Programmation 2 : Huitième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

21 octobre 2019

Une méthodologie pour bien nommer

Pourquoi bien nommer est important

Albert Camus (1944)

“Mal nommer un objet, c’est ajouter au malheur de ce monde”

Que veut dire le texte suivant ?

La L3 info : MIAGE ne dépend pas de la même UFR que la L2 info : MI, elle dépend de la FEG et non de la FS. Pour faire vos IA et IP, vous devez donc contacter la scol de Forbin et non celle de SCH.

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage

En anglais

- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards).
  $\Rightarrow$ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
$\Rightarrow$ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet ⇒ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashcode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Nommage des méthodes

Méthodes de conversion $\Rightarrow$ utilisation du to

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les partie du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confier à d'autres méthodes

Ne pas mentir

```
class User {
    private boolean authenticated;
    private String password;

    public boolean checkPassword(String password) {
        if (password.equals(this.password)) {
            authenticated = true;
            return true;
        }
        return false;
    }
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom.

Des principes pour bien programmer

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
void cook(){
    // On choisir la recette
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);

    // On réunit les ingrédients
    openFridge();
    (for Ingredient ingredient : recipe.getFreshIngredien
        takeInFrige(ingredient);
    }
    closeFridge();
    openCupboard();
    ...

    // On suit la recette
```

Bonne approche

```
void cook(){
    Recipe recipe = chooseRecipe();
    gatherIngredients(recipe);
    followRecipe(recipe);
}

void chooseRecipe(){
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
}

...
```

Programme bien conçu

Un programme est “bien conçu” s’il permet de :

- Absorber les changements avec un minimum d’effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs

- Limiter les modules impactés
  - Simplifier les tests unitaires
  - Rester conforme à la partie des spécifications qui n’ont pas changé
  - Facilité l’intégration
- Gagner du temps

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 :

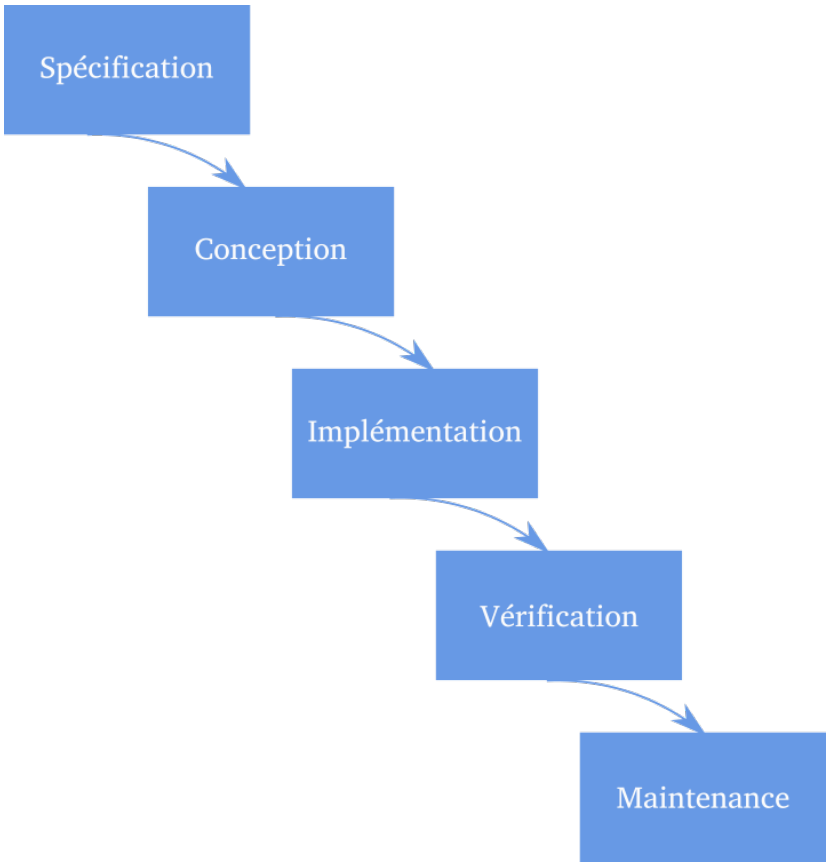
- méthodes traditionnelles;
- méthodes agiles.

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

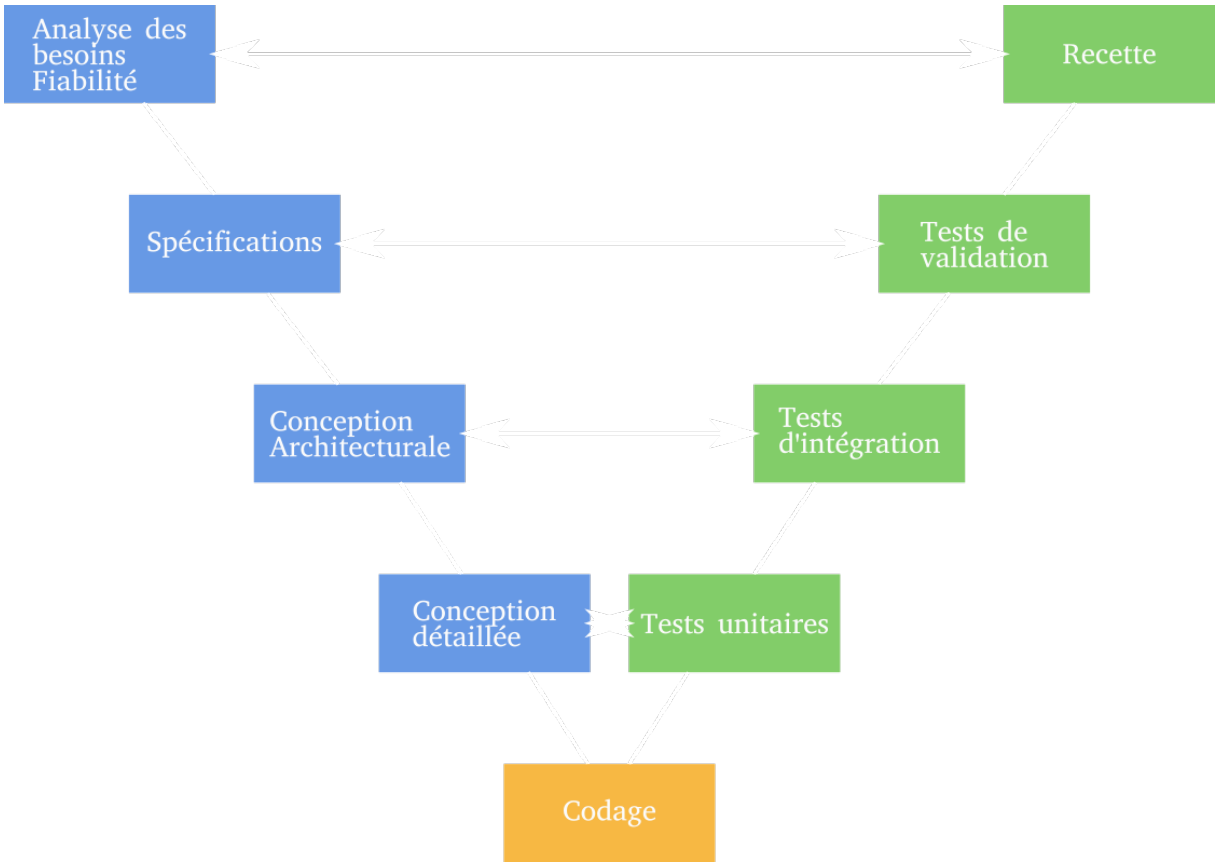


Inconvénient : très peu réactif en cas d'erreur ou de modification nécessaire en cours de projet.

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 :

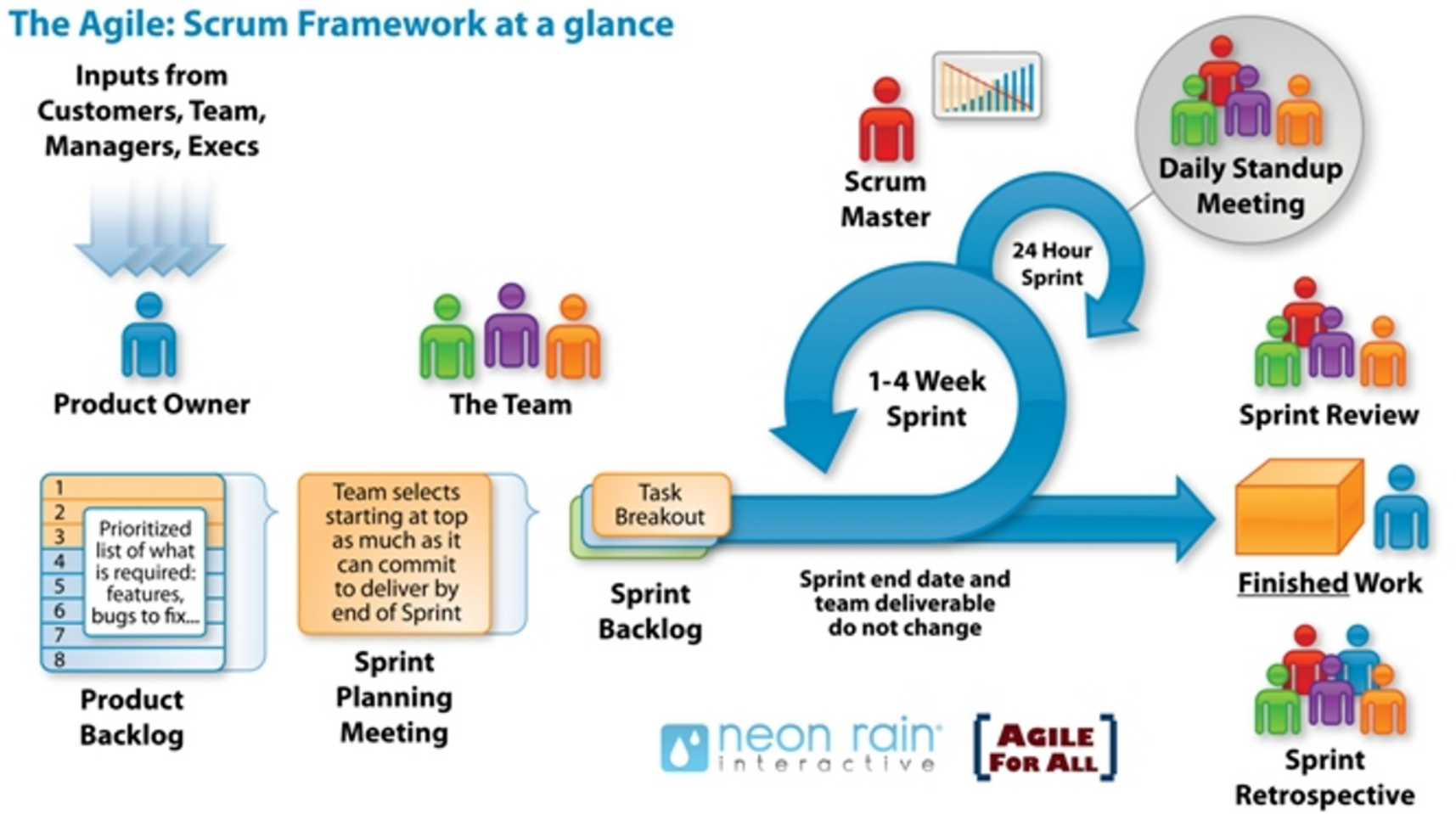
- plus pragmatique que les méthodes traditionnelles;
- impliquer le client pendant le développement;
- être réactif et s'adapter aux changements.

Définition par le manifeste Agile (2001).

Exemples :

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

SCRUM



Source : agileforall.com

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.

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 $\Rightarrow$ 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 ?