HELHa
Haute École Louvain en Hainaut

Syllabus

Programmation Orientée Objet

Les tests unitaires

Bachelier informatique de gestion Mons

BLOC 2 - UE 205

V. Altares, A. Colmant

Année académique 2019-2020

ÉCONOMIQUE

Table des matières

1) *Définition*

Un test unitaire permet de tester une méthode indépendamment de toutes les autres afin de vérifier que le comportement d'une méthode est conforme au résultat attendu. Le résultat renvoyé par un test unitaire sera toujours binaire (conforme ou non conforme). Il est donc aisé d'avoir une vue d'ensemble sur les résultats de tests sans devoir analyser chaque test.

2) *Assertions*

Il existe différentes assertions qui pourront être réalisées afin de vérifier le comportement d'une méthode. En voici une liste non exhaustive :

Method Summary	
`static void`	**`assertArrayEquals`**`(boolean[] expecteds, boolean[] actuals)` Asserts that two boolean arrays are equal.
`static void`	**`assertArrayEquals`**`(byte[] expecteds, byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	**`assertArrayEquals`**`(char[] expecteds, char[] actuals)` Asserts that two char arrays are equal.
`static void`	**`assertArrayEquals`**`(double[] expecteds, double[] actuals, double delta)` Asserts that two double arrays are equal.
`static void`	**`assertArrayEquals`**`(float[] expecteds, float[] actuals,`

	float delta) Asserts that two float arrays are equal.
static void	**assertArrayEquals**(int[] expecteds, int[] actuals) Asserts that two int arrays are equal.
static void	**assertArrayEquals**(long[] expecteds, long[] actuals) Asserts that two long arrays are equal.
static void	**assertArrayEquals**(Object[] expecteds, Object[] actuals) Asserts that two object arrays are equal.
static void	**assertArrayEquals**(short[] expecteds, short[] actuals) Asserts that two short arrays are equal.
static void	**assertArrayEquals**(String message, boolean[] expecteds, boolean[] actuals) Asserts that two boolean arrays are equal.
static void	**assertArrayEquals**(String message, byte[] expecteds, byte[] actuals) Asserts that two byte arrays are equal.
static void	**assertArrayEquals**(String message, char[] expecteds, char[] actuals) Asserts that two char arrays are equal.
static void	**assertArrayEquals**(String message, double[] expecteds, double[] actuals, double delta) Asserts that two double arrays are equal.
static void	**assertArrayEquals**(String message, float[] expecteds, float[] actuals, float delta) Asserts that two float arrays are equal.
static void	**assertArrayEquals**(String message, int[] expecteds, int[] actuals) Asserts that two int arrays are equal.
static void	**assertArrayEquals**(String message, long[] expecteds, long[] actuals) Asserts that two long arrays are equal.
static void	**assertArrayEquals**(String message, Object[] expecteds, Object[] actuals) Asserts that two object arrays are equal.
static void	**assertArrayEquals**(String message, short[] expecteds, short[] actuals) Asserts that two short arrays are equal.
static void	**assertEquals**(double expected, double actual, double delta) Asserts that two doubles are equal to within a positive delta.
static void	**assertEquals**(float expected, float actual, float delta) Asserts that two floats are equal to within a positive delta.

static void	**assertEquals**(long expected, long actual) Asserts that two longs are equal.
static void	**assertEquals**(Object expected, Object actual) Asserts that two objects are equal.
static void	**assertEquals**(String message, double expected, double actual, double delta) Asserts that two doubles are equal to within a positive delta.
static void	**assertEquals**(String message, float expected, float actual, float delta) Asserts that two floats are equal to within a positive delta.
static void	**assertEquals**(String message, long expected, long actual) Asserts that two longs are equal.
static void	**assertEquals**(String message, Object expected, Object actual) Asserts that two objects are equal.
static void	**assertFalse**(boolean condition) Asserts that a condition is false.
static void	**assertFalse**(String message, boolean condition) Asserts that a condition is false.
static void	**assertNotEquals**(double unexpected, double actual, double delta) Asserts that two doubles are **not** equal to within a positive delta.
static void	**assertNotEquals**(float unexpected, float actual, float delta) Asserts that two floats are **not** equal to within a positive delta.
static void	**assertNotEquals**(long unexpected, long actual) Asserts that two longs are **not** equals.
static void	**assertNotEquals**(Object unexpected, Object actual) Asserts that two objects are **not** equals.
static void	**assertNotEquals**(String message, double unexpected, double actual, double delta) Asserts that two doubles are **not** equal to within a positive delta.
static void	**assertNotEquals**(String message, float unexpected, float actual, float delta) Asserts that two floats are **not** equal to within a positive delta.
static void	**assertNotEquals**(String message, long unexpected, long actual) Asserts that two longs are **not** equals.
static void	**assertNotEquals**(String message, Object unexpected, Object actual) Asserts that two objects are **not** equals.

static void	**assertNotNull**(Object object) Asserts that an object isn't null.
static void	**assertNotNull**(String message, Object object) Asserts that an object isn't null.
static void	**assertNotSame**(Object unexpected, Object actual) Asserts that two objects do not refer to the same object.
static void	**assertNotSame**(String message, Object unexpected, Object actual) Asserts that two objects do not refer to the same object.
static void	**assertNull**(Object object) Asserts that an object is null.
static void	**assertNull**(String message, Object object) Asserts that an object is null.
static void	**assertSame**(Object expected, Object actual) Asserts that two objects refer to the same object.
static void	**assertSame**(String message, Object expected, Object actual) Asserts that two objects refer to the same object.
static <T> void	**assertThat**(String reason, T actual, Matcher<? super T> matcher) Asserts that actual satisfies the condition specified by matcher.
static <T> void	**assertThat**(T actual, Matcher<? super T> matcher) Asserts that actual satisfies the condition specified by matcher.
static void	**assertTrue**(boolean condition) Asserts that a condition is true.
static void	**assertTrue**(String message, boolean condition) Asserts that a condition is true.
static void	**fail**() Fails a test with no message.
static void	**fail**(String message) Fails a test with the given message.

Source : http://junit.org/junit4/javadoc/latest/org/junit/Assert.html

3) *Création d'un test unitaire*

3.1.Création d'une classe contenant différentes méthodes

La première étape consiste en la création d'une classe dont les méthodes pourront être testées par la suite.

```
public class DoublonException extends Exception {
	private static final long serialVersionUID = 1L;
	public DoublonException() {
		super("Déjà présent");
```

```
	}
}
public class Personne {
	private String nom, prenom;

	public Personne(String nom, String prenom) {
		super();
		this.nom = nom;
		this.prenom = prenom;
	}

	public boolean equals(Object o) {
		if (o instanceof Personne) {
			Personne p = (Personne)o;
			return this.nom.equals(p.nom) &&
					this.prenom.equals(p.prenom);
		}
		return false;
	}
}
import java.util.ArrayList;
import java.util.List;

public class Agenda {
	private List<Personne> personnes;

	public Agenda() {
		personnes = new ArrayList<Personne>();
	}

	public void ajouter(Personne p) throws DoublonException {
		if (!personnes.contains(p)) {
			personnes.add(p);
		}
		else {
			throw new DoublonException();
		}
	}

	public void supprimer(Personne p) {
		personnes.remove(p);
	}

	public Personne getIndice(int ind) {
		if (ind >= 0 && ind < personnes.size()) {
			return personnes.get(ind);
		}
		return null;
	}

	public int nombrePersonnes() {
		return personnes.size();
	}
}
```

3.2. Création d'un test unitaire

Les méthodes permettant de réaliser les tests seront précédées de l'annotation @Test. Par convention, les méthodes de test commencent par testNomMéthodeATester. Dans les méthodes de test seront placées des assertions qui permettront de contrôler le comportement de la méthode. Si une méthode testée propage une exception, alors cette exception devra aussi être propagée par le test.

```
@Test
public void testAjouter() throws DoublonException {
	/*Pour faire échouer le test tant que celui-ci n'a pas été écrit*/
	fail() ;
}
```

3.3. Création d'un test unitaire qui doit générer une exception

Lorsqu'un test doit vérifier si une méthode propage bien une exception quans cela est prévu, il faut préciser dans l'annotation du test quelle exception doit être générée par la méthode.

```
@Test (expected=DoublonException.class)
public void testAjouterDoublon() throws DoublonException {
	/*Pour faire échouer le test tant que celui-ci n'a pas été écrit*/
	fail() ;
}
```

3.4. Ignorer le résultat d'un test

Pour éviter qu'un test provoquant une erreur ne soit réutilisé, il faut placer la balise @Ignore au-dessus du test à ignorer. Cette annotation peut prendre en argument une chaîne de caractères précisant pourquoi le test n'est pas réalisé.

```
@Ignore
@Test
public void testNombrePersonne() {

}
```

3.5. Création d'un test case complet permettant de tester les méthodes d'une classe

Il est conseillé d'isoler les classes de tests des autres classes et donc de créer un package test spécifique. Pour créer un test case, il faut préciser sur quelle classe va porter le test case. Ensuite, il faut également préciser quelle version de Junit va être utilisée (les exemples sont basés sur la version 4).

L'annotation @Before précédant la déclaration de la méthode setUp permet de préciser dans la méthode ce qui doit être réalisé avant de commencer la réalisation de chaque test (instanciation d'une variable d'instance,...). L'annotation @After devant la déclaration de la méthode tearDown permet de spécifier ce qui sera accompli dès que les tests seront terminés.

```
import static org.junit.Assert.*;

import java.util.List;
```

```java
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TestAgenda {
	private Agenda agenda;
	private List<Personne> lesPersonnes;

	@Before
	public void setUp() throws Exception {
		agenda = new Agenda();
		lesPersonnes = (List<Personne>)Explorateur.getField(agenda, "personnes");
	}

	@After
	public void tearDown() throws Exception {
		agenda = null;
		lesPersonnes = null;
	}

	@Test
	public void testConstructeur() {
		Agenda ag = new Agenda();
		assertNotNull(ag);
		List<Personne> liste = (List<Personne>)Explorateur.getField(ag,
			"personnes");
		assertNotNull(liste);
	}

	@Test
	public void testAjouter() throws DoublonException {
		Personne p = new Personne("Goffin","David");
		agenda.ajouter(p);
		assertTrue("ajout normal",lesPersonnes.contains(p));
	}

	@Test (expected=DoublonException.class)
	public void testAjouterDoublon() throws DoublonException {
		Personne p = new Personne("Goffin","David");
		lesPersonnes.add(p);
		Personne p1 = new Personne("Goffin","David");
		agenda.ajouter(p1);
		assertEquals("ajout doublon",lesPersonnes.size(),1);
	}

	@Test
	public void testSupprimer() {
		Personne p = new Personne("Goffin","David");
		lesPersonnes.add(p);
		Personne p1 = new Personne("Hazard","Eden");
		lesPersonnes.add(p1);
		Personne p2 = new Personne("Hazard","Eden");
		agenda.supprimer(p2);
		assertFalse("test suppression",lesPersonnes.contains(p1));
	}

	@Test
	public void testGetIndice() {
```

```
		Personne p = new Personne("Goffin","David");
		lesPersonnes.add(p);
		Personne p1 = new Personne("Hazard","Eden");
		lesPersonnes.add(p1);
		Personne p2 = agenda.getIndice(1);
		assertTrue("test indice OK",p2.equals(p1));
		Personne p3 = agenda.getIndice(-2);
		assertTrue("test indice négatif",p3==null);
		Personne p4 = agenda.getIndice(2);
		assertTrue("test indice trop grand",p4==null);
	}

	@Test
	public void testNombrePersonne() {
		Personne p = new Personne("Goffin","David");
		lesPersonnes.add(p);
		assertEquals(1,agenda.nombrePersonnes());
	}
}
```

Pour lancer les tests, il faut démarrer un JUnit Test. Le résultat des tests sera présenté dans une fenêtre reprenant le détail des tests effectués ainsi que leur résultat. Si au moins 1 test échoue, la barre de progression apparaît en rouge. Dans le cas contraire, elle apparaît en vert. Le temps nécessaire pour effectuer les différents tests est également affiché.

4) Création d'un stub (bouchon)

4.1. Dépendance entre classes

Dans notre exemple, si on rajoute une méthode calculerSalaire dans la classe Entreprise qui permet d'obtenir le total des salaires des employés de l'entreprise, une dépendance est alors créée entre Entreprise et Personne. Cela signifie donc que pour tester la méthode calculerSalaire, la classe Personne devrait être écrite au préalable.

```
	public double calculerSalaire() {
		double salaire = 0.;
		for (int i=0;i<liste.size();i++) {
			salaire += liste.get(i).getSalaire();
		}
		return salaire;
	}
```

4.2. Inversion de dépendance

Afin de supprimer cette dépendance, il faut rajouter une interface à la classe Personne. Cette interface reprendra les services dont on aura besoin dans la classe Entreprise.

```
public interface InterfacePersonne {
	public double getSalaire();
}
```

4.3. Utilisation de l'interface au lieu de la classe

Dans la classe Entreprise, il ne faut plus faire référence qu'à l'interface de la classe Personne (au lieu de la classe Personne à proprement parler).

4.4. Création du stub

La création d'un stub va permettre de définir un comportement statique.

```
public class BouchonPersonne1500 implements InterfacePersonne {
	@Override
	public double getSalaire() {
		return 1500.0;
	}
}
```

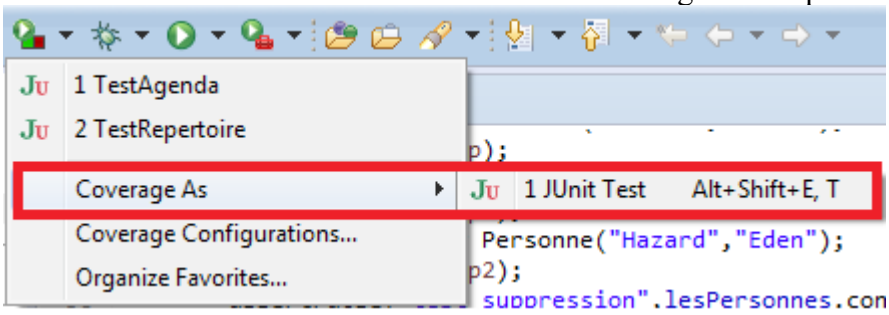
4.5. Utilisation du stub dans le test

```
	@Test
	public void testCalculerSalaire() throws Exception {
		InterfacePersonne p1 = new BouchonPersonne1500();
		InterfacePersonne p2 = new BouchonPersonne1500();
		entreprise.ajouterPersonne(p1);
		entreprise.ajouterPersonne(p2);
		assertEquals(3000.0,entreprise.calculerSalaire());
	}
```

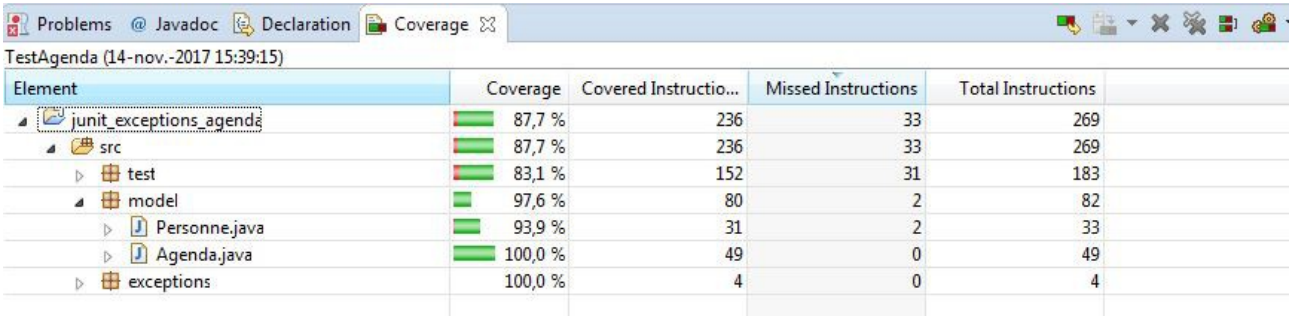
5) Couverture de code

5.1. Générer le rapport de couverture de code

Pour générer un rapport de couverture, il faut lancer le programme (à partir de la classe contenant le test case) en utilisant le nouveau bouton et en sélectionnant « coverage as » et puis « JUnit Test ».



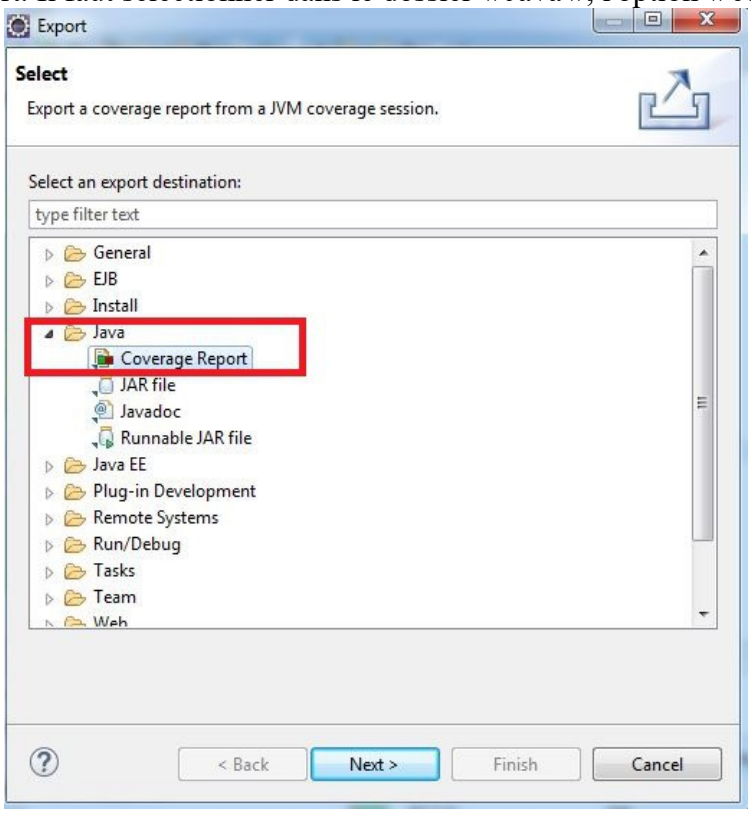
Un rapport de couverture de code va apparaître dans une nouvelle vue nommée « coverage ». Dans cette vue, le taux de couverture de chaque classe sera indiqué en pourcentage des instructions couvertes par un test.



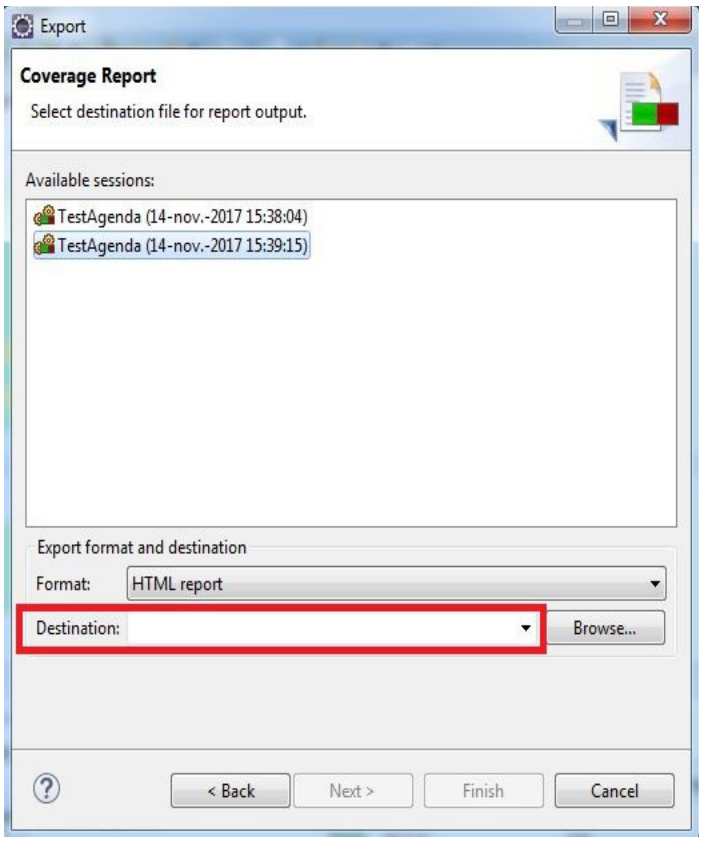
Chaque classe apparaîtra maintenant avec les instructions surlignées dans une des couleurs suivantes : verte (l'instruction est couverte par un test), jaune (une condition n'est pas testée entièrement) ou rouge (une instruction n'est pas couverte par un test).

```
public boolean equals(Object o) {
    if (o instanceof Personne) {
        Personne p = (Personne)o;
        return this.nom.equals(p.nom) &&
               this.prenom.equals(p.prenom);
    }
    return false;
}
```

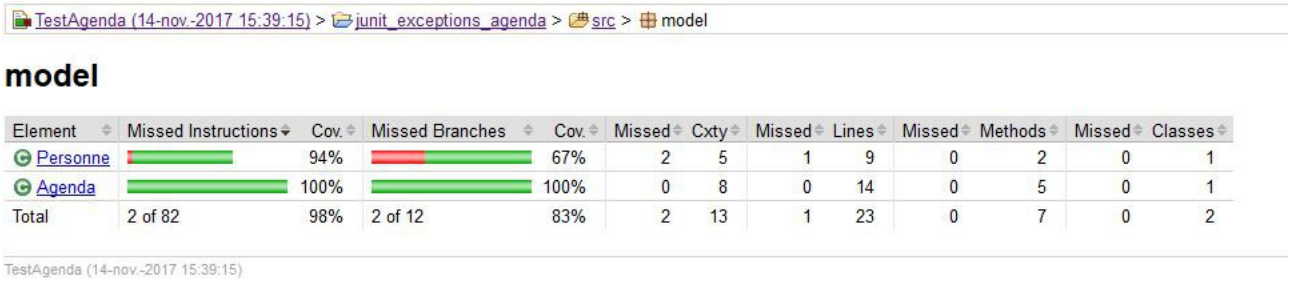
Pour récupérer le rapport sous forme d'un document HTML, il faut faire un clic droit sur le projet Java et choisir l'export. Il faut sélectionner dans le dossier « Java », l'option « coverage report ».



Dans l'écran suivant, il faut compléter l'emplacement de destination des fichiers de rapports (à mettre dans un dossier car plusieurs fichiers sont générés).



Dès que les fichiers sont générés, ils peuvent être consultés à l'emplacement de destination choisi en ouvrant le fichier nommé « index.html ».



6) Références

- Open classrooms : http://fr.openclassrooms.com/informatique/cours/les-tests-unitaires-en-java, dernière consultation le 14 novembre 2017.
- B. Gantaume, **Junit, mise en œuvre pour automatiser les tests en Java**, 1ère édition, 2011, ENI.
- C. Delannoy, **Programmer en Java**, 9ème édition, 2014, Eyrolles.