3 ~ Structurer le code proprement (SOLID)

? En un mot

Solid est un outil pour maîtriser la complexité croissante.

? Ce que SOLID est ?

Il facilite:

- □□ Modularité
- □□ Testabilité
- □□ Extensibilité
- □□ Réduction de dette technique
- □□ Architecture durable
- □□ Travail en équipe

? Ce que SOLID n'est pas ?

- Une obligation dogmatique
- Un truc à appliquer partout
- Un truc adapté à tout (micro-scripts inutiles)

? Les 5 piliers de SOLID

? S - Single Responsibility Principle

Une classe ou un module ne doit avoir qu’une seule responsabilité métier.

? Mauvais

```
class UserService {
  createUser(data) { /* ... */ }
  sendWelcomeEmail(user) { /* ... */ }
  logUserCreation(user) { /* ... */ }
}
```

? Bon

```
class UserService {
  createUser(data) { /* ... */ }
}

class EmailService {
  sendWelcomeEmail(user) { /* ... */ }
}

class Logger {
  log(message) { /* ... */ }
}
```

? Pourquoi?

- Tests plus simple
- Evolution indépendante
- Moins d’effets de bord

? O - Open Closed Principle (OCP)

Ouvert à l’extension, fermé à la modification. On dois pouvoir ajouter un comportement sans modifier le code existant.

? Mauvais

```
function calculateDiscount(user) {
  if (user.type === "premium") return 0.2;
  if (user.type === "vip") return 0.3;
}
```

? Bon

```
class DiscountStrategy {
  getDiscount() {
    return 0;
  }
}

class PremiumDiscount extends DiscountStrategy {
  getDiscount() {
    return 0.2;
  }
}

class VipDiscount extends DiscountStrategy {
  getDiscount() {
    return 0.3;
  }
}
```

? Pourquoi?

- Scalabilité
- Architecture plugin
- Systèmes extensibles

? L - Liskov Substitution Principle (LSP)

Un sous-type doit pouvoir remplacer son type parent sans casser le comportement attendu. Si tu hérites, tu dois respecter le contrat.

? Mauvais

```
class Bird {
  fly() {}
}

class Penguin extends Bird {
  fly() {
    throw new Error("I can't fly");
  }
}
```

? Bon

```
class Bird {}

class FlyingBird extends Bird {
  fly() {}
}

class Penguin extends Bird {}
```

? Pourquoi?

- Hiérarchies cohérentes
- Pas de surprises

? I - Interface Segregation Principle (ISP)

Ne force pas une classe à implémenter ce qu’elle n’utilise pas.

? Mauvais

```
class Employee {
  work() {}
  eat() {}
  sleep() {}
}

class Robot extends Employee {
  work() {
    console.log("Working...");
  }

  eat() {
    throw new Error("Robot does not eat");
  }

  sleep() {
    throw new Error("Robot does not sleep");
  }
}
```

? Bon

```
class Workable {
  work() {}
}

class Human extends Workable {
  eat() {
    console.log("Eating...");
  }

  sleep() {
    console.log("Sleeping...");
  }
```

```
}

class Robot extends Workable {
  work() {
    console.log("Working...");
  }
}
```

? Pourquoi?

- Architecture plus flexible

? D - Dependency Inversion Principle (DIP)

Dépendre d’abstractions, pas de concrétions (implémentation concrète).

? Mauvais

```
class UserService {
  constructor() {
    this.database = new MySQLDatabase();
  }
}
```

? Bon

```
class UserService {
  constructor(database) {
    this.database = database;
  }
}

const db = new MySQLDatabase();
const service = new UserService(db);
```

? Pourquoi?

- Interchangeable (Ici, passer à Postgres, Mongo, une API)
- Scalable

Révision #8
Créé 2026-02-13 12:01:00 CET par Larananas
Mis à jour 2026-02-13 12:47:17 CET par Larananas