

Software Engineering

Zusammenfassung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Author: Ruben Deisenroth
Basierend auf: Merktzettel von Fabian Damken
Stand: 21. Februar 2021

Semester: WiSe 2020/21

Fachbereich: Informatik

Inhaltsverzeichnis

1	Einführung	3
1.1	Software	3
1.1.1	Arten von Software	3
1.1.2	Softwarecharakteristiken	3
1.2	Engineering	4
1.2.1	Characteristics of Engineering Approaches	4
1.2.2	Probleme der Softwareentwicklung	4
2	Requirements Engineering	5
2.1	Anforderungsanalyse	5
2.1.1	Nutzeranforderungen (User requirements)	5
2.1.2	Systemanforderungen (System Requirements)	5
2.1.3	Domänenanforderungen (Domain Requirements)	6
2.1.4	Funktionale Anforderungen (Functional Requirements)	6
2.1.5	Nichtfunktionale Anforderungen (Non-Functional Requirements)	6
2.1.6	RE-Prozess	7
3	Anwendungsfälle (Use Cases)	7
3.1	Use Case Analysis	7
3.1.1	Requirements vs Use Cases	7
3.1.2	Use Case Formats	8
3.1.3	Richtlinien für das Entwickeln von Anwendungsfällen	8
3.2	UML Nutzungszweck Diagramme (UML Use Case Diagrams)	9
4	Domänenmodellierung (Domain Modelling)	10
4.1	Diagramm: Domain Model (UML)	11
4.1.1	Klasse vs Attribut	12
4.1.2	Attribut vs Verbindung	12
4.1.3	UML Zustandsdiagramm (State Machine Diagram)	12
5	Softwarearchitektur (Software Architecture)	13
5.1	Architekturcharakteristiken	14
5.2	Architekturstile	14
5.2.1	Monolithische Architekturstile (Monolithic Architecture Styles)	15
5.2.2	Verteilte Architekturstile (Distributed Architectural Styles)	16
6	Designprinzipien (Design Principles)	17
6.1	Softwarequalität	17
6.1.1	Faktoren	17
6.2	Messen von Softwarequalität	18
6.2.1	Übersicht Metriken zum Messen von Softwarequalität	18

6.2.2	Zyklomatische Komplexität (Cyclomatic Complexity)	18
6.2.3	Kontrollflussgraph (CFG)	18
6.2.4	Heuristiken	19
6.2.5	Verknüpfen/Koppeln (Coupling)	20
6.2.6	Zuständigkeiten	21
6.2.7	Kohäsion(Cohesion)	21
6.3	Designprinzipien	22
6.3.1	Single-Response-Principle (SRP)	22
6.3.2	Inheritance vs. Delegation	22
6.4	Kapselung(Encapsulation)	22
6.4.1	Zugriffsrechte (Field Access)	23
6.5	Probleme	23
6.5.1	Gottklassen (God Classes)	23
6.5.2	Class Proliferation	23

1 Einführung

1.1 Software

Definition – Software Der Begriff *Software* bezeichnet ein Programm und all die dazugehörigen Daten, Informationen und Materialien. Das beinhaltet z.B.

- Ein (installierbares), ausführbares (operational) Programm und dessen Daten
- Konfigurationsdateien (configuration files)
- Systemdokumentationen (system documentation), z.B. Architekturmodell, Designvorstellungen, ...
- Nutzerdokumentationen (user documentation), z.B. Anleitung, ...
- Support (z.B. Wartung, Website, Telefon, ...)

- W.S. Humphrey, SCM SIGSOFT, 1989

1.1.1 Arten von Software

Typ	Beschreibung
Application Software	- interagiert direkt mit dem Nutzer - Kann sowohl General purpose (z.B. Textverarbeitung, ...) als auch anwendungsspezifisch (z.B. Kassensystem, ...) sein
System Level Software	- interagiert üblicherweise nicht direkt mit Nutzer - Sorgt für funktionsfähiges System (z.B. Treiber, ...)
Software as a Service (SaaS)	- Läuft auf einem Server - Zugriff meist nur indirekt über Client (z.B. über Browser, SSH, ...)

TABELLE 1: Arten von Software

1.1.2 Softwarecharakteristiken

- Software nutzt sich nicht ab, aber muss sich ständig anpassen (z.B. an neue Hardware, ...)
- Lebenszeit meist länger als erwartet (schwer zu bestimmen, da zukünftige Anforderungen ungewiss)
- Nur schwer messbar: Softwarequalität, Entwicklungsfortschritt und Zuverlässigkeit

1.2 Engineering

Definition – Engineering (Entwickeln, Ingenieurwesen) bezeichnet den Prozess, wissenschaftliche Prinzipien zu nutzen, um Maschinen, Strukturen und viele weitere Dinge zu entwerfen.

- Cambridge Dictionary

Definition – Software Engineering ist ein Teilgebiet der Informatik und bezeichnet das Anwenden eines systematischen disziplinierten und qualifizierten Ansatzes der Entwicklung, Ausführung und Wartung der Software, sowie die Forschung und Entwicklung solcher Ansätze.

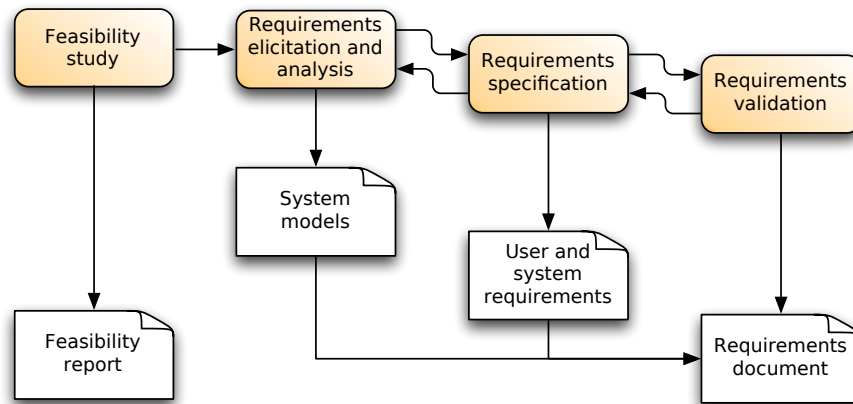
1.2.1 Characteristics of Engineering Approaches

- Fester/Definierter Prozess
- Getrennte Entwicklungsphasen, typischerweise: Analyse, Entwicklung (Design), Evaluation der Entwicklung, Konstruktion, Qualitätskontrolle (z.B. TÜV, oder mathematisch Korrektheit beweisen)
- Klare Trennung einzelner Teilsysteme

1.2.2 Probleme der Softwareentwicklung

- Kunde/Nutzer wenig/gar nicht an Entwicklung beteiligt
- Konstant verändernde Anforderungen an die Software
- Softwareentwickler oft nicht ausreichend geschult
- Managementprobleme
- Unpassende Methoden, Programmiersprachen, Tools

2 Requirements Engineering



(from: I. Sommerville, Software Engineering, Pearson)

2.1 Anforderungsanalyse

Definition – Anforderungen (requirements) sind die Beschreibungen der vom entworfenen System zu erfüllenden Aufgaben und dessen Einschränkungen

Die Anforderungsanalyse beschäftigt sich mit dem Erkennen, der Analyse, der Dokumentation und der Validierung der Anforderungen.

- Anforderungen werden in einem sog. Pflichtenheft (System Requirements Specification) festgehalten.
- use cases, state diagrams, usw werden im product backlog festgehalten.
- Anforderungen sind **keine** Lösungen/Implementierungen.

2.1.1 Nutzeranforderungen (User requirements)

- beschreiben Aufgaben und Einschränkungen in Natürlicher Sprache oder mit Diagrammen (meinst von Kunden geschrieben)

Beispiel: Das system soll alle Buchungen speichern, so wie es das Gesetz verlangt.

2.1.2 Systemanforderungen (System Requirements)

- Präzise und detaillierte Beschreibung von Aufgaben und Einschränkungen des Programmes (Meist von Entwickler geschrieben)
- Verfeinerung der Nutzeranforderungen

Beispiel: Die Buchungen müssen für 10 Jahre gespeichert werden ab dem Zeitpunkt der Buchung.

2.1.3 Domänenanforderungen (Domain Requirements)

- Meist nicht vom Kunden oder Entwickler spezifiziert, sondern von der Domäne (z.B. vom Gesetzgeber, ...)

Beispiel: Für die Polizeiliche Ermittlung muss in Deutschland jede Transaktion für 2 Wochen zwischengespeichert werden.

2.1.4 Funktionale Anforderungen (Functional Requirements)

- Die Dienste, die das System machen können soll
- die Reaktion des Systems auf bestimmte Eingaben und
- das Verhalten des Systems in bestimmten Situationen.

Beispiel: Wenn der Nutzer den Knopf „Neues Dokument“ drückt, wird eine neues Textdokument angelegt.

2.1.5 Nichtfunktionale Anforderungen (Non-Functional Requirements)

Spezifizieren Einschränkungen der Dienste/Funktionen des Systems, welche oft nicht vollständig von Tests abgedeckt werden können:

- Produktanforderungen (Product requirements)
 - Portabilität (Läuft auf verschiedenen Plattformen/lässt sich leicht darauf anpassen)
 - Zuverlässigkeit/Robustheit (Reagiert auch auf unvorhergesehene/Illegale Eingaben und Situationen sinnvoll)
 - Effizienz (Leistung, Speicherplatz)
 - Nutzbarkeit (Verständlichkeit, ...)
- Organisatorische Anforderungen (Organisational requirements)
 - Auslieferungsanforderungen
 - Implementation
 - Nutzung von Standards (ISO, IEEE, ...)
- Externe Anforderungen (External requirements)
 - Interoperabilität (Interoperability requirements), d.h. Zusammenspiel mit anderen Systemen
 - Ethische Anforderungen
 - Rechtliche Anforderungen (Datenschutz, Sicherheit, ...)

Nichtfunktionale Anforderungen sind oftmals großflächige Anforderungen, welche das gesamte System betreffen. Allerdings sind solche Anforderungen meistens kritischer als funktionale Anforderungen (bspw. „Das System soll sicher vor Angriffen sein.“).

Um nichtfunktionale Anforderungen überprüfbar zu machen, ist oftmals eine Umformulierung oder Abänderung der eigentlichen Anforderung nötig. Hierdurch können auch funktionale Anforderungen aufgedeckt werden.

Beispiel: Die Oberfläche soll ansprechend und einfach zu bedienen sein.

2.1.6 RE-Prozess

Viewpoint-Oriented approach

- Interactor viewpoints: direktes Interesse
- Indirect viewpoints: indirektes Interesse
- Domain viewpoints: indirektes Interesse

Definition – FURPS+ Modell Functionality, Usability, Reliability, Performance, Supportability
Plus Implementation (Interface, Operations, Packaging, Legal)

Anforderungvalidierung (Requirements validation checks)

- Korrektheit (Validity): Beinhalten die Anforderungen alle nötigen Funktionen oder werden weitere benötigt?
- Konsistenz: Gibt es Konflikte bei den Anforderungen?
- Komplettheit: Sind alle Funktionen und Einschränkungen wie erwünscht angegeben?
- Realisierbarkeit (Realism): Sind die Anforderungen realistisch erfüllbar?
- Testbarkeit (Verifiability): Lässt sich das Erfüllen der Anforderungen testen?
- Verfolgbarkeit (Tracability): Lässt sich nachvollziehen, warum die Anforderung existiert?

3 Anwendungsfälle (Use Cases)

3.1 Use Case Analysis

Definition – Anwendungsfälle (Use Cases) beschreiben, wie ein Angehöriger einer Rolle (*Akteur*) das System in einem bestimmten *Szenario* nutzt.

3.1.1 Requirements vs Use Cases

Requirements	Use Cases
- Fokus auf gewünschte <i>Funktionalität</i>	- Fokus auf mögliche <i>Szenarios</i>
- Werden meist einfach deklariert (declaratively)	- Werden anhand von Szenarios beschrieben (operationally)
- Perspektive des Clients	- Perspektive des Users

TABELLE 2: Requirements vs Use Cases

- Ohne Nutzerbeteiligung nahezu unmöglich, gute/vollständige Anwendungsfälle zu schreiben
- Anwendungsfälle ergänzen die Anforderungsanalyse, ersetzen sie aber nicht (können keine Nichtfunktionalen Anforderungen erfassen)

3.1.2 Use Case Formats

- kurz (brief): Kurze zusammenfassung, normalerweise das Haupterfolgsszenario
- informell (casual): Informelles Format, mehrere Zeilen die Mehrere Szenarios behandeln
- ausgearbeitet (Fully dressed): Alle Zwischenschritte und Variationen sind im Detail aufgeschrieben, es gibt hilfsektionen, z.B. Vorbedingungen (preconditions) und Erfolgsgarantien (sucess guarantees)

Ein vollständig ausgearbeiteter Anwendungsfall sieht so aus:

Abschnitt	Beschreibung/Einschränkung
Use Case Name	Der Name des Anwendungsfalls / Startet mit einem Verb
Bereich (Scope)	Betroffener Bereich des Systems
Ebene (Level)	Abstraktionsebene/ Nutzerziel, Zusammenfassung oder Unterfunktion
Hauptakteur (Primary actor)	Initiator des Anwendungsfalls
Stakeholders and Interests	Personen, die dieser Anwendungsfall betrifft
Vorbedingungen (preconditions)	Was muss beim start des Programmes gelten? Ist es das Wert dem Leser mitzuteilen?
Akzeptanzkriterien (Minimal Guarantee)	Minimalversprechen an Stakeholders
Erfolgskriterien (Success Guarantee)	Was sollte das Programm können, wenn es erfolgreich ist?
Haupterfolgsszenario (Main Success Scenario)	Typischer Ablauf des Szenarios / Nummerierte Schrittlste um das Ziel zu erreichen
Erweiterungen	Alternative Erfolgs- und Fehlschlagszenarien / Fehlschlagspunkte des Hauptszenarios
Spezialanforderungen	Verwandte, nichtfunktionale Anforderungen
Technologien	Einzusetzende Technologien
Häufigkeit (Frequency of occurrence)	Häufigkeit des Eintretens des Anwendungsfalls
Anderes (Misc)	Bspw. offene Tickets

nicht
immer
nötig

TABELLE 3: Fully Dressed Use Case schema

3.1.3 Ricktlinien für das Entwickeln von Anwendungsfällen

1. Akteure und deren Interessen aufzählen \implies Erste Ebene an Präzission
2. Stakeholders, Trigger (Erster Schritt des Haupterfolgsszenarios), Validieren \implies Zweite Ebene an Präzission
3. Alle Fehlschlagszenarien Indentifizieren und auflisten
4. Fehlerbehandlung Schreiben

3.2 UML Nutzungszweck Diagramme (UML Use Case Diagrams)

Definition – Unified Modeling Language (UML) Visuelle, aber präzise Entwurfsschreibweise für Softwareentwicklung

- Hauptziel: Objektmodellierung vereinheitlichen, indem sich auf einen festen Syntax und Semantik geeinigt wird

Definition – Nutzungszweck-Diagramm (Use case Diagram) Stellt Anforderungsfälle und deren Relationen zu dem System und dessen Akteuren dar.

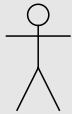
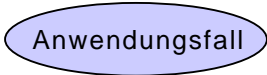
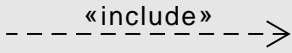
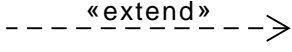
Element	Beschreibung
 Akteur	Stellt einen Akteur innerhalb des Systemes dar.
 Anwendungsfall	Stellt einen Anwendungsfall im System dar
	Erweitert einen Anwendungsfall um eine Funktionalität. Wird der erweiterte Anwendungsfall „ausgeführt“, so wird auch dieser Anwendungsfall „ausgeführt“.
	Erweitert einen Anwendungsfall um eine Funktionalität. Ist die Bedingung in der Beschreibung wahr, so wird der erweiternde Anwendungsfall mit „ausgeführt“.

TABELLE 4: Use Case Diagram Elemente

Beispiel

In einem Autohandel ist es möglich, sowohl Bar als auch mit Kreditkarte zu zahlen. Auch ist es dem Kunden möglich, Automobile zu mieten. Da der Handel neue Kunden gewinnen möchte, ist es ab sofort möglich, bei dem Mieten eines Autos Treuepunkte zu sammeln. Dem Ladeninhaber ist es möglich, neue Autos in das Sortiment aufzunehmen. Wurde ein Ausstellungsauto einmal gemietet, so sinkt der Kaufpreis von diesem.

Diese Anforderungen sind in Abbildung 1 dargestellt.

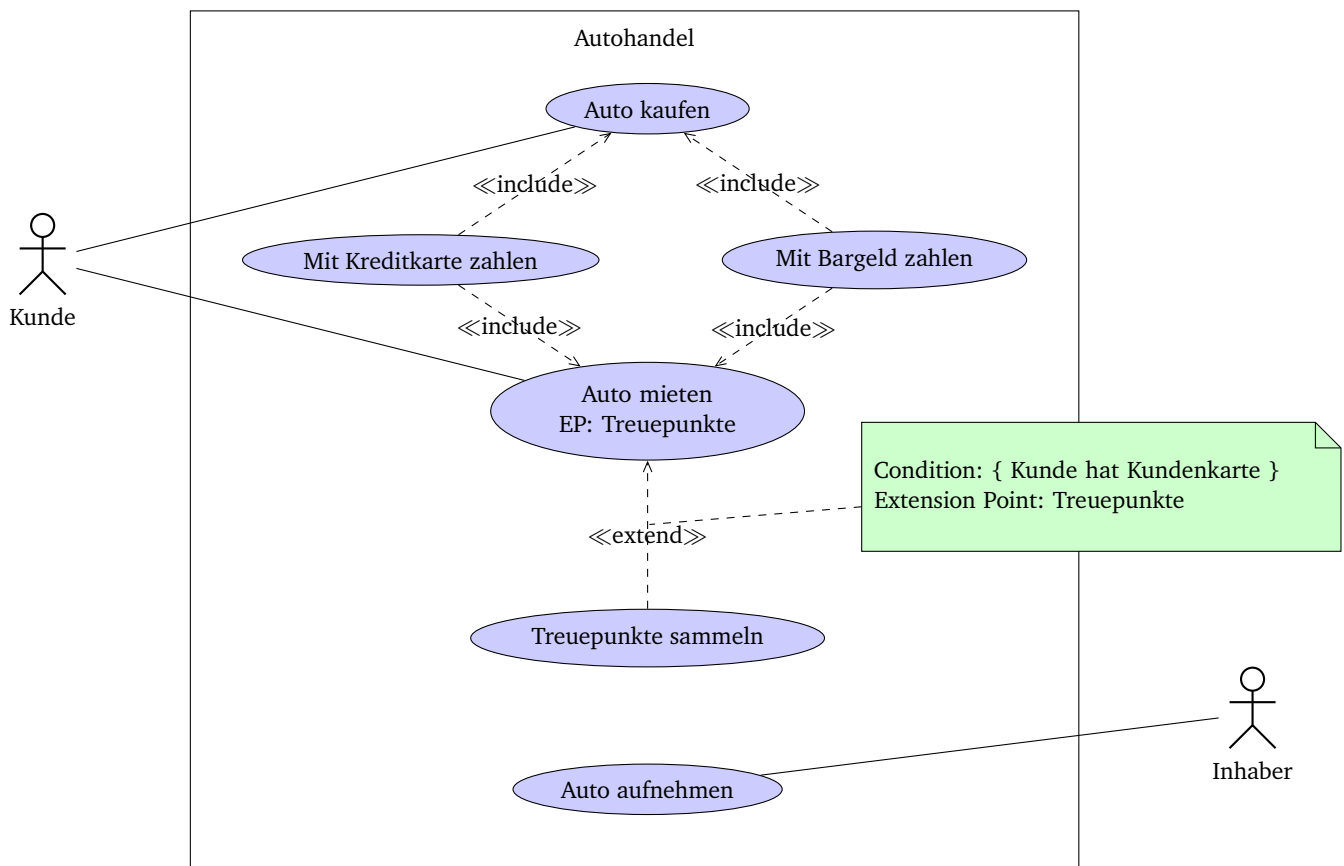


ABBILDUNG 1: Beispiel: Use Case Diagram

4 Domänenmodellierung (Domain Modelling)

Definition – Domain Modelling Reparieren von Terminologie und fundamentalen Aktivitäten im Zielraum (solution space)

Definition – Domänenmodell (Domain Model) Das Domänenmodell besteht aus den **Objekten** (inklusive deren **Attributen**) der Domäne und deren **Beziehung** untereinander. Man modelliert es, indem man während der objektorientierten Analyse die relevanten Konzepte, die aktuell benötigt werden, sowie deren identifiziert. Man benötigt ein tiefgreifendes Verständnis der Domäne (des Einsatzgebietes) für einen guten Softwareentwurf (Curtis Gesetz).

Ein Domänenmodell (Analysemodell, Konzeptmodell)

- spaltet die Domäne in Konzeptobjekte auf,
- sollte die Konzeptklassen ausarbeiten und
- wird iterativ vervollständigt und formt die Basis der Softwareentwicklung.

Domänenkonzepte/Konzeptklassen sind *keine* Softwareobjekte!

4.1 Diagramm: Domain Model (UML)

Beschreibung

Domänenmodelle werden mit Hilfe von einfachen UML Klassendiagrammen visualisiert, wenden aber nur einzelne Teile des Klassendiagramms an:

- Nur Domänenobjekte und Konzeptklassen
- Nur Assoziationen (keine Aggregationen oder Kompositionen)
- Attribute an Konzeptklassen (sollten aber vermieden werden)

Im Diagramm 2 ist ein Beispiel für ein Domänenmodell gegeben. Im folgenden werden die einzelnen Komponenten erläutert.

Domänen-/Konzeptklasse

Stellt ein Domänenobjekt/eine Konzeptklasse im Domänenmodell dar.

Klasse

attribut1: typ1
/abgeleitetesAttribut: typ2

Attribute: Logische Datenwerte eines Objektes, Abgeleitete Attribute werden mit einem „/“ vorm namen gekennzeichnet

roleA	Name	roleB
multA		multB

Repräsentiert eine bidirektionale Assoziation. Ließ: Ein A hat multB viele B und ein B hat multA viele A.

roleA	Name	roleB
multA		multB

Repräsentiert eine unidirektionale Assoziation. Ließ: Ein A hat multB viele B.

Stellt eine Vererbungsbeziehung (Association) dar.

Beispiel

In einer Universität wird jede Vorlesung von mindestens einem Dozenten gelesen. Im Rahmen der Vorlesungen werden Arbeiten angefertigt, welche die Studierenden in Lerngruppen von bis zu 3 Personen bearbeiten müssen. Hierbei kann jeder Studierende von genau einem Dozenten betreut werden, wenn der*die Student*in dies erfragt. Außerdem besuchen Studierende Vorlesungen. Erscheinen keine Studierenden bei einer Vorlesung, so findet diese nicht statt. *Diese textuelle Beschreibung sind im Diagramm 2 dargestellt.*

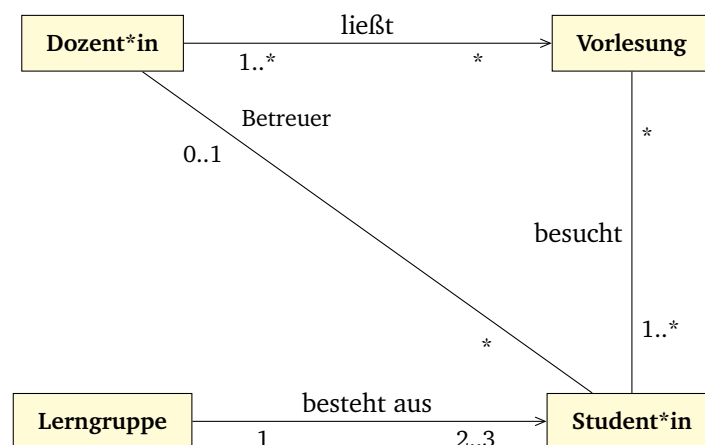


ABBILDUNG 2: Beispiel: Domänenmodell

4.1.1 Klasse vs Attribut

Definition – Beschreibungsklasse (Description Classes) Enthält Informationen/Attribute die ein Objekt beschreiben
Nötig, wenn:

- Informationen über ein Objekt oder eine Funktion benötigt wird
- Löschen des beschriebenen Objektes zu Datenverlust führt
- Informationsdopplung vermieden werden kann

Heuristik: Klasse oder Attribut – Wenn wir bei eine Konzeptklasse C nicht als eine Zahl, einen Text, oder ein Datum der echten Welt sehen, so ist C höchst wahrscheinlich eine Konzeptklasse, und kein Attribut.

Heuristik: Verbindung einfügen? Wenn mehrere Informationen, durch die Verbindung über einen längeren Zeitraum vorhanden sein sollen

Definition – Class name-Verb phrase-Class name - Format Wörter statt mit Leerzeichen mit „-“ trennen, Klassennamen im CamelCase

Verbindungsnamen

- im Class name-Verb phrase-Class name-Format
- Präzise

4.1.2 Attribut vs Verbindung

Attribut	Verbindung
• Primitive Datentypen sind immer Attribute	• Relationen zwischen Konzeptklassen

TABELLE 5: Attribut vs Verbindung

4.1.3 UML Zustandsdiagramm (State Machine Diagram)

Beschreibung

State Machine Diagramme nutzen vereinfachte endliche Automaten zur Darstellung von Eventgetriebenem Verhalten des Systems (Verhalten) und Interaktionssequenzen (Prot koll).

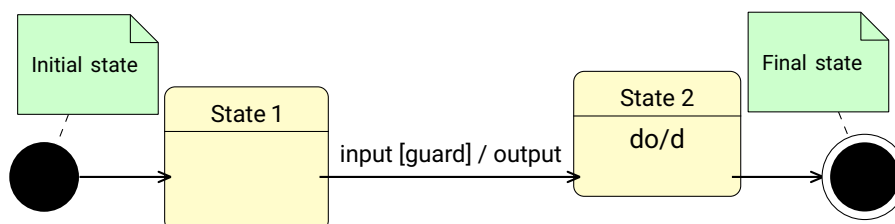


ABBILDUNG 3: Beispiel für UML State Machine Diagram

5 Softwarearchitektur (Software Architecture)

Softwarearchitektur umfasst:

- Architekturcharakteristiken
- Architekturstyles (architecture styles, software system structure)
- Architekturentscheidungen
- Entwurfsmuster (design principles)

Architekten	Entwicklungsteam
<ul style="list-style-type: none"> • Die Charakteristiken der Architektur aus der Abhängigkeitsanalyse zu ermitteln 	<ul style="list-style-type: none"> • Klassenstruktur für jede Komponente erstellen
<ul style="list-style-type: none"> • Einen Architekturstil für das System auswählen 	<ul style="list-style-type: none"> • UI(User Interface) entwerfen
<ul style="list-style-type: none"> • Komponentenstruktur erstellen (über Klassenebene) 	<ul style="list-style-type: none"> • Quellcode schreiben und testen

TABELLE 6: Verantwortungsaufteilung bei der Softwarearchitektur

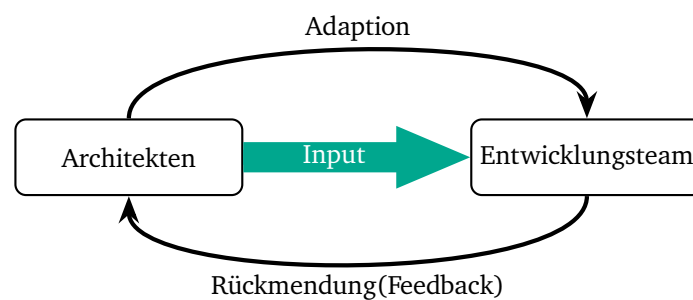


ABBILDUNG 4: Relation Architekten-Entwickler

5.1 Architekturcharakteristiken

Definition – Architekturcharakteristik

- Spezifiziert Operations-und Betriebskriterien um eine gewisse Anforderung zu implementieren
- Beeinflusst den Strukturentwurf, benötigt spezielle Architekturelemente (keine üblichen)
- Es ist nötig, dass die Anwendung wie gewünscht funktioniert (funktionale und nichtfunktionale Abhängigkeiten)

Operationscharakteristiken (Operational):

- Verfügbarkeit (Availability): Zeitspanne, in der das System online/verfügbar ist
- Leistung (Performance): Umfasst Spitzenauslastungsanalysen, Antwortzeiten, Stresstesten
- Skalierbarkeit: (Scalability): Fähigkeit auch mit einer Steigenden Anzahl an Anfragen Klarzukommen

Strukturell (Structural):

- Erweiterbarkeit (Entensibility): Wie einfach es ist, neue Funktionalität hinzuzufügen
- Wartbarkeit (Maintainability): Wie einfach es ist, das System zu verbessern oder auszuwechseln (also zu warten)
- Wiederverwendbarkeit (Leveredgability): Häufige Komponenten in mehreren Produkten wiederverwenden
- Lokalisierbarkeit (Localisation): Unterstützung mehrerer Sprachen, Währungen, Einheiten, ...
- Konfigurierbarkeit (Configuration): Möglichkeit für den Nutzer, das System nach seinen Vorlieben durch eine benutzbare Oberfläche anzupassen.

Cross-Cutting (Divide and Conquer)

- Barrierefreiheit: Muss auch nutzbar von Leuten mit Behinderung (Blinde, Taube, ...) sein
- Datenschutz (Pricary): Möglichkeit, bestimmte Daten für andere Nutzer (auch priveligierte) unzugänglich zu machen
- Sicherheit (Security): Verschlüssellung, Authentifizierung, ...

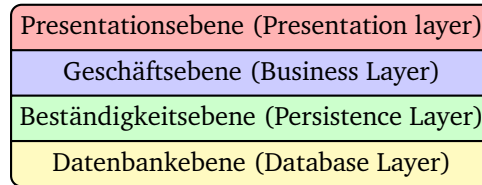
5.2 Architekturstile

- Helfen die Fundamentale Struktur eines Systems zu spezifizieren
- Haben einen großen Einfluss darauf, wie die fertige Architektur aussieht
- Definieren die Globalen Eigenschaften des Systemes (z.B. Wie daten ausgetauscht werden können, welche Einschränkungen die Subsysteme haben)

Softwaresysteme sind normalerweise aus mehreren Architekturstilen zusammengesetzt

5.2.1 Monolithische Architekturstile (Monolithic Architecture Styles)

Ebenen (Layered)



- Anzahl der Ebenen nicht festgelegt, manche Architekturen fassen Ebenen zusammen oder fügen neue hinzu

pro	contra
• Simplizität	• Skalierbarkeit
• Kosten	• Leistung (Parallelisierung nicht unterstützt)
• Architekturstil kann später ausgetauscht werden	• Verfügbarkeit (Lange startzeiten, ...)
• Gut für kleine-mittlere Anwendungen	•

TABELLE 7: Layered-Architekturstil Pro Kontra

Model-View-Controller (MVC)

Das MVC-Muster spaltet die Software in die fundamentalen Teile für interaktive Software auf:

- Model: Enthält die Kernfunktionalität und Daten
 - Unabhängig von dem Ausgabeformat und dem Eingabeverhalten
- View: Präsentiert die Daten dem Nutzer
 - Die Daten werden von dem Modell geladen
- Controller: Verarbeitet die Eingaben des Nutzers
 - Jeder View wird ein Controller zugewiesen
 - Empfängt Eingaben (bspw. durch Events) und übersetzt diese für das Modell oder die Views
 - Jede Interaktion geht durch den Controller

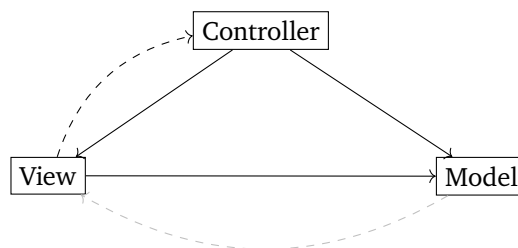


ABBILDUNG 5: Model-View-Controller

Controller und View sind direkt gekoppelt mit dem Modell, das Modell ist nicht direkt gekoppelt mit dem Controller oder der View (siehe 5).

Nachteile:

- Erhöhte Komplexität: Die Aufspaltung in View und Controller kann die Komplexität erhöhen ohne mehr Flexibilität zu gewinnen.
- Update Proliferation: Möglicherweise viele Updates; nicht alle Views sind immer interessiert an allen Änderungen.
- Kopplung View/Controller: View und Controller sind stark gekoppelt.

5.2.2 Verteilte Architekturstile (Distributed Architectural Styles)

Dienstbasierend (Service-Based)

- Hauptvariante:
 - Nur eine Bedienoberfläche (UI) für alle Services
 - Services bestehen aus mehreren Komponenten
 - Alle Services greifen auf eine gemeinsame Datenbank zu
- Nicht-Monolithische Variante (Non-Monolithic)
 - Servicebasierende UIs (eine UI per service)
- Servicelokale Datenbankvariante
 - Services können eigene oder auch gemeinsame Datenbanken haben

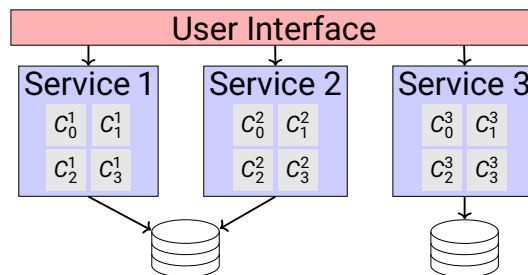


ABBILDUNG 6: Beispiel für Servicelokale Datenbankvariante einer Dienstbasierenden Architektur

6 Designprinzipien (Design Principles)

6.1 Softwarequalität

6.1.1 Faktoren

Die Faktoren guter Software trennen sich in *interne* und *externe Faktoren*:

- Interne Faktoren: Sicht der Entwickler (Code-Qualität). Stellt eine „White Box“ dar.
- Externe Faktoren: Sicht der Nutzer (interne Qualitätsfaktoren sind nicht bekannt). Stellt eine „Black Box“ dar.

Interne Faktoren

- Modularität
- Verständlichkeit
 - Namensgebung (Methoden, Parameter, Variablen, ...)
- Kohäsion
- Prägnanz (keine/wenige Duplikate, klarer (kurzer) Code)
- ...

Externe Faktoren

- Korrektheit
- Verlässlichkeit
- Erweiterbarkeit
- Wiederverwendbarkeit
- Kompatibilität
- Portabilität
- Effizienz
- Nutzbarkeit
- Funktionalität
- Wartbarkeit
- ...

Merkmale von Guter Software:

- Wartbarkeit: Kann an ansprüche des Kunden angepasst werden
- Effizienz: Keine Ressourcenverschwendung
- Usability: Muss für den Nutzer verständlich und benutzbar sein
- Verlässlichkeit (Dependability): Verursacht keinen Wirtschaftlich- oder Physikalischen Schaden falls das System fehlschlägt

6.2 Messen von Softwarequalität

6.2.1 Übersicht Metriken zum Messen von Softwarequalität

Fan In	Anzahl Methoden, welche m aufrufen
Fan Out	Anzahl Methoden, welche m aufruft
Codelänge	Anzahl Zeilen
Zyklomatische Komplexität	Linear unabhängige Pfade durch den Code (Kontrollflussgraph)
Verschachtelungstiefe	Tiefe Verschachtelung von if/else, switch..case, etc. sind schwer zu verstehen
Gewichtete Methodenkomplexität pro Klasse	Gewichtete Summe der Methodenkomplexitäten
Vererbungstiefe	Tiefe Vererbungsbäume sind hochkomplex (Unterklassen)

6.2.2 Zyklomatische Komplexität (Cyclomatic Complexity)

Die Zyklomatische Komplexität C berechnet sich durch $C = E - N + 2P$, wobei E die Anzahl der Kanten, N die Anzahl der Knoten und P die Anzahl der möglichen Zusammenhangskomponenten (in den meisten Fällen 1) des CFGs darstellt.

6.2.3 Kontrollflussgraph (CFG)

Ein Kontrollflussgraph stellt Code syntaxfrei dar, wodurch bessere Analysen möglich sind.

Beispiel

Der in Abbildung 7 gezeigte Code wird in Abbildung 8 als Kontrollflussgraph dargestellt.

```
1 public static int fibonacci(final int num) {
2     if (num <= 0) {
3         throw new IllegalArgumentException();
4     }
5     int current = 1;
6     int previous = 0;
7     for (int i = 0; i < num - 1; i++) {
8         int next = current + previous;
9         previous = current;
10        current = next;
11    }
12    return current;
13 }
```

ABBILDUNG 7: Beispiel: Kontrollflussgraph / Code

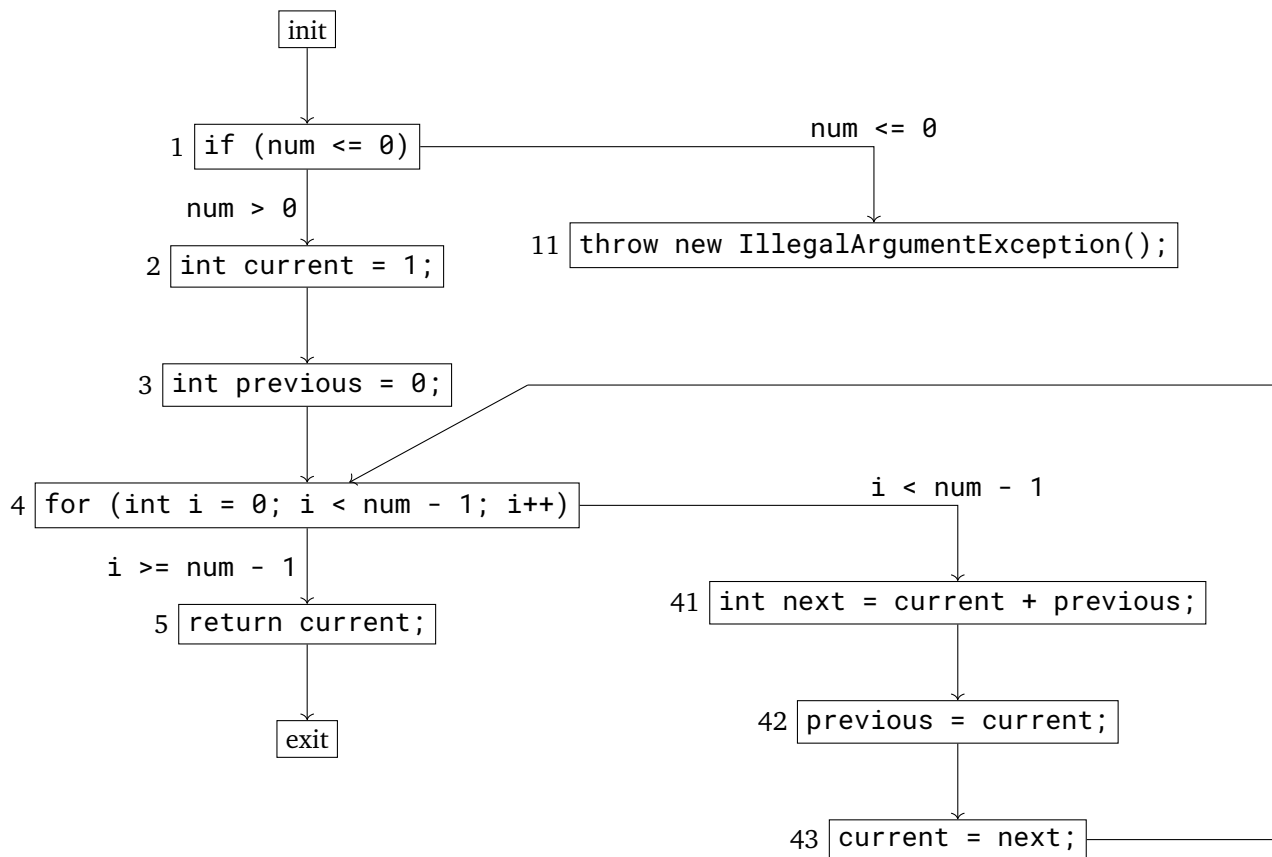


ABBILDUNG 8: Beispiel: Kontrollflussgraph

6.2.4 Heuristiken

Design-Heuristiken helfen dabei, die Frage zu beantworten, ob das Design einer Software gut, schlecht oder irgendwo dazwischen ist.

- Einsichten in OO-Design Verbesserungen
- Sprachunabhängig und erlauben das Einstufen der Integrität einer Software
- Nicht schwer aber schnell: Sollten Warnungen produzieren, welche unter anderem das ignorieren der selbigen erlauben wenn nötig

Beispiel: All Daten in einer Basisklasse sollten privat sein; die Nutzung von nicht-privaten Daten ist untersagt, es sollten Zugriffsmethoden erstellt werden, welche protected sind.

6.2.5 Verknüpfen/Koppeln (Coupling)

Definition – Class Coupling ist ein Richtwert zur Messung der Abhängigkeiten zwischen Klassen und zwischen Paketen:

- Eine Klasse C ist an Klasse D *gekuppelt*, wenn C direkt oder indirekt von d Abhängt
- Eine Klasse, welche auf 2 anderen Klassen basiert, hat eine lockerere Kopplung als eine Klasse, welche auf 8 anderen Klassen basiert.

Kopplung in Java

Die Klasse

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 public class QuitAction implements ActionListener {
5     @Override
6     public void actionPerformed(ActionEvent event) {
7         System.exit(0);
8     }
9 }
```

ist mit den folgenden anderen Klassen gekoppelt: ActionEvent, ActionListener, Override, System, Object

Lose Kopplung vs. Feste Kopplung

- Zu viele Verknüpfungen sind schlecht, denn:
 - Änderungen in verknüpfter Klasse kann zu Dominoeffekt an Änderungen führen
 - Eng verknüpfte Klassen sind isoliert betrachtet nur schwer zu verstehen
 - Eng verknüpfte Klassen lassen sich nur schwer wiederverwenden (da alle Abhängigkeiten mit kopiert werden müssten)
- Zu wenige bzw. gar keine Verknüpfungen sind aber auch unerwünscht, denn:
 - Geht gegen das Prinzip der Objektorientierten Programmierung: „Ein system von Verknüpften Objekten, die per Nachrichten Kommunizieren“
 - Führt zur Bildung von Gottklassen
 - Führt zu hoher Komplexität
- Generische Klassen müssen sehr wenige Verknüpfungen haben
- Viele Verknüpfungen von stabilen Bausteinen (Libraries, z.B. aus der Java-Standardbibliothek) ist kein Problem

Warnung: Die Anforderung an lose Kopplung zur Wiederverwendbarkeit in (mystischen) Zukunftsprojekte birgt die Gefahr von unnötiger Komplexität und hohen Projektkosten!

6.2.6 Zuständigkeiten

Beschreibung

Class Der Name der Klasse/des Akteurs.

Responsibilities Die Zuständigkeiten der Klasse; identifiziert die zu lösenden Probleme.

Collaborations Andere Klassen/Akteure mit denen die Klasse/der Akteur kooperiert um eine Aufgabe zu erfüllen.

Wichtige Konklusionen

Class	<ul style="list-style-type: none">• Der Name sollte deskriptiv und eindeutig sein.
Responsibilities	<ul style="list-style-type: none">• Lange Zuständigkeitsliste \Rightarrow Sollte die Klasse aufgeteilt werden?• Zuständigkeiten sollten zusammenhängen.
Collaborations	<ul style="list-style-type: none">• Viele Kollaboratoren \Rightarrow Sollte die Klasse aufgeteilt werden?• Vermeide kyklische Kollaboration! \Rightarrow Es sollten höhere Abstraktionsebenen eingeführt werden.

6.2.7 Kohäsion(Cohesion)

Definition – Kohäsion ist ein Richtwert zur Messung des Zusammenhangs zwischen Elementen einer Klasse. Alle Operationen und Daten innerhalb einer Klassen sollten „natürlich“ zu dem Konzept gehören, welches von der Klasse modelliert wird.

Arten von Kohäsion (geordnet von sehr schlecht zu sehr gut):

1. **Zufällig** (Coincidental, keine Kohäsion vorhanden): Kein sinnvoller Zusammenhang zwischen den Elementen einer Klasse (bspw. bei Utility-Klassen, **unerwünscht**).
2. **Zeitliche Kohäsion** (Temporal): Alle Elemente einer Klasse werden „zusammen“ ausgeführt.
3. **Sequentielle Kohäsion** (sequential): Das Ergebnis einer Methode wird an die nächste übergeben.
4. **Kommunikative Kohäsion** (Communicational): Alle Funktionen/Methoden einer Klasse lesen/schreiben auf der selben Eingabe/Ausgabe.
5. **Funktionale Kohäsion**: Alle Elemente einer Klasse arbeiten zusammen zur Ausführung einer einzigen, wohldefinierten, Aufgabe. (**Idealfall**)

Klassen mit geringer Kohäsion sind zu vermeiden, da:

- Schwer zu verstehen
- Schwer wiederzuverwenden
- Schwer wartbar (einfach änderbar)
- Symptomatisch für sehr grobkörnige Abstraktion
- Es wurden Aufgaben übernommen, die zu anderen Klassen delegiert werden sollten

Klassen mit hoher Kohäsion können oftmals mit einem einfachen Satz beschrieben werden.

Lack of Cohesion of Methods (LCOM)

Definition – LCOM-Wert Der *Lack of Cohesion of Methods*-Wert (kurz LCOM-Wert) ist ein Richtwert zur Bewertung der Kohäsion einer Klasse.

Sei C eine Klasse, F ihre Instanzvariablen und M ihre Methoden (Konstruktoren ausgenommen).

Sei $G = (M, E)$ ein ungerichteter Graph mit den Knoten $V = M$ und den Kanten

$$E = \{\langle m, n \rangle \in M \times M \mid \exists f \in F : (m \text{ nutzt } f) \wedge (n \text{ nutzt } f)\}$$

dann ist $\text{LCOM}(X)$ definiert als die Anzahl der zusammenhängenden Komponenten des Graphen G .

Ist $n = |M|$ die Anzahl der Methoden, so gilt $\text{LCOM}(X) \in [0, n]$.

Ein hoher LCOM-Wert ist ein Indikator für zu geringe Kohäsion in der Klasse.

6.3 Designprinzipien

6.3.1 Single-Response-Principle (SRP)

Definition – Single-Response-Principle „Eine Klasse sollte nur einen einzigen Grund haben, sich zu verändern“, Also möglichst wenige Abhängigkeiten pro Klasse (ideal wäre nur eine), denn Abhängigkeiten sind der Hauptgrund für Änderungen

6.3.2 Inheritance vs. Delegation

Inheritance (Übernehmen/Erben)	Delegation (Übergeben)
<ul style="list-style-type: none"> • Auch irrelevante Funktionalität wird vererbt • Schwer zu Warten 	<ul style="list-style-type: none"> • Es wird ein Objekt übergeben, dass nur die gewünschte Funktionalität implementiert • Abhängigkeiten unverändert

TABELLE 8: Inheritance vs Delegation

6.4 Kapselung(Encapsulation)

Definition – Interface-Konzept Ein Interface deklariert die Methodensignaturen und öffentlichen Konstanten seiner Subklassen

- Vorteile:
 - Interfaces machen es Möglich, die Funktionalität von der Implementierung zu trennen
 - Hilft dabei, ungerechtfertigte Vermutungen über die Implementierung zu vermeiden
 - Interfaces sind stabiler als Implementierungen
 - Um die Implementierung zu verändern, reicht es den Konstruktor zu verändern
 - Einfacher um Kupplung zu vermeiden

6.4.1 Zugriffsrechte (Field Access)

Angenommen es gäbe keine Zugriffsrechte und alles wäre Public, dann:

- Verletzt das Prinzip der Informationsverbergung (information hiding principle):
 - Keine Unterscheidung zwischen Implementationsspezifischen- und Öffentlichen Daten
 - Implementationsspezifische details werden dem Client enthüllt
- Instabiler Code, wenn die Implementation des Feldes sich ändert
- für Felder die als Argument oder unter einem Alias definiert sind ist es sehr schwer, die Abhängigkeiten zu erkennen

Also besser: Getter und Setter, sodass Zugriffe gezielt kontrolliert werden können Probleme, wenn zu wenig Freigegeben wird:

- Funktionalität muss ggf doppelt implementiert werden
- Verleitet einen dazu, die Abhängigkeiten falsch zu setzen

Warnung: Instanzfelder (vom Konstruktor gesetzt) sollten niemals Public sein

6.5 Probleme

6.5.1 Gottklassen (God Classes)

Eine Klasse kapselt einen Großteil oder alles der (Sub-) Systemlogik. Indikator von:

- Schlecht verteilter Systemintelligenz
- Schlecht OO-Design, welches auf der Idee von zusammenarbeitenden Objekten aufbaut



Lösungsansätze

- Gleichmäßige Verteilung der Systemintelligenz
Oberklassen sollten die Arbeiten so gleich wie möglich verteilen.
- Vermeidung von nichtkommunikativen Klassen (mit geringer Kohäsion)
Klassen mit geringer Kohäsion arbeiten oftmals auf einem eingeschränkten Teil der eigenen Daten und haben großes Potential, Gottklassen zu werden.
- Sorgfältige Deklaration und Nutzung von Zugriffsmethoden
Klassen mit vielen (öffentlichen) Zugriffsmethoden geben viele Daten nach außen und halten somit das Verhalten nicht an einem Punkt.

6.5.2 Class Proliferation

Zu viele Klassen in Relation zu der Größe des Problems. Oftmals ausgelöst durch zu frühes Ermöglichen von (mystischen) zukünftigen Erweiterungen.