Java Design By Contract

Contents

Design by Contract

Hauptidee

The supplier guarantees the postcondition if the precondition is fulfilled (by the client)

Aufrufer == Customer

Aufrufende == Supplier

Contracts for Methods

- In software, contracts can be used to specify the semantics of methods
  - Precondition: Specification what the supplier expects from the client
  - Statements: The method body
  - Postcondition: Specification of what the client can expect from the supplier *if the precondition is fulfilled*
- Contracts do not only specify the behavior of a method but also how they have to be used by a client
  - Client has to ensure that the precondition is fulfilled
  - Client has to correctly handle the guaranteed result
- Design by contract idea from Bertrand Meyer [Meyer1992, Meyer1997]
  - Contractual use of methods
  - The supplier guarantees the postcondition if the precondition is fulfilled (by the client)
  - Implemented in the *Eiffel* programming language

Non-Redundancy

The body of a routine shall not test for the routine's precondition.

Beispiel

Quellcode

```
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;
  }
}
```

Mit asserts

1. Alle Vorbedingungen (Null, Zustand, alte Liste speichern)
2. Aktion
3. Alle Nachbedingungen

```
@Override
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 (employees.contains(employee));
    assert (employee.isEmployed());
    assert (employees.containsAll(oldEmployees));
```

```
    assert (employees.size() == oldEmployees.size() + 1);

}

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

Liskov Substitution Principle

Class A extends Class B and overwrites the method foo()

A.foo() erfüllt LSP genau dann, wenn:

1. **Preconditions** von A.foo() sind **nicht strickter** als bei B.foo()
2. **Postconditions** von A.foo() sind **nicht schwächer** als bei B.foo()

Liskov Substitution Principle

- The Liskov substitution principle restricts the possible pre- and postconditions of an overwriting method
  - Preconditions must not be more restrictive than those of the overwritten method: $\texttt{Precondition}_{\texttt{Super}} \Rightarrow \texttt{Precondition}_{\texttt{Sub}}$
  - Postconditions must be at least as restrictive as those of the overwritten methods: $\texttt{Postcondition}_{\texttt{Sub}} \Rightarrow \texttt{Postcondition}_{\texttt{Super}}$
- Regarding a complete class, the following rules apply:
  - Pre- and postcondition relations must hold for all methods as stated above
  - The class invariants must be at least as restrictive as those of the superclass: $\texttt{Invariants}_{\texttt{Sub}} \Rightarrow \texttt{Invariants}_{\texttt{Super}}$
- A special case of this is known from parameter and return types of methods (Co-/Contravariance)

Folien

Preconditions and Postconditions for `Stack`

- Preconditions:
  - `push` may not be called if the stack is full
  - `pop` / `top` may not be called if the stack is empty
- Postconditions:
  - After calling `push`, the stack may not be empty, the top element is the one that was pushed and its number of elements was increased by one
  - After calling `pop`, the stack may not be full and its number of elements has been decreased by one
  - After calling `top`, nothing has to be changed
- Are these pre- and postconditions complete?

```
public void push(Object element) {
   size++;
   for (int i = 0; i < size; i++) {
      this.elements[i] = element;
   }
}
```

This code would also fulfill the specified postcondition

Obligations and Benefits

- Method `push` example:

	Obligations	Benefits
Client	*Satisfy precondition:* Only call `push(x)` on a non-full stack	*From postcondition:* Get stack updated: not empty, x on top, size increased by one
Supplier	*Satisfy postcondition:* Update stack representation to have x on top, `size` increased by one, not empty	*From precondition:* Simpler processing thanks to the assumption that stack is not full

adapted from [Meyer1997]

- With the precondition that the stack must not be full when calling `push`, the `Stack` class must not deal with that case
  - Alternatively, the precondition could be relaxed, but then the behavior of the stack in that case would have to be specified

Aufgaben

WS16/17

Gegeben sei folgendes Code-Fragment mit einer (unvollständigen) Implementierung einer Collection-Klasse und einer Verwendung dieser Klasse. Für die Methode `add` der Collection-Klasse ist im Javadoc zusätzlich ein Vertrag angegeben, der Vor- und Nachbedingungen der Methode definiert. Die Vor- und Nachbedingungen sind als Java-Ausdrücke definiert. Zusätzlich wird mittels `\old(<expression>)` der Wert von `<expression>` vor dem Aufruf der Methode referenziert.
Anmerkung: Die vierte Nachbedingung ist syntaktisch inkorrekt. In Zeile 20 müsste es korrekterweise `elements[i] == null` heißen. Dies sollte bei der Lösung der Aufgaben nicht beachtet werden.

```
1  public class ObjectCollection {
2      Object[] elements;
3      int elementCount;
4
5      public ObjectCollection(int collectionSize) {
6          elements = new Object[collectionSize];
7          elementCount = 0;
8      }
9
10     /**
11     * Vorbedingungen:
12     *   object != null;
13     *
14     * Nachbedingungen:
15     *   elementCount == \old(elementCount) + 1;
16     *   elements[elementCount-1] == object;
17     *   for (int i = 0; i < elementCount - 1; i++)
18     *       elements[i] == \old(elements[i]);
19     *   for (int i = elementCount; i < elements.length; i++)
20     *       elements[i] == 0;
21     */
22     public void add(Object object) {
23         if (elementCount < elements.length) {
24             elements[elementCount++] = object;
25         }
26     }
27 }
28
29
30 public class ObjectCollectionUser {
31     public static void main(String[] args) {
32         Object object = new Object();
33         ObjectCollection collection = new ObjectCollection(10);
34         collection.add(object);
35     }
36 }
```

(a) Wird der spezifizierte Vertrag vom Aufrufer erfüllt? Begründen Sie Ihre Antwort. [1,5 Punkte]

Beispiellösung:

Der Vertrag wird vom Aufrufer erfüllt, da die Methode mit einem gerade erzeugten Objekt aufgerufen wird, welches somit nicht `null` ist und damit die einzige Vorbedingung erfüllt ist.

(b) Geben Sie an, welche Nachbedingungen des Vertrages vom Aufgerufenen nicht erfüllt werden. Begründen Sie dies, indem Sie eine Situation beschreiben, in der die Nachbedingungen nicht erfüllt werden. [2 Punkte]

Beispiellösung:

Die erste und zweite Zeile der Nachbedingung (Zeile 15 und 16) werden nicht erfüllt.

Falls das `elements`-Array voll ist, wird bei einem Aufruf der `add`-Methode nichts am Zustand der Collection verändert. Insbesondere bleibt `elementCount` gleich (Widerspruch zur ersten Bedingung) und das Methodenargument wird nicht in den `(elementCount-1)`-ten Eintrag von `elements` geschrieben (Widerspruch zur zweiten Bedingung).
Die anderen beiden Bedingungen gelten weiterhin, da sie eigentlich keine Nachbedingungen, sondern Invarianten der Collection darstellen.

(c) Der Vertrag soll so verändert werden, dass er vom Aufgerufenen erfüllt wird. [2,5 Punkte]

i. Geben Sie eine Änderung der Vorbedingungen an, mit der dies erreicht werden kann.

Beispiellösung:
Hinzufügen der Vorbedingung `elementCount < elements.length`.

ii. Geben Sie eine Änderung der Nachbedingungen an, mit der dies erreicht werden kann.

Beispiellösung:
Die erste und zweite Nachbedingung einschränken, sodass sie nur erfüllt sein müssen, falls `\old(elementCount) < elements.length` gilt. Nicht notwendig für die Beantwortung der Aufgabe, aber sinnvoll wäre es zu fordern, dass in allen anderen Fällen `\old(elementCount) == elementCount` gelten muss.

(d) Nehmen Sie an, es würde die folgende Prüfung des Eingabeparameters am Beginn der Method `add` ergänzt: [1 Punkt]

```
if (object == null) return;
```

Würde diese Ergänzung den Vertrag verletzen? Begründen Sie Ihre Antwort kurz.

Beispiellösung:

Nein, die Prüfung würde den Vertrag nicht verletzen. Die Vorbedingung schließt ein Objekt mit Wert `null` als Methodenargument bereits aus. Somit wird die Bedingung bei vertragsgemäßer Verwendung der Methode sowieso immer zu `false` ausgewertet.

SS17

Set kann keine Duplicate haben!

```
1  public interface Option {}
2
3  public class Selection {
4      private Set<Option> options = new HashSet<>();
5
6      /**
7       * Vorbedingungen:
8       *   1. option != null;
9       *
10      * Nachbedingungen:
11      *   1. options.size() == \old(options).size() + 1;
12      *   2. options.contains(option);
13      */
14     public void select(Option option) {
15         this.options.add(option);
16     }
17
18     /**
19      * Vorbedingungen:
20      *   1. option != null;
21      */
22     public void deselect(Option option) {
23         this.options.remove(option);
24     }
25 }
```

(a) Erfüllt die Methode `select` die im Javadoc spezifizierten Nachbedingungen? [2,5 Punkte]
Begründen Sie Ihre Antwort.

Beispiellösung:

Die Nachbedingungen werden nicht erfüllt. Ist beispielsweise die übergebene `Option` bereits vorher selektiert worden, so ändert sich die Anzahl der Elemente in `options` nicht, da dieses ein `Set` ist. Somit ist in diesem Fall die erste Nachbedingung nicht erfüllt.

(b) Geben Sie Nachbedingungen für die Methode `deselect` an, die das folgende, [3,5 Punkte]
informell spezifizierte Verhalten garantieren:
Die Methode `deselect` entfernt die übergebene `Option` aus den `options`,
falls sie darin vorhanden ist, und hat ansonsten keine weiteren Effekte.
Hinweis: Das Interface `Set<E>` stellt unter anderem folgende Methoden bereit:

- `containsAll(Collection<E> elements): boolean`
- `contains(E element): boolean`
- `size(): int`

Beispiellösung:

Eine Möglichkeit ist zu garantieren, dass alle nachher selektierten Optionen auch vorher selektiert waren und dass falls die übergebene `Option` vorher selektiert war die Größe von `options` um eins reduziert wird und sich ansonsten nicht ändern.

```
1.  if (\old(options).contains(option) {
        options.size() = \old(options).size() - 1;
    } else {
        options.size() = \old(options).size();
    }
2.  \old(options).containsAll(options);
```