# Qualité et tests logiciels

ÉCOLE SUPÉRIEURE MULTINATIONALE DES TÉLÉCOMMUNICATIONS (ESMT)

LICENCE PROFESSIONNELLE EN TELECOMMUNICATIONS ET INFORMATIQUE (LPTI L3)

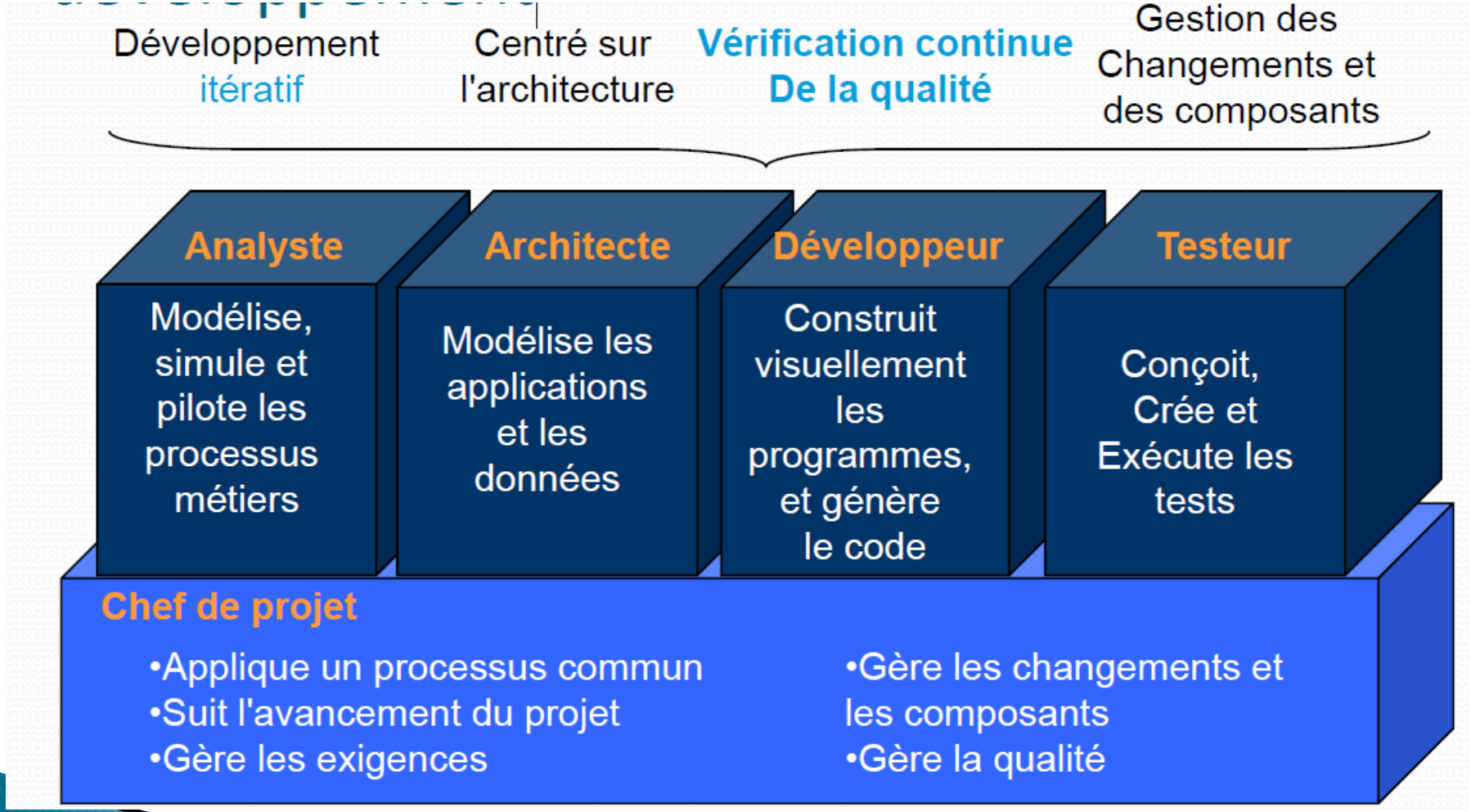
## Tests des logiciels

# Plan du Chapitre


- Contexte
- La vérification continue de la qualité
- Les dimensions du test
- JUnit

# Contexte

- Impératifs pour la réussite d'un développement



# Contexte

- Développement itératif
  - Adopter une démarche flexible;
  - Créer des versions exécutables itératives;
  - Valider les exigences et la qualité à chaque itération.
- Bénéfices métier :
  - Réduit les échecs, les coûts et minimise le gaspillage;
  - Unifie les équipes distribuées, les sous-traitants, les fournisseurs;
- Bénéfices technologiques :
  - Atténue les risques plus tôt dans le projet;
  - S'attaque aux causes des échecs;
  - Établit précisément le périmètre et mesure l'avancement du projet.

# Contexte


Exigences
Planning
Analyse et Conception
Modélisation métier
Gestion de Configuration et des changements
Implémentation
Planning Initial
Environnement
Test
Évaluation
Déploiement

# Vérification continue de la qualité


- Selon le Commissariat à l'Énergie Atomique (CEA), dans certains domaines – l'aéronautique par exemple –, le coût de vérification d'un logiciel peut atteindre 80% du coût total d'un système.

# Vérification continue de la qualité


- Traçabilité totale de la spécification aux plans de test en passant par le code
- Faire de la qualité une exigence centrale du processus;
- La qualité inclut fonctionnalité, fiabilité, montée en charge, maintenabilité, extensibilité, sécurité;
- Valider la qualité tôt et souvent;
- Gérer les changements pour garantir la réussite du déploiement;
- S'astreindre à la maintenance des tests;

# Nature des tests


- Qualité du code :
  - Découvrir les bugs avant qu'ils ne cassent l'application.
  - Assurer la qualité très tôt pendant le codage  en utilisant des outils d'analyse dynamique (débogueur, suivi exécution, etc.)
- Fonctionnalités :
  - réaliser des tests unitaires,
  - des tests de non-régression,
  - des tests d'intégration,
- Performances :
  - Test de charge
  - mise à l'épreuve d'une application en émulant des utilisateurs avec un outil de génération de charge
  - détecter les goulots d'étranglement sur les ressources systèmes à observer (temps de réponses, occupation mémoire, etc.)
- Réception :
  - Réalisé par le client (user acceptance tests, clean-room tests)
  - S'assurer que le cahier des charges a été respecté.

# Les tests fonctionnels


- Test unitaire (unit test) :
  - vérifie le bon fonctionnement de chaque composant d'un programme.
  - veille à ce que tous les chemins logiques du programme soient parcourus,
  - à ce que la plage de validité des données et sorties ait été explorée
  - et à ce que les résultats soient conformes au plan de conception.
- Test d'intégration (integration test) :
  - Test qui vérifie des modules (programmes, matériel) qui ont été intégrés précédemment.
  - se fait par assemblage et par essais progressifs de programmes ou de matériel jusqu'au bon fonctionnement du système complet.
- Test de non-régression (non-regression test) :
  - A effectuer dans le cas de changement de version, permet de vérifier que les modifications apportées n'ont pas fait régresser l'application
    - perte de fonctionnalités déjà existantes
    - réintroduction de bugs déjà résolus
    - dégradation du comportement du logiciel antérieurement validé.
  - portent sur l'exécution de tests déjà exécutés, afin de s'assurer que le système répond toujours aux exigences spécifiées.

# Les tests unitaires


Test White-box (test structurel)Twb:

- L'implémentation de la classe doit être connue.
- Test vérifiant si le code est robuste en contrôlant son comportement avec des cas inattendus (cas limites).
- Le but de ce test est d'exécuter chaque branche du code avec différentes conditions d'entrée afin de détecter tous les comportements anormaux.
  - Ex : valeur min, max, dépassement de capacité, etc.

Test Black-box (test "boîte noire")Tbb:

- Il s'effectue sans connaissance de la structure interne.
- Il permet de vérifier que les exigences fonctionnelles et techniques du cahier des charges sont respectées.

# Quelle catégorie de tests choisir ?


Les Twb sont très difficiles à mettre en œuvre tandis que les Tbb peuvent être automatisés.

De quels outils dispose t-on ?

- Tests boîtes blanches :
  - assertions embarquées dans le code;
  - Pré/post conditions d'exécution d'une instruction;
- Tests boîtes noires :
  - Conception par contrats (design by contracts) dans le langage Eiffel;
  - Pré/post conditions d'appel à des objets;
  - Assertions Java;
  - Outil d'exécution de tests unitaires (JUnit, etc.);

# Quel volume de tests ?


- Première réponse: "tout ce qui peut casser ("Extreme Programming") la taille des tests devient prohibitive !
- Deuxième réponse: au moins un test unitaire par méthode, voire plusieurs tests par méthode.
- Les tests les plus riches sont ceux qui testent les interfaces à des frontières significatives entre les niveaux.

# Qui doit écrire les tests ?


- Les tests doivent être écrits par le développeur qui a écrit le code.
- Cela fait partie de son métier
- Il a une meilleure idée des objectifs du code.
- Il a une meilleure compréhension du fonctionnement interne du code, des valeurs des paramètres d'entrée, des chemins d'accès conditionnels, ou des circonstances normales ou exceptionnelles.

# Quand exécuter les tests ?


Plusieurs granularités:

- l'application toute entière
- un niveau de l'architecture,
- une classe, ou même une méthode d'une classe.

Réponse : selon le niveau de responsabilité

- À chaque ajout de méthodes ou de classes à un niveau;
- À chaque ajout de fonctionnalités à un sous-système;
- Avant de basculer le code dans le référentiel de code source;
- Après la mise à jour d'une révision;
- A chaque installation d'un nouvel environnement,

l'application doit être testée en entier dans cet environnement (assurance qualité, démo, production).

# Le framework JUnit


JUnit offre :

- Des outils pour créer un test (assertions)
- Des outils pour gérer des suites de tests
- Des facilités pour l'exécution des tests
- Des statistiques sur l'exécution des tests
- Interface graphique pour la couverture des tests
- Points d'extensions pour des situations spécifiques

# Le framework JUnit


- JUnit Un framework:
- un ensemble d'interfaces et de classes qui collaborent qui ne s'utilisent pas directement
- c'est une application qui
  - offre la partie commune des traitements
  - et est spécialisée par chaque utilisateur
- il faut implanter/spécialiser les types fournis

ex : pour créer un cas de test on hérite de la classe *TestCase*

# Classes et méthodes de test


*Classe de test*

- *hérite de JUnit.framework.TestCase*
- *contient les méthodes de test*
- *peut utiliser les assertXXX sans import*
- *constructeur a paramètre String selon la version*

*Méthodes de test*

- *publique, type de retour void*
- *le nom commence obligatoirement par test*
- *pas de paramètre, peut lever une exception*
- *contiennent des assertions*

# Classes et méthodes de test


*Fixtures : Méthodes utilisées pour initialiser les données communes*

- ***setUp*** *: méthode appelée avant chaque méthode de test*

*permet de factoriser certains traitements.*

*ex* ***protected*** *void setUp() throws Exception {*

*super.setUp(); ...}*

- ***tearDown*** *: appelée après chaque méthode de test, permet de libérer les données.*

*Ex:* ***protected*** *void tearDown() throws Exception {*

- *super.tearDown(); ...}*

# Exemple JUnit


```
public class CalculatriceTest
{
@Before
public void setUp() throws Exception
{
}
@After
public void tearDown() throws Exception
{
}
@Test
public void additionnerNombres() {
final long resultat =
Calculatrice.additionner(10, 20);
Assert.assertEquals(30, resultat);
}
```

# Suites de test


Classe TestSuite

- ensemble de cas de test
- peut être nommée
- constructeur TestSuite (Class) : construit une suite de test contenant tous les cas de test

correspondant a une méthode commençant par test dans la classe passée en paramètre

- utilisation de la réflexion Java
- méthode addTest(Test) : pour ajouter un cas de test donné.

# JUnit 4+-@Test


- **@Test** marque les cas de test; on n'a pas besoin de préfixer les cas de test avec "test". La classe n'a pas besoin d'hériter de la classe "TestCase".

```
@Test
public void addition() {
assertEquals(12, simpleMath.add(7, 5));
}
@Test
public void subtraction() {
assertEquals(9, simpleMath.sub(12, 3));
}
```

# JUnit 4+-@Before and @After


- Annotations **@Before and @After** pour les méthodes "setup" et "tearDown". Elles s'exécutent avant et après chaque test case.
- @Before
- public void exécuterAvantChaqueTest() {
- simpleMath = new SimpleMath();
- }
- @After
- public void ru exécuterAprèsChaqueTest() {
- simpleMath = null;
- }

# JUnit 4+- @BeforeClass and @AfterClass


- @BeforeClass et @AfterClass exécutés une fois avant et après tous les cas de test d'une classe.
- @BeforeClass
- public static void exécuterAvantClasse() {
- // exécuter une fois avant tous les cas de test
- }
- @AfterClass
- public static void  exécuterAprèsClasse() {
- // exécuter une fois après tous les cas de test
- }

# JUnit 4+ Gestion des Exceptions


- *Exception Handling*

*Utiliser le paramètre «expected » avec @Test pour les cas de test qui s'attendent à une exception. Écrivez le nom de classe de l'exception qui sera levé.*

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
// divide by zero
simpleMath.divide(1, 0);
}
```

# JUnit 4+-@Ignore


- *@Ignore annotation pour les cas de test à ignorer. On peut ajouter un paramètre de chaîne qui définit la raison de l'ignorance.*

```
@Ignore("Non encore pret")
@Test
public void multiplication() {
assertEquals(15, simpleMath.multiplier(3, 5));
}
```

# JUnit 4+ - Timeout


- *Timeout définit un délai en milliseconde. Le test échoue lorsque le délai est dépassé.*

```
@Test(timeout = 1000)
public void boucleInfinie() {
while (true)
;
}
```

# JUnit 4+ Tableaux


- *Comparaison des tableaux. 2 tableaux sont égaux s'ils ont la même longueur et chaque élément est égal à l'élément correspondant dans l'autre tableau; sinon, ils ne le sont pas.*

```
public static void assertEquals(Object[] attendu, Object[] obtenu);
public static void assertEquals(String message, Object[] attendu, Object[] obtenu);
@Test
public void listeEgalite() {
List<Integer> attendu = new ArrayList<Integer>();
attendu.add(5);
List<Integer> obtenu = new ArrayList<Integer>();
obtenu.add(5);
assertEquals(attendu, obtenu);}
```

# Assertions


Une assertion est simplement une comparaison entre la valeur espérée et la valeur réelle obtenue.

- Détermine le succès ou l'échec d'un test
- Les assertions JUnit sont des méthodes dont le nom commence par "***assert***"
- Deux (2) catégories de méthodes d'assertion :
  - AssertXXX(…)
  - AssertXXX (String Message, …) où message est le message affiché quand il y a un échec

# Assertions : Quelques méthodes


Affirmer qu'une condition est vraie.

- assertTrue(boolean condition)
- assertTrue(String Message, boolean condition)
- Affirmer qu'une condition est fausse.
- assertFalse(boolean condition)
- assertFalse(String Message, boolean condition)

# Assertions : Quelques méthodes


Affirmer qu'une valeur est égale à une valeur (comparaison de contenus – types primitifs).

- assertEquals(ValeurAttendue, ValeurObtenue)
- assertEquals(String Message, ValeurAttendue, ValeurObtenue)

Affirmer que deux (2) références ont la même adresse.

- assertSame(Object ValeurAttendue, Object valeurRéelle)
- assertSame(String Message, Object ValeurAttendue, Object valeurRéelle)

# Assertions : Quelques méthodes


Affirmer qu'une référence est nulle.

- assertNull(Object objet)
- assertNull(String Message, Object objet)
- Affirmer qu'une référence n'est pas nulle.
- assertNotNull(Object objet)
- assertNotNull(String Message, Object objet)

# Ce que vous avez a faire


Créer une classe de test (par classe testée ou par méthode testée)

- Créer des méthodes de test dans cette classe
- Créer une suite de test associée a ces cas de test
- Lancer cette suite de test et observer les résultats :
  - success : pas d'erreur ni d'exception levées ;
  - failure (échec) : lancement d'une AssertionFailedError ;
  - error (erreur) : autre exception ou erreur levée.

# Comment ça marche ?


L'interface Test possède une méthode run() qui prend en paramètre un TestResult servant à stocker le résultat.

- Le TestRunner.run() appelle cette méthode pour la suite de test.
- Pour chaque cas de test :
  - exécution du setUp()
  - exécution de la méthode testXXX
  - exécution de la méthode tearDown
- Puis affichage formaté du TestResult

# Bonnes pratiques


- Ne jamais présumer de l'ordre dans lequel les tests dans une classe TestCase sont déroulés;
- Éviter d'écrire des TestCases avec des effets de bord;
- Nommer les tests de manière pertinente;
- S'assurer que les tests ne dépendent pas de données volatiles;

# Bonnes pratiques


- Prendre en compte les paramètres régionaux;
- Utiliser les méthodes assert/fail et le mécanisme d'exception de Java;
- Documenter les tests en Javadoc;
- Éviter la validation humaine;
- Écrire des tests petits et rapides;