

Software Engineering

Zusammenfassung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Author: Ruben Deisenroth
Basierend auf: Merktzettel von Fabian Damken
Stand: 23. Februar 2021

Semester: WiSe 2020/21

Fachbereich: Informatik

Inhaltsverzeichnis

1	Einführung	4
1.1	Software	4
1.1.1	Arten von Software	4
1.1.2	Softwarecharakteristiken	4
1.2	Engineering	5
1.2.1	Characteristics of Engineering Approaches	5
1.2.2	Probleme der Softwareentwicklung	5
2	Requirements Engineering	6
2.1	Anforderungsanalyse	6
2.1.1	Nutzeranforderungen (User requirements)	6
2.1.2	Systemanforderungen (System Requirements)	6
2.1.3	Domänenanforderungen (Domain Requirements)	7
2.1.4	Funktionale Anforderungen (Functional Requirements)	7
2.1.5	Nichtfunktionale Anforderungen (Non-Functional Requirements)	7
2.1.6	RE-Prozess	8
3	Anwendungsfälle (Use Cases)	8
3.1	Use Case Analysis	8
3.1.1	Requirements vs Use Cases	8
3.1.2	Use Case Formats	9
3.1.3	Richtlinien für das Entwickeln von Anwendungsfällen	9
3.2	UML Nutzungszweck Diagramme (UML Use Case Diagrams)	10
4	Domänenmodellierung (Domain Modelling)	11
4.1	Diagramm: Domain Model (UML)	12
4.1.1	Klasse vs Attribut	13
4.1.2	Attribut vs Verbindung	13
4.1.3	UML Zustandsdiagramm (State Machine Diagram)	13
5	Softwarearchitektur (Software Architecture)	14
5.1	Architekturcharakteristiken	15
5.2	Architekturstile	15
5.2.1	Monolithische Architekturstile (Monolithic Architecture Styles)	16
5.2.2	Verteilte Architekturstile (Distributed Architectural Styles)	17
6	Designprinzipien (Design Principles)	18
6.1	Softwarequalität	18
6.1.1	Faktoren	18
6.2	Messen von Softwarequalität	19
6.2.1	Übersicht Metriken zum Messen von Softwarequalität	19

6.2.2	Zyklomatische Komplexität (Cyclomatic Complexity)	19
6.2.3	Kontrollflussgraph (CFG)	19
6.2.4	Heuristiken	20
6.2.5	Verknüpfen/Koppeln (Coupling)	21
6.2.6	Zuständigkeiten	22
6.2.7	Kohäsion(Cohesion)	22
6.3	Designprinzipien	23
6.3.1	Single-Response-Principle (SRP)	23
6.3.2	Inheritance vs. Delegation	23
6.4	Kapselung(Encapsulation)	23
6.4.1	Zugriffsrechte (Field Access)	24
6.5	Probleme	24
6.5.1	Gottklassen (God Classes)	24
6.5.2	Class Proliferation	24
7	Designtechniken	25
7.1	Dokumentation	25
7.1.1	Lesbarkeit	25
7.1.2	Arten von Kommentaren	25
7.2	Überarbeiten (Refactoring)	26
7.2.1	Gründe für das Überarbeiten	26
7.2.2	Methode Extrahieren (Extract Method)	26
7.2.3	Methode Verschieben (Move Method)	27
7.2.4	Klasse Extrahieren (Extract Class)	27
8	Entwurfskonzepte (Design Patterns)	28
8.1	Idiome	29
8.1.1	Template Method	29
8.1.2	Strategy	30
8.1.3	Observer	31
8.1.4	Fabrikmethode (Factory Method)	33
8.1.5	Abstrakte Fabrik (Abstract Factory)	34
9	Verifikation (Verification)	35
9.1	Einführung	35
9.2	Verifikation und Validierung	35
9.2.1	Techniken	35
9.2.2	Codeuntersuchung (Code reviews)	36
9.3	Testen	37
9.3.1	Testtypen	37
9.4	Testabdeckung	38
9.4.1	Strukturell	38
9.4.2	Logisch	38
9.5	Testautomation	39
9.5.1	Automatisierte Test Case Generierung (ATCG)	39
10	Wartung und Weiterentwicklung (Maintanance & Evolution)	40
10.1	Wartung (Maintanance)	40
10.1.1	Auslöser	40
10.1.2	Parallelisierung als eine Wartungsarbeit	40
10.1.3	Softwareevolution (Software Evolution)	41
10.2	Software Variability Engineering	41
10.2.1	Herausforderungen in der Varaibilität	41
10.2.2	Software Product Line Engineering	41
10.2.3	Merkmaldiagramme (Feature Diagrams)	42

11 Verhaltensmodellierung	44
11.1 Diagramme	44
11.1.1 Diagramm: Interaction/Sequence Diagram (UML)	44
11.1.2 Diagramm: State Machine Diagram (UML)	44

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.

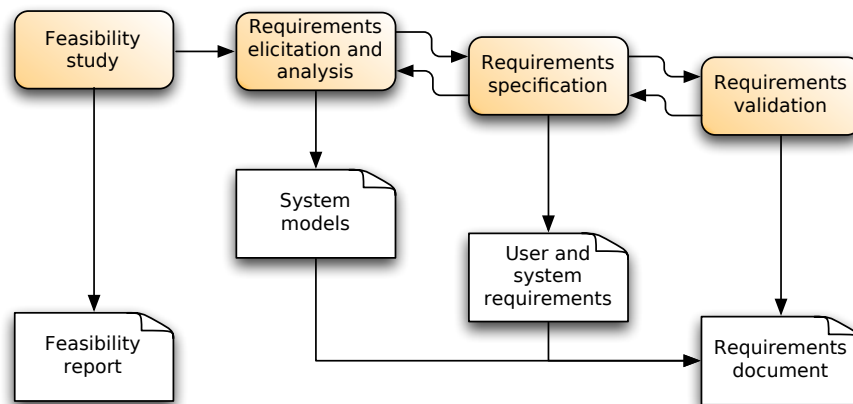
1.2.1 Characteristics of Engineering Approaches

- 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 ein 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.

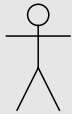
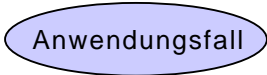
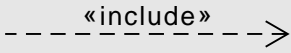
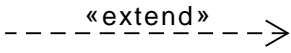
Element	Beschreibung
 Akteur	Stellt einen Akteur innerhalb des Systemes dar.
 Anwendungsfall	Stellt einen Anwendungsfall im System dar
	Erweitert einen Anwendungsfall um eine Funktionalität. Wird der erweiterte Anwendungsfall „ausgeführt“, so wird auch dieser Anwendungsfall „ausgeführt“.
	Erweitert einen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

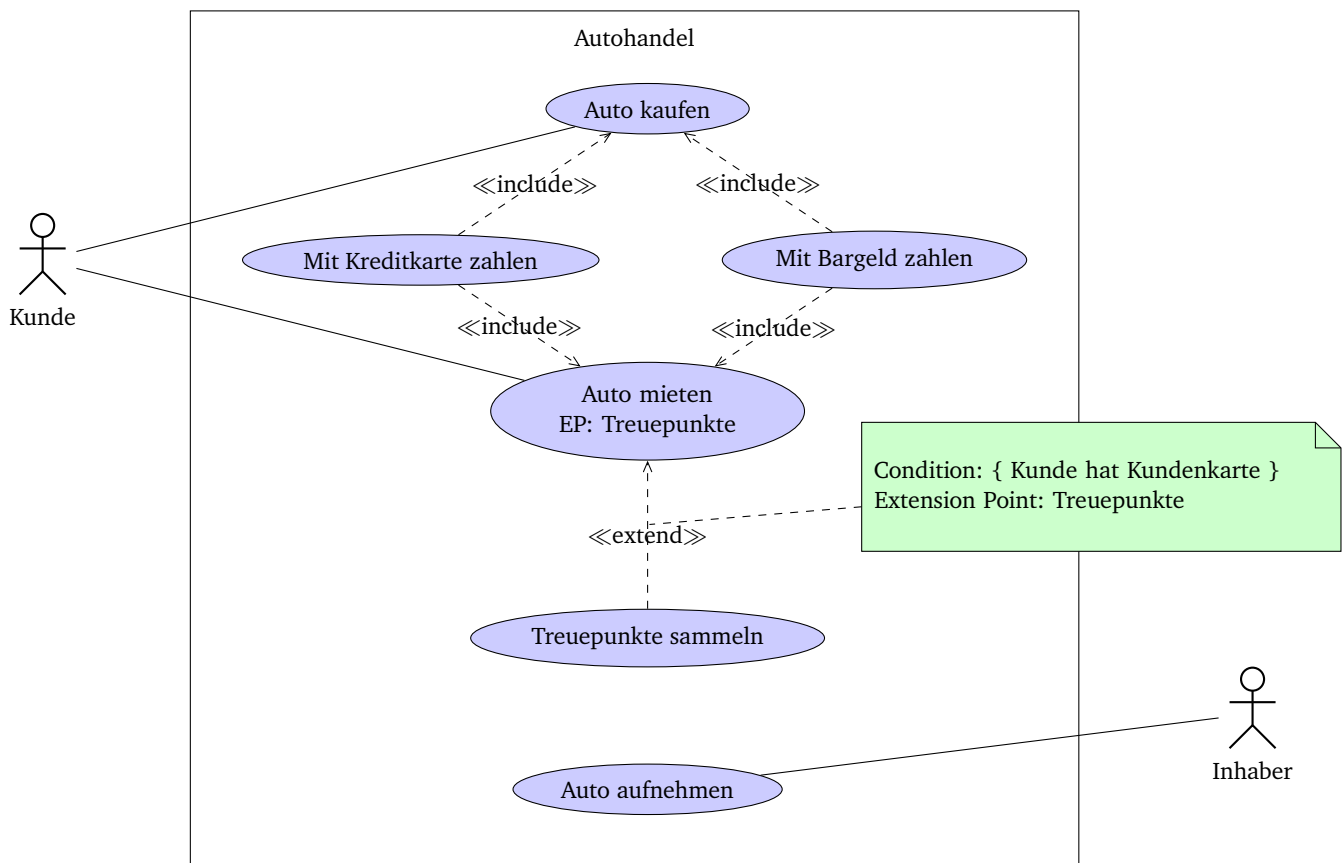


ABBILDUNG 1: Beispiel: Use Case Diagram

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).

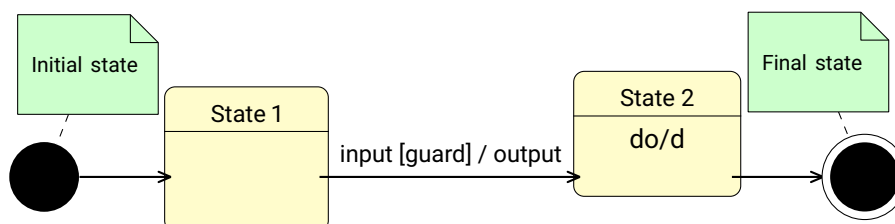


ABBILDUNG 3: Beispiel für UML State Machine Diagram

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

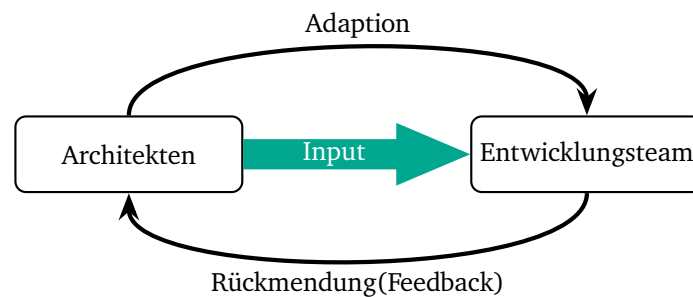


ABBILDUNG 4: Relation Architekten-Entwickler

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, ...

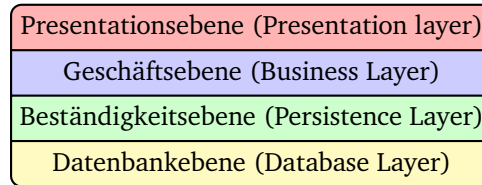
5.2 Architekturstile

- 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
• Einfachheit	• 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

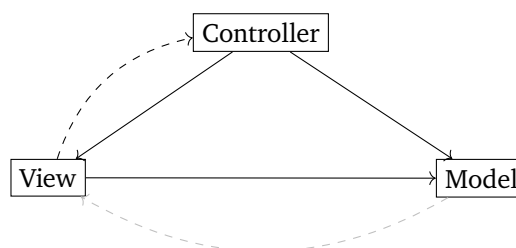


ABBILDUNG 5: Model-View-Controller

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

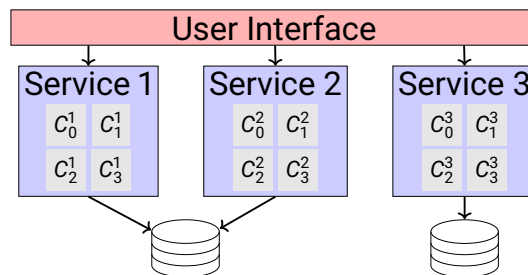


ABBILDUNG 6: Beispiel für Servicelokale Datenbankvariante einer Dienstbasierenden Architektur

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

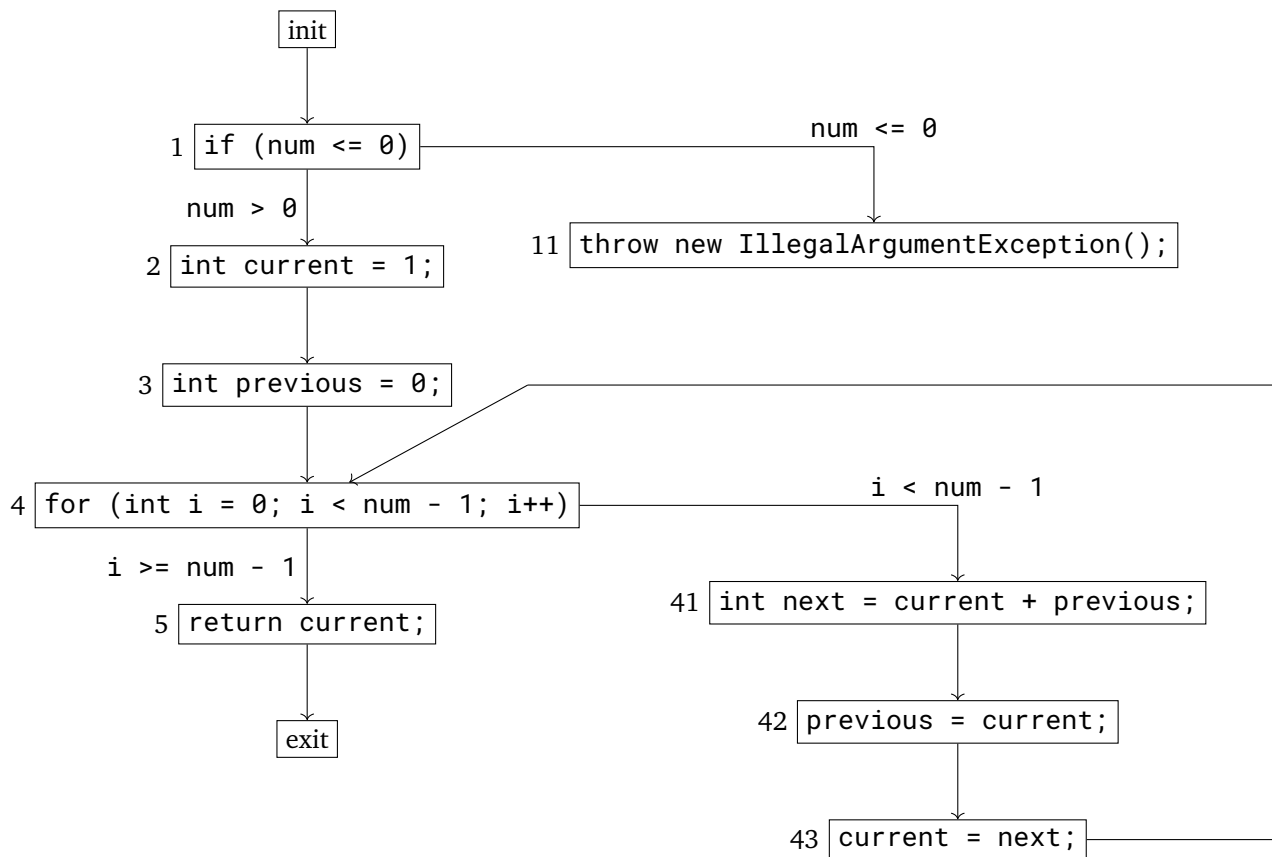


ABBILDUNG 8: Beispiel: Kontrollflussgraph

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 Stablen 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.

7 Designtechniken

7.1 Dokumentation

7.1.1 Lesbarkeit

- Zu einer guten Lesbarkeit gehören:
 - Dokumentation
 - Kommentare im Quellcode
 - der Quellcode selber:
 - * Codestilkonventionen
 - * Einschränkungen der Nutzung bestimmter Sprachkonstrukte (z.B. var in Java)
 - * gute Benennung von Klassen, Methoden und Variablen
 - * Aussagekräftige Kommentare

7.1.2 Arten von Kommentaren

- API-Kommentare:
 - Ausführliche Klassen- und Methodenlevelkommentare
 - Zielgruppe: Andere Entwickler, die den Code als Library/Framework nutzen
 - Muss enthalten:
 - * Einschränkungen von Methodenparametern über den Typ hinaus (z.B. nicht null oder nicht negativ)
 - * Auswirkungen auf den Zustand des Objektes
 - * Wann und welche Exceptions geworfen werden
- Statement-Level-Comments:
 - Kommentare darüber, warum welche Befehle innerhalb einer Methode genutzt wurden
 - Zweck: Implementation beschreiben, Code strukturieren
 - Zielgruppe: Entwickler, die an dem selben Projekt mitarbeiten
 - Sollte nur genutzt werden, wenn auch wirklich nötig
 - Kann ein Indikator für sehr komplexen Code sein ⇒ Überarbeiten (refactor)

7.2 Überarbeiten (Refactoring)

Definition – Refactoring ist eine Disziplinierte Technik, um einen existierenden Codekörper umzustrukturieren, und dabei seine Interne Struktur zu verändern, ohne die Verhaltensweise nach Außen hin zu verändern.

Ziele:

- Hinzufügen von neuen Funktionen vorbereiten
- Design verbessern, z.B. Kohäsion verbessern
- Verständlichkeit verbessern
- Wartbarkeit verbessern

7.2.1 Gründe für das Überarbeiten

- Macht das Hinzufügen neuer Funktionen einfacher
- Weniger Redundanten Code
- Vermeiden von Verschachtelten Bedingungen
- Macht Code einfacher/verständlicher

7.2.2 Methode Extrahieren (Extract Method)

- Wann?:
 - wenn eine Methode zu viele Dinge macht
 - wenn eine Funktionalität mehrfach innerhalb einer Methode oder von mehreren Methoden gebraucht wird.
- Vorgehensweise:
 1. Neue Methode erstellen (Target)
 2. Extrahierten Code zur neuen Methode kopieren
 3. Lokale Variablen identifizieren, die in dem extrahierten Code verwendet wurden und als Parameter übergeben
 4. Code in ursprünglicher Methode mit Methodenaufruf ersetzen.

7.2.3 Methode Verschieben (Move Method)

- Wann:
 - Methode nutzt keine anderen Funktionen ihrer Klasse,
 - Methode überschreibt keine andere Methode oder wird überschrieben **und**
 - Methode berechnet etwas für ein anderes Objekt
 - Vorgehensweise:
 1. Methode in Zielklasse deklarieren
 2. Methode in Zielklasse kopieren und anpassen
 3. Herausfinden, wie die Zielklasse am Besten referenziert werden kann
 4. Quellmethode zu Delegationsmethode umwandeln (Logik findet in Zielklasse statt, aber Aufruf in Quellklasse)
 5. Entscheiden, ob die Quellmethode noch gebraucht wird, oder ob der Aufruf in der Zielklasse sinnvoller ist

7.2.4 Klasse Extrahieren (Extract Class)

- Wann:
 - Klasse hat zwei oder mehr unabhängige Abhängigkeiten
 - Keine Andere Klasse ist für Move Method geeignet
- Vorgehensweise
 1. Neue Klasse erstellen
 2. Neue Klasse als Attribut der alten Klasse
 3. Move Method der Methode
 4. Abhängigkeiten erneut prüfen

8 Entwurfskonzepte (Design Patterns)

Definition – Design Pattern

- beschreibt ein Problem, welches öfter innerhalb der Domäne auftritt
- bietet Bauplan für Lösung des Problems, welcher oft wiederverwendet werden kann, aber niemals den Bauplan selbst sondern eine angepasste Form

- Dokumentiertes Expertenwissen
- Nutzung von generischen Lösungen
- Erhöhung des Abstraktionsgrades
- Ein Muster hat einen Namen
- Die Lösung kann einfach auf andere Varianten des Problems angewandt werden

Aufbau eines Patterns

Pattern Name Kurzer, deskriptiver Name

- Problembeschreibung
- Wann das Pattern angewandt werden soll
 - Fälle, für die das Pattern eine Lösung bietet

- Lösung
- Elemente, die das Design beschreiben
 - Relationen, Abhängigkeiten, Zusammenhänge

- Konsequenzen
- Nebenwirkungen, Einschränkungen, Auswirkungen auf Flexibilität, Erweiterbarkeit und Portabilität

Vorlage Design Pattern

- **Name:** Name des Patterns
 - **Absicht:** Ziele und Gründe warum man das Pattern verwenden sollte
- **Motivation:** Problemsituation beschreiben (Szenario)
 - **Anwendbarkeit:** Bedingungen, wann man es anwenden kann
- **Struktur:** statische Struktur des Patterns (z.B. UML Klassendiagramm)
 - **Participants**(Teilnehmer): Welche Klassen sind Beteiligt
 - **Collaboration:** Zusammenspiel der Teilnehmer
 - **Implementation:** Wie das Pattern angepasst/Implementiert werden kann
- **Konsequenzen:** Nebenwirkungen, Einschränkungen, Auswirkungen auf Flexibilität, Erweiterbarkeit und Portabilität
- **Bekannte Anwendungsfälle:** Beispiele, wann das Pattern verwendet werden soll
 - **Verwandte Patterns:** Referenzen und Vergleiche mit anderen Patterns

TABELLE 9: Vorlage Design Pattern

8.1 Idiome

Idiome sind *keine* Entwurfsmuster, sondern kleine Codeschnipsel:

- Limitiert in der Größe und
- oftmals spezifisch für eine Programmiersprache.

8.1.1 Template Method

Definition – Template Method Die Template-Methode definiert den Algorithmus unter Nutzung von abstrakten (und konkreten) Operationen.

Kurzfassung

Ziel: Implementierung eines Algorithmus, welcher erlaubt, ihn auf mehrere spezifische Probleme anzuwenden.

Idee: Es wird ein Algorithmus implementiert, welcher konkrete Aktionen an abstrakte Methoden und damit Unterklassen weiterreicht.

Konsequenzen:

- Aufteilung von variablen und statischen Teilen
- Verhinderung von Codeduplikation in Unterklassen
- Kontrolle über Erweiterungen von Unterklassen

Generisches Klassendiagramm

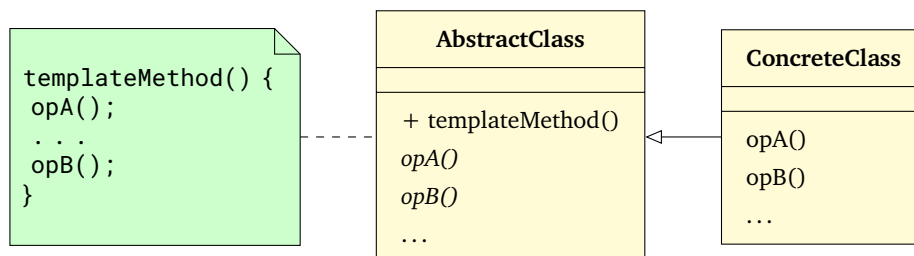


ABBILDUNG 9: UML: Template Method Pattern

Varianten/Erweiterungen

Statt abstrakten Operationen, welche implementiert werden *müssen*, können Hooks verwendet werden, welche implementiert werden *können*.

8.1.2 Strategy

Kurzfassung

Motivation:

- Viele verwandte Klassen unterscheiden sich ausschließlich in ihrem Verhalten statt unterschiedlich verwandte Abstraktionen zu implementieren

Ziel:

- Erlaubt das konfigurieren einer Klasse mit einem von vielen Verhaltensvarianten
- Implementierung unterschiedlicher Algorithmusvarianten können in der Klassenhierarchie verbaut werden

Idee: Definition einer Familie von Algorithmen, Kapselung von jedem und herstellen einer Austauschbarkeit.

Konsequenzen:

- Nutzer muss sich im klaren darüber sein, wie sich die Implementierungen unterscheiden und sich für eine entscheiden
- Nutzer sind Implementierungsfehlern ausgesetzt
- Strategy sollte nur genutzt werden, wenn das konkrete Verhalten relevant ist für den Nutzer

Generisches Klassendiagramm

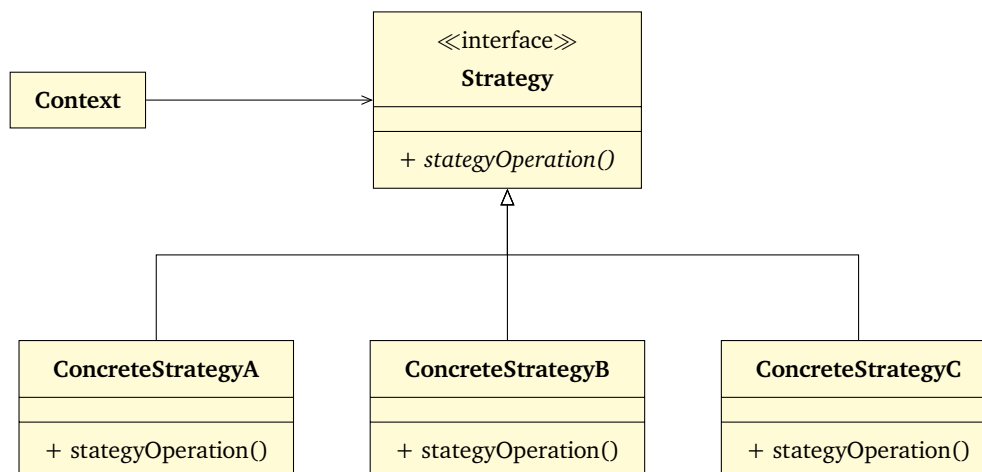


ABBILDUNG 10: UML: Strategy Pattern

Beschreibung

Der Context (Nutzer) erstellt Instanzen von konkreten Strategien, welche den Algorithmus in einem Interface definieren.

Varianten/Erweiterungen

- Optionale Strategy-Objekt
 - Context prüft, ob Strategy-Objekt gesetzt wurde und nutzt es entsprechend
 - Vorteil: Nutzer sind nur dem Strategy-Objekt ausgesetzt, wenn das Standardverhalten nicht genutzt werden soll

8.1.3 Observer

Kurzfassung

Motivation: OOP (Objektorientierte Programmierung) vereinfacht die Implementierung einzelner Objekte, die Verdrahtung dieser kann allerdings schwer sein, sofern man die Objekte nicht hart koppeln möchte.

Ziel: Entkopplung des Datenmodells (Subjekt) von den Stellen, welche an Änderungen des Zustands interessiert sind.
Voraussetzungen:

- Das Subjekt sollte nichts über die Observer wissen.
- Identität und Anzahl der Observer ist nicht vorher bekannt.
- Neue Observer sollen dem System später hinzugefügt werden können.
- Polling soll vermieden werden (da inperformant).

Idee: Erstellung von Observern (generalisiert mittels eines Interfaces), welche einem Subjekt hinzugefügt werden können und aufgerufen werden können.

Generisches Klassendiagramm

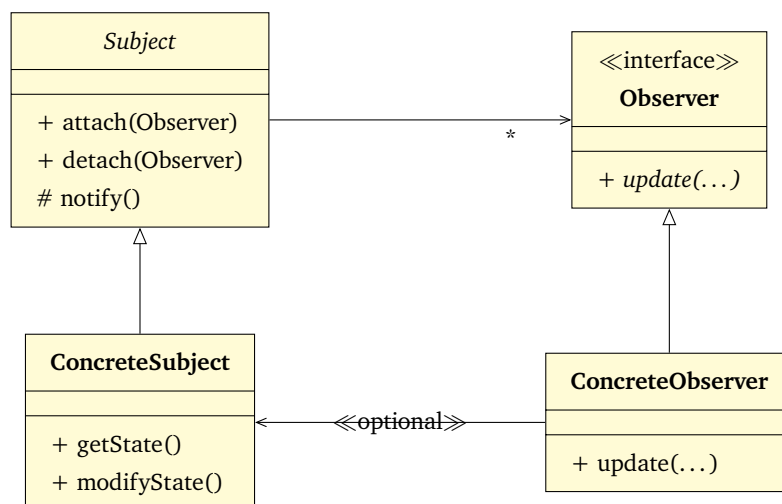


ABBILDUNG 11: UML: Observer Pattern

Beschreibung

Subject Abstrakte Klasse, bietet Methoden zur Implementierung des Musters an.

Observer Interface zum Empfangen von Signalen eines Subjekts.

ConcreteSubject Das konkrete Subjekt, sendet Benachrichtigungen an die Observer

ConcreteObserver Ein konkreter Observer (implementiert observer Interface), registriert sich beim Subjekt, empfängt Nachrichten von dem Subjekt

Konsequenzen:

- Vorteile
 - Abstrakte Kopplung zwischen Subjekt und Observer
 - Unterstützung von Broadcast-Kommunikation
- Nachteile
 - Risiko von Update-Kaskaden zwischen Subjekt, Observer und dessen Observern
 - Updates werden an alle gesendet, sind aber nur für einige relevant
 - Fehlende Details über die Änderungen (Observer muss dies selbst herausfinden)
 - Generelles Interface für Observer schränkt die Parameter stark ein

Varianten/Erweiterungen

- Pull Mode:
 - Subjekt wird nach Änderung gefragt
 - Führt oft dazu, dass nicht nur die Änderung sondern mehr Daten abgerufen werden müssen
- Push Mode:
 - Der update() Methode wird die Änderung als Parameter übergeben
 - Nur der Bereich der auch wirklich geändert werden muss wird geändert ⇒ bessere Antwortzeiten
 - Erfordert genaues Wissen dessen, was geupdated werden muss, also Fehleranfälliger

8.1.4 Fabrikmethode (Factory Method)

Definition – Factory Method Deklariert ein Interface für Objekterstellung, aber lässt die Subklassen selbst entscheiden, welche Klasse instanziiert werden soll (z.B. `public abstract Document createDocument()` und Klasse kann z.B. Textdokument oder ander Ableitung dafür nutzen)

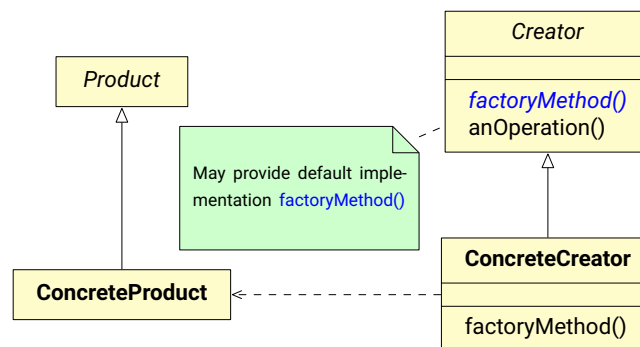


ABBILDUNG 12: Pattern Structure Factory Method

Product Interface für Objekte, die von der factory Methode erstellt werden

Konkretes Produkt Implementiert das Produkt-Interface

Generator (Creator)

- Deklariert die Factory-Methode die ein Objekt des Types **Product** erstellt
- Generator kann default Implementierung der Factory-Methode enthalten, die ein Konkretes Produkt erstellt

Konkreter Generator Überschreibt die Factory-Methode, sodass diese ein anderes Konkretes Produkt erstellt

Konsequenzen

- Der Klient-Code kennt nur das Produkt-Interface
- Stellt einen Hook für Superklassen bereit

Varianten

- Generator als Abstrakte Klasse
- Generator als Konkrete Klasse, mit einer Sinnvollen default-Implementierung

8.1.5 Abstrakte Fabrik (Abstract Factory)

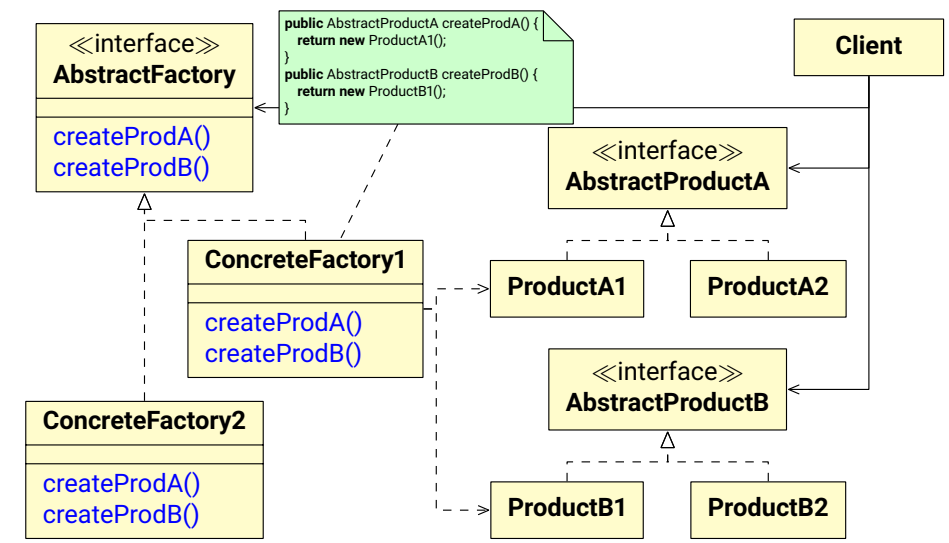


ABBILDUNG 13: Abstrakte Fabrik- Pattern structure

Abstrakte Fabrik Stellt Interface für die Produkterstellung einer Familie bereit

Konkrete Fabrik Implementiert Operationen um Konkrete Produkte zu erstellen

Abstraktes Produkt Deklariert ein Interface um die Konkreten Produkte zu erstellen

Konkretes Produkt Stellt implementation für das Produkt, welche von der Konkoreten Fabrik erstellt wurde bereit

Client Erstellt Produkt indem die Konkrete Fabrik genutzt wird, dabei wird das Abstrkte Produkt interface implementiert

- **Vorteile:**

- Code muss nicht über allen Konkreten Produkte bescheid wissen
- Eine Zeile Code zu verändern reicht aus, um andere Produkte zu unterstützen
- Kann mehrere Produkte unterstützen
- Erzwingt die Erstellung von Konsistenten Produktfamilien

- **Nachteile:**

- Code muss einer neuen Konvention zur Erstellung von Produktfamilien folgen (Statt den Standartkonstruktor zu nutzen)

9 Verifikation (Verification)

9.1 Einführung

9.2 Verifikation und Validierung

Definition – Verifikation Wird das System korrekt erstellt?

Definition – Validierung Wird das richtige System erstellt?

9.2.1 Techniken

Statische Techniken

Statische Techniken erfordern nicht, dass das Programm ausgeführt wird.

Software Reviews Händische/Manuelle Überprüfung

Automatisierte Softwareanalyse, bspw. Typprüfer
Statische Analyse

Formale Verifikation Formaler Beweis, dass ein Programm eine bestimmte Eigenschaft erfüllt

Siehe 6.2.1.

Dynamische Techniken

Dynamische Techniken erfordern, dass das Programm ausgeführt wird.

Testen Führt das Programm aus und Testet es auf bestimmte Eigenschaften (Verhalten)

Laufzeitüberprüfung Analysetools, welche Programme auf Einhaltung bestimmter Einschränkungen (bspw. Speichereinschränkungen) testen

9.2.2 Codeuntersuchung (Code reviews)

Definition – Code Review die Codeuntersuchung ist ein strukturierter Inspektionsprozess einer Software, der meist im Team ausgeführt wird, mit dem Ziel mögliche Fehler im Code zu finden.

Das Ziel der Codeuntersuchung, ist

- Programmfehler,
- Standardfehler und
- Designfehler

zu finden.

Dies wird üblicherweise an (externe) Teams ausgearbeitet, welche den Code systematisch analysieren.

Eine mögliche Checkliste ist bspw.:

Datenfehler Werden Variablen initialisiert, bevor sie genutzt werden? Gibt es mögliche Arra-Out-Of-Bounds Fehler? Werden deklarierte Variablen genutzt?

Kontrollflussfehler Sind die Bedingungen korrekt? Gibt es toten Code? Terminieren alle Schleifen? Sind switch..case Ausdrücke vollständig?

I/O Fehler Werden alle Eingabeparameter genutzt? Können unerwartete Eingaben zu einem Absturz führen?

Schnittstellenfehler Korrekte Anzahl/Typen der Parameter?

- Vorteile:
 - Studien zeigen, dass es funktioniert und Kosten sparen kann
 - Helfen dabei, Wissen über den Code bei den Teammitgliedern zu verteilen
 - Fehler können gefunden werden, bevor sie in Tests auftauchen
 - Können Codequalität erhöhen
 - Keine Codeausführung nötig
- Nachteile
 - Erfordert „Erwachsene“ Herangehensweise: Teammitglieder können sich kritisiert fühlen
 - Unter Zeitdruck fühlen sie sich unproduktiv an
 - Funktionieren nur gut wenn korrekt geführt

9.3 Testen

„Programmtesten kann zwar die Existenz von Fehlern aufzeigen, aber niemals ihre Abwesenheit beweisen!“
— E. W. Dijkstra, EWD 249

Definition – Testplan Ein *Testplan* Testplan ist zur Ausführung durch Menschen gedacht. Er dokumentiert die Schritte des Tests und das jeweilige erwartete Ergebnis. In der Testausführung kann das eigentliche Ergebnis dann mit dem erwarteten verglichen werden.

Definition – Testfall (Test case) besteht aus

- Startzustand (Pretest state)
- Testlogik
- Erwarteten Ergebnissen

Definition – Test Suite Eine Sammlung an Testfällen heißt Test Suite (Testsammlung)

Definition – Test Run Ausführung einer Test Suite auf der IUT, wenn der Test durch geht kriegt er den „Verdict Pass“ (Urteilspass), sonst bezeichnet man ihn als gescheitert (failed).

Definition – Test Driver (Treiber) Klasse oder Programm, welches den Test auf die IUT anwendet

Definition – Test Harness (Gerüst) System von Testtreibern und anderen Tools, um die Testausführung zu unterstützen

9.3.1 Testtypen

Unit Tests

Sehr kleine, automatisierte, Tests, welche eine Funktionalität testen. In typischen Softwareprojekten finden sich ≤ 1000 Unit Tests.

Integrations-Tests

Testen eines kompletten (Unter-) Systems, um die Zusammenarbeit der Komponenten zu Testen.

Systemtests

Testen einer komplett integrierten Applikation (Funktion, Performanz, Stresstest, ...).

9.4 Testabdeckung

Definition – Bedingung Eine *Bedingung* ist ein boolescher Ausdruck.

Definition – Entscheidung Eine *Entscheidung* ist eine Zusammensetzung von Bedingungen, welche bspw. den Test eines `if`-Ausdruckes darstellt.

Definition – Basisblock (Basic block) Ein Basisblock B ist eine Maximale Sequenz von Instruktionen ohne Sprünge.

9.4.1 Strukturell

Strukturelle Testabdeckung basiert auf dem Kontrollflussgraphen (CFG) eines Programms

- Statement Coverage (SC): Alle Ausdrücke wurden mindestens einmal ausgeführt.
- Basic Block Coverage (BBC): Alle Basisblöcke wurden mindestens einmal ausgeführt.
- Branch Coverage (BC): Jede Seite von jedem Knoten wurde mindestens einmal ausgeführt (d.h. jede Kante eines Kontrollflussgraphen).
- Path Coverage (PC): Alle Pfade wurden mindestens einmal ausgeführt (siehe CFG).

9.4.2 Logisch

- Condition Coverage (CC): Jede Bedingung wurde mindestens einmal zu wahr und falsch ausgewertet.
- Decision Coverage (DC): Jede Entscheidung wurde mindestens einmal zu wahr falsch ausgewertet.
- Modified Condition Decision Coverage (MCDC): Kombiniert Aspekte der Condition Coverage, Decision Coverage und Unabhängigkeitstests.

Ein Test für eine Bedingung c in Entscheidung d (Duplikate von c werden nicht gezählt) erfüllt MCDC gdw.

- er d mindestens zweimal ausgewertet,
- davon c einmal zu wahr und einmal zu falsch ausgewertet,
- d in beiden Fällen unterschiedlich ausgewertet wird und
- die anderen Bedingungen in d in beiden identisch oder in mindestens einem nicht ausgewertet werden.

Für 100%-ige MCDC-Abdeckung muss dies für jede Bedingung in dem Programm gelten.

- Multiple-Condition Coverage (MCC): Alle möglichen Kombinationen innerhalb einer Entscheidung wurden mindestens einmal ausgeführt.

9.5 Testautomation

Ein Testautomationssystem

- startet die „implementation under test“ (IUT),
- setzt die Umgebung auf,
- bringt das System in den erwarteten Ausgangszustand,
- wendet die Testdaten an und
- evaluiert die Ergebnisse und den Zustand des Systems.

9.5.1 Automatisierte Test Case Generierung (ATCG)

White Box

- Syntaktischer Ansatz: Suchen nach Bedingungen, Evaluierung ermöglicht logic-Based Coverage
- Unerreichbarer Code kann gefunden werden
- Symbolische Ausführung: versuchen, CFG mit verschiedensten Werten durchzulaufen, Evaluierung ermöglicht Structural Coverage

Black Box

- Analyse der Ein-und Ausgabedaten der IUT
- Probleme: Library Calls, Unnötige Testfälle, rekursion, ...

Test Coverage Recording

1. IUT Initialisieren
2. Test suite ausführen und Informationen während des Testlaufes sammeln
3. Test Coverage analysieren und darstellen

Test Oracle Synthesis

- Menschliches Orakel: Zeitaufwändig und Fehleranfällig
- Testen durch Code: Braucht Expertenwissen, schwer zu warten

Static Checking Basiert auf CFG, hilft bei:

- Runtime Exceptions, Informationsfluss
- Vollautomatisiert
- Gefahr: False Positives
- Zeitaufwand gering

Dynamic Checking

- Runtime Monitoring

Formal Checking

- Design-By-Contract (wie Racket Verträge mit preconditions und so)

10 Wartung und Weiterentwicklung (Maintanance & Evolution)

10.1 Wartung (Maintanance)

- Software altert nicht
- **Aber:** Die Umgebung der Software ändert sich schneller und öfter als in jeder anderen Ingenieursdisziplin

10.1.1 Auslöser

1. Bugs beheben
2. Abhängigkeit verändert sich
3. Neue Funktion wird benötigt
4. Hardware hat sich weiterentwickelt
5. Softwarearchitektur hat sich weiterentwickelt
6. Software stack (Libraries und so) hat sich weiterentwickelt

10.1.2 Parallelisierung als eine Wartungsarbeit

Beobachtung: Mehrkernprozessoren und -Systeme werden immer weiter verbreitet
⇒ Große Geschwindigkeitsverbesserungen möglich, aber die meiste Existierende Software ist noch sequenziell geschrieben.

Folge: Parallelisierung von Software wird zu einem wichtigen Teil der Wartung (kann als komplexes Refactoring betrachtete werden)

- Korrekte Parallelisierung erfordert ein gutes Wissen von:
 - Dem Problem Space
 - Parallelisierbaren Algorithmen und Datenstrukturen (AuD Vibes und so)
 - Die zugrundeliegende Hardwarearchitektur muss Parallelisierung unterstützen

Das Pipeline Parallelisierungsmuster (The Pipeline Parallelization Pattern)

1. Daten des Vorherigen Verarbeitungsschrittes (stage) empfangen (Beenden wenn alle Daten bereits abgearbeitet)
 2. Daten verarbeiten
 3. Daten an nächsten Verarbeitungsschritt senden
- Sorgt für:
 - Stabile, Syncrone Verarbeitungsreihenfolge
 - Spezifische Verarbeitungsschritte können an dafür optimierte Hardware abgegeben werden
 - Erlaubt einfaches Modifizieren, Hintufügen und Neuordnen der einzelnen Verarbeitungsschritten
 - Kann aber nicht für Input faults Kompensieren

10.1.3 Softwareevolution (Software Evolution)

Definition – Softwareevolution Die Reihe von Änderungen, neuen Versionen und Anpassung die während der Wartung zwischen entwicklungsbeginn und der letzten Version entstehen.

Probleme:

- Kann bereits existierende Funktionalität zerstören (Verschlimmbesserung)
 - Lösungsansatz: viel testen und ggf Betaversion an erfahrene Benutzer verteilen
- Kann Trennung zwischen Spezifikation, Dokumentation und Implementierung bewirken
 - Lösungsansatz: Neue Funktionen rufen einen „Full development cycle“ hervor.

10.2 Software Variability Engineering

Definition – Softwarevariabilität (Software Variability) Die Nötigkeit, eine Große Anzahl an eng verwandten Produktvarianten mit unterschiedlichen Abhängigkeiten und Funktionen zu warten.

10.2.1 Herausforderungen in der Variabilität

- Möglicherweise sehr viele Varianten (Millionen, ...)
- Abhängigkeitskonflikte sehr wahrscheinlich

10.2.2 Software Product Line Engineering

Definition – Software Product Line (Auch Product Family) Eine Sammlung von Softwaresystemen, die gemeinsame Bausteine und Produktionsmuster teilen.

- Typische Gemeinsamkeiten:
 - Codebasis der Kernfunktionalität
 - Architektur
 - Implementationssprache(n)

Ziel: Nicht jede Variante einzeln warten zu müssen

Definition – Merkmal (Software Feature) Ein Alleinstellungsmerkmal der Software (e.B. leistung, Portabilität oder Funktionalität)

Definition – Produkt Eine Ansammlung von Merkmalen und Paramterinstanzierungen, die ausreichen um einen Ausführbaren Code mit der SPL zu produzieren.

Definition – Variante (Variant) Ausführbarer Code, der ein Produkt einer SPL implementiert.

SPLE-Prinzipien

1. Entwurf des „Feature space“ ist getrennt vom Softwareentwurf und wird **vorher** ausgeführt.
2. Ein Merkmal sollte nie mehr als einmal implementiert werden
3. Es muss möglich sein, den Code der ein bestimmtes Merkmal implementiert zu lokalisieren.

10.2.3 Merkmaldiagramme (Feature Diagrams)

Aufbau

- Oben ist das ‚root Feature‘
- Kein Subfeature ohne Parent möglich
- Standartmodus: Oder (Ein oder mehrere Features auswählbar)

Probleme

- Code mit vielen Macros schwer zu verstehen
- Retundanz nur schwer zu vermeiden (Code wird von mehreren Features benutzt)
- Analyse, Testen und Typchecking schwer auf Familienebene





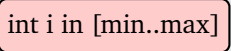
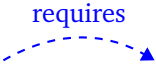
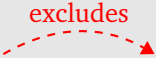
Symbol	Bedeutung
	Oder (Or)
	Exklusives Oder (Xor)
	Notwendiges Feature
	Optionales Feature
	Anzahl eines Features festlegen
	Legt fest, dass ein Feature ein anderes Feature benötigt um zu funktionieren
	Legt fest, dass ein Feature ein anderes Feature ausschließt

TABELLE 10: Feature Diagram Symbole

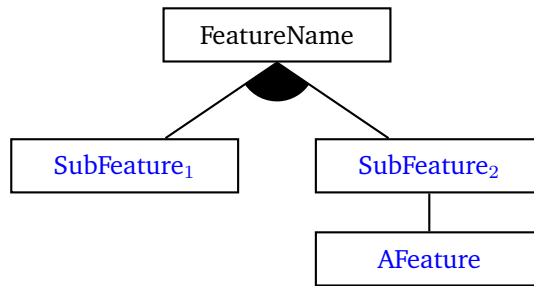


ABBILDUNG 14: Basisstruktur Feature Diagram

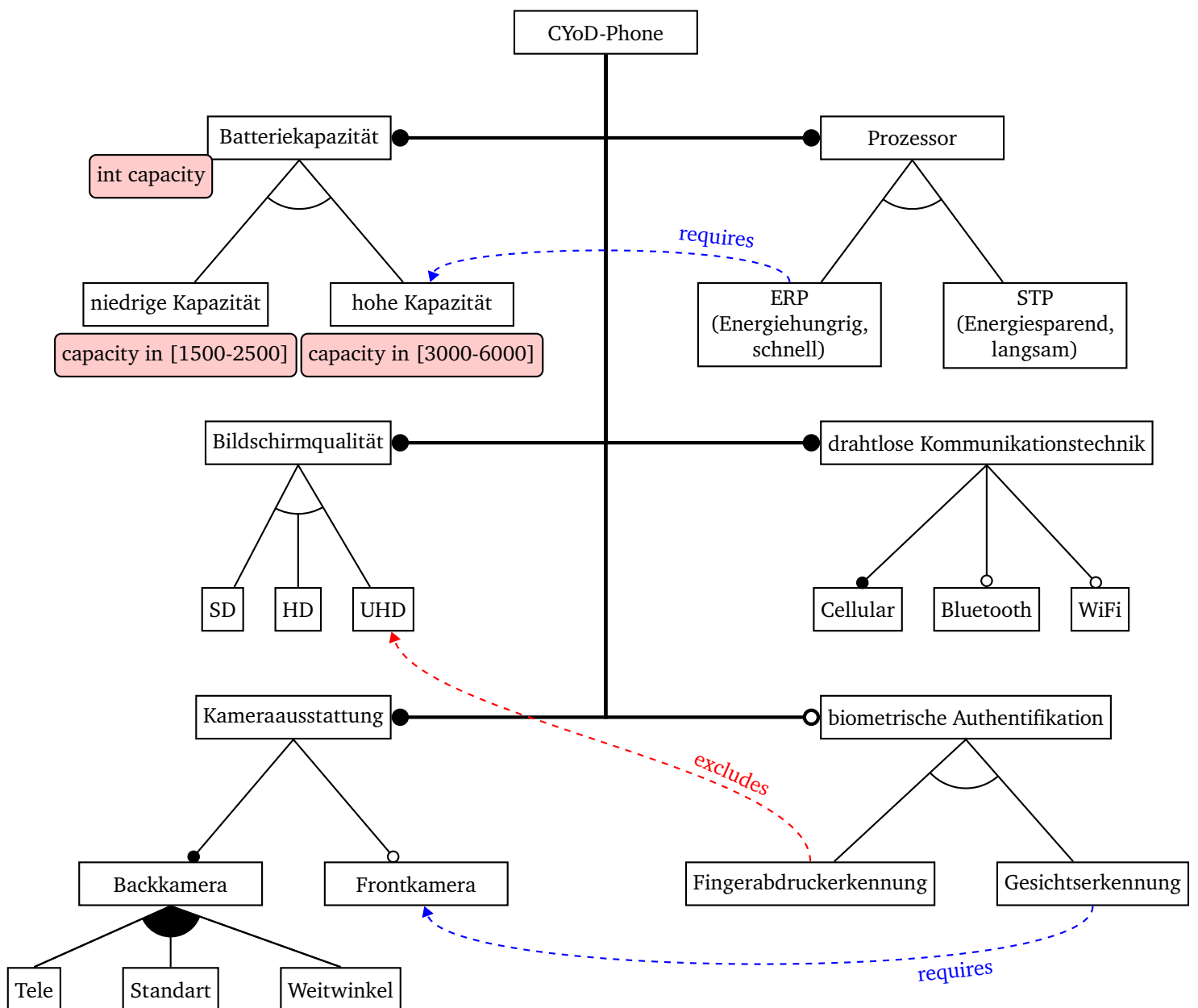


ABBILDUNG 15: Beispielhaftes Feature Diagram

11 Verhaltensmodellierung

11.1 Diagramme

11.1.1 Diagramm: Interaction/Sequence Diagram (UML)

UML Specification Version 2.5.1, Chapter 17

Beschreibung

- Sequenzendiagramme beschreiben die Interaktionen (Nachrichten) zwischen Objekten in einem konkretem Szenario.
- Nicht geeignet, um einen gesamten Algorithmus zu modellieren!

Im Diagramm 16 (Codebasis 17) ist ein Beispiel für ein Sequenzendiagramm gegeben. In diesem werden die einzelnen Komponenten erläutert

Beispiel

Das Verhalten der Methode `run(. . .)` in dem in 16 gezeigtem Code wird in 17 visualisiert.

11.1.2 Diagramm: State Machine Diagram (UML)

UML Specification Version 2.5.1, Chapter 14

Beschreibung

State Machine Diagramme nutzen vereinfachte endliche Automaten zur Darstellung von:

- Eventgetriebenem Verhalten des Systems (Verhalten)
- Interaktionssequenzen (Protokoll)

Im Diagramm ?? ist ein Beispiel für ein State Machine Diagramm gegeben. Im folgenden werden die einzelnen Komponenten erläutert.

```

1 public class Query {
2     public List<Student> run(List<Student> students, ISelector sel) {
3         List<Student> result = createEmptyResult();
4         for (Student s : students) {
5             boolean accepted = sel.accept(s);
6             if (accepted) {
7                 result.add(s);
8             } else {
9                 result.remove(s);
10            }
11        }
12        return result;
13    }
14 }

```

```

1 public interface ISelector {
2     boolean accept(Student s);
3 }

```

```

1 public class SemesterSelector implements ISelector {
2     private int semester;
3
4     public SemesterSelector(int semester) {
5         this.semester = semester;
6     }
7
8     @Override
9     public boolean accept(Student s) {
10        int current = s.getSemester();
11        boolean accepted = (semester == current);
12        return accepted;
13    }
14 }

```

```

1 public class Student {
2     private int semester;
3
4     public Student(int semester) {
5         this.semester = semester;
6     }
7
8     public int getSemester() {
9         return semester;
10    }
11 }

```

ABBILDUNG 16: Beispiel: UML Sequenzen Diagramm / Code

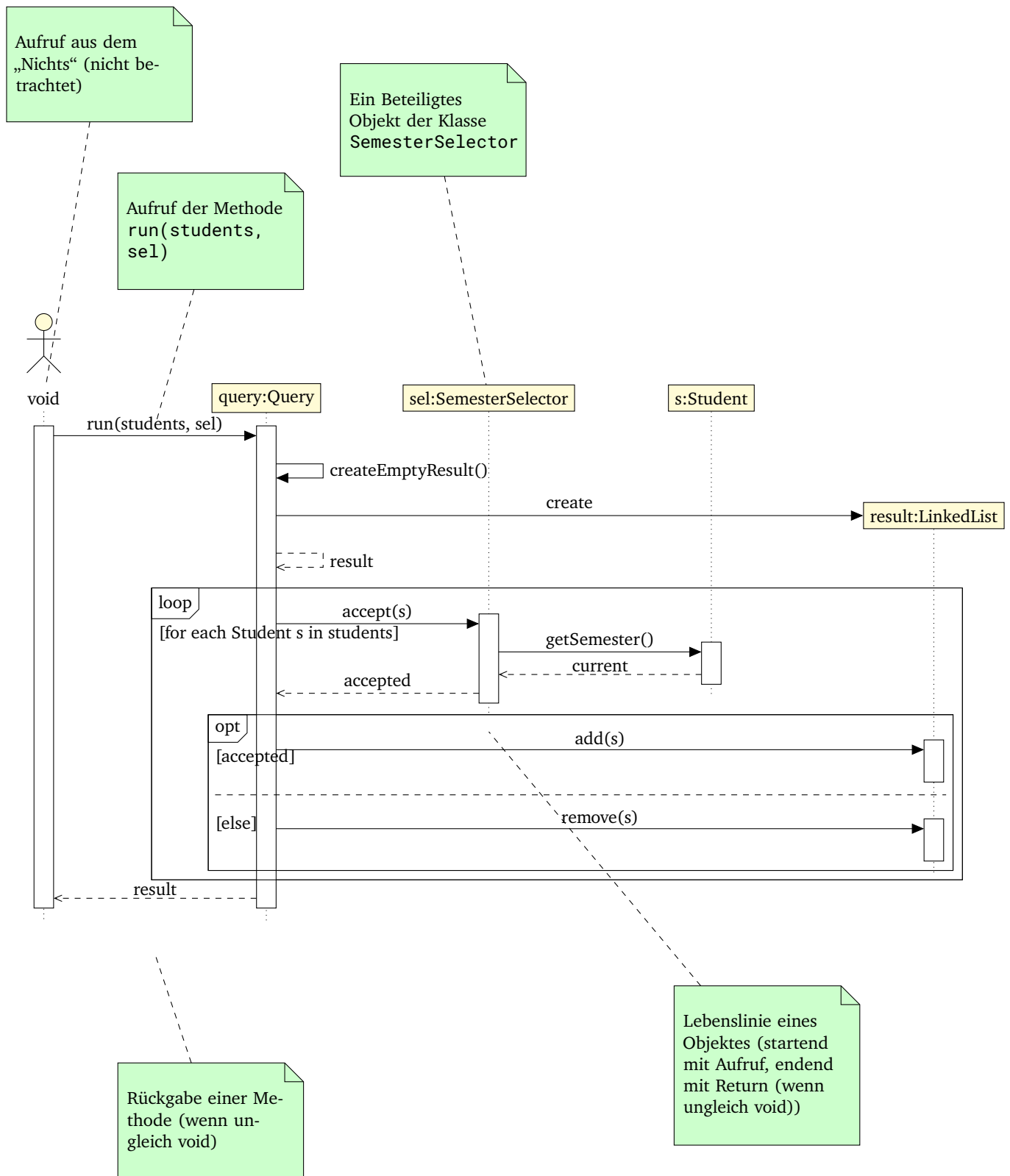


ABBILDUNG 17: Beispiel: UML Sequenzen Diagramm