Karlsruher Institut für Technologie

KASTEL / IPD
Prof. Dr. Sebastian Erdweg erdweg@kit.edu
Prof. Dr. Ralf H. Reussner reussner@kit.edu

# Programmierparadigmen – WS 2025/26

https://dsis.kastel.kit.edu/1181_1197.php

**Blatt 3: Design by Contract** Besprechung: 16.01.2026

## 1 Design by Contract – Warmup

Das folgende Software-Fragment beschreibt ein einfaches Einkaufsszenario, in dem Produkte (`Products`) aus einer Ladentheke (`Counter`) von Personen (`Persons`) in einen Einkaufswagen (`Cart`) gelegt werden können. Dazu dient die Methode `put(Product)`, die mit einem JML-Vertrag versehen ist.

1. Wo und durch wen (Aufrufer/Aufgerufener) wird der spezifizierte Vertrag verletzt?
2. Schreiben Sie eine `main`-Methode (oder eine Test-Methode), die eine neue Person erstellt und auf ihr einmalig `shop` aufruft. Der übergebene `Counter` soll immer dasselbe Produkt zurückliefern. Welche Ausgabe erwarten Sie, wenn Sie das Programm mehrfach mit JML-Laufzeitprüfungen ausführen? Geben Sie eine Begründung für die Ausgabe an.

```
public interface Product {}

public interface Counter {
   /**
    * Gibt ein Produkt aus der Theke zurück.
    * Gibt einen Nullpointer zurück, falls sie leer ist.
    */
   public Product takeSomeProduct();
}

public class Cart {
   private Set<Product> products = new HashSet<Product>();

   /*@
     @ requires product != null;
     @ ensures  getProducts().contains(product);
     @ ensures  getProducts().containsAll(\old(getProducts()));
     @ ensures  getProducts().size() == \old(getProducts().size()) + 1;
     @*/
   public void put(Product product) {
      products.add(product);
   }

   public /*@ pure @*/ Collection<Product> getProducts() {
      return Collections.unmodifiableSet(products);
```

```
    }
}

public class Person {
    public void shop(Counter counter) {
        Cart cart = new Cart();
        for (int i = 0; i < new Random().nextInt(20); i++) {
            cart.put(counter.takeSomeProduct());
        }
    }
}
```

## 2 Design by Contract – Personalverwaltung

Es soll eine neue Software zur Personalverwaltung eines Unternehmens entwickelt werden. Um die korrekte Funktionsfähigkeit der Software besser garantieren zu können, wird die Verwendung von Design by Contract in Erwägung gezogen. Dafür soll zunächst eine einfache Schnittstellen der Unternehmenssoftware umgesetzt werden, die eine Klasse zur Repräsentation von Mitarbeitern voraussetzt.

Die Methoden `fire()` und `hire()` der Schnittstellen `Company` sollen einen bestimmten Vertrag erfüllen. Neben selbstverständlichen Anforderungen, beispielsweise dass sich bei der Einstellung eines Mitarbeiters außer dessen Hinzufügen nichts am Personal ändert, gibt dazu es folgende Vorgaben:

- Ein Mitarbeiter kann nur bei einer Firma angestellt sein.
- Es darf nur versucht werden einen Mitarbeiter einzustellen, wenn dieser noch nicht in der Firma angestellt ist.
- Es darf nur versucht werden einen Mitarbeiter zu feuern, wenn dieser in der Firma angestellt ist.

Gehen Sie davon aus, dass die Methode `getEmployees()` keine interne Datenstruktur herausgibt, sondern eine Kopie davon.

**Assertion als Verträge:** In Java kann das Schlüsselwort **`assert`** verwendet um Verträge zu Simulieren und diese während der Laufzeit auszuwerten. Hierbei steht nach dem `assert`-Schlüsselwort ein bool'scher Ausdruck.

Syntax:

```
assert expression1;
assert expression1 : expression2;
```

Falls der Ausdruck nicht zu `true` ausgewertet wird, wird eine `Exception` geworfen. Durch Assertions vor und nach der Ausführungslogik kann somit Vor- und Nachbedingungen überprüft werden. Ein Beispiel und eine Diskussion zu Assertions finden Sie in den Design-By-Contract Vorlesungsfolien 33 und 34.

```
public class Employee {
   private boolean isEmployed;

   public Employee() {
      this.isEmployed = false;
   }

   protected void hire() {
      this.isEmployed = true;
   }

   protected void fire() {
      this.isEmployed = false;
   }

   public boolean isEmployed() {
      return isEmployed;
   }
}
```

```
public interface Company {
   public List<Employee> getEmployees();
   public void hire(Employee employee);
   public void fire(Employee employee);
}
```

1. Schreiben Sie JML-Verträge für die Methoden `fire()` und `hire()` der Klasse `Company`. Welche weitere Ergänzung am Programm müssen Sie vornehmen?

2. Entwickeln Sie eine konkrete Implementierung des Company-Interface in Java und stellen Sie den Vertrag durch die Nutzung von `assert` Ausdrücken sicher.

3. Mit Java `assert` Ausdrücken können Verträge für Methoden direkt in der Implementierung der Methode vorgenommen werden. Was sind die Vorteile dieses Ansatzes gegenüber Sprachen wie JML zur Beschreibung von Verträgen? Was sind die Nachteile?

## 3 Design by Contract - Liskov

In der mathematischen, wie auch in der Welt-Auffassung ist ein Quadrat ein Rechteck. In der Softwaretechnik kann diese Auffassung im Zusammenhang von Vererbung Probleme verursachen. Dieses Problem lässt sich durch die folgenden zwei Klassen beispielhaft beobachten.

```
public class Rectangle {
   protected int width = 0;
   protected int height = 0;

   /*@
     @ requires newWidth > 0;
     @ ensures getHeight() == \old(getHeight());
     @ ensures getWidth() == newWidth;
     @*/
   public void setWidth(int newWidth) {
      this.width = newWidth;
   }

   public /*@ pure @*/ int getWidth(){
      return this.width;
   }

   public /*@ pure @*/ int getHeight(){
      return this.height;
   }

   public int area(){
      return this.getWidth() * this.getHeight();
   }
}

public class Square extends Rectangle {
   /*@
     @ requires newWidth > 0;
     @ ensures getHeight() == newWidth;
     @ ensures getWidth() == newWidth;
     @*/
   @Override
   public void setWidth(int newWidth) {
      this.width = newWidth;
      this.height = newWidth;
   }
}
```

1. Betrachten Sie die Vor- und Nachbedingungen von `setWidth(int newWidth)` in den Klassen `Rectangle` und `Square`. Begründen Sie, warum die beiden Verträge bzgl. des Liskov'schen Substitutionsprinzips inkompatibel zueinander sind.

2. Wie können Sie die Implementierung verändern, sodass das Liskov'sche Substitutionsprinzip nicht mehr verletzt ist?

# 4 Fortgeschrittene Parallelisierungsprinzipien: Primzahlentest

Gegeben ist folgender Java Code, welcher sequenziell bis zu einer bestimmten Zahl einfache Primzahlentests durchführt und die Anzahl gefundener Primzahlen bestimmt:

```
public final class PrimeTester {
   public static boolean isPrime(int n) {
      if (n < 2) return false;
      for (int i = 2; i <= Math.sqrt(n); ++i) {
         if (n % i == 0) return false;
      }
      return true;
   }
}

public final class PrimeCounterSequential {
   public static int countPrimes(final int until) {
      int count = 0;
      for (int i = 2; i <= until; ++i) {
         if (PrimeTester.isPrime(i)) ++count;
      }
      return count;
   }

   public static void main(String[] args) {
      final int target = 100000000;

      final long startTime = System.currentTimeMillis();
      int count = countPrimes(target);
      final long endTime = System.currentTimeMillis();

      System.out.println("Duration␣for␣interval␣[2,␣" + target + "]␣is␣"
            + (endTime - startTime) + "␣ms\n" + count + "␣primes");
   }
}
```

Zur Reduktion der Laufzeit und Ausnutzung eines Mehrkernprozessors soll der Primzahlentest parallelisiert werden.

1. Erweitern Sie Ihr Programm zum Test der Primeigenschaft so, dass sich eine beliebige Anzahl Kerne die Arbeit teilen kann, sowie die obere Grenze des Intervalls, für welches die Anzahl der Primzahlen bestimmt werden soll (oben `target`) frei definiert werden kann. Benutzen Sie hierfür *keine* selbstverwalteten Threads, sondern beispielsweise `Executors` und `Futures`. Die Methode `isPrime` soll unverändert wiederverwendet werden.

   Entwickeln Sie Ihr Programm so, dass es als Argumente der `main`-Methode als erstes die obere Grenze des Intervalls bis zu dem die Anzahl Primzahlen bestimmt werden soll und als zweites die Anzahl verwendeter Threads erwartet werden. Es soll über die Standardausgabe in der ersten Zeile die benötigte Zeit, in der zweiten die Anzahl der gefunden Primzahlen (egal mit welcher Formatierung) ausgeben. Sie können sich dabei an obigem Code orientieren. Das Verhalten bei fehlerhaften Eingaben ist undefiniert.

2. Wie viele Primzahlen finden Sie mit ihrer parallelisierten Version im Intervall $[2, 10^8]$?

3. Wie groß ist der Speedup (im Vergleich zur sequentiellen Version) für 1, 2, 4 und 8 Threads? Wie viele Kerne hat Ihr Testrechner?

4. Wie ließe sich das Programm noch stärker beschleunigen als durch Parallelisierung?