Start: 15:15

SEPM Vorlesung – Block 4
Software Engineering & Projektmanagement

Software Engineering Phasen

Dietmar Winkler

Vienna University of Technology
Institute of Information Systems Engineering
Software Engineering Group

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.

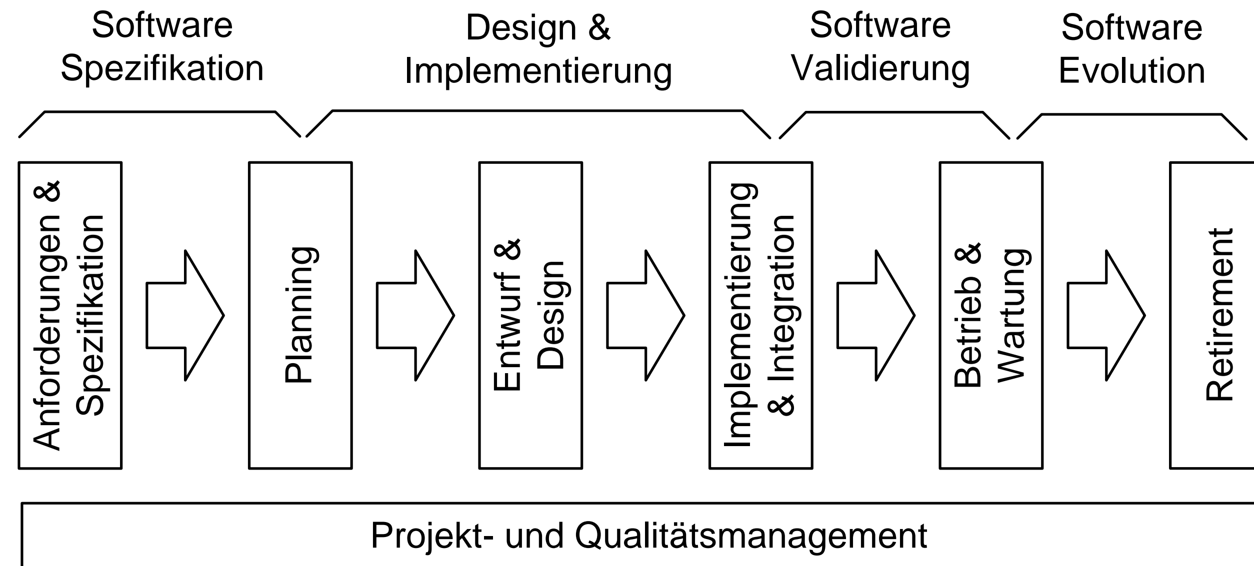
- Softwareprodukte umfassen ALLE Produkte im Rahmen der Softwareentwicklung, z.B. Spezifikation, Code, Testfälle, Protokolle.

- 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 in a Nutshell

TU WIEN

- 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)

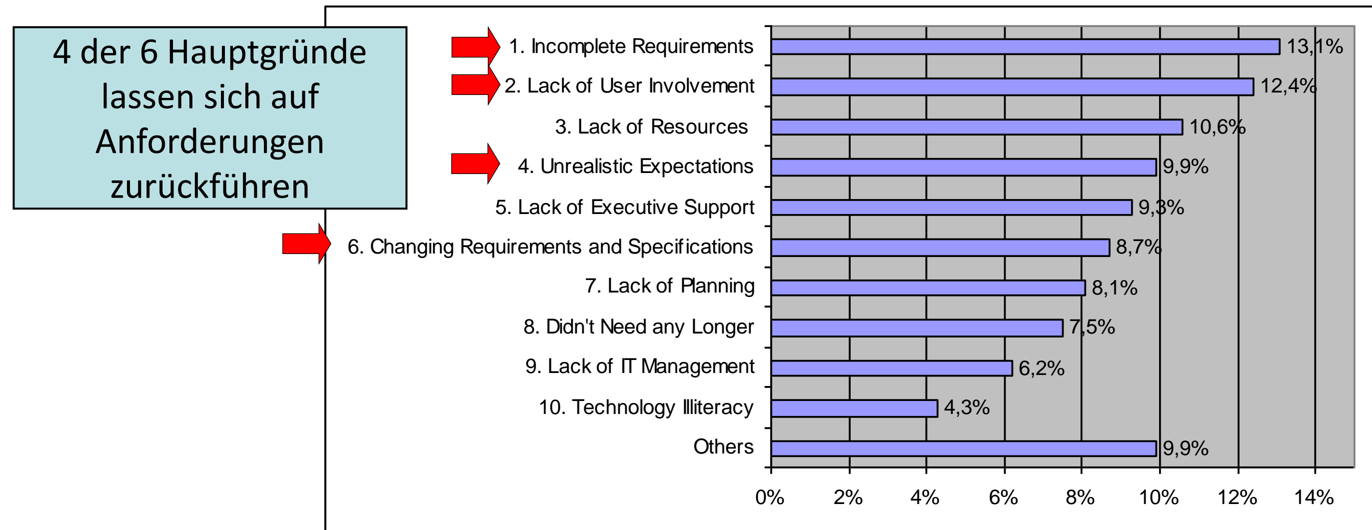
- Durch Anforderungen wird ein gemeinsames Verständnis hergestellt, was das Produkt können soll.
- Sie repräsentieren die „reale Welt“ und drücken das gewünschte Verhalten aus Nutzersicht aus.
- Berücksichtigung unterschiedlicher Stakeholders (Kunde/Anwender, Entwickler, usw.)
  - Unterschiedliche Betrachtungsweisen 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).

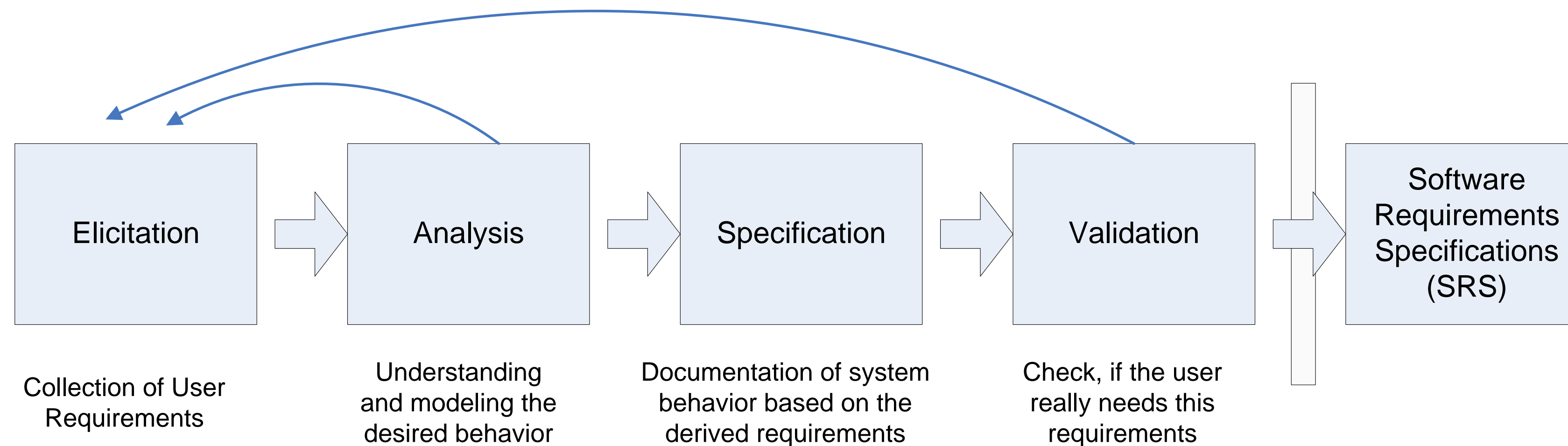


Table of Contents

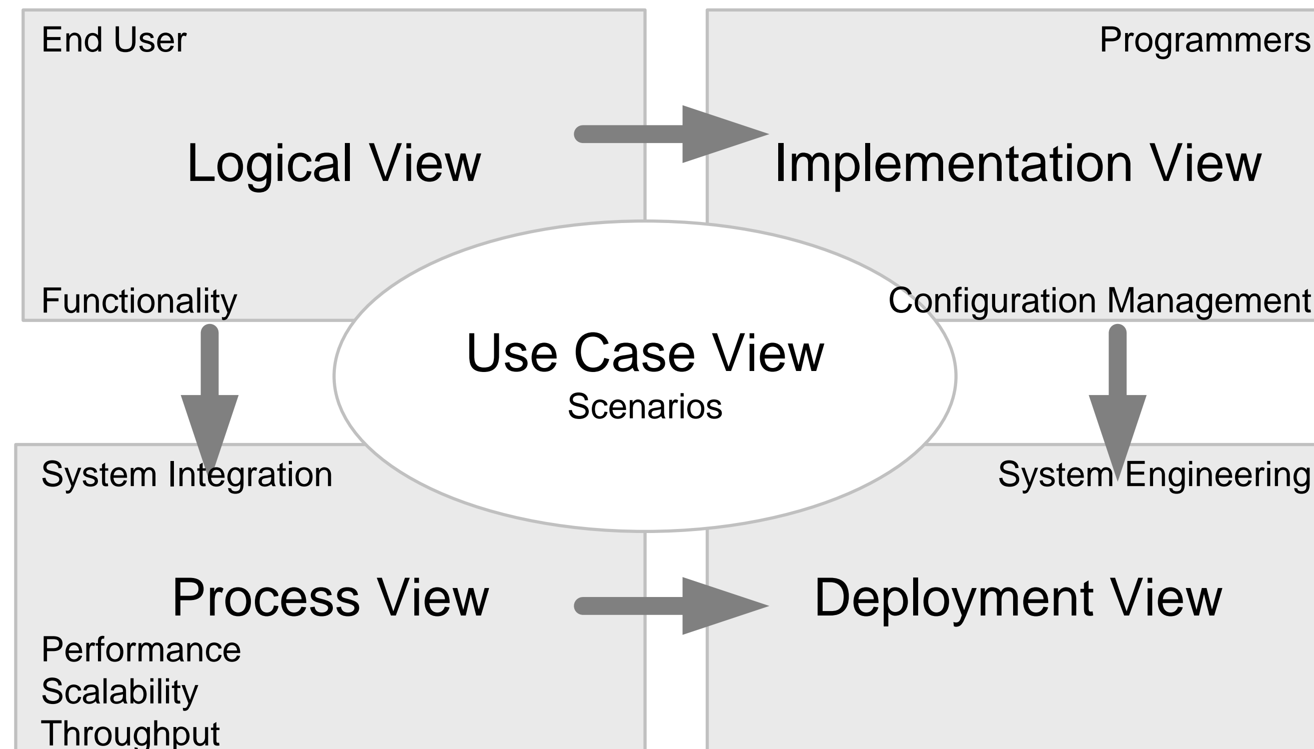
- Software Life-Cycle Prozess im Überblick
- Anforderungen & Spezifikation
- Entwurf & Design
  - 4+1 Model View
  - Ausgewählte Designprinzipien
- Implementierung & Integration
- Qualitätssicherung und Software Testen
- Wartung, Evolution und Retirement

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 P.: The Rational Unified Process: An Introduction, Addison-Wesley Longman, 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.

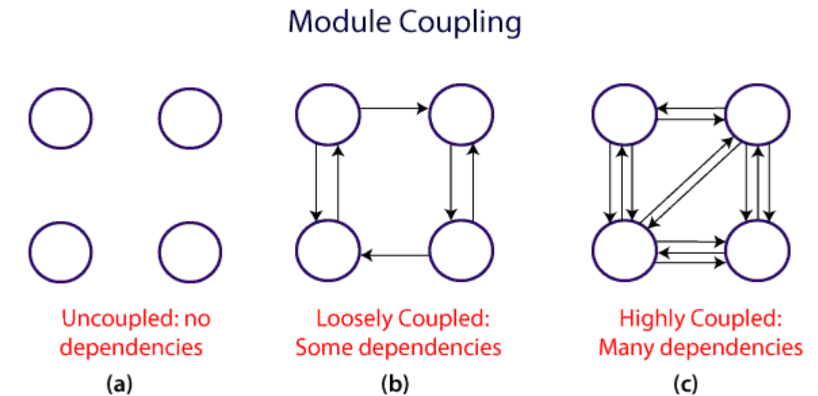
UML 2: Use Case Diagram + Beschreibung, Aktivitätsdiagramme.

→ **Einige wichtige Design Prinzipien ..**

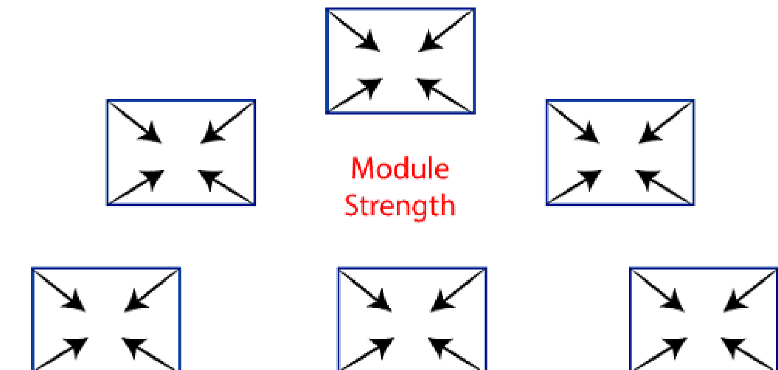
Design-Entscheidung - Komplexität: Coupling vs Cohesion

- 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.



https://www.javatpoint.com/software-engineering-coupling-and-cohesion

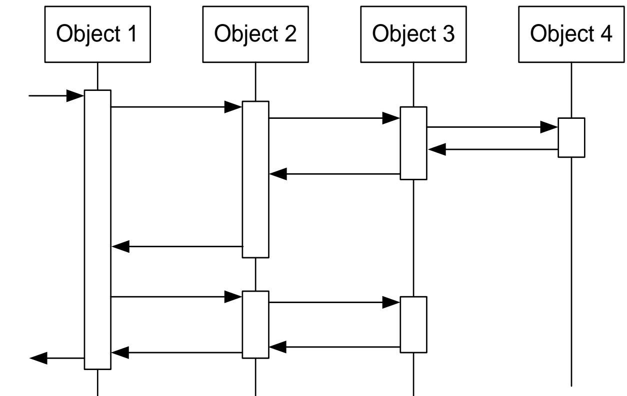


Cohesion= Strength of relations within Modules

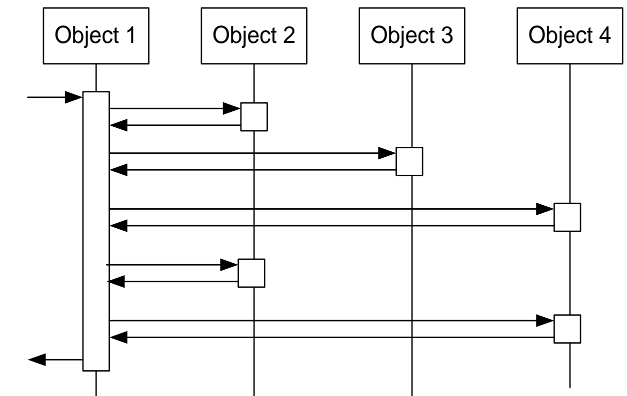
https://www.javatpoint.com/software-engineering-coupling-and-cohesion

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

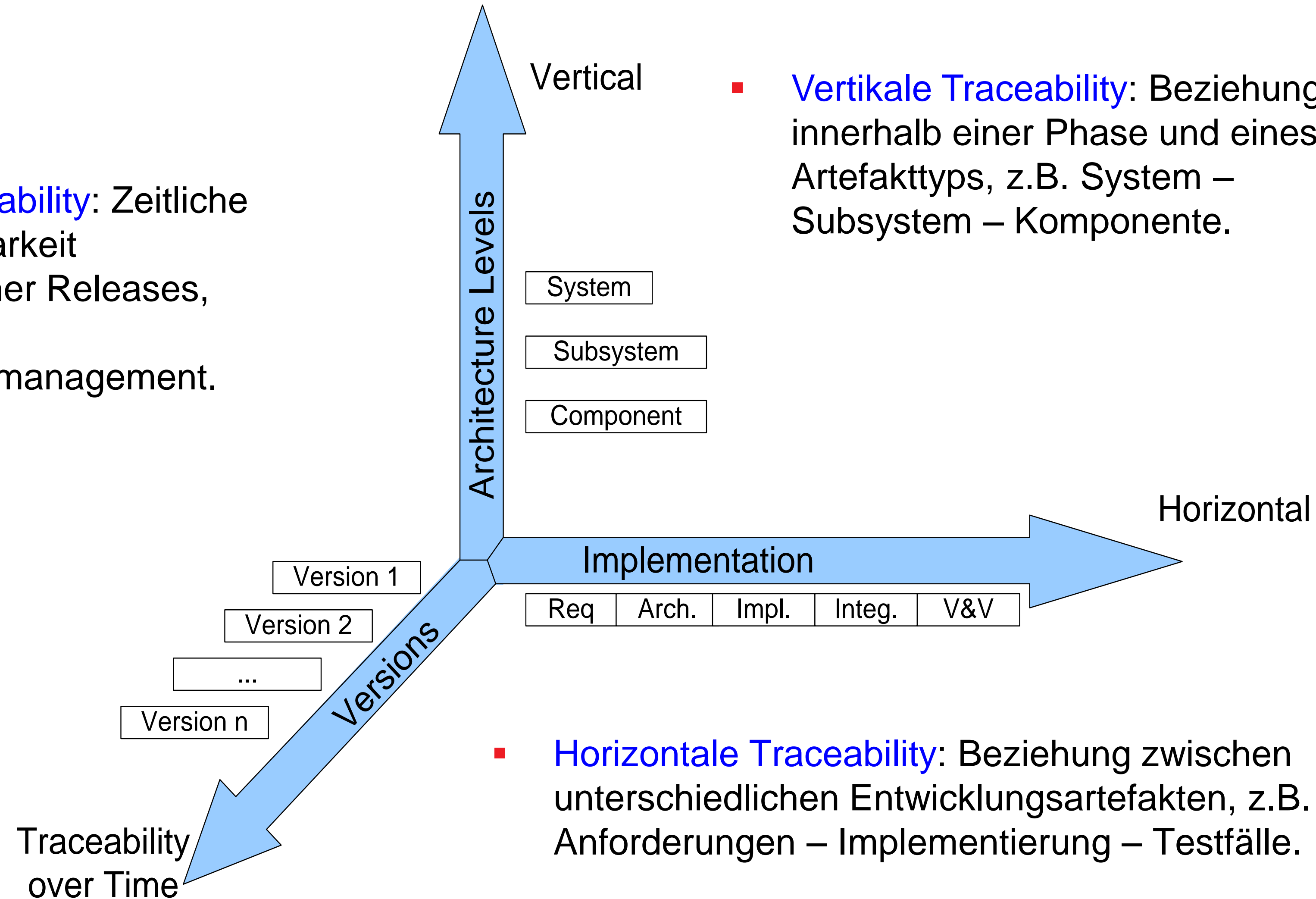
- “Software Engineering is a team-oriented contact sport” [Boehm, 2002].
- Die Zusammenarbeit von unterschiedlichen Stakeholdern (auch innerhalb von Entwicklerteams) erfordert auch im Bereich der Implementierung eine
  ausreichende und einheitliche Dokumentation.
- Standardisierung im Software Engineering (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 G., Dussa-Zieger K., Wentzel P-R.: Traceability – Anspruch und Realität, Proc. of SEE Conference, Bern, Switzerland, 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.

Chatrath G., Dussa-Zieger K., Wentzel P-R.: Traceability – Anspruch und Realität, Proc. of SEE Conference, Bern, Switzerland, 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, Shari Lawrence; Atlee, Joanne M.: Software Engineering – Theory and Practice, Third Edition 2006, Pearson Education.

Big-Bang Integration

Vorgehensweise:

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

Vorteile:

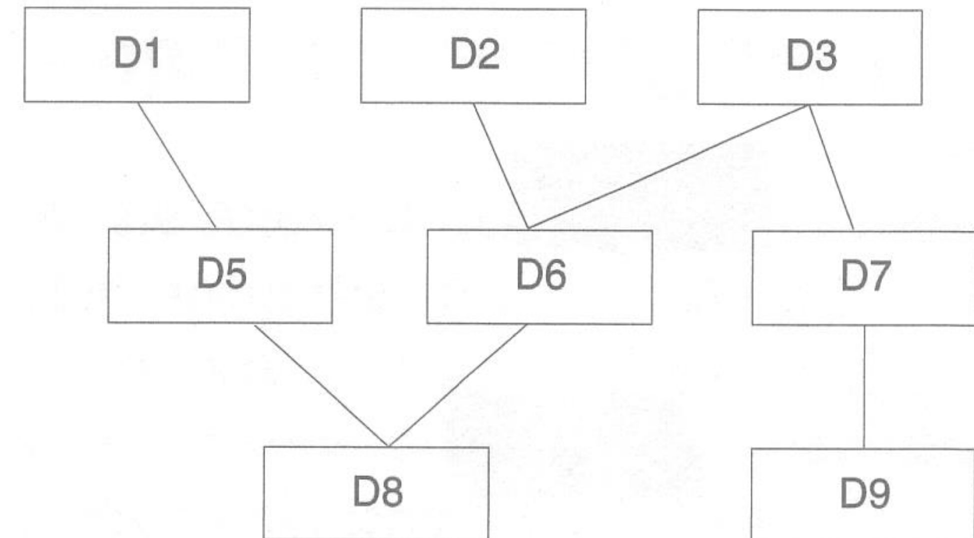
- Keine zusätzlichen Testaufwände für Test-Stubs (um fehlende Funktionalität zu simulieren) → alle Komponenten sind verfügbar.

Nachteile:

- Fehler sind sehr schwer zu lokalisieren (z.B. Seiteneffekte).
- Hohes Risiko bei der Integration.

Anwendung:

- Kleine und überschaubare Produkte.



Top-Down Integration

Vorgehensweise

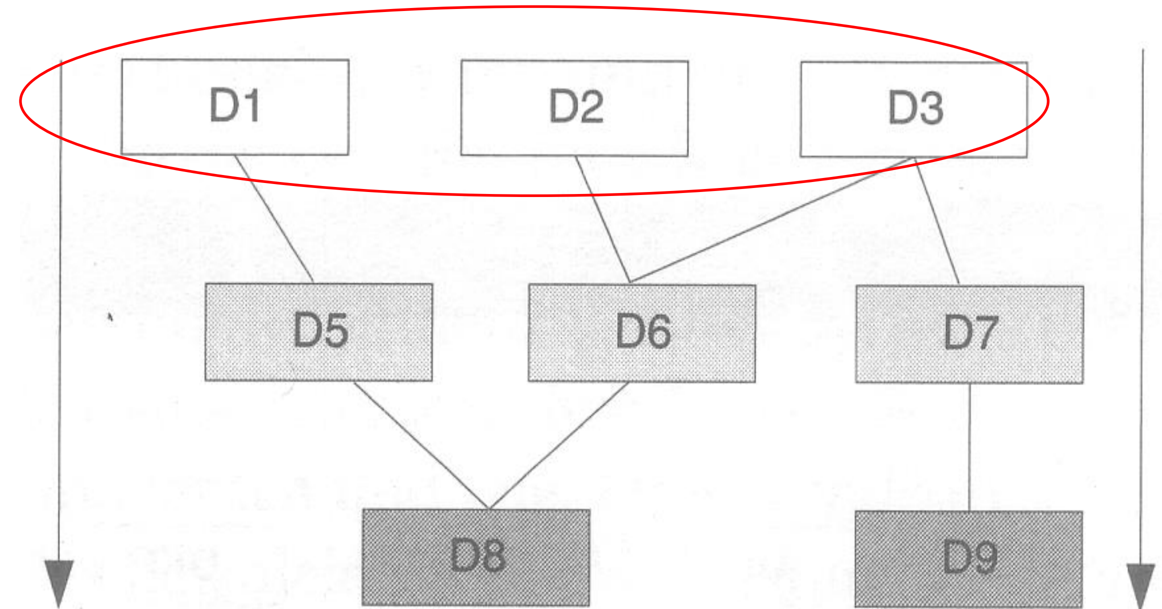
- Schrittweise Integration (z.B. D1, D2, D3), ausgehend von den Business Cases.

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:

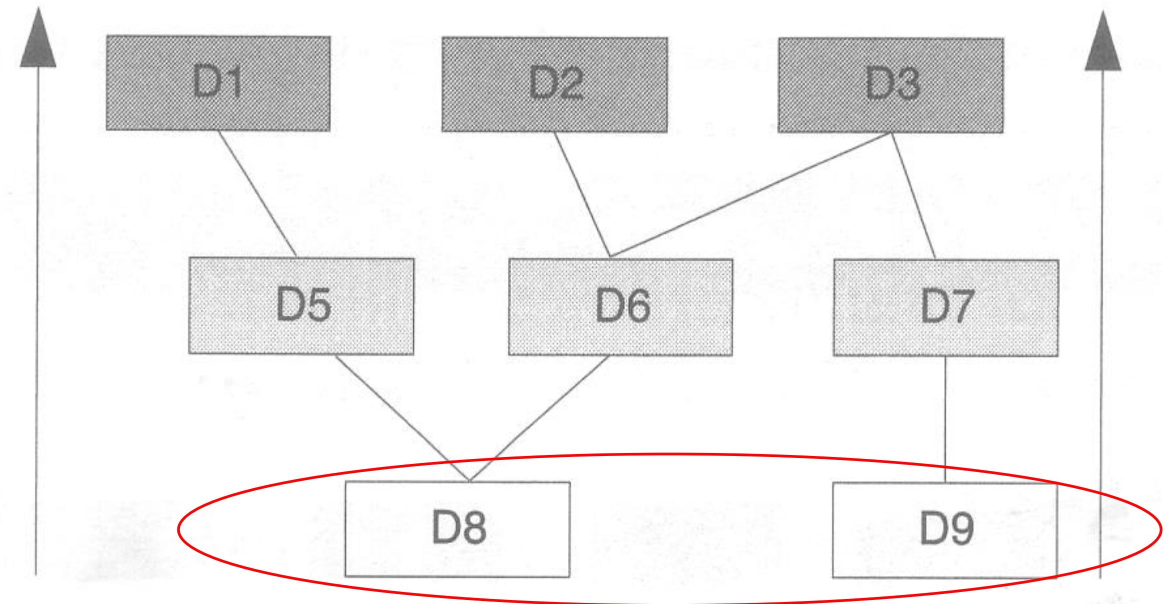
- Schrittweise Integration,
  beginnend bei der Hardware (z.B. D8/D9).

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.

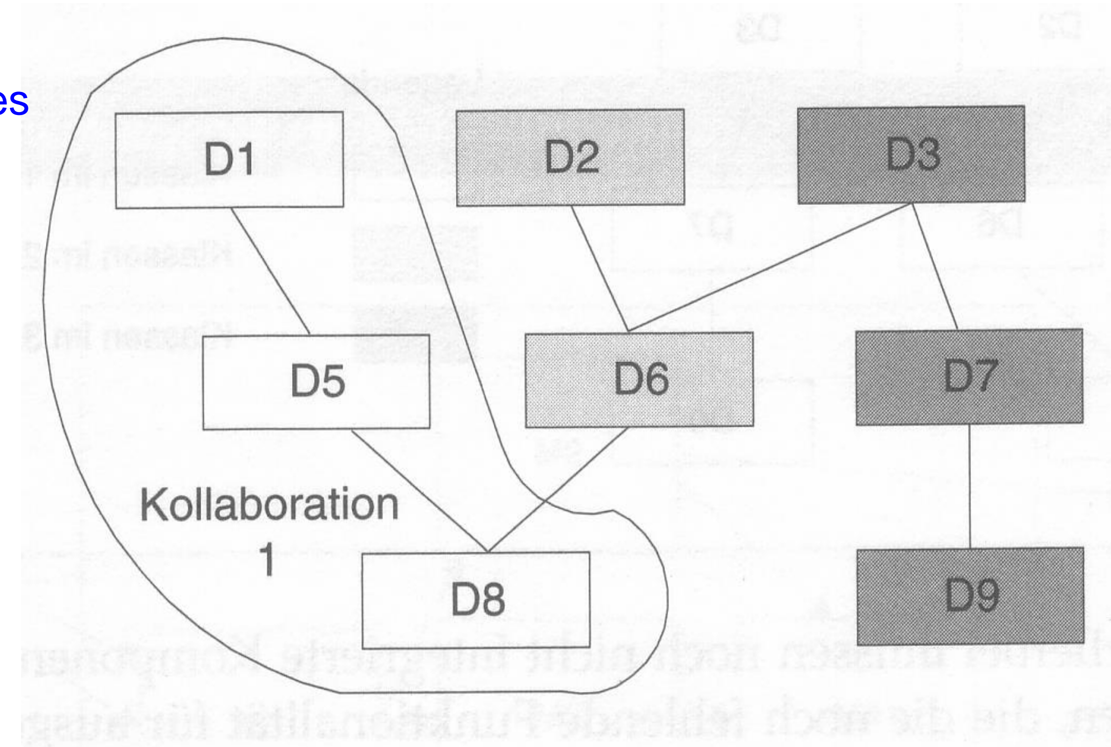


Table of Contents

- 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.

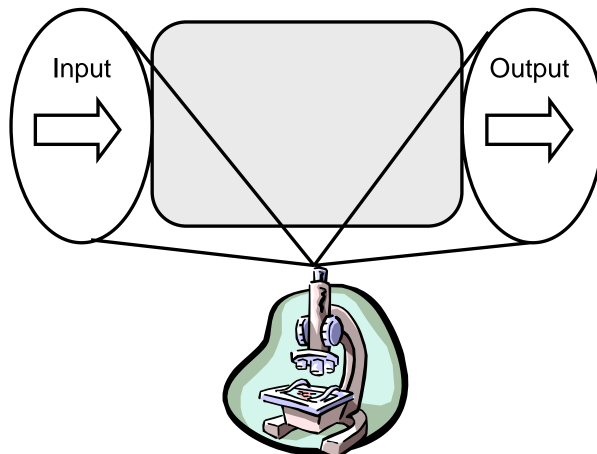
- Installationstest:
  Identifizieren von Fehlern während der Installation.

Teststrategien

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

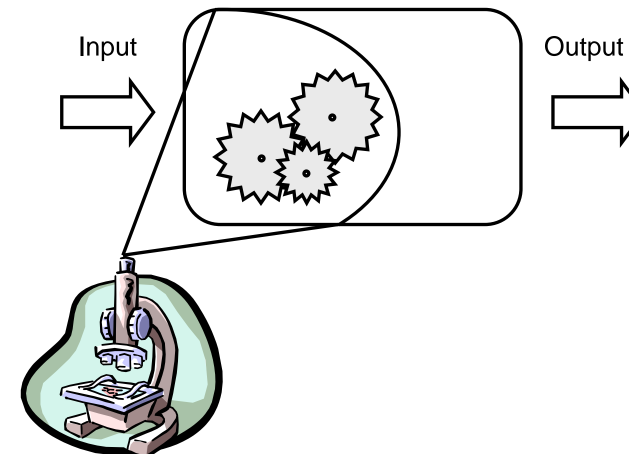
Black Box Tests

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



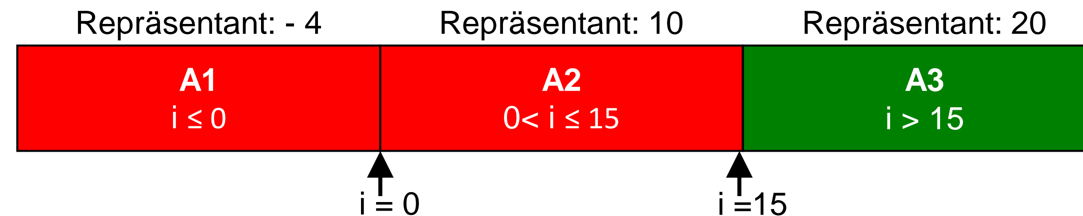
White Box Tests

- 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:

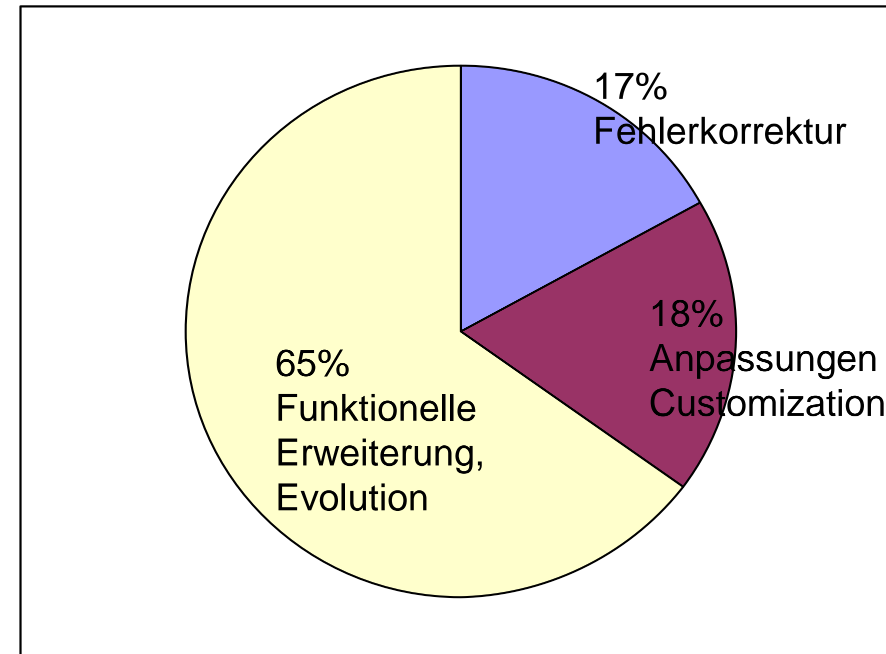
- Produktpflege und Verbesserung (Perfective):
  Produktverbesserung (Erweiterungen, Verbesserung der Effizienz).
- Vorbeugende Wartung (Preventive):
  Verbesserung im Hinblick auf zukünftige Wartung (z.B. Ergänzung der Dokumentation).

	Correction	Enhancement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

Van Vliet, Hans: Software Engineering – Principles and Practice, 2nd Edition, 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.