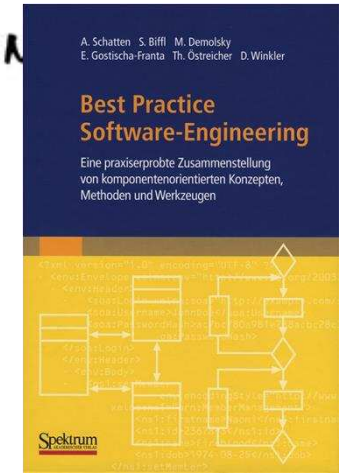
Software Engineering & Projektmanagement
„Einführung in Projektmanagement“



Peter Frühwirt
peter.fruehwirt@qse.ifs.tuwien.ac.at

Ziele der Vorlesung

- **Ergänzung zum SEPM Projekt.**

- **Erlernen wesentlicher Konzepte der Softwareentwicklung.**
  - Technische Grundlagen.
  - Ergänzungen / Erweiterungen zu Inhalten aus der Laborübung.
  - Aktuelle Methoden in der Softwareentwicklung.
  - Software Life Cycle.
  - Vorgehensmodelle.
  - Projektmanagement.

Vorlesungsprüfung

Prüfungsmodus

- Arbeitszeit: 120 Minuten
- Bei der Prüfung sind KEINE UNTERLAGEN zugelassen.
- Wissens- und Verständnisfragen: max. 60 Punkte, Arbeitsaufgaben: max. 40 Punkte (für eine positive Note müssen beide Teile zwingend positiv sein, d.h. jeweils mind. 50% der möglichen Punkte)
- Notenschlüssel: 0-49: N5, 50-62: G4, 63-74: B3, 75-87: U2, 88-100: S1

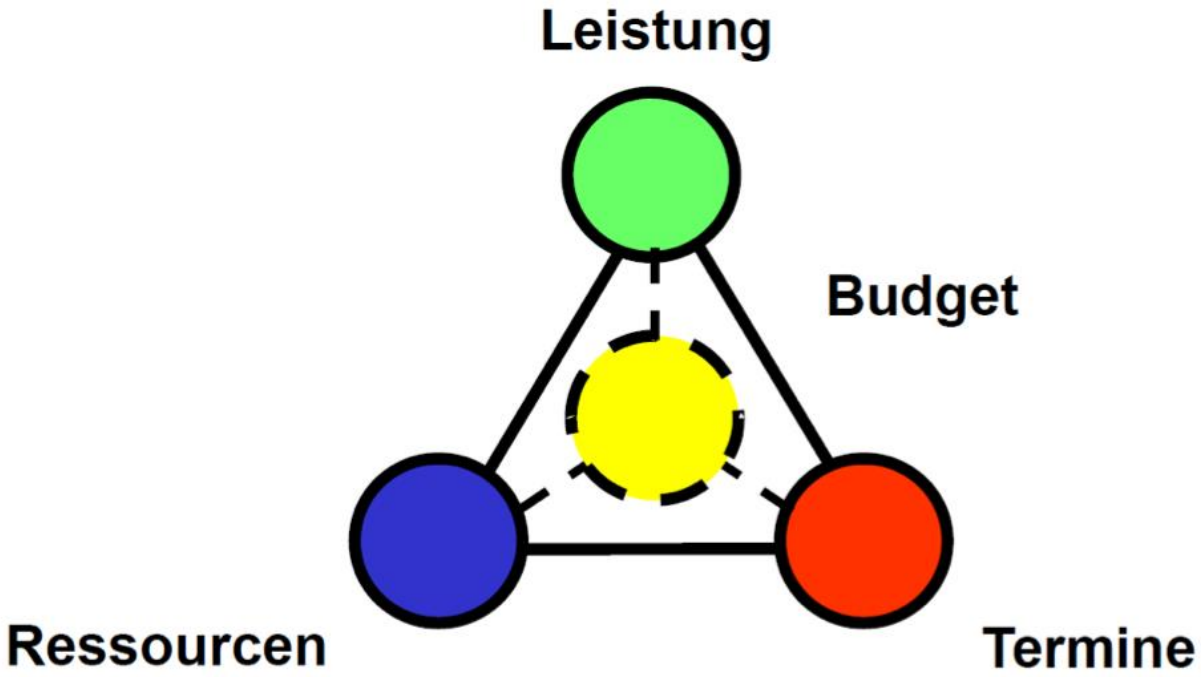
- Haupttermin: 28.06.2018

Was ist ein Projekt?

- „Ein Projekt ist (im Gegensatz zum normalen Tagesgeschäft oder zur Produktion) ein einmaliges Vorhaben mit einem definierten Anfang, einem definierten Ende und mehreren beteiligten Personen.“
- Kennzeichen eines Projekts
  - Ihre Abgrenzbarkeit bezüglich: Aufgabe, Ergebnis, Ressourceneinsatz und Zeitrahmen
  - Komplexität der Aufgabe
  - Einmaligkeit der Aufgabe

Abgrenzbarkeit von Projekten

- Aufgabe / erwartetes Ergebnis
  - Funktionsumfang
  - Leistungsumfang
  - Qualität
- Ressourcen
  - Personaleinsatz
  - Geld- und Sachmittel
- Zeitrahmen
  - Start
  - Ende



Was ist Projektmanagement?

- Projektmanagement ist eine Gesamtheit von Methoden und Verhaltensweisen zur effizienteren Steuerung der Abwicklung von besonderen Aufgabenstellungen (Projekten)

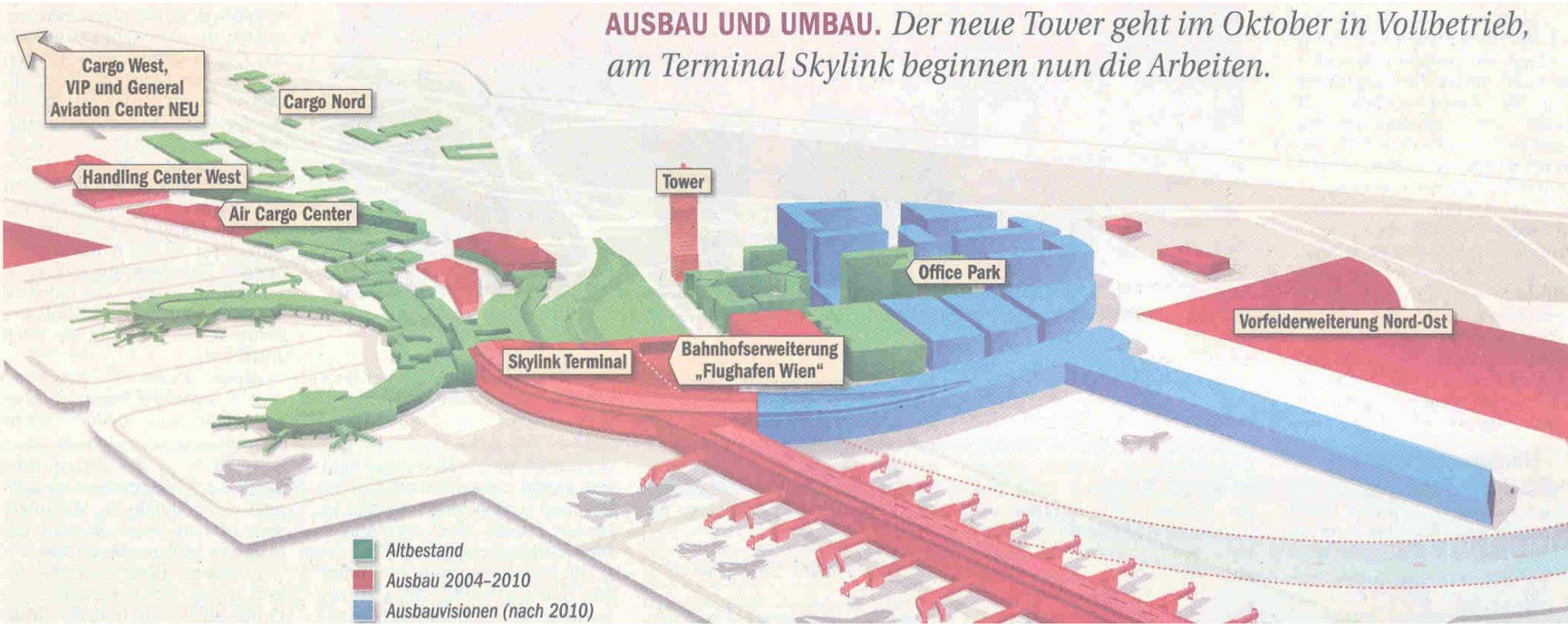
- Projektmanagement im engeren Sinn ist die Projektleitung, d.h. die für die Führung/Steuerung eines Projekts verantwortliche Person/Stelle

Wann setzt man Projektmanagement ein?

- In der Theorie: Wenn die Projektorganisation Vorteile bringt
  - Wenn eine Aufgabe dadurch leichter lösbar wird
  - Wenn sich die Arbeit dadurch besser organisieren lässt
- In der Praxis: Projektorganisation wird oft unreflektiert eingesetzt:
  - Projektinflation
  - Fehlendes Rollenverständnis

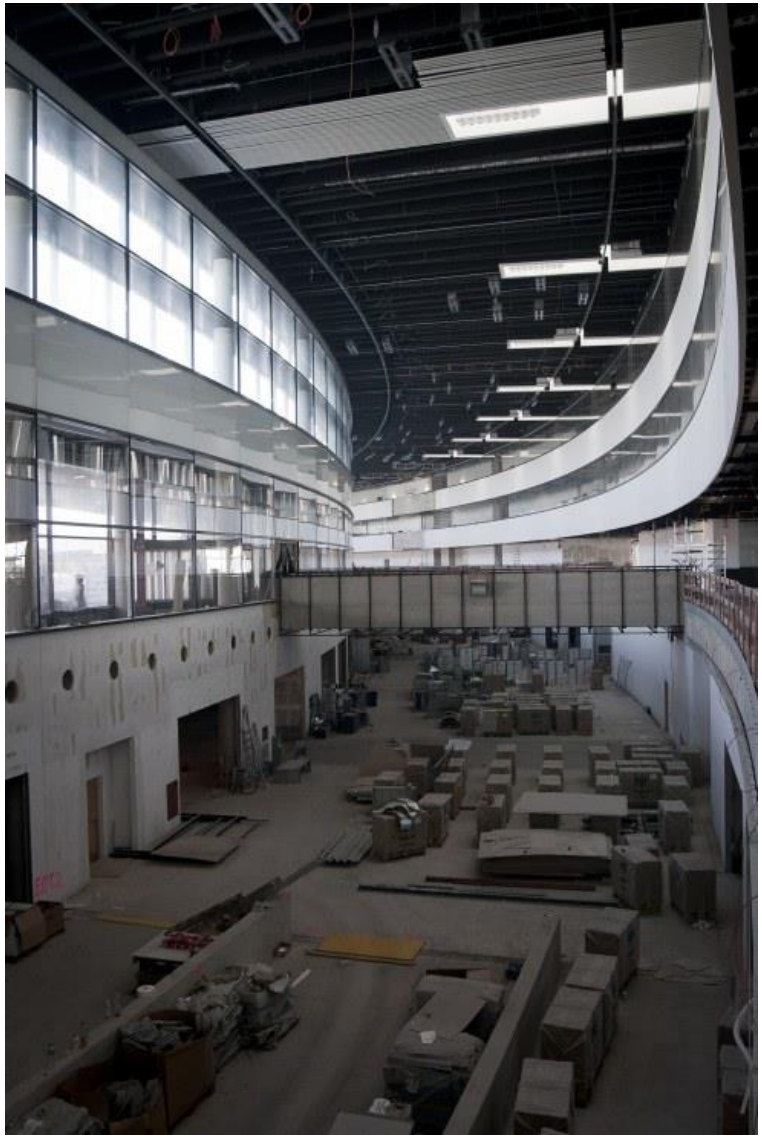
Ein Beispiel für ein Großprojekt

- Flughafen Wien – Skylink: Der Plan
  - Bauzeit: 2004 – 2008
  - Kosten: ca. € 400 Mio.



Skylink Umsetzung

- Laufende Verzögerungen
- Baumängel
- Umplanungen
- Kostensteigerungen
- Austausch der Projektleitung
- Untersuchungen des Rechnungshofes
- ....

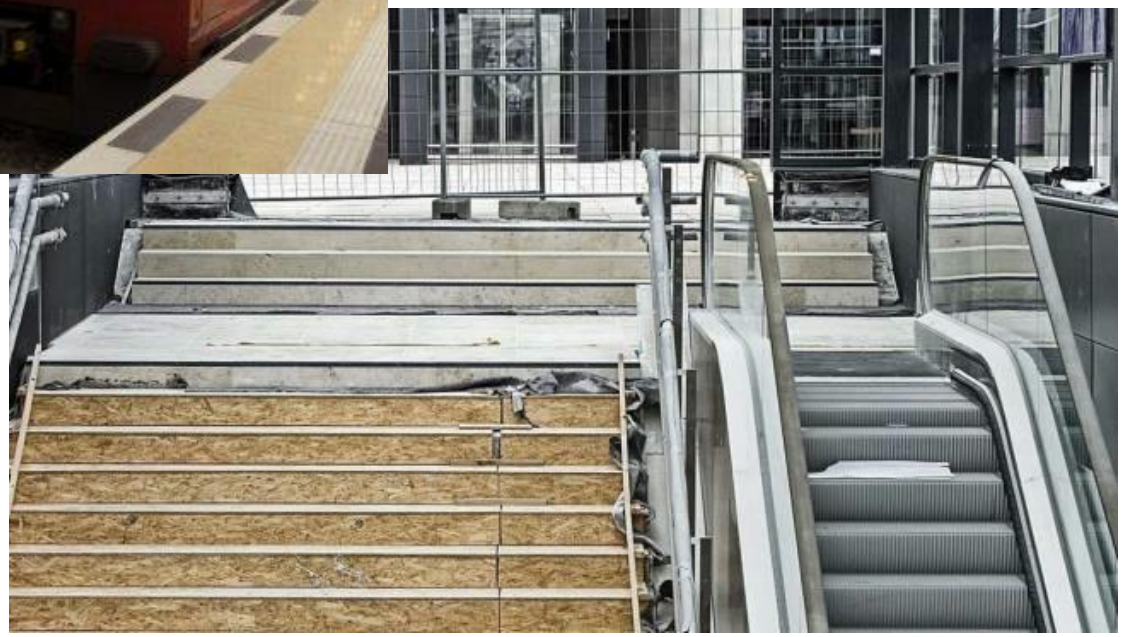


Skylink

- Das Ergebnis:
  - Fertigstellung: 2012 (+100%)
  - Kosten: über € 800 Mio. (+100%)
- Die Ursache:
  - Mängel im Projektmanagement
  - Mängel in der Bau-Ausführung und in der Bau Aufsicht
  - Lieferanten, die das (vermutlich) ausgenützt haben

Deutschland: Flughafen BER

Die Bahnanlagen der Linien zum BER müssen bis zur Eröffnung des Flughafens in einem funktionsfähigem Zustand gehalten werden, daher finden regelmäßige Fahrten zum Flughafenbahnhof statt.



Am künftigen Flughafen BER gibt es einige zu kurze Rolltreppen. Das fiel im Winter auf, nun gibt es eine "Lösung." Sie sollen aber nicht durch neue in richtiger Länge ersetzt werden.

BER

Hauptstadtflughafen wird auch 2017 nicht eröffnet

Die Eröffnung des Berliner Flughafens wird erneut verschoben. "2017 kann nicht mehr funktionieren", sagt Bürgermeister Michael Müller, es gebe Umbaubedarf. 2018 dann?

…. seufz

Google

flughafen berlin eröffnung

Alle Maps News Bilder Videos Mehr Einstellungen Tools

Ungefähr 514 000 Ergebnisse (0,55 Sekunden)

Flughafen Berlin Brandenburg / Eröffnung

2020

Feedback

Flughafen Berlin Brandenburg – Wikipedia
https://de.wikipedia.org/wiki/Flughafen_Berlin_Brandenburg
Der **Flughafen Berlin** Brandenburg „Willy Brandt" (IATA: BER, ICAO: EDDB; englisch **Berlin** Brandenburg Airport) ist ein im Bau befindlicher, an der südlichen Stadtgrenze Berlins im brandenburgischen Schönefeld gelegener, internationaler **Verkehrsflughafen**. Er trägt den Beinamen Willy Brandt und soll durch die …
Flughafen Berlin-Schönefeld · Flughafen Berlin-Tegel · Bahnhof Flughafen Berlin …

Flughafen Berlin Brandenburg: BER soll im Oktober 2020 in Betrieb …
www.zeit.de › Wirtschaft
15.12.2017 - Neun Jahre später, Milliarden teurer als einst geplant: Aufsichtsrat und Geschäftsführung des **Berliner** Flughafens haben sich auf einen neuen Eröffnungstermin geeinigt.

Flughafen BER eröffnet womöglich erst 2021 - Flughafen Berlin …
https://www.morgenpost.de › Flughafen BER
23.11.2017 - Der TÜV hat für den BER gravierende Mängel aufgelistet. Es könnte sein, dass erst 2021 die ersten Maschinen vom **Flughafen** abheben.

Flughafen Berlin Brandenburg

Flughafen in Brandenburg

Der Flughafen Berlin Brandenburg „Willy Brandt" ist ein im Bau befindlicher, an der südlichen Stadtgrenze Berlins im brandenburgischen Schönefeld gelegener, internationaler Verkehrsflughafen. Wikipedia

Flughafencode: BER
Fläche: 1470 ha
Eröffnung: 2020
Kapazität (PAX pro Jahr): 27 Mio. (erste Ausbaustufe)
Terminals: 1 (Sektionen: A, B, C, D)
ICAO-Code: EDDB

Andere suchten auch nach Über 15 weitere ansehen

Stuttgart 21

Flughafen Berlin-Sc…

Flughafen Berlin-Tegel

Berlin-Tegel

Flughafen München

Feedback

VICE DIGITAL

Der Postillon
Ehrliche Nachrichten - unabhängig, schnell, seit 1845

Mittwoch, 28. Dezember 2016

Neue Sprachregelung: BER eröffnet "noch in diesem Jahrtausend"

Berlin (dpo) - Um weitere Nachfragen und peinliche Verschiebungen des Eröffnungstermins zu vermeiden, haben sich die Länder Berlin und Brandenburg auf eine neue Kommunikationsstrategie geeinigt. Künftig wird es von offizieller Seite nur noch heißen, der neue Hauptstadtflughafen BER werde noch in diesem Jahrtausend eröffnen.

AUS DEM ARCHIV

Smartphone-Nutzer legt sich Blindenhund zu, um nicht mehr auf Verkehr achten zu müssen

REKLAME

Der Postillon
Seite gefällt mir 2,4 Mio. „Gefällt n

REKLAME

Flughafen München – Umzug in 24h

- Am 16. auf den 17. Mai 1992 zog der komplette Flugbetrieb relativ reibungslos über Nacht vom Flughafen München-Riem zum neuen Flughafen um, nachdem in Riem am Vortag kurz vor Mitternacht mit dem Start der Lufthansa-Boeing B737 *Freising* der letzte Linienflug durchgeführt wurde.
- Die Logistik des Umzugs erlangte weltweite Beachtung, was dazu führte, dass häufig Experten des Flughafens München für ähnliche Projekte angefordert wurden, so beispielsweise beim Umzug des Flughafens Bangkok oder des Flughafens Athen- Eleftherios Venizelos.

Risiken sind ein Akzeptanzproblem!

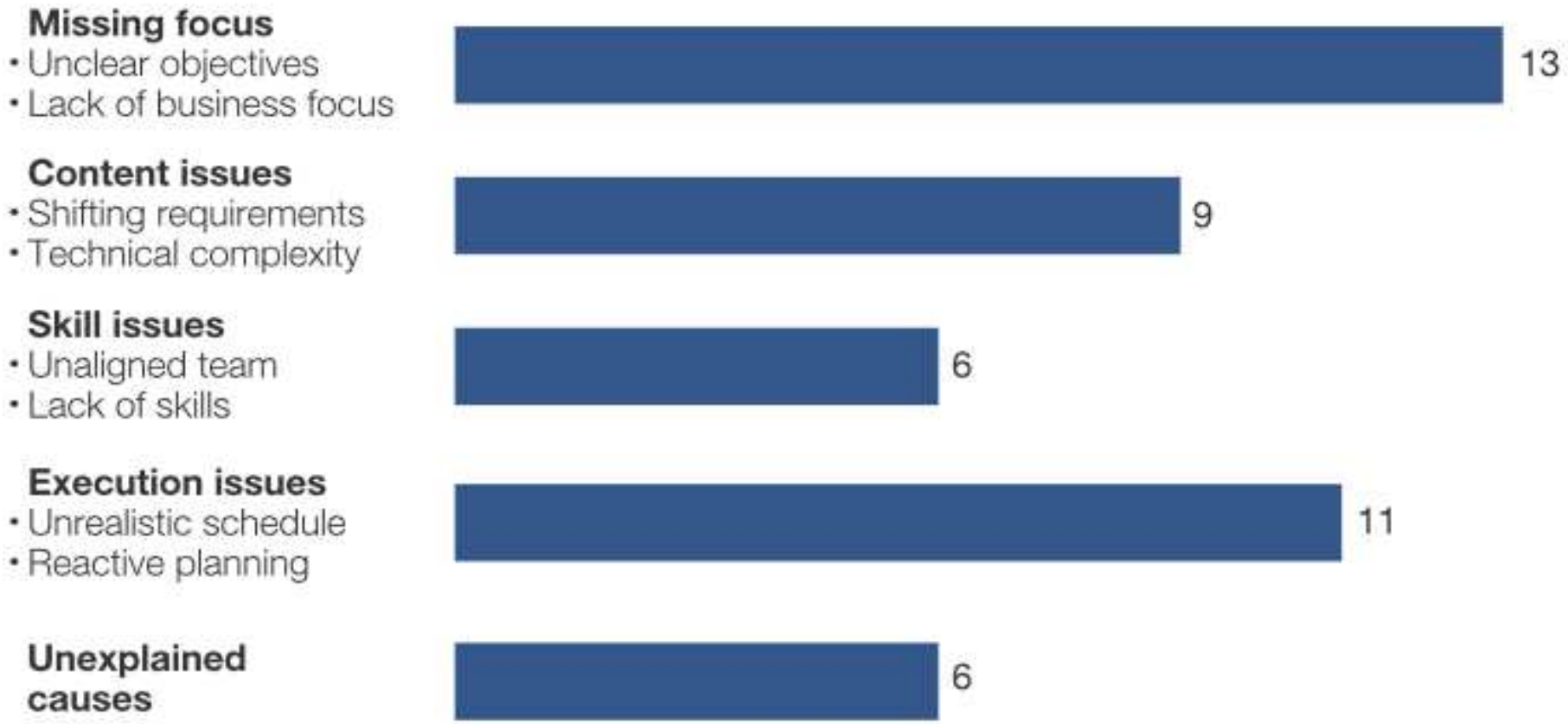
- Warum tun wir uns mit Risikomanagement so schwer?
  - Weil wir Ziel-/ ergebnisorientiert denken!
  - Weil ein Risiko die Interessen eines Stakeholders gefährdet!
  - Weil wir uns mit Wahrscheinlichkeiten schwer tun!
  - Weil wir am Ende des Problems nicht ein Bisschen haben, sondern ganz oder gar nicht!
- Risiken bezüglich der Produktqualität
  - Lösung ist nur eingeschränkt brauchbar / wartbar
- Risiken bezüglich der Projektabwicklung
  - Der Terminrahmen / Kostenrahmen wird überschritten

- „Wenn ein Projekt kein Risiko in sich bringt, dann lass die Finger davon“
  Tom DeMarco

Ein paar Ursachen

- Mangelhafte Anforderungen:
  - Die Anforderungen decken die Erfordernisse nicht ab
  - Die Anforderungen sind unklar oder mehrdeutig
- Mangelhafte Umsetzung der Anforderungen
  - Mangelndes Verständnis der Anforderungen
  - Mängel in der technischen Umsetzung (Implementierung)
  - Termin-Not und Ressourcenmangel
- Mängel im Projektmanagement
  - Mängel in der Kommunikation
  - Unrealistischer Termin- und Kostenrahmen
  - Ressourcenmangel: Verfügbarkeit, Qualifikation, Erfahrung

Rough distribution by cause of the 45% of IT projects that experience cost overruns (for those with budgets >$15 million in 2010 dollars), %



IT projects with budgets >$15 million

Cost overrun, 45% Schedule overrun, 7% Benefits shortfall, –56%

Source: McKinsey–Oxford study on reference-class forecasting for IT projects

Risikoanalyse und Risikomanagement

- Analyse der Risiken
  - Spezifische Risiken aus der Aufgabenstellung
  - Generelle Risiken des Projekts (Personalausfall, ...)
  - Eintrittswahrscheinlichkeit?
  - Schadenspotential?

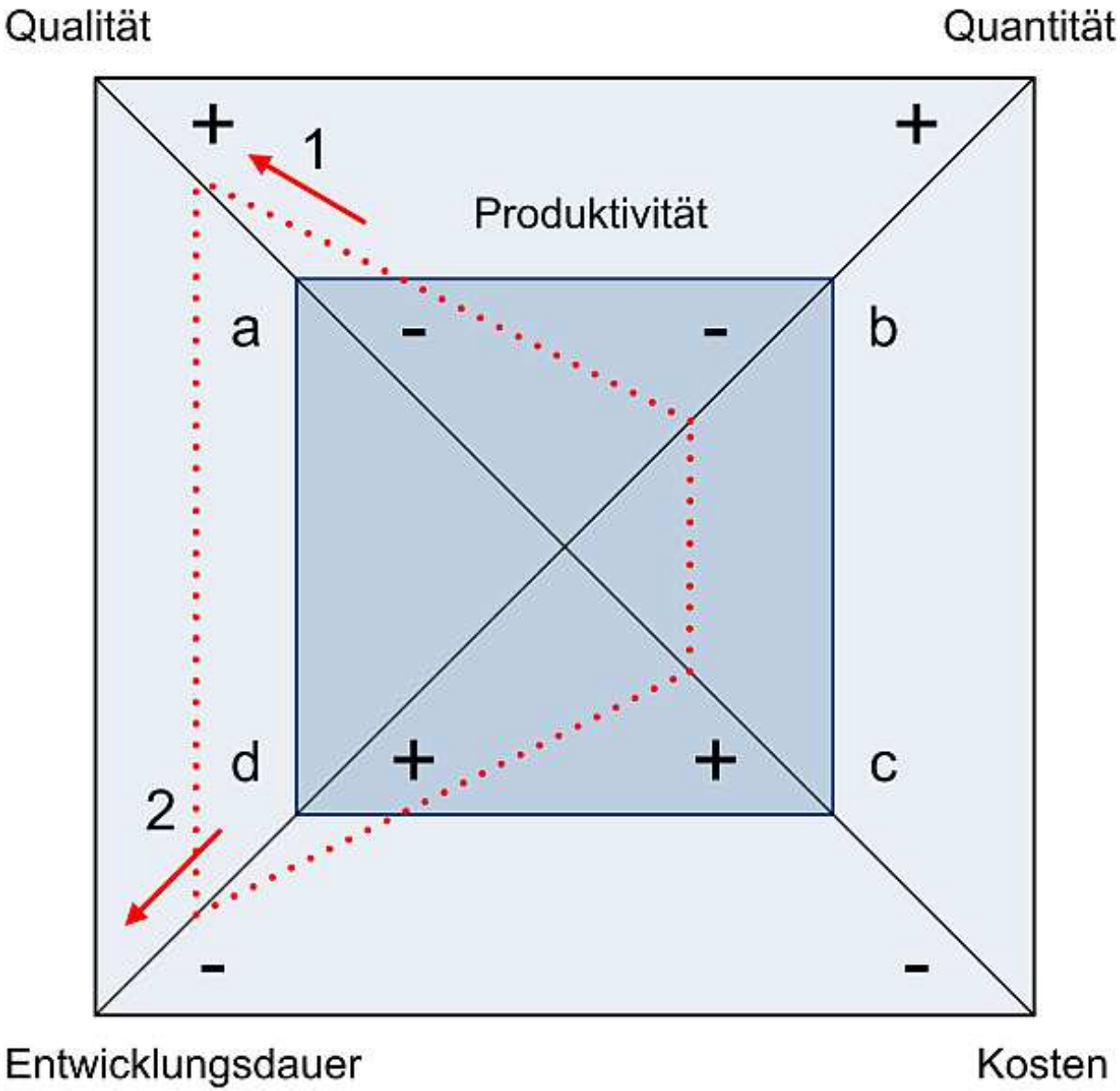
- Planung von Maßnahmen
  - Vorbeugemaßnahmen (reduzieren das Risiko)
  - Abhilfemaßnahmen (lindern die Folgen)
  - Kalkulation möglicher Mehrkosten

Wechselwirkungen
Termine / Personaleinsatz

- Die Kompensierbarkeit von Zeit und Personal ist nur fiktiv.
  - Beschränkte Teilbarkeit der Aufgaben
  - Vorgegebene Arbeits-Sequenzen
  - Beschränkte Kapazität wichtiger Mitarbeiter
- „Adding manpower to a late project makes it later!“
  F.H. Brooks: The Mythical Man-Month, 1975

1. Ziel: Verkürzung der Projektlaufzeit
2. Mehr Mitarbeiter
3. Höherer Kommunikations- Koordinierungs- und Dokumentationsaufwand
4. Geringere Produktivität
5. Resultat: Höherer Gesamtaufwand

Das „Teufelsquadrat“



Projektmanagement

- Management = Getting things done by people
- Projektmanagement = Getting the project done

Aufgaben im Projektmanagement

- Die Aufgabenstellung definieren
- Das Projekt planen
- Das Projekt(team) organisieren
- Aufgaben verteilen
- Den Projektfortschritt kontrollieren
  - Ergebnisse ⇔ Termine, Aufwand, Kosten
- Bei Bedarf steuernd eingreifen
- Sich mit Risiken beschäftigen
- Entscheidungen vorbereiten & Entscheidungen fällen
- Den nötigen Verwaltungskram erledigen, ...

- **Die Aufgaben sind vielfältig ⇔ Die Anforderungen sind vielfältig**

Projektplanung

Planung heißt den
Zufall durch Irrtum
zu ersetzen!

Grundlagen der Projektplanung

- Die Vorgaben des Kunden:
  - Aufgabenstellung
  - Termin- und Kostenrahmen
- Das Realisierungskonzept
- Die Möglichkeiten des Auftragnehmers / des Projektteams:
  - Verfügbares Ressourcen
  - Know-How und Erfahrung
  - Budget und Termindruck

Grundlagen der Projektplanung (SEPM PR)

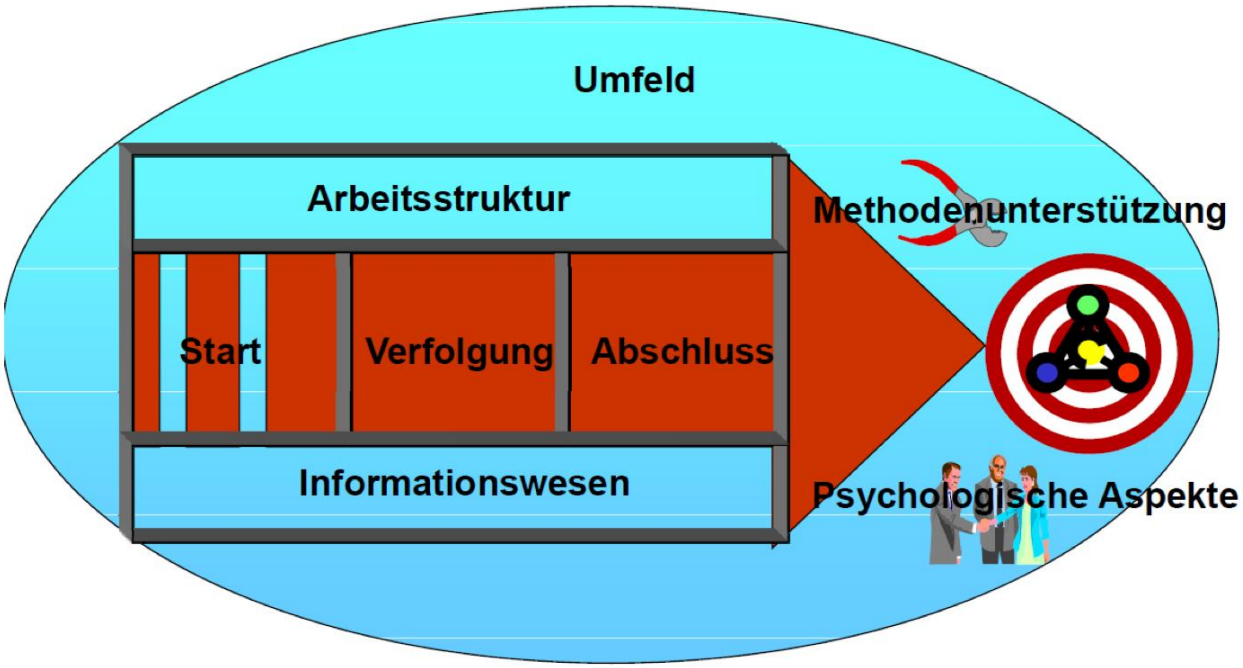
- Die Vorgaben des ~~Kunden~~ LVA-Leitung (+Kunde):
  - Aufgabenstellung: Projektauftrag
  - Termin- und Kostenrahmen: Bis zum Ende des Semesters, Kostenrahmen: 6 ECTS

- Das Realisierungskonzept

- Die Möglichkeiten des Auftragnehmers / des Projektteams:
  - Verfügbares Ressourcen: 5-6 Studierende
  - Know-How und Erfahrung: Einzelphase, vorherige LVAs, privat/beruflich
  - Budget und Termindruck: Deadlines, andere LVAs/Prüfungen, fixes „Budget“ (6 ECTS) etc.

Element des Projektmanagements

- Projektmanagement muss sich um folgende Elemente kümmern:
- Projektablauf – was ist zu tun? (Start, Verfolgung, Abschluss)
- Arbeitsstruktur – wer ist wofür zuständig?
- Informationswesen – wer informiert wen, worüber, wann, wie?
- Dabei müssen die notwendige Methodenunterstützung, psychologische Aspekte sowie Umfeldbedingungen beachtet werden.

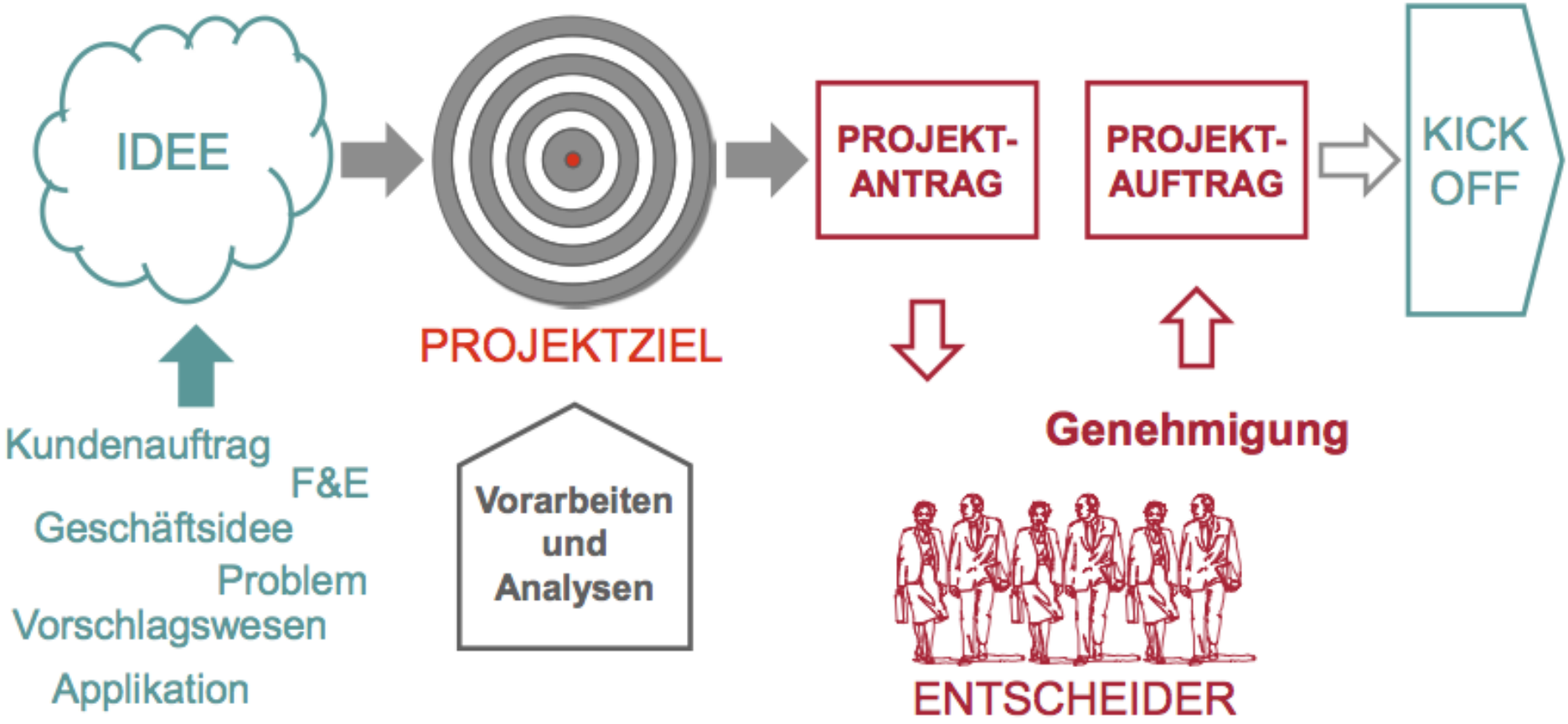


Schrittweises Vorgehen

- Projektinitiierung (Studie) → Grobe/Vorläufige Planung
- Angebot / Auftragserteilung (Pflichtenheft/Spezifikation) → Gültiger Projektplan
- Je Meilenstein / Phase → Detailplanung der nächsten Schritte
- Bei Abweichungen oder Änderungen → Aktualisierung der Planung

Plane mit den besten/aktuellsten Informationen die du hast!

Von der Idee zum Kick-Off



Projektstart

- Der Projektstart umfasst den Abschnitt von der Projektidee bis zum Kick off

- Projektinitiative: Was ist zu tun?
  - Bekanntgabe und Diskussion der Idee
  - Designierung Projektleitung
  - Suche nach kompetenten Gesprächspartnern
  - Organisation der ersten Planungsgespräche

Vorarbeiten

- Was ist zu tun?
  - Evaluierung der Projektidee
  - Eventuelle Vorstudien
  - Ersteinschätzung der Ressourcenanforderungen
- Projektumfeldanalyse
  - Projekttyp
  - Funktionalitäten
  - Qualität
  - Projektunterstützende Faktoren
  - Mögliche Risiken
- Vorarbeiten sind notwendig:
  - Um zu entscheiden, ob ein Projekt überhaupt in Angriff genommen werden soll
  - Zur Konkretisierung einzelner Punkte des Projektauftrags
  - Das Ziel ist die Risikoreduktion
  - Forschung und Entwicklung
  - Vorarbeiten können eigenständige Projekte mit dem Ziel der Erarbeitung von Entscheidungsunterlagen werden.

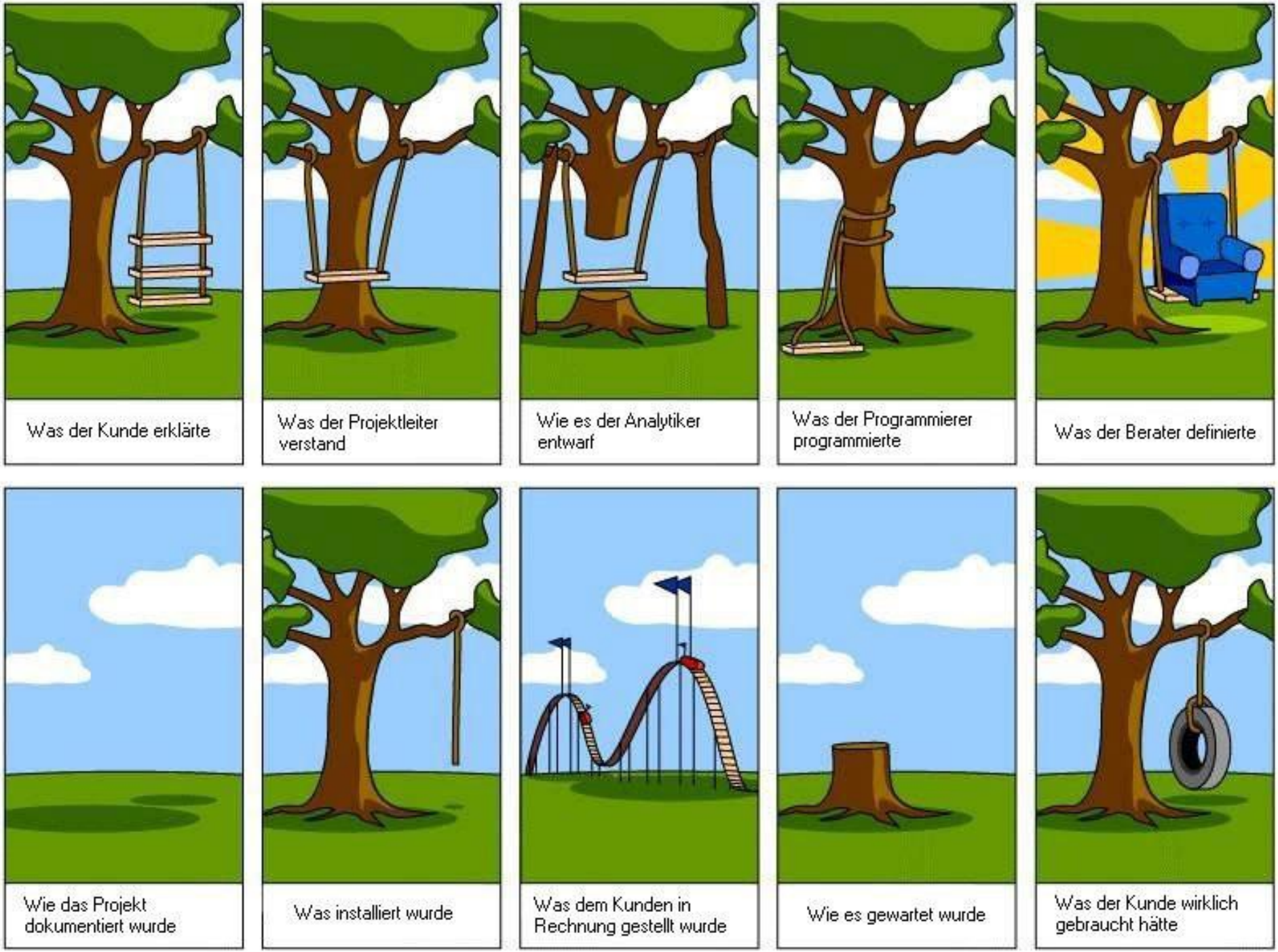
Mangelndes Verständnis der Anforderungen

- Je innovativer eine IT-Lösung sein soll, desto weniger wissen wir zu Beginn über die erforderlichen Eigenschaften des Systems.

- „I believe that you **think** you understand what I said, but I fear, that you don‘t realize, that **what I say** is not always **what I really mean**.“

Termindruck und seine Auswirkungen

- „We try to solve the problem by **rushing** through the analysis- and design-phase so that **enough time is left** at the end of the project to uncover all the errors, that we made, **because we rushed** through the analysis- and design-phase“ G. J. Myers


Was der Kunde erklärte
Was der Projektleiter verstand
Wie es der Analytiker entwarf
Was der Programmierer programmierte
Was der Berater definierte
Wie das Projekt dokumentiert wurde
Was installiert wurde
Was dem Kunden in Rechnung gestellt wurde
Wie es gewartet wurde
Was der Kunde wirklich gebraucht hätte

Mit der Veränderung leben

- Projekte sind selten stabil:
  - Sich verändernde Rahmenbedingungen sind **normal**!
  - Man muss mit Veränderungen rechnen!

Unter welchen Voraussetzungen können Projekte trotzdem erfolgsversprechend geführt werden?

- Wir brauchen klare Prioritäten, um zu wissen, was wichtig ist, und was nicht!
- Die Planung braucht Spielräume für Veränderungen!
- Die Planungszyklen werden kürzer, um auf Änderungen geordnet und trotzdem flexibel reagieren zu können!

„Ein Projekt das stabil läuft ist tot!“ Tom DeMarco

Projektvorschlag

- Projektbezeichnung und Entwicklungsteam
- Ausgangssituation
- Projektbeschreibung
- Typ des Projekts (z.B.: Neuentwicklung, Migration, Wartung, etc.)
- Zielgruppe
- Ziele und Nicht-Ziele
- Anforderungen und Features
- Aufwand und Priorität
- Domänenmodell

- Projektentscheidung erfolgt in der Kick-off Besprechung
  - Analyse der Umsatzbarkeit, Nutzen und Vollständigkeit

Rollen im Projekt

- Auftraggeber und Auftragnehmer
- Projektleiter
- Entwicklerteam
  - Software Architekt
  - Dokumentenbeauftragter
  - Testbeauftragter
  - Weitere Rollen..

Projektbeauftragung

- Der Projektauftrag ist eine schriftliche (Ziel)vereinbarung zwischen Auftraggeber und Auftragnehmer, ein Projekt zu bestimmten Bedingungen durchzuführen. Er wird auch Projektantrag, Projekterklärung, Projektdefinition und Projektbeschreibung genannt.

- Der Projektauftrag ist eine Management-Unterlage, d.h. er umfasst alle wichtigen Daten für die zukünftige Projektabwicklung

- Voraussetzung dafür ist die Ausarbeitung eines Projektvorschlags durch das PM (unter Mitwirkung der zukünftigen Team-Mitarbeiter)

Projektauftrag

- Projektname
- Projektbeschreibung
- Rollen und Verantwortlichkeiten
- Nicht-Ziele, Abgrenzung
- Komponentendiagramm
- Wiederverwendung von Komponenten, Technologiewahl
- Lieferumfang und Abnahme
- Nichtfunktionale Anforderungen
- Arbeitsstruktur
- Projektplan (Zeit- und Kostenplan)
- Informationswesen
- Besonderheiten
- Umfeld- und Risikoanalyse

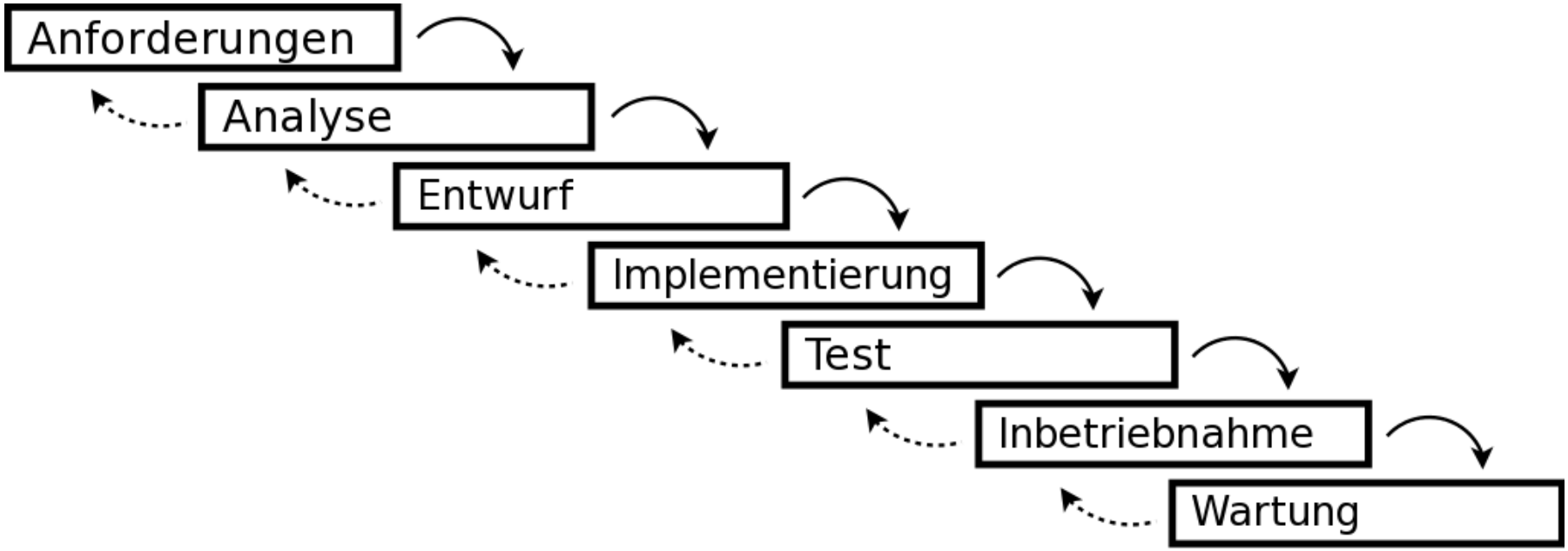
Vorgehensweise

1. Projektstrukturplan erarbeiten
2. Arbeitsplanung durch Arbeitspaketdefinition
3. Aufwandsschätzung
4. Ablauf und Terminplanung
5. Ressourcenplanung
6. Kostenplanung
7. Optimierung des Gesamtprojektes

Projektumsetzung

- Klassisch / Sequenziell
  - Wasserfall
  - V-Modell
- Iterativ:
  - RUP (Rational Unified Process)
- Agil:
  - SCRUM
  - eXtreme Programming

Sequenzielle Softwareentwicklung
Wasserfallmodell



Das sequentielle Dilemma

- Je innovativer eine IT-Lösung sein soll, desto weniger wissen wir zu Beginn über die erforderlichen Eigenschaften des Systems!
- Der sequentielle Entwicklungsprozess basiert aber auf stabilen Anforderungen!
- Je dynamischer sich ein Geschäftszweig entwickelt, desto rascher ändern sich die Anforderungen an die zugehörige IT-Unterstützung
- Der sequentielle Entwicklungsprozess hat damit eine System-immanente Bruchstelle!

Die Konsequenzen der sequentiellen Entwicklung

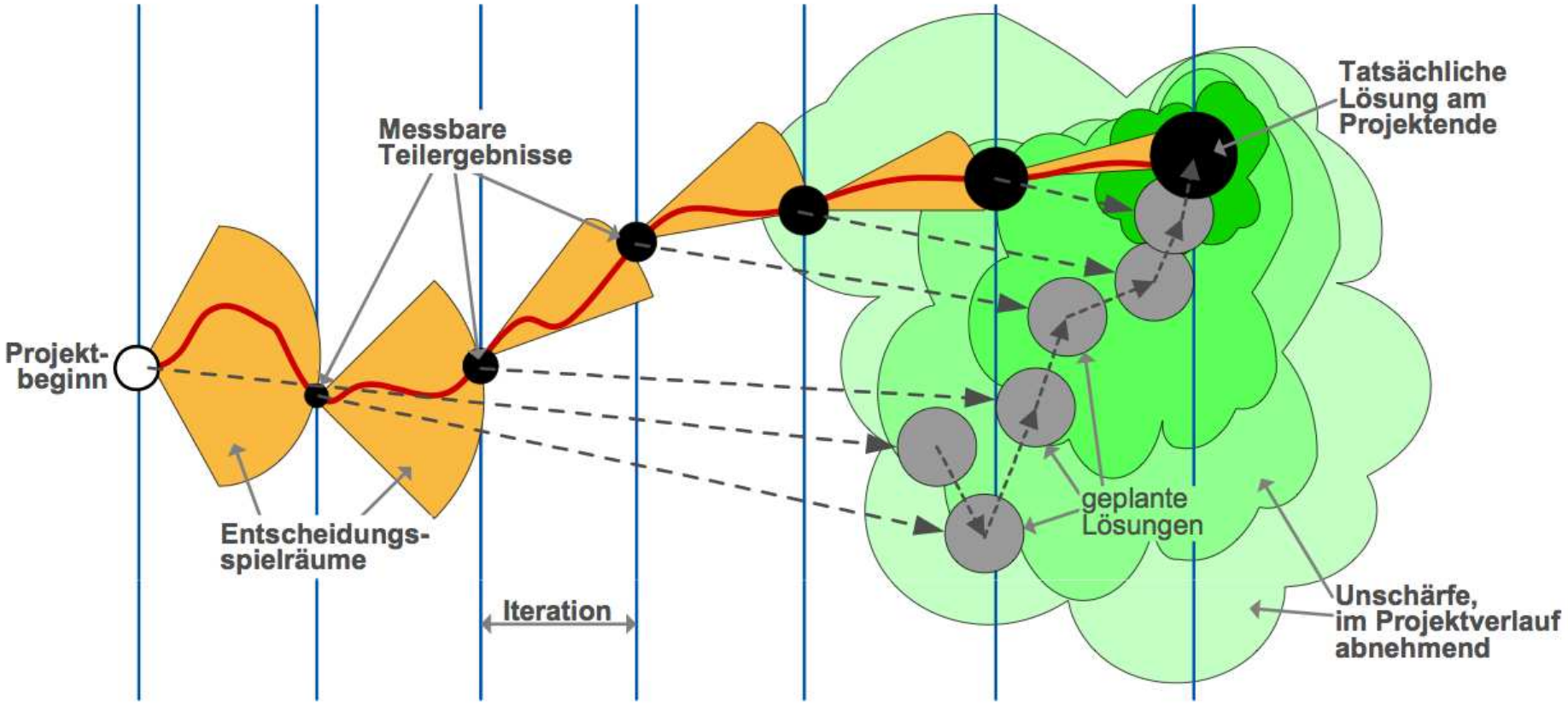
Selbst wenn wir die **Software richtig** entwickelt haben, haben wir oft **nicht die richtige Software** entwickelt:

- Das Verständnis unserer Kunden für die Anforderungen hat sich geändert!
- Das Geschäft hat sich weiter entwickelt!
- Damit haben sich auch die Anforderungen an unsere Software geändert!

Der sequentielle Prozess ist gar nicht so robust ...

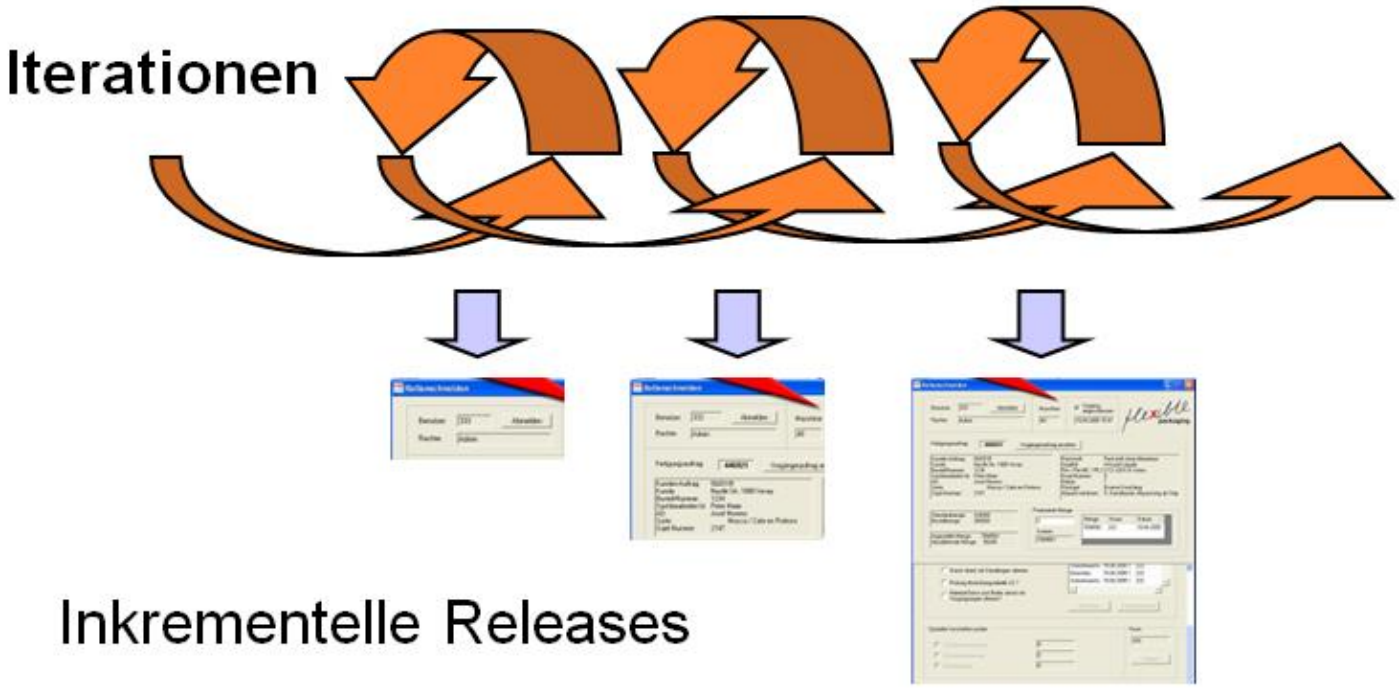
Die Konsequenzen der sequentiellen Entwicklung

Wir müssen auf Änderungen flexibler reagieren!



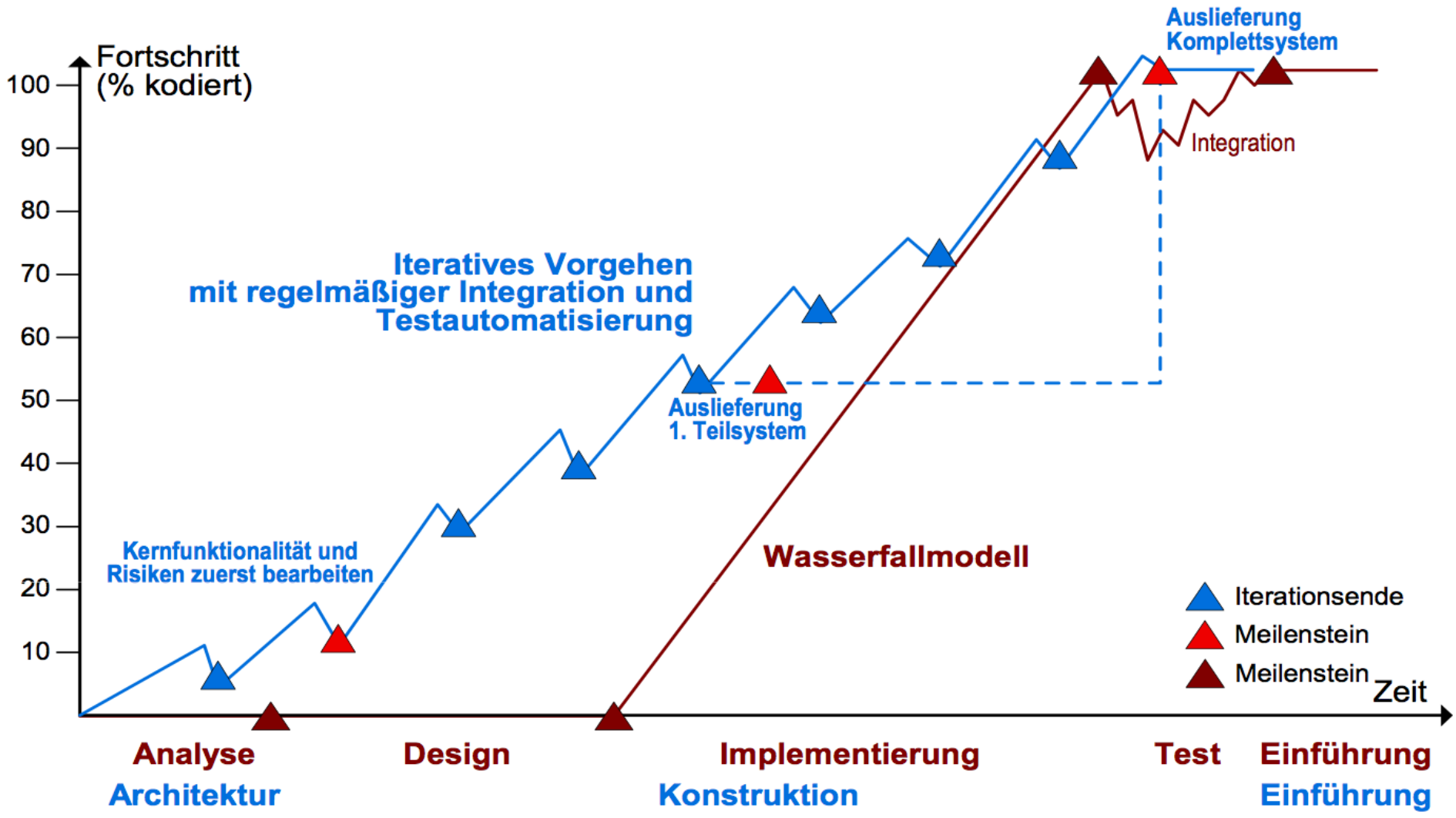
Iterative Softwareentwicklung

- Kurze Iterationen liefern lauffähige Software mit steigendem Funktionsumfang



- Grundprinzip: Fixe Iterationsdauern
- Das verändert das Projektmanagement: Timebox management

Kontinuierliche Software Entwicklung



Techniken der Projektsteuerung

- Sind wir noch im Plan?
- Wenn nein – warum nicht?
- Und – wo geht die Reise hin?

Steuerung durch Soll-Ist-Vergleich

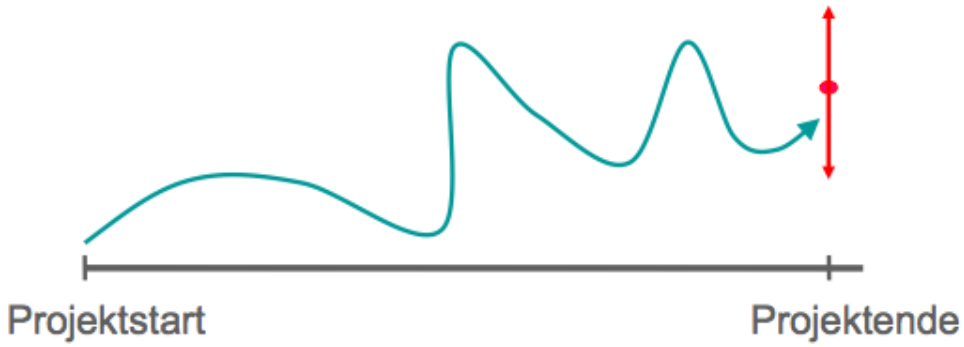
- Die Voraussetzung für den Soll-Ist-Vergleich:
  - Es gibt Vorgaben, Ziele, Erwartungen (Plan- oder Sollwerte)
  - Der Ist-Status ist feststellbar
  - Die Daten sind vergleichbar

Die Konsequenz:

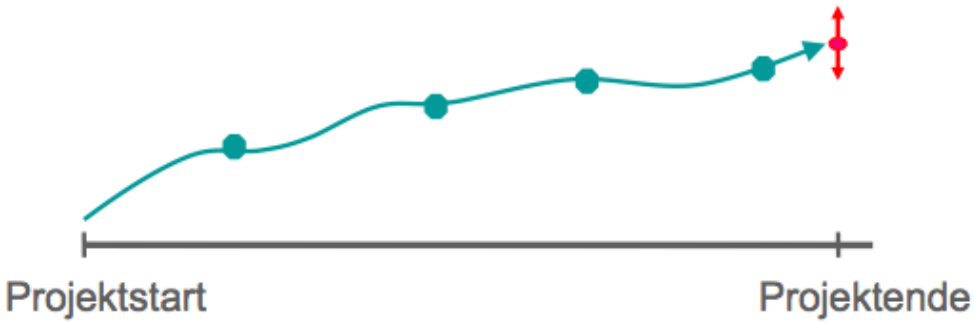
- Bei der Planung muss klar sein, was man vergleichen will!

Meilensteine

- Projektverlauf ohne Meilensteine

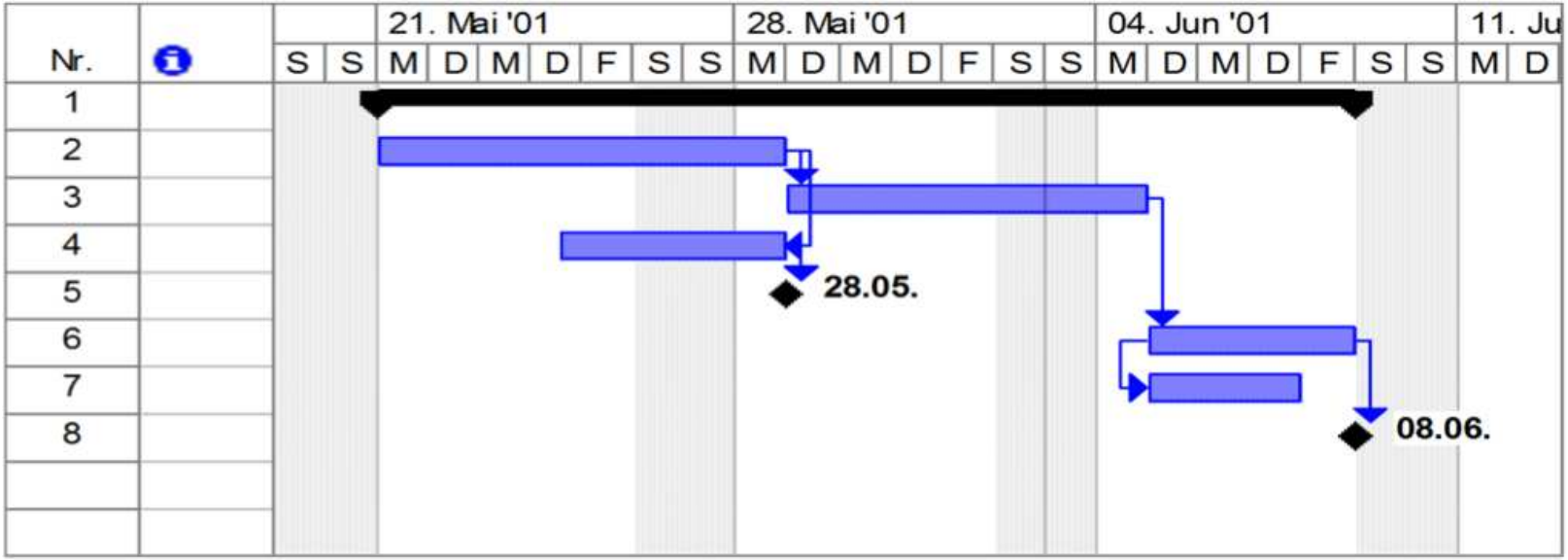


- Projektverlauf mit Meilensteine

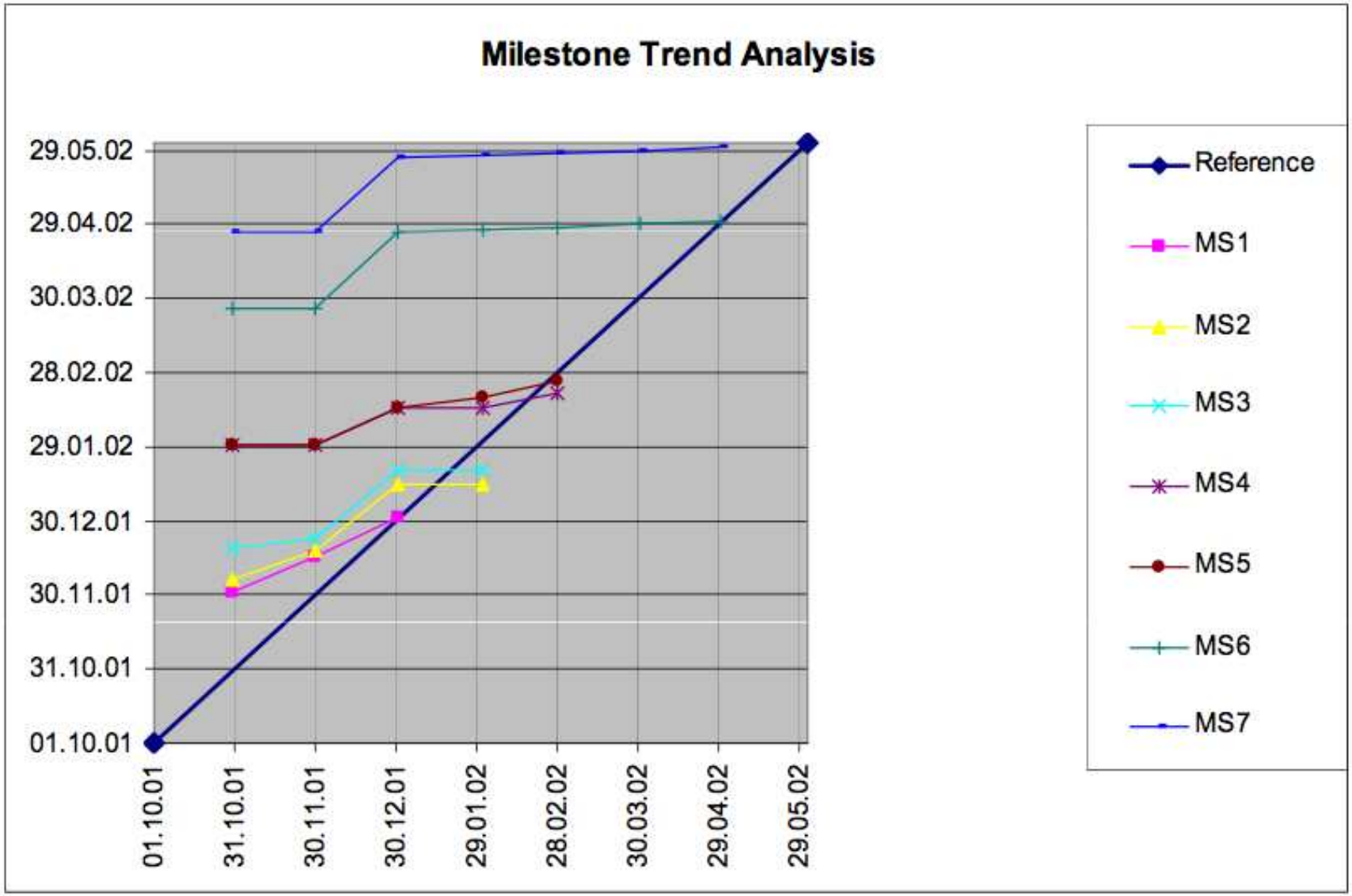


Balkendiagramme (Gantt-Diagramm)

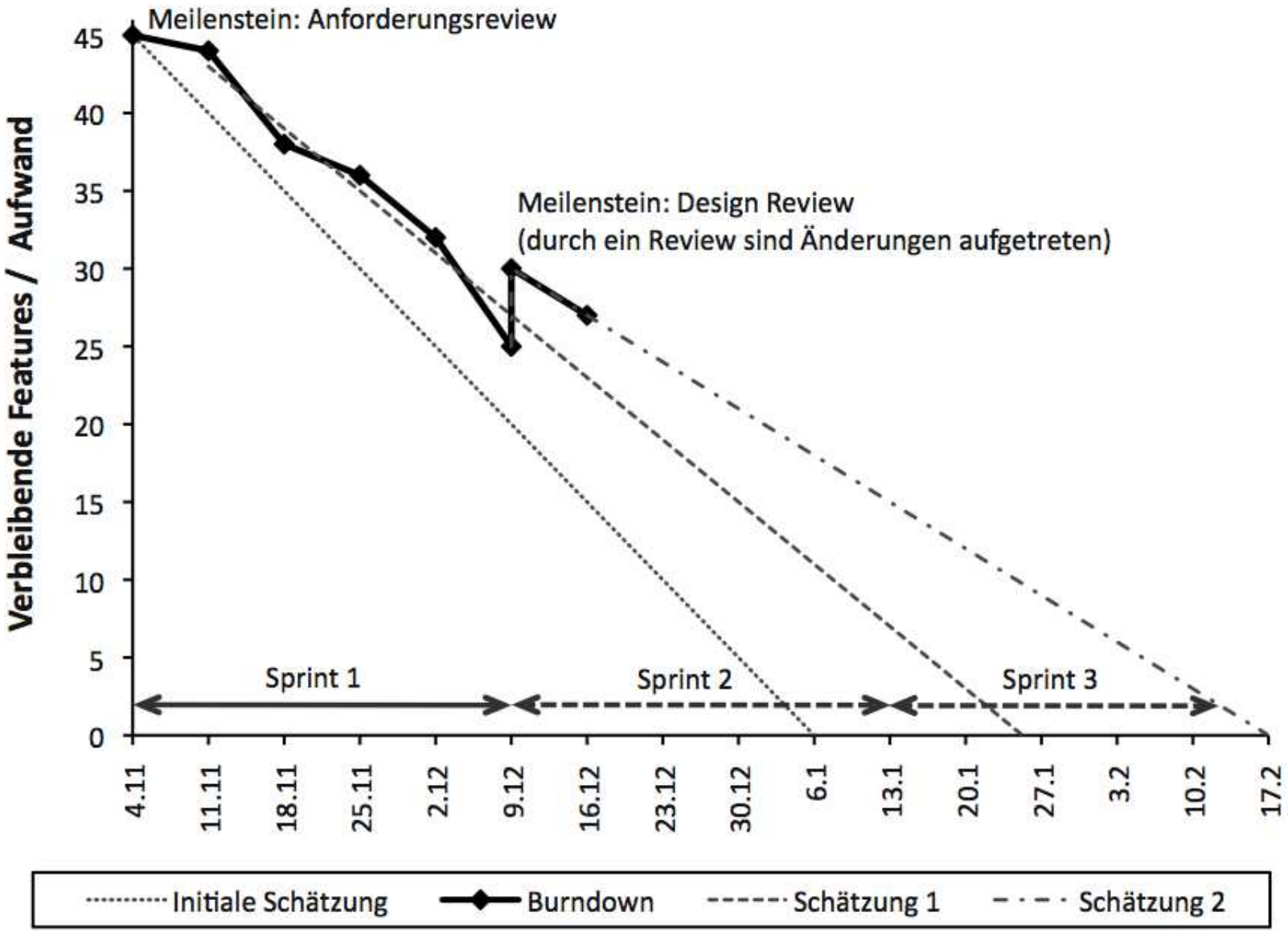
- Älteste und am weitesten verbreitete grafische Methode
- Aufgaben werden zur Terminplanung dargestellt
- Aufgaben oder Personenbezogen



Meilensteintrendanalyse (MTA)

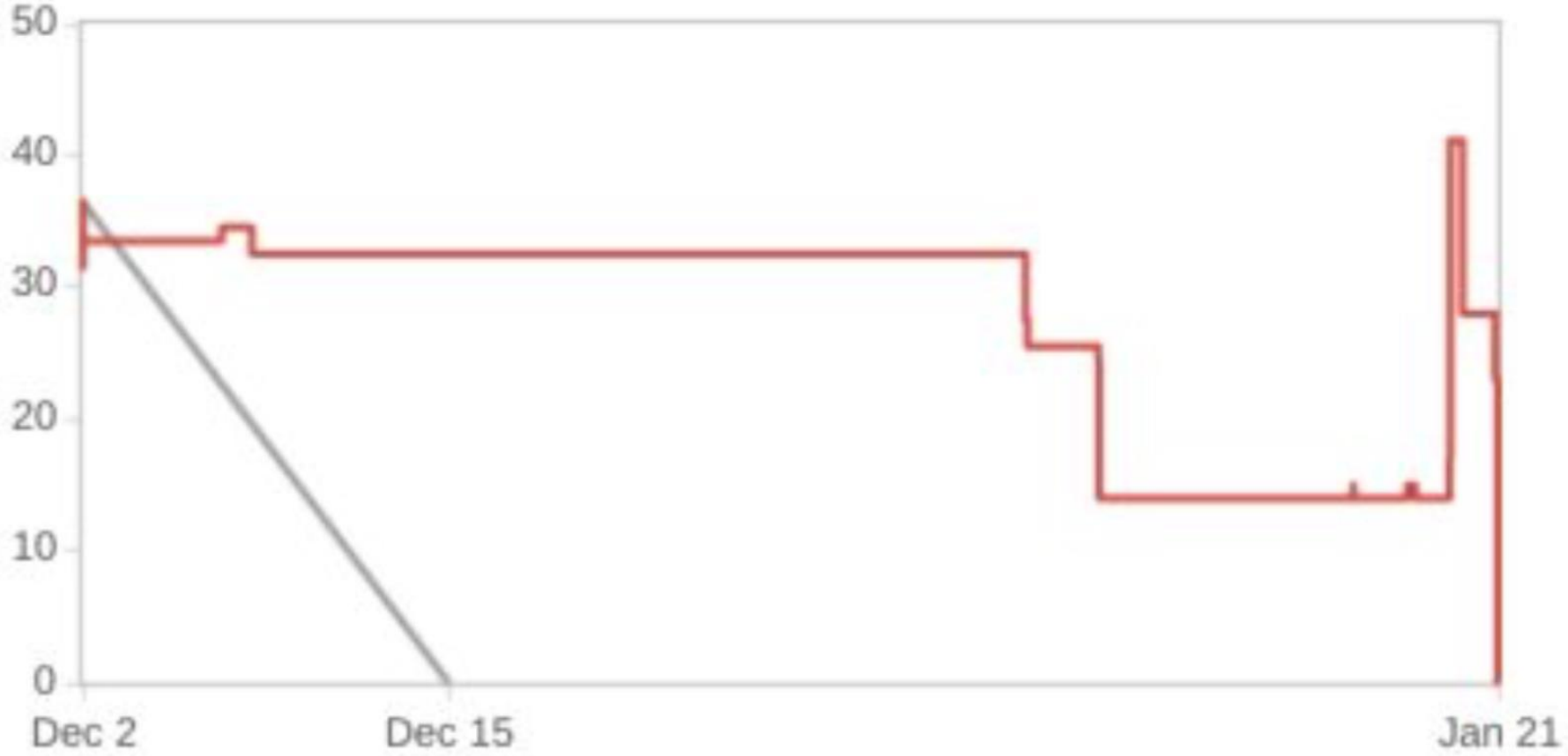


Burn-down-Chart



Burn-down-Chart Analyse

- Burn-down Chart einer Advanced Software Engineering Gruppe



WANTED
FOR DELAYING IMPORTANT PROJECT
DID NOT:
• ANSWER EMAIL
• RETURN CALLS
• MAKE UP MIND
• ATTEND MEETINGS
GOFF
© 1997 Ted Goff www.tedgoff.com

Software Engineering & Projektmanagement (SEPM) Vorlesung Block „Einführung in SE“

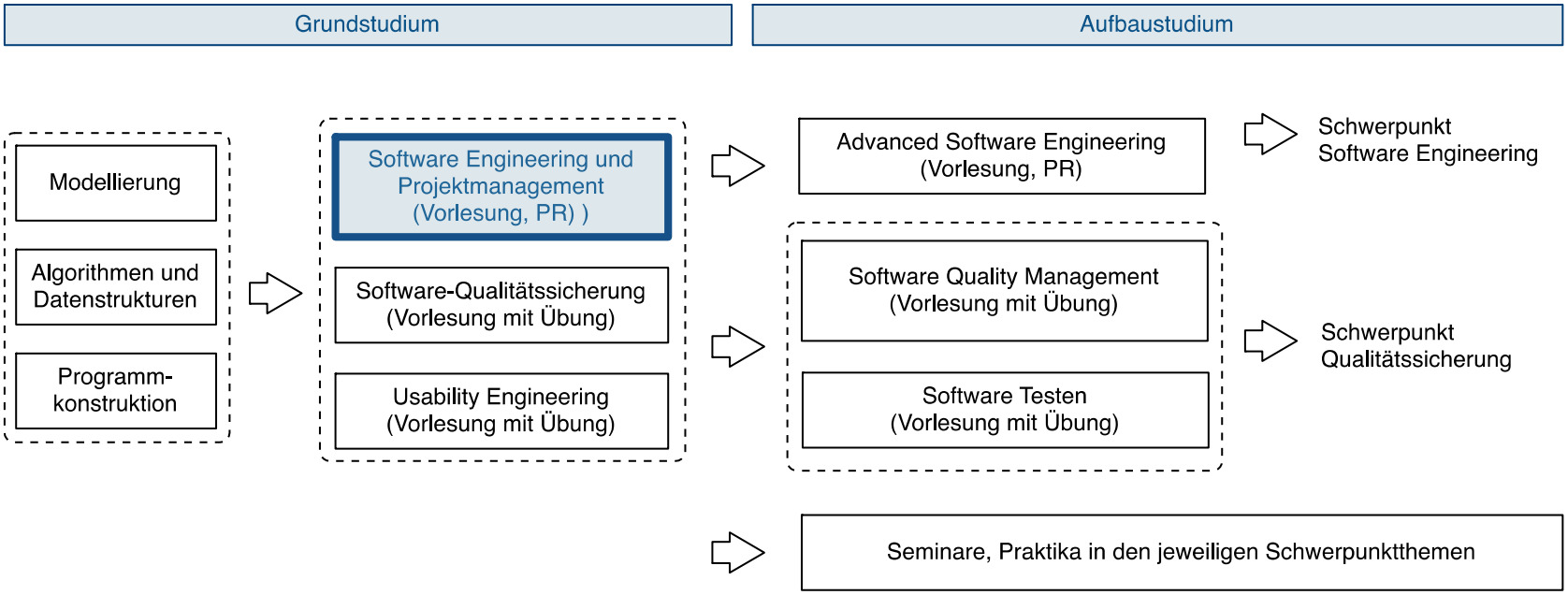
Stefan Biffl

Grundlagen SE-Projekte

Projekttypen

Software Prozesse und Produkte

Software Engineering im Studium



Featured Project ***Presentr***
Web-Plattform für effiziente interaktive Präsentationen

- *Presentr* ist ein aus einem erfolgreichen Projekt aus *Advanced Software Engineering* entstanden.

- Beiträge von *Presentr* (http://presentr.at)
  - Vortragende können via *Presentr* im Browser präsentieren.
  - Studierende können an Audience Response Tasks teilnehmen.
  - Studierende können die Präsentation mit den Ergebnissen des Audience Response Tasks als PDF herunterladen.

- Dadurch können Vorträge effizienter interaktiv gestaltet werden.

- *Presentr* wird anhand des Feedbacks kontinuierlich weiter entwickelt.
- *Presentr* wird in diesem Semester in der SEPM Vorlesung durchgehend verwendet.
- Ihre Rückmeldungen sollen in die Gestaltung von *Presentr* einfließen.

Ziele der Vorlesung

- **Ergänzung zur Laborübung**
  - Ein Team, ein mittel großes Produkt, ein Prozess.

- **Erlernen wesentlicher Konzepte der Softwareentwicklung**
  - Technische Grundlagen.
  - Ergänzungen/Erweiterungen zu Inhalten aus dem Projekt.
  - Aktuelle Methoden in der Software-Entwicklung.
  - Software Life Cycle.
  - Vorgehensmodelle.
  - Projektmanagement.

Inhalte 1/2

1. Einführung in Software Engineering
   - Projektypen: Eingebettete Systeme, kommerzielle Software
   - Umfeld von Software Engineering (Personen, Projekt, Markt, Technologie)
2. Projektmanagement Teil 1
   - Projektauftrag, Umfeldanalyse
   - Strukturpläne und Planungsablauf
3. Techniken und Werkzeuge
   - Technische Grundlagen (Configuration Management, Source Code Management, Build Management)
   - Standardtechnologien für die Projektorganisation
4. Modellierung von Anwendungsszenarien
   - Grundlagen (Modell & Diagramme, Bedarf an Modellierung)
   - Daten und Kontrollflussmodelle in UML
5. SE Phasen
   - Anforderungen, Entwurf & Design, Implementierung, QS & Testen, Wartung

Inhalte 2/2

6. SE Prozesse im Überblick
   - Software Life-Cycle, Vorgehendmodelle (Strukturierte & Agile Ansätze)
7. Software Design
   - Architekturevaluierung, Tests
   - Design Patterns
8. Projektmanagement Teil 2
   - Produktionsfaktor Mensch
   - Auswahl von Mitarbeitern, Team Management
9. Persistenz-Stragien
   - Objektorientierte und Relationale Datenbanken
   - Web-Services/REST, Cloud Computing
10. Qualitätssicherung in SE Projekten
    - Review von SE-Modellen (ER, UML)
    - Erstellen, Beurteilen und Anwenden von Testfällen
    - Test Driven Development (TDD)

Vorlesungsprüfung

Die Prüfung besteht aus

- 10 Theoriefragen (60 Punkte)
- 1 Kreativbeispiel (40 Punkte)
  Das Kreativbeispiel kann kleine Modellierungsbeispiele und andere praktische Aufgaben umfassen.
- Die Benotung orientiert sich am üblichen Schema.
- Beide Teile der Prüfung (Theorie- und Kreativteil) müssen positiv sein.

- Termin: Juni (Nebentermin April)
  (Anmeldung via TISS ab ca. 1 Monat vor der Prüfung)
- Prüfungsgrundlage sind die in der Vorlesung besprochenen Inhalte (Folien, BPSE Buch).
- Closed Book Prüfung!
- Empfohlene Literatur siehe TUWEL.
- Beispiele bisheriger Prüfungen: siehe Prüfungsordner im TUWEL.

Überblick Block 1: Einführung, SE-Projekte

Block 1-1 “Einführung” (30’)
- BPSE Buch, Kapitel 1
- Grundlagen, Begriffe
- Begleitendes Beispiel zur Vorlesung: e-Katalog

Block 1-2 “SE-Projekte” (45’)
- Projekttypen, Umfeld des SE-Prozesses
- Größenordnungen von Projekten
- Faktoren für ein erfolgreiches Projekt

Block 1-3 “Produkte und Prozesse” (45’)
- Produkte in der Software-Entwicklung
- Vorgehensmodelle

Motivation - Ziel

- Software ist zunehmend Teil des täglichen Lebens.
- Die Herstellung komplexer Software ist anspruchsvoll und braucht professionelles Herangehen: Software Engineering.
- Fehler und Qualitätsmängel betreffen immer mehr Menschen in immer weitergehendem Umfang: Ziel „No surprise“- Software.
- Software Engineering soll helfen, für große Software Systeme ähnliche Qualitätsmaßstäbe zu erreichen wie in klassischen Ingenieursdisziplinen.
  - Kostengünstige Entwicklung
  - Hohe Qualität
  - Innerhalb des geplanten Zeitrahmens

Produkt- und Projekttypen

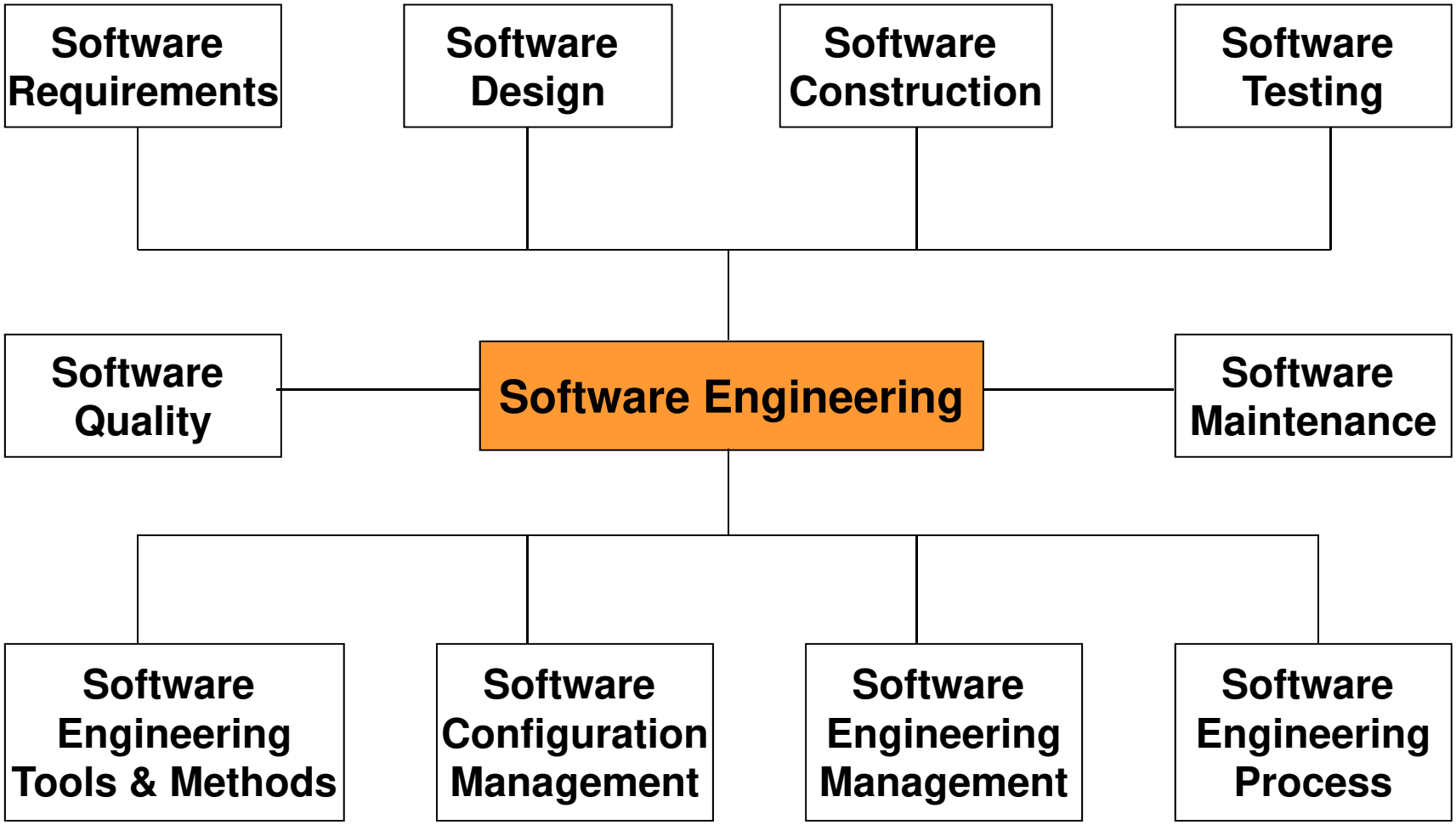
- Unterschiedliche Projekte verlangen unterschiedliche Vorgehensweisen.

Typ	Anforderung	Beispiel
Kommerzielle Software	Benutzbarkeit, Verfügbarkeit, Support	Datenbanktransaktionen, Finanz-/Touristikanwendg.
Eingebettete Systeme	Zeitgesteuert, Sicherheit, Echtzeitanforderungen	Handy, ABS-System, Liftsteuerung, Kraftwerk, Fabriksteuerung
Wissenschaftliche Software	Rechengenauigkeit, Korrektheit, Zuverlässigkeit	Medizinische oder Luftfahrtprogramme Analyse großer Datenmengen, Vorhersagen

Begriffsdefinitionen

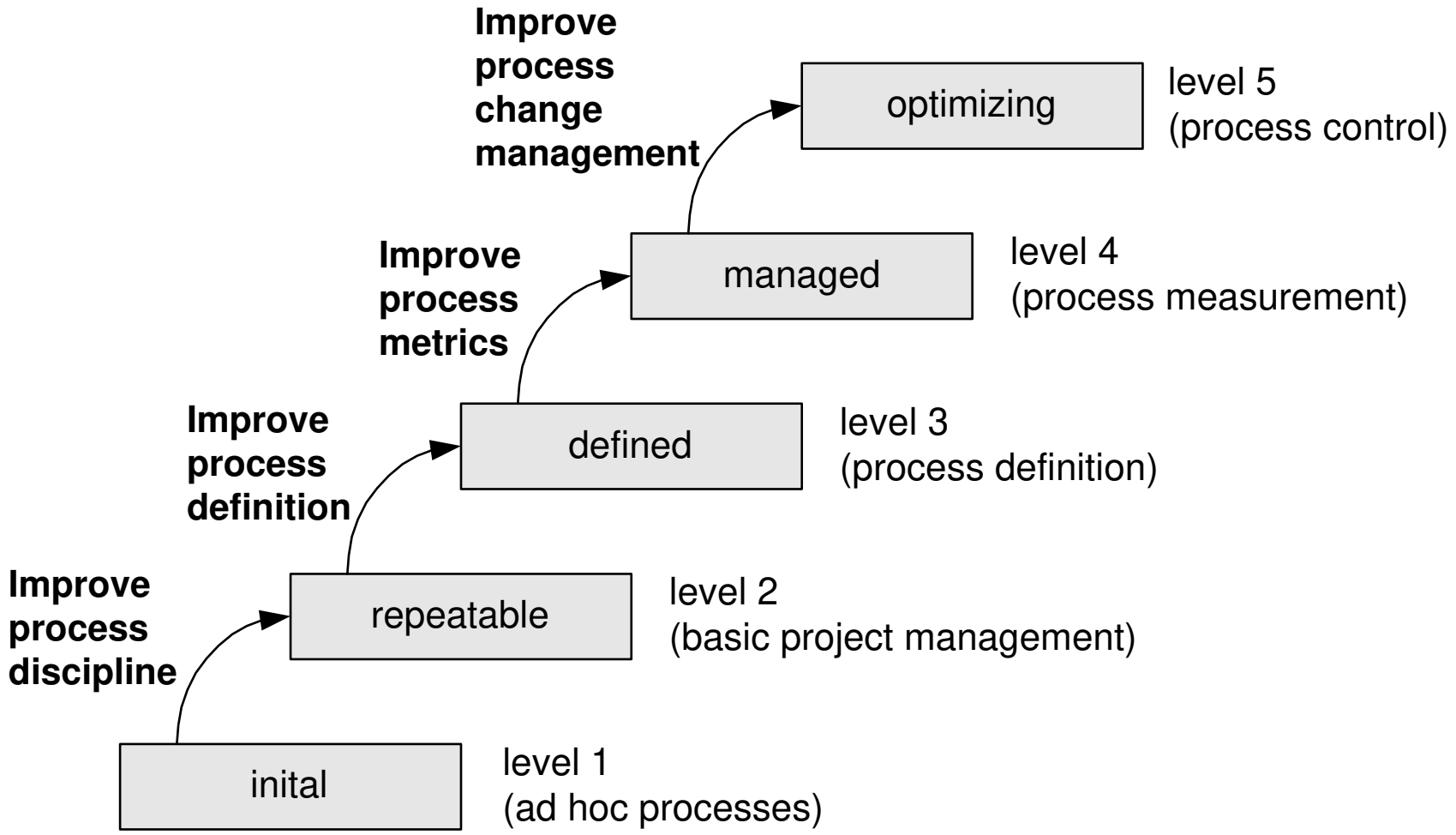
- **Software:** Beinhaltet alle Instruktionen die dem Computer vorschreiben, was er zu tun hat. Programme, die die Hardware des Computers kontrollieren.
- **Software Engineering:** Vorgehensweise zur systematischen Erstellung von Software nach „ingenieurmäßigen Prinzipien“.
  - Systematisch, effektiv,effizient.
- **Analyse:** Erfassen und Verarbeiten von Anforderungen (Pflichtenheft).
- **Design:** Entwurf der internen Struktur des Systems (Systemarchitektur).
- **Validierung:** Überprüfung des Verhaltens eines Programms durch festgesetzte Testfälle.

Software Engineering – Teilbereiche SWEBOK.org



Software Engineering Body of Knowledge – SWEBoK.org

Reifgrade der Software-Entwicklung
Capability Maturity Model Integrated CMMI



CMMI - http://www.sei.cmu.edu/cmmi/

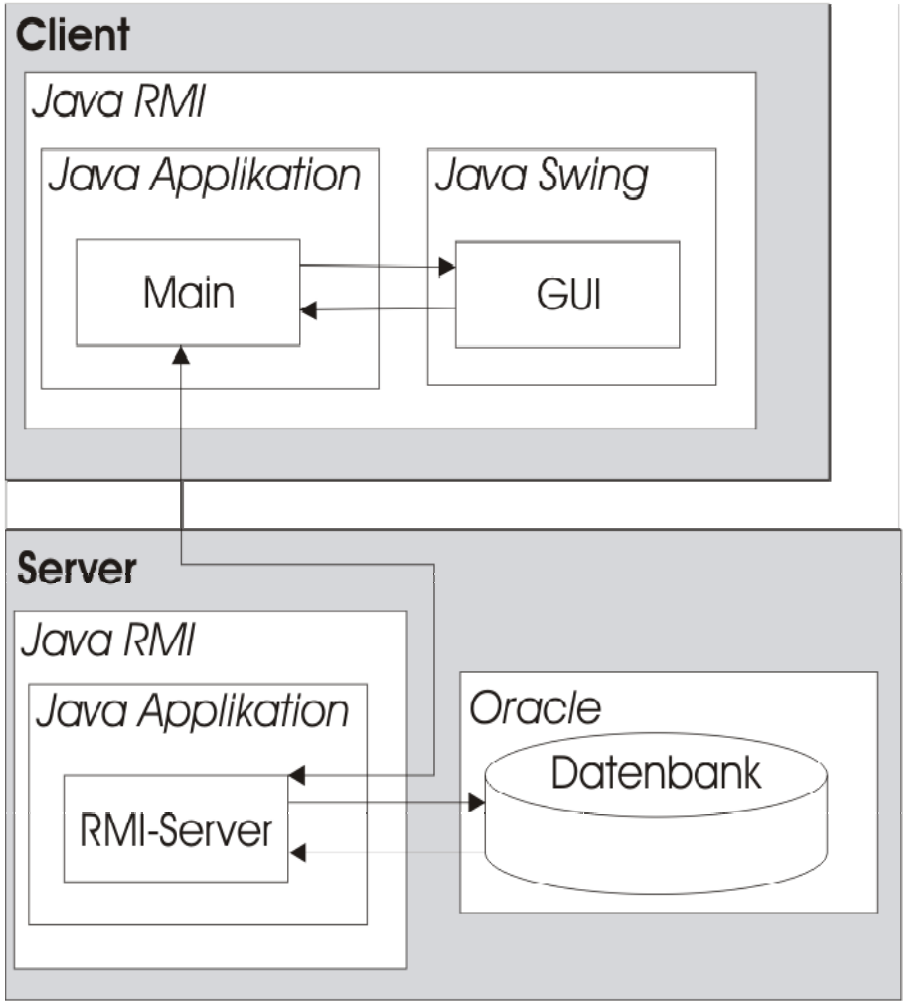
CMMI – Levels 1, 2, and 3

- **Level 1 (initial)**
  - ad-hoc processes („like it is“)
- **Level 2 (repeatable** –
  „basic project management“)
  - Configuration Management
  - ***Software Quality Assurance***
  - Subcontract Management
  - *Project Tracking and Oversight*
  - *Project Planning*
  - ***Requirements Management***
- **Level 3 (defined** –
  process definition)
  - ***Peer Reviews***
  - Intergroup Coordination
  - Software Product Engineering
  - Integrated Software Management
  - Training Programs
  - Organization Process Definition
  - Organization Process Focus

CMMI - http://www.sei.cmu.edu/cmmi/

Architekturbeispiel – Kleines Programm

- Wenige Komponenten
- Wenige Technologien
- Einfache Architektur

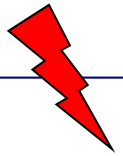


Beispiel: Mangelhafte Software

22. Juli 1962, Cape Canaveral / Florida:

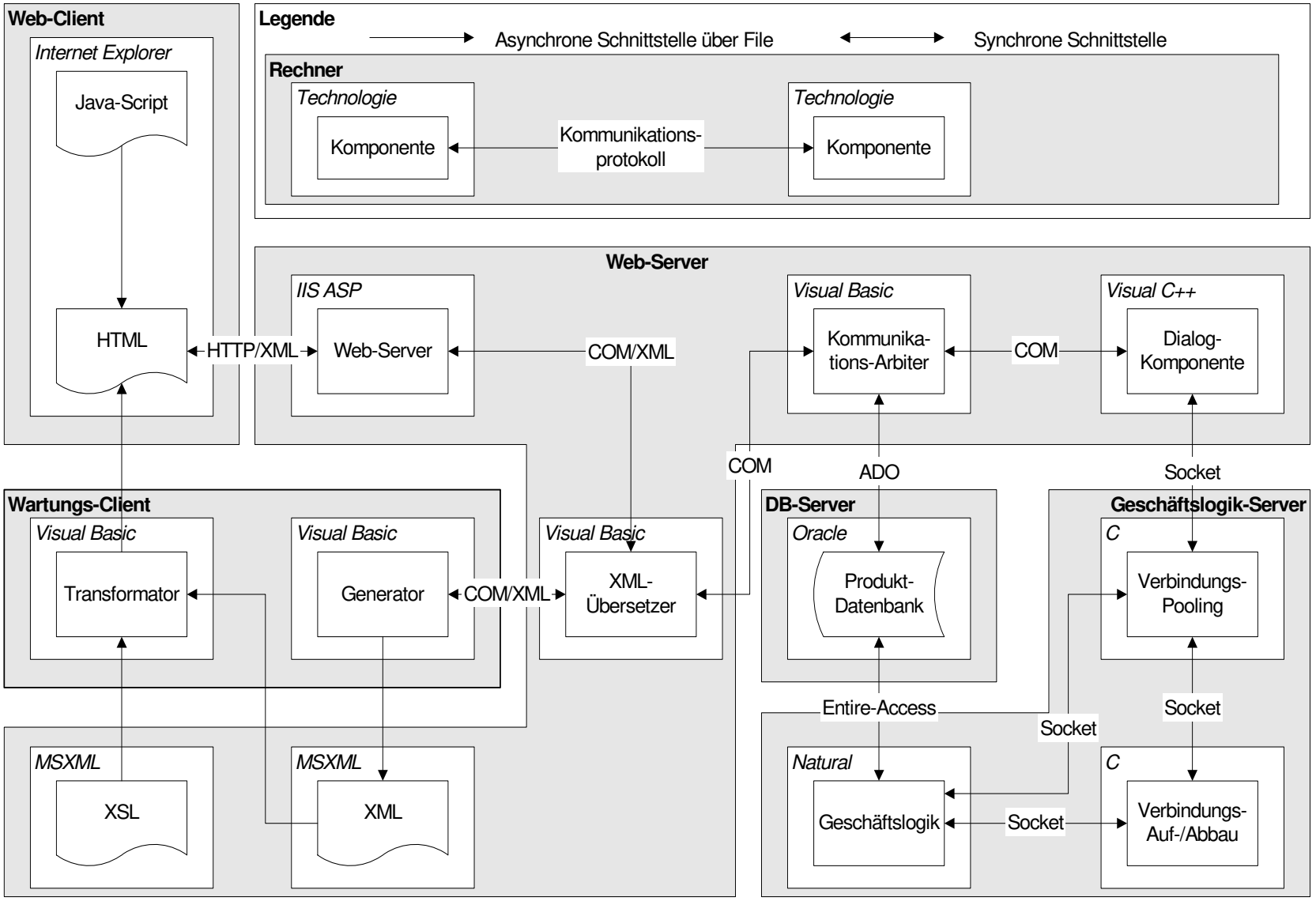
Fehlstart der ersten amerikanischen Venussonde Mariner 1.

Ausschnitt aus dem FORTRAN – Programm zur Steuerung der Trägerrakete (NASA):



```
IF (TVAL .LT. 0.2E-2) GOTO 40
DO 40 M = 1, 3
        W0 = (M-1)*0.5
        ...
        DO 5 K = 1.3
                T(K) = W0
                Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
                D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4(B0**2-X*B0*B1))/Z
                E(K) = H**2*93.2943*W0/SIN(W0)*Z
                H = D(K)-E(K)
        5 CONTINUE
        10 CONTINUE
        Y = H/W0-1

40 CONTINUE
```

Architekturbeispiel – Größeres Programm



Typische Probleme großer SW-Projekte

- Zeitverzögerung bei Software-Projekten
- Überschreiten des geplanten Budgets
- Mangelnde Qualität der Software
- Spätes Erkennen von Design-Fehlern („late design breakage“)
- Schwierige und teure Wartung

SE/QM-Ansätze

- Formale Methoden: Verifikation, bessere Programmiersprachen
- Prozessverbesserung: Produkte, Prozesse, Vorgehensmodelle
- Personen: Training, Motivation, „Kultur guter Arbeit“

Software Engineering Vorlesung
Einheit 1-2 Projekttypen

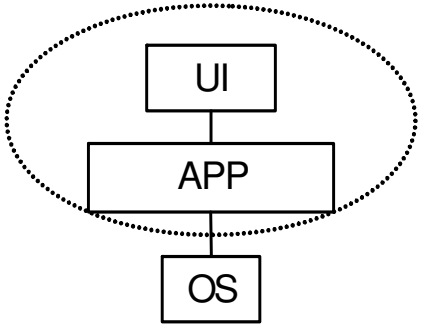
- Überblick Projekttypen
- Größenordnung von Projekten
- Umfeld des SE-Prozesses
- Software Beschaffung
- Faktoren für ein erfolgreiches Projekt

Motivation - Ziel

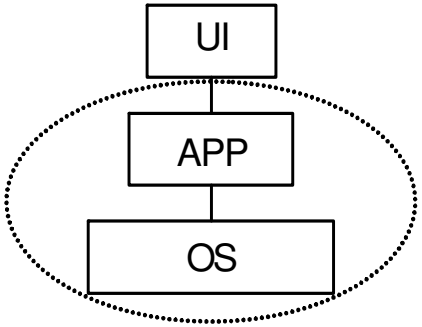
- Software-Herstellung ist nicht-trivial und braucht professionelles Herangehen
- Unterschiedliche Projekte verlangen unterschiedliche Vorgehensweisen
- Komplexität - Einfluss der Projektgröße
- Design-Strategien: Software kaufen oder selber entwickeln
- Risikofaktoren - Warum scheitern viele Projekte
- Erfolgsfaktoren - Was sind Grundlagen für ein erfolgreiches Projekt

Einige Projekttypen

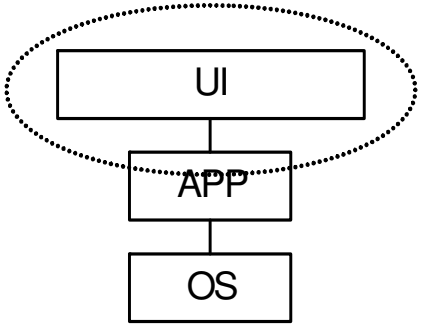
- Ein Software-System besteht typischerweise aus (a) Benutzerschnittstelle (UI), (b) Geschäftslogik der Anwendung (App) und (c) einem Betriebssystem (OS).
- Je nach Software-Typ ist der Schwerpunkt der Entwicklung unterschiedlich.



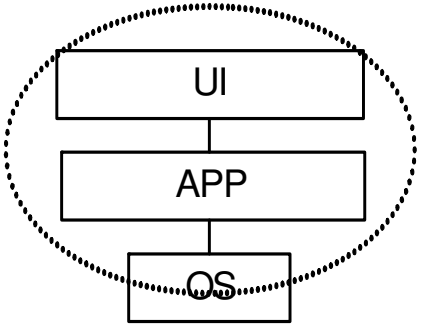
a) commercial system



b) real-time system



c) Web application



d) computer games

UI ... User Interface (Benutzerschnittstelle)
APP … Application (Anwendung)
OS … Operating System (Betriebssystem)

Komplexitätstreiber (PM & SE)

Merkmal	Messbare Attribute	Beispiel
Grösse (PM)	Anzahl der beteiligten Personen	Personen: 50
Dauer und Aufwand (PM)	Anzahl Wochen bzw. Monate	Dauer: 52 Wochen Personen-Jahre: 3 PJ
Verwendete Technologien (SE)	Art, Anzahl und Alter der Technologie	Art: Scriping language, Alter: 4 J., Anzahl: 5
Komplexität (SE)	Anzahl Klassen, Module, Datenbanken; verwendete Technologien, Zeilen Code	Datenbanken: 2 Klassen: 42 Code: 30 000 Zeilen

Projekttypen – Größe eines Projekts

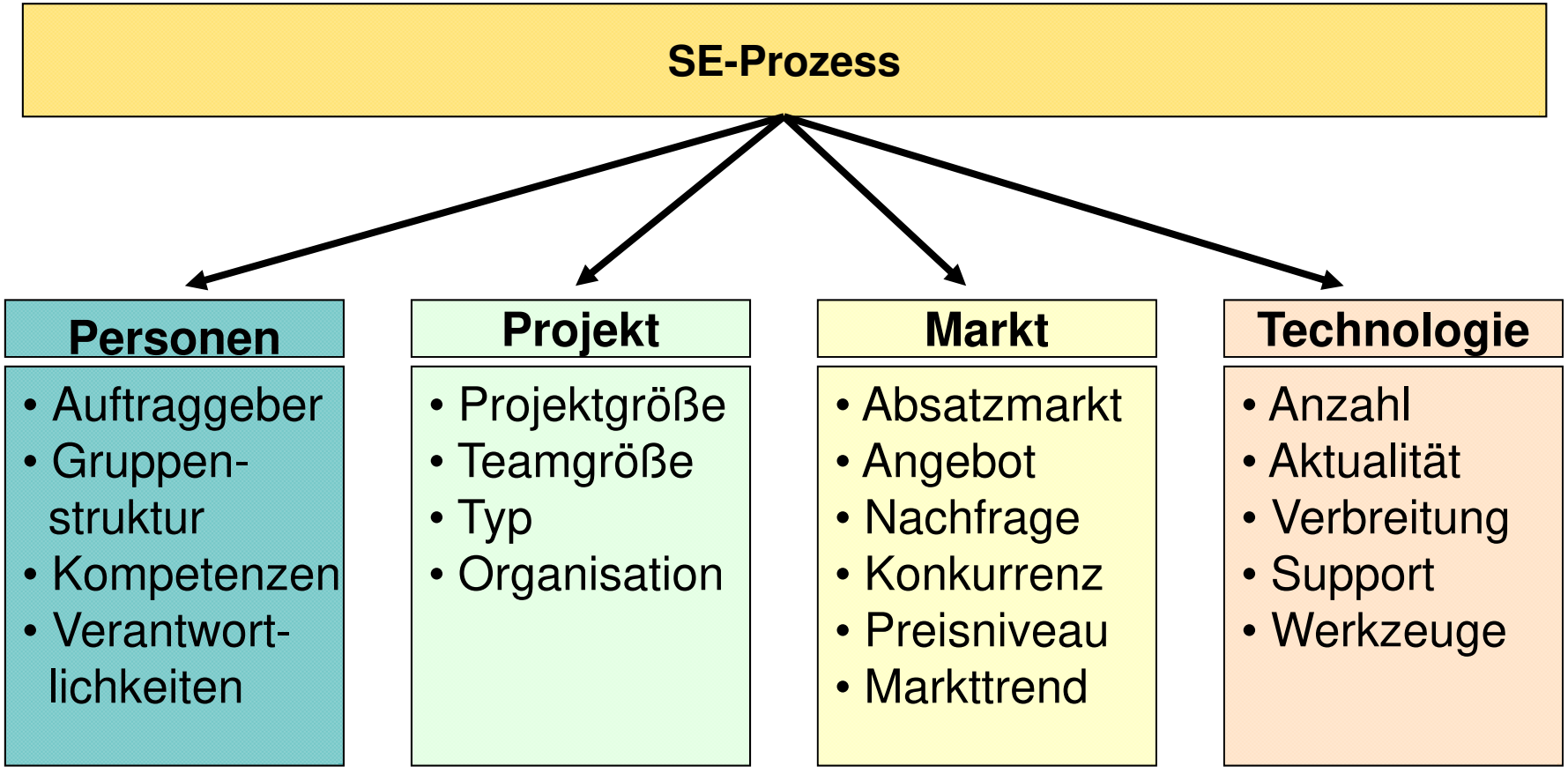
Größe	Kriterien	Beispiele
Klein	Bis zu 6 Personen Monate: 0-8 Anzahl Technologien: <5	Rechenprobleme, Algorithmen
Mittel	10-30 Personen Monate: 9-24 Anzahl Technologien: 5-12	Buchhaltung, Lagerverwaltung
Groß	50-100 Personen Monate: 25-45 Anzahl Technologien: 12-20	Compiler, Datenbank
Sehr groß	Ab 100 Personen Monate: 45-n Anzahl Technologien: >20	Raumfahrt, Atomkraftwerk, elektronische Börse, große Standardsoftware

Was ändert sich mit Zunahme der Größe

- Komplexität (SE, PM)
- Bedarf an Flexibilität (SE, PM)
- Bedarf an Organisation, Planung und Überblick steigt (PM)
- Bedarf an Prozessorientierung (PM)
- Human Factors – Bedarf an Kompetenzen steigt (PM)

- Kommunikationspfade – wer kommuniziert mit wem (SE, PM)
- Testaufwand steigt (SE)
- Saubere Dokumentation ungleich wichtiger, (SE)
  damit das Projekt nachvollziehbar bleibt, vor allem die Schnittstellen
- Versionenverwaltung bzw. Konfigurationsmanagement
  damit kein Versions-Chaos entsteht (SE)

Umfeld des Software Engineering Prozesses



Software Beschaffung 1/2
Kauf oder Entwicklung

	Kauf	Abänderung	Neuerstellung
Entspricht Anforde-rungen	Ungefähr (oft viele ungenützte Funktionen oder ein paar fehlende)	Oft ist keine exakte Anpassung möglich → Beschränkung durch technische Grenzen	Genau
Änderbarkeit	Schwierig, da technische Details oft nicht transparent sind	Ausgangsprodukt wurde selbst hergestellt: leicht änderbar, Durch andere hergestellt: Schwer änderbar	Gut möglich (Dokumentation der Entwicklung und somit techn. Transparenz vorhanden)
Preis	Nach Anforderung + Verbreitung	- durch eigene IT- Abteilung: kalkulierbar - durch andere: kostenintensiver	Teuer

Software Beschaffung 2/2
Kauf oder Entwicklung

Kauf von Software, wenn …

- die gewünschten Anforderungen weitgehend erfüllt werden.
- die individuelle Anpassung (leicht) möglich ist.
- die Kosten niedriger sind als bei Eigenentwicklung.
- ausführliche Dokumentation vorhanden ist.
- guter Support bei Problemen und Fragenstellungen geboten wird.

Vergleich zwischen kommerzieller Software und eingebetteten Systemen

	Embedded Systems	Kommerzielle Software
Steuerung	Ereignisgesteuert, oft auch vollständig automatisiert	Benutzergesteuert
Kosten	Teuer, wegen Neuentwicklung oder Anpassung	Kaufen billiger als selber entwickeln
Zuverlässigkeit	Sehr wichtig	Oft nicht entscheidend
Wartung	Schwierig, z.T. Hardware-technisch unmöglich	Meist professioneller Support
Sicherheit	Müssen sicher sein (safety)	Unterschiedliche Wichtigkeit: Online-Banking, Datenbank Systeme vs. Photobearbeitung
Usability	Benutzerinterface rudimentär oder nicht vorhanden	Oft entscheidend, vor allem bei grosser Konkurrenz
Beispiele	Handysteuerung, Liftsteuerung, ABS-System, Ampel	Datenbanksystem, Web-Applikationen, Texteditor

[SCHU01]

Beispiel Eingebettetes System

Liftsteuerung, Fabriksteuerung

- **Ereignisgesteuert**
  - Knopfdruck
  - Sensoren-Ereignis, wenn Stockwerk erreicht wird
- **Sicherheit**
  - System muss absolut stabil sein
  - Verschiedene Sicherheitsmaßnahmen bei Notfällen
- **Kostenfreundlich**
  - Berechnung des kürzesten Weges bei mehreren Anfragen
  - Standby Modus wenn keine Anfragen
- **Usability**
  - Einfache, leicht verständliche Benutzersteuerung
  - Logische Prozessabfolge

E- Katalog als VO begleitendes Beispiel 1/2

- Zu realisieren ist ein online Katalog, in dem die Produkte …
  - benutzerfreundlich dargestellt werden.
  - selektiert werden können.
  - in einem "Warenkorb" abgelegt werden können.
- Die Bestellliste kann jederzeit editiert werden.
- Beim Verlassen des Online-Shops kann die Bestellung (Verrechnung über Kreditkarte) aufgegeben werden.
- Der Inhalt des e-Katalogs wurde mit modernen Content-Management-Systemen bearbeitet und kann daher auch als CD oder normaler Katalog (Papier) bestellt werden (gegen eine Gebühr).

E- Katalog als VO begleitendes Beispiel 2/2

- **Benutzergesteuert**
  - Grafische Benutzeroberfläche, Web-Applikation, benutzerfreundlich
  - Z.T automatisierte Vorgänge: Datenbankeintrag, Datenbankabfragen, Versenden von Bestätigungs-E-Mail
- **Sicherheit:**
  - Sichere Übertragung vertraulicher Daten (Kreditkartennummer etc.)
  - System gegen unbefugte Zugriffe schützen (intern und extern)
- **Zuverlässigkeit:** Hohe Erreichbarkeit des Systems, auch bei erhöhter Belastung
- **Wartung:** Ohne längeren Ausfall des Systems
- **Kosten:** Erschwinglicher Betrieb
- **Verwendbarkeit:** Einfach, übersichtlich für Normalverbraucher

Interaktives Beispiel

Szenario

- Der E- Katalog soll in akzeptabler Zeit entwickelt werden.
- Wir nehmen an, dass wir das Folgende ausreichend zur Verfügung haben:
  - Ressourcen (Geld, Leute, Materialien)
  - Jede nötige fachliche Kompetenz
  - Infrastruktur (Arbeitsstationen, Entwicklungsumgebung, etc.)

- Schreiben Sie 5 Probleme auf, warum das Projekt dennoch scheitern kann.
- Nennen Sie zu jedem Problem einen Lösungsansatz.

Top Ten Risikofaktoren: warum Projekte scheitern

1. Unvollständige Anforderungen
2. Anwender nicht involviert
3. Zu wenig Ressourcen
4. Unrealistische Zeit- und Kostenpläne
5. Keine Management Unterstützung
6. Häufige Änderung der Anforderungen
7. Qualitätsmängel bei extern vergebenen Komponenten
8. Qualitätsmängel bei extern vergebenen Aufgaben
9. Fehlende Planung
10. Projekt wird nicht mehr benötigt.

[Boehm 1984]

Faktoren für ein erfolgreiches Projekt

1. Einbringung und Berücksichtigung der Anwender
2. Adäquates Projektmanagement: nicht zu viel aber auch nicht zu wenig
3. Anforderungen müssen eindeutig beschrieben, realisierbar und auch überprüfbar sein
4. Flexibler, realistischer Projektplan, der mögl. Verzögerungen berücksichtigt
5. Realistische Kostenschätzung und Budget, inkl. Risikoanalyse
6. Angemessene Ziele
7. Schlüsselteammitglieder haben genügend Projekterfahrung
8. Gute Teamarbeit, funktionierende Kommunikation im Team

Zusammenfassung und Ausblick

- **Zusammenfassung:**
  - Verschiedene Anforderungen erfordern verschiedene Lösungsansätze
  - Mit der Projektgröße ändern sich viele Faktoren des Projekts
    - Komplexität, Bedarf an Planung, Organisation etc.
  - Die Eigenentwicklung von Software ist nicht immer die beste Lösung
  - Unterschiede eingebettete Systeme vs. kommerzielle Software
- **Ausblick:**
  - Vorgehensmodelle, Vergleich einer Auswahl
  - Übersicht über Produkte, was muss beachtet werden
  - Verfolgen von Anforderungen

Software Engineering Vorlesung Einheit 1-3 Software Prozesse und Produkte

Inhalt

- Vorgehensmodelle
- Produkte des V-Modells
- Produktqualitäten
- Verfolgen von Anforderungen

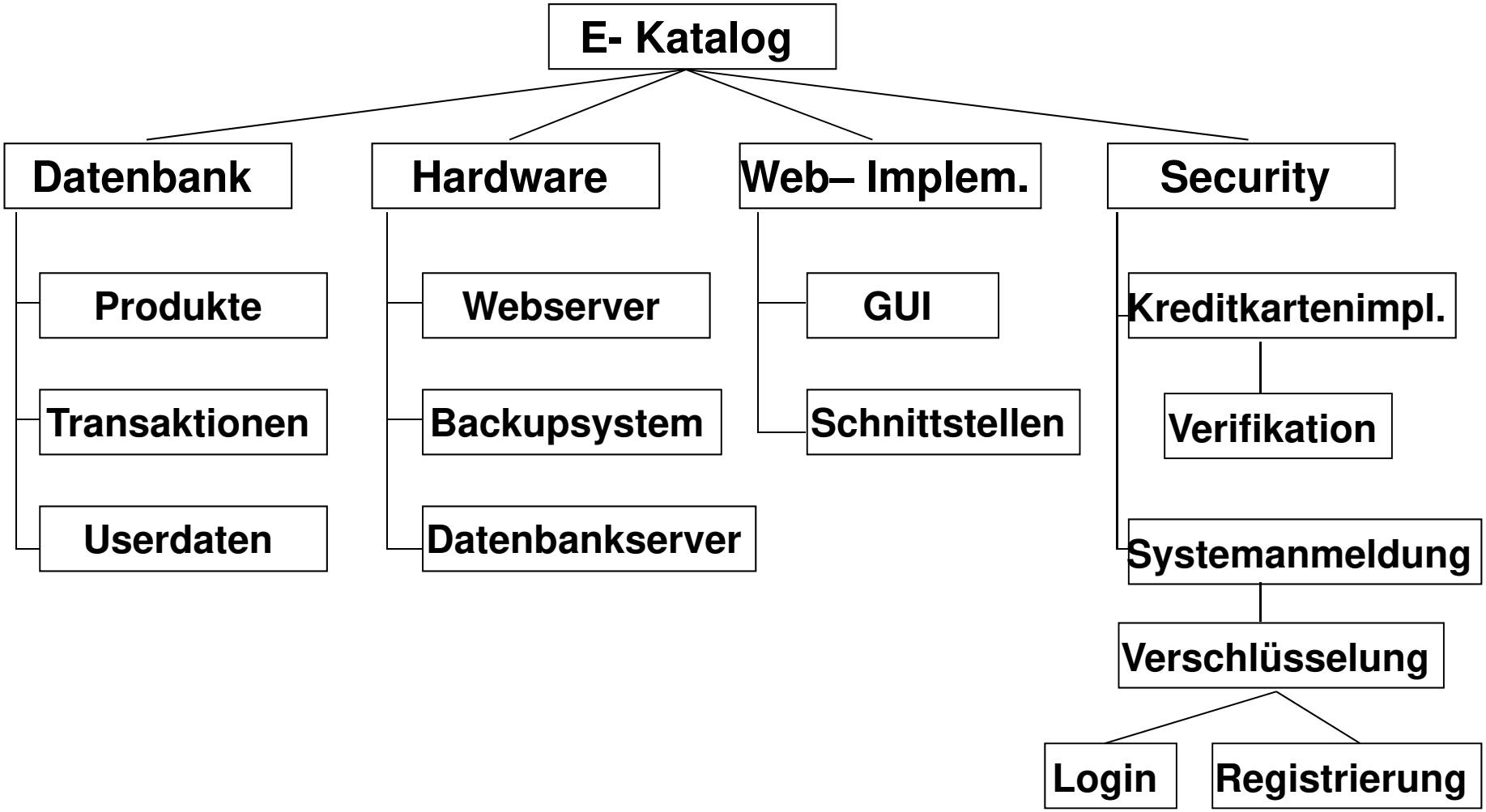
[BPSE Buch, Kap. 3]

Motivation - Ziel

- Software-Entwicklungsprozess strukturieren
- Verbesserung der Kommunikation zwischen allen Beteiligten
- Orientierungshilfe für die Produkte
- Produkte und deren Eigenschaften
- Wann ist welches Vorgehensmodell sinnvoll

CMMI Ebenen 2 & 3 – wiederholbare Produktmerkmale, Prozesse und Methoden

E-Katalog Projektmanagement
Produktbaumstruktur (PBS)



Produktqualitäten 1/2

- Anforderungen
  - stabil und ausreichend genau beschrieben, übersichtliche Darstellung
  - Systembeschreibung, Anwendungsfälle, Nichtfunktionale Anforderungen
- Design
  - Skalierbarkeit, Performance, Erweiterbarkeit
  - Übereinstimmung mit Anforderungen, ausreichend dokumentiert
- Testplan
  - Testrahmen, Teststrategie, Testfälle, Testzeitplan
  - Nachvollziehbare Beschreibung, Anforderungen vollständig abgedeckt
- Testbericht
  - Version des Systems und der DB, Testfälle mit unerwartetem Ergebnis
  - verwendete Eingabedaten, welche Korrekturen sind nötig

Produktqualitäten 2/2

- Benutzerschnittstelle
  - Usability für Zielgruppen, Überprüfung mit Prototyp
  - Dialoge nicht überladen, einheitliches Layout, Fehlermeldungen einheitlich
- Datenbank
  - Redundanzfrei, Wachstum der Datenbestände berücksichtigen
  - Übereinstimmung mit Klassenmodellen, Integritätsbedingungen
- Technische Dokumentation
  - Verständlich, korrekt, vollständig, Format und Stil ansprechend
  - Lehrbuchteil, Nachschlageteil, Stichworte/Glossar, Online-Hilfe
- Projektplan
  - Alle Tätigkeiten mit Verantwortlichen und Mitarbeitern
  - Realistische Angaben für Aufwand, geeignete Darstellung

Verfolgen von Anforderungen

- Validierung der Anforderungen → beschreiben Anforderungen was der Kunde haben möchte? (Überprüfung z.B mit Prototyp)
- Anforderungen an die Anforderungen:

[DROB03] Folie 83 ff

- Gültigkeit
- Vollständigkeit
- Überprüfbarkeit
- Verfolgbarkeit
- Konsistenz
- Realismus
- Verständlichkeit
- Anpassbarkeit

- Vier Richtlininien:

[BOEH00]

- **Value-driven requirements**
  Erfordert Geschäftsfall-Analyse
- **Shared-vision driven requirements**
  Stakeholder bewerten regelmässig die Anforderungen → gesamtheitliche Sicht ist wichtiger als präzise Anforderungen
- **Change-driven requirements**
  Die Erweiterbarkeit und Anpassung wird oft vernachlässigt
- **Risk-driven requirements**
  Wie detailliert müssen Anforderungen sein?
  “If it’s risky to leave it out, put it in“

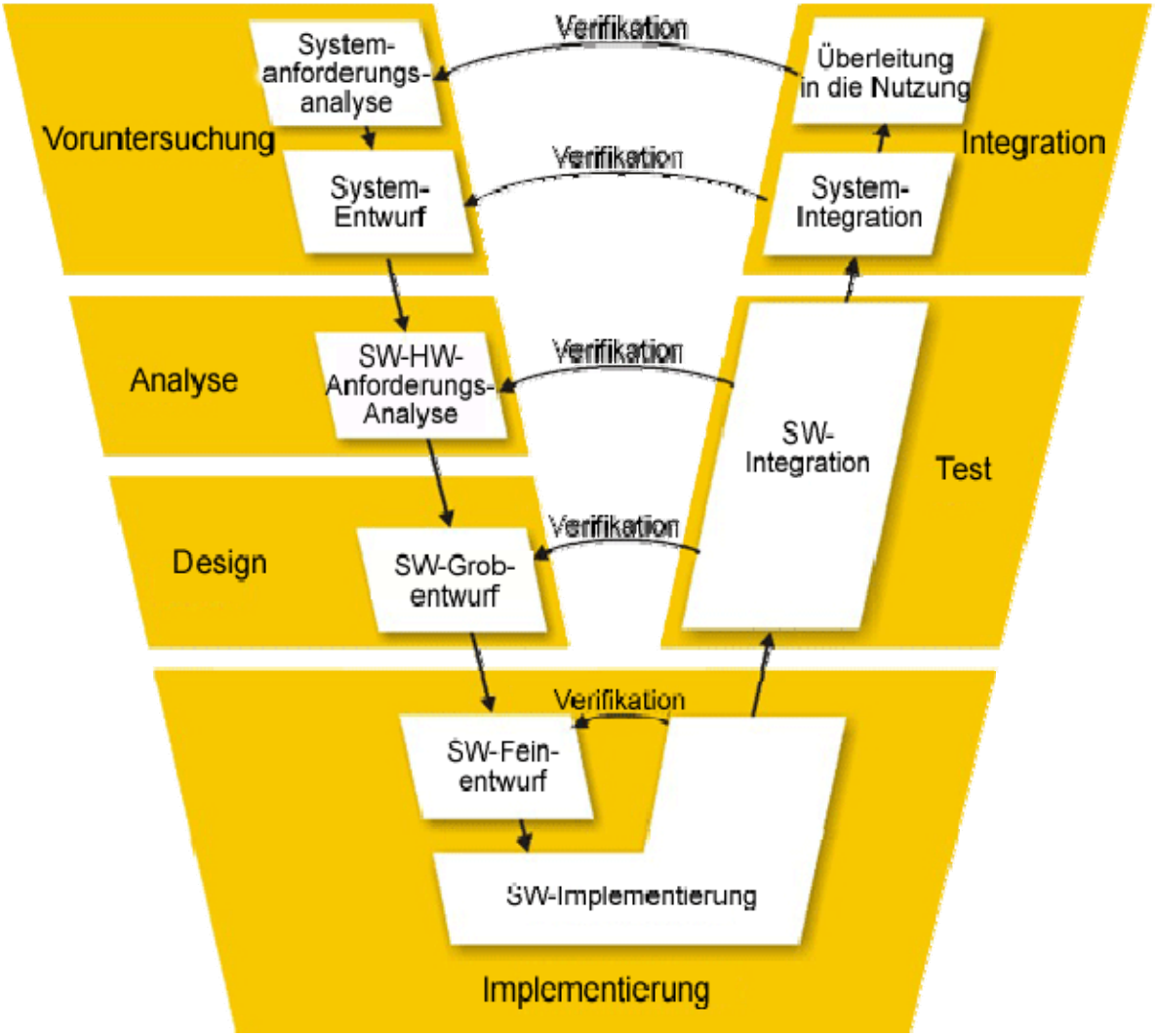
Siehe auch „Requirements Analysis & Specification“ VU

Überblick und Koordination – Charakteristika einiger Prozessmodelle

Modell	Phasen	Schwerpunkte
V-Modell	Voruntersuchung, Analyse, Design, Implementierung, Test, Integration	Dokumentation, Qualitätssicherung, Verbesserung der Kommunikation aller Beteiligten
Inkrementelles Modell	Analyse, Entwurf, Implementierung Integration, Auslieferung an Kunden	Minimale Entwicklungszeit, Risikominimierung, kurze Phasen
Iteratives Modell, z.B Unified Process	Etablierung, Entwurf, Konstruktion, Übergang	Architekturzentriert, Anwendungsfall gesteuert
Extreme Programming (XP)	Coding, Testing, Listening, Designing laufen dauern parallel ab und nicht in Phasen	Frühe Fehlererkennung, Minimale Entwicklungszeit, Schnelle Anpassung an sich ändernde Anforderungen

[BALZ98] S. 135

V-Modell


Voruntersuchung
System-anforderungs-analyse
System-Entwurf
Analyse
SW-HW-Anforderungs-Analyse
Design
SW-Grob-entwurf
SW-Fein-entwurf
SW-Implementierung
Implementierung
Verifikation
Verifikation
Verifikation
Verifikation
Verifikation
Überleitung in die Nutzung
System-Integration
Integration
SW-Integration
Test

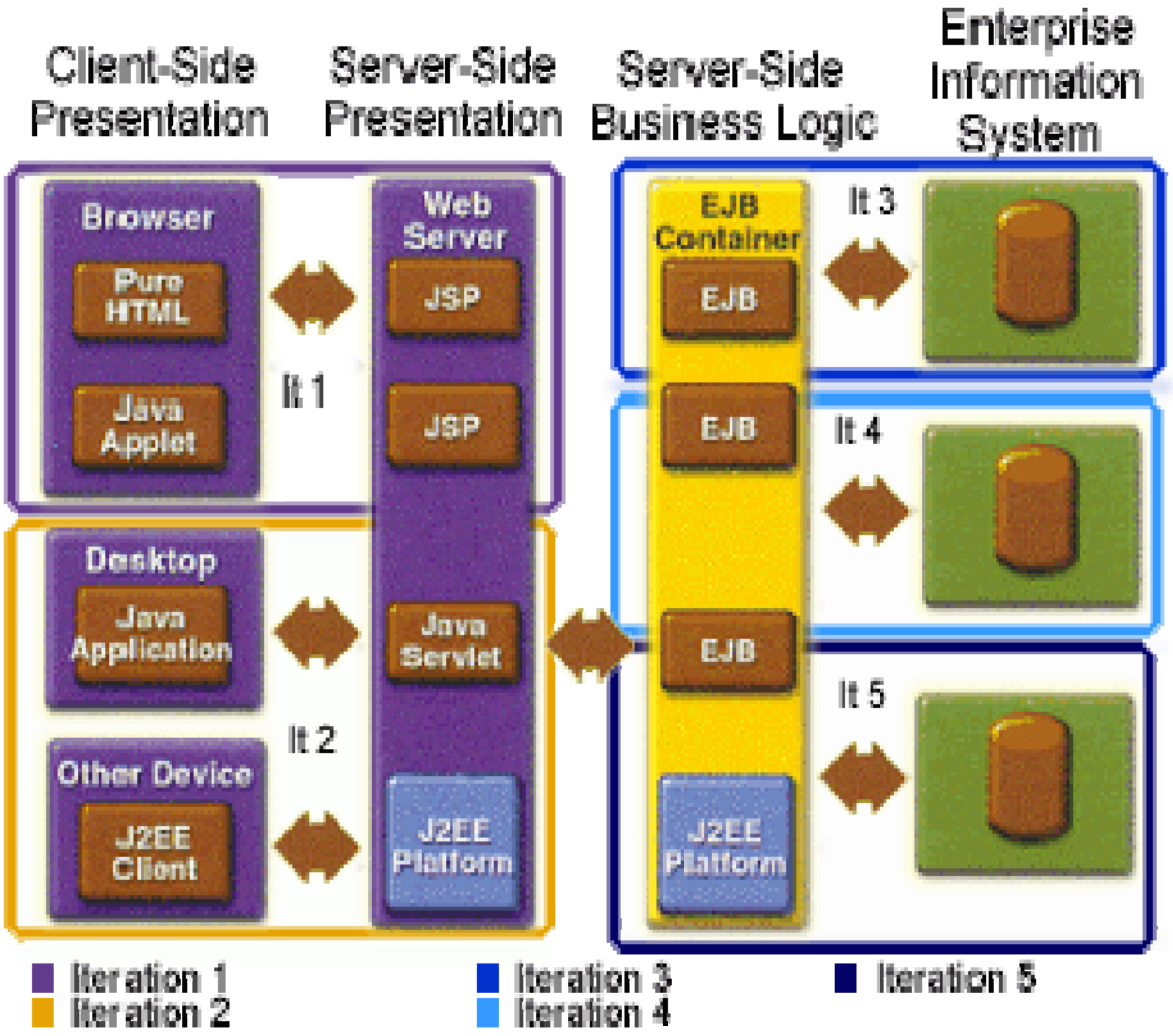
[BVM]

V-Modell

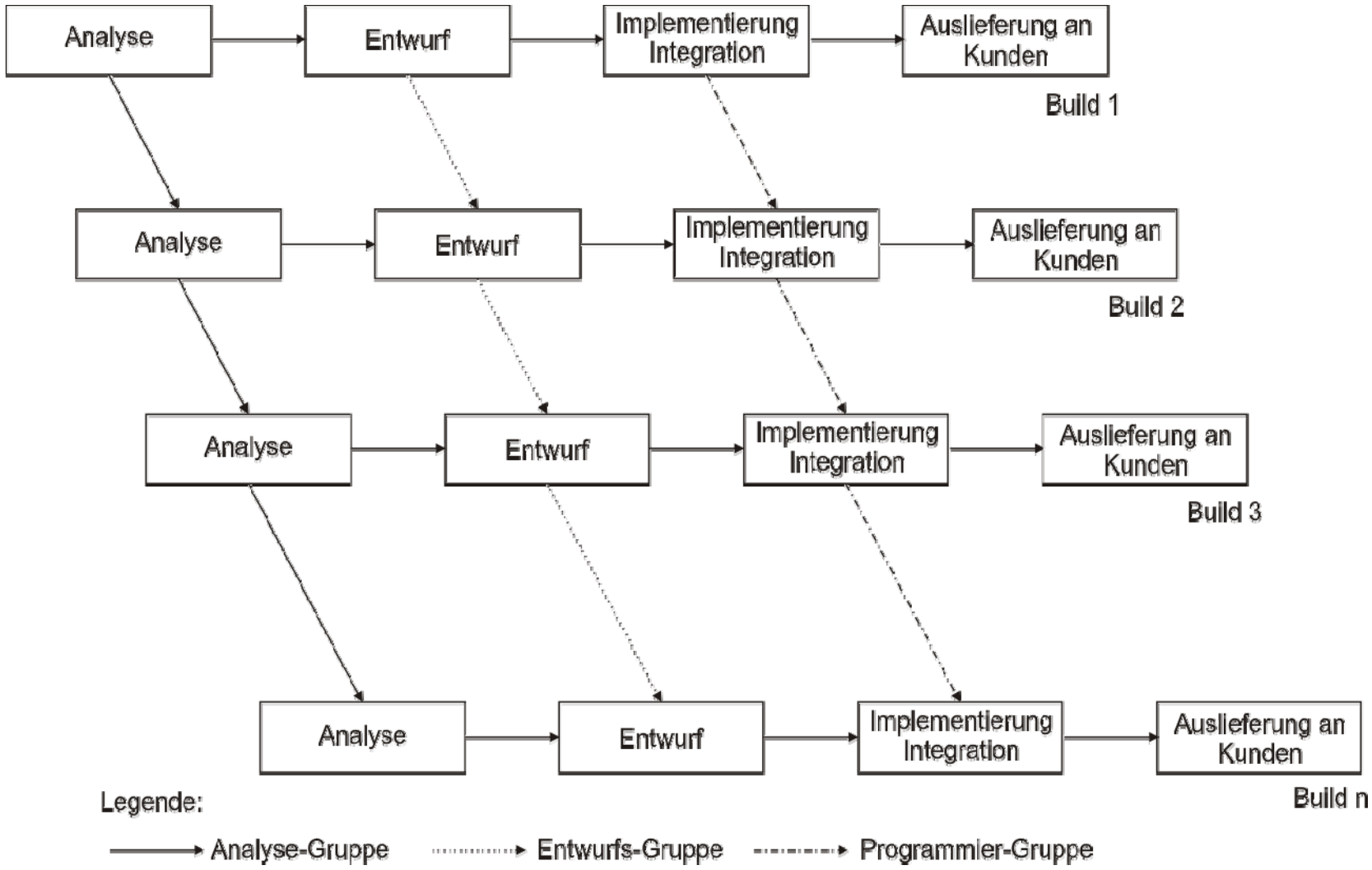
- Gesamtaufwand sinkt gemessen am Lebenszyklus einer IT-Anwendung (Kostenreduktion)
- Kürzere Einarbeitungszeiten für neue Mitarbeiter im Projekt
- Einfachere Kontrolle des Projektfortschritts (auch für den Auftraggeber)
- Projekte leicht vergleichbar → genauere Aufwandsabschätzung zukünftiger Projekte
- 4 Submodelle: Systemerstellung, Qualitätssicherung, Konfigurationsmanagement, Projektmanagement
- **Anwendung:** für kleine Projekte zu detailliert, für große Projekte mit hohem Qualitätsanspruch geeignet (speziell Embedded Systems)

[BVM]

Implementierung mit Iterationen / Inkrementen



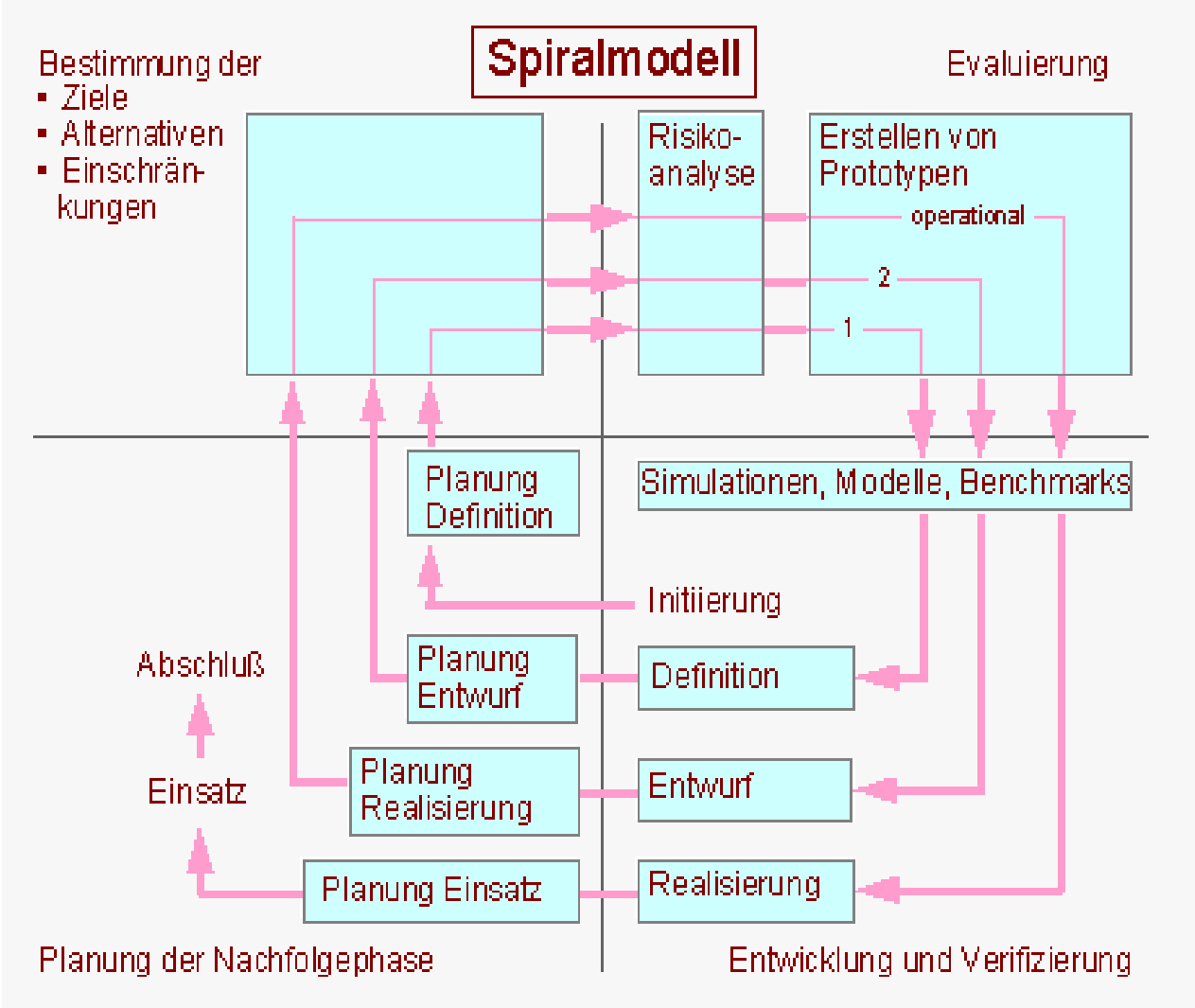
Inkrementelles Vorgehensmodell



Inkrementelles Modell

- Stufenweise Entwicklung des Produkts
- Integration findet kontinuierlich statt
- Koordinierung in kleinen Etappen
- Planung wird leichter
- Planverfolgung einfacher
- Kontinuierlich hohe Qualität
- Bei unklaren Anforderungen gut geeignet
- Jeder Zyklus wird mit Meilenstein abgeschlossen
- Zyklen werden gemeinsam geplant
- **Anwendung:** große, komplexe Systeme mit langer Entwicklungszeit, wenn das Basisprodukt schnell beim Kunden sein muss, dann weiterentwickelt wird

Spiralmodell



[Boehm, ZOPF03]

Spiralmodell

- Risikogetriebenes Vorgehensmodell
- Für jedes Teilprodukt und jede Verfeinerungsebene werden vier zyklische Schritte durchlaufen
- Ziele eines Zyklus ergeben sich aus den Ergebnissen des letzten Zyklus
- Sehr flexibles Modell
- Hoher Managementaufwand, da oft neu entschieden werden muss
- Testaufwand ist einfach einzuschätzen
- Anwendung: größere, risikoreiche Projekte, interne Softwareentwicklungen

XP: Extreme Programming

- Iteratives Vorgehensmodell, bei jeder Iteration wird mehr Funktionalität realisiert (sehr kurze Iterationen, Refactoring)
- **Coding:** kontinuierliche Integration, Programmieren in Paaren (der eine programmiert, der andere testet, Rollen können wechseln)
- **Design:** einfache Dinge nur einmal tun
- **Dokumentation:** selbstdokumentierend im Quelltext
- **Test:** Unit-Tests, Testfälle im Vornherein spezifizieren
- **Integration:** mehrmals täglich soll der Quellcode in die zentrale Code-Basis eingefügt werden, inklusive Tests
- **Anwendung:** Projekte mit schnell wechselnden oder vagen Anforderungen, mit kleinen Entwicklergruppen (bis 12 Personen), zeitkritische Projekte

http://agilemanifesto.org/

Interaktives Beispiel – e-Katalog

- Es soll eine Web-Applikation realisiert werden, die Produkte visualisiert darstellen kann
- Produkte können in einen Warenkorb gelegt werden
- Der Inhalt des Warenkorbs kann jederzeit bearbeitet werden
- Bestellung von Produkten (Verrechnung über Kreditkarte)
- Der Inhalt des e-Katalogs wurde mit modernen Content-Managementsystemen verarbeitet und kann so bei Wunsch auch per CD oder als normaler Katalog (Papier) bestellt werden (gegen eine Gebühr)
- Das Projektteam umfasst 15 Personen, der e-Katalog soll ständig verbessert und weiterentwickelt werden
- **Welches Vorgehensmodell wäre für dieses Projekt geeignet und warum?**

Zusammenfassung

- Unterschiedliche Projekttypen erfordern die Wahl des richtigen Vorgehensmodells
- Produkte haben mehrere Qualitäten, die zu beachten sind
- Im gesamten Projekt müssen die Anforderungen immer gegenwärtig bleiben, d.h. es muss überprüft werden, ob die gewünschten Anforderungen erfüllt werden und geänderte Anforderungen müssen berücksichtigt werden.

Referenzen

Bücher

[BPSE10] Schatten, Biffl, Winkler, et al. Best Practice Software-Engineering - Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen; Spektrum Akademischer Verlag, 2010.

[Balz98] Balzert H.: Lehrbuch der Software-Technik, Heidelberg: Spektrum Akad. Verlag, 1998

[Boeh04] Boehm B., Turner R.; *"Balancing Agility and Discipline",* Addison-Wesley, 2004.

[IEEE90] IEEE, 1990: IEEE Standard Glossary of Software Engineering Terminology. IEEE STD 610.12-1990

[Kruc99] Kruchten P.: The Rational Unified Process, Addison-Wesley, 1999

[SWEBoK04] Guide to the Software Engineering Body of Knowledge, IEEE, 2004.

[Somm10] Sommerville I.; Software Engineering; 9th ed., Addison Wesley, 2010.

[Vlie08] Vliet, H.: Software Engineering - Principles and Practice. Wiley & Sons, 2008.

[Wall11] Wallmüller, E.; Software Quality Engineering; 3. Auflage, Hanser, 2011.

Fachartikel

[Boeh00] Boehm, B.: Requirements that Handle IKIWISI, COTS,and Rapid Change, Juli 2000

[Boeh88] Boehm, B. W.: A spiral model of software development and enhancement. Computer, 21(5):61–72, May 1988.

Web-Ressourcen

[QSE] http://qse.ifs.tuwien.ac.at

[BPSE] http://bpse.ifs.tuwien.ac.at/

http://best-practice-software-engineering.blogspot.com/

http://bpse.ifs.tuwien.ac.at/podcast.html

[CMMI] http://www.sei.cmu.edu/cmmi/

[IEEE] IEEE Software Engineering Body of Knowledge (SWEBOK) www.swebok.org

[PMBoK] http://www.pmi.org/PMBOK-Guide-and-Standards.aspx

[Pris03] TU-Chemnitz: Lehrveranstaltung: Planung und Realisierung von Informationssystemen, http://www.tu-chemnitz.de/wirtschaft/wi1/lehre/2002_ws/pris/v/pris_v9.pdf

[Scha02] Schach, S.: Object-Oriented and Classical Software Engineering, WCB/McGraw-Hill, 2002 http://www.mhhe.com/engcs/compsci/schach5/pps/schach5-chap05-14%5B1%5D.htm

[Well] Wells, D.: When should Extreme Programming be Used http://www.extremeprogramming.org/when.html

[Zopf03] Zopf, S.: Lehrveranstaltung: Fortgeschrittene Aspekte des Qualitätsmanagements, http://qse.ifs.tuwien.ac.at/courses/F_A_QM/188151_VO.htm

TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

Technik und Werkzeuge

Software Engineering & Projektmanagement VO
(188.410)

Peter Frühwirt
peter.fruehwirt@qse.ifs.tuwien.ac.at

Agenda

- Source Code Management
- Build Management

- Kommunikation im Team

- Komponentenorientierte Software-Entwicklung

Sourcecode-Management (SCM) Systeme

- Motivation
  - (verteilte) Teamarbeit →Parallele Projektentwicklung
  - Verwaltung von Artefakten
    - Source Code, Dokumentation, Diagramme…
  - Parallele Entwicklungslinien

Sourcecode-Management (SCM) Systeme

- Motivation
  - (verteilte) Teamarbeit →Parallele Projektentwicklung
  - Verwaltung von Artefakten
    - Source Code, Dokumentation, Diagramme…
  - Parallele Entwicklungslinien

- Änderungen transparent machen
- Entwicklungsgeschichte nachvollziehen
- Zugriff auf ältere Versionen ermöglichen
- Gleichzeitige Entwicklung mehrerer Zweige

Versionierungssysteme

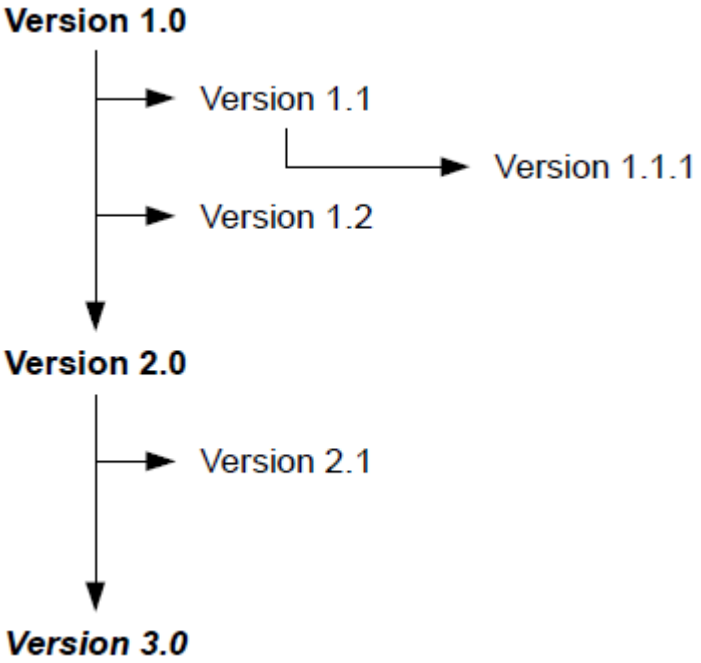
- Entwicklungsgeschichte
  - Revision-Control-Systeme (RCS)
    - Auf Basis einzelner Dateien
  - Concurrent-Version System (CVS)
    - Zentrales, Server-basiertes Repository
  - Verteiltes SCM
    - jeder Entwickler hat sein eigenes Repository
    - mit jedem beliebigen anderen Repository abgleichbar

Zentrale Systeme	CVS, Subversion (SVN)
Verteilte Systeme	Git, Mercurial, BitKeeper

Versionierungssysteme

- Terminologie
  - Check-out
    - Lokale Kopie einer Version anlegen
  - Check-in (Commit)
    - Änderungen seit dem letzten Update (changeset) in das Repository übernehmen
    - Commit-Message
  - Tagging
    - Markieren eines bestimmten Zustands des Systems
  - Merging
    - Änderungen aus einem Branch in einen anderen übernehmen

Parallele Entwicklungspfade

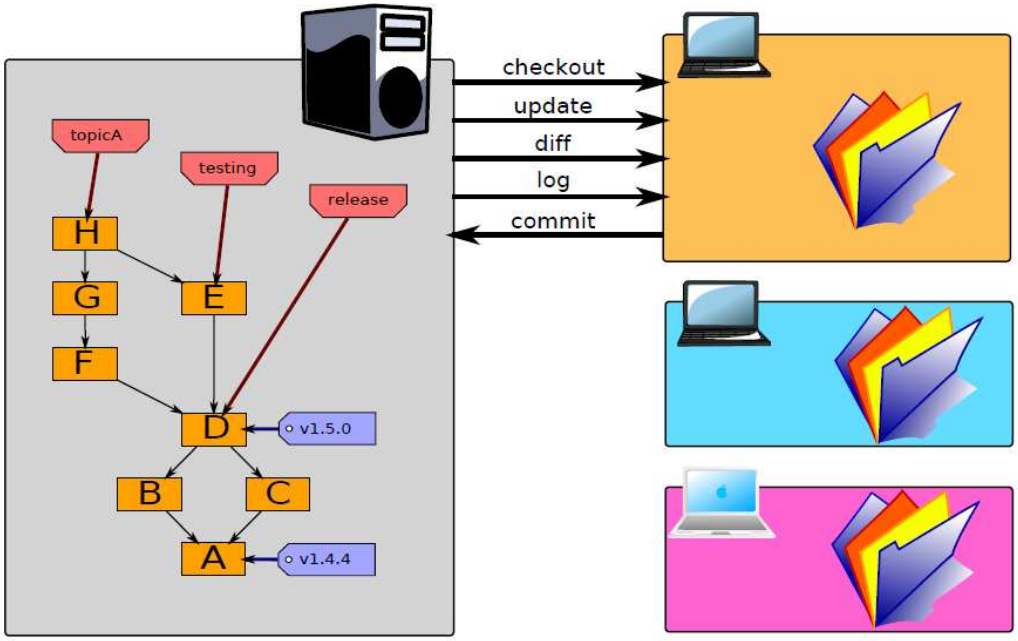


- Branching
  - Abspaltung einer Version
  - Entwickeln in parallelen Strängen
- Wartungsaufgaben
- Experimentelle Features

Zentralisierte Systeme

- Server speichert Versionen
  - Single-Point of Failure
  - Flaschenhals

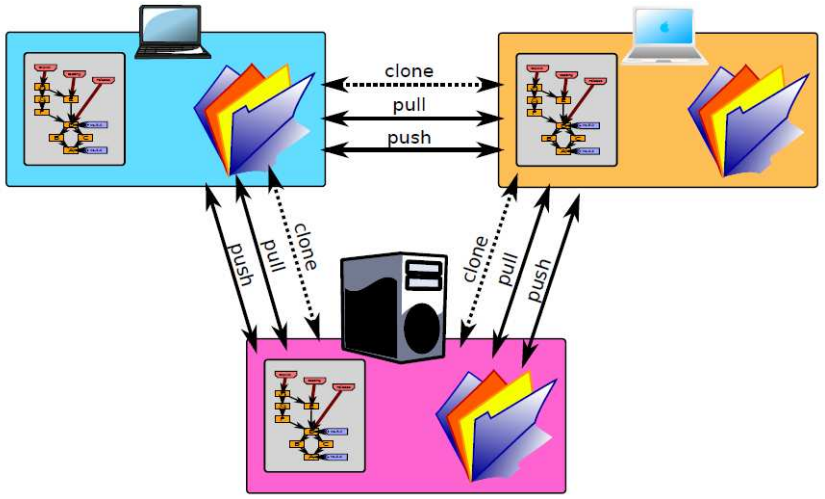
- Operationen erfordern Netzwerkverkehr



http://excess.org/article/2008/07/ogre-git-tutorial/

Verteilte Systeme

- Repository wird geklont
  - Große Datenmengen
  - Viele kleine Dateien
- Projektmanager mit „Main-Repository“
  - Pull-requests
- Jede Komponente in einer eigenen Repository
  - Minimierung der zu kopierenden Datenmenge
- Lokale Operationen
  - Lokale Versionierung



http://excess.org/article/2008/07/ogre-git-tutorial/

Zentral vs. Verteilte Systeme

- Empfehlungen
  - Code sauber und konsistent halten
    - Kompilierbar
    - Alle Tests positiv
    - Regelmäßig commiten
  - Nichts einchecken, das nicht erzeugt werden kann

- Performance
- Verwaltungsaufwand
- Scope
- Branching
- Offline Modus

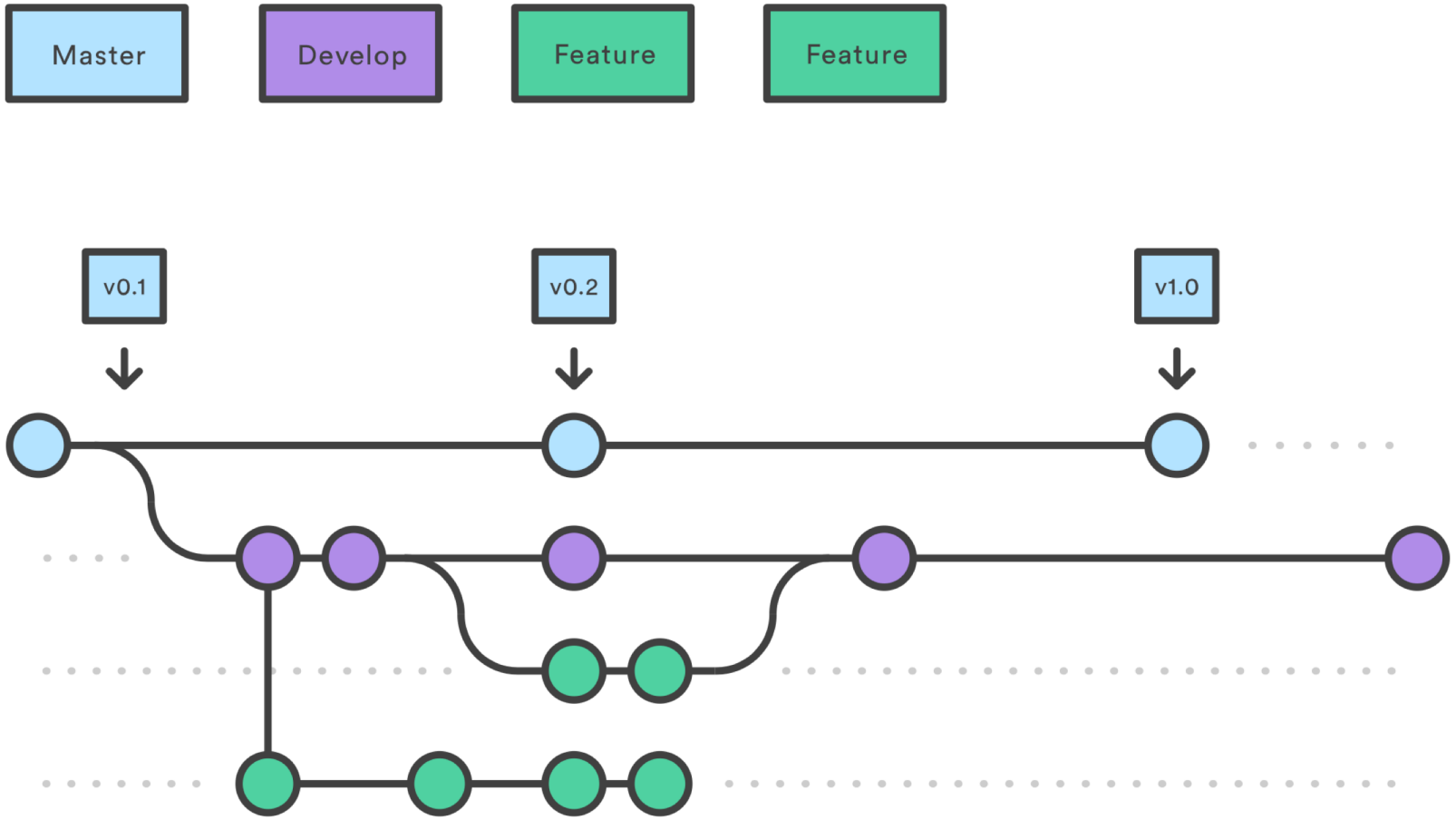
Umgang mit Konflikten

- Teamarbeit
  - Mehrere Personen arbeiten an derselben Datei
  - Konfliktpotential
- Strategien
  - Locking
  - Merging

- Arbeitsablauf - SVN
  - Update
  - Lokale Änderungen durchführen
  - Update
    - Änderungen remote und lokal
    - Merge
  - Commit

- Arbeitsablauf – Git
  - Kopie des Haupt-Repository
  - Lokale Änderungen durchführen
  - Lokale Commits
  - Merge
    - Push
    - Pull Request

git workflow



Quelle: https://www.atlassian.com/git/tutorials/comparing-workflows

Arbeitsabläufe im traditionellen Software Engineering

- Vielzahl an sich wiederholenden Aufgaben
- Manuelle Ausführung
  - Lästig
  - Fehleranfällig
  - Langsam
  - Unmöglich
- Software-Entwicklung sollte sein
  - Deterministisch
  - Fehlerfrei
  - Effizient

Was sollte automatisiert werden?

- Code Quality Checks
- Validierung von Source Code
- Code-Generierung
- Kompilierung
- Testausführung
- Informationssammlung für Reporting/Dokumentation
- Packaging
- Deployment
- Dependency Management

- Wie oft?

"The most fundamental part of the daily build is the 'daily' part. . . . Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead."

James McCarthy

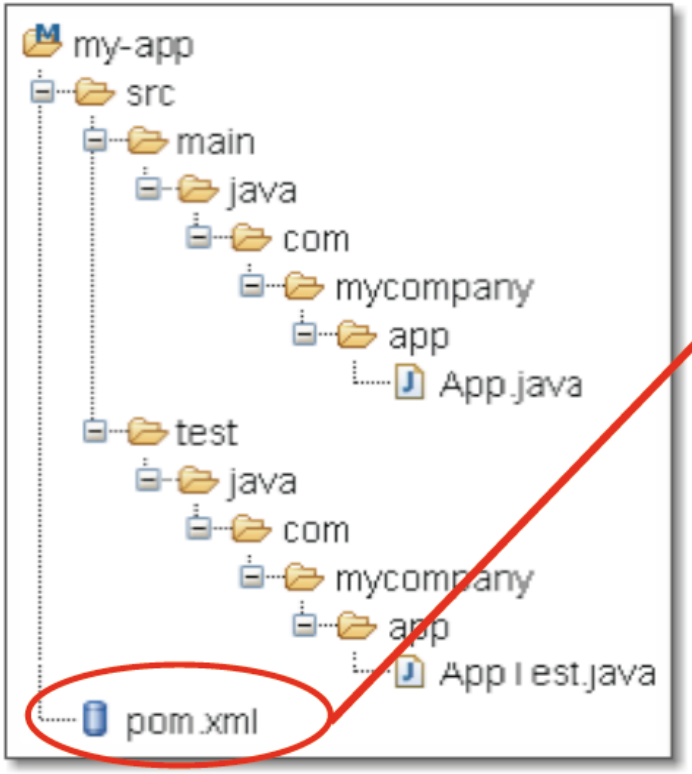
Maven - Build Management System

- Build Tool
  - Kompilierte Applikation automatisch erzeugen
- Vereinfacht Management von Java Projekten
- Dependency Management
- Deployment Tool
- Dokumentation aus javadoc

- maven
  - Best-Practices
  - Convention over Configuration
    - Archetypes
  - Project Object Model (POM)

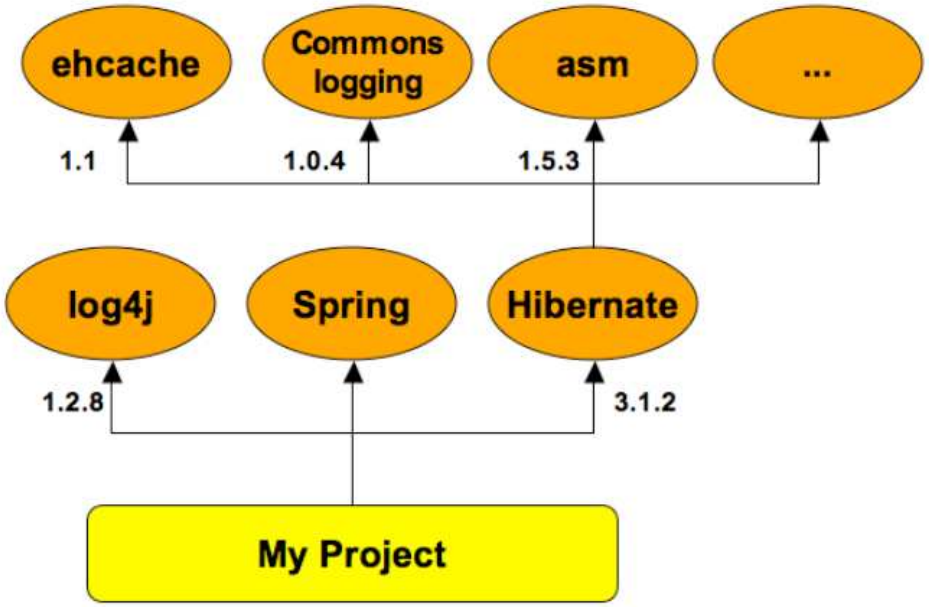
- Ant
  - Make
  - Build Xml
  - Definition von Tasks

Convention over Configuration




```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>my-app</artifactId>
    <packaging>jar</packaging>
    <name>Sample application</name>

    <dependencies>
      ...
    <dependencies>


</project>
```

Dependency Management



- Entwicklung im Team
  - Konsistenter Zustand
  - Entwicklung, Tests, Auslieferung
- Aktualisierung
  - Inkompatibilität
- transitive Abhängigkeiten
  - Versions-Konflikte

Reporting und Dokumentation

- Generelle Projektinformationen
  - Version, Name, Beschreibung
  - Beitragende
  - Lizenz
  - Entwicklungswerkzeuge
- Source Code (javadoc)
- Code Quality Reports
  - Test Reports
  - Checkstyle
  - Component Dependency Reports
- Dokumente
  - Architekturbeschreibung
  - Anwenderhandbuch
  - Website

Maven Build-Lifecycle

validate	validate the project is correct and all necessary information is available.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.

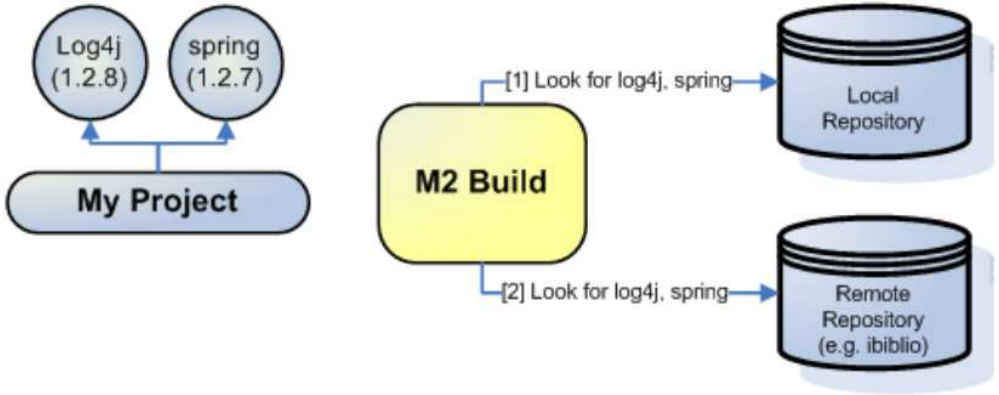
Maven Build-Lifecycle

test-compile	compile the test source code into the test destination directory
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
post-integration-test	perform actions required after integration tests have been executed. This may including cleaning up the environment.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Dependency Management in Maven

- Abhängigkeiten in pom.xml deklarieren
  - Name, Version
- Transitive Abhängigkeiten müssen nicht deklariert werden
- pom.xml-Datei im Sourcecode-Management-System ablegen
  - Änderungen der Abhängigkeiten nachvollziehbar
- Maven legt Bibliotheken aller Projekte in einem zentralen Verzeichnis ab

- Deklarierte Abhängigkeiten prüfen und ggf. herunterladen
- Transitive Abhängigkeiten prüfen und ggf. herunterladen
- Klassenpfad für Compile- und Test-Lauf erstellen



Continuous Integration

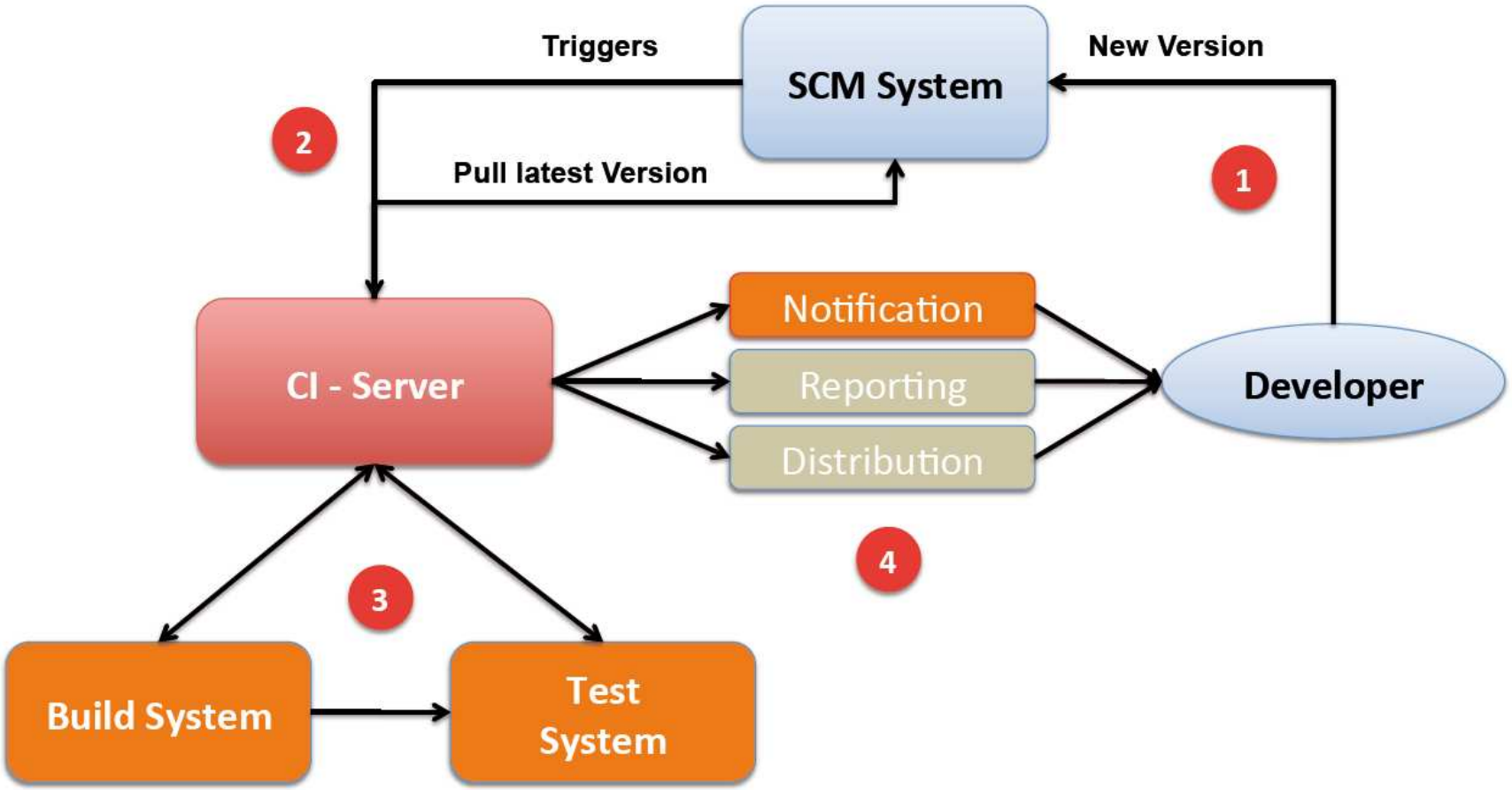
- Vorteile einer Build-Automatisierung
  - Projekt schnell mit Archetypen aufsetzen
  - Projekt ist portabel
  - Abhängigkeiten leichter darstellbar und auflösbar
  - Verbesserung der Projekt-Qualität durch Integration automatisierter Tests
  - Automatisiertes Reporting

- Einsatz im Continuous Integration Kontext
- Best-practices Aspekte
  - ein Sourcecode-Repository
  - Automatisierter Build
  - Automatisierte Testabläufe
  - Tägliche Commits der Entwickler
  - Schneller Build
  - Build auf einem neutralen Rechner
    - daily builds, snapshots…
  - Automatisiertes Deployment
  - Transparenz des Entwicklungsprozesses

Continuous Integration Werkzeuge

- Build-Automatisierungswerkzeuge wie Maven sind Entwicklerorientiert
- Continuous Integration Werkzeuge (Apache Continuum, Hudson, OpenCIT) unterstützen Server-basierte Integration und Ausführung von Tests
- Automatisierung beinhaltet
  - Ereignis oder Zeitgesteuerter Abruf
  - Verwendung von Build Werkzeugen
  - Ausführung von Tests
  - Erstellen von Berichten
  - Verschicken von Mitteilungen

Continuous Integration Arbeitsablauf



Zusammenfassung

- Sourcecode Management Systeme
  - Effiziente Zusammenarbeit in verteilter Software Entwicklung
- Build Management Systeme
  - Minimierung von Aufwänden durch hohen Grad an Automatisierung

References

- Vortrag Linus Torvald über Git
  - http://www.youtube.com/watch?v=4XpnKHJAok8
- http://git-scm.com/
- http://excess.org/article/2008/07/ogre-git-tutorial/
- http://subversion.tigris.org/
- http://maven.apache.org/
- http://opencit.openengsb.org/

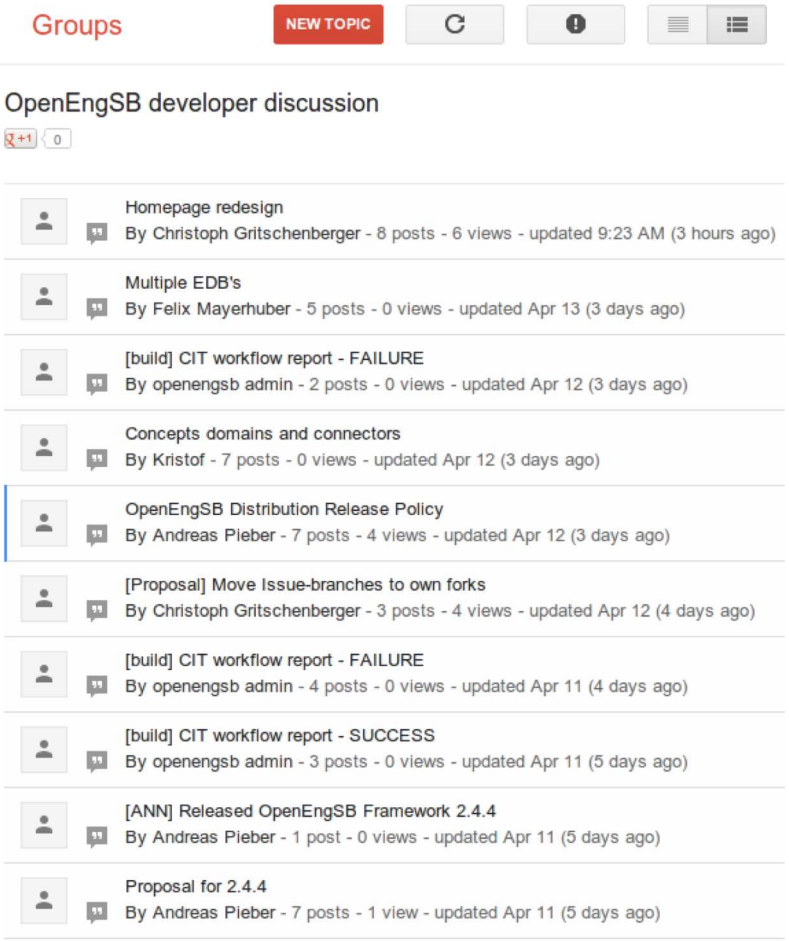
Kommunikation im Team

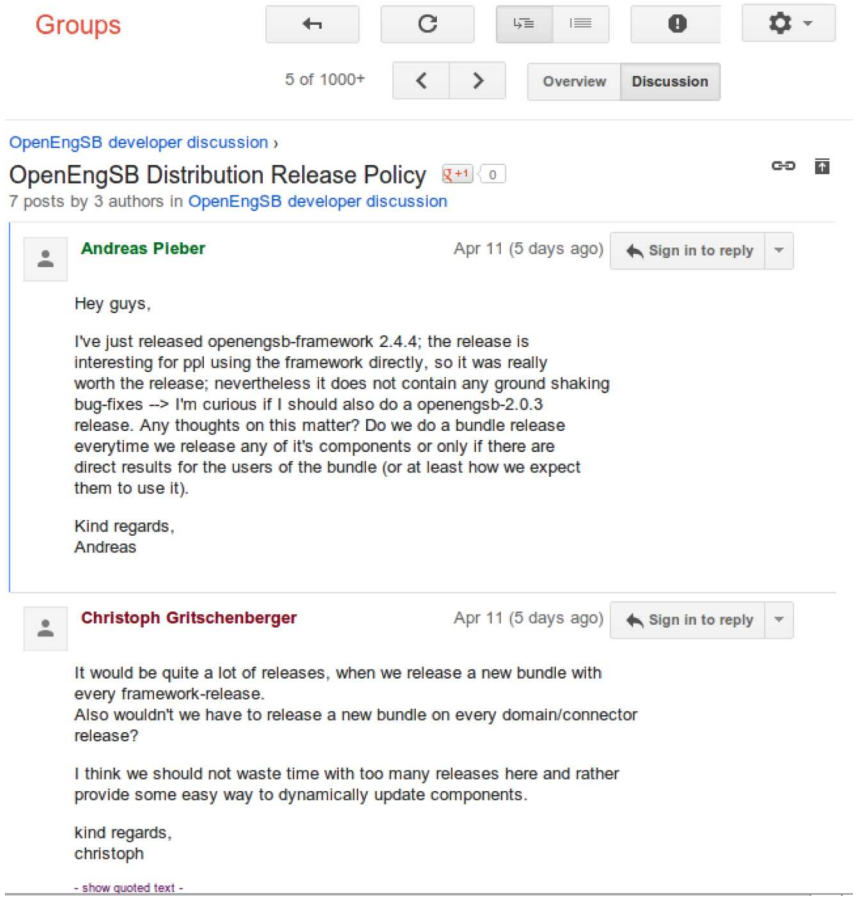
- Synchron
  - Face-to-Face
  - Telefon
  - Instant Messaging, VoIP (XMPP, Skype)
- Asynchron
  - Email
  - Wiki, CMS
  - Mailingliste/Forum
  - Issue-Tracker

Mailingliste/Forum

- Zentrale Verwaltung von Interessenten
- Diskussionen besser nachvollziehbar
- Mailinglisten für verschiedene Zwecke
  - developer
  - user
  - announce

Mailingliste Beispiel



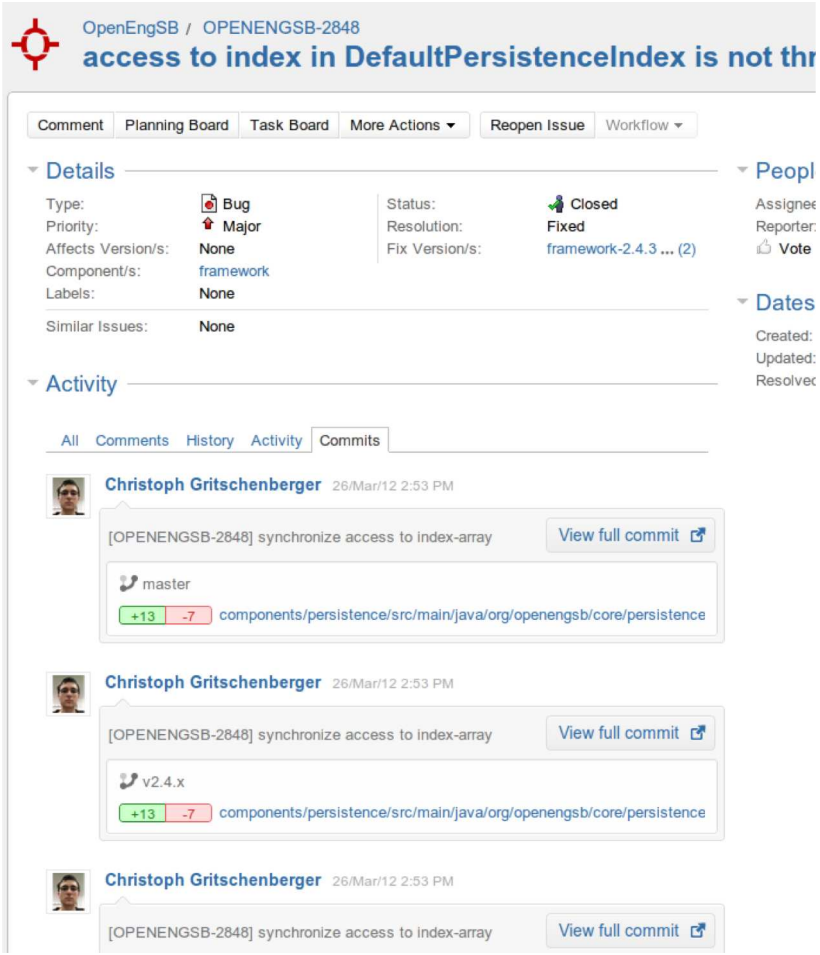


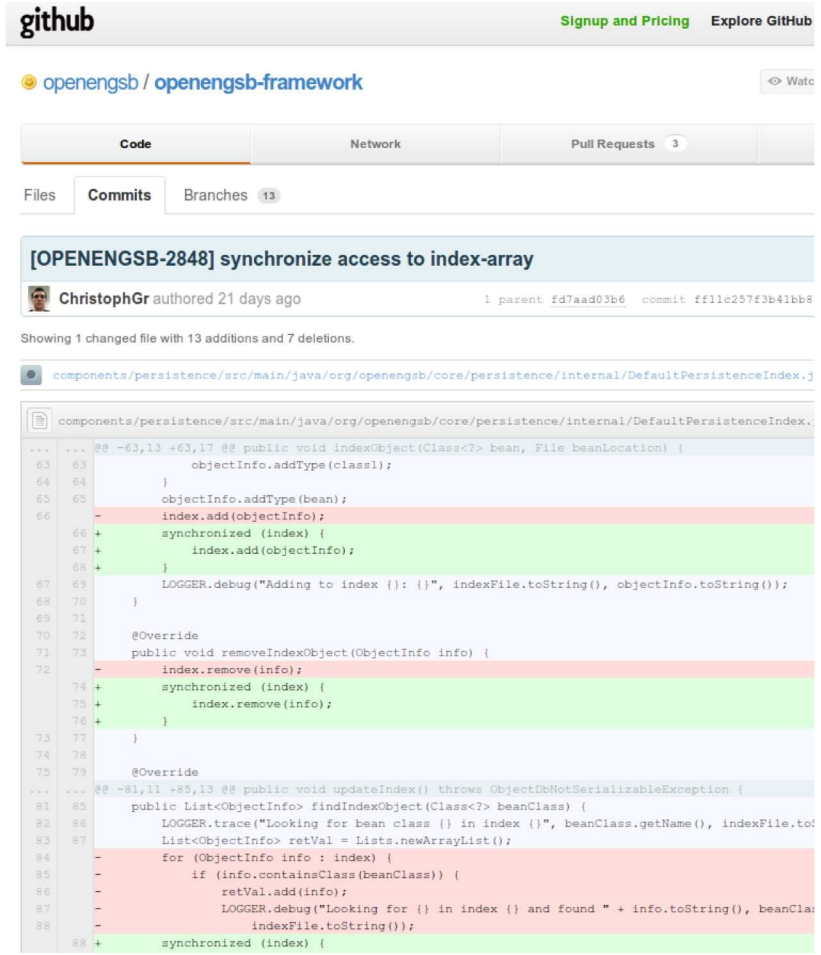
Issue-Tracker

… Bug Tracker, Project Tracker

- Erfassung von Fehlern und Änderungswünschen
- Verwaltung von Roadmaps
- Feature-Beschreibungen und Arbeitspakete

Integration von Issue-Tracker mit SCM





Beispiele für Issue-tracker

- Frei
  - Bugzilla
  - Trac
  - Redmine
- Kommerziell
  - Atlassian JIRA
  - Microsoft Team Foundation Server
- Eigenentwicklungen
  - Github
  - Google Code

Herausforderungen bei globaler Verteilung

- Große geographische Distanzen
  - Kein Face-2-Face
  - Austauschen von Handzeichnungen erschwert
- Verschiedene Arbeitszeiten
- Verschiedene Zeitzonen
- → Hauptsächlich asynchrone Kommunikation

Synchrone Kommunikation

- Chat
  - IRC
  - XMPP
- VoIP/Video-Chat
  - Skype
  - XMPP (Google Talk)
- Cloud Document Services (Google Docs, Office 365)
- VNC (Skype, Teamviewer)

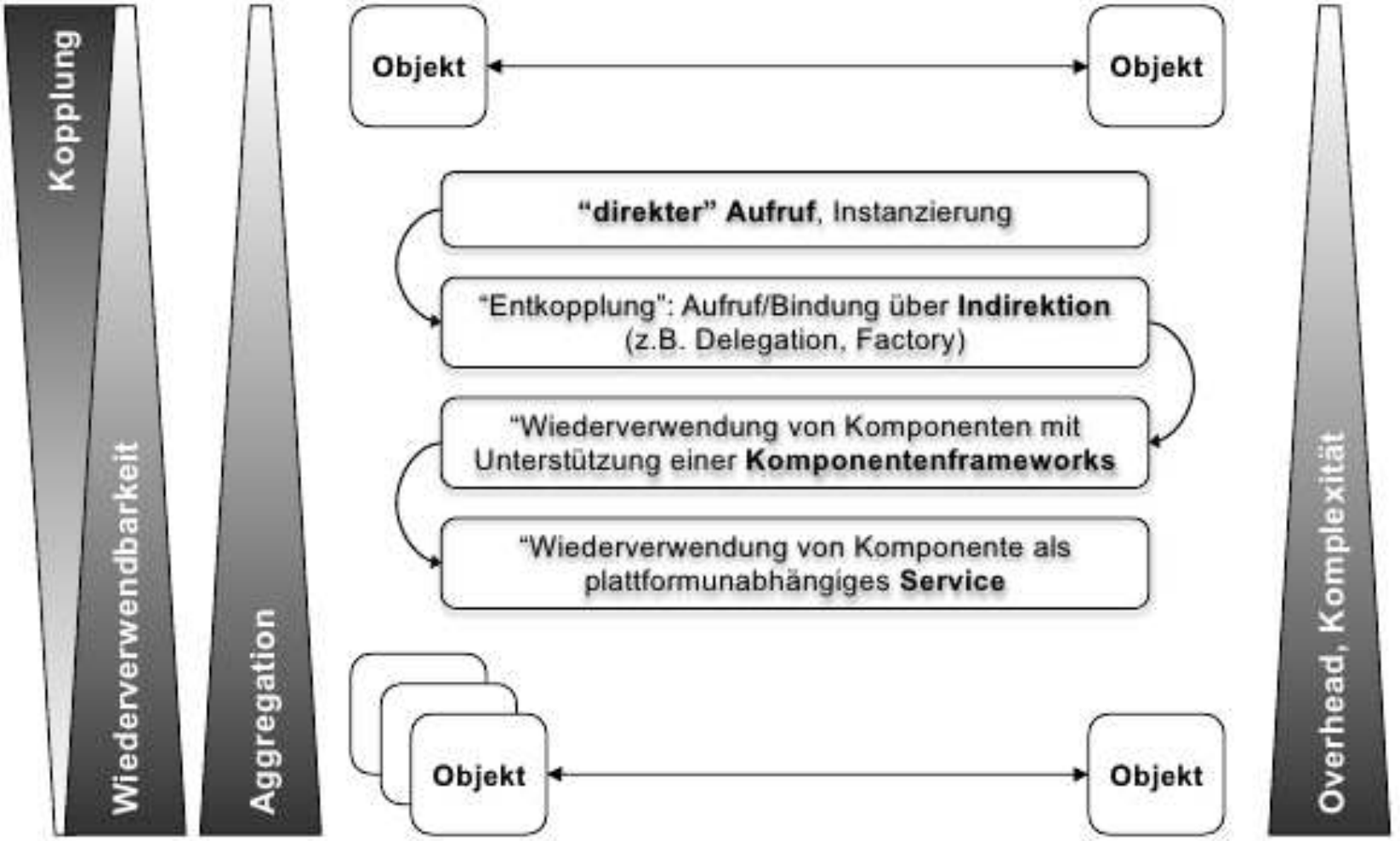
Kulturen

- Linux
  - Eigene Mailinglisten für Bereiche
  - Durchreichen von Patches durch Hierarchie
  - *„Benevolent dictator governance model“*
- Apache
  - Project Management Comittee (PMC)
  - Entscheidungen durch „Votes“ über Mailinglisten
  - *„Meritocratic governance model“*
- OPS4J (Open Participation for Java)
  - Öffentlich schreibbare Github-repositories
  - Diskussionen auf Mailinglisten
  - *„Open Contribution“*
  - *“Wiki brought to Coding”*

Zusammenfassung

- Synchrone vs. asynchrone Kommunikation
- Mailinglisten und Issue-Tracker
- Herausforderung bei globale Verteilung
- Verschiedene Projektmanagement-Varianten in OSS

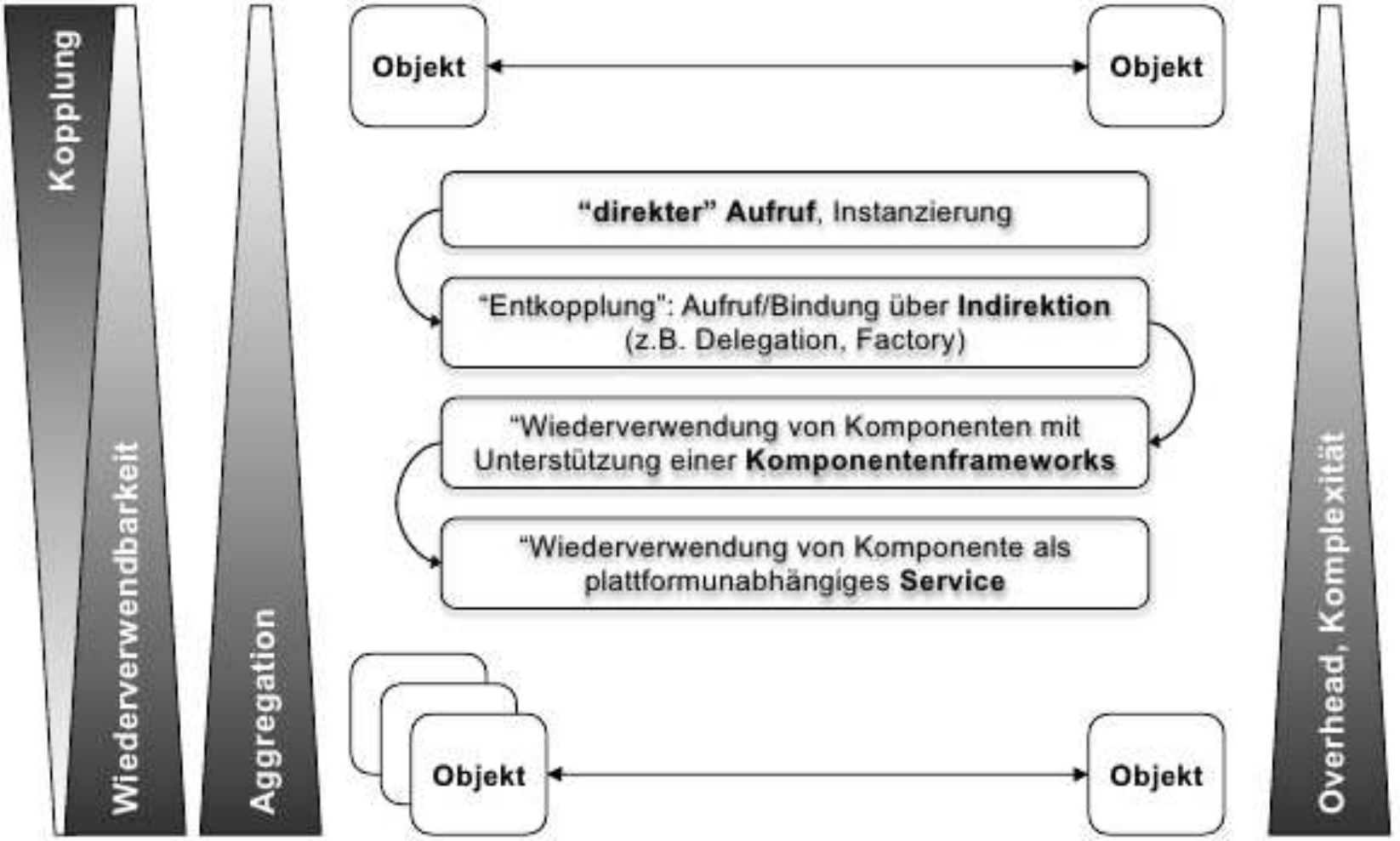
Komponentenorientierte Software-Entwicklung



Begriffe

- Komponente
  - klare, stabile Schnittstelle
  - höhere Granularität als eine Klasse
  - Wiederverwendbarkeit
  - locker verbunden
- Service
  - klar definierte Schnittstelle
  - plattformunabhängig
  - über ein Netzwerk angeboten
- Framework
  - Wiederverwendbarkeit
  - Rahmenbedingungen für Komponenten (z.B. Persistenz-Framework)

Komponentenorientierte Software-Entwicklung



Direkter Aufruf

```
public class Main {
    public static void main(String[] args) {
        Handler handler = new Handler();
        handler.notify();
    }
}
```

- leicht zu implementieren
- Binding beim Kompilieren
- Starke Kopplung

Interface

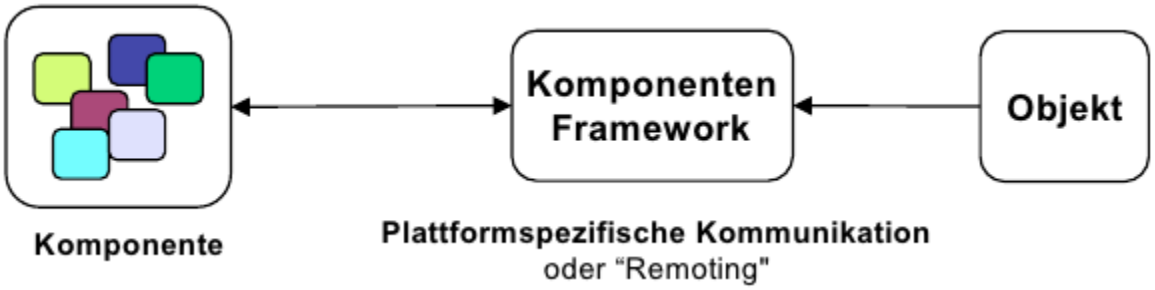
Interface

```
1 public interface TextImport {
2   public Document read (String filename);
3 }
```

```
1 class A {
2   TextImport ti = new HtmlImport();
3   public processDocument (String filename) {
4     Document doc = ti.read(filename);
5     ...
6   }
7 }
```

- Contract
- Einfacheres Austauschen der Implementierung
- Binding beim Kompilieren

Komponenten

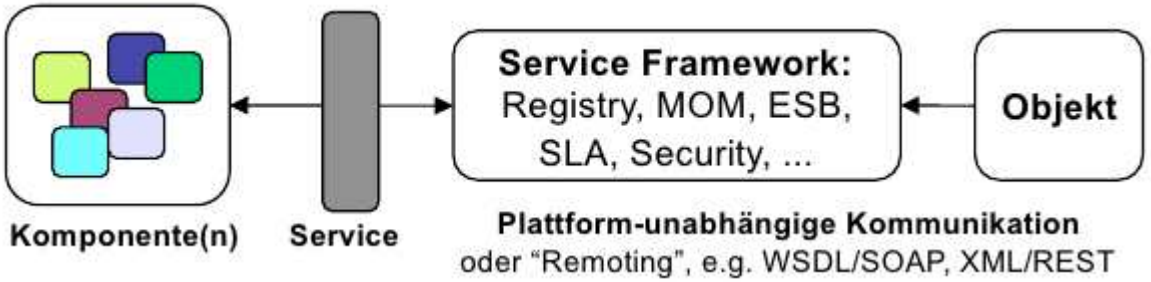


Komponenten

```
1  class KlasseA {
2    TextImport ti;
3    public KlasseA() {
4      // pseudocode:
5      ti = componentFramework.
6        getComponent ("htmlimport");
7    }
8    public processDocument (String filename) {
9      Document doc = ti.read(filename);
10     ...
11   }
12 }
```

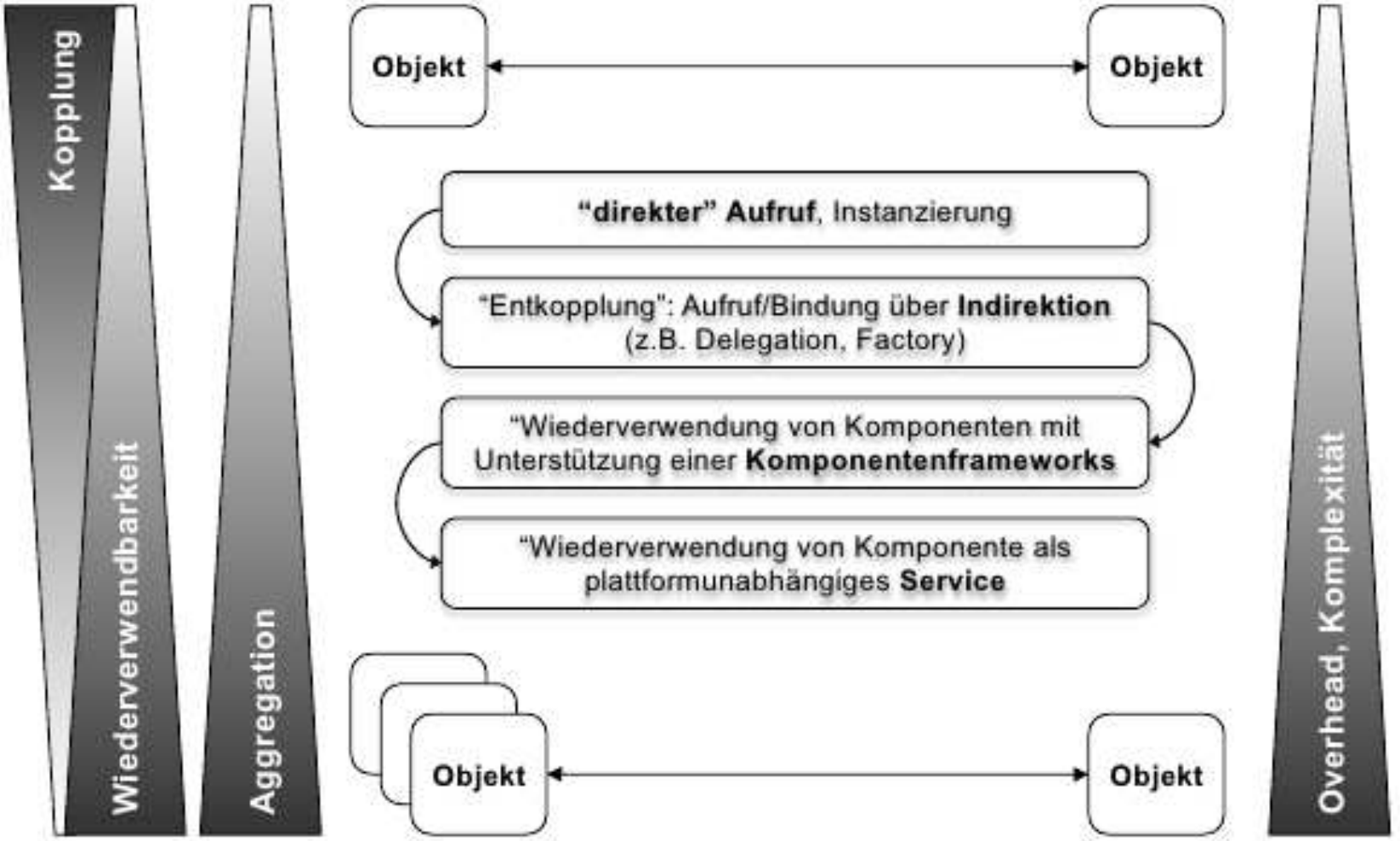
- Austauschen der Implementierung ohne Änderung im Sourcecode
- Verantwortlichkeit bei der Klasse selbst

Services



- Kommunikation über Systemgrenzen
- Webservices (WSDL und SOAP)

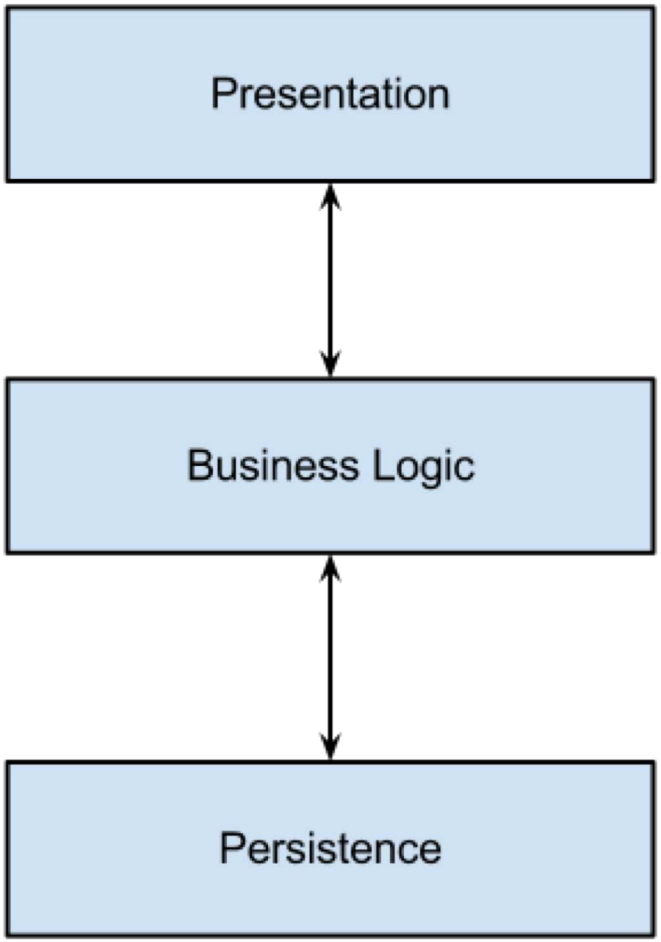
Komponentenorientierte Software-Entwicklung



Verbinden von Komponenten

- Software besteht aus mehreren Komponenten
- Architektur bestimmt Verbindungen und Zusammenspiel zwischen Komponenten
- Grad der Entkopplung
- Komplexität

Beispiel Anwendung



Beispiel Anwendung – Variante 1

```
public class MyPersistence {

    private MyBusinessLogic businessLogic = new MyBusinessLogic();

    // ...

}
public class MyPresentation {

    private MyBusinessLogic businessLogic = new MyBusinessLogic();

    // ...

}
public class MyBusinessLogic {

    private MyPersistence persistence = new MyPersistence("path/to/db");
    private MyPresentation presentation = new MyPresentation(300, 200);

    // ...

}
```

Beispiel Anwendung – Variante 2

```
public static void main(String[] args) {

    IPersistence persistence = new MyPersistence("path/to/db");

    IPresentation presentation = new MyPresentation(300, 200);

    IBusinessLogic businessLogic =
            new MyBusinessLogic(persistence, presentation);

    persistence.setBusinessLogic(businessLogic);
    presentation.setBusinessLogic(businessLogic);

}
```

Inversion of Control (IoC)

- Abhängigkeiten wird von einem Container verwaltet
- Komponenten wissen nichts darüber
- Abhängigkeiten werden in Komponenten injiziert
  → „Dependency Injection“ (DI)
- Frameworks mit Hollywood Prinzip:
  – *"Don't call us, we'll call you"*

Beispiel Anwendung – Variante 2

```
public static void main(String[] args) {

    IPersistence persistence = new MyPersistence("path/to/db");

    IPresentation presentation = new MyPresentation(300, 200);

    IBusinessLogic businessLogic =
            new MyBusinessLogic(persistence, presentation);

    persistence.setBusinessLogic(businessLogic);
    presentation.setBusinessLogic(businessLogic);

}
```

Beispiel Anwendung – Variante 3

```
public static void main(String[] args) {
    Factory factory = Factory.create();

    IPersistence myPersistence =
            (IPersistence) factory.getComp("persistence");

    IPresentation myPresentation =
            (IPresentation) factory.getComp("presentation");

    MyBusinessLogic myBusinessLogic =
            new MyBusinessLogic(myPersistence, myPresentation);

    myPersistence.setBusinessLogic(myBusinessLogic);
    myPresentation.setBusinessLogic(myBusinessLogic);
}
```

Beispiel Anwendung – Variante 4 (1/2)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean class="MyPersistence" id="persistence">
        <property name="path">path/to/db</property>
        <property name="businessLogic" ref="businessLogic" />
    </bean>

    <bean class="MyPresentation" id="presentation">
        <property name="width">300</property>
        <property name="height">200</property>
        <property name="businessLogic" ref="businessLogic" />
    </bean>

    <bean class="MyBusinessLogic" id="businessLogic">
        <property name="persistence" ref="persistence" />
        <property name="presentation" ref="presentation" />
    </bean>

</beans>
```

Beispiel Anwendung – Variante 4 (2/2)

```
public static void main(String[] args) {
    Resource res = new FileSystemResource("beans.xml");
    XmlBeanFactory factory = new XmlBeanFactory(res);

    IPresentation ui =
            factory.getBean("presentation", IPresentation.class);
    ui.show();
}
```

Vorteile von IoC

- Hohe Wiederverwendbarkeit durch zentrale Verwaltung
- Einfaches Austauschen einer Implementierung
- Verwalten von verschiedenen Konfigurationen
  - dev vs. deploy
- Automatisiertes Verdrahten
- Verteilung von Aufgaben

IoC Container Implementierungen

Java

- Spring-Framework
- Google Guice
- Pico-Container
- Apache Aries Blueprint

.NET

- Unity Framework
- Spring.NET

Constructor Injection

- Abhängigkeiten durch entsprechenden Konstruktor
- Beispiele: Pico, Guice, (Spring, Aries)

- Definition vollständiger Konfigurationen
- Komplexe Objekte können zu langen Konstruktoren führen
- Benennung der Argumente
- Problem mit einfachen Typen

Constructor Injection - Limitierungen

```
public class TestBean {

    String id;
    String internalLocation;
    String externalLocation;

    public TestBean(String id, String internalLocation) {
        this.id = id;
        this.internalLocation = internalLocation;
    }

    public TestBean(String id, String externalLocation) {
        this.id = id;
        this.externalLocation = externalLocation;
    }

}
```

Setter Injection

- Abhängigkeiten durch "setter"-Methode
- Beispiele: Spring, Aries, (Pico)

- Oft bevorzugte Variante
- Identifikation von Attributen mittels
  - Namenskonvention (z.B. setComponent())
  - Annotationen

Konfigurationsdatei

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean class="MyPersistence" id="persistence">
        <property name="path">path/to/db</property>
        <property name="businessLogic" ref="businessLogic" />
    </bean>

    <bean class="MyPresentation" id="presentation">
        <property name="width">300</property>
        <property name="height">200</property>
        <property name="businessLogic" ref="businessLogic" />
    </bean>

    <bean class="MyBusinessLogic" id="businessLogic">
        <property name="persistence" ref="persistence" />
        <property name="presentation" ref="presentation" />
    </bean>

</beans>
```

Konfiguration in Code

```java
@Configuration
public class AppConfig {

    @Bean
    public IPersistence persistence() {
        MyPersistence myPersistence = new MyPersistence("path/to/db");
        myPersistence.setBusinessLogic(businessLogic());
        return myPersistence;
    }

    @Bean
    public IPresentation presentation() {
        MyPresentation myPresentation = new MyPresentation(300, 200);
        myPresentation.setBusinessLogic(businessLogic());
        return myPresentation;
    }

    @Bean
    public IBusinessLogic businessLogic() {
        return new MyBusinessLogic(persistence(), presentation());
    }

}
```

Zusammenfassung

- Begriffe: Komponente und Service
- Verwalten von Komponenten
  - Direkt
  - Factory
  - Komponentenframework
- Dependency Injection/Inversion of Control
  - Constructor vs Setter Injection
  - Configuration File vs Code

References

- Best Practice Software-Engineering, Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen - Alexander Schatten et al
- http://martinfowler.com/articles/injection.html
- http://www.oss-watch.ac.uk/resources/benevolentdictatorgovernancemodel.xml
- http://www.oss-watch.ac.uk/resources/meritocraticGovernanceModel.xml

SEPM Vorlesung – Block 4
Software Engineering & Projektmanagement

Software Engineering Phasen

Dietmar Winkler

Vienna University of Technology
Institute of Information Systems Engineering
Research Group for Information and Software Engineering

dietmar.winkler@tuwien.ac.at
http://qse.ifs.tuwien.ac.at

Motivation

- Ziel ist die Herstellung eines qualitativ hochwertigen Softwareproduktes, z.B. durch
  - Minimale Anzahl an verbleibenden Fehlern.
  - Hohe Kundenzufriedenheit.
  - Anwendung von Best-Practice Methoden.

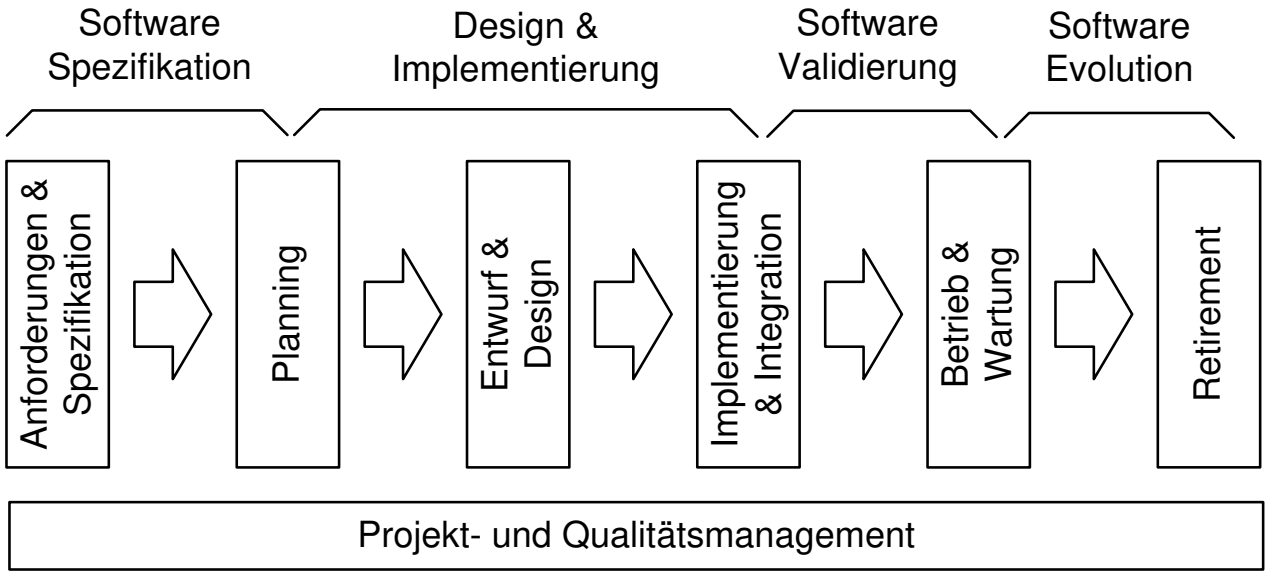
- Softwareprodukte umfassen ALLE Produkte im Rahmen der Softwareentwicklung z.B. Spezifikation, Code, Testfälle, Protokolle.

- Grundlegende Vorgehensweise
  - Systematische und strukturierte Vorgehensweise durch Software-Prozesse. (*wann* soll *welches* Produkt in *welchem Fertigstellungsgrad* verfügbar sein).
  - Konstruktive Methoden zur Herstellung von Softwareprodukten, z.B. für Spezifikationen, Testfälle, Source Code.
  - Analytische Methoden zur Überprüfung der Produktqualität, z.B. Reviews, Inspektionen und Tests.

- Absicherung einer durchgängig hohen Produktqualität während des gesamten Entwicklungsprozesses, z.B. durch integrierte QS-Methoden und Anwendung von Best-Practice Methoden.

Software Life-Cycle

- Ein Software-Prozess ist eine Abfolge von Schritten (Phasen) mit all seinen Aktivitäten, Beziehungen und Ressourcen.
- Einsatz von qualitätsverbessernden Maßnahmen in allen Phasen des Life-Cycles, d.h. von der ersten Idee über die Entwicklung bis zum kontrollierten Auslauf des Produktes.
- Der Software Life-Cycle beschreibt ein Basiskonzept für Software Engineering Prozesse und Vorgehensmodelle.



Software Life-Cycle

- Requirements (Anforderungen) zeigen die Wünsche des Kunden in Bezug auf das Softwareprodukt (*user/customer view*).
  Anforderungen müssen testbar sein und getestet werden!
- Eine Specification beschreibt das System aus technischer Sicht (*engineering view*).
- Planning: Erstellung des Projektplans bezüglich Zeit, Dauer, und Kosten (*project management*).
- Entwurf / Design: technische Lösung der Systemanforderungen (Komponenten, Packages, Datenbankdesign).
- Implementierung und Testen: Erzeugung des Softwareprodukts.
- Integration und Testen: Zusammenfügen und Test der einzelnen Komponenten auf Architektur- und Systemebene.
- Operation and Maintenance: Fehlerbehebung, Unterstützung, Erweiterungen des Softwareproduktes während des laufenden Betriebes.
- Retirement: Nach der Einsatzphase, d.h. am Ende des Produktlebenszyklus, muss das Softwareprodukt kontrolliert aus dem Betrieb genommen werden.

Table of Contents

- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
  - Warum sind Anforderungen wichtig?
  - Arten von Anforderungen
  - Prozess zur Anforderungserhebung
- Entwurf & Design
- Implementierung & Integration
- Qualitätssicherung und Software Testen
- Wartung, Evolution und Retirement

Über Anforderungen ..

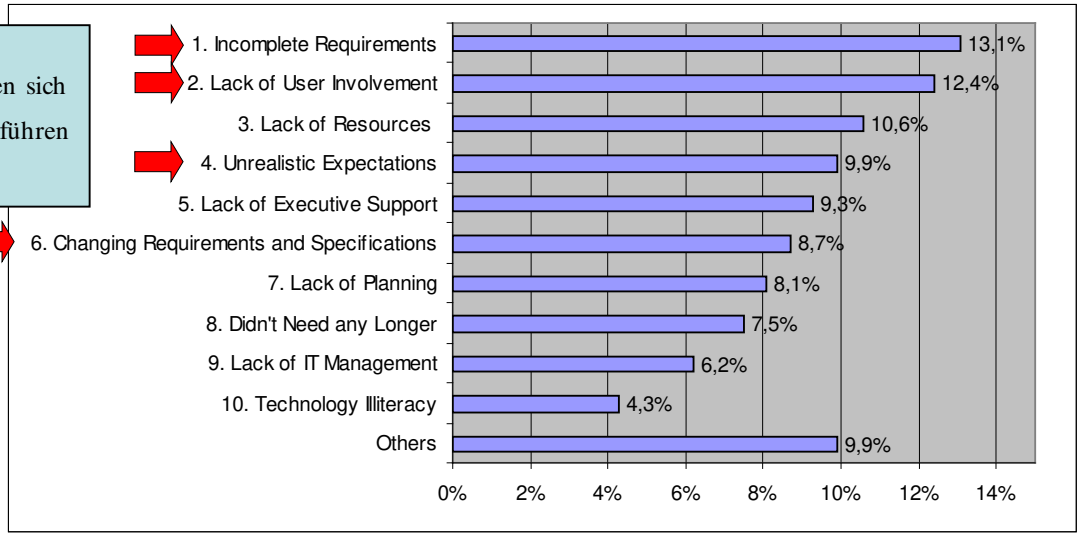
The hardest single part of building a system is deciding what do build. (B.W. Boehm, 1997)

- Durch Anforderungen wird ein gemeinsames Verständnis hergestellt, was das Produkt können soll.
- Sie repräsentieren die „reale Welt“ und drücken das gewünschte Verhalten aus Nutzersicht aus.
- Berücksichtigung unterschiedlicher Stakeholders (Kunde/Anwender, Entwickler, usw.)
  - Unterschiedliche Betrachtungsweisen des Projektes.
  - Unterschiedliche Begriffsbilder und Vokabular.
  - Die Projektbeteiligten müssen sich auf eine gemeinsame Sicht einigen.
- Typischerweise werden Anforderungen beschrieben bzw. grafisch dargestellt (z.B. Use-Case Modellierung aus der UML Familie).
- Anforderungen müssen testbar und nachvollziehbar sein!
  Unterstützung durch Requirements Management Tools, wie z.B. Doors, Rational Requisite Pro.

Warum Anforderungen wichtig sind ..

§ Gründe für Projektabbruch – Daten stammen aus einer Umfrage bei 365 Unternehmen (8.380 Anwendungen) [Chaos Report, 1994]:

4 der 6 Hauptgründe lassen sich auf Anforderungen zurückführen



§ Auswahl aus den Top-Ten Risiken für Projektfehlschläge [Boehm, 1991]

…

3) Developing wrong software functions.
4) Developing the wrong user interfaces.
5) Gold plating.
6) Continuing stream of requirement changes.

…

Um das richtige System zu erstellen, müssen wir wissen, was der Kunde will bzw. braucht!

Arten von Anforderungen

Funktionale Anforderungen

- Was bzw. welche Funktion soll umgesetzt werden?
- Wie soll sich das System in definierten Situationen verhalten?
- Datenformate

Nicht-funktionale Anforderungen (z.B. Qualitätskriterien)

- Welche sonstigen Anforderungen müssen umgesetzt werden, die nicht direkt auf die Funktionalität abzielen, sie aber beeinflussen, z.B.
  - Leistung und Performance (z.B. Optimierung des Informationsflusses).
  - Usability und menschliche Faktoren (z.B. Einfachheit der Bedienung, Training).
  - Sicherheit (z.B. Zugangskontrolle), Wartbarkeit (Trennung von Anwendung und Daten), Erweiterbarkeit.

Designbedingungen

- Worauf soll beim technischen Entwurf geachtet werden, z.B. Schnittstellen, Plattformen und Entwicklungsumgebung, Verteilte Entwicklung.

Prozessbedingungen

- Rahmenbedingungen im Softwareprojekt, z.B. Ressourcen / Dokumentation.

Stakeholder

- Abhängig von der Projektrolle müssen unterschiedliche Anforderungen berücksichtigt werden (siehe auch Vorlesungsblock 1):
  - Kunden bezahlen für ein Softwareprodukt.
    Anforderung: z.B. schnelle und kostengünstige Lieferung.
  - Anwender müssen mit dem System arbeiten.
    Anforderung: z.B.: Erfüllung von funktionalen und nicht-funktionalen Anforderungen (Usability, Einfachheit, Stabilität, usw.)
  - Entwickler erstellen das Softwareprodukt.
    Neuester Stand der Technik, “Vergolden” des Systems durch unnötige aber herausfordernde Funktionalitäten, Selbstverwirklichung.
  - etc.
- Hauptziel ist es, ein System zu entwickeln, das die Anforderungen möglichst aller Stakeholder berücksichtigt.
- Anforderungen umfassen dabei sowohl funktionale und nicht-funktionale Anforderungen als auch Design und Prozessvorgaben.

Prozess zur Anforderungserhebung

- Erhebung der Anforderungen aller relevanten Stakeholder ist zeitaufwendig und nicht trivial.
- Priorisierung von Anforderungen:
  - Hauptanforderungen (*must-be* Kritieren).
  - Gewünschte Anforderungen (*expected*, nicht entscheidend aber wichtig).
  - Optionale Anforderungen (*nice-to-have* Kriterien).

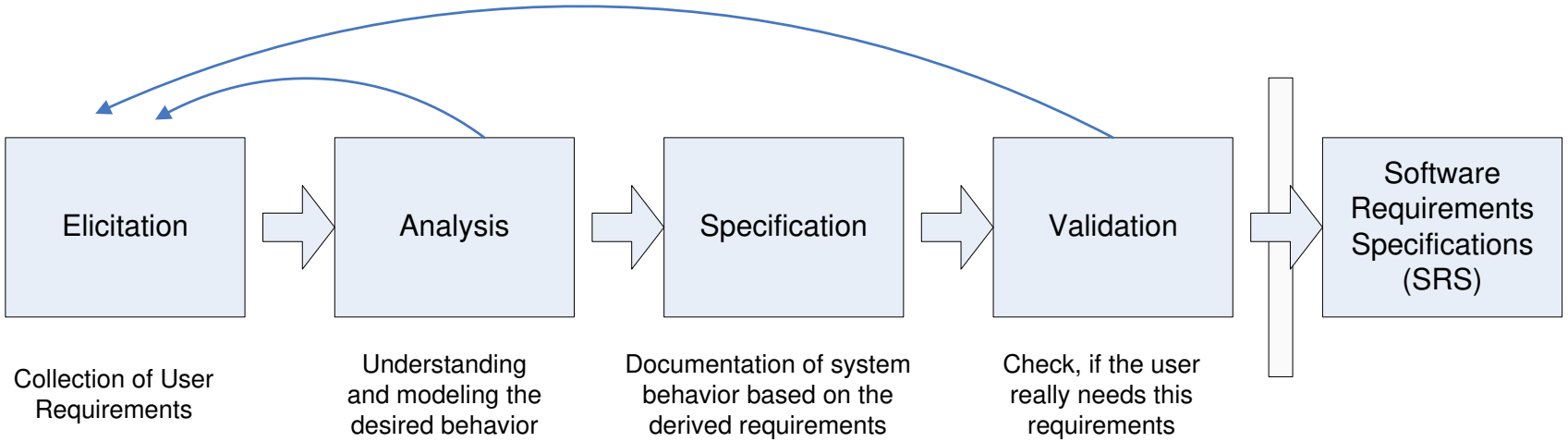


Table of Contents

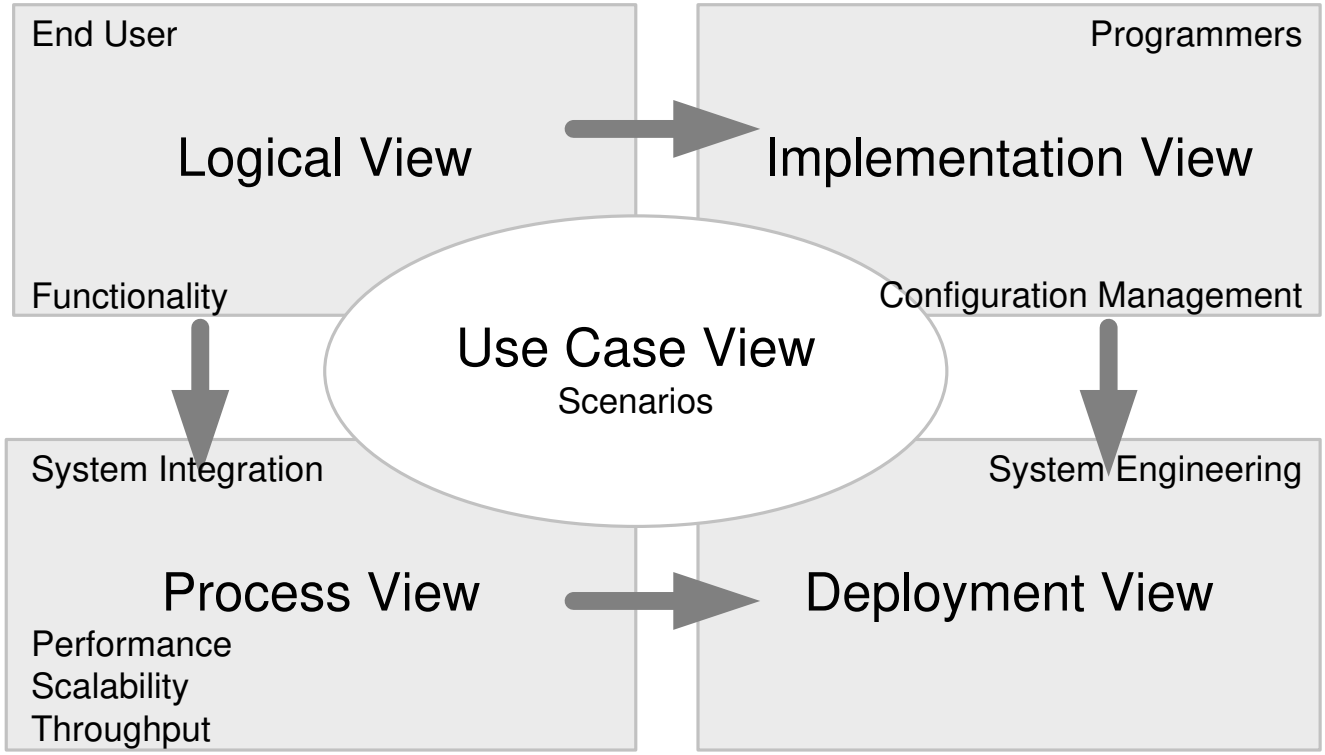
- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
- Entwurf & Design
  - 4+1 Model View
  - Ausgewählte Designprinzipien
- Implementierung & Integration
- Qualitätssicherung und Software Testen
- Wartung, Evolution und Retirement

Entwurf und Design

- IEEE 610.12-90 definiert *"software design"* folgendermaßen:
  - "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and
  - "the results of that process".
- Entwurf und Design beinhalten beispielsweise
  - Architekturdefinitionen & Evaluierungen (wie z.B. ATAM) zur systematischen Analyse unterschiedlicher Architekturvarianten.
  - Definition der Komponenten und Systemschnittstellen.
  - Domänen- und Datenbankmodelle.
  - User Interfaces
- Das Software Design beinhaltet technische Anwendungsfälle der Anforderungen, aller ihrer Subsysteme (Komponenten) und Detailinformation, wie die Lösung aus technischer Sicht aussehen soll (auch Datenstrukturen, Datenflüsse und Algorithmen).
- Sie ist in der Regel in der „Sprache der Techniker" geschrieben.
- Das Designdokument ist die Grundlage der Implementierung!

4+1 View Model of Architecture (1)

- Eine Sicht (*view*) beschreibt einen Teilaspekt der Architektur und des Designs und die spezifischen Eigenschaften eines Systems.
- Stammt aus der UML Familie.



[Kruchten, 2004]

4+1 View Model of Architecture (2)

Logical View:

- Funktionale Anforderungen.
- Fokus auf den End-User.
- Beispiele: Design Packages, Subsysteme, Klassen.
- UML 2: Klassendiagramme, State-Machines, Package Diagrams usw.

Process View:

- Nicht-funktionale Anforderungen
- Fokus auf Systemintegration.
- Beispiele: Laufzeitbedingungen, wie Concurrency, Lastverteilung, Fehlertoleranz.
- UML 2: Sequence, Activity Diagrams, Communication Diagrams.

Implementation View:

- Beschreibt statische Software Komponenten.
- Fokus auf Implementierung.
- Beispiele: Configuration Management, Source Code
- UML 2: Component Diagram der vorhandenen Softwareteile.

Deployment View:

- Ausführbare Applikationen (zur Laufzeit) unter Berücksichtigung der Plattform.
- Fokus auf System Engineers.
- Beispiele: Deployment, Installation, Performance.
- UML 2: Deployment Diagrams.

4+1 View Model of Architecture (3)

Use-Case View:

§ Als Ergänzung zu den bisherigen Views, bildet der Use-Case View den gemeinsamen Nenner, in dem die Anwendungsfälle und die Aktivitäten (als Szenarien) abbildet und in einen Zusammenhang gesetzt werden.

§ Fokus auf Systemanalyse sowie Entwurf und Design.

- Der *use-case view* bildet die Schnittstellenfunktion zwischen den anderen Sichten aus der Architektursicht …
- … und beinhaltet Schlüsselszenarien (*key scenarios*) der Applikation aus der Sicht der Geschäftsprozesse.

§ Fokus auf Implementierung und "Transition"

- Verifikation und Validierung der Anforderungen (Test) und der 4 Architektur-Views.

UML 2: Use Case Diagram + Beschreibung, Aktivitätsdiagramme.

Einige wichtige Design Prinzipien ..

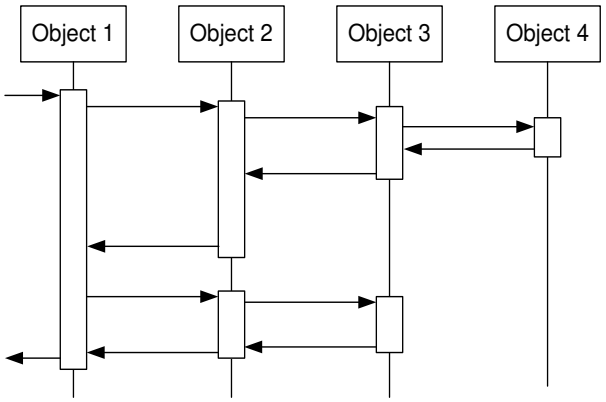
Design-Entscheidung: Coupling vs Cohesion

- Abhängigkeiten zwischen Komponenten bzw. innerhalb einer Komponente
  - **Coupling (Kopplung)** beschreibt die Abhängigkeit zwischen den einzelnen Komponenten (z.B. Anzahl der Methodenaufrufe). Eine hohe Kopplung bedeutet eine hohe Abhängigkeit zwischen unterschiedlichen Komponenten.
  - **Cohesion (Bindung)** ist ein Maß für den inneren Zusammenhalt einer Komponente. Falls sehr viel (auch ungenutzte) Funktionalität in eine Komponente gepackt wird, spricht man von einer niedrigen Kohäsion (zu komplexe Komponenten).

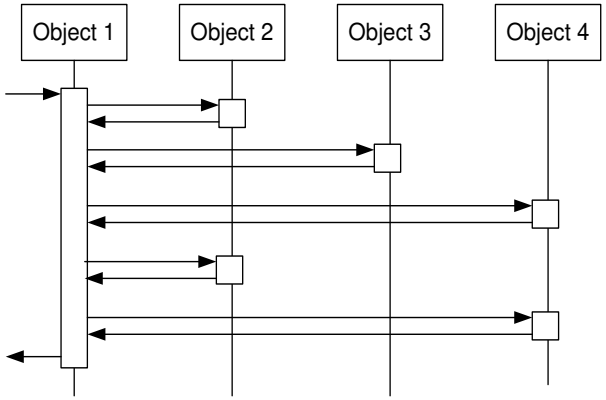
- **Ziel:** Gleichgewicht zwischen Kopplung und Kohäsion
  Erleichterung der späteren Wartung sowie Reduktion der Komplexität und Fehleranfälligkeit.
- **UML Diagramm(e):** Sequence- und Collaboration.
  Komplexe Diagramme bedeuten meistens eine hohe Kopplung und eine niedrige Kohäsion.

Design-Entscheidung: System Control – verteilt oder zentral

- Wer übernimmt die Kontrolle im System?
- **Stair**
  - Verteilte Kontrolle.
  - Schrittweise Ausführung von Funktionen, dadurch wechselt die Kontrolle.
  - Verbesserung der Wiederverwendbarkeit der Methoden z.B. durch Vererbung.
  - Die spätere Wartung wird erschwert.
- **Fork**
  - Ein zentrales Objekt kontrolliert den gesamten Use Case.
  - Wiederverwendung von Datenobjekten (ohne Business Logik) wird erleichtert.
  - Wartbarkeit wird verbessert, da nur ein Objekt geändert werden muss.



Sequence Chart: Stair



Sequence Chart: Fork

Weitere Design Principles

- **Abstraction**
  - Concept (z.B. Klasse) vs. Value (z.b. Objekt): Reduktion der Komplexität durch “ignorieren” von Details; z.B. Generalisierung in UML Klassendiagrammen.
- **Decomposition und Modularisierung**
  - Aufteilung großer Systeme in mehrere kleinere unabhängigere Teile (Komponentenorientierung).
  - Trennung von funktionalen Anforderungen.
  - Verbesserte Wiederverwendbarkeit.
- **Encapsulation / Information Hiding**
  - Packaging von Instanzvariablen und Methoden in eine Klasse um die Komplexität des Objekts und der Implementierung zu reduzieren.
  - z.B. private variables <> public properties.
- **Trennung von Interface und Implementierung**
  (z.B. durch Public und Private Interfaces)
  - Die Implementierung einer Komponenten kann geändert werden, solange die Interface unverändert sind.

Table of Contents

- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
- Entwurf & Design
- Implementierung & Integration
  - Interne und externe Standards
  - Integrationsstrategien
- Qualitätssicherung und Software Testen
- Wartung, Evolution und Retirement

Standardisierung im Projekt

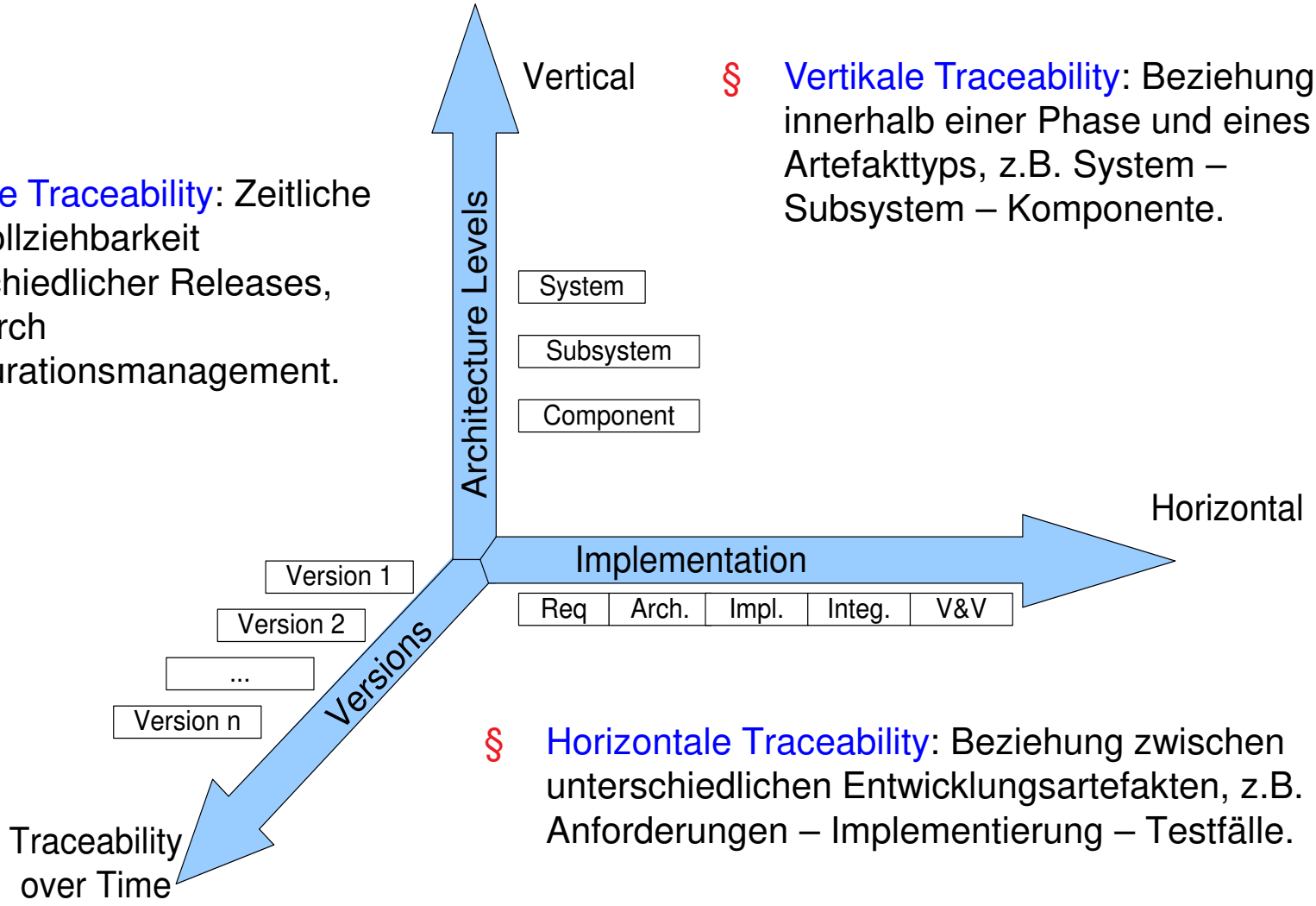
- "Software Engineering is a team-oriented contact sport" [Boehm, 2002].
- Die Zusammenarbeit von unterschiedlichen Stakeholdern (auch innerhalb von Entwicklerteams) erfordert auch im Bereich der Implementierung eine ausreichende und einheitliche Dokumentation.
- Standardisierung im Software Engineering (internal vs. external standardization).
- Ausgewählte Tipps zur effektiven und effizienten Code-Erstellung in Teams:
  - Namenskonventionen zur Verbesserung der Lesbarkeit des Codes, z.B. gleiche Notation von Variablen, Methoden und Klassen.
  - Formatierungsrichtlinien z.B. Einrückungen, gleicher Methoden und Komponentenaufbau erleichtern das Zurechtfinden in fremdem Code.
  - Versionierungen ermöglichen einen raschen Überblick aller (Teil-)Produkte innerhalb eines Projektes und die Verwendung der letztgültigen Versionen (z.B. CVS, Dokumenttagebücher)
  - Verwendung eines Headerblocks in jedem Komponente.
- Unternehmensstandards: Derartige Richtlinien werden meistens unternehmensweit festgelegt.

Nicht nur Doku … Traceability

- Traceability ist die Nach- oder Rückverfolgbarkeit einer Information durch ihren gesamten Entwicklungszyklus (z.B. bei sicherheitskritischen Anwendungen gefordert).
- Änderungsverfolgung ist die Fähigkeit, den Lebenszyklus einer Anforderung vom Ursprung der Anforderung über die verschiedenen Verfeinerungs- und Spezifikationsschritte bis hin zur Berücksichtigung der Anforderung in nachgelagerten Entwicklungsartefakten verfolgen zu können.
- Vorteile:
  - Nachvollziehbarkeit der Information bei Änderungen.
  - (Automatische) Benachrichtigung bei Änderungen.
- Typische Fragestellungen:
  - Woher kommt eine Anforderung und wo wurde sie umgesetzt?
  - Welche Artefakte sind von einer Änderung der Anforderung betroffen?
  - Welche Anforderungen sind von einer Änderung der Umsetzung betroffen?
- Informationen aus dem Headerblock können zur (automatisierten) Realisierung für Requirements Tracing eingesetzt werden (z.B. in IDEs).

[Chatrath et al, 2008]

Arten von Traceability



§ Zeitliche Traceability: Zeitliche Nachvollziehbarkeit unterschiedlicher Releases, z.B. durch Konfigurationsmanagement.

§ Vertikale Traceability: Beziehungen innerhalb einer Phase und eines Artefakttyps, z.B. System – Subsystem – Komponente.

§ Horizontale Traceability: Beziehung zwischen unterschiedlichen Entwicklungsartefakten, z.B. Anforderungen – Implementierung – Testfälle.

22. [Chatrath et al, 2008]

Übergreifende Standards

- Interne Standards werden typischerweise auf Unternehmensebene entwickelt und eingesetzt, um
  - Die Koordination von Teamaktivitäten zu verbessern.
  - Komplexität zu reduzieren.
  - Lesbarkeit und Verständnis der Dokumente und des Source Codes zu gewährleisten (z.B. für Reviews, Wartung oder zur Zusammenarbeit unterschiedlicher Standorte)

- Übergreifende Standards beeinflussen die Software Herstellung direkt:
  - Kommunikationsmethoden (z.B. Format und Inhalt der Nachrichten).
  - Programmiersprachen, z.B. Syntax in Java.
  - Plattformen, z.B. Interfaces zu Betriebssystemaufrufen.
  - Tools, z.B. Notationen wie UML.

- Solche Standards werden typischerweise von internationalen Organisationen, wie IEEE, ISO, OMG oder anderen veröffentlicht.

System Integration

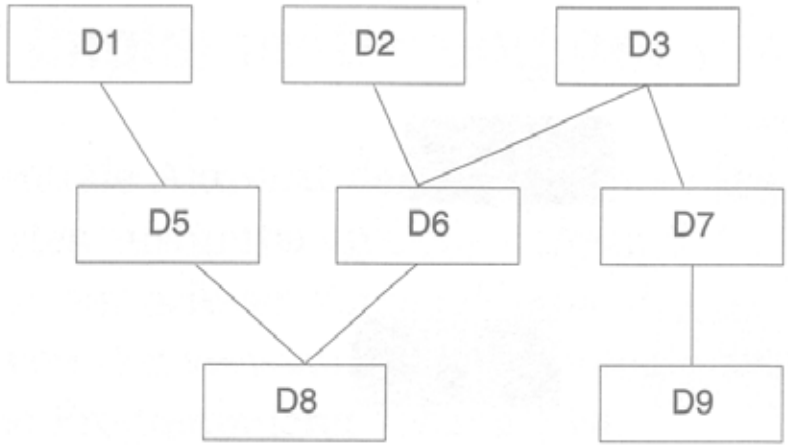
- Modularisierung erleichtert Lesbarkeit, die Verständlichkeit und Wartbarkeit von Softwarekomponenten.
- (Komplexe) Systeme umfassen meist viele unterschiedliche getestete Komponenten (z.B. nach Unit – oder Komponententests).
  Systemintegration bezeichnet die Integration unterschiedlicher Komponenten und Komponenten zu größeren (Sub-)Systemen.
- Integrationsstrategien sind vom Systemtyp und der Komplexität abhängig.
  - Big-Bang Integration:
    Gleichzeitige Integration aller Komponenten.
  - Top-Down / Bottom Up Integration:
    Integration ausgewählter Komponenten mit eingeschränkter Gesamt-Funktionalität (ohne Berücksichtigung der Business Cases).
  - Build Integration:
    Integration von ausgewählten Komponenten, um einzelne Anwendungsfälle (Use Cases) entsprechend dem Business Case umzusetzen
    (z.B. über priorisierte Liste von Use-Cases)

[Pfleeger et al., 2006]

Big-Bang Integration

Vorgehensweise:

§ Alle Komponenten werden gleichzeitig integriert (=„Big-Bang“).



Vorteile:

§ Keine zusätzlichen Testaufwände für Test-Stubs (um fehlende Funktionalität zu simulieren) alle Komponenten sind verfügbar.

Nachteile:

§ Fehler sind sehr schwer zu lokalisieren (z.B. Seiteneffekte).

§ Hohes Risiko bei der Integration.

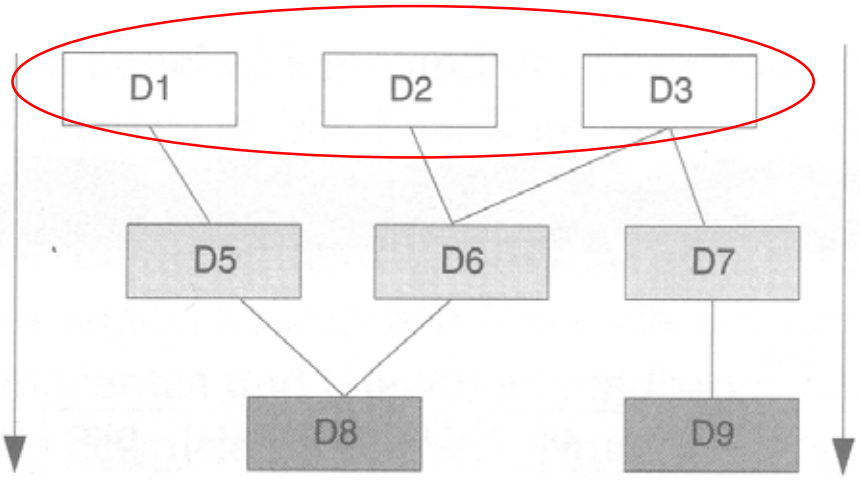
Anwendung:

§ Kleine und überschaubare Produkte.

Top-Down Integration

Vorgehensweise

§ Schrittweise Integration (z.B. D1, D2, D3), ausgehend von den Business Cases.



Vorteile:

§ Ausführbares Produkt Framework ist früh verfügbar.

§ "Prototypen" für Demo-Zwecke.

§ Framework für Testfälle.

Nachteile:

§ Zusätzlicher (hoher) Aufwand für Test stubs (um die fehlende Funktion zu simulieren).

§ Integration von Hardware erfolgt spät (zusätzliches Risiko).

Bottom Up Integration

Vorgehensweise:

§ Schrittweise Integration, beginnend bei der Hardware (z.B. D8/D9).

Vorteile:

§ Stabiles System (basierend auf Hardware Interfaces).

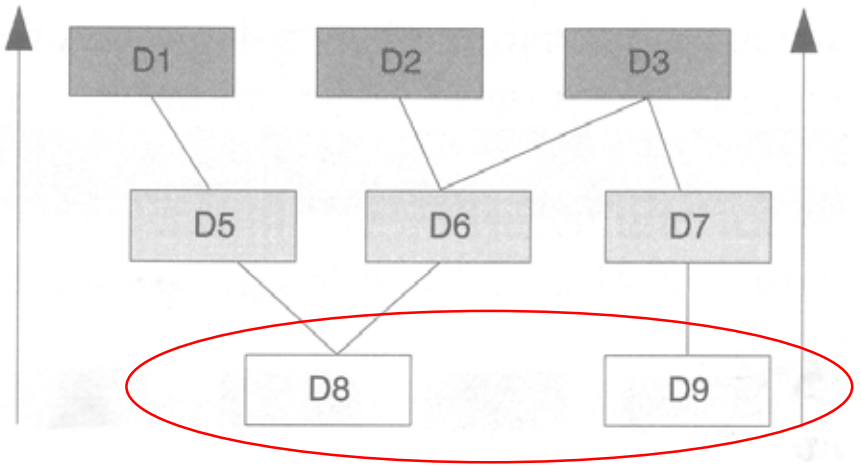
§ Schrittweise Integration Richtung Business Cases (Layers).

Nachteile:

§ Ausführbares Gesamtsystem ist spät verfügbar.

§ Zusätzlicher Aufwand für Prototypen.

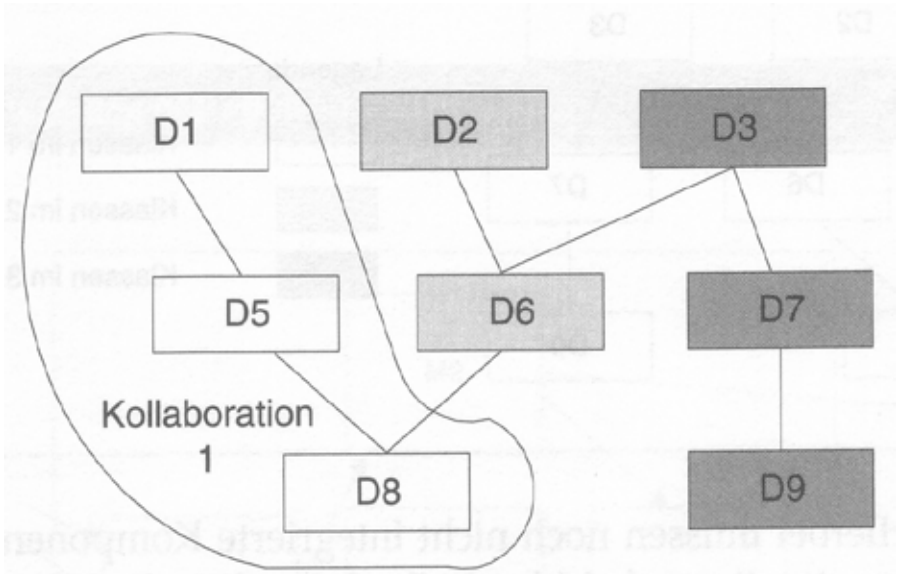
§ Zusätzlicher Aufwand für Test Drivers, um (lower-level) Komponenten zu testen.



Build Integration

Vorgehensweise:

- Schrittweise Integration entsprechend den Business Cases (über unter-schiedliche Layer hinweg), z.B. D1/D5/D8.
- Phasen-orientierte Integration.



Vorteile:

- Frühe Verfügbarkeit von funktionalen Anforderung (über alle Layer).
- Prototypen und Demo.
- Berücksichtigung priorisierter Anforderungen möglich.

Nachteile:

- Wiederverwendung von Komponenten kann schwierig sein.
- Regressions-Tests sind erforderlich.

Table of Contents

- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
- Entwurf & Design
- Implementierung & Integration
- Qualitätssicherung und Software Testen
  - Testen im Software Life-Cycle
  - Testtechniken / Strategien
  - Testfallerstellung
- Wartung, Evolution und Retirement

Qualitätssicherung und Testen

- Fehler im Softwaredesign haben oft immense Auswirkungen auf die Produktqualität, Projektdauer und Projektbudget und können bis zum Projektabbruch führen.
- Einsatz an Nacharbeit steigt je später ein Fehler gefunden wird.
  - Ziel ist es daher, Fehler möglichst frühzeitig zu erkennen und zu beheben.
  - Ein Fehler ist eine Abweichung der Lösung (z.B. einer Komponente) von der Spezifikation bzw. der erwarteten Eigenschaft.

Verifikation vs. Validierung:

- Verifikation:
  Spezifikation vs. Umsetzung („Wurde das Produkt richtig entwickelt?“)
  Beispiel: Komponententests (Prüfung gegen die technische Spezifikation)
- Validierung:
  Erwartung des Kunden vs. Umsetzung („Wurde das richtige Produkt entwickelt?“)
  Beispiel: Akzeptanz- und Abnahmetests (Prüfung gegen Anforderungen).

Testen im Life-Cycle Prozess

- Testen zielt darauf ab, Fehler zu finden (fehlende, falsche oder inkonsistente Informationen) und nicht der Nachweis, dass das Produkt funktioniert.
- Die zentrale Fragestellungen eines Testers ist [Kruchten, 2004]
  - Wie kann ein Softwareprodukt fehlschlagen?
  - Mit welchen Szenarien kann ich das Produkt in einen instabilen, nicht mehr vorhersehbaren Zustand bringen?
- Testen ist eine Aufgabe im Rahmen des Qualitätsmanagement und begleitet das Projekt laufend:
  - Testen der Funktionalität von Prototypen.
  - Testen von nicht-funktionalen Anforderungen wie Stabilität und Performance im Hinblick auf die Architektur.
  - Testen von nicht-funktionalen Anforderungen, z.B. Usability.
  - Akzeptanztest des fertigen Produktes für den Einsatz beim Kunden.
- Testen ist keine zusätzliche Aktivität am Ende des Softwareprojekts sondern begleitet das Projekt laufend!

Grundlegende Testtechniken

- Unit Test:
  Fokus auf Komponenten und Prüfung auf Übereinstimmung zwischen der Umsetzung der Komponente und der technischer Spezifikation.

- Integrationstest:
  Fokus auf Interfaces und Verbindungen zwischen Komponenten innerhalb des (Sub)Systems.

- Systemtest:
  Fokus auf Übereinstimmung zwischen funktionalen und technischen Bedingungen des Gesamtsystems (nahe an der Zielplattform).

- Regressionstest:
  Testen von geänderten Komponenten. Dadurch sollen Fehler, die durch Änderungen, z.B. auch durch Fehlerkorrektur, entstanden sind, verhindert werden.

- Akzeptanztest:
  Übereinstimmung der festgelegten Anforderungen in der Zielumgebung des Kunden.

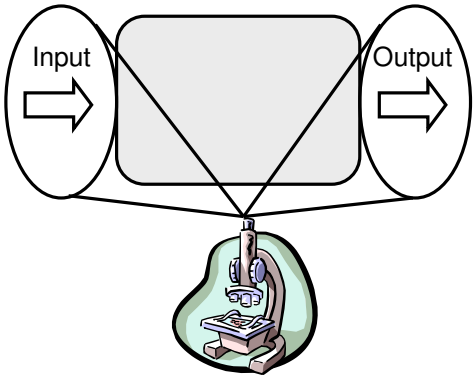
- Installationstest:
  Identifizieren von Fehlern während der Installation.

Teststrategien

"Testing is a quality assurance activity in order to find defects" [Myers, 1979].

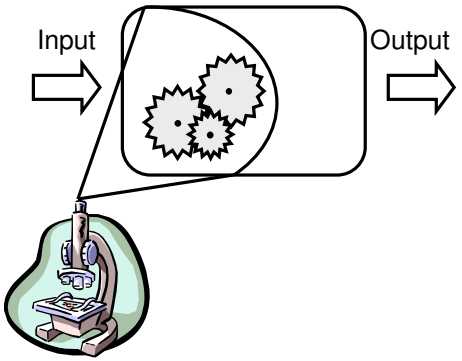
Black Box Tests

- Anforderungen / Spezifikation als Grundlage.
- Unabhängig von der Realisierung der Module.
- Data-driven (Input/Output).
- Anforderungsüberdeckung.
- Äquivalenzklassenbildung.
- Keine genaue Fehlerortung möglich.



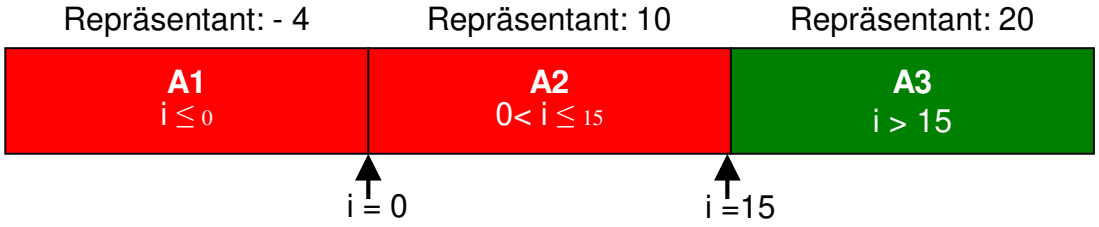
White Box Tests

- Software Code als Grundlage.
- Wissen über den internen Aufbau notwendig.
- Logic-driven.
- Kontrollflussüberdeckung.
- Äquivalenzklassen von internen Verzweigungen und Schleifen.
- Ermöglicht Fehlererkennung und –lokalisierung.



Ausgewählte Techniken

§ Äquivalenzklassen
  – Gleiches Verhalten aus einer Menge von Eingabedaten mit demselben Ergebnis Auswahl eines repräsentativen Wertes zur Reduktion der Testfälle.
  – Anforderung: i>15 3 Äquivalenzklassen



  – Testfälle müssen gültige (Normalfall, Sonderfall) UND ungültige Eingabewerte (Fehlerfall) berücksichtigen.

§ Grenzwertanalyse
  – Spezialfall einer Äquivalenzklasse.
  – Geeignete Auswahl der Repräsentanten in der Nähe der Datengrenzen.

Testfalldokumentation

§ Warum müssen Testfälle / Testergebnisse protokolliert werden?

- Wichtige Information für Entwickler (nicht nur bei aufgetretenen Fehlern).
- Wiederholbarkeit von Testfällen.
- Berichterstattung, Einschätzung der Produktqualität.
- Testfälle als Kommunikationswerkzeug.

Beispiel:

```
If (i > 15) {
    do something;
}
```

No	Type*	Pre-Condition	Test Case Description	Equivalence Classes	Expected Results	Actual Results
1	FF	i=-1	Invalid number, drop error message	AA	Drop Error message, no further action possible.	Error message
2	SF	i=15	Invalid number drop error message	AB	Drop Error message, no further action possible.	Error message
3	NF	i=20	Valid number “do something”	AC	Something Done	Error message
..						

* NF/NC: Normalfall / normal case; SF/SC: Spezialfall / special case; FF/FC: Fehlerfall / faulty case

Table of Contents

- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
- Entwurf & Design
- Implementierung & Integration
- Qualitätssicherung und Software Testen
- Wartung, Evolution und Retirement
  - Wartungskategorien
  - Wartungsaufwand
  - Retirement

Betrieb und Wartung

- "Software Maintenance is the modification of a software product after delivery
  - to correct faults,
  - to improve performance or other attributes,
  - or to adapt the product to a modified environment."
    [Definition acc. to IEEE 1219]

- Unterschiedliche Sichten auf die Wartung
  - Activity-View:
    Änderung des Software Systems nach Auslieferung (Delivery) und Inbetriebnahme (Deployment bzw. Product Launch).
  - Process-View:
    Schritte zur Durchführung einer Wartungsaufgabe.
  - Phase-Oriented-View:
    Die Wartungsphase beginnt mit der Auslieferung und Inbetriebnahme und Ende mit "Stilllegung" des Softwareproduktes.

Wartungskategorien

Reaktive Wartung:

§ Korrektive Wartung (Corrective):
Bug- und Fehlerkorrektur (patches, workarounds, updates).

§ Adaptive Wartungsaktivitäten (Adaptive):
Berücksichtigung geänderter externer Anforderungen (Hardware, Softwareänderungen).

Pro-Aktive Wartung:

§ Produktpflege und Verbesserung (Perfective):
Produktverbesserung (Erweiterungen, Verbesserung der Effizienz).

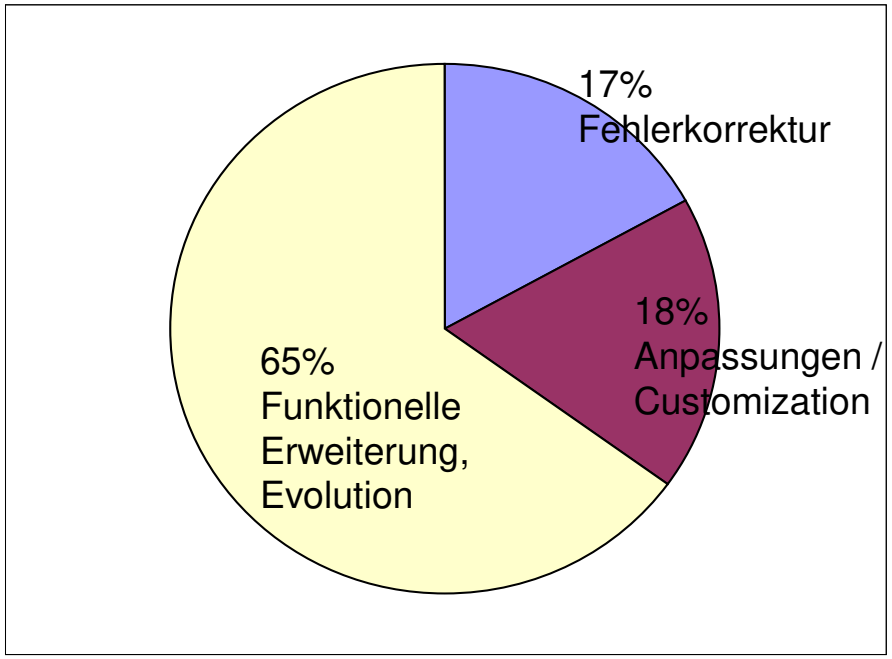
§ Vorbeugende Wartung (Preventive):
Verbesserung im Hinblick auf zukünftige Wartung (z.B. Ergänzung der Dokumentation).

	Correction	Enhancement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

[Van Vliet, 2000]

Wartungsaufwand

- Wartung verbraucht einen großen Teil der finanziellen Ressourcen.
- Mehr als 80 % der Wartungsarbeiten werden für nicht-korrektive Tätigkeiten benutzt.
- Wartungskosten hängen ab von
  - Applikationstyp.
  - Neuheit des Produktes.
  - Verfügbarkeit von Personal.
  - Hardwarecharakteristika.
  - Qualität des Softwaredesigns, Konstruktion, Dokumentation und Testen



Kosten von Software Wartung [Sommerville, 2007]

Wartungsprozesse beinhalten dieselben Phasen wie der Life-Cycle Prozess; einen speziellen Stellenwert nehmen die Anforderungsänderungen ein.

Techniken zur Wartung

- Herstellen des Produktverständnisses
  - Das Verständnis “fremder” Codestücke kann - speziell bei mangelnden Aufzeichnungen – sehr zeitaufwändig sein.
  - Key Tools: Code Browsers.
  - Produktanforderungen: Klare und präzise Dokumentation.

- Reengineering
  - Überprüfung und Überarbeitung des Software Codes.
  - Gravierende und teuere Form der Änderung.

- Reverse Engineering
  - Analyse der Software im Hinblick auf Komponenten und deren Zusammenhänge.
  - Erstellung von Modellen (basierend auf dem Code) auf einem höheren Abstraktionsniveau.
  - Z.B. Erstellung von UML-Modellen aus C# Code.

Phase Retirement

- Am Ende der Betriebsphase (Betrieb und Wartung) schließt der Life-Cycle Prozess mit der Stilllegung des Softwareproduktes ab.
- Kontrolliertes Außer-Betrieb-Setzen des Produktes bzw. störungsfreier Übergang zu einem Nachfolgeprodukt.

- Gründe für die Stilllegung einer Softwarelösung:
  - Zahlreiche Änderungen während der Wartungsphase können ein komplettes Redesign des Produktes verursachen.
  - Laufzeitfehler durch Nebeneffekte (kleine Änderungen im Programmcode können große Auswirkungen im Programmablauf haben).
  - Inkonsistenzen durch kurzfristige Änderungen ohne Aktualisierung der dazugehörigen Dokumentation.
  - Hardwareänderungen können ebenso ein komplettes Redesign oder Neuprogrammierung verursachen.

Zusammenfassung

- Der Life-Cycle Prozess umfasst sämtliche Schritte der Softwareentwicklung von der Konzeptphase bis zur Stilllegung des Softwareprodukts.
- Die Anforderungserhebung ist kritisch für den Erfolg eines Softwareproduktes. Änderungen oder Fehler wirken sich gravierend in späteren Phasen der Entwicklung aus.
- Entwurf und Design sind die Basis für die eigentliche Implementierung und legen den Aufbau des Systems fest. Designprinzipien sind zu beachten.
- Die Implementierung umfasst die Umsetzung der Anforderungen und des Entwurfs. Standards und Dokumentation unterstützen Softwareentwicklungsteams nicht nur bei der Erstellung sondern auch bei der Weiterentwicklung und Wartung.
- Qualitätssicherung und Testen begleiten ein Software Projekt während des gesamten Lebenszyklus. Sie dürfen nicht als Add-on sondern als integraler Bestandteil eines Projektes gesehen werden.
- Ein Grossteil der Wartung umfasst Erweiterungen des Produktes, aber auch Fehlerkorrektur und Anpassungen an geänderte Systemgegebenheiten.
- Die Retirement-Phase schließt den Software Life-Cycle ab, in dem das Softwareprodukt kontrolliert außer Betrieb genommen bzw. ersetzt wird.

Literaturreferenzen

- Biffl Stefan, Winkler Dietmar, Frast Denis: „Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung“, Skriptum zur Lehrveranstaltung, 2004. http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm
- Boehm B.: Software Risk Management: Principles and Practices, IEEE Software, 1991.
- Boehm B.: Software Engineering is a Value-Based Contact Sport, IEEE Software, 2002.
- Chatrath G., Dussa-Zieger K., Wentzel P-R.: Traceability – Anspruch und Realität, Proc. of SEE Conference, Bern, Switzerland, 2008.
- Kruchten P.: The Rational Unified Process: An Introduction, Addison-Wesley Longman, 2004.
- Myers G.J.: The Art of Software Testing, 1979.
- Pfleeger, Shari Lawrence; Atlee, Joanne M.: Software Engineering – Theory and Practice, Third Edition 2006, Pearson Education.
- Schatten A., Biffl S., Demolsky M., Gostischa-Franta E., Östreicher T., Winkler W.: „Best Practice Software Engineering. Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen“, Spektrum Akademischer Verlag, 2010, 978-3827424860.
- Software Engineering – Best practices: http://best-practice-software-engineering.blogspot.com/
- Software Engineering Body of Knowledge, http://www.swebok.org, 2004.
- Sommerville, Ian: „Software Engineering“, 8th Edition, Addison Wesley, 2007.
- Standish Group: Chaos Report, 1994.
- Van Vliet, Hans: Software Engineering – Principles and Practice, 2nd Edition, 2000.

Software Engineering & PM Vorlesung

„Modellierung von Anwendungsszenarien”

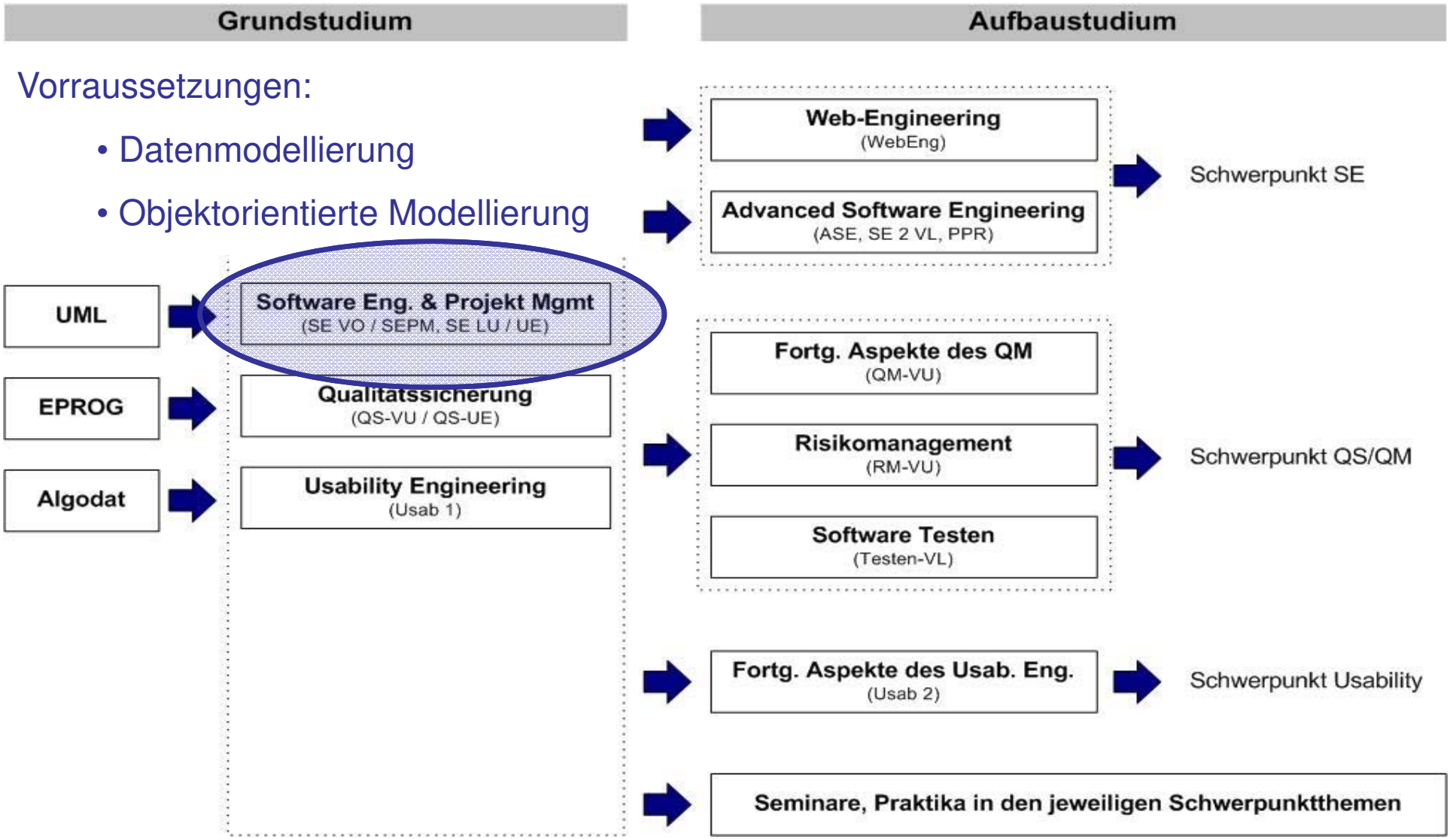
Institut für Information Systems Engineering

Stefan Biffl

Inhalt:

- Grundlagen der Modellierung
- Bedarf an Modellierung in SEPM
- SEPM PR: Kernmodelle und deren Anwendung
  - Datenmodell (Relationenmodell in UML)
  - Daten- und Kontrollfluss eines Teilsystems (UML, IDEFØ)
- SEPM VO: Prüfungsbeispiel „Restaurant“

Software Engineering im Studium



Überblick: Modellierung von Anwendungszenarien

1. "Modellierung in SEPM-Projekten" (20')

- Grundlagen, Begriffe
- Typische Modelle und deren Verwendung (UML 2.1, IDEFØ)
- Ausblick: Verifikation und Validierung, Test-Driven Development

2. "Datenmodellierung" (Struktur) (50')

- Entity Relationship Diagramm (UML 2.1 Notation)
- Anwendung von Datenmodellen in der Verifikation und Validierung
- Anwendungsbeispiel "Restaurant"

3. "Objektorientierte Modellierung" (Verhalten) (50')

- Aktivitätsdiagramm: Daten- und Kontrollfluss (UML 2.1, IDEFØ)
- Anwendungsbeispiel "Restaurant"
- Zustandsdiagramm (State Chart)
- *Anwendung der Modelle in der Verifikation und Validierung*

Modellierung in SEPM-Projekten

- Grundlagen, Begriffe
  - Konzeptionelle Modellierung
    - Abstrakte Repräsentation
    - Zeigt Struktur und Verhalten des Problembereichs sowie des zu entwickelnden Softwaresystems
  - Modelle, Diagramme und Werkzeuge
    - UML Modell vs. UML Diagramm
    - UML: statische und dynamische Modelle
    - IDEFØ: Daten- und Kontrollfluss

- Modellierung in der Softwareentwicklung
  - Verfeinerung von Modellen (IDEFØ, UML)

- *Ausblick: Test-Driven Development, Verifikation und Validierung*
  - Modelle als Basis für die Implementierung und QS

Motivation - Ziel

- Software-Entwicklung erfordert die Herstellung konsistenter Sichten auf Anforderungen und Entwurf eines Systems.
- Modelle helfen den Überblick zu bekommen und zu behalten.
  - Grundlage für effektives und effizientes Arbeiten im Team.
  - Gemeinsame Notation mit konsistenter Bedeutung.
- Prinzipien der Abstraktion; Information Hiding und Vererbung.
- Herausforderung: Unterschiedliche Modelle flexibel und konsistent zu verwenden.
  - Systemstruktur: Subsysteme, Komponenten, Schnittstellen.
  - Verhalten von Komponenten; Interaktion zwischen Komponenten.
- Anforderungen -> Modelle -> Daten/Komponenten/Testspezifikationen.

Aus: OO Modellierung, BIG, 2006

Modellierungssprachen

Eine Brille zur Betrachtung des Problembereichs

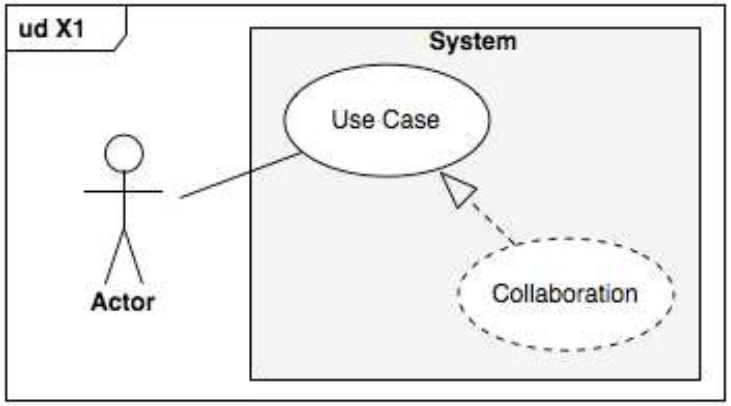
- beeinflusst die Wahrnehmung
- lässt Wesentliches sichtbar werden (ausser sie passt nicht – dann macht sie blind)

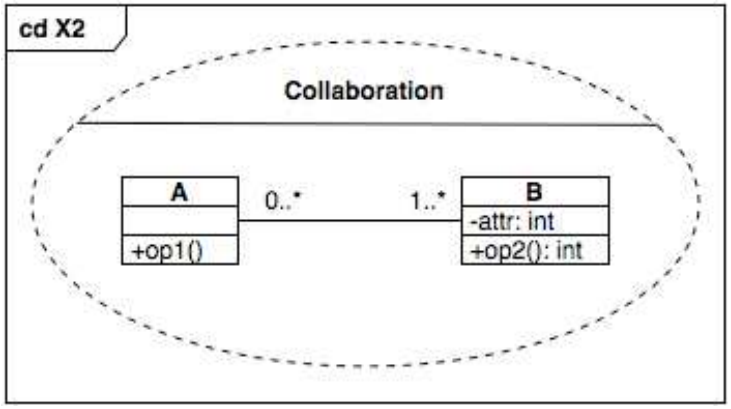
Ein Werkzeugkasten um neue Modelle zu bauen

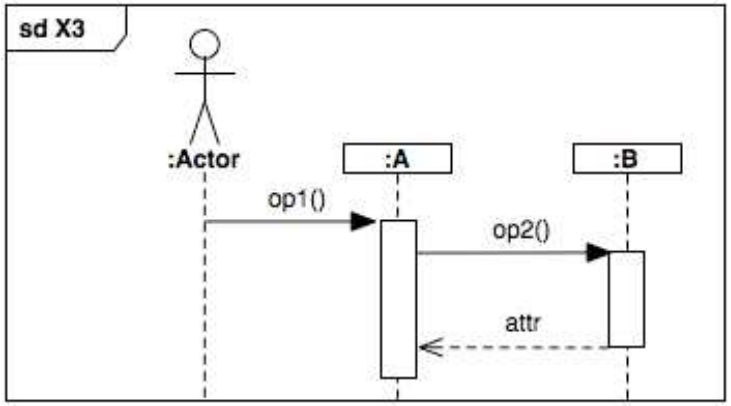
- gibt Möglichkeiten und Richtlinien vor
- schränkt allerdings auch ein

Aus: OO Modellierung, BIG, 2006

Modelle und Diagramme







UML **Modell** vs. UML **Diagramm**

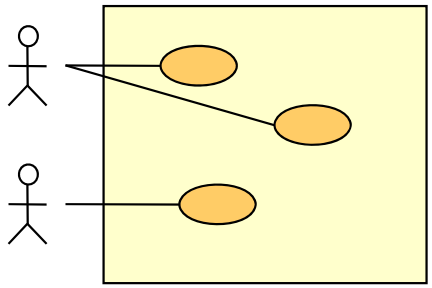
- Ein UML-Modell ist eine Menge von Modellelementen
  - Klassen, Attribute, Interaktionen, Anwendungsfälle, etc.
- Ein UML-Diagramm ist eine **Sicht** auf ein Modell)
  - Use Case Diagram (ud X1)
  - Class Diagram (cd X2)
  - Sequence Diagram (sd X3)
- Elemente in unterschiedlichen Diagrammen können sich auf ein und dasselbe Modellelement beziehen.

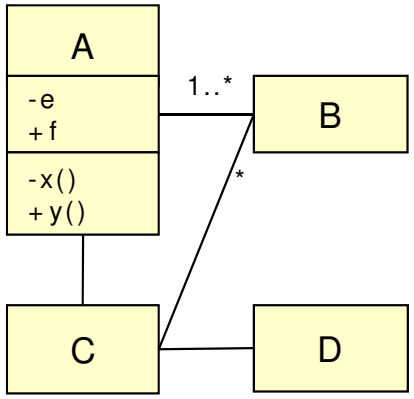
Werkzeuge

- **Modellierungswerkzeuge** implementieren die UML
  - Vorteile
    - Vorgabe der Modellstruktur, z.B. Konsistenzprüfer
    - Sourcecode-Generierung
  - Nachteil
    - oft komplex, schnelle Erstellung von Prototypen schwierig

- **Zeichenwerkzeuge** ermöglichen es nur, Diagramme zu zeichnen (z.B., auch ein Whiteboard ist ein Zeichenwerkzeug)
  - Vorteil
    - Freiheit in der Anwendung, evtl. besser Präsentationstauglich
  - Nachteil
    - unhandlich bei grösseren Modellen, z.B. mehrere Diagramme für ein Modell

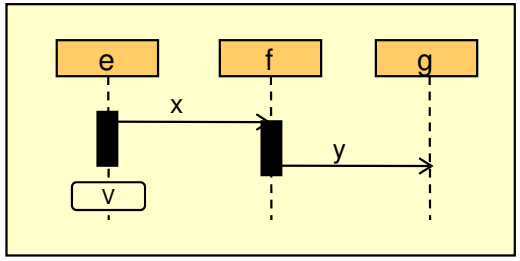
Schemata für statische Aspekte UML (“immer”)

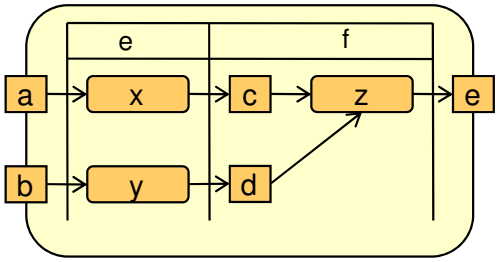


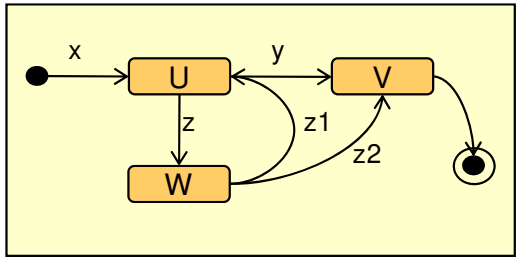


- **Anwendungsfälle**
  - die Benutzer und Nützlichkeit eines Systems
  - definiert durch Akteure und Hauptszenario
- **Klassen**
  - Menge gleichartiger Objekte
  - definiert durch Attribute, Operationen, und Beziehungen zu anderen Klassen
  - auch ein Datenbankschema kann in UML Klassendiagrammnotation modelliert werden.

Dynamische Aspekte mit UML ("wenn – dann")



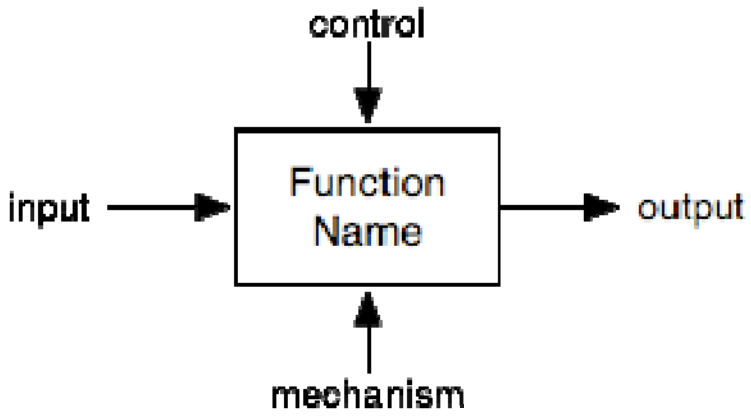




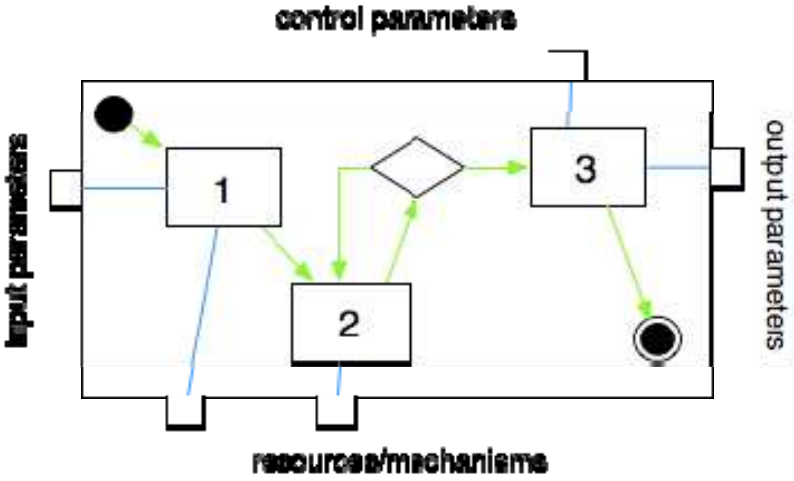
- **Sequenz**
  - ge- bzw. verbotene Interaktionsszenarien
  - definiert durch Folgen von Nachrichtenaustauschen zwischen Interaktionspartnern
- **Aktivität**
  - in sich geschlossenes, pro-aktives Verhalten
  - definiert durch Aktionen, Kontroll- und Datenfluss sowie evtl. Zuständigkeiten
- **Zustandsautomat**
  - reaktives Verhalten
  - definiert durch Zustände, Ereignisse und Transitionen

Aus: OO Modellierung, BIG, 2006

Prozess und Datenfluss Analyse (IDEFØ)



- Top-Down Modell
- Modellierung des Problembereichs
- Hilft neue Aufgabenbereiche zu erfassen und zu strukturieren



- Datenfluss (Daten-Token)
- Kontrollfluss (Kontroll-Token)
  - ergänzt um Start- und Endpunkt

Modellierung in der Softwareentwicklung

- **Konzeption**
  - Problemstellung, Geschäftsmodell
  - Big Picture – inkomplett, informal, vage
- **Anforderungen**
  - Problemstellung aufklären und ausarbeiten
  - Begriffe der Kundensprache (Fachbereich, Anwendung)
- **Entwurf**
  - Entwurfsalternativen explorieren und auswählen
  - Generische Begriffe der Softwaretechnik (Architektur, Komponenten, Algorithmen & Datenstrukturen)
- **Implementierung**
  - Implementierungsentscheidungen treffen, z.B., welche Datenbank, wie werden Assoziationen implementiert, etc.
  - Bestimmte Programmiersprachen, Bibliotheken, APIs

Modellierung

Aus: OO Modellierung, BIG, 2006

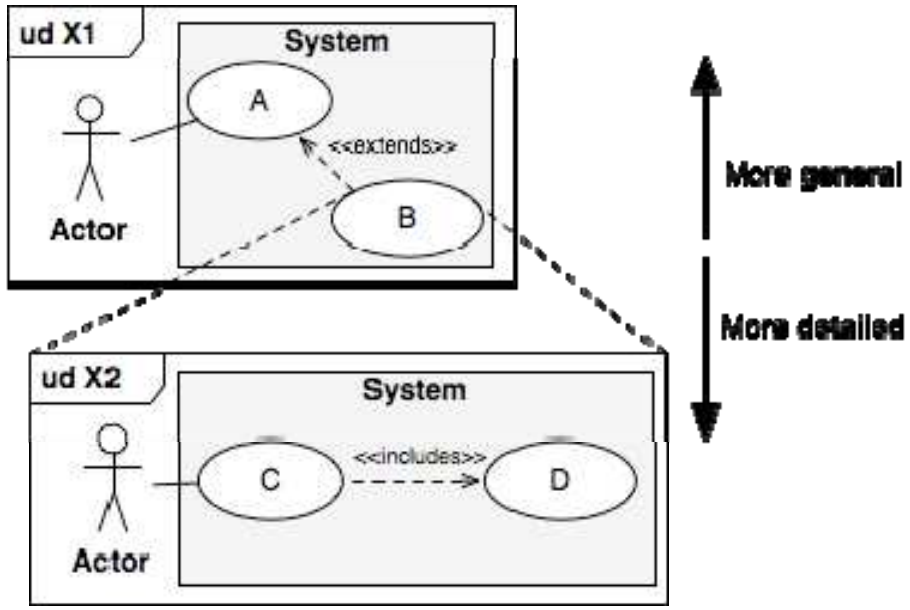
Ablauf SE: Wo werden Modelle eingesetzt

- Anforderungen ➔Modelle ➔Datenbank, Business Logic, GUI
  - Datenbanken: Datenmodellierung, Entity-Relationship Diagramm
  - Business Logic: Teilsysteme, Daten- und Kontrollfluss
  - Schnittstellen zwischen Teilsystemen und Benutzern

- Anforderungen ➔Qualitätsdefinition
- Modelle ➔Qualitätssicherung: Review, Testen
  - Anforderungen beschreiben Funktionen und Qualitäten
  - Zur operativen Definition sind die Qualitäten testbar zu beschreiben.

- Fokus: Test-Driven Development (TDD) Testfälle
  - Herstellen von ablauffähigen Testfällen **vor** der Implementierung
  - TDD überprüft die Spezifikation vor dem detaillierten Entwurf

Ablauf SE: Wie werden Modelle eingesetzt



- Anwendungsfalldiagramme sowie Daten- und Kontrollflussdiagramme werden Top-Down verfeinert
- Jede Phase im Entwicklungsprozess erfordert eine andere Abstraktionsebene in den Modellen
- System- und Modultests können anhand von Top-Down Modellen erstellt werden

Typische Fehler im Software Engineering SEPM LU Projekte

- Datenmodellierung, Reports
  - Schlüsselwerte nicht eindeutig, Fehlende Attribute
  - Fehler bei Beziehungen, z.B. 1:1 Relation im ER sehr selten
- Entwurf: Ergebniswerte von Methoden
  - Typkonversion falsch (z.B. Integer/Real)
- Initialisierung von Variablen
  - Fehlende Zuweisung: nicht eindeutiger Wert
  - Keyword static falsch verwendet: Klasse/Objekt (UML Rollen)
- Bedingungen
  - Logische Fehler; falsche Ausdrücke (und/oder)
- Schleifen
  - Anzahl Durchläufe nicht korrekt; z.B. Endlosschleifen
- Testfälle
  - Unzureichende Überdeckung (Auslassen von Use Cases, Programmteilen, GUI-Elementen, Datenzuständen)
  - Überflüssige Testfälle (gleiche Äquivalenzklasse)

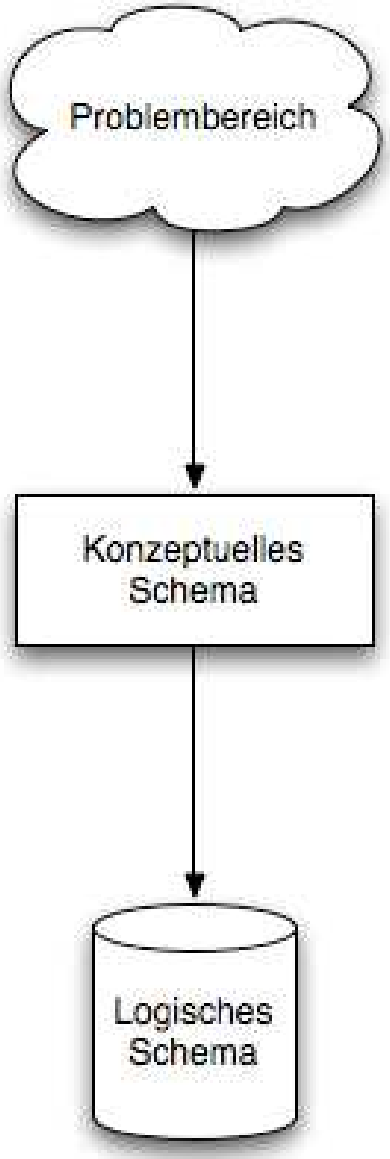
Ausblick: Test-Driven Development

- Ablauffähige Testfälle anhand eines Interfaces erstellen
- Assertions für „korrekte“ Durchführung des Testfalls ableiten (= erwartetes Ergebnis)
  - Normalfall soll nicht fehlschlagen
  - „Korrekter“ Fehlerfall muss auch wirklich fehlschlagen
    -> Testen mit Exceptions
- Brauchbare Modellierung unterstützt das Herstellen effektiver und effizienter Testfälle
  - Datenmodellierung (Datenbank)
  - Datenfluss und Kontrollfluss (Business Logic)
  - Zustandsübergangsmodelle (GUI)

Datenmodellierung (Struktur)

- Entity Relationship Diagramm in UML
  - Konzeptuelles und logisches Schema einer Datenbank
  - Entitäten, Attribute, Schlüssel, Relationen
  - Konsistenz- bzw. Integritätsbedingungen
  - Relationenmodell
- Anwendung von Datenmodellen in der Verifikation und Validierung
  - Datenorientierte Testfälle (Black Box)
  - Datenbank: SQLUnit
  - Unit Tests eines DAO Pattern via Mocking
- Anwendungsbeispiel “Restaurant”

Datenmodellierung Überblick



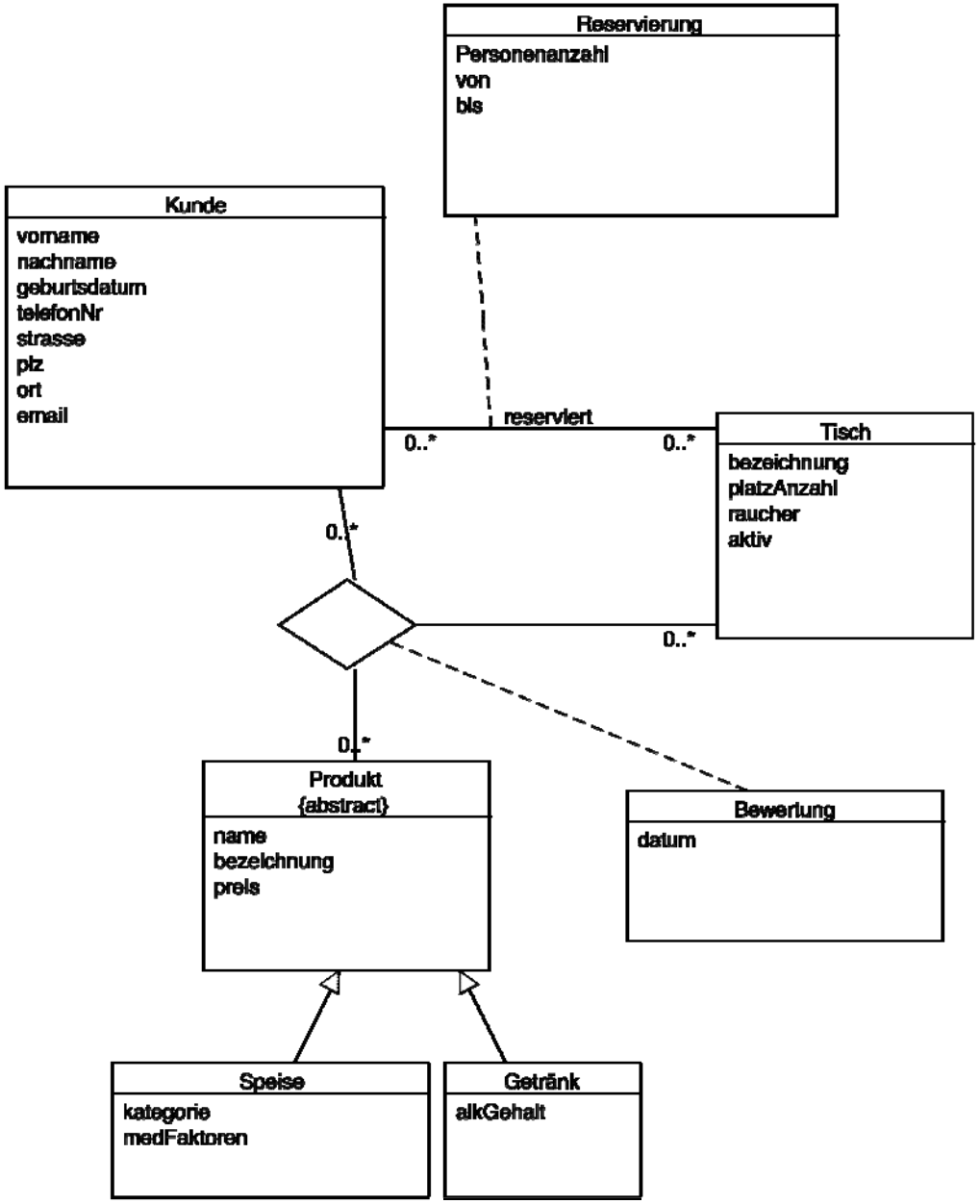
- Ausschnitt der Realen Welt
  - Beschrieben mit Text als Spezifikation oder Anforderungen.
  - Text Beschreibung Enthält Akteure, Entitäten, Aktivitäten und Operationen.
- Modellierung des Problembereichs durch Top-Down Diagramme wie Anwendungsfälle (UML) und Daten-/Kontrollfluss (IDEFØ)
  - Domänenmodelle, Klassendiagramme und Sequenzdiagramme (UML) führen zur Beschreibung des Softwaresystems
  - Datenmodelle (Entity Relationship) beschreiben die Persistenten Daten
  - Logisches Schema in dritter Normalform

Beispiel für Entity Relationship
Bewertung von Speisen und Getränken

Restaurant Datenbank:

- Kontext: Gehobenes Restaurant
- Ziel: Vorlieben der Stammgäste erheben
- Bewertung bei jedem Besuch

- Elemente:
  - Entitäten
  - Attribute
  - Beziehungen
    - Vielfachheiten

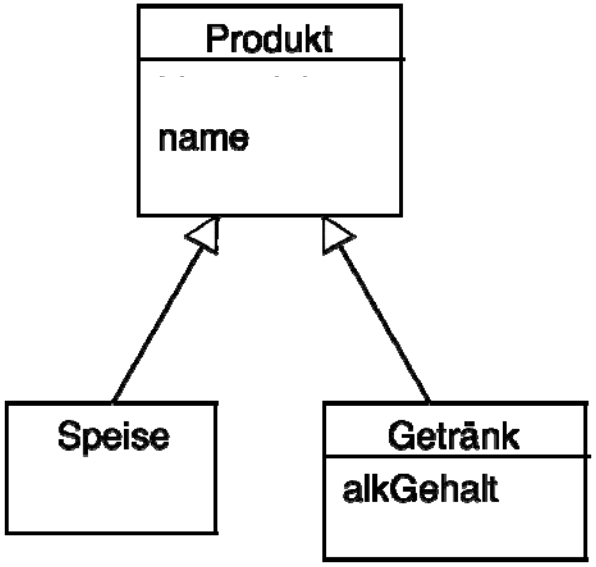


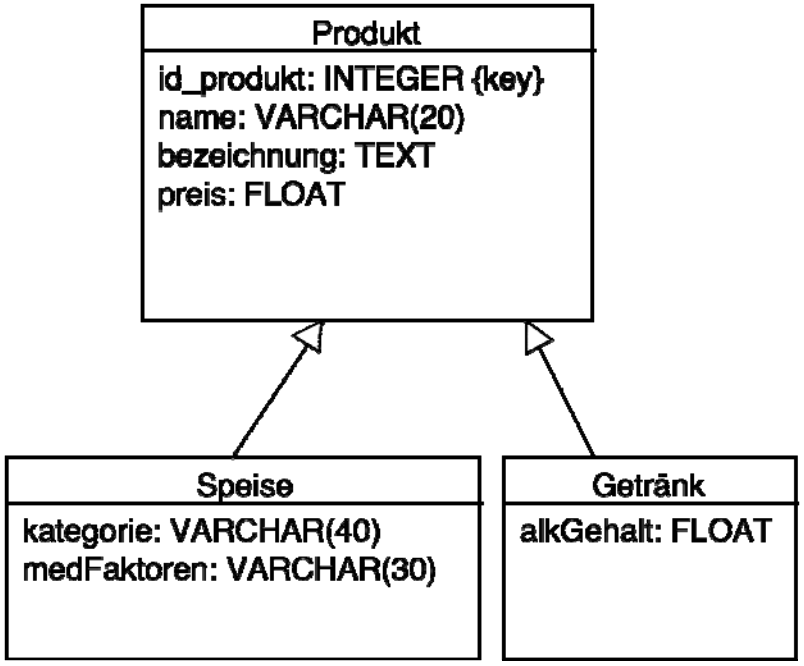
Beispiel „Restaurant“

- Typisches Prüfungsbeispiel
  - Siehe auch Prüfungsordner im TUWEL
- Auszug aus Anforderungsbeschreibung
  - Tip: Bei unklaren Anforderungen: Annahmen festhalten.
- Input für die Erstellung von diversen Modellen
- Einführung

Beispiel für Entity Relationship
Bewertung von Speisen und Getränken

- Bottom-Up Analyse: Entitäten ergänzen
  - Attribute, Typen und Schlüssel hinzufügen





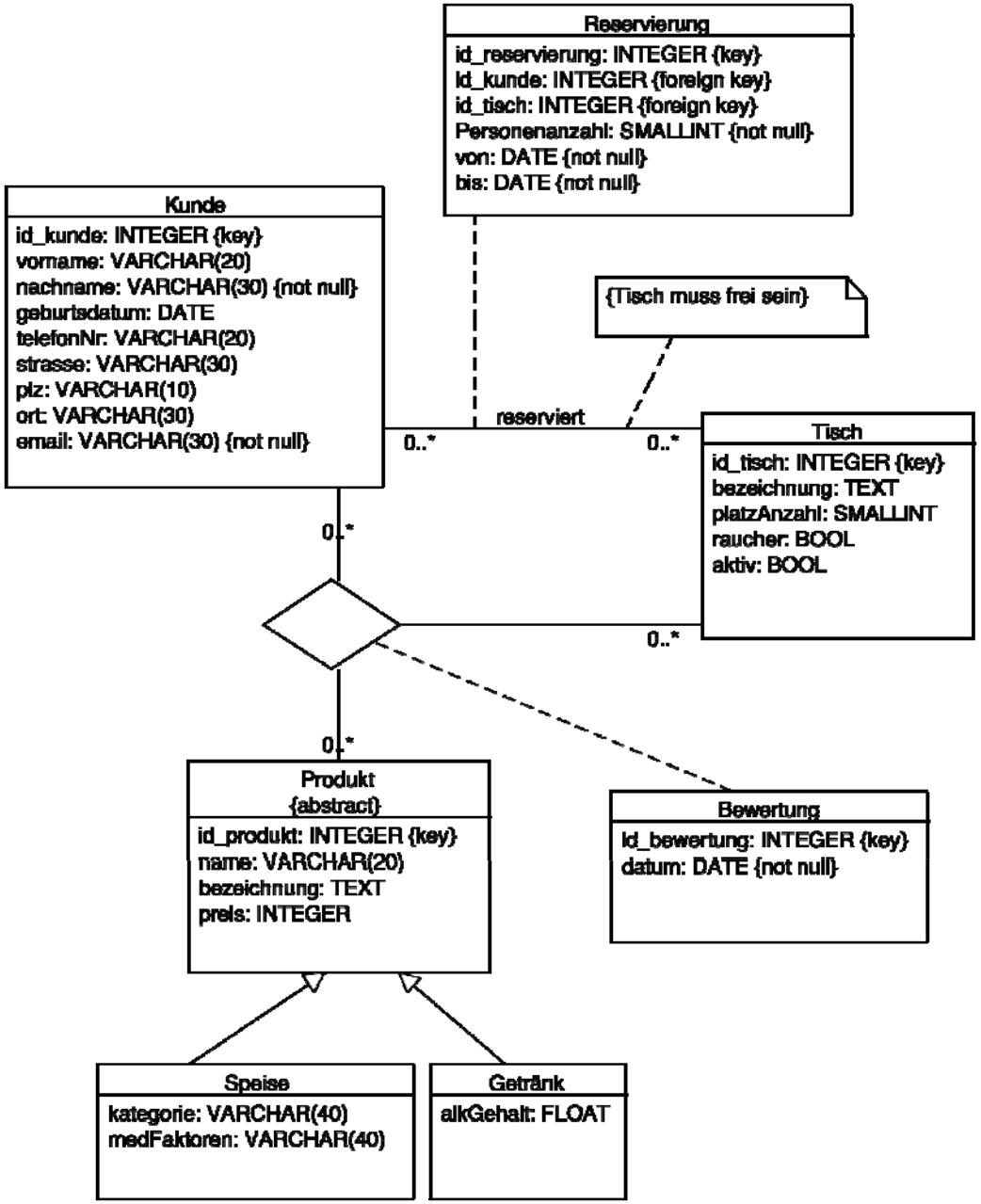
Entity Relationship: Verfeinerung

- Kandidaten für Entitäten identifizieren
  - Attribute zu Typen hinzufügen
  - Kandidaten für Schlüsselattribute markieren
- Kandidaten für Beziehungen zwischen Entitäten identifizieren
  - Vielfachheiten der beteiligten Entitäten überprüfen
    - Typisch: 1, 1..*, 0..1, 0..*
  - Beziehungen auf Attribute überprüfen
- Integritätsbedingungen
  - Pro Attribut (z.B. Zahl muss positiv sein)
  - Über mehrere Attribute (ev. auch Beziehungen) hinweg (z.B. eine Zahl muss kleiner als eine andere sein)

Beispiel für Entity Relationship
Bewertung von Speisen und Getränken

Verfeinerung:

- Entitäten
  - Eine Tabelle pro Entität
- Attribute
  - Schlüssel
    - Eigenschlüssel
    - Fremdschlüssel
  - Typen
- Integritätsbedingungen
- Beziehungen
  - Vielfachheiten
- Beschreibung als UML Notiz statt Tabelle



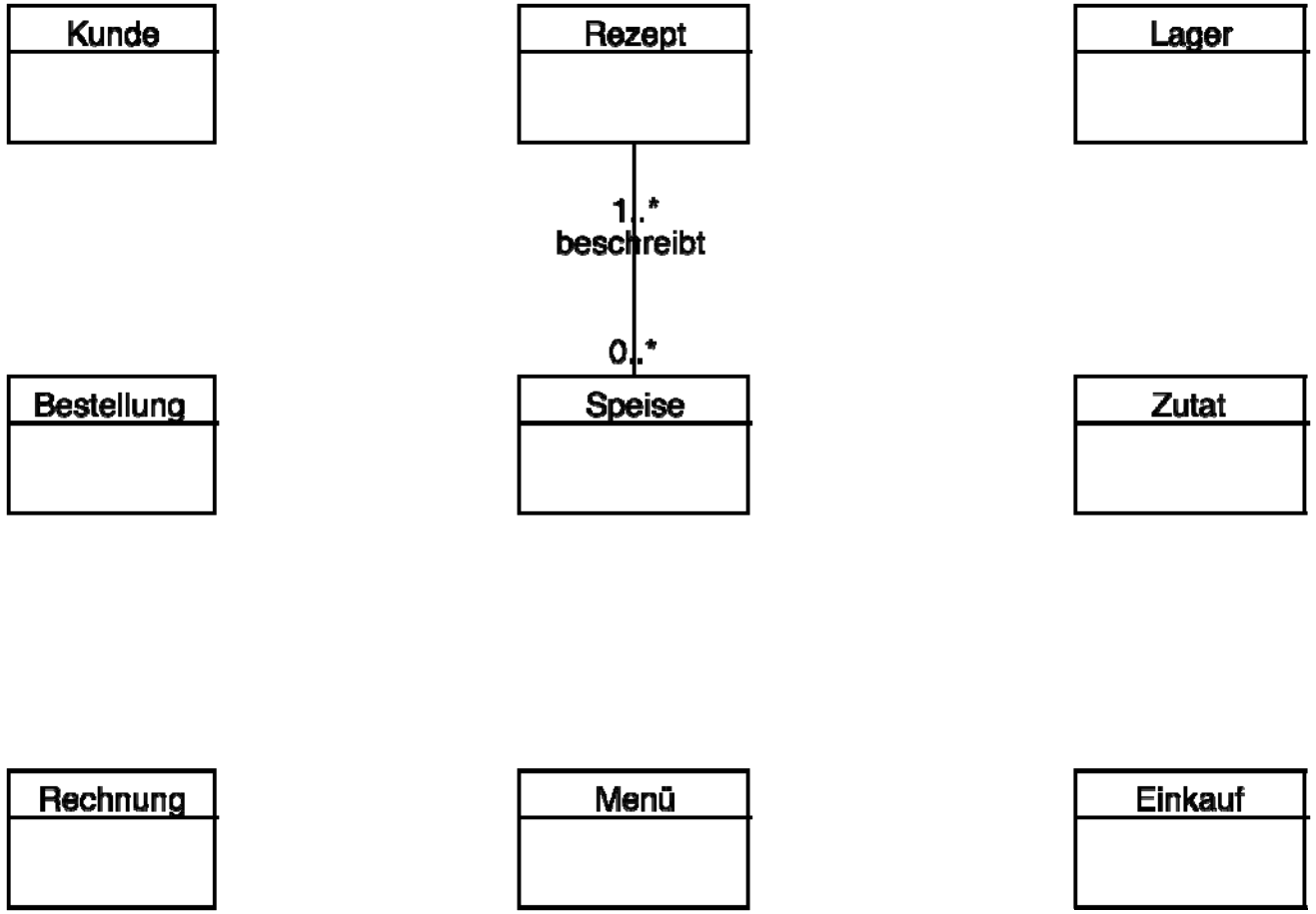
Zusammenfassung Datenmodellierung

- Ableitung von Datenbankrelationen
  - Jede Entität als Relation abbilden
  - Jede m:n Beziehung als neue Relation abbilden
  - Jede 1:n Beziehung einer bestehenden Relation hinzufügen.

- Datenbankrelationen: Attribute spezifizieren
  - Typen (VARCHAR, INTEGER, BOOL, FLOAT etc.)
  - Schlüssel überprüfen (müssen immer eindeutig sein)
    - Eigenschlüssel und Fremdschlüssel

Übung:
Restaurant – Sammlung von Entitäten

- Entitäten um Attribute ergänzen
- Beziehungen und Schlüssel hinzufügen

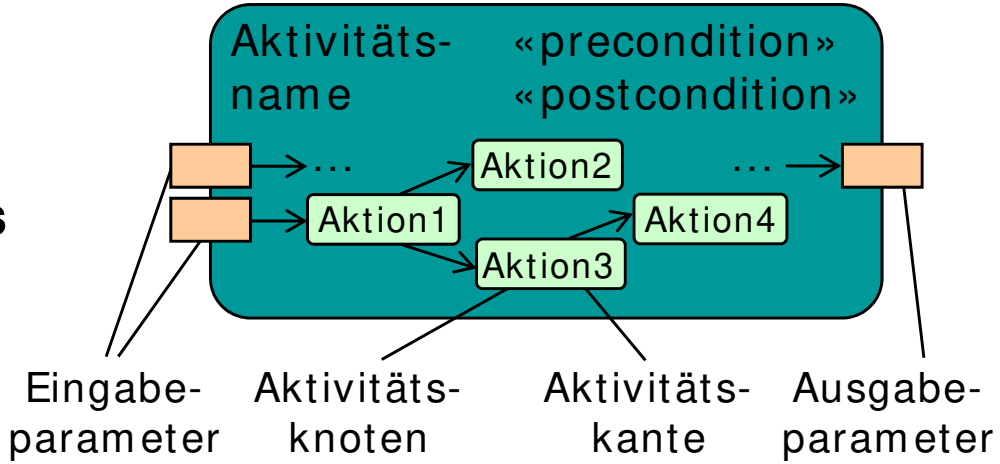


Objektorientierte Modellierung

- Aktivitätsdiagramm: Daten- und Kontrollfluss
  - Anwendungsfall: Typisches Verwendungsszenario für System
  - Aktivitätsdiagramm: Datenfluss zwischen Prozessabläufen
  - Zustandsdiagramm (State Chart): Falls bei Abläufen die Beschreibung von Zuständen und erlaubten Übergängen wichtig ist (GUI)
  - Sequenzdiagramm: Kommunikationsdetails zwischen Objekten.
- Anwendung der Modelle in der Verifikation und Validierung
  - Test-Driven Development: Testspezifikation, Testautomatisierung
  - Ableitung aus Modellen: Datenorientiert, Ablauforientiert
- Anwendungsbeispiel “Restaurant”

Aktivitätsdiagramm

Basiskonzepte – Bestandteile einer Aktivität



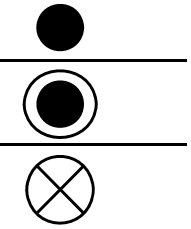
- Aktivität
  - Spezifiziert **benutzerdefiniertes Verhalten** auf unterschiedlichen Granularitätsebenen
  - Ist ein **gerichteter Graph**, bestehend aus **Knoten** und **Kanten**
  - Umfasst **Aktionen**, **Kontroll**- und **Datenflüsse**
  - Kann einem **Classifier zugeordnet** werden – direkt oder indirekt über Methode – oder aber »**autonom**« existieren
  - Wird durch **Operationsaufruf** oder **Classifier-Instanziierung** ausgeführt
  - Kann **Parameter** sowie Vor- und Nachbedingungen aufweisen und definiert einen Namensraum
- Aktion
  - Ist atomar
- Kontroll- und Datenflüsse legen potenzielle »Abläufe« fest

Aus: OO Modellierung, BIG, 2006

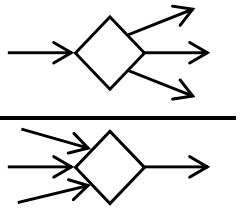
Aktivitätsdiagramm

Basiskonzepte – Knoten

- **Aktionsknoten** repräsentieren vordefinierte Aktionen
- **Objektknoten** dienen als formale Ein- und Ausgabeparameter sowie als zentrale Puffer
- **Kontrollknoten** steuern Aktivitätsabläufe
  - **Start und Ende von Abläufen**
    - Initialknoten
    - Aktivitätsendknoten
    - Ablaufendknoten



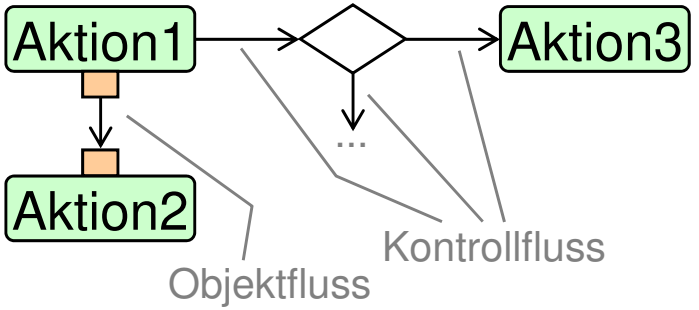
  - **Alternative Abläufe**
    - Entscheidungsknoten
    - Vereinigungsknoten



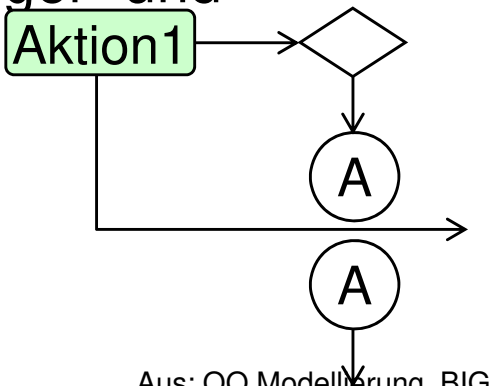
Aus: OO Modellierung, BIG, 2006

Aktivitätsdiagramm

Basiskonzepte – Kanten



- Kanten verbinden Knoten und legen **mögliche Abläufe** einer Aktivität fest
- **Kontrollflusskanten**
  - Drücken eine **reine Kontrollabhängigkeit** zwischen Vorgänger- und Nachfolgerknoten aus
- **Objektflusskanten**
  - Transportieren **zusätzlich Daten** und drücken dadurch auch eine **Datenabhängigkeit** zwischen Vorgänger- und Nachfolgerknoten aus
- **Überwachungsbedingung (guard)**
  - Bestimmt, ob Kontroll- und Datenfluss weiterläuft oder nicht
- **Fortsetzungsmarken (connectors)**

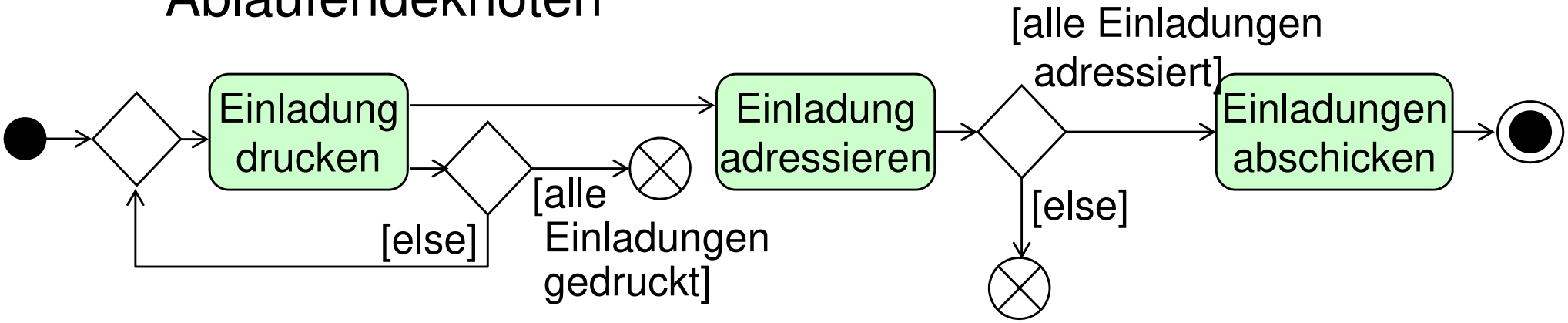


Aus: OO Modellierung, BIG, 2006

Aktivitätsdiagramm

Basiskonzepte – Start und Ende von Aktivitäten und Abläufen 2/2

- Beispiel mit Initial-, Aktivitätsende- und Ablaufendeknoten

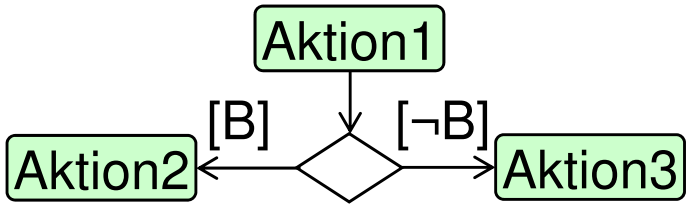


Aus: OO Modellierung, BIG, 2006

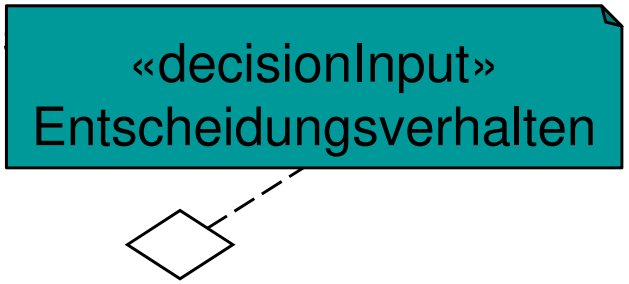
Aktivitätsdiagramm

Basiskonzepte – Alternative Abläufe 1/2

- **Entscheidungsknoten**
  - Definiert alternative Zweige und repräsentiert eine »Weiche« für den Tokenfluss
    - Verwendung auch zur Modellierung von Schleifen



  - **Überwachungsbedingungen**
    - Wählen den Zweig aus
    - Müssen wechselseitig ausschließend sein, [else] ist vordefiniert
  - **Entscheidungsverhalten**
    - Ermöglicht detailliertere Spezifikation der Au
      an zentraler Stelle
    - Ankunft von Token startet das Entscheidungsverhalten – Datentoken fungieren als Parameter

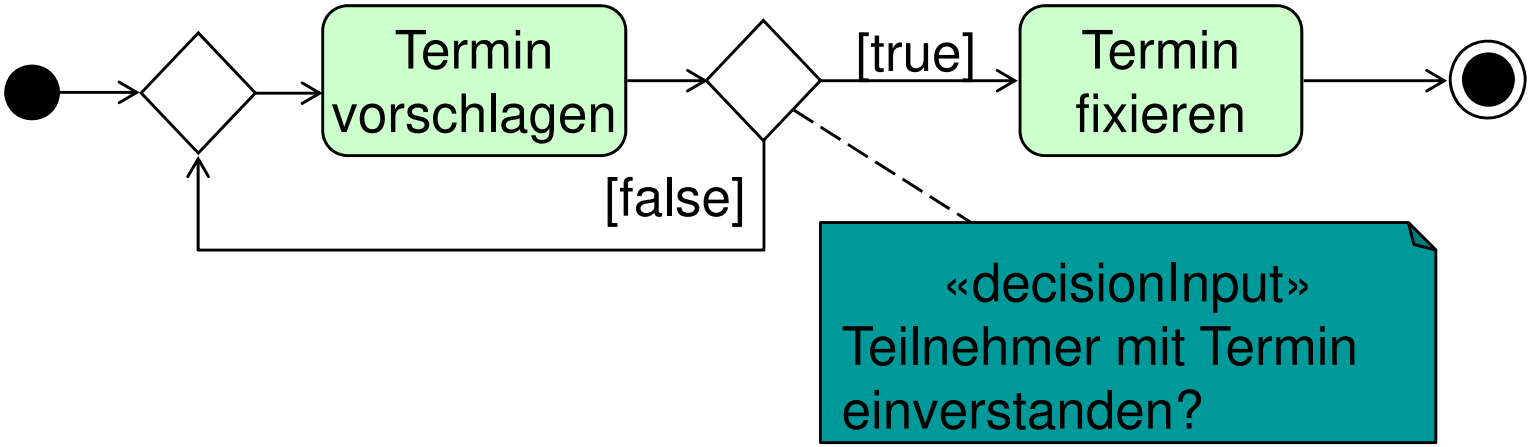


Aus: OO Modellierung, BIG, 2006

Aktivitätsdiagramm

Basiskonzepte – Alternative Abläufe 2/2

- Beispiel mit Entscheidungsknoten und -Verhalten

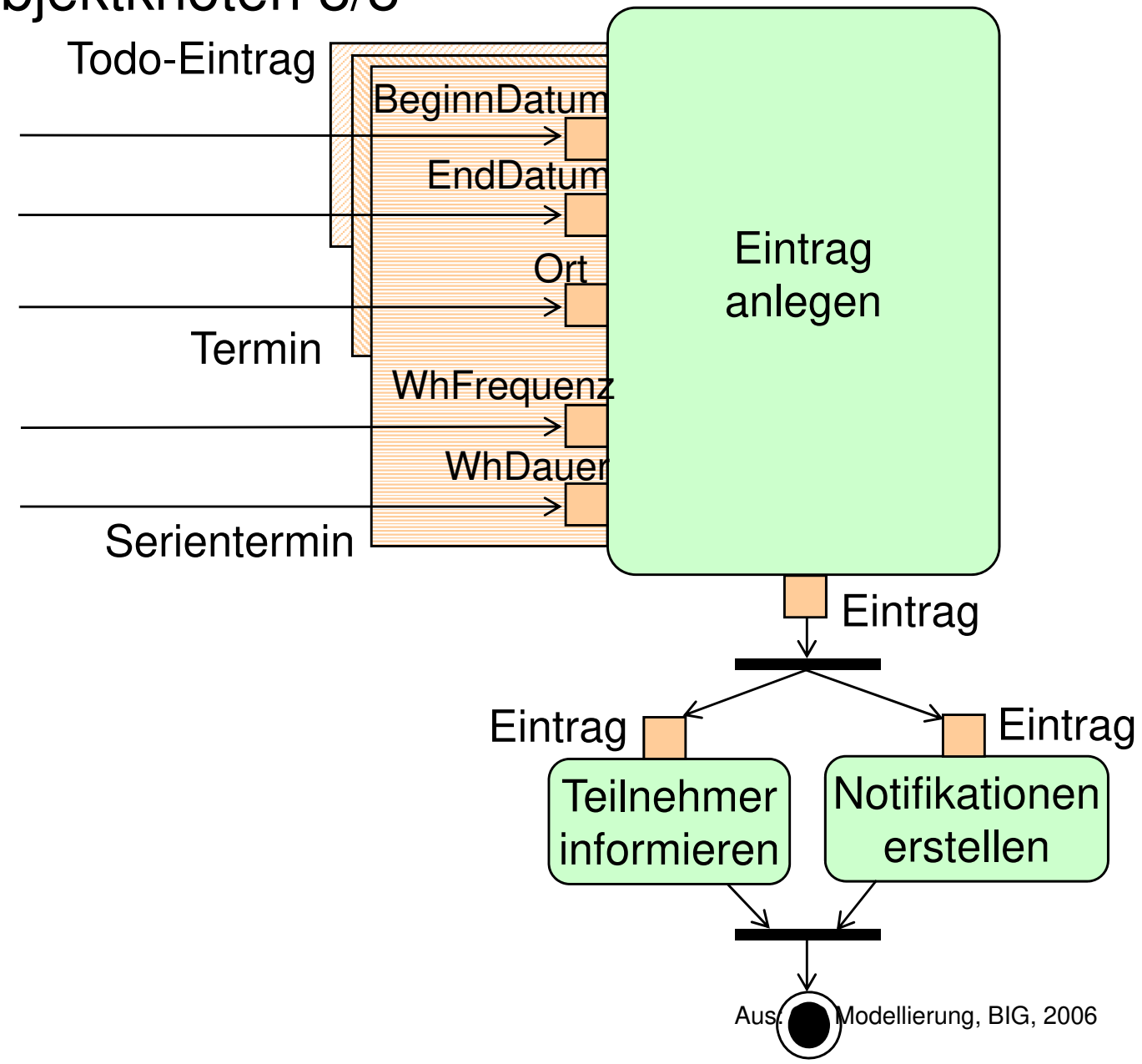


Aus: OO Modellierung, BIG, 2006

Aktivitätsdiagramm

Basiskonzepte – Objektknoten 8/8

- Beispiel für Parametersätze



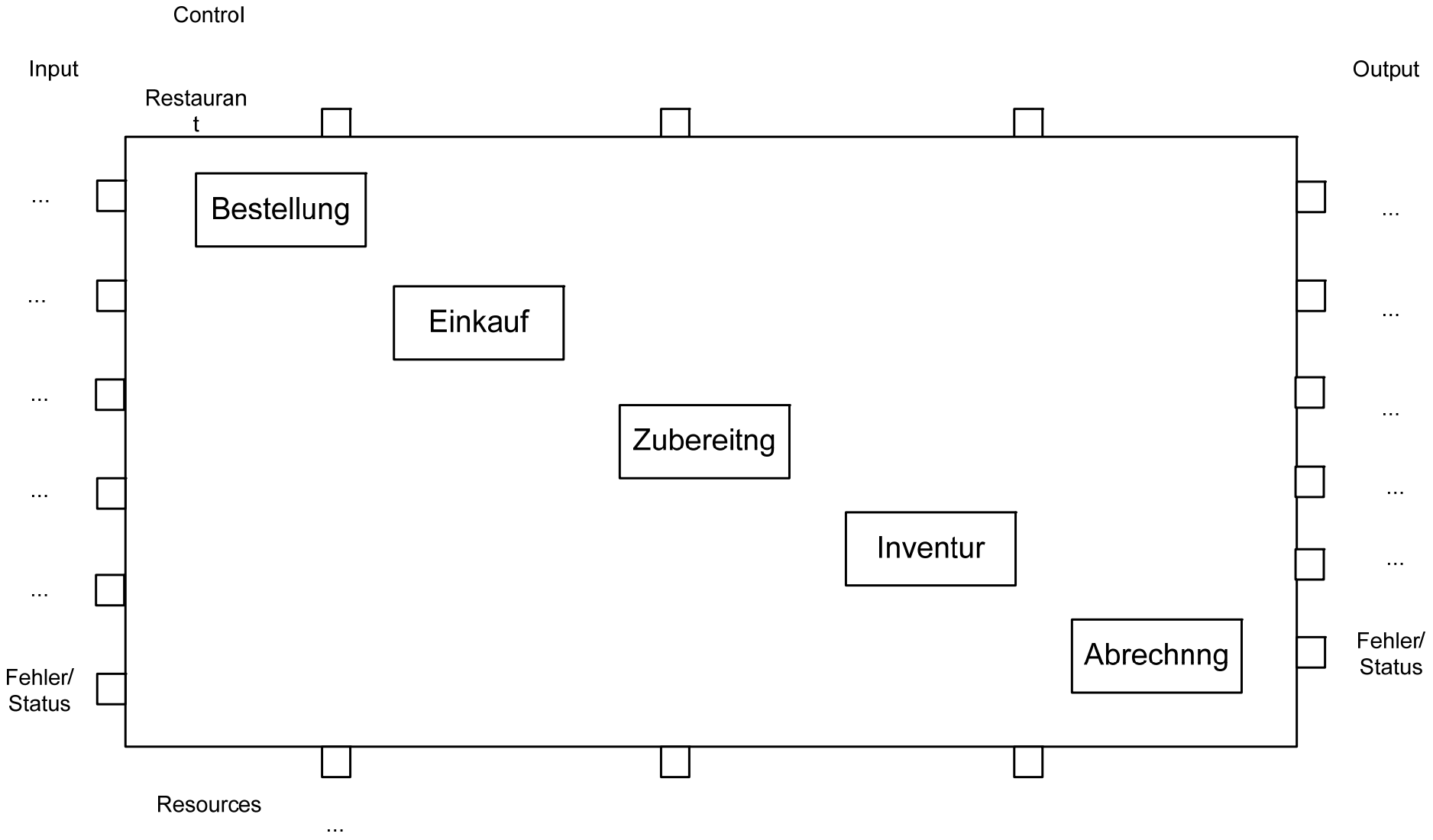
Aus: Modellierung, BIG, 2006

Aktivitätsdiagramm (Business Logic)
Modellieren von Daten- und Kontrollflüssen

- Basierend auf Anwendungsszenario
  - Rollen und Aktivitäten
- Detailgrad von Prozessen
  - System und Teilsysteme: Schnittstellen
  - Teilsystem und Aktionen: Detaillierter Kontrollfluss
- Ableitung von Prozessen
  - Datenfluss: Ein- und Ausgabedaten
  - Kontrollfluss: Logische Darstellung eines Ablaufes
    - Entscheidungen und Schleifen

Beispiel Datenfluss – organisatorische Untereinheiten

- Links: Eingabeparameter
- Rechts: Ausgabeparameter
- Oben: Steuerparameter
- Unten: Ressourcen



Ablauf Aktivitätsdiagramm - Kontrollfluss

- Fokus auf Transformation der Eingangsparameter in Ausgangsparameter
  - Startpunkt(e) identifizieren
  - Aktionen bzw. Sub-Aktivitäten beschreiben
  - Entscheidungspunkte ergänzen
  - Endpunkt(e) identifizieren

  - Optional: Parallele Kontrollflüsse
    (siehe fortgeschrittene Konzepte der OO-Modellierung)

Ablauf Aktivitätsdiagramm - Kontrollfluss

- Überprüfung
  - An Endpunkten müssen Ausgangsparameter gültige Werte haben (entsprechend der Spezifikation der Aktivität)
  - Testfälle entwickeln, die abdecken:
    - Alle Alternativen des Anwendungsszenarios (Use Case) zur Aktivität sind abgebildet
    - Alle Knoten und Kanten
    - Wesentliche Pfade (ev. mehrfaches Besuchen von Knoten notwendig)
    - Variationen der Eingabedaten (Äquivalenzklassen)
    - Variationen der Ausgabedaten (Fälle aus Ergebniskandidaten ableiten)

Beispiel für Testfallbeschreibung
Tische verwalten für Reservierungen

- Testfälle mit erwartetem Ergebnis.

Komponente: Tische
Vorbedingung: Programm gestartet, in die Tischübersicht gewechselt

Nr.	Typ	Kurzbeschreibung	Eingaben	Ergebnis	erw. Ergebnis	Fehlerfallbeschreibung.	ok / nok
14.	FF	Tische; Tisch anlegen	Tischnummer, Tischname	fehlerhaft	Eingabemaske: Tischdaten werden erfaßt und gespeichert	Anlegen von bereits vergebenen Tischnummern.	
15.	NF	Aktualisieren der Tischdaten	Tischdaten werden geändert	fehlerhaft	Infomeldung erfolgreiche Änderung		
16.	NF	Tisch löschen	Tisch auswählen, Löschvorgang starten	Bestätigung / Hinweis	Tisch wird aus der DBgelöscht		
17.	FF	Tisch löschen	unmögliche / doppelte Tischdaten	Bestätigung / Hinweis	Fehlermeldung		
18.	FF	Tisch anlegen	Doppelte / unmögliche Tischdaten	wird angelegt	Fehlermeldung		

Ablauf Testfälle bestimmen

- Ziel aus Testplan bestimmen
  - Z.B. Normal-, Sonder- und Fehlerfälle in einem Anwendungsszenario abdecken
  - Z.B. Alle Knoten und Kanten in einem Kontrollflussgraphen abdecken
- Fälle bestimmen
  - Eingabeparameter
  - Entscheidungen im Ablauf (Wahl der Entscheidung festhalten)
- Erwartetes Ergebnis bestimmen
- Erreichung des Testziels mit Menge der spezifizierten Testfälle überprüfen

Beispiel Testfälle Restaurant.Bestellung

- I. Prüfung Bestellwunsch
  - Äquivalenzklassen der Parameter
  - Mögliche Ergebnisse
- II. Berechnung des Bestellwerts
- III. Prüfung Anzahlung für Großbestellung
  - Prüfung Anzahlung in Ordnung
- Test-Driven Development
  - Ableitung von Testfällen
  - Assertions für Unit Test
    - Programmstück, das Korrektheit der Logik überprüft; Fehlermeldung, falls eine logische Bedingung verletzt wird.
    - Hilft Fehler zu lokalisieren.

Beispiel für Testfallbeschreibung
Tische verwalten für Reservierungen

- Testfälle mit erwartetem und tatsächlichem Ergebnis.

Komponente: Tische
Vorbedingung: Programm gestartet, in die Tischübersicht gewechselt

Nr.	Typ	Kurzbeschreibung	Eingaben	Ergebnis	erw. Ergebnis	Fehlerfallbeschreibung.	ok / nok
14.	FF	Tische; Tisch anlegen	Tischnummer, Tischname	Fehler mit Hinweis	Eingabemaske: Tischdaten werden erfaßt und gespeichert	Anlegen von bereits vergebenen Tischnummern.	NOK
15.	NF	Aktualisieren der Tischdaten	Tischdaten werden geändert	fehlerhaft	Infomeldung erfolgreiche Änderung	TischNr* und Sitzplatzzahl sind vertauscht	N OK
16.	NF	Tisch löschen	Tisch auswählen, Button „löschen“	Bestätigung / Hinweis	Tisch wird aus der DBgelöscht		OK
17.	FF	Tisch löschen	unmögliche / doppelte Tischdaten	Bestätigung / Hinweis	Fehlermeldung	Begründung falls Tisch nicht gelöscht werden kann; zB Reservierung aktiv.	NOK
18.	FF	Tisch anlegen	Doppelte / unmögliche Tischdaten	wird angelegt	Fehlermeldung	Tischnummer darf nicht mehrfach vergeben werden	NOK

Zustandsdiagramm
Einführung

- Ein Zustandsdiagramm (State Machine Diagram) beschreibt die möglichen **Folgen von Zuständen** eines **Modell-elements**, i.A. eines Objekts einer bestimmten Klasse
  - Während seines **Lebenslaufs** (Erzeugung bis Destruktion)
  - Während der **Ausführung** einer **Operation** oder **Interaktion**

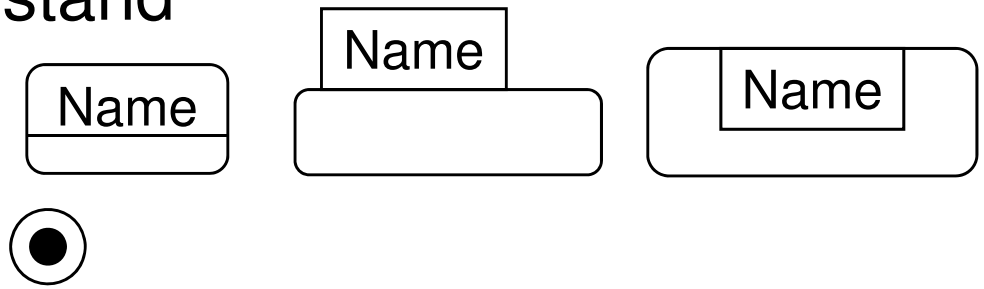
- Modelliert werden
  - Die **Zustände**, in denen sich die Objekte einer Klasse befinden können
  - Die möglichen **Zustandsübergänge** (Transitionen) von einem Zustand zum anderen
  - Die **Ereignisse**, die Transitionen auslösen
  - **Aktivitäten**, die in Zuständen bzw. im Zuge von Transitionen ausgeführt werden

Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Basiskonzepte – Zustand

- Zustand (state)
  - Zustand i.e.S.
  - Endzustand



- Pseudozustände (weil transient)
  - Initialzustand
  - History-Zustand, Synch-Zustand, Gabelung, Vereinigung, etc.

- **Aktivität** innerhalb eines Zustands
  - **entry** / *aktivität*
    Aktivität wird beim Eingang in den Zustand ausgeführt
  - **exit** / *aktivität*
    Aktivität wird beim Verlassen des Zustands ausgeführt
  - **do** / *aktivität*
    Aktivität wird ausgeführt, Parameter sind erlaubt
  - ***event*** / *aktivität*
    Aktivität behandelt Ereignis innerhalb des Zustands

Aus: OO Modellierung, BIG, 2006

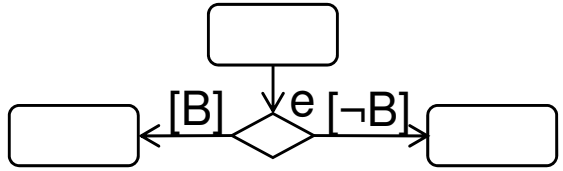
Zustandsdiagramm

Basiskonzepte – Zustandsübergang

- Ein Zustandsübergang (state transition) erfolgt, wenn
  - das **Ereignis** eintritt – eine evt. noch andauernde **Aktivität** im Vorzustand wird **unterbrochen**!
  - und die **Bedingung** (guard) erfüllt ist – bei Nicht-Erfüllung geht das nicht »konsumierte« Ereignis **verloren**
- Durch entsprechende Bedingungen können **Entscheidungsbäume** modelliert werden
- Standardannahmen
  - **Fehlendes Ereignis** entspricht dem Ereignis »Aktivität ist abgeschlossen«
  - **Fehlende Bedingung** entspricht der Bedingung [true]
- **Aktionen** auf Zustandsübergängen möglich
  - Spezielle Aktion: Nachricht an anderes Objekt senden

    send empfänger.nachricht()
  - Beispiel:

    right-mouse-down (loc) [loc in window]
    / obj:= pick-obj (loc); send obj.highlight()

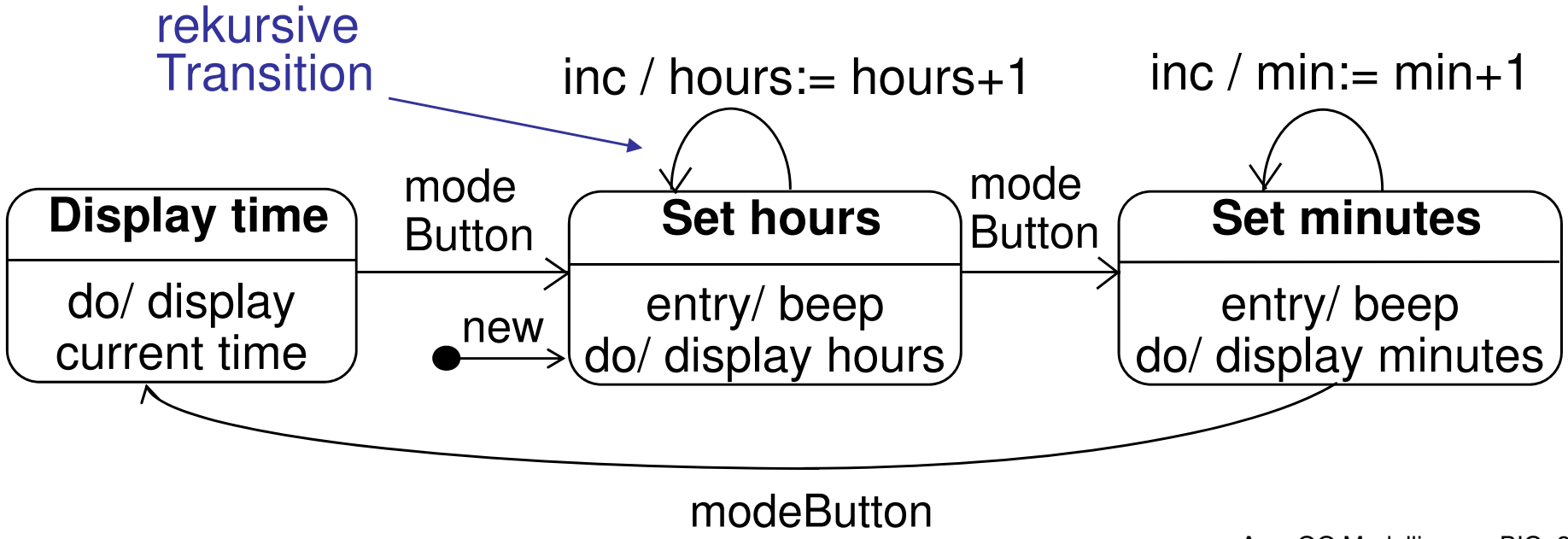


Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Basiskonzepte – Beispiel: Klasse DigitalWatch

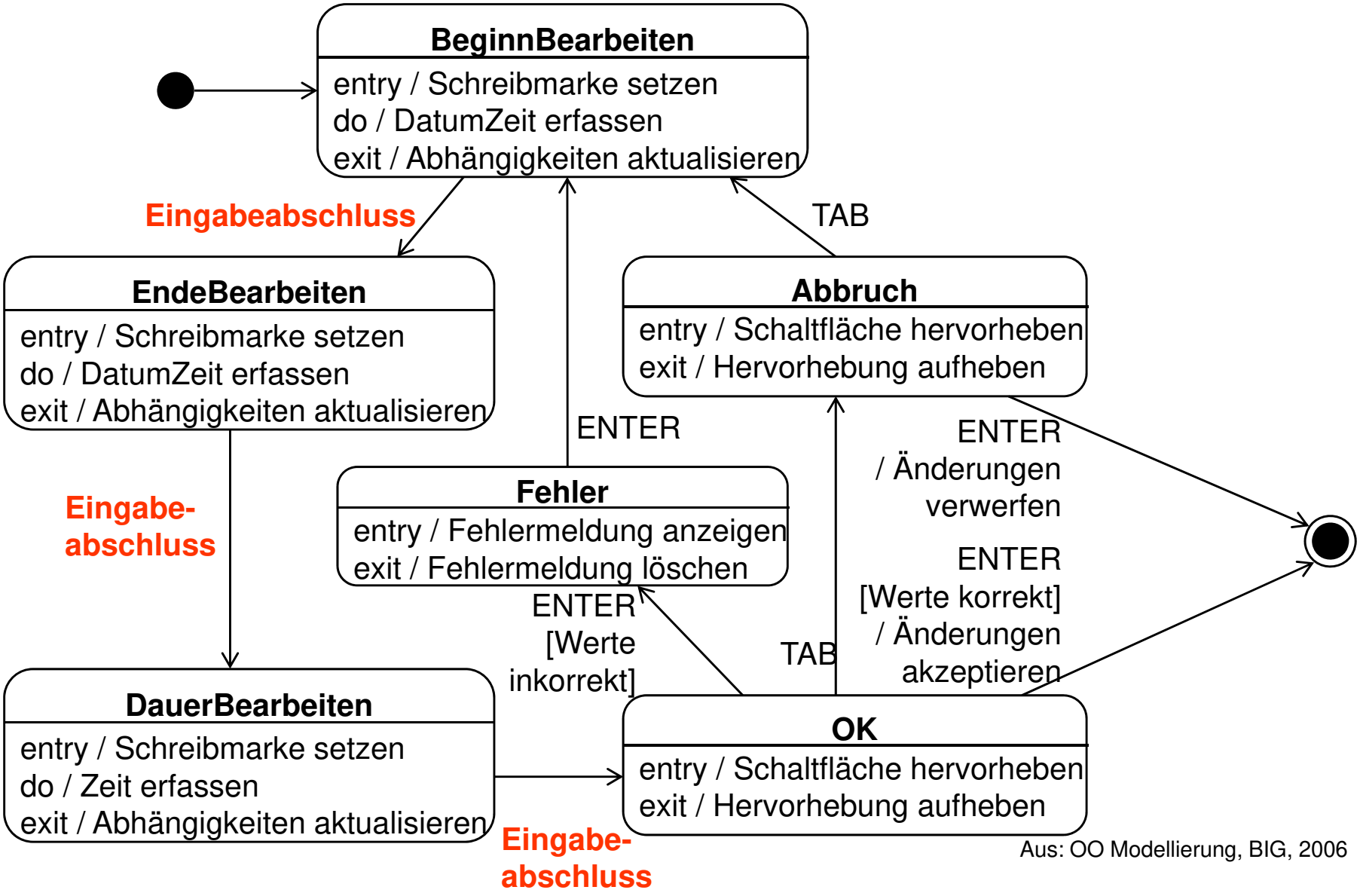
DigitalWatch
min : Integer = 0 hours : Integer = 0
modeButton() inc()



Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Basiskonzepte – Beispiel: Lebenslauf einer Termin-Eingabemaske 3/3

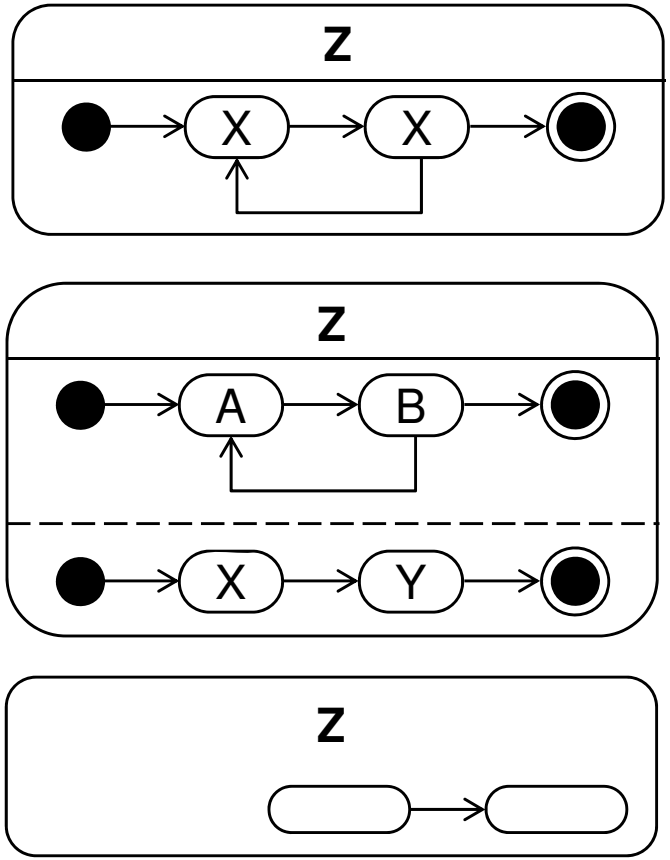


Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Strukturierung – ODER- vs. UND-Verfeinerung

- Verfeinerung eines **komplexen Zustands (composite state)** — geschachteltes Zustandsdiagramm
- **ODER-Verfeinerung**
  - Disjunkte Sub-Zustände, d.h. **genau ein Subzustand ist aktiv**, wenn der komplexe Zustand aktiv ist
- **UND-Verfeinerung**
  - Nebenläufige Sub-Zustände, d.h. **alle Subzustände sind aktiv**, wenn der Superzustand aktiv ist
  - Die Subzustände werden i.A. ihrerseits Oder-verfeinert
- Hinweis auf **ausgeblendete Verfeinerung**
  - Diese kann an anderer Stelle dargestellt werden

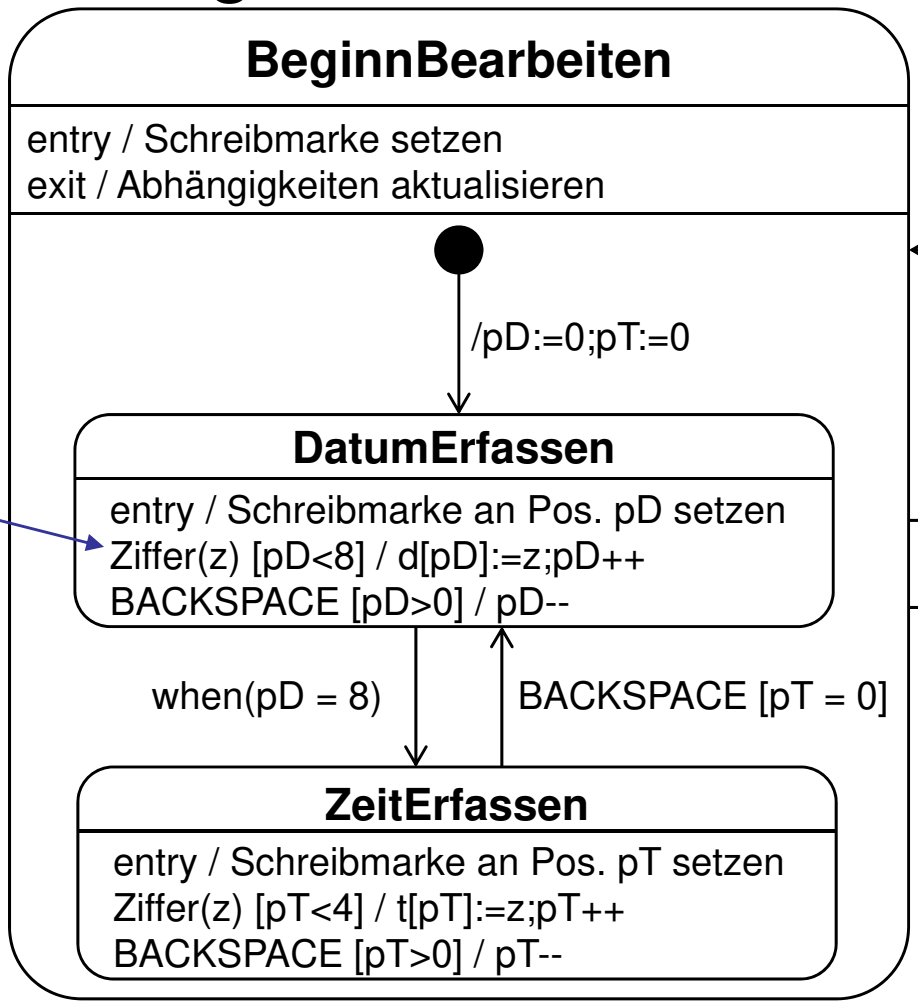


Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Strukturierung – Beispiel 1/6

- Lebenslauf einer Termin-Eingabemaske
- **ODER-Verfeinerung** der Aktivität »DatumZeit erfassen«



Parametrisiertes Ereignis

Gruppen-Transitionen (nur ausgehende!) werden an alle Subzustände »distribuiert«

Aus: OO Modellierung, BIG, 2006

Ablauf Zustandsübergangsdiagramm

- Entitäten und deren Zustände identifizieren
- Parallele Zustände aufteilen

- Für jede Entität
  - Initialzustand bestimmen
- Zustandsübergänge einzeichnen
    - Ereignis(se) bestimmen, die den Übergang anstoßen
  - Wo sinnvoll: Bedingungen für Übergänge bestimmen

- Überprüfung
  - Abläufe des Anwendungsszenarios durchspielen
  - Testfälle durchspielen
  - Überdeckung: Alle Knoten und Kanten abdecken.

Zusammenfassung

- Test-Driven Development als Basis für zielorientierte Software-Entwicklung (Implementierung und QS)
- Brauchbare Modellierung unterstützt das Herstellen effektiver und effizienter Testfälle
  - Datenmodellierung (Datenbank)
  - Datenfluss und Kontrollfluss (Business Logic)
  - Zustandsübergangsmodelle (z.B. GUI)
- Nächste Einheit zu diesem Themenbereich: SE Prozesse im Überblick

Referenzen

Bücher und Skripten

- „Datenmodellierung“; Unterlagen zur Vorlesung; TU Wien, Inst. f. DB & AI, 2006.
- Kemper A., Eickler A.; „Datenbanksysteme“; 6. Auflage, Oldenbourg, 2007.
- „Objektorientierte Modellierung“; Unterlagen zur Vorlesung; Business Informatics Group (BIG), TU Wien, 2006.
- Sommerville I.; „Software Engineering“; 8. Auflage; Addison Wesley, 2007 (v.a. Kapitel 2, 3, 8, 14, 16, 19, 22).
- Booch, Rumbaugh, Jacobson: "The UML User Guide", 2. Auflage, Addison Wesley, 2005

Web resources

- SEPM Tuwel: https://tuwel.tuwien.ac.at/course/view.php?id=33
- Sommerville book companion website: www.pearsoned.co.uk/sommerville
- Integrated DEFinition Methods: http://www.idef.com/idef0.html

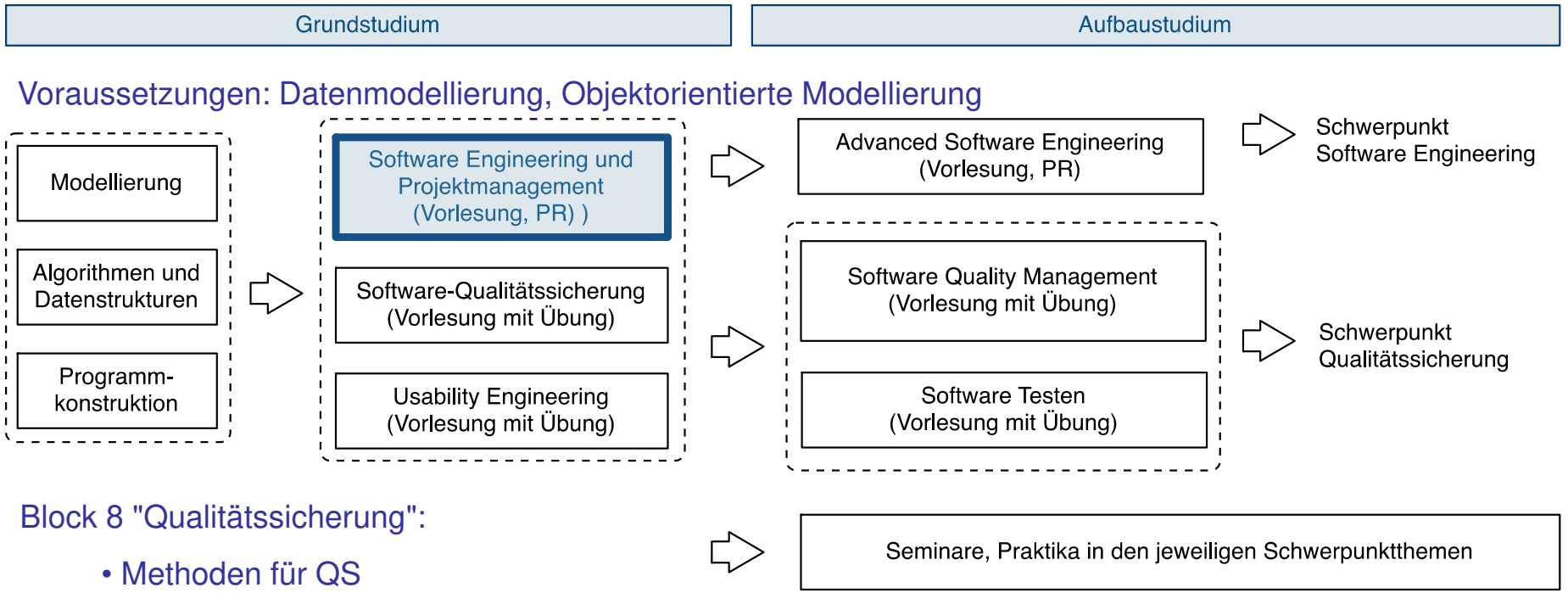
Software Engineering & PM Vorlesung
Qualitätssicherung

Institut für Information Systems Engineering
Stefan Biffl

Inhalt:

- Qualitätssicherung in SEPM
  - Grundlagen der QS
- SEPM PR: Kernmethoden und deren Anwendung
  - Review: Anforderungen – Modelle (ER, IDEF0, UML)
  - Testen: Erstellen, Beurteilen, Anwenden von Testfällen
  - Test-Driven Development
- SEPM VO: Prüfungsbeispiel "Restaurant„
- BPSE Buch, Kapitel 5

Software Engineering im Studium



Block 8 "Qualitätssicherung":

- Methoden für QS
  - Reviews
  - Test-Driven Development

Überblick Block 8: Qualitätssicherung

1. Qualitätssicherung in SEPM-Projekten

- Grundlagen, Begriffe
- Typische SE-Modelle und deren Verwendung für QS
- Testfälle, Test-Driven Development

2. Statische Methoden der QS in SEPM

- Definition und Arten von Reviews
- Anwendungsbeispiel “Restaurant”
  - Überprüfung der Anforderungen
  - Aus den Anforderungen abgeleitete Modelle
- Reviews von Anforderungen mit ER/UML/IDEF0 Diagrammen

3. Testansätze, Test-Driven Development

- Testen in der Qualitätssicherung und SE Prozessen
- Teststufen und Testarten
- JUnit, Test-Driven Development
- Beurteilung der Güte von Testfällen

Qualitätssicherung in SEPM

Qualitätssicherung in SEPM-Projekten

- Grundlagen, Begriffe
  - Qualität, Qualitätssicherung
  - Verifikation und Validierung
- Typische SE-Modelle und deren Verwendung für QS
  - Anwendungsszenarien, Datenmodelle, Zustandsdiagramme (UML)
  - Daten-/Kontrollflussmodelle (IDEF0, UML)
- Testfälle, Test-Driven Development
  - Herstellen von Testfällen vor der Implementierung
  - Ablauffähige Testfälle für automatischen Re-Test neuer Code-Teile
  - Frühe Review missionskritischer Akzeptanztestfälle

Motivation - Ziel

- Software-Entwicklung erfordert die Herstellung konsistenter Sichten auf Anforderungen und Entwurf eines Systems.
- Modelle helfen den Überblick zu bekommen und zu behalten.
  - Grundlage für effektives und effizientes Arbeiten im Team.
  - Gemeinsame Notation mit konsistenter Bedeutung.

- Reviews überprüfen die Korrektheit und Konsistenz von Artefakten an wohldefinierten Punkten des Entwicklungsprozesses.
  - Review der Anforderungen nach Änderungen
  - Review später hergestellter Dokumente gegen die aktuell gültigen Anforderungen

- Herausforderung: Konsistente Verwendung unterschiedlicher Modelle.
  - Systemstruktur: Subsysteme, Komponenten, Schnittstellen.
  - Verhalten von Komponenten; Interaktion zwischen Komponenten.
- Anforderungen ➔ Modelle ➔ Daten/Komponenten/Testspezifikationen.

- Tests überprüfen das Laufzeitverhalten von Systemteilen
  - Funktionen, Modelle und Qualitätsmerkmale
  - Ausgangsbasis: Anforderungen, Ein-/Ausgangsparameter des Systems, innere Systemstruktur

Qualität und Qualitätssicherung

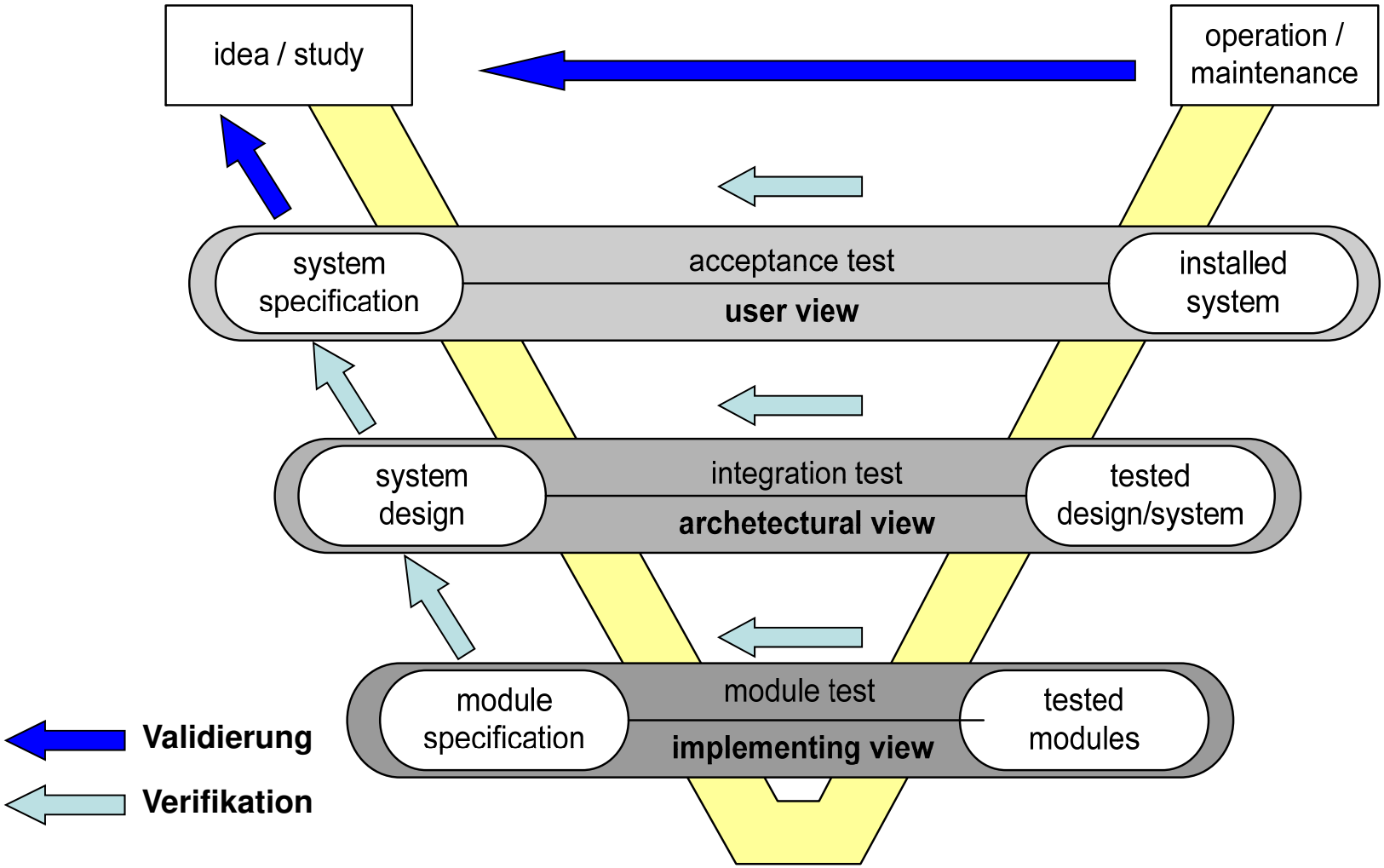
- Qualität ist die Eignung zur Erfüllung vordefinierter Anforderungen.
  - Beispiele für Qualitätsfaktoren (IEEE):
    - Korrektheit, Effizienz, Verwendbarkeit, Testbarkeit, Wartbarkeit
  - Qualität ist keine Eigenschaft, die später hinzugefügt werden kann.
  - Qualität muss während der Entwicklung gesichert werden.

- Qualitative hochwertige Software-Produkte sind
  - termingerecht und im Rahmen des Budgets erstellt.
  - für den Anwender verwendbar.
  - für den Professionisten verständlich und änderbar.
  - für den Betreiber effizient und administrierbar.

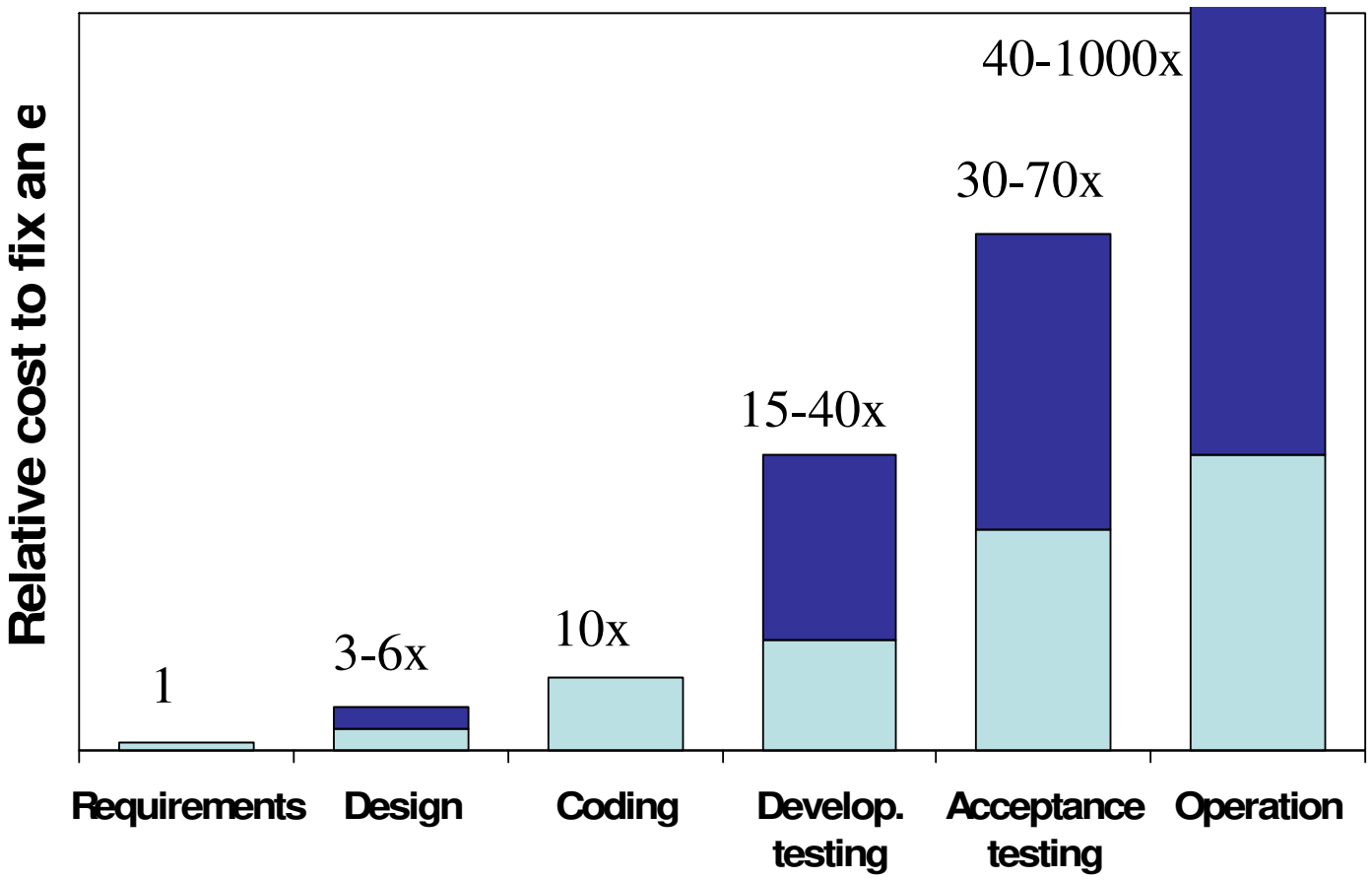
- Qualitätssicherung (QS) besteht in der Durchführung von Verifikation und Validierung in jeder Phase der Software-Herstellung

- Für die Umsetzung in SEPM sind daher notwendig:
  - Testbare Anforderungen
  - Verfolgbare Entwicklung der Anforderungen (mit Modellen) in testbare Produkte.

Testen im V-Modell



Software Development Rework Effort



The cost of fixing errors escalates as we move the project towards field use.

From an analysis of 63 projects cited in Boehm Barry, „Software Engineering Economics“, Prentice-Hall, 1981

Typische Fehler im Software Engineering SEPM PR Projekte

- Datenmodellierung, Reports
  - Schlüsselwerte nicht eindeutig, Fehlende Attribute
  - Fehler bei Beziehungen, z.B. 1:1 Relation im ER sehr selten
- Entwurf: Ergebniswerte von Methoden
  - Typkonversion falsch (z.B. Integer/Real)
- Initialisierung von Variablen
  - Fehlende Zuweisung: nicht eindeutiger Wert
  - Keyword static falsch verwendet: Klasse/Objekt (UML Rollen)
- Bedingungen
  - Logische Fehler; falsche Ausdrücke (und/oder)
- Schleifen
  - Anzahl Durchläufe nicht korrekt; z.B. Endlosschleifen
- Testfälle
  - Unzureichende Überdeckung (Auslassen von Anwendungsfällen, Programmteilen, GUI-Elementen, Datenzuständen)
  - Überflüssige Testfälle (gleiche Äquivalenzklasse)

Zusammenfassung: QS in SE-Projekten

- Für die Umsetzung von Qualität und QS in SEPM sind daher notwendig:
  - Testbare Anforderungen (Definition der Funktionen und Qualitäten)
  - Verfolgbare Entwicklung der Anforderungen (mit Modellen) in testbare Produkte.
- Typische SE-Modelle und deren Verwendung als Basis für die QS
  - Anwendungsszenarien (etwa Anwendungsfälle)
  - Datenmodelle (Relationenmodell(ER), Domänenmodell)
  - Datenflussmodelle (UML, IDEF0)
  - Kontrollflussmodelle (UML, Pseudocode)
  - Zustandsdiagramme, Sequenzdiagramme (UML)
  - Testfallspezifikationen
- Test-Driven Development
  - Herstellen von Testfällen vor der Implementierung
  - Ablauffähige Testfälle für automatischen Re-Test neuer Code-Teile
  - Frühe Review missionskritischer Akzeptanztestfälle

Statische Methoden der QS in SEPM

Statische Methoden der QS in SEPM

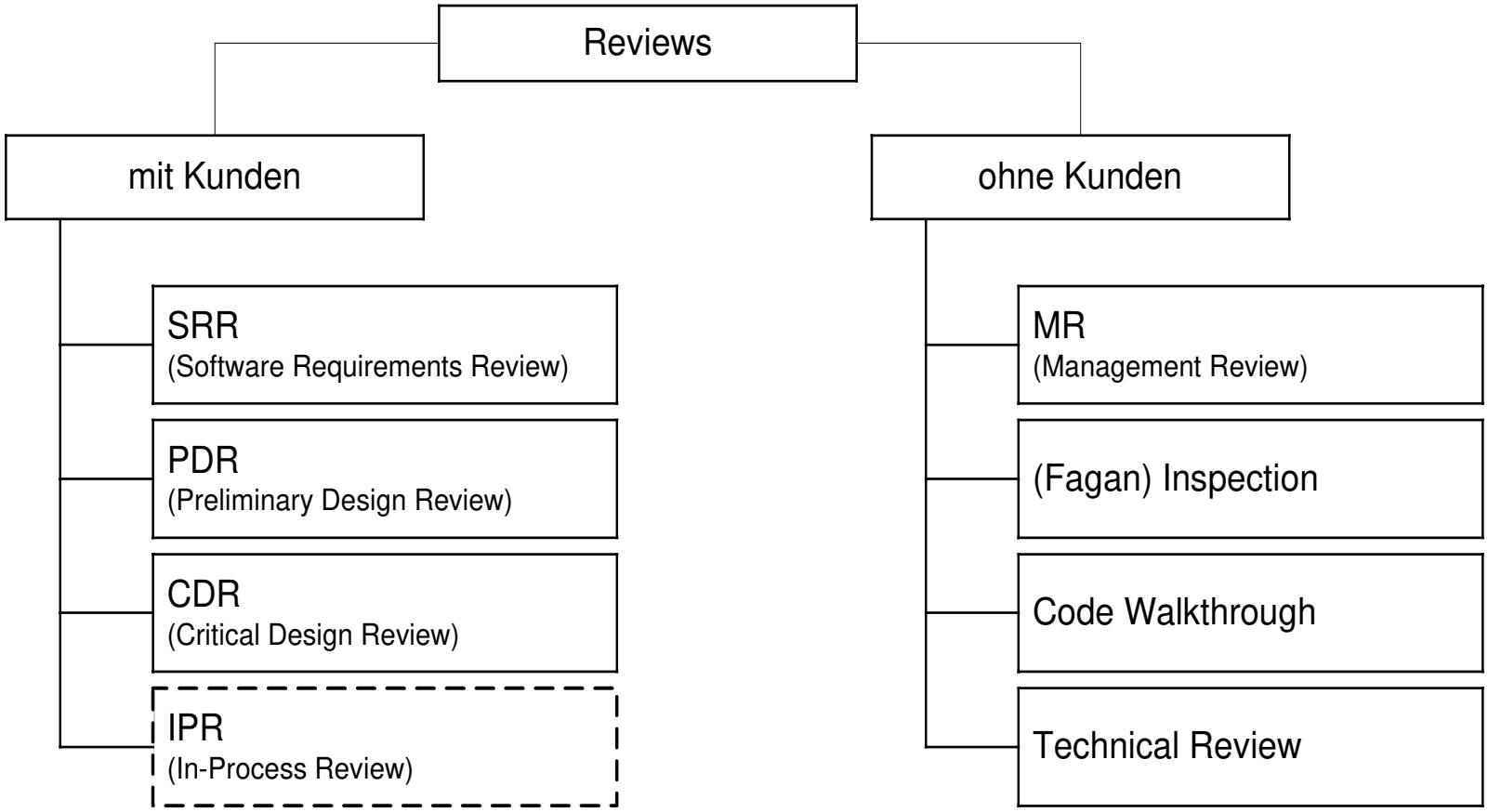
- Definition und Arten von Reviews
  - Rollen bei Reviews
  - Ablauf von Reviews
  - Richtlinien für technische Reviews
- Anwendungsbeispiel “Restaurant”
  - Überprüfung der Anforderungen
  - Aus den Anforderungen abgeleitete Modelle
    - Datenmodelle: Entity Relationship
    - Aktivitätsdiagramme: Datenfluss und Kontrollfluss
    - Zustandsdiagramm: State Machine Diagram
  - Reviews von Anforderungen mit ER/UML/IDEF0 Diagrammen

Reviews

Definition

- „Ein Review ist ein [..] formal geplanter und strukturierter Analyse- und Bewertungsprozess, in dem Projektergebnisse einem Team von Gutachtern präsentiert und von diesen kommentiert oder genehmigt werden.“
  [IEEE Std 610, Wallmüller 2001]
- Reviews dienen vor allem zur qualitativen Beurteilung von Produkten und Prozessen,
  die quantitativ nur schwer oder gar nicht beurteilt werden können (z.B. Modelle, Dokumente)
- Zentral ist die Beurteilung des Produktes und nicht des Autors!
- Unterschiedliche Ausprägungen von Reviews zielen auf unterschiedliche Ziele ab.
- In einen Reviewprozess sind definierte Rollen mit zugeordneten Aufgaben involviert.

Arten von Reviews [Thaller, 2000]



Reviews:
... und verwandte Tätigkeiten

- Inspektion
  - Ziel: Behebung von konkreten Mängeln
  - Bed.: Leser ist nicht Autor
  - Unb.: Alternativen, Stil
- Walkthrough
  - Ziel: Behebung von konkreten Mängeln
  - Bed.: Autor ist Leser & Moderator
  - Bed.: Alternativen, Stil

- Audit
  - Ziel: Überblickende Kontrolle
  - Bed.: Planung durch externe Personen
  - Bed.: Projektmitarbeiter nur als Information

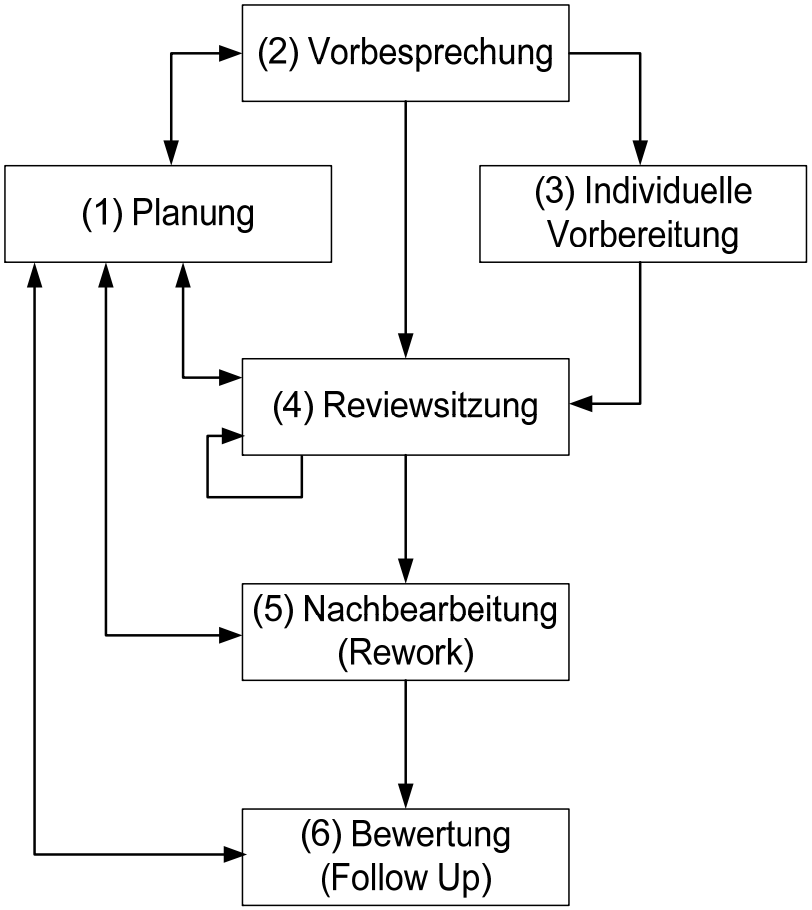
Rollen bei Reviews

- Die typische Größe eines Reviewteams liegt (abhängig von Projektart, -größe, usw.) im Bereich 3 bis 6 Personen, die sich auf folgende Rollen aufteilen:

  - Moderator („keeper of the process“)
    Leiter des Reviews
  - Leser („keeper of focus and pace“)
  - Gutachter („Reviewer“)
    Kommentierung des Reviewobjektes.
  - Schreiber („preserver of knowledge“)
    Protokollschreiber verfasst Protokoll
    (Notizen während der Reviewsitzung)

  - Autor („author“)
    Klärung von offenen Fragen.
    Keine Kommentierung und Rechtfertigung der Lösungen.

Ablauf einer Review



- Planung: Objekt, Prüfziele, Auslösekriterien (Einstiegskriterien), Teilnehmer, Ort, Zeit.
- Vorbesprechung: Vorstellung des Prüfobjekts bei komplexen und neuen Produkten.
- Intensive Einzeldurcharbeitung
- Durchführung: Gemeinsames Lesen, Aufzeichnung von Mängeln; während des Reviews sollen Mängel entdeckt, nicht korrigiert werden.
- In der Nachbearbeitung werden dokumentierte Mängel korrigiert und in der Bewertung überprüft.
- Berichterstattung.
- Wiederholungen von Reviews sind möglich.

- Checklisten unterstützen Reviews.
- Typische Dauer: 2h

Durchführung von Reviews

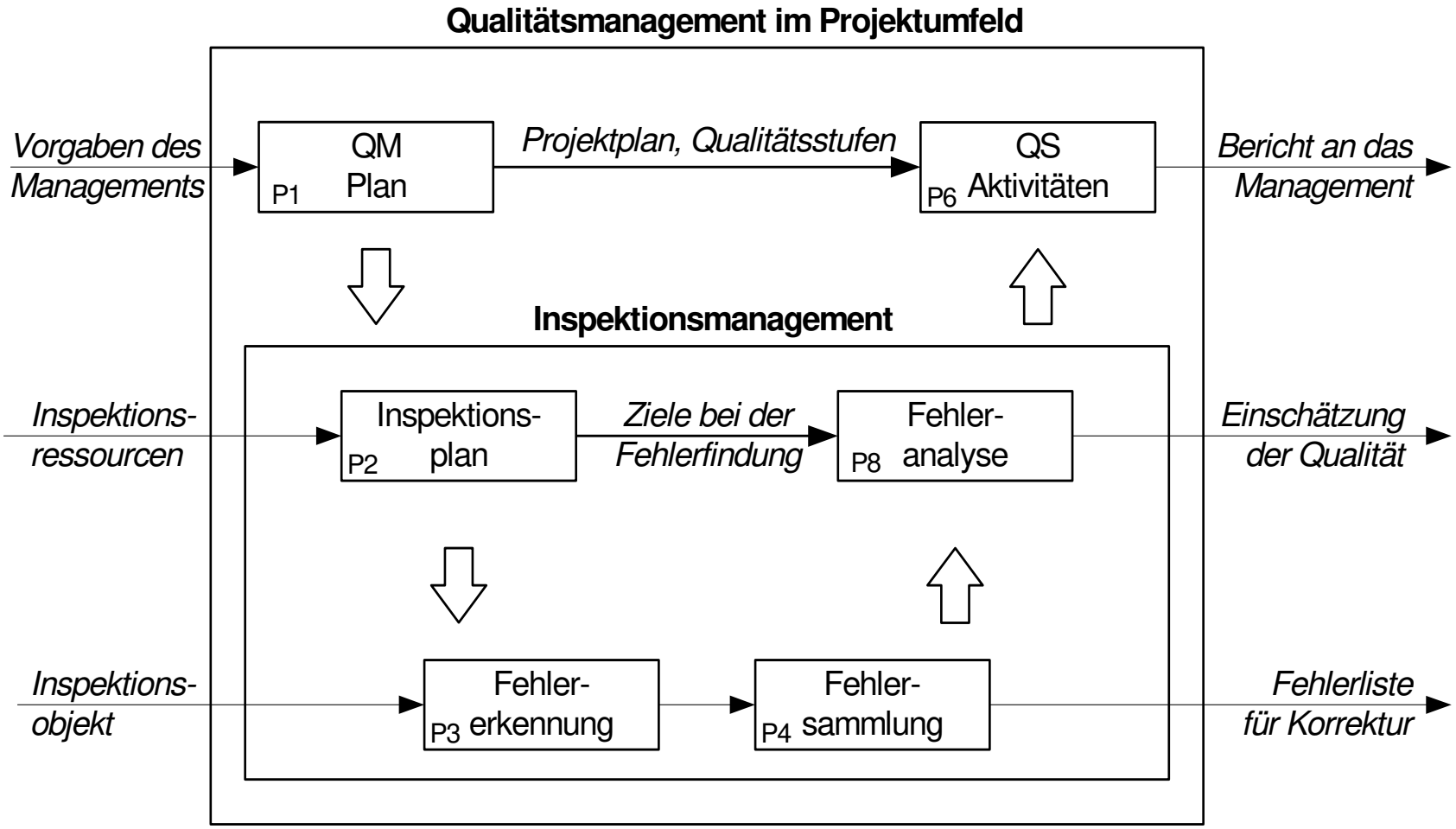
Um brauchbare Ergebnisse zu erzielen, ist der Ablauf einer Review klar definiert und umfasst folgende Phasen:

- Planung („planning phase“)
- Initialisierung („initialisation phase“)
- Vorbereitung („preparation phase“)
- Reviewsitzung („review“)
- Sammlung im Team („meeting“)
- Reviewbericht („reporting“)
- Nacharbeit („re-work“)

Richtlinien für technische Reviews

1. Das Produkt reviewen, nicht den Autor.
2. Verwenden Sie einen Arbeitsplan.
3. Diskussionen sachlich und kurz halten.
4. Problembereiche identifizieren, aber nicht jedes Problem gleich lösen.
5. Schriftliche Aufzeichnungen führen.
6. Anzahl der Teilnehmer begrenzen; gute Vorbereitung aller Teilnehmer.
7. Für jedes Produkt eine passende Checkliste verwenden.
8. Ausreichende Ressourcen und Zeitbudget zur Verfügung stellen.
9. Vorab Training für alle Reviewer.
10. Reviews im Nachhinein beurteilen für Verbesserung der Reviews.

Inspektionsplanung und Kontrolle



Qualitätsdefinition: Beispiel "Restaurant"

- Mögliche Formen der Anforderungen
  - Texte, Modelle, Referenzdokumente (z.B. Standards)
  - Liste mit Aussagen
  - Anwendungsszenarien
  - Text Beschreibung enthält Akteure, Entitäten und Aktivitäten/Operationen.
- Anforderungen beschreiben Funktionen und Qualitäten
  - Überprüfung der operativen Definition der Funktionen und Qualitäten.
- Auflisten von Kandidaten für Funktionen und Qualitäten
- Falls eine Funktion oder Qualität nicht vollständig testbar erscheint, Vorschlag für operativ testbare Variante machen.

Überprüfung der Anforderungen

- Kriterien für brauchbare Anforderungen
  - Notwendige Kriterien
    - Typ (z.B. FURPS)
    - Testbarkeit: Erfüllung einer Anforderung durch ein System muß eindeutig testbar sein (Messvorschrift: was ist wie zu messen? Z.B. Dauer eines Ablaufs).
    - Widerspruchsfrei unter einander
  - Erstrebenswerte Kriterien
    - Klare, knappe Beschreibung
    - Umsetzbarkeit
    - Konsistent mit bekannten Interessen der Stakeholder (richtig, „komplett“, relevant)
- Wichtig: Darstellung, die eine systematische Bearbeitung unterstützt
  - Vor der weiteren Verwendung (z.B. von Prosatext) kann eine Transformation die Verwendung der Anforderungen erleichtern; z.B. Herstellung einer Liste von Anforderungen.

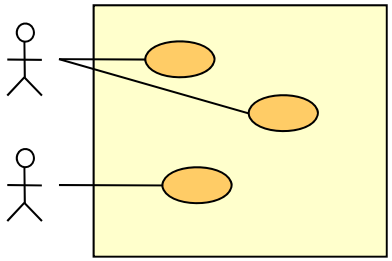
Von Anforderungen abgeleitete Modelle

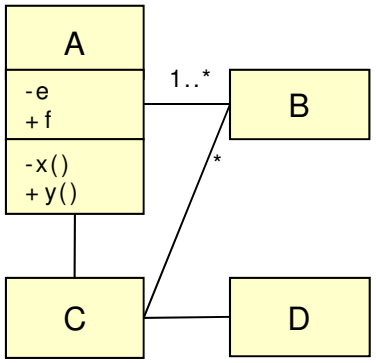
- Anforderungen ➔ Modelle ➔ Datenbank, Business Logic, GUI
  - Datenbanken: Datenmodellierung, Entity Relationship
  - Business Logic: Teilsysteme, Daten- und Kontrollfluss
  - Schnittstellen zwischen Teilsystemen und Benutzern

- Anforderungen ➔ Qualitätsdefinition; Modelle
  -> Qualitätssicherung: Review, Testen
  - Anforderungen beschreiben Funktionen und Qualitäten
  - Zur operativen Definition sind die Qualitäten testbar zu beschreiben.

- Fokus: Test-Driven Development (TDD) Testfälle
  - Herstellen von ablauffähigen Testfällen **vor** der Implementierung
  - TDD überprüft die Spezifikation vor dem detaillierten Entwurf

Überprüfung statischer Aspekte in UML





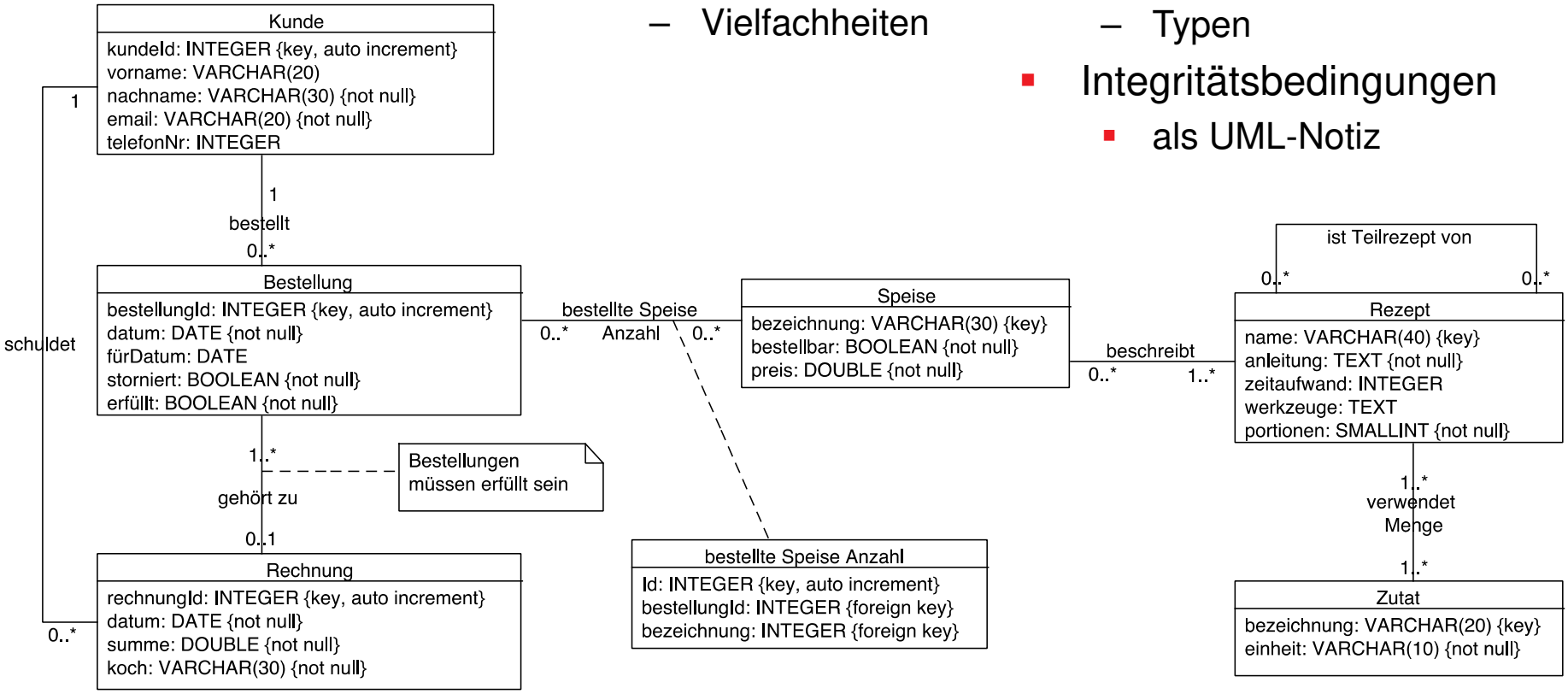
- **Anwendungsfälle**
  - die Benutzer und Nützlichkeit eines Systems
  - definiert durch Akteure und Hauptszenario
- **Klassen**
  - Menge gleichartiger Objekte
  - definiert durch Attribute, Operationen, und Beziehungen zu anderen Klassen
  - auch ein Datenbankschema kann in UML Klassendiagrammnotation modelliert werden.

Überprüfung statischer Aspekte

- Anwendungsfälle
  - Software Requirement Review
  - Jeder Anwendungsfall muss in den Anforderungen beschrieben werden
  - Akteure + Operationen in den Anforderungen führen zu den Anwendungsfällen
- Klassendiagramme
  - Preliminary Design Review
  - Die Daten welche in den Anforderungen zu finden sind werden von Model-Klassen gekapselt
  - Operationen + Entities in den Anforderungen führen zu Klassen
  - Aber auch: GUI Klassen, Support-Klassen für Internationalisierung, ...

Beispiel für UML Entity Relationship (Restaurant, unvollständig)

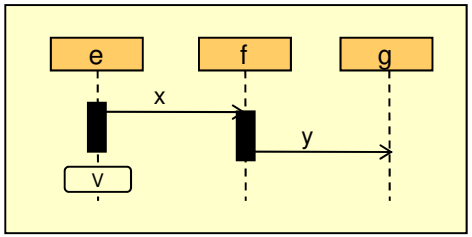
- Entitäten
  - Eine Tabelle pro Entität
- Beziehungen
  - Vielfachheiten
- Attribute
  - Schlüssel
    - Eigenschlüssel
    - Fremdschlüssel
  - Typen
- Integritätsbedingungen
  - als UML-Notiz

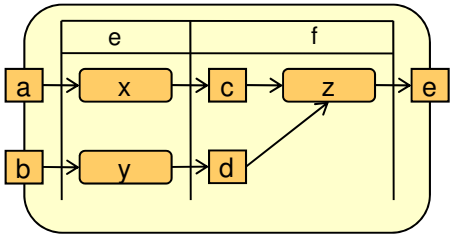


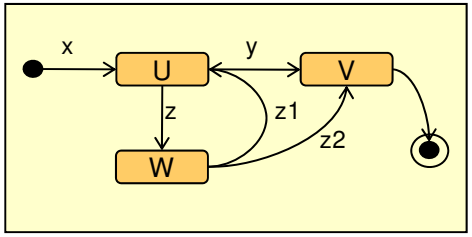
Überprüfung Datenbankrelationen
Review mit ER und Anforderungen

- Anforderungen strukturiert darstellen, z.B. als Liste.
- Anforderungen mit ER-Model reviewen und konsistent machen.
- ER-Modell systematisch durchgehen
  - Datenbanktabellen zu Entitäten und Beziehungen finden (markieren).
  - Schlüsselattribute auf Eindeutigkeit überprüfen
  - Attribute: Typen und Optionen überprüfen
    - Sind weitere Entitäten notwendig? (3. Normalform/Relational)
  - Integritätsbedingungen zu Attributen überprüfen bzw. ergänzen
    - Einzelne Attribute
    - Für Anforderungen relevante Bedingungen über mehrere Attribute
- Datenbanktabellen systematisch durchgehen.
  - Nicht markierte Elemente auf Sinnhaftigkeit überprüfen

Überprüfung dynamischer Aspekte in UML







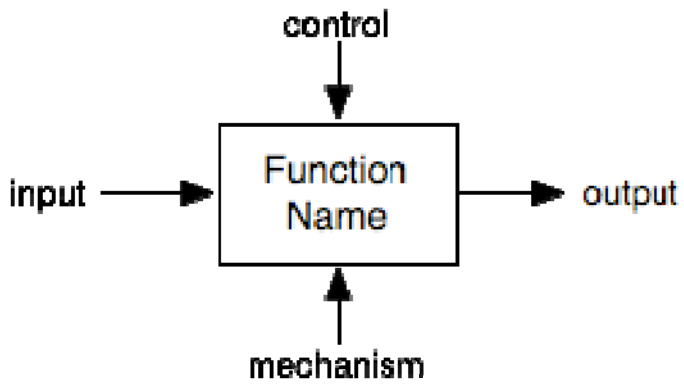
- **Sequenz**
  - ge- bzw. verbotene Interaktionsszenarien
  - definiert durch Folgen von Nachrichtenaustauschen zwischen Interaktionspartnern
- **Aktivität**
  - in sich geschlossenes, pro-aktives Verhalten
  - definiert durch Aktionen, Kontroll- und Datenfluss sowie evtl. Zuständigkeiten
- **Zustandsautomat**
  - reaktives Verhalten
  - definiert durch Zustände, Ereignisse und Transitionen

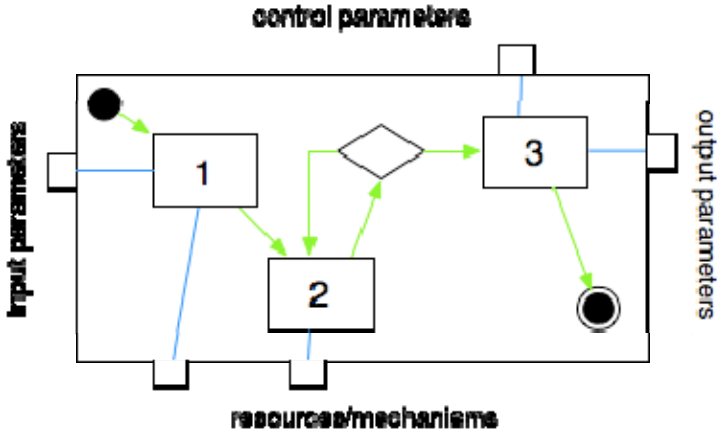
Aus: OO Modellierung, BIG, 2006

Aktivitätsdiagramm: Überprüfbare Aspekte von Daten- und Kontrollflüssen

- Basierend auf Anwendungsszenario
  - Rollen und Aktivitäten
- Detailgrad von Prozessen
  - System und Teilsysteme: Schnittstellen
  - Teilsystem und Aktionen: Detaillierter Kontrollfluss
- Ableitung von Prozessen
  - Datenfluss
    - Eingangs- und Ausgangsparameter
  - Kontrollfluss: Logische Darstellung eines Ablaufes
    - Entscheidungen und Schleifen

Überprüfung von Prozess- und Datenflussanalyse (IDEFØ)



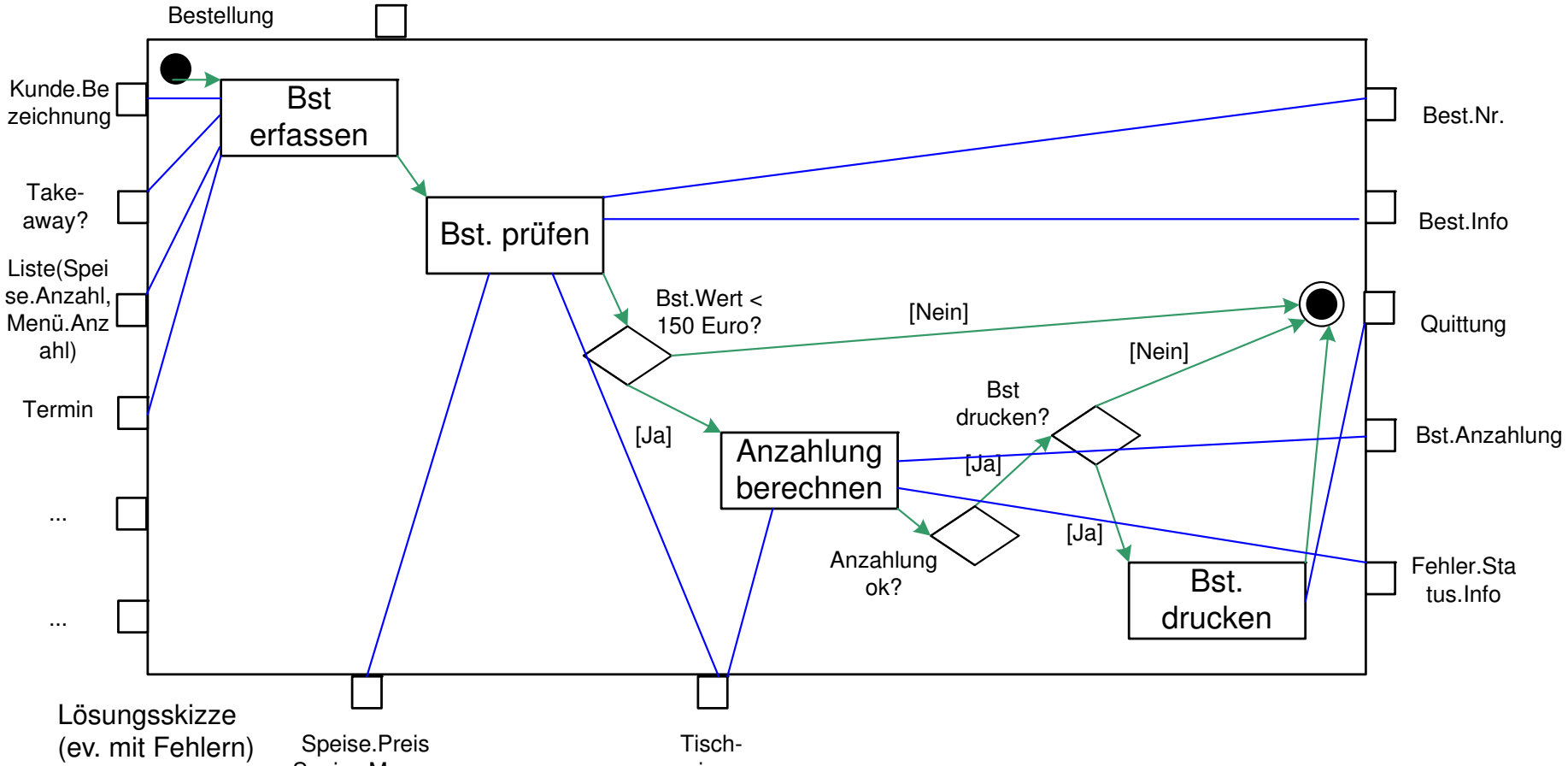


- Top-Down Modell
- Modellierung des Problembereichs
- Hilft neue Aufgabenbereiche zu erfassen und zu strukturieren

- Datenfluss (Daten-Token)
- Kontrollfluss (Kontroll-Token)
  - ergänzt um Start- und Endpunkt

Beispiel: Restaurant.Bestellung

- Datenfluss (Daten-Token)
- Ergänzt um Kontrollfluss (Kontroll-Token), Start, Ende;
- Entscheidungen
- Geltungsbereich von Daten-Token (lokal, global)
- Initialisierung aller Variablen
- Konsistenter Abschluss



Überprüfung Kontrollflussdiagramm
Review mit Anforderungen & Datenfluss

- Anforderungen strukturiert darstellen, z.B. als Liste.
- Anforderungen mit Datenfluss-Modell reviewen und konsistent machen.
  - Im Datenfluss-Modell Elemente markieren, die helfen sollen, die aktuellen Anforderungen zu erfüllen.
- Kontrollfluss-Modell systematisch durchgehen
  - Startpunkt(e) identifizieren
  - Entscheidungspunkte überprüfen: logische Bedingungen zu Entscheidungsvarianten
  - Endpunkt(e) identifizieren
- Überprüfung
  - Alle Alternativen des Anwendungsszenarios (Use Case) zur Aktivität sind abgebildet
  - An Endpunkten müssen Ausgangsparameter gültige Werte haben (entsprechend der Spezifikation der Aktivität)
  - Alle Knoten und Kanten können einer Alternative des Anwendungsszenarios zugeordnete werden.

Zustandsdiagramm
Einführung

- Ein Zustandsdiagramm (State Machine Diagram) beschreibt die möglichen Folgen von Zuständen eines Modell-elements, i.A. eines Objekts einer bestimmten Klasse
  - Während seines Lebenslaufs (Erzeugung bis Destruktion)
  - Während der Ausführung einer Operation oder Interaktion

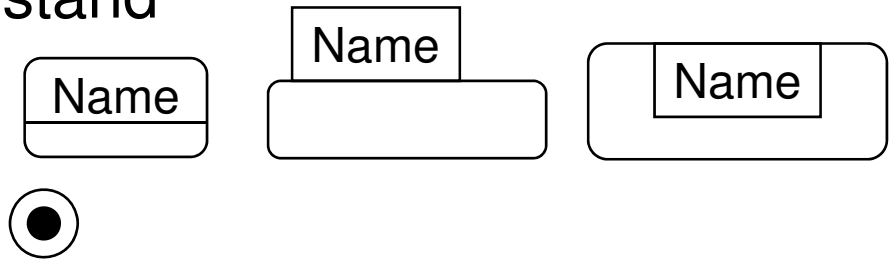
- Modelliert werden
  - Die Zustände, in denen sich die Objekte einer Klasse befinden können
  - Die möglichen Zustandsübergänge (Transitionen) von einem Zustand zum anderen
  - Die Ereignisse, die Transitionen auslösen
  - Aktivitäten, die in Zuständen bzw. im Zuge von Transitionen ausgeführt werden

Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

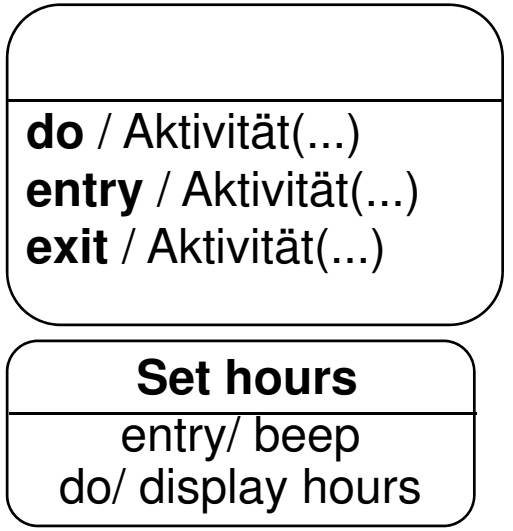
Basiskonzepte – Zustand

- Zustand (state)
  - Zustand i.e.S.
  - Endzustand



- Pseudozustände (weil transient)
  - Initialzustand
  - History-Zustand, Synch-Zustand, Gabelung, Vereinigung, etc.

- Aktivität innerhalb eines Zustands
  - entry / *aktivität*
    Aktivität wird beim Eingang in den Zustand ausgeführt
  - exit / *aktivität*
    Aktivität wird beim Verlassen des Zustands ausgeführt
  - do / *aktivität*
    Aktivität wird ausgeführt, Parameter sind erlaubt
  - *event* / *aktivität*
    Aktivität behandelt Ereignis innerhalb des Zustands



Aus: OO Modellierung, BIG, 2006

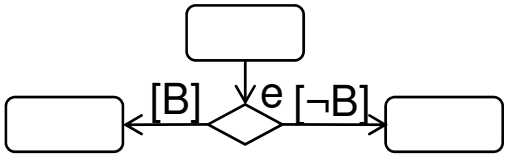
Zustandsdiagramm

Basiskonzepte – Zustandsübergang

- Ein Zustandsübergang (state transition) erfolgt, wenn
  - das Ereignis eintritt – eine evt. noch andauernde Aktivität im Vorzustand wird unterbrochen!
  - und die Bedingung (guard) erfüllt ist – bei Nicht-Erfüllung geht das nicht »konsumierte« Ereignis verloren
- Durch entsprechende Bedingungen können Entscheidungsbäume modelliert werden
- Standardannahmen
  - Fehlendes Ereignis entspricht dem Ereignis »Aktivität ist abgeschlossen«
  - Fehlende Bedingung entspricht der Bedingung [true]
- Aktionen auf Zustandsübergängen möglich
  - Spezielle Aktion: Nachricht an anderes Objekt senden

    send empfänger.nachricht()
  - Beispiel:

    right-mouse-down (loc) [loc in window]
    / obj:= pick-obj (loc); send obj.highlight()

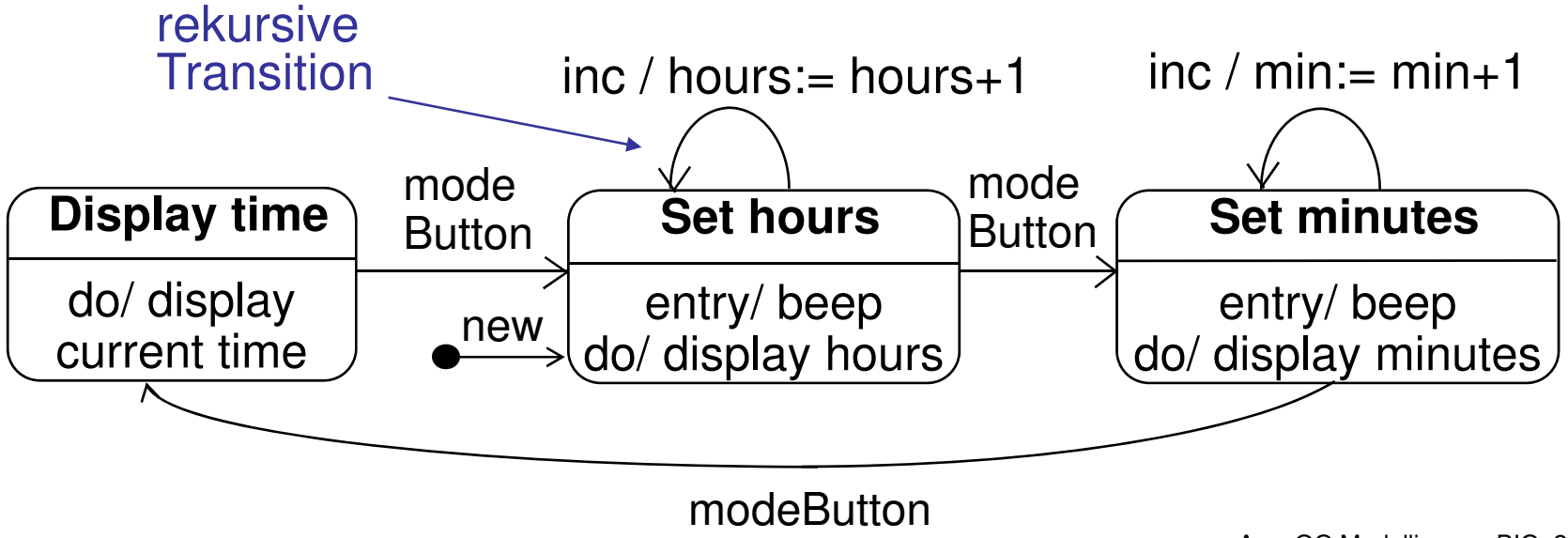


Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Basiskonzepte – Beispiel: Klasse DigitalWatch

DigitalWatch
min : Integer = 0 hours : Integer = 0
modeButton() inc()

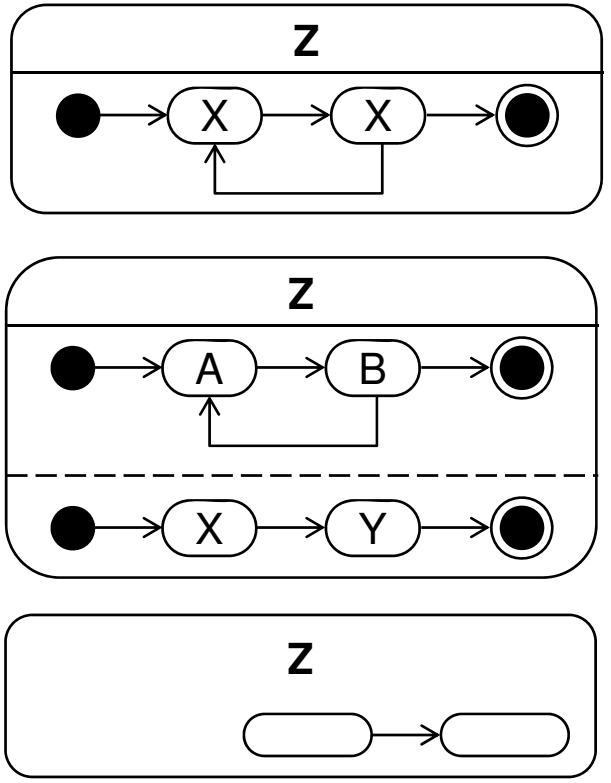


Aus: OO Modellierung, BIG, 2006

Zustandsdiagramm

Strukturierung – ODER- vs. UND-Verfeinerung

- Verfeinerung eines komplexen Zustands (composite state) — geschachteltes Zustandsdiagramm
- ODER-Verfeinerung
  - Disjunkte Sub-Zustände, d.h. genau ein Subzustand ist aktiv, wenn der komplexe Zustand aktiv ist
- UND-Verfeinerung
  - Nebenläufige Sub-Zustände, d.h. alle Subzustände sind aktiv, wenn der Superzustand aktiv ist
  - Die Subzustände werden i.A. ihrerseits Oder-verfeinert
- Hinweis auf ausgeblendete Verfeinerung
  - Diese kann an anderer Stelle dargestellt werden



Aus: OO Modellierung, BIG, 2006

Ablauf Zustandsübergangsdiagramm

- Entitäten und deren Zustände identifizieren
- Parallele Zustände aufteilen

- Für jede Entität
  - Initialzustand bestimmen
- Zustandsübergänge einzeichnen
    - Ereignis(se) bestimmen, die den Übergang anstoßen
  - Wo sinnvoll: Bedingungen für Übergänge bestimmen

- Überprüfung
  - Abläufe des Anwendungsszenarios durchspielen
  - Testfälle durchspielen
  - Überdeckung: Alle Knoten und Kanten abdecken.

Überprüfung Zustandsübergangsdiagramm Review mit Anforderungen

- Anforderungen strukturiert darstellen, z.B. als Liste.
- Anforderungen systematisch durchgehen
  - Im Zustandsübergang-Modell Elemente markieren, die helfen sollen, die aktuelle Anforderung zu erfüllen.
    - Zustände
    - Übergänge

- Zustandsübergang-Modell systematisch durchgehen
  - Nicht markierte Elemente auf Sinnhaftigkeit überprüfen
  - Startpunkt(e) identifizieren
  - Entscheidungspunkte überprüfen: logische Bedingungen zu Entscheidungsvarianten
  - Endpunkt(e) identifizieren

- Überprüfung
  - Alle Alternativen des Anwendungsszenarios (Use Case) zur Aktivität sind abgebildet.
  - Alle Knoten und Kanten können einer Alternative des Anwendungsszenarios zugeordnete werden.

Testansätze, Test-Driven Development

Testansätze, Test-Driven Development

- Testen in der Qualitätssicherung und SE Prozessen
- Teststufen und Testarten
  - Unit, Modul, System Stufen
  - Black-Box, White-Box Testarten
- JUnit, Test-Driven Development
- Beurteilung der Güte von Testfällen
- Anwendungsbeispiel “Restaurant”
  - Ableitung von Testfällen aus Anforderungen, Modellen und Entwürfen

Testen in der Qualitätssicherung

Verschiedene Teststufen:

- Unit-Test
- Modultest (Integrationstest)
- Systemtest
- Alphatest
- Betatest
- ...

Abhängig von der Testebene und dem aktuellen Anwendungsfall verfolgen Tests unterschiedliche Ziele:

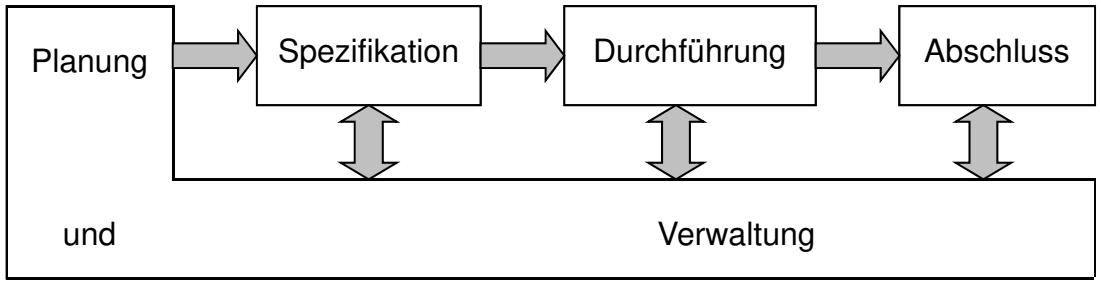
- Funktion (Funktionstest)
- Last (Lasttest)

Je nachdem, ob beim Testen die innere Struktur der Software berücksichtigt wird oder nicht, spricht man von White Box oder Black Box testen

Testen im SE Prozess

- Was tun Sie, wenn Sie die Aufgabe bekommen, ein Programm zu testen?
  - *„Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.“* [Pol et al., 2000]

- Phasen in der Softwareentwicklung
  - Planung und Verwaltung
  - Spezifikation
  - Durchführung
  - Abschluss



Rollen im Team

- Alle Teammitglieder sind auch Testverantwortliche.
  - Test sind Dokumentation ("living documentation")
  - Artefakt Testplan zeigt wie Testfälle implementiert werden.
  - Eine gute Abstimmung der Rollen und Fähigkeiten im Team wird benötigt.

- Rollenbeitrag zur Qualität der Tests
  - Technische Architekten:
    - Testbarer Code („kleine", eigenständige Programme)
    - Klassendiagramme, Systemarchitektur
  - Tester:
    - In den Anfangsphasen, Black-Box Tests der Interfaces (Modul-/Integrationstests welche Anwendungsszenarien abdecken)
    - Auf Verwendung von Test-Driven Development achten. (Unit-Tests)
    - System-Tests durchführen

Ablauf Testfälle bestimmen

- Ziel aus Testplan bestimmen
  - Z.B. Normal-, Sonder- und Fehlerfälle in einem Anwendungsszenario abdecken
  - Z.B. Alle Knoten und Kanten in einem Kontrollflussgraphen abdecken
- Fälle bestimmen
  - Eingabeparameter
  - Entscheidungen im Ablauf (Wahl der Entscheidung festhalten)
- Erwartetes Ergebnis bestimmen

Äquivalenzklassenzerlegung

- Zerlegung einer Menge von Daten (Input oder Output) in Untermengen (Klassen), die äquivalente Ergebnisse oder Auswirkungen produzieren.
- Jede Klasse(nkombination) sollte mindestens einmal getestet werden.
- Dadurch sinkt die Anzahl der Testfälle und somit der Testaufwand.

Vorgehensweise:

- Finde zu testende Funktionen.
- Finde Eingabe-Vektor (A, B, C, ...) (= Faktoren) je Funktion.
  - Explizite: Parameter.
  - Implizite: globale Variablen, Systemzustand (Objekte, Datenbank), …
- Finde für jeden Faktor seine un-/gültigen Äquivalenzklassen (Klassen A1, A2, ...; B1, B2, ...; ...) und wähle je einen Wert aus jeder Klasse.
- Wahl der Kombinationen: (A1, B1), (A1, B2), ... bestimmt Intensität.

Äquivalenzklassenzerlegung

- Anforderung: 18 < Alter ≤ 65 → drei Äquivalenzklassen
  - A1: Alter ≤ 18 (ungültig)
  - A2: 18 < Alter ≤ 65 (gültig)
  - A3: Alter > 65 (ungültig)

  → Auswahl eines Repräsentanten einer Äquivalenzklasse,

  → Drei Testfälle: x ∈ A1, z.B. 10, x ∈ A2, z.B. 35, x ∈ A3, z.B. 70

Grenzwertanalyse

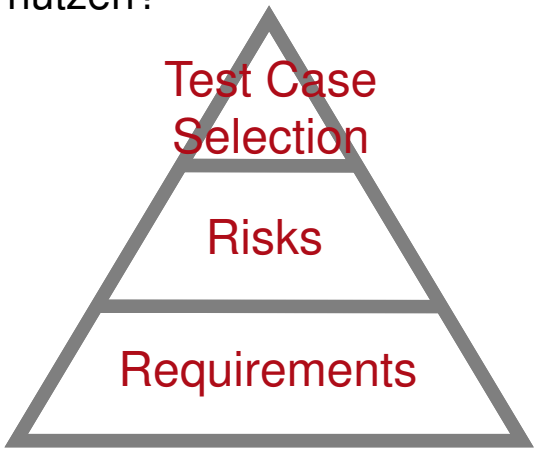
- Erweiterung der Äquivalenzklassenzerlegung für bessere Überdeckung.
- Grenzwerte werden als Klassenrepräsentanten gewählt
  → je Klassengrenze ein Testfall.

- Beispiel: Anforderung: 18 < Alter ≤ 65
  - 18 (ungültig), 19 (gültig), 65 (gültig) and 66 (ungültig)
  - Ev. mehrere Testfälle je Klassengrenze: Alter ≤ 18
  - 17 (gültig), 18 (gültig) and 19 (ungültig)
    Bedenke Tippfehler: Alter = 18!

Wertbeitrag von Software Testen

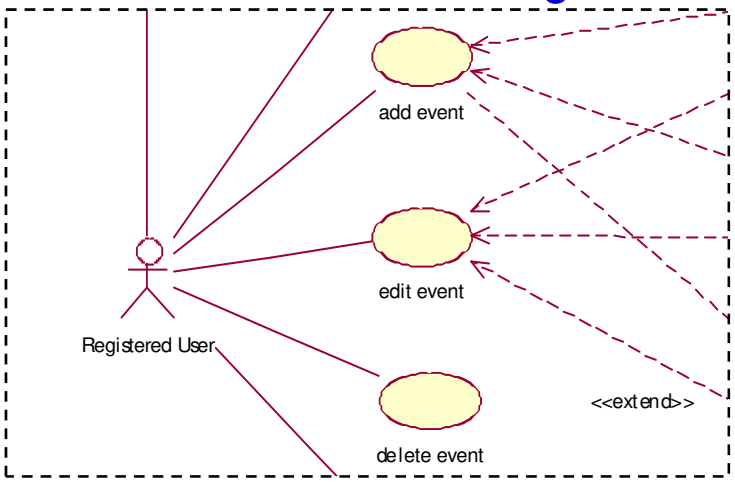
- Zentrale Fragestellung: Ist jeder Testfall, jeder Fehler, jede Anforderung gleich viel „wert“?
- Beispiel: 10 unterschiedliche Bezahlverfahren einer Online-Plattform, wobei 80% des Umsatzes immer über dasselbe Verfahren abgewickelt wird.
- Die Antwort: NEIN => Value-Based Testing.

- Requirements-Based testing
  → Welche Anforderungen bringen dem Kunden den meisten nutzen?
- Risk-Based testing
  → Welche Risiken gefährden diesen Nutzen?
- Test case selection techniques
  → Welche Testfälle addressieren dieses Risiko?

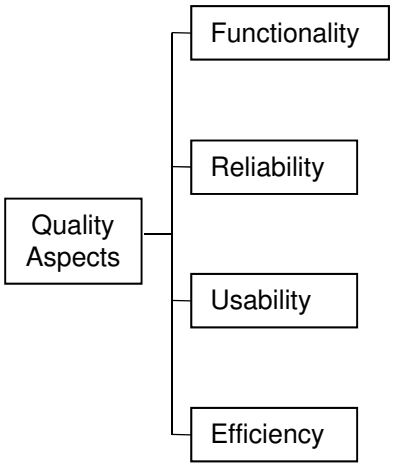


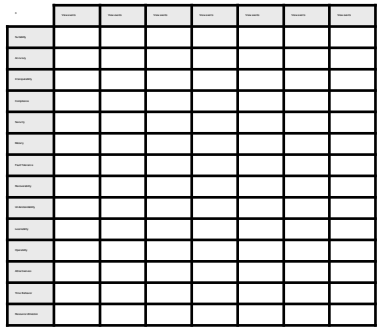
Beispiel: Risk-Based Testing (1)

Features / Anwendungsfälle



Quality Criteria





Risikoeinschätzung:
A … Critical,
B ... Important,
C ... Less important

Security aspect \ Feature	View events	View details	Add event	Edit event	Attach document	Set properties	Delete event
General identity	B	B	C	C	C	C	C
Message content authenticity	B	B	C	C	C	C	C
Message content origin	C	C	B	B	B	A	B
Integrity	C	C	A	A	A	A	
Secrecy and privacy	B	B	C	B	C	C	C
Accountability			B	B	B	B	B

Ramler et al, 2006

Beispiel für Testfallbeschreibung "Bestellung von Menü durchführen"

- Beschreibung ist die Komponente im System und eine Funktion (Ablauf aus Anforderungen)
- Vorbedingungen existieren oder eben nicht
- Eingabewerte sind konkrete Parameterbeschreibungen (Äquivalenzklassen)
- Aktionen sind Aktivitäten zur Eingabe der Werte
- Erwartete Ergebnisse sind Zustände und Ausgabeparameter
- Ergebnisse zeigen Abweichungen zum Tatsächlichen Ergebnis.
- OK/NOK zeigt ob der Test erfolgreich durchläuft.

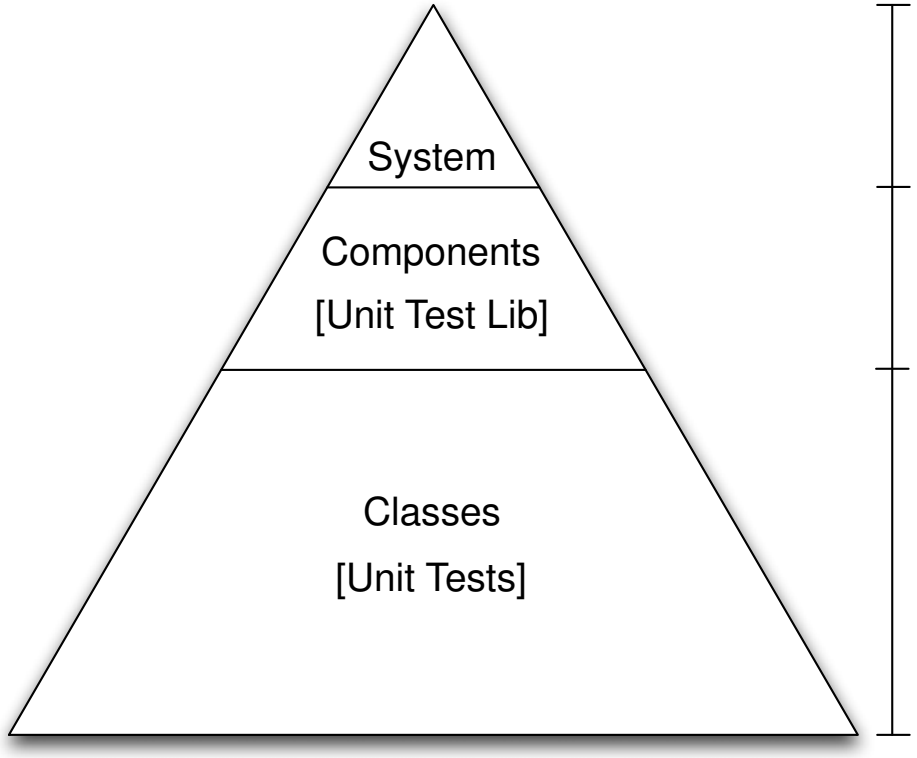
Nr.	Typ	Beschreibung	Vorbedingungen	Eingabewerte	Aktionen	Erwartete Ergebnisse	Ergebnisse	OK / NOK
34	NF	Bestellung von Menü durchführen	(Programm gestartet) (in die Bestellungs-übersicht gewechselt) Min. ein Menü mit preis<250 Euro muss vorhanden sein. Menü muss bestellbar sein.	-Bestellung. take_away -Bestellung. fürDatum -Menu.id -Menü Anzahl = 1	-Neue Bestellung erzeugen. Angabe wann die Bestellung konsumiert wird und ob "take away". -Menü wird aus einer Liste ausgewählt -Anzahl angeben -Bestellung speichern	Bestellung ist in der Datenbank gespeichert und enthält genau ein Menü. Das Attribut Bestellung.storniert ist auf den Booleanwert ‚false' gesetzt, der Primärschlüssel von Bestellung wird von der DB automatisch belegt.	Bestellung ist in der DB gespeichert, jedoch wird bei der Abfrage dessen kein Primärschlüssel mitgeliefert.	NOK

Testfall-Spezifikationsmethoden

- Strukturierte Anweisungen zur Generierung von Testfällen
- Vorteile gegenüber zufälliger Testfalldefinition
  - Höhere Qualität der Testfälle
  - Machen Qualität und Intensität eines Tests transparent
  - Erreichen gewünschte Abdeckung für jeden Teil der Applikation
  - Effektiver für gegebene Fehlertypen
  - Tests werden nachvollziehbar
  - Testprozess wird unabhängig von Personen
  - Testfall-Spezifikationen werden übertragbar und aktualisierbar
  - Testprozess wird besser plan- und steuerbar
- „Nachteile“
  - Fundiertes Wissen der Tester notwendig

Testen - Gesamtsystem

- Die Fläche des Pyramidenabschnitts ist proportional zu der Anzahl der jeweiligen Tests welche bei Auslieferung vorhanden sein sollten.



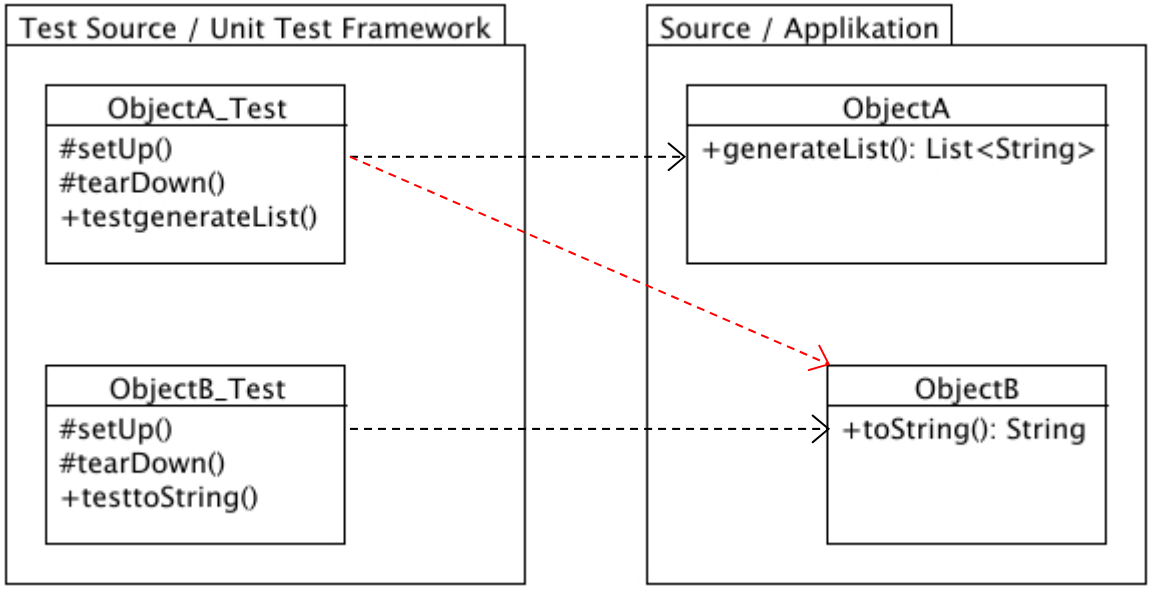
- System Tests
- Integrations/Modul Tests
- Unit Tests

Unit Tests

- Ziel: Dokumentation von Klassen
- Units werden jeweils einzeln gegen ihre Spezifikation getestet: Übergabe von Daten, Prüfung des Outputs.
- Unittests erlauben eine genaue Lokalisierung und frühe Erkennung von Implementierungsfehlern.
- Unittests können bei größeren Systemen sehr aufwendig werden; daher ist bei größeren Tests besonderer Wert auf strukturiertes Vorgehen und Dokumentation des tatsächlichen Vorgehens zu legen.
- Automatisierung essentiell da häufige Regressiontest nach Änderungen notwendig

Unit Testing

- Bei einem Unit-Test in einer OO Sprache liegt der Fokus auf einer Klasse oder eine Methode (Unit == Klasse)
- Alle benötigten Daten passen in den Arbeitsspeicher
- Unit-Tests müssen schnell durchlaufen (Regression-Library)



Module

- Ein Modul hat folgende Eigenschaften
  - Es ist das Werk eines Entwicklers
  - Es hat eine dokumentierte Spezifikation (API)
  - Es ist ein sichtbares, identifizierbares Produkt mit expliziter Integration ins Gesamtsystem (z.B. mit Interfaces)
  - Es kann kompiliert/interpretiert und getrennt von anderen Modulen getestet werden (z.B. mit Mocking)
  - Es ist notwendig

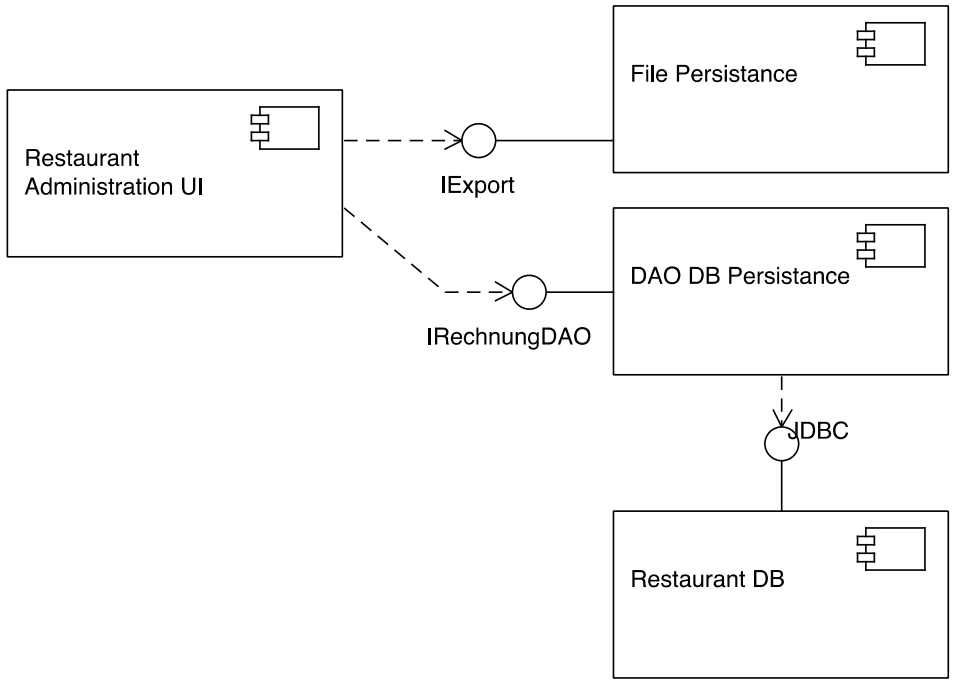
Modultests

- Ziel: Aufspürung von Fehlern in der Implementierung
- Module werden jeweils einzeln gegen ihre Spezifikation getestet: Übergabe von Daten, Prüfung des Outputs
- Modultests erlauben eine genaue Lokalisierung und frühe Erkennung von (Implementierungs-)Fehlern
- Modultests können v.a. bei größeren Systemen sehr aufwendig werden, weshalb sie dann unstrukturiert und undokumentiert gestaltet werden – das ist falsch!

Integrationstests

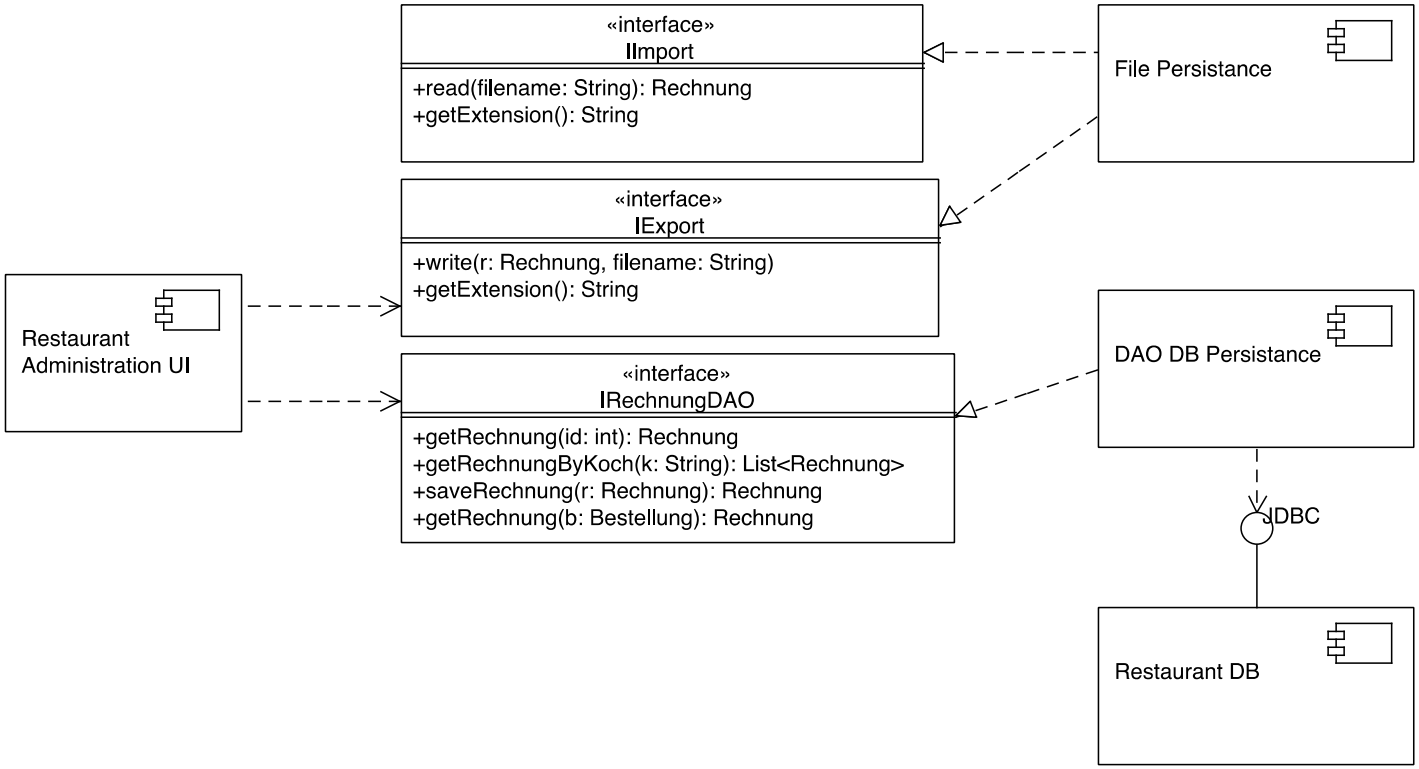
- Ziel: Test der Interaktion zwischen Modulen
- Zusammenführung von Modulen zu größeren Strukturen
  - „Big Bang“
  - Inkrementell: Top-Down oder Bottom-Up
- Häufig: Zusammenführung durch Entwickler, Test durch Tester
- Inkrementelles Testen ist vorzuziehen
  - Weniger Aufwand (weniger Stubs/Treiber nötig)
  - Frühere Erkennung von Schnittstellenfehlern
  - Fehler-Lokation besser eingrenzbar (leichteres Debugging)
  - Weniger Gefahr der Überdeckung von Fehlern in einem Modul durch Fehler in einem anderen Modul.

Beispiel für Module Komponentendiagramm



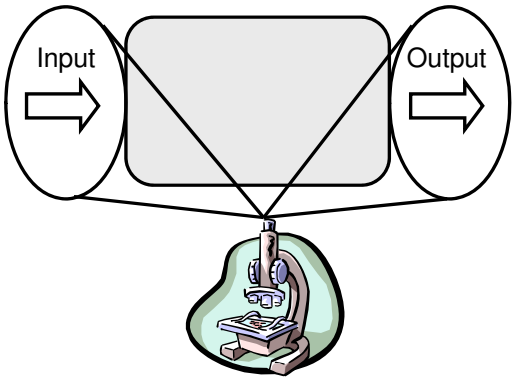
- Komponenten bieten Interfaces an (provides)
- Komponenten erfordern andere Interfaces (requires)
- Dies macht Komponenten zu abstrakten, austauschbaren Modulen

Komponentendiagramm Alternativnotation

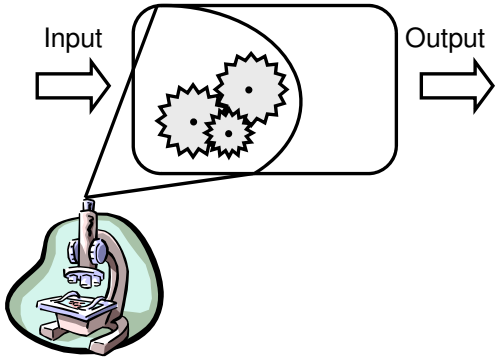


Black Box vs. White Box

- Black Box
  - Basiert auf Spezifikation
  - Wissen über innere Struktur wird ignoriert
  - Daten-getriebene, Input-Output-getriebene Tests
  - Anforderungsüberdeckung
  - Partitionierung der Eingabe-Daten



- White Box (Glass Box)
  - Basiert auf Struktur von Code bzw. SE-Modellen
  - Wissen über innere Struktur notwendig
  - Logik-getriebene Tests
  - Überdeckung von Knoten, Kanten, Pfaden
  - Partitionierung von Daten für Bedingungsauswertung



Teststufen und Testarten in SE

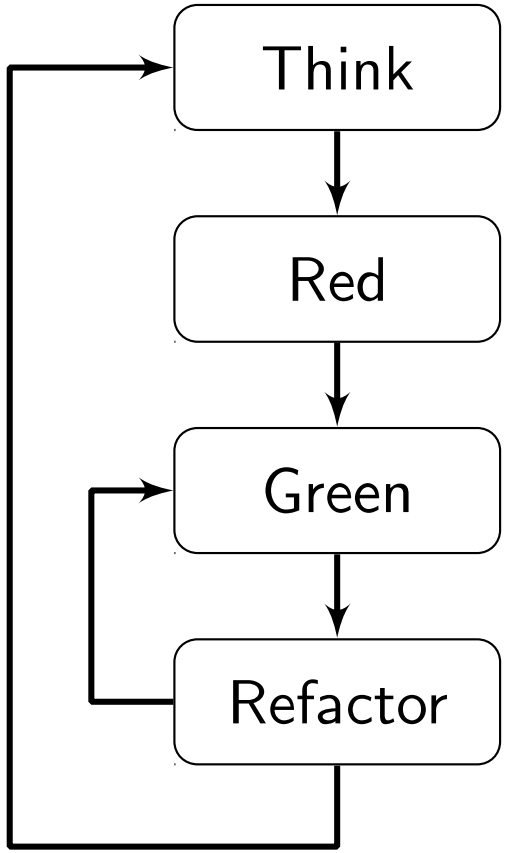
- Teststufe (Unit, Modul, System)
  - Je nach Teststufe ist meist ein anderes Testframework vorzuziehen
  - Jedoch können mit JUnit auch Modul Tests geschrieben werden
  - System Tests (z.B. GUI Tests) sehr mühsam
- Testart (Black-/Whitebox)
  - Unit Tests werden meist als Blackbox ausprogrammiert (TDD)
  - Whiteboxtests sind meist Modul oder Systemtests, sind jedoch visuell kaum von Blackboxtests zu unterscheiden

Test-Driven Development

- Unit Tests: Ablauffähige Testfälle erstellen
- Assertions für „korrekte“ Durchführung des Testfalls ableiten (= erwartetes Ergebnis)
  - Normalfall soll nicht fehlschlagen
  - „Korrekter“ Fehlerfall muss auch wirklich fehlschlagen -> Exception

- Brauchbare Modellierung unterstützt das Herstellen effektiver und effizienter Testfälle
  - Datenmodellierung (Datenbank)
  - Datenfluss und Kontrollfluss (Business Logic)
  - Zustandsübergangsmodelle (GUI)

Test-Driven Development Prozess



1. *Think*: Spezifikation des Tests
2. *Red*: Implementierung des Tests
   - Test schlägt fehl!
3. *Green*: Implementierung der zu-testenden Klasse oder Komponent.
   - Test ist erfolgreich!
4. *Refactor*: Veränderung einer Implementation -> Funktionalität bleibt erhalten -> Test sollte nie wieder fehl schlagen!
   - Lernen aus Test, Veränderung des Prozess

Best-Practices für Testen

- Eine gute Abstimmung der Rollen im Team wird benötigt
- Die Zeit für viele Unit Tests ist wohl investiert da sie die Integrations-Tests wesentlich vereinfachen. (bottom-up testing)

- Technische Architekten:
  - „kleine" Programme, „kleine" Methoden
  - Eigenständig Instanzierbare Klassen
    - Dependency Injection Pattern!
  - Klassendiagramme, Sequenzdiagramme

- Tester:
  - Testet die Objekte so wie sie nachher im Programm verwendet werden

JUnit, .NETUnit etc

- Entwickler erstellt Test-Methoden (Testfälle) während oder sogar vor der Programmierung
- Testfall: Aufruf von Methoden mit bestimmten Parametern, Überprüfung von Ergebniswerten bzw. –zuständen
- Herstellung eines Ausgangszustands innerhalb des Testfalls oder durch andere Testfälle
- Framework erlaubt einfache Erstellung, Ausführung und Auswertung der Testfälle
- somit einfache Wiederholung von Tests z.B. nach Änderungen und Korrekturen möglich (Regressionstests)

Hinweise / Referenzen

- Die Zeit für viele Unit Tests ist wohl investiert, da die Unit Tests die Integrationstests wesentlich vereinfachen (bottom-up testing).
- Test Cases sind „living documents" - sie unterstützen die Kommunikation innerhalb des Teams.
- Formuliert einen wesentlichen Testfall eures eigenen Projektes aus. Wie kann dieser Testfall mit JUnit implementiert werden?

- http://junit.sourceforge.net/doc/testinfected/testing.htm
- http://www.junitdoclet.org/

Beurteilung von Testfällen

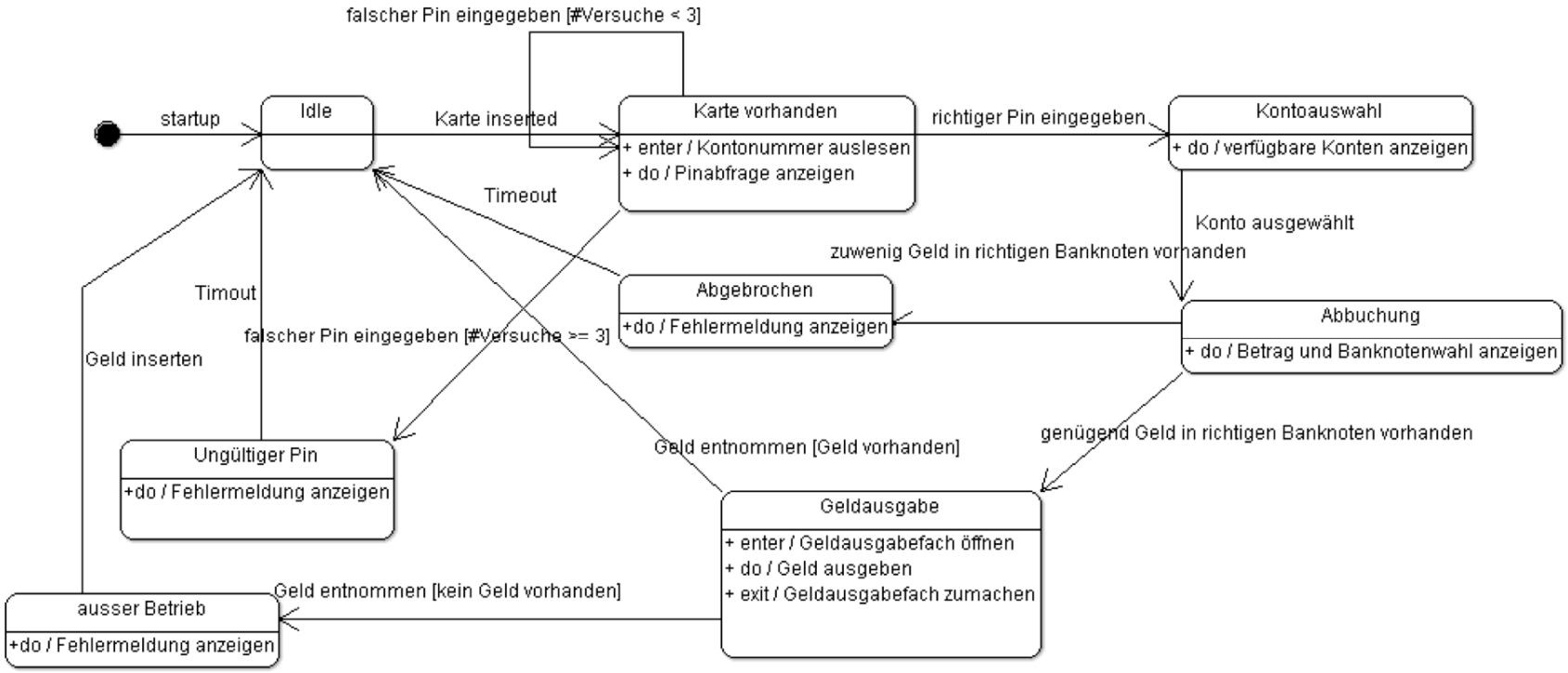
- Ziel aus Testplan bestimmen (Referenz auf Modell)
  - Z.B. Normal-, Sonder- und Fehlerfälle in einem Anwendungsszenario abdecken
  - Z.B. Alle Knoten und Kanten in einem Kontrollflussgraphen abdecken

- Erreichung des Testziels mit Menge der spezifizierten Testfälle überprüfen.

Beispiel Testfälle Cashmachine

- I. Prüfung Abhebewunsch
  - Äquivalenzklassen der Parameter
  - Mögliche Ergebnisse
- II. Berechnung des Geldbetrags

Cashmachine State Chart



Zusammenfassung

- Test-Driven Development als Basis für zielorientierte Software-Entwicklung (Implementierung und QS)
- Brauchbare Modelle unterstützen das Herstellen effektiver und effizienter Testfälle
  - Datenmodellierung (Datenbank)
  - Datenfluss und Kontrollfluss (Business Logic)
  - Zustandsübergangsmodelle (z.B. GUI)
- Reviews helfen, die Anforderungen und Modelle vorab zu überprüfen, um eine solide Basis für Testfälle herzustellen.

Referenzen

Bücher und Skripten

- „Objektorientierte Modellierung“; Unterlagen zur Vorlesung; Business Informatics Group (BIG), TU Wien, 2006.
- „Datenmodellierung“; Unterlagen zur Vorlesung; TU Wien, Inst. f. DB & AI, 2006.
- „ Testen als Entwickler“, Unterlagen zur Vorlesung Qualitätssicherung, Inst. f. Softwaretechnik und Interaktive SystemeM; 2006
- Kemper A., Eickler A.; „Datenbanksysteme“; 6. Auflage, Oldenbourg, 2007.
- Martin Pol, Tim Koomen, Andreas Spillner: “Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software, mit TPI und TMap”, dpunkt.verlag, 2000, ISBN 3-932588-65-7
- [Somm10] Sommerville I.; Software Engineering; 9th ed., Addison Wesley, 2010.
- Spillner et al., "Basiswissen Software-Test", dPunkt, 2003
- Thaller, "Software-Test", dPunkt, 2002
- [Wall11] Wallmüller, E.; Software Quality Engineering; 3. Auflage, Hanser, 2011.

Web Ressourcen

- SEPM Tuwel: https://tuwel.tuwien.ac.at/course/view.php?id=33
  - Prüfungsordner
- Sommerville book companion website: www.pearsoned.co.uk/sommerville
  - Case studies

SEPM Vorlesung – Block 7
Software Engineering & Projektmanagement

Ausgewählte Software Prozesse

Dietmar Winkler

Vienna University of Technology
Institute of Information Systems Engineering
Research Group for Information and Software Engineering

dietmar.winkler@tuwien.ac.at
http://qse.ifs.tuwien.ac.at

Motivation und Zielsetzung

- Die Herstellung von qualitativ hochwertigen Softwareprodukten innerhalb von zeitlichen und budgetären Rahmenbedingungen erfordert ein systematisches Vorgehen.
- Grundlegende Vorgehensweise
  - Systematische und strukturierte Vorgehensweise durch Softwareprozesse. (wann soll welches Produkt in welchem Fertigstellungsgrad verfügbar sein)
  - Konstruktive Methoden zur Herstellung von Software Produkten, z.B. für Spezifikationen, Testfälle, Source Code.
  - Analytische Methoden zur Überprüfung der Produktqualität, z.B. Reviews, Inspektionen und Tests.
- Vorgehensmodelle (Softwareprozesse) unterstützen den Projektleiter und das Entwicklungsteam durch die Bereitstellung eines Rahmenprozesses für den Projektablauf.
- Vorgehensmodelle orientieren sich grundsätzlich am Software Life-Cycle Prozess, betrachten also alle wesentlichen Schritte eines Entwicklungsprozesses.
- Unterschiedliche Projekte (Projektgröße, Anwendungsdomäne, Projekttyp) erfordern aber passende Vorgehensweisen!

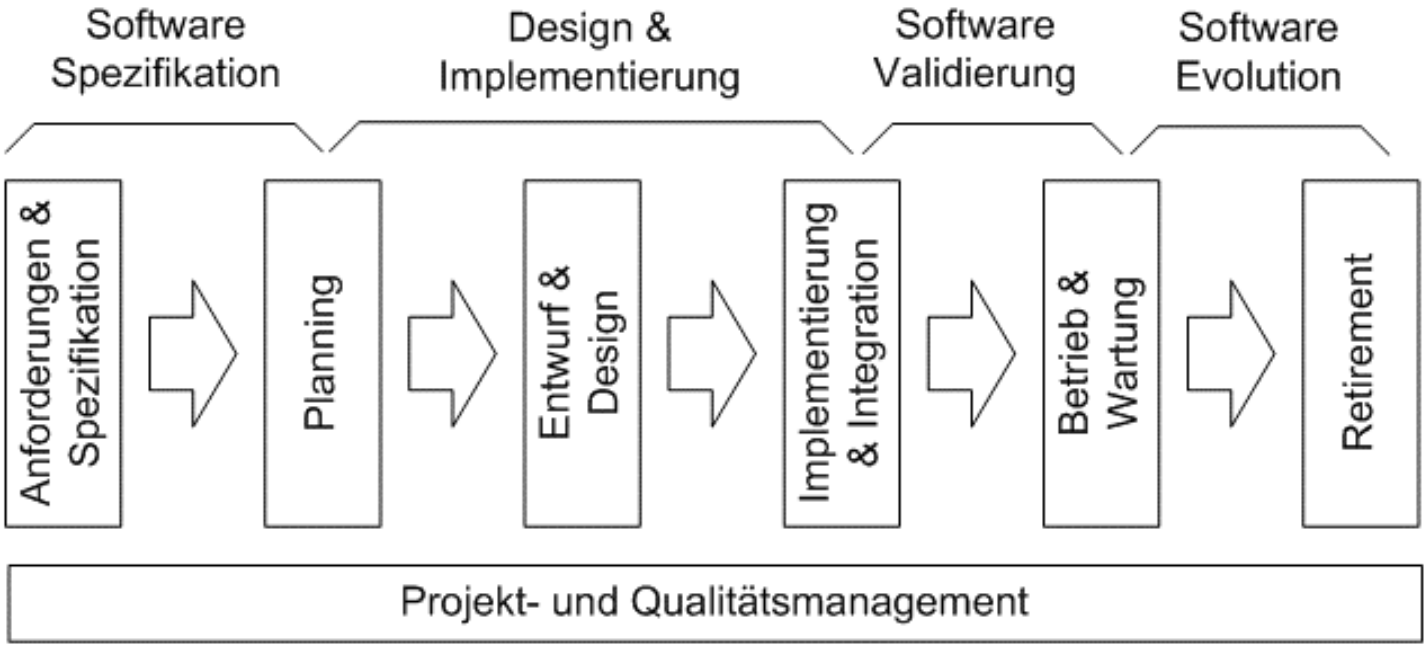
Auswahl eines passenden Software Prozesses.

Table of Contents

- Software Life-Cycle (Wiederholung)
  - Phasen im Software Life-Cycle
  - Vom Software Life-Cycle zum Software Prozess
- Traditionelle Ansätze
  - Wasserfall Modell
  - V-Modell Grundkonzept
  - V-Modell XT
  - Rational Unified Prozess
- Agile Ansätze
  - Agiles Manifest
  - Scrum
- Anpassung von Software Prozessen (Prozess Tailoring)

Software Life-Cycle

- Ein Software-Prozess ist eine Abfolge von Schritten (Phasen) mit all seinen Aktivitäten, Beziehungen und Ressourcen.
- Einsatz von qualitätsverbessernden Maßnahmen in allen Phasen des Life-Cycles, d.h. von der ersten Idee über die Entwicklung bis zum kontrollierten Auslauf des Produktes.
- Der Software Life-Cycle beschreibt ein Basiskonzept für Software Engineering Prozesse und Vorgehensmodelle.



Vom Software Life-Cycle zum Vorgehensmodell

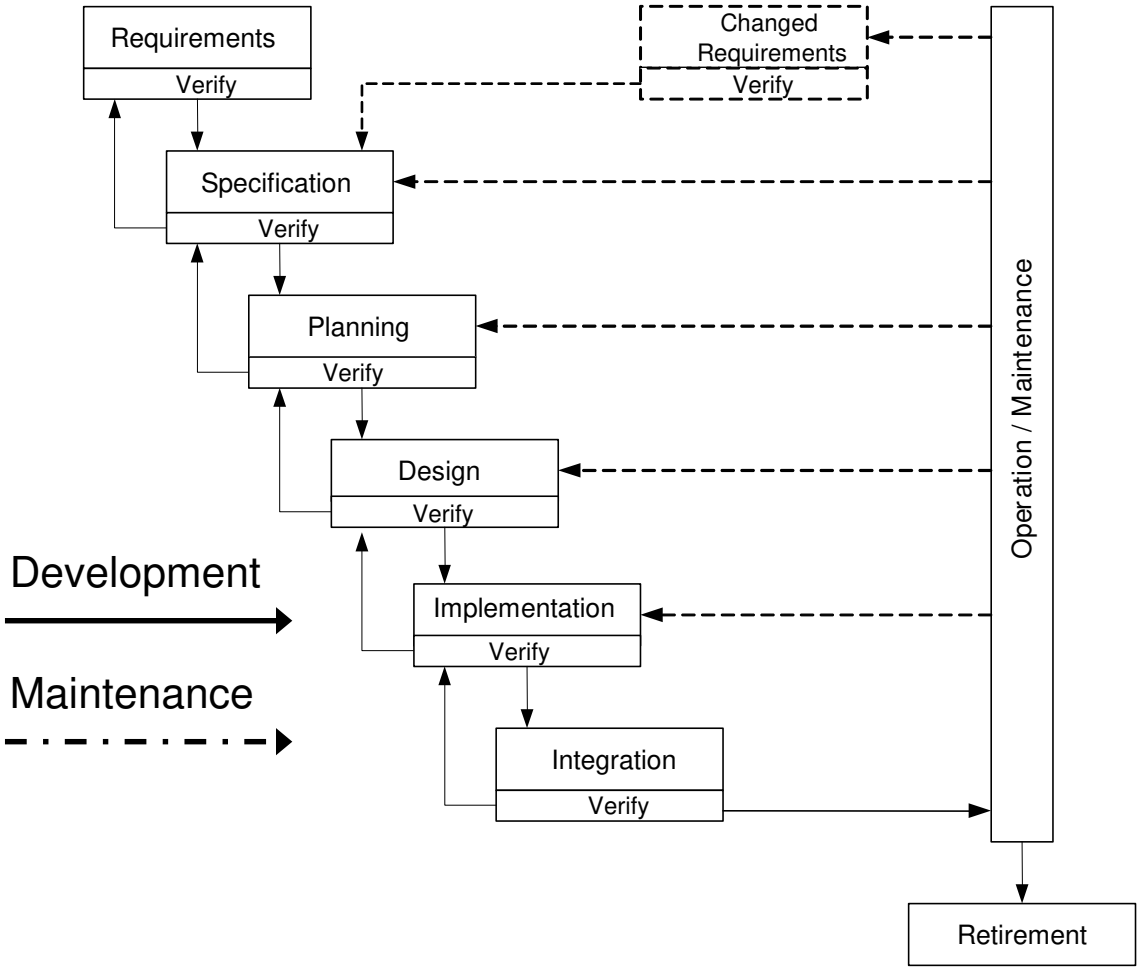
- Die grundlegenden Phasen des Software Life-Cycles finden sich in allen Projekten.
- Der Schwerpunkt der meisten Vorgehensmodelle liegt eher auf der technischen Seite, sie beginnen bei der Definition der Anforderungen und enden bei der Inbetriebnahme beim Kunden.
- In der Praxis finden wir eine Vielzahl von unterschiedlichen Prozessmodellen
  - Standardisierte “common” Prozessmodelle (V-Modell, RUP, Agile Ansätze)
  - Unternehmensspezifische Vorgehensmodelle, die an die Bedürfnisse der jeweiligen Unternehmen bzw. Projekte angepasst werden.
- Software Prozesse oder Vorgehensmodelle sind auf bestimmte Kriterien zugeschnitten und können – je nach Projektkontext – sinnvoll eingesetzt werden.

Ein Vorgehensmodell entspricht einer konkreten Strategie zur kontrollierten Durchführung eines spezifischen Projektes.

Table of Contents

- Software Life-Cycle (Wiederholung)
  - Phasen im Software Life-Cycle
  - Vom Software Life-Cycle zum Software Prozess
- Traditionelle Ansätze
  - Wasserfall Modell
  - V-Modell Grundkonzept
  - V-Modell XT
  - Rational Unified Prozess
- Agile Ansätze
  - Agiles Manifest
  - Scrum
- Anpassung von Software Prozessen (Prozess Tailoring)

Wasserfall Modell (1)



- Erste Veröffentlichung in den 80er Jahren (Royce).
- Umsetzung des Life-Cycles.
- (immer noch) stark verbreitet.
- Einfache Anwendung.
- Schwerpunkt auf Dokumentation.

Wasserfall Modell (2)

Vorteile

- Backtracking zu früherem Entwicklungsphasen.
- Risikominimierung durch “Abschluss” einer Phase.
- Weite Verbreitung und hoher Bekanntheitsgrad.
- Strikte Trennung der einzelnen Phasen.
- Unterstützung von kleinen Entwicklungsteams.

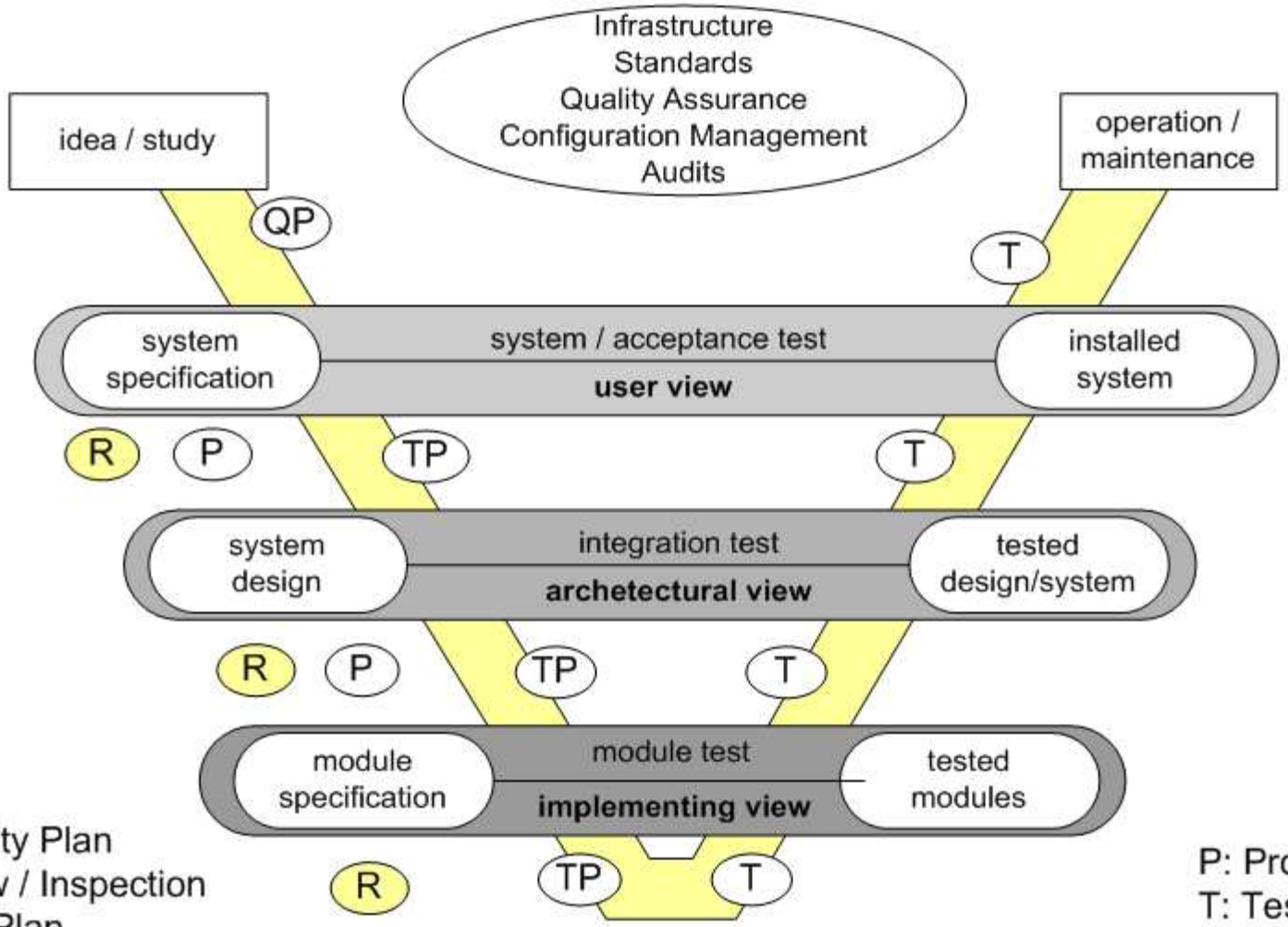
Nachteile

- Alle Tasks einer Phase müssen abgeschlossen werden (keine parallele Entwicklung möglich).
- Starke Auswirkung von Fehlern in frühen Phasen auf das Entwicklungsprojekt.

Anwendungsbereich

- Gute Kenntnis der Anforderungsdomäne erforderlich (*No-Surprise Software*)
- Klar definierte (und vollständige) Anforderungen erforderlich.

V-Modell Konzept mit QS-Methoden



V-Modell Konzept: Vor-/Nachteile

Vorteile

- Spezifikationsphase vs. Realisierung und Testen.
- Kontext von Produkten und Tests.
- Verschiedene Abstraktionslevels (User, Architekten und Implementierungssicht).
- Fehlerbehandlung in frühen Phasen des Softwareentwicklung (durch Einsatz von Reviews).
- Basiskonzept für VM 97 und VM XT.

Nachteile

- Klare Beschreibung der Systemanforderungen ist wichtig.
- Hoher Dokumentationsaufwand.
- Kritisch bei unklaren Anforderungen / sich ändernden Anforderungen.

Anwendungsbereich

- Große Projekte im öffentlichen Bereich.
- Klar definierten Anforderungen.

V-Modell XT

- Das V-Modell XT ist eine Weiterentwicklung des V-Modell 97.
- Veröffentlichung im Februar 2005.
- Laufende Weiterentwicklung (derzeit Version 1.3)
- Verpflichtendes Vorgehensmodell für IT Projekte im öffentlichen Bereich in Deutschland.

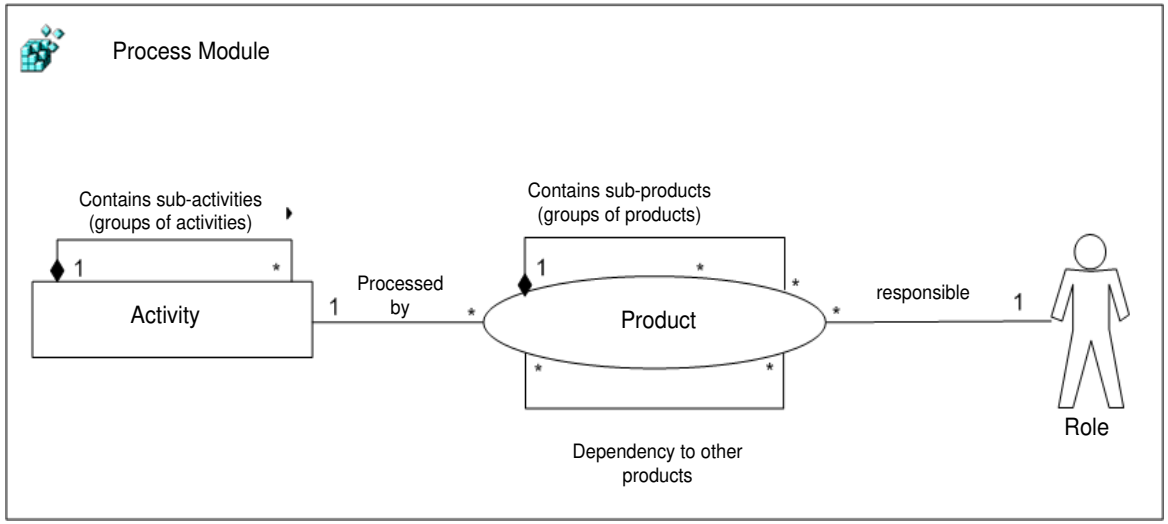
Zielsetzung der Entwicklung des V-Modell XT

- Verbesserung der Unterstützung von Anpassbarkeit, Anwendbarkeit, Skalierbarkeit und Änder- und Erweiterbarkeit des V-Modells.
- Berücksichtigung des neuesten Stand der Technik (Best-Practice).
- Kompatibilität zu formalen Richtlinien und Standards (z.B. ISO 9000 Standard, CMMI).
- Erweiterung des Anwendungsbereiches auf die Betrachtung des Systemlebenszyklus; Integration des Auftraggebers in das Projekt.
- Integration eines Prozessmodells zur “Einführung und Pflege eines organisations-spezifischen Vorgehensmodells).

[http://www.v-modell-xt.de]

Philosophie und Grundkonzept

- Produkte stehen im Mittelpunkt (=Projektergebnisse), für jedes Produkt gibt es definierte Rollen mit definierten Verantwortlichkeiten.
- Projektdurchführungsstrategien und Entscheidungspunkte geben die Reihenfolge der Produktfertigstellung und somit den Projektverlauf vor.
- Vorgehensbausteine sind die modularen Elemente des V-Modell XT.
  - kapselt Rollen, Produkte und Aktivitäten.
  - Kann als unabhängige Einheit eingesetzt werden.
  - Ist eine Einheit, die unabhängig veränder- und aktualisierbar ist.



Komponenten des V-Modell XT

- Projekttypen vs. Projektgegenstand
- Vorgehensbausteine kapseln Produkte, Aktivitäten und Rollen
  - Verpflichtende Elemente (core elements)
  - Optionale Elemente (um individuelle Projektanforderungen erfüllen zu können)
- Unterstützung der Anpassbarkeit durch integrierte Tailoringmechanismen.
- Integrierte Methoden- und Toolunterstützung zur
  - Erstellung von Produkten durch
  - Aktivitäten und
  - Rollen (verantwortlich für ein Produkt).
- Entscheidungspunkte (etwa Meilensteine) definieren einen Zeitpunkt, an dem eine Fortschrittsentscheidung getroffen wird.
- Projektdurchführungsstrategien definieren die Reihenfolge der im Projekt zu erreichenden Projektfortschrittsstufen (Sequenz von Entscheidungspunkten)
- Durch die Struktur des V-Modell XT ist eine Vergleichbarkeit zu herkömmlichen Prozessmodellen möglich
  (z.B. Konventionsabbildungen zu Prozessmodellen und Standards)

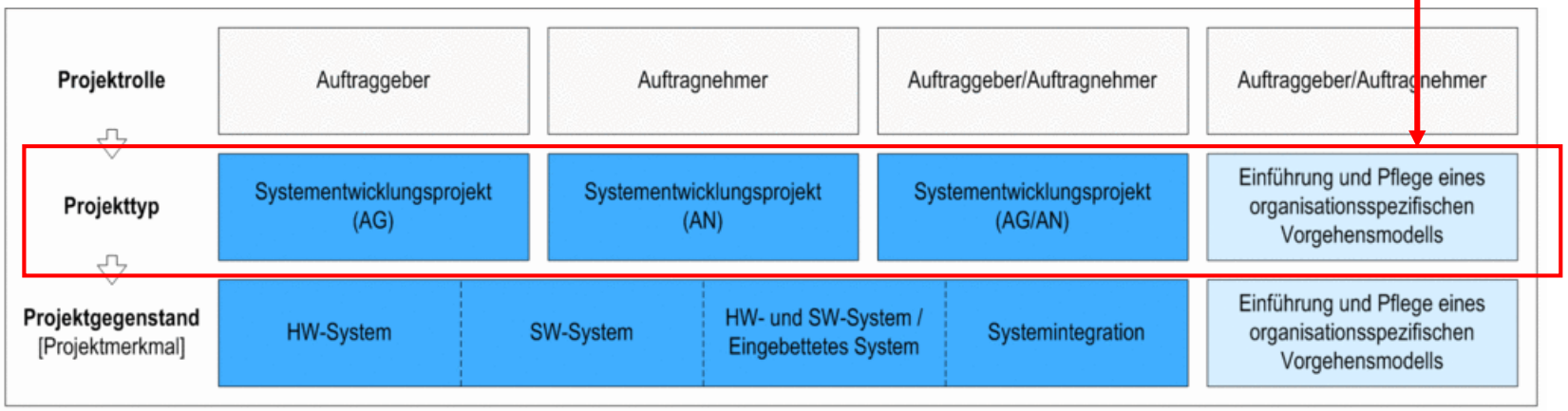
Projekttypen

Projekttypen werden eingeteilt:

- Nach Projektgegenstand (z.B. Hardwaresystem, Softwaresystem, Komplexes System)
- Projektrollen (z.B. Auftraggeber / Auftragnehmerprojekte)

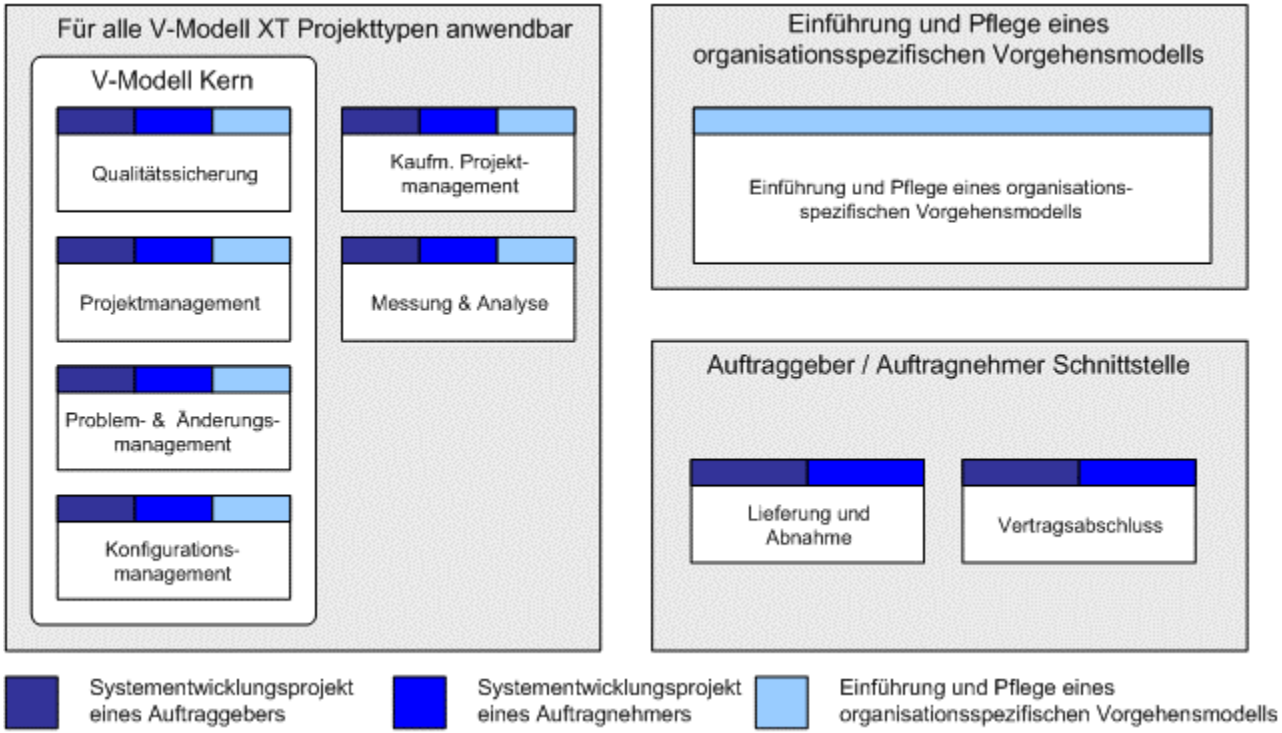
daraus resultieren (derzeit) 4 Projekttypen

- Systementwicklungsprojekt des Auftraggebers.
- Systementwicklungsprojekt des Auftragnehmers.
- Einführung und Pflege eines organisationsspezifischen Vorgehensmodells
- Systementwicklungsprojekt (Auftraggeber/Auftragnehmer);
  z.B. in-house System Entwicklung (seit der Version 1.2 integriert).



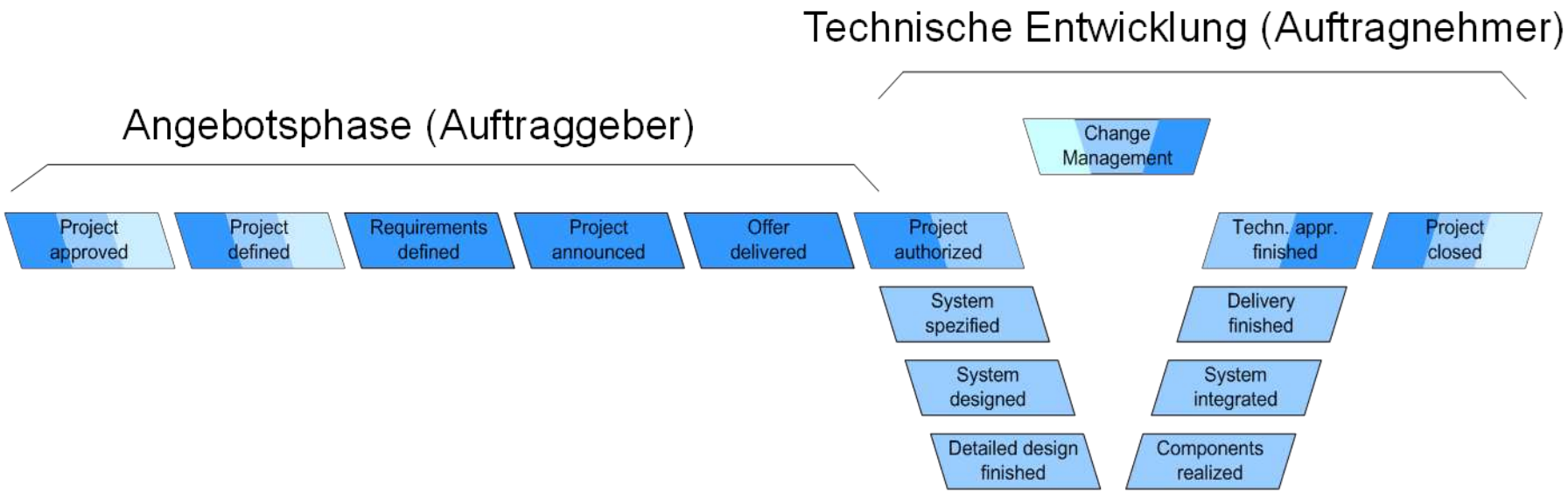
Vorgehensbausteine im V-Modell XT

- V-Modell Kern (verpflichtende Elemente für alle Projekttypen).
- Einführung und Pflege eines organisationsspezifischen Vorgehensmodells.
- Elemente für die Systementwicklung
- Auftraggeber / Auftragnehmer – Schnittstelle.
- Tool-Unterstützung durch den V-Modell Assistenten.



Projektdurchführung

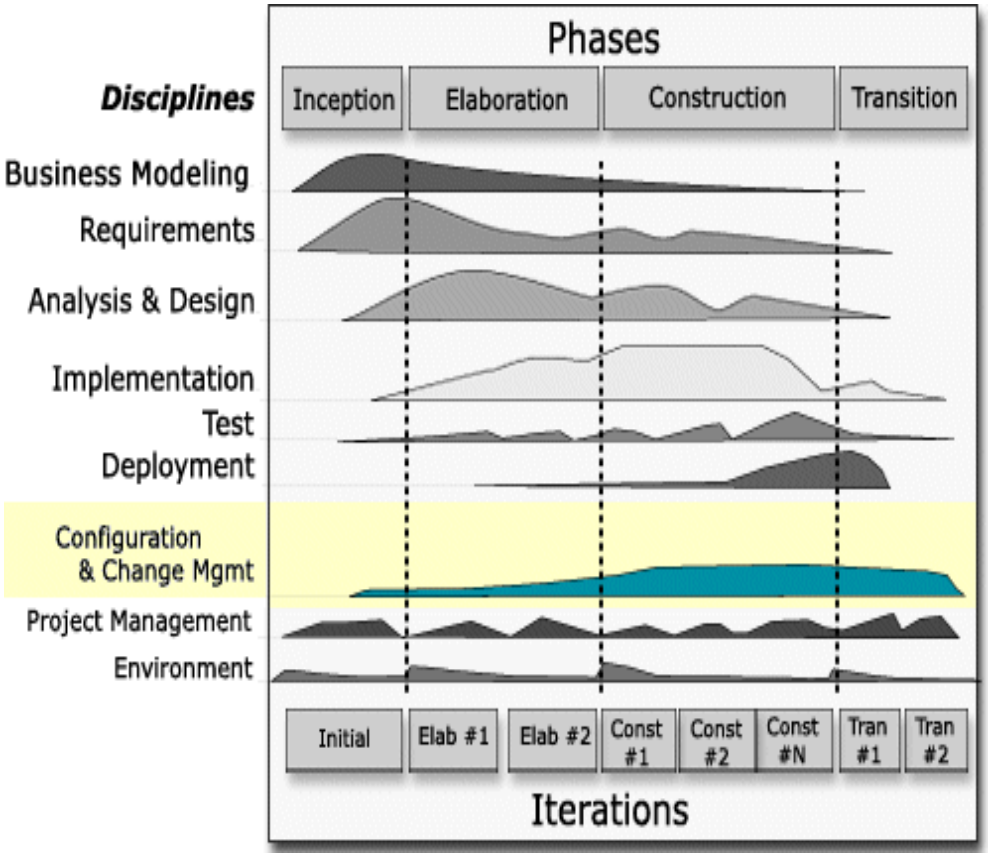
- Entscheidungspunkte und Projektdurchführungsstrategie.
  - Definitionen von Entscheidungspunkten (vergleichbar mit Meilensteinen)
  - An Entscheidungspunkten müssen definierte Produkte vorliegen.
  - Eine Projektdurchführungsstrategie ist eine definierte Abfolge von Entscheidungspunkten (z.B. inkrementelle oder agile Entwicklungsstrategie).
- Beispiel



V-Modell XT in der Anwendung

- Flexible Anwendung des V-Modell XT durch Anpassung des Modells an unterschiedliche Projektgegebenheiten (Projekttyp, Projektmerkmale).
  - Auswahl von benötigten Vorgehensbausteinen.
  - Definition der passenden Projektdurchführungsstrategie.
- Werkzeugunterstützung (Open Source)
  - V-Modell XT Projektassistent zur Anpassung des Modells an ein konkretes Projekt.
  - Ergebnis ist eine angepasste Vorgehensweise und angepasste Templates für die Projektdokumentation.
  - V-Modell XT Editor ermöglicht freie Konfigurationen des Vorgehensmodells, z.B. Anpassung auf ein Unternehmensmodell (Standard).
- Verpflichtendes Vorgehensmodell für öffentliche IT Projekte in Deutschland.
- Konventionsabbildungen ermöglichen die Kompatibilität zu Qualitätsmanagementstandards, wie CMMI und ISO 9000 sowie zu anderen Vorgehensmodellen, wie dem Rational Unified Process.

Rational Unified Process, RUP (1)



- Inkrementelle und iterative Vorgehensweise.

4 grundlegende Phasen:

- Inception (Beginn)
- Elaboration (Concept & Design)
- Construction
- Transition (Auslieferung)

Definierte Workflows und Disziplinen:

- 6 Engineering Workflows
- 3 Supporting Workflows
- Mehrere Iterationen innerhalb einer Phase.

[Kruchten, 2004]

Rational Unified Process, RUP (2)

- Iterative und inkrementeller Workflow.
- Integriertes Anforderungsmanagement.
- Komponenten-orientierte Architektur.
- Modellierung durch das UML Methodenframework.
- Produkt-Verifikation an Meilensteinen.
- Änderungsmanagement (*supporting discipline*).

Vorteile

- Real-world Szenarien.
- Werkzeugunterstützung (Rational XDE, IBM).
- Vordefinierte Liste mit erforderlichen Artefakten.

Nachteile

- Hohe Komplexität.
- Hoher Dokumentationsaufwand.
- Anbieterabhängigkeit?

Anwendungsbereich:

- Grosse Projekte durch eine ganzheitliche Prozess-Sicht auf das gesamte Projekt (inkl. Deployment).

Table of Contents

- Software Life-Cycle (Wiederholung)
  - Phasen im Software Life-Cycle
  - Vom Software Life-Cycle zum Software Prozess
- Traditionelle Ansätze
  - Wasserfall Modell
  - V-Modell Grundkonzept
  - V-Modell XT
  - Rational Unified Prozess
- Agile Ansätze
  - Agiles Manifest
  - Scrum
- Anpassung von Software Prozessen (Prozess Tailoring)

Agiles Manifest

- Aus den Kritikpunkten der systematischen und „schwergewichtigen“ Ansätzen entwickelte sich das Agile Manifest.
- Festgeschrieben durch 17 Softwareentwickler 2001.

„Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan“

- Das Manifest beinhaltet 12 Prinzipien, die die Basis für agile Software Entwicklung darstellen.
- Agile bedeutet aber nicht „unkontrolliert“ – auch hier existieren Prozesse und Regeln, die eingehalten werden müssen.
- Bekannte Vertreter: eXtreme Programming oder Scrum.

Quelle: http://agilemanifesto.org/.

12 Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

[http://agilemanifesto.org]

SCRUM

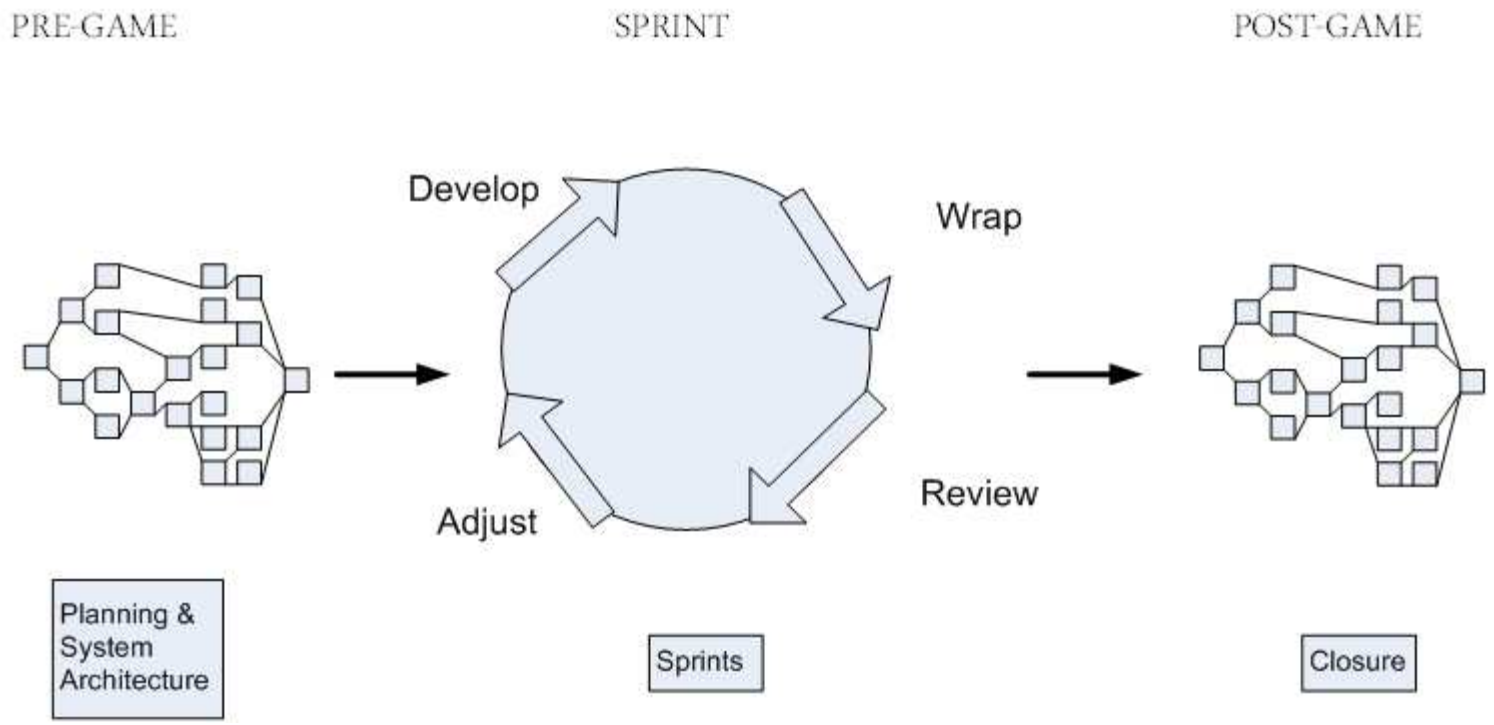
- SCRUM ist keine Abkürzung; der Begriff stammt aus der Rugby Start-Formation.
- Analogie zu Software Engineering ????

[Source: unknown]

- Was leistet Scrum?
  - Agiler Software Prozess aus Sicht des Projektmanagements (PM).
  - Kleine aber hoch-effiziente Teams (auch mehrere Teams möglich).
  - Flexibles Prozessmodell, um auf ändernde Anforderungen im Projektablauf reagieren zu können.
  - (Teil-)Produkte stehen dem Kunden frühzeitig zur Verfügung.
  - Das Projekt wird bestimmt durch *Zeit*, *Wettbewerb*, *Kosten*, und *Funktionalität*.
  - Deliverables werden beeinflusst von Marktinformationen, Kundenkontakt und Skills der Entwickler.
  - Hoher Bekanntheitsgrad in den letzten Jahren.
- Hoher Erfüllungsgrad der agilen Prinzipien.

Der SCRUM Prozess

- Scrum besteht aus einer Sammlung von Prozeduren, Rollen und Methoden für Projektmanagement.
- Selbst-Organisierende Teams.
- Aufbau:

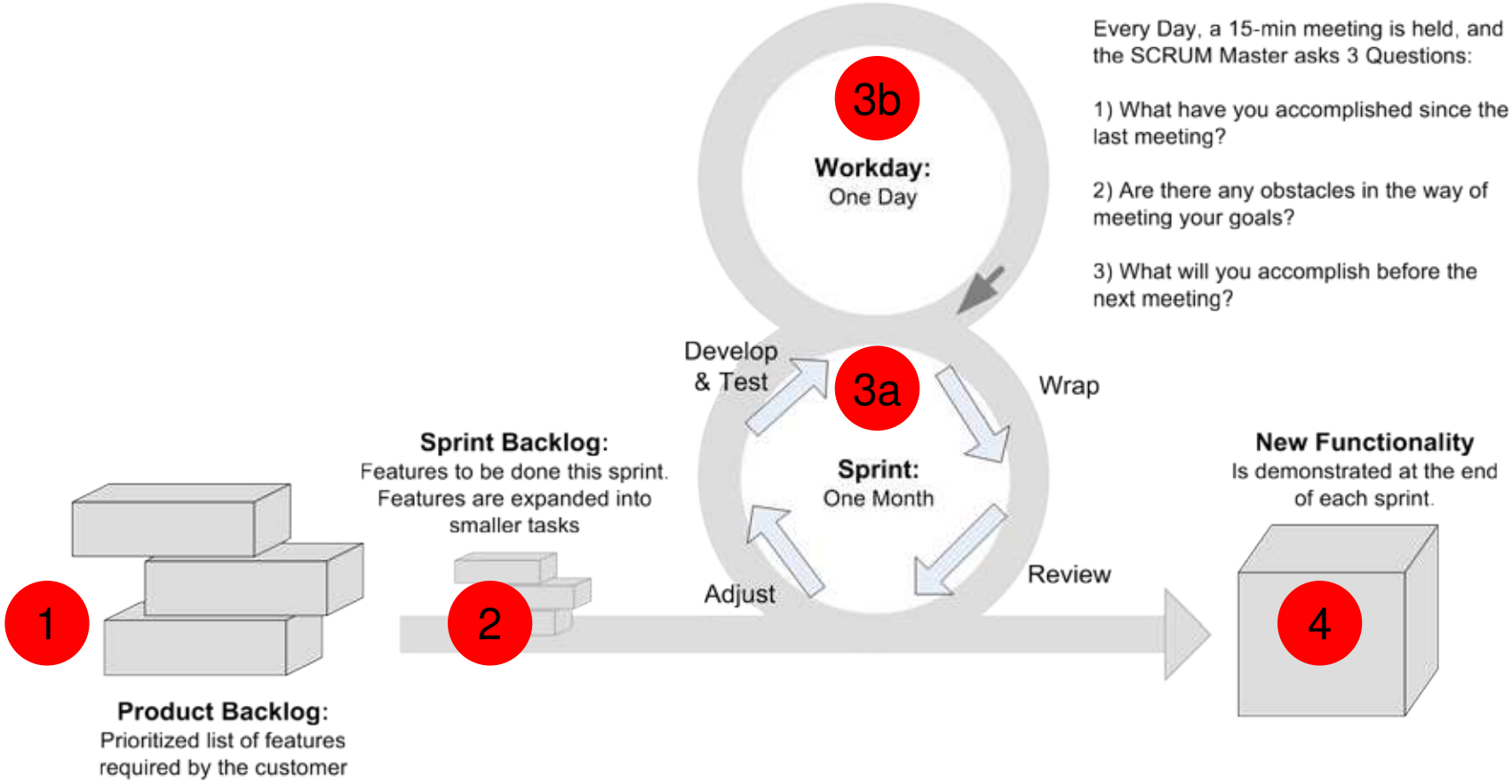


Sprints

PREGAME → SPRINT → POSTGAME

(Planning & System Architecture)

(closure)



SCRUM Charakteristika und Begriffe

Charakteristika

- Ein Team erstellt eine Einheit, ein Produkt oder Teilprodukt.
- Klare Arbeitsaufteilung innerhalb des Teams.
- Klar priorisierte Projektergebnisse (*Backlog Items*).
- Ein gemeinsames Ziel (Erstellung des Produktes).
- Der "Sprint" ist das zentrale Element.
- Das Sprint-Team kann ungestört arbeiten; es sind keine Eingriffe „von aussen" (z.B. durch den Kunden) zulässig.
- Temporal structure = daily Scrum Meeting + Review + Retrospective.

Begriffe:

- Backlog beinhaltet alle Arbeitspakete, die in "nächster" Zeit umgesetzt werden müssen (sowohl klar definierte als auch vage Anforderungen): Produkt vs. Sprint Backlog.
- Ein Daily-Scrum ist eine tägliche Projektbesprechung um etwaige Fragen / Missverständnisse zu klären; Definition des täglichen Arbeitsauftrags.
- Scrum Team: Funktionsübergreifendes Team, zuständig für den Sprint Backlog.
- Burndown Chart: Visuelle Darstellung des Projektfortschritts.

Table of Contents

- Software Life-Cycle (Wiederholung)
  - Phasen im Software Life-Cycle
  - Vom Software Life-Cycle zum Software Prozess
- Traditionelle Ansätze
  - Wasserfall Modell
  - V-Modell Grundkonzept
  - V-Modell XT
  - Rational Unified Prozess
- Agile Ansätze
  - Agiles Manifest
  - Scrum
- Anpassung von Software Prozessen (Prozess Tailoring)

Process Tailoring / Customization

- Anwendbarkeit von standardisierten Softwareprozessen im konkreten Projekt?

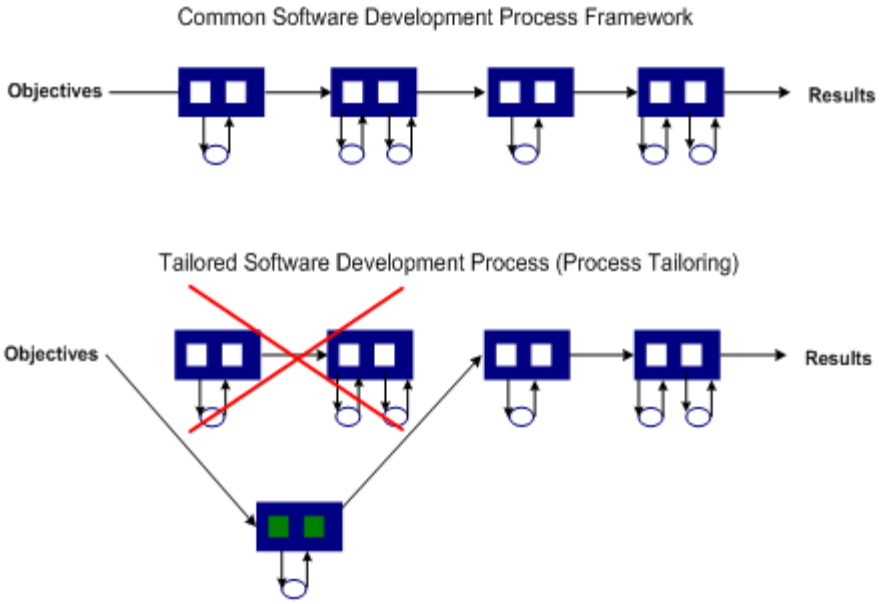
- Standardisierte Software Prozesse sind in der Praxis direkt eher schwer einsetzbar, da eine Vielzahl an Projektattributen berücksichtigt werden müssen; Beispiele: Projektgröße, Projekttyp und Anwendungsdomäne.
- Anpassungen an aktuelle Projektgegebenheiten sind notwendig.
- Diese Anpassungen von Vorgehensmodellen erfordern erfahrene Projektleiter und/oder effiziente Toolunterstützung (z.B. V-Modell XT Projektassistent).

- Prozess Tailoring (bezogen auf der jeweilige Projekt)
  - Anpassung an individuelle Projektgegebenheiten.

- Prozess Customization (Standardisierung)
  - Anpassungen an Unternehmensstandards (z.B. Siemens stdSEM)
  - Domänenabhängige Anpassungen, z.B. für Produktionsautomatisierung

Anpassung von Vorgehensmodellen an Projekte: Prozesstailoring

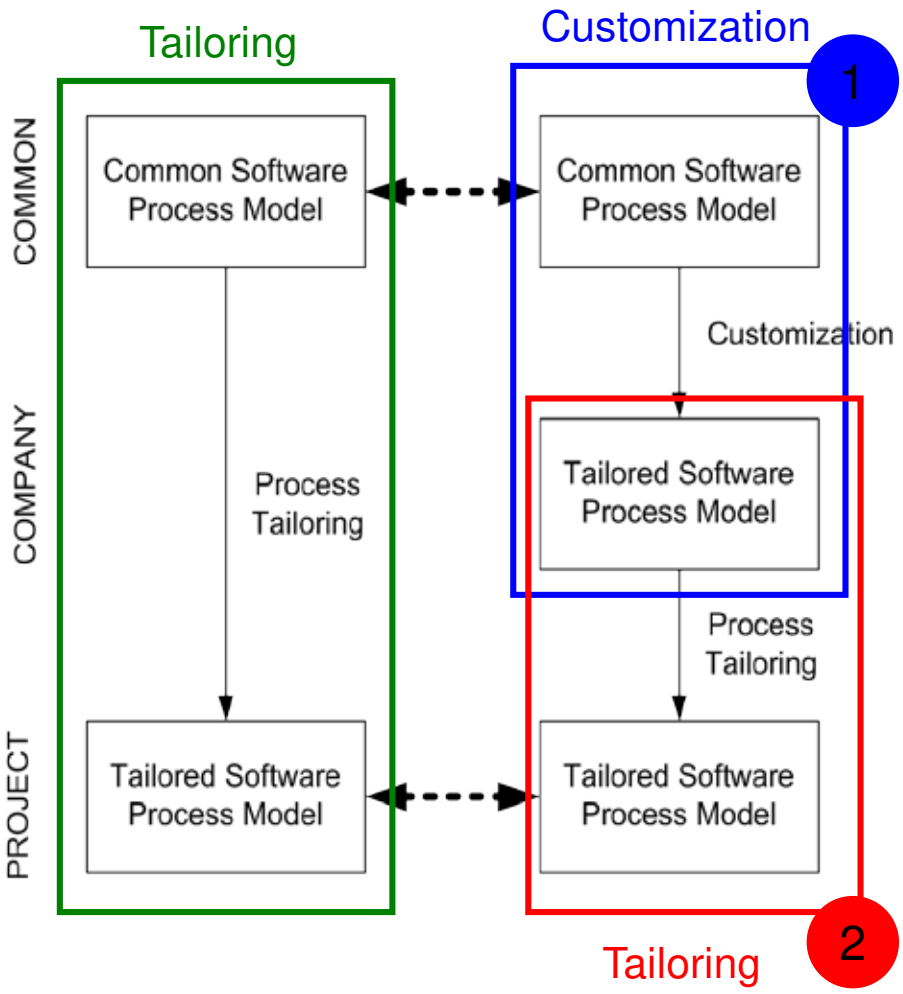
- Anpassbarkeit eines generischen Entwicklungsprozesses an spezifische Projektgegebenheiten durch Prozesstailoring.
- Ersetzen einzelner Prozess-Schritte (oder Vorgehensbausteine) durch passende alternative Lösungen.
- Wiederverwendung von Best-Practices (Methoden / Tools).
- Individuelle Anpassung des Projektplans.



- Achtung:
  - Tailoring erfordert erfahrene Projektleiter, da ein fundiertes Verständnis des Modellaufbaus erforderlich ist.
  - Berücksichtigung von definierten Tailoring-Kriterien und Produktabhängigkeiten.

Prozess „Customization“

- Verallgemeinerte Form des „Tailorings“.
- Anpassung des Vorgehensmodells an unternehmensspezifische Gegebenheiten.
- Effizienzsteigerung von Tailoring für ähnliche Aufgaben durch Customization:
  - Unternehmensstandards: Anpassung an das Unternehmen.
  - Projektstandards: Anpassung an ähnliche Projekte (z.B. Webapplikationen).
- Diese “angepassten Prozesse” dienen als Grundlage für projektspezifisches Tailoring.
- Achtung: Die Kompatibilität zum zugrunde liegenden Prozess muss sichergestellt werden!



Zusammenfassung

- Software Prozesse und Vorgehensmodelle ermöglichen vorhersagbare und nachvollziehbare Softwarelösungen.
- Sie definieren, wie ein Softwareprojekt durchgeführt wird bzw. in welcher Reihenfolge die einzelnen Phasen ablaufen.
- In der Praxis existiert eine Vielzahl an Prozessen für unterschiedliche Anwendungsbereiche (z.B. für spezifische Domänen, Projekttypen).
- Systematische Prozesse sind durch ihre Struktur *plan-driven* und orientieren sich eher an Abläufen mit Schwerpunkt auf Produkten und Dokumentation.
  Beispiele: Wasserfallmodell, V-Modell (XT), RUP, Inkrementelle Modelle.
- Agile Ansätze rücken den Kunden und seine konkreten (sich ändernden) Anforderungen in den Vordergrund.
  Beispiele: eXtreme Programming, SCRUM.
- Durch Tailoring wird ein allgemeines Modell auf ein individuelles Projekt angepasst.
- Customization ermöglicht die Erstellung unternehmensspezifischer Vorgehensmodelle für gleichartige Projekte / Produktgruppen.

Literaturreferenzen

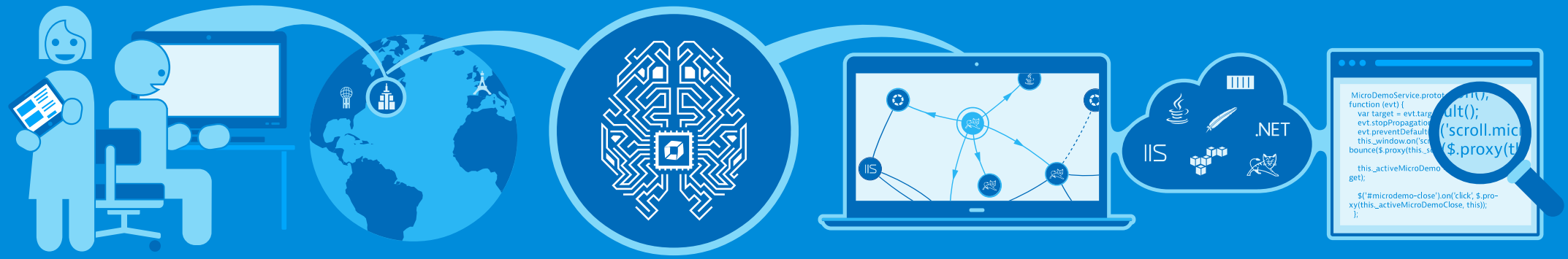
- Beck K.: „Extreme Programming. Das Manifest“, Addison Wesley, 2004.
- Biffl Stefan, Winkler Dietmar, Frast Denis: „Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung“, Skriptum zur Lehrveranstaltung, 2004. http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm
- Höhn R., Höppner S.: „Das V-Modell XT. Grundlagen, Methodik und Anwendungen“, Springer, eXamen Press, 2008.
- Kruchten P.: „The Rational Unified Process: An Introduction“, Addison-Wesley Longman, 2004.
- Schatten A., Biffl S., Demolsky M., Gostischa-Franta E., Östreicher T., Winkler W.: „Best Practice Software Engineering. Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen“, Spektrum Akademischer Verlag, 2010, 978-3827424860.
- Schwaber K., Irlbeck T.: „Agiles Projektmanagement mit Scrum“, Microsoft Press, 2007.
- SCRUM, www.controlchaos.com, February 2006.
- Software Engineering – Best practices: http://best-practice-software-engineering.blogspot.com/
- V-Modell XT: http://www.v-model-xt.de.

dynatrace

Dynatrace Guest Lecture @ TU Vienna

LVA SEPM SS2018

May 30, 2018



confidential

Daniela Rabiser – Technical Product Manager at Dynatrace

- 2017-today: **Technical Product Manager, Dynatrace**

Broadening and deepening monitoring capabilities for enterprise cloud platforms

- 2013-2016: Wissenschaftliche Projektmitarbeiterin / Doktoratsstudium der Technischen Wissenschaften, Dissertationsgebiet: Software Engineering

- 2010-2013: Studentische Mitarbeiterin / Wissenschaftliche Projektmitarbeiterin

- 2005-2010: Softwareentwicklerin

confidential

Stefan Frandl – Development Lead at Dynatrace

- 2016 - today: DevLead OneAgent, Dynatrace
- 2007 – 2015: Test Automation Team Lead, Dynatrace
- 2005 – 2006: Software Engineer, Dynatrace
- 2000 – 2005: Winf JKU, TU Wien

confidential

Dynatrace is worldwide market-leader in

DPM

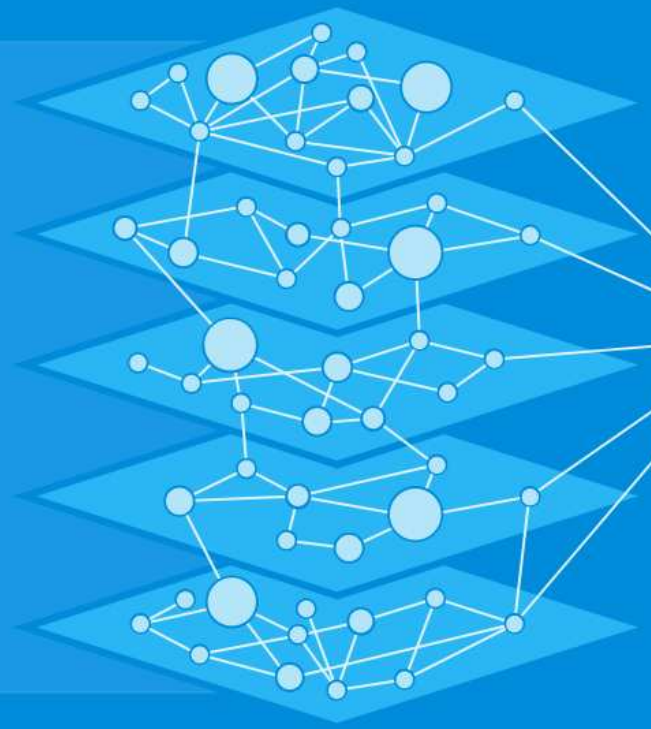
Digital Performance Management

Dynatrace monitors

Every user, every app, everywhere. AI-powered, full stack, & automated.

SaaS | On-premises

Mobile	Browser
Application	Synthetic
Code	Server
Database	Mainframe
Network	Log & Events
Container	API
Microservices	Cloud



- Optimize customer experience
- Modernize operations
- Accelerate innovation

Full stack monitoring
All-in-one platform

Smartscape & PurePath
Business transaction topology

Big data analytics
powered by A.I.

Some facts about Dynatrace

- Global R&D Labs
  Linz, Klagenfurt, Barcelona, Gdansk, Detroit
- Founded 2005 in Linz
- 1.7k employees ww
- 10k customers
  80% of Fortune500
- $500M revenue
- Strong Y/Y growth

Dynatrace is the Leader in the Gartner Magic Quadrant for 8 years running

2018 Gartner Magic Quadrant for Application Performance Monitoring (APM) Suites

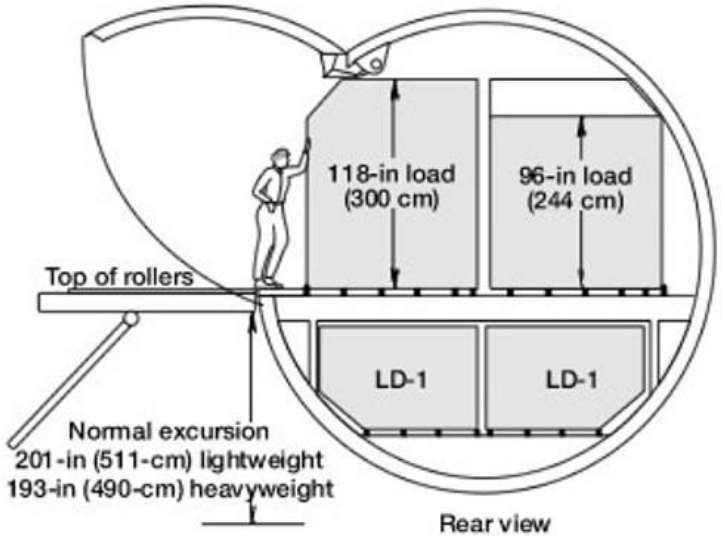
Dynatrace is an industry leader for the 8th consecutive year

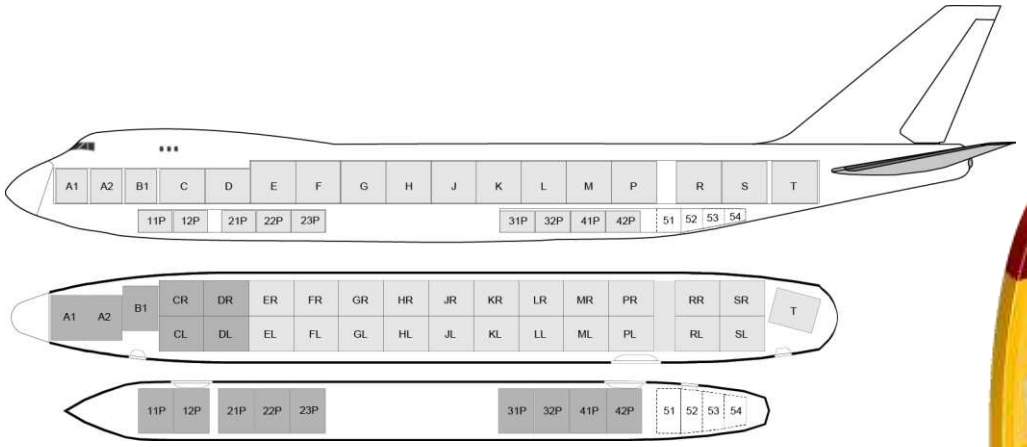
The #1 partner for digital performance management

Story

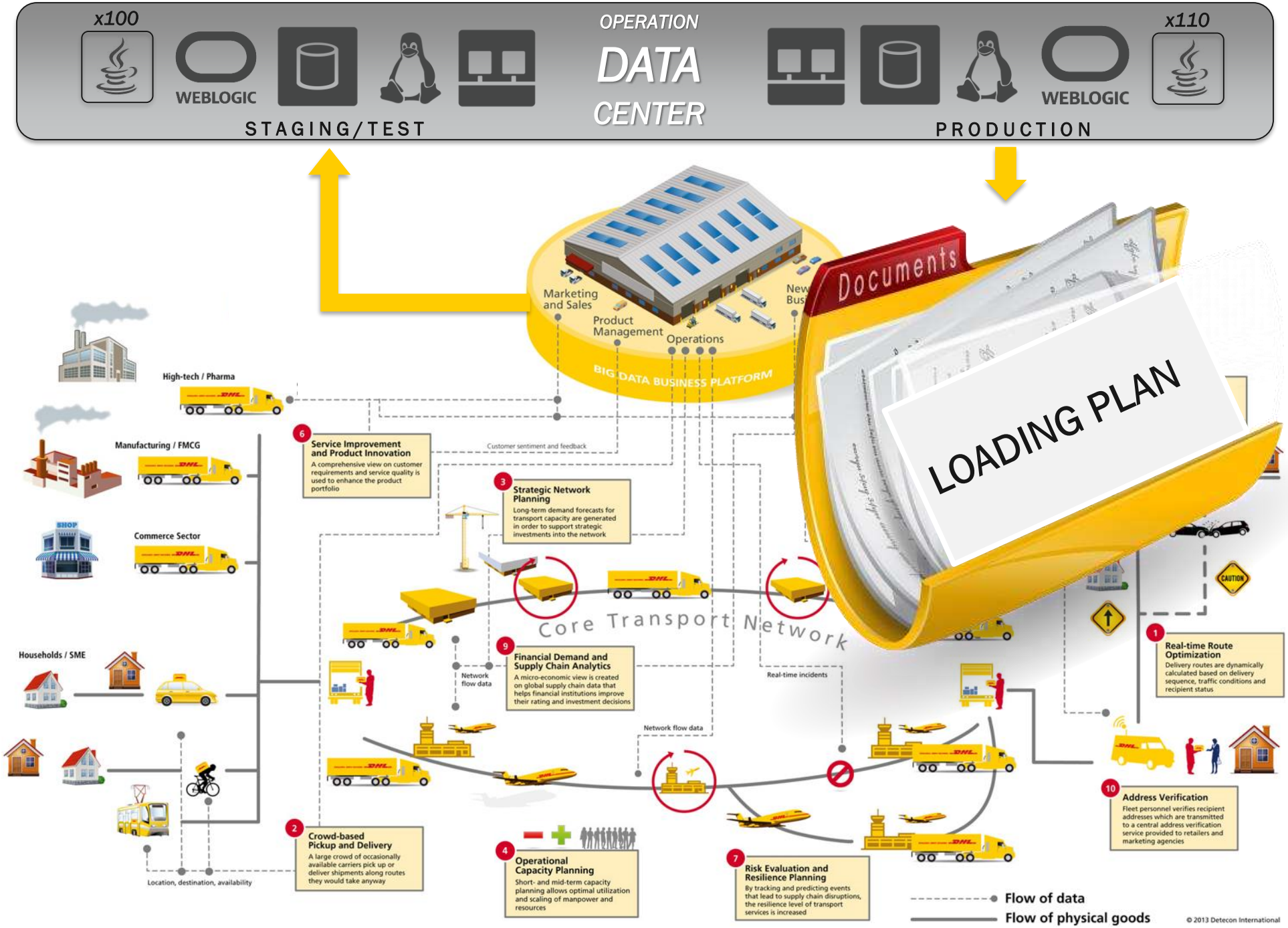
Global Logistics

A330
AIRBUS A330-200F


118-in load
(300 cm)
96-in load
(244 cm)
Top of rollers
LD-1
LD-1
Normal excursion
201-in (511-cm) lightweight
193-in (490-cm) heavyweight
Rear view


A1 A2 B1 C D E F G H J K L M P R S T
11P 12P 21P 22P 23P 31P 32P 41P 42P 51 52 53 54
A1 A2 B1 CR DR ER FR GR HR JR KR LR MR PR RR SR T
CL DL EL FL GL HL JL KL LL ML PL RL SL
11P 12P 21P 22P 23P 31P 32P 41P 42P 51 52 53 54

Documents
LOADING PLAN


x100
WEBLOGIC
STAGING/TEST
OPERATION
DATA
CENTER
PRODUCTION
WEBLOGIC
x110
Marketing and Sales
Product Management
Operations
BIG DATA BUSINESS PLATFORM
Documents
LOADING PLAN
High-tech / Pharma
Manufacturing / FMCG
Commerce Sector
Households / SME
Core Transport Network
6 Service Improvement and Product Innovation
A comprehensive view on customer requirements and service quality is used to enhance the product portfolio
Customer sentiment and feedback
3 Strategic Network Planning
Long-term demand forecasts for transport capacity are generated in order to support strategic investments into the network
9 Financial Demand and Supply Chain Analytics
A micro-economic view is created on global supply chain data that helps financial institutions improve their rating and investment decisions
Network flow data
Real-time incidents
Network flow data
1 Real-time Route Optimization
Delivery routes are dynamically calculated based on delivery sequence, traffic conditions and recipient status
10 Address Verification
Fleet personnel verifies recipient addresses which are transmitted to a central address verification service provided to retailers and marketing agencies
2 Crowd-based Pickup and Delivery
A large crowd of occasionally available carriers pick up or deliver shipments along routes they would take anyway
Location, destination, availability
4 Operational Capacity Planning
Short- and mid-term capacity planning allows optimal utilization and scaling of manpower and resources
7 Risk Evaluation and Resilience Planning
By tracking and predicting events that lead to supply chain disruptions, the resilience level of transport services is increased
Flow of data
Flow of physical goods
© 2013 Detecon International

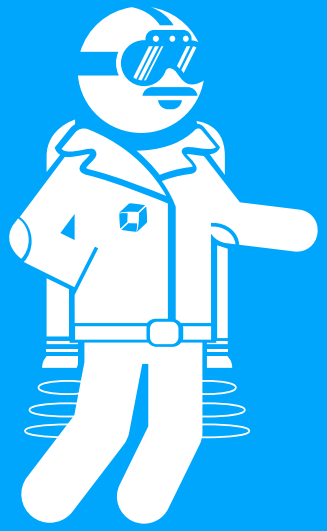
What if

LOADING PLAN

generation is too slow/late

?

CPU Load
25.0
20.0
15.0
10.0
5.0
%
04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 28-Nov 02:00
Current CPU load (RCTS_MonitoringProd@bezavwv40807, RCTS_MonitoringProd@bezavwv40807)
Used / Free Memory
8,000
6,000
4,000
2,000
0
MB
04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 28-Nov 02:00
Used Memory
Free Memory
Suspension Time
40,000
30,000
20,000
10,000
0
ms
04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 28-Nov 02:00
Suspension Time
Garbage Collection
16,000
12,000
8,000
4,000
0
ms
04:00 06:00 08:00 10:00 12:00 14:00 16:00 18:00 20:00 22:00 28-Nov 02:00
Garbage Collection Old Generation (RCTS_MonitoringProd@bezavwv40807, RCTS_MonitoringProd@bezavwv40807)
Garbage Collection Young Generation (RCTS_MonitoringProd@bezavwv40807, RCTS_MonitoringProd@bezavwv40807)



Development Process

confidential

Roles

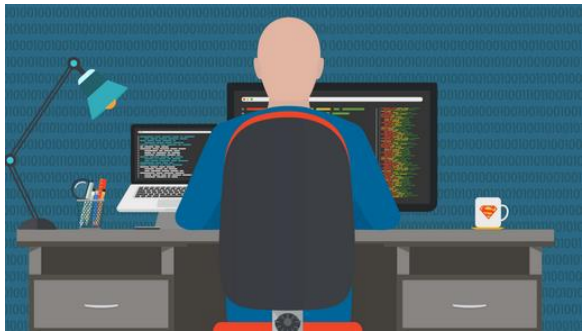
- Product Architects
  - Responsible for architecture design, scalability and reliability
  - Non-func

- Team Lead
  - Leads an engineering team of 4-8 engineers
  - Shared product owner role with Product Management

confidential

Roles

- Engineer
  - Implements user stories
  - Build it, run it (tests, quality, deployment, support)



- DevOps
  - Takes care of operations
  - Roll-out of new features, fixes, etc.
  - Direct feedback to the DevTeam (e.g. production requirements)

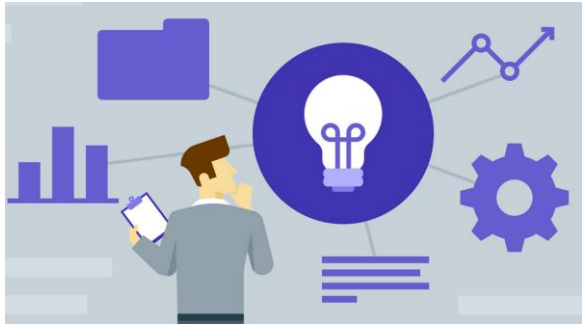
- Acceptance
  - Internal customer
  - Automated end-to-end coverage of use cases
  - Explorative testing

confidential

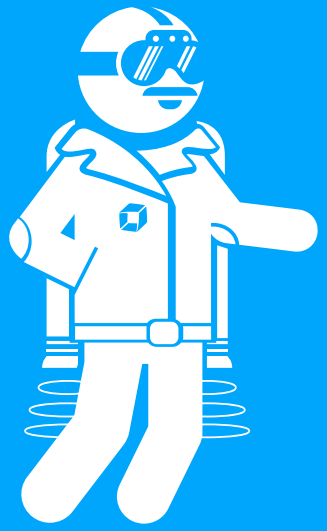
Roles

- Engineering Productivity
  - Supports DevTeam with build, CI environment, CD pipelines, test frameworks

- Infrastructure and Services
  - Provide lab infrastructure (VMs, Container, etc.)
  - Host services (e.g. Artifactory, Jenkins, ..)
- Product Management
  - Gathers requirements from the customers
  - Coordinates Roadmap with other stakeholders
  - Shared product owner role with Team Lead
  - Define functional requirements

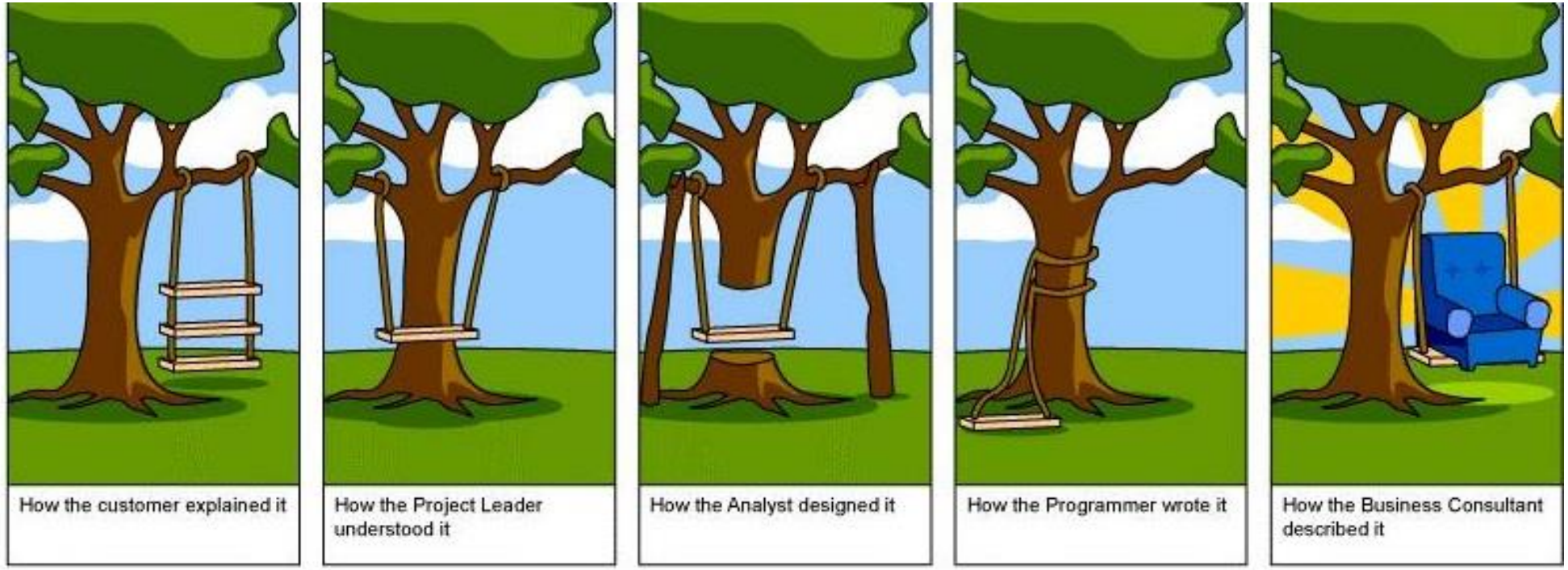


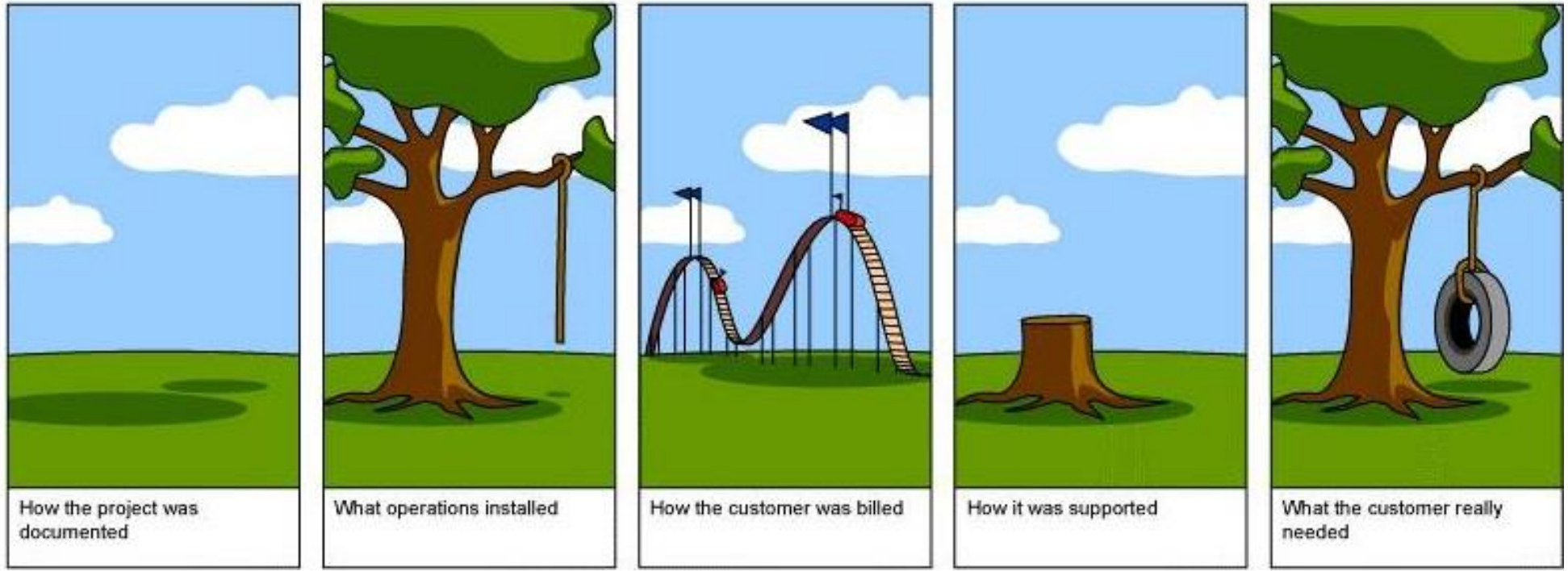
confidential



Challenges in Product Management

confidential





confidential

Project Development

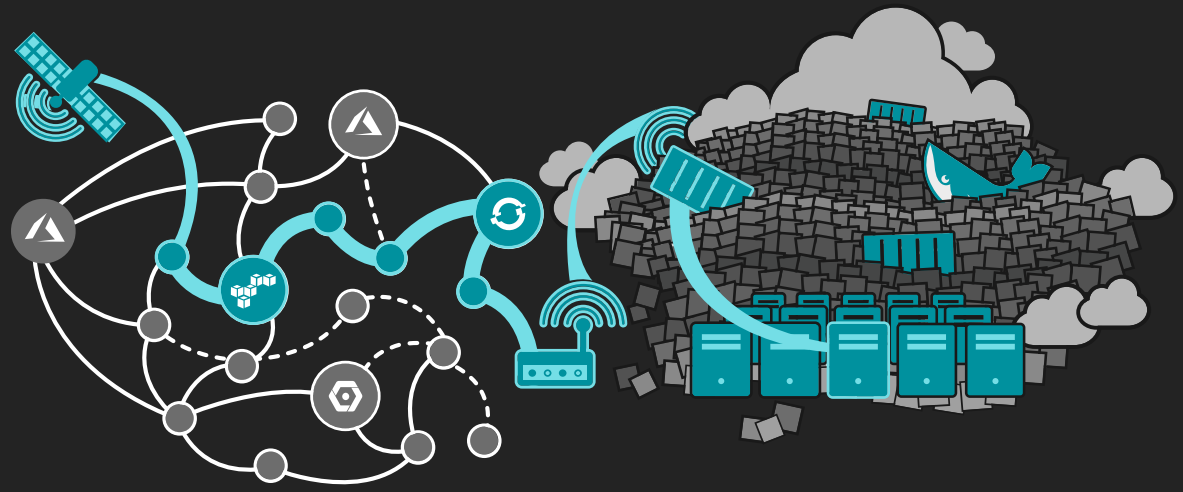
Definition nach DIN 69901:

Vorhaben, das im wesentlichen durch ***Einmaligkeit*** der Bedingungen in ihrer Gesamtheit gekennzeichnet ist, wie z.B. Zielvorgabe, zeitliche, personelle oder andere Begrenzungen, ***Abgrenzung*** gegenüber anderen Vorhaben und eine projektspezifische Organisation.

Kurz: *Einmaliges Vorhaben* einer *zeitlich begrenzten Aufgabenausführung*

confidential

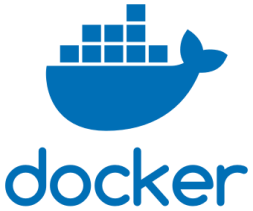
Product development driven by customers and market demand



confidential

confidential

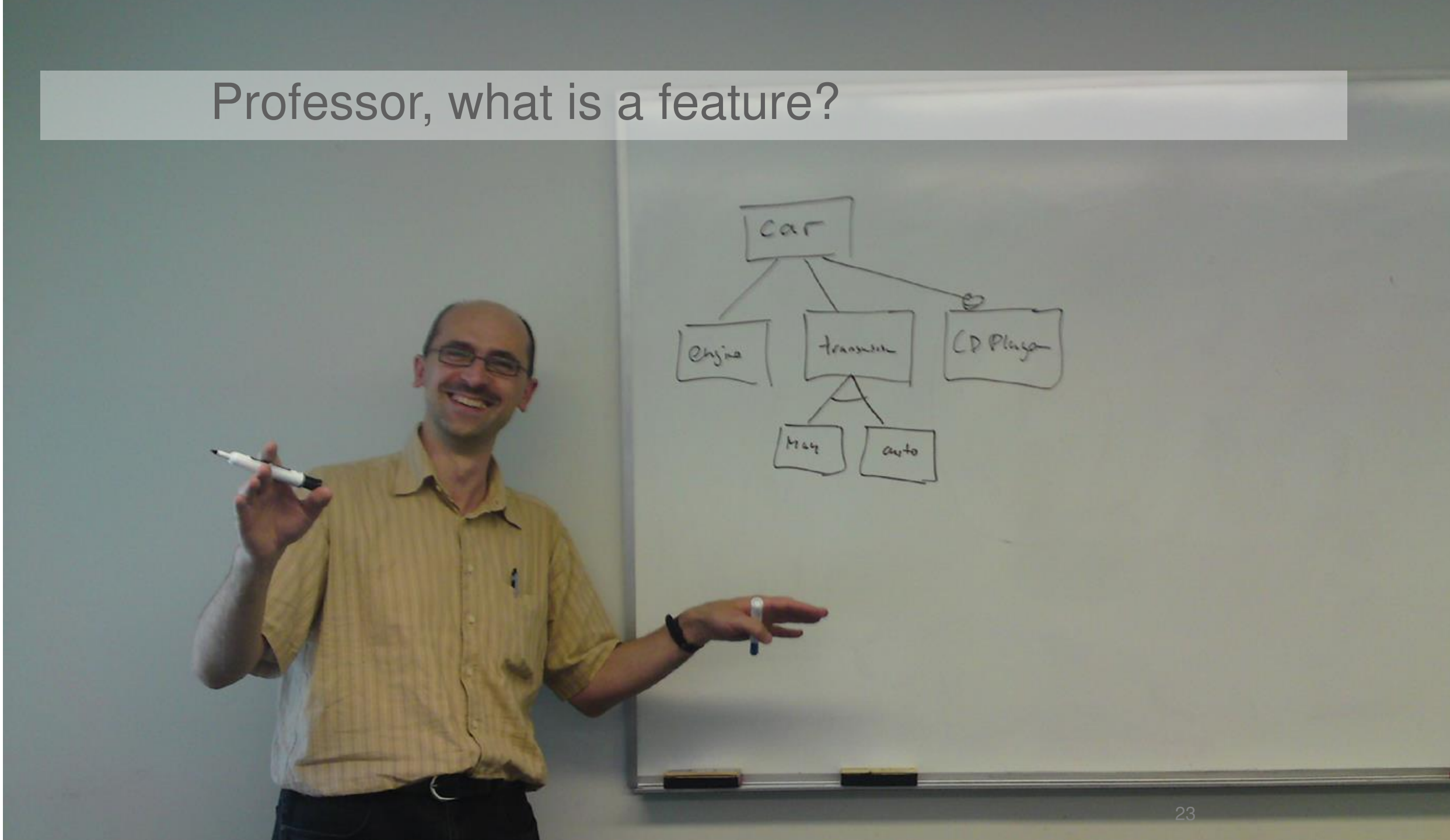
Roadmap



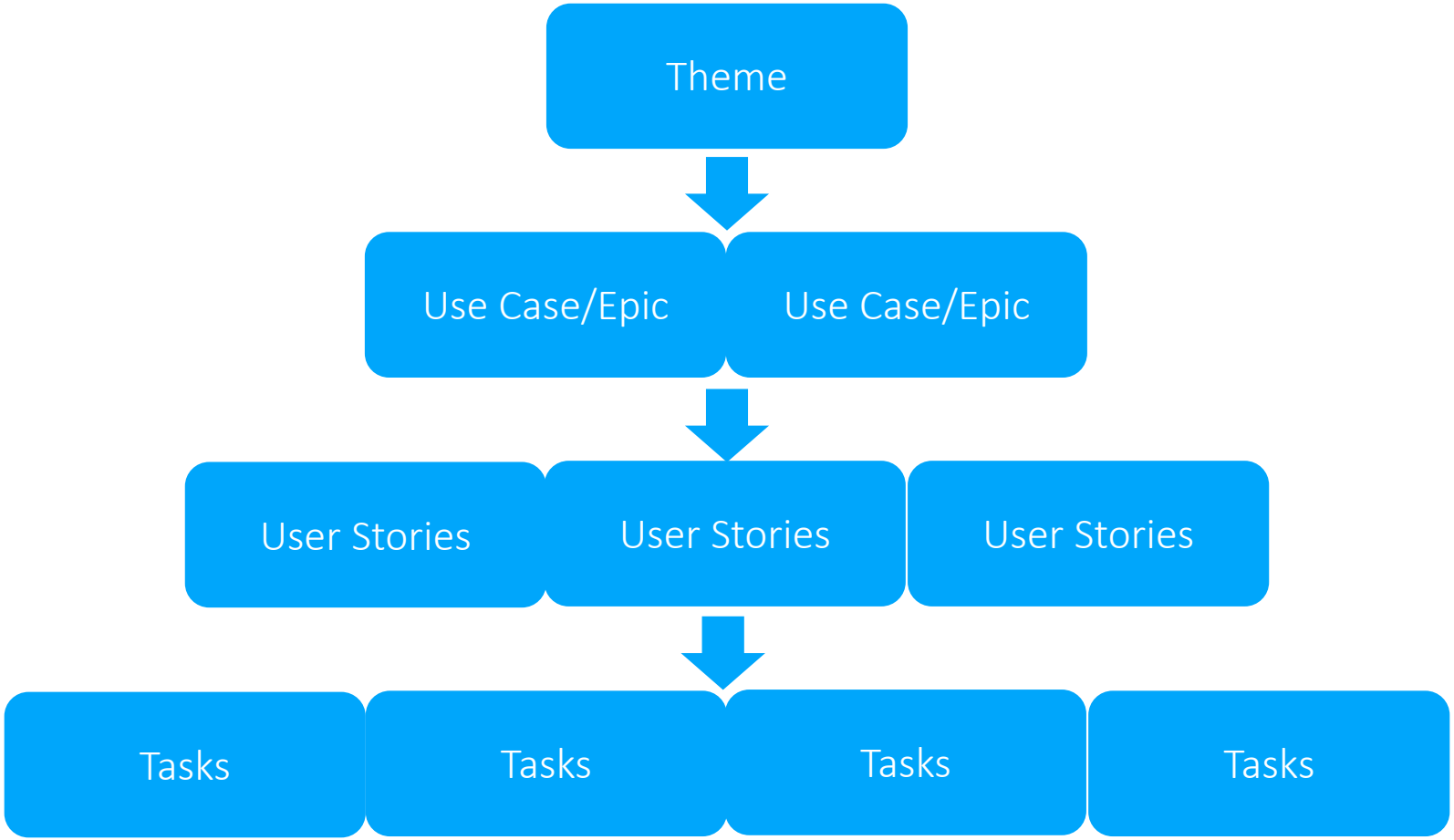
First	Second	Third	Fourth

confidential

Professor, what is a feature?



Roadmap transformation



confidential

Sprint

- 2 weeks cycle
- Sprint Planning
  - PM/PA together with DevTeam
  - Agree on Stories
  - Max 5 Story Points per Story and Engineer
    - Otherwise split up
- Sprint Review
  - Demos on live system
  - Move unfinished stories
- Retrospective
- Releases every two weeks

confidential

Cross Team Collaboration

- 400+ engineers working on one code base
- Standard approach works fine for “smaller” Themes involving few teams
- Larger topics/themes need to have an owner: Epic owner
  - Kick-off
  - Status
  - Team coordination
  - Update of stakeholders
  - Responsible from kick-off to roll out in production

- Large Themes for non-trivial and high-prio cross-team effort over multiple sprints, which requires close collaboration and synchronization
  - Squads (virtual teams)

confidential

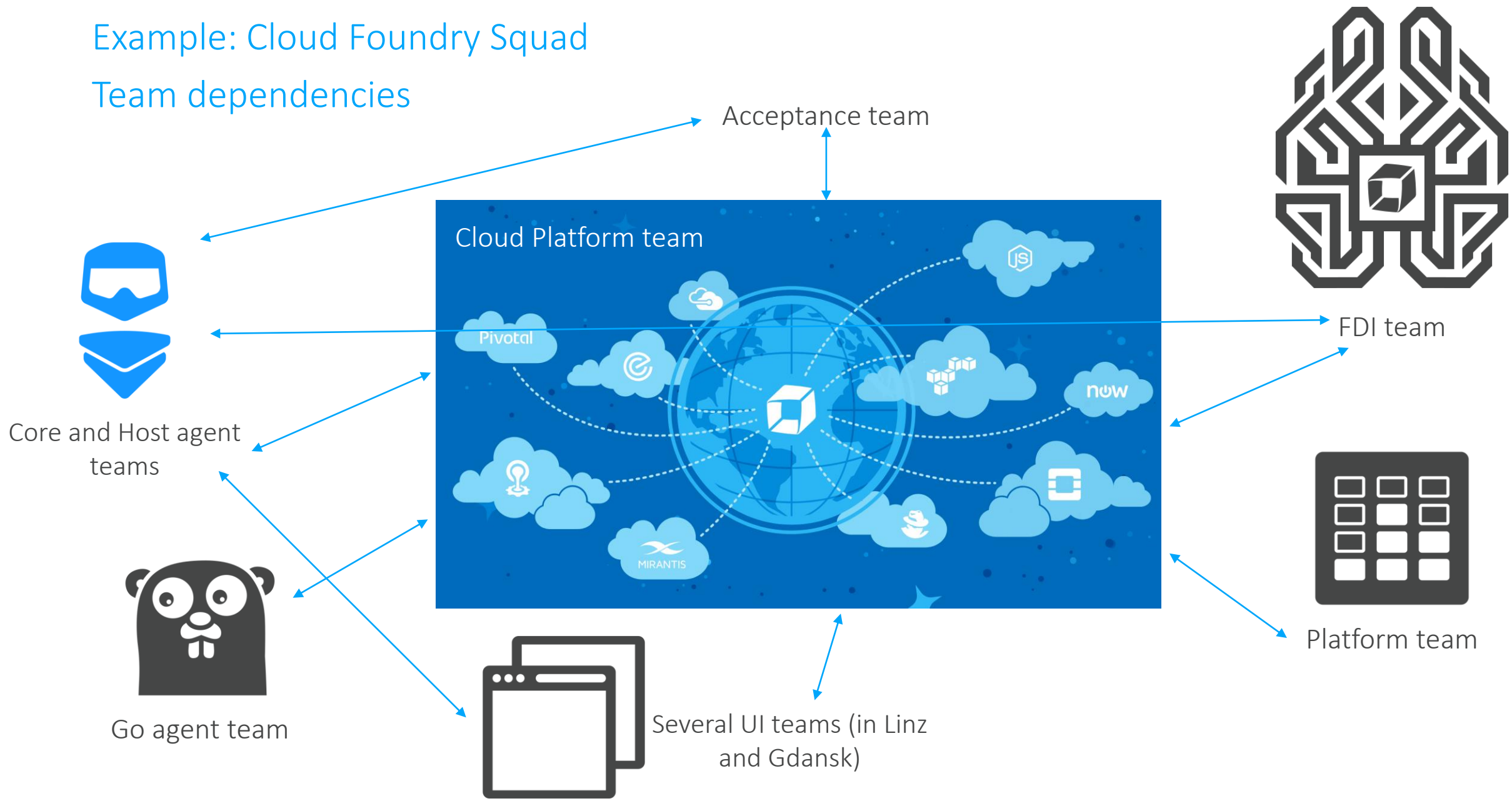
Squads - Roles

- **Squad kickoff group** (PM Leads, DevLeads), signs off squad creation. Ensure management buy-in.
- **Squad lead** (could be TL, Dev, ...), responsible for squad organization (org. and process). Like a TL for the squad scope. Simultaneously also acting as a member in his usual role (e.g. Dev)
- **Squad members** (Dev, PM, CSA,...), getting actual work done. Recruited from multiple teams - for a longer continuous period (but not necessarily the complete squad timeframe)
- **Squad associates** (Dev, EP, AT, Managed...) contributing to the squad goal in a smaller scope, or should be in-sync resp. join the sprint meetings.

confidential

Example: Cloud Foundry Squad

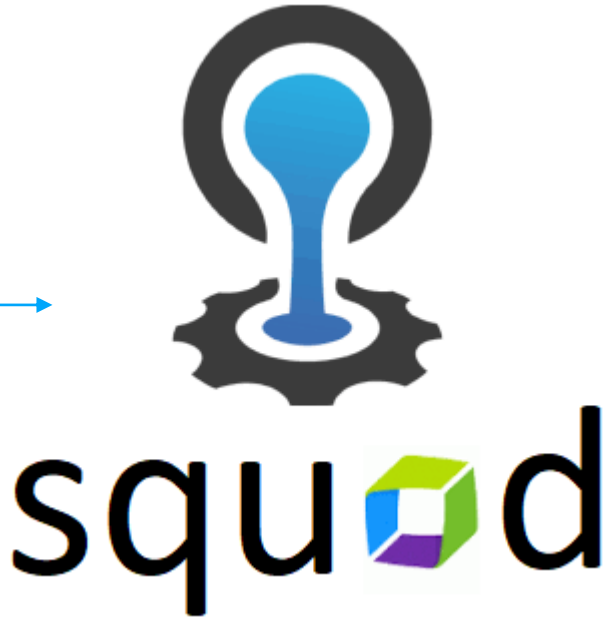
Team dependencies



confidential

Example: Cloud Foundry Squad

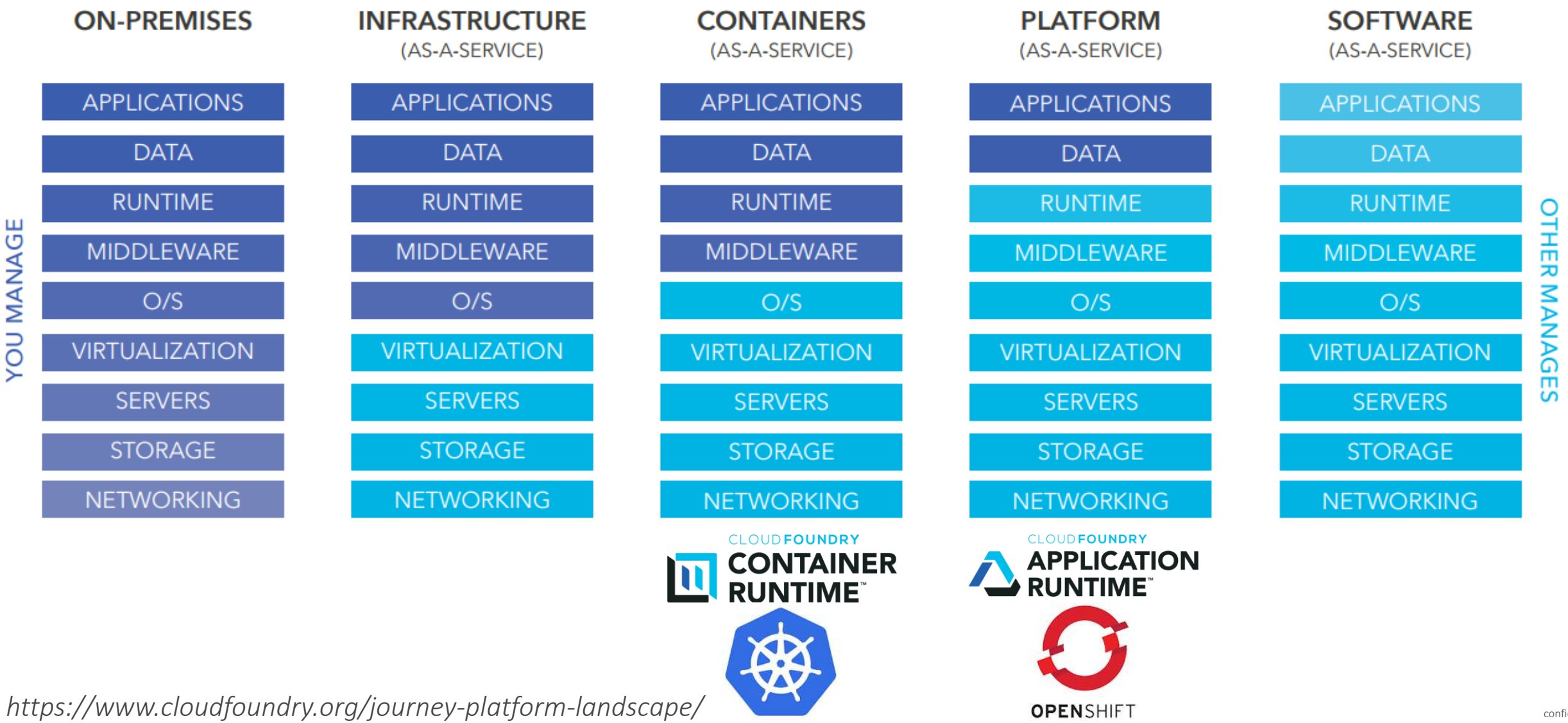
Squad dependencies



CLOUD FOUNDRY

confidential

CaaS vs. PaaS



https://www.cloudfoundry.org/journey-platform-landscape/

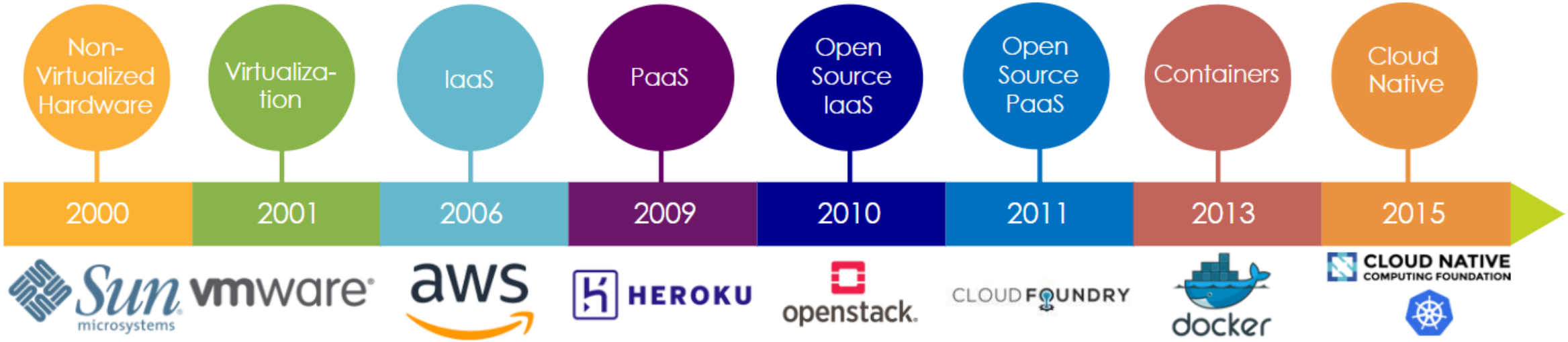
confidential

Cloud Native: CNCF (2015)

CLOUD NATIVE COMPUTING FOUNDATION

kubernetes

- Cloud native computing uses an open source software stack to:
  - segment applications into *microservices*,
  - package each part into its own *container*
  - and dynamically *orchestrate* those containers to optimize resource utilization



CNCF overview, 12/20/17, https://docs.google.com/presentation/d/1BoxFeENJcINgHbKfygXpXROchiRO2LBT-pzdaOFr4Zg/edit#slide=id.g1b3ebf2cb6_0_0

Cloud Foundry Distributions

Certified Distributions 7

Certified Provider distributions use the same core Cloud Foundry software and are portable across cloud application platforms in a multi-vendor, multi-cloud environment.

See All 7 Distribution Partners

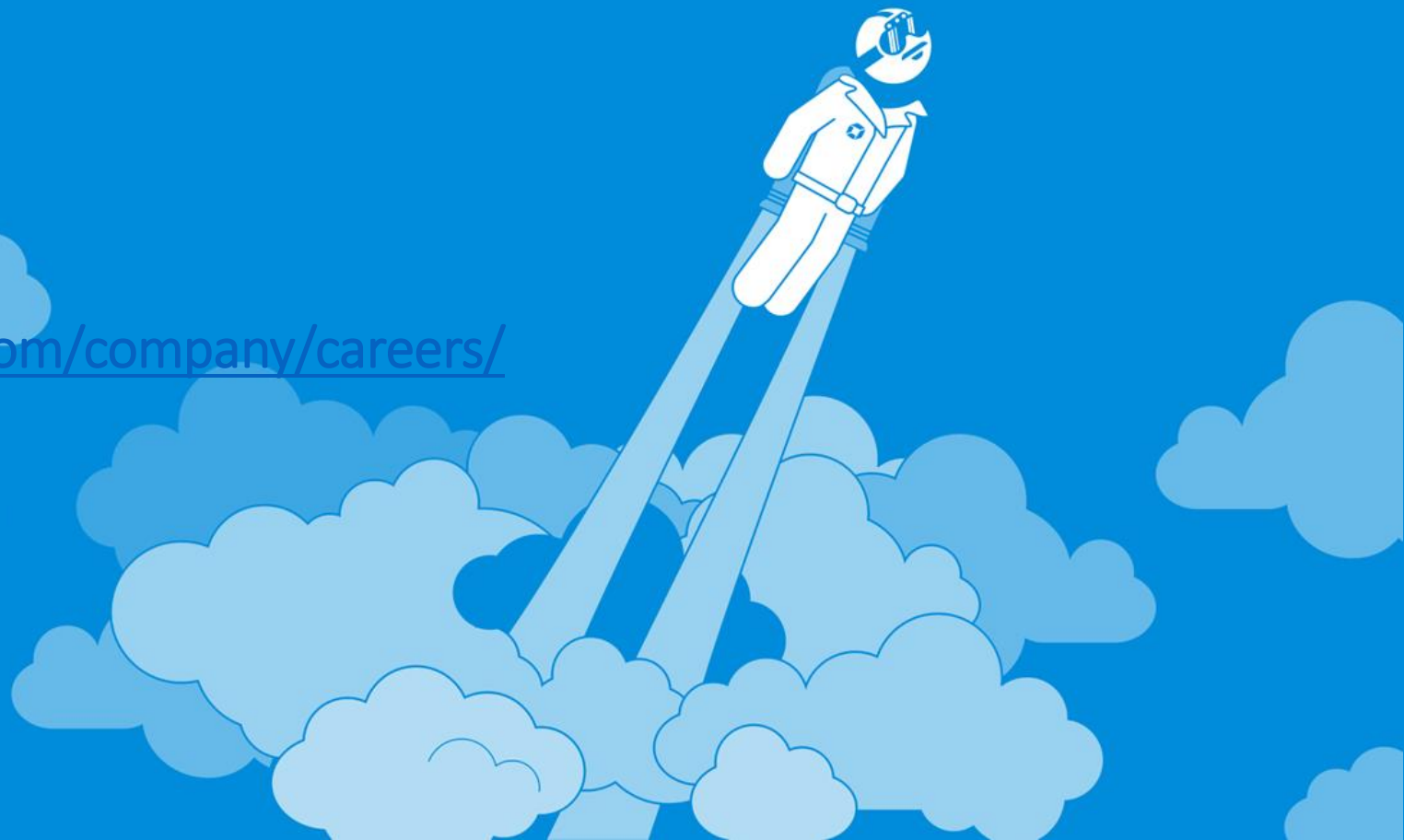
search

https://www.cloudfoundry.org/thefoundry/

confidential

Dynatrace offers

- Master Thesis Topics and Assistance
  - Samples:
    - SuperDump for Linux
    - G1 GC Analyzer
    - Smart Test Selection
- Internships
- Jobs
  - https://www.dynatrace.com/company/careers/



dynatrace

Visit us @ www.dynatrace.com

and start your Free Trial

www.dynatrace.com

confidential

Software Engineering & Projektmanagement VO
(188.410)

23.05.2018 | **Software Patterns**

Felix Rinker
felix.rinker@qse.ifs.tuwien.ac.at

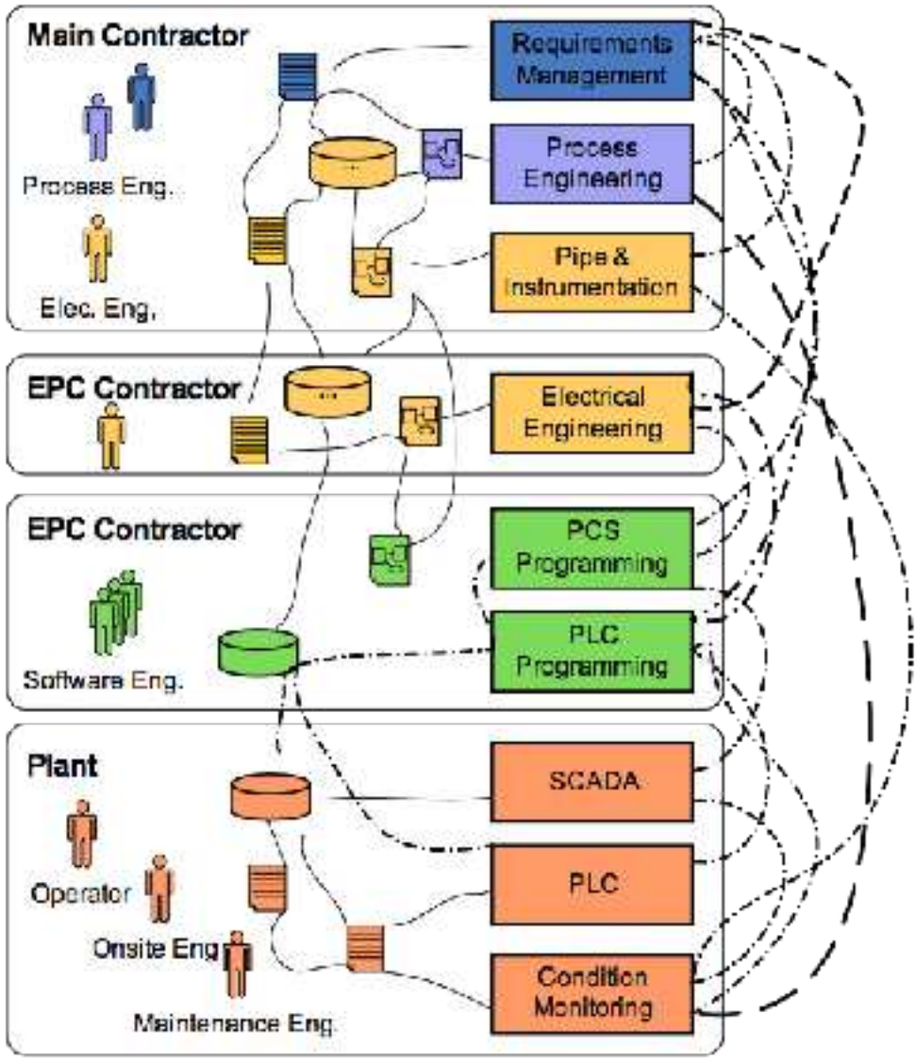
Agenda

- Industrial Use Case
  - Software Engineering Integration for Flexible Automation Systems
- Complex Systems and Complexity Management
- Motivation for Software Patterns
- Software Pattern Categories
- Practical Examples
  - Engineering Service Bus
- Conclusion

Industrial Scenario

- Large-scale engineering project
  - e.g.steel-mill, manufacturing plant engineering, car manufacturing plants, hydro power plants

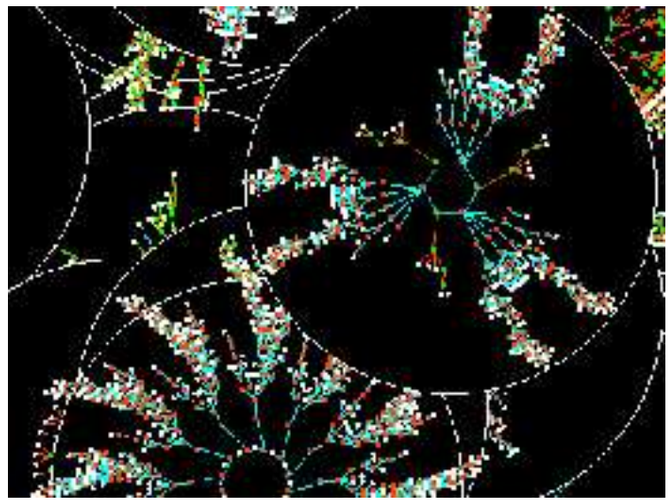
- Cooperation of different engineering disciplines required



- Disciplines have specific engineering tools

- Manual effort needed at for tool data exchange
  - High risks

Complex Systems



- Magnitude
  - Number of Elements in the system
  - Number of possible states of elements
  - Difference between number of possible and usable solutions
- Diversity
  - Magnitude of heterogeneity of elements
- Connectivity, structural complexity
  - Number of potential connections between elements
- Literature defines systems as complex if
  - … they consists of a large number of interacting components,
  - … simple linear modelling is insufficient for understanding,
  - but requires sophisticated dynamic approaches (e.g., simulations).

Managing Complexity

- Abstraction
  - simplification of a scenario
- Decoupling
  - identify the separation of system components that should not depend on each other
- Decomposition
  - KISS - Keep It Simple, Stupid
  - components that are easier to understand, manage, or maintain
  - problem of reassembling
- Classification
  - system parts with similar properties

Managing Complexity

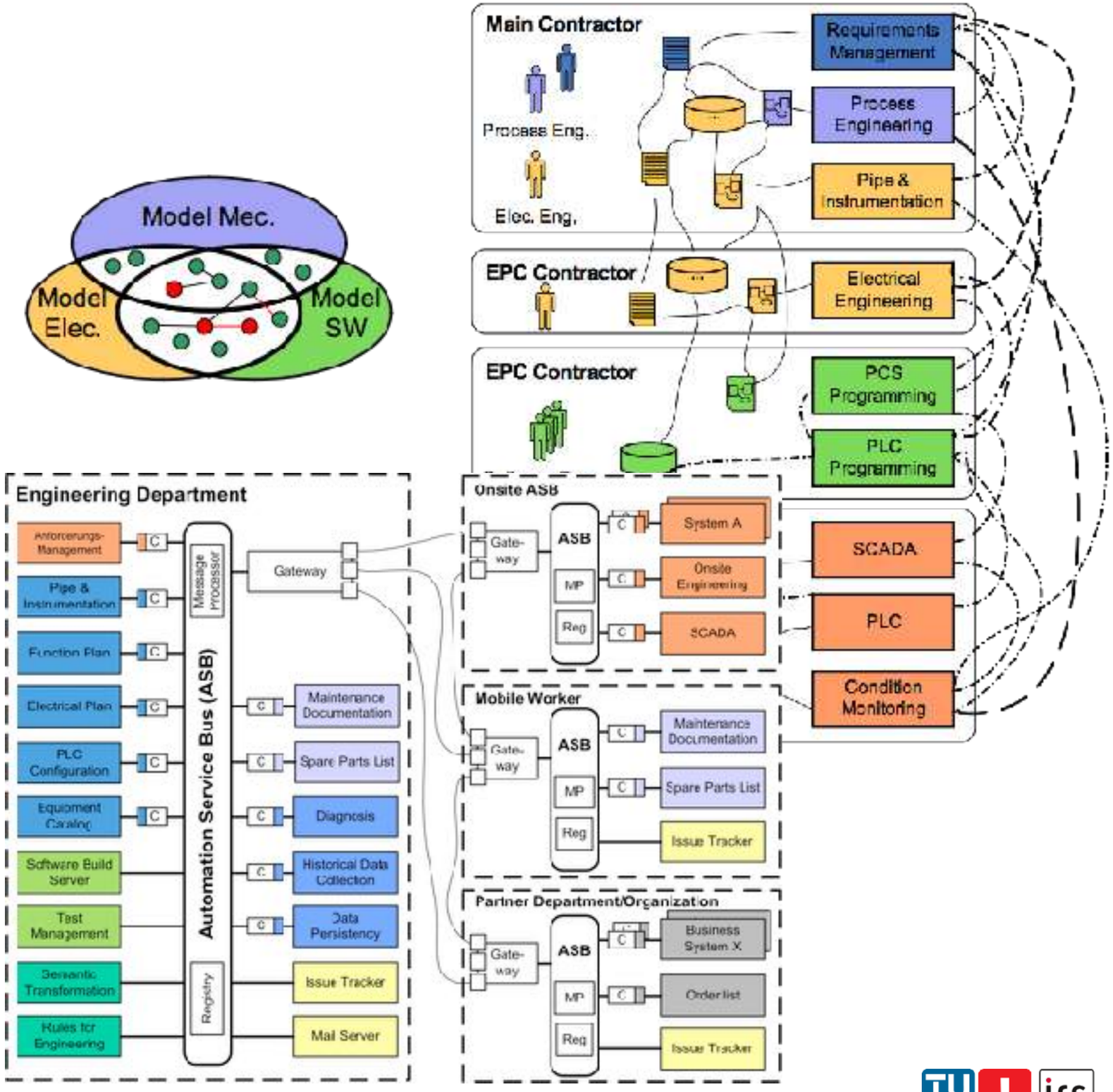
- Standardization
  - benefit of a structured and non-dynamic environment
- Modeling
  - generating an abstract and simplified view
- Transformation
  - transformation of the given problem to a domain with proven solution approach
- Experience
  - documented experiences from experienced contributors

Industrial Scenario

- Complexity-drivers
  - **Technical heterogeneity**
    “Engineering Polynesia”
  - **Semantic heterogeneity**
    “Engineering Babylon”
  - **Process heterogeneity**
    “Engineering Chaos”

- Engineering Service Bus
  (https://github.com/openengsb)

- Operating Numbers
  - 184 repositories
  - 5508 Issues
  - 170k LOC
  - 74k LOConf
  - 314 Project Dependencies



Pattern Definitions

- *„…a solution to a problem in a context…“*
- *„A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts”*
- *„Pattern“ has been defined as „an idea that has been useful in one practical context and will probably be useful in others.“*

Elements of a Pattern

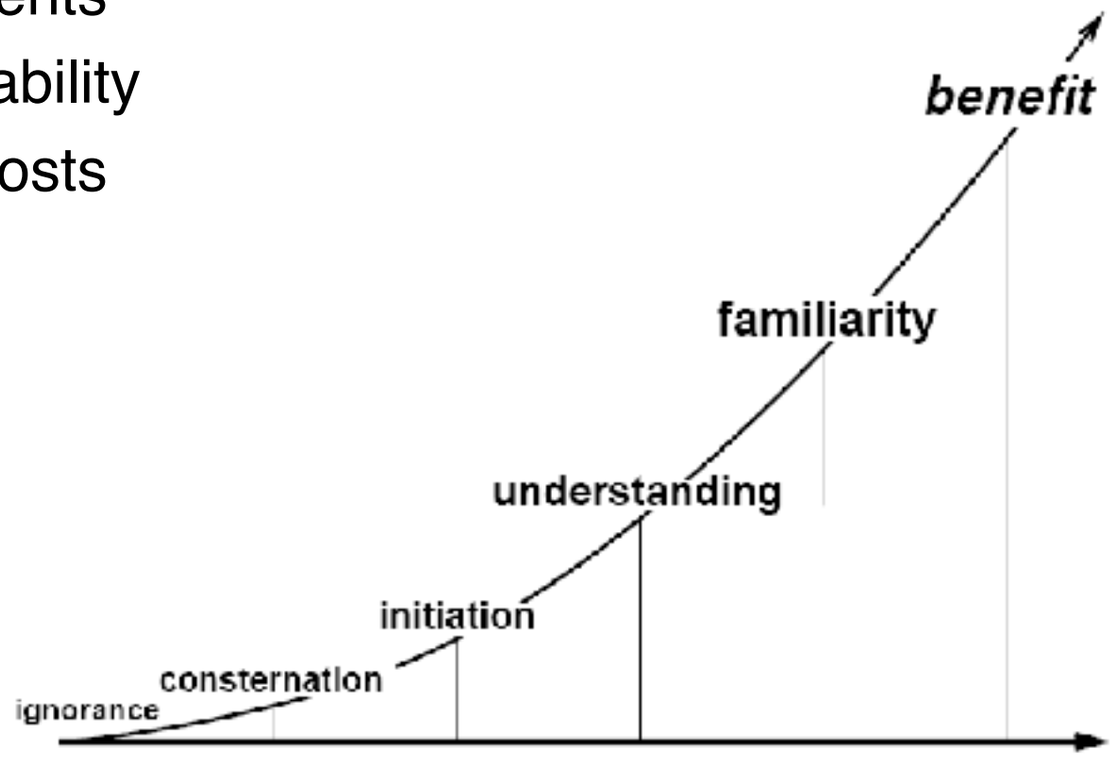
- A meaningful name
  - Aliases, classifications
- Motivation and problem statement
- Context

Elements of a Pattern

- A meaningful name
  - Aliases, classifications
- Motivation and problem statement
- Context
- Solution
  - Structure
  - Participants
  - Collaboration
  - Consequences
  - Implementation
  - Examples

Advantages for Software Development

- Common vocabulary saves discussions
- Help manage complex systems
  - Patterns explicitly capture expert knowledge and design tradeoffs
    - therefore make this expertise more widely available
  - Combination of patterns
- Facilitates non-functional requirements
  - Reusability, adaptability, extendability
- Minimizes development time and costs
- Improves documentation

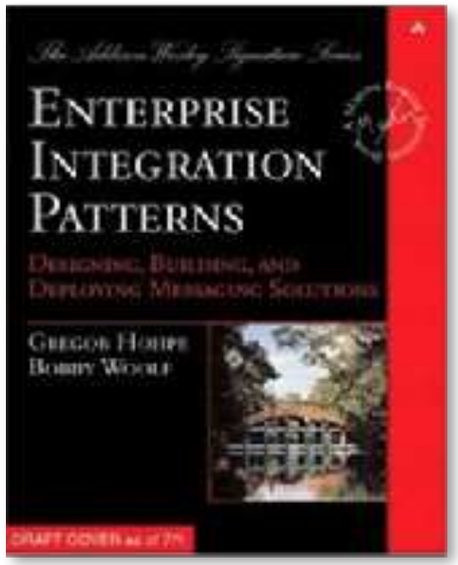


Experience

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
Volume 2
Patterns for Concurrent
and Networked Objects
WILEY

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
Volume 3
Patterns for
Resource Management

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
A Pattern Language for
Distributed Object Computing
Volume 4


ENTERPRISE
INTEGRATION
PATTERNS
Designing, Building, and
Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
On Patterns and Pattern Languages
Volume 5

SECURITY
PATTERNS
Integrating Security
and Systems Engineering

Drawbacks of Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion
  - rather than by automated testing
  - http://clean-code-developer.de/

Classification of Patterns

- Architectural Patterns
  - Structure of software systems
  - Subsystems, dependencies, communication
- Design Patterns
  - Describes the structure and relations at the level of classes
- Idioms
  - Focus on low-level details
  - Programming language specific
- Protopatterns
  - Particular case
  - A new, understandable solution to be used in larger scale
- Antipatterns
  - Commonly used but ineffective techniques

When to use Patterns

- Solutions to problems that recur with variations
  - No need for reuse if the problem only arises in one context
- Solutions that require several steps
  - Patterns can be overkill if solution is simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really wants to understand

Most popular Patterns

- The most popular design pattern is the Interface pattern
- The second most popular design pattern is Proxy Pattern
- The third most popular design pattern is "Big Ball of Mud"

http://dilbert.com/strips/comic/1994-06-10/

Types of Patterns

- Fundamental patterns
  - Deal with essential concepts of software architecture

- Creational patterns
  - Deal with initializing and configuring classes and objects

- Structural patterns
  - Deal with decoupling interface and implementation of classes and objects

- Behavioral patterns
  - Deal with dynamic interactions among objects

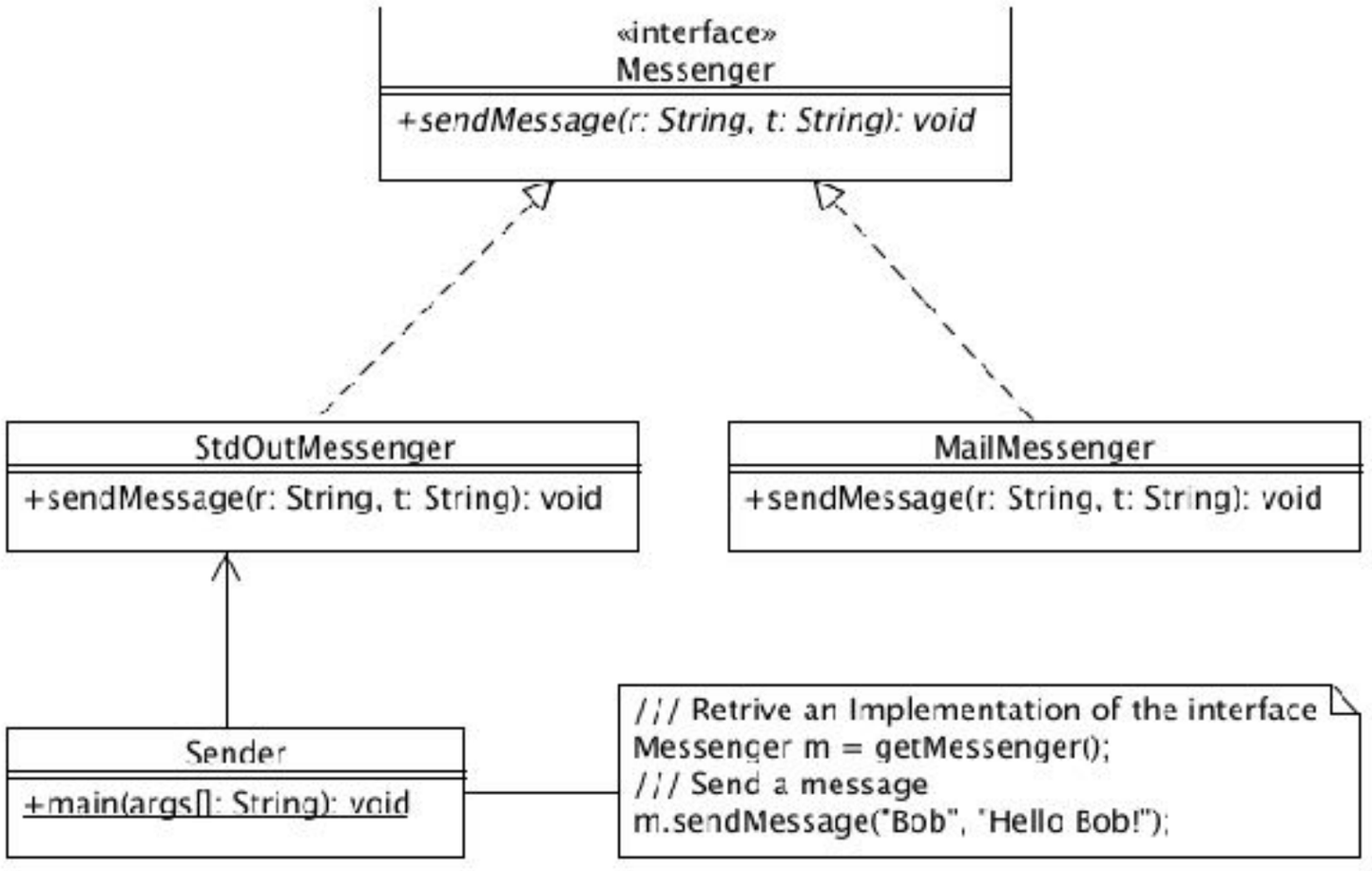
Fundamental Patterns - Overview

- Interface
  - Separation of interface description and implementation
- Delegation
  - Extension of functionality without inheritance
- Immutable
  - Provides unchangeable object after initialization
- Marker / Annotation
  - Enhances objects with metadata

Fundamental Pattern - Interface

Issue: Separate Interface description and concrete implementation

- defines the signature operations of an entity
- should be stable - in comparison to implementation
- implementations can be added / changed easily



Fundamental Pattern - Interface

Code Example

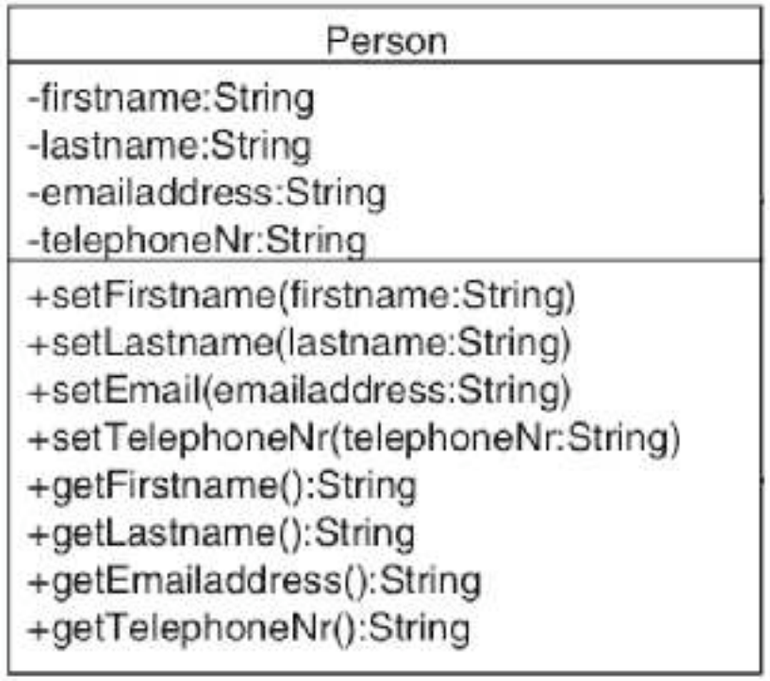
```java
package org.openengsb.domain.notification;

import org.openengsb.core.api.Domain;

// @extract-start NotificationDomain
public interface NotificationDomain extends Domain {

    void notify(Notification notification);

}
// @extract-end
```

```java
public class EmailNotifier extends AbstractOpenEngSBConnectorService implements N

private static final Logger LOGGER = LoggerFactory.getLogger(EmailNotifier.cl

private final MailAbstraction mailAbstraction;
private ServiceRegistration serviceRegistration;
private MailProperties properties;

public EmailNotifier(String instanceId, MailAbstraction mailAbstraction) {
    super(instanceId);
    this.mailAbstraction = mailAbstraction;
}

@Override
public void notify(Notification notification) {
    LOGGER.info("notifying {} via email...", notification.getRecipient());
    LOGGER.info("Subject: {}", notification.getSubject());
    LOGGER.info("Message: {}", StringUtils.abbreviate(notification.getMessage
    mailAbstraction.send(properties, notification.getSubject(), notification.
            .getRecipient());
    LOGGER.info("mail has been sent");
}
```

```java
public class FacebookNotifier extends AbstractOpenEngSBConnectorService implements NotificationDomain {
    private static final Logger LOGGER = LoggerFactory.getLogger(FacebookNotifier.class);
    private ServiceRegistration serviceRegistration;
    private FacebookProperties properties;
    private AliveState aliveState = AliveState.DISCONNECTED;

    public FacebookNotifier(String instanceId) {
        super(instanceId);
        properties = new FacebookProperties();
    }

    @Override
    public void notify(Notification notification) {
        LOGGER.info("Message: {}", StringUtils.abbreviate(notification.getMessage(), 200));
        send(notification.getMessage());
        LOGGER.info("facebook message has been sent");
    }

    @Override
    public AliveState getAliveState() {
        return aliveState;
    }

    public void send(String textContent) {
        try {
            aliveState = AliveState.CONNECTING;
            String httpsURL =
                "https://graph.facebook.com/" + properties.getUserID() + "/feed?access_token="
                        + properties.getUserToken();
            String params = "&message=" + textContent;
            String entryId = sendData(httpsURL, params);
            LOGGER.info("created wall entry with id \"{}\"", entryId);
            aliveState = AliveState.ONLINE;
        } catch (Exception e) {
            aliveState = AliveState.OFFLINE;
            throw new DomainMethodExecutionException(e);
        }
    }
```

Fundamental Pattern - Delegation

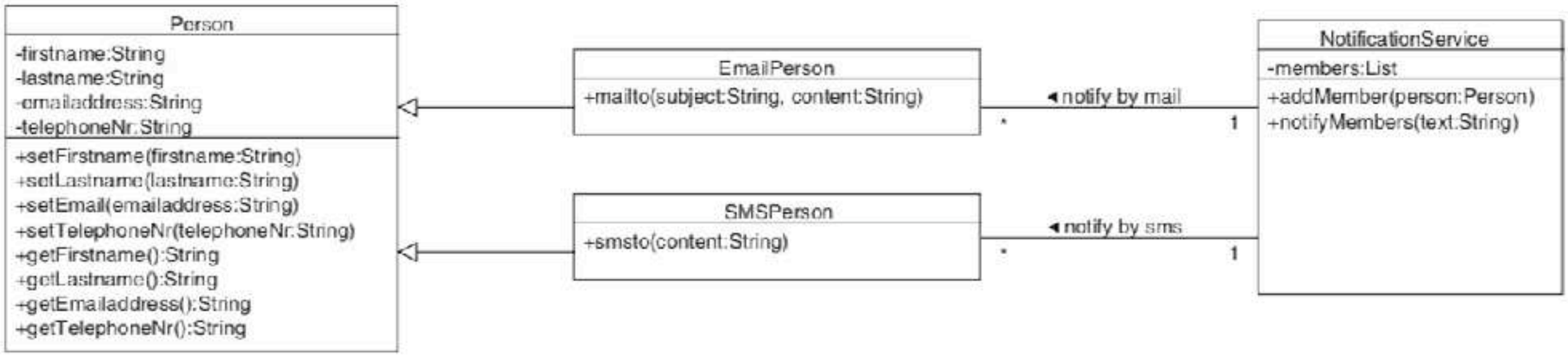
Issue: Class needs additional functionality



Fundamental Pattern - Delegation

Issue: Class needs additional functionality

Inheritance

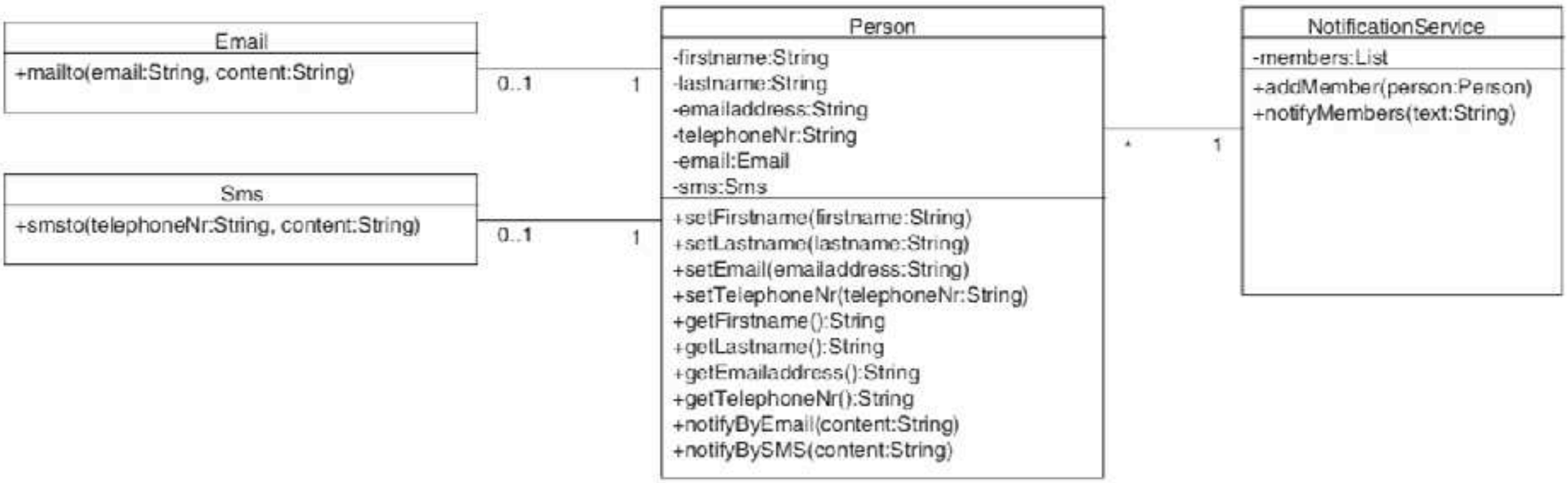


Fundamental Pattern - Delegation

Issue: Class needs additional functionality

Delegation

Outsource functionality into third class and use its instance via delegation



Fundamental Pattern - Delegation

Code Example

```
    }

    @Override
    public Object handleInvoke(Object proxy, Method method, Object[] args) throws IllegalAccessException,
        InvocationTargetException {
        checkMethod(method);
        forwardEvent((Event) args[0]);
        return null;
    }

    private void forwardEvent(Event event) throws InvocationTargetException {
        LOGGER.info("Forwarding event to workflow service");
        try {
            workflowService.processEvent(event);
        } catch (WorkflowException e) {
            throw new InvocationTargetException(e);
        }
    }

    private void checkMethod(Method method) {
        if (method.getParameterTypes().length != 1) {
            throw new EventProxyException(
                "Event proxy can only handle methods named raiseEvent where the first parameter is of type Event, "
                        + "but encountered invocation of method raiseEvent without parameter. Method: " + method);
        } else if (!Event.class.isAssignableFrom(method.getParameterTypes()[0])) {
            throw new EventProxyException(
                "Event proxy can only handle methods named raiseEvent where the first parameter is of type Event, "
                        + "but encountered invocation of method raiseEvent where first parameter is no Event. Method: "
                        + method);
        }
    }
```

Fundamental Pattern - Immutable

Issue: Object instance should be immutable

- Several threads accessing same object
- Configuration object properties

Immutable Object

- Initialize variables in constructor
- Provide readable only access via Getter-Methods

Fundamental Pattern - Immutable

Code Example

```java
package org.openengsb.core.ekb.internal;

/**
 * Helper class for easier working with the informations that define a connector: domainId, connectorId and instanceId.
 */
public class ConnectorInformation {
    private String domainId;
    private String connectorId;
    private String instanceId;

    public ConnectorInformation(String domainId, String connectorId, String instanceId) {
        this.domainId = domainId;
        this.connectorId = connectorId;
        this.instanceId = instanceId;
    }

    public String getDomainId() {
        return domainId;
    }

    public String getConnectorId() {
        return connectorId;
    }

    public String getInstanceId() {
        return instanceId;
    }
}
```

Creational Patterns - Overview

- Singleton
  - Provision of a single instance only
- Factory
  - Method in a derived class creates associates
- Abstract Factory
  - Factory for building related objects without specifying their concrete classes
- Builder
  - Factory for building complex objects in different variants
- Prototype
  - Factory for cloning new instances from a prototypical instance

Creational Pattern - Singleton

Issue: Only one object instance should exist

- Database access
- Id generator
- Logger
- Communication with hardware

Creational Pattern - Singleton

Issue: Only one object instance should exist

Singelton

```
public class NoficationManagerService {
  private NoficationManagerService() {
  }

  private static NoficationManagerService
                 notmanservice = null;

  public static NoficationManagerService
                getInstance() {
    if (notmanservice == null) {
     notmanservice = new NoficationManagerService();
    }
    return notmanservice;
  }
  ...
}
```

Threadsafe??

Creational Pattern - Singleton

Code Example

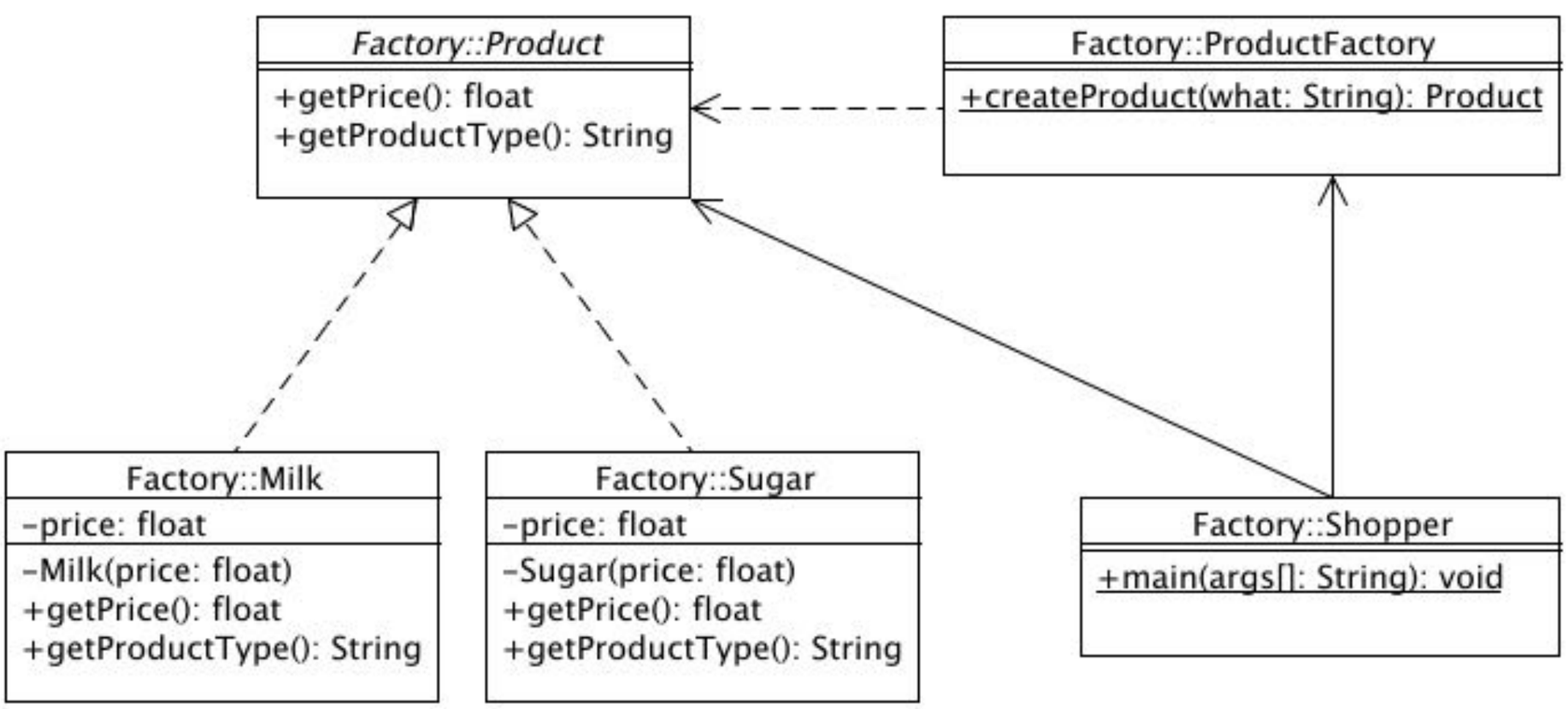
```
package org.openengsb.core.api.context;

/**
 * Singleton Class, that provides access to thread-local contextual attributes
 */
public final class ContextHolder {

    private static ContextHolder instance = new ContextHolder();

    private ThreadLocal<String> currentContextId = new InheritableThreadLocal<String>();

    /**
     * returns the singleton instance
     */
    public static ContextHolder get() {
        return instance;
    }

    /**
     * set the current Threads context id (it is inherited by threads spawned by the current process)
     */
    public void setCurrentContextId(String value) {
        currentContextId.set(value);
    }

    /**
     * get the current Threads context id
     */
    public String getCurrentContextId() {
        return currentContextId.get();
    }

    private ContextHolder() {
    }

}
```

Creational Pattern - Factory

Issue: Object creation depends on complex requirements

- Initialization of additional sub-instances required
- Complex configuration process steps

Helps decoupling as only interface is known!



Creational Pattern - Factory

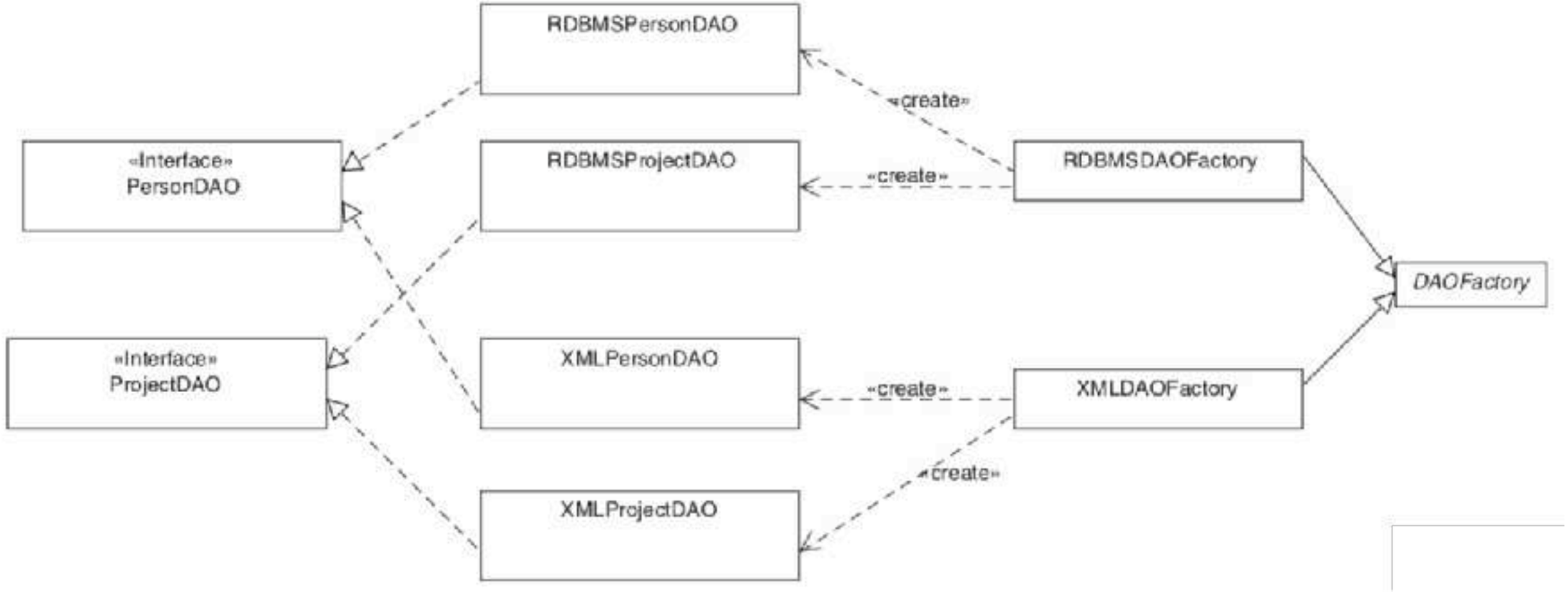
Code Example

```
public interface ConnectorInstanceFactory {

    /**
     * validates the attribute combination. This is used for validating attributes before creating a new service.
     * Therefore the supplied map contains all attributes
     *
     * returns a Collection of error-messages. should return an empty map if there are no errors
     */
    Map<String, String> getValidationErrors(Map<String, String> attributes);

    /**
     * validates the attribute combination. This is used for validating attributes before updating new service.
     * Therefore the supplied map contains only the attributes that should be changed. Other attributes must be
     * retrieved from the instance directly.
     *
     * returns a Collection of error-messages. should return an empty map if there are no errors
     */
    Map<String, String> getValidationErrors(Connector instance, Map<String, String> attributes);

    /**
     * creates a new instance with the given service-id. The serviceId should then be the the same as returned by
     * {@link OpenEngSBService#getInstanceId()}
     *
     * The created instance only contains default-values that are changed later.
     */
    Connector createNewInstance(String id);

    /**
     * This method is used for filling in the attributes of a service. It can be assumed that the attributes have been
     * validated before.
     *
     */
    void applyAttributes(Connector instance, Map<String, String> attributes);
}
```

Creational Pattern – Abstract Factory

Issue: Achieving higher abstraction by grouping individual factories with a common theme

- Abstract Factory
  - a group of individual factories that have a common theme
- Two hierarchies
  - various abstractions client is interested in
  - abstract AbstractFactory class provides interface
    - for each class that is responsible for creating the members of a particular family
- Client only knows abstract interface
  - Family may grow independently of the client



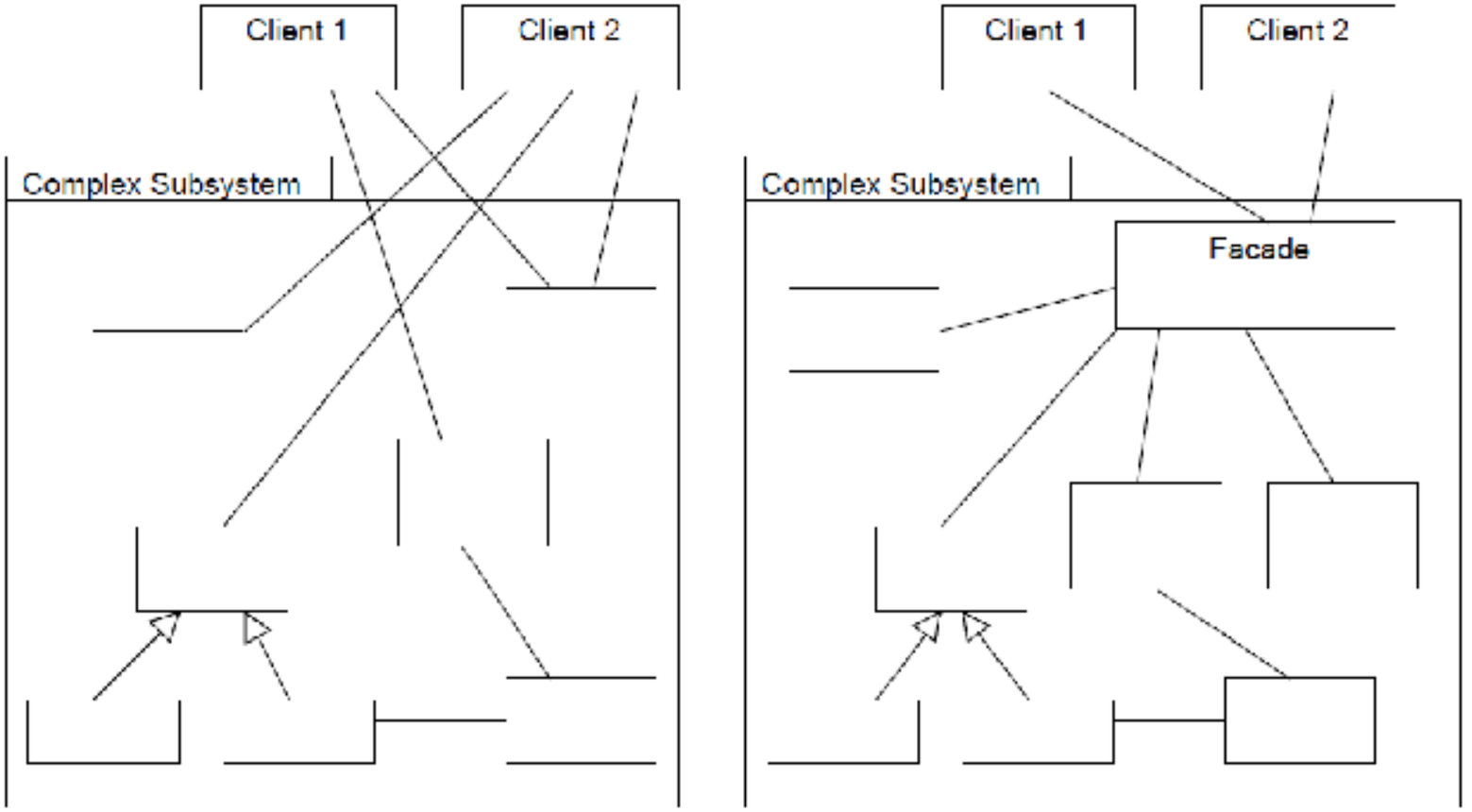
Structural Patterns - Overview

- Facade
  - Facade simplifies the interface for a subsystem
- Adapter
  - Translator adapts a server interface for a client
- Proxy
  - One object approximates another
- Bridge
  - Abstraction for binding one of many implementations
- Composite
  - Treats individual objects and compositions uniformly
- Flyweight
  - Many fine-grained objects shared efficiently

Structural Pattern - Facet

Issue: Need simplified access to a complex subsystem

- Provides a simplified, higher-level interface of a subsystem
  - easier to use, understand, and test subsystem
  - balance between simple but restricted and rich but complex
- May help creating a layered architecture



Structural Pattern - Facet

Code Example

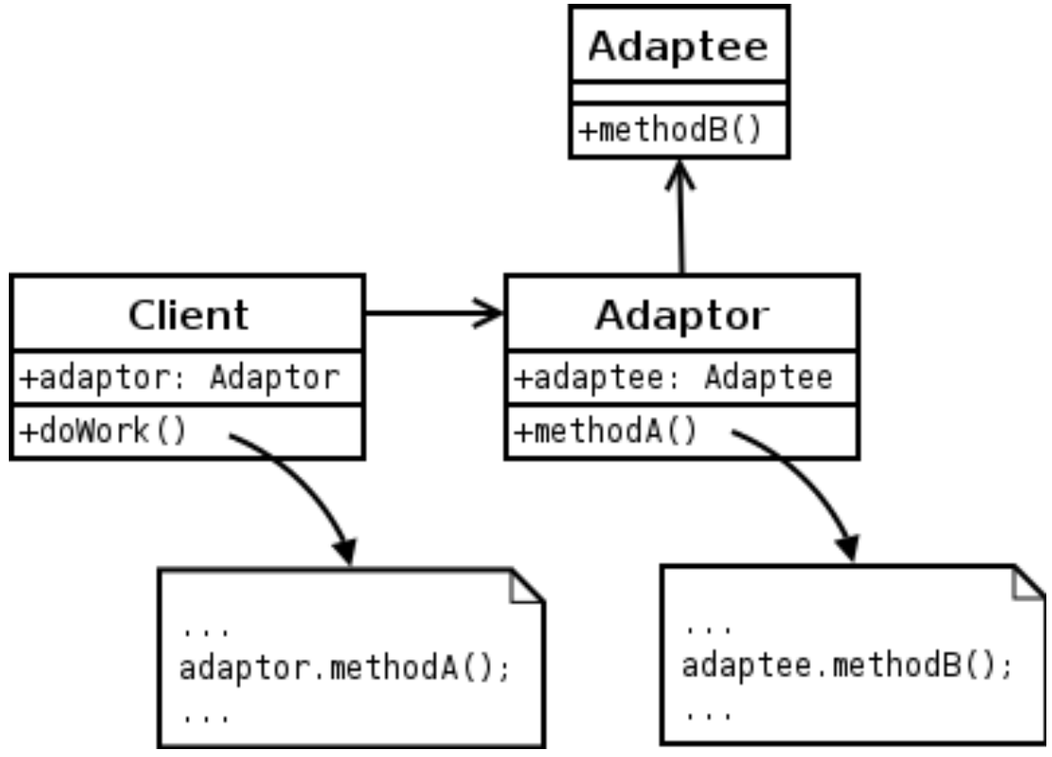
```
class SimpleMail {
  public static void sendMail(String address,
                              String subject,
                              String body) {
    // hier wird der Zugriff auf die JavaMail API
    // implementiert und mit verschiedenen
    // Klassen dieser API interagiert.
}
```

Structural Pattern - Adapter

Issue: Need to integrate incompatible external functionality

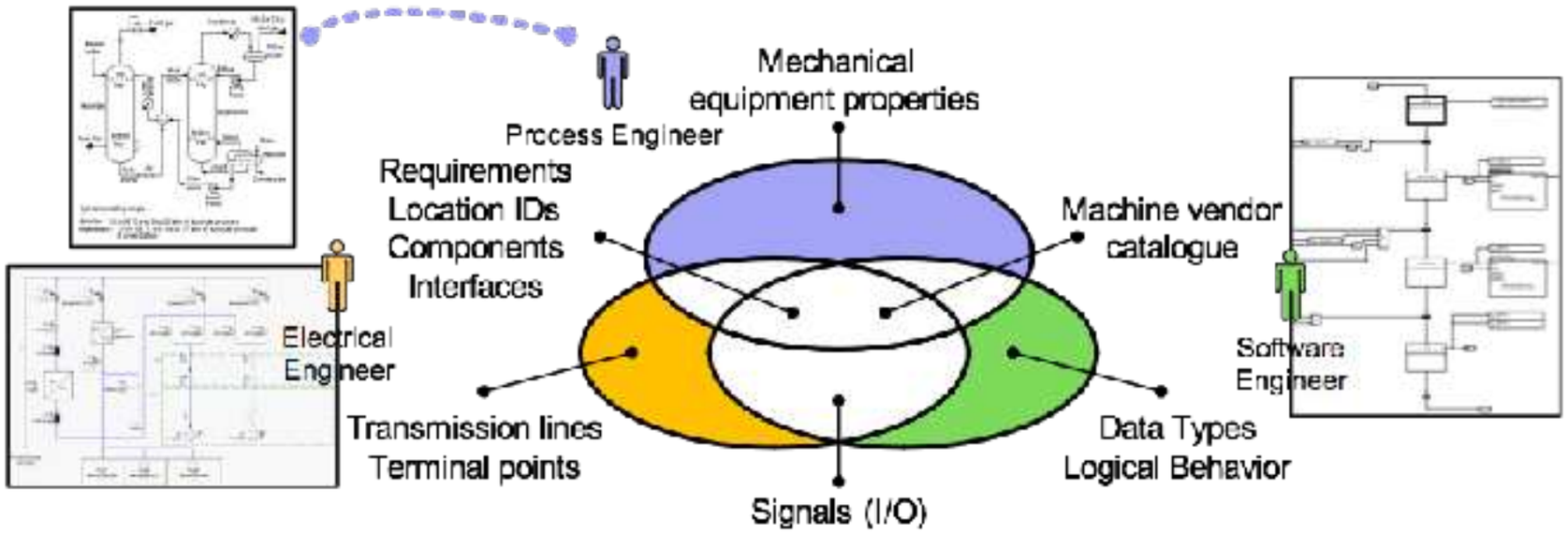
- wrapper pattern or simply a wrapper
- provides access to external functionality
  - e.g., access to external libraries, (proprietary) systems
  - typically no direct access because of incompatible interfaces

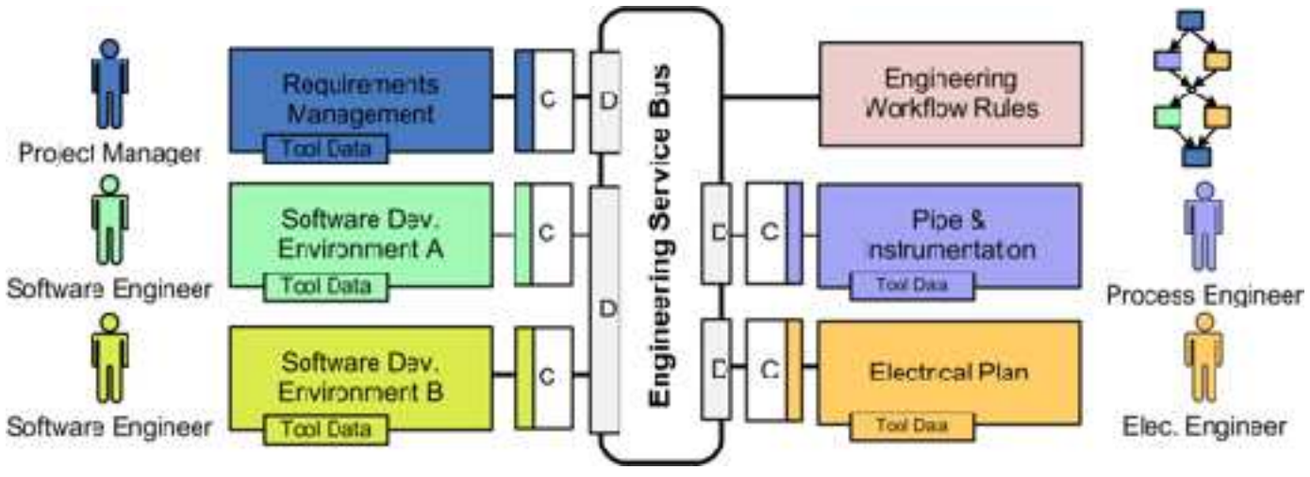
- translates an external interface into a compatible interface
  - Perform data transformations into appropriate forms



Structural Pattern - Adapter

OESB Example



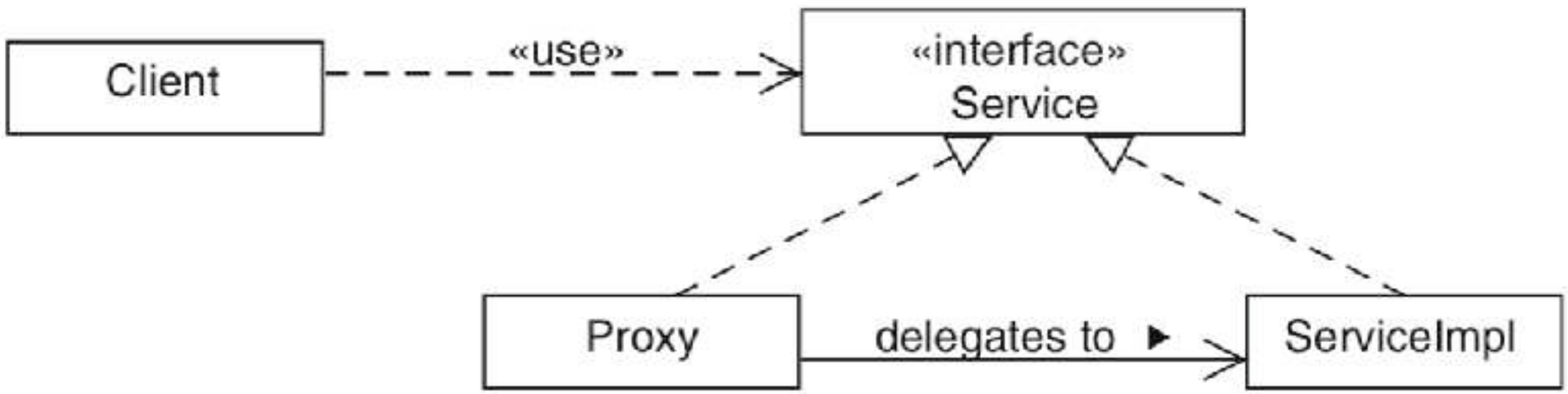


Structural Pattern - Proxy

Issue: Need to integrate further actions before intended method call

- Extends concept of the delegation pattern
- Enriches interface functionality
  - Implements interface and acts as a representative of the „original“ implementation
- Cascading Proxies

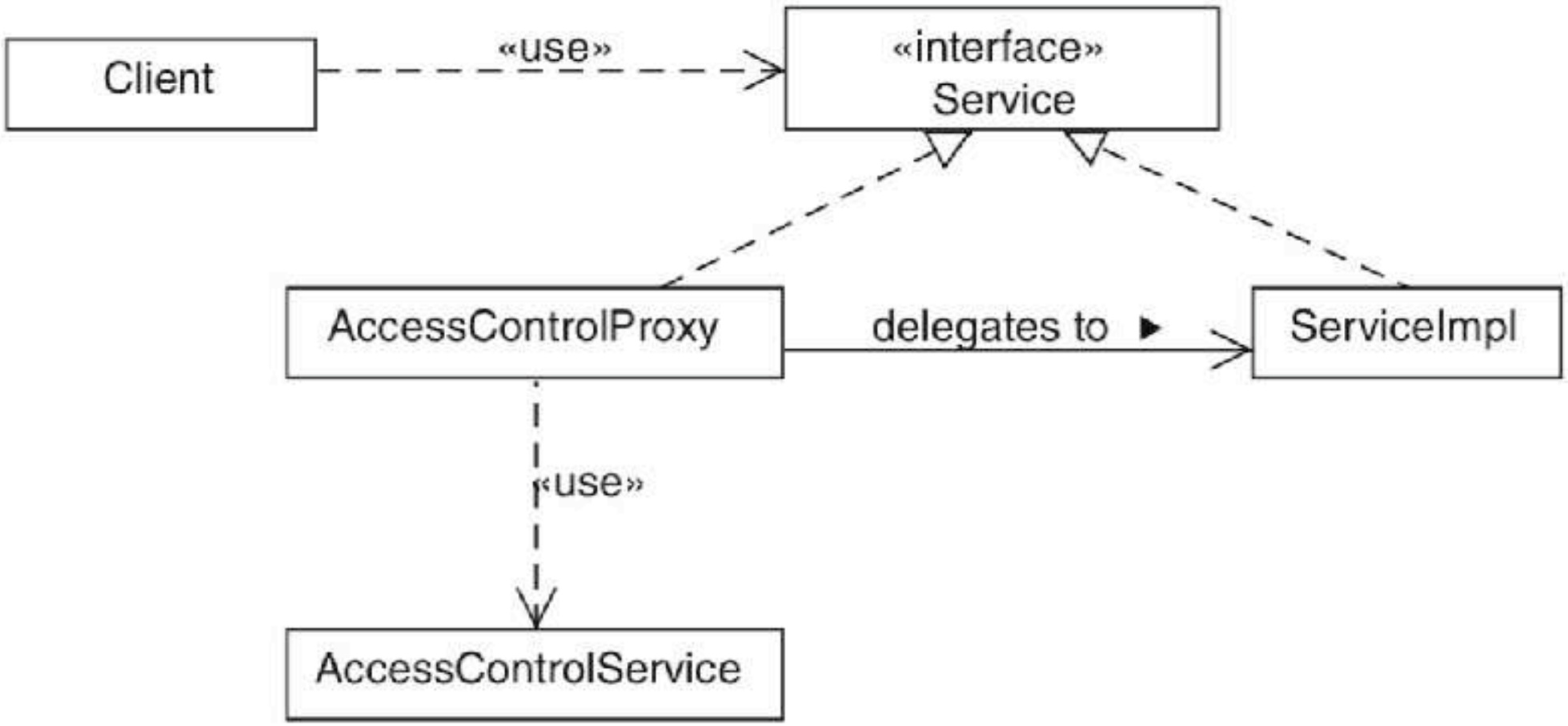
- Use cases
  - security
  - logging
  - caching



Structural Pattern - Proxy

Issue: Need to integrate further actions before intended method call

Use case: **Access control**



Remote Connectors

Code Example

```
@Override
public Object doInvoke(Object proxy, Method method, Object[] args) throws Throwable {
    List<String> paramTypeNames = getParameterTypesAsStrings(method);

    MethodCall methodCall = new MethodCall(method.getName(), args, metadata, paramTypeNames);
    MethodResult callResult = portService.sendMethodCallWithResult(portId, destination, methodCall);

    switch (callResult.getType()) {
        case Object:
            return callResult.getArg();
        case Void:
            return null;
        case Exception:
            throw new RuntimeException(callResult.getArg().toString());
        default:
            throw new IllegalStateException("Return Type has to be either Void, Object or Exception");
    }
}
```

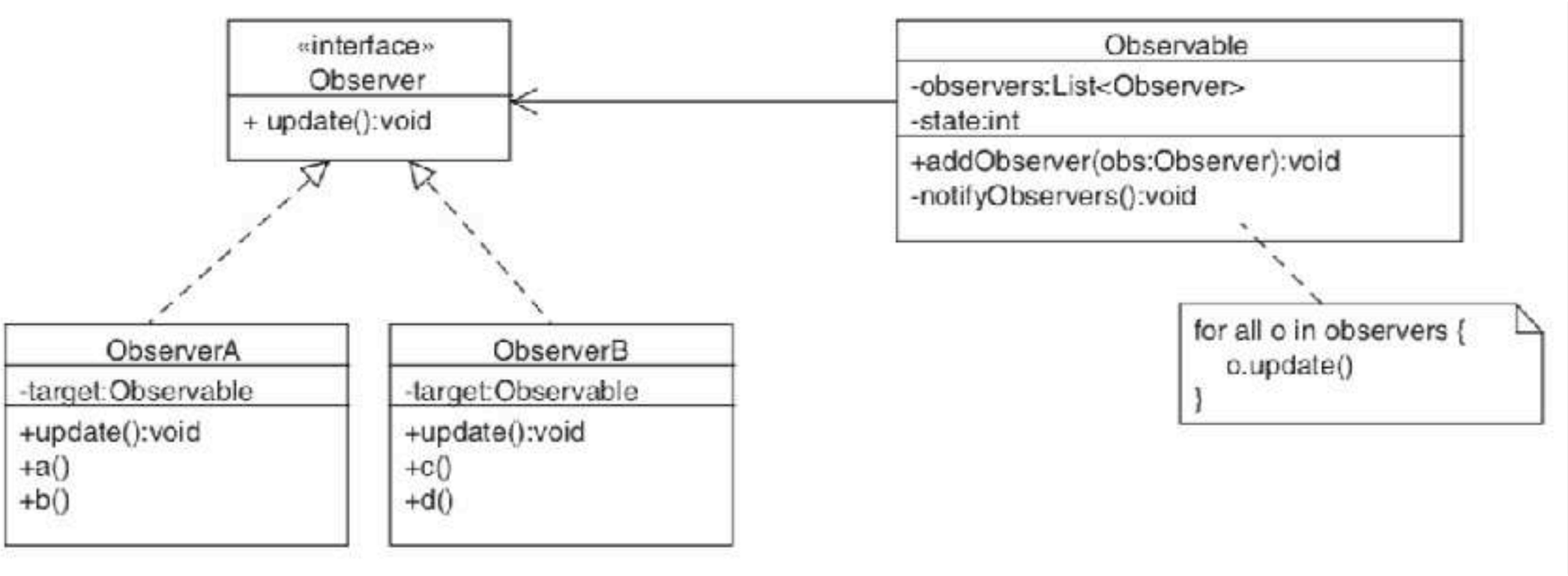
Behavioral Patterns - Overview

- Observer
  - Dependents update automatically when a subject changes
- Decorator
  - Decorator extends an object transparently
- State
  - Object whose behavior depends on its state
- Strategy
  - Vary algorithms independently
- Chain of Responsibility
  - Request delegated to the responsible service provider
- Iterator
  - Aggregate elements are accessed sequentially
- Command
  - Object represents all the information needed to call a method at a later time
- Mediator
  - Mediator coordinates interactions between its associates
- Memento
  - Snapshot captures and restores object states

Behavioral Pattern - Observer

Issue: Need to react to object state changes

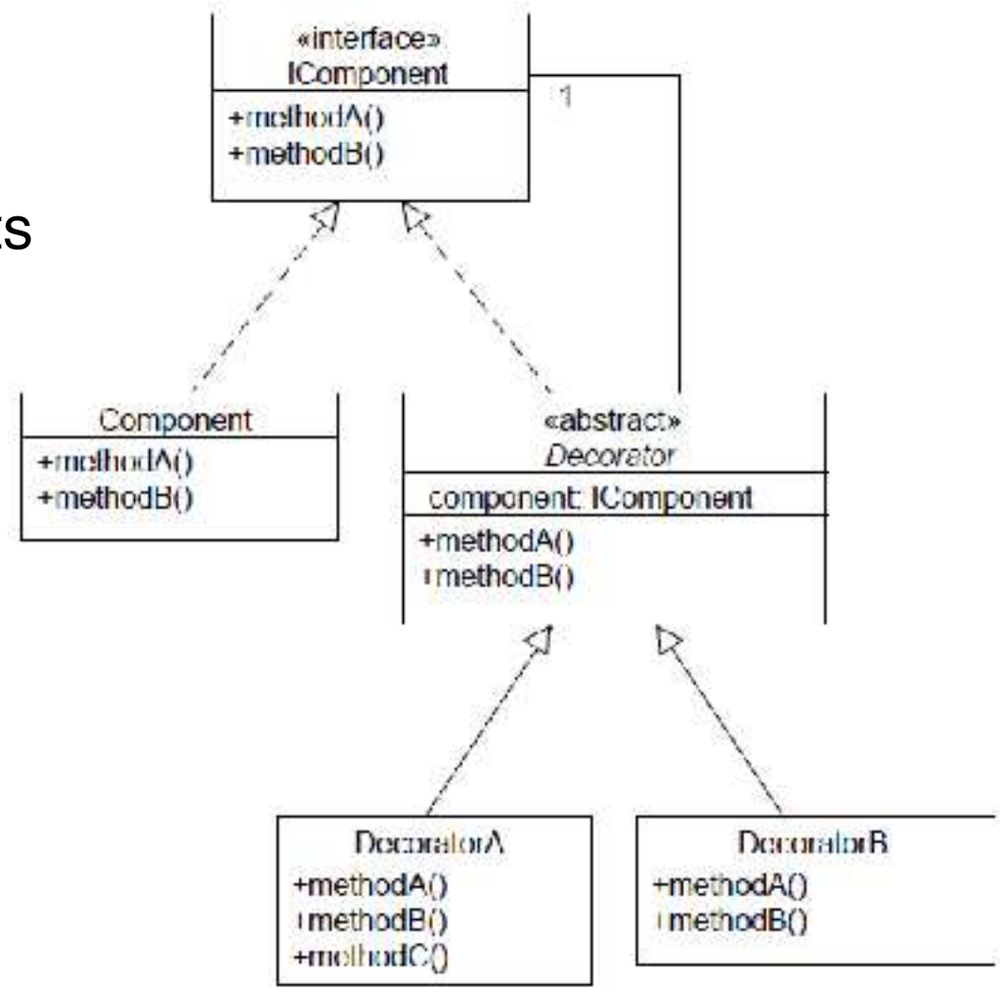
- in case of changes of the instance‘s state execute specific action(s)
  - e.g., notification of instances interested in change
  - one-to-many dependency



Behavioral Pattern - Decorator

Issue: Need to extend object functionality during runtime

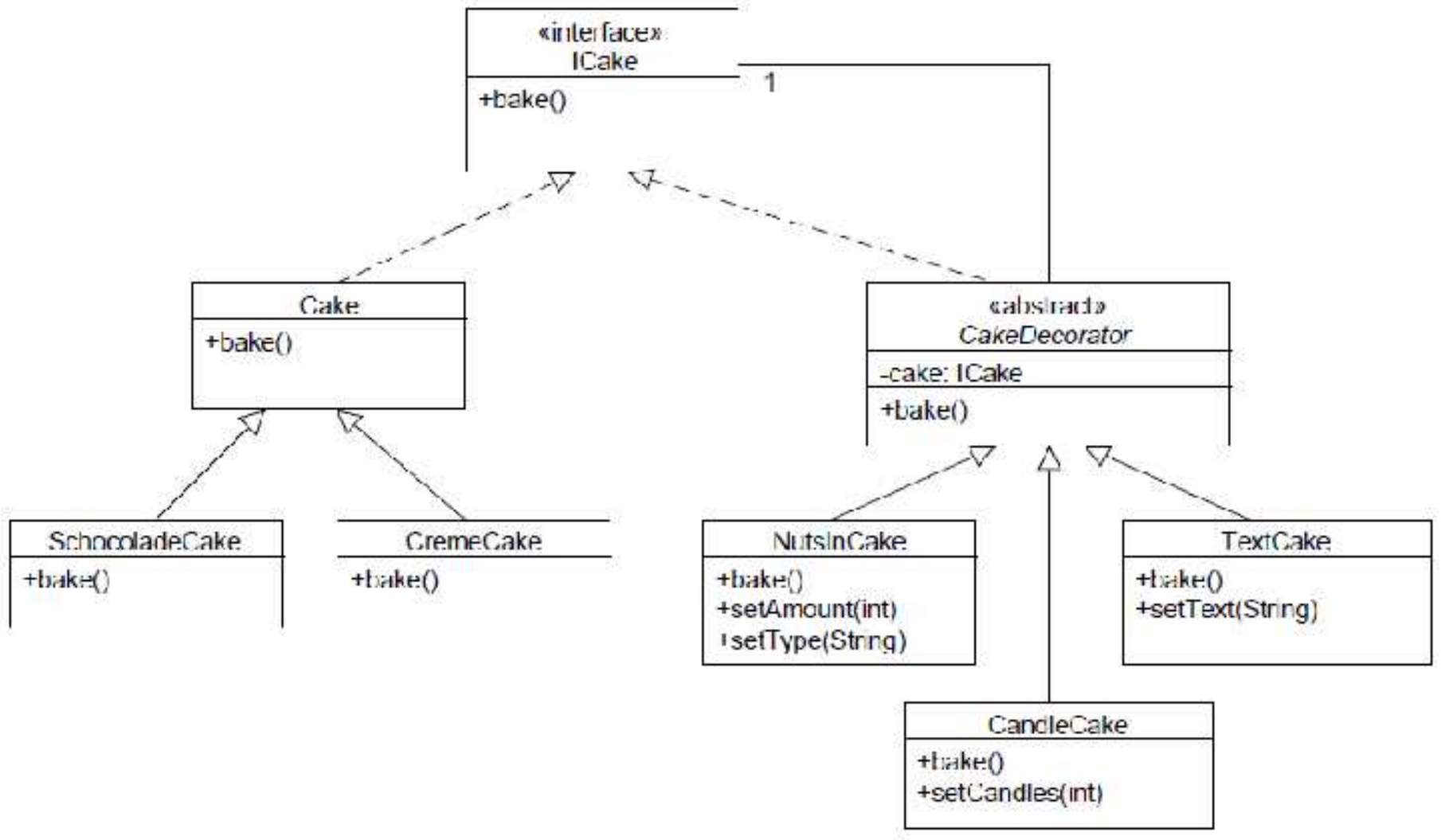
- Dynamically add new functionality to an existing object
  - Some basic work still has to be done at design time
- Elements
  - Interface Component
  - Implemented by concrete components
  - Abstract decorator class
    - Implements interface
    - and keeps reference to interface to forward functionality
  - Concrete decorator implementations
- Drawback
  - Testing
  - proxy



Behavioral Pattern - Decorator

Issue: Need to extend object functionality during runtime

Example: **Cake**

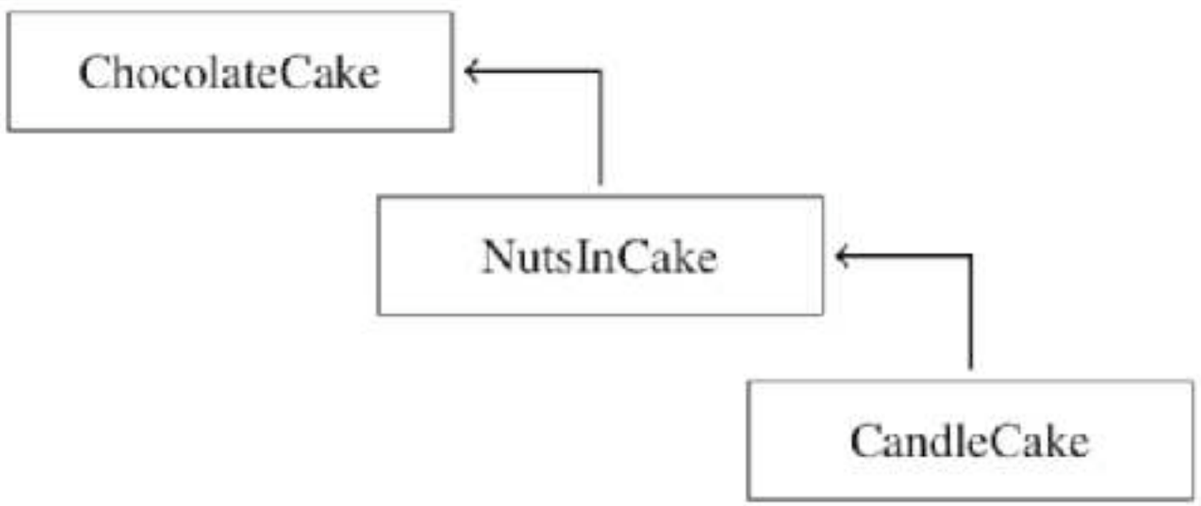


Behavioral Pattern - Decorator

Issue: Need to extend object functionality during runtime

Example: **Cake**

```
Cake cake = new ChocolateCake();
NutsInCake nic = new NutsInCake(cake);
nic.setAmount(15);
nic.setType("hazelnut");
CandleCake cc = new CandleCake (nic);
cc.setCandles(13);
cake = (Cake)cc;

cake.bake();
```



Behavioral Pattern - Decorator

Issue: Need to extend object functionality during runtime

Example: **GUI toolkit**

```
VisualComponent vc = new ScrollBar(
  new Border( new TextEditor() ));
vc.draw();
```

Behavioral Pattern - Decorator

Issue: Need to extend object functionality during runtime

Example: **Stream**

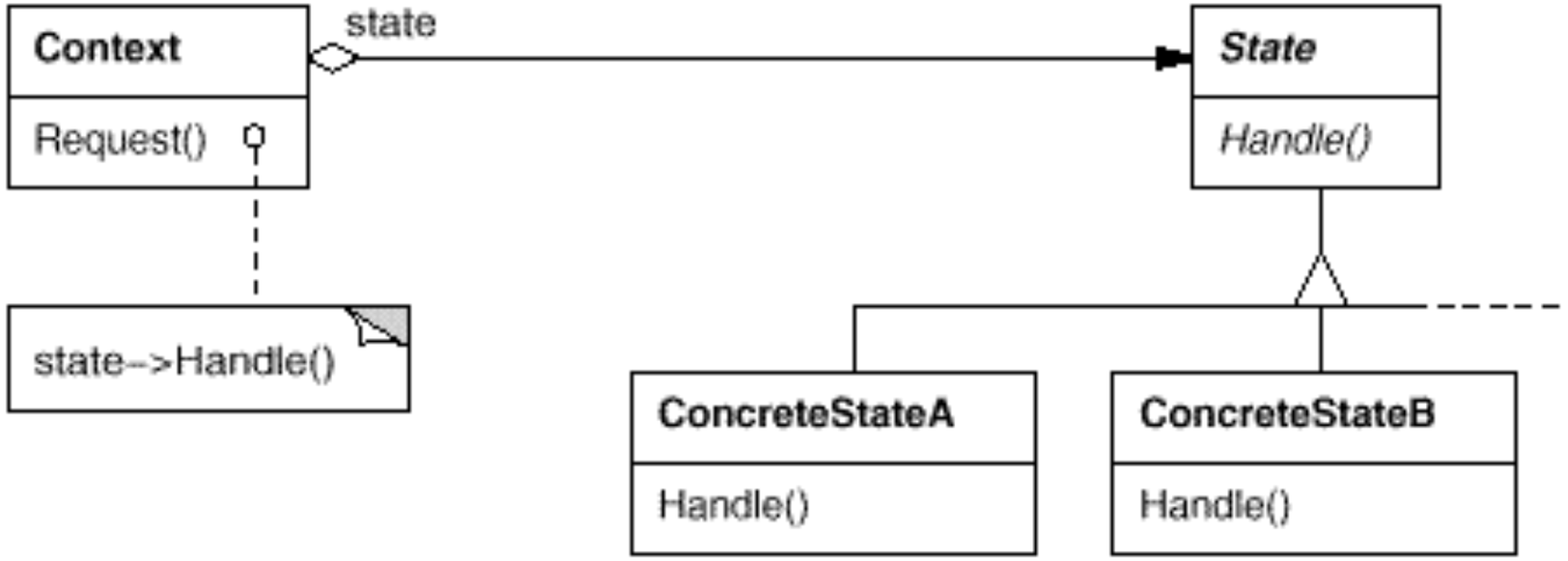
```
Stream s = new FileStream(filename);
```

```
Stream s = new CompressingStream(
               new BufferedStream(
                  new FileStream(filename)
               )
           );
```

Behavioral Pattern - State

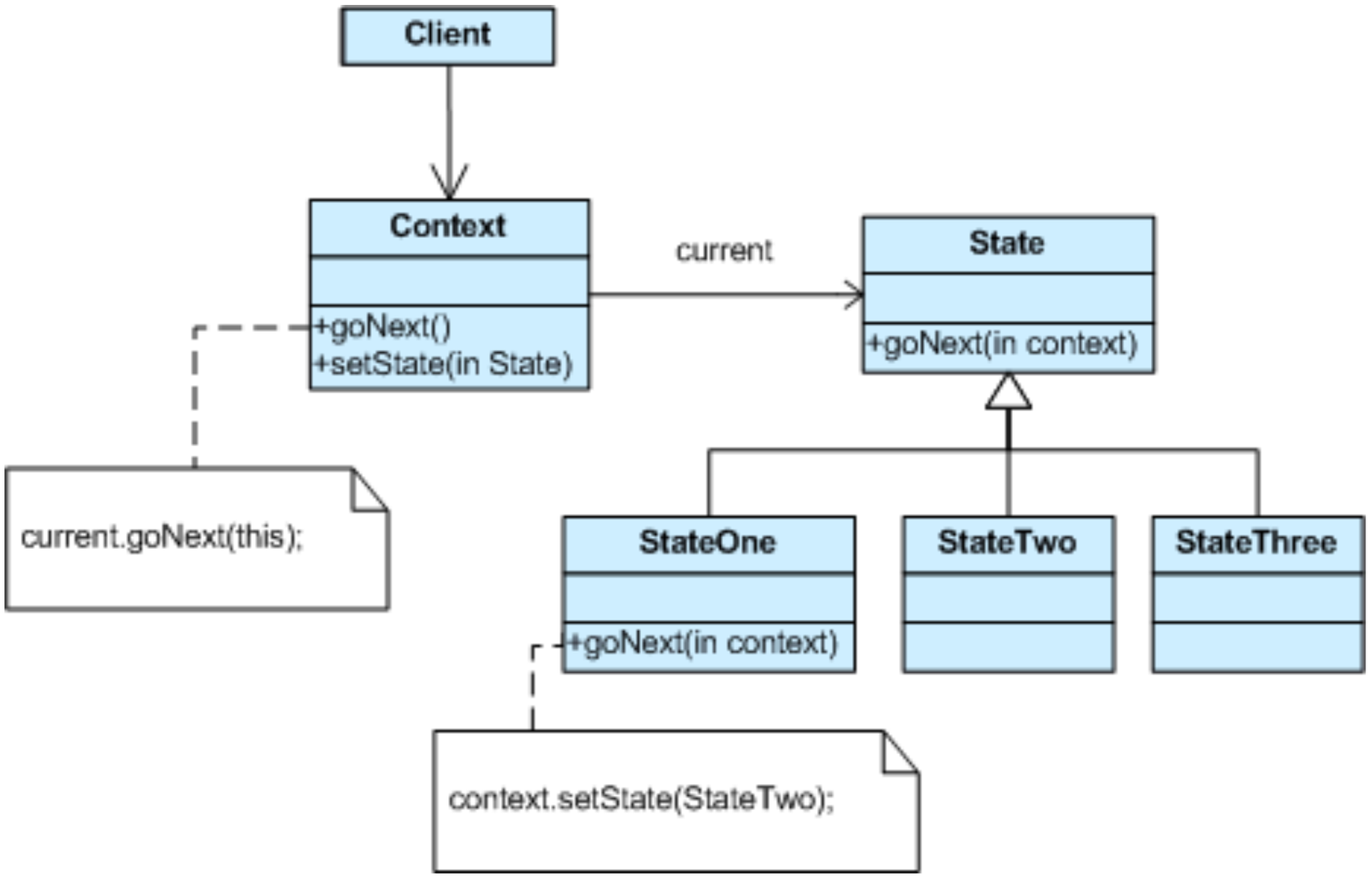
Issue: Need to change object behavior based on current state

- Allow an object to update its behavior when its internal state changes
  - Makes state transitions explicit
  - May result in lots of subclasses



Behavioral Pattern - State

Issue: Need to change object behavior based on current state

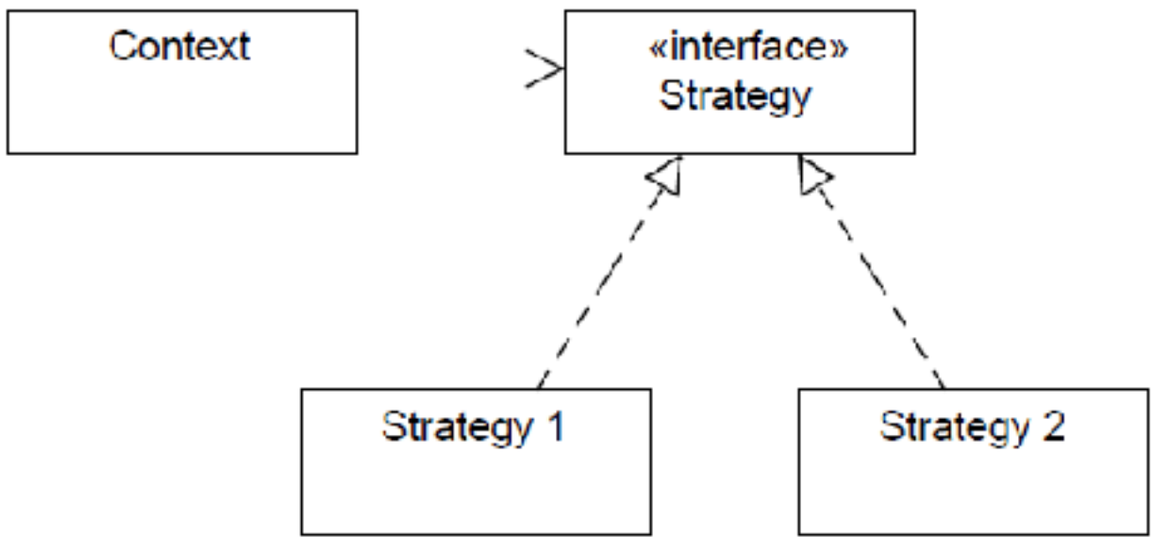


http://sourcemaking.com/design_patterns/state

Behavioral Pattern - Strategy

Issue: Need to extend strategies at runtime

- Dynamically add new algorithms
  - context choose algorithm to use

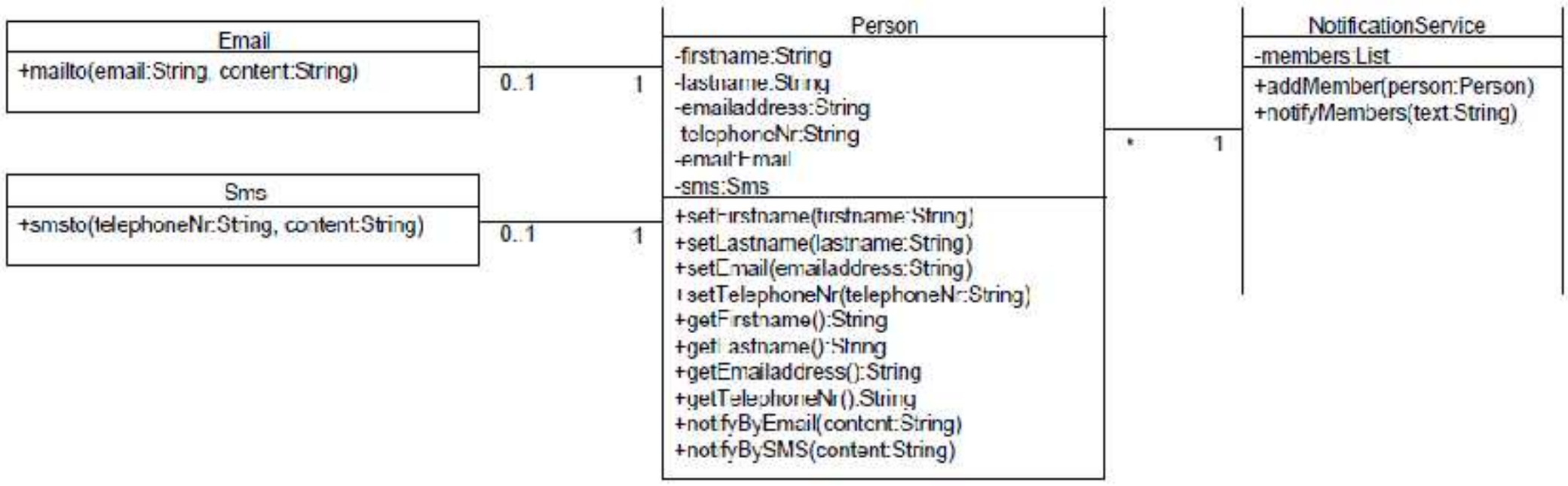


Behavioral Pattern - Strategy

Issue: Need to extend strategies at runtime

Example: **Notification strategy**

- Currently close binding between person and email/sms
  - no use of additional communication technique without changing code
  - Notification service decides technique of communication

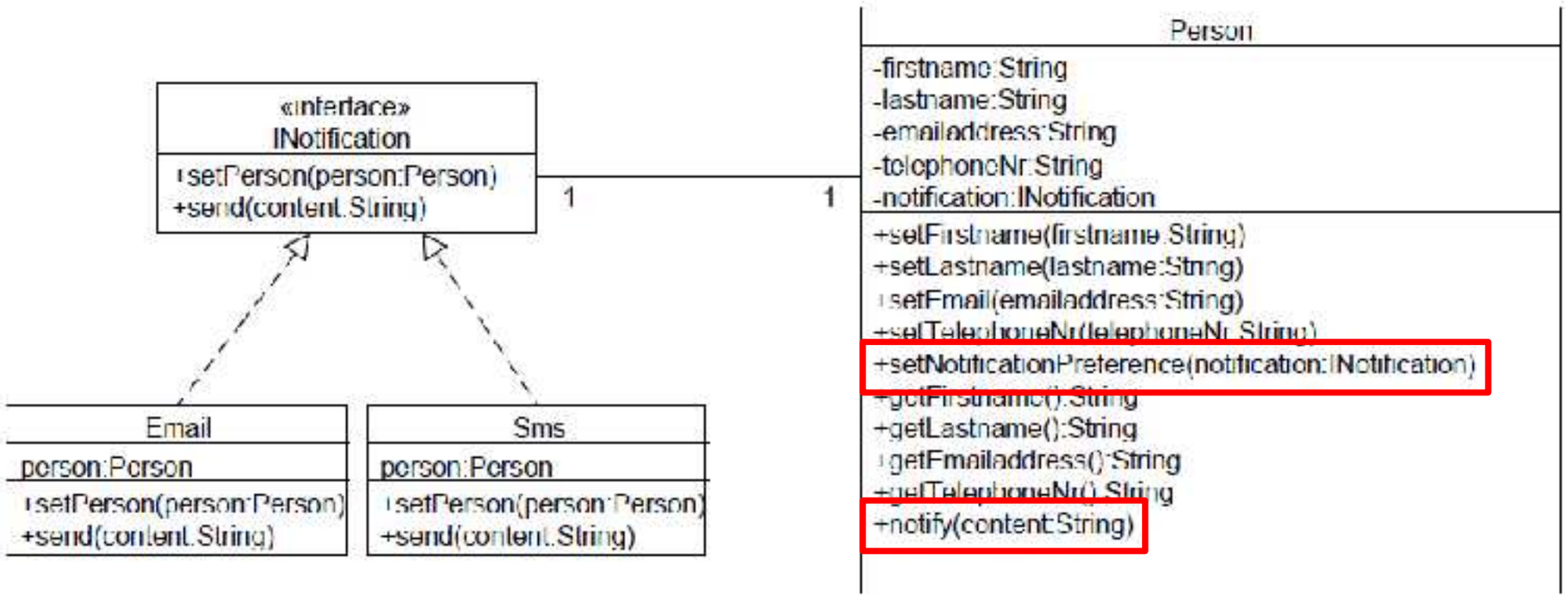


Behavioral Pattern - Strategy

Issue: Need to extend strategies at runtime

Example: **Notification strategy**

- Context object decides which strategy to use



Behavioral Pattern - Strategy

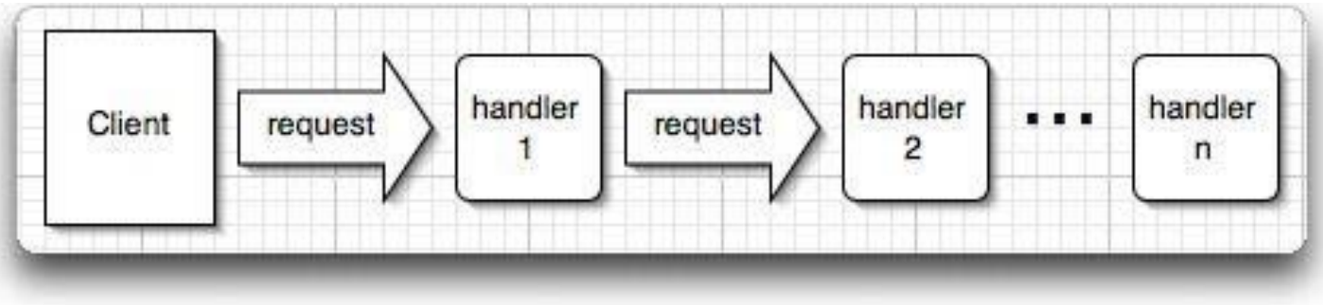
Code Example

```
class NotificationManager {
  public enum NotificationMethod {SMS, EMAIL};
  private List<Person> members;
  ...
  public void addMember(Person person,
            NotificationMethod notificationPref) {
    members.add(person);
    INotification notification;
    if (notificationPref == NotificationMethod.SMS)
    {
      notification = new Sms();
    } else {
      notification = new Email();
    }
    notification.setPerson(person);
    person.setNotificationPreference(notification);
  }

  public void sendNotifications(String message) {
    for (Person p : members) {
      p.sendMessage(message);
    }
  }
  ...
}
```

Behavioral Pattern - Chain of Responsibility

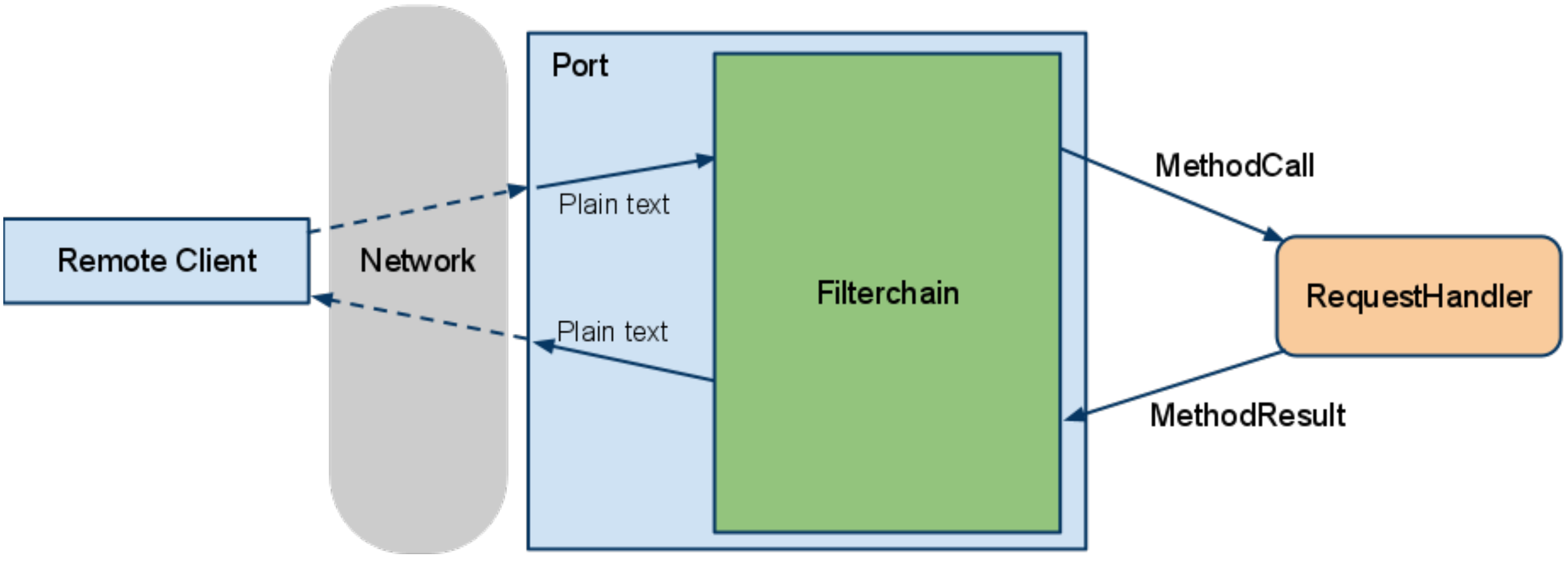
Issue: Improve loose coupling between a series of processing logic

- Chain of Objects
  - a source of command objects
  - a series of processing objects with logic capable of handling specific command objects



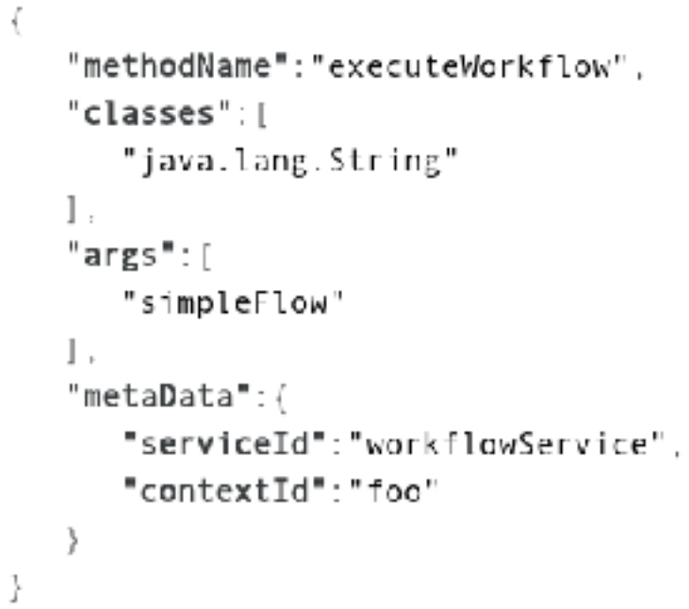
Behavioral Pattern - Chain of Responsibility

- Chain of Objects
  - a source of command objects
  - a series of processing objects with logic capable of handling specific command objects

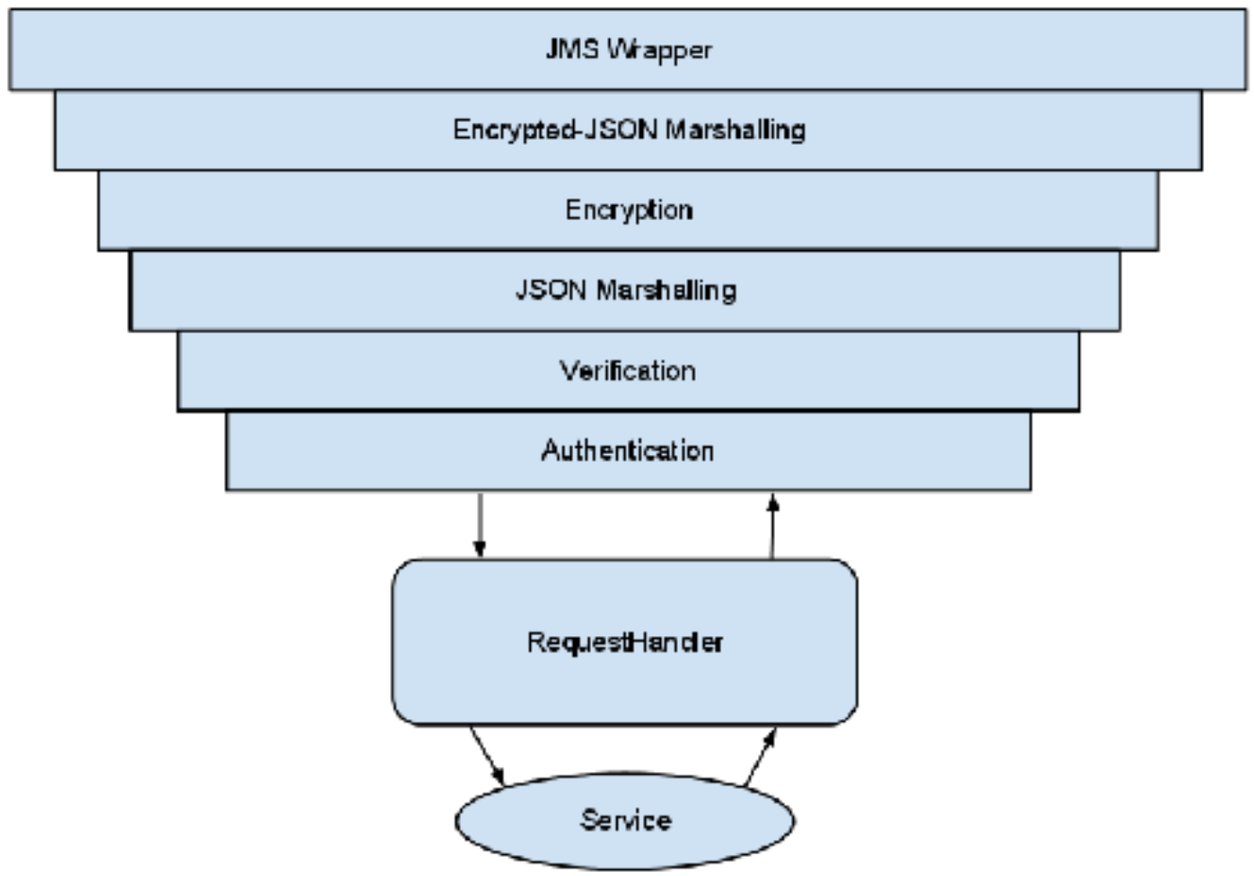


Behavioral Pattern - Chain of Responsibility

Example: **Remote service request**



Json encoded request

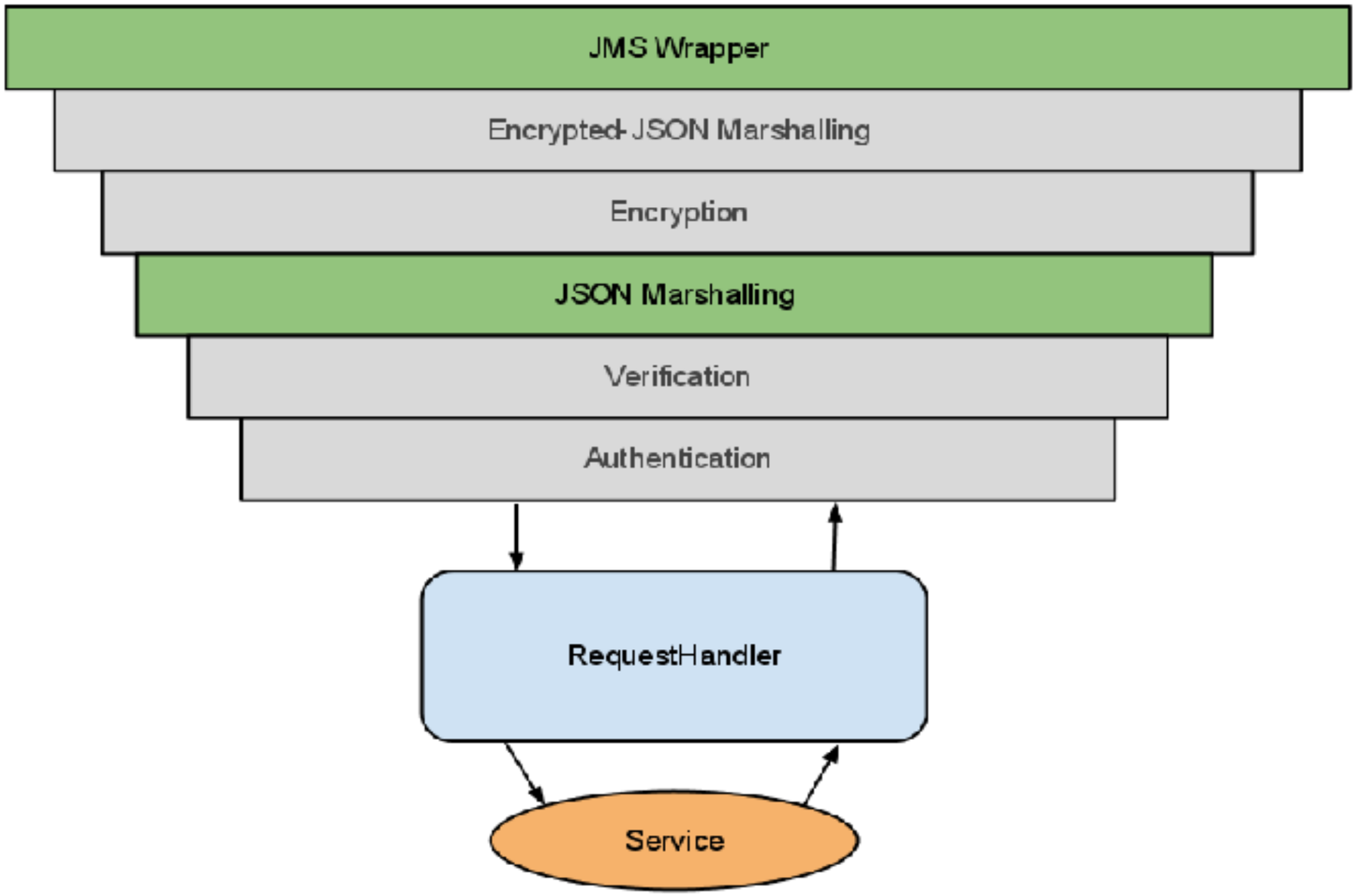


Behavioral Pattern - Chain of Responsibility

Example: **Remote service request**

```
 */
public class JsonMethodCallMarshalFilter extends AbstractFilterChainElement<String, String> {

    private FilterAction next;

    @Override
    public String doFilter(String input, Map<String, Object> metadata) throws FilterException {
        ObjectMapper objectMapper = JsonUtils.createObjectMapperWithIntroSpectors();
        MethodCallRequest call;
        try {
            call = objectMapper.readValue(input, MethodCallRequest.class);
            JsonUtils.convertAllArgs(call);
            MethodResultMessage returnValue = (MethodResultMessage) next.filter(call, metadata);
            return objectMapper.writeValueAsString(returnValue);
        } catch (IOException e) {
            throw new FilterException(e);
        }
    }

    @Override
    public void setNext(FilterAction next) throws FilterConfigurationException {
        checkNextInputAndOutputTypes(next, MethodCallRequest.class, MethodResultMessage.class);
        this.next = next;
    }

}
```

Example: **Remote service request**



Summary

- Industrial Use Case
- Engineering Service Bus
- Design patterns provide a structure in which problems can be solved.
  - Review different applications of one pattern
  - Gain experience
  - "code smells"

- Offering Topics
  - http://qse.ifs.tuwien.ac.at/topics.htm
  - felix.rinker@qse.ifs.tuwien.ac.at

How to gain experience

- Participate in open source projects, e.g.
  - OPS4J https://github.com/ops4j
  - Apache Projects https://projects.apache.org/projects.html
  - Google Summer of Code https://developers.google.com/open-source/gsoc/
  - …

- Build up your own technology radar
  - Martin Fowler: Catalog of Patterns of Enterprise Application Architecture https://martinfowler.com/eaaCatalog/
  - study Stack Overflow design pattern topics https://stackoverflow.com/questions/tagged/design-patterns

References

- Shannon C. E. A Mathematical Theory of Communication. Bell Syst. Techn. J., 1948.
- McDermid, J.A. Complexity: Concept, Causes and Control. in 6th IEEE Int. Conference on Complex Computer Systems. 2000: IEEE Computer
- Society..
- Norman D. O. and M. L. Kuras. Engineering Complex Systems. Technical Report, the MITRE Corporation, 2004.
- Developer.com, A Survey of Common Design Patterns, 2002, http://www.developer.com/design/article.php/1502691/A-Survey-of-Common-Design-Patterns.htm
- Anand, R. and H.C. Roy, What is the complexity of a distributed computing system? Complexity, 2007. 12(6): p. 37-45.
- Bob, C., Complexity in Design. IEEE Computer, 2005. 38(10): p. 10-12.
- Dirk Riehle and Heinz Zullighoven. 1996. Understanding and using patterns in software development. Theor. Pract. Object Syst. 2, 1 (November 1996)
- Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software AW, ’94
- Pattern Languages of Program Design series by AW, ’95-’99.
- Siemens & Schmidt, Pattern-Oriented Software Architecture, Wiley, volumes ’96 & ’00
- http://sourcemaking.com/design_patterns

EngSB - Patterns

- Interface Pattern: https://github.com/openengsb/openengsb-domain-notification/blob/master/src/main/java/org/openengsb/domain/notification/NotificationDomain.java
  - email: https://github.com/openengsb/openengsb-connector-email/blob/master/src/main/java/org/openengsb/connector/email/internal/EmailNotifier.java
  - facebook: https://github.com/openengsb/openengsb-connector-facebook/blob/master/src/main/java/org/openengsb/connector/facebook/internal/FacebookNotifier.java
- Delegation pattern: https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/events/ForwardHandler.java
- Immutable pattern: https://github.com/openengsb/openengsb-framework/blob/master/components/ekb/src/main/java/org/openengsb/core/ekb/internal/ConnectorInformation.java

EngSB - Patterns

- Singleton pattern: https://github.com/openengsb/openengsb-framework/blob/master/components/api/src/main/java/org/openengsb/core/api/context/ContextHolder.java
- Factory pattern: https://github.com/openengsb/openengsb-framework/blob/master/components/api/src/main/java/org/openengsb/core/api/ConnectorInstanceFactory.java
- Proxy pattern:
  - https://github.com/openengsb/openengsb-framework/blob/master/components/services/src/main/java/org/openengsb/core/services/internal/virtual/ProxyConnector.java
  - https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/virtual/InvokeAllIgnoreResultStrategy.java
- Chain of responsibility pattern: https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/remote/

ca technologies

Fail Fast to Succeed
Der Wert von Feedbackschleifen

Christoph Bonitz

Mai 2018

Warum dieser Vortrag?

- Bogen spannen über die Themen der Lehrveranstaltung aus der Perspektive eines Praktikers
- Idealerweise: Kontext + Motivation
- Notwendigerweise subjektiv
  - Weniger rigoros und wissenschaftlich als reguläre Vorlesung
  - Meinungen sind meine, nicht die meines Arbeitgebers
- Bitte jederzeit (hinter)fragen!

Copyright © 2018 CA. All rights reserved.

Agenda

WARUM FEEDBACK WICHTIG IST

FEEDBACK AUF INDIVIDUELLER EBENE

PROJEKTE AUF SCHNELLES FEEDBACK TRIMMEN

SCHNELLE RÜCKMELDUNGEN WÄHREND DES PROGRAMMIERENS

SCHNELLE RÜCKMELDUNGEN FÜR DAS TEAM

FEEDBACK FÜR DAS PRODUKT

ca technologies

Copyright © 2018 CA. All rights reserved.

Warum Feedback wichtig ist

- Wir verstehen unsere Umwelt durch Interaktion und Rückmeldungen
- Berufsleben: Ziel Erfolg, nicht Bewertung am Projektende
- Keine (Lebens-) Zeit für sinnlose Projekte
- Früher korrigieren = billiger
- Fehlervermeidung noch billiger

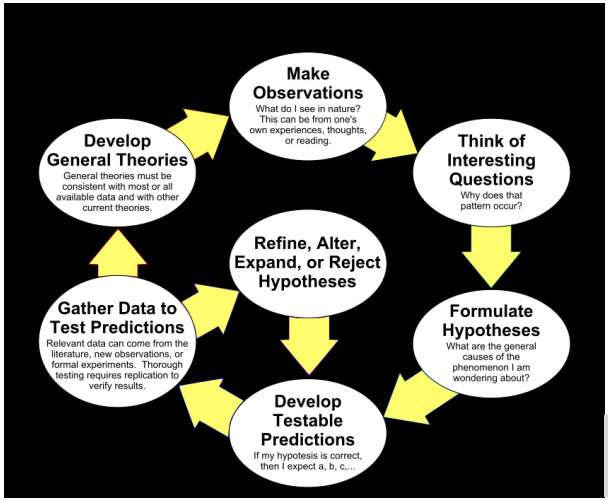
Kernthese: **Wir müssen uns aktiv um Feedback auf jeder Ebene bemühen.**

- ☺ Viele einfache & billige Methoden

Copyright © 2018 CA. All rights reserved.

Erfolgsgeschichte Wissenschaft

- Etablierung der *Wissenschaftlichen Methode* hat das Fortschreiten der Wissenschaft massiv beschleunigt. Schlüssel: Empirisches Arbeiten



Quelle: Wikimedia Commons

Copyright © 2018 CA. All rights reserved.

Erfolgsgeschichte (Lean) Startups

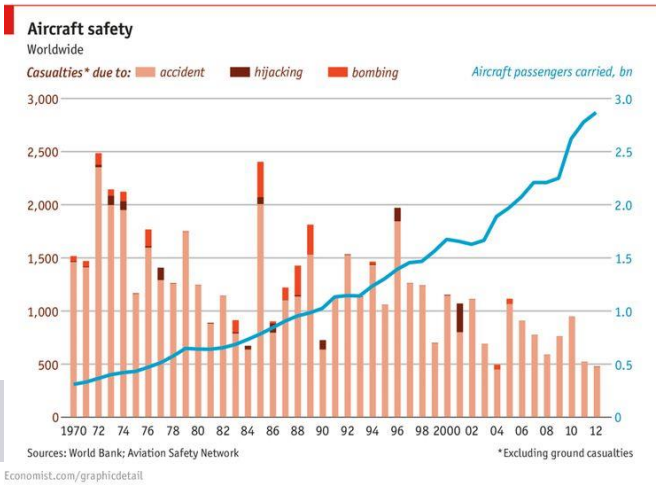
- Product/Market Fit
  (Steve Blank: The Four Steps to the Epiphany, 2006)
- Landing Pages ohne Produkt dahinter
- Elevator Pitch
- Minimum Viable Product
- Pivoting
- A/B Testing
- Analytics

http://theleanstartup.com/principles

Copyright © 2018 CA. All rights reserved.

Erfolgsgeschichte Fliegerei

- Crew Resource Management
- Unfall => Suche nach Verbesserungen (nicht Schuld)
- Lernen auf Flugzeugen mit niedrigen Betriebskosten
- Simulatoren
  (Lehre, Checks)



Quelle: The Economist

Copyright © 2018 CA. All rights reserved.

Persönliches Feedback

ANNEHMEN

GEBEN

Copyright © 2018 CA. All rights reserved.

Persönliches Feedback annehmen

- Wertschätzung: Erst zuhören, dann reagieren
  - Technik: Bedanken & Paraphrasieren
- Nicht immer angenehm
  - Will ich lieber nichts hören aber dann keinen Erfolg haben?
  - Auch nicht konstruktives Feedback kann mir helfen.
- Feedback einfordern
  - Leichter von vertrauten Menschen
  - Informelle Gespräche vor großen Meetings
- Auf Feedback reagieren = Schlüsselkompetenz
  - Direkte Kommunikation als Differenziator zu Offshoring

Copyright © 2018 CA. All rights reserved.

Persönliches Feedback geben I

- Höflich und wertschätzend
- Negatives Feedback unter vier Augen
- Die eigene Meinung aussprechen (nicht „Wahrheit“)
- Über das Problem, nicht die Person sprechen
  - *Der* Code war *für mich* schwer zu lesen (nicht: *Dein* Code *ist…*)

ca technologies

Copyright © 2018 CA. All rights reserved.

Persönliches Feedback geben II

- Alternativen anbieten
  - Wie wäre es, wenn wir hier einen Kommentar einfügen?
  - Könnten wir die Variablen leichter verständlich benennen
- Nicht erst beginnen, wenn es ein Problem gibt
  - Regelmäßig positive Rückmeldung geben
  - Feedback institutionalisieren, z.B. regelmäßige Code-Reviews, Retrospektiven, 1:1 Meetings

ca technologies

Copyright © 2018 CA. All rights reserved.

Weiterführende Hinweise

- Soft Skills-Lehrveranstaltungen nutzen
- *Miteinander Reden* Reihe
  (Friedmann Schulz v. Thun et al.)

Copyright © 2018 CA. All rights reserved.

Projekte auf schnelles Feedback trimmen

ITERATIVE ANSÄTZE

FEEDBACK IN DEN PROZESS EINBAUEN

DATEN SAMMELN UND VERFOLGEN

ca technologies

Copyright © 2018 CA. All rights reserved.

Iterative Ansätze – Bsp. Scrum

- Kurze Iterationen (1-4 Wochen)
  - Verhindert, unangenehme Erkenntnisse vor sich herzuschieben
- Potentially shippable product increment
  - Nichtfunktionale Anforderungen immer erfüllt
- Review Meetings mit Projekt-Stakeholder (Kunden!)
- Retrospektiven
  - Lernen aus dem letzten Sprint
- Bevorzugung von co-located Teams
  - Persönliche Kommunikation

Copyright © 2018 CA. All rights reserved.

Feedback in Prozess einbauen I

- Prototypen
  - Am besten Throwaway Prototyping
- Code Reviews, Pair Programming
  - Man übt regelmäßiges Feedback (geben/nehmen)
  - Man lernt schnell von anderen. Subjektiv: massive Verbesserung
  - Wichtig: JedeR reviewt jedeN
- Architektur top-down aus dem Elfenbeinturm
  - RfCs
  - Arbeitsgruppen
  - Beispielcode (*Exemplar*)

Copyright © 2018 CA. All rights reserved.

Feedback in Prozess einbauen II

- Neue Features sofort testen
  - Am besten auch durch Kunden
- Usability Tests
  - Paper prototypes
  - Hallway usability testing
  - Formal usability testing

ca technologies

Copyright © 2018 CA. All rights reserved.

Daten sammeln und verfolgen I

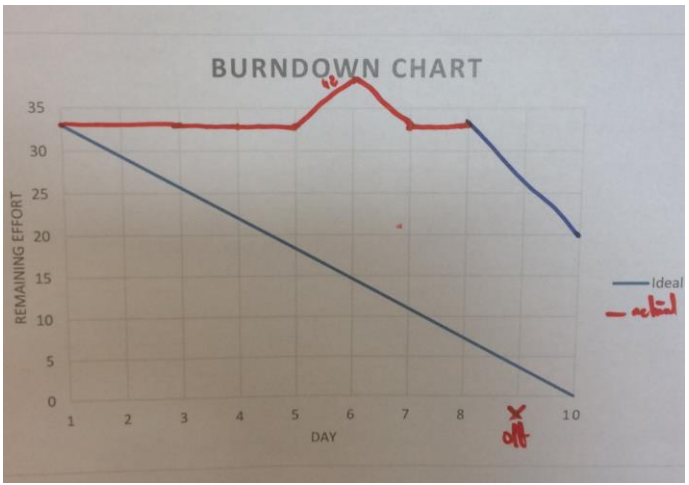
- Kein Projekt ohne Issue-Tracker!
- Ein Ticket für
  - Jedes Problem
  - Jede Story
  - Jede Infrastrukturänderung?
  - Sonstige TODOs?
- Nachverfolgbarkeit
- Vorteil: Auswertungen über längere Zeit
  - Auch im Nachhinein möglich
  - Diskussionen auf empirischer Basis
  - Interventionen überprüfen

ca technologies

Copyright © 2018 CA. All rights reserved.

Daten sammeln und verfolgen II

- Auch an analoge Methoden denken
  - Burndown
  - Stimmungsbarometer
  - Stimmungskurven



Copyright © 2018 CA. All rights reserved.

Während des Programmierens

EFFEKTE SOFORT SEHEN (NICHT IMMER MÖGLICH)

EFFEKTEN SCHNELLER SEHEN

Copyright © 2018 CA. All rights reserved.

Beispiel: Live coding with Notch (Mincraft)

- https://www.youtube.com/watch?v=BES9EKK4Aw4
- Ein Monitor: IDE
- Anderer Monitor: Spiel
- Ständiges Wechseln zwischen Spiel und Code
- Spiel auf schnellen Restart getrimmt
- Effekte sofort sichtbar

Copyright © 2018 CA. All rights reserved.

Beispiel: Distill

- Niveau eines wissenschaftlichen Papers
- Interaktive Teile zum besseren Verständnis
- http://distill.pub/2017/momentum/
- Inspiriert von Ideen von Bret Victor (http://worrydream.org)
  - Siehe auch: http://lighttable.com/ - Versuch, diese Prinzipien in einer IDE umzusetzen

ca technologies

Copyright © 2018 CA. All rights reserved.

Beispiel: Smalltalk

- Bereits in den 80er Jahren
- Programmier/Ausführungsumgebung vermischt
- System ist zur Laufzeit manipulierbar
- https://www.youtube.com/watch?v=JLPiMl8XUKU
  - Randnotiz: hier ist sogar die Maus noch neuartig genug, um extra hergezeigt zu werden
- Zum Ausprobieren: Squeak http://squeak.org/

ca technologies

Copyright © 2018 CA. All rights reserved.

Und was machen die anderen?

- Nicht jede Art von Software ermöglicht sofortiges Feedback
  - Schon gar nicht von Haus aus
- Wenn möglich, Technologien mit kurzem Turnaround bevorzugen
- Aktiv Feedbackschleifen schaffen

Copyright © 2018 CA. All rights reserved.

IDE, beste Freundin der EntwicklerInnen

- Erinnerungen: Java im Texteditor
- Moderne IDEs bieten in Echtzeit u.a.
  - Code completion
  - Statische Code-Analyse
  - Verwendungssuche
  - Rendering von Dokumentation
  - Kompilation beim Speichern
    - Hot code replacement (mehr oder weniger gut)
  - Refactoring-Unterstützung
  - Codegenerierung
- Es zahlt sich aus, eine IDE *gut zu beherrschen*

ca technologies

Copyright © 2018 CA. All rights reserved.

Unit Testing

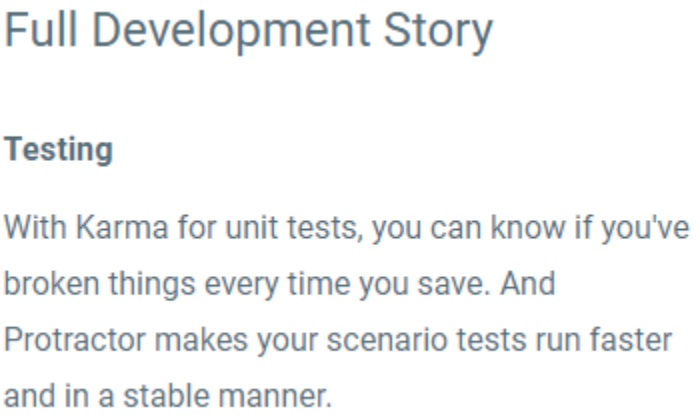
- Klassisch – *Red/Green Cycle* des *Test Driven Development*
- Heute in manchen Sprachen: test on save
  - Permanent laufender Prozess hört auf Änderungen im Dateisystem und startet die Tests
  - Ergebnisse werden sofort angezeigt
- Unit Tests helfen, den Code zu strukturieren
- Unit Test helfen, die Intention hinter Code auszudrücken
- Unit Tests helfen (zusammen mit Integrationstests), sicherer zu refaktorieren

ca technologies

Copyright © 2018 CA. All rights reserved.

Testability

- Heute heben Frameworks oft heraus, wie leicht und gut Code Unit-testbar ist. Bsp: angular.io:



Copyright © 2018 CA. All rights reserved.

Startup-Zeit der Anwendung verkürzen

Wie lange muss ich zwischen einer Änderung und Ausprobieren der Effekte warten?

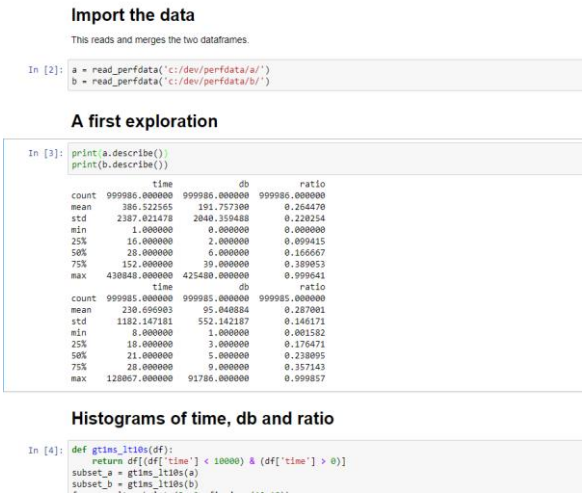
- Rentiert sich schnell:
  - Zig mal pro Tag
  - Flow nicht unterbrechen mindestens so wichtig wie die verlorene Zeit
- Situation unterschiedlich je nach Sprache
- Bei Anwendungen mit Login-Screen: Kann ich diesen für die Entwicklung deaktivieren (ohne Sicherheitslücken einzubauen!!!)?

ca technologies

Copyright © 2018 CA. All rights reserved.

Interaktive Umgebungen nutzen (speziell für Prototypen)

- Beispiel: Jupyter Notebooks: Literate Programming für Julia, Python, R
- Nichts davon ist komplett neu
  - Literate Programming vorgestellt von D.E. Knuth in 1980ern
  - Lisp REPLs gibt es schon sehr lange (kein genaues Datum gefunden)
  - Smalltalk Umgebungen sind sehr interaktiv (1980er)



Import the data

This reads and merges the two dataframes.

```
In [2]: a = read_perfdata('c:/dev/perfdata/a/')
        b = read_perfdata('c:/dev/perfdata/b/')
```

A first exploration

```
In [3]: print(a.describe())
        print(b.describe())
```

```
                time             db          ratio
count  999986.000000  999986.000000  999986.000000
mean      386.522565     191.757300       0.264470
std      2387.021478    2040.359488       0.220254
min         1.000000       0.000000       0.000000
25%        16.000000       2.000000       0.099415
50%        28.000000       6.000000       0.166667
75%       152.000000      39.000000       0.389053
max    430848.000000  425480.000000       0.999641
                time             db          ratio
count  999985.000000  999985.000000  999985.000000
mean      230.696903      95.848884       0.287001
std      1182.147181     552.142187       0.146171
min         8.000000       1.000000       0.001582
25%        18.000000       3.000000       0.176471
50%        21.000000       5.000000       0.238095
75%        28.000000       9.000000       0.357143
max    128067.000000   91786.000000       0.999857
```

Histograms of time, db and ratio

```
In [4]: def gtlms_lt10s(df):
            return df[(df['time'] < 10000) & (df['time'] > 0)]
        subset_a = gtlms_lt10s(a)
        subset_b = gtlms_lt10s(b)
```

ca technologies

Copyright © 2018 CA. All rights reserved.

Resultate visualisieren

- Code instrumentisieren, Visualisierung der Resultate
- Umgebung visualisieren
  - JVisualVM
  - Resource Monitor / Task Manager / htop
- *Application Performance Monitoring/Management*

ca technologies

Copyright © 2018 CA. All rights reserved.

Feedback für das Team

CONTINUOUS BUILD

STATISCHE CODEANALYSE

CONTINUOUS INTEGRATION

CONTINUOUS DELIVERY

CONTINUOUS INTEGRATOIN BEI CA

ca technologies

Copyright © 2018 CA. All rights reserved.

Continuous Build I

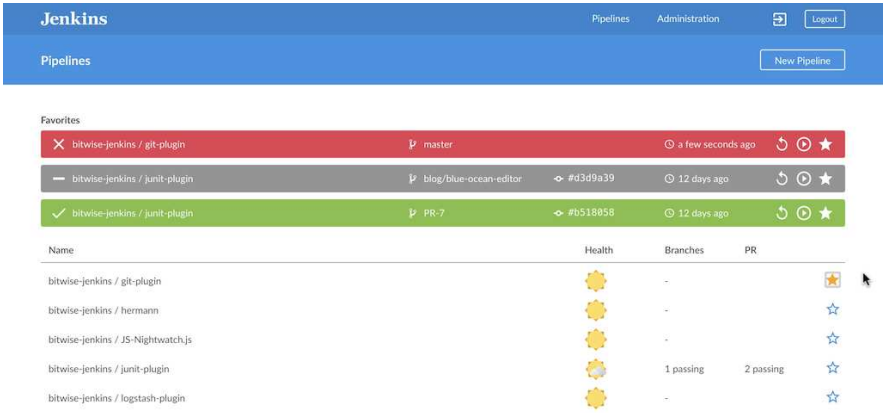
- Voraussetzung: Command Line Build
  - Standardtools!
    - Makefiles / Autotools
    - Ant / Maven / Gradle
    - Grunt / Gulp / …
  - Managed dependencies: Möglichst wenig Abhängigkeit von Umgebung
  - Sollte möglichst das Endprodukt bauen
  - Build nur erfolgreich, wenn Unit Tests laufen

ca technologies

Copyright © 2018 CA. All rights reserved.

Continuous Build II

- Build on Commit/Push
  - Single Source of Truth statt *works on my machine*
  - Tools wie Jenkins, Travis CI, Teamcity, …
  - Standardisierte Umgebungen

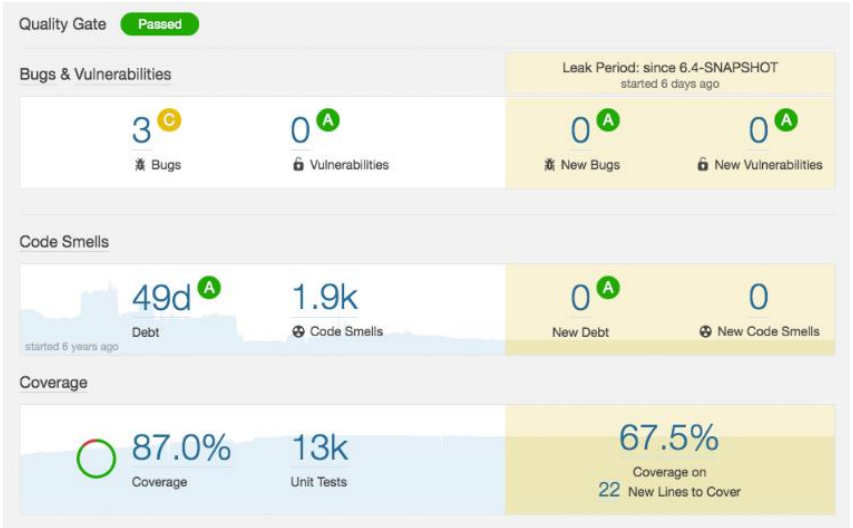


Quelle: Jenkins.org

Copyright © 2018 CA. All rights reserved.

Statische Codeanalyse

- Beispiel: SonarQube https://www.sonarqube.org/
  - Regeln
  - Maven Plugin
  - History
  - Quality Hotspots
  - IDE Plugins



Quelle: SonarQube

Copyright © 2018 CA. All rights reserved.

Continuous Integration

- Nach dem Build:
  Gesamte Anwendung zentraler Stelle deployen
- Single Source of Truth
- Automatisierte Integrationstests
- Sehr praktisch für manuelle Tests: Ich kann Changes testen, die ich noch nicht auf meinem Rechner habe (z.B. anderer Branch)
- In der Praxis nicht immer trivial – mehr dazu später

ca technologies

Copyright © 2018 CA. All rights reserved.

Automatisierte Integrationstests

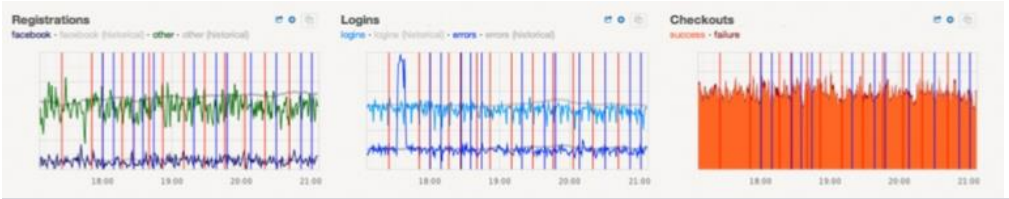
- Testen das System als Ganzes
- Enormer Wert für das Finden von Regressionen, speziell in der Zusammenarbeit zwischen Komponenten
- Zumindest einige davon sollten End-to-End sein, also z.B. vom webbasierten UI bis zur Datenbank möglichst Komponenten berühren
- Manchmal Tradeoff zwischen Komplexität und Verlässlichkeit
  - UI Tests sind oft nicht 100% stabil
  - Zu viele *false positives* zerstören die Signalwirkung

ca technologies

Copyright © 2018 CA. All rights reserved.

Continuous Delivery

- Noch ein Schritt weiter: Jeder Commit landet potenziell in der Produktion
- Braucht:
  - Solide Integrationstests
  - Monitoring der Betriebsdaten (real user data)
  - Strategien zur Risikominimierung (z.B. blue-green-deployments, Rollback)
  - Betrieb der eigenen Anwendung



https://www.slideshare.net/mikebrittain/principles-and-practices-in-continuous-deployment-at-etsy

Copyright © 2018 CA. All rights reserved.

Was tun als on-premise Vendor? Unsere Story

- Es gibt viele exzellente Vorträge und Blogeinträge von SaaS-Firmen über Continuous Delivery
  - z.B. Netflix, Etsy, Github
- Die Workload Automation Produkte von CA sind Standardsoftware, die on premise läuft.
  - Schnell iterieren = Wettbewerbsvorteil
  - Wie kann man diese Feedbackschleifen in unserem Kontext herstellen?
  - Die folgenden Folien kommen von meinem Kollegen Christopher Hejl, der dieses Projekt geleitet hat
    - Ganzer Vortrag: https://www.youtube.com/watch?v=xVtyrcliCdk
  - Slides: „Automic“ = Firmenname vor Akquisition durch CA

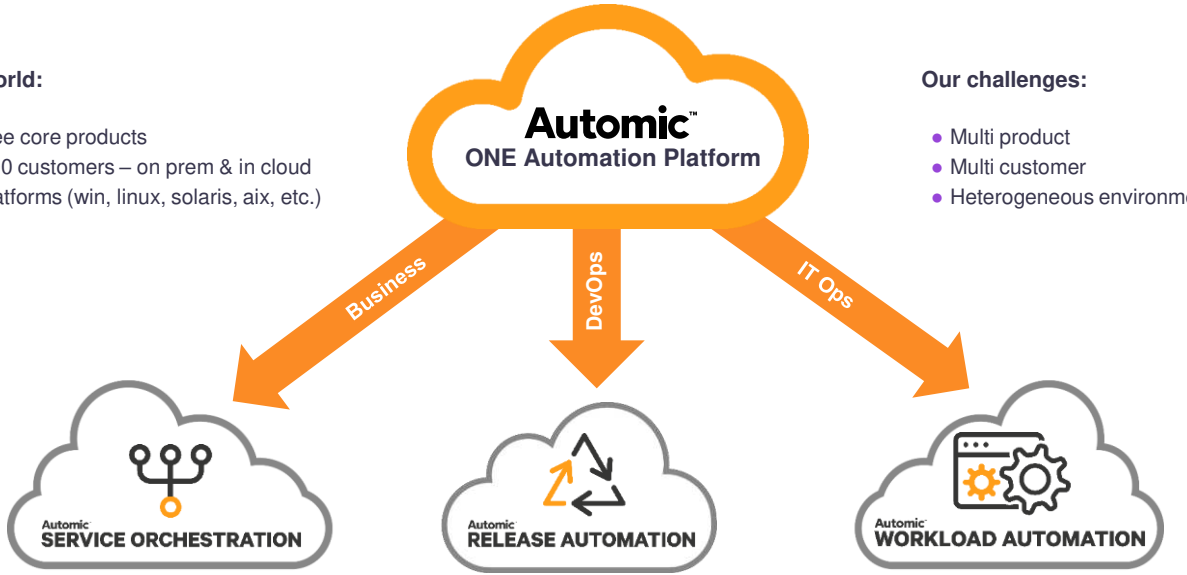
ca technologies

Copyright © 2018 CA. All rights reserved.

Automic in 2 minutes (in 2017)

Our world:

- Three core products
- 2.700 customers – on prem & in cloud
- x platforms (win, linux, solaris, aix, etc.)



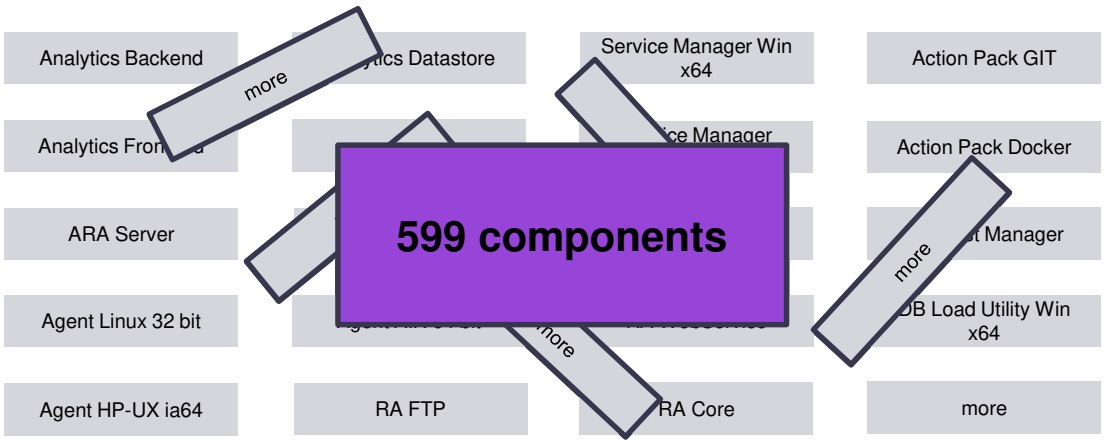
Our challenges:

- Multi product
- Multi customer
- Heterogeneous environments

ca technologies

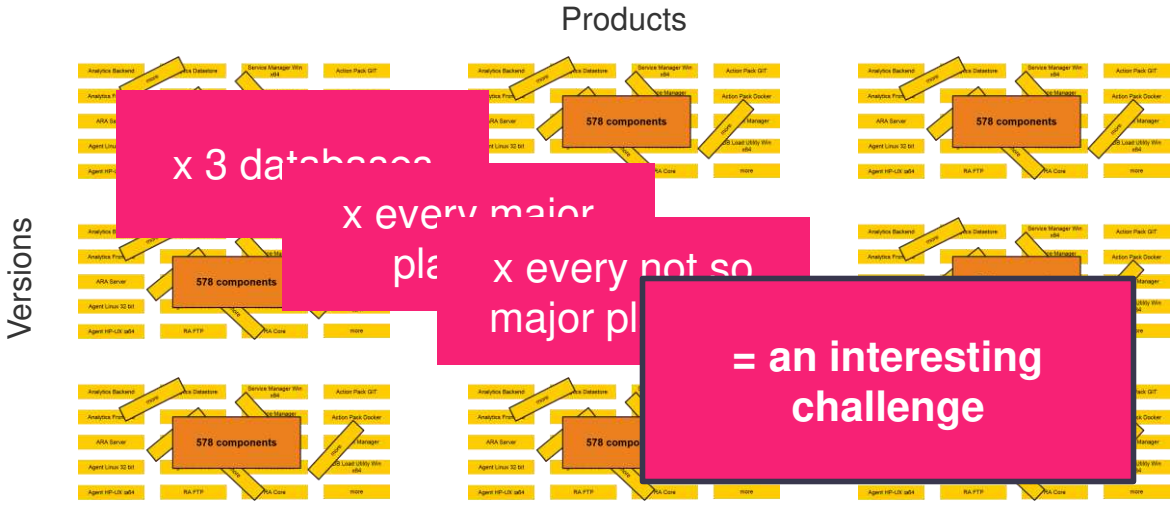
Copyright © 2018 CA. All rights reserved.

Components and Products



Copyright © 2018 CA. All rights reserved.

Components and Products

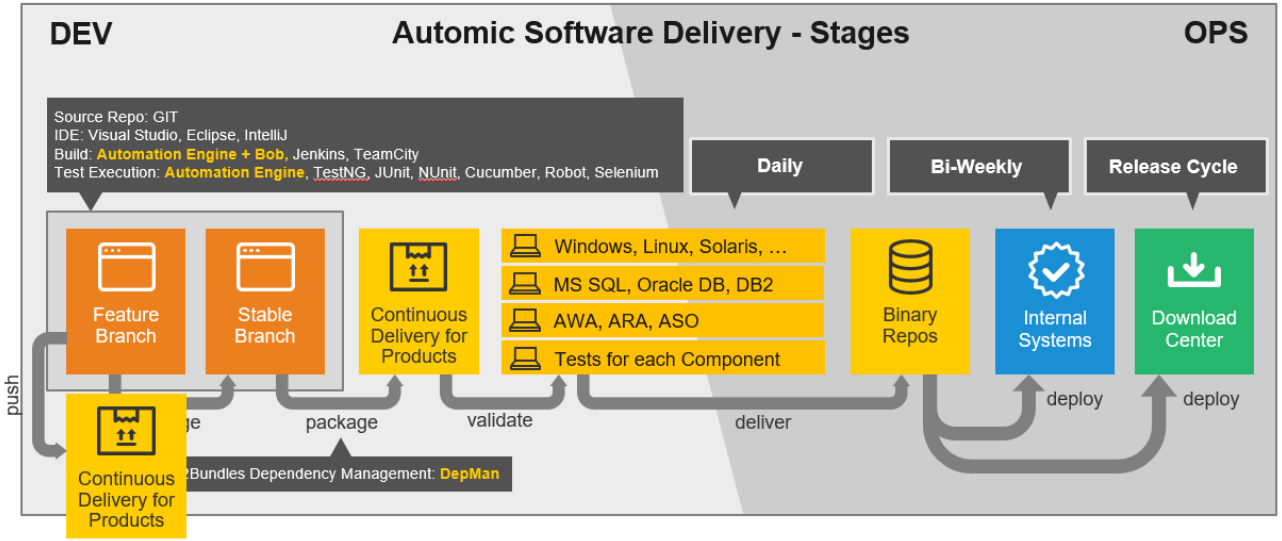


Copyright © 2018 CA. All rights reserved.

Let’s see how others solve this

Copyright © 2018 CA. All rights reserved.

How does it work in detail?



Copyright © 2018 CA. All rights reserved.

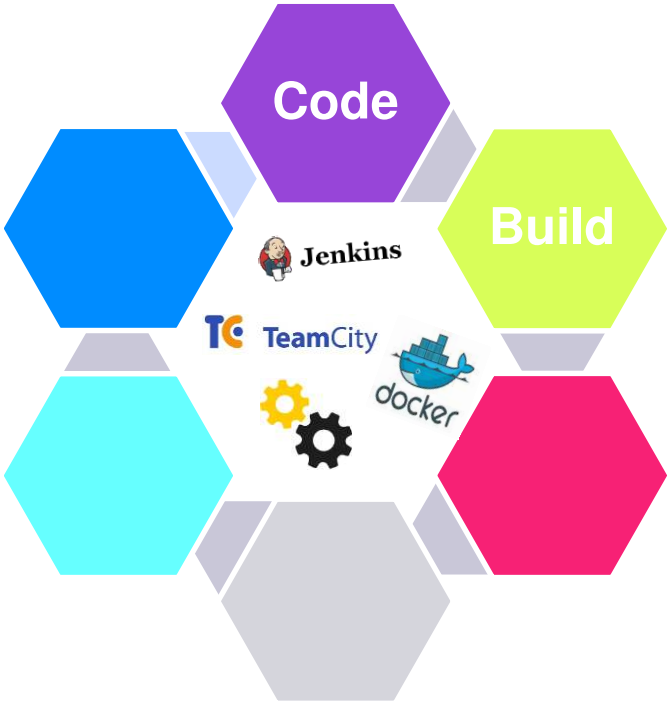
Code

Everything starts with a ticket
Branch off
Adapt code
Run local tests
Commit/push
Merge/pull request

All done via
- Ticket-system
- IDE

ca technologies

Copyright © 2018 CA. All rights reserved.



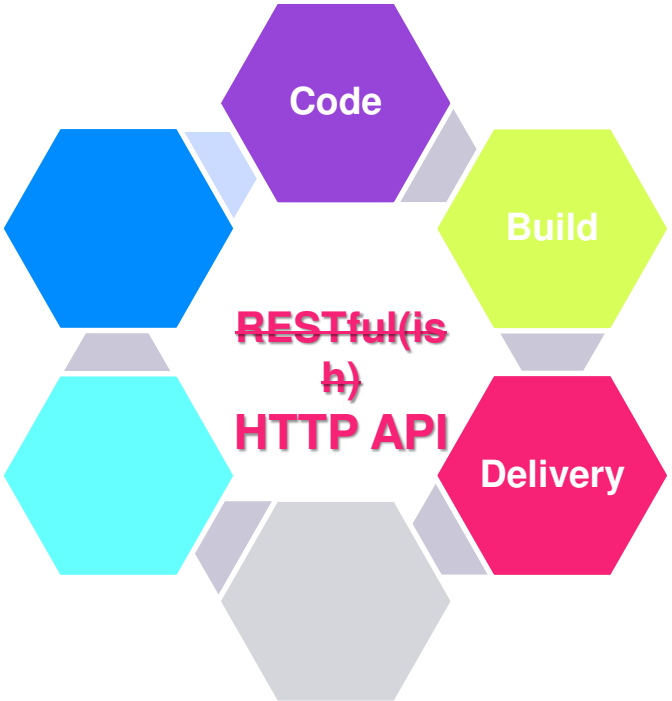
Build

Pull code & dependencies
Run unit tests
Run other tests
Build software
Package software
Build test image*
Upload test image

* Docker image which will run tests in a live environment

ca technologies

Copyright © 2018 CA. All rights reserved.



Delivery

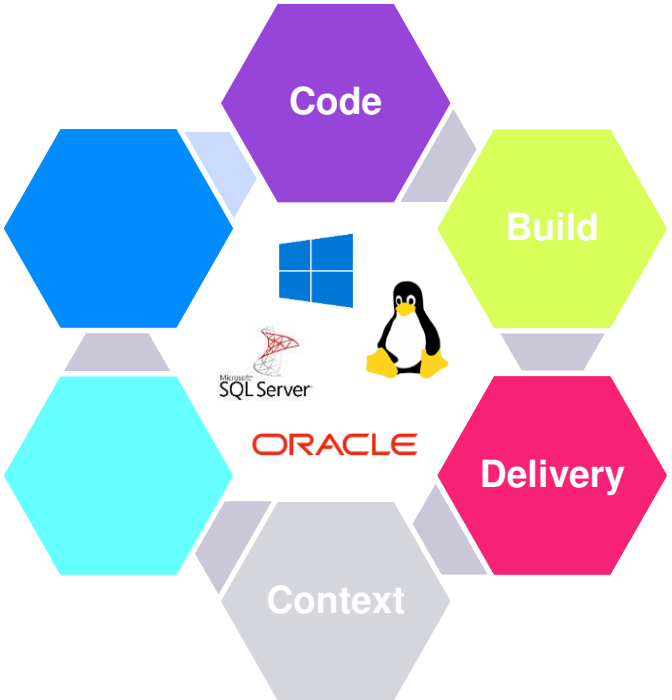
HTTP API
- Build artifact
- Test image*

- Repo
- Branch
- Commit
- Build report

* on the Docker registry

ca technologies

Copyright © 2018 CA. All rights reserved.



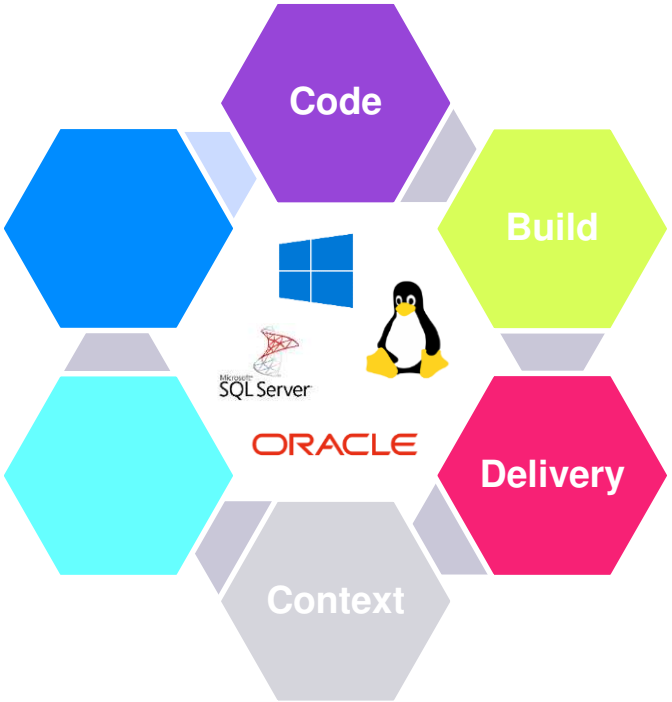
Context

Identify affected product
Map supported OS & DB

- Specific
  Engineer specified context in ticket, commit message, on demand form
- All
  Change was on a release branch
- Random*
  Branch is not a release branch and nothing was specified

ca technologies

Copyright © 2018 CA. All rights reserved.



Context

Verified Product

C C C
C C C

passed all smoke tests on all platforms & DBs

Verified Product

C

C C C
C C C

update with component from branch delivery

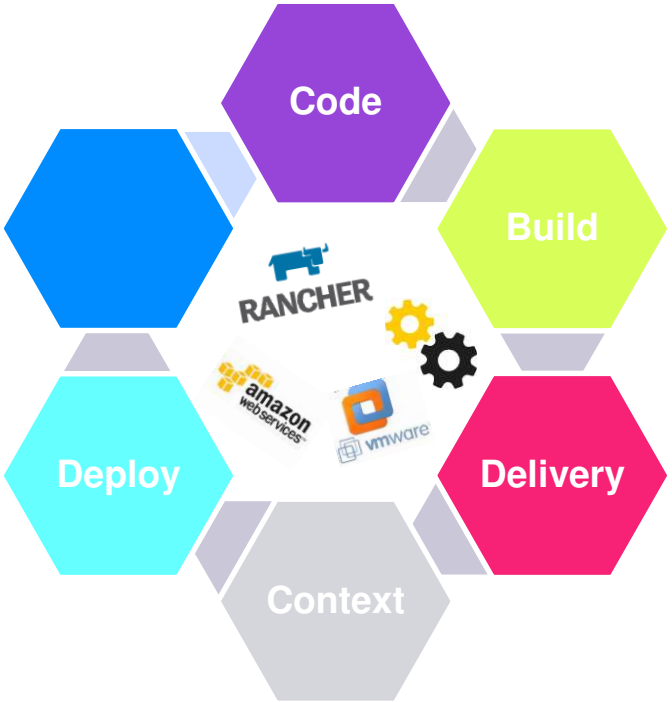
Test Base

C C C
C C C

possible errors only due to this code change

ca technologies

Copyright © 2018 CA. All rights reserved.



Deploy

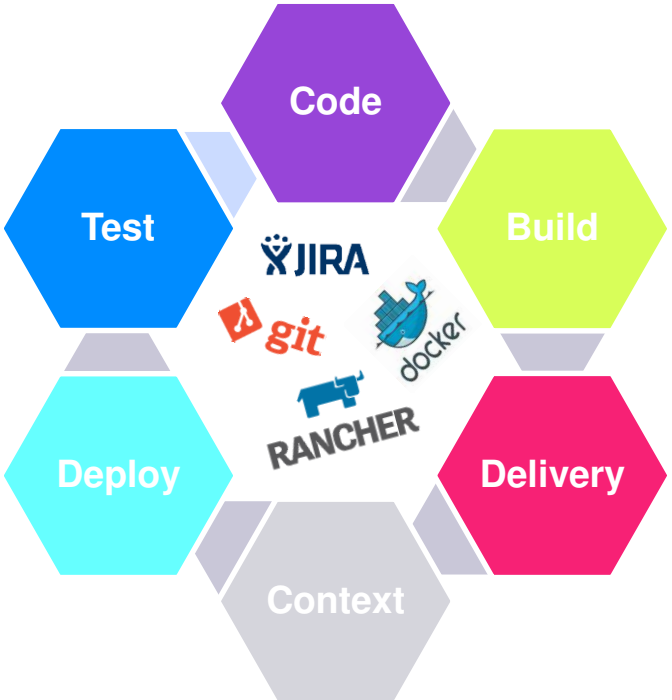
Runtime test dependencies
DB, Application Server, Java, AD, …

Dependent on the software
We execute an automated software deployment process for our products

De-provision after test
With an optional delay for manual testing

Private and/or public cloud

Copyright © 2018 CA. All rights reserved.



Test

Spawn every test image from every component

Your tests guard your component from your changes **and** breaking changes in other components

Unified way to run test

Provided with Docker while enabling dev teams to define their own environment

Results shown in Jira ticket

Parsing via xUnit-style reports
Option to provide additional output

ca technologies

Copyright © 2018 CA. All rights reserved.

Was heißt das für mich als Entwickler?

- Sicherheit, dass meine Änderung funktioniert
- Ich finde heraus, wenn ich eine Schnittstelle breche, von der ich nichts wusste
- Änderungen über mehrere Komponenten leichter
- Perspektivenwechsel: Gesamtanwendung

Positives Feedback aus der Organisation

ca technologies

Copyright © 2018 CA. All rights reserved.

Feedback für das Produkt

Genauso wichtig wie die anderen Themen aber nicht meine Kernkompetenz, daher nur einige Hinweise:

- Mit Kunden sprechen
  - Beziehungen aufbauen und pflegen
  - Prototypen validieren
  - Aktiv Feedback einholen
  - Kontakte zu Power-Usern pflegen
- Nutzerverhalten messen
  - Usability tests
  - Mit Einverständnis Daten über die Nutzung von live-Systemen sammeln

Copyright © 2018 CA. All rights reserved.

Aber...

- Unvollständige Liste von Dingen, die man in der Praxis antrifft und tendenziell nicht erfolgreich neu lösen wird (Ergebnis aus Jahrzehnten Forschung & Praxis)
  - Parser
  - Fundamentale Algorithmen
  - Kryptografie
  - Datenbanken
  - Verteilte Systeme
  - Nebenläufigkeit
- Auf Feedback reagieren ist orthogonal zu Planung, Sorgfalt und Wissen - kein Ersatz!
  - Beispiel: TDD kein Allheilmittel

ca technologies

Copyright © 2018 CA. All rights reserved.

Zusammenfassung

- Wir lernen durch Rückmeldung aus unserer Umgebung
- Viele kleine, rechtzeitige Kurskorrekturen führen oft zum Erfolg
- Wir müssen uns aktiv um Feedback bemühen
  - Auf persönlicher Ebene
  - In unserem Entwicklungsprozess
  - Beim Programmieren (individuell und für das Team)
  - Für das Produkt

ca technologies

Copyright © 2018 CA. All rights reserved.

Danke!

People Management:
Motivation, Produktivität und Teamstrukturen

Ao.Prof. Andreas Rauber
Institut für Softwaretechnik und Interaktive Systeme
http://www.ifs.tuwien.ac.at/~andi

FACULTY OF !NFORMATICS

People Management

Inhalt

- Motivation 2
- Produktionsfaktor Mensch. 15
- Auswahl von Mitarbeitern 32
- Team Management 36

Motivation

Motivation

Summe der Beweggründe, die jemandes Handlung beeinflussen

Unterscheidung in

- **extrinsische Motivation**
  durch äußere Zwänge (Strafen) verursachte Motivation
- **intrinsische Motivation**
  durch die von einer Aufgabe ausgehenden Anreize bedingte Motivation

Motivation

Motivationstheorien versuchen den Motivationsprozess und seine Auswirkungen zu beschreiben.

- Prozesstheorien
  Geben eine Erklärung darüber, wie das Verhalten gesteuert wird, warum ein bestimmtes Verhalten zur Erreichung eines definierten Ziels gewählt wird

- Inhaltstheorien
  Beschreiben Bedürfnisse, Antriebe und Ziele der Menschen

Die spezifischen Theorien und Modelle in der Praxis nur am Rande relevant - hier als Konzepte vorgestellt

Motivation

Prozesstheorien

Individuelle Einflussfaktoren des Motivationsprozesses

- Anspruchsniveau
  - Erwartung an die eigene Leistung und deren Belohnung (Lob, Anerkennung, finanzielle Belohnung etc.)
  - Wird das gesteckte Ziel häufig erreicht, so wird das Niveau meist von selbst (!) entsprechend angehoben

- Persönliche Einstellung
  - Grundhaltung gegenüber der Organisation, der Problemstellung, dem involvierten Umfeld/Personenkreis
  - Bezieht sich sowohl auf die Organisation als Ganzes als auch auf eine Serie spezifischer Einzelreize
  - Negative Einstellungen führen häufig zu Frustration

Motivation

Prozesstheorien (2)

Individuelle Einflussfaktoren des Motivationsprozesses (2)

- Subjektive Wahrscheinlichkeit
  - Abschätzung der Chancen, Erfolg zu haben.
    Abwägung der eigenen Fähigkeiten, der zu erwartenden Schwierigkeiten, dem zur Verfügung stehenden Zeitrahmen etc.
  - Letzterer steht in unmittelbarem Zusammenhang mit dem eigenen Anspruchsniveau: der eigene Qualitätsmaßstab muss erreicht werden können.

Motivation

Prozesstheorien (3)

- Die selben Anreize führen bei verschiedenen Personen aufgrund der völlig unterschiedlichen Einflussfaktoren zu vollkommen unterschiedlichen Motivationen und daher zu unterschiedlichen Reaktionen.
- Weiters ist die Motivation ein und derselben Person abhängig vom jeweiligen Zeitpunkt (Tagesverfassung)
- Jede Erfahrung beeinflusst unmittelbar die darauffolgenden Motivationsprozesse

Motivation

Prozesstheorien (4)

Folgerungen

- Jeder Mitarbeiter muss entsprechend als Individuum behandelt werden
- Mitarbeiter sind nicht eins-zu-eins austauschbar
- Der Umgang mit Personen richtet sich auch nach der jeweiligen Tagesverfassung Diese gilt es zu erkennen und entsprechend darauf zu reagieren.
- Mitarbeiter ‘lernen’ aus ihren Erfahrungen, die sie im Betrieb machen, und richten ihr zukünftiges Verhalten danach.

Motivation

Inhaltstheorien

Grundsätzlich gibt es 2 unterschiedliche Ansätze:

- **Monothematische Theorien**:
  Ein einzelnes Motiv ist ausschlaggebend für die zielgerichtete Dynamik des Menschen.
  (Bsp.: Freud: Sexualtrieb, Adler: Minderwertigkeitskomplex)

- **Polythematische Theorien**:
  Mehrere Grundmotive beeinflussen und bestimmen das Verhalten des Menschen.
  Verschiedene Triebe in unterschiedlicher Rangordnung und Stärke basieren auf einer Reihe von Bedürfnissen.

Motivation

Inhaltstheorien (2)

Während die monothematischen Theorien im Bezug auf die Erklärung von Arbeitszufriedenheit heute keine entscheidende Bedeutung mehr haben, entwickelten sich im Bereich der polythematischen Theorien eine Vielzahl von Modellen und Einflussfaktoren, welche die entscheidenden Prinzipien zu beschreiben versuchen.

Wichtige Modelle in diesem Zusammenhang sind

- Die Motivationspyramide von Maslow
- Das ERG-Modell (Existence, Relatedness, Growth; Alderfer)
- Die Zweidimensionale Arbeitszufriedenheitstheorie von Herzberg

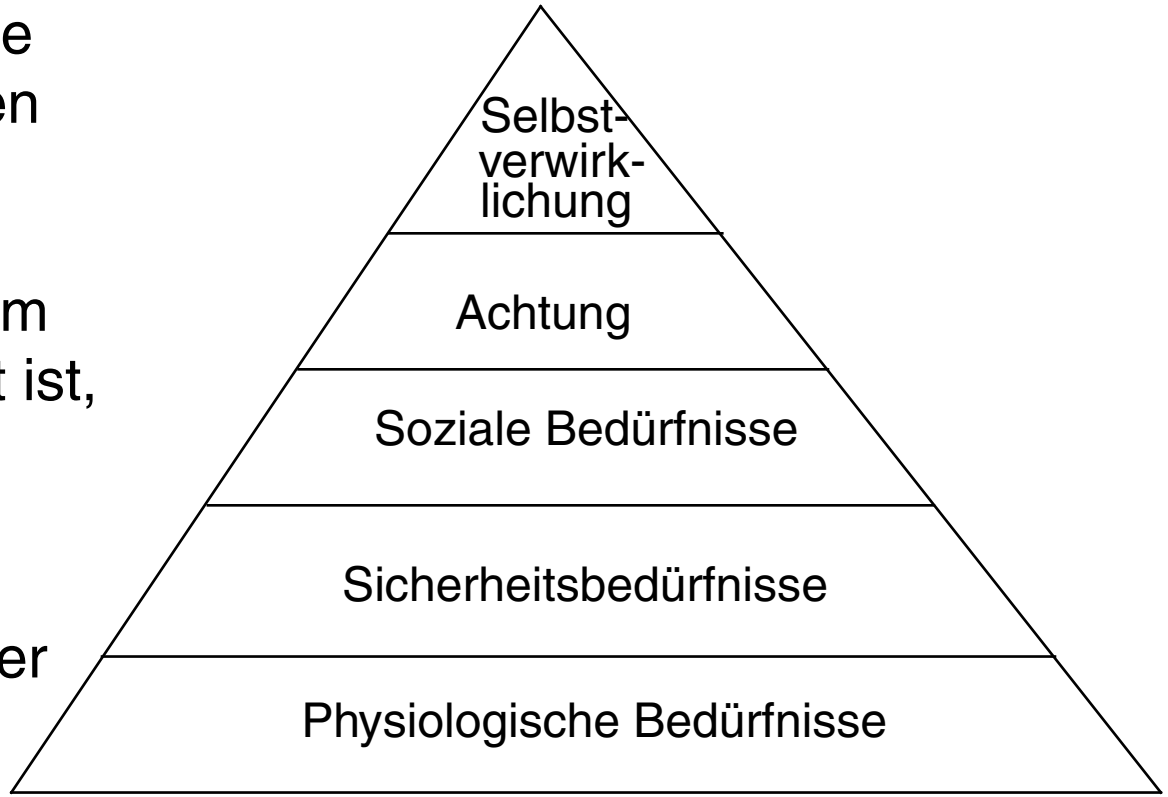
Motivation

Motivationspyramide nach Maslow

Abraham Maslow (1954) geht von 5 Kategorien menschlicher Bedürfnisse aus, welche eine Hierarchie der Dringlichkeiten bilden.

Erst wenn eine Stufe in einem gewissen Ausmaß befriedigt ist, wird die darauf aufbauende verfolgt.

In dieser Form überholt - aber als Ansatzpunkt hilfreich



Motivation

Motivationspyramide nach Maslow (2)

(1) Physiologische
- "lebensnotwendige" Bedürfnisse wie Schlaf, Nahrung

(2) Sicherheit
- Betrifft die Absicherung der Person hinsichtlich Schutz des Lebens, des Eigentums, Lebensstandards, Schutz vor Arbeitslosigkeit, Unfallfolgen bis hin zur Altersvorsorge

(3) Soziale
- Ziel ist die Eingliederung in eine Gruppe und deren soziale Akzeptanz, Zugehörigkeitsgefühl, Freundschaft, Zuneigung

Motivation

Motivationspyramide nach Maslow (3)

(4) Achtungsbedürfnis

- Hier kann zwischen Selbstachtung einerseits und Fremdachtung andererseits unterschieden werden.
- Wird gesteuert durch das Erleben der eigenen Leistung und der resultierenden Anerkennung durch andere (Mitarbeiter); Kompetenz, Status, Respekt

Motivation

Motivationspyramide nach Maslow (4)

(5) Selbstverwirklichung

- Das Verlangen des Individuums, zu verwirklichen, was es potentiell ist, d.h. seine potentiell gegebenen Fähigkeiten und Funktionsmöglichkeiten zu entfalten.
- Das Verhalten eines sich selbst verwirklichenden Menschen ist weitgehend durch die Freude motiviert, die er im Gebrauch und Entwickeln seiner Fähigkeiten findet.
- Menschen mit diesem Bedürfnis versuchen einen starken Einfluss auf ihre Umwelt zu nehmen und sie zu gestalten.
- Sie sind im allgemeinen sehr kreativ, aktiv, leistungsorientiert und versuchen, ihre Fähigkeiten weiterzuentwickeln.
- Ab der Stufe der Selbsverwirklichung kommt es zu einer gewissen Eigendynamik: je stärker dieses Bedürfnis befriedigt wird, desto stärker wird es angestrebt und desto bestimmender wird es für das Verhalten des Menschen.

Motivation

Zusammenfassung

- Grundsätzlich geht es weniger um ein absolut korrektes Modell des Motivationsprozesses, als um das Bewusstsein und die Sensibilität für die vielfältigen Faktoren, die die Motivation und Zufriedenheit mit der Arbeit beeinflussen.
- Personalmanagement betrifft vor allem den Umgang mit Menschen und fordert ein Eingehen auf deren Bedürfnisse.
- Dies beginnt und endet nicht mit dem Betreten bzw. Verlassen der Arbeitsstätte, sondern umfasst das gesamte Umfeld, in dem sie sich bewegen (Einfluss des Privatlebens auf die Arbeit).
- Die Bedeutung dieser ganzheitlichen Betrachtung und der Fähigkeit zum Umgang mit Mitarbeitern rückt immer stärker ins Bewusstsein und gilt mittlerweile als eine der wichtigsten Qualifikationen im Managementbereich!
  (oft noch vor fachlicher Qualifikation bzw. dieser gleichgestellt)

Produktionsfaktor Mensch

Produktivität

- Die Produktivität wird üblicherweise als Verhältnis von erbrachter Leistung zu aufgewendeter Zeit gemessen

- Beliebte Maßnahmen zur Produktivitätssteigerung sind:
  - Personal zu mehr Arbeit anhalten
  - Prozess der Produktentwicklung automatisieren
  - Qualitätsmaßstäbe für das Produkt modifizieren (i.e. reduzieren)
  - Vorgehensweisen standardisieren

- Diese Maßnahmen sind (langfristig) kontraproduktiv
  - Sie führen zu verringerter Arbeitszufriedenheit
  - Die Folge: erhöhte Fluktuation

Produktionsfaktor Mensch

Fluktuation

- Typische Fluktuationszahlen zwischen 33% und 80%, durchschnittliche Beschäftigungszeit zwischen 15 und 38 Monaten

- Mitarbeiter in Unternehmen mit hoher Fluktuation haben oft eine kurzfristige Arbeitsmoral ... Sie sind ohnedies nicht mehr lange beim Unternehmen

Produktionsfaktor Mensch

Fluktuation (2)

Beispiel: Kosten der Fluktuation

- Für die Suche nach einem Mitarbeiter fallen 2 Monatsgehälter als Gebühren für eine Personalvermittlung an (bzw. Fixkosten für die Personalabteilung im Hause)
- Für die Einarbeitung eines neuen Mitarbeiters fallen Kosten an: 3 Monatsgehälter
- Mit anderen Worten: Bei einer durchschnittlichen Beschäftigungsdauer von 2 Jahren betragen die Kosten einer Neueinstellung 20% der Personalkosten

Produktionsfaktor Mensch

Mythos der Austauschbarkeit

- Der Mythos der Austauschbarkeit wird durch die “Maßeinheit” Personen-Monat zur Angabe der Projektdauer beschrieben
- Dahinter steht die Vorstellung, dass sowohl Personen als auch Monate beliebig austauschbar sind
- Diese Vorstellung hat aber nur eine (annähernde) Berechtigung bei Arbeiten, die ohne Kommunikation ausführbar sind

  Beispiel: Holzfällen
  - Angenommen, eine Person braucht zum Fällen eines Baumes 2 Std.
  - Dann schafft diese Person an einem Arbeitstag mit 8 Std. 4 Bäume
  - Sollen 80 Bäume an einem Tag gefällt werden, braucht man also 20 Personen
  - Und selbst bei diesem Beispiel stimmt die Rechnung nicht ...

Produktionsfaktor Mensch

Mythos der Austauschbarkeit (2)

- Der Software-Entwicklungsprozess ist gekennzeichnet durch die Notwendigkeit eines hohen Maßes an Kommunikation zwischen den beteiligten Personen
- Je mehr Personen beteiligt sind, desto höher ist der notwendige Aufwand für Kommunikation und desto geringer ist somit die individuelle Produktivität
- Weiters müssen einem Projekt neu hinzugefügte Personen erst einmal in die Ziele des Projektes eingeweiht werden, sie müssen mit den bisherigen Produkten vertraut gemacht werden, sie müssen sich unter Umständen erst einmal die geforderten Fähigkeiten aneignen

Das Resultat könnte man folgendermaßen beschreiben:

“Adding manpower to a late software projekt makes it later” (Brooks’ Law)

Produktionsfaktor Mensch

“Kriegsspiele für Programmierer”

i.e. die Bezeichnung für eine Serie von Produktivitätsstudien, die mit Teams von Programmierern aus unterschiedlichen Organisationen durchgeführt wurden[1]

Ziel dieser Studien war es, die Faktoren zu finden, die Einfluss auf die Produktivität haben

Diese Studien waren als Wettbewerb inszeniert, wo Teams eine Reihe Programmier- und Testaufgaben in möglichst kurzer Zeit und mit möglichst wenig Fehlern zu lösen haben

1. T. DeMarco, T. Lister: *Wien wartet auf Dich - Der Faktor Mensch im DV-Management*. München: Hanser-Verlag. 1991.

Produktionsfaktor Mensch

“Versuchsanordnung” während der Studie

- Je zwei Programmierer einer Firma bilden ein Team
  Die Teammitglieder arbeiten aber nicht zusammen, sie stehen in Konkurrenz zueinander und zu den übrigen Teams
- Die Teammitglieder müssen genau die gleiche Aufgabe lösen: Ein Programm nach einer vorgegebenen Spezifikation entwerfen, implementieren und testen
- Während sie die Aufgabe lösen, notieren die Teilnehmer die dafür benötigte Zeit
- Wenn die Testarbeiten beendet sind, werden die Ergebnisse einem standardisierten Abnahmetest unterzogen
- Die Teams arbeiten in ihrer gewohnten Umgebung, zu ihrer normalen Arbeitszeit, mit den gewohnten Programmiersprachen, Werkzeugen und Rechnern

Produktionsfaktor Mensch

Die Ergebnisse der Studien

- Die besten Teilnehmer waren um einen Faktor 10 besser als die schlechtesten
- Die besten Teilnehmer waren um einen Faktor 2.5 besser als der Durchschnitt
- Die überdurchschnittlichen Teilnehmer übertreffen die unterdurchschnittlichen Teilnehmer im Verhältnis 2:1

Diese Größenordnungen gelten für fast jede Leistungsmetrik
e.g. benötigte Zeit, Anzahl der Fehler

Produktionsfaktor Mensch

Faktoren ohne Einfluss auf die Produktivität

- Programmiersprache
  - Die Qualität der Lösungen war gleich, unabhängig von der gewählten Sprache
  - Innerhalb der einzelnen Sprachgruppen zeigte sich die gleiche Leistungsverteilung wie über alle Sprachen hinweg
  - Die einzige Ausnahme bildeten die Assembler-Programmierer

- Berufserfahrung
  - Kein sichtbarer Zusammenhang zwischen den Jahren an Berufserfahrung und den erbrachten Leistungen
  - Die einzige Ausnahme waren Personen, die mit der gewählten Programmiersprache weniger als 6 Monate Erfahrung hatten

Produktionsfaktor Mensch

Faktoren ohne Einfluss auf die Produktivität (2)

- Anzahl der Fehler
  - Fast ein Drittel bewältigten die Aufgabe mit null Fehlern
  - Diese Gruppe brauchte nicht mehr Zeit als die übrigen Teilnehmer (sie waren zum Teil sogar schneller)

- Gehalt
  - Nur eine sehr schwache Beziehung zwischen Gehalt und Leistung
  - Die bessere Hälfte der Teilnehmer verdiente ca. 10% mehr als die schlechtere Hälfte
  - Die Abweichungen der Leistung auf jedem Gehaltsniveau war etwa genauso groß wie die Abweichungen im gesamten Spektrum

Produktionsfaktor Mensch

Faktoren mit Einfluss auf die Produktivität

- Teampartner
  - Hohe Korrelation zwischen den Leistungen der beiden Teampartner
  - Personen aus dem gleichen Unternehmen (d.h. gleiches Umfeld und gleiche Firmenkultur) bringen fast identische Leistungen

- Arbeitsplatz
  - Qualität des Arbeitsplatzes hat starken Einfluss auf die Leistung
  - Die Qualität des Arbeitsplatzes wurde in der Studie untersucht, indem die Teilnehmer Fragen über ihren Arbeitsplatz beantworten mussten

Produktionsfaktor Mensch

Der Einfluss des Arbeitsplatzes

Arbeitsplatz	bestes Viertel	schlechtestes Viertel
Wieviel Arbeitsplatz steht zur Verfügung?	7 m^2	4.1 m^2
Ist es annehmbar ruhig?	57% JA	29% JA
Ist Ihre Privatsphäre gewahrt?	62% JA	19% JA
Können Sie Ihr Telefon abstellen?	52% JA	10% JA
Können Sie Ihr Telefon umleiten?	76% JA	19% JA
Werden Sie oft von anderen Personen grundlos gestört?	38% JA	76% JA

Produktionsfaktor Mensch

Ruhe am Arbeitsplatz: der U-Faktor

‘In Fahrt’: Zustand tiefer, fast meditativer Versunkenheit (in die Arbeit): voller Arbeitseinsatz, man merkt die Arbeit nicht, sie geht leicht von der Hand, das Gefühl für die Zeit geht verloren.

Eintauchphase: es braucht ca. 15 Minuten ungestörter Konzentration, um diesen Zustand zu erreichen.

Umwelt-Faktor: Anwesenheitszeit vs. Zeit mit vollem Arbeitseinsatz

$$U = \frac{\text{ungestörte Stunden}}{\text{Stunden körperlicher Anwesenheit}}$$

Produktionsfaktor Mensch

Arbeitsplatzgestaltung

- **ArbeitsPLATZ**: Arbeitskojen vs. Büroräume
- **Privatsphäre**: Großraumbüros vs. Arbeitszimmer
- **Beleuchtung**: Neonröhren vs. Fenster (vgl. Bürogebäude - Hotels)
- **Außenraum**: geschlossener Gebäudekomplex vs. (Dach)Terassen und Innenhöfe
- **Innenraum**: Gestaltung: Farbe, Größe, Beleuchtung, Lärmschutz, Material (Fußboden...), Belüftung, Grünpflanzen,...
- **Kommunikationsräume**: Einzelkämpfer vs. Teambildung, Gesprächszonen, informelle Treffpunkte, Arbeitsumfeld
- **Individualität**: Corporate Identity vs. individuellen Gestaltungsmöglichkeiten
- **Flexibilität**: Anpassungsmöglichkeiten an Gruppenstrukturen, Module, Trennwände

Produktionsfaktor Mensch

Arbeitsplatz und Kosten

> “Bremsen Sie sich doch einmal ein. Wir reden hier über eine Menge Geld. Wenn wir unseren Entwicklern so viel Platz und Lärmschutz und sogar Privatsphäre geben, dann kostet das gut und gern 50 Dollar pro Person pro Monat. Multiplizieren Sie das mal mit der Anzahl unserer Entwickler und Sie kommen auf Beträge von Zehntausenden von Dollars. So viel Geld können wir nicht ausgeben. Ich bin genauso für Produktivität wie Sie, aber haben Sie unsere Ergebnisse des 3. Quartals schon gesehen?”

T. DeMarco, T. Lister: *Wien wartet auf Dich - Der Faktor Mensch im DV-Management*. München: Hanser-Verlag. 1991.

Produktionsfaktor Mensch

Arbeitsplatz und Kosten (2)

> "In einer Organisation mit starker Fluktuation ist keiner bereit, langfristig zu denken. Wenn es sich bei der Organisation um eine Bank handelt, wird sie vielleicht der Entwicklungs-GmbH Uganda einen Kredit gewähren, weil sich die 22 Prozent Kreditzinsen im Abschlussbericht für dieses Quartal gut machen. Selbstverständlich wird die EGU einige Zeit darauf Konkurs anmelden, aber wen kümmert das schon, wenn man selbst nicht mehr da ist?"

T. DeMarco, T. Lister: *Wien wartet auf Dich - Der Faktor Mensch im DV-Management*. München: Hanser-Verlag. 1991.

Produktionsfaktor Mensch

Auswandern und Verstecken

> "Wenn Sie einen Blick in einen Konferenzraum werfen, kann es ihnen passieren, dass dort 3 Mitarbeiter in aller Stille sitzen und arbeiten. Wenn Sie nachmittags gegen 3 Uhr in die Kantine kommen, finden sie vielleicht einige Mitarbeiter dort, jeweils einen pro Tisch, umgeben von seinen Unterlagen. Einige Mitarbeiter können Sie vielleicht gar nicht aufspüren. Sie verstecken sich, um ihre Arbeit erledigen zu können. Wenn Ihnen das in Ihrer Firma schon passiert ist, dann sollten Sie das als Alarmzeichen sehen. Kosten reduzieren durch Raumeinsparungen kann sehr teuer werden."

T. DeMarco, T. Lister: *Wien wartet auf Dich - Der Faktor Mensch im DV-Management*. München: Hanser-Verlag. 1991.

Auswahl von Mitarbeitern

Beispiel: Einstellungsgespräch mit einem Jongleur

Zirkusdirektor: Wie lange jonglieren Sie schon?
Bewerber: Ungefähr seit 6 Jahren
Zirkusdirektor: Können Sie mit 3 Bällen, 4 Bällen und mit 5 Bällen umgehen?
Bewerber: Ja, ja und ja.
Zirkusdirektor: Arbeiten Sie auch mit brennenden Objekten?
Bewerber: Selbstverständlich
Zirkusdirektor: ... mit Messern, Äxten, offenen Zigarettenschachteln und Hüten?
Bewerber: Ich jongliere mit allem.
Zirkusdirektor: Und haben Sie auch ein paar flotte Sprüche zur Untermalung Ihres Programms auf Lager?
Bewerber: Sie lachen sich tot dabei.
Zirkusdirektor: Das klingt alles sehr gut. Ich glaube, Sie haben den Job.
Bewerber: Hmmm... soll ich Ihnen nicht eine Kostprobe geben?
Zirkusdirektor: Oh, daran habe ich gar nicht gedacht.

T. DeMarco, T. Lister: *Wien wartet auf Dich - Der Faktor Mensch im DV-Management*. München: Hanser-Verlag. 1991.

Auswahl von Mitarbeitern

(1) Job specification
- Auflistung der Anforderungen des Jobs

(2) Job holder profile
- Auflistung der Anforderungen an die Person, die den Job ausfüllen soll
- e.g. Qualifikationen, Berufserfahrung, Ausbildung, ...

(3) Bewerbungen einholen
- Schalten der Stellenausschreibung in einem geeigneten Medium
- Das geeignete Medium lässt sich u.U. aus dem “Job holder profile” ableiten

(4) Auswertung der Bewerbungen
- Vergleich der Lebensläufe mit dem “Job holder profile”
- Ungeeignete Kandidaten frühzeitig ausscheiden

(5) Auswahl der Mitarbeiter

Auswahl von Mitarbeitern

Wege zur Auswahl von Mitarbeitern

- Portfolio
  - Bewerber bringen eine Auswahl von Produkten ihrer bisherigen Tätigkeiten
    (e.g. Entwürfe, Implementierungen, ...)

- Eignungstests
  - Achtung: Eignungstests überprüfen nur Fähigkeiten, die unmittelbar nach der Einstellung gefordert werden (e.g. statistische Analyse, Programmierung, ...)

Auswahl von Mitarbeitern

Wege zur Auswahl von Mitarbeitern (2)

- Anhörungen
  - Bewerber präsentieren ca. 15 Minuten lang Aspekte ihrer bisherigen Tätigkeit
  - Bei dieser Präsentation sind die zukünftigen Teampartner die Zuhörer
  - Die endgültige Entscheidung über eine Aufnahme eines Bewerbers liegt zwar bei der Projektleitung, aber die Meinung der Teampartner ist ein wichtiger Hinweis, ob eine Integration des Bewerbers in die Gruppe möglich ist

- Interview
  - Mehrere Interview-Runden mit den Bewerbern
  - Kleine Anzahl von Interviewern - eine große Anzahl verhindert oftmals das Aufkommen einer Diskussion bzw. Folgefragen

Team Management

Sieben Wege, die Teambildung zu verhindern

- Defensives Management
  - jede Entscheidung wird vom Projektmanagement getroffen, die Teammitglieder werden entmündigt
- Bürokratie
  - das sinnlose Produzieren von Projektdokumenten, das die Teammitglieder von ihrer eigentlichen Arbeit abhält
- Physikalische Trennung
  - die räumliche Trennung von Personen, die eigentlich zusammen arbeiten sollen
- Zersplitterung der Zeit der Mitarbeiter
  - die Aufteilung der Mitarbeiter auf mehrere Projekte

Team Management

Sieben Wege, die Teambildung zu verhindern (2)

- Qualitätsreduktion der Produkte
  - die Qualitätsreduktion erfolgt unter dem Argument der Kostenreduktion; der Effekt ist eine geringere Identifikation der Mitarbeiter mit dem Projekt

- Scheintermine
  - unrealistische Termine bewirken nur, dass sich die Mitarbeiter nicht ernst genommen fühlen und daher nur eine geringe Identifikation mit dem Projekt aufbringen werden

- Cliquenkontrolle
  - Veränderung der Teamstruktur während des Projektes bzw. Zerschlagung von Teams nach dem Projektende

Team Management

Formen der Team-Organisation

- die klassische hierarchische Organisation
- die typische Matrix-Organisation
- das “Chef-Programmierer”-Team
- das offen strukturierte Team
- das SWAT-Team
- das XP-Team

Team Management

Richtlinien zur Team-Organisation

- Setzen Sie bessere Leute ein :-)
- Setzen Sie weniger (!) Leute ein
- Stellen Sie den Teammitgliedern Aufgaben, die zu ihren Fähigkeiten passen
- Stellen Sie den Teammitgliedern Aufgaben, die zu ihren Interessen passen
- Achten Sie mittelfristig auf die persönliche Entwicklung der Mitarbeiter
- Achten Sie auf eine balancierte und harmonische Mischung
- Lösen Sie sich von Leuten, die nicht zum Team passen

People Management

Zusammenfassung

- Mitarbeiter sind mehr als nur Ressourcen, daher kommt der Mitarbeitermotivation besondere Bedeutung zu
- Bei Versuchen zur Steigerung der Produktivität ist zu beachten, ob damit nicht auch gleichzeitig die Fluktuation erhöht wird
- Hohe Fluktuation hat langfristig negative Auswirkung auf das Unternehmen
- Die Qualität des Arbeitsplatzes sowie die Qualität der Teammitarbeiter sind die einzigen Faktoren, die Einfluss auf die Produktivität haben