Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

15. März 2024

Übungen zur Vorlesung Software Engineering I
Sommersemester 2024

Übungsblatt Nr. 1
(Musterlösung)

Aufgabe 1 (Wiederholung Java, Erstes Muster, Blackbox-Testing, 25 Punkte):

Aufgabe 1.1)

Erkennbar gibt es Code, der beiden Implementierungen gemeinsam ist: Einerseits die Wertebereichsprüfung mitsamt der Fehlermeldung, andererseits die Tatsache, dass beide Implementierungen einen Typ speichern müssen.

Der naheliegendste Ansatz im Rahmen des objektorientierten Entwurfs besteht darin, eine abstrakte Oberklasse `AbstractNumberTransformer` zu definieren, die das Interface `NumberTransformer` implementiert (Antwort auf **Frage 1**), selbst die Wertebereichsprüfung vornimmt und die eigentliche Transformation (sofern zulässig) an eine abstrakte Methode delegiert, die von den konkreten Unterklassen `GermanFormatNumberTransformer` und `RomanNumberTransformer` implementiert wird. Diese Oberklasse kann im Konstruktor den Typ bei Instanziierung von der jeweiligen Unterklasse übernehmen und ihn einem Attribut zuweisen. Die Typrückgabenmethode des Interface wird direkt in dieser Klasse implementiert.

```
package org.hbrs.se1.ss24.uebung1.businesslogic;

public abstract class AbstractNumberTransformer implements
NumberTransformer {

    protected final String transformerType;

    protected AbstractNumberTransformer(String transformerType) {
        this.transformerType = transformerType;
    }

    @Override
    public String transformNumber(int number) {
        if (number < 1 || number > 3000) {
            return "Fehler: Zahl außerhalb des zugelassenen
Wertebereichs!";
        } else {
            return transformCheckedNumber(number);
        }
    }

    @Override
```

```
    public String getTransformerType() {
        return transformerType;
    }

    protected abstract String transformCheckedNumber(int number);
}
```

Die Klasse `GermanFormatNumberTransformer` lässt sich mit Hilfe der Java-Framework-Klasse `DecimalFormat` recht einfach implementieren:

```
package org.hbrs.se1.ss24.uebung1.businesslogic;

import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Locale;

public class GermanFormatNumberTransformer extends
AbstractNumberTransformer {

    public static final String TRANSFORMER_TYPE = "Transformer für deutsche
Zahlen-Formatierungen";
    private final NumberFormat numberFormat =
DecimalFormat.getInstance(Locale.GERMANY);
    public GermanFormatNumberTransformer() {
        super(TRANSFORMER_TYPE);
    }

    @Override
    protected String transformCheckedNumber(int number) {
        return numberFormat.format(number);
    }
}
```

Die Implementierung der Klasse `RomanNumberTransformer` ist vollkommen analog, nur dass dem Konstruktor von `AbstractNumberTransformer` eine andere Stringkonstante übergeben wird und `transformCheckedNumber()` der in der Übung bereits vorgegebenen Implementierung folgt.

Aufgabe 1.2)
Die Objekterzeugung kann mit Hilfe einer Factory-Klasse umgesetzt werden, die im gleichen Package liegt wie die Klassen, von denen sie Instanzen erzeugt (**Frage 2**):

```
package org.hbrs.se1.ss24.uebung1.businesslogic;

public class TransformerFactory {
    public static NumberTransformer createGermanFormatNumberTransformer() {
        return new GermanFormatNumberTransformer();
    }

    public static NumberTransformer createRomanNumberTransformer() {
        return new RomanNumberTransformer();
    }
}
```

Beim hier verwendeten Entwurfsmuster handelt es sich um das Factory-Method-Pattern (**Frage 3**). Dabei dient eine separate Klasse, die *Factory*, dazu, Instanzen einer Klasse zu erzeugen und auch zu liefern. Hierzu stellt die Factory statische Methoden für verschiedene Rückgabetypen bereit (z.B. eine für römische Zahlen und eine für die deutsche Zahlenformatierung).

Der Vorteil liegt in einer zentralen und konsistenten Erzeugung von Objekten, da der Rückgabetyp nur einmalig festgelegt wird.

Die Factory wird im gleichen Package untergebracht wie die Transformer-Implementierungen, um unnötige Imports und Abhängigkeiten zu vermeiden.

Die Verwendung der `TransformerFactory` erfolgt dann in der Klasse `Client` wie folgt:

```
package org.hbrs.se1.ss24.uebung1.client;

import org.hbrs.se1.ss24.uebung1.businesslogic.NumberTransformer;
import org.hbrs.se1.ss24.uebung1.businesslogic.TransformerFactory;

public class Client {
    public void printTransformation(int number) {
        NumberTransformer numberTransformer =
TransformerFactory.createRomanNumberTransformer();
        String result = numberTransformer.transformNumber(number);
        System.out.println("Die römische Schreibweise der Zahl " + number +
" ist: " + result);
    }
}
```

Aufgabe 1.3)
Ein Black-Box-Test dient dazu, die Funktionalität von Software anhand von repräsentativ gewählten Testdaten zu testen, ohne dabei Einblick in die innere Struktur dieser Software zu haben. Zu jedem getesteten Repräsentanten eines Eingabedatums wird die dazugehörige Ausgabe überprüft.

Ein Positivtest ist erfolgreich, wenn eine ordnungsgemäße Eingabe richtig verarbeitet wurde, ein Negativtest dient dazu, das Abfangen fehlerhafter Eingaben zu überprüfen. Deshalb ist ein Negativtest dann erfolgreich, wenn eine fehlerhafte Eingabe erfolgreich abgefangen und mit einer Fehlermeldung beantwortet wurde.

Im vorliegenden Fall lassen sich drei Äquivalenzklassen bilden: Einmal für den Wertebereich von 1 bis 3000 (zulässige Eingabe für Positivtest), dann noch für den Wertbereich <0 (Negativtest) und den Wertebereich >3000 (ebenfalls Negativtest). Der Fall 0 wird in einem vierten separaten Test betrachtet.

Als Repräsentanten dieser drei Klassen kann man z.B. die Zahlen 1900, -10 und 3001 wählen. Im Black-Box-Test wird davon ausgegangen, dass sich das Programm für alle Eingaben innerhalb einer Äquivalenzklasse korrekt verhält, wenn es sich für einen Repräsentanten dieser Äquivalenzklasse korrekt verhält (**Frage 5**) – eine Annahme, *die allerdings nicht immer zutreffen muss*. Im vorliegenden Fall könnte es etwa sein, dass bestimmte römische Zahlen korrekt transformiert werden und andere nicht – dies ließe sich im Rahmen eines Black-Box-Tests aber nicht aufdecken, ohne wirklich *jede einzelne Zahl im zulässigen Wertebereich* zu testen!

Eine geeignete Testklasse mit diesen Repräsentanten sieht aus wie folgt:

```
import org.hbrs.se1.ss24.uebung1.businesslogic.NumberTransformer;
import org.hbrs.se1.ss24.uebung1.businesslogic.TransformerFactory;
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class TestRomanNumberTransformer {

    private NumberTransformer transformer = null;
    @BeforeEach
    void setUp() {
        this.transformer =
TransformerFactory.createRomanNumberTransformer();
    }

    @Test
    void TestRomanNumbers() {
        assertEquals("MCM", transformer.transformNumber(1900));
        assertEquals("Fehler: Zahl außerhalb des zugelassenen
Wertebereichs!", transformer.transformNumber(-10));
        assertEquals("Fehler: Zahl außerhalb des zugelassenen
Wertebereichs!", transformer.transformNumber(3001));
        assertEquals("Fehler: Zahl außerhalb des zugelassenen
Wertebereichs!", transformer.transformNumber(0));
    }
}
```

Die Tests sollten in einer separaten Test-Klasse untergebracht werden, um keine Vermischung mit dem eigentlichen Quellcode zu bewirken (**Frage 4**). Der Quellcode ohne die Tests kann dann separat ausgeliefert oder auch von einem separaten Team entwickelt werden, das nur die Schnittstellen der zu testenden Software erhält.

Die Klasse `Client` lässt sich nicht ohne Weiteres mit dieser Methodik testen, da ihre Methode `printTransformation()` auf die Standardausgabe schreibt und gegenüber der Testmethode nur den Typ `void` zurückliefert (**Frage 6**). Das berechnete Ergebnis ist also bereits vom Stack entfernt und nicht als Rückgabewert gespeichert worden, wenn `printTransformation()` verlassen wird.

Die ausgefüllte Excel-Vorlage für die Äquivalenzklassen sieht aus wie folgt:

Simple Test Suite	
Testobjekt:	RomanNumberTransformer (Übung Nr. 1)

		Parameter	
TestCase Nr.	**Kategorie** (pos; neg)	**Input** (number)	**Output** (Erwartetes Ergebnis)
1	pos	1900	MCM
2	neg	-10	"Fehler: .."
3	neg	3001	"Fehler: .."
4	neg	0	"Fehler: .."
5			
...			
n			

Zugehörige Äquivalenzklassen:

Parameter	**Äquivalenzklasse**	**Repräsentant**	**Kategorie** (pos; neg)
Input	[1..3000]	1900	pos
	[<0]	-10	neg
	[>3000]	3001	neg

Test auf 0 erfolgt separat!

Aufgabe 2 (Modellierung mit UML-Klassendiagrammen, 5 Punkte):

Musterlösung für das Klassendiagramm: