Conception Logicielle

I - Développement Dirigé par les Tests (TDD)

- - - - - - - - - -

1 – Qualités d'un logiciel
2 – Cycles de développement & place des tests
3 – Tests unitaires & TDD
4 – Tests d'intégration

- - - - - - - - - -

ENSMA A3-S5 - période A

2023-2024

M. Richard
richardm@ensma.fr

Sommaire

I - Qualités d'un logiciel

1 – Qualités externes & internes

- Classées en deux catégories :
  1. *Qualités externes* : importantes car impactent l'utilisation du logiciel par les utilisateurs
  2. *Qualités internes* : très importantes car impactent fortement le cycle de vie du logiciel

- Mais les qualités internes influent aussi sur les qualités externes :
  - *Justesse* : juste par rapport aux spécifications
  - *Robustesse* : réagit correctement à des conditions anormales (i.e. non prévues par les spécifications)
  - *Évolutivité* : facilement ajustable aux changements de spécifications
  - *Vérifiabilité* : facilite la mise en place de procédures de test
  - *Extensibilité* : extension/ajout de fonctionnalités possible et facilitée
  - *Réutilisabilité* : facilement associable à d'autres éléments logiciels (Interface graphique, BDD, ...),
  - *Efficience* : nécessite peu de ressource (optimisation, complexité, ...)
    - parfois contradictoire avec réutilisabilité et extensibilité
  - *Portabilité* : Facilement transférable d'un environnement [1] à un autre
  - *Utilisabilité* : apprentissage rapide et utilisation aisée (nécessite très souvent de bien connaitre les futurs utilisateurs)
  - *Intègre* : sécurité des données, accès sécurisés, ...

[1] Un environnement est défini par l'ensemble matériel + Système d'exploitation

- Comment répondre au mieux à l'ensemble de ces critères ?
  $\Longrightarrow$ Faire un compromis . . .

Néanmoins, quatre qualités importantes doivent être privilégiées :

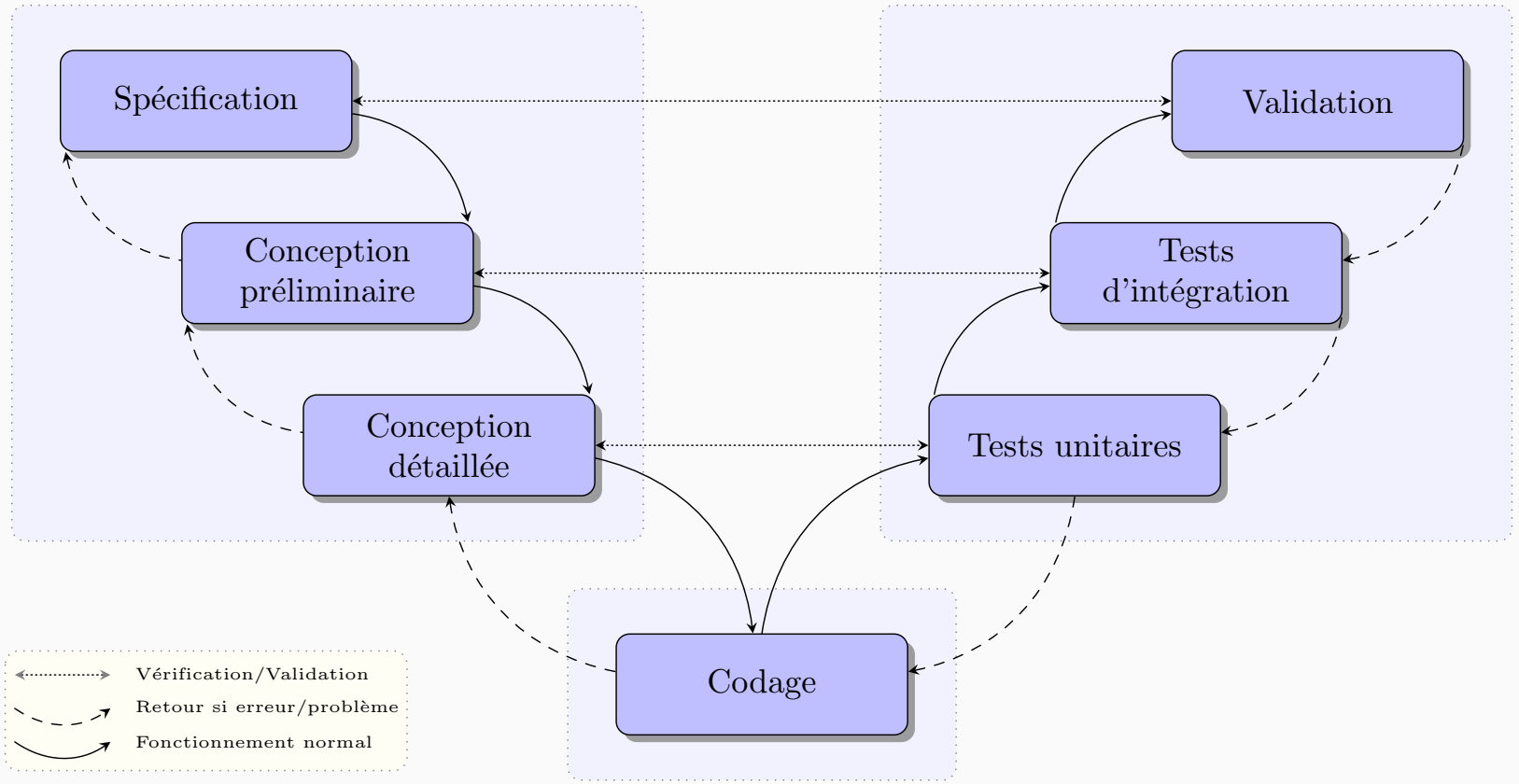
- *Justesse et robustesse :*
  Le respect de ces deux premiers points sera facilité par l'utilisation de :
  - Spécifications formelles (Méthode B, ...)
  - *Tests (unitaires, d'intégration)*
  - . . .

- *Extensibilité et réutilisabilité :*
  - utilisation de formalismes de modélisation (comme UML par exemple)
    - permettront de décomposer le logiciel en entités quasi autonomes afin d'obtenir le logiciel le plus modulaire possible.
  - mettre en œuvre le principe *d'architecture logicielle*
  - utilisation de *patrons de conception*
  - documentation interne
  - . . .

II - Cycles de développement & place des tests

1 – Cycle en V
2 – Cycle en spirale & Test Driven

↪ Rappel


Spécification
Conception préliminaire
Conception détaillée
Codage
Tests unitaires
Tests d'intégration
Validation
Vérification/Validation
Retour si erreur/problème
Fonctionnement normal

↪ Tests unitaires 1/2

Définition :

- Étape du cycle de développement consistant à vérifier le bon fonctionnement d'une partie d'un logiciel ou d'une unité
  - classe, module, ...

Objectif :

- Le but du test unitaire et de s'assurer que la réalisation, i.e. l'implémentation, correspond complètement à sa spécification

Méthode :

- le test permet de définir :
  - un état de sortie à l'issue de l'exécution du code testé, fonction de l'état d'entrée
  - un résultat attendu d'après la spécification
- le résultat du test correspond à la comparaison entre la sortie obtenue et la sortie attendue.

Quand :

- Dans un cycle de développement classique,
  - les tests unitaires viennent après la réalisation de tout ou partie du code
  - l'écriture des tests unitaires est souvent considérée comme une étape secondaire ...

↪ Tests unitaires 2/2

Avantages :

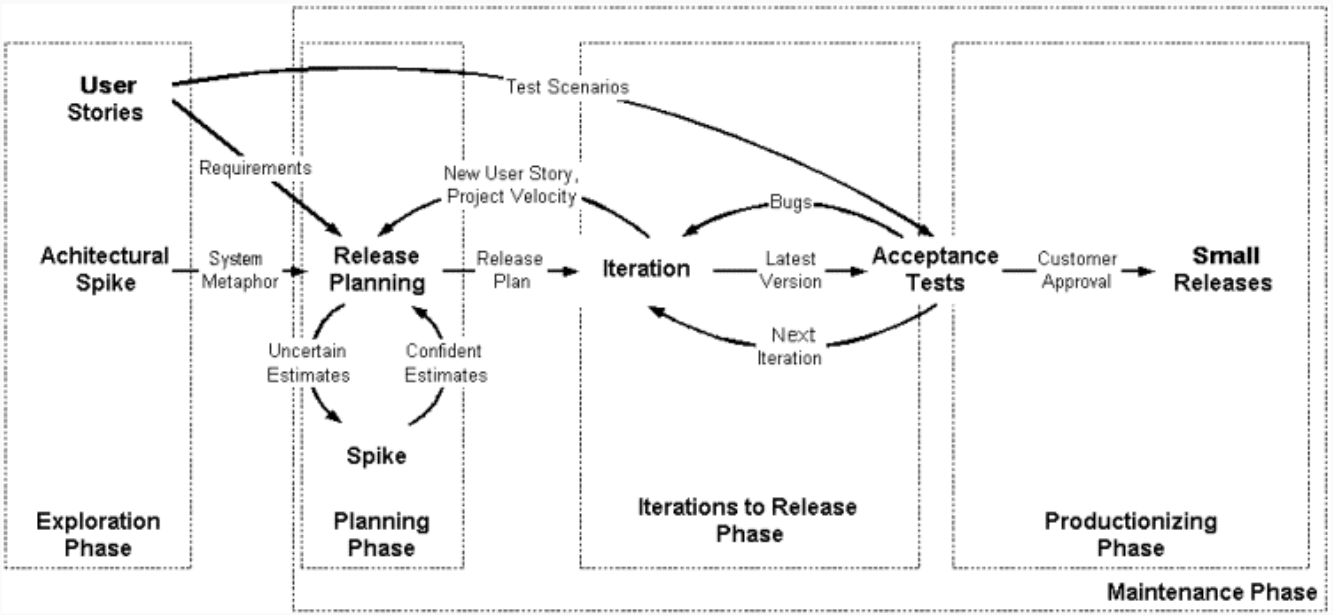
- Essentielle dans les applications critiques
- permet de mener en même temps des tests de couverture de code
  - vérification que lors du test l'ensemble du code de l'unité testée est exécuté
- assure la non-régression
  - à chaque modification du code du logiciel, l'ensemble de la batterie de tests doit être rejoué
  - permet ainsi de détecter très tôt l'introduction d'erreur(s) lors de la modification et donc la régression du code existant

L'outillage :

- Pourquoi des outils pour les tests :
  - éviter la pollution du code du logiciel par du code de test
  - diminuer le temps d'écriture des tests
  - proposer un environnement d'exécution et de gestion des tests
- Nombreux frameworks :
  - Il existe des frameworks de tests pour de nombreux langages
  - on utilise souvent le terme de xUnit ou x représente le langage visé par le framework
    - JUnit : Java
    - CUnit : C
    - NUnit : .NET
    - ...

↪ Définition

- Le développement dirigé par les tests est apparue avec la méthode d'Extreme Programming
  - l'objectif premier de cette méthode est de réduire les coûts dus aux changements en poussant à l'extrême des étapes déjà réalisées dans des méthodes classiques :
    - revue de code faite en permanence par un binôme
    - tests unitaires réalisés avant l'implémentation
    - conception tout au long du projet : c'est le refactoring
    - intégration des modifications au plus tôt
    - cycle très court
    - construction de "produit" partiel



↪ Fonctionnement

- Le cycle de réalisation d'une unité comporte 5 étapes
  1. Écrire le test
  2. Vérifier que le test échoue
     - le code n'existant pas, le test doit forcément échoué
  3. Écrire le code juste nécessaire (et pas plus) pour faire passer le test
  4. Vérifier que l'exécution du test passe
  5. Refactoriser le code
     - Amélioration du code sans changer les fonctionnalités

↪ Spécifications : Rappels

- Les spécifications doivent contenir :
  - la définition du domaine d'entrée
    - conditions pour chacun des paramètres
  - la définition du domaine de sortie
    - expression de ce que doit effectuer l'unité spécifiée
  - les éventuelles exceptions pouvant être levées
- Ce sont ces spécifications qui doivent être implémentées par la(les) méthode(s) de test

III - Tests unitaires & TDD

1 – Les bases
2 – Le Framework
3 – Vers les tests d'intégration

↪ Introduction à `JUnit`

- Présentation Framework JUnit :
  - Framework destiné au langage Java
  - associé à l'outil Maven, il propose un bon environnement pour la mise en place de tests
  - Permet d'écrire les différents tests correspondant àune classe
  - 2 versions courantes :
    - `JUnit 4` : plus ancienne version majeure, nécessitant java 1.5 ou supérieur
    - `JUnit 5` : version majeure la plus récente, nécessitant java 1.8 ou supérieur

- Principes généraux `JUnit 5`
  - proche en termes de philosophie de `JUnit 4`
    - basé sur les annotations et l'emploi de méthodes statiques permettant l'écriture des tests
  - S'appuie sur les nouveautés de java 1.8
    - *lambda expressions* en particulier
  - Améliore les tests de levée d'exception
  - enrichie les méthodes utilisées dans l'écriture des tests
  - propose de nouvelles catégories de tests

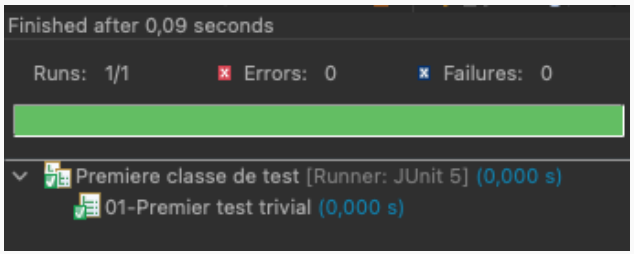
- Organisation des tests
  - dans une classe à part pour ne pas ≪ polluer ≫ le code
    - le principe général est de créer une classe de tests pour chacune des classes du projet
  - dans le ≪ même package ≫ grâce à l'utilisation avec *maven*
    - exécution des tests par la commande `mvn test`

↪ Écriture d'un test

- Un test est une méthode
  - annotée par `@Test`
  - annotation placée au-dessus de la signature
  - qui utilise une/des assertion·s pour l'écriture des cas des tests
- Possibilité de libeller une classe de tests et/ou une méthode de test
  - en utilisant l'annotation `@DisplayName`
- Exemple simple :

```
1           @DisplayName("Premiere classe de test")
2           public class AppTest {
3
4             @Test
5             @DisplayName("01-Premier test trivial")
6             public void tropFacile() {
7               int val = 1;
8               assertTrue(val == 1);
9             }
10          }
```

  - la méthode statique `assertTrue` est l'une des méthodes proposée par `JUnit`
- Résultat de l'exécution du test dans l'environnement Eclipse :



↪ Principe du ≪ Test Driven Development ≫ 1/3

- Étape 1 : écriture de la première version de la classe
  - les méthodes sont décrites avec un comportement par défaut
  - écriture des spécifications des différentes méthodes
  - exemple :

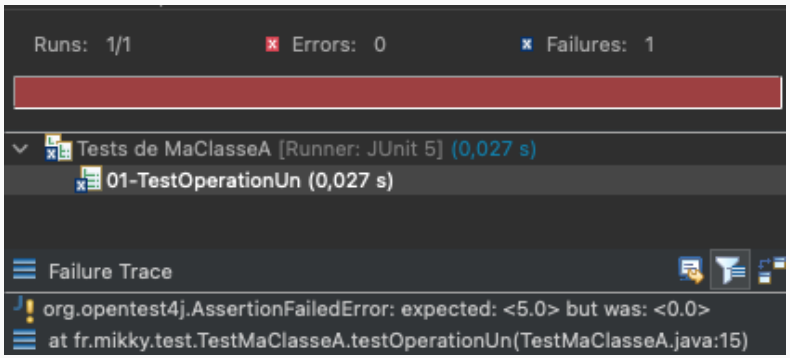
```
1                   public class MaClasseA {
2
3                     private int valA;
4                     private int valB;
5
6                     public MaClasseA(final int a, final int b) {
7                       valA = a;
8                       valB = b;
9                     }
10
11                    /**
12                    * nom: operationUn
13                    * E/ : un entier c
14                    * necessite : c/= 0
15                    * S/ : un flottant f
16                    * Entraine f=(a+b)/c
17                    */
18                    public float operationUn(final int c) {
19                      return 0.0f;
20                    }
21                  }
```

↪ Principe du ≪ Test Driven Development ≫ 2/3

- Étape 2 : écriture de la classe de tests
  - nom de la classe de test est généralement le nom de la classe préfixée par `Test`
  - pour chacune des méthodes à tester, écriture d'une ou plusieurs méthode·s de test
  - exemple :

```
1                   @DisplayName("Tests de MaClasseA")
2                   public class TestMaClasseA {
3
4                     @Test
5                     @DisplayName("01-TestOperationUn")
6                     public void testOperationUn() {
7                       MaClasseA mcl = new MaClasseA(5, 5);
8                       assertEquals(5.0, mcl.operationUn(2));
9                     }
10                  }
```

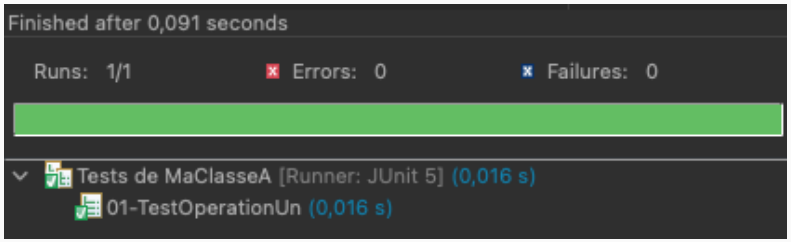
  - évidemment l'exécution du test échoue



↪ Principe du ≪ Test Driven Development ≫ 3/3

- Étape 3 : écriture du corps de la méthode pour obtenir la réussite du test

```
1           ...
2           public float operationUn(final int c) {
3             return (valA+valB)/c;
4           }
```



- Étape 4 : optimisation de l'écriture du code
  - le code de la méthode `operationUn` peut être optimisé si nécessaire
  - naturellement, le test doit toujours ≪ être au vert ≫

↪ Cycle de vie 1/3

- Principe :
  - `JUnit` construit une instance de la classe de test avant l'exécution de chacune des méthodes de test
  - Néanmoins, le framework met à disposition des annotations permettant de créer des méthodes s'exécutant à différents instants du cycle d'exécution.
    - Comme précédemment les annotations doivent être placées avant la signature de la méthode
  - on distingue 2 familles de méthodes :
    - la famille des méthodes s'appliquant pour chacun des tests
    - la famille des méthodes s'appliquant pour l'ensemble des tests

- Annotations :
  - `@BeforeAll` : s'exécute *une seule fois* avant le premier test
    - la méthode annotée est donc forcément `static`
  - `@AfterAll` : s'exécute *une seule fois* après tous les tests de la classe
    - la méthode annotée est donc forcément `static`
  - `@BeforeEach` : s'exécute *avant chacune* des méthodes de tests
  - `@AfterEach` : s'exécute *après chacune* des méthodes de tests
  - Ces quatre annotations sont généralement utilisées pour :
    - instancier des attributs utilisés dans tous les tests,
    - supprimer des instances, ressources, ...
    - préparer un contexte d'exécution particulier,
    - ...

↪ Cycle de vie 2/3

- Exemple :

```
1           @DisplayName("Tests de MaClasseA")
2           public class TestMaClasseA {
3
4             private static MaClasseA mcl;
5             private MaClasseA mcl2;
6
7             @BeforeAll
8             public static void initAll() {
9               System.out.println("Début des tests ...");
10              mcl = new MaClasseA(5, 5);
11            }
12
13            @BeforeEach
14            public void initEach() {
15              System.out.println("Début d'un test :");
16              mcl2 = mcl;
17            }
18
19            @Test
20            @DisplayName("01-TestOperationUn")
21            public void testOperationUn() {
22              assertEquals(5.0, mcl.operationUn(2));
23            }
24
25            @Test
26            @DisplayName("02-TestEquals")
27            public void testEquals() {
28              assertTrue(mcl.equals(mcl2));
29            }
```

↪ Cycle de vie 3/3

```
1           @AfterEach
2           public void finalizeEach() {
3             System.out.println("Fin d'un test.");
4             mcl2 = null;
5           }
6
7           @AfterAll
8           public static void finalizeAll() {
9             System.out.println("Fin de tous les tests ...");
10            mcl = null;
11          }
12        }
```

- résultat de l'exécution :

```
Début des tests ...
Début d'un test :
Fin d'un test.
Début d'un test :
Fin d'un test.
Fin de tous les tests ...
```

↪ Méthodes d'assertion classiques

- Différentes méthodes sont proposées pour exprimer au mieux les assertions :

Egalité	Nullité	Exceptions
`assertEquals()`	`assertNull()`	`assertThrows()`
`assertNotEquals()`	`assertNotNull()`	
`assertTrue()`		
`assertFalse()`		
`assertSame()`		
`assertNotSame()`		

  - Ces méthodes possèdent différentes surcharges permettant :
    - de s'adapter aux différents types prédéfinis du langage
    - d'utiliser des interfaces fonctionnelles (i.e. lambda expressions)
    - ...
  - nous verrons l'utilisation particulière de deux d'entre elles :
    - `assertThrows()` : pour la gestion des exceptions
    - `assertAll()` : pour regrouper un ensemble d'assertions

- D'autres méthodes plus spécifiques sont également proposées par le framework
  - pour les tableaux, les collections, les timeout, ...

↪ Gestion des exceptions 1/3

- Utilisation de l'assertion `assertThrows()`
  - uniquement basée sur l'utilisation d'interfaces fonctionnelles (i.e. lambda expressions)
  - vérifie que l'exécution du code passée en paramètre lève bien une exception du type (i.e. de la classe) précisé·e lors de l'appel de cette assertion.
  - dans le cas contraire, l'assertion fait échouer le test

- Exemple :
  - Ajout de la méthode `operationDeux` dans la classe `MaClasseA`

```
1                   /**
2                   * nom: operationDeux
3                   * E/ : un entier c
4                   * S/ : un flottant f
5                   * Entraine f=(a+b)/c si c/=0 ; ZeroException sinon
6                   */
7                   public float operationDeux(final int c) throws ZeroException {
8                     if ( c == 0) {
9                       throw new ZeroException();
10                    }
11                    return (valA+valB)/c;
12                  }
```

↪ Gestion des exceptions 2/3

- Exemple (suite)
  - Ajout d'une classe de définition de `ZeroException`

```
1                   public class ZeroException extends Exception {
2
3                     private String monMessage;
4
5                     public ZeroException() {};
6                     public ZeroException(final String msg) {
7                       monMessage = msg;
8                     }
9
10                    @Override
11                    public String getMessage() {
12                      return monMessage + " : " + super.getMessage();
13                    }
14
15                    public String getMonMessage() {
16                      return monMessage;
17                    }
18
19                    private static final long serialVersionUID = 1L;
20                  }
```

  - ce code vous montre, au passage, une manière de vous définir vos propres exceptions

↪ Gestion des exceptions 3/3

- Exemple (suite)
  - Il alors possible (et même nécessaire !) de tester notre méthode dans le cas d'une mauvaise utilisation
  - on ajoute alors le test suivant dans la classe de test :

```
1                   @Test
2                   @DisplayName("03-TestOperationDeux")
3                   public void testOperationDeux() {
4                     int valc = 0;
5                     assertThrows(ZeroException.class, () ->mcl.operationDeux(valc));
6                   }
```

    - comme attendu, le test est vérifié
  - mais il est possible de pousser un peu plus loin les vérifications :
    - la méthode `assertThrows` retourne l'instance de l'exception si cette dernière est levée
    - il est alors possible d'appeler des méthodes sur cette instance

```
1                   @Test
2                   @DisplayName("04-TestOperationDeux2")
3                   public void testOperationDeux2() {
4                     int valc = 0;
5                     ZeroException excep = assertThrows(ZeroException.class, ()
                            ->mcl.operationDeux(valc));
6                     assertEquals("Oupps !!", excep.getMonMessage());
7                     assertNull(excep.getCause());
8                   }
```

    - là encore, le test passe avec succès !

↪ Regroupement d'assertions

- `Junit` propose l'assertion `assertAll` :
  - permet de regrouper un ensemble d'assertions
    - sous la forme d'une suite d'interfaces fonctionnelles
  - toutes les assertions seront évaluées
    - même si l'une d'entre elles échoue
  - exemple : vérification que les attributs d'une instance de `MaClasseA` sont différents de 0

```
1                   @Test
2                   @DisplayName("04-TestOperationDeux3")
3                   public void testOperationDeux3() {
4                     int valc = 0;
5                     assertAll("Instance non conforme !",
6                     () -> assertTrue(mcl.getA() != 0, "valA vaut 0"),
7                     () -> assertTrue(mcl.getB() != 0, "valB vaut 0")
8                     );
9                     ZeroException excep = assertThrows(ZeroException.class, () ->
                            mcl.operationDeux(valc));
10                    assertEquals("Oupps !!", excep.getMonMessage());
11                    assertNull(excep.getCause());
12                  }
```

    - il peut être intéressant d'ajouter des messages aux différentes assertions afin d'obtenir un résultat de test plus précis

↪ Les suppositions

- Des méthodes particulières sont proposées afin de pouvoir conditionner l'exécution d'un test ou d'une partie de celui-ci
  - si l'évaluation d'une supposition échoue :
    - l'exécution du test est interrompu
    - le test est réussi (si la suite est correcte)
  - principales méthodes proposées :
    - `assumeTrue()`
    - `assumeFalse()`
    - `assumingThat()`
  - exemple : hypothèse sur la valeur du paramètre pour s'assurer d'un test dans le cas normal

```
1                   @Test
2                   @DisplayName("03-TestOperationDeuxAssume")
3                   public void testOperationDeuxAssume() {
4                     int valc = 2;
5                     assumeTrue(valc != 0);
6                     assertEquals(5.0, mcl.operationUn(2));
7                   }
```

↪ Autres annotations & méthodes . . .

- Ordonner vos tests :
  - plusieurs possibilités sont fournies pour contrôler l'ordre d'exécution des tests
    - `@Order(int val` : permet d'annoter chacune des méthodes de tests d'une classe en lui passant le numéro d'ordre
    - utiliser l'ordre alphabétique (sur le nom des méthodes de test)
  - Il faut alors annoter la classe avec :
    - `@TestMethodOrder(OrderAnnotation.class)` dans le premier cas
    - `@TestMethodOrder(Alphanumeric.class)` dans le second cas

- Désactiver un test :
  - l'annotation `@Disabled` permet de suspendre la méthode de test correspondante
    - accepte une chaîne en paramètre pour indiquer par exemple la raison de la désactivation

- Faire échouer un test :
  - dans certain cas, il peut être utile de provoquer volontairement l'échec d'une méthode de test
  - on utilise alors la méthode`Fail()` au sein du test
    - cette méthode accepte en paramètre une chaîne de caractères ou une exception
    - permet d'indiquer la cause de l'échec provoqué

↪ En pratique :

- Écriture des tests optionnelles lorsque les méthodes sont très simples
  - accesseurs par exemple
- Ne pas dépasser le contour de la classe testée
  - c'est-à-dire se limiter à tester ce qui relève de la classe et pas plus . . .
- Cas dans lesquels les tests unitaires ne peuvent s'appliquer ou ne suffisent pas :
  - les classes abstraites
  - les méthodes privées
  - les attributs privés ne possédant pas d'accesseurs
  - les dépendances : héritage, agrégation, composition, paramètres, . . .
    - Dans tous ces cas il est possible de mettre en œuvre le concept de doublure de tests
    - attention, ceci relève plus des *tests d'intégration* que des tests unitaires

IV - Tests d'intégration

1 – Le principe
2 – Le framework `Mockito`

↪ Doublure ou Mock

- Définition
  - Objet remplaçant un autre objet durant l'exécution du test.
  - Pour le code appelant, cette technique doit être transparente
    - il doit agir/réagir comme s'il manipulait du code normal
  - il existe plusieurs catégories de doublures :
    - les bouchons ou stubs
    - les simulacres ou mocks
    - les espions ou spy
    - les substituts ou fake
    - les fantomes ou dummy

Les tests d'intégration sont utiles lorsque que l'on veut tester une classe dans laquelle il y a une méthode provenant d'une autre classe ou autre.

Largement au dessus du test unitaire

- Quand mettre en œuvre les doublures ?
  - D'une manière générale, les doublures permettent de tester l'interaction entre objet et non de vérifier l'état de l'objet comme c'est le cas pour les tests unitaires
  - Ces doublures peuvent donc être utiles lorsque :
    - utilisation d'un composant difficile à mettre en œuvre
    - volonté de tester la classe dans une situation exceptionnelle (cas d'erreur, de mauvaise utilisation, etc.)
    - Composant non encore implémenté, mais bloquant pour l'écriture des tests
    - la méthode testée appelle un code lent
    - la méthode testée appelle un code non-déterministe
    - ...

↪ Coûts et bénéfices

- Cette technique possède un certain nombre d'avantages :
  - permet de réaliser bonne une partie des tests d'intégration
    - vérification des appels
    - vérification des paramètres d'appels
    - simulation des valeurs de retour (y compris des exceptions dans certains cas)
  - améliore donc la qualité des tests
  - force à réfléchir à la conception de l'application ...
    - couplage souple/fort des classes entre elles
    - utilisation plus importante des interfaces
  - peut être mise en œuvre dès que l'on dispose de l'interface de la classe (i.e. sans attendre son implémentation)

- mais elle apporte également quelques contraintes plus ou moins fortes
  - Doit suivre toute modification de l'entité dont la doublure à pris la place
    - les doublures risquent de ne plus être l'identique des implémentations concrètes
  - peut nécessiter par exemple :
    - de faciliter l'instanciation
    - de faciliter l'injection de dépendance (progrès des framework dans ce domaine)
    - de faciliter l'accès à une classe
    - ...
  - celles-ci dépendent toutefois du framework utilisé ...

↪ Différents (et nombreux) Framework

- même si l'écriture de ses propres doublures peut-être envisagée, l'utilisation d'un framework apporte de nombreuses et utiles fonctionnalités
- les principaux framework aujourd'hui sont :
  - EasyMock
  - Mockito
    - possède les principales fonctionnalités
    - assez simple à mettre en œuvre
    - fonctionne avec `JUnit 5`
  - JMock
  - JMockit
    - très complet et bien intégrer au framework `JUnit 4` ;
    - mise en œuvre relativement simple . . .

	Features						
Feature	EasyMock	jMock	Mockito	Unitils Mock	PowerMock (EasyMock)	PowerMock (Mockito)	JMockit
Invocation count constraints	√	√	√		√	√	√
Recording strict expectations	√	√			√		√
Explicit verification			√	√		√	√
Partial mocking	√		√	√	√	√	√
Easier argument matching based on properties of value objects	√		√	√	√	√	√
Cascading mocks			√	√		√	√
Mocking of multiple interfaces			√			√	√
Mocking of annotation types		√	√	√		√	√
Partially ordered expectations		√					√
Auto-injection of mocks			√	√		√	√
Mocking of enums					√	√	√
Declarative mocks for test methods (mock parameters)							√
Mocking of unspecified implementation classes							√
"Duck typing" fakes for integration tests							√
Total	4/14	4/14	8/14	6/14	5/14	9/14	14/14

Figure 1 – Comparaison de Framework

↪ Approche par l'exemple 1/5

- Ajout de la classe `MaClasseB` à notre projet
  - possède une relation (référence) vers `MaClasseA`

```
1  public class MaClasseB {
2
3    private MaClasseA classA;
4    private int coeff;
5    private float mult;
6
7    public MaClasseB(final MaClasseA cla, final int c,
8    final float m) {
9      classA = cla;
10     coeff = c;
11     mult = m;
12   }
13
14   public float operationTrois() {
15     try {
16       return classA.operationDeux(coeff)*mult;
17     } catch (ZeroException e) {
18       return Float.NaN;
19     }
20   }
21 }
```

  - la méthode `operationTrois` fait un appel à la méthode `operationDeux` de la classe `MaClasseA` en utilisant son attribut `coeff` comme paramètre
  - utilise le résultat de cet appel pour
    - retourner la valeur multiplier par son attribut `mult` s'il n'y a pas de levée de `ZeroException`
    - retourne `NaN` (Not a Number) sinon

↪ Approche par l'exemple 2/5

- la classe de test TestMaClasseB

```
1   @ExtendWith(MockitoExtension.class)
2   @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
3   public class TestMaClasseB {
4
5     private MaClasseB classeB;
6
7     // Déclaration (et creation) d'une doublure de MaClasseA
8     @Mock MaClasseA laClasseA;
9
10    @BeforeEach
11    public void initAll() {
12      //Instanciation en utilisant la doublure
13      classeB = new MaClasseB(laClasseA, 10, 2.5f);
14    }
15
16    @Test
17    @Order(1)
18    public void testOperationTrois() throws ZeroException {
19      // Définition du comportement
20      when(laClasseA.operationDeux(10)).thenReturn(1.0f);
21      // Appel
22      float res = classeB.operationTrois();
23      // Vérification intégration (i.e. appel)
24      verify(laClasseA).operationDeux(12);
25      // Vérification unitaire
26      assertEquals(2.5f,res);
27    }
28  }
```

Extension de Junit pour plus de fonctionnalites

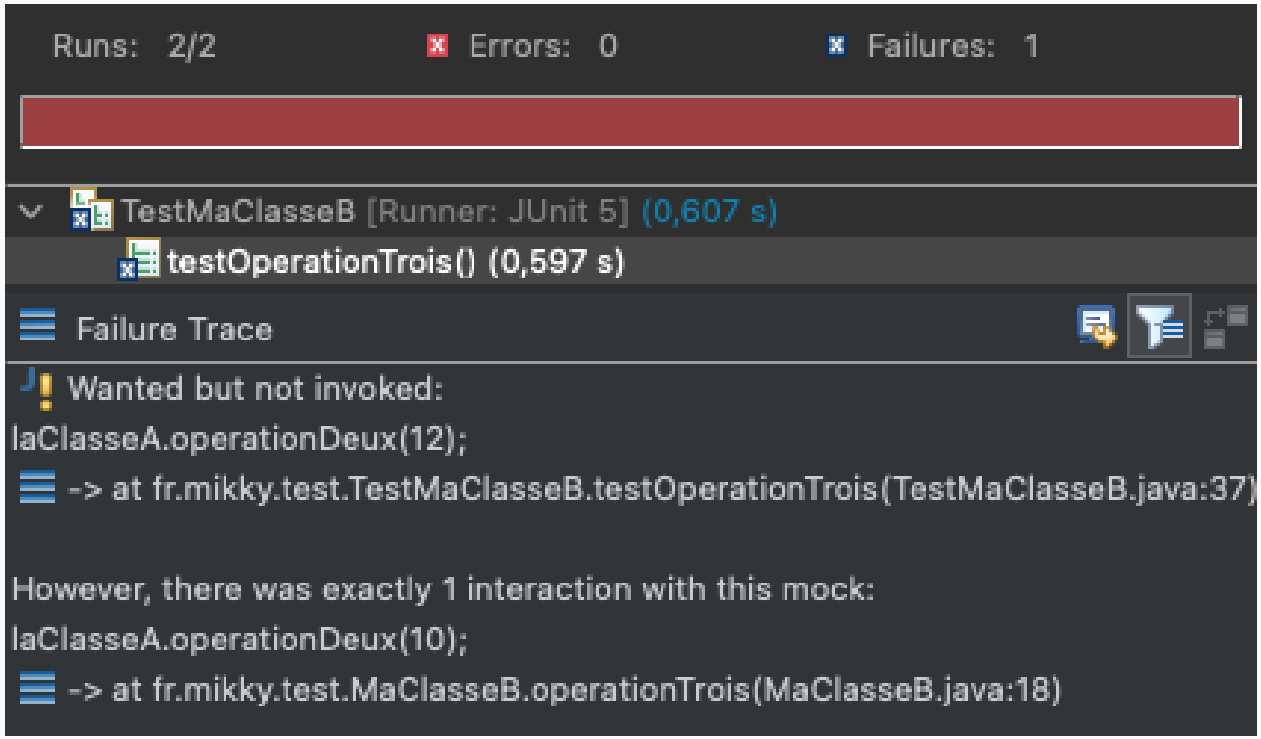
Propose des methodes avec la meme signature

↪ Approche par l'exemple 3/5

- Explications :
  - ligne 1 : annotation permettant d'indiquer à `JUnit` d'utiliser l'extension pour le framework `Mockito`
  - ligne 8 : Déclaration et création (par le framework) d'une doublure de `MaClasseA`
    - cette doublure possédera toutes les méthodes déclarées dans la véritable classe
  - ligne 20 : première phase d'un test avec doublure → définition du comportement
    - il s'agit de définir un comportement de la méthode appelée
    - en spécifiant des valeurs pour les paramètres d'appels
    - en spécifiant un comportement de retour (ici, valeur de retour valant 1.0)
    - utilise la méthode `when()` pour indiquer la méthode de la doublure spécifiée
    - utilise la méthode `thenReturn()` pour spécifier le comportement de retour
  - ligne 22 : deuxième phase d'un test avec doublure → appel de la méthode
    - on réalise l'appel de la méthode devant potentiellement provoquer l'appel de la méthode tel que spécifié dans la première phase
  - ligne 24 : troisième phase d'un test avec doublure → vérification de l'appel
    - on vérifie ici que l'appel de la méthode tel que spécifié dans la première phase a bien eu lieu
    - utilisation de la méthode `verify().methode` indiquer la doublure et la méthode de celle-ci dont on veut vérifier l'appel
  - ligne 26 : quatrième phase d'un test avec doublure → vérification du comportement unitaire
    - on vérifie que le résultat obtenu correspond bien à la valeur obtenue (en tenant compte de la spécification du comportement de retour faite lors de la première phase)
    - il s'agit ici d'un test unitaire classique utilisant les assertions `JUnit`

↪ Approche par l'exemple 4/5

- Résultat du test :



- ici, nous avons volontairement fait échouer le test en commettant une erreur sur la valeur du paramètre . . .

↪ Approche par l'exemple 5/5

- Test du comportement avec exception
  - ajoutons le test suivant à notre classe de test :

```
1                   @Test
2                   @Order(2)
3                   public void testOperationTroisExcep() throws ZeroException {
4                     // Définition du comportement
5                     when(laClasseA.operationDeux(10)).thenThrow(new
                            ZeroException("Test"));
6                     // Appel
7                     float res = classeB.operationTrois();
8                     // Vérification intégration (i.e. appel)
9                     verify(laClasseA).operationDeux(10);
10                    // Vérification unitaire
11                    assertEquals(Float.NaN,res);
12                  }
```

  - le comportement de la méthode `operationDeux` en cas d'appel a été modifié pour stipuler une levée e l'exception `ZeroException`
    - en utilisant la méthode `thenThrow()` (ligne 5)
    - le test est vérifié

↪ Définition du comportement : paramètres 1/2

- Lors de la phase de définition du comportement d'une méthode, les paramètres de cette dernière peuvent être stipulés de différentes manières
  - *valeur exacte :* une valeur du bon type est passée en paramètre
    - si la valeur est différente lors de l'appel, le test échoue
  - *type sans valeur :* il est également possible de stipuler un type au lieu d'une valeur
    - si le paramètre possède un type prédéfini `X` : utilisation de `anyX()`
    - si le type du paramètre est une classe `LaClasse` : utilisation de `any(LaClasse.class)`
    - lors de l'appel, seul le type du paramètre et non la valeur sera vérifié
  - *enregistrement pour vérification :* possibilité d'enregistrer la/les valeur·s d'un paramètre lors de l'appel
    - utilisation d'un élément de type `ArgumentCaptor<classe param>` en paramètre de la méthode lors de la phase de vérification

```
1                    ArgumentCaptor<Integer> arg =
                           ArgumentCaptor.forClass(Integer.class);
```

    - accès aux valeurs enregistrées par `getValues()` et à une valeur particulière par `get(index)` sur l'ensemble des valeurs précédentes

```
1                    arg.getAllValues().get(0)
```

↪ Définition du comportement : paramètres 2/2

- Exemple :

```
1           @Test
2           @Order(3)
3           public void testOperationTroisbis() throws ZeroException {
4             ArgumentCaptor<Integer> arg = ArgumentCaptor.forClass(Integer.class);
5             // Définition du comportement
6             when(laClasseA.operationDeux(anyInt())).thenReturn(1.0f);
7             // Appel
8             float res = classeB.operationTrois();
9             // Vérification intégration (i.e. appel)
10            //verify(laClasseA).operationDeux(10);
11            verify(laClasseA).operationDeux(arg.capture());
12            // Vérification unitaire
13            assertEquals(2.5f,res);
14            assertEquals(10, arg.getAllValues().get(0));
15          }
```

- la vérification des paramètres se fait alors par une assertion classique

↪ Définition du comportement : valeur·s de retour

- De même, plusieurs manières de spécifier le comportement du retour de la méthode :
  - une valeur : `thenReturn(val)` où `val doit être du bon type`
  - plusieurs valeurs :
    - chaînage de l'appel à `thenReturn`

```
1                    when(laClasseA.operationDeux(anyInt())).thenReturn(1.0f).thenRet
```

    - toutes les valeurs en paramètres :

```
1                    when(laClasseA.operationDeux(anyInt())).thenReturn(1.0f,
                           2.0f, 5.0f);
```

  - une exception : `thenThrow(new Exception())`
    - possibilité de chaîner les appels comme précédemment
- Cas particulier des procédures (i.e. type de retour `void`)
  - impossible d'utiliser la syntaxe `when ... thenThrow`
  - possibilité d'utiliser la syntaxe `doThrow(new Exception()).when ...` pour spécifier une exception
  - possibilité d'utiliser la syntaxe `doNothing().when()` si rien de particulier n'est à faire

↪ Définition de la vérification

- Vérifier l'ordre des appels
  - si l'on souhaite vérifier que l'appel de plusieurs méthodes sur une ou plusieurs doublure·s se fait dans le bon ordre il faut le spécifier en utilisant la classe (InOrder)

```
1                   when(laClasseA.operationDeux(10)).thenReturn(1.0f);
2                   when(laClasseA.operationDeux(15)).thenReturn(4.0f);
3                   ...
4                   InOrder inOrder = inOrder(laClasseA);
5                   ...
6                   inOrder.verify(laClasseA, times(2)).operationDeux(10);
7                   inOrder.verify(laClasseA, times(2)).operationDeux(15);
8                   ...
```

- Vérifier le nombre d'appels d'une méthode
  - le principe est d'ajouter un paramètre après la doublure lors de l'appel à la méthode `verify`
    - exactement n fois : utilisation de `times(n)`
    - au plus n fois : utilisation de `atMostOnce()` si $n = 1$ ou de `atMost(n)`
    - au moins n fois : utilisation de `atLeastOnce()` si $n = 1$ ou de `atLeast(n)`
    - jamais : utilisation de `never()`

↪ En conclusion

- Framework `Mockito` :
  - les fonctionnalités présentées permettent d'exprimer ce dont on a besoin pour une grande majorité de cas
  - possède d'autres possibilités permettant d'exprimer des comportements particuliers et autorise la personnalisation de différents éléments
  - les autres framework ont une syntaxe différente, mais le principe est le même (vérification par phase)

- Avantage des Doublures ou Mock :
  - permettent de réaliser bon nombre de vérifications relevant des tests d'intégration
  - utilisation relativement simple

- Inconvénient :
  - l'écriture se fait au sein des tests unitaires
  - risque de rendre difficile la ≪ maintenance ≫ des tests
- Bonne pratique :
  - se forcer, lors de l'écriture d'une classe de test, à ≪ séparer ≫ les méthodes de tests relevant des tests unitaires de celles intégrant les vérifications liées à l'intégration