Pattern Etat

Sommaire

- Introduction
- Mise en contexte
- Pattern du GoF
- Principes SOLID
- Lien avec Stratégie
- Application du pattern
- Live coding
- QCM
- Bibliographie

L’état français

L’état dmorv

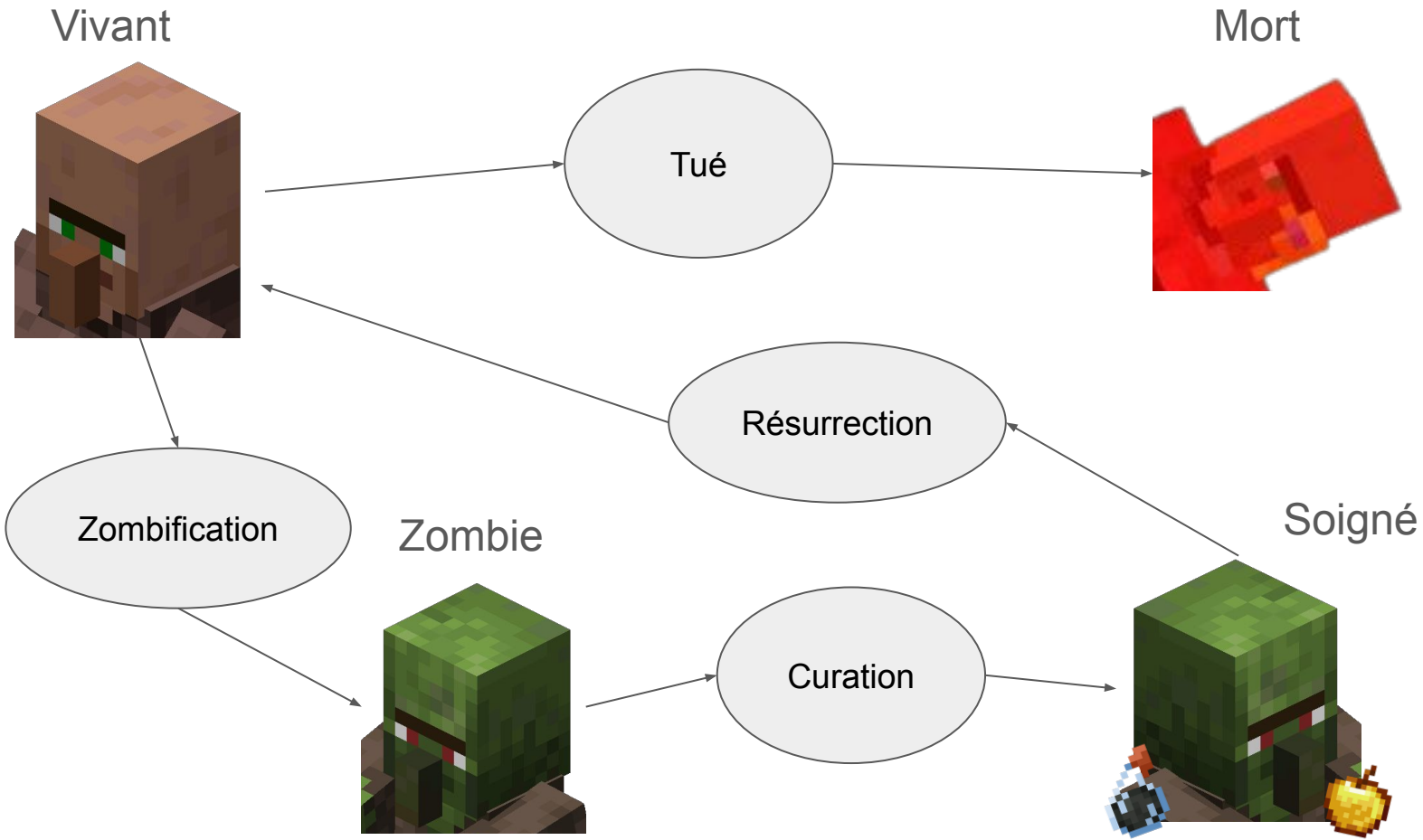
L’état xis

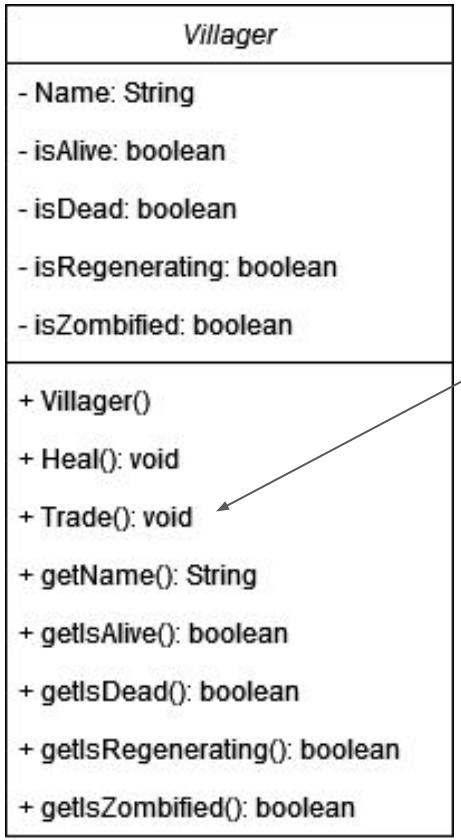
L'état ta Yoyo, qu'est-ce qu'y a sous ton grand chapeau?

L’état d’un objet

Quésaco ?

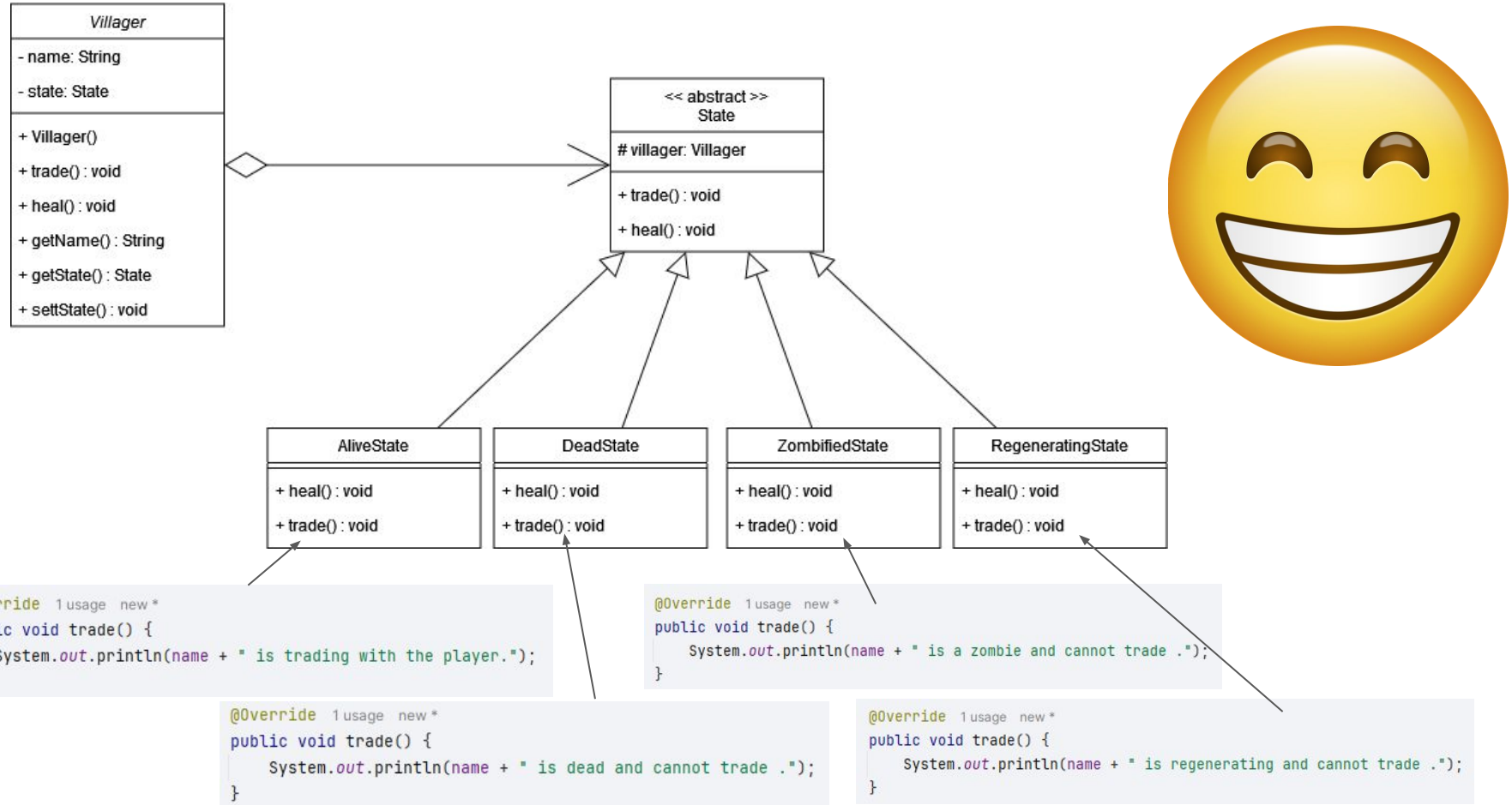
- La “forme” actuelle d’un objet
- L’objet agit différemment en fonction de son état actuel


Vivant
Mort
Tué
Résurrection
Zombification
Zombie
Curation
Soigné

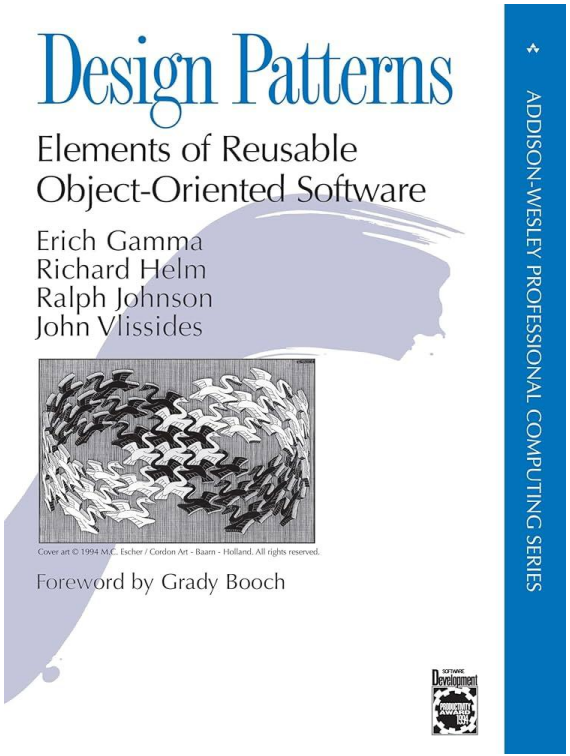



```java
public void trade() {  1 usage  new *
    if (isAlive) {
        System.out.println(name + " is trading with the player.");
    } else if (isDead) {
        System.out.println(name + " is dead and cannot trade.");
    } else if (isZombified) {
        System.out.println(name + " is a zombie and cannot trade.");
    } else if (isRegenerating) {
        System.out.println(name + " is regenerating and cannot trade.");
    }
}
```

Hmmm
sad villager noise


Villager
- name: String
- state: State
+ Villager()
+ trade() : void
+ heal() : void
+ getName() : String
+ getState() : State
+ settState() : void
<< abstract >>
State
villager: Villager
+ trade() : void
+ heal() : void
AliveState
+ heal() : void
+ trade() : void
DeadState
+ heal() : void
+ trade() : void
ZombifiedState
+ heal() : void
+ trade() : void
RegeneratingState
+ heal() : void
+ trade() : void
@Override 1 usage new *
public void trade() {
System.out.println(name + " is trading with the player.");
}
@Override 1 usage new *
public void trade() {
System.out.println(name + " is a zombie and cannot trade .");
}
@Override 1 usage new *
public void trade() {
System.out.println(name + " is dead and cannot trade .");
}
@Override 1 usage new *
public void trade() {
System.out.println(name + " is regenerating and cannot trade .");
}

Gang Of Four (GoF)



Pattern Etat

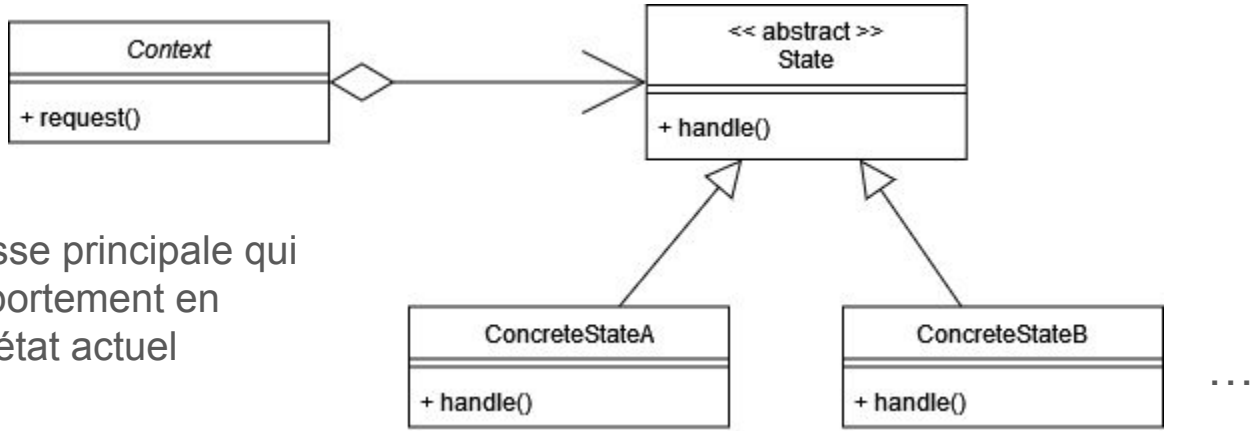
Type comportemental

- Ce type de pattern aide à améliorer les interactions entre les objets et dans les objets pour assurer une meilleure flexibilité

But du pattern

- Permet à l’objet de changer totalement de comportement quand son état interne est modifié

Diagramme de classes

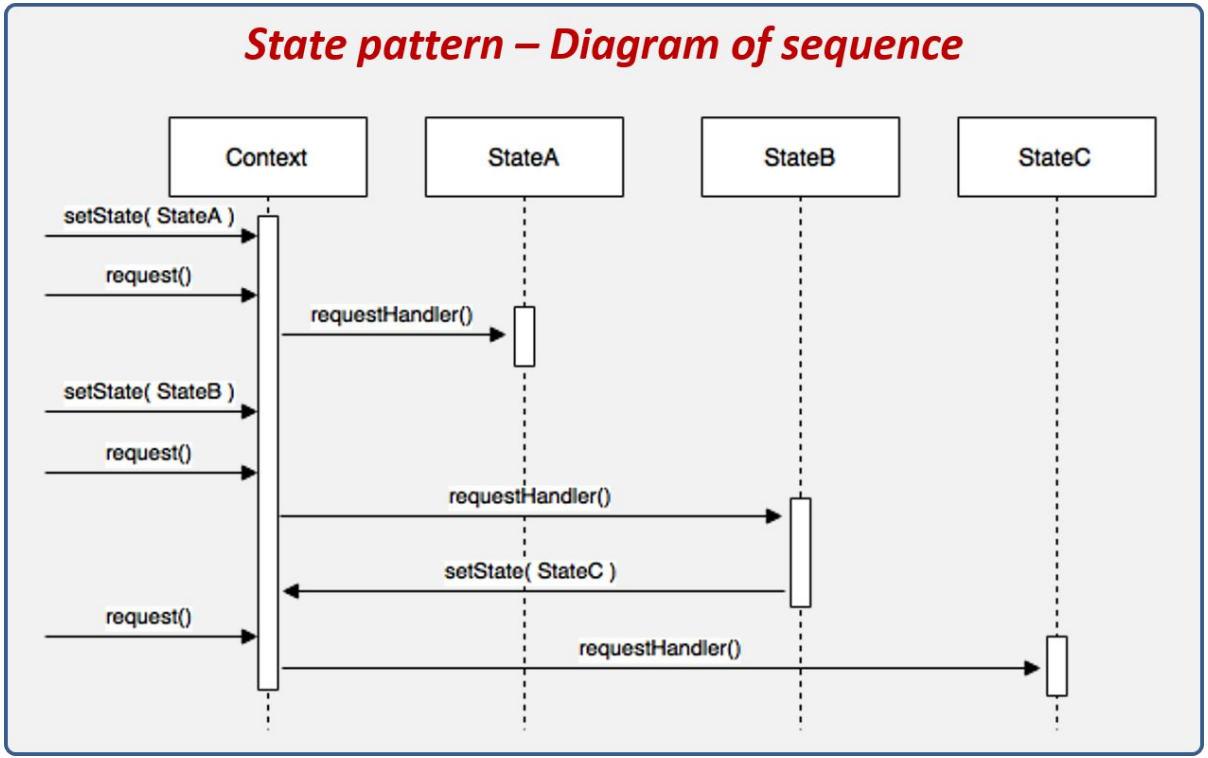


Context: La classe principale qui change de comportement en fonction de son état actuel

ConcreteState: Représentent les différents états que peut avoir la classe **Context**

State: Classe utilisé pour générer les différents états

Diagramme de séquences



Ce pattern permet de respecter 2 principes SOLID en évitant les IF imbriqués:

- SRP: **Single Responsibility Principle**, la classe **Context** ne dois pas gérer le changement de ses états.

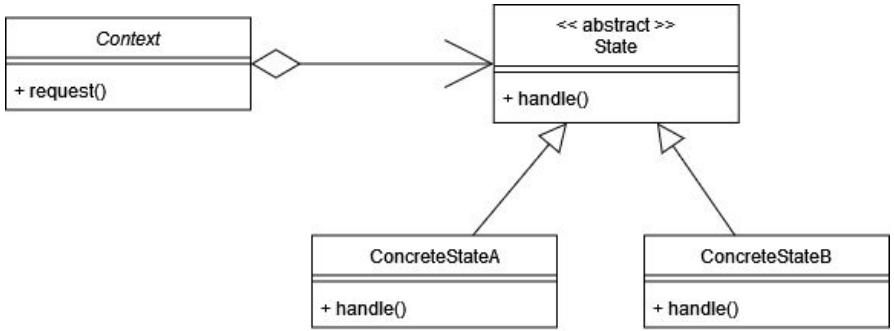
- OCP: **Open/Closed Principle**, l'ajout d'état doit se faire avec des nouvelles classes et non en ajoutant des imbrications de IF.

Cependant ce pattern possède quelques limites:

- **Sur-utilisation**: utiliser ce pattern avec un objet qui change rarement d'état ou qui n'en possède très peu peut représenter une sur-utilisation du pattern.

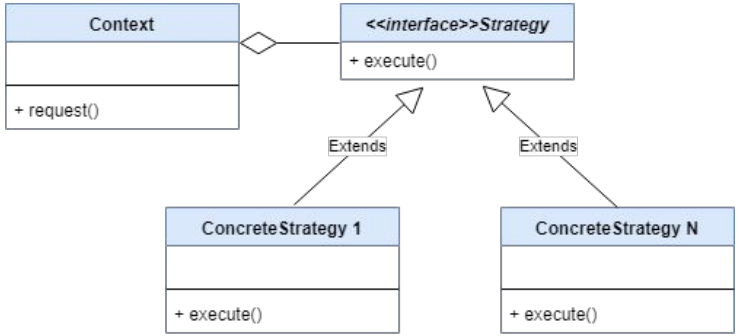
- (Difficile à comprendre: Dans le cas d'un objet qui aurait des dizaines d'état changeant souvent, la lecture du code peut être compliqué pour un débutant du pattern)

Pattern état



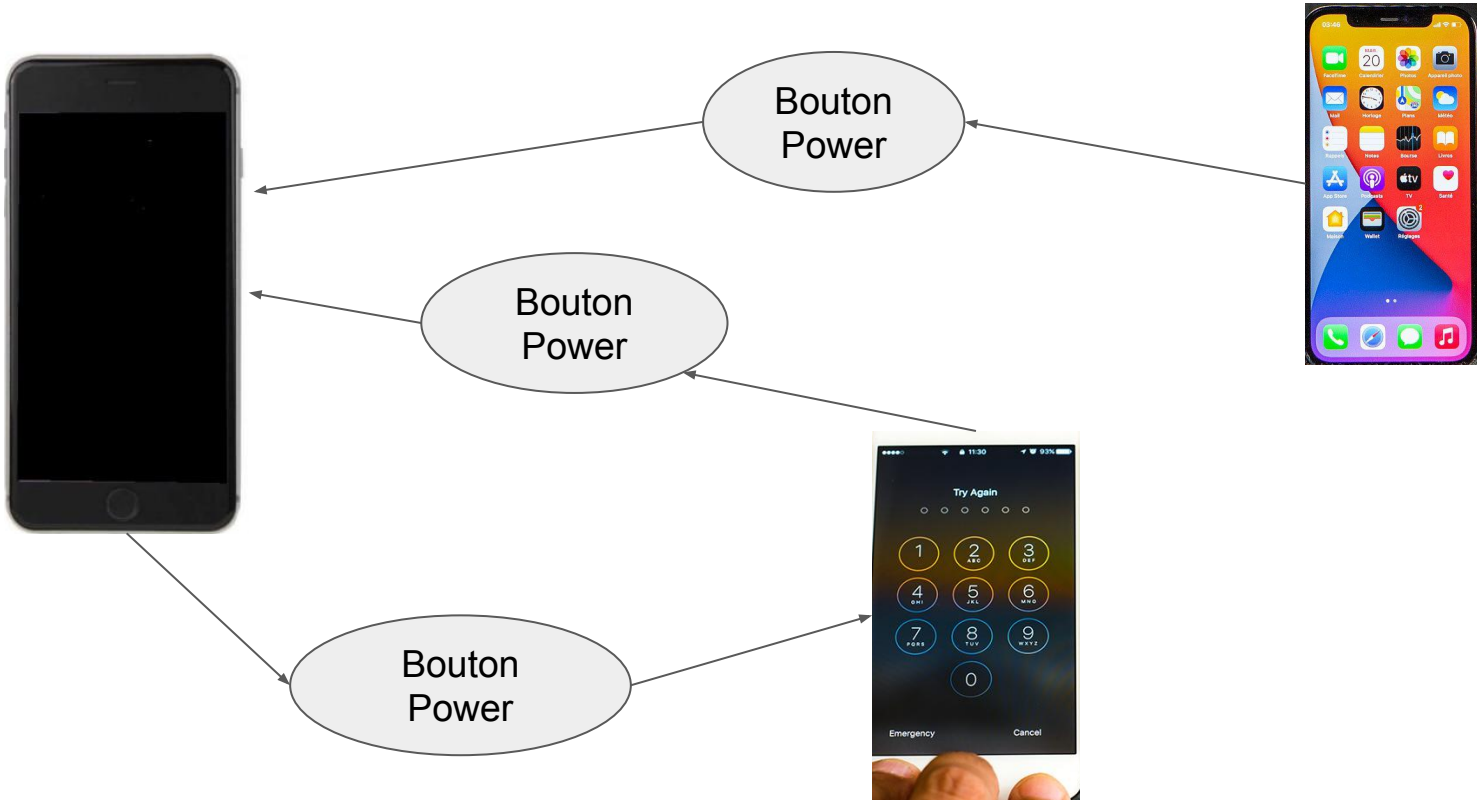
Les états sont complètement indépendant les uns des autres et change le comportement interne de l'objet

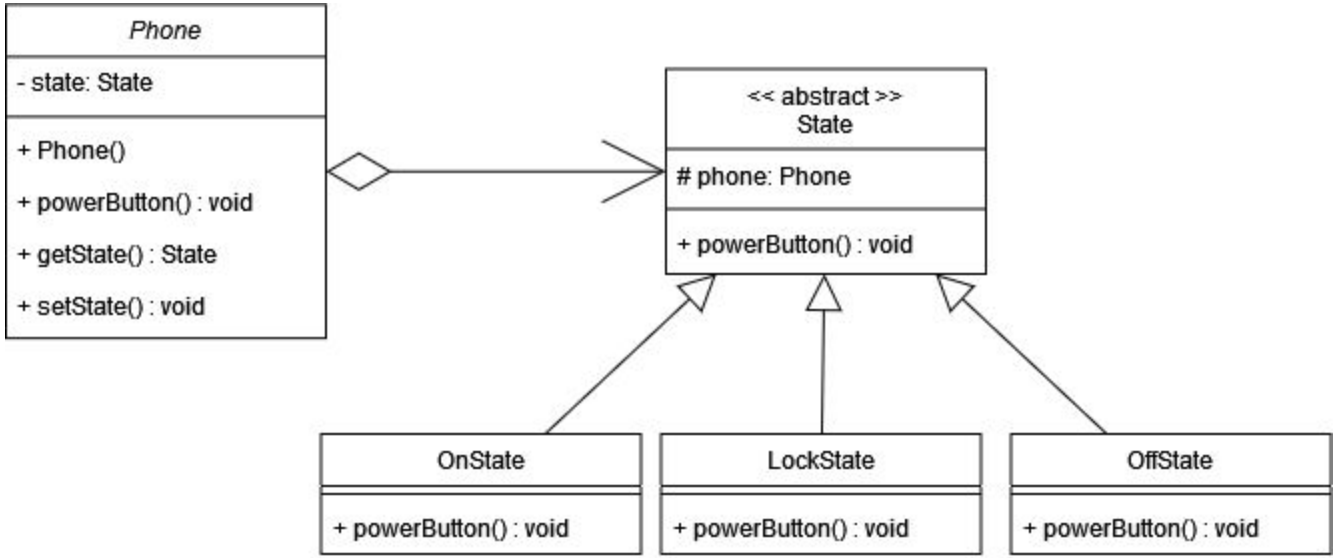
Pattern Stratégie



Ce pattern permet d'implémenter plusieurs stratégie qui ont le même but

— Lien avec Stratégie —


Bouton
Power
Bouton
Power
Bouton
Power
Try Again
Emergency
Cancel



— Application du pattern —

```java
public class Phone {  5 usages  new *
    private State state;  3 usages

    public Phone() {  no usages  new *
        state = new OffState( phone: this);
    }

    public void powerButton() {  no usages  new *
        state.powerButton();
    }

    public void setState(State state) {  3 usages  new *
        this.state = state;
    }
}
```

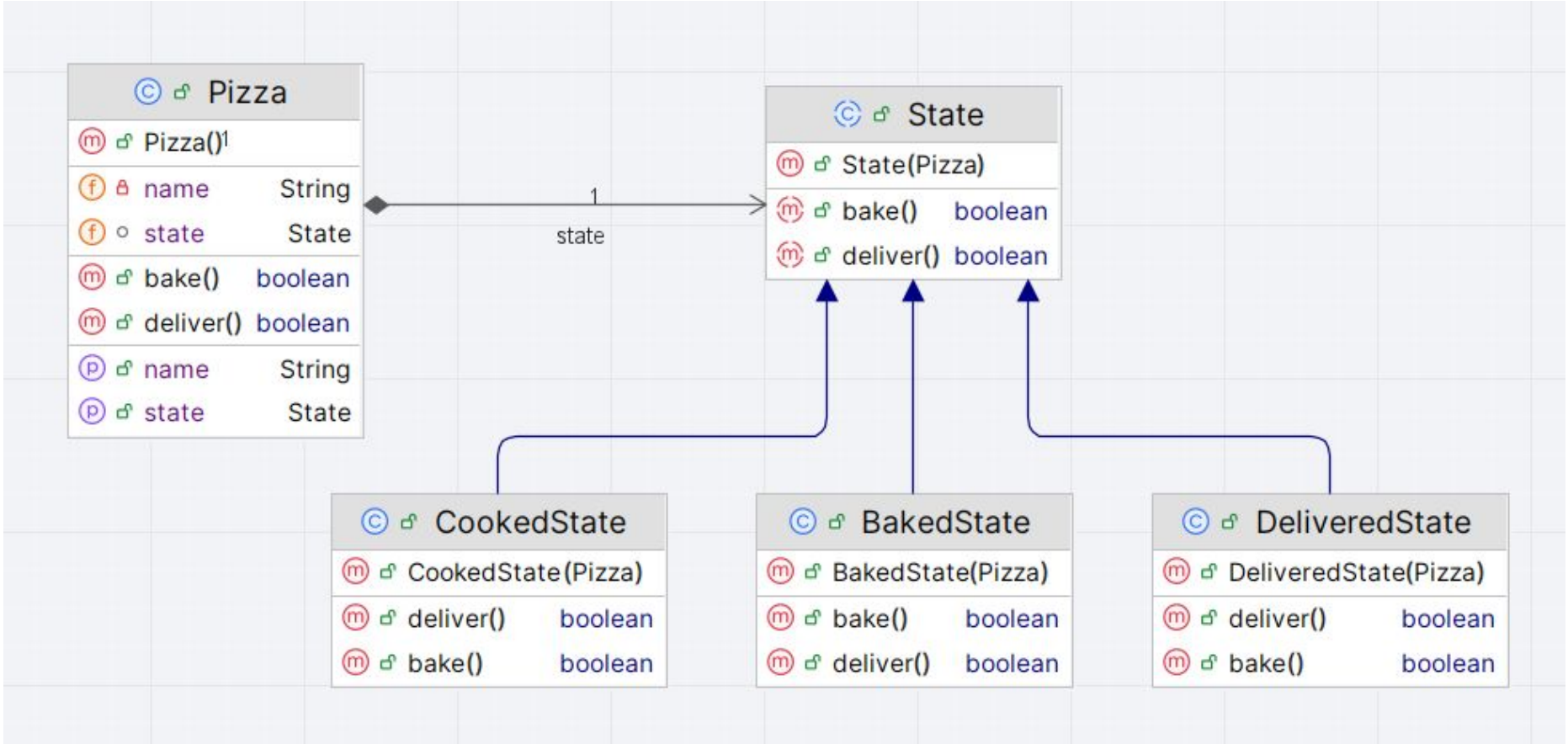
```java
public abstract class State {  11 usages  6 inheritors
    protected Phone phone;  7 usages

    protected State(Phone phone) {  no usages  new
        this.phone = phone;
    }

    public abstract void powerButton();  1 usage
}
```

```java
public class OffState extends State {  3 usages  new *

    public OffState(Phone phone) {  3 usages  new *
        super(phone);
    }

    @Override  1 usage  new *
    public void powerButton() {
        phone.setState(new LockState(phone));
    }
}
```

```java
public class OnState extends State {  no usages  new *

    public OnState(Phone phone) {  no usages  new *
        super(phone);
    }

    @Override  1 usage  new *
    public void powerButton() {
        phone.setState(new OffState(phone));
    }
```

```java
public class LockState extends State {  1 usage  new *

    public LockState(Phone phone) {  1 usage  new *
        super(phone);
    }

    @Override  1 usage  new *
    public void powerButton() {
        phone.setState(new OffState(phone));
    }
}
```

LIVE CODING


Pizza
Pizza()
name String
state State
bake() boolean
deliver() boolean
name String
state State
1
state
State
State(Pizza)
bake() boolean
deliver() boolean
CookedState
CookedState(Pizza)
deliver() boolean
bake() boolean
BakedState
BakedState(Pizza)
bake() boolean
deliver() boolean
DeliveredState
DeliveredState(Pizza)
deliver() boolean
bake() boolean

— Live Coding —

QCM

Bibliographie

- https://reactiveprogramming.io/blog/en/design-patterns/state
- https://refactoring.guru/design-patterns/state
- https://en.wikipedia.org/wiki/State_pattern
- https://www.youtube.com/watch?v=abX4xzaAsoc