Test logiciel

Michel Plasse (mplasse@free.fr)

Objectif

- Avoir une **méthode** pour développer un projet

1 Méthode

1.1 Tests unitaires

1.1.1 Plusieurs types de test

- Tout besoin exprimé par le client doit pouvoir être **validé** par un ou plusieurs tests.
  - => Le test doit être pensé dès l'expression du besoin
- Différents types de test :
  - **Unitaire** (= non régression) : chaque fonction/procédure suit-elle sa spécification ?
  - **Intégration** : les briques développées séparément fonctionnent-elles correctement ensemble ?
  - **Fonctionnel** : l'application traite-t-elle correctement les cas d'utilisation ?
  - **Montée en charge** : les performances restent-elles acceptables quand la charge augmente ? (volume de données, nombre de requêtes/seconde, nombre d'évènements, etc.)

1.1.2 Tests de non régression

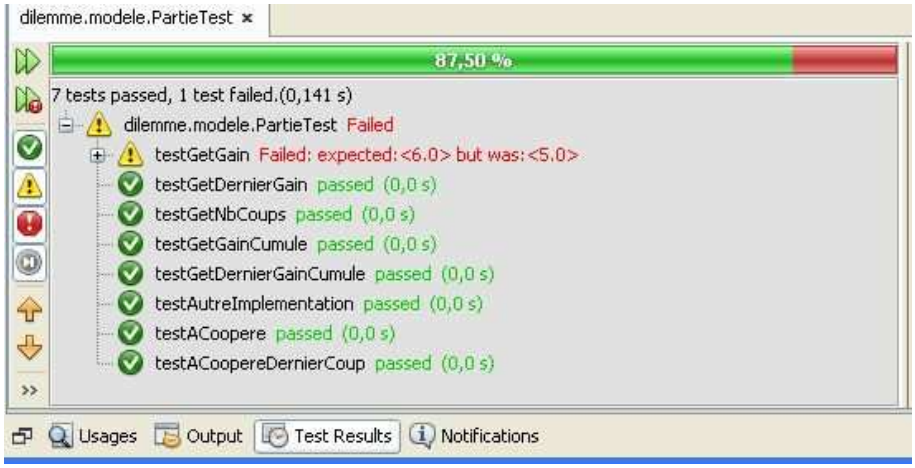
- Les classes d'une application évoluent au fur et à mesure que nous implémentons les fonctionnalités.
  - => Besoin crucial : vérifier que ce qui marchait avant une modification du code marche toujours.
  - => Besoin récurrent : face à un bug complexe, isoler la partie du code fautive.
- Les tests unitaires répondent à ce besoin : **les tests unitaires sont essentiels** aujourd'hui.
- Outil standard pour cela en Java : JUnit.
  Cet outil se décline dans la plupart des langages.

1.1.3 Les 3 composantes d'un test unitaire

- En anglais : **Given - When - Then**.
- Exemple :
  - **Etant donné** un solde positif de mon compte, et aucun retrait cette semaine,
  - **Lorsque** je retire un montant inférieur à la limite de retrait,
  - **Alors** mon retrait doit se dérouler sans erreur ou avertissement.
- Similaire à la conception par contrat : prérequis (conditions avant), action, garantie (conditions après).

1.1.4 Utiliser JUnit

- Nous écrivons une classe de test pour chaque classe du modèle, voire de l'IHM.
- Dans chacune, nous écrivons des tests sous la forme de procédures :
  1. Initialiser les données (**Given**)
  2. Effectuer l'opération à tester (**When**)
  3. Affirmer que telle condition est vraie (**Then**)
- Quand nous lançons JUnit sur une classe de test, il exécute toutes les procédures de test.
- Il produit un rapport visuel, comme (exemple dans netbeans) :



- Les tests réussis sont en vert.
- Les échecs en jaune ou rouge, avec la mention de l'erreur.
- Il indique également un pourcentage de réussite (utile quand on teste l'ensemble d'un projet).

1.1.5 Exemple d'une classe de test (avec JUnit 4)

- Test d'une simple classe `Compteur`, munie de `getNiveau` et `incrementer` :

```
public class CompteurTest {
  private Compteur compteur;

  @Before // La procedure setup sera exécutée avant chaque test
  public void setup() {
    // L'environnement est initialisé
    compteur = new Compteur();
  }

  @Test // la procedure testConstructeur est un test
  public void testConstructeur() {
    int expected = 0;
    int result = compteur.getNiveau();
    // Le test réussit quand la valeur reçue égale la valeur attendue
    // Si échec, JUnit rappelle dans son rapport "expected 0 but was ..."
    assertEquals(expected, result);
  }

  @Test // cette procedure aussi est un test
  public void testIncrementer() {
    compteur.incrementer();
    assertEquals(1, compteur.getNiveau());
    compteur.incrementer();
    compteur.incrementer();
    assertEquals(3, compteur.getNiveau());
  }
}
```

1.1.6 Les annotations

- Une classe de test peut contenir des fonctions, des procédures, des attributs, comme toute autre classe.
- Le nom de ces membres est totalement libre.
- JUnit prend en compte les procédures annotées de :
  - `@Test` : toutes les procédures de test sont ainsi annotées.
  - `@Before` : annote la procédure à exécuter avant *chaque* test.
  - `@After` : procédure à exécuter après *chaque* test.
  - `@BeforeClass` : procédure à exécuter avant *l'ensemble* des tests, pas avant chacun.
  - `@AfterClass` : procédure à exécuter après *l'ensemble* des tests, pas après chacun.

1.1.7 Exemple

Classe de test `MaClasseTest`	Exécution des tests de `MaClasseTest`

```
public class MaClasseTest {
  @BeforeClass
  public void doBeforeClass() { ... }

  @Before
  public void doBefore() { ... }

  @After
  public void doAfter() { ... }

  @AfterClass
  public void doAfterClass() { ... }

  @Test
  public void test1() { ... }

  @Test
  public void test2() { ... }
}
```

```
doBeforeClass();

doBefore();
test1();
doAfter();

doBefore();
test2();
doAfter();

doAfterClass();
```

1.1.8 Usages et conventions

- Par convention :
  - Le test de la classe `MaClasse` est nommé `MaClasseTest`.
  - Le test du membre `unMembre` est nommé `testUnMembre`.
- Penser à mettre `@Test` à chaque test, sinon la procédure sera ignorée par JUnit.
- `@Before` est utile pour dérouler un scénario (faire ceci, puis cela) : chaque test part de ces conditions initiales.
  Indispensable pour les routines accédant à des bases de données (voir support de cours sur les bases de données).
- Dans netbeans, dans l'explorateur de projet, clic droit sur la classe `MaClasse`, puis tools/create test génère la classe `MaClasseTest` avec un squelette de code.
- Il est préférable de mettre les sources et les tests dans deux dossiers différents, `MaClasse` et `MaClasseTest` étant cependant dans le même package.
  C'est ce que fait netbeans automatiquement.

1.1.9 Redéfinition de equals et hashCode

- `assertEquals(Object expected, Object result)` se base sur la méthode `equals` de `Object`.
- Celle-ci compare les références par défaut.
- Pour nos classes métier, il est bon de redéfinir ce `equals`, avec le sens « est identique à ».
- Il faut alors redéfinir aussi `hashCode`. Netbeans génère le code pour ces deux méthodes.

1.2 Conception par contrat

- Chaque opération d'une classe peut nécessiter des conditions pour être appelée : des **prérequis**, ou préconditions.
  Ex : enlever le sommet d'une pile *requiert* qu'elle ne soit pas vide.
- Cette opération peut aussi assurer des conditions après son exécution : des **garanties**, ou postconditions.
  Ex : ajouter un élément à une pile *garantit* qu'elle n'est pas vide après.
- Enfin, certaines conditions sont vraies dès l'instanciation, et après chaque opération : des **invariants**.
  Ex : une pile est vide si et seulement si son nombre d'éléments est vide.
- Détails sur http://www.eiffel.com/values/design-by-contract/.

1.2.1 Exemple

- Le langage *eiffel* implémente complètement la programmation par contrat, depuis l'origine (1987).
- Exemple de la pile avec une syntaxe eiffel : (les commentaires commencent par `--`)

```
class Pile[T]
feature -- groupe de membres
  empiler(element: T)
    require
      not plein
      element != null
    do
      -- (implementation)
    ensure
```

```
      not vide
      nb_elements = old nb_elements + 1
      sommet = element
    end

  sommet: T
    require
      not vide
    do
      -- (implementation)
    end

  vide: BOOLEAN
  plein: BOOLEAN
  nb_elements: INTEGER

invariant
  vide = (nb_elements = 0)
  vide implies not plein
  plein implies not vide
```

1.2.2 Spécification

- Les clauses `require`, `ensure` et `invariant` **spécifient** avec précision les opérations de la classe :
- C'est utile pour la *documentation* : l'équivalent de la javadoc les font apparaître.
- Partage clair des responsabilités :
  - L'appelant doit vérifier les prérequis avant l'appel de la méthode.
  - L'appelé doit satisfaire les garanties.
- Réfléchir aux contrats d'une classe aide à la rendre *cohérente*, et suggère des fonctions utiles (ex : `vide`)

1.2.3 Mise au point et test

- C'est utile aussi pour la *mise au point*, car les assertions génèrent du code, si nous compilons avec vérification des assertions :
  - `require une_condition` fait que si la condition n'est pas satisfaite lors de l'appel, le programme plante en levant une erreur de type « violation de prérequis ».
  - `ensure une_condition` fait que si la condition n'est pas satisfaite après l'appel, le programme lève une erreur « violation de garantie ».
  - `invariant une_condition` lèvera une erreur « violation d'invariant » si la condition n'est pas respectée après l'initialisation de l'objet ou après un appel de méthode.
- Méthode très efficace, qui fait gagner beaucoup de temps.
- Evite beaucoup de code de test unitaire, grâce notamment au mot clé `old`, qui fournit la valeur d'un objet avant l'appel de la méthode.

1.2.4 Java et la programmation par contrat

- Java ne dispose que du mot-clé `assert` (depuis la v1.4), *à ne pas confondre* avec la méthode `assertEquals` de JUnit.
  - Les `if` générés font partie du bytecode.
  - L'option `-ea` de la JVM (ea pour enable assertions) exécute les `if` générés.
  - Pas de mécanisme pour le `old` et pour le `invariant`.
  - La pile donnerait :

```
class Pile<T> {
  public void empiler(T element) {
    assert !estVide();
   // implementation
   // Le ensure n'est pas exprimable => il faut un test unitaire
  }
```

- Les classes du jdk codent en dur les `if` des prérequis, qu'elles expriment par des `throws TelleException`.
  - Problème de conception : les vraies exceptions et les prérequis sont mélangés.
  - Petit impact sur les performances : l'appelé et parfois aussi l'appelant testent les prérequis, si bien que le travail est dupliqué.

1.2.5 Bonnes pratiques

- Spécifier et documenter les prérequis, garanties et invariants de classe : on gagne un temps précieux quand on utilise la classe, on sait ce qu'il faut vérifier avant l'appel de toute méthode.

- Utiliser `assert` pour les prérequis, ou bien prévoir des exceptions ad hoc (ex : `SQLException`) ou génériques :
  - `IllegalStateException` : le prérequis concerne l'objet.
    Ex : la pile est pleine pour `empiler`).
  - `IllegalArgumentException` : le prérequis concerne l'un des paramètres .
    Ex : l'objet à empiler est `null`.
- Tester chaque garantie dans un test unitaire.

1.2.6 Tester des prérequis

- Pour tester des prérequis, JUnit propose de vérifier qu'une exception est bien produite, avec `@Test(expected = UneException.class)`.
- Exemple :
```
// Test produisant une exception attendue
@Test(expected = IndexOutOfBoundsException.class)
public void testElementAt() {
  ArrayList<String> instance = new ArrayList<String>();
  String string = instance.get(2);
}
```
- On peut aussi préciser l'exception :
```
@Test
public void test1() {
  try {
    uneOperationQuiProduitUneException();
    // Si une exception n'est pas levée, échec
    fail("Devrait produire une exception de type UneException");
  }
  catch (UneException exc) {
    assertEquals("Tel message", exc.getMessage());
  }
}
```

1.3 Documentation

1.3.1 Javadoc

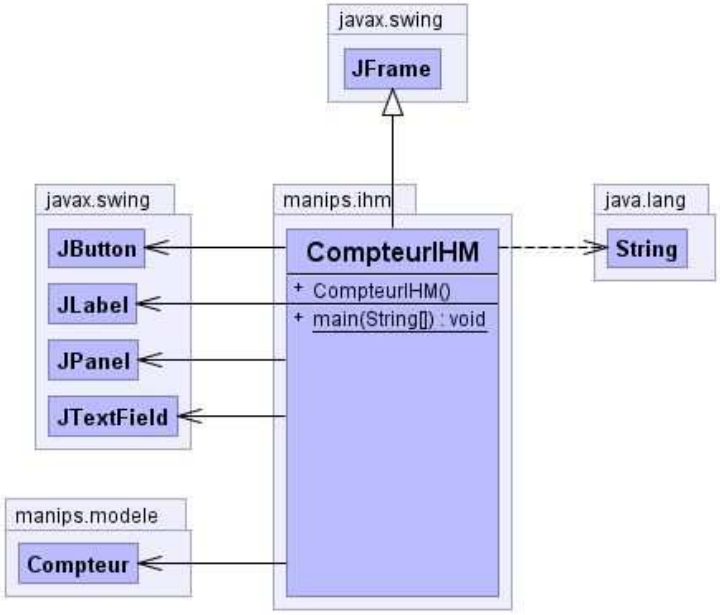
- Sans documentation, une API est très difficile à utiliser, un code très difficile à reprendre et corriger ou étendre.
- L'outil Javadoc transforme les commentaires entre `/**` (avec deux étoiles) et `*/` en documentation.
- Cette documentation est affichée par les IDE comme netbeans et eclipse avec la suggestion de saisie.
  - Très utile quand on code. Les contrats en particulier y sont très utiles.
  - Inutile de lancer l'outil javadoc, l'IDE lit la doc dès que le code source est sauvé (et compilé).
- Elle est rassemblée en HTML : une page par package, et une par classe.
- Les IDE génèrent un squelette de doc dès que nous saisissons `/**`.

1.3.2 Exemple

- Définissons un constructeur dans la classe `Compteur` :
```
public class Compteur {
  /** Constructeur. Met le niveau à 1 et le pas à unPas.
   * Requiert unPas != 0.
   * @param unPas pas du compteur. Doit être non nul.
   */
  public Compteur(int unPas) {
   // implementation
  }
  // etc.
}
```
- Dans les classes utilisant `Compteur`, la documentation apparait lors de la suggestion de saisie :

1.3.3 Schémas UML

- Nous pouvons obtenir des schémas UML via la javadoc grâce à l'outil yworks, disponible gratuitement sur le Web.
- Dans netbeans et à l'ESIAG, indiquer dans les propriétés du projet, au niveau "documenting" (clic droit sur le projet dans "Projects", "Properies", ouvrir "Build", cliquer sur "Documenting" et dans "Additional javadoc options") :
```
-docletpath "chemin-vers\yworks-uml-doclet-3.0_02-jdk1.5\lib\ydoc.jar"
-doclet ydoc.doclets.YStandard
-resourcepath "chemin-vers\yworks-uml-doclet-3.0_02-jdk1.5\resources"
-umlautogen
```
- Cela ajoute dans les pages HTML de la javadoc des schémas comme le suivant, où figurent les packages, les relations d'héritage, de référence via un attribut (flèches à traits pleins) et celles d'utilisation (flèches à traits pointillés).



1.4 Méthodes agiles

- Objectif : délivrer régulièrement le plus de valeur possible et motiver les acteurs (équipe + maître d'ouvrage).
- Le temps est divisé en périodes fixes (**itérations**).
- Le client/MOA et l'équipe définissent en début d'itération les fonctionnalités à délivrer à la fin de l'itération, en fonction de :
  - la **priorité** des fonctionnalités, indiquée par le client/MOA,

    - la **charge** estimée par l'équipe de développement.
- A la fin de chaque itération, le client dispose d'une *application opérationnelle*, qui grossit à chaque itération.

1.4.1 Scénario utilisateur (user story)

- Les fonctionnalités s'expriment avec profit par des scénarios utilisateurs (user stories).
- Exemple pour une banque en ligne :
    - En tant que client, (= **qui**)
    - je peux faire un virement sur un compte d'un autre client depuis le site Web de la banque, (= **quoi**)
    - afin de pouvoir en faire où je veux et quand je veux, plutôt que me déplacer en agence aux heures d'ouverture. (= **pour quoi**)
- Ajoute aux cas d'utilisation de UML, qui comporte rôle et fonctionnalité, la *valeur* produite.

1.4.2 Tests d'acceptation (acceptance test)

- Comment valider qu'un scénario utilisateur est bien implémenté ? Par des tests d'acceptation (tests fonctionnels).
- Exemple :
    1. Le site liste mes comptes pouvant être débités, avec leur solde.
    2. Je peux saisir le montant et le compte à créditer.
    3. L'application vérifie l'existence du compte à créditer en indiquant l'identité de son possesseur.
    4. Elle vérifie que le virement ne mettra pas mon compte débité à découvert.
    5. Elle demande une confirmation du virement.
    6. Toute l'opération se fait de façon sécurisée (via https).
    7. Le virement apparait ensuite dans la liste de mes virements.
- Ces tests s'expriment eux aussi en langage de tous les jours ou métier.

1.4.3 Scénario vs test

- Scénarios et tests s'expriment en *langage de tous les jours* ou dans le *langage métier du client* : ils sont donc compréhensibles par lui.
- Les scénarios sont *succincts*, si bien que leur liste se parcourt assez vite : utile pour avoir une idée globale du produit.
- Les tests d'acceptation sont beaucoup plus *détaillés* : ils listent les points clé à vérifier.
- Scénarios comme tests ne rentrent pas dans les détails techniques (ex : position d'un bouton dans une IHM).
  Ces détails se règlent au jour le jour : le client/MOA est facilement accessible en méthode agile.

1.4.4 Valider une itération

- A la fin d'une itération, une fonctionnalité prévue est considéré comme valide quand :
    - Tous ses **tests réussissent** :
        - Tests fonctionnels, validés par le client/MOA.
        - Tests unitaires.
    - Le **code est propre, simple et nettoyé de ce qui ne sert pas** (ce point demande une grande vigilance).
    - La **documentation** (notamment JavaDoc) est produite.

2 Tests d'IHM Web : Selenium

Voir documentation sur http://docs.seleniumhq.org/docs/02_selenium_ide.jsp.