DESIGN PATTERNS

DES SOLUTIONS A DES PROBLEMES COMMUNS

Présenté par Guilhem Beillard

Plan

Planning

La bible

Qu'est ce qu'un design pattern ?

Historique

Pourquoi devrais-je apprendre les pattern design ?

Classification

Critique des design patterns

Ecrire du code simple

SOLID

Etude de 5 cas

Et en functionnal programming ?

Planning

Raffraichissement de la théorie

max 4h

Projet supervisé

max 4h

Debrief

max 1h

Fin du projet a la maison

Pour ceux qui n'ont pas fini

La bible 👇

https://refactoring.guru/fr/design-patterns/

Qu'est ce qu'un design pattern ?

solutions classiques

à des problèmes récurrents

Ce n’est pas une librairie

ce n’est pas un bout de code spécifique,
mais un concept général

C’est un plan

L’implémention est differente sur chaques projets

Historique

A Pattern Language: Towns, Buildings, Construction

Christopher Alexander dans les années 70

➡

Design Patterns – Elements of Reusable Object-Oriented Software

"Gang of Four", Erich Gamma, John Vlissides, Ralph Johnson, et Richard Helm en 1994

Pourquoi devrais-je apprendre les pattern design ?

solutions fiables et éprouvées

langage commun

Classification

Idiomes

basiques et de plus bas niveau

Patrons d’architecture

peuvent être utilisés pour concevoir la totalité de l’architecture d’une application

Groupes principaux de patrons

- Les Patrons de création
- Les Patrons structurels
- Les Patrons comportementaux

Critique des design patterns

Dogme

Attention a ne pas les appliquer partout sans réflechir

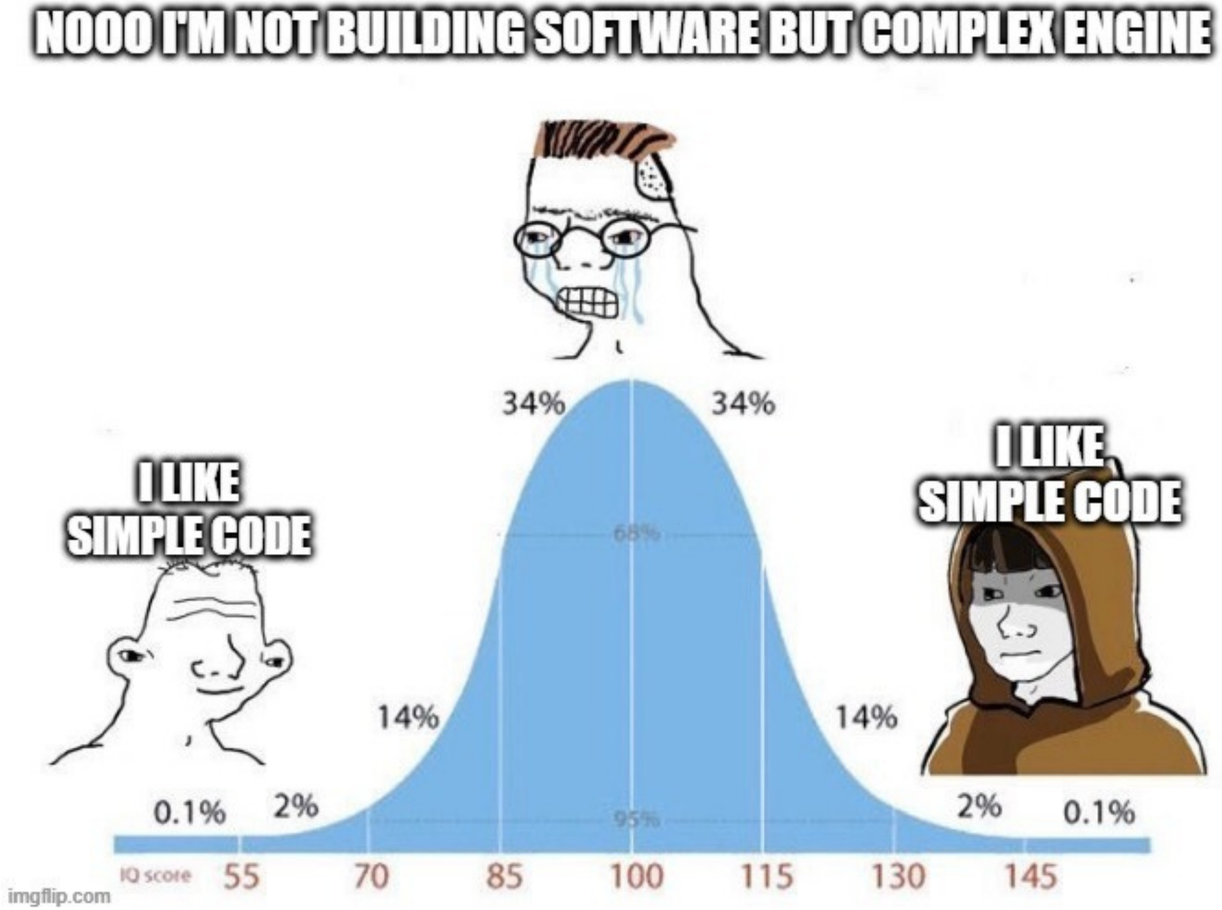
Stack

Parfois une lambda est plus simple

Utilisation injustifiée

Si tout ce que vous avez est un marteau, tout ressemble à un clou.

Ecrire du code simple



SOLID

Single-responsibility principle

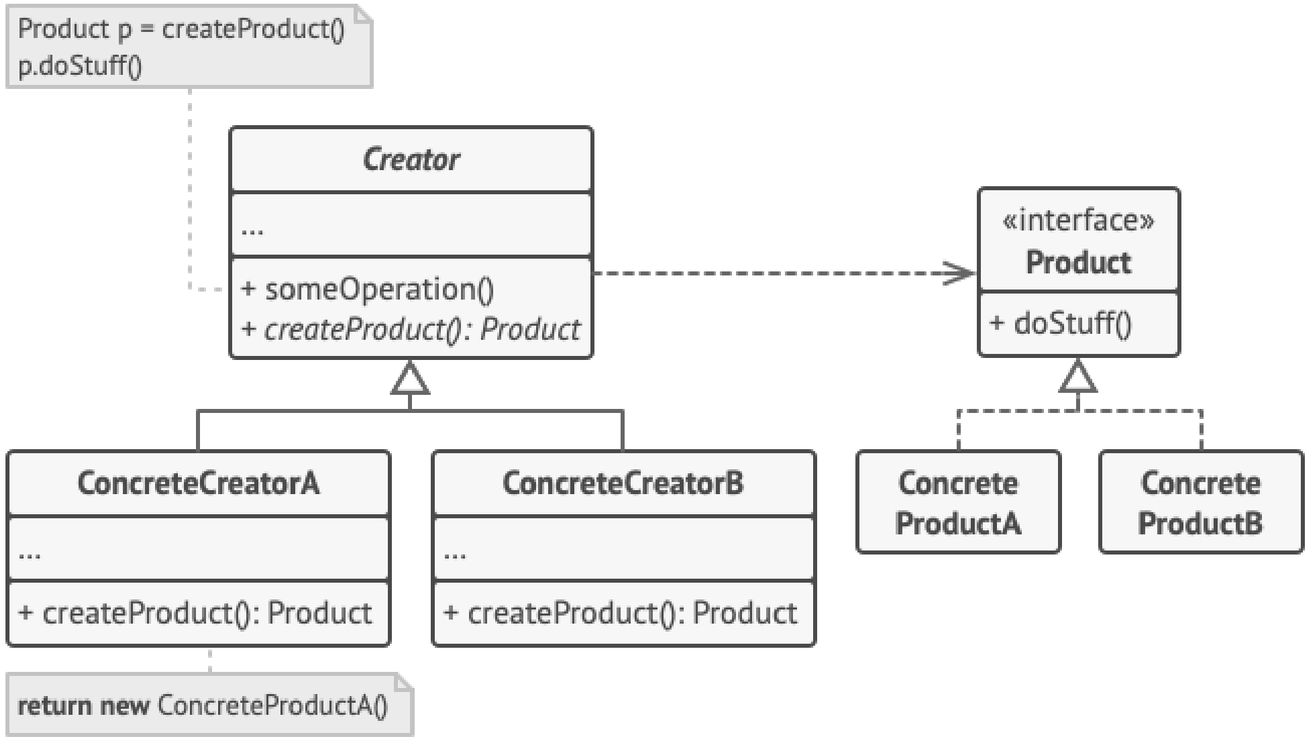
Open–closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Factory


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()

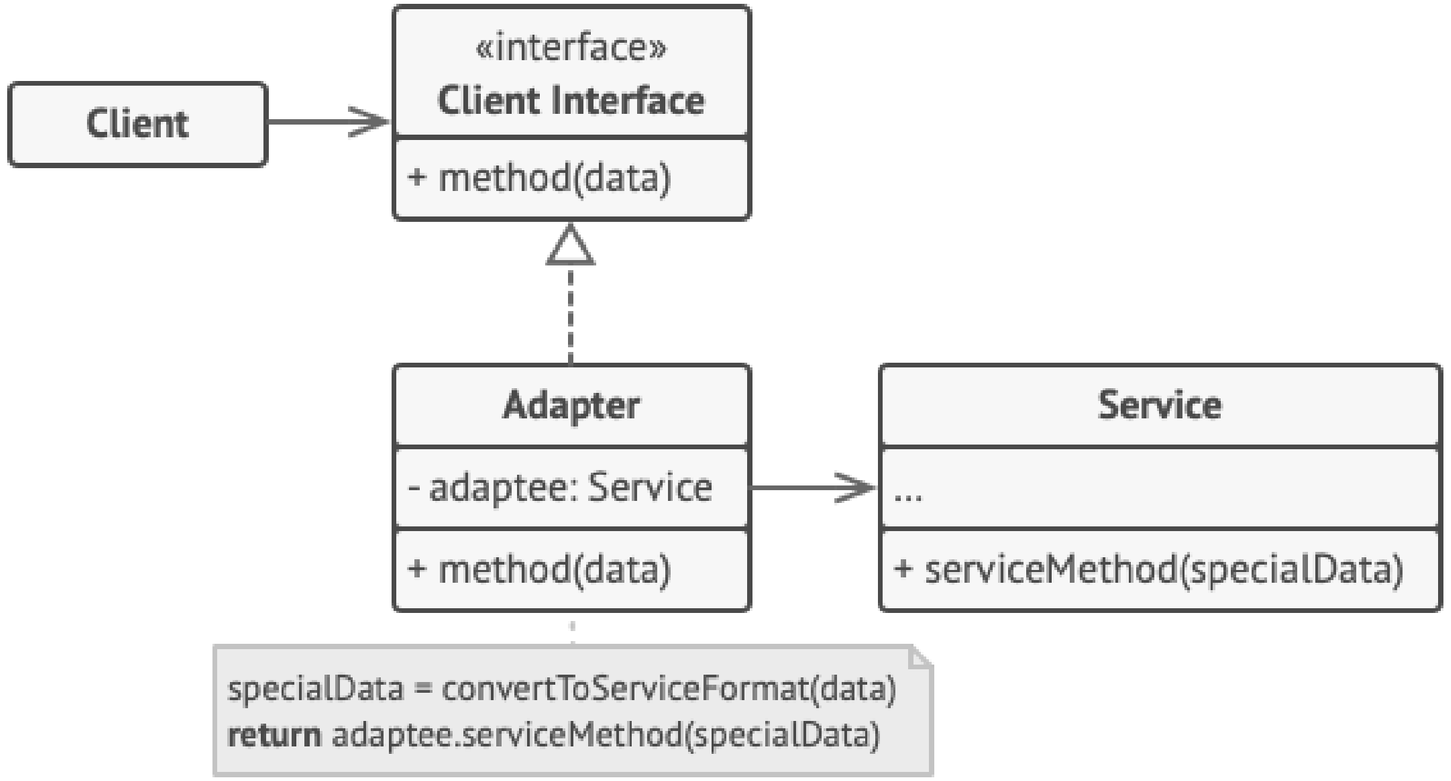
Factory

```ts
export interface Product {
    getName(): string;
    getPrice() : number;
    getStock(): number;
}

export class ProductA implements Product {
    getName(): string {
        return "Product A";
    }

    getPrice(): number {
       return 5.25;
    }

    getStock(): number {
        return 1;
    }
}

export class ProductB implements Product {
    getName(): string {
        return "Product B";
    }

    getPrice(): number {
       return 25;
    }

    getStock(): number {
        return 100;
    }
}
```

```ts
import {ProductA, ProductB} from "./product";

interface ProductFactory {
    createProduct();
}

class ProductAFactory implements ProductFactory {
    createProduct() {
        console.log("creating product A...");
        return new ProductA();
    }
}

class ProductBFactory implements ProductFactory {
    createProduct() {
        console.log("creating product B...");
        return new ProductB();
    }
}

export class Factory {
    private factories = {};

    constructor() {
        this.factories["ProductA"] = new ProductAFactory();
        this.factories["ProductB"] = new ProductBFactory();
    }

    public createProduct(productName: string) {
        return this.factories[productName].createProduct();
    }
}
```

```ts
import {Factory} from "./factory";

let f = new Factory();
const productA = f.createProduct("ProductA");

console.log("product:", productA.getName(), productA.getPrice(), productA.getStock());

const productB = f.createProduct("ProductB");

console.log("product:", productB.getName(), productB.getPrice(), productB.getStock());
```

Adapter


«interface»
Client Interface
+ method(data)
Client
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

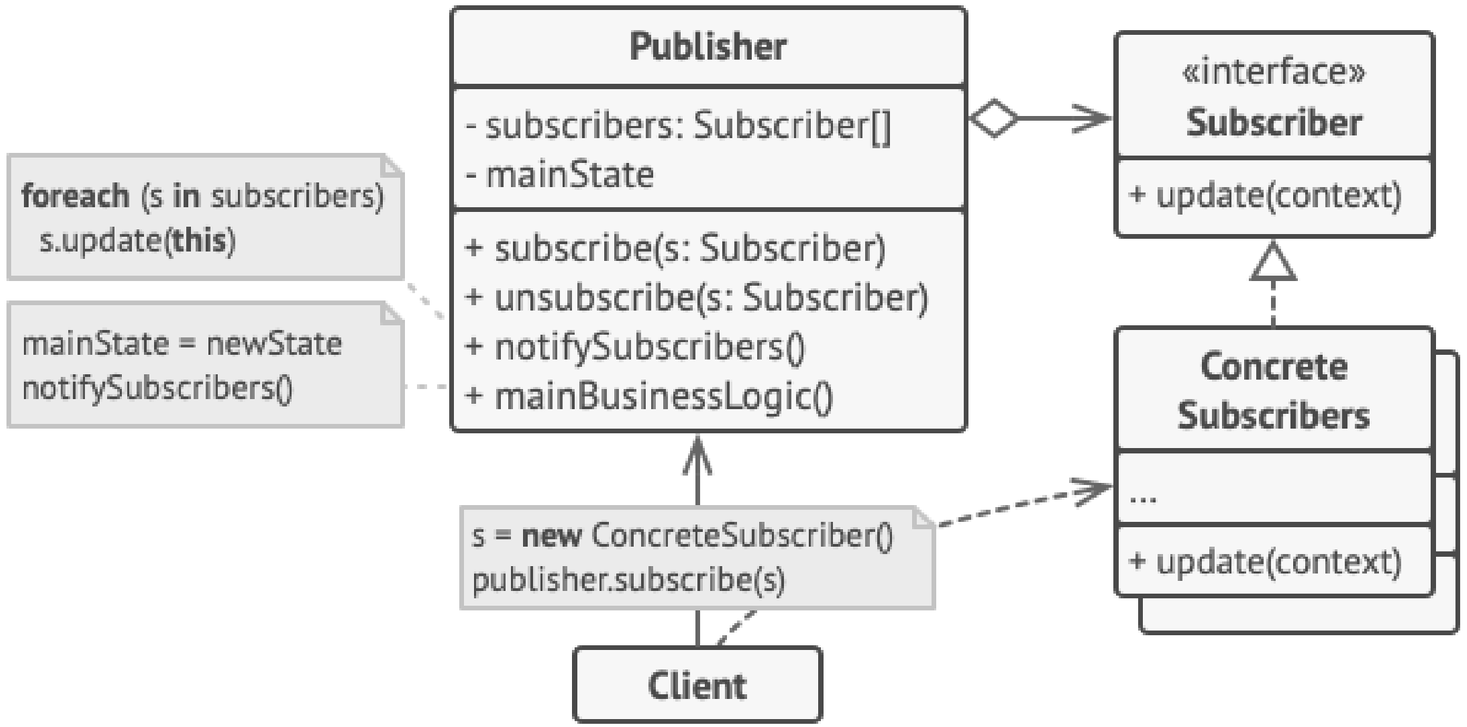
Adapter

```
class Target {
    public request(): string {
        return 'Target: The default target\'s behavior.';
    }
}


class Adaptee {
    public specificRequest(): string {
        return '.eetpadA eht fo roivaheb laicepS';
    }
}
```

```
class Adapter extends Target {
    private adaptee: Adaptee;

    constructor(adaptee: Adaptee) {
        super();
        this.adaptee = adaptee;
    }

    public request(): string {
        const result = this.adaptee.specificRequest().split('').reverse().join('');
        return `Adapter: (TRANSLATED) ${result}`;
    }
}
```

```
function clientCode(target: Target) {
    console.log(target.request());
}

console.log('Client: I can work just fine with the Target objects:');
const target = new Target();
clientCode(target);

console.log('');

const adaptee = new Adaptee();
console.log('Client: The Adaptee class has a weird interface. See, I don\'t underst
console.log(`Adaptee: ${adaptee.specificRequest()}`);

console.log('');

console.log('Client: But I can work with it via the Adapter:');
const adapter = new Adapter(adaptee);
clientCode(adapter);
```

observer


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
Concrete
Subscribers
...
+ update(context)
s = new ConcreteSubscriber()
publisher.subscribe(s)
Client

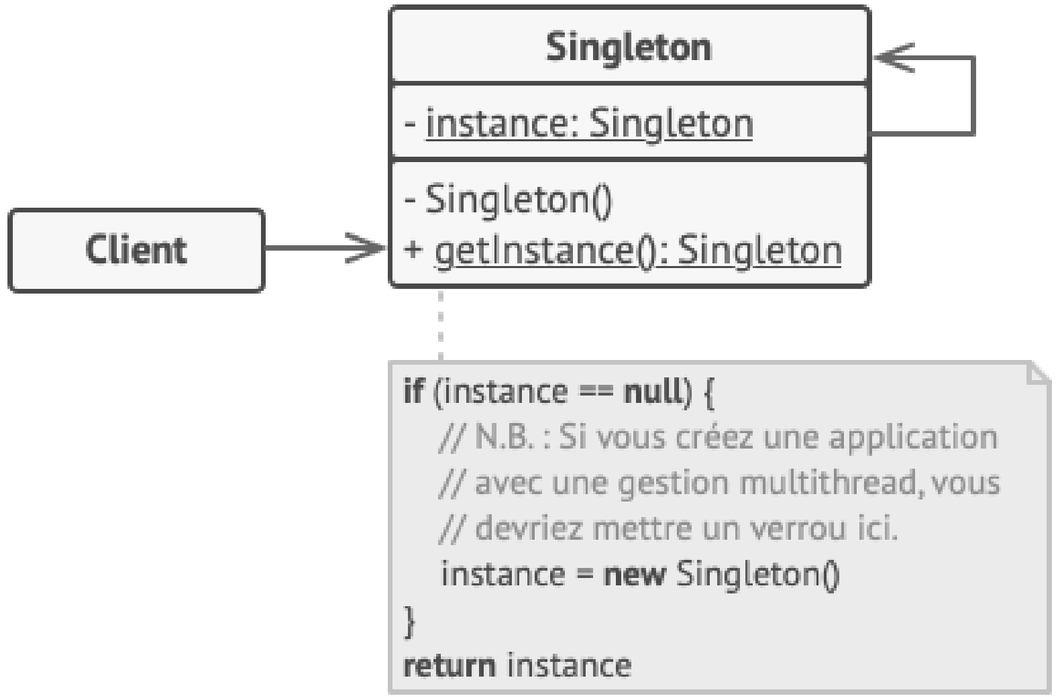
observer

```ts
interface Observer {
    update(event);
}

interface Observable {
    observers: Array<Observer>;
    notify(event);
    attach(observer : Observer)
    detach(observer: Observer)
}
```

```ts
export class Subscriber implements Observer {
    private name: string;

    constructor(name: string) {
        this.name = name;
    }
    update(event) {
        console.log(`Hello ${this.name}, you have unread news:  \n >${JSON.stringify(event)}`);
    }
}

export class Newspaper implements Observable {
    private news  = [];
    observers: Observer[] = [];

    addNews(news) {
        this.news.push(news);
        this.notify(news);
    }
    notify(event: any) {
        this.observers.map( (o) => {
            o.update(event);
        })
    }

    attach(observer: Observer) {
        this.observers.push(observer);
    }

    detach(observer: Observer) {
        throw new Error("Method not implemented =(.");
    }
}
```

```ts
import {Newspaper, Subscriber} from "./newspaper";
import {Auctioneer, Bidder} from "./bid";

console.log("Newspaper example...");
let newspaper = new Newspaper();
const subscriber = new Subscriber("Gabriel");

newspaper.attach(subscriber);
newspaper.addNews({title: "Aliens exist!", body: "asdasldkasld 1l2k3 1l2k3 ñ123ñ12j31231123"});


console.log("Bid example...");
let auctioneer = new Auctioneer();
let bidders = [new Bidder("gabriel", auctioneer), new Bidder("carlos", auctioneer)];

const rounds = 10;

for (let i = 0; i < rounds; ++i) {
    bidders[0].makeBid(Math.random()*100*i);
    bidders[1].makeBid(Math.random()*100*i);
}

auctioneer.closeSell();
```

singleton

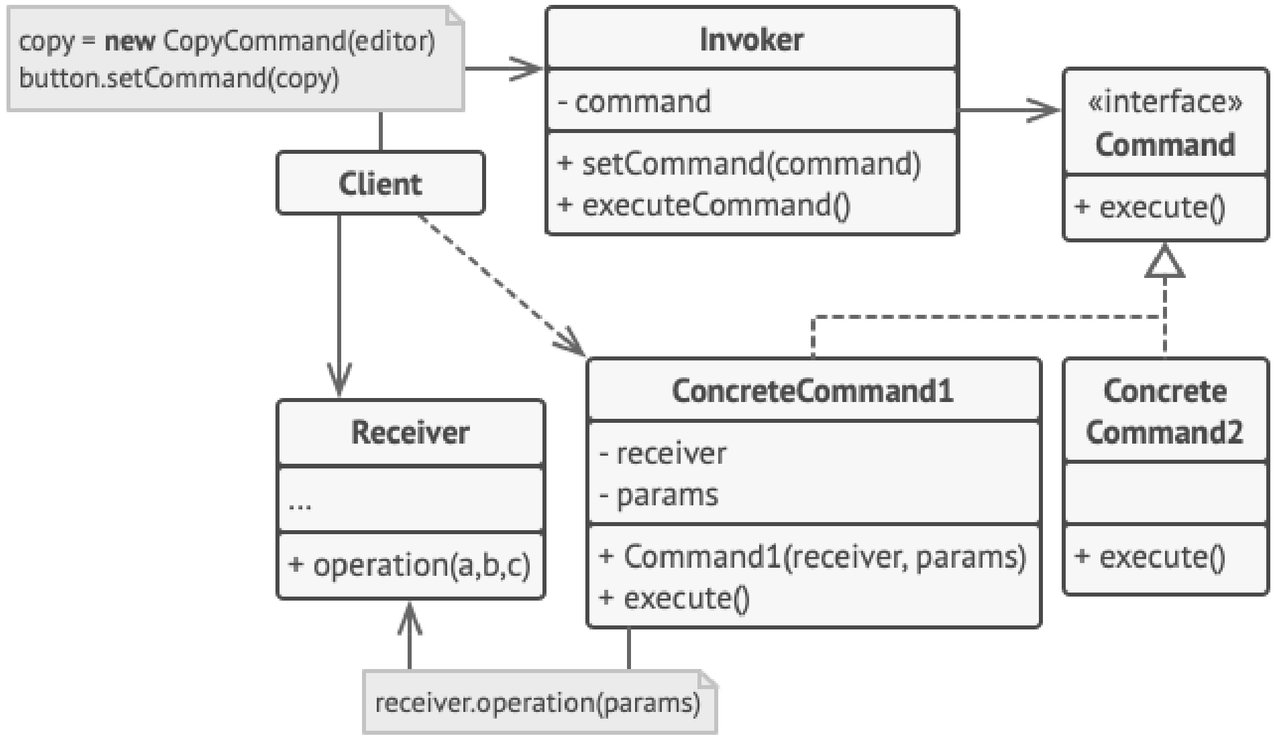

Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
if (instance == null) {
// N.B. : Si vous créez une application
// avec une gestion multithread, vous
// devriez mettre un verrou ici.
instance = new Singleton()
}
return instance

singleton

```
class Singleton {
    private static instance: Singleton;

    private constructor() { }

    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }

        return Singleton.instance;
    }

    public someBusinessLogic() {
        // ...
    }
}
```

```
function clientCode() {
    const s1 = Singleton.getInstance();
    const s2 = Singleton.getInstance();

    if (s1 === s2) {
        console.log('Singleton works, both variables contain the same instance.');
    } else {
        console.log('Singleton failed, variables contain different instances.');
    }
}

clientCode();
```

Command


copy = new CopyCommand(editor)
button.setCommand(copy)
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Client
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
receiver.operation(params)

Command

```
interface Command {
    execute();
}


export abstract class Joystick {
    buttons: {}

    doAction(action: string) {
        this.buttons[action].execute();
    }
}


export abstract class Button implements Command {
    abstract execute();
}
```

```
export default class NintendoJoystick extends Joystick {

    constructor() {
        super();
        this.buttons = {
            up: new NintendoUpButton(),
            down: new NintendoDownButton(),
            left: new NintendoLeftButton(),
            right: new NintendoRightButton(),
        };
    }

}


class NintendoUpButton extends Button {
    execute() {
        console.log("up pressed");
    }
}

class NintendoDownButton extends Button {
    execute() {
        console.log("down pressed");
    }
}

class NintendoLeftButton extends Button {
    execute() {
        console.log("left pressed");
    }
}


class NintendoRightButton extends Button {
    execute() {
        console.log("right pressed");
    }
}
```

```
import NintendoJoystick from "./nintendo";

function doKonamiCode(joystick) {
    console.log("Konami Code time!")
    const actions = [ "up", "up", "down", "down", "left", "right", "left", "right", "B", "A"];
    actions.map( (a) => joystick.doAction(a));
    console.log("Unlocked all levels!");
}

const joystick = new NintendoJoystick();
doKonamiCode(joystick);
```

Factory (FP)

```
const behavior1 = () => {
    console.log('do behavior 1');
  };

  const behavior2 = () => {
    console.log('do behavior 2');
  };

  const factory = (condition) => {
    /*
    Do other stuff
    */

    if (condition) behavior1;

    return behavior2;
  }
```

decorator (FP)

```
const compose = (...fns) => x => fns.reduceRight ((v, f) => f(v), x);

function isBiggerThanThree(value) {
  return value > 3
}

function mapBoolToHumanOutput(value) {
  return value ? "yes": "no"
}

const biggerThanThreeAndMapOutput = compose(
  mapBoolToHumanOutput,
  isBiggerThanThree
)

biggerThanThreeAndMapOutput(3)
```

Strategy (FP)

```
const strategy1 = () => {
    console.log('run strategy 1');
};

const strategy2 = () => {
    console.log('run strategy 2');
};

const consumer = (runStrategy) => {
    /*
    Do other stuff
    */

    runStrategy();
}

const selectedStrategy = condition ? strategy1 : strategy2;

consumer(selectedStrategy);
```