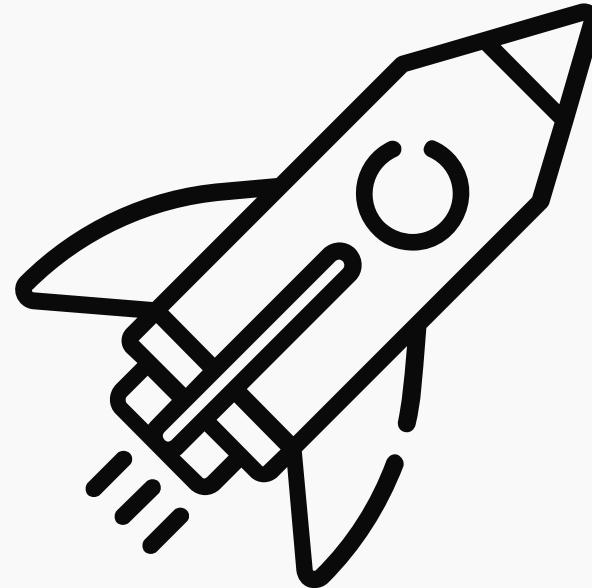


Introduction to Design Patterns



Our Team



Roberto Giménez
Fuentes

roberto.gimenez.19@ull.edu.es
linkedin.com/in/robertgim/



Darío Fajardo
Álvarez

dario.fajardo.31@ull.edu.es
linkedin.com/in/darío-fajardo/



Eric Ríos
Hamilton

eric.rios.41@ull.edu.es
linkedin.com/in/eric-ríos

Table of contents

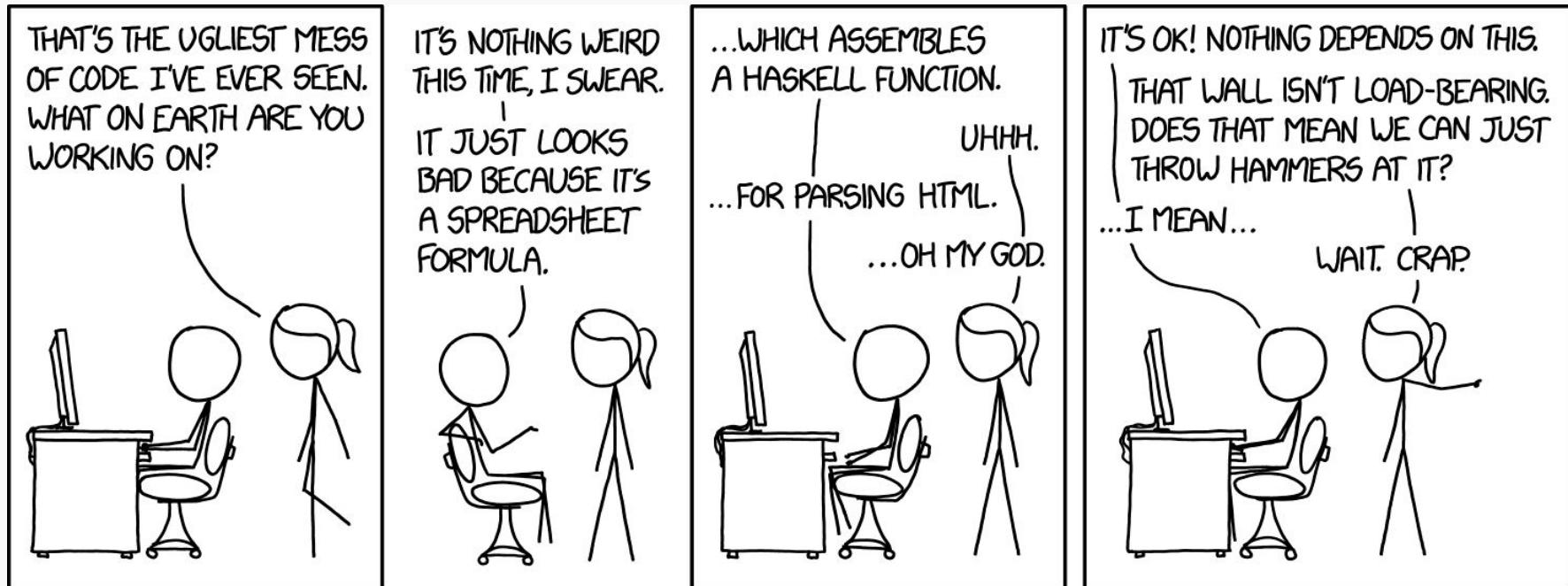
- | | | | |
|----|---------------------------------------|----|----------------|
| 01 | Why Do We
Need Design
Patterns? | 02 | Classification |
| 03 | Deep dive | 04 | Conclusion |
| 05 | References | | |
-

01

Why Do We Need
Design Patterns?

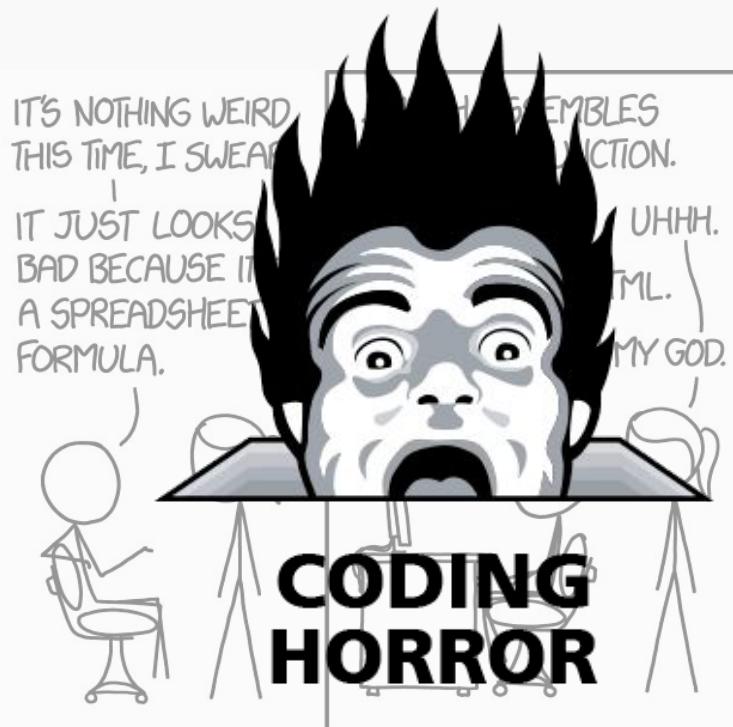


We want to avoid this



[\(xkcd: Bad Code\)](#)

We want to avoid this



[\(xkcd: Bad Code\)](#)

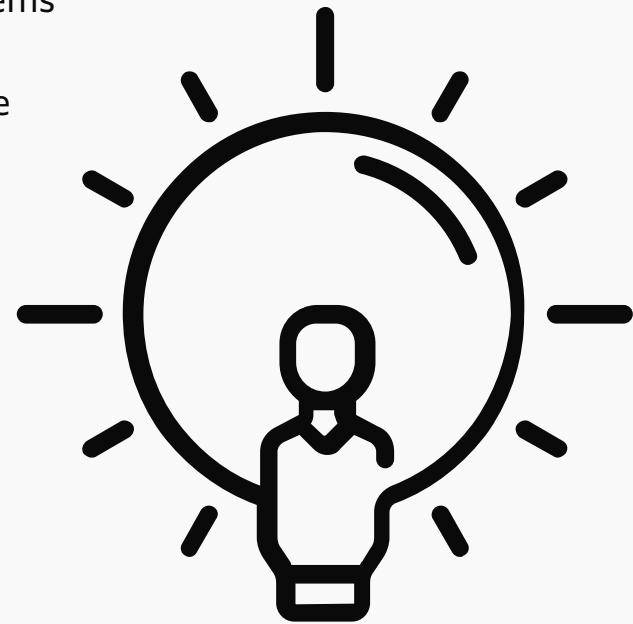
Why Do We Need Design Patterns?

- Software development is complex
- Without structure, code becomes hard to maintain
- Common problems appear repeatedly



Why Do We Need Design Patterns?

- Design Patterns are reusable solutions to common problems
- Provide proven best practices
- Help write code that is scalable, maintainable, and flexible

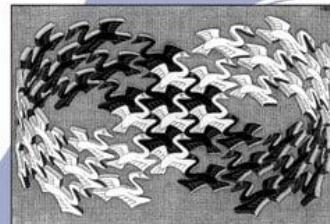


- (Almost) everything covered in these slides stems from this book.
- Recommended read if interested in more

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.

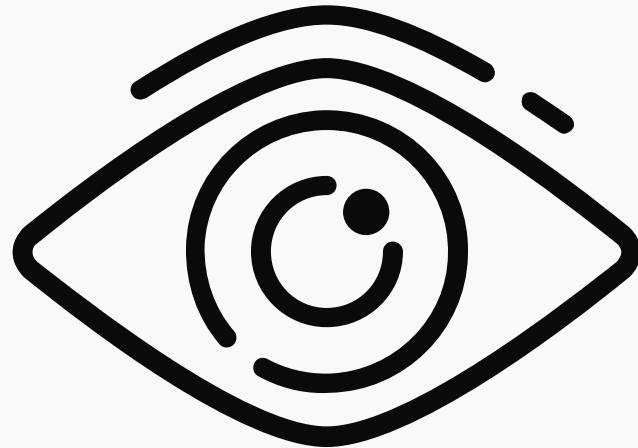
Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

02

Classification



Classification



Creational

Deal with object creation



Behavioural

Identify common behaviours between objects



Structural

Define relationships between objects

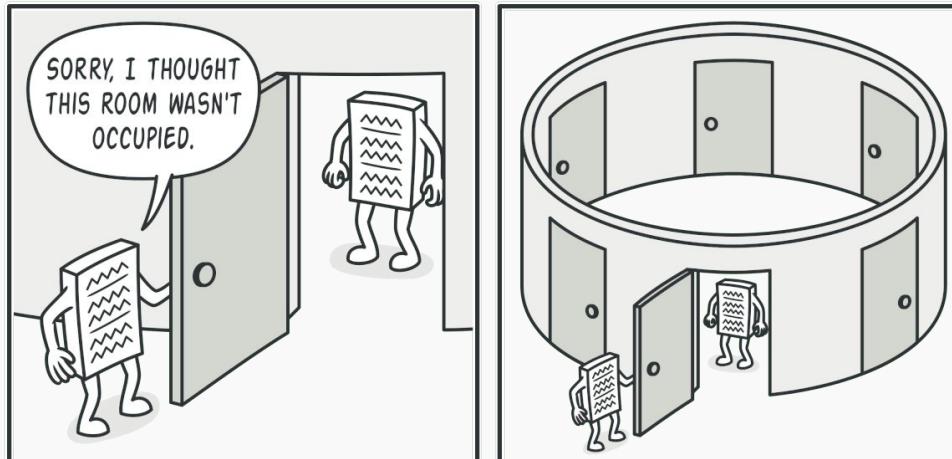
03

Deep dive

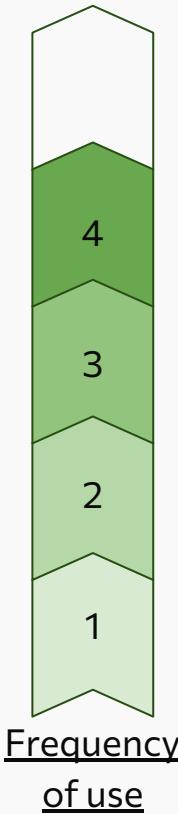


Singleton - Creational

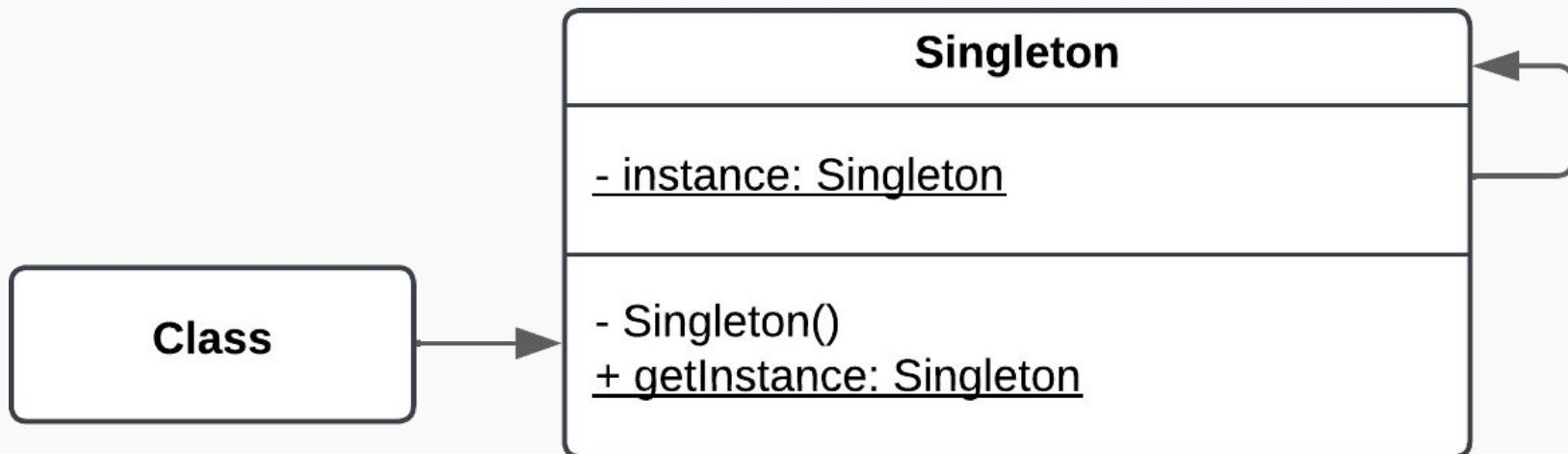
- Only one instance of the class
- Global access
- Violates Single responsibility principle



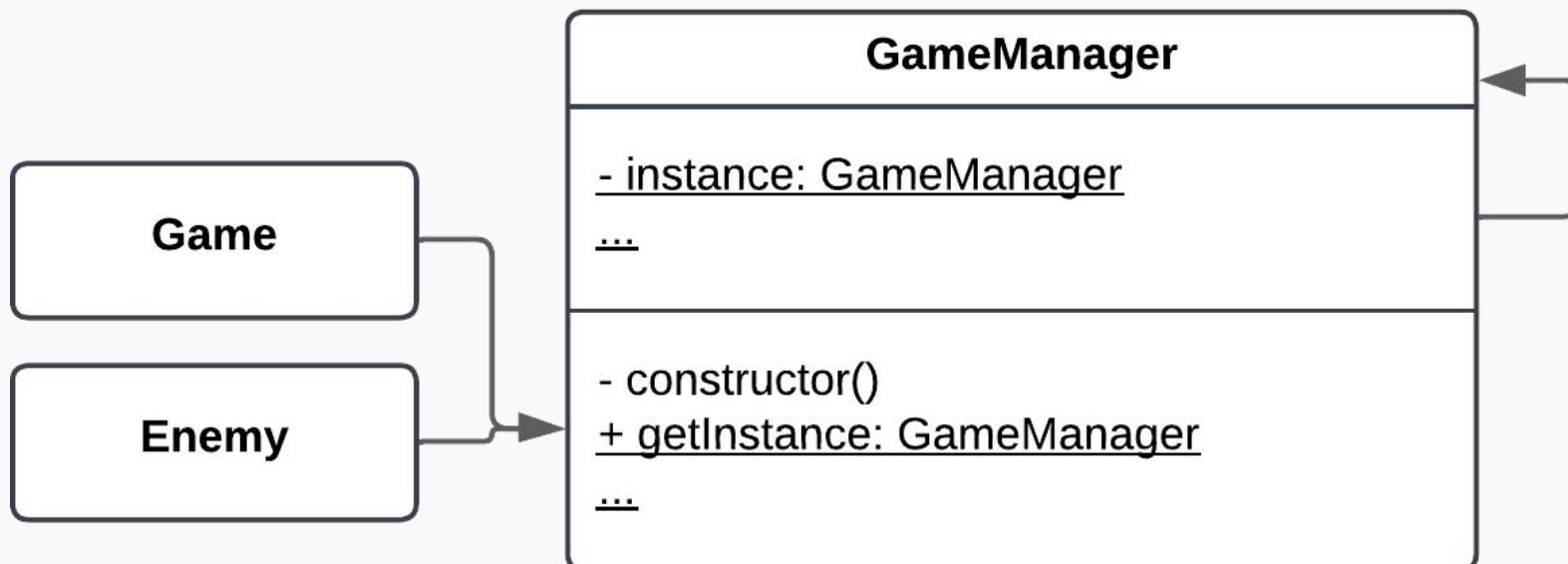
(Refactoring.Guru, 2025)



Class diagram - Generic



Class diagram - Example



Game manager Singleton

```
export class GameManager {  
    private static instance: GameManager;  
  
    private level: number;  
    private inventory: string[];  
    private points: number;  
  
    public static getInstance(): GameManager {  
        return GameManager.instance || (GameManager.instance = new GameManager());  
    }  
  
    private constructor() {  
        this.level = 1;  
        this.inventory = ['Sword', 'Potion'];  
        this.points = 0;  
    }  
    ...  
}
```

Singleton usage in game class

```
class Game {  
    // Access the GameManager singleton instance. If it doesn't exist, it will be created.  
    private gameManager = GameManager.getInstance();  
    ...  
    displayInventory(): void {  
        console.log('Your inventory:');  
        this.gameManager.getInventory().forEach((item) => {  
            console.log(item);  
        });  
    }  
    ...  
}
```

Singleton usage in enemy class

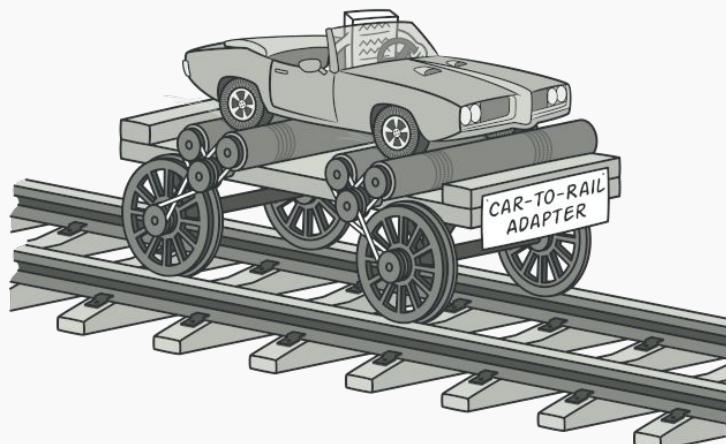
```
export class Enemy {  
    ...  
    defeat(): void {  
        // The gameManager is accessed here. If it  
        // was previously created, no new instance is created.  
        let gameManager = GameManager.getInstance();  
        gameManager.addPoints(this.pointValue);  
    }  
}
```

Singleton - Creational

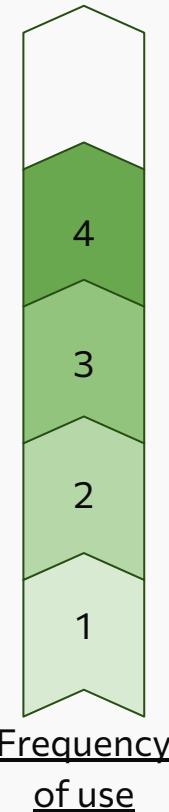
Advantages	Disadvantages
— Safe global access to data and functionality	— Violates Single responsibility principle
— Lazy initialization	— Difficult to unit test
— Assures that only one instance of the class exists	— Can turn into 'God' class
— Great for shared resources	— Tricky implementation in multithreaded environment

Adapter - Structural

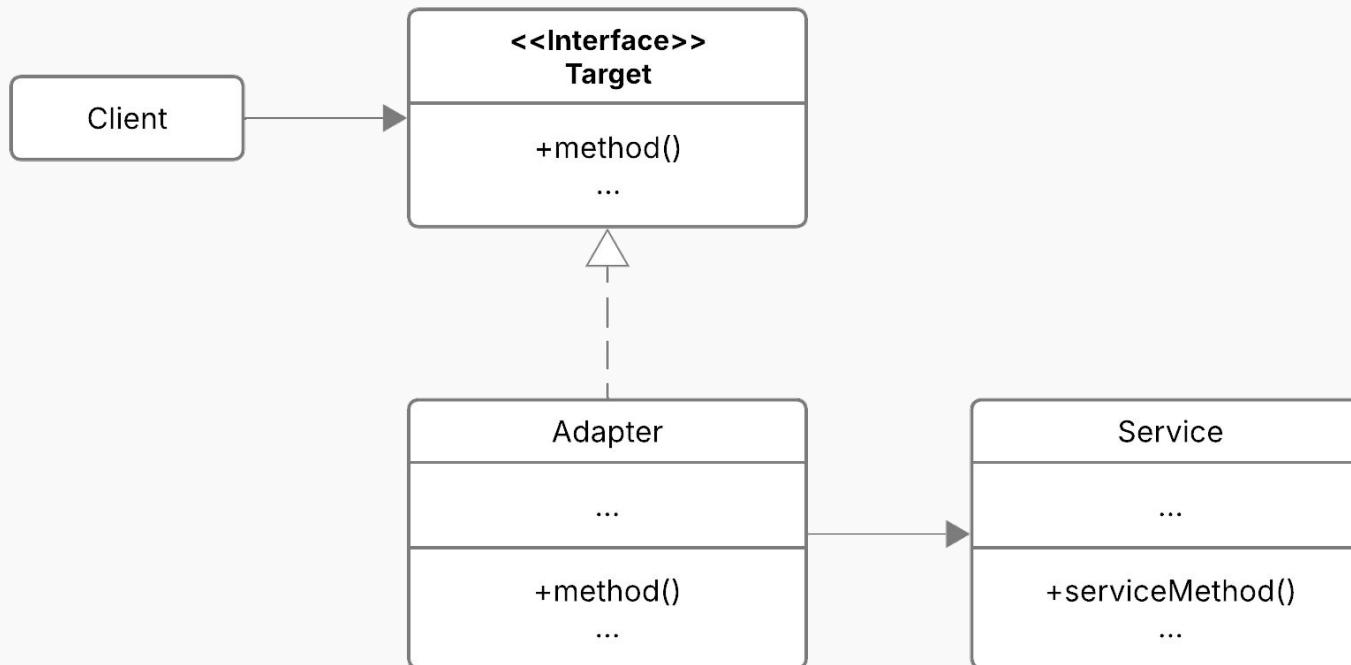
- Allows objects with incompatible interfaces to collaborate
- It uses an adapter object to translate between interfaces
- Provides an elegant solution but increases overall complexity of the code



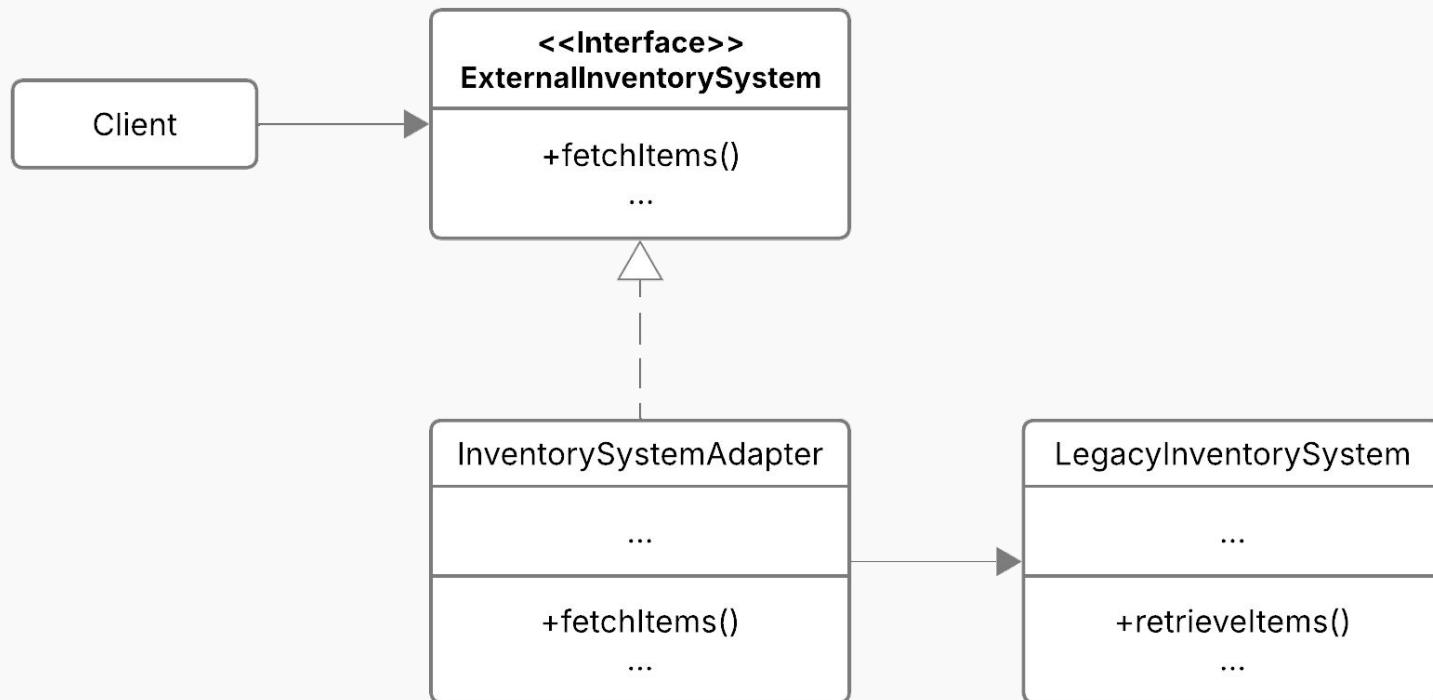
(Refactoring.Guru, 2025)



Class diagram - Generic



Class diagram - Example



External inventory system

```
interface ExternalInventorySystem {  
    fetchItems(): string[];  
}
```

Old inventory system

```
class LegacyInventorySystem {  
    retrieveItems(): string[] {  
        return ['Gun', 'Ammo'];  
    }  
}
```

Creating the adapter

```
export class InventorySystemAdapter implements ExternalInventorySystem {
    private legacySystem: LegacyInventorySystem;
    private itemMapping: Record<string, string>;

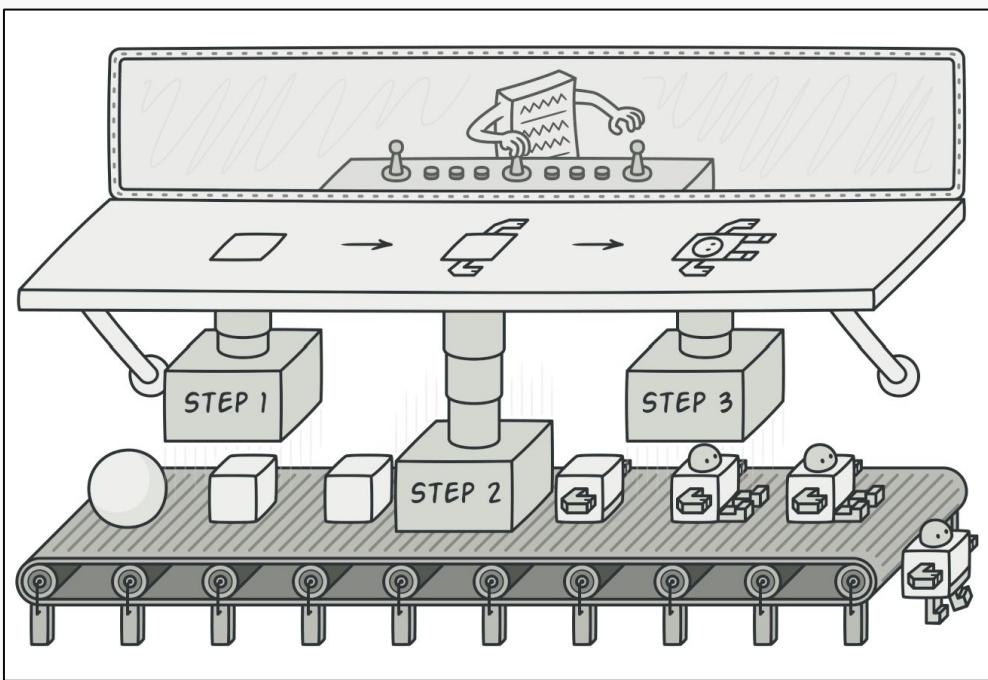
    constructor() {
        this.legacySystem = new LegacyInventorySystem();
        this.itemMapping = {
            'Gun': 'Sword',
            'Ammo': 'Potion'
        };
    }

    fetchItems(): string[] {
        return this.legacySystem.retrieveItems().map(item => this.itemMapping[item] || item);
    }
}
```

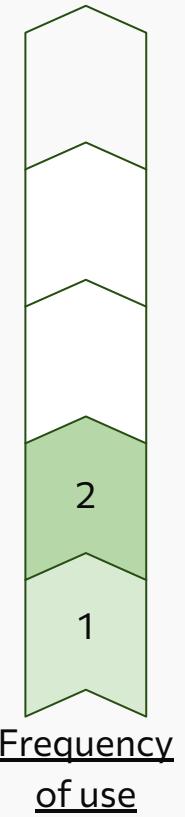
Adapter - Structural

Advantages	Disadvantages
— Easier integration	— Adds complexity to the code
— Reusable code	— Could worsen performance
— Flexibility	— Difficult maintainability
— Single responsibility	— Doesn't really solve design problems, just patches them up

Builder - Creational



(Refactoring.Guru, 2025)



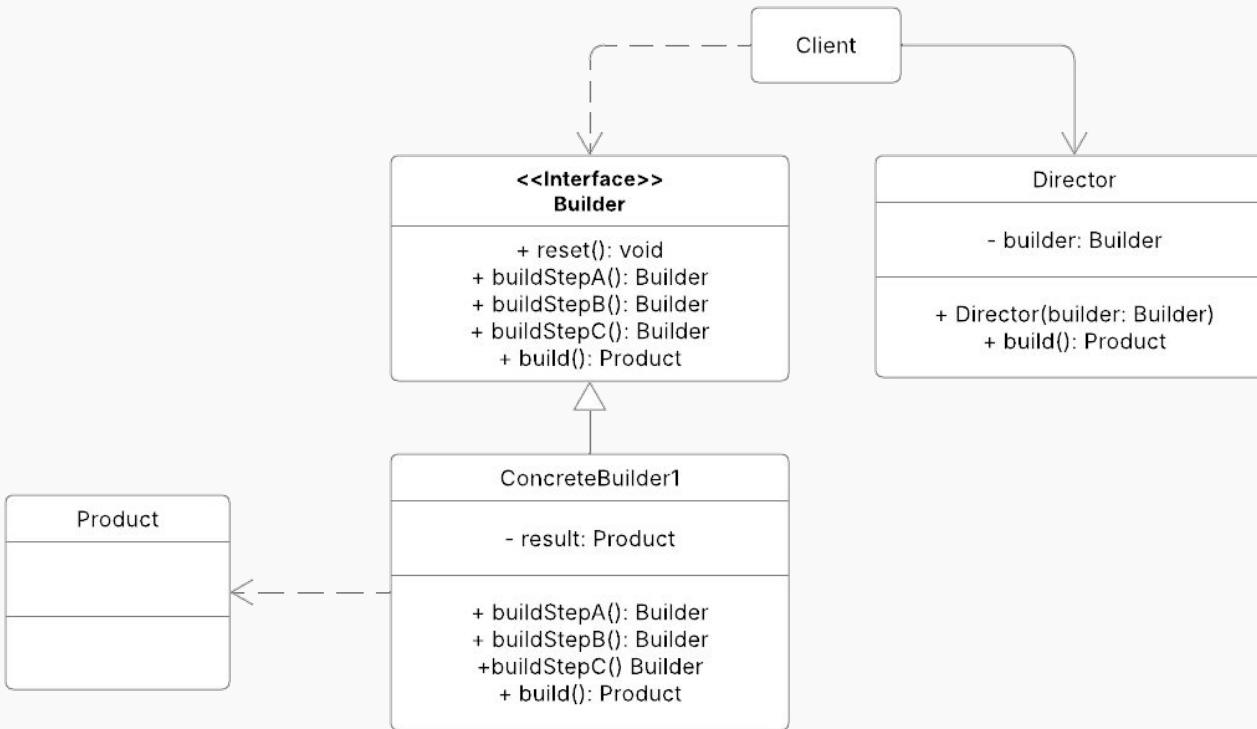
Creating a house

```
export class House {  
    public constructor (  
        private windows: number,  
        private rooms: number,  
        private doors: number,  
        private hasGarage: boolean,  
        private hasGarden: boolean,  
        private hasStatues: boolean) {}  
}
```

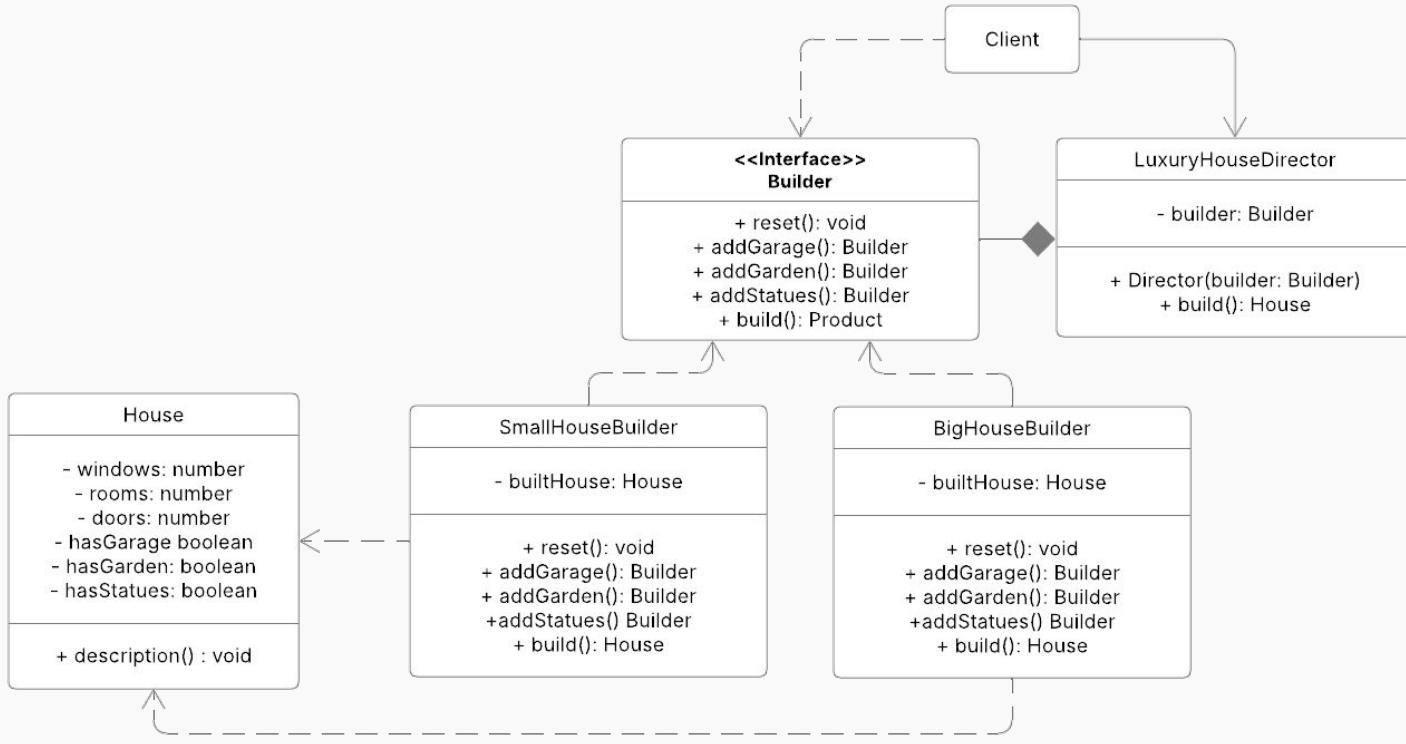
Large constructors are... complicated

```
const myBasicHouse = new House(4, 2, 4, false, false, false);
const myHouseWithGarage = new House(4, 2, 4, true, false, false);
const myLuxuryHouse = new House(4, 2, 4, true, true, true);
```

Class Diagram - Generic



Class Diagram - Specific



House builder interface

```
export interface Builder {  
    reset(): void;  
    builder(): House;  
    addGarage(): Builder;  
    addGarden(): Builder;  
    addStatues(): Builder;  
}
```

SmallHouseBuilder (1)

```
export class SmallHouseBuilder implements Builder {
    private hasGarden: boolean = false;
    private hasGarage: boolean = false;
    private hasStatues: boolean = false;
    private static readonly windows = 4;
    private static readonly rooms = 2;
    private static readonly doors = 4;

    public addGarage(): SmallHouseBuilder {
        this.hasGarage = true;
        return this;
    }
}
```

SmallHouseBuilder (2)

```
export class SmallHouseBuilder implements Builder {
    // ...
    public addGarden(): SmallHouseBuilder {
        this.hasGarden = true;
        return this;
    }

    public addStatues(): SmallHouseBuider {
        this.hasStatues = true;
        return this;
    }

    public build(): House {
        return new House(
            // All parameters...
        );
    }
}
```

LuxuryHouseDirector

```
export class LuxuryHouseBuilder {
    public constructor(private builder: Builder) {}

    public build(): House {
        this.builder.addGarage().addGarden().addStatues();
        return this.builder.build();
    }
}
```

Usage example

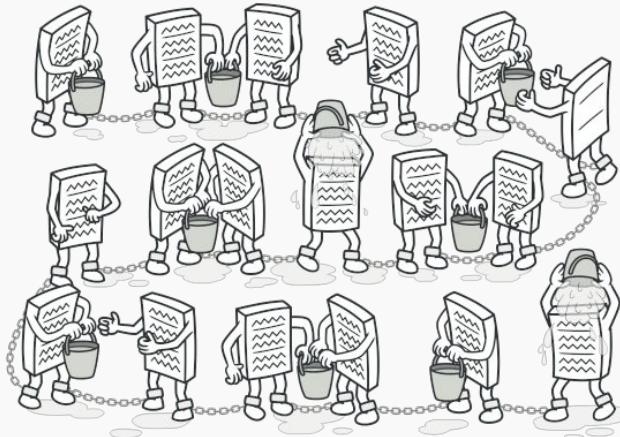
```
const smallNormalHouse = new SmallHouseBuilder().build();
const bigGardenHouse = new BigHouseBuilder().addGarden.build();
const smallLuxuryDirector = new LuxuryHouseDirector(new SmallHouseBuilder());
const smallLuxuryHouse = smallLuxuryDirector.build();
```

Builder - Creational

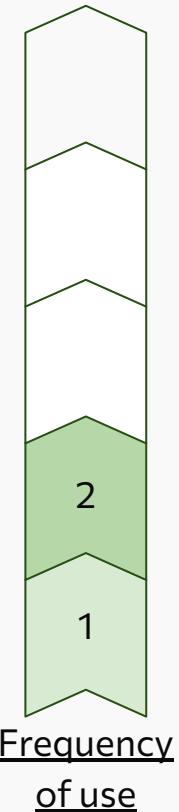
Advantages	Disadvantages
<ul style="list-style-type: none">— Allows constructing objects step by step.	<ul style="list-style-type: none">— Overall complexity increases, because of several new classes
<ul style="list-style-type: none">— Allows reusing the same construction code	
<ul style="list-style-type: none">— Enforces Single Responsibility Principle	

Chain of responsibility - Behavioral

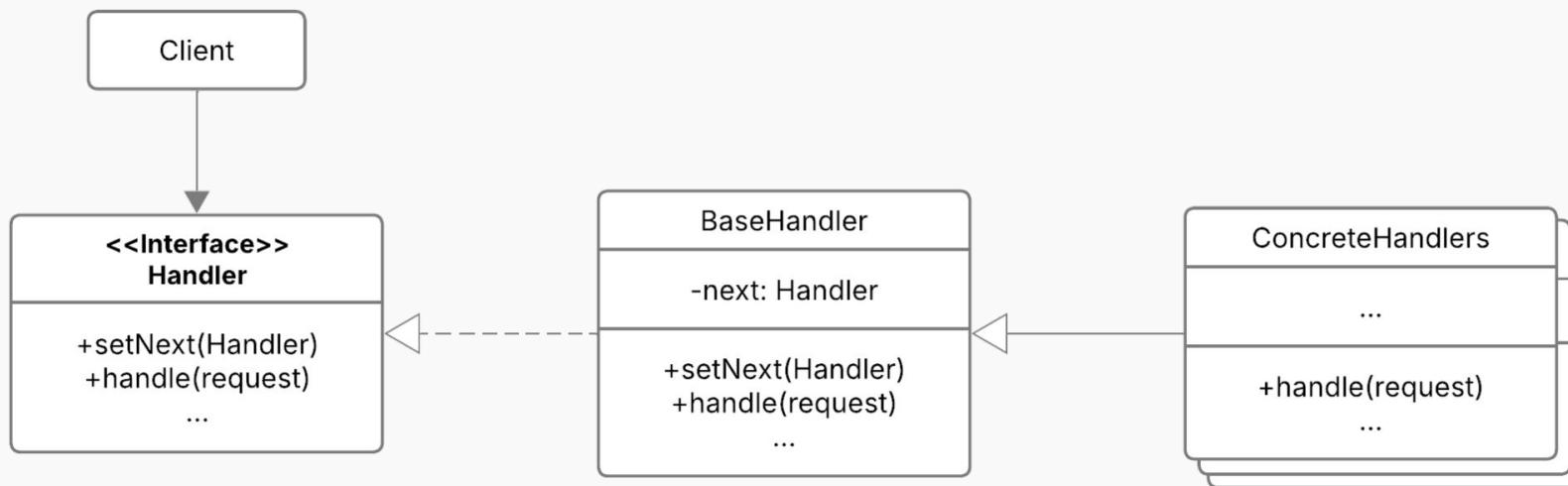
- Allows multiple objects to handle a request without specifying the receiver explicitly.
- It uses a chain of handler objects, where each handler decides to process the request or pass it to the next handler.
- Provides flexibility in handling requests but can lead to unclear flow control.



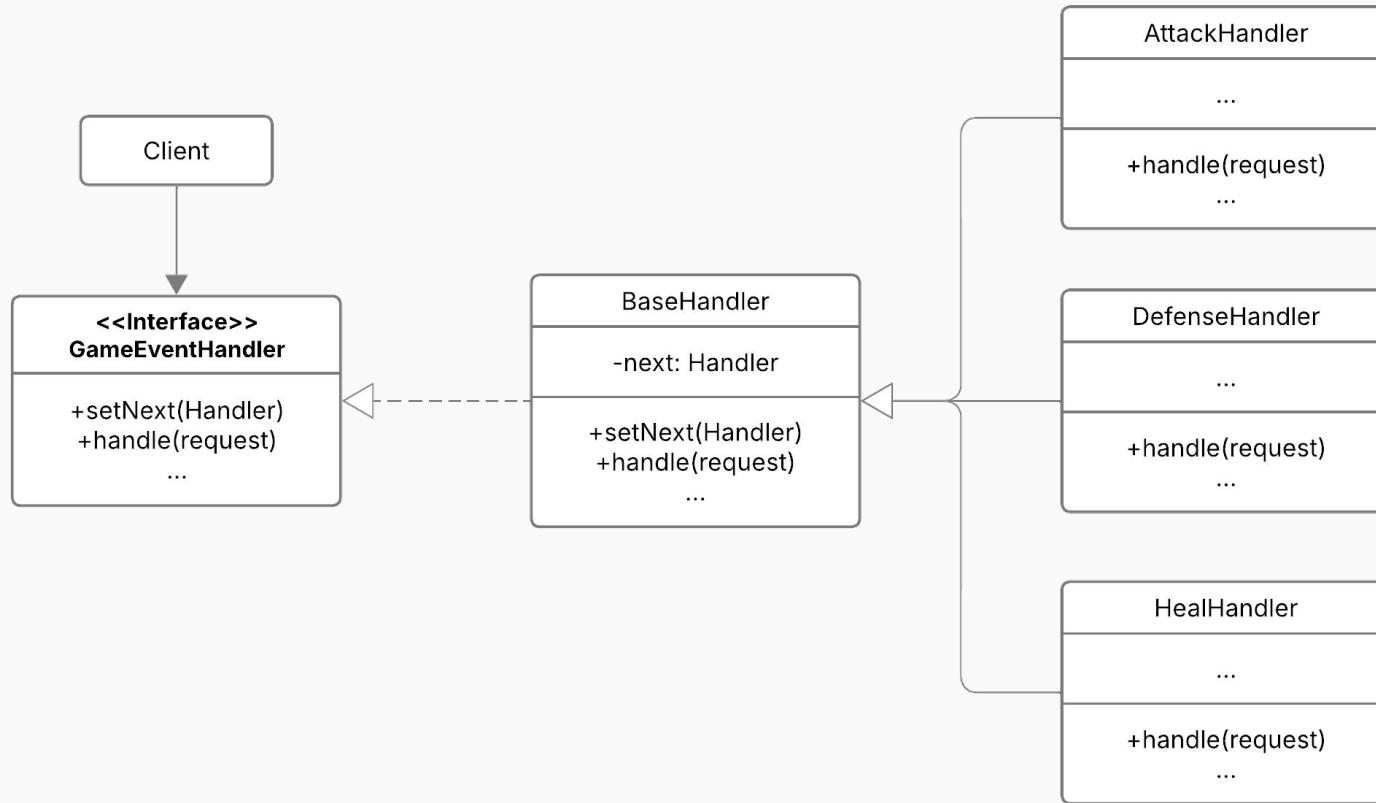
(Refactoring.Guru, 2025)



Class diagram - Generic



Class diagram - Example



GameEventHandler interface

```
interface GameEventHandler {  
    setNextHandler(handler: GameEventHandler): GameEventHandler;  
    handle(event: string): void;  
}
```

Creating the base handler

```
abstract class BaseHandler implements GameEventHandler {
    protected nextHandler: GameEventHandler | null = null;

    setNextHandler(handler: GameEventHandler): GameEventHandler {
        this.nextHandler = handler;
        return handler;
    }

    handle(event: string): void {
        if (this.nextHandler) {
            this.nextHandler.handle(event);
        } else {
            console.log(`No handler found for event: ${event}`);
        }
    }
}
```

Attack handler

```
class AttackHandler extends BaseHandler {  
    handle(event: string): void {  
        if (event === "attack") {  
            console.log("Player attacks the enemy!");  
        } else {  
            super.handle(event);  
        }  
    }  
}
```

Defense handler

```
class DefenseHandler extends BaseHandler {  
    handle(event: string): void {  
        if (event === "defend") {  
            console.log("Player blocks the attack!");  
        } else {  
            super.handle(event);  
        }  
    }  
}
```

Heal handler

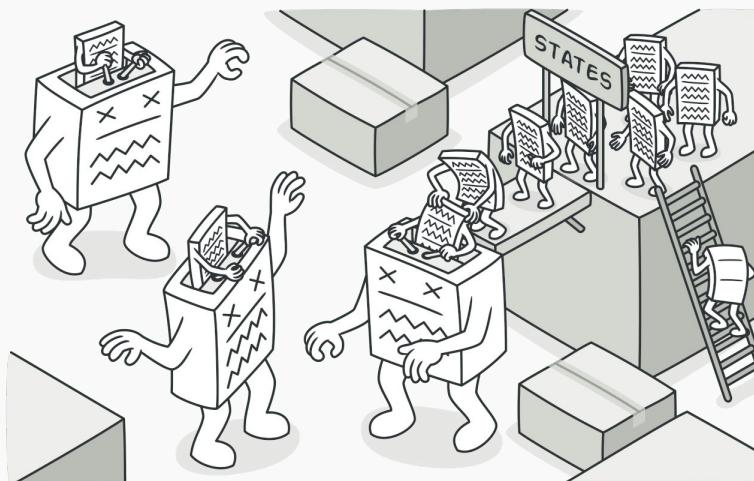
```
class HealHandler extends BaseHandler {  
    handle(event: string): void {  
        if (event === "heal") {  
            console.log("Player uses a potion to restore health!");  
        } else {  
            super.handle(event);  
        }  
    }  
}
```

Chain of responsibility - Behavioral

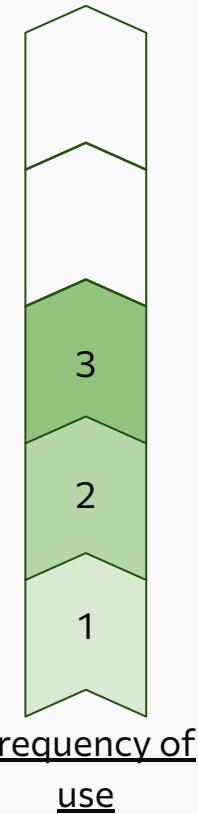
Advantages	Disadvantages
— Decoupling of the sender, the message and the receiver	— Control flow is not guaranteed
— Dynamic request handling	— Added complexity
— Flexibility	— Makes debugging difficult
— Single responsibility	— Could worsen performance

State - Behavioural

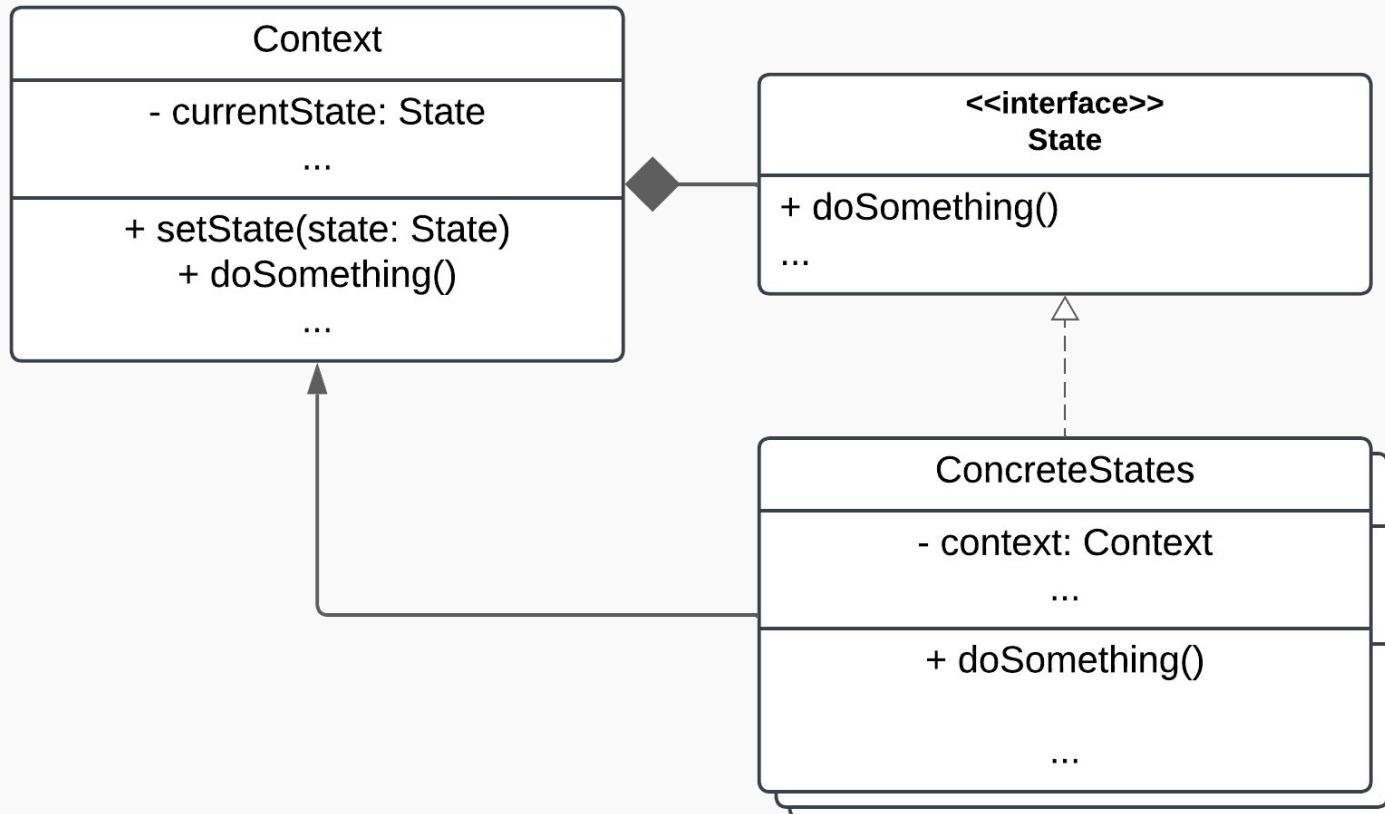
- One State machine object which changes behaviour
- Many State objects which alter behaviour
- More elegant than a chain of if or switch statements



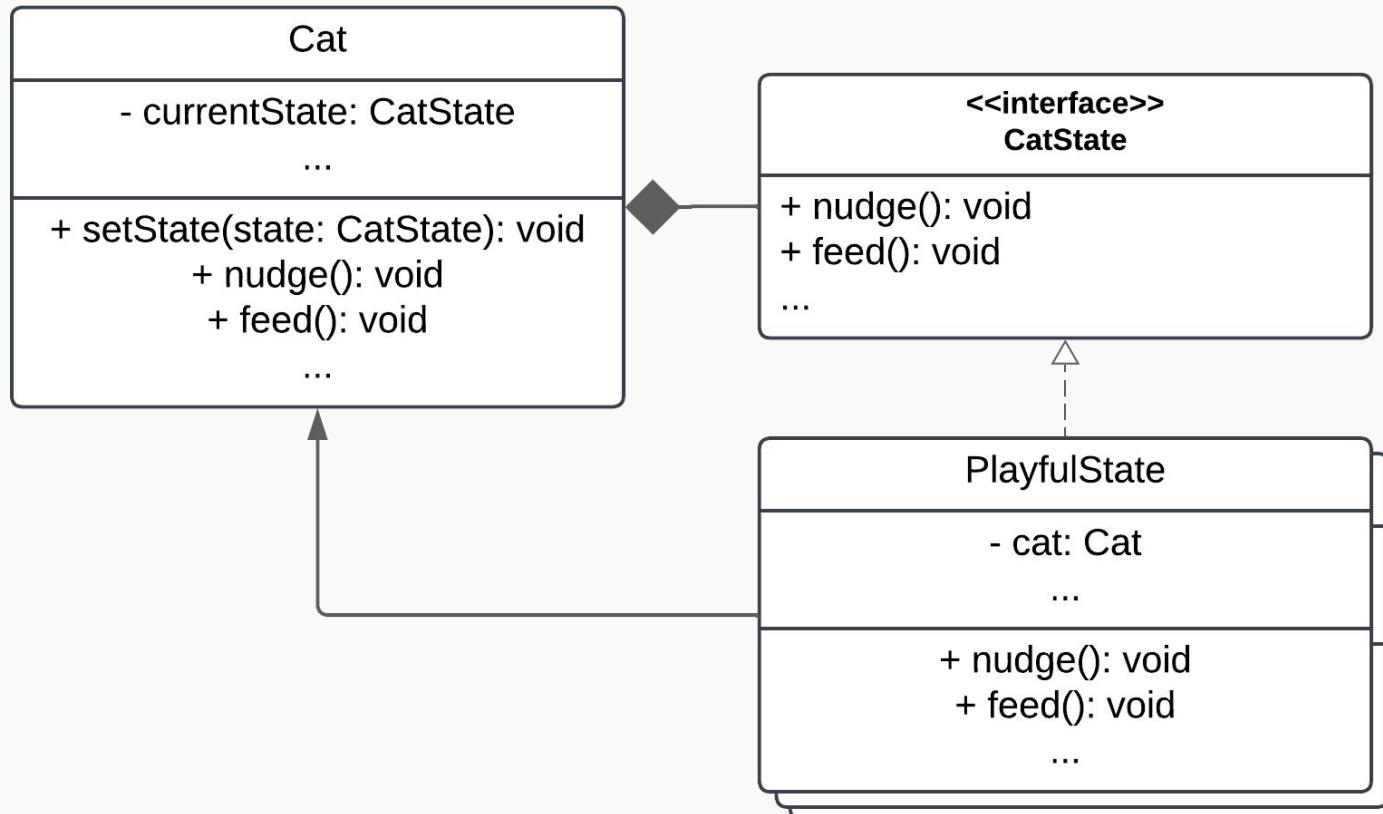
(Refactoring.Guru, 2025)



Class diagram - Generic



Class diagram - Example



Context - Cat

```
export class Cat {  
    private sleepingState: CatState;  
    private hungryState: CatState;  
    private playfulState: CatState;  
    private currentState: CatState;  
  
    constructor() {  
        this.sleepingState = new SleepingState(this);  
        this.hungryState = new HungryState(this);  
        this.playfulState = new PlayfulState(this);  
        // Cat starts off sleeping  
        this.currentState = this.sleepingState;  
    }  
    ...  
}
```

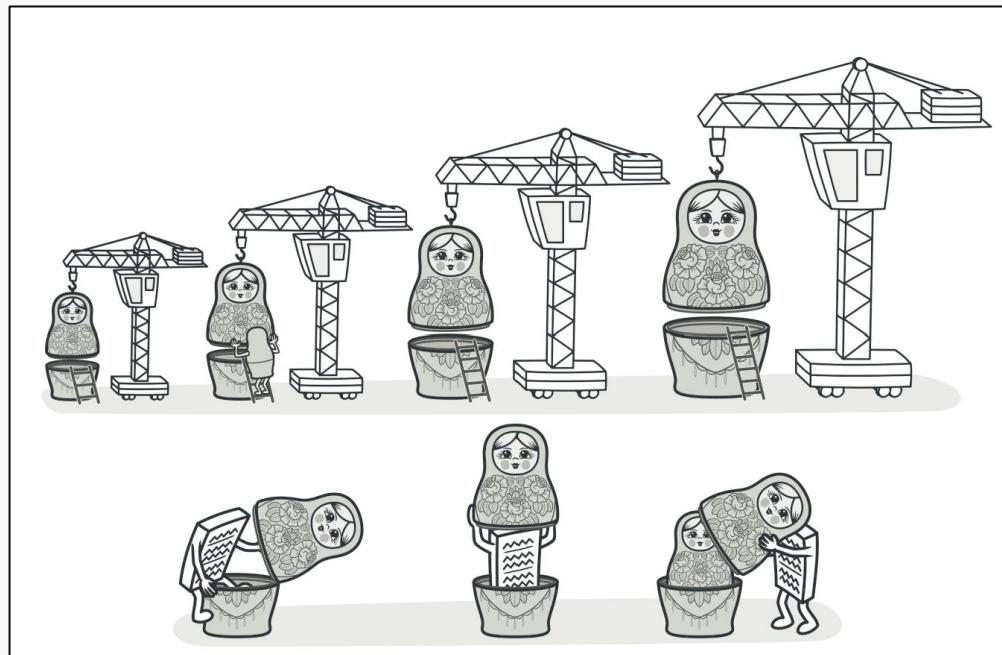
State - PlayfulState

```
export class PlayfulState implements CatState {  
    private cat: Cat;  
  
    constructor(cat: Cat) {  
        this.cat = cat;  
    }  
  
    nudge(): void {  
        console.log("The cat gets startled and decides it's time to sleep.");  
        this.cat.setState(this.cat.getSleepingState());  
    }  
}
```

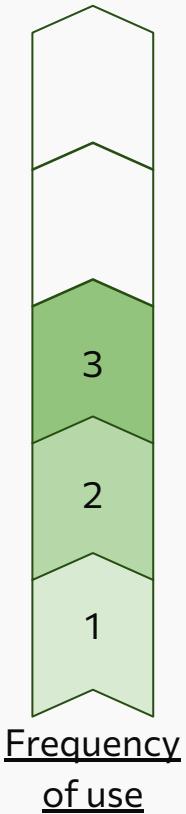
State - Behavioural

Advantages	Disadvantages
— Enforces Single responsibility principle	— Increases general code complexity
— Enforces Open/Closed principle	— Difficult to unit test
— Easy unit testing	— Complex initial setup
— Readable code	— Can be memory-heavy

Decorator - Structural



(Refactoring.Guru, 2025)



The Enemy interface

```
export interface Enemy {  
    name(): string();  
    receivedDamage(damage: number): number;  
}
```

Creating a concrete Enemy

```
export class ZombieEnemy implements Enemy {
    public name(): string {
        return 'Zombie';
    }
    public receiveDamage(damage: number): number {
        return damage;
    }
}
```

Creating a Zombie with Helmet

```
export class ZombieWithHelmet implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 5;  
    }  
}
```

Creating a Zombie with Chestplate

```
export class ZombieWithChestplate implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 10;  
    }  
}
```

Creating a Zombie with... both?

```
export class ZombieWithHelmetAndChestplate implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 5 - 10;  
    }  
}
```

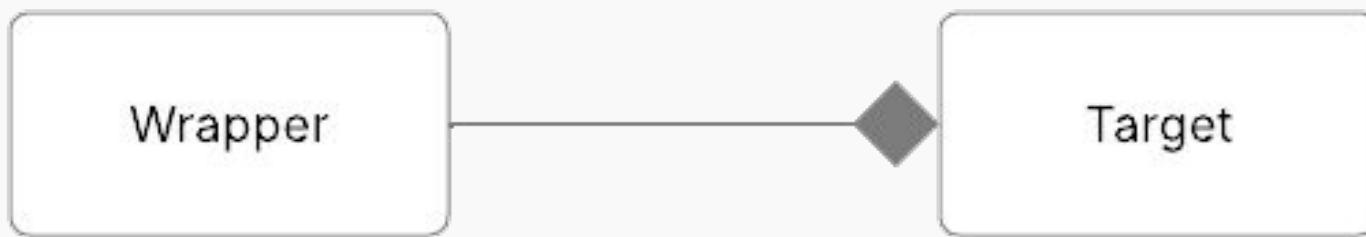
This is horrible

```
export class ZombieWithChestplate implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 5;  
    }  
}
```

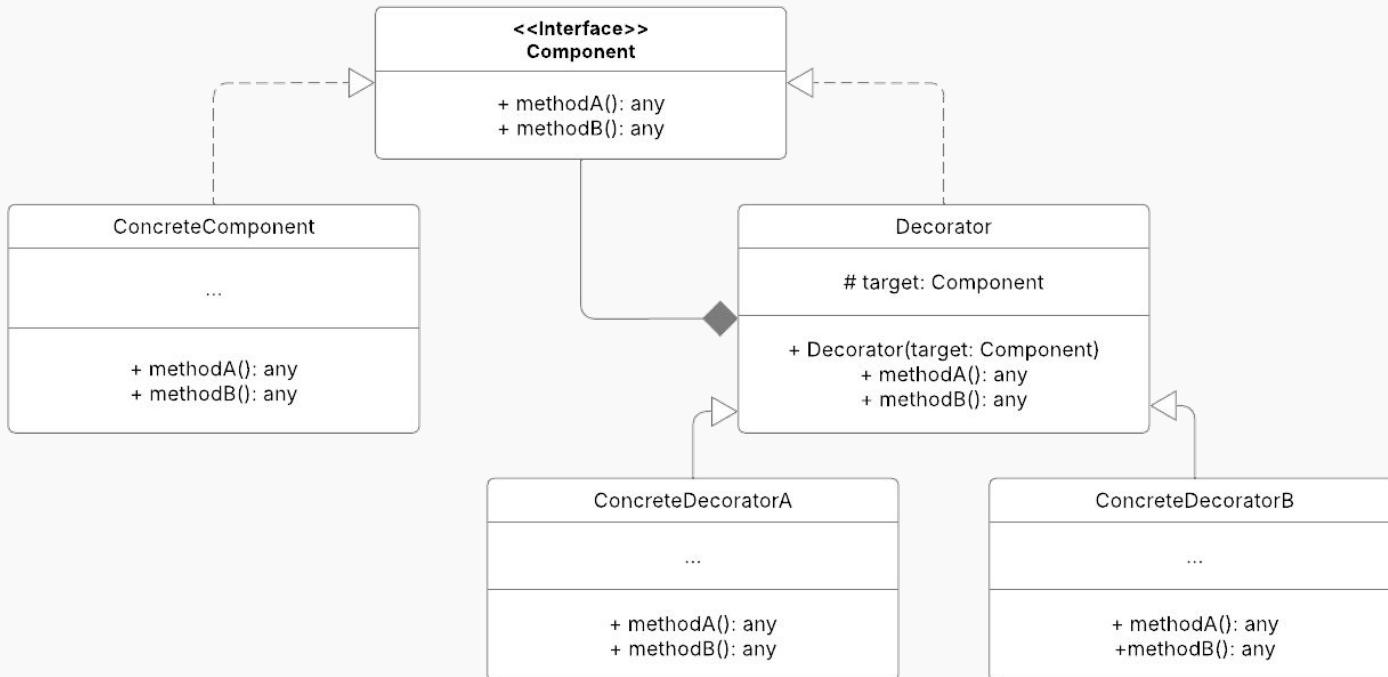
```
export class ZombieWithHelmet implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 5;  
    }  
}
```

```
export class ZombieWithHelmetAndChestplate implements Enemy {  
    public name(): string {  
        return 'Zombie';  
    }  
    public receiveDamage(damage: number): number {  
        return damage - 5 - 10;  
    }  
}
```

Wrapper ~= Decorator



Class Diagram - Generic



Creating the abstract decorator

```
export class EnemyDecorator implements Enemy {
    protected enemy: Enemy;
    public constructor(enemy: Enemy) {
        this.enemy = enemy;
    }
    public name(): string {
        return this.enemy.name();
    }
    public abstract receiveDamage(damage: number): number;
}
```

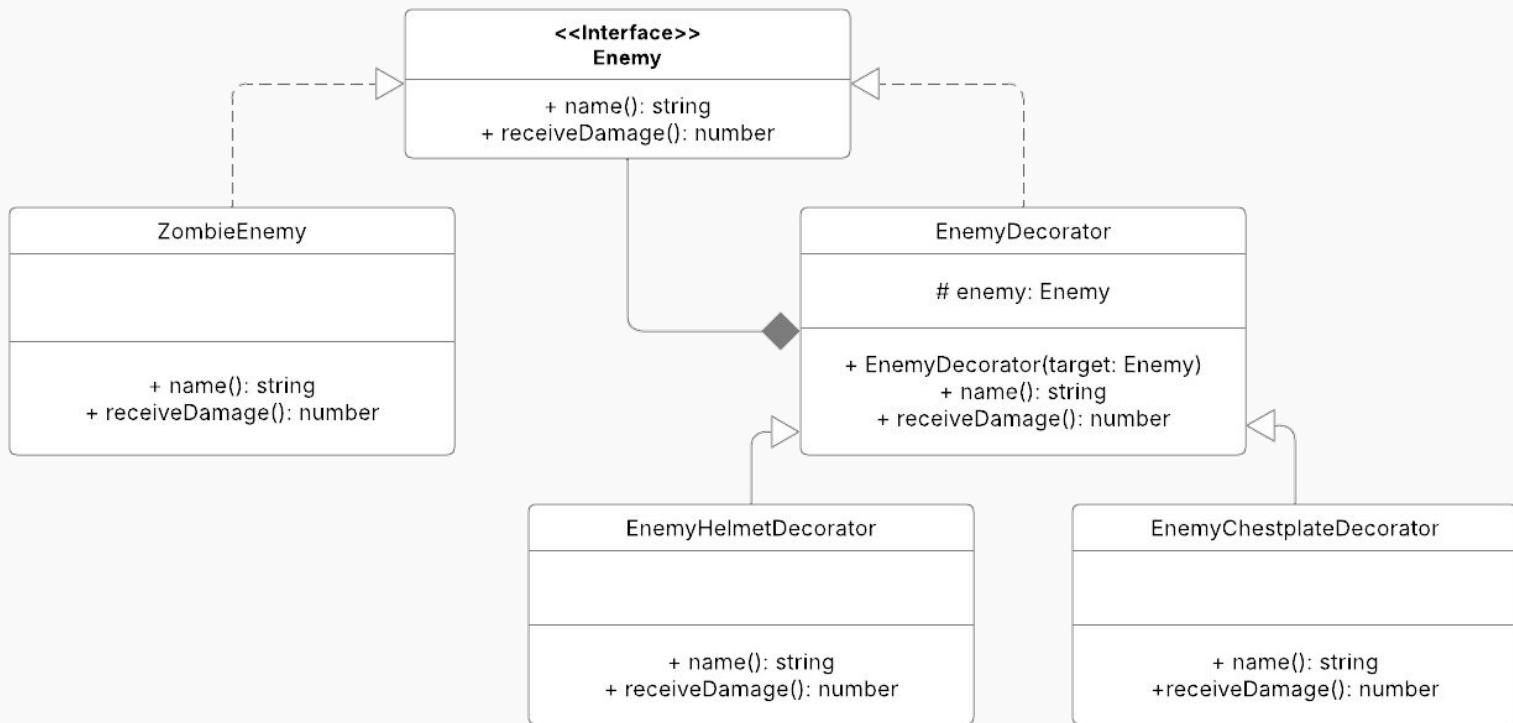
Creating the Helmet decorator

```
export class EnemyHelmetDecorator extends EnemyDecorator {
    public receiveDamage(damage: number) {
        return this.enemy.receiveDamage(damage) - 5;
    }
}
```

Creating the Chestplate decorator

```
export class EnemyChestplateDecorator extends EnemyDecorator {
    public receiveDamage(damage: number) {
        return this.enemy.receiveDamage(damage) - 10;
    }
}
```

Class Diagram - Example



Usage example (1)

```
let zombie = new ZombieEnemy();
console.log(zombie.receiveDamage(50)); // 50
let zombie = new EnemyWithHelmet(zombie);
console.log(zombie.name()); // Zombie
console.log(zombie.receiveDamage(50)); // 50 - 5 = 45
```

Usage example (2)

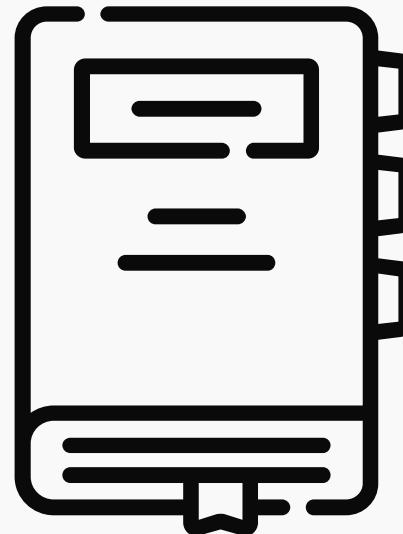
```
let zombie = new ZombieEnemy();
let zombie = new EnemyWithHelmet(zombie);
console.log(zombie.receiveDamage(50)) // 50 - 10 = 40
let zombie = new EnemyWithChestplate(zombie);
console.log(zombie.name()); // Zombie
console.log(zombie.receiveDamage(50)); // 50 - 5 - 10 = 35
```

Decorator - Structural

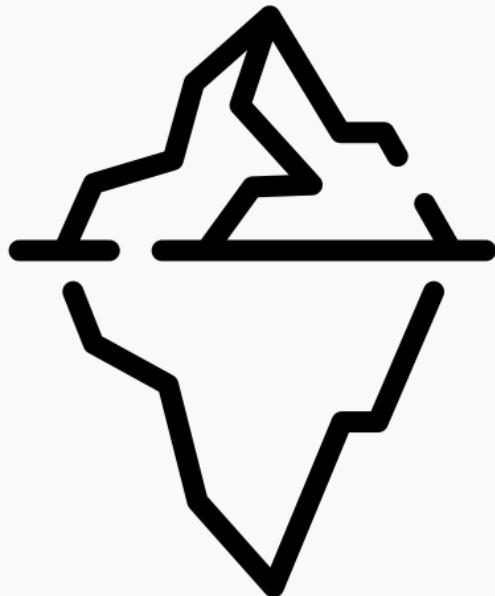
Advantages	Disadvantages
<ul style="list-style-type: none">— Enforces Single responsibility principle	<ul style="list-style-type: none">— Hard to un-do wrapping
<ul style="list-style-type: none">— You can extend behavior without new subclasses	<ul style="list-style-type: none">— Behavior depends on decorators order
<ul style="list-style-type: none">— Combine several behaviors easily	<ul style="list-style-type: none">— Initial code might look ugly

04

Conclusion



This is just the tip of the iceberg



- Observer
- Factory
- Abstract factory
- Strategy
- Template

...

References

- Dofactory. (s. f.). *C# design Patterns*. <https://dofactory.com/net/design-patterns>
- Buyya, R. (n.d.). *Patterns*. Grid Computing and Distributed Systems (GRIDS) Laboratory, The University of Melbourne. <http://www.buyya.com/254/Patterns/>
- Refactoring.Guru. (s. f.). *The Catalog of Design Patterns*.
<https://refactoring.guru/design-patterns/catalog>
- Sousa, B. L., Ferreira, M. M., Da Silva Bigonha, M. A., & Ferreira, K. A. M. (2020). Design Patterns in Practice from the Point of View of Developers. *Abakós*, 8(1), 20-42. <https://doi.org/10.5752/p.2316-9451.2020v8n1p20-42>
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Thanks!

Do you have any questions?

roberto.gimenez.19@ull.edu.com

dario.fajardo.31@ull.edu.es

eric.rios.41@ull.edu.es