Principes de Conception Logicielle (Design Principles) & Patrons de Conception (Design Patterns)

SOLID Principles

Voici la fiche sur les **Principes de Conception Logicielle (Design Principles)**, suivie des **Patrons de Conception (Design Patterns)** :

---

1. Single Responsibility Principle (SRP)

Définition : Chaque classe doit avoir une seule responsabilité. Cela signifie qu'une classe ne devrait avoir qu'une seule raison de changer.

Exemples de Code

JavaScript :

```
// Mauvaise conception : une classe gère plusieurs responsabilités
class User {
    constructor(name, email) {
        this.name = name;
        this.email = email;
    }

    saveToDatabase() {
        // Logique pour sauvegarder l'utilisateur dans la base de données
    }

    sendEmail() {
        // Logique pour envoyer un e-mail à l'utilisateur
    }
}

// Bonne conception : chaque classe a une responsabilité
class User {
    constructor(name, email) {
        this.name = name;
        this.email = email;
    }
}

class UserRepository {
    save(user) {
        // Logique pour sauvegarder l'utilisateur dans la base de données
    }
}

class EmailService {
```

```
    sendEmail(user) {
        // Logique pour envoyer un e-mail à l'utilisateur
    }
}
```

Java :

```
// Mauvaise conception : une classe gère plusieurs responsabilités
class User {
    String name;
    String email;

    void saveToDatabase() {
        // Logique pour sauvegarder l'utilisateur dans la base de données
    }

    void sendEmail() {
        // Logique pour envoyer un e-mail à l'utilisateur
    }
}

// Bonne conception : chaque classe a une responsabilité
class User {
    String name;
    String email;
}

class UserRepository {
    void save(User user) {
        // Logique pour sauvegarder l'utilisateur dans la base de données
    }
}

class EmailService {
    void sendEmail(User user) {
        // Logique pour envoyer un e-mail à l'utilisateur
    }
}
```

2. Open/Closed Principle (OCP)

Définition : Les entités logicielles doivent être ouvertes à l'extension mais fermées à la modification. Cela signifie qu'il doit être possible d'ajouter de nouvelles fonctionnalités sans modifier le code existant.

Exemples de Code

JavaScript :

```
// Mauvaise conception : nécessite des modifications pour chaque type de rapport
class Report {
    constructor(type) {
        this.type = type;
    }

    generate() {
        if (this.type === 'PDF') {
            // Logique pour générer un rapport PDF
        } else if (this.type === 'Excel') {
            // Logique pour générer un rapport Excel
        }
    }
}

// Bonne conception : ajoute de nouveaux types de rapports sans modifier la classe existante
class Report {
    generate() {
        // Logique pour générer un rapport
    }
}

class PDFReport extends Report {
    generate() {
        // Logique pour générer un rapport PDF
    }
}

class ExcelReport extends Report {
    generate() {
        // Logique pour générer un rapport Excel
    }
}
```

Java :

```
// Mauvaise conception : nécessite des modifications pour chaque type de rapport
class Report {
    String type;

    void generate() {
        if (type.equals("PDF")) {
            // Logique pour générer un rapport PDF
        } else if (type.equals("Excel")) {
            // Logique pour générer un rapport Excel
        }
    }
}
```

```
// Bonne conception : ajoute de nouveaux types de rapports sans modifier la classe existante
abstract class Report {
    abstract void generate();
}

class PDFReport extends Report {
    void generate() {
        // Logique pour générer un rapport PDF
    }
}

class ExcelReport extends Report {
    void generate() {
        // Logique pour générer un rapport Excel
    }
}
```

3. Liskov Substitution Principle (LSP)

Définition : Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe de base sans altérer le comportement du programme.

Exemples de Code

JavaScript :

```
class Bird {
    fly() {
        console.log("I can fly");
    }
}

class Duck extends Bird {
    quack() {
        console.log("Quack");
    }
}

// Mauvaise conception : la classe penguin ne peut pas voler
class Penguin extends Bird {
    fly() {
        throw new Error("Penguins can't fly");
    }
}

// Bonne conception : les classes respectent le LSP
class Sparrow extends Bird {
    fly() {
        console.log("I can fly");
```

```
    }
}
```

Java :

```
class Bird {
    void fly() {
        System.out.println("I can fly");
    }
}

class Duck extends Bird {
    void quack() {
        System.out.println("Quack");
    }
}

// Mauvaise conception : la classe Penguin ne peut pas voler
class Penguin extends Bird {
    void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}

// Bonne conception : les classes respectent le LSP
class Sparrow extends Bird {
    void fly() {
        System.out.println("I can fly");
    }
}
```

4. Interface Segregation Principle (ISP)

Définition : Les clients ne devraient pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas. Il vaut mieux avoir plusieurs interfaces spécifiques que de créer une seule interface générale.

Exemples de Code

JavaScript :

```
// Mauvaise conception : une interface générale
class Machine {
    print() {}
    scan() {}
    fax() {}
}

// Bonne conception : interfaces spécifiques
class Printer {
```

```
    print() {
        // Logique pour imprimer
    }
}

class Scanner {
    scan() {
        // Logique pour scanner
    }
}
```

Java :

```
// Mauvaise conception : une interface générale
interface Machine {
    void print();
    void scan();
    void fax();
}

// Bonne conception : interfaces spécifiques
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}
```

5. Dependency Inversion Principle (DIP)

Définition : Les modules de haut niveau ne devraient pas dépendre des modules de bas niveau, mais des abstractions. Cela réduit le couplage entre les modules.

Exemples de Code

JavaScript :

```
// Mauvaise conception : dépendance directe
class UserService {
    constructor() {
        this.database = new Database(); // Couplage fort
    }
}

// Bonne conception : dépendance par abstraction
class UserService {
    constructor(database) {
        this.database = database; // Couplage faible
```

```
    }
}
```

Java :

```java
// Mauvaise conception : dépendance directe
class UserService {
    private Database database;

    public UserService() {
        this.database = new Database(); // Couplage fort
    }
}

// Bonne conception : dépendance par abstraction
interface Database {
    void save(User user);
}

class UserService {
    private Database database;

    public UserService(Database database) {
        this.database = database; // Couplage faible
    }
}
```

Patrons de Conception (Design Patterns)

1. Créationnels

- **Singleton** : Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.
- **Factory** : Définit une interface pour créer un objet, mais laisse les sous-classes décider de la classe à instancier.
- **Abstract Factory** : Fournit une interface pour créer des familles d'objets sans spécifier leurs classes concrètes.
- **Builder** : Permet de créer des objets complexes étape par étape.
- **Prototype** : Permet de créer des objets en copiant un prototype existant.

2. Structuraux

- **Adapter** : Permet à des interfaces incompatibles de fonctionner ensemble.

- **Composite** : Permet de traiter des objets individuels et des compositions d'objets de manière uniforme.
- **Proxy** : Fournit un substitut ou un représentant d'un autre objet pour contrôler l'accès à celui-ci.
- **Decorator** : Ajoute des fonctionnalités à un objet dynamiquement sans modifier son interface.
- **Facade** : Fournit une interface simplifiée à un ensemble de classes ou sous-systèmes.

3. Comportementaux

- **Observer** : Définit une relation de dépendance entre objets, de sorte qu'un changement d'état d'un objet soit notifié à tous ses dépendants.
- **Strategy** : Définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables.
- **Command** : Encapsule une demande en tant qu'objet, permettant de paramétrer des clients avec des files d'attente ou des demandes.
- **State** : Permet à un objet de changer son comportement lorsqu'il change d'état.
- **Iterator** : Fournit un moyen d'accéder séquentiellement aux éléments d'un agrégat sans exposer sa représentation sous-jacente.
- **Mediator** : Définit un objet qui encapsule la manière dont un ensemble d'objets interagit.

Ces principes et patrons de conception sont essentiels pour écrire du code maintenable et extensible. Si tu veux approfondir un aspect particulier ou ajouter d'autres exemples, fais-le moi savoir !