Software Engineering – WS 2017/18

Held by Richard Bubel

Foliensatz 1: Intro + CRC-Cards

Was ist Software Engineering?

➔ Software-Krise

SE umfasst

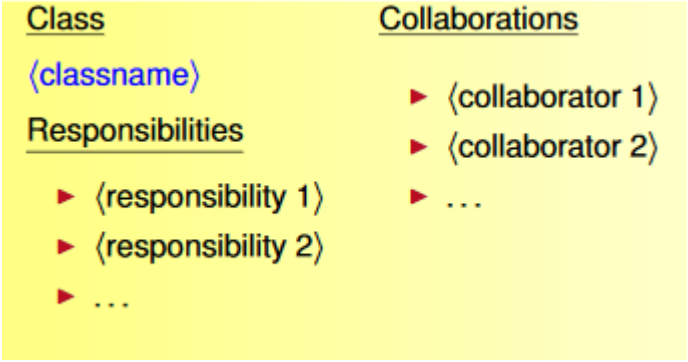
- Softwaredefinition (*Software Requirements*)
- Softwaredesign
- Testen der Software
- Wartung der Software
- Softwarekonfigurationsverwaltung
- Entwicklungsprozess
- Entwicklungstools und Methoden
- **Qualität**
- Software Engineering ←→ System Engineering
    - Systemrelevante Aktivitäten

CRC-Cards

Class: Name der Klasse

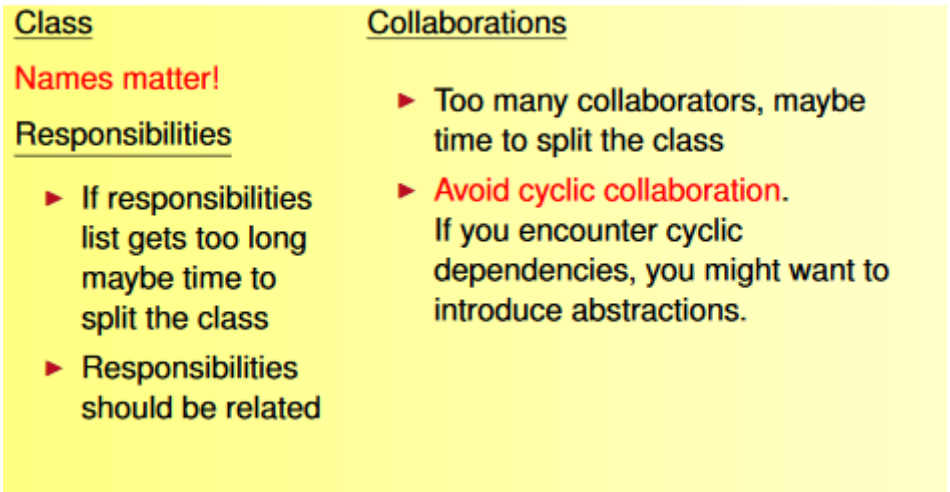
Responsibilities: Probleme die diese Klasse lösen soll

Collaborators: Klassen/Akteure, mit denen diese Klasse arbeitet



Verwendung:

1. Identifizieren der Klassen, für jede Klasse eine Karte
2. In mehreren Durchläufen verschiedene Szenarien durchgehen und
    a. Operationen der Klassen identifizieren
    b. Entsprechende Klassen finden



Foliensatz 2: Software Engineering Process

Software-Prozess-Modelle – Allgemein

Ein Software-Prozess Modell (SPM) ist eine abstrakte Beschreibung einer Herangehensweise zum Endprodukt. Weiterhin kann ein SPM Aktivitäten enthalten, die Teil des Software Project Managements sind.

Software Projekt Management

- Proposal Writing
    - Projektmanager müssen in der Lage sein, sich gut in Schrift und Worten ausdrücken können → Schnittstelle Entwickler Team und Kunde
- Projekt- und Zeitplanung
- Kostenschätzung
- Projektmonitoring (Überwachung) und Reviews
- Etc

Probleme bei Softwareprojekten

- Produkt ist nicht greifbar
    - *Projektmanager können nicht direkt sehen wie es um den Fortschritt steht*
    - *Er muss sich auf die Entwickler und deren Dokumentation verlassen*
- Es gibt keine Standards in der Softwareentwicklung
    - *In anderen Bereichen ist der Entwicklungsprozess leichter zu verfolgen*
- Große Projekte sind Unikate
    - *Vorausbestimmen von Problemen ist sehr schwer. Ändernde Technologien machen vorherige Erfahrungen unbrauchbar*

Verglichen mit Hardwareeigenschaften, besitzt die Software besondere Merkmale

- Software hat keinen **Verschleiß**! (*Dennoch **muss** sie **kontinuierlich gewartet** werden, muss an Umgebungsänderungen angepasst werden etc.*)
- Schwer **messbar**! (*Wie definiert man **die Qualität einer Software?** Kann eine Relation zwischen Codezeilen und Qualität erstellt werden?*)
- Wie kann der Entwicklungsfortschritt gemessen werden

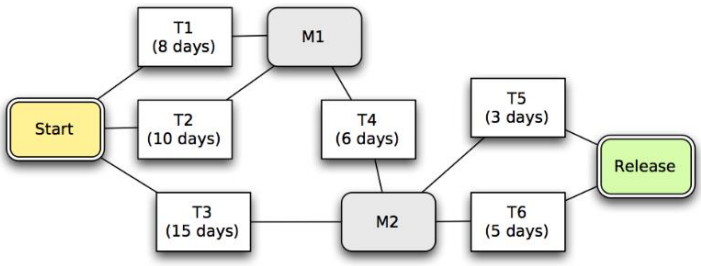
Projektplanung ist ein Iterativer Prozess

- Verschiedene Planungsmöglichkeiten

Projektplan

1. **Einführung:** Erläuterung der Ziele im Projekt
2. **Projekt Organisation:** Organisation des Entwickerteams, weiterer involvierter Personen und Ihre Rollen
3. **Risiko Analyse:** Mögliche bestehende / zukünftige Risiken für das Projekt samt ihrer Wahrscheinlichkeit
4. **Ressourcenanforderung:** Für das Projekt nötige Hard- / Software
5. **Aufschlüsselung:** Definieren von „Tasks“ die das Projekt in sich aufteilen

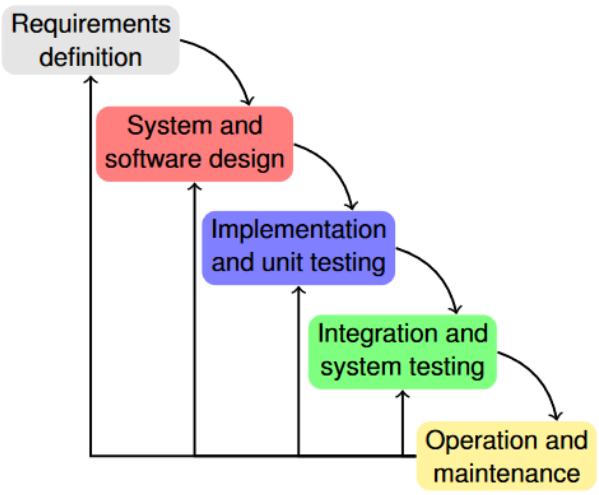
6. **Projekt (Zeit-)Plan:** Finden von Abhängigkeiten innerhalb des Projekts (Welcher Task hängt von welchem anderen ab?), Zeitschätzung der Tasks



Aktivitätendiagramm zur Projektzeitplanung – längster Pfad zum Ziel wird „Kritischer Pfad" genannt → betrifft das gesamte Projekt

Wasserfallmodell- Generisches Prozessmodell

Ablauf



1. **Requirements-Analyse**
   Beratung - Requirements festgestellen
   → als Systemspezifikation gehandhabt
2. **System und Softwaredesign**
   Systemarchitektur wird definiert
   Grundlegende Abstraktionsebenen identifizierten
3. **Implementierung und Testen von Komponenten**
   Komponenten implementierten Einzelne Teile testen + validieren
4. **Testen des gesamten Systems und Integration**
   Systemkomponenten werden als Einheit getestet
5. **Ausführung und Wartung**
   Software wird installiert und auf ihre praktische Verwendbarkeit geprüft

Schlüsseleigenschaften des Wasserfallmodells

- Das Ergebnis jeder Phase: Zusammensetzung verschiedener Produkte.
- Die nächste Phase startet, wenn vorherige beendet. In der Praxis überlappen sich diese manchmal
- Falls in einer Phase Fehler auftreten, wird die vorherige Phase wiederholt

Kritik

- Kein Iterativer Prozess
- Nachträgliche Änderung der Anforderungen schwer
- Andere Phase, anderes Team

Agiles Developement – iterative Entwicklungsmodelle

Ziel: Schnelle Entwicklung angesichts sich schnell ändernder Anforderungen

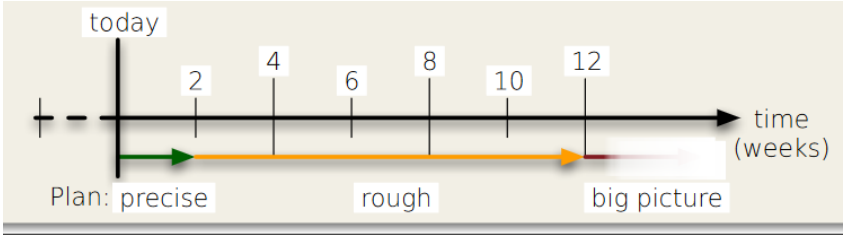
Wie wir Agilität erreichen:

- Verwenden von Methoden zur Sicherung der Disziplin, für gutes Feedback
- Verwenden von Designrichtlinien zur Sicherung der Flexibilität und Wartbarkeit
- Wissen über Design Pattern anwenden um die Balance über bestimmte Probleme zu waren
- **Stellt nicht sicher, dass die Interessengruppen auch das bekommen, was sie wollen, lediglich dass sie das Team kontrollieren → Minimaler Finanz-Input und maximaler Feature-Output**

Manifest für Agile Entwicklung

- **Mitarbeiter / Interaktion stehen über dem Prozess und den Tools**
  - Möglichst klein anfangen und nur bei Bedarf expandieren
- Die Struktur und Designwahl sollte begründet und dokumentiert sein
- Zusammenarbeit mit Kunden steht über Vertragsverhandlung
- Der Vertrag beschreibt Zusammenarbeit vom Team mit Kunden
- Ein Vertrag welcher einen Preis nennt, ist kacke

→ **Optimaler Plan zur Bewerkstelligung eines Planes**



Angemessene Reaktionszeiten bei Änderungen

Grundsätze

Höchste Priorität ist Zufriedenheit des Kunden durch frühe/schnelle Releases der Software mit kontinuierlicher Wartung

- Regelmäßig gut funktionierende Software abliefern (2 Wochen Zyklus bspw.)
- Stabile Software ist primäres Maß des Prozesses
- 30% der implementierten Funktionalität bereits fertig → 30% des Gesamtaufwandes erledigt
- Achtsamkeit über technische Vorzüge / gutes Design verbessert Agilität
- Ändernde Anforderungen willkommen heißen, auch in späten Phasen – agile Prozesse nutzen Wettbewerbsvorteil für den Kunden
- Einfachheit – Wert auf guten Code legen anstatt der Verwendung von Ad-Hoc Lösungen
- In regelmäßigen Zeitabständen besprechen, wie es bessergeht, evtl. Änderungen vornehmen
- Beste Architekturen, Requirements und Design **made by self-organized Teams**
- Die Kunden und Entwickler müssen jeden Tag bis zur Fertigstellung des Projektes zusammenarbeiten
- Agile Prozesse fördern zukunftsfähige Entwicklung. Sponsoren, Entwickler, und Endnutzer sollten ein bestimmtes Tempo auf unbestimmte Zeit einhalten können

Extreme Programming

Besteht aus einfachen und voneinander abhängigen Aufgaben

Rollenverteilung:

Customer Team Member: Dies bezeichnet den Kunden (egal ob Person oder Gruppe) welcher die Programm-Features definieren / priorisieren.

User Stories: Diese enthalten die Anforderungen, welche beim Meeting mit dem Kunden festgelegt wurden. Sie werden bloß in wenigen Worten zusammen gefasst mitsamt einer Schätzung des Aufwands. Sie werden hinterher auf Index Karten zusammengefasst.

Short Cycles: Laufende Kopie wird beispielsweise alle 2 Wochen ausgeliefert (eine Iteration); ausgelieferte Software kann in Produktion gehen, muss aber nicht. Iterationen sind zeitgebunden. Fristüberschreitungen nicht zulässig; Wenn die aktuelle Aufgabe nicht in der Iteration lösbar, werden andere Aufgaben aus der Iteration entfernt

Akzeptanz-Tests: Details der User Stories werden in diesen Test behandelt

Short Cycles →Iteration Plan: In jeder Iteration werden User Stories und ihre Eigenschaften angepasst. Kunde wählt User Stories die implementiert werden sollen. Anzahl der User Stories durch das Budget beschränkt, welches von Entwicklern gesetzt wird

Short Cycles → Release Plan: Im Beispiel sind wir in 6 Wochen fertig. Kann immer geändert werden

The Planning Game: Teilung der Verantwortung zwischen Geschäftlichem und Entwicklung. Für das Geschäft eingeteilten Leute legen fest, wie wichtig ein Feature ist, Entwickler entscheiden, wie viel dieses Feature kosten wird. Währung ist hier Zeit, wie lange werden die Entwickler brauchen?

Initial Exploration (Projektstart)

- Entwickler und Kunden versuchen signifikanten User Stories zu identifizieren. Identifizieren **nicht alle User Stories**
- Entwickler schätzen (jeder für sich) Aufwand einer User Story mittels Story Points; Wenn Story X 4 SP kostet und Story Y 8 SP, benötigt Implementierung von Story Y doppelt so viel Zeit wie Y
    - Ein wahres Maß ist die *Velocity (Zeit/Story Point)*
    - *wird immer genauer bestimmt umso weiter das Projekt fortschreitet*

Iteration Planning

- Kunde wählt Stories für eine Iteration
- Reihenfolge der Implementierung in einer Iteration ist eine technische Entscheidung
- Eine Iteration endet an festgelegten Datum, auch wenn nicht alle User Stories komplett bearbeiten wurden
- Erwartungen aller User Stories werden summiert und Velocity wird errechnet
- Die geplante Velocity für jede Iteration ist maßgebende Velocity der vorherigen Iteration

Simple Design

- Möglichst einfache Lösung für ein Problem finden
- Einfügen neuer Struktur nur dann, wenn nachweislich neue benötigt wird
- **0-Toleranzpolitik** wenn es um Code Duplikationen geht.
    - Anwenden von **Design Pattern** zur Redundanzminimierung

Testorientierte Entwicklung

- Features, die noch nicht automatisiert getestet wurden, existieren nicht
- Tests werden geschrieben von:

- Entwickler (Tests für einzelne Module)
- Kunden (Funktionsübergreifende- / Akzeptanztest

Continous Integration

- Entwickler prüfen ihren Code und integrieren ihren Code mehrmals täglich
  - Nicht-blockende Versionsverwaltung von Nöten
- Nach Integration wird das System gebaut und die Tests gestartet

Refactioring

- Ändern des Codes um „Toten Code“ zu finden / entfernen

Task Planning

- Zu Beginn jeder Iteration entwickeln Kunde und Entwickler einen Plan
- Stories werden in Teilaufgaben runter gebrochen, benötigen 4 bis 16 Stunden Zeit
- Jeder Entwickler wählt seine Aufgabe frei

Story	Time Estimate	Assigned Iteration	Assigned Release
Find lowest fare	3	2	1
Show available flights	2	1	1
Sort available flights by convenience	4		2
Purchase Ticket	2	1	1
Do customer profile	4		

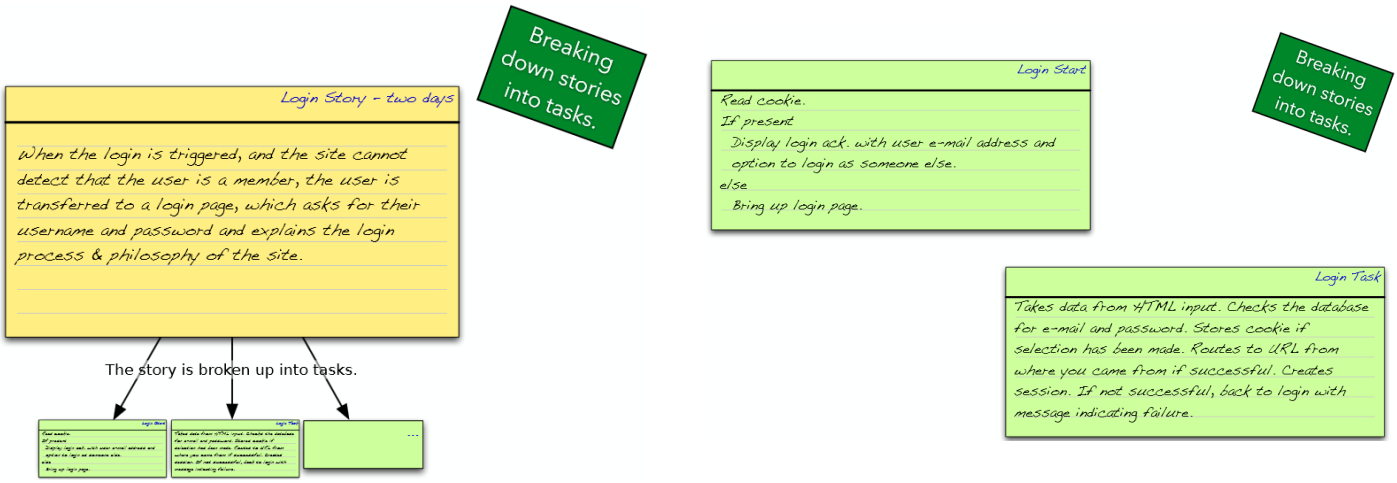
Time estimate given in story points.

Story	Time Estmated (in story points)	Time Needed (in hours)
Show available flights	2	1,5 h
Purchase Ticket	2	3,5 h

$$\text{Velocity} = \frac{\text{sum of story points}}{\text{sum of needed time per user story in iteration}} = \frac{2 \text{ story points} + 2 \text{ story points}}{1,5 \text{ h} + 3,5 \text{ h}} = 0,8 \text{ story points per hour}$$

Richtlinien für gute Stories

- Stories müssen für Kunden verständlich sein
- Jede Story muss Wert für Kunden haben
- Größe der Stories sollten so sein, dass man in einer Iteration mehrere Stories abhandeln kann
- Stories sollten unabhängig voneinander sein
- Jede Story sollte testbar sein



User Story Vorlagen

- **Langform:** Als <Rolle>, möchte ich <Ziel/Wunsch> erreichen, sodass <Erreichter Vorteil>
- **Kurzform:** Als <Rolle> möchte ich <Ziel/Wunsch> erreichen

ID	2
Name	Login
Beschreibung	Als Administrator muss ich mich am System mittels Benutzername und Passwort authentifizieren können, um Änderungen vornehmen zu
Akzeptanzkriterium	Der Dialog zum Einloggen wird korrekt angezeigt und es ist möglich sich als Administrator zu authentifizieren. Ungültige Eingaben werden ignoriert und normale Nutzer erhalten nicht die Rolle "Administrator".
Geschätzter Aufwand (Story Points)	3
Entwickler	Max Mustermann
Umgesetzt in Iteration	2
Tatsächlicher Aufwand (Std.)	12
Velocity (Std./Story Point)	4
Bemerkungen	/

Foliensatz 3: Requirements Engineering

Standards der Verwaltung von Aktivitäten

- Angebotserstellung
- Projektplanung und Fristsetzungen
- Kostenberechnung
- Projektüberwachung und Abnahme
- Wahl der Mitarbeiter
- Schreiben und Vorstellen des Berichts

Anforderungen allgemein:

- Beschreibung von
    - **Diensten** die das Zielsystem bereitstellt
        - Bspw. Autonavigation: Bereitstellung kürzester Route von A nach B
    - **Einschränkungen** die während dem Betrieb eingehalten werden müssen
        - Bspw. Verfügbarer Speicher, Verbindungen (GPS, LTE, LAN etc.)

Werden in das Pflichtenheft aufgeschrieben, oder als User Stories, Use Cases etc. vermerkt

Verschiedene Levels:

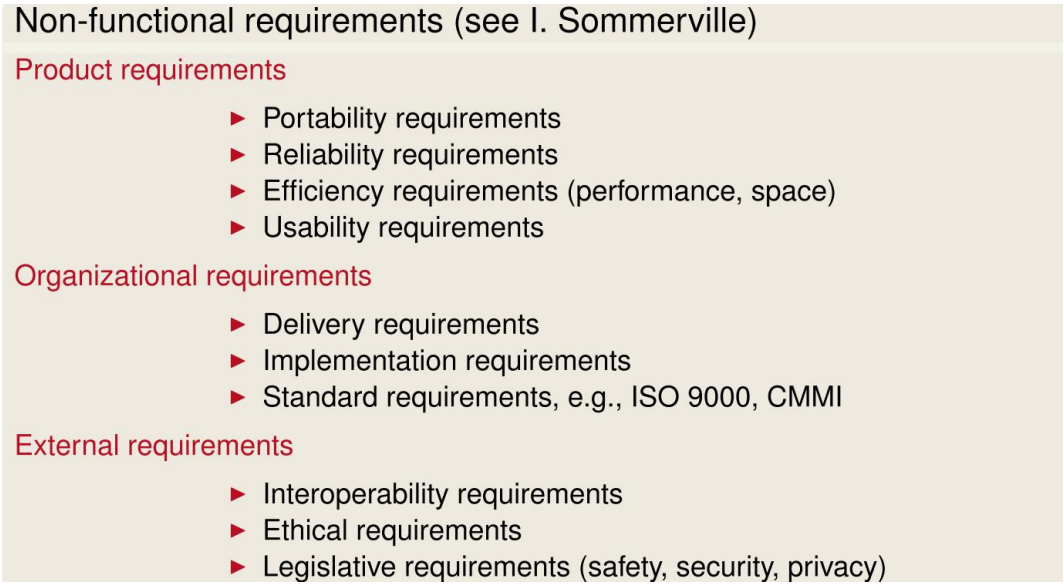
- **User Requirements**
    - Der User beschreibt – in **normaler Sprache und mit Diagrammen** – welche Dienste das System liefern soll, unter welchen Bedingungen es funktionieren muss
    - Normalerweise vom Kunden verfasst → **Lastenheft**

- **System Requirements**
  - **Beschreibt** Dienste, Funktionen und Ausführungsbedingungen des Systems **im Detail**
    - *Oft diese im Dokument für Systemanforderungen festgehalten (Funktionale Spezifikation)*
    - Teil des Vertrages → Requirements sollten präziser beschrieben werden
  - Vom Entwicklerteam/-Manager geschrieben

Aus Sicht aus des Entwicklers unterscheiden wir zwei Kategorien

- **Funktionale Anforderungen**
  - Spezifischen Aufgaben, die das System können muss, auf bestimmte Eingaben (**nicht**) reagieren soll und wie sich das System in bestimmten Szenarien (**nicht**) verhalten soll
    - *Implementierung erforderlich*
- **Nicht funktionale Anforderungen**
  - Bedingungen die an Funktionen von dem System gestellt werden
    - z.B. zeitliche Bedingungen
  - **Gelten für das System oft als allgemein**
  - **Nicht immer im Code einsehbar, aber testbar**

Schwelle zwischen Funktionalen und Nichtfunktionalen Anforderungen oft verschwommen.



- **Beobachtungen: Nicht funktionale Anforderungen (**Zum Beispiel: ***System muss sicher sein*****)**
  - Können zur Identifikation von Funktionalen Anforderungen führen
  - Sind in der Regel kritischer als Funktionale Anforderungen

Wenn die Qualität der Software erhöht und die Zeit verringert werden soll, muss auf Features verzichtet werden und die Kosten steigen an.

Typen von Anforderungen

- **Domain Anforderungen**
  - Werden durch die Anwendungsdomain abgeleitet, statt von den Nutzeranforderungen
    - In Domainspezifischer Sprache verfasst

- Von Domainexperten oft als trivial bezeichnet
- Sowohl Funktional als auch nicht Funktional

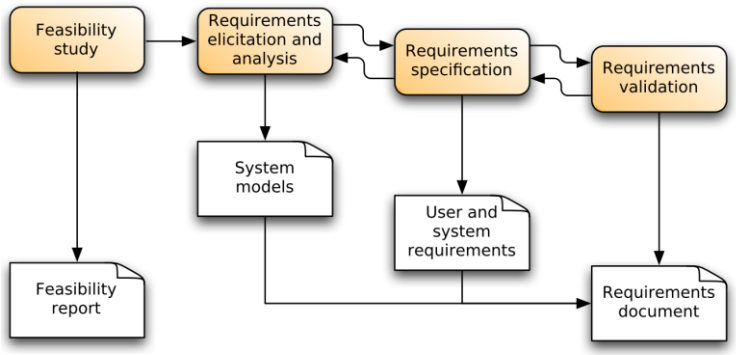
Ansätze für Requirements Engineering

Beinhaltet

- Auffinden
- Analyse
- Dokumentierung und
- Testen der Thesen, Dienste und Bedingungen

für das System. Das Dokument wird während dem Prozess erstellt und gewartet

Machbarkeitsstudie – Feasiblity Report



Der Vorteil einer Feasibility Study (Realisierbarkeitsstudie) ist, dass man hinterher anhand des Berichts sehen und gegebenfalls entscheiden kann, ob es sich lohnt, mit dem aktuellen Lastenheft den Entwicklungsprozess zu starten.

Input für die Studie:

- Vorläufige geschäftliche Anforderungen
- Zusammenfassende Beschreibung des Systems

Der Bericht sollte im nachhinein folgende Fragen beantworten können:

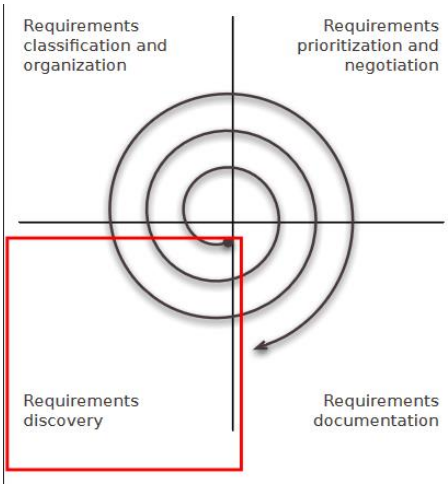
- Beteiligt sich das System daran, die Ziele des Unternehmen vollständig zu erfüllen?
- Kann das System mit aktueller Technik implementiert werden, sodass weder Zeit- noch Kostenrahmen gesprengt werden?
- Kann das System mit anderen System arbeiten, die bereits eingesetzt werden?

Standpunktbasierte Ansätze

- **Interactor Standpunkt**
  - *Nutzt das System später*
- **Indirekter Standpunkt**
  - *Interessensgruppen, welche Systemanforderungen beeinflussen, aber das System nicht direkt nutzen*
- **Domain Standpunkt**

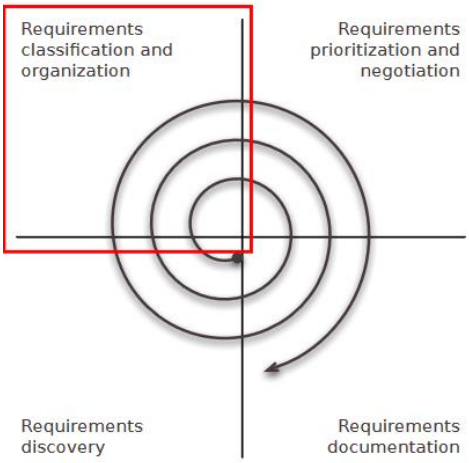
    - Domaincharakteristiken und Einschränkungen, welche Systemanforderungen beeinflussen
- Während der Anforderungsermittlung sollte man versuchen weitere Standpunkte zu finden
- Nachdem die wichtigsten gefunden wurden, sollte man mit der Requirements Discovery beginnen

Techniken zur Anforderungsermittlung



- **Interviews**
    - *Interviews sollten nur in Verbindung mit anderen Techniken durchgeführt werden. Interviewe kann Domain spezifische Sprachen benutzen, sich schwer tun, wichtige Informationen raus zu geben, oder sogar gegen das Projekt arbeiten, da sein Job auf dem Spiel steht (denkt er).*
- **Closed Interviews**
    - *Interessensgruppen beantworten vordefinierte Fragen*
- **Open Interviews**
    - *Es gibt kein festes Thema*
- **Szenarien**
    - Umfassen mögliche Interaktionen mit den System
    - Werden zu Beginn grob erläutert, dann während der Ermittlung detailliert besprochen
- **Use Cases (Anwendungsfalldiagramm)**

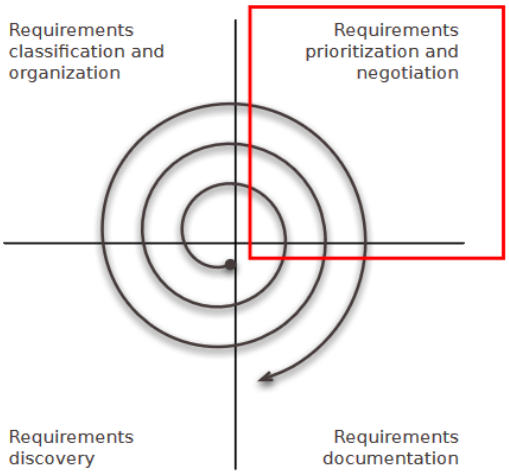
Klassifizierung und Organisation von Anforderungen



- Die unstrukturierte Sammlung von Anforderungen werden in einheitlichen Clustern sortiert
- Ein Modell : **FURPS+:**
    - **F**unctional
    - **U**sability
    - **R**eliability
    - **P**erformance
    - **S**upportability
        - Implementierung
        - Schnittstellen
        - Operationen
        - Packaging

Prioritisierung und Verhandlung

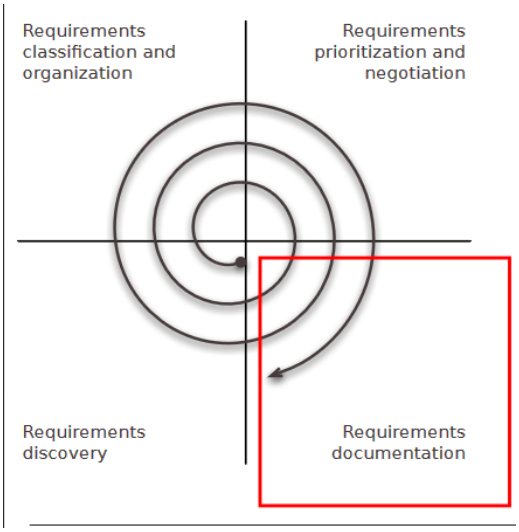
Durch Priorisierung und Verhandlungen werde mögliche Konflikte in den Anforderungen aufgelöst.



Anforderungsdokumentation

Anforderungen werden dokumentiert und später als Input für die nächste Runde genutzt.
Das Dokument kann sowohl formell als auch informell sein. → **Software Requirement Specification Document**

- Zielgruppe des SRD
  - Systemnutzer
  - Manager
  - System- Admins
- Detailgrad abhängig von
  - Systemtyp
  - Verwendetem Entwicklungsprozess
  - Wo die Anwendung / das System gebaut wurde (im Haus oder Extern)



Anforderungsvalidierung

- **Gültigkeit:**
  - Erfassen die Anforderungen die Funktionalitäten?
- **Konsistenz:**
  - Sind alle Anforderungen frei von internen Konflikten?
- **Umfassend:**
  - Werden **alle** Anforderungen und Einschränkungen erfasst?

Requirements specification document
(shortened from: ISO/IEC/IEEE 29148:2011)

1. Introduction
   a. Purpose of the requirements document
   b. Scope of the product
   c. Definitions, acronyms and abbreviations
   d. References
   e. Overview
2. General description
   a. Product perspective
   b. Product functions
   c. User characteristics
   d. Limitations
   e. Assumptions and dependencies
3. Specific requirements
4. Appendices, Index etc.

- **Realismus:**
  - Können die Anforderungen vernünftig implementiert werden?
- **Verifizierbarkeit**
  - Können alle Anforderungen anständig getestet werden?
- **Verfolgbarkeit:**
  - Kann jede Anforderung im Code aufgefunden / zurückverfolgt werden?

Anforderungen werden dokumentiert und für die nächste Runde der Spirale vorbereitet

- *Können formell, aber auch informell sein*

Anforderungsvalidierung dient um zu zeigen, dass die Anforderungen auch wirklich die Kundenwünsche abdecken *(Versucht Fehler in den Anforderungen zu finden)*

Softtware Requirements Document (Pflichtenheft) sagt aus, was der Entwickler implementieren muss

Foliensatz 4 – Use Cases

Ein Anwendungsfall ist ein Set aus verschiedenen Szenarien verbunden mit einem gemeinsamen Ziel.

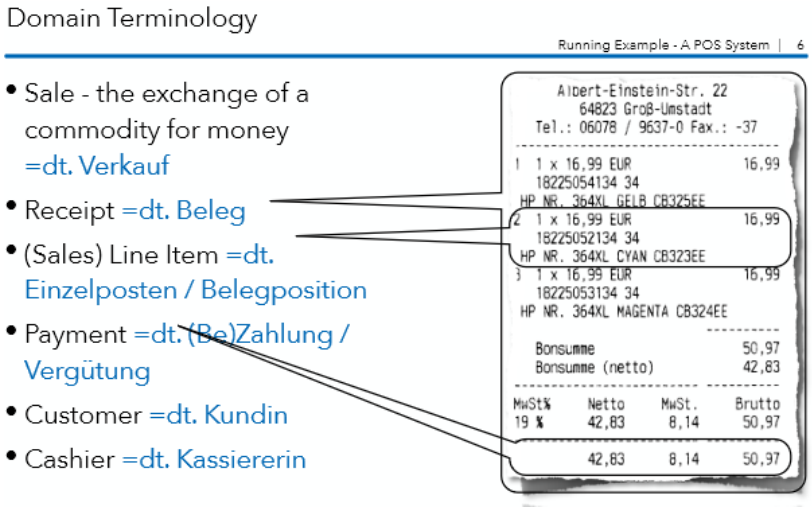
„Melde dich zu einer Klausur an“ „Bestätige diese Rechnung“

Use Cases sind Text Stories, werden genutzt um Anforderungen zu finden und diese festzuhalten.

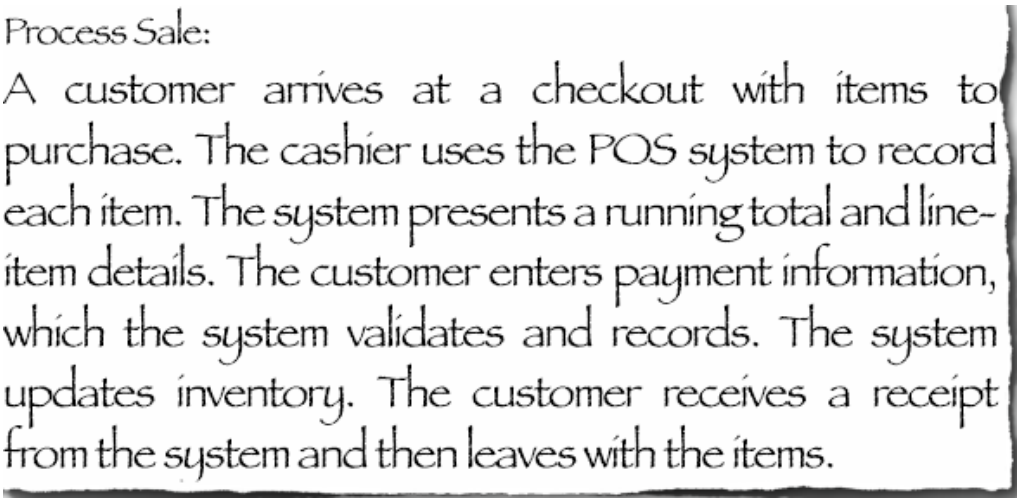
Ein Use Case enthält eine Anzahl von Aktionen die in einer wohldefinierten Reihenfolge stattfinden.

Beispiel: Kassensystem

- Werden genutzt um im Einzelhandel den Verkauf zu protokollieren
- Bieten Schnittstellen zu verschiedenen Diensten zur Errechnung von Steuern und Inventur
- Enthalten Computer Barcode Scanner und Software. Die Nutzerspanne reicht von kleiner Webapplikation bis zu komplexen Anwendungssoftware



Verkaufsprozess als Use Case(Textform)



Use Case – Glossar

- **Akteur:** etwas mit Verhalten, eine Person, PC System oder Organisation
- **Use Case:** Sammlung von dazugehörigen Szenarien, welche beschreiben wie der Akteur das System benutzt
- **Szenario:** (*Auch Use Case Instanz genannt*) ist eine spezifische Folge von Aktionen und Interaktionen zwischen Akteuren und System. Es ist genau ein Pfad durch die Use Cases, oder eine bestimmte Story, welche das System nutzt.
- **White box:** Bieten detaillierte Einsicht auf Interaktionen der internen Akteure
- **Blackbox:** Beschreiben nur Interaktionen mit externen Akteuren
- **Corporate:** Beschreiben einen Geschäftsprozess (oft unabhängig vom System)
    - **Meistens white box**
- **System:** Werden unter Rücksichtnahme des Systems definiert und ausgeführt
    - **Meistens Blackbox**
- **Brief (dt. kurz):** Ein Absatz welcher das Use Case zusammenfasst, in den meisten Fällen das Zielbeschreibung
- **Casual:** Informeller Absatzstil. Mehrere Paragraphen decken verschiedene Szenarien ab
- **Fully Dressed (vollständig bearbeitet):** Alle Schritte werden mit ihren unterstützenden Funktionen beschrieben (Voraussetzungen, Erfolgsgarantien etc.)
-

Der Fokus sollte zuerst auf die Korrektheit eines Use Cases liegen, dann auf der Genauigkeit.

1. Alle aktuell relevanten Use Cases identifizieren, welche auf einer hohen Ebene sind (niedriger Detailgrad, hohe Korrektheit

2. Details ausarbeiten (Genauigkeit verbessern)

Template FDUC

Use Case Section	Purpose/ Guidelines
Use Case Name	Startet mit einem Verb
Scope	Corperate, system (Systemnamen nennen), Subsystem
Level	User goals, summary goals, subfunction
Primary Actor	Wer leitet diesen Use Case ein?
Stakeholders and Interests	Wen nützt dieser Use Case was? Was erwarten sie?

Preconditions	Welche Bedingungen müssen erfüllt sein, bevor dieser Use Case gestartet wird?
Minimal Guarantees	Was soll das System **mindestens schaffen** um die Stakeholder zufrieden zu stellen?
Success Guarantees	Was muss erfüllt sein, damit der Use Case **erfolgreich abgeschlossen** wurde?
Main Success Scenario	Nennung von Schritten die zum Ziel führen.
Extensions	Alternative Möglichkeiten was passiert, wenn Use Case erfolgrech beendet / fehlgeschlagen

- **Scope**:
  - Function Scope: Grenzt Funktionalität unter Rücksicht auf das System ein
  - Design Scope (**Kommt oben vor)**: Definiert die Grenzen der Use Case
- **Level:**
  - **User Goal** (Nutzerziel): Die wichtigsten Ziele
  - **Summary Goal** (Zusammengefasstes Ziel): involviert mehrere Use Cases, wird genutzt und Systemkontext zu beschreiben
  - **Subfunction** (Subfunktion): Use Cases welche Teil eines User Goals sind
- **Stakeholders**
  - Jemand mit einem gewissen Interesse an diesem Use Case

Beispiel Kassensystem

Name: Buy Stocks over the Web
Primary Actor: Purchaser
Scope: Finance Package (PAF)
Level: User goal

Stakeholders and Interests:

Purchaser - wants to buy stocks and get them added to the portfolio.
Stock agency - wants full purchase information.

Precondition:

User is logged in.

Minimal guarantee:

Sufficient logging information will exist so that the PAF can detect that something went wrong and ask the user to provide details.

Success guarantee:

Web site has acknowledged the purchase; the logs and the user's portfolio are updated.

Richtlinien zur Erstellung von Use Cases

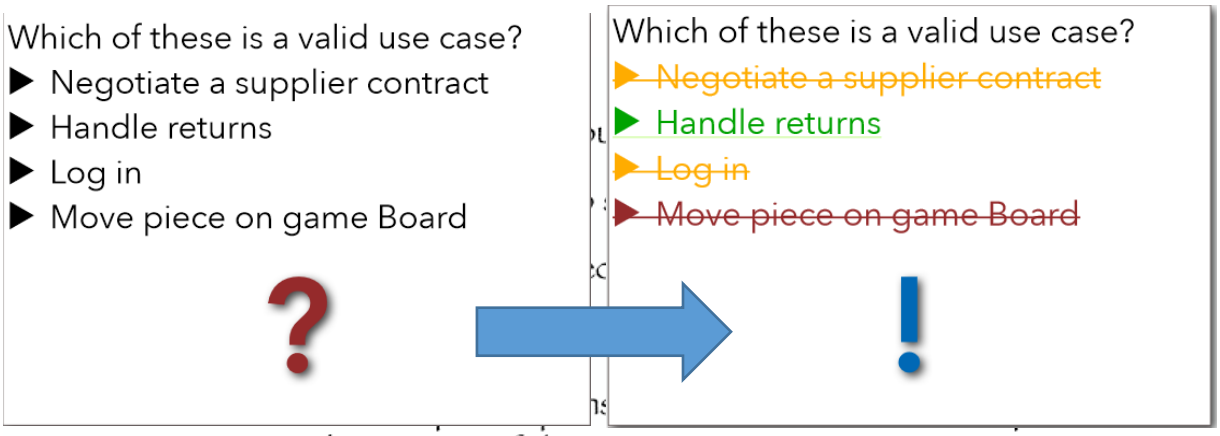
- Anfangs Benutzerschnittstelle unbeachtet lassen, auf Absicht des Use Cases fokussieren
- Kurze UCs schreiben
- Black-Box UCs schreiben
    - *„System speichert die Bezahlung“* ***anstelle von*** *„System schreibt Zahlung in die Datenbank“*
- Aus Sichtweise des Akteurs schreiben, an das Ziel des Akteurs orientieren

Finden von Use Cases während der anfänglichen Anforderungsanalyse

Faustregeln:

- Identifizieren aller aktuell relevanten Use Case, so korrekt wie möglich und abstrakt wie möglich
- Nach und nach Details hinzufügen
- Hat das Resultat einen messbaren Wert?
- Wird die Aufgabe von einer Person an einem Ort zu einer Zeit als Antwort auf ein geschäftliches Ereignis ausgeführt, sodass es einen geschäftlich messbaren Wert hat und die Daten in einem konsistenten Zustand hinterlässt? (**Elemantary Business Process EBP)**
- Ist es nur ein einzelner Schritt? (Size Test)

Beispiel – Finden von Use Cases



purchaser's portfolio

6.PAF shows the user the new portfolio standing

Extensions:

2a. Purchaser wants a web site PAF does not support

2a1. System gets new suggestion from purchaser, with option to cancel

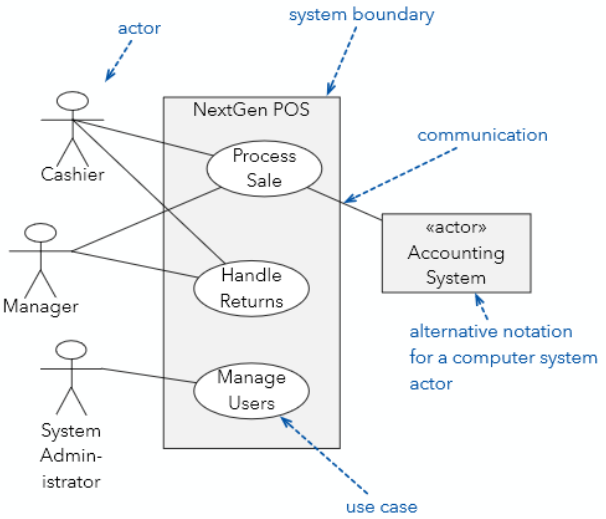
4a. Web site does not acknowledge purchase, but puts it on delay

4a1. PAF logs the delay, sets a timer to ask the purchaser about the outcome

- **Aushandeln eines Liefervertrages → Kein guter UC**, da er viel mehr umfasst als ein EBP
- **Einnahmen handhaben → Guter UC**; Erreicht ein Ziel, machbar in einem EBP
- Log-In → Kein UC; kein Wertgewinn
- **Kein UC;** Ist nur ein einzelner Schritt, besteht nicht den Size Test

UML-Notationen

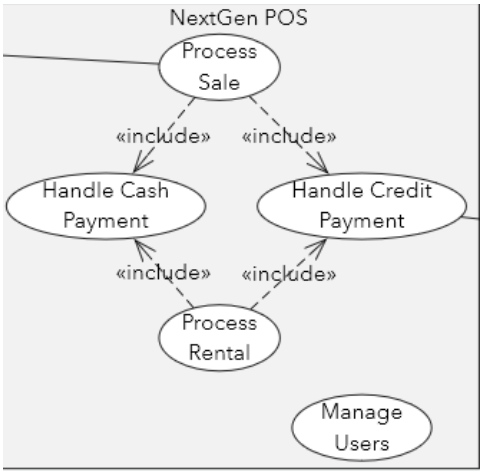
- schildern den Sachverhalt
- Nützlich in anfänglichen Phasen eines Projektes
- Kommunikation und Verständnis der UCs leichter zu sehen
- Verhindert redundanten Text
- Beeinflusst keine Systemanforderungen
- **Black-Box Ansicht** auf das gesamte System

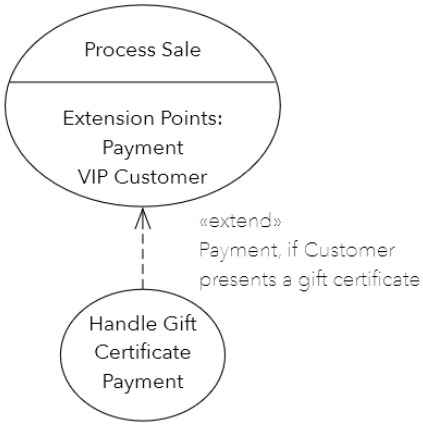


Include-Relation

Wird für anteiliges Verhalten benutzt, welches oft zwischen mehreren UCs vorkommt

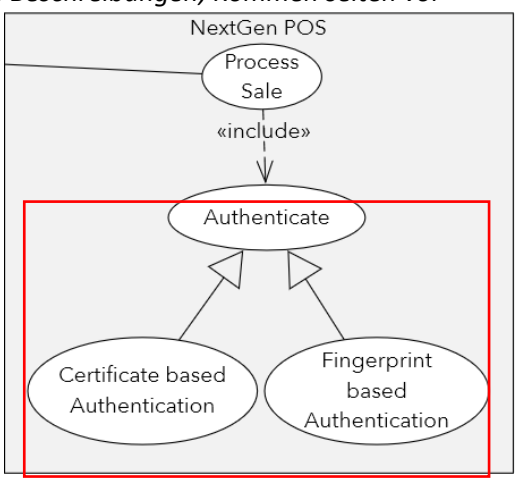
Der primäre Nutzen besteht darin, lange UCs nicht in kleine Teile zerlegen zu müssen, um sie verständlich zu machen





Extend Relation

- Beschreibt, wo unter welchen Umständen ein erweiternder oder zusätzlicher UC einen bereits vorhandenen UC erweitert
- *Fokus auf textuelle Beschreibungen; Kommen selten vor*



Inheritance Relation

- Erbender UC ersetzt eine oder mehrere Aktionen des *Super-UCs,*
- Erbender UC überschreibt Standardverhalten

Foliensatz 5 – Domain Modellierung

Warum? Ermöglicht es uns relevante Konzepte und Ideen dessen zu identifizieren

Wann? Während der Objektorientierten Analyse

Richtlinien: Erstelle ein Domain Modell nur für die bevorstehende Aufgabe

Curtis' Law: *„Ein guter Softwareentwurf benötigt tiefgreifendes Verständnis des Einsatzgebietes (der zu erstellenden Software)"*

Ein Domainmodell illustriert nennenswerte Konzepte einer Domain

- Wird während der OO-Analyse erstellt, um die Domain in Konzepte oder Objekte zu zerlegen
- Das Modell sollte ein Set von konzeptuellen Klassen identifizieren
- Bildet die Basis für die Software Design

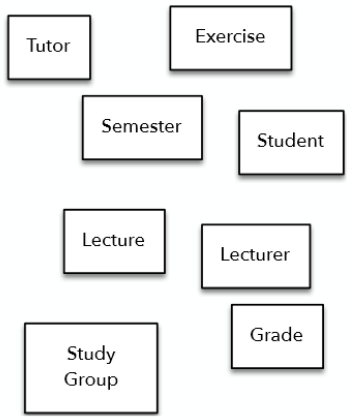
Konzeptuelle Klassen sind …

- Sind Klassen, Dinge oder Objekte in einer Domain
    - *Eine konzeptuelle Klasse besitzt ein repräsentierendes Symbol, eine Bedeutung(intension) und eine Ausprägung (extension) welche Gruppe von Klassen mit diesen Eigenschaften beschreiben*
- **Domainklassen oder Konzepte sind nicht zwingendermaßen Softwareklassen wie in Java/C#/Ruby etc.**
- Um ein **Domainmodell** zu visualisieren, werden **UML-Klassendiagramme** genutzt
    - Es werden keine Operationen im Modell definiert

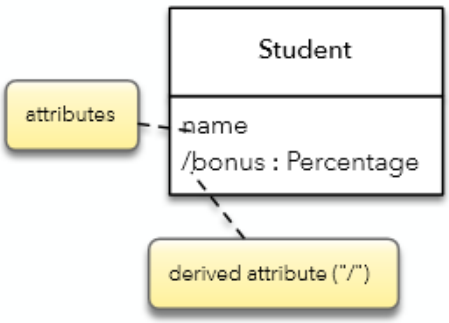
Visualisierung von Domainmodellen (Klassendiagramm)

- **Beispiel anhand eines Semestermoduls**

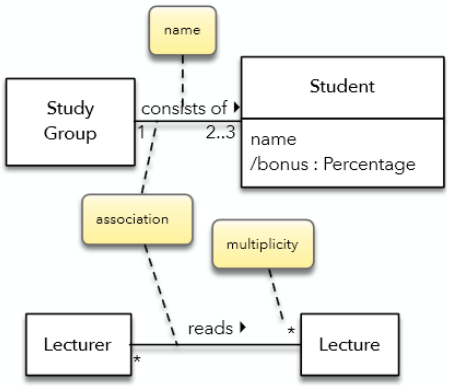
1. **Ein Modul enthält folgende „Klassen"**
    - **a.** Tutoren
    - **b.** Übungen
    - **c.** Semester
    - **d.** Studenten
    - **e.** Vorlesungen
    - **f.** Dozent
    - **g.** Note
    - **h.** Stundengruppen



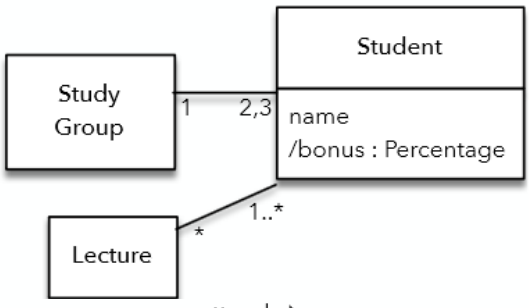
2. **Student** gibt Hausübungen ab, die vom **Tutor bewertet** werden
    - **a.** Bonus wird abgeleitet durch die Gesamtpunktzahl der HUs im Semester



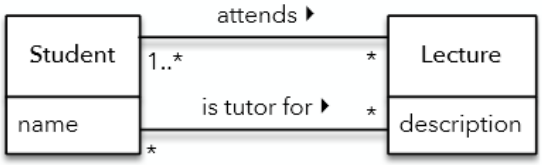
3. **Ein Dozent hält mindestens eine Vorlesung**
    a. **Student** nimmt normalerweise an mindestens einer **Vorlesung** teil
    b. **Studentengruppen** bestehen aus **2 oder 3 Studenten**
    c. Während des Semesters gibt es **mehrere Übungen**



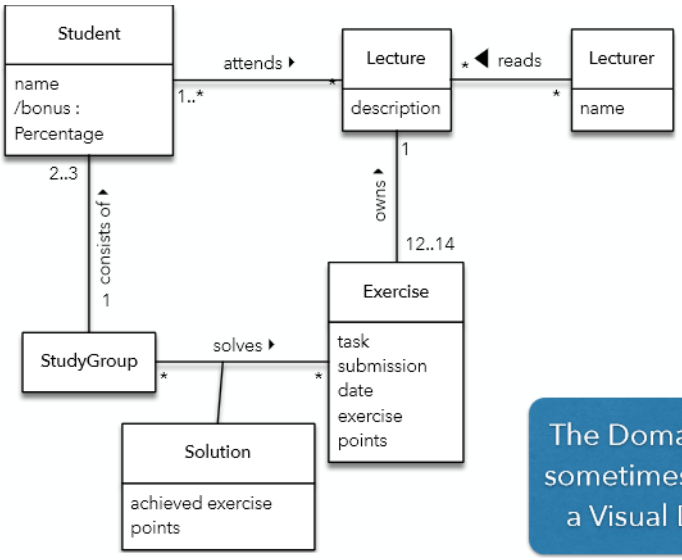
4. **Student geht zur Vorlesung**
    a. Im besten Fall
    b. Studentengruppe enthält 2 oder 3 Studenten



Zwei Klassen können mehrere Relationen haben



Das Ergebnis für unser Modul könnte so aussehen



The Domain Model is sometimes also called a Visual Dictionary.

Domainmodellierung – Überblick

- **Erstellen eines Domainmodelles bei fremder Domain**
  - Welche Konzepte und Objekte sind relevant?

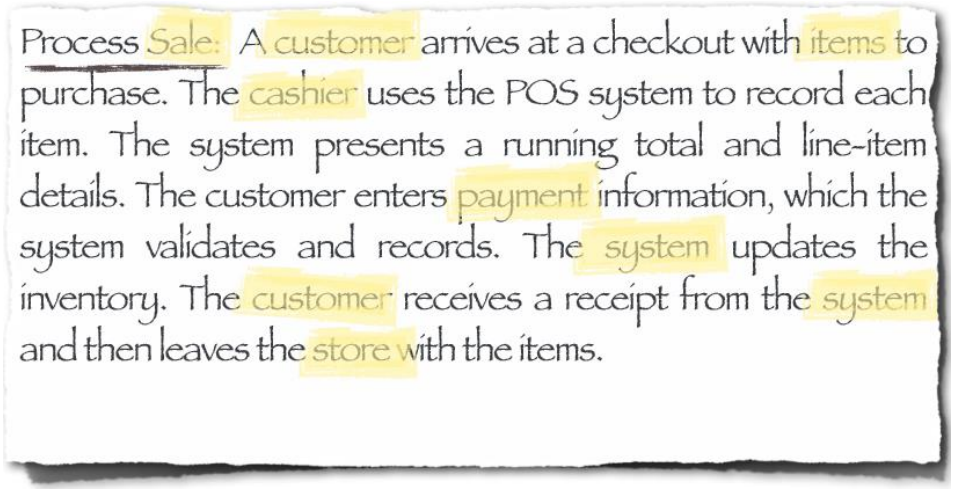
1. Finden von konzeptuellen Klassen
   a. Wiederverwendung oder Modifizierung eines vorhandenen Modells
   b. Kategorielisten verwenden
   c. Nomensätze identifizieren
2. **Als Klassen in UML aufzeichnen**
3. **Assoziation und Attribute hinzufügen**

Wird noch erläutert

Finden von konzeptuellen Klassen/Objekten mittels Nomensätze (linguistische Analyse)

- Identifizieren von Nomen und Nomensätzen in der textuellen Beschreibung, halte sie als mögliche Kandidaten vor konzeptuelle Klassen fest
- **Mechanisches/Algorithmisches Nomen-Klassen Mapping ist nicht möglich, da viele Nomen verschiedene Bedeutungen haben können. (Feder, Becken, etc.)**

Beispiel Kassensystem – Textstory



Identified candidate conceptual classes:
Customer, Item, Cashier, Store, Payment, Sales Line Item, Inventory, Receipt, Sale.

Richtlinien zur Identifizierung

- Wenn ein Kandidat aus anderen Kandidaten abgeleitet werden kann, kann man ihn Entfernen (Sales x Payment →Receipt
- Wenn es bestimmte Semantiken bzgl. des Geschäftlichen aufweist, sollte es miteinbezogen werden

- **Fakten**
- Identifizierter Kandidat: …, Receipt,…
- Ist eigentlich nur eine Kombi aus Verkauf und Zahlung

Sollte es im Domainmodell enthalten sein?

- **JA:** Wenn es uns interessiert, wie wir Einnahmen verwalten möchten
- **NEIN:** Falls uns das nicht interessiert

Beschreibende Klassen

- Enthält Informationen die andere Objekte beschreibt

Wann sollte eine beschreibende Klasse ins Modell miteinbezogen werden?

- Wenn man Beschreibungen so Objekten/Diensten haben will, unabhängig davon, ob es sie gibt oder nicht
- Vermeidung von redundanten Informationen

Attribut oder Klasse?

- Faustregel: Wenn wir von einem Objekt nicht behaupten können, dass es als Zahl, Text oder Bild etc. darstellbar ist, kann es kein Attribut sein, sondern eine konzeptuelle Klasse

Beispiel Flughäfen

- Ist das Ziel eines Fluges ein Attribut oder ein Objekt einer konzeptuellen Klasse?
    - ***Ein Flughafen kann nicht als Zahl oder Text im reinen dargestellt werden, daher ist Flughafen eine konzeptuelle Klasse***

Wann sollte man eine Assoziation in einem Domainmodell aufnehmen?

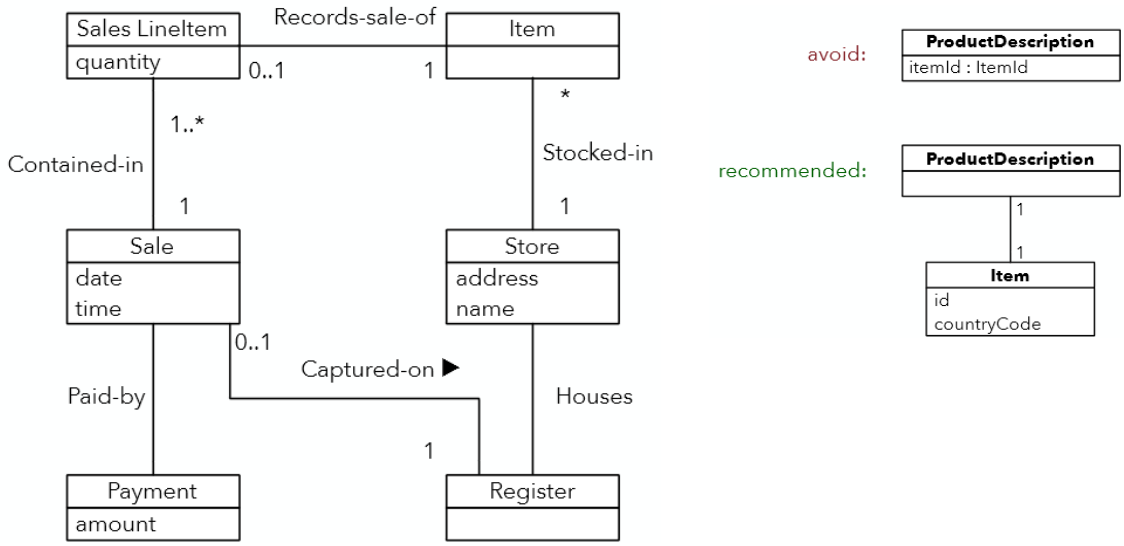
- Faustregel: Assoziationen sollten nur dann in ein Domainmodell aufgenommen werden, wenn bestimmtes Wissen erläutert werden soll

Man sollte Assoziationen miteinbeziehen, wenn eine der folgenden Eigenschaften zutrifft:

- A ist eine Transaktion welche mit in Zusammenhang mit einer Transaktion B steht
- A ein Einzelposten einer Transaktion B ist
- A ein Service oder Produkt der Transaktion B ist
- A eine Rolle/ein Akteur der Transaktion B ist
- A ein physisch/logischer Teil von B ist

Attribute in einem Domainmodell sollten möglichst immer primitive Datentypen sein!

Beispiel Kassensystem



Domainanforderungen werden eher durch die Domain des Systems abgeleitet als an der Nutzen der User

- Werden mit Domainspezifischen Ausdrücken formuliert, oft unverschändlich für Softwareentwickler
- Oft nicht explizit erwähnt, weil für den Domainexperten trivial
- Können funktional /nichtfunktional sein

Foliensatz 6 – Software Qualität

Wir unterscheiden zwischen **internen** und **externen Qualitätsfaktoren**

- **Intern:** können nur von Entwicklern bzw. Profis wahrgenommen werden
- **Extern:** werden von den Endnutzern wahrgenommen
    - *Externe Faktoren hängen aber von internen ab*

Intern

- Modular
- Verständlich
- Klare Rollenverteilung
- Keine Codeduplikationen

Extern

Wie stellt man Softwarequalität sicher?
Definierter Qualitätssicherungsplan (defined quality assurance (QA) plan): muss ständig in den Softwareentwicklungsprozess integriert werden

Verifikation und Validation: konkrete Aktion, um die Erfüllung der Anforderungen sicherzustellen (z.B. funktionale Korrektheit oder Systemgeschwindigkeit)

Korrektheit einer Software spiegelt sich darin wieder, dass ihre Methoden und Aufgaben wie in der Spezifikation angegeben funktionieren.

- sind detaillierte Anforderungsdefinitionen erforderlich

- Korrektheit ist nur bedingt möglich; Korrektheit der Software nur unter Annahme das untere Ebenen korrekt sind

Robustheit wird als die Eigenschaft einer Software bezeichnet, dass sie auch unter unnormalen Bedingungen korrekt arbeitet.

- Was passiert außerhalb der Spezifikation?
- Ergänzt Korrektheit

Schlechte Software – Warum?

- Weil er ein Noob ist
- Zu früh optimiert
- Design verhindert Erweiterbarkeit
- Keine Zeit

Erweiterbarkeit charakterisiert die Einfachheit der Anpassung von Softwarekomponenten als Reaktion auf sich ändernder Spezifikation.

- Wichtige Richtlinien zur Erweiterbarkeit
    - **Einfachheit des Designs**
    - **Dezentralisierung**

Wiederverwendbarkeit beschreibt die Eigenschaft aus einzelnen Softwarekomponenten neue Software bauen zu können.

Kompatibilität beschreibt die Einfachheit des Kombinierens verschiedener Softwarekomponenten.

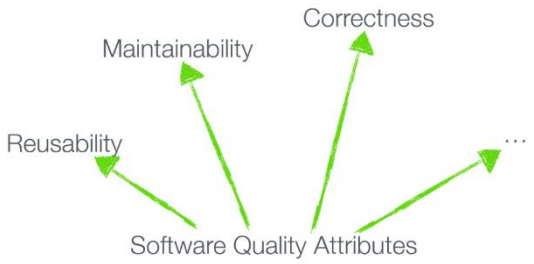
Übertragbarkeit beschreibt die Eigenschaft die Software auf verschiedenen Hard- und Softwareumgebungen lauffähig zu halten.

Effizienz bezeichnet die beste Auslagerung von Ansprüchen auf Hardwareressourcen

- CPU Zeiten / Auslastung, RAM/Festplatte, Stromverbrauch
- Immer erst „gute“ Algorithmen ausprobieren, bevor man zu „schlechten“ Algorithmen greift.
- Die Effizienz darf durchaus darunter leiden, dass die Software korrekt funktioniert

Funktionalität bezeichnet den Umfang der Möglichkeiten die einem das System bietet.

- Falls neue Features hinzugefügt werden, sollte darauf geachtet werden, dass sie einheitlich mit den bereits vorhandenen ist.



Oft ist es nicht möglich alle Qualitätsmerkmale zu bearbeiten, da sie im Widerspruch zu einander stehen.

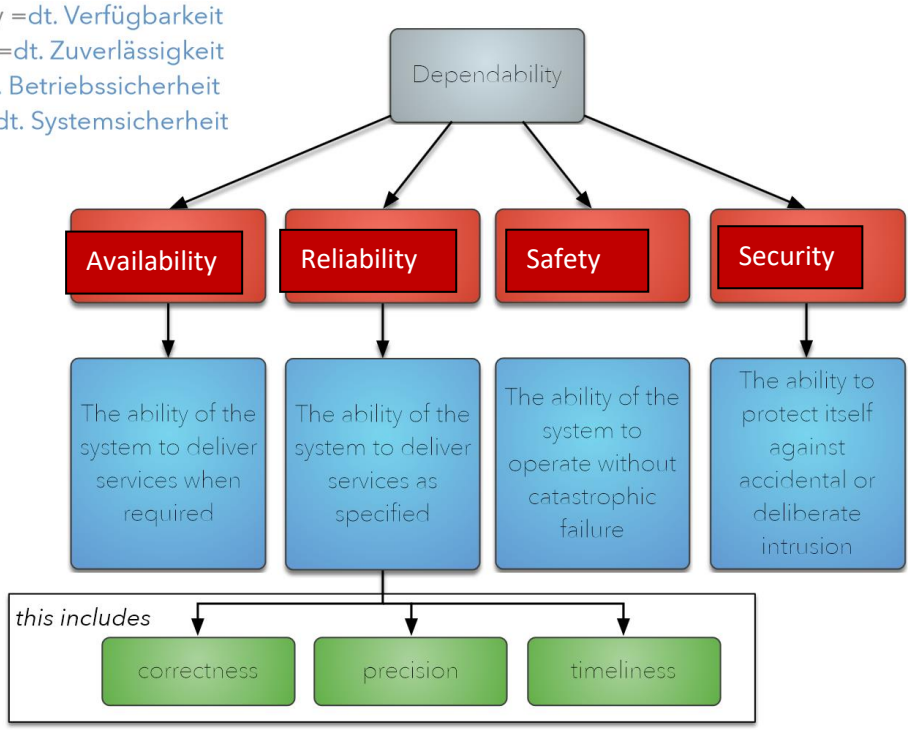
Gute Software und ihre Eigenschaften

- **Wartbarkeit**
    - Software sollte so geschrieben werden, dass sie auf Kundenwunsch beliebig angepasst werden kann
- **Effizienz**

- Sollte keine Systemressourcen verschwenden

- **Nutzbarkeit**
  - Sollte von dem Kunden bedient werden können
- **Verlässlichkeit**
  - Richtet im Falle eines Systemabsturzes keinen physischen oder wirtschaftlichen Schaden an.

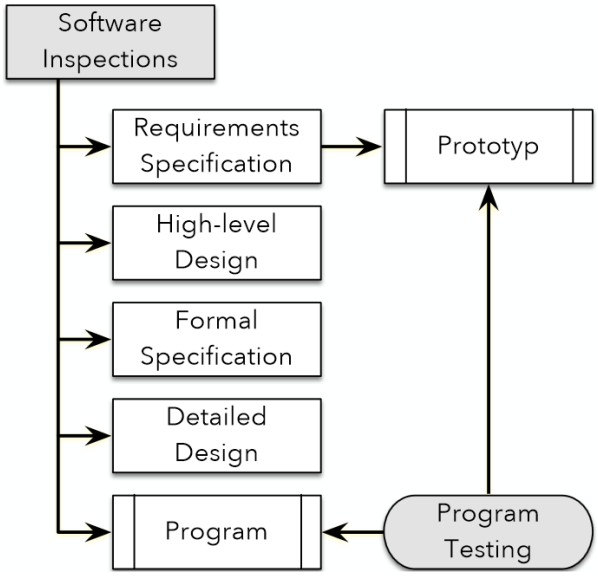
Availability =dt. Verfügbarkeit
Reliability =dt. Zuverlässigkeit
Safety =dt. Betriebssicherheit
Security =dt. Systemsicherheit



Verifikation und Validierung

Validierung: *Bauen wir das richtige Produkt?*

Verifizierung: *Bauen wir das Produkt richtig?*



Statische Techniken (es ist nicht notwendig, dass das Programm läuft)

- **Software Inspektionen oder Peer-Review (Statische Technik)**
  - *Inspektionen können in jeder Phase der Entwicklung stattfinden*
- **Automatisierte Statische Analyse** (automatisierte Softwareüberprüfung, z.B. Typ Checker)
- **Formale Verifikation** (kann den spezifischen Wert eines Programms garantieren)
- **Testen von Software (Dynamische Technik)**

Dynamische Techniken (verlangt, dass das Programm ausgeführt wird)

- **Testen** (**test harness:** in der Qualitätssicherung von Software bezeichnet ein Test-Harnisch (engl. test harness) bzw. Testrahmen eine Sammlung von Software und Testdaten, die zum systematischen automatisierten Testen eines Programms unter verschiedenen Umgebungsbedingungen verwendet wird) → stellt sicher, dass das Programmverhalten bei der Ausführung dem erwarteten entspricht

    - Testsuits durchlaufen
- **Laufzeitevaluierung** (instrumentiert das Programm mit Behauptungen → z.B. Methoden für Vor- und Nachbedingungen)

Code Inspektion

- Data fault:
    - Variablen benutzt bevor sie initializiert wurden
    - IndexOutOfBounds
- Control Faults
    - Richtige Konditionen?
    - Nichtgenutzter Code
    - Unendliche Schleifen
- I/O Faults:
    - Alle Eingabevariablen genutzt?
    - Unerwartete Eingabe gehandelt?

Software Metriken

Fan-in/out:

- In: Anzahl der Methoden, welche die Methode M aufrufen
- Out: Anzahl Methoden die M aufruft

Cyclomatic Complexity:

- Linear unabhängige Pfade durch den Code (CFG)
- C = E – N + 2P, N = Knoten, E = Kanten, P = Verbundene Komponenten
- Falls C > 10, Design überarbeiten

Länge des Codes

- Länge des Codes kann Anzeichen über die Komplexität geben

Foliensatz 7: Testen von Software

Zwei sich ergänzende Ansätze können wir für Verifikation und Validierung festhalten.

Programminspektion

- *Das Ziel ist eher, Programmdefekte, Verstöße und schlechten Code zu finden als das Bedenken von weiten Designproblemen*
- *Wird von einem Team ausgeführt, welche den Code systematisch checken, meist mit Checklisten*

Automatische Quellcode Analyse

- *Enthält unter anderem Kontrollflussanalysen, Datennutzungs- und Flussanalysen, Informations- und Pfadanalysen*
  Statische Analysen decken oft Anomalien auf

Formale Verifikation

- *Kann die Abwesenheit von spezifischen Bugs, Dead Locks und Race Conditions garantieren*

Softwaretests beziehen sich auf das Nutzen der Software unter mit bestimmten Testeingaben und Fehlfunktionen aufzudecken

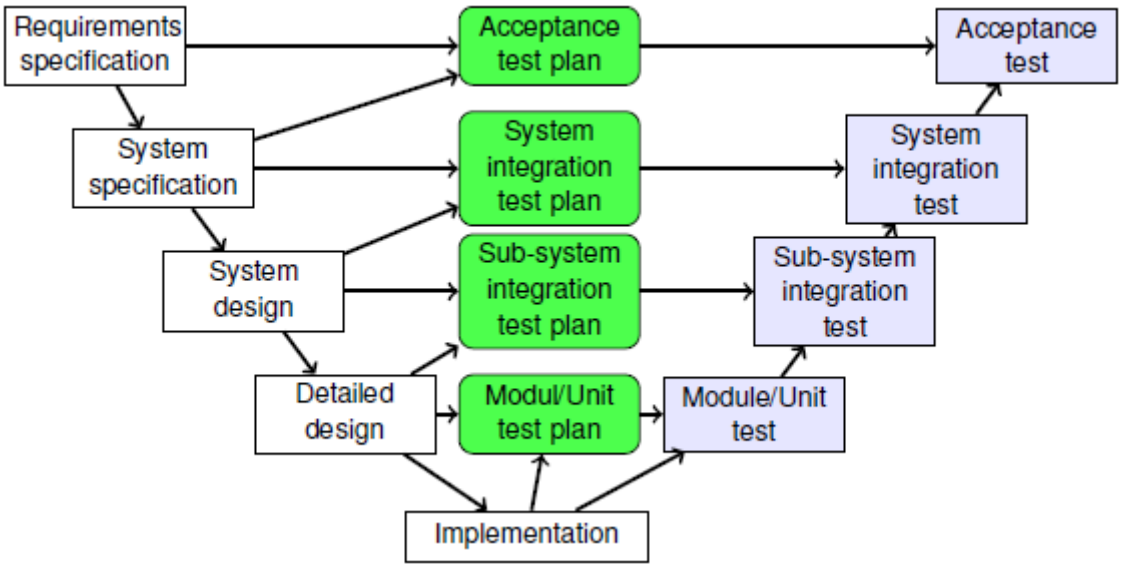
Validierungstest

- Beabsichtigt dem Kunden zu zeigen, dass die Software genau die ist, die er haben möchte (Je ein Test für eine Anforderung)

Testen auf Defekts

- Beabsichtigt Fehler zu finden
- Beinhaltet:
    - Aktive Fehlersuche: Wenn ein Test gewollt fehlschlagen soll
    - Übereinstimmungsorientiert: Soll demonstrieren, dass bestimmte geforderte Fähigkeiten übereinstimmen

Testpläne geben die Reihenfolgen und Prozeduren vor, setzen Standards für das Testen und entwickeln sich während des Implementierungsprozesses weiter.



Testplan: V-Model

Der Scope der Test die Sammlung an Komponenten, welche verifiziert werden müssen

- **Modultest**
    - Enthalten nur kleine ausführbare Objekte
- **Integrationstest**
    - Ein komplettes (Sub-)System, wo die Schnittstellen zeigen sollen, dass mehrere Komponenten miteinander zusammenarbeiten können
- **Systemtest**
    - Ein kompletter Softwaretest mit allen Aspekten.
    - Funktional, Leistung, Stress oder Beanspruchung

Testdesing
Das Systemobjekt (oder Implementierungsobjekt) wird übergeben als SUT (oder IUT)

1. Identifiziere, modelliere und analysiere die Zuständigkeiten des SUT
2. Kreiere Testfälle basierend auf dieser externen Perspektive
3. Füge Testfälle hinzu, basierend auf Codeanalyse, Verdacht und Heuristik
4. Bestimme für jeden Testfall, wie der erfolgreiche/erfolgslose Durchlauf bewertet werden kann, z.B. liefere für jeden Testfall die erwarteten Ergebnisse oder nutze eine andere Art von Testorakel

Das Testen muss auf einem Mangelmodell basieren. Anzahl an Tests ist möglicherweise endlos, wir müssen also uns Gedanken machen, wo Fehler auftreten können und diese Stellen testen.

Nach dem Testdesign muss ein Automatisiertes Testsystem (ATS) entwickelt werden

Eine ATS…

- Startet die Implementation unter Testbedingungen
- Stellt die Umgebung ein
- Bringt die Implementation zur benötigten Vorteststufe
- Wendet Testeingaben an
- Evaluiert den Endzustand und die Ausgaben

Das Ziel von Testen besteht darin zu überprüfen, dass die Testimplementierung (IUT) die minimalen Operationen ausführt.

- Test Suits müssen erstellt werden; Ergebnis jedes Tests mit Nicht-Bestanden/Bestanden bewerten
- Nutzung von Coverage-Tools, beurteilen nach Tests erreichte Coverage
- Wiederhole Test Suite und bewerte den angezeigten Coverage
- Falls notwendig, für nicht abgedeckten Code eigene Test schreiben
- Beende das Testen, sobald alle Tests bestanden sind!

Testpoint (Prüfpunkte)

Ein Testpoint kann eine spezifische Eingabe für einen Testfall oder eine statische Variable sein. Der Testpoint wird aus einer Domain entnommen. Die Domain an sich enthält eine Anzahl an Werten für Eingaben oder Zustandsvariablen.

Heuristiken für die Wahl von Testpoints

- Äquivalenzklassen
- Grenzwert Analysen
- Testen von speziellen Werten

Test Case (Testfall)

Ein Testfall spezifiziert…

- die Vorteststufe einer IUT (Implementation under test)
- Testeingaben und Bedingungen
- Erwartete Ergebnisse

Test Suite: Ansammlung von Testfällen

Test Run: Ausführung (mit Ergebnissen) von einer Test Suite. Ein Test kann fehlschlagen oder korrekt (entspricht den erwarteten Ergebnissen)

Test Driver: sind Klassen oder Programme welche die Tests ausführen

Test Harness ist ein System bestehend aus Test Drivern und anderen Tools, um die Ausführung von Tests zu unterstützen

Failures, Errors und Bugs

- **Failure:** Die Unfähigkeit einer Komponente oder eines Systems eine bestimmte Aufgabe unter gegeben Einschränkungen auszuführen
- **Fault:** fehlender oder fehlerhafter Code
- **Error:** Humane Aktion welche den Fehler verursacht hat

- **Bug:** Error oder Fault

Testplan

- Ein Dokument welche die Herangehensweise während dem Testen enthalten kann
  - Arbeitsplan
  - Allgemeine Aufgaben
  - Erklärung des Testdesigns
  - Etc.

Zwei Modelle:

  - Übereinstimmungsorientierte Tests
  - Fault-orientierte Tests

Coverage (Abdeckung)

- Die Gesamtheit einer Test Suite in Bezug auf einen bestimmten Testfall
- Coverage beschreibt die Prozentale Anzahl an Elemente die zum Bestehen der Tests benötigt werden

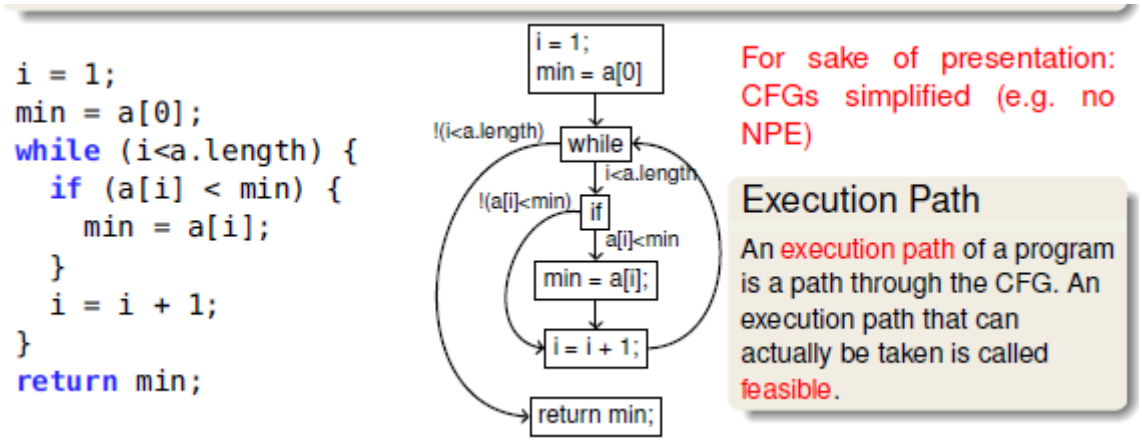
Strukturelle Coverage Kriterien:
Statement Coverage (SC): alle Statements müssen mindestens einmal ausgeführt werden
Basic Block Coverage (BBC): alle grundlegenden Blöcke müssen mindestens einmal ausgeführt werden
Branch Coverage (BC): jede Kante von einem Knoten wird mindestens einmal ausgeführt
Path Coverage (PC): alle Ausführungspfade werden mindestens einmal ausgeführt

Logik-basierende Coverage Kriterien:
Condition Coverage (CC): jede Bedingung wird mindestens einmal mit true oder false bewertet
Decision Coverage (DC): jede Entscheidung wird mindestens einmal mit true oder false bewertet
Modified Condition Decision Coverage (MCDC): kombiniert Aspekte von CC, DC und unabhängige Tests
Multiple-condition Coverage (MCC): alle true-false Kombinationen von Bedingungen werden mindestens einmal geübt

Bekannte Methodenscopes für Coverage Modelle – Strukurell und Logik-basierend (Alle TestSuites beziehen sich auf auf den Callgraph)

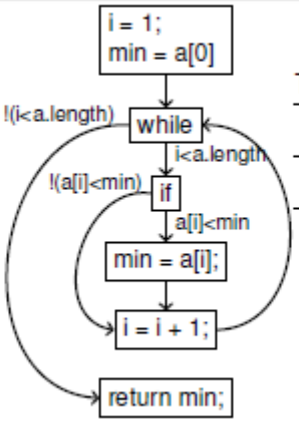
Strukturelles Coverage basiert auf das Kontrollflussdiagramm eines Programms.

(Struk.) Statement Coverage (SC): Eine Testsuit erreicht Statemant Coverage, wenn das Programm mit einer Eingabe das Programm durchlaufen lässt.



Strukturelles Covarge: Statement Coverage: Ein Test Suite TS erreicht Statement Coverage für ein Programm p, wenn für jedes Statement s in p ein Test TS existiert, unter welchem s ausgeführt wird.

```
i = 1;
min = a[0];
while (i<a.length) {
  if (a[i] < min) {
    min = a[i];
  }
  i = i + 1;
}
return min;
```



Test Suite 1:

Test Case	Test Input	SC
1	a=[0]	X

Test Suite 2:

Test Case	Test Input	SC
1	a=[0,1]	X

Test Suite 3:

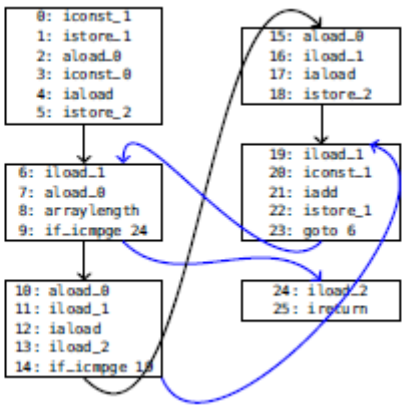
Test Case	Test Input	SC
1	a=[1,0]	✓

Strukturelles Coverage: Basic Block Coverage

Coverage Kriterium für Byte Code: Basic Block Coverage ist ähnlich zu Statement Coverage, aber es wird gewöhnlich auf Maschinen/Bitcodierten Anweisungen definiert.

- Ein Basis Block B ist eine maximal lange Sequenz von aufeinanderfolgenden Anweisungen, ohne die Möglichkeit anzuhalten oder ein Branching auszuführen

```
i = 1;
min = a[0];
while (i<a.length) {
  if (a[i] < min) {
    min = a[i];
  }
  i = i + 1;
}
return min;
```



6 basic blocks
bytecode CFG≠source CFG
⇝ BBC ≠ SC

Definition BBC (Basic block coverage): Eine Test Suite erreicht Basic Block Coverage für ein Programm p, wenn jeder Basisblock in p mindestens einmal durch ein Test in TS ausgeführt wird.

Strukturelles Coverage: Branch Coverage (BC): Eine Test Suite erreicht Branch Coverage für ein Programm p, wenn jeder Pfad von einem Knoten in einer Test Suite mindestens einmal durchlaufen wird

Test Suite 3:

Test Case	Test Input	BC
1	a=[0,1]	✓
2	a=[1,0]	

Test Suite 1:

Test Case	Test Input	BC
1	a=[0]	X

Test Suite 2:

Test Case	Test Input	BC
1	a=[0,1]	X

Test Suite 4:

Test Case	Test Input	BC
1	a=[1,0,2]	✓

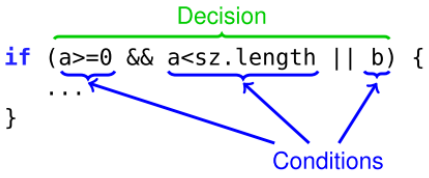
Strukturelles Coverage: Path Coverage (PC): Eine Testsuit erreicht Path Coverage, wenn alle ausführbaren Pfade mindestens einmal ausgeführt wird.

```
if (i>=0) {
  m = 1;
}
if (j<0) {
  m = 2;
}
```

- How many execution paths? 4
- Test suite with PC?

$$TC \hat{=} (i, j) : \{(-1, 1), (-1, -1), (1, 1), (1, -1)\}$$

Definition Condition: Eine Kondition ist ein Boolescher Ausdruck, welcher nicht in weitere Booleschen Ausdrücke heruntergebrochen werden kann. (atomar)



Definition Decision: Bezeichnet einen gesamten Booleschen Ausdruck.

Logischer Covarage: Decision Coverage (DC): Jede Entscheidung wird mindestens einmal zu true/false evaluiert

Logischer Coverage: Condition Coverage: Jede Kondition wird mindestens einmal zu true/false evaluiert

Logischer Coverage: Modified Condition Decision Coverage (MCDC): Für eine gegebene Kondition *C* in der Entscheidung *D,* wird MCDC durch eine Testtuit *TS* erreicht, wenn *C* jeweils einmal zu true und zu false evaluiert wird, wobei *d* in beiden Fällen anders evaluiert wird.

Example (Decision: a>=0 & (b<0 | c))

- Has $TS = \{(-1, 1, false), (1, 1, true)\}$, $TC \hat{=} (a, b, c)$ for condition a>=0 MCDC? No (does not evaluate condition c to same value in both test cases)
- Has $TS = \{(1, 1, false), (-1, 1, false)\}$ for condition a>=0 MCDC? No (does not evaluate whole decision differently)
- Has $TS = \{(1, 1, true), (-1, 1, true)\}$ for condition a>=0 MCDC? Yes

Software Qualitätssicherung - Klassifizierung von Gefundenen Problem

True und **False Positives/ Negatives**

Irrelevante True Positives**/ Erkannte** False Positives

True Positives sind ein korrektes Vorkommen on etwas und sollten von Statischen Analysetools gefunden werden

False Positives sind einfach nur falsch, werden normalerweise durch eine schwache Analyse verursacht

Foliensatz 8 – Interaktive UML – Sequenzdiagramme, Kom-Diagramme

Datenorientierte Modellierung: Fokus auf statische Aspekte

- Welche Daten werden von welcher Klasse verwaltet?
- Welche Operationen finden auf den Daten statt?

Verhaltensorientierte Modellierung

- Wie arbeiten die Objekte zusammen, um das gemeinsame Ziel zu erreichen?
- Wie sieht die Rollenverteilung aus?

Häufig vorkommende UML Interaktionsnotationen

s1:Sale — Beschreibt: `Sale s1 = …;`

«metaclass» Font — Beschreibt: `Class<Font> fontClass = Font.class;`

Sales:ArrayList<Sale> — Beschreibt: `ArrayList<Sale> sales = …;`

sales[i]:Sale — Beschreibt:

```
ArrayList<Sale> sales = …;
Sale sale = sales.get(i);
```

Nachrichtenformatstandards bei UML IDs

- Oft genutzt:
  `return = message(parameter:parameterType):returnType`
- Klammern werden weggelassen, wenn keine Parameter erwartet werden
- Typinformationen werden weggelassen, falls unnötig

Allgemein gültige Notationen

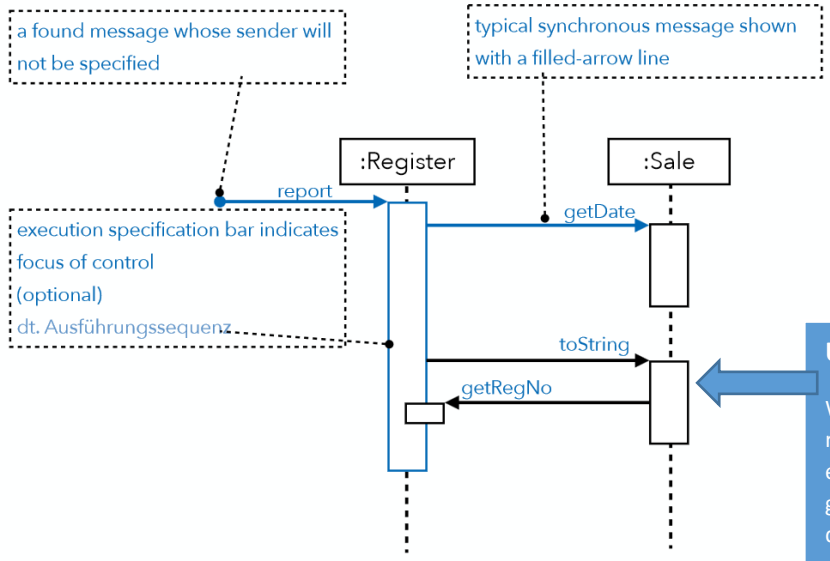
`initialize(code)` **äquivalent zu** `initialize`

`d = getProductDescription(id)` **äquivalent zu**

`d = getProductDescription(id : ItemId)` **äquivalent zu**

`d = getProductDescription(id : ItemId): ProductDescription`

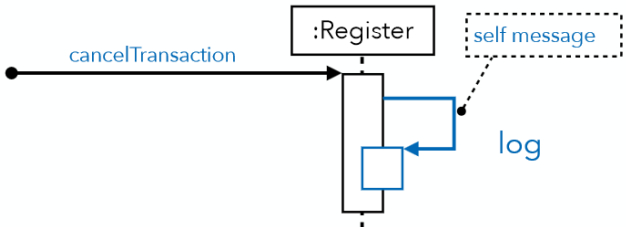
Sequenzdiagramme – Modellieren von Synchronen Nachrichten



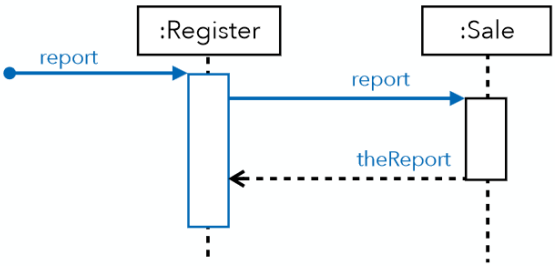
UML Superstruktur

Wenn die Nachricht eine CallAction repräsentiert, wird es normalerweise eine Antwort auf diese Nachricht geben, und zwar auf die LifeLine von der sie aufgerufen wurde

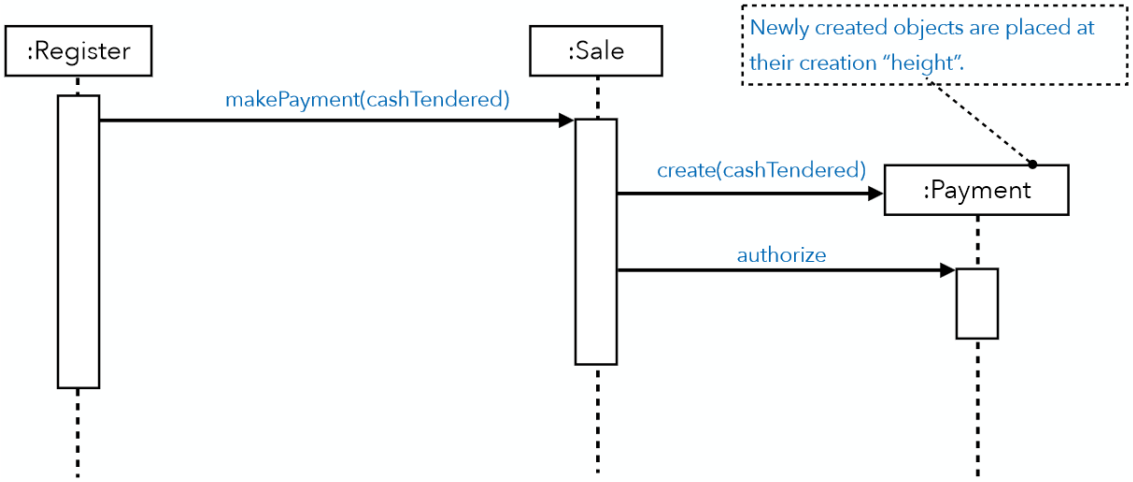
Self-Messaging:



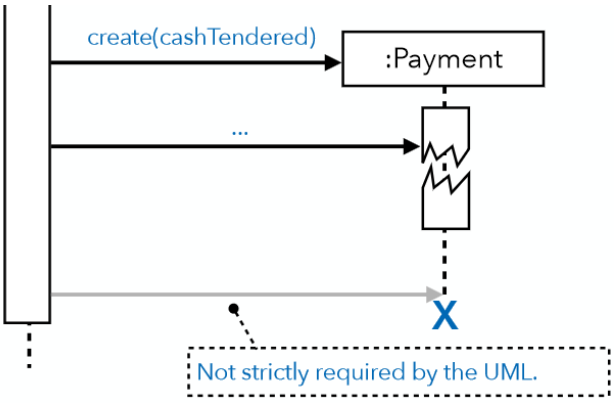
Nutzen einer MessageBar um Returns zu Modellieren



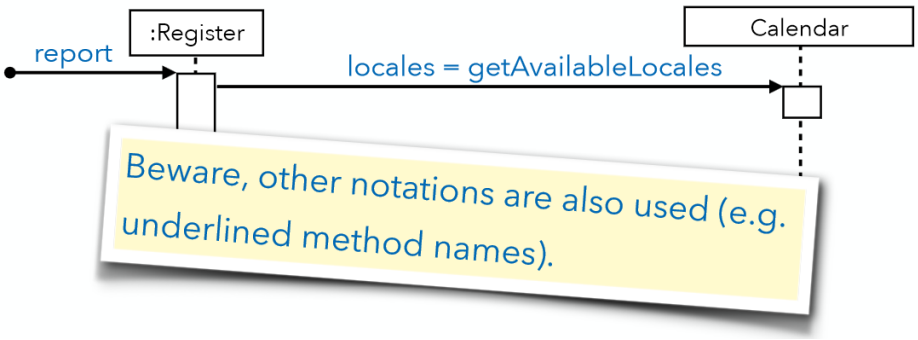
Erzeugen von neuen Objektinstanzen



Zerstören von Objektinstanzen - wird auch oft benutzt um zu zeigen, dass ein Objekt nicht mehr länger nutzbar ist



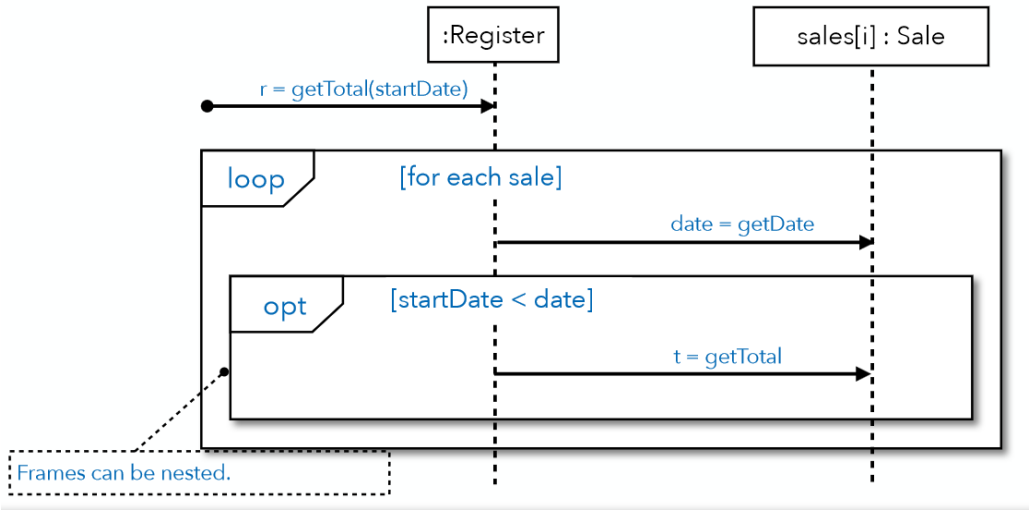
Aufrufen von statischen Methoden (Klassenmethoden)



In Java:

```
Public class Register{
   public void report(){
     Locale[] locales = Calender.getAvailableLocales();}
  }
```

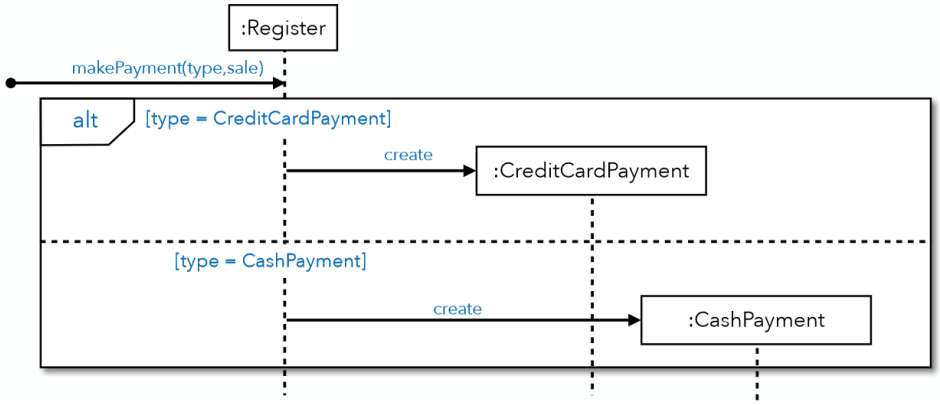
Modellierung von Schleifen und Bedingungen



Loop: gibt an, dass dies eine Schleife ist

Opt: wird ausgeführt, wenn die Bedingungen stimmen

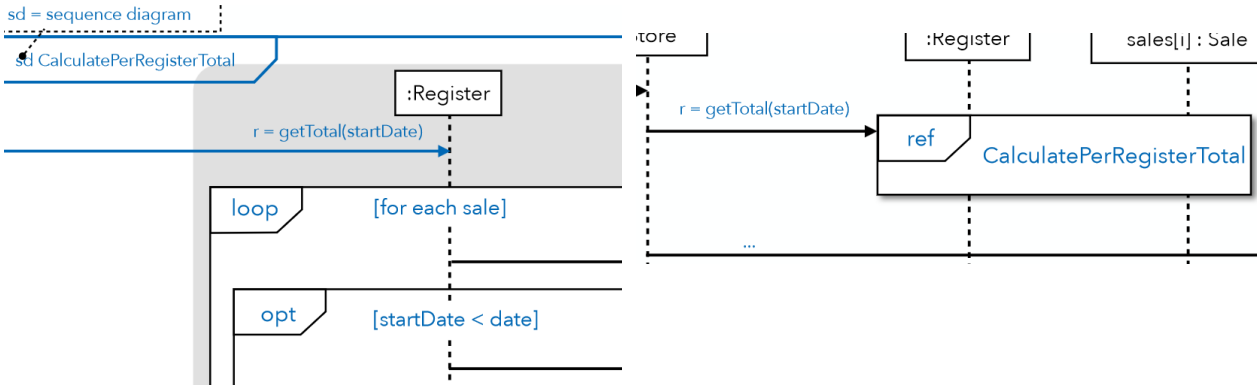
Entscheidungsfälle bei 2 oder mehr Möglichkeiten



Vergleichbar mit switch-case in Java - Verschachtelung und Abstraktion möglich

Referenzen

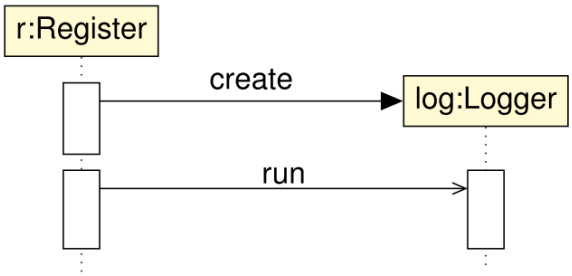
Eine Referenz ist ein InteractionFrame welches auf einen anderen Frame verweist. Referenzen, dienen zur Vereinfachung des Diagrams, da sie ganze Methodenaufrufe oder Anweisungen auslagern können.



Wie senden die Modelle asynchrone Nachrichten?

Aufgabe: Die Log-Informationen sollten automatisch gesammelt und im Hintergrund verarbeitet werden.

Asynchrone Nachrichten sind **nicht blockend**.



Kommunikationsdiagramme

Links und Nachrichten in Kommunikationsdiagrammen

- Ein **Link** ist ein Verbindungspfad zwischen zwei Objekten (Instanz einer Assoziation)
  - Ein Link gibt an, dass eine Art Navigation und Sichtbarkeit zwischen Objekten möglich ist
- Jede **Nachricht** zwischen Objekten wird mit einem Ausdruck und einen kleinen Pfeil welche die Richtung der Nachricht repräsentiert
  - Sequenznummern werden hinzugefügt um die sequentielle Reihenfolge zu zeigen, in der die Nachrichten gesendet werden. **Die Startnachricht enthält keine Nummer**.

Systemsequenzdiagramme (SSD)

Eine SSD illustriert Input und Output Events

- Eine SSD zeigt – je nach Szenario-
  - Events, welche externe Akteure erstellen
  - Ihre Reihenfolge
  - Systemübergreifende Events

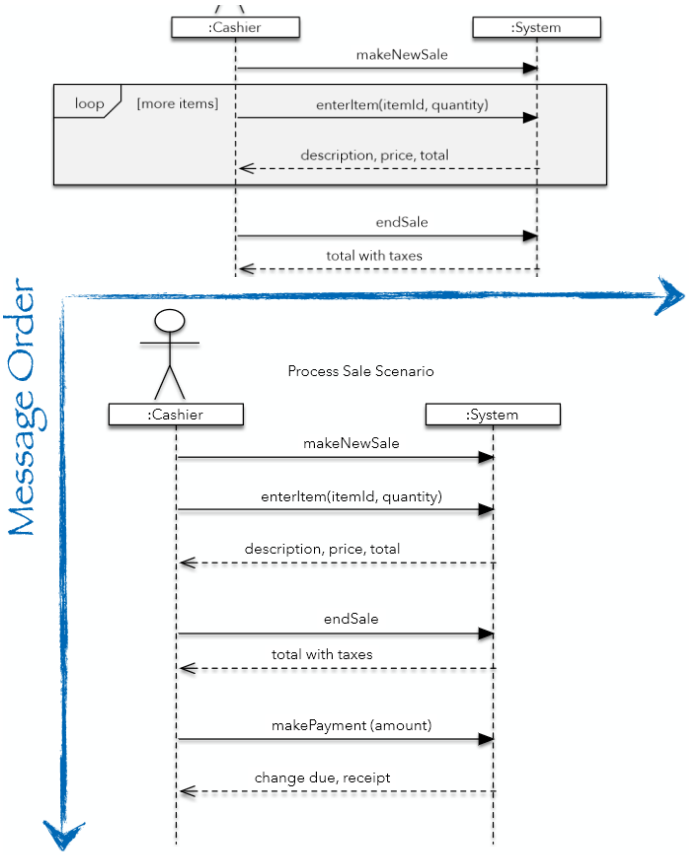
- Das System wird als Black-Box gehandhabt
- SSDs werden aus Use Cases abgeleitet; werden oft zur Darstellung der Main Success Szenarien einzelner Use Cases genutzt, oder für oft vorkommende und komplexe Use Cases
- Werden als Input für Objektdesign genutzt

Systemevents und Operationen

- SysOps sind Operationen die als Blackbox in Schnittstellen angeboten werden.
    - High Level Operationen, durch externe Trigger oder User
- Während der Systemverhaltensanalyse, werden die konzeptuellen Klassensysteme zugeschrieben
- SysOps werden in der SSD angezeigt

Use Case: Verkauf

1. Kassierer beginnt neuen Verkauf
2. Kassierer gibt Warennummer ein
3. System registriert die eingegebene Ware und stellt die Beschreibung zur Verfügung
4. System präsentiert vollen Preis samt Steuern
5. Kassierer teilt Kunden den Gesamtpreis mit
6. Kunde zahlt und System kümmert sich um die Bezahlung



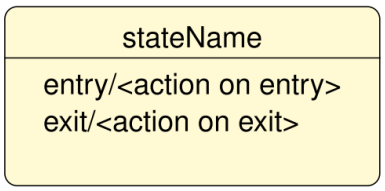
State Machine Diagram

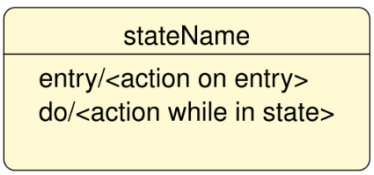
Ist eine Variante von endlichen Automaten.

Sie werden genutzt um …

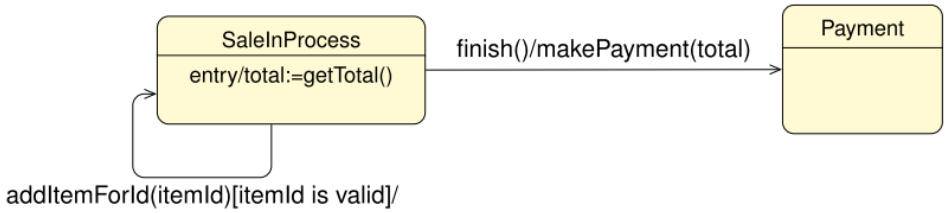
- Diskrete Eventorientierte Verhalten zu modellieren (behaviour state machine)
- Gültige Interaktionssequenzen auszudrücken (protocol state machine)

SMDs werden in der Regel genutzt, um das Verhalten eines **Klassenobjektes** zu beschreiben.





Transitionen



Labeling: **trigger? (guard)? / action)?**

Triggers:

- Methodenaufrufe
- Zeitgesteuertes Event
- Eventwechsel

Guard:

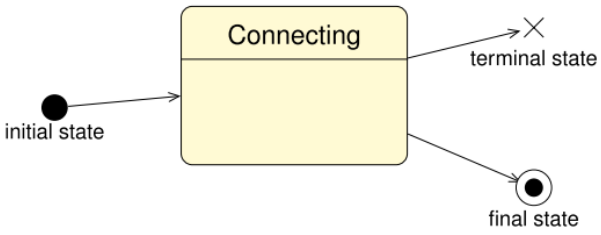
- Bool'scher Ausdruck
  Action:
- Methodenaufruf oder "Vendor Specific expression"

Start/Ziel/Ende

Initial: Startzustand

Terminal: Komplett durchlaufen + Objekt zerstört

Final: signalisiert Ende des Durchlaufs



Foliensatz 9 – Design Heuristiken

Werkzeuge welche als Input für ein OO Design genutzt werden können

- Ein Domainmodell
- Beschreibungen von Use Cases (User Stories) welche in der aktuellen Iteration entwickelt werden
- Eine SSD

Nächste Schritte

Bauen eines Interaktionsdiagrammes für Systemoperationen mit vorliegenden Use Cases durch anwenden von Richtlinien und Prinzipien zur Festlegung der Responsibilities

Verantwortung für Systemoperationen

- Während der Systemverhaltensanalyse, werden Systemoperationen zu bestimmten konzeptuellen Klasen zugewiesen
- Eine **Controllerklasse** wird angelegt, welche die Systemoperationen ausführen wird

Interaktionsdiagramme für Systemoperationen (IDS)

- Erstelle für jede Systemoperation ein eigenes Diagramm der aktuellen (in der Iteration vorkommenden) Systemoperationen
- Nutze die Systemoperation als Startnachricht
- **Falls das Diagramm zu groß wird, teile es in kleinere auf!**
- Teile die Verantwortungen unter den Klassen auf
    - Auf die des konzeptuellen Modells und andere die während der Entwicklung entstanden sind
    - Basierend auf der Beschreibung der Systemverhaltensanalyse

Grundlagen des OO Designs

Verantwortungen

Each responsibility is an axis of change. When the requirements change, a change will manifest through a change in responsibility amongst the classes. If a class has multiple responsibilities, it has multiple reasons to change.

Aktive Verantwortungen vs. implizite Verantwortungen

Verantwortungen beziehen sich auf

- Verpflichtungen eines Objektes,
- Das Verhalten eines Objektes

Wir trennen 2 verschiedene Arten von einander

Aktive Verantwortungen

- Instanz Erstellung
- Instanziierung einer Aktion oder eines Objektes
- Koordinierung und Steuerung von Activities und anderen Objekten

Implizite Verantwortungen

Wissen über

- private gekapselte Daten
- verwandte Objekte

Kopplung: Misst die Stärke der Abhängigkeiten zwischen Klassen und Packages.

- Klasse C1 ist mit C2 gekoppelt, wenn C1 direkt oder auch indirekt C2 benötigt
- Eine Klasse die an 2 andere gekoppelt ist, hat eine geringere Kopplung als eine die mit 8 gekoppelt ist.

Hohe Kopplung

Eine Klasse mit hoher Kopplung ist in der Regel unerwünscht, da

- Änderungen einen extrem großen Aufwand mit sich ziehen
- Sie als einzelnes viel schwerer zu verstehen sind
- Die Wiederverwendbarkeit darunter leidet, weil sie viele Sachen sich aus anderen Klassen holt

Geringe Kopplung

Eine Klasse mit geringer Kopplung ist beinahe unabhängig und unterstützt das Design, daher viel leichter wiederverwendbar.

- Generische Klassen mit einer hohen Wahrscheinlichkeit der Wiederverwendung sollten eine geringe Kopplung haben
- Keine oder minimale Kopplung ist auch nicht erwünscht
- **Hohe Kopplung um die Robustheit von Elementen zu gewährleiten ist selten ein Problem**
- **Niedrige Kopplung ist eine beinahe mystische Aufgabe**

Kohäsion: Misst die Stärke der Beziehungen zwischen Elementen von Klassen.

- Alle Daten und Operationen sollten zum Konzept welche die Klasse modelliert im Zusammenhang stehen

Verschiedene Arten von Kohäsion

- Zufällig: Keine bedeutenden Relationen zwischen Klassenelementen
- Temporär: Alle Elemente einer Klasse werden „gemeinsam" ausgeführt
- Sequentiell: Ausgabe einer Methode ist die Eingabe in eine andere Methode (***Piping***)
    - Funktional: Alle Elemente der Klasse tragen zum Erreichen wohldefinierter Verantwortungen /Ziele bei

Geringe Kohäsion ist unerwünscht, Klassen mit geringer Kohäsion schwer zu verstehen sind, es ihr an Wiederverwendbarkeit mangelt, in der Wartung aufwändig sind etc….

Klassen mit hoher Kohäsion können oft mit nur einem Satz beschrieben werden.

LCOM – Lack of Cohesion of Methods

Sei C eine Klasse, F die Menge der Instanzvariablen von C und M die Menge der Methoden. Man stellt sich einen einfachen ungerichteten Graphen $G(V, E)$ mit den Kanten $V = M$ wobei die Kanten

$$E = \{< m, n > \epsilon\, V \times V | \exists\, f\, \epsilon F : (m\ greift\ auf\ f\ zu\ und\ n\ greift\ auf\ f\ zu).$$

LCOM(X) ist dann definiert als Anzahl der stark miteinander verbundenen Komponenten (SCC) von G.

Der LCOM ist immer ein nicht-negativer Integer Wert, welcher sich zwischen [0, n] bewegt, wobei n die Anzahl der Methoden widerspiegelt. Umso höher LCOM(C), desto geringer die Kohäsion.

Beispiel:

```
public class SimpleLinkedList {

 private final Object value;
 private final SimpleLinkedList tail;

 public SimpleLinkedList(Object value,
                         SimpleLinkedList tail) {
   this.value = value;
   this.tail = tail;
 }

 public Object getValue() { return value; }

 public SimpleLinkedList getTail() { return tail; }
}
```

LCOM = 1

```
import java.awt.Color;
abstract class AbstractColorableFigure
 implements Figure {
  private Color lineColor = Color.BLACK;
  private Color fillColor = Color.WHITE;

  public Color getLineColor() { return lineColor; }

  public void setLineColor(Color c) {
    lineColor = c;
  }

  public Color getFillColor() { return fillColor; }

  public void setFillColor(Color c) {
    this.fillColor = c;
  }
}
```

LCOM = 2

Um die Komplexität des Designs im akzeptablen Rahmen zu halten, sollte man Verantwortungen zuweisen, während man versucht eine hohe Kohäsion weiterhin aufrecht zu erhalten.

Klassen mit niedriger Kohäsion sind unerwünscht, weil…

- Sie schwer zu verstehen sind,
- schwer wiederverwendbar sind und
- aufwändig in der Wartung sind + bei jeder kleinsten Änderung im Projekt angepasst werden müssen
- Sie oft eine grobkörnige Abstraktion repräsentieren
- Sie Verantwortungen übernehmen, die zu anderen Klassen besser passen

➔ **Klassen mit einer hohen Kohäsion können oft mit nur einem Satz beschrieben werden**

Design benötigt Richtlinien - Quadrat-Rechteck Problem

Ist ein Rechteck ein Quadrat Ein?

- Ist ein Rechteck immer auch ein Quadrat? → Nö

Ist Quadrat ein Rechteck?

- Gewissermaßen schon. Wie sieht es mit dem Verhalten aus?
  - *Logisch gesehen schon, aber als Code weißt zum Beispiel die Flächenberechnung erste Unterschiede auf*

Eine Anzahl an Design Heuristiken und Prinzipien helfen einem, ein „gutes“ Design zu entwerfen

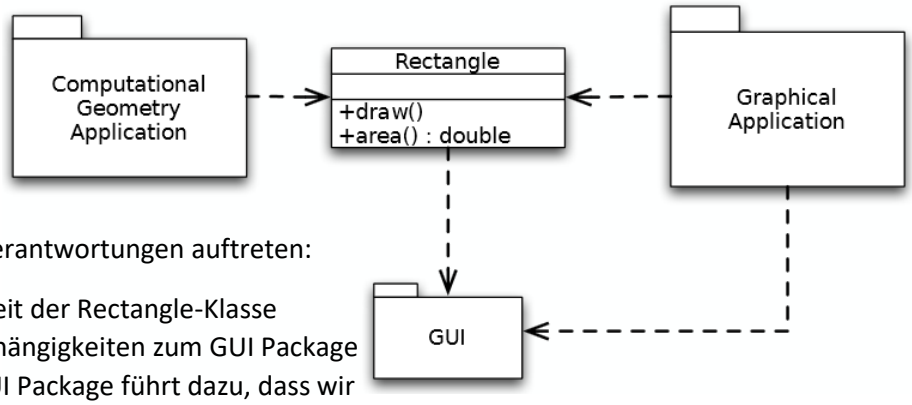
- Niedrige Kopplung
- Hohe Kohäsion
- Eine-Verantwortungsregel (Eine Klasse hat eine Verantwortung)
- Wiederholungen vermeiden
- Zyklische Abhängigkeiten vermeiden

Eine-Verantwortungsregel – Hat die Rectangle-Klasse eine oder mehrere Verantwortungen?

Die Rectangle-Klasse hat mehrere Verantwortungen.

- Mathematische Größe berechnen
- Rendern des Rectangles, GUI bezogene Funktionalitäten

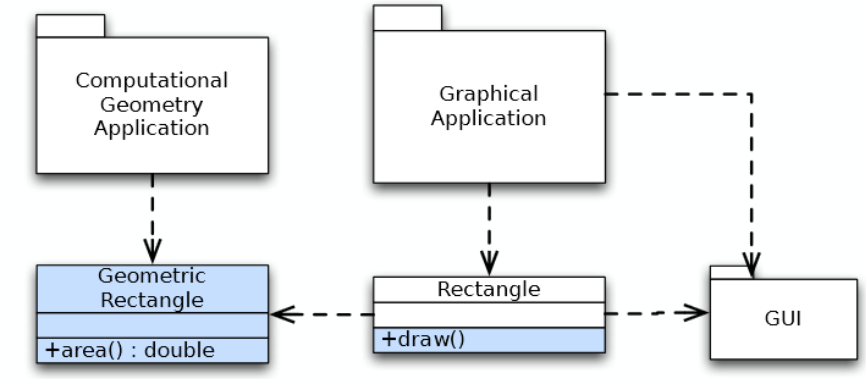
- **Probleme in Sicht?**



Probleme die bei Mehrfachverantwortungen auftreten:

- Wiederverwendbarkeit der Rectangle-Klasse ist schwer wegen Abhängigkeiten zum GUI Package
- Eine Änderung im GUI Package führt dazu, dass wir Unsere Rectangle-Klasse neu testen und aufsetzen müssen

VERBESSERUNG – Anwendung der Ein-Verantwortungsregel

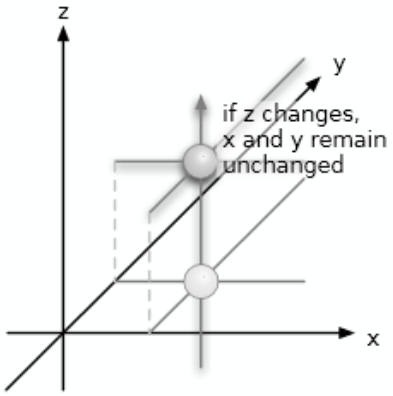


Die Lösung besteht darin, die zeichnenden Methoden und rechnenden Methoden in separate Klassen zu verfrachten

Orthogonalität

Zwei Dinge sind orthogonal zueinander, wenn Änderung in einem Ding das andere Ding nicht beeinflussen. DingDing!

➔ Wenn Änderungen in der GUI Klasse die Datenbankklasse nicht beeinträchtigen, sind diese orthogonal zueinander



Design Heuristiken

- Helfen bei der Frage *„Ist dieses Design gut, schlecht, oder irgendwas dazwischen?“*
- OO-Design Heuristiken helfen bei der Einsicht in OO –Designoptimierungen
- Die folgenden Richtlinien sind komplett sprachenunabhängig und erlauben das Bewerten von Software Design
- Heuristiken sind keine verbindlichen Regeln; Sie dienen als Warnmechanismus und erlauben einem die ein oder andere Heuristik zu ignorieren
- Heuristiken sind nur kleine Feinoptimierungen

2 Bereiche, wo uns OO-Paradigmen in hässliche Situationen bringen können

- Schlecht verteilte Systemintelligenzen
    - God Class Problem

- Erstellen von zu vielen Klassen im Vergleich zum Design
  →Proliferation (starke Vermehrung) von Klassen

Eine sehr grundlegende Heuristik

Daten in einer Basisklasse sollten private sein;
nutze nie nicht-private Daten!

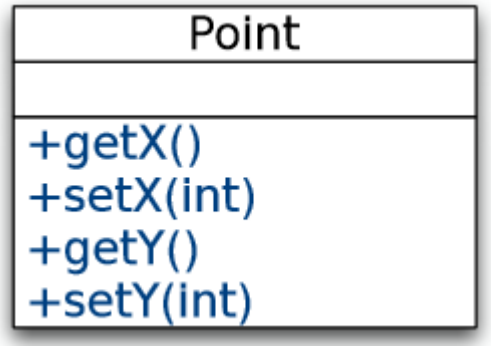
Foliensatz 10 – God Class Problem – Design Heuristiken 2

God Class Problem

- Verteile Systemfunktionalitäten so gleichmäßig wie möglich, sodass Top-Level Klassen die Arbeit gleichmäßig teilen und verrichten
- Vorsicht ist vor Klassen mit vielen in öffentlichen Interfaces definierten Accessormethoden geboten, da dies ein Anzeichen auf eine schlechte Daten- und Verhaltensdistribution ist
- Vorsicht ist auch vor Klassen mit viel *„stummen“* Verhalten, sprich die Methoden arbeiten nur auf einem Teil der Klasse
  → God Klassen enthalten viele solcher

Das Problem mit den Accessormethoden

Wir betrachten die Klasse Point:



- Sie besitzt Accessormethoden in dem Interface welches sie implementiert; Designprobleme erkennbar?
- Gibt die Klasse zu viele Informationen an den Kunden weiter?

➔ Nö, da sie nicht wirklich Implementierungs - details verraten

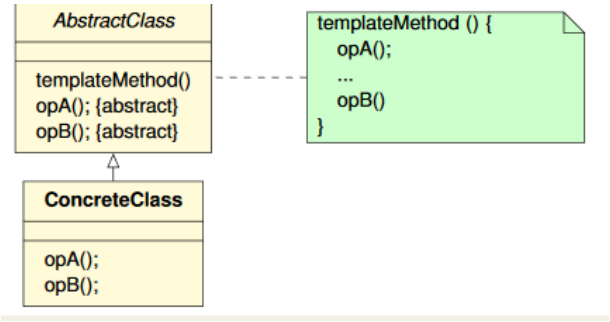
- Accessormethoden decken zeigen, dass es sich um eine schlechte Datenkapselung von relevanten Daten und Verhalten; Jemand holt sich die x und y-Koordinate - Welche aus verhaltenstechnischer Sicht zu Point gehört – mit Methoden die die Klasse nicht bereitstellt
- Ein Kunde welcher auf Daten zugreift, nutzt eigentlich die God Class welche auf die sinnlose Pointklasse zugreift um die Daten zu bekommen

2 Gute Gründe welche die Nutzung von Accessormethoden befürworten

- Eine Klasse die gets und sets ausführt implementiert ein Verfahren
- Oder ist einer Interfacesektion welche sich aus dem OO-Modell und einem Nutzerinterface zusammensetzt
    - Die UI-Layer sollte in der Lage sein, auf die Daten zugreifen zu können

Design Patterns – Template Pattern

Ziel: Implementieren von Algorithmen, sodass sie im Nachhinein leicht verändert werden können.



Definiert Algorithmen mittels abstrakten UND konkreten Operationen.

Gewinn:

- Separation von Variante und Invariante
- Verhindert Code Duplikationen in Subklassen
- Erweiterungen von Subklassen können gesteuert werden

Architekturielle Pattern

Eine Beispiele:

- Pipes und Filter
- Broken Pattern
- Client-Server Pattern
- Model-View-Controller

Model-View Controller

Beschreibt die grundlegende strukturelle Organisation interaktiver Softwaresysteme

Beinhaltet die Kernfunktionalitäten und Daten

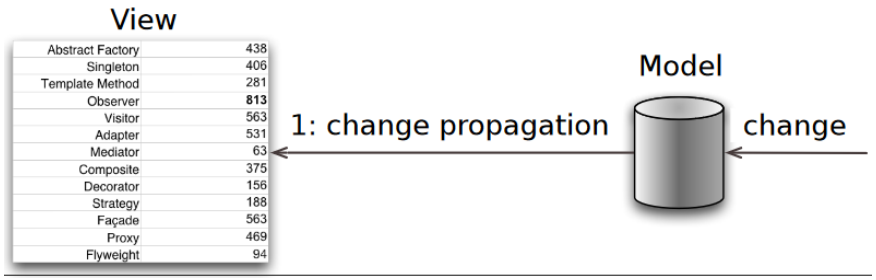
- Model ist unabhängig von Output oder Input Funktionalitäten

Das Userinterface wird folgende Aspekte gefährdet:

- Anzeigeelemente, welche Informationen visualisieren
  - Anzeigeelemente erhalten Daten vom Model
- Controller verarbeiten Nutzereingaben
  - Jedes Anzeigeelement besitzt einen Controller
  - Controller bekommt Input, die Events werden dann in Serviceanfragen übersetzt, welche dann vom Model/View verarbeitet werden können
  - Alle Interaktionen gehen durch den Controller

Änderungsverarbeitung

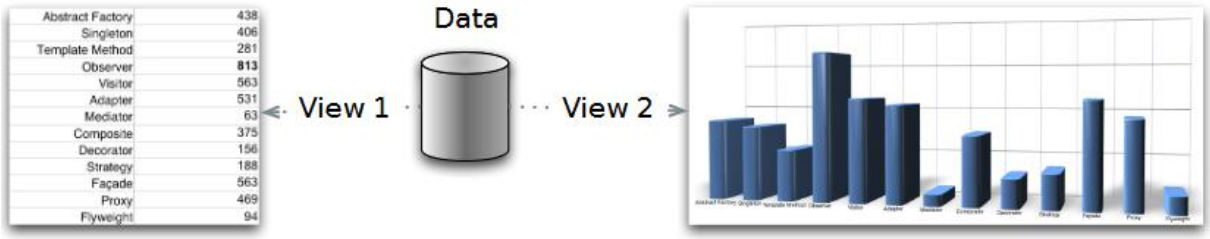
Der integrierte Änderungsmechanismus garantiert Konsistenz zwischen dem Userinterface und Model. (Wird normalerweise mit dem Observer / Publisher-Subscriber Pattern realisiert).



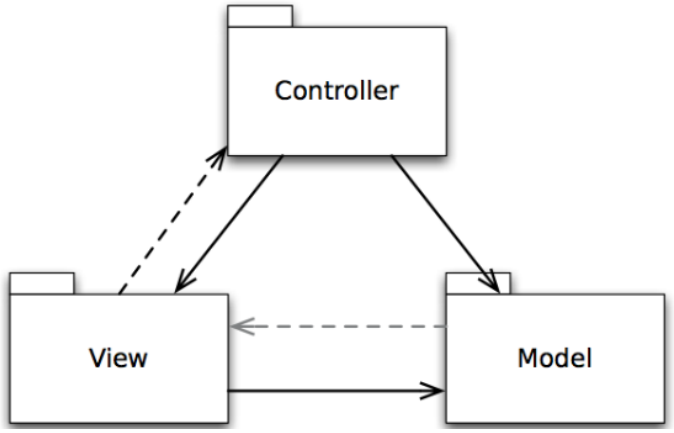
MVC – Anwendung

Empfohlene Nutzung beim Entwickeln von Interaktiven Anwendungen mit einem flexiblen Userinterface

- Selbe Information auf verschiedene Art und Weise darstellen (auch verschiedene Fenster)
- Anzeige und Verhalten der Daten sollte Datenänderung in Echtzeit reflektieren
- Änderungen am Userinterface sollten nicht den Core-Code (Model) beeinträchtigen



Allgemeine Struktur des MVC



Controller und View sind **direkt mit dem Model gekoppelt**, aber Model **ist nicht direkt an Controller und View gekoppelt**.

MVC- Verpflichtungen

Welche Kompromisse müssen wir eingehen?

- **Erhöhte Komplexität:** Getrenntes Nutzen von View- und Controller Komponenten kann die Komplexität erhöhen ohne großartig die Flexibilität zu verbessern
- **Updateverbreitung:** Potenzial für deutlich erhöhte Aufrufe der Update Routine – Nicht alle Views sind an alle Änderungen interessiert
- **Kopplung von View und Controller:** Deutet Verbindung zwischen

Motivation

- Entwerfen von wiederverwendbarer Software ist schwer
- Neulinge sind überfordert
- Experten berufen sich auf Erfahrung
- Einige Designs tauchen wieder auf

Architekturelle Pattern sind keine Design Patterns. Sie helfen bei der Spezifizierung der fundamentalen Systemstruktur und haben einen großen Einfluss auf die Erscheinung von konkreten Softwarestrukturen.

Vorteile von Design Patterns

- Systematische Softwareentwicklung
    - Fachwissen dokumentieren
    - Nutze genersiche Lösungen
    - **Erhöhen des Abstraktionslevels**

Essentielles

- Ein Pattern hat einen Namen, you dont say
- Das Problem sollte wiederauftauchen, damit das Pattern wiederverwendbar bleibt
- Es sollte möglich sein, das Pattern auf ein ähnliches Problem zu übertragen

Elemente eines Pattern

- **Name:** Ein Name der dazu passt
- **Problem:** Beschreibt wann das Pattern angewendet wird
- **Lösung:** Die Elemente samt ihrer Assoziation, Relationen und Verantwortungen
- **Konsequenzen:** Kosten und Vorteile der Nutzung eines Patterns

1.	▶ Name ▶ Intent
2.	▶ Motivation ▶ Applicability
3.	▶ Structure ▶ Participants ▶ Collaboration ▶ Implementation
4.	▶ Consequences
5.	▶ Known Uses ▶ Related Patterns

Foliensatz 11 - Strategy Design Pattern (SDP)

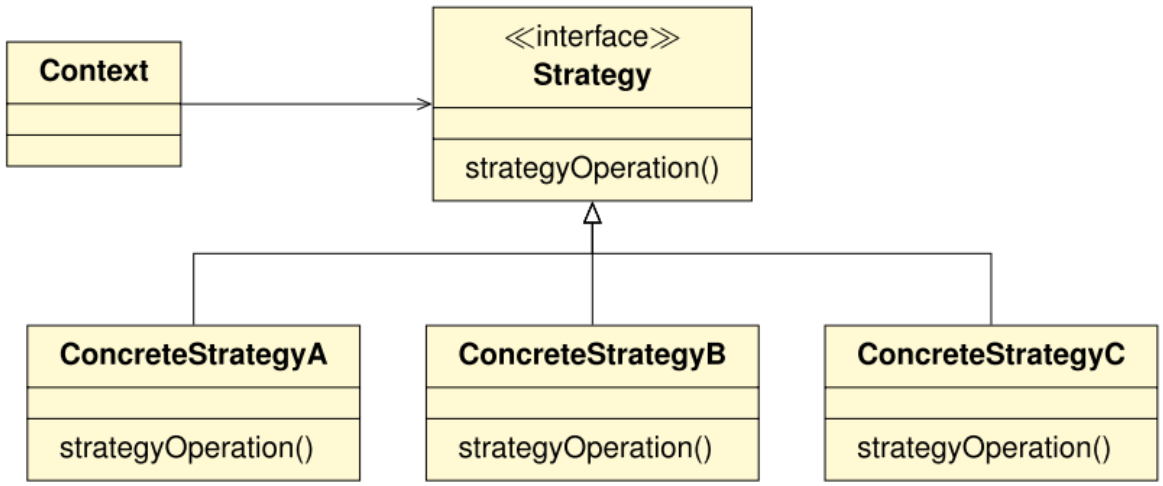
Motivation

- Viele verschiedene Klassen unterscheiden sich nur im Verhalten anstatt der Implementiertung
- Verschiedene Varianten von Algorithmen können in einer Hierarchie implementiert werden.

Vorteile

- Unterstützt viele Drittanbieterdienste für Steuerberechnungen
- verschiedene Datenbankanbindungen

- Ermöglicht verschiedenste Elemente zu sortieren



Die Absicht besteht darin, eine Familie von Algorithmen zu definieren, sie abzukapseln und austauschfähig zu machen. Das SDP lässt diese Algorithmen unabhängig voneinander variieren.

SDP – Eine Alternative zu Subclassing

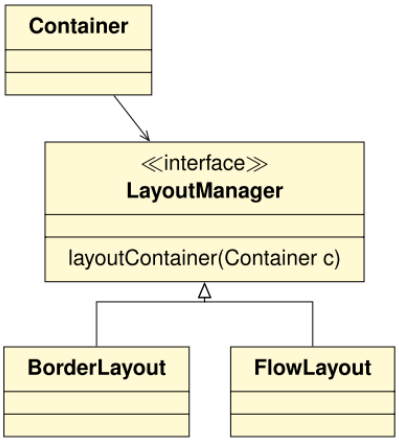
- Der Kontext des Subclassings vermischt sich mit denen der implementierten Algorithmen
- Kontext schwerer zu verstehen, warten und erweitern
- Beim Subclassing können wir Algorithmen nicht dynamisch variieren lassen
- Subclassing läuft auf viele Klassen mit vielen Beziehungen hinaus
    - Sie unterscheiden sich nur um Algorithmus oder im Verhalten

Kapselung in SDP…

- Lässt einen alle Algorithmen unabhängig von den anderen variieren
- Wechseln, verstehen, wiederverwenden und erweitern von einem Algorithmus geht einfacher von statten

Andwendung des SDP in der Java AWT/Swing Klasse

```
public class Container extends Component {
 /** Sets the layout manager for this container.
  * @param mgr the specified layout manager
  */
 public void setLayout(LayoutManager mgr) {
  layoutMgr = mgr; invalidateIfValid();
 }
 /** Causes this container to lay out its
  components.*/
 public void doLayout() {
  LayoutManager layoutMgr = this.layoutMgr;
  if (layoutMgr != null) {
   layoutMgr.layoutContainer(this);
  }
 }
}
...
container.setLayout(new BorderLayout());
...
```



Wann sollte man das SDP nutzen?

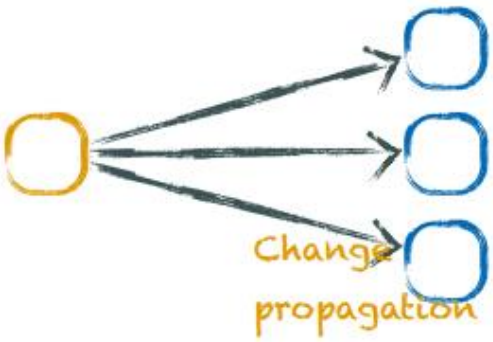
- Viele miteinander verbundene Klassen unterscheiden sich meistens nur im Verhalten als in der Implementierung

- Strategien erlauben einem das Konfigurieren einer Klasse mt einer oder mehreren Verhalten
- Man braucht verschiedene Varianten von Algorithmen; Strategien können genutzt werden, wenn Algorithmen als eine Klassenhierarchie implementiert werden
- Eine Klasse definiert verschiedene Verhalten in Form von Mehrfachverzweigungen in ihren Methoden

Konsequenzen

- Kunde muss sich über die verschiedenen Strategien und ihre Unterschiede im Klaren sein, um den die rchtige Strategie zu wählen
- Kunde wird eventuellen Implementationsproblemen ausgesetzt
- SP nur nutzen, wenn das Verhaltensmuster für den Kunden relevant ist

Observe Pattern



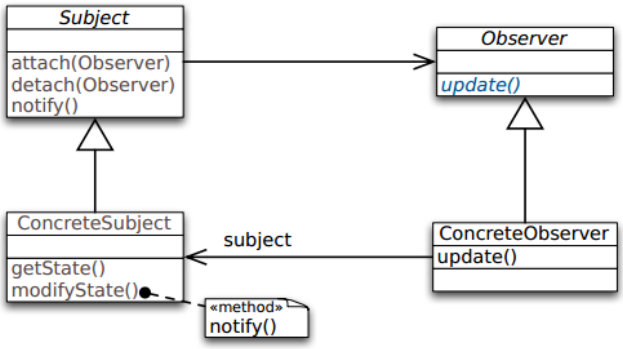
Ziel:

- Kommunikation ohne Kopplung
- Flexibel Weg benötigt

Vorteil: Jedes Objekt ist leichter zu implementieren und warten, erhöhte Wiederverwendbarkeit und flexible Kombis sind möglich

Nachteil: Verhalten ist über mehrere Objekte verteilt; Eine Änderung kann viele andere verbindliche Änderung mit sich ziehen

Struktur und Teile



- **Subject**
  - Kennt seinen Observer
  - Stellt Methoden für Hinzufügen/entfernen von Observerelementen zur Verfügung
- **Observer**
  - Definiert ein sich aktualisierendes Interface zur Benachrichtigung über ein sich änderndes Objekt
- **ConcreteSubject**
  - Speichert Interessenstand
  - Sendet Benachrichtung an den Observer bei Änderungen
- **ConcreteObserver**
  - Pflegt eine Referenz zu einem ConcreteSubject Object
  - Speichert den State welcher mit dem Objekt gleich bleiben soll
  - Implementiert Observerupdate Interface

Die konkrete Aufgabe beim Observer Pattern besteht darun, dass Datenmodell (Subjekt) von den an ihm interessierten „Parteien“ zu entkoppeln.

Anforderungen

- Subjekt sollte nichts von seinen Observern wissen
- Identität und Nummer der Observer ist nicht von vorn herein bestimmt
- Neue Enpfänger (Observer?) sollen in Zukunft hinzugefügt werden können
- *Polling* (Abfragen?) sind unerwünscht, sehr ineffizient

Grundproblem: Erzeugen einer Eins-Zu-Alle Relation zwischen Objekten

Konsequenzen

- Absrakte Kopplung
- Gefahr von Updateproblemen
- Update werden an ALLE (auch irrelevanten) Observern
- Keine Details über geänderte Daten
- Subjekt kann keine optionalen Informationen an den Observer ändern

Implementierung

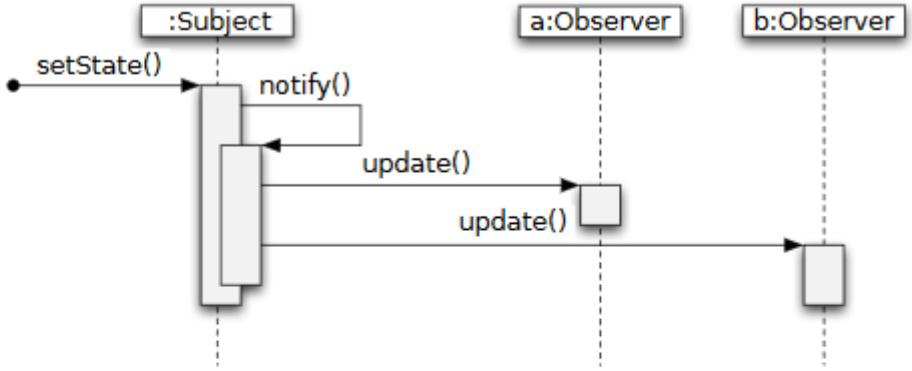
```
public class Subject {
 private List<Observer> observers;
 ...
 public void attach(Observer o) {
  if (o == null)
    throw new IllegalArgumentException();
   observers.add(o);
 }

 public void detach(Observer o) {
  if (o == null)
    throw new IllegalArgumentException();
  observers.remove(o);
 }
}
```

Remarks:

- observers are added and removed from a list of observers
- if an object of the model class (Subject or sub class) can be used by different threads: synchronization necessary

Änderungsbenachrichtigungen




```
private List<Observer> observers;
...
void notify() {
 for (Observer o : observers.clone()) {
   o.update();
 }
}
```

Vorteile:

- **Abstrakte Kopplung zwischen Subjekt und Observer**
- **Unterstützt Broadcast Kommunikation**
    - Benachrichtigung gibt keinen expliziten Empfänger an
    - Sender kennt den (konkreten) Empfägnertypen nicht

Nachteile:

- **Nicht alle Observer sind an einem Update interessiert**
- ***Update Kaskaden***
- **Keine Information darüber, was sich geändert hat, Observer muss selbst herausfinden was geändert wurde**
- **Subjekt kann keine optionalen Parameter an Observer senden, da Interface limitiert**

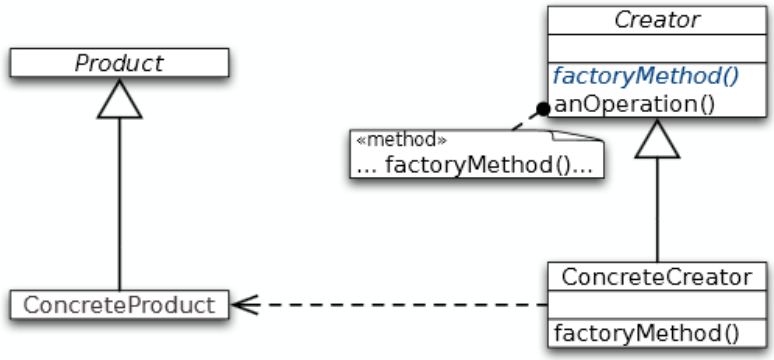
Foliensatz 12 – Factory Design Pattern (FDP)

Absicht: Interface zur Erzeugung von Objekten wird definiert. Lasse aber Subklassen entscheiden welche Klasse instanziiert wird.

Wo wird es angewandt?

- Texteditoren
- Wortprozessoren
- Vectordarstellungsprogamme
- Dokumentbetrachter

Struktur



Teile

- **Produkt**
    - Definiert das Interface der Objekte welche die Factory Methode erstellt
- **ConcreteProduct**
    - Implementiert das Produktinterface
- **Creator**
    - Deklariert die Factorymethode, welche ein Objekt vom Typ Product zurückliefert.
    - Definiert möglicherweise eine Standardimplementation der Factorymethode welche ein Standard ConcreteProduct Objekt zurückliefert,
- **ConcreteCreator**

- Überschreibt die Factorymethode , um eine Instanz eines ConcreteProducts zurückzuliefern

Konsequenzen

- Interface-Code arbeitet nur mit dem Produktinterface
- Stellt einen „Haken“ für Subklassen bereit
  - *Kann genutzt werden, um erweiterte Versionen von einem Objekt zu erstellen*

2 große Varianten

- Creator ist abstrakt
- Creator ist konkret und stellt eine durchdachte Standardimplementation zur Verfügung

Parametrisierte Factorymethoden

```
public abstract class Creator {
      public abstract
            Product createProduct(ProductId pid);
}
```

Ähnliche Patterns

Factory Methoden werden normalerweise in Template Methoden aufgerufen

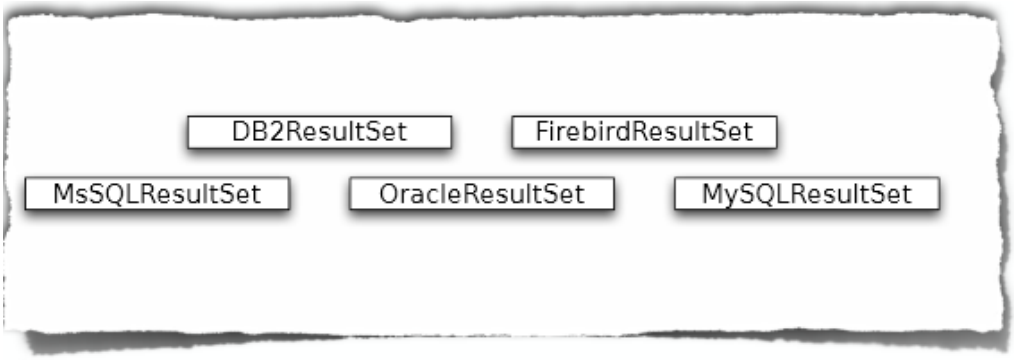
Abstrakte Factory Methoden werden oft mit Factory Methoden implementiert

Abstraktes Factory Design Pattern (AFDP)

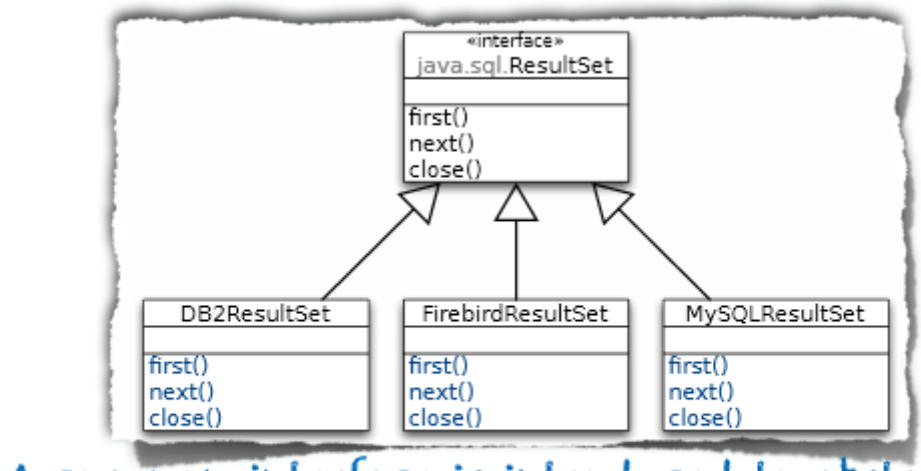
Wie erstellt man ein Interface, welche eine ganze Familie an Objekten beschreibt.

Ziel:

- Unterstützung von verschiedenen Datenbanken
- **Anforderungen**
  - *Die Anwendung sollte mehrere Datenbanken unterstützen*
  - *Wir möchten später noch mehr Datenbanken unterstützen*



How to provide an interface to all of these different kinds of ResultSets?

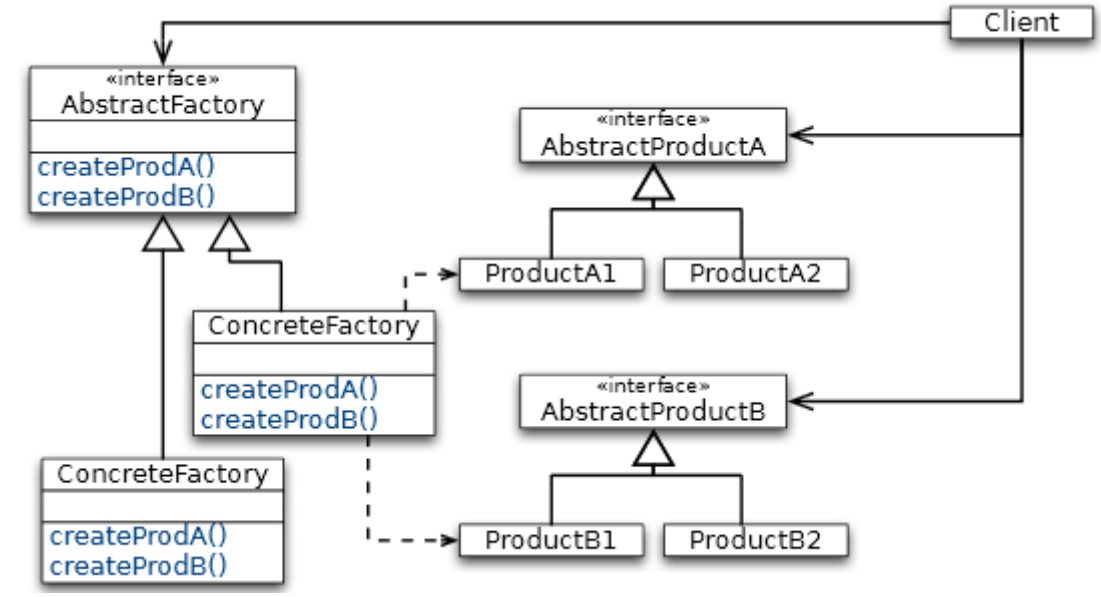


A common interface is introduced to abstract from the concrete classes.

Motivation / Beispielszenario

Um die Abstraktion einer Datenbank zu komplettieren, muss man noch eine Klassenhierachie erstellen. Diese gilt für `CallableStatements, PreparedStatements, Blobs,…`

Der Code interagiert mit der Datenbank und kann nun mit `ResultSet` and SQL Statements ohne Bezug auf konkrete Klassen arbeiten. **Struktur**

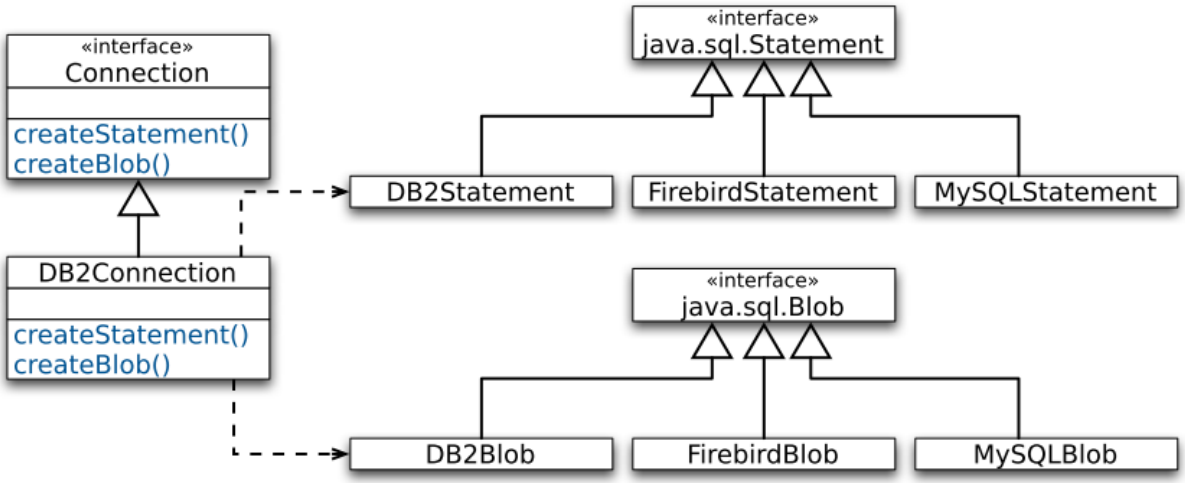


Teile

- **`AbstractFactory`**
  - Stellt ein Interface für die Erstellung einer Produktfamilie zur Verfügung
- **`ConcreteFactory`**
  - Implementiert die Operationen zum Erstellen von ConcreteProduct Objekten
- **`AbstractProduct`**
  - Deklariert das Interface für ConcreteProduct
- **`ConcreteProduct`**
  - Stellt eine Implementierung der Produkte welche die ConcreteFactory werden müssen zur Verfügung

- **`Client`**
    - Erstellt Produkte durch Aufrufen der ConcreteFactory, nutzt das AbstractProduct Interface

Angewandtes AFP



Konsequenzen

- Das Ändern der Produktfamilien ist einfach
- Sichert die Konsistenz zwischen Produkten
- Unterstüzung von neuen Produktfamilien ist schwer
- Objekterstellung ist kein Standard