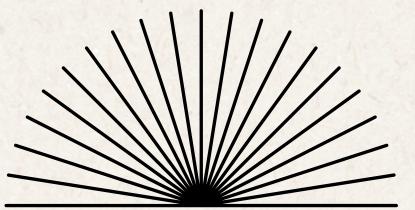


# Pattern Etat

Présenté par :

Mathis GRANIER - Thurian SACRISTAN  
Milan LOI - Arthur LABREGERE



# Sommaire

<b>03</b>	<b>Pattern Gof</b>
<b>04</b>	<b>Exemple de besoin</b>
<b>05</b>	<b>Modélisation</b>
<b>06</b>	<b>Solution avec le pattern</b>
<b>07</b>	<b>Généralisation du pattern</b>
<b>08</b>	<b>Définition et but</b>
<b>09</b>	<b>Diagramme UML du pattern</b>
<b>10</b>	<b>Classes Participantes</b>
<b>11</b>	<b>Principe SOLID</b>
<b>12</b>	<b>Limites</b>
<b>13</b>	<b>Comparaison avec un autre pattern</b>
<b>14</b>	<b>Deuxième exemple</b>
<b>15</b>	<b>Live Coding</b>
<b>16</b>	<b>Bibliographie</b>

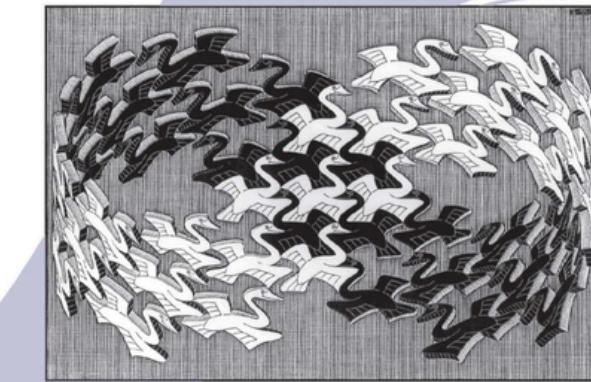
# Pattern Gof

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

# Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

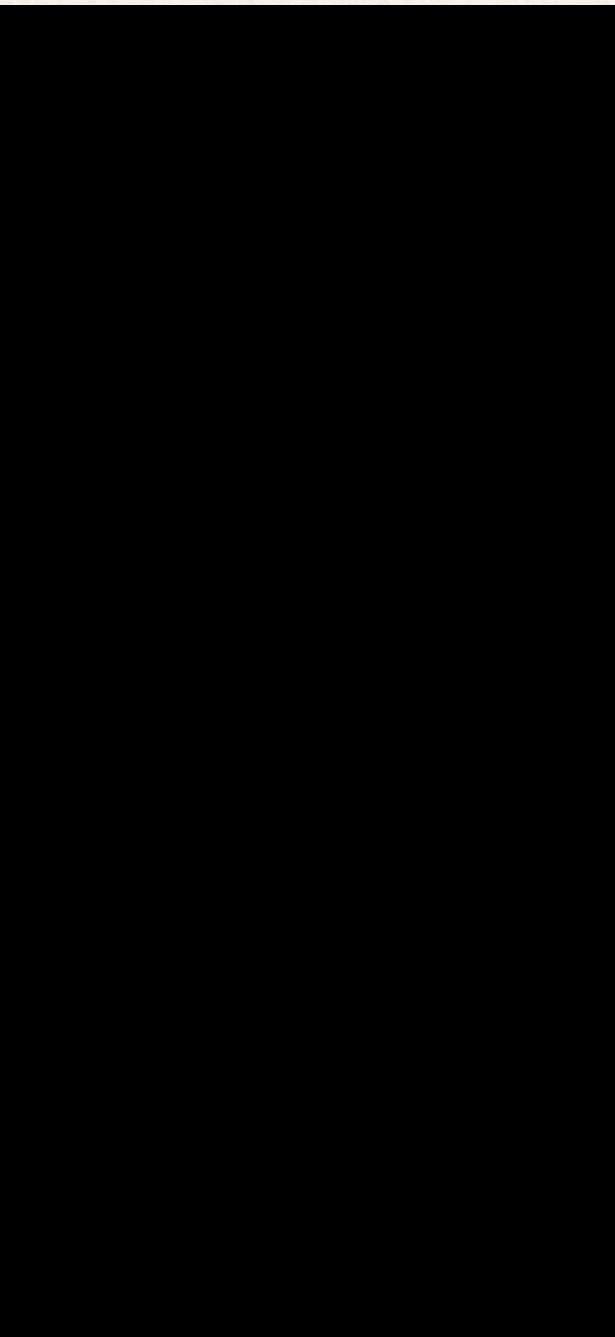
Foreword by Grady Booch



1994

# Exemple de besoin

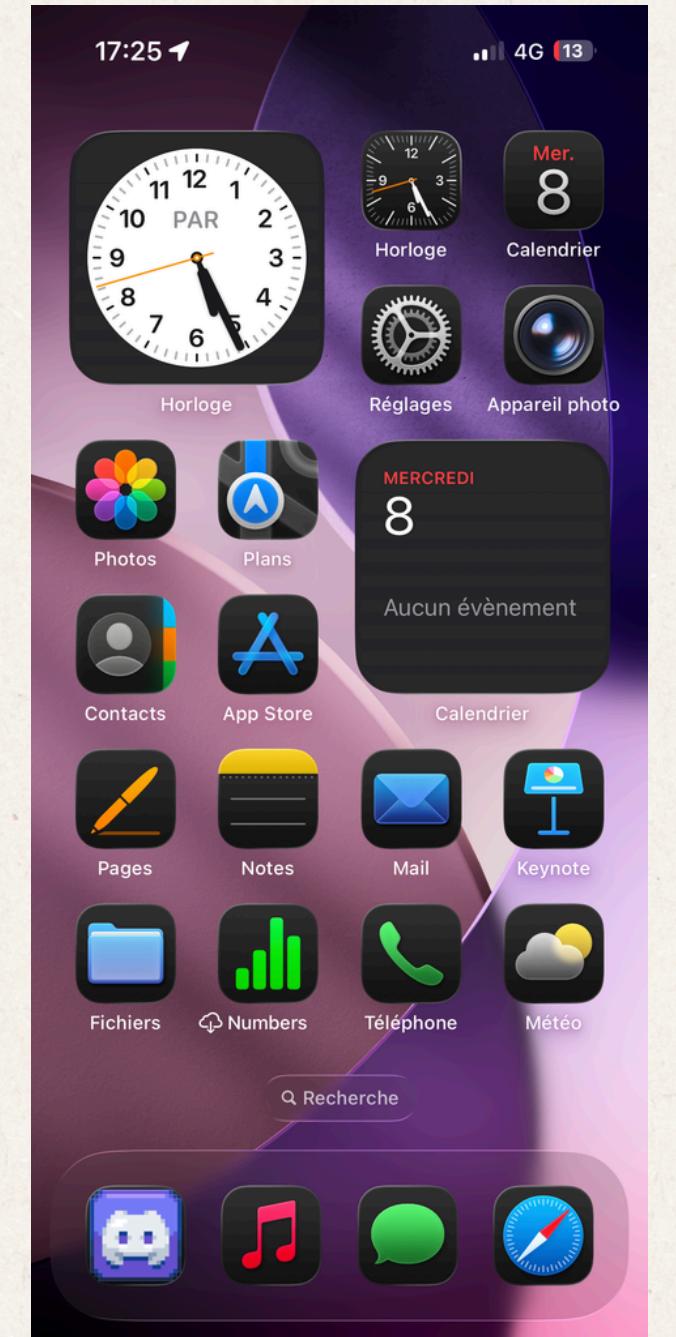
Off



Locked



Ready



# Modélisation

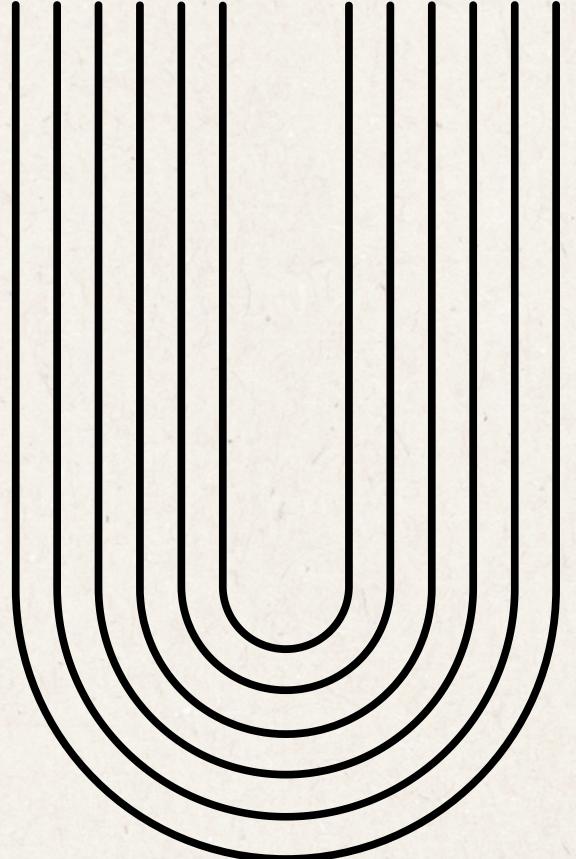
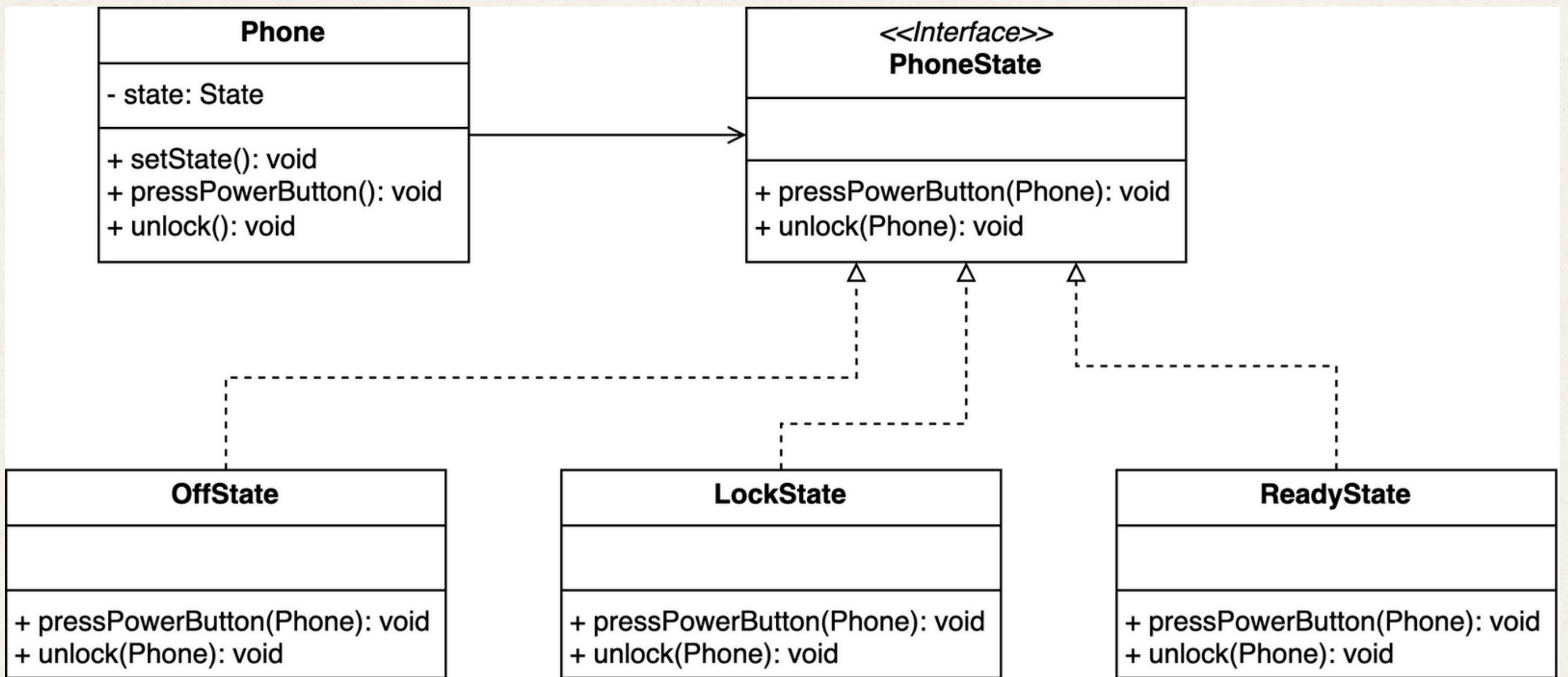
Phone
- state: String
+ Phone()
+ pressPowerButton(): void
+ unlock(): void

✗ Aucun découplage entre les comportements selon l'état.

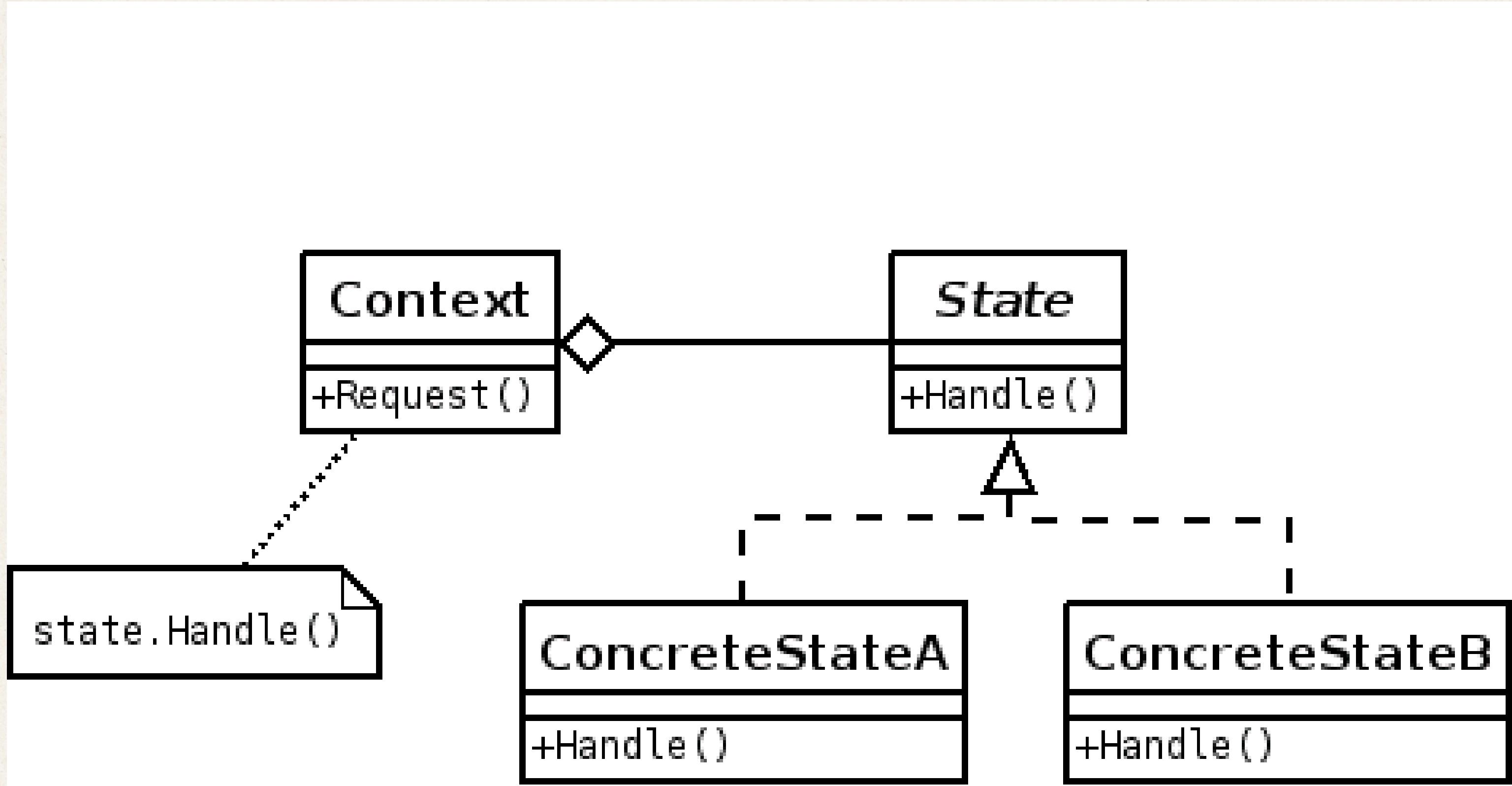
✗ Ajouter un nouvel état nécessiterait de modifier le code du Phone.

```
© Phone.java ×
1  public class Phone { 10 usages
2      private String state; 7 usages
3
4      public Phone() { 1 usage
5          this.state = "off"; // Par défaut, le téléphone est éteint
6      }
7
8      public void pressPowerButton() { 3 usages
9          switch (state) {
10             case "off":
11                 System.out.println("Le téléphone s'allume...");
12                 state = "locked";
13                 break;
14             case "locked":
15                 System.out.println("Le téléphone s'éteint...");
16                 state = "off";
17                 break;
18             case "ready":
19                 System.out.println("Le téléphone se verrouille...");
20                 state = "locked";
21                 break;
22         }
23     }
24
25     public void unlock() { 2 usages
26         switch (state) {
27             case "off":
28                 System.out.println("Impossible : le téléphone est éteint !");
29                 break;
30             case "locked":
31                 System.out.println("Déverrouillage réussi !");
32                 state = "ready";
33                 break;
34             case "ready":
35                 System.out.println("Le téléphone est déjà déverrouillé !");
36                 break;
37         }
38     }
39 }
```

# Solution avec le Pattern Etat



# Généralisation du pattern



# Définition et but

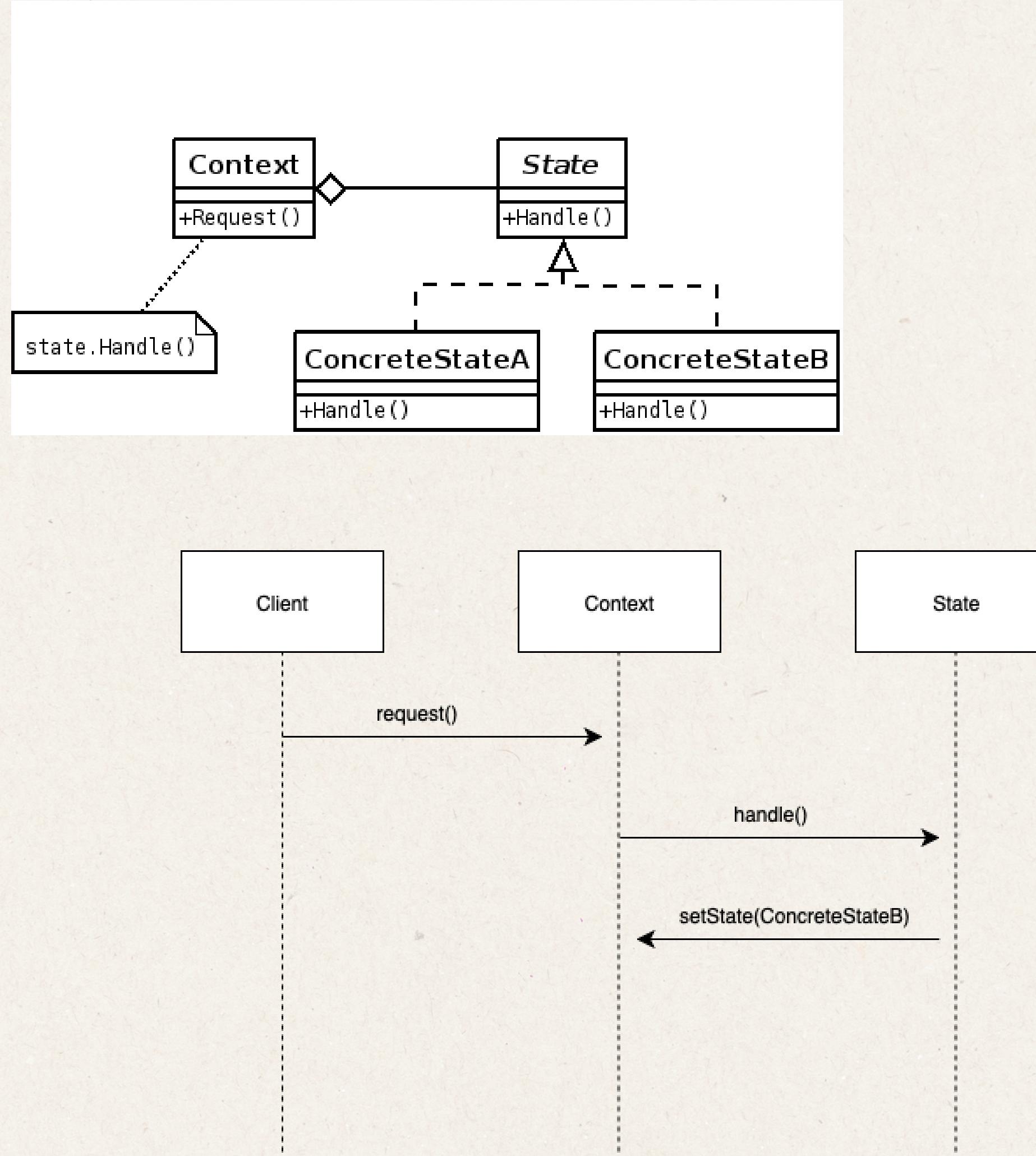
01 GOF : “Permet à un objet de modifier son comportement lorsque son état interne change. L’objet semble changer de classe

02 Téléphone :

■ Eteint → ■ Allumé → ☺ En veille

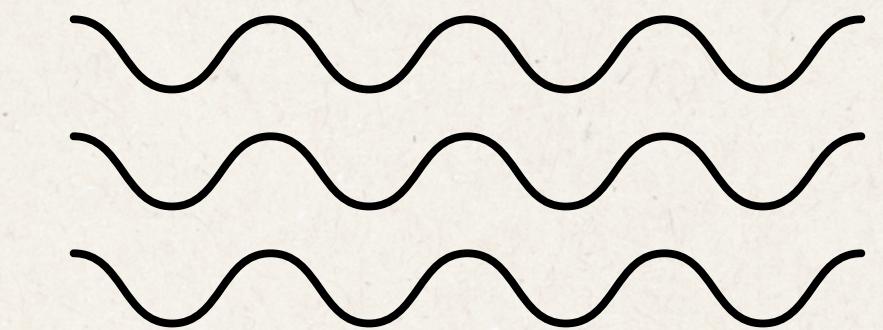
03 Pas besoin de condition ou de code dupliqué pour changer son état automatiquement

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



# Diagramme UML officiel du pattern État (State)

UML et SEQUENCE

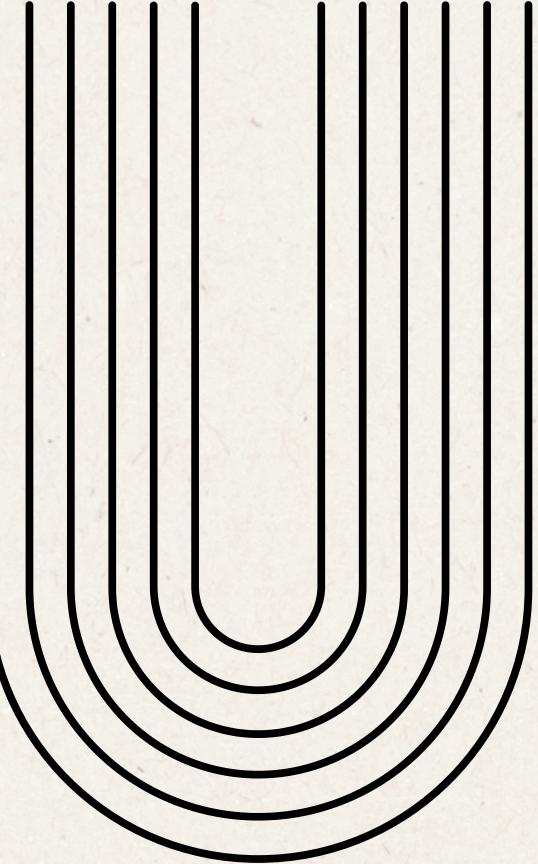


# Classes Participantes

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

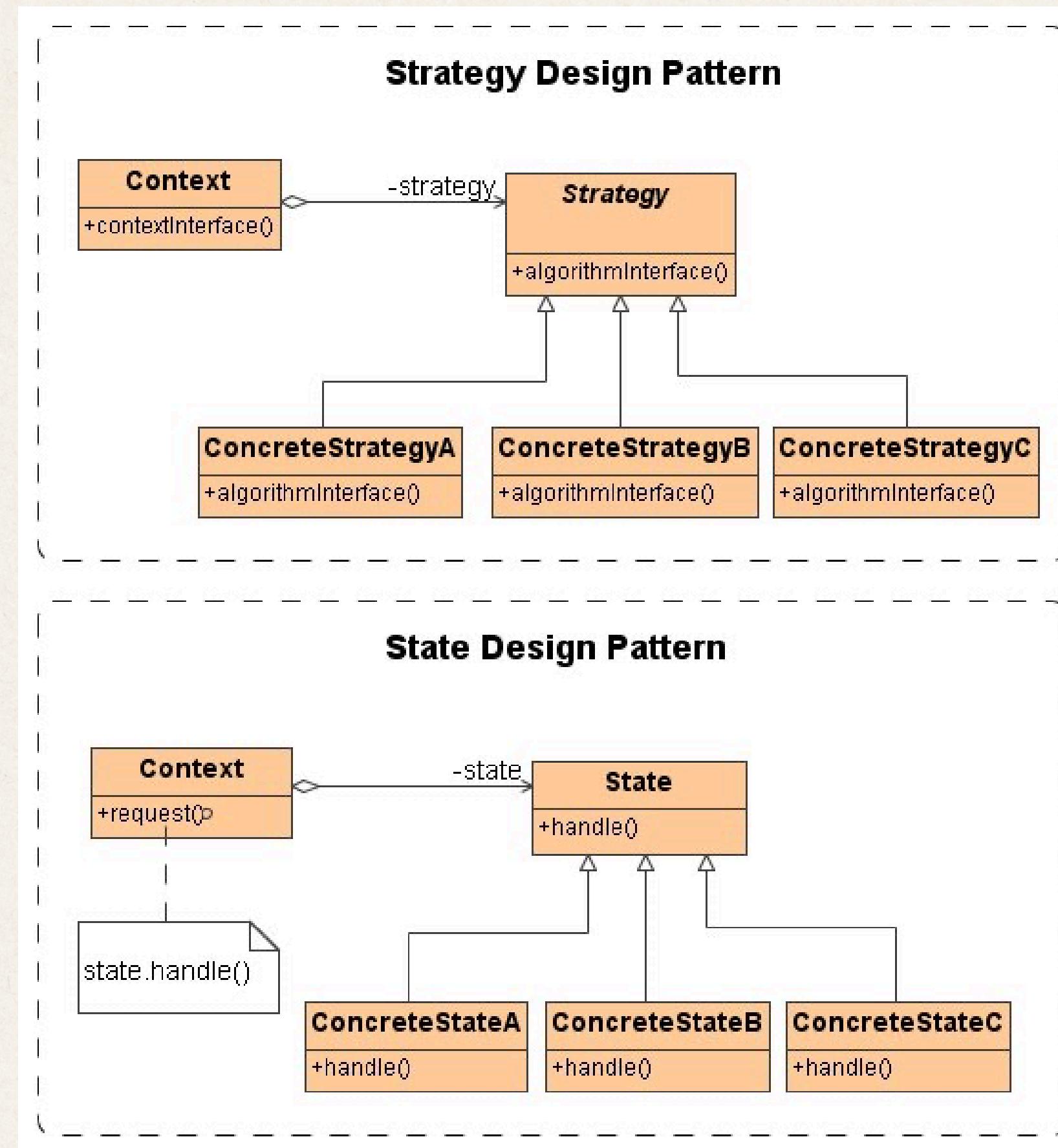
Principe SOLID	Application dans le pattern État	Effet obtenu
S – Single Responsibility Principle	Chaque ConcreteState gère un comportement unique (un seul état). Le Context ne fait que déléguer.	Code plus lisible, mieux séparé, plus testable.
O – Open/Closed Principle	On peut ajouter de nouveaux états 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.



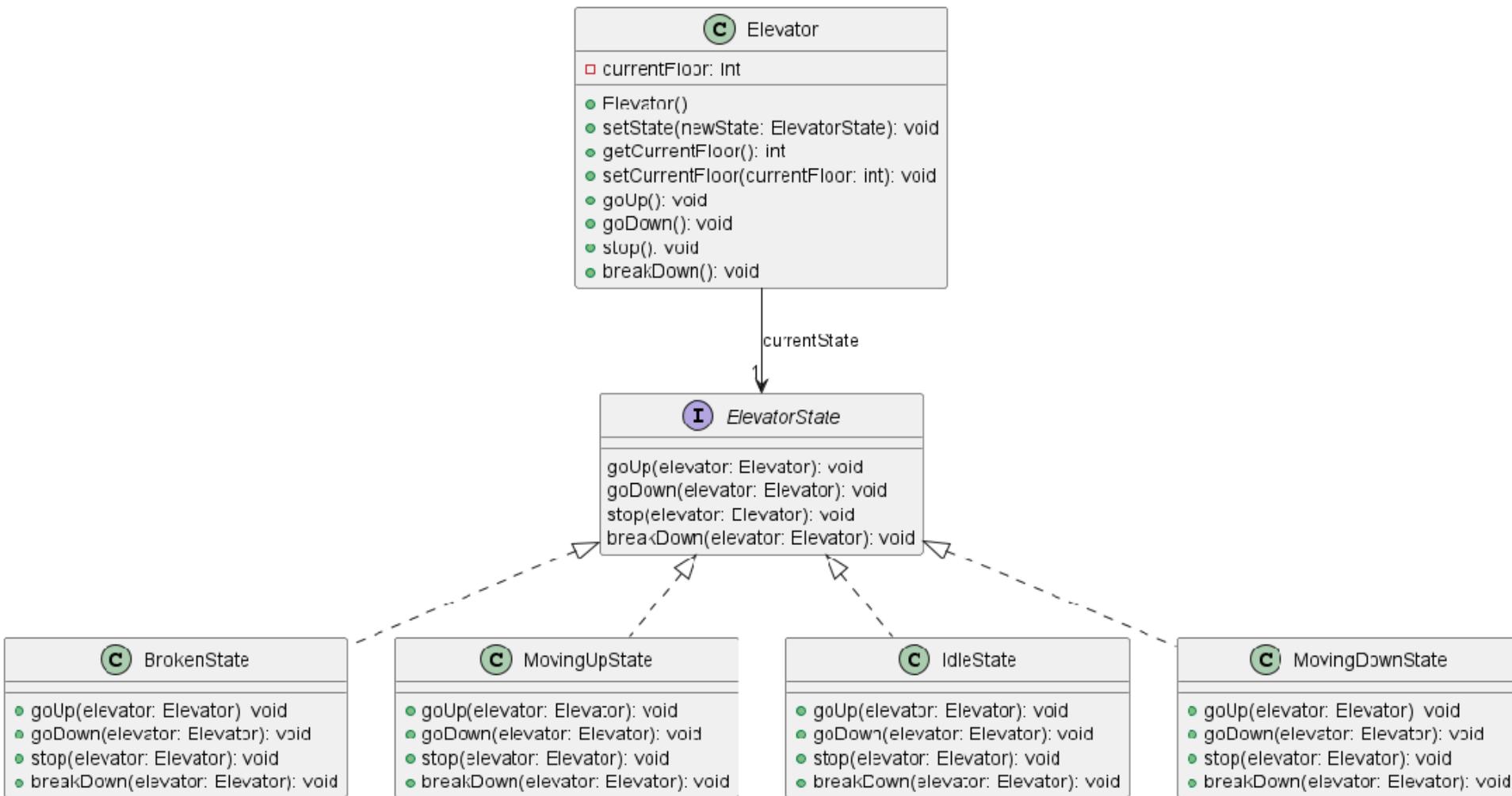
# Limites

- 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



```

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);
    }
}

```

```

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

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());
    }
}

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());
    }
}

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!");
    }
}

```

```

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());
    }
}

```

```

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());
    }
}

```

```

public class BrokenState implements ElevatorState {

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

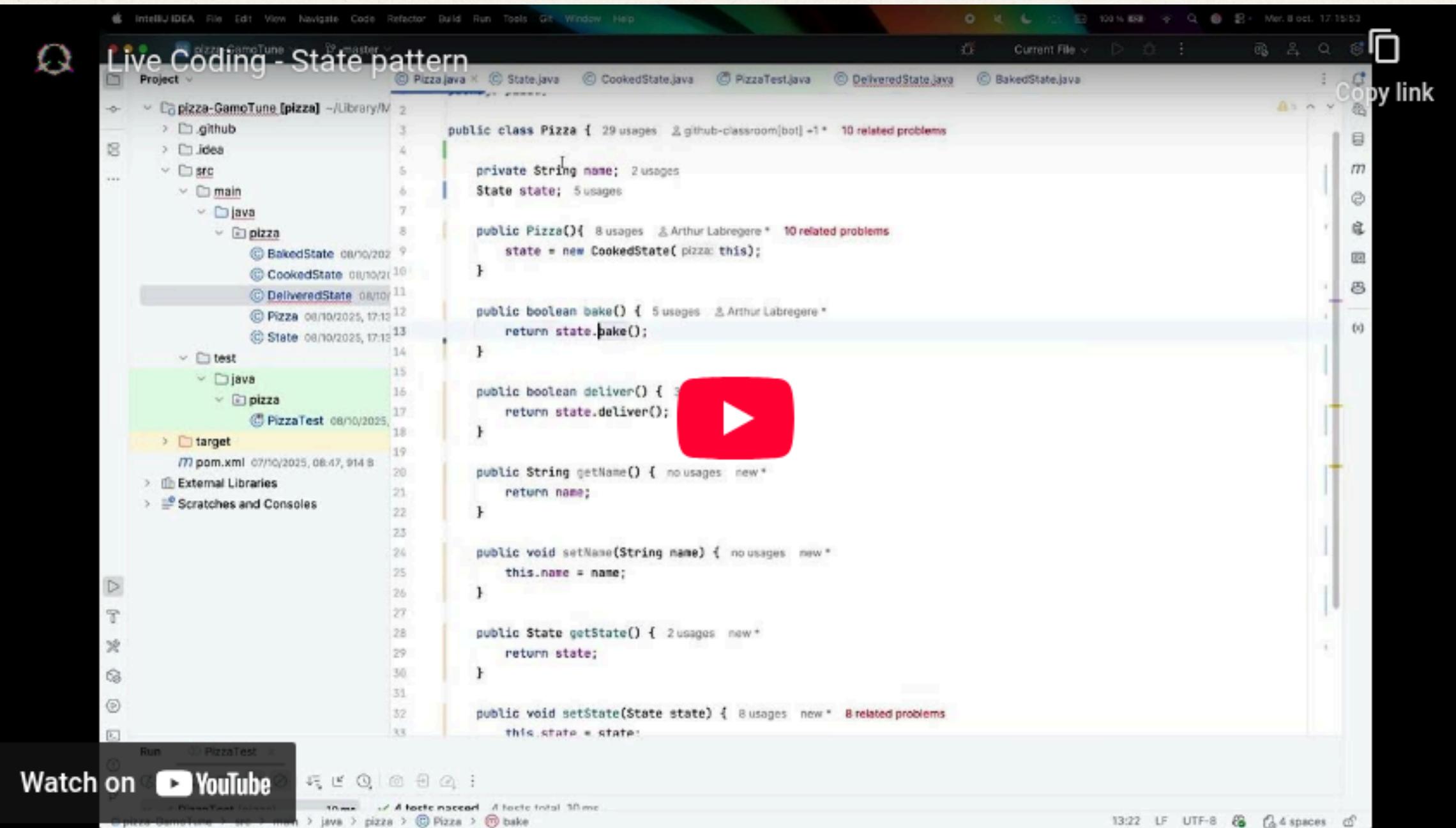
    @Override
    public void goDown(Elevator elevator) {
        System.out.println("Elevator is broken. Cannot move down.");
    }

    @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



The screenshot shows the IntelliJ IDEA interface with the following details:

- Title Bar:** IntelliJ IDEA, File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, Git, Window, Help.
- Toolbar:** Current File, Copy link (highlighted in red).
- Project Tool Window:** Shows the project structure for "pizza-GamoTune [pizza]".
- Code Editor:** Displays the `Pizza.java` file containing the following code:

```
public class Pizza { 29 usages 2 github-classroom[bot] +1 * 10 related problems
    private String name; 2 usages
    State state; 5 usages

    public Pizza(){ 8 usages 2 Arthur Labregere * 10 related problems
        state = new CookedState( pizza: this);
    }

    public boolean bake() { 5 usages 2 Arthur Labregere *
        return state.bake();
    }

    public boolean deliver() { 2 usages 2 Arthur Labregere *
        return state.deliver();
    }

    public String getName() { no usages new *
        return name;
    }

    public void setName(String name) { no usages new *
        this.name = name;
    }

    public State getState() { 2 usages new *
        return state;
    }

    public void setState(State state) { 8 usages new * 8 related problems
        this.state = state;
    }
}
```

**Bottom Status Bar:** Watch on YouTube, Run PizzaTest, 13:22, LF, UTF-8, 4 spaces.

# Bibliographie

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