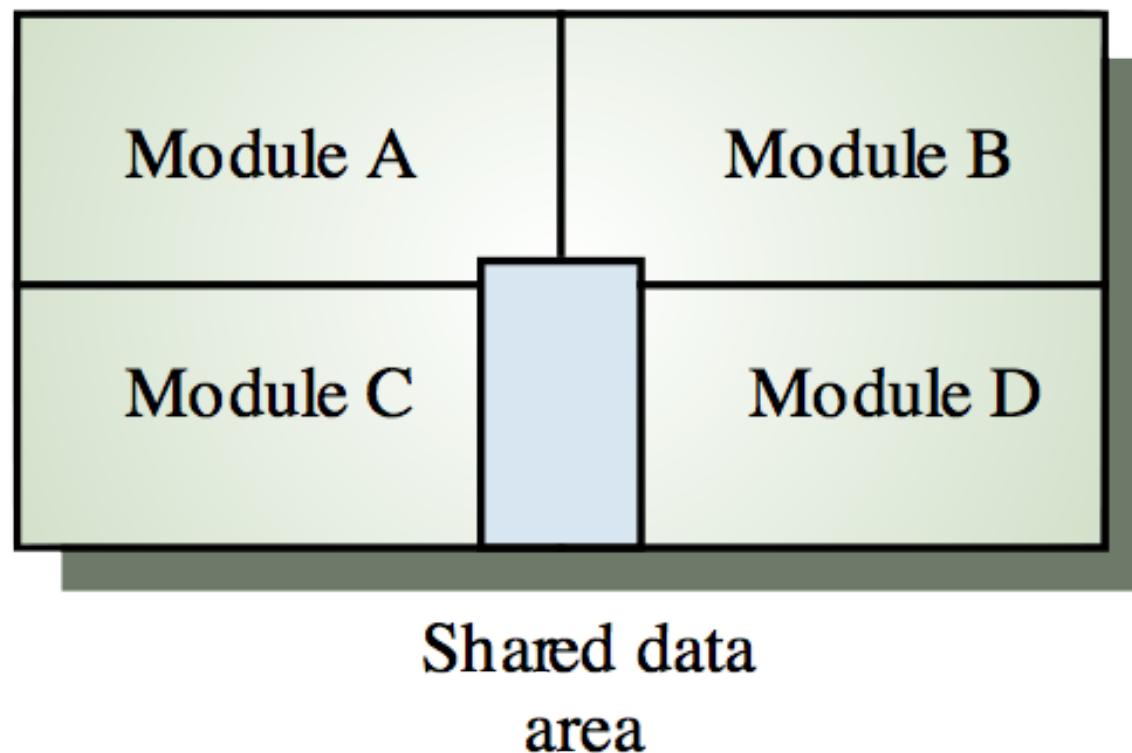
Coupling ist ein Maß der *Stärke der Verbindungen* zwischen Systemkomponenten.

- Coupling ist eng (tight) zwischen Komponenten, wenn sie stark voneinander abhängig sind (z.B., wenn sehr viel Kommunikationen zw. beiden stattfindet).
- Coupling ist lose (loose), wenn es nur wenige Abhängigkeiten zwischen Komponenten gibt.

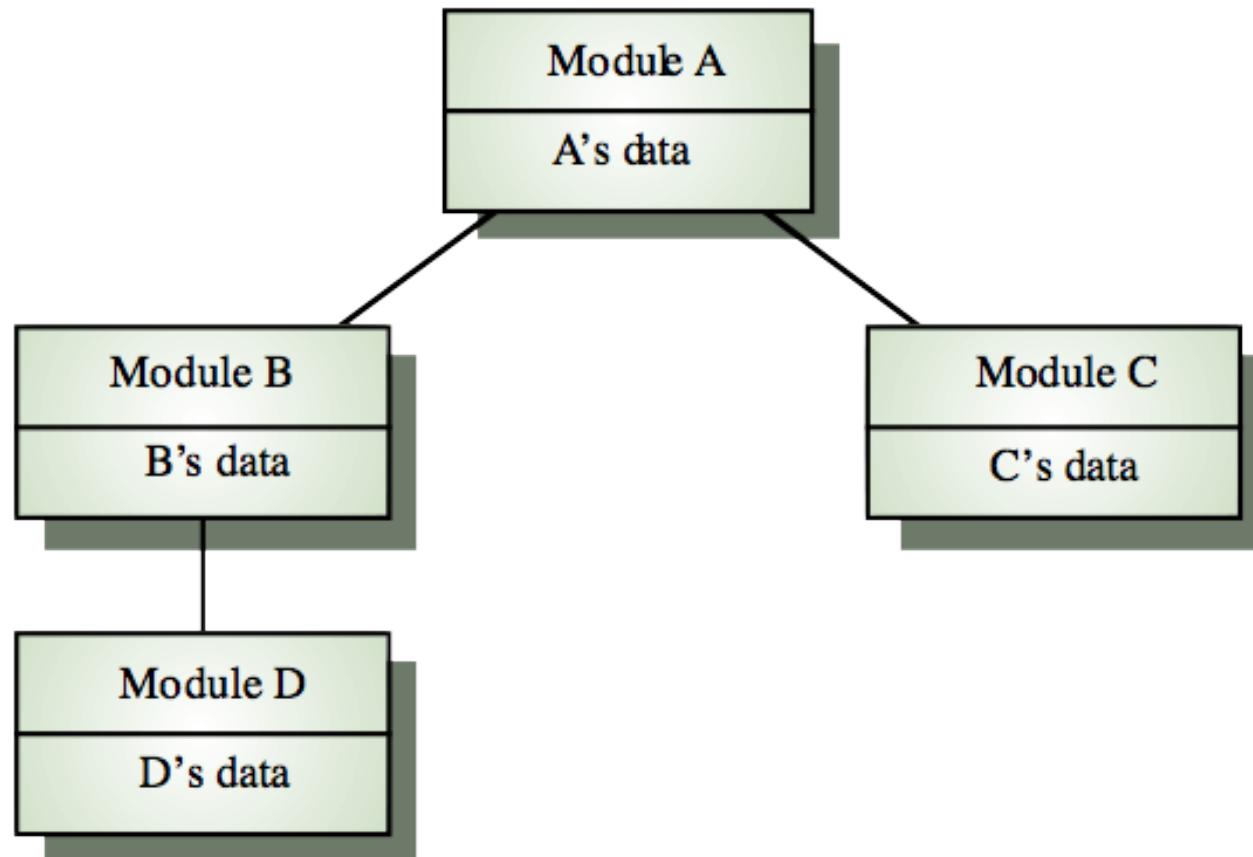
Welche Eigenschaft führt zu einer besseren Qualität?

- **Loose coupling** verbessert Wartbarkeit und Adaptabilität, da *Änderungen in einer Komponente sich weniger wahrscheinlich auf anderen Komponenten auswirkt*.

Tight Coupling



Loose Coupling



Exkurs: Cohesion / Kohäsion

Cohesion ist ein Maß, *wie gut Teile einer Komponente “zusammen gehören”*.

- Cohesion ist schwach, wenn Elemente nur wegen ihrer Ähnlichkeit ihrer Funktionen zusammengefasst sind (z.B., **java.lang.Math**).
- Cohesion ist stark, wenn alle Teile für eine Funktionalität tatsächlich benötigt werden (z.B. **java.lang.String**).

Welche Eigenschaft führt zu einer besseren Qualität?

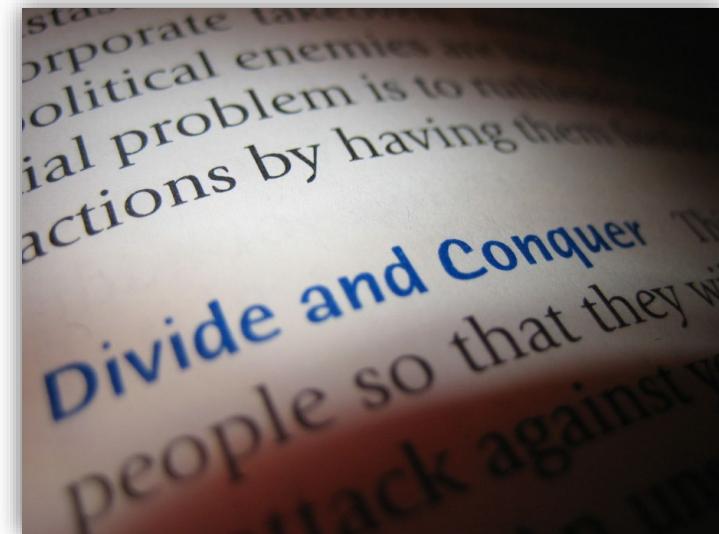
- Starke cohesion *verbessert Wartbarkeit* und Adaptivität durch *die Einschränkung des Ausmaßes von Änderungen* auf eine kleine Anzahl von Komponenten.

Fünf Kriterien für gutes Design

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection

Fünf Kriterien: Modular Decomposability

- Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
- Sub-Probleme sind durch einfache Struktur verbunden
- Sub-Probleme sind unabhängig genug, dass sie einzeln bearbeitet werden können



Fünf Kriterien: Modular Decomposability

- Modular decomposability setzt voraus: Teilung von Arbeit möglich
- Beispiel: Top-Down Design
- Gegenbeispiel: Ein globales Initialisierungsmodul, eine große Main-Methode

Fünf Kriterien: Modular Composability

- Gegenstück zu modular decomposability
 - Softwarekomponenten können beliebig kombiniert werden
 - Möglicherweise auch in anderer Domäne
-
- Beispiel: Tischreservierung aus NoMoreWaiting kann auch für das Vormerken von Büchern benutzt werden (gutes Design!)

Fünf Kriterien: Modular Understandability

- Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen bzw. möglichst wenige andere kennen zu müssen
- Wichtig für Wartung
- Gilt für alle Softwareartefakte, nicht nur Quelltext
- Gegenbeispiel: Sequentielle Abhängigkeit zwischen Modulen

Fünf Kriterien: Modular Continuity

- Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul bzw. möglichst wenigen Modulen
- Beispiel 1: Symbolische Konstanten (im Gegensatz zu Magic Numbers)
- Beispiel 2: Datendarstellung hinter Interface kapseln
- Gegenbeispiel: Magic Numbers, (zu viele) globale Variablen

Fünf Kriterien: Modular Protection

- Abnormales Programmverhalten in einem Modul bleibt in diesem Modul bzw. wird zu möglichst wenigen Modulen propagiert
- Motivation: Große Software wird immer Fehler enthalten
- Beispiel: Defensives Programmieren
- Gegenbeispiel: Nullpointer in einem Modul führt zu Fehler in anderem Modul

Fünf Regeln für gutes Design



Fünf Regeln

- Fünf Regeln für qualitativ hochwertiges Software-Design:
 - Direct Mapping
 - Few Interfaces
 - Small Interfaces
 - Explicit Interfaces
 - Information Hiding

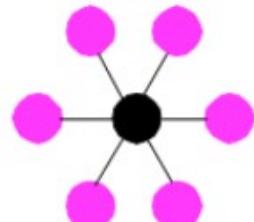
Regel 1: Direct Mapping

- Modulare Struktur des Softwaresystems sollte modularer Struktur des Modells der Problemdomäne entsprechen
- Folgt aus continuity und decomposability
- A.k.a. “low representational gap”[C. Larman]

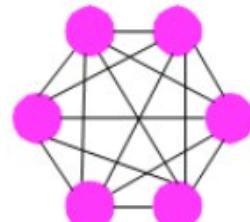
Regel 2: Few Interfaces

- Jedes Modul sollte mit möglichst wenigen anderen Modulen kommunizieren
- Folgt aus continuity and protection
- Struktur mit möglichst wenigen Verbindungen

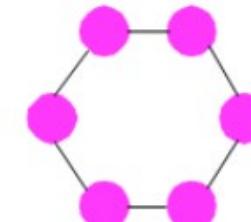
*Types of module
interconnection
structures*



(A)



(B)



(C)

Regel 3: Small Interfaces

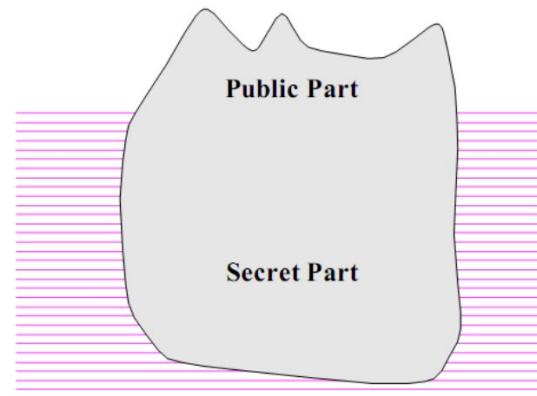
- Wenn zwei Module kommunizieren, sollten sie so wenig Informationen wie möglich austauschen
- Folgt aus continuity und protection, notwendig für composability
- Gegenbeispiel: Big Interfaces ☺
 - Wenn ich jede Methode ins Interface aufnehme, muss jede Klasse, die das Interface implementiert, alle Methoden realisieren -> Vorteil geht verloren
 - Man weiß nicht, welche Methoden entscheidend sind
 - Zu viele Einfallstore für Fehler

Regel 4: Explicit Interfaces

- Wenn zwei Module miteinander kommunizieren, muss das aus dem Interface von mindestens einem hervorgehen
- Gegenbeispiel: Globale Variablen
 - Wer benutzt sie?
 - Welche Auswirkungen (und wo) werden durch Änderungen an der globalen Variable verursacht?
 - Wer ändert sie und wann?
 - Wer war verantwortlich für einen inkonsistenten oder fehlerhaften Zustand?

Regel 5: Information Hiding

- Jedes Modul muss eine Teilmenge seiner Eigenschaften definieren, die nach außen gezeigt werden
- Alles andere wird “versteckt”
- Nicht nur Inhalt, auch Implementierung wird versteckt
- Impliziert durch continuity



SOLID: Prinzipien für OO Design

Ziel: Software verständlicher, flexibler und wartbarer zu machen

Single-Responsibility Principle
Open-Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Prinzipien stellen Basis eines jeden OO-SW Designs dar

Single-Responsibility Principle

- Jedes Modul, Klasse, Funktion eines Programms soll Verantwortung über eine bestimmte Teilfunktionalität haben
- Es sollte nur diese **eine** Verantwortung haben (siehe Responsibility-Driven Design)
- Effekt:
 - Robust gegenüber zukünftigen Änderungen
 - Änderungen sind lokalisiert an einen Ort
 - Einfaches Verständnis über die Verantwortung, wenn limitiert auf eins

Open-Closed Principle

- Software Entitäten (Module, Klassen, Funktionen, etc.) sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein
- Funktionalität kann erweitert werden ohne existierenden Code zu modifizieren
- Effekte:
 - Klasse kann z.B. durch Vererbung erweitert werden ohne vorhandenen Code zu modifizieren
 - Klasse kann von anderen benutzt werden, ohne sie verändern zu müssen (Information Hiding)

Liskov Substitution Principle

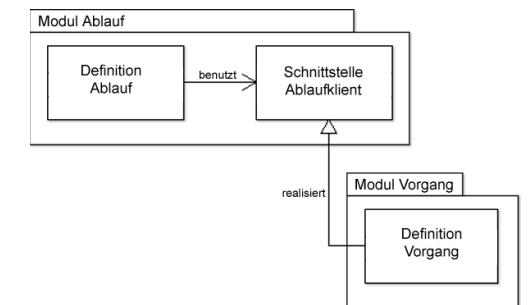
- Jedes Objekt S, welches ein Subtyp (Kindklasse) eines Typs T ist, sollte anstelle eines Objekts von Typ T benutzen werden können, ohne das sich das Verhalten des Programmes (Korrektheit, Performance, etc.) verändert
 - Strong behavioral subtyping genannt
-
- Effekte:
 - Garantiert semantische Interoperabilität von Typen in einer Hierarchie

Interface Segregation Principle

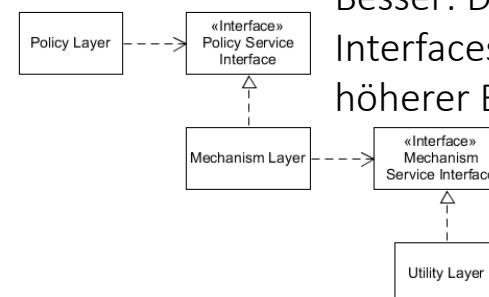
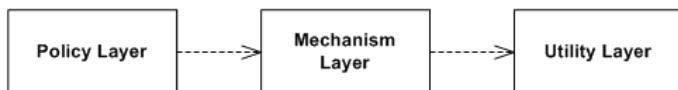
- Kein Klient sollte nicht dazu gezwungen werden, abhängig von Methoden zu sein, die man nicht nutzt
- Große Interfaces in mehrere kleinere aufteilen, so dass Klienten nur tatsächlich benutzt Teile implementieren müssen
- Effekte:
 - Komponenten werden in kleinere, leichter zu wartende Systeme zerlegt
 - SW ist einfacher zu refaktorisieren durch kleinere Interfaces
 - Klassen müssen nur die für sie relevanten Methoden kennen
 - Führt zu höherer Kohäsion (siehe GRASP)

Dependency Inversion Principle (DIP)

- Module höherer Ebenen sollten nicht von Modulen von unteren Ebenen abhängig sein, stattdessen nur von Abstraktionen
- Abstraktionen sollten nicht von Details abhängen, sondern anders herum
- Effekte:
 - Führt zu verringerte Kopplung von Komponenten
 - Verhindert zyklische Abhängigkeiten zwischen Komponenten



Höhere Ebene ist abhängig von Detailebene



Besser: Details sind durch Interfaces abgekoppelt von höherer Ebene

Beispiel: DIP

```
public class Lampe {  
    private boolean leuchtet;  
  
    public void anschalten() {  
        leuchtet = true;  
    }  
  
    public void ausschalten() {  
        leuchtet = false;  
    }  
}
```

Schalter steuert Lampe, daher höheres Modul, aber Schalter ist abhängig von Lampe; falsche Abhängigkeitsrichtung

```
public interface SchalterClient {  
    public void anschalten();  
    public void ausschalten();  
}
```

Entkopplung der Abhängigkeiten durch Interface

```
public class Lampe implements SchalterClient {  
    private boolean leuchtet;  
  
    public void anschalten() {  
        leuchtet = true;  
    }  
  
    public void ausschalten() {  
        leuchtet = false;  
    }  
}
```

Lampe wird nun abhängig von Abstraktion (Interface); Richtung low level to high level

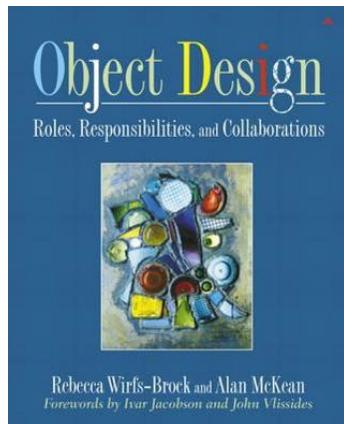
Schalter kann nun beliebige Entitäten steuern

```
public class Schalter {  
    private Lampe lampe;  
    private boolean gedrueckt;  
  
    public Schalter(Lampe lampe) {  
        this.lampe = lampe;  
    }  
  
    public void drueckeSchalter() {  
        gedrueckt = !gedrueckt;  
        if(gedrueckt) {  
            lampe.anSchalten();  
        } else {  
            lampe.ausschalten();  
        }  
    }  
  
    public class Schalter {  
        private SchalterClient client;  
        private boolean gedrueckt;  
  
        public Schalter(SchalterClient client) {  
            this.client = client;  
        }  
  
        public void drueckeSchalter() {  
            gedrueckt = !gedrueckt;  
            if(gedrueckt) {  
                client.anSchalten();  
            } else {  
                client.ausschalten();  
            }  
        }  
    }
```


GRASP Pattern

- Allgemeine Lernhilfe, um
 - grundlegendes objekt-orientiertes Designen zu verstehen
 - Design-Entscheidungen methodisch, rational und erklärbar zu treffen
- Basiert of Responsibilities (Responsibility-Driven Design)

<http://www.wirfs-brock.com/>



Responsibilities (Recap)

- Bezogen darauf, wie sich Objekt verhalten soll
- Zwei Arten
 - Wissen
 - Wissen über private Daten
 - Wissen über Objekte, die sich auf einander beziehen
 - Wissen, was es ableiten oder berechnen kann
 - Tätigkeit
 - Etwas selbst tun, z.B. Objekt erstellen oder berechnen
 - Aktionen in anderen Objekten initiieren
 - Kontrolle und Koordination von Aktivitäten anderer Objekte

GRASP-Pattern

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations
- Pure Fabrication
- Protected Variations

GRASP-Prinzip: Information Expert

- Problem: Wer soll die Responsibilities bekommen?
- Lösung: Das Objekt, das die meisten Informationen (Daten) hat, diese Responsibility zu erfüllen

- Beispiel NoMoreWaiting: Wer hat die Information, ob ein Tisch frei ist oder nicht?

GRASP-Prinzip: Creator

- Problem: Wer erstellt Instanzen eines Objekts?
- Lösung: Creator braucht erstelltes Objekt häufig in seinem Lebenszyklus
- Objekt B bekommt Verantwortung, Objekt A zu erstellen, wenn:
 - B A-Objekte aggregiert
 - B A-Objekte enthält
 - B Instanzen von A-Objekten loggt
 - B Initialisierungsdaten von A hat

GRASP-Prinzip: Controller

- Problem: Welches Objekt nach dem UI koordiniert die Business-Logik / Programmablauf?
- Lösung: Verantwortung bekommt ein Objekt mit folgenden Möglichkeiten:
 - Fassade Controller
 - Objekt repräsentiert das Gesamtsystem,
 - Ist ein Root-objekt,
 - Repräsentiert das Gerät, worauf die SW läuft
 - Use Case Controller
 - Alle Ereignisse eines Use Cases werden in dieser Klasse koordiniert
- Controller in MVC?
 - Nein, sollte nur delegieren und nicht koordinieren
 - Service im Modell sollte koordinieren

GRASP-Prinzip: Loose Coupling

- Problem: Wie kann ich die Auswirkungen von Änderungen und Abhängigkeiten reduzieren und Wiederverwendung erhöhen?
- Lösung: Verantwortung so zuteilen, dass Kopplung verringert wird
 - Siehe SOLID und Interfaces

GRASP-Prinzip: High Cohesion

- Problem: Wie können Objekte fokussiert, verständlich und wartbar gehalten werden?
- Lösung: Verantwortung so zuteilen, dass Kohäsion hoch ist
 - Nicht-verwandte Daten und Funktionen aus Klassen entfernen
 - Nur auf tatsächliche Verantwortung konzentrieren
 - Faustregel: Klasse mit starker Kohäsion hat meistens wenige Methoden, die verwandte Funktionalität haben, und macht nicht zu viel (d.h., ist keine Gottklasse)

GRASP-Prinzip: Indirection

- Problem: Wo Verantwortung zuweisen, um direkte Kopplung zweier Klassen zu verhindern?
- Lösung: Verantwortung auf eine Zwischenklasse zuweisen, so dass direkte Kopplung verhindert wird
 - Siehe Mediator Design Pattern
 - Reduziert jedoch Lesbarkeit und Verständlichkeit

GRASP-Prinzip: Polymorphism

- Problem: Wie kann ich Alternativen basierend auf Typen handeln?
- Lösung: Wenn Alternativen variieren durch den Typen, weise Verantwortlichkeiten abhängig von Typ zu
 - Siehe Strategy Design Pattern (initialisiere im Konstruktor ein Objekt vom Typ, dessen Verantwortlichkeit gerade benötigt wird)
 - Ermöglicht unterschiedliche Implementierungen für denselben Input

GRASP-Prinzip: Pure Fabrication

- Problem: Welches Objekt bekommt Verantwortlichkeiten, wenn ich dadurch die anderen Prinzipien (High Cohesion, Low Coupling) verletzen würde?
- Lösung: Weise ein kohäsive Menge an Verantwortlichkeiten zu einer künstlichen oder unterstützenden Klasse (z.B. Utility Class) zu, welche kein Domänenkonzept repräsentiert
 - Einführung von Utility- / Supportklassen

GRASP-Prinzip: Protected Variations

- Problem: Wie designe ich Klassen, Systeme und Subsystem so, dass Variationen und Änderungen keinen ungewünschten Effekt auf andere Elemente hat?
- Lösung: Identifiziere Stellen dieser Variationen und weise Verantwortlichkeiten zu, um ein stabiles Interface herum zu schaffen
 - Encapsulation durch Interfaces und Information Hiding
 - Modularity erhöhen
 - Design Patterns anwenden
 - Virtualisierung und Containerisierung für (Sub)Systeme
 - Ereignisbasierte Kommunikation und Architekturen

Software Qualität



Aufgabe

- Was ist Software Qualität?
- Wie würden Sie Software Qualität messen?

Software Qualität

- Korrekte Implementierung der Anforderungen
- Design-Qualität:
 - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit, Wiederverwendbarkeit,...
 - Robustheit gegen Änderungen
 - Modularität, Low Coupling, High Cohesion
- Zuverlässigkeit, geringe Fehleranfälligkeit, Testbarkeit, Performance
- Usability, Effizienz, Erlernbarkeit

Software Qualität: Probleme

- Spannung zwischen Stakeholdern
 - Kunde: Effizienz, Usability
 - Entwickler: Wartbarkeit, Wiederverwendbarkeit, Verständlichkeit
- Wie misst man eigentlich Software Qualität?
 - „You can't control what you can't measure“*
 - Software-Metriken? -> direkt messbar
 - Empirische Untersuchungen? -> indirekt messbar

*Tom DeMarco, 1986

Software Metriken

- Lines of Code (LoC)
- Bugs per line of code
- Comment density
- Cyclomatic complexity (measures the number of linearly independent paths through a program's source code)
- Halstead complexity measures (derive software complexity from numbers of (distinct) operands and operators)
- Program execution time
- Test Coverage
- Number of classes and interfaces
- Abstractness = ratio of abstract classes to total number of classes
- ...

Software Metriken werden selten benutzt

- Einige Firmen erstellen Messprogramme, noch weniger haben Erfolg damit
- Firmen, die Metriken benutzen, machen das oft nur, um bestimmten Qualitätsstandards zu genügen
 - Bei Interesse: N. E. Fenton, "Software Metrics: Successes, Failures & New Directions", 1999.
- Hier kommt wieder der Unterschied zur Ingenieursdisziplin zum Tragen
 - Ingenieure sind erfolgreich, weil sie Qualität gut messen können
 - Geht eben nicht bei Softwareprodukten

Software Metriken sind selten sinnvoll

- Formal definierte Metriken sind zwar objektiv, aber was bedeuten sie?
 - Was sagt eine zyklomatische Komplexität von 12 über die Qualität des Quelltextes aus?
 - Gar nichts
- Oft unklar, ob Metrik mit irgendeinem sinnvollen Qualitätskriterium zusammenhängt
 - Wäre so, als würde man Intelligenz mit Hirnmasse oder Kopfumfang messen wollen

Software Qualität

- Wenn Metriken nicht sinnvoll sind, wie dann Software-Qualität messen?
- Analyse für jedes Softwareprojekt
 - Design-Qualität, Erweiterbarkeit
 - Vergleich zu anderen Designs, die sich als sinnvoll erwiesen haben (z.B. Design Patterns)
 - Code smells und Antipatterns suchen
 - Umfangreiche Tests und Code Reviews
 - (Formale) Verifikation
 - Nutzerstudien

Qualitätsmerkmale

- Beziehen sich auf Produkt und Prozess
 - Produkt: an Kunden geliefert, externe Qualität
 - Prozess: produziert das Produkt, interne Qualität
- Qualitätsprozess ist notwendige Bedingung für ein Qualitätsprodukt

Qualitätsmerkmale

- Korrektheit (siehe Testen-Vorlesung)
- Zuverlässigkeit (Wahrscheinlichkeit, dass System erwartungsgemäß läuft)
- Robustheit („gutes“ Verhalten in unbekannten Systemzuständen)
- Effizienz (geringer Ressourcenverbrauch)
- Usability (Maß der Bedienbarkeit, Nutzerkontrolle/-freiheit, Ähnlichkeit zur Welt)
- Wartbarkeit (wie einfach es ist das System zu ändern / anzupassen nach Release)
- Verifizierbarkeit (Einfachheit der Überprüfung gewünschter Eigenschaften)
- Verständlichkeit (Einfachheit des Verstehens für interne und externe Nutzer)
- Produktivität (wie produktiv sind Entwickler für Teile des Systems?)
- Pünktlichkeit (pünktliche Auslieferung)
- Transparenz (Einsehbarkeit des Projektzustands)

Qualität auf Implementierungsebene: Code Smells

- Jedes Symptom im Quelltext, das auf größeres Problem (z.B. Antipattern) hinweist
- Nicht jeder Code Smell ist schlechter Code; bei jedem Code Smell überprüfen, ob er wirklich behoben werden muss
- Typische Code Smells:
 - Duplizierte Code
 - Lange Methode/Klasse
 - High coupling/low cohesion
 - Schrotkugeln (shotgun surgery)

Qualität auf Implementierungsebene: Antipatterns

- Siehe Vorlesung Design Patterns
- Beispiele:
 - The Blob (Gottklasse)
 - Action at a distance: Unerwartete Interaktion zwischen eigentlich getrennten Modulen
 - Reihenfolge: Wenn Methoden in einer Klasse in bestimmter Reihenfolge aufgerufen werden müssen
 - Zirkuläre Abhängigkeit: Unnötige Abhängigkeiten zwischen Modulen
- Mehr Beispiele: <http://c2.com/cgi/wiki?AntiPatternsCatalog>

Was Sie mitgenommen haben sollten:

- Anwendung:
 - [Modell/Quelltext]
 - Bewerten Sie anhand der 5 Kriterien
 - Bewerten Sie anhand der 5 Regeln
 - Bewerten Sie anhand der vorgestellten Elemente des GRASP-Patterns
- Wissen:
 - Was ist modular decomposability? Erklären Sie an einem Beispiel
 - [analog die restlichen Kriterien, Regeln, und Prinzipien]

Literatur

- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997 [Chapter 3, 4]
- *Software Engineering*, I. Sommerville, 7th Edn., 2004.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996

Softwaretechnik

Testen



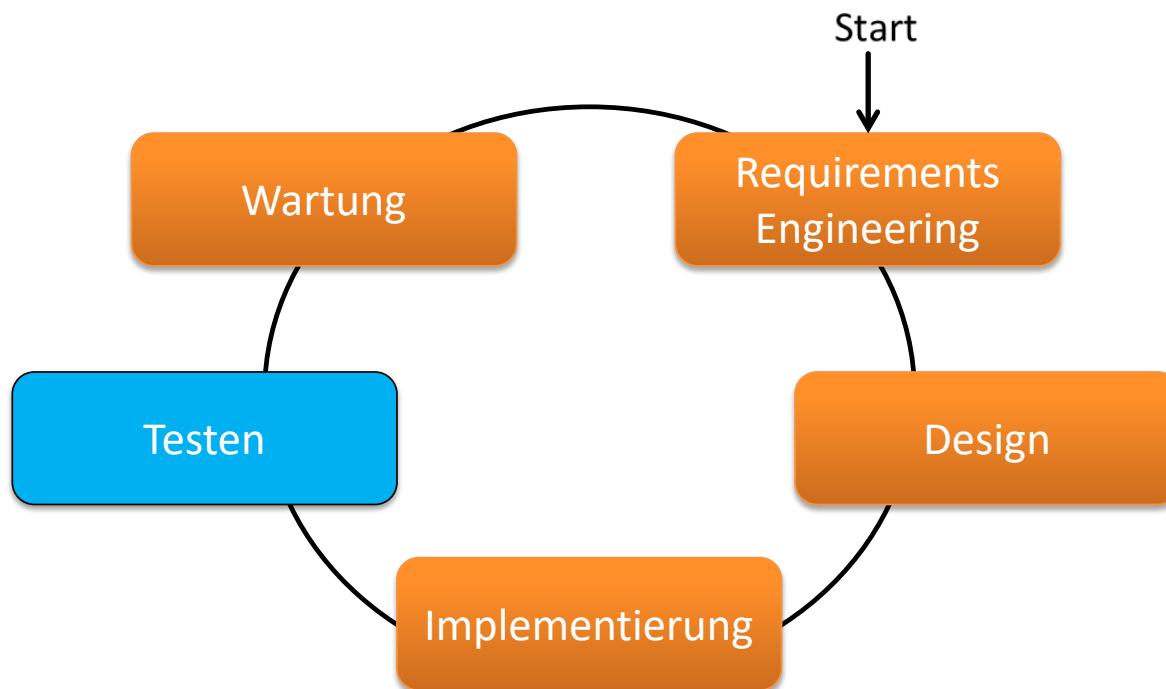
**SOFTWARE
SYSTEME**

Prof. Dr.-Ing. Norbert Siegmund
Software Systems

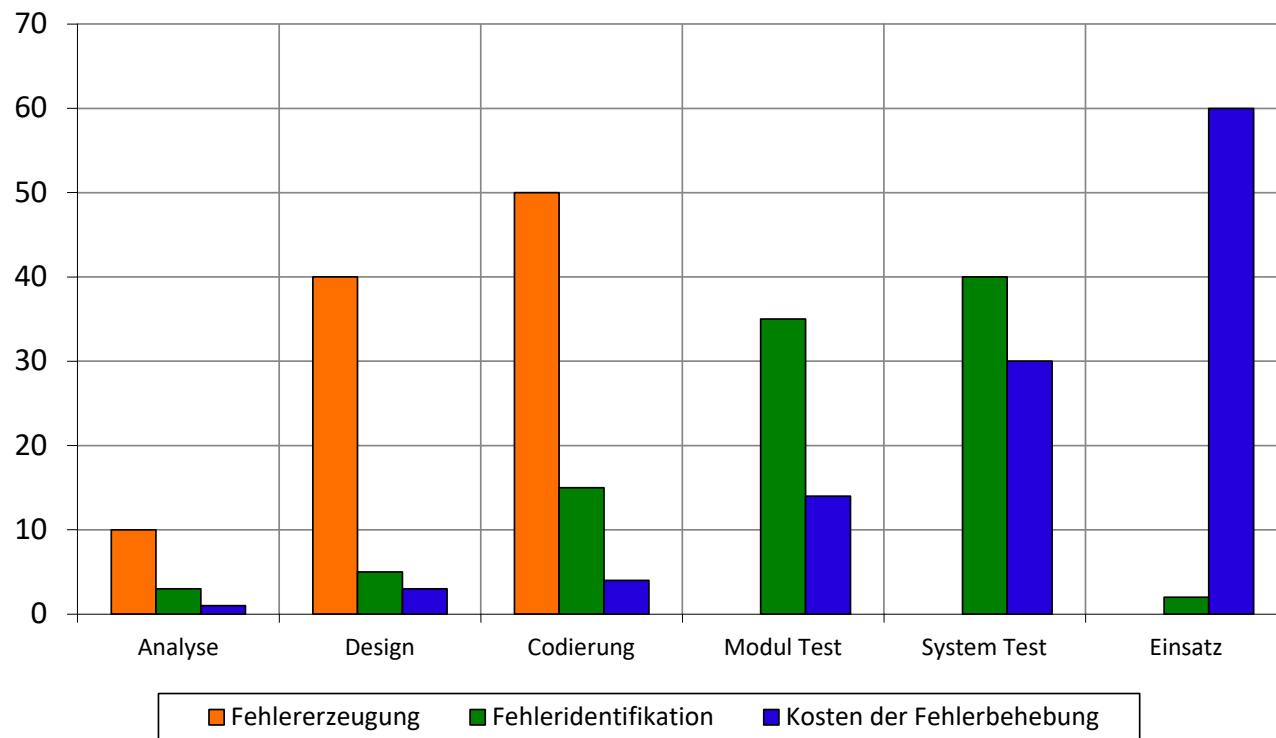


UNIVERSITÄT
LEIPZIG

Einordnung



Kosten für die Behebung von Fehlern



Lernziele

- Notwendigkeit von Testen und Code Reviews verstehen
- Verschiedene Arten von Testen, Verifikation und Code Reviews kennen lernen

Ziele des Testings

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"

(Edsger Dijkstra, The Humble Programmer, ACM Turing lecture, 1972)

- Ziel von Testen:
 - Fehler finden
 - Vertrauen in Software herstellen
 - Sicherstellen, dass das Programm nicht abstürzt
 - Regressionstest: keine neuen Fehler einführen bei neuen commits
 - Sicherstellen, dass die Anforderungen erfüllt sind
 - Sicherstellen, dass es keine Seiteneffekte gibt
- Ein erfolgreicher Test findet Fehler

Herausforderungen

- Man muss annehmen, dass ein Programm fehlerhaft ist; nicht, dass es korrekt ist^{*1}
- Entgegen jeder anderen Software-Entwicklungsaktivität (Fehler finden vs. Fehler vermeiden)
- Testen ist teuer
- Effektivität von Tests schwer zu messen
- Unvollständige, nicht-formalisierte und sich ändernde Anforderungen
- Integrationstest zwischen verschiedenen Produkten
- Steigende Anzahl von Versionen
- Patching nightmare



*1: Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. 1978.

Von Microsoft Office EULA...

11. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, **IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) **ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT**, THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.



Von GPL

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. **THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.**

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING **WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES **ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**



Mittl. Anzahl von Fehlern

- Industrie:
 - 30-85 Fehler per 1000 LOC (vor dem Testen);
 - 0,5-3 Fehler per 1000 LOC werden nicht erkannt vor Auslieferung.
 - 1% der Software-Fehler rufen 50% der Crashes hervor
- Snapshot von Mozilla's Bugzilla Bug Datenbank (ca. 2015)
 - Gesamte Historie von Mozilla; alle Produkte und Versionen
 - 60.866 offene Bug Reports
 - 109.756 zusätzliche Reports markiert als Duplikate
- Snapshot von Mozilla's Talkback Crash Reporter
 - Firefox 2.0.0.4 der letzten 10 Jahre
 - 101.812 eindeutige Nutzer
 - 183.066 Crash Reports
 - 6.736.697 Stunden von user-driven “testing”

Arten von Fehlern

<i>Fehlerklasse</i>	<i>Beschreibung</i>
<i>Transient</i>	Tritt nur bei <i>bestimmten Eingaben</i> auf
<i>Permanent</i>	Tritt bei <i>allen Eingaben</i> auf
<i>Recoverable</i>	System erholt sich <i>ohne Intervention eines Nutzers</i>
<i>Unrecoverable</i>	<i>Nutzerintervention</i> ist <i>benötigt</i> zur Wiederherstellung des Systems
<i>Non-corrupting</i>	Fehler korrumptiert <i>nicht</i> die Daten
<i>Corrupting</i>	Fehler <i>korrumptiert</i> die Daten

Fehlervermeidung

Fehlervermeidung ist abhängig von:

1. Einer genauen *Systemspezifikation* (siehe Requirements Engineering)
2. Software Design basierend auf *information hiding and encapsulation* (siehe SW-Qualität vorlesung + Design Patterns)
3. Extensive *Validierungsreviews* während des Entwicklungsprozesses
4. Eine organisatorische *Philosophie von Qualität*, welche den Softwareprozess prägt
5. Geplante Phasen von *System Testen und Verifikation*, um Fehler zu entdecken und die Zuverlässigkeit zu ermitteln

Häufige Software -Fehler

Verschiedene Features von Programmiersprachen und Systemen sind häufige Quellen von Fehlern:

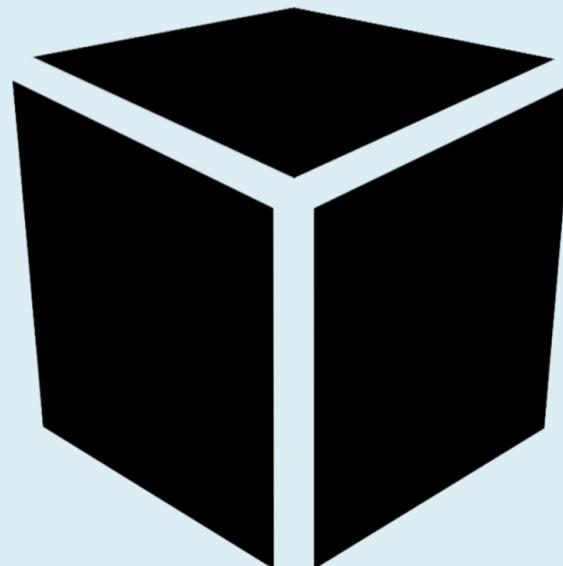
- **Goto Statements** und anderen unstrukturierte Programmierkonstrukte machen Programme *schwer zu verstehen und zu modifizieren*.
 - Verwendet nur strukturierte Programmierkonstrukte
- **Gleitkommazahlen** sind *inhärent ungenau* und können zu fälschlichen Vergleichen führen.
 - Ganzzahlen sind sicherer für exakte Vergleiche
- **Zeiger** sind gefährlich, durch *Aliasing* und dem Risiko den *Speicher zu korrumpern*
- **Parallelisierung** ist gefährlich, da *zeitliche Unterschiede* einen Einfluss auf das Programmverhalten haben können, die *schwer vorhersagbar* sind.
 - Minimiere inter-Process Abhängigkeiten
- **Rekursion** kann zu *verworrener Logik* führen und den Stack-Speicher überladen.
 - Verwende Rekursion in eine disziplinierten und kontrollierten Weise
- **Interrupts** erzwingen den Transfer der Kontrolle *unabhängig vom derzeitigen Kontext* und können daher zum Abbruch / Unterbrechung kritischer Operationen führen.
 - Minimiere die Verwendung von Interrupts; bevorzuge Exceptions



Strategien des Testens

- Black-Box-Tests
 - Ohne auf den Code zu schauen
 - Beziehung zwischen Eingaben und Ausgaben
- White-Box-Tests/Glass-Box-Tests
 - Code anschauen und systematisch versuchen, Fehler zu erzeugen
 - Ausführungspfade untersuchen

Black-Box Testen



Black-Box Testing

- Jede Funktionalität des Systems überprüfen
- Alles kann nicht mit vertretbarem Aufwand getestet werden
- Siehe Dreamliner (Ausfall aller Turbinen):
 - Entdeckung des Zählerüberlaufs erfordert zeitliche Simulation
 - 4 Zähler müssen über 248 Tage emuliert werden



Boeing 787 Dreamliners contain a potentially catastrophic software bug

Beware of integer overflow-like bug in aircraft's electrical system, FAA warns.

DAN GOODIN - 5/1/2015, 7:55 PM

152



A software vulnerability in Boeing's new 787 Dreamliner jet has the potential to cause pilots to lose control of the aircraft, possibly in mid-flight, Federal Aviation Administration officials warned airlines recently.

The bug—which is either a classic [integer overflow](#) or one very much resembling it—resides in one of the electrical systems responsible for generating power, according to [memo the FAA issued last week](#). The vulnerability, which Boeing reported to the FAA, is triggered when a generator has been running continuously for a little more than eight months. As a result, FAA officials have adopted a new airworthiness directive (AD) that airlines will be required to follow, at least until the underlying flaw is fixed.

"This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode," the memo stated. "This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane."

The memo went on to say that Dreamliners have four main GCUs associated with the engine mounted generators. If all of them were powered up at the same time, "after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase." Boeing is in the process of developing a GCU software upgrade that will remedy the unsafe condition. The new model plane previously experienced a [battery problem that caused a fire](#) while one aircraft was parked on a runway.

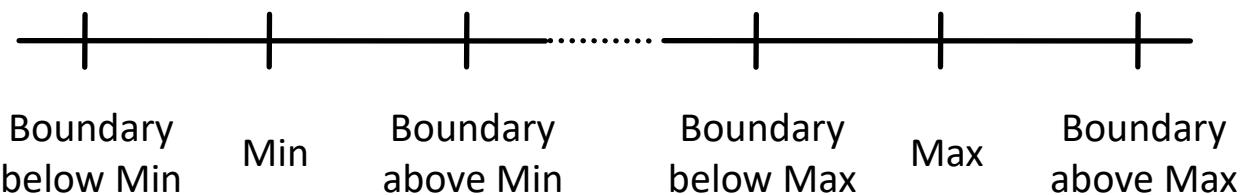
The memo doesn't provide additional details about the underlying software bug. Informed speculation suggests it's a [signed 32-bit integer overflow](#) that is triggered after 2^{31} centiseconds (i.e. 248.55 days) of continuous operation.

Äquivalenzklassen finden

- Nur Zahlen
 - Beispiel: System fragt nach Zahlen zwischen 100 und 999
 - Dann teste mit: <100; 100-999; >999
 - Tests haben auch nicht charakteristische/valide Werte!
- Nur Buchstaben
- Kombination aus Zahlen und Buchstaben
- Sonderzeichen? Umlaute?
- Äquivalenzklassen zu finden, ist oft nicht trivial

Grenzwerte Analysieren

- Eingaben an Grenzen machen oft Probleme
- Bsp: `ArrayIndexOutOfBoundsException()`



- Tests für folgende Werte:

$\frac{99 \quad 100 \quad 101}{\text{lower boundary}}$ $\frac{998 \quad 999 \quad 1000}{\text{upper boundary}}$

Erfahrung und Heuristik

- Bisherige Erfahrungen und Heuristiken nutzen
 - Sonderzeichen haben schon immer Probleme gemacht -> Sonderzeichen einschließen
 - Umlaute machen Probleme in anderen Sprachen -> Umlaute einschließen

Einfache Daten

- 3,14159265 ist schwerer manuell zu überprüfen als 2
 - Z.B., wenn Code etwas verdoppeln soll
-
- Daten sollten daher nachvollziehbar ausgewählt werden (z.B. im Kopf ausrechenbar sein)

Systematisches Vorgehen

- Funktionalitäten aus Anforderungen ableiten und Eingabe- und zugehörige Ausgabedaten bestimmen
- Äquivalenzklassen von Eingaben testen
- Daten an Intervallgrenzen testen
- Inkorrekte Eingaben testen
- Jede definierte Fehlermeldung erzeugen
- Kombination von Funktionalitäten testen (**fett+kursiv+Schriftgröße**)
- Seltene Fälle testen

Zusammenfassung Black-Box Testen

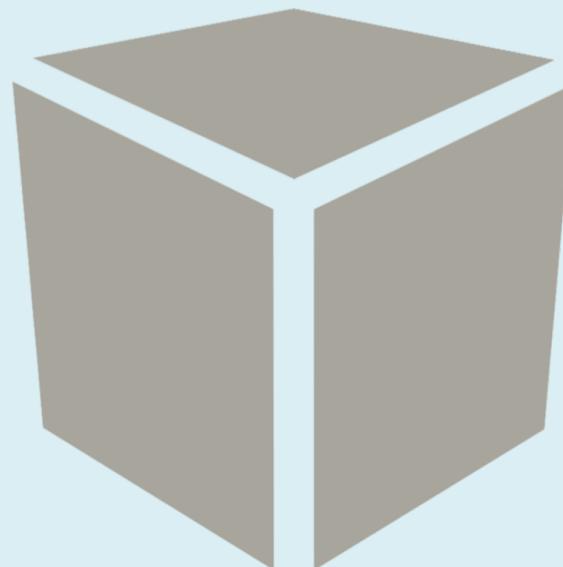
- Geeignet zum Finden von:
 - Inkorrekt oder fehlender Funktionalität (aus Spezifikation)
 - Schnittstellenfehler
 - Fehler in Datenstrukturen oder externen Zugriffen
 - Problemen von nicht-funktionalen Eigenschaften
 - Fehlern beim Ablauf von Prozessen
- Grenzen:
 - Spezifikation ist meist abstrakt und spiegelt nicht Implementierung wieder
 - Ein externer Zustand kann mehreren internen Zuständen entsprechen (wie Testen?)
 - Nicht alle Element einer Äquivalenzklasse werden auch im Code gleich behandelt (fehlende Äquivalenzklassen)

Weitere Limitierungen von Black-Box Tests

Können Sie weitere Gründe benennen warum Black-Box Tests allein unzureichend sind?

- Spezifikationen und Sonderfälle können vergessen / übersehen werden
- Fehler, die unabhängig von der Eingabe sind, können evtl. nicht entdeckt werden (z.B. bei parallel laufenden Programmen)
- Ausnahmefälle (wie z.B. Hardwareausfall) und deren Fehlerbehandlungen können oft nicht ausreichend getestet werden

White-Box / Glass-Box Testen



White-Box Testing

- Idee: Code anschauen und systematisch alle Anweisungen, Bedingungen und Pfade mindestens einmal ausführen
- Ideal: alle Ausführungspfade testen (aber, nicht praktikabel)
- Stattdessen: Teste z.B. jedes Statement mind. 1 mal
- Beispiel:

```
if (x > 5) {  
    System.out.println("hello");  
} else {  
    System.out.println("bye");  
}
```

*Es gibt zwei mögliche Pfade durch den Code,
 $x > 5$ und $x \leq 5$.
Ziele darauf ab, jeden auszuführen.*

Herausforderungen

- Oft durch Entwickler selbst durchgeführt
- Welcher Entwicklerin geht schon gern davon aus, dass ihr Programm Fehler hat?
- Man testet oft das Verhalten, was man sowieso bereits im Kopf / programmiert hat
 - Daher: Andere Personen wichtig zum Testen, um alternative Herangehensweise / Benutzungen / etc vom System zu testen

Systematisches Vorgehen

- Wurde jede Funktion mindestens einmal ausgeführt?
- Wurde jede Anweisung mindestens einmal ausgeführt?
- Wurde jeder Zweig von if/case-Anweisungen ausgeführt?
- Wurde überprüft, ob jeder boolsche (Sub-) Ausdruck wahr und falsch werden kann?
- Ausführungspfade:
 - Wurde jeder mögliche Pfad ausgeführt?
 - Problem: Unendlich viele Pfade
 - Typischerweise Kompromiss: 0-1-viele Ausführungen
- Datenstrukturen: Jeder mögliche Zustand

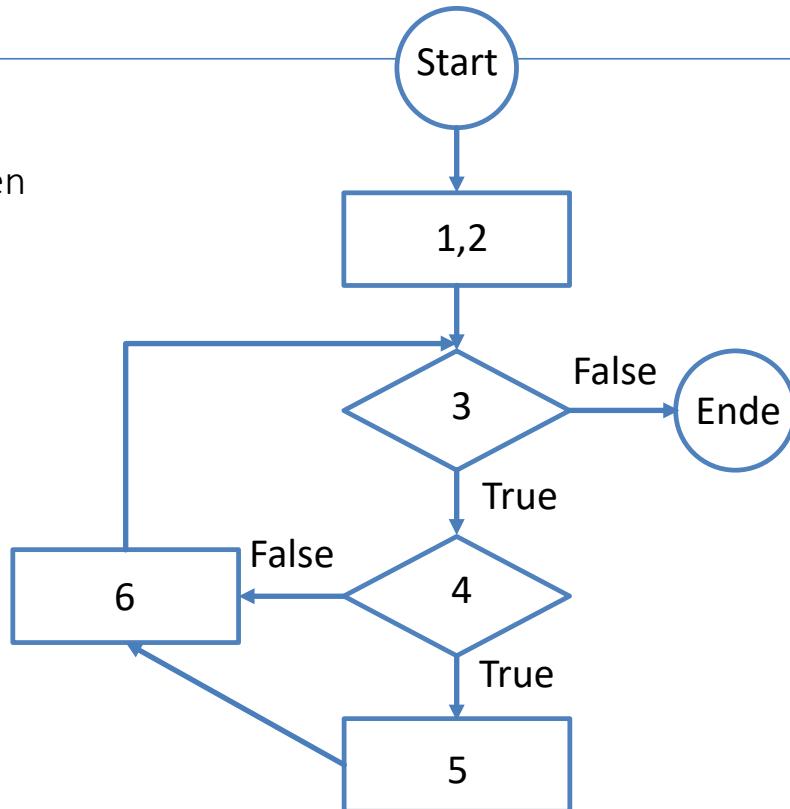
Arten

- Kontrollflussorientiert
 - Anweisungsüberdeckung (C0)
 - Kantenüberdeckung (C1)
 - Bedingungsüberdeckung (C2, C3)
 - Pfadüberdeckung (C4)
- Datenflussorientiert
 - Nicht behandelt

Kontrollflussgraph

- $G = (V, E)$ wobei
 - V ist eine Menge von Basisblöcken
 - E ist eine Menge von gerichteten Kontrollflüssen
- Beispiel:

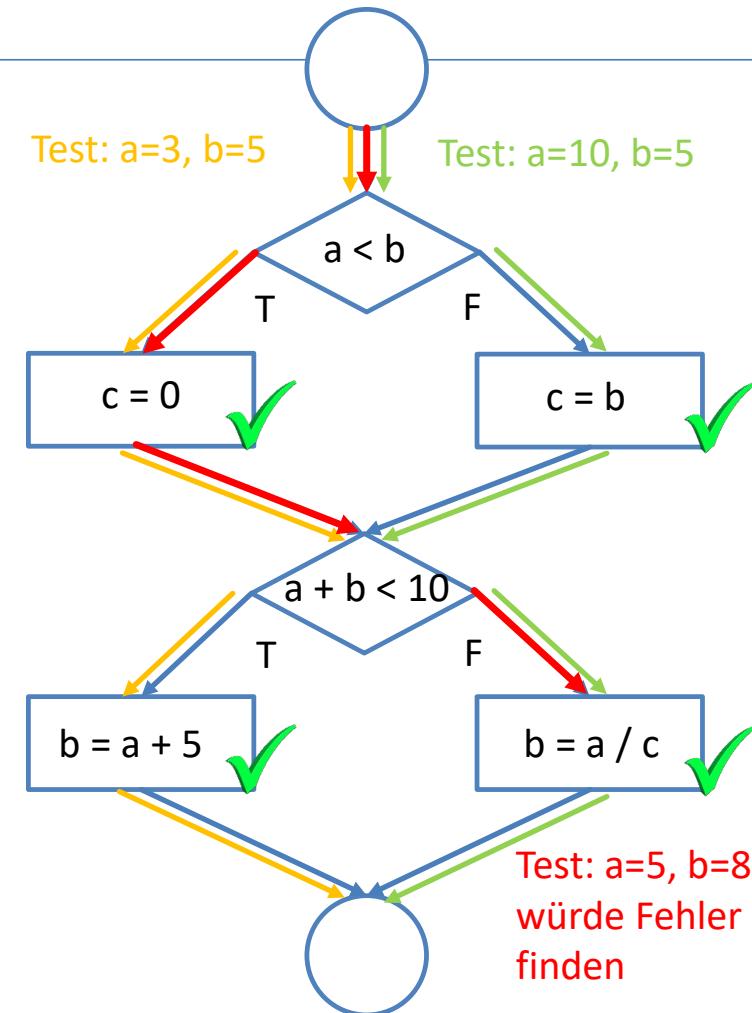
```
1. a = Read(b)
2. c = 0
3. while (a > 1)
4.   If (a^2 > c)
5.     c = c + a
6.   a = a - 2
```



Graph = Menge aller möglichen Ausführungen eines Programmes.
Pfad = Eine konkrete Ausführung eines Programmes.

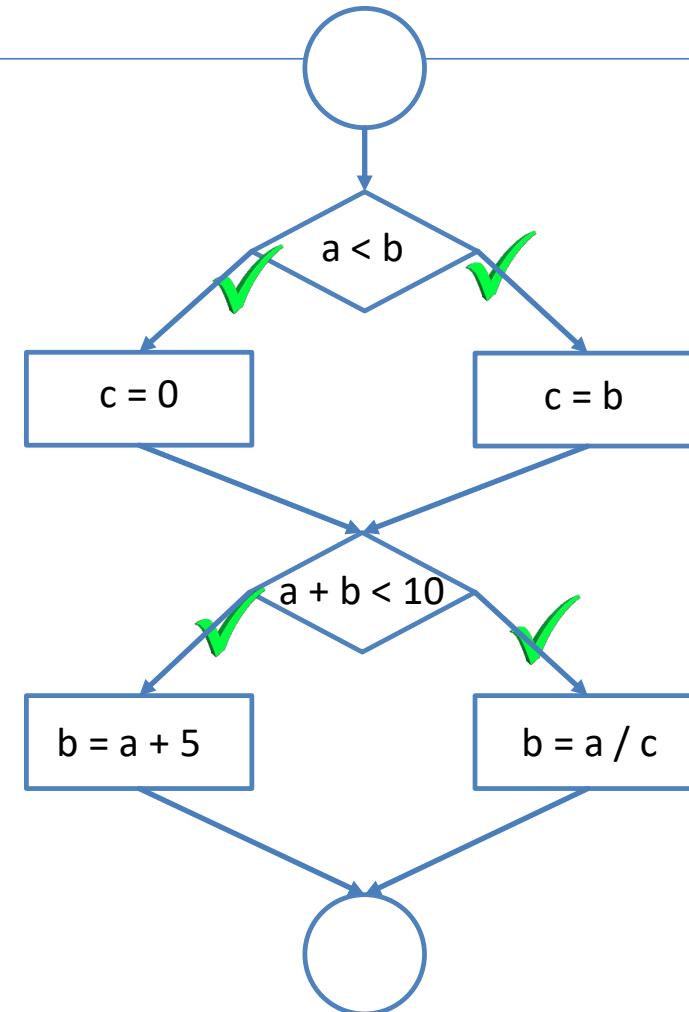
Anweisungsüberdeckungsverfahren (C0- Test)

- Wähle Testmenge so, dass alle Anweisungen des Testobjektes mind. einmal ausgeführt wurden
- Probleme:
 - Nicht alle Wege müssen geprüft werden
 - Schleifen werden unzureichend geprüft



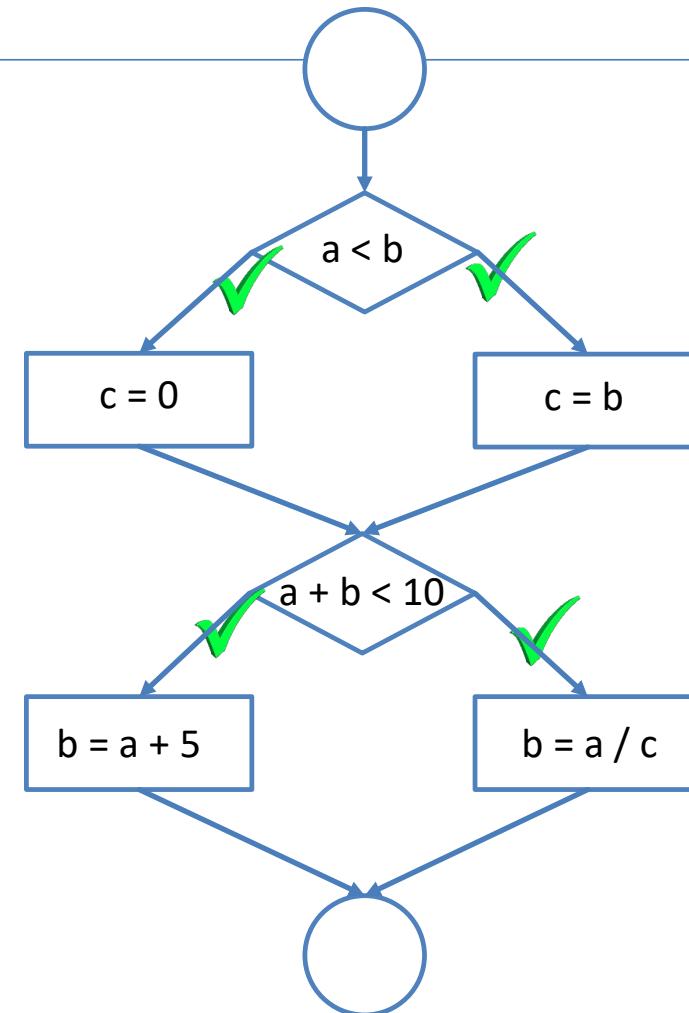
Zweigüberdeckungstest (C1 - Test)

- Wähle Testmenge so, dass alle Kanten des Testobjektes mind. einmal ausgeführt wurden
 - Inkludiert C0 Test
 - Minimaler Test
 - Auch Zweige ohne Code werden ausgeführt
- Probleme:
 - Nicht alle Wege müssen geprüft werden
 - Schleifen werden unzureichend geprüft



Bedingungsüberdeckungstest (C2 / C3 Test)

- Wähle Testmenge so, dass alle Teilbedingungen des Testobjektes überdeckt werden
 - Alle Teilformeln von Bedingungen auf true und false prüfen (C2)
 - Alle Wahrheitskombinationen atomarer Teilformeln auf true und false prüfen (C3)
- Probleme:
 - C3 führt zu exp. Anstieg der Testfälle
 - Schleifen werden unzureichend geprüft

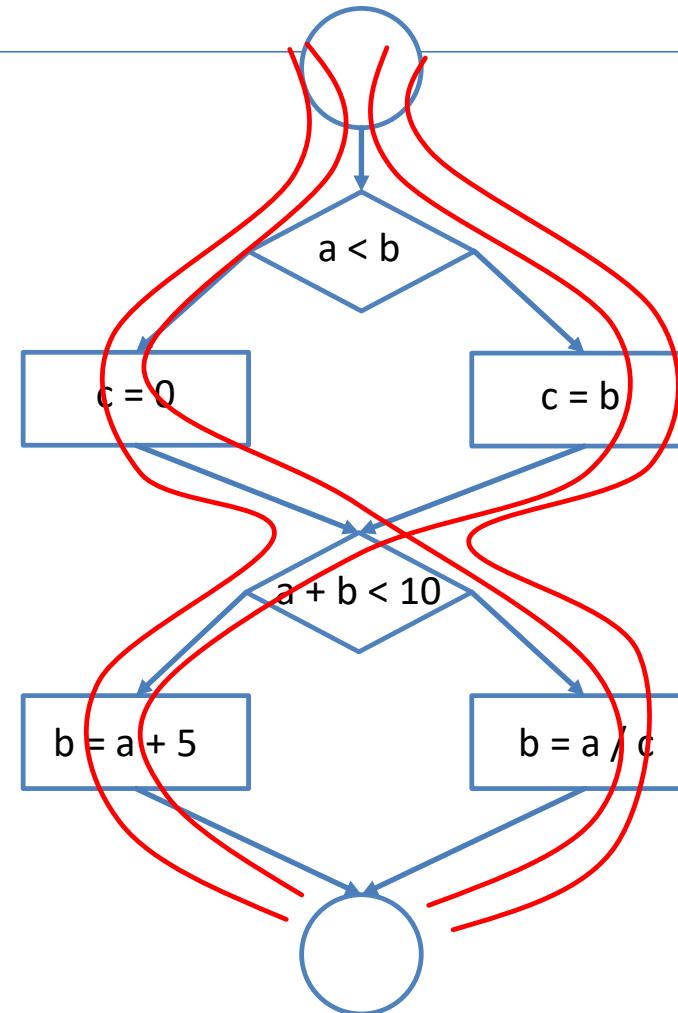


Pfadüberdeckungsverfahren

- Wähle eine Testmenge so, dass alle Pfade vom Eingangs- bis zum Ausgangsknoten durchlaufen werden
 - Probleme bei Schleifen (insb. While)
 - Anz. der Wiederholungen in Schleifen wird meist eingeschränkt
- Motivation:
 - Logische Fehler und inkorrekte Annahmen sind umgekehrt proportional zur Wahrscheinlichkeit der Ausführung des Pfades
 - Entwickler haben häufig fälschliche Annahme, dass ein bestimmter Pfad nicht ausgeführt wird
 - Tippfehler sind zufällig; somit wahrscheinlicher in ungetesteten Pfaden

Beispiel

- 2 Bedingungen = $2^2 = 4$ Pfade
- Was ist mit Schleifen?



Probleme der Pfadüberdeckung

- Sehr schnell zu viele Pfade
 - Sollte nur bei kritischen Modulen verwendet werden
 - Spezialfälle für Schleifen
 - Kein Durchlauf
 - 1 Durchlauf
 - 2 Durchläufe
 - M Durchläufe bei $m < n : n = \text{max. Anzahl Durchläufe}$
 - $N-1, n, n+1$ Durchläufe
- Zyklomatische Komplexität als Maß der Pfadüberdeckung

Grenzen und Zusammenfasung

- 100%-Testabdeckung praktisch nicht möglich, besonders bei Ausführungspfade und Fallunterscheidungen
- White-Box Tests ergänzen Black-Box Tests
- Viele Tools, die Testabdeckung messen und visualisieren (<https://about.codecov.io/blog/the-best-code-coverage-tools-by-programming-language/>)
- Trotz systematischem Vorgehen kann man nie sicher sein, dass der Quelltext fehlerfrei ist
 - Dijkstra hat nach 40 Jahren immer noch Recht

Konkrete Testverfahren

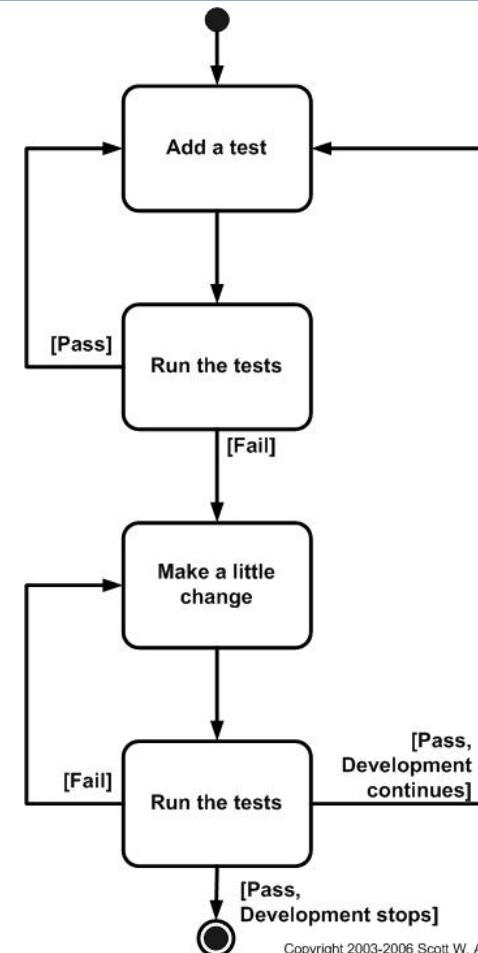
Testverfahren

1. Test-Driven Development
2. Unit-Tests
3. Integrations-Tests
4. System-Tests
5. Acceptance-Tests



Test-Driven Development

- Evolutionärer Entwicklungsansatz, welcher die folgenden Methoden kombiniert:
 - Test-first design: zuerst Test schreiben bevor der Code geschrieben wird, welcher getestet werden soll
 - Refactoring



Test-Driven Development II

- Umgekehrter Entwicklungsansatz
 - Ist das existierende Design das best-mögliche Design, um ein Feature zu implementieren?
 - Falls ja, setze mit nächstem Feature fort
 - Falls nein, refaktorisiere es lokal, so dass das neue Feature so einfach wie möglich hinzugefügt werden kann
 - Ergebnis: kontinuierliche Verbesserung der Qualität des Designs

Prinzipien von TDD

- Schreibe Test-Code vor dem funktionellen Code
- Sehr kleine Schritte --- ein Test und eine kleine Einheit korrespondierenden Codes zur gleichen Zeit
- Programmierer verweigern das Hinzufügen auch nur einer einzelnen Zeile Code solange dafür kein Test existiert
- Sobald ein Test vorhanden ist, setzt der Programmierer alles daran, dass die Test Suite erfolgreich durchläuft

“If it's worth building, it's worth testing.

If it's not worth testing, why are you wasting your time working on it?”

—Scott W. Ambler



Unit Tests

- Ziel: Individuelle Module (meist auf Klassenebene) und Funktionen werden getestet, um deren korrekte Funktionsweise sicherzustellen
- Typischerweise automatisiert
- Oft durch Entwickler spezifiziert
- Fokus auf eine Funktion/Methode/Modul
- Stubs/Mock-Objekte, wenn dabei andere Module aufgerufen werden
 - Stubs/Mocks emulieren anderen Objekte/Methoden im Programm, welche notwendig sind, um das eigentliche Modul zu testen

JUnit – Unit Testing Framework

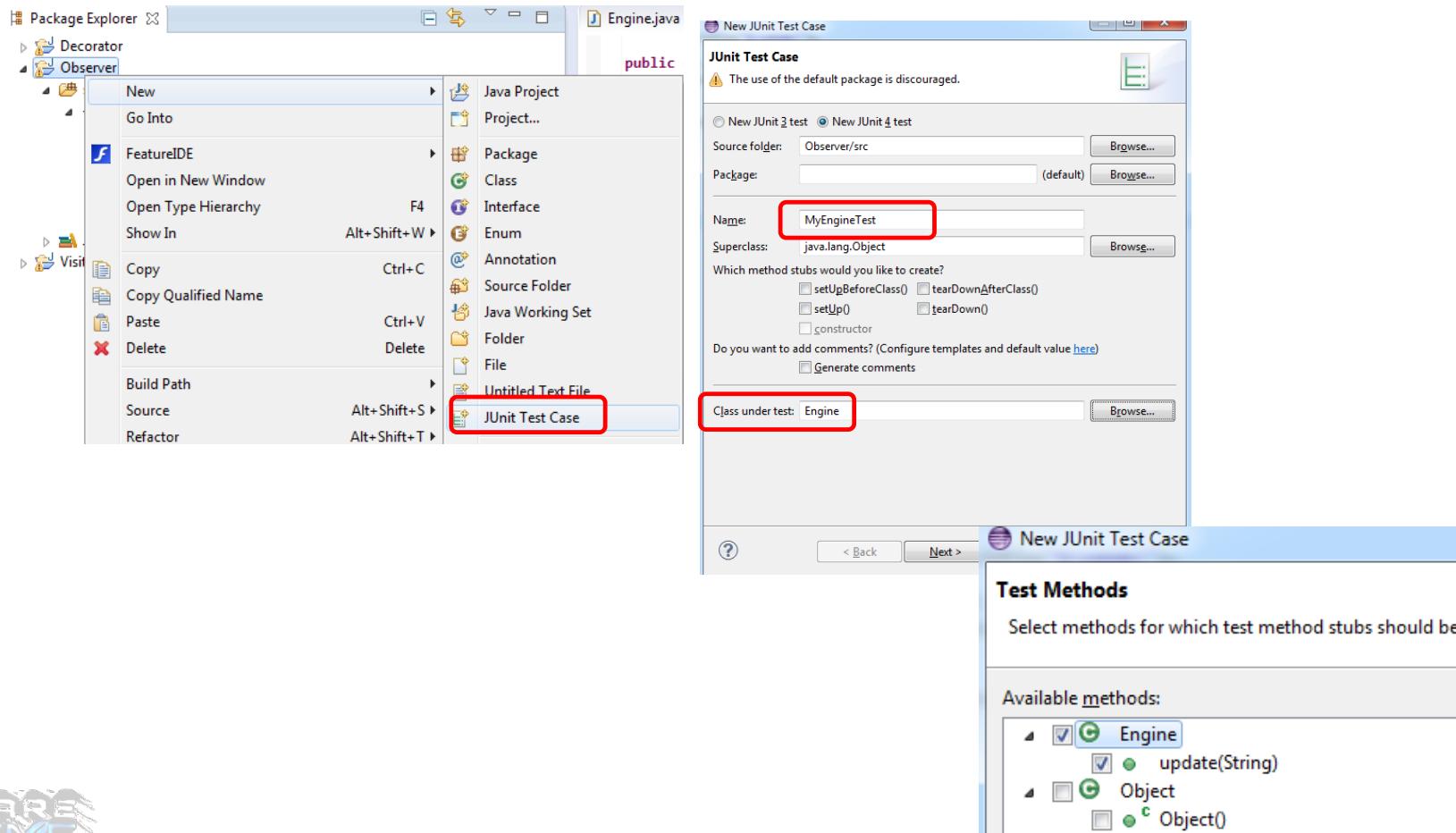
- JUnit Test ist eine Methode in einer Klasse, die nur für das Testen verwendet wird
 - Annotationen markieren Methoden, die einen Test spezifizieren (@org.junit.Test)
- Innerhalb der Methode, wird eine Methode des Frameworks verwendet, welche das erwartete Ergebnis gegen das des ausführten Codes vergleicht

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();
    // Tests
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

JUnit – Methoden und Annotationen

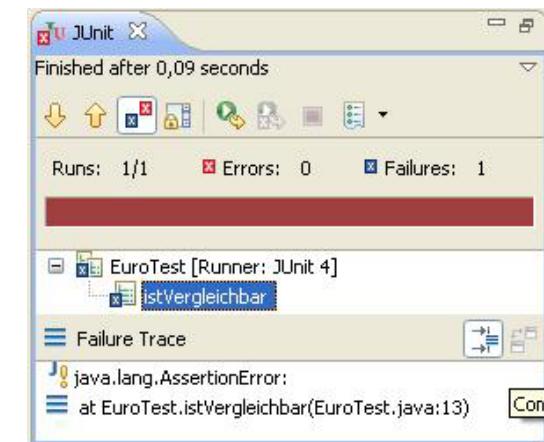
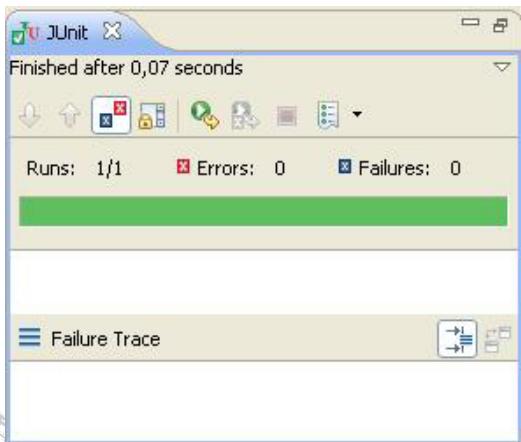
Statement	Description	Annotation	Description
<code>fail(message)</code>	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.		
<code>assertTrue([message,] boolean condition)</code>	Checks that the boolean condition is true.		
<code>assertFalse([message,] boolean condition)</code>	Checks that the boolean condition is false.		
<code>assertEquals([message,] expected, actual)</code>	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.		
<code>assertEquals([message,] expected, actual, tolerance)</code>	Test that float or double values match. The tolerance is the number of decimals which must be the same.	<code>@Test public void method()</code>	The <code>@Test</code> annotation identifies a method as a test method.
<code>assertNull([message,] object)</code>	Checks that the object is null.	<code>@Test(expected = Exception.class)</code>	Fails if the method does not throw the named exception.
<code>assertNotNull([message,] object)</code>	Checks that the object is not null.	<code>@Test(timeout=100)</code>	Fails if the method takes longer than 100 milliseconds.
<code>assertSame([message,] expected, actual)</code>	Checks that both variables refer to the same object.	<code>@Before public void method()</code>	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
<code>assertNotSame([message,] expected, actual)</code>	Checks that both variables refer to different objects.	<code>@After public void method()</code>	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
		<code>@BeforeClass public static void method()</code>	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as <code>static</code> to work with JUnit.
		<code>@AfterClass public static void method()</code>	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as <code>static</code> to work with JUnit.
		<code>@Ignore or @Ignore("Why disabled")</code>	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

JUnit in Eclipse



JUnit – Beispiel

```
+ import static org.junit.Assert.*;  
  
public class MyEngineTest {  
  
    @Test  
    public void testUpdate() {  
        Engine eng = new Engine();  
        assertTrue("Test ob Engine Geräusche macht beim fahren", "Brummm!".equals(eng.update("fahrt")));  
        assertTrue("Test ob Engine Geräusche macht beim fahren", "zzz".equals(eng.update("parkt")));  
    }  
}
```



2. Integrations-Tests (IT)

- Module werden kombiniert und als Gruppe getestet
- Nach Unit-Tests, vor System-Tests
- Zweck: Erfüllen Module im Zusammenspiel funktionale und nicht-funktionale Anforderungen?
- IT basiert auf Black-Box-Tests
- IT oft als Top-down IT or Bottom-up IT

IT: Top-Down

-
1. Kontrollflussmodul führt Test aus, alle untergeordneten Komponenten werden durch Stubs ausgetauscht
 2. Untegordnete Komponenten werden durch eigentliche Komponenten Schritt für Schritt getauscht
 3. Nach jedem Tausch wird getestet

IT: Bottom-Up

1. Low-level-Module, die bestimmte Funktion ausführen, werden zu Cluster kombiniert
2. Komponente, die Tests koordiniert
3. Cluster wird getestet
4. Getestet Cluster werden kombiniert und wieder getestet, bis höchste Ebene erreicht ist

Aufgabe

- Was ist besser, Top-Down oder Bottom-Up Tests?
- Top-Down:
 - Gesamtsystem im Blick
 - Ganze Use-cases testen
 - Prozess (also Ablauf und Reihenfolge mehrerer Aktionen) kann getestet werden
- Bottom-Up:
 - Lokalisierung von Fehlern ist einfach
 - Fokus auf komplizierte Kombinationen von Komponenten

3. System-Tests

- Black-Box-Test des kompletten Systems
- Konzentration auf:
 - Fehler, die aus Interaktionen zwischen Sub-Systemen herkommen
 - Validierung, dass das gesamte System funktioniert und nicht-funktionale Anforderungen erfüllt
- Orientierung oft an use cases
- Meistens durch separates Test-Team
- Viele verschiedene Arten von System-Tests
 - GUI, Usability, Performance, Barrierefreiheit, Stresstests,...

Regressions-Test

- Idee: Nach *jeder* Änderungen werden *alle* Testfälle wieder ausgeführt
- *Sicherstellung*, dass alles, was *vor* der *Änderung* funktioniert hat, auch *nach* der Änderung weiterhin funktioniert
- Tests müssen *deterministisch* und *wiederholbar* sein
- Tests sollten gesamte Funktionalität umfassen
 - Jedes Interface
 - Alle Grenzsituationen /-fälle
 - Jedes Feature
 - Jede Zeile Code
 - Alles was irgendwie falsch gehen kann

Nightly/Daily Builds

- Release eines großen Projekts wird zu fest definiertem Zeitpunkt erstellt
- Zeitpunkt: Wenn keine Änderungen am Code zu erwarten sind
- Gefundener Fehler ist großes Problem:
 - Verantwortlicher Entwickler muss ggfs. nachts das Problem beheben
- Nach einem Build oft Regressions-Tests

“Treat the daily build as the heartbeat of the project. If there is no heartbeat, the project is dead.” - Jim McCarthy (<http://www.mccarthyshow.com/>)

4. Acceptance-Tests

- Erfüllt das System alle spezifizierten Anforderungen?
- Funktionstest, die der Kunde ausführt, um Qualität zu bewerten
- Echte statt simulierte Daten
- Alpha-Tests:
 - White-Box-Tests durch Entwickler
 - Danach Black-Box-Test durch andere Teams
- Beta-Tests:
 - Endnutzer testen System

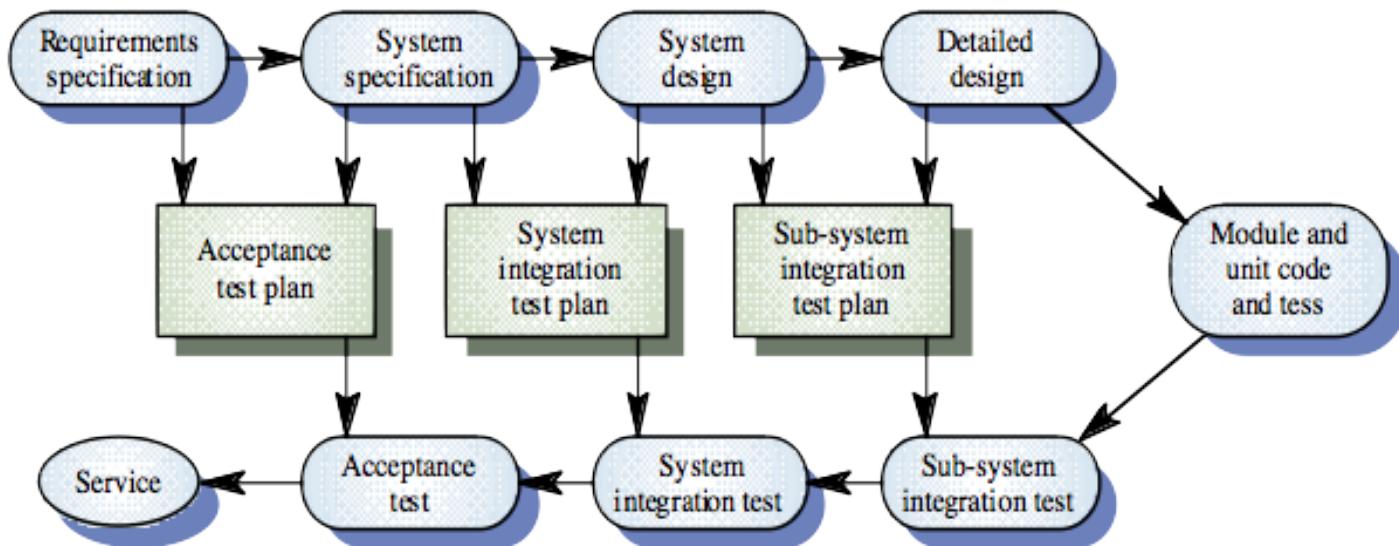
Wann findet man die meisten Fehler?

Testing technique	Rate
Unit test	30%
Integration test	35%
System test	40%
Beta test	bis zu 75%

- Rate:
 - Anzahl gefundener Fehler beim Anwenden einer Technik
 - Gefundene Fehler pro Strategie überlappen sich teilweise

Wie spielt alles zusammen?

- Testplan muss erstellt werden, *sobald die Anforderungen formuliert* sind und *ständig verfeinert* werden



- Plan sollte *regulär* überarbeitet und Tests *wiederholt* und *erweitert* werden

Design für Testen

- Stelle sicher, dass Komponenten in Isolation getestet werden können
 - Minimiere Abhängigkeiten zu anderen Komponenten
 - Biete Konstruktoren an, um Objekte für das Testen zu erstellen
- Design Techniken existieren für verbesserte Testbarkeit
 - Benutze Interfaces, um Mock Objekte oder Stubs zu nutzen

Nochmal: THE limitation of testing

- *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
 - E.W. Dijkstra
- Keine Fehler bei Tests kann bedeuten:
 - Es gibt keine Fehler
 - Testfälle sind unvollständig
- Lösung: Verifikation von Software (nicht Teil der Vorlesung)

Software Verifikation und Modell-basiertes Testen

Was bedeutet fehlerfrei?

- Programm kann kompiliert werden
 - Main-Methode wirft keine Exception
 - Generell keine NullpointerException, ClassCastException
 - Keine Zugriffe auf nicht-initialisierten Speicher
-
- Sagt alles noch nichts über „richtige“ Ergebnisse aus
→ Spezifikation

Verifikation und Validation

Verifikation:

- Bauen wir das das *Produkt richtig*?
 - D.h., entspricht es der Spezifikation?

Validation:

- Bauen wir das *richtige Produkt*?
 - D.h., entspricht es den Nutzererwartungen?

Statische Verifikation

Programminspektionen:

- Kleine Team prüfen systematisch den Code
- Checklisten sind oft erforderlich
 - z.B., "Sind alle Invarianten, Vor- und Nachbedingungen überprüft?" ...

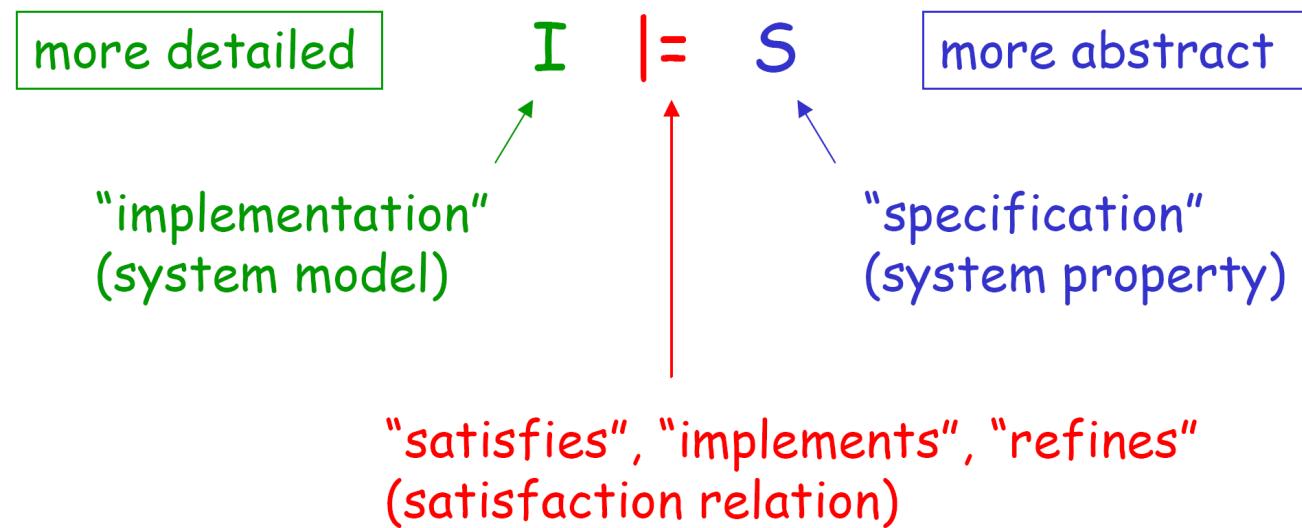
Statische Programmanalyse:

- Komplementiert Compiler-Checks für gewöhnliche Fehler
 - z.B., Variable benutzt vor Initialisierung

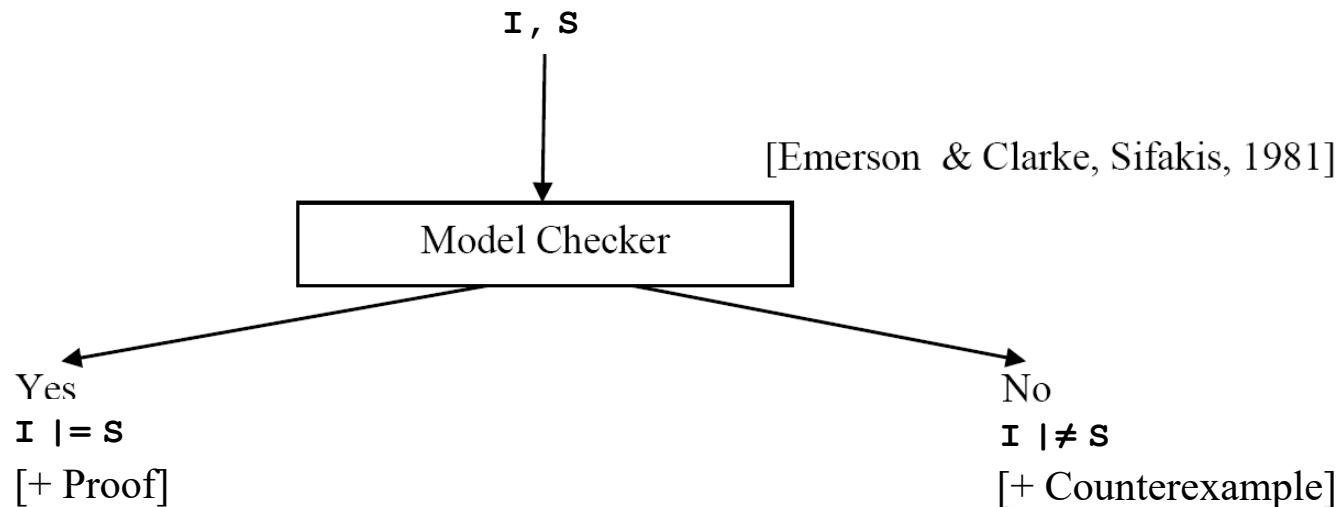
Mathematisch-basierte Verifikation:

- Verwendung von mathematischen Herleitung zur Demonstration, dass das Programm die Spezifikationen erfüllt
 - z.B., dass Invarianten nicht verletzt werden, Schleifen terminieren, etc.
 - z.B., Model-checking Tools

Model Checking



Model Checking



Beispiel

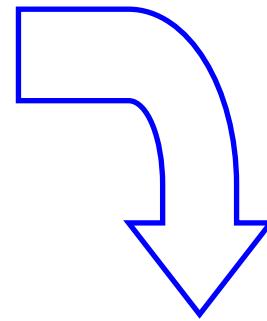
```
import java.util.Random;
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42); // (1)

        int a = random.nextInt(2);      // (2)
        System.out.println("a=" + a);

        //... lots of code here

        int b = random.nextInt(3);      // (3)
        System.out.println("b=" + b);

        int c = a/(b+a -2);           // (4)
        System.out.println("c=" + c);
    }
}
```



```
> java Rand
a=1
b=0
c=-1
>
```

Test Run

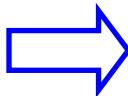
```
import java.util.Random;
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42); // (1)

        int a = random.nextInt(2);           // (2)
        System.out.println("a=" + a);

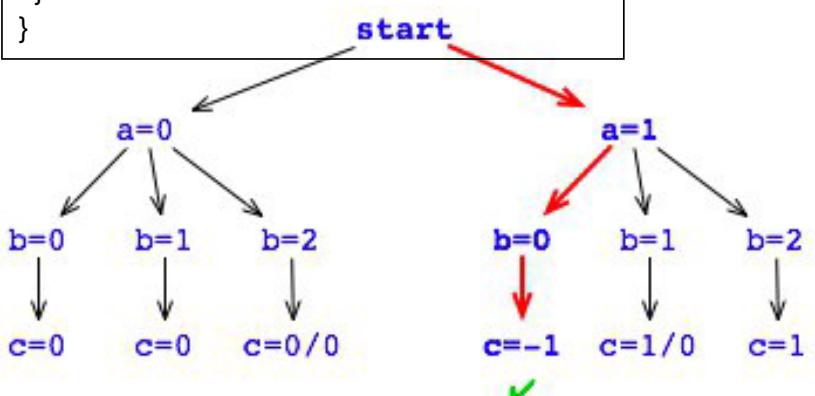
        //... lots of code here

        int b = random.nextInt(3);           // (3)
        System.out.println("b=" + b);

        int c = a/(b+a -2);                // (4)
        System.out.println("c=" + c);
    }
}
```



```
> java Rand
a=1
b=0
c=-1
>
```



- ① `Random random = new Random()`
- ② `int a = random.nextInt(2)`
- ③ `int b = random.nextInt(3)`
- ④ `int c = a/(b+a -2)`

Model Checking

```
import java.util.Random;
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42); // (1)

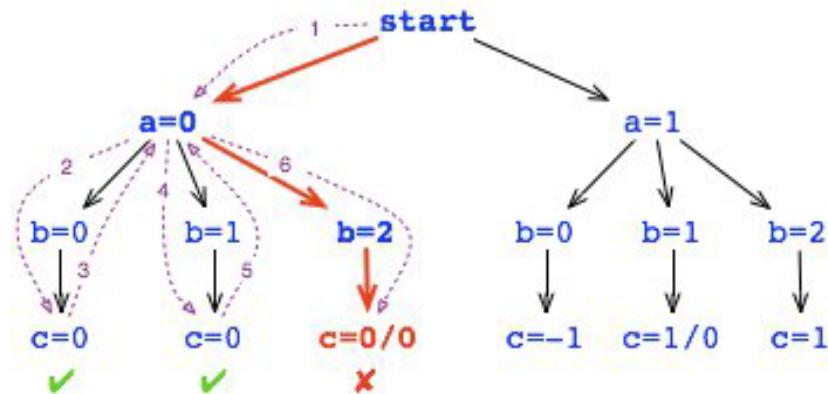
        int a = random.nextInt(2);           // (2)
        System.out.println("a=" + a);

        //... lots of code here

        int b = random.nextInt(3);           // (3)
        System.out.println("b=" + b);

        int c = a/(b+a -2);                // (4)
        System.out.println("c=" + c);
    }
}
```

Idee: Prüfe alle möglichen Werte von Ausdrücken



Informelle Spezifikation

Beispiel: JavaDoc

```
/**  
 * The method takes a given array of integers, sorts and returns the array.  
 *  
 * @param values a potentially unsorted array  
 * @return array sorted in ascending order  
 */  
public int[] sort(int[] values) {  
    for (int j = values.length; j > 1; j--) {  
        for (int i = 0; i < j - 1; i++) {  
            if (values[i] > values[i + 1]) {  
                int temp = values[i];  
                values[i] = values[i + 1];  
                values[i + 1] = temp;  
            }  
        }  
    }  
    return values;  
}
```

Problem 1: Nur durch manuelles Testen überprüfbar
Problem 2: Mehrdeutigkeiten

Formale Spezifikation

Beispiel: Java Modeling Language (JML)

```
/*@
@ requires values != null;
@ ensures (\forall int i; 0 < i && i < values.length; \result[i-1] <= \result[i]);
*/
public int[] sort(int[] values) {
    for (int j = values.length; j > 1; j--) {
        for (int i = 0; i < j - 1; i++) {
            if (values[i] > values[i + 1]) {
                int temp = values[i];
                values[i] = values[i + 1];
                values[i + 1] = temp;
            }
        }
    }
    return values;
}
```

- Runtime Assertions
- Deduktive Verifikation
- Statische Analysen

Runtime Assertions

```
public int[] sort(int[] values) {  
  
    assert values != null;  
  
    for (int j = values.length; j > 1; j--) {  
        for (int i = 0; i < j - 1; i++) {  
            if (values[i] > values[i + 1]) {  
                int temp = values[i];  
                values[i] = values[i + 1];  
                values[i + 1] = temp;  
            }  
        }  
  
        for (int i = 1; i < values.length; i++)  
            assert values[i-1] <= values[i];  
  
    }  
  
    return values;  
}
```

Vorteile

- Präzise Fehlerlokalisierung (Blame assignment)
- Keine False-Positives
- Automatisierbar
- Orthogonal zum Testen

Nachteile

- Findet nicht alle Fehler
- Ausbremsen des Systems

Deduktive Verifikation

```
public int[] sort(int[] values) {  
    //@ assert values != null;  
    for (int j = values.length; j > 1; j--) {  
        //@ assert (\forall int k; j-1 < k && k  
        < values.length; values[k-1] <= values[k]);  
        for (int i = 0; i < j - 1; i++) {  
            if (values[i] > values[i + 1]) {  
                int temp = values[i];  
                values[i] = values[i + 1];  
                values[i + 1] = temp;  
            }  
        }  
        //@ assert (\forall int k; j-2 < k && k  
        < values.length; values[k-1] <= values[k]);  
    }  
    //@ assert (\forall int k; 0 < k && k <  
    values.length; values[k-1] <= values[k]);  
    return values;  
}
```

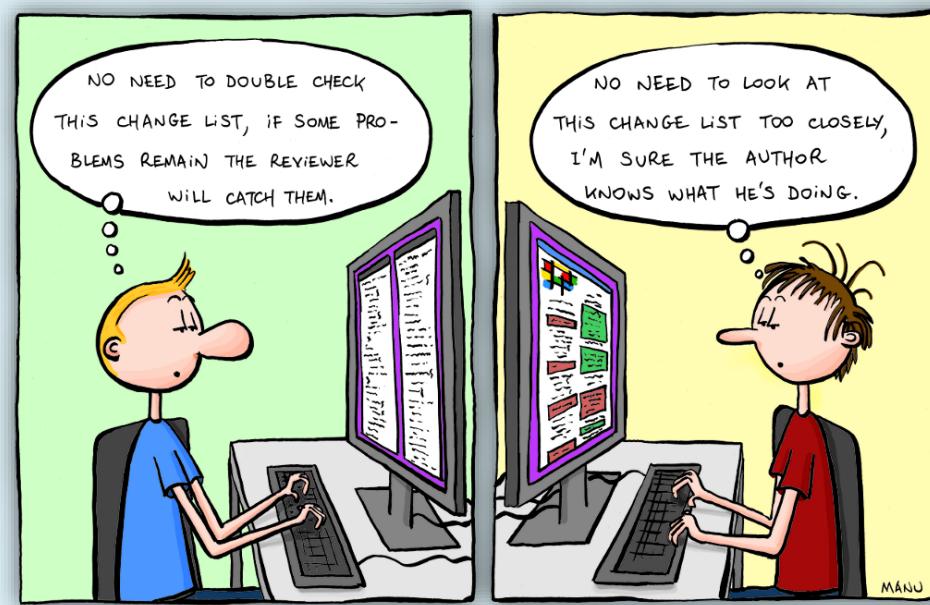
Vorteile

- Präzise Fehlerlokalisierung (Blame assignment)
- Findet alle Fehler
- Keine Laufzeitbeeinflussung

Nachteile

- False Positives (Unentscheidbarkeit)
- Interaktion teilweise notwendig (Schleifeninvarianten)
- Großer Aufwand u. Expertise nötig

Code Reviews

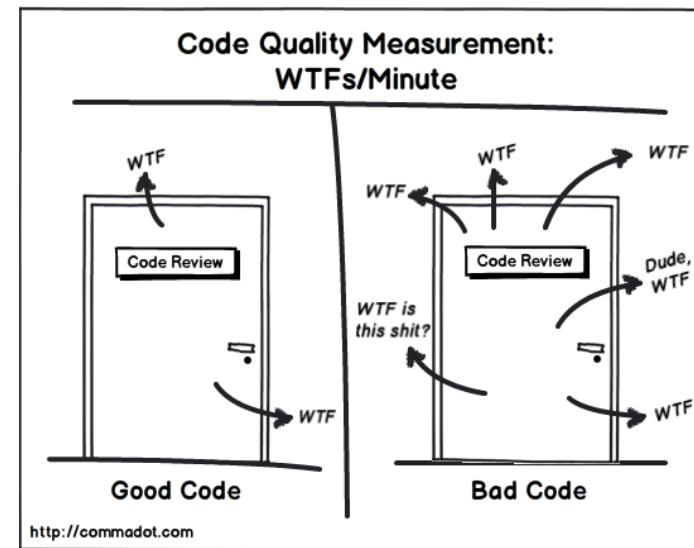


Beispiel

- Code Reviews
 - Kannst du mal auf den Code schauen? Ich finde das Problem nicht... Es kann nicht daran liegen, weil ich X gemacht habe. Und es kann auch nicht daran liegen, weil ich Y gemacht. Und es kann auch nicht—Moment, es **kann** daran liegen. Danke, du hast mir sehr geholfen!

Code Reviews

- Eine Familie verschiedener Techniken
 - Pair Programming
 - Walkthroughs
 - Inspections
 - Personal reviews
 - Formal technical reviews
- Review/inspizieren:
 - Zur genauen Begutachtung
 - Mit einem Auge auf Korrektur und Bewertung
- Menschen (peers) sind die Begutachter



Pair Programming

- Zwei Entwicklerinnen arbeiten zusammen an einem Rechner
- Eine schreibt Code, während die Andere jede getippte Zeile überprüft
- Beide Rollen werden regelmäßig gewechselt
- Vorteile
 - Wissen verteilt sich zwischen Programmierenden
 - Anzahl Fehler wird reduziert, Produktivität wird gesteigert
 - Keine Vorbereitung notwendig
- Nachteile
 - Paar muss miteinander klarkommen
 - Ggf. brauchen Entwicklerinnen länger



Allgemeines Vorgehen bei Code Reviews

- Entwicklerin stellt Code zur Verfügung
 - Projektleiterin setzt Meeting an
 - Teilnehmer bereiten sich vor
 - Meeting findet statt
 - Projektmanagerin bekommt Bericht
-
- Verschiedene Umsetzungen: Walkthroughs, Inspection, (Formal) technical review
 - IEEE1028 Standard

Walkthroughs

- Entwicklerin "führt" andere Personen durch den Quelltext
- Personen geben Feedback über mögliche Fehler, Einhaltung von Standards,...
- Vorteile:
 - Größere Gruppen können teilnehmen, dadurch mehr Wissensaustausch
 - Kaum Vorbereitungszeit
- Nachteile:
 - Entwicklerin tendieren dazu, ihren Code zu rechtfertigen
 - Es ist schwieriger, Code und Entwickelnde zu trennen

Inspections

- Teammitglieder schauen sich Material an
- Team trifft sich und diskutiert über Material
- Ggf. werden nur ausgewählte Aspekte betrachtet
- Vorteile:
 - Fokus auf wichtige Dinge (wenn man sie kennt)
- Nachteil:
 - Guter Moderator notwendig

Formal Technical Review

- Eingeplantes Meeting mit festgeschriebenem Ablauf
- Ergebnis wird in Bericht zusammengefasst
- Fokus auf technische Aspekte, z.B. Abweichung von Anforderungen oder Standards
- Unabhängiges Team ohne Entwickler
- Vorteil:
 - Unabhängig vom Entwickler
 - Festgeschriebener Ablauf
- Nachteil:
 - Aufwand

Personal Review

- Informell durch Entwicklerin selbst
- Kann jede Entwicklerin einfach durchführen
- Nicht besonders objektiv

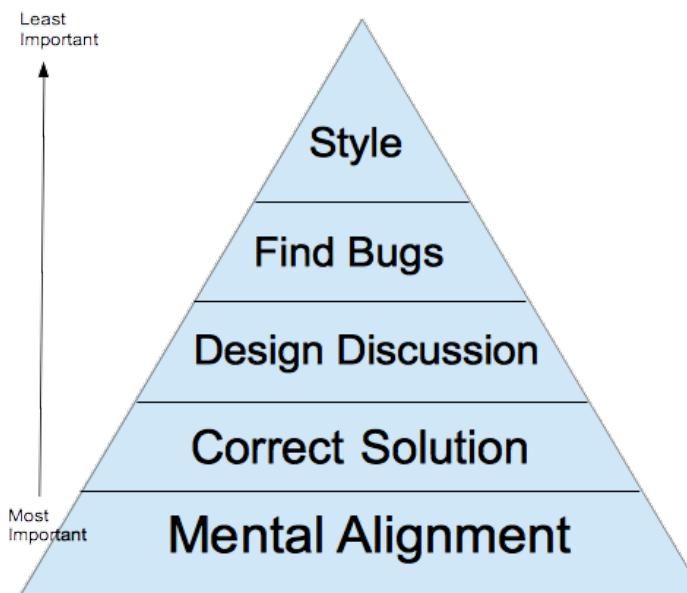
Was sollte wann gereviewed werden

- Jedes Projekt-Artefakt:
 - Anforderungen, Design, Code, Dokumentation, Tests, Konfigurationen, ...
- Meetings sind fest eingeplant und sollten definierte Dauer haben

Review benötigt	Review nicht zwingend
Komplizierte Algorithmen	Triviale Prozesse
Kritische Pfade, wo Fehler zu Systemfehlern führen würden	Pfade, wo Fehler keine oder nur geringe Auswirkungen auf das System haben
Teile mit neuer Technologie, Frameworks, etc.	Teile, die ähnlich zu bereits zuvor positiv begutachteten Teilen sind
Teile, die von unerfahrenen Teammitgliedern entwickelt worden	Wiederverwendete oder redundante Teile

Ziele

Code Review: Hierarchy of Needs



Teammitglieder zusammen halten (gleiches Verständnis über Aufgaben und Code)

Gute Pull Requests definieren:

- Richtiges Item für jetzige Situation?
- Zustimmung des Teams, ob das der richtige Change ist?
- Wie kann ich die Änderung in kleine Teile zerlegen, die einfach zu reviewen sind?
- Wie kann ich die Änderungen Testen und auf Korrektheit überprüfen?

Gemeinsames mentales Modell hilft:

- Risiken zu vermeiden
- Änderungen schneller nachzuvollziehen
- Besseres Softwaredesign zu entwickeln

Pull Request: Schlechtes Beispiele

Title: Fix uninitialized memory bug

Description:

This is the bug Bob and I talked about earlier. I had trouble with the compiler but managed to make this work. Let me know what you guys think.

Kein Kontext, unklarer Titel

Wo ist der Bug?

Wie kritisch ist die Änderung?

Was war dieser Bug von Bob?

Welche Probleme gab es mit dem Compiler?

Pull Requests: Gutes Beispiele

Title: Fix process crash on startup from uninitialized memory [#54633]

Description: This bug was causing process crashes on boot due to a memory initialization error in our statistics Counter class. I talked this over with Bob, and we both agree the crashes are a rare edge case that don't warrant a hot-fix release. Here's a summary of the changes:

- Moved the underlying int variable into the class initializer to prevent uninitializied memory in the Counter.
- Reworked the Counter interface to simplify caller conditional logic and prevent further off-by-one counting problems.
- Added a unit test that exposes the crash

Testing:

I've verified the test suite still passes, and verified manually that the crash doesn't happen locally.

Titel mit Link zu Subsystem (z.B. Bug Repo, Issue Tracker, Commit Message, etc.)

Klarer Titel -> Prozess Crash durch nicht-initialisierten Speicher

Beschreibung gibt Auskunft, wo der Bug und unter welchen Bedingungen auftrat

Kritikalität ist beschrieben

Zusammenfassung der Änderung als strukturierte Liste (mentales Bild + Erwartungen vor Code Review)

Testing gibt dem Reviewer Vertrauen, dass die Änderungen durchdacht und überprüft sind.



Aufgaben des Teams

- Guten Review erstellen
 - Team ist verantwortlich für Review, nicht das Produkt
- Probleme finden (nicht beheben)
- Entscheidung treffen:
 - Akzeptiert, akzeptiert mit kleinen Änderungen (einstimmig)
 - Bedeutende Änderungen, abgelehnt (ein Veto reicht)

Aufgaben der Teamleiterin

- Voreilige Reviews vermeiden
- Guten Review sicherstellen...
- ...oder Gründe für Scheitern berichten
 - Fehlendes Material
 - Fehlende, unvorbereitete Gutachter
- Meetings koordinieren
 - Material verteilen
 - Zeitplan für Meeting (und dessen Einhaltung)
 - Ort für Meeting

Aufgaben der Gutachterin

- Vorbereitung durch Begutachtung des Materials und aktive Teilnahme
- Professionelles Verhalten: Berechtigte positive und negative Kommentare. Gut: Fragen als Feedback

“This design is broken.”

Warum? Wie kann es verbessert werden?

“I don’t like this change.”

Warum? Was würde dir besser gefallen?

“Can you rewrite this to be more clear?”

Was ist das Problem hier? Wie sollte ich es umschreiben? Was ist unklar?

“How does this code handle negative integers?”

“This section is confusing to me, I don’t understand why class A is talking to class B”

“It looks like you broke an interface boundary here. How will that affect the user?”

Spezifisches Feedback damit der Entwickler über die Auswirkung der Änderung nachdenken kann (auch wenn der Reviewerin der Crash her schon klar ist). Testing Gap? Issues wie unten könnten auch beabsichtigt sein. Gebt den Entwicklern die Möglichkeit der Begründung.

Bericht

- Zusammenfassung
 - Gefundene Probleme
 - Empfehlung
-
- Projektleiter über Status informieren
 - Frühwarnsystem für mögliche Probleme
 - Logbuch für Fortschritt und involvierte Leute

Wann Review?

- Vor dem Commit oder danach?

Pre-Commit Review

- Stellt sicher, dass Code Guidelines und Qualitätsstandards eingehalten werden
- Hilft, dass eigenes Review auch durchgeführt und nicht auf andere Gutachter abgeschoben wird (siehe Comic)
- Verhindert, dass andere Entwicklerinnen durch selbst eingeführte Bugs leiden

Aber:

- Reduziert Produktivität, da nicht weiter am Code gearbeitet werden kann, bis Review erfolgt ist
- Nach dem Review könnte die Entwicklerin aber eine geänderte (nicht-gereviewte) Version (versehentlich/absichtlich) Commiten

Post-Commit Review

- Entwicklerin kann Änderung commiten und weiter arbeiten
- Andere Entwicklerinnen sehen frühzeitig die Änderung und können ihre eigene Arbeit daran anpassen
- Bei komplexen Änderungen kann dies in mehrere Teile zerlegt und diese individuell gerefviewt werden

Aber:

- Erhöhte Chance, dass schlechter Code ins Repository geht und somit das gesamte Team betrifft
- Falls Probleme bei Review auftreten, dann braucht es evtl. eine Zeit, bis die Entwicklerin wieder an dem betroffenen Modul arbeitet

Best Practices

- Pre-Commit für kritischen Code (major changes)
- Post-Commit für triviale Änderungen
- Commit früh und oft, um viele kleine Reviews zu ermöglichen
- Entwicklerinnen sollten Commit gut dokumentieren (z.B. gute Commit Message / Pull Request)
- Commit Code zu einem Feature / Test Branch
- Track Bugs, um sicherzustellen, dass sie auch tatsächlich gefixt werden
- Review den Code und nicht die Entwicklerin

Tools

- Review Ninja (<http://review.ninja>)
 - Gerrit (<https://code.google.com/p/gerrit/>)
 - FishEye(<https://www.atlassian.com/software/fisheye/overview>)
-
- SimpleCov(code coverage, <https://github.com/colszowka/simplecov>)
 - Flog (code complexity,<http://ruby.sadi.st/Flog.html>)
 - Reek (code smells,<https://github.com/troessner/reek>)
 - Cane (code quality, <https://github.com/square/cane>)
 - Rails_best_practices(Rails specific, https://github.com/flyerhzm/rails_best_practices)

Testen vs. Reviews

- Testen verläuft in 2 Phasen:
 - Testfälle finden Fehler
 - Ursache von Fehlern muss gefunden werden
- Bei Reviews findet man Fehler und deren Ursache in einem Schritt
- Erst Review, dann Testen
- Erst Testen, dann Verifikation

Was Sie mitgenommen haben sollten:

- Warum brauchen wir Tests/Verifikation/Code Reviews?
- Was kann man mit Tests nicht zeigen? Warum?
- Nennen/Erklären Sie die 4 vorgestellten Ebenen von Tests.
- Erklären Sie Black-Box/White-Box/Regressions/ Nightly/Daily-Builds.
- Nennen/Erklären Sie den Vorteil von Code Reviews gegenüber testen.
- Welche Strategie würden Sie einem kleinen Unternehmen (3 Mitarbeiter) empfehlen, um möglichst fehlerfreie Software auszuliefern? Begründen Sie Ihre Entscheidung.

Literatur

- McConnell. Code Complete. 2004. [Chapter 20-22] (contains also references of further interesting papers)
- Sommerville. Software Engineering. 2002 [Chapter 22-23]
- Beckert and others. Verification of object-oriented software: The KeY approach. 2007
- Code Reviews: http://www.baskent.edu.tr/~zaktas/courses/Bil573/IEEE_Standards/1028_2008.pdf
- Wikipedia

Softwaretechnik

Softwareentwicklungsprozesse



SOFTWARE
SYSTEME

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Übung: Anforderungen des SWT Praktikums

Wie ist das gemeint?

Alle Messdaten soll das Backend **automatisch** vom MQTT-Broker anfordern. Der MQTT-Broker stellt die Messdaten aller Geräte pro Sekunde in Watt zur Verfügung. Im Backend sollte sowohl für das temporäre, als auch für das langfristige Speichern der Daten eine Datenbank verwendet werden. Es sollen alle neuen Energiemessdaten **temporär** **Wie lange?** abgelegt werden. Ist das **Speicherlimit** der Datenbank erreicht, werden die ältesten Daten zuerst **gelöscht.** **Wie hoch?**

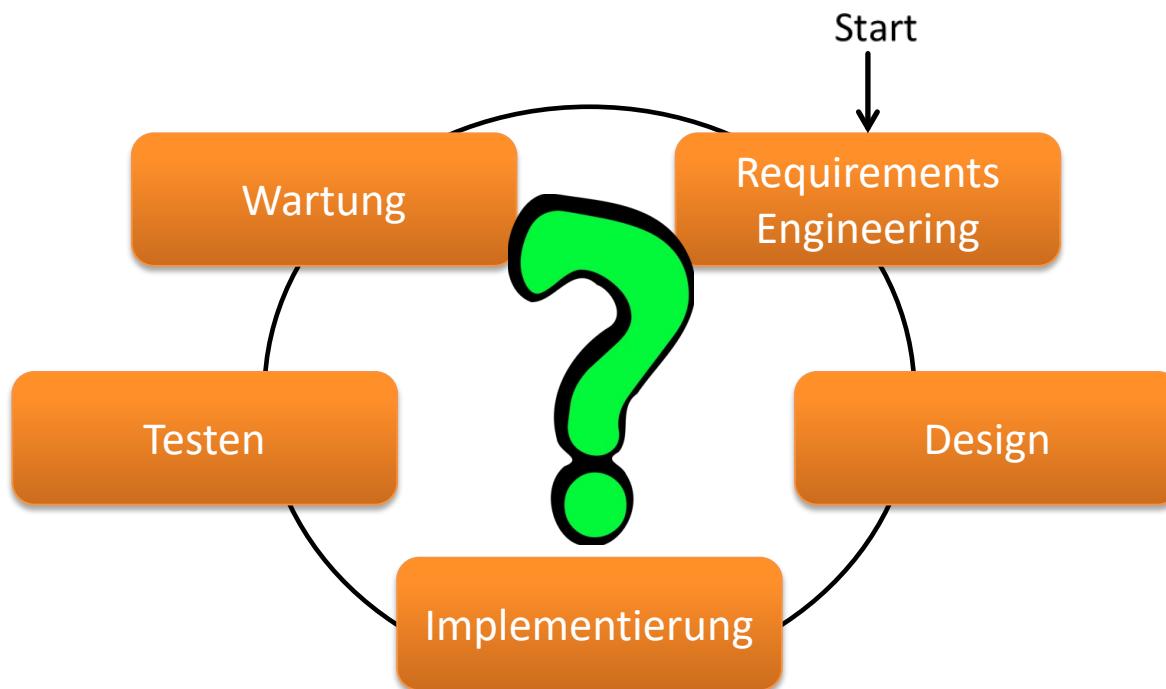
Behalten wir die Statistiken für gelöschte Daten?

Im Backend sollen die **statistischen Berechnungen** durchgeführt werden, da diese mitunter sehr aufwändig werden könnten. Es sollen der Idle-, Durchschnitts- und der **maximale Energieverbrauch** für **alle Geräte** berechnet werden. Die Ergebnisse aller statistischen Berechnungen sollen **langfristig** gespeichert werden. Das Backend soll innerhalb eines **Microservices** gekapselt werden und über eine REST-API mit dem Frontend kommunizieren. **Vorgaben?** **Unbegrenzt? In Wochenscheiben?** **Technologie?** **Aggregiert über den gesamten Zeitraum?** **Sprache?** **Auswirkungen auf Speicherlimit?** **Wie bestimmen?**

Einzeln oder aggregiert?

Verschlüsselt?
A-/Synchron?

Einordnung



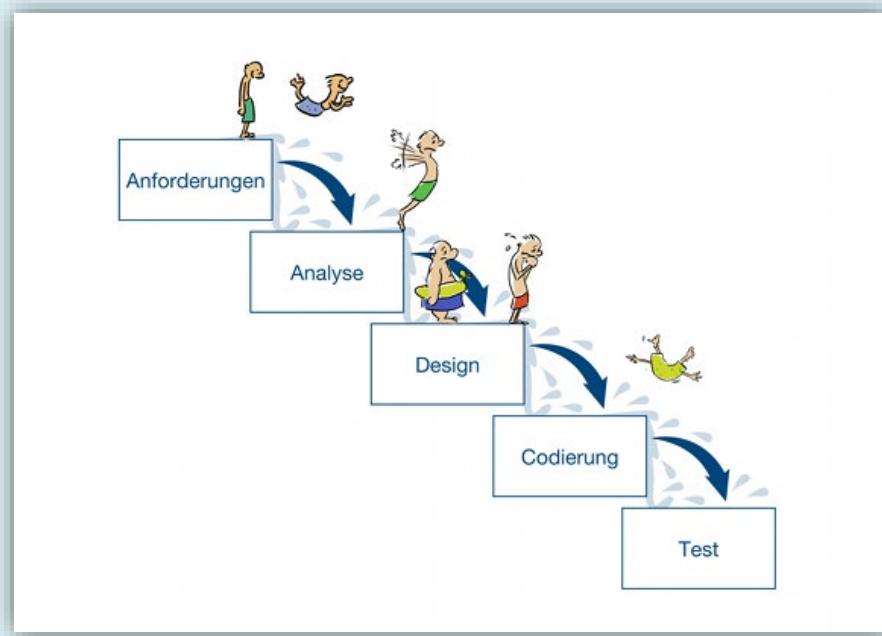
Lernziele

- Überblick und grundlegendes Verständnis für Entwicklungsprozesse haben
- Traditionelle Softwareentwicklungsprozesse (sequenzielle Modelle) kennen und deren Probleme verstehen
- Alternative, aktuelle Ansätze der Softwareentwicklung bewerten können und abhängig von der Projekt- / Betriebsgröße entscheiden können, welcher Ansatz zu präferieren ist

Vorgehensmodelle

- Beschreiben *wie, wann, welche* der Tätigkeiten der Softwareentwicklung (Phasen des Lebenszyklus) ausgeführt werden
- Bisher: Brute-Force-Modell: Einfach drauf losprogrammieren
- Jetzt:
 - Traditionelle Entwicklungsmodelle
 - Große Teams, feste Anforderungen
 - Neuartige Modelle (agile Methoden)
 - Angepasst auf schnelle Entwicklungsphasen, kurze Release-Zeiten und sich ständig ändernde Anforderungen

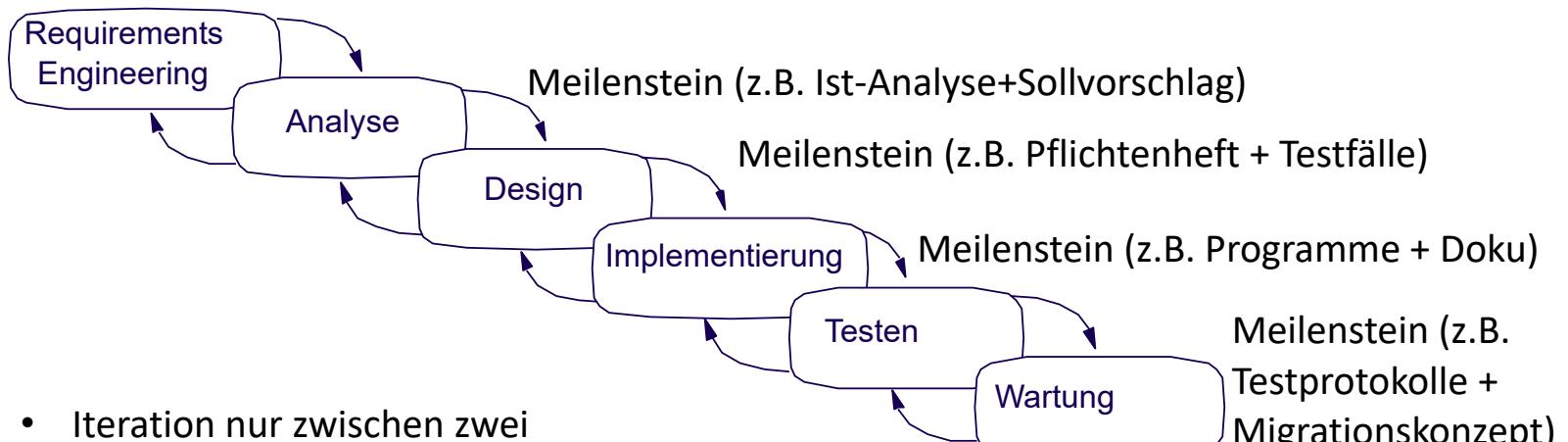
Sequentielle Modelle



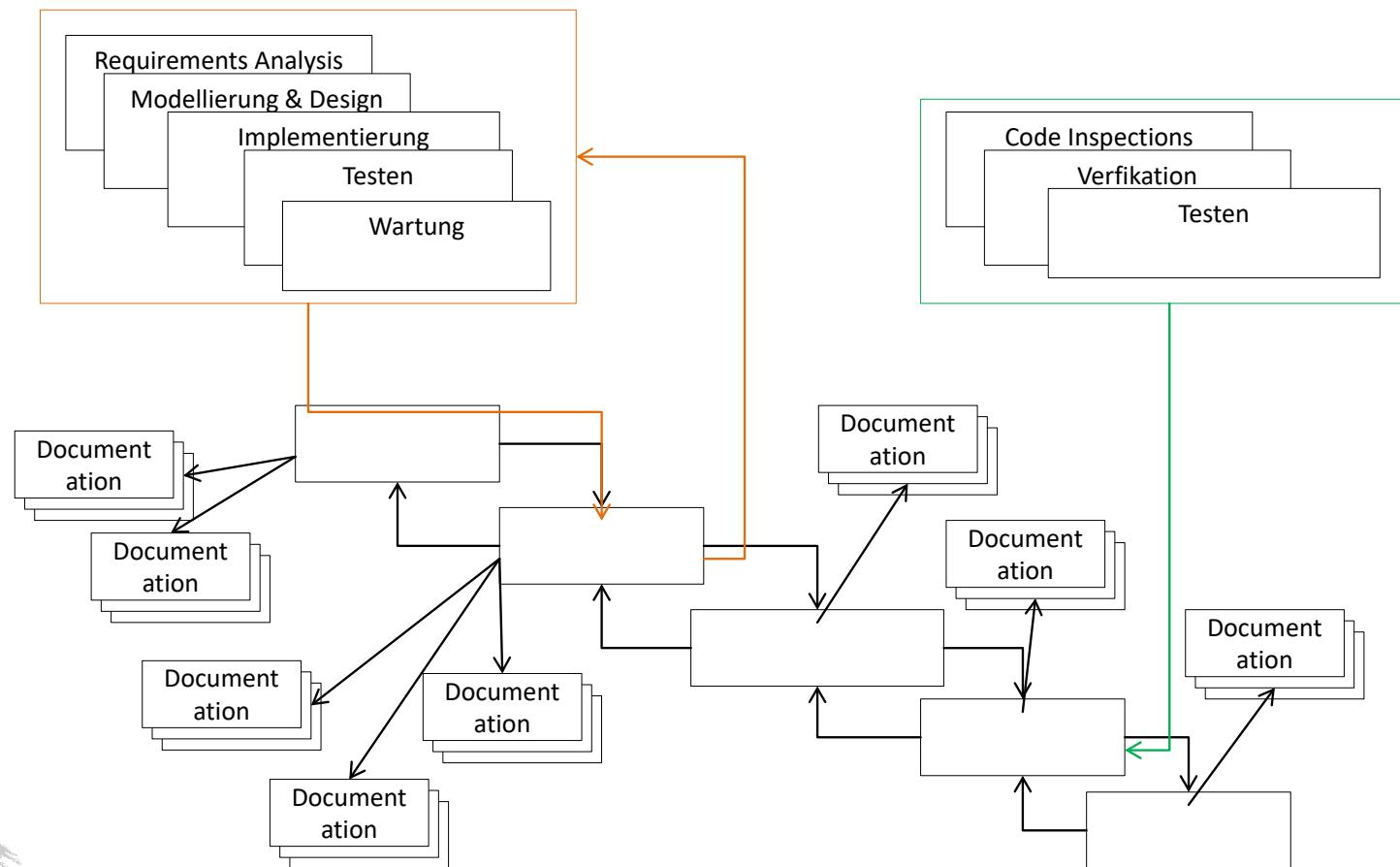
Grundidee von Seq. Modellen

- Schritt für Schritt werden alle Phasen des Lebenszyklus nacheinander „abgearbeitet“
 - Planen / Konzipieren
 - Entwerfen / Ausarbeiten
 - Inbetriebnehmen / Warten
- Modelle:
 - Wasserfallmodell
 - V-Modell

Wasserfallmodell



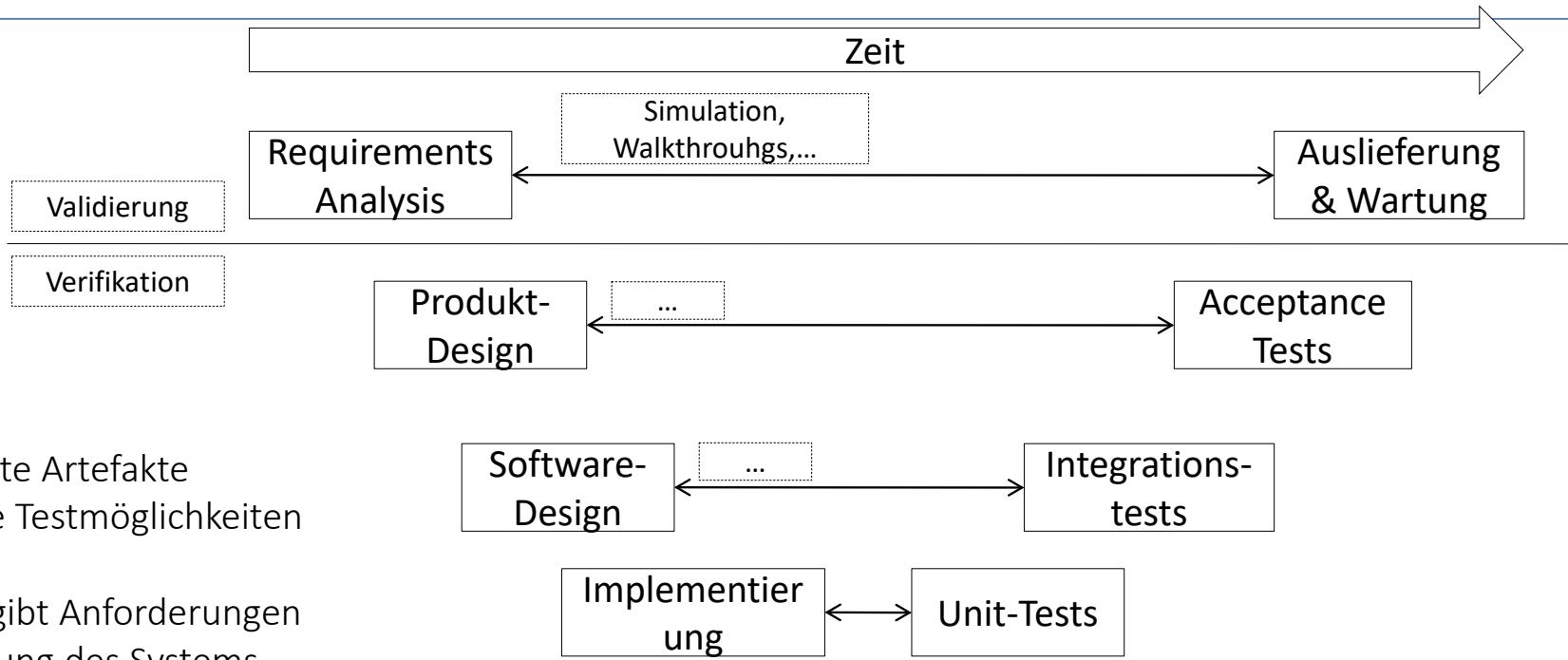
Wasserfallmodell: Dokumentation



Wasserfallmodell: Diskussion

- Vorteile:
 - *Dokumentation* nach jeder Phase verfügbar
 - Klare *Trennung* der Phasen und Verantwortlichkeiten
 - Analog zu Ingenieurprojekten (Brückenbau etc.)
- Nachteile:
 - *Starres* Vorgehen (veraltete Dokumentation)
 - Reaktionen auf *geänderte* Anforderungen schwierig
 - Anforderungen, Design, etc. *früh fixiert*, Änderungen nicht vorgesehen (aber Änderungen sind natürlich!)
 - *Späte* Qualitätsprüfung (Baue ich überhaupt das *richtige Produkt?* Erster *Prototyp sehr spät* verfügbar!)
 - Meist unrealistisch in der Praxis

V-Modell



Artefakte vom V-Modell

- Anwenderanforderungen: **Lastenheft**
- Technische Anforderungen: **Grobentwurf/Pflichtenheft**
- SW-Architektur: Feinspezifikation
- SW-Entwurf: Codedokumentation (z.B. javadoc)
- Implementierungsdokumentation: Installationsanleitung
- Prüfprozedur und Prüfergebnis: Abnahmedokumente

Probleme sequentieller Modelle

- Anforderungen oft nicht klar und können sich ändern
 - Bei großen Projekten dauert Entwicklung lange und es fehlt Erfolgskontrolle
 - Erfordert viel Dokumentation → Overhead für kleine Projekte
-
- Gefahr von Missverständnissen
 - Prototyp erst am Ende des Projektes
 - Je grundlegender ein Fehler, umso später wird er gefunden

Teilweise Lösung: Prototypen

Ein Prototyp ist ein Softwareprogramm, entwickelt, um Hypothesen zu testen, zu explorieren und zu validieren. Dient der Reduzierung von Risiken.

Ein explorierender Prototyp, auch bekannt als *Wegwerfprototyp*, zielt auf die *Validierung von Anforderungen* oder *Erprobung von Designentscheidungen ab*.

- UI prototype: validiert Nutzeranforderungen
- Rapid prototype: validiert funktionale Anforderungen
- Experimental prototype: validiert technische Machbarkeit

Weitere Arten von Prototypen

Ein evolutionärer Prototyp ist dafür gedacht sich zu entwickeln, so dass er in Schritten zum finalen Produkt ausreift.

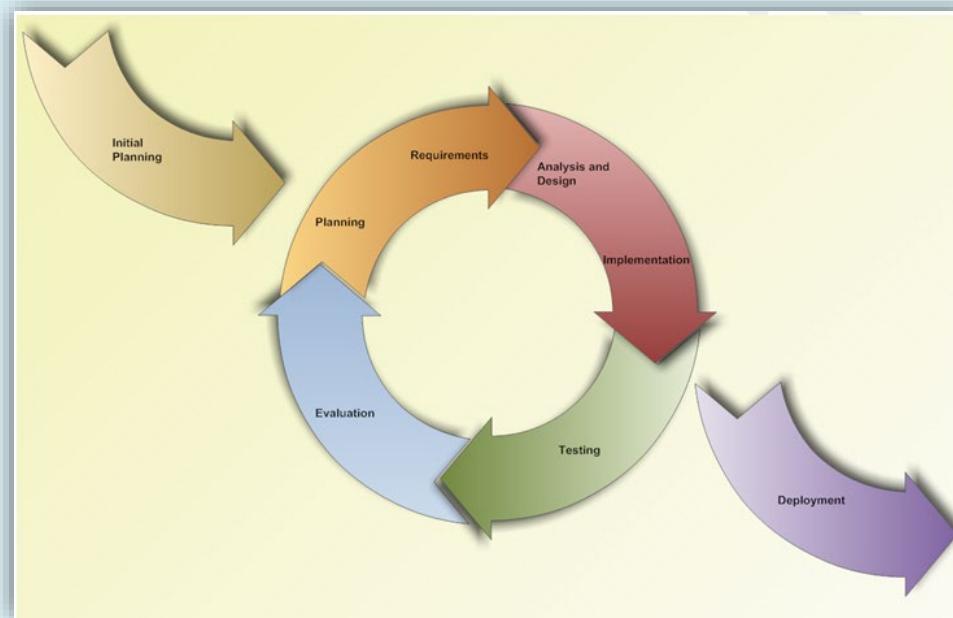
- Beim iterativen “Wachsen” der Anwendung ist *Redesign* und *Refactoring* ständiger Begleiter.



Horizontaler Prototyp wird für die Ebene mit dem größten Risiko erstellt (reduziert Missverständnisse) und realisiert alle Funktionen einer Ebene (gut, um Beziehungen zwischen Funktionen zu erkennen).

Vertikaler Prototyp setzt Kernfunktionalität um (Machbarkeits- / Effizienztest, Aufwandsabschätzung) und dient als Pilotensystem (gut, um eine komplexe Funktion besser zu verstehen).

Iterative Modelle



Idee von iterativen Modellen

- Lat. iterare: wiederholen
- Idee:
 - Vollständiges Analysieren und Planen ist a priori unmöglich, daher iterative “Annäherung” an das Ziel
 - Erkenntnisse aus jedem Iterationsschritt werden benutzt, so lange, bis definiertes Ziel erreicht ist
- Beim ersten Mal macht man typischerweise Fehler
- Darum integrieren, validieren und testen so oft wie möglich

Prototypen

- Möglichst früh eine erste Version erstellen
- Insbesondere bei Unklarheiten bei Kundinnenwünschen
- Selbst wenn Kernfunktionalität fehlt
- Kunde kann Fortschritt erkennen und (Teil-) Lösung bewerten
- Erinnerung: horizontale vs. vertikale Prototypen

Prototyp vs. Inkrementelle Entwicklung

- Prototypen werden oft weggeworfen (da keine Qualitätskriterien eingehalten wurden, sondern nur zum Zeigen entwickelt wurden)
- Inkrement:
 - Software-Baustein(e), die zu einem existierenden System oder Subsystem hinzugefügt werden, um dessen Funktionalität oder Leistung zu vergrößern oder zu verändern
 - Inkrement kann Subsystem entsprechen
 - Inkrementelle Systementwicklung: beginnt mit Kernsystem, schrittweise Erweiterung

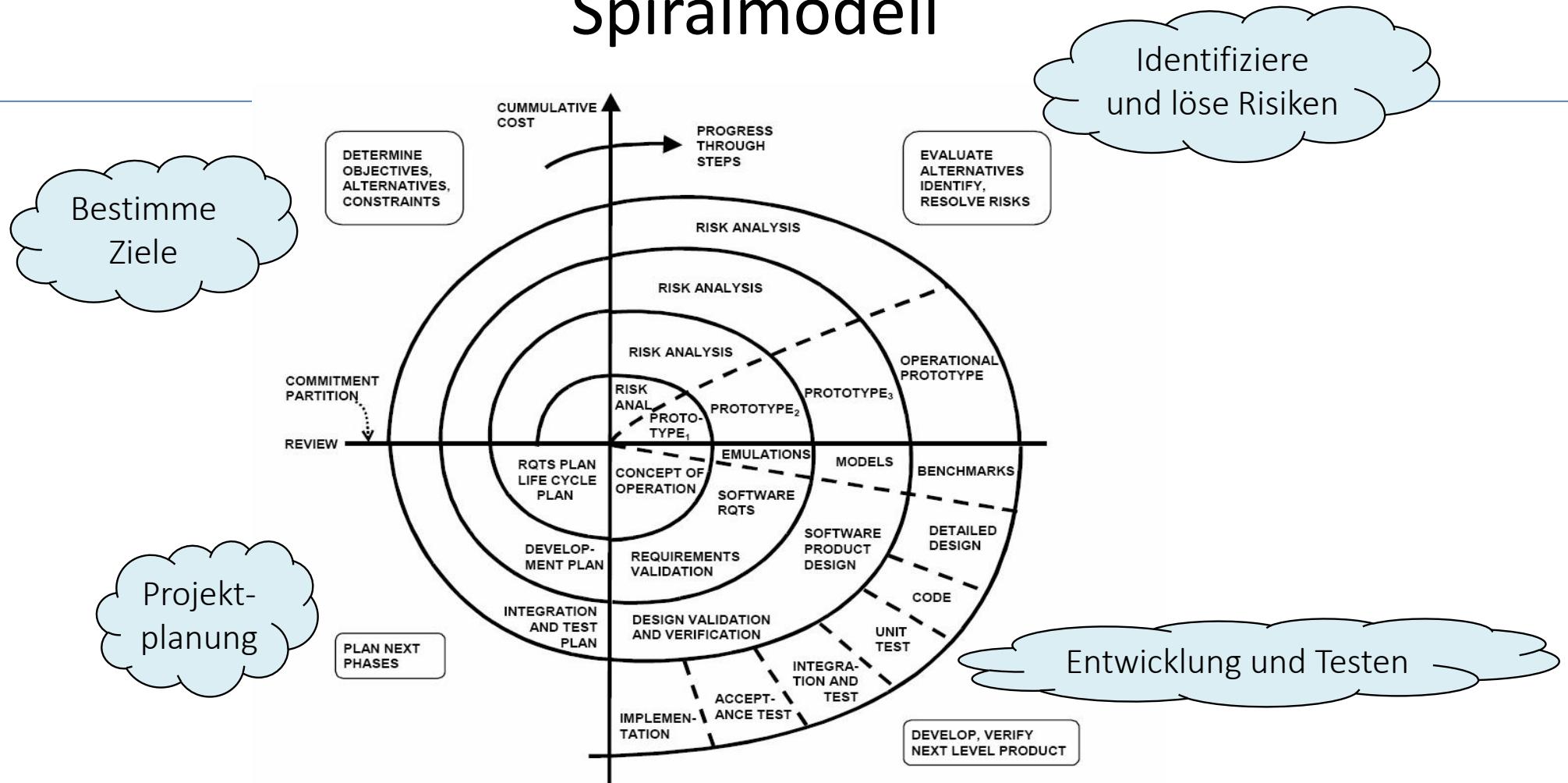
Prinzipien der inkrementellen Entwicklung

- Falls möglich, habe *immer eine laufende Version des Systems*, selbst, wenn der größte Teil der Funktionalität nicht implementiert ist
- *Integriere* neue Funktionalität so früh wie möglich
- *Validiere* inkrementelle Versionen gegen Nutzeranforderungen

Spiralmodell

- Risiko-orientiertes Vorgehen
 - Suche alle Risiken, von denen das Projekt bedroht ist. Wenn es keine gibt, ist das Projekt erfolgreich abgeschlossen.
 - Bewerte die erkannten Risiken, um das Größte zu identifizieren.
 - Suche einen Weg, um das größte Risiko zu beseitigen, und gehe diesen Weg. Wenn sich das größte Risiko nicht beseitigen lässt, ist das Projekt gescheitert.
- Generisches Modell (kann bspw. auch zu Wasserfallmodell werden)

Spiralmodell



Spiralmodell

- Kombiniert Wasserfallmodell mit Prototypen und iterativer Entwicklung
 - Gleiche Aufgaben wie im Wasserfallmodell
 - Jede Aufgabe wird durch Prototypen abgeschlossen
 - Fortschritt kann besser kommuniziert werden
 - Risikoanalyse erlaubt frühe Erkennung von Risiken
- War erfolgreich im Einsatz
 - Microsoft
 - IBM
 - US Militär (future combat system)

Weitere Modelle

- Unified Process (UP)
 - Inkrementelle Implementierung der funktionalen Anforderungen (wichtigste zuerst)
 - Kurze Iterationen (Wochen)
 - Jede Iteration endet mit vollständig laufendem System
- Evolutionäres Modell
 - Kundin bekommt früh Vorab-Version (erfordert Einbindung der Kundin, schwierige Gesamtplanung)
 - Ermöglicht früh Return-on-Investment

Aufgabe

- Welcher Entwicklungsprozess würde sich für NoMoreWaiting eignen?
 - Wasserfallmodell
 - V-Modell
 - Prototypen
 - Spiralmodell
 - Unified Process

Agile Softwareentwicklung



Erinnerung: Gründe für das Scheitern

- Zeit aufgebraucht
- Budget aufgebraucht
- Falsches Produkt
- Schlechtes Produkt
- Nicht wartbar
- Ineffiziente Software
- Ökonomische Ursachen

Gründe:

- Dokumentation hat hohen Aufwand und veraltet schnell -> trotzdem folgen wir ihr
- Projektpläne sind ungenauer, ändern sich leider nicht -> trotzdem halten wir daran fest



Agile Softwareentwicklung

- Relativ neuer Ansatz (ca. 1999)
- Im Spannungsfeld zwischen:
 - Qualität, Kosten und Zeit
 - Ungenaue Kundenwünsche und instabile Anforderungen
 - Langen Entwicklungszeiten und überzogenen Terminen
 - Unzureichender Qualität
- Gegenbewegung zu schwergewichtigen, bürokratischen iterativen Prozessen, die oft zu viel Dokumentation erfordern

Manifesto für Agile Software Entwicklung

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions** over processes and tools
- Working software** over comprehensive documentation
- Customer collaboration** over contract negotiation
- Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.



*Kent Beck
Ward Cunningham
Andrew Hunt
Robert C. Martin
Dave Thomas*

*Mike Beedle
Martin Fowler
Ron Jeffries
Steve Meller*

*Arie van Bennekum
James Grenning
Jon Kern
Ken Schwaber*

*Alistair Cockburn
Jim Highsmith
Brian Marick
Jeff Sutherland*
[Agilemanifesto.org]

Manifest der agilen Softwareentwicklung

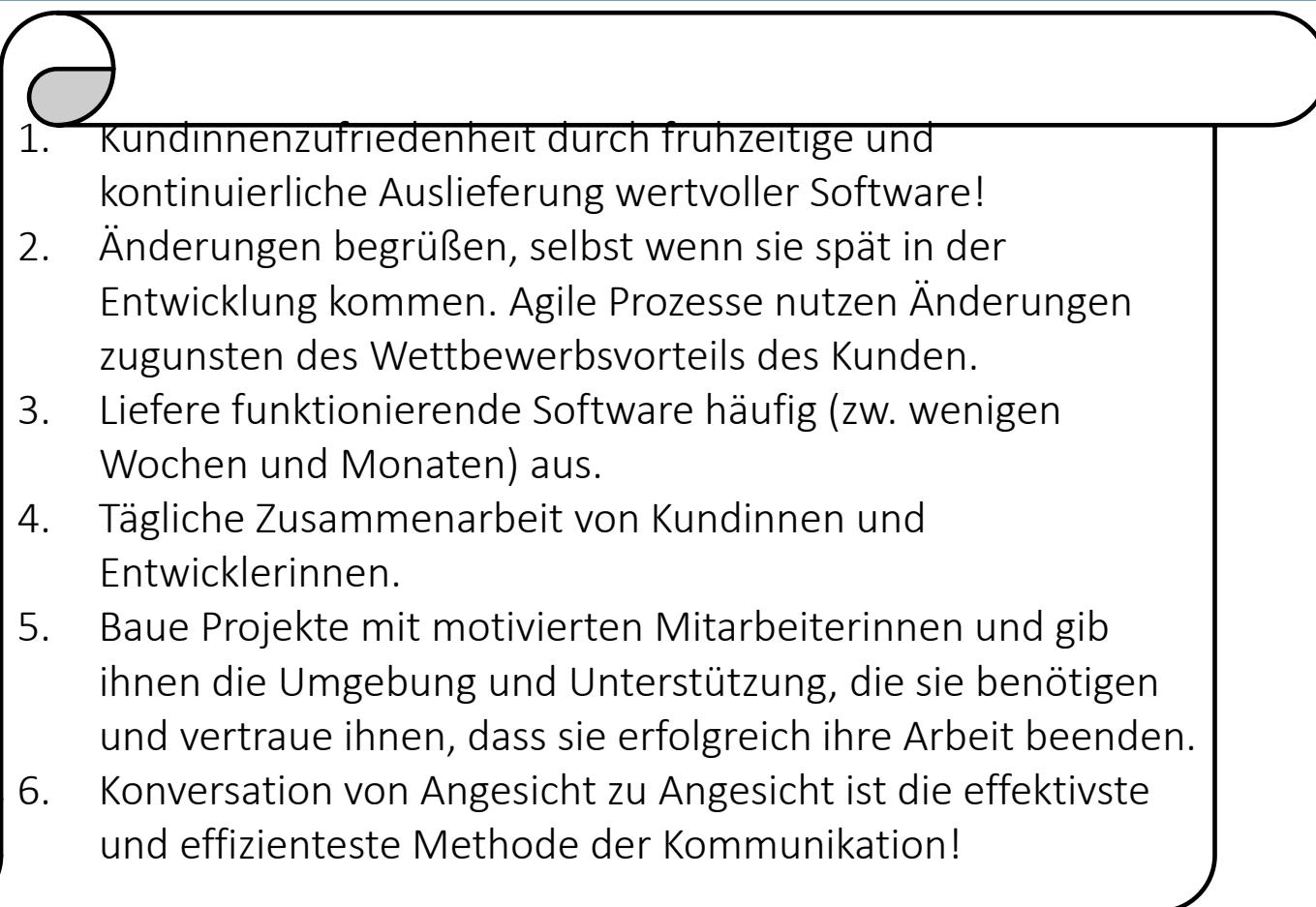
- Menschen und Kooperation vor Werkzeugen und (automatisierten) Prozessen
- Funktionsfähige Software vor umfassender Dokumentation
- Zusammenarbeit mit Kundinnen vor bürokratischen Vertragsverhandlungen
- Dynamische Reaktion auf Veränderungen vor statischer Planeinhaltung
- (trotzdem sind Prozesse, Dokumentation, ... vorhanden und wichtig)

Was ist Agile Softwareentwicklung?

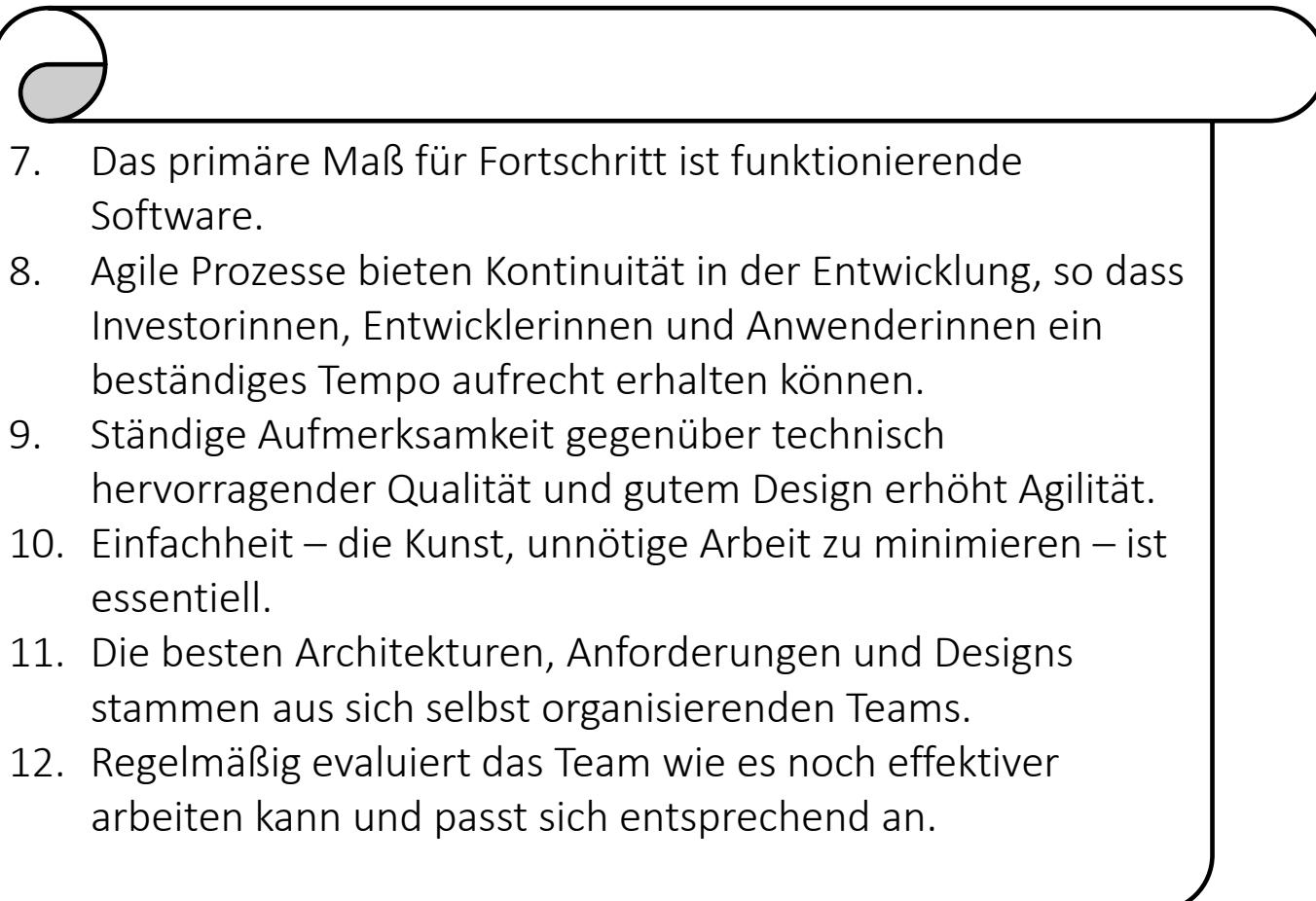
- Menge von Softwareentwicklungsmethoden
 - Basierend auf iterativer und inkrementeller Entwicklung
- Meist kleine Gruppen (6-8)
- Kunde ist in Projekt integriert

Schwergewichtige Prozesse	Agile Prozesse
Dokumentenzentriert	Codezentriert
Up-Front Design	Minimale Analyse zu Beginn
Reglementiert	Adaptiv, Prozess wird angepasst
Abarbeitung eines Plans	Ständige Anpassung der Ziele
Lange Releasezyklen	Häufiges Deployment

Die 12 agilen Prinzipien

- 
1. Kundinnenzufriedenheit durch frühzeitige und kontinuierliche Auslieferung wertvoller Software!
 2. Änderungen begrüßen, selbst wenn sie spät in der Entwicklung kommen. Agile Prozesse nutzen Änderungen zugunsten des Wettbewerbsvorteils des Kunden.
 3. Liefere funktionierende Software häufig (zw. wenigen Wochen und Monaten) aus.
 4. Tägliche Zusammenarbeit von Kundinnen und Entwicklerinnen.
 5. Baue Projekte mit motivierten Mitarbeiterinnen und gib ihnen die Umgebung und Unterstützung, die sie benötigen und vertraue ihnen, dass sie erfolgreich ihre Arbeit beenden.
 6. Konversation von Angesicht zu Angesicht ist die effektivste und effizienteste Methode der Kommunikation!

Die 12 agilen Prinzipien

- 
7. Das primäre Maß für Fortschritt ist funktionierende Software.
 8. Agile Prozesse bieten Kontinuität in der Entwicklung, so dass Investorinnen, Entwicklerinnen und Anwenderinnen ein beständiges Tempo aufrecht erhalten können.
 9. Ständige Aufmerksamkeit gegenüber technisch hervorragender Qualität und gutem Design erhöht Agilität.
 10. Einfachheit – die Kunst, unnötige Arbeit zu minimieren – ist essentiell.
 11. Die besten Architekturen, Anforderungen und Designs stammen aus sich selbst organisierenden Teams.
 12. Regelmäßig evaluiert das Team wie es noch effektiver arbeiten kann und passt sich entsprechend an.

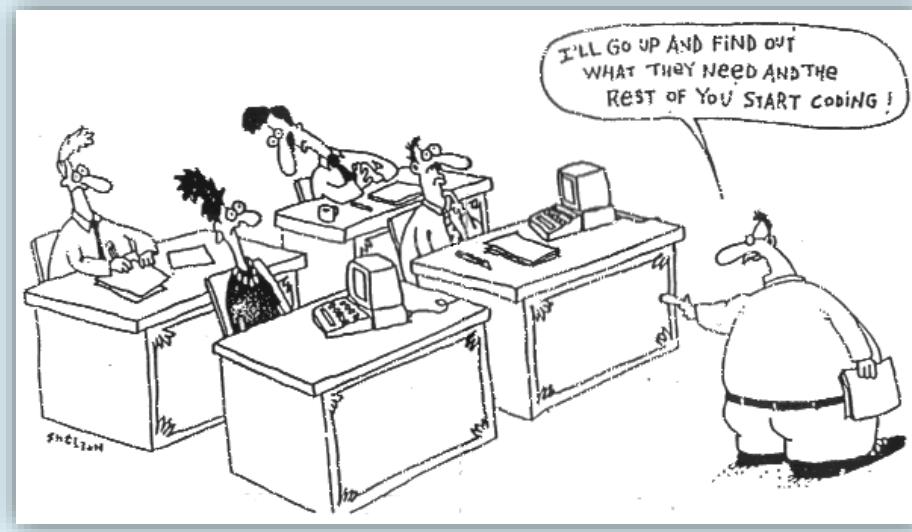
Projektrisiken

- Zeit und Budget
 - Kleine Iterationen mit kalkulierbaren Budget
 - Benötigte Funktionalität wird priorisiert
 - Auslieferbares Produkt nach jeder Iteration
- Falsches oder schlechtes Produkt
 - Stakeholder und Kundinnen sind in jeder Iteration involviert
 - Testen und Benutzen des Produktes am Ende jeder Iteration
 - Schnelle Rückmeldungen zur Neupriorisierung
- Wartbar, da ständig im Einsatz

Konsequenzen

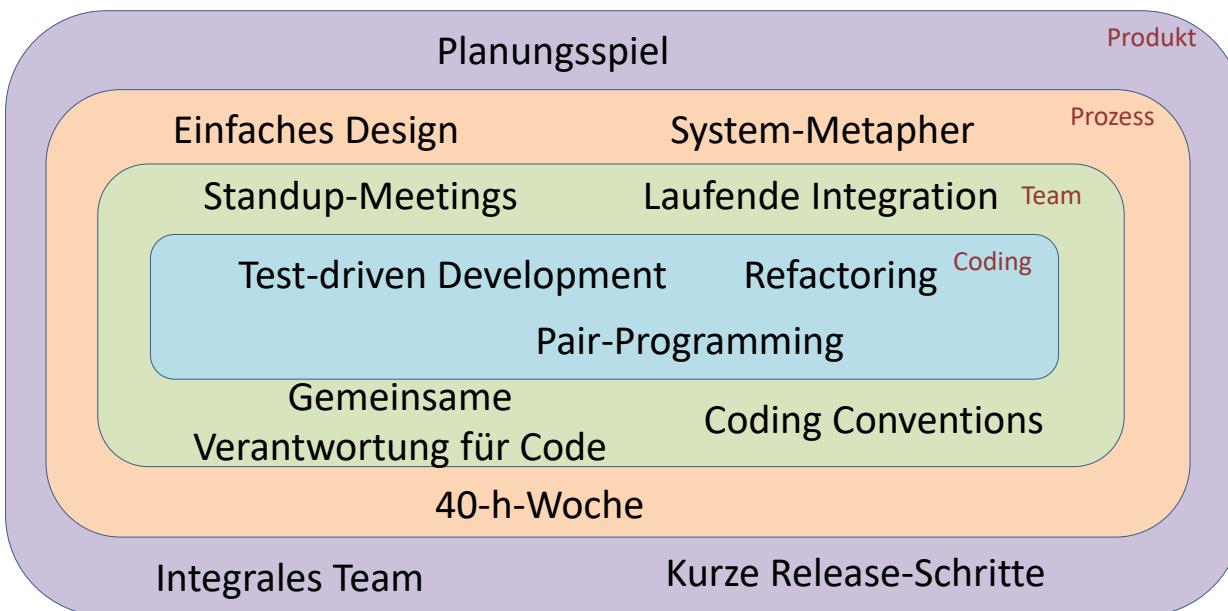
- Kein vollständiger Projektplan (wann [alle] Anforderungen erfüllt?)
- Anforderung als ständiger Input (wie sammeln und ordnen?)
- Evolutionäres Softwaredesign statt einmaliger Entwurf
- Häufige Refaktorisierung des Codes, um neue Anforderungen und geändertes Design umzusetzen (mit Reimplementierung existierender Funktionalität)
- Technische Schulden können sich aufhäufen
- CI/CD Prozess benötigt für schnelle Iterationen
- Regression und kontinuierliches Testing erforderlich
- Hoher Kommunikationsaufwand mit Teams und entsprechende Organisationsstruktur notwendig

Extreme Programming (XP)



Extreme Programming (XP)

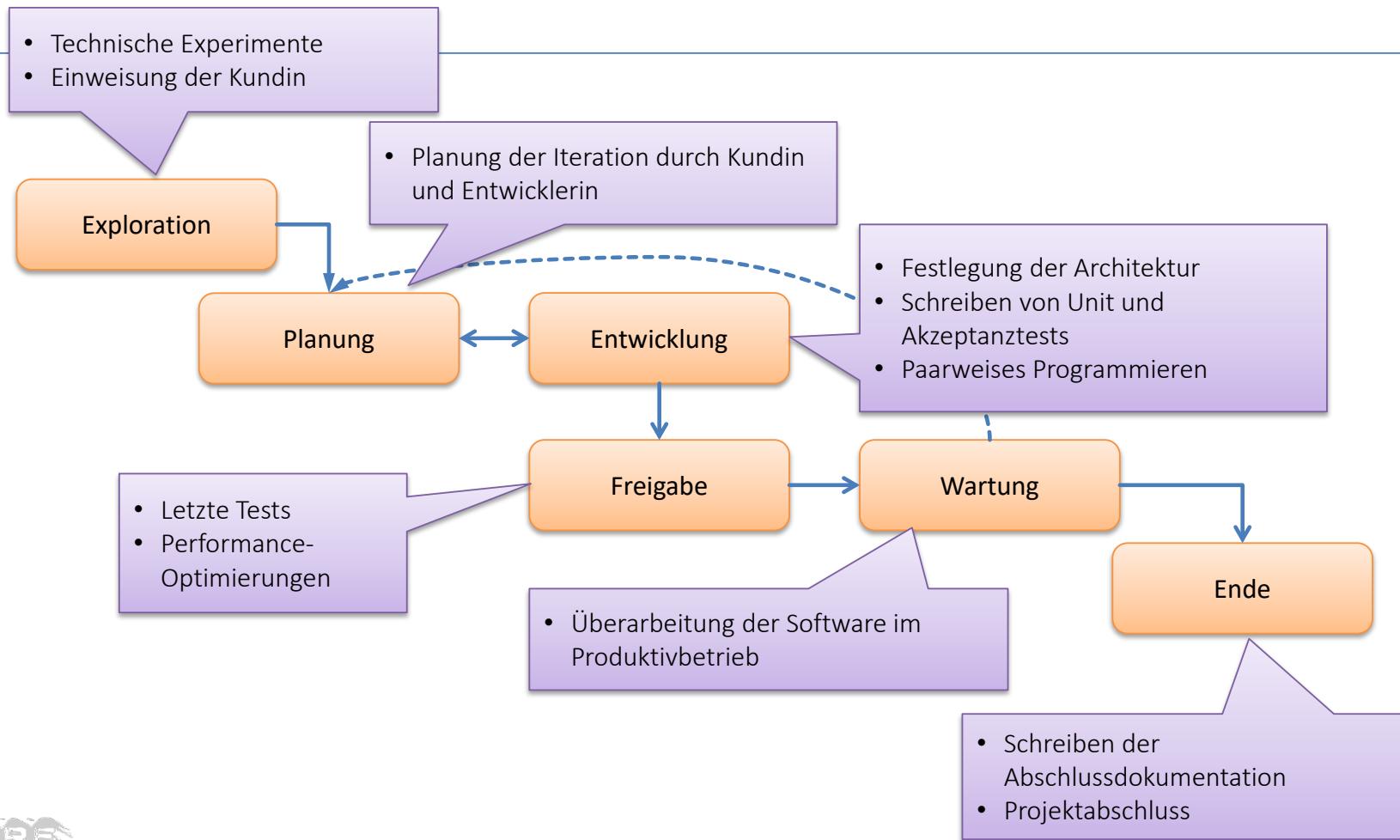
- Menge von Methoden (Praktiken), um qualitativ hochwertige Software zu entwickeln



Extreme Programming: Grundsätze

- Iterative Entwicklung (getrieben durch neue oder geänderte Features)
- Wenig Analyse- und Entwurfstätigkeiten, nur rudimentäre Spezifikationen, selbst-dokumentierender Code ("simple design")
- Frühes Programmieren, prototypisches Umsetzen einzelner "stories"
- Testfälle stehen am Anfang und ersetzen Spezifikation ("test first")
- Ständige Kommunikation der Entwicklerin mit Management und Benutzenden, kurze Rückkopplungsschleifen, schnelle Rückmeldungen
- Schrittweise Änderungen, schrittweise angepasste Tests ("refactoring"), fortlaufende Integration ("continuous integration")
- Fahrerin-/Beifahrerin-Prinzip beim Programmieren ("Pair programming"); schnelle Code Reviews
- Gemeinsame Standards aller Entwicklerinnen, gemeinsames Eigentum am Code ("collective code ownership")

Ablauf der Entwicklung im XP



Planspiel / User Stories

- User stories sind ein oder mehrere Sätze in allgemeiner Sprache aus Sicht der Kundin / Nutzerin der Software, welche eine benötigte Funktion oder Verwendung des Systems beschreibt.
 - Art Anforderungen und benötigte Funktionen auszudrücken
- Beschreibt *Wer, Was, Warum* einer Anforderung in einer einfachen Art und Weise.



As a librarian, I
want to be able
to search for books
by publication year.

Bewertung/Anwendbarkeit des XP

- Nur für *kleine bis mittlere Teams* ausgelegt (bis zu 15 Entwicklerinnen)
- Erfordert *hochqualifizierte* Mitarbeitende
- Die XP-Praktiken sind untereinander *stark verketten*
- Erzwingt ein „*Alles oder Nichts*“-Vorgehen
- Wenn kein Kunde vor Ort möglich, da keiner existent (Software für den Massenmarkt), sollten ausgewählte Teammitglieder diese Rolle *Vollzeit* übernehmen

Test-driven Development

- Erst die Testfälle, dann den Code
- Hauptsächlich bei Unit Tests
- Vorteile:
 - Entwickler nimmt mehr Sicht des Anwenders an
 - Denken in kleinen Modulen, die getestet werden können, dadurch bessere Modularisierung
- Probleme:
 - Schwer bei kompletten Funktionstest (z.B. Usability, Datenbankanbindungen)
 - Testen und Implementierung durch denselben Entwickler

Read-Me-driven Development

- Fokus auf Anwendung des Softwaresystems, um richtiges Produkt zu bauen
- Zuerst die Read-Me schreiben, dann erst mit Testfällen, Design, usw. anfangen

Behavior-driven Development

- Verhalten von Software wird beschrieben
- Dabei Schlüsselwörter, die Vorbedingungen und Nachbedingungen beschreiben
- Implementierung der Software anhand der Szenarien

Lean Software

Eine Variante von Agile

Idee: Verschwendungen Eliminieren

Müll / Verschwendungen ist es, eine Aktivität auszuführen, auf die verzichtet werden kann, ohne das sich etwas am Ergebnis ändert.

Fokus auf Verschlankung von Produkten und Prozessen:

- Nur wertvolle Features berücksichtigen
- Inkrementelle Entwicklung (Probieren statt Dokumentieren)
- Entscheidungen (z.B. über Features) nach hinten schieben
- Möglichst schnell liefern, um sofortige Rückmeldung zu bekommen
- Verantwortung an Team übergeben
- Integrität des Kunden erhöhen (Endnutzer Testing)
- Das große Ganze sehen (besseres Verständnis von Szenarien)
- Qualität einbauen (CI, Test-Driven-Development, etc.)
- Wissen schaffen (mit wissenschaftlichen Methoden lernen, um beste Alternative zu finden)

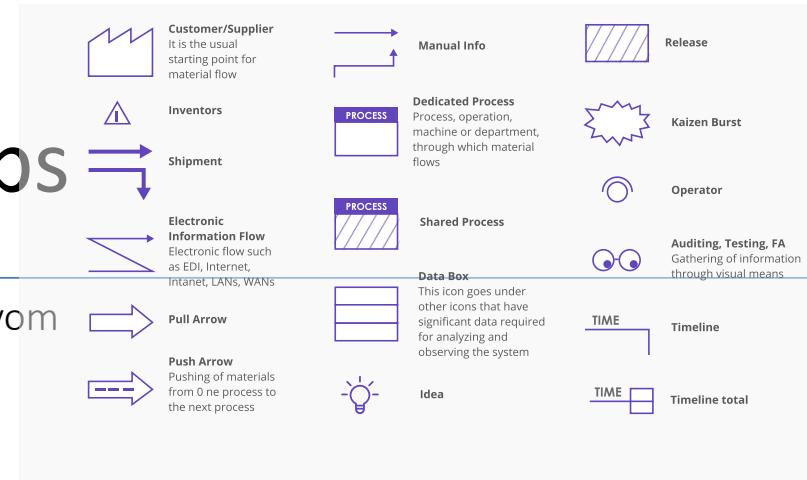


<https://www.bund-berlin.de>

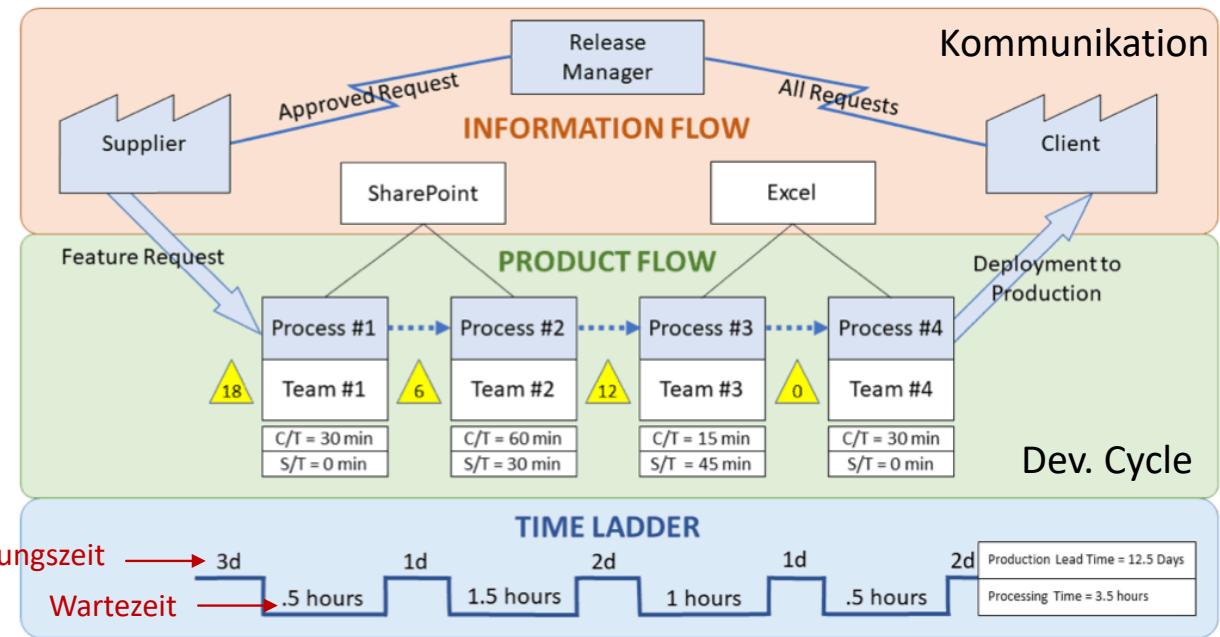
Value-Stream Maps

Methode, um derzeitigen Stand zu ermitteln, wie Informationen und Materialien vom Beginn des Produktionsprozesses bis zum Kunden fließen.

Neben Kanban, eine der wichtigsten Visualisierungen im Lean SW Development



Simplified Value Stream Map for Software Development

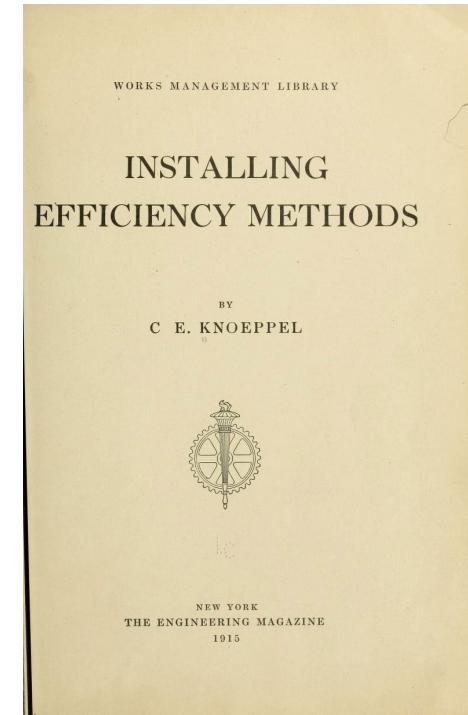


Weiterführende Infos: <https://www.plutora.com/blog/value-stream-mapping>

Historie: Toyota und Knoeppel



Zweiten Hälfte des 20. Jahrhunderts von Toyota
erfolgreich im Produktionssystem eingeführt



INSTALLING
EFFICIENCY METHODS

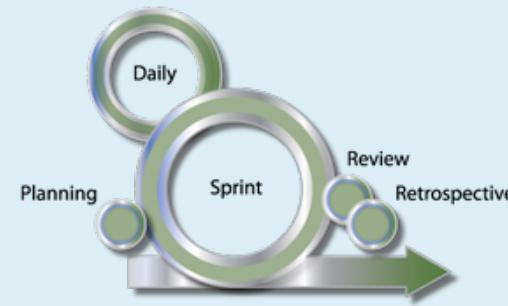
BY
C. E. KNOEPPEL



NEW YORK
THE ENGINEERING MAGAZINE
1915

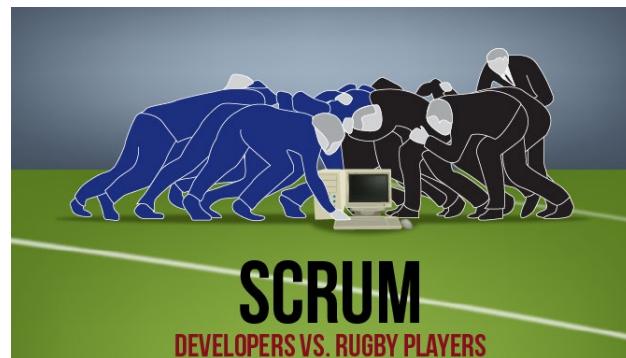
Ursprung: Buch „Installing Efficiency Methods“ von
Charles Knoeppel, 1915.

Scrum – Projektmanagement für agile und lean SW-Entwicklung



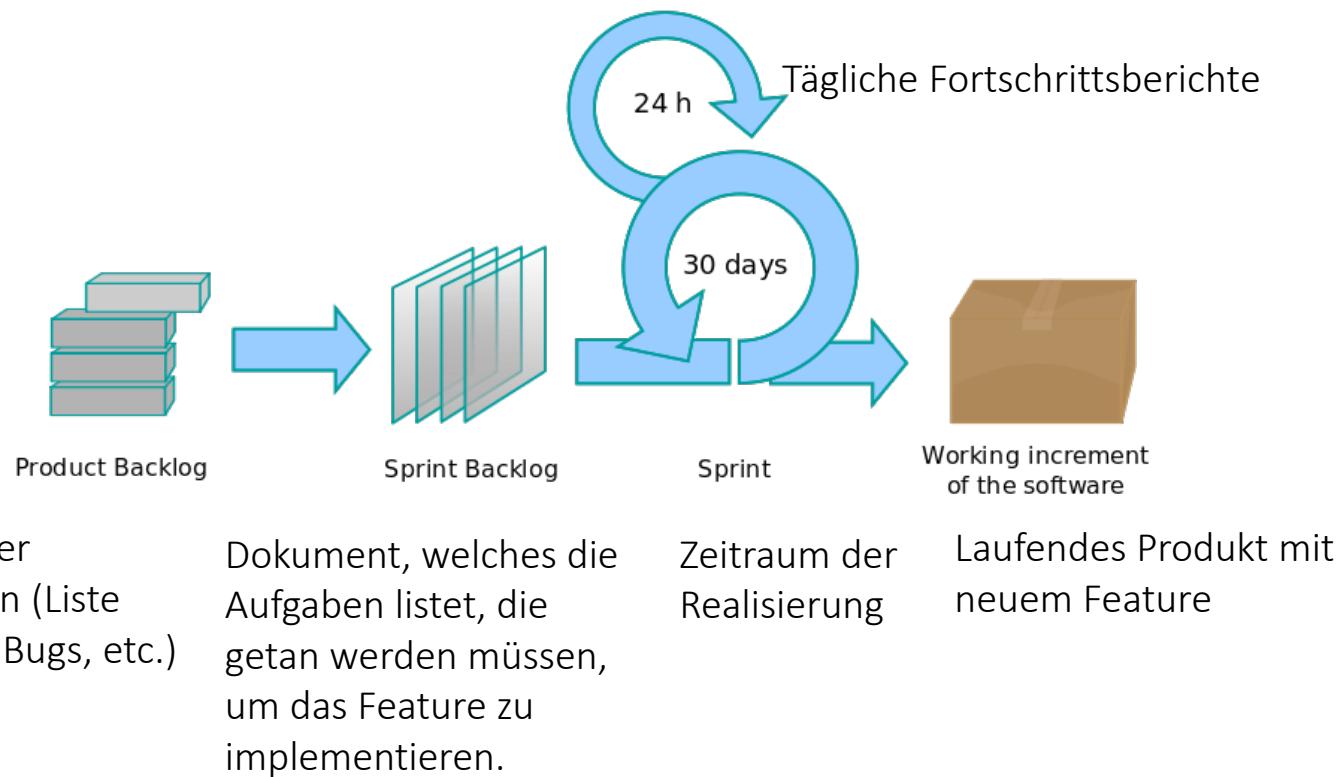
Was ist Scrum?

- SCRUM ist ein *agiles Managementframework*, bei dem Entwicklerteams als eine *Einheit* zusammenarbeiten, um ein gemeinsames Ziel zu erreichen.
- Ziel: Ordnung ins Chaos der Entwicklung zu bringen
 - Flexibel auf Änderungen der Anforderungen durch kurze Entwicklungszyklen reagieren
 - Enthält *keine* Praktiken, die vorschreiben, wie die Software entwickelt werden soll



Leitidee und Ablauf

- Ein Team ist besonders produktiv, wenn es sich mit seinem Produkt identifiziert und auch dafür die Verantwortung trägt.



Prinzipien von Scrum

- Es gibt *keine Projektleiterin*!
 - Team teilt sich selbständig Arbeit auf.
- *Pull-Prinzip*
 - Nur das Team kann entscheiden, wieviel Arbeit in gegebener Zeit geleistet werden kann
- *TimeBox*
 - Es existieren klare zeitliche Grenzen
- Potential *Shippable Code*/Produkt
 - Ergebnis sind immer fertig Produkte

Rollen in Scrum

- Produkt Owner (Die Visionärin, kein Chef!)
 - Pflegt und priorisiert Product Backlog, fachlicher Ansprechpartnerin für Kunden (evtl. bei tägl. SCRUMs dabei)
 - Anforderungsmanagement, Releasemanagement, Kosten und Nutzen Betrachtung
- Das Team (Die Lieferanten)
 - 5-10 Personen meist interdisziplinär
 - Selbst-organisierend, tägl. Meetings
- Scrum Coach*
 - Verantwortung für SCRUM-Prozess
 - Moderiert, vermittelt, optimiert
- Die Kunden (Geldgeber)
- Die Anwender (Benutzen das Produkt am Ende)
- Die Manager (Organisatorische Leitung des Teams)



*wird meist als Master, hergeleitet aus Mastery, bezeichnet

Vorbereitung für Scrum

- Ziel: Erstellen eines Backlogs
- Weg:
 - Produkt Owner erstellt Vision
 - Product Owner und Team erarbeiten gemeinsam Einträge für Backlog
 - **Produkt Owner** priorisiert Backlog-Items
 - **Team** schätzt(!) Aufwand für Backlog-Items

Anforderungen/Features als User Stories

- Anforderungen sollten die INVEST Eigenschaften erfüllen:
 - Independent -> nicht abhängig von anderen Anforderungen
 - Negotiable -> kein Vertrag, Details verhandelbar
 - Valuable -> Wertvoll für die Kundin
 - Estimable -> Für Planung und Ranking
 - Small -> Wenige Personentage, -stunden
 - Testable -> Überprüfbarkeit

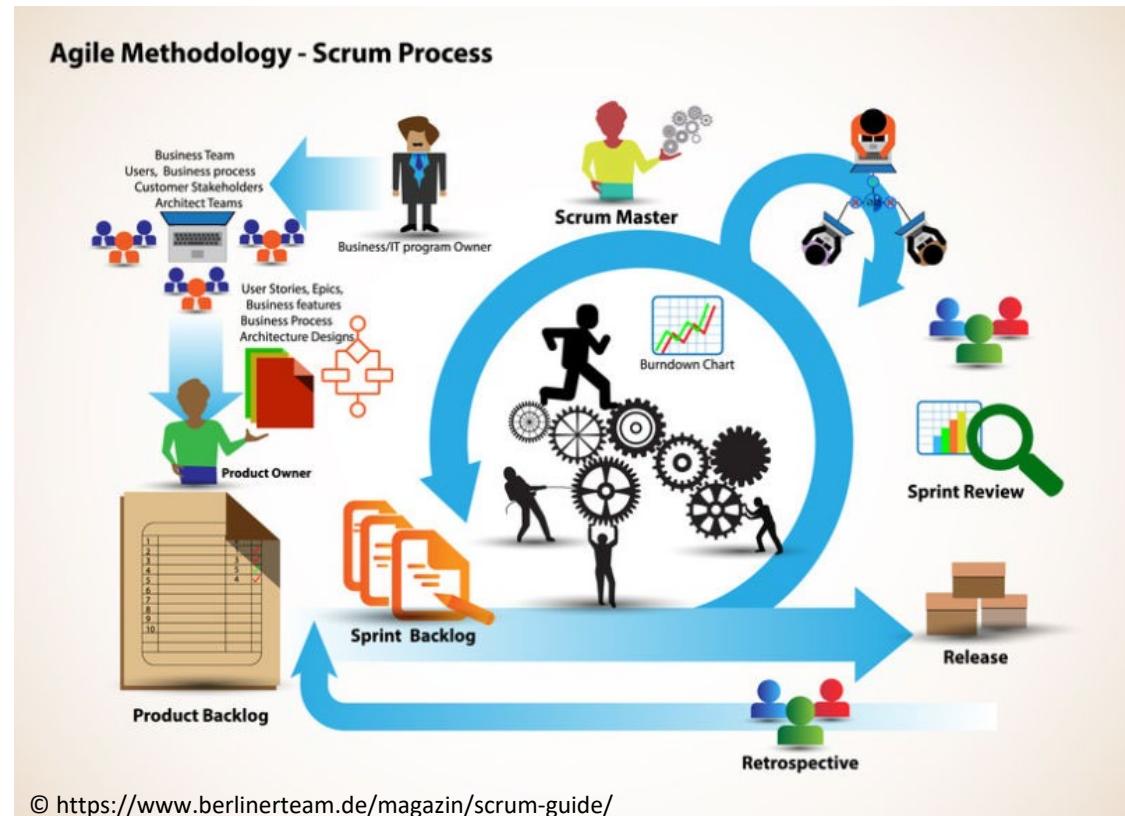
Tasks aus User Stories

- Folgen den SMART-Eigenschaften
 - Specific -> Aufgabe kann verstanden werden
 - Measurable -> Zustand ist überprüf- und messbar
 - Achievable -> Sollte auch Lösbar im Sprint sein
 - Relevant -> Sollte relevant für die User Story sein
 - Time-boxed -> Limitiert in einer Zeitspanne (Stunden oder Tage)

Sprint

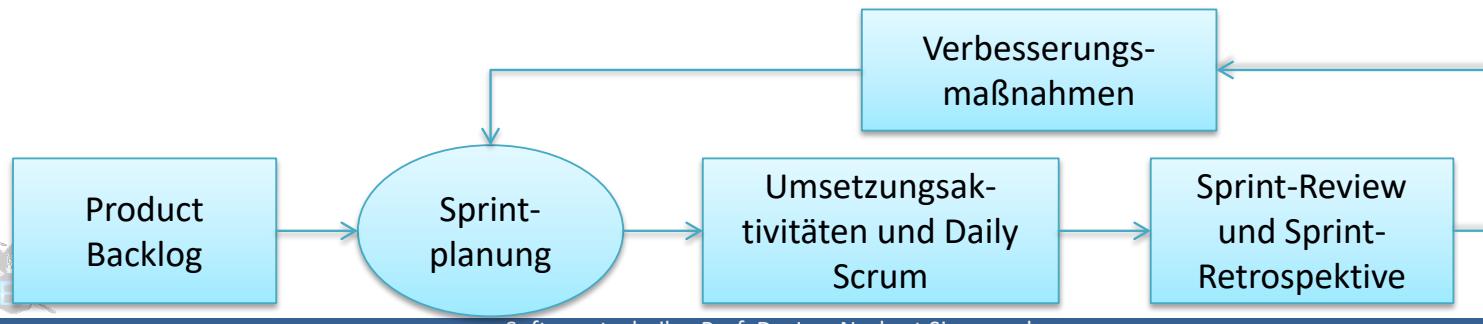
- Ziel: Umsetzung eines Teils des Backlogs in *auslieferbaren Code*

- Phasen eines Sprints
 - Planung
 - Durchführung
 - Abschluss



Sprints

- Wandelt Anforderungen in lauffähige, getestete und dokumentierte Software um
 - Overhead für Scrum-spezifische Aktivitäten $\leq 10\%$
- Vorgehen ist iterativ und inkrementell
 - Agile Entwicklungspraktiken (z.B. TDD) einsetzbar aber nicht festgelegt
- Während des Sprints keine Änderung an dessen Dauer, den Anforderungen im Sprint Backlog und der Teamzusammensetzung



Daily Scrum

- Ablauf:
 - Jedes Mitglied wählt Tagesaufgabe selbst
 - Jedes Mitglied informiert über eigenen Fortschritt
 - Jedes Mitglied berichtet über Blockaden und aufkommende Probleme
- Bedingungen:
 - Teamgröße i.d.R. nicht mehr als 8 Personen
 - 15-Minuten-Regel (SCRUM-Master moderiert)
 - Bei größeren Projekten „SCRUM of SCRUMs“

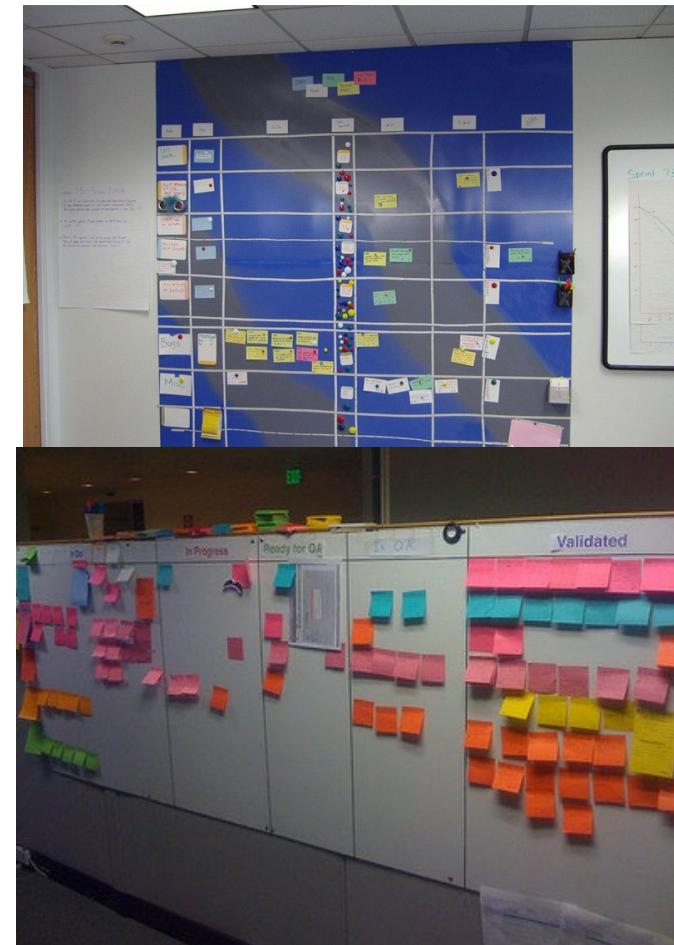
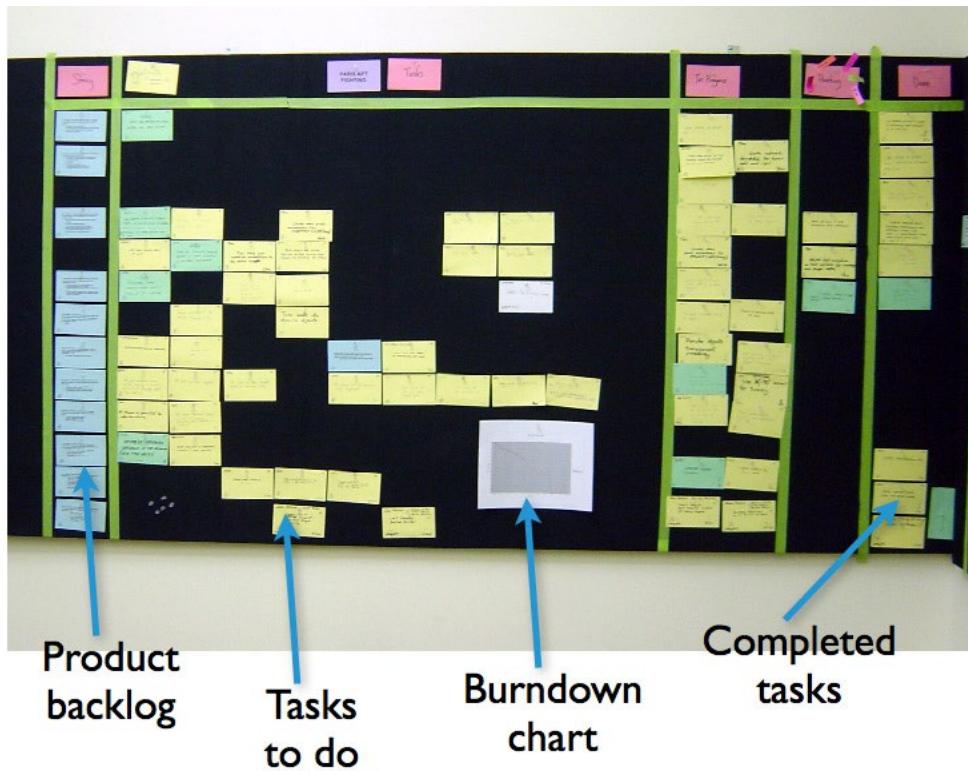


Scrum – Task Board

- Entwicklerinnen heften neue Karten an und verschieben sie selbstständig (oft während / nach Daily Scrum)

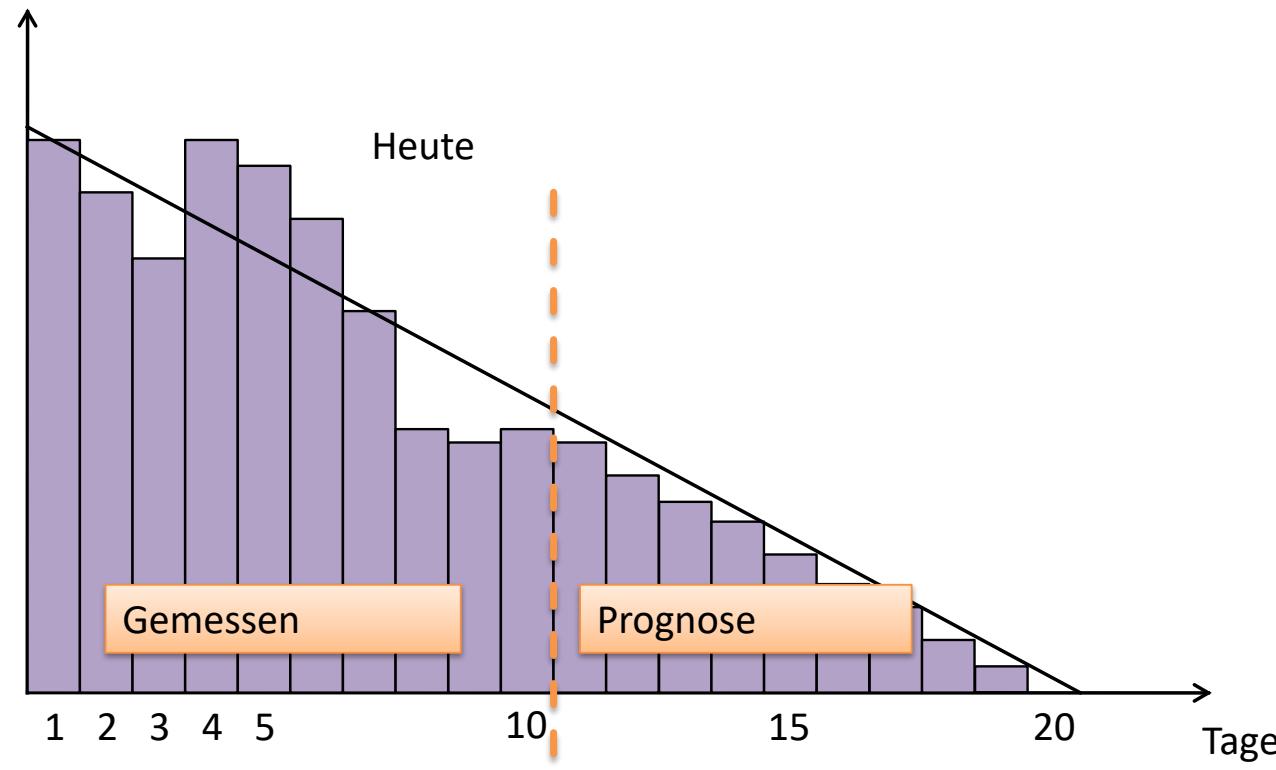
Story	To Do	In Process	To Verify	Done
As a user, I... 8 points	Code the... 9 Code the... 2 Test the... 8	Test the... 8 Code the... 8 Test the... SC 8	Code the... DC 4 Test the... SC 6	Code the... D Test the... SC 8 Test the... SC Test the... SC Test the... SC 6
As a user, I... 5 points	Code the... 8 Code the... 4	Code the... DC 8	Test the... SC Test the... SC Test the... SC 6	

Beispiele



Sprint Burndown Charts

Offene Aufwände im
Sprint Backlog

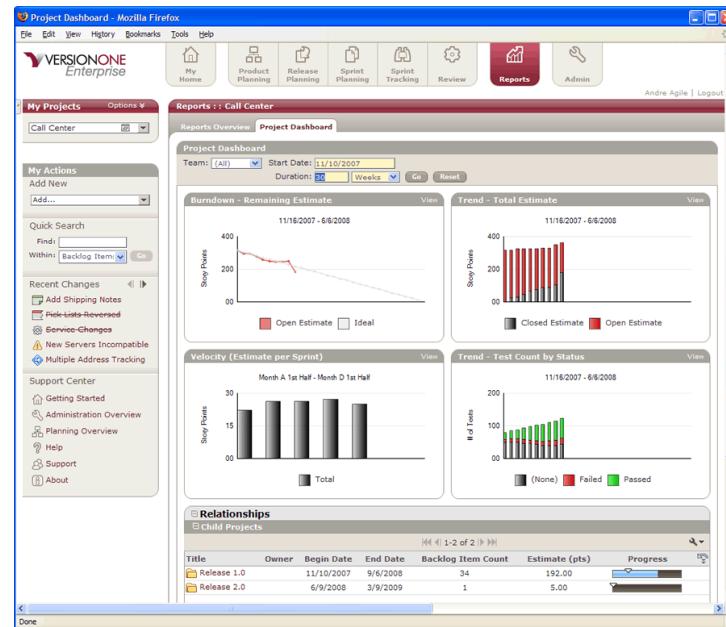


Sprint-Abschluss

- Estimation-Meeting
Ziel: *Aktualisierung* und *Anpassung* des Product Backlogs
Teilnehmer: PO, SM, Team
- Sprint Review
Ziel: Präsentation der *fertigen* Product-Backlog-Items
- Sprint Retrospektive
Ziel: Verbesserung der *zukünftigen* Planung von Sprints

Scrum – Überblick

- *Team* steht in Vordergrund und trägt Verantwortung
- *Tägliche* feste Meetings
- Sprint-Iterationen mit jeweils *shipable code*
- Continuous Improvement Process
 - Schätzungen in JEDEM Sprint anpassen
- Tool-support:
Version One, Rally



Einordnung: Scrum vs. XP

- Scrum ist Projekt Management Methode
 - Spezifiziert den *Prozess* von Idee zum finalen Produkt
 - Unabhängig von der Entwicklungsmethode (z.B. Wasserfall)
- XP ist Entwicklungsmethode
 - Definiert wie Software agil entwickelt wird
- Wenn Scrum für SW-Entwicklung eingesetzt wird, dann ähnelt es XP
- Kombiniert man beide erhält man:
 - Sprints, Artefakte, etc. von Scrum
 - Methoden der SW-Entwicklung (Pair-Programming, TDD, Refactoring, etc.) von XP

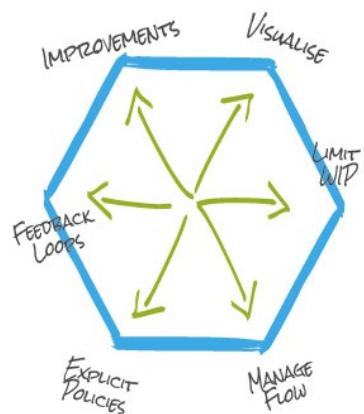
Kanban

Die Alternative zu Scrum



Ziel: Minimiere Laufende Arbeiten

- Kanban (japanisch für Hinweistafel, Anschlagswand, etc.)
- Visuelles Prozessmanagement
- Aus Automobilindustrie (Toyota) um „Just-In-Time Production“ sicher zu stellen



<https://www.agil8.com/blog/six-practices-kanban/>

Visuelles Projektmanagement

Pool of Ideas	Feature Preparation		Feature Selected	User Story Identified	User Story Preparation	User Story Development	Feature Acceptance	Deployment	Delivered
Epic 431	In Progress 3 - 10		Ready	2 - 5	30	In Progress 15	Ready	In Progress 15	Ready (Done) 5
Epic 478	Epic 444	Epic 662	Epic 602			Story 602-02 Story 602-03	Story 602-06 Story 602-04	Story 602-05 Story 602-01	Epic 294
Epic 562	Epic 589		Epic 302	Story 302-03 Story 302-02 Story 302-06	Story 302-07 Story 302-08	Story 302-09	Story 302-05 Story 302-04	Epic 694	Epic 386
Epic 439	Epic 651		Epic 335	Story 335-09 Story 335-08 Story 335-01 Story 335-03	Story 335-05 Story 335-02	Story 335-06	Epic 577	Epic 276	Epic 419
Epic 329			Epic 512	Story 512-04 Story 512-05 Story 512-07 Story 512-06 Story 512-03	Story 512-01			Epic 339	Epic 388
Epic 287								Epic 521	Epic 287
Epic 606								Epic 582	Epic 274
Discarded									
	Epic 511	Epic 213							
	Epic 221								

Policy

Business case showing value, cost of delay, size estimate and design outline.

Policy

Selection at Replenishment meeting chaired by Product Director.

Policy

Small, well-understood, testable, agreed with PD & Team

Policy

As per "Definition of Done" (see...)

Policy

Risk assessed per Continuous Deployment policy (see...)



By Andy Carmichael - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=55448101>

Kernpraktiken I

- Visualisiere:
 - Geteiltes Kanban-Board für alle Aufgaben, Prozessschritte und Mitarbeiterinnen (siehe Folie zuvor)
- Kanban Limits: Limitiere „Work in Progress“
 - Begrenze Anzahl an Tickets pro Prozessschritt
 - Fokussiere auf die produktivste Aufgabe im Projekt
 - Pull-Ansatz: Nimm Item aus vorheriger Spalte
 - Reduziere Context-Switches
- Manage Flow:
 - Messe die Länge der Warteschlange, Mittlere Durchlaufzeit
 - Identifiziere Flaschenhälse zur besseren Planung

Kernpraktiken II

- Explizite Vorgaben:
 - Explizite Regeln, Vorgaben für alle über Bedeutung der Prozesse, wann der Transfer stattfindet in eine andere Spalte, etc.
- Feedback Loops:
 - Kontinuierliche Verbesserung durch Einbau von Mentoring, nicht zeitlich festgelegte Retrospektiven (auch zwischen Teams)
- Verbesserungen:
 - Keine neuen Rollen und Verantwortlichkeiten am Anfang benötigt, aber kontinuierliche, inkrementelle Verbesserungen als Ziel, die auch Organisationen verändern können

Gemeinsamkeiten mit Scrum

- Beide Projektmanagementmethoden unterstützen das Agile Manifesto und Lean Software Development
 - Reagieren flexibel auf Änderungen
 - Fokus auf Qualität, klarer Zeitpunkt der Fertigstellung
-
- Beide verwenden das Pull-Prinzip:
 - Scrum für Projektplanung in Kanban für das Board
 - Selbstorganisierende Teams
 - Inkrementelle Softwareentwicklung mit auslieferbarer SW

Unterschiede Generell

- Kanban
 - Konzentration auf Visualisierung von Aufgaben
 - Begrenzen von laufenden Aufgaben
 - Optimierung der Effizienz
 - Projekt (oder User Story) möglichst schnell abschließen
- Scrum
 - Fixe Intervalle zur Auslieferung neuer Softwareinkemente
 - Direktes und schnelles Feedback von Kunden und Lernschleife
 - Regelmäßige Zeremonien, feste Rollen zur Einbeziehung des Kunden

Unterschiede zu Scrum I

Teams	Scrum	Kanban
Rollen	Fest (Product Owner, Scrum Coach)	Keine initialen Rollen vorgeschrieben
Größe	3-9 Mitglieder, cross-funktional, kollaborativ	Keine Vorgaben, cross-funktional oder spezialisiert

Meetings	Scrum	Kanban
TimeBox	Daily standup	Keine Vorgaben
Austausch	Team-Retrospektive nach Sprint	Keine Vorgaben
Steering	Review-Meeting nach Sprint	Keine Vorgaben
Forecast	Preplanning vor Sprint	Regelmäßige Nachschubmeetings

Unterschiede zu Scrum II

Artefakte	Scrum	Kanban
Anforderungen	Product Backlog	Backlog auf Board
Lieferzyklus	Sprint	Durchlaufzeit des Tickets
Board	Scrumboard (neu bei jedem Sprint)	Kanban-Board (dauerhaft)
Lieferung	Potential auslieferbares Produkt	Abgearbeitetes Ticket
Metriken	Velocity	Lead Time, Cycle Team, etc.

Kanban eher für Wartung und Evolution, wo keine komplexe Lösung erforderlich ist und kontinuierlich Verbesserungen erreicht werden sollen.

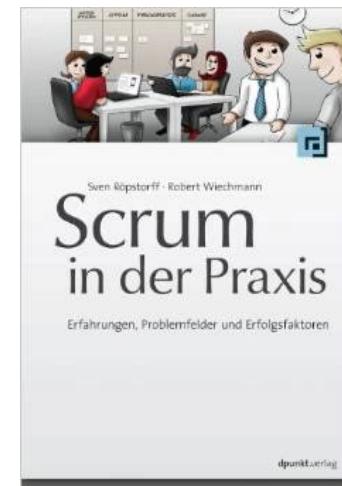
Scrum konzentriert sich auf komplexen SW-Entwicklungsprozess, um größere SW mit interdisziplinären Teams zusammen mit der Kundin zu entwickeln

Was Sie mitgenommen haben sollten:

- Nennen/Erläutern Sie X Softwareentwicklungs-prozesse
- Stellen Sie einen sequentiellen und einen iterativen Softwareentwicklungsprozess gegenüber
- [Anwendungsszenario]
 - Welchen Softwareentwicklungsprozess würden Sie einsetzen? Warum?
 - Horizontaler oder vertikaler Prototyp? Warum?
 - Iterativer oder sequentieller Prozess? Warum?
- Was sind die Eigenschaften von Scrum und wann würden Sie es einsetzen?
- Kennen Sie die Unterschied von Scrum und Kanban, um bei einem gegebenen Szenario die vielversprechendste Methode einzusetzen?
- Worin liegt der Unterschied zu Extreme Programming?

Literatur

- Sommerville: Software Engineering.
- Ludewig and Licher: Software Engineering: Grundlagen, Menschen, Prozesse, Techniken
- Kent Beck: eXtreme Programming Explained: Embrace Change. Addison-Wesley Pub Co; ISBN: 0201616416; 1st edition (October 5, 1999)
- Mik Kersten: Project to Production. (Flow Framework)



Literatur 2

- Agile!: The Good, the Hype and the Ugly. Bertrand Meyer. Springer Publishing Company, Incorporated, 2014. ISBN 9783319051543.
- Stop Starting, Start Finishing!. Arne Roock. Illustrated Edition. Blue Hole Press, 2012. ISBN 9780985305161.
- Kanban: Successful Evolutionary Change for Your Technology Business. David J. Anderson. Blue Hole Press, 2010. ISBN 0984521402.
- Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise. Mark Lines, Scott W. Ambler. IBM Press, 2012. ISBN 978-0-13-281013-5

Softwaretechnik

Projektmanagement



**SOFTWARE
SYSTEME**

Prof. Dr.-Ing. Norbert Siegmund
Software Systems



UNIVERSITÄT
LEIPZIG

Inhalt

- Aufgaben des Projektmanagements
 - Projektplanung
 - Projektzeitplan
 - Reagieren auf Terminprobleme
- Risikomanagement

Lernziele

- Aufgaben des Projektmanagements verstehen
- Nötiges Wissen über die Formalitäten von Projekten erfahren
- Zeitpläne für Projekte erstellen
- Grundlegendes Verständnis für Risikomanagement haben

Warum Projektmanagement?

- Viele Softwareprodukte wurde innerhalb eines *Projektes* erstellt (im Gegensatz zum produzierenden Gewerbe)

Projektherausforderung = *rechtzeitige Auslieferung* im *festgelegten Budget*

- Kernmerkmale eines Projektes
 - Zeitlich abgeschlossen
 - Definiertes Ziel
 - Einmaliges Unterfangen
 - Ressourcenbeschränkung

Was ist Projektmanagement?

Project Management = *Plan the work* and *work the plan*

- Managementfunktionen:
 - *Planung*: Abschätzung und zeitl. Einteilung von Ressourcen
 - *Organisation*: Wer macht was?
 - *Mitarbeiterinnen*: Rekrutierung von motivierten Personen
 - *Dirigieren*: Sicherstellung, dass das Team zusammenarbeitet
 - *Monitoring* (Controlling): Erkenne Abweichungen im Plan und korrigiere Aktionen

Arten von Projekten

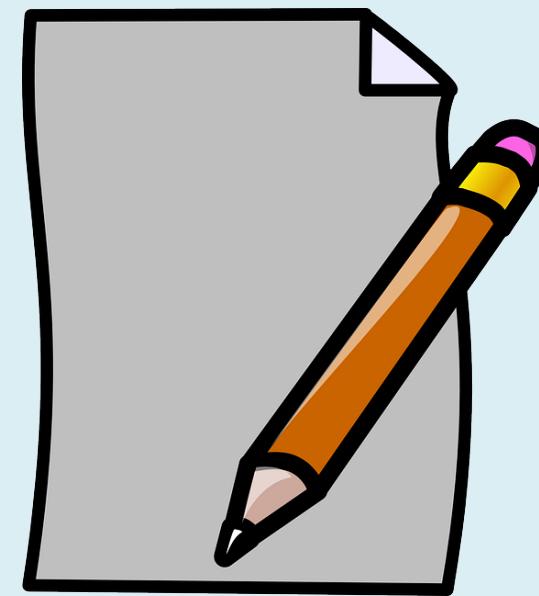
- Entwicklungsprojekt („Wir bauen SW als Produkt“)
 - Auftraggeberin und Auftragnehmerin i.d.R. Teil derselben Organisation
 - Nutzende sind in der Regel außerhalb der Organisation
- Auftragsprojekt („Wir bauen SW als Produkt für Auftraggeber“)
 - Auftragnehmerin und Auftraggeberin strikt getrennt
 - Auftraggeberin nicht unbedingt gleich Nutzende
- EDV-Projekt
 - Nutzende, Auftragnehmerin und Auftraggeberin sind Teil der gemeinsamen Organisation
 - Gemeinsame Vorgesetzte
- System-Projekt
 - Nutzende, Auftraggeberin und Auftragnehmerin evtl. vermischt
 - Projektteam nur teilweise Teil der Organisation
 - Stark unterschiedliche Kompetenzen



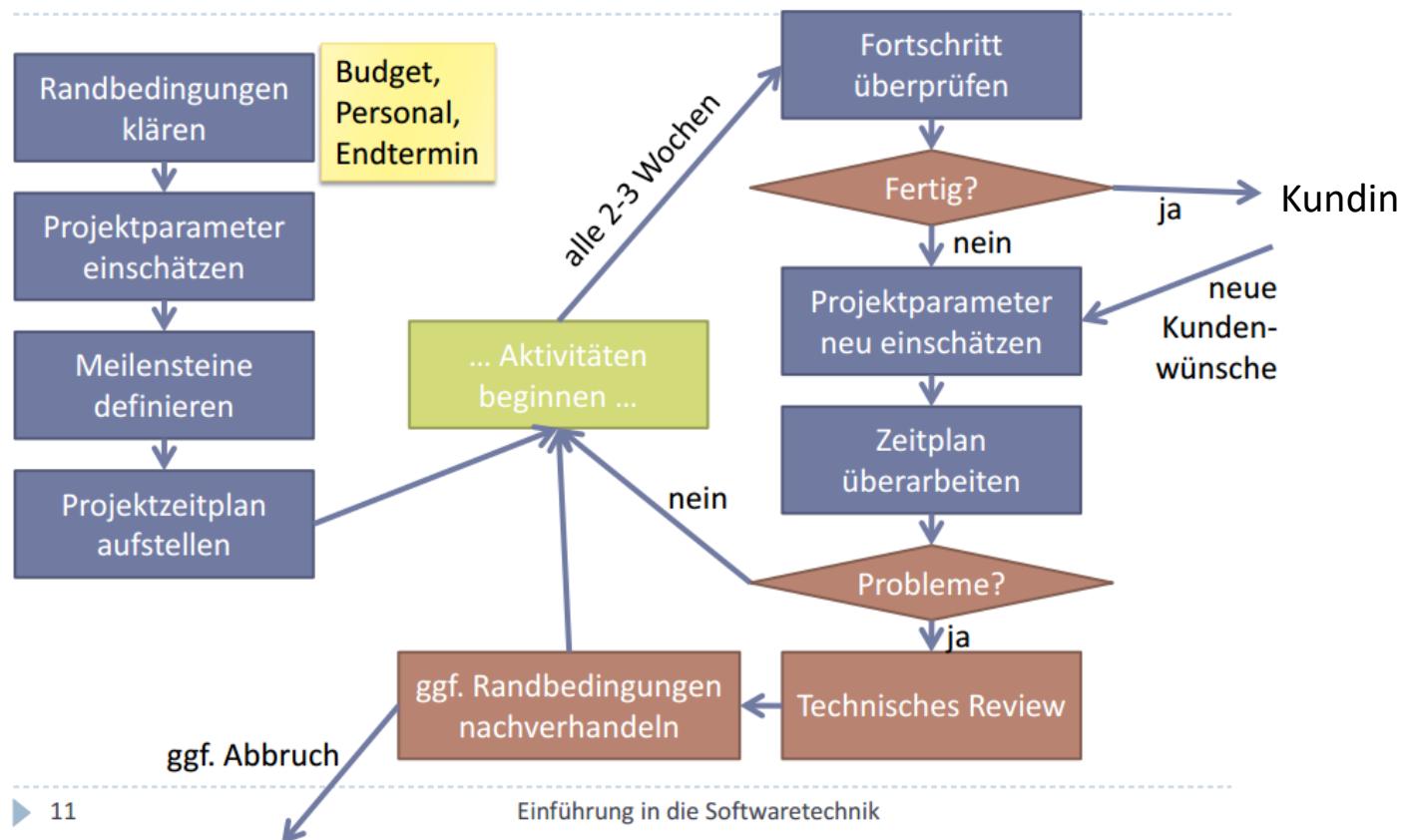
Aufgaben während Projektmanagement

- Projektantrag
- Projekt- und Zeitplanung
- Risikomanagement
- Projektkostenkalkulation
- Projektüberwachung
- Auswahl und Beurteilung des Personals
- Präsentation und Erstellen von Berichten

Projektplanung



Projektplanung



Projektplan

- Einführung: Ziele und Randbedingungen festlegen (Pflichtenheft)
- Projektorganisation: Personen, Rollen, Teams
- Risikoanalyse: Beschreibung und Bewertung von Risiken
- Arbeitsaufteilung, Verantwortlichkeiten, Weisungsbefugnisse
- Projektzeitplan: Wer, wann, was? Meilensteine, Lieferschritte
- Überwachungs- und Berichterstattungsinstrumente: Wann und wie wird geprüft und berichtet?

- Der Projektplan wird während des Projekts angepasst

Meilensteine

- Erkennbarer Endpunkt einer Teilaufgabe
- Für Projektmanagerin zur Überwachung/Überprüfung des Fortschritts
- Berichte, Prototypen, fertige Teilsysteme
- Überprüfbarkeit:
 - “Implementierung zu 80% abgeschlossen” kein geeigneter Meilenstein
 - Besser: Anforderung X erfüllt

Lieferschritte

- Projektresultat für Kundin
 - Ähnlich Meilenstein, aber für Kundin
 - Berichte, Prototypen, fertige Teilsysteme
-
- Sollten genau wie Meilensteine etwa alle 2-3 Wochen fällig sein

Lastenheft

- Spezifiziert durch Kundinnen / *Auftraggeberin*
- Beschreibt Sicht der Auftraggeberin
 - Was ist der IST-Zustand und was sind Gründe für das Projekt?
 - Was sind die Ziele des Projektes?
 - Welche Anforderungen gibt es (Katalog, Spezifikation)?
- Wird oft in Ausschreibungen verwendet
- Anforderungen sind sehr allgemein und wenig beschränkend formuliert

Möglicher Aufbau eines Lastenheftes

- Einführung
- Beschreibung des Ist-Zustands
- Beschreibung des Soll-Konzepts
- Beschreibung von Schnittstellen
- Funktionale Anforderungen
- Nichtfunktionale Anforderungen
 - Benutzbarkeit, Zuverlässigkeit, Effizienz, Änderbarkeit, etc.
- Risikoakzeptanz
- Skizze des Entwicklungszyklus und der Systemarchitektur oder auch ein Struktogramm
- Lieferumfang
- Abnahmekriterien

Pflichtenheft

- Spezifiziert durch *Auftragnehmerin*
 - Fasst alle Anforderungen *konkret und vollständig* zusammen
 - Bildet Grundlage für vertraglich festgehaltene Leistungen
 - *Präzisiert* das Lastenheft und beschreibt *wie* die Anforderungen aus dem Lastenheft realisiert werden
- Folgende Punkte sind enthalten:
 - Funktionale Anforderungen (inkl. Datendefinitionen)
 - Nicht-funktionale Anforderungen (Performance, ...)
 - Anforderungen an technische Realisierung (welche HW/OS,...)
 - Anforderungen an Projektablauf (Meilensteine, Risiko,...)
 - Benutzungsschnittstelle (Wie Präsentation)

Zeitplanung



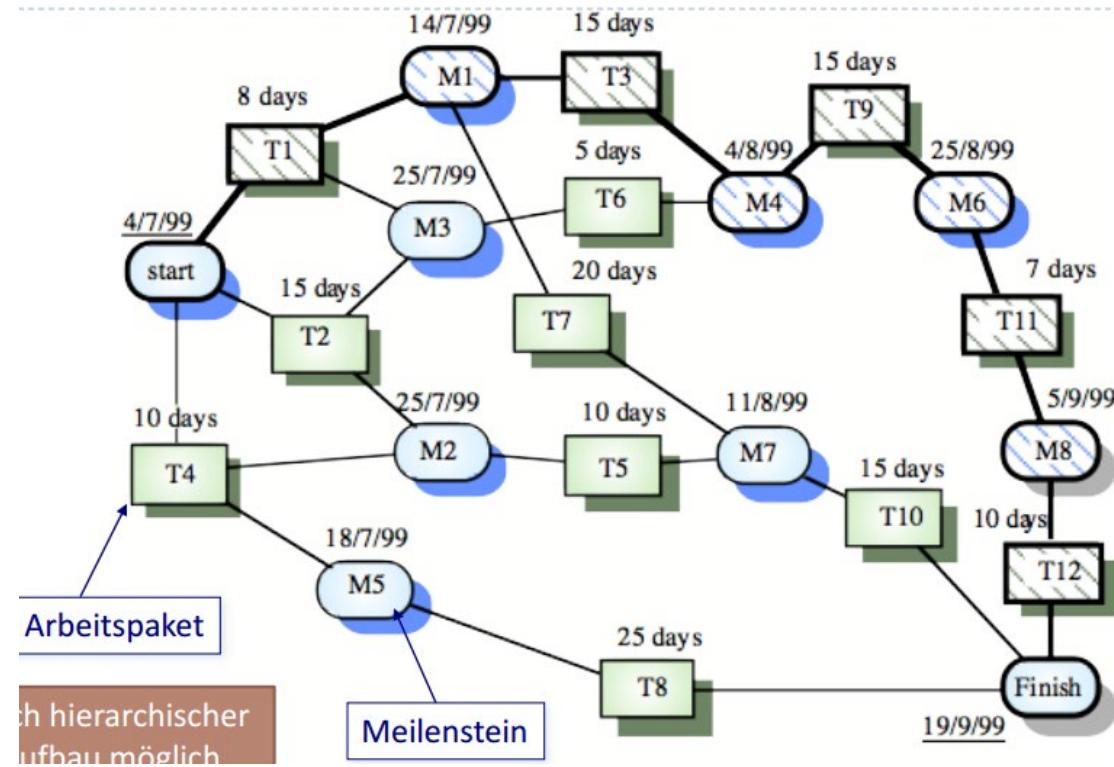
Zeitplanung

- Zerlegt Projekt in Arbeitspakete (Dauer 1 bis 10 Wochen)
- Arbeitspakete klein genug wählen, dass realistische Kostenschätzung möglich ist
- Abhängigkeiten zwischen Arbeitspaketen definieren und minimieren
- Schätzt Zeiten und Ressourcen
- Erstellt sinnvolle Reihenfolge und Parallelität
- Zeitpuffer einplanen, eventuelle Probleme berücksichtigen
- Softwareunterstützung hilfreich, z.B. Microsoft Project, GanttProject, Kplato, uvm.

Was ist die minimale Projektdauer?

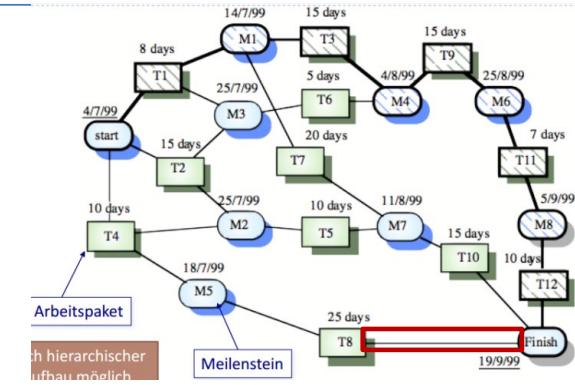
Arbeitspaket	Dauer in Tagen	Abhaengigkeiten
T1	8	
T2	15	
T3	15	T1
T4	10	
T5	10	T2, T4
T6	5	T1, T2
T7	20	T1
T8	25	T4
T9	15	T3, T6
T10	15	T5, T7
T11	7	T9
T12	10	T11

Netzplan

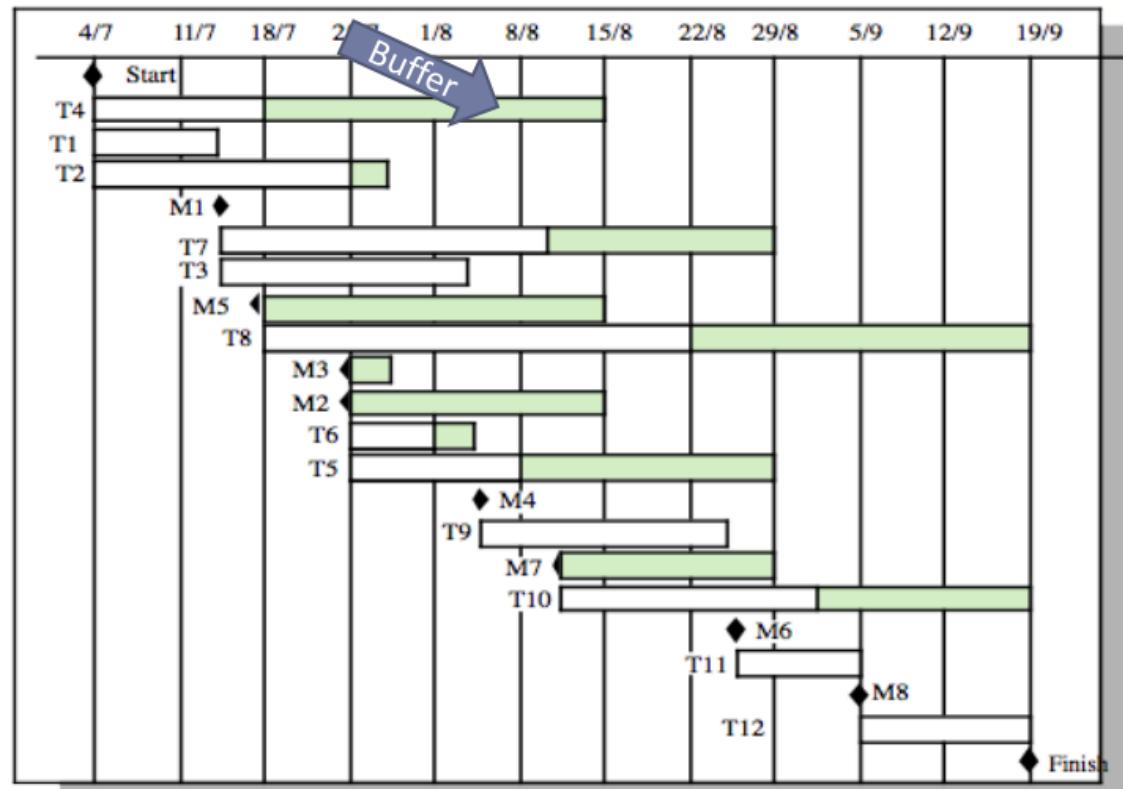


Kritischer Pfad

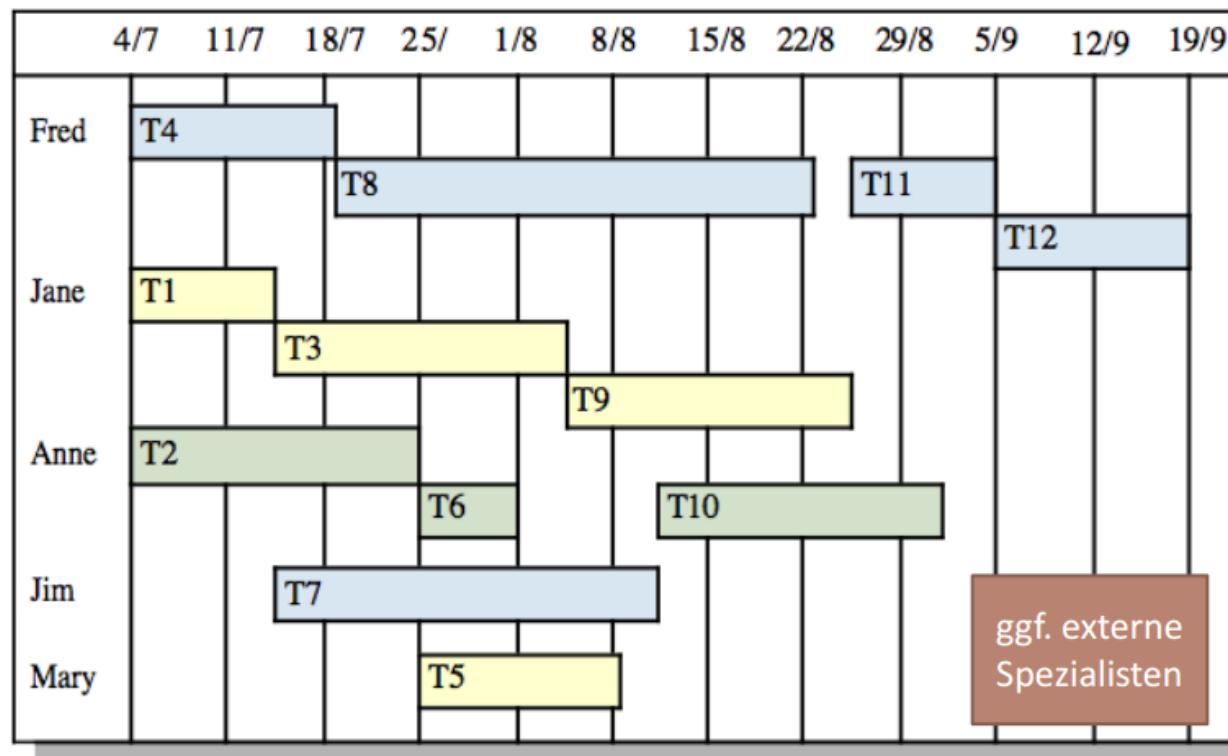
- Längster Pfad im Netzplan:
 - 55 Tage
 - Puffer T8: 20 Tage
- Verzögerung vom Paketen auf kritischem Pfad -> Gesamtverzögerung
 - Dort besonders genau planen
 - Zeiten ggf. verkürzen durch Projektaufgaben umstrukturieren;
 - Pessimistisch planen
- Andere Pakete ggf. unkritisch, berechenbarer Puffer



Gantt-Diagramm



Gantt-Diagramm für Ressourcen

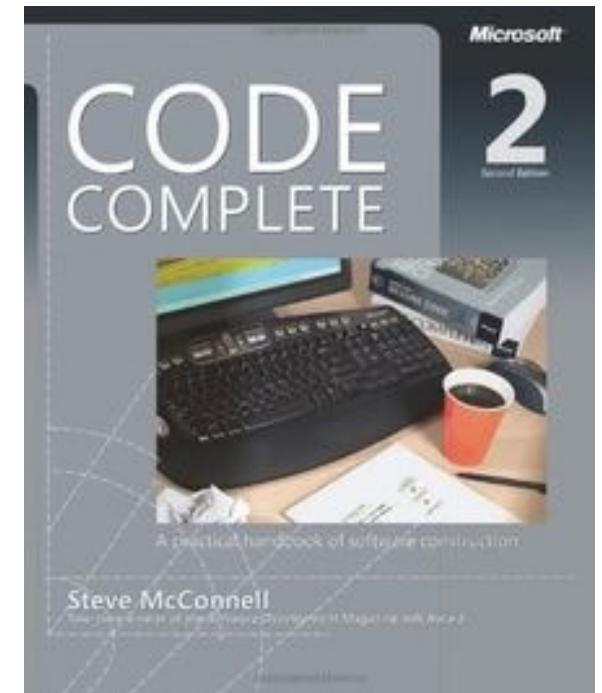


Zeitplanung

- Zeitplan ändert sich ständig
- Erfahrung zum Schätzen notwendig
- Trotzdem schwierig durch Neuartigkeit des Projekts und schnell wechselnde Technologie
- Vergleich mit ähnlichen Projekten zur besseren Zeitplanung (sinnvoll, diese in einer Datenbank zu speichern)

Reagieren auf Zeitprobleme

- *Myth:*
 - “If we get behind schedule, we can add more programmers and catch up.”
- *Reality:*
 - Adding more people typically slows a project down.



Zeitprobleme I

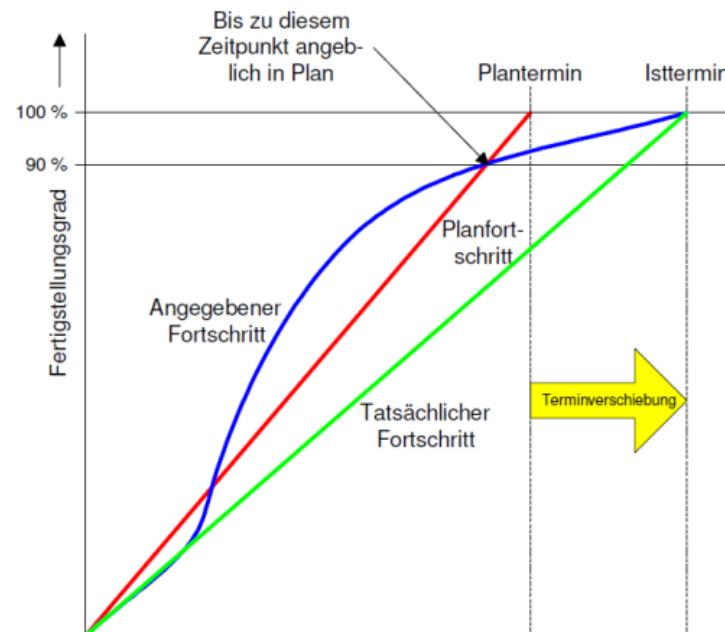
- *Abschätzung* der Schwierigkeit eines Problems und die Kosten für die Entwicklung einer Lösung *ist schwierig*
- Produktivität ist nicht proportional *zur Anzahl der Leute die an einer Aufgabe arbeiten*
- *Hinzufügen von Leuten* in einer späten Projektphase *verlangsamt* das Projekt durch *Kommunikationsoverhead*
- Das *Unerwartete passiert immer.*
- Das Herunterfahren von Testen und Reviews ist ein *Rezept für ein Desaster.*
- *Nachts Arbeiten? Nur ein kurzfristiger Nutzen!*

Zeitprobleme II

- Personalmangel (Krankheit, Fluktuation, ...)
- Fehlende Qualifikation
- Unvorhergesehene Schwierigkeiten
- Unrealistische Aufwandsabschätzungen
- Nicht bedachte Abhängigkeiten
- Zusätzliche Leistungsanforderungen
- Typisch bei Studierendenprojekten:
 - Überraschende Prüfungszeit
 - Ungleichmäßige Arbeitsverteilung
 - Einarbeitungszeit unterschätzt

Fast-schon-fertig-Syndrom

- Letzten 10 % der Arbeit -> 40 % der Zeit
- Fortschritt messbar machen
- Nicht nur auf Schätzungen der Entwicklerin verlassen



Umgehen mit Zeitproblemen

- Welche Möglichkeiten gibt es, mit Zeitproblemen umzugehen?

Umgehen mit Zeitproblemen: Planungsphase

- Berichte eindeutig *was du weißt und was du nicht weißt und warum!*
- Berichte eindeutig *was du planst, um das Unwissen abzustellen*
- Stelle sicher, dass *alle frühen Meilensteine* erreicht werden können
- Zeitprobleme so *früh wie möglich* entdecken
- Plan to *replan*

Umgehen mit Zeitproblemen: Umsetzungsphase

- Einsatz von zusätzlichem Personal, insb. hochqualifiziertes Personal für spezielle Aufgaben
- Temporäres Erhöhen der Arbeitszeit (Überstunden, Urlaubssperre), aber nur kurzfristig möglich
- Verbesserter Tool- und Methodeneinsatz
- Optimierung der Arbeitsabläufe
- Verschiebung der Deadline
- Geringerer Leistungsumfang
 - Prioritäten vergeben, inkrementelles Ausliefern
 - Fertigstellungstermin verschieben

Kostenschätzung und Risiko



Risiken

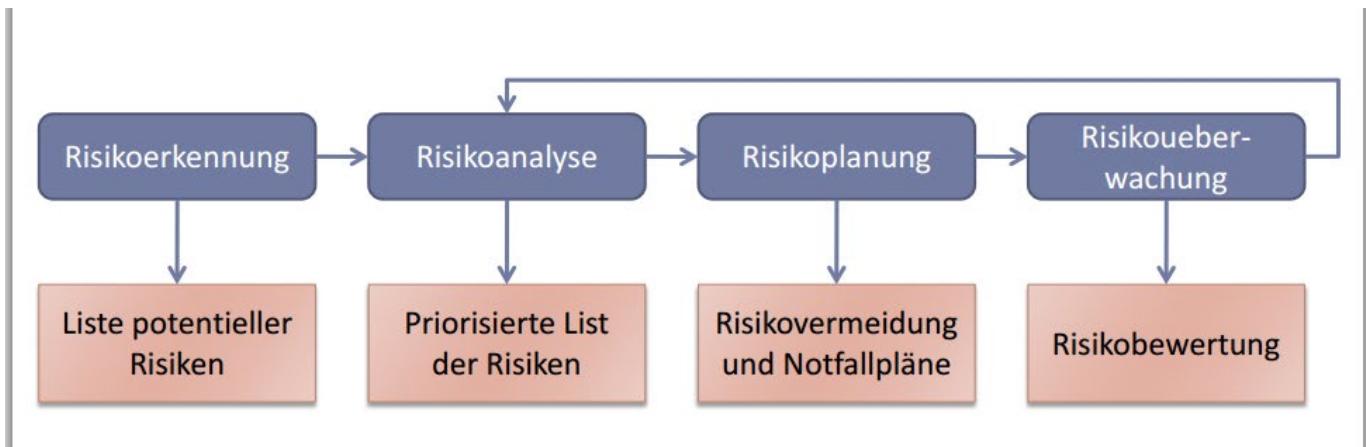
- *"If you don't actively attack risks, they will actively attack you."*
- Projektrisiken: Schedule, Ressourcen, Größe, Personal, Moral, ändernde Anforderungen, ...
- Produktrisiken: Technologien (Implementierung, Sprachen), Verifikation, Wartung, ...
- Businessrisiken: Markt, Verkäufe, Management, Standards, ...

– Tom Gilb

Typische Risiken

Risiko	Art	Beschreibung
Personalveraenderung	Projekt	Erfahrener Personal verlässt das Projekt vorzeitig, Krankheit
Managementveraenderung	Projekt	Neues Management mit anderen Prioritäten
Hardware/Software nicht verfügbar	Projekt	Zulieferung unverzichtbarer Hardware/Software unzureichend
Aenderung von Anforderungen	Projekt und Produkt	Mehr Änderungen als erwartet
Verzögerung in Spezifikation	Projekt und Produkt	Wichtige Schnittstellen nicht rechtzeitig bekannt
Unterschätzung des Umfangs	Projekt und Produkt	
Technologieveraenderung	Wirtschaftlich	Neue Technologie verdrängt benutzte
Produktkonkurrenz	Wirtschaftlich	Konkurrenzprodukt vorher auf dem Markt

Risikomanagementprozess



Risikoerkennung

- Teamarbeit, Ideensammlung, Checklisten
- Beispiele
 - Technologische Risiken: langsame Datenbank, fehlerhafte Komponente
 - Personenbezogene Risiken: Krankheit, unqualifiziertes Personal
 - Unternehmensbezogene Risiken: Managementwechsel
 - Risiken durch Werkzeuge: Code-Generator ineffizient
 - Anforderungsrisiken: Kundin versteht Konsequenzen von Anforderungsänderungen nicht
 - Schätzrisiken: Anzahl der Fehlerbehebungen wird unterschätzt

Risikoanalyse

- Schätzung von Wahrscheinlichkeit und Auswirkungen
- Erfahrung des Projektleiters nötig
- Grobe Skalen reichen
 - gering (<10%), niedrig (<25%), mittel (<50%), hoch (<75%), sehr hoch
 - katastrophal, ernst, tolerierbar, unbedeutsam
- Fokus auf die Top-10-Risiken

Risikoplanung

- Vermeidungsstrategien (Risiko vermeiden)
- Minimierungsstrategien (Konsequenzen minimieren)
- Notfallpläne
- -> *Erfahrung der Projektleiterin nötig*
- Beispiele:
 - Kundinnenakzeptanz unklar: Prototyp entwickeln
 - Krankheit des Personals: Überschneidungen bei Arbeiten einplanen, Abhängigkeiten vermeiden
 - Datenbankleistung: Andere Datenbank kaufen
 - Finanzielle Probleme des Unternehmens: Zusammenfassung an Management, die Beitrag des Projekts erklärt

Typische Strategien im Risikomanagement

- Früh *Prototypen* entwickeln
- *Inkrementelle* Entwicklung
- *Gutes* Personal rekrutieren
- *Teambildende* Maßnahmen
- *Wiederverwendung*, Komponenten einkaufen

Chief Programming Teams (Beispiel)

- Besteht aus einem Kern von Spezialistinnen, die von anderen unterstützt werden
 - Chefprogrammiererin übernimmt *volle Verantwortung für Design, Programmierung, Testen und Installation* des Systems
 - Backup-Programmiererin hält sich über den Stand der Arbeiten aktuell und *entwickelt Testfälle*
 - Bibliothekarin verwaltet *sämtliche Information*
 - Andere Rollen: Projektadmin, Tool-Dev, Doku-Schreiberin, Sprach-/Systemexpertin, Testerin, und Programmiererin,
...
- Erfolgreich, aber mit Problemen:
 - Schwierig einen talentierten Chefprogrammiererin zu finden
 - Kann normale Organisationsstrukturen stören
 - Kann demotivierend für Nicht-Chefprogrammiererin sein

Directing Teams

- *Managerin unterstützen / dienen ihrem Team*
 - Managerinnen stellen sicher, dass das Team alle notwendigen Ressourcen und Informationen besitzt
 - “*The manager’s function is not to make people work, it is to make it possible for people to work*”
- Tom DeMarco
- *Verantwortung erfordert Autorität*
 - Managerinnen müssen **delegieren**: *Vertraue deinen eigenen Leuten und sie werden dir vertrauen*

Was Sie mitgenommen haben sollten:

- Nennen und erklären Sie die Aufgaben einer Projektmanagerin.
- Skizzieren Sie den Prozess zur Projektplanung
- Erklären Sie die Begriffe Meilenstein und Lieferschritt und nennen Sie je ein gutes und ein schlechtes Beispiel. Warum sind diese besonders bei Softwareprojekten notwendig?
- Bestimmen Sie die minimale Projektdauer aus Tabelle X, entweder mit Gantt oder einem Netzplan.
- Nennen/Erklären Sie X typische Zeitprobleme und Techniken, mit diesen umzugehen.
- Nennen/Erklären Sie X typische Risiken und Techniken, mit diesen umzugehen.