



Design Patterns



Outline

- Historical background
- What are design patterns
- Why you should learn design patterns
- Design patterns downsides
- Design patterns classification
- How to apply design patterns
- Design Principles
- Anti patterns
- Examples

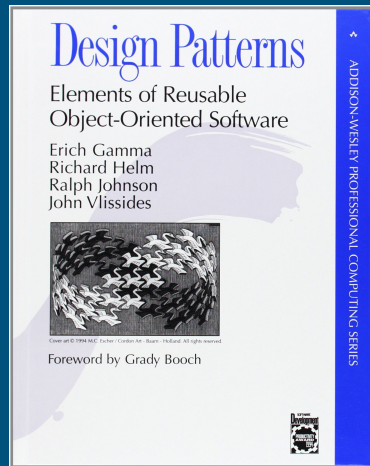
Historical Background

Historical Background

- The concept of patterns was first described in the late 1970s by Christopher Alexander, an architect.
- He described in his book “A Pattern Language: Towns, Buildings, Construction” a language for designing the urban environment.
- The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. In 1994, they published Design Patterns: Elements of Reusable Object-Oriented Software.
- The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly.

What are design patterns?

- Design patterns are general solutions to commonly occurring problems in software design.
- Blueprint not an implementation.
- Design patterns are discovered not invented.
- They're proven OO experience.
- Primary purpose is to increase software reuse and flexibility



Four Essential parts

- Intent
- Problem
- Solution
- Drawbacks

Why you should learn design patterns?

- You'll have a toolkit of tried and tested solutions to common problems in software design.
- Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
- You'll learn how to code maintainable and reusable software.
- Design patterns provide a shared language among developers.

Design patterns downsides

- Hard to debug when we have a lot of them in place.
- Sometimes design patterns add a lot of complexity.
- The size of our code might explode because we're thinking too far ahead.
- Unjustified use.

Scope of Design Patterns

A design pattern's scope specifies whether it applies primarily to classes or objects.

Object Patterns: specify the relationships between objects.

In general, the purpose of an object pattern is to allow the instances of different classes to be used in the same place in a pattern. Object patterns avoid fixing the class that accomplishes a given task at compile time.

Scope of Design Patterns (Continued)

Class Patterns: Specify the relationship between classes and their subclasses.

Thus, class patterns tend to use inheritance to establish relationships. Unlike object patterns and object relationships, class patterns generally fix the relationship at compile time.

Design Patterns Classification:

Structural Patterns: Are concerned with how classes and objects are composed to form larger structures, while keeping these structures flexible and efficient.

- **Structural class patterns:** Use inheritance to compose classes.
- **Structural object patterns:** Describe ways to assemble objects.

Design Patterns Classification (Continued)

Behavioural patterns: Concerned with interactions between objects and the assignment of responsibilities between them.

- Behavioral class patterns: Use inheritance to describe algorithms and flow of control.
- Behavioral object patterns: Describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Design Patterns Classification (Continued)

Creationional patterns: Deal with object creation mechanisms.

- Creational class patterns: Defer some part of object creation to subclasses.
- Creational object patterns: Defer object creation to another object.

Design Patterns Classification:

Table 1-1. *The Gang-of-Four Design Patterns Classified*

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Class Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

How design patterns are applied

- Design patterns usually find their way into the code through refactoring.
- Implement what you want in the simplest way possible, later on if you find that this code changes and it would make things easier if you apply a certain design pattern that's when you put the pattern in place.

Design Principles

Design Principles:

Design principles provide high-level guidelines to design better software applications. They do not provide implementation guidelines and are not bound to any programming language.

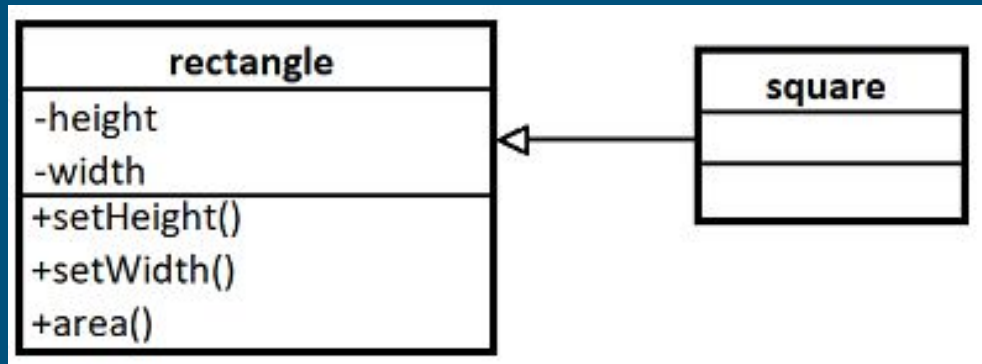
SOLID Design Principles

One of the most popular sets of design principles, consists of:

- Single responsibility principle: A class should have one, and only one, reason to change.
- Open-closed principle: You should be able to extend a class's behavior without modifying it.

SOLID Design Principles (Continued)

- Liskov Substitution Principle: Every derived class should be substitutable for its parent class.

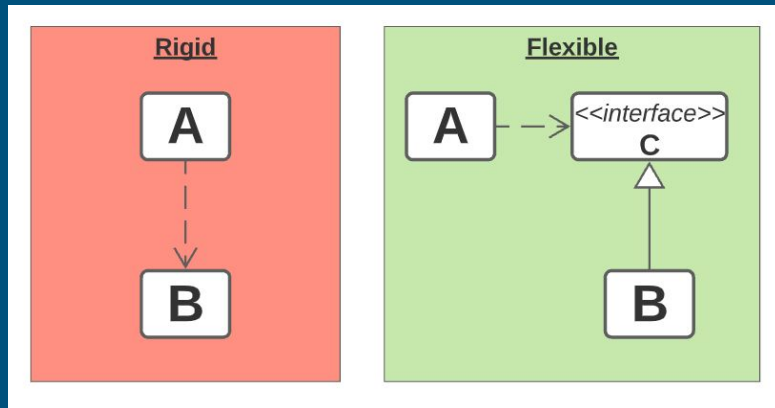


SOLID Design Principles (Continued)

- Interface Segregation Principle: Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not need.

SOLID Design Principles (Continued)

- Dependency Inversion Principle: states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.



Anti-Patterns

An anti-pattern is the opposite side of the design pattern. You can also call it design smell which is caused by bad software design.

Examples:

- Golden hammer: a commonly occurring solution to a problem that generates decidedly negative consequences.
- God class: Having all the functionality handled by one class.
- Copy and paste programming.

— That's good and all, but everybody needs some examples to truly understand.



Ok then, here are some!

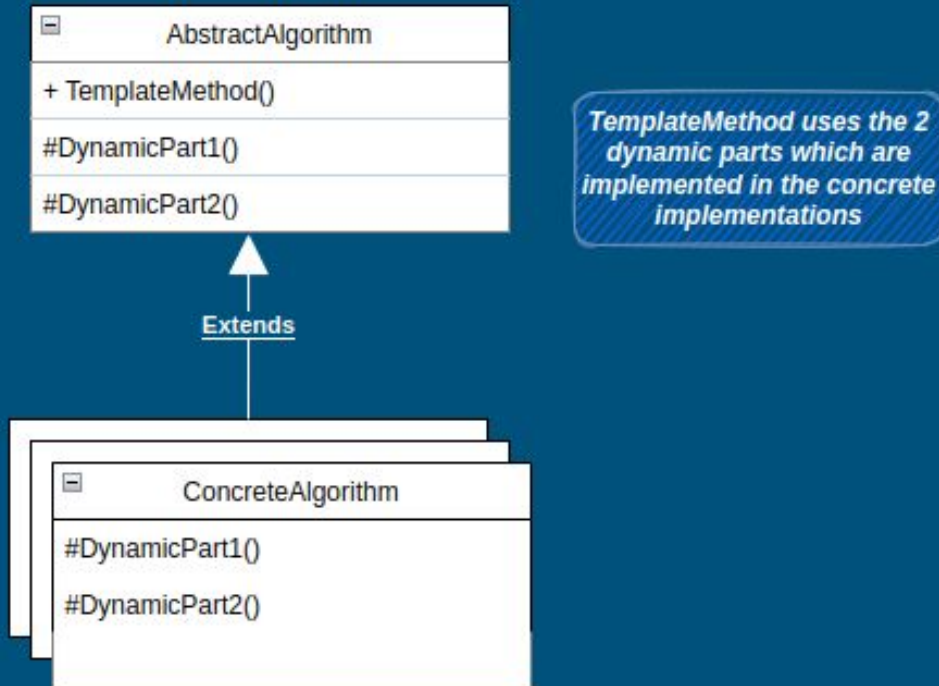


The Template Method Pattern!

The Intent Behind it

Define an algorithm while keeping the ability to change certain parts of the algorithm using inheritance.

A UML should be helpful



Pseudo Code Example

```
class AbstractClass
{
    Protected abstract DynamicPart1();
    Protected abstract DynamicPart2();

    Public final TemplateMethod()
    {
        //do some stuff
        DynamicPart1();
        //some other stuff here
        DynamicPart2();
        //final touches here
    }
}
```

```
class ConcreteClass
{
    Protected DynamicPart1()
    {
        //some implementation
    }
    Protected DynamicPart2()
    {
        //some implementation
    }
}
```

Now we can use the whole algorithm by only writing specific parts of it and we don't even need to know how the algorithm works.

If we were to change how our algorithm work, we would change the parent class and vualá, all concreations changed their behavior.


Is there a real world example where I would want to use the template method pattern?

Yes, actually you're probably using right now. Most frameworks use this approach where extend a class and override a method to add your functionality and then the frameworks proceed with it's magic.

But isn't there any drawbacks to it?!

Of course there are.

When you are not sure if the structure is constant across all algorithms you should stay away from Template Method; as it's expensive to modify it.



Let's discover another pattern
using a problem

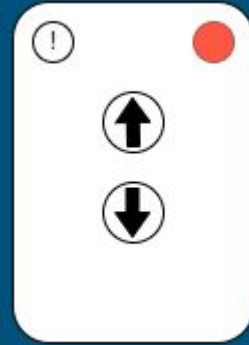
We have a system that consists of smart device (air conditioner) and a remote controller for that device.

Air Conditioner Interface

```
class AirConditioner
{
    public void TurnOn();
    public void TurnOff();
    public int GetTemperature();
    public void SetTemperature(int temperature);
}
```

We are supposed to implement a class that represents the behavior of this remote

This is how the controller looks.



Remote Controller Implementation

```
class Controller
{
    private readonly AirConditioner ac;
    private bool isOn;
    public Controller(AirConditioner ac);
    public void PressUp();
    public void PressDown();
    public void PressOnOff();
}
```

Constructor Implementation

```
public Controller(AirConditioner ac)
{
    this.ac = ac;
    isOn = false;
}
```

PressUp Implementation

```
public void PressUp()
{
    ac.SetTemperature(GetTemperature() + 1);
}
```

PressDown Implementation

```
public void PressDown()
{
    ac.SetTemperature(GetTemperature() - 1);
}
```


PressOnOff Implementation

```
public void PressOnOff()
{
    if(isOn)
        ac.TurnOff();
    else
        ac.TurnOn();
    isOn = !isOn;
}
```

Nice, are we done already?

No, of course not :^{

Requirements Update!

Our company now provides brand new smart light bulb that can be controlled using the same controller you just have to reprogram it!!

Light Bulb Interface

```
class LightBulb
{
    public void TurnOn();
    public void TurnOff();
    public float GetLightIntensity();
    public void SetLightIntensity(float lightIntensity);
}
```

Ok so we have to change our code. That sucks but this is the reason I'm getting payed, life is life I guess.

New Controller Implementation

```
class Controller
{
    private readonly AirConditioner ac;
    private readonly LightBulb lb;
    private bool isOn;
    public Controller(AirConditioner ac);
    public Controller(LightBulb lb);
    //the rest is the same
}
```

Constructors Implementation

```
public Controller(AirConditioner ac)
{
    this.ac = ac;
    isOn = false;
}
```

```
public Controller(LightBulb lb)
{
    this.lb = lb;
    isOn = false;
}
```

New PressUp Implementation

```
public void PressUp()
{
    if(ac is not null)
        ac.SetTemperature(GetTemperature() + 1);
    else
        lb.SetLightIntensity(GetLightIntensity() * 1.1);
}
```


New PressDown Implementation

```
public void PressDown()
{
    if(ac is not null)
        ac.SetTemperature(GetTemperature() - 1);
    else
        lb.SetLightIntensity(GetLightIntensity() / 1.1);
}
```


New PressOnOff Implementation

```
public void PressOnOff()
{
    if(lb is null)
        TurnAcOnOff();
    else
        TurnLbOnOff();
    isOn = !isOn;
}
```

New PressOnOff Implementation cont.

```
private void TurnLbOnOff(){  
    if(isOn)  
        lb.TurnOff();  
    else  
        lb.TurnOn();  
}
```

```
private void TurnAcOnOff(){  
    if(isOn)  
        ac.TurnOff();  
    else  
        ac.TurnOn();  
}
```



Ok it should work now.
Are we done?

No! This code is ugly, and here's why (SOLID):

- This code doesn't adhere to the SRP. It has many reasons to change.
- Neither does it adhere to the OCP. We need to modify the existing code to add functionality.
- Even the DIP is broken. We are depending on implementation details.

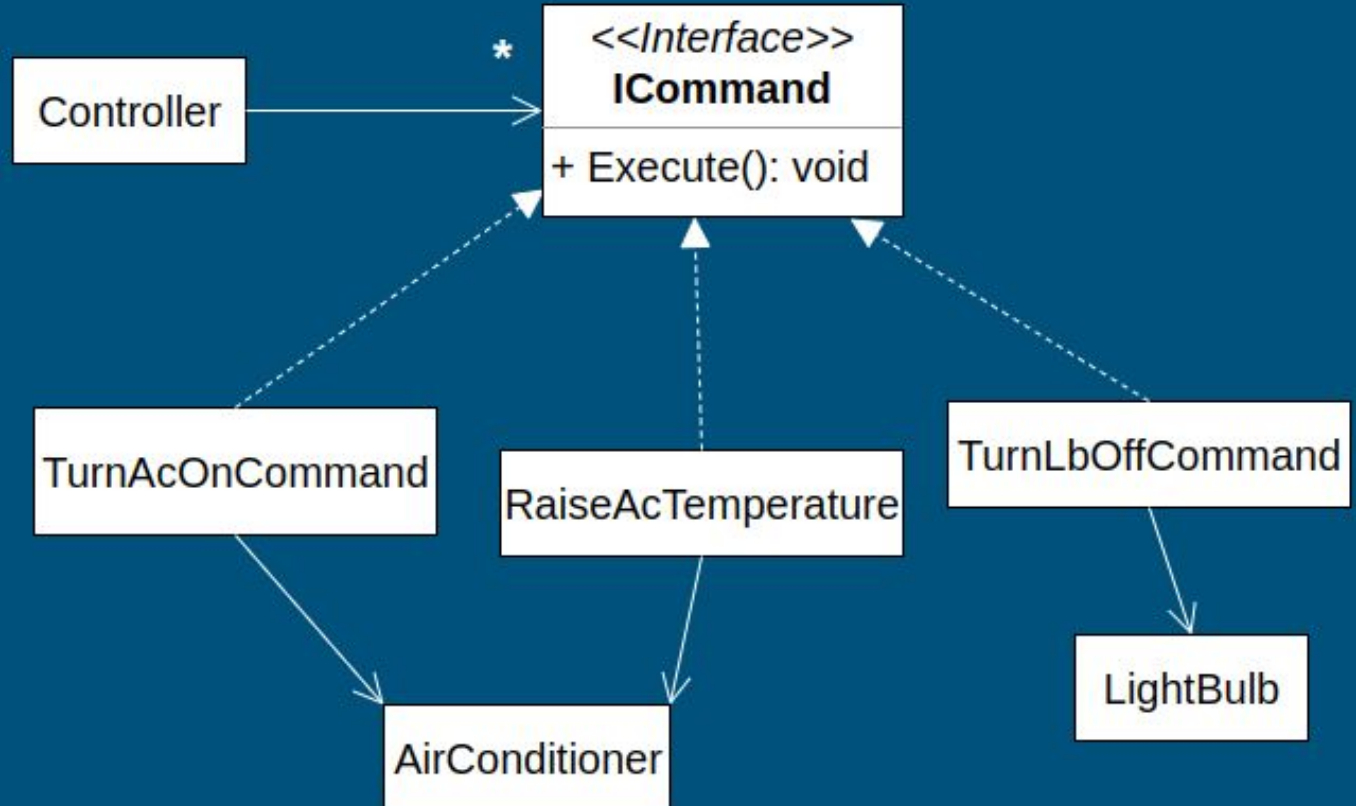
Even if you reason about it you would see that:

- you need to fix your implementation every time we release a new device.
- Plus it's going to become an if-else hell very quickly

My suggested solution is to “encapsulate what changes”.

We will try to encapsulate the command of the button.

New Design UML



Refactored Controller Implementation

```
class Controller
{
    public ICommand TurnOnOffCommand{ private get; set;}
    public ICommand ArrowUpComand{ private get; set;}
    public ICommand ArrowDownComand{ private get; set;}
    public Controller(turnOnOff, up, down);
    //the rest is the same
}
```

Constructor Implementation

```
public Controller(ICommand [turnOnOff, up, down])  
{  
    TurnOnOffCommand = trunOnOff;  
    ArrowUpCommand = up;  
    ArrowDownCommand = down;  
}
```

PressUp Implementation

```
public void PressUp()
{
    ArrowUpCommand.Execute();
}
```

PressDown Implementation

```
public void PressDown()
{
    ArrowDownCommand.Execute();
}
```

PressOnOff Implementation

```
public void PressOnOff()
{
    TurnOnOffCommand.Execute();
}
```

Raise AC Temperature Command

```
class RaiseAcTemperature : ICommand{
    private AirConditioner ac;
    public RaiseAcTemperature(AirConditioner ac){
        this.ac = ac;
    }
    public void Execute(){
        ac.SetTemperature(GetTemperature() + 1);
    }
}
```

Turn Light Bulb On/Off Command

```
class TurnLightBulbOnOff : ICommand{
    private LightBulb lb;
    private bool isOn;
    public RaiseAcTemperature(LightBulb lb){
        this.lb = lb;
        isOn=false;
    }
    public void Execute(){
        isOn ? lb.TurnOff() : lb.TurnOn() ;
        isOn = !isOn;
    }
}
```



Aaaand so on.

You get it by now.

Modifications' Benefits

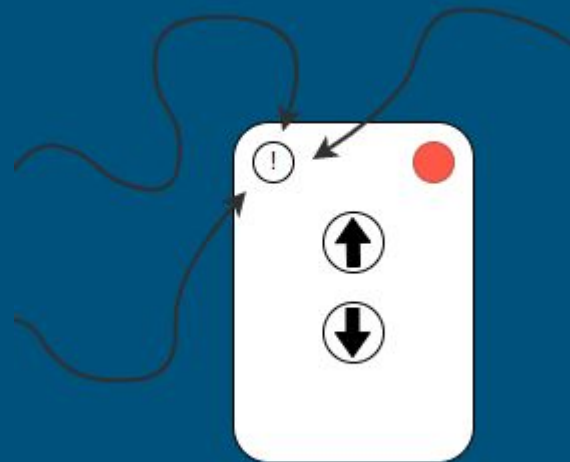
- SRP is followed; the controller class changes only when the physical controller changes.
- OCP is followed; adding functionality to our system doesn't require code modification, adding a new implementation is sufficient.
- DIP is followed; controller class knows nothing about concrete devices and their interfaces. It depends only on the ICommand interface.

But wait! We are not done with our example.

Remember this button?

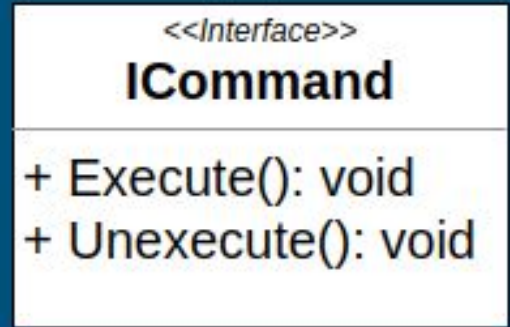
We want to utilize it.

Let's use it as a revert button!



First, we define the revert of a command

```
class RaiseAcTemperature : ICommand{
    //the rest
    public void Execute(){
        ac.SetTemperature(GetTemperature() + 1);
    }
    public void Unexecute(){
        ac.SetTemperature(GetTemperature() - 1);
    }
}
```

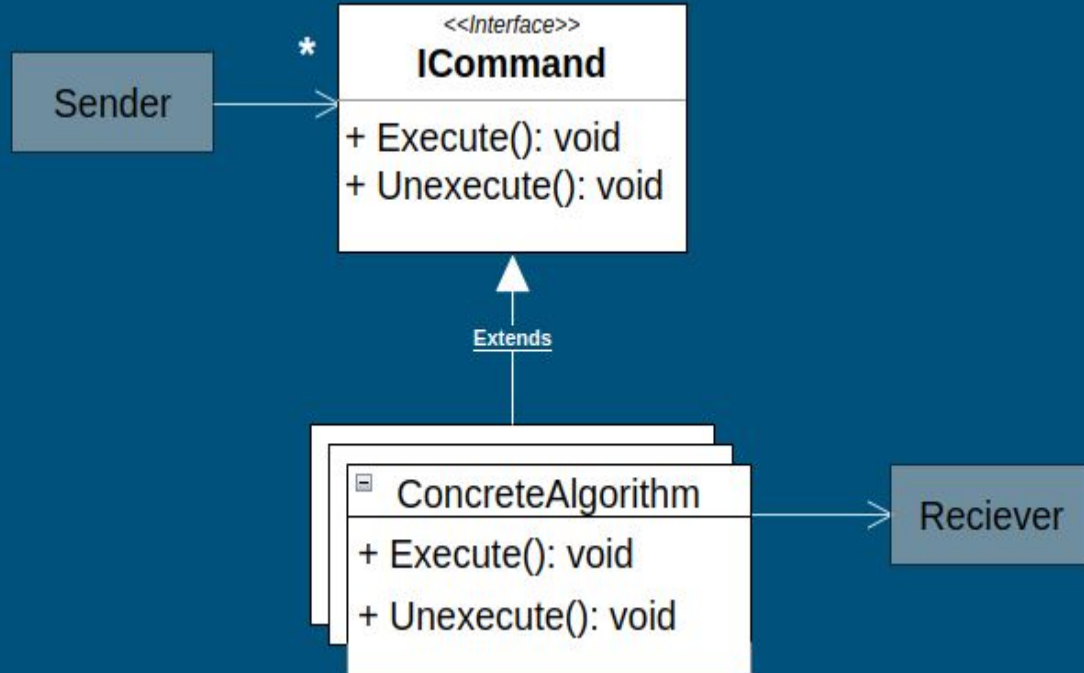


Controller With Revert Implementation

```
class Controller
{
    //previous properties stay the same
    private readonly Stack<ICommand> commands;
    // in the end of each command call add the
    // executed command to the stack
    public PressRevert(){
        commands.Pop()?.Unexecute();
    }
}
```

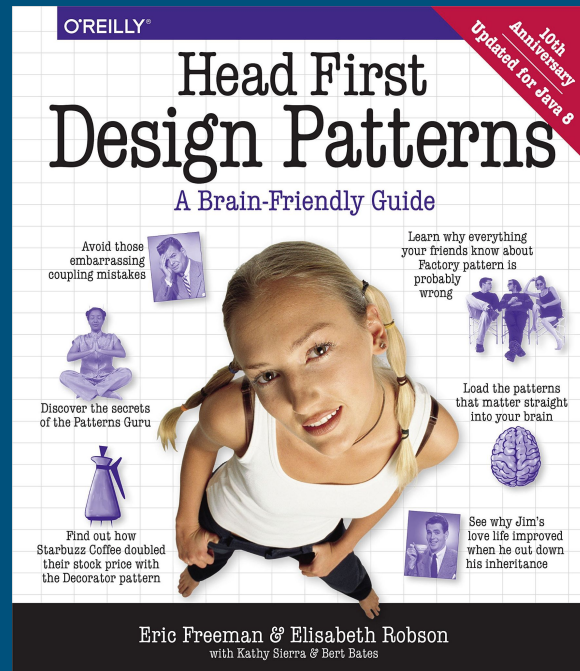
And This Was The Command Pattern!

It encapsulates a request as an object thereby letting you parameterize other objects with different requests queue or log request and support undoable operations.



References:

- Head First Design Patterns.
- Design Patterns: Elements of Reusable Object-Oriented Software by GOF.
- Design Patterns in Object Oriented Programming by Christopher Okhravi
- C# 10 Design Patterns by Kevin Dockx.
- The Catalog of Design Patterns (refactoring.guru)
- Design Patterns (umn.edu)
- Distinguish between Class and Object [Patterns] (gofpatterns.com)



Thanks!