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

1
Introduction

2
Creational Patterns

3
Structural Patterns

4
Behavioral Patterns

5
Usage considerations

6
Bibliography

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

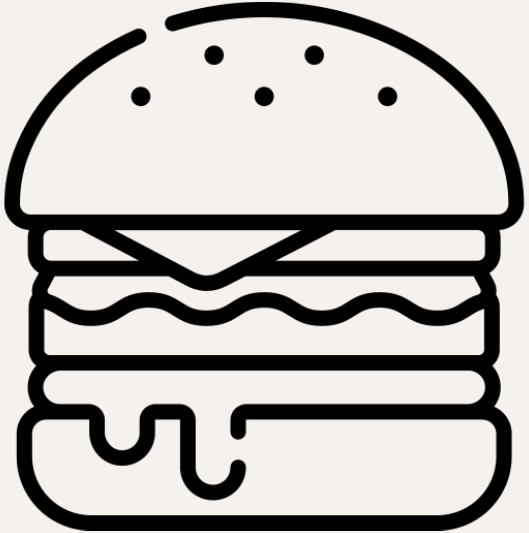
Creational Patterns

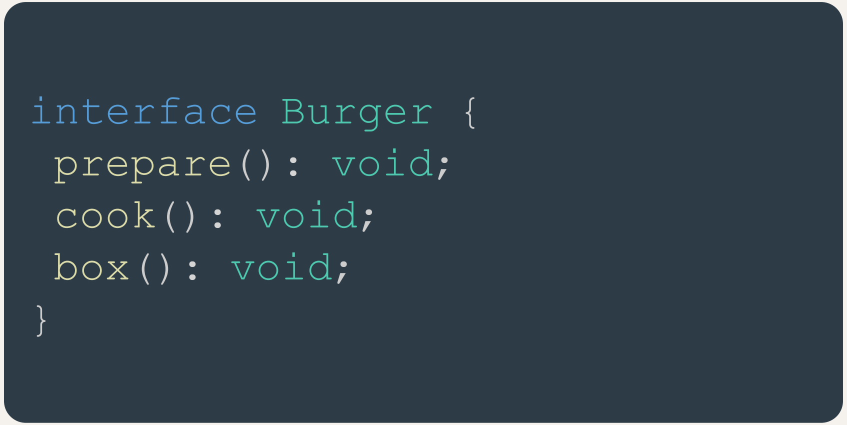
- Provide object creation mechanisms
    - Encapsulate knowledge about construction
    - Increase flexibility

Factory method ★★★

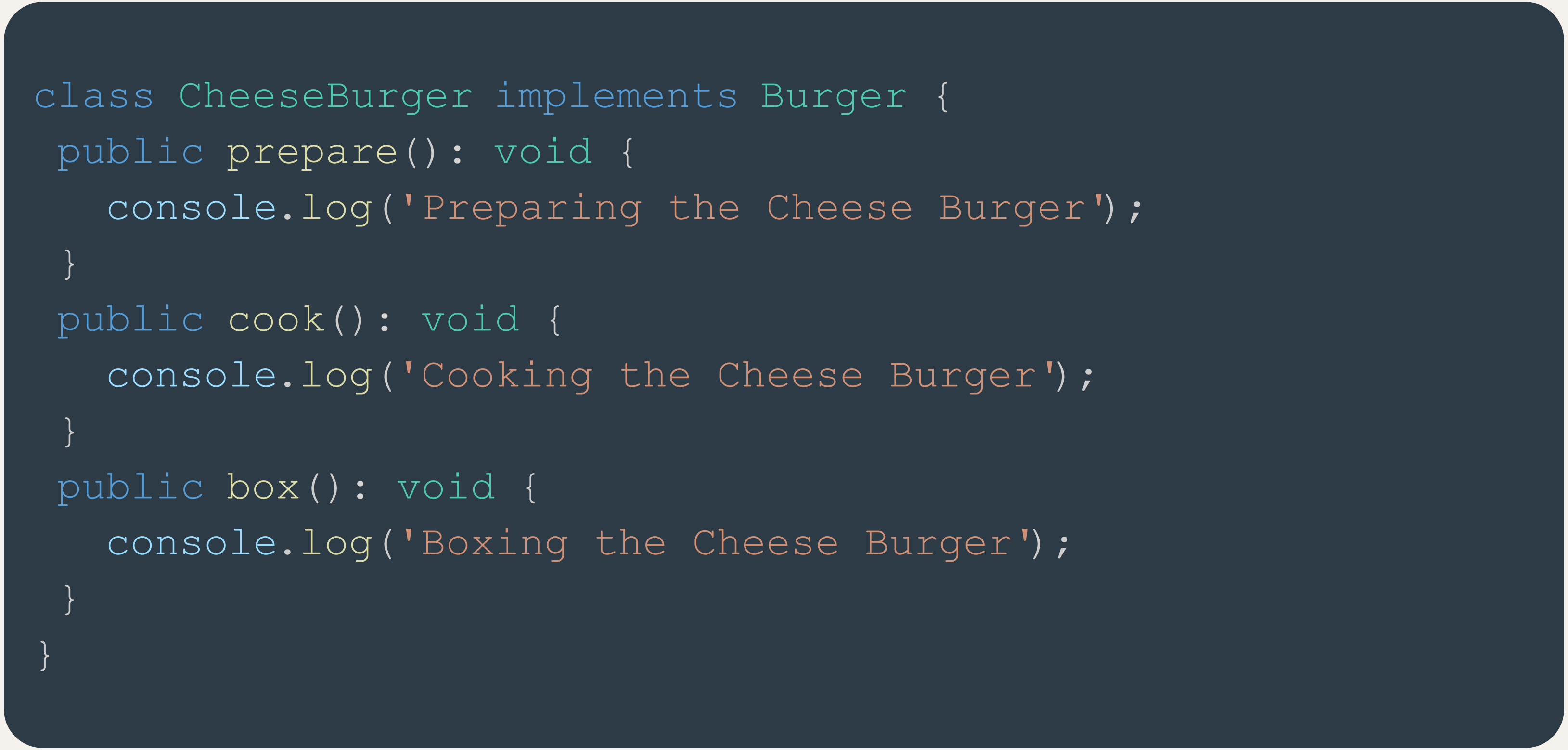
- Interface for creating objects
- Different object types may be created

Factory method - Example

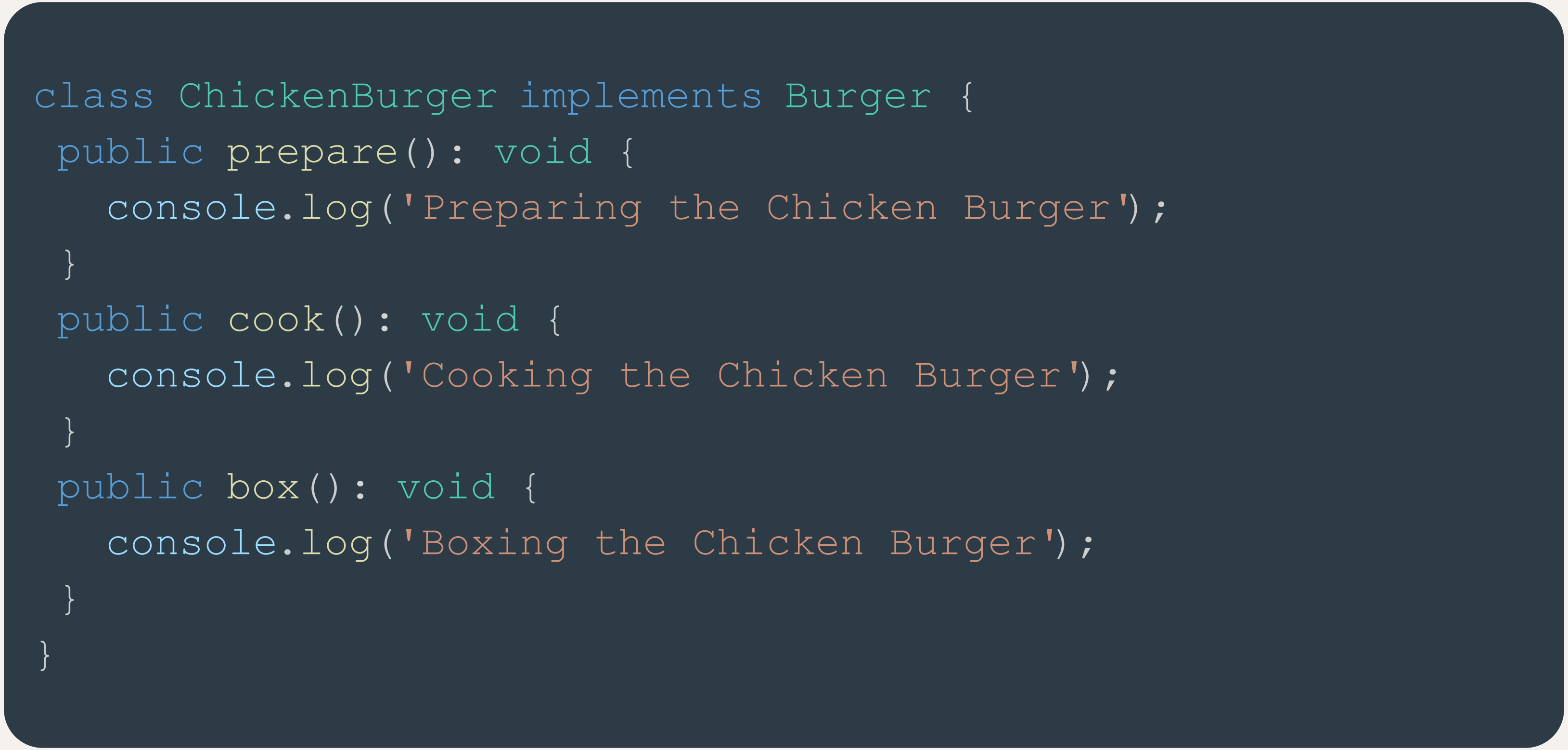




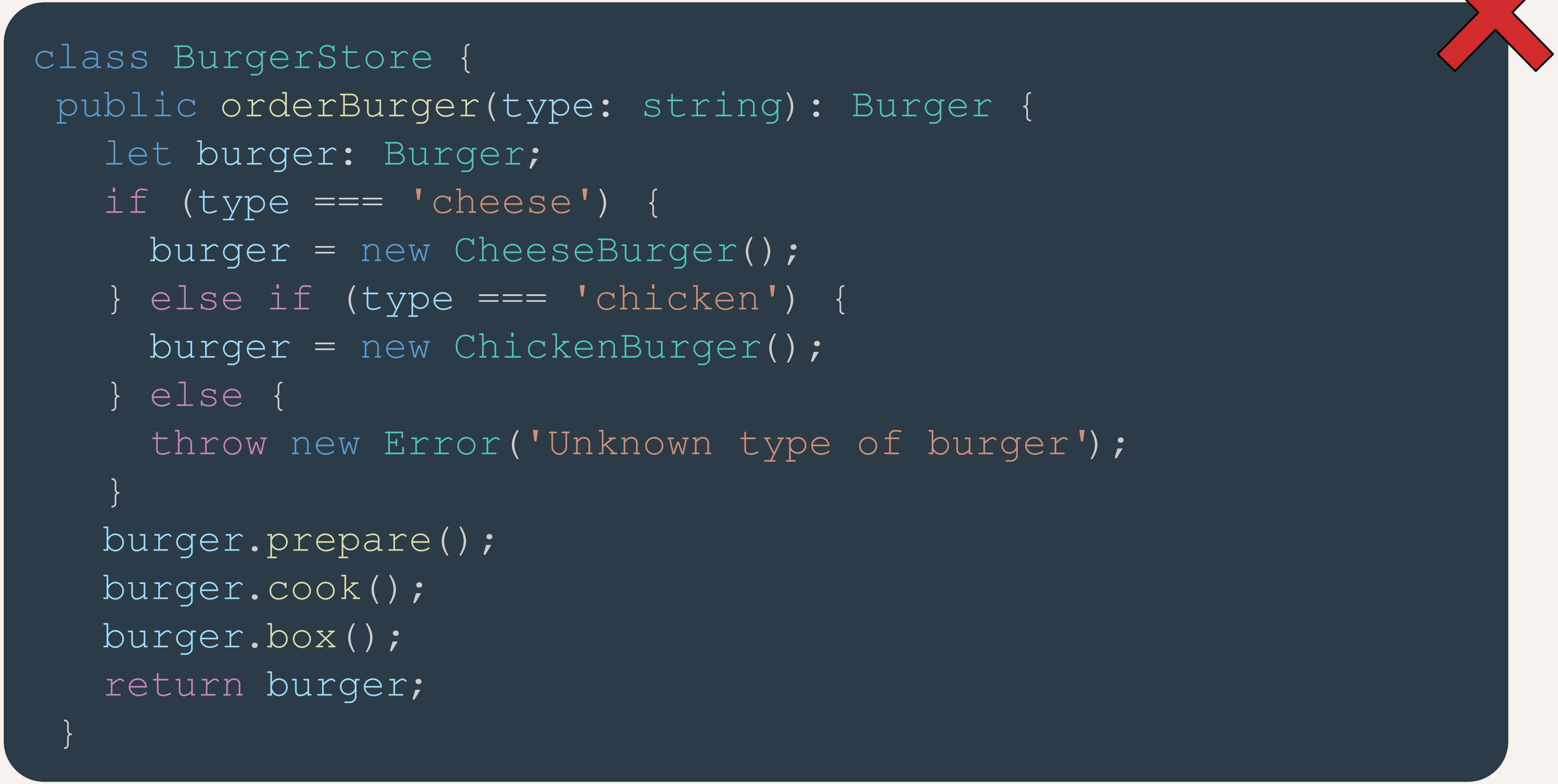
```
interface Burger {
 prepare(): void;
 cook(): void;
 box(): void;
}
```



```
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;
   if (type === 'cheese') {
     burger = new CheeseBurger();
   } else if (type === 'chicken') {
     burger = new ChickenBurger();
   } else {
     throw new Error('Unknown type of burger');
   }
   burger.prepare();
   burger.cook();
   burger.box();
   return burger;
 }
```

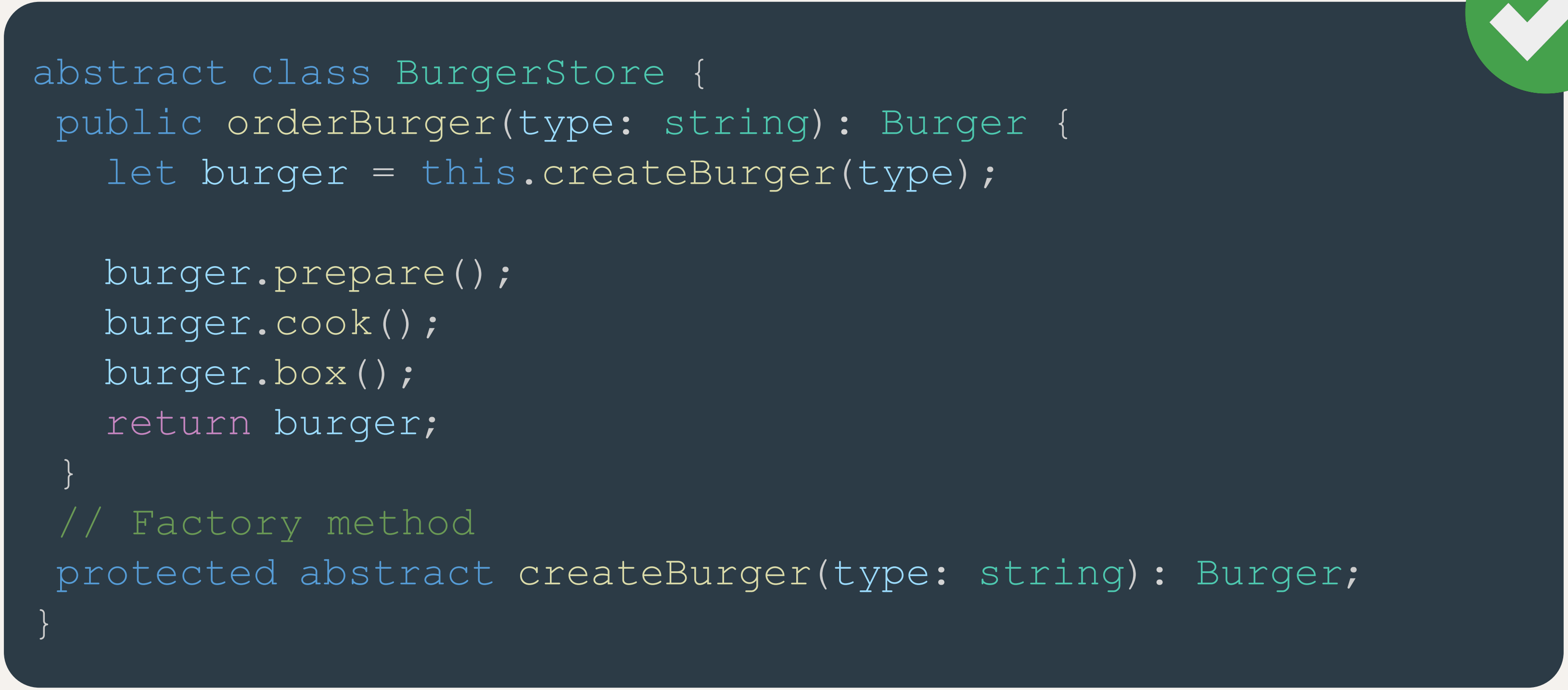
Factory method - Problems

- Violates Open/Closed principle
- Violates dependency inversion principle

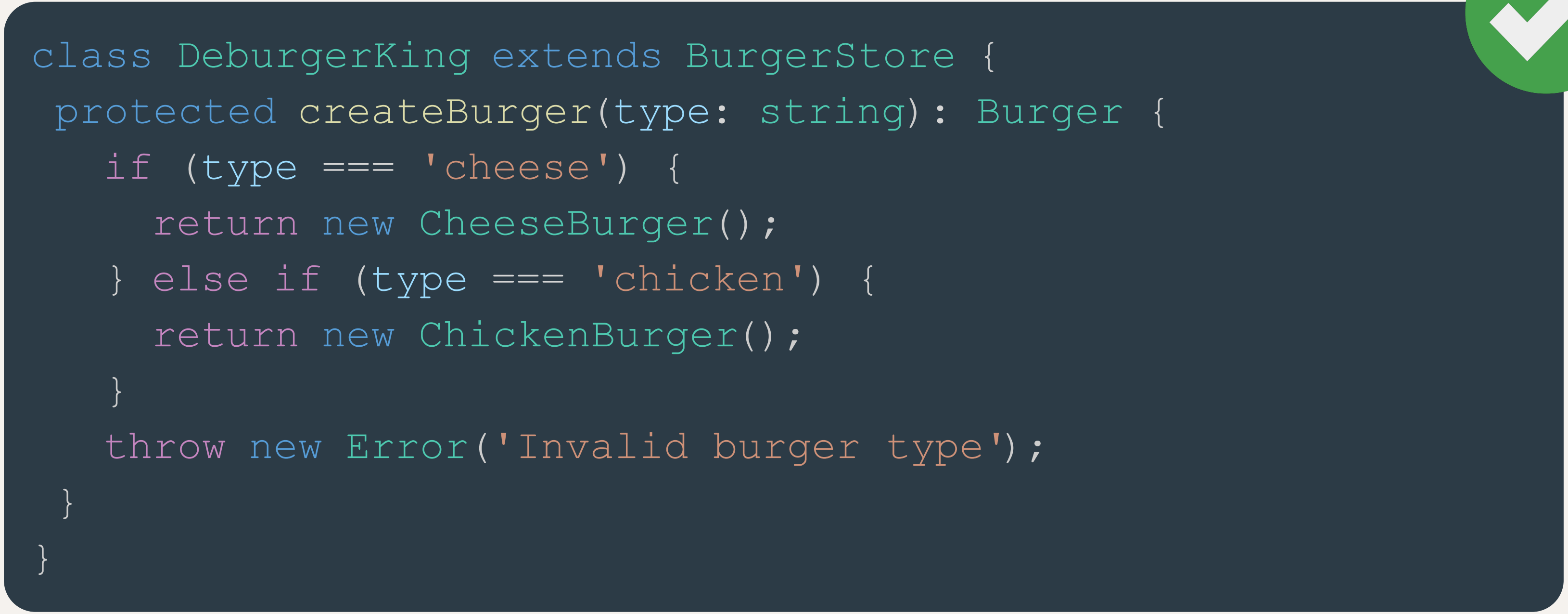
Factory method - Solution

- Delegate object creation
- Factory method may be overridden

✓

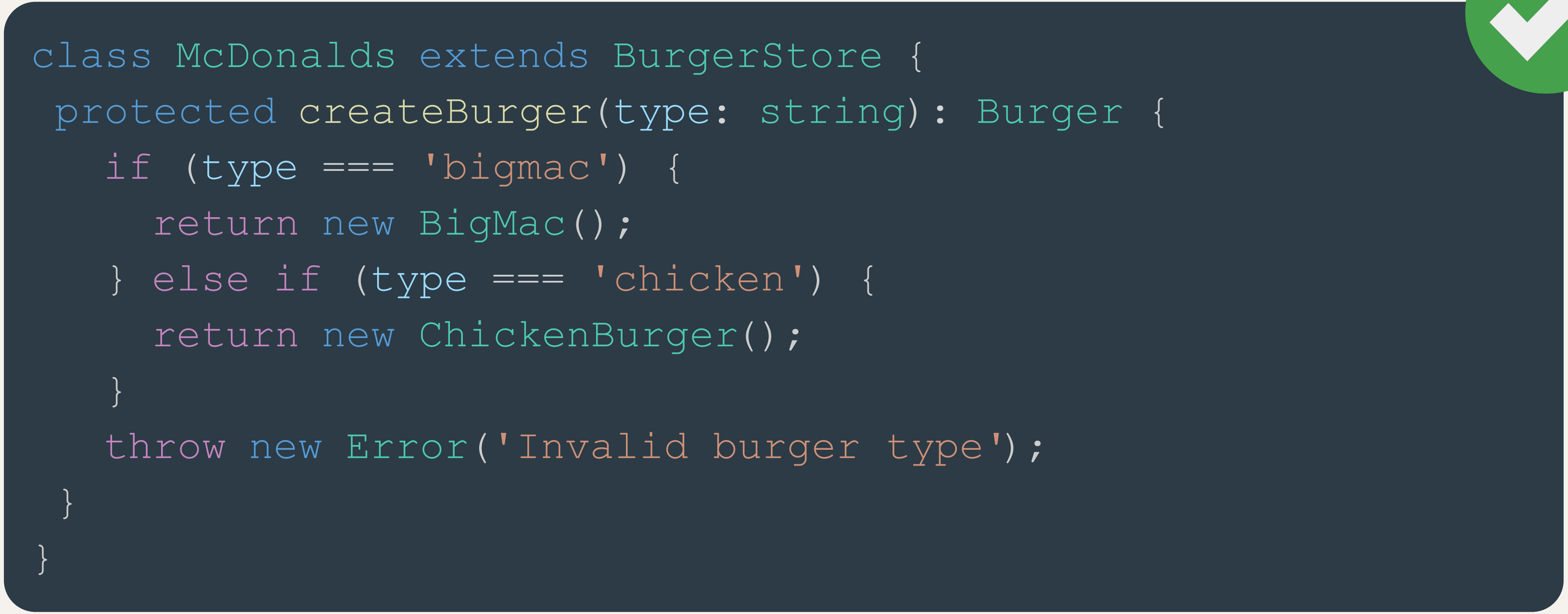


```
abstract class BurgerStore {
 public orderBurger(type: string): Burger {
    let burger = this.createBurger(type);

    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 {
   if (type === 'cheese') {
     return new CheeseBurger();
   } else if (type === 'chicken') {
     return new ChickenBurger();
   }
   throw new Error('Invalid burger type');
 }
}
```

✓

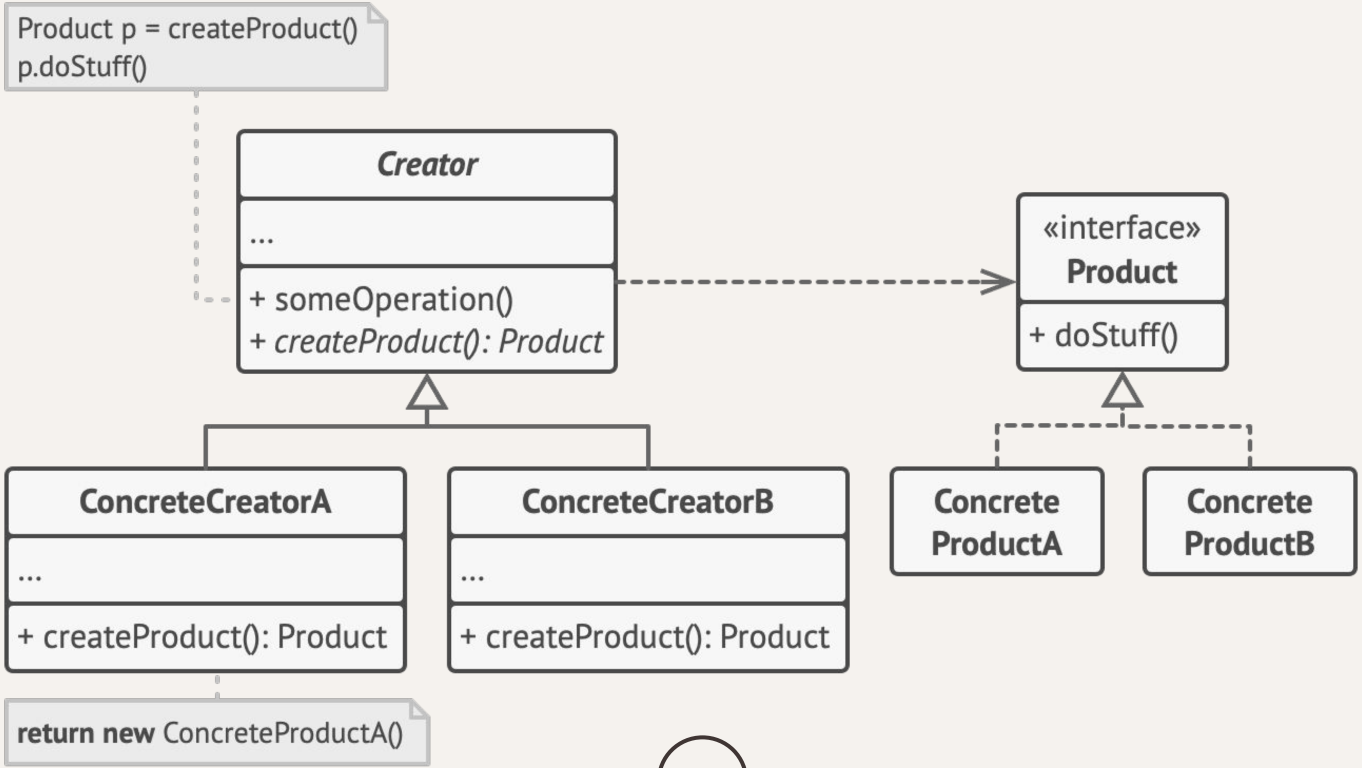


```
class McDonalds extends BurgerStore {
 protected createBurger(type: string): Burger {
   if (type === 'bigmac') {
     return new BigMac();
   } else if (type === 'chicken') {
     return new ChickenBurger();
   }
   throw new Error('Invalid burger type');
 }
}
```

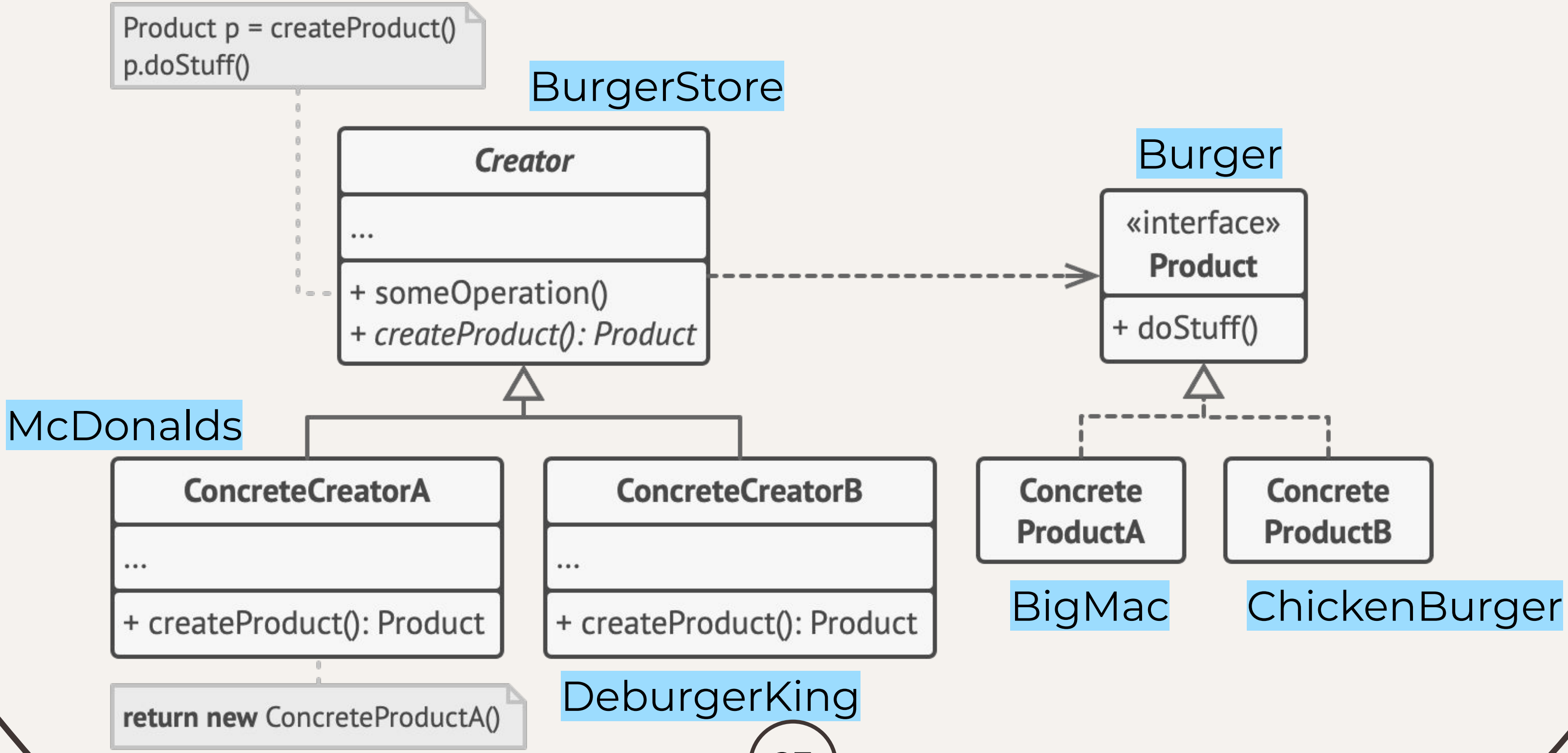
Factory method - Benefits

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

Factory method - Diagram



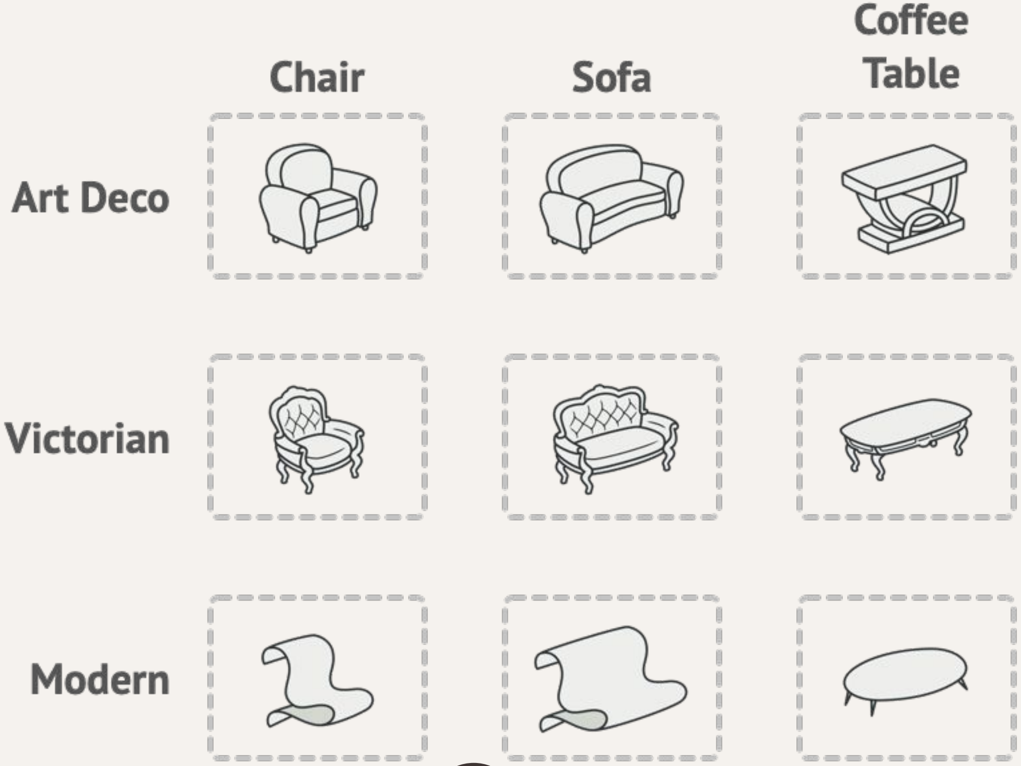
Factory method - Diagram

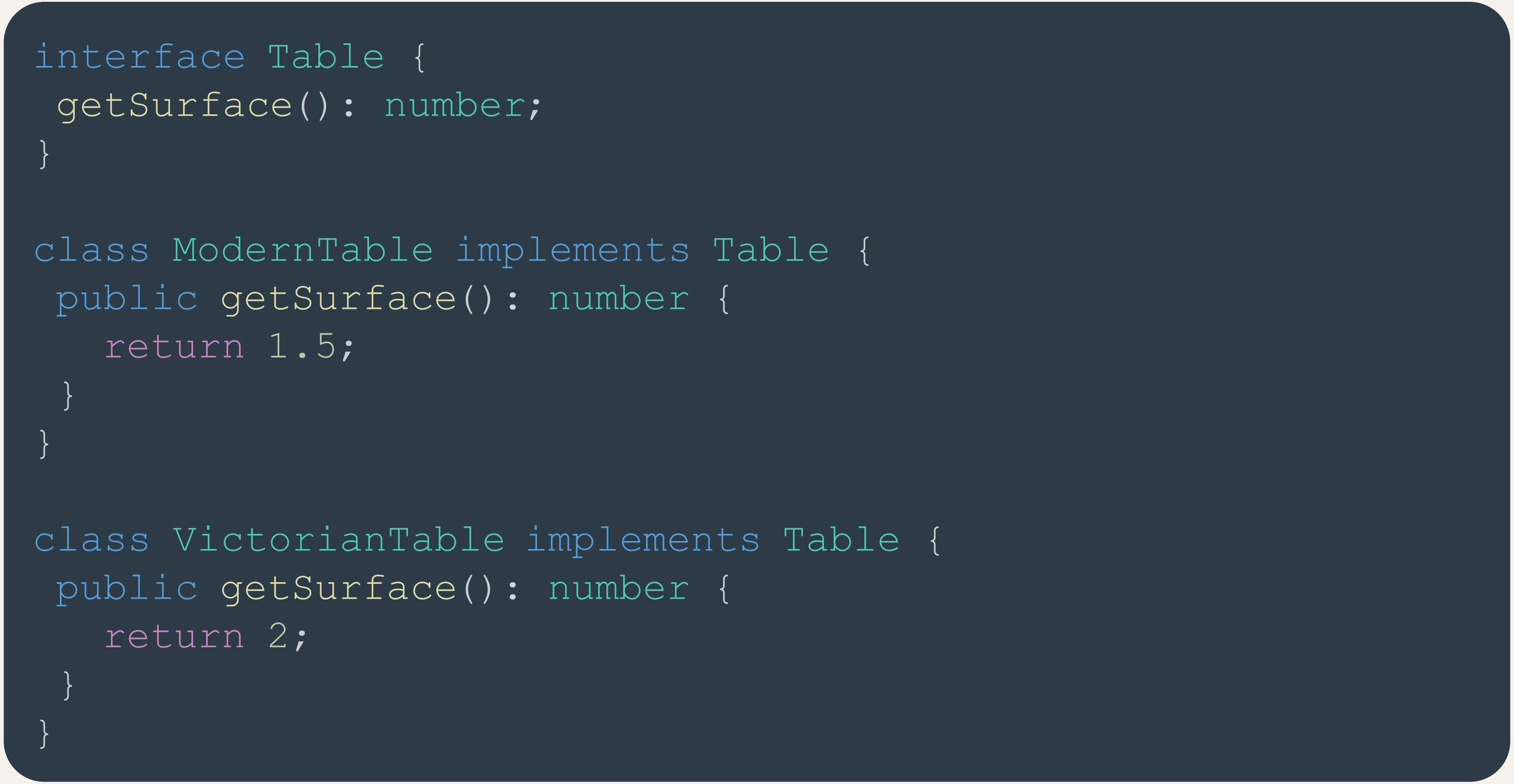


Abstract Factory ★★★

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

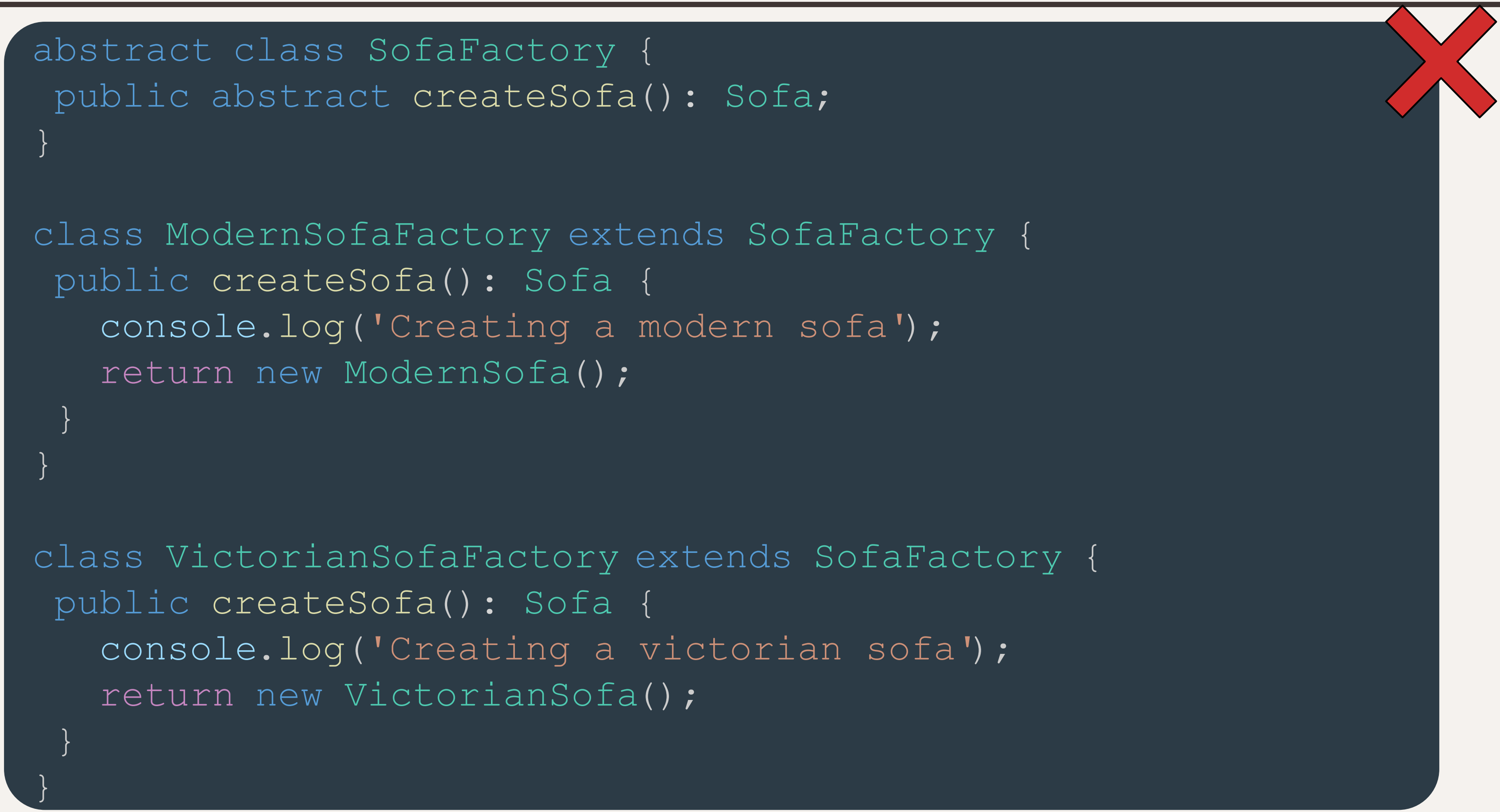
Abstract Factory - Example





```
interface Table {
 getSurface(): number;
}

class ModernTable implements Table {
 public getSurface(): number {
   return 1.5;
 }
}

class VictorianTable implements Table {
 public getSurface(): number {
   return 2;
 }
}
```

```
interface Sofa {
 getNumberOfSeats(): number;
}

class ModernSofa implements Sofa {
 public getNumberOfSeats(): number {
   return 3;
 }
}

class VictorianSofa implements Sofa {
 public getNumberOfSeats(): number {
   return 4;
 }
}
```

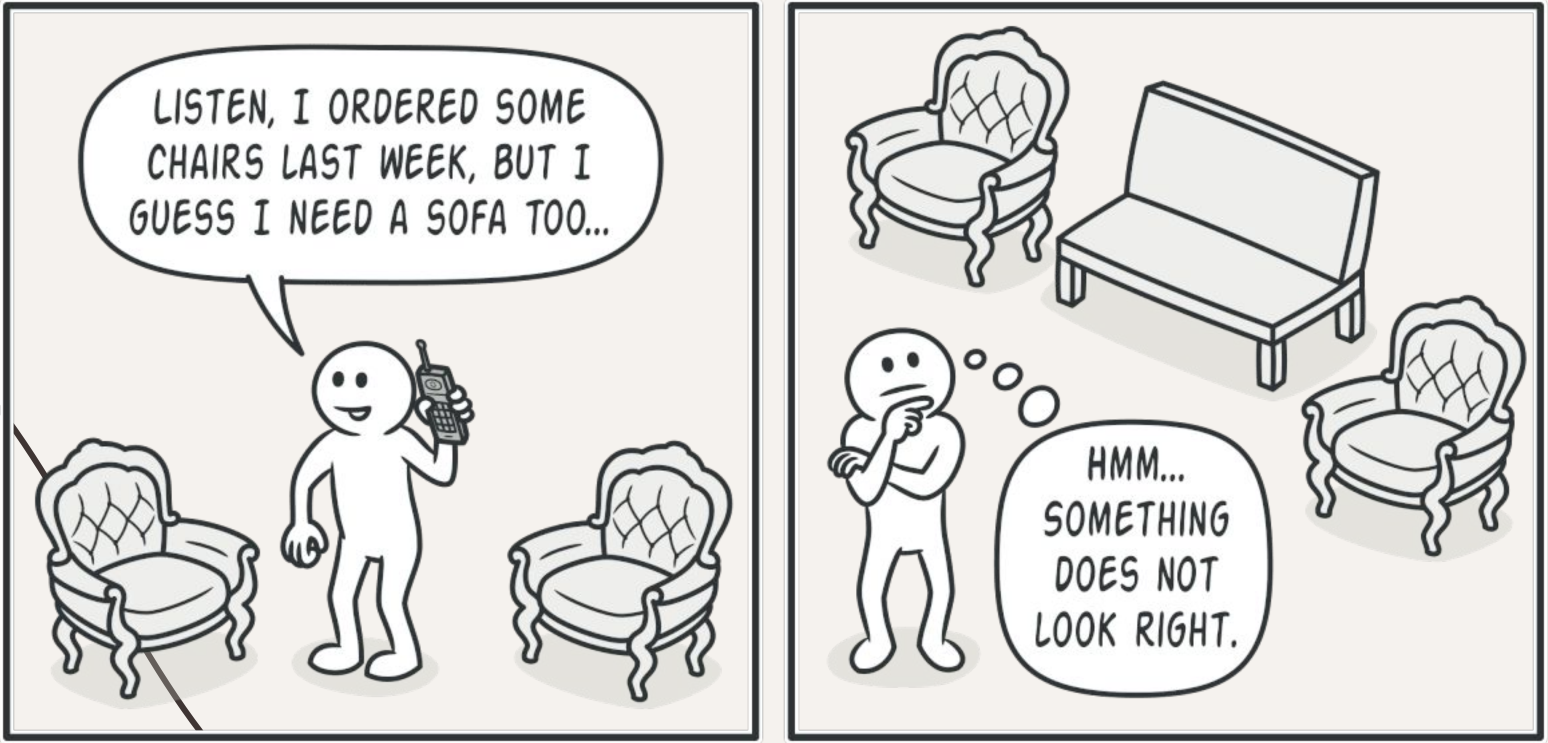


```
abstract class SofaFactory {
 public abstract createSofa(): Sofa;
}

class ModernSofaFactory extends SofaFactory {
 public createSofa(): Sofa {
   console.log('Creating a modern sofa');
   return new ModernSofa();
 }
}

class VictorianSofaFactory extends SofaFactory {
 public createSofa(): Sofa {
   console.log('Creating a victorian sofa');
   return new VictorianSofa();
 }
}
```

```
abstract class TableFactory {
 public abstract createTable(): Table;
}

class ModernTableFactory extends TableFactory {
 public createTable(): Table {
   console.log('Creating a modern table');
   return new ModernTable();
 }
}

class VictorianTableFactory extends TableFactory {
 public createTable(): Table {
   console.log('Creating a victorian table');
   return new VictorianTable();
 }
}
```

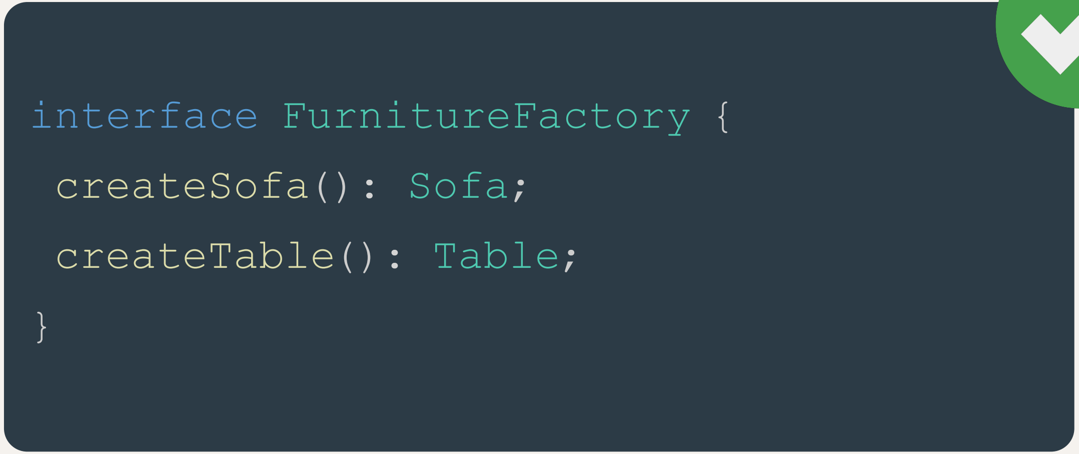
```
export function main(): void {
 let loungeTable = new ModernTableFactory().createTable();
 /** Imagine some other code */
 let loungeSofa = new VictorianSofaFactory().createSofa();
 // Oh no, we messed up and did not match funiture!
}
```

Abstract Factory - Problems


LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

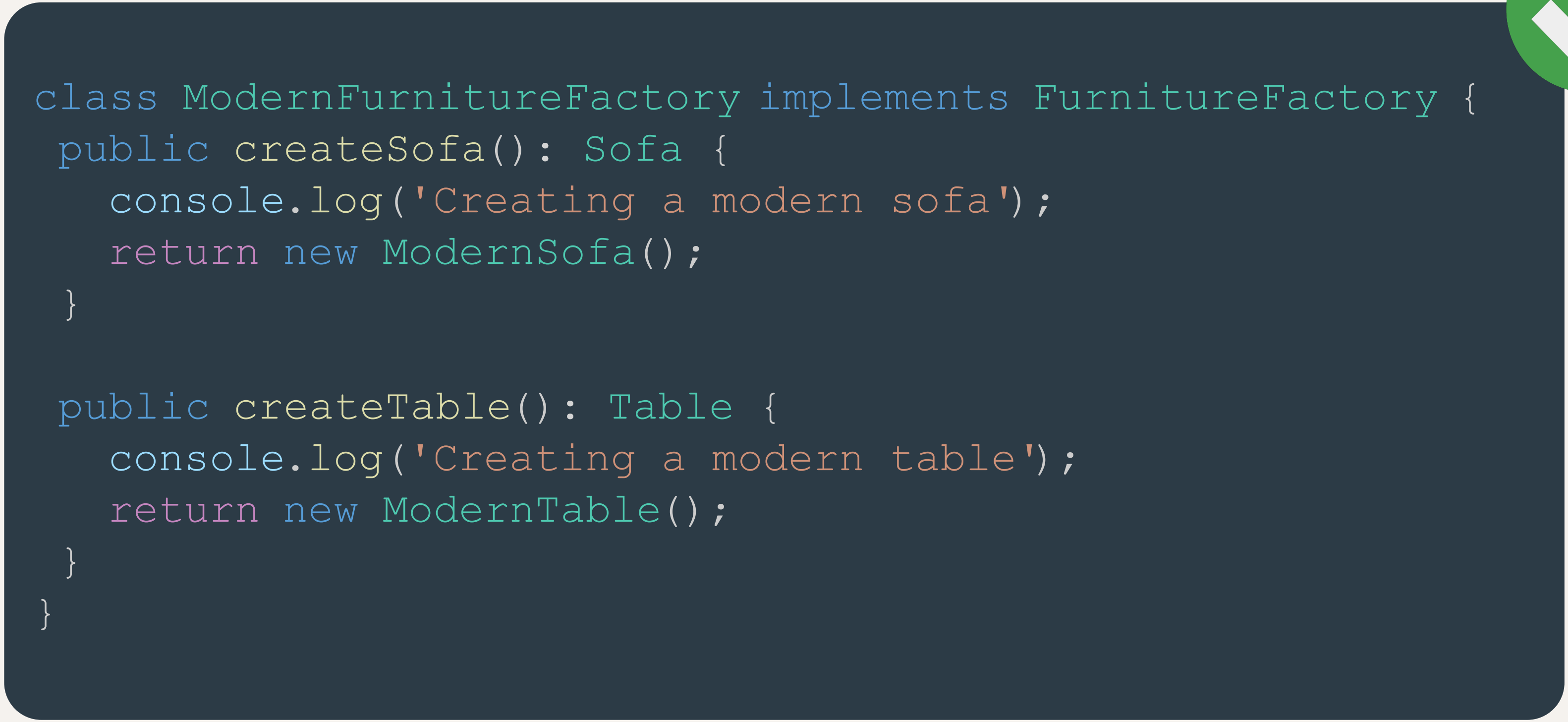
Abstract Factory - Solution

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



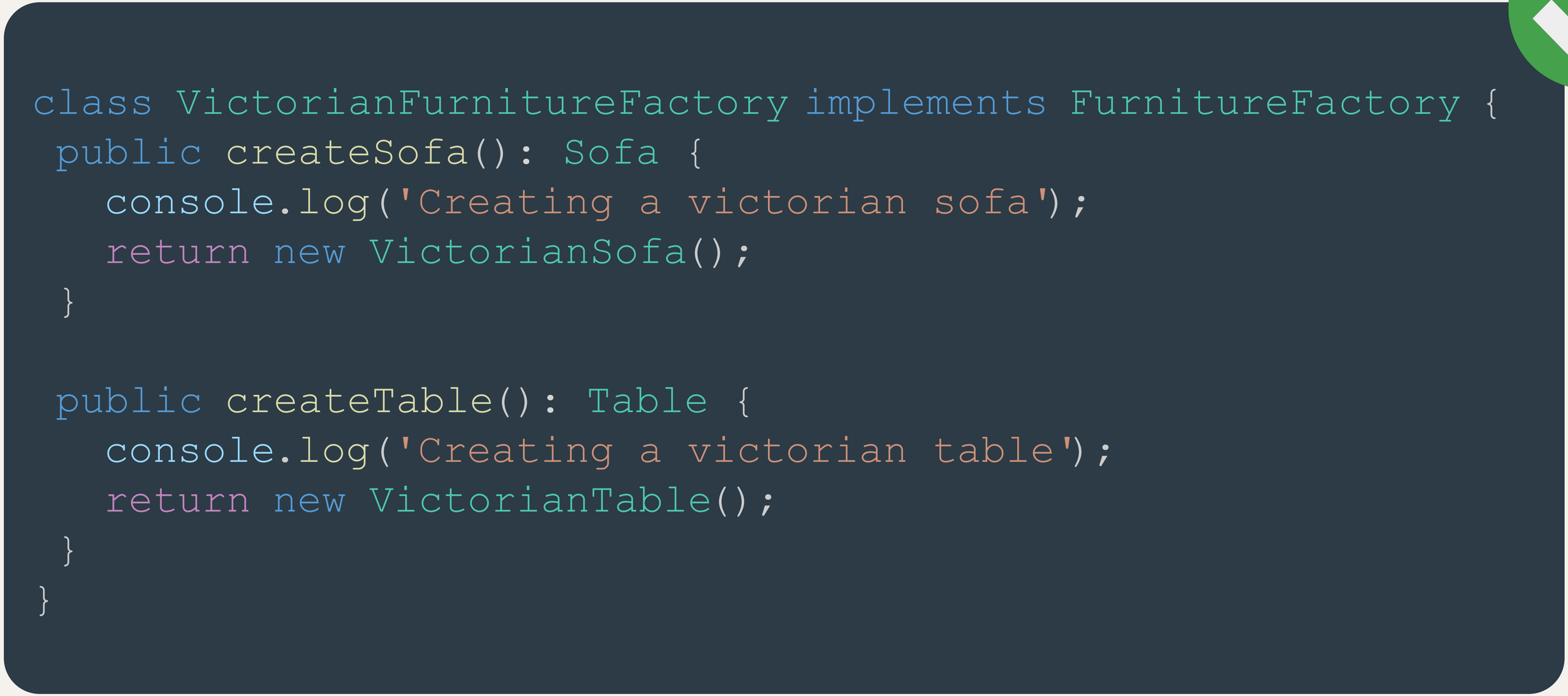
```
interface FurnitureFactory {
 createSofa(): Sofa;
 createTable(): Table;
}
```

✓



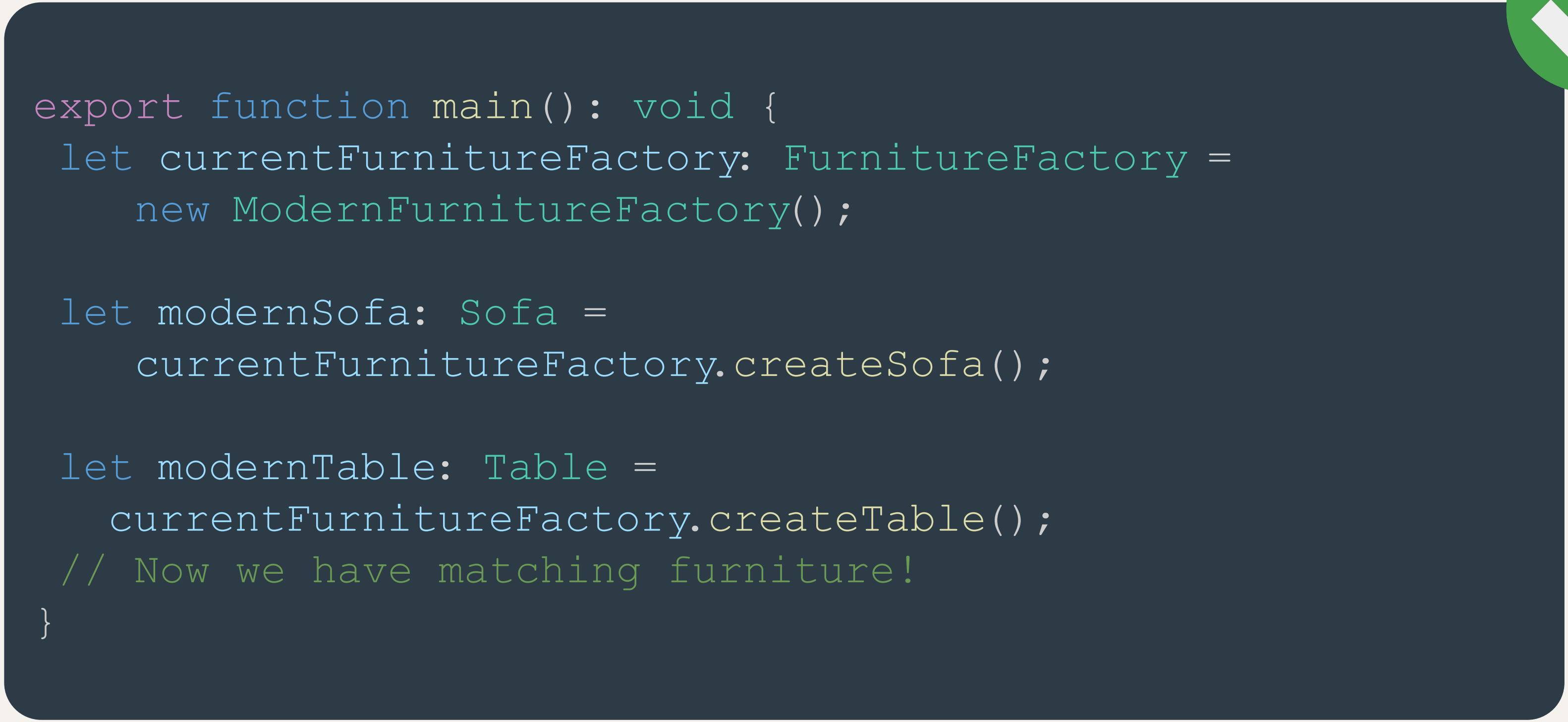
```
class ModernFurnitureFactory implements FurnitureFactory {
 public createSofa(): Sofa {
   console.log('Creating a modern sofa');
   return new ModernSofa();
 }

 public createTable(): Table {
   console.log('Creating a modern table');
   return new ModernTable();
 }
}
```

✓



```
class VictorianFurnitureFactory implements FurnitureFactory {
 public createSofa(): Sofa {
   console.log('Creating a victorian sofa');
   return new VictorianSofa();
 }

 public createTable(): Table {
   console.log('Creating a victorian table');
   return new VictorianTable();
 }
}
```

✓

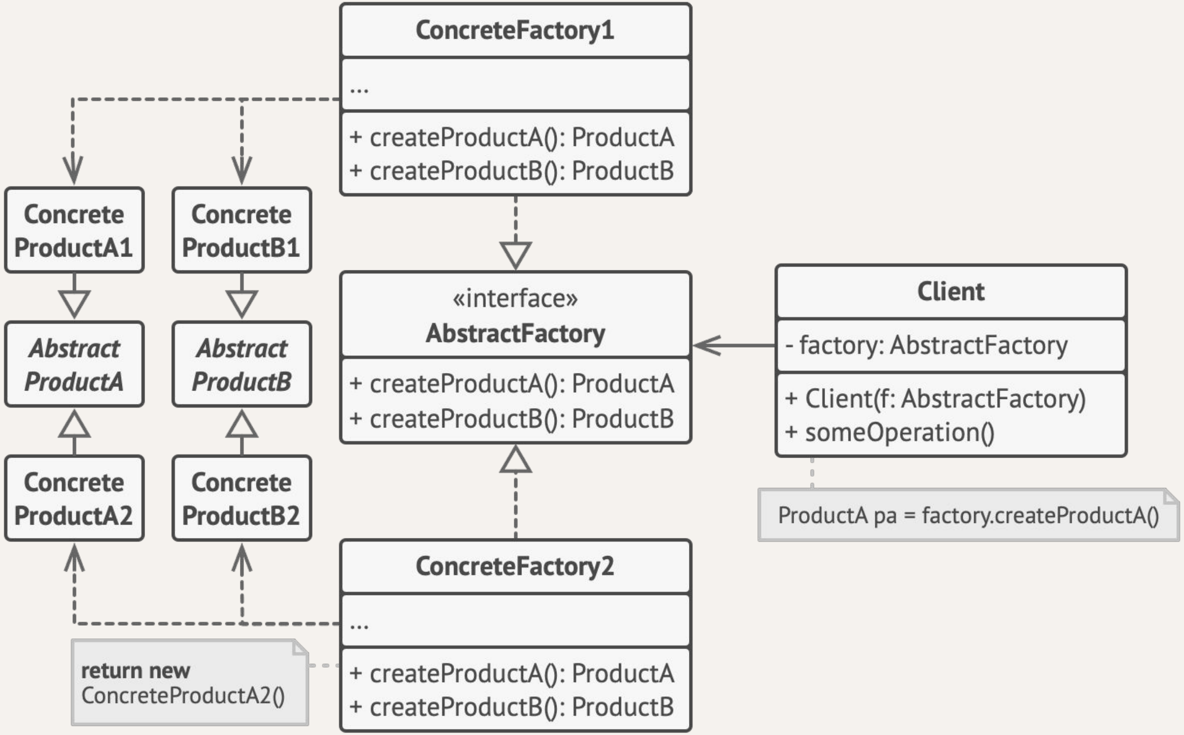


```
export function main(): void {
 let currentFurnitureFactory: FurnitureFactory =
     new ModernFurnitureFactory();

 let modernSofa: Sofa =
     currentFurnitureFactory.createSofa();

 let modernTable: Table =
    currentFurnitureFactory.createTable();
 // Now we have matching furniture!
}
```

Abstract Factory - Benefits

- Change objects type dynamically
- Ensure object compatibility

Abstract Factory - Diagram



ConcreteFactory1

...

+ createProductA(): ProductA
+ createProductB(): ProductB

Concrete ProductA1

Concrete ProductB1

Abstract ProductA

Abstract ProductB

Concrete ProductA2

Concrete ProductB2

«interface»
AbstractFactory

+ createProductA(): ProductA
+ createProductB(): ProductB

Client

- factory: AbstractFactory

+ Client(f: AbstractFactory)
+ someOperation()

ProductA pa = factory.createProductA()

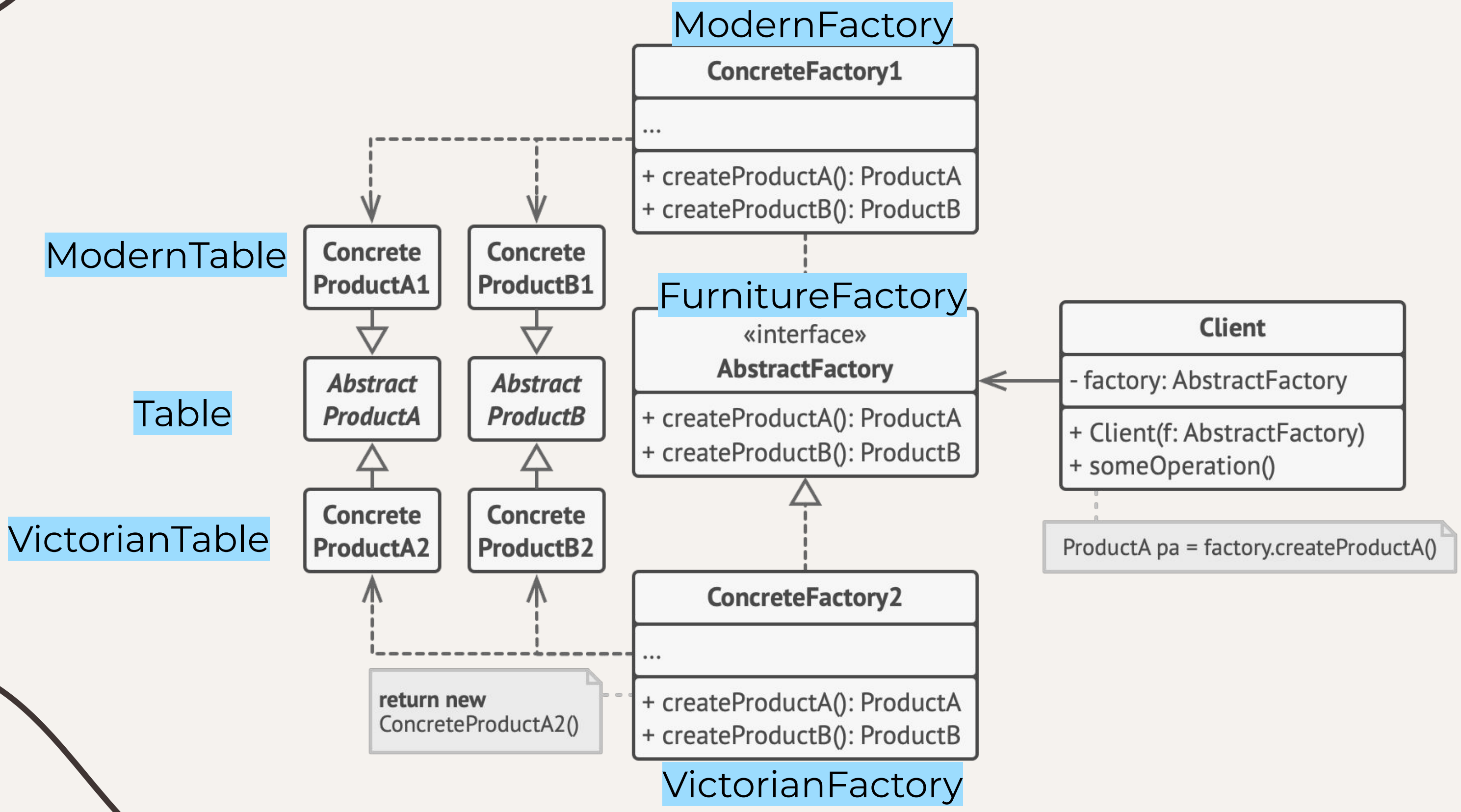
ConcreteFactory2

...

+ createProductA(): ProductA
+ createProductB(): ProductB

return new
ConcreteProductA2()

Abstract Factory - Diagram


ModernFactory
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
ModernTable
Concrete ProductA1
Concrete ProductB1
FurnitureFactory
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
Table
Abstract ProductA
Abstract ProductB
VictorianTable
Concrete ProductA2
Concrete ProductB2
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()
VictorianFactory

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”

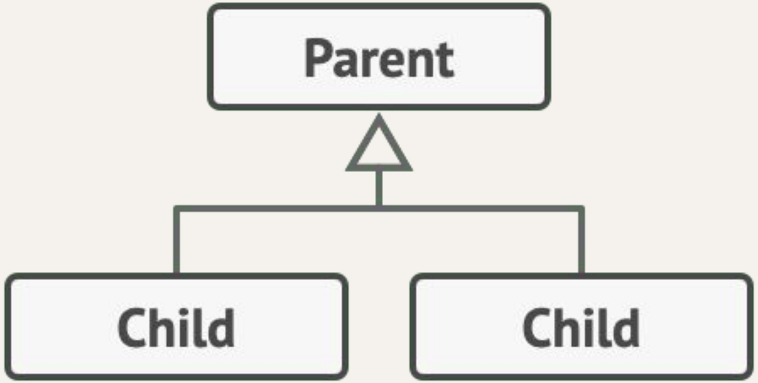
Decorator

★★☆

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

Composition vs Inheritance

Inheritance

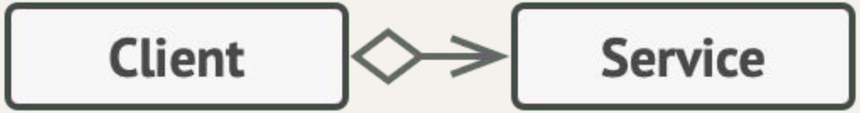


Parent

Child

Child

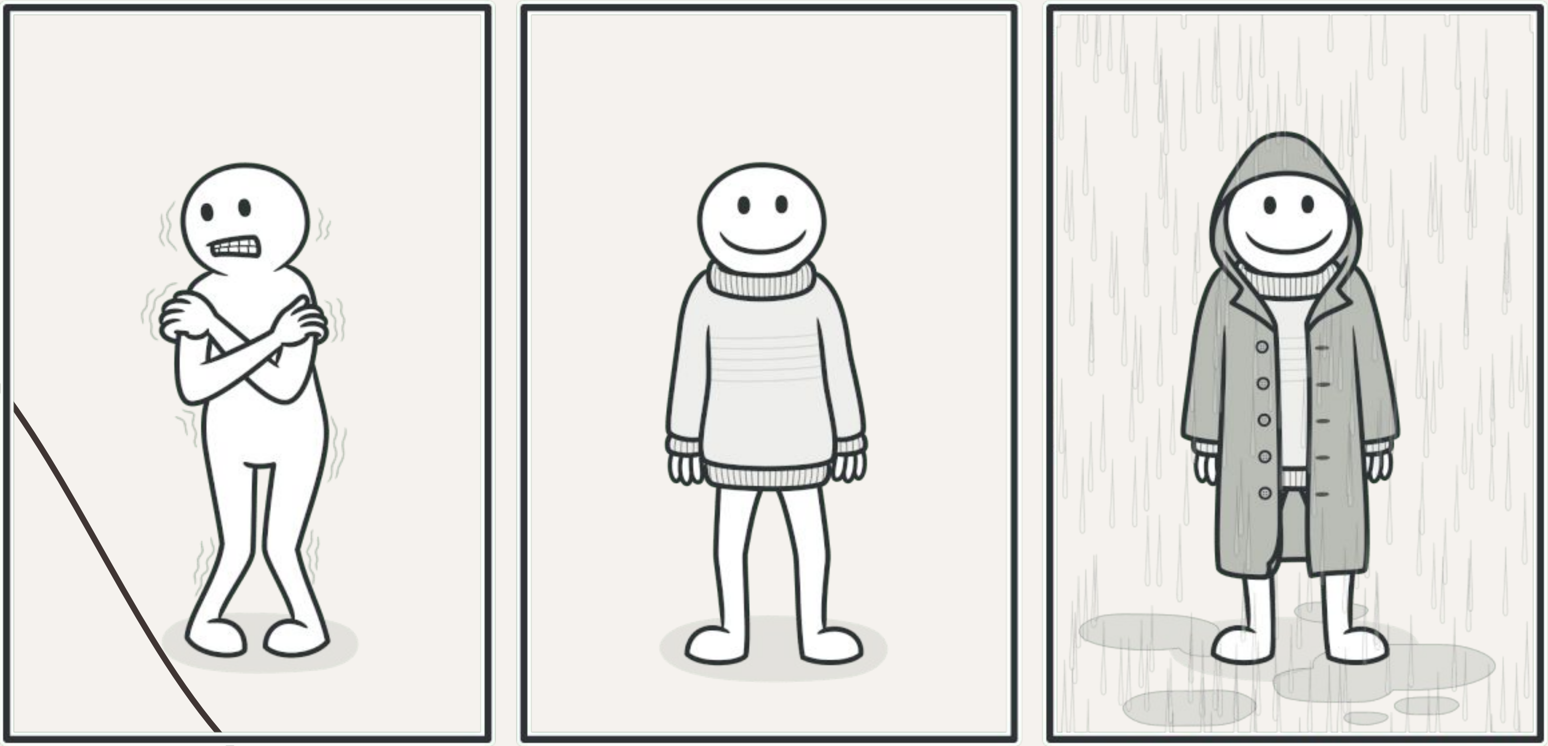
Aggregation



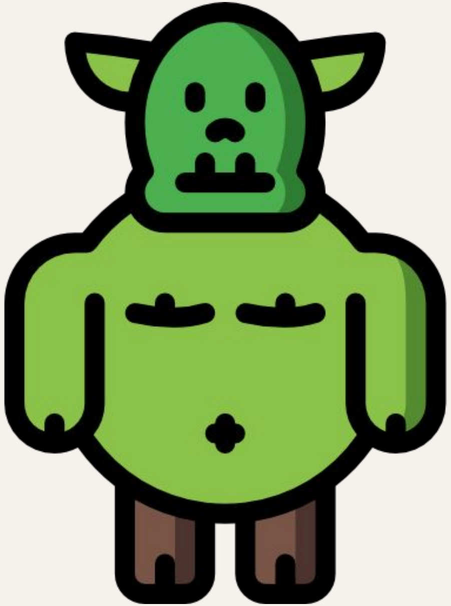
Client

Service

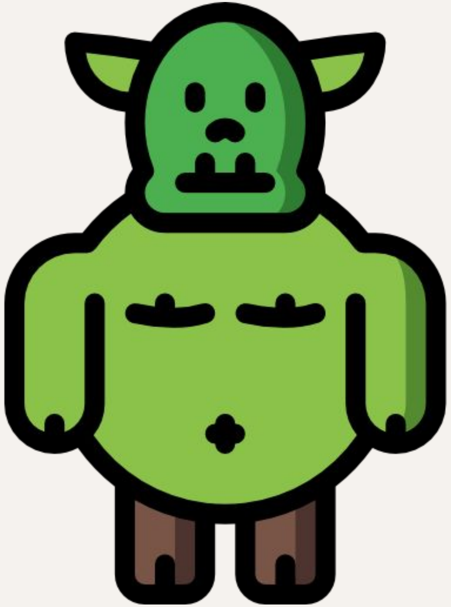
Decorator - Analogy



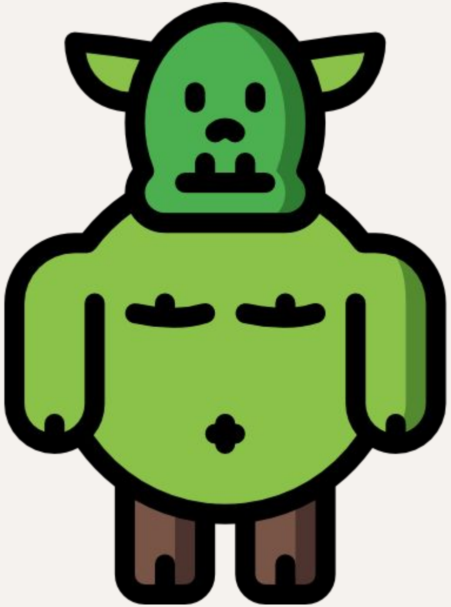
Decorator - Example



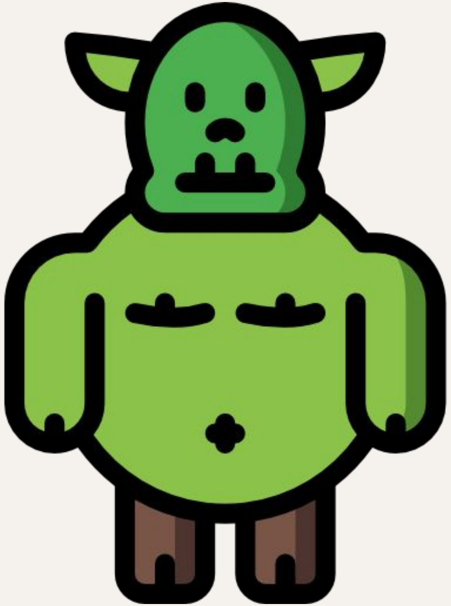
Decorator - Example

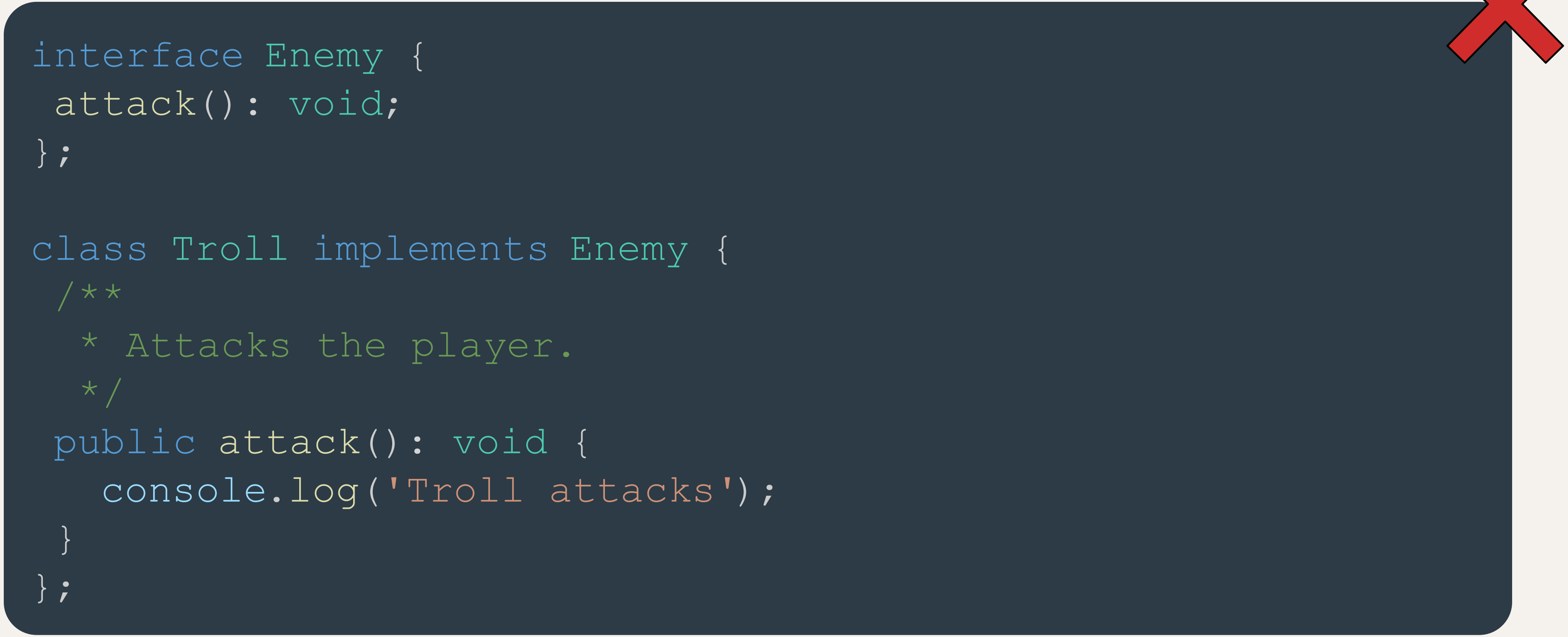


Decorator - Example

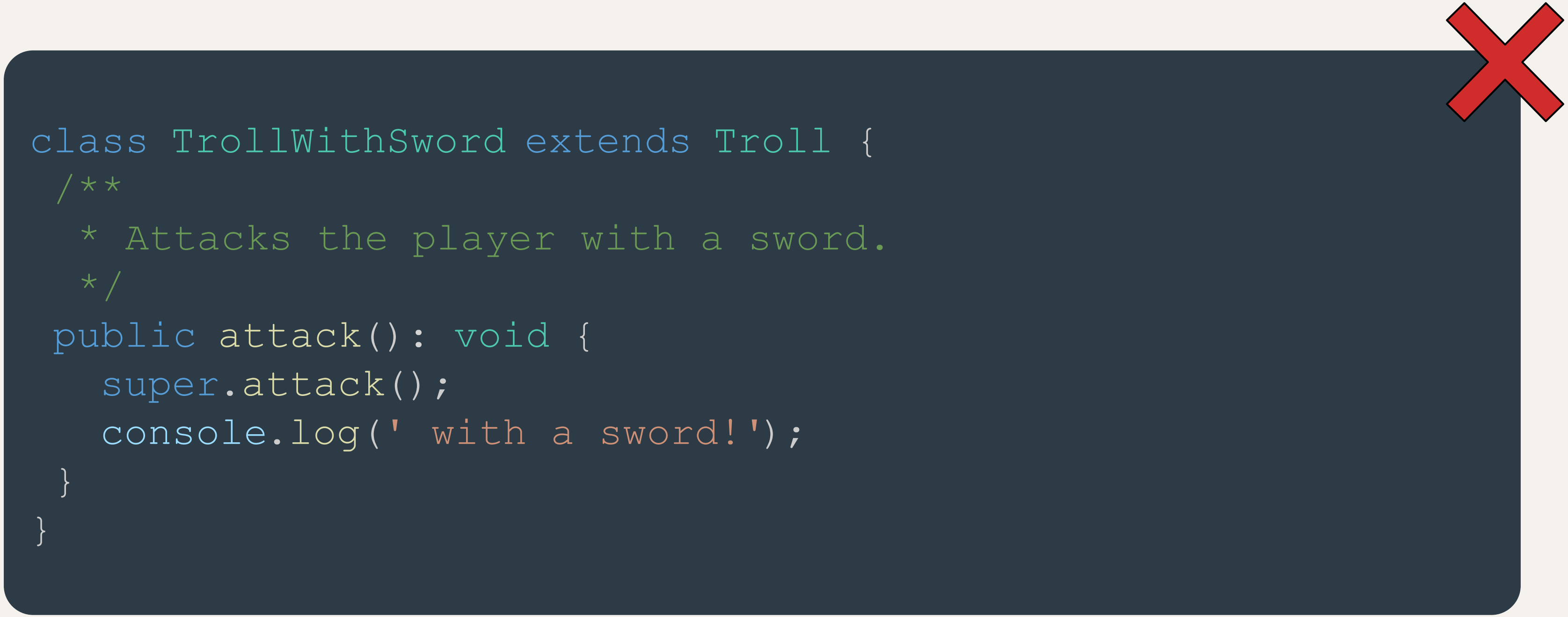


Decorator - Example

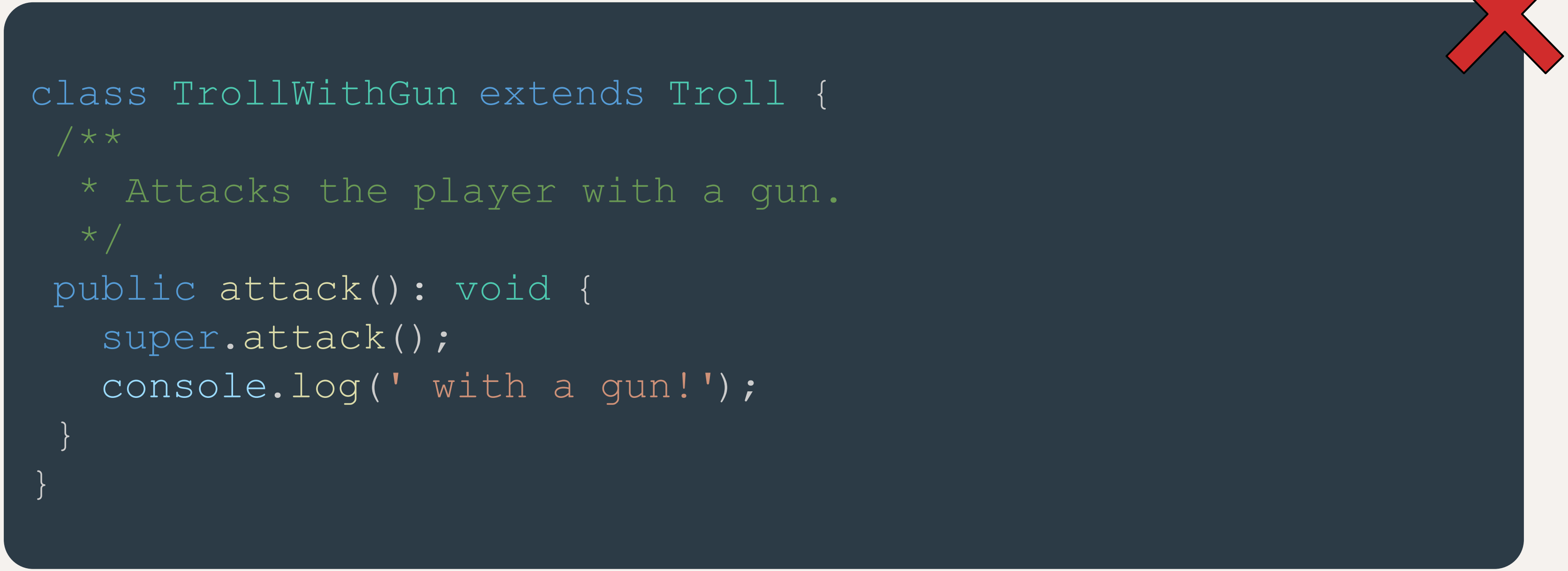




```
interface Enemy {
 attack(): void;
};

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!');
 }
}
```

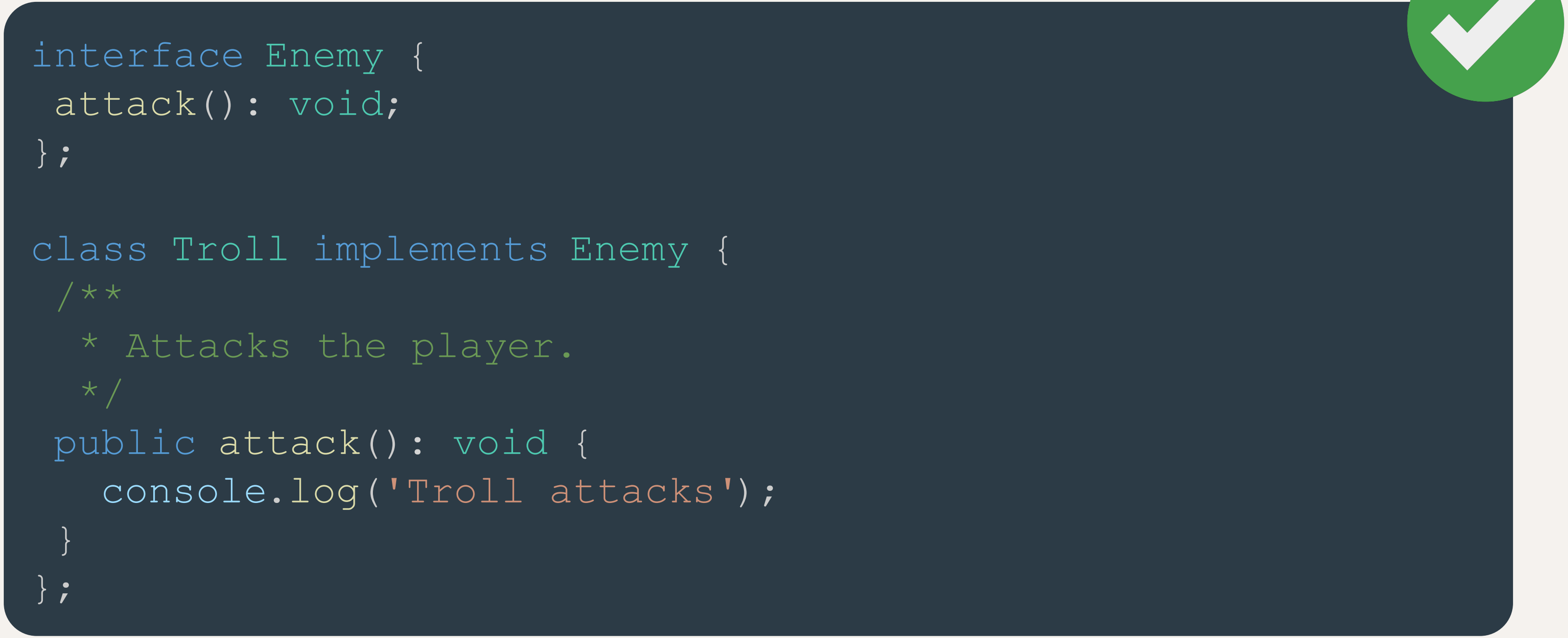
Decorator- Problems

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

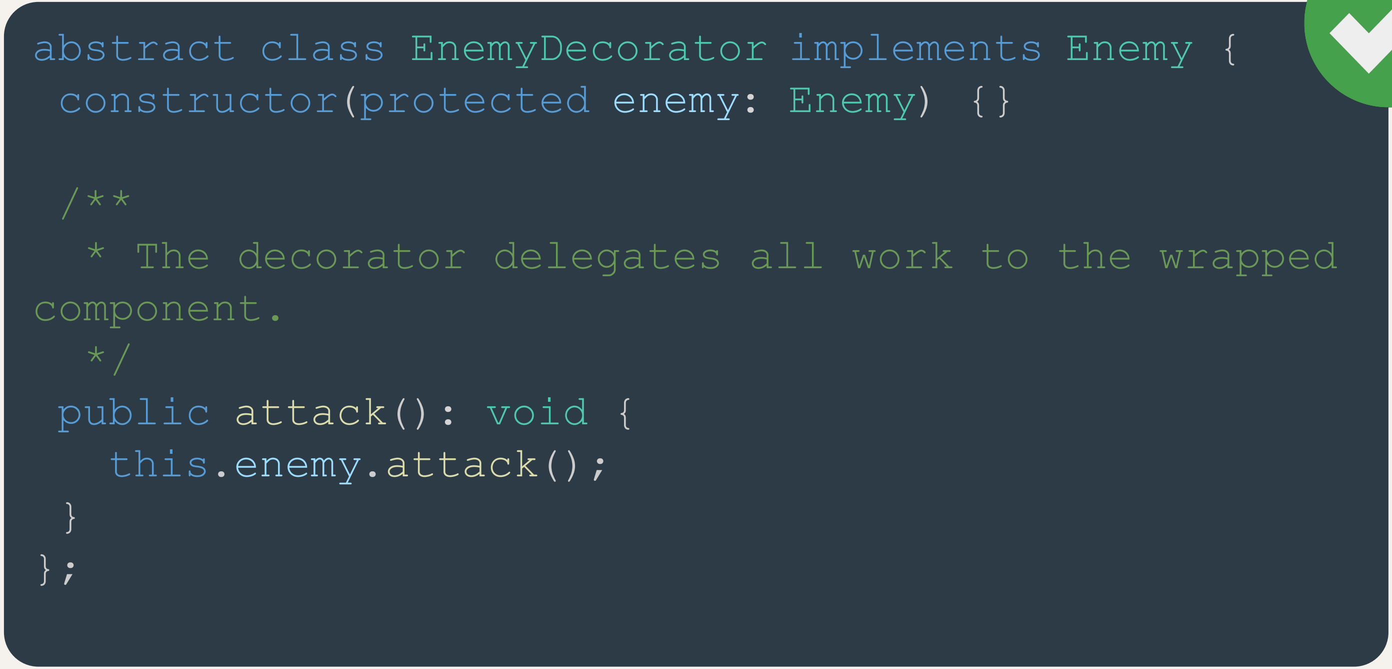
Decorator - Solution

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

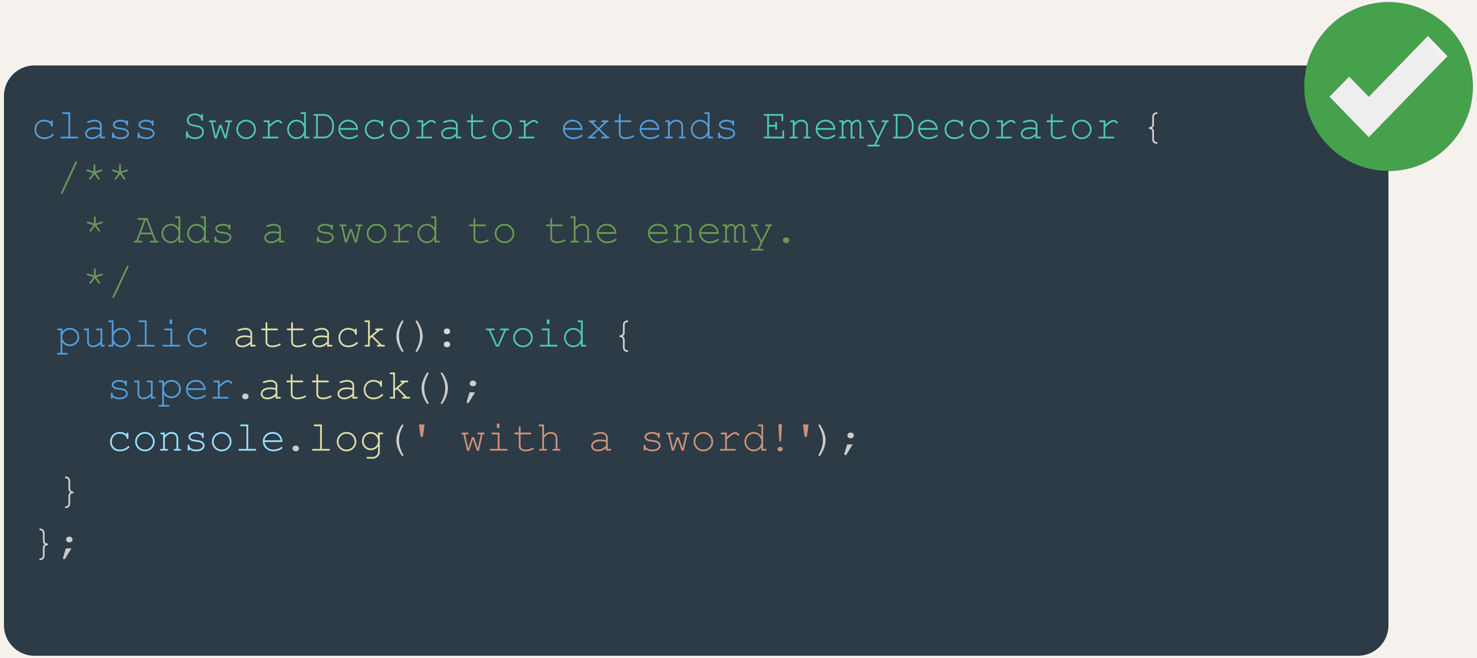
✓



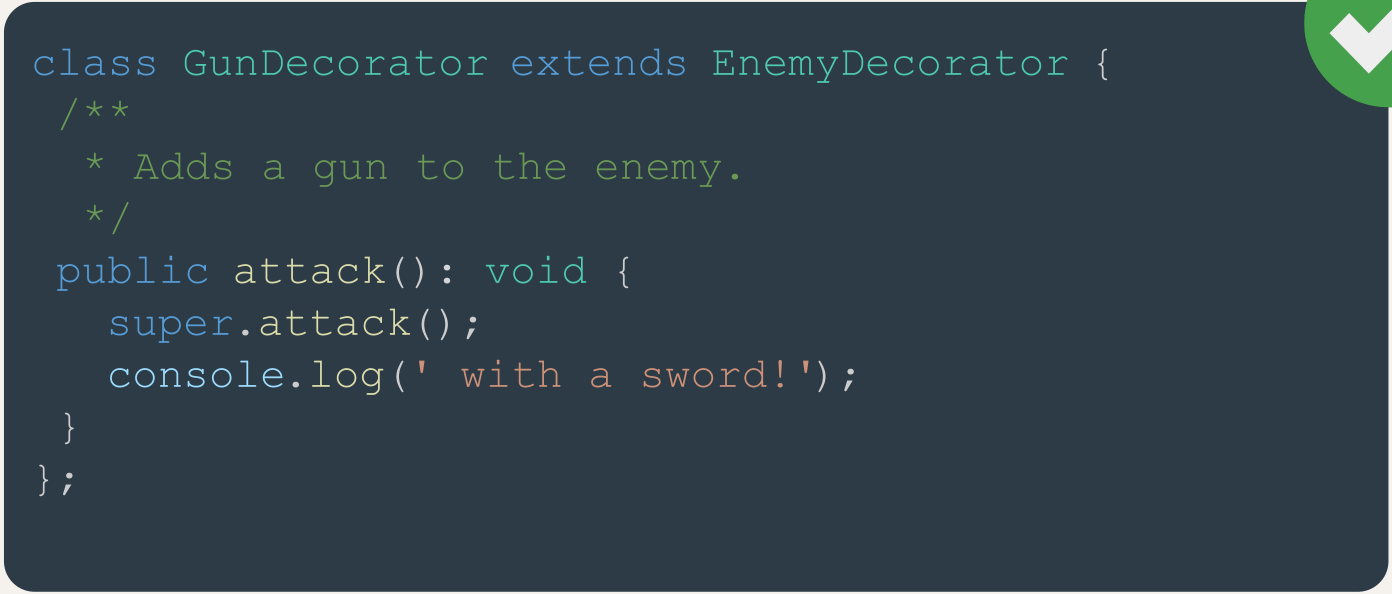
```
interface Enemy {
 attack(): void;
};

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!');
 }
};
```



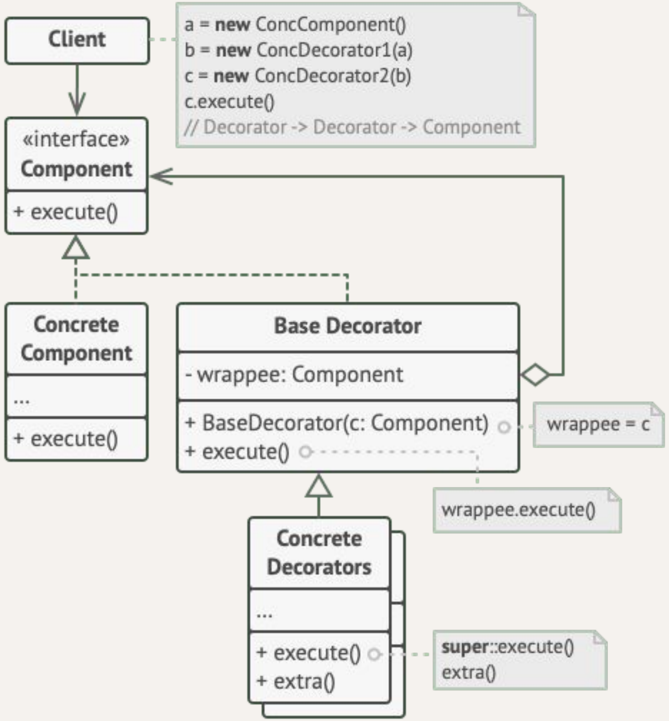
```
class GunDecorator extends EnemyDecorator {
 /**
  * Adds a gun to the enemy.
  */
 public attack(): void {
   super.attack();
   console.log(' with a sword!');
 }
};
```

✓

Decorator - Benefits

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

Decorator - Diagram



Client

```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

«interface»
Component

+ execute()

Concrete Component

...

+ execute()

Base Decorator

- wrappee: Component

+ BaseDecorator(c: Component)
+ execute()

```
wrappee = c
```

```
wrappee.execute()
```

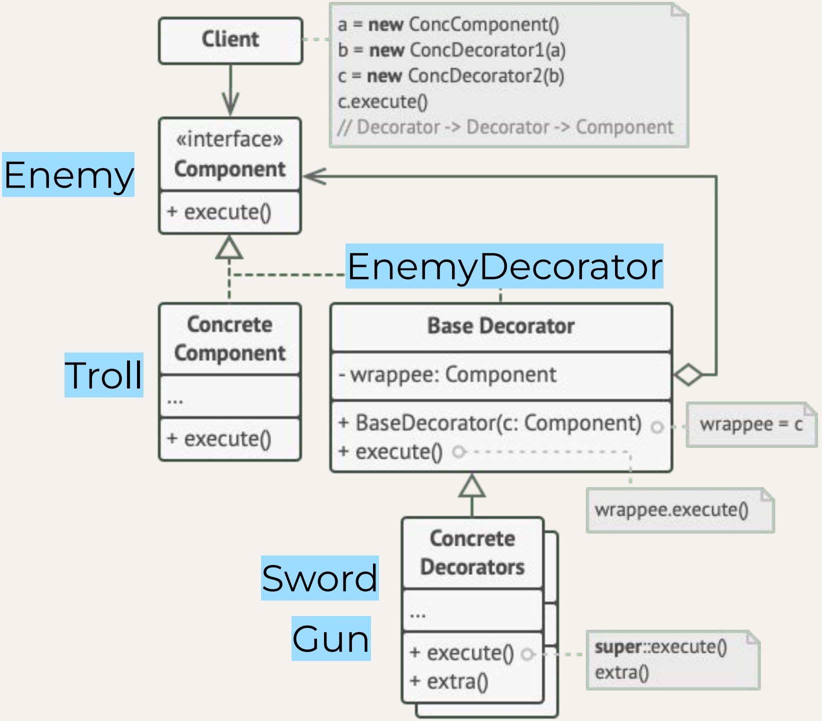
Concrete Decorators

...

+ execute()
+ extra()

```
super::execute()
extra()
```

Decorator - Diagram

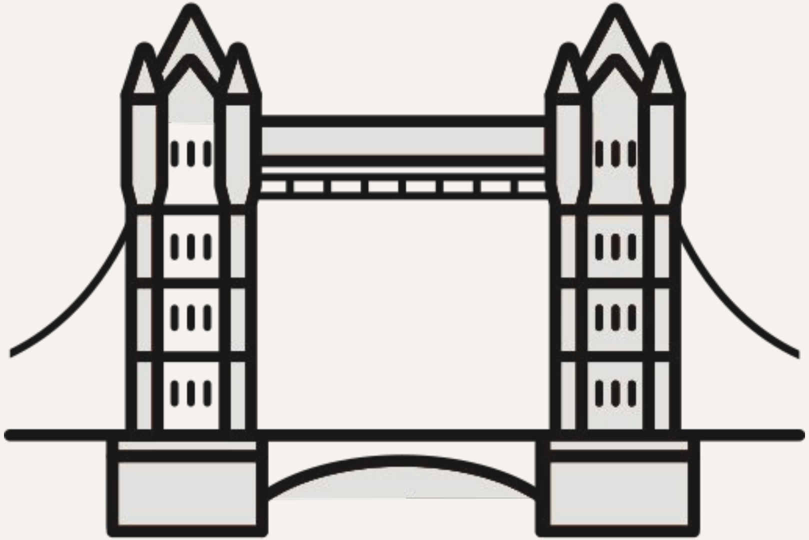


```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

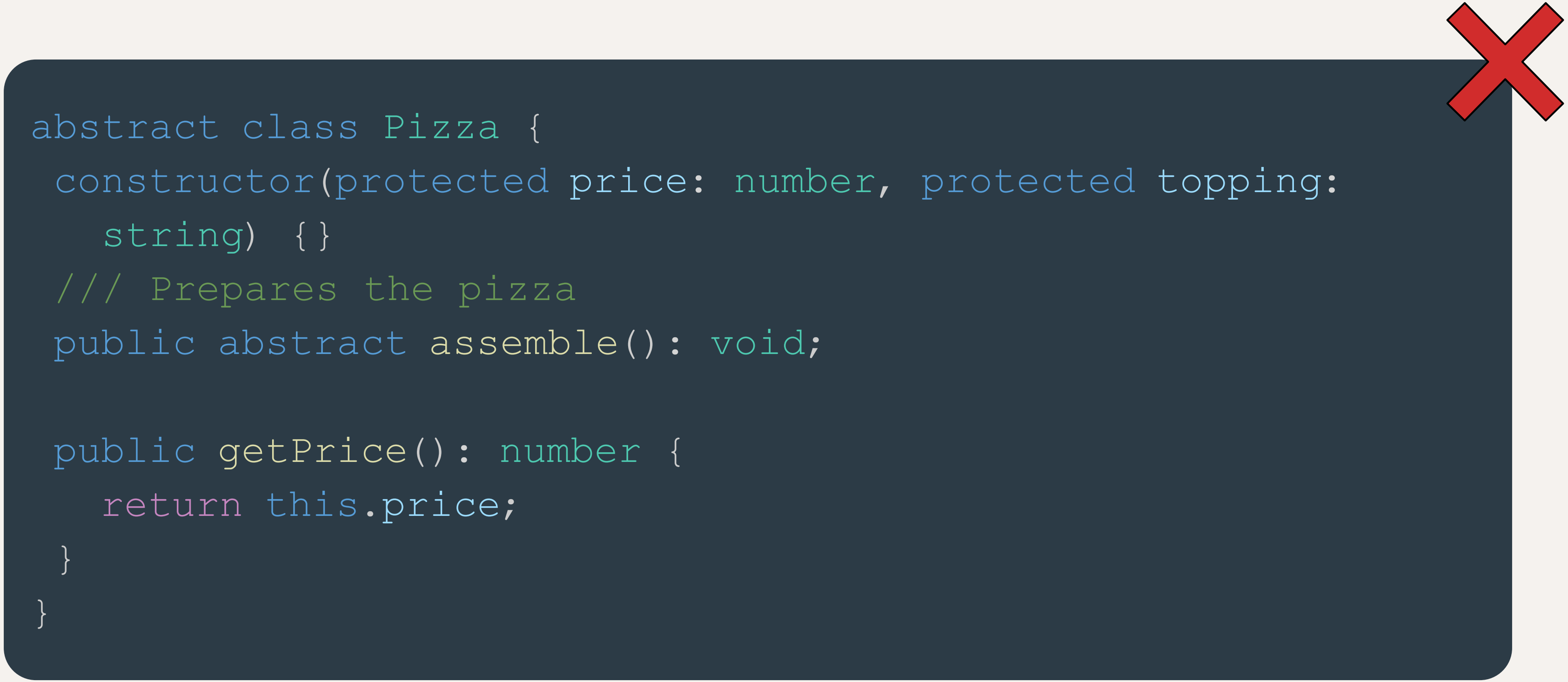
Bridge

★☆☆

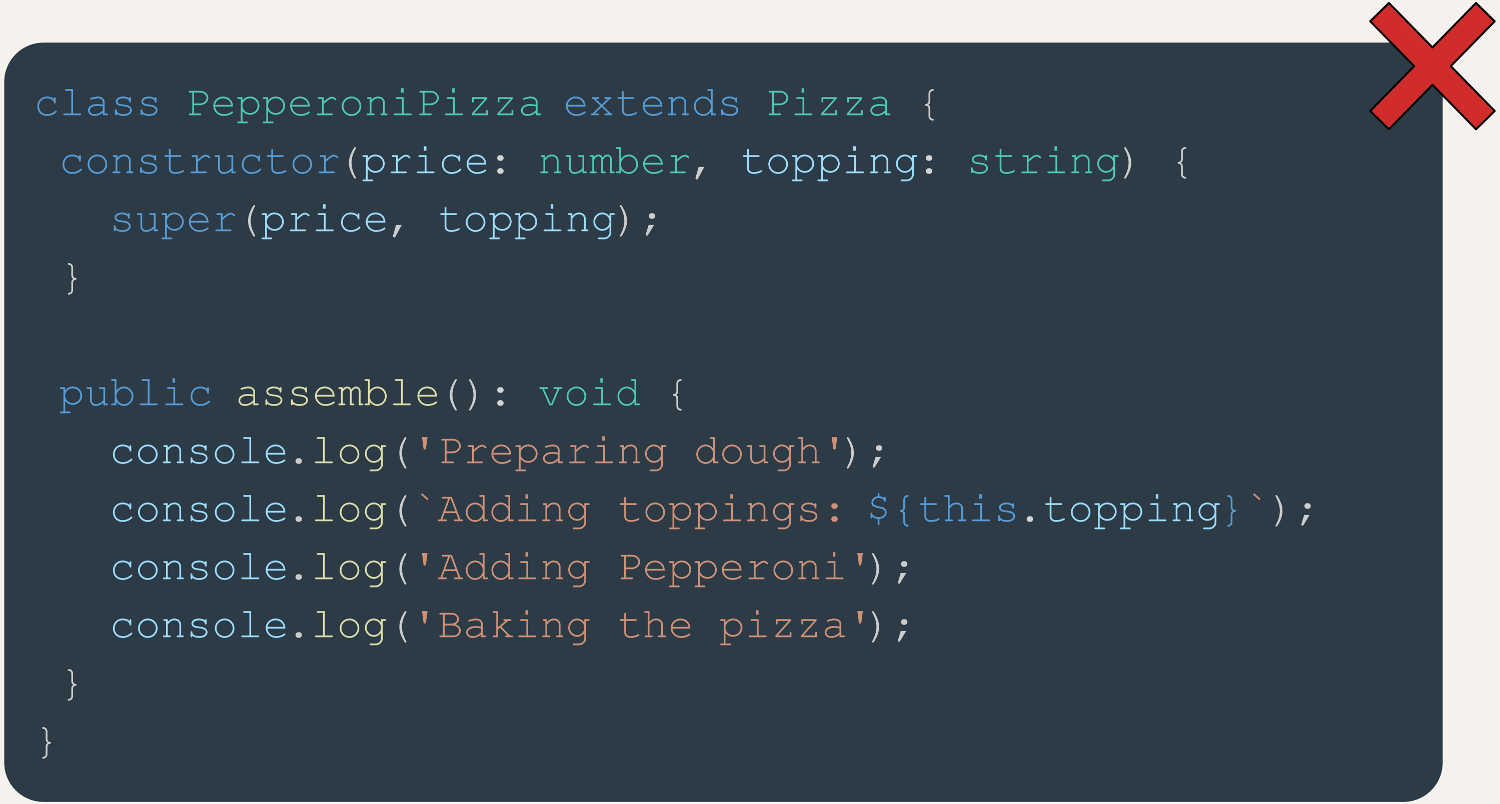
- Splits a large class into two separate hierarchies



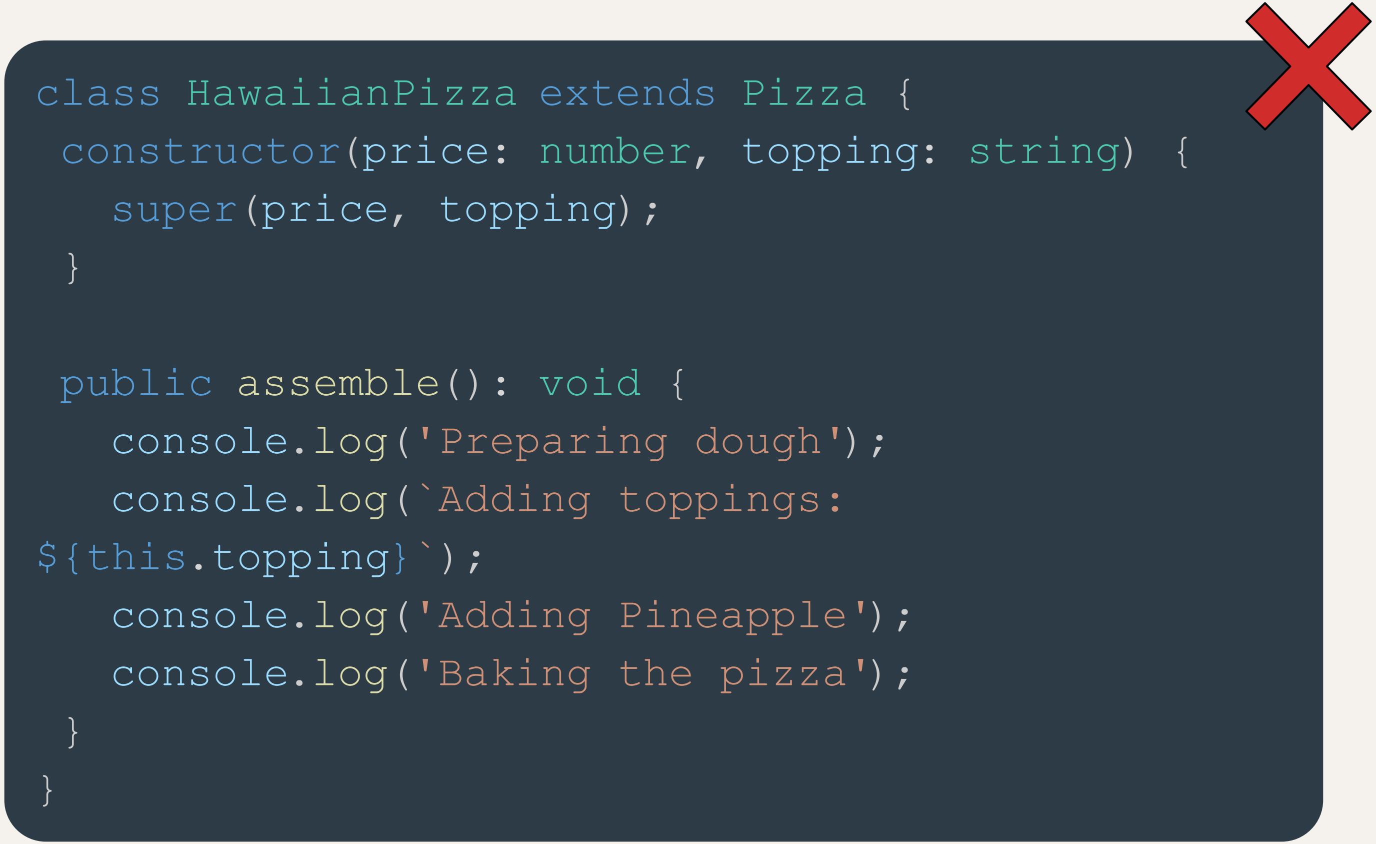
Bridge - Example



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

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

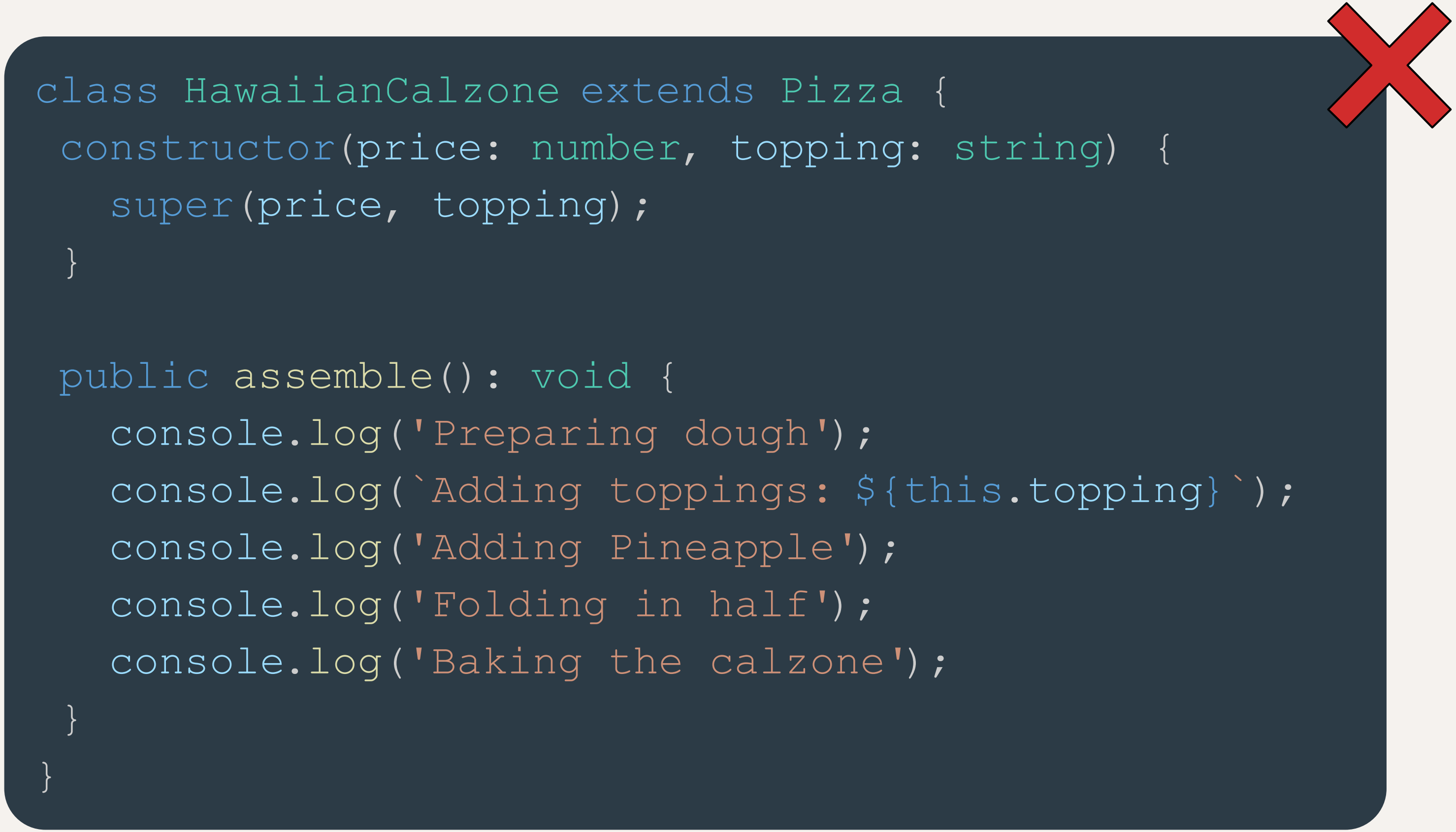


```
class PepperoniPizza extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 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');
 }
}
```



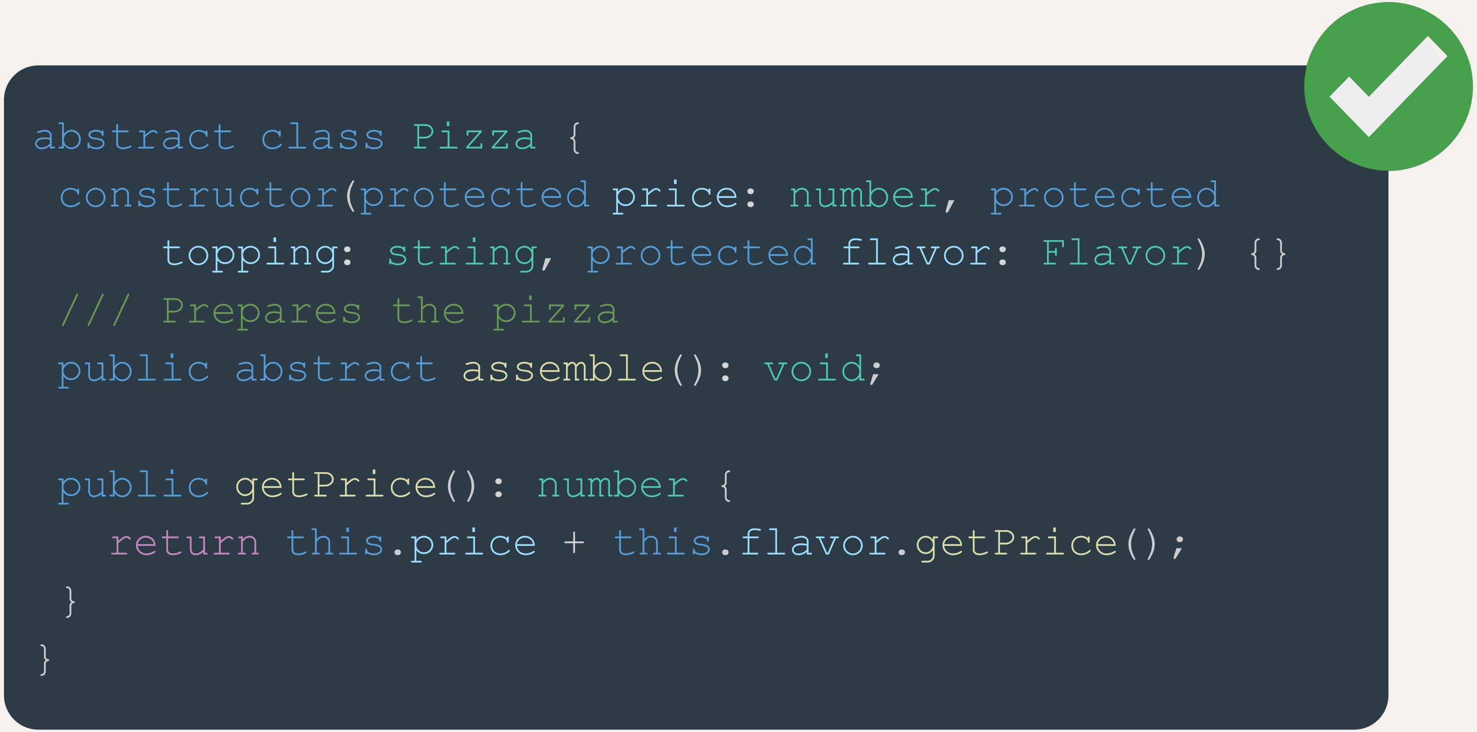
```typescript
class HawaiianCalzone extends Pizza {
 constructor(price: number, topping: string) {
   super(price, topping);
 }

 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

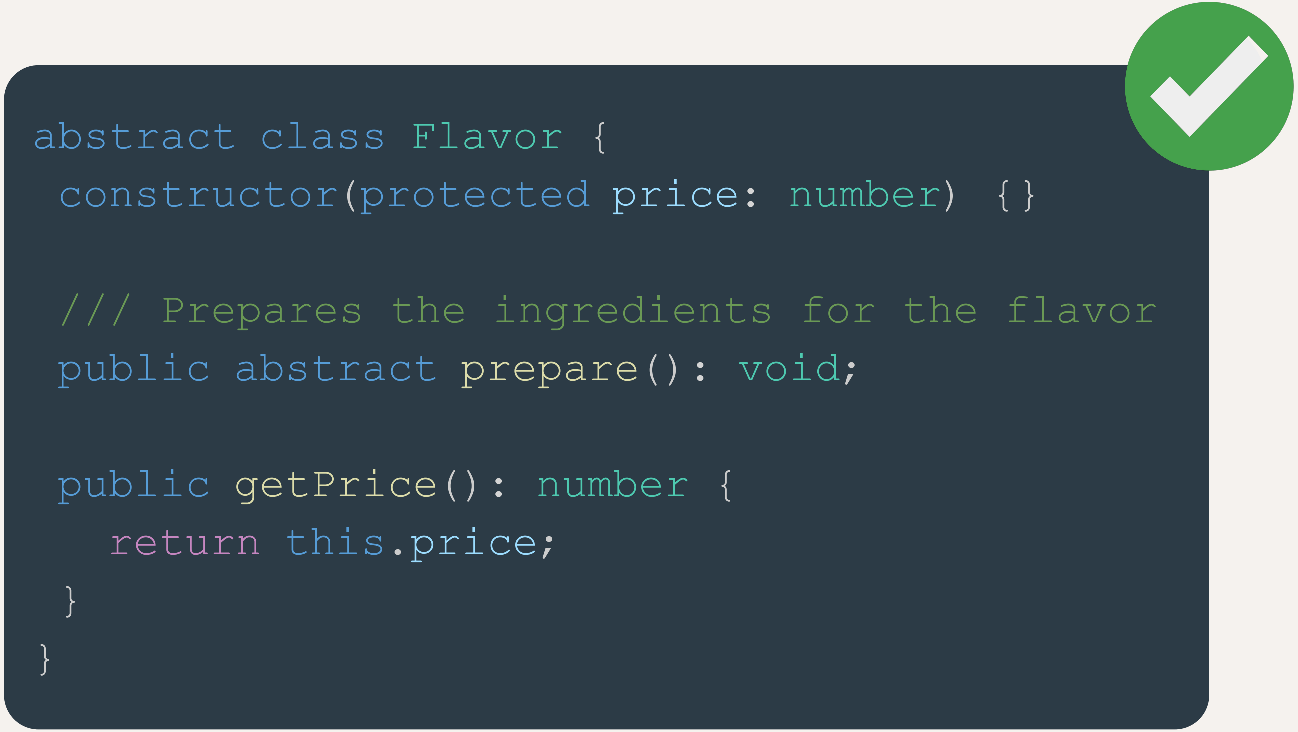
Bridge - Solution

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

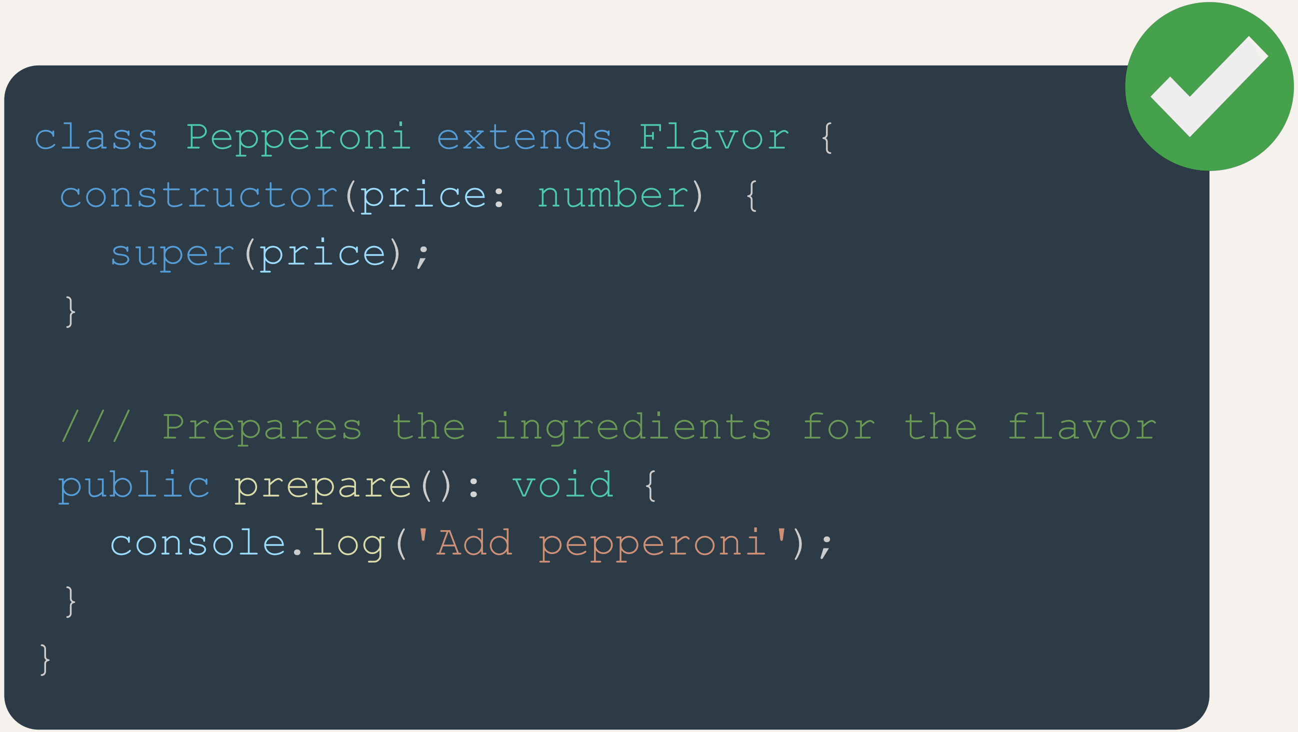


✓

```
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');
 }
}
```

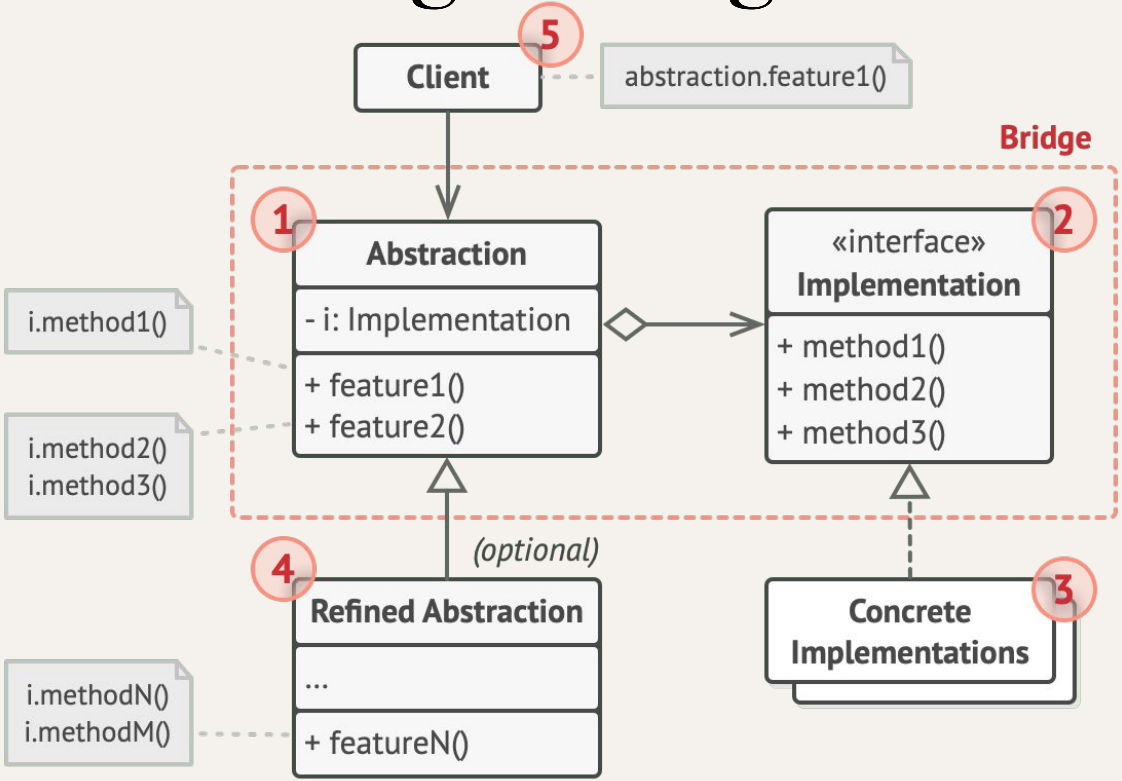
✓

```
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');
 }
}
```

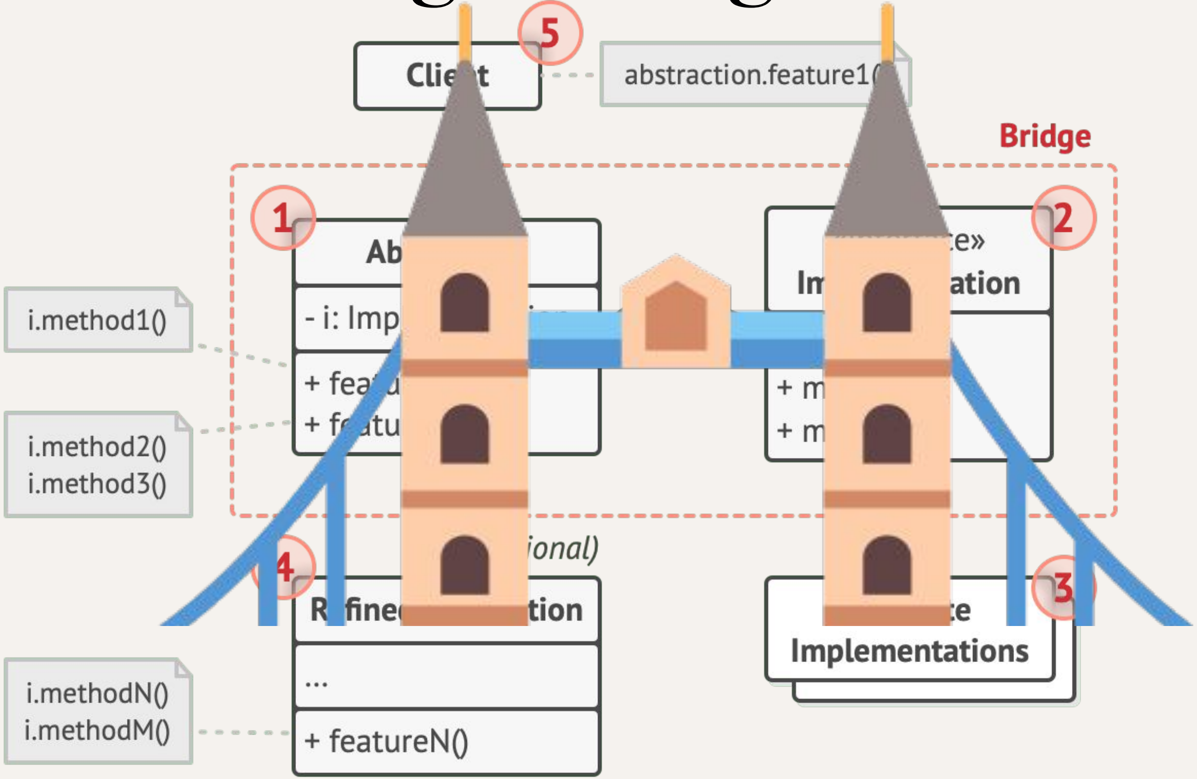
Bridge - Benefits

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

Bridge - Diagram



Bridge - Diagram



04

Behavioral Patterns

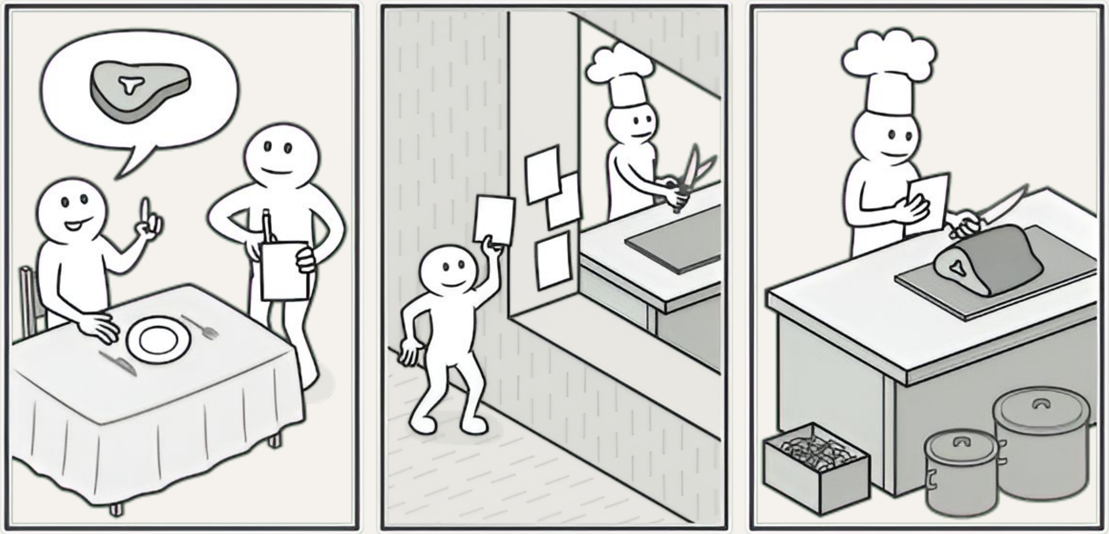
Behavioral Patterns

- Handle communication between objects
  - Distribute responsibilities
  - Improve encapsulation

Command

★★★

- Encapsulates a request as an object

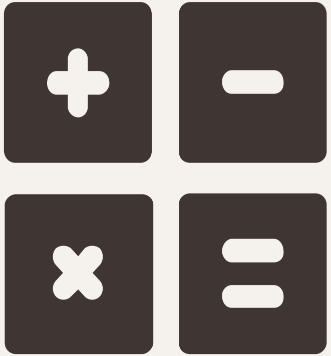


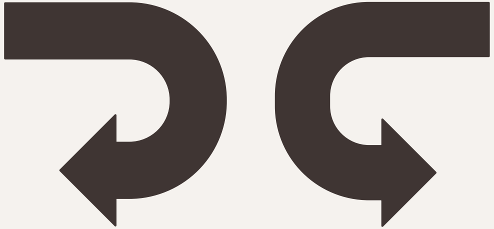
Command - Benefits

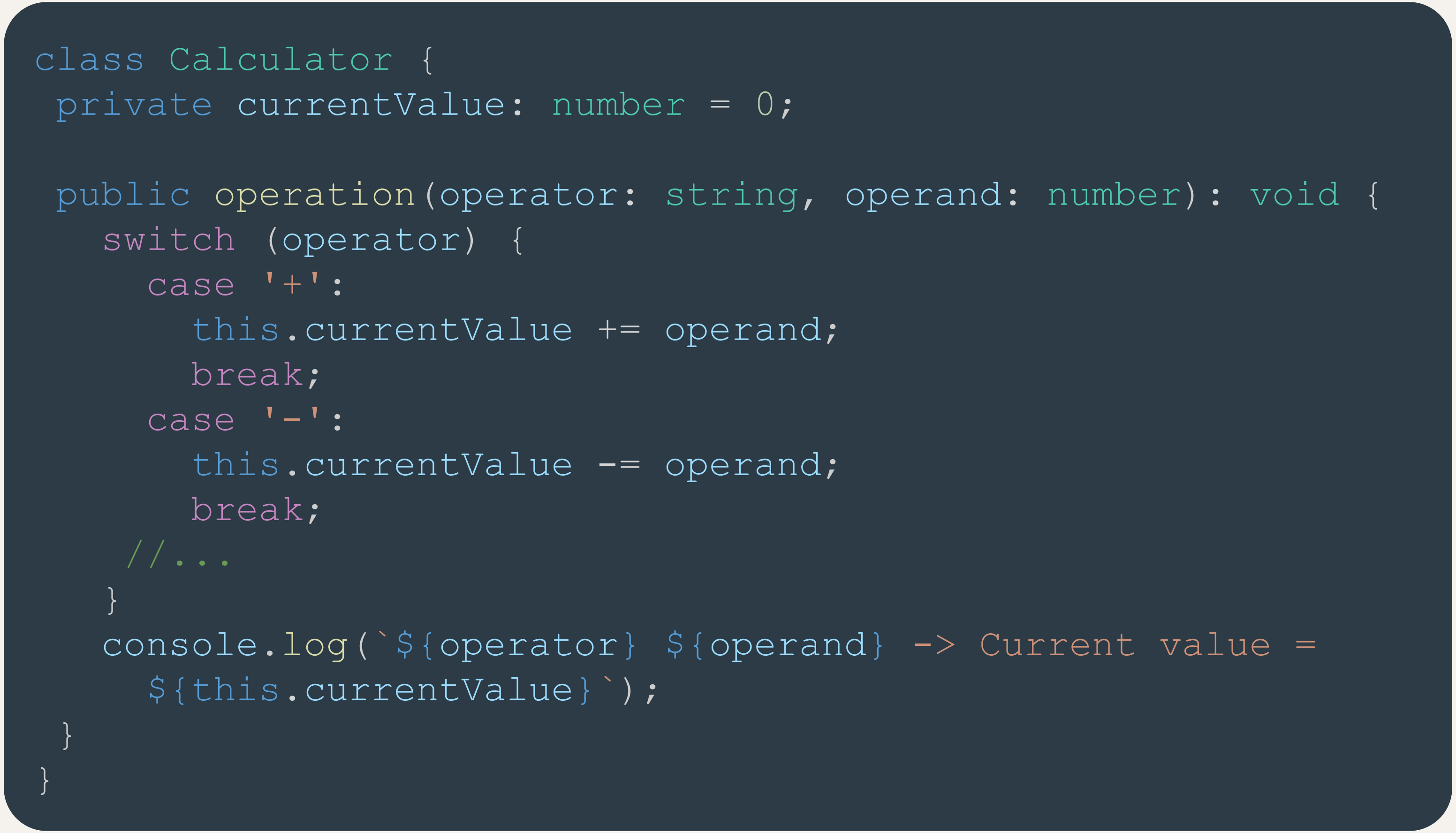
- Store Commands
  - Queue or schedule operations
  - Implement reversible operations
  - Access information about a request
  - Reuse petitions

- Decouple invocation from implementation

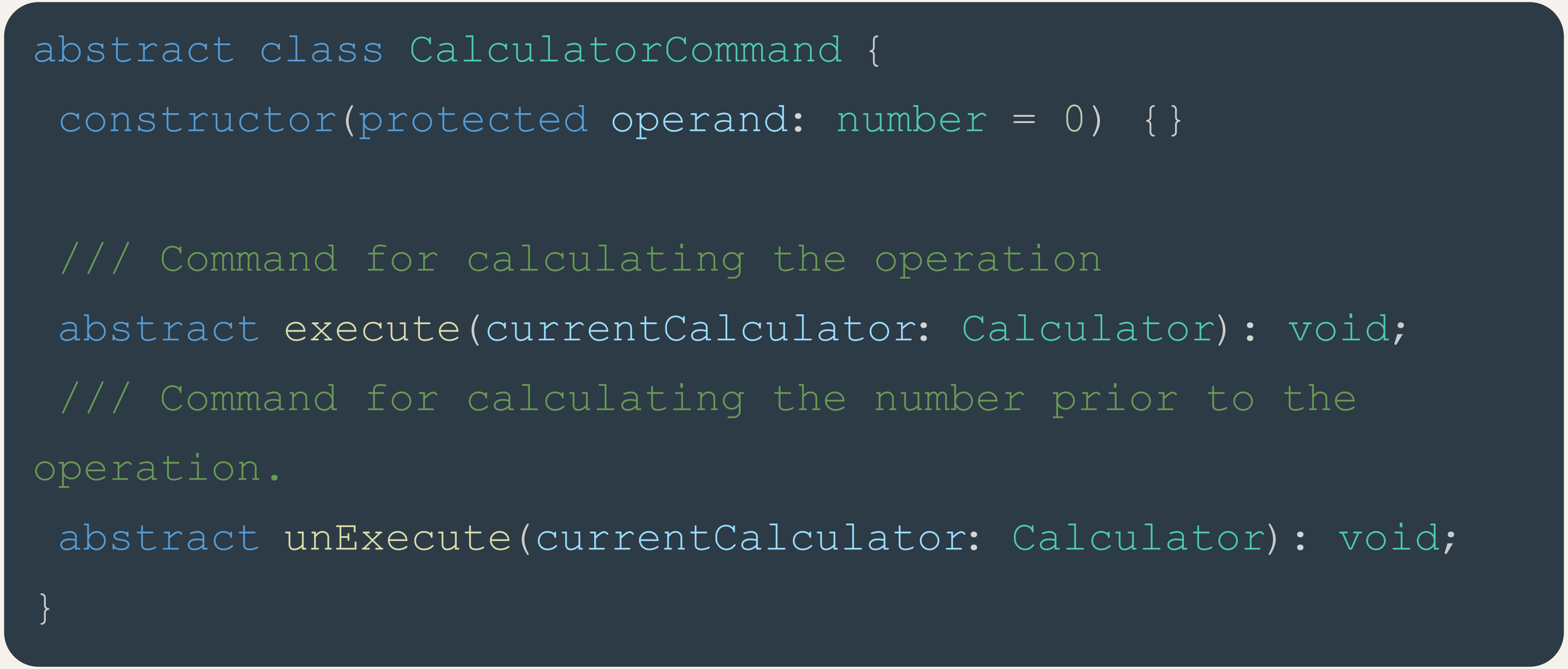
Command - Example



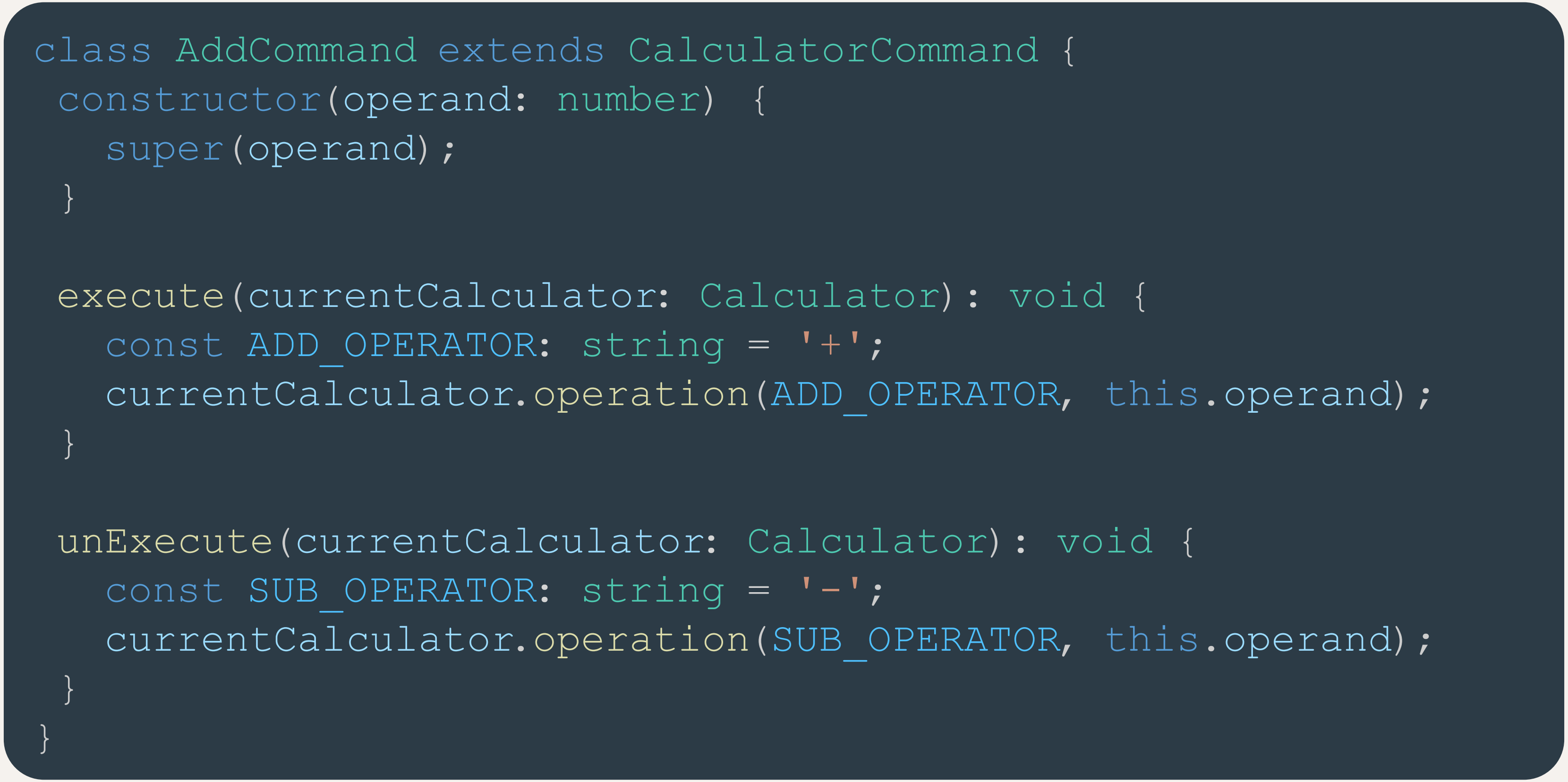




```
class Calculator {
 private currentValue: number = 0;

 public operation(operator: string, operand: number): void {
   switch (operator) {
     case '+':
       this.currentValue += operand;
       break;
     case '-':
       this.currentValue -= operand;
       break;
    //...
   }
   console.log(`${operator} ${operand} -> Current value =
     ${this.currentValue}`);
 }
}
```

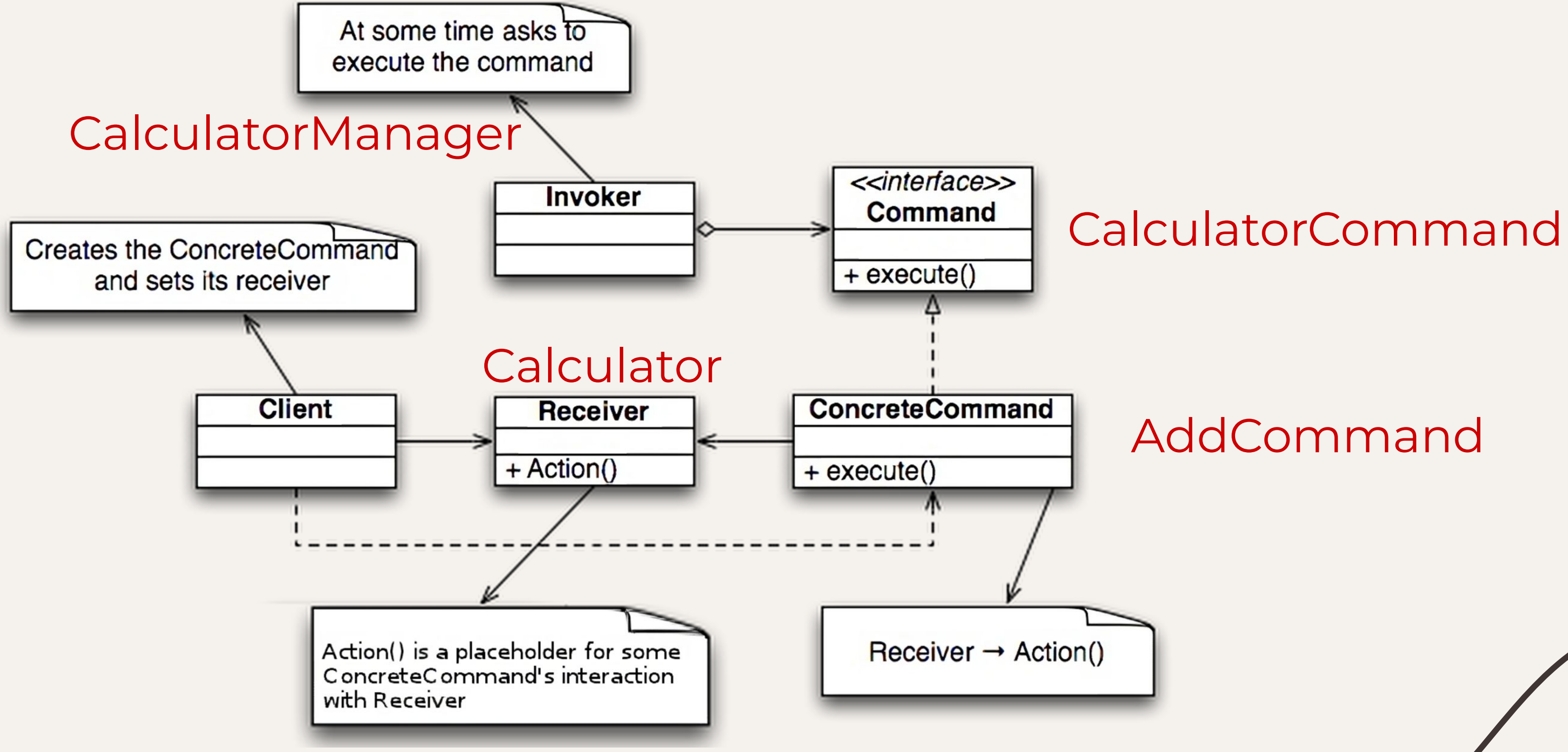


```
abstract class CalculatorCommand {
 constructor(protected operand: number = 0) {}

 /// Command for calculating the operation
 abstract execute(currentCalculator: Calculator): void;
 /// Command for calculating the number prior to the
operation.
 abstract unExecute(currentCalculator: Calculator): void;
}
```



```
class AddCommand extends CalculatorCommand {
 constructor(operand: number) {
   super(operand);
 }

 execute(currentCalculator: Calculator): void {
   const ADD_OPERATOR: string = '+';
   currentCalculator.operation(ADD_OPERATOR, this.operand);
 }

 unExecute(currentCalculator: Calculator): void {
   const SUB_OPERATOR: string = '-';
   currentCalculator.operation(SUB_OPERATOR, this.operand);
 }
}
```

Command - Diagram

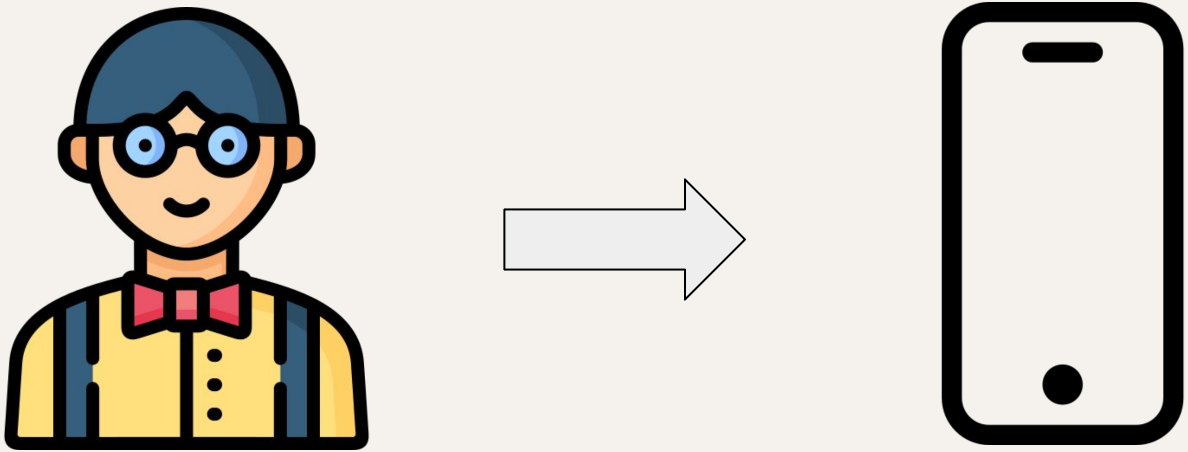


At some time asks to execute the command
CalculatorManager
Invoker
<<interface>>
Command
+ execute()
CalculatorCommand
Creates the ConcreteCommand and sets its receiver
Calculator
Client
Receiver
+ Action()
ConcreteCommand
+ execute()
AddCommand
Action() is a placeholder for some ConcreteCommand's interaction with Receiver
Receiver → Action()

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

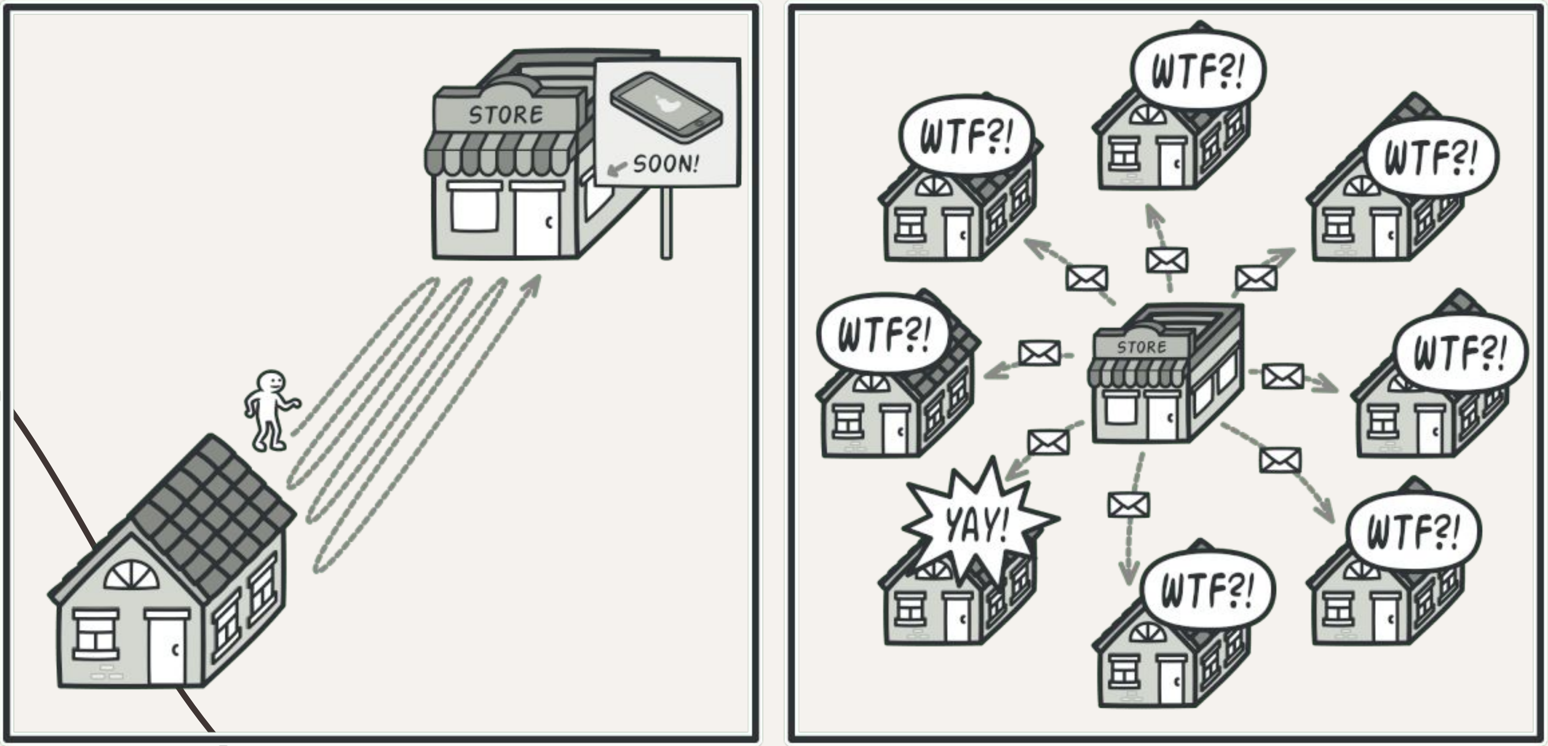
Observer - Example



Robert

iPhone 16

Observer - Example

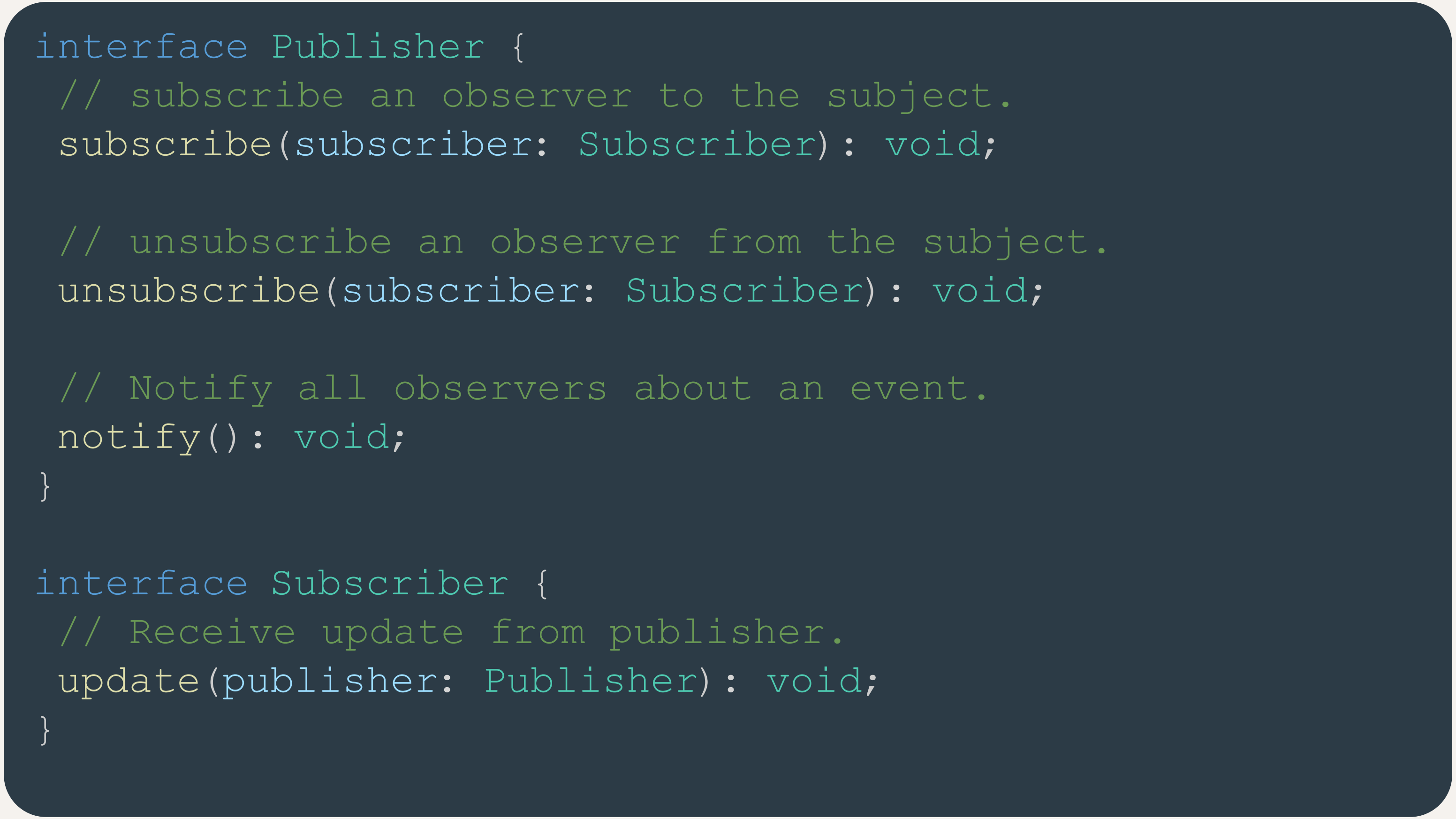


Observer - Problems

- The user has to check everyday if any event has happened
- If the store notifies everyone, lots of people would receive unwanted information

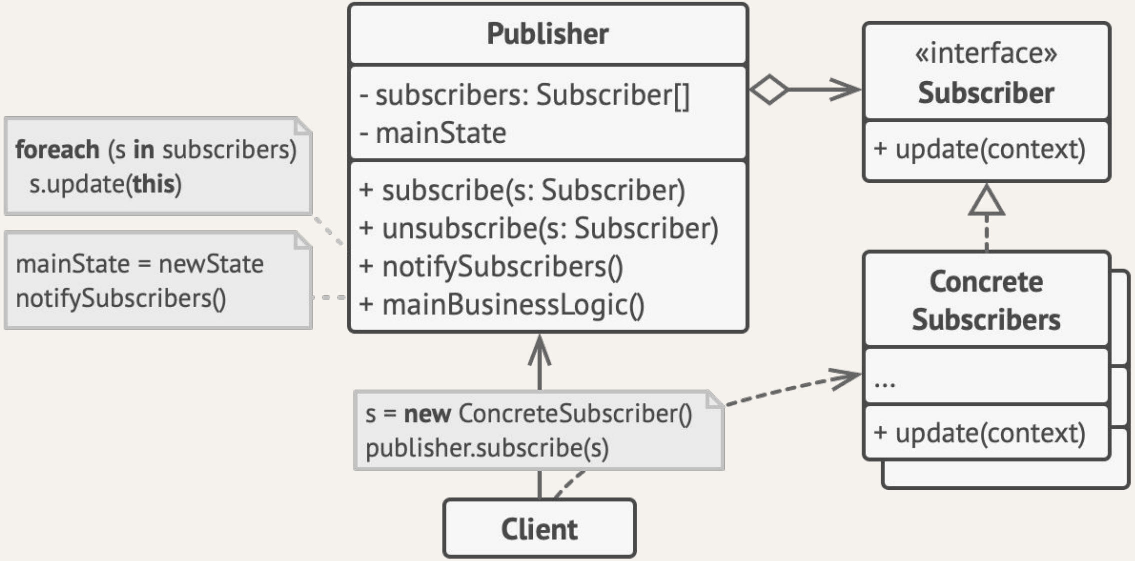
Observer - Solution

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



```
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



foreach (s in subscribers)
  s.update(this)

mainState = newState
notifySubscribers()

Publisher

- subscribers: Subscriber[]
- mainState

+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()

«interface»
Subscriber

+ update(context)

Concrete Subscribers

...

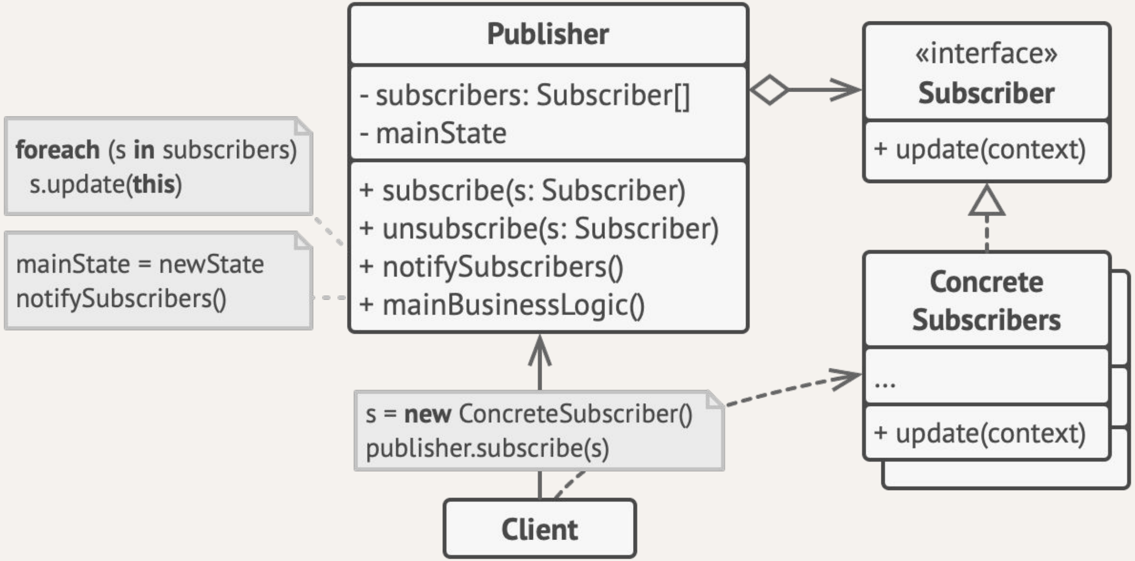
+ update(context)

s = new ConcreteSubscriber()
publisher.subscribe(s)

Client

Observer - Diagram

Apple Store



Publisher

- subscribers: Subscriber[]
- mainState

+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()

```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

«interface»
Subscriber

+ update(context)

Concrete Subscribers

...

+ update(context)

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

Client

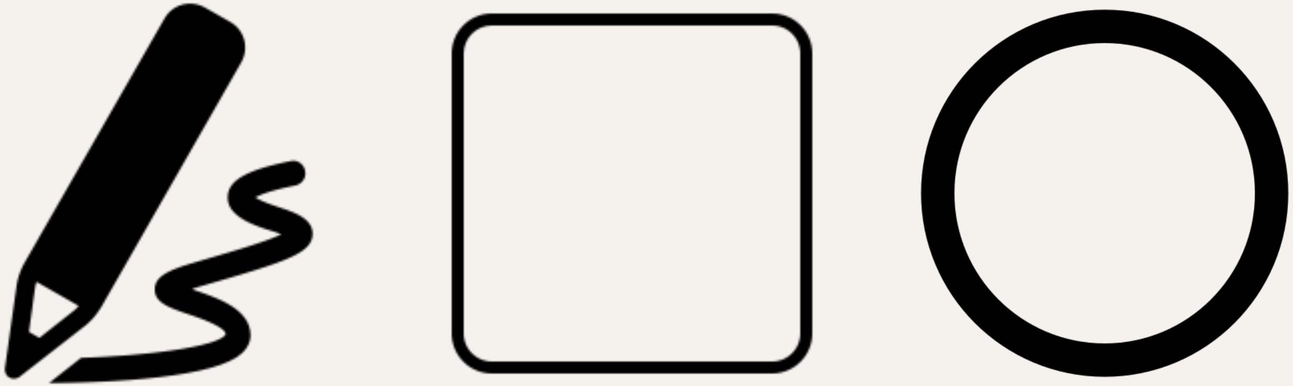
Robert

Strategy ★★★

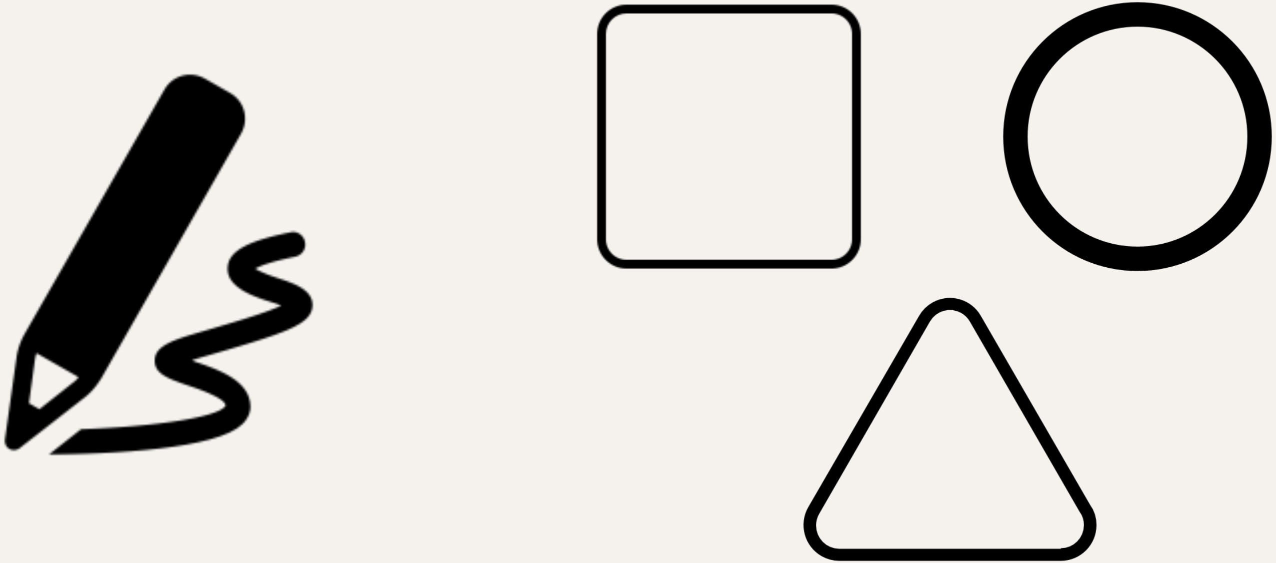
- Lets you define a family of algorithms
- We put each of them in a separate class and make their objects interchangeable
- We can switch between algorithms during runtime

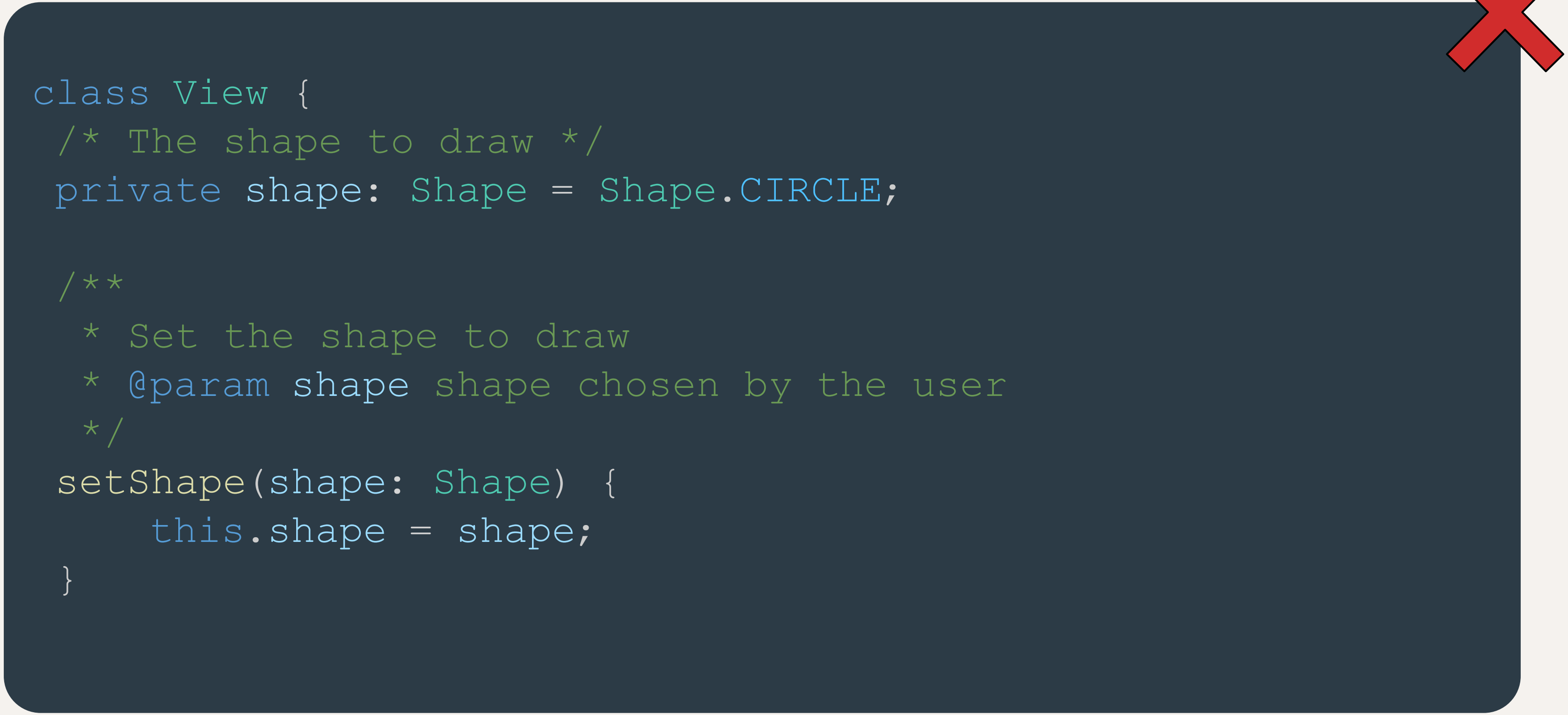
Strategy - Example

Strategy - Example

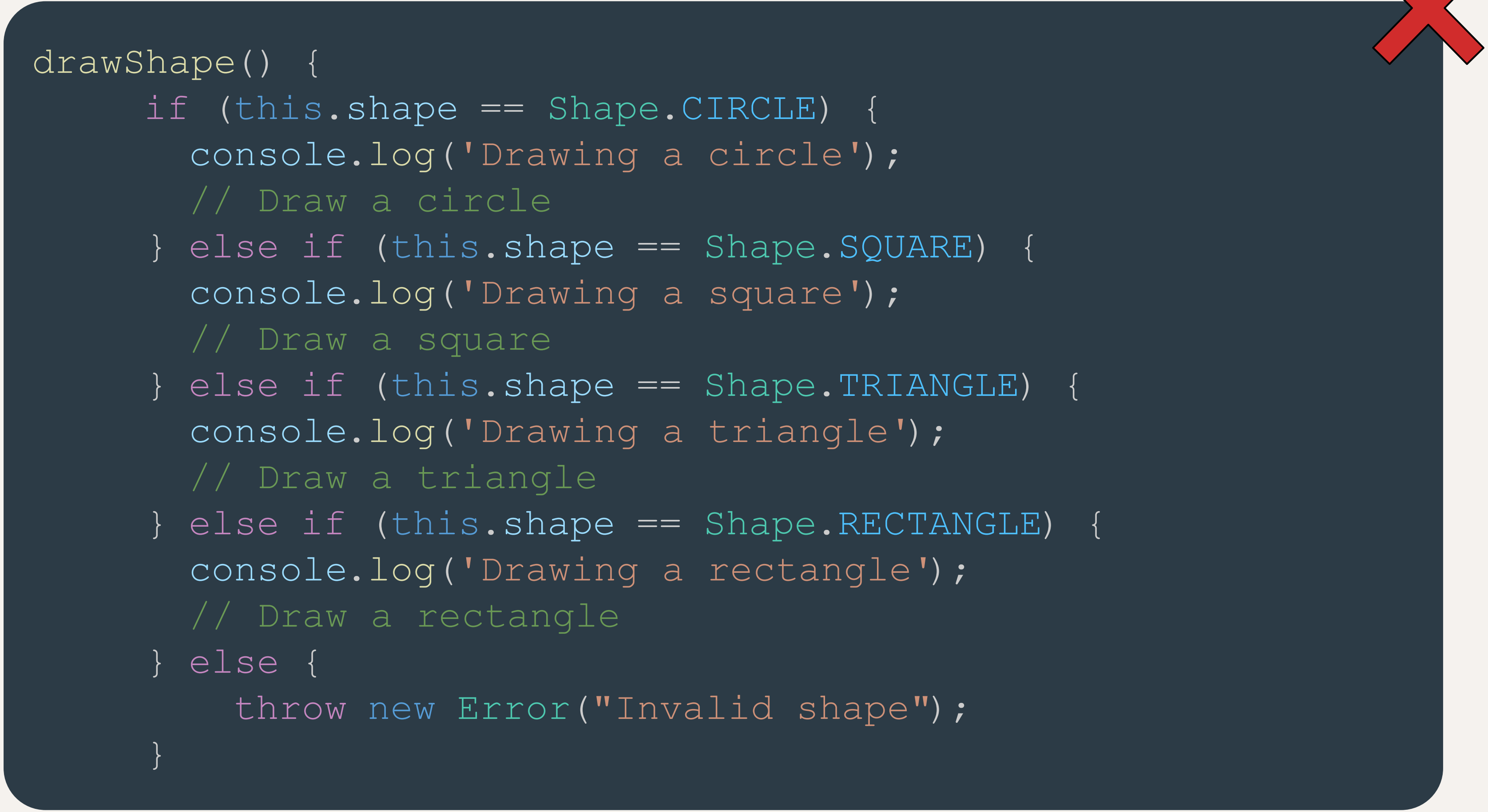


Strategy - Example





```
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;
 }
```



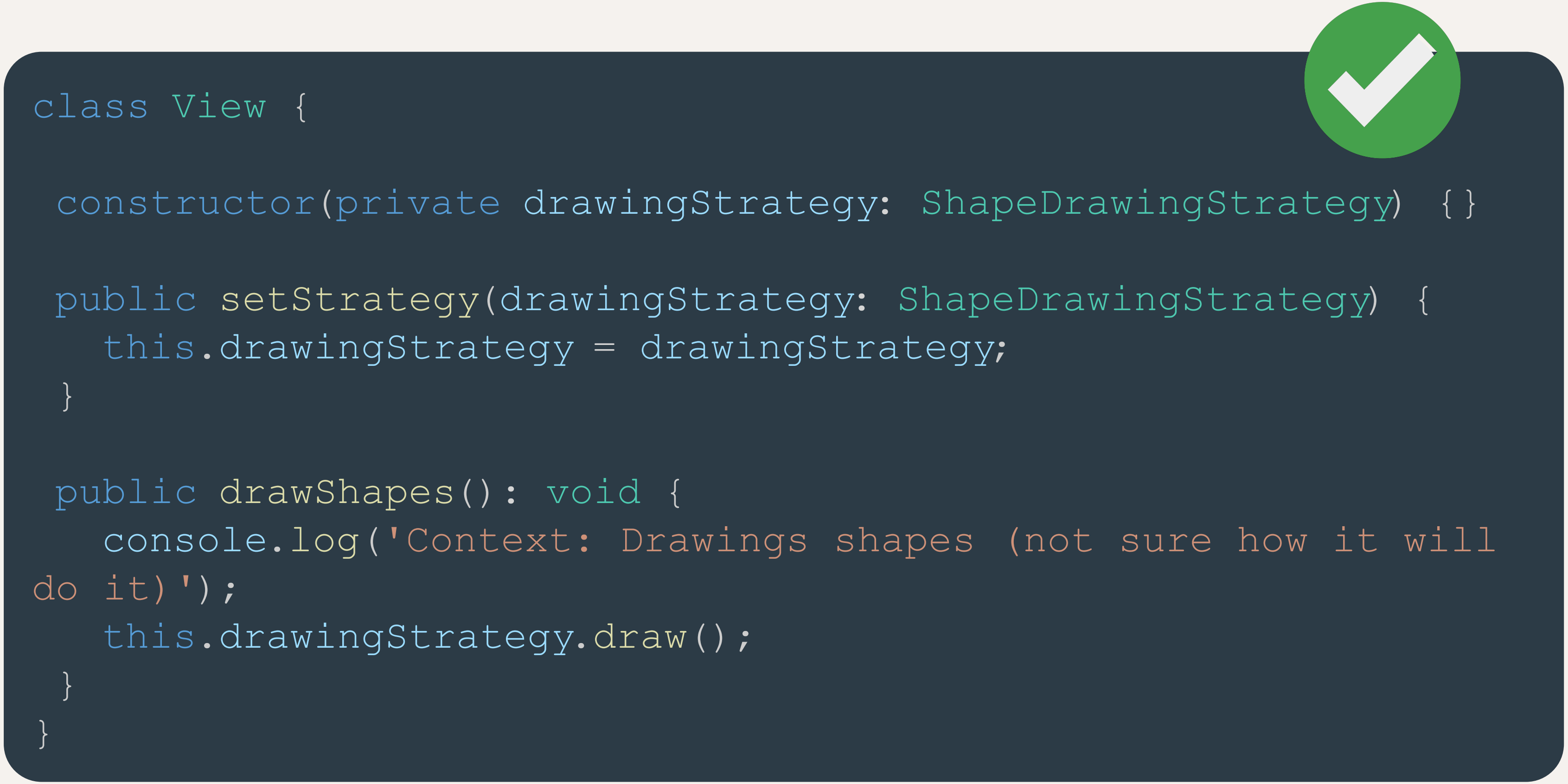
```
drawShape() {
     if (this.shape == Shape.CIRCLE) {
       console.log('Drawing a circle');
       // Draw a circle
     } else if (this.shape == Shape.SQUARE) {
       console.log('Drawing a square');
       // Draw a square
     } else if (this.shape == Shape.TRIANGLE) {
       console.log('Drawing 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

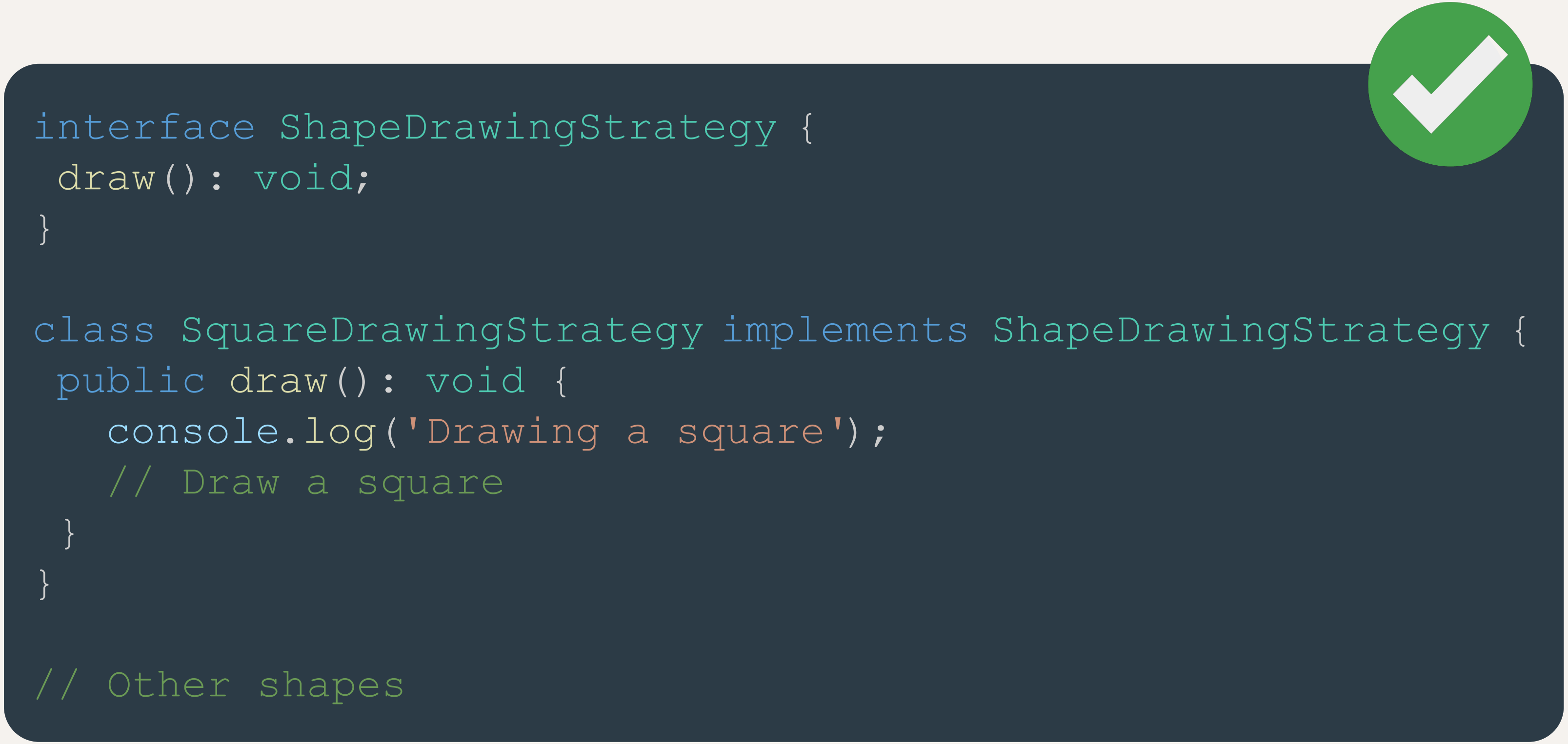
- The class makes something specific in different ways
- Unscalable code. Exponential growth!
- Violates the Open-Closed and SRP principles

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

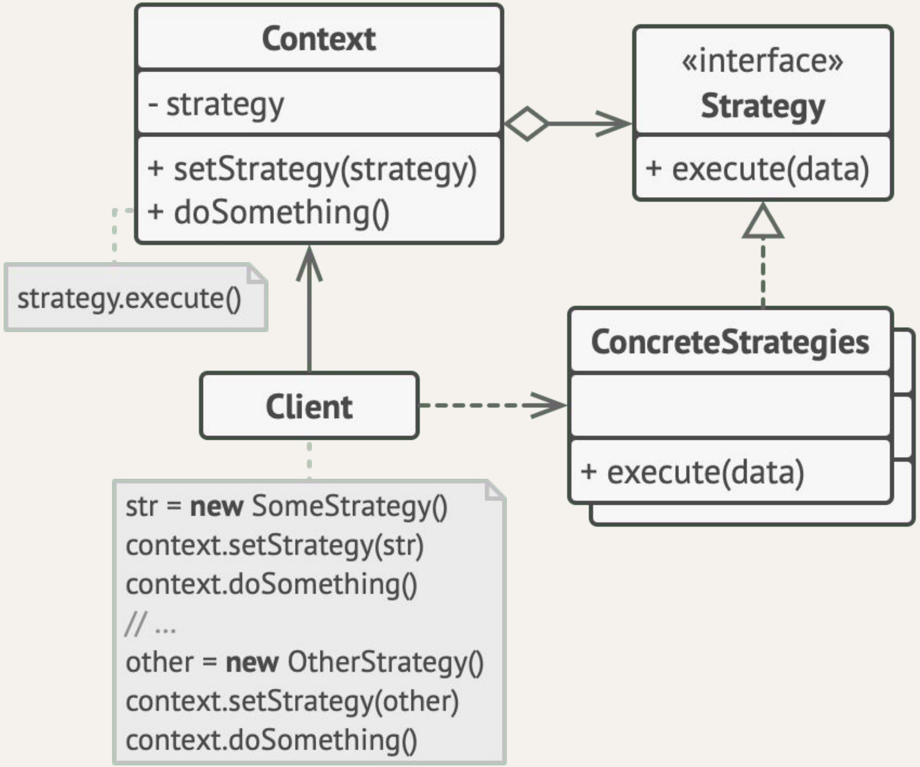


```
class View {

 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
```

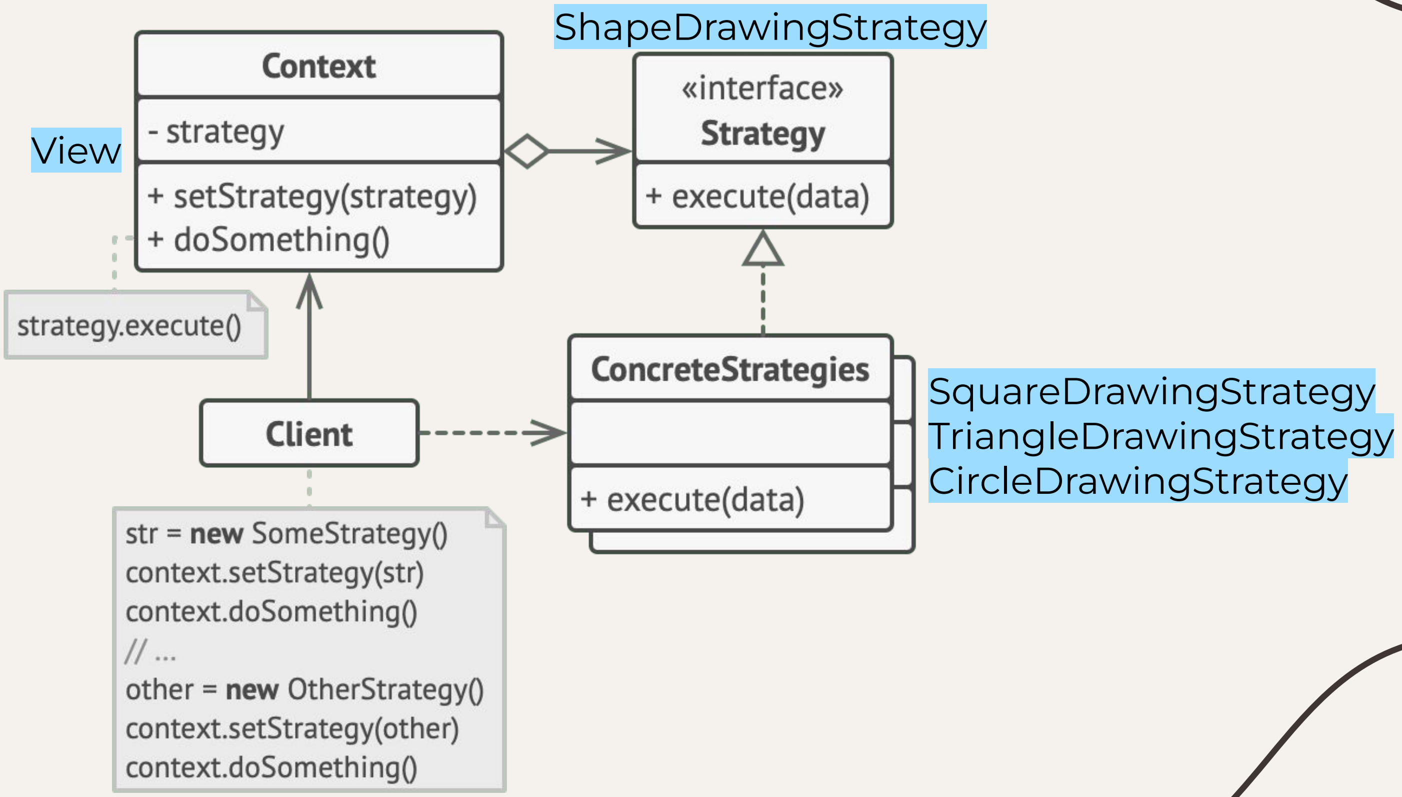
Strategy - Diagram



```
strategy.execute()
```

```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

Strategy - Diagram

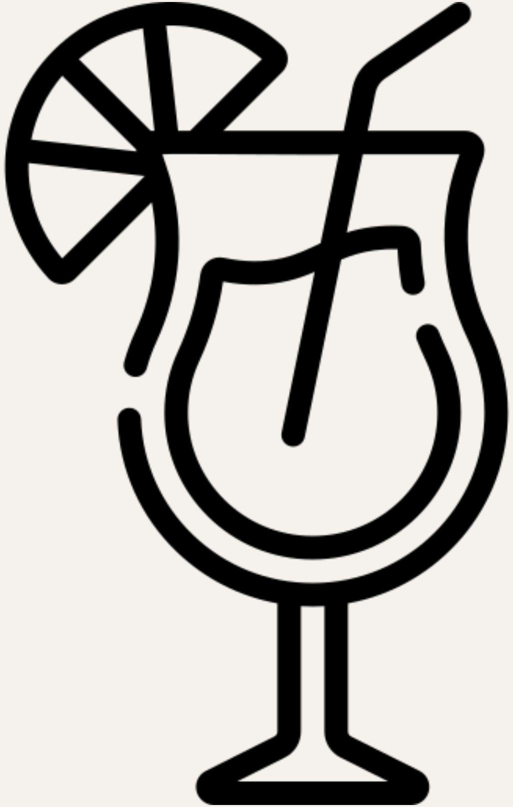


```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

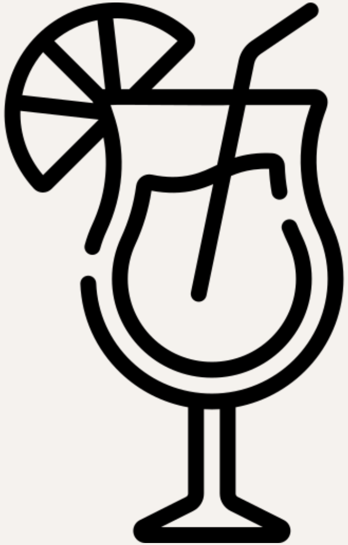
Template method ★★☆

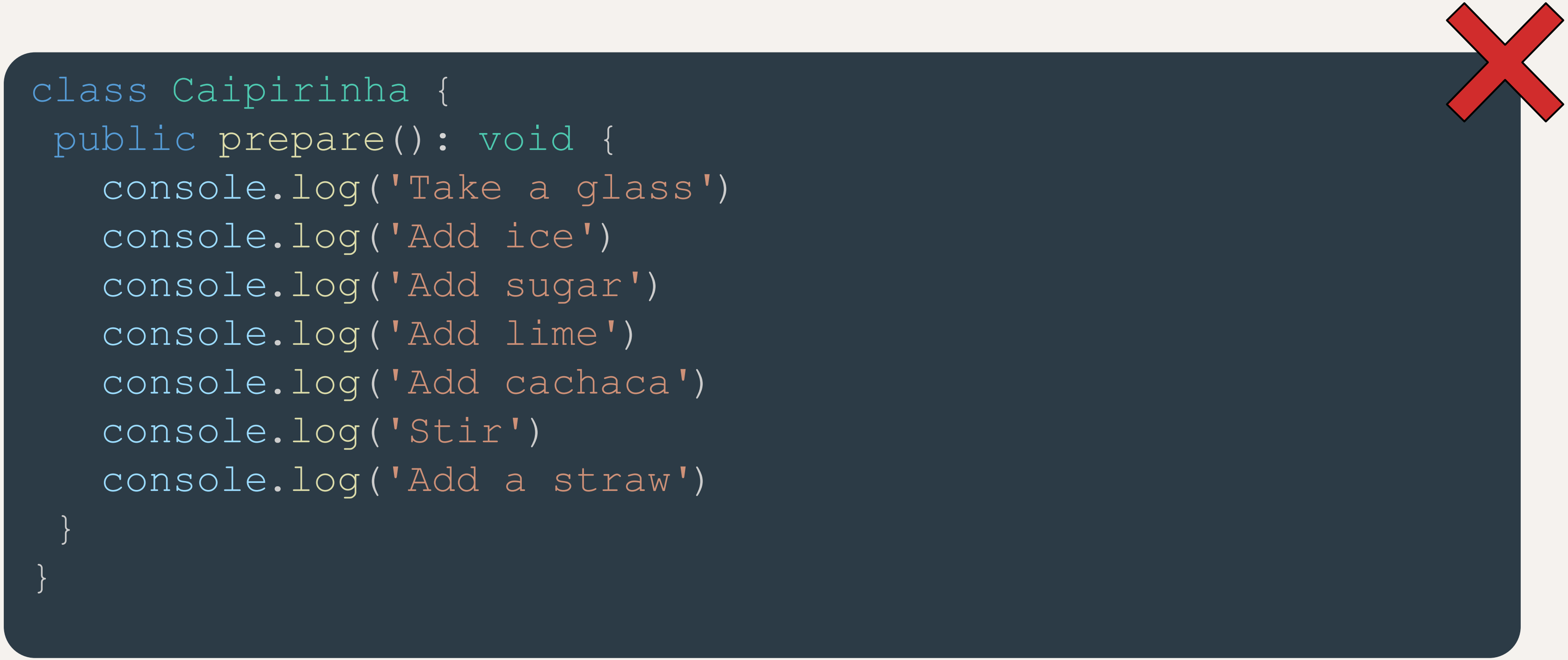
- Defines the skeleton of an algorithm in the superclass
- Lets subclasses override specific steps of the algorithm without changing its structure.
- Uses polymorphism

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')
 }
}
```

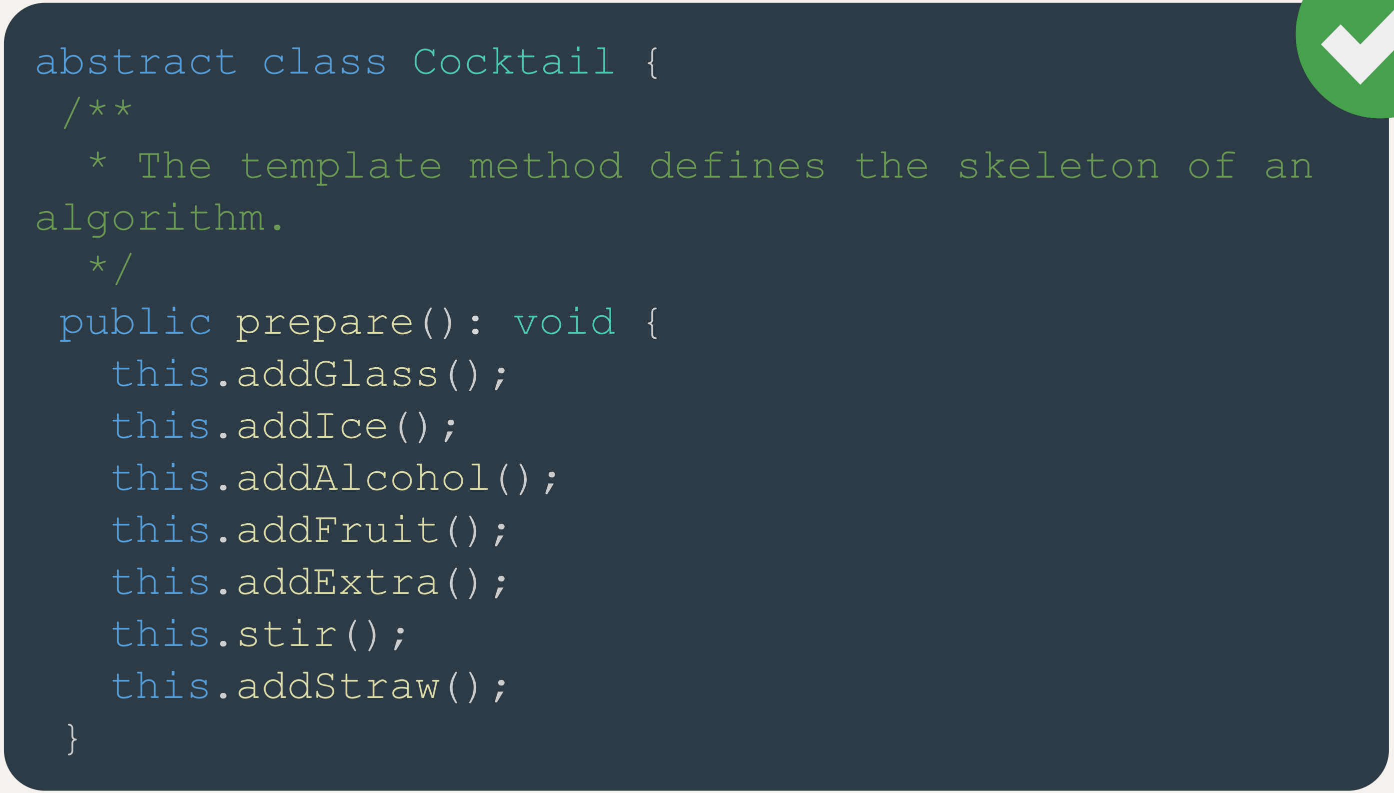
```
class Mojito {
 public prepare(): void {
   console.log('Take a glass')
   console.log('Add ice')
   console.log('Add sugar')
   console.log('Add lime')
   console.log('Add rum’)
   console.log('Add mint)
   console.log('Stir')
   console.log('Add a straw')
 }
}
```

Template method - Problems

- Duplicated code, a lot of duplicated code

Template method - Solution

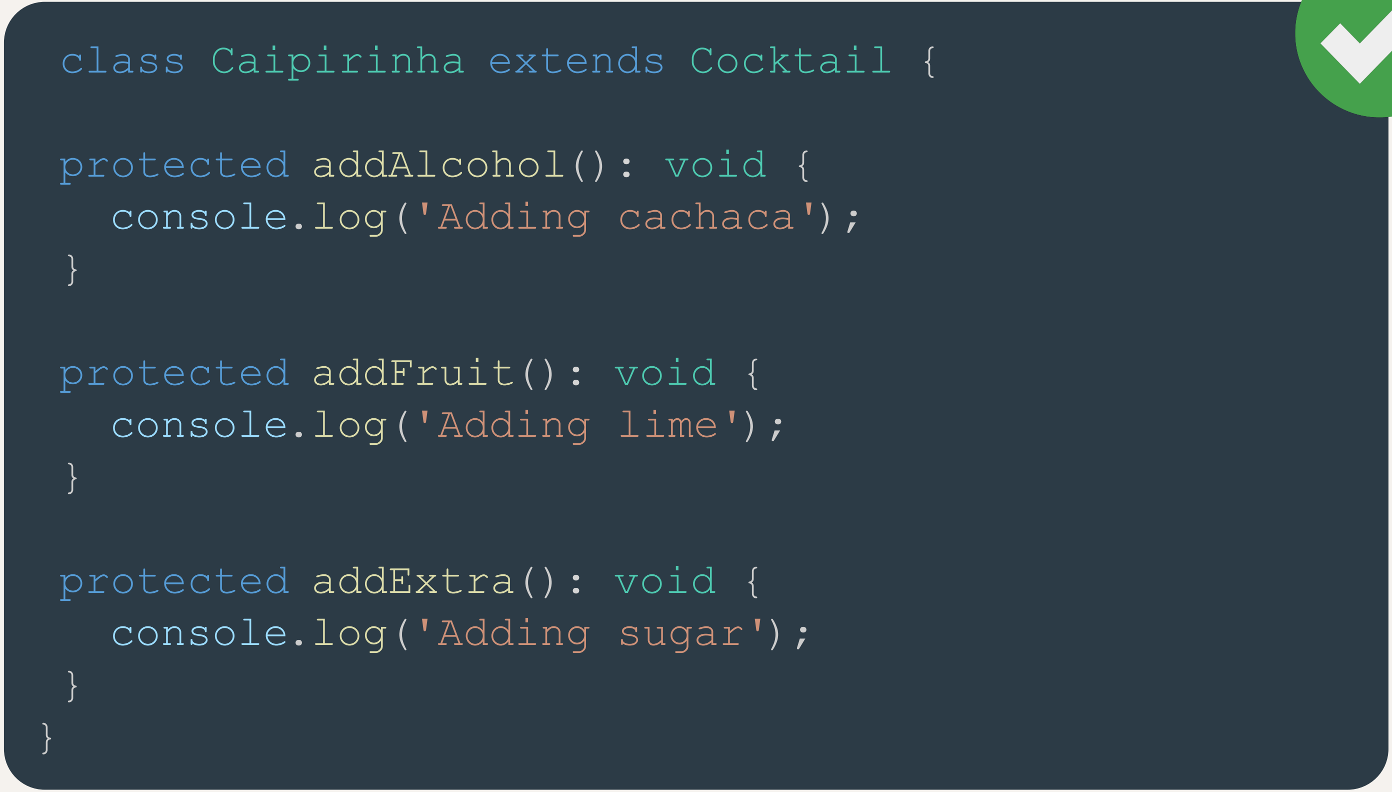
- Break down the algorithm into a series of steps
- Create a template method that puts together those steps in an abstract superclass
- Create as many subclasses as algorithms you need to implement



```
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;
```

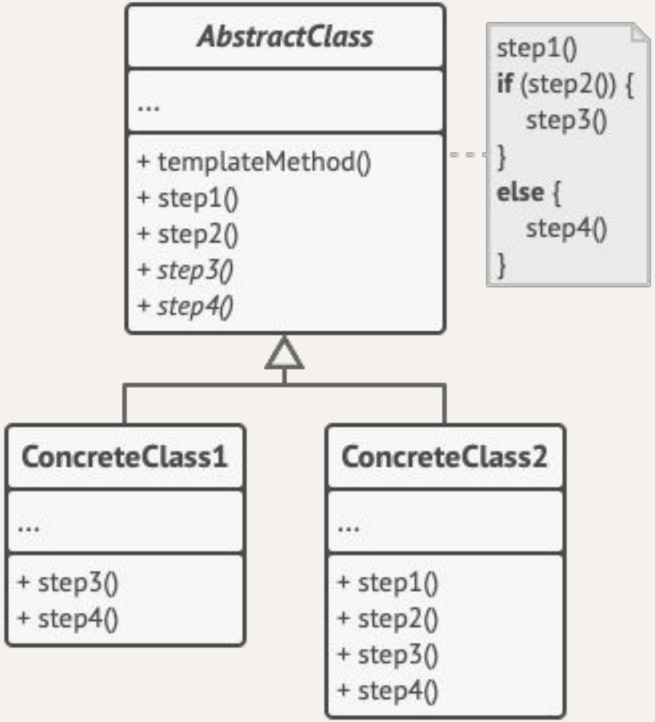


```
 class Caipirinha extends Cocktail {

 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



AbstractClass

...

+ templateMethod()
+ step1()
+ step2()
+ *step3()*
+ *step4()*

```
step1()
if (step2()) {
   step3()
}
else {
   step4()
}
```

ConcreteClass1

...

+ step3()
+ step4()

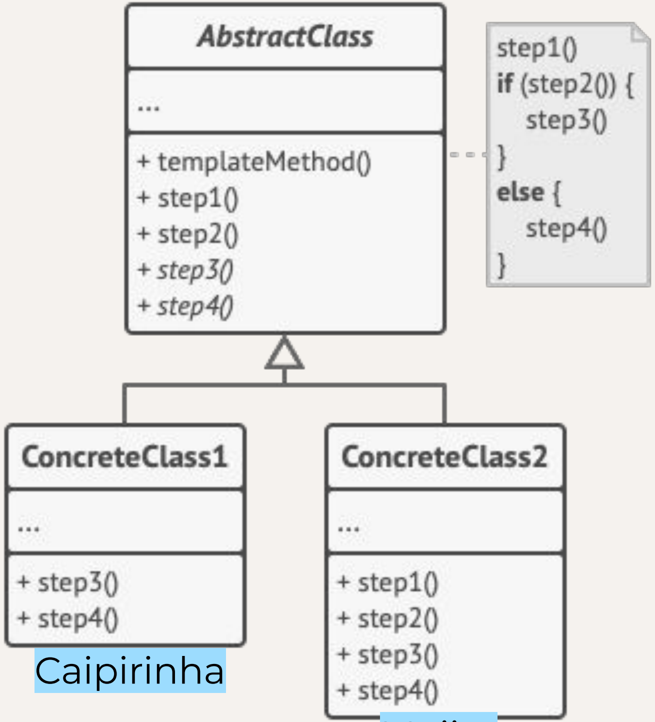
ConcreteClass2

...

+ step1()
+ step2()
+ step3()
+ step4()

Template method - Diagram

Cocktail



```
step1()
if (step2()) {
   step3()
}
else {
   step4()
}
```

Caipirinha

Mojito

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

- https://refactoring.guru/design-patterns/typescript
- https://www.dofactory.com/net/design-patterns
- https://learning.oreilly.com/library/view/head-first-design/0596007124/
- https://sourcemaking.com/design_patterns
- https://en.wikipedia.org/wiki/Software_design_pattern

- https://learning.oreilly.com/course/design-patterns-clean/9780135485965/
- https://www.geeksforgeeks.org/javascript-design-patterns/?ref=lbp
- https://absysnet.bbtk.ull.es/cgi-bin/abnetopac?TITN=8809

Any questions?

igor.dragone.13@ull.edu.es

jose.morera.27@ull.edu.es

Thanks for watching!