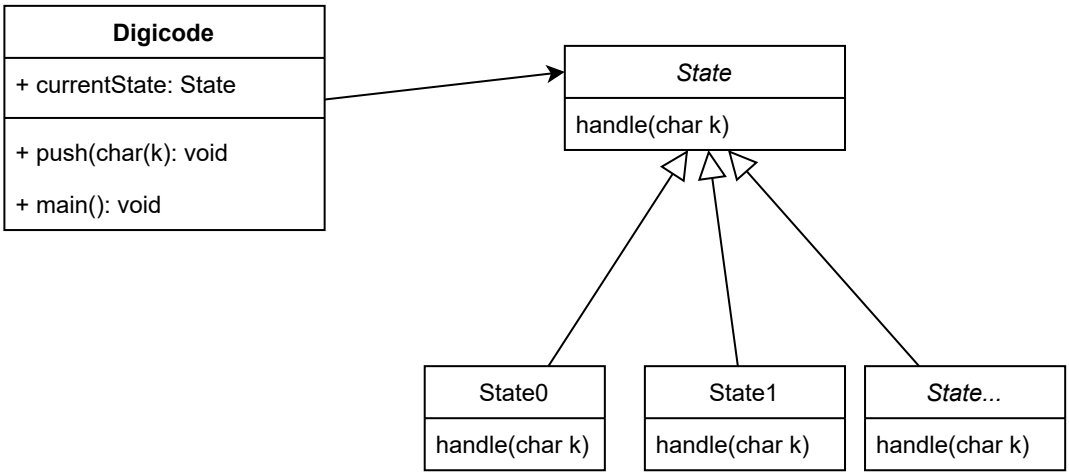
TP7 - État / Stratégie

#DesignPattern

État

Le pattern d'état permet de modifier le comportement d'un objet lorsque son état interne change (tout se passe comme si l'objet changeait de classe) Pour cela, on va externaliser les méthodes qui dépendent de l'état particulier d'un objet




```
// créons notre classe abstraite State, ainsi que ses états concrets :
public abstract class State {

    public State handle(char k) {
        return new StateNotOK(); // état par défaut
    }

}

/*********************************    State 0
*****************************/
public class State0 extends State {

    @Override
    public State handle(char k) {
        if (k == 'a') {
            return new State1();
        } else {
            return super.handle(k);
```

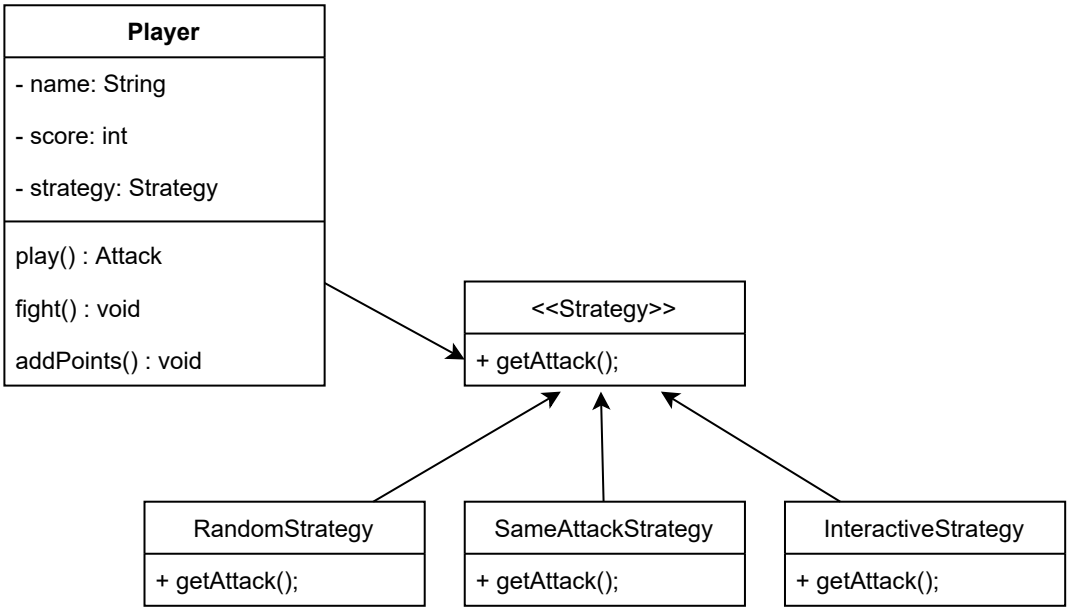
```
        }
    }

}
/************************************    StateNotOk
*****************************/
public class StateNotOK extends State {
}

// et les autres...

/*************************************************************************

// implémentons notre Client, ici un Digicode qui va utiliser le
pattern d'état.

public class Digicode {

    private State currentState;

    private static Scanner sc;

    public Digicode() {
        currentState = new State0();
    }

    public void push(char entry) {
        currentState = currentState.handle(entry);
    }

    public static void main(String[] args) {
        Digicode d = new Digicode();
        sc = new Scanner(System.in);
        while (true) {
            String s = sc.next();
            if (s.length() == 1
                    && (s.charAt(0) == 'a' || s.charAt(0) == 'b'
                            || s.charAt(0) == 'c' || s.charAt(0) ==
'd'))
                d.push(s.charAt(0));
            else
```

```
                    break;
                }
            }
        }
```

- Digicode définit l'inteface qu'utilise le client et gère une instance d'une sous-classe State"X" qui définit l'état actuel.
- State définit une interface qui encapsule le comportement associé à un état particulier du Digicode.
- State0, State1, etc... implémentent un comportement associé à un état de Digicode.

Stratégie

La stratégie permet de définir une famille d'algorithmes, d'encapsuler chacun d'entre eux et de les rendre interchangeables, d'utiliser différents algorithmes identiques sur le plan conceptuel en fonction du contexte et de permettre à ces algorothmes d'évoluer indépendamment des clients qui les utilisent. Comment ? En encaspulant les algorithmes dans une hierarchie et lier la hierarchie à l'objet en appelant par composition.




```
// crééons notre Strategy, une interface
public interface Strategy {

    Attack getAttack();

}
/**********************************************************************
```

```java
// implémentons quelques stratégies
public class RandomStrategy implements Strategy {

    @Override
    public Attack getAttack() {
        return Attack.values()[new Random().nextInt(3)];
    }

}

public class SameAttackStrategy implements Strategy {

    private Attack attack;

    public SameAttackStrategy(Attack attack) {
        this.attack = attack;
    }

    @Override
    public Attack getAttack() {
        return this.attack;
    }

}

public class InteractiveStrategy implements Strategy {

    private static Scanner sc = new Scanner(System.in);

    @Override
    public Attack getAttack() {
        int choice;
        do {
            System.out.println("Que voulez-vous faire ?");
            for (int i = 0; i < Attack.values().length; i++)
                System.out.println((i+1) + ". " + Attack.values()[i]);
            choice = sc.nextInt();
        } while (choice < 1 || choice > Attack.values().length);
        return Attack.values()[choice - 1];
    }
```

```java
}
/*********************************************************************

// implémentons un joueur qui va utiliser ces stratégies
public class Player {


    private String name;

    private int score;

    private Strategy strategy;

    public Player(String name, Strategy strategy) {
        this.name = name;
        this.strategy = strategy;
        this.score = 0;
    }

    public String getName() {
        return this.name;
    }

    public int getScore() {
        return this.score;
    }

    public void addPoints(int points) {
        this.score += points;
    }

    public Attack play() {
        return this.strategy.getAttack();
    }

    public String toString() {
        return this.getName() + " : " + this.getScore() + " point"
                + (this.getScore() > 1 ? "s" : "");
    }

    private static void fight(Player p1, Player p2, int nbTurns) {
```

```
        for (int i = 0; i < nbTurns; i++) {
            Attack a1 = p1.play();
            System.out.println(p1.getName() + " fait " + a1);
            Attack a2 = p2.play();
            System.out.println(p2.getName() + " fait " + a2);
            if (a1.compare(a2) > 0) {
                p1.addPoints(1);

                // System.out.println(p1);
            } else if (a1.compareTo(a2) < 0) {
                p2.addPoints(1);
                // System.out.println(p2);
            }
        }
        System.out.println(">>> SCORES <<<");
        System.out.println(p1);
        System.out.println(p2);
    }

    public static void main(String[] args) {
        fight(new Player("Joueur 1", new InteractiveStrategy()), new
Player(
                "Joueur 2", new RandomStrategy()), 10);
    }
}
```

- Strategy déclare une interface commune à tous les algorithmes et est utilisée par Player pour appeler l'algorithme défini par une stratégie complète.
- InteractiveStrategy, RandomStrategy et SameAttackStrategy implémentent l'algorithme en utilisant l'interface Strategy.
- Player choisit une référence à un objet de stratégie complète et peut définir une interface qui permet à cet objet d'accéder à ses données