

Software Engineering

Zusammenfassung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Author: Ruben Deisenroth
Stand: 20. 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	Softwarequalität	13
5.1	Faktoren	13
5.1.1	Interne Faktoren	13
5.1.2	Externe Faktoren	13
5.2	Verifikation und Validierung	13
5.2.1	Techniken	14
5.2.2	Codeuntersuchung	14
5.3	Metriken	14
5.3.1	Kontrollflussgraph (CFG)	15
5.3.2	Zyklomatische Komplexität	15
5.4	Testen	15
5.4.1	Testplan	15

5.4.2	Testtypen	15
5.4.3	Testautomation	16
5.4.4	Testabdeckung	16
6	Verhaltensmodellierung	18
6.1	Diagramme	18
6.1.1	Diagramm: Interaction/Sequence Diagram (UML)	18
6.1.2	Diagramm: State Machine Diagram (UML)	18
7	Software design	21
7.1	Heuristiken	21
7.1.1	Zuständigkeiten	21
7.1.2	Kopplung	21
7.1.3	Kohäsion	22
7.2	Prinzipien	23
7.2.1	Single-Response-Principle (SRP)	23
7.3	Probleme	23
7.3.1	God Class	23
7.3.2	Class Proliferation	24
8	Entwurfskonzepte	25
8.1	Idiome	25
8.2	Entwurfsmuster	25
8.2.1	Template Method	25
8.2.2	Strategy	26
8.2.3	Observer	27
8.3	Architekturmuster	28
8.3.1	Model-View-Controller (MVC)	28

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.

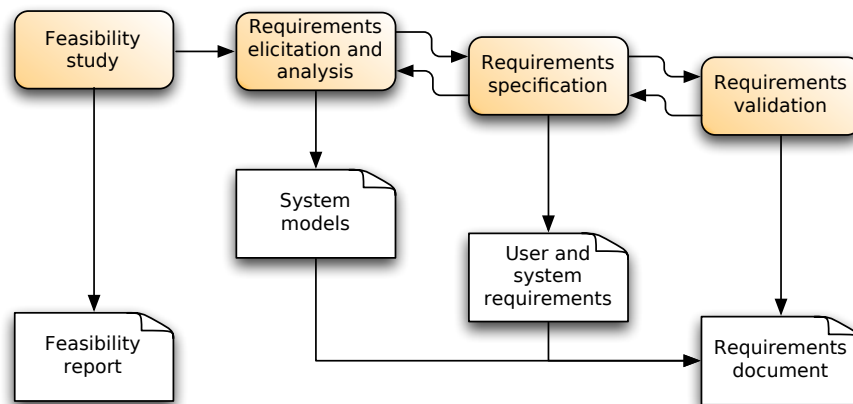
1.2.1 Characteristics of Engineering Approaches

- 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.

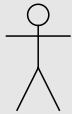
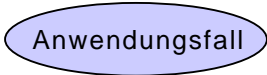
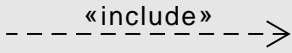
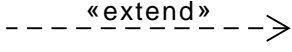
Element	Beschreibung
 Akteur	Stellt einen Akteur innerhalb des Systemes dar.
 Anwendungsfall	Stellt einen Anwendungsfall im System dar
	Erweitert einen Anwendungsfall um eine Funktionalität. Wird der erweiterte Anwendungsfall „ausgeführt“, so wird auch dieser Anwendungsfall „ausgeführt“.
	Erweitert einen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

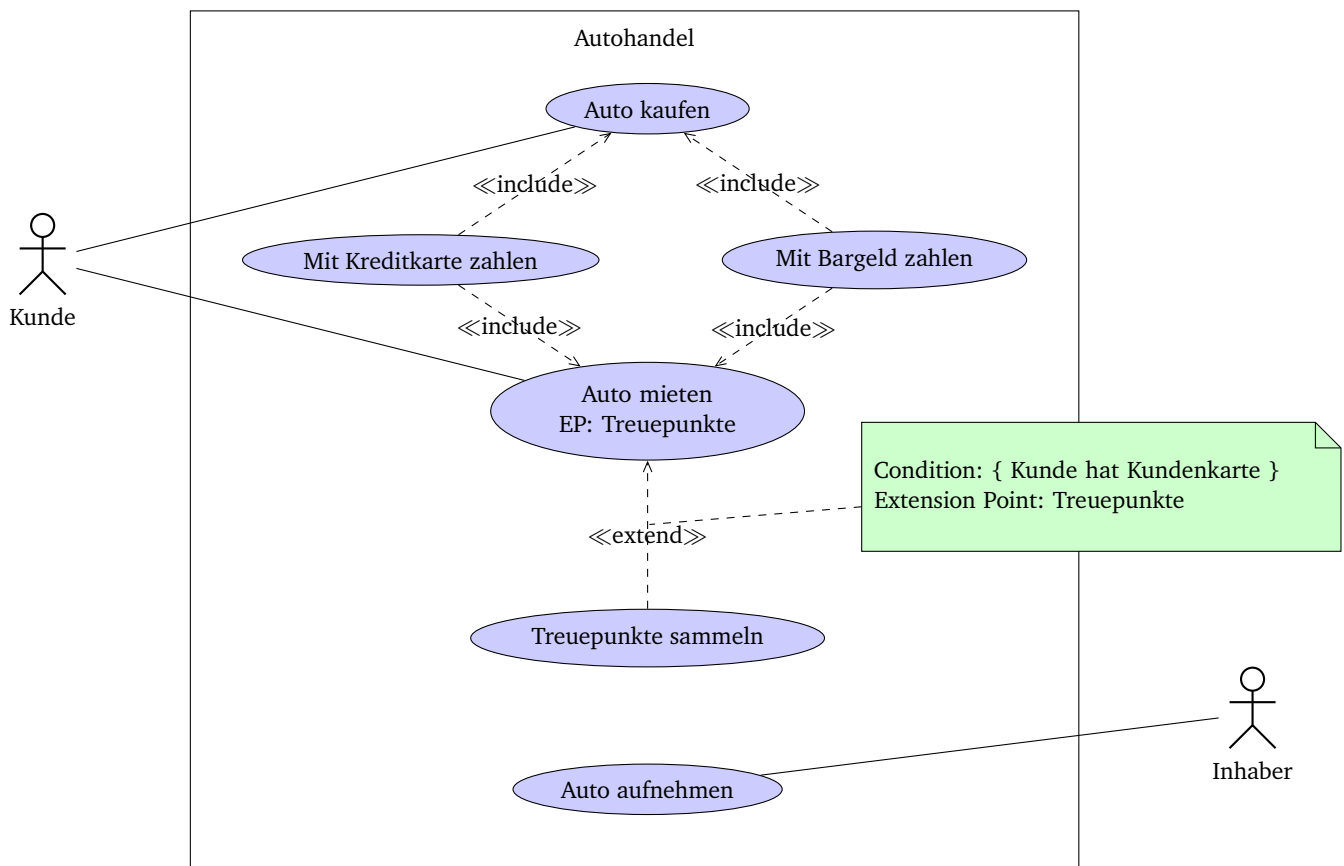


ABBILDUNG 1: Beispiel: Use Case Diagram

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.*

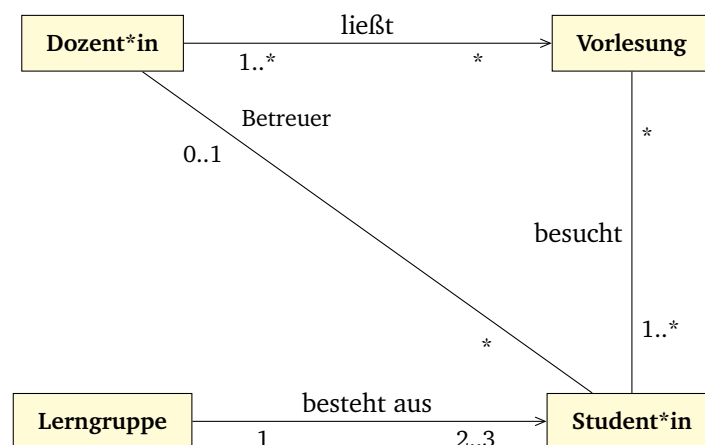


ABBILDUNG 2: Beispiel: Domänenmodell

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 (Protokoll).

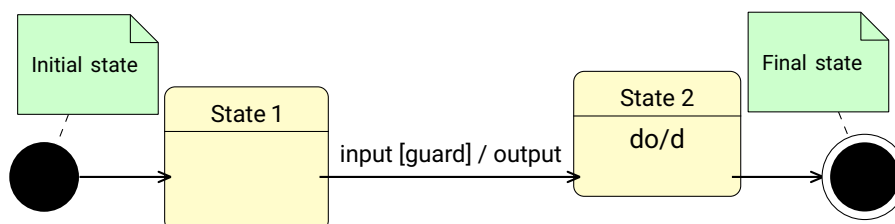


ABBILDUNG 3: Beispiel für UML State Machine Diagram

5 Softwarequalität

5.1 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.

5.1.1 Interne Faktoren

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

5.1.2 Externe Faktoren

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

5.2 Verifikation und Validierung

Verifikation: Wird das System korrekt erstellt?

Validierung: Wird das richtige System erstellt?

5.2.1 Techniken

Statische Techniken

Statische Techniken erfordern nicht, dass das Programm ausgeführt wird.

Software Reviews Händische/Manuelle Überprüfung

Automatisierte Statische Analyse Softwareanalyse, bspw. Typprüfer

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

Siehe 5.3.

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

5.2.2 Codeuntersuchung

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 Array-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?

5.3 Metriken

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)

5.3.1 Kontrollflussgraph (CFG)

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

Beispiel

Der in 4 gezeigte Code wird in 5 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 4: Beispiel: Kontrollflussgraph / Code

5.3.2 Zyklomatische Komplexität

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) darstellt.

5.4 Testen

5.4.1 Testplan

Ein 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.

5.4.2 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, ...).

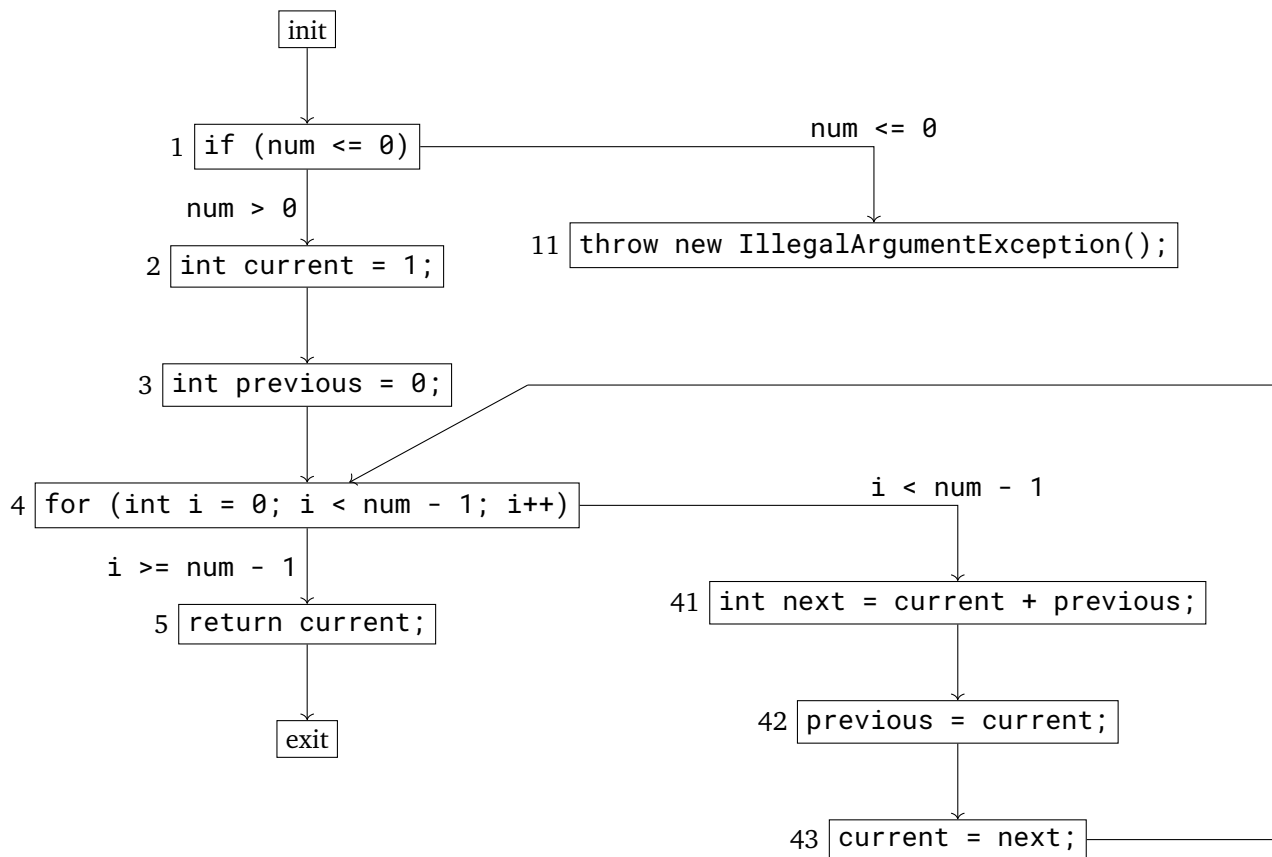


ABBILDUNG 5: Beispiel: Kontrollflussgraph

5.4.3 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.

5.4.4 Testabdeckung

Strukturell

Strukturelle Testabdeckung basiert auf dem Kontrollflussgraphen (CFG) eines Programms.

Statement Coverage (SC)

Alle Ausdrücke wurden mindestens einmal ausgeführt.

Basic Block Coverage (BBC)

Alle Blöcke 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).

Logisch

Definition:

Bedingung Eine *Bedingung* ist ein boolescher Ausdruck.

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

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.

6 Verhaltensmodellierung

6.1 Diagramme

6.1.1 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 6 (Codebasis 7) ist ein Beispiel für ein Sequenzendiagramm gegeben. In diesem werden die einzelnen Komponenten erläutert

Beispiel

Das Verhalten der Methode `run(. . .)` in dem in 6 gezeigtem Code wird in 7 visualisiert.

6.1.2 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 6: Beispiel: UML Sequenzen Diagramm / Code

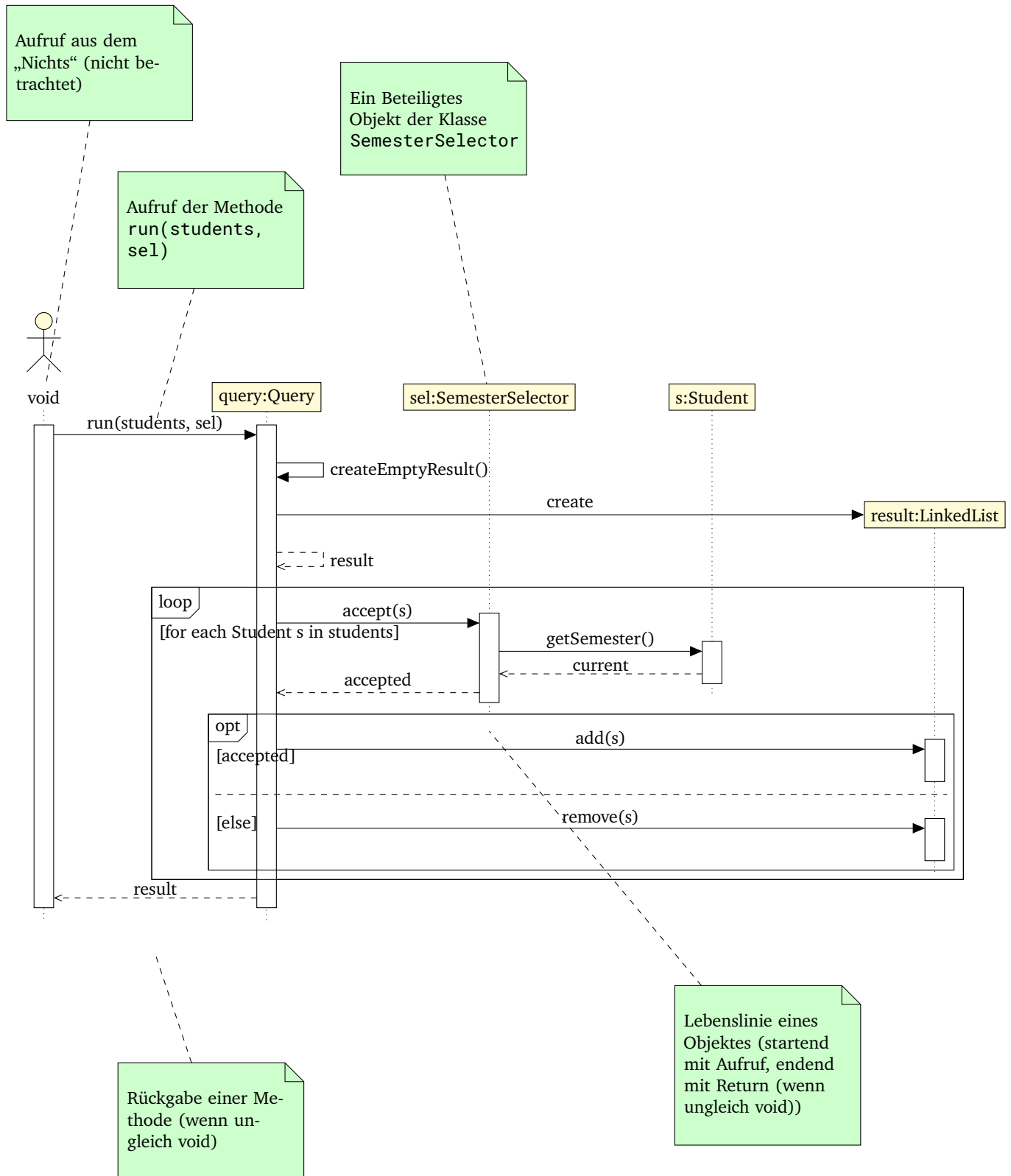


ABBILDUNG 7: Beispiel: UML Sequenzen Diagramm

7 Softwaredesign

7.1 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.

7.1.1 Zuständigkeiten

Diagramm: Class-Responsibility-Collaborator-Karten (CRC-Karten)

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 • Der Name sollte deskriptiv und eindeutig sein.

Responsibilities • Lange Zuständigkeitsliste \implies Sollte die Klasse aufgeteilt werden?
• Zuständigkeiten sollten zusammenhängen.

Collaborations • Viele Kollaboratoren \implies Sollte die Klasse aufgeteilt werden?
• **Vermeide kyklische Kollaboration!** \implies Es sollten höhere Abstraktionsebenen eingeführt werden.

7.1.2 Kopplung

Kopplung ist ein Richtwert zur Messung der Abhängigkeiten zwischen Klassen und zwischen Paketen an:

- Die Klasse C1 ist gekoppelt mit Klasse C2, wenn C2 eine direkte oder indirekte Abhängigkeit von C1 ist.
- Eine Klasse, welche auf 2 anderen Klassen basiert, hat eine lockere 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

Eine Klasse mit fester Kopplung ist zu vermeiden, da:

- Änderungen in verbundenen Klassen ändern mglw. die lokale Logik.
- Schwer zu verstehen ist, ohne das Restsystem zu betrachten.
- Schwer wiederzuverwenden ist in anderen Projekten, da alle Abhängigkeiten ebenfalls erforderlich sind.

Lose Kopplung unterstützt das Design von relativ unabhängigen und dadurch wiederverwendbaren Klassen:

- Generische Klassen mit hoher Wiederverwendbarkeitswahrscheinlichkeit sollten eine geringe Kopplung aufweisen.
- Wenig oder keine Kopplung ist aber kein generelles Ziel:
 - Das Grundkonzept von OOP ist, dass Klassen miteinander kommunizieren.
 - Lose Kopplung birgt die Gefahr, einzelne Objekte zu bauen, welche zu viele Aufgaben übernehmen.

Starke Kopplung an stabile Elemente und fundamentale Bibliotheken ist (normalerweise) kein Problem.

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

7.1.3 Kohäsion

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

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

1. **Zufällig** (keine Kohäsion vorhanden): Kein sinnvoller Zusammenhang zwischen den Elementen einer Klasse (bspw. bei Utility-Klassen).
2. **Zeitliche Kohäsion**: Alle Elemente einer Klasse werden „zusammen“ ausgeführt.
3. **Sequentielle Kohäsion**: Das Ergebnis einer Methode wird an die nächste übergeben.
4. **Kommunikative Kohäsion**: 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.

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)

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

Definition: Sei C eine Klasse, F die Instanzvariablen der Klasse C und M die Methoden der Klasse C (Konstruktoren sind keine Methoden). Sei $G = (V, E)$ ein ungerichteter Graph mit den Knoten $V = M$ und den Kanten

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

dann ist $LCOM(X)$ definiert als die Anzahl der zusammenhängenden Komponenten des Graphen G . Ist N die Anzahl der Methoden, so gilt $LCOM(X) \in [0, N]$.

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

7.2 Prinzipien

7.2.1 Single-Response-Principle (SRP)

Eine Klasse sollte nur eine Zuständigkeit haben.

7.3 Probleme

7.3.1 God Class

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.

7.3.2 Class Proliferation

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

8 Entwurfskonzepte

- 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

8.1 Idiome

Idiome sind *keine* Entwurfsmuster:

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

8.2 Entwurfsmuster

Ein Entwurfsmuster beschreibt...

- ein Problem, welches immer wieder auftritt,
- den Kern der Lösung des Problems, soweit abstrahiert, dass die Lösung auf viele Probleme anwendbar ist, ohne zweimal genau das gleich zu tun.

8.2.1 Template Method

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

Beschreibung

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

Varianten/Erweiterungen

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

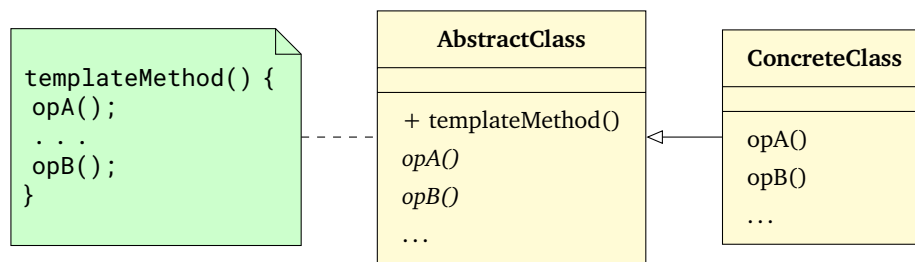


ABBILDUNG 8: UML: Template Method Pattern

8.2.2 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

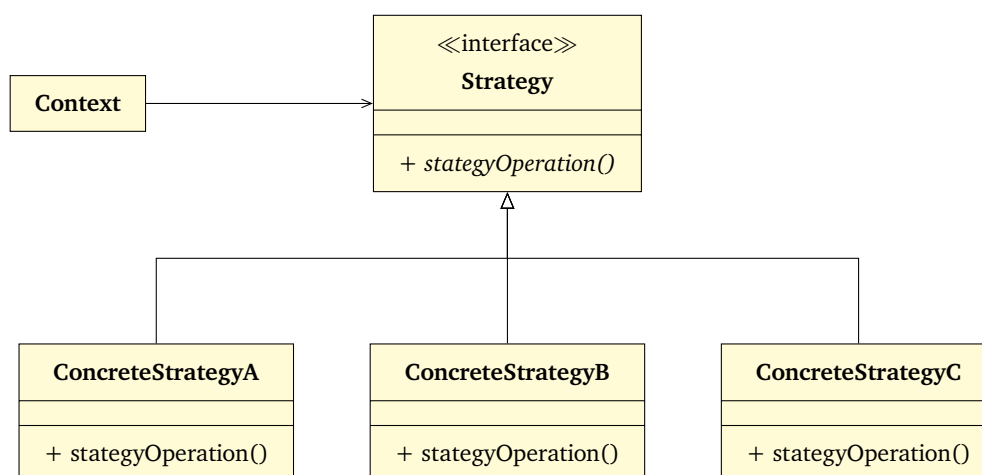


ABBILDUNG 9: UML: Strategy Pattern

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

8.2.3 Observer

Kurzfassung

Motivation: OOP 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.

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

Generisches Klassendiagramm

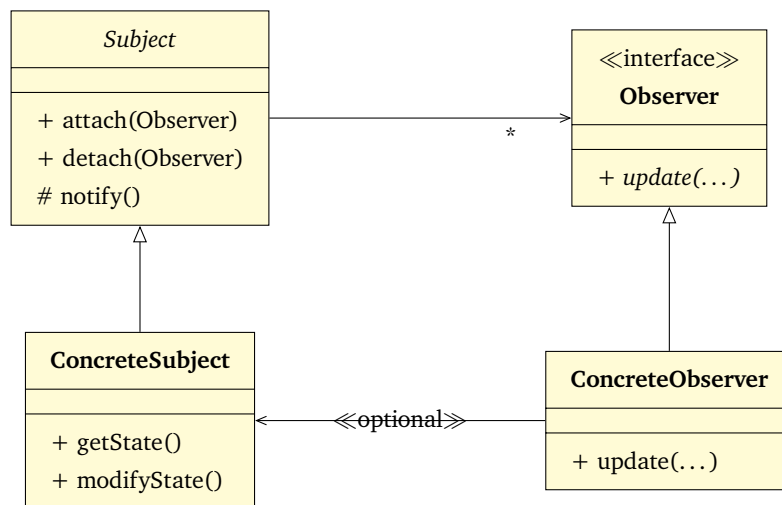


ABBILDUNG 10: UML: Observer Pattern

Beschreibung

Subject Bietet Methoden zur Implementierung des Musters an.

Observer Definiert ein Interface zum Empfangen von Signalen eines Subjekts.

ConcreteSubject Das konkrete Subjekt, sendet Benachrichtigungen an die Observer

ConcreteObserver Ein konkreter Observer, registriert sich beim Subjekt, empfängt Nachrichten von dem Subjekt

Varianten/Erweiterungen

8.3 Architekturmuster

Architekturmuster sind *keine* Entwurfsmuster:

- Hilfe bei der Spezifikation der Grundlegenden Struktur der Software
- Großer Effekt auf die konkrete Softwarearchitektur
- Definition von globalen Eigenschaften, bspw.:
 - Wie unterschiedliche Komponenten zusammenarbeiten und Daten austauschen
 - Einschränkungen der Subsysteme
 - ...

8.3.1 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

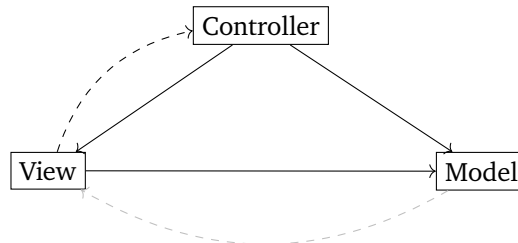


ABBILDUNG 11: Model-View-Controller

Controller und View sind direkt gekoppelt mit dem Modell, das Modell ist nicht direkt gekoppelt mit dem Controller oder der View (siehe 11).

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.