Inhaltsverzeichnis

Fallbeispiel 1: Stadt21 1
Das Projekt Stadt21 (Buch S116): 1
Allgemeiner Aufbau (Buch S 117-118, Folie S4, 12-16) 1
ELAK 2
Produktkatalog der öffentlichen Verwaltung 2
Abbilden der Geschäftsprozesse 2
Anliegen-Management 2
Postverwaltung 3
Administration 3
Legacy 3
DMS 3
Prozesserstellung (Buch S118-120) 4
Aufbau eines Prozesses 4
Sonstige Anforderungen 5
Aktenbearbeitung (Buch S121-122) 5
Administration (Buch S122) 6
Entwicklungsumgebung und Technologien (Buch S122-123, Folie S9) 6
Spezifische Aspekte (Folien S5-6) 7
Projektteam (Folien S7) 7
Steuerungsausschuss (Folien S8) 8
Entwicklungsprozess und Vorgehen (Folie S17-19) 8
Erkenntnisse (Folien S20-23) 9
Fallbeispiel 2: HISS 10
Projektentstehung (Buch S131, Folien S3) 10
Ziele und Umfang von HISS (S132-133, Folie S4-5) 10
Phasen und grober Aufbau (S133-134) 12
Phase 1 (Vorbereitung der Migration) 12
Phase 2 (Beginn der Migration) 12
Phase 3 (Vollbetrieb und Reifung) 12
Projektstruktur (Folien S9) 12
Stakeholder (Folien S10) 13
Motivation und Ziele des Fachkonzeptes (Buch S134-135) 14
Vorgehensweise bei der fachlichen Analyse (Buch S 135-136) 14
Re-Dokumentation von Datenbanktabellen und –spalten (Folien S16) 16

Phase Analyse – Was wird gebaut? 19
Beispiel Anforderungsqualität (Buch S141 – 142) 19
Woher kommen Anforderungen? - Stakeholder und Anforderungsquellen (S143-144, Folien S6-9) 19
Stakeholder („Anteilhalter“) 19
Stakeholder-Gruppe 20
Typische Stakeholder 20
Weitere wichtige Anforderungsquellen 20
In welchem Kontext wird gebaut? – Grundlagen der Analyse (S142-143, Folien S10-12) 21
Zielsetzung der Analyse 21
Vision (üblicherweise stabil) 21
Scope (üblicherweise veränderbar) 21
Machbarkeit und Machbarkeitsstudie (Buch S146, Folien S13) 21
Make or Buy- Einflussfaktoren (Buch S146-148, Folien S14) 22
Grobarchitektur (S148, Folien S15-16) 22
Grobarchitektur 22
Schnittstelle 23
Beispiel Inhaltsverzeichnis Schnittstellendokument 23
Was wird gebaut? – Anforderungen ermitteln (S188-192, Folien S19) 23
Workshops 23
Interviews 24
Beobachtung 24
Mitarbeit 24
Anforderungsvolatilität (Buch S149-150, Folien S20) 24
Stabile Anforderungen 25
Volatile Anforderungen 25
Systemdesign 25
Baseline und Änderungen an selbiger 25
Klassifikation von Anforderungen (S151-152, Folien S22) 25
Funktionale Anforderungen 26
Nichtfunktionale Anforderungen 26
Domänenanforderungen 26
Was sind gute Anforderungen? (Buch S153-154, Folie S23-27) 26
Überprüfen von Anforderungen 26
Anforderungs-Review: 26

Prototyping 26
Abnahmetestfallerstellung 26
Typische Probleme bei Anforderungen 26
Verständlichkeit 26
Selbstverständlichkeit 27
Wenig Präzision 27
Vermischung 27
Zusammenführung 27
Mehrdeutigkeit 27
Überflexibilität 27
Beispiele für Anforderungen 27
Input 27
Output 27
Qualitätsattribute von Anforderungen 28
Vollständig 28
Eindeutig definiert/abgegrenzt 28
Verständlich beschrieben 28
Atomar 28
Identifizierbar 28
Einheitlich dokumentiert 28
Notwendig 28
Nachprüfbar 28
Rück-und vorwärtsverfolgbar 28
Priorisiert 28
Anforderungsmodellierung (Buch S155) 28
Unified Modeling Language (UML) 29
Anforderungsanalyse (S163, Folien S32) 29
Nicht strukturierte Methoden 29
Semi-strukturierte Methoden 29
Strukturierte Methoden 29
Anwendungsfälle (UseCases) „Was aber nicht wie“ 29
Anwendungsfälle (Use Cases) nach Cokurn 2001 (Buch S164-165, Folie 33-34) 30
User Stories/Kunde vor Ort (S166-167, Folie S37) 31
Geschäftsprozessmodellierung insb. (S168-172, Folien S39) 31
Entwurf und Prototyping der Anwenderschnittstelle 32

Anforderungsdokumentation (S175-176, Folien S41-42) 32
Projektglossar (S177, Folie 43) 33
Anforderungsverwaltung – Anforderungsmanagement-Werkzeug (S179-180, Folien S44) 33
Anforderungsnachverfolgung - Traceability (S180-181, Folien S45-47) 34
Integration von unterschiedlichen Anforderungsquellen 34
Phase Entwurf – Wie wird technisch gebaut? 35
Von der Anforderung zur Architektur – Stellenwert und Aufgabe des Entwurfs (S203, Folie S4) 35
Ziele und Aufgabe des Entwurfs 35
Entwurfsparadigmen (S219, Folie S6) - Objektorientiertes Design (S219-222, Folie S5) 35
Objektorientiertes Design (S219-222, Folie S5) 35
Definition „Architektur“ (S199, Folie S10) 37
Lastenheft (Auftraggeber) / Pflichtenheft (Auftragnehmer) 37
Von der Anforderung zur Architektur (S203-204, Folien S12-13) 37
Strategischer Entwurf „Architektur“ 38
Taktischer Entwurf „Design“ 38
Aufbau einer Architektur – Beispiele (Folien S16-17) 39
Abstrakt 39
Real 39
Umsetzung von Anforderungen, Beispiele (Folien S 18) 40
Nicht-funktionale Anforderungen 40
Funktionale Anforderung 40
Architektur im agilen Kontext (Folien S19) 40
Emergent Design 40
Modellieren im agilen Kontext (Folie S20) 40
Der Entwurf im V-Modell (Folie S21) 41
Entwicklung der Softwarearchitektur (S204-207, Folien S23) 41
Faktoren zur Weiterentwicklung von Softwarearchitektur 41
Historische Softwarearchitekturen 41
Entwurfsparadigmen - Serviceorientiertes Design (S223-229, Folie S6, 25) 42
Service-Design-Paradigmen und Benefits (Buch S238-239, Folien S26) 43
Lose Kopplung / Loose Coupling 43
Abstraktion 43
Standardisierung 43
Composable 43
Business Process Modelling (BPM) und Service Oriented Architecture (SOA) (Folien S27-28) 43

Zusammenhang von Business Process Modelling (BPM) und einer SOA 44
Service-orientierte Architektur (Folien S30) 44
Architekturmuster (Folien S31-34 , Buch S232) 44
Designpatterns 44
Patterns allgemein 44
Architektur-Patterns 45
Implementierung, Integration & Testen 46
Programmiersprachen (Folie S6, S248) 46
Framework (S248-250, Folie S7-8) 46
Kriterien zur Auswahl eines Frameworks 46
Framework vs. Library (Folien S9, Buch S251) 46
Entwicklungsumgebung (IDE) (Buch S258, Folien S11) 47
Funktionsumfang einer IDE (S258-260, Folie S12) 47
Version Control System (Buch S 285-286, Folien S13-14) 47
Zentrales/Dezentrales Versionsmanagement (Buch S 287, Folien 15) 48
Zentrales Versionsmanagement 48
Dezentrales Versionsmanagement 48
Build Management System (Buch S284-285, Folien S15) 49
Kopplung, Kohäsion, Schnittstellen (S264-265, Folie S19-20) 49
Kopplung 49
Kohäsion 49
Interfaces 49
Information Hiding 49
Entwurfsmuster (Design Pattern) (Buch S252-264, Folien S21-22) 50
Erzeugermuster / Creational Patterns 50
Strukturmuster / Structural Patterns 50
Verhaltensmuster / Behavioral Patterns 50
Beispiel: Observer Pattern 50
Entwicklungsrichtlinien (Folien S23) 50
Dokumentation des Codes (Folien S24, Buch S273-274) 51
Testen 51
Automatisierte Tests (Folien S26, S267) 51
Test Driven Development (TDD) 52
Integration und Testen – Wie wird zusammengefügt / überprüft 53
Stellenwert des Softwaretests (Buch S299-301, VO6 S33) 53

Grundsätze des Softwaretests (S303-304, VO6 S45) 53
Ziele und Methodik des Softwaretestens (VO6 S36) 53
Ziele 53
Methodik 54
Test- und Integrationsstufen (Buch S305, VO7 S4) 54
Komponententest (S307-308, VO7 S6) 54
Ablauf des Tests 55
Integrationstests (S308-312, Folien S9-12) 56
Big-Bang-Integration 56
Horizontale Integration – Top-Down-Integration 56
Horizontale Integration – Bottom-Up-Integration 57
Vertikale Integration 57
Systemtest (S312-312, VO7 S13) 58
Akzeptanztest (Abnahmetest) (S313-314, VO7 S13) 58
Anwender-Abnahmetest 58
Betrieblicher Abnahmetest 58
Regulatorischer und vertraglicher Abnahmetest 58
Feldtest 58
Funktionale Softwaretests (S 315 -327, VO7 S15-19) 59
Funktionale Softwaretests - Strukturelle Abdeckung 59
Funktionale Softwaretests - Funktionale Abdeckung 59
Äquivalenzklassenanalyse 60
Zustandsbasierte Testmethoden 60
Klassifikationsbaummethode 61
Informelle Testmethode 61
Nichtfunktionale Softwaretests (S327-331, VO7 S20-23) 61
Testautomatisierung (S332-337, VO7 S21-23) 61
Automatisierte Komponententests 61
Automatisierte GUI-Tests 61
Regressionstests 62
Anwenderakzeptanz, Integration der Systemanwender 62
Migration von bestehenden Systemteilen/Daten 62
Inbetriebsetzung (S354-358, VO7 S30) 63
Best Practices 63
Anti-Patterns 63

Softwarewartung (S361-366, VO7 S32-35) 63
Definition Softwarewartung 63
Auslöser von Wartungstätigkeiten 63
Wartungstypen 64
Maßnahmen während der Wartung (Buch S365, VO7 S35) 64
Vorgehensmodelle 65
Grundlagen und Bedeutung von Vorgehensmodellen (Buch S371-373, VO8 S3) 65
Die drei Steuergrößen eines Projekts (VO8 S4) 65
Fundamentale Prozessmodelle (Buch S373-379, VO8 S6) 65
Traditionelles Modell: Wasserfallmodell nach Royce (1970) 66
Traditionelles Modell: Fundamentales V-Modell nach Böhm 67
V-Modell 97 – Entwicklungsphilosophie (S401, VO8 S12) 67
V-Modell XT („extreme Tailoring“) – 21.07.2015 v2.0 – Entwicklungsphilosophie (VO08 S12) 67
Traditionelles Modell: Spiralmodell nach Böhm (1988) 68
Vergleich Wasserfallmodell und iterative Entwicklung (S377-378, VO8 S15) 69
Beispiel iterative Entwicklung im Projekt LiSaMe (BS378, VO8 S16) 69
Industriell geprägte Prozessmodelle (S379-387, VO8 S17-20) 70
Rational Unified Process (RUP) 1998 70
RUP – Entwicklungsphilosophie 70
RUP – Tailoring / Anpassungsmöglichkeiten (S381-382, VO8 S19) 71
Agile Modelle – von Personen geprägte Prozessmodelle (S387-398, VO 8 S21-33) 72
Scrum (S387-390, Folien S22-25) 72
Scrum – Entwicklungsphilosophie 72
Scrum – Elemente 73
Scrum – Rollen 73
eXtreme Programming (XP) (Buch S391-398, VO8 S26-29) 73
eXtreme Programming – Prinzipien & Techniken 74
eXtreme Programming – Entwicklungsphilosophie 74
XP@Scrum 75
Kanban (VO8 S31-32) 75
Agile Versus traditionelle Vorgehensmodelle (Buch S409-412, VO8 S34-36) 76
Qualitätssicherung und Qualitätsmanagement 78
Abbildungsverzeichnis 82

Fallbeispiel 1: Stadt21

- Das Projekt21 stellt eine *eGovernment-Lösung* für kleine bis mittelgroße österreichische Städte dar.
- Wurde in enger Zusammenarbeit mit der Stadt Schwechat entwickelt und die Systeme und Erfahrungen lassen sich einfach auf andere Städte oder behördliche Einrichtungen anpassen.

Das Projekt Stadt21 (Buch S116):

- Stadtverwaltung benötigt eine einfache zu handhabende, aber dennoch funktional umfangreiche Software zur Unterstützung des täglichen Arbeit seiner Mitarbeiter
- Beim Analyse und Erstellungsprozess wurde in enger Zusammenarbeit mit der Stadtamtsdirektion erstellt
- Es wurden regelmäßig Prototypen zur Verfügung gestellt.
- Es konnte so immer ein qualitativ und quantitativ hochwertiges Feedback eingebaut werden und dies hatte zwei Auswirkungen:
  (1) Das Endprodukt ***entsprach den Vorstellungen des Kunden*** und führte zu einer ***hohen Akzeptanz***.
    - ✓ Das Produkt wurde schrittweise eingeführt und führt deshalb zu einer hohen Akzeptanz.
    - ✓ Es gab Benutzerschulungen, Vorstellungs- und Informationsveranstaltungen.
    - ✓ Eine plötzliche Einführung hätte zu einer Ablehnung des Produktes geführt
    - ✓ graduelle Einführung konnte die Erleichterung, die eine Verwendung der Applikation bringt, verdeutlichen

  (2) enge Zusammenarbeit führte zu einem ***starken „Wachstumseffekt" der Applikation***
    - ✓ Als „Hauptmodul" der Applikation wurde als erstes der „Elektronische Akt" (ELAK" entwickelt.
    - ✓ Es entstanden zwei weitere Module („Anliegenverwaltung" und die „Postverwaltung")
    - ✓ Waren nicht geplant, aber in der Testphase stellte sich heraus, dass diese einen Mehrwert bringen würden
- Der funktionale Umfang sowie die Benutzbarkeit war von großer Bedeutung bei der allgemeinen Designentscheidung
- Es wurde statt einer Desktop-Applikation eine Webapplikation erstellt
- Es wurde die Technologie ***Asynchronious Javascript and XML (AJAX)*** verwendet und somit wurden dynamische Elemente in die klassische Webbedienung integriert.
- Für den Prozesseditort wurde ***Java Web Start*** verwendet. Diese Technologie ist plattformunabhängig und konnte alle grafischen Möglichkeiten von Java unterstützen.

Allgemeiner Aufbau (Buch S 117-118, Folie S4, 12-16)

- Besteht aus verschiedenen Komponenten, die zusammen die Tätigkeit eines Stadtmitarbeiter auf elektronischen Weg unterstützen und erleichtern

ELAK

- Der übliche Ablauf eines Aktes, auch eines elektronischen Aktes (ELAKs) ist anhand eines Prozesses vorgegeben.
- Prozess gibt an, welche Schritte durchzuführen sind, um ein Produkt ausführen zu können (wird von der Stadt angeboten)

Produktkatalog der öffentlichen Verwaltung

- Basis der Use-Cases
- Abbildung in Geschäftsprozessen
- Basis der Optimierung (quantitativ)

Abbilden der Geschäftsprozesse

- Stellt die Grundlage für die eigentliche Arbeit eines Sachbearbeiters dar.
- Diese muss z.B. jedes Mal wenn ein neuer Antrag auf Bewilligung eines Bauvorhabens eingeht, einen neuen Akt starten, der auf dem zugehörigen Prozess basiert
- Muss die Möglichkeit haben, weitere Dokumente (die im Laufe eingehen) zu einem Akt hinzufügen zu können.
- Eine Erstellung neuer Dokumente zu diesem Akt sollte auch möglich sein. Die Erstellung eines Dokuments soll in Abhängigkeit des Ablaufes eines Aktes passieren.
- Es wird ein Werkzeug benötigt, das den Benutzer in der tägliche Abreit unterstützt und das Erstellen und Bearbeiten von Akten erlaubt
- Wer denkt bei seiner Arbeit an Prozesse?
- Conclusio: „Verstecken“ der Prozesse
- Jeder Vorgang / Arbeitsschritt wird vom System verteilt
- Ausnahmen.....
  - ✓ freie Prozessschritte
  - ✓ nicht standardisierte Abläufe
  - ✓ "kleine" Prozesse

Anliegen-Management

- Erlaubt es ein aufkommendes Anliegen, das keinem definierten Produkt entspricht, aufzunehmen und in kooperativer Art und Weise zu bearbeiten
- Anliegen können dezentral aufgenommen werden und dann an die zentralen Anliegestellen weitergeleitet werden.
- Werden von dort aus an die Facharbeiter geleitet oder direkt dort bearbeitet.
- Der Kern des Anliegenmoduls basiert auf dem normalen Aktenmodul.
- Anliegen ist ein einfacher Prozess, der mit einem speziellen User Interface versehen ist.
- interne wie externe Anliegen
- Nachvollziehbarkeit
- kooperative Vorbereitung von Antworten
- Stellungnahmen
- Integration Dritter
  - ✓ Amtsärzte
  - ✓ Sachverständige
  - ✓ Gutachter

Postverwaltung

- erlaubt es eingehende Poststücke (eingescannte Dokumente oder auch E-Mail) in der Applikation aufzunehmen.
- Von der zentralen Posteinlaufstelle werden diese an die zuständigen Abteilungen weitergeleitet.
- Jeder Mitarbeiter, für den das Poststück von Bedeutung ist, erhält eine Benachrichtigung und können sich dieses direkt ansehen, ohne das physische Dokument abholen zu müssen.
- Ist auch eine elektronische Archivierung der Post
- Formulare
- Integration in Akten, Anliegen, ...
- Integration Scan-Straße
- Erfassung, Verteilung, Nachvollziehbarkeit, Fristen
- Mehrere hundert Dokumente pro Tag

Administration

- administriert das ELAK-System und räumt den Benutzern die benötigten Berechtigungen ein.

Legacy

- immer ein wesentlicher Task in großen Projekten
- Abbildung der Hierarchie
- Berechtigungen
- Zugriffe
- Drittsysteme
- Jedes Projekt schafft eine Legacy –Vorbereitung der eigenen Evolution (Standards)

DMS

- neuestes Modul
- Face-Lift und echter Evolutionsschritt
- nicht im Buch abgedeckt (zum Drucktermin noch nicht umgesetzt)
- MoReq (Model Requirements for the Management of Electronic Records) compliant
- Reduktion der Legacy Problematik in der Zukunft

Zusammenfassung – Module in STADT21

- Prozesserstellung
- Elektronischer Akt (ELAK)
- Anliegenverwaltung / Bürgerbeteiligung (Anliegen)
- Postverwaltung
- Administration
- Bürgerbefragung
- Schnittstellen für Prozesse
- Integration (z.B. SSO)
- Amtssignatur
- Dokumentenmanagement
- Legacy
- DMS

Prozesserstellung (Buch S118-120)

- Jede Tätigkeit eines Stadtmitarbeiters kann als ein Teil eines Geschäftsprozesses identifiziert werden.
- Für eine sinnvolle elektronische Unterstützung ist es wichtig, dass die vorhandenen Prozesse bekannt sind, festgehalten werden und in der festgehaltenen Form auch gelebt werden.
- In der Software sollte es möglich sein, Abbildungen von komplexen Prozessen zu erlauben und andererseits von einem geschulten Personal der Fachabteilung abgebildet werden zu können.
- Prozesserstellung wurde durch einen Flussdiagramm ähnlichen Aufbau mit wenigen Elementen und einfacher Notation erreicht. Zusätzlich wurde eine Modellierungs-Software entwickelt, die diesen Aufbau erreicht

Aufbau eines Prozesses

- Jeder Prozess besteht aus einem Startknoten, einem Endknoten, sowie beliebig vielen weiteren Knoten. Diese Knoten können entweder Aktivitätsknoten (Aktivitäten), Entscheidungsknoten oder Subprozesse sein.
- Aufbau hängt vom Prozess ab, aber jeder Prozess folgt einem ähnlichen Schema
- Jeder Prozess beginnt beim Startknoten und endet beim Endknoten. Aufeinanderfolgende Knoten können in einem Subprozess zusammengefasst werden
- Jeder Knoten (außer Endknoten) besitz einen Nachfolgerknoten und so wird Ablauf des Prozesses dargestellt
- Falls Knoten zwei Nachfolger hat bedeutet dies einen Entscheidung oder eine Nebenläufigkeit in den Prozessen.
- Verbindung zwischen zwei Knoten kann benannt werden, bei einer Entscheidung ist dies verpflichtend.

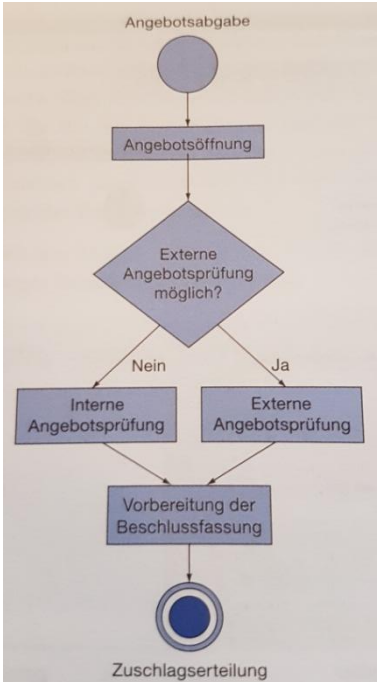


Abbildung 1: Beispielsprozess zur Prüfung und Erteilung von

Sonstige Anforderungen

Prozesseditor muss noch einige funktionale und nicht funktionale Anforderungen erfüllen. Überblick über die weiteren Anforderungen:

- **Visualisierung des Prozesses:** Benutzer soll einfach ein Bild des modellierten Prozesses bekommen. Prozess muss deshalb übersichtlich visualisiert werden.
- **Drafting:** Muss möglich sein, Prozesse auch zwischen zu speichern, ohne diese produktiv zu erstellen. z.B. auch Prozesse die nicht den Korrektheitskriterien entsprechen.
- **Überprüfen auf Korrektheit:** Muss überprüfen, ob ein Prozess den Kriterien entspricht.
- **Verwaltung der Prozesse:** Prozesse müssen leicht hinzugefügt, bearbeitet, gelöscht, umbenannt etc. werden können.
- **Benutzbarkeit:** Muss von einem geschulten Mitarbeiter bedient werden können. Nicht nur einem IT Experten.
- **Performance:** Reaktions- und Bearbeitungszeit muss im zumutbaren Rahmen liegen.
- **Sicherheit:** nur berechtigte Benutzer dürfen Prozesseditor benutzen und Prozesse verwalten.

Aktenbearbeitung (Buch S121-122)

- Darf nicht davon ausgehen, dass jeder Sachbearbeiter ein versierter PC-User ist, deshalb ist ***Usability*** in diesem Bereich sehr wichtig.
- ELAK-Modul zielt darauf ab, dem User möglichst viel Arbeit abzunehmen. Hintergrundwissen des zugrundeliegenden Prozesses sollte also auf ein Minimum reduziert werden.
- User wird Schritt für Schritt durch das Bearbeiten geführt. Muss sich immer nur auf die aktuelle Aufgabe konzentrieren und nicht auf den schrittweisen Ablauf.
- Es wird dem User auch ein Überblick über den gesamten Prozessablauf gegeben
- Hauptaufgabe des ELAK-Systems ist den modellierten Prozess in Form von einzelnen Akten abzuarbeiten
- Muss also möglich sein, einen neuen Prozess zu starten und dann ausgehend vom Startknoten den Prozess Schritt für Schritt abzuarbeiten.

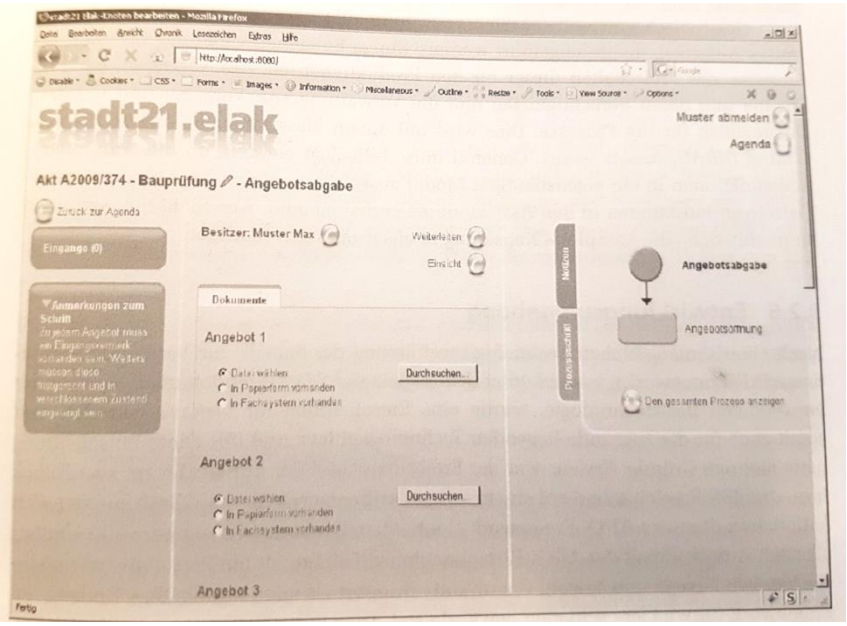


Abbildung 2: Interface der Aktenbearbeitung

- Sollte auch möglich sein, ***Notizen*** zu einem Akt hinzuzufügen, um zusätzliche Informationen anmerken zu können.
- Muss ***kooperatives Arbeiten*** unterstützen, da selten nur ein Sachbearbeiter an dem Akt arbeitet.
- Eine Weiterleitung und Informieren von Akten zu anderen Mitarbeitern wird unterstützt. Es gibt immer einen Besitzer/Hauptverantwortlichen für den Akt. Dieser kann beliebig viel weitere Bearbeiter haben.

Administration (Buch S122)

- Das ELAK-System hat eine hohe Anzahl an Benutzern.
- Meiste haben keine besondere Berechtigung, aber administrative Tätigkeiten an den Akten oder allgemeiner Natur müssen durchgeführt werden können
- Benutzer, die ***speziellere administrativer Aufgaben*** erfüllen müssen, sind etwas komplexer (z.B. Erstellen der Prozesse, Freischalten der Prozesse für den Produktivbetrieb, Verwalten der Mitarbeiter und Berechtigung und Verwaltung von Vorlagen und weiteren Dokumenten für die Prozesse)
- Wurde mit ***Role Based Access Control*** (RABC)-Ansatz gelöst.
- Generell muss festgelegt werden, ob die Administrationsfunktion in ein eigenständiges Modul ausgelagert wird, oder ob diese Funktionen im Basismodul bleiben. Aus Sicherheitsgründen empfiehlt sich eine Kapselung in einem eigenständigen Modul.

Entwicklungsumgebung und Technologien (Buch S122-123, Folie S9)

- ***AJAX-Framework***
- Man hat sich für ***Icefaces*** und somit auch für die zugrundeliegende Technologie ***Java*** und ***JSF***.
- Diese Entscheidung hatte verschiedenen Gründe:
  (1) Das Wissen der Programmierer war in diesem Bereich schon auf einem fortgeschrittenen Niveau. Mussten sich nur noch in das AJAX-Framework einarbeiten.
  (2) Umfang und Ausgereiftheit der AJAX-Komponenten schienen auf einem Produktivsystem ausreichendem Niveau zu sein. Community Support sowie regelmäßige Updates schienen gewährleistet zu sein
  (3) Icefaces und die zugrundeliegende Technologie ist Open Source verfügbar und eine komplexe Lizenzierung entfiel.
- Neben der Unabhängigkeit der Client-Plattform sollte auch die Server-Technologie weitgehend unabhängig bleiben
- Die Iceface Technologie verwendet ***J2EE*** als Basis. Diese konnte in vielen Sever-Technologien verwendet werden
- Für Datenbank-Zugriffe wurde ***Hibernate*** verwendet. Diese hat sich in vorhergehenden Projekten als zuverlässige ORM-Implementierung erwiesen. Diese Wahl war sowohl aus Performance- sowie Stabilitätssicht und der DBMS-Unabhängigkeit gut.

Abbildung 3: Verwendete Technologien

Spezifische Aspekte (Folien S5-6)

- Langlaufend (mehrere Jahre)
  - ✓ Erster Go-Live: nach 1 Jahr
  - ✓ Weiterentwicklung intensiv über 5 Jahre
  - ✓ natürliches Wachstum (bis heute)
- Für alle Geschäftsbereiche
  - ✓ Querliegend über alle Leistungen
  - ✓ Anforderungsgranularität sehr unterschiedlich
  - ✓ Projektmitwirkung unterschiedlich
- Benutzbarkeit
  - ✓ Verordnung (dienstrechtlich erforderlich)
  - ✓ Akzeptanz (durch Verordnung nicht lösbar)
- Öffentliche Verwaltung
  - ✓ Vorgehensmodelle (Pluralität)
  - ✓ Anforderungsmanagement
  - ✓ Innovationskraft (2006)
  - ✓ Testen und Abnahme
  - ✓ Verständnis für Projektprozesse meist gering ausgeprägt
  - ✓ kaufmännisches Sicht dominiert
  - ✓ Controlling Aspekte hoch
  - ✓ Risikomanagement gering
  - ✓ Externe Audits durch Rechnungshof

Projektteam (Folien S7)

- Gesamt ca. 50 Beteiligte
  - ✓ PM Team
  - ✓ Anforderungserhebung
  - ✓ Entwickler
  - ✓ Usability
  - ✓ Tester
  - ✓ Fachbereiche
  - ✓ Sponsor

- Kommunikation primär über Workshops
- Steuerung über Lenkungsausschüsse
- Vorgehen iterativ

Steuerungsausschuss (Folien S8)

- Zusammensetzung:
  - ✓ Sponsor
  - ✓ Auftragnehmer
  - ✓ PM Team
- Periodizität: alle 6 Wochen
- Aufgaben:
  - ✓ Globales Tracking (Kosten / Fortschritt)
  - ✓ Risikomanagement
  - ✓ Build / Buy Entscheidungen

Entwicklungsprozess und Vorgehen (Folie S17-19)

- Releases
  - ✓ Jährliche Hauptreleases
  - ✓ Zwischenreleases bei Bedarf
- Anforderungsmanagement
  - ✓ Workshops mit vielen Teilnehmern
  - ✓ Nur bedingt zu empfehlen bei Innovationsprojekten
  - ✓ Gut anwendbar bei Standardprodukten
  - ✓ Evolution im Rahmen des Projektes zu kleinen Gruppen
    - Vergleichbar SCRUM mit relativ langen Sprints
    - Evolution ist dem Projekterkenntnisgewinn zuzuschreiben
  - ✓ Usability Engineering als permanenter Begleitprozess
- Entwicklerteam
  - ✓ 3-10 Personen
  - ✓ Ca. 50/50 Frontend / Backend overall, inital primär Backend
  - ✓ Stabiles Team, daher nur individuelle Einarbeitungen als Einmaleffekte
- Testen
  - ✓ 1-2 Personen
  - ✓ Extensive Automatisierung
  - ✓ Vorbereitung der UATs
- Exkurs: Rolle des AG, Stellenwert des Projektes
- Releaseplanung
  - ✓ Laufende Anforderungserhebung
  - ✓ Scope / Features (Backlog)
  - ✓ Kategorisierung
  - ✓ Hinunterbrechen auf „Sprint“ (Anfangs jährliche Planung)
  - ✓ Entwicklung und Testen
  - ✓ Release für spezifische Nutzergruppe des Sprints
  - ✓ Durchführung weiterer Sprints …
  - ✓ Zusammenfassung der Einzelreleases zu Hauptrelease
  - ✓ Durchführung UAT für Hauptrelease

- ✓ Hauptrelease

Erkenntnisse (Folien S20-23)

- Wahl des Vorgehensmodells entscheidend für den Projektverlauf
  - ✓ EIN Vorgehensmodell ist zu wenig
  - ✓ Individuelles Abstimmen und Tailoring notwendig
  - ✓ Pers. Erkenntnis:
    - Standards sind gut, lernen Sie alle
    - verwenden Sie alle
    - aber nicht exklusiv
    - sondern immer gemeinsam
- Langlaufende Projekte
  - ✓ Vorgehen ändert sich
  - ✓ Prioritäten ändern sich
  - ✓ Technologie ändert sich
  - ✓ Personen ändern sich
  - ✓ Changemanagement als Schlüssel zum Erfolg
  - ✓ Risikomanagement notwendig
  - ✓ Die „eigene Legacy“
- Stakeholder
  - ✓ unterschiedliche Kenntnisse erschweren den Projektstart
    - Glättung bei langlaufenden Projekten
    - Aktive Rolle des PM zur Beschleunigung des Prozesses
  - ✓ Stakeholder treten oft erst später in Erscheinung
    - Bsp. Rechnungshof -Prüfung mitten im Projekt
    - man kann nicht alles verordnen
- Technik
  - ✓ ändert sich (immer)
  - ✓ Änderungsbereitschaft entscheidend für Erfolg
  - ✓ Optimierung
    - Trifft gerne dort, wo man es nicht erwartet (Bsp. Agenda)
    - Planung von Änderungszyklen (kann auch in der Produktphase kommen)
    - Serviceverträge

Fallbeispiel 2: HISS

- Entwicklung von modernen ***Softwaresystemen und –services für Hochschulen*** umfasst ein breites Spektrum von Geschäftsprozessen in vielen Bereichen
- Leistungsumfang umschließt nicht nur das ***Management von Kursen und Prüfungen*** zur Abwicklung von Lehrtätigkeiten, sondern deckt auch viele ***Verantwortungen für die Verwaltung*** ab:
  Abbildung von Daten und Prozessen der Bereiche
  - ✓ Liegenschafts- Management
  - ✓ Controlling
  - ✓ Personal-Management
  - ✓ Facility Management
  - ✓ oder des Zulassungsverfahrens.
- Unterstützendes System für wissenschaftliche Mitarbeit zur Abwicklung ihrer Forschungstätigkeiten.

Projektentstehung (Buch S131, Folien S3)

- Es fiel eine Entscheidung zur Neuentwicklung sämtlicher zentral an Hochschulen eingesetzten Applikation. Die Applikationen von früher konnten nur noch schwer erweitert werden und deshalb wurde eine neue entwickelt.
- Das neue Projekt trug den Namen HISS (Hochschul-Informations-Systeme und –Services).
- Das Ziel war es eine gemeinschaftliche Architektur bereitzustellen, die die Altsysteme schrittweise ablösen konnte.
- Die Wartung konnte sichergestellt werden und das Einpflegen neuer Dienste sowie die Anbindung benachbarter Systeme (z.B. VoIP, SAP, e-learning) wurde ermöglicht.
- 40 Jahre Universitäts-und Gesetzesgeschichte, abgebildet in bis zu 40 Jahre alten, inhomogenen und stark interdependenten Systemen
- Altes „Flickwerk“
- „Schnittstellen-Chaos“
- Mangelnde Systemische Trennung von Datenhaltung, Anwendungslogik und Präsentationsebene
- „Zu Tode gewartet“
- Nicht zukunftsfähig

Ziele und Umfang von HISS (S132-133, Folie S4-5)

- Es wurde eine Integration der betreffenden Einzelsysteme auf eine homogene und konsolidierte Plattform durchgeführt um so den vorhin genannten Kriterien zu entsprechen.
- Es ergaben sich **folgende Ziele**:
  - ✓ Entwicklung eines einheitlichen Informationssystems mit den Funktionen aller Einzel-Systeme, die zu Beginn des Projektes an der gesamten Hochschule im Einsatz waren.

- ✓ Aufbau eines einfachen und umfassenden ***relationalen Datenmodells*** durch Integration der einzelnen Applikationen zu einem Ganzen.
- ✓ Online-Recherche-Möglichkeiten in den Daten und benutzerkonfigurierbare Auswertungen
- ✓ Integrierte, einheitliche und einfach bedienbare Online-Benutzer-Dialogführung im Web.
- ✓ Leichtere Erweiterbarkeit der geplanten Plattform um weitere Applikationen (z.B. Customer Relationship Management, Facility-Management)
- ✓ Offene und standardisierte Schnittstellen in verschiedenen Technologien (XML, CSV, SOAP, etc.) zur Anbindung von zusätzlichen externen Systemen
- ✓ Sicherung der Prozess- und Datenqualität
- ✓ Schrittweise Ablöse der Altsysteme
- ✓ Langfristige Sicherstellung der adaptiven und perfektiven Wartbarkeit
- ✓ Flexible Plattform für neue Services
- ✓ Qualifizierter Docking-Partner für externe und interne Nachbarsysteme

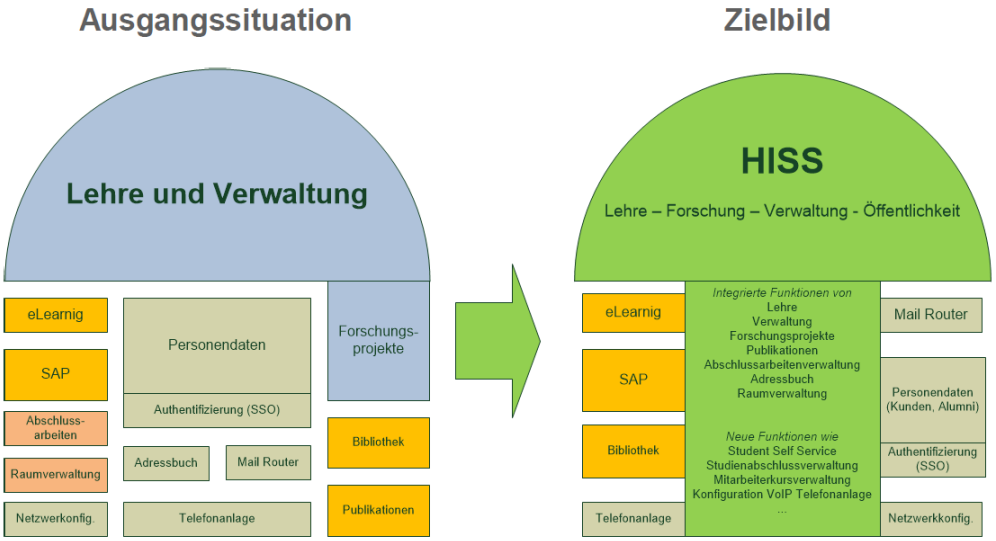


Abbildung 4: Ausgangs- und Zielschema von HISS

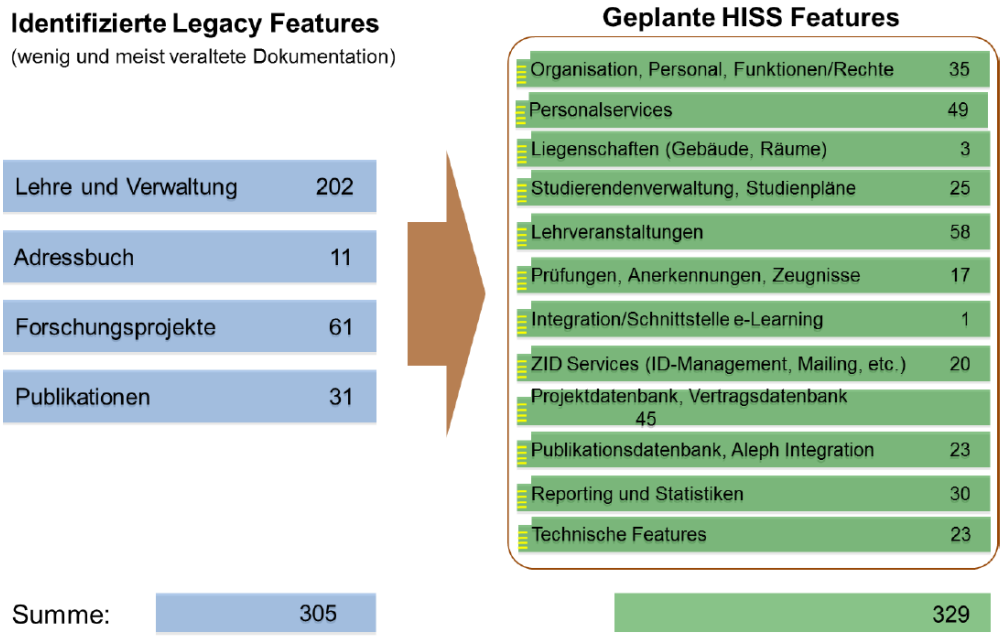


Abbildung 5: Featureliste (Initialplanung)

Phasen und grober Aufbau (S133-134)

Die Hiss Entwicklung gliedert sich in drei strategische Projektphasen:

Phase 1 (Vorbereitung der Migration)

- Vorbereitung der ***Migration*** mit dem Fokus auf der Erhaltung/Wiedergewinnung der formalen und informellen Ablaufinformationen und der Integration der verschiedenen Quellen.
- Diese Phase war ein ***iterativer Prozess*** zur Etablierung eines ***Fachkonzeptes***, der Technologiebasis und der Architektur

Phase 2 (Beginn der Migration)

- Beginn der Migration mit dem Fokus auf der Ablöse der ersten Altsysteme
- Integration der Felderfahrungen in die Entwicklung
- Anpassung des Fachkonzeptes aufgrund des gewonnen Feedbacks,
- dem Freeze der Technologiebasis und der Architektur sowie der Etablierung der Release-, und Schulungs- und Rollout-Prozesse

Phase 3 (Vollbetrieb und Reifung)

- Vollbetrieb und Reifung mit dem Fokus auf der Ablösung aller Altsysteme
- Etablierung des laufenden Release- und Change-Management-Prozesses
- Vorbereitung für den laufenden Betrieb und die Wartung.

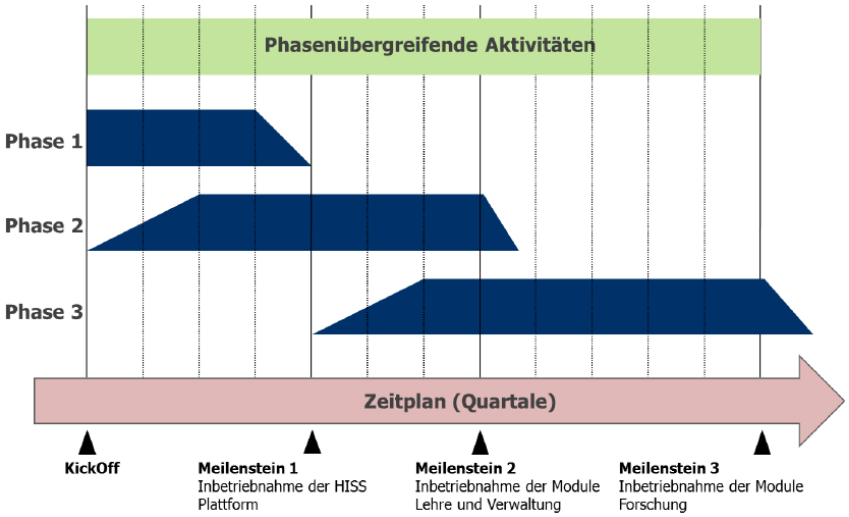


Abbildung 6: Übersicht der Projektphasen

Projektstruktur (Folien S9)

- Während dieser drei Phasen tagte in regelmäßigen Abständen ein ***Steering Committee*** als Projektlenkungsgremium

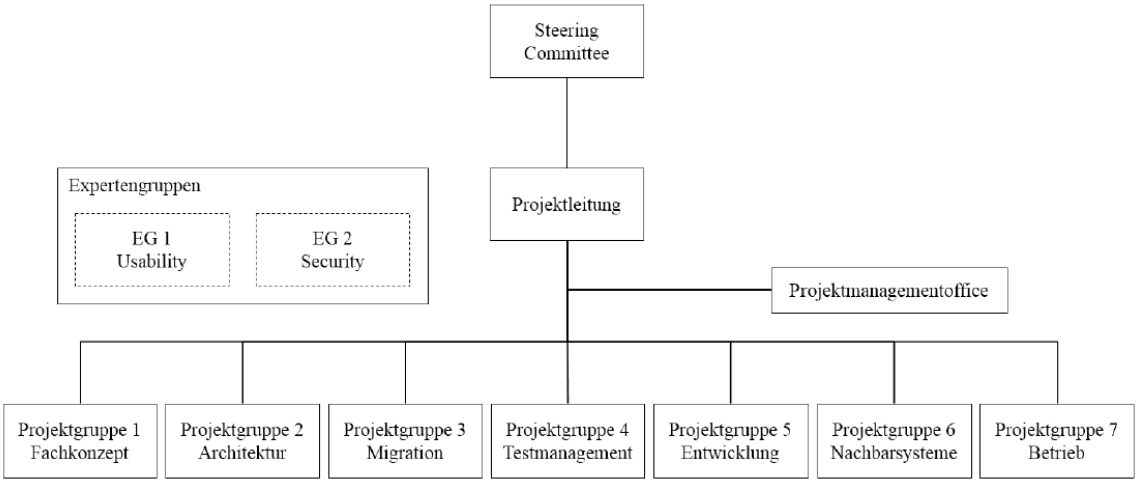


Abbildung 7: Struktur des Projektes aus organisatorischer Sicht

- **Steering Committe**
  - ✓ Steuerungsgremium
  - ✓ Vertreter aus Rektorat, Betriebsrat, Professorenschaft und Hochschülerschaft
  - ✓ tagt vierteljährlich

- **Expertengruppen**
  - ✓ Security und Usability Engineering als permanente Begleitprozesse

- **Projektteam**
  - ✓ Zwischen 10 und 32 Vollzeitmitarbeiter
  - ✓ Wöchentliche Jour fixes

Stakeholder (Folien S10)

- Alle am Projekt beteiligten Personen, die ein Interesse an dem jeweiligen Softwareprojekt haben
- Wichtige Anforderungsquelle
- Beeinflussen ein Softwareprojekt und werden selbst wiederum vom organisatorischen und politischen Umfeld beeinflusst

Stakeholder (Auszug) / Teilbereiche (Auszug)	Studierende	Vortragende	Studien-dekane	Studien-abteilung
Lehrveranstaltungsdaten, Teilnehmerverwaltung	x	x	x	
Prüfungen und Zeugnisse	x	x	x	x
Termin-, Raumverwaltung, Kalender	x			
Lehrunterlagen, Abgaben	x	x		
Lehrveranstaltungsevaluierung	x	x	x	
Studienplan, -fortschritt	x		x	x
Nachbarsysteme	x	x		x

Abbildung 8: Stakeholder

Motivation und Ziele des Fachkonzeptes (Buch S134-135)

- Es wurde versucht eine homogene, zukunftsfähige ***Systembasis*** zu schaffen.
- Eine bewusste und intensive Herangehensweise bei der Erstellung des Fachkonzeptes und der daraus gewonnen ***Anforderungsanalyse*** war der Schlüssel für die Entwicklung eines langlebigen Systems.
- Im Rahmen des Fachkonzepts wurden sämtliche Lösungen unter der Einbindung der ***Stakeholder*** evaluiert.
- Anforderungsanalyse geschah in einem ***iterativen Prozess***. Hier wurde großer Wert auf ***User - centered – Design*** gelegt.
- Vor der eigentlichen Implementierung wurde bereits mit ***Prototypen*** experimentiert (Benutzbarkeit optimieren, Probleme abfangen)
- ***Dokumentation der Abläufe und zugehörige Regelwerke***. Basierend auf diesen Dokumenten konnte entwickelt werden und der Änderungsaufwand konnte gering gehalten werden.

Vorgehensweise bei der fachlichen Analyse (Buch S 135-136)

- Zu den ***fachlichen Zielen*** zählt:
  - ✓ der Aufbau eines konsistenten und umfassenden Daten- und Service-Bestandes.
  - ✓ aus den bestehenden und neuen Anwendungen in den Bereichen Lehre, Verwaltung und Forschung und
  - ✓ die Implementierung von neuen Workflows.
- Für Umsetzung der ***fachlichen Analyse*** bis hin zum Rollout einer konkreten Implementierung wurde ein Prozess definiert, der im Folgenden kurz erklärt wird:
  - ✓ Die bestehende Features aus den Legacy Systemen wurden zusammen mit bereits bekannten neuen Features aggregiert (zusammengefasst). Abbildung 9 veranschaulicht diesen Prozess.

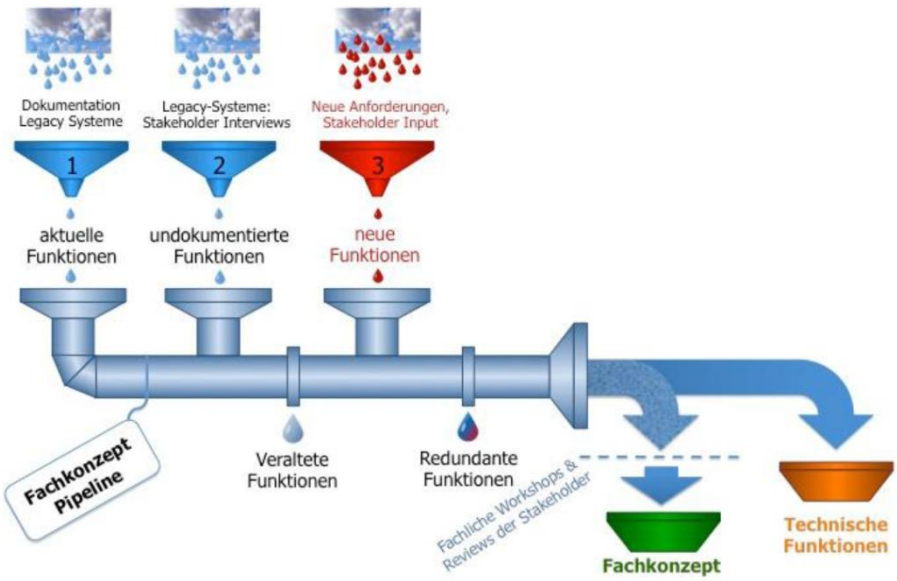


Abbildung 9: Fachkonzept Pipeline

  - ✓ Ein ***Feature*** ist ein fachlich abgeschlossener Teil, der eine greifbare Größe hat und in weniger als 5 Personentage umgesetzt werden kann. (z.B. Anmelden zu einer LVA)

- ✓ Neben den fachlichen gibt es auch ***technische Feature***, die als „Enabler" für die fachliche Logik dienen (Rollen- oder Rechensysteme, E-Mail-Newsletter, SAP-Schnittstellen)
- ✓ Features wurden in ***fachlichen Workshops*** auf Vollständigkeit der Anforderungen überprüft bzw. weiterentwickelt. Folge-Workshops, Interviews und hohe Einbeziehung der Hochschulmitarbeiter beschleunigten das Projekt.
- ✓ Aggregation der bestehenden Anforderungen und die gemeinsame Ausarbeitung von Lösungsvorschlägen mit den Angehörigen der Hochschule, waren ein integraler Bestandteil des Fachkonzeptprozesses.

Workshops (Folien S13)

- 50 Workshops 2008-2009 zu einer sehr breiten Themenpalette
  - ✓ Einbindung von Vertretern aller Nutzergruppen
  - ✓ Erhaltung und Wiedergewinnung der formalen und informellen Ablaufinformationen
- Die faktischen Ergebnisse der Workshops
  - ✓ Unterschiedlich gelebte Prozesse, selbst innerhalb einer Fachabteilung
  - ✓ Vielfach sehr unterschiedliche Vorstellungen bzgl. des SOLL Zustands
- Abwarten aller (mehrerer tausend) Entscheidungen im Abstrakten wäre enorm zeit-und kostenintensiv gewesen
- Bereitstellung der Systeminfrastruktur für eine iterative Entwicklung zum Realziel

Herausforderungen Altdatenbestand (Folien S14-15)

- Veraltete oder fehlende Dokumentation
  - ✓ 77% (346) der Tabellen und 84% (5616) der Spalten des abzulösenden Systems nicht dokumentiert
- Fehlende Normalisierung
  - ✓ Datenmodell spiegelte dateibasierte Datenhaltung (aus den 60er und 70er Jahren) wider
  - ✓ Keinerlei Fremdschlüssel-Definitionen
- Intransparentes Datenmodel
  - ✓ Legacy Tabellen im Produktivsystem, nicht erkennbar auf Basis der Bezeichnung
  - ✓ Tabellen mit unterschiedlichstem und nicht augenscheinlich erkennbarem Verwendungszweck (z.B. temporäre Tabellen für Batch-Prozesse)
- Unklare Datenhoheit
  - ✓ „Zerstreuung" der Daten über mehrere Systeme und Datenbanken
  - ✓ Diffuse organisatorische Datenverantwortlichkeiten
- Systeme, die auf korrupten Daten aufsetzen, erzeugen „Information" aus falschen Daten und sind damit unbrauchbar
  - ✓ Ausführliche Analyse des Altdatenbestands als Voraussetzung zur Identifikation der zu migrierenden Daten
    - 1) Analyse des IST Zustands
    - 2) Kategorisierung der Datenobjekte
    - 3) Zugriffs-und Traffic Analyse

    } Zentrale und umfassende (Re-) Dokumentation des Altdatenbestands
  - ✓ Abgleich aller Attribute aller zu migrierenden Datenobjekte gegen das Soll-Datenmodell

- ✓ Kompensation fehlender und fehlerhafter Attribute durch Entwicklung geeigneter Algorithmen zur Vervollständigung der inkonsistenten Datensätze
- ✓ Integritätsprüfungen zur Sicherstellung der Datenintegrität

Re-Dokumentation von Datenbanktabellen und –spalten (Folien S16)

- Verwendungszweck der Tabelle
- Spaltenname
- Datentyp
- Nullwerte
- Bedeutung der Spalte
- Indexe

Tabelle PR

alle Prüfungen seit 1968, Verrechnungssätze sind auf ORACLE.PRV

Tabellendefinition

Name	Typ	nullable	Bemerkung
UNIV	varchar2(1)	N	Universitätscode
MNR	number(22)	N	Matrikelnummer
LFD	number(22)	Y	Laufnummer der Datensätze (s. 1.)
PRART	varchar2(1)	Y	Art der Prüfung (s. 2.)
LV	varchar2(8)	N	Lehrveranstaltungsnummer oder Studienplanpunkt
DATUM	number(22)	N	Prüfungsdatum (JJJJMMTT)
ART	varchar2(1)	Y	Prüfungsart (s. 3.)
KNR1	varchar2(3)	N	Studienkennzahl 1
KNR2	varchar2(3)	Y	Studienkennzahl 1
MODUS	varchar2(1)	Y	Prüfungsmodus (s. 4.)
VERR	varchar2(1)	Y	Verrechnungskennzeichen
NOTE	varchar2(1)	Y	Note (s. 5.)
FRIST	number(22)	Y	Reprobationsfrist (s. 6.)
SEM	varchar2(5)	Y	Prüfungsemester (JJJJS)
PRUEFER	varchar2(10)	Y	Personalnummer des Prüfers
ZEUGNIS	number(22)	Y	Datum Zeugnis gedruckt (JJJJMMTT - geplant!)

Indexes

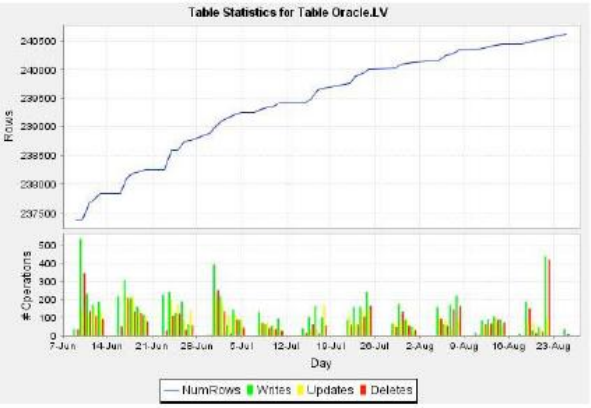
PRIND [UNIQUE]

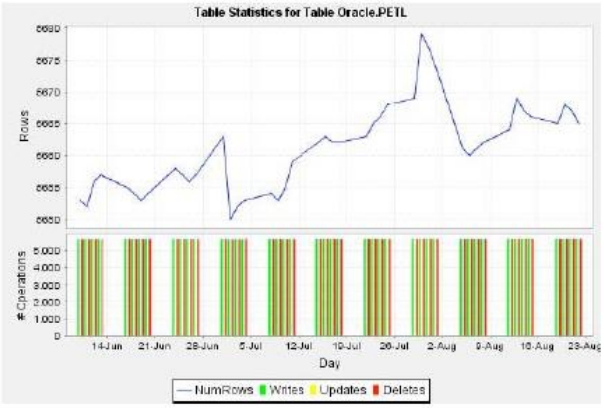
Name
MNR
LFD

Abbildung 10: Re-Dokumentation

Zugriffs-und Traffic Analyse (Folien S17)

- für sämtliche Tabellen wurden statistische Zugriffsdaten erhoben
- Tabellarische und grafische Auswertungen





numrows	writes	updates	deletes	lastwrite
4037991	32653	0	31678	15.07.2008 09:55
4037991	58170	0	56495	15.07.2008 09:55
4037991	75962	0	73901	15.07.2008 09:55

Abbildung 11: Zugriffs- und Traffic Analyse

Modularisierung (Folien S18-19)

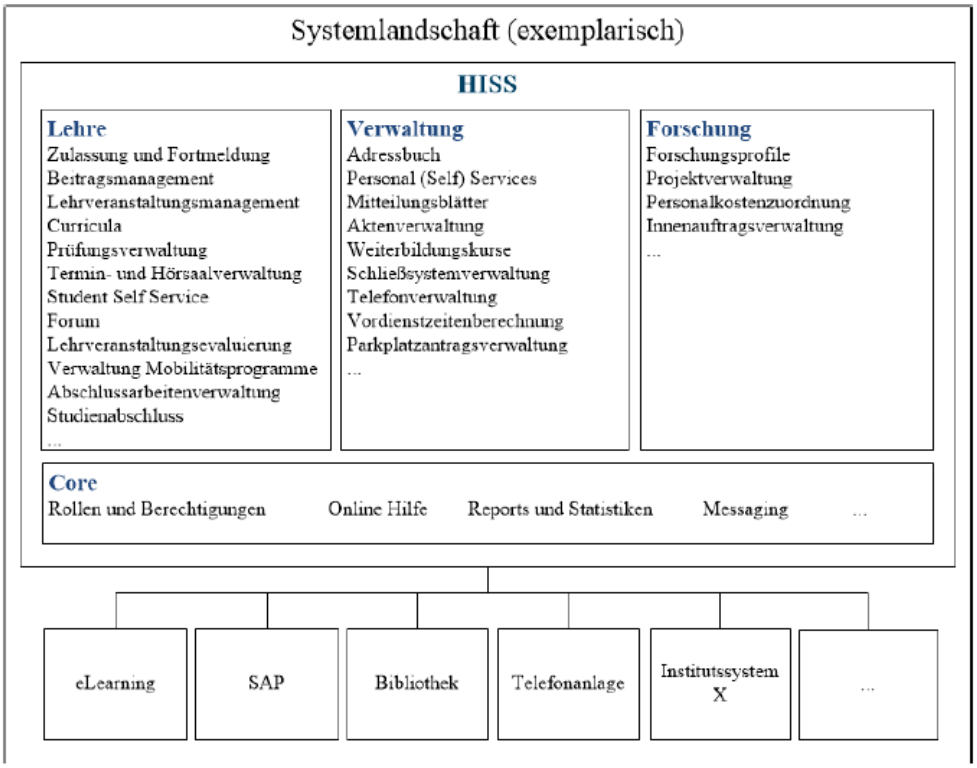


Abbildung 12: Modularisierung

- Fachliche Bereiche bleiben strikt getrennt
- Klare Datenhoheit bei überschaubaren Modulgrößen
  - ✓ Jedes Modul hält seine Daten in einer eigenen Datenbank
  - ✓ Fachlich zusammenhängende, aus wenigen Tabellen bestehende Datenbanken
- Unabhängige (Weiter-)Entwicklung einzelner Module
  - ✓ Unterstütz eine schrittweise Einführung
- Module sind einzeln läuffähig und compilierbar
  - ✓ Keine Beeinträchtigung des Gesamtsystems während Entwicklungs-und Testphasen
  - ✓ Erlaubt das Austauschen einzelner Module ohne das Gesamtsystem austauschen zu müssen

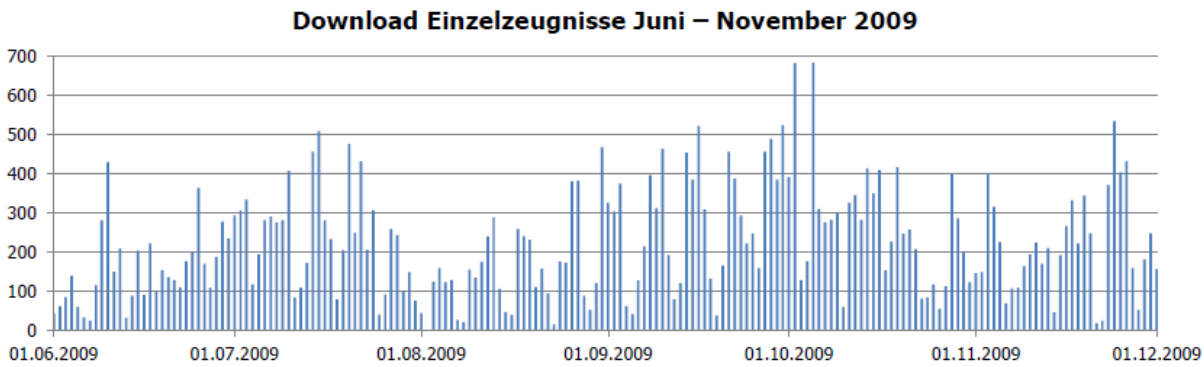
Student Self Service (Folien S20-21)

- Ablöse eines zeit-und kostenintensiven, manuellen Prozesses
  - ✓ Selbstdruck von Zeugnissen und Bestätigungen für Studierende, z.B.
    - Einzelzeugnisse
    - Sammelzeugnisse
    - Studienblätter
    - Studienbestätigungen
    - Erfolgsbestätigungen
    - Bestätigung gem. Familienlastenausgleichsgesetz
  - ✓ Entlastung der Studien-und Prüfungsabteilung um ~300.000 Dokumente pro Jahr
  - ✓ Postversand entfällt
  - ✓ Online Validierung mittels Barcode und URL durch Dritte möglich

- Zugriffsmöglichkeiten für Dritte individuell und pro Dokument konfigurierbar

Juni – November 2008:
- 60.000 Poststücke
- 10.000 Studierende in der Studien- und Prüfungsabteilung

Juni – November 2009:
- 0 Poststücke
- 62.000 Downloads



Releases (Folie S22)

- Inbetriebnahme erster Funktionalitäten initial geplant 1 Jahr nach Projektstart, tatsächlich 4 Monate früher erfolgt (Ruf nach Quick Wins und schnelle Sichtbarkeit von ersten Projektergebnissen)
- Iterative Ablöse bzw. Inbetriebnahme von (teils gänzlich neuen) Teilfunktionalitäten über 4 Jahre → intial identifiziertes Feature-Set von 329 Features wuchs innerhalb der ersten 2 Jahre Projektlaufzeit auf > 800!
- Wöchentlich (Zwischen-)Releases ab Inbetriebnahme eines Moduls

Phase Analyse – Was wird gebaut?

Beispiel Anforderungsqualität (Buch S141 – 142)

Dies ist ein Beispiel für widersprüchliche Anforderungen.
Die Verkehrszeichen wurden für eine Kurzparkzone von Montag bis Freitag mit einem Zusatzschild ergänzt.
Die maximale Parkdauer wurde auf 2 Stunden erhöht.
Diese bezieht sich auf alle Kurzparktage inkl. Samstag. Auf dem alten Schild ist jedoch eine Parkdauer von 1,5 Stunden angegeben.

Aus diesem Beispiel können folgende Analogien abgeleitet werden:

- Die beschriebenen Anforderungen sind zueinander widersprüchlich.
- Es existieren weitere „unsichtbare“ Vorgaben, z.B. Gesetzestexte, die in diesem Fall eine Relevanz haben.
- Die „richtige“ Interpretation (und damit die Auflösung des Widerspruches) ist nur mit Wissen über die weiteren Vorgaben möglich.
- Es existiert die Gefahr, dass unterschiedliche Interpretationen in den Projektphasen und von verschiedenen Projektbeteiligten getroffen werden.
- Die Interpretation wird oftmals nicht gesondert dokumentiert.

Definition Anforderung
Folgende Definitionen werden nach dem IEEE-Standard 610.12 (1990) gegeben:
1. „A condition or capability needed by a user to solve a problem or achieve an objective.“
2. „A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.“
3. „A documented representation of a condition or capability as in (1) or (2)“

Woher kommen Anforderungen? - Stakeholder und Anforderungsquellen (S143-144, Folien S6-9)

Stakeholder („Anteilhalter“)

- Sind alle am Projekt beteiligten Personen, die ein Interesse an dem jeweiligen Softwareprojekt haben.
- Stellen die wichtigste Anforderungsquelle dar.
- beeinflussen ein Projekt und werden selbst vom organisatorischen und politischen Umfeld beeinflusst (siehe Abbildung)

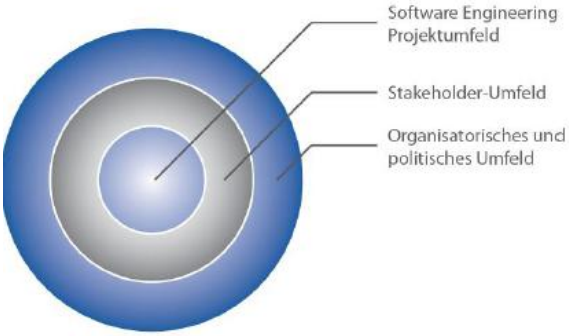


Abbildung 13: Projektumfeld

- Ein mittelgroßes Softwareprojekt hat nicht nur einen Stakeholder, sondern mehrere mit sehr unterschiedlichen Interessen
- Stakeholder ist jede Person/Institution, der/die ein Projekt beeinflussen kann oder die durch das Projekt beeinflusst wird

Stakeholder-Gruppe

- Hat eigene Sicht auf die Anforderungen.
- Die Sichten müssen durch Definition und Repräsentation der Anforderungen unterstützt werden
- Viele Stakeholder sind keine Techniker und verstehen den typischen Technikerlingo und technische Systembeschreibungen nicht. Anforderungen werden deshalb oft zweistufig definiert.
- Bei so vielen Interessen ist es wichtig, Anforderungen zu priorisieren und bei Konflikten den Projektleiter oder eine Projektsteuerungsgruppe mit dem Mandat auszustatten, die Konflikte aufzulösen.

Typische Stakeholder

- Kunde/Auftraggeber/Sponsor
- Geschäftsführung/Abteilungsleiter (Auftraggeber-und Auftragnehmerseite)
- Legacy Owner (Besitzer des Altsystems oder der Altdaten)
- Betreiber/Entwickler von Schnittstellensystemen (Systemen, mit denen integriert wird)
- Benutzer/Benutzervertreter (pro Benutzergruppe)
- Projektleiter
- Projektsteuergruppe
- Systemarchitekten/IT-Stratege
- Softwareentwickler
- Qualitätssicherung und Test
- Betrieb
- Wartung

Weitere wichtige Anforderungsquellen

- Standards und Normen
- Datenschutz
- Gesetze
- Projektumfeld
  - ✓ Anforderungen und Funktionsweisen von alternativen Systemen (z.B. Konkurrenzanbieter)
  - ✓ Unternehmenskultur
- Verdeckte Regeln
  - ✓ Glaubensgrundsätze/Tabus/Macht/Werte/Einstellungen/Status

In welchem Kontext wird gebaut? – Grundlagen der Analyse (S142-143, Folien S10-12)

Zielsetzung der Analyse

- Strategische Ziele
  - ✓ langfristige Unternehmensziele, z.B. Hebung des Unternehmensimages

- Operative Ziele
  - ✓ mittel-bis kurzfristige Ziele, z.B. Einhaltung der Projektkosten

- Führungsziele
  - ✓ lang-, mittel-, oder kurzfristige Ziele, die sich mit internen Abläufen und dem Faktor Mensch befassen, z.B. Steigerung der Mitarbeitermotivation im Unternehmen/im Projekt

Vision (üblicherweise stabil)

- Gibt an, welchen ***Zweck*** und welchen ***Nutzen*** das System primär erfüllen soll.
- Wird oft auch ***Business Case*** erstellt, um so die Wirtschaftlichkeit des Systems darzustellen
- Dient als Entscheidungsgrundlage, ob ein Softwareprojekt überhaupt beauftragt wird.
- **Beispiel:** Im Jahr 2020 werden infrastrukturelle Änderungen notwendig, die Anpassungen an einem end-of-life Softwareprodukt erfordern. Von dem neuen Produkt wird erwartet, dass die 30 Jahre fachliches Wissen der alten Lösung enthalten sind, sowie dass möglichst rasch die Lizenzkosten für das alte System eingespart werden können.

Scope (üblicherweise veränderbar)

- Definiert den Umfang, den das System abdecken soll
- Es ist erforderlich die Nicht-Ziele explizit zu dokumentieren, um einer Ausdehnung des Scopes entgegen zu wirken
- Durch Analyse mit vielen Stakeholdern, ist eine Ausdehnung der Projektlaufzeit wahrscheinlich
- Muss festgelegt werden, wer den Scope definiert (z.B. Projektsteuergruppe)
- Scope hat Auswirkungen auf die Projektkosten
- **Beispiel:** Eine Funktionsliste der existierenden Software; Eine Ansammlung von Use Cases, die umgesetzt werden müssen;
- **Achtung:** fachlicher Scope != Scope des gesamten Projekts (inkludiert z.B. Deployment, PM Overhead, QS, etc.)

Nachdem sich der Umfang im Projekt ändern kann, wird üblicherweise eine Projektsteuergruppe für die Freigabe der Änderung des Scopes eingerichtet → Aufgabe des Projektmanagement

Machbarkeit und Machbarkeitsstudie (Buch S146, Folien S13)

- Muss analysiert werden, ob das Vorhaben überhaupt umgesetz werden kann.
- Hier kommt aus auf den Inventionsanteil an

- Wenn ein Vorhaben technisch und/oder organisatorisch sehr neuartig ist, steigt Risiko der Nichtmachbarkeit.
- Oft wird eine Machbarkeitsstudio durchgeführt, diese enthält folgende Elemente:
  - ✓ Identifikation des Inventionsanteils
  - ✓ Identifikation von ähnlichen Systemen (z.B. auch von Konkurrenzanbietern)
  - ✓ Technische Modellierung
  - ✓ Technische Prototypen
  - ✓ Rechtliche Prüfung (z.B. landesweit, europaweit oder weltweit)
  - ✓ Akzeptanzanalyse bei den zukünftigen Benutzern
  - ✓ Identifizierung der wesentlichen Kostentreiber (z.B. nichtfunktionale Anforderungen)
  - ✓ Kostenanalyse

Make or Buy- Einflussfaktoren (Buch S146-148, Folien S14)

- In Softwareprojekten stellt sich die Frage, ob man die technischen Komponenten individuell entwickelt (***make***) oder ob z.B. ein COTS (Commercial Of The Shelf)-System zum Einsatz kommen kann (***buy***)
- Entscheidung muss oft für Teilkomponenten und Subsystemen getroffen werden.
- Folgende Aspekte müssen bei der „Make or Buy"-Entscheidung berücksichtigt werden:
  - ✓ Abdeckung des aktuellen und des antizipierten Projekt-Scope
  - ✓ Entwicklungskosten
  - ✓ Lizenzkosten (Anschaffung und laufender Betrieb)
  - ✓ Anpassungsmöglichkeiten und Anpassungsaufwand
  - ✓ Schulungsaufwand für Entwickler und Betreiber
  - ✓ Hardwareanforderungen
  - ✓ Support und Wartung
  - ✓ Nutzung von offenen oder proprietären Protokollen oder Formaten
  - ✓ Herstellerabhängigkeit (VendorLock-In)
  - ✓ Besitzverhältnisse
  - ✓ Weitere strategische Elemente, z.B. Weiterverkaufsmöglichkeit einer Neuentwicklung

Grobarchitektur (S148, Folien S15-16)

Grobarchitektur

- Die Grobarchitektur eines Softwaresystems hat einen wesentlichen Einfluss auf die Anforderungen und die Analyseprozesse.
- Es existieren folgende Zusammenhänge zwischen einer Grobarchitektur und der Analyse:
  - ✓ Architektur bestimmt wesentliche ***nichtfunktionale Eigenschaften*** des Systems und umgekehrt.
  - ✓ Hat einen Einfluss auf die Methode der ***Anforderungsanalyse und –beschreibung***.
  - ✓ Wenn die wesentlichen ***Systemkomponenten identifiziert*** sind, können diese auch in den Beschreibungen der Anforderung integriert werden.
  - ✓ Der ***Applikationstyp*** hat Auswirkungen auf die Anforderungsbeschreibung.
  - ✓ Die Architektur bestimmt die ***flexiblen oder festen Systemteile*** oder Funktionen bzw. das Erfordernis der Flexibilisierung wird aus der Analyse abgeleitet.

Schnittstelle

- Neben der groben inneren Systemarchitektur muss analysiert werden, mit welchen Systemen integriert wird und in welcher Form
- Es müssen Schnittstellen analysiert werden oder geg. vorhandenen Schnittstellen auf Tauglichkeit überprüft werden
- Folgende Aspekte sind für die Schnittstellenbeschreibung relevant:
  - ✓ Die meisten aktuellen Software (SW)-Systeme interoperieren mit anderen vorhandenen Systemen.
  - ✓ Viele Projekte haben einen höheren Integrationsaufwand (Gluing) als Implementierungsaufwand (Building).
  - ✓ Strukturiere Notationen (z.B. UML-Sequenzdiagramme) sind effektiv zur Beschreibung von Schnittstellen.
  - ✓ Eine exakte Schnittstellenbeschreibung ist notwendig für die rechtliche Abgrenzung von Systemteilen und Verantwortlichkeiten.
  - ✓ Schnittstelle = Bruchstelle: Schnittstellentestfälle (Szenarien über die Schnittstellengrenzen hinaus) werden von der Schnittstellenbeschreibung abgeleitet.
  - ✓ Das Management des Fehlverhaltens (Exceptionhandling) muss über die Schnittstellen hinaus spezifiziert werden.

Beispiel Inhaltsverzeichnis Schnittstellendokument

- Allgemein
  - ✓ Überblick
  - ✓ Anbindung der Komponente
  - ✓ Verwendete Protokolle
  - ✓ Methodenheader
  - ✓ Mengengerüst
  - ✓ Involvierte Business Entitäten
- Relevante Prozessschritte
  - ✓ Z.B.: Auflösen technische ID
  - ✓ Schnittstellenbeschreibung
  - ✓ Schnittstellensignatur
  - ✓ Fehlerhandling
- Datenbeschreibung
- Anhang
  - ✓ Beispiel Nachricht
  - ✓ Beispiel Fehlerhandling

Was wird gebaut? – Anforderungen ermitteln (S188-192, Folien S19)

Wenn die Ziele und die Stakeholder vorhanden sind, dann kann der Analytiker beginnen, die nötigen Anforderungen zu ermitteln. Dazu werden verschiedenen Methoden angewandt, die je nach Projekt, Zeit und Budget zum Einsatz kommen.

Workshops

- Ein Workshop ist eine Arbeitsgruppe, bei der sich der Analytiker mit mehreren Stakeholdern trifft um bestimmte Ergebnisse zu erarbeiten

- Ist eine der effektivsten Methoden zur Anforderungserhebung
- In kurzer Zeit wird intensiv an einer Konsensfindung erarbeitet
- Vorteile:
  - ✓ Alle Stakeholder werden einbezogen → schafft Stakeholder Buy-In
  - ✓ Erarbeitung eines Konsens, welche Eigenschaften ein zu entwickelndes System haben muss
  - ✓ Es können politische Aspekte identifiziert und gelöst werden, die den Projekterfolg gefährden würden.
  - ✓ Das Ergebnis ist eine Vorabversion der funktionalen oder nicht-funktionalen Anforderung

Interviews

- Hier trifft sich der Analytiker mit einem Vertreter einer Stakeholder-klasse, um dessen Wünsche und Anforderungen aufzunehmen
- Organisatorische Aufwand ist hier deutlich geringer als jener bei einem Workshop
- Ist weniger offen, aber strukturierter als ein Workshop
- Es sollten kontextfreie Fragen gestellt werden, um den Interviewten nicht zu beeinflussen z.B.:
  - ✓ Warum existiert dieses Problem?
  - ✓ Wie lösen Sie das Problem heute?
  - ✓ Wie würden Sie das Problem gerne lösen?

Beobachtung

- Ist vor allem bei der Erhebung der Anforderungen von Anwendern sinnvoll
- Diese Methode ist nur sinnvoll, wenn es bereits ein System gibt, welches durch ein neues System abgelöst wird.
- Analytiker besucht den Anwender in seiner Arbeitsumgebung und beobachtet ihn bei der Ausführung der Aufgabe
- Analyst versucht durch Befragung das Vorgehen des Anwenders zu verstehen und geg. auftretende Spezialfälle zu finden.

Mitarbeit

- Ist ähnlich der Beobachtung, nur arbeitet der Analyst wirklich mit dem Anwender mit.
- Wenn die Materie zu komplex ist, um sie durch Erklärungen darzustellen, oder Anwender & Analytiker keine gemeinsame Kommunikationsebene finden, dann ist diese Methode sinnvoll.

Bei allen Interaktionen mit Stakeholdern sollten Protokolle/ Mitschriften erstellt werden.

Anforderungsvolatilität (Buch S149-150, Folien S20)

- Es ist wichtig, auf Anforderungsänderungen reagieren und geg. auch noch zu einem späteren Zeitpunkt im Softwareprojekt Änderungen unter geringen Kosten durchführen zu können.
- Anforderungsdokumentation, das Systemdesign und alle weiteren abhängigen Artefakte müssen mit möglichst geringem Aufwand aktualisiert werden können. Um dabei strategisch vorzugehen, werden Anforderungen in zwei Klassen unterteilt:

Stabile Anforderungen

- Anforderungen, die sich (voraussichtlich) nicht oder nur geringfügig ändern werden.

Volatile Anforderungen

- Anforderungen, die sich (voraussichtlich) während der Anforderungsanalyse oder während des Betriebs des Systems ändern werden.

Die Einteilung in stabile und volatile Anforderungen hat eine direkte Auswirklung auf das Systemdesign

Systemdesign

- Stabile Anforderungen können mit simpleren Designs bzw. „hard-wired“ implementiert werden.
- Die Umsetzung volatiler Anforderungen erfordert ein adaptiveres Systemdesign („soft-wired“).
- Stabile Anforderungen können früher und detaillierter dokumentiert bzw. modelliert werden.
- Stabile Anforderungen können früher im Entwicklungsprozess implementiert werden.

Baseline und Änderungen an selbiger

- Trotz eines ständigen Änderungsprozesses ist es erforderlich, eine stabile Zwischensituation in der Analyse zu schaffen.
- Auf dieser Basis müssen weitere Aktivitäten (z.B. Systemdesign) aufbauen können.
- Es gibt also kein echtes Einfrieren (***Freeze***) der Anforderungen, aber zu einem definierten Zeitpunkt oftmals eine sogenannte ***Baseline***

> **Definition Baseline**
> „A specification or system that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development and can be changed only through formal change control procedures.“ –IEEE-Standard 610.12 (1990)

- Anforderungsänderungen werden vor der Baseline direkt und ohne komplexe Entscheidungsprozesse durchgeführt.
- Nach dem Setzern einer Baseline, dies kann in größeren Projekten auch in mehreren Iterationen passieren, werden formelle Entscheidungswege eingehalten (z.B. über eine Projektsteuerungsgruppe), weil diese Änderungen oft mit einer Änderung der Kosten (Re-Costing) verbunden sind.
- Je nach möglicher Auswirkung, wird eine ***Auswirkungsanalyse (Change-Impact-Analysis)*** einer neuen Anforderung oder Anforderungsänderung auf die Baseline durchgeführt.
- **Achtung:** Änderungen der Baseline sind Teil des Projektmanagements
  - ✓ Im Rahmen der agilen Softwareentwicklung werden ständig Trade-offs vereinbart – einen vernünftigen Change Request Prozess hierfür zu finden liegt im Rahmen des Projektmanagements bzw. des vertraglichen Rahmenwerks.

Klassifikation von Anforderungen (S151-152, Folien S22)

Wir unterscheiden hier zwischen nicht-funktionalen und funktionalen Anforderungen sowie zwischen fachlichen Benutzeranforderungen und technischen Systemanforderungen:

Funktionale Anforderungen

Beschreibung der Funktionen/Services, die das System zur Verfügung stellen soll. Dabei muss geklärt werden, wie das System auf bestimmte Eingaben und spezielle Situationen reagieren soll.

Nichtfunktionale Anforderungen

Anforderungen an die Funktionen/Services, die nicht deren Verhalten, sondern Vorgaben etwa an Qualitätseigenschaften (z.B. nach der ISO9126), den Entwicklungsprozess, einzuhaltende Standards und Normen und gesetzliche Rahmenbedingungen beschreiben.

Domänenanforderungen

Funktionale oder nichtfunktionale Anforderungen, die von der Domäne des Systems definiert werden und die besonderen Eigenschaften/Einschränkungen dieses Gebietes darstellen. Domänenanforderungen sind oft inhärent und nicht explizit pro Projekt dokumentiert.

Was sind gute Anforderungen? (Buch S153-154, Folie S23-27)

Gute Anforderungen sollen in erster Linie komplett und konsistent sein:

- **Vollständigkeit:** Alle Anforderungen sollen abgedeckt sein
- **Konsistenz:** Anforderungen enthalten keine Konflikte und keine Widersprüche.

Überprüfen von Anforderungen

Sind sehr kritische Artefakte, da das Risiko einer Fehlentwicklung zu hohen Kosten führen kann. Anforderungen sollen deshalb aktiv verifiziert werden. Dabei können folgende Methoden angewendet werden:

Anforderungs-Review:

- strukturierte und systematische Überprüfung der Qualitätseigenschaften von Anforderungen.
- Konflikte zwischen Stakeholdern gilt es zu erkennen und qualifiziert aufzulösen (z.B. mittels Priorisierung).
- Die Beteiligung unterschiedlicher Stakeholder (fachliche und technische) ist erforderlich.

Prototyping

- Dabei wird geprüft, ob die Anforderungen umsetzbar und insbesondere auch eindeutig und vollständig sind.

Abnahmetestfallerstellung

- Dadurch wird die Testbarkeit (Testability) von Anforderungen erreicht und eine Prüfung auf Eindeutigkeit und Vollständigkeit ermöglicht.

Typische Probleme bei Anforderungen

Verständlichkeit

- Anforderungen werden in der Sprache der Applikations-Domainerepräsentiert. Diese wird von Softwareingenieuren oft nicht verstanden.

Selbstverständlichkeit

- Domain-Spezialisten sind mit der Domain so vertraut, dass sie bestimmte Anforderungen als selbstverständlich erachten und nicht explizit darstellen und kommunizieren.

Wenig Präzision

- Es ist schwierig, Anforderungen präzise zu definieren, ohne diese schwer lesbar zu gestalten.

Vermischung

- Funktionale und nichtfunktionale Anforderungen werden häufig vermischt.

Zusammenführung

- Verschiedene Anforderungen werden gemeinsam beschrieben, ohne diese genau zu identifizieren und voneinander abzugrenzen.

Mehrdeutigkeit

- Spezifikationen in natürlicher Sprache sind oft mehrdeutig.

Überflexibilität

- Die Beschreibungsart der Anforderung (z.B. die natürliche Sprache) ist oft mehrdeutig interpretierbar und somit zu flexibel, um exakte Beschreibungen zu erzeugen.

Beispiele für Anforderungen

Input

- Ich will, dass das System immer funktioniert
- Wieso dauert das so lang bis ich auf den Knopf drücken kann?
- Jeder Anwender MUSS sich dem System gegenüber eindeutig identifizieren
- Als Anwender möchte ich eine Nachricht eingeben, die auf allen Anzeigen möglichst prominent angezeigt wird, damit ich die Gäste unsere Shopping Mall auf einen Brand hinweisen kann

Output

- Nicht-funktionale Anforderung: Das System MUSS eine Verfügbarkeit von 99,98% bei 24x7h Betriebszeit gewährleisten
- Nicht-funktionale Anforderung: Das System MUSS mindestens 90% der Requests innerhalb von 0,5s beantworten, bei 300 gleichzeitig aktiven Nutzern
- **Wurzelanforderung:** Jeder Anwender MUSS sich dem System gegenüber eindeutig identifizieren
  - ✓ Jeder Anwender MUSS einen eindeutigen Usernamen haben, mit dem er sich dem System gegenüber identifiziert
  - ✓ Jeder Anwender MUSS ein Passwort zusätzlich zu seinem Usernamen verwenden
  - ✓ Jedes Passwort MUSS mindestens 6 Zeichen lang sein und alpha-numerisch sein, sowie mindestens eine Zahl, sowie einen Groß-bzw. Kleinbuchstaben enthalten
- **User Story:** Weitere Analyse Schritte sind notwendig
  - ✓ Formulieren von Abnahmekriterien –z.B. Nachricht wird nach Anzeigeaktualisierung angezeigt, Nachricht entspricht abgenommenen Design
  - ✓ Abgestimmtes und abgenommenes Design mit Usern

Qualitätsattribute von Anforderungen

Um den Problemen von Anforderungen entgegenzuwirken, ist es notwendig diese nach folgenden Qualitäten zu entwickeln und zu verifizieren.

Vollständig

Alle Anforderungen müssen explizit beschrieben sein, es darf keine impliziten Annahmen eines Stakeholder über das zu entwickelnde System geben.

Eindeutig definiert/abgegrenzt

Präzise Definitionen helfen, Missverständnisse zwischen Stakeholdern(z.B. Entwickler und Auftraggeber) zu vermeiden.

Verständlich beschrieben

Dadurch können alle Stakeholder unter vertretbarem Aufwand die gesamte Anforderung lesen und verstehen.

Atomar

Es darf nur eine Anforderung pro Anforderungs-ID beschrieben sein.

Identifizierbar

Jede Anforderung muss eindeutig identifizierbar sein (z.B. über eine ID).

Einheitlich dokumentiert

Die Anforderungen und ihre Quellen sollten nicht in unterschiedlichen Dokumenten stehen oder unterschiedliche Strukturen haben.

Notwendig

Klären, ob Anforderungen unabdingbar sind.

Nachprüfbar

Die Anforderungen sollten mit Abnahmetestfällen verknüpft werden, damit bei der Abnahme auch geprüft werden kann, ob die Anforderungen erfüllt wurden.

Rück-und vorwärtsverfolgbar

Dadurch ist einerseits erkennbar, ob jede Anforderung vollständig erfüllt wurde, und andererseits ist für jede implementierte Funktionalität erkennbar, aus welcher Anforderung sie resultiert, so dass nichts Überflüssiges entwickelt wird.

Priorisiert

Die Priorität muss festgelegt werden, damit diese im Entwicklungsprozess berücksichtigt werden kann (z.B. durch die Entwicklungsreihenfolge) bzw. konkurrierende Anforderungen aufgelöst werden können.

Anforderungsmodellierung (Buch S155)

Im Rahmen der Analyse spielen Modellierungssprachen eine wesentliche Rolle. Unified Modeling Language (UML) ist eine der relevantesten zurzeit.

Unified Modeling Language (UML)

Eine sinnvolle Nutzung von UML bietet dem Projekt eine gute Dokumentationsbasis, mit der die verschiedenen Anforderungen der Stakeholder an die Sicht auf ein System erfüllt werden können. Die folgende Auflistung nennt alle Modelltypen, die nach UML 2.0 definiert sind:

- **Verhaltensmodelle**
  - ✓ Anwendungsfalldiagramm (Usecase Diagram)
  - ✓ Aktivitätsdiagramm (Activity Diagram)
  - ✓ Zeitverlaufdiagramm (Timing Diagram)
  - ✓ Zustandsdiagramm (Statechart Diagram)
  - ✓ Kommunikationsdiagramm (Communication Diagram)
  - ✓ Interaktion-Übersichts-Diagramm (Interaction Overview Diagram)
  - ✓ Sequenzdiagramm (Sequence Diagram)

- **Strukturmodelle**
  - ✓ Klassendiagramm (Class Diagram)
  - ✓ Paketdiagramm (Package Diagram)
  - ✓ Komponentendiagram (Component Diagram)
  - ✓ Verteilungsdiagramm (Deployment Diagram)
  - ✓ Kompositions-Struktur-Diagramm (Composite Structure Diagram)
  - ✓ Objektdiagramm (Object Diagram)

Anforderungsanalyse (S163, Folien S32)

Die Anforderungsanalyse ist der Prozess

- des Herleitens der von einem Stakeholder benötigten Funktionen/Services eines Systems und
- der Einschränkungen, unter denen das System entwickelt und betrieben wird.

Die Anforderungen selbst, sind die Ergebnisse die während der Anforderungsanalyse erzeugt werden. Um Anforderungen zu beschreiben und zu analysieren werden folgende Methoden verwendet:

Nicht strukturierte Methoden

Die natürliche Sprache ist die häufigste Methode für die Anforderungsanalyse.

Semi-strukturierte Methoden

Semi-strukturierte Methoden basieren im Kern auf der natürlichen Sprache, werden aber durch Vorlagen, ein eingeschränktes Vokabular und ergänzende Eigenschaften (Properties) vorgegeben.

Strukturierte Methoden

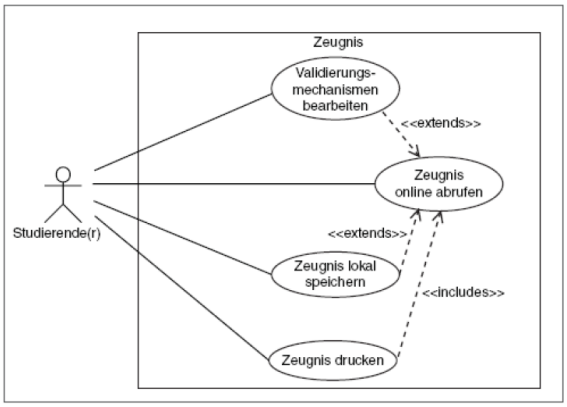
Strukturierte Methoden folgen einer definierten Syntax, Notation oder formalen Spezifikation.

Anwendungsfälle (UseCases) „Was aber nicht wie“

- Sine eine weit verbreitete Methode der Analyse
- Versteht man die Interaktion zwischen Nutzer und System mit der Absicht, ein fachliches Ziel des Nutzers zu verwirklichen. Ein Anwendungsfall repräsentiert die Fragestellung „was passiert“ und nicht „wie passiert etwas“.

Anwendungsfälle (Use Cases) nach Cokurn 2001 (Buch S164-165, Folie 33-34)

- **Name und Identifier:**
  - ✓ Benennung von Anwendungsfällen und geordnete Nummerierung
- **Beschreibung:**
  - ✓ Beschreibung des Anwendungsfalles
- **Beteiligte Akteure:**
  - ✓ Akteure sind involvierte Personen oder Systeme wie z.B. Benutzer, Kunde, Lieferanten oder ein System. Grundsätzlich werden zwei Arten von Akteuren unterschieden:
  - ✓ primäre Akteure, die eigentlichen Benutzer des Systems, und
  - ✓ sekundäre Akteure, die das System überwachen, warten und die primären Akteure bei der Zielerreichung unterstützen.
- **Status:**
  - ✓ Sagt aus, wie weit die Arbeit an dem Anwendungsfall fortgeschritten ist. Beispiele hierfür können sein: in Arbeit, bereit zum Review, im Review, abgelehnt und abgenommen.
- **Verwendete Anwendungsfälle:**
  - ✓ Wenn ein Anwendungsfall auf andere Anwendungsfälle zurückgreift, werden diese Fälle, mit Name und Identifikation, aufgezählt.
- **Auslöser:**
  - ✓ Gründe für das Ausführen eines Anwendungsfalles
- **Vorbedingungen**
  - ✓ Bedingungen, die erfüllt sein müssen, damit dieser Anwendungsfall ausgeführt werden kann
- **Invarianten**
  - ✓ Bedingungen, die durch den Anwendungsfall nicht verändert werden dürfen. Diese Bedingungen müssen in einem Misserfolgs-oder Fehlerszenario immer noch erfüllt werden.
- **Nachbedingung/Ergebnis**
  - ✓ der zu erwartende Zustand nach einem erfolgreichen Durchlauf des Anwendungsfalles.
- **Standardablauf**
  - ✓ Darstellung eines typischen Szenarios, das leicht zu verstehen oder der am häufigsten vorkommende Fall ist. Am Ende des Anwendungsfalles steht die Zielerreichung des Primärakteurs. Die Ablaufschritte werden nummeriert und häufig als Ablaufpläne dargestellt. Mittels der UML können diese Ablaufpläne in Aktivitätsdiagrammen oder Sequenzdiagrammen dargestellt werden.
- **Alternative Ablaufschritte**
  - ✓ Szenarien, die neben dem Standardablauf auch auftreten können, unabhängig davon, ob das Ziel erreicht wurde oder nicht. Sie werden häufig als bedingte Verzweigungen der normalen Ablaufschritte dargestellt.
- **Hinweise**
  - ✓ kurze Beschreibungen und Erklärungen zum besseren Verständnis des Anwendungsfalles, möglicherweise auftretende Nebeneffekte und alles andere, das nicht unter die bereits dargestellten Punkte fällt.
- **Änderungsgeschichte**
  - ✓ Versionierung des Anwendungsfalles mit Name des Autors, Datum etc.

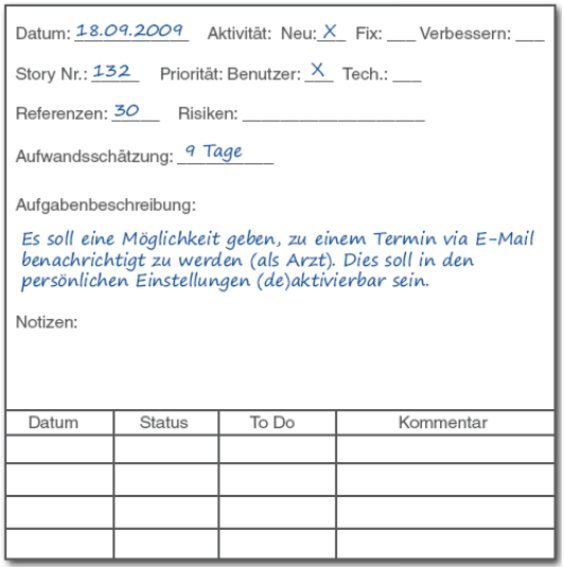


Name	Selbstausdruck Zeugnis
Beschreibung	Studierende können sich Prüfungszeugnisse online abrufen, selbst ausdrucken, lokal speichern und Validierungsmechanismen für Dritte bearbeiten.
Beteiligte Akteure	StudentIn
Status	Abgenommen
Verwendete Anwendungsfälle	Zeugnis online abrufen
Auslöser	StudentIn möchte sich ein Zeugnis ausdrucken.
Vorbedingungen	StudentIn hat eine Prüfung absolviert, die Prüfungsdaten wurden im System erfasst und StudentIn ist im System angemeldet.
Invarianten	Das Zeugnis muss im System gespeichert bleiben.
Ergebnis	Das Zeugnis wurde gedruckt.
Standardablauf	1. StudentIn loggt sich in HISS ein. 2. StudentIn ruft das gewünschte Zeugnis in HISS ab. 3. StudentIn bearbeitet Validierungsmechanismen nicht. 4. StudentIn öffnet das Zeugnis im Browser. 3. StudentIn druckt das Zeugnis.

Abbildung 14: Anwendungsfallbeschreibung & -diagramm

User Stories/Kunde vor Ort (S166-167, Folie S37)

- Eine User Story ist eine Beschreibung der Systemfunktionalität in wenigen Sätzen.
- Die Systemfunktionalitäten werden benannt, aber nicht ausführlich beschrieben.
- **Vorteile von User Stories:**
  - ✓ User Stories sind kurz und prägnant gehalten und damit gut wartbar.
  - ✓ User Stories fördern die Diskussion/Abstimmung zwischen Kunde und Entwickler während der gesamten Projektdauer.
  - ✓ User Stories reduzieren damit das Risiko von Fehlentwicklungen aus Benutzersicht.
  - ✓ User Stories funktionieren insbesondere in Kombination mit einer Metapher (gemeinsame Sicht auf das System).
- **Nachteile von User Stories:**
  - ✓ Ohne zugehörige präzisierte Akzeptanz-Testfälle sind User-Stories sehr stark interpretierbar und können ein Risiko für eine erfolgreiche Abnahme darstellen.
  - ✓ User Stories setzen eine starke Involvierung des Kunden/Benutzers während des gesamten Prozesses voraus. Dies ist jedoch oft unrealistisch.
  - ✓ User Stories (bzw. die erforderliche Kundeninvolvierung) sind personenzentriert.
  - ✓ User Stories können kaum als Vertragsgrundlage dienen.


Datum: 18.09.2009 Aktivität: Neu: X Fix: ___ Verbessern: ___

Story Nr.: 132 Priorität: Benutzer: X Tech.: ___

Referenzen: 30 Risiken: _____

Aufwandsschätzung: 9 Tage

Aufgabenbeschreibung:

Es soll eine Möglichkeit geben, zu einem Termin via E-Mail benachrichtigt zu werden (als Arzt). Dies soll in den persönlichen Einstellungen (de)aktivierbar sein.

Notizen:

Datum	Status	To Do	Kommentar

Abbildung 15: Beispiel User Card Story

Geschäftsprozessmodellierung insb. (S168-172, Folien S39)

- Business Processing Modeling Notation (BPMN)

Entwurf und Prototyping der Anwenderschnittstelle

Anforderungsdokumentation (S175-176, Folien S41-42)

- Ein Anforderungsdokument ist ein offizielles Dokument und Teil der Vertragsgrundlage für ein Softwareprojekt
- Unabhängig davon, welche Methoden zur Analyse und Modellierung der Anforderung angewandt werden, wird das Ergebnis in einem Dokument zusammengefasst.
- Für Anforderungsdokumentationen werden verschiedenen Namen in verschiedenen Kontexten verwendet:
  - ✓ Spezifikation
  - ✓ Anforderungsanalyse (-dokument)
  - ✓ Lastenheft/Pflichtenheft
  - ✓ Software-Requirement-Specification (SRS)

- Aufgrund des offiziellen Charakters einer Anforderungsbeschreibung, werden häufig die nach dem ***RFC2119 von Brandt (1997)*** definierten und immer großgeschriebenen Schlüsselwörter für das Anforderungslevel verwendet:
  - ✓ **MUSS (MUST):** absolute Festlegung einer Anforderung
  - ✓ **DARF NICHT (MUST NOT):** absoluter Ausschluss einer Anforderung
  - ✓ **SOLL (SHOULD):** Empfehlung für eine Anforderung. Abweichungen zu diesen Festlegungen sind in begründeten Fällen möglich.
  - ✓ **SOLL NICHT (SHOULD NOT):** Empfehlung, eine Anforderung auszuschließen. Abweichungen sind in begründeten Fällen möglich.
  - ✓ **KANN (MAY):** Die Anforderung ist optional.

- Alternativ beschreibt die ***IEEE-Standard 830*** eine im englischsprachigen Raum üblichen Struktur für eine Software Requirement Specification (SRS) wie folgt:
  - ✓ **Einleitung**
    - ➢ Zweck
    - ➢ Abgrenzung (Systemname, Allgemeine Systembeschreibung, Vorteile und Nutzen)
    - ➢ Definitionen, Akronyme und Abkürzungen
    - ➢ Referenzen
    - ➢ Überblick
  - ✓ **Beschreibung**
    - ➢ Produktperspektive: Schnittstellen des Systems, Benutzerschnittstellen, HWSchnittstellen
    - ➢ SW-Schnittstellen, Kommunikationsschnittstellen, Speicherumfang
    - ➢ Produktfunktionen
    - ➢ Nutzermerkmale
    - ➢ Rahmenbedingungen
    - ➢ Voraussetzungen und Abhängigkeiten
  - ✓ **Spezifische Anforderungen**
    - ➢ Externe Schnittstellen
    - ➢ Funktionen
    - ➢ Anforderungen an die Performanz
    - ➢ Logische Datenbankanforderungen
    - ➢ Designvorgaben

- ➢ Zu erfüllende Standards
- ➢ Systemattribute
  - Zuverlässigkeit
  - Verfügbarkeit
  - Sicherheit
  - Wartung und Pflege
  - Portierbarkeit

✓ **Anhang**

Projektglossar (S177, Folie 43)

- Die Identifikation und die Definition der wesentlichen Begriffe sind für die Analyse unbedingt erforderlich
- Hier soll sich eine in der Analyse gemeinsame Sprache entwickeln (auch wenn fachlicher oder technischer Hintergrund unterschiedlich)
- Glossar deckt fachliche und technische Begriffe ab. Es darf hier nicht zu einer Begriffsüberlagerung kommen

Anforderungsverwaltung – Anforderungsmanagement-Werkzeug (S179-180, Folien S44)

- Anforderungen werden entweder in Dokumentenform oder in spezialisierten Anforderungsmanagement-Werkzeugen verwaltet.
- Folgende Funktionen werden in der Regel in einem spezifischen Werkzeug zum Anforderungsmanagement abgedeckt:
  - ✓ Verwaltung der Anforderungen in einer Anforderungsdatenbank
  - ✓ Versionsmanagement der Anforderungen
  - ✓ Kollektives Editieren der Anforderungen
  - ✓ „Festklammern“ (Tagging) von Analyse-Baselines
  - ✓ Import und Export in Dokumentenform
  - ✓ Integration von Systemmodellen (z.B. UML-Modelle)
  - ✓ Selektive Notifikationen der Stakeholder bei Anforderungsänderungen
  - ✓ Nachverfolgbarkeit (Traceability) von Anforderungen; Darstellung einer Traceability-Matrix
  - ✓ Integration von weiteren relevanten Werkzeugen wie z.B. einem Testmanagement-Werkzeug
  - ✓ Verwaltung von Eigenschaften und Metadaten (z.B. die Priorität der Anforderung oder Aufwandsschätzungen)
  - ✓ Auswertungen der Anforderungen
  - ✓ Unterstützung für Anforderungs-Reviews und Kommentierungsverfahren
  - ✓ Geordnete Anforderungswiederverwendung
  - ✓ Verwaltung eines Projektglossars

Anforderungsnachverfolgung - Traceability (S180-181, Folien S45-47)

Definition Traceability
Nach dem IEEE-Standard 610.12 (1990) wird Traceability wie folgt definiert:
„The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; e.g., the degree to which the requirements and design of a given system element match."

- Die Sicherstellung der Nachverfolgbarkeit ist eine wesentliche Eigenschaft des Anforderungsmanagements.
- Dabei werden Zusammenhänge (Trace) zwischen verschiedenen Elementen entweder manuell oder automatisiert hergestellt.
- Es können folgende Typen der Nachverfolgbarkeit unterschieden werden:
  - ✓ **Nachverfolgbarkeit der Quelle:** Zusammenhang zum Stakeholder, der diese Anforderung eingebracht hat. „Wann, wie?"
  - ✓ **Nachverfolgbarkeit zu anderen Anforderungen:** Zusammenhänge zwischen interdependenten Anforderungen
  - ✓ **Nachverfolgbarkeit zu abhängigen Artefakten:** Zusammenhänge der Anforderungen z.B. zu Elementen des Systemdesigns oder auch die Nachverfolgbarkeit zu den Testfällen, die diese Anforderung abdecken

Integration von unterschiedlichen Anforderungsquellen

- **Use Cases vs. Requirements**
  - ✓ Methoden zur Anforderungserhebung sind komplementär einsetzbar
  - ✓ In formalen Anforderungslisten fehlt ein Narrativ
  - ✓ Use Cases sind schwer auf Vollständigkeit von formalen Anforderungen prüfbar
  - ✓ Mapping von Requirements zu Use Cases in Requirementsmanagement Werkzeug
  - ✓ N zu N Beziehung zwischen Requirements und Use Cases
- **Beispiel 1:** Kunde beauftragt ein Smartcard-Verwaltungssystem, das später sicherheitszertifiziert werden soll
  - ✓ Anforderungen des Kunden an den Workflow der Verwaltung in Form von Use Cases und Aktivitätsdiagrammen
  - ✓ Anforderungen der Sicherheitszertifizierung in Form von formalen Anforderungslisten
  - ✓ Abnahme des Gesamtsystems durch Abnahmetests und formale Reviews

- **User Stories vs. Use Cases**
  - ✓ Integration von User Stories und Use Cases auf zwei Arten
  - ✓ Use Cases verfeinern User Stories
  - ✓ User Stories verfeinern Use Cases
  - ✓ User Stories sind wesentlich informeller als Use Cases, benötigen also meistens Zusatzinformationen
- **Beispiel 2:** Pflichtenheft in Form von Use Cases, agile Vorgehensweise im Projekt gewünscht
  - ✓ Use Cases aus Pflichtenheft werden in Aktivitätsdiagrammen abgebildet
  - ✓ Aktivitätsdiagramme werden in 1 bis N User Stories (möglichst horizontal) geschnitten und umgesetzt

Phase Entwurf – Wie wird technisch gebaut?

Von der Anforderung zur Architektur – Stellenwert und Aufgabe des Entwurfs (S203, Folie S4)

Ziele und Aufgabe des Entwurfs

- Entwurf eines Systems, das die Anforderungen des Kunden optimal abdeckt
- Sorgt für Umsetzung der nicht-funktionalen und funktionalen Anforderungen
- Beschreibung des Systems auf technischer Ebene
- Ausgangspunkt für Implementierung
- Festlegung auf Architektur des Systems
  - ✓ Auf Bausteinebene
  - ✓ Auf Gesamtsystemebene
- Kommunikationsmittel zwischen den Projekt-Stakeholdern
- Aufteilung des Systems in funktional getrennte Blöcke
- Notwendig um Anforderungen an Skalierbarkeit und Wartbarkeit zu erfüllen

Entwurfsparadigmen (S219, Folie S6) - Objektorientiertes Design (S219-222, Folie S5)

Für den Entwurf einer Applikation gib es mehrere Ansätze. Je nach Anforderung kann es günstiger sein, ein System rein objektorientiert zu entwerfen oder aber auf eine serviceorientierte Architektur zu setzen.

Objektorientiertes Design (S219-222, Folie S5)

- Hier werden die Diagramme und Modelle der objektorientierten Analyse übernommen und erweiter.
- Ziel ist es, eine möglichst genaue Abbildung des später zu implementierenden (Sub-) Systems zu erstellen
- Müssen nicht alle Implementierungsdetails im Entwurf vorhanden sein da es z.B. Methoden geben kann, die für die konkrete Implementierung relevant werden und deshalb nicht im OO-Entwurf erscheinen.
- Verwendete ***Diagramme***:
  - ✓ Klassendiagramm – legen Struktur fest
  - ✓ Zustands-und Sequenzdiagramm – geben das Verhalten des Systems und Abläufe vor.
  - ✓ Komponentendiagramm – die die Systemkomponenten und ihren Zusammenhang abbilden.
- ***Notation*** ist meistens UML
- Klassendiagramm steht im Mittelpunkt
- ***Vorgehensweise nach Booch (1994)***:
  - ✓ Klassen und Objekte auf jeder Abstraktionsebene identifizieren
  - ✓ Semantiken zwischen Klassen und Objekten festlegen
  - ✓ Beziehungen der Klassen und Objekte identifizieren
  - ✓ Schnittstellen und Implementationen der Klassen spezifizieren
- Die hier angegeben Tasks finden sowohl in der Analyse als auch im Entwurf statt.
- Folgende zwei Punkte sollen im objektorientierten Entwurf erfüllt sein:

- ✓ Festlegen einer Architektur für die Implementierung (logische und physikalische Architektur)
- ✓ Policies festlegen für die Systemkomponenten

- ***Unified Modeling Language (UML)*** zur Unterstützung des objekt-orientierten Entwurfs
  - ✓ Modellierungssprache der OMG für komplexe und große Systeme
  - ✓ Basiert auf objekt-orientierter Entwicklung und Design
  - ✓ Sinnvolle Nutzung der UML → gute Dokumentationsbasis
- Zwei Arten von Modellen
  - ✓ ***Verhaltensmodelle*** – beschreiben dynamisches Verhalten, z.B. Aktivitätsdiagramme, Zustandsdiagramme, Sequenzdiagramme
  - ✓ ***Strukturmodelle*** – beschreiben statische Teile des Systems, z.B. Klassendiagramm, Komponentendiagramm, Verteilungsdiagramm

- Beispiel für ***Zustandsdiagramm***
  - ✓ Beschreibt mögliche Systemzustände und bildet Zustandsänderungen sowie auslösende Ereignisse ab
  - ✓ Abbildung in Form eines endlichen Zustandsautomaten

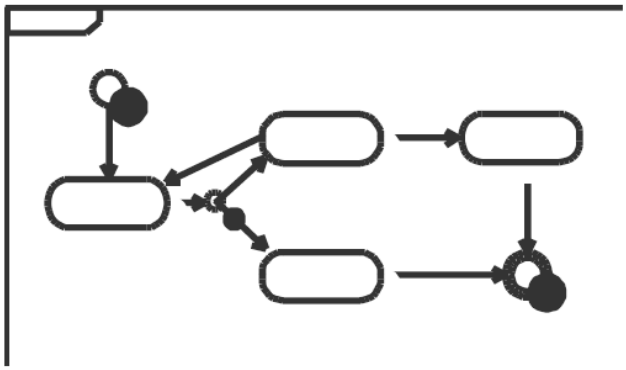


Abbildung 16: Entwurfsparadigmen - Zustandsdiagramm

- Kontrollfluss am Beispiel ***Stair vs. Fork***
  Im Entwurf sollte festgelegt werden, wie die Kontrolle (Steuerungslogik) über die vorhandenen Objekte aufgeteilt wird. Es gibt hier zwei Möglichkeiten der Kontrolle:
  - ✓ **<u>Fork</u>**
    - ➢ Ein zentrales Objekt übernimmt Kontrolle
    - ➢ Der gesamte Ablauf wird von einem zentralen Objekt gesteuert.
    - ➢ Hier wird die Wartbarkeit durch die zentrale Steuerung erhöht.
  - ✓ **<u>Stair</u>**
    - ➢ Hier wird die Kontrolle auf die beteiligten Objekte aufgeteilt.
    - ➢ Kapselung in den Schritten
    - ➢ Jeder Schritt des Ablaufs wird von einem eigenen Objekt gesteuert
    - ➢ Hier wird sehr viel Funktionalität in den jeweiligen Schritten gekapselt, was unter Umständen zu höherer Wiederverwendung führt

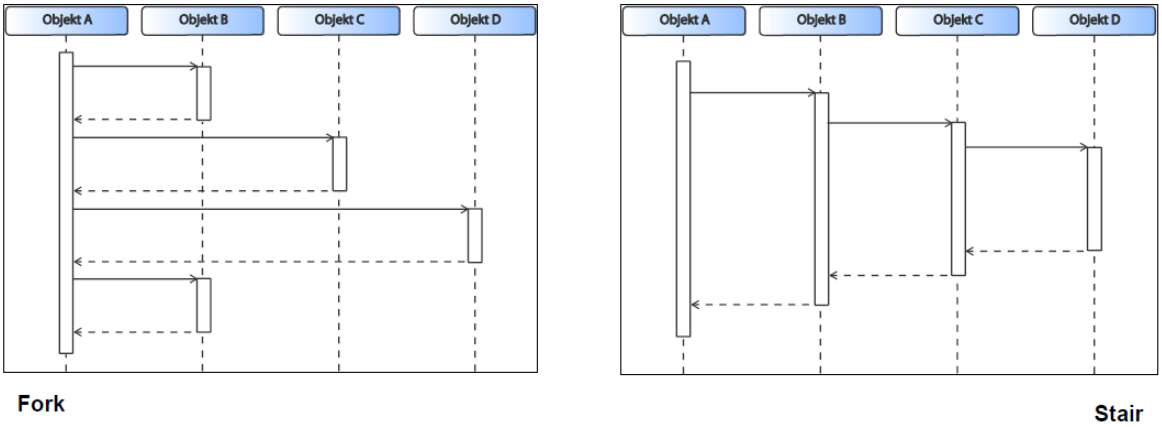


Abbildung 17: Fork und Stair

Definition „Architektur“ (S199, Folie S10)

- Eine Architektur ist eine Beschreibung von System-Strukturen(Datenströme, Modulare Strukturen, Prozess Strukturen usw.) [Bass et al. (1998)]
- Eine Architektur ist das erste Artefakt zur Analyse der Einhaltung von Qualitätseigenschaften [Bass et al. (1998)]
- Eine Architektur ist eine Beschreibung der Beziehung von Komponenten und Verbindungen [Bass et al. (1998)]
- Eine Architektur ist die fundamentale Organisation eines Systems, verkörpert durch seine Komponenten, deren Beziehungen und deren Umwelt, sowie die Prinzipien hinter dem Design und der Evolution des Systems [IEEE Standard1471 (2000)]

Lastenheft (Auftraggeber) / Pflichtenheft (Auftragnehmer)

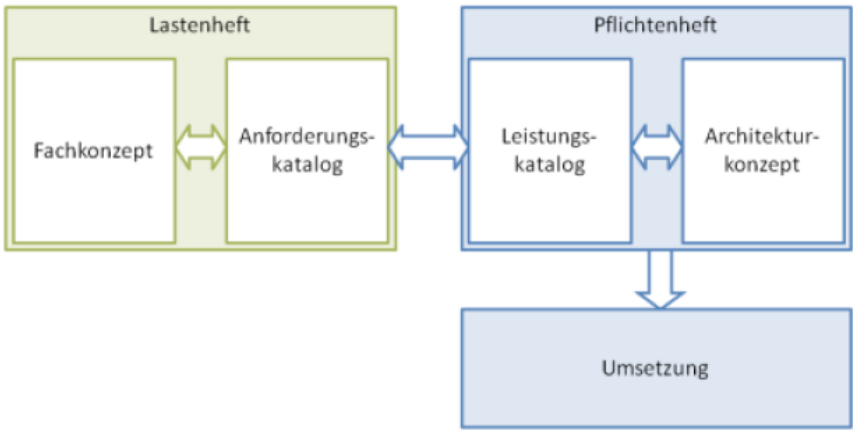


Abbildung 18: Lastenheft & Pflichtenheft

Von der Anforderung zur Architektur (S203-204, Folien S12-13)

- Das Hauptziel des Softwareentwurfs ist ein System zu entwerfen, das die Anforderungen des Kunden optimal beschreibt
- Generell gibt es zwei Arten von Anforderungen – funktionale und nichtfunktionale Anforderungen (NFA).
- NFA beschreiben die Qualitätskriterien eines Systems und somit ist die Architektur der Hauptträger der NFAs und dazu gehören

- ✓ Performanz
- ✓ Veränderbarkeit
- ✓ Sicherheit

- ***Ziel des Entwurfs:*** Abbildung eines Systems, das Anforderungen optimal beschreibt
- Zwei Arten von Anforderungen (funktional und nicht-funktional)
  - ✓ Nicht-funktionale Anforderungen (z.B. Zuverlässigkeit, Benutzbarkeit, etc.) sind Faktoren hinter strategischem Entwurf
  - ✓ Funktionale Anforderungen (z.B. Verarbeitung von Eingaben, Erstellenvon Ausgaben, etc.) sind Faktoren hinter taktischem Entwurf
- Voraussetzungen um Anforderungen in der Architektur umzusetzen zu können
  - ✓ Erfahrung des Softwarearchitekten
  - ✓ Unterstützung durch Auftraggeber

Strategischer Entwurf „Architektur“

- Nichtfunktionale Anforderungen (z.B. Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, etc.) sind die treibende Kraft hinter dem strategischen Entwurf eines Softwaresystems.
- Architektur im „Großen“
- Es werden hier Entscheidungen bezüglich der Zielplattform, der Art der Anwendung (Webapplikation, Rich-Client-Anwendung, etc.) und der verwendeten Technologie getroffen
- Ist ein Entwurf auf Systemebene

Taktischer Entwurf „Design“

- Architektur im „Kleinen“
- funktionellen Anforderungen sind die treibende Kraft für den taktischen Entwurf
- Entwurf im klassischen Software Engineering Sinne (auf Bausteinebene, nicht auf Systemebene)
- Entscheidungen über Aufbau der einzelnen Bausteine (z.B. Entwurf der Klassen, usw.)

Aufbau einer Architektur – Beispiele (Folien S16-17)

Abstrakt

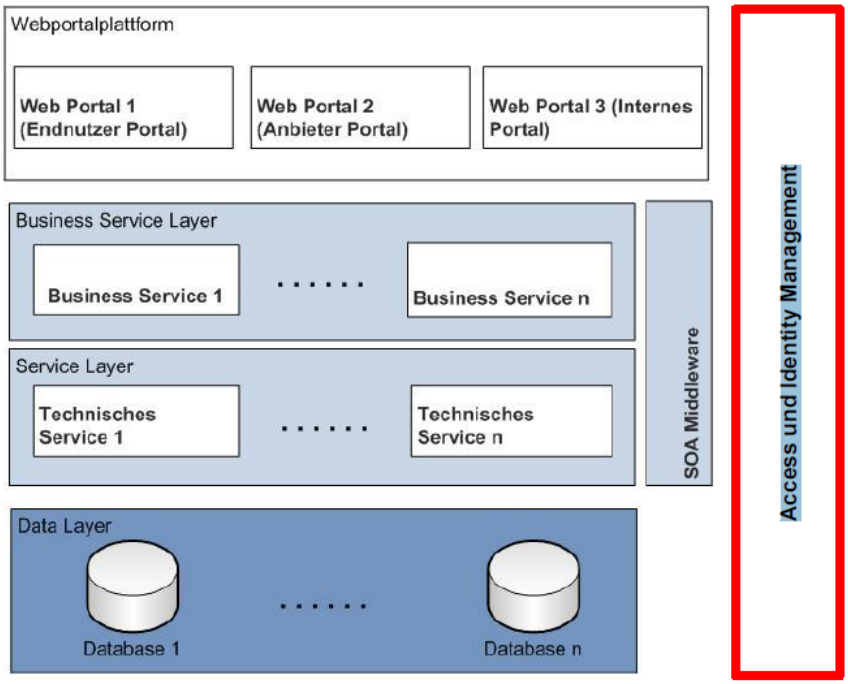


Abbildung 19: Architektur abstrakt

Real

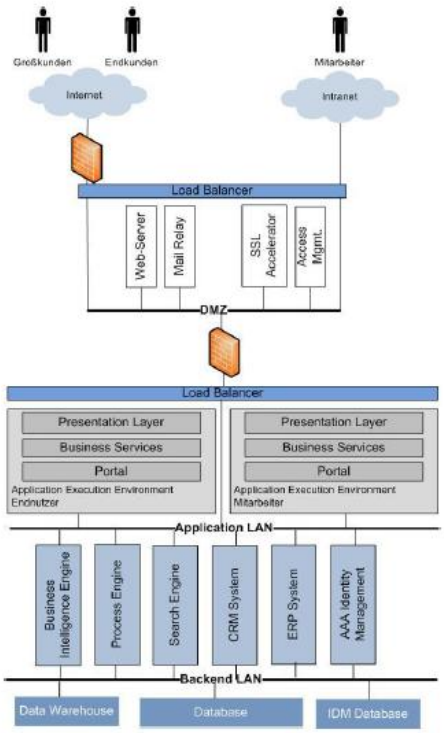


Abbildung 20: Architektur real mit Infrastruktur

Umsetzung von Anforderungen, Beispiele (Folien S 18)

Nicht-funktionale Anforderungen

- Wartbarkeit
- Umsetzung: Trennung des Systems in Schichten

Funktionale Anforderung

- „Dokumentation durchführen“
  - ✓ Eine Dokumentation wird inital erstellt.
  - ✓ Nachdem die Dokumentation erstellt wurde, befindet sie sich im Status „Entwurf“.
  - ✓ In diesem Status ist die Erstellung und Modifikation von Formularen möglich. Sobald die Formulare fertig entworfen wurden, wird die Dokumentation in den Zustand „Durchführung“ gestellt, bei dem es möglich ist Patienten anzulegen und Daten zu den Formularen einzugeben.
  - ✓ Es ist aber jederzeit möglich, den Zustand wieder in den Status „Entwurf“ überzuführen, um nachträgliche Änderungen oder Ergänzungen an den Formularen durchzuführen.
  - ✓ Wurde die Dateneingabe abgeschlossen, so wird die Dokumentation in den Zustand „Analyse“ übergeführt.
  - ✓ In diesem Zustand ist lediglich die Auswertung (=Export) der Daten möglich. Weder die Formulare noch eingegebene Daten können dann geändert werden.
  - ✓ Auch dieser Zustand ist aber nicht endgültig, d.h. man kann die Dokumentation wieder auf „Durchführung“ setzen, um weitere Daten einzugeben.
- Umsetzung: Zustandsdiagramm erstellen

Architektur im agilen Kontext (Folien S19)

Emergent Design

- Basiert auf Refactoring von erstellten, lauffähigen Komponenten
- Zuerst Deliverables (Best Practices!) → Dann entsteht Design
- Beispiele für notwendige Best Practices:
  - ✓ Code Conventions
  - ✓ Code Reviews
- High Quality Design ergibt sich aus 4 einfachen Regeln:
  - ✓ Test everything;
  - ✓ eliminate duplication;
  - ✓ express all ideas;
  - ✓ minimize entities

- Rahmenbedingungen trotzdem notwendig
  - ✓ Benachbarte Systeme, existierende Systemlandschaft
  - ✓ strategischer Rahmen muss definiert sein

Modellieren im agilen Kontext (Folie S20)

„If software development were model building, then the valid measure of the quality of the software or of the development process would be the quality of the models (fidelity, completeness)“ –Alistair Cockburn on Software Modeling

- Probleme von Design upfront
  - ✓ Volatile (oder neue) Anforderungen

- ✓ Nicht jedes Detail kann definiert/bedacht werden
- Barely good enough
- Is Design Dead?
- YAGNI

Der Entwurf im V-Modell (Folie S21)

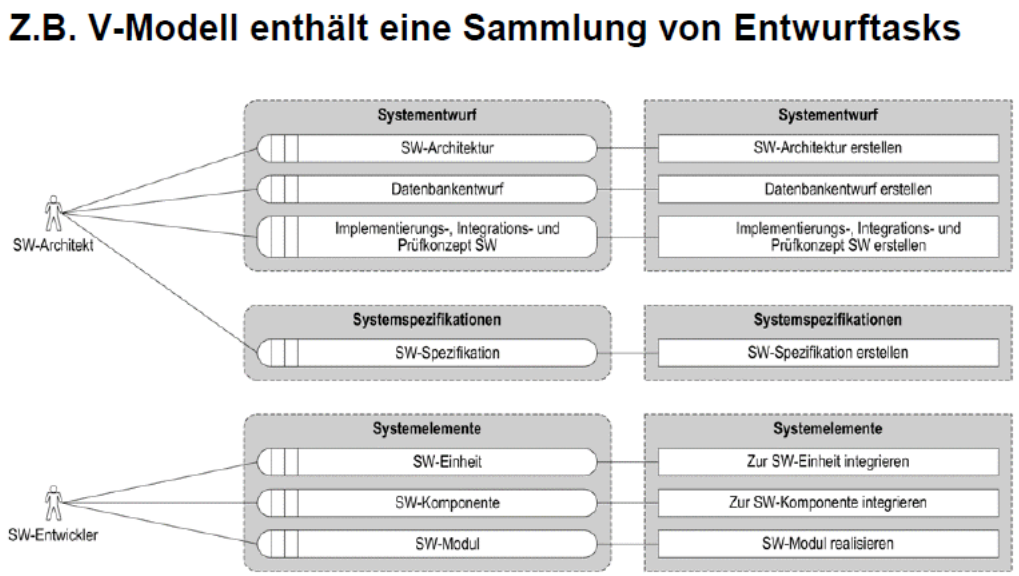


Abbildung 21: Entwurf im V-Modell

Entwicklung der Softwarearchitektur (S204-207, Folien S23)

Faktoren zur Weiterentwicklung von Softwarearchitektur

- Von Prozedural zu objektorientierten Programmiersprachen
- Vom Großcomputer zum Heimcomputer
- Durch objektorientierte Programmiersprachen Trend von monolithischen Applikationen, die sehr schwer erweiterbar waren, zu modularen Systemen
- In den 80er, 90er und frühen 2000er Jahren gab es hauptsächlich Rich-Client-Applikationen (die Geschäftslogik und Funktionalität der Anwendung wird in eine Desktop-Applikation verlagert)
- Trend zu Webapplikationen. Diese hatten eine ähnliche Funktionalität wie Rich-Client-Applikation nur konnte alles über einen Webbrowser ablaufen.
- Self Containt Systems (SCS, Microservices) und Service Oriented Architecture (SOA)

Historische Softwarearchitekturen

Tier-Architekturen sind Schichten-Architekturen. Sie teilen Applikation in funktional abgegrenzte Schichten ein.

- **<u>2-Tier Architekturen</u>**
  - ✓ Auf der einen Seite die Anwendung, mit der der Benutzer arbeitet und sich um die eigentliche Verarbeitung der Daten kümmert
  - ✓ und auf der anderen Seite die Backend-Komponente. Hier handelt es sich im Standardfall um eine Datenbank.
  - ✓ ***Vorteil:*** Daten sind hier zentral verfügbar, die von mehreren Benutzern verwendet werden können

- ✓ ***Nachteil:*** erweiterte Maßnahmen zur Sicherstellung der Datenkonsistenz sind notwendig.

- **<u>3-Tier Architekturen</u>**
  - ✓ Im Unterschied zu 2-Tier gibt es hier noch eine weitere Schicht
  - ✓ Hier wird die Geschäftslogik aus der Anwendungsschicht extrahiert und ähnlich wie die Datenbank zentral verfügbar gemacht
  - ✓ ***Vorteil:*** konsistente Datenzugriffe und eine bessere Wartbarkeit
  - ✓ ***Nachteil:*** längere und detaillierte Planungsphase

- **<u>Service-orientierte Architekturen</u>**

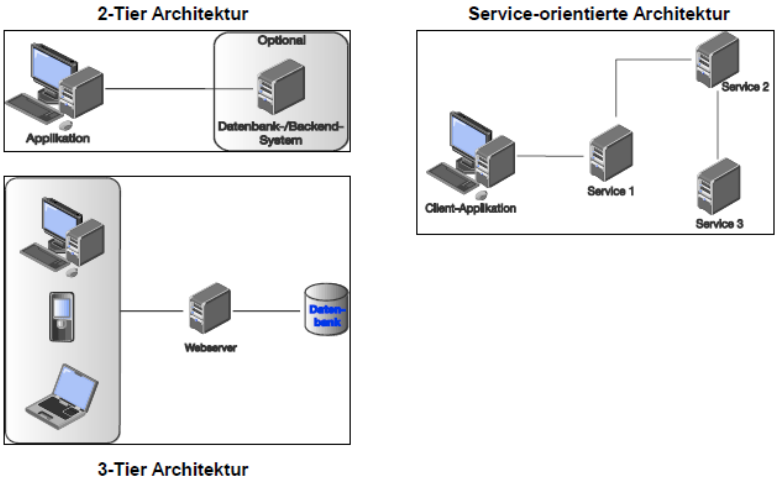


Abbildung 22: 2-/3-Tier und Service-oriented Architektur

Entwurfsparadigmen - Serviceorientiertes Design (S223-229, Folie S6, 25)

- Bei serviceorientierten Designs bedient man sich gleich mehreren Notationen, weil diese mehrere Abstraktionsebenen umfassen.
- Der Wechsel von fachlicher Ansicht zur technischen Realisierung geschieht hier fließend.
- Bei einem serviceorientierten Entwurf werden die Anwendungen erstellt z.B. mit Hilfe eines objektorientierten Entwurfs, miteinander verknüpft zu einer „großen“ Anwendung. Diese kleinen Anwendungen, die zu einer großen verknüpft werden, heißen Services
- die großen Services werden Business Services genannt.
- Eine Applikation wird in mehrere Business Services aufgeteilt, diese wiederum bis hin zu atomaren Services.
- Nachfolgende Abbildung zeigt den Aufbau eines serviceorientierten Entwurfs.
- Umspannt mehrere Abstraktionsebenen (Service Layer, Component Layer, Class Layer)

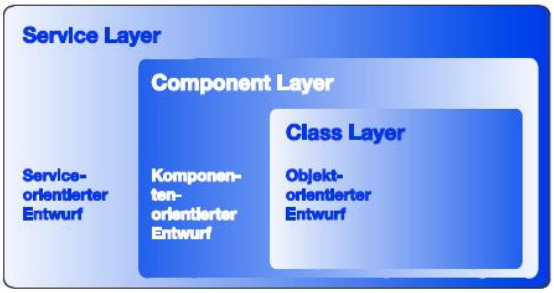


Abbildung 23: Serviceorientierter Entwurf

- Unterschiedliche Notationen je Layer
  - ✓ Z.B. Webservice (Service Layer), Komponentendiagramme (Component Layer), UML (Class Layer)

Service-Design-Paradigmen und Benefits (Buch S238-239, Folien S26)

Lose Kopplung / Loose Coupling

- Ist auf die größte Flexibilität ausgerichtet
- Es sollen möglichst generische Schnittstellen geschaffen werden, um den Zugriff auf den Service möglichst einfach und flexibel zu halten
- Die Wartbarkeit der Services wird dadurch erhöht und das benötigte Wissen über diese wird verringert.
- Die Integration eines atomaren Service wird reduziert.

Abstraktion

- Das Einsatzfeld wird dadurch erhöht
- Ein abstrakter Service kann aus verschiedenen Kontexten heraus aufgerufen werden
- Durch Abstrahierungen werden Neuentwicklungen reduziert
- Abstraktion ist nicht immer passend, da manche Services konkret sein müssen, um Zweck zu erfüllen. Wenn diese abstrahiert würde, würde man jede Funktionalität entfernen.

Standardisierung

- Die Zusammenarbeit von Services und die Zusammenfügbarkeit der Services gewährleistet.
- Vorteil dieser Technik ist, dass eine erhöhte Reusability der Services und eine leichte Integrierbarkeit der Services bestehen.

Composable

- Technik ermöglicht es, standardisierte Services auf niedrigen Schichten
- oder es spezialisierte Services auf höheren Ebenen zu Hierarchien oder Komponenten zusammenzuführen.

Business Process Modelling (BPM) und Service Oriented Architecture (SOA) (Folien S27-28)

- Im Service-orientierten Entwurf wird ein Schritt in einem Geschäftsprozess extrahiert und gekapselt → dabei handelt es sich um ein sog. Business Service.
- Anmerkung: Business Services können auch Geschäftsziele oder KPI (Key Performance Indicator) realisieren und müssen keinen Schritt im Geschäftsprozess darstellen –z.B. Kostenreduktion einen Report zu generieren kann auch dazu dienen ein Business Service zu definieren
- Service-orientierte Architekturen sind business driven, d.h. sie sind an Geschäftsprozesse angelehnt
- Business Process Modelling (BPM) und SOA sind eng miteinander verknüpft – Anwendungen werden maßgeblich von der Geschäftsprozess-Modellierung beeinflusst

Zusammenhang von Business Process Modelling (BPM) und einer SOA

- Geschäftsprozess A und B greifen auf das gleiche Business Service „Update Report“ zu
- „Update Report“ wird in Form von zwei technischen Services gelöst

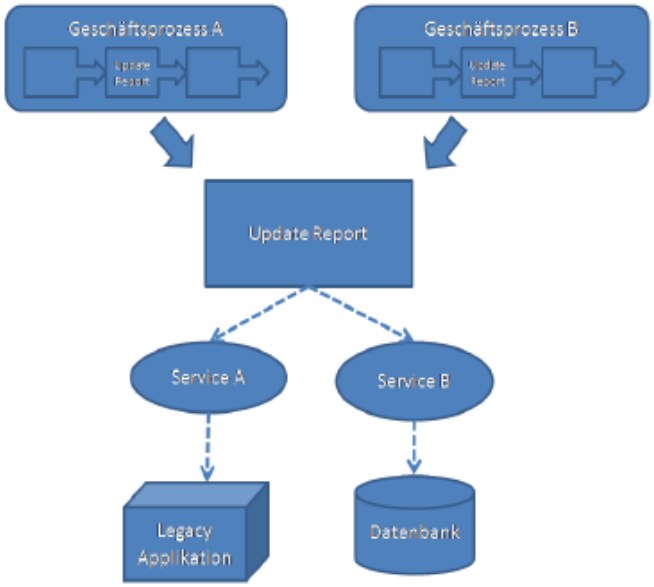


Abbildung 24: Business Service zusammengesetzt aus technischen Services

Service-orientierte Architektur (Folien S30)

- Geschäftsprozess-orientiert
- Anwendungen werden durch mehrere Services gelöst
- Services in einer Anwendung sollen auch aus anderen Anwendungen heraus aufgerufen werden können
  - ✓ Reuse
  - ✓ Modularisierung
- Zusammenstellung von Services in bestimmten Abläufen zur Umsetzung von Geschäftsprozessen (Orchestrierung)
- Services existieren auf mehreren Abstraktionsebenen (Business Service, Component Service, technisches Service, usw.)
- Beschreibt eine Architektur, keine Technologie

Architekturmuster (Folien S31-34 , Buch S232)

Designpatterns

- Beschreiben wiederholt auftretende Strukturen von miteinander kommunizierenden Komponenten
- Dienen dazu ein allgemein definiertes Problem innerhalb eines bestimmten Kontexts zu lösen
- Patterns sind auf Abstraktionsebene unter den Architekturpatterns

Patterns allgemein

- Patterns stellen allgemeine Lösungsstrategien für wiederkehrende Probleme der Softwareentwicklung dar

- ✓ Abstrakt
- ✓ Weitestgehend technologie-unabhängig
- ✓ Wesentlicher Input in Entwurfsentscheidungen

- Es gibt zwei Arten von Patterns nach Abstraktionsebene
  - ✓ ***Architekturpatterns:*** Muster auf architektonischer Ebene (Makroarchitektur-Ebene)
  - ✓ ***Designpatterns:*** Muster auf Klassen-Ebene (Mikroarchitektur)
- Pattern Language
  - ✓ Sammlung von Entwurfsmustern
  - ✓ Schaffung einer einheitlichen Sprache für Probleme und Lösungen

Architektur-Patterns

- Schablonen für konkrete Softwarearchitekturen
- Spezifizieren systemweite, strukturelle Eigenschaften einer Anwendung
- Großer Einfluss auf die Architektur der eigenen Subsysteme
- Fundamentale Designentscheidung
- Bekannte Architekturmuster
  - ✓ ***Model-View-Controller*** (S231-232, Folien S34)
    - Beispiel für einen Model-View-Controller
      - Szenario: Interaktive Anwendungen mit einer flexiblen Schnittstelle zwischen Mensch und Computer
      - Problem: GUI wird oft geändert
      - Lösung: Teilt eine Anwendung in drei Gebiete: Verarbeitung, Output und Input
    - Das Model kapselt die Daten und die Funktionalität
    - Es ist unabhängig von der Darstellung des Outputs und des Input-Verhaltens
    - Die View stellt die Informationen für den User dar
    - Die View erhält die Daten vom Model (es kann für ein Model mehrere Views geben)
    - Jede View hat einen Controller, der Input in Form von Events empfängt und diese in Form von Service Request an das Model oder die View weitergibt.
    - User-Interaktion findet ausschließlich über den Controller statt.

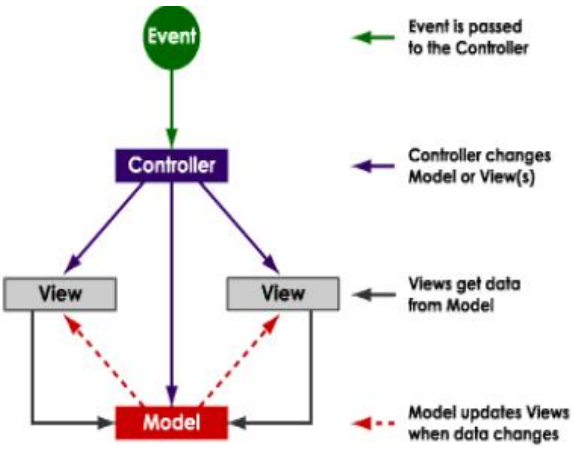


Abbildung 25: Model View Controller

- ✓ Pipes and Filters
- ✓ Layered Architecture

Implementierung, Integration & Testen

Programmiersprachen (Folie S6, S248)

- Wahl der Programmiersprachen hat wesentlichen Einfluss auf die Implementierungsphase
- Selten direkt für das Scheitern eines Projektes verantwortlich
- Beherrschung der Sprache notwendig aber nicht hinreichend für Projekterfolg
- Faktoren die diese Sprachwahl beeinflussen:
  - ✓ Kenntnisse und Erfahrungen des Teams
  - ✓ Eignung der Sprache für die Problem-Domäne (Enterprise Systems vs. Mobile Devices / Embedded Systems)
  - ✓ Verfügbarkeit von Entwicklungstools, Frameworks und Libraries
  - ✓ Verbreitung der Sprache
  - ✓ Verfügbarkeit von Entwickler
  - ✓ Kundenvorgabe

Framework (S248-250, Folie S7-8)

- Ein Framework ist ein Applikationsskelett, das vom Entwickler angepasst (spezialisiert) wird
- Beispiele: Spring, Hibernate, AngularJS, JUnit, Node.js
- Vorteile
  - ✓ Wiederverwendung
  - ✓ Entwicklungszeit kürzer, Kosten senken
  - ✓ Softwarequalität erhöhen
  - ✓ Entwicklung kann auf Spezialisierung vorgegebener Strukturen reduziert werden
- Nachteile
  - ✓ Komplexität (Black/Magic Box), daher Einarbeitung und Erfahrung notwendig

Kriterien zur Auswahl eines Frameworks

- Deckt das Framework die Anforderungen ab? → Siehe Dokumentation des jeweiligen Frameworks
- Wartung und Weiterentwicklung gesichert? →Hoher Verbreitungsgrad bedeutet meist gute Wartung, Weiterentwicklung, Stabilität
- Kommerziell: Support, aber Kosten, auch für Änderungen
- Frei: Support bei Bedarf zukaufen, aber Risiko durch fehlende Garantien

Framework vs. Library (Folien S9, Buch S251)

- Die Begriffe Bibliothek (Library) und Framework werden häufig verwechselt, die Unterschiede sind jedoch essentiell!
- Die folgenden Punkte sollen die typischen Unterschiede verdeutlichen:
  - ✓ Eine Bibliothek bietet wiederverwendbare Funktionalität an, während ein Framework wiederverwendbares Verhalten bereitstellt.

- ✓ Methoden von Bibliotheken werden durch den eigenen Code aufgerufen, während bei der Verwendung von Frameworks der eigene Code hingegen durch das Framework aufgerufen wird
- ✓ Frameworks bestehen aus abstrakten Klassen und bieten Funktionalität in (teilweise) implementierten Methoden an. Diese Methoden können im Weiteren die Funktionalität aus Bibliotheken beziehen
  - ➢ Inversion of Control (Buch S255-256): Der Kontrollfluss bleibt beim Framework und der Code des Entwicklers wird in das Framework eingebaut. Die Abhängigkeiten werden dabei extern beschrieben und sind somit leichter zu warten.
  - ➢ Hollywood-Prinzip

Entwicklungsumgebung (IDE) (Buch S258, Folien S11)

- IDE = Integrated Development Environment
- Zentraler Arbeitsbereich für Entwickler
- Zugriff auf alle benötigten Tools
- Flexibles Layouting und individuelle Gestaltung/Anpassung
- Beispiele:
  - ✓ JetBrainsFamilie (IntelliJ IDEA, WebStorm, …)
  - ✓ Eclipse
  - ✓ NetBeans
  - ✓ XCode
  - ✓ Visual Studio

Funktionsumfang einer IDE (S258-260, Folie S12)

- Source File Editor
  - ✓ Syntax Highlighting
  - ✓ Code Completion
  - ✓ Snippets/ Templates
- Integriert Compiler (inkl. inkrementelles Compilen), Linker, Interpreter
- Debugger, Profiler
- Unterstützung für Refaktorisierung
- Unterstützung für einheitlichen Code-Qualität (statische Analyse) und -Formatierung
- Außerdem: Build Tools, Unit Testing, Volltextsuche, Strukturelle Suche, VCS Integration, …

Version Control System (Buch S 285-286, Folien S13-14)

- VCS (Versionsmanagementsystem)
- ist ein Computersysteme, das Änderungen an Dateien über die Zeit hinweg archiviert.
- erlaubt jederzeit das Laden der aktuellen und jeder archivierten Version einer Datei.
- Repository: Datenbestand aus dem die Arbeitskopie generiert wird.
- Checkout: Übertragung (Abholen) der Daten aus dem Repository
- Check-in oder Commit: Übertragung der Daten in das Repository
- Verzweigung/Branching: Unter Branching versteht man das Anlegen einer Kopie von Objekten im VCS

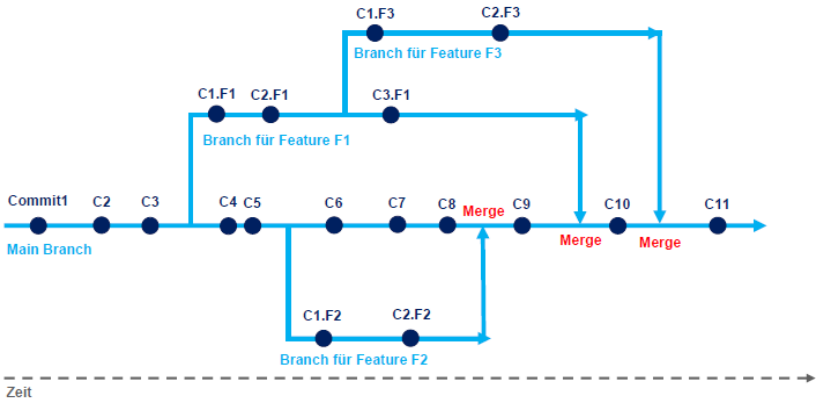


Abbildung 26: Version Control - Branching

Zentrales/Dezentrales Versionsmanagement (Buch S 287, Folien 15)

Zentrales Versionsmanagement

- Stellt einen typische Client/Server Ansatz dar
- Das gemeinsame Repository ist auf einem zentralen Server gelagert und sämtliche Clients können darauf zugreifen
- Spielt keine Rolle ob sich diese im LAN oder im Internet verteilt sind
- Beispiele
  - ✓ SVN (Subversion)
  - ✓ CVS (Concurrent Versions System)

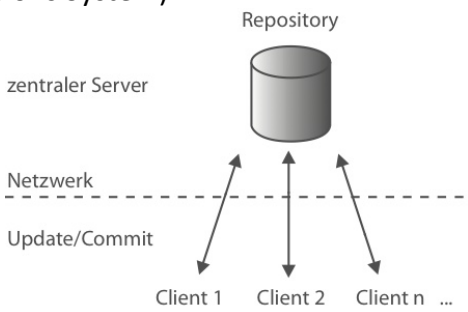


Abbildung 27: Zentrales Versionsmanagement

Dezentrales Versionsmanagement

- Hier existiert kein zentrales Repository
- Jeder Entwickler besitzt ein eigenes Repository und arbeitet mit seinem eigenen Entwicklungszweig.
- Änderungen landen immer im Repository des Entwicklers.
- Jeder Commit ist konfliktfrei
- Änderungen werden zwischen den Repositories ausgetauscht
- Beispiel: Git

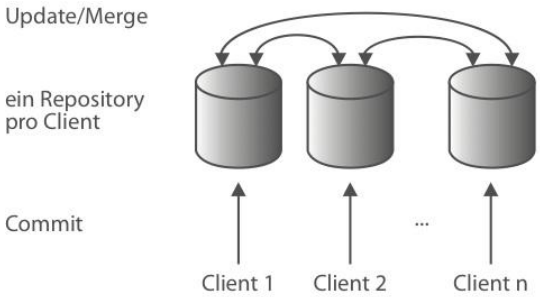


Abbildung 28: Dezentrales Versionsmanagement

Build Management System (Buch S284-285, Folien S15)

- Ist ein Werkzeug, um immer wiederkehrende Aufgaben zum Bauen einer Applikation automatisiert ablaufen zu lassen.
- Build System muss sicherstellen, dass die Abhängigkeiten von anderen Komponenten vorhanden sind.
- Dies kann geschehen, indem die Abhängigkeiten gebaut werden, oder dass sie als fertige Bibliothek vorliegen.
- Beispiele
  - ✓ Make(Urvater der BMS), für C/C++
  - ✓ Maven, für Java
  - ✓ Ant, für Java
  - ✓ Gradle, für Java, insb. Android

Kopplung, Kohäsion, Schnittstellen (S264-265, Folie S19-20)

Kopplung

- Ist ein Maß der Abhängigkeiten einer Klasse von einer anderen
- Aus Wartungsperspektive ist es zielführend, wenn eine Klasse eine möglichst geringe Kopplung aufweist
- Bei einer hohen Kopplung ist es notwendig, bei einer Änderung in Klasse A auch Klasse B zu ändern.
- Eine hohe Kopplung bedeutet einen schlechten Programmierstil

Kohäsion

- Die Kohäsion einer Klasse gibt an, ob die Klasse einen klar definierten Zweck hat.
- Eine hohe Kohäsion wird durch Methodengruppierungen nach Zweck gewährleistet.
- Wenn die Methoden einer klasse mehrere unabhängige Zwecke bediene, sinkt die Kohäsion der Klasse
- Ist also ein Maß der Zusammengehörigkeit innerhalb einer Komponente
- Single-Responsibility-Prinzip: eine Komponente erfüllt genau eine Aufgabe
- Noch besser: nur diese Komponente erfüllt diese Aufgabe (Don‘t-Repeat-Yourself-Prinzip)

➔ Eine geringe Kopplung wird durch eine hohe Kohäsion erreicht!

Interfaces

- Interfaces zwischen Komponenten (Klasse, Modul, …) definieren

Information Hiding

- Ist ein grundlegendes Konzept, dass eine Klasse von der anderen so wenig wie möglich und so viel wie nötig wissen soll
- Abhängigkeiten zwischen den Klassen soll gering sein um aufwändiges Neuimplementieren im Falle von Änderungen vorzubeugen.
- Die Kommunikation zwischen den Komponenten erfolgt über Schnittstellen (Interfaces).
- Der Zugriff auf die eigentliche Implementierung ist somit nicht notwendig

Entwurfsmuster (Design Pattern) (Buch S252-264, Folien S21-22)

- Hier wird auf die Muster für die Implementierung eingegangen (= Design Pattern)
- (Dies ist ein anderes Pattern als das für die Architektur)
- Sind eine Toolbox für Programmierer
- Bieten für bestimmtes Programmierprobleme bewährte Lösungen
- Es gibt eine große Anzahl an verfügbaren Entwurfsmuster
- Die „Gang of Four“ (Gamma et al) haben die Pattern folgendermaßen eingeteilt:

Erzeugermuster / Creational Patterns

- Beispiele: Abstract Factory, Builder, Singleton, Prototype,…

Strukturmuster / Structural Patterns

- Beispiele: Adapter, Bridge, Decorator, Facade, Proxy,…

Verhaltensmuster / Behavioral Patterns

- Beispiele: Command, Iterator, Observer, Strategy, ...

Beispiel: Observer Pattern

- Dies ist rein funktionell gesehen ein publish/subscribe Pattern, bei der KomponenteA (Observer) über den Zustand und Zustandsänderungen von KomponenteB (Subject) Bescheid wissen muss.
- Observer besitzt eine Liste an Subjects und deren derzeitigen Status
- Subjects können sich am Observer auch explizit an– oder abmelden und benachrichtig diesen im Fall einer Statusänderung

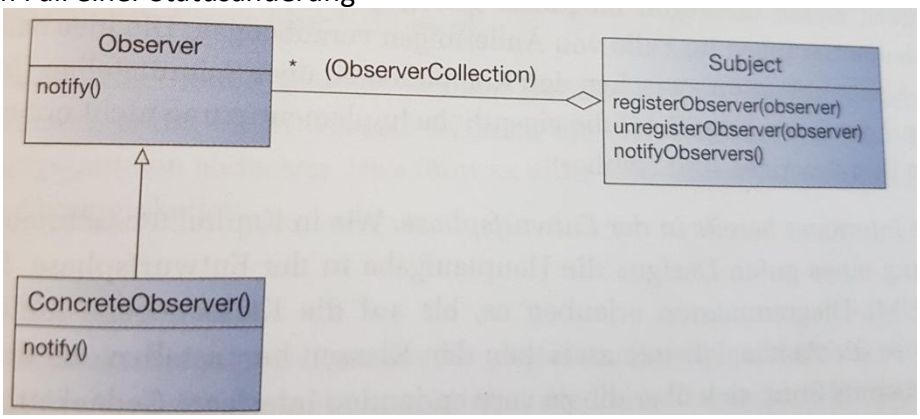


Abbildung 29: Klassendiagramm eines Observer Pattern

Entwicklungsrichtlinien (Folien S23)

- Diverse Richtlinien für Entwickler eines Projektes
- Ziel: konsistenter Code
- Motivation: kürzere Einarbeitungszeit für Entwickler
- Richtlinien betreffen:
  - ✓ Namenskonventionen für Variablen/Methoden/Klassen/…
  - ✓ Programmierstil
  - ✓ Formatierung des Quellcodes
  - ✓ Kommentarstil
  - ✓ Verwendete Design Patterns

- Teilweise automatisiert überprüfbar (Checkstyle,…)

Dokumentation des Codes (Folien S24, Buch S273-274)

- Erhöht die Lesbarkeit und Übersicht
- Code kann auch nach Monaten wieder verstanden werden
- Code kann von anderen Entwicklern besser nachvollzogen werden
- Kommentierung
- Entwicklungsrichtlinien
- Oftmals die einzige Art der Dokumentation die zum Zeitpunkt der Ablöse des Systems noch erhalten ist

Testen

- Jeder Entwickler soll auch Tester sein
- In jeder Entwicklungstätigkeit findet sich ein kontinuierlicher Zyklus bestehend aus dem Verstehen von Anforderungen, Scheiben oder Anpassen des Quellcodes, dem Bauen einer lauffähigen Version und schließlich dem Ausführen und Testen

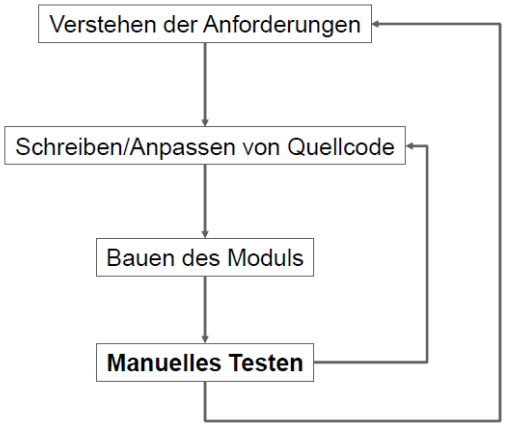


Abbildung 30: Testen

Automatisierte Tests (Folien S26, S267)

- Tests oft unstrukturiert, oberflächlich und lückenhaft –da es sich lediglich um ein „ausprobieren“ handelt
- Lösung um diese Fehlerquellen zu vermeiden sind automatisierte, feingranulare Funktionstests (Unit Tests)
  - ✓ Diese Tests überprüfen die Funktionalität von einer möglichst kleinen funktionalen Einheit, einer Unit und beziehen sich daher auf einen kleinen funktionalen Aspekt
  - ✓ Sie sind automatisiert ausführbar und daher billig und schnell wiederholbar.
- Diese Tests sind einfach ausführbar und werden von den gängigsten Entwicklungsumgebungen als auch Build Tools das Ausführen der Tests unterstützt.
- Sind sehr wichtig zur Vermeidung von ungewollten Nebeneffekten
- Erst eine gute Codeabdeckung durch automatisierte Tests erlaubt das Bauen größerer, robuster Softwaresysteme (und die „unbeschwerter“ Weiterentwicklung)!

Test Driven Development (TDD)

- Ist jene Form der Integration von Test und Entwicklung, bei der das Testen die vorrangige Stellung einnimmt.
- Hier wird eine Entwicklungsweise angewendet, bei der die Testfälle bei der Entwicklung als erstes kommen und erst danach der dazugehörige funktionale Quellcode.
- Dabei werden folgende Schritte iterativ durchgeführt:
  - ✓ ***Test schreiben:*** Implementierung jedes neuen Features beginnt damit, den Test zu verfassen. Entwickler muss sich vorher überlegen, welche Anforderungen dieses Feature erfüllen muss
  - ✓ ***Alle Tests ausführen:*** Tests sollten zunächst fehlschlagen, weil die Feature noch nicht implementiert wurden. Muss sich dann entscheiden, ob Tests nur zufällig durchlaufen, oder weil die Funktionalität schon vorhanden ist
  - ✓ ***Quellcode schreiben:*** Funktionalität des Features wird hinzugefügt
  - ✓ ***Alle Tests ausführen:*** Wenn Testfälle durchlaufen, erhält Entwickler die Bestätigung, dass die Anforderungen erfüllt sind.
  - ✓ ***Refactor:*** Da Der Quellcode funktional richtig ist, kann der Entwickler den Code so umstrukturieren, dass er den gegeben Entwurfsrichtlinien entspricht.

Nach Beck (2002) gibt es für das TDD zwei goldenen Regeln:

- You should write new business code only when an automated test has failed“
- „You should eliminate any duplication that you find“

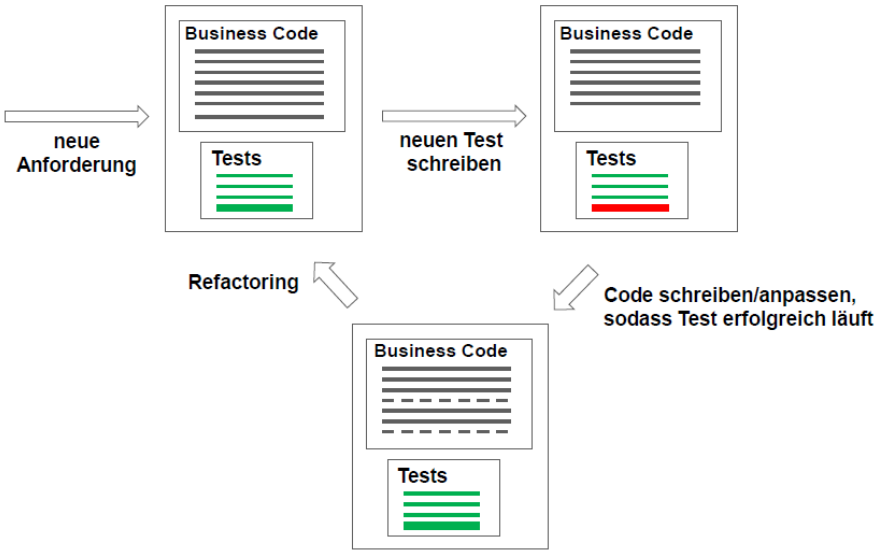


Abbildung 31: Test Driven Development

Integration und Testen – Wie wird zusammengefügt / überprüft

Stellenwert des Softwaretests (Buch S299-301, VO6 S33)

- Keine Software ist frei von Fehlern
- Testen gehört zu den wichtigsten Aktivitäten des Entwicklungsprozesses.
- Durch systematisches Testen werden Abweichungen eines lauffähigen Softwareprodukts von der Spezifikation aufgedeckt und Fehler identifiziert
- Eine gut geplante Testphase ist die Grundlage für ein erfolgreiches Projekt.
- Ziel ist es, Fehler so früh wie möglich zu finden
- Softwaretests sind eine dynamische und produktorientierte Qualitätssicherungstechnik.
- Aufgrund wachsender Komplexität und hoher Abhängigkeiten von Softwaresystemen, wird der Softwaretest strategisch immer bedeutender.

Definition Softwaretest:
Definition nach IEEE 610.12, 1990:
„Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.
All the associated tests are performed according to approved test procedures on approved test cases.“

Grundsätze des Softwaretests (S303-304, VO6 S45)

(1) Testen zeigt die Anwesenheit von Fehlern
(2) Vollständiges Testen ist nicht möglich
(3) Mit dem Testen frühzeitig beginnen
(4) Häufung von Fehlern (Dort wo schon eine Fehlerwirkung nachgewiesen wurde, findet man meist noch weitere, Gleichverteilung von Fehlerzustände auf Testobjekt ist meist nicht gegeben)
(5) Wiederholungen haben keine Wirksamkeit (Wiederholen derselben Testfälle ist nicht wirksam. Überprüfen und erstellen von Testfällen ist nötig)
(6) Testen ist abhängig vom Umfeld (Je nach verwendeten System, muss das Testen angepasst werden. Zwei unterschiedliche Systeme, sind nicht gleich zu testen)
(7) Trugschluss: Keine Fehler bedeutet ein brauchbares System

Ziele und Methodik des Softwaretestens (VO6 S36)

Ziele

- ***Verifikation:*** Prüfen des Systems gegen Spezifikation
  - ✓ „Erstellen wir das Produkt richtig“
- ***Validierung:*** Prüfen des Systems gegen (Kunden-)Anforderungen
  - ✓ „Erstellen wir das richtige Produkt“

Methodik

- Testfälle identifizieren, mit denen die höchste Wahrscheinlichkeit gegeben ist, festzustellen, ob das Softwaresystem korrekt funktioniert
- Möglichst hohe funktionale oder nichtfunktionale Abdeckung

Test- und Integrationsstufen (Buch S305, VO7 S4)

- Eine wichtige Strategie bei der Herstellung von Softwaresystemen ist die Zerlegung der gestellten Aufgabe in beherrschbare Teile ***(„Divide and Conquer“)***
- Auch eine Festlegung verständlicher und prüfbarer Übergabepunkte zwischen den Teilen
- Die Bedeutung dieser Strategie für die Testphase ist, dass jeder Projektphase bzw. deren Ergebnissen ist ein korrespondierender Testschritt zugeordnet. Dieser ist direkt im Anschluss der Phase durchzuführen.
- Am Anfang werden fachliche Aspekte des Projekts geprüft (z.B. die Machbarkeit durch eine Visions- und Zielprüfung)
- Später im Projekt werden konkrete technische Artefakte (z.B. die Leistungsfähigkeit und die Korrektheit des technischen Designs durch einen Modul- oder Klassentest
- Nachfolgend ein Beispiel der Projektphase und der darauffolgenden Testphase

Projektphase	Teststufe
Architekturdesign	Systemintegrationstest
Technisches Design	Komponentenintegrationstest
Modul-/Klassendesign	Komponenten-/Unittest

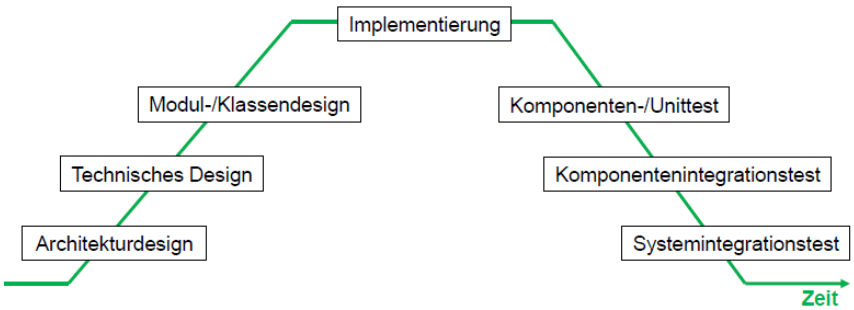


Abbildung 32: Test- und Integrationsstufen

Komponententest (S307-308, VO7 S6)

- Ziel ist das prüfen von separat testbaren Komponenten (z.B. Modulen, Programmen, Objekten, Klassen) auf vorhandenen Fehler
- Dies wird durch Isolation der einzelnen Komponenten vom Rest des Systems erreicht.
- Es werden hier funktionale und nichtfunktionale Aspekte getestet wie z.B.
  - ✓ Umgang mit Ressourcen (z.B. Speicherengpässe)
  - ✓ Robustheitstest
  - ✓ oder strukturelle Prüfungen (z.B. Zweigüberdeckung)
- Testfälle für Komponententests werden aus Entwicklungsdokumenten abgeleitet:
  - ✓ Komponentenspezifikationen
  - ✓ Softwareentwürfe
  - ✓ Datenmodelle
  - ✓ Quellcode

- ✓ Unterstützungen aus diversen Entwicklungsumgebungen
- ✓ Debugging-Werkzeugen

Ablauf des Tests

- Eine Komponente bietet ihre Funktionalität über definierte Schnittstellen an
- Diese Schnittstellen werden beim Komponententest durch einen Testtreiber bedient.
- Dieser ist entweder vollautomatisiert oder stellt eine Benutzerschnittstelle für einen manuellen Test bereit
- Eine Komponente verwendet wieder andere Komponenten und diese müssen zur Verfügung gestellt werden.

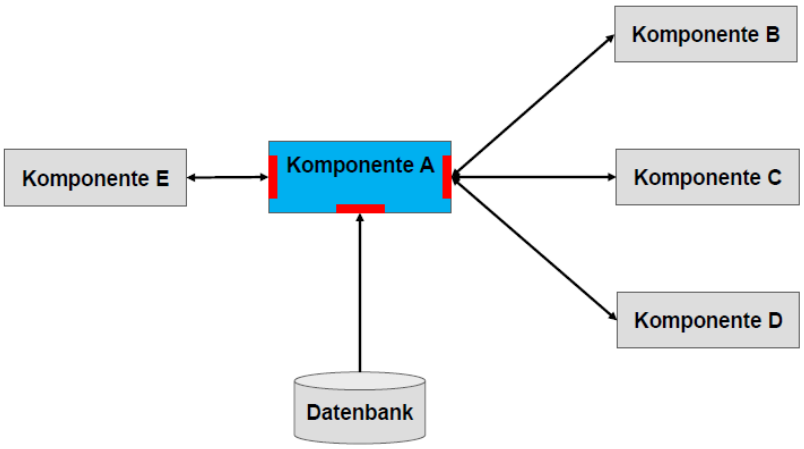


Abbildung 33: Komponententest - Ausgangssituation

- Wegen der Testisolation, werden diese verwendeten Komponenten isoliert und deshalb ein Simulator bereit gestellt
- Der Simulator muss die Schnittstellen zur getesteten Komponenten so versorgen, dass die Funktionsfähigkeit der Komponente getestet werden kann. Spricht hier vom einen ***Stub***, wenn dieser selbst keine Funktionalität simuliert, sondern nur die Schnittstelle strukturell bedient.
- Von einer ***Simulation*** ist die Rede, wenn die Schnittstelle auch funktional vollständig bedient wird.
- Die Konfiguration und die Testdaten müssen von allen beteiligten Entitäten (Simulatoren, zu testende Komponente und Testtreiber) übereinstimmen.

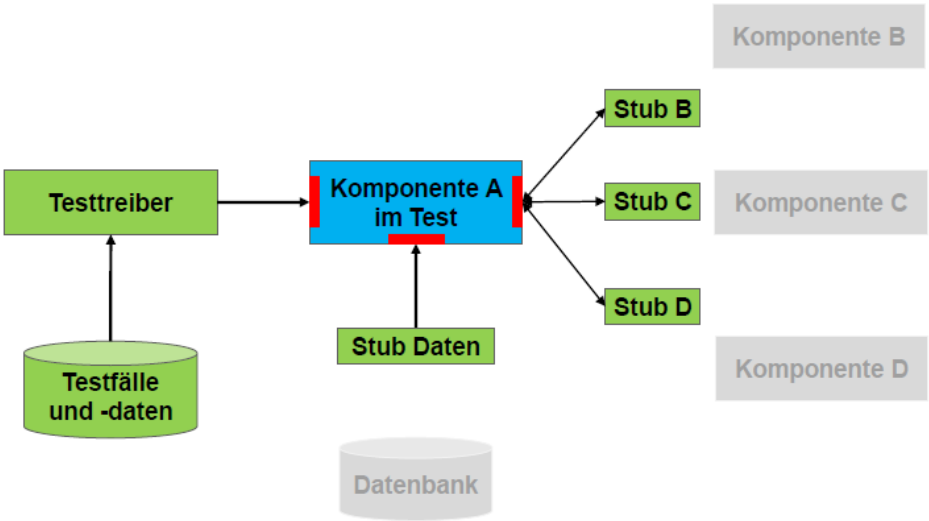


Abbildung 34: Komponententests - Test-Setup

Integrationstests (S308-312, Folien S9-12)

- Hier werden Schnittstellen zwischen Komponenten, die Interaktion zwischen unterschiedlichen Systemteilen (z.B. zum Betriebs- oder Dateisystem) oder systemübergreifende Schnittstellen zwischen verschiedenen Systemen geprüft.
- Die Integration und der Test eines teilintegrierten Systems können auf verschiedene Arten erfolgen:

Big-Bang-Integration

- *Vorgehensweise:*
  - ✓ Alle Komponenten werden gleichzeitig integriert (= „Big-Bang“).
  - ✓ Nicht-iterative Integrationsform
- *Vorteile:*
  - ✓ Keine zusätzlichen Testaufwände für Test-Stubs (um fehlende Funktionalität zu simulieren) alle Komponenten sind verfügbar.
  - ✓ Werden auch keine Testtreiber benötigt
- *Nachteil:*
  - ✓ Fehler sind sehr schwer zu lokalisieren (z.B. Seiteneffekte)
  - ✓ Hohes Risiko bei der Integration
  - ✓ Ist erst spät im Projekt möglich (Risiko)
- *Anwendung:* Kleine und überschaubare Produkte

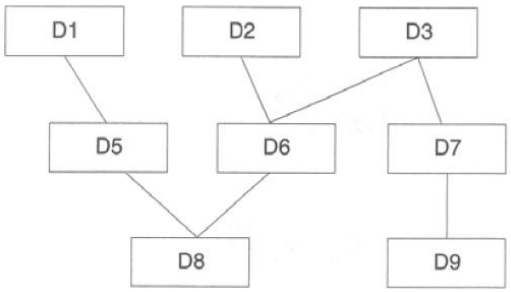


Abbildung 35: Big-Bang-Integration

Horizontale Integration – Top-Down-Integration

- Ist eine iterative, schichtorientierte Integration
- *Vorgehensweise:* Schrittweise Integration (z.B. D1, D2, D3), ausgehend von den Business Cases.
- *Vorteile:*
  - ✓ Ausführbares Produkt Framework ist früh verfügbar.
  - ✓ Externe Schnittstellen 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).

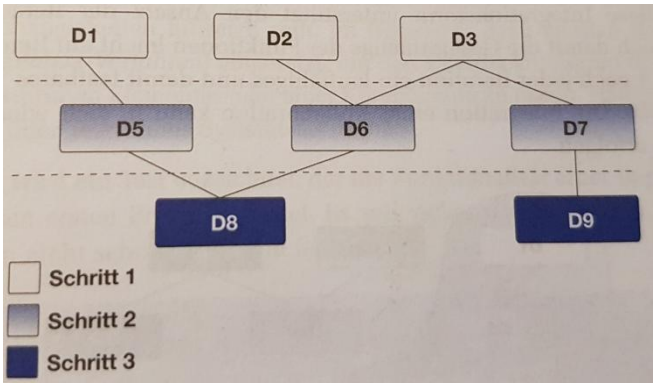


Abbildung 36: Top-Down-Integration

Horizontale Integration – 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).
  - ✓ keine Stubs notwendig (dafür mehr Testtreiber, geringerer Aufwand)
- *Nachteile:*
  - ✓ Ausführbares Gesamtsystem ist spät verfügbar.
  - ✓ Externe Schnittstellen spät verfügbar
  - ✓ Zusätzlicher Aufwand für Prototypen.
  - ✓ Zusätzlicher Aufwand für Test Drivers, um (lower-level) Komponenten zu testen.

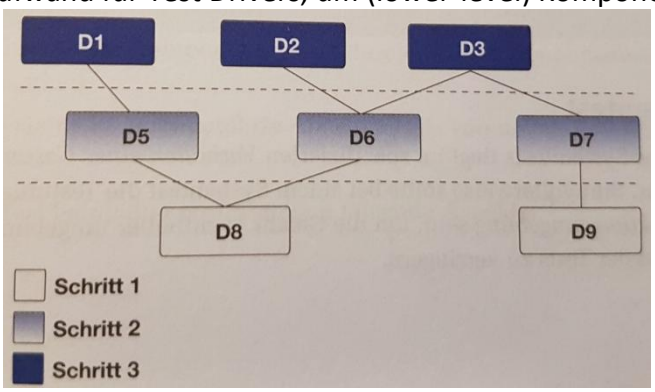


Abbildung 37: Bottom-Up-Integration

Vertikale Integration

- Integration von ausgewählten Komponenten, um einzelne Anwendungsfälle (Use Cases) entsprechend dem Business Case umzusetzen (z.B. über priorisierte Liste von Use-Cases)
- *Vorgehensweise:*
  - ✓ Schrittweise Integration entsprechend den Business Cases (über unterschiedliche Layer hinweg), z.B. D1/D5/D8.
  - ✓ Phasen-orientierte Integration.
- *Vorteile:*
  - ✓ Frühe Verfügbarkeit von funktionalen Anforderungen (über alle Layer).
  - ✓ Prototypen und Demo.
  - ✓ Berücksichtigung priorisierter Anforderungen möglich.
  - ✓ Nach jeder Iteration: lauffähiges, testbares Teilsystem (vgl. Scrum)

- *Nachteile:*
  - ✓ Wiederverwendung von Komponenten kann schwierig sein.
  - ✓ Regressions-Tests sind erforderlich.
  - ✓ Alle Schichten müssen Funktionalität unterstützen

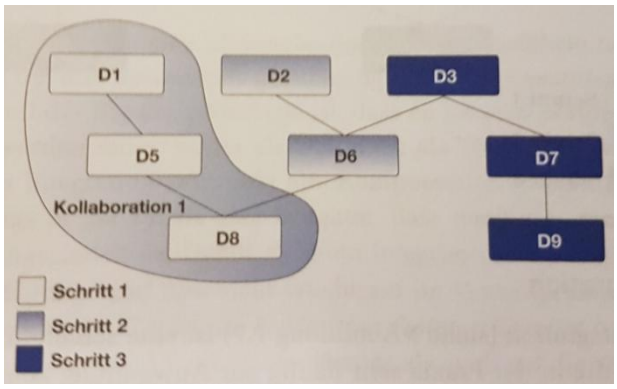


Abbildung 38: Vertikale Integration

Systemtest (S312-312, VO7 S13)

- Testet das spezifizierte Verhalten des Gesamtsystems
- Bei einem Systemtest sollte die Testumgebung äquivalent zur Produktivumgebung sein
- Sollten funktionalen und nichtfunktionalen Anforderungen an das System abdecken.

Akzeptanztest (Abnahmetest) (S313-314, VO7 S13)

- Dieser wird meist vom Kunden oder Benutzer des Systems durchgeführt
- Prüft, ob Auftragnehmer die vereinbarten Leistungen umgesetzt hat
- Es gibt verschiedenen Arten von Akzeptanztests:

Anwender-Abnahmetest

- Prüft die Benutzbarkeit eines Systems zum Gebrauch durch den Anwender/Kunden

Betrieblicher Abnahmetest

- Eine Abnahme durch den Systemadministrator beinhaltet z.B. Erstellen und Wiedereinstellen von Sicherungskopien (Backup-/Restore-Möglichkeit), Wiederherstellbarkeit, Benutzermanagement, Wartungsaufgaben, Periodische Überprüfung von Sicherheitslücken

Regulatorischer und vertraglicher Abnahmetest

- vertragliche Abnahmekriterien zwischen Auftraggeber und Auftragnehmer in Bezug auf das erstelle Produkt
- Abnahmekriterien werden üblicherweise mit dem Vertragsabschluss definiert
- Regulatorisch: Hier wird überprüft ob das System den Gesetzen oder Standards entspricht.

Feldtest

- Feedback von existierenden oder potenziellen Kunden einholen

Funktionale Softwaretests (S 315 -327, VO7 S15-19)

- Als typischer Softwaretest wird häufig die funktionale Verifikation bezeichnet.
- Das Ziel von der funktionalen Verifikation ist, eine möglichst hohe Abdeckung (Coverage) zu erhalten
- Es muss unbedingt eine Selektion der Testfälle erfolgen.
- Es sollen die Testfälle ausgewählt werden, die mit größter Wahrscheinlichkeit Fehler finden bzw. die größtmögliche Abdeckung erreichen.
- Es wird hier zwischen zwei Ansätzen der Abdeckung unterschieden:

Funktionale Softwaretests - Strukturelle Abdeckung

- Man spricht hier von einem Whitebox-Test
- Es wird hier versucht den Systemaufbau als Basis für ein Abdeckungsmaß heranzuziehen. Die innere Struktur ist die strategische Grundlage für die Testfallselektion
- Orientiert sich am Kontrollflussgraphen des Quellcodes des Softwaresystems.
- Je nach Art des gewählten kontrollflussorientierten Testverfahrens kann mittels Hilfsmittels die Testabdeckung (Code-Coverage) berechnet werden.
- Es werden verschiedenen Arten des kontrollflussorientierten Testverfahrens unterschiedene:
  - ✓ Anweisungsüberdeckung
  - ✓ Kantenüberdeckung
  - ✓ Bedingungsüberdeckung
  - ✓ Pfadüberdeckung

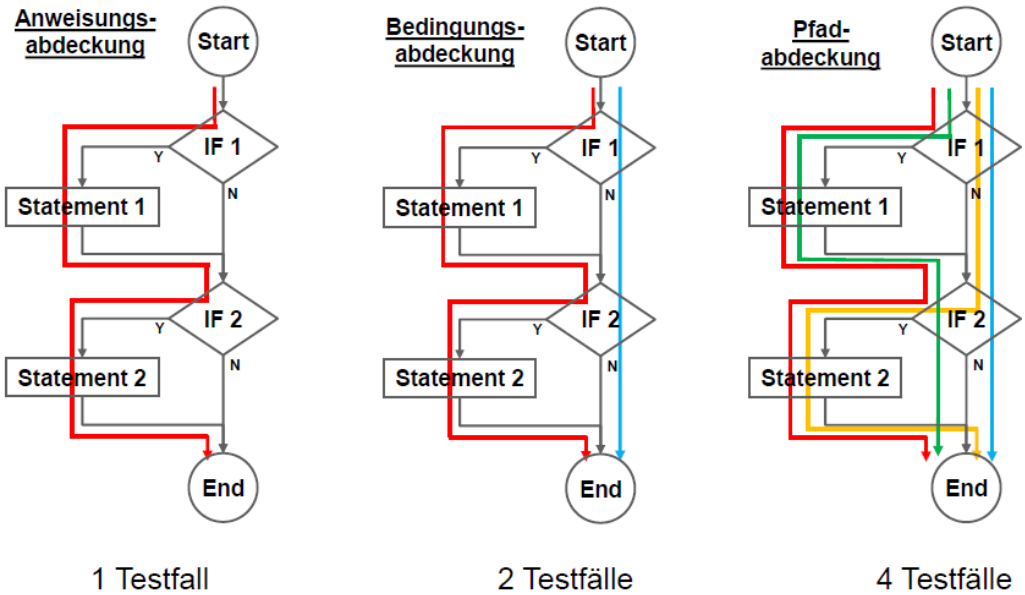


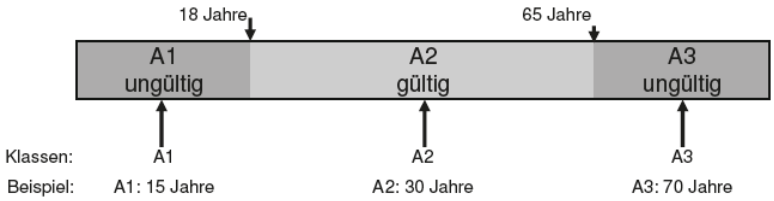
Abbildung 39: Strukturelle Abdeckung

Funktionale Softwaretests - Funktionale Abdeckung

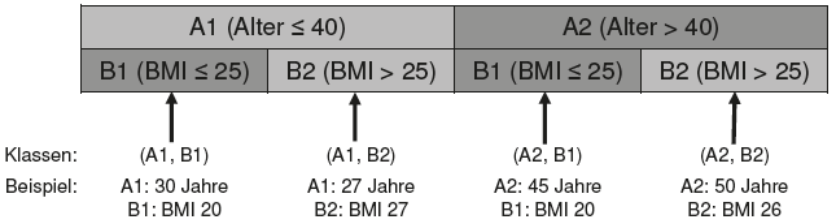
- Hier ist die Testfallselektion spezifikationsgetrieben und versuchen auf Basis einer externen Systembeschreibung, eine möglichst effektive Selektion zu treffen
- Diese Art wird auch als Blackbox-Test bezeichnet.
- Wird überprüft, ob das Zielsystem der vorgegeben Spezifikation entspricht

Äquivalenzklassenanalyse

- Zählt auch zu den funktionalen Methoden
- Darunter versteht man die Einteilung möglicher Ein- und Ausgabewerte in Klassen
- Es werden Werte gewählt, bei denen angenommen wird, dass Fehler, die bei der Anwendung dieser Werte aus jener Klasse auftreten, auch bei allen anderen möglichen Werten der Klasse auftreten
- Andererseits wird angenommen, dass bei einer korrekten Arbeitsweise mit anderen Werten, keine Fehler auftreten.
- Ausgehend von dieser Annahme, werden alle Werte einer Klasse als äquivalent angesehen.
- Aus diesen Klassen von Eingabewerten wählt man ***einen Vertreter*** (Repräsentanten) aus, der dann für den konkreten Testfall verwendet wird.
- **Beispiel:** Wir wollen wir eine Altersüberprüfung testen: 18 < Alter <= 65. Wir haben hier also drei grundlegende Klassen von Eingabewerten.
  A1: Alter <= 18
  A2: Alter zwischen 18 und 65
  A3 Alter >65.
  Alle Werte aus einer Äquivalenzklasse liefern dasselbe Ergebnis, daher wählt man nur einen Vertreter pro Klasse aus.



- **Grenzwertanalyse**
  - ✓ Ein Spezialfall von Äquivalenzklassen
  - ✓ Einen besonderen Stellenwert nehmen Bereichsgrenzen von Äquivalenzklassen ein, da diese häufig Ursachen für Fehler sind, die beispielswiese durch Tippfehler verursacht werden (z.B. <= wird <)
  - ✓ Diese Fehler können, speziell an den Systemgrenzen, zu einem Fehlerverhalten führen.
  - ✓ Diese Fehlerquelle kann mit Grenzwertanalyse erkannt werden.
  - ✓ **Kernidee:** Die Grenzbereiche zu identifizieren und Testdaten aus dem nahem Umfeld dieser Bereichsgrenze auswählen.
  - ✓ Man sollte hier einen Wert aus beiden Klassen auswählen.



Zustandsbasierte Testmethoden

- Basieren auf Zustandsautomaten
- Testfälle werden von diesen Zustandsautomaten abgeleitet

Klassifikationsbaummethode

- Einteilung der Testfälle in Klassifikationen.
- Jede dieser Klassifikationen wird wiederum in disjunkte Teilmengen, also Äquivalenzklassen zerlegt.
- Testfälle werden durch Kombination verschiedener Klassen erstellt, wobei von jeder Klassifikation nur eine Klasse gewählt wird.

Informelle Testmethode

- Hier wird ganz oder teilweise auf eine systematische Ableitung von Testfällen verzichtet.
- Es werden vorab keine Testfälle und keine erwarteten Ergebnisse spezifiziert und die Durchführung der Tests erfolgt mehr oder minder nach der Intuition des Testers.

Nichtfunktionale Softwaretests (S327-331, VO7 S20-23)

- Hier werden die nichtfunktionalen Anforderungen eines Systems getestet
- Der Test ist immer in Abhängigkeit des zu testenden Qualitätsattributs aufgebaut.
- Ein Test für die Benutzbarkeit ist deshalb anders aufgebaut als ein Test für die Sicherheit.
- Diese Art des Testens erfordert viel mehr Expertenwissen als funktionale Softwaretests. Hier ist ein besonderes Wissen über die Domäne und das Zielsystem erforderlich.
- Oft werden diese Tests von spezialisierten Testteams durchgeführt und nicht von den Testern, die für die funktionalen Tests zuständig sind
- Beispiele:
  - ✓ Benutzbarkeit (Usability)
  - ✓ Leistungsfähigkeit (Performance)
  - ✓ Sicherheit (Security)

Testautomatisierung (S332-337, VO7 S21-23)

- Bei Softwaretests können verschiedene Aktivitäten automatisiert werden
- Die Erstellung von Testfällen nach formellen Methoden (z.B. Äquivalenzklassenanalyse)
- Der Initialaufwand für eine automatisierte Durchführung von Testfällen ist in der Regel wesentlich höher als der Initialaufwand für einen manuellen Test.
- Im Gegenzug ist der Aufwand pro Testlauf im automatisierten Fall deutlich niedriger.
- Der Wartungsaufwand ist bei automatisierten Testfällen im Allgemeinen viel höher als für manuelle Tests

Automatisierte Komponententests

- Diese Tests werden mit speziellen Test-Frameworks durchgeführt
- Test-Framework bietet die Möglichkeit einen Komponententest direkt in der Entwicklungsumgebung zu definieren und ständig parallel zur Entwicklung durchzuführen
- Werden in der Regel in derselben Programmiersprache und –umgebung definiert wie das Zielsystem selbst
- Bieten eine gute Framework-Unterstützung (z.B. JUnit, TestNG)

Automatisierte GUI-Tests

- Hier handelt es sich um eine Simulation des Benutzers über die Benutzerschnittstelle des Systems.

- Sind dadurch definiert, dass sowohl der ***Point of Control (PoC)*** und der ***Point of Oservation (PoC)*** die Benutzerschnittstelle des Systems ist.
- Ein übliches Verfahren zur Bestimmung der Aufbauelemente eines GUI-Tests ist das ***Capture/Replay-Verfahren***. Dabei wird die Benutzerinteraktion aufgezeichnet und daraus ein Testskript erzeugt
- Nachteile: Wartung ist sehr hoch, Testobjekt muss bereits verfügbar sein (Navigation durch das Testobjekt)
- Alternative: Scripting, Hybrider Ansatz

Regressionstests

- Wird durchgeführt, um festzustellen, ob die Systemfunktionalität nach einer Systemänderung gewährleistet ist.
- z.B. Quellcodeänderung, Austausch von Komponenten, Änderung der Systemkonfiguration, Installation des Systems auf einer neuen Umgebung.
- **Kontinuierliche Integration (Continuous Integration)**
  - ✓ werden dazu verwendet, den ständig ändernden Integrationsstand zu überprüfen und den Entwicklern permanentes Feedback über die Systemfunktionalität zu geben.
  - ✓ Wenn eine Systemänderung negative Auswirkungen ergibt, dann wird der Entwickler sofort informiert und kann diese Änderungen frühzeitig rückgängig machen oder korrigieren.

Anwenderakzeptanz, Integration der Systemanwender

Wird erreicht durch:

- Anwendergerechte Dokumentation
  - ✓ Benutzerhandbuch
  - ✓ Online-Hilfe
- Schulungen
  - ✓ Meist mehrere Benutzergruppen (unterschiedliche Funktionalität/Berechtigungen)
- Rollout
  - ✓ „Big Bang“-Rollout ist zu vermeiden
  - ✓ Testbetrieb mit kleinen Benutzergruppe
  - ✓ Parallelbetrieb

Migration von bestehenden Systemteilen/Daten

- Ablöse eines bestehenden Systems durch ein neues System
- Laufenden Betrieb nicht beeinflussen (unterbrechungsfrei)
- Softwaremigration
- Datenmigration
  - ✓ Transfer von Daten in ein neues System
- Hardwaremigration
- Kombination, insb. Softwaremigration + Datenmigration

Inbetriebsetzung (S354-358, VO7 S30)

- Das wichtigste ist, dass der gesamte Inbetriebsetzungsprozess ***im Vorhinein geplant*** und alle benötigten Schritte vor der Durchführung definiert und von allen relevanten akzeptiert werden
- Welche Schritte durchzuführen sind, hängt vom Projekt ab. Nachfolgend die zentralen Fragen:
  - ✓ Gibt es eine eigene Maschine für die Applikation oder läuft sie auf einem Rechner gemeinsam mit anderen Applikationen?
  - ✓ Gibt es eine eigene Datenbank für die Applikation oder wird eine bestehende verwendet?
  - ✓ Welche Systemgrenzen hat die Applikation und über welche Interfaces wird von bzw. auf die Applikation zugegriffen?
  - ✓ Müssen Fremdsysteme oder Bibliotheken in das eigene System integriert werden?
  - ✓ Wer ist für die Installation der Software, Einrichtung der Datenbank, die Bereitstellung der benötigten Hard- oder Software, die Einrichtung von Benutzerkonten oder Netzwerkfreigaben verantwortlich?
  - ✓ Wer nimmt die Software formal ab, sowohl auf technischer als auch auf inhaltlicher Basis?
  - ✓ Wer erstellt die Betriebshandbücher und in welcher Form werden diese zur Verfügung gestellt?
  - ✓ Wann beginnt der Betrieb?

Best Practices

- ✓ Zusammenarbeit zwischen Entwicklern und Betrieb
- ✓ Verantwortlichkeiten festlegen
- ✓ Frühzeitiges Testrollout

Anti-Patterns

- ✓ Manuelle Installation der Software
- ✓ Installation erst nach Abschluss der Entwicklung
- ✓ Manuelle Konfiguration
- ✓ Änderungen direkt in Betriebsumgebung („Hot Fixes“)

Softwarewartung (S361-366, VO7 S32-35)

Definition Softwarewartung

Es handelt sich hier um Änderungen einer Software zur:

- Fehlerbehebung,
- Verbesserung gewisser Eigenschaften der Software (z.B. Performance),
- Anpassung an eine geänderte Umgebung

Die Änderungen erfolgen nach der Lieferung und verändern die Funktionalität nicht.

Auslöser von Wartungstätigkeiten

- Gefundener Fehler
- Performance-Verbesserungen
- Updaten von verwendeten Frameworks und Bibliotheken

- Behebung von Sicherheitslücken
- Optimierung der Konfiguration

Wartungstypen

- **Korrektive Wartung**
  - ✓ Entspricht allen Wartungstätigkeiten, die durchgeführt werden um einen oder mehrere aufgetreten Fehler auszubessern.
- **Präventive Wartung**
  - ✓ Entspricht allen Wartungstätigkeiten, die durchgeführt werden um Fehler im Produktivbetrieb zu verhindern, bevor diese auftreten können
- **Adaptive Wartung**
  - ✓ Entspricht allen Wartungstätigkeiten, die durchgeführt werden um eine Applikation an eine geänderte Umgebung anzupassen
  - ✓ Investititonsschutz
- **Perfektionierende Wartung**
  - ✓ Entspricht allen Wartungstätigkeiten, die durchgeführt werden um eine Applikation zu verbessern
  - ✓ z.B. Usability, Performance, Wartbarkeit, Speicherverbrauch etc.

Maßnahmen während der Wartung (Buch S365, VO7 S35)

Nachfolgend sind typische Maßnahmen, die während einer Wartung durchgeführt, oder zumindest diskutiert werden:

- **Reengineering:**
  - ✓ Darunter versteht man die Neuentwicklung eines bereits funktionierenden Teils einer Software bei gleichbleibender Funktionalität
  - ✓ Das Ziel ist es, die Qualität der Software zu verbessern z.B. Vorbereitung der Applikation auf die Weiterentwicklung.
- **Refaktorisierung:**
  - ✓ Hier handelt es sich um eine Umstrukturierung bei gleichbleibender Funktionalität
  - ✓ Das Ziel ist das Beibehalten der Wartbarkeit, die durch ständige Wartungsanpassungen negativ beeinflusst werden kann.
- **Anti-Patterns:**
  - ✓ Es sollen Anti-Patterns, also das Gegenstück zu Patterns, gefunden und eliminiert werden.
  - ✓ Anti-Patterns sollen früh im Lebenszyklus gefunden werden, damit diese leicht ausgebessert werden können, da bereits Verbesserungsstrategien bekannt sind (z.B. Refaktorisierung)

Vorgehensmodelle

Grundlagen und Bedeutung von Vorgehensmodellen (Buch S371-373, VO8 S3)

- Die heutzutage von Softwareentwicklern zu erstellenden Softwareprodukte werden immer umfangreicher und komplexer.
- Diese Softwaresysteme müssen die Erwartungen des Kunden hinsichtlich der Qualität erfüllen und termingerecht und innerhalb eines gewissen Budgetrahmens kreiert werden.
- Effektive und effiziente Software-Entwicklungsprozessmodelle und ihre praktische Umsetzung gewinnen immer mehr an Bedeutung und sind wesentlich am Geschäftserfolg eines Softwareunternehmens beteiligt.
- Ziel ist es, die Softwareentwicklung auf eine etablierte, ingenieurmäßige Basis zu stellen, um Verbesserungen in der Produktivität, Qualität und Vorhersagbarkeit zu erreichen.
- Als Konsequenz sollte die Industrie die Software nach expliziten Prozessmodellen entwickeln.
- Es wurde bisher kein optimaler Entwicklungsprozess, eine Art Universalprozess, der für jedes Softwareprojekt anzuwenden ist, entwickelt.
- Es gibt kein einheitliches und immer anwendbares Prozessmodell, sondern dies muss je nach Projektart das geeignetste Vorgehen nach Ermessen ausgewählt werden.
- Es haben sich sogenannte State-of-the-Art-Software-Entwicklungsprozessmodelle etabliert.

Die drei Steuergrößen eines Projekts (VO8 S4)

- Kosten
- Umfang und Inhalt (Qualität)
- Zeit

Abbildung 40: Steuergrößen eines Projekts

Fundamentale Prozessmodelle (Buch S373-379, VO8 S6)

- Die im Folgenden vorgestellten Vorgehensmodelle basieren auf der grundlegenden Idee des Phasenmodells (siehe Abbildung 41).

Abbildung 41: Phasenmodell

- Die Phasen dieses Modells sind in unterschiedlicher Form in jedem Prozessmodell enthalten und können somit als kleinster gemeinsamer Nenner in jedem Software-Entwicklungsprojekt verstanden werden.
- Im Laufe der Lebensdauer eines Systems ist zumindest eine einmalige sequentielle Durchführung alles notwendigen Phasen vorgehsehen.
- Auf Änderungen wird erst im nächsten Projekt eingegangen, welche wieder jede Phasen durchläuft.
- **Nachteil:** Dieser vorgehen stell eine fehlende Möglichkeit der raschen Reaktion auf Änderungen der Anforderungen während der Durchführung des Projektes dar.

Bei den nachfolgenden Modellen handelt es sich um ***traditionelle Modelle.***

Traditionelles Modell: Wasserfallmodell nach Royce (1970)

- Löst teilweise das Problem der fehlenden Möglichkeit von Rückschritten zur raschen Reaktion auf Änderungen.
- Es sieht eine Abfolge der Phasen Anforderungen, Analyse, Entwurf, Implementierung, Test und Betrieb vor und lässt in einer Erweiterungsstufe einen ***Rückschritt*** von einer Phase ***auf den direkten Vorgänger*** zu.
- Ein Arbeitsschritt kann nur abgeschlossen werden, wenn alle vorgesehenen Produkte fertiggestellt worden sind → soll zu einer Minimierung des Risikos für die nächsten Phasen führen.

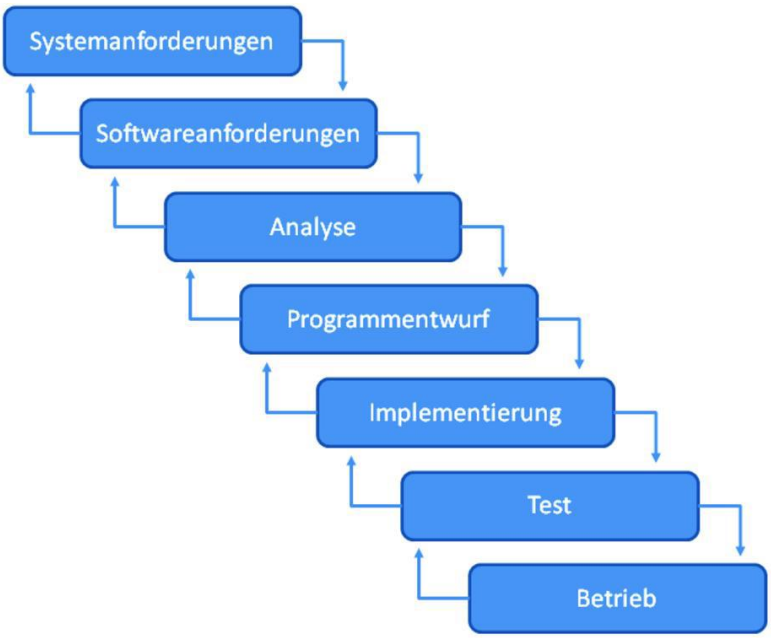


Abbildung 42: Wasserfallmodell nach Royce (1970) mit erweitertem Backtracking

- Ist das industriell am weitesten verbreitete Vorgehensmodell, von dem es viele verschiedene Ausprägungen und Varianten gibt.
- Das Modell ist in seiner Grundform ***vollständig sequentiell***.
- Diese Charakteristik wird in der modernen Softwareentwicklung als inhärentes Risiko betrachtet und hat zur ***Entwicklung der iterativen Modelle*** geführt.

Traditionelles Modell: Fundamentales V-Modell nach Böhm

- Das von Böhm vorgestellte Modell ist die Basis für viele aktuelle Entwicklungsmodelle, die im öffentlichen Sektor zum Einsatz kommen (V-Model 97, V-Modell XT, BVM)
- Das V-Modell ist auf folgenden Eigenschaften aufgebaut:
  - ✓ **Korrelierende Phasen:**
    - ➢ Ist wie das Wasserfallmodell sequenziell aufgebaut
    - ➢ Für jede konstruktive Phase auf der linken Seite existiert eine korrelierende prüfende Phase auf der rechten Seite
    - ➢ Hier wurden zum ersten Mal Teststufen integriert
  - ✓ **Validierung und Verifikation:**
    - ➢ Das V-Modell unterscheidet zwischen den prüfenden Phasen der Validierung (=wurde passendes Produkt für Einsatzzweck gebaut) und der Verifikation (=wurde Projekt nach den Vorgaben der vorherigen Phasen gebaut?)

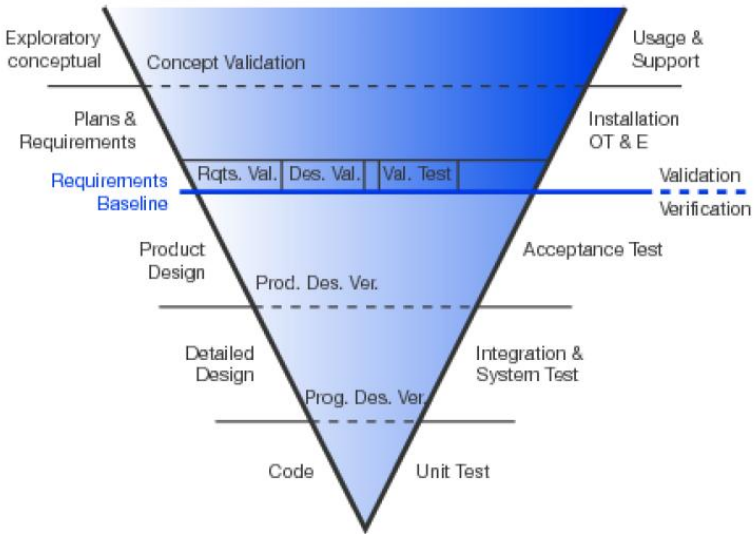


Abbildung 43: fundamentales V-Modell nach Böhm (1979)

V-Modell 97 – Entwicklungsphilosophie (S401, VO8 S12)

- Die Entwicklung im V-Modell 97 erfolgt inkrementell.
- Zu Beginn wird die Gesamtfunktionalität abgegrenzt und die einzelnen Funktionen des Softwaresystems werden priorisiert
- Die einzelnen Funktionen werden dann gemäß ihrer Priorität zu Ausbaustufen zugeordnet und im Rahmen dieser umgesetzt
- Dabei muss jeweils die nächste zu implementierende Ausbaustufe detailliert beschrieben werden.
- Das Ergebnis einer Stufe wird den Endanwendern zur Verfügung gestellt und ausgehend von deren Feedback erfolgt eine Überarbeitung des Gesamtplans und eine Detailplanung der nächsten Ausbaustufe
- Der Umfang des Systems wächst somit ausgehend von einer Basisfunktionalität stetig an

V-Modell XT („extreme Tailoring“) – 21.07.2015 v2.0 – Entwicklungsphilosophie (VO08 S12)

- Skalierbarkeit hinsichtlich verschiedener Projektgrößen (reduzierte Menge an Aktivitäten und Produkte für kleinere Projekte)
- Mehr Orientierung Richtung agilem Vorgehen: keinerlei Vorschriften über die zeitliche Abfolge von Vorgehensbausteinen

Traditionelles Modell: Spiralmodell nach Böhm (1988)

- Das Spiralmodell berücksichtigt die möglichen Projektrisiken der sequentiellen Entwicklung und ist daher iterativ aufgebaut.
- der gesamte Prozess ist in vier Phasen gegliedert
- Alle Phasen werden im Rahmen einer evolutionären Softwareentwicklung mehrmals durchlaufen d.h., dass in jeder Iteration nur Produkte entwickelt werden, die auf den Produkten des vorhergehenden Durchlaufs basieren und als Grundlage für den nächsten Durchlauf dienen
- Die 4 Phasen und ihre Aktivitäten sind:

(1) <u>Definition von Zielen</u>
- In jeder Phase wird mit der Definition von genauen Zielen und Produkten des Durchlaufs begonnen.
- Auch mögliche Alternativen und Einschränkungen werden analysiert.

(2) <u>Risikoanalyse</u>
- Zuvor identifizierte Ziele und Alternativen werden unter Berücksichtigung der Einschränkungen bewertet und mögliche Risiken festgestellt
- Für die Risiken werden anschließend Vermeidungs- und Lösungsstrategien entwickelt

(3) <u>Durchführen der Arbeitsschritte</u>
- Es werden die vorgesehen Produkte erstellt
- Abhängig von der Art des Projektes und der identifizierten Risiken, können die Anzahl und die Reihenfolge der Produkte in den einzelnen Durchläufen variieren.

(4) <u>Planen der nächsten Phase</u>
- Nach Abschluss jedes Durchlaufs werden Reviews durchgeführt.
- Ergebnisse und Erkenntnisse der Reviews bilden die Grundlage für die Planung des nächsten Zyklus.

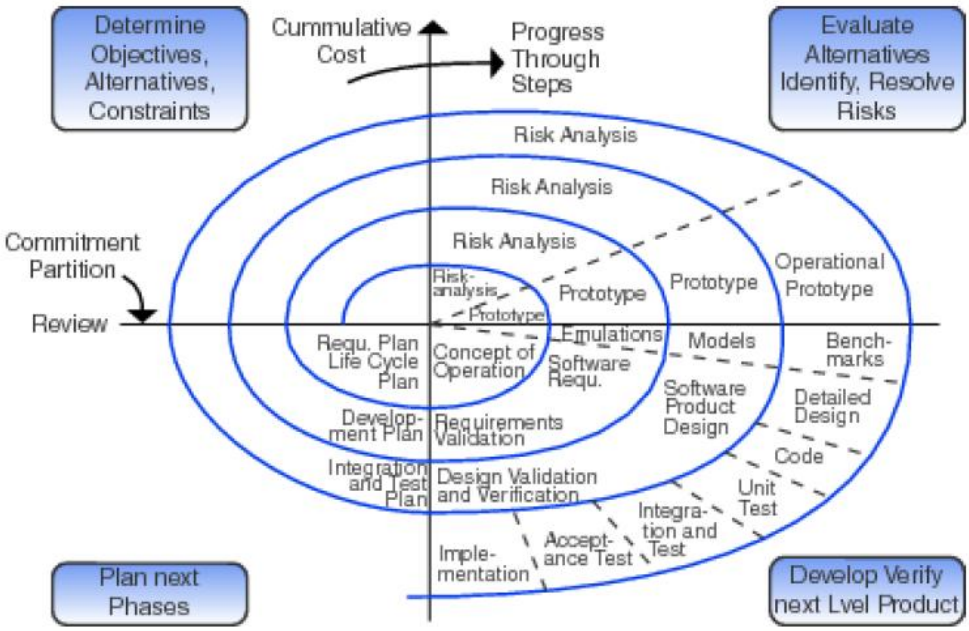


Abbildung 44: Spiralmodell nach Böhm

In Abbildung 44 sieht man ein Spiralmodell mit vier Durchläufen. Der vierte Durchgang ist für die Erstellung des Systems verantwortlich (wird erst durchlaufen, wenn alle Risiken beseitigt sind).

Vergleich Wasserfallmodell und iterative Entwicklung (S377-378, VO8 S15)

- Der Vergleich zwischen dem Wasserfallmodell und einer iterativen Entwicklung zeigt, dass man mit dem ***Wasserfallmodell*** rascher ***integrieren*** kann und in den früheren Phasen des Projektes aktiver ist.
- Durch die verspätete Validierung, können größere Änderungen zum späten Designbruch (***Late Design Breakage***) führen.
- Das System wird destabilisiert und es dauert dann umso länger einen finalen Test (***Final Quality Test***) zu bestehen.
- Die ***iterative Entwicklung*** baut auf ***inkrementelle Builds*** und Zwischenvalidierungen auf
- Probleme und Änderungswünsche werden früher erkannt und das Risiko einer Fehlentwicklung wird minimiert.

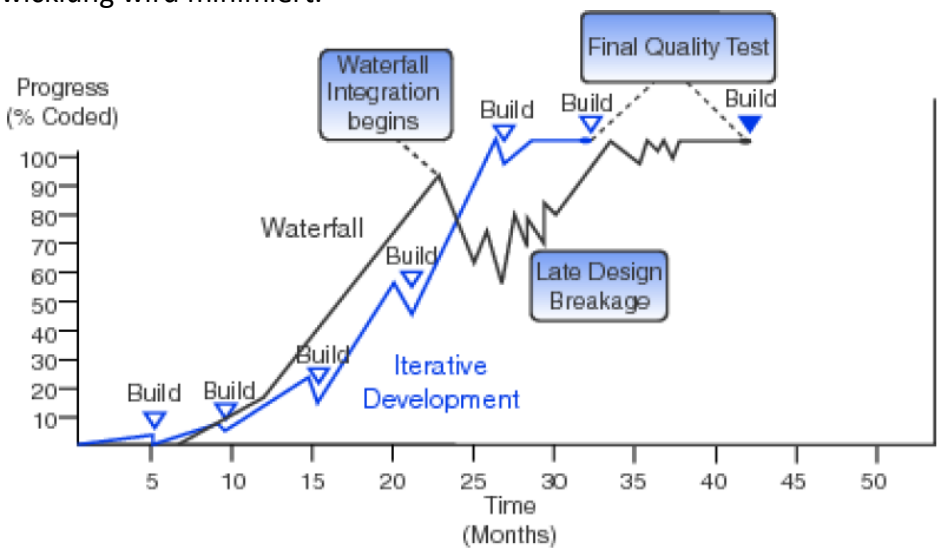


Abbildung 45: Vergleich Wasserfallmodell und iterative Entwicklung

Beispiel iterative Entwicklung im Projekt LiSaMe (BS378, VO8 S16)

- Die Iterationslänge kann variabel oder fix definiert sein.
- In vielen Projekten wird eine Iterationslänge von maximal vier Monaten und minimal zwei Wochen definiert
- In der Abbildung sieht man die Arbeitsaufträge zugeordnet zu den Iterationen im Drei-Monatszyklus. Hier wurde eine verhältnismäßig lange Iterationsdauer gewählt.

Abbildung 46: Beispiel für eine iterative Entwicklung

Industriell geprägte Prozessmodelle (S379-387, VO8 S17-20)

- Diese Modelle und deren Entstehung von industriellen Organisationen beeinflusst bzw. deren Weiterentwicklung von Unternehmen vorangetrieben wurde.

Rational Unified Process (RUP) 1998

- Beim Rational Unified Process handelt es sich um ein Prozess-Framework, dass von IBM Rational gepflegt, ständig weiterentwickelt und momentan in der Version 9.0 (2006) angeboten wird
- Seit seiner Einführung im Jahre 1999 wurde RUP mittlerweile zum De-facto-Standard im Bereich der Softwareentwicklung
- RUP beruht auf folgenden Vorgehensweisen, die sich bei vielen Projekten bewährt haben:
  - ✓ **Software iterativ entwickeln**
    - ➢ iterative Entwicklung bringt Vorteile wie früheres Überprüfen von Risiken
    - ➢ Berücksichtigung von neuer bzw. geänderten Anforderungen und
    - ➢ Anpassen von Plänen
  - ✓ **Anforderungen managen**
    - ➢ Erfassen und Dokumentieren sowie das Organisieren und Nachverfolgen von sich ändernden Anforderungen
  - ✓ **Komponentenbasierte Architektur verwenden**
    - ➢ Dadurch werden einerseits die Größe und die Komplexität der Architektur reduziert
    - ➢ andererseits wird diese aber auch stabiler und flexibler gestaltet.
  - ✓ **Software visuell modellieren**
    - ➢ Durch die Verwendung von Modellen können komplexe Systeme besser verstanden werden.
  - ✓ **Softwarequalität kontinuierlich prüfen**
    - ➢ Dadurch sollen die mit der Zeit steigenden Kosten der Behebung von Softwarefehlern minimiert werden
  - ✓ **Änderungen kontrolliert in die Software einbringen**
    - ➢ Das Managen, Nachverfolgen und Überwachen von Änderungen wird als wichtiger Bestandteil gesehen
- RUP kann durch die Eigenschaft iterativ, Use-Case-getrieben und architekturzentriert charakterisiert werden.

RUP – Entwicklungsphilosophie

- Die Vorgehensstrategien werden anhand von Abbildung 47 dargestellt.
- Bei RUP wird zwischen zwei Dimensionen unterschieden:
  - ✓ **Dynamische Dimension**
    - ➢ Die horizontale Dimension repräsentiert die zeitliche Struktur des Prozesses und kennzeichnet die Verteilung von Phasen, Iterationen und Meilensteinen auf den Entwicklungszyklus eines Softwareprojekts
  - ✓ **Statische Dimension**
    - ➢ Die vertikale Dimension stellt die inhaltliche Struktur des Prozesses dar und beschreibt wie die Prozesselemente (Aktivitäten, Artefakte und Rollen) logisch zusammenhängen und in sogenannte Dimensionen unterteilt werden.

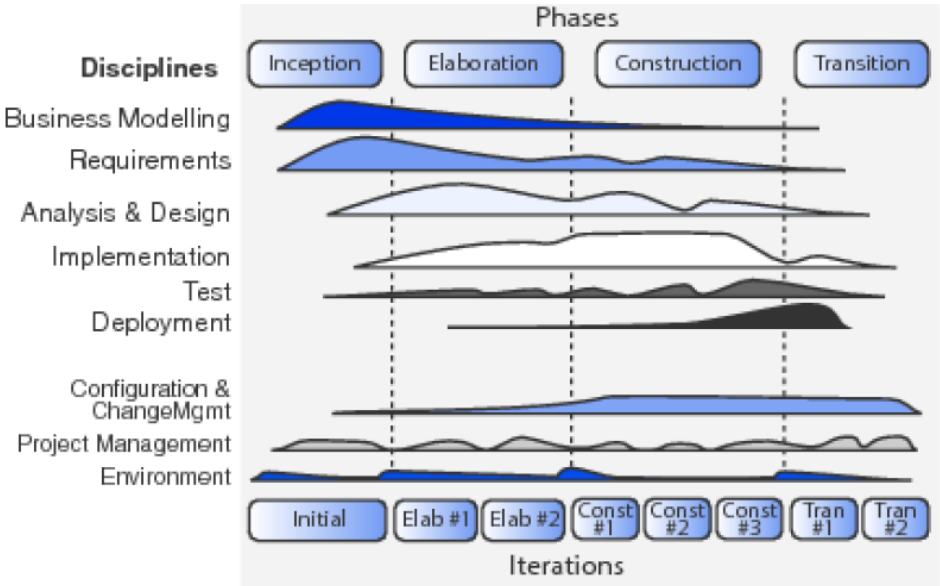


Abbildung 47: Die Struktur des RUP

Die vier Phasen lassen sich wie folgt darstellen:

(1) <u>Inception</u>

- in dieser Vorbereitungsphase geht es um die Erarbeitung der Projektvision und einer ersten Risikoabschätzung.
- Anforderungen an das System sollten zum Großteil bekannt sein und das Projektbudget sowie der Projektplan festgelegt werden

(2) <u>Elaboration</u>

- In der Ausarbeitungsphase geht es um die detaillierte Erhebung der Anforderungen, die Entwicklung einer prototypischen Architektur, das Testen von unterschiedlichen Lösungsansätze und die Vertiefung der Kenntnisse über Techniken und Werkzeuge des Lösungsbereichs

(3) <u>Construction</u>

- Das Entwickeln eines lauffähigen Systems und dessen Testung stellen den Schwerpunkt dieser Konstruktionsphase dar.
- Am Ende hat man ein produktiv einsetzbares Produkt

(4) <u>Transition</u>

- Im Vordergrund der Übergangsphase steht der Transfer des Softwaresystems zum Auftraggeber.
- Es werden Tests beim Kunden, Fehlerbehebungen, Anwenderschulung und das Komplettieren der begleitenden Materialien durchgeführt
- Endet mit dem Meilenstein Product Release.

RUP – Tailoring / Anpassungsmöglichkeiten (S381-382, VO8 S19)

- Da der RUP ein umfangreiches Framework darstellt, ist für jedes Projekt bzw. für jede Organisation eine ***selektive Adaption*** vorzunehmen, um den erwarteten Nutzen auch tatsächlich zu erzielen
- Für die Anpassung (Tailoring) des RUP bietet sich das kostenpflichtige Tool ***IBM Rational Method Composer (RMC)*** an mit u.a. vordefinierten RUP-Prozessmodellen

- Dieses Tool unterstützt das detaillierte Anpassen an spezifische Gegebenheiten und das Erzeugen eines komplett eigenen Modells bzw. das Mischen mit anderen Prozessmodellen
- Die Rolle, den RUP an die jeweiligen speziellen Rahmenbedingungen anzupassen, kommt dabei dem Process Engineer zu.

Bei den nachfolgenden Modellen handelt es sich um ***agile Modelle.***

Agile Modelle – von Personen geprägte Prozessmodelle (S387-398, VO 8 S21-33)

Auf Prozessmodelle, an deren Entstehung, Verbreitung und Weiterentwicklung bestimmte Personen(gruppen) hauptbeteiligt waren bzw. sind, wird in diesem Abschnitt eingegangen.

Scrum (S387-390, Folien S22-25)

- Laut Schwaber und Beedle (2002) bezeichnet der Begriff Scrum eine hyper-produktive Produktentwicklung.
- Die zentralen Werte dieses Prozessmodells werden folgendermaßen dargestellt:
  - ✓ hohe Produktivität und Anpassungsfähigkeit
  - ✓ wenig Risiko und Ungewissheit
  - ✓ größerer Komfort für die Projektmitglieder
- Scrum hat seine Wurzeln in der empirischen Prozesssteuerung, die im Gegensatz zur definierten Prozesssteuerung über ständige Rückkopplung geführt wird. Voraussetzung hierfür sind regelmäßige Überprüfungen und Adaptierungen sowie die Sichtbarkeit des aktuellen Zustandes.

Scrum – Entwicklungsphilosophie

- Abbildung 48 zeigt die Vorgehensstrategie von Scrum
- Alle funktionalen und nichtfunktionalen Anforderungen werden in einem ***Product Backlog*** gereiht und nach Wichtigkeit aufgelistet, wobei diese Liste während des Projekts ständig angepasst wird.
- Die eigentliche Entwicklung wird von einem kleinen ***Scrum Team*** (7 Personen +/-2 pro Team) durchgeführt.
- Das zentrale und wichtigste Element von Scrum ist der ***Sprint***. Dieser hat eine definierte Zeitdauer (in der Regel 30 Tage) und kann als die „Umsetzung einer Iteration“ bezeichnet werden. Jeder Sprint muss mit der Fertigstellung einer neuen Funktionalität enden.
- Zu Beginn eines Sprints findet ein ***Sprint Planning Meeting*** statt. Hier werden die Aufgaben basierend auf dem Product Backlog festgelegt und im ***Sprint Backlog***, der aus einzelnen ***Backlog Items*** besteht, fixiert.
- Das Team verpflichtet sich damit zur Umsetzung der in den Sprint Backlog aufgenommen Funktionalität.

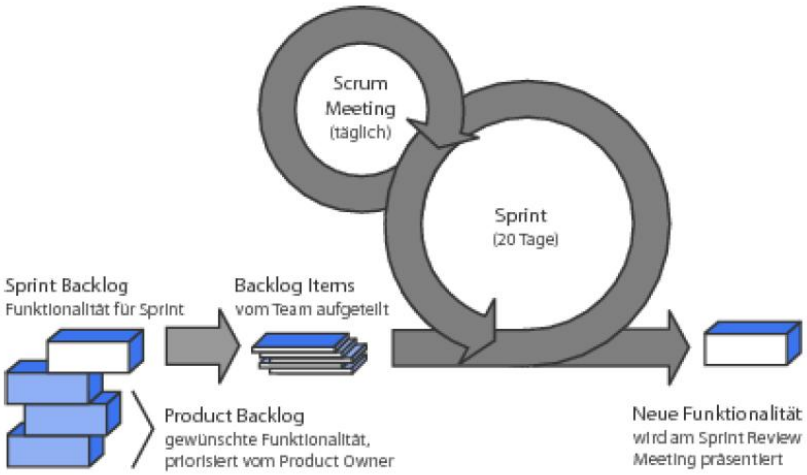


Abbildung 48: Vorgehensstrategie von Scrum

Scrum – Elemente

- **Sprint:** Innerhalb eines Sprints wird versucht, so schnell wie möglich zum zuvor definierten Sprint Goal zu kommen.
- **Daily Scrum:** Dieses Meeting findet immer zur selben Zeit am selben Ort statt.
- **Product Backlog:** Diese nach Dringlichkeit geordnete Liste aller Anforderungen kann von allen Projektbeteiligten jederzeit eingesehen werden
- **Increment:** Bei Scrum entsteht ein Softwareprodukt in Inkrementen, wobei schon der erste, aber spätestens der zweite Sprint ein lauffähiges Softwaresystem (mit Kernfunktionalität) liefert.
- **Burndown Chart:** In dieses Diagramm werden vom Scrum Master tagesaktuelle Aufwandsschätzungen für den aktuellen Sprint eingetragen um so den Fortschritt des Projekts zu verfolgen und möglichst bald Probleme und Hindernisse erkennen zu können.

Scrum – Rollen

- **Product Owner:** Neben der Hauptaufgabe, der Erstellung und Priorisierung des Product & Release Backlogs, ist diese Person wesentlich dafür verantwortlich, dass das Team frei von Störungen und Unterbrechungen bleibt.
- **Scrum Team:** Die Mission des Teams ist die Umsetzung der über den Sprint Backlog geforderten Funktionalität bzw. die Erreichung des über das Sprint Goal bestimmten Ziels.
- **Scrum Master:** Die wesentliche Aufgabe dieser Rolle besteht darin, die Dynamik von Scrum zur ungestörten Entfaltung zu bringen, indem der Scrum Master einerseits auf die Umsetzung der Grundsätze und Regeln von Scrum achtet und andererseits das Team vor äußeren Einflüssen abschottet

eXtreme Programming (XP) (Buch S391-398, VO8 S26-29)

- Am besten kann XP als „eine kompakte Methode zur Entwicklung von Software in kleinen bis mittelgroßen Teams, deren Arbeit vagen oder sich rasch ändernden Anforderungen unterliegt“ (Beck, 2003) beschrieben werden.
- Als Hauptziel wird das termingerechte Abliefern von auftragsgemäßer Software gesehen
- Charakteristische Merkmale von XP sind das eigene Wertesystem, viele Prinzipien und ein System von Techniken, die einander gegenseitig in höchstem Maße unterstützen.

eXtreme Programming – Prinzipien & Techniken

- Das Wertesystem von XP, also die gemeinsamen Wertvorstellungen der Mitarbeiter, besteht aus folgenden Komponenten:
  - ✓ **Kommunikation:** ständige Kommunikation zwischen allen Beteiligten (Entwicklern, Kunden, Sponsoren, Usern, ...)
  - ✓ **Einfachheit:** Das einfachste Design welches die Aufgabe löst
  - ✓ **Feedback:** Wissen wo man steht und was man verbessern kann (empirisches Prozessmodell)
  - ✓ **Mut:** zur Offenheit und Transparenz; notwendige Änderungen durchzuführen
  - ✓ **Respekt:** vor sich selbst, dem Team und dem Produkt

- Diese vier Werte werden durch 5 Prinzipien (unmittelbares Feedback, Einfachheit anstreben, Inkrementelle Veränderung, Veränderung wollen, Qualitätsarbeit) und 12 Techniken unterstützt. Abbildung 49 zeigt einen Überblick über die Techniken und deren Verbindungen.

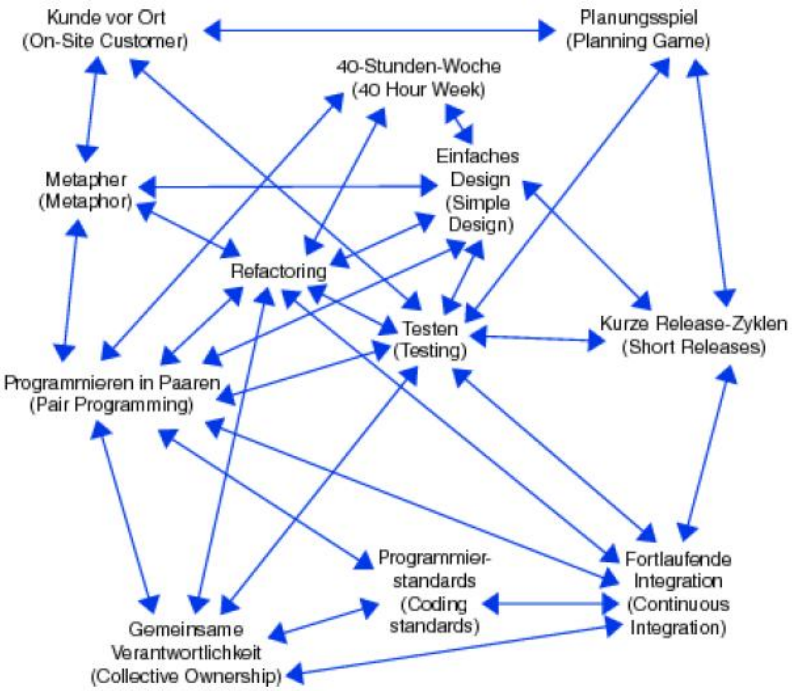


Abbildung 49: Zusammenhang der XP-Techniken

eXtreme Programming – Entwicklungsphilosophie

- Die Vorgehensstrategie wird in Abbildung 50 veranschaulicht.
- Die ***Anforderungen*** werden zu Projektbeginn mittels ***User Stories*** festgehalten, sowie ein Prototyp hinsichtlich der Architektur erstellt (***Architectural Spike***).
- Innerhalb der ***Release Planung*** (Planungsspiel) wird der Aufwand für alle User Stories geschätzt, die in der nächsten umzusetzenden Anforderung priorisiert.
- User Stories werden in mehreren ***Iterationen*** umgesetzt (Release Plan muss eventuell angepasst werden)
- Der Code, der nach der Iteration entsteht, wird mittels ***Akzeptanztests*** geprüft und entweder in der nächsten Iteration korrigiert oder als ***Release*** freigegeben.

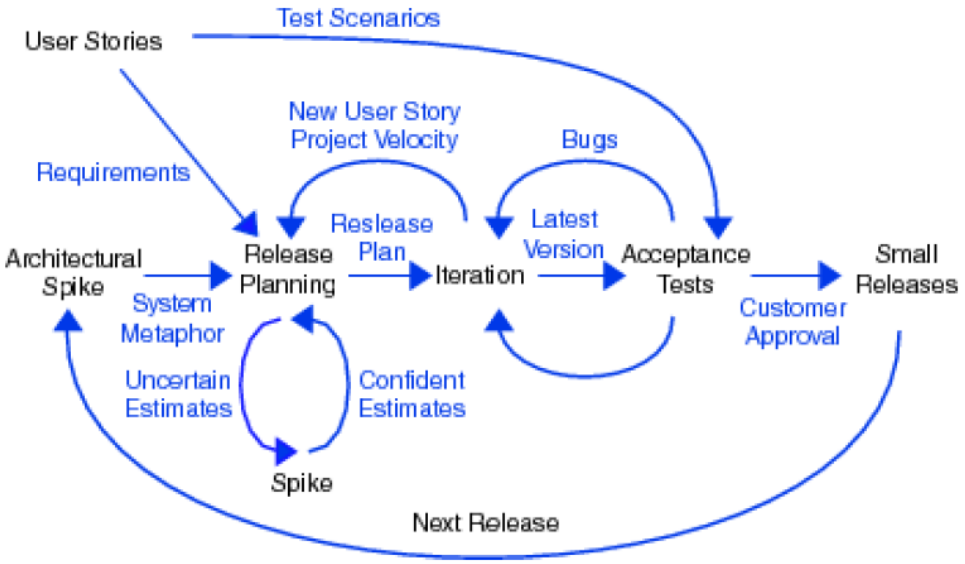


Abbildung 50: Ein XP-Projekt nach Beck (2003)

XP@Scrum

In der Praxis sehr beliebt:

- Scrum Prozess inkl. Meetings, Rollen und Vorgehensmodell
- Zusätzliche Feedbackschleifen über XP-Techniken:
  - ✓ Pair Programming
  - ✓ Code Reviews
  - ✓ Continuous Integration
  - ✓ etc.
- Scrum als Vorgehensmodell, erweitert mit XP Entwicklungstechniken

Kanban (VO8 S31-32)

- Lean Management/Thinking: Ursprünglich aus der Automobilindustrie (1978 Toyota Production System)
- Von Poppendieck 2003 für die Softwareentwicklung adaptiert:
  - ✓ Eliminate waste
  - ✓ Amplify learning
  - ✓ Decide as late as possible
  - ✓ Deliver as fast as possible
  - ✓ Empower the team
  - ✓ Build integrity in
  - ✓ See the whole
- Keine vorgeschriebenen Rollen, Meetings, Artefakte, außer:
  - ✓ Kontinuierlicher “Pull” Workflow
  - ✓ Kanbanboard
  - ✓ Work In Progress Limit

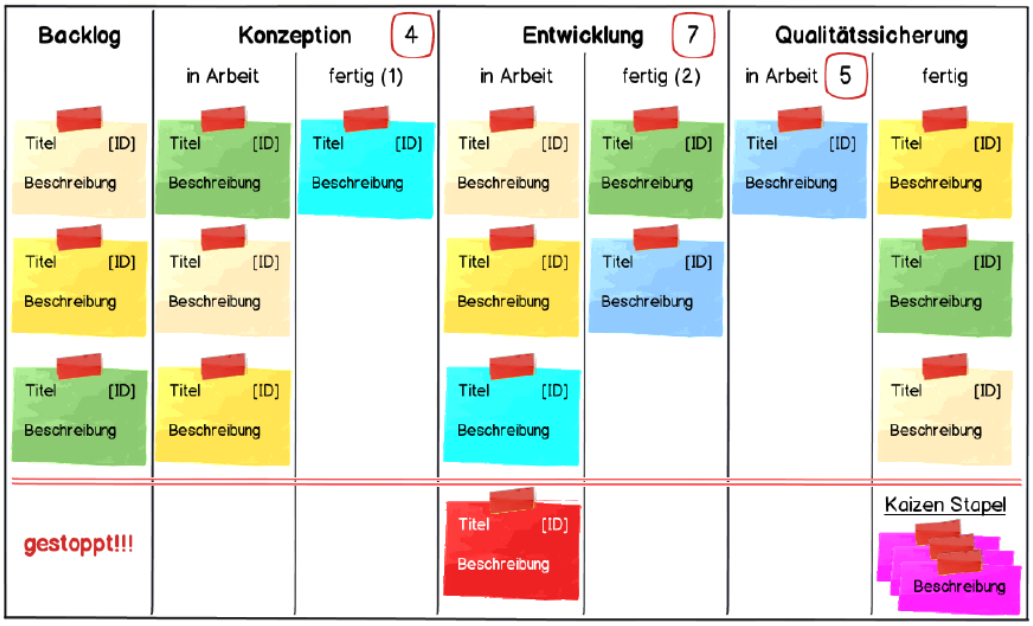


Abbildung 51: Kanban Board Beispiel

Agile Versus traditionelle Vorgehensmodelle (Buch S409-412, VO8 S34-36)

- Weder agile noch plangetriebene Methoden stellen eine „Silver Bullet“ (Allheilmittel, keine allgemein gültige Methode) dar
- Sowohl agile als auch plangetriebene Methoden haben für gewisse Einsatzgebiete deutliche Vorteile gegenüber dem anderen Modell. Zum Beispiel sind agile Methoden für Softwareentwicklungsprojekte mit einer raschen time-to-market vorzuziehen
- In Zukunft werden sowohl agile als auch plangetriebene Aspekte eine wesentliche Rolle in der Applikationsentwicklung spielen
- Methoden werden in der Zukunft ausbalanciert sein
- Es ist besser, eine Methode für seine eigenen Bedürfnisse zu ergänzen als eine Methode zu reduzieren
- Methoden sind wichtig, aber potenzielle „Silver Bullets“ liegen im Bereich „Personen“, „Werte“, „Kommunikation“ und „Management der Erwartungen“.

- Böhm und Thurner haben sich weiter der Fragestellung angenommen, welches Maß an Agilität in einem Softwareentwicklungsprojekt verträglich ist und ein entsprechendes Entscheidungsmodell entwickelt (siehe Abbildung 52).
- Je mehr man in den jeweiligen Entscheidungsfaktoren die Mitte erreicht, desto eher würde sich die Anwendung von agilen Methoden eignen.
- Folgende Entscheidungsfaktoren sind in diesem Modell bestimmt:
  - ✓ **Teamstruktur (Personnel):** Boehm und Thurner nehmen in ihrem Modell an, dass die Qualifikation des Teams bzw. der beteiligten Personen beim Einsatz agiler Methoden höher sein muss als beim Einsatz plangetriebener Methoden.
  - ✓ **Dynamik der Anforderungen (Dynamism):** Je häufiger Anforderungen geändert werden, desto einfacher muss dies möglich sein.
  - ✓ **Entwicklungskultur (Culture):** Je nach organisatorischem Umfeld bzw. Entwicklungskultur können agile Methoden eingesetzt werden.
  - ✓ **Teamgröße (Size):** Es wird oftmals argumentiert, dass agile Methoden in kleinen bis mittelgroßen Teams effizient einsetzbar sind.

- ✓ **<u>Kritikalität (Criticality):</u>** Wenn Menschenleben von einem Softwaresystem abhängig sind, werden natürlich viel höhere Maßstäbe an die rigorose Prüfung und Nachvollziehbarkeit auf allen Ebenen gelegt.

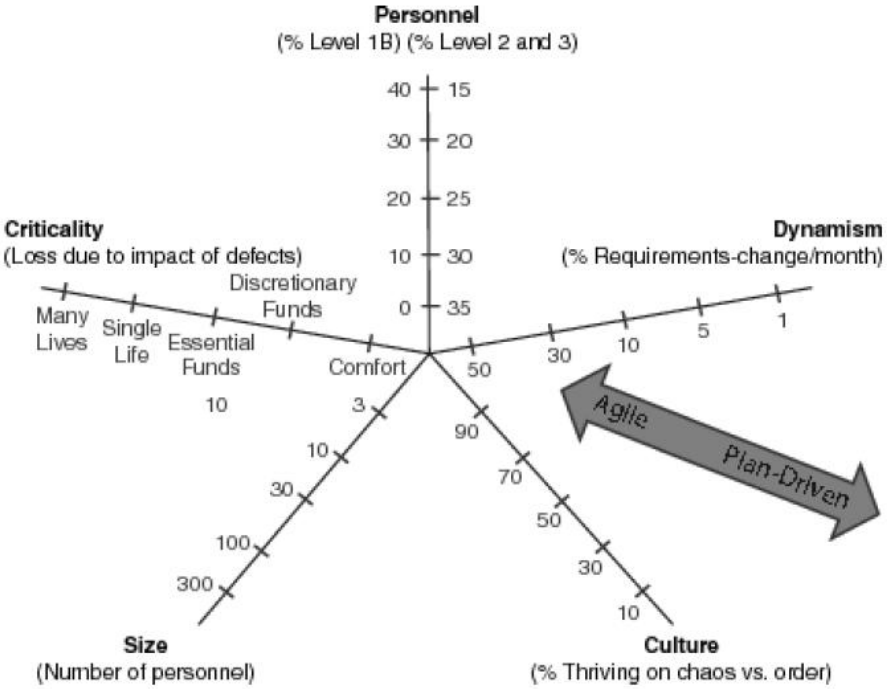


Abbildung 52: Entscheidungsmodell nach Böhm und Thurner (2003)

Qualitätssicherung und Qualitätsmanagement

Grundlagen der Softwarequalitätssicherung (S460, VO8 S38)

Eine angemessene Softwarequalität entsteht durch ein systematisches Vorgehen nach bewährten Techniken der Softwarekonstruktion. Unterschiedliche Maßnahmen, kommen hier zur Anwendung (siehe Abbildung 53).

- ***Organisatorische Maßnahmen:*** schaffen die notwenigen Rahmenbedingung und sorgen dafür, dass analytische Maßnahmen stattfinden können.
- ***Analytische Maßnahmen:*** Versuchen Fehler und Mängel zu erkennen und zu lokalisieren.

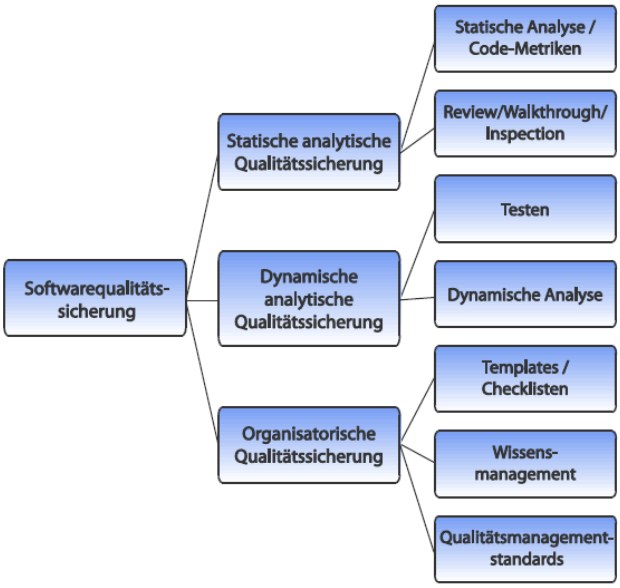


Abbildung 53: Software-Qualitätssicherungsmethoden

Softwarequalitätsfaktoren (S463, VO39)

- Mit Hilfe von verschiedenen Qualitätsfaktoren versucht man Aussagen über die Qualität von Software zu treffen. Einen Einsatz, Qualitätsfaktoren zu definieren und in Gruppen zusammenzufassen, beschreibt die ISO/IEC 9126 Norm. Folgende Gruppen von Qualitätsfaktoren werden beschrieben:
  - ✓ **Funktionalität:** Gruppe von Attributen, die sich auf das Vorhandensein von Funktionen mit festgelegten Eigenschaften bezieht. Das sind also jene Anforderungen, die spezifizierte oder implizite Anforderungen erfüllen.
  - ✓ **Zuverlässigkeit:** Gruppe von Attributen, die sich darauf beziehen, ob Software ein bestimmtes Leistungsniveau unter bestimmten Bedingungen über einen bestimmten Zeitraum aufrechterhalten kann.
  - ✓ **Benutzbarkeit:** Gruppe von Attributen, die den erforderlichen Aufwand für den Einsatz der Software von den Benutzern beschreibt. Damit ist der Aufwand gemeint, der zur Benutzung erforderlich ist.
  - ✓ **Effizienz:** Gruppe von Attributen, die das Verhältnis zwischen Leistungsniveau der Software und eingesetzten Betriebsmitteln beschreibt.

- ✓ **Änderbarkeit:** Gruppe von Attributen, die den erforderlichen Aufwand für die Durchführung vorgegebene Änderungen an der Software beschreibt.
- ✓ **Übertragbarkeit:** Gruppe von Attributen, die beschreibt, wie einfach sich die Software in eine andere Umgebung migrieren lässt.

Ursachen von Softwarefehlern (S466-67, VO8 S40)

- Fehlerhafte Definitionen in den Anforderungsdokumenten, üblicherweise durch den Kunden bereitgestellt, sind die Hauptursache von Softwarefehlern
- Falsch definierte Anforderungen, fehlende zentrale Anforderungen, unvollständige Anforderungen und nicht benötigte Anforderungen
- Missverständnisse durch eine nicht funktionierende Kunde-Entwickler-Kommunikation, können Fehler verursachen.
- falsche Interpretation von Kundenanforderungen
- Falsche Interpretation von Kundenrückmeldungen zu Designproblemen.
- Logische Designfehler aufgrund fehlerhafter Anforderungen von Designexperten, Softwareentwickler und Analysten
- Fehlinterpretationen des Designdokuments
- Fehler bei der Test-Datenauswahl
- Nichteinhaltung von Coding Standards

Fehlerentstehung und Fehlerentdeckung (S467-468, VO8 S41)

- der Softwareentwicklungsprozess wird in verschiedenen Phasen unterteilt.
- Fehler können in allen Phasen der Entwicklung entstehen.
- Fehlerbehebungskosten steigen exponentiell mit der Latenzzeit eines Fehlers.
- Kosten für die Behebung eines in der Analyse der Anforderungsbeschreibung gemachten Fehlers steigen auf das Mehrfache, wenn er nicht sofort, sondern erst in einer späteren Phase gefunden wird.
- Abbildung 54 zeigt, dass Fehler der früheren Phasen bei weitem teurer sind und deshalb mit konstruktiven Maßnahmen verhindert werden sollten.

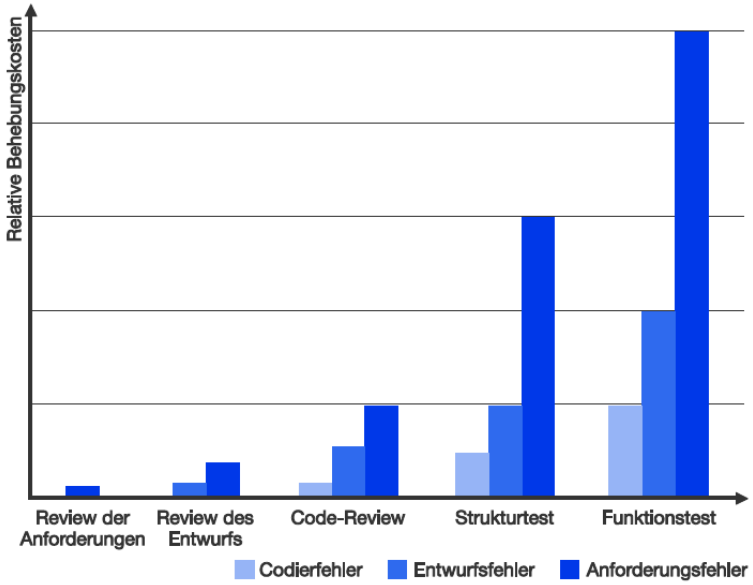


Abbildung 54: Relative Fehlerbehebungskosten

Kosten der Qualitätssicherung

- Die Kosten setzen sich aus den Herstellungskosten und den Qualitätskosten zusammen.
- Herstellungskosten beinhalten all jene Kosten, welche zur Erbringung der geforderten Leistung und deren Qualität notwendig sind.
- Qualitätskosten umfassen alle zusätzlichen Aufwendungen für das Vermeiden, Erkenne, Lokalisieren und Beheben von Fehlern.
- Ein wichtiges Ziel in der Softwareproduktion ist daher die Vermeidung von Fehlern und deren Folgekosten.
- Feigenbaum (1951; 2004) eine Qualitätskostenmodell entwickelt. Dieses Modell sieht eine Aufteilung in Prüf- und Fehlerkosten vor. Prüfkosten stellen sämtliche Kosten dar, die für die Prüfung und Fehlervermeidung entstehen. Fehlerkosten sind Kosten, die aufgrund von Fehler entstehen. Hier unterscheidet man zwischen internen und externen Fehlerkosten

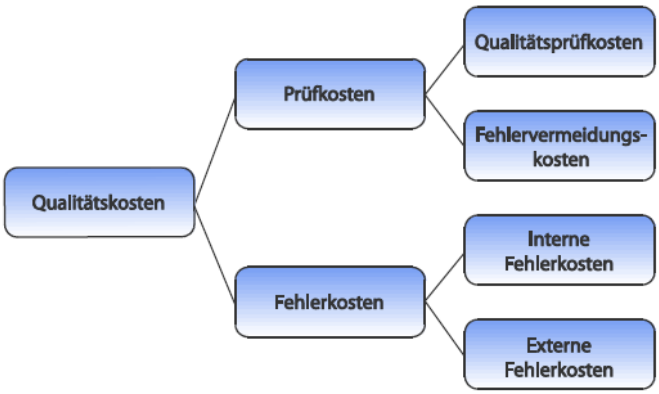


Abbildung 55: Qualitätskostenmodell nach Feigenbaum

Kostenoptimierung

- Laufende Qualitätsverbesserung muss auch unter ökonomischen Gesichtspunkten betrachtet werden.
- Es muss also jener Punkt gefunden werden, an dem das Verhältnis zwischen Fehlerkosten, Fehlervermeidungskosten und Qualität optimal ist (siehe Abbildung 56)

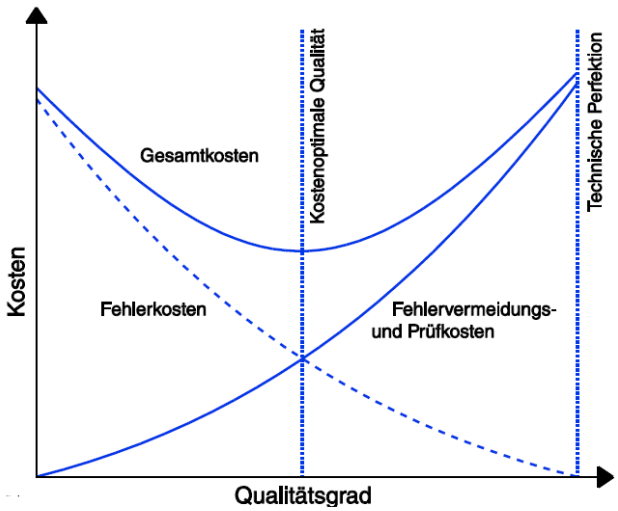


Abbildung 56: Kostenoptimale Qualität

- Die Qualität steigt mit zunehmender ***Fehlerverhütungs- und Prüfkosten***. Die ***Kosten für*** die Behebung von ***Fehlern*** verringern sich
- ***Kostenoptimale Qualität*** wird erreicht, wenn die Kosten für die Vermeidung und die Prüfung von Fehlern genauso hoch sind wie die Kosten für deren Behebung. Dies ist auch der Punkt, an dem die ***Gesamtkosten*** am geringsten sind.

Abbildungsverzeichnis

Abbildung 1: Beispielsprozess zur Prüfung und Erteilung von 4
Abbildung 2: Interface der Aktenbearbeitung 5
Abbildung 3: Verwendete Technologien 7
Abbildung 4: Ausgangs- und Zielschema von HISS 11
Abbildung 5: Featureliste (Initialplanung) 11
Abbildung 6: Übersicht der Projektphasen 12
Abbildung 7: Struktur des Projektes aus organisatorischer Sicht 13
Abbildung 8: Stakeholder 13
Abbildung 9: Fachkonzept Pipeline 14
Abbildung 10: Re-Dokumentation 16
Abbildung 11: Zugriffs- und Traffic Analyse 16
Abbildung 12: Modularisierung 17
Abbildung 13: Projektumfeld 20
Abbildung 14: Anwendungsfallbeschreibung & -diagramm 31
Abbildung 15: Beispiel User Card Story 31
Abbildung 16: Entwurfsparadigmen - Zustandsdiagramm 36
Abbildung 17: Fork und Stair 37
Abbildung 18: Lastenheft & Pflichtenheft 37
Abbildung 19: Architektur abstrakt 39
Abbildung 20: Architektur real mit Infrastruktur 39
Abbildung 21: Entwurf im V-Modell 41
Abbildung 22: 2-/3-Tier und Service-oriented Architektur 42
Abbildung 23: Serviceorientierter Entwurf 42
Abbildung 24: Business Service zusammengesetzt aus technischen Services 44
Abbildung 25: Model View Controller 45
Abbildung 26: Version Control - Branching 48
Abbildung 27: Zentrales Versionsmanagement 48
Abbildung 28: Dezentrales Versionsmanagement 48
Abbildung 29: Klassendiagramm eines Observer Pattern 50
Abbildung 30: Testen 51
Abbildung 31: Test Driven Development 52
Abbildung 32: Test- und Integrationsstufen 54
Abbildung 33: Komponententest - Ausgangssituation 55
Abbildung 34: Komponententests - Test-Setup 55
Abbildung 35: Big-Bang-Integration 56
Abbildung 36: Top-Down-Integration 57
Abbildung 37: Bottom-Up-Integration 57
Abbildung 38: Vertikale Integration 58
Abbildung 39: Strukturelle Abdeckung 59
Abbildung 40: Steuergrößen eines Projekts 65
Abbildung 41: Phasenmodell 65
Abbildung 42: Wasserfallmodell nach Royce (1970) mit erweitertem Backtracking 66
Abbildung 43: fundamentales V-Modell nach Böhm (1979) 67
Abbildung 44: Spiralmodell nach Böhm 68

Abbildung 45: Vergleich Wasserfallmodell und iterative Entwicklung 69
Abbildung 46: Beispiel für eine iterative Entwicklung 69
Abbildung 47: Die Struktur des RUP 71
Abbildung 48: Vorgehensstrategie von Scrum 73
Abbildung 49: Zusammenhang der XP-Techniken 74
Abbildung 50: Ein XP-Projekt nach Beck (2003) 75
Abbildung 51: Kanban Board Beispiel 76
Abbildung 52: Entscheidungsmodell nach Böhm und Thurner (2003) 77