18/03/2024

# Design Patterns II

# About us

Igor
Dragone
igor.dragone.13@ull.edu.es

José Ramón
Morera Campos
jose.morera.27@ull.edu.es

# Table of contents

# 01
# Introduction

“Design patterns are **named** solutions to
a problem in a context”

—Robert C. Martin

# What are Design Patterns?

- Collective knowledge
- Language for communication

# Why to use Design Patterns?

- Code efficiency
- Reusability
- Maintainability

# Patterns classification

01

Creational

02

Structural

03

Behavioral

Learn More

# Patterns popularity

Indicate how frequently the patterns are used

# 02

# Creational Patterns

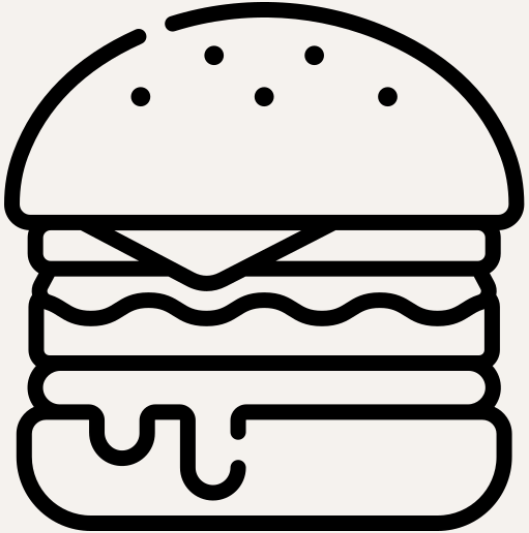
# Creational Patterns

- Provide object creation mechanisms
  - Encapsulate knowledge about construction
  - Increase flexibility

# Factory method ★★★

- Interface for creating objects
- Different object types may be created

# Factory method - Example



```
interface Burger {
 prepare(): void;
 cook(): void;
 box(): void;
}
```

```
class CheeseBurger implements Burger {
 public prepare(): void {
   console.log('Preparing the Cheese Burger');
 }
 public cook(): void {
   console.log('Cooking the Cheese Burger');
 }
 public box(): void {
   console.log('Boxing the Cheese Burger');
 }
}
```

```
class ChickenBurger implements Burger {
 public prepare(): void {
   console.log('Preparing the Chicken Burger');
 }
 public cook(): void {
   console.log('Cooking the Chicken Burger');
 }
 public box(): void {
   console.log('Boxing the Chicken Burger');
 }
}
```

```
class BurgerStore {
 public orderBurger(type: string): Burger {
   let burger: Burger;
   switch (type) {
     case 'cheese':
     burger = new CheeseBurger();
     break;
     case 'chicken':
     burger = new ChickenBurger();
     break;
     default:
     throw new Error('Invalid burger type');
   }
   burger.prepare();
   burger.cook();
   burger.box();
   return burger;
 }
```

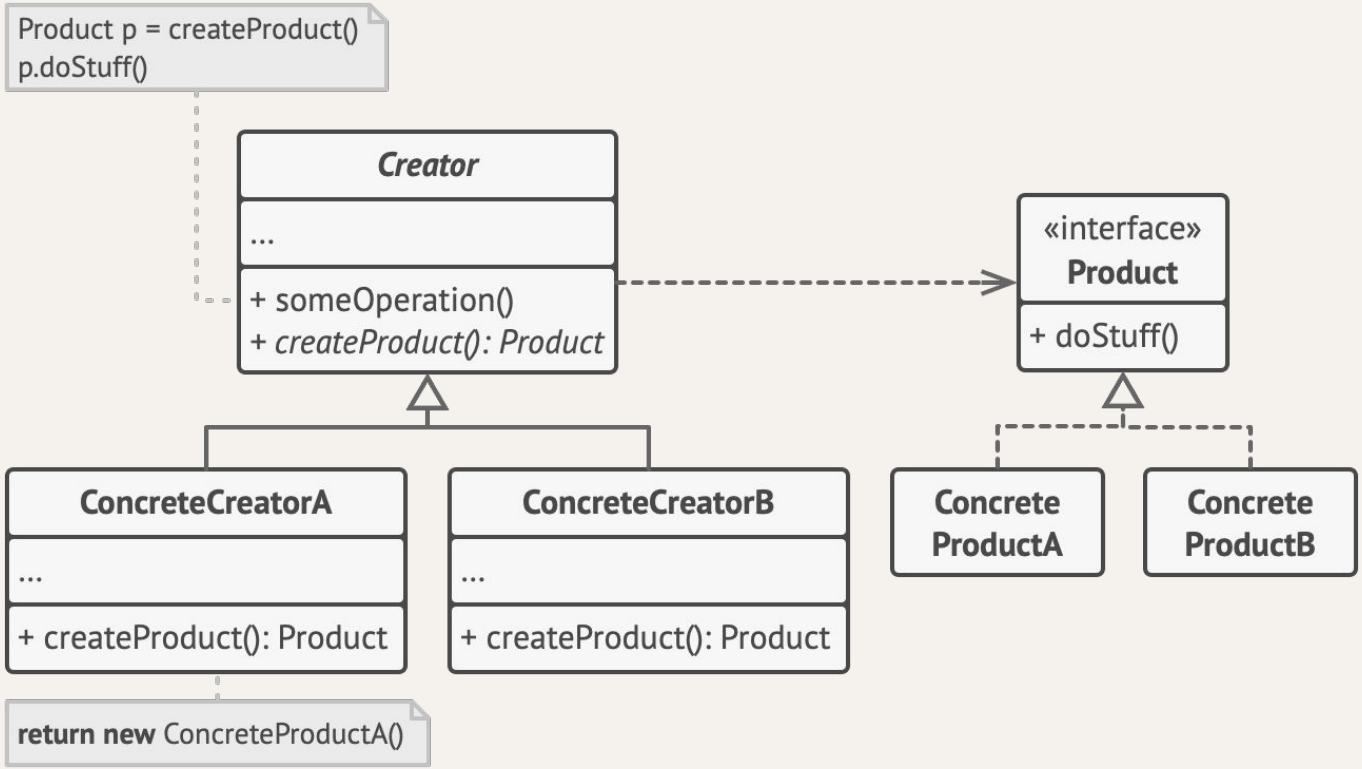
# Factory method - Problems

- Violates Open/Closed principle
- Violates dependency inversion principle

# Factory method - Solution

- Delegate object creation
- Factory method may be overridden

# Factory method - Diagram



```
abstract class BurgerStore {
 public orderBurger(type: string): Burger {
   let burger = this.createBurger(type);

   burger.prepare();
   burger.cook();
   burger.box();
   return burger;
 }
 // Factory method
 protected abstract createBurger(type: string): Burger;
}
```

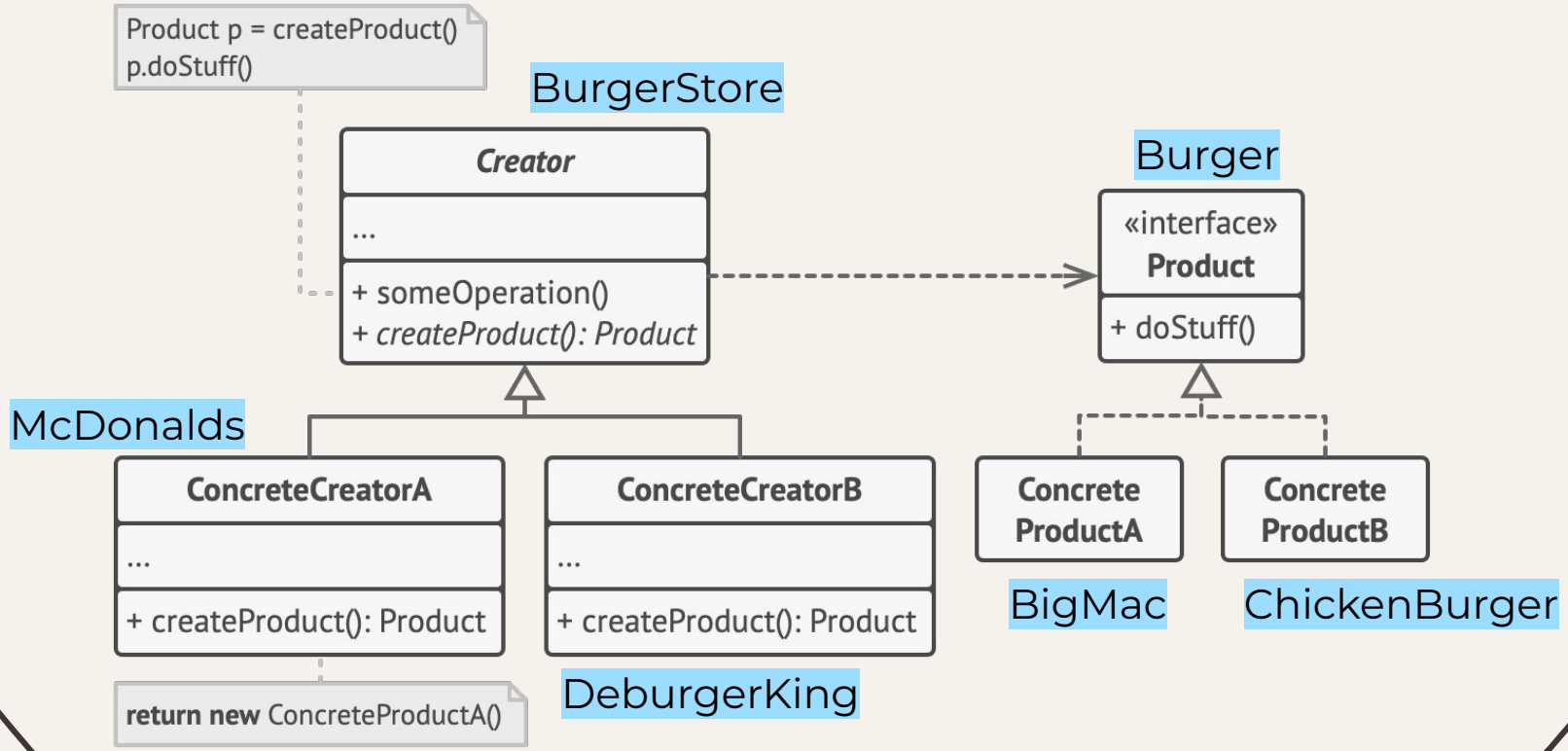
```
class DeburgerKing extends BurgerStore {
 protected createBurger(type: string): Burger {
   switch(type) {
     case 'cheese':
       return new CheeseBuger();
     case 'chicken':
       return new ChickenBurger();
     default:
       throw new Error('Invalid burger type');
   }
 }
}
```

```
class McDonalds extends BurgerStore {
 protected createBurger(type: string): Burger {
   switch(type) {
     case 'bigmac':
       return new BigMac();
     case 'chicken':
       return new ChickenBurger();
     default:
       throw new Error('Invalid burger type');
   }
 }
}
```

# Factory method - Benefits

- Easy to introduce new products
- Avoid dependency between use and creation

# Factory method - Diagram



# Abstract Factory ★★★

- Lets you produce families of related objects
- Abstracts from implementation

# Abstract Factory - Example

```
interface SUV {
 getHorsepower(): number;
}

class PeugeotSUV implements SUV {
 public getHorsepower(): number {
   return 120;
 }
}

class PorscheSUV implements SUV {
 public getHorsepower(): number {
   return 300;
 }
}
```

```
interface Sedan {
 /// Get the cargo capacity in liters
 getCargo(): number;
}

class PeugeotSedan implements Sedan {
 public getCargo(): number {
   return 150;
 }
}
class PorscheSedan implements Sedan {
 public getCargo(): number {
   return 100;
 }
}
```

```
abstract class SedanFactory {
 public abstract createSedan(): Sedan;
}

class PeugeotSedanFactory extends SedanFactory {
 public createSedan(): Sedan {
   console.log('Creating a Peugeot Sedan');
   return new PuegeotSedan();
 }
}
class PorscheSedanFactory extends SedanFactory {
 public createSedan(): Sedan {
   console.log('Creating a Porsche Sedan');
   return new PorscheSedan();
 }
}
```

```
abstract class SUVFactory {
 public abstract createSUV(): SUV;
}

class PeugeotSUVFactory extends SUVFactory {
 public createSUV(): SUV {
   console.log('Creating a Peugeot SUV');
   return new PuegeotSUV();
 }
}
class PorscheSUVFactory extends SUVFactory {
 public createSUV(): SUV {
   console.log('Creating a Porsche SUV');
   return new PorscheSUV();
 }
}
```

```
export function main(): void {
 let firstCar = new PorscheSUVFactory().createSUV();

 /** Imagine some other code */

 let secondCar = new PeugeotSUVFactory().createSUV();
 // Oh no, we messed up and built the wrong car!
}
```

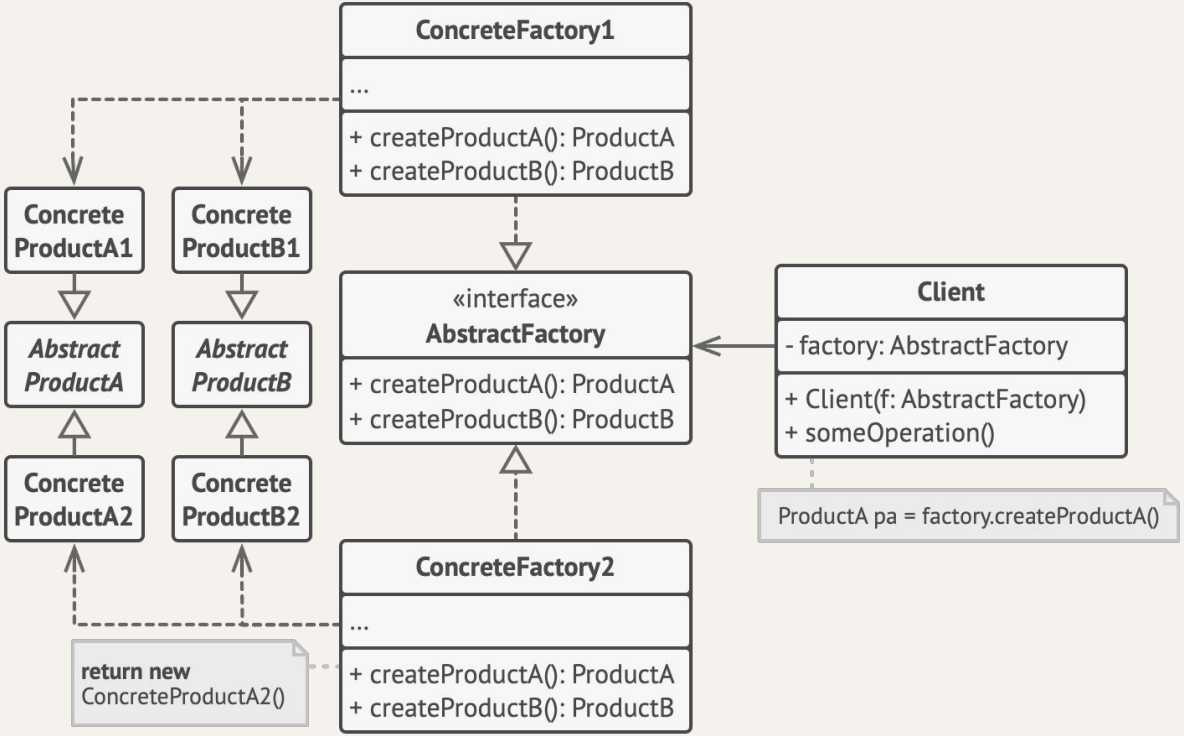
# Abstract Factory - Problems

- Easy to get the wrong variant
- Too many classes

# Abstract Factory - Solution

- Declare interfaces for each distinct product
- Abstract factory interface with a set of creation methods for all abstract products

# Abstract Factory - Diagram



```
/// Abstract factory interface.
interface CarFactory {
 createSedan(): Sedan;
 createSUV(): SUV;
}
```

```
class PeugeotCarFactory implements CarFactory {
 /// Create a Peugeot SUV
 public createSUV(): SUV {
   console.log('Creating a Peugeot SUV');
   return new PuegeotSUV();
 }

 /// Create Peugeot Sedan
 public createSedan(): Sedan {
   console.log('Creating a Peugeot Sedan');
   return new PuegeotSedan();
 }
}
```

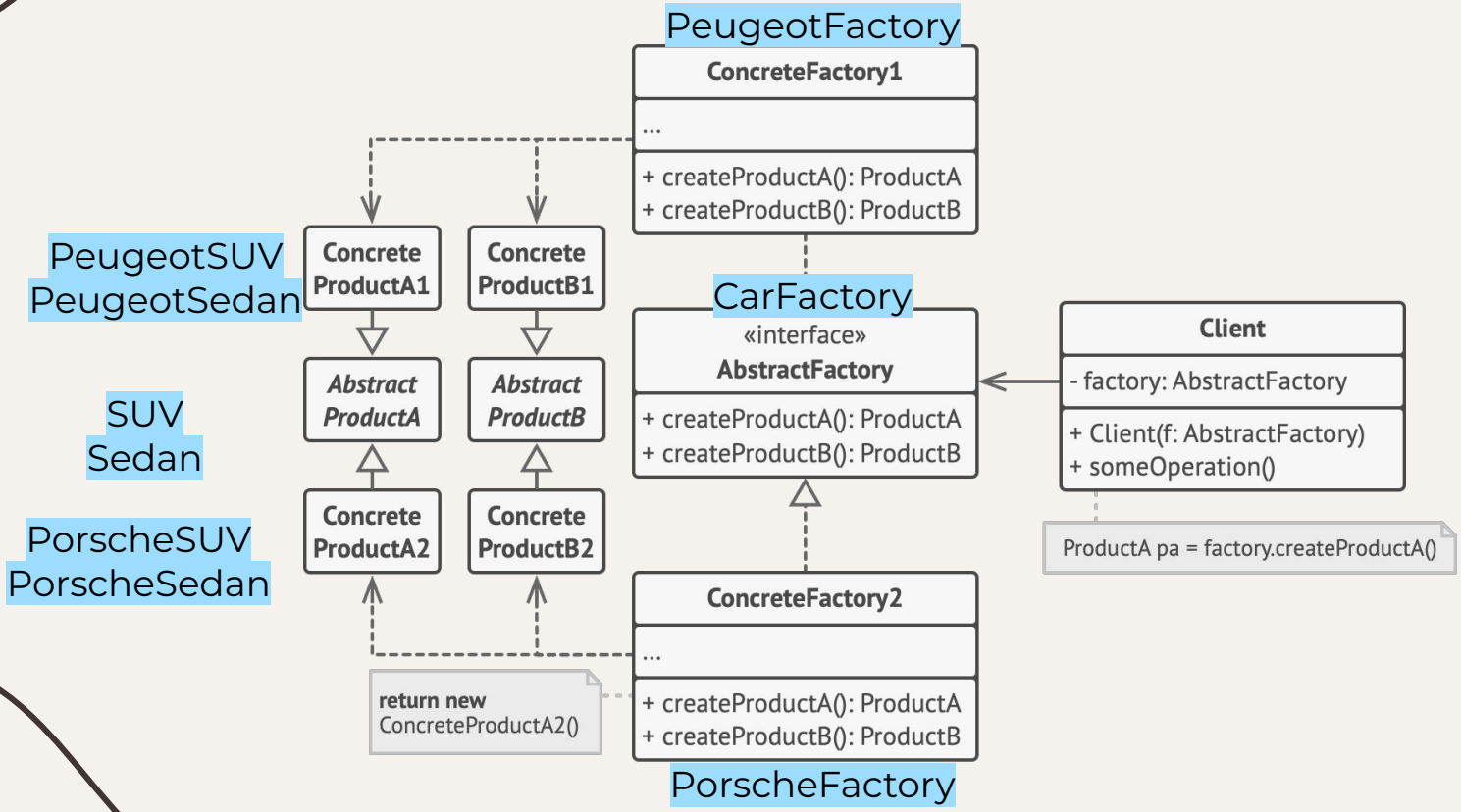
```
class PorscheCarFactory implements CarFactory {
 /// Create a Porsche SUV
 public createSUV(): SUV {
   console.log('Creating a Porsche SUV');
   return new PorscheSUV();
 }

 /// Create Porsche Sedan
 public createSedan(): Sedan {
   console.log('Creating a Porsche Sedan');
   return new PorscheSedan();
 }
}
```

```
export function main(): void {
 let currentCarFactory: PorscheCarFactory = new
   PorscheCarFactory();
 let mySUV: SUV = currentCarFactory.createSUV();
 let mySedan: Sedan = currentCarFactory.createSedan();

 /// Now we have matching cars!
}
```

# Abstract Factory - Benefits

- Change objects type dynamically
- Ensure object compatibility

# Abstract Factory - Diagram



# 03
# Structural Patterns

# Structural Patterns

- Assemble objects and classes into larger structures
  - Simplify design
  - Reduce duplications
  - Keep structures flexible and efficient

“Most of the design patterns that have appeared in the last 15 years are just well-known ways to eliminate duplication”

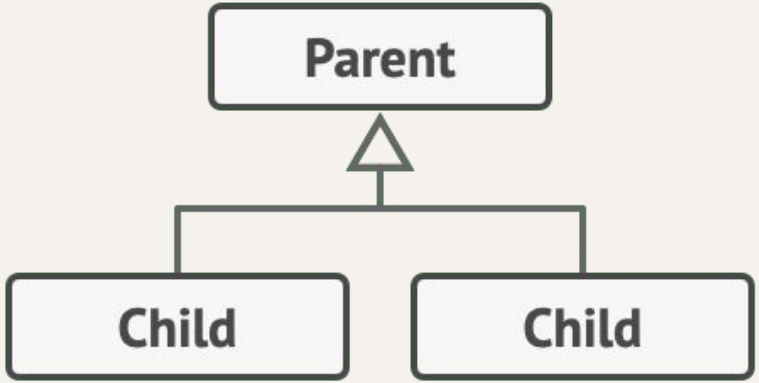
—Robert C. Martin, “Clean Code”

# Decorator

- Lets you attach new behaviors to objects
- Wraps them in a decorator that contains the behavior
- Composition over inheritance

# Composition vs Inheritance

*Inheritance*

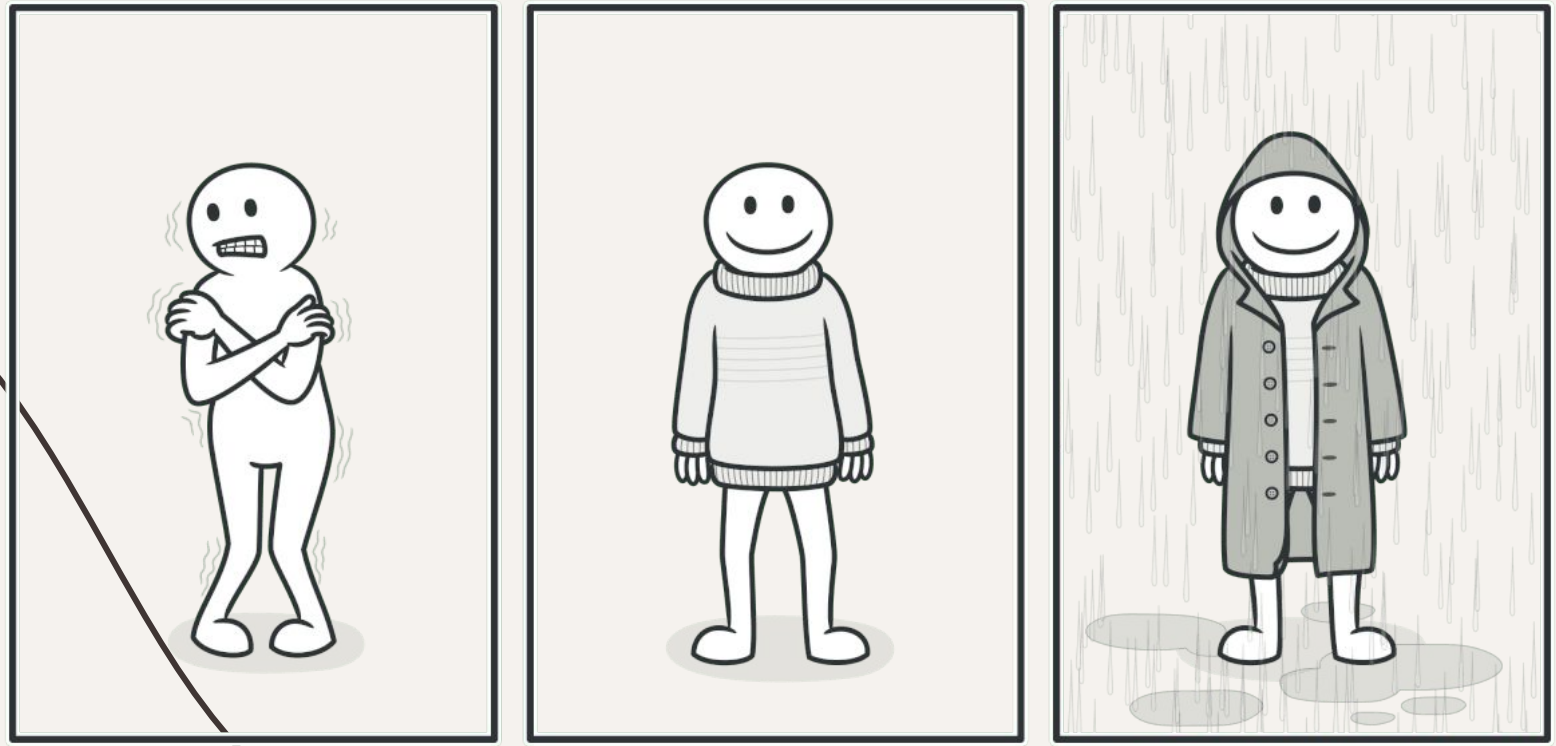



Class B **is** Class A

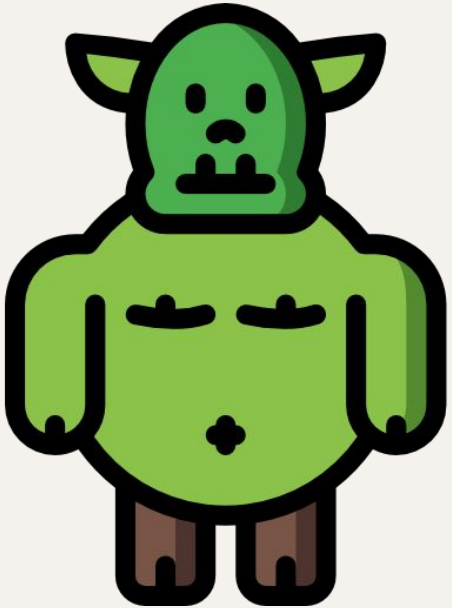
*Aggregation*


Class B **has** Class A

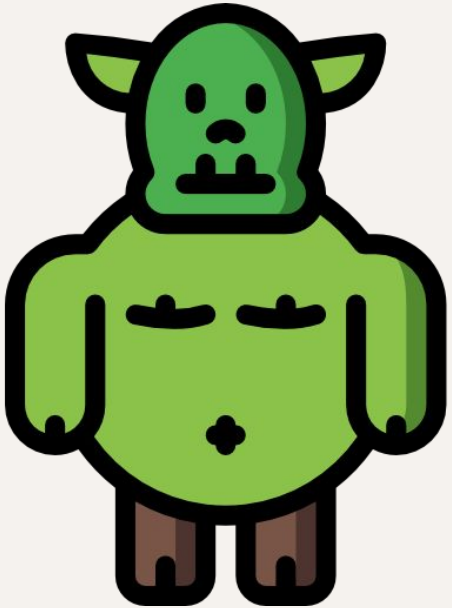
# Decorator - Analogy



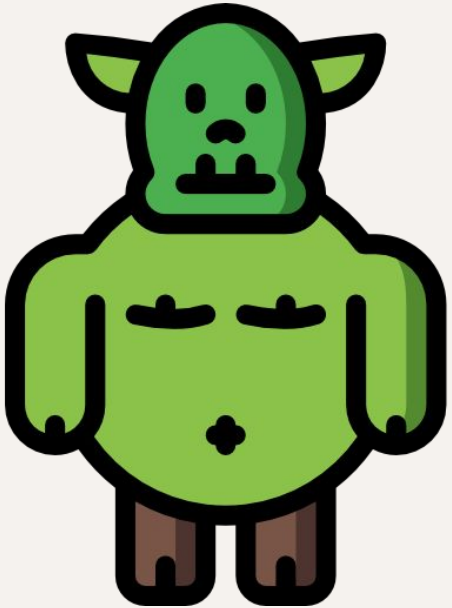
# Decorator - Example



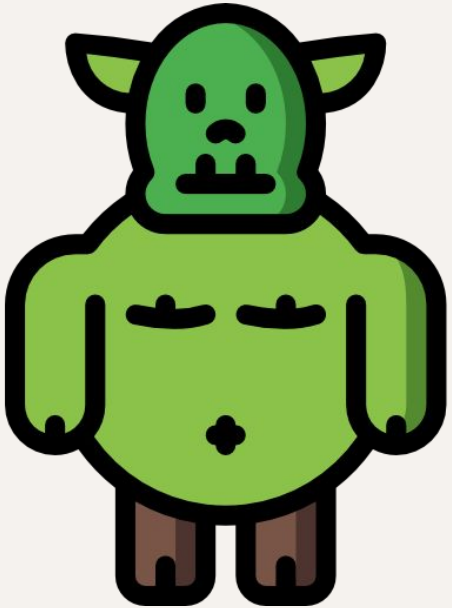
# Decorator - Example



# Decorator - Example



# Decorator - Example



```
interface Enemy {
 attack(): void;
};

class Troll implements Enemy {
 /**
  * Attacks the player.
  */
 public attack(): void {
   console.log('Troll attacks');
 }
};
```

```
class TrollWithSword extends Troll {
 /**
  * Attacks the player with a sword.
  */
 public attack(): void {
   super.attack();
   console.log(' with a sword!');
 }
}
```

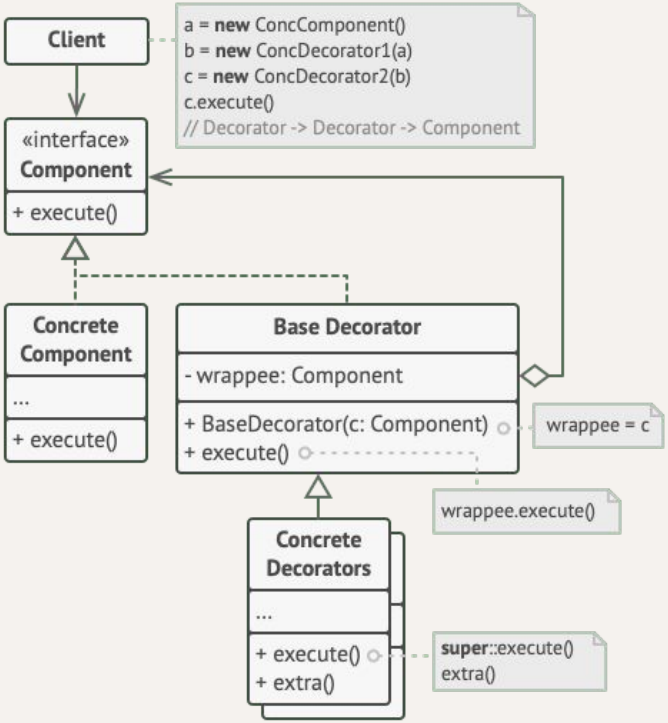
```
class TrollWithGun extends Troll {
 /**
  * Attacks the player with a gun.
  */
 public attack(): void {
   super.attack();
   console.log(' with a gun!');
 }
}
```

```
class TrollWithSwordAndGun extends Troll {
 /**
  * Attacks the player with a sword and a gun.
  */
 public attack(): void {
   super.attack();
   console.log(' with a sword and a gun!');
 }
}
```

# Decorator- Problems

- Unscalable code. Exponential growth!
- Duplicated code
- Cannot change behavior at runtime

# Decorator - Diagram



# Decorator - Solution

- Separate the code in:
  - Component (Wrapped)
  - ConcreteComponent
  - Decorator (Wrapper)
  - ConcreteDecorator

```
interface Enemy {
 attack(): void;
};

class Troll implements Enemy {
 /**
  * Attacks the player.
  */
 public attack(): void {
   console.log('Troll attacks');
 }
};
```

```
abstract class EnemyDecorator implements Enemy {
 constructor(protected enemy: Enemy) {}

 /**
  * The decorator delegates all work to the wrapped
component.
  */
 public attack(): void {
   this.enemy.attack();
 }
};
```

```
class SwordDecorator extends EnemyDecorator {
 /**
  * Adds a sword to the enemy.
  */
 public attack(): void {
   super.attack();
   console.log(' with a sword!');
 }
};
```

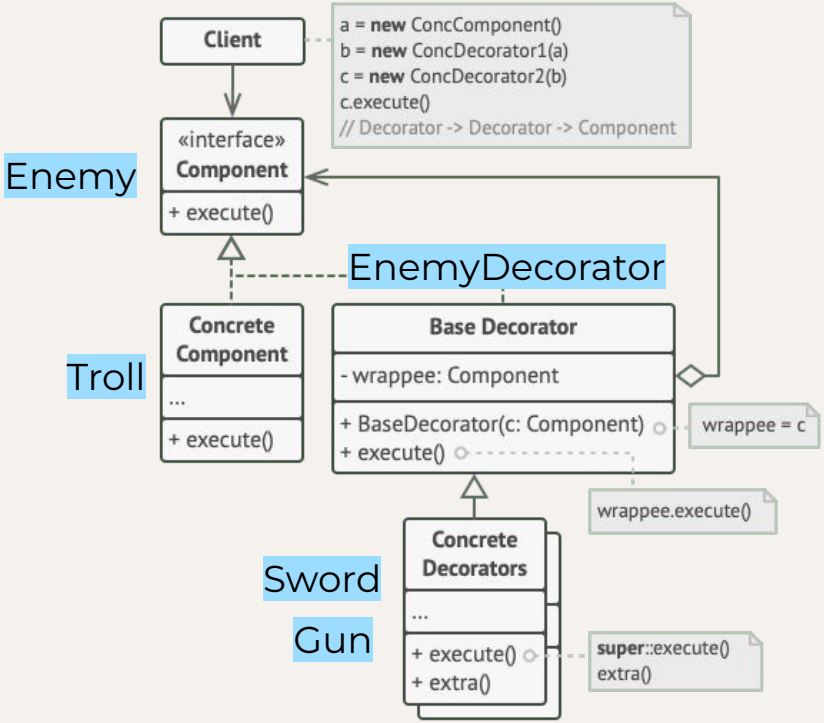
```
class GunDecorator extends EnemyDecorator {
 /**
  * Adds a gun to the enemy.
  */
 public attack(): void {
   super.attack();
   console.log(' with a sword!');
 }
};
```

```typescript
export function main() {
 const troll: Enemy = new Troll();
 troll.attack();

 const trollWithSword: Enemy = new Sword(troll);
 trollWithSword.attack();

 const trollWithSwordAndGun: Enemy = new Gun(trollWithSword);
 trollWithSwordAndGun.attack();
}
```

# Decorator - Benefits

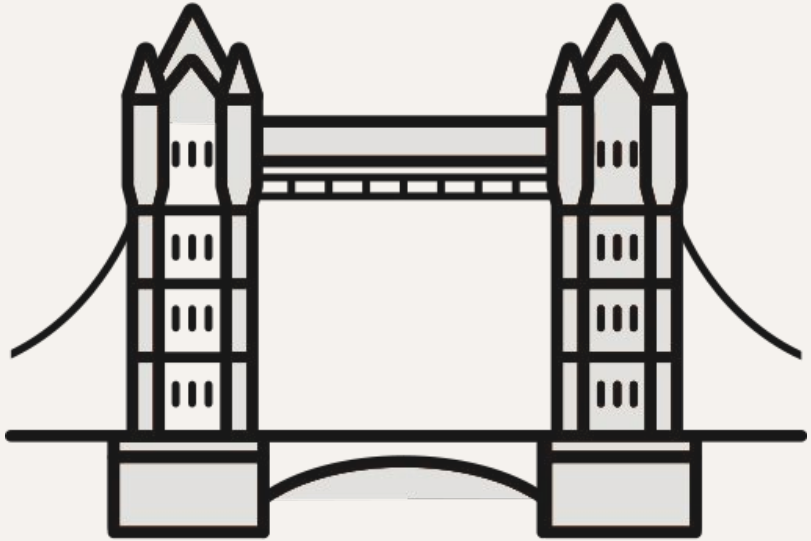
- SOLID friendly
- Possibility to extend behavior without a new subclass
- Easy to add or remove responsibilities at runtime

# Decorator - Diagram



# Bridge

- Splits a large class into two separate hierarchies



# Bridge - Example

```
abstract class Pizza {
 constructor(protected price: number, protected topping:
   string) {}
 /// Prepares the pizza
 public abstract assemble(): void;

 public getPrice(): number {
   return this.price;
 }
}
```

```
class PepperoniPizza extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 public assemble(): void {
   console.log('Preparing dough');
   console.log(`Adding toppings: ${this.topping}`);
   console.log('Adding Pepperoni');
   console.log('Baking the pizza');
 }
}
```

```
class HawaiianPizza extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 public assemble(): void {
   console.log('Preparing dough');
   console.log(`Adding toppings:
${this.topping}`);
   console.log('Adding Pineapple');
   console.log('Baking the pizza');
 }
}
```

```
class PepperoniCalzone extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 public assemble(): void {
   console.log('Preparing dough');
   console.log(`Adding toppings: ${this.topping}`);
   console.log('Adding Pepperoni');
   console.log('Folding in half');
   console.log('Baking the calzone');
 }
}
```

```
class HawaiianCalzone extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 public assemble(): void {
   console.log('Preparing dough');
   console.log(`Adding toppings: ${this.topping}`);
   console.log('Adding Pineapple);
   console.log('Folding in half');
   console.log('Baking the calzone');
 }
}
```

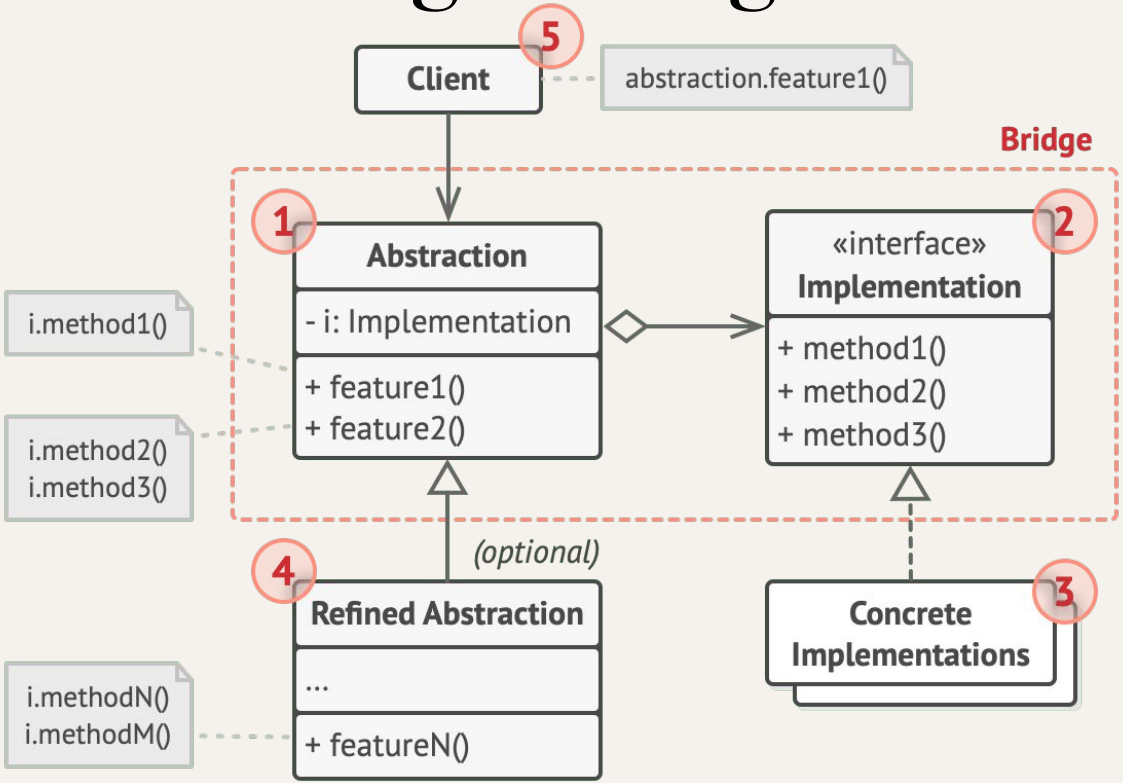
# Bridge - Problems

- Unscalable code. Exponential growth!
- Duplicated code

# Bridge - Solution

- Divide into two classes
- Use composition: one class has the other.

# Bridge - Diagram



```
abstract class Pizza {
 constructor(protected price: number, protected
     topping: string, protected flavor: Flavor) {}
 /// Prepares the pizza
 public abstract assemble(): void;

 public getPrice(): number {
   return this.price + this.flavor.getPrice();
 }
}
```

```
abstract class Flavor {
 constructor(protected price: number) {}

 /// Prepares the ingredients for the flavor
 public abstract prepare(): void;

 public getPrice(): number {
   return this.price;
 }
}
```

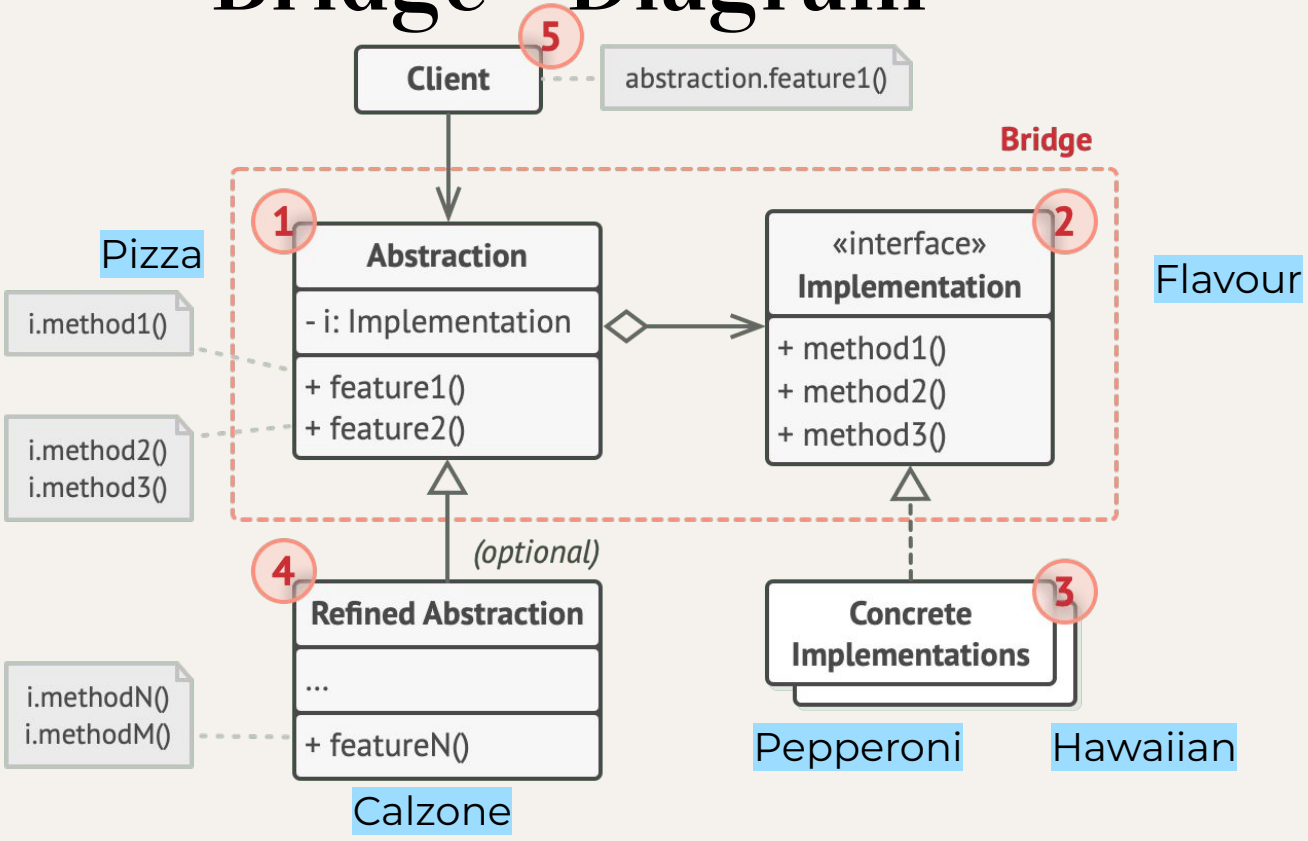
```
class Pepperoni extends Flavor {
 constructor(price: number) {
   super(price);
 }

 /// Prepares the ingredients for the flavor
 public prepare(): void {
   console.log('Add pepperoni');
 }
}
```

```
class Calzone extends Pizza {
 constructor(price: number, topping: string, flavor: Flavor) {
   super(price, topping, flavor);
 }

 public assemble(): void {
   console.log('Preparing dough');
   console.log(`Adding toppings: ${this.topping}`);
   this.flavor.prepare();
   console.log('Folding in half');
   console.log('Baking the calzone');
 }
}
```

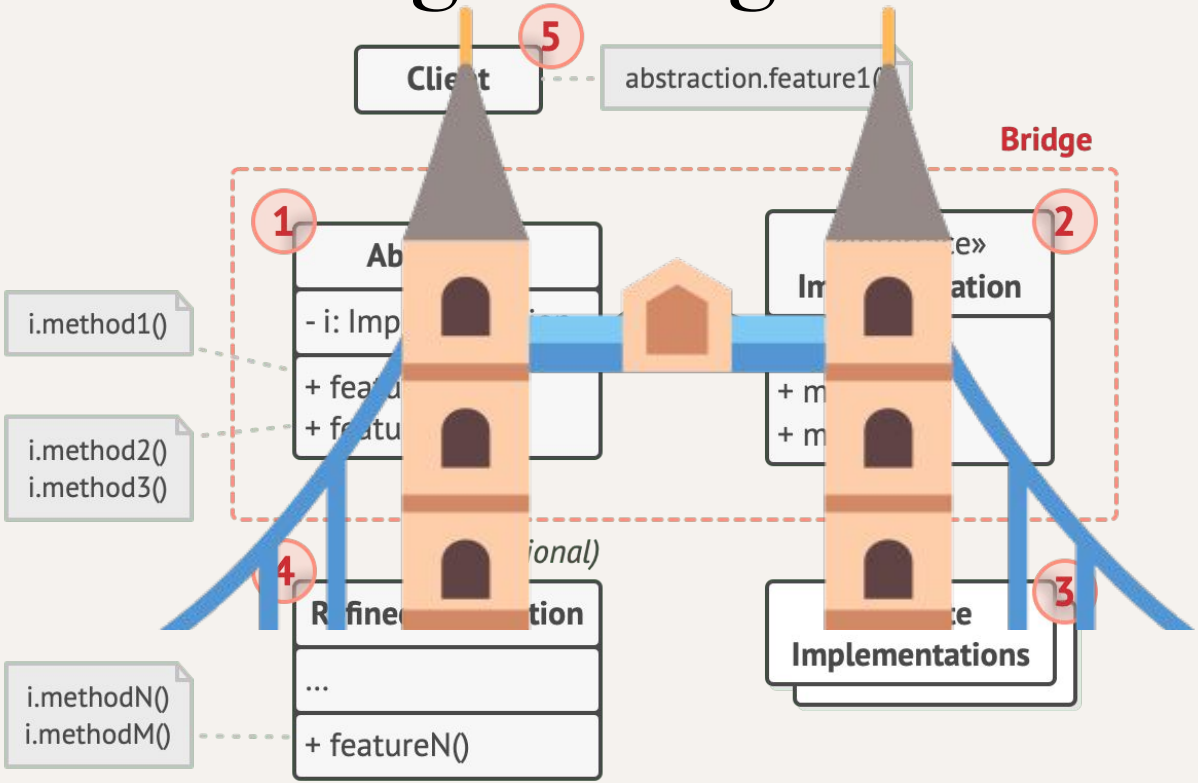
# Bridge - Benefits

- Divide a monolithic class with variants
- Extend a class in several dimensions
- Decouple abstraction and implementation

# Bridge - Diagram



# Bridge - Diagram



# 04
# Behavioral Patterns

# Behavioral Patterns

- Handle communication between objects
  - Distribute responsibilities
  - Improve encapsulation

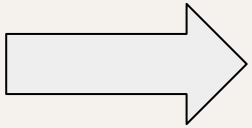
# Observer

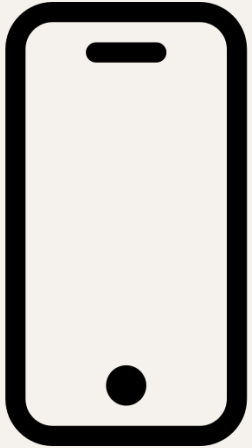
# Observer

- Lets you define a subscription mechanism
- Notify multiple objects about any state changes in the object they're observing
- Useful when a change in one object may require changing other objects

# Observer - Example

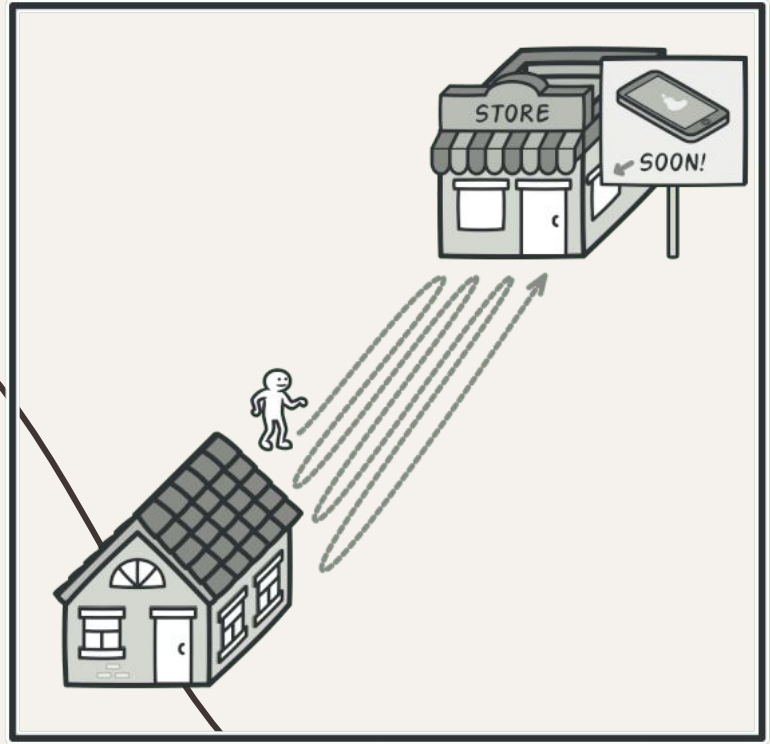
Robert

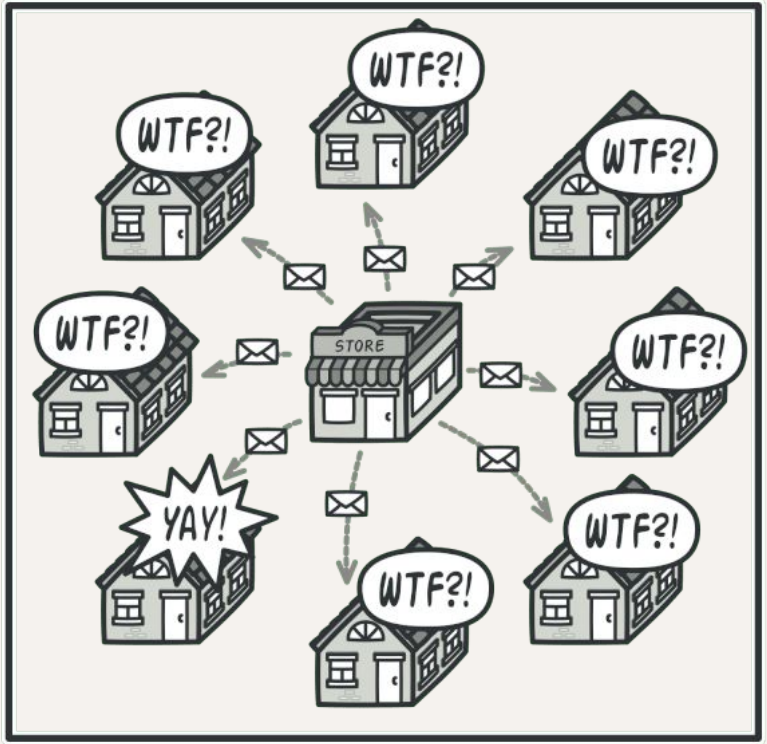




iPhone 16

# Observer - Problem

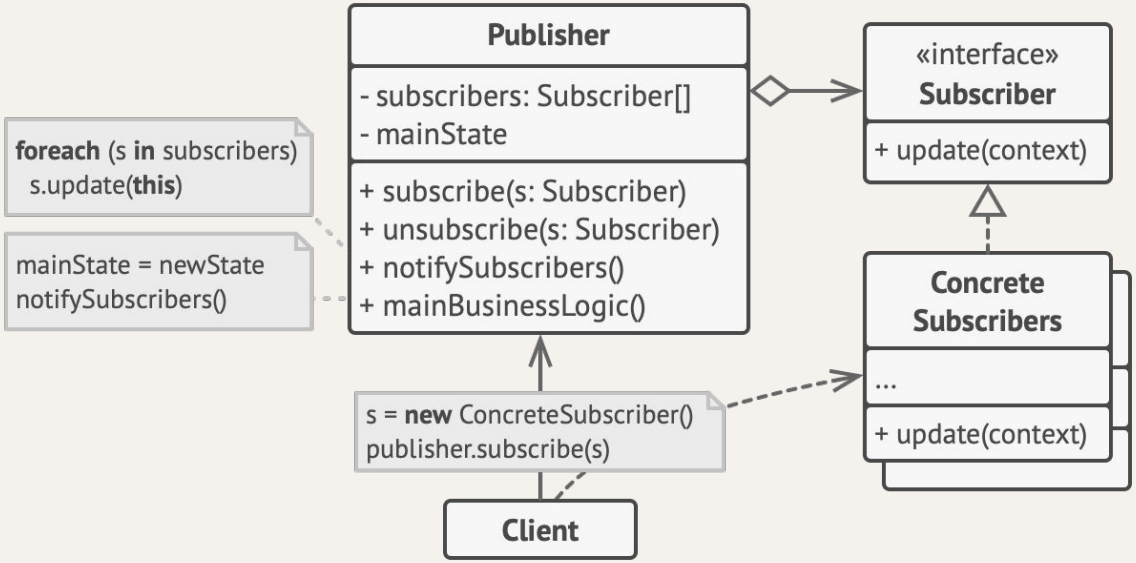


# Observer - Solution

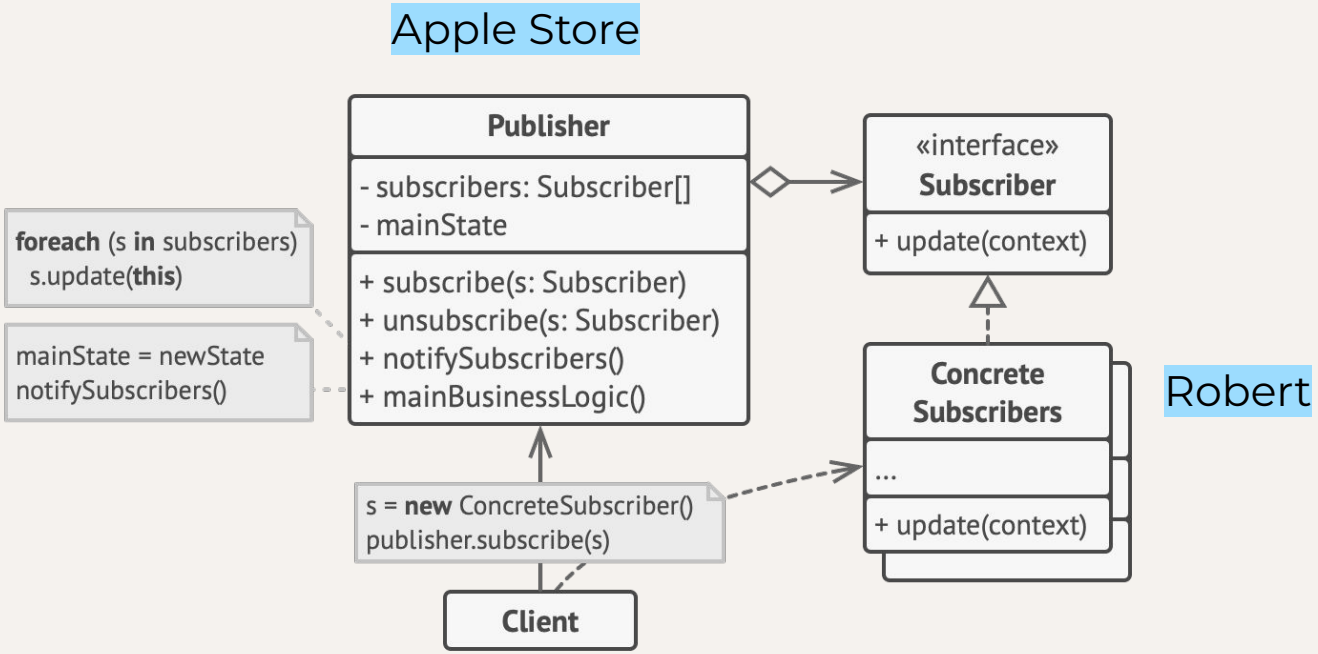
- Separate the code in:
  - A publisher interface, with a subscription mechanism
  - Concrete publishers (observed objects)
  - An interface for subscribers
  - Concrete subscribers (observing objects)

# Observer - Diagram



```
interface Publisher {
 // subscribe an observer to the subject.
 subscribe(subscriber: Subscriber): void;

 // unsubscribe an observer from the subject.
 unsubscribe(subscriber: Subscriber): void;

 // Notify all observers about an event.
 notify(): void;
}

interface Subscriber {
 // Receive update from publisher.
 update(publisher: Publisher): void;
}
```

# Observer - Diagram



# Command

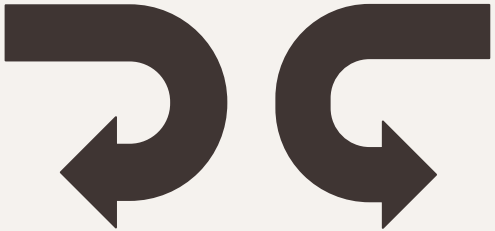
- Encapsulates a request as an object

# Command - Benefits

- Decouple invocation from implementation

- Store Commands
  - Queue or schedule operations
  - Access information about a request
  - Reuse petitions
  - Implement reversible operations

# Command - Example



```
export abstract class PaintCommand {
 /// Applies the effects of the command
 abstract execute(currentPainting: View): void;
}
```

```
class DrawCircleCommand extends PaintCommand {
 execute(currentPainting: View): void {
   currentPainting.drawRandomCircle();
 }
}
```

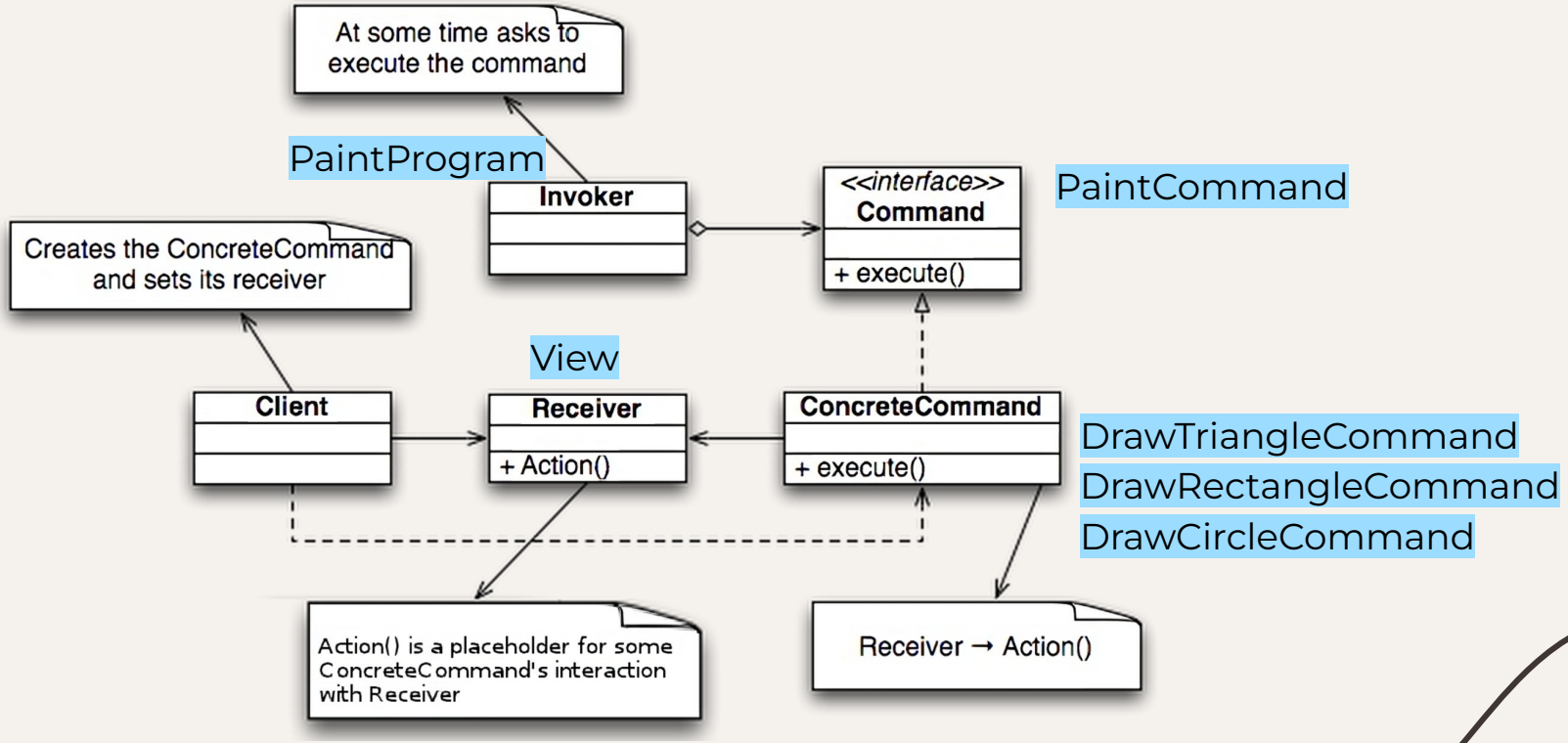
```
class DrawRectangleCommand extends PaintCommand {
 execute(currentPainting: View): void {
   currentPainting.drawRandomRectangle();
 }
}
```

```
class DrawTriangleCommand extends PaintCommand {
 execute(currentPainting: View): void {
   currentPainting.drawRandomTriangle();
 }
}
```

```
export class PaintProgram {
 private commands: PaintCommand[] = [];
 private readonly VIEW: View = new View();
 constructor() {}

 public execute(command: PaintCommand): void {
   command.execute(this.VIEW);
   this.commands.push(command);
 }
 // ...
}
```

```
export class PaintProgram {
 /// ...
 public undo(): void {
   this.commands.pop();  /// Remove last command
   this.VIEW.clear();    /// Reset view to original state

   /// Repeat the commands
   for (let currentComand of this.commands) {
     currentComand.execute(this.VIEW);
   }
 }
}
```

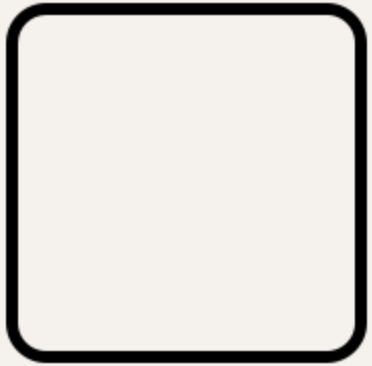
# Command - Diagram



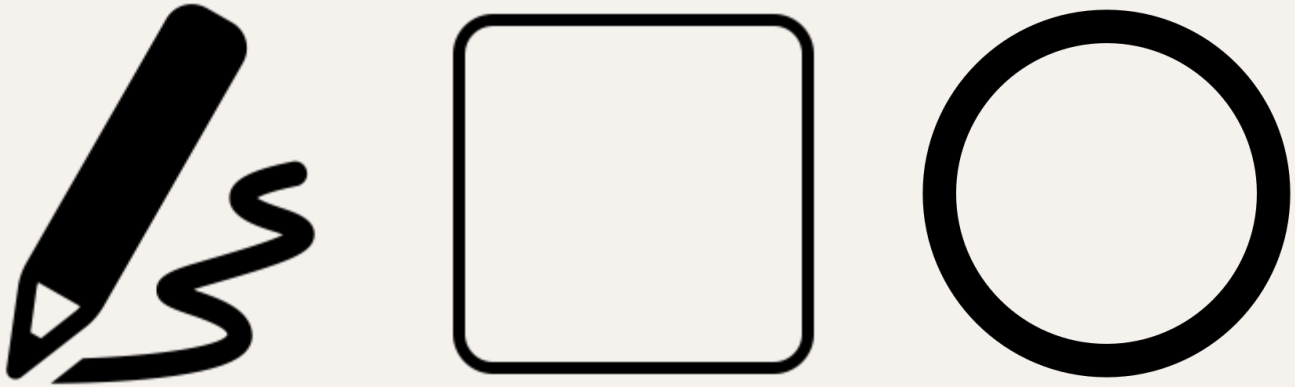
# Strategy

- Lets you define a family of algorithms
- We put each of them in a separate class and make their objects interchangeable
- We can switch between algorithms during runtime

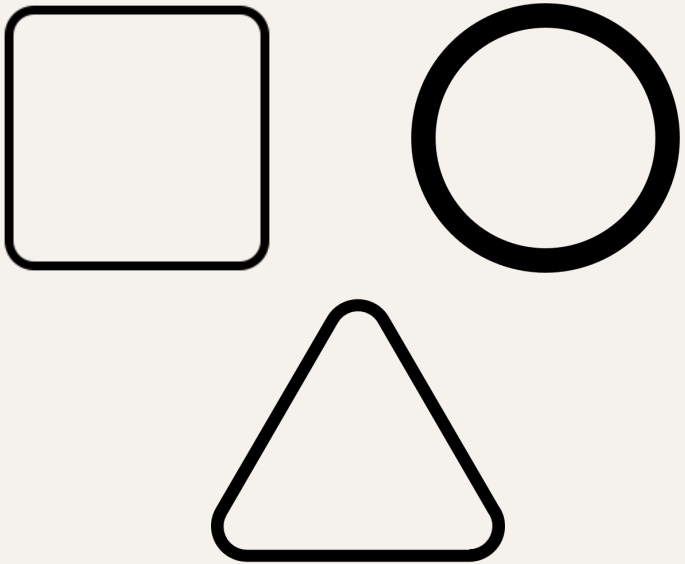
# Strategy - Example



# Strategy - Example



# Strategy - Example



```
class View {
 /* The shape to draw */
 private shape: Shape = Shape.CIRCLE;

 /**
  * Set the shape to draw
  * @param shape shape chosen by the user
  */
 setShape(shape: Shape) {
     this.shape = shape;
 }
```

```
drawShape() {
    if (this.shape == Shape.CIRCLE) {
      console.log('Drawing a circle');
      // Draw a circle
    } else if (this.shape == Shape.SQUARE) {
      console.log('Drawing a square');
      // Draw a square
    } else if (this.shape == Shape.TRIANGLE) {
      console.log('Drawing a triangle');
      // Draw a triangle
    } else if (this.shape == Shape.RECTANGLE) {
      console.log('Drawing a rectangle');
      // Draw a rectangle
    } else {
        throw new Error("Invalid shape");
    }
```

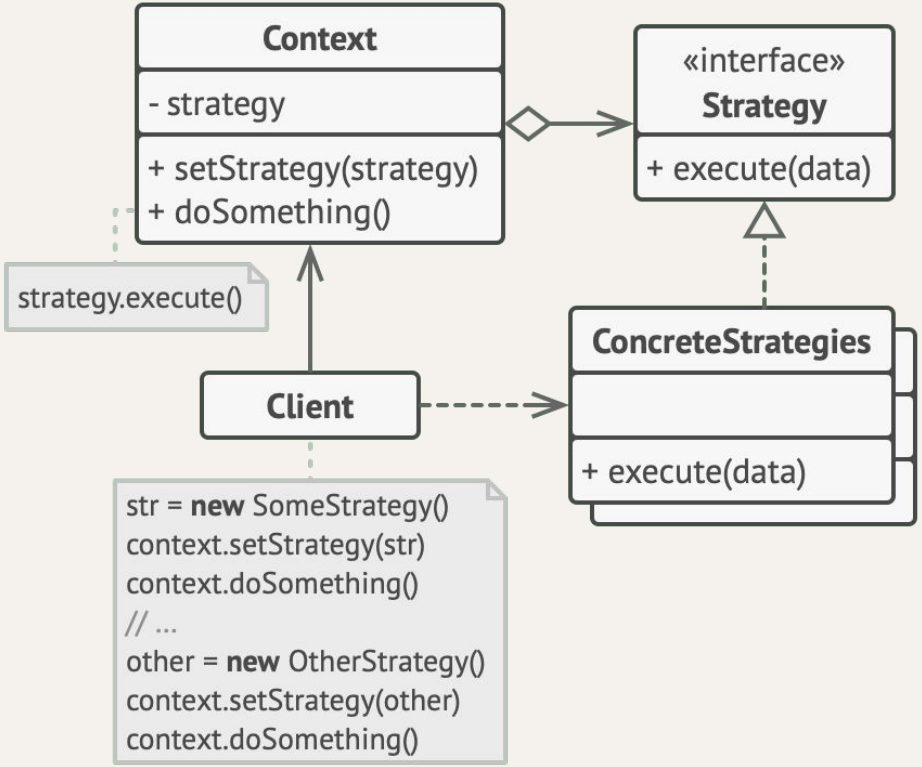
# Strategy - Problems

- The class makes something specific in different ways
- Unscalable code. Exponential growth!
- Violates the Open-Closed and SRP principles

# Strategy - Solution

- Separate all the algorithms into separate classes called strategies
- We create a generic interface for the strategies
- We pass the strategy we want to use to the context class, so it can delegate the execution to the chosen algorithm

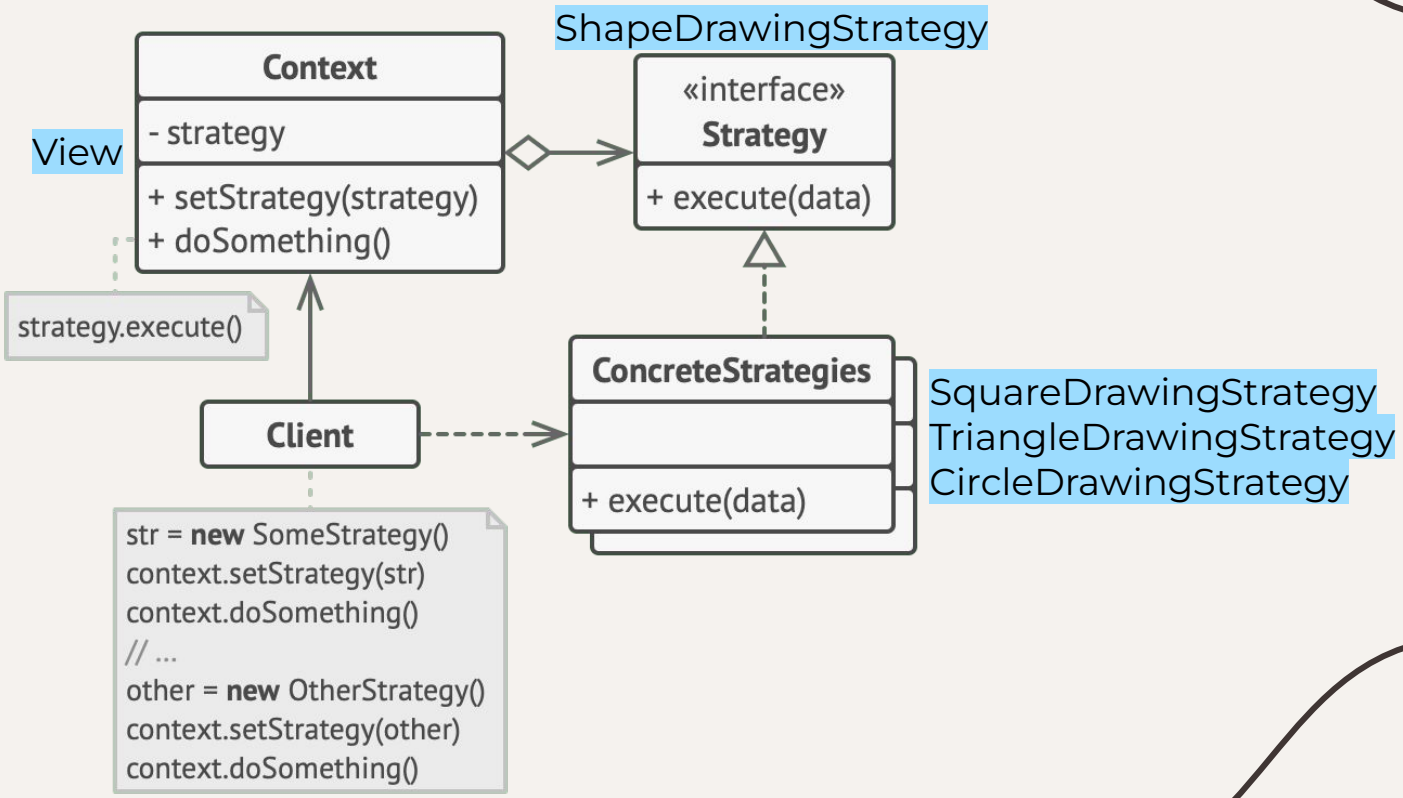
# Strategy - Diagram



```
class View {

 constructor(private drawingStrategy: ShapeDrawingStrategy) {}

 public setStrategy(drawingStrategy: ShapeDrawingStrategy) {
   this.drawingStrategy = drawingStrategy;
 }

 public drawShapes(): void {
   console.log('Context: Drawings shapes (not sure how it will
do it)');
   this.drawingStrategy.draw();
 }
}
```

```
interface ShapeDrawingStrategy {
 draw(): void;
}

class SquareDrawingStrategy implements ShapeDrawingStrategy {
 public draw(): void {
   console.log('Drawing a square');
   // Draw a square
 }
}

// Other shapes
```

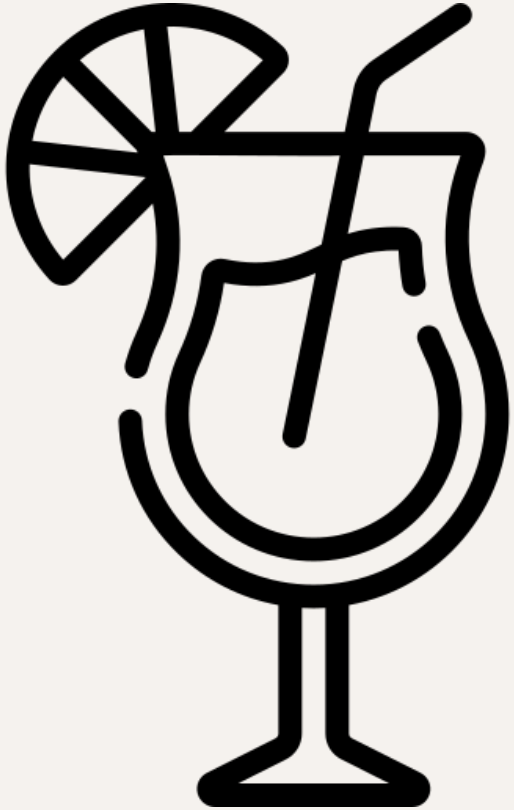
# Strategy - Diagram



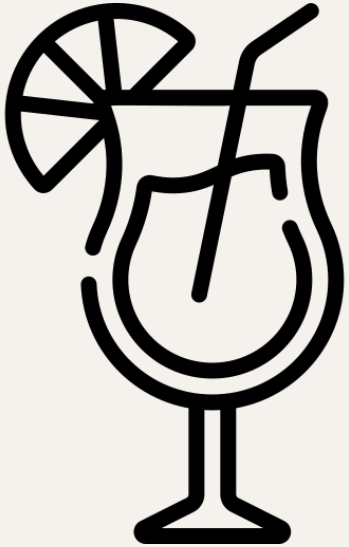
# Template method ★★☆

- Defines the skeleton of an algorithm in the superclass
- Lets subclasses override specific steps of the algorithm without changing its structure.
- Uses polymorphism

# Template method - Example



# Template method - Example



```
class Caipirinha {
 public prepare(): void {
   console.log('Take a glass')
   console.log('Add ice')
   console.log('Add sugar')
   console.log('Add lime')
   console.log('Add cachaca')
   console.log('Stir')
   console.log('Add a straw')
 }
}
```

```
class Mojito {
 public prepare(): void {
   console.log('Take a glass')
   console.log('Add ice')
   console.log('Add sugar')
   console.log('Add lime')
   console.log('Add rum’)
   console.log('Add mint)
   console.log('Stir')
   console.log('Add a straw')
 }
}
```

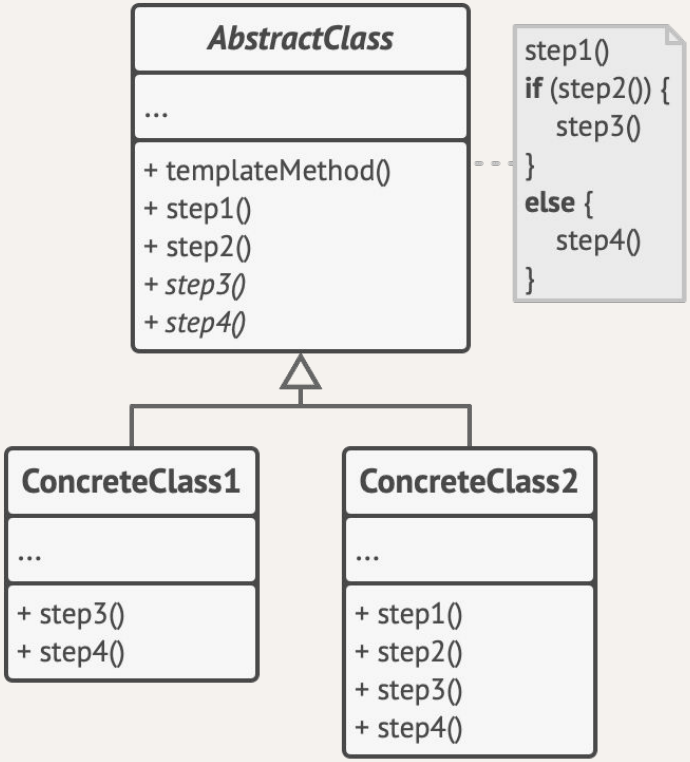
# Template method - Problems

- Duplicated code, a lot of duplicated code

# Template method - Solution

- Break down the algorithm into a series of steps
- Create a template method that puts together those steps in an abstract superclass
- Create as many subclasses as algorithms you need to implement

# Template method - Diagram



```
abstract class Cocktail {
 /**
  * The template method defines the skeleton of an
algorithm.
  */
 public prepare(): void {
   this.addGlass();
   this.addIce();
   this.addAlcohol();
   this.addFruit();
   this.addExtra();
   this.stir();
   this.addStraw();
 }
```

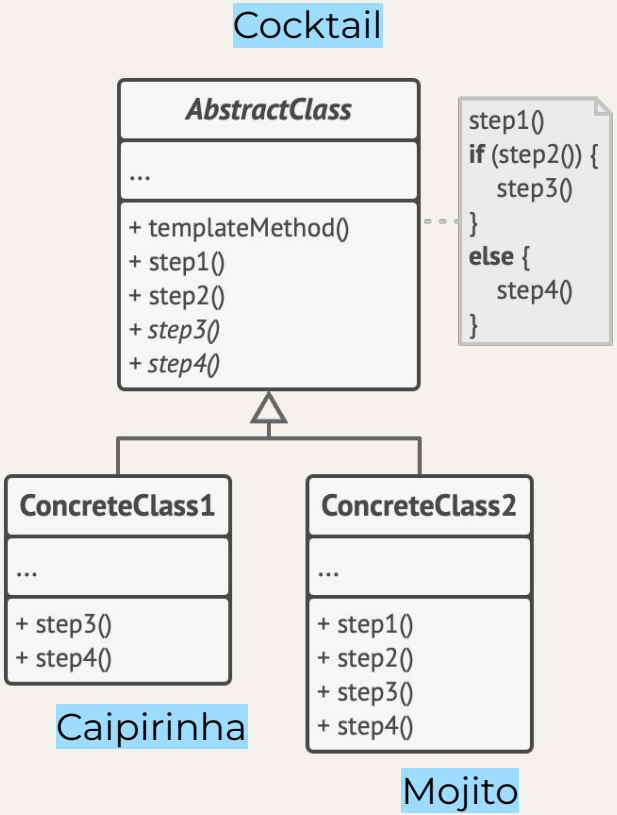
```
protected addGlass(): void {
  console.log('Adding a glass');
}

protected addIce(): void {
  console.log('Adding ice');
}
 protected stir(): void {
  console.log('Stiring');
}

protected addStraw(): void {
  console.log('Adding a straw');
}
```

```
/**
 * These operations have to be implemented in
subclasses.
 */
protected abstract addAlcohol(): void;
protected abstract addFruit(): void;
protected abstract addExtra(): void;
```

```
class Caipirinha extends Cocktail {

 protected addAlcohol(): void {
   console.log('Adding cachaca');
 }

 protected addFruit(): void {
   console.log('Adding lime');
 }

 protected addExtra(): void {
   console.log('Adding sugar');
 }
}
```

```
class Mojito extends Cocktail {

protected addAlcohol(): void {
  console.log('Adding rum);
}

protected addFruit(): void {
  console.log('Adding lime');
}

protected addExtra(): void {
  console.log('Adding sugar');
  console.log('Adding mint);
}
}
```

# Template method - Diagram



# 05
# Usage considerations

# Patterns are great, but…

“If all you have is a hammer, everything looks like a nail”

–Abraham Maslow

# The most important things

- The programmer’s criteria always comes first
- Do not believe everything, try it first
- PRACTICE!

# 06

# Bibliography

- Head First Design Patterns - O'Reilly
  https://learning.oreilly.com/library/view/head-first-design/0596007124/

- Refactoring Guru: patterns examples in typescript
  https://refactoring.guru/design-patterns/typescript

- DoFactory: Design Patterns theory and popularity
  https://www.dofactory.com/net/design-patterns

- Design patterns elements of reusable object-oriented software - GoF
  https://absysnet.bbtk.ull.es/cgi-bin/abnetopac?TITN=8809

- Robert C. Martin - Videos on Design Patterns / Clean Code
  https://learning.oreilly.com/course/design-patterns-clean/9780135485965/

- Geeks for geeks - Various patterns examples in JS
  https://www.geeksforgeeks.org/javascript-design-patterns/?ref=lbp

- Design patterns - Wikipedia
  https://en.wikipedia.org/wiki/Software_design_pattern

- Source Making: theory and diagrams on design patterns
  https://sourcemaking.com/design_patterns

# Any questions?

igor.dragone.13@ull.edu.es

jose.morera.27@ull.edu.es

# Thanks for watching!