



samen sterk voor werk

Test Driven Development met

JUnit



Testers willen geen dingen breken, maar de illusie verdrijven dat sommige dingen werken

Deze cursus is eigendom van de VDAB©

Inhoudsopgave

1	INLEIDING.....	4
1.1	Doelstelling.....	4
1.2	Vereiste voorkennis.....	4
1.3	Nodige software	4
1.4	De noodzaak van testen	4
1.5	Geautomatiseerd testen en JUnit	4
1.6	Unit tests en integration tests.....	4
1.7	Project structuur	5
1.8	Testen bijhouden.....	5
2	KENNISMAKING	6
2.1	Voorbeeldproject	6
2.2	Te testen class	6
2.3	Unit test.....	7
2.4	Unit test uitvoeren	8
2.5	Één @Test method uitvoeren	8
2.6	Assert methods	9
2.7	import static	9
2.7.1	Algemeen	9
2.7.2	Code completion	10
2.8	assertEquals en BigDecimals	10
2.9	Unit test als documentatie.....	11
2.10	Refactoring	12
2.11	Overzicht van wat je tot nu leerde.....	12
3	EERST TESTS SCHRIJVEN, DAARNA IMPLEMENTATIE	13
3.1	Te testen class	13
3.2	Unit test.....	14

3.3	Eerste implementatie	15
3.4	Refactoring	15
3.5	Samenvatting.....	15
4	TEST FIXTURES EN @BEFORE	16
4.1	Samenvatting van wat je hier leerde	17
5	TO TEST OR NOT TO TEST	18
5.1	Getters en Setters	18
5.2	equals en hashCode	18
5.3	Exceptions	19
5.4	Grenswaarden en extreme waarden	19
5.4.1	Algemeen	19
5.4.2	Voorbeeld: een class Rekeningnummer (van een bankrekening).....	20
5.4.3	Voorbeeld: een verzameling	22
6	CODE COVERAGE	24
6.1	EclEmma gebruiken.....	24
7	MEERDERE UNIT TESTS UITVOEREN	25
7.1	Alle unit tests van één package uitvoeren	25
7.2	Alle unit tests van het project uitvoeren.....	25
7.3	Test suite	25
8	DEPENDENCIES	26
8.1	Voorbeeld.....	26
8.2	Problemen	27
8.3	De dependency als een interface	28
8.4	Dependency injection.....	28
9	STUB.....	29
9.1	Classes en interfaces gebruikt in de test	29
9.2	Classes en interfaces gebruikt in productiecode	29
9.3	Praktisch	29

10	MOCK	31
10.1	Algemeen	31
10.1.1	Variabel resultaat van method oproep	31
10.1.2	Verificaties	31
10.2	Mockito	31
10.2.1	pom.xml	31
10.2.2	Mock aanmaken	32
10.2.3	Mock trainen	32
10.2.4	Verificaties	33
10.3	Getuigenis op het einde van deze cursus	33
11	HERHALINGSOEFENING	34
12	STAPPENPLAN 	35
13	COLOFON	36

1 INLEIDING

1.1 Doelstelling

Code ontwikkelen op de manier van Test Driven Development en testen met JUnit.

1.2 Vereiste voorkennis

- Java
- Maven

1.3 Nodige software

- Een JDK (Java Developer Kit) met versie 7 of hoger.
- Eclipse IDE for Java EE Developers (versie Oxygen of hoger).

1.4 De noodzaak van testen

Een studie van het National Institute of Standards and Technology geeft aan dat softwarefouten jaarlijks 59,5 miljard \$ kosten aan de economie van de USA.

Uit de studie blijkt ook dat men een derde van deze kosten kan vermijden door de manier van testen te verbeteren. Ook blijkt dat hoe vroeger men een fout ontdekt in de ontwikkeltijd van een applicatie, hoe minder deze fout kost.

Test Driven Development (TDD) is een manier van software ontwikkelen waarbij:

- testen een centraal onderdeel van de ontwikkeling zijn
- testen vroeg in de ontwikkeltijd worden toegepast

1.5 Geautomatiseerd testen en JUnit

Wanneer je nieuwe code schrijft, test je de werking van die code.

Als je de code wijzigt, herhaal je de testen. Testen is dus een repetitieve taak.

Een ontwikkelaar automatiseert repetitieve taken, door ze als een programma te schrijven.

Bij TDD doet de ontwikkelaar dit ook voor tests: hij test met een testprogramma.

JUnit is het populairste Java framework waarmee je zo'n testprogramma vlot schrijft.

JUnit is ook geïntegreerd in Java IDE's (Eclipse NetBeans, ...) en Java build tools (Maven, ...)

1.6 Unit tests en integration tests

Een class waarvan je de code wil testen heet een *unit*.

Een class die de tests bevat voor één te testen class heet een *unit test* of *test case*.

Terwijl je een class ontwikkelt, kan je al de unit test van die class schrijven en uitvoeren.

Je krijgt zo snel feedback over fouten in de class.

Als de class af is, is ze ook volledig getest. De kans is minimaal dat ze nog fouten bevat.

Je debugt minder. Dit is goed: debuggen is een tijdrovende, vervelende taak.

Een integration test is een class waarin je de samenwerking van meerdere classes test.

In het onderdeel "Klant toevoegen" van een website werken deze classes samen:

- een servlet class `KlantToevoegenServlet`
- een entity class `Klant`
- een class `KlantService` uit de services layer
- een class `KlantRepository` uit de repositories layer

Je leert hier unit tests kennen. Je leert in de cursus Spring integration tests kennen.

1.7 Project structuur

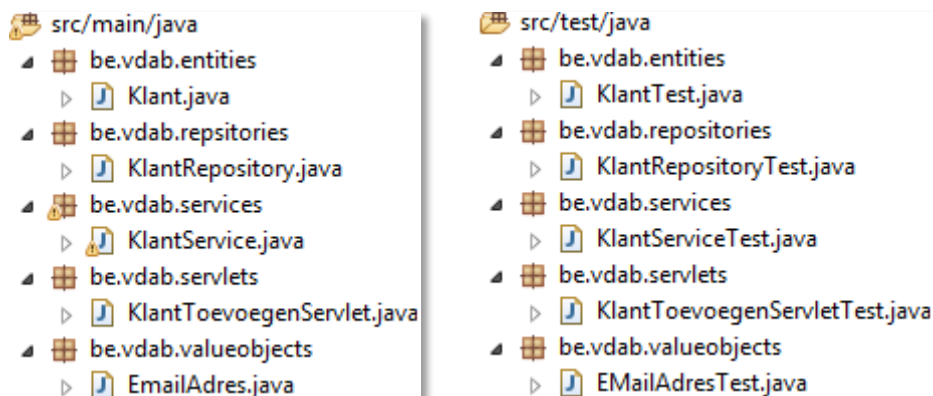
We raden af unit tests in dezelfde folder te plaatsen als gewone classes:

- De folder bevat dan veel classes, wat het overzicht vermindert.
 - Je levert de unit test code niet mee wanneer je de applicatie in productie neemt.
- Als de unit tests en de gewone classes zich in dezelfde folder bevinden, is het moeilijk een JAR bestand of WAR bestand te maken dat geen test code bevat

We plaatsen de unit tests in een aparte project folder, volgens de Maven folder structuur:

- `src/main/java` bevat de gewone classes
- `src/test/java` bevat de unit tests

`src/test/java` heeft dezelfde package structuur als `src/main/java`:



Voor de Java compiler bevinden `KlantTest` en `Klant` zich in dezelfde package (hoewel hun bovenliggende source folder verschilt).

Je kan zo vanuit `KlantTest` niet enkel de public methods van `Klant` testen, maar ook de protected methods en de methods met package visibility (methods waar geen public, protected of private voor staat).

Je kan vanuit `KlantTest` de private methods van `Klant` niet aanspreken.

Dit is niet erg. Deze private methods worden in `Klant` opgeroepen door public methods, protected methods of methods met package visibility.

Je kan deze methods wel testen vanuit `KlantTest`.

Door deze methods op te roepen, voer je indirect toch de private methods van `Klant` uit en test je ook de code in die private methods.



Opmerking: entities is een verzamelnaam voor objecten. Een entity heeft meerdere private variabelen, waarvan één de entity uniek identificeert ten opzichte van andere entities uit dezelfde class. Bij de entity `Klant` is dit de private variabele `klantNr`.

Value objects is ook een verzamelnaam voor objecten. Bij een value object identificeren alle private variabelen samen het object ten opzichte van andere objecten uit dezelfde class. Bij het value object `Datum` zijn dit de private variabelen `dag`, `maand` en `jaar`. Je leert meer over het onderscheid tussen entities en value objects in de cursus JPA met Hibernate.

1.8 Testen bijhouden

Je verwijdt nooit unit tests, tenzij ze niet meer relevant zijn omdat de analyse wijzigt.

Elke keer je de te testen class achteraf bijwerkt, zie je na of die aanpassing geen nieuwe fouten introduceerde, door de bijbehorende unit test uit te voeren.

2 KENNISMAKING

2.1 Voorbeeldproject

Je maakt in Eclipse een Maven console applicatie:

1. Je kiest File, New, Other, Maven, Maven Project en je kiest Next.
2. Je plaatst een vinkje bij Create a simple project en je kiest Next.
3. Je tikt be.vdab bij Group Id.
4. Je tikt applicatieMetTesten bij Artifact Id en je kiest Finish.

Je ziet in de Project Explorer twee source folders:

- src/main/java gewone classes
- src/test/java unit tests

Je voegt de JUnit dependency toe aan pom.xml, onder <version>:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

(1) Je plaatst de scope op test.
De JUnit classes zijn ter beschikking gedurende de applicatie ontwikkeling, Maven neemt deze classes niet op in het eindproduct (JAR, WAR) van je applicatie.

Je geeft aan dat je Java 8 gebruikt en de sources uitgedrukt zijn in UTF-8, onder </dependencies>:

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Je klikt met de rechtermuisknop op het project en je kiest Maven, Update Project en OK.

2.2 Te testen class

Jaar
-jaar: int
+Jaar(jaar: int)
+isSchrikkeljaar(): boolean

- Een jaar deelbaar door 4 is een schrikkeljaar.
- Een jaar deelbaar door 100 is echter geen schrikkeljaar.
- Een jaar deelbaar door 400 is echter wel een schrikkeljaar.

Je voegt deze class toe aan het project:

1. Je klikt in de Project Explorer met de rechtermuisknop op src/main/java.
2. Je kiest New, Class.
3. Je tikt be.vdab.valueobjects bij Package.
4. Je tikt Jaar bij Name en je kiest Finish.

```
package be.vdab.valueobjects;
public class Jaar {
    private final int jaar;
    public Jaar(int jaar) {
        this.jaar = jaar;
    }
    public boolean isSchrikkeljaar() {
        if (jaar % 400 == 0) {
            return true;
        }
        if (jaar % 100 == 0) {
            return false;
        }
        return jaar % 4 == 0;
    }
}
```

Zonder unit test zou je de class testen door in de user interface van de applicatie jaartallen te tikken en te controleren of je een juist bericht ziet.

Nadelen:

- ⊖ Er is veel tijd tussen het schrijven van de class en het testen van de class. Het zou beter zijn dat je de class kunt testen terwijl je ze schrijft: op dat moment zijn je gedachten geconcentreerd op de class.
- ⊖ Het tikken van jaartallen en het klikken op de knop Controleer jaar is een manuele activiteit die tijd vraagt en na enkele keren saai is.

Jaartal: 2013	Controleer jaar
Dit is geen schrikkeljaar	

2.3 Unit test

1. Je klikt in de Project Explorer met de rechtermuisknop op src/test/java.
2. Je kiest New, Other, Java, JUnit, JUnit Test Case en je kiest Next.
3. Je tikt be.vdab.valueobjects bij Package.
4. Je tikt JaarTest bij Name en je kiest Finish.

```
package be.vdab.valueobjects;
import org.junit.Assert;
import org.junit.Test;
public class JaarTest {
    @Test
    public void eenJaarDeelbaarDoor400IsEenSchrikkeljaar() {
        Jaar jaar = new Jaar(2000);
        Assert.assertEquals(true, jaar.isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor100IsGeenSchrikkeljaar() {
        Assert.assertEquals(false, new Jaar(1900).isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor4IsEenSchrikkeljaar() {
        Assert.assertEquals(true, new Jaar(2012).isSchrikkeljaar());
    }
    @Test
    public void eenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar() {
        Assert.assertEquals(false, new Jaar(2015).isSchrikkeljaar());
    }
}
```

①
②
③
④
⑤
⑥

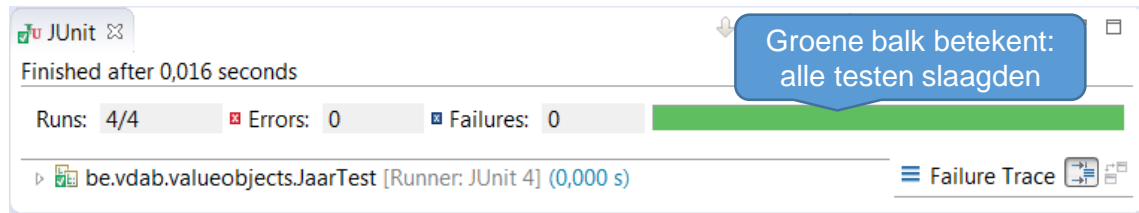
- (1) Je importeert de package org.junit, niet uit de oudere package junit.framework.
- (2) Een unit test is een public class met een vrij te kiezen naam.
De conventie is dat de naam gelijk is aan de te testen class, gevolgd door Test.
- (3) Een test is één method waarbij je @Test schrijft (een @Test method).
- (4) Een @Test method is public, heeft als returntype void en heeft geen parameters.
Je kan de naam vrij kiezen. Het is aan te raden dat die aangeeft wat je gaat testen.
- (5) Je maakt een object van de te testen class. Je zorgt er voor dat dit object in de toestand komt die je wilt testen. Je gebruikt een voorbeeldwaarde (400) die past bij de test.
Je vindt zo'n waarde in de analyse, in voorbeelddocumenten, op het internet, ...
of je bedenkt zelf een waarde. Deze waarde is hard gecodeerd. Het is niet de bedoeling dat je de waarde instelt met berekeningen of algoritmes. Er is dan kans op denkfouten.
Een foute test kan de werking van de gewone class nooit correct testen !
- (6) Je voert isSchrikkeljaar uit. Als deze method correct is, geeft ze true.
De JUnit class Assert bevat static methods waarmee je dit controleert.
Je geeft aan assertEquals twee waarden mee: de waarde die je verwacht (true) en de echte waarde die je krijgt bij de method oproep. Als de waarden aan mekaar gelijk zijn, aanziet JUnit de test als correct verlopen. Anders is de test mislukt.

2.4 Unit test uitvoeren

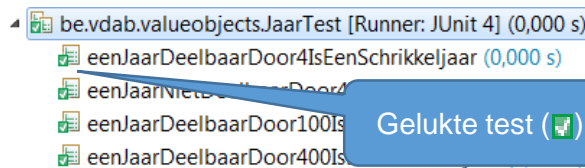
Je laat JUnit de unit test uitvoeren. JUnit voert hierbij alle @Test methods uit.

Je klikt met de rechtermuisknop op `JaarTest.java` en je kiest `Run As, JUnit Test`.

JUnit voert de unit test uit en toont een rapport, onder in Eclipse;

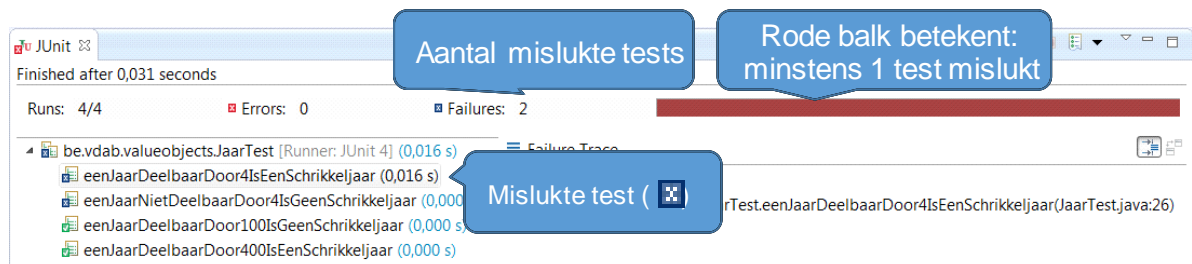


Je klikt op `▶`. Bemerkt dat JUnit de @Test methods in willekeurige volgorde uitvoert:



Je tikt een fout in de method `isSchrikkeljaar`: je vervangt 4 door 5. Je bewaart de source.

Je voert de unit test opnieuw uit met een klik op `↻` in het rapport van de test:



De eerste foutieve test is geselecteerd. De rechterhelft bevat detailinformatie over de test.

- `java.lang.AssertionError: expected<true> but was <false>` betekent: je beschreef met `Assert.assertEquals(true, ...)` dat je de waarde `true` verwacht maar de waarde die je kreeg van `isSchrikkeljaar` verschillend was: `false`.
- `at be.vdab.valueobjects.JaarTest.EenJaarDeelbaarDoor4IsEenSchrikkeljaar (JaarTest.java:26)` duidt de class, de method en het lijnnummer aan van de mislukte test. Als je de regel dubbelklikt, of links de mislukte test dubbelklikt, opent Eclipse de unit test source en toont de regel met de mislukte test met een blauwe achtergrond.

Je corrigeert in `isSchrikkeljaar` 5 door 4. Je voert de unit test opnieuw uit. Die slaagt.

Elke keer je een method van een class wijzigt, voer je de unit tests van die class uit.

Je krijgt direct feedback of die wijziging correct was. Dit is leuk en productief !

2.5 Één @Test method uitvoeren

1. Je klikt in de Project Explorer op `▶` voor `JaarTest.java`.
2. Je klikt op `▶` voor `JaarTest`. Je ziet een lijstje met de unit test methods.
3. Je klikt met de rechtermuisknop op zo'n method en je kiest `Run As, JUnit Test`.

2.6 Assert methods

De class Assert bevat naast assertEquals nog methods:

- `assertNotEquals(verwachteExpressie, teTestenExpressie)`
De test lukt als teTestenExpressie verschilt van verwachteExpressie.
- `assertArrayEquals(verwachteArray, teTestenArray)`
De test lukt als het aantal elementen in beide array's gelijk zijn en elk element in teTestenArray gelijk is aan het element met hetzelfde volgnummer in verwachteArray.
- `assertTrue(teTestenExpressie)`
De test lukt als teTestenExpressie gelijk is aan true.
- `assertFalse(teTestenExpressie)`
De test lukt als teTestenExpressie gelijk is aan false.
- `assertNull(teTestenExpressie)`
De test lukt als teTestenExpressie gelijk is aan null.
- `assertNotNull(teTestenExpressie)`
De test lukt als teTestenExpressie verschilt van null.
- `assertSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`
De test lukt als teTestenReferenceVariabele naar hetzelfde object wijst als verwachteReferenceVariabele.
- `assertNotSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`
De test lukt als teTestenReferenceVariabele niet naar hetzelfde object wijst als verwachteReferenceVariabele.

JaarTest wordt korter met assertTrue en assertFalse:

```
public class JaarTest {
    @Test
    public void eenJaarDeelbaarDoor400IsEenSchrikkeljaar() {
        Assert.assertTrue(new Jaar(2000).isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor100IsGeenSchrikkeljaar() {
        Assert.assertFalse(new Jaar(1900).isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor4IsEenSchrikkeljaar() {
        Assert.assertTrue(new Jaar(2012).isSchrikkeljaar());
    }
    @Test
    public void eenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar() {
        Assert.assertFalse(new Jaar(2015).isSchrikkeljaar());
    }
}
```

2.7 import static

2.7.1 Algemeen

Om een static method aan te spreken, schrijf je de bijbehorende class, een punt en daarna de static method: `Assert.assertTrue(...)`;

Je kan een static method korter aanspreken met de sleutelwoorden `import static`

Je schrijft bij `import static` een package, een class, een punt en

- de naam van een static method
Je kan daarna die static method oproepen, zonder er de class voor te schrijven
Voorbeeld: boven in de source `import static org.junit.Assert.assertTrue;`
Verder in de source `assertTrue(...);`

- of een sterretje
Je kan daarna elke static method van die class oproepen, zonder er de class voor te schrijven.
Voorbeeld: boven in de source `import static org.junit.Assert.*;`
Verder in de source `assertTrue(...);`

```
package be.vdab.valueobjects;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import org.junit.Test;
public class JaarTest {
    @Test
    public void eenJaarDeelbaarDoor400IsEenSchrikkeljaar() {
        assertTrue(new Jaar(2000).isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor100IsGeenSchrikkeljaar() {
        assertFalse(new Jaar(1900).isSchrikkeljaar());
    }
    @Test
    public void eenJaarDeelbaarDoor4IsEenSchrikkeljaar() {
        assertTrue(new Jaar(2012).isSchrikkeljaar());
    }
    @Test
    public void eenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar() {
        assertFalse(new Jaar(2015).isSchrikkeljaar());
    }
}
```

2.7.2 Code completion

Als je het begin van een assert method tikt, voorziet Eclipse code completion.

Als je bijvoorbeeld ass tikt gevolgd door Ctrl + Spatie,

krijg je een pop-up venster waarin je de method assertTrue kan kiezen.

Eclipse zal ook de opdracht `import static org.junit.Assert.assertTrue;` toevoegen.

2.8 assertEquals en BigDecimal

De method `assertEquals(verwachteExpressie, teTestenExpressie)` controleert of `verwachteExpressie` gelijk is aan `teTestenExpressie`.

- Als de expressies een primitief datatype hebben, gebruikt `assertEquals ==`.
- Als de expressies objecten zijn, roept `assertEquals` op `verwachteExpressie` de method `equals` op en geeft `teTestenExpressie` mee als parameter.

Dit werkt goed, behalve bij `BigDecimal` objecten.

Twee `BigDecimal` objecten zijn volgens hun method `equals` aan mekaar gelijk als ze dezelfde wiskundige waarde bevatten én hetzelfde aantal cijfers na de komma bijhouden.

Voorbeeld: een `BigDecimal` kan overvallige decimale nullen bevatten

```
BigDecimal.valueOf(3).equals(BigDecimal.valueOf(1.5).add(BigDecimal.valueOf(1.5)));
```

geeft false : de som (3.0) bevat evenveel decimalen als het grootste aantal decimalen in één van de oorspronkelijke `BigDecimal` objecten (1.5)

Je vergelijkt twee `BigDecimals` beter met de method `compareTo`. Deze geeft 0 terug als deze `BigDecimals` dezelfde waarde bevatten, los van het aantal cijfers na de komma.

```
BigDecimal.valueOf(3).compareTo(
    BigDecimal.valueOf(1.5).add(BigDecimal.valueOf(1.5)))
```

geeft bijvoorbeeld 0

Je maakt in `src/main/java` een package `be.vdab.util` en daarin een class `Converter`. Deze bevat een method `inchesNaarCentimeters`:

```
package be.vdab.util;
import java.math.BigDecimal;
public class Converter {
    private static final BigDecimal AANTAL_CENTIMETER_IN_EEN_INCH
        = BigDecimal.valueOf(2.54);
    public BigDecimal inchesNaarCentimeters(BigDecimal inches) {
        return inches.multiply(AANTAL_CENTIMETER_IN_EEN_INCH);
    }
}
```

Je maakt in `src/test/java` een package `be.vdab.util`. Je maakt daarin een JUnit Test Case `ConverterTest`. Deze bevat tests voor de method `inchesNaarCentimeters`:

```
package be.vdab.util;
import static org.junit.Assert.*;
import java.math.BigDecimal;
import org.junit.Test;
public class ConverterTest {
    @Test
    public void eenInchIs2Punt54Centimeters() {
        Converter converter = new Converter();
        assertEquals(0, BigDecimal.valueOf(2.54).compareTo(
            converter.inchesNaarCentimeters(BigDecimal.ONE)));
    }
    @Test
    public void tweehonderdInchesIsVijfhonderdenachtCentimeters() {
        Converter converter = new Converter();
        assertEquals(0, BigDecimal.valueOf(508)
            .compareTo(converter.inchesNaarCentimeters(BigDecimal.valueOf(200))));
    }
}
```

Je kan de unit test uitvoeren.

2.9 Unit test als documentatie

Het rapport van een unit test is tegelijk ook prima documentatie over de werking van die class:

```
eenJaarDeelbaarDoor4IsEenSchrikkeljaar
eenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar
eenJaarDeelbaarDoor100IsGeenSchrikkeljaar
eenJaarDeelbaarDoor400IsEenSchrikkeljaar
```

Als Mieke de class `Jaar` en de unit test `JaarTest` schreef, leert Jan de werking van de class `Jaar` kennen door het rapport van de bijbehorende unit test `JaarTest` in te kijken.

Hij ziet bijvoorbeeld dat een jaar dat deelbaar is door 400 een schrikkeljaar is.

Dit lukt als de `@Test` methods duidelijke namen hebben:

`eenJaarDeelbaarDoor400IsEenSchrikkeljaar`

Sommige ontwikkelaars wijken in de namen van unit test methods af van de standaard Java naamgevingconventies en gebruiken underscores tussen de woorden.

Ze doen dit om de leesbaarheid te bevorderen. Gezien enkel JUnit deze methods oproept (jij roept ze niet op in je applicatie code), heeft dit geen nadelen in je applicatie code.

```
een_jaar_niet_deelbaar_door_4_is_geen_schrikkeljaar
een_jaar_deelbaar_door_4_is_een_schrikkeljaar
een_jaar_deelbaar_door_400_is_een_schrikkeljaar
een_jaar_deelbaar_door_100_is_geen_schrikkeljaar
```

2.10 Refactoring

Nu de class Jaar werkt, zie je na of je in deze class refactoring kan toepassen. Je verbetert hierbij de leesbaarheid en onderhoudbaarheid van de code. Het kan ook dat je de performantie of geheugengebruik verbetert.

Je voert daarna de unit test opnieuw uit.

- Als de tests slagen, is je refactoring OK.
- Als een test mislukt, corrigeer je de refactoring tot alle tests slagen.

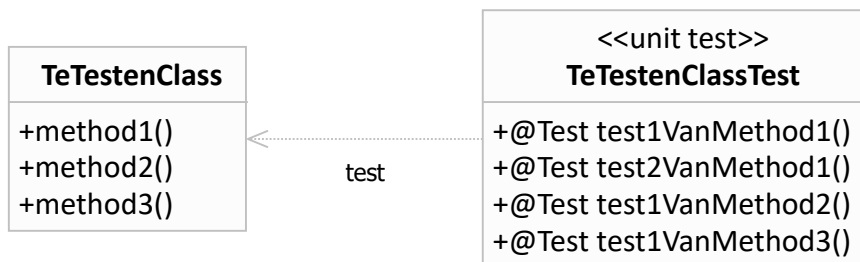
Je wijzigt als voorbeeld de code in de method isSchrikkeljaar:

```
return jaar % 4 == 0 && jaar % 100 != 0 || jaar % 400 == 0;
```

Je commit de sources en je publiceert op GitHub.

Je voert de unit tests opnieuw uit. Gezien deze slagen is deze wijziging correct.

2.11 Overzicht van wat je tot nu leerde



Palindroom: zie takenbundel

3 EERST TESTS SCHRIJVEN, DAARNA IMPLEMENTATIE

Bij Jaar en JaarTest heb je Jaar volledig geschreven, pas daarna de JaarTest.

In dit hoofdstuk wijzigt deze volgorde, zoals aanbevolen door Test Driven Development:

1. Je schrijft *eerst* de unit test.
2. Je schrijft *daarna* de gewone class.

Voordelen:

- ⊕ Je schrijft de testen op basis van de analyse van de class, als `@Test` methods:
 - een jaar deelbaar door 4 is een schrikkeljaar.
`@Test public void eenJaarDeelbaarDoor4IsEenSchrikkeljaar()`
`@Test public void eenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()`
 - een jaar deelbaar door 100 is echter geen schrikkeljaar.
`@Test public void eenJaarDeelbaarDoor100IsGeenSchrikkeljaar()`
 - een jaar deelbaar door 400 is echter wel een schrikkeljaar.
`@Test public void eenJaarDeelbaarDoor400IsEenSchrikkeljaar()`

Je bent zo verplicht de vereisten uit de analyse nog eens grondig te bestuderen vooraleer je die uitwerkt in de class Jaar.

Je zal ook geen overbodige functionaliteit in de class Jaar schrijven: zodra alle testen slagen heb je genoeg code in de class geschreven.

- ⊕ Je roept binnen de `@Test` methods de methods op van de te testen class. Als deze method oproepen vreemd aandoen, wegens een verkeerd gekozen methodnaam, te veel of te weinig parameters, ... kan je snel de method signatuur wijzigen: ze zijn enkel vanuit de `@Test` methods opgeroepen. Het kan ook dat je tijdens het uitvoeren van de `@Test` methods merkt dat er nog methods ontbreken in de te testen class.
- ⊕ Tijdens het schrijven van de unit test denk je na over de werking van de te testen class. Pas daarna schrijf je deze class uit. Dit leidt tot betere code.
- ⊕ Als je eerst code schrijft en pas daarna de tests voor die code, volg je in de tests onbewust hetzelfde algoritme dat je in de code geschreven hebt. Als dit algoritme fout is, is de test ook fout.

3.1 Te testen class

Als je geen enkele letter code zou mogen schrijven van de te testen class, kan je ook niet naar deze class verwijzen binnen de unit test.

Je schrijft daarom de class met minimale methods.

Elke method bevat één opdracht: `throw new UnsupportedOperationException();`

`UnsupportedOperationException` geeft expliciet aan dat een oproep van deze method momenteel nog niet ondersteund (*unsupported*) is.

```
package be.vdab.entities;
import java.math.BigDecimal;
public class Rekening {
    public void storten(BigDecimal bedrag) {
        throw new UnsupportedOperationException();
    }
    public BigDecimal getSaldo() {
        throw new UnsupportedOperationException();
    }
}
```

Rekening
-saldo: BigDecimal
+storten(bedrag: BigDecimal) +getSaldo(): BigDecimal()

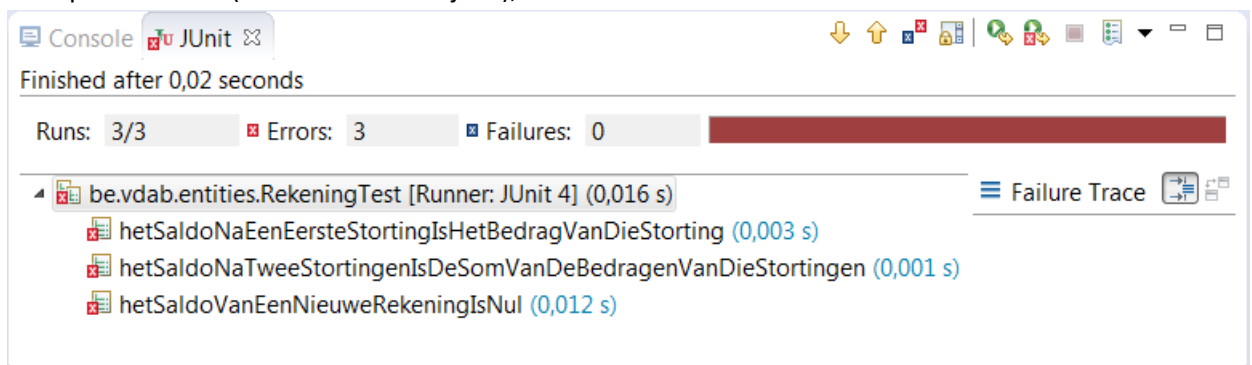
3.2 Unit test

Je schrijft nu de unit test voor deze class, op basis van de analyse. Daaruit blijkt:

- Het saldo van een nieuwe rekening is 0 €.
- Het saldo van een rekening met één storting is gelijk aan die storting.
- Het saldo van een rekening met twee stortingen is de som van die stortingen.

```
package be.vdab.entities;
import java.math.BigDecimal;
import org.junit.Test;
import static org.junit.Assert.*;
public class RekeningTest {
    @Test
    public void hetSaldoVanEenNieuweRekeningIsNul() {
        Rekening rekening = new Rekening();
        assertEquals(0, BigDecimal.ZERO.compareTo(rekening.getSaldo()));
    }
    @Test
    public void hetSaldoNaEenEersteStortingIsHetBedragVanDieStorting() {
        Rekening rekening = new Rekening();
        BigDecimal bedrag = BigDecimal.valueOf(2.5);
        rekening.storten(bedrag);
        assertEquals(0, bedrag.compareTo(rekening.getSaldo()));
    }
    @Test
    public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen() {
        Rekening rekening = new Rekening();
        rekening.storten(BigDecimal.valueOf(2.5));
        rekening.storten(BigDecimal.valueOf(1.2));
        assertEquals(0, BigDecimal.valueOf(3.7).compareTo(rekening.getSaldo()));
    }
}
```

Je controleert of je aan de verleiding kon weerstaan om methods in de class Rekening te implementeren (er code in te schrijven), door de unit test uit te voeren:



Alle tests mislukten als Errors omdat de Rekening methods een exception werpen. Dit is deze keer goed nieuws.

3.3 Eerste implementatie

Je schrijft nu code in de Rekening methods. De eerste versie van die code is niet noodzakelijk de optimaalste code (performantie, geheugengebruik, leesbaarheid,...).

```
package be.vdab.entities;
import java.math.BigDecimal;
public class Rekening {
    private BigDecimal saldo = BigDecimal.valueOf(0);
    public void storten(BigDecimal bedrag) {
        saldo = saldo.add(bedrag);
    }
    public BigDecimal getSaldo() {
        return saldo;
    }
}
```

Je voert de bijbehorende unit test (RekeningTest) uit en alle tests slagen.

Dit betekent dat de class Rekening voldoet aan de vereisten van de analyse.

3.4 Refactoring

Nu de class Rekening werkt, zie je na of je in deze class refactoring kan toepassen.

Je verbetert hierbij de leesbaarheid en onderhoudbaarheid van de code.

Het kan ook dat je de performantie of geheugengebruik verbetert.

Je voert daarna de unit test opnieuw uit.

- Als de tests slagen, is je refactoring OK.
- Als een test mislukt, corrigeer je de refactoring tot alle tests slagen.

Je kan in de class Rekening één kleine refactoring doen. Je wijzigt de regel

```
private BigDecimal saldo = BigDecimal.valueOf(0); naar
```

```
private BigDecimal saldo = BigDecimal.ZERO;
```

①

- (1) Je maakt geen nieuwe BigDecimal met het getal 0, maar je gebruikt de constante ZERO die verwijst naar een bestaande BigDecimal met het getal 0.

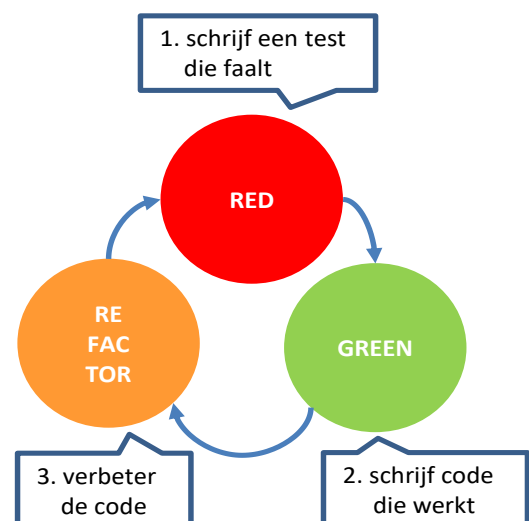
Je commit de sources en je publiceert op GitHub.

Je voert RekeningTest uit. Alle tests slagen. De refactoring introduceerde dus geen fouten.

3.5 Samenvatting

1. Je analyseert de te schrijven class.
2. Je schrijft de class en zijn methods.
De methods bevatten één opdracht:
`throw new UnsupportedOperationException();`
3. Je schrijft de unit test van de class, gebaseerd op de analyse.
4. Je voert de unit test uit, die mislukken.
5. Je schrijft echte code in de class en voert de unit tests uit, tot de tests slagen.
6. Je past op de class refactoring toe en controleert met unit tests of deze geen nieuwe fouten introduceren, tot de code van de class optimaal is.

Motto: **keep the bar green to keep the code clean**



Veiling: zie takenbundel

4 TEST FIXTURES EN @BEFORE

Je initialiseert in elke RekeningTest test eenzelfde Rekening object:

```
Rekening rekening = new Rekening();
```

Zo'n object heet een test fixture.

Je vermijdt dit herhalen in twee stappen:

1. Je declareert een private variabele per test fixture: **private** Rekening rekening;
2. Je initialiseert de variabele in een @Before method.

Dit is een **public void** method, zonder parameters, waarvoor je @Before tikt.

```
@Before
public void before() {
    rekening = new Rekening();
}
```

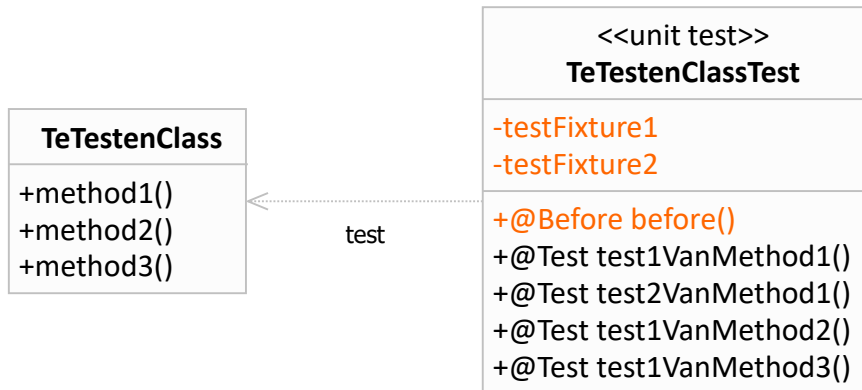
JUnit voert voor elke @Test method de @Before method uit.

Je kan nu in @Test methods deze private variabele rekening gebruiken. JUnit initialiseert deze variabele bij elke oproep van de @Before method met een Rekening object.

```
package be.vdab.entities;
import static org.junit.Assert.assertEquals;
import java.math.BigDecimal;
import org.junit.Before;
import org.junit.Test;
public class RekeningTest {
    private Rekening rekening;
    @Before
    public void before() {
        rekening = new Rekening();
    }
    @Test
    public void hetSaldoVanEenNieuweRekeningIsNul() {
        assertEquals(0, BigDecimal.ZERO.compareTo(rekening.getSaldo()));
    }
    @Test
    public void hetSaldoNaEenEersteStortingIsHetBedragVanDieStorting() {
        BigDecimal bedrag = BigDecimal.valueOf(2.5);
        rekening.storten(bedrag);
        assertEquals(0, bedrag.compareTo(rekening.getSaldo()));
    }
    @Test
    public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen() {
        rekening.storten(BigDecimal.valueOf(2.5));
        rekening.storten(BigDecimal.valueOf(1.2));
        assertEquals(0, BigDecimal.valueOf(3.7).compareTo(rekening.getSaldo()));
    }
}
```

Je commit de sources en je publiceert op GitHub. Je kan de unit test opnieuw uitvoeren.

4.1 Samenvatting van wat je hier leerde



Opmerking: je kan in een unit test ook een method opnemen waarvoor je @After tikt. JUnit voert deze @After method uit na elke @Test method. Je hebt een @After method enkel nodig als je in de @Before method iets aanmaakt buiten het RAM geheugen (bijvoorbeeld een bestand). Je verwijdert dit bestand in een @After method.



Test fixtures: zie takenbundel

5 TO TEST OR NOT TO TEST

5.1 Getters en Setters

Je IDE kan getters en setters maken rond de private variabelen van je class. Deze gegenereerde code bevat geen fouten. Je moet deze code dus niet testen.

Als je de code van getters en setters uitbreidt, test je deze code wel.

5.2 equals en hashCode

Als twee objecten inhoudelijk gelijk zijn, moet het uitvoeren van de equals method op die objecten true teruggeven. Anders moet deze method false teruggeven.

Je voegt testen toe aan JaarTest:

```
@Test
public void equalsOpTweeDezelfdeJarenMoetTrueZijn() {
    assertEquals(new Jaar(2015), new Jaar(2015));
}

@Test
public void equalsOpTweeVerschillendeJarenMoetFalseZijn() {
    assertEquals(new Jaar(2015), new Jaar(2016));
}
```

Deze tests mislukken: je hebt zelf geen equals method in de class Jaar geschreven. Je voert dan de equals method uit die de class Jaar erft van de class Object. Die versie van de equals method is verkeerd: ze is niet gebaseerd op de *inhoud* van objecten, maar op de *adressen* van objecten.

Je voegt de method equals toe aan de class Jaar, die wel gebaseerd is op de inhoud:

```
@Override
public boolean equals(Object object) {
    if ( ! (object instanceof Jaar)) {
        return false;
    }
    Jaar anderJaar = (Jaar) object;
    return jaar == anderJaar.jaar;
}
```

Je voert de tests opnieuw uit. Deze lukken.

Als twee objecten inhoudelijk gelijk zijn, moet de returnwaarde van hun method hashCode gelijk zijn. Als twee objecten inhoudelijk verschillend zijn, is het wenselijk (maar niet verplicht) dat de returnwaarde van hun method hashCode verschillend is. Je kan dit laatste niet testen.

Je voegt een test toe aan JaarTest:

```
@Test
public void hashCodeOpTweeDezelfdeJarenMoetGelijkZijn() {
    assertEquals(new Jaar(2015).hashCode(), new Jaar(2015).hashCode());
}
```

Deze tests mislukken: je hebt zelf geen hashCode method in de class Jaar geschreven.

Je voert dan de hashCode method uit die de class Jaar erft van de class Object.

Die versie van de hashCode method is verkeerd: ze is niet gebaseerd op de *inhoud* van objecten, maar op de *adressen* van objecten.

Je voegt de method hashCode toe aan de class Jaar, die wel gebaseerd is op de inhoud:

```
@Override
public int hashCode() {
    return jaar;
}
```

Je voert de test opnieuw uit. Deze lukt.

5.3 Exceptions

Als je een verkeerde waarde meegeeft aan de constructor of een method, is deze correct als hij een exception werpt. Je test dit met JUnit.

De parameter van de method storten moet een positief getal zijn. Zoniet moet de method een `IllegalArgumentException` werpen.

Je schrijft eerst extra tests voor deze nieuwe vereiste:

```
@Test(expected = IllegalArgumentException.class)
public void hetBedragVanEenStortingMagNietNulZijn() {
    rekening.storten(BigDecimal.ZERO);
}

@Test(expected = IllegalArgumentException.class)
public void hetBedragVanEenStortingMagNietNegatiefZijn() {
    rekening.storten(BigDecimal.valueOf(-1));
}
```

①

②

- (1) Je test in deze `@Test` method een situatie die tot een `IllegalArgumentException` moet leiden: je probeert 0 € te storten bij (2).

Je vermeldt deze verwachte exceptie als de parameter `expected` van `@Test`.

Enkel als deze exception optreedt is deze test volgens JUnit geslaagd.

Je voert de test uit. De twee extra `@Test` methods mislukken.

Je moet dus de class `Rekening` aanpassen om aan deze extra vereiste te voldoen:

```
public void storten(BigDecimal bedrag) {
    if (bedrag.compareTo(BigDecimal.ZERO) <= 0) {
        throw new IllegalArgumentException("Bedrag moet positief zijn");
    }
    saldo = saldo.add(bedrag);
}
```

Je voert de unit test `RekeningTest` opnieuw uit. Alle tests slagen.

De code in de class `Rekening` voldoet dus aan de extra vereiste.

Er is nog een vereiste voor method `storten`:

als je `null` meegeeft als parameter, moet de method een `NullPointerException` werpen.

Je test dit met een extra method in `RekeningTest`:

```
@Test(expected = NullPointerException.class)
public void hetBedragVanEenStortingMagNietNullZijn() {
    rekening.storten(null);
}
```

Je voert de test uit. De test lukt. Je hoeft dus de class `Rekening` niet te wijzigen.

5.4 Grenswaarden en extreme waarden

5.4.1 Algemeen

Hoe meer tests, hoe beter de kwaliteit van de geteste code is.

Je moet ook grenswaarden testen:

- waarden die juist op de grens liggen van wat mag volgens de analyse
- waarden die juist boven de grens liggen van wat mag volgens de analyse
- waarden die juist onder de grens liggen van wat mag volgens de analyse

Je moet ook extreme waarden testen:

- de waarde `null` meegeven aan een parameter bij een method oproep
- een lege string meegeven aan een `String` parameter
- `Integer.MAX_VALUE` meegeven aan een `int` parameter

5.4.2 Voorbeeld: een class Rekeningnummer (van een bankrekening)

Rekeningnummer
-nummer: String
+Rekeningnummer(nummer: String) +toString(): String

- Het nummer moet 12 cijfers bevatten. Je test dus een nummer met 12 cijfers, een met 13 cijfers en een met 11 cijfers.
- De 4^e en 12^e positie bevat een - teken. Je test dus een nummer met streepjes en een zonder streepjes.
- Het getal, voorgesteld door de laatste 2 cijfers, moet gelijk zijn aan het getal, voorgesteld door de eerste 10 cijfers modulus 97. Je test dus een nummer waarbij deze berekening klopt en een waarbij deze berekening niet klopt.
- Als de eerste 10 cijfers modulus 97 0 is, moeten de laatste twee cijfers 97 zijn. Je test dit.

Als extreme waarden doe je ook een test waarbij je null meegeeft aan de constructor en een test waarbij je een lege string meegeeft aan de constructor.

5.4.2.1 De class

```
package be.vdab.valueobjects;
public class Rekeningnummer {
    public Rekeningnummer(String nummer) {
        throw new UnsupportedOperationException();
    }
    @Override
    public String toString() {
        throw new UnsupportedOperationException();
    }
}
```

5.4.2.2 Unit test

```
package be.vdab.valueobjects;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class RekeningnummerTest {
    @Test
    public void nummerMet12CijfersMetCorrectControleGetalIsOK() {
        new Rekeningnummer("063-1547564-61"); // dit mag geen exception veroorzaken
    }
    @Test
    public void alsRest0IsMoetControleGetal97Zijn() {
        new Rekeningnummer("063-1547503-97");
    }
    @Test(expected = IllegalArgumentException.class)
    public void nummerMet12CijfersMetVerkeerdeControleIsVerkeerd() {
        new Rekeningnummer("063-1547564-62");
    }
    @Test(expected = IllegalArgumentException.class)
    public void nummerMet12CijfersZonderStreepjesIsVerkeerd() {
        new Rekeningnummer("063154756461");
    }
    @Test(expected = IllegalArgumentException.class)
    public void nummerMet13CijfersIsVerkeerd() {
        new Rekeningnummer("063-1547564-623");
    }
    @Test(expected = IllegalArgumentException.class)
    public void nummerMet11CijfersIsVerkeerd() {
        new Rekeningnummer("063-1547564-6");
    }
}
```

```

@Test(expected = IllegalArgumentException.class)
public void leegNummerIsVerkeerd() {
    new Rekeningnummer("");
}
@Test(expected = NullPointerException.class)
public void nummerMetNullIsVerkeerd() {
    new Rekeningnummer(null);
}
@Test
public void toStringMoetHetNummerTeruggeven() {
    String nummer = "063-1547564-61";
    assertEquals(nummer, new Rekeningnummer(nummer).toString());
}
}

```

Je controleert als volgende stap of je aan de verleiding kon weerstaan om methods in de class Rekeningnummer te implementeren, door de unit test uit te voeren.

Je krijgt een rapport waarin alle tests mislukten als Errors. Dit is goed nieuws.

5.4.2.3 Eerste implementatie

```

package be.vdab.valueobjects;
public class Rekeningnummer {
    private static final Pattern REG_EXPR =
        Pattern.compile("^\\d{3}-\\d{7}-\\d{2}$");
    private final String nummer;
    public Rekeningnummer(String nummer) {
        if ( ! REG_EXPR.matcher(nummer).matches()) {
            throw new IllegalArgumentException("Verkeerd formaat");
        }
        long eerste10Cijfers =
            Long.parseLong(nummer.substring(0, 3) + nummer.substring(4, 11));
        long laatste2Cijfers = Long.parseLong(nummer.substring(12, 14));
        long rest = eerste10Cijfers % 97L;
        if (rest == 0) {
            if (laatste2Cijfers != 97) {
                throw new IllegalArgumentException("Verkeerd controlegetal");
            }
        } else {
            if (rest != laatste2Cijfers) {
                throw new IllegalArgumentException("Verkeerd controlegetal");
            }
        }
        this.nummer = nummer;
    }
    @Override
    public String toString() {
        return nummer;
    }
}

```

- (1) Deze String stelt een regular expression (tekstpatroon) voor.
 In een regular expression hebben bepaalde tekens een speciale betekenis:
 - ^ het begin van de tekst
 - \$ het einde van de tekst
 - \\d een cijfer
 - {x} x aantal keer het vorige teken uit de regular expression.
 We leggen nu de regular expression uit, die een rekeningnummer voorstelt:
 - ^ het begin van het rekeningnummer
 - \\d{3} daarna komen 3 cijfers

- daarna komt letterlijk het teken -
 - \\d{7} daarna komen 7 cijfers
 - daarna komt letterlijk het teken -
 - \\d{2} daarna komen 2 cijfers
 - \$ het einde van het rekeningnummer
- Opmerking: in een Java string stel je één \ voor als \\.
- (2) De method matches geeft enkel true terug als de String die je meegeeft aan de method matcher past bij de regular expression in het Pattern object.

Je kan de regular expression grafisch voorstellen op <http://jex.im/regulex>



Je voert RekeningnummerTest uit en alle tests slagen.

Dit betekent dat de class Rekeningnummer voldoet aan de vereisten van de analyse.

5.4.2.4 Refactoring

Je ziet na of je refactoring kan toepassen. Dit is niet het geval. Rekeningnummer is dus af.

5.4.3 Voorbeeld: een verzameling

Als je een method test die een parameter met een verzameling (array, List, ...) bevat, moet je ook volgende grenswaarden en extreme waarden testen:

- een verzameling met één element
- een lege verzameling
- in de plaats van een verzameling de waarde null meegeven
- een verzameling meegeven met null als element

Je geeft aan de method
getGemiddelde
een BigDecimal array mee.
Je krijgt het gemiddelde
van deze BigDecimals terug

Statistiek
<u>+getGemiddelde(getallen: BigDecimal[*]): BigDecimal</u>

5.4.3.1 De class

```
package be.vdab.util;
import java.math.BigDecimal;
public class Statistiek {
    public static BigDecimal getGemiddelde(BigDecimal[] getallen) {
        throw new UnsupportedOperationException();
    }
}
```

5.4.3.2 De unit test

```
package be.vdab.util;
import static org.junit.Assert.assertEquals;
import java.math.BigDecimal;
import org.junit.Test;
public class StatistiekTest {
    @Test
    public void hetGemiddeldeVan0en10is5() {
        assertEquals(0, BigDecimal.valueOf(5).compareTo(Statistiek.getGemiddelde(
            new BigDecimal[] { BigDecimal.ZERO, BigDecimal.TEN })));
    }
}
```

```

@Test
public void hetGemiddeldeVanEenGetalIsDatGetal() {
    BigDecimal enigGetal = BigDecimal.valueOf(1.23);
    assertEquals(0, enigGetal.compareTo(
        Statistiek.gemiddelde(new BigDecimal[] { enigGetal })));
}
@Test(expected = IllegalArgumentException.class)
public void hetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen() {
    Statistiek.gemiddelde(new BigDecimal[] { });
}
@Test(expected = NullPointerException.class)
public void hetGemiddeldeVanNullKanJeNietBerekenen() {
    Statistiek.gemiddelde(null);
}
@Test(expected = NullPointerException.class)
public void eenArrayElementMagNietNullBevatten() {
    Statistiek.gemiddelde(new BigDecimal[] { null });
}
}

```

Je controleert als volgende stap of je aan de verleiding kon weerstaan om methods in de class Statistiek te implementeren, door de unit test uit te voeren. Je krijgt een rapport waarin alle tests mislukten als Errors. Dit is goed nieuws.

5.4.3.3 Eerste implementatie

```

package be.vdab.util;
import java.math.BigDecimal;
public class Statistiek {
    public static BigDecimal gemiddelde(BigDecimal[] getallen) {
        if (getallen.length == 0) {
            throw new IllegalArgumentException("Lege array");
        }
        return Arrays.stream(getallen)
            .reduce(BigDecimal.ZERO, (vorigTotaal, getal) -> vorigTotaal.add(getal))
            .divide(BigDecimal.valueOf(getallen.length));
    }
}

```

Je commit de sources en je publiceert op GitHub.

Je voert de bijbehorende unit test (StatistiekTest) uit. Alle tests slagen.

Dit betekent dat de class Statistiek voldoet aan de vereisten van de analyse.

5.4.3.4 Refactoring

Je ziet na of je refactoring kan toepassen. Dit is niet het geval. Statistiek is dus af.



De uitdaging van unit testen is te bedenken *wat* er moet getest worden. Enerzijds zijn dit positieve dingen: dingen die moeten werken. Anderzijds zijn dit negatieve dingen: dingen die niet mogen werken en die tot een exception moeten leiden.



ISBN: zie takenbundel

6 CODE COVERAGE

Een code coverage tool toont je welke stukken code je wél getest hebt en welke niet.

Dit is interessant: als je stukken code hebt die nog niet getest zijn, en het gaat om code die je zelf geschreven hebt, kan je beter nog wat testen bijschrijven voor die stukken niet geteste code.

Er bestaan meerdere code coverage tools voor Java. Je gebruikt in deze cursus EclEmma.

6.1 EclEmma gebruiken

Je plaatst in de class `StatistiekTest` de method `hetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen` even in commentaar.

Je klikt met de rechtermuisknop op `StatistiekTest` in de Project Explorer en je kiest `Coverage As, Junit Test`.

Je opent de class die getest werd: `Statistiek`.

- De code met een groene achtergrond werd door je getest.
- De code met een rode achtergrond werd niet getest.
- De code met een gele achtergrond werd gedeeltelijk getest.



Opmerking: ook de code in `StatistiekTest` heeft nu een mengeling van groene en rode achtergrondkleuren. Deze hebben hier echter geen betekenis.

Je haalt in de class `StatistiekTest` de method `hetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen` terug uit commentaar.


Je klikt met de rechtermuisknop op `StatistiekTest` in de Project Explorer en je kiest `Coverage As, Junit Test`.

Alle code binnen de class is nu groen en dus getest. De class zelf heeft een rode achtergrond, maar dit heeft geen betekenis.



Opmerking: Je zal zelden 100% (alle code groen) coverage bereiken. Je class bevat snel wat code die je niet test, bijvoorbeeld omdat je ze liet genereren door je IDE.

Je kan alle code terug een witte achtergrond geven met volgende stappen:

1. Je kiest in het menu `Window` de opdracht `Show View, Other`
2. Je kiest in de categorie `Java` voor `Coverage`
3. Je kiest in de knoppenbalk van het tabblad `Coverage` de knop  (`Remove All Sessions`)

7 MEERDERE UNIT TESTS UITVOEREN

7.1 Alle unit tests van één package uitvoeren

Je klikt in de Project Explorer met de rechtermuisknop op één van de packages in `src/test/java` en je kiest `Run As, JUnit Test`.

Als je dit doet op de package `be.vdab.valueobjects`, zie je dat JUnit alle unit tests in die package uitvoert: `VeilingTest`, `RekeningnummerTest`, `JaarTest`, `WoordTest`.

7.2 Alle unit tests van het project uitvoeren

Je klikt in de Project Explorer met de rechtermuisknop op `src/test/java` en je kiest `Run As, JUnit Test`.

7.3 Test suite

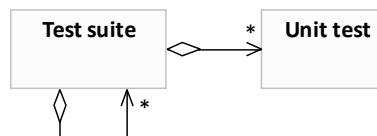
Een test suite is een class waarin je unit tests verzamelt.

Als JUnit de test suite uitvoert, voert JUnit alle unit tests in die test suite uit.

Je kan in een test suite de unit tests `RekeningTest` en `RekeningnummerTest` verzamelen.

Als JUnit deze test suite uitvoert, voert hij beide unit tests uit.

Je kan in een test suite niet enkel unit tests verzamelen, maar ook andere test suites. Dit geeft je grote flexibiliteit.



Je maakt een test suite die een verzameling is van `RekeningTest` en `RekeningnummerTest`.

Je maakt in `src/test/java` een package `be.vdab.testsuites`. Je kan deze test suite niet in een bestaande packages opnemen: deze test suite bevat unit tests uit verschillende packages.

Je maakt in die package de test suite:

1. Je klikt met de rechtermuisknop op de package `be.vdab.testsuites`.
2. Je kiest `New, Other; Java, JUnit, JUnit Test Suite` en je kiest `Next`.
3. Je tikt de test suite naam bij `Name: AllesMetRekeningenTest`
4. Als de huidige packages unit tests of test suites zou bevatten, zou je die zien in `Test classes to include in the suite` en zou je kunnen aanvinken om ze toe te voegen aan de huidige test suite. Dit is niet het geval. Je kiest `Finish`.

Je ziet dat een test suite een class is. Je past deze class aan.

```

package be.vdab.testsuites;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
import be.vdab.entities.RekeningTest;
import be.vdab.valueobjects.RekeningnummerTest;
@RunWith(Suite.class)
@SuiteClasses({RekeningnummerTest.class,
               RekeningTest.class})
public class AllesMetRekeningenTest {
}
  
```

①
②

- (1) Je tikt voor de class die een test suite voorstelt `@RunWith(Suite.class)`.
- (2) Je tikt voor de class die een test suite voorstelt ook `@SuiteClasses`
Je geeft een array mee met de unit tests (of test suites) die je verzamelt in de huidige test suite.

Je commit de sources en je publiceert op GitHub.

Je voert de test suite uit:

1. Je klikt in de Project Explorer met de rechtermuisknop op `AllesMetRekeningenTest`.
2. Je kiest `Run As, JUnit Test`.

8 DEPENDENCIES

Als een class A methods oproept van een class B, heeft de class A een dependency (“afhankelijkheid”) op de class B. De class A kan niet functioneren zonder de class B.

Je leert in dit hoofdstuk hoe je de problemen oplost bij het testen van zo’n classes.

- Deze problemen stellen zich niet als de classes entities of value objecten (concepten uit de werkelijkheid) voorstellen.
- Deze problemen stellen zich wel als de classes technische classes zijn (servlets, service classes, repository classes).

Technische classes zijn meestal thread-safe.

Je moet dan niet in elke method van de class A een instance maken van de class B.

Je kan één instance van de class B in een private variabele in de class A bijhouden.

Je gebruikt deze instance in alle methods van de class A.

Je maakt deze private variabele ook best `final`. Je verhindert zo dat je (per ongeluk) de variabele na het initialiseren nog wijzigt (bvb. op `null` plaatst).

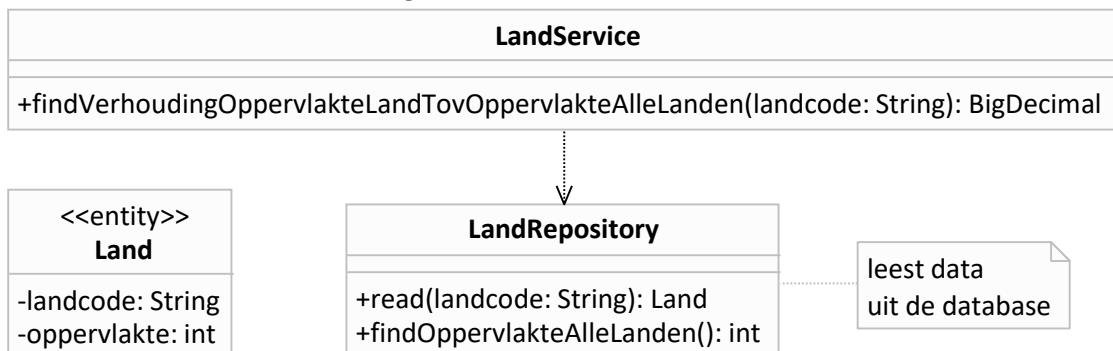
```
public class A {
    private final B b = new B();
    public void a1() {
        b.b1();
    }
    public int a2() {
        b.b1();
        b.b2();
    }
}
```

Je drukt deze dependency in een class diagram uit met een pijl, met gestippelde lijn, van de class A naar de class B.



8.1 Voorbeeld

Je zal de problemen bij het testen van een technische class met dependencies leren kennen aan de hand van volgend voorbeeld:



De class `LandService` heeft een dependency op de class `LandRepository`.

De method `findVerhoudingOppervlakteLandTovOppervlakteAlleLanden` geeft een `BigDecimal` terug met de verhouding van de oppervlakte van één land ten opzichte van de oppervlakte van alle landen. Je geeft de `landcode` van dit land mee als parameter.

Je zal in deze method de `LandRepository` method `read` oproepen.

Je krijgt de detailinformatie van dat land (waaronder de oppervlakte) terug.

Je zal in deze method ook de `LandRepository` method `findOppervlakteAlleLanden` oproepen.

Je krijgt de totale oppervlakte van alle landen terug.

Je zal daarna de oppervlakte van het ene land delen door de oppervlakte van alle landen.

Je maakt in de package `be.vdab.entities` de class `Land`:

```
package be.vdab.entities;
public class Land {
    private final String landcode;
    private final int oppervlakte; // je maakt getters voor de private variabelen
    public Land(String landcode, int oppervlakte) {
        this.landcode = landcode;
        this.oppervlakte = oppervlakte;
    }
}
```

Je maakt een package `be.vdab.services`. Je maakt daarin de class `LandService`. Deze code zal voorlopig fouten bevatten, omdat `LandRepository` nog niet bestaat.

```
package be.vdab.services;
import java.math.BigDecimal;
import be.vdab.repositories.LandRepository;
import be.vdab.entities.Land;
public class LandService {
    private final LandRepository landRepository = new LandRepository();
    public BigDecimal findVerhoudingOppervlakteLandTovOppervlakteAlleLanden(
        String landcode) {
        throw new UnsupportedOperationException();
    }
}
```

8.2 Problemen

Er zijn drie problemen bij het testen van de class `LandService`:

- ⊖ In een groot team kan elke programmeur zijn “specialiteit” hebben.
“front-end” programmeurs zijn gespecialiseerd in user interface code: servlets, JSP
“back-end” programmeurs zijn gespecialiseerd in repository code en services code.
Het is dus mogelijk dat jij `LandService` schrijft en Jan `LandRepository`.
Jij kan `LandService` niet schrijven zolang Jan de class `LandRepository` niet afwerkt.
Als Jan verlof heeft, ziek is, ander werk heeft, ... kan jij niet verder werken.
- ⊖ Jij test `LandService` met een unit test.
Deze voert indirect ook code uit van de `LandRepository` methods.
Jij wordt dan geconfronteerd met fouten in `LandRepository` methods.
Dit is niet de bedoeling: jij wil fouten opsporen in jouw code, niet in die van Jan.
Zelfs als jij `LandService` én `LandRepository` schrijft, wil je bij een unit test van `LandService` niet geconfronteerd worden met fouten in `LandRepository`.
- ⊖ Je test `LandService` met een unit test.
Deze test voert indirect ook code uit van de `LandRepository` methods.
Gezien deze methods de database aanspreken, loopt de unit test traag.
Dit maakt test driven development vervelend: je wilt snelle feedback.
Dit geldt ook als jij zowel de class `LandService` als de class `LandRepository` schrijft.

8.3 De dependency als een interface

Stap 1: je drukt de functionaliteit van de dependency uit in een interface.

Je maakt in `src/main/java` een package `be.vdab.repositories` en daarin `LandRepository`:

```
package be.vdab.repositories;
import be.vdab.entities.Land;
public interface LandRepository {
    Land read(String landcode);
    int findOppervlakteAlleLanden();
}
```

```
<<interface>>
LandDAO
+read(landcode: String): Land
+findOppervlakteAlleLanden(): int
```

Jij kan deze interface ontwerpen, of Jan, of jullie analyst, ...

Stap 2: je gebruikt deze interface in elke class die zo'n dependency heeft:

```
...
public class LandService {
private final LandRepository landRepository = new LandRepository();
    private final LandRepository landRepository;
    ...
}
```

8.4 Dependency injection

In deze nieuwe versie is de variabele `landRepository` niet geïnitieerd.

Je lost dit probleem op met een constructor in dezelfde class:

```
...
public class LandService {
    private final LandRepository landRepository;
    public LandService(LandRepository landRepository) {
        this.landRepository = landRepository;
    }
    ...
}
```

Je geeft aan de constructor een object mee waarvan de class `LandRepository` implementeert.

De private variabele `landRepository` wijst daarna naar dit object.

De oproepen `landRepository.read(landcode);`

en `landRepository.findOppervlakteAlleLanden();`

verder in `LandService`, gebeuren op dit object.



De class `LandService` maakt dus zelf geen object meer dat de dependency voorstelt, maar krijgt dit object aangereikt (als constructor parameter).

Dit heet "dependency injection".



Opmerking: een class kan meerdere dependencies hebben.

De class bevat dan per dependency een extra private final variabele en een extra parameter in de constructor.

9 STUB

Je maakt in de unit test voor de class `LandService` een `LandService` object.

Je roept hierbij de `LandService` constructor op.

Je moet een object meegeven waarvan de class `LandRepository` implementeert.

De class van Jan zal die interface implementeren.

Jan zal deze class bijvoorbeeld `LandRepositoryJPA` noemen (omdat hij JPA gebruikt).

Maar jij gebruikt in `LandServiceTest` geen object van deze class.

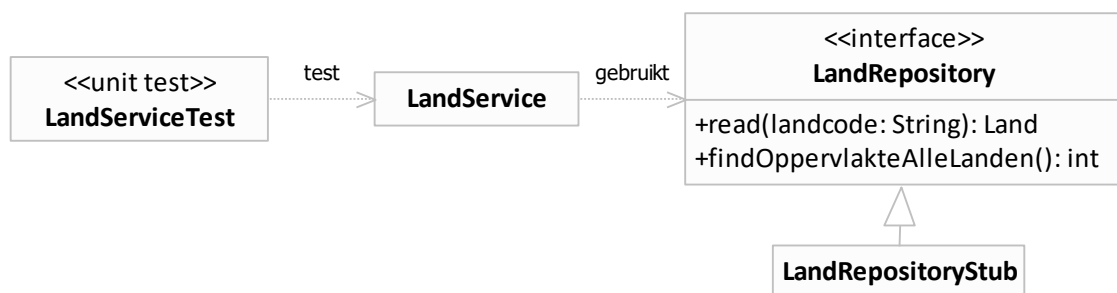
Anders heb je de problemen beschreven in het vorige hoofdstuk.

Je gebruikt in de plaats een stub: een object van een class die ook `LandRepository` implementeert.

De stub code is minimaal: ze zorgt er enkel voor dat jij `LandServiceTest` kan schrijven.

De stub class heet bijvoorbeeld `LandRepositoryStub`

9.1 Classes en interfaces gebruikt in de test



Een stub lijkt op een crash test dummy die ontwerpers gebruiken bij een auto test.



9.2 Classes en interfaces gebruikt in productiecode



9.3 Praktisch

Je maakt stubs in `src/test/java`: je gebruikt stubs enkel in tests.

Je maakt in `src/test/java` een package `be.vdab.repositories` en daarin de class `LandRepositoryStub`:

```
package be.vdab.repositories;
import be.vdab.entities.Land;
public class LandRepositoryStub implements LandRepository {
    @Override public Land read(String landcode) {
        return new Land(landcode, 5);
    }
    @Override public int findOppervlakteAlleLanden() {
        return 20;
    }
}
```

- (1) De stub geeft een `Land` object terug met de gevraagde `landcode` en een fictieve oppervlakte (5).
- (2) De stub geeft een fictieve totale oppervlakte terug (20).

Je maakt in `src/test/java` de package `be.vdab.services`, met de class `LandServiceTest`:

```
package be.vdab.services;
import static org.junit.Assert.assertEquals;
import java.math.BigDecimal;
import org.junit.Before;
import org.junit.Test;
import be.vdab.repositories.LandRepository;
import be.vdab.repositories.LandRepositoryStub;
public class LandServiceTest {
    private LandRepository landRepository;
    private LandService landService;
    @Before public void before() {
        landRepository = new LandRepositoryStub();
        landService = new LandService(landRepository);
    }
    @Test public void findVerhoudingOppervlakteLandTovOppervlakteAlleLanden() {
        assertEquals(0, BigDecimal.valueOf(0.25).compareTo(
            landService.findVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B")));
    }
}
```

- (1) Je maakt een stub.
- (2) Je geeft die mee aan de constructor van de te testen class (dependency injection).
- (3) Op basis van de data in de stub moet de verhouding 0.25 zijn.

Je voert de unit test uit. Deze mislukt, omdat `LandService` nog geen echte code bevat.

Je schrijft nu echte code in de method

`findVerhoudingOppervlakteLandTovOppervlakteAlleLanden` in de class `LandService`:

```
public BigDecimal findVerhoudingOppervlakteLandTovOppervlakteAlleLanden(
    String landcode) {
    Land land = landRepository.read(landcode);
    int oppervlakteAlleLanden = landRepository.findOppervlakteAlleLanden();
    return new BigDecimal(land.getOppervlakte())
        .divide(BigDecimal.valueOf(oppervlakteAlleLanden));
}
```

Je commit de sources en je publiceert op GitHub. Je voert de unit test opnieuw uit. Deze lukt.

Je kan dus `LandService` schrijven én testen, terwijl `LandRepositoryJPA` nog niet geschreven is.



Stub: zie takenbundel

10 MOCK

10.1 Algemeen

Een mock is een stub met twee extra eigenschappen:

- Variabel resultaat van method oproep
- Verificaties

10.1.1 Variabel resultaat van method oproep

Elke method oproep van een stub geeft eenzelfde resultaat.

Als je op de `LandRepositoryStub` de method `readLand("")` oproept, krijg je een `Land` object terug. Bij bepaalde testen zou het interessant zijn dat je dan `null` terugkrijgt.

Het resultaat van een method oproep van een mock kan variëren, afhankelijk van de parameterwaarden die je meegeeft bij de method oproep.

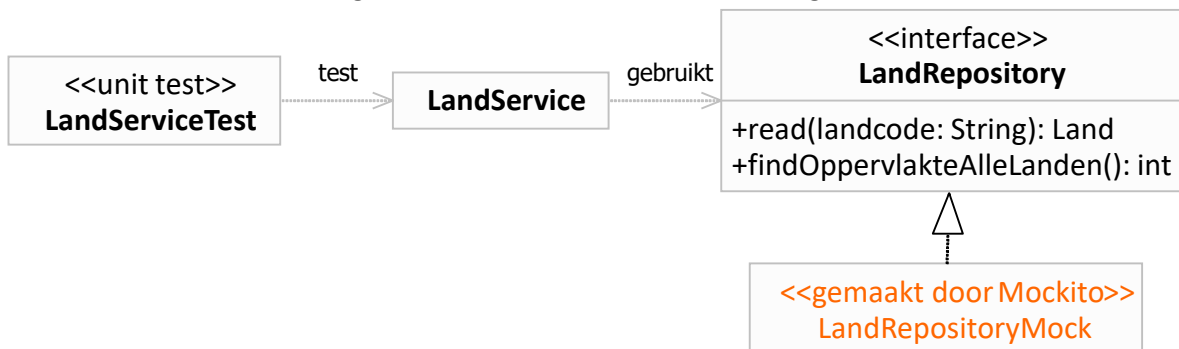
10.1.2 Verificaties

Nadat je in een `@Test` method een method hebt opgeroepen van de te testen class, kan je bij een mock verifiëren of deze class zijn dependency aangesproken heeft

```
public void findVerhoudingOppervlakteLandTovOppervlakteAlleLanden() {
    assertEquals(0, BigDecimal.valueOf(0.25).compareTo(
        landService.findVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B")));
    // hier gaan we straks verifiëren of landService de methods read("B")
    // en findOppervlakteAlleLanden() heeft opgeroepen op landRepository.
}
```

10.2 Mockito

Een mock class zelf schrijven is tijdrovend. Je maakt een mock met een mocking library. Er bestaan meerdere mocking libraries in Java. Mockito wordt veel gebruikt.



10.2.1 pom.xml

Je voegt de Mockito dependency toe aan `pom.xml`:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>[1.10, 2.0.0-beta)</version>
  <scope>test</scope>
</dependency>
```



- (1) Je plaatst de scope op `test`.

Op die manier zijn de classes van Mockito ter beschikking gedurende het ontwikkelen van je applicatie, maar neemt Maven de classes van Mockito niet op in het eindproduct van je applicatie. Bij een console applicatie is dit eindproduct een JAR bestand, bij een website is dit een WAR bestand.

10.2.2 Mock aanmaken

Je vervangt in LandServiceTest de stub door een mock, aangemaakt door Mockito.

Je voegt enkele `import static` methods toe aan LandServiceTest:

```
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
```

①

(1) Je zal static methods van de class Mockito oproepen.

Deze oproepen zijn kort (en dus leesbaar) met deze `import static`.

Je tikt juist voor de class

```
@RunWith(MockitoJUnitRunner.class)
```

①

Je vraagt JUnit de test uit te voeren *in samenwerking met* Mockito.

Je tikt `@Mock` voor de private variabele `landRepository`.

Mockito maakt dan “at runtime” een class die de interface `LandRepository` (het type van de variabele `landRepository`) implementeert. Mockito maakt een object van deze class en verwijst de variabele `landRepository` naar dit object.

10.2.3 Mock trainen

De method `mock` heeft een class gemaakt die de interface `LandRepository` implementeert.

Deze “mock” class bevat dus de methods beschreven in deze interface.

Deze method implementaties zijn zeer eenvoudig. Je ziet hier onder welke waarde ze teruggeven. Deze hangt af van het returntype van de method in de interface.

Interface method returntype	Returnwaarde door method in de mock
<code>boolean</code>	<code>false</code>
<code>byte</code> , <code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>	<code>0 (nul)</code>
Een class of interface	<code>null</code>
<code>void</code>	niets

De “mock” class die de interface `LandRepository` implementeert bevat dus

- de method `read`, die `null` teruggeeft
- de method `findOppervlakteAlleLanden`, die `0 (nul)` teruggeeft

Deze waarden zijn zelden bruikbaar in je unit test.

Als je bijvoorbeeld `LandServiceTest` uitvoert, krijg je een `NullPointerException`.

Gelukkig kan je de mock trainen.

Je geeft hierbij een waarde aan die een method van de “mock” class moet teruggeven.

Je doet dit met extra opdrachten vooraan in de method `before`:

```
when(landRepository.findOppervlakteAlleLanden()).thenReturn(20);
when(landRepository.read("B")).thenReturn(new Land("B", 5));
```

①

②

(1) Je traint een mock met de static Mockito method `when`.

Je geeft als parameter de variabele mee die naar de mock wijst, gevolgd door de method van die mock.

De method `when` geeft je een `OnGoingStubbing` object terug.

Je roept daarop de method `thenReturn` op.

Je geeft de waarde mee die de mock method moet teruggeven.

De mock zal dus de waarde `20` teruggeven

als je er de method `findOppervlakteAlleLanden` op uitvoert.

(2) Je traint de mock zodat hij een `Land` object met een landcode “B” en een oppervlakte 5 teruggeeft als je op de mock de method `read` oproept met een parameter “B”.

Als je de unit test nu uitvoert, slaagt de test.



Opmerking 1: je kan de mock trainen, zodat zijn methods verschillende waarden teruggeven, afhankelijk van de parameterwaarden die ze binnenkrijgen.

Als je in de method before de opdracht

```
when(landRepository.read("NL")).thenReturn(new Land("NL", 6));
```

toevoegt geeft de method read een ander Land object terug, naargelang je de landcode "B" of "NL" meegeeft als parameter.



Opmerking 2: je kan de mock trainen, zodat zijn methods een exception werpen als ze een bepaalde parameterwaarde binnenkrijgen:

```
when(landRepository.read(""))
    .thenThrow(new IllegalArgumentException());
```

10.2.4 Verificaties

Je kan verifiëren of de te testen class zijn dependencies heeft opgeroepen tijdens het uitvoeren van zijn methods.

De LandService method findVerhoudingOppervlakteLandTovOppervlakteAlleLanden moet op zijn LandRepository dependency de methods findOppervlakteAlleLanden en read("B") oproepen. Zoniet is deze method verkeerd geschreven.

Deze verificaties zijn ingebouwd in de mocks die Mockito aanmaakt.

Je voegt opdrachten toe als laatste in de @Test method findVerhoudingOppervlakteLandTovOppervlakteAlleLanden:

```
verify(landRepository).findOppervlakteAlleLanden();
verify(landRepository).read("B");
```

①

(1) Je verifieert met de static Mockito method verify.

Je geeft als parameter de mock mee. Je krijgt een object terug. Je geeft hierop aan welke method oproep moet gebeurd zijn. Als dit niet het geval is, mislukt de test.

Je commit de sources en je publiceert op GitHub.

Je ziet hier onder een applicatie met een class (groen), zijn dependencies (blauw) en de dependencies (grijs) van die dependencies:



Je ziet hier dezelfde class (groen) in een test. De dependencies zijn vervangen door mocks (geel):



Mock: zie takenbundel

10.3 Getuigenis op het einde van deze cursus

Hopelijk groeit bij u hetzelfde TDD enthousiasme als bij volgende ontwikkelaar:

TDD is fun! It's like a game where you navigate a maze of technical decisions that lead to highly robust software while avoiding the quagmire of long debug sessions. With each test there is a renewed sense of accomplishment and clear progress toward the goal. Automated tests record assumptions, capture decisions, and free the mind to focus on the next challenge.

11 HERHALINGSOEFENING



WoordTeller: zie takenbundel

12 STAPPENPLAN

1. Analyseer de class.
2. Schrijf de class met zijn methods.
Elke method bevat één method: `throw new UnsupportedOperationException();`
3. Schrijf de tests voor de class.
Schrijf tests voor dingen die moeten lukken én tests voor dingen die moeten mislukken.
4. Voer de tests uit. Die moeten allen mislukken.
5. Implementeer de class.
6. Voer de tests uit.
7. Als er tests mislukken, wijzig je de code in de class en ga je terug naar 6.
8. Refactor de class.
 - a. Kan je de code korter schrijven ?
 - b. Kan je de code duidelijker maken ?
 - c. Kan je de code performanter maken ?
 - d. Kan je minder geheugen gebruiken ?Ga naar 6.
9. Voer code coverage uit.
Als er code in de class niet uitgevoerd wordt door je test, pas je de test of de code in de class aan. Ga naar 6.

13 COLOFON

Domeinexpertisemanager:	Jean Smits
Moduleverantwoordelijke:	Jean Smits
Medewerkers:	Hans Desmet
Versie	2/5/2018