

Les outils de Tests Unitaires

Hakim MOKEDDEM

École nationale Supérieure d'Informatique

Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests Doubles
- Les Tests Doubles avec Mockito
- Qualité des tests unitaires
 - Couverture du code
 - Les tests de mutations
 - Bonnes pratiques

Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests Doubles
- Les Tests Doubles avec Mockito
- Qualité des tests unitaires
 - Couverture du code
 - Les tests de mutations
 - Bonnes pratiques

Généralités sur les tests unitaires: C'est quoi un test unitaire ?

- Un programme qui vérifie le bon fonctionnement d'une partie d'un logiciel.
- Un programme qui vérifie que **des données connues en entrée** produisent des **résultats connues**.



Généralités sur les tests unitaires:

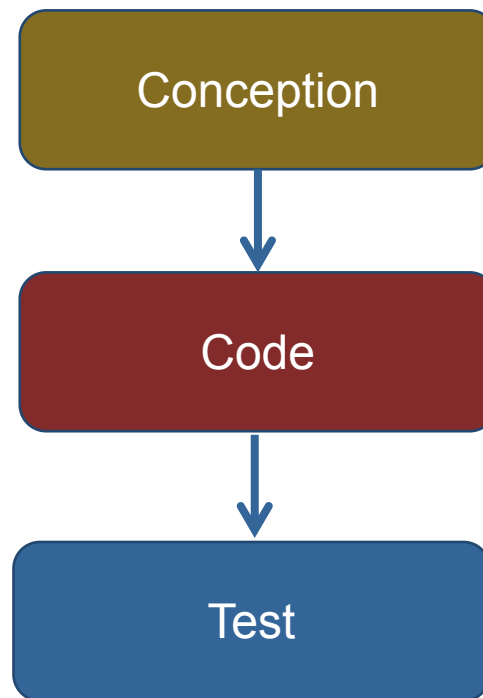
Pourquoi les tests unitaires ?

- Identifier tôt les bugs.
- Minimiser le coût de traitement des bugs.
- Faciliter la détection des bugs après la maintenance.
- Faciliter l'intégration des modules.

Généralités sur les tests unitaires: Les approches de test

- **Approche classique**

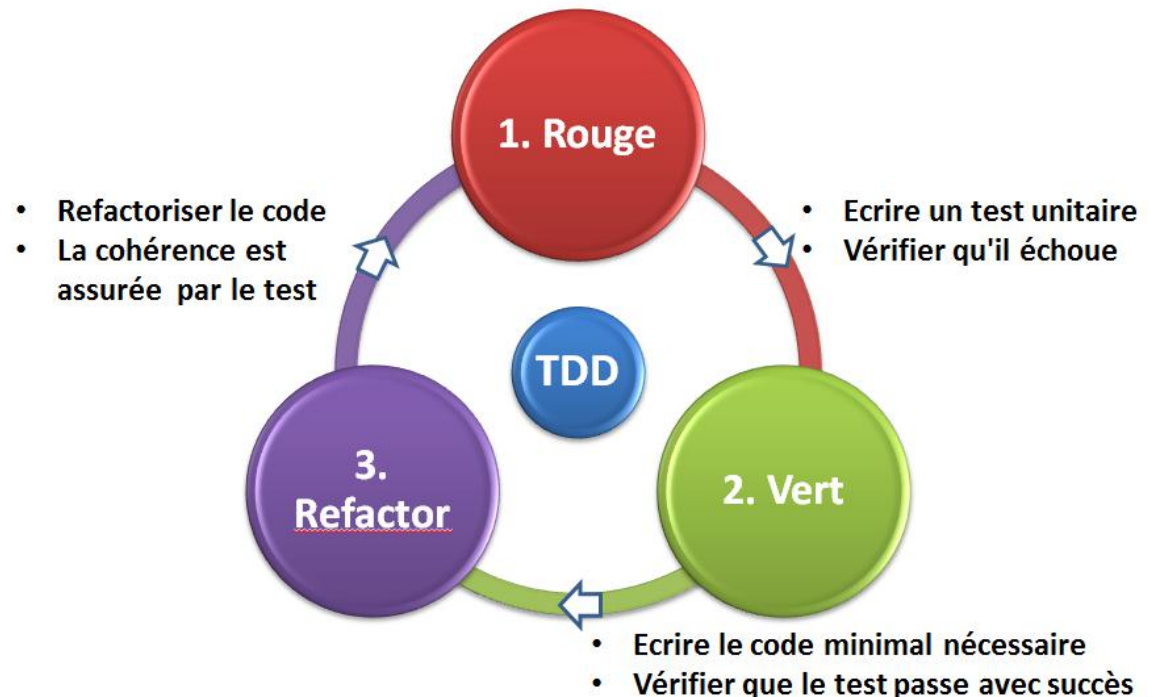
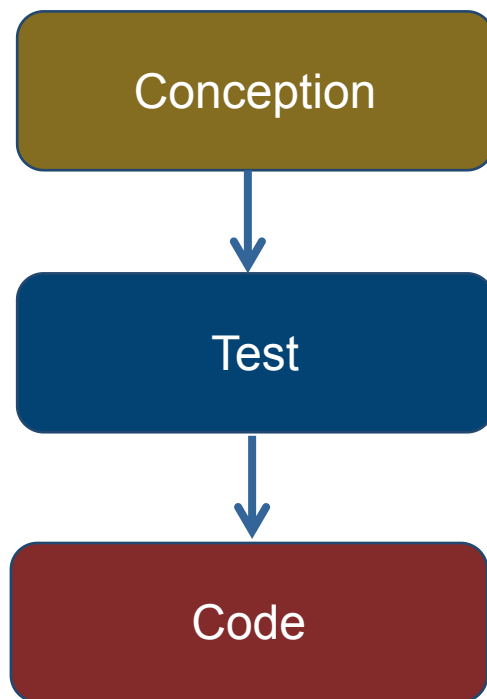
Ecrire le code et par la suite le tester.



Généralités sur les tests unitaires: les approches de test

- **Approche TDD (Test-Driven Development)**

L'écriture du code est guidée par les tests.



Généralités sur les tests unitaires: caractéristiques d'un bon test

1. Rapidité

- L'exécution d'un test unitaire doit être rapide.
- Un test devrait s'exécuter en une seconde ou moins.
- Un ensemble de tests devrait s'exécuter en quelques minutes.

Généralités sur les tests unitaires: caractéristiques d'un bon test

2. Isolation

- Les données de tests ne doivent pas dépendre de:
 - L'environnement dans lequel le test est exécuté.
 - Les résultats d'un autre test.
- Aucun ordre dans l'exécution des tests.

Généralités sur les tests unitaires: caractéristiques d'un bon test

3. Répétabilité

- Avoir le même résultat pour chaque exécution.
- Aucune dépendance avec la date/temps.
- Aucun résultat aléatoire.

Généralités sur les tests unitaires: caractéristiques d'un bon test

4. Auto-validation

- Aucune vérification manuelle de l'exécution (**Pass** ou **Fail**).



Plan

- Généralités sur les tests unitaires
- **Les tests unitaires avec JUnit**
- Les Tests doubles
- Les Tests Doubles avec Mockito
- Qualité des tests unitaires
 - Couverture du code
 - Les tests de mutations
 - Bonnes pratiques

Le framework JUnit

- Un framework de tests unitaires pour Java.
- Un framework de la famille xUnit.

Exemple

```
import org.junit.*;

public class ClacTest {
    @Test
    public void add() {
        assertEquals(new CalcService().add(2,4),6);
    }
}
```

Tests unitaires avec JUnit:

Test case Vs. Test suite

Test case. Une classe Java pour tester des méthodes.

Exemple

```
public class CalcServiceTest {  
    @Test  
    public void add() {  
        assertEquals(new CalcService().add(2,4),6);  
    }  
    @Test public void mult() {  
        assertEquals(new CalcService().mult(2,4),8);  
    }  
}
```

Tests unitaires avec JUnit:

Test case Vs. Test suite

Test suite. Une classe Java qui contient des Tests cases.

Exemple

```
@RunWith(Suite.class)
@SuiteClasses({
    CalcServiceTest.class,
    CustomerServiceTest.class,
})
public class FeatureTestSuite {
}
```

Tests unitaires avec JUnit: les annotations

- *@Test*. Spécifier qu'une méthode est un test.
- *@Test(expected = Exception class)*. Tester une exception.
- *@Test(timeout= 100)*. Vérifie que la durée d'exécution du test est ≤ 100 ms.

Tests unitaires avec JUnit: les annotations

- *@Before*. Exécuter la méthode avant chaque test.
- *@After*. Exécuter la méthode après chaque test .
- *@BeforeClass*. Exécuter la méthode au début des tests.
- *@AfterClass*. Exécuter la méthode à la fin des tests.
- *@Ignore*. Ignorer l'exécution d'un test .

Tests unitaires avec JUnit: les assertions

- *assertEquals*. Vérifier que deux valeurs sont égales.
- *assertTrue*. Vérifier que la condition est True.
- *assertNotNull*. Vérifier qu'un objet est non null.
- *assertSame*. Vérifier que deux objets ont la même référence.
- *assertArrayEquals*. Vérifier que deux tableaux sont égaux.

Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- **Les Tests doubles**
- Les Tests Doubles avec Mockito
- Qualité des tests unitaires
 - Couverture du code
 - Les tests de mutations
 - Bonnes pratiques

Les Tests doubles: limites de JUnit

Problèmes

- Comment tester de manière isolée une méthode métier qui interagit avec des méthodes techniques ?
- Comment tester une méthode qui appelle une méthode non encore implémentée ?

Les Tests doubles: limites de JUnit

Exemple

```
public class CartService() {  
    IProductDao pDao; // iPoroductDao une interface  
    CartDao cDao;  
    public boolean addToCart(Product p) {  
        boolean added = false;  
        if(pDao.getQte(p)>0) {  
            added = cDao.add(p); }  
        return added;  
    } }  

```

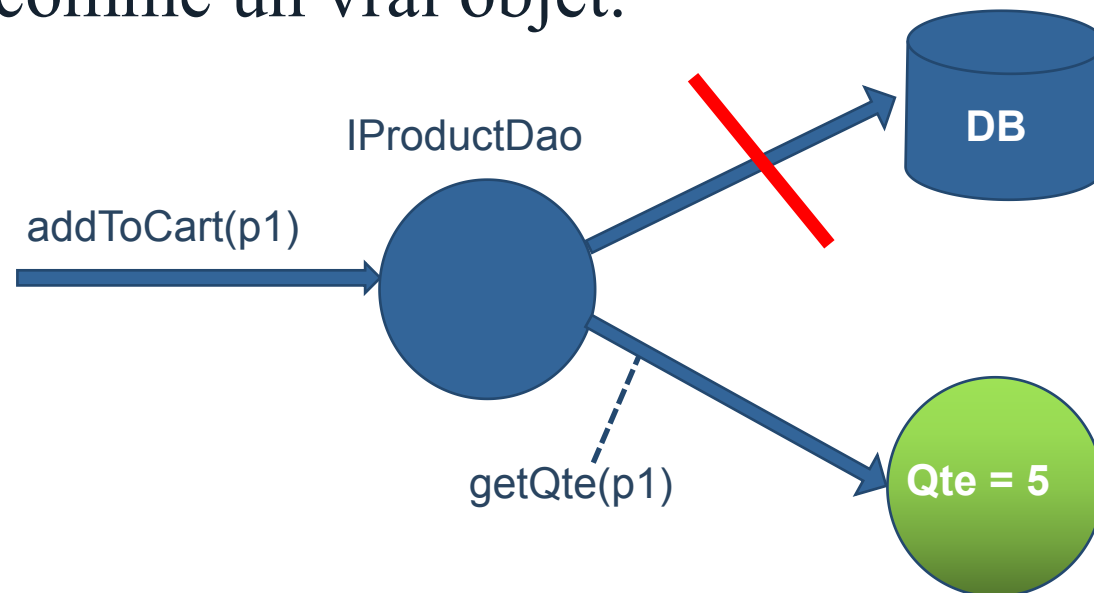
Comment isoler le test unitaire de **addToCart** avec **pDao.getQte** et **cDao.add** ?

Les Tests doubles: principe

Solution. Utiliser le principe du **Test Double**

C'est quoi un Test Double ?

Un **Test Double (Mock object)** est un objet de test qui se comporte comme un vrai objet.



Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests doubles
- **Les Test Doubles avec Mockito**
- Qualité des tests unitaires
 - Couverture du code
 - Les tests de mutations

Les Tests Doubles avec Mockito

Un framework Java open source pour les tests doubles.

Pourquoi Mockito ?

- Une syntaxe simple.
- Une bonne documentation.
- Une grande communauté de développeurs.



Les Tests Doubles avec Mockito: fonctionnalités

- Création d'un Mock (Test Double).

Exemple

```
public interface IProductDao {  
    public int getQte(Product p);  
}
```

```
public class CartDao {  
    public boolean add(Product p) {  
        // Implémentation  
    }  
}
```

```
IProductDao pDao=Mockito.mock(IProductDao.class);  
CartDao cDao=Mockito.mock(CartDao.class);
```

Les Tests Doubles avec Mockito: fonctionnalités

- Test Stub. Le résultat attendu d'un Mock.

Exemple

```
Product p1 = new Product (22445, "Smathpone",...)  
CartService cartService = new CartService();  
IProductDao pDao=Mockito.mock(IProductDao.class);  
CartDao cDao =Mockito.mock(CartDao .class);  
Mockito.when(pDao.getQte(p1)).thenReturn(5);  
Mockit.wohen(cDao.add(p1)).thenReturn(true);  
assertTrue(cartService.addToCart(p1));
```

Les Tests Doubles avec Mockito: fonctionnalités

- Test Spy. Vérifier l'appel d'une méthode.

Exemple

```
IProductDao pDao=Mockito.mock(IProductDao.class);
```

```
CartDao cDao =Mockito.mock(CartDao .class);
```

```
when(pDao.getQte(p1)).thenReturn(5);
```

```
when(cDao.add(p1)).thenReturn(true);
```

```
assertTrue(cartService.addToCart(p1));
```

```
Mockito.verify(pDao).getQte(p1);
```

```
Mockito.verify(cDao).add(p1);
```

Les Tests Doubles avec Mockito: bonnes pratiques

- Créer des mocks pour séparer les tests des modules.

Exemple. Séparer le test d'une méthode métier avec le test d'une méthode technique.

- Ne pas créer les mocks sur tous les objets qui existent.
- Ne pas créer les mocks sur une entité.

Exemple. *Product p1 = Mockito.mock(Product.class).*

- Ne pas créer les mocks sur les librairies externes.

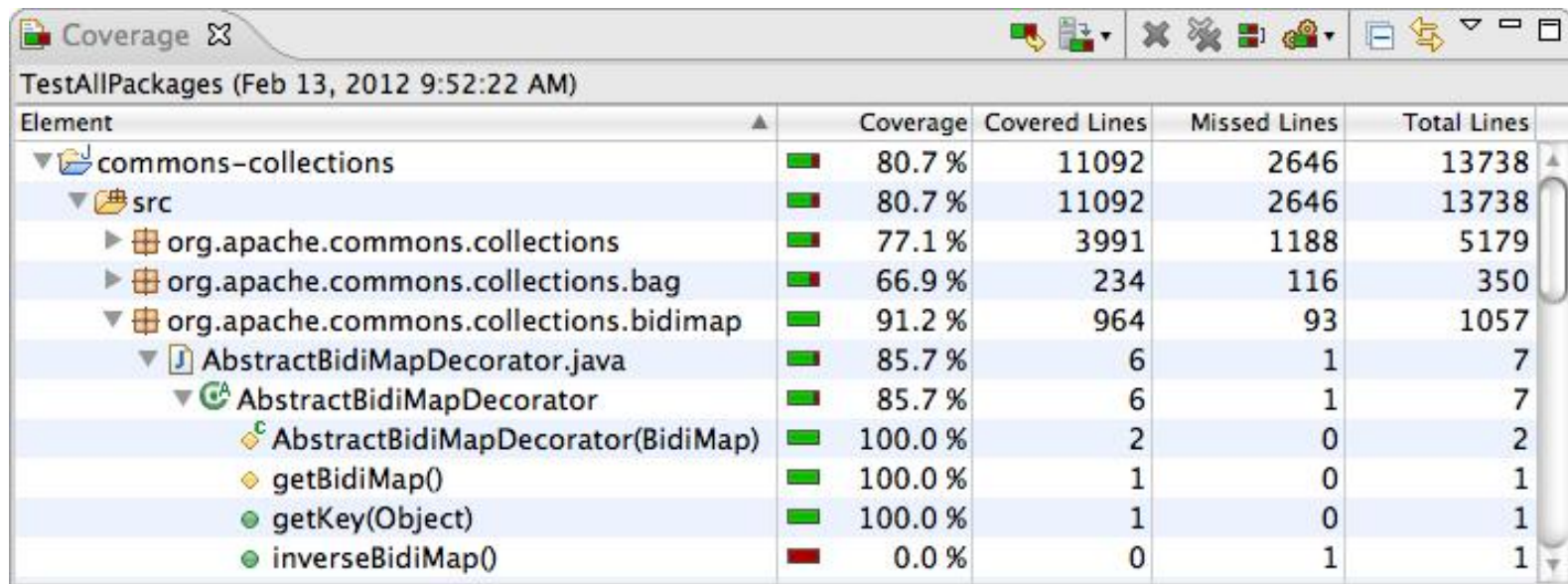
Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests doubles
- Les Tests Doubles avec Mockito
- **Qualité des tests unitaires**
 - Couverture du code
 - Les tests de mutations
 - Bonnes pratiques

Qualité des tests unitaire: la couverture du code

C'est quoi la couverture du code ?

Une mesure exprimée en pourcentage de la proportion du code exécuté par un test suite.



The screenshot shows the 'Coverage' window in an IDE, titled 'TestAllPackages (Feb 13, 2012 9:52:22 AM)'. It displays a tree view of the project structure with columns for 'Element', 'Coverage', 'Covered Lines', 'Missed Lines', and 'Total Lines'. The tree shows the 'commons-collections' package and its sub-packages, with detailed coverage data for the 'org.apache.commons.collections.bidimap' package and its classes and methods.

Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

Qualité des tests unitaire: la couverture du code

Avantages de la couverture du code

- Identification des parties du code testées.
- Identification des parties non-testées.
- Une métrique qui pourrait mesurer la qualité des tests

Les outils de couverture du code

- JaCoCo, Clover, JCov, Cobertura.
- Intégrée par défaut dans IntelliJ IDEA.

Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests doubles
- Les Tests Doubles avec Mockito
- **Qualité des tests unitaires**
 - Couverture du code
 - **Les tests de mutations**
 - Bonnes pratiques

Les tests de mutations: limites de la couverture du code

- Non-détection de tous les bugs.
- Non-détection des parties critiques non-testées.
- Aucune évaluation des données utilisées pour le test.
- Avoir 100% de couverture du code ne signifie pas un test de bonne qualité.

Code Coverage != Software Quality

Les tests de mutations: limites de la couverture du code

Exemple

```
public void add(int a,int b) {  
    return a+b;  
}
```

@Test

```
public void add() {  
    assertEquals(new CalcService().add(0,4),4);  
}
```

- Couverture du code = 100%
- Si on change $a+b$ par $a-b$ dans la méthode *add*, le test passe.

Les tests de mutations

- Proposés par Richard Lipton en 1971.
- Utilisés pour améliorer la qualité d'un test.

C'est quoi une mutation ?

- Une modification légère du code source original.

Qu'est ce qu'elle mesure ?

- Elle mesure la capacité d'un test à détecter les bugs après une modification du code.

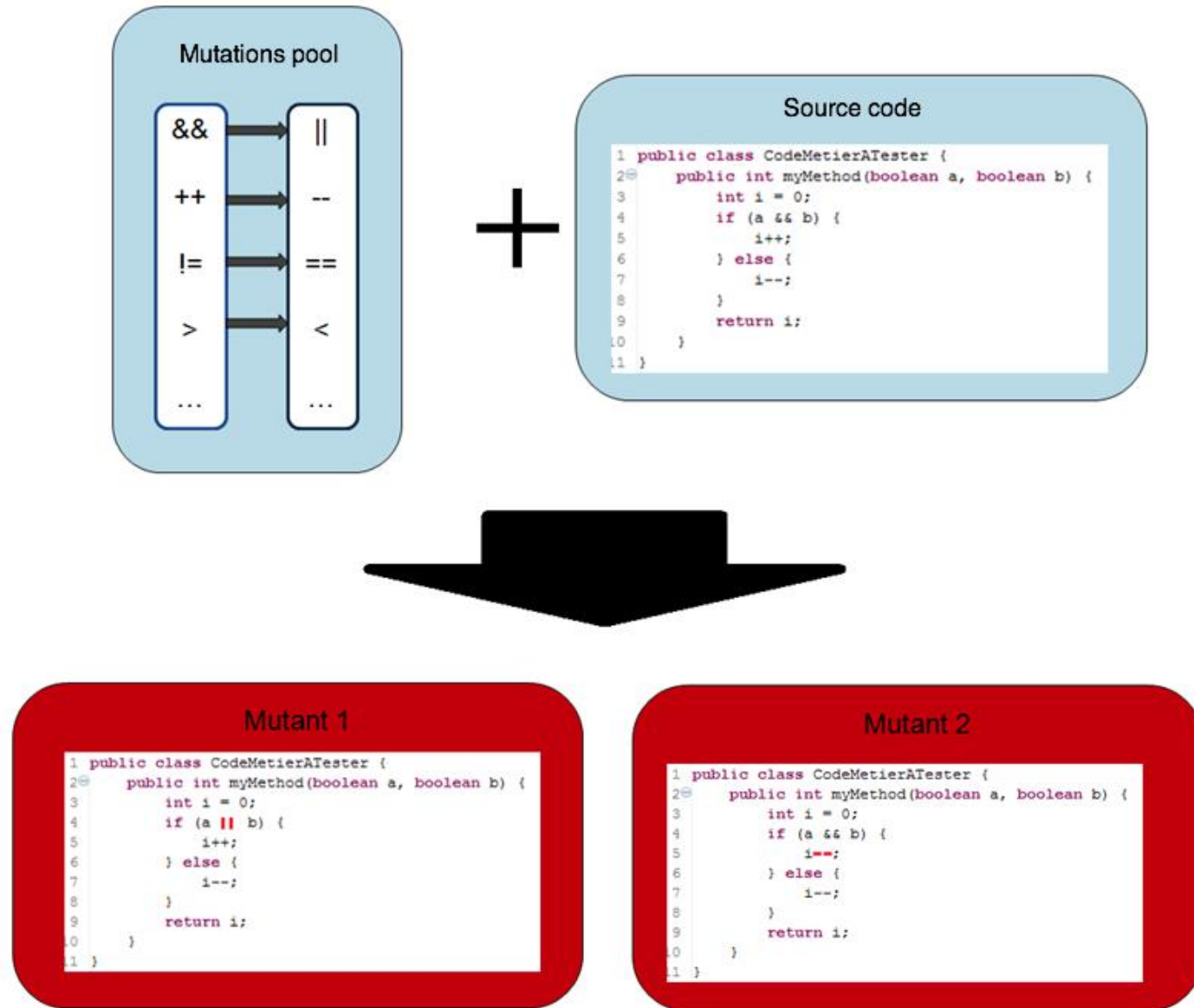
Les tests de mutations: création d'une mutation

Hypothèse de création d'une mutation

"Les erreurs des développeurs expérimentés sont dues à des erreurs syntaxiques simples".

Program p	Mutant p'
<pre>... if (a > 0 && b > 0) return 1; ...</pre>	<pre>... if (a > 0 b > 0) return 1; ...</pre>

Les tests de mutations: création d'une mutation



Les tests de mutations: les étapes

1. Créer des mutations.
2. Pour chaque mutation.
 1. Tester la mutation.
 2. Si le test **passe** alors garder la mutation.
 3. Si le test **échoue** alors tuer la mutation.
3. La qualité des tests est mesurée par le pourcentage des mutations tuées.

Les tests de mutations: l'outil PIT

- Un outil pour les tests de mutation en Java
- Disponibilité des plugins Eclipse/IntelliJ
- Les mutations utilisées par PIT:

<http://pitest.org/quickstart/mutators/>



Plan

- Généralités sur les tests unitaires
- Les tests unitaires avec JUnit
- Les Tests doubles
- Les Tests Doubles avec Mockito
- **Qualité des tests unitaires**
 - Couverture du code
 - Les tests de mutations
 - **Bonnes pratiques**

Qualité des tests unitaire: bonnes pratiques

Couverture du code

- Ne pas toujours fixer 100% de couverture de test comme objectif.
- Identifier les parties critiques non-testées.

Les tests de mutations

- Utiliser les tests de mutations pour les algorithmes complexes à tester.

Références

1. Kaczanowski, Tomek. Practical Unit Testing with JUnit and Mockito. Tomasz Kaczanowski, 2013.
2. Acharya, Sujoy. Mastering Unit Testing Using Mockito and JUnit. Packt Publishing Ltd, 2014.
3. <https://junit.org/junit4/>
4. <https://site.mockito.org/>
5. <https://github.com/mockito/mockito/wiki/How-to-write-good-tests>
6. <http://pitest.org/>