Software Engineering I

Übung 11: Verifikation und Integration von Software

Mariia Merzliakova, Thicha Melissa Thephasdin na Ayuthya

Aufgabe 11.1

1. Die Integration besteht aus zwei übergeordneten Anteilen: der Planung und der Umsetzung. Wichtig ist, dass alle Module, die in einem Integrationsschritt genutzt werden, in einer geprüften Form vorliegen müssen. Sie müssen somit unter anderem ihren Schnittstellenspezifikationen genügen.

   Zu der Planung gehört folgendes:
   - Festlegung der Integrationsstrategie
   - Festlegung er Build- und Integrationsumgebung

   Zu der Umsetzung gehören folgende Anteile:
   - Erstellung von Integrationsskripten
   - Durchführung von Modulintegrationen
   - Durchführung von Systemintegrationen
   - Erstellung und Auslieferung von Release

   Im Anschluss erfolgt die Verifikation, also die Prüfung des integrierten Systems. In diesem Schritt werden Fehler identifiziert. Eine Diagnose und Korrektur der Fehler erfolgt vor einer erneuten Iteration der Integration.

   Die Schritte werden in der Regel mehrfach wiederholt durchlaufen, da sie idealerweise verschränkt mit der Durchführung von Codierung und Test ablaufen sollen. Das Ziel ist die möglichst frühzeitige Erkennung von Fehlern und im Einklang damit eine schnelle Rückkopplung und Fehlerbehebung.

2. Die Hauptaufgabe der Integration ist das Zusammenfassen von Komponenten wie Modulen zu komplexeren Komponenten und schlussendlich zu einem Gesamtsystem. Das Zusammenfassen soll dabei auf der im Entwurf entwickelten Strukturierung des Systems basieren. Um die optimale Integration zu erhalten, müssen verschiedene Qualitätssicherungen eingehalten werden. Bei ungenügender Qualitätssicherung der Teilsysteme oder der Softwarearchitektur können Fehler auftreten. Diese können sich in ungünstigen Fällen negativ auf den Integrationsprozess auswirken oder diesen sogar behindern.

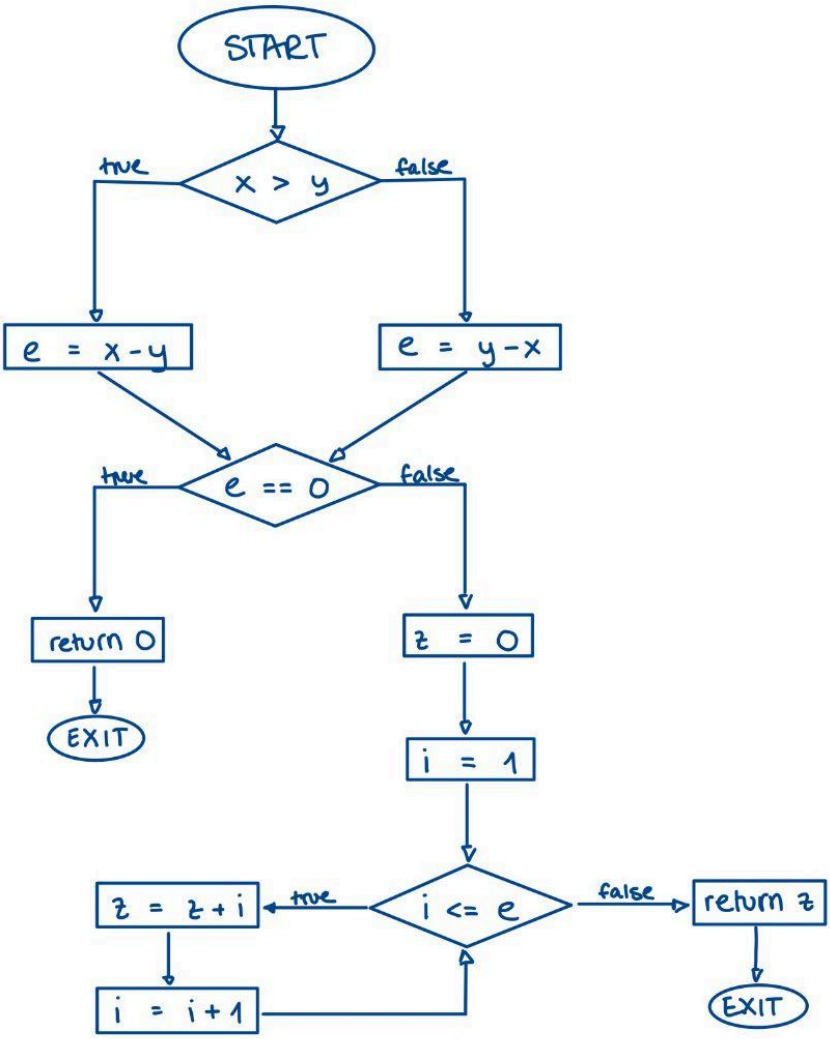
   Ein enges und fortlaufendes Zusammenspiel von Integration und Softwarearchitektur sind daher unumgänglich.

Aufgabe 11.2

1. Falls die Differenz zwischen x und y ungleich 0 ist, berechnet die Funktion für e = Differenz folgende Summe: $z = \sum_{i=1}^{e} i$ und gibt den Wert z aus.

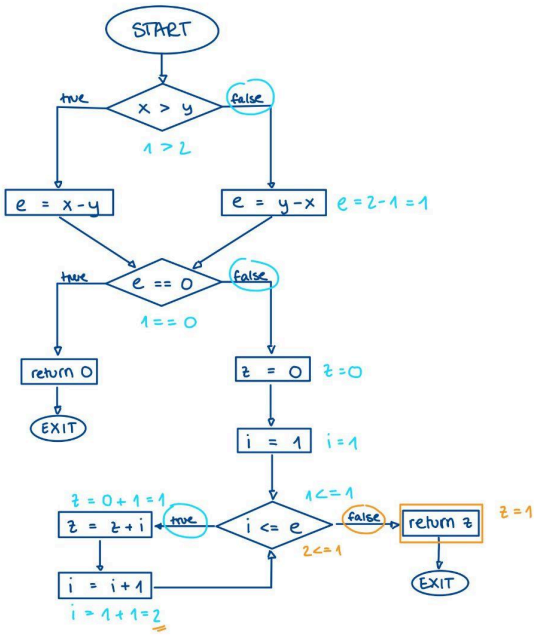
   Für den Fall, dass die Differenz Null beträgt, gibt die Funktion den Wert 0 aus.

2.

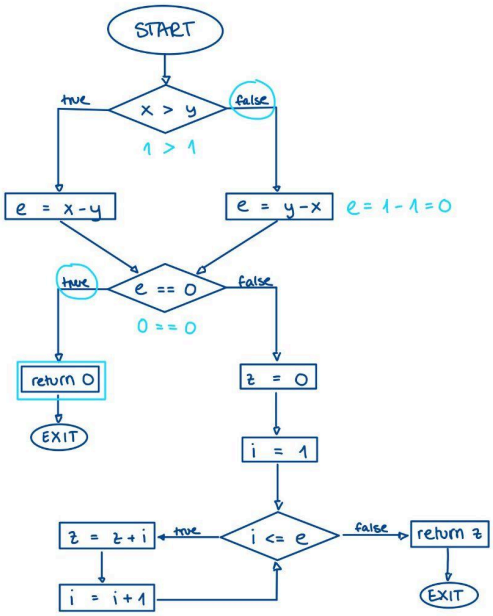


3. 3 Testfälle:
   - x < y
   - x = y
   - x > y

   Beispiele:
   Fall 1) x = 1, y = 2
   Zeilen: 1, 2, 3, 5, 6, 8, 9, 10, 11, 12, 10, 13, 14

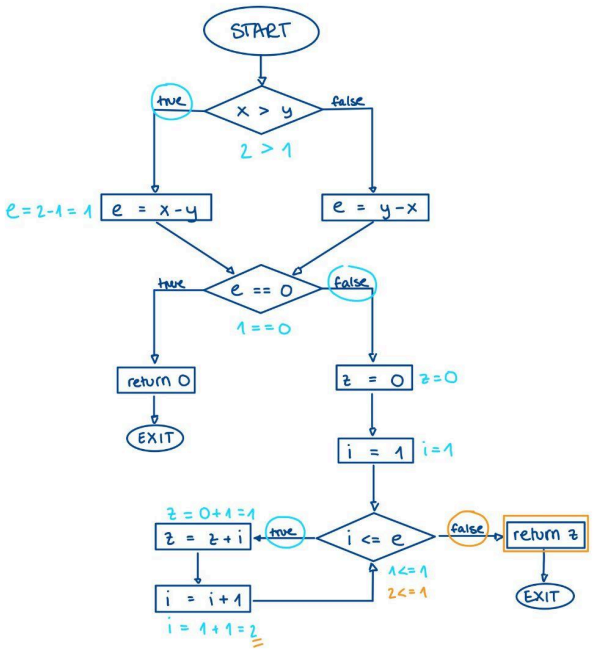


   Fall 2) x = 1, y = 1
   Zeilen: 1, 2, 3, 5, 6, 7, 14



Fall 3) x = 2, y = 1

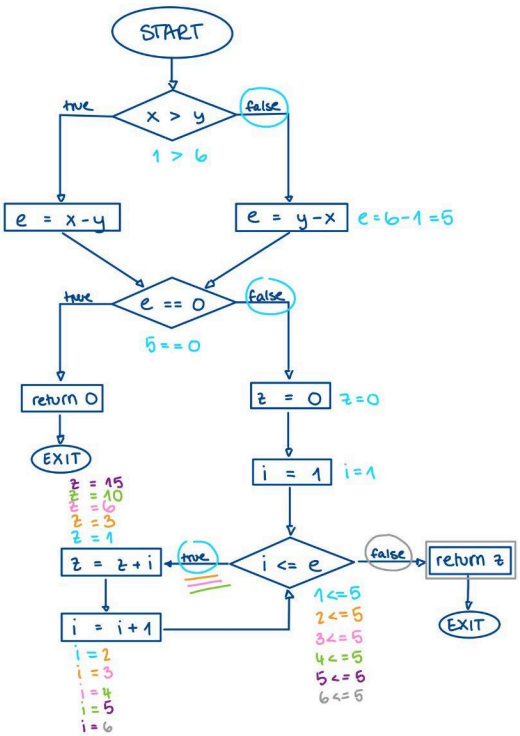
Zeilen: 1, 2, 3, 4, 6, 8, 9, 10, 11, 12, 10, 13, 14



Um die Korrektheit der Annahme aus 1.) zu beweisen haben wir zusätzlich 2 Fälle mit größeren Differenzen hinzugefügt. Diese sind für die 3) nicht zwingend notwendig.

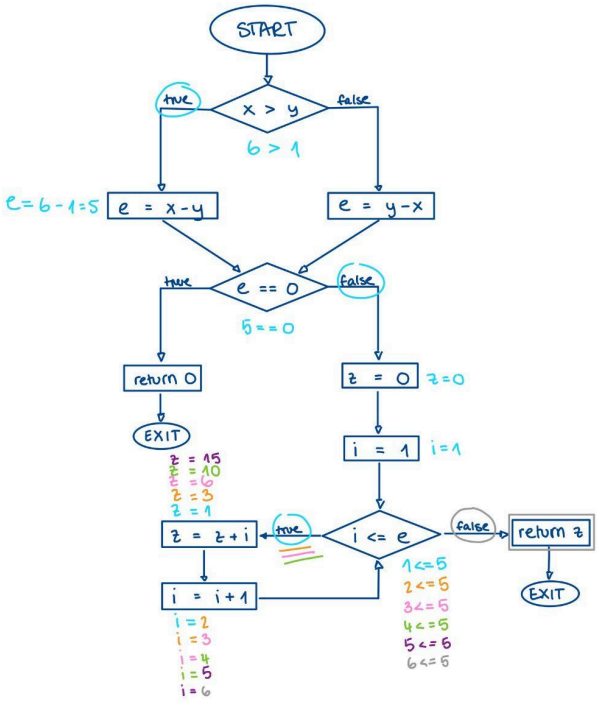
Fall 4) x = 1, y = 6

Zeilen: 1, 2, 3, 5, 6, 8, 9, 10, 5x (11, 12, 10), 13, 14



Fall 5) x = 6, y = 1

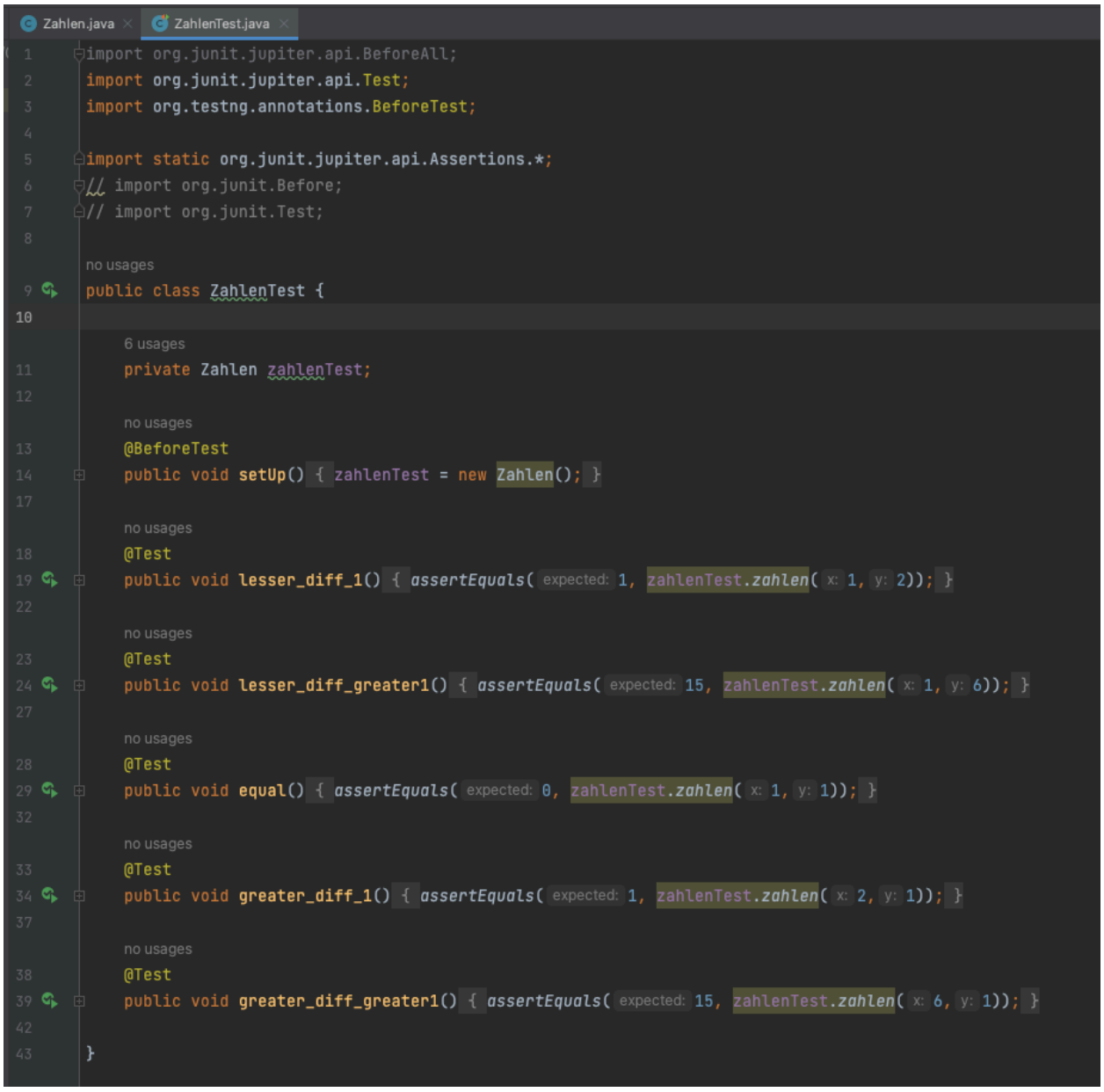
Zeilen: 1, 2, 3, 4, 6, 8, 9, 10, 5x (11, 12, 10), 13, 14



4.

Zahlen.java | ZahlenTest.java

```
2 usages
public class Zahlen {
    5 usages
    public static int zahlen(int x, int y) {
        int e;
        if(x > y) {
            e = x-y;
        } else
            e = y - x;
        if (e == 0)
            return 0;
        int z = 0;
        int i = 1;
        while(i <= e) {
            z = z + i;
            i++;
        }
        return z;
    }
}
```



Zahlen.java | ZahlenTest.java

```
1  import org.junit.jupiter.api.BeforeAll;
2  import org.junit.jupiter.api.Test;
3  import org.testng.annotations.BeforeTest;
4
5  import static org.junit.jupiter.api.Assertions.*;
6  // import org.junit.Before;
7  // import org.junit.Test;
8
   no usages
9  public class ZahlenTest {
10
       6 usages
11     private Zahlen zahlenTest;
12
       no usages
13     @BeforeTest
14     public void setUp() { zahlenTest = new Zahlen(); }
17
       no usages
18     @Test
19     public void lesser_diff_1() { assertEquals( expected: 1, zahlenTest.zahlen( x: 1, y: 2)); }
22
       no usages
23     @Test
24     public void lesser_diff_greater1() { assertEquals( expected: 15, zahlenTest.zahlen( x: 1, y: 6)); }
27
       no usages
28     @Test
29     public void equal() { assertEquals( expected: 0, zahlenTest.zahlen( x: 1, y: 1)); }
32
       no usages
33     @Test
34     public void greater_diff_1() { assertEquals( expected: 1, zahlenTest.zahlen( x: 2, y: 1)); }
37
       no usages
38     @Test
39     public void greater_diff_greater1() { assertEquals( expected: 15, zahlenTest.zahlen( x: 6, y: 1)); }
42
43 }
```

Aufgabe 11.3

1. Fünf Äquivalenzklassen:

Eq1: [a-z, A-Z, 0-9] - gültiger Bereich, aufsteigend und alphabetisch sortiert
Eq2: not [a-z, A-Z, 0-9] - gültiger Bereich, wird ignoriert
Eq3: Leere String - gültiger Bereich, leere String
Eq4: [A-Z] - gültiger Bereich, wird zu Kleinbuchstaben
Eq5: Doubles in [a-b, A-Z, 0-9] - gültiger Bereich, nur einmal aufgeführt

2.

Testfallnummer	Eingabe	Äquivalenzklasse	Ausgabe (Soll-Ergebnis)
TF1	“HelloWorld2024”	Eq1	“024dehlow”
TF2	““He!!oW@rld2024”	Eq2	“024dehlow”
TF3	“"	Eq3	“"
TF4	“A"	Eq4	“a”
TF5	“ooAA00"	Eq5	“0ao"