Software Engineering

Zusammenfassung

HTA Horw

Rainer Meier
Käserei
6288 Schongau
skybeam@skybeam.ch

Klasse: 4
2002 - 2006

2005-09-20

1. Inhaltsverzeichnis

2. Einführung

Software Engineering ist die Entwicklung, Pflege und der Einsatz von qualitativ hochstehender Software. Dies wird erreicht durch den Einsatz wissenschaftlicher Methoden, wirtschaftlicher Prinzipien, geplanter Vorgehensmodelle, Werkzeugen und quantifizierbaren Zielen.

2.1. Prinzipien der Software-Entwicklung

Prinzipien sind die Regeln und Grundsätze, dei man seinem Handeln zugrundelegt.

Prinzipien der Software-Entwicklung:

- Abstraktion
- Strukturierung
- Hierarchisierung
- Modularisierung
- Lokalität
- Integrierte Doku
- Standardisierung
- Adäquate Typen
- Verbalisierung
- Lineare Strukturen

2.2. Methoden der Implementierung

Die Methoden sind die auf einem Regelsystem aufbauenden Verfahren. Durch planmässiges Vorgehen führen diese Methoden zum Erlangen von Erkenntnissen und praktischen Ergebnissen.

- Stepwise Refinement
    - Datentypen & Funktionen
    - To-down Entwurf
- Modularer Entwurf
    - Decomposition
    - Coupling & cohesion
    - Information hiding
- Object-Oriented Design (OOD)
    - Abstract data types
    - Inheritance
    - Late binding
- XP
    - Metapher & simple design
    - Test & refactoring
    - Pair-programming

2.3. Aufwandsverteilung

Das Ziel der Methodischen Entwicklung ist die Verlängerung der Lebensdauer , die Reduktion der Kosten in der Wartungsphase und die frühzeitige Erkennung von Fehlern. Dies wird erkauft durch einen möglicherweise höheren Aufwand in der Anfangsphase.

2.4. Erfolg von Softwareprojekten

Laut Statistik wird 47% der Software zwar ausgeliefert aber aus diversen Gründen nie verwendet. Dies liegt häufig daran, dass sich der Kunde das Produkt nicht so vorgestellt hat wie es geliefert wurde oder gar nicht damit arbeiten kann.

Nur 2% der Software wird so benutzt wie sie geliefert wurde.

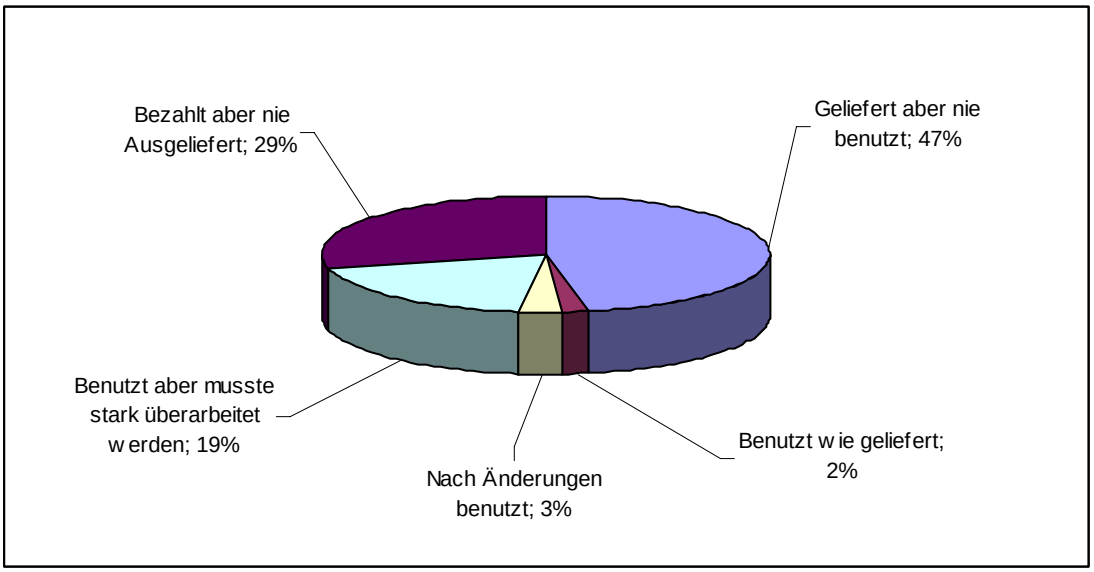


Abbildung 1 Erfolg von Softwareprojekten

Weitere Probleme:

- 66% aller Projekte werden später fertig als geplant
- 58% aller Projekte haben gravierende Probleme
- 55% aller Projekte benötigen mehr Aufwand als geplant
- 50% aller Projekte überschreiten das Zeit- und Kosten-Budget

2.4.1. Hauptursachen für die Entwicklungsprobleme

Häufig werden die Benutzeranforderungen falsch verstanden oder nur unpräzise aufgenommen. Die Kommunikation innerhalb des Projektes ist unzureichend. Häufig ist man unfähig sich mit geänderten Anforderungen auseinander zu setzen. Die entwickelten Module bzw. Komponenten passen nicht zusammen und die Qualität ist zu gering. Häufig ist auch die Performance der Software unzureichend.

Die Gründe liegen häufig in der instabilen Architektur, der zu grossen Komplexität und der Inkonsistenz zwischen Anforderungen, Design und Realisierung. Diese Probleme wirken sich umso schwere aus wenn der Projektstatus ungenügend verfolgt wird und die Software ungenügend getestet wird.

2.5. Projektvariabeln

Jedes Projekt unterliegt gewissen Ressourcen-Beschränkungen. Ein Projekt lässt sich grob in vier Dimensionen definieren: Kosten, Kompatibilität, Fehlerhäufigkeit und Dauer.

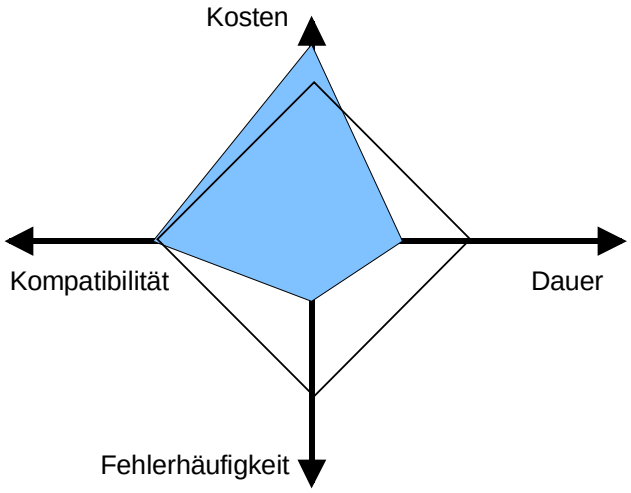


Abbildung 2 Bullseye

2.6. Die 4 P's der Software Entwicklung

Bei der Software-Entwicklung muss man insbesondere wert auf die 4 P's legen: Personen (People), Prozess (Process), Projekt (Project) und Produkt (Product).

2.6.1. Personen

Es ist wichtig die Verantwortlichkeiten festzulegen. Die Personen sollen selbständig arbeiten können. Es ist darauf zu achten, dass die Personen richtig ausgebildet sind um ihre Aufgabe zu erfüllen.

2.6.2. Prozess

Ein geeignetes Prozessmodell muss gefunden und ausgewählt werden. Entweder ein lineares Modell wie das Wasserfall-Modell oder ein iteratives. Dabei muss zwischen dem Personellen Software Prozess und dem Team Software Prozess unterschieden werden. Der Weiterbildungsgrad kann nach dem Capability Maturity Model erfasst werden.

2.6.3. Projekt

Ein Projekt ist eine Ansammlung von Aktivitäten um eine Anwendung zu erstellen. Ein Projekt hat die folgenden Eigenschaften:

- Zeitlich begrenzt
- Löst probleme innerhalb eines vorgegebenen Zielsystems
- Umfasst die Gesamtheit der für die Problemlösung notwendigen Entwicklungsarbeiten

2.6.4. Produkt

Als Produkt bezeichnet man die zur Applikation gehörenden „Artefakte". Artefakte können Dokumente, Modelle, Test Prozeduren, Testfälle und natürlich auch der dazugehörende Code und die Binärdateien sein.

2.7. Qualität

Die Qualität einer Software hängt ab von der Zeit, die zur Entwicklung zur Verfügung steht und dem finanziellen Rahmen. Daraus ergeben sich Qualitätsanforderungen. Einige davon sind für die Benutzer direkt von Bedeutung. Andere sind für die Entwickler und die Weiterentwicklung der Software von Bedeutung:

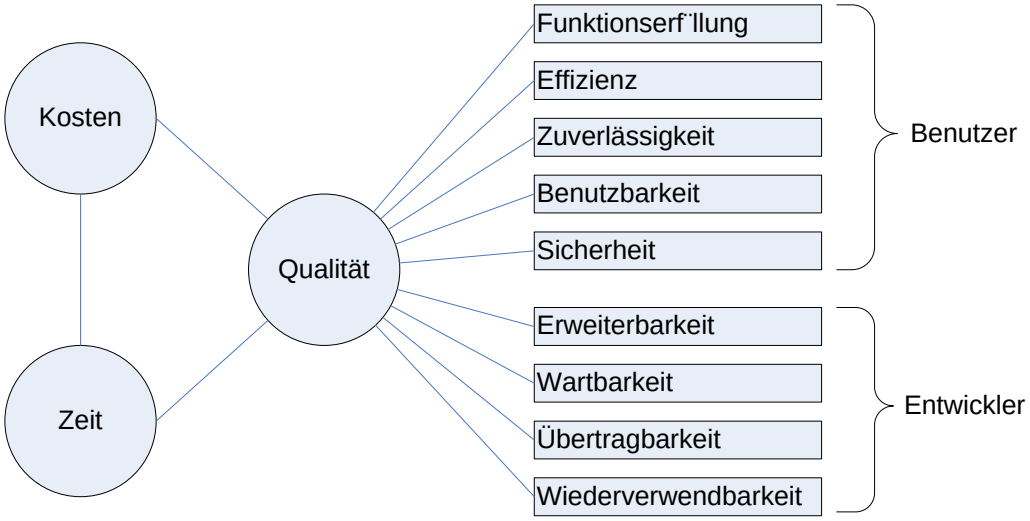


Abbildung 3 Qualität-Kosten-Zeit

2.7.1. Funktionserfüllung

- Grad der Übereinstimmung zwischen geplantem und realisiertem Funktionsumfang
- Kann man mit der Software das erledigen für das sie entwickelt wurde?
- Funktionale Anforderungen ändern häufig während dem Projektverlauf
- Nichtfunktionale Anforderungen: Sicherheit, Benutzbarkeit, Erweiterbarkeit, Wartbarkeit...

2.7.2. Effizienz

Die Effizienz gibt an wie viele Ressourcen (CPU, Hauptspeicher, Festplatte) durch die Software in Anspruch genommen wird.

2.7.3. Zuverlässigkeit

Zuverlässigkeit besteht aus der Erfüllung der geforderten Leistung ebenso wie der Sicherheit nicht in unerwünschte oder gar gefährliche Zustände zu geraten.

Ein zuverlässiges system ist korrekt, robust und hoch verfügbar.

Die Verfügbarkeit wird mit folgender Formel berechnet:

$$V = \frac{MTBF}{(MTBF + MTTR)}$$

V: Verfügbarkeit

MTBF: Mean Time Between Failurse

MTTR: Mean Time To Repair

2.7.4. Benutzbarkeit

Die Benutzbarkeit ist eine nichtfunktionale Anforderung und kann schwer in Zahlen gefasst werden. Durch Usability Tests kann die Benutzbarkeit einer Software getestet werden. Natürlich kann das Ergebnis stark von den Testpersonen und deren Hintergrundwissen abhängen.

Es ist wichtig so viele Benutzbarkeits-Anforderungen wie möglich schriftlich zu definieren. Die Schwierigkeit liegt dabei darin die Anforderungen messbar zu machen.

2.7.5. Sicherheit

Definiert die Anforderungen and die Sicherheit gegenüber Manipulationen und Fehlbedienungen.

2.7.6. Erweiterbarkeit

Definiert die Möglichkeit weitere Funktionalität einzufügen ohne wesentliche Eingriffe an der bestehenden Code-Basis vornehmen zu müssen.

2.7.7. Wartbarkeit

Fehlerursachen sollten mit möglichst geringem Aufwand erkannt und behoben werden können.

2.7.8. Übertragbarkeit / Portierbarkeit

Software sollte mit möglichst geringem Aufwand auf andere technische Umgebungen (Betriebssysteme, Hardware, Middleware usw.) übertragbar sein.

2.7.9. Wiederverwendbarkeit

Die Software oder Teile davon können als Funktionsbausteine oder Komponenten in verschiedenen weiteren Problemlösungen zum Einsatz kommen.

2.8. Qualitätssicherung

Es gibt viele Methoden wie die Qualität gesichert werden kann:

- Inspektion: Reviews des aktuellen Standes in allen Projektphasen
- Formale Methoden: Selektive Anwendung
- Test: Test der entwickelten Software auf Unit- und Anwendungs-Ebene
- Projektsteuerung Termine, Kosten und Dokumente überwachen

3. Prozess

Dieses Kapitel beschäftigt sich mit verschiedenen Prozessen. Ein Prozess definiert den Ablauf der Software Entwicklung von der Analyse bis zum Unterhalt beim Kunden. Er beschreibt wer was wie wann macht. Der Unified Software Development Process (USDP) stützt sich auf vier Modeling Elements:

1. Workers/Arbeiter: Definiert wer etwas macht.
2. Activities/Aktivitäten: Definiert wie etwas gemacht wird.
3. Workflows/Arbeitsprozesse: Definiert wann etwas gemacht wird.
4. Artifacts/Artefakte: Definiert was gemacht wird.

3.1. Software-Kategorien

Software kann grob in Kategorien eingeteilt werden. Jede dieser Anwendungen hat andere Anforderungen. Im speziellen an die Qualität und die damit verbundenen Faktoren (siehe Qualität).

- Stand-alone: Anwendungen, die auf einem einzigen Rechner laufen und nicht mit Umsystemen (Hardware/Software) verbunden sind. Ein typisches Beispiel ist eine Textverarbeitungssoftware
- Embedded: Speziell auf eine Hardware zugeschnitten. Ein typisches Beispiel wäre ein programmierbarer Videorekorder
- Realtime: Sehr zeitkritische Anwendungen. Beispielsweise Radar Software
- Netzwerk: Bestehend aus mehreren Teilen, die im Netzwerk miteinander verbunden sind. Beispielsweise Online-Spiele

3.2. Aktivitäten, Artefakte und Arbeiter

Die Verantwortlichkeiten sind Personen/Arbeitern zugeordnet:

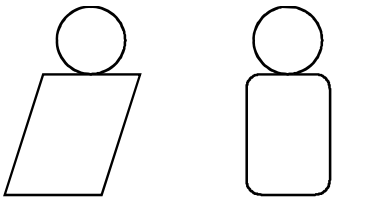


Abbildung 4 Arbeiter

Aktivitäten Sind die Aufgaben, welche von den Arbeiten übernommen werden:

Abbildung 5 Aktivität

Artefakte sind die bei den Arbeiten entstehenden Erzeugnisse:

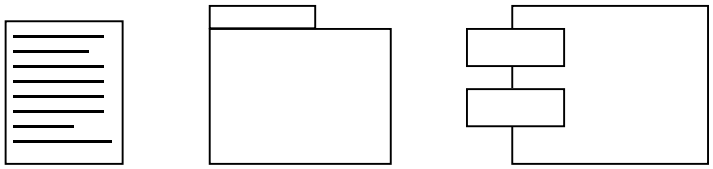


Abbildung 6 Artefakte

3.3. Workflow / Arbeitsprozesse

Es gibt unterschiedlichste Phasenmodelle. Alle sind unterschiedlich und nicht kompatibel zum Rest der Welt. Im wesentlichen lassen sich die Modelle aber in sequenzielle, iterative oder V-Modelle einteilen.

3.4. Sequenzielle-Modelle

Das wichtigste sequenzielle Modell ist das Wasserfall-Modell. Charakterisierend dafür ist ein linearer/sequenzieller Ablauf. Das Projekt durchläuft nacheinander die Phasen Konzept, Requirements, Design, Implementation, Test, Installation und Unterhalt. Es gibt keine Möglichkeit zurück und eine Phase wird immer nur genau einmal durchlaufen. Der Vorteil dieser Modelle ist eine einfache Strukturierung. Insbesondere für statische Projekte ist ein Wasserfall-Modell gut geeignet. Da es keine Iterationen gibt können Änderungen während des Projektes schlecht einfliessen. Ein weiterer Nachteil ist, dass immer von einer idealen Welt ausgegangen werden muss, da es kaum möglich ist auf unerwartete Änderungen einzugehen.

3.5. V-Modelle

Auch die V-Modelle werden als ganzes durchlaufen und es gibt keine Iterationen. In jeder Phase kann aber eine Verifizierung und Validierung erfolgen:

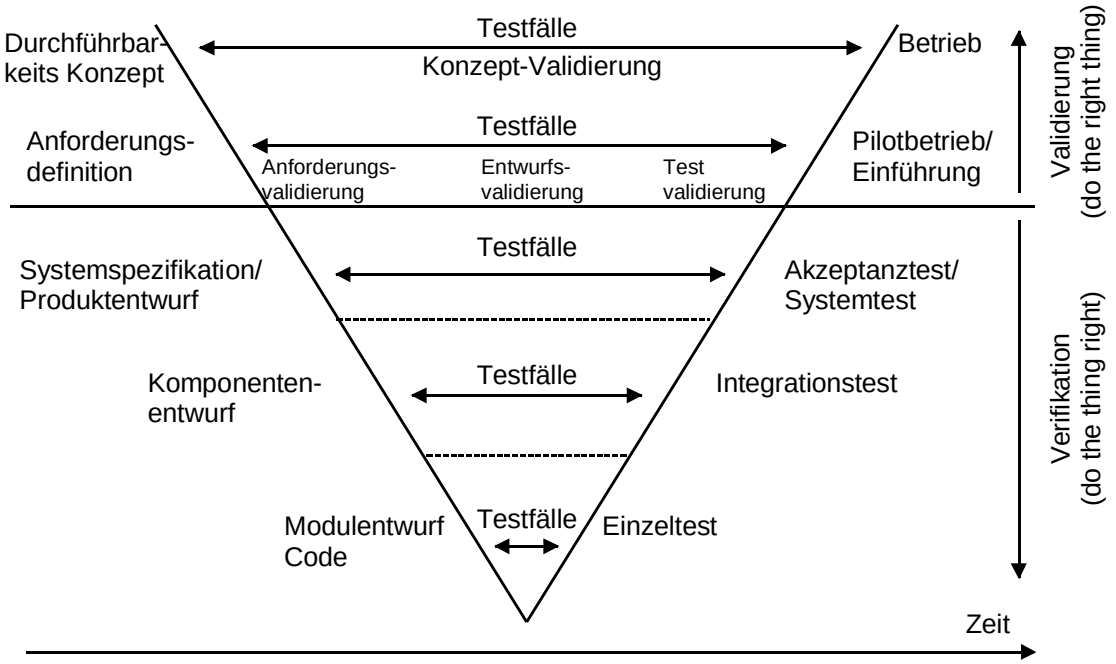


Abbildung 7 V-Modell

3.6. Iterative Modelle

Das wohl bedeutendste iterative Modell ist der Rational Unified Process (RUP) auch als Unified Software Development Process (USDP) bezeichnet. Der Prozess sieht Iterationen innerhalb einer Phase vor.

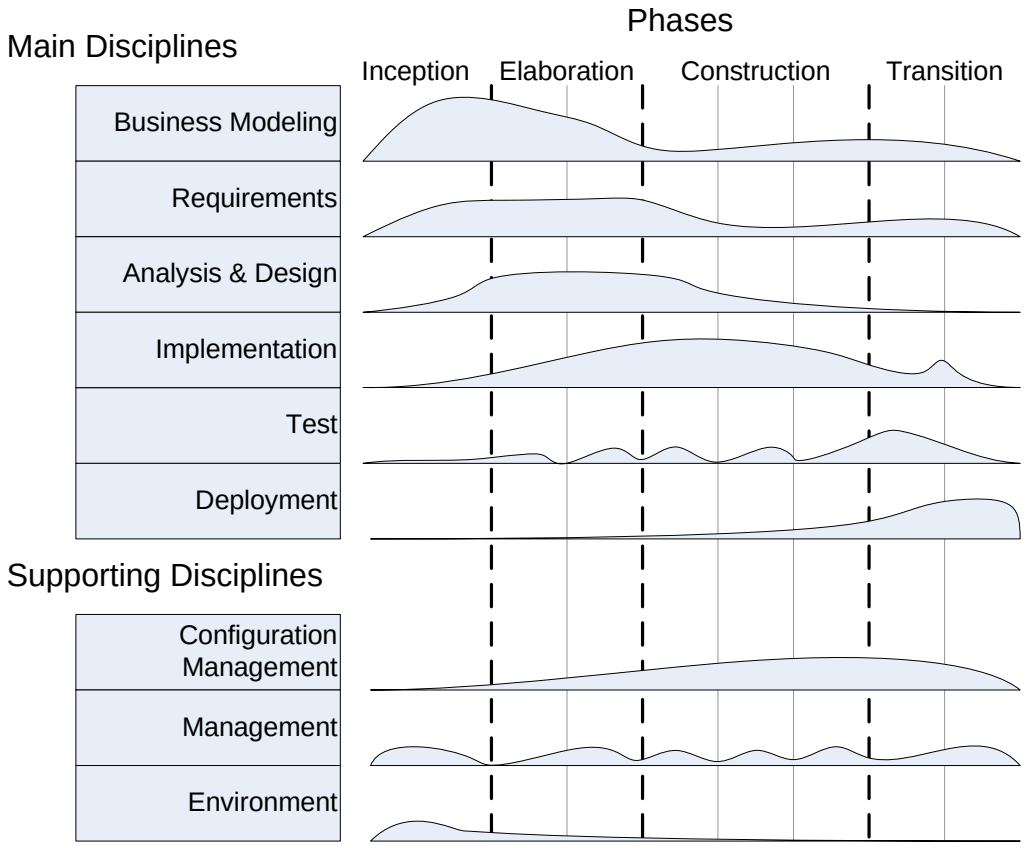


Abbildung 8 Unified Software Development Process (USDP)/RUP

Zu den Iterativen Modellen gehören auch die risikogetriebenen Spiralmodelle und Modelle mit einem Makro- und einem Mikroprozess. Bei letzteren werden die Phasen in einem Makroprozess durchlaufen. Bei jeder Phase wird ein Mikroprozess gestartet, der iterativ sein kann.

Bei Spiralmodellen wird bei jeder Iteration eine Risikoanalyse durchgeführt.

3.7. Weitere Modelle

Extreme Programming (XP) geht einen etwas anderen Weg. Trotzdem handelt es sich im Grund um ein iteratives Modell nur dass die Iterationen nicht nur innerhalb der Phasen stattfinden sondern der ganze Prozess mehrmals durchlaufen wird.

Dazu wird bei XP die Aufgabe in kleine Teilaufgaben aufgeteilt und diese nach dem XP-Prozess abgearbeitet. Dies erlaubt es möglichst schnell ein kleines Release und dann in kurzen Abständen neue Versionen herauszugeben. Die einzelnen Schritte sind dabei überblickbar klein gehalten.

Dieses Vorgehen erlaubt ein sehr schnelles Feedback in den einzelnen Projektphasen. Das Projekt wird immer in sehr kleinen Schritten vorangetrieben. Bei XP gibt es keine Releases im eigentlichen Sinne sondern ein stetig überarbeitetes Produkt. Die Aufteilung in kleine Einheiten erleichtert auch den Test, da diese für sich gesehen besser getestet werden können.

Dabei ist es wichtig das System zu jedem Zeitpunkt so einfach wie möglich zu halten. Unnötige Komplexität wird beseitigt, sobald sie entdeckt wird.

Gemäss dem Prinzip des Pair-Programming wird der Code vollständig von zwei Entwicklern an einem Rechner geschrieben. Ausserdem darf jeder Entwickler zu jeder Zeit Änderungen an jedem beliebigen Code-Fragment vornehmen. Nach Abschluss einer Arbeit wird eine Systemintegration durchgeführt.

XP macht auch Angaben über die Arbeitszeit. Es soll nicht mehr als 40 Stunden pro Woche gearbeitet werden und auf keinen Fall sollen in zwei aufeinander folgenden Wochen Überstunden gemacht werden.

Der Anwender soll direkt ins Team aufgenommen werden.

Die Programmierer müssen sich an Codier-Richtlinien halten.

4. Qualitätssicherungsprozesse

Es gibt verschiedene welche die Qualität sicherstellen sollen. Ziel ist es die Qualität zu messen und eine fortlaufende Qualitätssteigerung sicherzustellen.

4.1. Personal Software Process (PSP)

Ein Prozess zur Bildung von individuellen Fähigkeiten und der Disziplin.

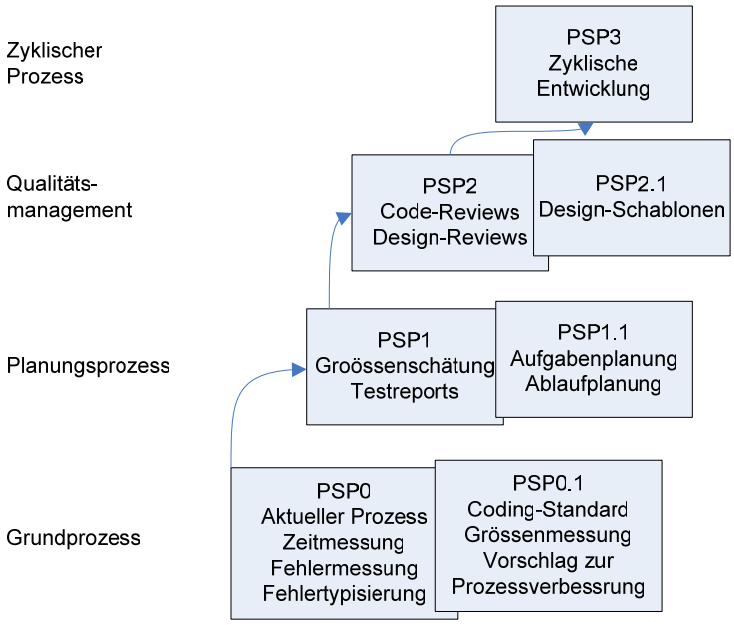


Abbildung 9 PSP-Evolution

4.2. Team Software Process (TSP)

Der Team Software Process hat zur Aufgabe selbstagierende Teams zu bilden. Die Teams bestehen aus 3-20 Entwicklern, die alle an ihren eigenen Zielen arbeiten und ihre eigenen Prozesse und Pläne haben. Ausserdem soll der Prozess die Arbeit besser überwachen lassen.

Das Ziel ist es möglichst schnell CMM Level 5 zu erreichen..

4.3. Capability Maturity Model (CMM)

Das CMM ist ein Modell um die Qualität der Softwareentwicklung zu beurteilen. Das Ziel ist es einen Definierten Prozess zu folgen, überwachbare Vorgänge zu haben und diese bei neuen Projekten wiederverwenden zu können. Dazu werden mehrere Stufen definiert. Ziel ist es natürlich Stufe 5 zu erreichen:

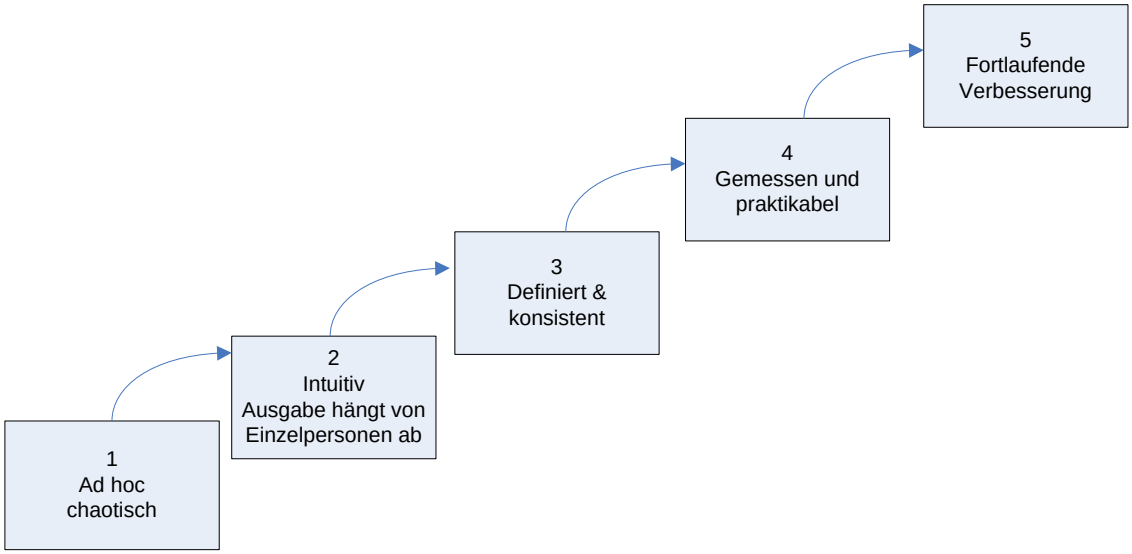


Abbildung 10 Capability Maturity Model

5. Team Management

Ein Team zusammenzustellen ist eine schwierige Aufgabe. Dabei müssen einige wichtige Faktoren berücksichtigt werden.

5.1. Teamgrösse und Interaktion

Die Teamgrösse ist entscheidend für die Effizienz eines Teams. Ist ein Entwickler total isoliert und arbeitet für sich alleine so ist er in der Regel nicht sehr Effizient und produziert womöglich Module, die vor ihrem Einsatz überarbeitet werden müssen um überhaupt ins Gesamtsystem zu passen. Umgekehrt ist ein Entwickler, der mit 11 Leuten in häufigem Kontakt seht ineffizient, da er nicht mehr dazu kommt seine Arbeit zu erledigen sondern immer damit beschäftigt ist diese mit den anderen Teammitgliedern zu synchronisieren und zu diskutieren.

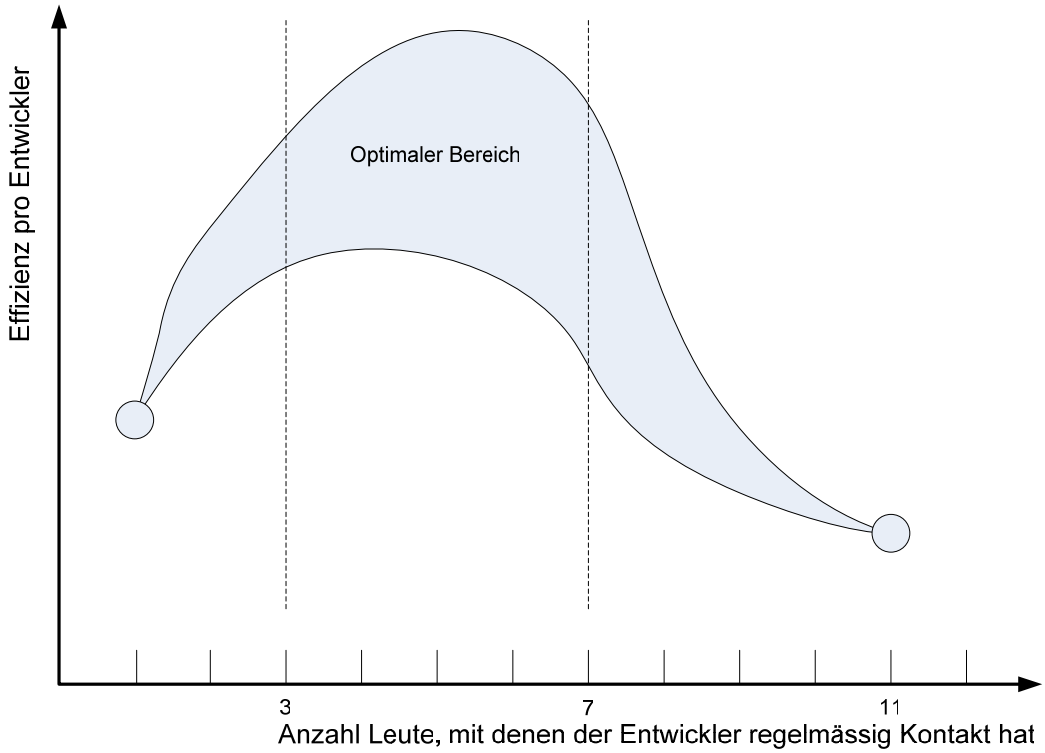


Abbildung 11 Optimale Teamgrösse

Es zeigt sich auch hier, dass die optimale Grösse bei der magischen Zahl 7 liegt. Dies liegt auch daran, dass der Mensch fähig ist gleichzeitig mit 7 plus/minus 2 Werten umzugehen (in diesem Falle Teammitglieder).

6. Risiko Management

Das Risiko-Management teilt sich im wesentlichen in die folgenden Unterkategorien:

6.1. Risiken identifizieren

Risiken müssen systematisch identifiziert werden. Dies beinhaltet strategische, Markt- finazielle, rechtliche und technische Risiken.

- Systematik zur Risikoidentifikation: Alle Mitarbeiter werden befragt. Dies ist recht zeitaufwändig und man bekommt nicht notwendigerweise die Projektrelevanten Risiken sondern eher die gerade aktuell vorhandenen (die eher für laufende Projekte zutreffen).
- Bezug zu den Erfolgsfaktoren: Es sollen die Risiken identifiziert werden, welche einen direkten Einfluss auf die Erfolgsfaktoren (wie beispielsweise die Kernkompetenzen) haben.
- Einsatz von Fachexperten: Häufig werden externe Berater beigezogen. Diese können die Risiken unabhängig analysieren und haben das entsprechende Know-How.

Sind die Risiken erst mal identifiziert müssen sie nach Dringlichkeit geordnet werden.

6.2. Risiken analysieren

Risiken können nur verglichen werden, wenn sie einheitlich bewertet werden. Dazu ist es unerlässlich objektive Beurteilungen und Begründungen aufzuführen. Risiken mit einmaligem Schaden und solche mit langfristiger Wirkung müssen unterschieden werden. Um eine Übehrbewertung zu verhindern dürfen keine Überlappungen bei den Risiken existieren. Auch Risiken mit kleiner Schadenshöhe sind zu betrachten. Insbesondere dann, wenn es wahrscheinlich ist, dass es häufiger eintritt.

Tabelle 1 Risikoberechnung

Wertebereich: [0..10]	Eintretens-wahrscheinlichkeit 1=klein	Schadenshöhe 1=klein	Kosten zur Vermeidung 1=klein	Priorität (Berechnung)	Priorität Niedrig bedeutet höchste Priorität
„Hohes Risiko“	10	10	1	(11-10)*(11-10)*1	1
„Niedriges Risiko“	1	1	10	(11-1)*(11-1)*10	1000

7. Dokumentation

Dokumentation wird häufig unterschätzt und nicht sauber gehandhabt. Programmierer haben oft kein grosses Interesse daran ihre Arbeit in separaten Dokumenten zu beschreiben. Der Code reicht ihnen selbst dazu völlig aus. Ein weiteres Problem ist oft die Inkonsistenz zwischen der Software und der Dokumentation. Falsche oder unvollständige Dokumentation ist oft schlimmer als überhaupt keine Dokumentation. Trotzdem gehört die Dokumentation in der Regel zum Lieferumfang.

Je nach gewähltem Entwicklungsprozess, Komplexität und Umfang der Software kann auf gewisse Dokumente verzichtet werden.

Nachfolgend eine Beschreibung der wichtigsten Dokumente:

7.1. Software Project Management Plan (SPMP)

Der SPMP ist das zentrale Planungsdokument für ein Software-Entwicklungs-Projekt. Es enthält Die Projektorganisation ebenso wie eine detaillierte Zeit- und Ressourcenplanung.

Ausserdem beinhaltet es essenzielle Informationen über das Projekt selbst wie die Verantwortlichkeiten, Zielsetzungen, Prioritäten, Rahmenbedingungen, Risikomanagement, Überwachung und die verwendeten Technischen Hilfsmittel.

1. Introduction	3.3. Risk management
1.1. Project overview	3.4. Monitoring & controlling mechanisms
1.2. Project deliverables	3.5. Staffing plan
1.3. Evolution of the SPMP	**4. Technical process**
1.4. Referenced materials	4.1. Methods, tools & techniques
1.5. Definitions and acronyms	4.2. Software documentation
2. Project organization	4.3. Project support functions
2.1. Process model	**5. Work packages, schedule & budget**
2.2. Organizational structure	5.1. Work packages
2.3. Organizational boundaries and interfaces	5.2. Dependencies
2.4. Project responsibilities	5.3. Resource requirements
3. Managerial process	5.4. Budget & resource allocation
3.1. Managerial objectives & priorities	5.5. Schedule
3.2. Assumptions, dependencies & constraints	

Abbildung 12 IEEE 1058.11987: SPMP Inhalt

7.2. Software Configuration Management Plan (SCMP)

Der SCMP ist meist erst bei mittleren und grösseren Projekten von grösserer Bedeutung. Darin wird festgelegt wie die einzelnen Artefakte versioniert, verifiziert und freigegeben werden. Der SCMP kann Namenskonventionen und weitere Richtlinien im Zusammenhang mit dem Konfigurations-Management enthalten.

Das Konfigurations-Management befasst sich dabei mit der Behandlung von sogenannten Configuration Items (CI's). Dies Schliesst Dokumente und den Code mit ein.

1. Introduction
2. SCM management
2.1. Organization
2.2. SCM responsibilities
2.3. Applicable policies, directives & procedures
3. SCM activities
3.1. Configuration identification
3.1.1. Identifying configuration items
3.1.2. Naming configuration items
3.1.3. Acquiring configuration items
3.2. Configuration control
3.2.1. Requesting changes
3.2.2. Evaluating changes
3.2.3. Approving or disapproving changes
3.2.4. Implementing changes
3.3. Configuration status accounting
3.4. Configuration audits & reviews
3.5. Interface control
3.6. Subcontactor / vendor control
4. SCM schedules
5. SCM resources
6. SCM plan maintenance

Abbildung 13 IEEE 828-1990: SCMP Inhalt

7.3. Software Quality Assurance Plan (SQAP)

Dieses Dokument soll die Qualität der Software sicherstellen. Aus diesem Grunde definiert es hauptsächlich Reviews, die durchgeführt werden müssen.

1. Purpose
2. Reference documents
3. Management
3.1. Organization
3.2. Tasks
3.3. Responsibilities
4. Documentation
4.1. Purpose
4.2. Minimum documentation requirements
4.3. Other
5. Standards, practices, conventions and metrics
5.1. Purpose
5.2. Content
6. Reviews and audits
6.1. Purpose
6.2. Minimum requirements
6.2.1. Software requirements review
6.2.2. Preliminary design review
6.2.3. Critical design review
6.2.4. SVVP review
6.2.5. Functional audit
6.2.6. Physical audit
6.2.7. In-process audits
6.2.8. Managerial review
6.2.9. SCMP review
6.2.10. Post mortem review
6.3. Other

Abbildung 14 IEEE 730-1989: SQAP Inhalt

7.4. Software Requirements Specifications (SRS)

Das SRS-Dokument ist das Pflichtenheft und definiert, was genau geliefert werden muss. Es enthält die D-Requirements und spezifiziert somit detailliert was genau geliefert werden muss (siehe Anforderungsanalyse).

1. Introduction
1.1. Purpose
1.2. Scope
1.3. Definitions, acronyms & abbreviations
1.4. References
1.5. Overview
2. Overall description
2.1. Product perspective
2.1.1. System interfaces
2.1.2. Hardware interfaces
2.1.3. Software interfaces
2.1.4. Communications interfaces
2.1.5. Memory constraints
2.1.6. Operations
2.1.7. Site adaptation requirements
2.2. Product functions
2.3. User characteristics
2.4. Constraints
2.5. Assumptions and dependencies
2.6. Apportioning of requirements
3. Specific requirements
[see chapter 4]
4. Supporting information
[see chapter 4]

Abbildung 15 IEEE 830-1993: SRS Inhalt

Der Inhalt des Kapitels 3 hängt vom Typ der Requirements ab und für in OO bzw. non-OO Requirements unterschiedlich:

Specific requirements (non-OO)	**Specific requirements (OO)**
3.1. External Interfaces 3.2. Functions 3.3. Performance requirements 3.4. Logical database requirements 3.5. Design constraints 3.5.1. Standards compliance 3.6. Software system Attributes 3.6.1. Reliability 3.6.2. Availability 3.6.3. Security 3.6.4. Maintainability 3.6.5. Portability 3.7. Organizing the specific req. 3.7.1. System mode -- or 3.7.2. User class -- or 3.7.3. Objects (see right) -- or 3.7.4. Feature -- or 3.7.5. Stimulus -- or 3.7.6. Response -- or 3.7.7. Functional hierarchy -- or 3.7.8. Additional comments -- or	3.1. External interface requirements 3.1.1. User interfaces 3.1.2. Hardware interfaces 3.1.3. Software interfaces 3.1.4. Communication interfaces 3.2. Classes/Objects 3.2.1. Class/Object1 3.2.1.1. Attributes (direct or inherited) 3.2.1.1.1 Attribute 1 ... 3.2.1.2. Functions (services, methods, direct or inherited) 3.2.1.2.1 Functional requirement ... ... 3.3. Performance requirements 3.4. Design constraints 3.5. Software system attributes 3.6. Other requirements

Abbildung 16 IEEE 830-1994: Specific (D-)Requirements

7.5. Software Test Documentation (STD)

Dieses Dokument beinhaltet die Test-Cases anhand derer die Tests durchgeführt werden. Die Testergebnisse werden ebenfalls in diesem Dokument festgehalten.

1. Introduction

2. Test plan
Items under test, scope, approach, resources, schedule, personnel

3. Test design
Items to be tested, the approach, the plan in detail

4. Test cases
Sets of inputs and events

5. Test procedures
Steps for setting up and executing the test cases

6. Test item transmittal report
Item under test, physical location of results, person responsible for transmitting

7. Test log
Chronological record, physical location of test, tester name

8. Test incident report
Documentation of any event occurring during testing which requires further investigations

9. Test summary report
Summarizes the above

Abbildung 17 IEEE 829-1983: STD Inhalt

7.6. Software Validation & Verification Plan (SVVP)

Dieses Dokument spezifiziert die Software Verifikation und Validierung.

1. Purpose
2. Referenced Documents
3. Definitions
4. V&V overview
4.1. Organization
4.2. Master schedule
4.3. Resource summary
4.4. Responsibilities
4.5. Tools, techniques & methodologies
5. Lifecycle V&V
5.1. Management of V&V
5.2. Concept phase V&V
5.3. Requirements phase V&V
5.4. Design phase V&V
5.5. Implementation phase V&V
Test phase V&V
5.6. Installation & checkout phase V&V
5.7. Operation & maintenance phase V&V
6. Software V&V reporting
6.1. Required reports
6.2. Optional reports
7. V&V administrative procedures
7.1. Anomaly reporting & resolution
7.2. Task iteration policy
7.3. Deviation policy
7.4. Standards, practices & convention

Abbildung 18 IEEE 1012-1986: SVVP Inhalt

7.7. Software Design Document (SDD)

Das Software Design Dokument enthält Informationen über die Architektur der Software:

1. Introduction	**4. Dependency description**
1.1. Purpose	4.1. Intermodule dependencies
1.2. Scope	4.2. Interprocess dependencies
1.3. Definitions & abbreviations	4.3. Data dependencies
2. References	**5. Interface description**
3. Decomposition description	5.1. Module interface
3.1. Module decomposition	5.1.1. Module 1 description
3.1.1. Module 1 description	5.1.2. Module 2 description
3.1.2. Module 2 description	5.2. Process interface
3.2. Concurrent process decomposition	5.2.1. Process 1 description
	5.2.2. Process 2 description
3.2.1. Process 1 description	**6. Detailed design**
3.2.2. Process 2 description	6.1. Module detailed design
3.3. Data decomposition	6.1.1. Module 1 detail
3.3.1. Data entry 1 description	6.1.2. Module 2 detail
3.3.2. Data entry 2 description	6.2. Data detailed design
	6.2.1. Data entity 1 detail
	6.2.2. Data entity 2 detail

Abbildung 19 IEEE 1016: SDD Inhalt

Die Kapitel 3 bis 5 beschäftigen sich dabei intensiv mit der Architektur.

7.8. Benutzerhandbuch

Das Benutzerhandbuch ist eine Kundendokumentation. Im speziellen für die Personen, welche die Software benutzen werden. Es ist wichtig dieses Dokument verständlich für einen Benutzer zu schreiben damit es nicht nur von Entwicklern verstanden werden kann.

7.9. Source Code

Auch der Source-Code kann zur Dokumentation gehören. Insbesondere wenn Code-Dokumentationen wie JavaDoc zur Anwendung kommen spielt der Source-Code eine Wichtige Rolle als Dokumentation. Schnittstellen und die Zusammenhänge der einzelnen Komponenten können damit am besten dokumentiert werden. Ausserdem ist Dokumentation direkt aus dem Code natürlich immer aktuell und stimmt mit der Software überein.

8. Reviews und Audits

Audits und Revies dienen dazu die bereits erledigte Arbeit zu überprüfen und gegenüber den Spezifikationen zu verifizieren. Dabei wird sowohl der Code als auch die Dokumentation geprüft. Das Ziel eines Reviews ist Fehler und Differenzen frühzeitig zu erkennen und entsprechende Gegenmassnahmen einleiten zu können.

Es gibt verschiedene Stufen von Reviews:

- Inspektion: Rigorose Komplettprüfung (klar definierter Prozess)
- Team Review: Ein wenig weniger formell; innerhalb des Teams
- Walkthrough: Der Autor Präsentiert seine Arbeit einer Gruppe
- Pair Programming: Zwei Entwickler arbeiten gemeinsam an der selben Aufgabe und kontrollieren sich gegenseitig
- Passaround: Die Arbeit wird an einige Leute weitergegeben und deren Rückmeldung analysiert

- Ad Hoc Review: Meist im Rahmen eines kleinen Meetings wird über die Arbeitsergebnisse diskutiert

Im Quality Assurance Plan (siehe Software Quality Assurance Plan (SQAP)) sollen die Reviews aufgelistet werden.

Folgende Reviews sind empfohlen:

- Feasibility Review
- Requirements Review
- Preliminary Design Review
- Critical Design Review
- Source Code Review
- Productive Release Review

9. Kosten

Kostenberechnungen gibt es in verschiedenen Ausprägungen. Beispielsweise gibt es Faustformeln für die Berechnung der Kosten anhand der Codezeilen oder der Kosten pro Fehler.

9.1. Kostenschätzung

Natürlich ist die Kostenschätzung umso genauer je mehr sich das Projekt dem Ende nähert. In der konzeptuellen Phase können die Kosten nur recht ungenau geschätzt werden. Dieser Schätzwert verbessert sich in der Analyse der Anfordrungen (Requirement analysis), dem Design, bei der Implementation und schlussendlich bei der Implementation kann man relativ genau sagen welche Kosten entstehen werden.

Typischerweise geht man zur Schätzung der Kosten folgendermassen vor:

1. Vergleich mit früheren Projekten um die Kosten und die Dauer direkt abzuschätzen oder einen Anhaltspunkt für die Anzahl Codezeilen (Lines of Code: LOC) zu bekommen.
2. Einsatz der Function-point Methode um die Anzahl Codezeilen zu schätzen.
3. Benutzen der erhaltenen geschätzten Anzahl Codezeilen um mit COCOMO die Dauer und Koten abzuschätzen.

9.1.1. Mittelwert-Methode

Die einfachste Schätztechnik ist mehrere Leute zu befragen und den Mittelwert daraus zu bestimme. Die Genauigkeit dieses Verfahrens kann verbessert werden, wenn die Gruppe möglichst heterogen gemischt ist und die Aufgabe in kleinere Teilblöcke aufgeteilt werden kann.

9.1.2. Schätzpunkt-Methode

Beim Schätzpunkt-Verfahren wird die Wahrscheinlichkeit für jede mögliche Projektdauer geschätzt. Der Mittelwert bildet eine sogenannte Schätzkurve. Teilt man die Fläche in zwei optisch gleich grosse Flächen so erhält man einen mittelwert (Schätzwert der Projektdauer).

9.1.3. 3-Punkte Methode

Die 3-Punkte Methode verwendet drei Werte:

- Optimistische Schätzung (minimale Dauer)
- Der am häufigsten eintretende Wert (modale)
- Pessimistische Schätzung (maximale Dauer)

Der mittlere Aufwand ergibt sich dann aus:

$$Aufw_{mittel} = \frac{\left(Aufw_{optimist} + 4 \cdot Aufw_{\mathrm{mod}\,al} + Aufw_{pessimist}\right)}{6}$$

9.1.4. Essenzschritt-Verfahren

Beim Essenzschritt-Verfahren werden zuerst die Anwendungsfälle (Use-Cases) beschrieben. Diese sollten etwa vergleichbar komplex sein. Danach wird jeder Use-Case im Aufwand geschätzt. Das Problem besteht insbesondere darin, dass die Use-Cases von sehr unterschiedlicher Komplexität sein können.

9.1.5. FunctionPoint-Methode

Bei diesem Verfahren werden Die Funktionen des Systems aus Anwendersicht definiert. Dann kann folgendermassen vorgegangen werden:

1. Funktionale Anforderungen in elementare Teilprozesse bzw. Datenbestände zerlegen.
2. Komplexität ermitteln (Anzahl Attribute und Beziehungen)

3. Diese unbewerteten FunctionPoints in das Berechnungsforumlar eintragen. Dies ergibt ein Mass für die Produktgrösse.
4. Bewerten der eingetragenen FunctionPoints entsprechend de geplanten Umsetzung und Qualitäts- und Leistungsanforderungen.
5. Ermittlung des Aufwandes.
6. Einflussfaktoren bestimmen. Mithilfe der Einflussfaktoren können FunctionPoints um bis zu 30 Prozent auf- oder abgewertet werden.
7. Auf der Basis früher abgeschlossener Projekte die FunctionPoints mit entsprechendem Aufwand versehen.

Tabelle 2 FunctionPoint Tabelle

	Simple		Medium		Complex		Sub-total	Total
	#	**Factor**	#	**Factor**	#	**Factor**		
Ext Inputs	1	**3**	1	**4**	1	**6**	13	
Comments	Name		Ready/move		Qualities			
Ext. Outputs	0	**4**	0	**5**	0	**7**	0	
Ext. Inquiries	0	**3**	0	**4**	0	**6**	0	25
Int. Logical files	1	**7**	0	**10**	0	**15**	7	
Comments	Data about the user's character							
Ext. interface files	1	**5**	0	**7**	0	**10**	5	
Comments	Data about the user's character							

9.1.6. Constructive Cost Model (COCOMO)

COCOMO besagt, dass alleine aus der Anzahl geschätzter Codezeilen der Aufwand und die Projektdauer geschätzt werden kann:

$$\text{Effort} = a \cdot KLOC^{b}$$

$$\text{Duration} = c \cdot Effort^{d}$$

Dabei wird der Aufwand (Effort) in Personen-Monaten angegeben. KLOC ist die Anzahl Codezeilen in Tausendereinheiten. Die Parameter a, b, c und d hängen vom Typ des Projektes ab:

Tabelle 3 COCOMO Variabeln

Software Project	a	b	c	d
Orcanic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.2	2.5	0.32

Was sagen die COCOMO-Formeln aus?

Die Formeln stellen die Abhängigkeit von Aufwand und Anzahl Codezeilen dar. Ebenso die Projektdauer in Abhängigkeit vom Aufwand nach folgender Kurve:

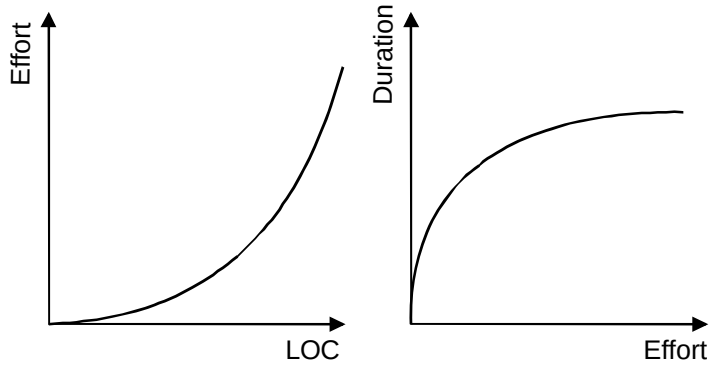


Abbildung 20 COCOMO Visualisiert

9.2. Kosten pro Fehler

Tabelle 4 Kosten Pro Fehler

	Phase inspection	Phase integration
Stunden um einen Fehler zu finden	0.7 bis 2	02. bis 10
Stunden um einen Fehler zu beheben	0.3 bis 1.2	9+
Total	1.0 bis 3.2	9.2 bis 19+

Je später ein Fehler erkannt wird umso teuerer wird dieser. Insbesondere steigt der Aufwand zur Beseitigung des Fehlers massiv an.

10. Anforderungsanalyse

Anforderungen können über Use-Case Diagramme, State-Diagramme, Datenflussdiagramme und Skizzen von Benutzermasken definiert werden.

Die Anforderungen (Requirements) müssen in C-(Customer) Requirements und D-(Detailed) Requirements unterschieden werden.

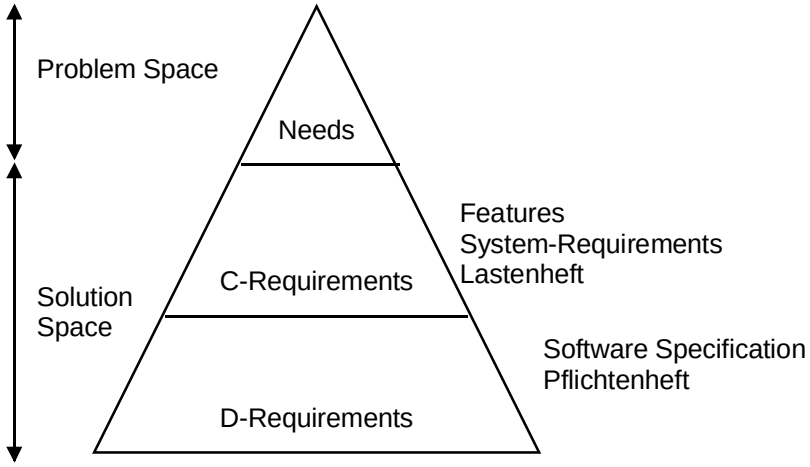


Abbildung 21 Anforderungsanalyse

Das Lastheft ist die Grundlage zum einholen von Angeboten und beinhaltet die vom Kunden vorgegebenen Anforderungen. „Gesamtheit der Forderungen des Auftraggebers an die Lieferung und Leistungen eines Auftragnehmers“ - DIN69905

Das Pflichtenheft beinhaltet die zu erbringenden Ergebnisse. „Vom Auftragnehmer erarbeitete Realisierungsvorhaben aufgrund der Umsetzung des Lastenheftes“ - DIN69905

Ein Beispiel:

C-Requirement (Feature):

1 Eine Lichtquelle kann zeitgesteuert ein- und ausgeschaltet werden.

D-Requirement (Software Specification):

Timer.

#T1: Ein Timer ist frei programmierbar. Für jedes Zeitereignis kann eine Aktion ausgeführt werden.

#T2: Einmalige Zeitereignisse: Datum/Uhrzeit - Auflösung Minuten.

#T3: Zyklische Ereignisse: minütlich, stündlich, täglich Wochentag.

#T4: Mögliche Aktionen: Lichtquelle ein/aus.

#T5: Protokoll, welcher Timer bei Ereignis-Eintritt aussendet: [Zeit/Datum][Art][Aktion]

Die D-Requirements spezifizieren immer das ‚was’ und nicht das ‚wie’ etwas gelöst wird. Also nicht die Technische Realisierung des Problems.

Anforderungen werden üblicherweise im SRS (siehe Software Requirements Specifications (SRS)) festgeschrieben.

10.1. C-Requirement Erfassung

Nachfolgend der typische Ablauf bei der Erfassung von Anforderungen.

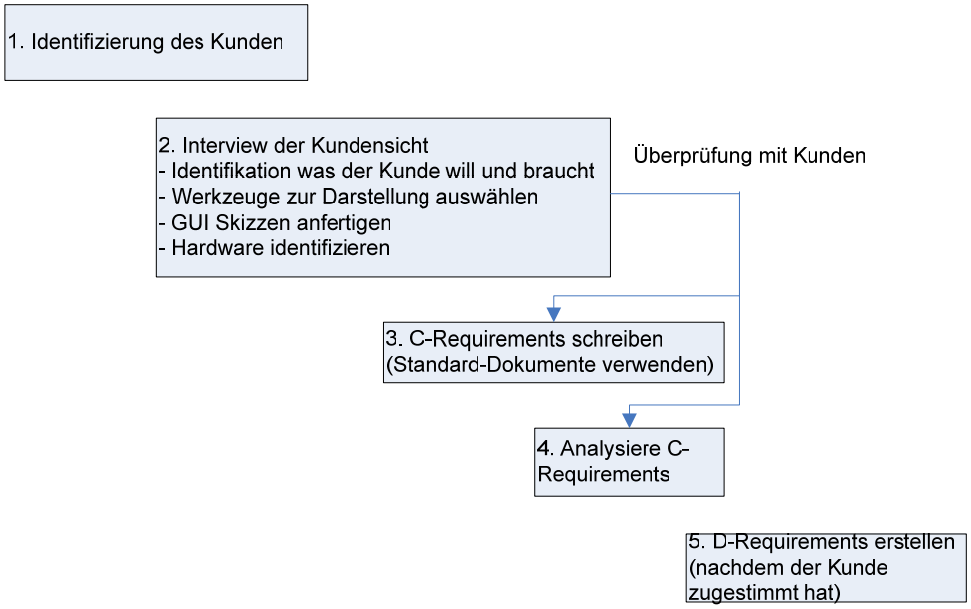


Abbildung 22 C-Requirements Erfassung

10.2. Requirements nach USDP

Im Unified Software Development Process (siehe Iterative Modelle) versteht man unter Requirements folgendes:

Tabelle 5 Requirements nach USDP

Tätigkeit	Resultat
Anforderungen aufzählen	Feature List
Systemkontext erfassen	Geschäftsmodell (Domain/Business Model)
Funktionale Anforderungen aufnehmen	UseCase-Modell
Nichtfunktionale Anforderungen aufnehmen	Zusätzlicher Input für SRS

Der Zweck des USDP Domain Modelles ist es die wichtigsten Business-Klassen im Kontext der Aufgabenstellung zu verstehen und zu dokumentieren. Business-Klassen sind abstrakte Objekte (z.B. Mensch, Steuerung, Schraube, Förderband usw.).

Die identifizierten Business-Klassen dienen zur Beschreibung der UseCases und für den Entwurf der Benutzerschnittstelle. Bei einfachen Systemen genügt auch ein Glossar.

10.3. D-Requirement Erfassung

Es gibt verschiedene Arten von Anforderungen:

- Funktionale Anforderungen: Anforderungen an die Direkte Funktion der Applikation.
- Nichtfunktionale Anforderungen: Performance, Geschwindigkeit, Kapazität, Ressourcen-Belegung (RAM, Festplatte).
- Zuverlässigkeit und Verfügbarkeit.
- Fehlerbehandlung.
- Benutzerinteraktion: Benutzermasken, Fehlermeldungen, Hilfe usw.
- Rahmenbedingungen: Genauigkeit, Entwicklungstools, Programmiersprache, Design-Anforderungen (3-Tier Architektur usw.), zu benutzende Standards, Hardware Plattformen.
- Invertierte Anforderungen: Was die Anwendung nicht macht.

Die Anforderungen sind detailliert im SRS zu beschreiben (siehe Software Requirements Specifications (SRS)).

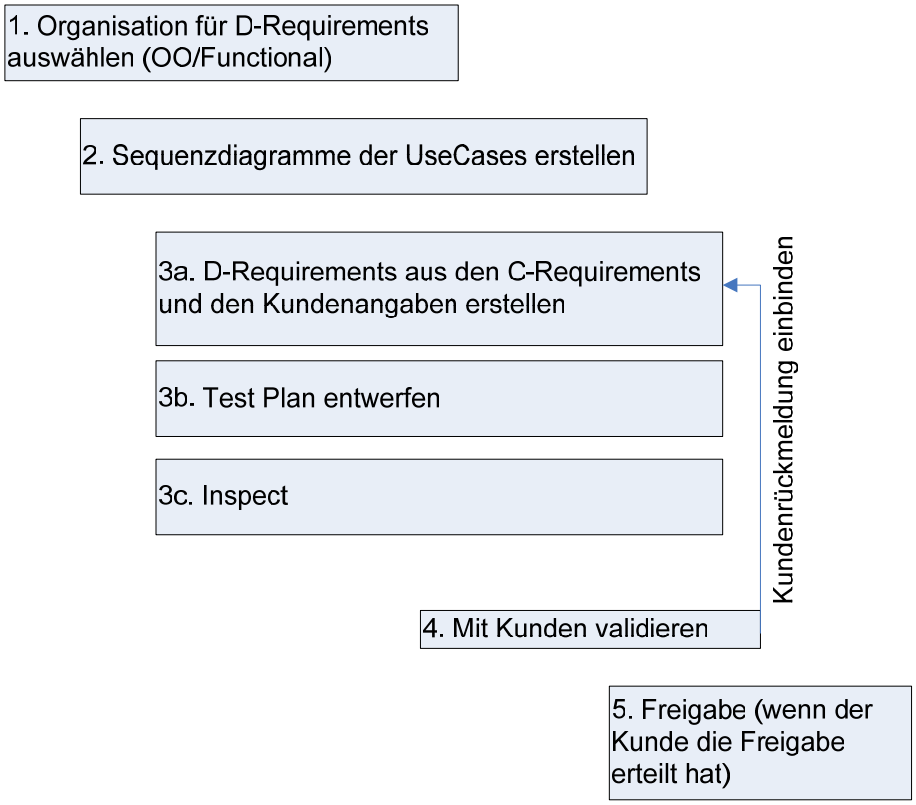


Abbildung 23 D-Requirements Erfassung

Bei der Erfassung ist immer darauf zu achten, dass die Anforderungen eindeutig, konsistent und testbar sind. Ausserdem sollen sie in Kategorien wie „essenziell“, „gewünscht“ und „optional“ eingeteilt werden.

10.4. Business Model: externe Sicht

Das Modell eines Geschäftssystem besteht aus:

- Externe Sicht (Umgebung des Systems): Beschreibt die Geschäftsprozesse, an denen nur Aussenstehende beteiligt sind.
- Interne Sicht (für aussenstehende nicht sichtbar): Beschreibt die Arbeitsabläufe um die Anforderungen der Externen Sicht zu erfüllen (Mitarbeiter, Hilfsmittel und Organisationsstruktur).

Die externe Sicht beschreibt das Geschäftssystem dabei als Blackbox. Es beschreibt welche Leistungen angeboten werden und wie diese in Anspruch genommen werden können (Kunden <-> Partner Interaktion).

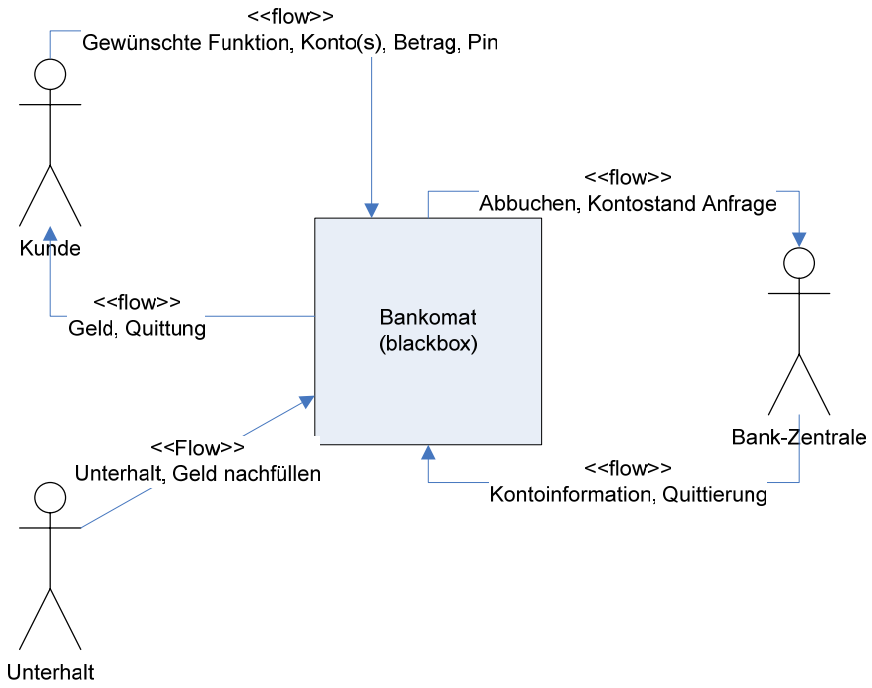


Abbildung 24 Kontext-Diagramm

Ein Anwendungsfall beschreibt eine Menge von Abläufen (inkl. Varianten) die ein System ausführen kann und die einen erkennbaren Nutzen für den jeweiligen Akteur bringen.

Die geschäfts-interne Funktionalität, welche für den Anwender nicht sichtbar ist gehört nicht dazu).

Elemente der externen sicht sind:

- UseCase Diagramme (Anwendungsfälle): Zeigen Akteure, Business UseCases und deren Beziehungen.
- Aktivitätsdiagramme: Beschreiben die Geschäftsprozesse als Sequenzen, Alternativen und parallelen Abläufen.
- Sequenzdiagramme: Zeigen den Nachrichtenaustausch mit Partnern und Kunden in zeitlicher Reihenfolge.

10.4.1. UseCase

UseCases werden in den verschiedenen Entwicklungsstufen ausgearbeitet:

- Inception: Die meisten UseCases werden in der Einstiegsphase erfasst. Die wichtigsten 10% werden auch schon ausgearbietet.
- Elaboration: Die restlichen UseCases werden in der Ausarbeitungsphase erfasst und bis zum Ende der Phase auch weitgehend ausgearbeitet).
- Construction: Die noch verbleibenden UseCases werden ausgearbeitet.

Die Modellierung geschieht meist grafisch. Da diese Darstellungsform sehr intuitiv zu lesen ist und wenige Missverständnisse entstehen.

Beispiel:

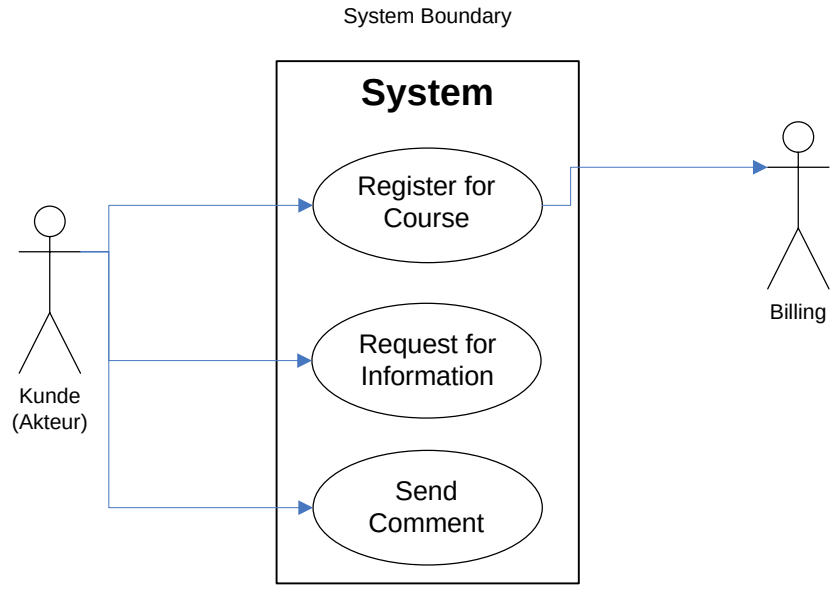


Abbildung 25 UseCase Beispiel

10.4.2. Aktivitätsdiagramm

Aktivitätsdiagramme visualisieren Arbeitsabläufe (also einzelne Sequenzen eines UseCases).

Jedes Aktivitätsdiagramm sollte einen Startpunkt haben. Normalerweise haben sie auch einen Endpunkt.

Beispiel:

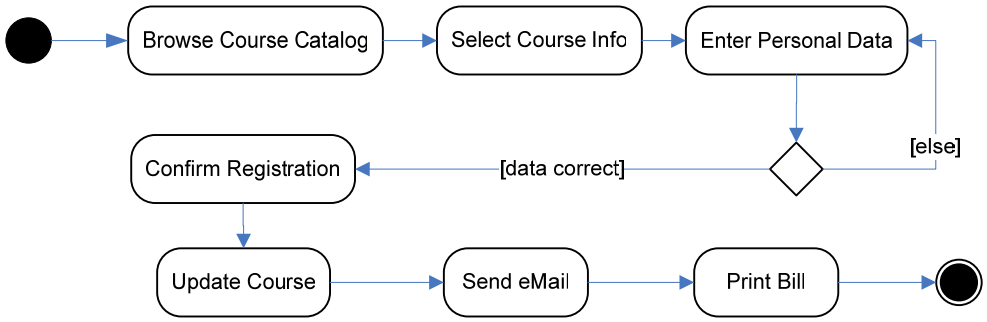


Abbildung 26 Aktivitätsdiagramm

10.4.3. Sequenzdiagramme

Das Ziel der Sequenzdiagramme ist es den Austausch von Nachrichten zwischen Objekten in einer limitierten Zeitspanne aufzuzeigen.

Beispiel:

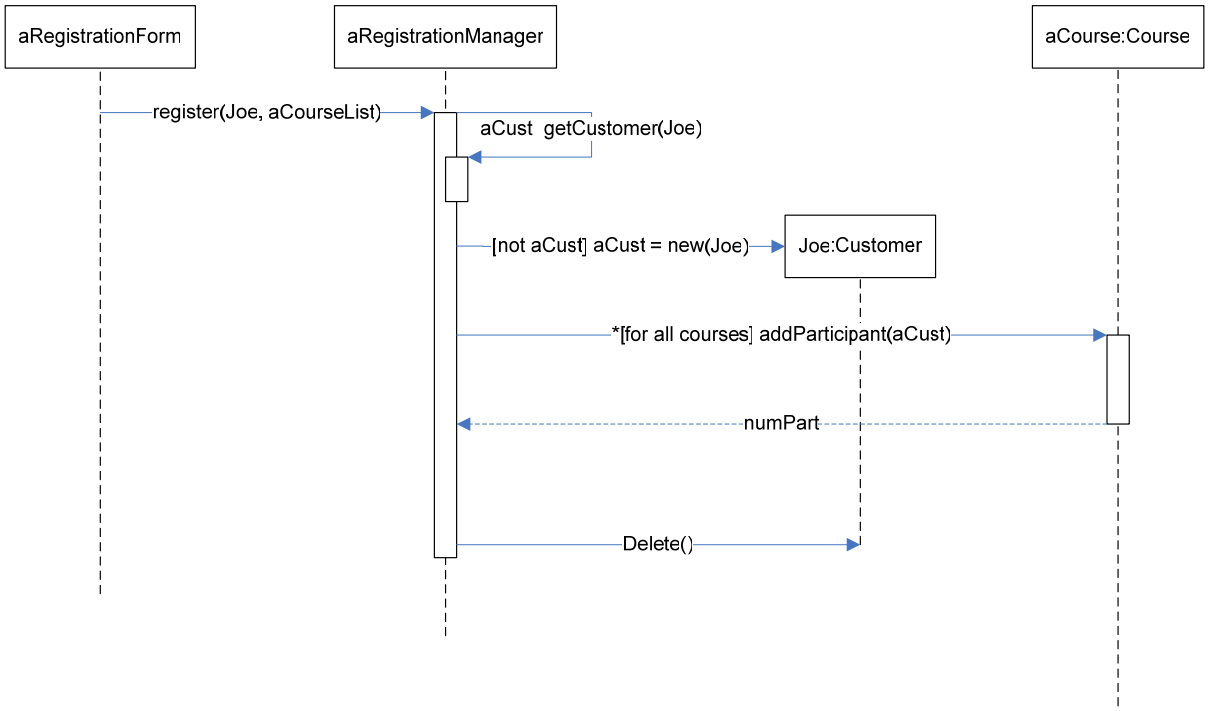


Abbildung 27 Sequenzdiagramm

10.5. Business Modell: Interne Sicht

Das Modell eines Geschäftssystem besteht aus:

- Externe Sicht (Umgebung des Systems): Beschreibt die Geschäftsprozesse, an denen nur Aussenstehende beteiligt sind.
- Interne Sicht (für aussenstehende nicht sichtbar): Beschreibt die Arbeitsabläufe um die Anforderungen der Externen Sicht zu erfüllen (Mitarbeiter, Hilfsmittel und Organisationsstruktur).

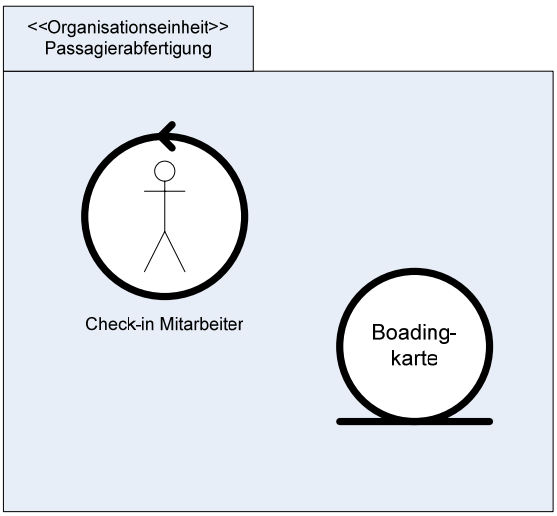
Die Interne Sicht kann dabei folgendermassen dargestellt werden:

- Paketdiagramme: Beschreibt die Organisationseinheiten.
- Klassendiagramme: Beschreibt die Zusammenhänge und Beziehungen der Mitarbeiter mit den Geschäftsobjekten.
- Aktivitätsdiagramme: Beschreibt die Geschäftsprozesse innerhalb des Geschäftssystems.

10.5.1. Packages

Ein Package gruppiert Modell-Elemente. Der Zweck besteht darin grosse Systeme aufzuteilen und Klassen in Einheiten auf höheren Ebene zu gruppieren.

UML 2 Stereotypen:



- <<Organisaitonseinheit>>: Umfasst weitere Organisationseinheigen, Workers, Business Objekte und deren Beziehungen.
- <<Worker>>: An der Abwicklung eines Geschäftsprozesses beteiligte Mitarbeiter innerhalb des Geschäftsprozesssystems.
- <<BusinessObject>>: Passive Objekte, die einzelne Iterationen überdauern können.

Abbildung 28 Package Stereotypen

<<Worker>> sind Rollen. Es braucht also nicht jede einzelne Person ein entsprechendes Symbol.

Paketdiagramme können hierarchisch geordnet sein. Eine Organisationseinheit kann mehrere Organisationseinheiten beinhalten.

UML 1.x Package Diagramm:

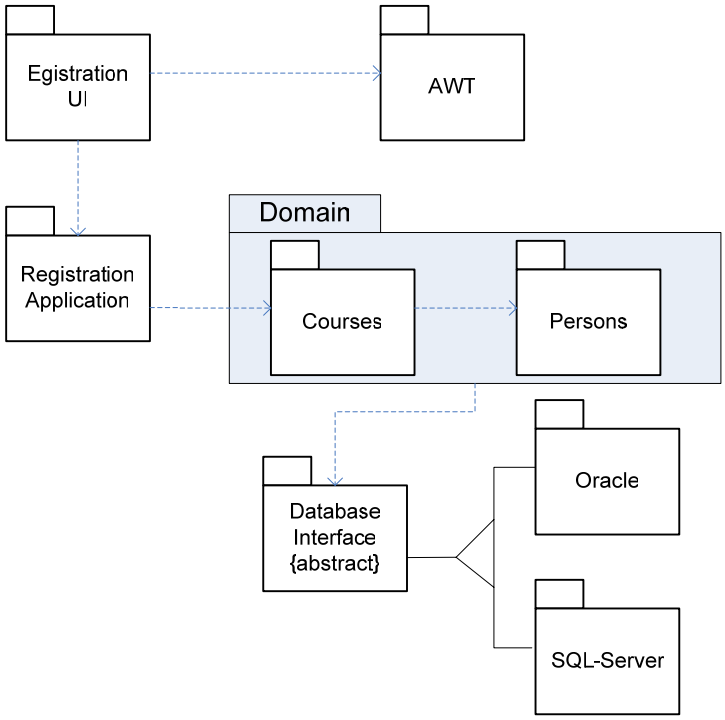
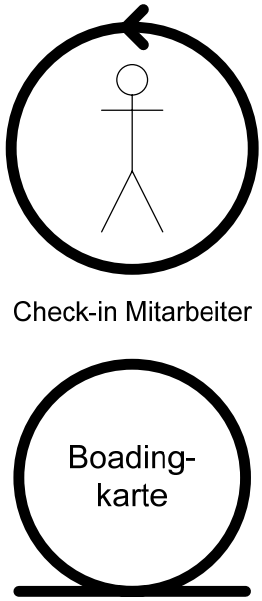


Abbildung 29 UML 1.x Package Diagram

10.5.2. Geschäftsklassen-Diagramme

Stark vereinfachte Klassendiagramme zeigen wichtige Beziehungen zwischen Mitarbeitern, Geschäftsobjekten und aussenstehenden.

Stereotypen für Geschäftsklassen:



- <<Worker>>: An der Abwicklung eines Geschäftsprozesses beteiligte Mitarbeiter innerhalb des Geschäftsprozess-Systems (siehe Paketdiagramm)
- <<BusinessObject>>: Passive Objekte, die einzelne Iterationen überdauern können. Modellieren wie im Paketdiagramm Information, die meist auch gespeichert werden muss.

Abbildung 30 Geschäftsklassen Stereotypen

Beispiel:

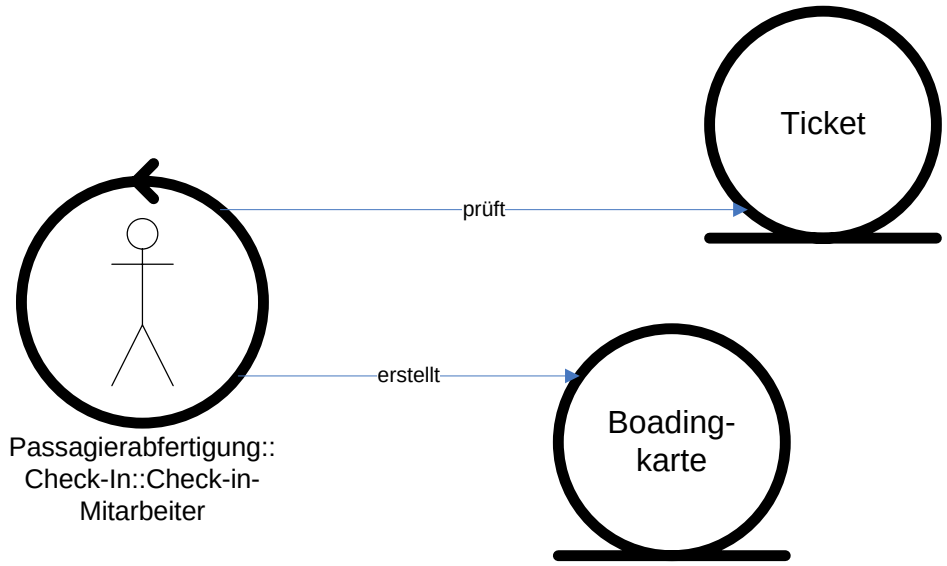


Abbildung 31 Geschäftsklassen-Diagramm

10.6. Analysemodell

Ein Analysemodell besteht aus:

- Dem Analyse-Klassenmodell mit den Stereotypen <<entity>>, <<boundary>> und <<control>>
- Sequenzdiagrammen die beschreiben, wie diese Klassen zusammenspielen.
- Dem konzeptionellen Datenmodell, ein redundanzfreies ER-Modell.

10.7. Das Analyse-Klassenmodell

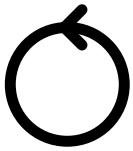
Das Analyse-Klassenmodell ist ein UML-Klassenmodell, welches das Problemverständnis repräsentiert. Es beschreibt die Anforderungen, keine Lösungen und enthält nur UML-Basiskonstrukte. Es sollte auch mit Anwendern und Fachsteilungsmitarbeitern diskutierbar sein. Ein Analyse-Klassenmodell muss nicht unbedingt auf ein Design-Klassenmodell abgebildet werden können.

Stereotypen:

Boundary-Klassen: Klassen für die Präsentation und Manipulation. Jeder Aktor eines UseCases benötigt ein eigenes Boundary-Objekt.

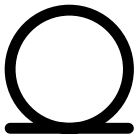
z.B.: Window, Dialogbox, Kommunikationskanal

Abbildung 32 Boundary Klasse



Control-Klassen: Verbinden Boundary-Klassen mit ihren Entity-Klassen. Behandeln das Verarbeiten der Eintity-Klassen. In einem ersten Schritt kann gesagt werden: Für jeden UseCase eine Control-Klasse.

Abbildung 33 Control-Klasse



Entity-Klassen: Modellieren von Informationen, die meist auch gespeichert werden müssen. Business-Objekte.

Abbildung 34 Entity-Klasse

Die Identifikation der Klassen ist nicht immer Trivial. Am besten lassen sich diese aus den UseCases ableiten:

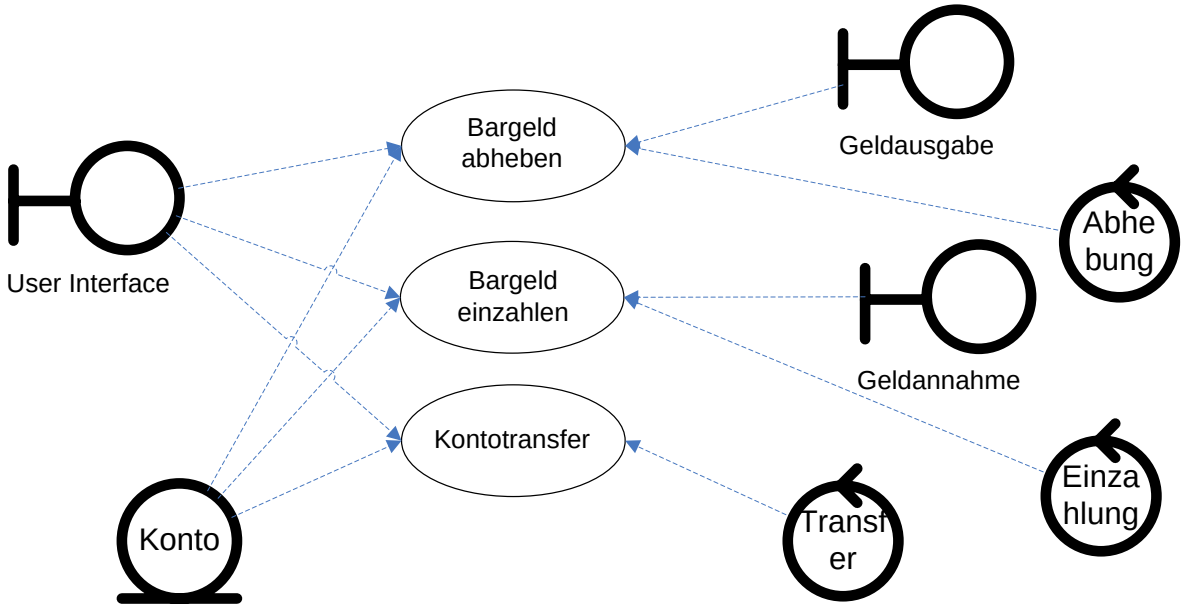


Abbildung 35 Bestimmen der Analyse-Klassen

Daraus kann dann das Analysemodell abgeleitet werden:

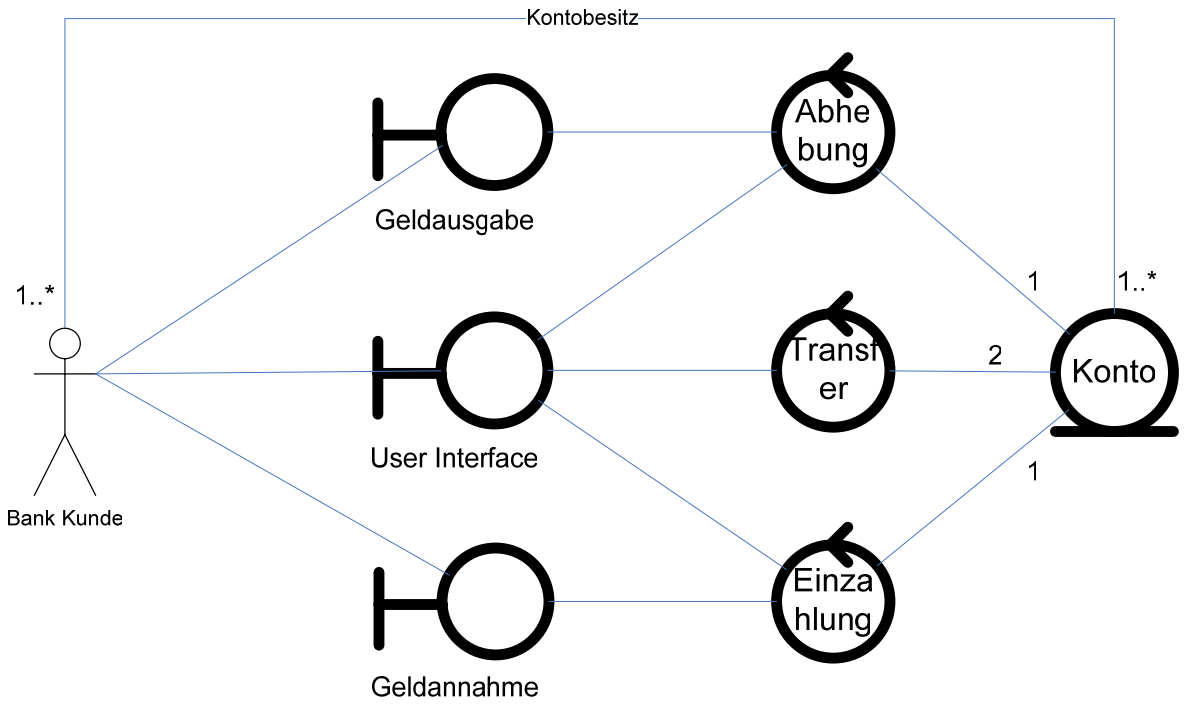


Abbildung 36 Analyse-Klassenmodell

11. Architektur und Design

Eine gute Architektur macht es viel einfacher Änderungen an einem System vornehmen zu können. Die Schnittstellen sind klar definiert und meist auch gut dokumentiert. Die Architektur selber beschäftigt sich nicht mit Details der Implementation.

Eine Architektur beschreibt die einzelnen Komponenten, deren Attribute und Aufgaben sowie ihre Beziehung untereinander.

RUP und IEEE definieren die Begriffe Architektur und Design auf ihre eigene Weise:

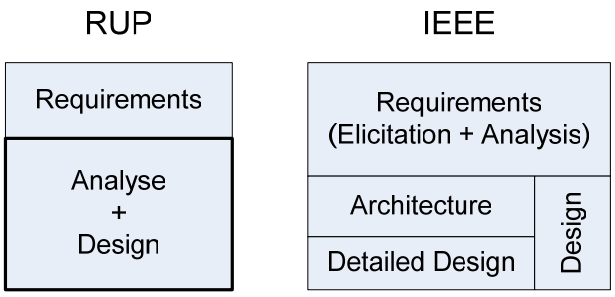


Abbildung 37 RUP vs. IEEE, Architektur und Design

Die Design- und Architekturvorschriften werden im SDD gesammelt (siehe Software Design Document (SDD)).

11.1. Aufgaben der Architektur

Aufgaben des Architekturentwurfes sind:

- Aufgabe analysieren
    - Anforderungen verstehen
    - Vorhandene bzw. beschaffbare Technologien und Mittel analysieren
- Architektur modellieren und dokumentieren
    - Grundlegende Systemarchitektur festlegen
    - Festlegung des Architekturstils
    - Modularisierung
    - Nebenläufige Lösungen in Prozesse gliedern
    - Wiederverwendungs- und Beschaffungsentscheide treffen
    - Ressourcen zuordnen
    - Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen
    - Lösungskonzept (als Dokument) erstellen
- Lösungskonzept prüfen
    - Anforderungen erfüllt?
    - Softwaretechnisch gut?
    - Wirtschaftlich?

Architektur umfasst:

- Techniken
    - Software Struktur (Layer, Client-Server, ...)
    - Datenfluss, Flusskontrolle

    - Kommunikation
    - Verteilung, Parallelität
    - Persistenz
- Qualität:
    - Sicherheit, Stabilität
    - Fehlertoleranz
    - Testbarkeit
    - Unterhaltbarkeit
    - Wiederverwendbarkeit

11.2. Aufgaben des Designs

Sauberes Software-Design hat folgende Ziele:

- Erleichterung beim Hinzufügen neuer Module
- Erleichterung bei der Anpassung bestehender Module
- Erleichterung des Verständnisses durch Einfachheit
- Erleichterung der Implementation durch Einfachheit
- Erhöhung der Effizienz durch Optimierung der Geschwindigkeit
- Erhöhung der Effizienz durch Reduzierung der Grösse

Um dies zu erreichen werden häufig sogenannte Design Patterns eingesetzt. Dies sind vorgefertigte Konstrukte um gewisse Probleme zu lösen. Zudem bieten einige Programmiersprachen diese von sich aus bereits an. Typische Design Patterns sind „Iterator“, „Observer“ oder „Facade“.

Durch den Einsatz eines Frameworks kann der Entwicklungsaufwand häufig noch weiter reduziert werden. Die Java Runtime (API) ist ein Beispiel für ein solches Framework. Es bietet bereits viele vorgefertigte Funktionalitäten, diese müssen natürlich nicht mehr selbst implementiert werden.

12. Komponenten und Verbinder (Components and Connectors)

Eine Software sollte aus Komponenten und Verbindern bestehen. Das macht die einzelnen Komponenten austauschbar.

Verbinder (Connectors) sind Mechanismen, die den Datenaustausch zwischen den Komponenten erlauben. Sie sind implementiert mit COTS, Technologien und COTS. Das Format der ausgetauschten Daten kann selbstdefiniert oder standardisiert sein.

12.1. Komponenten

Komponenten (Components) sind Laufzeitobjekte und Datenablagen. Sie sind implementiert mit eigenem Code, Wiederverwendbarem Code und COTS (Component of the shelf, Vorgefertigtem Code).

Eigenschaften von Komponenten:

- Um eine Komponente zu nutzen muss man ihre Export-Schnittstellen kennen.
- Haben zwei Komponenten die selben Export-Schnittstellen so sind sie austauschbar.
- Eine Komponente ist testbar ohne den Inhalt zu kennen.
- Komponenten können unabhängig voneinander entwickelt werden.
- Komponenten sind wichtig für die Wiederverwendung

Vorteile von Komponenten:

- Wiederverwendbarkeit
    - Komplexität kann aufgebrochen werden
    - Kürzere Auslieferungszeit (wiederverwenden ist schneller als neu schreiben, Tests können entfallen)
    - Standardisierte Komponenten sichern die Konsistenz
    - Die Entwicklung beginnt nicht immer bei null: Bessere Qualität
- Vorteile von Schnittstellen:
    - Produktivitätssteigerung
    - Qualitätssteigerung durch klar definierte Schnittstellen und bessere Tests
- Ersetzungs-Vorteile
    - Entwicklung und Test können schneller durchgeführt werden
    - Es kann parallel entwickelt werden
    - Änderungen an einer Komponente bedingen nicht die Änderung des gesamten Systems

12.1.1. UML 2.0 Komponenten Notation

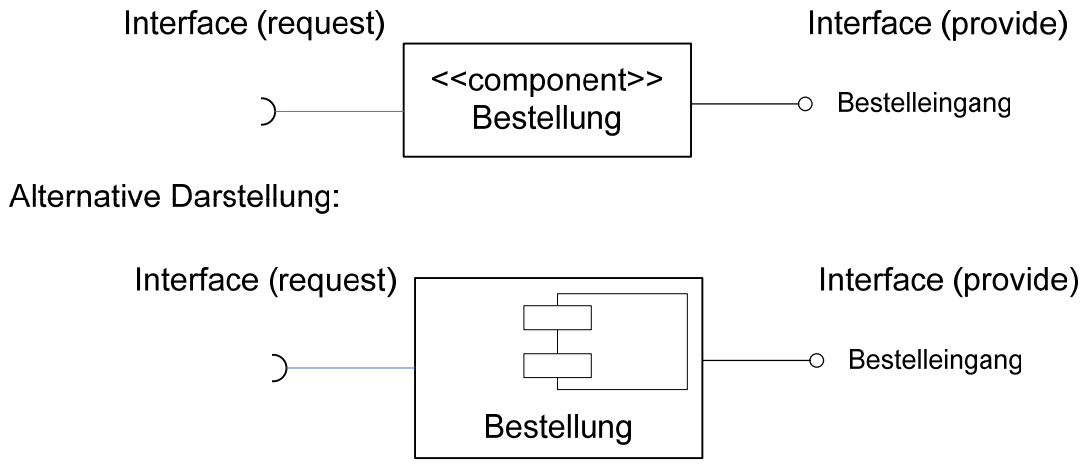


Abbildung 38 UML 2.0: Komponenten Darstellung

12.2. Komponenten und Module

Was Ist eine Komponente?

- Eine Komponente ist ein nicht trivialer, fast unabhängiger und austauschbarer Teil eines Systems mit einer klaren Funktion im Kontext einer klar definierten Architektur.
- Eine Komponente kann installiert und/oder instanziert werden und hat somit eine Laufzeit-Präsenz.
- Eine Komponente gehört zu einer bestimmten Schnittstelle und bietet die Realisierung einer Sammlung von Schnittstellen.
- Es gibt eine Komponenten Spezifikation und Implementation.

Was ist ein Modul?

- Ein Teil der Software für Speicherung und Manipulierung.
- Implementiert eine Sammlung von „Responsibilities“.

12.2.1. Coupling and Cohesion

Coupling:

- Grad der Kommunikation zwischen den Modulen. Gibt an wie unabhängig die Module sind. Der Grad der Kopplung sollte minimiert werden (einfacher Austausch möglich).

Änderungen innerhalb eines Moduls sollte keine Anpassungen an den aussenstehenden (benutzenden oder benutzen) Modulen mit sich ziehen (low coupling). Der innere Aufbau des Modules sollte nach aussen nicht sichtbar sein (information hiding, black-box). Dadurch wird die Modulabhängigkeit kleiner und die Wartung einfacher.

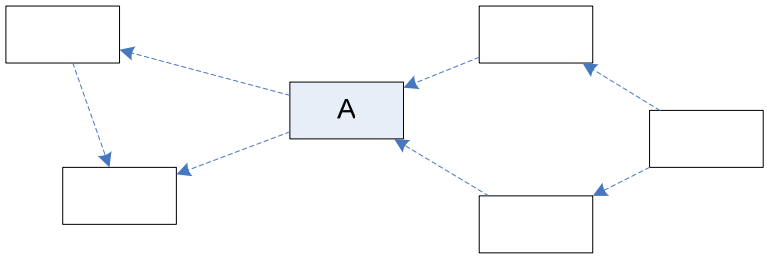


Abbildung 39 Coupling

Cohesion:

- Grad der Kommunikation innerhalb der Module. Gibt an wie stark die Elemente eines Moduls untereinander verknüpft sind. Die Kohäsion sollte so hoch wie möglich sein. Ist sie niedrig oder gar nicht vorhanden wurden die falschen Komponenten zu einem Modul zusammengefasst (unterschiedliche Funktionalitäten).

Ein Modul soll für eine geschlossene Aufgabe zuständig sein. Dies kann nach unterschiedlichen Aspekten geschehen:

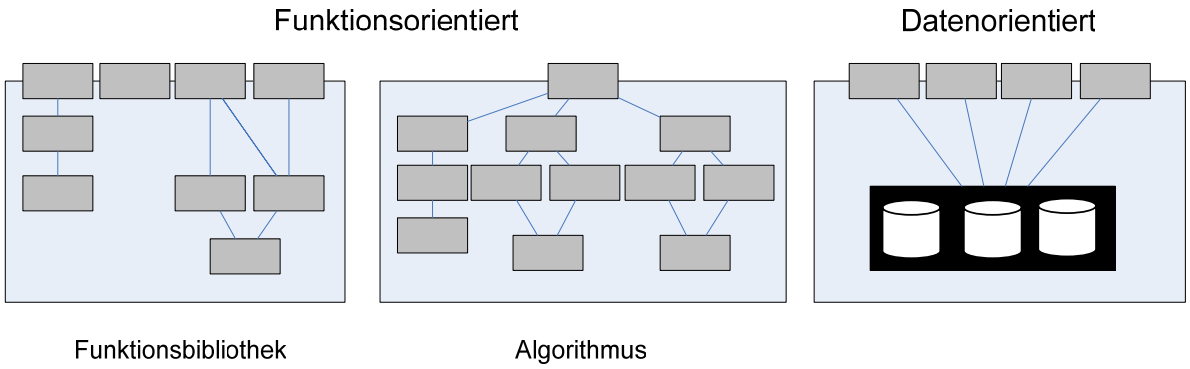


Abbildung 40 Cohesion

Die Abhängigkeit der Kopplung und Kohäsion von der Projektgrösse:

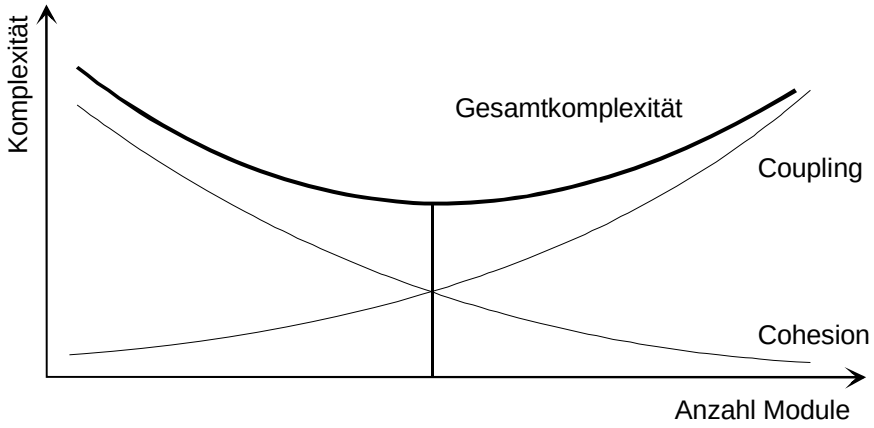


Abbildung 41 Coupling and Cohesion

12.3. Verbinder

Zu beachten gilt es insbesondere folgende Punkte:

- Kontrollfluss (Control Flow)
  - Welche Komponente initialisiert die Verbindung?
  - Welche Komponente beendet die Verbindung?
  - Kann die Komponente beendet werden bevor der Datenaustausch komplett ist?
- Datenfluss (Data Flow)
  - Welches Datenformat wird verwendet?
  - In welche Richtung fliessen die Daten (eine oder beide Richtungen)?
- Übertragungsmedium (Transmission Media)
  - Welches Medium wird verwendet (Papier, Stimme, elektronisch)?
  - Brauchen die Daten übersetzt zu werden?

12.3.1. Push/Pull Beispiel

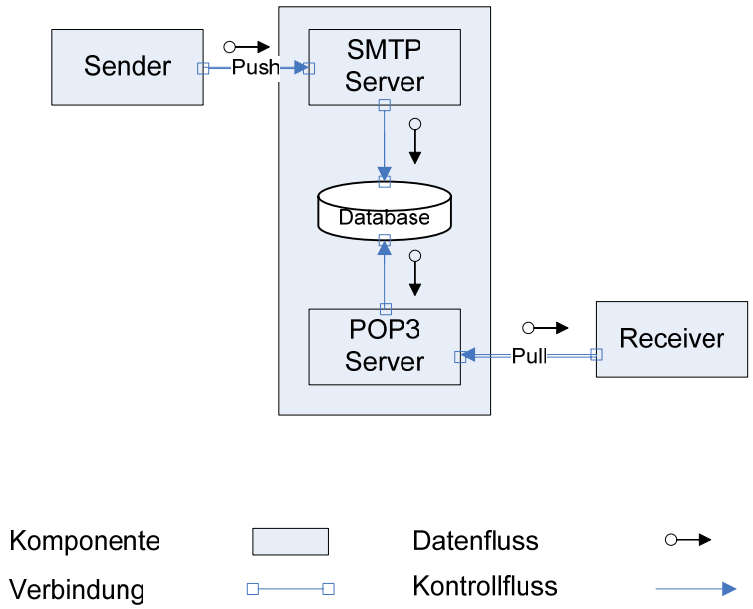


Abbildung 42 Push/Pull Example

12.4. Schnittstellen-Typen

Schnittstellen können nach verschiedenen Typen unterschieden werden:

Operational Interface: Eine Komponente bietet eine Sammlung von Diensten. Diese werden von sogenannten call-interface Operationen aufgerufen:

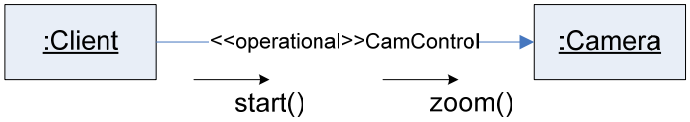


Abbildung 43 Operationale Schnittstellen (Operational Interface)

Signal Interface: Eine Sammlung von Signalen, die von einer Komponente gesendet oder empfangen werden können.

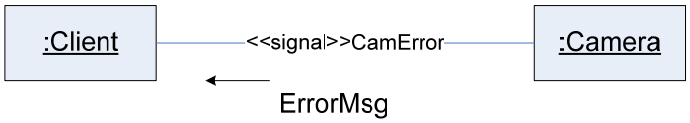


Abbildung 44 Signal Interface

Stream Interface: Sammlung von Datenströmen, die von einer Komponente benutzt oder erzeugt werden können.

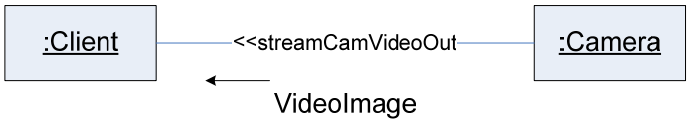


Abbildung 45 Stream Interface

12.5. Layer

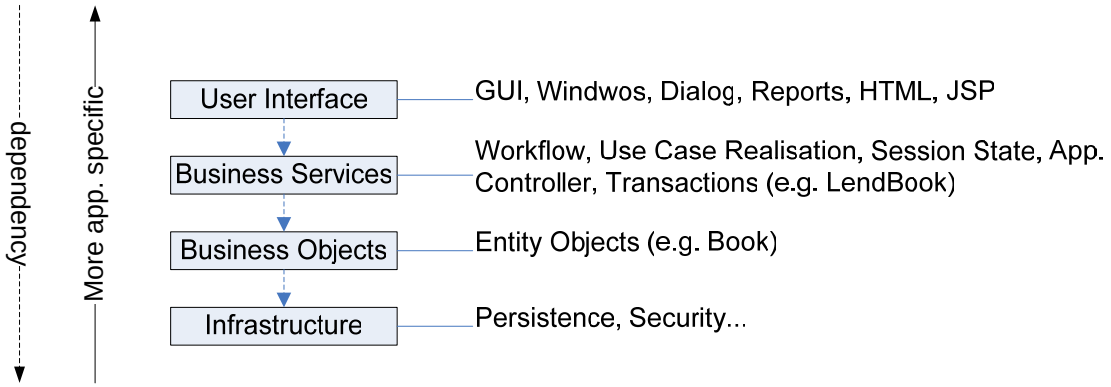


Abbildung 46 Layer Architektur

Jeder Layer hat bestimmte Aufgaben und kommuniziert jeweils mit dem direkt darunter angeordneten. Layer können auch horizontal nochmals unterteilt sein. Diese Unterteilungen nennt man dann Partitionen. Layer stellen eine geordnete Kommunikation nach Funktionseinheiten sicher.

Layer sollten nicht mit Tiers verwechselt werden. Bei einer Multi-Tier Architektur kommunizieren die einzelnen Schichten bidirektional miteinander. Bei Layern in der Regel nicht.

13. Design Pattern

Design Pattern sind vorgefertigte Design-Elemente. Diese können die Entwicklung beschleunigen indem sie helfen die Applikation strukturiert aufzubauen. Ausserdem bieten viele Frameworks (wie die Java Runtime) vorgefertigte Lösungen für diverse Pattern, die man dann nicht mal mehr zu implementieren braucht.

13.1. Singleton

Ein Singleton garantiert, dass es nur eine Einzige Instanz einer bestimmten Klasse gibt. Für die referenzierenden Objekte muss es einfach sein Objekte muss es einfach sein dieses anzusprechen.

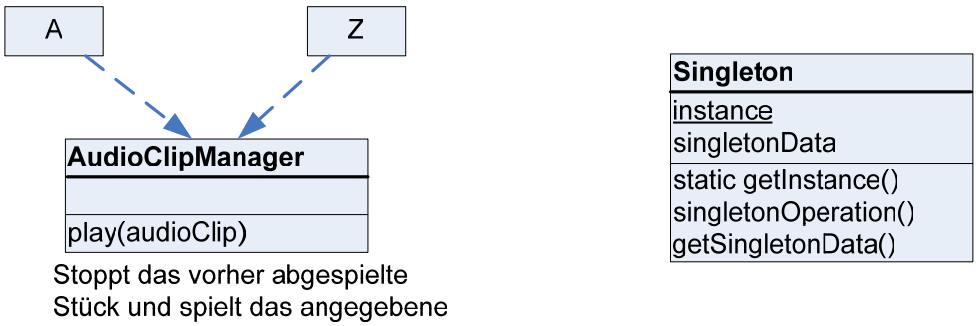


Abbildung 47 Singleton

Implementation in Java:

```
public Class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {} // keinen öffentlichen Konstruktor

    public static Singleton getInstance() {
          return instance;
    }

    Public void singletonOperation() {}
}
```

13.2. Facade

Eine Facade stellt ein einheitliches Interface gegen aussen dar (z.B. eines Teilsystems). Der Vorteil liegt darin, dass die interne Struktur der Anwendung oder Teilen davon nicht bekannt sein müssen, da eine Einheitliche Schnittstelle zum System zur Verfügung gestellt wird. Umstrukturierungen im inneren der Applikation bedingen nicht die Änderung der umliegenden Systeme.

Die Facade ist die einzige Instanz, welche auf die Objekte im Subsystem zugreifen kann. Die Facade muss sich auch um die Erstellung und Handhabung der innerhalb des Subsystems befindlichen Objekte kümmern.

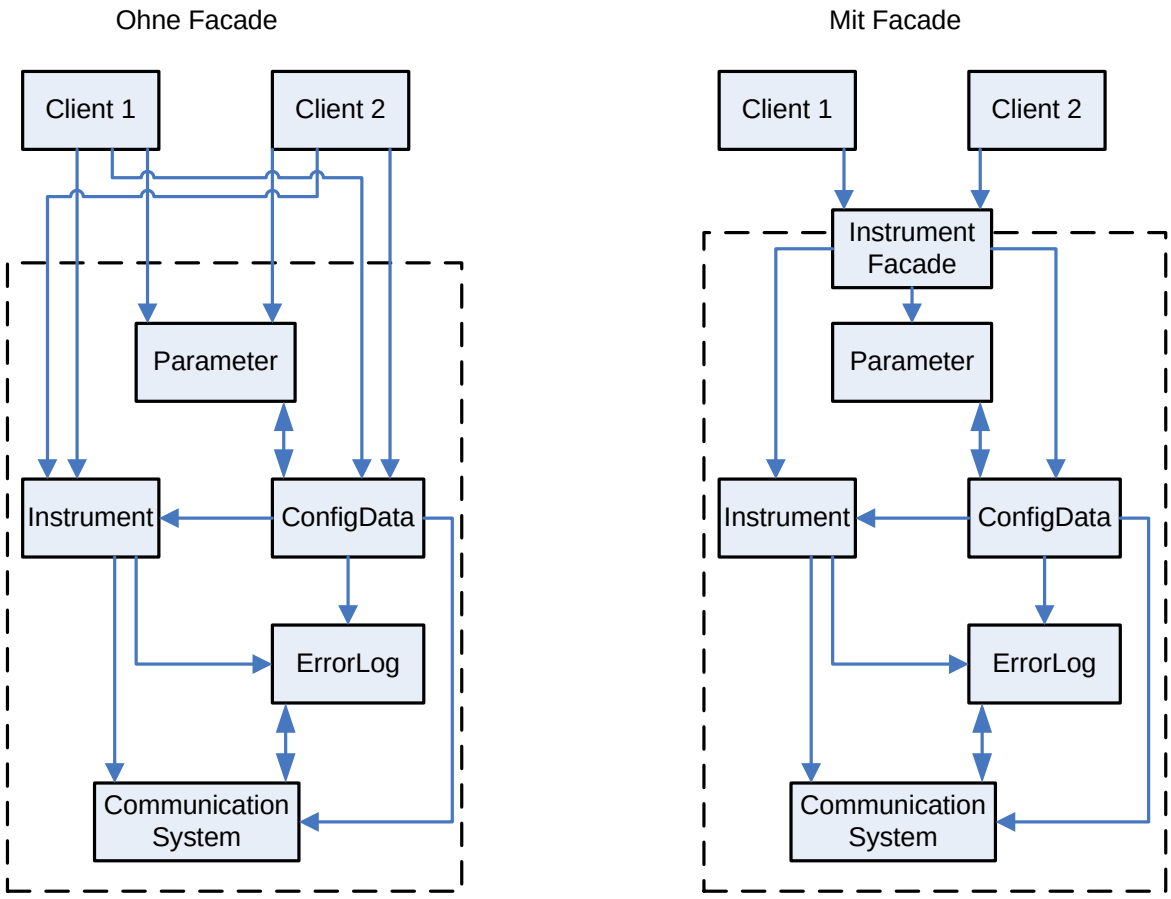


Abbildung 48 Facade

Implementierung:

- Kann als Klasse Modelliert sein.
- Kann entweder ein Singleton sein oder mehrere Instanzen erlauben.
- Bei mehreren Kommunikationswegen kann jede Instanz der Facade eine Verbindung repräsentieren.

13.3. Composite

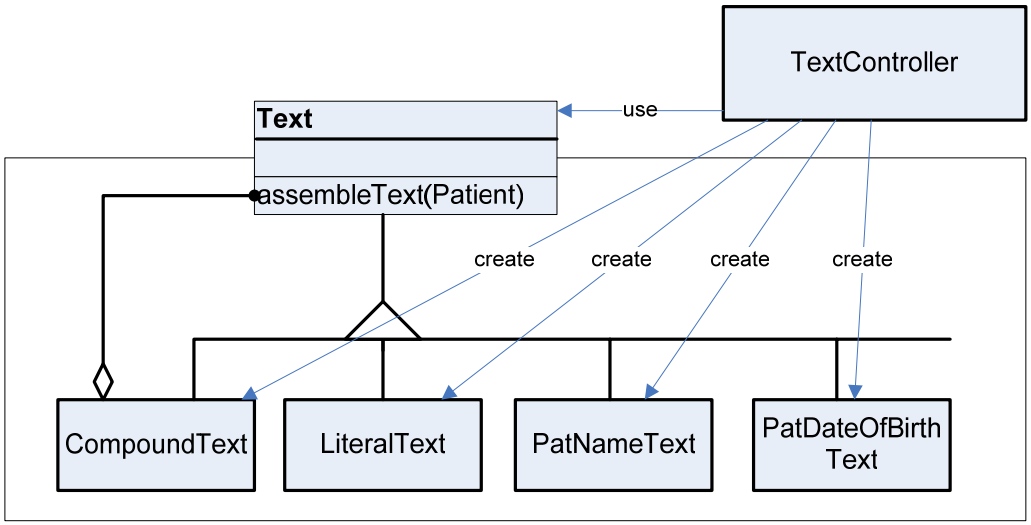


Abbildung 49 Text Skeleton modelliert mit dem Composite Pattern

Das Composite Pattern schliesst das Subsystem vollständig ein. Es bietet ein einheitliches Protokoll für alle Klassen eines Subsystems. Über eine Factory-Methode lassen sich neue Instanzen der konkreten Klassen erzeugen und die Referenzen werden zurückgegeben.

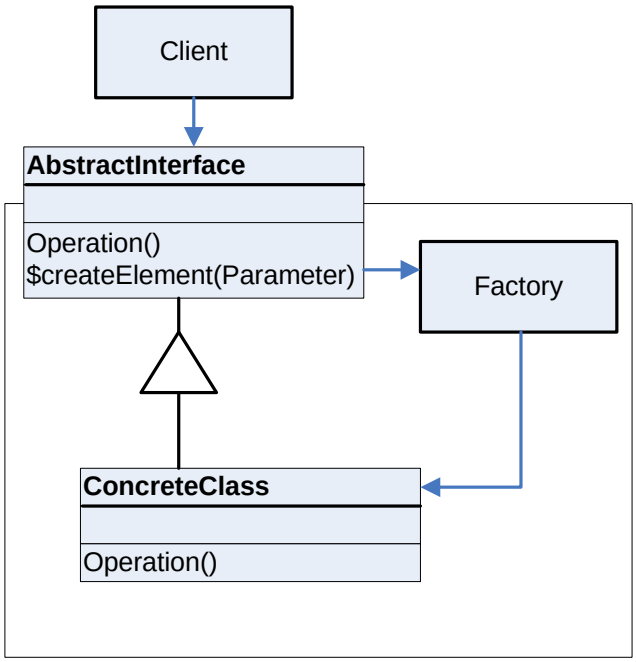


Abbildung 50 Composite Pattern

Concrete Classs:

- Implementiert das Protokoll des AbstractInterface
- Weiss für welche Patameter des createElement Parameters es zuständig ist

Factory:

- Kennt alle konkreten Klassen
- Findet die konkrete Klasse, welche für einen bestimmten createElement Parameter zuständig ist
- Erzeugt die konkrete Klasse
- Meist als Singleton implementiert

Client:

- Benutzt nur das Protokoll aus AbstractInterface
- Kennt die ConcreteClass nicht

13.4. Inverted Associations / Callback

Ein Client registriert sich beim Server. Dazu muss er ein Client Protokoll implementieren welches zum Server Subsystem gehört. Durch die Anmeldung am Server ist der Server in der Lage ein Callback auf den Client durchzuführen.

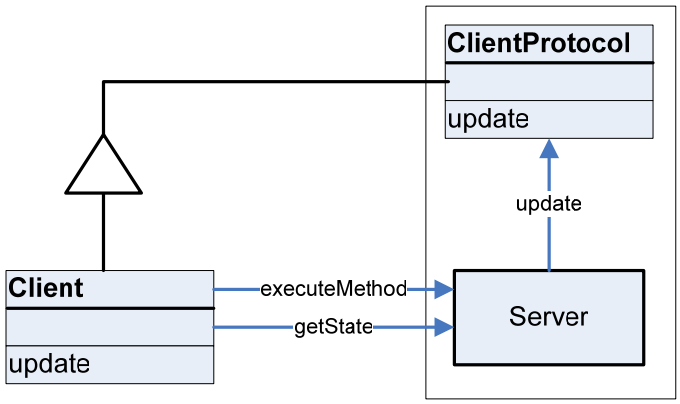


Abbildung 51 Inverted Association

Client:

- Benutzt Dienste des Servers
- Überwacht Server-Status

Client Protocol:

- Gehört zum Server Subsystem
- Definiert die vom Client geforderten Methoden

Server:

- Bietet Dienste an (unabhängig vom Client)
- Benutzt das Client Protokoll für Callbacks

13.5. Observer

Das Observer Pattern besteht grundsätzlich aus einem Observer und einem Observable Objekt. Das Observable Objekt nimmt Anmeldungen von Observern entgegen. Tritt ein bestimmtes Ereignis ein werden die Observer davon in Kenntnis gesetzt (notify).

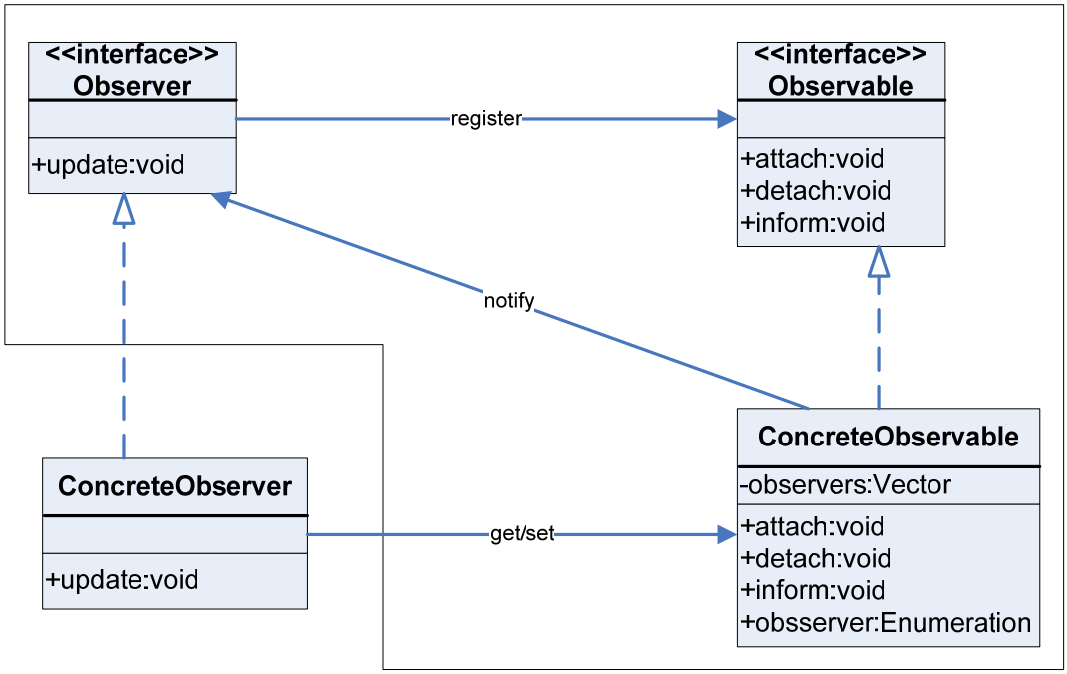


Abbildung 52 Observer

14. Abbildungsverzeichnis

15. Tabellenverzeichnis

16. Index