

Bonne pratiques

Les bonnes pratiques données ci-dessous viennent principalement de la chaîne YouTube "Code Aesthetics" et du langage de programmation Rust

Nommer les choses

- ☐ Eviter d'utiliser des nombres random dans le code et les remplacer par des constantes
- ☐ Ne pas abrégier le noms des méthodes ou le nom des variables
- ☐ Ne pas répéter les types dans les noms des variables
- ☐ Spécifier les unités dans les variables, ou mieux, créer un type qui le représente
- ☐ Eviter d'utiliser des noms/préfixes génériques tel que "I", "Utils", "Helper", "Abstract" dans les noms des classes
- ☐ Il n'est pas nécessaire de donner des informations sur les détails de fonctionnement (par exemple écrire "Interface" ou "I" dans le nom d'une interface n'est pas nécessaire)

Eviter les commentaires et rendre le code plus robuste

- ☐ Eviter d'utiliser trop de commentaires, car ils peuvent devenir obsolètes et ne sont pas nécessaire quand le code est proprement documenté. A la place essayer de rendre le code si clair qu'il s'exprime comme le commentaire.
- ☐ Utiliser des commentaires quand l'implémentation semble quelque peu bizarre (par exemple pour des raisons d'optimisation)
- ☐ Les conditions peuvent être extraites dans des méthodes séparées
- ☐ Utiliser des enums pour tous les états fixent pour empêcher des états impossibles d'être représenter
- ☐ Si possible utiliser des types tel que "Option" ou "Result" pour obliger de prendre en compte les cas de valeurs `null` ou d'erreurs

Améliorer la lisibilité du code encore plus

- ☐ Eviter d'utiliser plus de 3 niveau d'intendation dans une méthode
 - ☐ Extraire des bouts d'une méthode dans plusieurs autres
 - ☐ Notter les conditions d'une méthode au dessus (cela permet de supprimer les `else`)
- ☐ Ecrire une documentation (comment *utiliser* le code) de très bonne qualité et la garder à jour.
 - ☐ Décrire le but d'une classe et des interfaces et ce qu'elle représente
 - ☐ Décrire les attentes que les interfaces ont par rapport à leurs implémentations (comment elles doivent gérer X ou Y)
 - ☐ Décrire les buts des méthodes, leurs préconditions et leurs post-conditions

Principes orienté objet

- ☐ Eviter de faire des classes intuelles (qui ne font que très peu de choses) → data class
- ☐ Eviter de faire des classes qui font tout (classes dieu)
- ☐ Eviter d'appeler des méthodes d'objets qui ne sont pas des attributs, des paramètres, créé par, ou de même classe dans une méthode. → éviter d'appeler des méthodes d'objets retournés par une autre méthode. Et éviter de les donner en premier lieu
- ☐ Rendre tous les attributs `private` et rendre un maximum de classes immuable si possible
- ☐ Vérifier les paramètres des méthodes et des constructeurs pour empêcher que des états invalides se présentent (par exemple avec des valeurs `null`)
- ☐ Quand quelque chose en lien direct avec les attributs est renvoyé, s'assurer qu'il est immuable
- ☐ Eviter de faire en sorte que les autres classes fassent le boulot d'une classe à sa place → extraire des bouts de méthode dans une autre classe
- ☐ Eviter les dépendances mutuelles ou circulaires entre classes
- ☐ Empêcher l'héritage en mettant `final` sur toutes les classes

MVC et MVP

- ☐ Les classes métier ne doivent avoir aucun contact avec des classes de la vue
- ☐ La vue peut être liée au superviseur par le biais d'une interface (on crée d'abord le superviseur, crée une vue qui implémente une interface de vue en lui passant le superviseur, on donne la vue à travers l'interface au superviseur)
- ☐ Pour des objets qui doivent avoir un état partagé entre tous les superviseurs on peut utiliser une Factory qui va être passée à tous les superviseurs et qui va garder en mémoire l'objet en cours pour le créer et le récupérer)
- ☐ Dans le cas d'un *presenter (MVP)* les superviseurs vont récupérer les informations brutes depuis les classes métier, les process et les afficher dans la vue
- ☐ Dans le cas d'un *controlleur (MVC)* les superviseurs vont récupérer les informations déjà bien formatée et vont simplement les mettre dans la vue sans plus de traitement
- ☐ Les classes métier ne doivent pas dépendre du superviseur

Tests unitaires

- ☐ Créer une interface d'aléatoire et une fausse et vraie classe aléatoire pour les tests (et faire une surcharge de constructeurs)
- ☐ Tenter d'obtenir un code coverage de 90% au minimum sur toutes les classes
- ☐ Décrire les tests sous forme de "Étant donné *une condition* Quand *une action* Alors *une/plusieurs postcondition*"

Git

- ☐ Do not make bundled commits of a lot of things, really commit one thing at the time
- ☐ You can use emojis in your commits to represent different sort of tasks (documentation, refactoring, testing, etc) see [GitMoji](#)
- ☐ Add a README (you can pimp it using [shields](#), GIF, installation instructions, usage instructions, license information, general description, and features)

Revision #5

Created 5 May 2023 23:25:43 by SnowCode

Updated 10 May 2023 15:47:53 by SnowCode