



Module 3

Mise en œuvre des tests logiciels pour la qualité

Les types de tests en développement

- Tests unitaires
 - Test d'une fonctionnalité unique
 - Au niveau des fonctions/méthodes
 - Simples à écrire mais très nombreux
- Tests d'intégration
 - Test de composants
 - Vérification du comportement dans l'environnement
 - Serveur Web, Base de données, ...
 - Relativement simples à écrire (si outillés)
 - Mais l'environnement de test peut être complexe à mettre en place et à réinitialiser entre chaque tests
- Tests fonctionnels
 - Test de scénario utilisateur
 - Impliquent la rédaction de l'ensemble des scénarios et leur jeu (et rejeu)
 - Mettent en œuvre l'IHM
 - Donc plus ou moins facilement programmable
 - Longs en rédaction/enregistrement et ajustements

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud shape. A blue horizontal band with a white cloud outline is superimposed over the middle of the image.

Les tests unitaires

xUnit

- Le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme.
- xUnit désigne une famille de frameworks de rédaction et d'exécutions de tests unitaires.
 - Des déclinaisons existent pour de très nombreux langages !
 - JUnit : Java
 - PyUnit : Python
 - CppUnit : C++
 - NUnit : C#
- Imaginé et développé en Java par Kent Beck et Erich Gamma, auteurs des ouvrages "SmallTalk Best Practice Patterns" et "Design Patterns : Catalogue de modèles de conception réutilisables".
- Avantages :
 - Très simple d'utilisation
 - Utilisation graphique ou non
 - Intégrés aux outils de développement

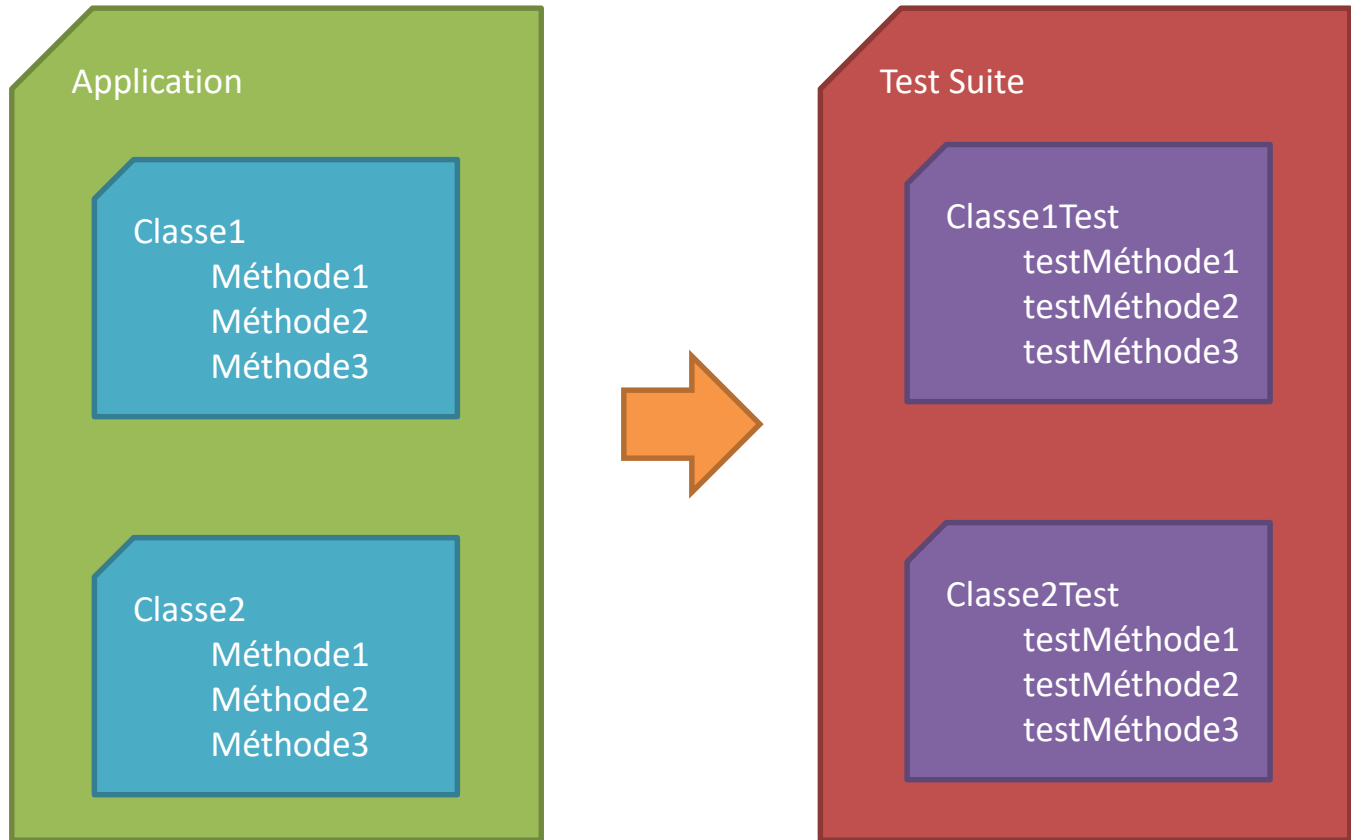
xUnit

- But:
 - Offrir au développeur un environnement de développement simple, le plus familier possible, et ne nécessitant qu'un travail minimal pour rédiger de nouveaux tests.
- Idée principale:
 - Représenter chaque test par un objet
 - Un test correspond souvent à une classe du programme
 - Un test pourra être composé de plusieurs tests unitaires dont le rôle sera de valider les différentes méthodes de vos classes

Les tests xUnit

- Les frameworks xUnit comporte les concepts suivants :
 - Le *Test Case* ou test unitaire
 - Il teste qu'une classe données répond aux exigences en matière de traitement
 - Le *Test*
 - Il correspond à l'évaluation du comportement d'une méthode, il se trouve dans un *Test Case*
 - La *Test Suite* ou suite de test
 - Elle permet de regrouper tous les tests dans une suite afin de lancer la batterie de tests

Symétrie Code / Test



Construction des tests

- Les tests sont implémentés sous forme de méthodes
 - Invocation de la méthode à tester
 - Vérification du résultat produit par rapport au résultat attendu
- Un test pour chaque scénario de méthode
 - Scénario principal
 - Scénario secondaire : levée d'exception, ...
- Il est souvent nécessaire de prévoir des traitements :
 - Avant tous les tests
 - Avant chaque test
 - Après chaque test
 - Après tous les tests

Implémentation des tests

- Prévoir la création de l'objet à tester
- Appeler la méthode à tester
- Utiliser des *assertions*
 - Les assertions sont des méthodes des frameworks de test permettant de comparer un résultat obtenu avec un résultat attendu
 - Plusieurs types d'assertions
 - Egalité, différence, nullité, non-nullité, ...
- Les assertions non vérifiées entraînent un échec du test

Les tests unitaires dans les langages objets

- En Java
 - JUnit reste la référence absolue !
 - Le tout premier...
 - Intégré aux IDE
 - TestNG
 - Offre une approche similaire à JUnit
 - Un peu plus déclaratif...
 - Permet nativement l'exécution de tests en parallèle
- En C#
 - NUnit
 - Référence historique, intégré à Visual Studio (addons)
 - MSTest
 - Intégré à Visual Studio

En Java - JUnit

- Les méthodes de test sont annotées avec `@Test`
 - Beaucoup de simplification par rapport à JUnit 3
 - pas besoin de dériver de `TestCase`
 - les assertions sont des méthodes statiques de la classe `org.junit.Assert`
 - les méthodes n'ont pas l'obligation d'être nommées en `testXxxx`
- Paramètres principaux de l'annotation
 - `expected` : classe `Throwable` attendue
 - `timeout` : durée maximale en millisecondes
 - `@Ignore` permet d'ignorer un test

JUnit - annotation @Test

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

public class TestsUnitaires
{
    @Test
    public void testAddition()
    {
        Calculatrice calc = new Calculatrice();
        assertTrue(5 == calc.additionner(2, 3));
    }
}
```

```
public class TestTimeout
{
    @Test(timeout = 1000)
    public void dureeRespectee()
    {}

    @Test(timeout = 1000)
    public void dureeNonRespectee() throws InterruptedException
    {
        Thread.sleep(10000);
    }
}
```

JUnit - Initialisation et nettoyage

- Initialisation et nettoyage des tests
 - Méthodes annotées avec `@Before` ou `@After`
 - Méthodes exécutées avant et après chaque méthode de test
 - Méthode publique
 - Plusieurs méthodes peuvent être annotées avec la même annotation
 - L'ordre d'invocation des méthodes annotées est indéterminé
- Initialisation et nettoyage du cas de test
 - Méthodes annotées avec `@BeforeClass` ou `@AfterClass`
 - Méthodes exécutées avant l'invocation du premier test de la classe, et après le dernier test de la classe
 - Méthodes publiques et statiques
 - Une seule méthode par annotation

JUnit - Exemple

```
public class TestFixture
{
    @BeforeClass public static void montageClasse()
    {
        System.out.println(">> Montage avant tous Les tests");
    }

    @AfterClass public static void demontageClasse()
    {
        System.out.println(">> Démontage après tous Les tests");
    }

    @Before public void montage()
    {
        System.out.println("----- AVANT");
    }
    @After public void demontage()
    {
        System.out.println("----- APRES");
    }

    @Test public void test1()
    {
        System.out.println("Test 1");
    }

    @Test public void test2()
    {
        System.out.println("Test 2");
    }
}
```

JUnit - Les suites de tests

- Classe vide annotée avec
 - `@RunWith`, possibilité de changer la classe d'exécution de la suite
 - `Suite.class` par défaut
 - `@SuiteClasses(Class[])` pour former la suite de tests

```
@RunWith(Suite.class)
@SuiteClasses({ AppTest.class, MoneyTest.class })
public class AllTests {

}
```

En C# - NUnit

- Il est courant en C# d'avoir un projet différent pour le code des tests
 - Il doit alors référencer le projet contenant les classes à tester
- Les classes de tests doivent posséder l'attribut [TestFixture]
 - L'instruction `using NUnit.Framework` permet de l'utiliser
- Les méthodes de tests doivent posséder l'attribut [Test]

NUnit - Exemple

```
using NUnit.Framework;

namespace BanqueTest {

    [TestFixture]
    public class ClientTest {

        ...

        [Test]
        public void testClient() {
            Assert.IsNotNull(client, "L'objet client est null");
        }

        ...

    }
}
```

NUnit - Initialisation et nettoyage

- Initialisation et nettoyage des tests
 - Méthodes avec l'attribut avec [SetUp] ou [TearDown]
 - Méthodes exécutées avant et après chaque méthode de test
 - Méthodes publiques
- Initialisation et nettoyage du cas de test
 - Méthodes avec l'attribut [OneTimeSetUp] ou [OneTimeTearDown]
 - Méthodes exécutées avant l'invocation du premier test de la classe, et après le dernier test de la classe
 - Méthodes publiques

NUnit - Exemple

```
using NUnit.Framework;

namespace BanqueTest {

    [TestFixture]
    public class ClientTest {

        [OneTimeSetUp]
        public void setUpClass() {
            ...
        }

        [SetUp]
        public void setUp() {
            ...
        }

        [TearDown]
        public void tearDown() {
            ...
        }

        [OneTimeTearDown]
        public void tearDownClass() {
            ...
        }
    }
}
```

Le traitement des exceptions

- Concernant la gestion des exceptions, il existe deux situations
 - Les exceptions non-attendues
 - Qui feront donc échouer le test
 - Dans le cas d'un scénario principal
 - Les exceptions attendues
 - Qui doivent survenir pour que le test passe !
 - Cas typique d'un scénario secondaire

Les exceptions non-attendues

- Deux possibilités :
 - Ne pas les gérer
 - Si elles se présentent, elle feront de toute façon échouer le test
 - Les gérer
 - Pour une analyse potentiellement plus fine des raisons d'échec
- Pour gérer les exceptions, on fait explicitement échouer le test dans un catch

Les exceptions attendues

- Le fait qu'une exception survienne est prévu par le test
 - Si elle n'est pas levée, le test est en échec
- Les frameworks de test proposent une telle mise en œuvre pour permettre de simplifier l'écriture des scénarios secondaires des tests
 - On est assuré que le test est en échec si l'exception n'est pas déclenchée !

JUnit - Les exceptions attendues

- On indique le type de l'exception attendue avec l'attribut `expected` de l'annotation `@Test`
 - La méthode de test doit faire remonter l'exception avec `throws`

```
@Test(expected=LocationException.class)
```

```
public void testLouerPasOk() throws LocationException {  
    Client client1 = new Client("DUPONT", "Robert", LocalDate.of(2016, 10, 21));  
    voiture.louer(client1, 300);  
}
```

NUnit – Les exceptions attendues

- NUnit dispose d'une assertion spécifique permettant de s'assurer qu'une exception sera bien levée par une instruction
 - `Exception Assert.Throws(`
 Type expectedExceptionType, TestDelegate code
)
 - `Exception Assert.Throws(`
 Type expectedExceptionType, TestDelegate code,
 string message, params object[] params
)
- La mise en œuvre peut se faire de différente manière :
 - En utilisant une méthode déléguée
 - En utilisant un délégué anonyme
 - Avec une expression lambda

NUnit – Assert.Throws()

- En utilisant une méthode déléguée

```
[Test]
public void Tests() {
    Assert.Throws<ArgumentException>(MethodThatThrows);
}

public void MethodThatThrows() {
    throw new ArgumentException();
}
```

- En utilisant un délégué anonyme

```
Assert.Throws<ArgumentException>(
    delegate { throw new ArgumentException(); });
```

- Avec une expression lambda

```
Assert.Throws<ArgumentException>(
    () => { throw new ArgumentException(); });
```

Les assertions

- Les assertions servent à valider une instance d'objet ou une valeur
- Par exemple, on va valider que le prénom de l'utilisateur est bien « Robert »
- Les frameworks xUnit fournissent des assertions de base (égalité, non-nullité, etc...)
 - Pour simplifier l'écriture des assertions on pourra aussi s'équiper d'un outillage complémentaire
 - Bibliothèques additionnelles ajoutant des assertions
- Lors de l'utilisation des assertions, il est important de respecter l'ordre des paramètres pour ce qui est des valeurs attendues et des valeurs produites
 - Les messages d'erreurs en seront plus compréhensibles !

JUnit - Assertions

- Ce sont des méthodes statiques de la classe **org.junit.Assert**
 - On peut donc les rendre toutes disponibles par un import statique
 - Exemple :
 - `assertEquals(expected, printedCart);`
- Référence :
 - junit.org/junit4/javadoc/4.12/org/junit/Assert.html

NUnit - Assertions

- La classe Assert propose les assertions standards de NUnit
 - Elle est disponible grâce à `using NUnit.Framework;`
- Référence :
 - docs.nunit.org/articles/nunit/writing-tests/assertions/assertion-models/classic.html

A grayscale photograph of a person in a business suit, with their hands cupped together in front of them. Overlaid on the image is a large, stylized white cloud with a thick white outline. A horizontal blue bar with rounded ends cuts across the middle of the cloud, serving as a background for the title text.

Mock Objects

Tests unitaires vs. Tests d'intégration

- Tests unitaires
 - On se concentre sur des fonctionnalités simples (méthodes, fonctions, ...)
- Tests d'intégration
 - On vérifie le comportement dans un environnement réel (base de données, serveur Web, ...)
- Si les composants d'accès aux données sont testés avec une base de données réelle, ce sont des tests d'intégration.
 - Si le test échoue, est-ce la faute de la logique métier du composant ?
 - Ou bien la faute de l'environnement ?
- Les deux types de tests sont nécessaires
 - On ne teste pas la même chose à chaque fois !

La problématique des tests unitaires

- D'un point des vues des tests unitaires certains objets peuvent être difficile à tester
 - Ils dépendent d'objets complexes, externes, ...
 - On pourrait penser qu'il ne sont pas testable « unitairement »
- Pour pouvoir tester ce type d'objet, il faut simuler son environnement.
 - Créer des objets factices permettant de satisfaire aux conditions attendues par l'objet à tester.
 - Une fois ces objets mis en place autour de l'objet à tester, on pourra se concentrer sur le test des fonctionnalités
 - On parle de « bouchonnage »

Des Stubs ou des Mocks ?

- En Français on confond les deux termes sous le nom de « bouchon »
- Un Stub : une implémentation « vide » d'une dépendance
 - Exemple : pour un DAO, faire une implémentation qui n'accède pas en base de données mais renvoie toujours les mêmes valeurs
- Un Mock : une implémentation générée par une librairie spécialisée, qui la crée à la volée en fonction de l'interface à respecter.
 - Le mock vérifie également le comportement (nombre d'appels, etc...)
- martinfowler.com/articles/mocksArentStubs.html

Exemple de Stub

- Le Stub implémente la même interface que la dépendance injectée
 - Le Stub peut être une classe anonyme, pour éviter de créer trop de fichiers
 - Cela peut être également une vraie classe, afin de pouvoir le réutiliser sur plusieurs tests

```
AccountService accountService = new AccountService();
accountService.setUserService(new UserService() {
    public User getCurrentUser() {
        User user = new User();
        user.setLogin("test");
        user.setFirstName("Robert");
        return user;
    }
});
assertEquals(
    accountService.getCurrentUser().getFirstName(),
    "Robert"
);
```

Le concept des Mock Objects

- Les frameworks de Mock permettent de générer automatiquement une implémentation à partir d'une interface ou d'une classe abstraite
 - Le but est souvent de simuler des objets apportés par des librairies
- Cette implémentation fonctionne ensuite comme un magnétoscope :
 - On liste les méthodes qui vont être appelées
 - Phase « *expect* »
 - On dit ensuite au framework que l'on va jouer ce scénario
 - Phase « *replay* »
 - En fin de test, on demande au framework de valider que le scénario s'est déroulé comme prévu
 - Phase « *verify* »

Les frameworks de Mock Objects

- En Java
 - 3 frameworks principaux
 - EasyMock
 - JMock
 - Mockito
 - Mockito est aujourd'hui le plus populaire
 - Syntaxe plus simple à apprendre
 - Plus concis
- En C#
 - Moq
 - Le plus répandu
 - FakeItEasy

Exemple en Java - Mockito

```
public interface ICalculator {  
    double add(double num1, double num2);  
    double subtract(double num1, double num2);  
    double multiply(double num1, double num2);  
    double divide(double num1, double num2);  
}
```

```
public class TestCalculator {  
  
    @Mock  
    private ICalculator calculator;  
  
    @Before  
    public void setUp() throws Exception {  
        MockitoAnnotations.initMocks(this);  
    }  
  
    @Test  
    public void passingTest() {  
        when(calculator.add(2.0, 2.0)).thenReturn(4.0);  
        assertEquals(4.0, calculator.add(2.0, 2.0), 0);  
    }  
}
```

Exemple en C# - Moq

```
public interface Icalculator
{
    decimal Add(decimal num1, decimal num2);
    decimal Subtract(decimal num1, decimal num2);
    decimal Multiply(decimal num1, decimal num2);
    decimal Divide(decimal num1, decimal num2);
}
```

```
[TestFixture]
public class CalculatorTests
{
    [Test]
    public void PassingTest()
    {
        var calculator = new Mock<Icalculator>();
        calculator.Setup(x => x.Add(2, 2)).Returns(4);
        Assert.AreEqual(4, calculator.Object.Add(2, 2));
    }
}
```

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud graphic. A blue horizontal band is superimposed over the middle of the image, containing the title text.

Les tests d'intégration

Les tests d'intégration

- Les frameworks xUnit permet aussi de concevoir et de lancer les tests d'intégration
 - Les principes d'évaluation restent sensiblement les mêmes vis-à-vis des évaluations à réaliser
- Ils permettent donc aussi de lancer leur exécution
 - Mais ce ne sont pas des « tests unitaires »
 - Mais il est trop souvent commun de les catégoriser comme des tests unitaires

Rappels sur les tests (unitaire vs. Intégration)

- Test d'un composant unique (métier ou technique), en isolation du reste des autres composants : ce sont les **tests unitaires**
 - Cela exclut l'utilisation de bases de données, serveurs Web, ...
- Test d'un ensemble de composants dans un environnement comparable à la production : ce sont les **tests d'intégration**
 - Cela inclut l'utilisation de bases de données, serveurs Web, etc... sans doute avec une configuration d'infrastructure spécifique

L'intégration

- Assembler plusieurs composants logiciels élémentaires pour réaliser un composant de plus haut niveau.
 - Utiliser une classe **Client** et une classe **Produit** pour créer un module de commande sur un site marchand, c'est de l'intégration !
- Un test d'intégration vise à s'assurer du bon fonctionnement de la mise en œuvre conjointe de plusieurs unités de programme, testés unitairement au préalable.

La mise en place des tests d'intégration

- Certains composants doivent être testés vis-à-vis de l'environnement avec lequel ils vont interagir
 - Base de données, Serveurs Web par exemple
- Il est donc nécessaire de pouvoir :
 - Déployer les composants dans un serveur Web
 - Mettre en place un jeu de données de test dans une base de test
- Des outils complémentaire peuvent être nécessaire pour ensuite écrire du code utilisant des composants déployés
 - Ecrire une navigation Web par exemple

Mise en place de jeux de données

- Un jeu de données va permettre de créer un environnement propice au test lorsque celui-ci doit manipuler des données en base ou dans un autre référentiel (Document XML, ...)
- Pour un composant testant des méthodes d'accès aux données, il est impératif de prévoir
 - L'injection d'un jeu de données dans la base de test **avant chaque test**
 - [SetUp] / @Before
 - La suppression du jeu de données de la base **après chaque test**
 - [TearDown] / @After
- Des scripts devront donc être prévus à cet effet pour éviter d'avoir à « programmer » ces actions
 - Des outils additionnels aux frameworks de test pourront lancer ces scripts avant et après chaque test

En résumé...

- **Tests d'intégration composants**

- Valident que les unités de code (testées unitairement) collaborent réellement
 - Comme avec les mocks

- **Tests d'intégration système**

- Tests d'intégration composants qui testent au sein d'un système en analysant le fonctionnement avec d'autres collaborateurs au niveau du système (bases de données, ...)

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized cloud graphic. The cloud has a white outline and a semi-transparent blue center. A dark blue horizontal band is overlaid across the middle of the cloud, containing the title text.

Les tests fonctionnels

Les tests fonctionnels

- Les tests fonctionnels permettent de valider les scénarios d'usage de l'application par les utilisateurs finaux
 - Il est donc nécessaire d'écrire les scénarios avant d'écrire les tests !
 - Les tests impliquent donc l'IHM !
- La difficulté des tests fonctionnels réside dans le fait de pouvoir piloter l'IHM
 - Application Console
 - Simuler des frappes clavier
 - Application Graphique
 - Programmer des interactions sur les composants graphiques
 - Application Web
 - Piloter le navigateur Web

Tests fonctionnels des applications Web

- Les contraintes :
 - Tests sur plusieurs combinaisons :
 - Navigateur
 - Système d'exploitation
 - Tests sur les périphériques mobiles
- Les tests fonctionnels des applications Web nécessitent :
 - Le déploiement de l'application dans un serveur (de test)
 - Un outillage spécifique pour contrôler différents navigateurs Web
 - Selenium, Katalon, ...

La suite d'outils Selenium

- www.selenium.dev
- Selenium est constitué d'un ensemble d'outils pour l'enregistrement et l'exécution de tests Web
 - Selenium IDE
 - Un plugin pour navigateur Web (Firefox / Chrome)
 - Permet d'enregistrer un scénario de navigation
 - Export du scénario : Langage + Framework de test
 - Selenium Web Driver
 - Permet l'exécution des tests enregistrés pour différents navigateurs
 - API de pilotage
 - Selenium Grid
 - Répartition de l'exécution des tests sur une « grille » de machines avec des « capabilities » variables
 - Capabilities = OS + Navigateur

Selenium IDE

Selenium IDE - Element Verification* - Mozilla Firefox

Project: Element Verification*

Executing ▾

TextVerification* https://www.facebook.com

	Command	Target	Value
1	open	//login	
2	verify element not present	id=xyz	
3	verify element present	id=email	
4	verify element present	id=xyz	

Command // Target // Value

Runs: 1 Failures: 1

Log Reference

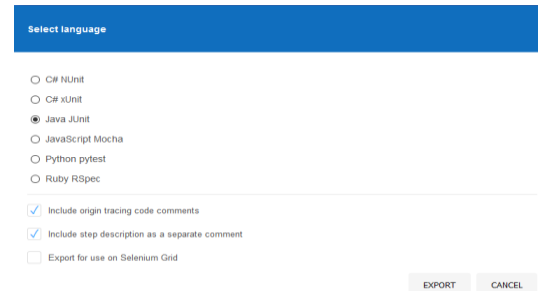
Running 'TextVerification' 21:46:48

- 1. open on //login OK 21:46:49
- 2. verifyElementNotPresent on id=xyz OK 21:46:49
- 3. verifyElementPresent on id=email OK 21:46:50
- 4. verifyElementPresent on id=xyz Failed: 21:46:50
Element with locator id=xyz could not be found

'TextVerification' ended with 1 error(s) 21:46:50

Fonctionnement de Selenium IDE

- En mode enregistrement, il permet de capturer les actions à partir d'une URL de base
 - Toutes les actions sur l'application Web sont enregistrées sous forme de « Selenese »
- Une fois l'enregistrement terminé, on peut rejouer et ajuster le test
 - Reprendre des séquences
 - Revenir sur certaines localisations d'éléments
 - Epurer les « rebonds »
 - Compléter des assertions
- Lorsque le scénario est terminé, il est exportable sous forme de code
 - Langage + Framework de test



The screenshot shows the 'Select language' dialog box in Selenium IDE. It features a blue header with the title 'Select language'. Below the header, there is a list of radio buttons for selecting a language/framework: C# NUnit, C# xUnit, Java JUnit (which is selected), JavaScript Mocha, Python pytest, and Ruby RSpec. Below the radio buttons, there are three checkboxes: 'Include origin tracing code comments' (checked), 'Include step description as a separate comment' (checked), and 'Export for use on Selenium Grid' (unchecked). At the bottom right, there are two buttons: 'EXPORT' and 'CANCEL'.

Selenium WebDriver

- Selenium WebDriver permet d'automatiser l'exécution des scénarios de test fonctionnel enregistrés avec Selenium IDE et exportés en code
- Il s'appuie sur les éléments suivants :
 - Les tests sous forme de code
 - Un « Language Binding »
 - L'API Selenium pour un langage donné
 - Des pilotes de navigateurs Web
 - Des exécutables natifs qui permettent à l'API Selenium de contrôler les navigateurs
 - Optionnellement, le Selenium Server
 - Si les navigateurs doivent être pilotés sur des machines distantes

Selenium WebDriver - Exemple

- Test Selenium Java + JUnit

```
public class ThatWebSite {  
  
    private WebDriver driver;  
    private String baseUrl;  
  
    @Before  
    public void setUp() throws Exception {  
        driver = new FirefoxDriver();  
        baseUrl = "http://www.thatwebsiteundertest.com/";  
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);  
    }  
  
    @Test  
    public void testStatsPage() throws Exception {  
        driver.get(baseUrl + "/stats");  
        driver.findElement(By.id("loadajax")).click();  
  
        assertEquals(  
            "This text should be present on the page",  
            driver.findElement(By.cssSelector("#ajaxdiv > p")).getText()  
        );  
    }  
  
    @After  
    public void tearDown() throws Exception {  
        driver.quit();  
    }  
}
```

A grayscale photograph of a person in a dark suit and white shirt, holding a large, stylized cloud graphic. The cloud has a white outline and a blue gradient fill. The text 'Les bonnes pratiques du test' is written in white on the blue part of the cloud.

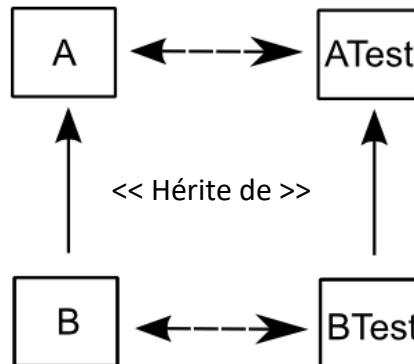
Les bonnes pratiques du test

Structuration du code de test

- L'architecture du code des tests est en fait facile à concevoir :
- **Les tests doivent avoir une architecture symétrique au code.**
- Une règle très simple qui a un impact important sur le code.
 - Négliger cette règle constitue un risque fort pour la batterie de tests qui deviendra incompréhensible et impossible à maintenir.
- Cette symétrie s'entend à tous les points de vue :
 - organisation des fichiers,
 - des packages,
 - des classes.

Symétries et conventions

- Des fichiers
 - cas d'usage à systématiser : l'organisation d'un projet Maven
 - `src/main/java` : Le code
 - `src/test/java` : Les tests
- De nommage
 - Les classes de tests seront rangées dans les mêmes paquets que les classes testées.
 - Cela permettra d'accéder facilement à la portée du paquet et en particulier aux membres protégés.
- D'architecture



Classer les tests

- Il est important que la batterie de tests reste rapide à exécuter pour fournir un feedback quasiment immédiat aux développeurs au risque de laisser passer des régressions.
 - Dès que le développeur n'attend plus après le résultat pour passer à autre chose, c'est que la durée d'exécution devient trop longue.
- L'exécution des tests rapides doit permettre de donner un niveau de confiance suffisant pour prévenir une régression importante.
- L'exécution des tests lents permettra une couverture complète et un retour exhaustif sur la santé du code.
- Il appartient aux développeurs de définir autant le contenu que le rythme de chaque batterie.
 - Il est assez classique de lancer les tests rapides très souvent dans la journée
 - Les tests lents seulement quelques fois par jour, voir une seule fois lors de la construction nocturne.
- L'utilisation de catégories ou la fabrication de suites de tests adaptées permettra cela.

Refactoring de tests

- Remanier le code des tests sous-entend que nous devons le modifier.
 - Quand il s'agit de remanier le code d'exécution, les tests protègent contre les régressions.
 - Mais lorsqu'il s'agit de modifier les tests, qui les protège contre les régressions ? Il faut donc pouvoir tester les tests.
- **Si le test valide le code, le code valide le test.**
- Il ne faut jamais refactoriser la batterie de test et le code en même temps.
- Pour passer de l'un à l'autre, il faut toujours s'assurer que la batterie de test passe entièrement.

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud shape. A blue horizontal band with a white cloud-like border is superimposed over the middle of the image.

La couverture de tests

La couverture de tests

- La couverture de test, aussi appelée **couverture de code** (*code coverage* en anglais) est la mesure du **taux de code source testé** dans un système informatique.
 - C'est un pourcentage !
- Intérêts
 - Donne une valeur quantitative représentant la part de logiciel testé
 - Apporte une visualisation du code exécuté
 - Mais surtout du code non exécuté !
 - Permet de mieux maîtriser le risque : Le code non testé est connu.
 - Permet d'améliorer le test lui-même.

Calcul de la couverture de test

- Il y a de nombreuses méthodes pour mesurer la couverture de code. Les principales sont :
 - Couverture des fonctions (*Function Coverage*)
 - Chaque fonction dans le programme a-t-elle été appelée ?
 - Couverture des instructions (*Statement Coverage*)
 - Chaque ligne du code a-t-elle été exécutée et vérifiée ?
 - Couverture des points de tests (*Condition Coverage*)
 - Toutes les conditions (tel que le test d'une variable) sont-elles exécutées et vérifiées ? (Le point de test teste-t-il ce qu'il faut ?)
 - Couverture des chemins d'exécution (*Path Coverage*)
 - Chaque parcours possible (par exemple les 2 cas *vrai* et *faux* d'un test) a-t'il été exécuté et vérifié ?

La couverture fonctionnelle

- Couverture de code = un indicateur parmi d'autres
- Se focaliser sur la couverture fonctionnelle procurée par les tests mis en œuvre.
 - Est-ce que les test unitaires développés permettent de vérifier 80% des fonctionnalités du système
 - Ou bien de vérifier de manière très fine 5% du système ?
- Une couverture fonctionnelle forte est généralement un gage de réussite en matière de testing : **Il faut tout tester.**
 - Pas forcément dans les moindres recoins, mais testez tous les modules.

Les outils de mesure de la couverture de test

- Les outils de mesure de la couverture de test sont le plus souvent intégrés aux plateformes d'intégration continue
 - Nous y reviendrons donc ;)
- Il faut un outillage qui instrumente le code lors des tests, et qui est capable de générer un rapport
 - Cobertura
 - cobertura.sourceforge.net
 - Sonar Qube
 - www.sonarqube.org

Travaux Pratiques



www.eni-service.fr

Travaux pratiques

- Tests Unitaires / Intégration
 - Création de projet Java / C#
 - Codage de fonctionnalités simples
 - Ajout de tests unitaires
 - Exécution des tests unitaires
- Tests Fonctionnels
 - Création et enregistrement d'un scénario Web avec Selenium IDE
 - Export en Java / C#
 - Intégration du code dans un projet Java / C#
 - Mise en œuvre de Selenium WebDriver pour automatiser l'exécution du scénario



Annexes

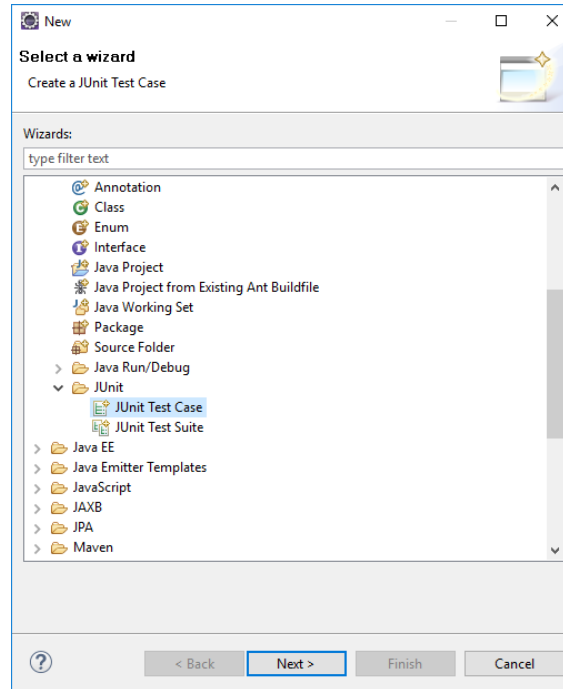


Création de tests JUnit avec Eclipse

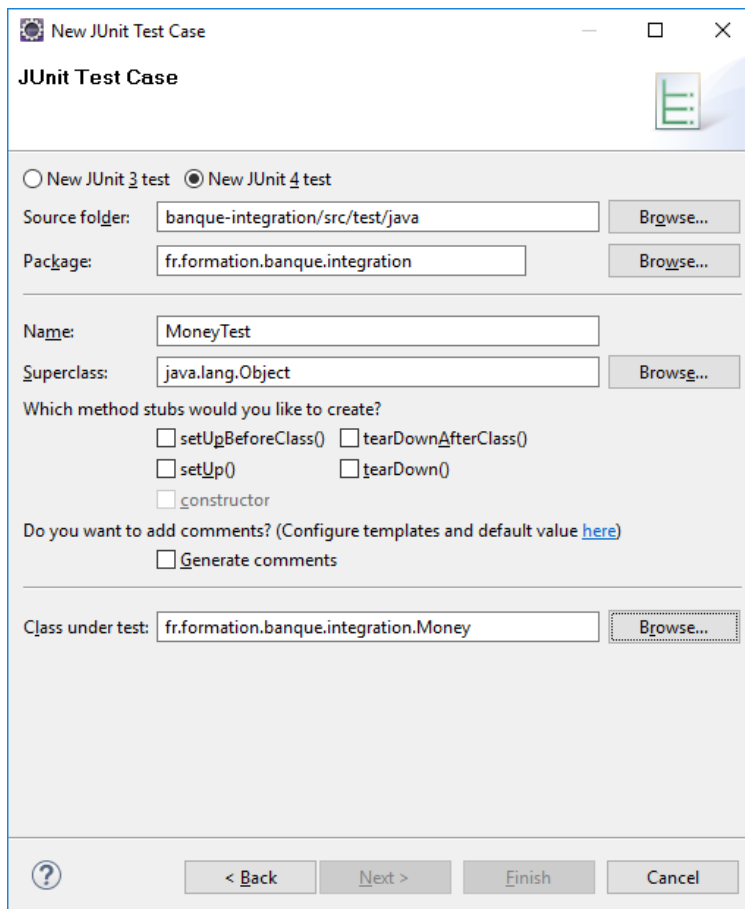


Création d'un test case

- Aller dans le menu File > New > JUnit Test Case
- Si la librairie JUnit n'est pas incluse dans la librairie du projet, Eclipse proposera de l'inclure automatiquement



Définition du test / Sélection de la classe à tester



New JUnit Test Case

JUnit Test Case

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

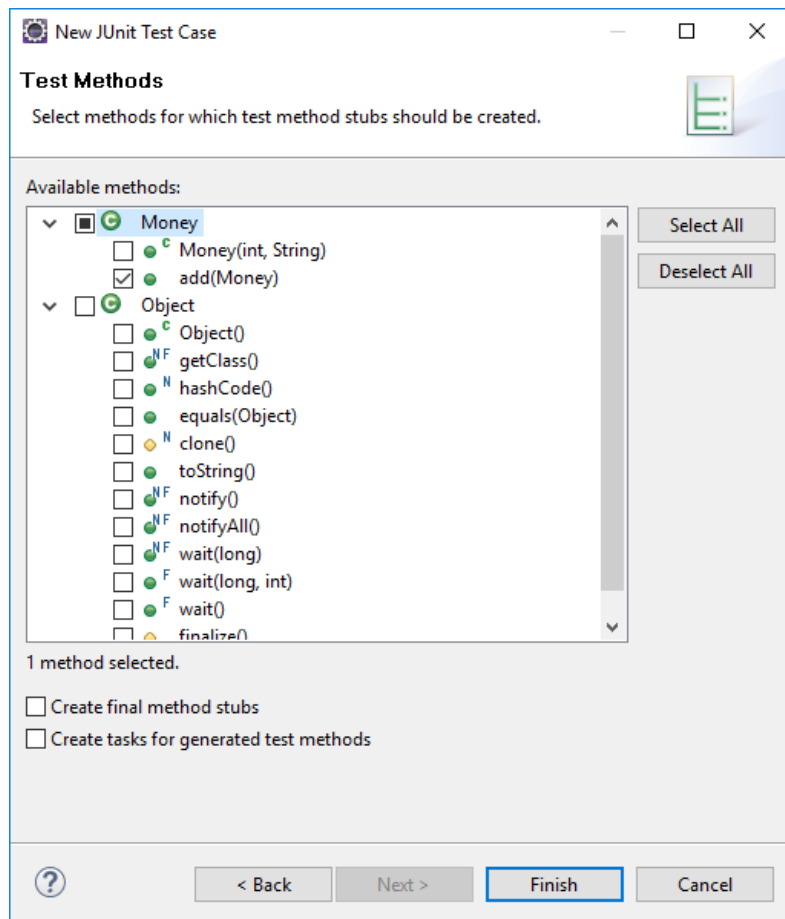
☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

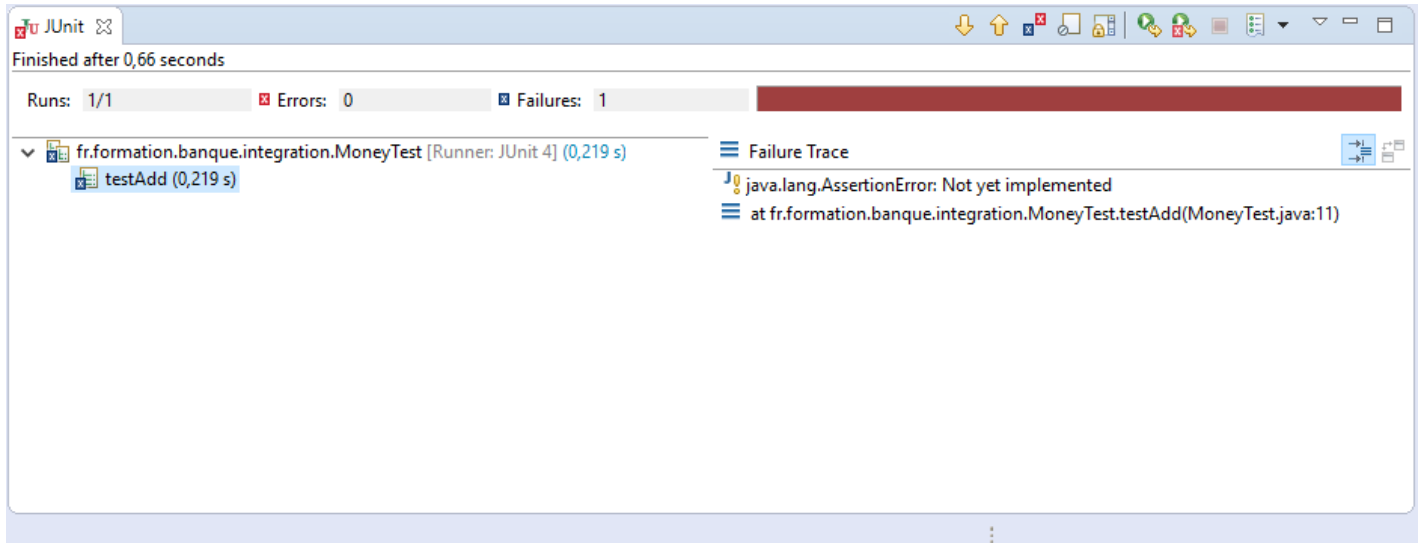
Class under test:

Sélection des méthodes à tester

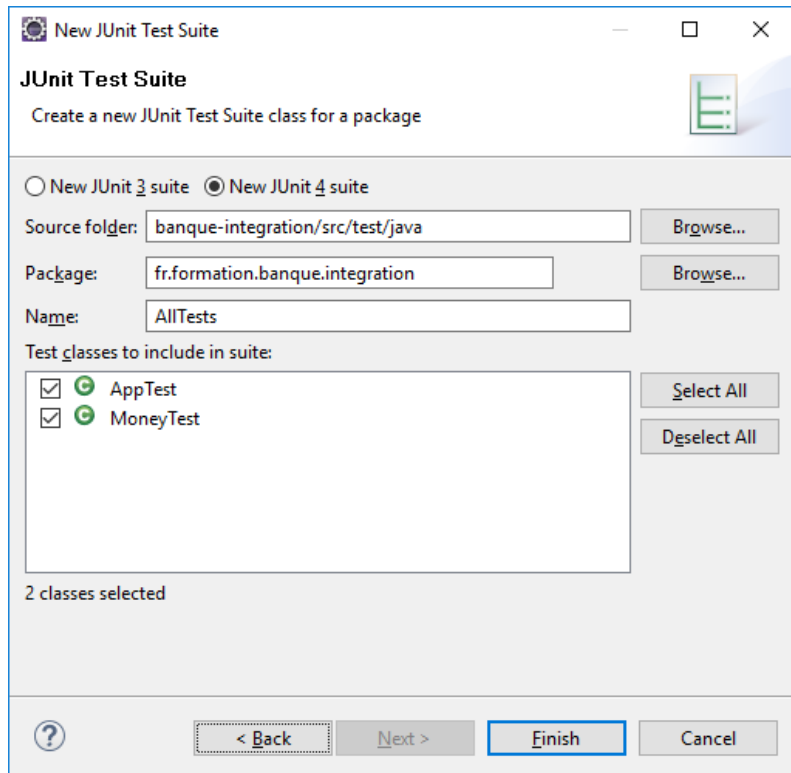
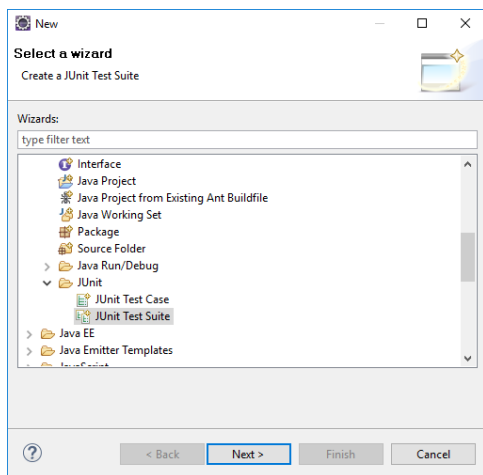


Exécution et résultat des tests

- Run as ... > JUnit Test



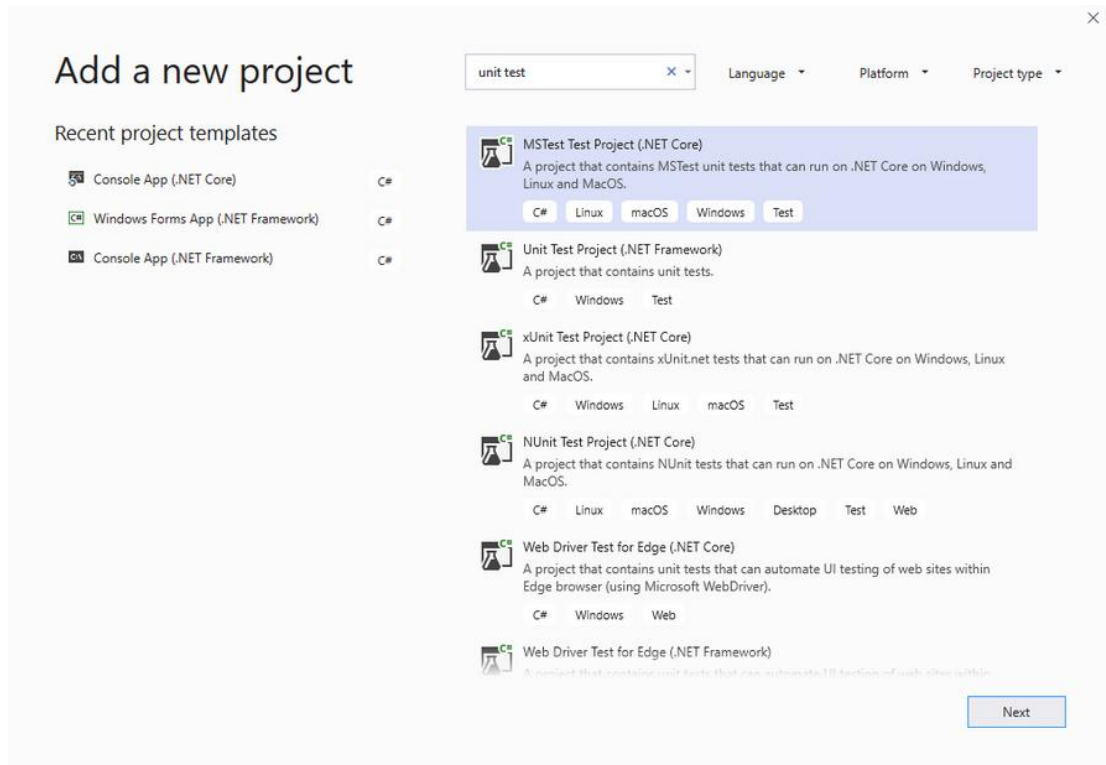
Suite de test avec Eclipse



Création de tests avec Visual Studio

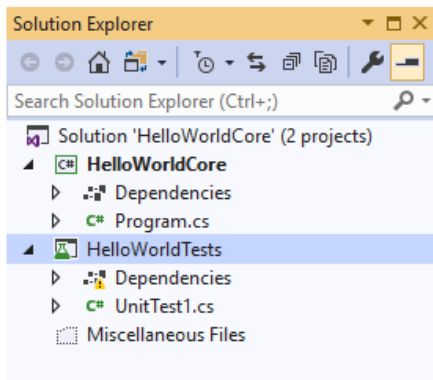
Création de tests

- Il faut créer un nouveau projet de type Test et l'ajouter à la solution
 - Plusieurs frameworks de tests sont disponibles



Les projets

- Le projet de test apparait dans la solution :



- Il faut ensuite implémenter les classes de test en fonction du framework de test choisi

Exécuter les tests dans Visual Studio

- Explorateur de tests
 - Tester -> Explorateur de tests
- Exécutez vos tests unitaires en cliquant sur « Tout exécuter »

