IUT Nantes
Pôle Sciences et technologie

Nantes Université

Test Unitaire
Junit 5

Jean-Marie Mottu (Lanoix, Le Traon, Baudry, Sunye)

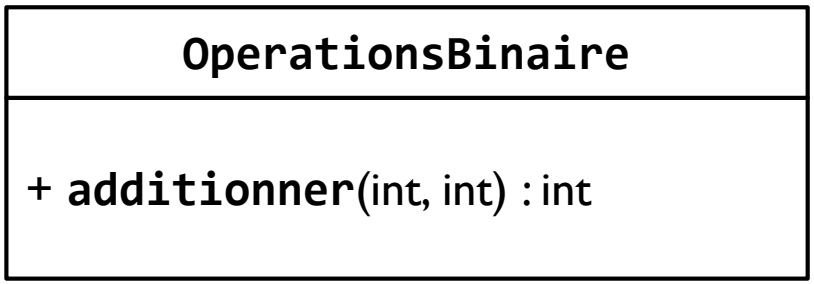
Cas des test

- Tester c'est Construire/Programmer/Exécuter un ensemble de
  - Cas de test
    - Une description : *"vérifier le passage à l'heure d'hiver"*
    - Une initialisation : *on doit instantier l'horloge du péage*
      - Par exemple créer un objet, le mettre dans un état précis
    - Une donnée de test : *+1 minute le* 31 oct. 2017 *à 02h59*
      - Par exemple certains paramètres de la méthode à tester

        (on détermine ces paramètres avec les techniques de prochains cours)
    - Un oracle : *il doit être 02h00 (et pas 03h00)*
      - Contrôler que l'exécution de la donnée de test respecte la specification

Test unitaire : Test intensif des unités de test

- La première étape de test après la programmation est le test unitaire
  - Tester une unité isolée du reste du système
- En Prog Objet, l'unité est la classe
  - Test unitaire = test de l'unité classe
  - Classe Sous Test
- On considère les classes indépendamment les unes des autres.




```java
public class OperationsBinaire {//java
  /**
   * additionner deux entiers
   */
  public int additionner(int x, int y)
  {
                  return x + y;
  }
}
```

Test Unitaire != Mise au point

- Trop long de taper en console et de contrôler les résultats pour de multiples tests :

```
public class OperationsBinaire {//java

/**
 * additionner deux int
 */
public int additionner(int x, int y) {
        return x + y;
}

public static void main(String[] args) {
        OperationsBinaire op = new OperationsBinaire();
        int sum = op.additionner(Integer.valueOf(args[0]),
                                 Integer.valueOf(args[1]));
        System.out.println(args[0]+"+"+args[1]+" rend "+ sum);
}
}
```

Test Unitaire != Mise au point

- Un peu mieux :

```
public class OperationsBinaire {//java

/**
 * additionner deux int
 */
public int additionner(int x, int y) {
        return x + y;
}

public static void main(String[] args) {
        OperationsBinaire op = new OperationsBinaire();
        int sum = op.additionner(1, 2);
        System.out.println("1+2 censé rendre 3 rend "+ sum);
        sum = op.additionner(2, 3);
        System.out.println("2+3 censé rendre 5 rend "+ sum);

}
}
```

Test Unitaire != Mise au point

- On s'approche :

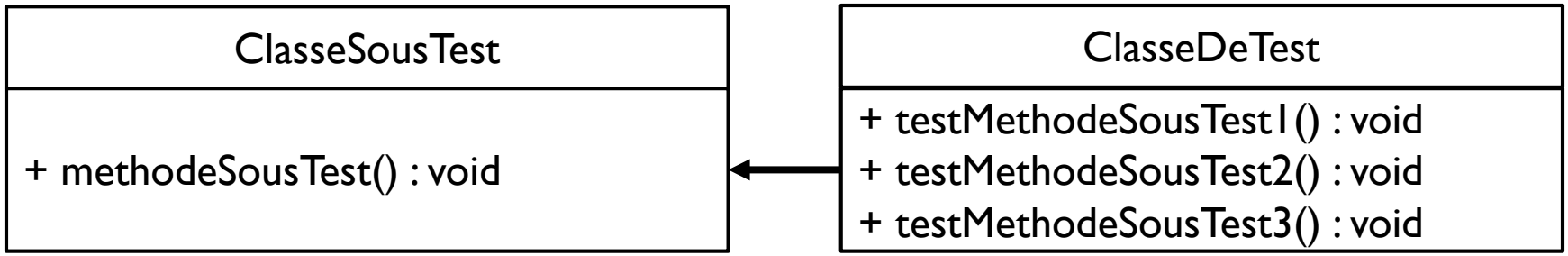
```
public class OperationsBinaire {//java

/**
 * additionner deux entiers
 */
public int additionner(int x, int y) {
        return x + y;
}

public static void main(String[] args) {
        OperationsBinaire op = new OperationsBinaire();
        int sum = op.additionner(1, 2);
        System.out.println("test de 1+2 donne "+ sum +
                                « ce qui est " + (sum == 3));
        sum = op.additionner(2, 3);
        System.out.println ("test de 2+3 est "+ sum +
                                « ce qui est " + (sum == 5));
}
```

Test Unitaire : Classes de test

- Consacrer des classes et des méthodes pour lancer des cas de test qui contrôlent le comportement du programme

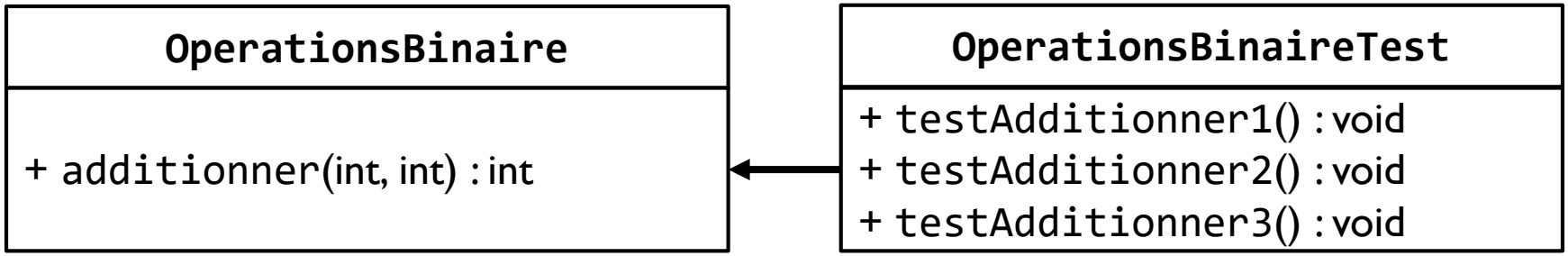



```
class OperationsBinaireTest {//java
	OperationsBinaire op = new OperationsBinaire();

	@Test
	void testAdditionner1() {
		int result = op.additionner(1, 2);
		assertEquals(3, result, "Somme de 1+2 devrait être 3");
}
```

Test Unitaire : Classes de test

- Par exemple




```java
public class OperationsBinaire {//java

  /**
   * additionner deux entiers
   */
  public int additionner(int x, int y)
  {
                return x + y;
  }
}
```

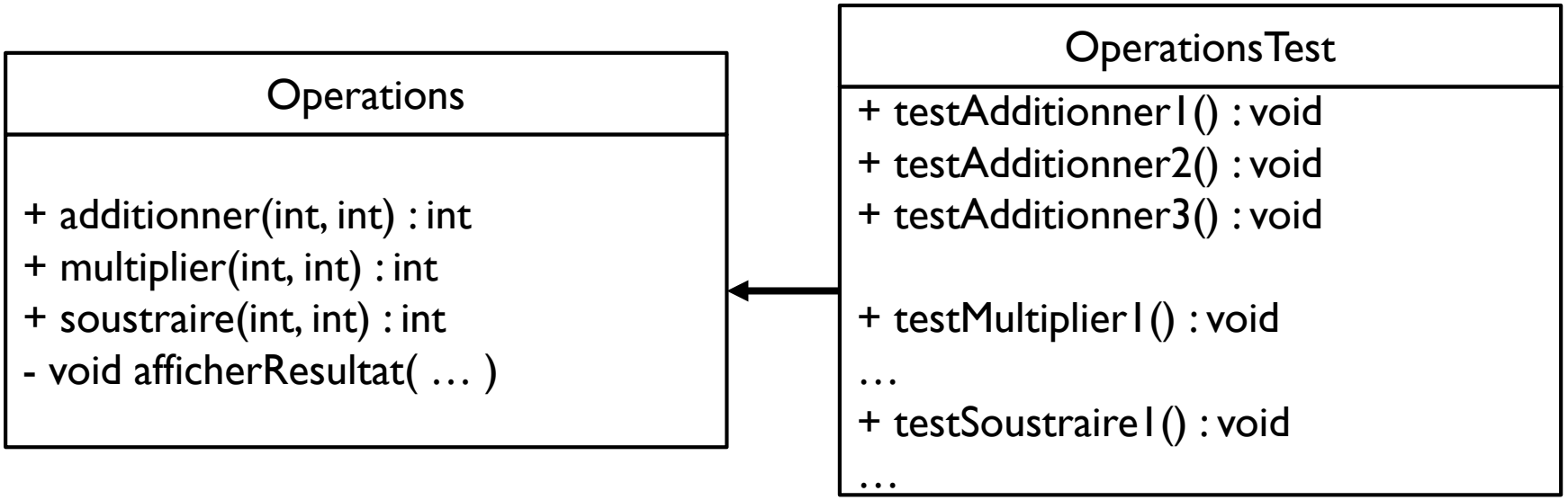
```java
class OperationsBinaireTest {//java
  OperationsBinaire op =
        new OperationsBinaire();

  @Test
  void testAdditionner1() {
    int result = op.additionner(1, 2);
    assertEquals(3, result,
       "Somme de 1+2 devrait être 3");
}
```

Méthodes de test

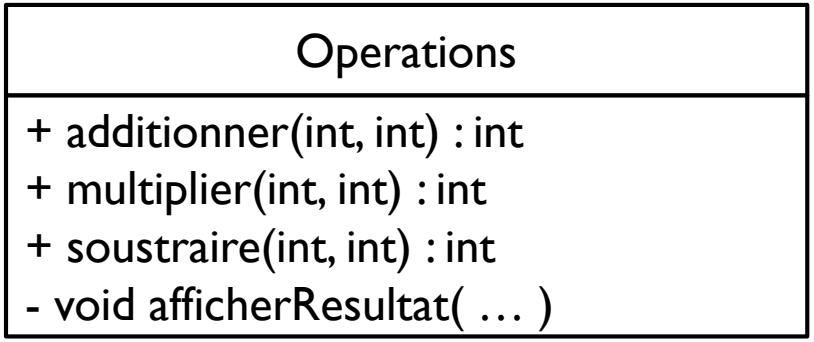
- Une méthode de test implémente un cas de test
  - Variante : une méthode de test peut éventuellement être paramétré(e) par des séries de données, implémentant une suite de cas de test (cf dernier partie de ce CM)
- Classe de test = Suite de tests indépendants
- Au moins une classe de test par classe testée
  - Regroupe les cas de test
  - Il peut y avoir plusieurs classes de test pour une classe testée

Exemple : test de la classe Operations



- Créer une classe de test qui manipule des instances de la classe Operations
- Au moins 3 cas de test (1 par méthode publique)
- Pas d'accès direct à la méthode afficherResultat, il faudra l'atteindre par l'intermédiaire d'autres méthodes

Exemple d'un cas de test : Addition de deux entiers



spécification du cas de test

```
//Premier test de l'addition de deux nombres
//L'addition de deux valeurs nulles donne nul
@Test
public void testAddition1(){//java
```

initialisation

```
    Operations op = new Operations();
```

appel avec donnée de test

```
    int res = op.additionner(0,0);
```

oracle

```
    assertEquals(0 , res, "Somme de nuls");
}
```

Test du point de vue client

- Le test d'une classe se fait à partir de classes extérieures du même package
- Les tests s'exécutent indépendamment les uns des autres
  - A la fin tout doit fonctionner
  - Néanmoins tant qu'il y a des fautes il est plus efficace d'ordonner la création des tests
    - quelles méthodes sont (inter)dépendantes ?
- Difficulté pour l'oracle
  - L'oracle prédit le résultat
  - Encapsulation : les attributs sont souvent privés
  - Difficile de récupérer l'état d'un objet
  - Penser à la testabilité au moment de la conception :
    - prévoir des accesseurs en lecture sur les attributs privés
    - des méthodes pour accéder à l'état de l'objet

IUT Nantes
Pôle Sciences et technologie

Nantes Université

Implémentation de tests unitaires Kotlin avec JUnit

Kotlin et les tests

- Rappel : Kotlin est compilé en bytecode et exécuté par la JVM comme Java
- Kotlin peut continuer à utiliser le framework de test (par défaut) de Java : JUnit
- Kotlin propose aussi son propre framework de test : Kotest

JUnit

- Origine
  - Extreme programming (test-first development)
  - framework de test écrit en Java par E. Gamma et K. Beck
  - open source : www.junit.org
- Généralisation des concepts (xUnit) :
  - Architecture introduite en 1994 avec SUnit (Smalltalk)
  - https://en.wikipedia.org/wiki/XUnit
- Objectifs
  - test d'applications en Java
  - faciliter la création des tests
  - tests de non régression

Codage JUnit5 (1/4)

- Organisation du code des tests
  - Méthode de test :
    - Chaque méthode de test annotée avec @Test implémente un seul cas de test
    - Chacune contient : description, initialisation, appel avec donnée de test, oracle d'un cas de test (sauf cas des tests paramétriques cf. plus loin)
  - Classe de Test : (TestCase dans le vocabulaire de Junit)
    - Contient les méthodes de test des différents cas de test
    - setUp() et tearDown() annoté avec @BeforeEach @AfterEach
      - Appelée avant et après (resp.) chaque méthode de test
      - Peut factoriser l'initialisation de plusieurs méthodes de test
    - une classe de test définie une suite de cas de test

Codage (2/4)

- Codage d'un « TestCase » Junit dans un programme Kotlin:
  - déclaration de la classe:

```
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.AfterEach

internal class OperationsTest {

    val op = Operations() // instance de la classe sous test

    @BeforeEach
    fun setUp() {...}

    @AfterEach
    fun tearDown() {...}

    @Test
    fun testAdditionner1() {       ...    }
}
```

Codage (3/4)

- la méthode setUp avec @BeforeEach:
  - //appelée avant chaque cas de test

```
@BeforeEach
fun setUp() {
    //réinitialisation d'attribut d'objet
    //rappel de constructeur, etc.
}
```

- la méthode tearDown avec @AfterEach:
  - //appelée après chaque cas de test

```
@ AfterEach
fun tearDown() {
        //libération de variable
        //remise à jour de BDD, etc.
}
```

Codage (4/4)

- les méthodes de test
  - chaque méthode implémente un et un seul cas de test
- caractéristiques:
  - nom préfixé par « test » par convention
  - Annotation @Test
  - contient obligatoirement un oracle programmé
    - Par défaut, il faut donc au moins une assertion
      - Obligatoire (sauf pour le test des levées d'exceptions)

```
@Test
fun testAdditionner1() {
   val op = Operations() // initialisation (qui aurait pu être anticipée)
   val res = op.additionner(0, 0) // lancement du test :
                        appel de la méthode sous test avec la donnée de test
   assertEquals(0, res, "0+0=0") // oracle: vérification active du résultat attendu
}
```

Les assertions

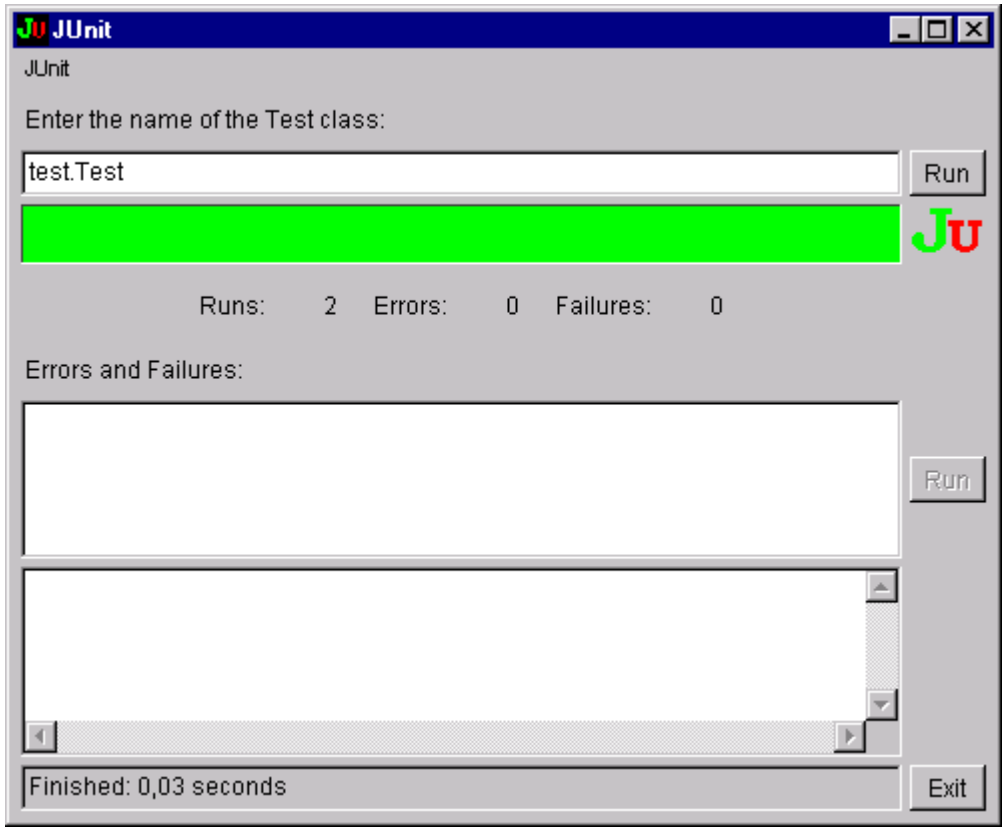
- assertEquals(Object expected, Object actual, String msg)
- assertSame(Object exp, Object act, String msg)
- assertEquals(Object exp [], Object act [], String msg)
- assertEquals(float exp, float act , float delta, String msg)
- assertTrue(boolean b, String msg)
- assertFalse(boolean b, String msg)
- assertNull(Object o, String msg)
- assertNotNull(Object o, String msg)
- fail(String msg)

- + des variantes (sans msg), etc.

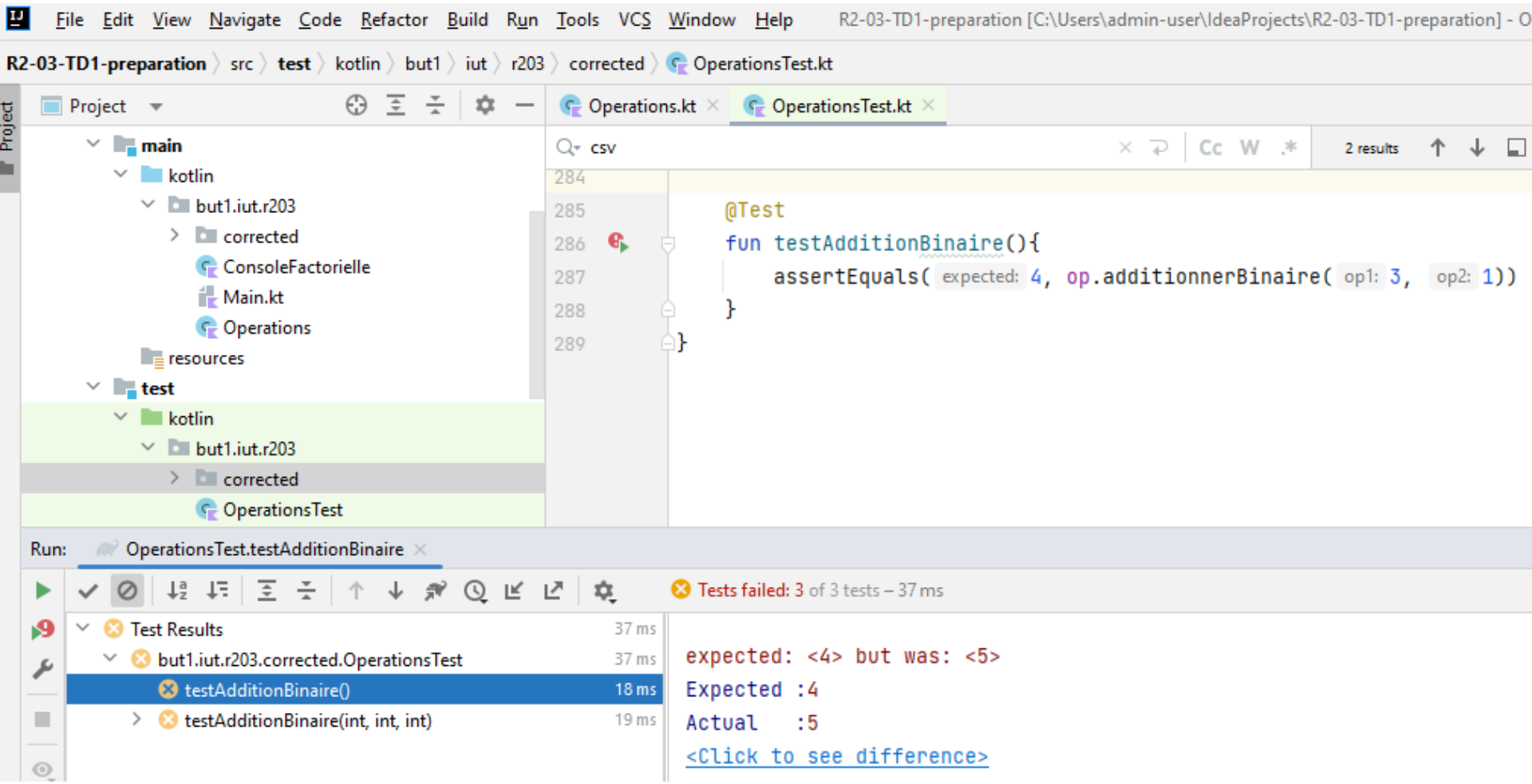
Verdict des tests JUnit

- pass
- failure
  - Une assertion n'est pas satisfaite
  - Révèle un bug
- error
  - Le test n'a pas pu terminer
    - Test erroné
    - Ou programme erroné
- Dans tous les cas, méfiance :
  - Les tests peuvent eux-mêmes être mal écrits
    - Mettant l'équipe projet en confiance à tort
    - Il faut d'abord bien comprendre les tests

TestRunner


JUnit
JUnit
Enter the name of the Test class:
test.Test
Run
Runs: 2 Errors: 0 Failures: 0
Errors and Failures:
Run
Finished: 0,03 seconds
Exit

Intégration dans IntelliJ



IUT Nantes
Pôle Sciences et technologie
Nantes Université

Test paramétrique

Tests paramétriques

- constatation : beaucoup de répétition de code quand on teste de grande séries de valeurs
- définition de patrons de code pour les classes de test, pour éviter la duplication du code

- Junit 5 intègre nativement la possibilité de faire des tests paramétriques paramétrés de multiple manière. (contrairement à Junit 4 qui possède dans sa librairie une fonctionnalité pour implémenter des tests paramétriques, mais souffrant de lacunes, nous lui préférons JUnitParams : http://pragmatists.github.io/JUnitParams/)

- CsvSource
- Permettant de passer plusieurs paramètres par test mais limité à des types convertissables depuis un String :

```
import org.junit.jupiter.params.ParameterizedTest
import org.junit.jupiter.params.provider.CsvSource

internal class OperationsTest {

        val op = Operations()

    @ParameterizedTest
    @CsvSource(
        "1, 2, 3",
        "2, 3, 5"
    )
    fun testAdditionBinaire(dt1: Int, dt2: Int, oracle: Int){
        assertEquals(oracle, op.additionnerBinaire(dt1, dt2))
    }
}
```

Permettant de passer plusieurs paramètres de tous types :

```
import org.junit.jupiter.params.ParameterizedTest
import org.junit.jupiter.params.provider.Arguments
import org.junit.jupiter.params.provider.MethodSource
import java.util.stream.Stream

internal class OperationsTest {
    val op = Operations()

    @ParameterizedTest
    @MethodSource("intTabProviderAdditioner")
    fun testAdditionBinaire2(dt1: Int, dt2: Int, oracle: Int, message: String){
        assertEquals(oracle, op.additionnerBinaire(dt1, dt2), message)
    }

    companion object {//nécessaire en kotlin pour utiliser des méthodes static (répandues en Java)
        @JvmStatic
        fun intTabProviderAdditioner(): Stream<Arguments?>? {
            return Stream.of(
                Arguments.of(1, 2, 3,"1+2=3"),
                Arguments.of(2, 3, 5, "2+3=5")
            )
        }
    }
}
```

Bénéfices de framework de test unitaire comme JUnit

- Simple à comprendre : méthode, classe et suites de test.
- Simple à utiliser: @Test, @BeforeEach, @AfterEach, etc.
- Structuré : cas de test, suite de tests.
- Exécution de programme simple et reproductible permettant le débogage
- Permet de sauvegarder les cas de test :
  - indispensable pour la non régression
    - quand une classe évolue, on ré-exécute les cas de test.
- Plusieurs extensions : tests de BD et IHM, rapports, etc.