# Classes imbriqués, gestion de projet, TDD et principes solides

Arnaud Labourel arnaud.labourel@univ-amu.fr

26 novembre 2021

Section 1

# Classe interne

# Classe imbriquée statique

Il est possible de définir une classe à l'intérieur d'une autre (classe imbriquée ou *nested class*) :

```
public class LinkedList {
    public static class Node {
        private String data; private Node next;
        public Node(String data, Node next) {
            this.data = data; this.next = next;
        }
    }
}
```

Il est possible d'instancier la classe interne sans qu'une instance de `LinkedList` existe car elle est statique :

```
LinkedList.Node node = new LinkedList.Node("a", null);
```

# Classe imbriquée statique

## Rappel

Une classe non-imbriquée publique (`public`) doit être dans un fichier portant son nom.

## Interdit !

Fichier `LinkedList.java`

```
public class LinkedList {/*...*/}
public class Node {/*...*/}
```

⇒ erreur à la compilation :

```
Error:(9, 8) java: class Node is public, should be
  declared in a file named Node.java
```

## Remarque

Une classe imbriquée peut être publique et accessible depuis l'extérieur.

# Classe imbriquée statique

Il est également possible de la rendre privée à la classe `LinkedList` :

```
public class LinkedList {
    private static class Node {
        private String data;
        private /*LinkedList.*/Node next;
        public Node(String data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Dans ce cas, seules les méthodes de `LinkedList` pourront l'utiliser.
Notez que des méthodes statiques définies dans `LinkedList` peuvent également utiliser cette classe interne du fait qu'elle soit statique.

# Classe imbriquée statique

Exemple d'implémentation de méthodes dans la classe LinkedList :

```
public class LinkedList {
    /* Code de la classe interne statique Node. */
    private Node first = null;
    public void add(String data) {
        first = new Node(data, first);
    }
    public void print() {
        Node node = first;
        while (node!=null) {
            System.out.print("["+node.data+"]");
            node = node.next;
        }
    }
}
```

# Classe imbriquée statique

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList();
list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c][b][a]
```

# Classe imbriquée statique

Une classe imbriquée statique ne peut accéder qu'aux attributs et méthodes statiques de la classe qui la contient :

```
public class LinkedList {
    private static class Node {
        /* Champs et méthodes de Node. */
        boolean isFirst() {
            return this==first; // interdit !
        }
    }
    private Node first;
    /* Autres champs et méthodes de LinkedList. */
}
```

# Classe interne

En revanche, si la classe interne n'est pas statique, elle peut accéder aux champs de classe qui la contient :

```
public class LinkedList {
    private class Node {
        /* Champs et méthodes de Node. */
        private boolean isFirst() {
            return this==first; // autorisé !
        }
    }
    private Node first;
    /* Autres champs et méthodes de LinkedList. */
}
```

# Classe interne

Java insère dans l'instance de `Node` une référence vers l'instance de `LinkedList` qui a permis de la créer :

```
public class LinkedList {
  private class Node {
   /* Champs et méthodes de Node. */
    private boolean isFirst() {
      return this==/*référenceVersLinkedList.*/first;
    }
  }
  public void add(String data) {
    first = new Node(data, first);
  }
  /* Autres champs et méthodes de la classe LinkedList.
}
```

# Classe interne

Il est possible d'utiliser la méthode `isFirst` dans `LinkedList` :

```
public class LinkedList {
  /* Code de la classe interne statique Node
     et champs et méthodes de la classe LinkedList. */
  public void print() {
    Node node = first;
    while (node!=null) {
      System.out.print("["+node.data
          +","+node.isFirst()+"]");
      node = node.next;
    }
  }
}
```

# Classe interne

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList(); list.add("a");
list.add("b");
list.add("c");
list.print();
```

Cet exemple produit la sortie suivante :

```
[c,true][b,false][a,false]
```

Section 2

# La gestion de projet

# Analyse fonctionnelle

1. Rechercher et caractériser les fonctions qu'un logiciel devrait avoir pour satisfaire les besoins de son utilisateur.
2. Hiérarchiser ces fonctions.

Utilisée pour **créer** mais aussi **améliorer** un logiciel (ou plus généralement un produit).

# Identification des fonctions

- Fonction **principale** : la raison pour laquelle le logiciel est créé. Cette fonction peut être divisée en plusieurs fonctions simples.
- Fonction **contrainte** : conditions que le produit doit vérifier mais qui ne sont pas sa raison d'exister (par exemple la sécurité).
- Fonction **complémentaire** : ce qui facilite l'utilisation du logiciel, l'améliore ou le complète.

# Cahier des charges

Un outil de l'analyse fonctionnelle : lister, décrire et hiérarchiser les fonctions.

## Exemple de la tondeuse

Fonction principale :

- Couper l'herbe.

Fonctions contraintes :

- Respecter les normes de sécurité;
- Pouvoir être conservée à l'extérieur;
- S'adapter au terrain;
- Ne pas être trop bruyante;
- Ne pas être trop encombrante;
- etc,...

Fonctions complémentaires:

- Ramasser l'herbe;
- Être automatique.

# Quelques notions de gestion de projet

- **Livrable** : produit destiné à la livraison : documentation, code, tests, etc...
- **Jalon** : fin d'une étape ou évènement important.

Le début d'un projet et sa fin sont des jalons. On fixe des jalons intermédiaires pour mesurer l'avancée du projet.

Un jalon peut être un livrable lié à une date.

# La gestion de projet en informatique

Développer un logiciel implique de :

- comprendre le besoin du client;
- en tirer un cahier des charges;
- concevoir l'architecture du logiciel;
- développer le logiciel;
- tester s'il fonctionne comme prévu;
- le maintenir.

Deux types de management pour y parvenir :

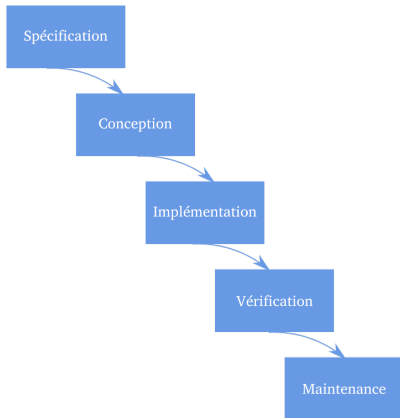
- méthodes traditionnelles;
- méthodes agiles.

# Cascade (ou Waterfall)

Méthode traditionnelle, inspirée par le BTP.

Passage d'une phase à l'autre uniquement quand la précédente est terminée et vérifiée.

# Cascade

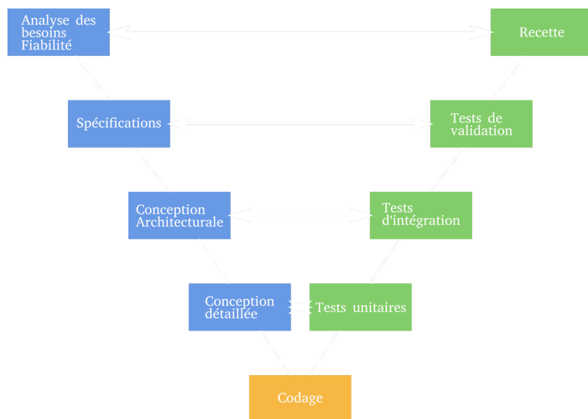



Inconvénient : très peu réactif en cas d'erreur ou de modification nécessaire en cours de projet.

# Cycle en V

Un amélioration du modèle en cascade : limiter le retour aux étapes précédentes.

Standard depuis 1980.



# Pro et con du cycle en V

**Phase ascendante :** renvoie de l'information sur la phase correspondante pour améliorer le logiciel.

**Phase descendante :** anticipation des attendus des étapes montantes.

**Inconvénient :** détache complètement la conception de la réalisation.

# Agile

Objectifs :

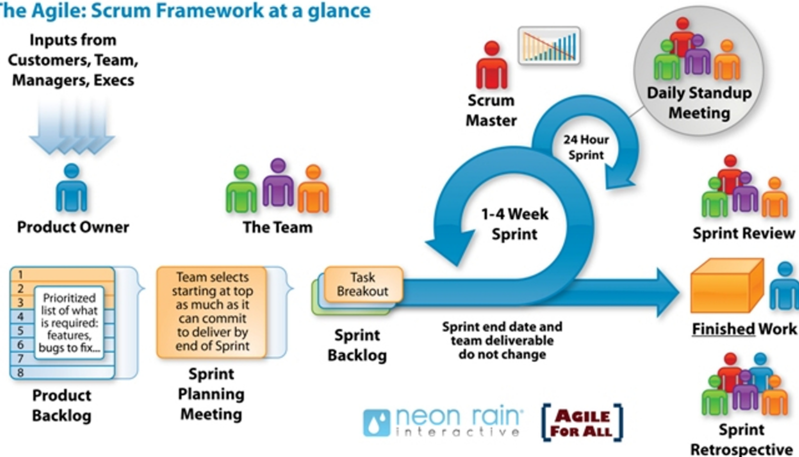
- plus pragmatique que les méthodes traditionnelles;
- impliquer le client pendant le développement;
- être réactif et s'adapter aux changements.

Définition par le manifeste Agile (2001).

Exemples :

- développement rapide d'applications (RAD);
- adaptative software development;
- extreme programming (XP);
- Scrum.

# SCRUM




Source : agileforall.com

Section 3

# Tests unitaires et développement par les tests (TDD)

# Pourquoi les tests unitaires ?

- assurer la correction du code;
- trouver rapidement les bugs pendant le développement;
- identifier des manques dans les spécifications;
- faciliter les modifications.

# Cycle de développement du TDD

1. Écrire un test : définit une fonction ou l'amélioration d'une fonction.
2. Lancer le test : il doit échouer (pour montrer que le test fait référence à une fonctionnalité qui n'existe pas encore, et qu'il fonctionne bien).
3. Écrire le code qui fait passer le test (et rien d'autre).
4. Lancer les tests : si le test ne passe pas, retourner à l'étape 3.
5. Refactorer : déplacer du code si besoin, supprimer la duplication, vérifier les noms...

Théorisé par Kent Beck

# Intérêts

- réfléchir à ce que fait le code avant de coder;
- garantir que tout est testé (puisque rien n'est écrit sans test);
- réduire significativement le temps passé à debugger;
- avancer par petits pas;
- voir son avancée.

Section 4

# Les principes solides

# Les cinq principes pour créer du code SOLID

- **Single Responsibility Principle (SRP) :** Une classe ne doit avoir qu'une seule responsabilité
- **Open/Closed Principle (OCP) :** Programme ouvert pour l'extension, fermé à la modification
- **Liskov Substitution Principle (LSP) :** Les sous-types doivent être substituables par leurs types de base
- **Interface Segregation Principle (ISP) :** Éviter les interfaces qui contiennent beaucoup de méthodes
- **Dependency Inversion Principle (DIP) :** Les modules d'un programme doivent être indépendants. Les modules doivent dépendre d'abstractions.

# Single Responsability Principle (SRP)

## Principe SRP

Une classe ne doit avoir qu'une **responsabilité = raison de changer**

## Les effets néfastes des responsabilités multiples

- Difficulté à nommer car la classe est trop complexe
- Difficulté à réutiliser le code car seulement une des responsabilités est réutilisable
- Difficulté à mettre à jour le code car changer l'implémentation d'une partie impacte les autres parties

## Pourquoi SRP ?

- Facilite le nommage et documentation
- Facilite l'écriture des tests
- Facilite la réutilisation des classes

# Open/Closed Principle (OCP)

## Principe OCP

Programme ouvert pour l'extension, fermé à la modification

## Signification

Vous devez pouvoir ajouter une nouvelle fonctionnalité :

- en ajoutant des classes (Ouvert pour l'extension)
- sans modifier le code existant (fermé à la modification)

## Avantages

- Le code existant n'est pas modifié ⇒ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

# Violation OCP (1/2)

```
public class Circle {
    Point center;
    int radius;
}

public class Rectangle {
    Point TopLeft, RightBottom;
}
```

# Violation OCP (2/2)

```
public class GraphicTools {
    static void draw(Rectangle r){}
    static void draw(Circle c){}
    static void draw(Object[] shapes){
        for(Object o : shapes){
            if(o instanceof Rectangle){
                draw((Rectangle) o);
            }
            if(o instanceof Circle){
                draw((Circle) o);
            }
        }
    }
}
```

# Programme propre (1/2)

```
interface Shape {
  void draw();
}

public class Circle implements Shape {
    Point center;
    int radius;
    public void draw(){ ... }
}

public class Rectangle implements Shape {
    Point TopLeft, RightBottom;
    public void draw(){ ... }
}

public class GraphicTools {
```

# Liskov Substitution Principle (LSP)

## Principe

Les sous-types doivent être substituables par leurs types de base.

## Signification

Si une classe **A** étend une classe **B** alors un programme **P** écrit pour manipuler des instances de type **B** doit avoir le même comportement s'il manipule des instances de la classe **A**.

## Avantages

- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité

# Liskov Substitution Principle (LSP)

Violation de LSP :

```
public void test(Rectangle r) {
  r.setWidth (2);
}
r.setHeight(3);
if (r.getArea()!=3*2)

System.out.println("bizarre !");
```

**La mauvaise question :**

Un carré est-il un rectangle ?

**La bonne question :**

Pour les utilisateurs, votre carré a-t-il le même comportement que votre rectangle ?