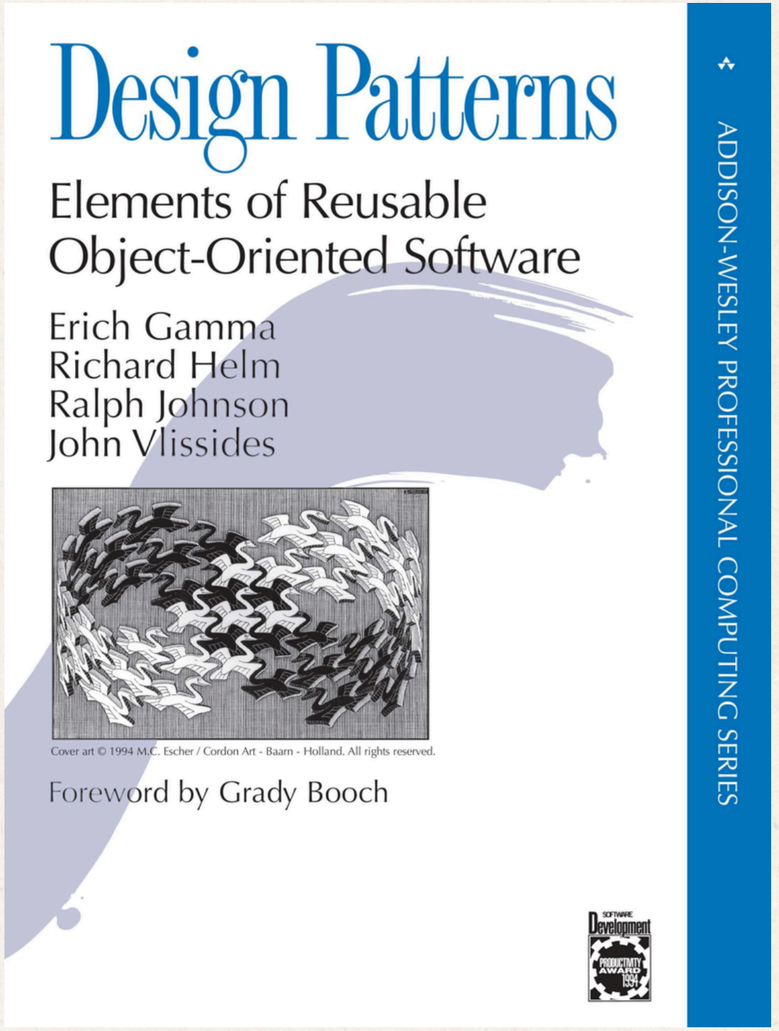
Pattern Etat

Présenté par :
Mathis GRANIER - Thurian SACRISTAN
Milan LOI - Arthur LABREGERE

Sommaire

Pattern Gof

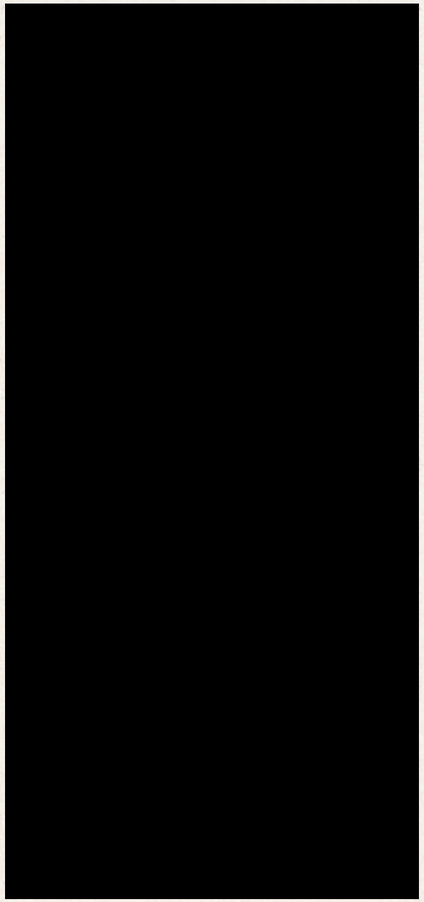
- Modèles conceptuels, pas du prêt à l’emploi
- 23 patterns
- 3 familles de patterns :
  - Création
  - Structurels
  - Comportementaux
- Une boite a outils



1994

Exemple de besoin

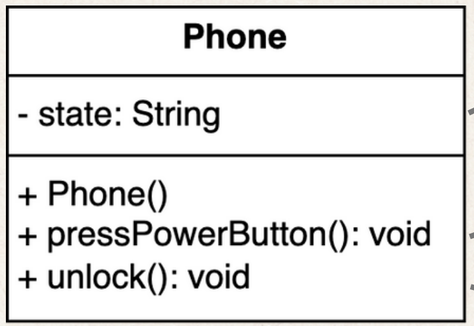
Off



Locked

Ready

Modélisation



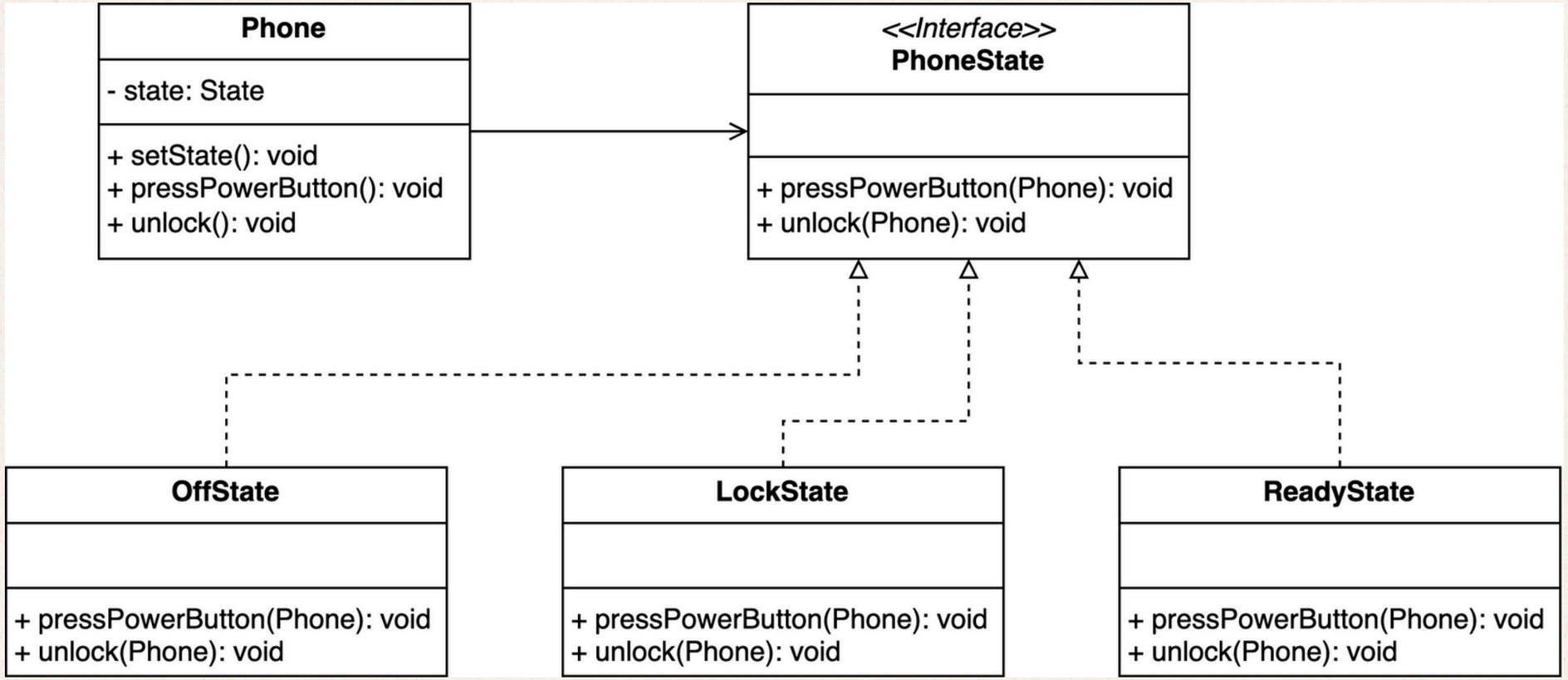
❌ Aucun découplage entre les comportements selon l'état.

❌ Ajouter un nouvel état nécessiterait de modifier le code du Phone.

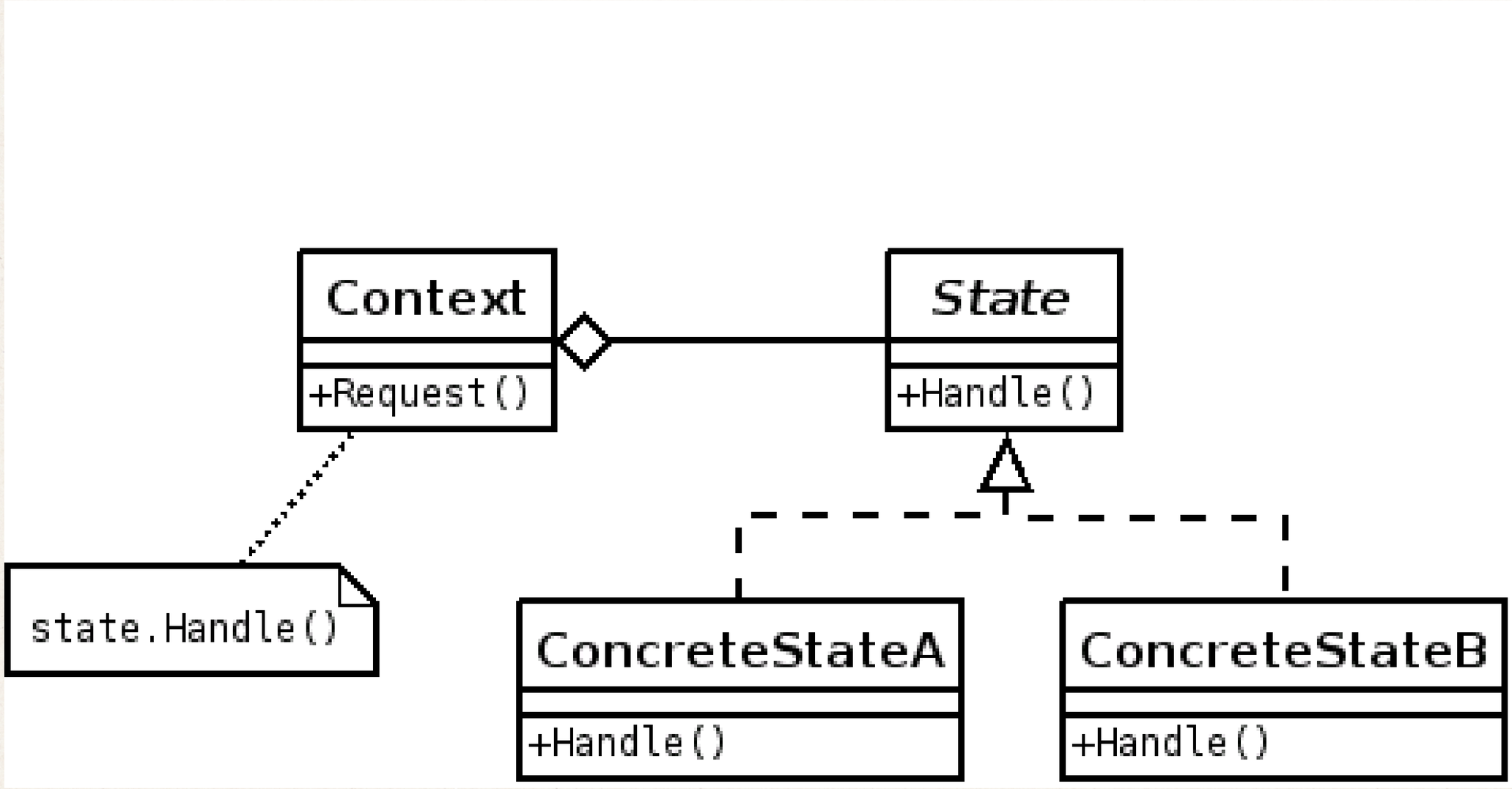
Phone.java

```java
public class Phone {  10 usages
    private String state;  7 usages

    public Phone() {  1 usage
        this.state = "off"; // Par défaut, le téléphone est éteint
    }

    public void pressPowerButton() {  3 usages
        switch (state) {
            case "off":
                System.out.println("Le téléphone s'allume...");
                state = "locked";
                break;
            case "locked":
                System.out.println("Le téléphone s'éteint...");
                state = "off";
                break;
            case "ready":
                System.out.println("Le téléphone se verrouille...");
                state = "locked";
                break;
        }
    }

    public void unlock() {  2 usages
        switch (state) {
            case "off":
                System.out.println("Impossible : le téléphone est éteint !");
                break;
            case "locked":
                System.out.println("Déverrouillage réussi !");
                state = "ready";
                break;
            case "ready":
                System.out.println("Le téléphone est déjà déverrouillé !");
                break;
        }
    }
}
```

Solution avec le Pattern Etat



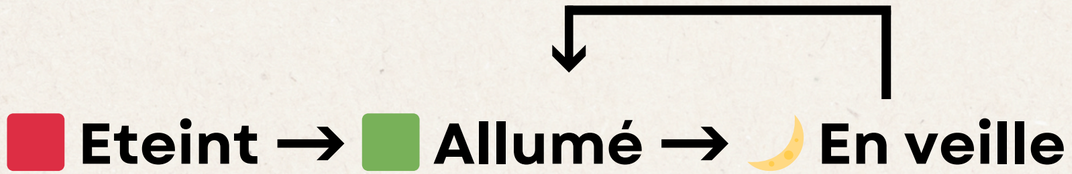
Généralisation du pattern



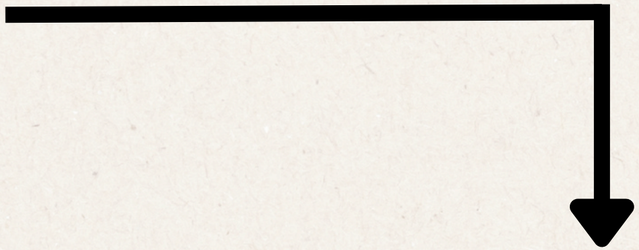
Définition et but

01 GOF : “Permet à un objet de modifier son comportement lorsque son état interne change. L’objet semble changer de classe

02 Téléphone :



03 Pas besoin de condition ou de code dupliqué pour changer son état automatiquement



L’objet ne change pas de classe, il change d’état

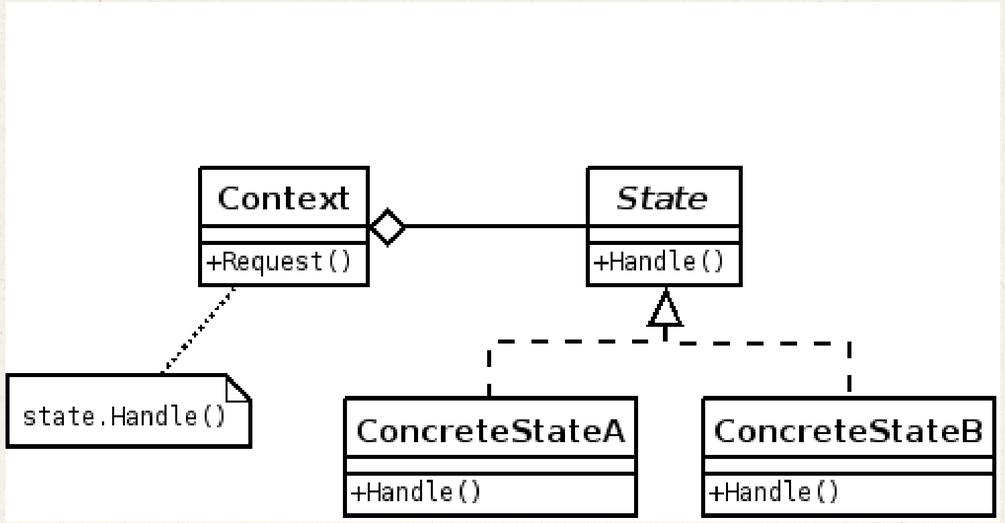
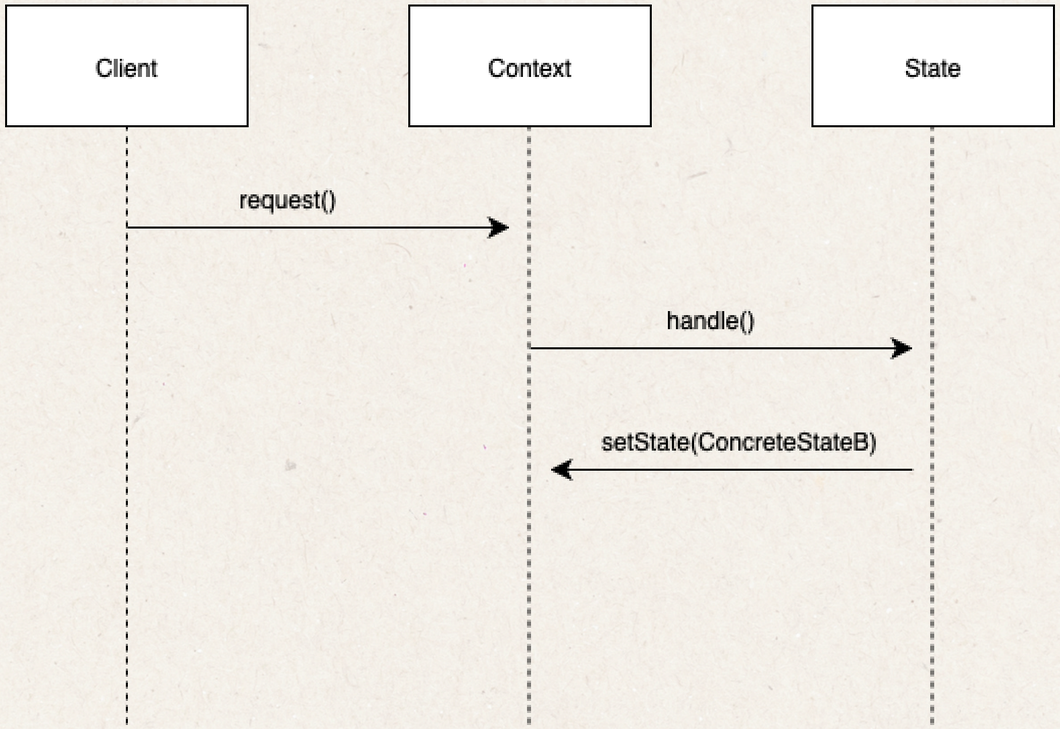


Diagramme UML officiel du pattern État (State)

UML et SEQUENCE



Classes Participantes

Classe	Rôle	Description
Context	Déléguant	Contient une référence vers un State et délègue son comportement à celui-ci. Peut changer d'état via setState().
State (interface / classe abstraite)	Contrat commun	Définit les méthodes que tous les états doivent implémenter (ex. handle()).
ConcreteStateA / B	Etats concrets	Implémentent les comportements spécifiques à chaque état. Peuvent modifier le contexte.

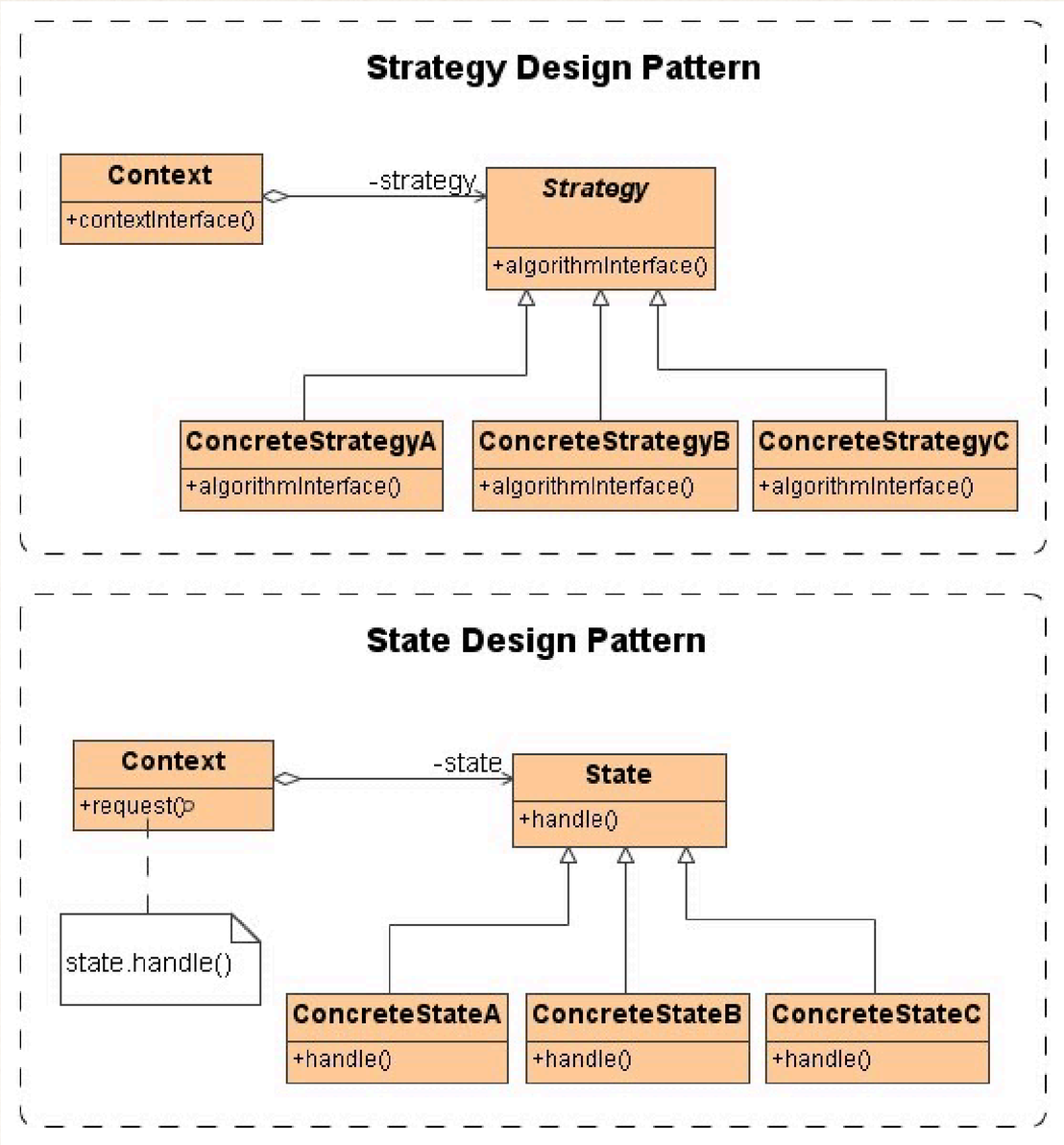
Principe SOLID

Principe SOLID	Application dans le pattern État	Effet obtenu
S – Single Responsibility Principle	Chaque ConcreteState gère un comportement unique (un seul état). Le Context ne fait que déléguer.	Code plus lisible, mieux séparé, plus testable.
O – Open/Closed Principle	On peut ajouter de nouveaux états sans modifier les classes existantes (Context et State).	Le système est extensible sans casser le code.
D – Dependency Inversion Principle	Le Context dépend de l'abstraction State, pas des états concrets.	Faible couplage, haute flexibilité.

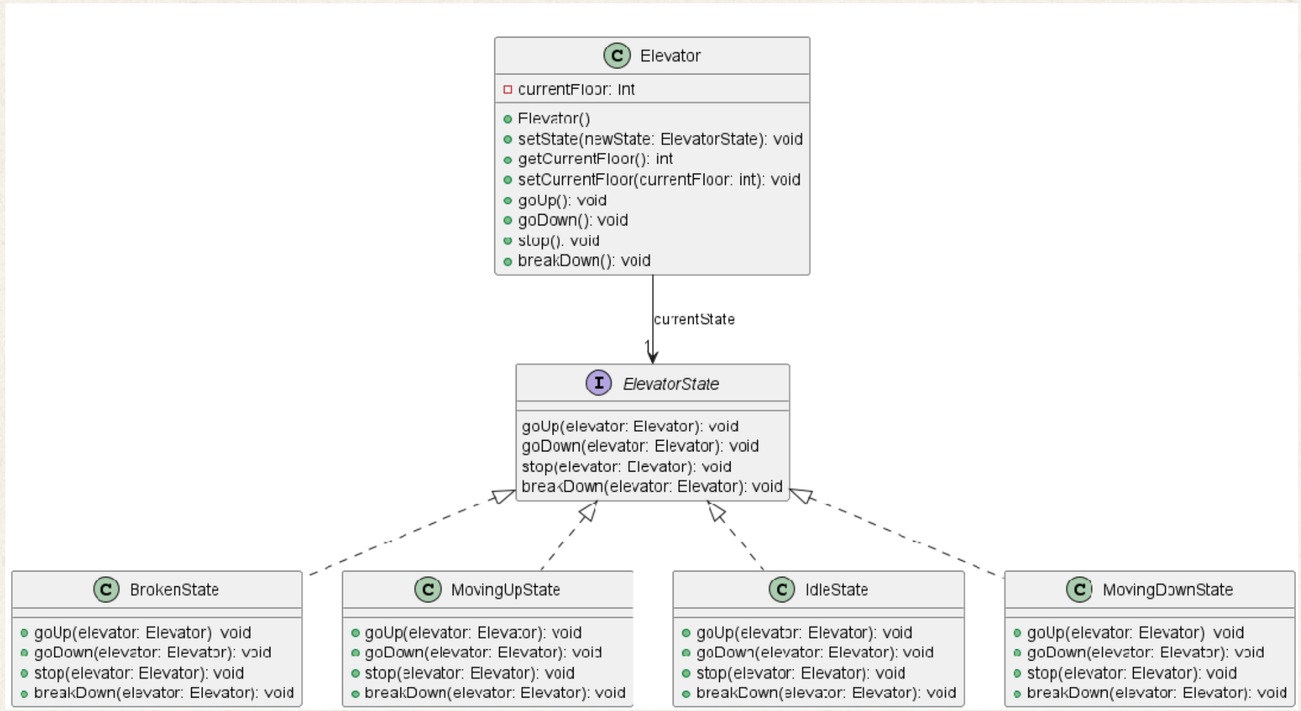
Limites

- Beaucoup de classes à gérer.
- Plus complexe pour de petits cas
- Transitions moins visibles
- Risque de sur-ingénierie

Comparaison : State - Strategy



Deuxième exemple : un ascenseur




```java
public class Elevator {
    private ElevatorState currentState;
    private int currentFloor;

    public Elevator() {
        this.currentState = new IdleState();
        this.currentFloor = 0;
    }

    public void setState(ElevatorState newState) {
        this.currentState = newState;
    }

    public int getCurrentFloor() {
        return currentFloor;
    }

    public void setCurrentFloor(int currentFloor) {
        this.currentFloor = currentFloor;
    }

    public void goUp() {
        currentState.goUp(this);
    }

    public void goDown() {
        currentState.goDown(this);
    }

    public void stop() {
        currentState.stop(this);
    }

    public void breakDown() {
        currentState.breakDown(this);
    }
}
```

```java
public interface ElevatorState {
    void goUp(Elevator elevator);
    void goDown(Elevator elevator);
    void stop(Elevator elevator);
    void breakDown(Elevator elevator);
}
```

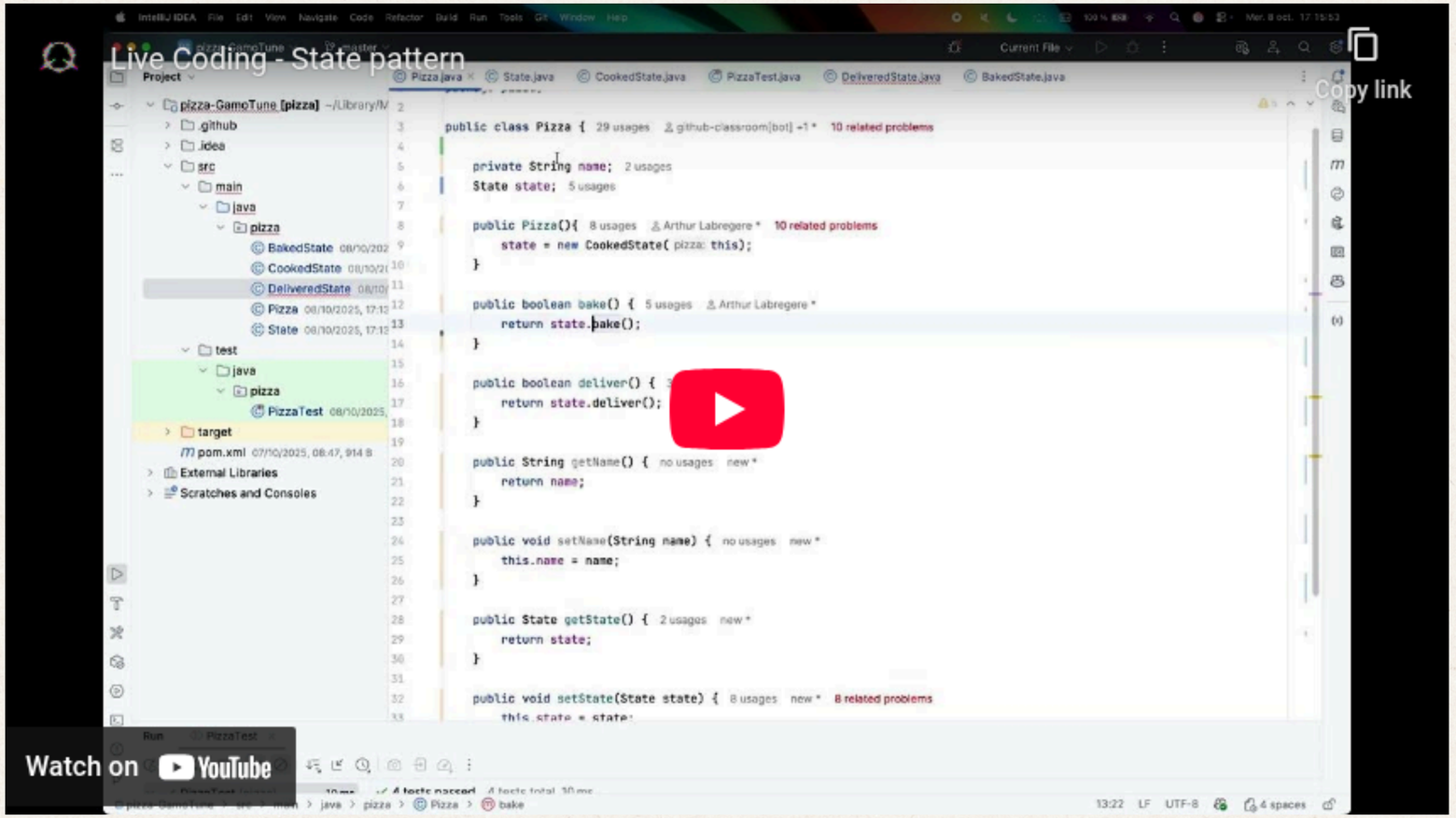
```java
public class IdleState implements ElevatorState {

    @Override
    public void goUp(Elevator elevator) {
        System.out.println("Elevator starts going up...");
        elevator.setState(new MovingUpState());
    }

    @Override
    public void goDown(Elevator elevator) {
        System.out.println("Elevator starts going down...");
        elevator.setState(new MovingDownState());
    }

    @Override
    public void stop(Elevator elevator) {
        System.out.println("Elevator is already idle.");
    }

    @Override
    public void breakDown(Elevator elevator) {
        System.out.println("Elevator broke down!");
        elevator.setState(new BrokenState());
    }
}
```

```java
public class MovingDownState implements ElevatorState {

    @Override
    public void goUp(Elevator elevator) {
        System.out.println("Cannot go up while moving down!");
    }

    @Override
    public void goDown(Elevator elevator) {
        elevator.setCurrentFloor(elevator.getCurrentFloor() - 1);
        System.out.println("Elevator is moving down to floor " + elevator.getCurrentFloor());
    }

    @Override
    public void stop(Elevator elevator) {
        System.out.println("Elevator stops at floor " + elevator.getCurrentFloor());
        elevator.setState(new IdleState());
    }

    @Override
    public void breakDown(Elevator elevator) {
        System.out.println("Elevator broke down while going down!");
        elevator.setState(new BrokenState());
    }
}
```

```java
public class MovingUpState implements ElevatorState {

    @Override
    public void goUp(Elevator elevator) {
        elevator.setCurrentFloor(elevator.getCurrentFloor() + 1);
        System.out.println("Elevator is moving up to floor " + elevator.getCurrentFloor());
    }

    @Override
    public void goDown(Elevator elevator) {
        System.out.println("Cannot go down while moving up!");
    }

    @Override
    public void stop(Elevator elevator) {
        System.out.println("Elevator stops at floor " + elevator.getCurrentFloor());
        elevator.setState(new IdleState());
    }

    @Override
    public void breakDown(Elevator elevator) {
        System.out.println("Elevator broke down while going up!");
        elevator.setState(new BrokenState());
    }
}
```

```java
public class BrokenState implements ElevatorState {

    @Override
    public void goUp(Elevator elevator) {
        System.out.println("Elevator is broken. Cannot move up.");
    }

    @Override
    public void goDown(Elevator elevator) {
        System.out.println("Elevator is broken. Cannot move down.");
    }

    @Override
    public void stop(Elevator elevator) {
        System.out.println("Elevator is broken and already stopped.");
    }

    @Override
    public void breakDown(Elevator elevator) {
        System.out.println("Elevator is already broken!");
    }
}
```

Live Coding



Bibliographie

- refactoring.guru
- fr.wikipedia.org
- programisto.fr
- sourcemaking.com
- dofactory.com
- digitalocean.com