Software Design Essentials

Software Testing met JUnit

Opgave 5

1 Doel van deze les

In dit werkcollege leer je hoe je software test met behulp van JUnit, een veelgebruikt framework in Java. Je ontdekt ook waarom Test-Driven Development (TDD) belangrijk is en hoe je het in de praktijk toepast.

2 Testen zonder een framework

Stel je voor dat je een eenvoudige Calculator klasse wilt testen zonder gebruik te maken van JUnit:

```
public class Calculator {
 public double add(double n1, double n2) {
  return n1 + n2;
 }
}
```

Dan kan je bijvoorbeeld testen via:

```
public class CalculatorTest {

    public void testAdd() {
        Calculator calc = new Calculator();
        double result = calc.add(10, 20);
        if (result != 30) {
            throw new IllegalStateException("Fout resultaat: " + result);
        }
    }

    public static void main(String[] args) {
```

```
        CalculatorTest test = new CalculatorTest();
        test.testAdd();
    }
}
```

Waarom is dit niet ideaal?

Deze aanpak kan werken voor kleine programma's, maar is onhandig bij grotere projecten. Daarom gebruiken we testframeworks zoals JUnit, die:

- helpen bij het schrijven van bruikbare testen,
- testen waardevol houden op lange termijn,
- het testproces versnellen en automatiseren.

3 Test-Driven Development (TDD)

In combinatie met JUnit wordt vaak Test-Driven Development (TDD) toegepast. Bij TDD schrijf je eerst de test, daarna pas de code.

Stappenplan bij TDD:

1. Schrijf een test voor de nieuwe functionaliteit
2. Laat de test compileren met "lege" methodes (stubs)
3. Run de test (die gaat falen)
4. Implementeer net genoeg code zodat de test slaagt
5. Run de test opnieuw (nu zou ze moeten slagen!)

4 Eerste JUnit Test: Maximum van drie getallen

We schrijven een methode berekenMax(int a, int b, int c) die het maximum van drie getallen teruggeeft. Deze methode komt in een klasse MathTool.

1. Maak een nieuw Java-project, bv. TestDrivenDevelopment.
2. Maak een testklasse MathToolTest.
3. Zet als package-naam: be.ehb.gdt.tools.math.

Gebruik de wizard om JUnit 5 toe te voegen aan je project.

```
package be.ehb.gdt.tools.math;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

```
public class MathToolTest {

    @Test
    public void testMax() {
        int expected = 10;
        int actual = MathTool.berekenMax(10, 5, 2);
        Assert.assertEquals(expected, actual);
    }
}
```

De testmethode moet de @Test annotatie krijgen, anders wordt ze niet uitgevoerd.

Voor deze test te laten slagen moet de methode berekenMax(...) bestaan, maar je mag voorlopig een dummywaarde teruggeven. Als je test faalt: perfect! Dan pas implementeer je de echte logica.

5 Extra testen toevoegen

Je kan meerdere tests toevoegen in één methode of aparte testmethodes maken:

```
@Test
public void testMaxCases() {
    Assert.assertEquals(3, MathTool.berekenMax(1,2,3));
    Assert.assertEquals(3, MathTool.berekenMax(3,2,1));
    Assert.assertEquals(3, MathTool.berekenMax(2,3,1));
    // enzovoort...
}
```

Of meerdere aparte testmethodes schrijven voor meer overzicht.

```
@Test
public void testMax123() {
    Assert.assertEquals(3, MathTool.berekenMax(1,2,3));
}

@Test
public void testMax321() {
    Assert.assertEquals(3, MathTool.berekenMax(3,2,1));
}
```

6 Een nieuwe test: berekenGem

Maak een nieuwe testmethode voor een methode berekenGem(float a, float b) die het gemiddelde berekent:

```
@Test
void testBerekenGem() {
    float expected = 5.5f;
    float actual = MathTool.berekenGem(4.0f, 7.0f);
    assertEquals(expected, actual, 0.001f);
}
```

Implementeer de methode pas nadat de test faalt. Dit is de kern van TDD.

7 Testonafhankelijkheid en initialisatie

Testmethoden mogen niet afhankelijk zijn van elkaar. De volgorde waarin ze worden uitgevoerd is willekeurig.

Gebruik @BeforeEach en @AfterEach om voorbereidend werk te doen of op te ruimen:

```
@BeforeEach
void setUp() {
    // Initialisatie, bv. een object aanmaken
}

@AfterEach
void tearDown() {
    // Resources vrijgeven
}
```

Voorbeeld met Bankrekening

```
public class Bankrekening {
    private int rekeningnummer;
    private double saldo;

    public Bankrekening(int nr, double startBedrag) {
        rekeningnummer = nr;
        saldo = startBedrag;
    }

    public double getSaldo() {
        return saldo;
    }

    public void stort(double bedrag) {
        saldo += bedrag;
    }
```

```
    public void haalAf(double bedrag) {
        saldo -= bedrag;
    }
}
```

Tests met setup:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class BankrekeningTest {

    Bankrekening rekening;

    @BeforeEach
    void setUp() {
        rekening = new Bankrekening(1, 100);
    }

    @Test
    void testStort() {
        rekening.stort(200);
        assertEquals(300, rekening.getSaldo(), 0.001);
    }

    @Test
    void testHaalAf() {
        rekening.haalAf(50);
        assertEquals(50, rekening.getSaldo(), 0.001);
    }
}
```

8 Veelgebruikte Assert methoden

Methode	Wat doet het?
assertEquals(expected, actual)	Vergelijkt waarden
assertTrue(condition)	Test dat de conditie waar is
assertFalse(condition)	Test dat de conditie niet waar is
assertNull(object)	Test dat iets null is
assertNotNull(object)	Test dat iets niet null is
assertSame(obj1, obj2)	Vergelijkt of referenties hetzelfde zijn
fail()	Forceert falen van een test

9 Uitbreiding Bankrekening

Breid de klasse Bankrekening uit:

1. haalAfVeilig(double bedrag)
    - Haalt geld af, maar gooit een exception als het saldo onder nul zou gaan.
2. voegSamen(Bankrekening ander)
    - Voegt het saldo van een andere rekening toe aan deze.

Voorbeeldtest voor exception:

```
@Test
void testHaalAfVeilig() {
    assertThrows(IllegalArgumentException.class, () -> {
        rekening.haalAfVeilig(200);
    });
}
```

Tips

- Laat elke test onafhankelijk zijn van andere testen
- Zorg voor duidelijke namen en foutmeldingen
- Denk logisch na over mogelijke foute input