

Sonar

- Est-ce que je me suis assuré qu'il n'y ait plus d'anomalies dans Sonar dans les classes que j'ai ajoutées / modifiées ?
- Est-ce que mon code respecte les standards de sécurité (owasp) ?
- Est-ce que j'ai une couverture de code qui se rapproche de 100% ?

Standard Desjardins

- J'utilise le français lorsque applicable et je vérifie la syntaxe
- J'ai fait un squash de mes commits

Réflexivité

- Est-ce que j'ai évité d'utiliser la réflexion ?
- Peut briser l'encapsulation, moins performant

DRY (Don't repeat yourself)

- Éviter la duplication à l'intérieur d'une même application

Accès

- Est-ce que les classes, méthodes et propriétés ont l'accès minimal ?
- Est-ce que tous les get et set sont nécessaires, avec les bons accès ?
- Respecter l'encapsulation, facilite les tests, diminue la duplication

Utiliser les design patterns avec parcimonie

- Utiliser les design patterns seulement pour régler un réel problème

Single Responsibility principle

- Une classe devrait avoir une seule raison de changer
- Robustesse du code, moins de bugs sur les autres fonctions. Une seule fonctionnalité testée avec un junit
- https://en.wikipedia.org/wiki/Single_responsibility_principle

Open-Closed principle

- Une classe ne devrait plus être modifiée, on devrait ajouter du code, pas modifier du code fonctionnel
- Faciliter la maintenance du code et la réutilisation
- https://en.wikipedia.org/wiki/Open/closed_principle

Liskov Substitution principle

- Si on vérifie un type d'objet, on ne respecte pas le principe
- La sous-classe doit pouvoir recevoir les mêmes types de paramètres que la superclasse. La sous-classe devrait retourner le même type que la superclasse ou un objet parent du type. La sous-classe doit lancer les mêmes exceptions ou les sous-type de l'exception de la superclasse
- https://en.wikipedia.org/wiki/Liskov_substitution_principle

Interface Segregation principle

- Faire de petites interfaces au lieu de grosses
- Éviter que l'interface ne devienne trop spécifique et non réutilisable
- https://en.wikipedia.org/wiki/Interface_segregation_principle

Dependency Inversion principle

- Utilise des interfaces au lieu de classes
- Permet de diminuer les impacts quand on change l'implémentation et facilite les tests
- https://en.wikipedia.org/wiki/Dependency_inversion_principle

Récurivité

- Est-ce que j'ai évité d'utiliser la récursion s'il y a une meilleure façon de faire ?
- Difficile à déboguer, moins performant

Singleton

- Une seule instance de classe par jvm nécessaire
- Éviter que plusieurs threads exécutent le même code au même moment, ie façade et gestionnaire d'accès
- https://sourcemaking.com/design_patterns/singleton



Strategy Pattern

Permet de sélectionner un algorithme à l'exécution

Éviter d'avoir des conditions à plusieurs endroits pour vérifier l'algorithme à exécuter

https://sourcemaking.com/design_patterns/strategy

Factory Pattern

Permet de créer de nouveaux objets sans connaître les détails de construction ou de dépendances

<https://softwareengineering.stackexchange.com/questions/253254/-why-should-i-use-a-factory-class-instead-of-direct-object-construction>

https://sourcemaking.com/design_patterns/factory_method

Chain of Responsibility Pattern

Permet de définir une série d'objets qui peuvent répondre à une requête selon des conditions fixées à l'exécution, ou de déléguer la suite du traitement à un autre objet de la série

ie Filter

https://sourcemaking.com/design_patterns/chain_of_responsibility

Template Method Pattern

Définit le squelette d'un algorithme, en déléguant certaines parties de l'algorithme à des sous-classes

Éviter d'avoir de la duplication dans des composants similaires

https://sourcemaking.com/design_patterns/template_method

State Pattern

Permet de changer l'exécution d'un objet selon son état interne

Éviter succession de if dans le code

https://sourcemaking.com/design_patterns/state

Adapter Pattern

Permet de convertir l'interface d'une classe en une autre interface attendu

Permet d'accéder à des fonctionnalités non disponibles dans l'api courant

https://sourcemaking.com/design_patterns/adapter

Decorator Pattern

Ajouter des responsabilités à un objet dynamiquement. Aussi appelé wrapper

https://sourcemaking.com/design_patterns/decorator

Builder pattern

Permet d'éviter la multiplication des constructeurs lorsqu'il y a des paramètres optionnels

https://sourcemaking.com/design_patterns/builder