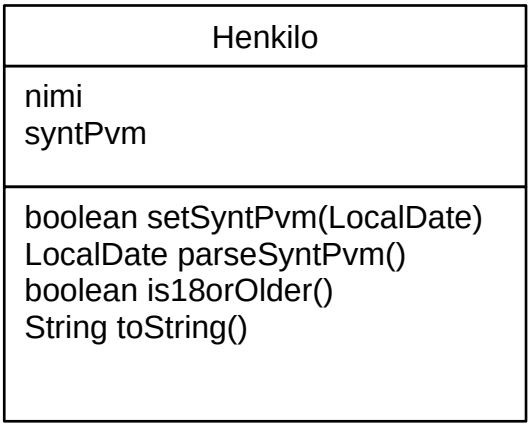
Junit
Sauli Isonikkilä

Yksikkötestaus - JUnit

Yksikkötestaus eli Unit Test kuuluu ohjelmistokehittäjän tehtäviin. Laajemman testauksen olisi hyvä tehdä muut siihen erikoistuneet testaajat. Kun varsinaisen testauksen esim. integrointitestauksen, tekee erit henkilöt, tehdään testausta eri ”silmin”. Yksikkötestaus kuitenkin kuuluu kehittäjän tehtäviin. Kehittäjä siis testaa itse tekemäänsä ohjelmakoodin. Testaus asetelma on siis erilainen – ohjelmoija itse tietää, miten ohjelmakoodin pitää toimia. Ohjelmoija voi helposti tehdä testejä, joissa kaikki toimii. Ja hän varoo tekemästä testejä, joita ei ole otettu huomioon ohjelmaa tehtäessä. Testauksessa on kuitenkin juuri etsittävä mahdollisia ongelma tilanteita. Juuri tehdyssä ohjelmassa voi olla loogisia virheitä, käyttäjien syöttämiä datoja eri tilanteissa ei olla otettu huomioon. Käyttäjä voi kirjoittaa lisättäväksi dataksi tekstiä, vaikka pyydetään numeroita. Käyttäjä voi epähuomiossa kirjoittaa desimaalinumeron desimaalieroittimeksi pisteen vaikka pitäisi käyttää pilkkua. Myös tuotteen hinnaksi voi tulla annettua negatiivinen numero, vaikka sen pitäisi olla positiivinen numero. Tietysti myös syntaksi virheitä voi vielä löytyä, vaikka tässä vaiheessa ohjelman debuggaus olisi pitänyt olla jo tehtynä. On siis hyödyllistä muuttaa ohjelmakehittäjän roolista vaikkapa käyttäjän rooliin, joka haluaa kaataa juuri tekemäsi ohjelman.

Ohjelmakehittäjän kannattaa aina miettiä heti alussa, miten tehtävä ohjelma kannattaa testata. Vasta sen jälkeen voi ryhtyä tekemään itse ohjelmaa. Samalla voi aina testata, meneekö uusi koodi testeistä läpi. Tämä olisi hyvä lähetymistapa. Esimerkiksi jos tehtävänäsi on tehdä Henkilö -luokka tekeillä olevaan henkilöstöhallinnan järjestelmään, heti alkuun on hyvä miettiä, miten Henkilö -luokan testaisit. Kuvassa 1 alla on esitetty Henkilö -luokka. Nimi muuttujan tietotyyppi tulee olemaan String, koska siihen pitää pystyä tallentamaan (sijoittamaan) tekstiä. SyntPvm muuttujaan pitää pystyä sijoittamaan päivämääriä, joten sen tietyyppi voisi olla Date tai LocalDate. Tässä valitaan sen tietotyypiksi LocalDate. Tämän lisäksi nimi ja syntPvm tulevat olemaan private:ja. Alla olevassa kuvassa olevat metodit taas ovat public:ja. SetSyntPvm -metodissa asetetaan henkilön syntymäpäivämäärä. Samalla tehdään tarkistus siitä, että kyseessä on syntymäpäivämäärä. SetSyntPvm -metodissa on siis testattavaa. Toimiiko se, jos annetaan jotain sellaista, mikä ei ole päivämäärä. Esimerkiksi 35.13.2002 tai jotain muuta vastaavaa. Päivämäärä datan pitäisi antaa päivämäärä formaattia noudattaen, esim pv.kk.vvvv tai pv-kk-vvvv. Lisäksi olisi hyvä, jos annetun syntymäpäivämäärän perusteella voisi laskea henkilön sen hetkisen iän vuosina.

Junit
Sauli Isonikkilä



Kuva 1. Henkilö -luokka ja sen keskeiset ominaisuudet (properties) ja metodit.

Henkilö -luokkaan tulee tietysti myös oletusmuodostin, parametrillinen muodostin ja loput niin sanotut setterit ja getterit nimi ja syntPvm muuttujille. Tämän pohjalta voimme seuraavaksi luoda Henkilö -luokan. Alla olevassa esimerkki 1:ssä on esitetty ohjelmakoodi:

Esimerkki 1: Henkilö -luokka.

```
package henkilo;

import java.time.LocalDate;
import java.time.Period;

public class Henkilo {

	private String nimi;
	private LocalDate syntPvm;

	public Henkilo() {
		// TODO Auto-generated constructor stub
	}

	public Henkilo(String nimi, String syntPvm) {
		this.nimi = nimi;
		this.syntPvm = parseSyntPvm(syntPvm);
	}

	private LocalDate parseSyntPvm(String syntPvm) {
		int vvvv = Integer.parseInt(syntPvm.substring(0,4));
		int kk = Integer.parseInt(syntPvm.substring(5,7));
```

```
            int pv = Integer.parseInt(syntPvm.substring(8,10));
            return LocalDate.of(vvvv, kk, pv);
      }

      public int getIka() {
            Period ika = Period.between(syntPvm, LocalDate.now());
            return ika.getYears();
      }

      public boolean is18orOlder() {
            return getIka() >= 18;
      }

      public String getNimi() {
            return nimi;
      }

      public void setNimi(String nimi) {
            this.nimi = nimi;
      }

      public LocalDate getSyntPvm() {
            return syntPvm;
      }

      public void setSyntPvm(String syntPvm) {
            this.syntPvm = parseSyntPvm(syntPvm);
      }

      @Override
      public String toString() {
            return "Henkilo [nimi=" + nimi + ", syntPvm=" + syntPvm + "]";
      }
}
```

Yllä olevista metodeista potenttiaalisia testaus kohteita ovat kaikki public metodit. Niistä setSyntPvm ja toString sisältävät monimutkaisempaa koodia, jossa on jotain mitä testata. Muissa metodeissa on vain sijoituslause tai pelkkä return lause. Luomme siis testitapaukset setSyntPvm metodille alla olevassa ohjelmakoodissa. Siinä on kaksi testiä setSyntPvm metodille. Ensimmäisessä `testGetIka()` metodissa testataan, että kun Henkilo luokasta luotuun olioon sijoitetaan (talletetaan) data-arvo kokonaislukuna, niin saadaan palautettua oikein laskettu ikä vuosina metodin ajohetki huomioon ottaen.

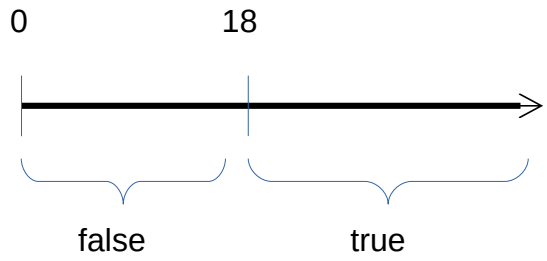
Junit
Sauli Isonikkilä

`Assert.assertTrue` -metodi olettaa, että sen argumentiksi annettu ehto `h1.getIka() == ikaToTest` on tosi (true). TestGetIka -metodissa halutaan testata, toimiiko getIka metodi oikein, jos syntymäpäivämääräksi on annettu ajohetkellä 30 vuotiaan henkilön syntymäpäivämäärä. Tämä tietysti muuttuu ajan kuluessa. Nyt 30 vuotias henkilö täyttää jossain vaiheessa 31 vuotta. Sen vuoksi tarvitaan tämän testaamiseen apumetodia `createHenkiloWithIka(ikaToTest.` Sen avulla on mahdollista testata 30 vuotiaan syntymäpäivämäärää testitapauksena ens. vuonnakin, jos silloinkin tarvitaan tätä testitapausta.

```
package henkilo;

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

import java.time.LocalDate;

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

class HenkiloTest {

	@Test
	void testGetIka() {
		int ikaToTest = 30;
		Henkilo h1 = createHenkiloWithIka(ikaToTest);
		Assert.assertTrue(h1.getIka() == ikaToTest);
	}

	private Henkilo createHenkiloWithIka(int ikaToTest) {
		int currentYear = LocalDate.now().getYear();
		int syntVuosi = currentYear - ikaToTest;
		String syntPvm = syntVuosi + "-01-01";
		Henkilo h1 = new Henkilo("Matti", syntPvm);
		return h1;
	}

	@Test
	void testIs17YearOldIsNot18OrOlder() {
		Henkilo h1 = createHenkiloWithIka(17);
		Assert.assertFalse(h1.is18orOlder());
	}
```

```
	@Test
	void testIs18YearOldIs18orOlder() {
		Henkilo h1 = createHenkiloWithIka(18);
		Assert.assertTrue(h1.is18orOlder());
	}


	@Test
	void testIs30YearOldIs18orOlder() {
		Henkilo h1 = createHenkiloWithIka(30);
		Assert.assertTrue(h1.is18orOlder());
	}

}
```

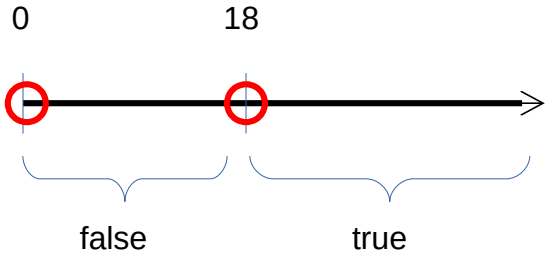
Seuraavaksi on testattu is18orOlder -metodi. Tässä metodissa on tärkeää selvittä testeissä, että sen avulla saadaan selville täysikäisyys. Jos siis henkilö on 17 vuotias, pitää is18orOlder -metodin palauttaa vastauksenaan `false`. Jos henkilö on 18 vuotias tai sitä vanhempi, pitää is18orOlder -metodin palauttaa vastauksenaan `true`. Tätä testataan yllä olevassa ohjelmakoodissa kahdella testitapauksella. Jos is18orOlder -metodin käsittelemää lukualuetta ryhtyy tarkastelemaan, voisi sen piirtää alla olevan kuvan muotoon.



Kuva 2. Onko henkilö 18 vuotta tai vanhempi.

Ihmisten ikä voi olla nolla tai enemmän. Tarkastelemassamme metodissa ikä 18 on tärkeä. Sitä testataan, onko henkilön ikä vähemmän kuin 18. Myös testataan tilannetta, jossa henkilö on 18 tai enemmän.

Junit
Sauli Isonikkilä



Kuva 3. Mitä numeroalueita kannattaa testata henkilön ikään liittyen.

Kannattaa siis testata kuvan 3 esittämiä punaisia alueita. Olisi siis lisättävä testitapaukset, joissa henkilön ikä on nolla vuotta (vastasyntynyt). Myös testitapaus, jossa henkilö on 18 vuotta, olisi hyvä testata. Entä jos henkilön syntymäpäiväksi annetaan vahingossa nykyhetkeen verrattuna tuleva päivämäärä. Tällaistakin tilannetta kannattaa testata. Miten tällaisessa tapauksessa ohjelmamme toimisi – se tulisi selvittää.