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?

   Beispiellösung:

   - Der Aufruf der Methode durch `Person` verletzt den Vertrag, da `Person` nicht sicherstellt, dass das Argument nicht `null` ist. Wie im Interface `Counter` angegeben, kann die dort aufgerufene Methode zum Erhalten eines Gutes `null` zurückliefern.
   - `Cart` selbst verletzt den Vertrag, da die Methode die Nachbedingungen nicht erfüllt. Die ersten beiden Nachbedingungen, dass alle vorher enthaltenen und das hinzugefügte Produkt im Einkaufswagen enthalten sind, werden erfüllt. Da als Datenstruktur für die Produkte jedoch ein `Set` verwendet wird, kann es sein, dass das hinzugefügte Element bereits in der Datenstruktur enthalten war und damit die dritte Bedingung, dass sich die Anzahl der Produkte um eins erhöht, nicht erfüllt ist.

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.

   Beispiellösung:

   ```
   public static void main(String[] args) {
       new Person().shop(new Counter() {
           private Product product = new Product() {};
           @Override
           public Product takeSomeProduct() {
               return product;
           }
       });
   }
   ```

   Die Ausgabe des Programms besagt, dass die Nachbedingung bzgl. der Größenänderung von `products` nicht erfüllt ist. Die Ausgabe tritt zwischen 0 und 19 Mal auf, da aufgrund der Implementierung der `shop`-Methode maximal 20 gleiche Produkte hinzugefügt werden und somit bei bis zu 19 die Nachbedingung nicht erfüllt ist.

```
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?

   Beispiellösung:

   ```
   /*@
     @ requires employee != null;
     @ requires !getEmployees().contains(employee);
     @ requires !employee.isEmployed();
     @ ensures  employee.isEmployed();
     @ ensures  getEmployees().size() == \old(getEmployees().size())+1;
     @ ensures  getEmployees().containsAll(\old(getEmployees()));
     @ ensures  getEmployees().contains(employee);
     @*/
   public void hire(Employee employee);

   /*@
     @ requires employee != null;
     @ requires getEmployees().contains(employee);
     @ requires employee.isEmployed();
     @ ensures  !employee.isEmployed();
     @ ensures  getEmployees().size() == \old(getEmployees().size())-1;
     @ ensures  \old(getEmployees()).containsAll(getEmployees());
     @ ensures  !getEmployees().contains(employee);
     @*/
   public void fire(Employee employee);

   public /*@ pure @*/ List<Employee> getEmployees();
   ```

   Zusätzlich zur Spezifikation der Verträge für die Methoden `hire` und `fire` müssen die Methoden `Company.getEmployees()`, sowie `Employee.isEmployed()` als `pure` deklariert werden, da diese innerhalb der Verträge verwendet werden und daher seiteneffektfrei sein müssen.

   Es ist hier wichtig, dass die Methode `getEmployees()` eine Kopie der internen Datenstruktur zurückgibt und nicht die interne Datenstruktur selbst. In letzterem Fall würde nämlich bei der Auswertung von `\old(getEmployees())` die alte Referenz auf die Employees-Datenstruktur vor dem Methodenaufruf gespeichert, die jedoch während der Ausführung geändert wird. Wenn die Nachbedingungen ausgewertet werden, enthielte diese Referenz ebenfalls die neuen Daten.

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

   Beispiellösung:

   ```
   public class CompanyImpl implements Company {
   ```

```
    private List<Employee> employees;

    public CompanyImpl() {
        this.employees = new ArrayList<>();
    }

    public List<Employee> getEmployees() {
        return new ArrayList<Employee>(employees);
    }

    public void hire(Employee employee) {
        assert employee != null;
        assert !employees.contains(employee);
        assert !employee.isEmployed();
        List<Employee> oldEmployees = new ArrayList<>(employees);

        employee.hire();
        employees.add(employee);

        assert employee.isEmployed();
        assert employees.size() == oldEmployees.size() + 1;
        assert employees.containsAll(oldEmployees);
        assert employees.contains(employee);
    }

    public void fire(Employee employee) {
        assert employee != null;
        assert employees.contains(employee);
        assert employee.isEmployed();
        List<Employee> oldEmployees = new ArrayList<>(employees);

        employee.fire();
        employees.remove(employee);

        assert !employee.isEmployed();
        assert employees.size() == oldEmployees.size() - 1;
        assert oldEmployees.containsAll(employees);
        assert !employees.contains(employee);
    }
}
```

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?

   Beispiellösung: Der Vorteil bei der Verwendung von `assert` Ausdrücken ist deren garantierte Kompatibilität mit der Implementierung der Methode und die Möglichkeit die Verträge während der Ausführung des Programms zu überprüfen. Die Auswertung von `assert` Ausdrücken zur Laufzeit lässt sich über das Kommandozeilenargument "-ea" erreichen. Der Hürde zur Verwendung von `assert` Ausdrücken ist gering, wenn sowieso bereits in Java implementiert wird, da keine zusätzliche Sprache oder ein zusätzliches Tool verwendet werden muss.

   Nachteilig an der Verwendung von `assert` Ausdrücken ist, dass diese Teil der Implementierung sind und damit auch nur in einer Implementierung spezifiziert werden können. Es ist nicht möglich

einen Vertrag für eine Schnittstelle zu definieren, wie es beispielsweise mit JML möglich ist. Die Bedingungen eines Vertrages müssten also in jeder Implementierung einer Schnittstelle neu definiert werden. Damit ist insbesondere nicht sichergestellt, dass alle Implementierungen der gleichen Schnittstelle die gleichen Vertragsbedingungen prüfen.
Weiterhin lassen sich Verstöße gegen den Vertrag erst zur Laufzeit feststellen. Da Verträge auch nur bei gewissen Eingaben nicht erfüllt sein können, die in der praktischen Anwendung nie oder nur selten auftreten, kann es sein, dass eine nicht-vertragsgemäße Verwendung einer Schnittstelle nicht auffällt, da sie nie mit entsprechenden Eingaben verwendet wird. Tools für JML und andere Sprachen können den Code statisch dahingehend verifizieren, dass dieser die Bedingungen erfüllt, oder Testfälle generieren, die beispielsweise die Randfälle der Verträge automatisiert überprüfen. Insbesondere wird dort statisch geprüft, ob sich ein Aufrufer an den spezifizierten Vertrag hält. Laufzeitfehler sind damit ausgeschlossen.
Letztendlich hat die Verwendung von `assert` Ausdrücken außerdem den Nachteil, dass sie ebenfalls in Java spezifiziert sind und somit eher dieselben Denkfehler bei deren Spezifizierung gemacht werden, wie bei der Implementierung einer Methode selbst. Eine separate Sprache mit adäquaten Sprachkonstrukten kann das Fehlerpotential senken. Beispielweise muss in Java-Code der Zustand zu Beginn eines Methodenaufrufen explizit abgespeichert werden, um ihn in den Nachbedingungen adressieren zu können. Gerade bei komplexen Datenstrukturen kann es schwierig sein diesen Zustand zu speichern, da die Datenstruktur im Allgemeinen vollständig geklont werden muss.

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.

   Beispiellösung: Der Vertrag der Methode `setWidth(int newWidth)` in `Square` verletzt das Liskov'sche Substitutionsprinzip, weil die Nachbedingungen abgeschwächt werden. Explizit wird die erste Nachbedingung (`getHeight() == \old(getHeight())`) nicht mehr erfüllt,

womit die notwendige Bedingung $Postcondition_{Sub} \Rightarrow Postcondition_{Super}$ (siehe Folie 23) nicht mehr gilt.

Bei der Ausführung mit JML-Laufzeitprüfungen wird entsprechend der Fehler angezeigt, dass die erste Nachbedingung der `setWidth`-Methode in `Rectangle` durch den Aufruf der `setWidth`-Methode in `Square` verletzt wird.

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

   Beispiellösung: Das Problem kann beispielsweise dadurch gelöst werden, dass sowohl `Square` als auch `Rectangle` ein Interface `Shape` implementieren, welches die Methode `area` definiert. Somit müsste jede Klasse ihre eigenen `setter`- und `getter`-Methoden und die zugehörigen Verträge unabhängig voneinander definieren.

   Eine alternative Lösung ist die Bereitstellung einer Superklasse `AbstractRectangle`, was `width` und `height` verwaltet und die `getter`-Methoden dafür bereitstellt, sowie die Methode `area` auf Basis dieser `getter`-Methoden implementiert. Die `setter`-Methoden müssen dann in den konkreten Klassen `Square` und `Rectangle` implementiert und ihre Verträge definiert werden.

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.

Beispiellösung: Die Beispiellösung verwendet so viele `CompletableFutures` wie Threads verwendet werden sollen, welche via Callback solange eine neue zu prüfende Zahl anfordern bis keine Zahlen mehr zu prüfen sind.

Diese Lösung ist effizienter als beispielsweise die Aufteilung in feste Intervalle, da die Berechnungszeit pro Zahl mit ihrer Höhe im Durchschnitt steigt. Nichtsdestotrotz lassen sich selbstverständlich noch andere, auch effizientere Lösungen entwickeln. Diese und eine weitere Beispiellösung finden Sie auf der Webseite zur Übung.

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public final class PrimeCounterParallelSupplier {
   private static class State {
      private int currentNumber = 1;
      private int result = 0;
      private final int target;

      State(int target) {
         this.target = target;
      }

      synchronized int nextNumber() {
         return currentNumber < target ? ++currentNumber : -1;
      }

      synchronized void addToResult(Integer count) {
         this.result += count;
      }

      int getResult() {
         return result;
      }
   }

   public static int countPrimes(final int until,
                                 final int threads) {
      final State state = new State(until);
      Set<CompletableFuture<Void>> futures = new HashSet<>();
      for (int i = 0; i < threads; ++i) {
         futures.add(CompletableFuture.supplyAsync(() -> {
            int currentNumber = 0;
            int partialCount = 0;
            while (currentNumber != -1) {
               if (PrimeTester.isPrime(currentNumber))
                  ++partialCount;
               currentNumber = state.nextNumber();
            }
            return partialCount;
         }).thenAccept(state::addToResult));
      }
```

```
        for (CompletableFuture<Void> future : futures) {
            try {
                future.get();
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
                return -1;
            }
        }
        return state.getResult();
    }

    public static void main(String[] args) {
        final long startTime = System.currentTimeMillis();
        int until = Integer.valueOf(args[0]);
        int threads = Integer.valueOf(args[1]);
        int count = countPrimes(until, threads);
        final long endTime = System.currentTimeMillis();
        System.out.println(threads + " thread duration for interval
            [2, " + until + "] is " + (endTime - startTime)
                + " ms\n" + count + " primes");
    }

}
```

Es gibt bereits in dieser Parallelisierungslösung verschiedene Entwurfsalternativen, z.B.:

- Anstatt lokal zu zählen, kann der Zustand beim Finden einer Primzahl aktualisiert werden.
- Anstatt eines `CompletableFuture` könnte ein einfaches `Future` verwendet werden, da der `thenApply` Aufruf weggelassen werden kann.

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

   Beispiellösung: 5761455 primes

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

   Beispiellösung: Für 4 physische / 8 logische Kerne

   $T_{seq} = 221730ms$
   $T_1 = 225987ms$
   $T_2 = 118853ms$
   $T_4 = 62152ms$
   $T_8 = 50045ms$
   $S_{seq}(1) = \frac{T_{seq}}{T_1} = 0,98$
   $S_{seq}(2) = \frac{T_{seq}}{T_2} = 1,86$
   $S_{seq}(4) = \frac{T_{seq}}{T_4} = 3,57$
   $S_{seq}(8) = \frac{T_{seq}}{T_8} = 4,43$

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

   Beispiellösung: Im verwendeten Programm wird für jede Zahl individuell getestet, ob sie ein Primzahl ist. Die Verwendung eines passenderen Algorithmus würde das Programm stark beschleunigen. Hierzu bietet sich beispielsweise das Sieb des Eratosthenes an. Dieses lässt sich ebenfalls zusätzlich parallelisieren.