Programmation Orientée Objet

Développement dirigé par les tests (TDD)

Lom Messan Hillah

Université Paris Nanterre
UFR SEGMI / MIAGE
2017

Objectifs

Étude des tests à travers les concepts clés suivants :

- Qu'est-ce qu'un test ?
- Développement dirigé par les tests (TDD)
- Test unitaire
- Frameworks de test
- Couverture de code

Sommaire

Sommaire

Qu'est-ce qu'un test ?

Une définition informelle[a]

[a] Adaptée de Oxford Dictionaries

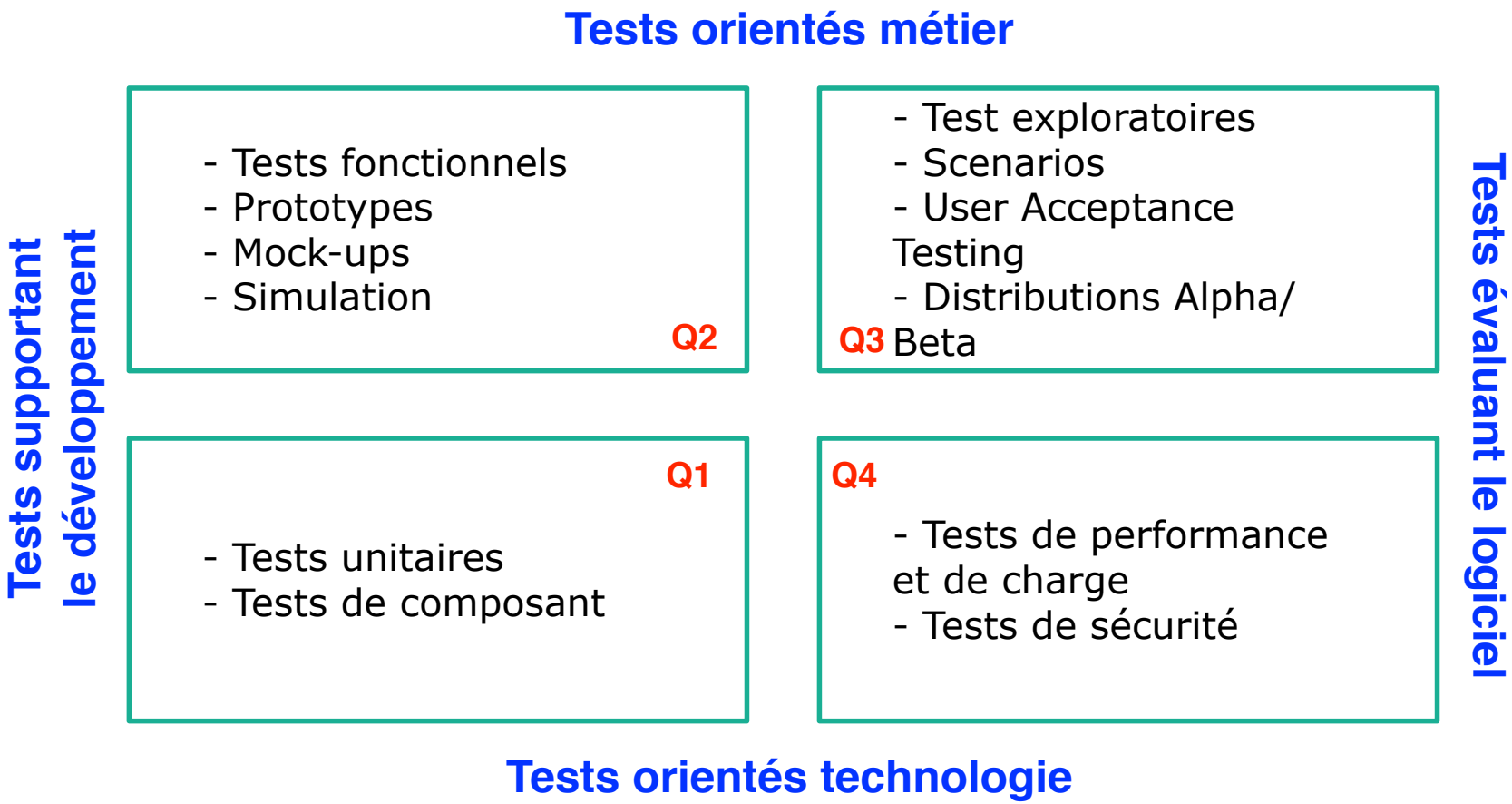
C'est une procédure d'évaluation critique, dans le but d'établir la qualité, la performance et la fiabilité d'un *système*, avant qu'il ne soit distribué pour utilisation.

- Dans notre contexte, il s'agit de *système logiciel*
- Cela sert à certifier un fonctionnement prévisible du logiciel
- Le système testé est appelé `System Under Test` (SUT)

Intérêt des tests

- Raccourcir le cycle de développement et de publication de nouvelles versions
  - `Release more software in less time`
- Supporter le développement itératif et incrémental
- Certifier un fonctionnement prévisible du logiciel
- Augmenter la confiance des développeurs et des clients dans le fonctionnement prévisible du logiciel
- Le système testé est appelé `System Under Test` (SUT)

Quadrants de test



Objectifs des tests

- **Tests orientés métier** : feedback continue de l'usage du produit par un panel d'utilisateurs ; aide à orienter le développement selon les besoin des utilisateurs
- **Tests orientés technologie** : critiques pour les choix de conception et d'architecture
- **Tests évaluant le produit** : aident à déterminer les améliorations à implanter dans le produit
- **Tests supportant le développement** : aident les développeurs à déterminer comment implanter les fonctionnalités demandées par le client

Tests orientés métier

- La spécification fournie par le client correspond-t-elle vraiment à ce que le client souhaite ?
- Les développeurs sont-ils en train d'implanter ce que souhaite réellement le client ?
- La valeur ajoutée du produit au métier du client correspond-t-elle à ce qu'il en attend ?

Tests orientés technologie

- Le système est-il performant, sécurisé ?
- Le système est-il fiable et robuste ?
- Les exigences non-fonctionnelles sont-elles considérées ?
  - Maintenance, modularité, réutilisation, capacité à être testé, etc.
- La conception est-elle pensée pour l'extension ?
- Le code peut-il être testé ? Réutilisé ?

Tests évaluant le produit

- Des améliorations peuvent-elles être implantées dans le produit ?
- Le produit final est-il meilleur que ceux de la compétition ?
- L'interface avec l'utilisateur fonctionne-t-elle bien ?

Tests supportant le développement

- Des améliorations peuvent-elles être implantées dans le produit ?
- Le produit final est-il meilleur que ceux de la compétition ?
- L'interface avec l'utilisateur fonctionne-t-elle bien ?
  - Utilisation de GUI mockups, wireframes

Utilité d'utilisation des quadrants

- Les résultats des tests évaluant le produit constituent un feedback intéressant pour les tests supportant le développement
- Utilise-t-on des test unitaires et d'intégration pour déterminer la bonne conception pour l'application ?
- Utilise-t-on une infrastructure d'automatisation de la production du logiciel, qui exécute systématiquement les tests pour un feedback rapide, et aider à conserver/améliorer la qualité de l'application ?
- Utilise-t-on les bons exemples du comportement du système pour écrire les tests ?

Niveaux de test

- **Unit testing** : test des fonctions/méthodes, ou composants individuels afin de s'assurer de leur bon fonctionnement et respect de leurs spécifications
- **Integration testing** : test de l'interaction entre composants du système afin de s'assurer de leur bonne intégration
- **System testing** : test de l'ensemble du système ou du logiciel complètement intégré, afin de s'assurer de son respect des spécifications
- **Acceptance testing** : test du système pour s'assurer qu'il est conforme aux exigences métier et évaluer s'il est acceptable pour être livré.

Types courants de test (1/2)

- **Smoke (Build verification) testing** : tests succincts visant à s'assurer du bon fonctionnement des fonctions les plus importantes du logiciel
- **Functional testing** : test du système sur les exigences fonctionnelles et les spécifications
- **Usability testing** : test du système sur sa facilité d'utilisation par l'utilisateur final
- **Security testing** : test du systèmes pour détecter les vulnérabilités, et s'assurer de la bonne protection des données contre les intrusions

Types courants de test (2/2)

- **Performance testing** : test de la réactivité du système et de sa stabilité face à des montées en charge (passage à l'échelle)
- **Regression testing** : test de la non-régression du système lors de mises à jour (améliorations ou corrections de bugs)
- **Compliance (conformance, regulation, standards) testing** : test de la conformité du système vis-à-vis de régulations ou standards internes ou externes.

Méthodes générales de test

- **Black box testing** : détails d'implémentations inconnus ; tests fonctionnels ou non fonctionnels ; equivalence partitioning, boundary value analysis
- White box testing : connaissance des détails d'implémentations ; control flow, data flow, branching, path
- **Gray box testing** : combinaison de black box and white box testing
- **Agile testing** : suit les principes du développement agile
- **Ad hoc** : pas de plannification, pas de documentation

Sommaire

Développement dirigé par les tests

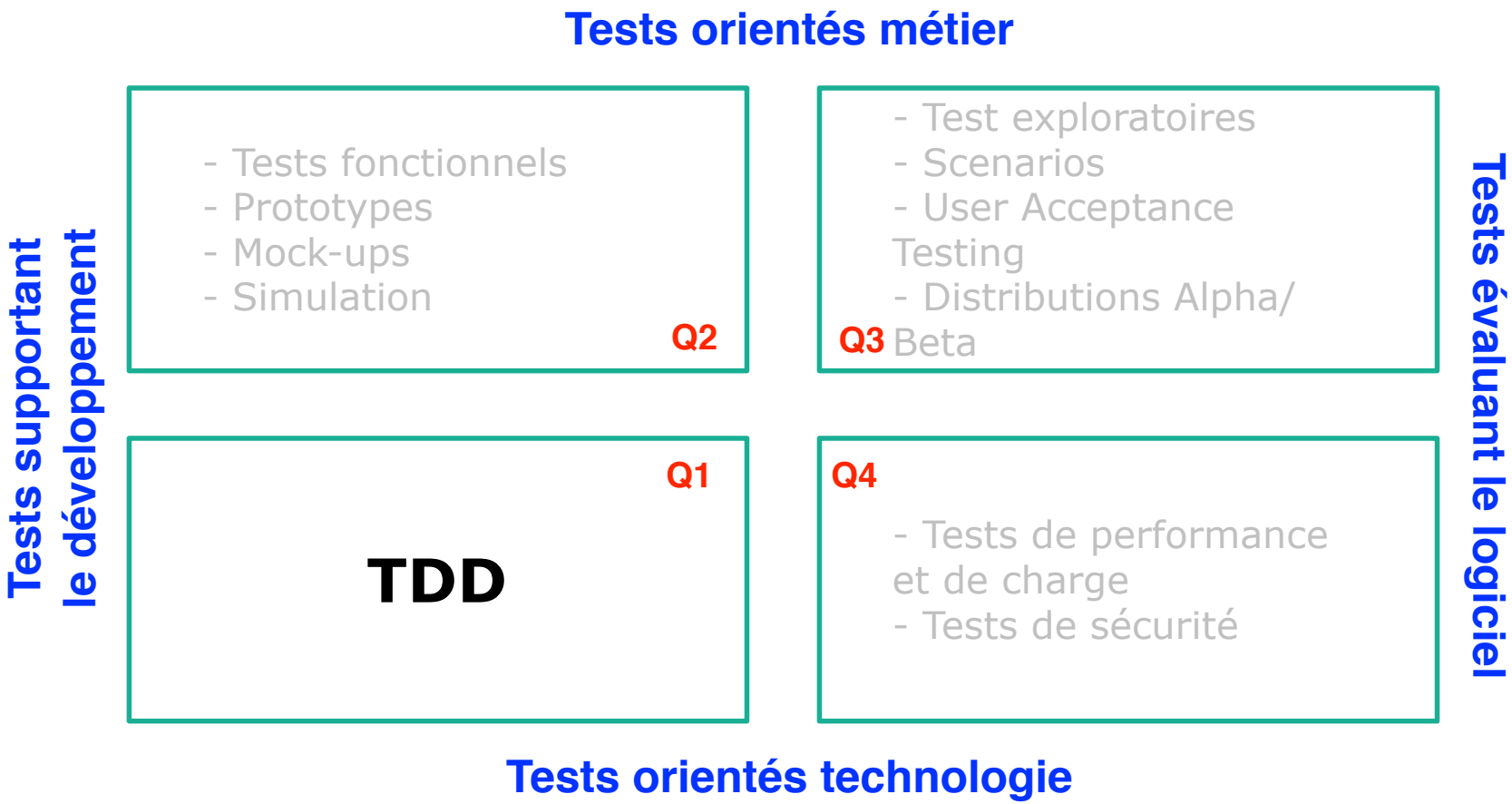
- L'idée est simple : **écrire les tests avant le code testé**
  - En anglais: Test-Driven Development (TDD)
- Pratique courante dans les méthodes agiles telles que Extreme Programming et Scrum
- TDD aide les développeurs à comprendre les fonctionnalités demandées et les délivrer de façon fiable et prévisible.

☞ Même si le test est une activité que les développeurs ont du mal à mettre en pratique, *vous n'avez rien d'autre à perdre que les bugs*.

Pourquoi `dirigé` ?

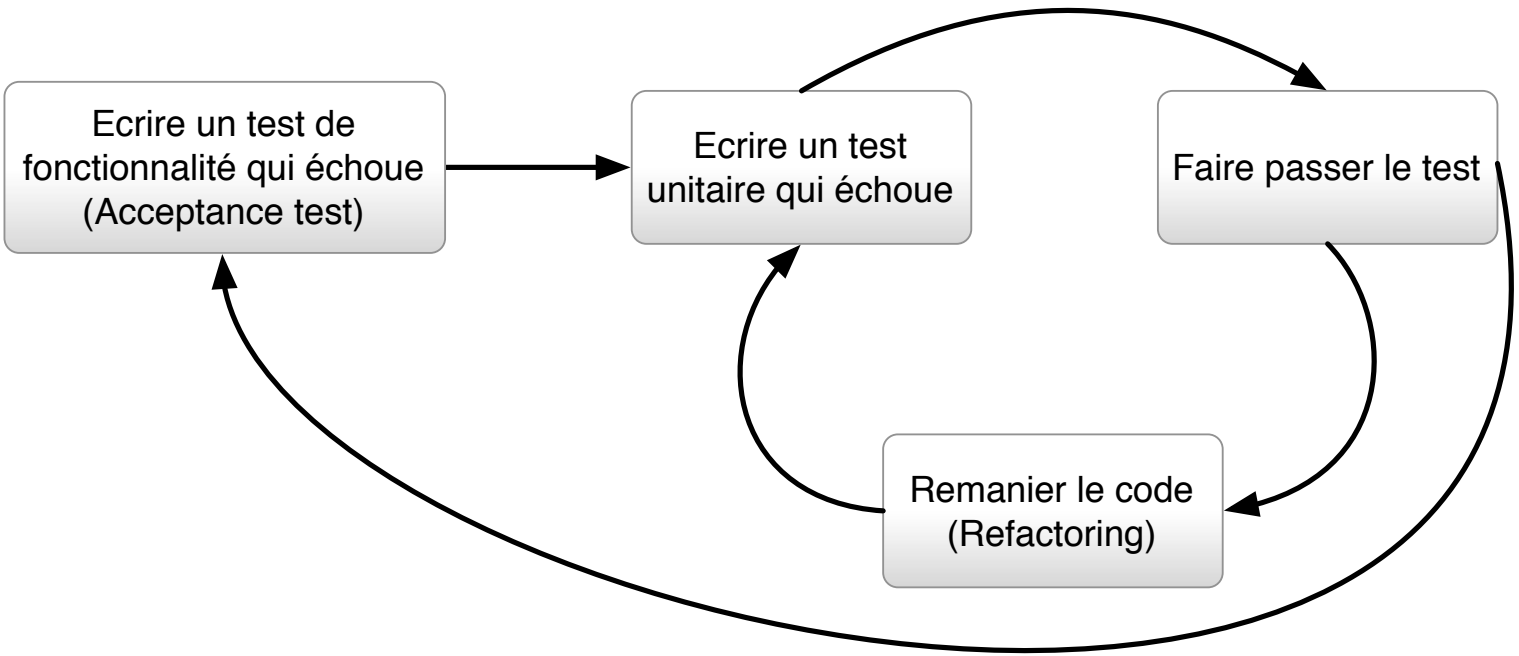
- Le test devient une **activité de conception**
  - Séparer la *conception logique* du code de sa *conception physique*
- Clarifier les idées sur les fonctionnalités demandées
  - Feedback rapide sur la qualité des choix de conception et de l'architecture
  - Gratification rapide au fil du développement, aide à acquérir plus de confiance dans l'amélioration du code
- Détecter rapidement et automatiquement les régressions pendant l'évolution du logiciel

Position des TDD dans les quadrants de test



Fondamentaux du TDD

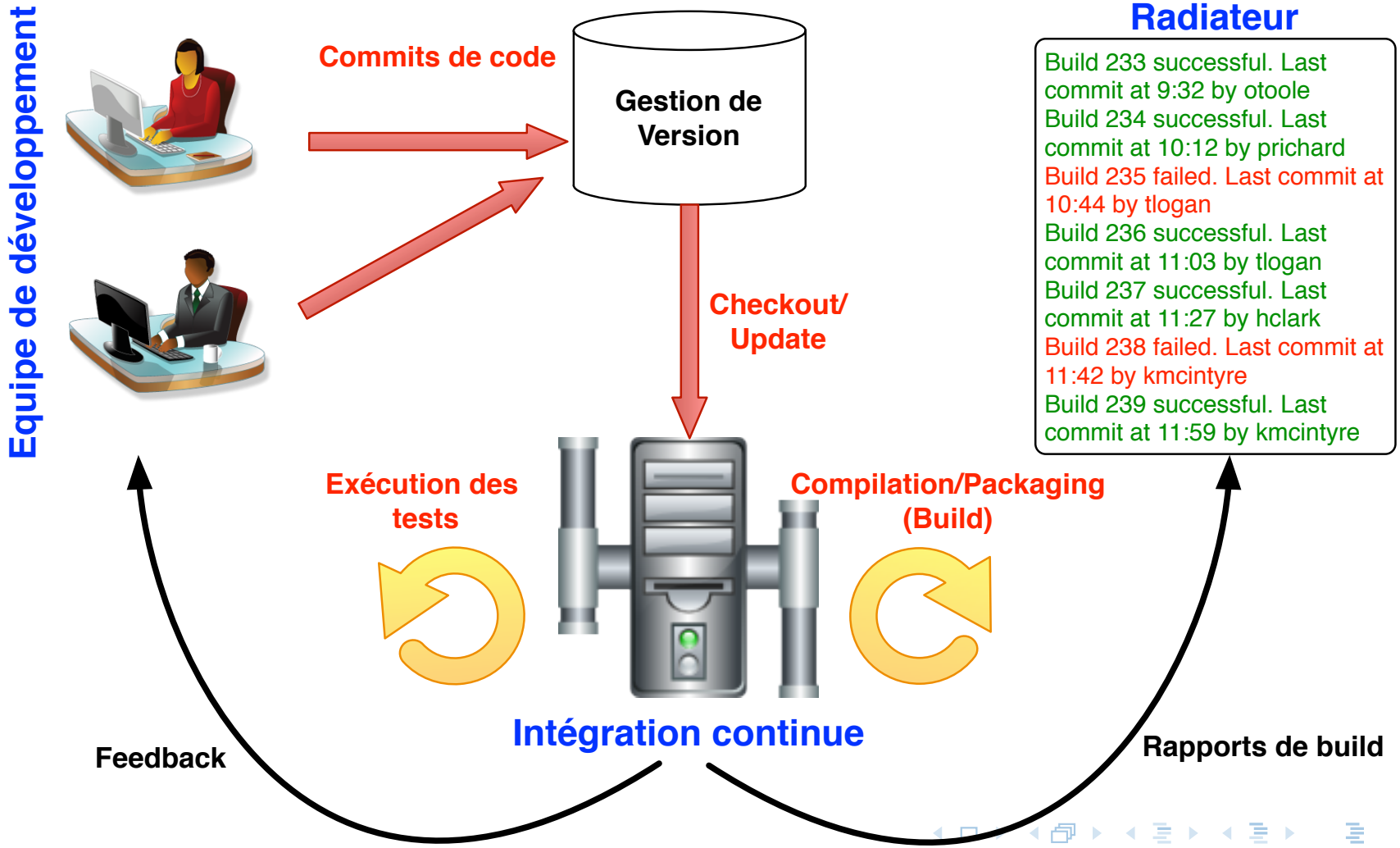
- Ne jamais écrire une nouvelle fonctionnalité sans avoir d'abord écrit le test correspondant qui échoue.
- Cela peut prendre du temps avant de faire passer un test de fonctionnalité qui échoue.



TDD dans un environnement de développement

- TDD doit être supporté par un ensemble de techniques et d'outils
  - Processus de production automatique du logiciel
    - Dans le monde Java, Maven est un outil de référence
  - Gestion de version (SVN, Git)
  - Intégration continue
    - p. ex. en utilisant Hudson, ou Jenkins, ou Bamboo, ou Teamcity, ou encore Apache Continuum.
  - Rapports de construction issus de l'intégration continue (Radiateurs)
  - Inspection continue de la qualité du code
    - p. ex. en utilisant SonarQube

TDD dans un processus d'intégration continue



Niveaux de test

- **Acceptance test** : le système complet fonctionne-t-il selon les exigences métiers? Il s'agit de tests fonctionnels, tests pour obtenir l'accord du client sur la prochaine fonctionnalité à développer
- **System test** : le système complet fonctionne-t-il selon les exigences fonctionnels et les spécifications ? dans l'environnement attendu ?
- **Integration test** : les composants interagissent-ils bien entre eux selon les interactions attendues ? le code développé fonctionne-t-il dans l'environnement logiciel (e.g. infrastructure d'exécution) tierce partie dans lequel il est intégré ?
- **Unit test** : les objets ont-ils le comportement attendu ?

Sommaire

Test unitaire

- Une partie du code est testée *en isolation* du reste du système. Dans notre contexte, ce sont les objets.
- L'interaction avec les autres objets n'est pas encore considérée
- Les dépendances peuvent être simulées avec les **mock objects**.
- Les tests unitaires aident à se focaliser sur les *bonnes pratiques* :
  - Encapsuler les variables (i.e ce qui varie)
  - Programmer par les interfaces
  - Cohérence (une seule responsabilité par classe)
  - Ne pas se répéter : toute fonctionnalité doit avoir une seule implémentation de référence bien identifiée

Intérêt des tests unitaires

- Evaluer le *couplage* et la *cohérence* des objets/composants
  - **Couplage** : deux objets sont très couplés si un changement dans l'un force une mise à jour dans l'autre (e.g, sinon le code ne compilera plus). Être capable de juste remplacer une implémentation tout en conservant l'interface abstraite chez l'objet utilisateur est une caractéristique de faible couplage.
  - **Cohérence** : liée à la responsabilité d'une classe, implantée complètement. Par exemple, une classe qui gère à la fois les URL et les dates n'est pas cohérente. Une classe qui ne gère qu'une partie du schéma des URL n'est pas cohérente non plus.
- Plus le couplage est faible, mieux c'est.
- Plus la cohérence est grande, mieux c'est.

Vocabulaire des tests

- Le système testé est appelé *System Under Test* (SUT)
- Un test est généralement appelé *Test Case*
- Un SUT peut être l'objet d'une suite de tests, *Test Suite*
- Une *assertion* est une fonction ou une macro qui vérifie le comportement ou l'état du SUT. Il s'agit généralement d'une condition logique, qui renvoie `vrai` pour un SUT fonctionnant comme attendu.
- Un *échec* est un problème que vous anticipez, dans le cadre d'un test, par rapport à un résultat attendu.
- Une *erreur* est plus dramatique, car elle résulte d'une condition d'erreur que vous n'avez pas prévue.

Sommaire

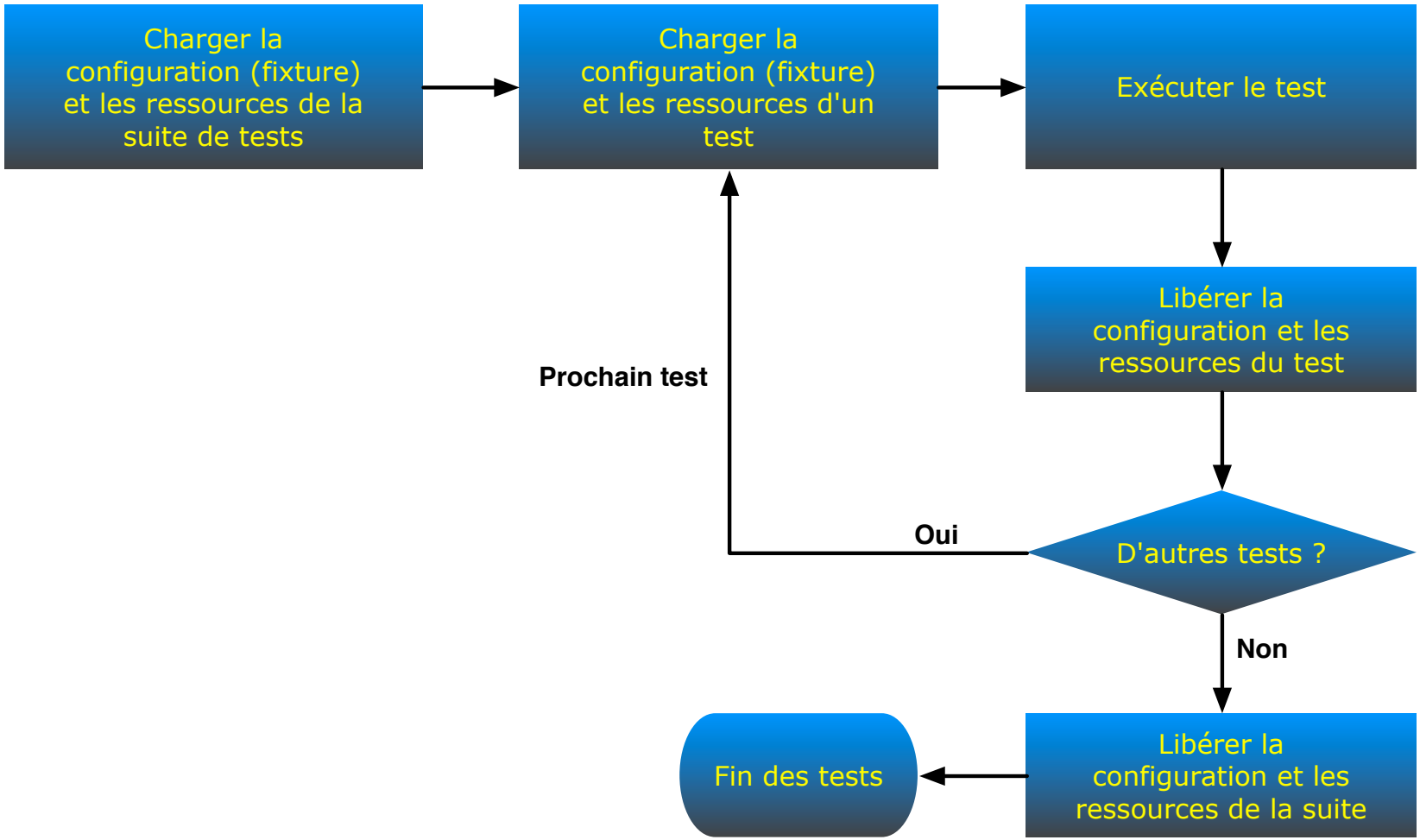
Frameworks de test

- L'ensemble des frameworks de test est généralement connu sous le nom de *xUnit* (http://www.martinfowler.com/bliki/Xunit.html)
- Dans le monde Java :
  - JUnit, TestNG
  - Et beaucoup d'autres : http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java
- Dans le monde .NET :
  - MSTest, NUnit, xUnit, MSpec
  - Spec#: https://www.microsoft.com/en-us/research/project/spec/
  - Et beaucoup d'autres : https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#.NET_programming_languages

Recette de test

- *Test fixture - Début* : configuration, initialisation d'objets et acquisition de resources (e.g. fichiers, flux réseaux) Désigne l'ensemble des opérations qui réalisent un ensemble de pré-conditions à l'exécution des tests. Généralement, il s'agit de placer le SUT et le système de test dans l'état qu'ils doivent avoir avant l'exécution des tests.
- *Test fixture - Fin* : configuration et libération de ressources. Désigne l'ensemble des opération réalisant les post-conditions suite à l'exécution des tests. Généralement, il s'agit de placer le SUT et le système de test dans l'état qu'ils doivent avoir après l'exécution des tests.
- *Test case* : cas de test
- *Check* : vérification du résultat attendu
- *Test suite* : suite de tests qui partagent la même configuration initiale.

Cycle de vie simplifié d'une classe de test JUnit



Structure d'une classe de test JUnit

```
1  public class ExempleTest {
2    @BeforeClass // Fixture de niveau classe, exécutée avant tout test
3    public static void setUpTest() {...}
4    @Before // Fixture de niveau méthode, exécutée avant chaque test
5    public void setUpTestCase() {...}
6    @Test // Méthode de test
7    public void testMethod1() {...}
8    @Test // Méthode de test
9    public void testMethod2() {...}
10   @After // Fixture de niveau méthode, exécutée après chaque test
11   public void tearDownTestCase() {...}
12   @AfterClass // Fixture de niveau classe, exécutée après tous les
       tests
13   public void tearDownTest() {...}
14 }
```

Démarrer

- Importer statiquement la classe `Assert`
  - `import static org.junit.Assert.*;`
- Écrire les fixtures
- Écrire les tests, donner un nom intelligible aux méthodes de test. Les annotez avec `@Test` (importer org.junit.Test)
- Pour tester effectivement le SUT, utiliser l'une des méthodes `assertXXX` de la classe `org.junit.Assert`

```
1  @Test
2   public void testAssertArrayEquals() {
3     byte[] expected = "trial".getBytes();
4     byte[] actual = "trial".getBytes();
5     org.junit.Assert.assertArrayEquals("failure - byte arrays not
        same", expected, actual);
6   }
```

Quelques méthodes de la classe `org.junit.Assert`

- `assertArrayEquals` : teste l'égalité de deux tableaux
- `assertEquals`: teste l'égalité de deux objets
- `assertFalse` : teste l'égalité d'une variable, du résultat d'une expression logique à la constante `faux`
- `assertNotNull` : vérifie qu'une référence est différente de `null`
- `assertNull` : vérifie qu'une référence est égale à `null`
- `assertNotSame` : vérifie que deux objets ne référencent pas le même objet
- `assertSame` : vérifie que deux objets référencent le même objet
- `assertThat` : vérifie qu'un objet est caractérisé par la condition exprimée en 2ème argument

Exceptions attendues et timeouts

```
1  @Test(expected=NullPointerException.class)
2  public void testShouldThrowNPE() {
3    Object someNullObject = null;
4    someNullObject.toString();
5  }
```

```
1  // timeout à1000 ms
2  // Echec déclenché par TimeoutException si l'exécution dépasse 1 sec
3  @Test(timeout=1000)
4  public void testShouldCompleteInLessThanOneSecond() {
5    // Un calcul complexe sur une donnée complexe...
6  }
```

```
1  // Timeout global - 10 sec max par méthode de test
2  @Rule
3  public Timeout globalTimeout = new Timeout(10000);
```

Exécuteurs de tests

- NetBeans, Eclipse et IntelliJ Idea intègrent nativement un exécuteur de tests avec interface graphique.
- L'annotation `@RunWith` permet de spécifier quel exécuteur l'on souhaite utiliser, à la place de l'exécuteur par défaut
  - `@RunWith(JUnit4.class)` invoque l'exécuteur par défaut de JUnit dans la version utilisée.
- Les exécuteurs spécialisés intégrés :
  - `Suite` : permet d'agréger soi-même des tests provenant de différentes classes en une suite de tests
  - `Parameterized` : permet de programmer des tests paramétrés
  - `Categories` : permet de spécifier un sous-ensemble de tests labélisés par des noms de catégories d'être inclus/exclus d'une suite de tests

Agréger des tests dans une suite

```
1  @RunWith(Suite.class)
2  // Inclusion de classes de test dans l'ordre suivant :
3  @Suite.SuiteClasses({
4    TestFeatureLogin.class,
5    TestFeatureLogout.class,
6    TestFeatureNavigate.class,
7    TestFeatureUpdate.class
8  })
9
10 public class FeatureTestSuite {
11   // cette class reste vide,
12   // elle est déclarée juste pour contenir les annotations ci-dessus
13 }
```

Tests paramétrés

```
1  @RunWith(Parameterized.class)
2  public class FibonacciTest {
3      @Parameters // Données pour le constructeur FibonacciTest
4      public static Collection<Object[]> data() {
5          return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, {
             2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 },{ 6, 8 } });
6      }
7      private int fInput;
8      private int fExpected;
9      // Chaque instance sera construite avec les données de data()
10     public FibonacciTest(int input, int expected) {
11         fInput= input;
12         fExpected= expected;
13     }
14     @Test
15     public void test() {
16         assertEquals(fExpected, Fibonacci.compute(fInput));
17     }
18 }
```

Catégories (1/2)

```
 1  public interface FastTests { /* marqueur de categorie */ }
 2  public interface SlowTests { /* marqueur de categorie */ }
 3  public class A {
 4    @Test
 5    public void a() { fail(); }
 6    @Category(SlowTests.class)
 7    @Test
 8    public void b() {}
 9  }
10  @Category({SlowTests.class, FastTests.class})
11  public class B {
12    @Test
13    public void c() {}
14  }
```

Catégories (2/2)

```
 1  @RunWith(Categories.class)
 2  @IncludeCategory(SlowTests.class)
 3  @SuiteClasses( { A.class, B.class }) // Les catégories constituent
      une sorte de suite de tests
 4  public class SlowTestSuite {
 5    // Exécutera A.b et B.c, mais pas A.a
 6  }
 7  @RunWith(Categories.class)
 8  @IncludeCategory(SlowTests.class)
 9  @ExcludeCategory(FastTests.class)
10  @SuiteClasses( { A.class, B.class }) // Les catégories constituent
      une sorte de suite de tests
11  public class SlowTestSuite {
12    // Exécutera A.b, mais pas A.a ou B.c
13  }
```

Pour en apprendre plus sur JUnit

https://github.com/junit-team/junit/wiki/Test-fixtures

https://github.com/junit-team/junit/wiki/Assertions

http://junit.sourceforge.net/javadoc/org/junit/Assert.html

https://github.com/junit-team/junit/wiki/Parameterized-tests

TestNG

- Un des meilleurs frameworks de test du monde Java
- Aussi puissant que JUnit
- Paramètres de tests, *DataProviders*
  - similaires aux tests paramétrés de JUnit
- Groupes de test, groupes de groupes, groupes d'exclusion, groupes partiels (niveau classe et niveau méthode entrelacés)
  - similaires aux catégories de JUnit
- Précédence / dépendance entre tests
- Tests en parallèle et timeouts
- Capacité d'exécution de tests JUnit
- Capacité d'invocation dynamique de TestNG depuis le code utilisateur
- Injection de dépendance
- Logging et rapports de test

Pour en apprendre plus sur TestNG

Une seule documentation de référence :

- http://testng.org/doc/index.html

Sommaire

Couverture de code

- Mesure de l'étendue de la couverture des instructions d'un programme par un test

Programmer à tester (SUT):

```
 1  public int foo(int a, int b) {
 2    int result = -1;
 3    if (a > 0 && b > 0) {
 4      result = a / b;
 5    }
 6    if (a <= 0 || b <= 0) {
 7      result = -1;
 8    }
 9    return result;
10  }
```

Couverture de code

Test pour le programme ci-dessus :

```
1  @Test
2  public void testFoo() {
3    SystemUnderTest sut = new SimpleSUTImplementation();
4    assertEquals(-1, sut.foo(-2, 3));
5  }
```

- A quel point le test ci-dessus couvre-t-il les instructions du programme plus haut (SUT) ?

Métriques de base de la couverture de code

- **Nombre de hits** : nombre de fois qu'une instruction a été exécutée pendant un test
- **Couverture de ligne** : pourcentage de lignes de code couvertes par le test, par rapport au nombre total de lignes à tester
- **Couverture de branchement** : pourcentage de branchements de code (dans les structures de contrôle) couverts par le test, par rapport au nombre total de branchements
- **Complexité** : complexité moyenne selon la complexité cyclomatique de McCabe (http://fr.wikipedia.org/wiki/Nombre_cyclomatique)

Exemple de couverture de branchement

```
1        ⊣       ⊣if (a > 0 && b > 0) {
2        ⊣
```

Valeurs de A	**Valeurs de B**	Couverture	Couverture cumulative
> 0	> 0	25%	25%
> 0	< 0	25%	50%
< 0	> 0	25%	75%
< 0	< 0	25%	100%

Ne pas confondre 100% de couverture et absence de bogues !

100% *code coverage* ≠ *bug free*

Une couverture de 100% ne signifie pas que le code est exempt de bogues. Il indique juste que les tests exécutent tous les instructions et branchements du code.
Cette métrique ne fait donc aucune hypothèse sur la bonne compréhension du problème et une **implantation correcte** de sa solution.

Complexité (1/2)

- Nombre de points de décision (donnés par les branchements) + 1

Le programmer à tester suivant possède une complexité égale à 5.

```
 1  public int foo(int a, int b) {
 2    int result = -1;
 3    if (a > 0 && b > 0) { // 2 points de décisions
 4      result = a / b;
 5    }
 6    if (a <= 0 || b <= 0) { // 2 points de décisions
 7      result = -1;
 8    }
 9    return result;
10  }
```

Complexité (2/2)

Les points de décision ci-dessus peuvent être réécrits en :

```
 1  if (a > 0) {
 2    if ( b > 0) {
 3      result = a / b;
 4    }
 5  }
 6
 7  if (a <= 0) {
 8    result = -1;
 9  }
10  if (b <= 0) {
11    result = -1;
12  }
```

Faible complexité, maintenance facilitée

Maîtriser la complexité des méthodes

Une complexité élevée est préjudiciable à la maintenance de votre application et augmente la probabilité d'apparition de bogues.
Une borne supérieure de 10 est la valeur couramment admise.

Outils de mesure de la couverture de code

- **Cobertura** : http://cobertura.github.io/cobertura/
  - Est intégré avec maven : `$ mvn cobertura:cobertura`
  - http://mojo.codehaus.org/cobertura-maven-plugin/
- **EclEmma** : Couverture de code d'application Java, pour intégration dans Eclipse
  - http://www.eclemma.org, un plugin pour Maven est également disponible (JaCoCo)
- **Clover**, produit commercial, en principe gratuit pour les projets open-source et les organisations à but non lucratif.
- **Coveralls:** outil en ligne
- **dotCover**: unit test runner and code coverage by JetBrains (integrates in Visual Studio)
- **NCover**: code coverage for .NET

Bibliographie

- Steve Freeman et Nat Pryce : *Growing Object-Oriented Software, Guided By Test*. Addison Wesley, ISBN-13 : 978-0321503626
- Martin Fowler et al. : *Refactoring: Improving the Design of Existing Code*. Pearson Education, ISBN-13 : 978-0201485677
- Kent Beck: *Simple Smalltalk Testing: With Patterns*, http://www.xprogramming.com/testfram.htm
- Atlassion: *Comparison of code coverage tools*, https://confluence.atlassian.com/clover/comparison-of-code-coverage-tools-681706101.html
- Rise4fun: https://rise4fun.com