

# Design Patterns explained with sketches

by Steven Giesel

# Table of contents

## Prolog

- 1 What is a “Design Pattern”?
- 2 Disclaimer

## Behavioral

- 3 Mediator
- 4 Strategy
- 5 Memento
- 6 Chain of Command
- 7 Iterator

## Creational

- 8 Singleton
- 9 Builder
- 10 Factory method

## Structural

- 11 Adapter
- 12 Proxy
- 13 Facade

## Epilog

- 14 Thanks
- 15 Version

# What is a “Design Pattern”?

Design patterns describe a solution to a common set of problems. Like a blueprint, which you can customize for your specific problem you want to solve. There are basically 3 categories we can put those design patterns in:

- **Creational** gives a mechanism how we can create objects with the most flexibility
- **Structural** is about giving flexibility and efficiency to a larger structure of objects
- **Behavioral** is about how objects or components communicate with each other as well as how they share responsibilities.

This small eBook gives you a nice introduction or refresher over some of them.

# Disclaimer



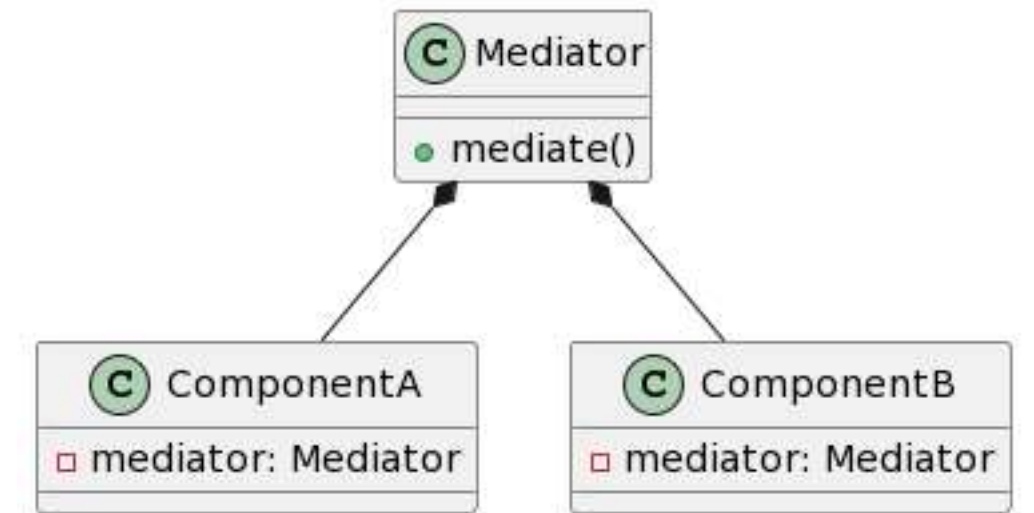
Once you have learned a new design pattern you want to use it everywhere. But there is a certain risk that you make your code more complex and less readable just by "over-introducing" certain patterns. Sometimes a new `MyObject()` is easier than using a factory.

I will just show some examples in this small eBook. As always practice makes the master.

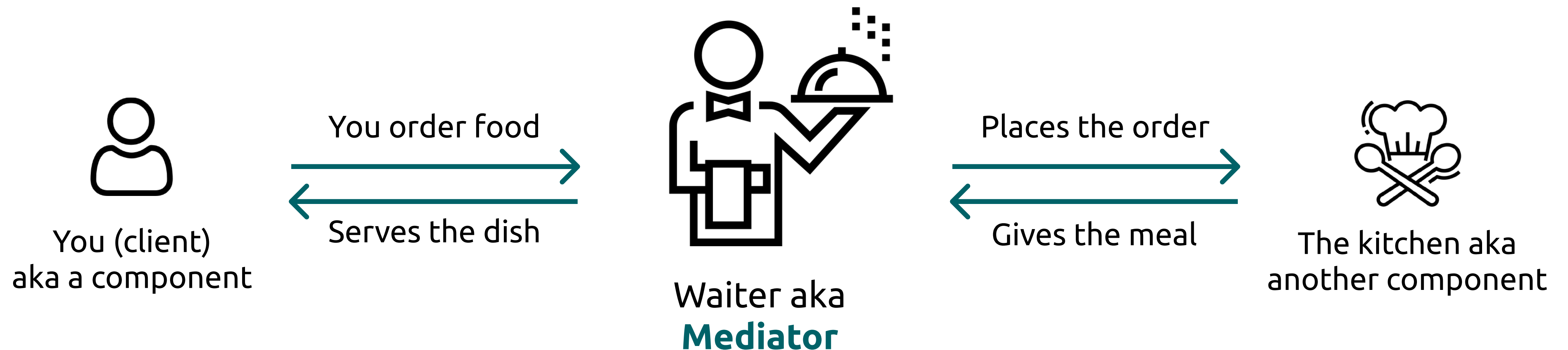
**“If all you have is a hammer, everything looks like a nail.”**

# Mediator

## Behavioral



The waiter mediates between you (the client) and the kitchen. Neither the kitchen nor you know directly from each other. He takes your order and goes with that to the kitchen, which then will (hopefully) come back with food.



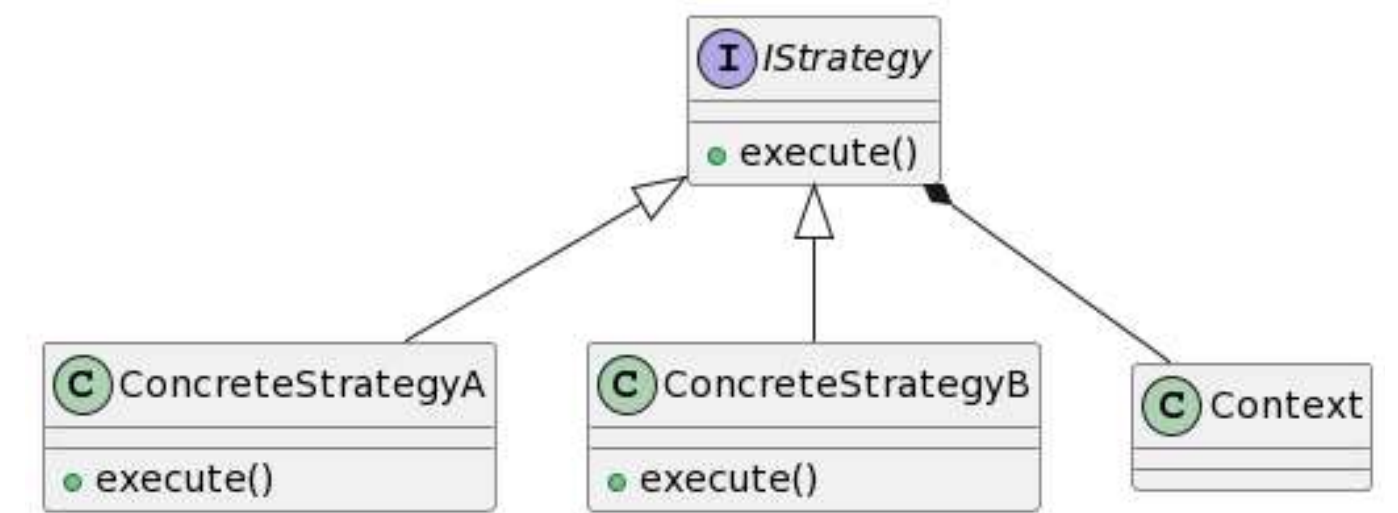
### Take Away

- Reduces dependencies between components
- Change interactions between objects independently

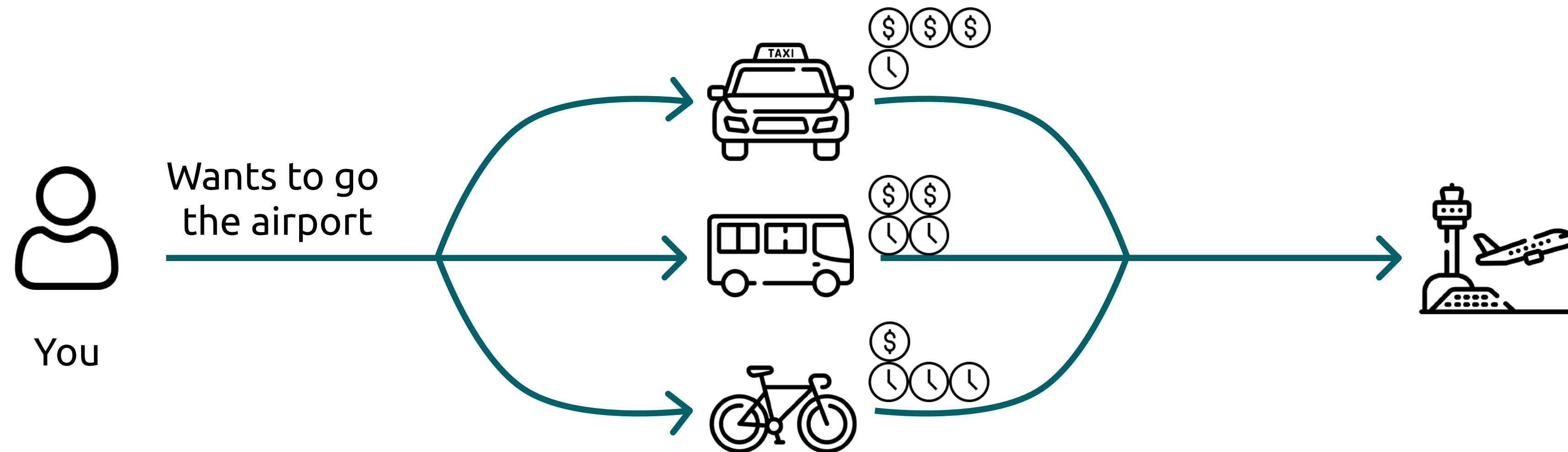
[More info](#)

# Strategy

## Behavioral



You have different **strategies** to go to the airport (by taxi, bus or bicycle). You decide in the moment, depending on some conditions like traffic and cost, which means of transportation you use. All of them are similar and they serve the same purpose.



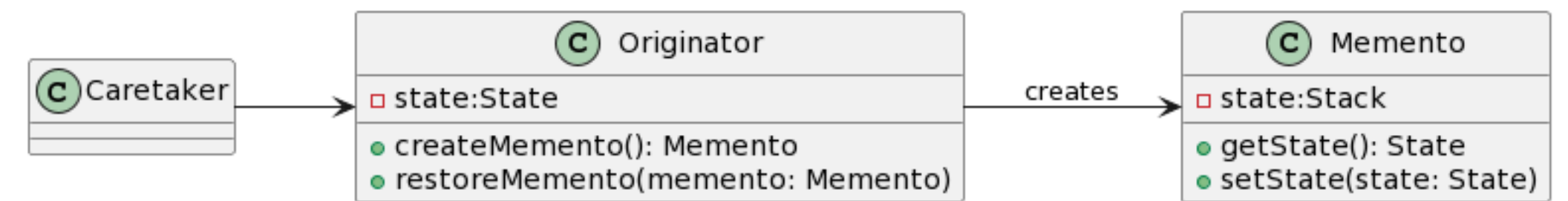
### Take Away

- Easy to swap out algorithmn. Also they are decoupled from the other logic
- Goes hand in hand with Open/Closed Principle as well as “favor composition over inheritance”
- Isolate details away (seperation of concerns)



# Memento

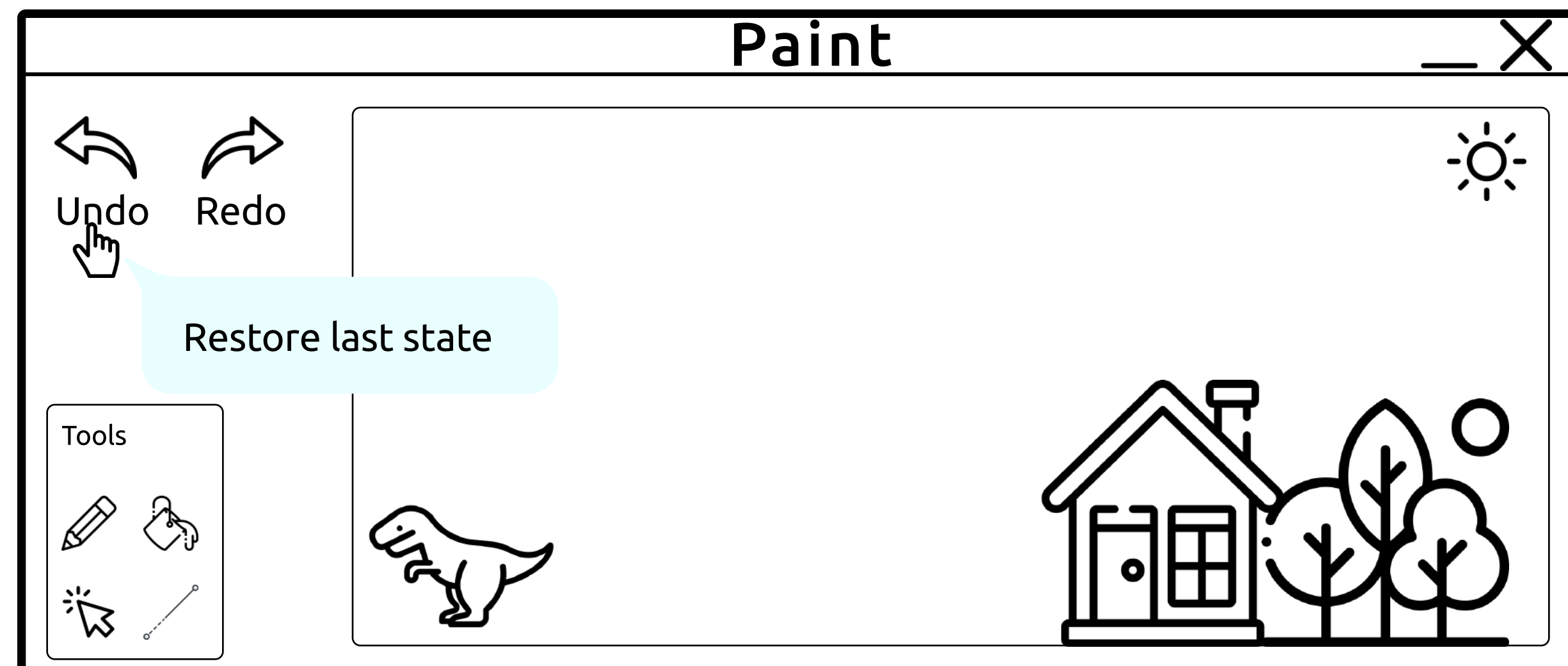
## Behavioral



Paint **memorizes** the last actions we did, so if we accidentally add a dinosaur to our picture, Paint can revert or undo to the last state. So Paint stores every state of our nice picture!



Damn it, I accidentally drew a dinosaur! Let's revert the last action!

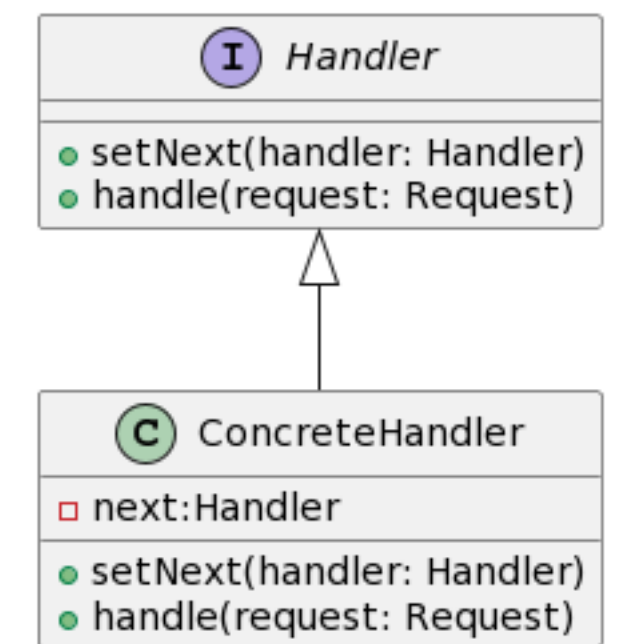


## Take Away

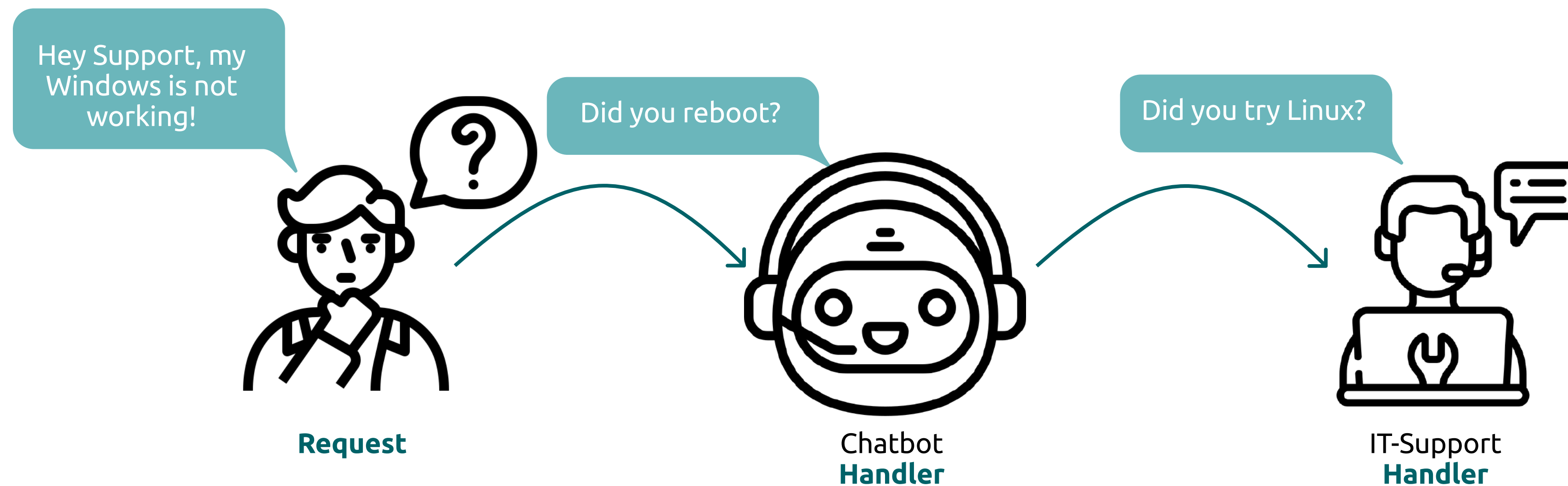
- Store a snapshot of the state without revealing the state to the outside world
- Every concern is encapsulated in its own object
- The state can grow quite fast if not controlled
- Undo-Redo can be done with two stack's with the **memento** pattern

# Chain of Responsibility

## Behavioral



There is a clear **chain** of people involved when calling the IT support. First we have a chatbot, which tries to solve our issue. If that doesn't help, we go further with a IT support person, which tries to resolve our issue.



### Take Away

- Perfect if we want to process something in specific order
- Separation of concerns - Each command handler can do one thing
- We can introduce new handlers with ease - Open/Closed Principle



# Iterator

## Behavioral



When pressing the next button in your music app, you **iterate** through your playlist one by one. You don't care how the app organizes the list, you just want to get away from getting rick rolled!



### Take Away

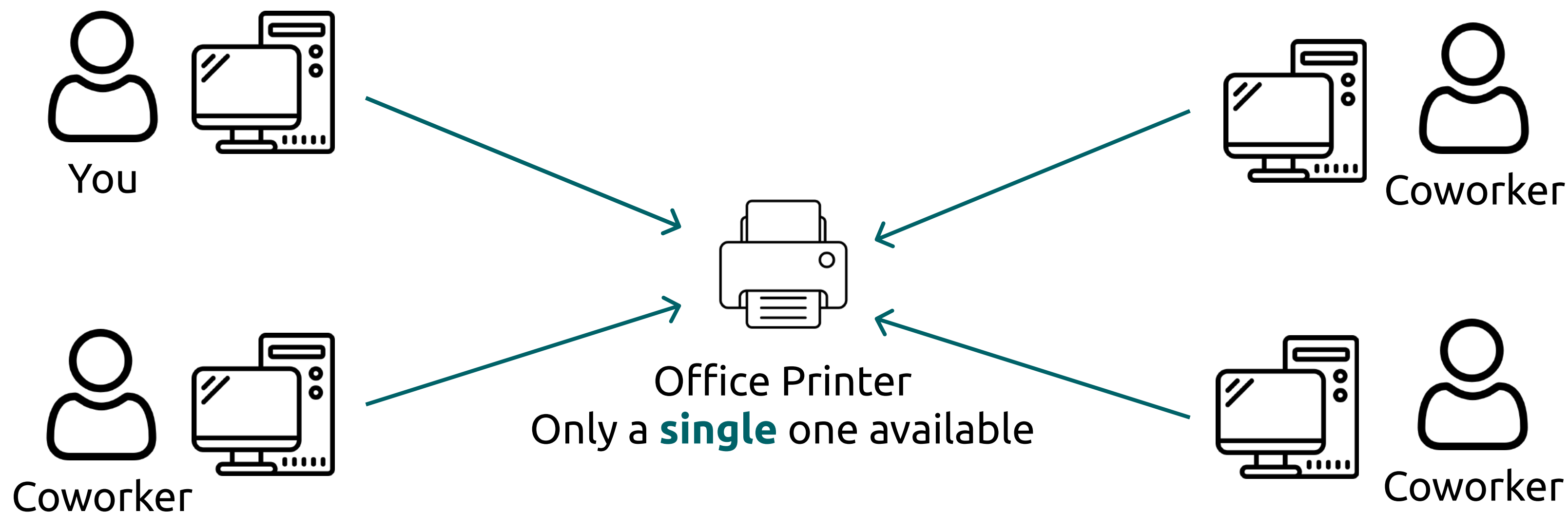
- Every **foreach** / **IEnumerable** in .NET uses the **Iterator** pattern
- Single Responsibility Principle: We can abstract away how we iterate through a complex object
- The UML describes how the .NET type **IEnumerable** works

# Singleton

Creational

	Singleton
	instance:Singleton
	Singleton():Singleton
	getInstance():Singleton

Imagine your office has one printer, and one printer only. You get the same printer instance independent if you choose the printer from your PC or from your coworkers PC.

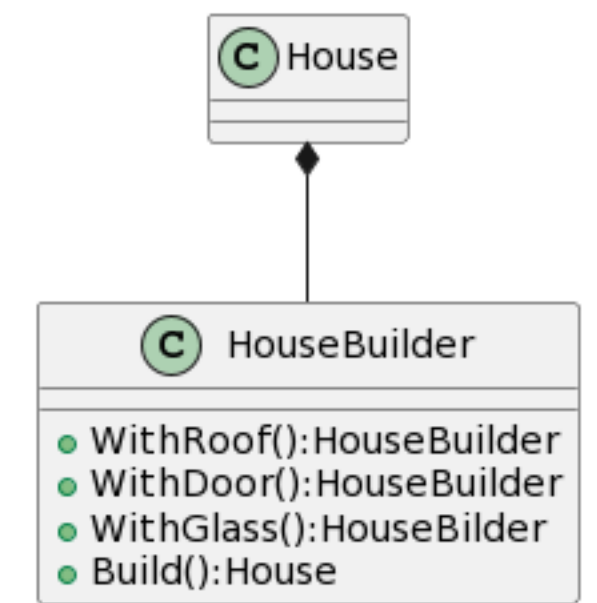


## Take Away

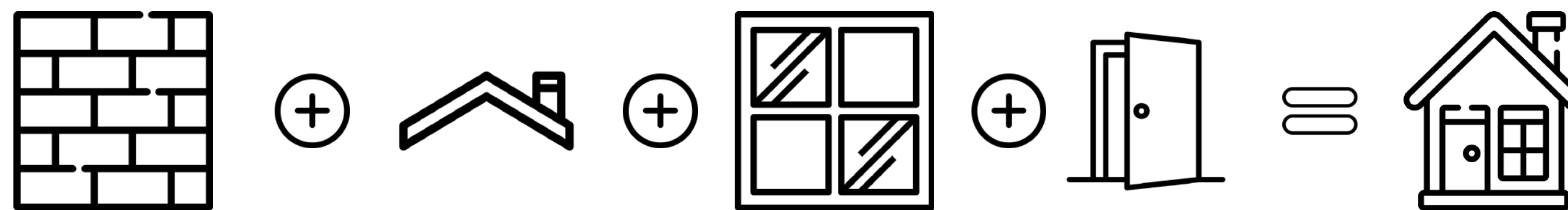
- Only one instance, which is only initialized once.
- Can be useful for things like logging
- Be careful though, as it can hide bad design and violates the "Single Responsibility" principle

# Builder

## Creational




A house is a very complex object to build. It involves walls, doors, and a rooftop. So when we built one, we can hire a builder for us, which does the job one by one, piece by piece until we have our beautiful house!



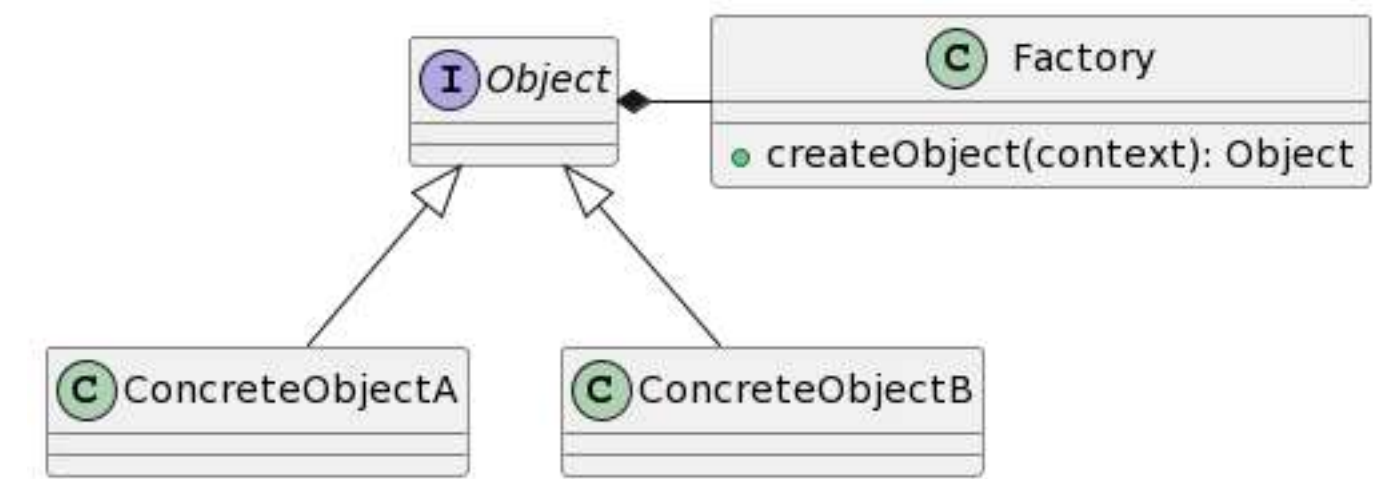
### Take Away

- Perfect to get rid of big constructors, which take a huge amount of parameters
- Encapsulate code for construction and representation
- The builder needs “access” to the internal representation and creation
- StringBuilder is a famous example for the pattern

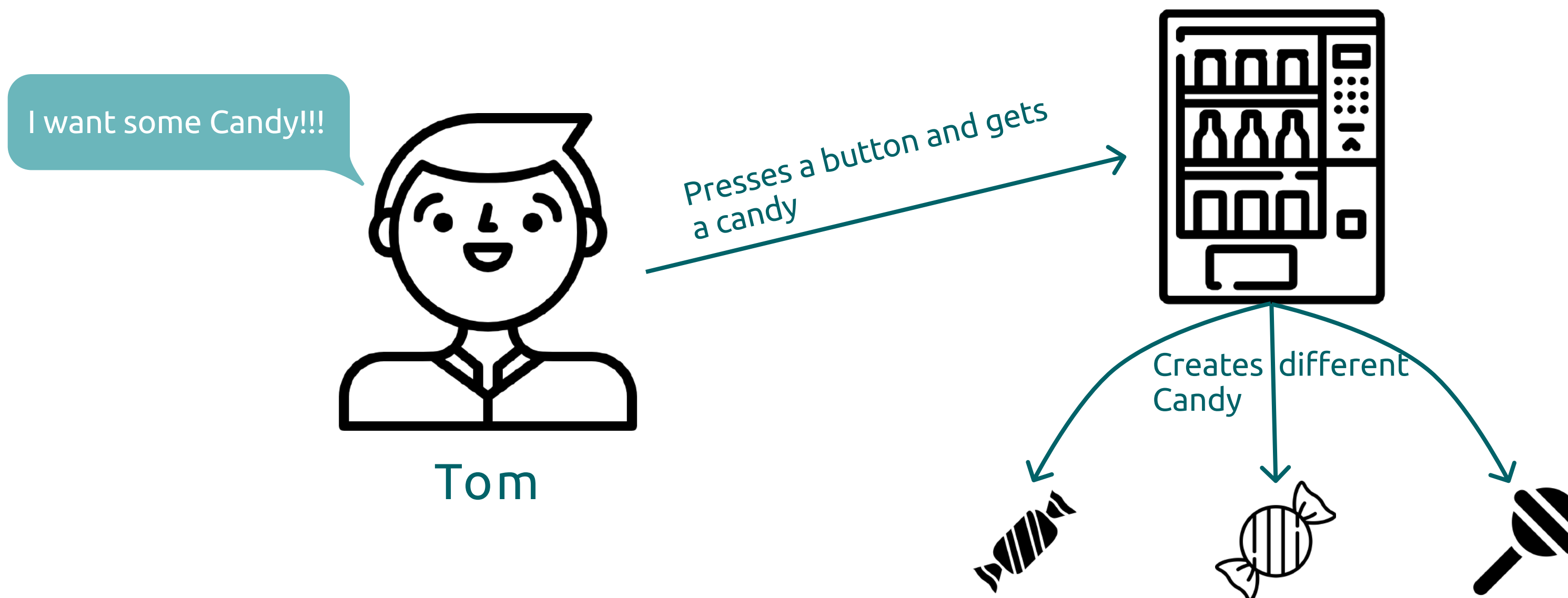
 [More info](#)

# Factory method

## Creational



Tom wants to get some candy. So he goes to the vending machine, which has candy for him. With a press of a button he can get different kinds of candy, which the machine creates for him.

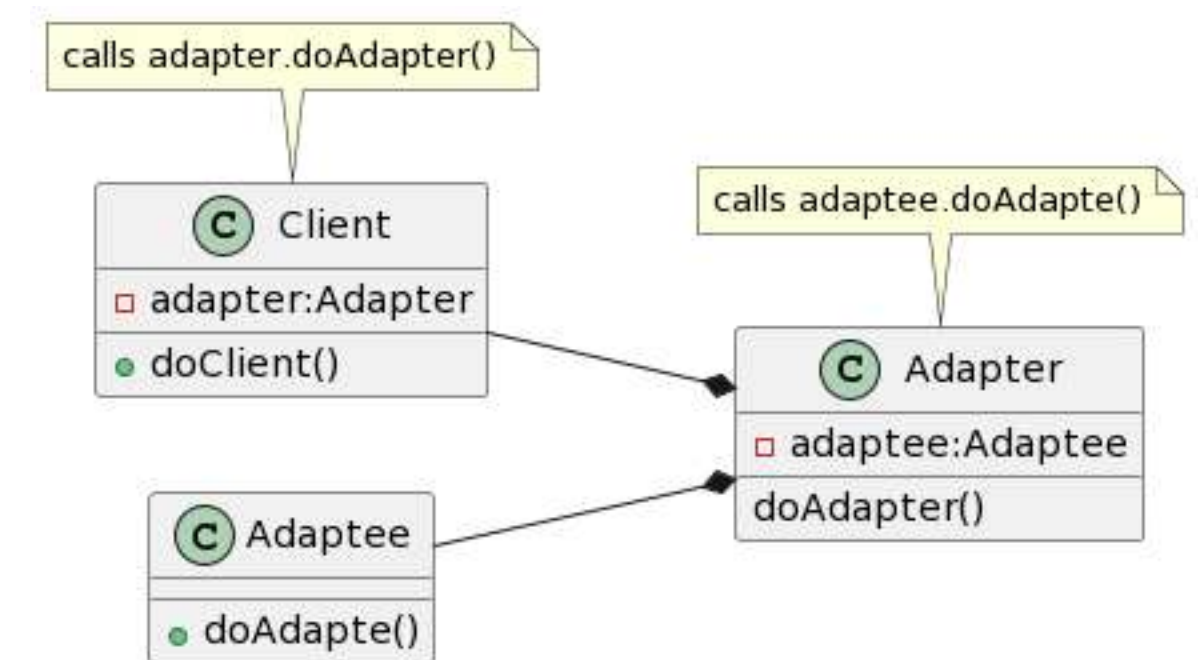


### Take Away

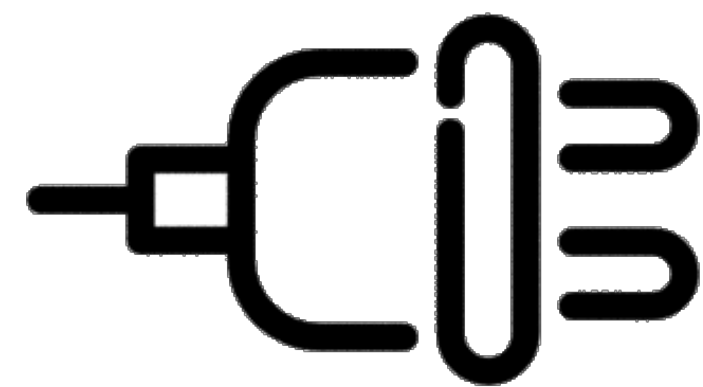
- Avoid tight coupling between creator and the concrete implementations
- Can be extended to Abstract Factory pattern
- Factory only exposes a common shared type - perfect for the Open-Close Principle

# Adapter

## Structural



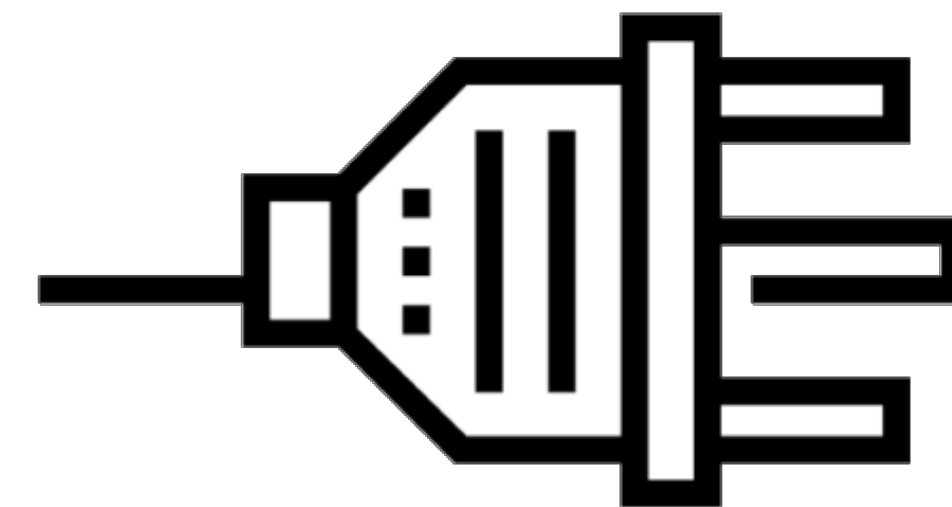
An EU power plug does not fit into a US wall socket, they are incompatible. With the EU to US power **adapter**, we operate seemingly incompatible interfaces to operate together. It is a translator between two objects.



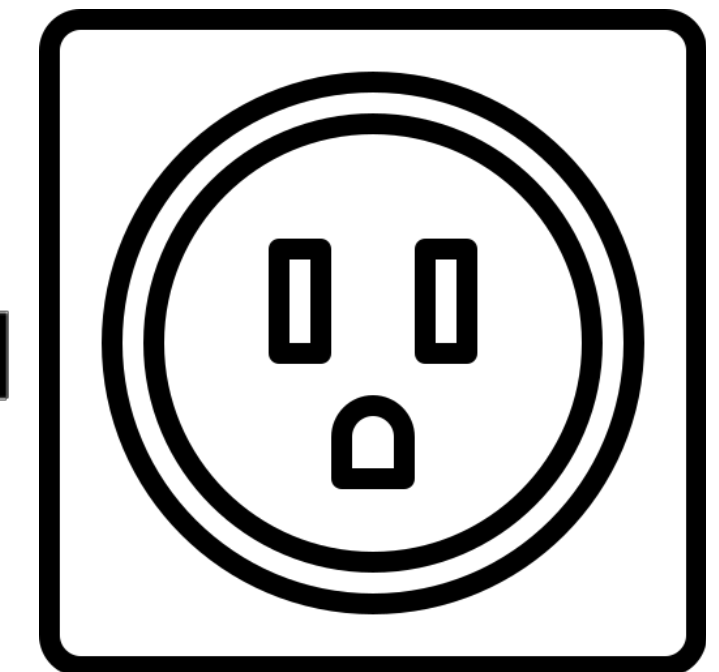
EU Power Plug



EU to US  
**Adapter**



US Power Plug



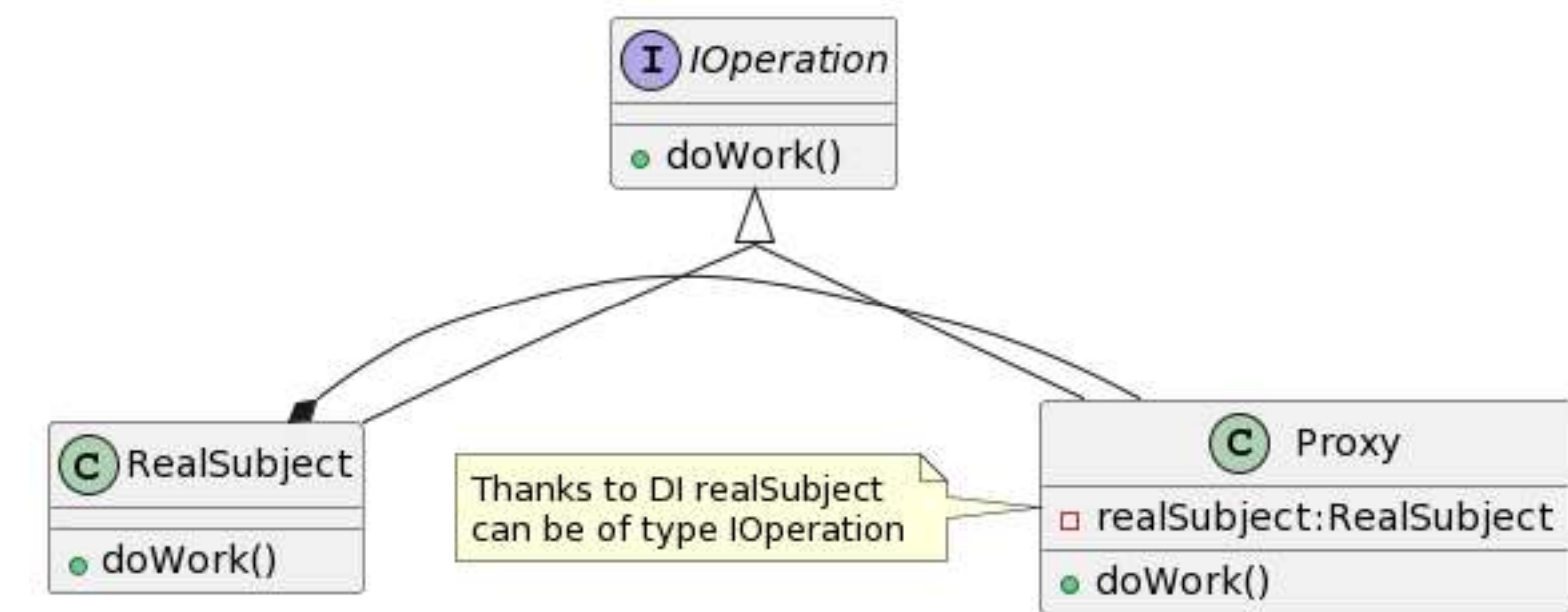
US Wall Socket

## Take Away

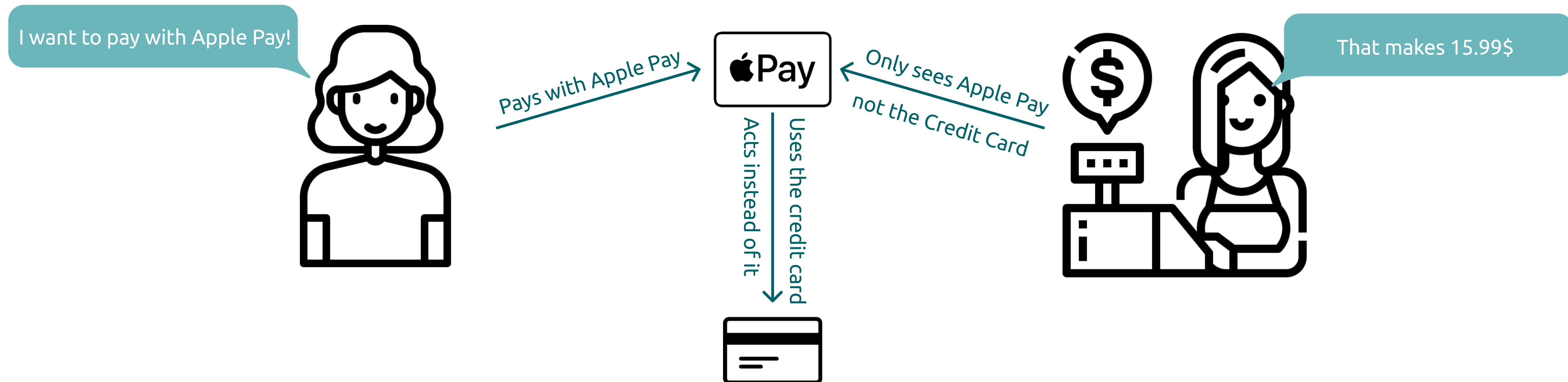
- Holds up Single-Responsibility and the Open-Close principle as we decouple the logic how different objects have to communicate with each other.
- Every Mapper in your code is an implementation of the **adapter** pattern.



# Proxy Structural



When we pay with Apple Pay and similar services the merchant does not see the credit card which is linked to Apple Pay. Apple Pay acts as a **proxy** for the “real” credit card.



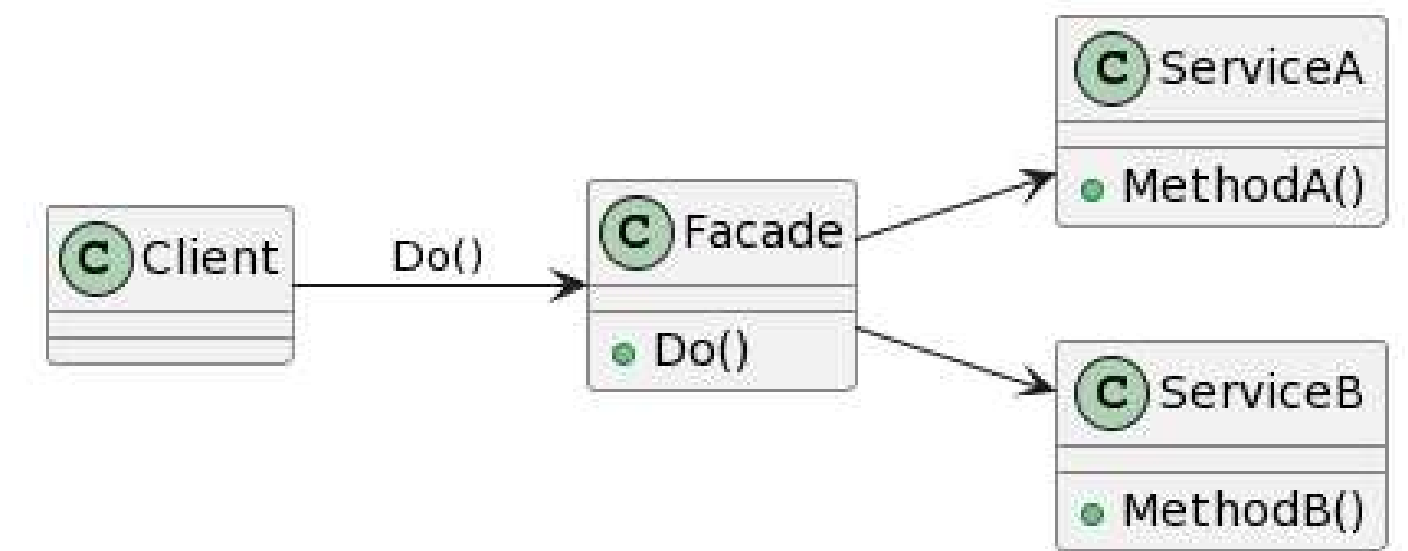
## Take Away

- Extending a class without subclassing it (Open-Close Principle)
- Is transparent from the outside world. Your user doesn't know it is a proxy.
- Entity Framework uses them to lazy load entities. (It sits on top of your getter's.)

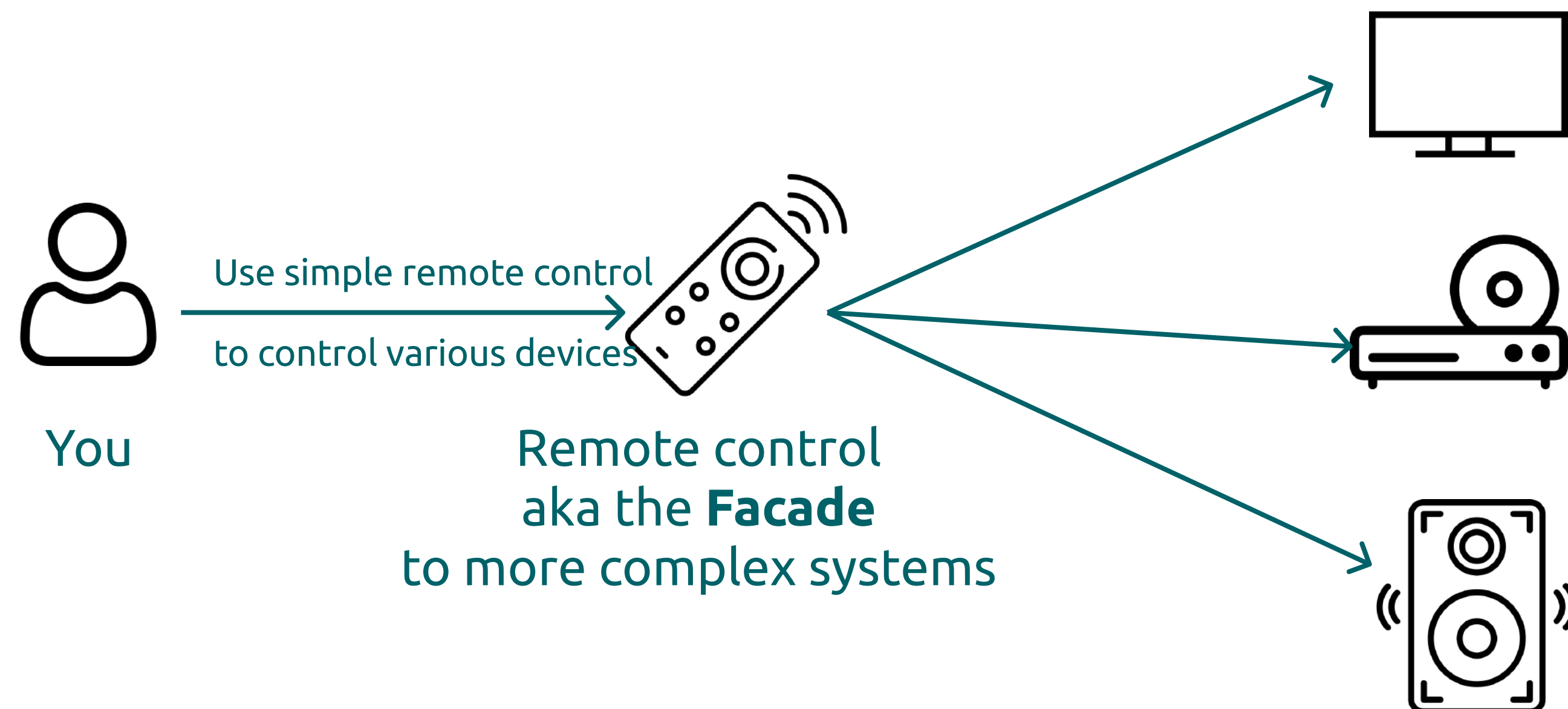


# Facade

## Structural



Imagine a home theater system. The remote control is the facade, while the various devices it controls (e.g. TV, Blu-ray player, sound system) are the subsystems. The remote control provides a simplified interface for controlling the various devices,



### Take Away

- Simplifying the interface of a complex system, making it easier to use
- Hiding the complexity of the subsystems from the client
- Allowing for easier changes to the subsystems without affecting the clients

# Thanks

A big shout out to [Mahdiye Ijavi](#) for the design template.

Also thanks to [flaticon.com](#) as I am using their icons all over the place.

A big shout-out goes to you dear reader. If you have any input or requests let me know. Down there you will find some links to how you can reach out to me. I will be more than happy to get some feedback and new ideas to continue the journey!

As you might know, creating those free eBooks takes time and resources. If you want to support me in any way you also find opportunities on my GitHub account or on my webpage.



# Version

## 1.2 - 2023-01-19

- New Patterns: Facade

## 1.1 - 2022-11-28

- Added Iterator pattern

## 1.01 - 2022-11-25

- Fixed some grammar and spelling errors
- Made Chain example clearer

## 1.0 - 2022-11-16

- Initial Release