# Design by Contract

# Design by Contract

Teile das zu entwickelnde Programm in kleine Einheiten (Klassen, Methoden), die unabhängig voneinander entwickelt und überprüft werden können.

Einheiten mit klar definierten Aufgaben und wohldefiniertem Verhalten, z.B.

- Klasse `List`
- Methode `List.add`

Setze größere Komponenten (letztendlich das Gesamtprogramm) aus kleineren zusammen.

Wenn sich alle Teilkomponenten korrekt verhalten, dann auch die zusammengesetzte Komponente.

# Entwicklung von Komponenten

Jede Komponente

- hat eine klar definierte Aufgabe.
- stellt ihre Dienste durch eine öffentliche Schnittstelle bereit.
- kann selbst andere Komponenten benutzen.

Komponenten sollen voneinander *unabhängig* sein:

- unabhängige Entwicklung und Überprüfung (z.B. von verschiedenen Teammitgliedern)
- Austauschbarkeit
- interne Änderungen in einer Komponente beeinflussen andere Komponenten nicht negativ

Wie kann man solche Komponenten entwickeln und sicherstellen, dass sich zusammengesetzte Komponenten wunschgemäß verhalten?

# Entwicklung von Komponenten

In einer objektorientierten Sprache spielen Klassen die Rolle von Komponenten.

Beim Implementieren einer Klasse sollte man sich genau überlegen

- welche *Leistungen* die einzelnen Funktionen der Klasse erbringen sollen.
- welche *Annahmen* die Funktionen über ihre Argumente und den Zustand der Klasse machen.

*Dokumentiere* Leistungen und Annahmen der Klasse.

Benutze nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.

Java-Typsystem oft zu schwach, um Annahmen zu garantieren (z.B. Wert von $x$ immer positiv).

# Beispiel

```
public class Konto {

   public double getKontostand()

   /**
    *  Wenn vorher getKontoStand() = x
    *  und betrag >= 0,
    *  dann danach getKontoStand() = x + betrag
    */
   public void einzahlen(double betrag)

   /**
    *  Wenn vorher getKontoStand() = x
    *  und x > betrag >= 0,
    *  dann danach getKontoStand() = x - betrag */
   public void abheben(double betrag)
}
```

# Entwicklungsprinzip

*Benutze immer nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.*

Austauschbarkeit Jede Klasse kann durch eine andere mit der gleichen öffentlichen Schnittstelle ersetzt werden, solange diese mindestens die gleichen Leistungen erbringt und nicht mehr Annahmen macht und

unabhängige Entwicklung Teammitglieder einigen sich über Leistungen und Annahmen und können diese dann unabhängig voneinander implementieren.

unabhängige Überprüfung systematisches Testen der dokumentierten Leistungen

# Design by Contract

Dieses Entwicklungsprinzip ist unter dem Namen Design by Contract (etwa: Entwurf gemäß Vertrag) bekannt.

Design by Contract enthält eine Reihe von methodologischen Prinzipien zur komponentenbasierten Softwareentwicklung.

Analogie mit Verträgen im Geschäftsleben:

- Die Komponenten schließen untereinander Verträge ab.
- Ein Vertrag beinhaltet das Versprechen einer Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind.
- Setze Programm so aus Komponenten zusammen, dass es richtig funktioniert, wenn nur alle ihre Verträge einhalten.

# Verträge

Verträge werden zwischen Klassen abgeschlossen und haben folgende Form:

Wenn vor dem Aufruf einer Methode in einer Klasse eine bestimmte Voraussetzung erfüllt ist, dann wird die Klasse durch die Ausführung der Methode in einen bestimmten Zustand versetzt.

## Beispiele

Für jedes Objekt `List l` gilt:

- Nach der Ausführung von `l.add(x)` gilt die Eigenschaft `l.isEmpty() = false`.
- Ist die Voraussetzung `l.size() > 0` erfüllt, so wird die Funktion `l.remove(0)` keine Exception werfen.

# Verträge

Ein Vertrag für eine Methode besteht aus:

Vorbedingung: Welche Annahmen werden an die Argumente der Methode und den Zustand der Klasse gemacht.

Nachbedingung: Welche Leistung erbringt die Methode, d.h. welche Eigenschaft gilt nach ihrer Ausführung.

Effektbeschreibung: Welche Nebeneffekte hat die Ausführung der Methode (Exceptions, Ein- und Ausgabe, usw.)

Für uns genügt es, Verträge informell zu beschreiben.

# Beispiel – Klasseninvarianten

Klasseninvariante Zu jedem Zeitpunkt erfüllt der Zustand der Klasse eine bestimmte Eigenschaft (die *Invariante*).

(Achtung: "zu jedem Zeitpunkt" fragwürdig, warum?)

Beispiel

- Klasse `Universitaet` mit privatem Feld

  ```
  private Map<Student, Set<Seminar>> seminareProStudent;
  ```

- Um herauszufinden, welche Studenten ein bestimmtes Seminar besuchen wird, muss man alle Studenten durchgehen.
- effizienter: Daten auch noch nach Seminaren indizieren.

  ```
  private Map<Seminar, Set<Student>> studentenInSeminar;
  ```

- Invariante: Die beiden Felder `seminareProStudent` und `studentenInSeminar` enthalten die gleichen Daten.

# Beispiel – Klasseninvarianten

Implementierung von Klasseninvarianten

- Konstruktoren haben Invariante als Nachbedingung.

  Im Beispiel: `seminareProStudent` und `studentenInSeminar` beide anfangs leer

- Alle Methoden haben Invariante sowohl als Vor- als auch als Nachbedingung: Wenn die Invariante vor der Ausführung gilt, dann auch danach.

  Im Beispiel: Alle Methoden können annehmen, dass `seminareProStudent` und `studentenInSeminar` die gleichen Daten enthalten, müssen aber auch sicherstellen, dass das nach ihrer Ausführung immer noch gilt.

# Überprüfung von Bedingungen mit `assert`

Bedingungen und Invarianten können mit der Java-Instruktion `assert` getestet werden.

Die Instruktion

```
assert test : errorValue;
```

prüft, ob der boolesche Ausdruck `test` zu `true` auswertet, und wirft eine `AssertException`, wenn das nicht der Fall ist.

äquivalent:

```
if (!test) {
  throw new AssertionError(errorValue)
}
```

Warum benutzt man nicht die `if`-Abfrage?

# Überprüfung von Bedingungen mit `assert`

`assert`-Instruktionen werden nur zum Testen benutzt.

- Standardmäßig werden Assertions ignoriert, d.h. sie haben keinen Einfluss auf die Progammgeschwindingkeit.
- Assertions müssen beim Programmstart ausdrücklich eingeschaltet werden.

  ```
  java -ea Main
  ```

  (`-ea` für „enable assertions“)

⇒ Das Programm sollte sich mit und ohne Assertions gleich verhalten.

`assert`-Instruktionen dienen auch der Dokumentation von Verträgen.

# Assertions – Beispiele

## Nachbedingung prüfen

```
class NichtUeberziehbaresKonto {
  private double kontoStand;
  ...
  public void abheben(double betrag) {
     ...
     assert (kontoStand >= 0) : "Interner Fehler " +
                                "Argument nicht richtig geprüft," +
                                "Konto überzogen";
  }
}
```

## Unerreichbare Programmteile

```
  ...
  for (...) {
    if (...) return;
  }
  assert false; // Diese Instuktion sollte unerreichbar sein
```

# Assertions – Beispiele

Argumente von öffentlichen Methoden nicht durch Assertions überprüfen.

öffentliche Methoden müssen Argumente immer selbst validieren

```
public void setX(int x) {
  if ((0 > x) || (x >= xmax)) {
    throw new IndexOutOfRangeException();
  }
  ...
}
```

Programm soll sich gleich verhalten, egal ob Assertions eingeschaltet sind oder nicht.

# Assertions – Beispiele

## Argumente von privaten Methoden

Private Methoden müssen falsche Argumente nicht immer mit Exceptions behandeln.

Da sie nur von der Klasse selbst benutzt werden können, kann man intern sicherstellen, dass sie nur mit guten Argumenten aufgerufen werden.

Dies kann man mit `assert` testen.

```
private void setX(int x) {
  assert (0 <= x) && (x < xmax);
  ...
}
```