Du code SOLID

SOLID

- **S**ingle responsability principle
- **O**pen/closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency injection

Wikipedia

Single Responsability Principle

> *A class should have only one reason to change*
>
> *— Robert C. Martin, Clean Code*

Pourquoi ?

- Modularité
- Lisibilité
- Evolutivité
- Testabilité

Identifier les responsabilités

Pour quelles raisons ce code pourrait changer ?

Bien ou pas bien ?

- Allons voir un exemple : ArticleService

Bien ou pas bien ?

- le chemin vers le fichier de configuration peut changer
- le type de fichier de configuration peut changer (ie: yaml)
- la source de configuration peut évoluer (ie: environnement)
- la requête SQL peut changer
- le type de stockage peut changer (ie: API externe)
- l'algorithme de récupération peut changer (ie: status)

On refactore tout ça !

- isoler la responsabilité de la configuration
- isoler la responsabilité du stockage
- garder la responsabilité du code métier

Les limites du SRP

- Trop de fragmentation avec de très petites classes
- Augmentation de la complexité
- Découplage trop tôt

Récapitulatif

On analyse le code :

- Est-ce que le code peut avoir plusieurs raisons de changer ?

On arbitre :

- Est-ce que c’est grave ?
- Est-ce que c’est coûteux aujourd’hui ? demain ?

Dependency Injection

Pourquoi ?

- Mécanisme d’inversion de contrôle
- Réduire le couplage
- Facilite la réutilisation
- Simplifie la mise en place de tests unitaires

Comment faire ?

- Par le constructeur
- Par un paramètre de méthode
- Si on ne peut pas faire autrement :
  - Par un setter
  - Par manipulation du code dynamiquement (*introspection*)

Cas pratique

- On reprend l'exemple refactoré : ArticleService
- on passe en paramètre les variables d'instance

Limites de la DI

- Perte de lisibilité/traçabilité
- Couplage caché

Surtout lorsqu'on a une injection *magique* fournie par le framework

Récapitulatif

On analyse le code :

- Quelles sont mes dépendances ?

On arbitre :

- Qu’est-ce que je gagne à les injecter ?

Open / Closed

> *Les entités logicielles doivent être **ouvertes à l'extension**, mais **fermées à la modification**.*
>
> *— Bertrand Meyer*

Ouvert à l’extension

- On peut ajouter un nouveau comportement

Fermé à la modification

- Mais le code déjà présent ne devrait pas changer

Pourquoi ?

- Respect du *contrat d'interface*
- Pas de surprise à l'exécution

Bien ou pas bien ?

```
public class DiscountCalculator {

    public double calculateDiscount(String customerType, double amount) {
        if ("REGULAR".equals(customerType)) {
            return amount * 0.05;
        } else if ("PREMIUM".equals(customerType)) {
            return amount * 0.10;
        } else if ("VIP".equals(customerType)) {
            return amount * 0.20;
        }
        return 0.0;
    }
}
```

On refactore tout ça

```
public class DiscountCalculator {
    private final List<DiscountPolicy> policies;

    public double calculateDiscount(String customerType, double amount) {
        return findByCustomerType(customerType)
            .map(policy -> policy.applyDiscount(amount))
            .orElse(0.0);
    }

    private Optional<DiscountPolicy> findByCustomerType(String customerType) {
        return policies.stream()
            .filter(policy -> policy.appliesTo(customerType))
            .findFirst();
    }
}
```

On refactore tout ça

```
public class RegularDiscount implements DiscountPolicy {

    @Override
    public boolean appliesTo(String customerType) {
        return "REGULAR".equals(customerType);
    }

    @Override
    public double applyDiscount(double amount) {
        return amount * 0.05;
    }
}
```

Techniques

- Héritage ou composition pour déléguer
- Abstractions (interfaces, classes abstraites, design patterns)

Les limites

- Overengineering
- Fragmentation du code métier

⇒ Cibler le code qui change souvent

Récapitulatif

On analyse le code :

- Est-ce que c'est normal de modifier cette classe pour ma fonctionnalité ?

On arbitre :

- Est-ce que c'est grave ?
- Est-ce que c'est coûteux aujourd'hui ? demain ?

Substitution de Liskov

> *Les objets d'une classe dérivée doivent pouvoir être remplacés par des objets de la classe de base sans altérer la correction du programme.*

Exemple : une méthode qui utilise List doit pouvoir fonctionner avec ArrayList, LinkedList ou toute autre implémentation de List

Pourquoi ?

- Eviter les surprises
- Respect des contrats d’interface

Bien ou pas bien ?

```java
public class Character {
    public void move() { System.out.println("Character moves"); }
    public void attack() { System.out.println("Character attacks!"); }
}

public class Merchant extends Character {
    @Override
    public void attack() {
        throw new UnsupportedOperationException("Merchants do not attack!");
    }
}
```

```java
public void fight(Character ...characters) { /* appel de attack() */ }
fight(new Character(), new Merchant());
// ❌ exception inattendue lors de l'exécution
```

On refactore tout ça !

⇒ On extrait la partie problématique dans une interface

```
public interface Combatant {
    void attack();
}

public abstract class Character {
    public void move() { System.out.println("Character moves"); }
}
```

On refactore tout ça !

⇒ On utilise l'interface uniquement là où c'est pertinent

```
public class Player extends Character implements Combatant {
    @Override
    public void attack() { System.out.println("Warrior swings sword!"); }
}

public class Merchant extends Character {
    public void trade() { System.out.println("Merchant opens shop..."); }
}
```

```
public void fight(Combatatant ...combatants) { /* appel de attack() */ }
fight(new Character(), new Merchant());
// 🚨 ne compile plus, Merchant != Combatant
```

Récapitulatif

On analyse le code :

- Est-ce que ça pose problème si je remplace une classe/interface par son implémentation ?
- J’ai une "UnsupportedOperation", est-ce que c’est légitime ?

On arbitre :

- Est-ce que c’est grave ?
- Est-ce que c’est coûteux aujourd’hui ? demain ?

Ségrégation d'interface

> *Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.*

Pourquoi

- Eviter de la complexité inutile
- Meilleure modularité

Symptôme

- Implémentation vide ou qui retourne un "not supported"

Bien ou pas bien ?

```
public interface Quest {
    void start();
    void complete();
    void trackProgress();
    void giveReward();
}

public class CinematicQuest implements Quest {
    public void start() { System.out.println("Début d'une cinématique."); }
    public void complete() { System.out.println("Fin de la cinématique."); }
    // La progression ne s'affiche jamais à l'utilisateur, mais est tout de même
    public void trackProgress() { System.out.println("Progression interne (non vi
    // Pas de récompense, mais méthode appelée pour rien
    public void giveReward() { System.out.println("Pas de récompense prévue."); }
}
```

On refactore tout ça

⇒ Séparation des interfaces

```
public interface Quest {
    void start();
    void complete();
}

public interface ProgressTrackable {
    void trackProgress();
}

public interface Rewardable {
    void giveReward();
}
```

On refactore tout ça

⇒ Utilisation à la demande

```
public class SimpleFetchQuest implements Quest, ProgressTrackable, Rewardable {
    public void start() { System.out.println("Va chercher 10 pommes."); }
    public void complete() { System.out.println("Tu as apporté 10 pommes."); }
    public void trackProgress() { System.out.println("Pommes récupérées : 7/10");
    public void giveReward() { System.out.println("Tu reçois 100 pièces."); }
}

public class CinematicQuest implements Quest {
    public void start() { System.out.println("Début de la cinématique."); }
    public void complete() { System.out.println("Fin de la cinématique."); }
}
```

Récapitulatif

On analyse le code :

- Qu’est-ce qui peut changer ?

On arbitre :

- Est-ce que c’est grave ?
- Est-ce que c’est coûteux aujourd’hui ? demain ?

Merci de votre attention