Grille d'évaluation d'un programme

Programmation objets, web et mobiles en Java
Licence 3 – Projets web, développement et communication multimedia

Pierre Talbot (ptalbot@hyc.io)
Université de Pierre et Marie Curie

10 septembre 2017

Indentation

L'indentation permet d'organiser son code de manière lisible. On peut utiliser des espaces (généralement 2 ou 4) ou tabulations mais il ne faut pas mélanger les deux. Par exemple :

```
if (x > 9)
    {
  x = x - 1;
}
```

est clairement mal indenté, on doit décaler le code à chaque embranchement (if/while/...) ou fonction/classe/... Une des indentations corrects de l'exemple précédent sera :

```
if (x > 9) {
  x = x - 1;
}
```

La position des `{}` peut être différente. Il faut juste faire partout pareil.

- Indentation correct et cohérente ☐
- Uniformisation du caractère d'indentation (espaces OU tabulations) ☐
- Cohérence des `{}` ☐

Noms

Le nommage des variables, méthodes et classes est très important pour la relecture et pour exprimer clairement ce qu'on veut faire. Dans la *checklist* suivante, on a les choix généralement utilisé pour Java, vous pouvez utiliser d'autres conventions mais soyez cohérent.

- Variables et méthodes nommées en minuscule avec des _ pour les espaces (ex : `user_choice`) ☐
- Classes nommées en *PascalCase* (ex : `UserChoice`) ☐

– Constantes statiques en majuscules (ex : `DAMAGE_PER_SECOND`) ☐

– Utiliser l'anglais est souvent plus judicieux (vu que toutes les classes/méthodes/mot-clés externes sont en anglais) mais le français reste acceptable. Néanmoins vous ne devez pas mélanger ! ☐

Les points suivants sont plus généraux. Une variable doit être nommée avec un nom de plus en plus explicite si elle est loin de sa déclaration, par exemple :

```
static public int add(int a, int b) {
  return a + b;
}
```

est tout à fait valide, la portée de `a` et `b` est très courte donc on peut se permettre des noms courts. Par contre pour les membres d'une classe ou les noms de méthodes, il ne faut pas hésiter à être explicite. Par défaut, toujours donner des noms explicites.

– Uniformité et cohérence des choix de nommage ☐

– Nommage précis et explicite ☐

Factorisation

La factorisation empêche le code d'être répété à plusieurs endroit. Si vous avez envie de faire un copier-coller de code (même si vous le modifier après), c'est généralement mauvais signe. Il faut faire une fonction regroupant le code commun.

– Pas de copier-coller de code ☐

– Pas de répétition dans la logique du code ☐

Fonctions/méthodes/classes

Les fonctions doivent rester courtes et ne faire qu'une chose mais bien. Si vous avez envie de documenter un long code pour dire "étape 1 : vérification qu'une carte n'existe pas déjà", "étape 2 : ajout de la carte dans la collection", ... Ça signifie que votre code doit être divisé en plusieurs sous-fonctions.

– Les fonctions sont assez courtes ☐

– Les classes ne font qu'une chose mais bien, elles n'ont pas 2 responsabilités ☐

Documentation

La documentation ne doit pas décrire ce qui est évident, vous devez préférer une documentation de plus haut niveau. C'est généralement une meilleure idée de décrire en début de fichier le rôle d'une classe et comment celle-ci s'intègre dans le projet / comment l'utiliser, plutôt que de décrire exactement ce que font chacune des méthodes. Normalement, si vous donnez des bons noms et que les méthodes sont assez découpées alors votre code doit être tellement simple que ça se passe de commentaire. Quand on relit votre code, on doit pouvoir comprendre rapidement comment il est architecturé. Vous pouvez également dire pourquoi vous avez choisi une solution plutôt qu'une autre,

pour que les suivants ne se posent pas la question. Demandez-vous ce que les gens qui liront votre code se poseront comme question et répondez-y.

On commente des méthodes publiques lorsque celles-ci ont pour prétention d'être utilisée par d'autres personnes sous forme de librairie, ainsi ils ne sont pas obligé d'aller voir le code source, par exemple les librairies standards en Java (ArrayList, String, ...) sont documentés de la sorte. C'est néanmoins une perte de temps d'être aussi précis dans un projet car vous avez rarement assez de temps pour ça.

- Les classes sont décrites par une documentation de haut niveau ☐
- Le nom et le découpage des méthodes/variables permet de se passer de documentation, le code est la documentation. ☐

Getter/setter

Quand vous ajoutez des méthodes `get` ou `set` à une classe, c'est presque identique à exposer publiquement un membre en `public`. Vous devez faire un effort pour éviter ces méthodes. Posez-vous les questions "Pourquoi ai-je besoin de get/set ?" et "Quels traitements vais-je réaliser avec ?", et normalement vous pourrez déplacer ce traitement dans la dites-classe. Ainsi la classe rendra réellement un service et ne servira pas juste de container. Vos designs n'en seront que plus propre et vous pourrez mieux séparer les traitements. En effet, quand la classe aura trop de responsabilités, il sera temps de la séparer en plusieurs sous-classes.

- Les get/set ne sont presque pas présent dans mon programme ☐
- Les noms des méthodes sont plus judicieux et représente réellement un service que la classe rend ☐
- J'ai considéré l'utilisation d'une classe "container" avec seulement des membres en publique, qui n'hérite pas et ne possède pas de méthode (POJO ou POD) au lieu de mélanger des get/set et des traitements. ☐

Pour ce dernier point, vous avez sûrement ce problème dans le Pokedeck dans la mesure où l'utilisateur doit rentrer des données et donc qu'il faut bien des get/set. Vous pouvez outre-passer ce problème en créant une classe `PokemonCardDescriptor` qui contient seulement les attributs de cette carte (et qui n'hérite de rien, et n'a aucun membres) ET une classe `PokemonCard` contenant cette classe mais proposant de réels services.