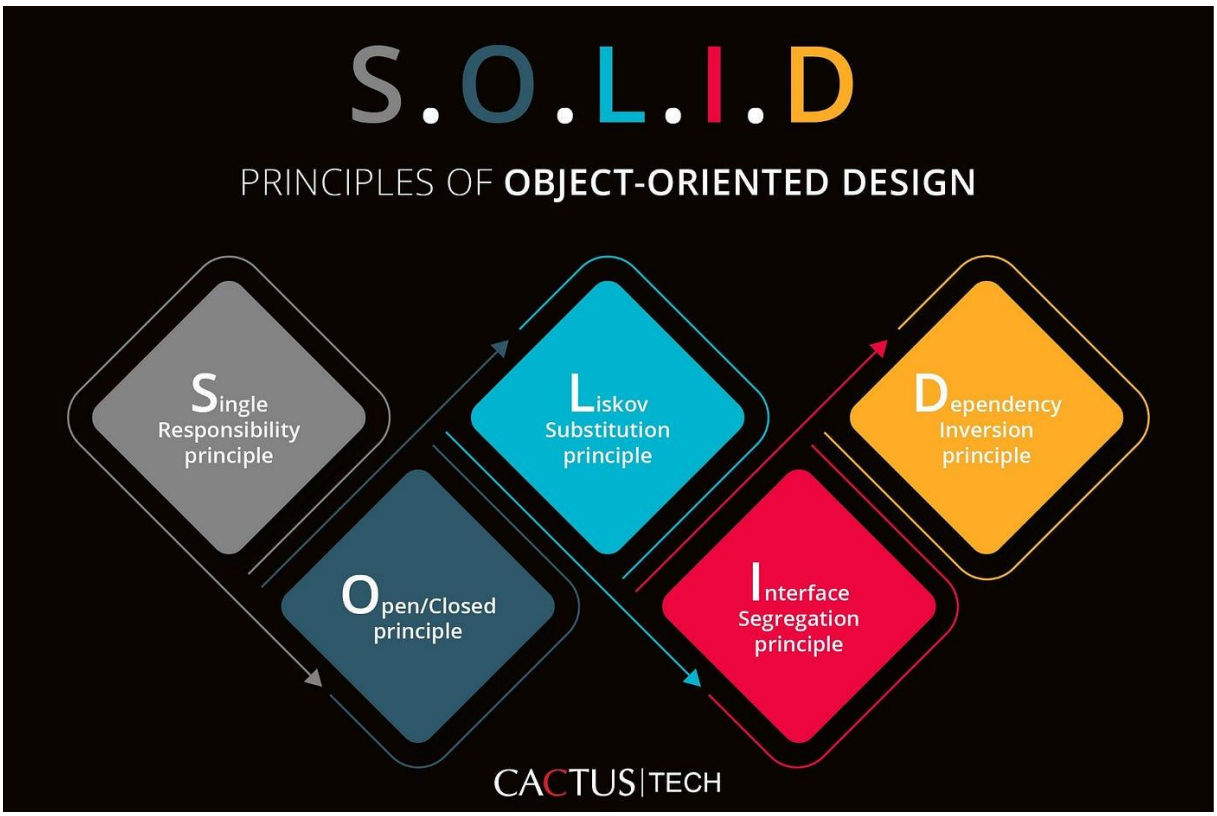
Lecture Individuelle 4

SOLID



[1]

[1] https://miro.medium.com/v2/resize:fit:1400/1*-j-WDZlgOWP1EjvcMSL13w.jpeg

Table des matières

Introduction

Les principes SOLID, élaborés par Robert C. Martin dans son essai influent intitulé "Design Principles and Design Patterns" en 2000, représentent un jalon dans le paysage en constante évolution du développement logiciel. Bien que l'acronyme SOLID ait été forgé plus tard par Michael Feathers, ces principes ont joué un rôle central dans la façon dont les ingénieurs abordent la conception logicielle.

Dans son essai perspicace, Martin a reconnu une vérité fondamentale sur le logiciel : il évolue et subit des changements. Cependant, sans une base solide en matière de principes de conception, cette évolution peut conduire à une complexité mettant en péril l'essence même d'un logiciel réussi. Martin a mis en garde contre les pièges potentiels tels que la rigidité, la fragilité, l'immobilité, voire une certaine malveillance dans le code.

Les principes SOLID ont émergé en réponse à ces défis, fournissant un ensemble de lignes directrices pour combattre les modèles de conception problématiques et favoriser l'adaptabilité face au changement.

Fondamentalement, l'objectif global des principes SOLID est de réduire les dépendances, permettant ainsi aux ingénieurs de modifier une partie du code logiciel sans affecter les autres. De plus, ils visent à rendre les conceptions plus faciles à comprendre, à maintenir et à étendre. En fin de compte, l'utilisation de ces principes de conception facilite la prévention des problèmes et la construction d'un logiciel adaptable, efficace et agile.

1. Single Responsibility Principle (SRP)

1.1. Définition

Il s'agit du premier principe de SOLID, le principe de responsabilité unique.

Ce principe stipule qu'une classe ne devrait avoir qu'une seule raison de changer, impliquant ainsi que chaque classe ne doit avoir qu'une seule tâche et qu'un seul objectif.

Si nous prenons l'exemple du développement logicielle, nous pouvons voir que normalement, chacun à un rôle bien spécifique, car la tâche est divisée entre plusieurs personnes qui font chacune, des choses différentes (frontend, backend…).

La plupart des développeurs ont tendance à implémenter des nouvelles fonctionnalités ou comportements dans une classe existante, ce qui est incorrect, car cela va rendre notre code plus long et plus complexe.

Il est important de diviser les classes volumineuses en classes plus petites dans notre application.

1.2. Avantages

Il y a plusieurs avantages à mettre en place ce principe dans nos applications :

- Testabilité : une classe avec une seule responsabilité, aura besoin de beaucoup moins de cas de test.
- Couplage réduit : moins de fonctionnalités dans une seule classe signifie moins de dépendances à gérer.
- Organisation : les classes plus petites et bien organisées sont plus faciles à chercher.

1.3. Exemples d'utilisation

Voici un exemple d'une simple classe représentant un livre :

```
public class Book {

   private String name;
   private String author;
   private String text;

   // constructeur, getters et setters

   // méthodes liées directement aux propriétés du livre
   public String replaceWordInText(String word, String replacementWord){
      return text.replaceAll(word, replacementWord);
   }

   public boolean isWordInText(String word){
      return text.contains(word);
   }
}
```

Imaginons que maintenant, nous ajoutons une méthode qui imprime des informations sur le livre, le faire dans la même, violerai le principe de responsabilité unique.

Pour respecter ce principe, il faut créer une classe distincte qui s'occupe uniquement de l'impression de nos textes :

```
public class BookPrinter {

   // méthodes pour afficher du texte
   void printTextToConsole(String text){
      // notre code pour formater et imprimer le texte
   }

   void printTextToAnotherMedium(String text){
      // code pour écrire vers n'importe quel autre emplacement...
   }
}
```

Ce code permet non seulement de décharger la classe livre, mais également d'exploiter cette classe pour envoyer notre texte vers d'autres supports.

2. Open/Closed Principle (OCP)

2.1. Définition

Le principe d'ouverture/fermeture est que les classes existantes, bien testées, n'auront pas besoin d'être modifiées lorsque nous voulons ajouter des améliorations.

Imaginons que nous voulons ajouter quelque chose à notre classe fonctionnel, nous allons prendre le risque d'avoir des problèmes ou des bugs, ce qui entraînera davantage de travail.

La solution est donc d'étendre la classe.

Le respect de ce principe est essentiel afin d'écrire un code facile à entretenir et à examiner. Nous respectons ce principe, si ces deux caractéristiques sont remplies :

- Le code est ouvert à l'extension
- Le code est fermé à la modification

2.2. Avantages

Il y a plusieurs avantages à mettre en place ce principe dans nos applications :

- Flexibilité : ce principe permet d'étendre ou de modifier le comportement d'un système sans changer son code source. Cela rend le développement du code plus flexible et adaptable aux exigences changeantes.
- Réutilisabilité : en séparant le comportement de l'implémentation, nous pouvons réutiliser facilement le code logiciel dans d'autres parties de l'application.
- Maintenabilité : le fait d'avoir un code fermé à la modification, permet que les modifications apportées à une partie du logiciel n'affectent pas les autres parties, réduisant ainsi la probabilité de bugs.

2.3. Exemples d'utilisation

Voici un exemple d'un code respectant ce principe :

```
abstract class GameCharacter {
        abstract void attack() {}

        abstract void jump() {}

}

Class Mario extends GameCharacter {
        void attack() {
                //logique pour une attaque de Mario
        }
        void jump() {
                //logique pour un saut de Mario
        }
}
```

```
class Luigi extends GameCharacter {
	void attack() {
		//logique pour une attaque de Luigi
	}
	void jump() {
		//logique pour un saut de Luigi
	}
}
```

Dans ce code, nous pouvons facilement ajouter un nouveau personnage au jeu, avec des modifications minime au code existant.

3. Liskov Substitution Principle (LSP)

3.1. Définition

Ce principe, proposé par Barbara Liskon en 1987, dit que « les classes dérivées ou enfants doivent être substituables à leurs de classes de base ou parentes ». Ce principe garantit qu'une classe enfant d'une classe parent peut être utilisée à la place de la classe parent sans causer de comportement inattendu.

On peut comprendre ce principe de la manière suivante : le fils d'un fermier devrait héritier des compétences en agriculture de son père et devrait remplacer son père si nécessaire. Si le fils veut devenir fermier, il peut remplacer son père, cependant, si il veut devenir joueur de cricket, alors le fils ne peut certainement pas remplacer son père.

3.2. Avantages

Il existe plusieurs avantages à l'utilisation de ce principe:

- Augmentation de la réutilisabilité de notre code : en suivant ce principe, nous pouvons créer un ensemble de classes connexes qui peuvent être utilisées de manière interchangeable sans modification du code.
- Simplification de la maintenance du code : en suivant ce principe, nous pouvons apporter des modifications à une classe sans affecter le comportement des autres classes du groupe.

3.3. Exemples pratiques

Voici un exemple d'un code respectant ce principe :

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  makeSound() {
    console.log(`${this.name} makes a sound`);
  }
}

class Dog extends Animal {
  makeSound() {
    console.log(`${this.name} barks`);
  }
}

class Cat extends Animal {
  makeSound() {
    console.log(`${this.name} meows`);
  }
}

function makeAnimalSound(animal) {
  animal.makeSound();
}

const cheetah = new Animal('Cheetah');
makeAnimalSound(cheetah); // Cheetah makes a sound

const dog = new Dog('Jack');
makeAnimalSound(dog); // Jack barks

const cat = new Cat('Khloe');
makeAnimalSound(cat); // Khloe meows
```

4. Interface Segregation Principle (ISP)

4.1. Définition

Ce principe est le premier principe de SOLID qui s'applique aux interfaces, plutôt que aux classes. Il est similaire au principe de responsabilité unique.

Il stipule que nous devons pas forcer un client à implémenter une interface qui ne lui est pertinente. L'objectif principal est d'éviter les interfaces surchargées et de privilégier de nombreuses petites interfaces spécifiques au client. Chaque interface devrait avoir une responsabilité spécifique.

Prenons comme exemple un restaurant. Imaginons que vous allez dans un restaurant et que vous êtes strictement végétarien. Le serveur de ce restaurant vous remet le menu, comprenant des plats végétariens, des plats non végétariens, des boissons et des desserts. Dans ce cas, en tant que client, vous devriez avoir un menu ne comprenant que des plats végétariens, et non tout ce que vous ne mangez pas. Ici, le menu devrait être différent pour différents types de clients. Le menu commun ou général pour tout le monde peut être divisé en plusieurs menus au lieu d'un seul. L'utilisation de ce principe aide à réduire les effets secondaires et la fréquence des modifications nécessaires.

4.2. Avantages

Il existe plusieurs avantages à l'utilisation de ce principe :

- Il rend le code plus modulable et réutilisable.
- Il réduit la complexité du code et le rend plus facile à comprendre et utiliser.

4.3. Exemple pratique

Voici un exemple pratique de ce principe :

Imaginons que nous travaillons dans un zoo, et que nous nous occupons d'un ours :

```
public interface BearKeeper {
   void washTheBear();
   void feedTheBear();
   void petTheBear();
}
```

En effet, tout les rôles ne sont pas attribués que à une seul personne, il faut donc séparer nos interfaces :

```
public interface BearCleaner {
   void washTheBear();
}

public interface BearFeeder {
   void feedTheBear();
}

public interface BearPetter {
   void petTheBear();
}
```

Ainsi, avec la division d'interfaces, nous pouvons implémenter que les méthodes que l'on désire :

```
public class BearCarer implements BearCleaner, BearFeeder {

   public void washTheBear() {

   }

   public void feedTheBear() {

   }
}
```

5. Dependency Inversion Principle (DIP)

5.1. Définition

Ce principe fonctionne sur deux principes :

- Les modules/classes de haut niveau ne devraient pas dépendre des modules/classes de bas niveau. Les deux devraient dépendre d'abstractions.
- Les abstractions ne devraient pas dépendre des détails. Les détails devraient dépendre des abstractions.

Ces phrases expliquent que si un module ou une classe de haut niveau dépend davantage de modules ou de classes de bas niveau, votre code sera fortement dépendant.

Si vous modifiez une classe, une autre pourrait être affectée, ce qui est risqué à un niveau de production. Il est donc important de toujours rendre les classes aussi peu connectées que possible, ce que vous pouvez réaliser grâce à l'abstraction.

L'objectif fondamental de cette approche est de découpler les dépendances de telle sorte que si la classe A changé, la classe B n'ait pas à être concernée ni consciente des changements.

Prenons comme exemple une pile de télécommande pour la télévision. Votre télécommande nécessite une pile, mais la marque n'a pas d'importance. Vous pouvez utiliser n'importe quelle pile de marque XYZ, et cela fonctionnera toujours. Par conséquent, nous pouvons dire que la télécommande de la télévision est seulement marginalement liée au nom de la marque. L'inversion de dépendance améliore la réutilisabilité de votre code.

5.2. Avantages

Ce principe nous permet d'avoir un code beaucoup plus facile à modifier, et dont nous pouvons ajouter des fonctionnalités sans crainte de bug.

Ce principe permet aussi d'améliorer la lisibilité et la qualité du code.

5.3. Exemple pratique

Voici un exemple d'un code respectant ce principe :

```php
<?php
/**
 * Dependency Inversion Principle (DIP) in PHP
 */

interface PaymentInterface
{
   public function pay(int $amount): void;
}

class PayPal implements PaymentInterface
{
	public function pay(int $amount): void
	{
		echo "Discussing with PayPal...\n";
	}
}

class Stripe implements PaymentInterface
{
	public function pay(int $amount): void
	{
		echo "Discussing with Stripe...\n";
	}
}

class AliPay implements PaymentInterface
{
	public function pay(int $amount): void
	{
		echo "Discussing with AliPay...\n";
	}
}

// So many providers exist...

class PaymentProvider
{
	public function goToPaymentPage( PaymentInterface $paymentChoosen, int $amount ): void
	{
		$paymentChoosen->pay($amount);
	}
}

$paymentProvider = new PaymentProvider();
$paymentProvider->goToPaymentPage(new PayPal(), 100);
$paymentProvider->goToPaymentPage(new Stripe(), 100);
$paymentProvider->goToPaymentPage(new AliPay(), 100);
```

Bibliographie

Alex (2023). A Solid Guide to SOLID Principles

Consulté le 20 décembre 2023, à l'adresse Principes SOLID : Le guide (avec 5 exemples) – Alex so yes

Kolade Chris (2023). SOLID Design Principles in Sofware Development

Consulté le 20 décembre 2023, à l'adresse SOLID Design Principles in Software Development (freecodecamp.org)

Mahendra Chouhan (2023). Solid Principle Part 2: Open Closed Principle (OCP)

Consulté le 20 décembre 2023, à l'adresse https://www.enjoyalgorithms.com/blog/open-close-principle

Mahendra Chouhan (2023). Solid Principle Part 4: Interface Segregation Principle (ISP)

Consulté le 20 décembre 2023, à l'adresse Interface Segregation Principle (ISP) (enjoyalgorithms.com)

Sam Millington (2023). A Solid Guide to SOLID Principles

Consulté le 20 décembre 2023, à l'adresse A Solid Guide to SOLID Principles | Baeldung