# SWT Panikzettel

Philipp Schröer, Caspar Zecha, Luca Oeljeklaus,
Christoph von Oy, “der Dude”

5. April 2018

Dieser Panikzettel ist über die Vorlesung Softwaretechnik. Er basiert auf dem Foliensatz von Prof. Dr. Bernhard Rumpe aus dem Wintersemester 16/17.
Dieser Panikzettel ist Open Source. Wir freuen uns über Anmerkungen und Verbesserungsvorschläge auf `https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

## Inhaltsverzeichnis



*Pseudonyme gehören anonymen Autoren, die anonym bleiben wollen.

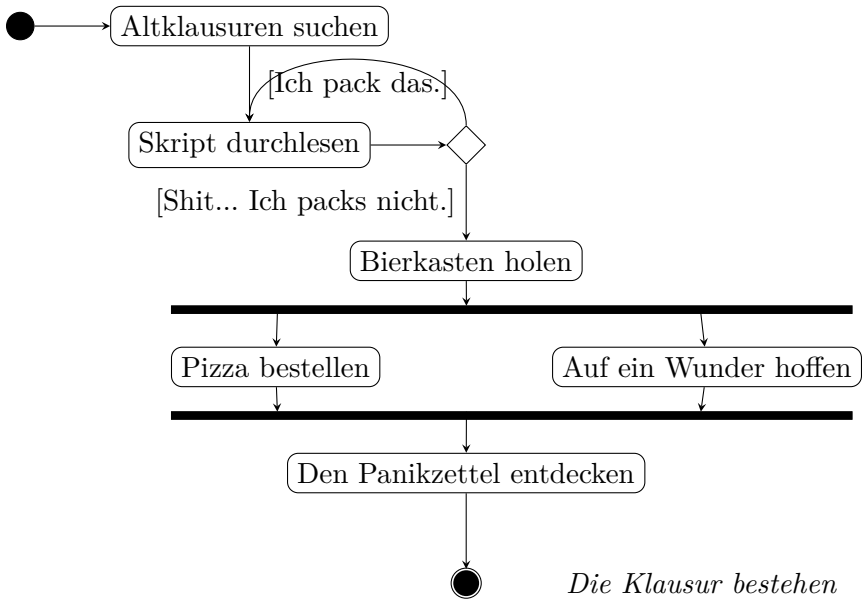
# 1 Einleitung: Don't Panic!

Es scheint, als hätten selbst die Panikzettel-Autoren Dich verlassen. 27 Seiten! Wer soll das Lernen? Verzweifle nicht. Wir haben hier nur etwas anders gemacht als sonst: Wir haben fast alle Stichworte aus der Vorlesung mal erwähnt, von denen man wahrscheinlich in der Klausur was gehört haben sollten könnte. So hilft dir der Panikzettel auch bei maximal schlechter Vorbereitung.

Die meisten Abschnitte solltest du wahrscheinlich nur ein- oder zweimal gelesen haben, dann hast du wahrscheinlich den Großteil verstanden. Deinen Fokus solltest du auf die Abschnitte Diagrammtypen (2) und Vorgehensmodelle (3). Muster (6) werden auch relevant sein. Generative Softwareentwicklung (9) wurde außerdem besonders hervorgehoben.

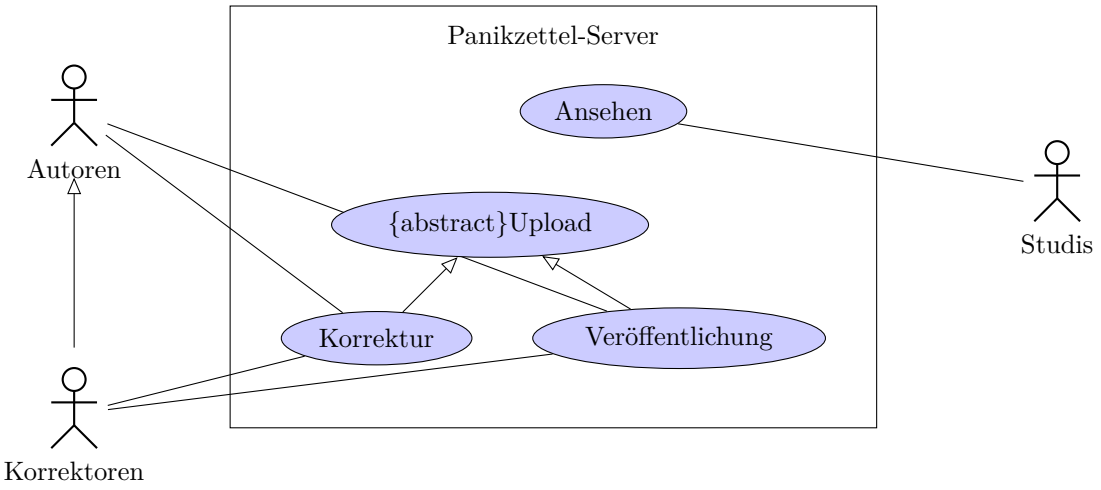
# 2 Diagrammtypen

## 2.1 Aktivitätsdiagramm




Diese Zeichnung nennt man *Aktivitätsdiagramm*, man nutzt es um Abläufe und Kausalitäten zu beschreiben. Rauten ermöglichen es, *Fallunterscheidungen* zu signalisieren, während die schwarzen Balken darauf hindeuten, dass verschiedene Ereignisse *parallel* stattfinden.

## 2.2 Use-Case-Diagramm

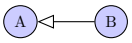



Im obigen Diagramm sieht man ein *Use-Case-Diagramm* für unseren Panikzettel-Server. Die drei *Akteure*, Autoren, Korrektoren[1] und Studis interagieren mit dem System und haben *nehmen an* Use-Cases *teil*. Die Korrektoren *erweitern* dabei die Akteure Autoren, und nehmen an allen Use-Cases der Autoren teil. Use-Cases können *spezialisiert* werden, wie etwa Upload durch die Korrektur spezialisiert wird. Der Use-Case Upload selber ist abstrakt und hält nur Gemeinsamkeiten zwischen anderen Use-Cases fest.

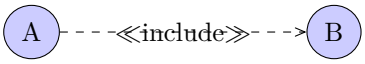
### 2.2.1 Use-Case-Beziehung

Es gibt verschieden Arten von Use-Case-Beziehungen:

**Spezialisierung:** (Im Diagramm oben).

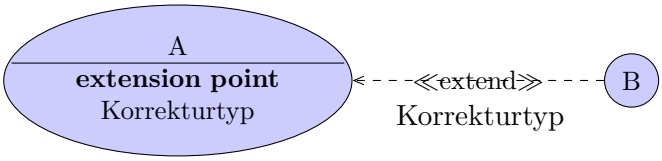



**Einbindung:** Use-Case `A` ruft `B` auf.

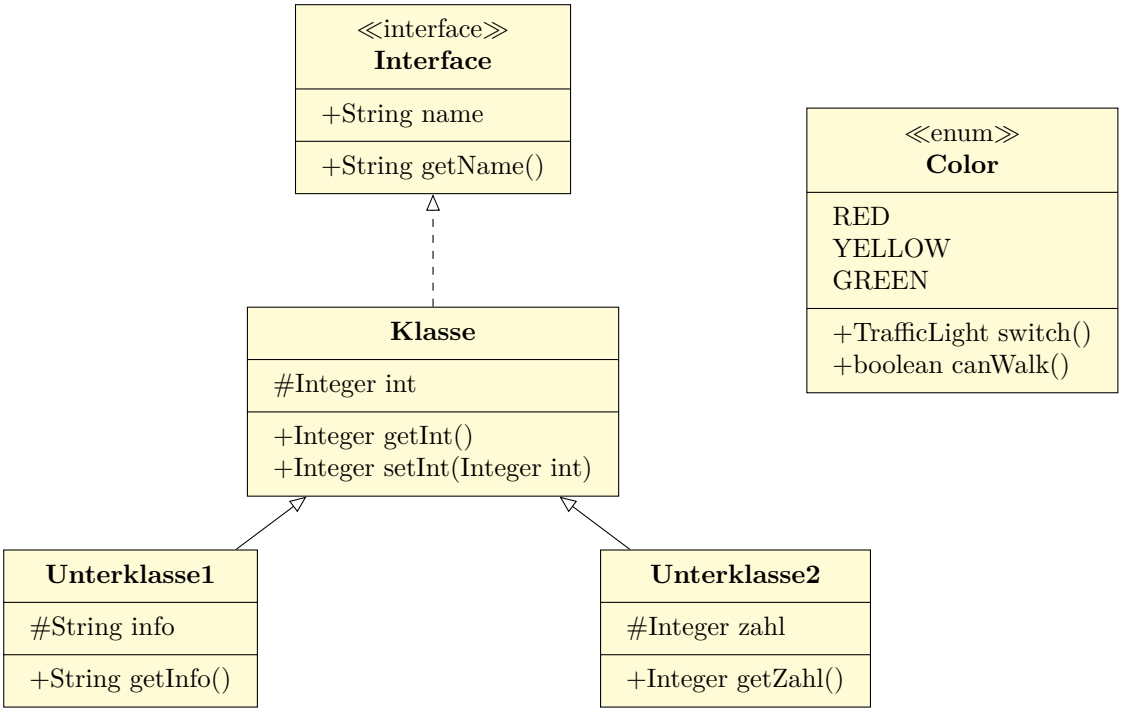



**Erweiterung:** Use-Case `A` wird durch `B` erweitert. Es muss spezielle *extension points* geben, an die die Erweiterungen anknüpfen. Hier heißt unser extension point “Korrekturtyp”.

[1] Weil wir keine Fehler machen, momentan nicht vorhanden.




## 2.3 Klassendiagramm

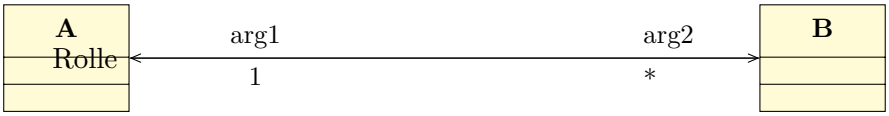



**Achtung**: In der Vorlesung wird im Gegensatz zum Panikzettel der Name abstrakter Klassen *kursiv* geschrieben. Es müsste daher im obigen UML-Diagramm ***Klasse*** statt **Klasse** heißen.

In diesem Klassendiagramm sind zwei Unterklassen die von der Klasse *Klasse* erben. Diese wiederum implementiert das Interface *Interface*. Außerdem ist oben rechts eine Enumerations-Klasse mit drei Konstanten und Funktionen auf diesen Konstanten.

Allgemein kann eine Klasse nur von einer (abstrakten) Klasse erben, aber von mehreren Interfaces. Ein Interface kann auch von mehreren Interfaces erben.

### 2.3.1 Assozationen



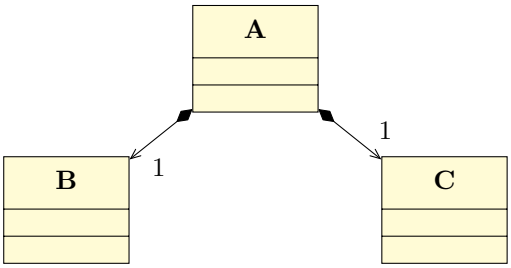

Diese Assozation verbindet die Klassen $A$ und $B$. Die Assozationsrolle von $B$ ist hier als *arg1* (Platzhalter) bzgl. $A$ bezeichnet bzw. als *arg2* bezeichnet. Die Kardinalität (wieviele "Instanzen" der Assoziation vorliegen) sieht man unterhalb des Assoziationspfeils und kann folgende Werte annehmen:

- genau a: a (oben z.B. 1)
- Intervall: a. . . b
- beliebig: * (z.B. im obigen Diagramm)

Eine geordnete Assoziation, gekennzeichnet durch *{ordered}*, gibt an, dass die Reihenfolge der Objekte von Bedeutung ist.

### 2.3.2 Komposition

Eine Komposition ist eine spezielle Form der Assoziation.

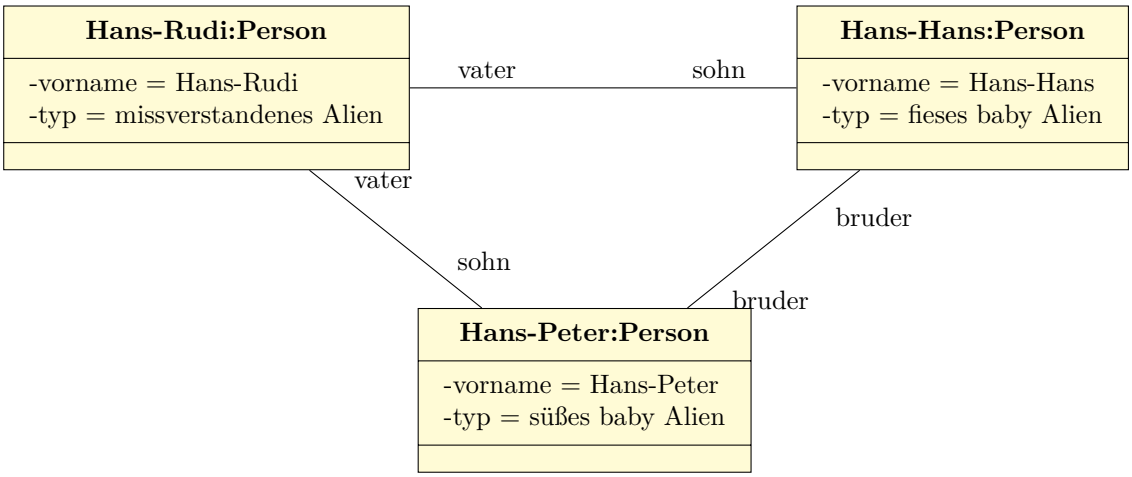



In diesem Klassendiagramm beinhaltet $A$ immer genau ein $B$ und $C$. Falls $A$, $B$ oder $C$ gelöscht werden, sollten alle Objekte gelöschte werden.

### 2.3.3 Sichtbarkeit

| UML | + | # | - | |
|---|---|---|---|---|
| Java | public | protected | private | (default) |
| Gleiche Klasse | ✓ | ✓ | ✓ | ✓ |
| Andere Klasse (gleiches Paket) | ✓ | UML: ✗, Java: ✓ | ✗ | ✓ |
| Andere Klasse (anderes Paket) | ✓ | ✗ | ✗ | ✗ |
| Unterklasse (gleiches Paket) | ✓ | ✓ | ✗ | ✓ |
| Unterklasse (anderes Paket) | ✓ | ✓ | ✗ | ✗ |

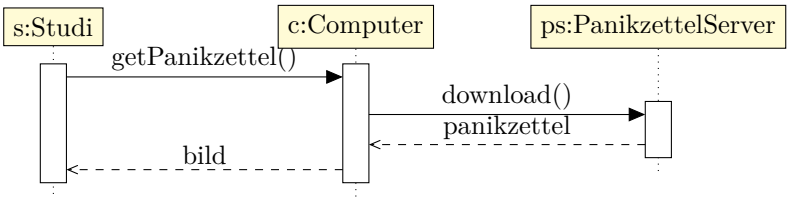
## 2.4 Objektdiagramm




*Objektdiagramme* sind im Endeffekt ähnlich zu Klassendiagrammen, bis auf die Tatsache, dass man *konkrete Objekte* ohne Methoden und deren Beziehungen darstellt.

## 2.5 Sequenzdiagramme

Wir können visualisieren, wie Objekte nacheinander verändert werden. Dazu verwenden wir *Sequenzdiagramme*. Zwischen den *Objekten*, für die die Zeit alle gleichmäßig nach unten voranschreitet, werden *Nachrichten* gesendet und empfangen.




Im obigen Beispiel sehen wir *Aktivierungsbalken*, die vertikalen Balken unter den Objektnamen, die die Lebenszeit eines Objektes darstellen. Diese Aktivierungsbalken können auch verschachtelt werden. Objekte können auch explizit erstellt werden, indem die Pfeile direkt auf den Namen (etwa "s:Studi") zeigen. Ebenso kann durch ein Kreuz auf der Linie unter dem Objekt explizite Objektlöschung eingezeichnet werden.

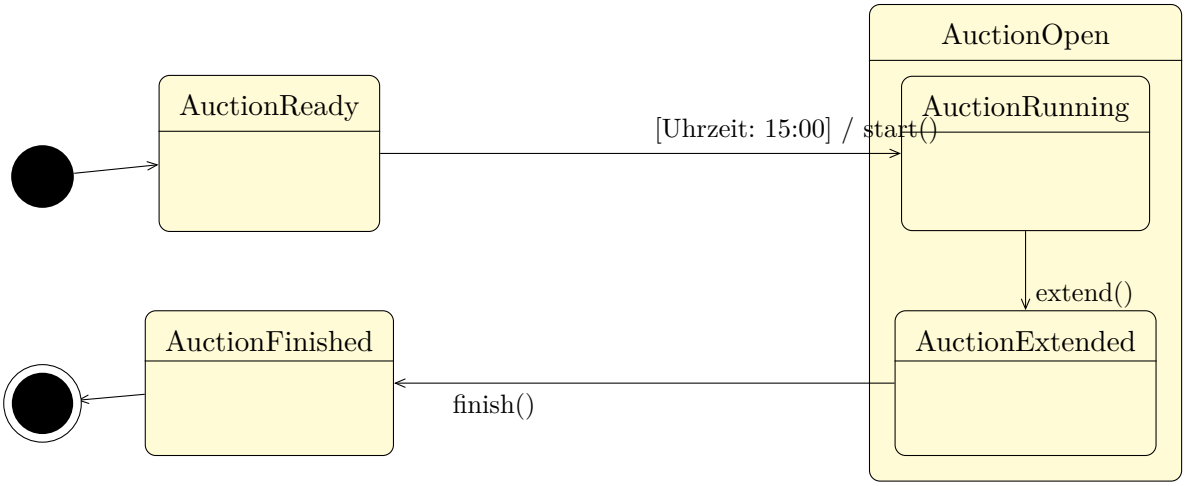
Zwischen den Objekten werden Nachrichten gesendet (oder auch zu sich selbst). Dabei gibt es vier Pfeilarten: Im Diagramm sind *synchrone* Pfeile zu sehen, wo es keine Verzögerung gibt. Die Pfeiler mit gestrichelten Linien sind *Returns* und stellen Rückgabewerten von Aufrufen dar. Außerdem gibt es noch *neutrale* Pfeile, diese stellen keine Festlegung des Kommunikationsmechanismus fest. *Asynchrone* Pfeile sind für Interaktionen, wo Senden und Empfang unterschiedliche Ereignisse sind (etwa bei SMS-Kommunikation).

Neutraler Pfeil: ⟶

Asynchroner Pfeil (wird schräg eingezeichnet): ⟶

## 2.6 Statecharts

*Statecharts* beschreiben das Verhalten von Objekten durch endliche Zustandsübergangsdiagramme (Vgl. endliche Automaten in FOSAP). Objekte haben einen endlichen Zustandsraum und Transitionen sind markiert mit `Ereignis [Bedingung] / Aktion`. Es gibt immer einen Start- und Endpunkt.




Wie man am Beispiel von `AuctionOpen` sieht, können Zustände verschachtelt werden.

**Achtung:** Wegen technischer Schwierigkeiten sind hier nicht-hierarchische Zustände auch mit Trennlinien gemalt, dies ist aber in den Folien so nicht vorgesehen!

Zusammengesetze Zustände können ihre eigenen Start- und Endpunkte sowie Transitionen haben, die von allen inneren Zuständen benutzt werden können.

## 2.7 Weitere Diagramme

Es gibt auch weitere Diagramme aus der Vorlesung, die möglicherweise relevant sein könnten:

- **Blockdiagramm:** Man zeichnet (möglicherweise geschachtelte) Blöcke, die Subsysteme strukturieren. Diese werden durch Linien, die Schnittstellen repräsentieren, verbunden.

- **UML-Implementationsdiagramm:** Die UML-Variante von Blockdiagrammen. Hier werden Namen unterstrichen in die Blöcke geschrieben. Schnittstellen sind Kreise, die mit durchgezogenen Linien an Blöcke angehangen werden. An diese können dann mit den Assoziationen (siehe 2.3) andere Blöcke angehangen werden.
- **Konfigurationsdiagramm:** Grobe schematische Zeichnung physikalischer Verteilung von Komponenten.
- **UML-Verteilungsdiagramm:** UML-Variante von Konfigurationsdiagrammen. Komponenten werden durch Linien verbunden. Die Namen haben zusätzlich "Stereotypen" wie etwa `<<processor>>` oder `<<network>>`.

# 3 Vorgehensmodelle

Es gibt verschiedene inhaltlich bestimmte *Aktivitäten*, die von zeitlich begrenzten *Phasen* zu unterscheiden sind:

1. Analyse
2. Entwurf
3. Implementierung
4. Test/Validierung
5. Deployment (Installation und etwa Schulung)
6. Evolution inkl. Wartung

**Das Wasserfall-Modell:** Analyse, Entwurf, Implementierung, Test und Wartung werden nacheinander ausgeführt. Eignet sich gut für gut im Voraus beschreibbare Projekte.

**V-Modell:** Zur Qualitätssicherung werden verschiedenen Entwurfsphasen Tests gegenüber gestellt. Die Entwurfsphasen gehen von oben nach unten, die Tests können dann in umgekehrter Reihenfolge durchgeführt werden.

1. Analyse: Abnahmetest
2. Grobentwurf: Systemtest
3. Feinentwurf: Integrationstest
4. Implementierung: Modultest

**eXtreme Programming:** Evolutionäre Entwicklung in kleinen Schritten. Tests und Programmcode als Analyseergebnis. Tests werden oft geschrieben, bevor die Implementierung beginnt. Für kleine Projekte geeignet.

**Scrum:** "Agile Methode". Inkrementell und iterativ entwickeln und durch Transparenz, Überprüfung und Anpassung während jedem Zeitpunkt der Entwicklung. Die drei verschiedenen Aktivitäten, "Sprint Planning", "Sprint Review" und "Daily Scrum" sind zeitlich fest beschränkt. Ein Sprint dauert i.d.R. 30 Tage und versucht den "Sprint Backlog", einen Teil des "Produkt-Backlogs" für diesen Monat, zu entwickeln. Ein sogenanntes "Burndown-Chart" visualisiert detailliert den täglichen Fortschritt im Vergleich zum geplanten Fortschritt.

# 4 Anforderungsanalyse

## 4.1 Anforderungsermittlung

Dies ist die erste Hälfte der Analyse und soll vor der Systemmodellierung feststellen, was überhaupt verlangt wird, die Ergebnisse werden im Pflichtenheft festgehalten.

**Funktionale Anforderung:** Was soll das System tun? Beispiel: Das System soll Nutzer identifizieren können.

**Nicht-funktionale Anforderung:** Wie sollen die funktionalen Anforderungen umgesetzt werden? Unter diese Kategorie fallen verschiedenste Aspekte wie etwa Effizienzanforderungen, Zuverlässigkeitsanforderungen, Anforderungen an den Entwicklungsprozess selber oder juristische oder ethische Anforderungen.

Mischfälle können in in funktionale und nicht-funktionale Anforderungen zerlegt werden.

**Geschäftsprozess:** Die zeitlich-logische Abfolge von Tätigkeiten zur Erfüllung einer betriebswirtschaftlichen Aufgaben.

Für *grobe* Geschäftsprozessmodelle können wir sehr einfache Flussdiagramme verwenden. Bei *detaillierten* Geschäftsprozessmodellen werden UML-Aktivitätsdiagramme (2.1) verwendet.

## 4.2 Anforderungsmodellierung

**Anwendungsfall:** Beschreibung einer Klasse von Aktionsfolgen, die ein System ausführen kann, wenn es mit Akteuren interagiert.

**Akteur:** Beschreibung einer Rolle, die ein Benutzer in Verbindung mit dem System spielt. Kann auch etwa ein anderes System sein.

Mit Use-Case-Diagrammen (2.2) organisieren wir solche Anwendungsfälle.

### 4.3 Modellierung von Aktivitäten

Eine Verfeinerung der oben genannten Use-Case-Diagramme sind Aktivitätsdiagramme. Wir verweisen hier wieder auf Abschnitt 2.1.

### 4.4 Prototyping

Wir unterscheiden zwischen drei Arten von Prototypen:

- Explorativ: Nur zur Analyse verwendet.
- Experimentell: Zur Analyse, Entwurf und Implementierung verwendet, aber der experimentelle Prototyp wird selber nicht weiter benutzt.
- Evolutionär: Wird weiter zum System ausgebaut.

## 5 Systemanalyse und Systemmodellierung

### 5.1 Objektorientierte Analyse (OOA)

Man betrachtet, wie Objekte miteinander interagieren. Einmal *statisch* mit etwa Klassendiagrammen (2.3) oder *dynamisch* über Objektdiagramme (2.4), welche Zustände und Verhalten betrachten.

1. Klassen finden
2. *Iteration:*
    a) Assoziationen und Aggregationen finden
    b) Attribute finden
    c) Operationen finden, Szenarien finden und überprüfen
3. Vererbungsstrukturen finden
4. Zustandsdiagramme erstellen
5. Operationen spezifizieren
6. Strukturen überprüfen
7. Subsysteme finden

## 5.2 Grundlagen der Objektorientierung

Ein System besteht aus variabel vielen *Objekten*. Objekte haben definiertes *Verhalten*, *inneren Zustand* und eine eindeutige *Identität*. Ein Objekt gehört zu einer *Klasse*, die ein Verhaltensschema und Struktur vorgibt. Klassen können vererbt werden. *Polymorphie* ermöglicht differenziertes Verhalten von Objekten abhängig davon, zu welcher Unterklasse ein Objekt gehört.

## 5.3 CRC-Karten

Eine Technik zur Gruppenarbeit, wobei auf Karteikarten auf die Vorderseite Ober- und Unterklassen, Verantwortlichkeiten und Mithelfer eingetragen werden, auf der anderen Seite die Klasse in Form der UML-Klassendiagrammnotation (2.3) aufgetragen wird.

## 5.4 Szenarien

**Szenario:** Eine beispielhafte Folge von Interaktionen von Akteuren mit dem System zur Beschreibung eines Anwendungsfalls. Hier gibt es die Unterscheidung zwischen *Normalfällen* und *Ausnahmefällen*.

Für die Beschreibung von Szenarien verwenden wir Sequenzdiagramme (2.5).

## 5.5 Modellierung mit Statecharts

Objektverhalten soll durch einen endlichen Zustandsraum und Transitionen modelliert werden. Auf die verwendeten Statecharts gehen wir in Abschnitt 2.6 genauer ein.

## 5.6 Komponenten

Eine Komponente ist ein Teil einer Software die *explizite Schnittstellen* besitzt, von der Umgebung *unabhängig* ist und *kombinierbar* ist. Man versucht die Komponenten so zu entwerfen, dass man eine hohe Kohäsion und eine schwache Kopplung erreicht.

**Kohäsion**: Bezeichnet wie gut ein Programmkomponenten eine logische Einheit bildet.
**Kopplung**: Unter Kopplung versteht man, wie stark eine Programmkomponente von anderen abhängt.

# 6 Muster

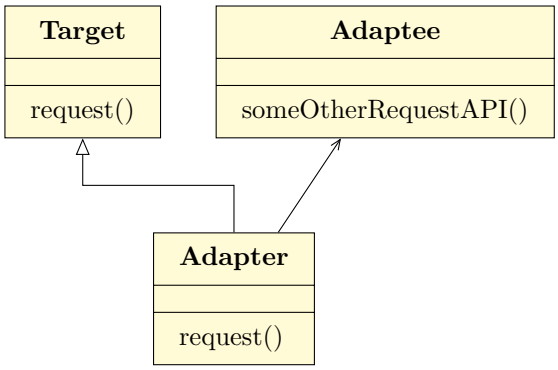
## 6.1 Entwurfsmuster

Erfahrung zeigt, dass sich manche Muster in der objektorientierten Softwareentwicklung oft wiederholen. Dazu haben die Apostel der Softwarearchitektur, die sogenannte "Gang of Four", die Bibel "Design Patterns" veröffentlicht.

### 6.1.1 Adapter/Wrapper

Ein vorgegebenes Objekt ("adaptee") wird auf eine gewünschte Schnittstelle ("target") angepasst. Man kann sich hier vorstellen, eine eigene Liste auf das `List`-Interface von Java anzupassen, indem der Adapter Methodenaufrufe des `List`-Interface auf die eigene Listen-API überträgt.

Man unterscheidet zwischen Objekt- und Klassenadaptern.

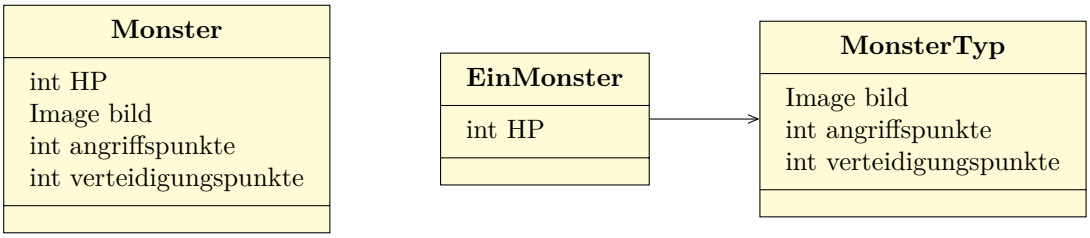



## 6.2 Analysemuster

### 6.2.1 Exemplar und Beschreibung

Auch genannt *Item-Item Description*. Bei vielen Objekten wiederholen sich systematisch immer wieder Werte.

Vergleiche hierzu das Beispiel der `Monster`-Klasse, wo in einem fiktiven Spiel jedes Monster, das auftaucht, Instanz dieser Klasse ist. Wir können die sich immer wieder wiederholenden Daten in der `MonsterTyp`-Klasse zusammenfassen, sodass die vielen `EinMonster`-Objekte klein und überschaubar bleiben.



### 6.2.2 Koordinator

Das Koordinator-Muster fasst Eigenschaften einer Beziehung zwischen zwei Klassen zusammen, die so wirklich nicht zu einer der beiden Klassen allein passen.

Man kann sich etwa die Beziehung "gebucht haben" zwischen den Klassen `Kunde` und `Veranstaltung` vorstellen. Das Buchungsdatum gehört weder zum Kunden noch zur Veranstaltung selber, also führt man den Koordinator `Buchung` hinzu.

### 6.2.3 Abstrakte Oberklasse

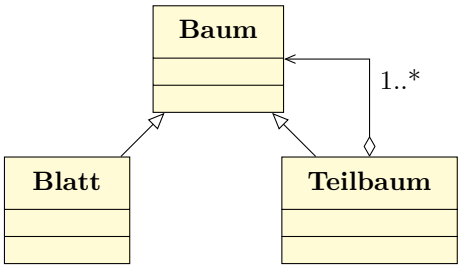
Genau das, was der Titel sagt. Gemeinsame Eigenschaften/Operationen werden in einer abstrakten Oberklasse zusammengefasst.

### 6.2.4 Wechselnde Rolle

Ein Objekt einer Unterklasse kann in eine andere Unterklasse wechseln. Man führt nun eine abstrakte Klasse zwischen den beiden Unterklassen und der Oberklasse ein. Etwa kann ein `Student` als Unterklasse von `Person` später auch ein `Mitarbeiter` werden.

### 6.2.5 Komposition

Angenommen, man kann kein Haskell[2] und muss einen Baum implementieren. Das Kompositionsmuster nutzt eine abstrakte Oberklassen und zwei Unterklassen, das *Blatt*, und das *Kompositum* die in unserem Fall etwa mehrere Bäume speichert.

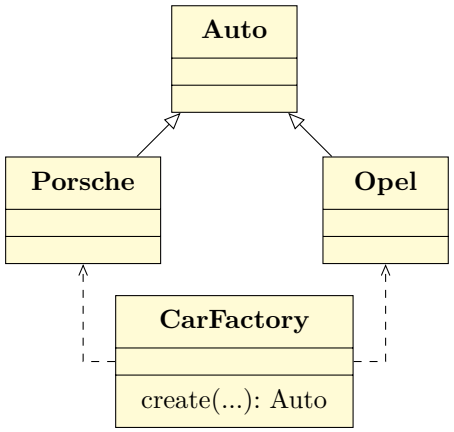



## 6.3 Entwurfsmuster 2

### 6.3.1 Factory Method

[2] `data Baum = Teilbaum [Baum] | Blatt`. Fertig.

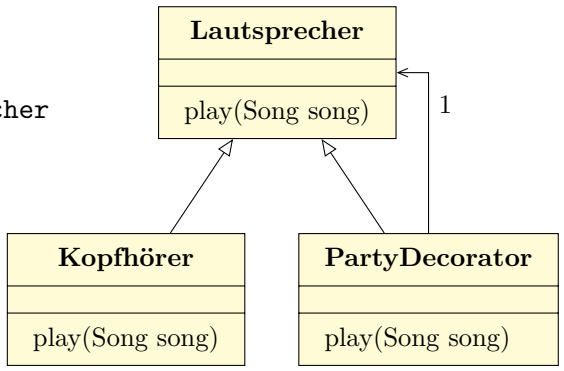
Bei der Objekterzeugung soll zwischen Varianten (d.h. Unterklassen) gewählt werden, ohne dass der Erzeuger sich darum kümmern muss. Wir schalten eine *Factory* dazwischen, die sich um das konkrete Erstellen von Objekten kümmert.




Jahrhunderte alte Tradition zwingt uns an dieser Stelle, die klassische Autometapher zu verwenden. Abhängig davon, ob man Student ist oder nicht, kann hier etwa die `CarFactory` entscheiden, welches Auto man bekommt. Der Nutzer dieser API muss sich dann nicht darum kümmern, wie genau das Auto zusammengebaut wird, denn am Ende des Tages will man doch eigentlich nur Auto fahren, richtig?

### 6.3.2 Decorator

Das *Decorator*-Muster erweitert bestehende Schnittstellen um weitere Funktionalität.




In diesem Beispiel betrachten wir `Lautsprecher` als abstrakte Komponente. Ein `Kopfhörer` spielt einen Song, ist also eine konkrete Implementation. Der `PartyDecorator` hingegen *erweitert* lediglich die bestehenden `Lautsprecher` so, dass etwa alle Tiefen verstärkt werden und nutzt dann die bestehende `play`-Methode.

Auf den ersten Blick erinnert das Klassendiagramm auf der rechten Seite doch sehr an das Muster der Komposition (6.2.5). Der Unterschied liegt aber darin, nicht einfach mehrere Objekte zu tragen, sondern bestehende Schnittstellen zu erweitern. Vergleiche hierzu auch die Kardinalität der Assoziationen.

### 6.3.3 Singleton

Es gibt Klassen, von denen es zu jedem Zeitpunkt nur eine Instanz geben sollte. Man stelle sich etwa eine Datenbankschnittstelle vor, von der es in einem Programm nur eine Instanz geben sollte. Code sagt mehr als tausend Worte, also hier eine beispielhafte Implementation.

```
public class Database {
  protected static Database instance;
  protected Database() {}
  public static Database getInstance() {
    if (instance == null) instance = new Database();
    return instance;
  }
}
```

`instance` hält die eindeutige Instanz der Datenbank. Der Konstruktor wird nicht-öffentlich gemacht. Eine "Factory-Method" `getInstance` gibt dann die einzige Instanz zurück (und erstellt sie, wenn sie noch nicht existiert).

Auch praktisch: "Override Static". Um sich die ganzen Aufrufe á la `Database.getInstance().doStuff()` zu verkürzen, können auch *statische* Hilfsfunktionen hinzugefügt werden, sodass man dann `Database.doStuff()` schreiben kann:

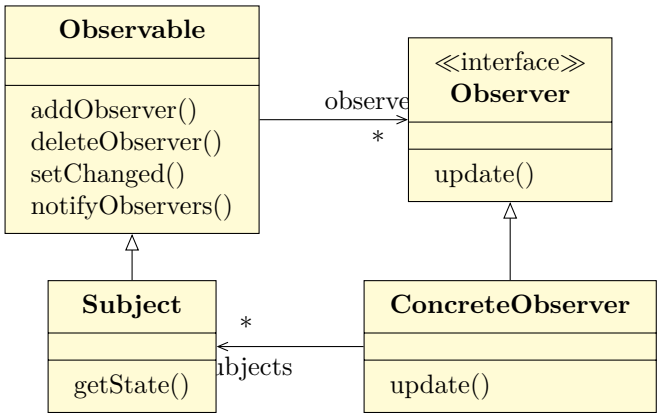
```
...
public static void doStuff() { getInstance().doStuff(); }
...
```

### 6.3.4 Observer

Wenn mehrere Objekte an Zustandsänderungen eines Objektes interessiert sind, bietet sich das *Observer*-Muster an.

Damit muss das `Subject` nichts über die `Observer` wissen.

Man kann auch `notifyObservers` so modellieren, dass auch nur die Art der Änderung übertragen wird.



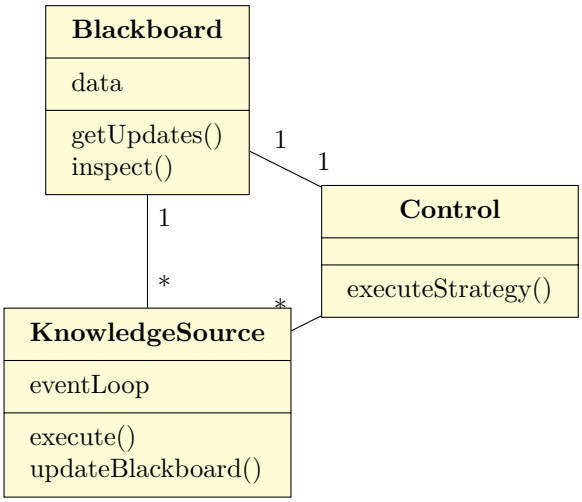

## 6.4 Architekturmuster für die Struktur

### 6.4.1 Schichten

Man erinnere sich an das TCP/IP-Modell.

### 6.4.2 Blackboard

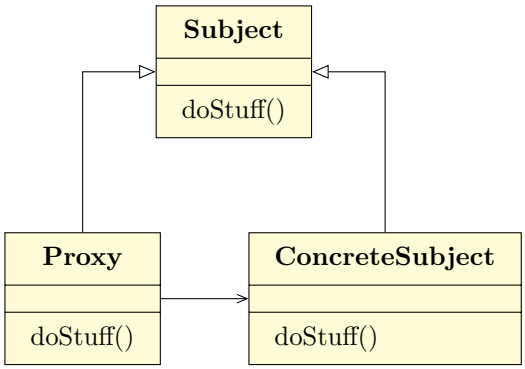
Das *Blackboard*-Muster führt eine *Control*-Komponente ein, die verschiedene Strategien auf Daten ausführt. Diese Daten werden im namensgebenden *Blackboard* gespeichert. Die ausgeführten Strategien werden *Knowledge Sources* genannt.




Man stelle sich einen Compiler vor, der mit verschiedenen Strategien versucht, den Code zu optimieren. Manche Ansätze funktionieren, manche nicht. Hier wären mögliche Knowledge Sources die Optimierungsstrategien. Die Blackboard-Daten wären die Datenstruktur gegeben, die das Programm darstellt.

### 6.4.3 Proxy

Aus irgendeinem Grund kann man nicht direkt auf ein Objekt zugreifen. Möglicherweise liegt das Objekt auf einem anderen Rechner, man will aber das Interface einfach halten und so tun, als wäre es lokal. Ein *Proxy*-Objekt delegiert Verwendungen des Originals durch die gleiche API wie die des Originals.




So kann der Client jetzt die `Subject`-API benutzen und es ist egal, woher genau das Objekt kommt. Es könnte lokal liegen, als `ConcreteSubject` oder auch über den `Proxy` am anderen Ende der Welt.

Kann auch für andere Zugriffshürden verwendet werden, wie Sicherheitsprobleme oder einen großen Aufwand.

### 6.4.4 Model-View-Controller

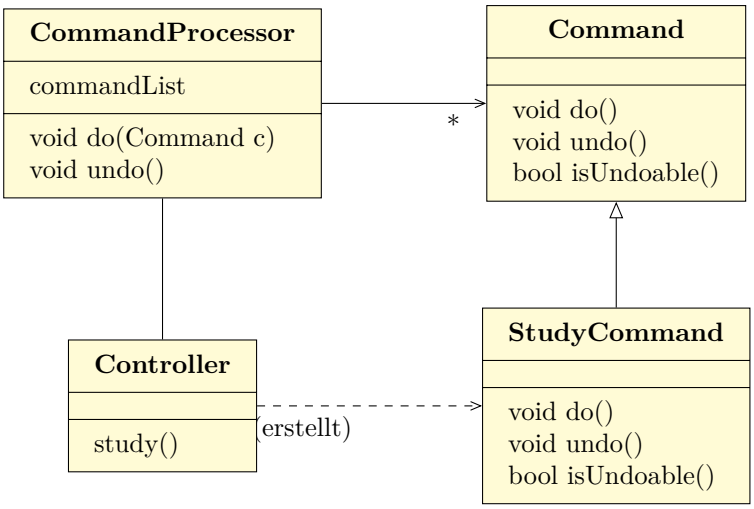
Man teilt ein Programm in drei Teilaspekte strukturell auf:

- *Model:* Kümmert sich nur um die reine Datenhaltung.
- *View:* Kümmert sich um die Sicht auf Daten, etwa über ein GUI.
- *Controller:* Kümmert sich um die Benutzerschnittstelle und Modifikation.

## 6.5 Architekturmuster für die Kommunikation

### 6.5.1 Command

Wir abstrahieren auszuführende Aktionen nicht als Code, sondern als Daten. Dafür erstellen wir *Commands*, also Objekte, die ihre Operation ausführen können (und ggf. sogar rückgäng machen können). `CommandProcessor` verwaltet `Command`-Objekte. Ein `Controller` stellt die Commands über normale Funktionen zur Verfügung.

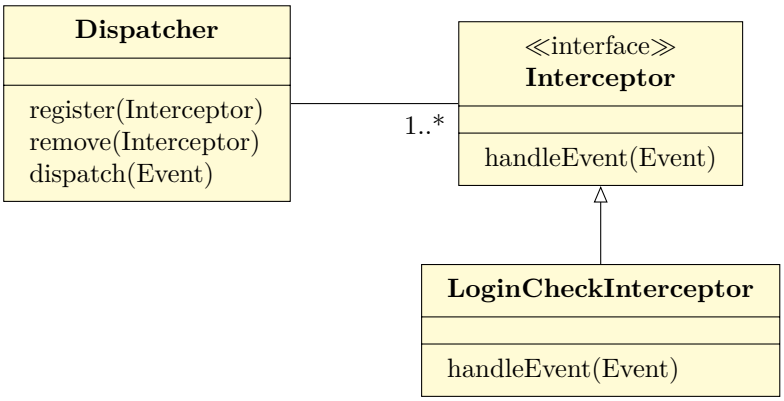



Der `CommandProcessor` speichert also alle bisher ausgeführten Commands und kann diese dann rückgängig machen (`undo`). Der Rest des Programmes kann nun im Controller `study()` aufrufen, welches einen `StudyCommand` erstellt und diesen mit dem `CommandProcessor` ausführt.

### 6.5.2 Interceptor

Angenommen, wir wollen einem Webserver modular Features hinzufügen, etwa um vor jedem Aufruf einer Website zu prüfen, ob der Nutzer angemeldet ist. Die einzelnen Funktionen, die Seiten erstellen, sollen aber sich darum nicht kümmern, ob und wie Zugangschecks gemacht werden.

Wir schalten also einen `Interceptor`-Interface dazwischen, der von einem `Dispatcher` aufgerufen wird, um etwa ein "Website-Aufruf-Objekt" zu bearbeiten. Der `Dispatcher` merkt sich alle `Interceptor`s und ruft diese nacheinander auf einem Ereignis auf.




Nun kann das Webserver-Framework einfach immer den Dispatcher nutzen und das System kann beliebig erweitert werden, indem zum Dispatcher neue Interceptors registriert werden.

# 7 Software- und Systementwurf

Von einer Anforderungsspezifikation (Pflichtenheft) und einer Funktionalen Spezifikation (Produktdefinition) wollen wir zu genauen Aussagen über eine Implementierung kommen.

Ein guter Entwurf erfüllt die Kriterien der *Korrektheit*, *Verständlichkeit und Präzision*, *Hohe Kohäsion*, *Schwacher Kopplung* zwischen Komponenten.

## 7.1 "4+1 Sichten"-Modell aus dem Rational Unified Process

Bei dem Systementwurf ist eine Gliederung der Betrachtungsaspekte sinnvoll. Nach Philippe Kruchten ist das "4+1 Sichten"-Modell:

- **Logische Sicht:** Klassenmodell, Verfeinerung des Analysemodells (Endanwender)
- **Struktursicht:** Subsysteme, Schnittstellen (Programmierung, Wartung)
- **Ablaufsicht:** Prozesse, Koordination (System-Integratoren)
- **Physikalische Sicht:** Komponenten, Hardwaresysteme, Netze (Kommunikation, Verteilung, Auslieferung, Installation)

Alle vier Sichten überschneiden sich mit den **Szenarien** (vgl. Use-Case-Diagramme 2.2).

## 7.2 Taktiken im Softwareentwurf

### 7.2.1 Verfügbarkeit

**Failure:** Von außen beobachtbarer Fehler eines Systems. Ggf. resultieren mehrere Faults in einem Failure.

**Fault:** Intern aufgetretener Fehler. Kann korrigiert werden oder zum Failure werden.

**Mean Time to Failure (MTF):** Durchschnittliche Zeit zwischen zwei Failures (ohne Down-Zeiten).

**Mean Time to Repair (MTR):** Durchschnittliche Zeit zur Reparatur eines Failures.

**Verfügbarkeit:** Wir rechnen $\frac{\text{MTF}}{\text{MTF+MTR}}$.

Für die Fehlererkennung (Fault) bieten sich verschiedene Diagnostiken an: *Exceptions*, regelmäßige *Pings/Echos* durch andere Komponenten oder regelmäßige Meldungen der zu prüfenden Komponente selbst (*Heartbeats*).

Aufgetretene Faults können verschieden behandelt werden:

- Abstimmen: Redundante Systeme wählen einen Ersatz
- Aktive Redundanz: Redundante Komponenten laufen parallel und bei Ausfall wird einfach die Arbeit übernommen.
- Passive Redundanz: Bei Ausfall wird eine passive Komponente zwecks Ersatz zu einer aktiven Komponente gemacht.
- Spare: Ein Ersatz*system* übernimmt die Arbeit.

# 8 Implementation

## 8.1 Auswahl der Programmiersprache

Wählt Haskell.

## 8.2 Codingstandards

## 8.3 Datenstrukturen

Eine Monade ist ein Monoid in der Kategorie der Endofunktoren.

### 8.3.1 WTFs pro Minute

Die wichtigste und einzige valide Metrik für guten Code ist "WTFs pro Minute". Leider noch keine SI-Einheit.

### 8.3.2 Google Guava

*Guava* von Google ist eine Java-Bibliothek, die viele oft vorkommende Schwierigkeiten in Java vereinfacht.

**Fail Fast:** Hilfsfunktionen übernehmen das Werfen von Exceptions in Fehlerfällen.
`checkNotNull(x, "x ist null!")` stoppt, wenn `x` `null` ist,
`checkArgument(param \% 2 == 0, "param ist nicht gerade!")` ist speziell für die Validierung von Parametern vorgesehen.

**Optionals:** Objekte, die `null` repräsentieren können, oder einen Wert enthalten können. Wird benutzt, weil man bei `null`s immer so viele `NullPointerException`s bekommt, während `Optional` nette Funktionen bietet, wie etwa
`myOptional.or("my default value").toString()`.

**Immutables:** Es gibt nicht veränderliche Implementation der Java-Typen `List`, `Set`, etc. `ImmutableList.of("a", "b", "c").add("x")` würde etwa crashen.

**Builders:** Um etwa nicht veränderliche Objekte einfacher zu erzeugen, gibt es sogenannte "Builder", denen man Einträge hinzufügen und dann aus diesem Builder etwa eine `ImmutableList` erstellen kann.

### 8.3.3 Vererbung oder Komposition?

In der Regel sollte man Komposition vor Vererbung wählen. Hier ist Komposition im einfachen Sinne "wir speichern dieses Objekt als Attribut" gemeint, *nicht* als Muster (6.2.5).

# 9 Generative Softwareentwicklung

**Generative Software Engineering:** Methode der effizienten Generierung von Software mit Modellen wie UML oder einer DSL, um Qualität zu erhöhen und Entwicklungszeit zu senken.

**Domain Specific Language (DSL):** Eine Sprache für eine spezielle Domäne/einen speziellen Anwendungsbereich. Es gibt eine graphische Sprache zur Gleisnetzbeschreibung, "NESTML", eine Sprache zur Beschreibung von neuronalen Netzen, "ProNet$^{\text{sim}}$" zur Modellierung und Simulation von Logistik.

## 9.1 MontiCore

"MontiCore" ist eine Sprache zur Beschreibung von "modularen Sprachfragmenten". Dazu gibt es auch viele Werkzeuge zur generativen Softwareentwicklung. MontiCore wird hier an der RWTH entwickelt. Es gibt verschiedene Komponenten:

**Freemarker:** Eine Template-Sprache, die Einbettung von Daten in Text erlaubt. Etwa einen `name` einfügen: `Mein Name ist ${name}`. Man kann auch komplexere Ausdrücke einfügen oder Template-Variablen zuweisen:

```
<#assign person = ${x.getPerson()}>
Mein Name ist ${person.getName()}
```

**Data Explorer (DEx):** Nimmt ein Datemodell und generiert ein Java-Programm mit GUI, Speicher- und Sharingfunktionalität.

**CD4A:** Eine Modellierungssprache, die ein Klassendiagramm in Textversion mit Java-ähnlicher Syntax ist. Dies ist das Eingabeformat für DEx.

```
classdiagram MyBlog {
  class Post {
    String title;
    String content;
    Post parentPost;
  }
  class MediaPost extends Post {
    String category;
    MediaType mediaType;
  }
  enum MediaType {
    DOCUMENT,
    PICTURE;
  }
}
```

`abstract` ist vor `class` möglich.
Es gibt außerdem Syntax für Assoziationen, die im `classdiagram`-Block einzufügen sind.

```
association [*] Blog -> Member[*];
```

Multiplizitäten optional; default: `*`.
Richtungen: `->`, `<-`, `<->`, `--`.
Zwei Formen: `assocation` und `composition`.

Der generierte Java-Code verhindert etwa `null`s durch Assoziationen-Verwaltung. Es werden automatisch für `class`-Statements im CD4A-Code vier verschiedene Java-Typen generiert: `interface Post`, `PostImpl` (erweitert `Post`), `PostBuilder` (vgl. 8.3.2) und ein `PostManager` für das Verwalten und Speichern von `Post`-Objekten. Getter und Setter werden auch generiert.

# 10 Werkzeuge

Hier gehen wir weniger ins Detail, denn vieles aus diesem Teil der Vorlesung ist eher wichtig für die Anwendung in der Praxis.

## 10.1 Versionierung

Softwareversionen können und sollten mit Versionsverwaltungssoftware verfolgt werden, sodass jede Änderung, die eine Person macht, protokolliert wird. Versionsverwaltung ermöglicht außerdem parallele Bearbeitung von Code, Rücknahme von einzelnen Änderungen und Datensicherung.

Wir haben in der Vorlesung "git" und "subversion" kennengelernt.

## 10.2 Build-Automatisierung

Das Kompilieren von Code sollte automatisiert werden.

"Make" ist das klassische Werkzeug von 1977. "Ant" und "Maven" sind neuer und insbesondere für Java gedacht.

## 10.3 Projekt- und Wissensmanagement

"Trac" ist eine Software, die gleichzeitig als Wiki, Bugreporting-Software, Repository-Browser und mehr dient.

Mit "Sonar" können qualitätsrelevante Projektkennzahlen wie Anzahl der Codezeilen oder Tiefe von Vererbungshierarchien überwacht werden.

## 10.4 Projektmanagement Plattformen

Auf `https://git.rwth-aachen.de/` findet Ihr nicht nur das Repository dieser Panikzettel, sondern könnt auch gleich eure eigenen Repositories verwalten.

# 11 Qualitätsmangement

## 11.1 Prozessintegrierte Qualitätssicherung

In jeden Übergang zwischen Entwicklungsstadien kann Qualitätssicherung eingebettet werden.

- Analyse → Entwurf: Reviews, Prototyp der Oberfläche
- Entwurf → Implementierung: Reviews, Konsistenzprüfungen, Realisierbarkeitsstudien
- Implementierung → Test: Code-Reviews, Test-Spezifikation, Integrationsplan, Test-Management
- Test → Wartung: Fehlerberichte, Änderungsmangement, Versionsplanung

## 11.2 Test und Integration

**Testobjekt:** Komponente, die zu testen ist.

**Testfall:** Satz von Eingaben und Ausgaben.

**Testdaten:** Daten, die für den Testfall benötigt werden.

**Testtreiber:** Werkzeug, dass das Testobjekt/den Test aufruft.

**Platzhalter:** Auch "stub"/"dummy". Platzhalter für ein nicht vorhandenes Unterprogramm.

**Regressionstest:** Erneuter Test, um zu prüfen, dass Änderungen keine neuen Fehler eingeführt haben.

## 11.3 Testverfahren

**Black box test:** Funktionaler Test nur auf Basis der Spezifikation. Man verwendet *Äquivalenzklassen*, Tests die das selbe Ergebnis erzeugen, und *Grenzwertttests*, Tests für Randfälle.

**White box test:** Strukturtest. Testfälle werden auf Basis des Codes ausgewählt.

**Kontrollflussorientierter Test.**

**Datenflussorientierter Test**.

**Überdeckungsgrad:** Für Tests sollte möglichst jede Zeile Code getestet werden. Verschiedene Metriken:

- Anweisungsüberdeckung: Anteil ausgeführter Anweisungen.
- Zweigüberdeckung: Anteil ausgeführter Anweisungen und Verzweigungen.
- Pfadüberdeckung: Anteil ausgeführter Programmpfade.

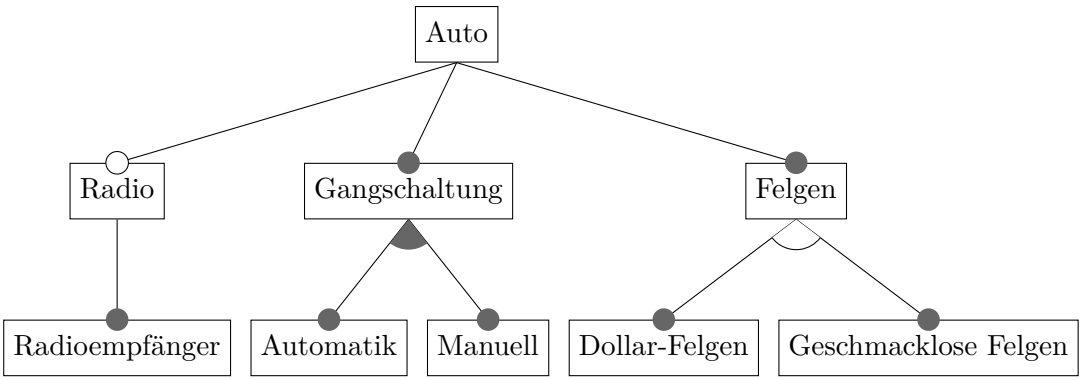
### 11.4 JUnit

JUnit ist ein Java Test Framework für automatische Tests.

Methoden und Klassen werden mit Markern markiert.

- `@Test` markiert eine Testmethode
- `@Before` wird vor jedem Test ausgeführt
- `@After` wird nach jedem Test ausgeführt
- `@BeforeClass` wird vor dem erste Test ausgeführt
- `@AfterClass` wird nach dem letzten Test ausgeführt

## 12 Featurediagramme




Legende:

- weißer Kreis = optional
- schwarzer Kreis = notwendig
- weißer Bogen = XOR (genau eins)
- schwarzer Bogen = OR (mindestens eins)

# 13 Softwareentwicklung

Im Vergleich zu anderen Ingenieursdisziplinen gibt es bei hier nur reine Entwicklung. Die Fertigung ist reines Kopieren, da Software ein immaterielles Gut ist. Dafür ist das Finden und Beheben von Fehlern viel aufwendiger. Kosten sind fast ausschließlich durch die Personalkosten bestimmt.

Gründe, warum Softwareentwicklung schief gehen kann:

- Mangelnde Einbeziehung der Benutzer
- Änderung und Unvollständigkeit von Anforderungen
- Mangelnde Managementunterstützung
- Fehlende Ressourcen
- Unklare Ziele und unrealistische Erwartungen
- Zeitmangel