Software Engineering

Zusammenfassung

Fabian Damken

8. November 2023

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einführung **4**
1.1 Software . . . 4
1.2 Bereiche der Softwareentwicklung . . . 4

2 Softwareprozesse **6**
2.1 Grundlegende Begriffe . . . 6
2.1.1 Grundlegende Schritte in der Softwareentwicklung . . . 6
2.2 Projektplanung . . . 6
2.2.1 Diagramm: Aktivitätsdiagramm . . . 6
2.2.2 Diagramm: Gantt Chart . . . 6
2.3 Projektmanagement . . . 7
2.3.1 Prozessmodelle . . . 7
2.3.2 Aktivitäten . . . 8
2.3.3 Projektplanung . . . 8
2.3.4 Prozessmodell: Wasserfall . . . 9
2.3.5 Kritik . . . 9
2.3.6 V-Modell . . . 10
2.3.7 Agile Entwicklung . . . 10
2.3.8 Prozessmodell: eXtreme Programming (XP) . . . 10
2.4 Anforderungsanalyse . . . 13
2.4.1 Anforderungstypen . . . 13
2.4.2 Anforderungsklassifizierungen . . . 14
2.5 Anwendungsfälle . . . 15
2.5.1 „Fully Dressed“ Use Case . . . 15
2.5.2 Diagramm: Use Case (UML) . . . 15
2.6 Domänenmodellierung . . . 17
2.6.1 Diagramm: Domain Model (UML) . . . 18

3 Softwarequalität **19**
3.1 Faktoren . . . 19
3.1.1 Interne Faktoren . . . 19
3.1.2 Externe Faktoren . . . 19
3.2 Verifikation und Validierung . . . 20
3.2.1 Techniken . . . 20
3.2.2 Codeuntersuchung . . . 20
3.3 Metriken . . . 21
3.3.1 Kontrollflussgraph (CFG) . . . 21
3.3.2 Zyklomatische Komplexität . . . 21
3.4 Testen . . . 22
3.4.1 Testplan . . . 22

3.4.2 Testtypen . . . 22
3.4.3 Testautomation . . . 23
3.4.4 Testabdeckung . . . 23

4 Verhaltensmodellierung 25
4.1 Diagramme . . . 25
4.1.1 Diagramm: Interaction/Sequence Diagram (UML) . . . 25
4.1.2 Diagramm: State Machine Diagram (UML) . . . 25

5 Softwaredesign 28
5.1 Heuristiken . . . 28
5.1.1 Zuständigkeiten . . . 28
5.1.2 Kopplung . . . 29
5.1.3 Kohäsion . . . 30
5.2 Prinzipien . . . 31
5.2.1 Single-Response-Principle (SRP) . . . 31
5.3 Probleme . . . 31
5.3.1 God Class . . . 31
5.3.2 Class Proliferation . . . 31

6 Entwurfskonzepte 32
6.1 Idiome . . . 32
6.2 Entwurfsmuster . . . 32
6.2.1 Template Method . . . 32
6.2.2 Strategy . . . 33
6.2.3 Observer . . . 34
6.3 Architekturmuster . . . 35
6.3.1 Model-View-Controller (MVC) . . . 35

7 Diagramme 37

8 Glossar 38

1 Einführung

1.1 Software

Software enthält nicht nur ein einziges Programm, sondern unter anderem:

- ein Ausführbares Programm und dessen Daten
- Konfigurationsdateien
- System Dokumentation (bspw. Architektur, Analyse, Design, ...)
- Nutzerdokumentation (Handbuch)
- Support (bspw. Webseite, Telefon, ...)
- ...

Softwaretypen

Applikationssoftware Software, welche direkt mit dem Nutzer interagiert wie Textverarbeitungsprogramme oder komplexe Software wie Steuerverwaltung.

Systemsoftware Software, welche üblicherweise nicht direkt mit dem (normalen) Nutzer interagiert, beispielsweise Firmware und Treiber.

Software as a Service (SaaS) Serverseitige Software, welche meist über den Browser an den Nutzer ausgeliefert wird.

1.2 Bereiche der Softwareentwicklung

Zu dem Prozess der Softwareentwicklung gehören:

- **Software Anforderungen** Erkennung und Definition der Erwartungen an das System und dessen Verhalten.
- **Software Design** Die Struktur der Software (Module, Klassen, API, ...).
- **Software Qualität** Qualitätssicherung: Einschätzung der Qualität der Software und die Entwicklung, diese zu verbessern.
- **Testen, Verifikation und Validierung der Software** Die systematische Erkennung (und Eliminierung) von Fehlern.

- **Software Wartung** Der Betrieb der Software, die Weiterentwicklung und die Anpassung an neue Anforderungen und Umgebungen.
- **Software Konfiguration und Management** Verwaltung von unterschiedlichen Versionen und Konfigurationen der Software.
- **Softwareentwicklungstools und -methoden** UML Tools, IDEs, Versionskontrollsysteme, Aufgabenverwaltung, Statische Analyse, ...
- **Softwareentwicklungsprozesse** Definition und Verbesserung des Enwticklungsprozesses.

2 Softwareprozesse

2.1 Grundlegende Begriffe

2.1.1 Grundlegende Schritte in der Softwareentwicklung

Anforderungsspezifikation Definition der Software, welche entwickelt werden soll und die Einschränkungen dieser Operation.

Entwicklung Design und Implementierung der Software.

Validierung Sicherung, dass die Software erfüllt, was der Kunde verlangt.

Evolution Adaption und Modifikation der Software, um mit Änderungen des Kunden oder der Marktbedingungen umzugehen.

2.2 Projektplanung

2.2.1 Diagramm: Aktivitätsdiagramm

Beschreibung Ein Aktivitätendiagramm stellt die Abhängigkeiten zwischen Aktivitäten und Meilensteinen als Graph dar, aus welchem Schlüsse über die Projektdauer gezogen werden können.

Beispiel In Abbildung 2.1 ist ein Beispiel für ein Aktivitätendiagramm mit den Aktivitäten T1, T2, T3, T4, T5 und T6 und den Meilensteinen M1 und M2 gegeben.
In Abbildung 2.2 wurden die kritischen Pfade markiert.

Metriken

Kritischer Pfad Der längste Pfad zwischen Start- und Endpunkt. Sollten sich Aktivitäten auf dem kritischen Pfad verspäten, verspätet sich das gesamte Projekt. In Abbildung 2.2 wurde der kritische Pfad zwischen Start und Release markiert.

2.2.2 Diagramm: Gantt Chart

Beschreibung Ein Gantt Char stellt die Aktivitäten in einem Projekt im zeitlichen Verlauf in Relation zu den Meilensteinen dar.
Das Beispiel in 2.3 stellt das Aktivitätendiagramm aus 2.1 als Gantt Chart dar.

Beispiel

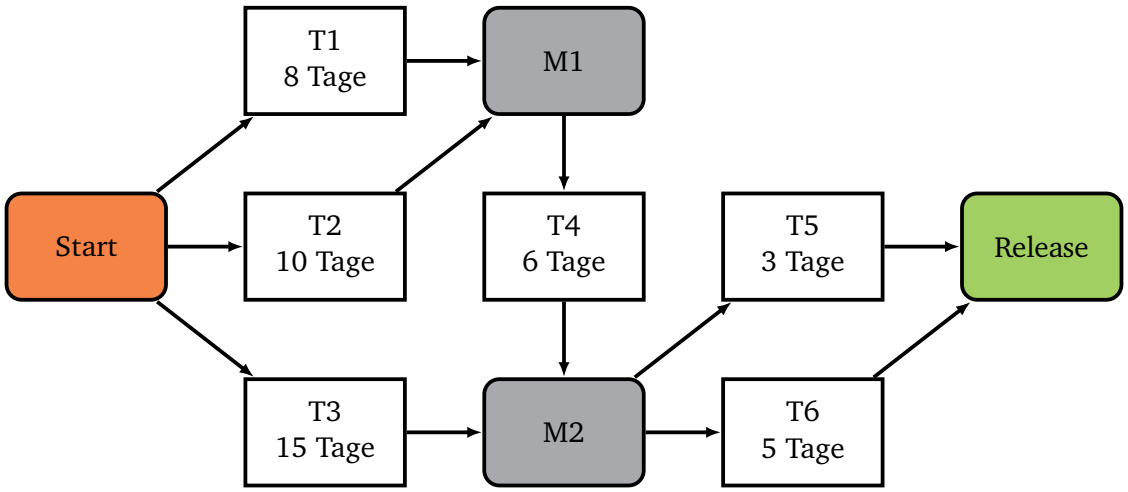


Abbildung 2.1: Aktivitätsdiagramm: Beispiel

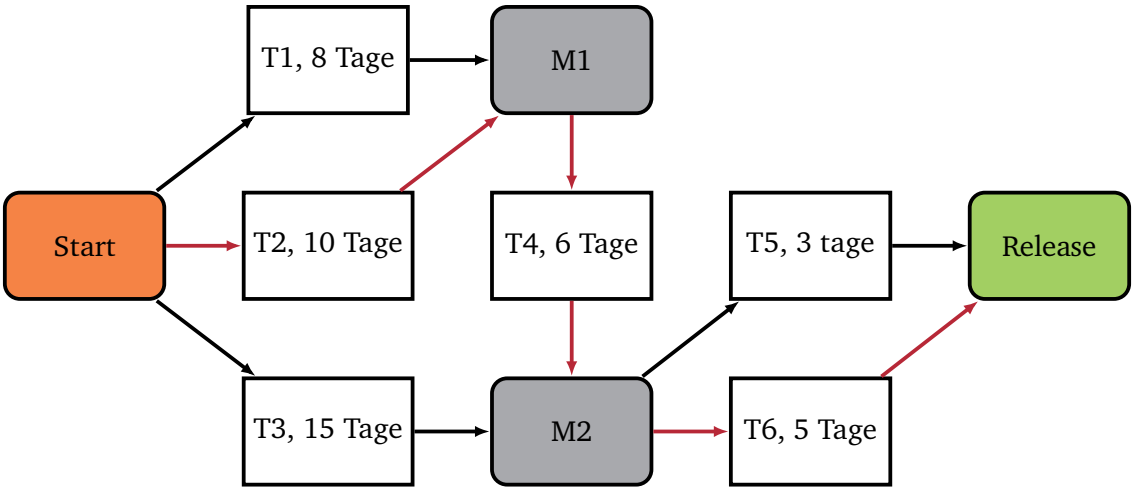


Abbildung 2.2: Aktivitätsdiagramm: Beispiel mit markiertem kritischen Pfad (rot)

2.3 Projektmanagement

2.3.1 Prozessmodelle

Softwareprozessmodelle sind simplifizierte und abstrakte Beschreibungen eines Entwicklungsprozesses, welche eine Sicht auf die Entwicklung darstellen. Sie enthalten Aktivitäten innerhalb der Entwiclung, Softwareprodukte (bspw. Architekturbeschreibungen), Quellcode, Nutzer-Dokumentation und die Rollen der an der Entwicklung beteiligten Personen.

Beispiele für Prozessmodelle sind:

- Waterfall
- Spiral
- V-Model (XT)

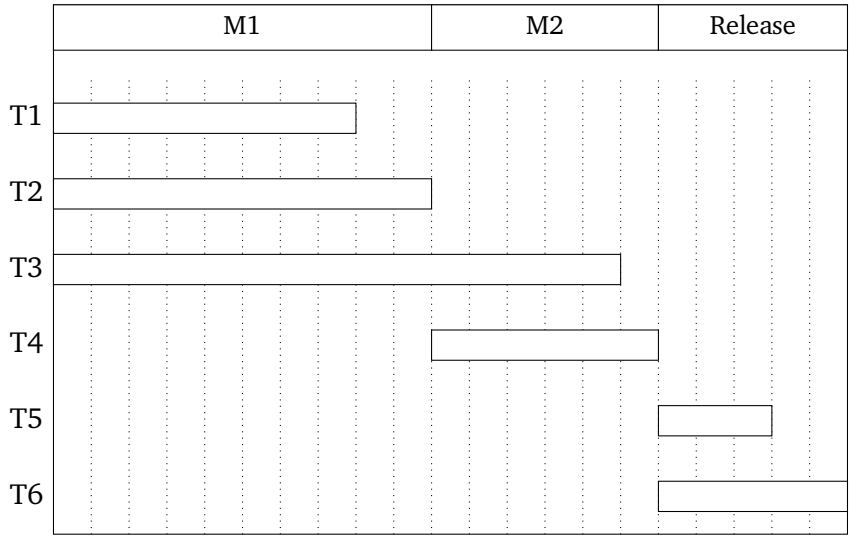


Abbildung 2.3: Gantt Chart: Beispiel

- eXtreme Programming

2.3.2 Aktivitäten

- Projektplanung
- Kostenabschätzung
- Projektüberwachung und -sichtigungen
- Personalselektion und -evaluierung
- Präsentation

2.3.3 Projektplanung

Ein Softwareprojekt enthält viele Pläne, wobei ein guter Startplan mit allen vorliegenden Informationen Vorraussetzung ist für das Gelingen des Projektes.

Es gibt viele Unterschiedliche Pläne, bspw.:

- **Projektplan** Siehe Sektionen „Projektplan“ (2.3.3).
- **Qualitätssicherungsplan** Beschreibung, welche Qualitätssicherungsprozeduren und Standards genutzt werden.
- **Personalentwicklungsplan** Beschreibt, wie die Fähigkeiten und Erfahrungen der einzelnen Mitarbeiter und des Teams eingesetzt und weiterentwickelt werden.

Projektplan

Struktur

Einleitung Ziele des Projektes und Einschränkungen (Zeit, Budget, ...).

Projektorganisation Organisation des Entwicklungsteams, die eingebundenen Personen und deren Rollen.

Risikoanalyse Mögliche Projektrisiken, deren Eintrittswahrscheinlichkeit und Risikominderungsstrategien.

Resourcenanforderungen Nötige Hard- und Software für das Durchführen des Projektes.

Arbeitsaufteilung Aufteilung des Projektes in Aktivitäten, Identifikation von Meilensteinen und Ergebnisse jeder Aktivität.

Zeitplan Abhängigkeiten zwischen Aktivitäten, geschätzte Zeit bis zu jedem Meilenstein und die Zuweisung von Personen zu Aktivitäten.

Mechanismen zur Überwachung und Mitteilung

Die Aktivitäten können in einem Aktivitätendiagramm („Activity Diagram“) dargestellt werden, siehe 2.2.1.

2.3.4 Prozessmodell: Wasserfall

Beschreibung

Die folgenden Aktivitäten werden strikt in der folgenden Reihenfolge abgearbeitet und der nächste Schritt wartet auf den vorherigen:

1. Anforderungsdefinition
2. System- und Softwaredesign
3. Implementation und Unittests
4. Integrations- und Systemtests
5. Inbetriebnahme und Wartung

Sollten Fehler in einer Phase auftreten, so wird diese gestoppt und die vorherige Phase muss wiederholt werden. Sommit fließen die Ergebnisse aller Phasen in die vorherigen ein, wie in 2.4 zu sehen ist.

2.3.5 Kritik

- Kein iterativer Prozess
- Änderung von Anforderungen und Design schwierig und Zeitaufwändig
- Das Testen der Software startet erst in späteren Phasen $\Longrightarrow$ Fehler werden erst spät erkannt
- Einzelne Phasen werden von unterschiedlichen Teams vorangetrieben, welche später möglicherweise nicht mehr zur Verfügung stehen können.

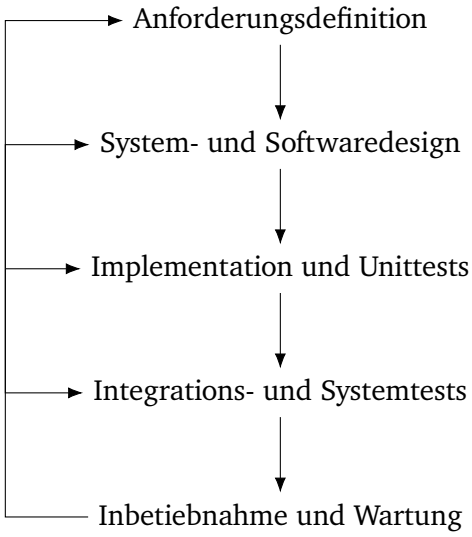


Abbildung 2.4: Prozessmodell: Waterfall

2.3.6 V-Modell

Siehe 2.5.
Nachteil: Bei einem V-Modell wird 50% des Budgets für das V-Modell ausgegeben.

2.3.7 Agile Entwicklung

Ziel: Möglichst schnelle Entwicklung von Software und Einarbeitung von Änderungen der Anforderungen

- Höchste Priorität hat das Zufriedenstellen des Kunden
- Regelmäßige Auslieferung von Zwischenständen (bspw. alle zwei Wochen)
- Funktionierende Software ist das Primärziel und der Fortschritt (30% Implementiert $\rightarrow$ 30% des Projektes abgeschlossen)
- Einfachheit ist essentiell
- Änderungen der Anforderungen
- Das Team reflektiert in regelmäßigen Abständen, um effektiver zu werden
- Manager und Entwickler müssen zusammen arbeiten

2.3.8 Prozessmodell: eXtreme Programming (XP)

In XP wird in kurzen Iterationen gearbeitet, wobei in jeder Iteration bestimmte User Stories abgearbeitet werden. Die abzuarbeitenden User Stories werden in der Iterationsplanung festgelegt.

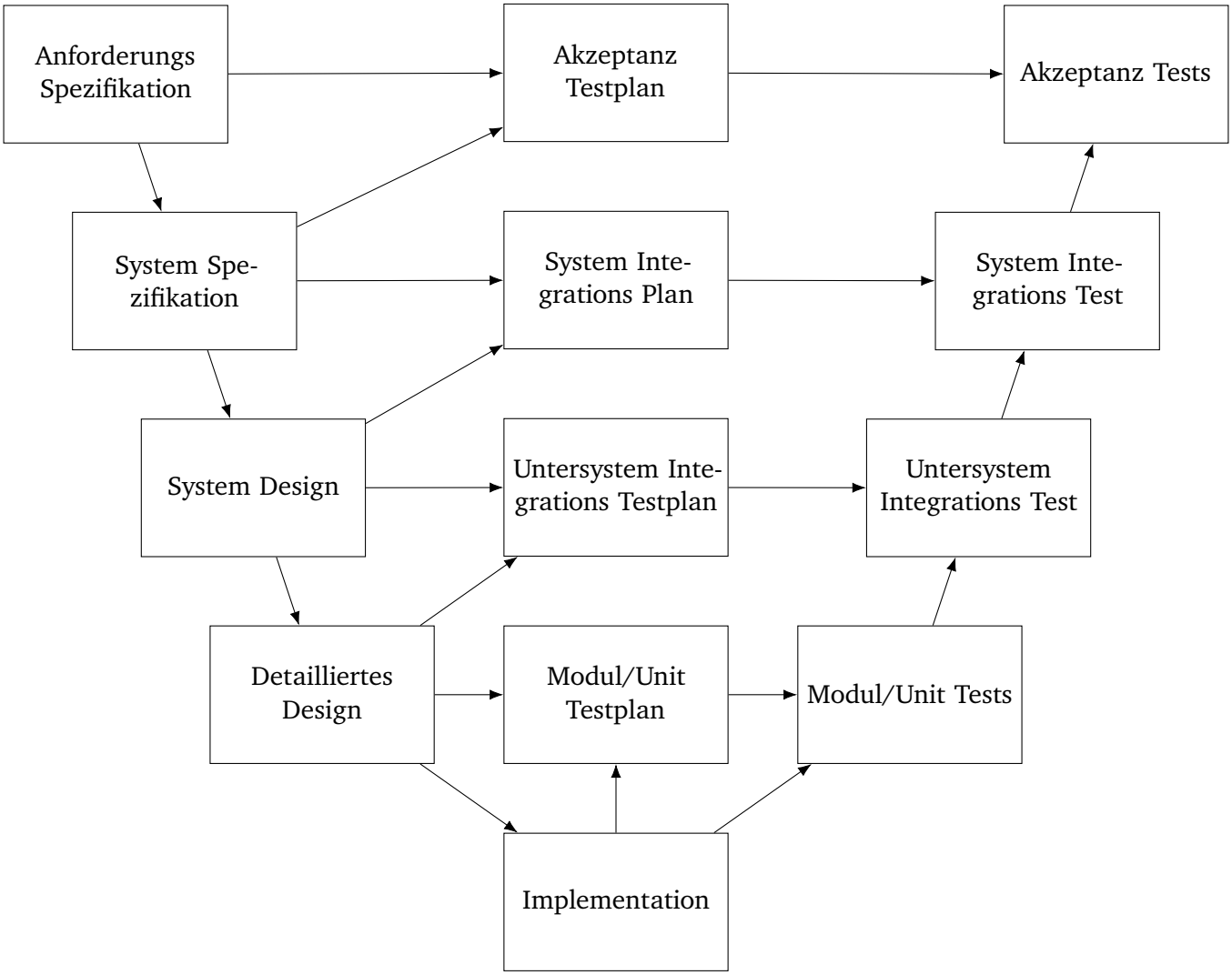


Abbildung 2.5: V-Modell

User Stories

Der essentielle Teil der Entwicklung. Eine User Story beschreibt eine Anforderung. Eine User Story wird anschließend in einzelne Tasks aufgebrochen.

- User Stories müssen für den Kunden verständlich sein
- Jede User Story muss von Wert sein für den Kunden
- User Storie sollten so umfangreich sein, dass man einige abarbeiten kann pro Iteration
- User Stories sollten unabhängig sein
- Jede User Story muss testbar sein
- Jeder User Story werden *Story Points* zugewiesen, welche ein abstraktes Zeitmaß darstellen zur Einschätzung der Bearbeitungsdauer

- In jeder User Story sollte ein Akzeptanzkriterium gegeben sein

Langes Template: Als ein <Rolle> möchte ich <Ziel> sodass <Vorteile>.
Kurzes Template: Als ein <Rolle> möchte ich <Ziel>.

Entwicklung

Testgetriebene Entwicklung Jede Implementierung startet mit der Entwicklung von Tests, sodass die Implementierung korrekt ist, sobald alle Tests funktionieren.

Kontinuierliche Integration Entwickler checken ihren Code regelmäßig ein (mehrmals am Tag), sodass ein CI-Server regelmäßig Tests ausführen kann (automatisierte Tests und Build System).

Pair Programming Code ist von genau zwei Entwicklern geschrieben

- Einer fokussiert sich auf den besten Weg der Implementierung
- Der andere fokussiert sich auf den Code (aus einer strategischen Sicht)

Planung

Initiale Planung (Start des Projektes)

- Entwickler und Kunden arbeiten gemeinsam die relevanten User Stories heraus
- Entwickler schätzen die Story Points für jede User Story
  Eine User Story mit doppelt so vielen Story Points wie eine andere sollten doppelt so lange dauern.
- Für eine genaue Zeitabschätzung ist die *Velocity* (Story Points pro Zeit) erforderlich, diese wird mit der Zeit genauer (eine Prototyp-Implementierung zur Messung der Velocity nennt sich *Spike*)

Release Planung

- Entwickler und Kunden einigen sich auf ein Datum für das erste Release (2-4 Monate)
- Kunden entscheiden sich für grobe Zusammensetzung der User Stories (unter Beachtung der Velotiy)
- Wird die Velocity genauer, wird der Releaseplan geändert

Iterations Planung

- Der Kunde sucht die User Stories für die Iteration
- Die Reihenfolge der Abarbeitung ist eine technische Entscheidung
- Die Iteration endet an einem bestimmten Tag, auch wenn nicht alle Stories abgearbeitet wurden
- Die Schätzung für alle *erfolgreich abgearbeiteten* Stories wird summiert und die Velocity berechnet
- Die geplante Velocity der nächsten Iteration basiert auf der Velocity der vorherigen Iteration

$$\text{Velocity} = \frac{\text{Summe an Story Points}}{\text{Summe der verbrauchten Zeit pro User Story in einer Iteration}}$$

Aufgaben Planung

- Stories werden auf Aufgaben (Tasks) heruntergebrochen, welche 4h bis 16h benötigen
- Jeder Entwickler kann sich Aufgaben aussuchen

2.4 Anforderungsanalyse

Anforderungen sind Beschreibungen der Dienste, welche von dem System zur Verfügung gestellt werden sollen und die Einschränkungen des selbigen. Die Anforderungsanalyse beschäftigt sich mit

- dem Erkennen,
- der Analyse,
- der Dokumentation und
- der Validierung

der Anforderungen. Der Prozess ist in 2.6 aufgezeigt.

Diese Anforderungen werden in einem sogenannten Pflichtenheft (eng. „System Requirements Specification“) niedergeschrieben.

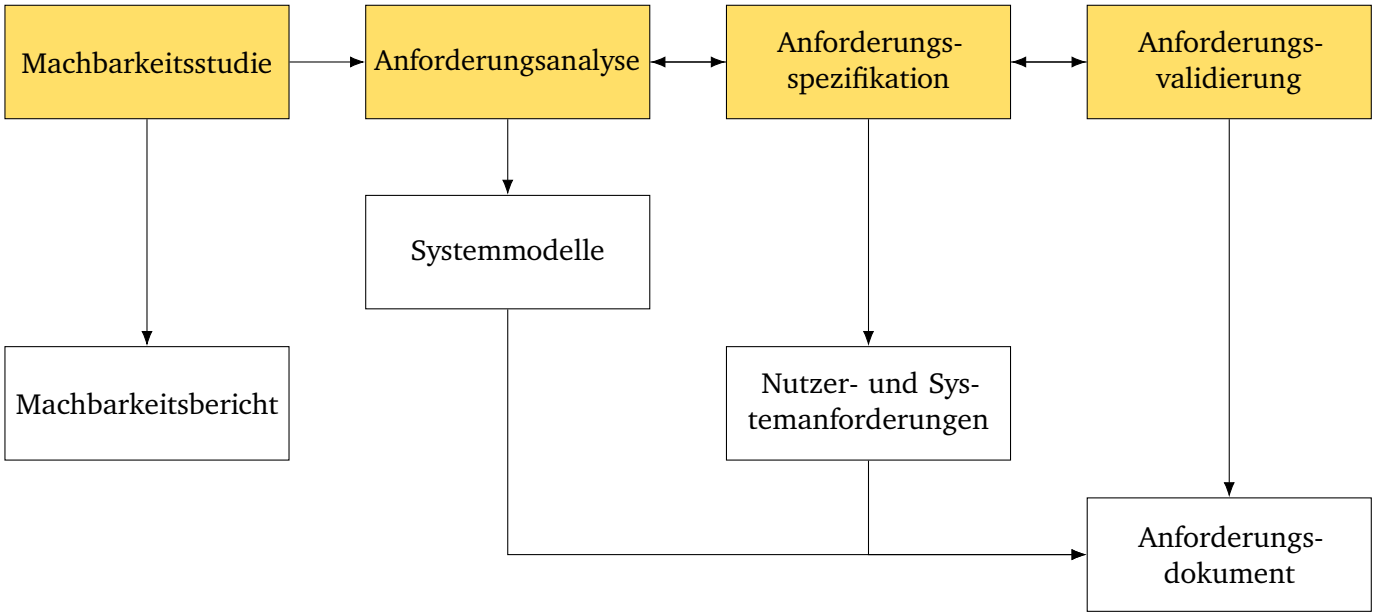


Abbildung 2.6: Anforderungsanalyse: Arbeitsablauf

2.4.1 Anforderungstypen

Nutzeranforderungen

- Natürlicher Sprache oder Diagramme

- Oftmals auf einem hohen Level und abstrakt (meist von dem Kunden geschrieben)
- Beschreibung von:
  - Diensten
  - Produktionseinschränkungen

Beispiel Das Büchereisystem soll alle Daten halten, welche zur Lizensierung nötig sind.

Systemanforderungen

- Präzise und detaillierte Beschreibungen
- Meist von dem Entwickler geschrieben
- Beschreibung von:
  - Funktionen und Diensten
  - Einschränkungen

Beispiele Alle Anfragen an das System sollen fünf Jahre gespeichert werden.

Domänenanforderungen

Domänenanforderungen sind meist nicht von dem Kunden oder dem Entwickler spezifiziert, sondern von der jeweiligen Anforderungsdomäne.

- Meist ausgedrückt in Domänen-spezifischen Sprachen und daher für den Softwareentwickler schwer zu verstehen.
- Von Domänen-Experten oftmals implizit angenommen.

Beispiel Neue Veröffentlichungen in der Bücherei sollen automatisch an die nationale Bibliothek geleitet werden.

2.4.2 Anforderungsklassifizierungen

Die Anforderungen (Nutzeranforderungen, Systemanforderungen, Domänenanforderungen) können jeweils noch in *funktionale* und *nichtfunktionale* Anforderungen klassifiziert werden.

Funktionale Anforderungen

Funktionale Anforderungen spezifizieren

- die Dienste, welche das System zur Verfügung stellt,
- die Reaktion des Systems auf bestimmte Eingaben und
- das Verhalten des Systems in bestimmten Situationen.

Beispiel Wenn der Nutzer den Knopf „Seite Anlegen“ drückt, wird eine neue Seite angelegt.

Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen spezifizieren Einschränkungen der Dienste/Funktionen des Systems, welche oftmals nicht vollständig von Tests abgedeckt werden können:

- Produktanforderungen
  - Portabilität
  - Verlässlichkeit
  - Effizienz (Performanz, Speicherplatz, ...)
  - Nutzbarkeit
- Organisatorische Anforderungen
  - Auslieferung
  - Implementation
  - Nutzung von Standards (ISO, IEEE, ...)
- Externe Anforderungen
  - Interoperabilität
  - Ethik
  - Recht (Sicherheit, Datenschutz, ...)

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.

2.5 Anwendungsfälle

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

2.5.1 „Fully Dressed“ Use Case

Ein *Fully Dressed Use Case* beschreibt einen Anwendungsfall sehr genau und sieht folgendermaßen aus:

2.5.2 Diagramm: Use Case (UML)

UML Specification Version 2.5.1, Chapter 18

Abschnitt	Beschreibung / Einschränkungen
Use Case Name	Der Name des Anwendungsfalls / Startet mit einem Verb
Bereich	Betroffene Bereiche des Systems
Ebene	Abstraktionsebene / Nutzerziel, Zusammenfassung oder Unterfunktion
Primärakteur	Initiator des Anwendungsfalls
Stakeholder	Personen, die dieser Anwendungsfall betrifft
Vorbedingungen	
Akzeptanzkriterien	Korrektheit der Implementierung
Erfolgskriterien	Nice-to-Have
Hauptszenario	Typische Schritte, des Anwendungsfalls / Nummerierte Liste
Erweiterungen	Alternative Erfolgs- und Fehlschlagszenarien / Fehlschlagspunkte des Hauptszenarios
Spezialanforderungen	Verwandte, nichtfunktionale Anforderungen
Technologien	Einzusetzende Technologien
Häufigkeit	Häufigkeit des Eintretens des Anwendungsfalls
Anderes	Bspw. offene Tickets

- Ebene

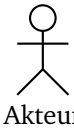
Nutzerziel Wichtigste Ziele; Essentielle Ziele des Nutzers

Zusammenfassung Beschreibt den Kontext mehrerer Nutzerziele

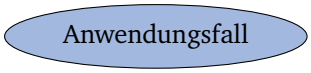
Unterfunktion Anwendungsfälle, welche Teil eines anderen Nutzerziels sind

Beschreibung Ein Use Case Diagramm stellt Anforderungsfälle und deren Relation zu dem System und den Akteuren dar.

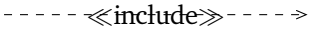
Im Diagramm 2.7 ist ein Beispiel für ein Use Case Diagramm gegeben. Im folgenden werden die einzelnen Teile erläutert.



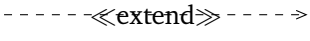
Stellt einen Akteur im System dar.



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“.

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 im Diagramm 2.7 dargestellt.

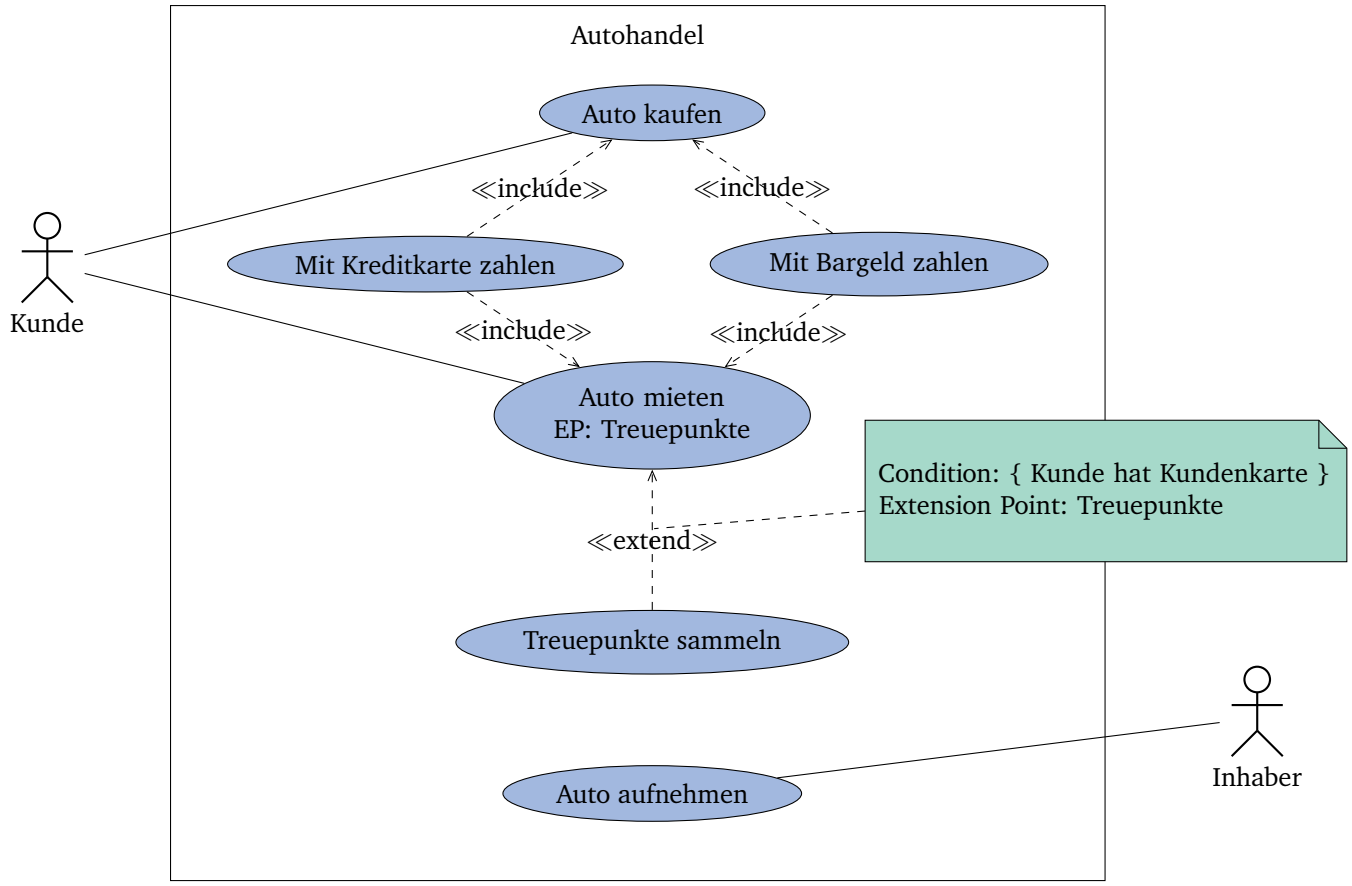


Abbildung 2.7: Beispiel: Use Case Diagram

2.6 Domänenmodellierung

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!

2.6.1 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 2.8 ist ein Beispiel für ein Domänenmodell gegeben. Im folgenden werden die einzelnen Komponenten erläutert.

Domänen-/Konzeptklasse	Stellt ein Domänenobjekt/eine Konzeptklasse im Domänenmodell dar.
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 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 2.8 dargestellt.*

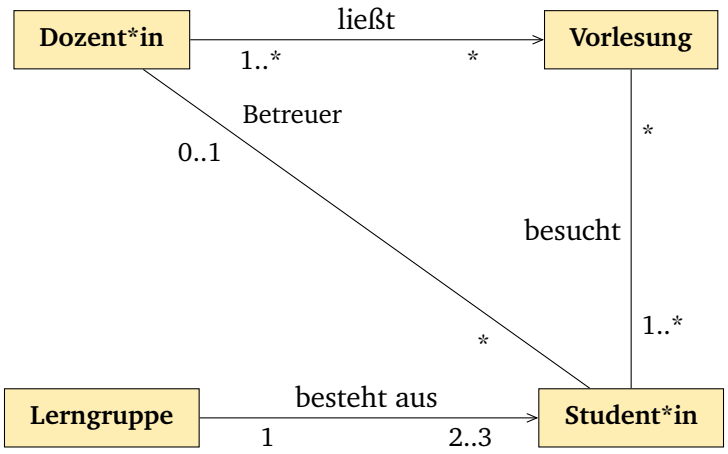


Abbildung 2.8: Beispiel: Domänenmodell

3 Softwarequalität

3.1 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.

3.1.1 Interne Faktoren

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

3.1.2 Externe Faktoren

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

3.2 Verifikation und Validierung

Verifikation: Wird das System korrekt erstellt?
Validierung: Wird das richtige System erstellt?

3.2.1 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 bestimme Eigenschaft erfüllt

Siehe 3.3.

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

3.2.2 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 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?

3.3 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)

3.3.1 Kontrollflussgraph (CFG)

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

Beispiel Der in 3.1 gezeigte Code wird in 3.2 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 3.1: Beispiel: Kontrollflussgraph / Code

3.3.2 Zyklomatische Komplexität

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

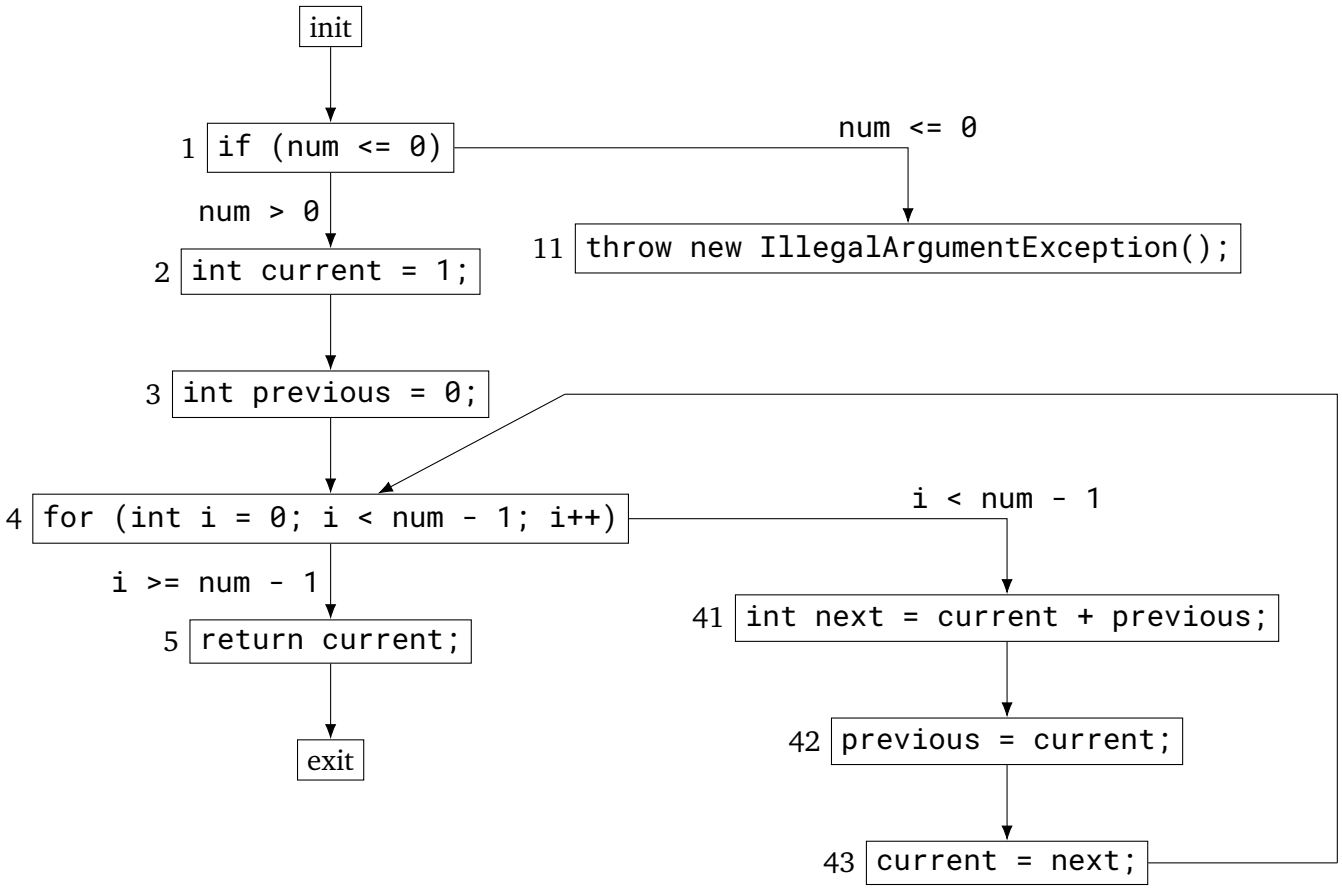


Abbildung 3.2: Beispiel: Kontrollflussgraph

3.4 Testen

3.4.1 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.

3.4.2 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, ...).

3.4.3 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.

3.4.4 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 Audruck.

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 auswertet,
- davon c einmal zu wahr und einmal zu falsch auswertet,
- 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.

4 Verhaltensmodellierung

4.1 Diagramme

4.1.1 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 4.1 (Codebasis 4.2) ist ein Beispiel für ein Sequenzendiagramm gegeben. In diesem werden die einzelnen Komponenten erläutert

Beispiel Das Verhalten der Methode `run(...)` in dem in 4.1 gezeigtem Code wird in 4.2 visualisiert.

4.1.2 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 (Protkoll)

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

```java
public class Query {
    public List<Student> run(List<Student> students, ISelector sel) {
        List<Student> result = createEmptyResult();
        for (Student s : students) {
            boolean accepted = sel.accept(s);
            if (accepted) {
                result.add(s);
            } else {
                result.remove(s);
            }
        }
        return result;
    }
}
```

```java
public interface ISelector {
    boolean accept(Student s);
}
```

```java
public class SemesterSelector implements ISelector {
    private int semester;

    public SemesterSelector(int semester) {
        this.semester = semester;
    }

    @Override
    public boolean accept(Student s) {
        int current = s.getSemester();
        boolean accepted = (semester == current);
        return accepted;
    }
}
```

```java
public class Student {
    private int semester;

    public Student(int semester) {
        this.semester = semester;
    }

    public int getSemester() {
        return semester;
    }
}
```

Abbildung 4.1: Beispiel: UML Sequenzen Diagramm / Code

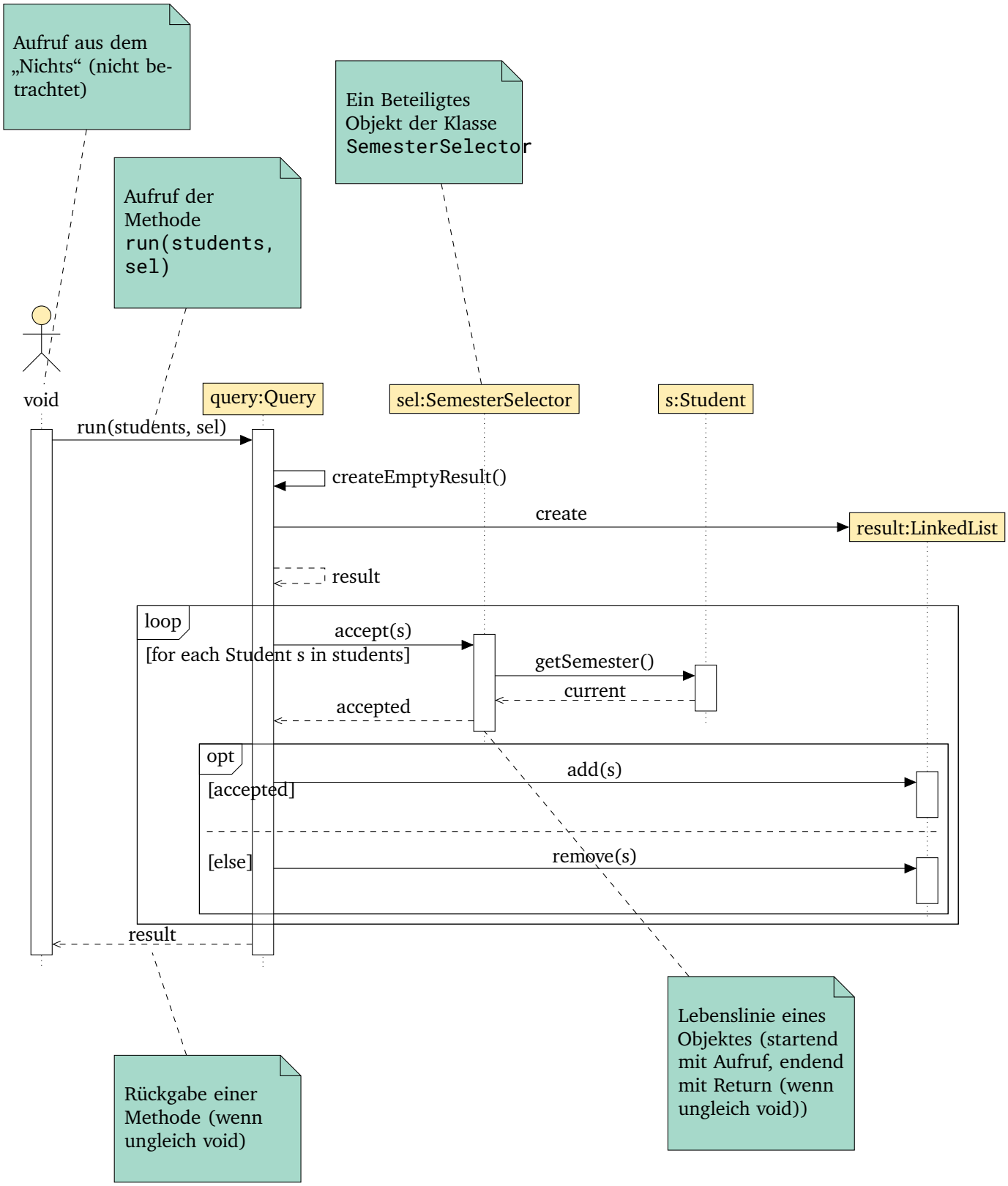


Abbildung 4.2: Beispiel: UML Sequenzen Diagramm

5 Softwaredesign

5.1 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 prottected sind.

5.1.1 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 deskreptiv und eindeutig sein.

Responsibilities
- Lange Zuständigkeitsliste $\Longrightarrow$ Sollte die Klasse aufgeteilt werden?
- Zuständigkeiten sollten zusammenhängen.

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

5.1.2 Kopplung

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

- Die Klasse `C1` is 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, das 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) Zukunftsprojekte birgt die Gefahr von unnötiger Komplexität und hohen Projektkosten!

5.1.3 Kohäsion

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** (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.

5.2 Prinzipien

5.2.1 Single-Response-Principle (SRP)

> Eine Klasse sollte nur eine Zuständigkeit haben.

5.3 Probleme

5.3.1 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.

5.3.2 Class Proliferation

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

6 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

6.1 Idiome

Idiome sind *keine* Entwurfsmuster:

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

6.2 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.

6.2.1 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

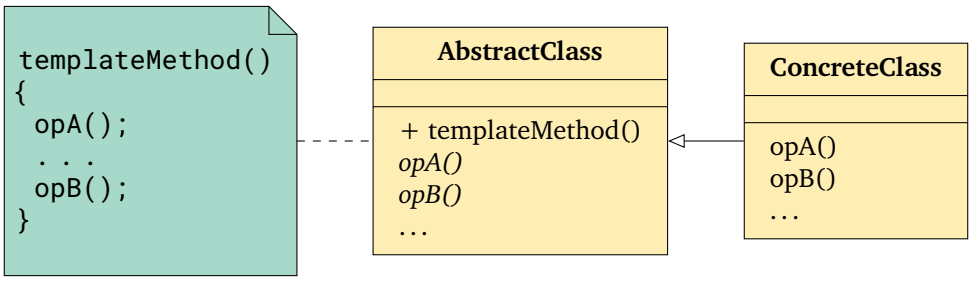


Abbildung 6.1: UML: Template Method Pattern

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*.

6.2.2 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

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

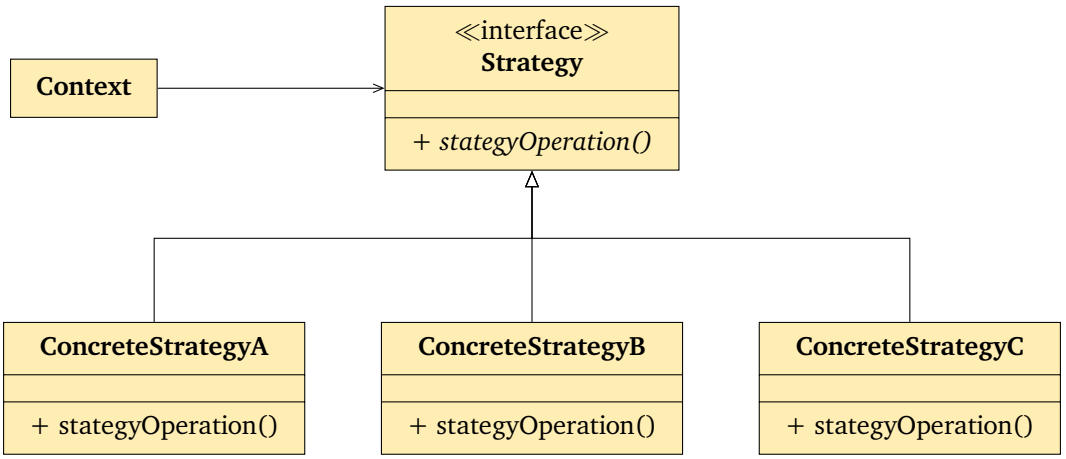


Abbildung 6.2: UML: Stategy Pattern

6.2.3 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

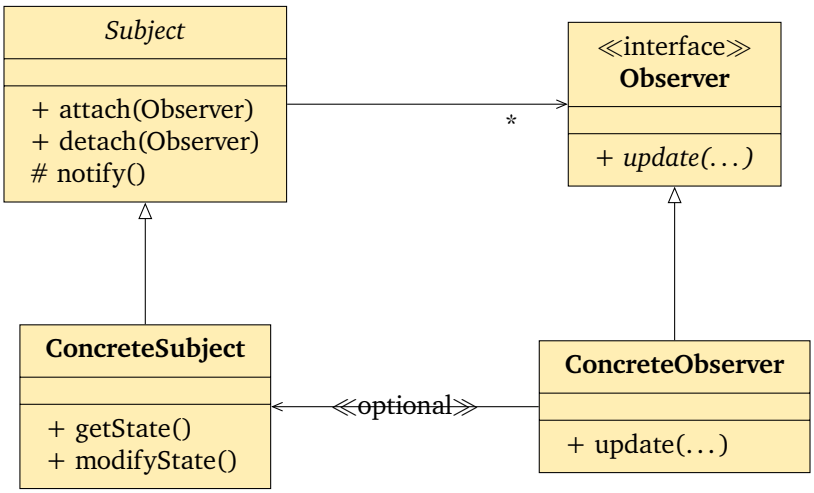


Abbildung 6.3: UML: Observer Pattern

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

6.3 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
  - ...

6.3.1 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

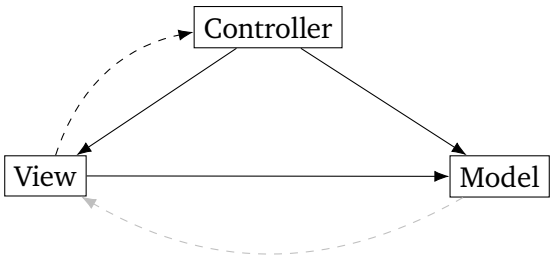


Abbildung 6.4: Model-View-Controller

Controller und View sind direkt gekoppelt mit dem Modell, das Modell ist nicht direkt gekoppelt mit dem Controller oder der View (siehe 6.4).

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.

7 Diagramme

Aktivitätsdiagramm 2.2.1

Gantt Chart 2.2.2

State Machine Diagram (UML) 4.1.2

Class-Responsiblity-Collaborator-Karten 5.1.1

Domain Model 2.6.1

Use Case (UML) 2.5.2

Interaction/Sequence Diagram (UML) 4.1.1

Kontrollflussgraph 3.3.1

8 Glossar

Pflichtenheft Das Dokument, welches die Anforderungen an eine Software enthält. Siehe 2.4.