Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université
Socialement engagée

Année 2023-24

Programmation avancée : Bonnes pratiques
Code UE : SBICU05

Bonnes pratiques de programmation

Introduction

Avant de présenter les principes SOLID qui sont cinq principes de conception orienté objet visant à produire des architectures logicielles qui sont plus compréhensibles et maintenables, il est important de faire un rappel sur les bonnes pratiques de programmation qui permettent de rendre le code plus lisible. Une des choses les plus importantes lorsqu'on écrit du code est de bien nommer les éléments du code. Nous allons donc commencer par expliquer les principes à respecter pour avoir un nommage efficace.

Une méthodologie pour bien nommer

Trouver des noms appropriés peut prendre beaucoup de temps sur le moment, mais va vous permettre de gagner du temps par la suite. Les conventions de nommage sont extrêmement importante pour la maintenance et la lisibilité d'un programme. Il est donc important de :

- utiliser des noms cohérents pour tous les symboles :
- choisir un nom qui correspond au but/rôle du symbole (Le nom doit révéler le rôle de l'élément) ;
- utiliser l'anglais pour nommer vos éléments ;
- choisir un nom en regardant le code source des *librairies* sérieuses comme celles fournies par la JDK.

Les raisons de l'importance du nommage sont les suivantes.

- Un nom bien choisi rend plus facile la lecture du programme.
- Lorsqu'on n'arrive pas à choisir un nom adapté, c'est souvent parce que son rôle n'est pas bien défini.
- Un programmeur passe 80% de son temps à lire le programme : il faut lui faciliter ce travail surtout qu'on peut passer beaucoup de temps à relire son propre code.
- Des incohérences logiques évidentes peuvent sauter aux yeux avec un bon nommage.
- Toutes les *librairies* sont codées en anglais.

Nommage des variables/arguments/attributs

Il y a certains écueils à éviter lorsqu'on cherche à trouver des noms pour des variables. 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`, ...
- 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`, ...

Comme toutes les bonnes pratiques, ce ne sont pas des règles absolues et on peut évidemment y déroger. Par exemple, il est généralement accepté d'utiliser `x` et `y` pour les coordonnées d'un Point comme cela est fait pour la classe `Point2D` de JavaFX. Néanmoins, comme pour toutes les règles qui ne sont pas absolues, il est nécessaire de se poser la question de savoir si on fait bien de ne pas les respecter.

Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards). Par conséquent, on se doit de 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 donc d'avoir de nombreux exemples existants pour s'en inspirer.

Une autre règle importante pour le nommage est de respecter le Code Style de Java pour le nommage des variables/arguments/attributs. Pour les noms composés d'un seul mot, on écrit tout simplement le mot en minuscule. Pour un nom composé de plusieurs mots, on n'utilise ni espace ni ponctuation, et on sépare les mots en mettant en capitale la première lettre de chaque mot. Par exemple, cela donne : `flaggedCells`, `gameBoard`, ... Si le nom comportait des lettres en dehors des 26 caractères non accentués classiques de l'anglais (de `a` à `z`), on les remplace par des caractères inclus dans les 26 caractères de base. Par exemple, "root computed by Müller's method" devient `rootComputedByMuellersMethod`.

Nommage des méthodes

Comme pour les variables, il est important de bien nommer les méthodes. Le nom d'une méthode doit décrire le service rendu à celui qui l'appelle, et non pas comment elle le fait. La convention de nommage est la même que pour les variables (capitale pour la première lettre de chaque mot sauf le premier). Dans une très grande majorité des cas, la méthode se trouve dans une des catégories suivantes :

— **Ordre** : méthodes exécutant une action avec comme sujet l'objet avec lequel la méthode est appelée. Dans ce cas, on utilise le groupe verbal à l'infinitif. Cela donne par exemple : `connection.open()`, `list.sort(comparator)`, `comparator.compare(object1, object2)`.
— **Requête booléenne** : méthodes testant un prédicat sur l'objet. Dans ce cas, on utilise un groupe verbal au présent. Cela donne par exemple : `connection.isClosed()`, `list.isEmpty()`, `list.contains(object)`, `node.hasNext()`, `frame.canClose()`.
— **Requête non-booléenne** : méthodes renvoyant une partie de l'état de l'objet. Dans ce cas, on utilise un groupe nominal ou bien un *getter*. Cela donne par exemple : `list.size()`, `connection.getMetaData()`.
— **Conversion** : méthodes convertissant l'objet en un objet d'un autre type. Dans ce cas, on utilise `to` suivi du type ciblé. Cela donne par exemple : `list.toArray(...)`, `object.toString()`.

Comme pour le nommage des variables, ces 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.

Si vous avez des difficultés à nommer vos méthodes, c'est sans doute qu'elles font trop d'actions. Une méthode

devrait avoir au maximum une dizaine de lignes de code. Il est toujours possible de satisfaire à cette contrainte en extrayant le plus possible les parties du code d'une méthode à d'autres méthodes. Il est donc important de :

— réfléchir avant de coder au rôle de la méthode ;
— se demander ce qui peut être confié à d'autres méthodes.

Une fonction ne doit donc faire qu'**une seule chose**. Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction. Nous allons illustrer cela sur l'action de cuisiner. Pour faire la cuisine on doit (premier niveau d'abstraction) :

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

Pour choisir une recette, on doit (deuxième niveau d'abstraction):

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

Considérons le code suivant pour une méthode `cook` :

```
void cook(){
    // On choisit la recette
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    // On réunit les ingrédients
    openFridge();
    for(Ingredient ingredient :
        recipe.getFreshIngredients()){
        takeInFrige(ingredient);
    }
    closeFridge();
    openCupboard();
    ...
    // On suit la recette
    ...
}
```

Dans le code de la méthode `cook`, on utilise un niveau d'abstraction trop bas. Cette approche n'est pas la bonne et on a été obligé de rajouter des commentaires pour indiquer les étapes du premier niveau d'abstraction. La bonne approche consiste à définir des méthodes correspondant aux étapes du premier niveau d'abstraction et de les appeler dans la méthode `cook`. Cela nous donne le code suivant :

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

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

```
        return recipe;
    }

    ...
```

Un autre écueil à éviter est de mentir dans le nom d'une méthode. Par exemple, si on considère la méthode `checkPassword` dans la classe ci-dessous :

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

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

Cette méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom. Il y a donc un mensonge (la méthode fait plus que ce qu'elle dit) ce qui complique fortement la compréhension du code. C'est donc un comportement à éviter.

Nommage des classes/interfaces/records/enums

En Java, pour tous les éléments du code qui correspondent à des types (c'est-à-dire des classes, interfaces, records ou enums), on utilise une majuscule pour la première lettre de chaque mot composant le nom du type (y compris le premier mot contrairement aux variables et méthodes). Cela donne par exemple : `Shape`, `Rectangle`, `ArrayList`, `MountainBike`. Un type définissant généralement une catégorie d'objet, le nom d'une classe/interface/record/enum correspond dans la plupart des cas à un groupe nominal au singulier. Cela vaut en particulier pour les `enum` de java qui doivent être au singulier. L'`enum` `DayOfWeek` qui définit les sept jours de la semaine est au singulier car un objet de type `DayOfWeek` correspond à un jour de la semaine.

Parfois, on a besoin de regrouper un certain nombre de fonctions `static` ensemble. C'est par exemple ce qui se passe pour `Math` qui contient des fonctions mathématiques de base ou Collections qui contient des méthodes s'appliquant sur des collections. Dans ce cas très particuliers, la classe ne permet pas de créer des objets (par exemple le constructeur de `Math` est privé et n'est pas appelé dans la classe) et donc la convention de nommage ne s'applique pas vraiment. Le nom de la classe doit juste donner les liens entre les différentes méthodes statiques qu'on a regroupées dans celle-ci.