DREIECK TESTS

Martin Wölfer

Zusammenfassung

Es wird eine Klasse Dreieck.java mit dem Plugin ECLEmma genaustens getestet

Inhalt

EclEmma Plugin installieren

1. Help => Eclipse Marketplace
2. Nach EclEmma suchen
3. Installieren
4. Man kann nun bei einer Testklasse mit einem neuen Button die coverage checken bei der CUT (Class under Test)

```
    public boolean istDreieck() {
        if (this.seite_a == 0
                || this.seite_b == 0
                || this.seite_c == 0) {
            return false;
        }
        if (this.seite_a < 0
                || this.seite_b < 0
                || this.seite_c < 0) {
            return false;
        }
        if ((this.seite_a + this.seite_b == this.seite_c)
                || (this.seite_a + this.seite_c == this.seite_b)
                || (this.seite_c + this.seite_b == this.seite_a)) {
            return false;
        }
        if (this.seite_a + this.seite_b < 0
                || this.seite_b + this.seite_c < 0
                || this.seite_a + this.seite_c < 0) {
            return false;
        }
        if ((this.seite_a + this.seite_b < this.seite_c)
                || (this.seite_a + this.seite_c < this.seite_b)
                || (this.seite_c + this.seite_b < this.seite_a)) {
            return false;
        }

        return true;
    }
```

   i. Grün => Alle möglichkeiten gecovered
   ii. Gelb => Ein paar Möglichkeiten gecovered
   iii. Rot => Garkeine Möglichkeiten gecovered

Projekt erstellen

Zuerst wird in Eclipse ein neues Projekt erstellt. In diesem Projekt werden zunächst 2 Source ordner angelegt:

- Src => Ordner wo Dreieck.java liegt
- Test => Ordner wo die Testklassen liegen

Dann wurden die nötigen Klassen erstellt / hinzugefügt sowie die Junit Library (Rechtsklick => Java build path => add library => JUnit)

Git

1. Lokales repository anlegen mit *git init*
2. Remote repository hinzufügen mit *git remote add origin* *https://github.com/mwoelfer-tgm/SEW_15-16*
3. Synchronisieren mit *git pull origin master*
4. Files hinzufügen mit *git add IstDreieckTest.java*
5. Commiten mit *git commit -m „Add class for testing“*
6. Jedesmal wenn erneuerung hinzugefügt Schritte 4 und 5 wiederholen
7. Mit *git push* alles auf den externen Server „speichern“

Testklasse hinzufügen: TestIstDreieck

1. Man schaut in der CUT nach was in der Methode istDreieck steht und stellt dann TestMethoden dafür auf
2. Beispiel:

```
public boolean istDreieck() {
    if (this.seite_a == 0
            || this.seite_b == 0
            || this.seite_c == 0) {
        return false;
    }
```

   ⇨ Es gibt 6 sogenannte „branches“:
   a. Seite A = 0
   b. Seite A!= 0
   c. Seite B = 0
   d. Seite B!= 0
   e. Seite C = 0
   f. Seite C!= 0

```
@Test
public void testIstDreickSeiteANull(){
    dreieck.setSeite_a(0);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteBNull(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(0);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteCNull(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(0);

    assertFalse(dreieck.istDreieck());
}
```

Mit der ersten Methode wurden gleich 3 Branches gededeckt:

⇨ Seite A = 0
⇨ Seite B!= 0
⇨ Seite C!= 0

Mit der nächsten Methode wurden dann zwei weiter Branches gedeckt:

⇨ Seite A!= 0
⇨ Seite B = 0

Mit der letzten Methode wurde schließlich der letzte branch gedeckt:

- Seite C = 0

istDreieck()

testIstDreieckSeiteNull

Es wurde jede Seite mindestens einmal auf 0 und nicht 0 gesetzt, damit wurden die ersten Überprüfungen der Methode istDreieck gedeckt

```
@Test
public void testIstDreickSeiteANull(){
    dreieck.setSeite_a(0);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteBNull(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(0);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteCNull(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(0);

    assertFalse(dreieck.istDreieck());
}
```

testIstDreickSeiteMinus

Diesmal wurde jede Seite mind. einmal größer 0 und kleiner 0 gesetzt.

```
@Test
public void testIstDreickSeiteAMinus(){
    dreieck.setSeite_a(-1);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteBMinus(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(-1);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstDreickSeiteCMinus(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(-1);

    assertFalse(dreieck.istDreieck());
}
```

testIstDreieckSeiteAPlusBGleichC

Mit diesen Methoden wurde der nächste Block in der Dreieck Klasse überprüft. Nur mit der ersten Methode wurden schon 3 von 6 branches gedeckt:

- A + B = C
- A + C!= B
- C + B!= A

Mit der nächsten wurden folgende Branches gedeckt

- A + C = B
- A + B!=C

Mit der letzten Methode wurde der letzte branch gedeckt

- C + B = A

```
@Test
public void testIstAPlusBGleichC(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(3);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstAPlusCGleichB(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(3);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstCPlusBGleichA(){
    dreieck.setSeite_a(3);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(1);

    assertFalse(dreieck.istDreieck());
}
```

testIstDreieckSeiteAPlusBMinus

Mit diesen Methoden musste man erzwingen dass
A + B < 0 usw. sein soll. Nun konnte man nicht einfach eines von beiden auf eine minuszahl setzen weil es schon bei der Bedingung davor „gecatched“ werden würde. Also musste man mit Overflow arbeiten, weil 1 + Integer.MAX_VALUE = INTEGER.MIN_VALUE, und diese Zahl ist minus

Das Prinzip zum decken aller branches ist das gleiche, man muss jede Bedingung einmal auf true und false haben.

```
@Test
public void testIstAPlusBMinus(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(Integer.MAX_VALUE);
    dreieck.setSeite_c(1);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstBPlusCMinus(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(Integer.MAX_VALUE);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstAPlusCMinus(){
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(Integer.MAX_VALUE-1);

    assertFalse(dreieck.istDreieck());
}
```

TestIstDreieckSeiteAPlusBKleinerC

Wie die methodennamen schon sagen, musste man hier lediglich erzwingen dass 2 Seiten zusammen kleiner eine andere sind.

```
@Test
public void testIstAPlusBKleinerC(){
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(5);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstAPlusCKleinerB(){
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(5);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}

@Test
public void testIstBPlusCKleinerA(){
    dreieck.setSeite_a(5);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.istDreieck());
}
```

TestIstDreieckTrue

Ganz zum schluss kommt diese Methode noch weil wir bis jetzt die ganze zeit nur erzwungen haben das die Methode false zurück gibt, doch wir wollen auch diesmal beide branches decken deswegen einmal auf true

```
@Test
public void testIstDreieckTrue(){
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(1);

    assertTrue(dreieck.istDreieck());
}
```

gleichSeitig()

TestGleichSeitig

Mit diesen beiden Methoden wurden die „Hauptbranches“ getestet, quasi die Grundfunktionalität. „Gibt die Funktion false aus wenn es kein gleichseitiges Dreieck ist?“

```
@Test
public void testGleichSeitig() {
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(1);

    assertTrue(dreieck.gleichSeitig());
}


@Test
public void testNichtGleichSeitig() {
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(3);

    assertFalse(dreieck.gleichSeitig());
}
```

TestSeitenGleichAusserA

Damit Methode true zurückgibt muss A gleich B sein und B gleich C. Man kann nun alle diese Branches decken indem man immer 2 Seiten gleich macht, außer eine bestimmte.

```
@Test
public void testSeitenGleichAusserA() {
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.gleichSeitig());
}

@Test
public void testSeitenGleichAusserB() {
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(1);
    dreieck.setSeite_c(2);

    assertFalse(dreieck.gleichSeitig());
}

@Test
public void testSeitenGleichAusserC() {
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(1);

    assertFalse(dreieck.gleichSeitig());
}
```

TestKeinDreieck

Es fehlt nur mehr ein letzter branch in der methode:

Wenn die Seiten gleichseitig sind aber es kein Dreieck ist.

```
@Test
public void testSeitenGleichKeinDreieck() {
    dreieck.setSeite_a(-1);
    dreieck.setSeite_b(-1);
    dreieck.setSeite_c(-1);

    assertFalse(dreieck.gleichSeitig());
}
```

gleichSchenkelig()

TestGleichSchenkelig

Es wird überprüft ob false zurückgegeben wird falls es keine 2 gleichen Seiten hat

=>man braucht nicht true überprüfen weil das im nächsten Test überprüft wird!

```
@Test
public void testKeinGleichSchenkeligesDreieck() {
    dreieck.setSeite_a(1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(3);

    assertFalse(dreieck.gleichSchenkelig());
}
```

TestA+B=C

In der methode gleichschenkelig wird überprüft ob A + B=C ist. Mit diesen Methoden wird jeder dieser branches gedeckt, außer ein bestimmter, der gleich erwähnt wird.

Es wird bei diesen tests auch der branch gedeckt das es ein gleichschenkeliges Dreieck ist.

```
@Test
public void testAPlusBGleich() {
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(3);

    assertTrue(dreieck.gleichSchenkelig());
}

@Test
public void testAPlusCGleich() {
    dreieck.setSeite_a(2);
    dreieck.setSeite_b(3);
    dreieck.setSeite_c(2);

    assertTrue(dreieck.gleichSchenkelig());
}

@Test
public void testBPlusCGleich() {
    dreieck.setSeite_a(3);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(2);

    assertTrue(dreieck.gleichSchenkelig());
}
```

TestA!=C

Dies ist ein Spezialfall, weil eigentlich sollte dieser branch doch schon oben gedeckt sein, oder nicht? Falsch! Das einzige Beispiel wo A ungleich C war, war das erste, und da wurde schon bei der ersten Überprüfen „getriggered“.

```
@Test
public void testANichtC() {
    dreieck.setSeite_a(3);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(4);

    assertFalse(dreieck.gleichSchenkelig());
}
```

TestKeinDreieck

Zum Schluss wird wieder der letzte branch gedeckt, und zwar derjenige der überprüft ob es ein Rechteck ist.

```
@Test
public void testKeinDreieck() {
    dreieck.setSeite_a(0);
    dreieck.setSeite_b(0);
    dreieck.setSeite_c(3);

    assertFalse(dreieck.gleichSchenkelig());
}
```

rechtWinkelig

TestIstSeiteGroesste

Es wird bei der Methode überprüft ob eine Seite die größte is, und falls ja, dann wird diese die Hypothenuse. Nun haben wir das simpleste rechtwinklige Dreieck(3,4,5), womit wir schon einmal mindestens einen branch gedeckt – es ist rechtwinkelig. Wenn wir jetzt jedesmal eine andere Seite die hypohenuse lassen werden, können wir fast alle branches.

```
@Test
public void testIstAGroesste() {
    dreieck.setSeite_a(5);
    dreieck.setSeite_b(4);
    dreieck.setSeite_c(3);

    assertTrue(dreieck.rechtWinkelig());
}

@Test
public void testIstBGroesste() {
    dreieck.setSeite_a(4);
    dreieck.setSeite_b(5);
    dreieck.setSeite_c(3);

    assertTrue(dreieck.rechtWinkelig());
}

@Test
public void testIstCGroesste() {
    dreieck.setSeite_a(3);
    dreieck.setSeite_b(4);
    dreieck.setSeite_c(5);

    assertTrue(dreieck.rechtWinkelig());
}
```

TestIstC<B & TestIstC>B<A

Es fehlen mit den oberen Methoden nur mehr 2 Branches, und zwar dass einmal C kleiner B ist und das C groesser B aber kleiner als A ist. Diese wurden wieder oben nicht gedeckt weil sie schon davor „getriggered“ wurden.

```
@Test
public void testIstCKleinerB() {
    dreieck.setSeite_a(5);
    dreieck.setSeite_b(5);
    dreieck.setSeite_c(4);

    assertFalse(dreieck.rechtWinkelig());
}

@Test
public void testIstCGroesserBKleinerA() {
    dreieck.setSeite_a(6);
    dreieck.setSeite_b(3);
    dreieck.setSeite_c(6);

    assertFalse(dreieck.rechtWinkelig());
}
```

testIstNichtRechtwinkel & TestIstKeinDreieck

Mit der methode IstNichtRechtwinkelig() wird der branch gedeckt was passiert wenn es eine hypothenus gibt aber dann doch kein rechtwinkliges Dreieck ist.

Mit IstKeinDreieck() wird wie schon erwähnt der Branch gedeckt was passiert wenn es erst garnicht ein Dreieck ist

```
@Test
public void testIstNichtRechtWinkelig() {
    dreieck.setSeite_a(3);
    dreieck.setSeite_b(3);
    dreieck.setSeite_c(4);

    assertFalse(dreieck.rechtWinkelig());
}

@Test
public void testIstKeinDreieck() {
    dreieck.setSeite_a(-1);
    dreieck.setSeite_b(2);
    dreieck.setSeite_c(3);

    assertFalse(dreieck.rechtWinkelig());
}
```

TestSuite erstellen und coverage checken

Man kann mit Eclipse gleich automatisch eine Testsuite erstellen mit:

Neu => Junit TestSuite => TestMethoden Anhackeln => erstellen

Diese TestSuite nun ausführen und man kann dann seine Coverage checken

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Dreieck	100,0 %	247	0	247
src	100,0 %	247	0	247
dreieck	100,0 %	247	0	247
Dreieck.java	100,0 %	247	0	247
Dreieck	100,0 %	247	0	247
Dreieck(int, int, int)	100,0 %	12	0	12
gleichSchenkelig()	100,0 %	22	0	22
gleichSeitig()	100,0 %	17	0	17
istDreieck()	100,0 %	96	0	96
rechtWinkelig()	100,0 %	88	0	88
setSeite_a(int)	100,0 %	4	0	4
setSeite_b(int)	100,0 %	4	0	4
setSeite_c(int)	100,0 %	4	0	4