TECHNISCHE
UNIVERSITÄT
DARMSTADT

Skript zur Vorlesung

Einführung in Software Engineering

Wintersemester 2010 / 2011

Fachbereich Informatik

---

9. Februar 2011

Disclaimer

Wesentliche Teile dieses Skript entstehen durch die gemeinsame Anstrengung der Studierenden und des Veranstalters der Veranstaltung Einführung in Software Engineering. Jeder, der Fehler entdeckt, oder inhaltlich etwas ergänzen möchte, wird dazu aufgerufen, zu diesem Skript beizutragen. Die Beiträge werden vom Veranstalter in unregelmäßigen Abständen auf inhaltliche Korrektheit überprüft. Teile, die inhaltlich korrekt sind, werden in eine vom Veranstalter freigegebene Fassung übernommen. Da die Inhalte im Wesentlichen jedoch von Studierenden erstellt werden, *übernimmt der Veranstalter keine Garantie auf inhaltliche Vollständigkeit.*

Im Wintersemester 2010/2011 werden Beiträge zum Skript von Kathrin Ballweg (ballweg(at)rbg.informatik.tu-darmstadt.de) koordiniert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Vorlesungsüberblick

Die wesentlichen Themen der Vorlesung sind:

- Was ist Software(?) Engineering(?)
- Software-Projektmanagement
- Softwareentwicklungswerkzeuge
- Objektorientierte Analyse und Entwurf
- UML Modellierung

Die inhaltlichen Schwerpunkte der einzelnen Themen, sowie ihre Abhängigkeiten sind in den Abbildungen 1.1, 1.2 und 1.3 dargestellt.

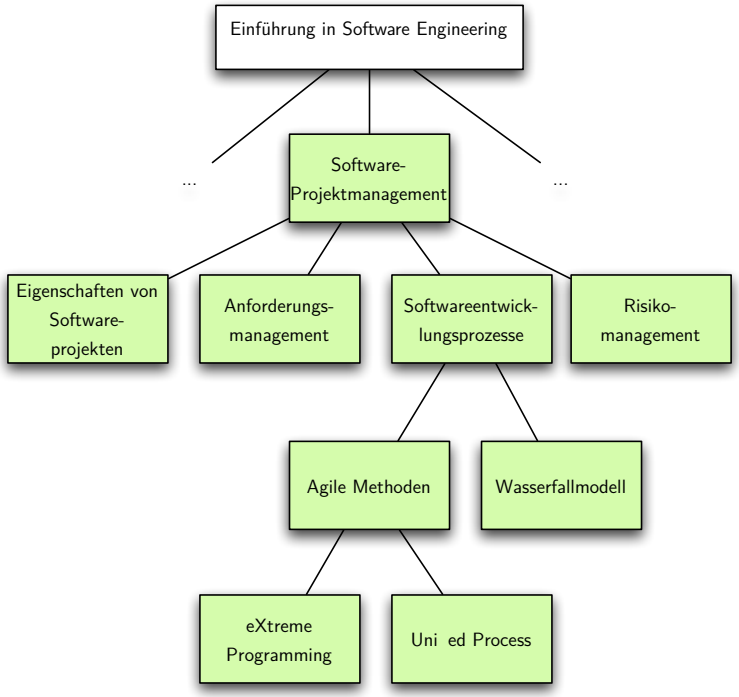


Abbildung 1.1: Grobübersicht der behandelten Themen im Bereich Software-Projektmanagement

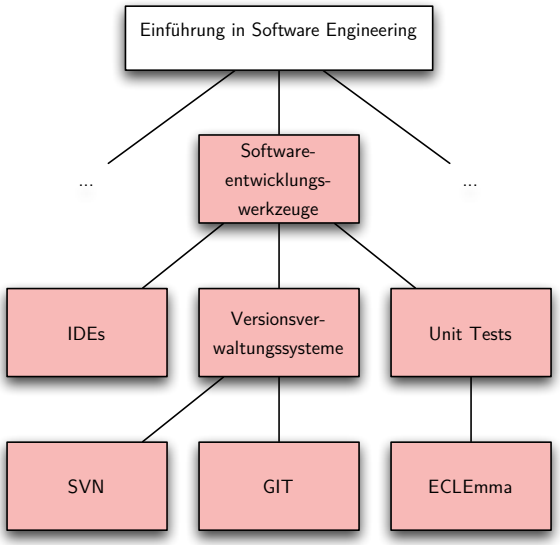


Abbildung 1.2: Grobübersicht der behandelten Themen im Bereich Softwareentwicklungswerkzeuge

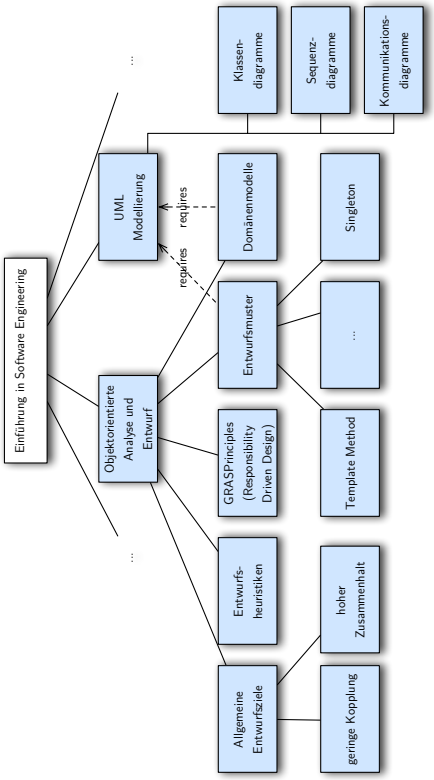


Abbildung 1.3: Grobübersicht der behandelten Themen im Bereich objektorientierte Analyse und Entwurf

Kapitel 2

What is Software Engineering?

- Software:
  - ausführbares Programm, Nutzer-Dokumentation, System-Dokumentation, Website (informieren über Probleme, bereit Stellen von Downloads) , Konfigurationsdateien, ;
  - sowohl als allg. Produkte als auch dem Verbraucher angepasst
  - keine physischen Grenzen
  - keine Ersatzteile
  - langlebig → Updates: dauerhaft aktualisieren, sonst veraltet (→ software aging)
  - Qualität ist schwer zu bewerten
- Engineering:
  - Design, Bau und Gebrauch von Maschinen, Strukturen, Motoren
- Softwarekrise(60er):
  - zu schnell steigende Kosten, unpünktliche Projekte, nicht im Budget und zu viele Fehler, Mangel an geeigneten Programmiersprachen, Mangel an Methoden und Tool-Support
- Gründe für gescheiterte Software Projekte:
  - Anforderungen und Systemabhängigkeiten waren nicht gut definiert
  - Software kann leichter angepasst werden als die Hardware ⇒ Software an Hardware-„Probleme" anpassen
  - Mangel an Werkzeugen, Methoden, Bildung, Planung
  - sich ändernde Anforderungen

- Gebiete von Software Engineering:
  - Software Requirements/Anforderungen
  - Design der Software
  - Testen der Software
  - Wartung der Software
  - Verwaltung der Konfigurationen und Versionen von Software
  - Software Engineering Prozesse
  - Entwicklungswerkzeug und -methoden
  - Qualität der Software
  - Entwicklungsfortschritte der Software
- System Engineering $\leftrightarrow$ Software Engineering
  - Systembezogene Aktivitäten (wie übergreifende System Objekte und Anforderungen, bereitgestellte Systemfunktionen zwischen Hardware und Software, Definieren von Hardware/Software Schnittstellen, Tests für das gesamte System sind essentiell) sind Teil des System Engineering
  - $\Rightarrow$ Software Engineering ist ein Teil von System Engineering
- kritische Sicht:
  - **was nicht gemessen werden kann, kann nicht kontrolliert werden!**

Kapitel 3

Software Quality

- Unterscheidung zwischen internen und externen Qualitätsfaktoren für Software
  - intern: können nur vom Spezialisten erkannt werden
  - extern: sind relevant, denn sie werden vom Nutzer erkannt
- Korrektheit:
  - wenn Software die übertragenen/erforderten Aufgaben ausführt
  - dafür ist eine präzise Beschreibung der Anforderungen notwendig
  - Korrektheit ist bedingt durch tieferliegende Ebenen
- Robustheit:
  - Software reagiert angemessen bei abweichenden Bedingungen
  - charakterisiert was außerhalb der gegebenen Spezifikation passiert
  - ergänzt die Korrektheit
- Erweiterbarkeit:
  - beschreibt, wie leicht/schwer Software angepasst werden kann
  - erforderliche Prinzipien: einfaches Design und Dezentralisierung mit autonomen Modulen
- Wiederverwendbarkeit:
  - Software kann für viele andere Anwendungen noch verwendet werden
- Vereinbarkeit/Kompatibilität:
  - wie leicht Software Elemente miteinander kombiniert werden können

- Beweglichkeit:
  - wie leicht Software in eine andere Hardware bzw. Software Umgebung gebracht werden kann
- Effizienz:
  - möglichst wenige Anforderungen an die Hardware stellen
- Funktionalität:
  - beschreibt den Umfang der Möglichkeiten die ein System bereit stellt
- einfache Handhabung:
  - wie schnell/leicht unterschiedl. Leute die Software benutzen können

⇒ **gute Algorithmen verwenden**
⇒ **neue Features sollen mit bereits existierenden konsistent sein**

- Eigenschaften guter Software:
  - wartbar: kann den Kundenwünschen angepasst werden
  - verlässlich: keine Systemfehler, Fehlertoleranz, Überlebensfähigkeit, reparierbar
  - effizient: keine Systemressourcen verschwenden, Empfindlichkeit, Speicherverbrauch
  - brauchbar: für den angedachten Nutzer

Kapitel 4

Software Projekt Management

- Management Tätigkeiten:
  - Angebotserstellung, Projektplanung, Projektkostenkalkulation, Projektüberwachung, Projektevaluation, Bericht verfassen und Präsentation
- Projektplanung ist ein iterativer Prozess:
  - zu Beginn eine (Fahr)Plan aufstellen mit allen vorhandenen Informationen (Projektplan, Qualitätssicherungsplan, Personalentwicklungsplan, Konfigurations-Management-Plan)
- Teile eines Projekt-Plans:
  1. Einführung
  2. Projekt Organisation
  3. Risiko Analyse
  4. Anforderung an die Hardware & Software Ressourcen
  5. Arbeitsaufteilung
  6. Projekt Zeit/Ablaufplan
  7. Überwachungs- und Rechenschaftsmechanismen verscheidene Möglichkeiten der Illustration
- Streitfragen bei Software Projekten
  - Produkt ist nicht greifbar: kein Fortschritt ersichtlich
  - keine Standard Software Prozesse
  - große Projekte sind oft einmalige Projekte: zudem schnelle techn. Entwicklung

- → noch keine Erfahrungswerte

  - **Sneed's Teufels Quadrat** („Sneed's Devil's Square"):
    Produktivität hängt von Qualität, Quantität, Zeit und Kosten ab

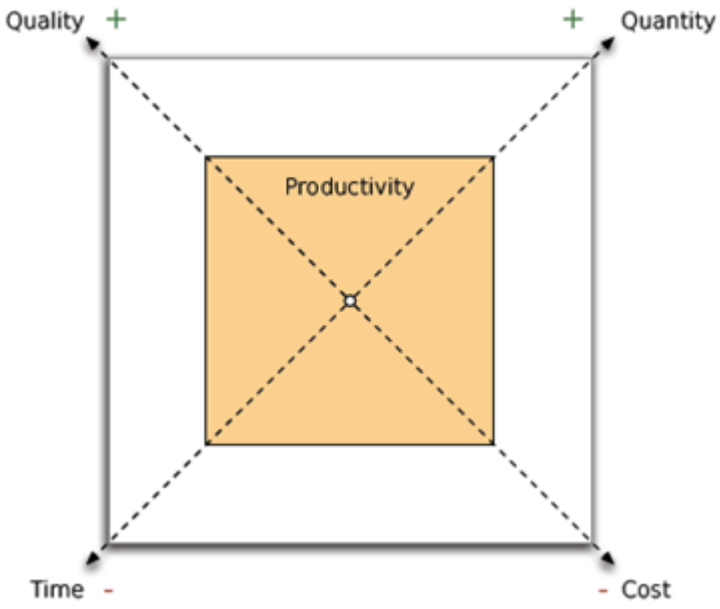


Abbildung 4.1: Teufelsquadrat nach Sneed

  - in der Mitte davon ist die Produktivität(kann auch verschoben/verzerrt werden um die Gewichtung der einzelnen Faktoren zu verändern)

Kapitel 5

Risk Management

- Risiko Typen:
  - Risiken, die von den verwendeten Hardware/Software Technologien ausgehen
  - Risiken, die mit Leuten aus dem Entwicklungsteam verbunden sind
  - Organisationsrisiken
  - „Tools risks", die von den verwendeten CASE-Tools und anderer Software ausgehen
  - Risiken, die durch Änderungen in den Anforderungen der Kunden entstehen
  - Abschätzungsrisiken bzgl der benötigten Ressourcen
  - konkrete Bsp.: Personalveränderungen, Wechsel des Managements, Hardware ist nicht verfügbar, Spezifikationen sind in Verzug, neue Technologien,...

- Risiko Managemenet Prozess

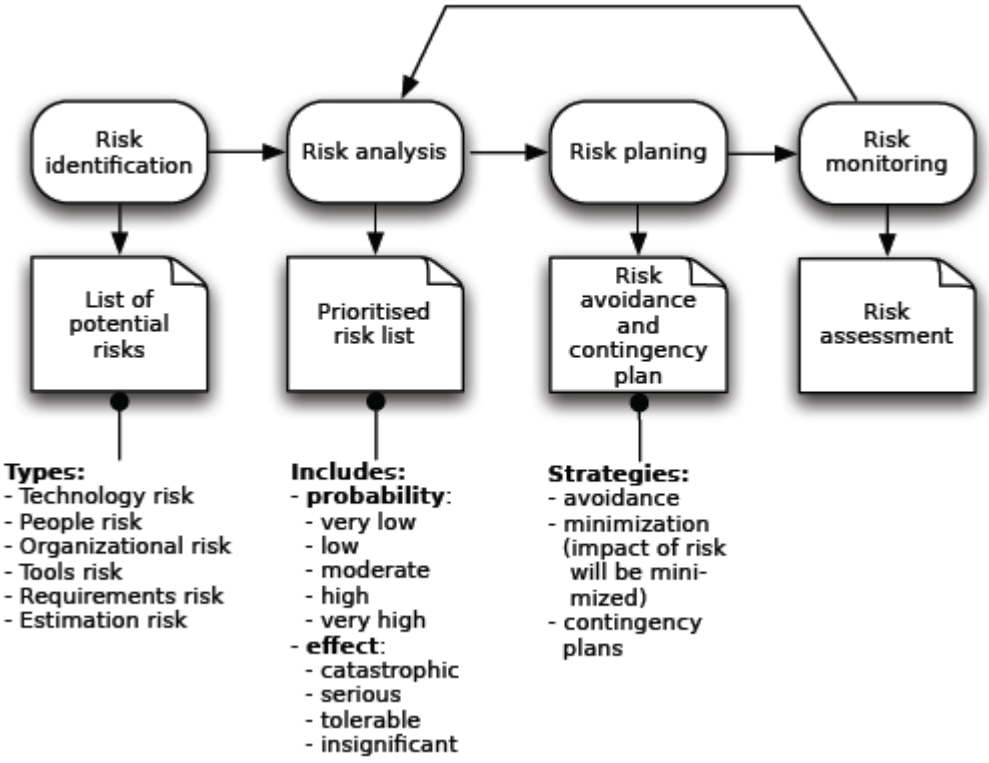


Abbildung 5.1: Ablauf des Risk Managements

Kapitel 6

Computer Aided Software Engineering (CASE)

- CASE tools....
  - ...unterstützen Prozessabläufe wie z.B. Design, Programmentwicklung, Testen und Entwicklung der Anforderungen
  - ...sind z.B.: Compiler, Debugger; integrierte Entwicklungsumgebungen; Design Editor....
- Klassifizierung:
  - Tools: unterstützen die individuellen Prozessaufgaben(z.B. Programm compilieren)
  - Workbenches: unterstützen die Prozessphase (z.B. Spezifikation der Anforderungen o. Design)
  - Environments: fördern erheblich den Software Entwicklungsprozess
- Beispiele: Doors, SVN, Merge Tool and File Comparison, Eclipse IDE, Xcode IDE, Rengineering Tool Code-Crawler
- $\Rightarrow$ aber Verbesserungen/Erleichterung durch CASE-Tools in der Software Entwicklung ist begrenzt.
  Gründe:
  - SE ist ein Entwurfsprozess, der aus kreativen Ideen besteht
  - SE ist Teamarbeit und somit wird viel Zeit benötigt um sich mit anderen Teammitgliedern auszutauschen, was (noch) nicht von CASE-Tools unterstützt wird

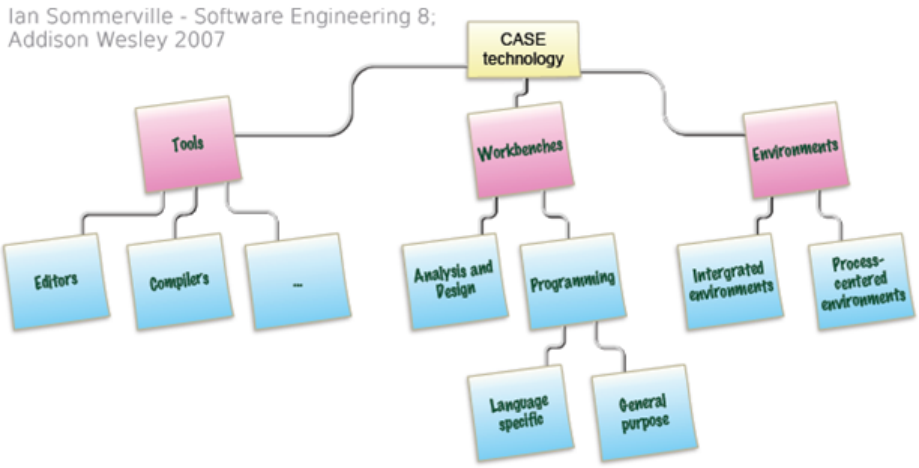


Abbildung 6.1: CASE-Technologien

Kapitel 7

Version Control Systems

7.1 Überblick

Vorab sollen zwei Begriffe, die bei der Arbeit mit Versionskontrollsystemen (VCS) in der Softwareentwicklung wichtig sind, geklärt werden.

- Branch
  Einen Branch kann man sich als das Kopieren der aktuellen Version in ein anderes Verzeichnis vorstellen. An beiden Versionen kann dann unabhängig voneinander weiter gearbeitet werden.

- Merge
  Das Vereinigen zweier unabhängiger Versionen wird als Merge bezeichnet. Das Resultat ist eine neue Version mit zwei Vorgängern.

Eine Auswahl verschiedener VCS

- SCCS[Roc75] (1972)
  - Arbeitet rein auf Dateiebene, ein einzelner Bug-Fix über drei Dateien wird als drei unterschiedliche Änderungen wahrgenommen.
  - Branch ist möglich, jedoch gibt es keine Merge-Unterstützung.

- RCS[Tic85] (1982)
  - Mehrere Änderungen werden als sogenannter Commit zusammengefasst.
  - Branch und Merge werden unterstützt.
  - Es gibt kein zentral verwaltetes Repositorium. Zusammenarbeit geschieht über freigebene Netzlaufwerke in denen die gemeinsamen Dateien liegen.

  - Ausser dem Sperren von Dateien gibt es keine Möglichkeit Konflikte zu vermeiden. Änderungen sind zum Beispiel nicht auf die neueste Version beschränkt.

- CVS[Gru86] (1990)
  - Mehrere Dateien innerhalb eines Commits werden beim Synchronisieren einzeln nacheinander Übertragen. Bricht die Übertragung ab kann es daher zu inkonsistenten Repositorien kommen. (no atomic transactions)
  - Verwendet ein verwaltetes Repositorium und ein Client/Server Model.

- SVN (2000)
  - Wird heutzutage häufig eingesetzt, zum Beispiel in EiSE an der TU-Darmstadt 2010.
  - Verwendet atomic transactions
  - Unter der Oberfläche kommt eine relationale Datenbank (BerkleyDB) zum Einsatz, um das Repositorium zu speichern, was manuelle Eingriffe praktisch sehr schwierig macht.
  - Ist eine von Grund auf neu geschriebene CVS Variante. (slogan: CVS done right)

- BitKeeper (1998 Distributed and Closed Source)
  - Eines der ersten VCS mit intelligenten Merges, basierend auf einer Commit-Historie.

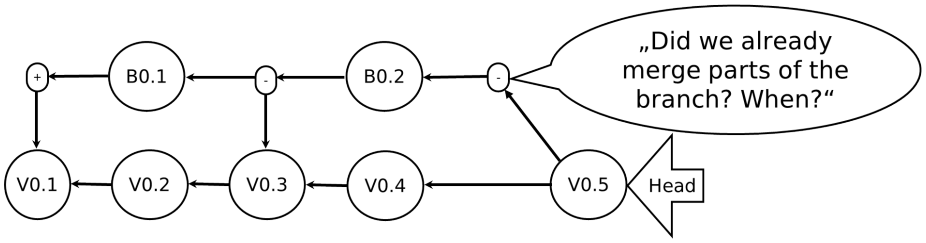


Abbildung 7.1: Historie mit mehreren Merges des selben Branch

  - Die Community Edition wurde für die Entwicklung des Linux Kernels von 2002 bis 2005, als die Lizenz geändert wurde, eingesetzt.

- Git[tea05] (2005)
  - Legt den Focus auf Inhalt und weg von Dateien. Dadurch soll es möglich sein Funktionen/Methoden, die zwischen verschiedenen Dateien verschoben wurden, zu verfolgen.

  - Verwendet eine Staging Area, um den Inhalt eines commits festzulegen. Dadurch ist es z.B. möglich nur Änderungen eines bestimmten Aspektes in einem Commit zu vereinigen.
  - Gits Branches sind explizit. Man muss manuell einen Branch erstellen wenn man nicht direkt auf der aktuellen Version (Head-Version) arbeiten möchte.
  - Designed von Linus Torvalds für die Linux Kernel Entwicklung

- Mercurial (2005)
  - Implizite Branches (jede Kopie/Klon ist automatisch ein Branch)
  - Es wird oft gesagt, dass es im Vergleich zu Git leichter zu erlernen ist.
  - Verwendet keine Staging Area, was die Anzahl der nötigen Befehle verringert.

7.2 Verteilte und Zentralisierte Versionskontrolle im Vergleich

Zentralisierte Versionskontrollsysteme (CVCS) setzen auf einen gemeinsamen Server (Abbildung 7.2), über den der Quellcode aller Klienten synchronisiert wird.

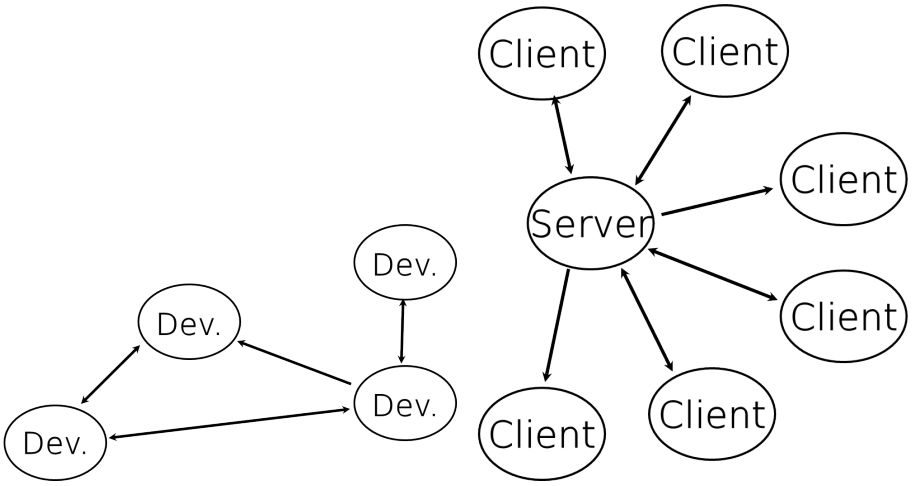


Abbildung 7.2: Verteilt unter Gleichen

Abbildung 7.3: Zentralisiert

Verteilte Systeme (DVCS) dagegen ermöglichen beliebigen Austausch unter den Entwicklern (Abbildung 7.3). Bei grossen Projekten gibt es jedoch oft einen

Projektleiter, der mit Hilfe von ihm bekannten Entwicklern entscheidet, welche Änderungen der Community in die neue Version übernommen werden.
Die neue Version wird dieser als Head-Version zur Verfügung gestellt. (Abbildung 7.4)

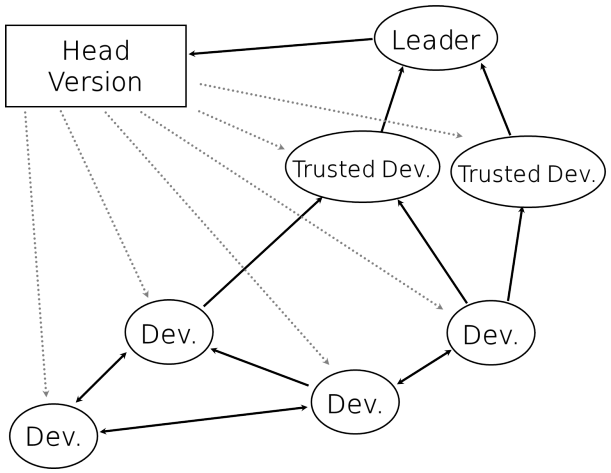


Abbildung 7.4: Verteiltes Model mit Projektleiter

Vergleich zweier etablierter Systeme

Name	Git	SVN
Typ	verteilt (distributed)	zentralisiert (centralized)
Position des Repositoriums	Lokal	auf dem Server
Optionen um auf mehrere Repositorien zuzugreifen	E-Mail, Team-Server, USB-Stick, etc...	Keine
Branch, merge, commit, diff, browse history	Lokal verfügbar	benötigen Server-Zugang
Identifikation von Commits	160 bit Hash-Code	Global geordnete, aufsteigende Integer
Binäre Dateien (z.B. Bilder)	Alle Dateien werden als binär betrachtet, Text-Dateien werden optional automatisch erkannt	alle Dateien werden als Text behandelt, optionale Erkennung von binären Dateien
Umbenennen von Dateien	Wird automatisch erkannt und als Löschen und neu Erstellen behandelt	möglich, aber das System erkennt es nur korrekt bei Verwendung des SVN-Werkzeugs

7.3 Typische Arbeitsschritte

Die Arbeit mit zentralisierten Versionskontrollsystemen (CVCS) bietet relativ wenige Optionen (Abbildung 7.5). Viele CVCS bieten zwar die Möglichkeit für Branch und Merge, jedoch werden diese selten verwendet, da ein Merge häufig sehr aufwendig ist, was an den vorhandenen Informationen innerhalb der Historie der verwendeten Systeme liegt. Im Repository befindet sich zu jedem Zeitpunkt die aktuelle Version, die immer lauffähig sein muss, da man ansonsten Mitarbeiter behindern könnte.
In Firmen gibt es häufig Regelungen, die verlangen, dass erst alle Tests erfolgreich durchlaufen werden müssen, bevor der Code eingecheckt werden darf.

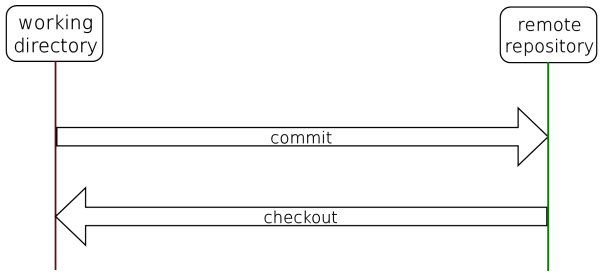


Abbildung 7.5: CVCS Workflow & Commands[Qua09]

Verteilte Versionskontrollsysteme (DVCS) führen ein lokales Repositorium ein (Abbildung 7.6) in das man beliebig häufig einchecken kann, ohne dass man Angst haben muss, dass ein Kollege nicht mehr weiterarbeiten kann. Man kann jederzeit zu vergangenen Versionen zurückkehren. Das lokale Repositorium kann man mit Kollegen oder einem definierten zentralem Server abgleichen.

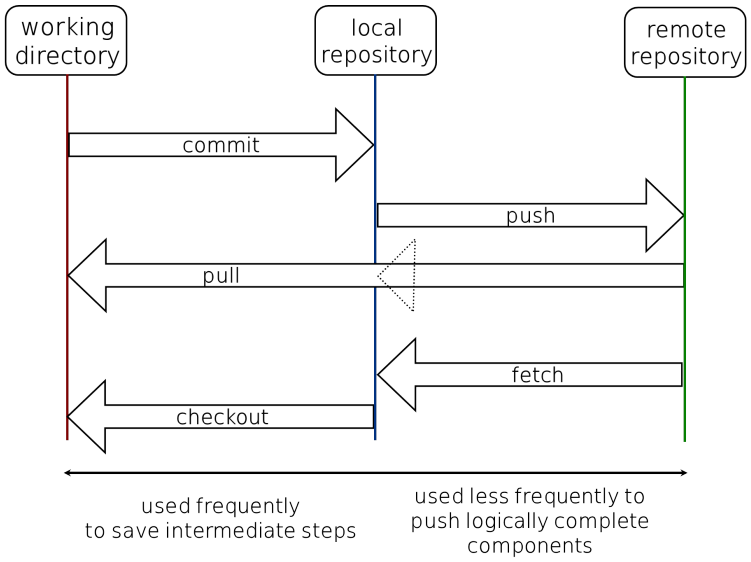


Abbildung 7.6: DVCS Workflow & Commands[Qua09]

Git führt zusätzlich eine Staging Area ein (Abbildung 7.7). Sie ermöglicht Commits vor dem eigentlichen Commit anzupassen, in dem man zum Beispiel nur Änderungen eines bestimmten Aspektes hinzufügt. Sie reduziert ausserdem die Anzahl der Commits und so die Grösse der Historie, was wiederum deren Übersichtlichkeit steigert.

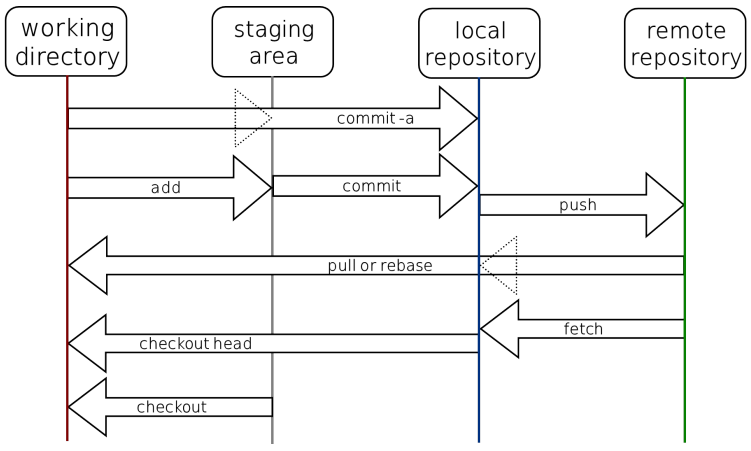
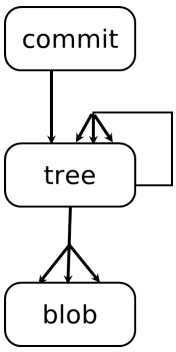


Abbildung 7.7: Git Specific Workflow & Commands[Qua09]

7.4 Git Arbeitsablauf[Dua, tea05]

Im folgenden werden Git spezifische Begriffe erklärt.



- Das .git Verzeichnis enthält
  - Commit-Objects
  - Referenzen zu Commits (heads)
- Ein Commit-Objects besteht aus
  - Einer Momentaufnahme des Projekts zum entsprechendem Zeitpunkt
    - Trees (Unterverzeichnisse)
    - Blobs (Dateien)
  - Einer Referenz zum vorhergehenden Commit-Objects
  - 40 Charakter SHA1 Hashes zu Identifikation von Dateien. (Identische Dateien haben also immer den selben Hashwert)
- Tags
  - Ein von Menschen lesbarer Name für besondere Commits

Durch die Referenzen von Commit-Objects auf deren Vorgänger lässt sich eine Historie konstruieren, die einen gerichteten azyklischen Graphen (DAG) bildet. Auf den folgenden Seiten werden solche Graphen verwendet, um anschaulich die Arbeit mit Git zu demonstrieren.

In diesem Beispiel wird an einer Flashcards Applikation gearbeitet.
Unser letzter stabiler, sauberer Commit ist die Version 0.2, während Version 0.3 eine instabile Testversion darstellt.
Wie man in Abbildung 7.8 leicht erkennen kann ist Version 0.3 ein direkter Nachfolger von Version 0.2.
Bob, ein freundlicher Mitarbeiter, soll die grafische Oberfläche (GUI) für unsere Applikation bauen. Natürlich wollen wir nicht, dass Bob mit unserer instabilen Version arbeiten muss.

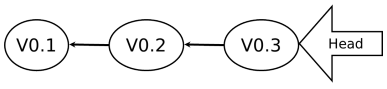


Abbildung 7.8: Commit DAG 1

Daher kehren wir zur Version 0.2 zurück, erstellen einen Branch, erweitern unseren Code um einige Kommentare für Bob, wodurch Commit Br1-0 entsteht und senden Bob das Repositorium (Abbildung 7.9). In diesem ist allerdings unsere instabile Version 0.3 für Bob sichtbar, was möglicherweise zu Missverständnissen führen könnte. Um das zu vermeiden hätten wir für uns einen eigenen Branch erstellen sollen, um unseren primären Branch (Head-Version) sauber zu halten.

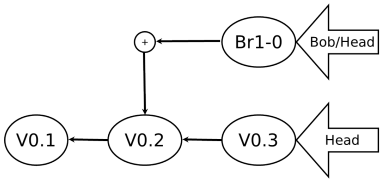


Abbildung 7.9: Commit DAG 2

Wir erzeugen also einen Branch für uns und nach getaner Arbeit entsteht Commit Br2. Bob arbeitet währenddessen an seinem Branch und es entsteht Version Br1-1. (Abbildung 7.10)

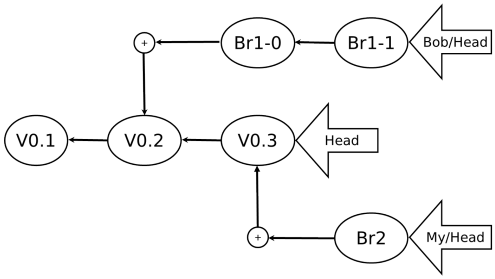


Abbildung 7.10: Commit DAG 3

Da wir mit Version Br2 sehr zufrieden sind, wollen wir sie mit dem primären Branch zusammenfügen (merge). Da am primären Branch jedoch nicht weitergearbeitet wurde und V0.3 ein direkter Vorgänger von Br2 ist, wird ein sogenannter forward merge durchgeführt und es entsteht der Commit-Tree aus Abbildung 7.11

Bob ist mittlerweile mit der GUI fertig und merged daher Br1-1 mit der Head-Version, wodurch die finale Version 0.5 entsteht (Abbildung 7.12). Git merkt sich die komplette Historie (history) und trifft mit ihrer Hilfe Entscheidungen bei Merge-Vorgängen. Dies ist insbesondere dann relevant, wenn ein Branch mit

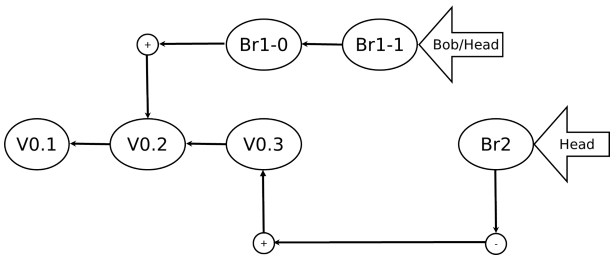


Abbildung 7.11: Commit DAG 4

der Head-Version vereinigt wird, der bereits in einer früheren Version schon einmal vereinigt wurde. Nur dadurch, dass Git die Vergangenheit kennt, weiss die Versionskontrolle, welche Teile in der neuen Version des Branch hinzugekommen sind. (Siehe auch Abbildung 7.1)

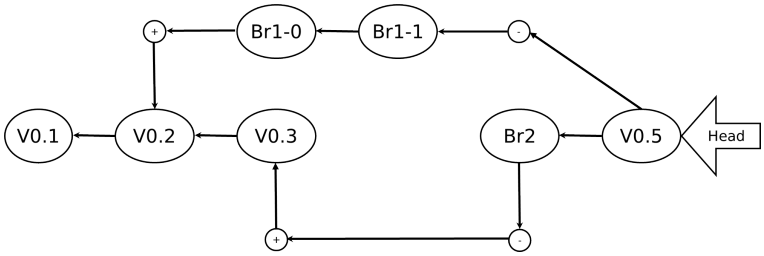


Abbildung 7.12: Final Commit DAG

Es soll zum Abschluss noch eine andere Art der Zusammenführung von Versionen erwähnt werden, rebase. Nachdem Bob seine GUI fertig hat und feststellt, dass sie wunderbar mit unserer Head-Version BR2 harmoniert, kann er die Head-Version als neue Basis für seine Version Br1-1 verwenden. Die Information, dass die GUI eigentlich für V0.2 entwickelt wurde geht dabei verloren, allerdings wird die Historie natürlich Übersichtlicher (Abbildung 7.13).
Rebasing kann verwirrend sein, wenn Leute die Versionen und Branches kannten, bevor Rebasing zum Einsatz kam.
Man sollte in jedem Fall die Commit-Historie im Auge behalten, um zu verstehen was man tatsächlich bewirkt.

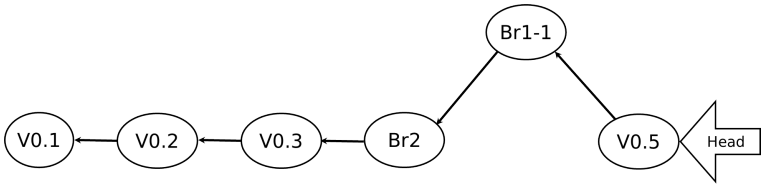


Abbildung 7.13: Rebasing

Kapitel 8

Software Engineering Processes

8.1 Software Process Models - A First Glimpse

- fundamentale Prozesse um Software herzustellen: Spezifikation Entwicklung, Validierung und Weiterentwicklung(Evolution)
- Modelle sind eine vereinfachte und abstrahierte Beschreibung eines Software Prozesses, das eine Sicht von diesem Prozess aufzeigt
- Wasserfall-Modell:
  - Analyse der Anforderungen und Definition der Spezifikationen
  - Design von System und Software(Abstraktionen)
  - Implementierung und Testen der einzelnen Elemente
  - Integration/Verbinden der Elemente und Testen des Systems
  - Operation und Wartbarkeit
- Phasen werden linear nacheinander durchlaufen, wenn Fehler auftreten, werden die vorherigen Phasen durchlaufen, bis diese fehlerfrei sind
- unterschiedliche Entwicklungs-Prozess-Modelle notwendig fr unterschiedliche Software Systeme, es gibt keinen idealen Prozess

8.2 Software Engineering Processes - Overview

- allg. Prozess Modelle:

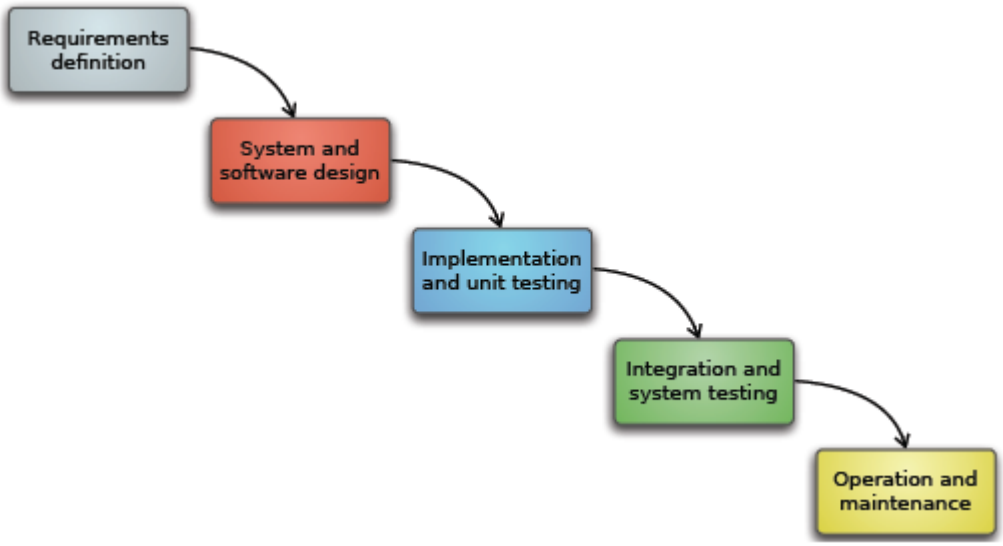


Abbildung 8.1: Wasserfall-Model

  - evolutionary Development: erste Version wird dem Nutzer gezeigt und sein Feedback wird genutzt um das System nach und nach anzupassen/ zu verbessern
  - component-based SE:
    - Bestandteilanalyse: Komponenten, die die Anforderungen implmentieren werden gesucht
    - Anpassung der Anforderungen: um die verfügbaren Komponenten zu verwenden
    - Systemdesign mit Wiederverwendung: bestehendes Framework wird verwendet oder so designt, dass es wiederverwendet werden kann
    - Entwicklung und Einpassung

- agile Entwicklung:
  - Ziel: SW schnell entwickeln angesichts sich schnell ändernder Anforderungen
  - um Agilität zu erreichen:
    - Praktiken anwenden, die die nötige Disziplin haben
    - Design Prinzipien anwenden, die die SW flexibel und wartbar halten
    - Kenntnis von Design Pattern für spezielle Probleme
  - Grundsätze für agiles SE

    - Zusammenarbeit im Team ist wichtig, sonst helfen die besten Tools nichts
    - Struktur des Systems und die Argumentation dokumentieren
    - Vertrag soll festhalten wie Zusammenarbeit zw Kunde und Entwicklern aussieht
    - Evtl. den verfolgten Plan anpassen/ändern
  - Prinzipien:
    - Kunden zufriedenstellen früh und regelmäßig abgabefähige SW zeigen
    - auf gutes Design und techn Exzellenz achten
    - Schlichtheit
    - von Zeit zu Zeit über effizientere Arbeitsweise nachdenken
    - jeder einzelne muss motiviert sein
    - nachhaltige Entwicklung

- vereinheitlichte Prozesse (unified process)
  - Phasen:
    1. Anfang: nötige Nachforschungen
    2. Ausarbeitung: Kern Architektur wurde iterativ implementiert; gr. Risiken wurden abgeschwächt
    3. Konstruktion: iterative Implementierung von einfachen Elementen und Vorbereitung auf die Verwendung
    4. Überleitung: Beta Tests
  - gr. Risiko vermeiden; kontinuierliche Evaluation/Feedback & Verifizierung/Testen; Use Cases anwenden; visuelle Modellierung

8.3 Case Study

Momentan kein Inhalt vorhanden

8.4 Software Engineering Processes - Extreme Programming

- Customer Team Member: Kunden äußern ihre Wünsche und sind ein Teil des Teams
- User Stories:
  kurze Stichpunkte zu den Anforderungen
- Short Cycles:
  SW wird in regelmäßigen, festgelegten Abständen ausgeliefert, ggf. Aufgaben anpassen, wenn sie nicht in geg. Zeitrahmen erledigt werden können

- Iteration Plan:
  Plan für jeden Schritt im Zyklus(short cycles), Prioritäten und User Stories festlegen, Kundenwünsche berücksichtigen, Beschränkungen durch das Budget

- Release Plan:
  Plan für i.d.R. 6 Iterationen

- Planning game:
  Aufteilen der Verantwortlichkeiten zw Business(welche Funktionen sind wichtig) und Entwicklung(wie viel diese Fkt wird es kosten)

- acceptance test:
  Black-Box Tests vom Kunden, vor/während der Implementierung geschrieben, ein einmal bestandener Test muss danach immer bestanden werden

- Pair Programming:
  zu zweit programmieren, einer tippt, der andere unterstützt ihn, abwechselnd

- collective ownership:
  der Code gehört dem Team, jeder hat Zugriff auf alles

- Test-Driven Development:
  white-box Tests von Entwicklern, ermöglicht Refactoring und führt zu niedriger Kopplung

- continous Integration:
  Programmierer kontrollieren ihren Code, Änderungen mehrmals am Tag

- Refactoring:
  kontinuierliches überarbeiten/verbessern, ohne dabei die Funktionalität zu verändern

- Sustainable pace:
  nicht zu viele Überstunden(Ausnahme letzte Woche)

- open workspace:
  team arbeitet in einem offenen Raum zusammen

- simple Design:
  Design möglichst einfach halten, auf vorhandene User Stories konzentrieren(nichts unnötiges, bis jetzt nicht gefordertes hinzufügen), keine Redundanzen

- **Vorgehensweise in der Praxis**

  - Start: wichtige User Stories herausfiltern, Story Points (SP) abschätzen, Velocity(benötigte zeit pro SP)

- release Planning: erstes Veröffentlichungsdatum vereinbaren, Kunde wählt Stories unter Einhaltung der velocity, Plan entsprechend Velocity anpassen
- Iteration planning: Kunde wählt die Stories für die Iteration aus, Entwickler entscheidet über Ordnung innerhalb der Iteration, Ende der Iteration festgelegt (auch wenn noch nicht alle Punkte abgearbeitet), neue velocity (vorherigen Wert) berechnen
- Task Planning: zu Beginn jeder Iteration Stories unterteilen (ca 4-16 std) und Aufgaben aufteilen

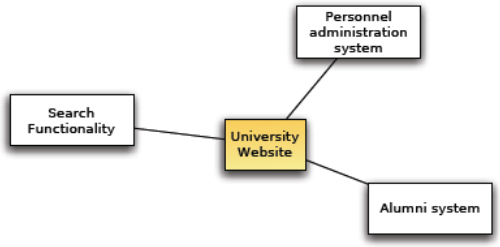
8.5 Software Engineering Processes - Aftermath

Momentan kein Inhalt vorhanden

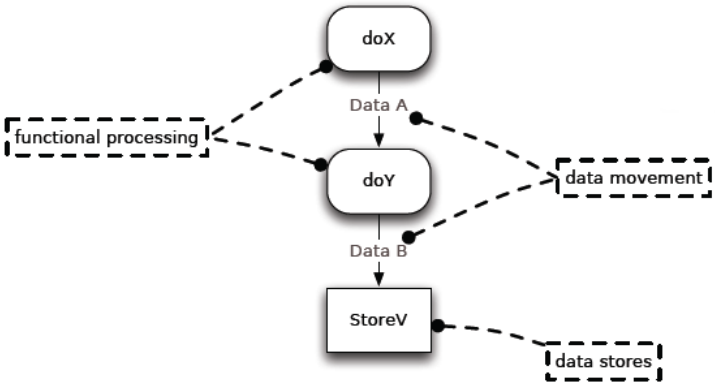
Kapitel 9

System Models

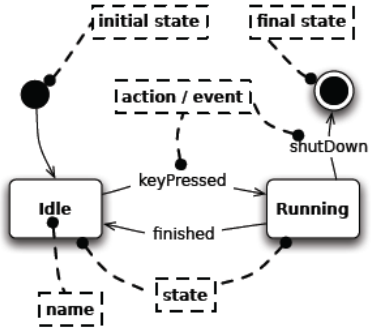
- High Level Architekturmodelle:
  um die Grenzen d. Systems zu verdeutlichen



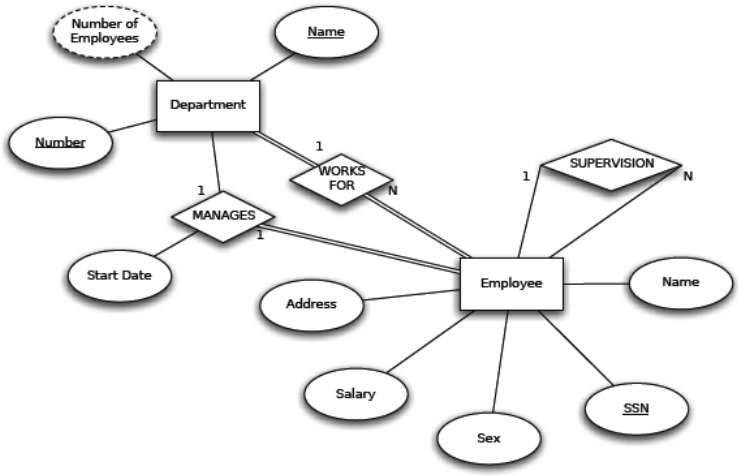
- Data-Flow Model:
  wenn Datenflüsse im Vordergrund stehen



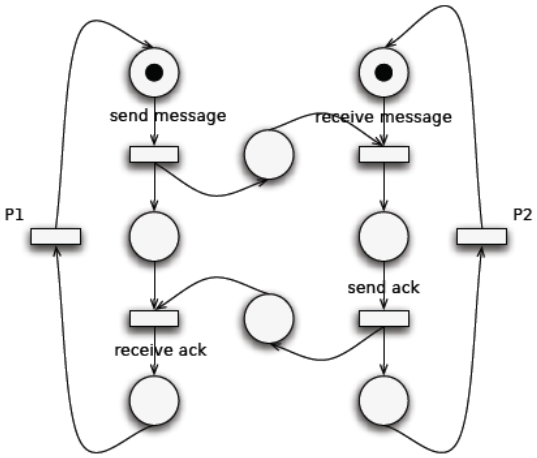
- State Machine Model:
  beschreibt, wie das System auf interne und externe Ereignisse reagiert



- EntityRelationship Model:
  Design von high-level conceptual data model



- Petrinetze:
  formale, graphische, ausführbare Technik zur Spezifikation und Analyse von parallelablaufenden, ereignisdiskreten dynamischen Systeme



Kapitel 10

Object-oriented Thinking

- Class-Responsibility-Collaboration Cards (CRC)
  - Klasse: Name e. Objekts, guter Name ist wichtig
  - Responsibilities/Verantwortlichkeit identifiziert das zu lösende Problem
  - Collaborators sind Objekte, die Nachrichten senden oder empfangen
- Beachten:
  - nicht zu viele Verantwortlichkeiten/Collaborations → besser neue Klasse
  - nicht zu unterschiedliche Verantwortlichkeiten
  - zyklische Abhängigkeiten vermeiden → Abstraktion
- Bsp. in VL: Darstellung von versch. geometr. Figuren

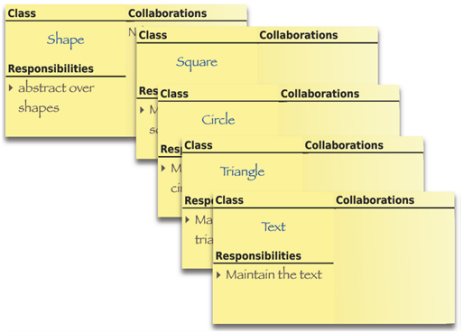


Abbildung 10.1: Beispiel für CRC-Cards

- Design-Richtlinien: niedrige Kopplung, hohe Kohäsion, keine zyklischen Abhängigkeiten, Prinzip der Einzelverantwortlichkeiten(„single responsibility Prinzip") (s. Design Heuristiken und Design Patterns später in der VL)

Kapitel 11

Domain Model and Domain Modelling

- conceptual classes:
  - Dinge, Ideen o. Objekte in einer Domäne
  - Diese Klasse hat ein Symbol, das die Klasse repräsentiert, eine Absicht und Erweiterung, die an Beispielen definiert zu was die conceptual class gehört (keine SW objekte wie zB in Java, C#, .NET!)
- Schritte:
  - conceptual class finden(bestehendes Modell übernehmen/abändern; Nomen in textueller Beschreibung)
  - UML Klassen Diagramm zeichnen
  - Assoziationen/Attribute hinzufügen

- Beispiel:

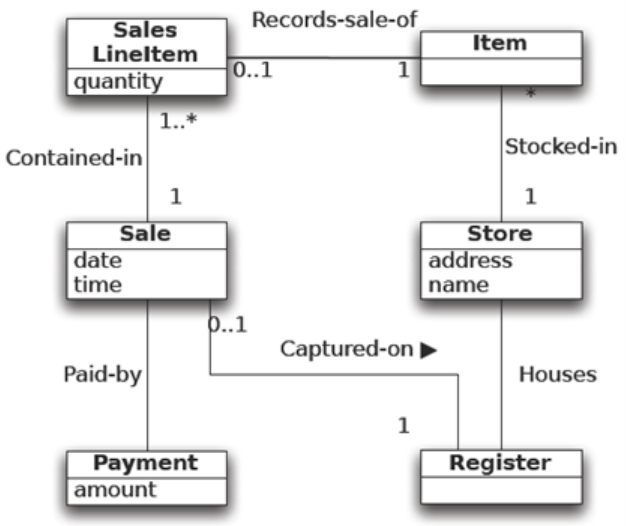


Abbildung 11.1: beispielhaftes Domain Model

- Wenn wir in der Realität X nicht als Nummer oder Test betrachten, dann ist X wahrscheinlich eine konzeptuelle Klasse und kein Attribut
- Assoziation hinzufügen, wenn das Wissen aus der Beziehung für eine gewissen Zeitraum benötigt wird (Abhängigkeit)

Kapitel 12

Software Testing & Unit Tests

- **Validierung**: Fertigen wir das richtige Produkt?
- **Verifikation**: Fertigen wir das Produkt richtig?
- 2 Möglichkeiten:
  - SW Inspektion/Prüfung oder ebenbürtige Betrachtung (peer review) → statische Technik
    - Betrachtung des Programms:
      Fehler, minderwertiger Code, Abweichung von Standards
    - automatische Quellcodeanalyse:
      Kontrollflussanalyse, Datenverwendungs-/Datenflussanalyse, Informationsflussanalyse, Pfadanalyse
    - formale Verifikation
    - SW Inspektion liefert keinen Beweis, ob die SW nützlich ist
  - SW Testen → dynamische Technik
    - Validierungs Tests: um dem Kunden zu zeigen, dass das die SW ist, die er wollte
    - Fehlertesten: um Fehler aufzudecken
      * Fehlerorientiert:
        Ziel ist es Fehler aufzudecken
      * Fähigkeitenorientiert:
        ist es zu zeigen, dass die SW die geforderten Fähigkeiten hat
- Testen ist teuer, teilweise das halbe Budget
- unterschiedliche Tests:
  - Unit-Test: überprüft einen relativ kleinen ausführbaren Teil

  - Integrationstest: komplettes Untersystem mit Interfaces um zu zeigen, dass diese zusammen korrekt funktionieren
  - Systemtest: die komplette Anwendung, in Bezug auf Funktionalität, Performanz, Belastbarkeit

- Design eines Tests:

  1. erstens Verantwortungen des Systems identifizieren, modellieren, analysieren
     → SUT **(System under Test)**
  2. zweitens Designe Testfälle aus der externen Sichtweise
  3. drittens Füge Testfälle aufgrund von Codeanalyse, Verdacht, Heuristiken hinzu
  4. viertens Entwickle die erwarteten Ergebnisse(evtl. Annäherung) für jeden Testfall

- IUT **(Implementation under Test)**:

  1. erstens Test-Suite ausführen, Ergebnisse evaluieren
  2. zweitens Coverage Tool verwenden → erreichtes Coverage evaluieren
  3. drittens evtl. zusätzliche Testfälle schreiben (um Coverage zu erreichen)
  4. viertens Aufhören, wenn gewünschtes Coverage erreicht ist und die Testfälle durchlaufen

- Begriffe:

  - Test Point: ein spezieller Wert für die Eingabewerte des Testfalls oder eine Zustandsvariable
  - Test Case: testet Eingabewerte/ Bedingungen, erwarteten Ergebnisse, IUT
  - Test Suite: Sammlung von Test Cases
  - Test Run: Ausführung eines Tests; ist bestanden, wenn erwartetes und erreichtes Ergebnis übereinstimmen
  - Test Driver: eine Klasse oder nützliches Programm, dass Testfälle an einer IUT anwendet
  - Test Harness: Ein System aus test drivern und anderen Tools, die die Ausführung unterstützen
  - Stub: teilweise, temporäre Implementierung einer Komponente (z.B. Platzhalter) für eine unvollständige Komponente)
  - Failures, Errors, Bugs: failure ist die Unfähigkeit eines Systems geforderte Funktionen nicht ausführen zu können unter bestimmten Bedingungen; software failure ist fehlender oder inkorrekter Code; Error ist eine menschl. Fehler, der einen SW Mangel verursacht; bug ist ein Fehler oder Mangel

  - Test Plan: beschreibt die Vorgehensweise für die Tests

- es gibt unendl. viel Tests → Überlegen wo Fehler sein können
- keine Garantie, das SW fehlerfrei ist
- Fault Modelle:
  Conformance-directed testing & fault-directed testing
- Fault sensivity:
  Fähigkeit des Codes Fehler zu verstecken
- Coverage:
  - Statement Coverage:
    alle Anweisungen wurden mind. einmal ausgeführt
  - Branch Coverage (aka Decision Coverage):
    jeder Pfad von einem Knoten wurde mind. einmal ausgeführt; verbundene Bedingungen müssen beide einzeln erfüllt sein (z.B. or im if)

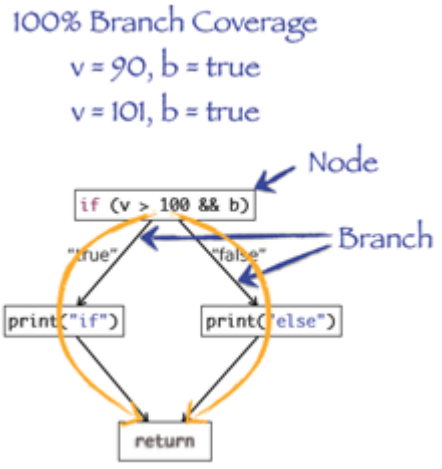


Abbildung 12.1: Beispiel für 100% Branch Coverage

  - Condition Coverage:
    jede Bedingung muss einmal zu true und false ausgewertet werden(es müssen nicht alle möglichen Wege abgedeckt werden) (a OR b → 2 Tests)
  - Multiple-Condition Coverage:
    alle true-false Kombinationen müssen mind. einmal ausgeführt werden(a OR b → 4 Tests)

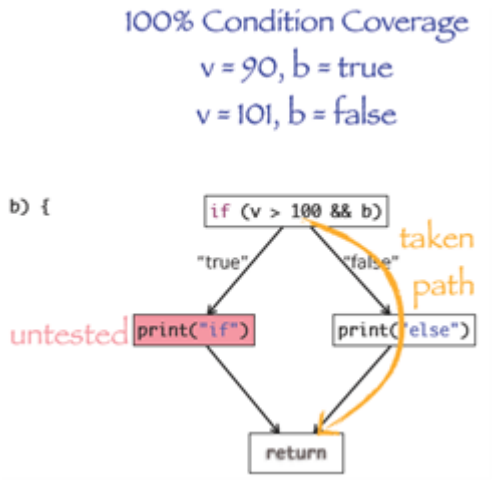


Abbildung 12.2: Beispiel für 100% Condition Coverage

- Basic Block Coverage:
  wenn alle Basic Blöcke(Folge von aufeinanderfolgenden Befehlen) ausgeführt wurden

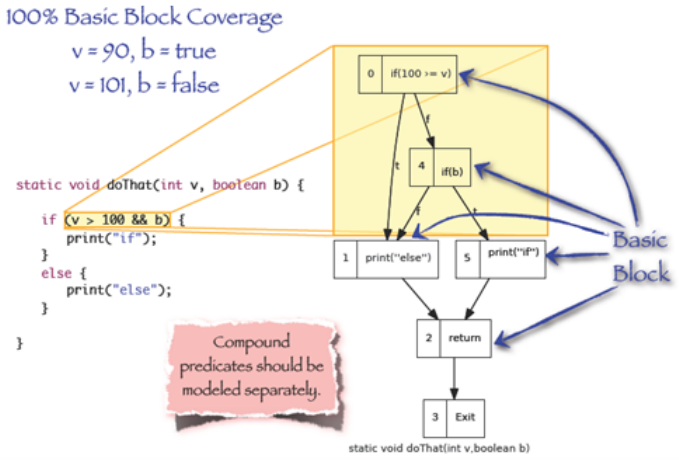


Abbildung 12.3: Beispiel für 100% Basic Block Coverage

Kapitel 13

Requirements Engineering

- Anforderungen sind die Beschreibungen von Diensten, die durch das System sichergestellt werden und funktionelle Beschränkungen
- Arten:
  - Nutzeranforderungen → Lastenheft
    - welche Funktionen sind nötig und unter welchen Bedingungen
    - vom Kunden verfassst
  - Systemanforderungen → Pflichtenheft
    - enthält die Funktionen, Einsatzbedingungen und Dienste
    - sollte präzise sein
    - vom SW Entwickler verfasst
- Anforderungen aus Sicht des Entwicklers:
  - funktionale Anforderungen
    - spezifizieren die Funktionen, die das System zur Verfügung stellen soll, wie das System auf bestimmte Eingaben reagieren soll und wie es sich in bestimmten Situationen verhalten soll
  - nicht-funktionale Anforderungen
    - z.B. ist die Benutzeroberfläche einfach zu bedienen
    - Ein-/Beschränkungen an die Funktionen/Dienste, die vom System bereit gestellt werden, bzgl. Zeit, Entwicklungsprozess,
    - finden oft auf das gesamte System Anwendung
  - die Grenzen zw. fkt. und nicht-fkt. Anforderungen sind oft nicht klar
  - nicht-fkt. Anforderungen sind i.d.R. kritischer als individuelle fkt. Anforderungen
- Anforderungsbereiche kommen eher von den Bedürfnissen des Nutzers als von den Anwendungsgebiet des Systems

- normal mit Fachbegriffen → schwer für SE'ler zu verstehen
- können u.U. nicht explizit erklärt sein

- Pflichtenheft:
  - unterschiedl. Nutzer: Manager, Auftraggeber, Test-/Wartungsleute
  - Genauigkeit hängt von der Art d. Systems, Entwicklungsfortschritt, extern/interner Auftraggeber
  - IEEE/ANSI 830/1998 Standard:

1) Einführung a) Absicht des Plichtenheftes b) Anwendungsbereich des Produkts c) Definitionen, Abkürzungen d) Referenzen e) Übersicht

allgemeine Beschreibung a) Produktperspektive b) Funktion des Produkts c) Nutzercharakteristik d) allgemeine Beschränkungen e) Annahmen und Abhängigkeiten 3) spezifische Anforderungen 4) Anhang/Anlagen 5) Verzeichnis

- einzelne Prozessschritte der Anforderungsanalyse:
  - Herausfinden
  - Analyse
  - Dokumentation
  - Kontrolle der Funktionen und Beschränkungen
  - Ziel: Pflichtenheft

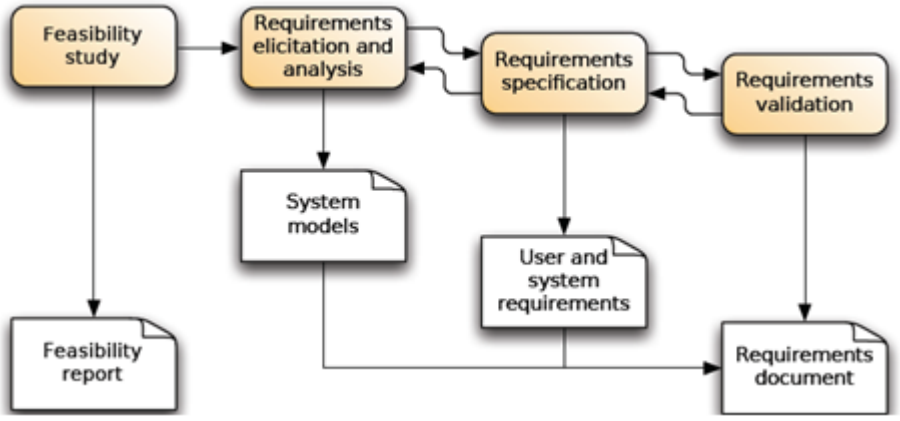


Abbildung 13.1: Anforderungsanalyse – Übersicht

- Durchführbarkeitsbericht, der aussagt ob es lohnenswert/sinnvoll ist, die Anforderungsanalyse und den Systementwicklungsprozess fort zu führen
  - Input:
    - bestehende Geschäftsanforderungen
    - Rahmenbeschreibung des Systems, wie das System den Geschäftsablauf unterstützen soll
  - zu beantwortenden Fragen:
    - Kann das System mit den momentan vorhandenen Technologien und mit den veranschlagten Kosten sowie zeitl. Beschränkungen erfüllt werden?
    - Ist das System in die bereits bestehenden Systeme integrierbar?
    - Trägt das System zu dem allgemeinen Ziel der Organisation bei?
- Anforderungsermittlung:

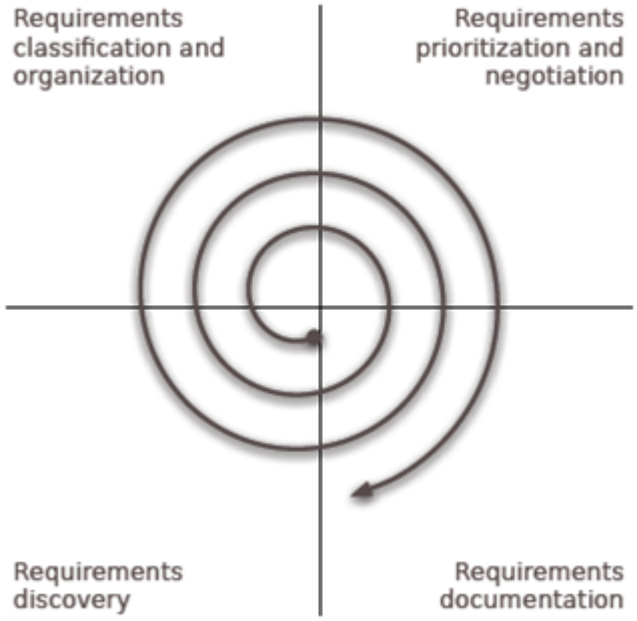


Abbildung 13.2: Anforderungsanalyse – Requirements Engineering

- zum 1. Quadrant
  - größtes Problem: Wie kann man die Anforderungen systematisch herausfinden ?
  - → standpunktorientierte Annäherung

  - gewöhnliche Standpunkte(viewpoints):
    - Interagierer: Personen, die mit dem System umgehen/arbeiten/-nutzen werden
    - indirekte Standpunkt: Interessensvertreter/Akteur der die Anforderungen beeinflusst, aber später nicht das System nutzen wird
    - fachspezifischer Standpunkt: Fachspezifisches, dass die Systemanforderungen beeinflusst
  - → während der Analyse genauere/speziellere Standpunkte herausfinden und damit dann die Anforderungen herausfinden
  - Interviews: sollten nur zusätzlich zu anderen Analysemethoden genutzt werden, da fachspezifische Begriffe der SE Entwickler oft nicht kennt
    - closed interviews: mit vordefinierten Fragen
    - open interviews: mit nicht vordefinierten Ordnungspunkten
  - Szenario:
    - deckt mögliche Interaktionen mit dem System ab
    - Zu Beginn nur grob, später dann detailliertere Beschreibungen
    - für die meisten Leute gut zu verstehen
    - hilfreich um Anforderungen ein zu grenzen

- zum 2. Quadrant
  - gesammelte unstrukturierte Anforderungen werden sortiert und in zusammenhängende Cluster aufgeteilt
- zum 3. Quadrant
  - Prioritäten werden angegeben und Konflikte mit Verhandlungen gelöst;
- zum 4. Quadrant
  - Anforderungen werden dokumentiert und für den nächsten Durchlauf der Spirale verwendet

13.1 Use Cases

- Definition:
  - Texte, die weitestgehend dazu benutzt werden, um die Anforderungen heraus zu finden und fest zu halten
  - somit werden wichtige Aktionen herausgefiltert
  - ist wichtig für die nachfolgende SW Entwicklung: Analyse, Design, Implementierung

- Actor:
  - etwas /jemand mit Verhalten, zB Person, Computer, System, Organisation,...
- Scenario(o.a. use case instance):
  - eine spezielle Folge von Aktionen und Interaktionen zwischen dem Actor und dem System ⇒ also eine spezielle Handlungsfolge/Pfad im System
- Use Case:
  - Sammlung von erfolgreichen und fehlgeschlagenen Szenarien, die beschreiben, wie der Actor ein System benutzt um ein bestimmtes Ziel zu erreichen
- Formate eines Use Case:
  - brief(kurz):
    - knappe Zusammenfassung, in 1 Absatz, i.d.R. das Haupterfolgsszenario
  - casual(ungezwungen):
    - informelles, mehrere Absätze für unterschiedl. Szenarien
  - fully dressed(vollständig bearbeitet):
    - alle Schritte und Möglichkeiten sind im Detail beschrieben, und hilfreiche Zusatzinformationen, wie Vorbedingungen, Erfolgsgarantie
    - zuerst auf die Korrektheit achten, danach auf die Genauigkeit/-Details
- Richtlinien für Use Cases:
  - knapp
  - Black-Box: beschreiben WAS das System macht und nicht wie !!!
  - Nimmt den Blickwinkel des Actors ein: typische Situationen, was soll das Ziel des Systems sein, was ist ein nützliches Ergebnis
  - Daumenregel zum Finden eines Use Cases:
    - + EBP(elementary Business Process): von einer Person als Business-Event bearbeitete kurze Aufgabe
    - Ergebnis hat messbaren Wert
    - Größe: nicht nur 1 Schritt
  - Validierung, um zu überprüfen, ob die Anforderungen dem vom Kunden gefordertem System entsprechen:
    - Gültigkeitskontrolle: sind alle Funktionalitäten korrekt abgedeckt?

- Konsistenzkontrolle: keine Konflikte unter den Anforderungen ?!
- Vollständigkeitskontrolle: sind alle Funktionen & Beschränkungen richtig (wie vom Systemnutzer beabsichtigt) definiert worden
- Realitätskontrolle: können die Anforderungen vernünftig implementiert werden?
- Nachprüfbarkeitskontrolle: kann man Tests entwickeln, die prüfen, ob die Anforderungen erfüllt sind?
- Nachvollziehbarkeitskontrolle: Ist jede Anforderung nachvollziehbar(woher kommt diese?)

Kapitel 14

OO Design

14.1 Responsibility & Coupling & Cohesion

- Betrachtung von Verantwortlichkeiten, Rollen und Collaborations
- erste Schritte: Domänenmodell, Sequenzdiagramm, Interaktionsdiagramm
- System Operationen werden im Sequenzdiagramm dargestellt
- Operation contract sind evtl. hilfreich (ähnl. zu Use Cases: Operation, cross references, Preconditions, Postconditions)
- Interaktionsdiagramm für jede Systemoperation ⇒ wenn zu komplex, dann aufspalten
- Systemoperationen werden einer conceptual class zugewiesen
- **Responsibility (Verantwortlichkeiten)**
  - Zuweisung von Verantwortlichkeiten zu Klassen ist die wichtigste Tätigkeit beim Design ⇒ Pattern, Prinzipien,.. helfen dabei
  - RDD (Responsibility-driven Design): SW Objekte haben Responsibilities
  - zu unterscheiden: Responsibilities ausführen(eine Aktion initiieren, Aktivitäten koordinieren und kontrollieren, etw selber machen) ↔ responsibilities kennen(Wissen von abgekapselten Daten, in Bezug stehenden Objekten)
  - Responsibilities werden Objekten übertragen über die Implementierung von Methoden einer Klasse
  - Responsibility != Methode
- **Kopplung(Coupling)**
  - misst die Abhängigkeit zw Klassen und Packages

  - Formen v. Kopplung: X hat ein Attribut, das sich auf den Typ Y selbst oder einer Instanz davon bezieht; Objekt vom Typ X ruft Methoden vom Objekt vom Typ Y auf; Typ X ist ein Untertyp von Typ Y,
  - hohe Kopplung nicht wünschenswert, denn Isolation ist schwerer zu verstehen, die Klasse ist schwerer wiederzuverwenden(wg der Abhängigkeiten), Änderungen haben Auswirkungen auf die „abhängigen" Klassen(ebenfalls Änderungen nötig)
  - schwache Kopplung → relativ unabhängige, leichter wiederverwendbare Klassen: aber zu geringe Kopplung ist auch nicht wünschenswert

- **Kohäsion (cohesion)**
  - misst die Stärke der Beziehung unter den Klassenelementen
  - Typen:
    - coincidental(zufällig): keine bedeutungsvolle Beziehug zw den Klassenelementen
    - logical/functional cohesion: Elemente e. Klasse bilden zus. eine logische Funktion
    - temporal cohesion: alle Klassenelemente werden zusammen ausgeführt
  - niedrige Kohäsion: nicht erstrebenswert, denn Verständnis, Wiederverwendung, Wartung erschwert → responsibilities an andere Objekete vergeben

- **Orthogonalität (Orthogonality)**
  2 oder mehr Dinge sind orthogonal wenn Änderungen keinen Einfluss auf irgendwas anderes haben(z.B. Änderungen im Datenbank Code haben keine Auswirkungen auf den GUI code)

14.2 GRASP - General Responsibility Assignment Principles

- **Controller**
  - Typen:
    - Façade Controller: Klasse, die das gesamte System präsentiert → bei wenigen Systemereignissen
    - Use Case controller: Klasse die einen künstlichen Händler von allen Ereignissen eines Use Cases hat → bei vielen Systemereignissen
  - koordiniert Abläufe/Aktivitäten, delegiert Aufgaben

  - **„aufgeblähter" Controller**: empfängt alle Systemereignisse, deligiert nichts(macht alles selbst), hat viele Attribute/informationen, dupliziert Informationen aus anderen Objeketen → aufteilen in Use Case Controller(somit niedrige Kopplung und hohe Kohäsion)
  - UI Objekte und UI Layer sollten nicht die Verantwortlichkeit für die Systemereignisse haben

- **Creator**
  - erzeugt eine neue Instanz e. Klasse

- **(Information) Expert**
  - Klasse, die die Informationen besitzt um Verantwortlichkeiten zu erfüllen
  - Information hiding
  - „leichtgewichtige" Klassen

14.3 GRASP - Advanced Principles

- **Polymorphismus**
  - wenn in Beziehung stehende Alternativen oder Verhalten sich vom Typ(Klasse) ändern, dann gebe die Verantwortlichkeit für das Verhalten an und nutze dabei polymorphe Operationen

- **Pure Fabrication**
  - welches Objekt sollte die Verantwortlichkeit besitzen, wenn die hohe Kohäsion und die niedrige Kopplung nicht verletzt werden sollen?
  - → Lsg.: hoch kohäsive Verantwortlichkeiten werden einer künstlichen oder convenience Klasse, die kein problematisches Domänenkonzept präsentiert.

- **Protected Variations**
  - wie kann man Objekte, Subsysteme und Systeme so designen, dass die Veränderungen der Instabilität in diesen Elementen keinen unerwünschten Auswirkungen auf andere Elemente hat?!
  - Lsg.: vorhergesagte Veränderungs- oder Instabilitätspunkte heraussuchen → stabiles Interface drum herum bilden(Open closed Prinzip, Information Hiding, Liskov Substitution Prinzip)

14.4 OO Design Heuristics

- **Basisheuristik**
  alle Daten in der Basisklassen sollten private sein, Methoden protected
- **Heuristiken**
  - God Class Problem:
    eine Klasse/Objekt wei zu viel → Arbeit unter den Klassen aufteilen
  - Inheritance Data-Encapsulation
    - Definition von protected Zugriffsmethoden anstelle von geschtzten Feldern
    - Alle Daten in Hauptklassen sollen private sein
  - Modell-View-Controller
    - Dem Datenmodell ist es nicht erlaubt von dem Userinterface abhngig zu sein
  - Roles vs. Classes
    - Klassen vermeiden die nur eine Rolle bernehmen die auch Objekte in dieser Zeit bernehmen knnten
  - Irrelevant Classes
    - Eine irrelevante Klasse hat keine sinnvolle Bedeutung in der Domne des Systems
- **Proliferation (Vermehrung) von Klassen**
  - bei einem OO Modell mit einem UI, sollte das Modell nie vom Interface abhngen, sondern genau andersrum
  - modellierte Abstraktionen sollen Klassen sein und nicht einfach die Rollen der Objekte
  - irrelevante Klassen entfernen (d.h. bedeutungsloses Verhalten), als Attribut zurck stufen

Kapitel 15

Design Patterns

15.1 Introduction to Patterns

15.1.1 Motivation für den Einsatz/Formulierung von Patterns

Formulierung:

- Experten profitieren von ihrer Erfahrung und erkennen immer wiederkehrende Designprobleme und deren Lösungen
- dadurch verstehen die Experten immer besser, wann es angebracht ist diese Design- Lösung einzusetzen, können sie so allgemein formulieren, dass sie die Wiederverwendbarkeit erreichen

  → durch deren häufige Wiederkehr und Analyse werden Konsequenzen (Kompromisse, Einschränkungen) der Lösungsansätze verstanden und aufgeschrieben

  → somit leichter beachtbar

Einsatz:

- Software-Entwicklung ist teuer

  → Ziel: Wiederverwendbare Software designen.

  → Problem hierbei: Schwierig Wiederverwendbarkeit zu erreichen
- Anfängern helfen, für oft auftauchende und schon effizient gelöste Probleme, zu erkennen, um die effiziente Lösung einsetzen zu können
- einige Designprobleme tauchen und somit auch ihre Lösungen, tauchen immer wieder auf

15.1.2 Was ist ein Pattern(= Entwurfsmuster)?

- Beschreibt ein Problem, welches immer und immer wieder gelöst werden muss
- Die Lösung des Problems, auf eine recht allgemeine Weise und allgemeinem Kontext, dass die einmal umgesetzte Lösung immer und immer wieder verwendet werden kann (→ ermöglicht wiederverwendbare Software/Software-Komponenten) (Christopher Alexander)

15.1.3 Profit durch den Einsatz von Patterns

- Systematische Software-Entwicklung
- Verwendung allgemeiner Lösungen → Steigerung der Wiederverwendbarkeit
- Steigern des Abstraktionslevels

15.1.4 Bestandteile eines Patterns

- Name
- Problem
  Beschreibung wann das Pattern angewendet werden kann (welche Bedingungen erfüllt sein müssen, damit Anwendung Sinn macht)
- Lösung
  Beschreibung der Lösung – d.h. der Elemente des Designs, ihrer Beziehungen, Verantwortlichkeiten und Zusammenarbeit
- Konsequenzen:
  Profit und Verluste durch Anwendung, Einfluss auf System Flexibilität, Erweiterbarkeit, Übertragbarkeit,

15.1.5 Exkursion

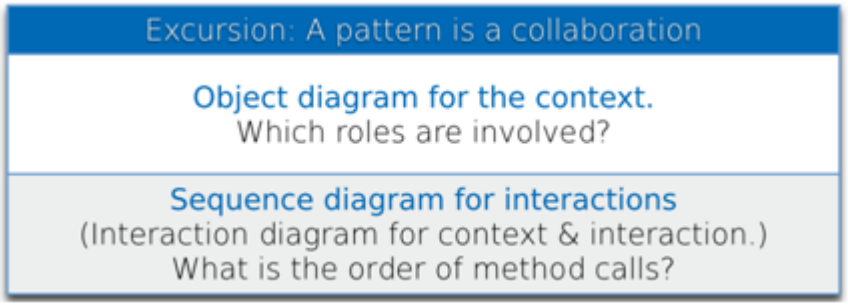


Abbildung 15.1: Exkursion: Pattern – Zusammenwirken von Objekt Diagramm & Sequence Diagramm

15.1.6 Unterschied: Pattern & Idiom

Ein Idiom kann als sprachspezifisches Pattern bezeichnet werden. Mit Idiomen werden in einer Programmiersprache bestimmte wiederkehrende Probleme gelöst.
Beispiel:

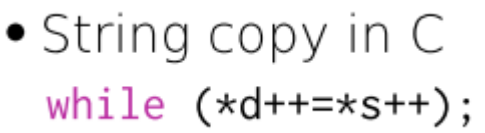


Abbildung 15.2: Idiom-Beispiel: String Copy in C

15.1.7 Was ist ein „Software Design Pattern" ?

Ein Software Design Pattern beschreibt eine üblicherweise immer wiederkehrende Struktur von Software-Komponenten, die ein allgemeines Software Design-Problem, in einem bestimmten Kontext, löst. Dieser Kontext ist in dem Software Design Pattern beschrieben.

15.1.8 Dokumentation der Anwendung eines „Software Design Patterns"

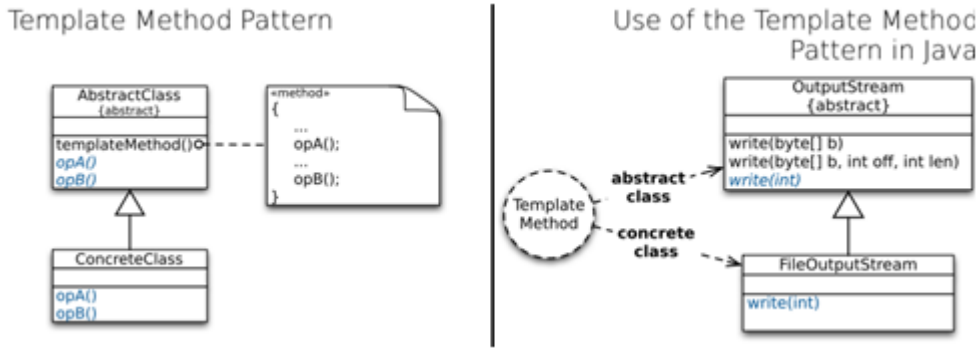


Abbildung 15.3: Dokumentation der Anwendung eines „Software Design Patterns"

15.1.9 Was ist ein „Architectural Pattern" ?

„Architectural Patterns" helfen die grundlegende Struktur eines Software Systems – oder von wichtigen Teilen des Software Systems – zu spezifizieren

Architectural Patterns

- haben einen starken Einfluss auf die äußere Gestaltung eines Software Systems, da hiervon abhängt wie es mit anderen Systemen oder wie andere Systeme mit ihm kommunizieren können
- definiert Zieleigenschaften des Systems
  → z.B.: Art der Kommunikation der verschiedenen Teile des Systems, Art des Datenaustauschs, Grenzen der Sub-Systeme

Die Wahl eines „Architectural Patterns" ist eine fundamental Design-Entscheidung – sie bestimmt alle folgenden Entwicklungsaktivitäten!

15.1.10 Beispiele für „Architectural Patterns":

- Pipes and Filters (nicht genauer behandelt)
- Broker Pattern (nicht genauer behandelt)
- Model-View Controller (MCV):

- Das „model" enthält die zentrale Funktionalität und Daten = unabhängig von Repräsentation der Ausgabedaten & Ver- haltens der Eingaben
- User Interface ist zusammengesetzt aus:
  - „views", die dem Benutzer Informationen anzeigen - erhält die Daten vom „model"
  - „controllers", die Benutzereingaben verarbeiten
    * jede „view" hat einen „controller"
    * „controller" erhält ein Event, übersetzt es in Serviceanfrage für „model" oder „view"
    * Alle Interaktionen laufen über einen „controller" ab
- Change Propagation-Mechanismus (Veränderungs-Fortpflazungs-Mechanismus) sichert Konsistenz zw. „view"/Interface und „model"
  - Oft implementiert mit Observer Pattern oder Publish-Subscriber Pattern
  - Ablauf: „view" registriert sich bei „model" →(„contoller" hängt von „model"- Zustand ab) „model" ändert sich → „controller" registriert sich bei Change Propagation Mechanismus → „view" änder sich
- Struktur:

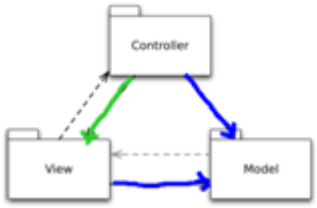


Abbildung 15.4: Struktur des Architectural Patterns „Model View Controller"

„view" & „controller" direkt gekoppelt mit „model" „controller" direkt gekoppelt mit „view" „model" indirekt gekoppelt mit „view" oder „controller" „view" indirekt gekoppelt mit „controller"

- Konsequenzen:
  * Steigende Komplexität, ohne durch die Nutzung von unterschiedl. „views" & „controllern" wirklich mehr Felxibilität zu erreichen
  * Effizient für ohne Anzahl von Veränderungen, da nicht immer gleichzeitig alle „views" geändert werden müsse
  * Innige Verbindung zwischen „view" & „controller"

- Broker
-

15.2 Template Method Pattern

15.2.1 Ziel (Benutze wenn):

- separieren von veränderlichen und unveränderlichen Teilen
- zum Vermeiden von doppeltem Code in Subklassen; das allgemeine, bei allen Subklassen gleichbleibende, Verhalten ist in einer gemeinsamen (Ober-)Klasse lokalisiert und wird dort auch implementiert
- kontrollieren von Subklassen-Erweiterungen

Abbildung 15.5: Design Ziele des Template Method Patterns

15.2.2 Struktur:

- definiere ein Skelett eines Algorithmuses in einer Methode
- verlagere die Implementierung der veränderlichen Methoden, bzgl. der genauen Funktionalität der jeweiligen Subklasse, in die Subklasse selbst

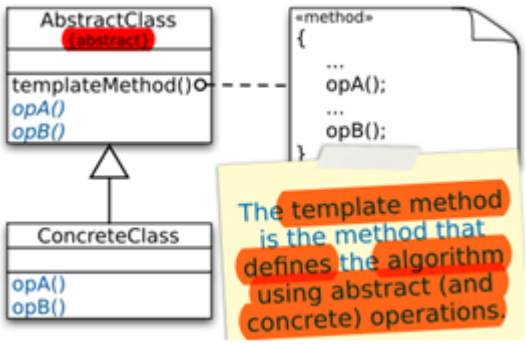


Abbildung 15.6: Struktur des Template Method Patterns

Beispiel:

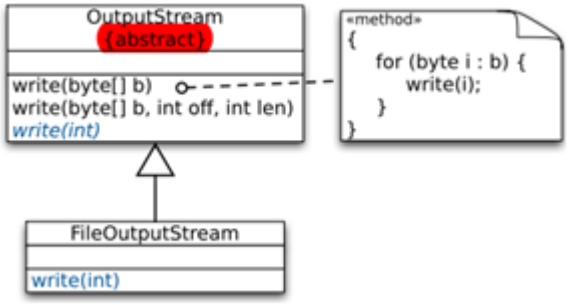


Abbildung 15.7: Beispiel: Struktur des Template Method Patterns

15.2.3 Wichtige Begriffe

- „template method”:
  Methode, die das Skelett des Algorithmuses definiert, indem sie abstrakte und ausimplementierte Methoden nutzt

15.2.4 Konsequenzen

- Grundlegende Technik zur Wiederverwendung von Code – besonders wichtig in Klassenbibliotheken, da sie allgemein gleichbleibendes Verhalten be-

reits implementieren und Implementierung von veränderlichem Verhalten in die Sub-Klassen verlagern

- Führen zur Invertierung des Kontrollflusses. Dies bezieht sich darauf, dass Elternklasse (bereits implementierte Methode der Elternklasse) Methoden einer Sub-Klasse aufruft und nicht umgekehrt (wie üblich)

- „Template Methods" rufen folgende Arten von Methoden auf:

  - Konkrete Methoden (entweder von ConcreteClass oder der Client-Klasse)
  - Konkrete Methoden der AbstractClass ((Methoden, die allgemein gleichbleiben und sinnvoll von Sub-Klassen wiederverwendet werden können)
  - Abstrakte Methoden, die dann von den Sub-Klassen erst implementiert werden müssen, da sich Verhalten dieser Methoden in Sub-Klassen unterschiedlich sein kann
  - „Factory Methods"
  - Einschubmethoden, die Defaultimplementierung anbieten, die aber, falls nötig, von Sub-Klasse überschrieben werden kann. Einschubmethoden machen in ihrem Defaultverhalten oft gar nichts

Wichtig: „Template Method"/abstrakte Klasse, die „Template Method" enthält, festlegen welche Methoden Einschubmethoden und welche Methoden abstrakt sind
→ effiziente Wiederverwendbarkeit einer abstrakten Klasse mit „Template Methods" ist nur gegeben, wenn Programmierer versteht welche Methoden abstrakt und welche Einschubmethoden s

15.2.5 Implementierung

(in VL nicht behandelt; für genauere Informationen siehe „Entwurfsmuster" (Gamma, Helm, Johnson, Vlissides) Seite 370-372)

- *Minimierung abstrakter Operationen*
  Ziel bei der Implementierung soll es sein, dass Sub-Klasse so wenig wie möglich abstrakte Methoden der abstrakten Elternklasse, zum Ausführen des Algorithmuses überschreiben muss

15.2.6 Anwendungsgebiet

- Frameworks
- APIs

15.2.7 In Zusammenhang stehende Patterns

- „Factory Method Pattern":
  - Siehe „Factory Method Pattern"
  - „Factory Methods" werden oft durch „Template Methods" aufgerufen
- „Strategie Pattern" (Strategie-Muster)
  - Auch bekannt als „Policy Pattern"
  - Definiert Familie von Algorithmen, kapselt jeden einzelnen und macht sie somit austauschbar
  - Algorithmus kann unabhängig vom ihn benutzenden Client geändert werden
  - „Template Method Pattern" verwendet Vererbung um Teile des Algorithmuses zu variieren; „Strategie Pattern" verwendet Delegation, um kompletten Algorithmus zu variieren

15.3 Composite Pattern

15.3.1 Ziel (Benutze wenn)

- Zusammengesetzte Objekte in eine Baum-Struktur bringen (= Repräsentation von Teil-Ganze-Hierarchien)
- „Composite Pattern" erlaubt Client einfache Objekte (= nicht zusammengesetzt) & Kompositionen gleich zu behandeln

15.3.2 Beispiel

Zeichenprogramm, indem komplexe Bilder aus primitiven Komponenten entstehen, aber gleich den Primitiven behandelt werden sollen.

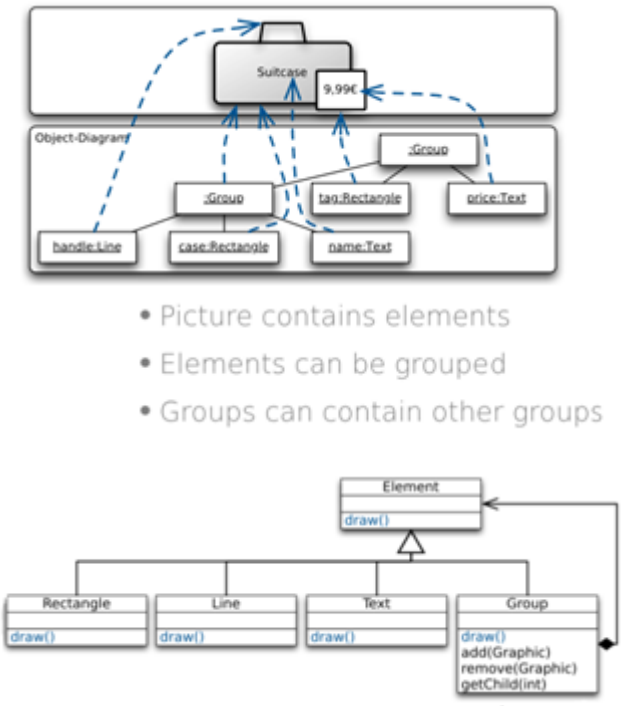


Abbildung 15.8: Beispiel: Composite Patterns

15.3.3 Struktur

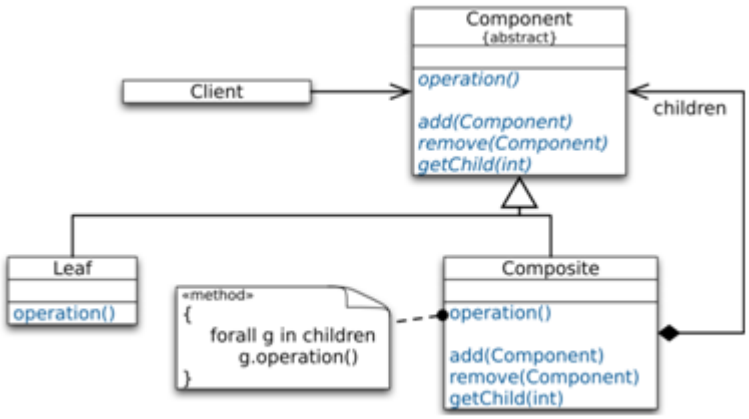


Abbildung 15.9: Struktur des Composite Patterns

- Component:
  - Deklariert das Interface für Objekte im Composite
  - Implementiert das Standard-Verhalten
  - Deklariert (oft) auch ein Interface für Zugriff und Handhabung der Child-Komponenten
- Leaf:
  - Repräsentiert die Blatt-Objekte der Komposition & definiert das primitive Verhalten
  - Reagieren direkt auf Requests des Client
- Composite:
  - Speichert die Child-Objekte
  - Implementiert das Verhalten des Composite-Objekts
  - Reicht Requests des Clients an seine Kinder weiter
- Client:
  - Greift auf Objekte des Composite, über das Component-Interface, zu

15.3.4 Konsequenzen

- Primitive Objekte können rekursiv kombiniert werden
- Clients können Composite-Objekte und Leaf-Objekte gleich behandeln
  (keine Funktionen im tag-and-case Stiel)
- Neue Bestandteile können leicht hinzugefügt werden
- Design wird übergeneralisiert
  (kann sich nichtmehr auf das Typ-System verlassen – z.B. dass Composite nur Objekte von bestimmtem Typ enthält)

15.3.5 Implementierung

- Explizite Eltern-Referenz *(direkte Auswirkung auf Implementierung durch „Composite Pattern")*
  (kann das traversieren und organisieren der Composite-Struktur erleichtern; oft im Component definiert. Definition aus Component MUSS beibehalten werden)
- Größe des Component-Interfaces *(direkte Auswirkung auf Implementierung durch „Composite Pattern")*
  Um Client im Unklaren zu lassen, was genau eine Leaf- oder Composite-Klasse ist, muss Component-Klasse so viele Methoden wie möglich für Leaf und Composite implementieren
- Frage wo Child-Management-Methoden zu deklarieren sind *(direkte Auswirkung auf Implementierung durch Composite Pattern)*
  - Im Component ist es günstig, aber weniger sicher, da Clients versuchen könnten mit diesen Methoden zu arbeiten
  - Im Composite ist es sicher, da Client bewusst nur Component kennt und somit nichts über Methoden es Composite weiß

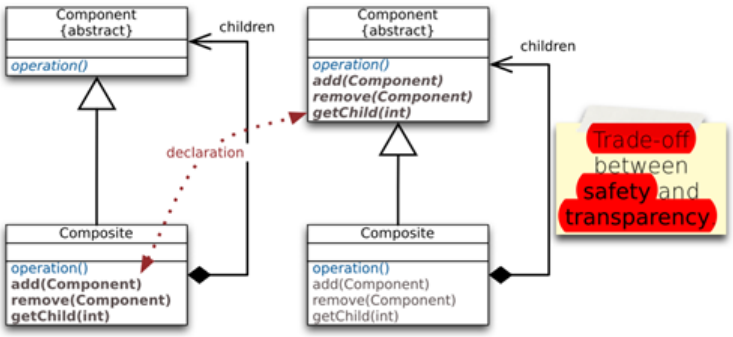


Abbildung 15.10: Wo ist die Child-Meanagement-Methode zu implementieren? - Kompromiss zwischen Sicherheit und Transparenz

- Teilen von Komponenten *(Verbesserungsmöglichkeit der „Composite Pattern"-Implementierung)*
  nützlich um z.B. den Speicherbedarf möglichst gering zu halten
  → Implementierung des „ Flyweight Patterns" zusammen mit „Composite Patter"

15.3.6 In Zusammenhang stehende Patterns

- „Iterator Pattern": um über den Composite zu iterieren
- „Visitor Pattern":
  - um Methode zu lokalisieren, die über Leaf- und Composite-Klassen verteilt sind
  - kapselt eine auf Elementen einer Objektstruktur auszuführende Operation als ein Objekt
  - ermöglicht es neue Operation hinzuzufügen, ohne die Klasse der zu bearbeitenden Elemente zu verändern
- „Chain of Responsibilty Pattern':
  - Vermeidung der Koppelung von Request-Auslöser und Empfänger, indem mehr als ein Objekt Möglichkeit erhält die Aufgabe zu erledigen. Verkette empfangenden Objekte und leite Request an Kette entlang bis ein Objekt der Kette Request erledigt
  - Verbindung zwischen Eltern- und Kindobjekt wird oft zur Implementierung einer „Chain of Responsibility" wiederverwendet

- „Flyweight Pattern":
  - zum Teilen von Komponenten
  - gemeinsames Nutzen von kleinsten Objekten gemeinsam, um große Mengen von ihnen Nutzen zu können, Speicheraufwand zu reduzieren und Effizienz zu gewinnen (z.B. beim Laden des Composite)

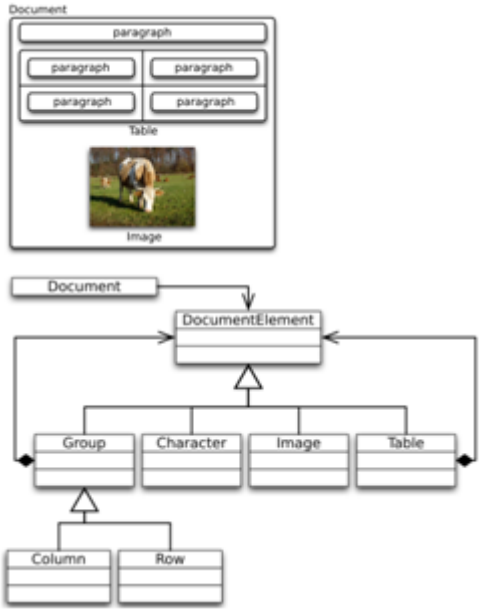
15.4 Iterator Pattern

15.4.1 Ziel (Benutze wenn)

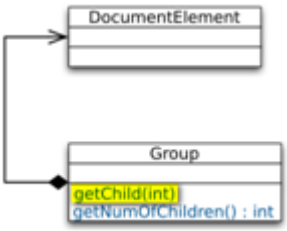
- Wenn die Traversierung nicht vom internen Aggregierungsmechanismus für die Composite-Klassen abhängen soll bzw. nichts über internen Aggregationsmechanismus des Composites preisgeben soll
  (z.B. Array oder LinkedList)
- Um mehrfache gleichzeitige Traversierungen auf einem Composite zu ermöglichen
- Um polymorphe Traversierung zu ermöglichen
  (= einheitliche Schnittstelle zur Traversierung unterschiedlicher Composites, die aber das selbe Interface implementieren, anzubieten – z.B. unterschiedliche Arten von Listen)

15.4.2 Beispiel

Traversieren eines Dokuments – z.B. um Rechtschreibprüfung durchzuführen



BEISPIEL – LÖSUNG 1:



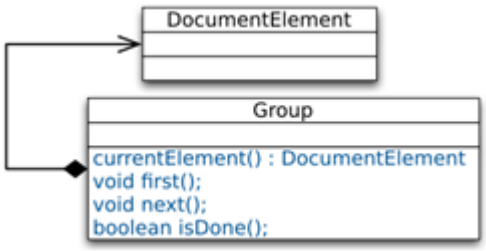
definiere eine getChildren(int) -Methode, die ein Kind an einem bestimmten Indes zurückgibt

Auswertung:

schlecht, weil:

- Ineffizient, wenn Children in einer LinkedList aggregiert werden
- Das Interface sollte keine bestimmte Implementierung voraussetzen
  (d.h. in dieser Implementierung würde vom Interface vorausgesetzt werden, dass die Children nicht in LinkedList aggregiert werden sollte, aufgrund der Ineffizienz von getChildren(int) bzgl. LinkedLists)

BEISPIEL – LÖSUNG 2:




```
void draw(Group group) {
   DocumentElement elem;
   group.first();
   while (!group.isDone()) {
      elem = group.currentElement();
      elem.draw();
      group.next();
   }
}
```

currentElement(): gibt das aktuelle Element „current" zurück

first(): setzt „current" auf das erste Element

next(): setzt „current" auf des nächste Element von der vorherigen „current"-Belegung

isDone(): gibt true zurück, wenn das letzte Element des Container-Elements erreicht ist

Auswertung:

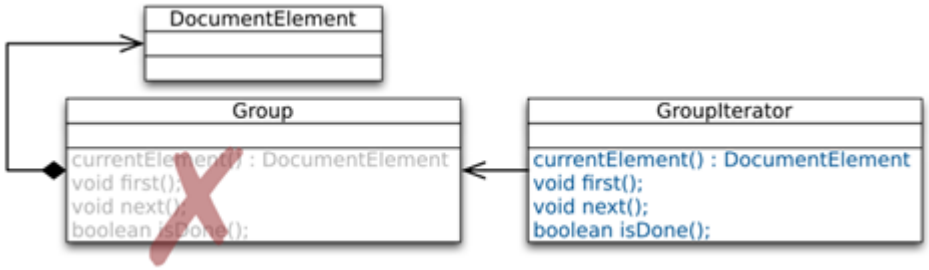
besser, weil:

- ein allgemeineres Interface definiert **Linearen Zugriff**
- jede Art von Container-Implementierung funktioniert hier gleichermaßen
- Sogar **Streams** – berechne von Elementen in der Reihenfolge wie sie angefragt werden, ist möglich

VERBESSERUNG VON „BEISPIEL – LÖSUNG 2":
Jeder einzelne Iterator hat seine eigene Referenz auf das aktuelle Element

Auswertung:

besser, weil:



- Mehrere gleichzeitige Iterationen auf einem Composite sind möglich
- Composite ist von Iterator-Funktionalität entbunden
  (erhöht die Kohäsion der Klasse) (!)

15.4.3 Struktur

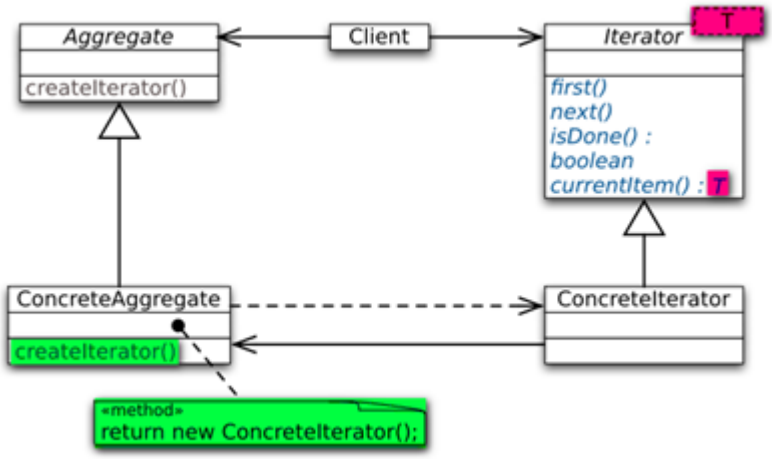


Abbildung 15.11: Struktur des Iterator Patterns

- Iterator-Typ ist für Client sichtbar
- createIterator() ist eine **factory method**

- Iterator
  definiert ein Interface für die Traversierung von aggregierten Objekte/-Composites

- ConcreteIterator
  Bietet eine konkrete Implementierung des Iterator-Interfaces – basierend auf speziellen ConcreteAggregate – an.
  Verfolgt die Spur der aktuellen Position im Aggregat
  ⇒ Steigerung der Effizienz von Zugriffen auf Aggregat vor allem bei Datenstrukturen wie LinkedLists

- Aggregate
  Definiert ein Interface für Erzeugung von Iterators

- ConcreteAggregate
  Ist eine Aggregation von einfachereren Elementen und implementiert die Iterato-Creation-Funktion und gibt einen ConcreteIterator zurück

15.4.4 Konsequenzen

- ***Abstrakte Traversierung eines Aggregate***
  Clients kennen die interne Struktur des Aggregates nicht

- ***Iterators vereinfachen das Aggregationsinterface***
  Aggregates müssen nicht die Traversierungsfunktionalität unterstützen – außer die Iterator-Creation-Funktion

- ***Unterstützt Variation in der Traversierungsstrategie***
  Konkrete Iteratoren können unterschiedliche Traversierungsmechanismen anbieten (z.B. rückwärts durch die Aggragation gehen)

- ***Mehrere Traversierungen können zur selben Zeit auf einem Aggregate stattfinden***
  Jeder Iterator verfolgt selbst die Spur seiner aktuellen Position im Aggregate

15.4.5 Implementierung

- *Robust Iterators (Robuste Iteratoren):*
  Ein robuster Iterator stellt sicher, dass das Hinzufügen oder Entfernen eines Elements des Aggregates nicht in Konflikt mit der Traversierung gerät

  - **mögliche Lösungen:**

  - erstelle Kopie des Aggregates für die Traversierung – so können keine Elemente gelöscht werden
    (ist aber nicht unbedingt beste Lösung, da Erstellung einer Kopie generell betrachtet zu teuer ist)
  - mache Iterator zum Observer des Aggregates, sodass Iterator benachrichtigt wird, wenn eine Zustandsänderung des Aggregates statt gefunden hat
    * zusätzliche Funktionalität nötig:
      - skipTo
      - remove
      - previous

- *externer vs. interner Iterator (external vs. Internal Iterator):*
  - **externer Iterator:**
    - Client steuert die Iteration selbst
    - Clients, die externe Itertoren verwenden, müssen Traversierung selbst vorantreiben – d.h. selbst aktiv nächstes Element von Iterator verlangen
    - Felxibler als interne Iteratoren – z.B. einfach mit externen Iterator zwei Container zu vergleichen (mit internen praktisch unmöglich)
  - **Interner Iterator:**
    - Iterator selbst steuert Traversierung (mit eigener Traversierungsmethode)
    - Besonders nützlich, wenn Zugriff auf internen Zustand (= private Variablen des Aggregates) gewünscht oder Iterationslogik komplex ist
    - Gleicheitsvergleich von zwei Contanern ist sehr schwierig, da informationen über momentan berachtete Elmente von Iterator nichz weitergegeben werden

- *NullIterator (Nulliterator):*
  - Dient zur Handhabung von Grenzbedingungen – z.B. Blatt-Knoten in einer Baumstruktur
  - Per Definition ist NullIterator immer am Ende der Iteration angekommen (isDone() liefert immer true)
  - Machen Traversierung von baumähnlichen Daten-Objekten/Composites einfacher, denn:
    - Zu jedem Zeitpunkt in Iteration wird Child-Knoten nach Iteratorobjekt gefragt
    - Composites geben, wie üblich, konkreten Iterator zurück
    - Blatt-Knoten geben NullIterator zurück

- Möglichkeit gesamte Struktur einheitlich zu traversieren

15.4.6 In Zusammenhang stehende Patterns

- „Composite Pattern":
  - Siehe „Composite Pattern"
  - Iteratoren werden oft auf rekursive Strukturen wie Composites angewendet
- „Factory Method Pattern":
  - Siehe „Factory Method Pattern"
  - Polymorphe Iteratoren basierne auf „factory methods" um die passenden Iterator-Sub-Klasse zu instanziieren

15.5 Observer Pattern

15.5.1 Ziel (Benutze wenn)

- wenn ein flexibler Weg nötig ist, dass Objekte einander über ihre Änderungen informieren – **OHNE STARKE KOPPELUNG** der Klassen („Communication without Coupling")
- Dekopplung vom Daten-Modell und „Parteien", die an Änderungen des internen Zustands des Modells interessiert sind

  Anforderungen, um Ziel zu erreichen:
  - Subject sollte nichts über seine Observer wissen – d.h. über die konkreten Observer-Klassen (nur, dass sie Observer-Interface implementieren)
  - Identität und Anzahl der Observer ist nicht vorherbestimmt
  - Neue Observer-Klassen (=Receiver-Classes/Empfänger-Klassen) werden möglicherweise zukünftig dem System hinzugefügt
  - Zyklische Abfragen (= **polling**) ist ungeeignet (zu ineffizient), deswegen sendet ConcreteSubject bei Zustandsänderung immer eine notifyObserver-Message

15.5.2 Beispiel

Dokument-Editor:

- Präsentationskomponenten, die Sichten auf das Dokument betreffen sollen von der Kern-Datenstruktur des Dokuments getrennt sein
  (Kommunikation muss aufgebaut werden)
- verschiedene Sichten auf ein Dokument simultan möglich sein
  (Updates, die Dokument präsentieren müssen gehandhabt werden)

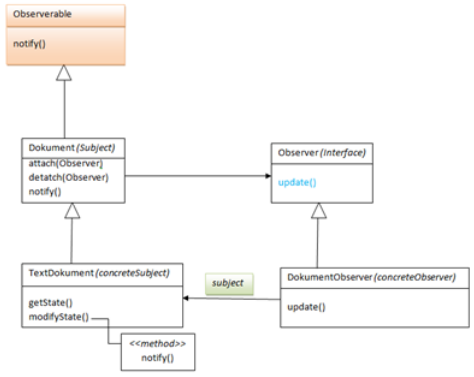


Abbildung 15.12: Dokument-Editor – ein Beispiel für den Einsatz des Observer Pattern

15.5.3 Struktur

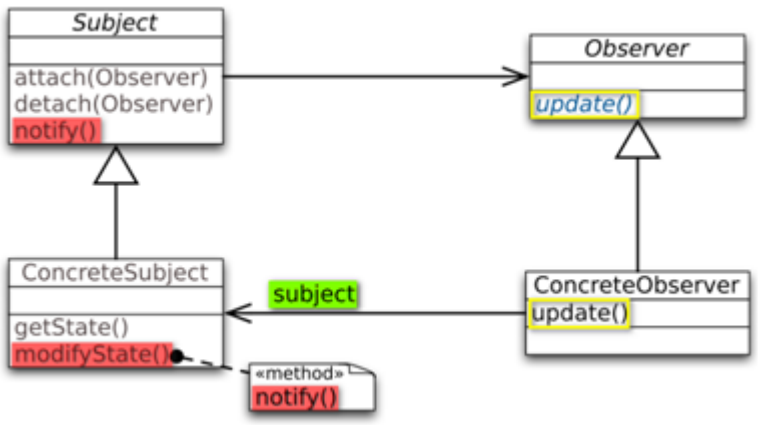


Abbildung 15.13: Struktur des Observer Pattern

- Subject
  - Kennt seine(n) Observer – aber nur insoweit, dass ihm klar ist, dass es eine Liste von Observern hat; kennt aber nicht die KONKRETEN Klassen
  - Bietet Methoden an Observer an- und abzumelden
  - und Methoden, um Observer Zustandsänderung mitzuteilen

- Observer
  - Definiert ein Aktualisierungs-Interface für Objekte, die über eine Änderung des Subjects benachrichtigt werden sollen
- ConcreteSubject
  - Speichert den, für ConcreteObserver, interessanten Zustand
  - Sendet Änderungsbekanntgabe gefolgt auf eine Zustandsänderung an seine Observer
- ConcreteObserver
  - Verwaltet Referenz auf ein ConcreteSubject
  - Speichert den Zustand, der mit dem des subjects in Einklang stehen sollte
  - Implementiert das Aktualisierungs-Interface des Observers, um seinen Zustand mit dem des subjects konsistent zu halten

Interaktion der Teilnehmer:

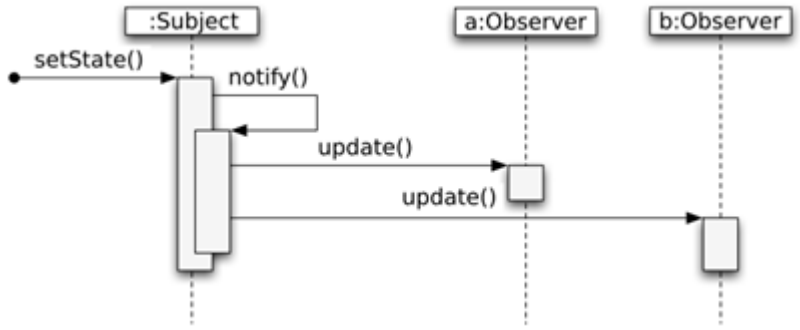


Abbildung 15.14: Interaktion der Klassen des Observer Patterns

1. Zustand des Subjects wird festgestellt und in den Observern gespeichert
2. Eine Änderung findet statt
   2.1 Subject teil Änderungen seinen Observern mit notify() mit
   2.2 ruft update()-Methode seiner Observer auf, damit diese wieder aktuellen Zustand des Subjects haben

Protokoll der Teilnehmer:

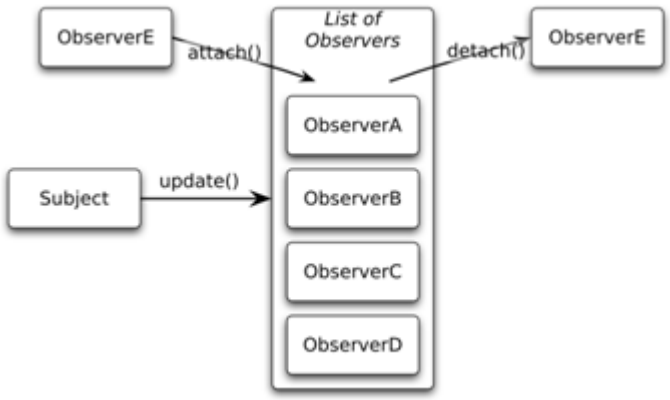


Abbildung 15.15: Protokoll der Klassen des Observer Patterns

15.5.4 Konsequenzen

- Möglichkeit Subject und Observer unabhängig voneinander zu variieren
- Wiederverwendbarkeit von Subject- und Observer-Klasse steigt, da sie unabhängig voneinander wiederverwendet werden können
- Neue Observer können ohne Probleme hinzugefügt werden
- Anzahl und Art der Observer ist nicht vorherbestimmt
- Verhindert folgendes Problem:
  Change Propagation (=Änderungsweitergabe) von Zuständen eines Objekts kann in den Objekten fest verdrahtet sein (z.B. nur bzgl. bestimmter konkreter Klassen funktionieren. Dies bindet die Objekte zusammen und verringert deren Flexibilität und ihre (getrennte) Wiederverwendbarkeit
- Weitere Vorteile und Verpflichtungen bei Entscheidung für „Observer Pattern”:
  - *Abstrakte Kopplung zwischen Subject und Observer:*
    - Subject weiß nur, dass es Observer-Liste hat
    - Subject weiß nur, dass jeder Observer in Liste Observer-Interface implementiert

  - Subject kennt keine der konkreten Observer-Klassen
    ⇒ Folge: **Kopplung** zwischen Subject und Observer abstrakt und auf Minimum reduziert

- **Unterstützung von Broadcast-Kommunikation:**
  - Im Gegensatz zu normalen Anfragen muss Sender (Subject) Empfänger seiner Nachricht nicht spezifizieren
  - Sender kennt den konkreten Typ des Empfänger-Objekts nicht
  - Benachrichtigung wird automatisch an interessierte Objekte weitergeleitet, die sich bei Sender registriert haben
  - Anzahl der registrierten Objekte ist unerheblich
  - Sender (Subject) hat nur Aufgabe interessierte Objekte (Observer) zu benachrichtigen. Empfänger (Observer) hat Aufgabe Benachrichtigung zu bearbeiten oder zu ignorieren
  - Aufgrund dessen Observer ohne Einschränkung an- und abmeldbar

- Unerwartete Akutalisierungen:
  - Gründe hierfür, welche dann zum Problem – u.a. für Performance – werden können:
    * Observer wissen nichts von einander, d.h. es kann nur schwer abgeschätzt werden was die Aktualisierung eines Subjects wirklich kostet. Denn es könnte sein, dass eine Aktualisierung eine Kaskade weiterer auslöst
      (falls es z.B. eine Kette von Observern gibt, über die man sich auch nicht direkt im Klaren ist)
    * Schlecht definierte oder schlecht gewartete Abhängigkeiten können zu unsinnigen Aktualisierungen führen
      (schwer aufzuspüren)

  Problem der unerwarteten Aktualisierungen wird dadurch erschwert, dass im einfachen Aktualisierungsprotokoll (in Java: notifyObservers() ohne Parameter) nicht beschreibt, was sich im Subject geändert hat. Somit nicht nur unsinnige/unerwartete Aktualisierungen möglich, sonder unnötige – wie z.B. falls etwas im Fließtext eines Dokuments geändert wird, muss sich nicht Gliederungs-View updaten.
  → Observer haben (mitunter) große Probleme herauszufinden was sich geändert hat
  → *Problemlösung:*
  erweitertes Aktualisierungsprotokoll, in dem Observern mitgeteilt werden kann was
  sich geändert hat (in Java: notifyObservers(Observerable subject, Object message)
  subject = beobachtetes Objekt, message = Objekt mit dem Observer Infos bzgl.
  Änderung erhalten kann)

- Ein allgemeines Update-Interface für alle Observer beschneidet die Flexibilität des Kommunikationsinterfaces: Subject kann keine optionalen Parameter an die Observer schicken

15.5.5 Implementierung

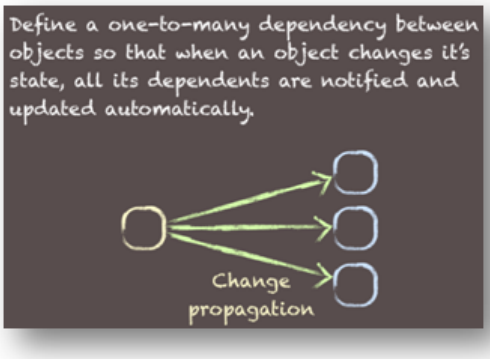


Abbildung 15.16: Implementierung des Observer Patterns

Wichtige Frage: Wer soll die Updates, folgend auf eine Änderung, steuern – der Client oder zustandsändernde Methoden von Subject?

- Frage ist nicht einfach zu entscheiden
- Jedoch, falls es mehrere Änderungen zum selben Zeitpunkt gibt, man nicht will, dass alle ein Update auslösen und man nach dem einfachen Aktualisierungsprotokoll vorgeht (= Änderungsbekanntgabe wird ohne weitere Parameter gesendet) kann die Entscheidung es den Änderungsmethoden von Subject zu überlassen ineffizient werden

siehe: „Entwurfsmuster" (Gamma, Helm, Johnson, Vlissides), Seite 292-296
(in VL nicht behandelt)

15.5.6 In Zusammenhang stehende Patterns

- „Mediator Patter" (Vermittler-Muster):
  - Definiert ein Objekt, welches Zusammenspiel einer Menge von Objekten sin sich kapselt
  - **Fördert lose Kopplung, indem zusammenarbeitende Objekt davon abgehalten werden direkt aufeinander bezug zu nehmen**
  - Zusammenspiel der Objekte kann von Objekten unabhängig geändert werden

  (Durch Kapselung komplexer Aktualisierungssemantik wirkt Änderungs-Manager als Vermittler zwischen Subject und Observern)
- „Sigleton Pattern":
  - Siehe „Sigleton Patter"

  (ÄnderungsManager kann Sigleton Pattern" verwenden, um einmalig und global zugreifbar zu sein)

15.5.7 Exkurs: Konsequenzen der objektorientierten Programmierung & Ausgleichen der Nachteile durch das „Observer Pattern" bzw. ein Nachteil für das „Observer Pattern"

- Objektorientierte Programmierung: Fördert das Aufbrechen von Problemen in Objekte, die kleine Menge von Verantwortlichkeiten haben; aber zusammenarbeiten können, um komplexe Aufgaben zu lösen.

  - Vorteil:
    - Macht jedes Objekt einfacher zu implementieren und zu handhaben
    - Höhere Wiederverwendbarkeit von Objekten bzw. Klassen
    - Ermöglichen flexibler Kombinationen
  - Nachteil:
    - Unterschiedliche Objekte bekommen Zustandsänderungen des jeweils anderen nicht mit
    - Das Verhalten ist auf mehrere Objekte aufgeteilt; jede Zustandsänderung eines Objekts betrifft oft viele andere

- Ausgleichen der Nachteile durch das „Observer Pattern":
  Durch Implementierung des „Observer Patterns" kann ausgeglichen werden, dass normaler Weise ein Objekt die Zustandsänderung eines anderen nicht mitbekommt

- Nachteil für das „Observer Pattern":
  Dadurch, dass Verhalten auf mehrere Objekte verteilt ist, kann es bei der Implementierung des „Observer Pattern" zu unterwarteten Aktualisierungen kommen.
  Bsp: Falls man eine unbeabsichtigte Kette von Observern hat, kann folgendes passieren:
  Objekt A erfährt Zustandsänderung → Observer ObsA aktualisiert sich → ObsA wird von ObsB beobachtet → ObsB aktualisiert sich wieder → usw.

15.6 Command Pattern

Momentan noch kein Inhalt vorhanden

15.7 Decorator Pattern

Momentan noch kein Inhalt vorhanden

15.8 Adaptor Pattern

15.8.1 Ziel (Benutze wenn)

- Ermöglichen der Zusammenarbeit von Klassen, welche nicht zusammenarbeiten könnten, aufgrund von inkompatiblen Interfaces durch:
- passt ein Interface einer Klasse in ein anderes Interface, welches ein Client erwartet, an
  → Anpassung nötig, falls:
  - Library-Komponenten passen nicht zur Art der Strukturierung des Programms, die Programmierer gewählt hat
  - Unterstützung der Wiederverwendbarkeit von Komponenten, dadurch, dass sie durch Adaptor passend gemacht werden
  - Objekte einer Library sind nicht dafür designed worden, um mit Objekten einer anderen Library zusammenzuarbeiten, aber nun braucht man das

15.8.2 Beispiel

Es soll über eine Datei-System traversiert werden, welches folgendermaßen aufgebaut ist:
Es gibt Directories und Files. Die Klassen der beiden erben von einem Interface „Node". Zur Traversierung soll die Klasse „TreeViewWidgets" genutzt werden, welche das Interface „TreeItem" nutzt, d.h. es wird über TreeItems traversiert.
Lösung:
Objektadapter „TreeItemAdaptor", der das Interface „Node" (=Adaptee) an das Interface „TreeItem" (=Target) anpasst (hier nur einfache Schnittstellenkonvertierung notwendig, d.h. Methoden von „Node", die die benötigten Informationen liefern, in den korrespondierenden von „TreeItemAdaptor" aufrufen)

15.8.3 Struktur

- Klassenadapter (Class Form Adaptor):

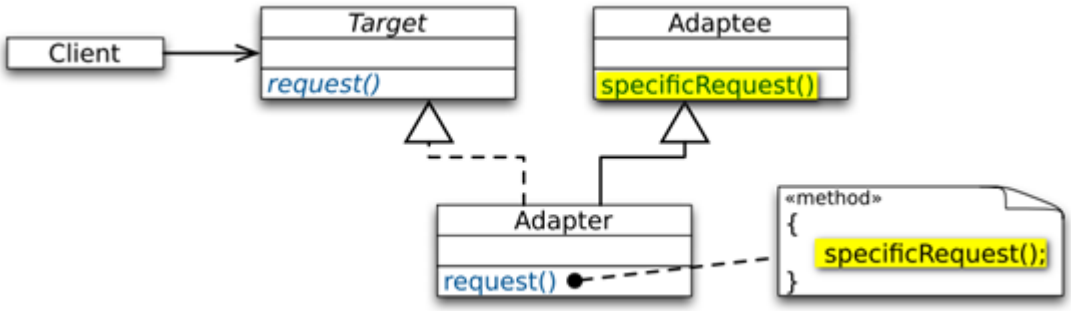


Abbildung 15.17: Struktur des Adaptor Patterns bei Implementierung eines Class Form Adaptor

spezielles Verhalten des Adaptors:
- Nutzt Mehrfachvererbung, um von Target und Adaptee zu erben

- Objektadapter (Object Form Adaptor):

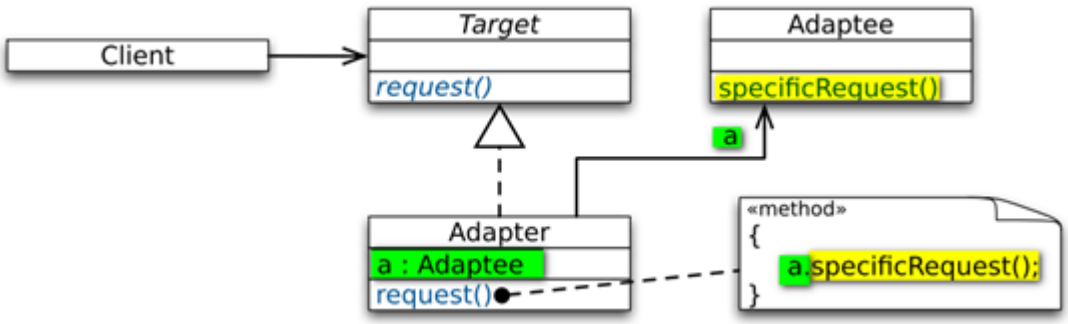


Abbildung 15.18: Struktur des Adaptor Patterns bei Implementierung eines Object Form Adaptor

spezielles Verhalten des Adaptors:
- Reicht Requests an den Adaptee weiter

- Target (Ziel):
  - Definiert die anwendungsspezifische vom Client verwendete Schnittstelle (=Zielschnittstelle)
- Adaptee (AdaptierteKlasse):
  - Definiert eine existierende und anzupassende Schnittstelle
- Adaptor (Adapter):
  - Passt die Schnittstelle der anzupassenden Klasse an die Zielschnittstelle an
  - Passt bereits existierende Komponenten in ein existierendes Design ein
  - Implementiert Funktionalität, die im Adaptee fehlt

15.8.4 Konsequenzen

- Klassenadaper (Class Form Adaptor):
  - Ermöglicht es in der Adaptor-Klasse Teile des Verhaltens des Adaptees zu überschreiben, da Adaptor Sub-Klasse des Adaptees ist.
  - Adaptor und Adaptee sind ein Objekt – keine Weiterleitung von Requests
  - Passt EINEN Adaptee an GENAU EINE konkretes Target (Zielschnittstelle) an
    → somit funktioniert Klassenadapter nicht, wenn eine Klasse/Interface und all ihre Sub-Klassen adaptiert werden sollen
- Objektadapter (Object Form Adaptor):
  - Ermöglicht es Adaptor mit Adaptee und seinen Sub-Klassen zu arbeiten und zwar mit dem Adaptee selbst und all seinen Sub-Klassen
  - Adaptor kann neue Funktionalität Adaptee und seinen Sub-Klassen auf einmal hinzufügen
  - Kann das Verhalten des Adaptees nicht überschreiben
  - Adaptor und Adaptee sind unterschiedliche Objekte
- Weitere zu bedenkende Konsequenzen der Entscheidung für das „Adaptor Pattern": (in VL nicht behandelt)
  - Ausmaß der Anpassung durch einen Adaptor:
    Das Ausmaß und somit der Aufwand für Anpassungen kann stark schwanken. Spektrum reicht von einfachere Schnittstellenkonvertierung durch Änderung der Methodennamen (= Methoden des Adaptee in Methoden des Adaptors aufrufen) zu Implementierung gänzlich

neuer Funktionalität
→ **Arbeitsaufwand hängt von Ähnlichkeit der Schnittstellen von Target und Adaptee ab!**

- Überlegung, ob man bei einer Klasse, die hohe Wiederverwendbarkeit haben soll nicht sofort Fähigkeit der Schnittstellenadaptierung integriert (d.h. sie um Adaptorfunktionalität ergänzt), denn:
  - Schaltet Bedingung aus, dass andere Klassen, die mit obiger Zusammenarbeiten sollen selbe Interface nutzen müssen
  - Ermöglicht es obige Klasse leichter in andere Software-Systeme einzufügen, da sie vorgefundenen Interfaces adaptieren kann ⇒ somit dann Zusammenarbeit möglich
- Verwendung von **Zweiweg-Adaptern**:
  - Gleicht Schwachstelle von Adaptern aus, dass Adapter nicht für alle Clients gleich transparent sind, d.h. adaptiertes Objekt entspricht nichtmehr Adaptee-Schnittstelle, sondern nur noch Target-Schnittstelle.
    ⇒ folglich nicht mehr als Adaptee-Objekt verwendbar
  - Besonders nützlich dort, wo die selben Objekte von unterschiedlichen Clients mit unterschiedlichen Schnittstellen betrachtet werden.

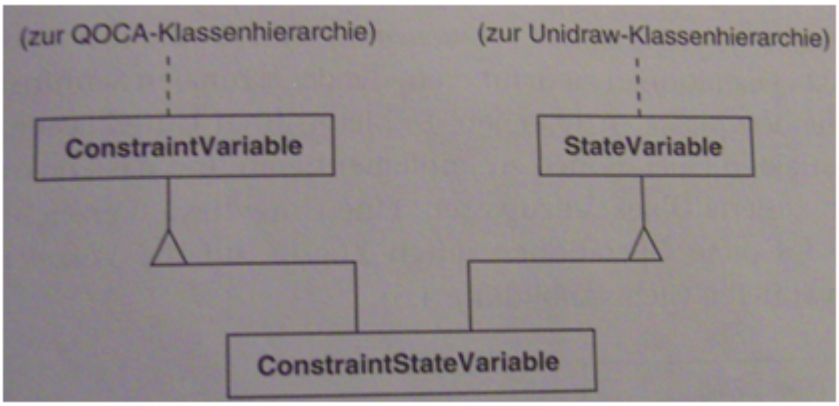


Abbildung 15.19: Beispiel eines Zweiweg-Adapters

Ein Client nutzt Variablen als ConstraintVariable und der andere als StateVariable und der Zweiweg-Klassenadapter passt die Variablen für beide Clients an.
(nutzt wieder Mehrfachvererbung wie bei Klassenadapter üblich)

15.8.5 Implementierung

(in VL nicht behandelt)

- Steckbare Adapter (pluggable Adaptors) (=Objektadapter) sind auf drei verschiedene Weisen implementierbar:
  Allen drei Weisen gemeinsam ist:
  „schmale" Stelle für Adaptee finden, d.h. Schnittstelle finden, die aus kleinster Menge der Methoden besteht, die für Adaptierung notwendig sind. (erleichtert die Adaptierung = Programmierung des Adaptors)

  - Verwendung abstrakter Methoden:
    im Adaptor werden alle Methoden, die in „schmaler" Schnittstelle enthalten sind abstrakt deklariert und erst später in einer konkreten Adaptor-Sub-Klasse konkret implementiert. Alle weiteren Methoden, die in allen Adaptor-Sub-Klassen gleich sind, werden sofort in Adaptor-Schnittstelle implementiert

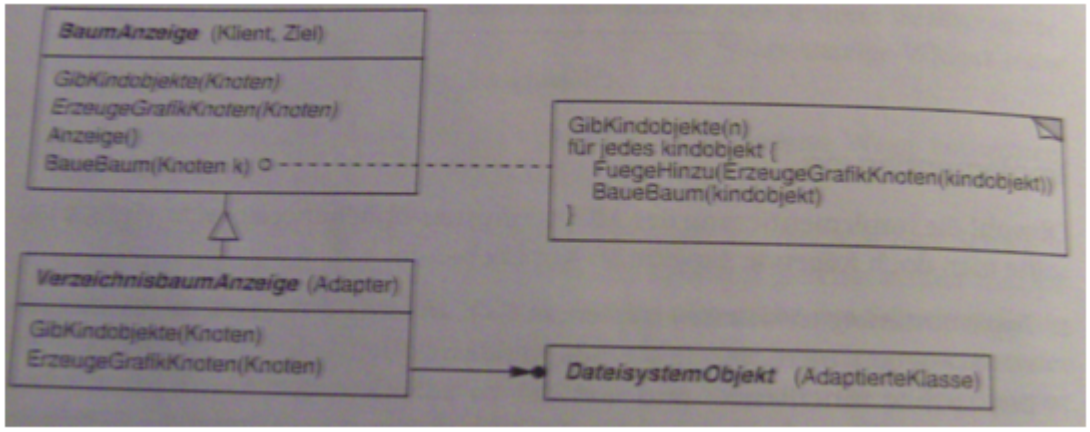


Abbildung 15.20: Implementierung von pluggable Adaptors/Object Form Adaptors durch abstrakte Methoden

  - Verwendung von Delegationsobjekten:
    Target leitet Requests an Delegationsobjekt weiter. Clients von Target können Art der Adaptierung steuern, indem sie entsprechendes Delegationsobjekt zu Verfügung stellen
    - in dynamisch Typisierten Sprachen:
      * Schnittstelle um Delegationsobjekt beim Adaptor zu registrieren. Adaptor leitet nun seine Requests an Delegationsobjekt weiter, welches nun die eigentliche Adaptierung des Adaptees übernimmt

    * Delegationsobjekt muss Funktionalität der „schmalen" Schnittstelle haben, es ist aber nicht nötig neues Interface zu implementieren
  - In statisch typisierten Sprachen: (u.a. Java)
    * Schnittstelle für Definition des Delagationsobjekts nötig
    * Delagationsobjekt muss „schmale" Schnittstelle die zu Adaptierung nötig ist implementieren
    * Adaptor muss nun Objekt des Delegationsobjekts haben

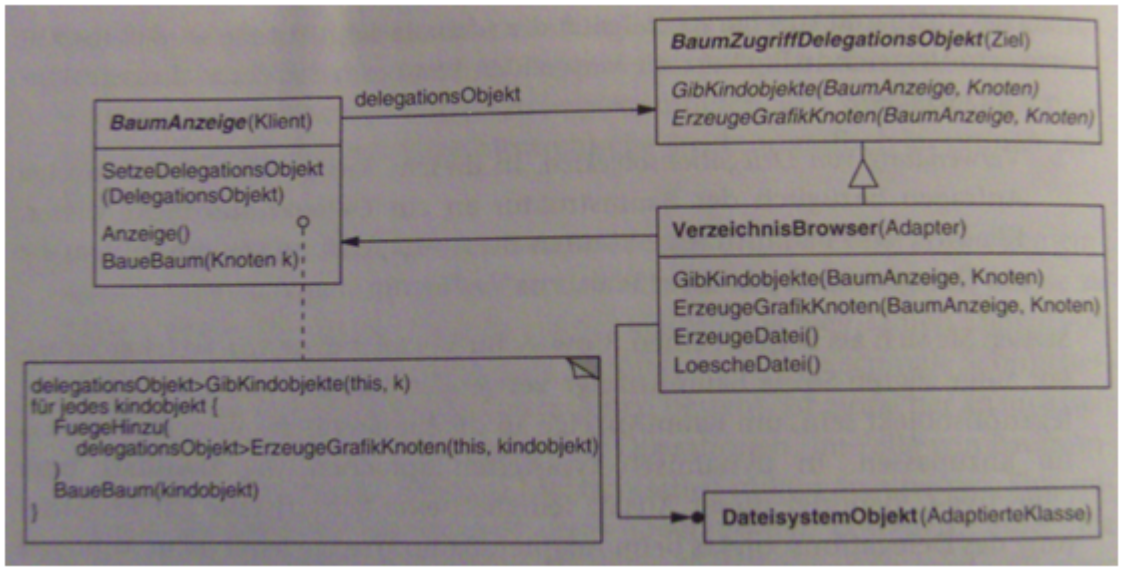


Abbildung 15.21: Implementierung von pluggable Adaptors/Object Form Adaptors durch Delegationsobjekten

15.8.6 In Zusammenhang stehende Patterns

- „Proxy Pattern":
  - siehe „Proxy Pattern"
- „Decorator Pattern":
  - definiert Stellvertreter oder Ersatzobjekt für ein anderes Objekt, verändert aber nicht dessen Interface, sondern implementiert sogar das gleiche
- „Bridge Pattern" (Brückenmuster):
  - (in VL nicht behandelt)
  - entkoppelt Interface von dessen Implementierung, sodass beide unabhängig voneinander geändert werden können

15.9 Strategy Pattern

Momentan noch kein Inhalt vorhanden

15.10 Proxy Pattern

15.10.1 Ziel (Benutze wenn)

- Kontrollieren des Zugriffs auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreters (= Proxy)
  Client soll aber nicht merken, dass es nicht das eigentliche Objekt ist!

  → möglicher Grund: Verlagerung des Erzeugen eines Objekts bis es wirklich nötig wird, beispielsweise beim Laden von Bildern in Dokumenteditor bei dem Dokumentöffnung schnell von statten gehen soll (Bild erst laden, wenn sichtbar und zuvor Proxy vorschalten)

  → so ist jede unterschiedliche Behandlung der einzelnen Elemente für Client (hier: Dokumenteditor) sichtbar, aber man umgeht andere Probleme – wie z.B. Performance-Probleme (VERSTECKEN, DASS OBJEKT ERST ON-DEMAND KREIERT)

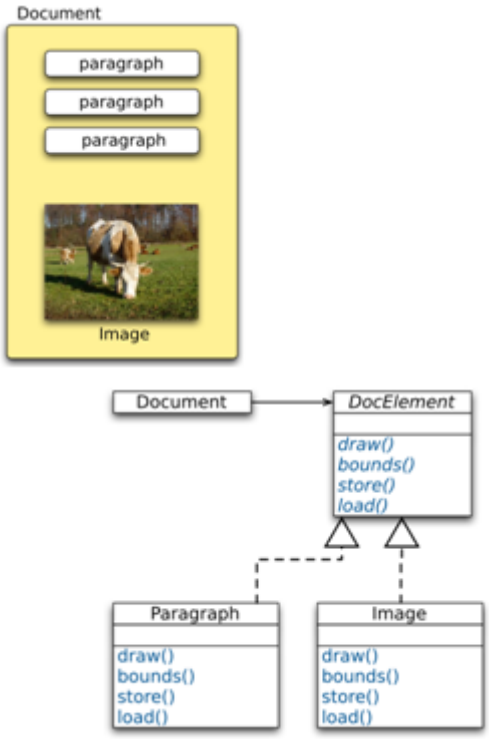
  → Möglichkeit guten Trade-off zwischen der Objekt-Unterstützung (alle Bilder beim öffnen sofort laden) und des Bestrebens des schnellen Öffnens eines Dokumentes zu finden

  Aus Sicht des Clients verhält sich Proxy genauso wie eigentliches Objekt

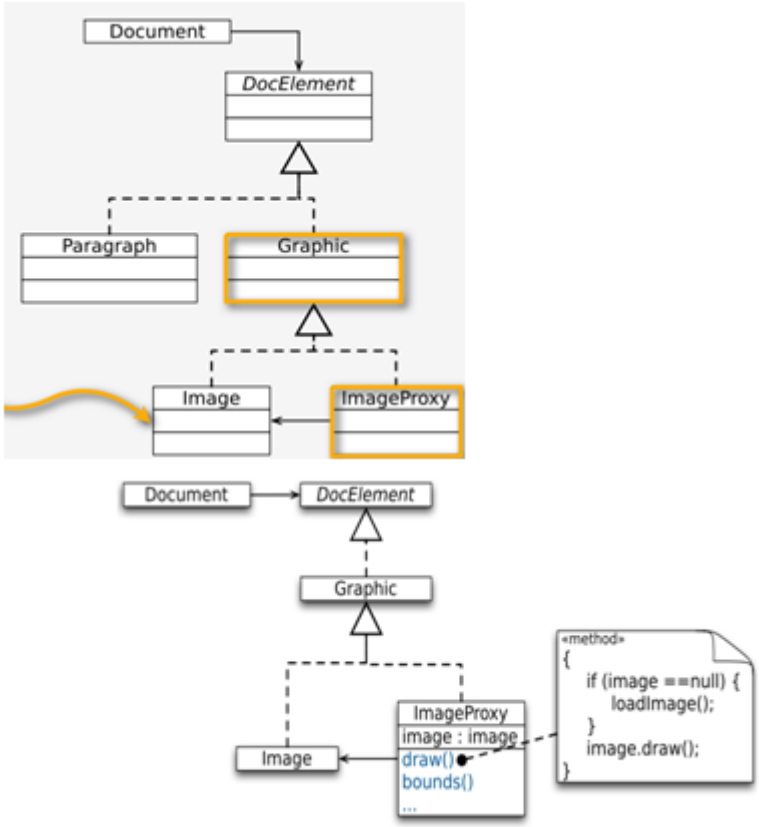
15.10.2 Beispiel

Dokumenteditor:

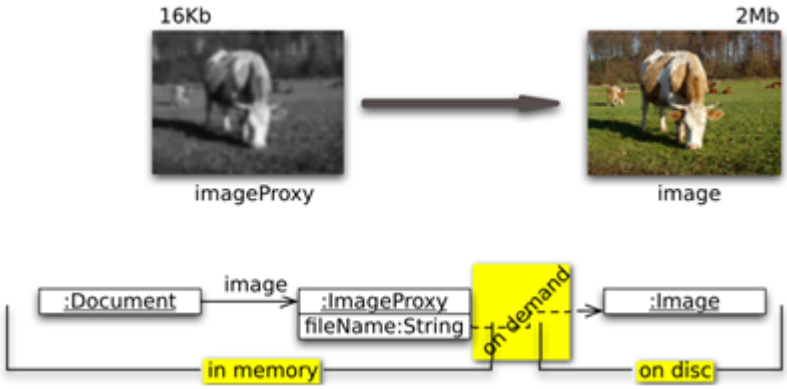
- **Implementierung, die Probleme – z.B. bzgl. Performance – verursachen könnte**



- **- Implementierung mit „Proxy Design Pattern”:**



Genauere Struktur des Anfragen-Weiterreichens von Proxy an konkretes Objekt:



15.10.3 Struktur

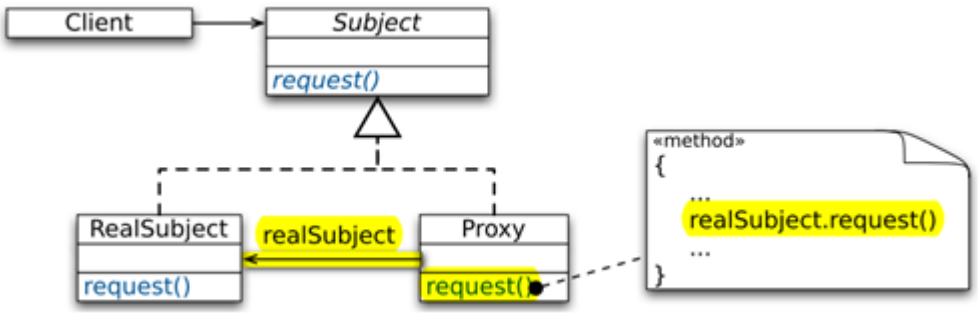


Abbildung 15.22: Struktur des Proxy Patterns

- Proxy:
  - Implementiert das selbe Interface, wie das reale Objekt
    Client ist nicht bewusst, dass es sich um ein Proxy-Objekt und nicht, um das reale handelt!
  - Instanziiert reales Objekt wenn es benötigt wird, bspw. wenn Client Proxy dazu auffordert sich anzuzeigen, reicht Proxy Display-Request an reales Objekt weiter und dieses wird angezeigt
    Behält Referenz auf reales Objekt bei, um anschließende Requests des Clients an reales Objekt weiterleiten zu können
  - Verschiedene Arten eines Proxy's:
    (siehe zusätzlich: „Entwurfsmuster": „Proxy Pattern" – Proxy-Arten)
    - **Vitual Proxies** (Platzhalter, wie im obigen Beispiel)
      * Instanziieren teure Objekte, wie Bilder, nur on demand
      * Können zusätzliche Informationen über eigentliches Objekt zwischenspeichern (nötig für Platzhalterfunktion des Proxies)
    - **Smart References** (zusätzliche Funktionalität)
      * Ersetzen einfache Pointer und bieten zusätzliche Funktionalität an
      * Beispiele:
        - Schließen/Lösen von Referenzen auf Objekte, die von mehreren Threads genutzt werden
        - Zählen von Referenzen – z.B. für Ressourcen- Management (Garbage-Collection, Observer Aktivität)
    - **Remote Proxies** (Transparent machen Verteiltheit von Proxy und eigentlichem Objekt – wie z.B. in Netzwerk – sichtbar)

        * Stellen Interface zur Kommunikation im eigentlichen Objekten in anderem Adressraum dar
        * Kommunikation erfolgt, indem Request mit Argumenten in Proxy erstellt, an eigentliches Objekt weitergeleitet und dessen Antwort über Proxy am Client gesendet wird
    - **Protection Proxies** (Verwalten von Zugriffsrechten)
        * Auch: Schutz Proxies
        * Überprüfen, dass Aufrufer die zum Ausführen des Befehls notwendigen Zugriffsrechte besitzt

- Subject:
    - Definiert gemeinsames Interface von RealSubject und Proxy, sodass Proxy überall dort eingesetzt werden kann wo eigentlich RealSubject benötigt
- RealSubject:
    - Definiert eigentliches Objekt, welches durch Proxy repräsentiert wird

15.10.4 Konsequenzen

„Proxy Pattern" führt Ebene der Indirektion beim Zugriff auf ein Objekt ein. Diese zusätzliche Indirektion kann vielfältig genutzt werden. Dies hängt wiederum vom Proxy-Typ ab:

- Remote-Proxy kann Tatsache verstecken, dass sich eigentliches Objekt in anderem Adressraum befindet
- Virtual Proxy kann Optimierungen durchführen – wie z.B. das Erzeugen eines Bildes erst on demand
- Protection Proxies und Smart Reference Proxies ermöglichen Durchführung zusätzlicher Verwaltungsaufgaben, wenn auf Objekt zugegriffen wird
- Weitere positive Konsequenz:
  Copy-on-Write:
    - Ist mit Erzeugung on demand verwandt
    - Kopieren eines großen, komplizierten Objekts kann teuer sein und ist unnötig, wenn das Objekt nie verändert wird
    - Verwendung eines Proxy kann man sicher stellen, dass Preis für Kopie nur gezahlt wird, falls Objekt sich wirklich verändert

15.10.5 Implementierung

- Proxy muss nicht immer dynamischen Typ des eigentlichen Objekts kennen, wenn Proxy mit allen realSubject-Klassen über ein Interface kommunizieren kann, d.h. Proxy kann mit allen realSubject-Klassen einheitlich umgehen (Funktionalität des Proxies unterscheidet sich nicht zwischen den realSubject Klassen)
  - Falls dies doch der Fall sein sollte, muss für jede realSubject-Klasse ein Proxy angelegt werden (= sie müssen die konkreten Klassen kennen)
  - Falls Proxies Objekte der jeweiligen realSubject-Klasse erzeugen müssen, müssen diese die konkrete Klasse kennen
- Überlegen wie Zugriff auf realSubject-Objekt gestaltet wird
  - Manche Proxies müssen realSubject-Objekt referenzieren, unabhängig davon, ob es in Hauptspeicher oder auf Festplatte
    → adressraumunabhängige Objekt-Referenz nötig (Beispiel: Datei-Name)

15.10.6 In Zusammenhang stehende Patterns

- „Adaptor Pattern":
  - siehe nachfolgendes Pattern
  - im Gegensatz zu Proxy bietet Adaptor andere Schnittstelle zum Objekt, das es anpasst
- „Decorator Pattern":
  - (in VL nicht behandelt)
  - Auch bekannt als: Wrapper Pattern
  - Ziel (Benutze wenn):
    - Objekt dynamisch um Zuständigkeiten erweitern
    - Flexible Alternative zur Sub-Klassen-Bildung, um Klassen zu erweitern

15.11 Façade Pattern

Momentan noch kein Inhalt vorhanden

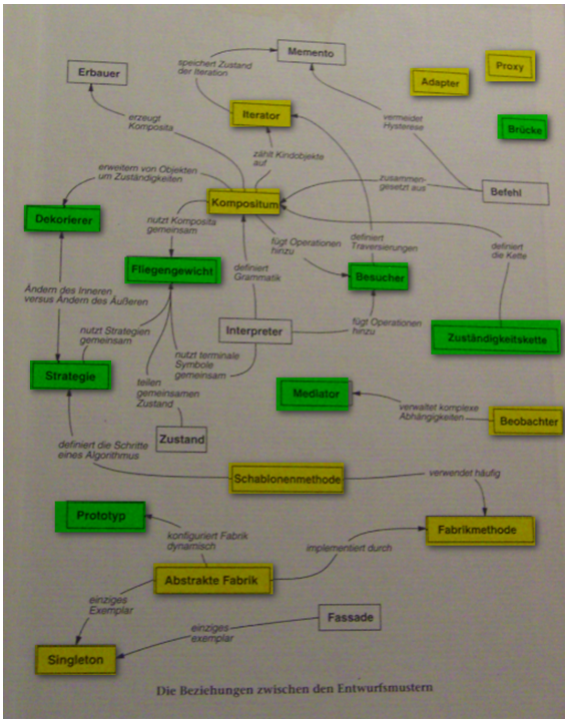


Abbildung 15.23: Design Patterns – Übersicht

15.12 Design Patterns – Übersicht

in VL besprochen mit Patterns aus der VL in Zusammenhang stehende Patterns

Kapitel 16

Frameworks

16.1 Definition „Framework”

- Erfasst die gesamte Struktur einer Familie von Anwendungen
- Erfasst die allgemeine Interaktion (Kontroll-Fluss)
- Lässt aber konkrete Implementierungsdetails bzgl. des Verhaltens außen vor
- Nutzen des „Template Method”, „Abstract Factory” und des „Command” Patterns ist typisch beim implementieren von Frameworks
- Verkörpern Design-Einblicke
- Verden charakterisiert durch die Inversion des Kontroll-Flusses Das Framework koordiniert und ruft den vom Nutzer gelieferten Code auf

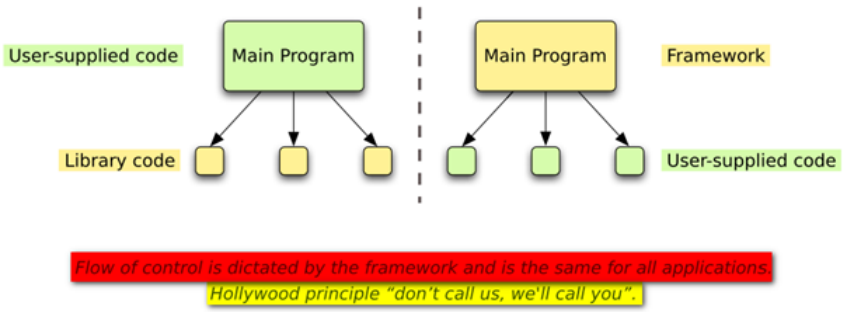


Abbildung 16.1: Kontrollfluss bei Framework-Designs

16.2 Was ist ein Framework einer objektorientierten Anwendung?

= Sammlung von zusammenarbeitenden Klassen, die zusammen eine allgemeine Lösung für eine Familie von Anwendungen definieren und noch bzgl. einer speziellen zu implementierenden Anwendung angepasst werden muss

16.3 Das Framework verkörpert...

- Modellierte Konzepte einer Domäne
- Ein allgemeines Design einer Anwendung:
  - Verteilung der Verantwortlichkeiten auf die verschiedenen Klassen
  - Reduzierte Koppelung der Klassen durch Interfaces
  - Bieten von Raum für mögliche Laufzeit-Objekt-Konfigurationen
  - Erlaubter Kontrollfluss zwischen Objekten
- Teilweise Implementierung der Anwendung
  - Allgemeine, wiederverwendbare Funktionalität
  - Definierte Erweiterungspunkte (=Hot-Spots) und Reuse-Contracts

16.4 Vorteile von Frameworks

- Anwendungsspezifische Details werden erst bei der Anpassung des Frameworks implementiert
  (= präzisieren der Hot-Spots)
- Anwendungsprogrammierer müssen sich nur mit anwendungsspezifischen Dingen auseinander setzen
  (allgemeine Struktur ist bereits vorgegeben)
- Design- und Code-Wiederverwendung
  (Design-Reuse wird als wichtiger angesehen)
  - Weder Code noch Design müssen nochmals geprüft werden
    (außer, natürlich, das Framework hat einen Fehler)

16.5 Profit durch die Anwendung eines Frameworks/ Stärken eines Frameworks

- Modularität veränderliche Implementierungsdetails werden durch stabile Interfaces gekapselt
- Wiederverwendbarkeit
  - Framework macht sich Domain-Wissen und frühere Bemühungen von erfahrenen Entwicklern zu Nutze, um erneutes Implementieren und Validieren von geläufigen Lösungen zu verhindern
  - Stabile Interfaces stärken die Wiederverwendbarkeit, indem sie allgemeine Komponenten definieren, die wiederverwendet werden können, um neue Anwendungen zu implementieren
- Erweiterbarkeit
  - Werden explizit Methoden angeboten, die es Anwendungen erlauben die stabilen Interfaces zu erweitern (= hook methods)
- Inversion des Kontollflusses

16.6 Schwächen eines Frameworks

- Entwickeln eines qualitativ hochwertigen, erweiterbarem und wiederverwendbarem Frameworks ist schwerer als das Entwickeln einer komplexen Software
- Lernen des Verwendens eines OO-Anwendunmgs-Frameworks benötigt eine beachtliche Anstregung
  (dauert oft Monate bis man Framework effizient nutzen kann)
- „Integrierbarkeit"
  (Integration verschiedener Frameworks kann schwierig werden)
- „Debug-Fähigkeit"
  (aufgrund des Mangels an explizitem Kontrollfluss und der Inversion des Kontrollflusses kann es schwierig sein des Framework zu debuggen)

16.7 Es können 3 Framework-Typen unterschieden werden

16.7.1 Black-Box Framework

- Der Client kennt keine Implementierungsdetails des Frameworks

- Aufgrund dessen erfolgt die Anpassung durch:
  - Eingeben von Parametern oder dem Einhängen von vorgegebenen Klassen, die eine komplette Implementierung von Framework-Interfaces darstellen → Anwendungsspezifische Funktionalität wird durch Parametrierung und Einhängen vorgegebener Klassen bewirkt
  - Flexibilität ist auf die Anzahl der, durch die Framework-Entwickler, vorgesehenen Auswahlmöglichkeiten beschränkt

16.7.2 White-Box Framework

- Angepasst durch Sub-Klassen-Bildung bzgl. der durch das Framework gegebenen Klassen und durch Anbieten konkreter Implementierungen für die abstrakten Klassen und Methoden, die als „Hooks"/Haken markiert sind (= erweitern der Framework Hot-Spots)
- Andere Teile des Frameworks, die nicht als Hot-Spots beabsichtigt sind, sind auch anpassbar (hohe FlexibilitÄt, aber Vorsicht muss walten gelassen werden)

16.7.3 Grey-Box Framework

- Frameworks entwickeln sich typischer Weise von White-Box zu Black-Box Frameworks
  - Wenn noch keine Hot-Spots bekannt sind, ist Vererbung ein probater Adaptionsmechanismus
  - Wenn die Variierungspunkte und typischen Entscheidungen für diese bekannt sind, dann ist Parametrierung ein probates Mittel

Diese Art von Frameworks siedelt sich zwischen Black- und White-Box Frameworks an – somit heissen sie Grey-Box Frameworks Grey-Box Frameworks sind Frameworks die beides nutzen – Verfeinerung (Vererbung u.ä.) und Parametrierung(= Vorgabe von Parametern aus gegeben Menge möglicher Parameter-Typen)

Die meisten Frameworks sind Grey-Box Frameworks.

Kapitel 17

UML - Unified Modelling Language

17.1 Logical Architekture and Package Diagrams

- Verwaltung/Aufteilen der SW Klassen in Packages, Subsysteme und Layers(Schichten)
- Layer: eine grobe Aufteilung von Klassen, Packages und Subsystemen, die in einem größerem Zusammenhang bzgl des Gesamtsystems stehen
- Vorgehensweise für Schichtenarchitektur:
  - Abstraktionskriterien definieren
  - Anzahl der Abstraktionsebenen festlegen
  - Schichten benennen und Aufgaben benennen
  - ein Interface für jede Schicht
  - in individuelle Schichten gliedern
  - Fehlerbehandlungsstrategie entwerfen(idR in der untersten Schicht)
- SW Problem lösen zusätzl. Schicht
- Performanz Problem Schicht entfernen
- Visualisierung mithilfe des Package-Diagramm
- **UML Package Diagramm**
  - Ein Paket kann alles enthalten: Klassen, andere Pakete und Use Cases
  - ein Paket ist eine Sammlung von Elementen die sich den gleichen Namespace teilen
  - Pakete können verschachtelt werden
  - Pakete repräsentieren Namespaces

- Abhängigkeiten werden mithilfe von Standard-UML-Abhängigkeiten dargestellt
- Beispiel für Package-Nesting

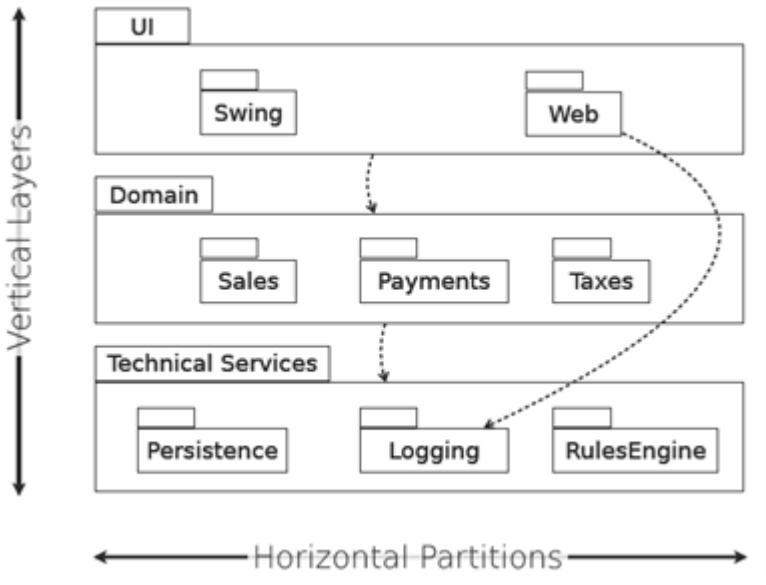


Abbildung 17.1: Beispiel für Package Nesting

- ¡¡access¿¿ ist ein private import

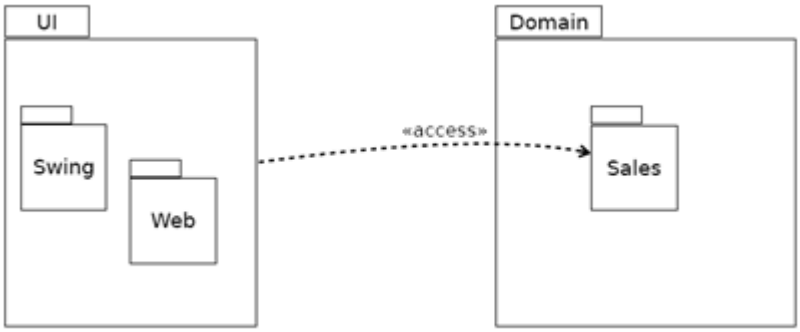


Abbildung 17.2: Beispiel für Package Nesting

- ¡¡import¿¿ ist ein public import
- Paketdiagramm in Java und UML-Abhängigkeit

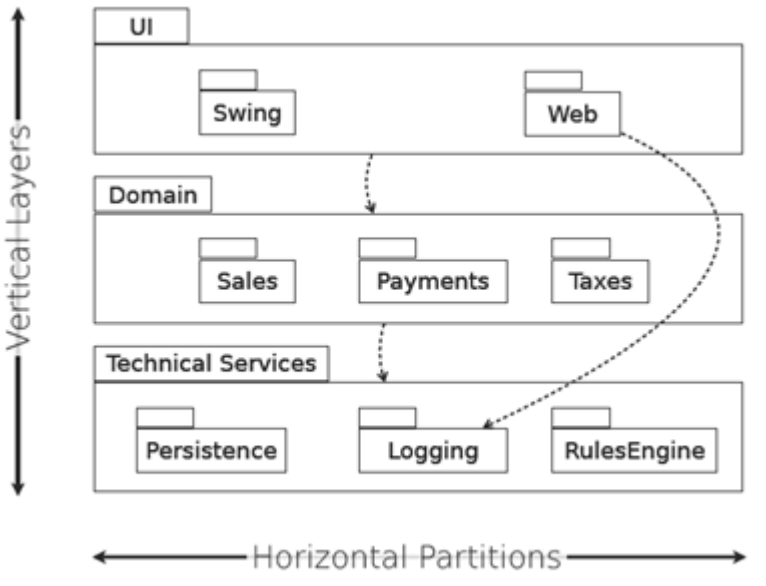


Abbildung 17.3: ¡¡access¿¿ als public Import

- Bei der Darstellung mit UML-Abhängigkeiten muss die Verbindung zu nested-Packages nicht gekennzeichnet werden
- Beispiel:
  - Java-Abhängigkeit
  - UML-Abhängigkeit

- Schichtenmodell
  - Es handelt sich immer dann um ein striktes Schichtenmodell wenn von jeder oberen Schicht auf alle darunterliegenden zugegriffen werden kann

17.2 Interaction Diagrams

- Interaktionsdiagramme dienen zur Visualisierung der Interaktion über Nachrichten zwischen den Objekten
- Sie werden für die dynamische Objekt-Modellierung genutzt
- Typen:
  - Sequenzdiagramm
  - Kommunikationsdiagramm

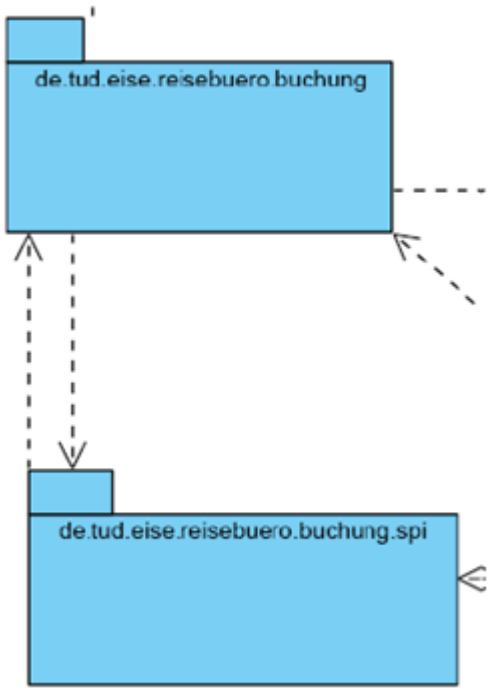


Abbildung 17.4: Package Diagram mit Java-Abhängigkeiten

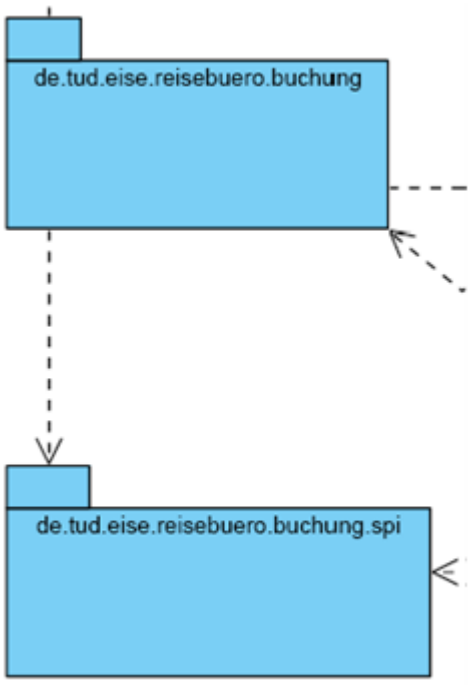


Abbildung 17.5: Package Diagram mit UML-Abhängigkeiten

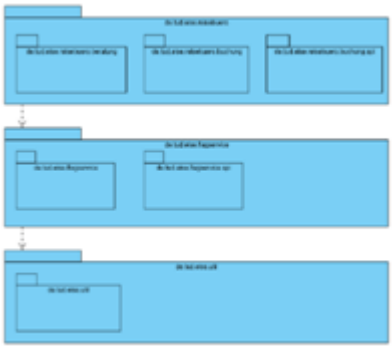


Abbildung 17.6: Modellierung eines Schichtenmodells durch ein Package Diagram

17.2.1 Sequence Diagrams

Das Sequenzdiagramm ist ein Verhaltensdiagramm, genauer eines der vier Interaktionsdiagramme in UML. Es ist eine grafische Darstellung einer Interaktion und beschreibt den Austausch von Nachrichten zwischen Ausprägungen mittels Lebenslinien.

17.2.2 Beispiel:

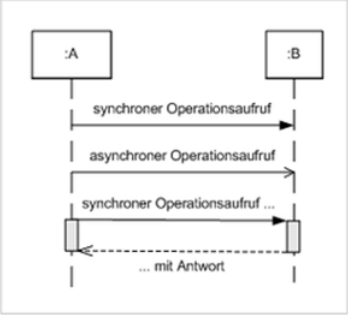


Abbildung 17.7: Beispiel 1: Sequence Diagram

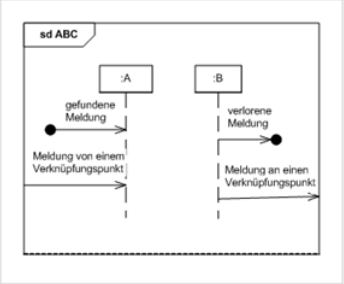


Abbildung 17.8: Beispiel 2: Sequence Diagram

17.2.3 Aufruftypen:

Synchrone
Hier wird genau eine Folge von Ereignisauftritten modelliert: ¡ S1 E1 , S2 E2¿

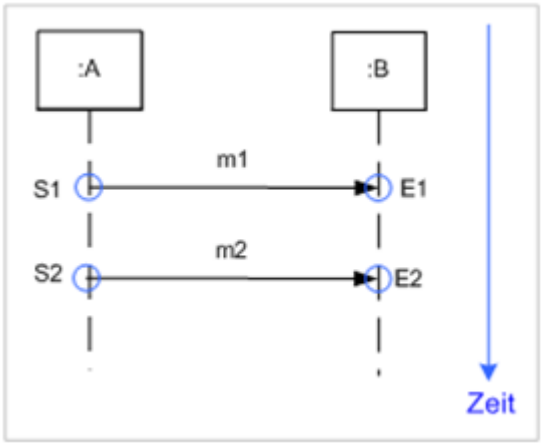


Abbildung 17.9: synchroner Aufruf

Asynchrone
Hier werden zwei mögliche Folgen vonEreignisauftritten modelliert: ¡ S1 E1 , S2 E2¿ & ¡ S1 S2 , E1 E2¿
Bei Asynchronen aufrufen muss der aufruf nicht erst abgeschlossen sein bevor der nächste starten kann.

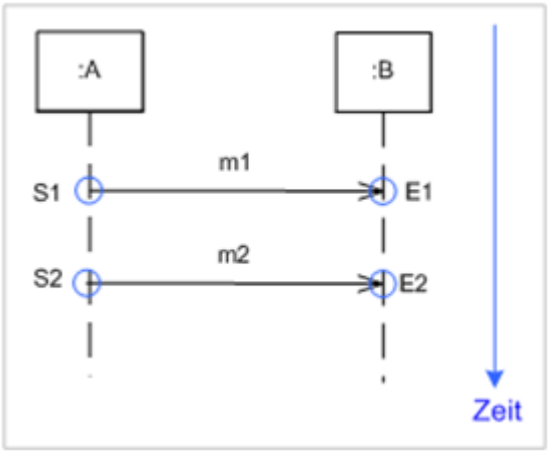


Abbildung 17.10: asynchroner Aufruf

17.2.4 Nachrichten:

- Eine Nachricht an sich selbst

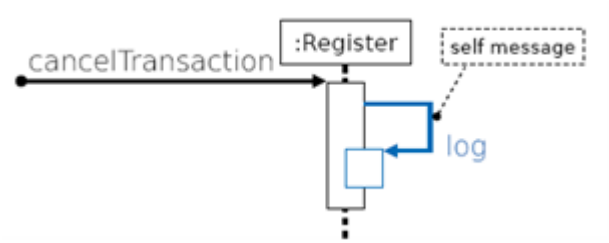


Abbildung 17.11: Self-Message im Sequence Diagram

- Return-Message

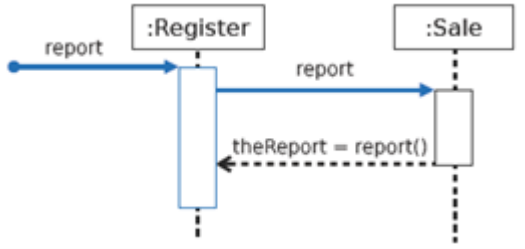


Abbildung 17.12: Return-Message im Sequence Diagram

17.2.5 neue Objekte:

- Neu erstellte Objekte auf der Höhe platzieren, auf der sie erstellt wurden

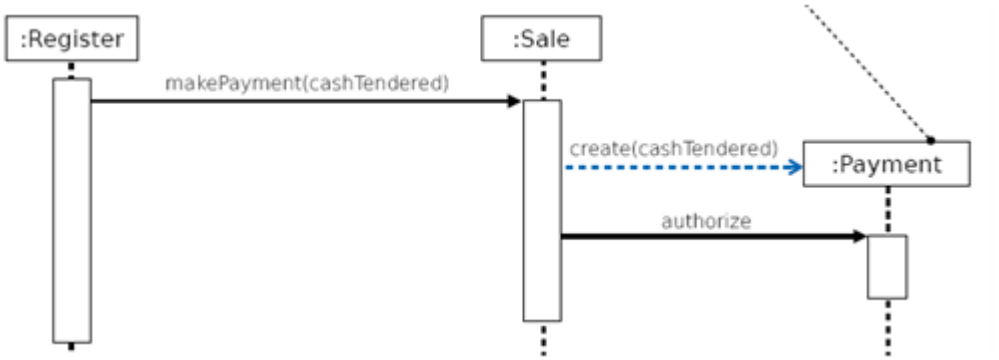


Abbildung 17.13: Plazieren neu erstellter Objekte im Sequence Diagram

17.2.6 Fragmente:

- Alt Alternative Ablaufmöglichkeit
- Loop Schleife. Iterationen in Interaktionen
- Opt Optionaler Teil einer Interaktion/Optionale Teilabläufe
- ref um andere Diagramme zu integrieren

Nicht in SE besprochene:

- Assert unabdingbare Interaktion
- Break Ausnahmefälle (Abbruchfragment)
- Consider Filter für wichtige Nachrichten
- Critical Nicht unterbrechbare Interaktionen
- Ignore Filtern für unwichtige Nachrichten
- Neg Ungültige Interaktionen
- Par Nebenläufige Teile einer Interaktion
- Seq Abläufe, die von Lebenslinien und Operanden abhängen
- Strict Abläufe, die nicht von Lebenslinien und Operanden abhängen

System Sequence Diagrams (SSD)

- Zeigt ein bestimmtes Szenario aus einem Use Case
  - Die Ereignisse die externe Akteure generieren
  - Ihre Reihenfolge
  - Systemevents
- Das System wird als eine Blackbox angesehen
- SystemSequenzdiagramme werden als Input fr Objektdesign genutzt
- Kommunikationspartner und Massagereihenfolge werden dargestellt

17.2.7 Communication Diagrams

Beispiele

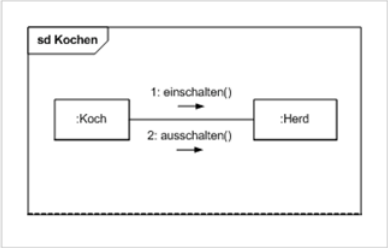


Abbildung 17.14: Beispiel 1: Communication Diagram

- :Koch entspricht einer unbekannten Instanz der Klasse Koch
- :Herd entspricht einer unbekannten Instanz der Klasse Herd

- :A entspricht einer unbekannten Instanz der Klasse A
- myB:B entspricht einer Instanz myB der Klasse B

Erläterungen

- Ein Link ist ein Verbindungsweg zwischen zwei Objekten (⇒ eine Instanz der Verbindung)
- Jede Nachricht zwischen den Objekten wird mit einer Meldung versehen und ein kleiner Pfeil zeigt die Richtung

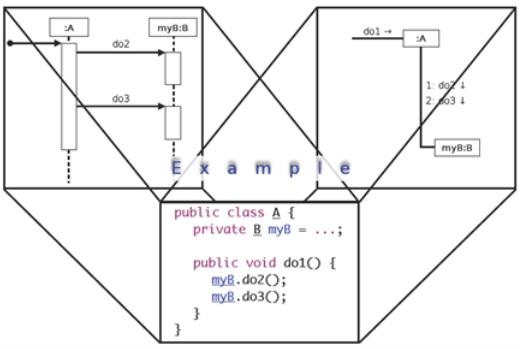


Abbildung 17.15: Beispiel 2: Communication Diagram

Abbildung 17.16: Nachrichten im Communication Diagram

- Self-Message

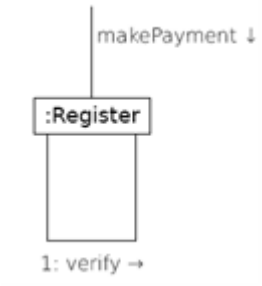


Abbildung 17.17: Self-Message im Communication Diagram

- Sequenznummern zeigen die Reihenfolge
- Erzeugen einer Instanz

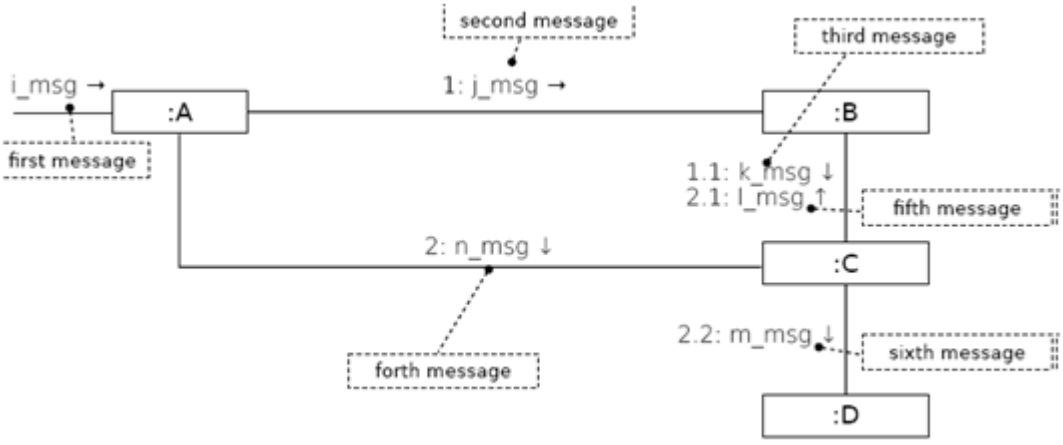


Abbildung 17.18: Sequenznummern im Communication Diagram

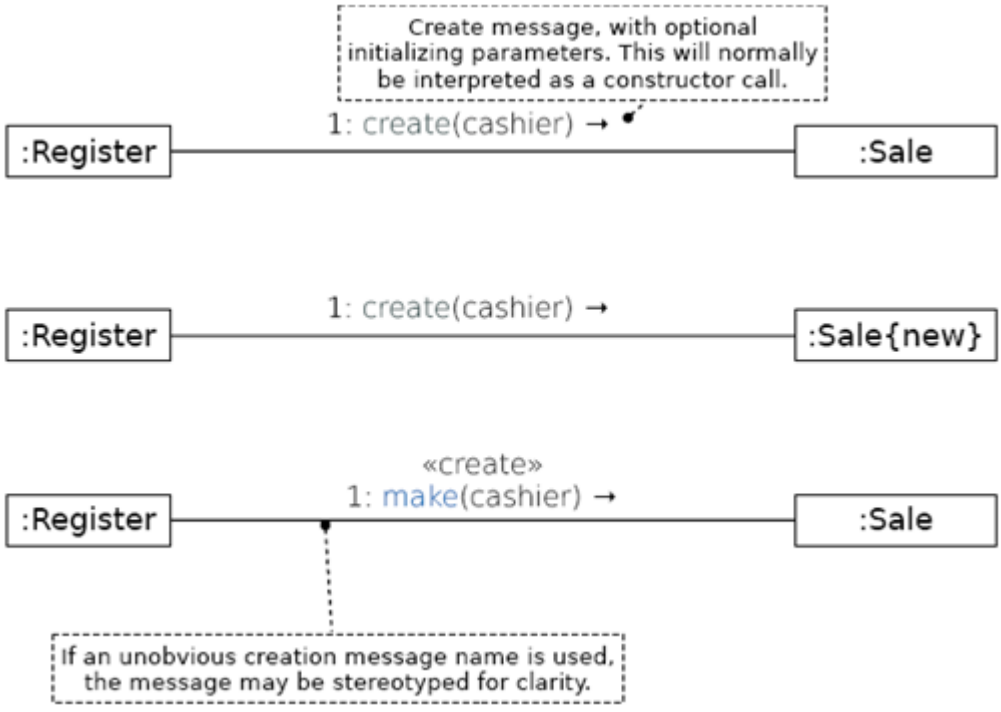


Abbildung 17.19: Erzeugen einer neuen Instanz im Communication Diagram

Typ	Stärken
Sequence Diagram	zeigt deutlich den zeitlichen Ablauf ,viele detaillierte Notationsmöglichkeit
Communication Diagram	platz sparend neue Objekte können flexibel hinzugefügt werden in 2 Dim

Tabelle 17.1: Personen, die zu diesem Skript beigetragen haben. Die Angabe der Personen erfolgt in chronologischer Reihenfolge.

17.2.8 Vergleich von Sequence und Comminication Diagram

17.3 Class Diagrams

17.3.1 Beispiel

Person	Name der Klasse
- Name: String	Attribute
- Vorname: String	attribute: Typ
+ getName(): String	Operationen
+ getVorname(): String	operation(parameter): Erg_Typ

Tabelle 17.2: Personen, die zu diesem Skript beigetragen haben. Die Angabe der Personen erfolgt in chronologischer Reihenfolge.

17.3.2 Kennzeichnung der Sicherheit von Attributen und Operationen:

- + public
- # protected
- - private
- ˜package (Innerhalb des Pakets sichtbar)

17.3.3 Klassenarten:

- Abstrakte:
  Eigenschaft = abstract oder Klassenname Kursiv geschrieben Von einer abstrakten Klasse kann keine Instanz angelegt werden.
- Interface:
  Schlüsselwort = Interface Ein Interface stellt eine Schnittstelle dar.
- Aktive :
  Doppelte Linie rechts und links

Abbildung 17.20: aktive Klasse

17.3.4 Beziehungen

- Assoziation
  Eine Assoziation beschreibt eine Beziehung zwischen zwei oder mehr Klassen. An den Enden sind häufig Multiplizitäten vermerkt.

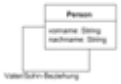


Abbildung 17.21: Assoziation

- Aggregation
  Beziehung zwischen einem ganzen und seinen Teilen. (Vorlesung Student)
- Komposition
  (Gebäude Raum). Die Komposition ist ein Spezialfall der Aggregation. Sie bildet den Fall ab, bei dem die Teile nicht ohne das Ganze existieren können. (Existenzabhängigkeit)

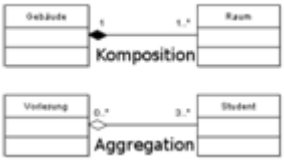


Abbildung 17.22: Komposition und Aggregation

- Generalisierung
  Gerichtete Beziehung zwischen einem generelleren und einem spezielleren Classifier. Instanzen des speziellen sind auch Instanzen des generellen. **Eine Generalisierung kann nicht zwischen zwei Paketen bestehen.**
  Durchgezogenerstrich: bedeutung extends
  Gestrichelterstrich: bedeutung implements

Abbildung 17.23: Generalisierung

17.4 Object Diagrams

17.4.1 Beispiel

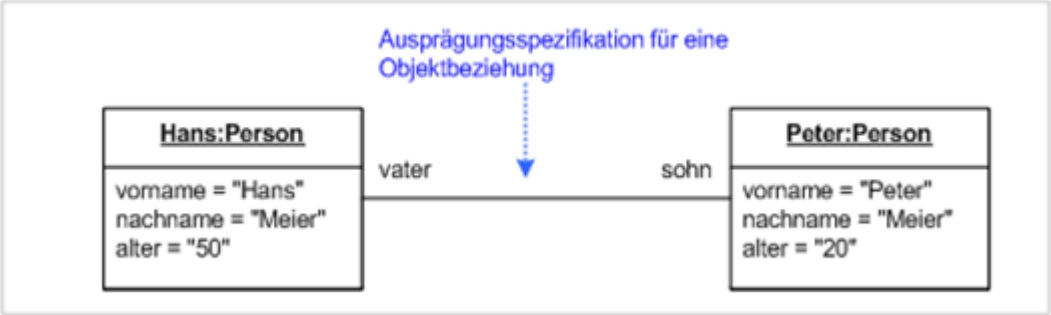


Abbildung 17.24: Beispiel: Object Diagram

17.4.2 Erläuterungen

- Modelliert den Zustand des Systems zu einem bestimmten Zeitpunkt
- Werden oft dazu eingesetzt Klassendiagramme zu überprüfen (Test ob alle Verbindungen im Klassendiagramm enthalten sind)
- Es ist möglich Instanzen von Abstrakten Klassen zu modellieren
- Es werden ausschließlich Objektinstanzen und deren Beziehungen modelliert

- Schreibweise:

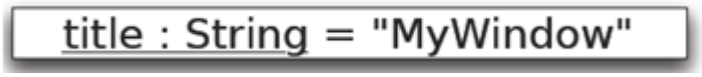


Abbildung 17.25: Object Diagram: Schreibweise

  - Unterstrichen Objektname:Klassenname = Wert

- Einschub einer Variable kann kann als slot kenntlich gemacht werden

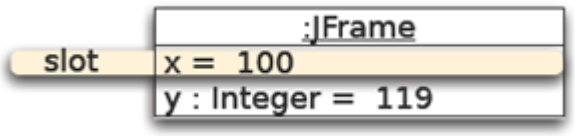


Abbildung 17.26: Object Diagram: Einschub einer Variablen

- Immer nur die Elemente darstellen die für die Dokumentation auch wirklich relevant sind
- Überblick zur Notation:

17.5 Use Case Diagrams

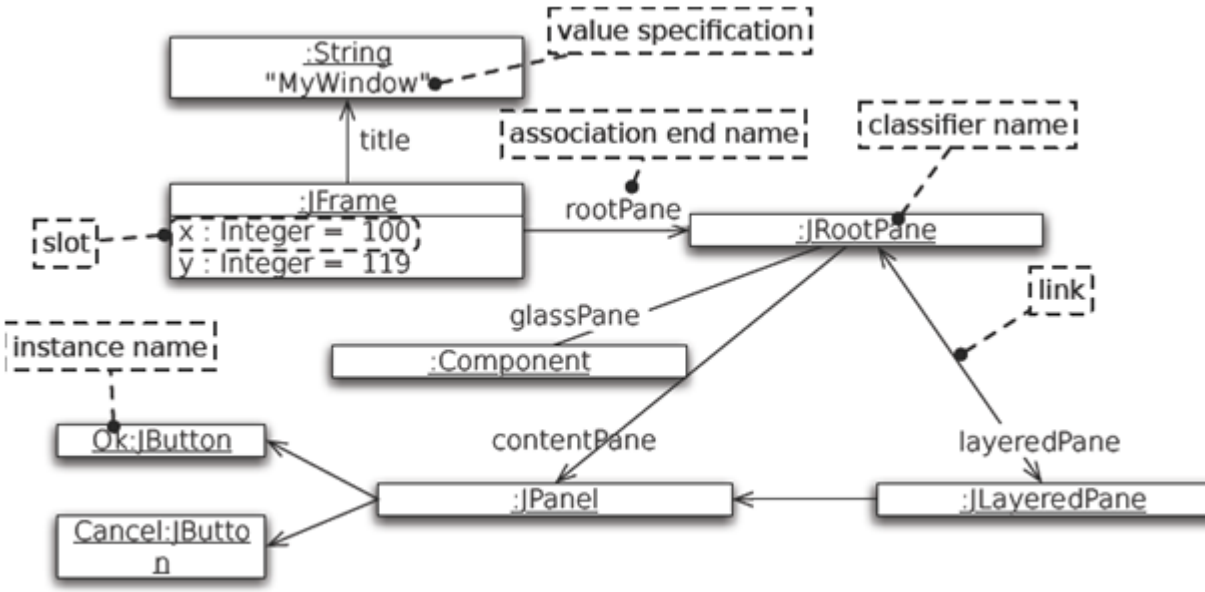


Abbildung 17.27: Überblick zur Notation eines Object Diagrams

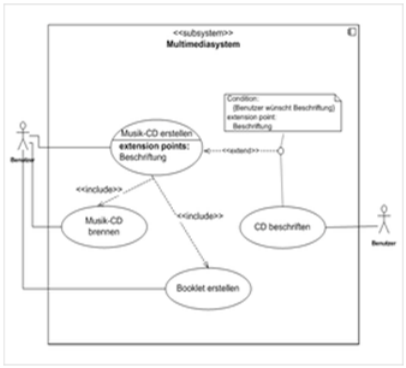


Abbildung 17.28: Beispiel 1: Use Case Diagram

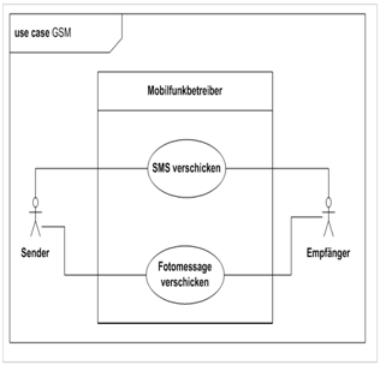


Abbildung 17.29: Beispiel 2: Use Case Diagram

17.5.1 Elemente:

- Systemkontext:
  stellt die jeweiligen Systemgrenzen dar.

Abbildung 17.30: Systemkontext

- Akteure:
  Können Personen wie Systeme darstellen

Abbildung 17.31: Akteur/Actor

- Anwendungsfälle:
  Stellen einen anderen Anwendungsfall dar und dienen dazu um mehrere Anwendungsfälle zu verknüpfen. Benötigen eine Beschreibung (Beispielsweise in einem Kommentar oder in einer separaten Datei.)

Abbildung 17.32: Anwendungsfall/Use Case

17.5.2 Beziehungen:

- Assoziation/Kommunikation:
  Strich von Akteur zur Anwendung
- Multiplizität:
  Strich von Akteur zur Anwendung mit Anzahl(wobei default bei Akteur 1 ist)
- Generalisierung

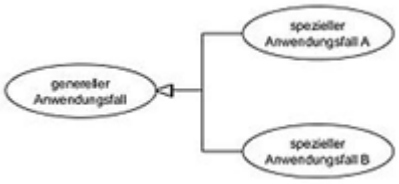


Abbildung 17.33: Generalisierung

- Include-Beziehung:
  Anwendungsfall A beinhaltet Anwendungsfall B

Abbildung 17.34: Include-Beziehung

- Extend-Beziehung:
  Anwendungsfall A erweitert Anwendungsfall B

17.5.3

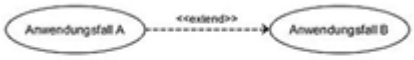


Abbildung 17.35: Extend-Beziehung

Kapitel 18

Klausur

18.1 Allgemeines

Die Klausur wird kurz nach dem Ende der Vorlesungszeit statt finden. Alle Inhalte der Vorlesung sind klausurrelevant.

Die Bearbeitungszeit wird 90 Minuten sein. *Eine vollständige Bearbeitung der Klausur wird in der zur Verfügung stehenden Zeit nicht möglich sein.* Suchen Sie sich am Anfang der Klausur die Aufgaben heraus, die für Sie das beste Zeit-Nutzen Verhältnis haben.

Als Hilfsmittel sind beliebige schriftliche Unterlagen zugelassen (Neudeutsch: *Open-Book-Klausur*); es sind keine elektronischen Hilfsmittel zugelassen.[1]

18.2 Struktur

18.2.1 Multiplechoicefragen

Es gibt typischerweise eine Aufgabe mit Multiplechoicefragen. Bei diesen Fragen handelt es sich meist um Wissensfragen bzw. Fragen, die durch einfache Transferleistung/kurzes Nachdenken, beantwortet werden können. Beispiele für Multiplechoicefragen finden Sie in Tabelle 18.1.

Aussage	Richtig	Falsch
Durch den Einsatz von *Class-Responsibility-Cards* ist sichergestellt, dass ein gutes Design erstellt wird.		
Inversion of Discipline ist charakteristisch für Frameworks.		
...		

Tabelle 18.1: Beispiele von Multiplechoicefragen

[1] Elektronische Übersetzungshilfen sind nur nach expliziter Rücksprache zugelassen.

18.2.2 Allgemeine Wissensfragen

Es wird eine Aufgabe mit Wissensfragen geben. Beispiele:

- Benennen Sie drei allgemeine Entwurfsprinzipien. Begründen Sie für jedes Prinzip, warum es sich um ein allgemeines Entwurfsprinzip handelt und warum es nicht ausschließlich für die Entwicklung objektorientierter Programme gilt.
- ...

18.2.3 Software Design

Aufgaben zum Thema Software Design erfordern meist die Beurteilung eines gegebenen Designs (unabhängig von konkreten Design Patterns oder Idiomen).

Eine Aufgabe wäre:

> Geben ist folgendes Design:
>
> ```
> class Student
>
> class BachelorStudent extends Student
>
> class MasterStudent extends Student
> ```
>
> Ist dieses Design sinnvoll? Begründen Sie Ihre Antwort! Wenn Sie nicht genügend Informationen haben sollten, um eine Entscheidung treffen zu können, dann geben Sie an wovon Ihre Entscheidung ggf. abhängt.

18.2.4 Use Cases

Es gibt typischerweise eine Aufgabe, bei der ein einfaches Anwendungsfalldiagramm zu erstellen ist. Darüber hinaus gibt es meist eine Aufgabe, die das Erstellen eines "fully-dressed" Use Cases verlangt.

18.2.5 UML Klassendiagramme

Es gibt meist eine Aufgabe, im Rahmen derer ein Klassendiagramm zu erstellen oder zu übersetzen ist.

18.2.6 Testen und Testabdeckung

Eine Beispielaufgabe zum Thema Testen und Testabdeckung wäre:

> Die folgende Java-Implementierung einer Methode, zur Partitionierung einer Liste (hier ein Array) von double Werten, soll getestet werden.

```
int partition(double[] a, int left, int right) {
        int i = left - 1;
        int j = right;
        while (true) {
                while (a[++i] < a[right])
                        ;
                while (a[right] < a[--j])
                        if (j == left)
                                break;
                if (i >= j)
                        break;
                swap(a, i, j);
        }
        swap(a, i, right);
        return i;
}
```

Bewerten Sie die Testabdeckung des Testfalls $a = \{12.0, 8.3\}, left = 0, right = 1$ in Hinblick auf Anweisungsabdeckung (statement coverage). Geben Sie für jede Anweisung an, ob diese durch den Testfall abgedeckt wird.

18.2.7 Design Patterns

Fragen zum Thema Design Patterns fallen in die folgenden Kategorien:

- Ein Stück Code oder ein UML Diagramm ist gegeben und Sie müssen erkennen welche Variante eines Design Patterns umgesetzt wurde und welche Klassen welche Rollen inne haben.
- Geben ist ein bestimmtes Designproblem und Sie müssen entscheiden welches Design Pattern Sie zur Lösung einsetzten wollen. Darüber hinaus ist es oft erforderlich die Lösung – unter Verwendung des vorgeschlagenen Patterns – zu skizzieren.
- Es gibt ein partielles Design und Sie müssen dieses erweitern.

Anhang A

Abkürzungsverzeichnis

Abkürzung	
O	
OOA/D	Object-oriented Analysis and Design (*Objektorientierte Analyse und Entwurf*)
U	
UML	Unified Modeling Language

Anhang B

Kleines SE Wörterbuch

Sowohl die deutschsprachige Literatur als auch (deutschsprachige) Softwareentwickler verwenden häufig englische und deutsche Begriffe – Rund um das Thema Softwareentwicklung – wild gemischt. Dieses kleine Wörterbuch soll dabei helfen, sich im Dschungel der Begriffe besser zurecht zu finden. Fett markiert sind die Begriffe, die nach subjektiver Meinung, die gängigeren sind.

Im Rahmen der Vorlesung Einführung in Software Engineering betrachten wir die Namen der Entwurfsmuster als Eigennamen und übersetzen diese nicht. D.h. im Rahmen der Vorlesung verwenden wir z.B. immer den Namen "Template Method" und niemals "Schablonenmethode" verwenden.

Deutsch	Englisch
A	
Anforderungsmanagement	Requirements Engineering
Anwendungsfall	**Use case**
E	
Entwurf	**Design**
Entwurfsmuster	**Design Pattern**
R	
Risikomanagement	Risk Management
S	
Softwaretechnik	**Software Engineering**
Z	
Zusammenhalt	Cohesion

Anhang C

Bonusregelung

C.1 Allgemeines

Der maximal zu erreichende Bonus entspricht einer kompletten Notenstufe. D.h. wenn durch die Übungen der volle Bonus erreicht wurde, und aus der Klausur heraus die Note 2,3 erreicht wurde, dann wäre die Endnote 1,3. Der Bonus kann zum Bestehen der Klausur führen.

C.2 Erlangen des Bonuses

Durch die Bearbeitung eines Übungsblatts können immer zwischen null und zehn Bonuspunkte erreicht werden. Die genaue Anzahl der Bonuspunkte, die pro Übungsblatt erreicht werden können, ist abhängig von der Komplexität der konkreten Aufgabenstellung und geht immer eindeutig aus dem Übungslatt hervor. Für die meisten Übungen wird es zwischen acht und zehn Bonuspunkte geben. Der maximal zu erreichende Bonus ergibt sich somit aus der Summe der Bonuspunkte über alle Übungsblätter.

Sie können davon ausgehen, dass das letzte Übungsblatt optional sein wird. Es ist somit möglich auch dann 100% des Bonusses zu erreichen, wenn Sie im Laufe des Semesters nicht bei allen Übungen 100% erreicht haben. Es ist nicht möglich mehr alls 100% zu erreichen.

Durch aktives Beitragen zum Skript können bis zu fünfzehn weitere – optionale – Bonuspunkte erworben werden. Sowohl für originäre Beiträge, als auch für die Überarbeitung bestehenden Textes können Bonuspunkte erworben werden. Pro 400 Worte gibt es einen Bonuspunkt.

Originäre Beiträge Für neue Beiträge erhalten Sie einen Bonuspunkt, wenn es sich um einen qualitativ hochwertigen, originären, und signifikanten Beitrag handelt. Ein qualitativ hochwertiger Text zeichnet sich insbesondere dadurch aus, dass alle Inhalte prägnant, präzise und korrekt sind. Darüber hinaus muss die Rechtschreibung fast einwandfrei sein. Für den Fall, dass

der Text grundsätzlich verwendbar ist, aber einer größeren Nachbearbeitung bedarf, werden nur 60% der Anzahl Worte des eingereichten Textes anerkannt. Für Inhalte, die direkt aus den Vorlesungsfolien übernommen werden, oder die zu starke Redundanz mit existierenden Beiträgen aufweisen, gibt es keine Bonuspunkte. Dies gilt insbesondere für Grafiken, da diese vom Veranstalter frei zur Verwendung zur Verfügung gestellt werden. Sollten Sie Ideen für eigene Visualisierungen haben, dann sprechen Sie dies mit dem Veranstalter ab. In jedem Fall müssen alle Inhalte, die Sie nicht selber erstellt haben, entsprechende Referenzen aufweisen. Im Falle von Plagiatismus werden Ihnen alle bisher erreichten Bonuspunkte aberkannt und Sie werden von der weiteren Teilnahme ausgeschlossen.

Überarbeitungen Änderungen des bestehenden Textes, die der Qualitätssteigerung dienen, werden je Änderung mit einem Wort verrechnet. Die Anzahl der Änderungen wird mit dem Tool dwdiff[1] berechnet.

Die Ermittlung der Bonuspunkte ergibt sich am Ende des Semester auf der Basis der Addition der gewichteten Anzahl Worte aller Ihrer Beiträge.

C.3 Verrechnung mit der Klausur

Der von Ihnen relativ erzielte Bonus wird in Klausurpunkte umgerechnet und auf die erzielten Klausurpunkte aufaddiert. Die Gesamtzahl der Klausurpunkte bestimmt dann die Endnote. D.h. wenn in der Klausur zum Beispiel zehn Punkte mehr erreicht werden müssen, um sich eine komplette Notenstufe zu verbessern (2,0 → 1,0), und 90% der maximal möglichen Bonuspunkte erreicht wurden, dann ist der effektive Bonus neun Klausurpunkte.

Sie müssen mind. 25% der Klausurpunkte erreichen, um den Bonus zu erhalten.

C.4 Gültigkeit des erreichten Bonus

Der Bonus ist gültig für das aktuelle Wintersemester und das darauf folgende Sommersemester. Der Bonus verfällt insbesondere auch dann nicht, wenn die Klausur am Ende des Wintersemesters – trotz Verrechnung des Bonuses – nicht bestanden wird.

[1] dwdiff -s <OLD TEXT> <NEW TEXT>

Anhang D

Autoren

Die folgenden Personen haben Beiträge zu dem Skript geleistet.

Autor	Kontakt Beiträge
Michael Eichberg	eichberg(at)informatik.tu-darmstadt.de “überall”
Kathrin Ballweg	ballweg(at)rbg.informatik.tu-darmstadt.de 15, 16
Kristin Rammelt	rammelt(at)rbg.informatik.tu-darmstadt.de 3, 4, 5, 6, 8, 14
Melanie Weiland	mweiland(at)rbg.informatik.tu-darmstadt.de 17
Barbara Zöller	bzoeller(at)rbg.informatik.tu-darmstadt.de 2, 9, 10, 11, 12, 13
Heiko Guckes	h_guckes(at)rbg.informatik.tu-darmstadt.de 7

Tabelle D.1: Personen, die zu diesem Skript beigetragen haben. Die Angabe der Personen erfolgt in chronologischer Reihenfolge.

Literaturverzeichnis

[Dua] Charles Duan. Understanding git conceptually. `http://www.eecs.harvard.edu/~cduan/technical/git/`.

[Gru86] Dick Grune. Concurrent versions system, a method for independent cooperation. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, pages 364–370. IEEE Transactions on Software Engineering, 1986.

[Qua09] Nick Quaranto. Git started with git. `http://www.slideshare.net/qrush/git-started-with-git`, 2009.

[Roc75] Marc J. Rochkind. The source code control system. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, pages 364–370. IEEE Transactions on Software Engineering, 1975.

[tea05] GitHub team. Git reference. `http://gitref.org/`, 2005.

[Tic85] Walter F. Tichy. Rcs a system for version control. pages 637 – 654. John Wiley & Sons, Inc. New York, NY, USA, 1985.