Karlsruhe Institute of Technology

Programmierparadigmen
Übung 12: Design By Contract

M.Sc. Larissa Schmid

08.02.2024

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

Überblick heutige Übung

- Warmup (Übungsblatt Aufgabe 1)
- Personalverwaltung (Übungsblatt Aufgabe 2)
- Liskov (Übungsblatt Aufgabe 3)
- Java Happens-Before (Klausur WS22/23)

Warmup
(Übungsblatt Aufgabe 1)

```
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());
      }
   }
}
```

- Wo und durch wen wird der spezifizierte Vertrag verletzt?

Warmup
(Übungsblatt Aufgabe 1)

```
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());
      }
   }
}
```

- Aufrufende: Person stellt nicht sicher, dass Argument nicht null ist
- Aufgerufene: Anzahl der Produkte nicht zwingend um eins erhöht

Personalverwaltung
(Übungsblatt Aufgabe 2)

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

- JML-Verträge für *fire()* und *hire()*?
- Weitere Ergänzung am Programm?

```
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;
    }
}
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
/*@
  @ 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);
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
/*@
  @ 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);

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

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
/*@
  @ 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);

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

/*@
  @ 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);
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
/*@
  @ 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);

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

```
/*@
  @ 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);
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

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

- Konkrete Implementierung des Company-Interfaces?
- Vertrag durch assert-Ausdrücke sicherstellen

```
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;
    }
}
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
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);
   }
```

Personalverwaltung
(Übungsblatt Aufgabe 2)

```
    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);
    }
}
```

Personalverwaltung – Vor-/Nachteile assert
(Übungsblatt Aufgabe 2)

Vorteil	Nachteil
Verträge während Laufzeit überprüfen möglich	Überprüfen erst zur Laufzeit möglich
Geringe Hürde, da kein zusätzliches Tooling benötigt	Teil der Implementierung – kein Spezifizieren gegen Schnittstellen möglich
	Ebenfalls in Java - eher dieselben Denkfehler wie bei Implementierung

Liskov
(Übungsblatt Aufgabe 3)

```
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();
   }
}
```

- Warum sind die beiden Methoden *setWidth(int newWidth)* inkompatibel zueinander bzgl. des Liskov‘schen Substitutionsprinzips?

```
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;
   }
}
```

Liskov
(Übungsblatt Aufgabe 3)

```
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();
   }
}
```

- Square schwächt Nachbedingungen ab!

```
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;
   }
}
```

Liskov
(Übungsblatt Aufgabe 3)

```
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();
   }
}
```

- Square schwächt Nachbedingungen ab!

→ *Wie Implementierung verändern?*

```
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;
   }
}
```

Liskov
(Übungsblatt Aufgabe 3)

```
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();
   }
}
```

- Square schwächt Nachbedingungen ab!

→ *Wie Implementierung verändern?*

- z.B. gemeinsames Interface *Shape* → eigene *getter* und *setter*

```
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;
   }
}
```

Java Happens-Before
(Klausur WS22/23)

```java
public class CarDealership {
    private static CarSale carSale = new CarSale();

    public static void main(String[] args){
        Car car = new Car();
        carSale.carSold(car);
        carSale.carReturned(car);
        System.out.println("Sold Cars: " + carSale.getNumberOfSoldCars());
    }
}

public class CarSale {
    public int numberOfSoldCars = 0;
    public List<Car> soldCars = new ArrayList<Car>();

    public void carSold(Car car) {
        new Thread(()-> { numberOfSoldCars += 1; }).start();
        new Thread(()-> { soldCars.add(car); }).start();
        new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
    }

    public void carReturned(Car car) {
        new Thread(()-> { numberOfSoldCars -= 1; }).start();
    }

    public int getNumberOfSoldCars() {
        return numberOfSoldCars;
    }
}

public class Car { }
```

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- *carSold* gibt nicht immer die erwartete Anzahl an *soldCars* aus, selbst wenn die Threads in der Reihenfolge von oben nach unten ausgeführt werden.
- Wodurch entsteht dieser Effekt? Verwenden Sie den Begriff der Happens-Before-Beziehung.

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- *carSold* gibt nicht immer die erwartete Anzahl an *soldCars* aus, selbst wenn die Threads in der Reihenfolge von oben nach unten ausgeführt werden.
- Wodurch entsteht dieser Effekt? Verwenden Sie den Begriff der Happens-Before-Beziehung.
- **Es liegt keine Happens-Before-Beziehung vor zwischen den Threads.**
- **Das ermöglicht memory inconsistencies.**

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Was müssen Sie am Programm ändern, damit die Ausgabe wie erwartet funktioniert?
- Verändern Sie nicht das Starten jeder Anweisung als eigener Thread.

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Was müssen Sie am Programm ändern, damit die Ausgabe wie erwartet funktioniert?
- Verändern Sie nicht das Starten jeder Anweisung als eigener Thread.
- **Auf den beiden Threads die join-Methode aufrufen, bevor die Ausgabe gemacht wird → Etablieren einer Happens-Before-Beziehung.**

Java Happens-Before
(Klausur WS22/23)

```
12 public class CarSale {
13     public int numberOfSoldCars = 0;
14     public List<Car> soldCars = new ArrayList<Car>();
15
16     public void carSold(Car car) {
17         new Thread(()-> { numberOfSoldCars += 1; }).start();
18         new Thread(()-> { soldCars.add(car); }).start();
19         new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20     }
21
22     public void carReturned(Car car) {
23         new Thread(()-> { numberOfSoldCars -= 1; }).start();
24     }
```

- Ist es sicher, die Methoden carSold und carReturned in zwei Threads gleichzeitig aufzurufen? Wenn nicht, wie könnte dies sichergestellt werden?

Java Happens-Before
(Klausur WS22/23)

```
12 public class CarSale {
13     public int numberOfSoldCars = 0;
14     public List<Car> soldCars = new ArrayList<Car>();
15
16     public void carSold(Car car) {
17         new Thread(()-> { numberOfSoldCars += 1; }).start();
18         new Thread(()-> { soldCars.add(car); }).start();
19         new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20     }
21
22     public void carReturned(Car car) {
23         new Thread(()-> { numberOfSoldCars -= 1; }).start();
24     }
```

- Ist es sicher, die Methoden carSold und carReturned in zwei Threads gleichzeitig aufzurufen? Wenn nicht, wie könnte dies sichergestellt werden?
- **Nein, es kommt zu einer Race Condition, da beide Methoden schreibend auf numberOfSoldCars zugreifen.**
- **Sicherstellen z.B. durch Verwenden von AtomicInteger**

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Handelt es sich um Daten- oder Taskparallelismus?

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Handelt es sich um Daten- oder Taskparallelismus?
- **Taskparallelismus – verschiedene Aufgaben, die parallel ausgeführt werden.**

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Ist es aus Performance-Perspektive sinnvoll, jede Anweisung als eigenen Thread zu starten?

Java Happens-Before
(Klausur WS22/23)

```
16      public void carSold(Car car) {
17          new Thread(()-> { numberOfSoldCars += 1; }).start();
18          new Thread(()-> { soldCars.add(car); }).start();
19          new Thread(()-> { System.out.println(numberOfSoldCars); }).start();
20      }
21
```

- Ist es aus Performance-Perspektive sinnvoll, jede Anweisung als eigenen Thread zu starten?
- **Nein – zu viel Overhead durch Starten der Threads, Koordinieren etc.**