DESIGN PATTERN

COMPOSITE

COMPOSITE

Sommaire

Notre situation

La théorie

Les limites

Live Coding

Kahoot

Bibliographie

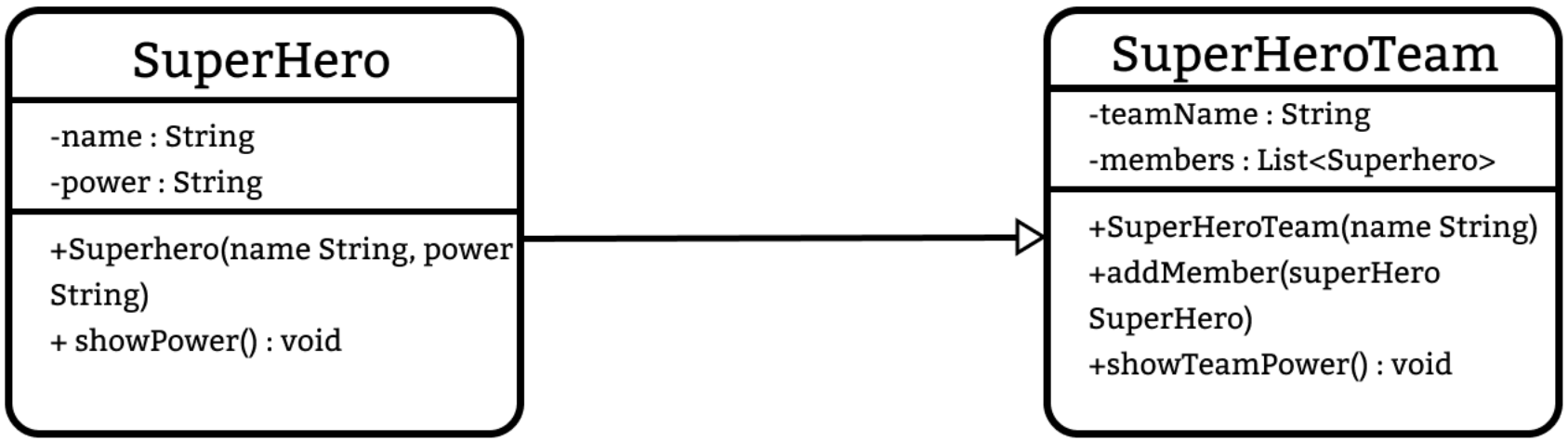
Épisode 1

The Data Avengers

Byte Man Captain Cache

COMPOSITE

SUPER - UML :



COMPOSITE

SUPER - IMPLEMENTATION :

```java
// Classe représentant un super-héros solo
class Superhero {
    private String name;
    private String power;

    public Superhero(String name, String power) {
        this.name = name;
        this.power = power;
    }

    public void showPower() {
        System.out.println(name + " utilise son pouvoir : " + power);
    }
}
```

COMPOSITE

SUPER - IMPLEMENTATION :

```java
// Classe représentant une équipe de super-héros
class SuperheroTeam {
    private String teamName;
    private List<Superhero> members = new ArrayList<>();

    public SuperheroTeam(String teamName) {
        this.teamName = teamName;
    }

    public void addMember(Superhero hero) {
        members.add(hero);
    }

    public void showTeamPowers() {
        System.out.println("L'équipe " + teamName + " utilise ses pouvoirs !");
        for (Superhero hero : members) {
            hero.showPower();
        }
    }
}
```

COMPOSITE

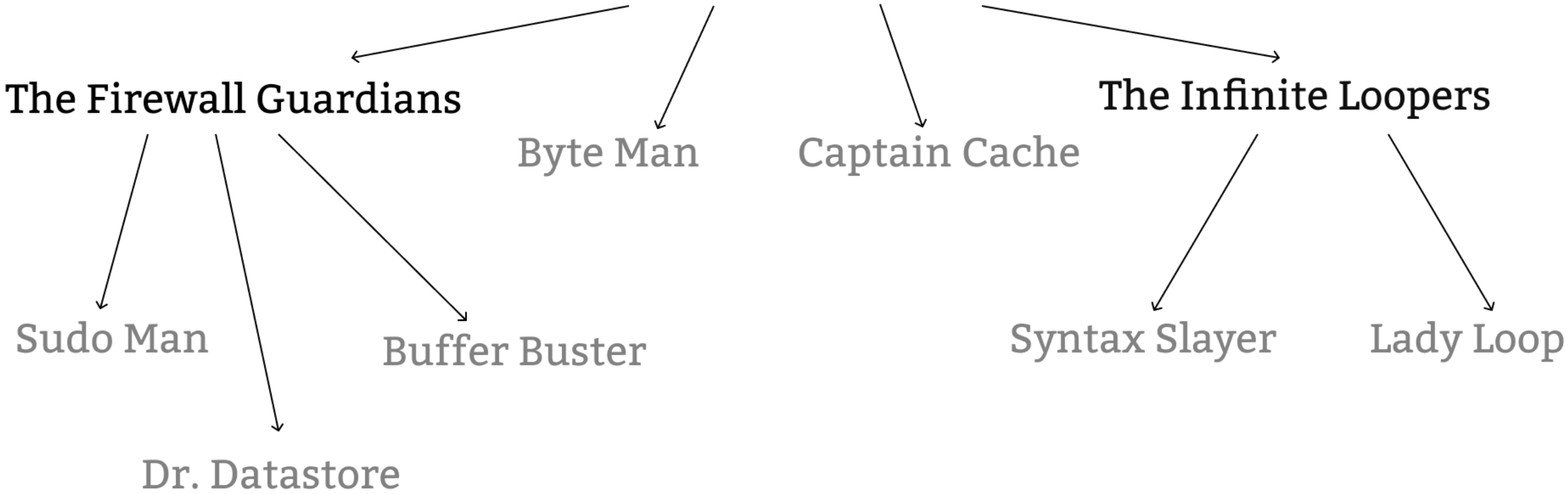
SUPER - IMPLEMENTATION :

```java
public class Main {
    public static void main(String[] args) {
        // Super-héros individuels
        Superhero byteMan = new Superhero("Byte Man", "0 or 1");
        Superhero captainCache = new Superhero("Captain Cache", "Vider le cache");

        // Équipe de super-héros
        SuperheroTeam vengers = new SuperheroTeam("Data Avengers");
        dataAvengers.addMember(ByteMan);
        dataAvengers.addMember(CaptainCache);

        // Afficher les pouvoirs individuels et de l'équipe
        captainCache.showPower();
        dataAvengers.showTeamPowers();
    }
}
```

COMPOSITE

Épisode 2

The Data Avengers



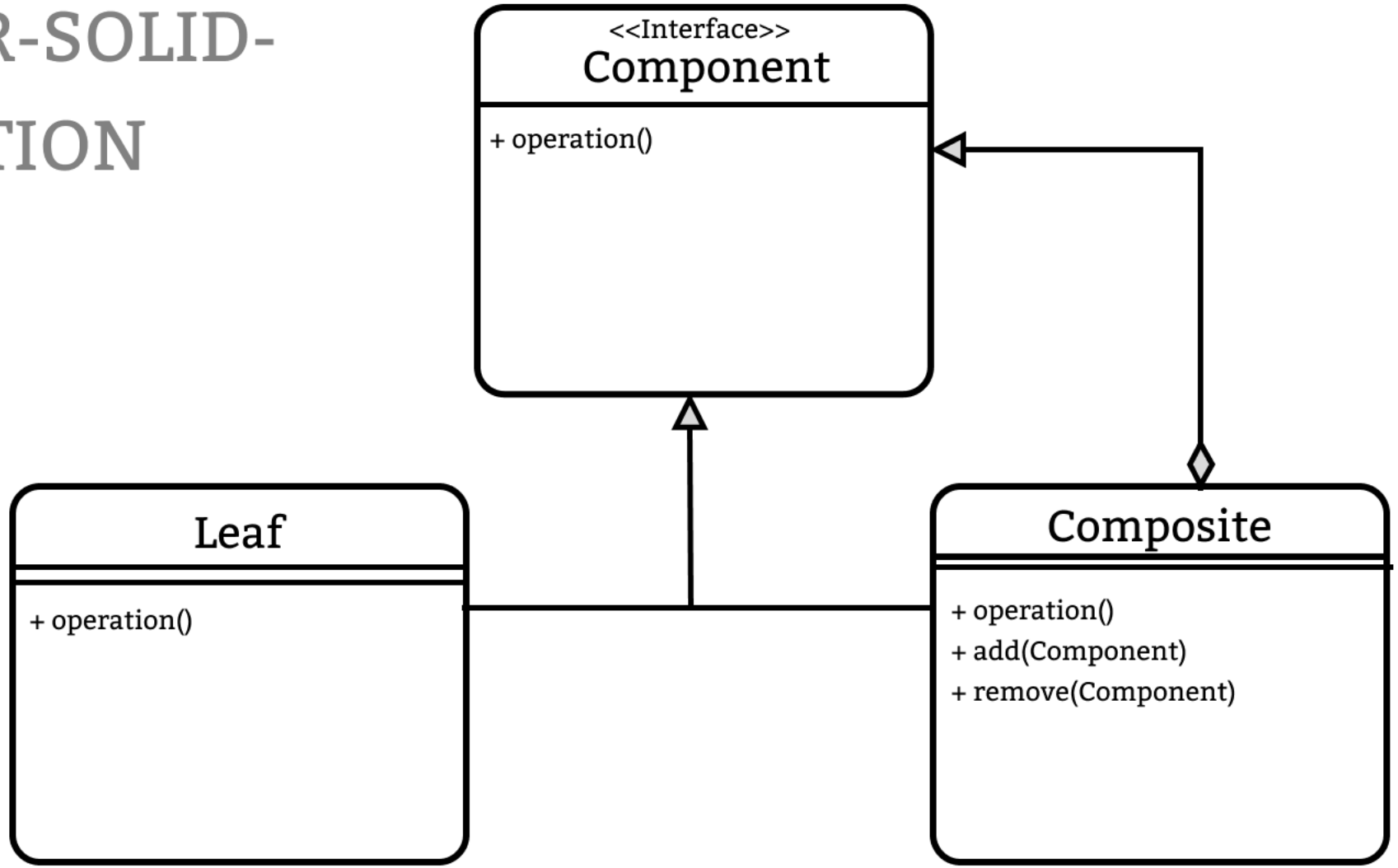
COMPOSITE

SUPER-PROBLEME

-Nos équipes ne peuvent pas contenir d’autres équipes

COMPOSITE

SUPER-SOLID-SOLUTION



COMPOSITE

Dans Notre Cas

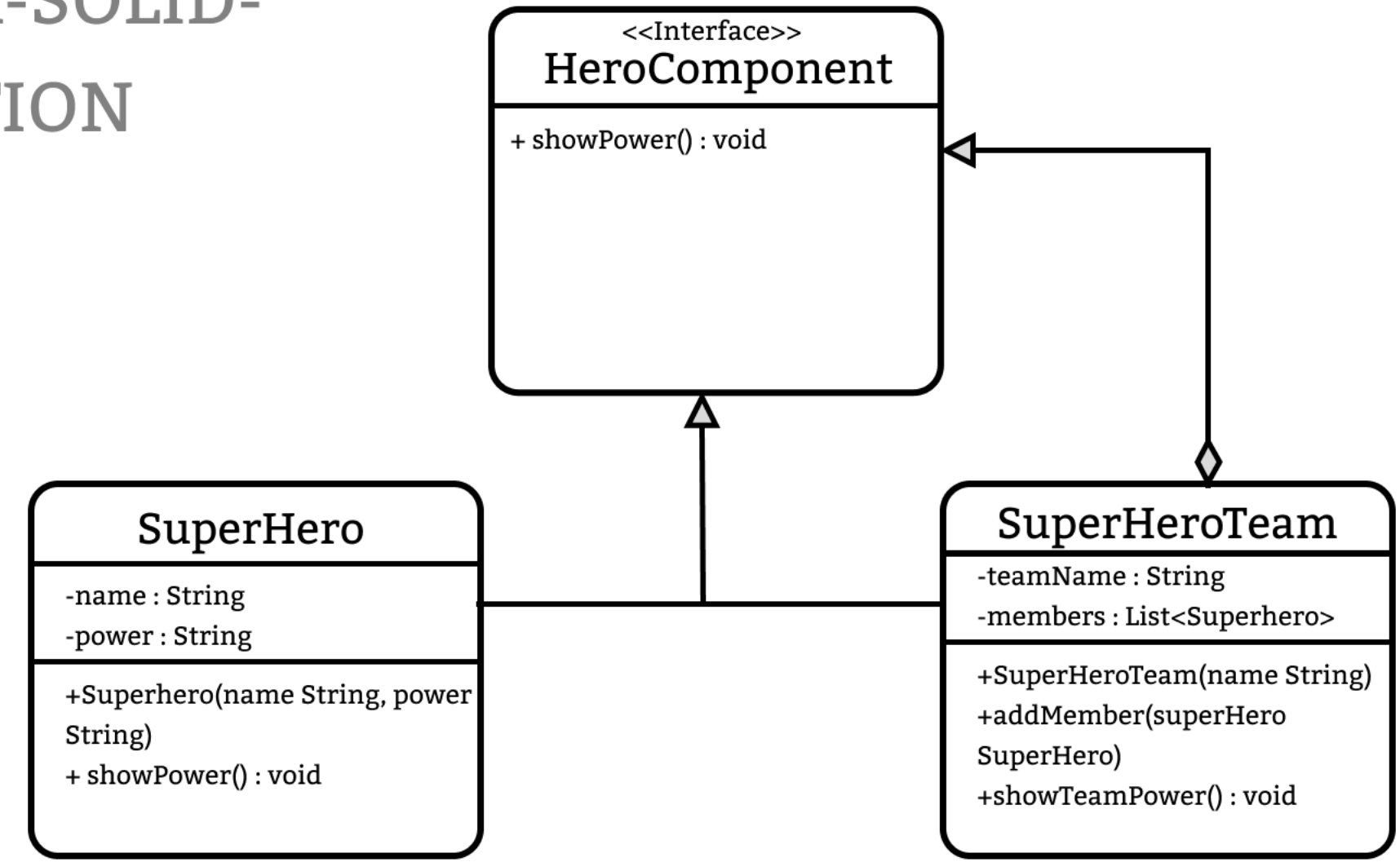
Leaf = SuperHero

Component = HeroComponent

Composite = SuperHeroTeam

COMPOSITE

SUPER-SOLID-SOLUTION



COMPOSITE

SUPER-SOLID-SOLUTION

```java
interface HeroComponent {
    void showPower();
}
```

```java
// Classe représentant un super-héros solo (Leaf)
class Superhero implements HeroComponent {
    private String name;
    private String power;

    public Superhero(String name, String power) {
        this.name = name;
        this.power = power;
    }

    @Override
    public void showPower() {
        System.out.println(name + " utilise son pouvoir : " + power);
    }
}
```

COMPOSITE

SUPER-SOLID-SOLUTION

```java
// Classe représentant une équipe de super-héros (Composite)
class SuperheroTeam implements HeroComponent {
    private String teamName;
    private List<HeroComponent> members = new ArrayList<>();

    public SuperheroTeam(String teamName) {
        this.teamName = teamName;
    }

    public void addMember(HeroComponent hero) {
        members.add(hero);
    }

    @Override
    public void showPower() {
        System.out.println("L'équipe " + teamName + " montre sa puissance collective !");
        for (HeroComponent hero : members) {
            hero.showPower();
        }
    }
}
```

COMPOSITE

SUPER-SOLID-SOLUTION

```java
public class Main {
    public static void main(String[] args) {
        // Super-héros individuels
        HeroComponent byteMan = new Superhero("Byte Man", "0 or 1");
        HeroComponent captainCache = new Superhero("CaptainCache", "Vider Le Cache");
        HeroComponent sudoMan = new Superhero("Sudo Man", "sudo /kill");
        HeroComponent drDatastore = new Superhero("Dr Datastore", "mkdir");
        HeroComponent bufferBuster = new Superhero("Buffer Buster", "DDOS");
        HeroComponent syntaxSlayer = new Superhero("Syntax Slayer", "Ajout des semi-colon ;");
        HeroComponent ladyLoop = new Superhero("Lady-Loop", "While True");
```

COMPOSITE

SUPER-SOLID-SOLUTION

```java
public class Main {
    public static void main(String[] args) {
        // Super-héros individuels
        HeroComponent byteMan = new Superhero("Byte Man", "0 or 1");
        HeroComponent captainCache = new Superhero("CaptainCache", "Vider Le Cache");
        HeroComponent sudoMan = new Superhero("Sudo Man", "sudo /kill");
        HeroComponent drDatastore = new Superhero("Dr Datastore", "mkdir");
        HeroComponent bufferBuster = new Superhero("Buffer Buster", "DDOS");
        HeroComponent syntaxSlayer = new Superhero("Syntax Slayer", "Ajout des semi-colon ;");
        HeroComponent ladyLoop = new Superhero("Lady-Loop", "While True");
```

COMPOSITE

SUPER-SOLID-SOLUTION

```
// Équipe de super-héros
SuperheroTeam firewallGuardians = new SuperheroTeam("Firewall Guardians");
firewallGuardians.addMember(sudoMan);
firewallGuardians.addMember(drDatastore);
firewallGuardians.addMember(bufferBuster);

// Équipe de super-héros
SuperheroTeam infiniteLoopers = new SuperheroTeam("Infinite Loopers");
infiniteLoopers.addMember(ladyLoop);
infiniteLoopers.addMember(syntaxSlayer);

SuperheroTeam dataAvengers = new SuperheroTeam("DataAvengers");
dataAvengers.addMember(byteMan);
dataAvengers.addMember(captainCache);
dataAvengers.addMember(firewallGuardians);
dataAvengers.addMember(infiniteLoopers);
```

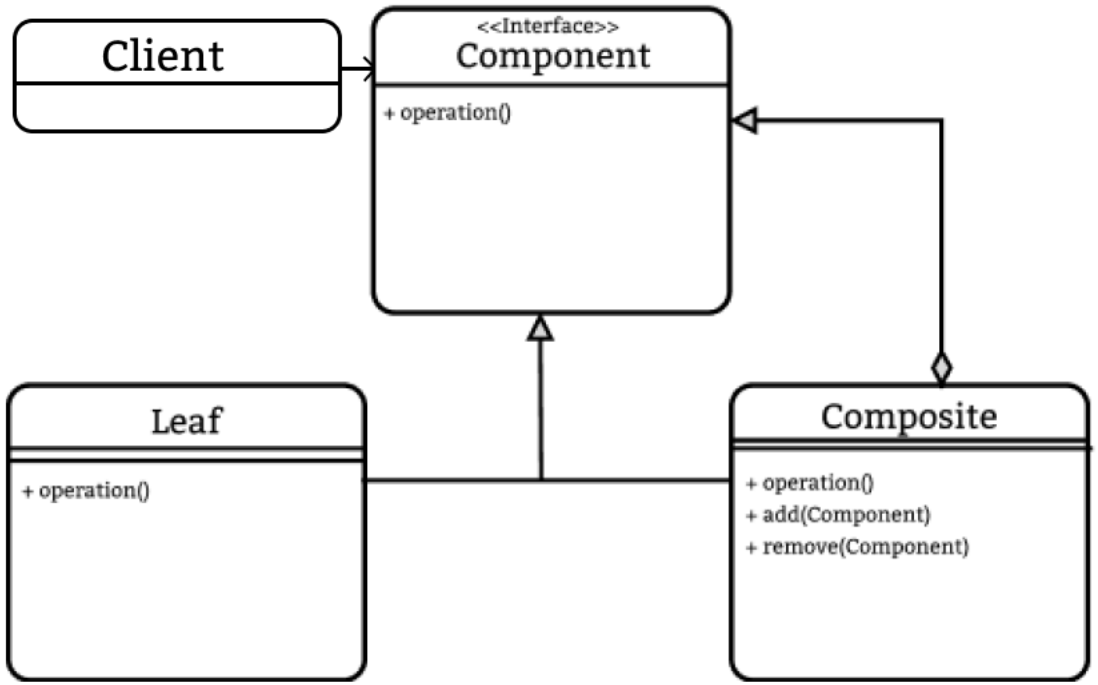
COMPOSITE

SUPER-SOLID-SOLUTION

```
L'équipe DataAvengers montre sa puissance collective !
Byte Man utilise son pouvoir : 0 or 1
CaptainCache utilise son pouvoir : Vider Le Cache

L'équipe FirewallGuardians montre sa puissance collective !
Sudo Man utilise son pouvoir : sudo /kill
Dr Datastore utilise son pouvoir : mkdir
Buffer Buster utilise son pouvoir : DDOS

L'équipe Infinite Loopers montre sa puissance collective !
Lady-Loop utilise son pouvoir : While True
Syntax Slayer utilise son pouvoir : Ajout des semi-colon ;
```

COMPOSITE

Composite : la théorie



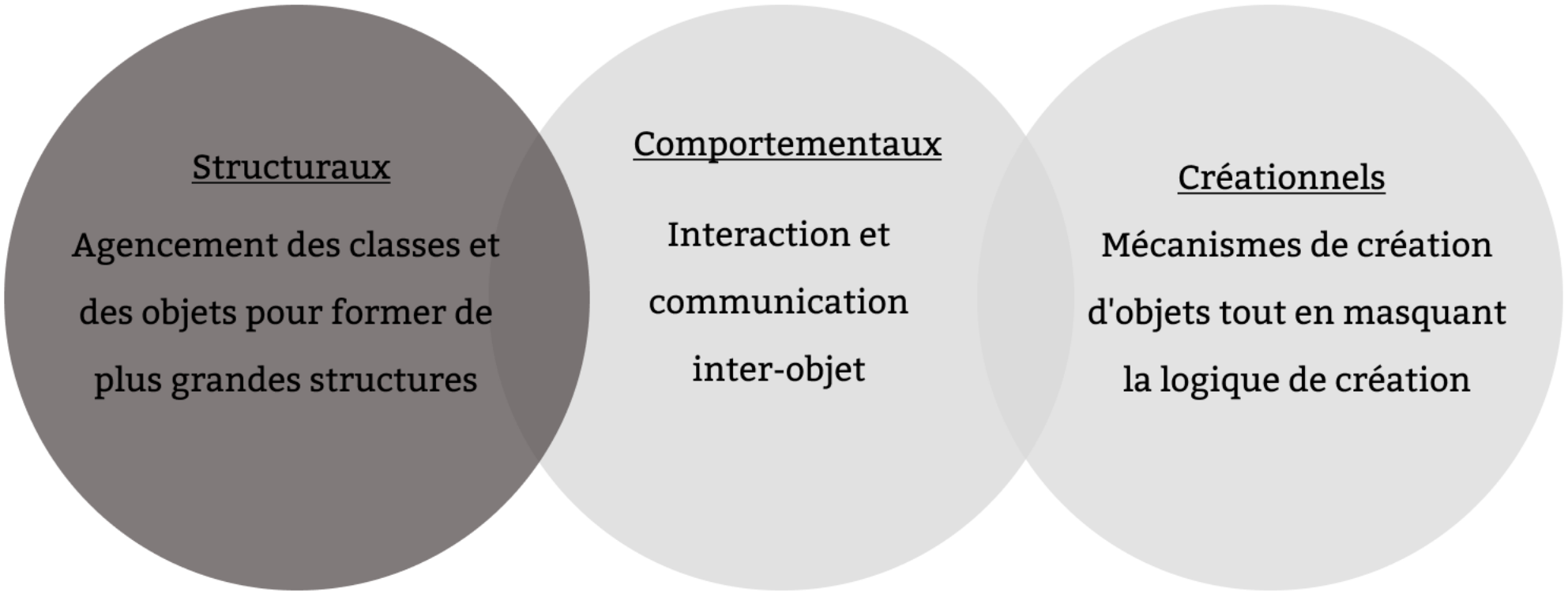
- Le pattern Composite est un patron de conception structurel qui permet de traiter de manière uniforme des objets individuels et des compositions d'objets.
- Il est particulièrement utile pour représenter des hiérarchies d'éléments qui peuvent être des composants simples (appelés "feuilles") ou des composants composites qui contiennent d'autres composants

*

COMPOSITE

Comparaison avec Decorator

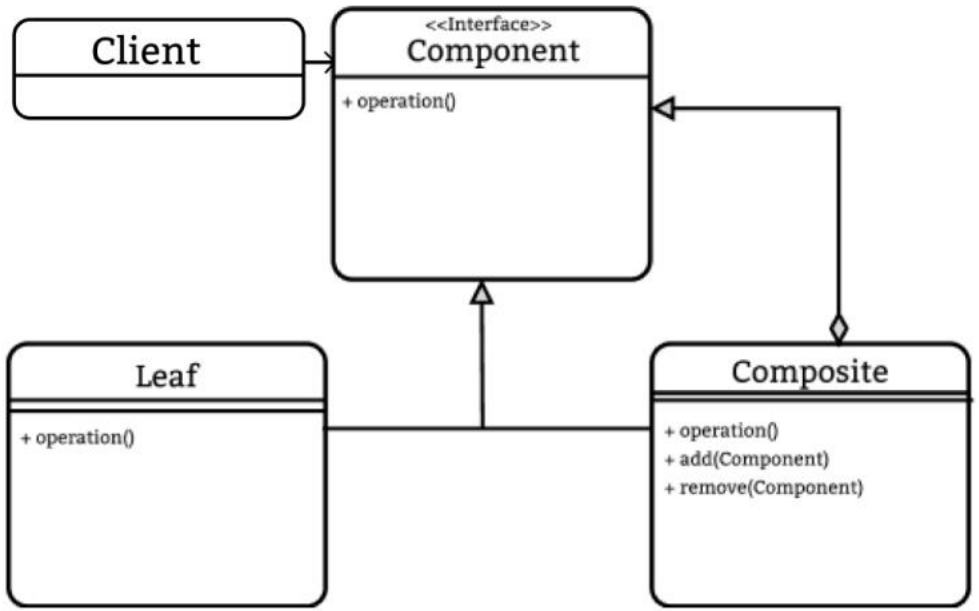
Rappel Du GoF

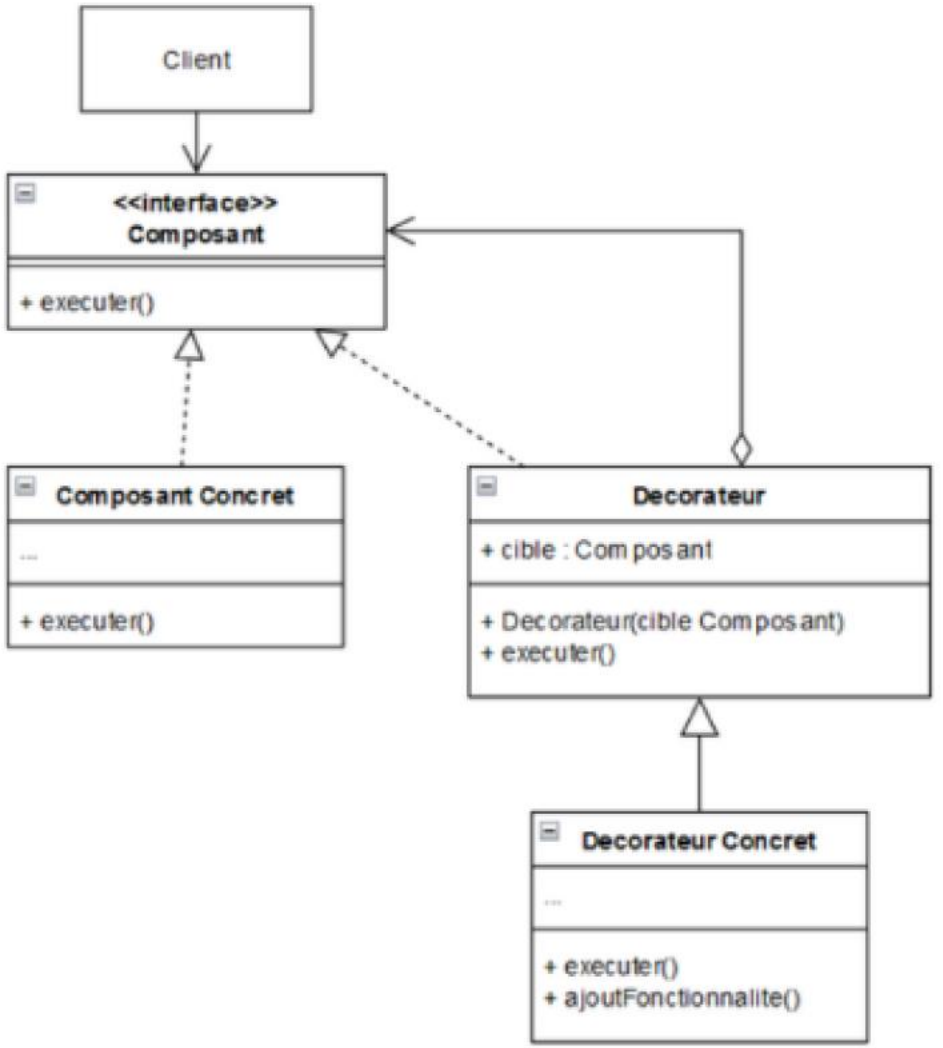


Gang Of Four

COMPOSITE

Comparaison avec Decorator





*

COMPOSITE

Les limites du pattern :

- Complexité accrue du code
- Difficulté à restreindre certains comportements
- Problèmes de performance
- Gestion des contraintes spécifiques
- Rigidité dans l'ajout de nouveaux éléments

*

COMPOSITE

Qu'en est t'il de SOLID

S - Single Responsibility Principle - composite gère ses enfant, leaf s'auto gère

O - Open/Closed Principle - Ajout de nouveau composant sans modification

L - Liskov Substitution Principle - Échange entre leaf et composite

I - Interface Segregation Principle - Partage d'interface = méthode imposée

D - Dependency Inversion Principle - Client interagit uniquement avec interface

*

COMPOSITE

Kahoot Time

*

Bibliographie

- https://refactoring.guru/
- https://www.ionos.fr/
- https://www.sfeir.dev/
- (ChatGPT peut être)