Holy bible of ESE

Nora Jasharaj

February 3, 2025

Contents

1 Anforderungsanalysen und Spezifikation **7**
- 1.0.1 Definition . . . 7
- 1.0.2 Funktionalität . . . 7
- 1.0.3 Anwendungsdomänen . . . 7
- 1.0.4 Klassifikation der Anforderungen . . . 7

1.1 Spezifikation . . . 7
- 1.1.1 Definition . . . 7
- 1.1.2 Komponenten . . . 8
- 1.1.3 Lasten und Pflichtenheft . . . 8
- 1.1.4 Ziel . . . 8
- 1.1.5 Prozess . . . 8
- 1.1.6 Anforderungsquellen . . . 8
- 1.1.7 Befragung/ Story-writing workshop . . . 8
- 1.1.8 Personas . . . 8
- 1.1.9 Spezifikationsmethoden . . . 8
- 1.1.10 Wichtige Merkmale der Anforderungsspezifikation . . . 9
- 1.1.11 **INVEST** . . . 9
- 1.1.12 Begriffslexikon . . . 9

1.2 Use cases . . . 9
- 1.2.1 Definition . . . 9
- 1.2.2 **Anwendungsfalldiagramm** . . . 10
- 1.2.3 Nachteile bei gezwungener gleicher Granularität am Anfang . . . 10
- 1.2.4 Definition of done . . . 10
- 1.2.5 **Außerdem wichtig für den Entwurf abgesehen von den Anforderungn (HIER falls du es nicht findest :))** . . . 10

2 Grundbegriffe **13**

2.1 Geschichte des Softwareengineering . . . 13
- 2.1.1 Softwarekrise . . . 13
- 2.1.2 Beiträge der Frühzeit . . . 13
- 2.1.3 Definition Software Engineering . . . 13
- 2.1.4 Ziele von Software Engineering als globale Optimierung . . . 13
- 2.1.5 Software Engineering als defensive Disziplin . . . 13
- 2.1.6 Shu-Ha-Ri . . . 13

2.2 Muster . . . 13
- 2.2.1 Muster . . . 13
- 2.2.2 Muster im Software Engineering . . . 14
- 2.2.3 **MVC** Model-View Controller . . . 14
- 2.2.4 Artefakte . . . 14
- 2.2.5 Konfigurationsmanagement . . . 14
- 2.2.6 Änderungsmanagement . . . 14

2.3 Git . . . 14
- 2.3.1 Tags . . . 14

2.4 Verzweigungsmuster . . . 14
- 2.4.1 Integration . . . 14
- 2.4.2 Auslieferung . . . 15

3 Tests **17**
3.0.1 Definition . . . 17
3.0.2 Vorteile . . . 17
3.0.3 Nachteile . . . 17
3.0.4 Formen der Ausführung . . . 17
3.0.5 Treiber für Spezifikation von Tests (nach Glass (2009)) . . . 17
3.0.6 Test-Automatisierung . . . 18
3.0.7 Äquivalenzklassen . . . 18
3.0.8 Glassbox Tests . . . 18
3.1 Teststufen . . . 18
3.1.1 Unit- Modultest . . . 18
3.1.2 Testcase per feature: . . . 19
3.1.3 Four phase test . . . 19
3.1.4 Assertion messages . . . 19
3.1.5 State verification . . . 19
3.1.6 Behaviour verification . . . 19
3.1.7 Test double . . . 19
3.1.8 Akzeptanztests . . . 19
3.1.9 Akzeptanztests getriebene Entwicklung . . . 19
3.1.10 Systemtests . . . 20
3.1.11 Usability Test: . . . 20
3.1.12 Sicherheitstest . . . 20
3.1.13 Testcase Class per Test: . . . 20
3.1.14 Integrationstest . . . 20
3.1.15 Definition: . . . 20
3.2 Muster . . . 21
3.2.1 Common Includes . . . 21
3.2.2 Parameterized includes . . . 21
3.2.3 Antimuster: Creating scenarios by domain experts or developers in isolation . . . 21
3.2.4 **Schwächen von test coverage** . . . 21

4 Integration **23**
4.0.1 Definition: . . . 23
4.0.2 Teilintegriertes System . . . 23
4.0.3 Probleme der Integration . . . 23
4.0.4 Big Bang Integration . . . 23
4.0.5 Bauen . . . 24
4.0.6 Continuous Integration CI . . . 24
4.0.7 Definition . . . 24
4.0.8 CI Pipeline . . . 25
4.0.9 CI config . . . 25
4.0.10 Build Script . . . 25
4.0.11 Risiken, die durch CI reduziert werden . . . 25
4.1 Deployment . . . 25
4.1.1 Definition . . . 25
4.1.2 Release . . . 25
4.1.3 Häufige Releases . . . 25
4.2 Continuous Delivery CD . . . 26
4.2.1 Nutzen von CD . . . 26
4.2.2 Continuous Deployment . . . 26
4.2.3 Continuous Deployment patterns . . . 27
4.2.4 Blue-Green Deployment . . . 27
4.2.5 Canary Releasing . . . 27
4.2.6 Dark launching . . . 27
4.2.7 A/B Testing . . . 27
4.2.8 Feature Toggles . . . 27

5 Qualität **29**
5.0.1 Gravins Qualitätsansätze . . . 29
5.0.2 Stakeholders . . . 29
5.0.3 Qualitätenbaum . . . 29
5.0.4 Qualitätsmodell nach Avizienies . . . 30
5.0.5 ISO 25010 . . . 30
5.0.6 Zuverlässigkeit . . . 30
5.0.7 Fehlerterminologie . . . 30
5.1 Test-driven Development . . . 31

6 Code verstehen(wtf) **33**
6.1 Inhärente/zufällige Komplexität . . . 33
6.1.1 Mentales Modell . . . 33
6.1.2 **McCabes zyklomatische Komplexität** . . . 33
6.1.3 Gewünschte Eigenschaften von Bezeichnern . . . 33
6.1.4 Programmierrichtlinien . . . 33
6.1.5 Refactoring . . . 34
6.2 Qualitätssicherung/ Quality assurance . . . 34
6.2.1 Validierung/ Verifikation . . . 35
6.2.2 Reviews . . . 35
6.2.3 Rollen . . . 35
6.2.4 Fragenkatalog . . . 35

7 Projektmanagement **37**
7.0.1 Ziel . . . 37
7.0.2 Kickoffs . . . 37
7.0.3 Managementaufgaben . . . 37
7.0.4 Planung . . . 38
7.0.5 Planungsaspekte . . . 38
7.0.6 Arbeitspakete . . . 38
7.0.7 Kriterien für Arbeitspakete . . . 38
7.0.8 Interne und externe Meilensteine . . . 38
7.1 Vorgehensmodelle . . . 39
7.1.1 Wasserfallmodell . . . 39
7.1.2 Prototyping . . . 39
7.1.3 Iterative Entwicklung . . . 39
7.1.4 Inkrementelle Entwicklung . . . 39
7.1.5 Scrum . . . 40
7.1.6 Scrum Charakteristika . . . 40
7.1.7 Risikomanagement . . . 41
7.1.8 Risk Items and their solutions . . . 41
7.1.9 **Arten von Teams** . . . 41
7.1.10 **Funktionale Organisation** . . . 42
7.1.11 Aufwandsschätzungen . . . 42

8 Modellierung **45**
8.0.1 Komplexität . . . 45
8.0.2 Präskriptive und deskriptive . . . 45
8.0.3 Sichten(Views) und Perspektive(Viewpoints) . . . 45
8.0.4 UML . . . 45
8.0.5 **Komposition vs Aggregation** . . . 45
8.1 Diagramme . . . 45

9 Architektur **47**
9.0.1 Grob und Feinentwurf . . . 47
9.0.2 Aspekte der Architektur . . . 47
9.0.3 Systemsicht und statische Sicht: . . . 47
9.0.4 Verteilungsdiagramm . . . 47
9.0.5 Modularisierung . . . 48
9.0.6 Ziele . . . 48

9.0.7 Separation of concerns . . . 48
9.0.8 Architekturmuster . . . 48
9.0.9 Architecutal decision AD . . . 49
9.0.10 Entwurfsprinzipien **SOLID** . . . 49
9.0.11 Law of Demeter . . . 49
9.1 Entwurfsmuster . . . 49
9.1.1 Singelton . . . 49
9.1.2 Composite . . . 49
9.2 Domain driven design . . . 50
9.2.1 Strategischer und taktischer Entwurf . . . 50
9.2.2 **Bounded context** . . . 50
9.2.3 Subdomains . . . 50
9.2.4 Context Mapping . . . 51
9.2.5 Ubiquitous language . . . 51
9.2.6 Entity . . . 51
9.2.7 DDD aggregate . . . 51
9.2.8 Aggregate . . . 51
9.2.9 Value Object . . . 51
9.2.10 Domain Event . . . 51
9.3 Wartung . . . 51
9.3.1 Risiken und Probleme . . . 51
9.3.2 Wartungskrise 1980 . . . 51
9.3.3 Legacy Software 1990 . . . 51
9.3.4 Langelebige Systeme 2000 . . . 52
9.3.5 Continous engineering . . . 52
9.3.6 Klassen von Wartungsfällen . . . 52
9.3.7 Lehmans Gesetze der Software Evolution . . . 52
9.3.8 Auswirkungsanalyse . . . 52
9.3.9 Nachverfolgbarkeit/ Traceability . . . 52
9.3.10 Ablösung . . . 52
9.3.11 Reengineering . . . 53
9.4 Dokumentation . . . 53
9.4.1 Arten von Dokumentation . . . 53

Chapter 1

Anforderungsanalysen und Spezifikation

1.0.1 Definition

Ein Statement, was die Bedingungen, Grenzen, Randbedingungen und Erwartungen vom Softwaresystem beschreibt.

1.0.2 Funktionalität

Funktion der Software = Beziehung zwischen Ein- und Ausgabe, im weiten Sinne auch das Zeitverhalten. Alle Aufforderungen, die sich auf Funktion beziehen → funktionale Anforderungen (NFRs). Abgrenzung aber unscharf z.B. bei Robustheit, Zeitverhalten, Bedienbarkeit oft als nichtfunktional klassifiziert. Alle Aussagen zur Wartbarkeit → nicht funktional.

1.0.3 Anwendungsdomänen

Weil Software in allen Bereichen und verschiedenen Domänen benötigt wird. Anforderungsanalyse hängt am stärksten von der Domäne ab, aber vielleicht auch von Größe des Unternehmens, der Entwickler und Kunden. Bei manchen Produkten nur grobe textuelle Anforderungen festhalten, während z.B. bei einem Banksystem auch Gesetze auf bürokratischem Wege beschrieben werden müssen.

1.0.4 Klassifikation der Anforderungen

- Project Requirement
- System Requirement
- Process Requirement
- Functional Requirement
- Attribute
- Constraint
- Performance Requirement
- Quality Requirement

1.1 Spezifikation

1.1.1 Definition

Zentrales Artefakt der Software. Dokumentation der essenziellen Anforderungen und der Benutzeroberfläche. Strukturierte Kollektion der Anforderungen (Performance, Design, Attribute, Randbedingungen).

1.1.2 Komponenten

Vor der Erteilung des Auftrags muss festgelegt werden: **was** das System leisten soll, **welche** Qualitätseigenschaften gewollt sind und **welche** Kosten bei der Entwicklung entstehen. Analysieren besteht aus Klassifizieren und Organisieren, Priorisieren und Verhandeln und Dokumentieren.

- Kunde/Marketing
- Test
- Arbeitsplanung
- Klärung von Einwänden/Ansprüchen
- Benutzungshandbuch
- Re-implementierung
- Entwurf und Implementierung

1.1.3 Lasten und Pflichtenheft

Lastenheft: problemorientiert und abstrakt (vom Auftraggeber) Pflichtenheft: konkret, präzise, konsistent, abgestimmt, vollständig, messbar (vom Auftragnehmer)

1.1.4 Ziel

Ist- und Soll-Analyse festhalten.

1.1.5 Prozess

Besteht aus Sammeln, Analysieren, Spezifizieren und Validieren.

1.1.6 Anforderungsquellen

Domänenmodell, Bedürfnisse der Stakeholder, Aktuelle Organisation und Systeme, existierende Dokumente, Anforderungsbibliothek (wiederverwendbare Anforderungen), Anforderungsvorlage (vorgeschlagene Typen von Anforderungen).

1.1.7 Befragung/ Story-writing workshop

Sollte gut geplant sein und mit kontextfreien Fragen wie "Wer sind die Nutzer" einleiten. Kunden, Nutzer und Entwickler setzen sich zusammen und schreiben User Stories.

1.1.8 Personas

Beschreiben vollkommen fiktive, aber realistische Personen/Nutzer:innen. Genutzt, um bei der Anforderungsanalyse leichter die Perspektive der Nutzer:innen zu bekommen und die passenden Anforderungen zu identifizieren. Es werden tatsächlich sehr detaillierte Vorstellungen der Personen entworfen.

1.1.9 Spezifikationsmethoden

- Zustandsmodell
- Ablaufmodell
- formales Modell
- Strukturmodell
- Datenmodell
- Begriffsmodell
- Anwendungsfälle

- Strukturierter Text
- textuell
- Verteilungsmodell

1.1.10 Wichtige Merkmale der Anforderungsspezifikation

1. adäquat: beschreibt das, was der Kunde will, braucht.
2. vollständig: beschreibt alles, was der Kunde will, braucht.
3. widerspruchsfrei
4. verständlich für alle Beteiligten.
5. eindeutig.
6. prüfbar.
7. risikogerecht: Umfang umgekehrt proportional zum Risiko, das man eingehen will.

1.1.11 INVEST

- **I**ndependent (um Auswahl der Abarbeitung zu erleichtern: ist es in sich abgeschlossen oder hängt die Erfüllung auch von anderen Komponenten ab bzw. ist verbunden?)
- **N**egotiable (um nicht zu viel vorwegzunehmen, damit man im Gespräch mit Stakeholdern noch verfeinern kann: beschreiben sie, was umgesetzt werden soll und **nicht wie**, einem wird nichts vorgeschrieben)
- **V**aluable (gibt es einen Mehrwert? Gegenbeispiel: unnötige Design Patterns vorgeschrieben, die z.B. gar nicht in dem Kontext passen)
- **E**stimatable (fürs Team und Aufwand den Aufwand einschätzen können, klarer Umfang mit wenig unbekannten Einflüssen)
- **S**ized Appropriately (gut beschrieben, aber braucht trotzdem ein Jahr wäre eine schlechte Anforderung, weil braucht ja dennoch so lange), im Sprint durchsetzbar?
- **T**estable (gute Akzeptanzkriterien definieren, man kann es testen, weil klar auch beschrieben wurde, wann sie erfolgreich umgesetzt wurde), Soll und Ist- Zustand vergleichbar?

1.1.12 Begriffslexikon

Wird bei der Analyse angelegt, enthält die Definitionen.

1.2 Use cases

1.2.1 Definition

Dt: Anwendungsfall.

- neben dem System ist immer mindestens ein Akteur:in beteiligt.
- Anstoß durch einen **Trigger/spezielles Ereignis**, den die Hauptakteur:in auslöst.
- ist **zielorientiert**
- beschreibt alle Interaktionen zwischen dem System und den beteiligten Akteur:innen.
- endet, wenn das angestrebte Ziel erreicht ist oder klar ist, dass es nicht mehr erreicht werden kann.

Beschreiben also **Außensicht**, was das System leisten soll und welche Interaktionen dazu notwendig sind. Benötigt immer:

- Name

- Ziel
- Vorbedingung
- Nachbedingung
- Nachbedingung im Sonderfall
- Akteure
- Normalablauf

1.2.2 Anwendungsfalldiagramm

Unterscheidung zwischen **Hauptfunktionen** und **Basisfunktionen**. Hauptfunktionen: beschreiben die geforderte fachliche Funktionalität des Systems. Basisfunktionen werden in Hauptfunktionen verwendet und liefern dort Beitrag zur Funktionalität. Anwendungsfälle sind Pakete. **Vorteile:**

- zeigt, welche Akteure in welchen Anwendungsfällen mit dem System interagieren
- sehr übersichtlich bei kleinen Modellen
- gute Grundlage für Testfälle, Prototypen, Benutzungsdokumentation.

Nachteile:

- modelliert Zusammenhänge zwischen Anwendungsfällen nur rudimentär.
- modelliert eigentliche Inhalte der Anwendungsfälle nicht.

1.2.3 Nachteile bei gezwungener gleicher Granularität am Anfang

- geringstes Wissen am Anfang.
- Aufwand für nicht umgesetzte Anforderungen.
- große Änderungskosten
- weniger Gespräche, da Anforderungen bereits "fertig".

1.2.4 Definition of done

Design reviewed. Code completed (refactored, in standard format, inspected). User documentation updated. Tested. Zero known defects. Acceptance tested. Live on production servers.

1.2.5 Außerdem wichtig für den Entwurf abgesehen von den Anforderungn (HIER falls du es nicht findest :))

- Nutzerrollen
- Zugriffskonzept
- Art der Datenspeicherung
- System und Softwarearchitektur
- Design-Prinzipien (SOLID, DRY, KISS, YAGNI)
- Datenmodell
- UI und UX
- Sicherheit
- Performance
- Testbarkeit
- Wartbarkeit
- Aufteilung der Arbeitspakete

Name	Authentifizieren
Ziel	Der Kunde möchte Zugang zu einem Bankautomaten BA42 erhalten
Vorbedingung	– Der Automat ist in Betrieb, die Willkommen-Botschaft wird angezeigt – Karte und PIN des Kunden sind verfügbar
Nachbedingung	– Der Kunde wurde akzeptiert – Die Leistungen des Bankautomaten stehen dem Kunden zur Verfügung
Nachbedingung im Sonderfall	Der Zugang wird verweigert, die Karte wird entweder zurückgegeben oder einbehalten, die Willkommen-Botschaft wird angezeigt
Akteure	Kunde (Hauptakteur), Banksystem

Figure 1.1: Use case table

Normalablauf	1. Der Kunde führt eine Karte ein 2. Der Bankautomat liest d. Karte und sendet d. Daten z. Prüfung ans Banksystem 3. Das Banksystem prüft, ob die Karte gültig ist 4. Der Bankautomat zeigt die Aufforderung zur PIN-Eingabe 5. Der Kunde gibt die PIN ein 6. Der Bankautomat liest die PIN und sendet sie zur Prüfung an das Banksystem 7. Das Banksystem prüft die PIN 8. Der Bankautomat akzeptiert den Kunden und zeigt das Hauptmenü

Figure 1.2: Normalablauf

Sonderfall 2a	*Die Karte kann nicht gelesen werden* 2a.1 Der BA42 zeigt die Meldung »Karte nicht lesbar« (4 s) 2a.2 Der BA42 gibt die Karte zurück 2a.3 Der BA42 zeigt die Willkommen-Botschaft
Sonderfall 2b	*Die Karte ist lesbar, aber keine BA42-Karte* 2b.1 Der BA42 zeigt die Meldung »Karte nicht akzeptiert« (4 s) 2b.2 Der BA42 gibt die Karte zurück 2b.3 Der BA42 zeigt die Willkommen-Botschaft
Sonderfall 2c	*Das Banksystem ist nicht erreichbar* 2c.1 Der BA42 zeigt die Meldung »Banksystem nicht erreichbar« (4 s) 2c.2 Der BA42 gibt die Karte zurück 2c.3 Der BA42 zeigt die Willkommen-Botschaft
Sonderfall 3a	*Die Karte ist nicht gültig oder gesperrt* 3a.1 Der BA42 zeigt die Meldung »Karte ungültig oder gesperrt« (4 s) 3a.2 Der BA42 zeigt die Meldung »Karte wird eingezogen« (5 s) 3a.3 Der BA42 behält die Karte ein 3a.4 Der BA42 zeigt die Willkommen-Botschaft

Figure 1.3: Sonderfälle

Chapter 2

Grundbegriffe

2.1 Geschichte des Softwareengineering

2.1.1 Softwarekrise

Ca. Ende 60er Jahre: Wandel, da Kosten von Software die bis dahin Kosten von Hardware übernommen hat. Zuvor Software nur zur Behebung von Hardwareausfällen. Software wurde also komplexer, es fehlten Methoden, Kommunikationsprobleme, mangelnde Qualitätssicherung. **Folgen:** Software Schwachpunkt des Gesamtsystems. Vertrauensverlust, steigende Kosten, Innovationsschub. **Lösung:** Modularisierung, bessere Qualitätssicherung, agile Methoden und strukturierte Vorgehensweise.

2.1.2 Beiträge der Frühzeit

- "Entdeckung" des Software Life Cycles (1970)
- Überlegung zur Strukturierung der Programme (Parnas, 1972)
- Sammlung empirischer Daten (Boehm et al, 1973) zum besseren Verständnis der Kosten-Ursachen und der Kosten-Verteilung.

2.1.3 Definition Software Engineering

Die Entdeckung und Anwendung solider Ingenieur-Prinzipien mit dem Ziel, auf wirtschaftliche Art Software zu bekommen, die zuverlässig ist und auf realen Rechnern läuft. Jede Aktivität, bei der es um die Erstellung oder Veränderung von Software geht, soweit mit der Software selbst hinausgehend.

2.1.4 Ziele von Software Engineering als globale Optimierung

- Gesamtkosten senken (z.B. Spezifikationen oder Architekturentwurf weglassen, aber dann bei Wartung deswegen große Probleme, also nein) → globales Optimum suchen
- Effizienteres Bild der Kosten und ihrer Beziehungen entwickeln, um dem Optimum zu nähern.

2.1.5 Software Engineering als defensive Disziplin

Dient nur der Verhinderung von Schäden und Abwehr von Schäden.

2.1.6 Shu-Ha-Ri

Shu (gehorchen) zu ha (abweichen) zu ri (separieren). Anfangs sollte man also die Techniken genauso nachahmen, wie es die Meister vorgeben. Dann leicht abweichen und erst wenn man selbst Meister:in ist, sollte man sich ganz von den Vorgaben lösen.

2.2 Muster

2.2.1 Muster

Etwas mit Wiedererkennungswert, das eine bestimmte Art zu Denken, Verhalten oder Gestaltebene beschreibt.

2.2.2 Muster im Software Engineering

- Entwurfsmuster
- Antimuster
- Prozessmuster
- Analysemuster
- Testmuster
- Architekturmuster
- Muster im Softwarekonfiugurationsmanagement

2.2.3 MVC Model-View Controller

Muster bei dem Daten (*Model*), Darstellung/UI(*View*), und Steuerung/Brain (*Controller*) getrennt werden.

2.2.4 Artefakte

Definition: Zwischen und Endprodukte der Software.

2.2.5 Konfigurationsmanagement

Definition: die Gesamtheit der Verfahren, um die Konfiguration eines Software Systems zu identifizieren, zu verwalten, bei Bedarf bereitzustellen und ihre zu Änderungen überwachen und zu dokumentieren. Dazu gehört auch die Möglichkeit, ältere Artefakte und Konfigurationen zu rekonstruieren. Das Konfigurationsmanagement schließt also die Versions- und Variantenverwaltung ein.

2.2.6 Änderungsmanagement

auch Issue Tracking. Änderungsforderungen werden verfolgt, die aus verschiedenen Quellen an ein Softwareprojekt gestellt werden. z.B. Fehlerfeldungen von Anwender:innen oder auch Wünsche für Features/Funktionalität. Wichtig, dass es einen festgelegten Prozess, wie umgegangen wird gibt.

2.3 Git

2.3.1 Tags

Einfache Tags: lightweight, ist wie ein Branch, der sich niemals ändert. Nur ein Zeiger auf einen bestimmten Commit.
Kommentierte Tags: vollwertige Objekt ein der Git Datenbank.Sie haben eine Checksumme, beinhalten Namen und E-Mail Adresse desjenigen, der den Tag angelegt hat, das jeweilige Datum sowie eine Meldung. Sie können überdies mit GNU Privacy Guard (GPG) signiert und verifiziert werden. Generell empfiehlt sich deshalb, kommentierte Tags anzulegen. Wenn man aber aus irgendeinem Grund einen temporären Tag anlegen will, für den all diese zusätzlichen Informationen nicht nötig sind, dann kann man auf einfache Tags zurückgreifen. So ein kommentierter Tag stellt dann eine Konfiguration dar

2.4 Verzweigungsmuster

2.4.1 Integration

- **Mainline integration:** Alle gesunden Branches in die Mainline.
  Vorteil: man kann so einfach kontinuierliche Integration auf der Mainline machen.
- **Feature Branch:** Features in einzelnen Branches gemacht und erst in Mainline gepusht wenn fertig.
  Nachteil: bei langlebigen Feature Branches ist kontinuierliche Integration erschwert.

- **Continous integration:** häufiges integrieren in die Mainline.
  Vorteil: Merge conflicts werden risikoärmer und weniger Aufwand, aber
  Nachteil: dass man mehr Merge conflicts zu lösen hat durch CI.
- **Peer reviewed commit:** bei jedem Commit am besten.

2.4.2 Auslieferung

- **Release branch:** Erstellung einer separaten Release Branch in den keine neuen Features mehr kommen, sondern nur Commits zur Stabilisierung (zB Fehlerbehebung). Man kann auch mehrere Release Branches parallel haben, falls Kund:innen noch alte Software Veionen am Laufen haben. *Nachteil:* kann relativ schwer sein, die Änderung im Release Branch an wieder in Mainline zu bekommen
- **Maturity branch:** man erstellt mehrere Ebenen von Branches je nach Reife. zB Mainline, Release Branch, producition branch.

Chapter 3

Tests

3.0.1 Definition

die Ausführung eines Programms auf dem Rechner unter Bedingungen, für die das korrekte Ergebnis bekannt ist, so dass Ist und Soll Resultat verglichen werden können. Bei Nichtübereinstimmung liegt ein Fehler vor. Ist **gut**, wenn er hohe Chance hat, Fehler anzuzeigen.
Zweck: die Entdeckung von Fehlern. Testen ist *destruktiv*.

3.0.2 Vorteile

- natürliches Prüfverfahren
- ist **reproduzierbar** und damit **objektiv**
- einmal gut organisiert, lässt es sich **billig** wiederholen
- Zielumgebung (Übersetzer, BS usw.) wird mitgeprüft.
- Systemverhalten wird sichtbar gemacht.

3.0.3 Nachteile

- Aussagekraft des Tests wird **überschätzt**. zeigt nicht Korrektheit, denn schon die Zustandsräume kleiner Programme sind riesig.
- man kann nicht alle Anwendungssituationen nachbilden.
- zeigt nicht die **Fehlerursache**.

3.0.4 Formen der Ausführung

Automatisierte Tests werden häufig ausgeführt und bringen Sicherheit für Entwickler:innen und helfen Regressionsfehler zu finden.
Explorative Tests manuelles Arbeiten, wo man kreativ neue Szenarien überlegt. Vor allem bei neuen Features wichtig.

3.0.5 Treiber für Spezifikation von Tests (nach Glass (2009))

1. durch **Anforderungen:** wir verwenden Anforderungsspezifikation um Testfälle abzuleiten. Ziel alle Anforderungen abzudecken. **Blackbox Tests**
2. durch **Struktur:** man verwendet die interne Struktur des Quelltexts mit dem Ziel verschiedener Aspekte dieser Struktur abzudecken **Whitebox tests**
3. durch **Statistik:** die Eingaben, die in Testfällen verwendet werden sind durch statische Methoden verwendet worden. Folgen beispielsweise einer statischen Verteilung oder sind zufällig.
4. durch **Risiko:** verschiedene Risiken, z.B. finanzielle Risiken von Fehlern oder hohe Verwendung bestimmter Funktionalität.

3.0.6 Test-Automatisierung

Vorteile

- verbesserte Produktqualität
- hohe Überdeckung
- verkürzte Testzeit
- Zuverlässigkeit
- erhöhtes Vertrauen
- Wiederverwendbarkeit der Tests
- weniger menschlicher Aufwand
- Kosteneinsparung
- erhöhte Fehlerfindungsrate

Nachteile:

- kann manuelles Testen nicht ersetzen
- Wartung von automatisierten Testfällen
- Einführung des Prozesses braucht Zeit
- fehlende Mitarbeiter mit entsprechenden Fähigkeiten

3.0.7 Äquivalenzklassen

Menge der Zustände in denen das Programm gleich auf den Sachverhalt reagiert. Aus jeder Klasse muss mindestens ein Wert getestet werden. Werden intuitiv identifiziert (heuristisch).
Lieber eine Klasse deswegen zu viel als zu wenig.
Grenzwerte sollten speziell berücksichtigt werden.

3.0.8 Glassbox Tests

Da liegt der Ablaufparagraph also dein Flussdiagramm zu Grunde. Arten von Überdeckungen:

- **Befehlsüberdeckung:** Alle Befehle müssen ausgeführt werden.
- **Zweigüberdeckung:** alle Verzweigungen werden abgedeckt.
- **Termüberdeckung:** alle möglichen Ursachen für Verzweigungen müssen wirksam geworden sein.
- **Pfadüberdeckung:** alle Pfade müssen durchlaufen worden sein.

Werkzeuge: Der Glass-Box-Test setzt **Test-Werkzeuge** voraus, die instrumentieren die Quellprogramme (d.h. sie fügen zusätzliche Anweisungen ein), so dass während der Ausführung statistische Daten gesammelt werden.
Außerdem kumulieren die Werkzeuge diese Daten, so dass eine **Aussage** entsteht, wie oft ein bestimmter Befehl, Zweig o.ä. insgesamt, also in allen Testläufen, durchlaufen wurde. Schließlich präsentiert das Werkzeug die Resultate in leicht lesbarer Form, z.B. durch unterschiedlich eingefärbten Quellcode, der unmittelbar die noch nicht erreichten Teile erkennen lässt.

3.1 Teststufen

3.1.1 Unit- Modultest

Einzelne Einheiten (Klassen, Module) werden separat/isoliert getestet.
Wechselwirkungen mit anderen Units sind **nicht** von Interesse.

3.1.2 Testcase per feature:

Falls die Zahl der Methoden in der Testklasse zu stark ansteigt: **separate Testfälle** pro Feature.
Dabei aber Schwierigkeit, sinnvolle Features zu identifizieren.
Andere Alternativen: Testcase class per fixture, weil dies Code Duplikate vermeidet

3.1.3 Four phase test

Bei automatisierten Tests kann man über vier Teile unterscheiden:

- **Vorbereitungsteil/Setup:** baut Umgebung und die Daten für den Testfall auf
- **Fixture:** verbindet SUT mit anderen davon abhängigen Klassen und ersetzt diese evtl. auch durch Dummies.
- **Ausführungsteil/Exercise:** ruft die zu testenden Funktionen im SUT auf
- **Verifikationsteil/Verify:** führt den Vergleich zwischen Resultat und zu erwartendem Resultat aus.
- **Teardown/Abschlussteil:** räumt die aufgebauten Ressourcen wieder auf.

3.1.4 Assertion messages

Zusicherungen, um unsere Testlogik zu strukturieren.

3.1.5 State verification

Wenn SUT keinen Wert zurückliefert muss man Zustandsänderung z.B. über Abprüfen einzelner Attribute testen.

3.1.6 Behaviour verification

Es zeigt sich ein sichtbares Verhalten, was man mit dem zu erwarteten Verhalten vergleichen kann. z.B. durch Protokollierungen von Veränderungen

3.1.7 Test double

Schwierigkeit bei Unit Tests: SUT hängt von anderen Klassen/Komponenten ab -¿ test double ersetzt die Klassen, API sollte übereinstimmen, aber nicht ganze Funktionalität sollte repliziert werden. Soll Test **isolieren**. Dazu gehören:

- **Dummy object:** Manchmal werden Werte im Test benötigt, damit SUT etwas aufrufen kann. Kann Test kompakter machen anstatt jedes einzelne Attribut separat innerhalb des Testfalls zu setzen.
- **Fake object:** eng mit mock objekt verwandt, Fokus hier auf sehr leichtgewichtiger Implementierung. Ersetzt den Zugriff auf ein anderes Objekt, auf die nicht zugegriffen werden sollte. Man versucht unerwünschte Seiteneffekte zu vermeiden. z.B. fake Databases sind beliebt.
- **Test spy:** Gegenstück zum Test stub: fängt die Ausgabe des SUT an andere Komponenten ab und gibt sie dem Testfall zur Verifikation über.
- **Mock object:** Objekt, dass sich auf SUT verlässt wird mit testspezifischem Objekt ersetzt um zu verifizieren, dass es vom SUT korrekt verwendet wird. Ist insbesondere bei **state verification** wichtig.
- **Test stub:** liefert bestimmte Werte an das SUT. Kann immer gleicher Wert (**hard coded test stub**) oder ein konfigurierbarer Wert sein. Aber besonders wichtig, dass der Test stub installierbar ist.

3.1.8 Akzeptanztests

3.1.9 Akzeptanztests getriebene Entwicklung

Akzeptanztests auf Systemebene als Grundlage für ganze weitere Entwicklungsaktivitäten. Aus Anforderungen meisten Use case Diagramme werden dann auch Akzeptanzkriterien abgeleitet.

3.1.10 Systemtests

Definition: Test auf **kompletten, integrierten** System.

Lasttests: testen gesamte Prozessketten auf **Performanz**, d.h. die Verknüpfung der Einzelprozesse. D.h. konkrete Prozesse Vorgänge aus der Nutzung werden **simuliert**.

Skalierbarkeit ist von entscheidender Bedeutung. Häufige Fehlerwirkung sind **Deadlocks** beim Datenbankzugriff, die sonst nur schwer testbar sind.
Wird das System **bewusst über die definierte Lastgrenze hinaus** beansprucht, spricht man vom **Stresstest.**

Dabei sollte die Last (Anzahl der virtuellen Nutzer:innen) **schrittweise** bis über die definierte Lastgrenze hinaus erhöht werden.

Performanztests: wiederholen ausgewählte Testfälle bzw. Einzelprozesse aus dem Systemtest unter einer **Grundlast:** einzelne Funktionen auf ihre Performanzeigenschaften geprüft zum Testen der Skalierbarkeit einzelner Funktion.

3.1.11 Usability Test:

Test mit Nutzer:innen zur Benutzbarkeit. Konzentration auf dem **Qualitätsattribut** Benutzbarkeit, denn viele Aspekte lassen sich erst sinnvoll auf Systemebene prüfen.
Es gibt systematische Evaluierungen mit potenziellen echten Nutzer:innen. Messmethoden z.B. Videos, eye tracking wodurch dann **Rückschlüsse auf Benutzbarkeit** gezogen werden.

3.1.12 Sicherheitstest

Zur Informationssicherheit, sollten nicht erst auf Systemebene ansetzen, auch bereits auf kleinerer Ebene testbar, aber sinnvoll nochmal auf Systemebene nach Lücken zu suchen. Hat die Funktion des Nachweises, dass eine Software keine Funktion enthält, die sie nicht enthalten soll meistens Negativtests. Sollen Beweis erbringen, dass keine unsicheren Nebeneffekte vorhanden sind.
Üblicher Ansatz: Penetrationstests (White Hat Hacker)

3.1.13 Testcase Class per Test:

Einfach für jede Klasse eine **eigene** Testklasse um am einfachsten die jeweiligen Testmethoden die zur zugehörigen Testklasse zuzuordnen.

3.1.14 Integrationstest

3.1.15 Definition:

Es geht nicht um Fehler einzelner Komponenten, sondern um **Konsistenzprobleme** zwischen den Komponenten. z.B. Schnittstellenfehler zwischen den Modulen.

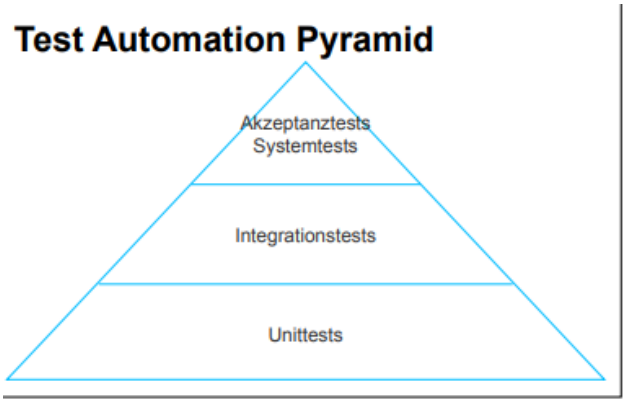


Figure 3.1: Test automation pyramid

3.2 Muster

3.2.1 Common Includes

Das Zusammennehmen von Gemeinsamkeiten die dann so in den Test eingebunden werden um unnötige Mehrarbeit zu sparen, wenn sie in der ähnlichen Umgebung aufgebaut wurden.

3.2.2 Parameterized includes

In Tests werden **Parameter** integriert, die mit separat spezifizierten Werten belegt werden können, wenn sie bis auf die Parameter gleich sind (basically **überschreiben**)

3.2.3 Antimuster: Creating scenarios by domain experts or developers in isolation

Wenn Tests nicht gemeinsam besprochen, **kein gemeinsames Verständnis** der Anforderungen der Software, fehlt die **technische Expertise** und die Tests sind **schwer zu automatisieren.**
Wenn Entwickler alleine an Szenarien arbeiten, dann entsprechen sie **nicht** den eigentlichen Nutzerbedürfnissen.
Antimuster: Creating Scenarios by Domain Experts or Developers in Isolation

3.2.4 Schwächen von test coverage

- keine Aussage über Qualität und Sinnhaftigkeit
- kein Testen der Edge cases evtl
- ignoriert nicht ausgeführten Code
- falsches Sicherheitsgefühl
- imagine dein Test ist buggy und du hast dann ein buggy Programm, das nt auffällt

Chapter 4

Integration

4.0.1 Definition:

Prozess der Kombination aller Software- und Hardware-Komponenten in ein eingebettetes System.

4.0.2 Teilintegriertes System

Solange System noch **nicht vollständig integriert** ist, wird das Ergebnis jedes Integrationsschrittes als Teilintegriertes System bezeichnet.

4.0.3 Probleme der Integration

- **inkompatible** Schnittstellen (syntaktische oder semantische Konflikte z.B. durch Fehlen von Spezifikationen

4.0.4 Big Bang Integration

einfach alles in einem Modul testen und reinschmeißen.
Vorteile:

- **keine** Test doubles sind notwendig (weil man ja alles zusammen reinschmeißt)
- **keine** Testtreiber und Platzhalter
- sofort **vollständiges** System

Nachteile:

- System wahrscheinlich **nicht lauffähig**.
- Fehler nur auf **Systemebene**.
- in Praxis kaum möglich, da Komponenten zu viele Fehler und **Inkonsistenzen** enthalten:
  System ist **nicht ausführbar**, Probleme lassen sich nur einem **großen ganzen System** zuordnen.

4.0.5 Bauen

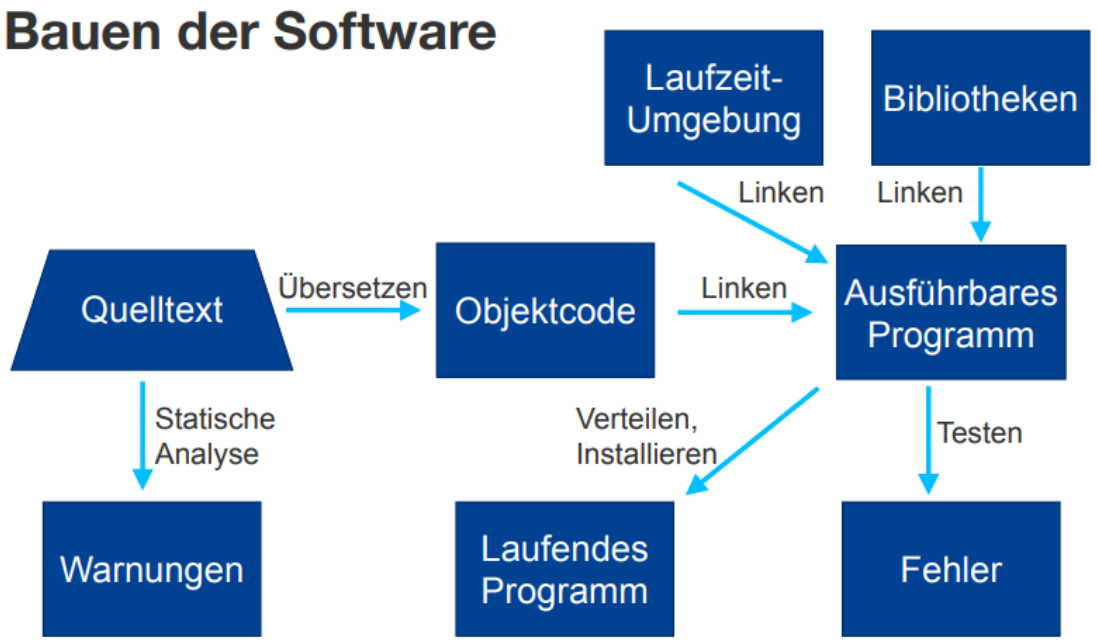


Figure 4.1: Bauen der Software

Automatisierung durch Build scripts: bauen wegen Komplexität der **Zusammenhänge** und **Sicherstellung** der Wiederholbarkeit notwendiger Schritte immer automatisiert.

4.0.6 Continuous Integration CI

4.0.7 Definition

Ist eine laufende Integration auf dem Integrationsrechner der von der Entwicklungsumgebung getrennt ist. Integration als fortlaufender Prozess. Sobald neue Komponente fertig ist wird sie integriert und getestet.

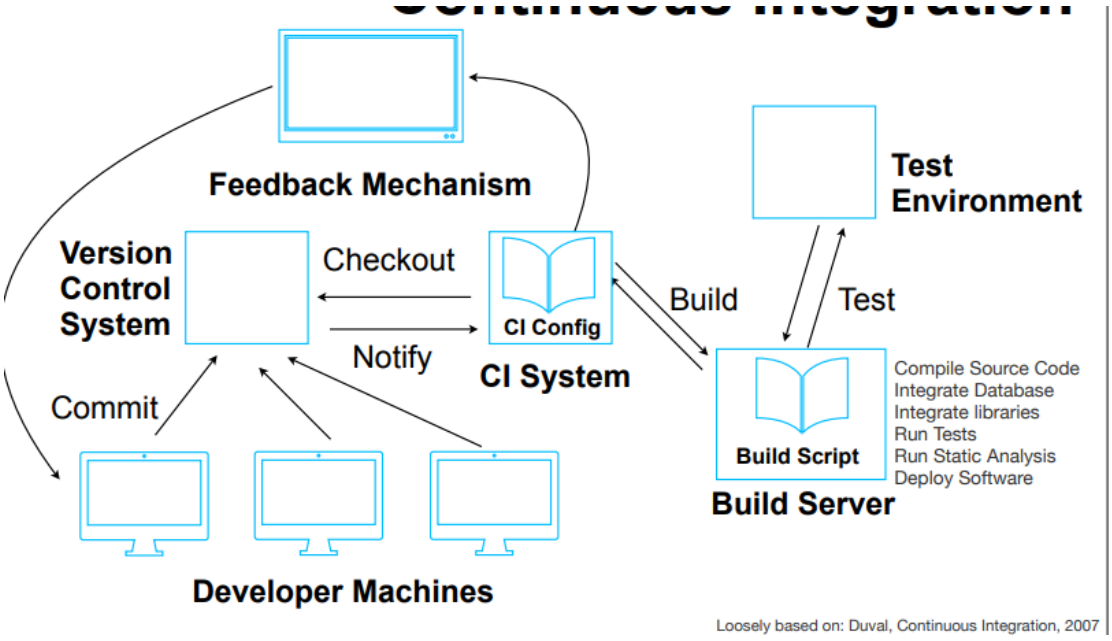


Figure 4.2: CI

4.0.8 CI Pipeline

Man teilt Pipeline in **Stages**. Regel hinter Reihenfolge: "fail fast", also wenn schief geht, **möglichst schnell**, da schnelle Rückmeldung und schnelle Behebung.

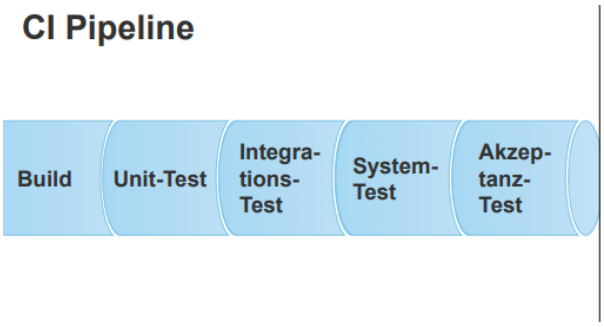


Figure 4.3: CI Pipeline

4.0.9 CI config

Beschreibt, **was** zu tun ist. Z.B. Betriebssystem auf dem build, tests ausgeführt werden, Datenbanken und dazugehörige Passwörter.
Kennt die Ablaufumgebung, ruft entsprechende Teile des Build Scripts auf und unter Versionskontrolle.

4.0.10 Build Script

Kennt hingegen, wie das Quellcode repo und quellcode aufgebaut ist. Es weiß, was übersetzt werden muss. Es weiß, wo Tests sind die ausgeführt werden müssen.
Wenn ich also meine Tests in Unterverzeichnisse für Unit-Tests und Integrationstests aufteile, dann muss sich das **Build Script ändern**.
Die CI Config sollte davon aber **nicht betroffen** sein

4.0.11 Risiken, die durch CI reduziert werden

- **Keine ausliefbare Software:** manuelles Anstoßen, Datenbank Synchronisation
- **Späte Fehlerentdeckung:** Regressionstest, Testüberdeckung
- **Fehlende Sicht auf Projektstatus**
- **Niedrige Softwarequalität**

4.1 Deployment

4.1.1 Definition

Phase im Projekt wo das System **into operation** gebracht wird und von **alten** zu **neuen** Systemversionen aktualisiert werden Fehler resolved werden (Cutovers).

4.1.2 Release

Definition: bestimmte Versionen einer Konfiguration die zu einem bestimmten Grund gemacht wurden *oder* eine **Ansammlung** an neuen oder veränderten Konfigurationen, welche getestet wurden und live in die neue Umgebung zusammen gelassen werden.

4.1.3 Häufige Releases

Sehr viele **kleine** Releases um **Fehleranfälligkeit zu vermeiden** und dann nicht auf einmal das ganze System am Arsch ist. Risiko bleibt klein, **aber** Release sollte um das möglich zu machen weitgehend automatisiert sein.

4.2 Continuous Delivery CD

Erweitert CI indem es fordert, dass **jede Änderung gebaut und getestet,** sondern auch in einer Staging Umgebung durchgeführt werden.

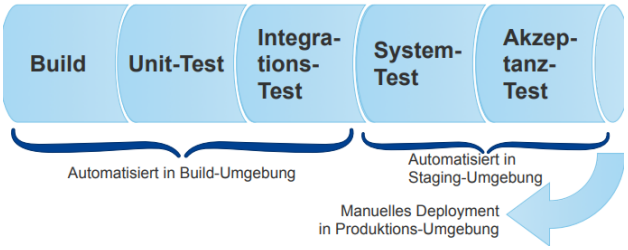


Figure 4.4: CD

4.2.1 Nutzen von CD

- bei jeder Änderung **schnelles, automatisiertes Feedback** zur Produktionsreife der Software.
- Releases hängen nur noch von **Geschäftsanforderungen** ab, nicht von Einschränkungen aus Entwicklung oder Betrieb.
- für Änderungen am Code bekommt man in jeden Fall eine **Rückmeldung**, ob die Software in **Produktionsumgebung** gehen kann.

4.2.2 Continuous Deployment

Fügt CD das direkte, automatische Deployment in die **Produktionsumgebung** hinzu.
Extrem schnelle Rückmeldung zu Änderung und Admit Möglichkeit zu experimentieren.

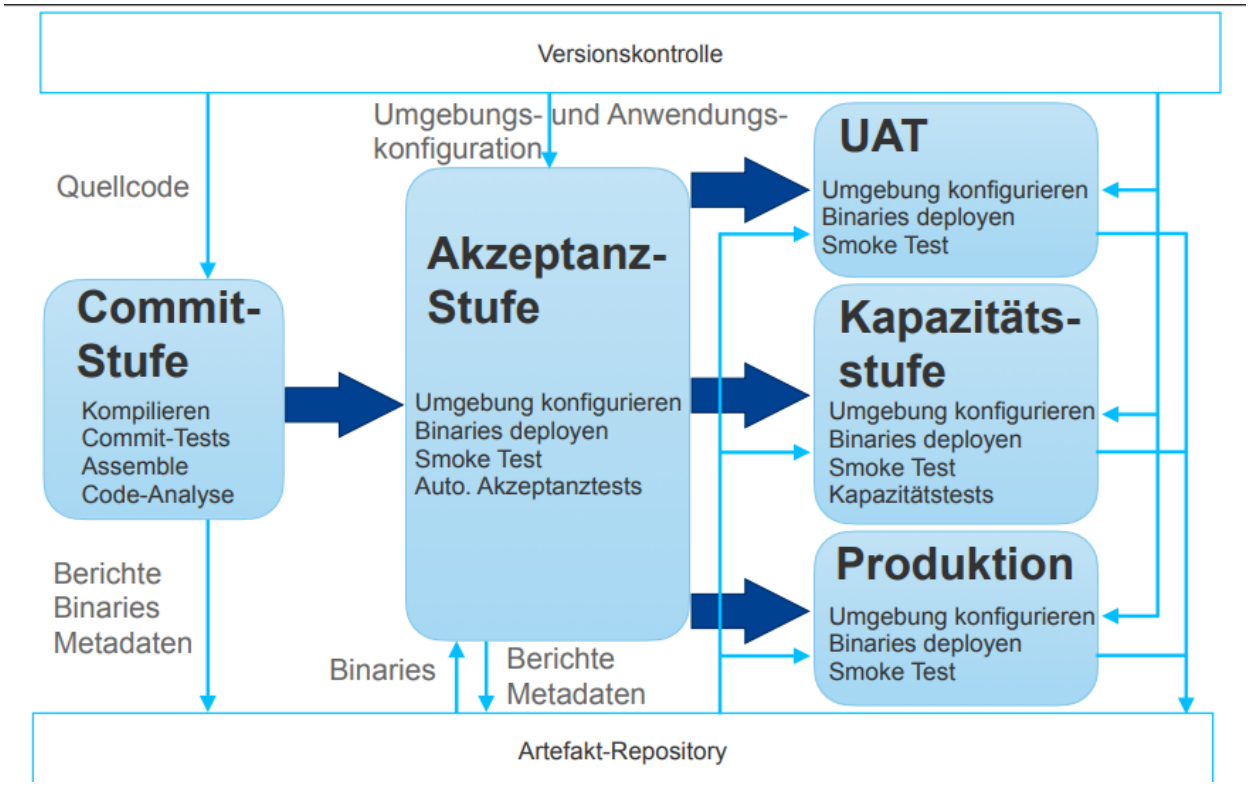


Figure 4.5: ausgereifte Deployment Pipeline mit großen Stufen
UAT= User Acceptance Test

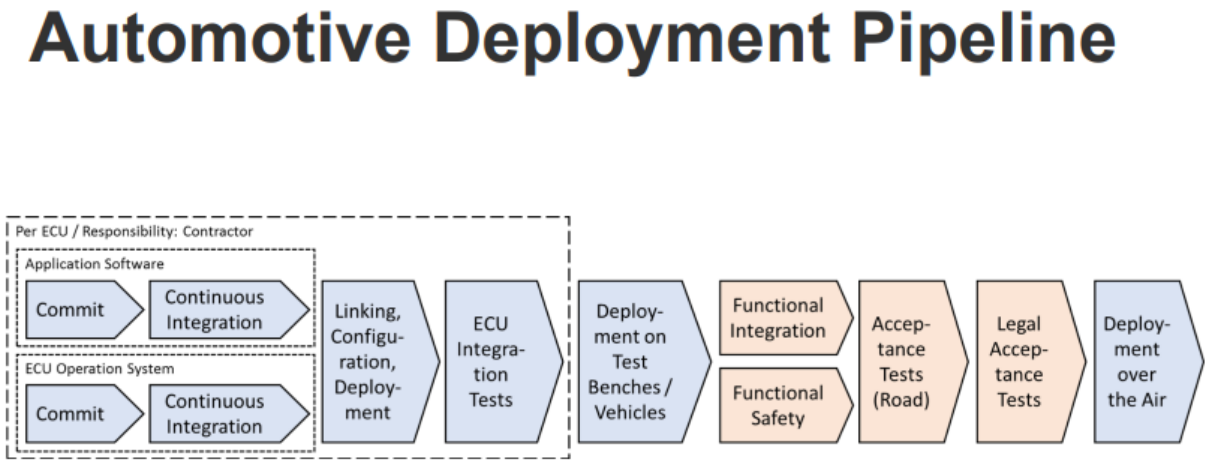


Figure 4.6: Automotive Deployment Pipeline
ECU= Electronic Control Unit / Steuergerät

4.2.3 Continuous Deployment patterns

4.2.4 Blue-Green Deployment

Einfach, dass man immer **zwei lauffähige Systeme** bereithält (eins mit bisherigen Version und anderes mit der neu deployten Version), zwischen man immer **schnell umschalten** kann, damit Umschalten beim Deployment vom Cutover schnell gehen kann.

4.2.5 Canary Releasing

Nicht alle Nutzer:innen werden sofort auf die neue Version umgestellt, damit bei Fehlern in der Version nicht direkt alle Nutzer:innen von den Auswirkungen betroffen sind.
Der Anteil wird dann **beobachtet** und sobald man sich **sicher genug** ist, dass keine Probleme auftreten, lässt wird die neue Version auf alle Nutzer:innen ausgerollt.

4.2.6 Dark launching

Neuen Features für Nutzer:innen werden **unsichtbar deployt**. Echte Nutzer*innen-Interaktionen werden zusätzlich zur alten Version auch an das **Feature im Backend weitergegeben** und das Verhalten analysiert.
Erst wenn das neue Feature sich auch mit den **echten Interaktionen verhält wie erwartet**, schalten wir auf das neue Feature um.

4.2.7 A/B Testing

Ein **Teil** der Nutzer:innen wird zur einen Variante geleitet, der andere Teil zur anderen. Anhand passender Messungen wird dann verglichen, welche Variante besser ist.

4.2.8 Feature Toggles

Ins System einbauen, einzelne Features **ein und auszuschalten**, damit unfertige Features im Deployment deaktiviert werden können. Weil Mainline Features enthalten kann, die unfertig sind.

Chapter 5

Qualität

5.0.1 Gravins Qualitätsansätze

1. **User based value:** nutzerbasierter Blickwinkel um mit den Nutzern zu diskutieren.
2. **Product based:** zu messbaren Produkteigenschaften transferiert.
3. **Manufacturing based** Eigentliche Software im Entwicklungsprozess

Schlussfolgerung: Zu unterschiedlichen Zeitpunkten braucht man **unterschiedliche Blickwinkel** in der Entwicklung.

5.0.2 Stakeholders

Ein Stakeholder ist jeder, der ein **berechtigtes Interesse** an dem System hat. Übliche Stakeholder sind die Endanwender*innen, die Kund*innen, die Entwickler*innen und die Betreiber*innen.
Alle haben eigene, manchmal sogar sich widersprechende Bedürfnisse und Ziele.
Ob und wie diese Ziele erfüllt werden hängt wiederum eng mit der Qualität zusammen. **Qualität hängt vom Stakeholder ab.**

5.0.3 Qualitätenbaum

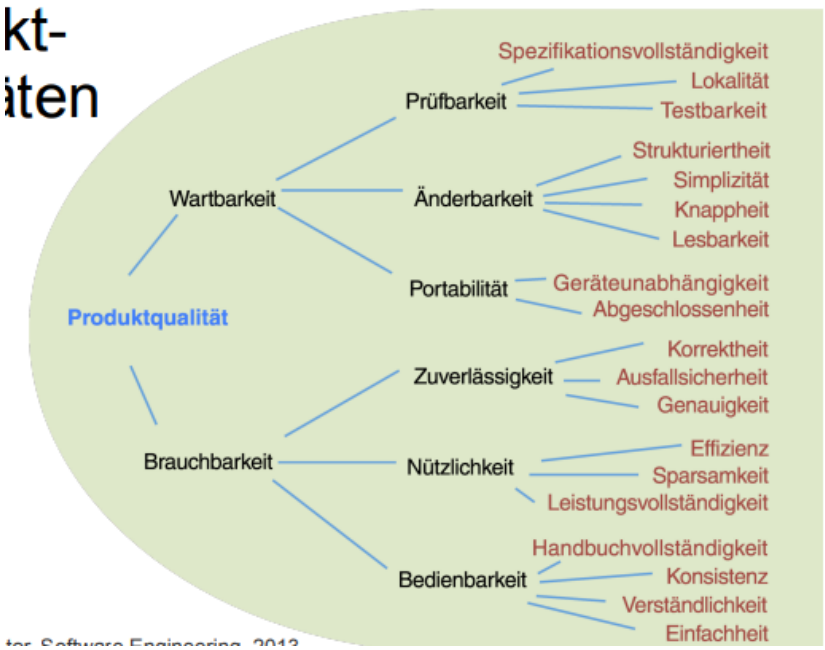


Figure 5.1: Baum
rot:Produktqualitäten

5.0.4 Qualitätsmodell nach Avizienies

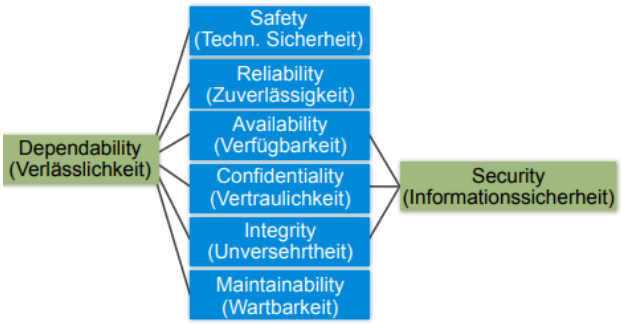


Figure 5.2: Qualitätsmodell nach Avizienies

5.0.5 ISO 25010

Quality in use

- **efficiency**
- **effectiveness**
- **context coverage:** context completeness flexibility
- **freedom from risk:** economic high risk mitigation, safety and health risk, environmental risks
- **Satisfaction:** usefulness, trust, comfort

Product Quality

- **Functional suitability:** completeness, correctness, appropriation
- **performance efficiency:** time behaviour, capacity, resource util.
- **Compatibility:** co-existence, interoperability
- **Usability:** Appropriateness recognizability, Operability, User error protection, User interface aesthetics, Accessibility
- **Portability:** adaptability, installability, replaceability.
- **Security:** confidentiality integrity, accountability, authenticity.
- **Maintainability:** modularity, reusability, analysability, testability, modifiability.
- **Reliability:** maturity availability, fault tolerance, recoverability

5.0.6 Zuverlässigkeit

Die Wahrscheinlichkeit einer fehlerfreien Arbeitsweise einer Software für eine bestimmte Zeitdauer in einer bestimmten Umgebung.
Es beschreibt die **Häufigkeit des Auftretens** von Fehlern und wird oft mit Qualität gleichgesetzt.

5.0.7 Fehlerterminologie

- **Mistake:** Fehlerbehandlung
- **Fault:** Fehlerursache
- **Error:** Fehlerzustand
- **Failure:** Ausfall

5.1 Test-driven Development

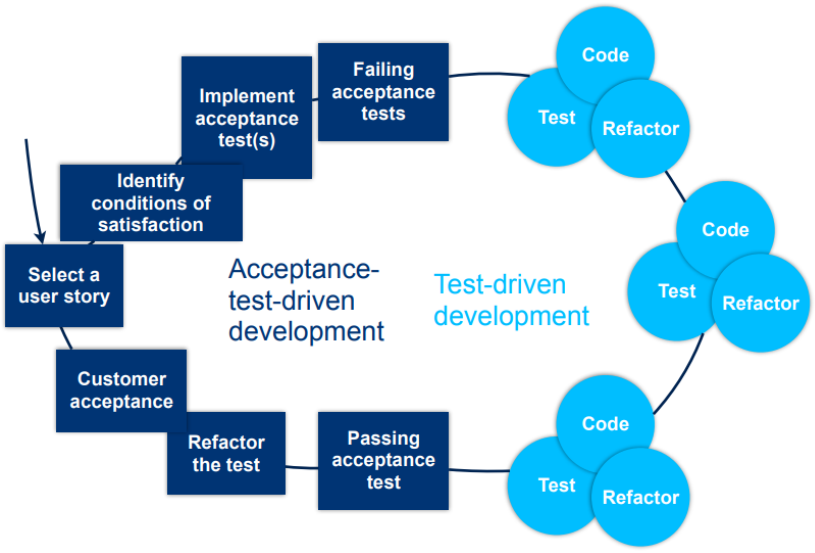


Figure 5.3: ATDD und TDD

Die drei Gesetze des TDD:

1. Du darfst keinen Produktionscode schreiben bevor du einen fehlgeschlagenen Unit-Test hast.
2. Du darfst nicht mehr von einem Unit-Test schreiben als notwendig ist, um fehlzuschlagen; nicht zu kompilieren ist fehlschlagen.
3. Du darfst nicht mehr Produktionscode schreiben, als notwendig ist, um den derzeit fehlschlagenden Test zu bestehen.

Chapter 6

Code verstehen(wtf)

6.1 Inhärente/zufällige Komplexität

Inhärent: Weil Domain schwierig/ komplex: Code komplex. **Zufällige:** schlechter Code

6.1.1 Mentales Modell

- **Wissen über Textstruktur:** Kontrollstrukturen, Variablendefinitionen, Aufrufhierarchien, Parameterdefinitionen
- **Chunks:** Verschiedene Ebenen der **Abstraktion** von Textstrukturen
- **Pläne:** Wissenselemente, um **Erwartungen und Interpretationen** zu entwickeln und validieren, wie z.B. Sortieralgorithmen oder Domänenwissen
- **Hypothesen:** Annahmen über das Programm, die beim Verstehen entstehen (warum, wie und was?)
- **Chunking:** Das Erschaffen **neuer, abstrakterer** Chunks
- **Cross-Referencing: Teilen** des Programms **funktionale Beschreibungen** zuzuweisen
- **Beacons:** Marken im Programm, die vorhandenes Wissen bei Leser*innen aufruft, z.B. Programmiererfahrung, dass Swap-Funktion oft in Sortierfunktionen benutzt wird.
- **Rules of Discourse:** Regeln zu **Programmierkonventionen und -standards**

6.1.2 McCabes zyklomatische Komplexität

Formel: E -V + 2p (p verbidende Komponenten) **Schwächen:** beschränkte Aussagekraft, keine Aussage über Lesbarkeit, Abhängigkeit von Darstellung.

6.1.3 Gewünschte Eigenschaften von Bezeichnern

Konsistenz: Eindeutiger Name für ein Konzept
Prägnanz: Korrektur direkter Namen des Konzepts

6.1.4 Programmierrichtlinien

Häufig sind unterschiedliche Lösungen möglich. Programmierrichtlinien sind eine Festlegung (teilweise nicht objektiv begründbar), um Einheitlichkeit zu erreichen. Darüber hinaus werden dadurch gewonnene Erfahrungen festgehalten in der Hoffnung Probleme nicht mehrfach zu erzeugen. Um Qualität im Code zu versichern, sind folgende Richtlinien möglich:

- Strukturierte Programmierung:
- Gültigkeitsbereich von Variablen: Welche Programmteile sind von Änderungen der Variable betroffen?
- Use Intention-Revealing Names: sinnvolle Bezeichner
- Länge und Schachtelung: Aufteilen in mehrere Methoden

- Don't Repeat Yourself (DRY) aka avoiding clones
- Don't comment bad code – rewrite it.

6.1.5 Refactoring

ändern des Code ohne das Verhalten zu ändern → genannte Probleme lösen

Mögliche Dinge:

- Verbesserung von Variablen- oder Methodennamen
- Anwendung eines konsistenten Styleguide
- Verbesserung der Package-Struktur (Stichwort Coupling and Cohesion)
- Verbesserung der Code-Kommentare
- Verbesserung der Dokumentation (README)
- Verbesserung der Fehler Behandlung
- Verbesseruf der Usability/ User experience
- Behebung von durch Tools identifizierter Code Smells oder Bug Patterns
- Extract Method oder andere refactorings(extract Interface, pull up method)
- Einsatz von Patterns wie zb Mediator zur Verbesserung des Designs
- Anwendung von SOLID Prinzipien

Extract Method (Gegenstück: Inline Method): Teil einer Methode wird in eine eigene Methode ausgelagert

Pull-Up Method (Gegenstück: Push-Down Method): Methode wird von Sub-Klasse in die Basis-Klasse verschoben

6.2 Qualitätssicherung/ Quality assurance

Alles, was zu tun ist, damit man der Qualität eines Produkts ***trauen kann, gehört dazu.***
Nachweisen von, *ob* es gut ist und *wie* man es besser machen kann.

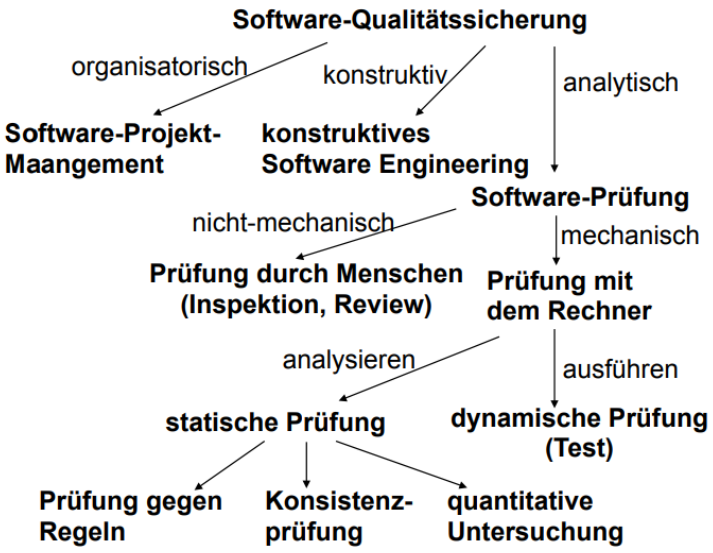


Figure 6.1: Qualitätssicherung

6.2.1 Validierung/ Verifikation

Validierung: richtges System für die Nutzer:innen ?
Verifikation: System richtig entwickelt?

6.2.2 Reviews

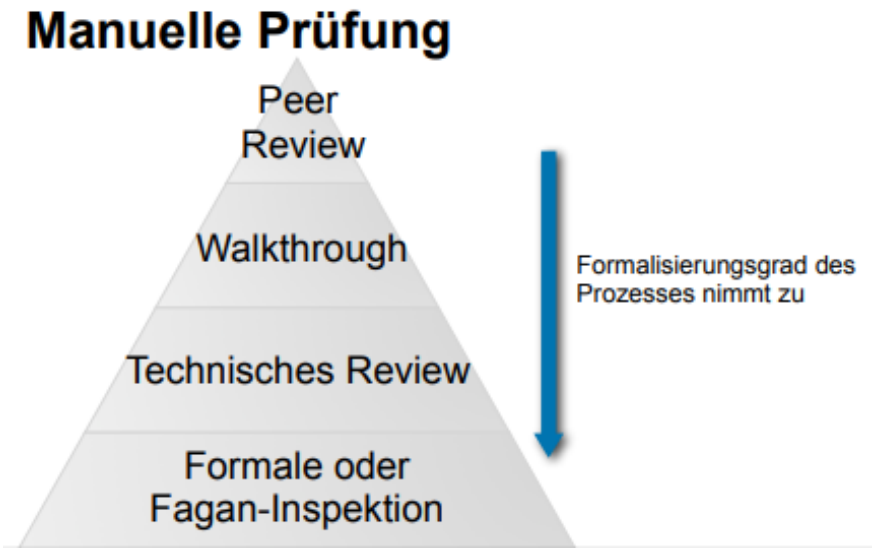


Figure 6.2: Manuelle Prüfung

6.2.3 Rollen

Moderator:in:leitet das Review.
Notar:in: führt das Protokoll.
Autor:in: Urheber:in des Prüflings
Gutachter:in: Kolleg:innen, die Prüfling begutachten.
Manager:in: hat Auftrag zur Erstellung des Prüflings gegeben. Sollte vor allem in den **ersten paar Reviews** einfach nicht anwesend sein lol.
Reviewteam: alle außer Autor:in

6.2.4 Fragenkatalog

Aspekt "Form": Ist die Darstellung im Dokument **sinnvoll?**
1. Sind **alle Anforderungen erkennbar**, d. h. von Erklärungen unterscheidbar?
2. Sind alle Anforderungen **eindeutig referenzierbar**?
3. Ist die Spezifikation jeder **Anforderung eindeutig**?
4. Sind alle Anforderungen **überprüfbar formuliert**?
Aspekt "Benutzerschnittstellen": Sind alle Schnittstellen **eindeutig spezifiziert**?
5. Sind alle Benutzerklassen des Systems (Dauerbenutzer, gelegentliche Benutzer, System-Administrator etc.) **identifiziert**?
6. Ist die **Bedienschnittstelle** für jede der Benutzerklassen festgelegt?
7. Ist die **Bedienphilosophie einheitlich**?
8. Ist das beschriebene Bedienkonzept den **Vorkenntnissen der Benutzer angemessen**?

Chapter 7

Projektmanagement

7.0.1 Ziel

Das Projekt erfolgreich durchzuführen und abzuschließen. Erfolgreich bedeutet: Das Projekt erzielt die definierten Resultate in der geforderten Qualität innerhalb der vorgegebenen Zeit und mit den vorgesehenen Mitteln. Weitere sekundäre Ziele sind in der Regel

- Aufbau oder Verstärkung eines guten Rufs auf dem Markt,
- Aneignung von Kenntnissen, die zukünftig benötigt werden,
- Entwicklung wiederverwendbarer Komponenten,
- Wahrung eines attraktiven Arbeitsklimas für die Mitarbeiter

7.0.2 Kickoffs

Müssen drei große Bereiche abdecken:
Purpose/Zweck (Vision, Mission),
Alignment/ Ausrichtung(values, principles, core team, working agreements) und
Context (Boundaries, committed resoruces, project community interaction, perspecive analysis).

7.0.3 Managementaufgaben

- **Erstellung der Aufgaben:** Das Angebot ist der eigentliche Startschuss des Projekts. Es legt grundsätzlichen Dinge des Projekts fest und hat damit bereits starken Einfluss auf den Projekterfolg.
  Projekt- Zeitplanung:
  Ohne Pläne kann ein Projekt nicht geführt werden; Planungsfehler lassen sich später nicht kompensieren.

- **Projektkostenkalkulation:** Die Basis für das Projekts und für die weitere Kontrolle über das Projekts ist initial eine Schätzung der Kosten, später die Überwachung.

- **Projektüberwachung und Reviews:** Arbeitsergebnisse und der Projektfortschritt müssen bewertet werden, es muss überwacht werden, ob sich die Beteiligten an Vereinbarungen halten.

- **Auswahl, Beurteilung und Führung des Personals:** Führen heißt: vorangehen, den Weg zeigen, auch die Gruppe mitziehen. Die meisten Entwickler wollen gute Leistungen erbringen, brauchen aber auch Orientierung und Bestätigung.

- **Präsentation und Erstellen von Berichten:** Der Projektleiter steht zwischen Management, Kunden oder dem Marketing und Mitarbeitern. Für das Management repräsentiert er das Projekt, für den Kunden die Herstellerfirma, für das Marketing die Technik, für die Mitarbeiter die Leitung der Firma. Nur wenn er auf allen Seiten zuhört und nach allen Seiten Informationen weitergibt, hat er eine Chance.

- **Sicherstellung günstiger Rahmenbedingungen:** Ein Projekt gedeiht am besten, wenn die Mitarbeiter, ausgestattet mit der notwendigen Infrastruktur, konzentriert und ungestört stabile Ziele verfolgen können. Aber um das Projekt herum gibt es viele Anfechtungen (wankelmütige Kunden, unklare Zielsetzungen, Restrukturierungen, Sparmaßnahmen, enge Büros, lange Wege etc.).
  Es ist Aufgabe des Projektleiters, das Projekt vor diesen störenden Einflüssen zu schützen.

7.0.4 Planung

7.0.5 Planungsaspekte

Planung der Aufgaben: Arbeitspakete werden festgelegt.
Planung der Termine: Deadlines werden festegelegt + Endtermin
Planung der Resourcen: Aufwandeinschätzung, Kosten und Budgetermittlung.

7.0.6 Arbeitspakete

Man zerteilt die Entwicklungsaufgaben so lange in Pakete bis es nicht mehr geht.
Arbeitspaket: eine Aufgabe, die Entwickler:innen oder kleines Team in max. einem Monat mit gut planbarem Aufwand lösen kann.

7.0.7 Kriterien für Arbeitspakete

- kann durchgängig erledigt werden, ohne dass es Koordinationszwänge gibt
- Fortschritt und das Ende sind objektiv feststellbar(messbar)
- der Aufwand und Termine sind einschätzbar.

7.0.8 Interne und externe Meilensteine

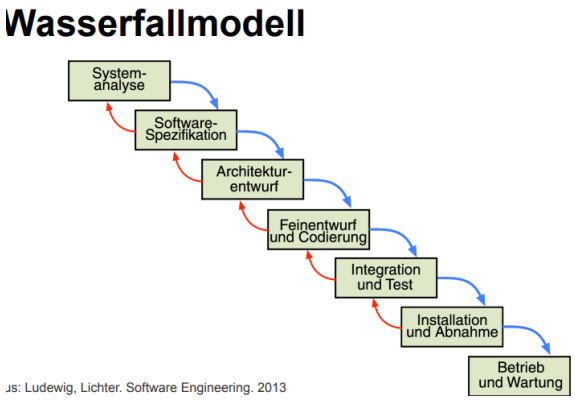
- **externe Meilensteine:** definieren Ergebnisse, die aus Sicht des Auftraggebers wichtig sind. Auftraggeber entscheidet nach Bewertung der erzielten Ergebnisse, ob Arbeitspakete der nächsten Phase gestartet werden dürfen.
- **interne Meilensteine:** sinnvoll, wenn Zeitraum zwischen zwei Meilensteinen so groß ist, dass man Zwischenkontrollpunkte braucht. Sind da, um den Fortschritt des Projekts sichtbar zu machen.

7.1 Vorgehensmodelle

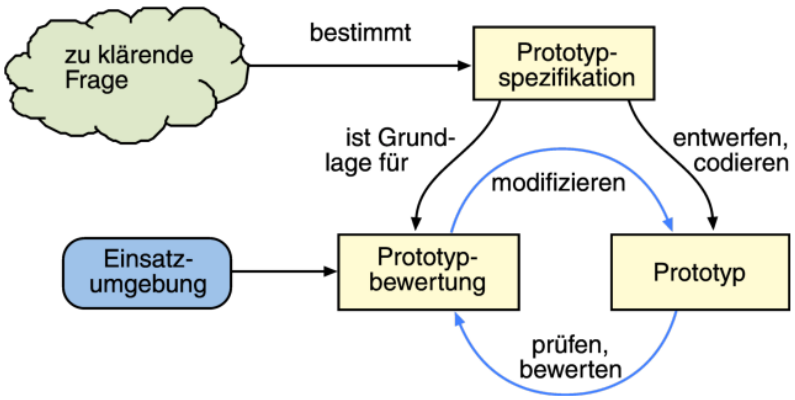
7.1.1 Wasserfallmodell

Sind erfolgreich, weil im Laufe der Entwicklung Änderungen und Korrekturen notwendig werden. *Probleme:*

- Einbahnstraßenmodell
- unvermeindliche Rücksprünge



7.1.2 Prototyping



Nach Floyd:

- **Exploratives Prototyping:** Ziel: Analyse und ergänzen (funktionale Protypen)
- **Experimentelles Prototyping** Ziel: technisch Umsetzung eines Entwicklungsziels (funktionale Prototypen, Labormuster)
- **Evolutionäres Prototyping:** eigentlich kein Prototyping, sondern spezielles Verständnis des Entwicklungsprozesses.

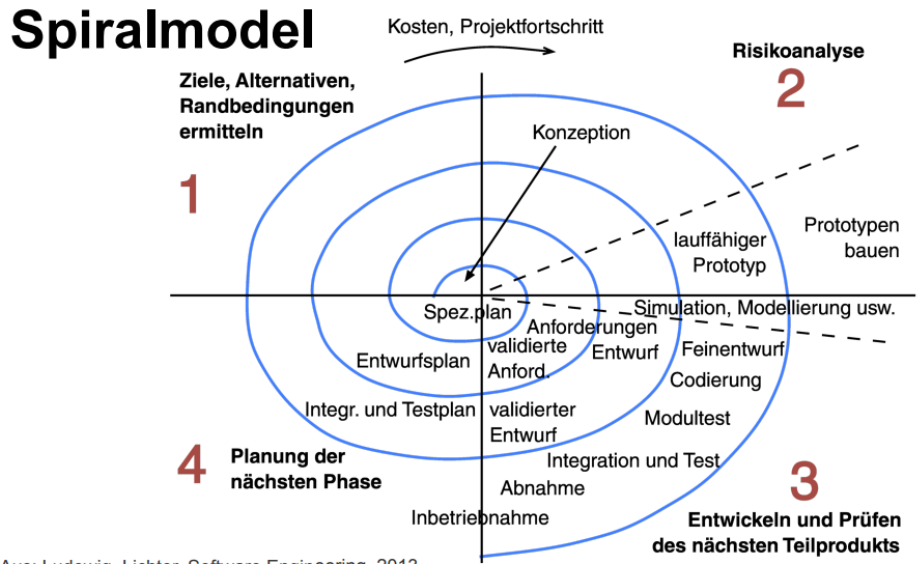
7.1.3 Iterative Entwicklung

Ein großes Projekt wird in kleinere Folgen/Iterationsschritte gegliedert. Das Endprodukt ist dann aus allen erfahrungen von den ersten Durchgängen.In jeder Iteration werden die Tätigkeiten Analysieren, Entwerfen, Codieren und Testen ausgeführt, und das resultierende System wird erprobt.

7.1.4 Inkrementelle Entwicklung

Das zu entwickelnde System wird nicht in einem Zug konstruiert, sondern in einer Reihe von aufeinander aufbauenden Ausbaustufen. Jede Ausbaustufe wird in einem eigenen Projekt erstellt, in der Regel auch

ausgeliefert und eingesetzt. Zu Beginn einer inkrementellen Entwicklung muss sichergestellt sein, dass in der ersten Ausbaustufe, dem Kernsystem, ein zentraler, funktional nutzbringend einsetzbarer Ausschnitt des gesamten Systems realisiert ist. Nachdem das Kernsystem realisiert ist, kann das System im Anwendungsbereich eingesetzt werden.Das zu entwickelnde System bleibt in seinem **Gesamtumfang offen**; es wird in Ausbaustufen realisiert. Die erste Stufe ist das Kernsystem.Das System wird schrittweise realisiert, wobei es typisch keinen definierten Endzustand gibt. **Vorteile:** frühe Rückkopplung der Erfahrung und kurze Entwicklungszeiten für Inkrements.

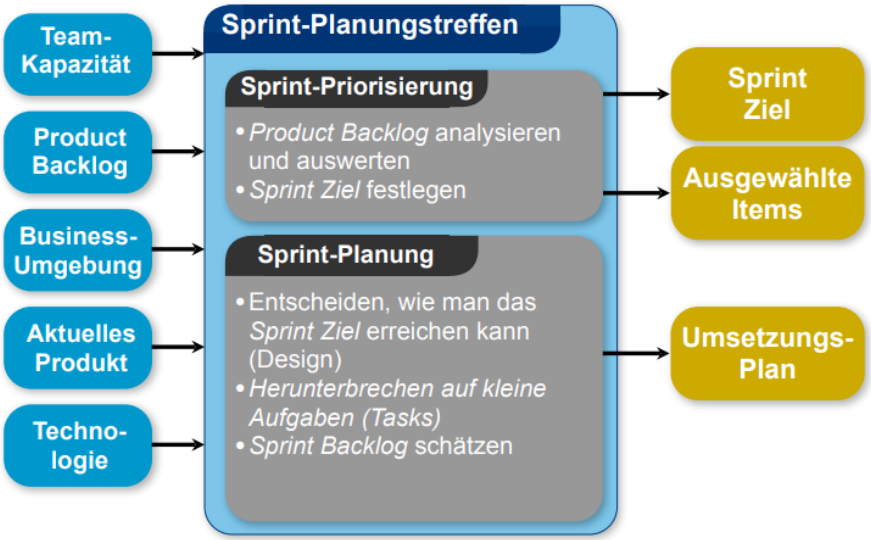


Fundamental für Spiralenmodell ist das *Risiko*. Diese sollten schnell und früh gefunden und bekämpft werden.

7.1.5 Scrum

7.1.6 Scrum Charakteristika

- **selbst organisierte** Teams
- Produkt schreitet in **Sprints** fort
- Anforderungen sind als Listeneinträge im Prodct Backlog festgehalten
- **keine spezifische Entwicklungsmethode** vorgeschrieben, eher generative Regeln im agilen Umfeld



7.1.7 Risikomanagement

- I**Identifikation** von Risiken
- **Analyse** und **Bewertung** der Risiken
- Planung von **Gegenmaßnahmen**
- **Verfolgen der Risiken** und der gewählten Gegenmaßnahmen

7.1.8 Risk Items and their solutions

1. **Personelle Probleme:** *Gegenmaßnahme:*Team Building, Morale Building, Schlüsselpersonen früh ins Projekt.
2. **Unrealistische Zeit- und Kostenpläne:** *Gegenmaßnahme:*Detaillierte Kosten und Zeitplanung, Design to cost, inkrementelle Entwicklung, Wiederverwendung.
3. **Entwicklung falscher Fuktionalität** *Gegenmaßnahme:* Organisationsanalyse, Nutzungsanalyse, Nutzerbefragung, Prototyping, frühes Benutzerhandbuch
4. **Entiwcklung der flaschen UI** *Gegenmaßnahmen:* Prototyping, Nutzungsanalyse
5. **Gold plating** *Gegenmaßnahmen:*Anforderungen zurückstellen, Kosten Nutzen Analyse, Design to cost
6. **Kontinuierlicher Strom von Anforderungsänderungen** *Gegenmaßnahmen:* Hohe Änderungsschwelle, inkrementelle Entwicklung
7. **Probleme mit extern durchgeführter Aufgabe** *Gegenmaßnahme:*Referenzen prüfen, Prüfungen vor der Auftragsvergabe, Prototyping, Beauftragung mehrerer Zulieferer
8. **Probleme mit extern entwickelten Komponenten:** *Gegenmaßnahme:* Benchmarking, Modellierung, Protyping, Kompatibilitätsanalyse
9. **Probleme mit Echtzeit Performanz** *Gegenmaßnahme:* Simulation, Benchmarking, Modellierung, Tuning
10. **Überforderung der Möglichkeiten der Informatik** *Gegenmaßnahmen:* technische Analyse, Kosten/nutzen Analyse, Prototyping

7.1.9 Arten von Teams

- **Ein Personen Team**
  Merkmal: **Eigenständige Arbeit** von einer Person an einer Aufgabe.
  Vorteil: : Fast **kein Kommunikationsaufwand**.
  Nachteil: **Keine Gesprächspartner**, kein **Dokumentationsdruck**, Risiko des Ausfalls.

- **Gruppen aus zwei Personen** (Pair programming mit Driver und Observer)
  Merkmale: durch den freien Beschluss der Beteiligten zur **Zusammenarbeit**. **Keine Führungsrolle!** (Doppel) oder durch die Weisung an einen Helfer (Sherpa), einen Spezialisten zu unterstützen. (Tandem)
  Vorteil: Gespräch möglich, keine Katastrophe durch Ausfall einer Person. Sinn des Tandems kann es auch sein, das Wissen des Spezialisten auf zwei Köpfe zu verteilen. Folge: Sherpa = Schüler
  Nachteil: Schwierig, wenn sich die beiden nicht gut verstehen

- **Anarchisches Team mit autonomen Entwickler:innen.**
  Merkmale: **Autonomes** Arbeiten von Entwickler:innen nach eigenen Maßstäben und Vorgaben.
  Vorteil: Entwickler sind **selbstbestimmt,** keine Hierarchie-Probleme, **kaum bürokratische** Hemmnisse.
  Nachteil: Standards, **Normen** lassen sich **nicht durchsetzen.** Die Entstehung der erforderlichen Resultate ist Glückssache (d. h. gewisse Dokumente entstehen in aller Regel nicht). Die Organisation insgesamt ist nicht lernfähig, Planung, Einführung neuer Methoden und Werkzeuge sind von der Laune der Mitarbeiter abhängig.

- **demokratisches Team**
  Merkmale: Die Beteiligten sind **grundsätzlich gleichberechtigt.** Sie erzielen durch ausreichende Kommunikation einen **Konsens** über die Ziele und Wege, und sie verhalten sich diszipliniert.
  Vorteile: Die **Fähigkeiten** der Beteiligten werden optimal genutzt, Probleme werden frühzeitig erkannt und gemeinsam bekämpft.
  Nachteile: Hoher **Kommunikationsaufwand**. Unter Umständen Paralyse (Dissens, **Fraktionsbildung**)

- **hierarchisches Team**
  Merkmale: : Die Gruppe steht unter der **Leitung** einer Person, die für die Personalführung, je nach Projektform auch für das Projekt verantwortlich ist. Varianten: Gruppenleiter übernimmt Stabs- oder Linienfunktion.
  Vorteile: : **Einfache Kommunikationsstruktur**, klare Zuständigkeiten, auf Mitarbeiterebene gute Ersetzbarkeit.
  Nachteile: Lange Kommunikationswege (d. h. oft schlechte Information), der Gruppenleiter stellt ein hohes Risiko dar; Gruppenmitglieder sind **kaum zur Kooperation motiviert.**

- Chief- Programmer TeamMerkmale:
  Merkmale: Gruppe aus Chief Programmer und seinem Stellvertreter, Bibliothekar, der alle Verwaltungsfunktionen übernimmt, sowie einige Programmierer. Kann auch weitere Spezialisten geben.
  Vorteile: : Die Gruppe kann — wie ein Operationsteam, das Vorbild dieser Struktur war — außerordentlich **effizient arbeiten.**
  Nachteile: : **Hohe Ansprüche** an die Disziplin, vermutlich auch die Gefahr, dass der Chief-Programmer abhebt, sich überschätzt

7.1.10 Funktionale Organisation

Hersteller hat immer wieder **ähnliche Aufgaben.** Gruppen oder Abteilungen mit **Spezialisten** für die Teilaufgaben. Das Produkt entsteht durch das **Zusammenwirken der Abteilungen.**
Die Linienorganisation reicht aus, eine Sekundärorganisation ist nicht erforderlich. Die Strukturen sind **projektunabhängig;** es gibt kein Projekt! Jede Person hat eine definierte (feste) **Rolle.** Die Zwischenresultate fließen von Abteilung zu Abteilung.

Vorteile: stabile Zuordnung der Mitarbeiter, keine Konflikte um Prioritäten.
Nachteile: : Alles, was am Projektbegriff hängt, fehlt: Identifikation mit einem Projekt, Reaktion auf Probleme, Motivation durch Ziel.

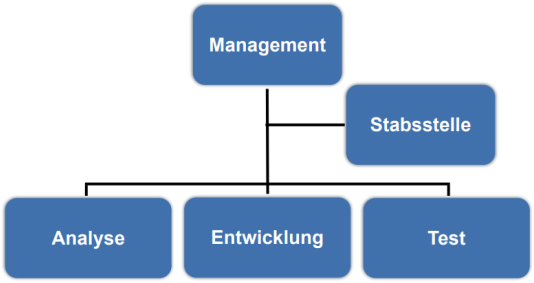


Figure 7.1: Funktionale Organisation

7.1.11 Aufwandsschätzungen

- Expertenschätzung
- Algorithmische Schätzung
- Top Down: aus globalen Größen werden zu **erwartende Größen abgeleitet**.
- Bottom up: aus atomaren Größen wie Codezeile wird Aufwand **abgeleitet**
- **Aktivitäten-basierte** Schätzung

- **Produkt basierte** Schätzung

Chapter 8

Modellierung

8.0.1 Komplexität

Detailkomplexität: einfach viel und schwierig.
dynamische Komplexität: unklare Ursache- Wirkungszusammenhänge

8.0.2 Präskriptive und deskriptive

Präskriptiv: existiert noch nicht
deskriptiv: existiert schon und beschreibt

8.0.3 Sichten(Views) und Perspektive(Viewpoints)

Perspektive: modelliert **System**aspekt. Menge von Sichttypen für einen **bestimmten Zweck**. Definiert **Inhalt ihrer Sicht**.
Definiert durch die Interessengruppen/ Stakeholder.

Sichttypen: modellieren **Systemteile** unter einem **bestimmten Zweck/ bestimmter Aspekt**. Definieren die Notation und Auswahlkriterien.

8.0.4 UML

8.0.5 Komposition vs Aggregation

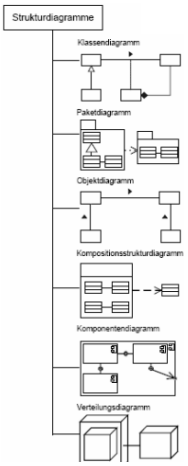
Beides sind Teil-Ganzes-Beziehungen zwischen Objekten (Parent und Child). Bei der Aggregation besteht aber eine größere Eigenständigkeit des Teils und es kann auch unabhängig vom Ganzen existieren, z.B. Studierende (Children), die einer bestimmten Vorlesung (Parent) zugeordnet sind. Wenn die Vorlesung gelöscht wird, existieren die Studierenden weiterhin. Bei der Komposition dagegen ist der Lebenszyklus des Teils an das Ganze gekoppelt, d.h. das Teil kann nicht ohne das Ganze existieren, z.B. die Lieferadressen eines Kunden. Wenn der Kunde gelöscht wird, werden auch alle seine Adressen mitgelöscht.
Komposition: stirbt ohne das andere.
Aggregation: gehört dazu, kann auch ohne.

Komponentendiagramme

Modelliert die Software Architektur, Bibliotheken, Komponenten, Datenbanken, usw.

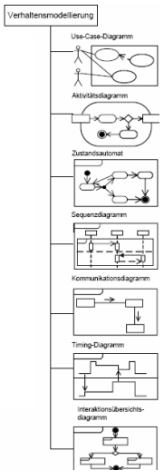
8.1 Diagramme

Strukturdiagramme



- **Klassendiagramm** (class diagram)
  Modellierung der Klassen mit Methoden, Attributen und Beziehungen
- **Paketdiagramm** (paket diagram)
  Modellierung der logische Struktur der Modellelemente mit deren (z.T. abstrakten) Abhängigkeiten
- **Objektdiagramm** (object diagram)
  Modellierung von „Momentaufnahmen“ konkreter Objekte mit Werten und Beziehungsausprägungen
- **Kompositionsstrukturdiagramm** (composite structure diagram)
  Modellierung der internen Struktur eines Modell-elements (Classifiers) sowie dessen Möglichkeiten zu Interaktion mit anderen Systemkomponenten
- **Komponentendiagramm** (component diagram)
  Modellierung der Komponenten (=Module) und deren Abhängigkeiten
- **Verteilungsdiagramm** (deployment diagram)
  Zeigt die (physischen) Geräte des Systems im Betrieb

Verhaltensdiagramme



- **Use-Case Diagramm** (use case diagram)
  Modellierung der Anwendungsfunktionalität aus der Sicht der Anwender
- **Aktivitätsdiagramm** (activity diagram)
  Modellierung des dynamischen Ablaufs (i.d.R. eines Use Case)
- **Zustandsdiagramm** (state transition diagram)
  Modelliert die Zustände und Zustandswechsel (i.d.R. zu einer Klasse)
- **Sequenzdiagramm** (sequence diagram)
  Modelliert ein Szenario: den Botschaftsfluss zwischen Objekten
- **Kommunikationsdiagramm** (communication diagram)
  Dynamik wie Sequenzdiagramm; zusätzlich Objekteigenschaften
- **Timing-Diagramm**
  Präziser zeitlicher Verlauf
- **Interaktionsübersichtsdiagramm**
  Modellierung des Zusammenspiels einzelner Interaktionsdiagramme

Quelle: Mario Jeckle, 2003

Chapter 9

Architektur

Entwurf: Struktur und **Architektur** wird hier festgelegt. Spezifikation sagt nur, was das Produkt leistet, **aber nicht wie**: Architektur

9.0.1 Grob und Feinentwurf

Grobentwurf: vereinfachte Vorstellung der zu entwickelnden Systems zB. Systemteile und Subsysteme und deren Zusammenspiel
Feinentwurf: Verfeinerung der Subsysteme.

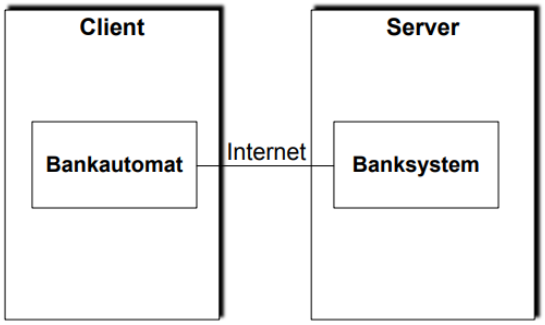
9.0.2 Aspekte der Architektur

Gliederung in überschaubare Einheiten
Festlegung der Lösungsstruktur
Hierarchische Gliederung: um Zusammenhänge systematisch zu entwickeln und zu verstehen.

9.0.3 Systemsicht und statische Sicht:

Systemsicht: zeigt, **wie** das zu entwickelnde System in die Umgebung **eingebettet** ist und wie es mit **anderen Systemen interagiert**(Systemgrenze, Schnittstellen).
Statische Sicht: zeigt zentralen Komponenten und ihre Schnittstellen, in der Regel hierarchisch verfeinert.
Dynamische Sicht: modelliert, wie die Komponenten zur Laufzeit zusammenarbeiten. Beschränkt auf wichtige Interaktionen.

9.0.4 Verteilungsdiagramm



9.0.5 Modularisierung

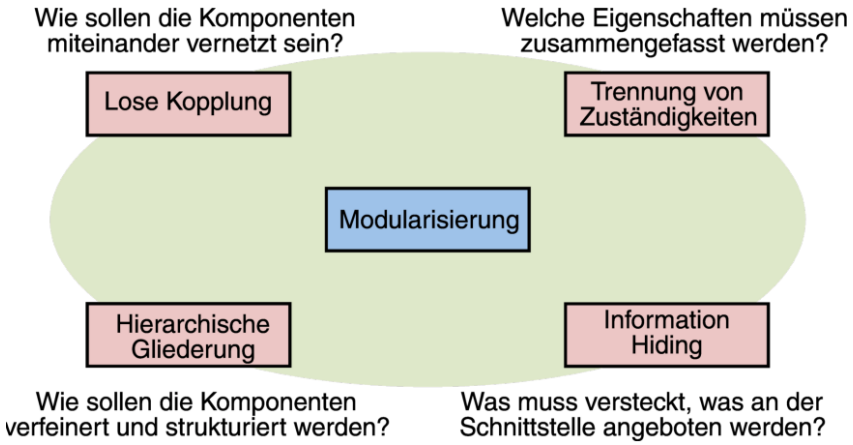
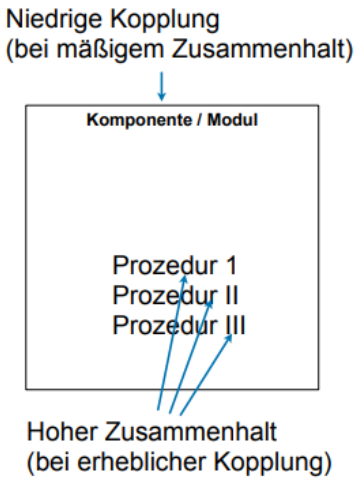


Figure 9.1: Prinzipien

Modularisierung eine Eigenschaft, wenn es dem Architekt:in gelungen ist, das System in solche Komponenten **zu unterteilen**, dass sie möglichst unabhängig voneinander verändert und weiterentwickelt werden können.

9.0.6 Ziele

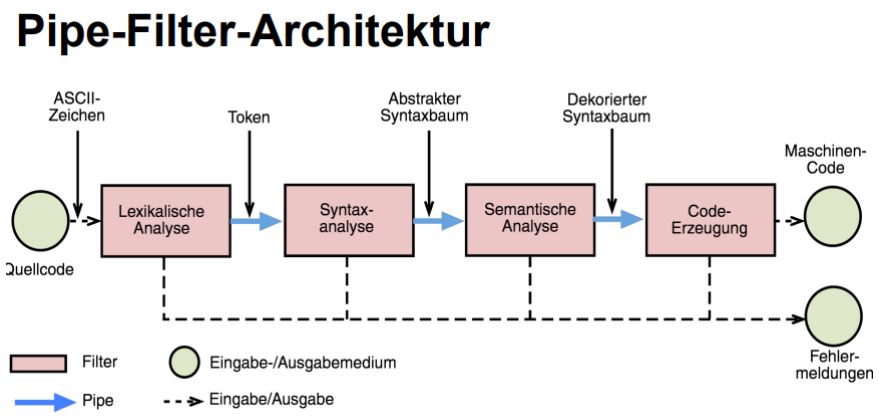


9.0.7 Separation of concerns

Trennung von Zuständigkeiten, denn jede Komponente sollte nur für einen ganz bestimmten Aufgabenbereich zuständig sein. Also **fachliche** und **technische** Komponenten **trennen** und Funktionalitäten eigenen Komponenten zu ordnen.

9.0.8 Architekturmuster

Schichtenmuster
man zerlegt das System in Schichten, also logisch zusammenhängende Komponenten und eine Schicht stellt Dienstleistungen zur Verfügung, die an der oberen Schnittstelle der Schicht angeboten werden. Schichten **bauen direkt aufeinander auf**.
Drei Schichten Architektur Aus **Präsentationsschicht** (Bedienungsoberfläche, Benutzersystem), **Anwendungsschicht**(Komponenten der Funktionalität), **Datenerhaltungsschicht** (Datenbank).
Pipe Filter Architektur Verarbeitungsschritte in Filtern realisiert, ein Filter verbraucht und erzeugt Daten. Pipes leiten Ergebnisse des Filters an nachfolgende Filter weiter. Das erste Filter bekommt seine



Daten aus der Datenquelle, das letzte liefert sie an die Datensenke.
Microservices Architektur Viele kleine separat installierbaren Einheiten, die die Architektur der Software bilden, aus mehreren Service Komponenten.

9.0.9 Architecutal decision AD

Technologieauswahl (Programmiersprache, IDE, Bibliothek oder Features)
Y Statements Zur Strukturierung von Architekturentscheidungen.

1. **Context:** funktionale Anforderungen, Architekturkomponente
2. **Facing:** nicht funktionale Anforderungen
3. **We decided:** Entscheidungsergebnis (wichtigster Teil)
4. **and negledcted:** nicht gewählte Alternativen
5. **to achieve:** Vorteile, die vollständige oder teilweise Erfüllung von Anforderungen
6. **acceoting that:** Nachteile oder andere Konsequenzen.

9.0.10 Entwurfsprinzipien SOLID

Single Responsibility (eine Aufgabe, Modularität)
Open closed (Information hiding und open für Vererbung)
Liskovian Substitution Principle
Interface segregation
Dependency inversion (alles von abstrakten Klassen abhängig machen)

9.0.11 Law of Demeter

Methode M eines Objekts O sollte nur auf Objekt, Parameter von M, Objekte die M selbst instanziiert und Objekte der eigenen Komponente zugreifen können.

9.1 Entwurfsmuster

9.1.1 Singelton

Eine global verfügbare Instanz pro Klasse blyat. Problem: keine Redefinition möglich.

9.1.2 Composite

hierarchische Teil- Ganzes Beziehung

9.2 Domain driven design

Fokussiert auf die Anwendungsdomände und die Fachlichkeit, erlaubt durchgängige Modellierung. BC, Domains, entities und Aggregates bilden die statische Sicht, domain Events ergänzen die dynamische Sicht. Event Storming ist ein Entwurfshilfsmittel u im Team mit den fachexpert:innen an einem Entwurf zu arbeiten.

9.2.1 Strategischer und taktischer Entwurf

strategischer Entwurf: grobe Strukturen, Wichtigkeit für das Geschäft, Prioritäten, mit bounded comtext arbeiten.
Taktischer Entwurf: feine Details, bounded context weiter ausarbeiten um sie in die technische Umsetzung zu bringen (aggregate eingesetzt)

9.2.2 Bounded context

Begrenzter Kontext mit einheitlicher Sprache damit alle Beteiligten auf dem gleichen Wissensstand sind. Beschreibt zuerst den Problemraum, wandert aber dann in den Lösungsraum und wird sich in der Form von Softwareartefakten in der fertigen Software wiederfinden. Auch viele Subdomains weil ohne bounded context würde das Domänenmodell immer komplexer werden.

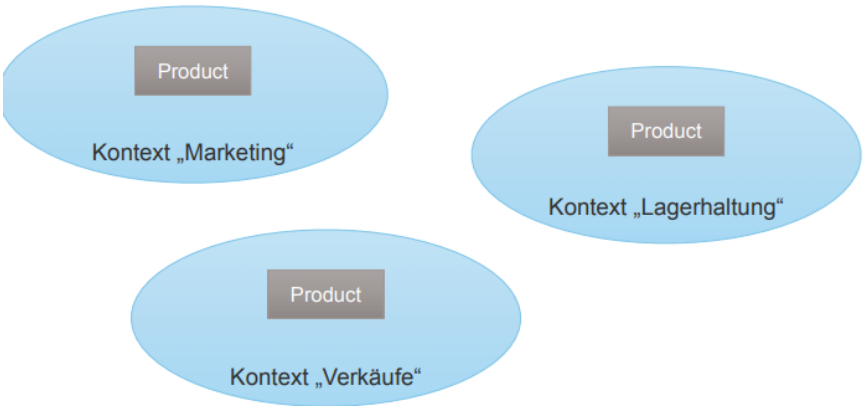


Figure 9.2: Beispiel bounded context für unterschiedliche Kontexte

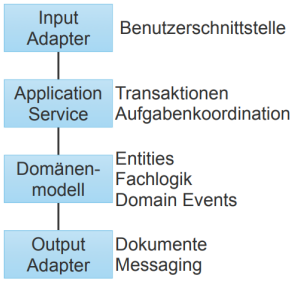


Figure 9.3: Bounded context

9.2.3 Subdomains

1. **Core domains:** strategisch Domäne, in die investiert werden soll und die den **Hauptwettbewerbsvorteil** darstellt.
2. **Supporting Subdomain:** individuelle Entwicklung ist notwendig, da es spezifisch für die Core Domain ist und diese unterstützt.
3. **Generic Subdomain:** kann evtl. als Standardsoftware eingekauft oder an anderes Unternehmen abgegeben werden. darstellt

9.2.4 Context Mapping

9.2.5 Ubiquitous language

für einen Bounded Context gültige Sprache erarbeitet, die alle, die in diesem Bounded Context arbeiten verstehen können. Die Sprache soll präzise aber knapp sein.

9.2.6 Entity

Modelliert ein individuelles Ding mit Identität, einfach eine Entität/Objekt blyat.

9.2.7 DDD aggregate

Problem: in einem bounded Contect können viele Entitäten enthalten sein, schwerer Überblick.
Lösung: nicht alle Entitäten sind eigenständig, einige können in Aggregaten zusammengefasst und behandelt werden.
Muster zur Modellierung auf Feindesign-Ebene zur strukturierung von Entitäten. **Achtung:** bruder nicht mit Collectionsverwechseln, die sind generisch und Aggregates beschreiben einen fachlichen Zusammenhang, der aber wiederum technisch durch eine Collection umgesetzt werde *kann*.

9.2.8 Aggregate

Zwei oder mehrere Entitäen, wobei eines das Aggregate root (gibt dem Aggregate den Namen) ist. Kann auch value objects enthalten. Ein Aggregate bildet eine **transaktionale Konsistenzgruppe**. Also alle Elemente innerhalb eines Aggregates sind bezüglich der Geschäftsregeln **konsistent**.

9.2.9 Value Object

Ein unveränderliches, konzeptionelles Ganzes. Einfach ein Wert der keine Identiät besitzt, weil es ein Wert/ Attribut ist.

9.2.10 Domain Event

Protokolliert ein Ereignis in einem BC, das **fachlich relevant** ist. in einem Bounded Context, das fachlich relevant ist. Der Name des Domain-Event-Typs sollte eine Aussage über ein **vergangenes Ereignis** sein, also ein Verb in der Vergangenheitsform. Sie bringen die **dynamische Sicht** in den Entwurf. Der Name muss konsistent zur Ubiquitous Language sein.

9.3 Wartung

Erblast: wird dringend gebraucht aber von niemanden verstanden lol. Vor allem wenn: sehr große Software, Entwickler:innen und Architekten nicht mehr verfügbar, veraltete Methoden und Sprachen, obsolete oder keine Dokumentation, basiert auf veralteter Hardware.

9.3.1 Risiken und Probleme

Changing prioties Wartungstätigkeiten werden oft unterbrochen.
Inadequate testing methods Der Regressionstest nach der Wartung bereitet Probleme, weil spezielles Know-how fehlt und weil die Testfälle und Testdaten nicht ausreichen
Performance measuremenrs difficulties Nicht definiert, wie die Wartung durchgeführt werden soll.
System doc is incomplete
Adapting to a rapidly changing business environment

9.3.2 Wartungskrise 1980

Wegen mangelnde Qualität der Entwicklung mit Ziel der Verbesserung der Wartbarkeit durch Prozesse, Testen und OO.

9.3.3 Legacy Software 1990

Wunsch nach Ablösung von 30 Jahre alter Software, weil hohe Kosten und Risiko.

9.3.4 Langelebige Systeme 2000

Die Relevanz der Systeme wird wahrgenommen Neue Forschungsgebiete, wie Clone Detection, Software-Leitstände, Software-Verjüngung,

9.3.5 Continous engineering

Continuous Integration and Delivery, Agile Vorgehensmodelle mit kurzen Iterationen, kein klarer Unterschied zwischen Initialentwicklung und Wartung erkennbar.

9.3.6 Klassen von Wartungsfällen

- perfective maintenance
- adaptive maintenance
- corrective maintenance
- preventive maintenance

9.3.7 Lehmans Gesetze der Software Evolution

- **Continuing change:** ein System, das benutzt wird muss kontinuierlich verfeinert werden.
- **Increasing complexity**
- **Self regulation:** die Evolution soll self regulating sein. **Conservation of organizational stability**
- **Conservation of familiarity**
- **Continuing growth**
- **Declining quality:** passiert mit der Qualität außer es wird rigoros an Wartungsaufgaben gemacht
- **Feedback system:** Programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

9.3.8 Auswirkungsanalyse

Zur Abschätzung der Kosten einer Änderung, Verstehen der Bedeutung und Abhängigkeiten der Artefakte, Dokumentation und Abschätzung der Qualität der Änderung und Festlegen, was in Regressionstests überprüft werden muss.

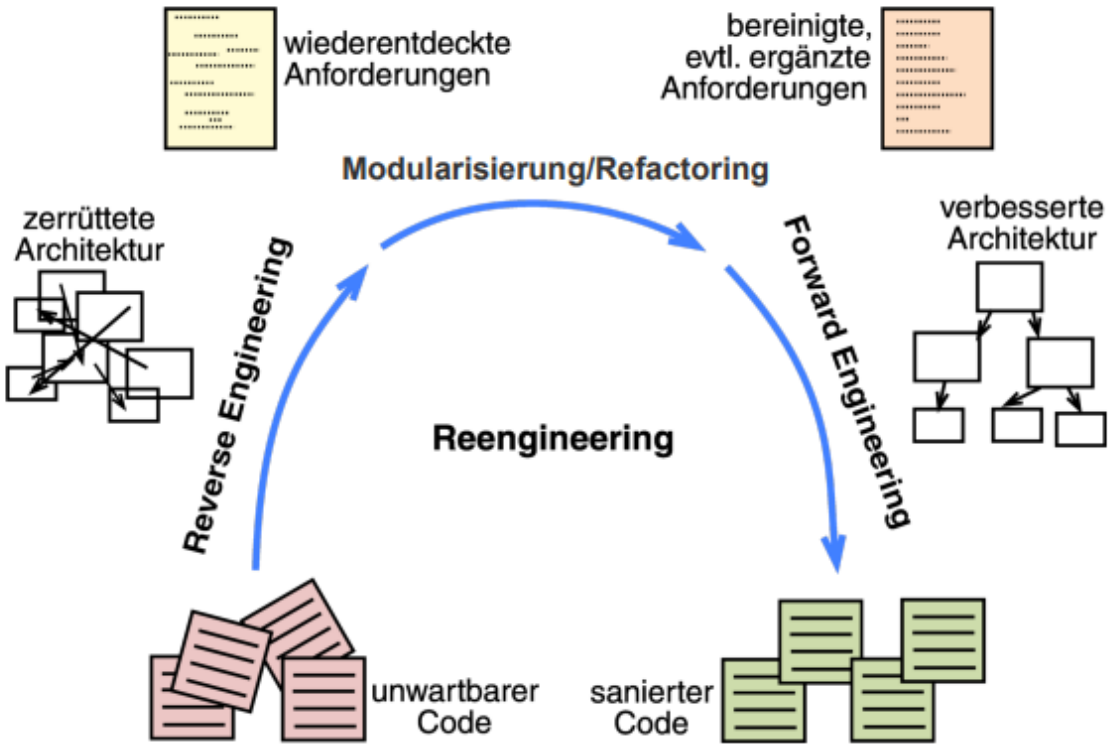
9.3.9 Nachverfolgbarkeit/ Traceability

Verikale Nachverfolgbarkeit: Abhängigkeit zwischen Teilen eines Artefakts
horizontale Nachverfolgbarkeit: die Abhängigkeit über Artefaktsammlungen hinweg.

9.3.10 Ablösung

Komplette Ersetzung zu teuer, zeitaufwendig und riskant.

9.3.11 Reengineering



9.4 Dokumentation

9.4.1 Arten von Dokumentation

- **integrierte Dokumentation:**die im **Programm** enthaltenen Kommentare, Bezeichner, das Layout
- **separate Dokumentation:** ist der Teil der Software, der **nicht** in den Programmen enthalten ist.