|  |  |  |  |
| --- | --- | --- | --- |
| **DRY (Don’t Repeat Yrslf)** | **KISS (Keep It Simple Stupid)** | **SOLID** | **STUPID** |
| Limiter répétition  Simplifie maintenance  Privilégier abstraction | Simplicité  Eviter complexité car impact sur maintenance | Single Responsibility : une responsabilité / classe  Open/Closed : permettre extension sans modifier  Liskov Substitution  Interface Segregation : interfaces spécifiques ≠ générales  Dependency Inversion : détails dépendent des absractions <>≠ | Singleton abuse  Tight coupling : A appelle direct méthode de B  Untestability  Premature optimization : Complexifier le code pour optimiser un cas rare  Indescriptible naming  Duplication |

Anti-patrons (odeurs de code)

|  |  |  |  |
| --- | --- | --- | --- |
| **ANTI-PATRONS** | **PROBLEME** | **SOLUTION** | **CONSÉQUENCES** |
| **Primitive Obsession** | Utiliser des types primitifs pour représenter des concepts complexes | Un concept = une classe | Code difficile a lire  Pas d’encapsulation  Erreurs faciles |
| **Innapropriate Intimacy** | Fuites de références  A accède à B et B à C, donc A accède à C | Eviter getters et setters auto  On devrait passer par l’interface pr modif les données | Fuite de références car une classe accède à un élément à partir d’une autre classe |
| **Switch Statement** | Utilisation de Switch / case() | Favoriser Strategy / template method |  |
| **Speculative Generality** | Prévoir des fonctions pour une utilisation future incertaine | Implémenter le nécessaire uniquement |  |
|  |  |  |  |
| **PATRONS :** | **‘’** | **‘’** | **‘’** |
| **Iterator** | Accéder à une collection en respectant encapsulation / masquage |  | Expose la collection en respectant l’encapsulation et le masquage |
| **Template Method** |  | Définir un algorithme générique dans une  classe de base et affiner certaines étapes dans les sous-classes. | Divers résultats pour une même méthode selon le type d’objet |
| **Strategy** | Eviter le surplus de conditions (Switch Statement) |  | Choix dynamiques à l’exécution |
| **Factory Method** |  |  | Le client n’a pas accès à l’objet directement mais à d’autres classes qui intéragissent avec celui-ci |
| **Null Object** | Aucune valeur |  |  |
| **Singleton** | Besoin d’unicité totale | Constructeur privé, méthode qui renvoie le singleton ou lance le constructeur si pas déjà existant | Instance unique d’une classe |
| **Flyweight** | Besoin d’unicité | « Singleton++ », variable statique privée (collection) | Permet N valeurs uniques (combinaisons de valeur) d’une classe (cartes dans un deck) |
| **State** | Representer l’etat d’un objet |  | Dynamique des comportements selon l’état |
| **Adapter** | Adapte 2 interfaces incompatibles | undefined |  |
| **Decorator** | Ajoute de nouvelles fonctionnalitées sans modification de la classe | Une image contenant texte, capture d’écran, diagramme, Police  Description générée automatiquement |  |
| **Prototype** | Crée une copie d’un objet (le clone) au lieu de faire New | La structure du patron de conception prototype dans l’exemple utilisé |  |
| **Observer** | Plusieurs objets réagissent automatiquement aux chang | Structure de l’exemple utilisé pour le patron de conception Observateur |  |
| **Composite** |  |  |  |
| **Visiteur** | séparer un algo de la structure des objets sur lesquels il opère | Visitor Pattern in C# - 5 Versions- CodeProject |  |
| **Abstract Factory** | Crée des familles d’objets liés sans spécifier leur classes concrètes |  |  |

**Diagrammes :**

|  |  |
| --- | --- |
| **Diagramme de classe (au dessus)** |  |
| **Diagramme de séquence** | Une image contenant texte, capture d’écran, ligne, diagramme  Description générée automatiquement |
| **Diagramme d’objets** | Une image contenant texte, ligne, capture d’écran, Police  Description générée automatiquement |
| **Diagramme d’état transition** |  |

Il faut tjr redéfinir les méthodes compareTo(), equals() et hashCode dans nos classes.

Immutabilité des données souvent souhaitable. On veut éviter de modif les données sans passer par l’interface.

Eviter la fuite de données sauf si immutables.

Exposer données, pas implémentation -> retours de copies ou wrappers d’immutabilité, voire copies profondes.

Contrats si possible (assert rank != null). Programmation par contrats (pré-, post-, invariant, etc …) (ex : @pre, @param)

Dépendre abstraction plutôt que implémentation (définition d’une interface -> couplage faible, vérification typage, polymorphisme).

Périmètre d’interface :

* Attention au périmètre lors de la définition des interfaces et à la cohérence
* Principe ISP de SOLID
* Dépendre uniquement des services dont on a besoin
* Séparation des préoccupations

État exprimable (représentable par ces données) vs

atteignable (peut obtenir par appel d’opérations à partir de l’état initial vs

souhaitable (souhaite atteindre).

Etat concret (j’ai 10 pts et mon adversaire en a 9) vs abstrait (j’ai gagné et il a perdu).

Option[T] ==> soit la variable ne contient rien, soit T

vavr.io: Either[E,T] ==> soit la variable contient E, soit T par exemple Either[String, T]

Try[T] = Either[Exception, T], Try permet de remplacer la levée d'une exception.