ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

07 Chapitre

Test logiciel

1I2AC1 : Génie logiciel et Conception orientée objet

Régis Clouard, ENSICAEN - GREYC

« Durant le débogage, les novices insèrent du code correctif,
alors que les experts suppriment du code défectueux. »
Richard Pattis (Professeur université de Californie)

Objectif du chapitre

- Initiation aux tests logiciels et en particulier aux tests dynamiques.
- À l'issue de ce chapitre :
  - Vous serez sensibilisé à l'importance des tests logiciels dynamiques.
  - Vous serez capable de construire du code testable.
  - Vous saurez programmer des tests unitaires et utiliser des « bouchons ».

Plan du chapitre

1
Généralités
sur les tests

Pourquoi tester ?

- Seul moyen pour chasser les bugs.
  - Pas de preuve formelle pour des programmes quelconques, donc pas de programme automatique pouvant calculer une preuve de code.
- Éviter les mauvaises surprises.
  - Découverte de bugs après la mise en production.
- Prémunir contre la régression de code.
  - Les tests de non-régression assurent que les modifications du code lors de l'extension ou de la correction de bug n'affectent pas le bon fonctionnement des parties existantes.
- Rassurer le développeur.
  - Lorsque le code a atteint une certaine complexité, on commence à craindre les modifications.
- Rassurer le client.
  - Produit final de meilleure qualité.

Bug

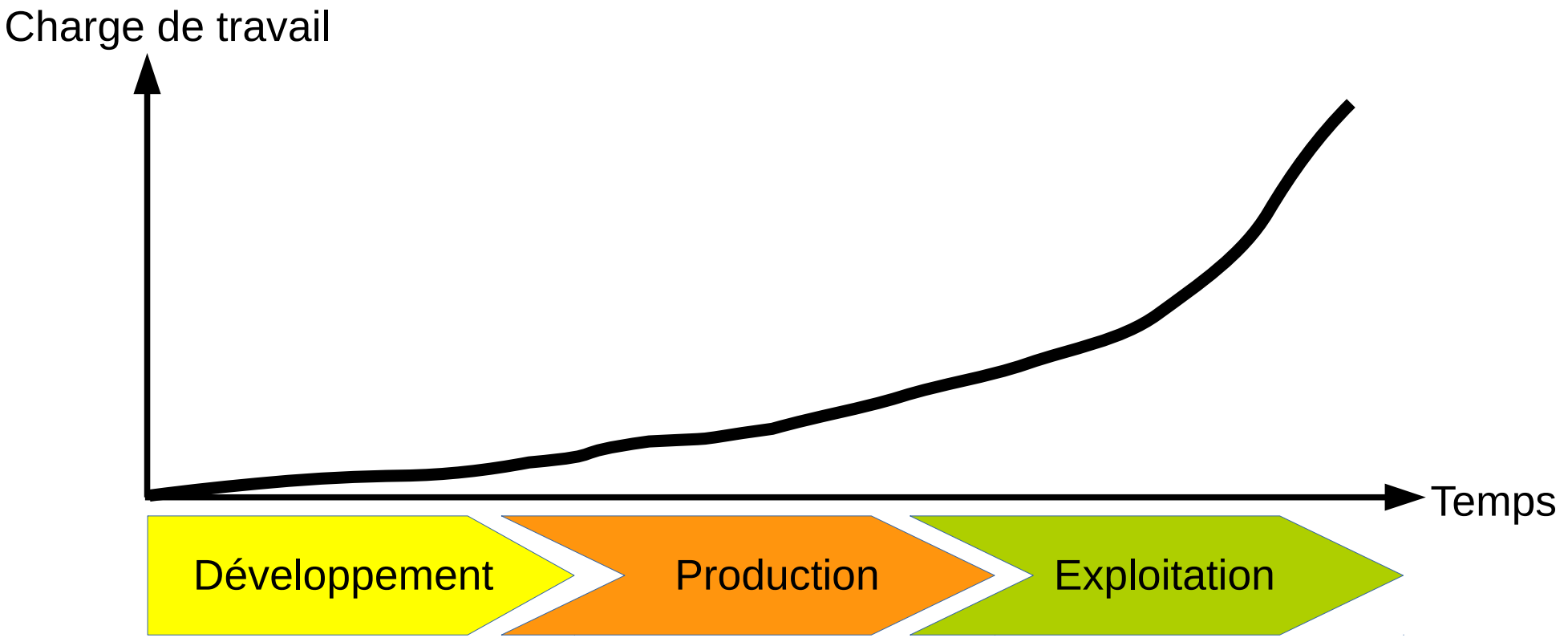
- Mot popularisé en 1947 par **Grace Hopper** pionnière de l'informatique (mot issu du matériel).
- Catégories de bug :
  - **Bohrbug** (inspiré de l'atome de Niels Bohr) : un bug qui a toutes les bonnes propriétés, en particulier il est répétable dans les mêmes conditions. (*bug classique*)
  - **Heisenbug** (inspiré du principe d'incertitude de Heisenberg) : un bug dont le comportement est modifié quand on essaye de l'isoler. (c*as typiques des exécutions sous dévermineurs*.)
  - **Mandelbug** (inspiré des fractales de Mandelbrot) : un bug dont les étapes pour le reproduire sont tellement complexes qu'il semble se reproduire de façon aléatoire et chaotique.
  - **Schroedinbug** (inspiré du chat de Schrödinger) : un bug qui ne se manifeste pas à l'exécution, mais qui est découvert après que quelqu'un ait relu le code source ou utilisé le logiciel d'une façon non habituelle.

La chasse aux bugs

- Les bugs sont inhérents à l'activité de codage.
  - Le code zéro bug n'existe pas (sauf exceptions).
  - Rappel de l'estimation : 1 à 10 bugs / KLOC
- Mais il est possible de détecter certains bugs en testant les programmes pour limiter la casse.
  - **Dans l’approche agile, il est du devoir du développeur de tester son code.**

Quand tester ?

- Le plus tôt possible
  - Coût d'un bug dans le cycle de vie d'un logiciel



Mauvaise pratique : le pistage

- Pratique classique (parfois enseignée)
  1. Truffer le code d'appels d'affichage (*printf*) pour localiser la partie du code fautive.
  2. Lancer le programme pour espérer isoler les lignes de bugs.
  3. Corriger le bug en modifiant et relançant le programme autant de fois que nécessaire.
  4. Effacer les appels d'affichage.
- Mais cette pratique est inefficace :
  - Plus le code est développé, plus le temps de recherche de bug est important.
  - Ne prémunie pas contre une régression future (n'empêche pas le bug de se reproduire).
  - L'introduction de lignes de code peut changer le programme (cf. Heisenbug).

Bonne pratique : le test dynamique

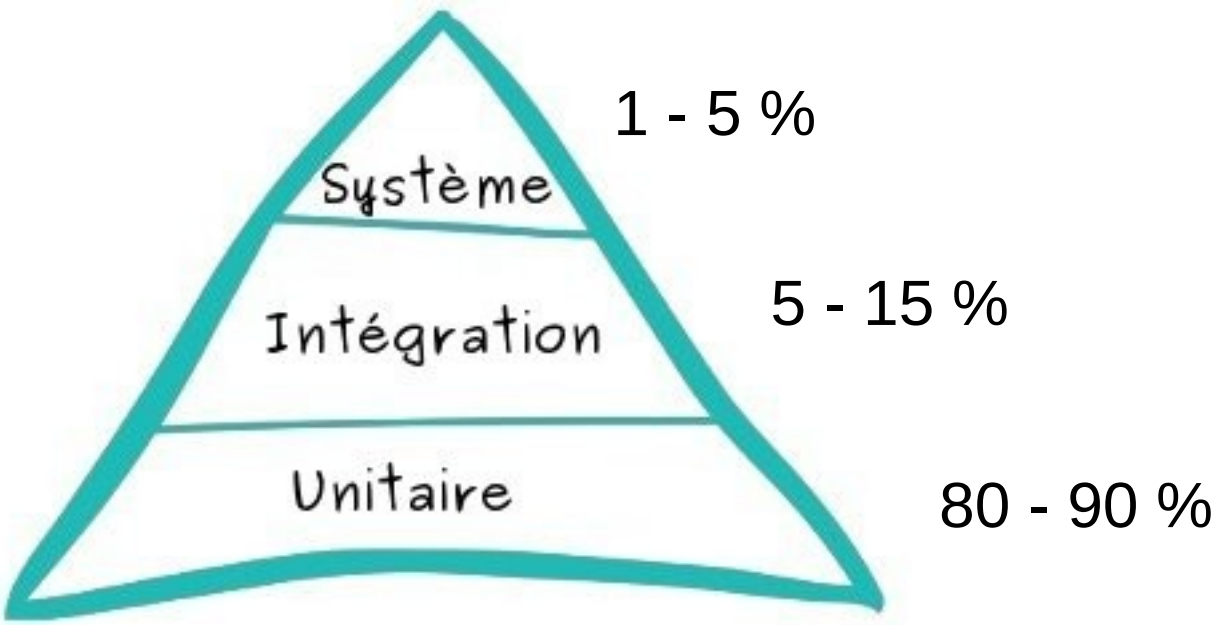
- Le test s'entend comme **test dynamique** (ie. exécutable) :
  - « *Tester est l'activité qui consiste à exécuter un programme avec l'intention de trouver des bugs* » - Glenford Myers.
- Remarque : le test inclut aussi des tests manuels qui ne sont pas présentés ici.
  - Voir les « quadrants de test agile ».

Qu'est ce qu'un test ?

- Procédé de Vérification & Validation (V&V)
  - Vérification : **le logiciel fonctionne t-il correctement ?**
    - Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que des exigences spécifiées ont été remplies.
  - Validation : **a-t-on construit le bon logiciel ?**
    - Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que les exigences, pour un usage ou une application voulue, ont été remplies.

Pyramide des tests

Validation



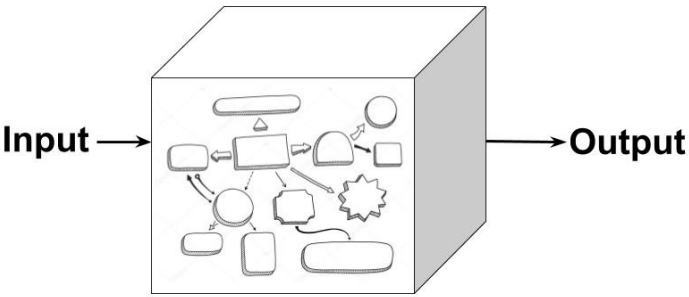
Vérification

Niveaux de test

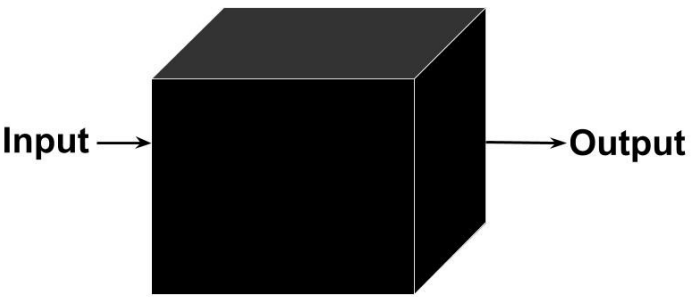
- Test système : critiquer le produit.
  - Test du système dans son ensemble, IHM.
  - Erreurs recherchées : absence de fonctionnalités et fonctionnalités mal mises en œuvre.
  - En production, plusieurs niveaux de tests du système.
    - Version alpha : système en utilisation interne.
    - Version bêta : utilisation chez les utilisateurs externes avertis.
- Test d'intégration : guider l'équipe de développement.
  - Test de l'assemblage des unités.
  - Erreurs recherchées : erreurs d'interface entre unités.
- Test unitaire : guider le développeur.
  - Test en isolation des unités de développement.
  - Erreurs recherchées : erreurs de codage et erreurs fonctionnelles.

Types de test

- Test **boîte blanche** (*white-box testing)*
  - Tient compte de la structure interne de l'unité testée.
  - Peut utiliser les parties protégées et privées.



- Test **boîte noire** (*black-box testing*)
  - Ne connaît pas la structure interne de l'unité testée.
  - Ne peut utiliser que les parties publiques.



Exemple fil rouge

- On suppose une classe `Human` possédant les méthodes publiques suivantes :
  - `void setAge( int age )`
  - `void setAgeLimit( int age )`
  - `boolean isAdult( )` qui :
    - lève une exception si `age` $\notin$ [0, age_limit]
    - retourne `true` si `age` $\in$ [18, age_limit]
    - retourne `false` si `age` $\in$ [0, 18]

Étape 1 : Objectif de test

- On choisit une caractéristique ou une fonction à tester :
  - C'est l'objectif de test.
- Par exemple
  - On décide de tester la méthode `isAdult()` dans le cas nominal où le paramètre `age` est dans `[18, age_limit]`

Étape 2 : Jeu de test

- Ensuite, on choisit le jeu de test.
- Pour notre exemple, il faut choisir :
  - une valeur pour le seuil d'âge limite.
  - une valeur pour le paramètre `age` qui soit dans l'intervalle : `[18, age_limit]`.
- Par exemple :
  - (`age_limit = 150, age = 35`)

Étape 3 : Test exécutable

- On code le test.
- On exécute le système avec le jeu de test.
- Dans notre exemple :
  - si `h` est un objet de type `Human` tel que :

    ```
    h.age_limit = 150
    ```
  - alors on effectue le test :

    ```
    h.setAge(35);
    h.isAdult();
    ```

Étape 4 : Oracle

- On compare le résultat obtenu au résultat attendu :
  - C'est l'oracle.
- Représenté par une assertion : une expression booléenne censée être vraie.
- Dans notre exemple :
  - `h.isAdult()` retourne `true`

Étape 5 : Verdict

- On en déduit si le test a réussi ou échoué :
  - C'est le verdict.
- Verdicts classiques :
  - si on récupère true : le test passe
  - si on récupère false : le test échoue
  - s'il y a levée d'exception, c'est une erreur, le test est inconclusif.
- Exemple d'implémentation naïve en Java (voir JUnit)

```
try {
    h.setAge(35);
    if (h.isAdult()) {
        System.out.println("test passe");
    } else {
        System.out.println("test echoue");
   }
} catch (Exception e) {
    System.out.println("erreur, test inconclusif");
}
```

Fixture

- Avant d'exécuter l'oracle, il faut amener l'objet dans un état donné : **fixture** (*installation*).
- Dans notre exemple
  - Création de l'objet et positionnement de son état.

```
Human h = new Human();
h.setAgeLimit(150);
try { ... // oracle ...}
```

Cas de test

- L'ensemble du test exécutable s'appelle un cas de test :
  - Définition IEEE 610 : Un cas de test est un ensemble de **données de test**, de **pré-conditions** d'exécution, de **résultats attendus** développés pour un objectif, tel qu'exécuter un chemin particulier d'un programme ou vérifier le respect d'une exigence spécifique.

```
Human h = new Human();                          // fixture
h.setAgeLimit(150);
try {
    h.setAge(35);                               // donnée de test
    if (h.isAdult()) {                          // oracle
        System.out.println("test passe");       // verdict
    } else {
        System.out.println("test echoue");
   }
} catch (Exception e) {
    System.out.println("erreur, test inconclusif");
}
```

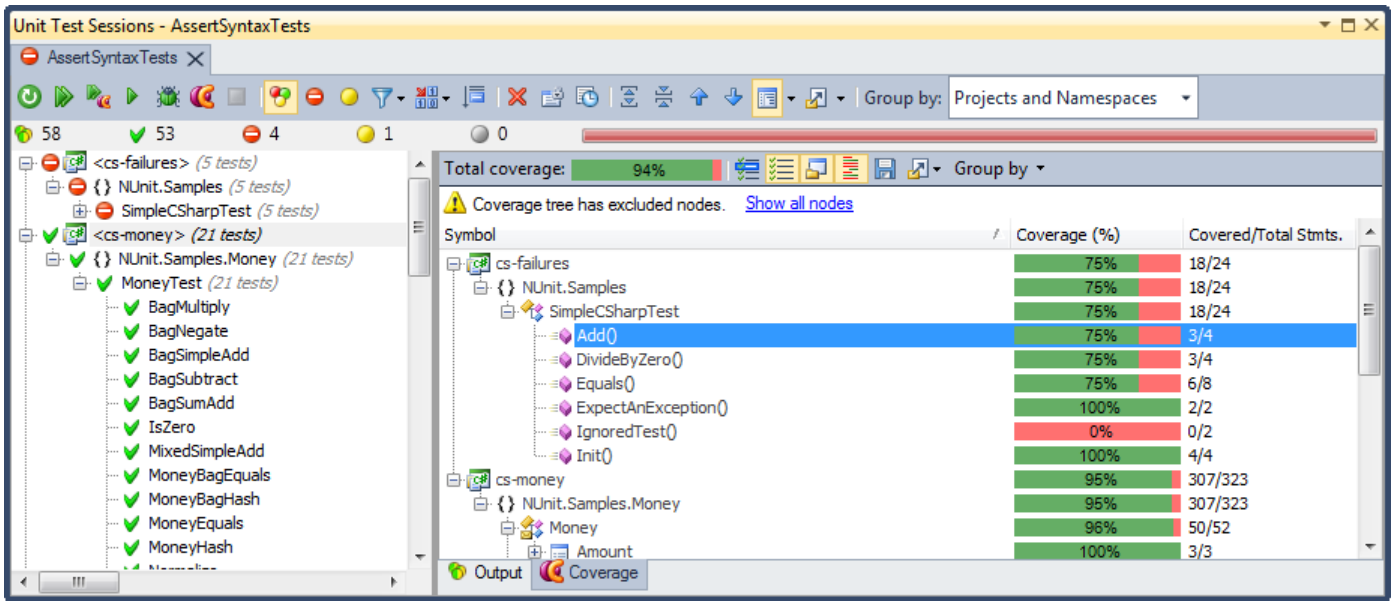
Quand s'arrêter de tester ?

- Les tests sont incomplets par nature.
  - On a testé `isAdult()` pour une valeur d'entrée seulement, on n'aurait pas pu les tester toutes !
  - Savoir quand s'arrêter est une question d'expérience.
- Un indice, la **couverture** des tests
  - Mesure pour décrire le taux de code source exécuté d'un programme quand une suite de tests est lancée.
- Mais, le test ne certifie pas le code :

  « Le test de programme peut être utilisé pour prouver la présence de bugs, mais jamais leur absence » - Edsger Dijkstra.
- Parfois le code est faux, mais les tests aussi !
  - Question : Doit-on faire des tests de tests ?

Couverture de code

- La couverture de code est une mesure qui permet d'identifier la proportion du code testé.
  - Pragmatiquement, elle correspond au taux de code source exécuté d'un programme quand une suite de test est lancée.
- Outils d'analyse de couverture
  - Les IDE (IntelliJ) proposent des outils de mesure de la couverture de code.



Plan du chapitre

1
Généralités
sur les tests

2
Tests
unitaires
(avec JUnit)

Test unitaire

- Procédé de vérification d'une unité.
  - Vérifier le bon fonctionnement d'une unité précise d'un logiciel.
  - Rien à voir avec la chasse au bug. Il s'agit essentiellement de se prémunir contre la régression de code dans l'unité.
- En programmation orientée objet, l'**unité** est la **classe**.
  - À chaque classe, on associe une autre classe qui la teste.
  - Les tests unitaires testent une classe et une seule et sont indépendants les uns des autres (test en isolation).
    - Assurer que si un test échoue, c'est la classe testée qui est fautive.
- Par exemple:
  - On teste les méthodes publiques et protégées de la classe `TelecommandeTV` en isolation.
  - Si on teste la classe `TelecommandeTV` en utilisant une télévision dans les tests, c'est du **test d'intégration**.

FIRST : Qualité d'un test unitaire

- **[F]**ast (Rapide)
  - Plusieurs centaines de tests par seconde.
- **[I]**solated (Isolé)
  - Les causes d'échec des tests sont évidentes.
- **[R]**epeatable (Répétable)
  - Exécutions répétitives dans n'importe quel ordre et à n'importe quel moment.
- **[S]**elf-validating (Auto-évaluable)
  - Pas de recours à un utilisateur pour l'oracle.
- **[T]**imely (Juste à temps)
  - Programmé dès que l'on a la connaissance sur la fonctionnalité.

Testabilité d'un code

- On n'écrit pas du code testable comme du code classique.
- Par exemple, une alarme qui se déclenche à une heure butoir donnée dans un agenda.
  - On part du très mauvais code suivant :

```
public final class Agenda {
    ...
    public void check() {
        if (System.getTimeInMillis() > 100) {
            new Bell().ring();
        }
    }
}
```

  - Voyez-vous pourquoi il est mauvais ?

Contrôler une entrée indirecte

- La méthode `check()` a deux données de test :
  - la date à laquelle l'alarme se déclenche : p. ex 18h.
  - la date courante : p. ex 18h05.
- Or dans `check()` la date courante est nécessairement fournie par `System`.
  - `System` étant incontrôlable, donc **ce code n'est pas testable**.
- La date courante est une entrée indirecte de `check()` qu'il faut pouvoir contrôler.

Observer une sortie indirecte

- L'objectif du test est :
  - vérifier que l'alarme se déclenche si la date courante est postérieure à la date butoir.
- Quel est l'oracle ?
  - écouter si on entend ou pas quelque chose.
    - *mais ce n'est pas auto-évaluable*.
  - observer si la méthode `ring()` de `Bell` a été appelée.
  - Or l'objet Bell est créé par `check()` :
    - il n'est pas observable, donc ce code n'est **pas testable**.
- L'interaction avec `Bell` est une sortie indirecte de `check()`, qu'il faut pouvoir observer.

Code testable

- Solution : Encapsulation et externalisation des dépendances pour rendre un code testable.

```
public final class Agenda {
  public Agenda( MyTime limit, Clock clock, Bell bell ) {
     _clock = clock; _bell = bell; _limit = limit;
  }
  public Agenda( ) {
     _clock = System; _bell = new Bell(); _limit = 100;
  }
  public void check() {
     if (isTimeToRing()) {
        _bell.ring();
     }
  }
  private boolean isTimeToRing() {
     MyTime now = _clock.getTimeInMillis();
     return _limit.isBefore(now);
  }
}
```

Limites du test unitaire

- Classe abstraite.
  - Solution : créer dans la classe de test une classe héritant de la classe abstraite pour ne tester que les méthodes de la classe abstraite.
- Méthode privée.
  - Solution : même principe que précédemment
- Classe avec dépendances à d'autres classes.
  - Solution : utilisation des doublures (voir section 4).

Le framework JUnit

- Intégré à IntelliJ.
- Ce qu'il offre :
  - Des assertions expressives pour automatiser le verdict.
  - La visualisation du verdict.
  - La possibilité de lancer facilement les tests.

Organisation des codes

- Les tests ne doivent pas être dans le source de l'applicatif.
  - Une organisation possible :
    - **bin** pour les exécutables.
    - **src** contenant les paquets pour les sources applicatifs.
    - **test** avec les mêmes paquets mais contenant les sources des tests.
  - Toutefois, l'organisation en paquet des tests suit celle des sources.
- Avantages :
  - Pas de pollution des sources par les tests, pas de pollution des tests par les sources.
  - Permet de livrer l'applicatif avec ou sans les tests.

Les assertions de base de JUnit

- Servent à décrire l'oracle du cas de test.
- Variations autour du assert du C :
  - `assertTrue(boolean condition)`
  - `assertFalse(boolean condition)`
  - `assertEquals(Object expected, Object actual)`
    - Utilise `equals()` des objets.
  - `assertEquals(double expected, double actual, double delta)`
  - `assert[Not]Same(Object expected, Object actual)`
    - Même objet en mémoire
  - `assert[Not]Null(Object actual)`
  - `assertArrayEquals([] expecteds, [] actuals)`
  - `fail()`

Méthode de test JUnit : @Test

- Méthode d'une classe de test, représentant un cas de test.
  - annotée par @Test.
  - publique, sans paramètre et de type de retour `void`.
- Exemple reprenant le cas d'étude `Human` :

```
@Test
public void testIsAnAdultWhenAgeGreaterThan18() {
    Human h = new Human();
    h.setAgeLimit(150);
    h.setAge(35);
    assertTrue(h.isAdult());
}
```

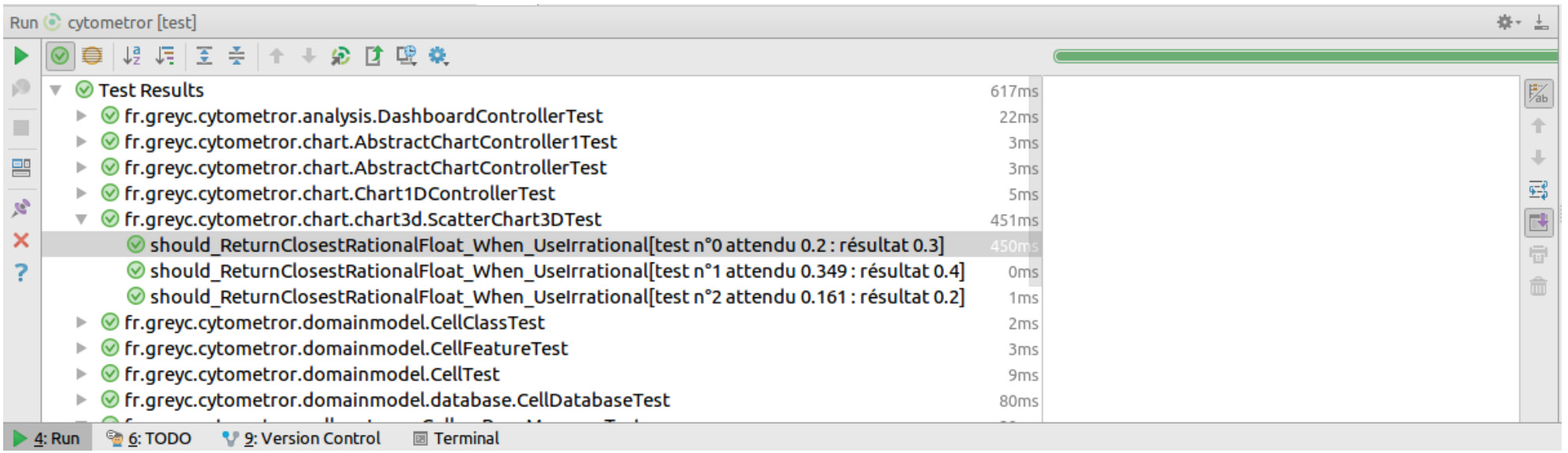
Gestion des exceptions

- Exemple
  - Vérifier que l'appel de la méthode lève une exception.

```
@Test(expected = OutOfLevelException.class)
public void testThrowExceptionIfAgeIsGreaterThanTheMaxiumum()
        throws OutOfLevelException {
    Human h = new Human();
    h.setAgeLimit(150);
    h.setAge(151);
}
```

Le verdict de JUnit

- Verdict :
  - le test passe : barre verte.
  - le test échoue : barre rouge.
  - levée d'une exception attendue : barre verte.
  - levée d'une exception inattendue : barre rouge.



Fixture JUnit : @Before

- Préambule : méthode annotée par @Before :
  - appelée avant chaque méthode de test.
  - sert à factoriser la construction des objets.

```
public final class TestLevelManagement {
  private Human _human;

  @Before
  public void setUp() throws Exception {
    _human = new human();
    _human.setMaxiumAge(150);
  }

  @Test(expected = OutOfLevelException.class)
  public void testAgeGreaterThanMax()throws OutOfLevelException {
        h.setAge(151);
        h.isAdult();
  }
}
```

Fixture : les autres méthodes

- Postambule : méthode annotée par @After.
  - Appelée après chaque méthode de test.
- Préambule et postambule de classe : méthodes annotées @BeforeClass et @AfterClass.
  - publiques et statiques.
  - exécutées avant (resp. après) l'appel de la première (resp. dernière) méthode de test.
  - une seule méthode pour chaque annotation.

Plan du chapitre

1
Généralités
sur les tests

2
Tests
unitaires
(avec JUnit)

3
Test en isolation
et doublures
(avec Mockito)

Pourquoi des doublures ?

- Offrir une solution pour développer des tests lorsqu'ils nécessitent :
  - Un composant dont le code n'est pas encore disponible.
    - p. ex : persistance des données.
  - Un composant difficile à mettre en place.
    - p. ex : une base de données.
  - Un comportement exceptionnel d'une classe.
    - p. ex : déconnexion dans un réseau.
  - Un composant dont le code est lent.
    - p. ex : construction d'un maillage.
  - Une fonction qui a un comportement non-déterministe.
    - p. ex : réseau.
- On parle aussi de bouchon quand une classe est remplacée par une doublure.
- Les doublures peuvent aussi être employées dans le code source.

La doublure pour les tests

- Classe créée dans le paquet `test` pour remplacer une classe du logiciel dans le paquet `src`.
- Rôles :
  - Le substitut (*fake*)
    - classe qui est une implémentation partielle et qui retourne toujours les mêmes réponses selon les paramètres fournis.
  - L'espion (*spy*)
    - classe qui vérifie l'utilisation qui en est faite après l'exécution.

Exemple du framework Mockito

- On suppose la classe `MonInterface` à doubler :
  - D'abord dans le fichier test on importe le paquet :
    - `import static org.mockito.Mockito.*;`
  - On crée les doublures par la méthode `mock()` :
    - `MonInterface mi = mock(MonInterface.class);`
  - On décrit le comportement attendu de la doublure :
    - `when(mi.maMethode()).thenReturn(56)`
    - `when(mi.maMethode()).thenThrow(new Exception())`
  - On crée l'objet de la classe à tester en utilisant les doublures à la place des objets réels.
  - On vérifie que l'interaction avec les doublures est correcte par la méthode `Mockito.verify()`.

Exemple de substitut


```
@Test
public void testLireMessagesAvecBonMotDePasse() {
  IdentificationUtilisateur mi;
  mi = mock(IdentificationUtilisateur.class);
  when(mi.lireMessages("toto", "mdp")).thenReturn(true);
  when(mi.lireMessages("toto", "mauvais_mdp")).thenReturn(false);
  // On introduit la doublure
  MessagerieUtilisateur msg = new MessagerieUtilisateur(mi);
  // Oracle
  assertTrue(msg.lireMessages("toto", "mdp"));
}
```

Exemple d'espionnage


```
@Test
public void testLireMessages() {
IdentificationUtilisateur mi;
mi = Mockito.mock(IdentificationUtilisateur.class);
    Mockito.when(mi.identifier("toto", "mdp")).thenReturn(true);
    Mockito.when(mi.identifier("toto", "mauvais_mdp")).thenReturn(false);
    // étape 1 : on introduit la doublure
    MessagerieUtilisateur msg = new MessagerieUtilisateur(mi);
    // étape 2 : on lance le traitement
    try {
        msg.lireMessages("toto", "mdp");
        msg.lireMessages("toto", "mauvais_mot_de_passe");
    catch(Exception e) { }
    // étape 3 : vérifions que la méthode identifier() a bien été
    // appelée exactement une fois avec ces paramètres
    assertTrue(Mockito.verify(mi, times(1)).identifier("toto", "mdp"));
    // et exactement une fois avec ces paramètres
    assertFalse(Mockito.verify(mi, times(1)).identifier("toto",
                                                        "mauvais_mdp"));
}
```

Plan du chapitre

1
Généralités sur les tests

2
Tests unitaires (avec JUnit)

3
Test en isolation et doublures (avec Mockito)

4
Développement dirigé par les tests

Quand tester ?

- Le plus tôt possible
  - Après l'écriture du code d'une fonctionnalité.
  - Après la découverte d'un bug.
- Et pourquoi pas avant l'écriture du code d'une fonctionnalité : TDD.

TDD : Test Driven Development

- Méthodologie de développement.
  - Issue de la méthode *Extreme-Programming*.
  - On écrit les tests avant d'écrire le code.
- La conception est aussi dirigée par les tests.
  - Émerge au fil du développement.
  - Seule l'architecture globale est décidée a priori.
- Règle d'or du TDD :
  - « Ne jamais écrire une ligne de code fonctionnel sans qu'une ligne de code de test ne l'exige. »

En pratique : le mantra du TDD

- Mantra
  - Garder la barre verte pour garder le code propre.
    - Red - Green - Refactor
- Et plus précisément :
  1. Écrire un test pour une fonctionnalité à développer.
  2. Exécuter et constater que la barre est rouge.
  3. Écrire le code qui permet de faire passer le test (et rien que ce code).
  4. Lancer le test et vérifier qu'il passe, barre verte.
  5. Relancer tous les tests précédents.
  6. Remanier le code.

Pourquoi écrire un seul test à la fois ?

- Développement itératif.
  - Optique du « petit pas ».
  - Le test représente une partie du contrat total avec la fonctionnalité testée.
  - On ne passe au comportement suivant que lorsque le précédent a été validé.
    - Validé = implanté et test passé.

Pourquoi ne pas écrire tout le code fonctionnel d'un coup ?

- On n'écrit que le code qui a besoin d'être validé par un test.
- Sinon on risque :
  - D'introduire du code applicatif non testé.
  - D'ajouter des fonctionnalités inutiles.

Pourquoi exécuter le test avant d'écrire le code ?

- Pour commencer par une barre rouge.
  - S'assurer que le test ne passe pas et éviter les « *happy tests* ».
- Permet de détecter des étourderies :
  - Construire le test par copier-coller d'un autre test, et oublier de le modifier (⇒ barre verte).
  - Oublier d'annoter la méthode de test par @Test ⇒ ne teste pas.
  - Construire un test qui s'exécute avec la barre verte sans le code à tester.
  - Oublier ce que devait être l'oracle.

Avantages psychologiques

- Travailler plus sereinement.
  - Si un bug apparaît, il sera détecté très tôt par les tests.
  - Grâce aux petits pas, on a moins peur d'attaquer une tâche complexe.
  - Grâce au code testé :
    - On est serein.
    - On planifie mieux son travail.
    - On évite les paniques de dernière minute.
- Focaliser sur la tâche.
  - Oblige à réfléchir à ce que doit faire le code avant de coder.

Avantages techniques

- Garder un temps de développement constant
  - Tout au long du développement, il y aura le même temps consacré aux tests et au développement.
  - On passe beaucoup moins de temps à déboguer.
- On a toujours quelque chose à montrer au client (dont des tests).
  - Impossible de livrer du code non testé.
  - Tests de **non-régression** directement inclus.
  - L'architecture est testable et de bonne qualité.
- Quand on écrit le test, on ne se réfère qu'à la spécification et pas à l'implémentation (test boîte noire).
  - Moins de biais.
  - Moins de dépendance au code.

Que retenir de ce chapitre ?

- Les tests sont une obligation.
- Ils doivent forcément accompagner le code d'une application.
  - Un code sans test est inutile.
- Il y a différents types de test selon le niveau considéré.
  - Unitaire : test en isolation (utilisation de bouchons si nécessaire)
  - Intégration.
  - Système.
- Le code des tests étant du code, il doit donc être propre au même titre que le code source.
- Le TDD est un processus de conception, pas une procédure de test.