
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 ?

Exemple

- Allons voir un exemple : ArticleService

Problèmes

- 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

- 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

Open / Closed

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

Ouvert à l’extension

- On doit pouvoir ajouter un nouveau comportement à un programme

Fermé à la modification

- Mais sans avoir à modifier le fonctionnement existant

Analogie

J’ai un robot de cuisine, il permet de battre des œufs en neige

Il est fermé à la modification

Je ne veux pas le démonter à chaque fois que je veux lui ajouter une nouvelle fonctionnalité, comme pétrir de la pâte par exemple

Analogie

En revanche, **il est ouvert à l'extension**, je peux ajouter de nouvelles fonctionnalités grâce à des accessoires et/ou des réglages

Si je veux pétrir de la pâte :

- je change le fouet par un crochet
- et je réduis la vitesse

Pourquoi ?

- On ne touche pas aux comportements existants
- On rend le code plus modulaire
- On rend le code plus extensible

Exemple

```
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;
    }
}
```

Problème

- Pour ajouter un nouveau type de ristourne on doit **modifier** le fonctionnement existant

On refactore, étape 1

- On peut extraire la partie répétée

```
public class DiscountCalculator {

    Map<String, Function<Double,Double>> discountPerCustomerType = Map.of(
        "REGULAR", amount -> amount * 0.05,
        "PREMIUM", amount -> amount * 0.10,
        "VIP", amount -> amount * 0.20
    );

    public double calculateDiscount(String customerType, double amount) {
        if(discountPerCustomerType.hasKey(customerType)) {
            return discountPerCustomerType.get(customerType).apply(amount);
        }

        return 0.0;
    }
}
```

On refactore, étape 2

- On passe les discounts dans une interface

```
public interface Discount {
    boolean appliesTo(String customerType);
    double applyDiscount(double amount);
}
```

On refactore, étape 2

```
public class RegularDiscount implements Discount {

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

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

On refactore, étape 2

- Version finale

```
public class DiscountCalculator {
    private final List<Discount> discounts;

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

    private Optional<Discount> findDiscountByCustomerType(String customerType) {
        return discounts.stream()
            .filter(discount -> discount.appliesTo(customerType))
            .findFirst();
    }
}
```

Techniques

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

Les limites

- Overengineering
- Fragmentation du code métier
- Abstraction prématurée

⇒ Cibler le code qui change souvent

Liskov substitutions

> *Les objets d'une classe dérivée doivent pouvoir remplacer ceux de la classe parente sans altérer la cohérence du programme.*

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

> *Une sous-classe ne doit pas renforcer les préconditions ni affaiblir les postconditions de la classe mère.*

Ne pas être plus dur que le contrat de la classe mère

Pourquoi ?

- Eviter les surprises
- Respect des contrats d’interface

Exemple

```
public interface Character {
    void move(Position newPosition);
    void attack(Character opponent);
    void trade(Character other);
}

public class Player extends Character { /* Tout est implémenté */ }

public class NPC extends Character { /* Un PNJ ne peux pas attaquer ! */
    @Override
    public void attack(Character opponent) {
        throw new UnsupportedOperationException("Merchants do not attack!");
    }
}
```

Le problème

Pour respecter Liskov, je dois pouvoir transformer cette fonction :

```
public class FightService {
    public void fight(Character a, Character b) {
        a.attack(b);
    }
}
```

en :

```
public class FightService {
    public void fight(Merchant a, Merchant b) {
        a.attack(b);
        // ❌ exception inattendue lors de l'exécution
    }
}
```

On refactore

⇒ On extrait la partie problématique de l'interface

```
public interface Character {
    void move(Position newPosition);
    void trade(Character other);
}
```

On refactore

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

public class Merchant extends Character {
    /* Plus d'implémentation inutile */
}
```

```
public class FightService {
    public void fight(Player a, Player b) {
        a.attack(b);
    }
}
```

Interface segregation

> *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"

Exemple

```
public interface Character {
    void move(Position newPosition);
    void attack(Character opponent);
    void trade(Character other, Item item, Price price);
}
```

Le problème

```
public class NPC implements Character {
    /* Un PNJ ne peut pas attaquer ! */
    @Override
    public void attack(Character opponent) {
        throw new UnsupportedOperationException("Merchants do not attack!");
    }
}

public class Monster implements Character {
    /* Un monstre ne peut pas commercer */
    @Override
    public void trade(Character other, Item item, Price price) {
        throw new UnsupportedOperationException("Monsters don't trade");
    }
}
```

On refactore tout ça

⇒ Séparation des interfaces

```
public interface Character {
    void move(Position newPosition);
}

public interface Combattant {
    void attack(Combattant other);
}

public interface Trader {
    void trade(Trader other, Item item, Price price);
}
```

On refactore tout ça

⇒ Utilisation à la demande

```
public class Player implements Character, Combattant, Trader {
    /* ... */
}
```

```
public class NPC implements Character, Trader {
    /* ... */
}
```

```
public class Monster implements Character, Combattant {
    /* ... */
}
```

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

Merci de votre attention