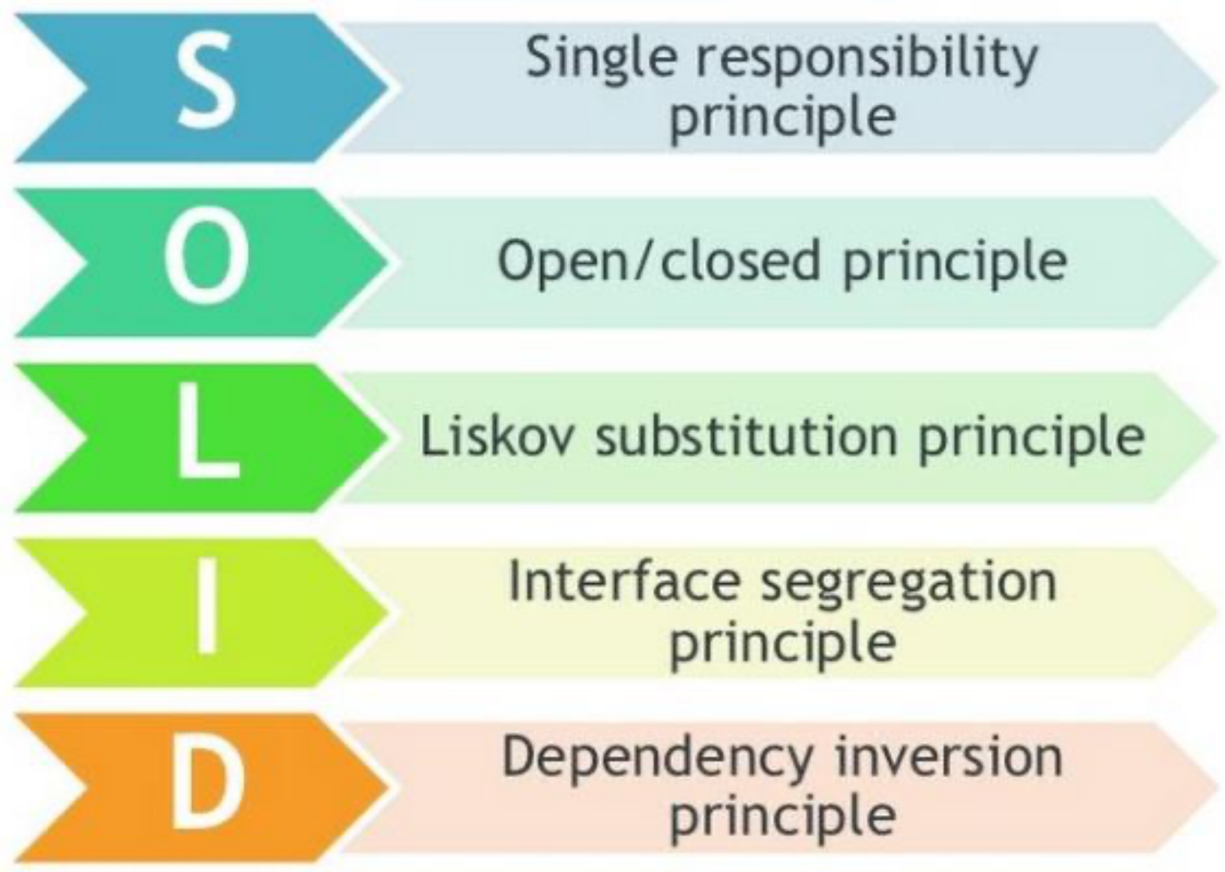
ECE444: Software Engineering

Design Patterns 3

Shurui Zhou

OO Design Principles



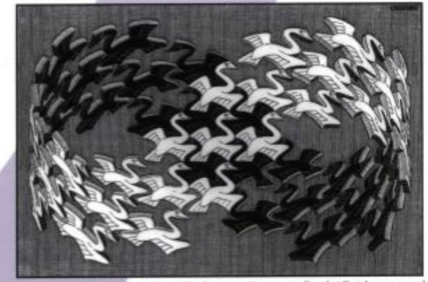
Building stable and flexible systems

Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

Copyrighted Material

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- the GoF book
- Elements of Reusable Object-Oriented Software
- 23 OO patterns

Design Patterns

- Design Patterns – expert solutions to recurring problems in a certain domain
- Description usually involves problem definition, driving forces, solution, benefits, difficulties, related patterns.
- Pattern Language - a collection of patterns, guiding the users through the decision process in building a system
- Patterns are related (high level-low level)

Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Singleton

- The **Singleton** class declares the static method getInstance that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Client → Singleton

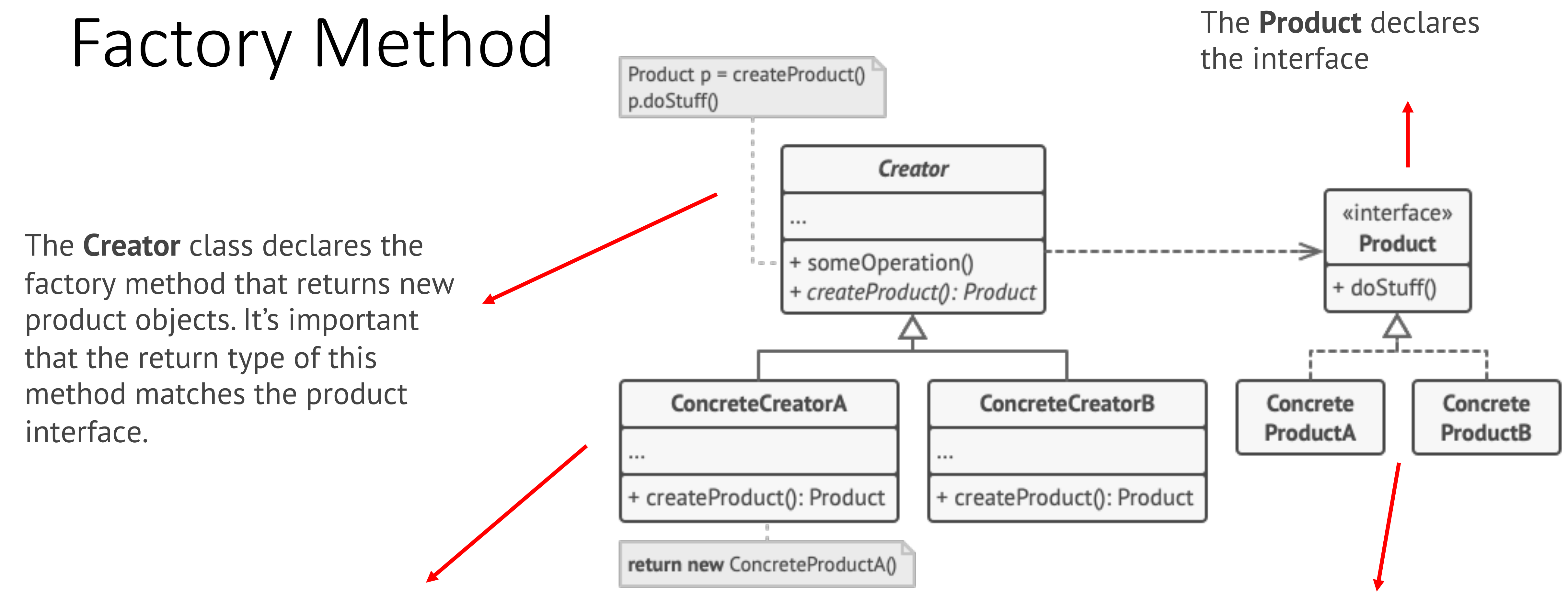
Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Factory Method

The **Product** declares the interface

The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.



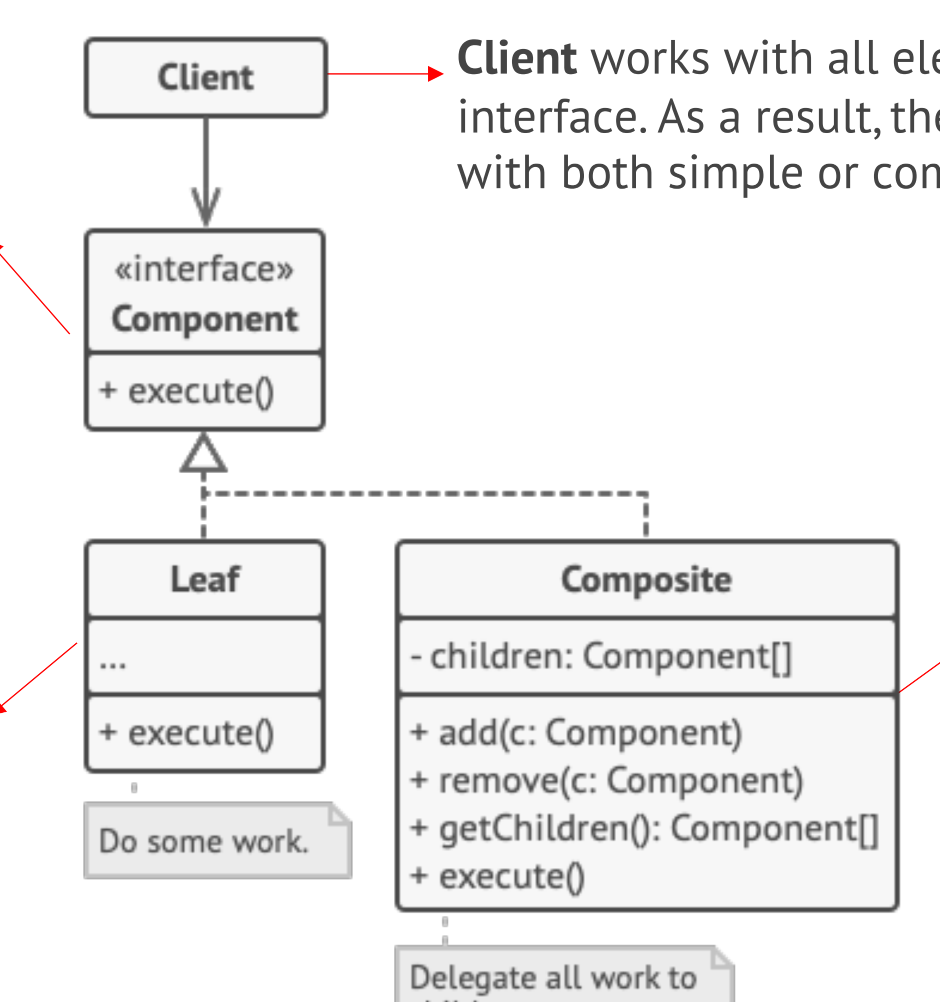
Concrete Creators override the base factory method so it returns a different type of product. Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Concrete Products are different implementations of the product interface.

Composite Design Pattern - Structure

The **Component** interface describes operations that are common to both simple and complex elements of the tree.

Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.



The **Leaf** is a basic element of a tree that doesn't have sub-elements.

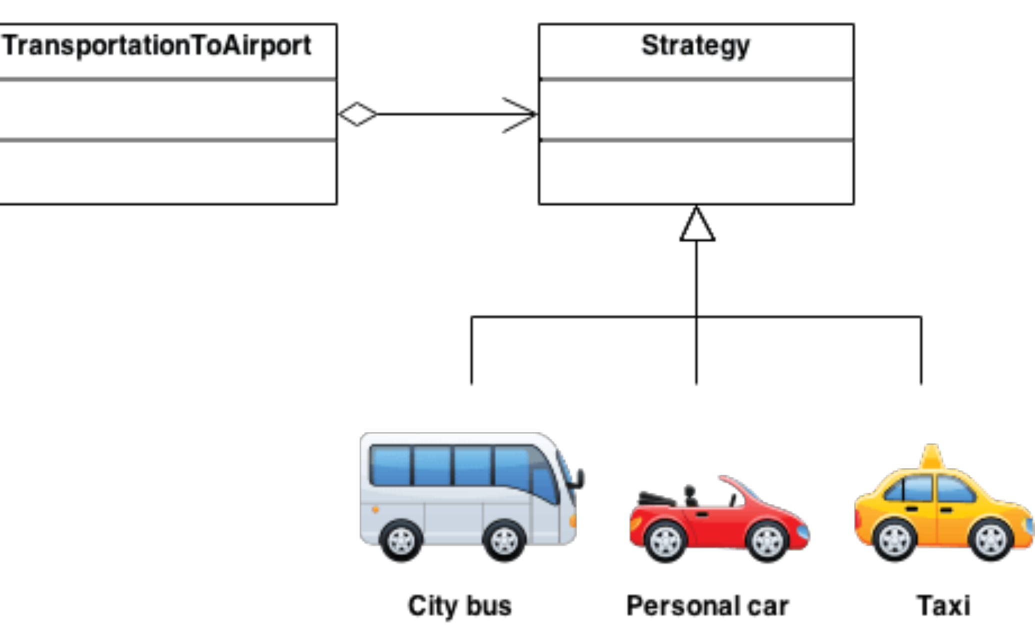
The **Composite/container** is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Strategy

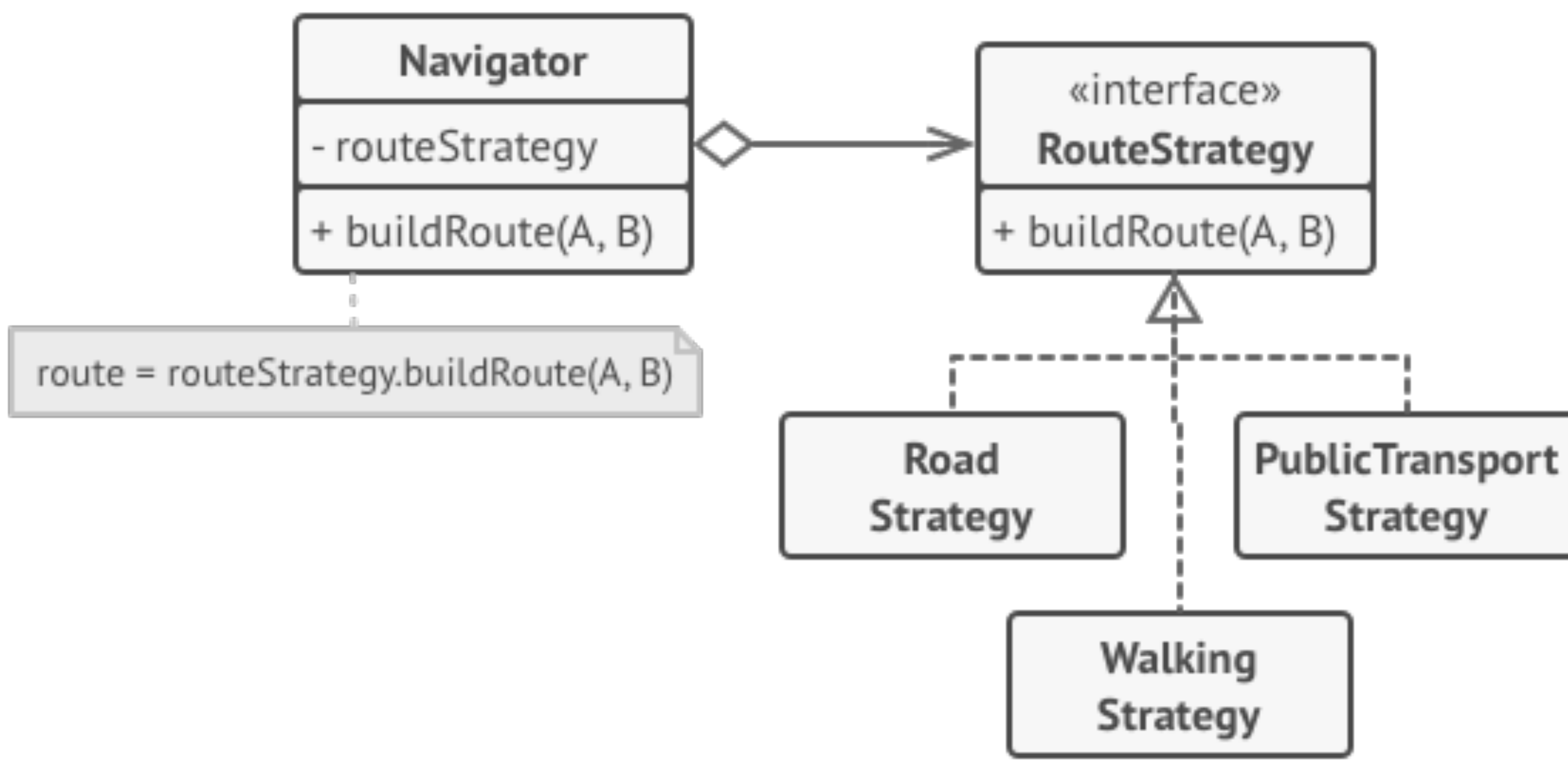
- **Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



Strategy

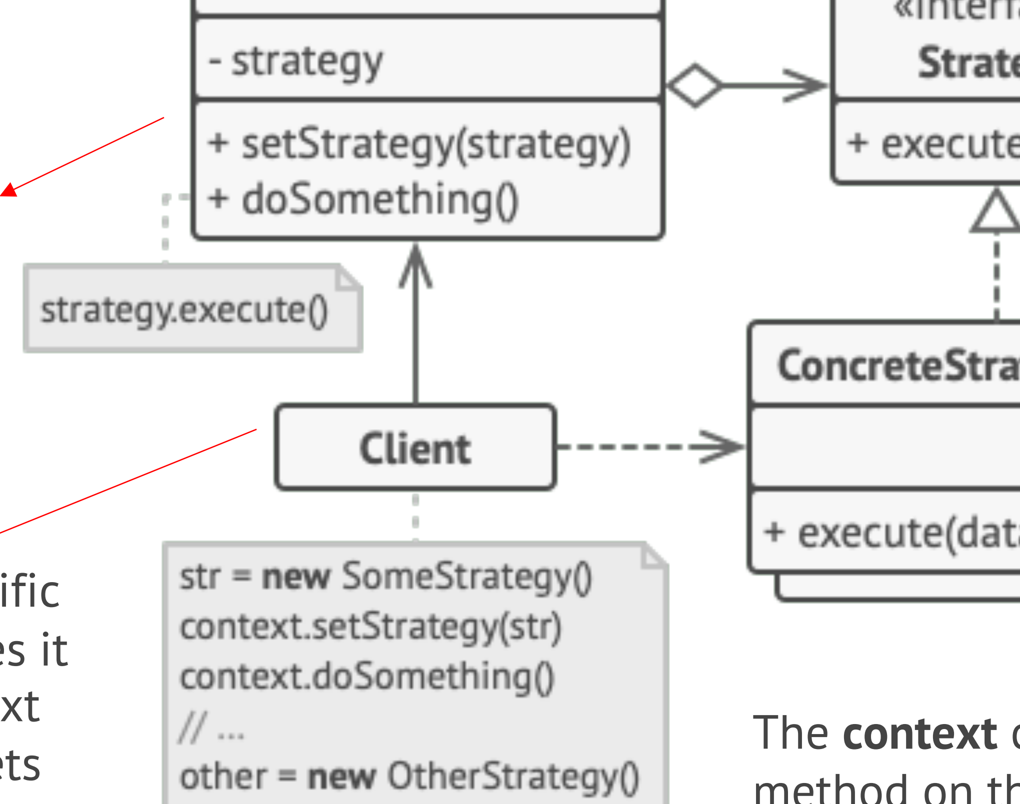
- The strategy pattern allows grouping related algorithms under an abstraction, which allows switching out one algorithm or policy for another without modifying the client.
- Instead of directly implementing a single algorithm, the code receives runtime instructions specifying which of the group of algorithms to run.

Strategy



Strategy

The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.



The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

Concrete Strategies implement different variations of an algorithm the context uses.

The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

The **context** calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

Strategy - Applicability

- **When you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.**
- **When you have a lot of similar classes that only differ in the way they execute some behavior.**
- **To isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.**
- **When your class has a massive conditional operator that switches between different variants of the same algorithm.**

Strategy – Pros & Cons

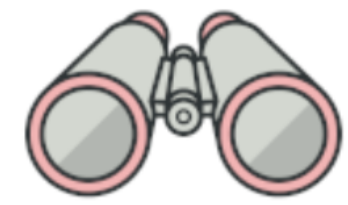
- ✓ You can swap algorithms used inside an object at runtime.
- ✓ You can isolate the implementation details of an algorithm from the code that uses it.
- ✓ You can replace inheritance with composition.
- ✓ *Open/Closed Principle*. You can introduce new strategies without having to change the context.

- ✗ If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- ✗ Clients must be aware of the differences between strategies to be able to select a proper one.
- ✗ A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Observer Pattern



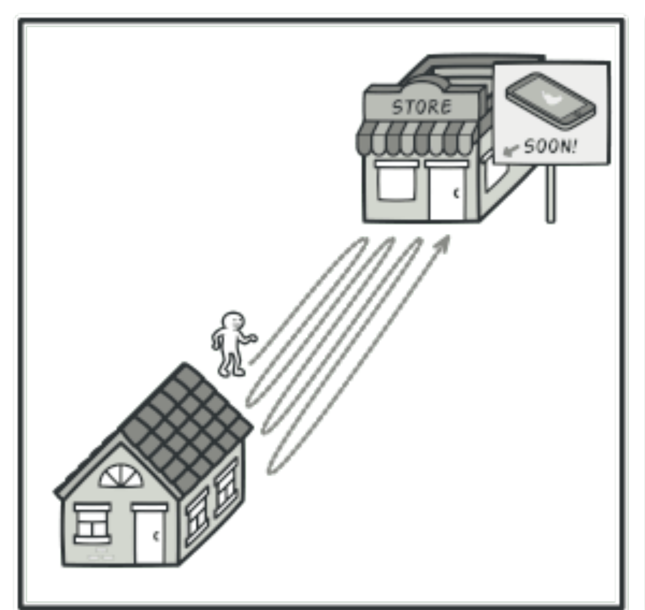
Observer Pattern

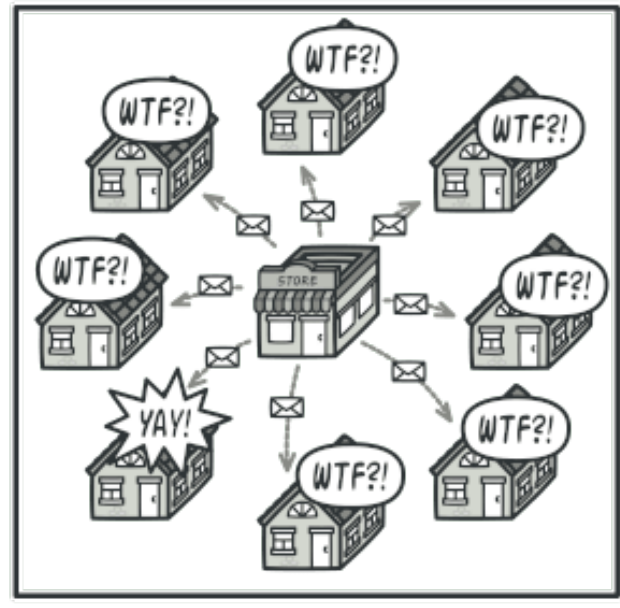
- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- Publishers + Subscribers = Observer Pattern

How newspaper or magazine subscriptions work?

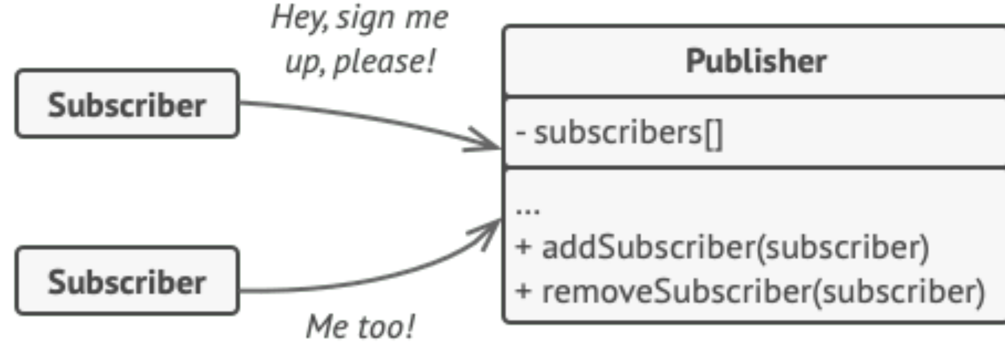
1. A newspaper publisher goes into business and begins publishing newspapers.
2. You subscribe to a particular publisher, and every time there’s a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
3. You unsubscribe when you don’t want papers anymore, and they stop being delivered.
4. While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

Observer Pattern





Visiting the store vs. sending spam

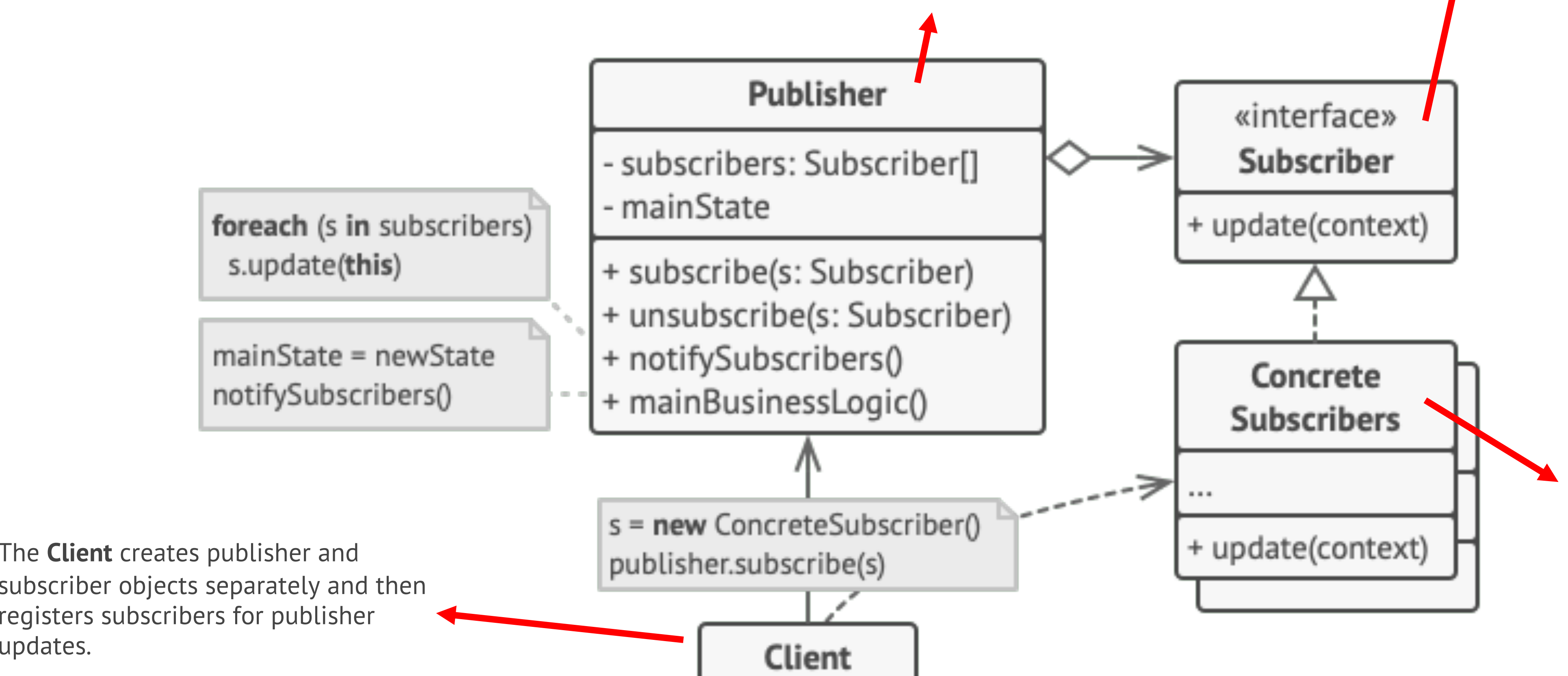


A subscription mechanism lets individual objects subscribe to event notifications.

Observer Pattern

The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.



Concrete Subscribers perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Observer - Applicability

- When changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

- When some objects in your app must observe others, but only for a limited time or in specific cases.

Real world Application

- **Splitwise group** : Anyone adds or updates any entry in the group - all members of group get a notification
- **Following a post/event**: If one follows a post , (s)he gets added to the observers & any further comments on the same post , send a notification to all the other observers
- **Software Repository**: Under the push notification model , devices are observable for the central software repository & as soon as there is new software from one of the observers , all the devices registered will be sent a push notification to check for that software
- **Weather update**
- **Stock prices update**
- **Train ticket confirmation**

Observer - Pros and Cons

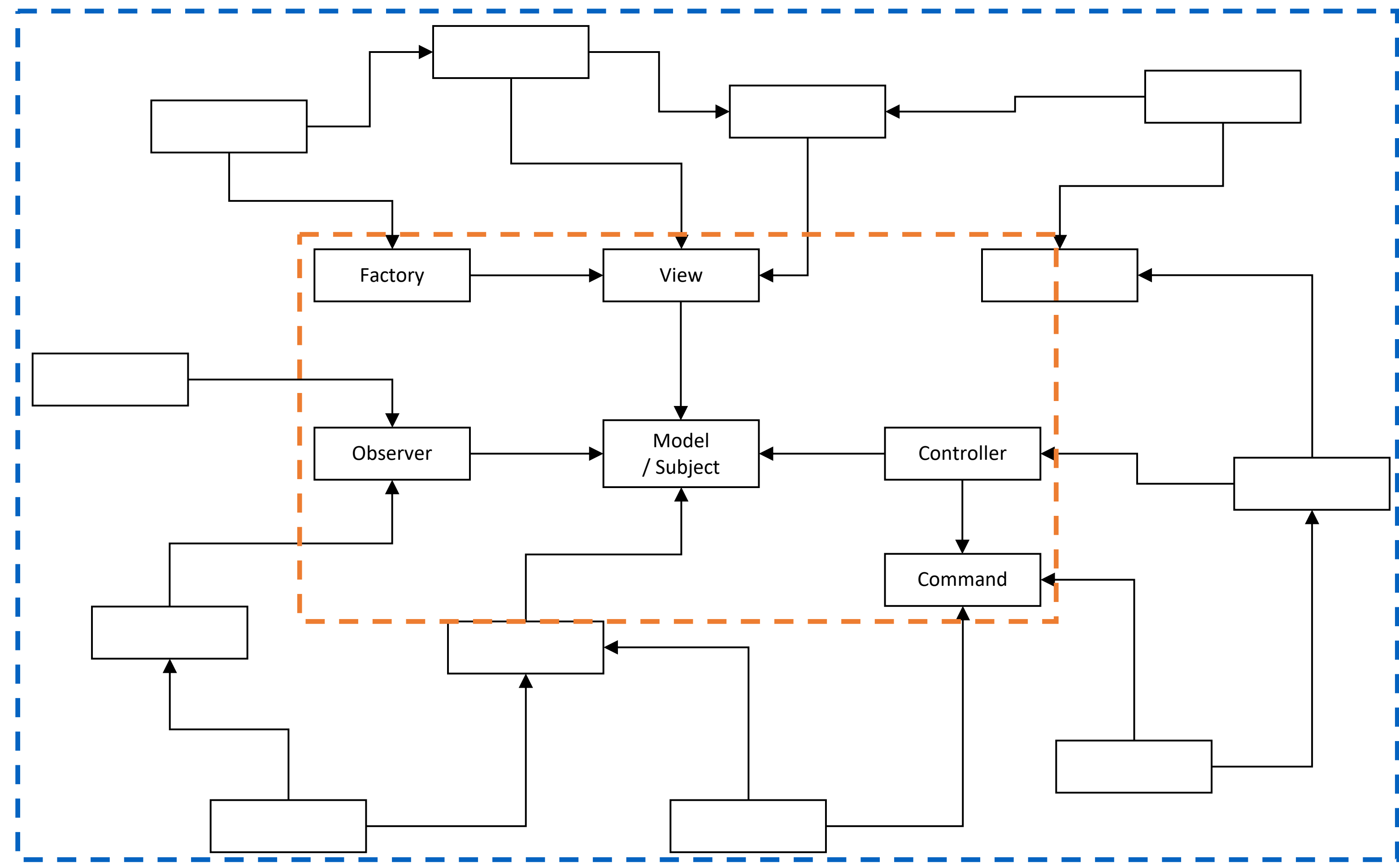
- ✓ *Open/Closed Principle*. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- ✓ You can establish relations between objects at runtime.

- ✗ Subscribers are notified in random order.

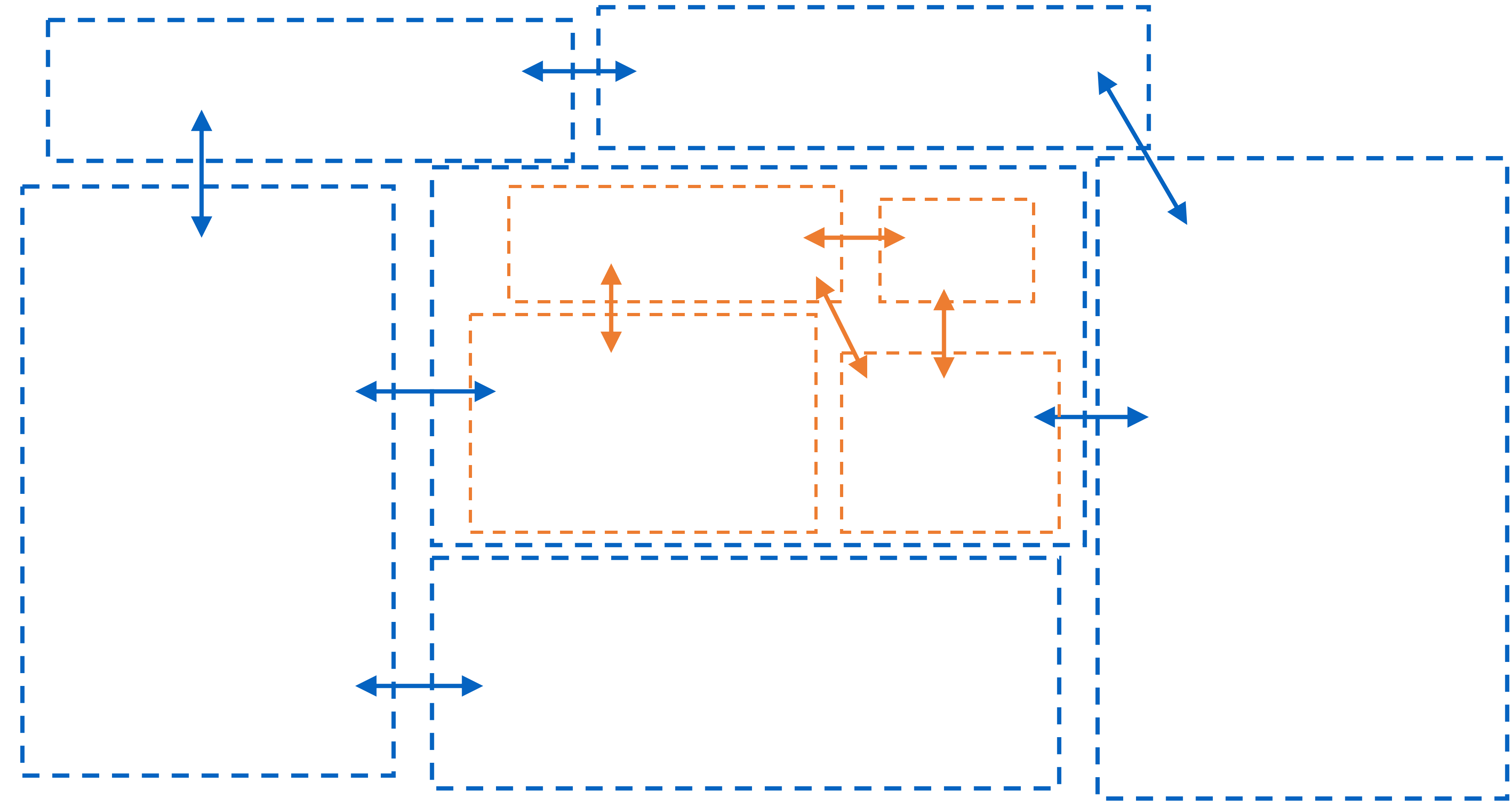
Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

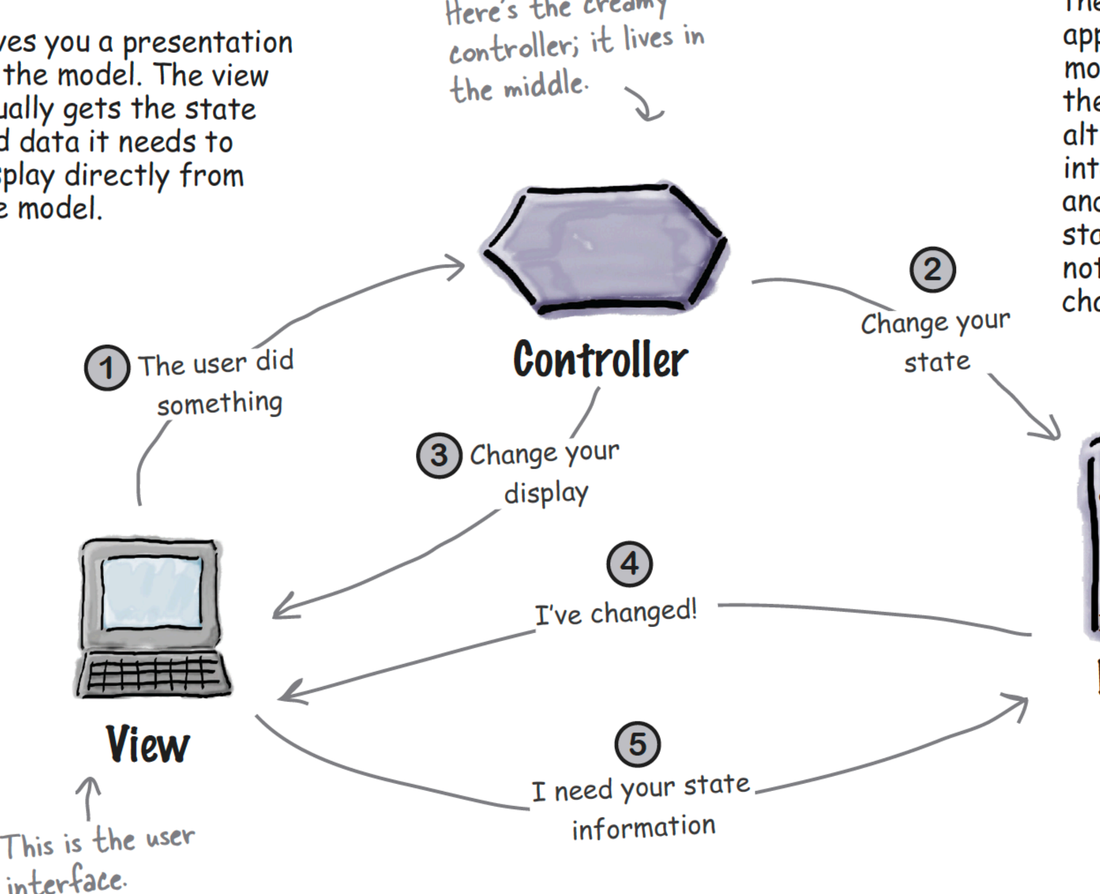
Design Patterns

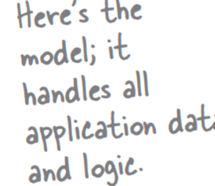


Architecture



MVC Architecture

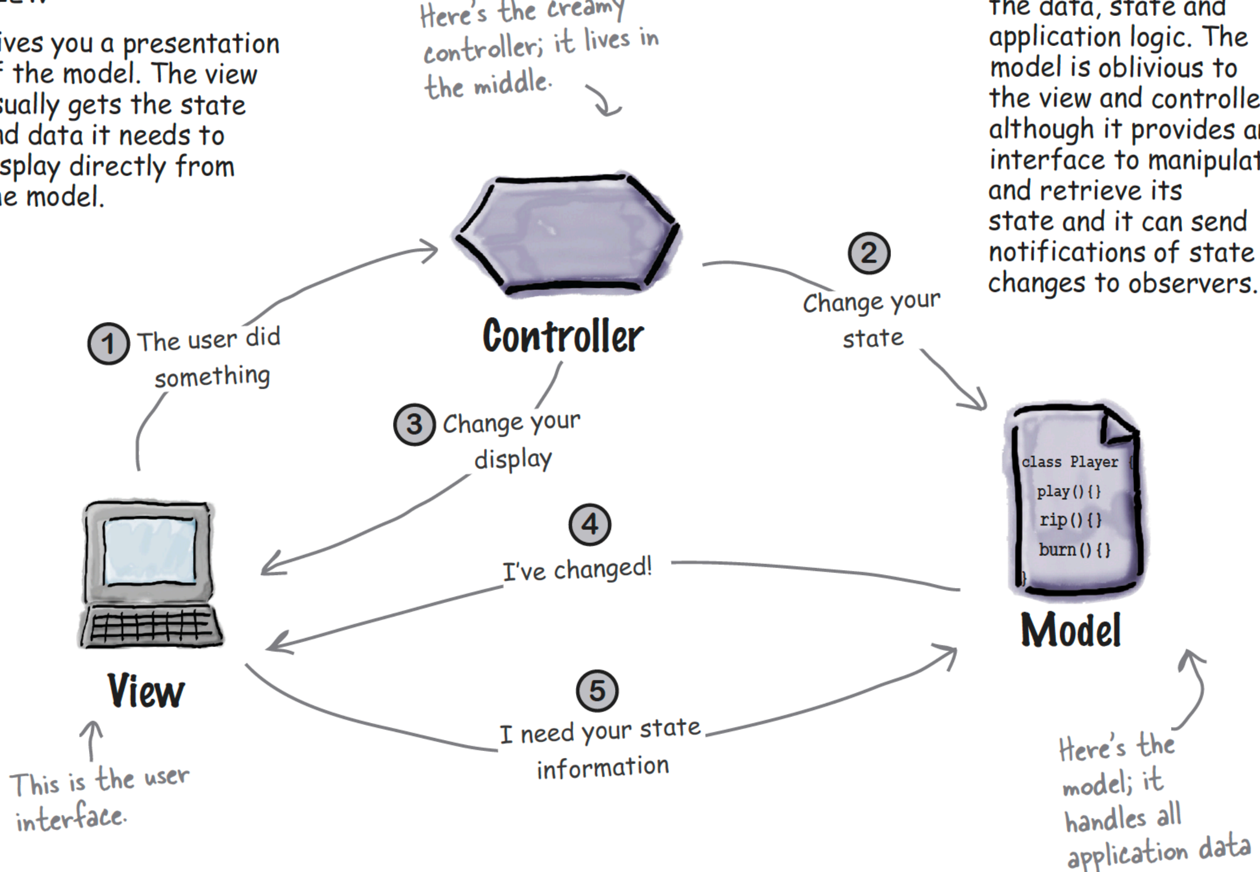




MVC Architecture

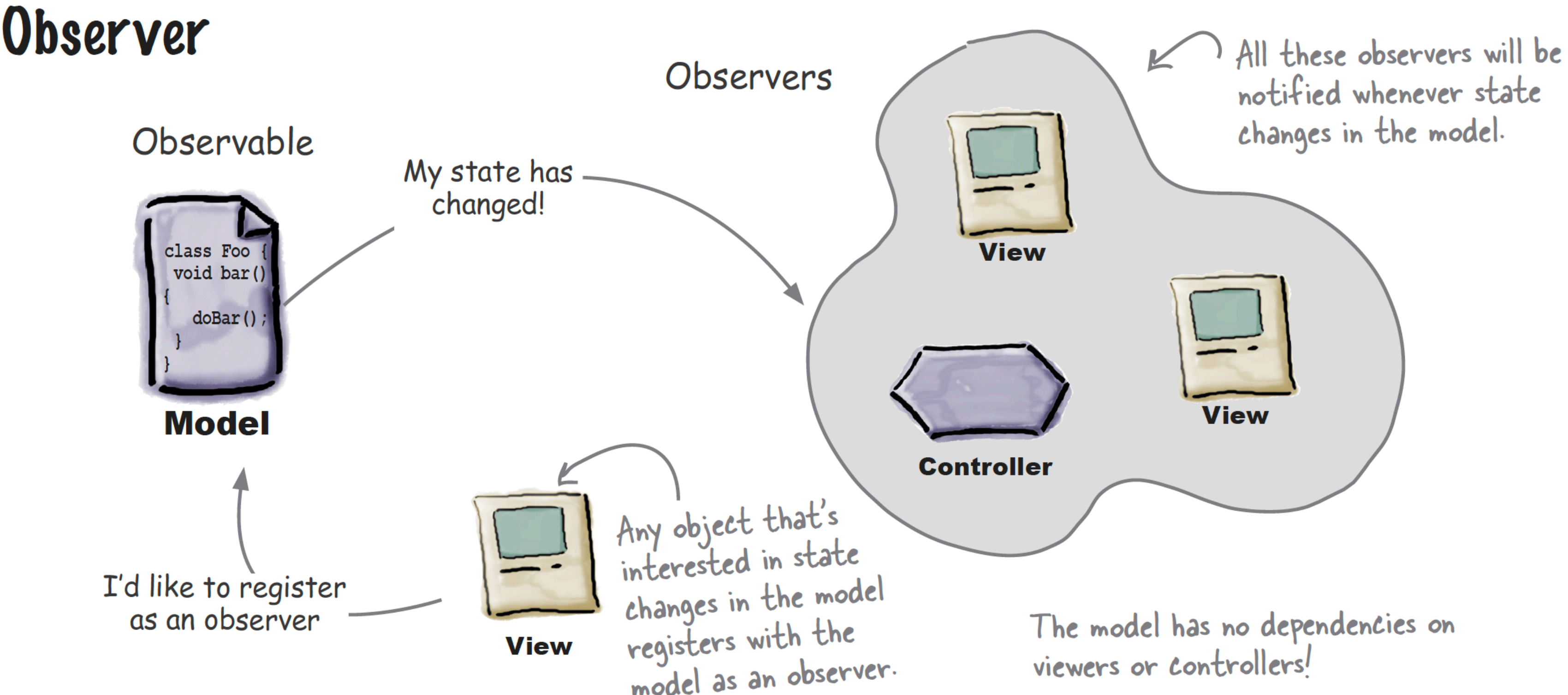
- Model – Observer Pattern
- View – Composite + Strategy
- Controller -- Strategy Pattern

Takes user input and figures out what it means to the model.



MVC Architecture

- Model – Observer Pattern
- View – Composite + Strategy
- Controller -- Strategy Pattern



MVC Architecture

- Model – Observer Pattern
- View – Composite + Strategy
- Controller -- Strategy Pattern

Strategy

The view delegates to the controller to handle the user actions.

View

The user did something

Controller

The controller is the strategy for the view – it's the object that knows how to handle the user actions.

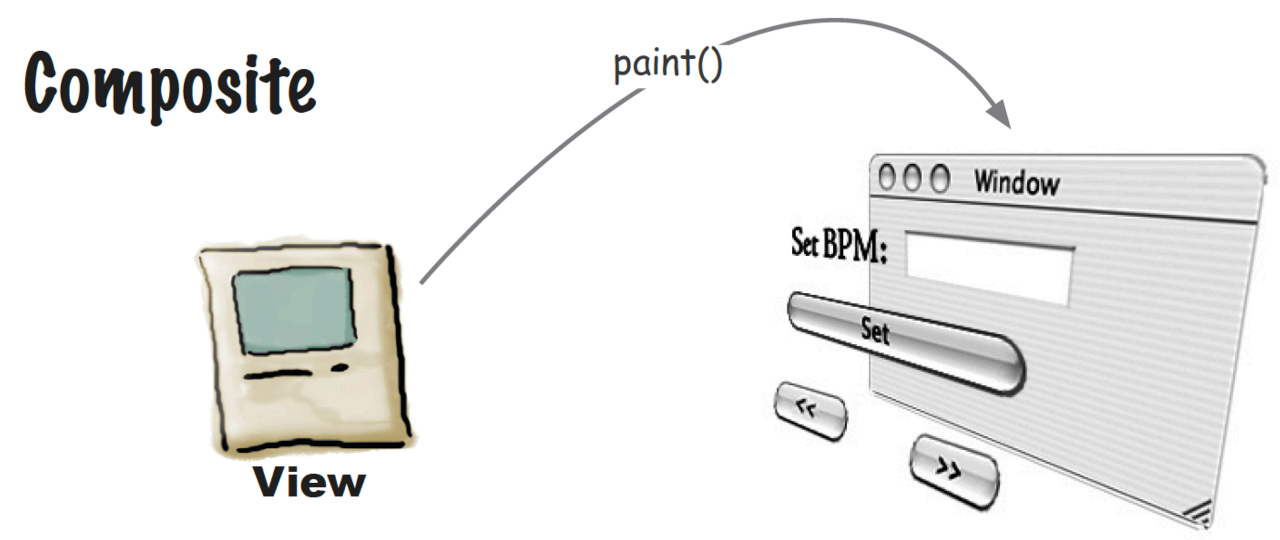
Controller

We can swap in another behavior for the view by changing the controller.

The view only worries about presentation. The controller worries about translating user input to actions on the model.

MVC Architecture

- Model – Observer Pattern
- View – Composite + Strategy
- Controller -- Strategy Pattern

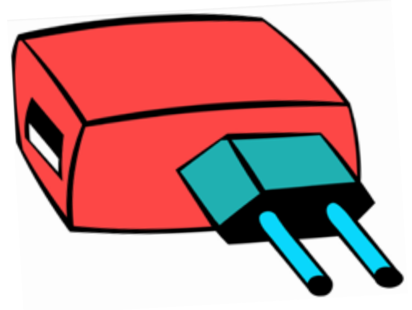


The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components, and so on until you get to the leaf nodes.

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
  - **Adapter**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

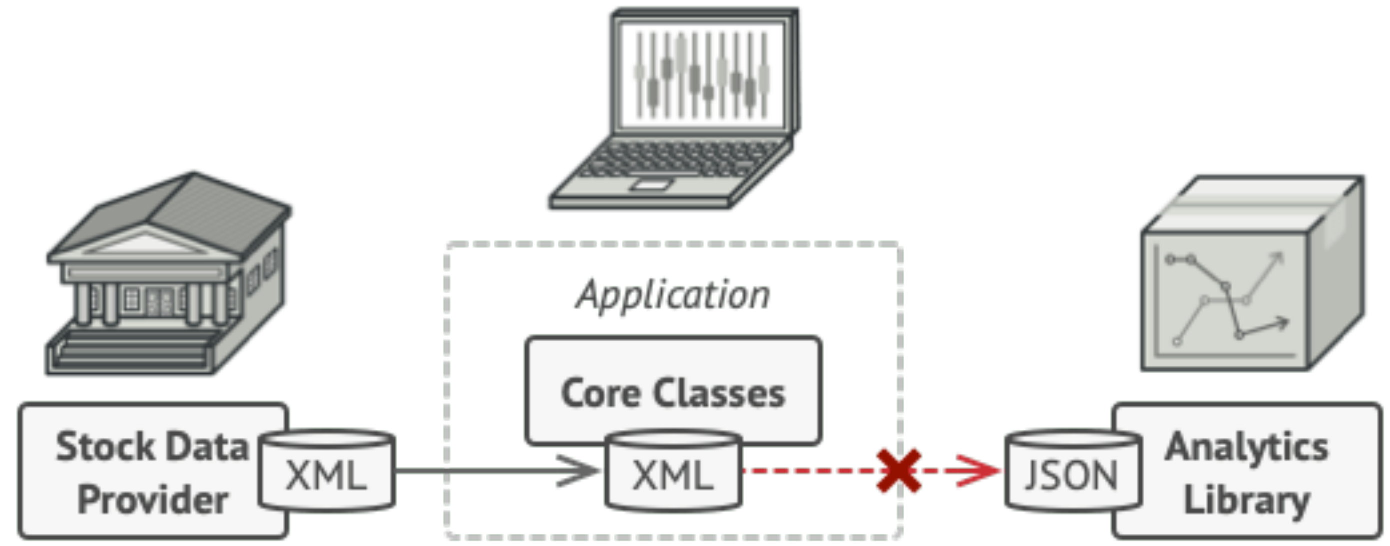
Adapter



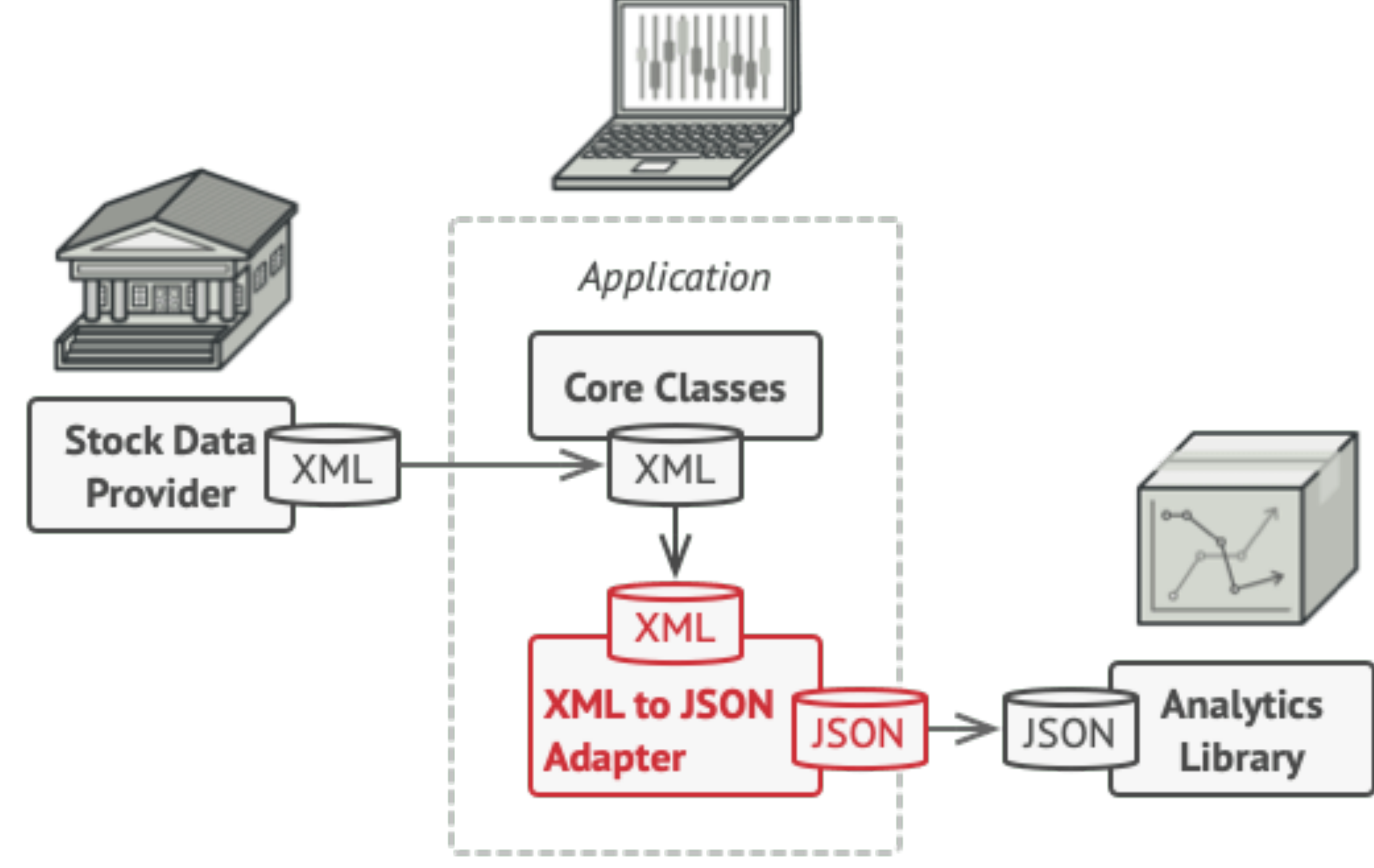
CAR-TO-RAIL ADAPTER

Adapter

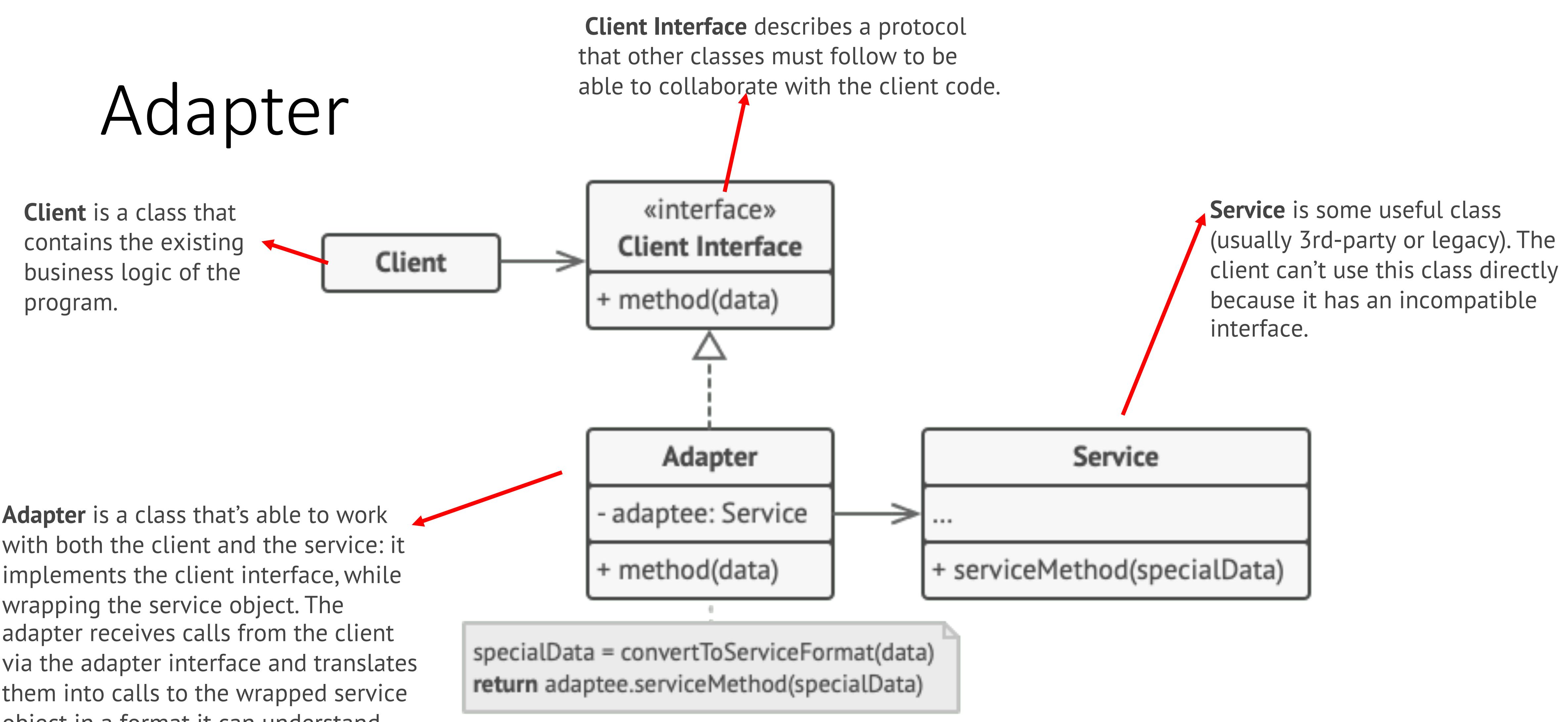
- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



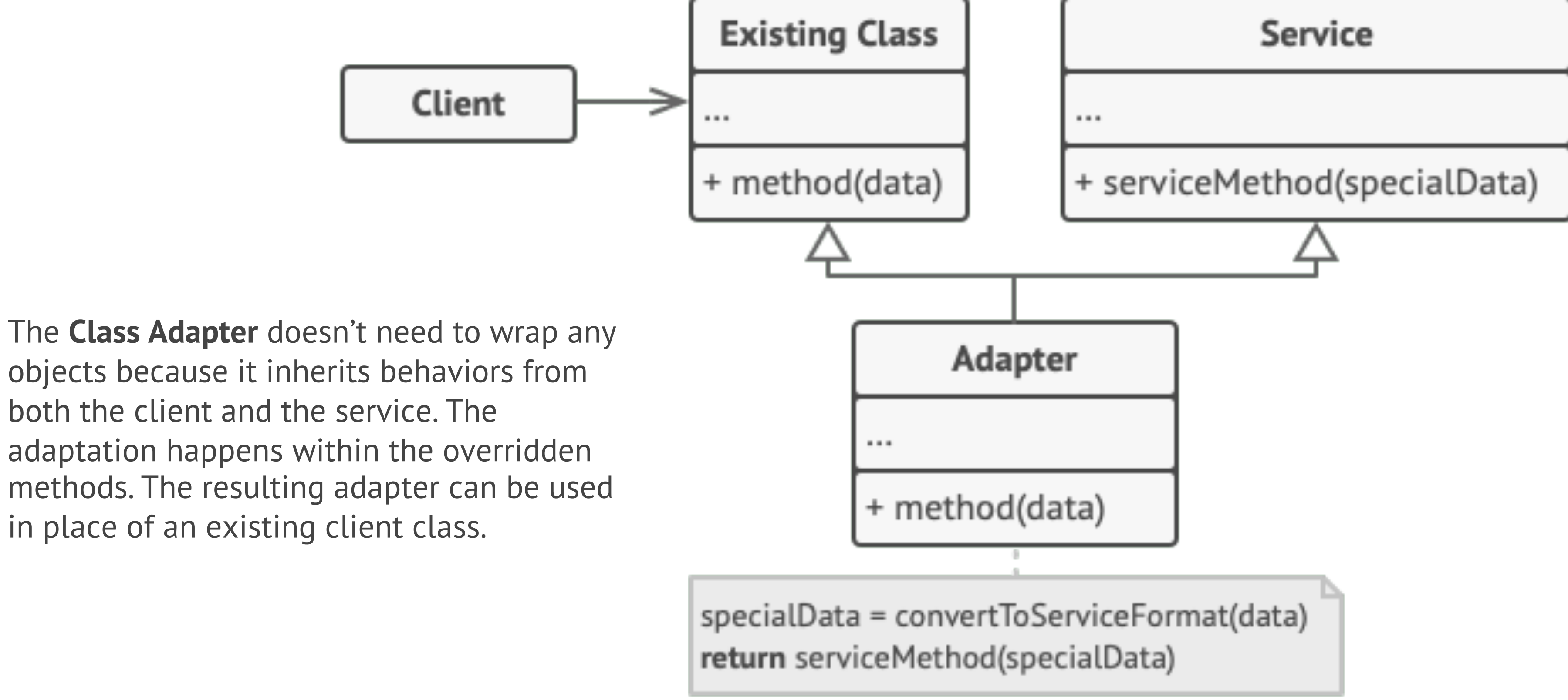
Adapter



Adapter



Adapter



The **Class Adapter** doesn’t need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Adapter - Applicability

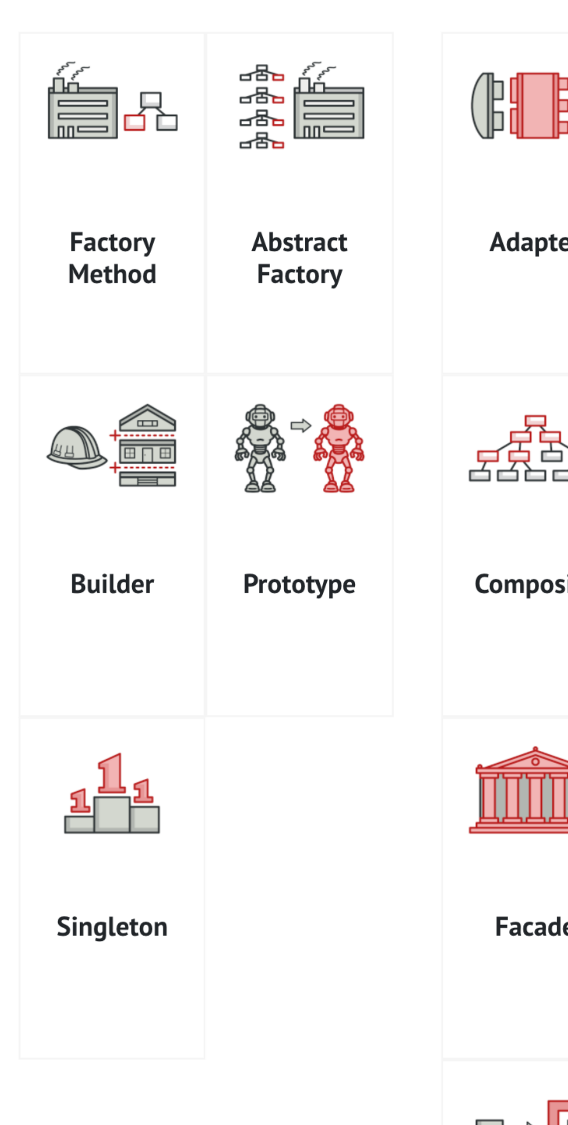
- When you want to use some existing class, but its interface isn’t compatible with the rest of your code.

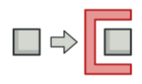
- When you want to reuse several existing subclasses that lack some common functionality that can’t be added to the superclass.

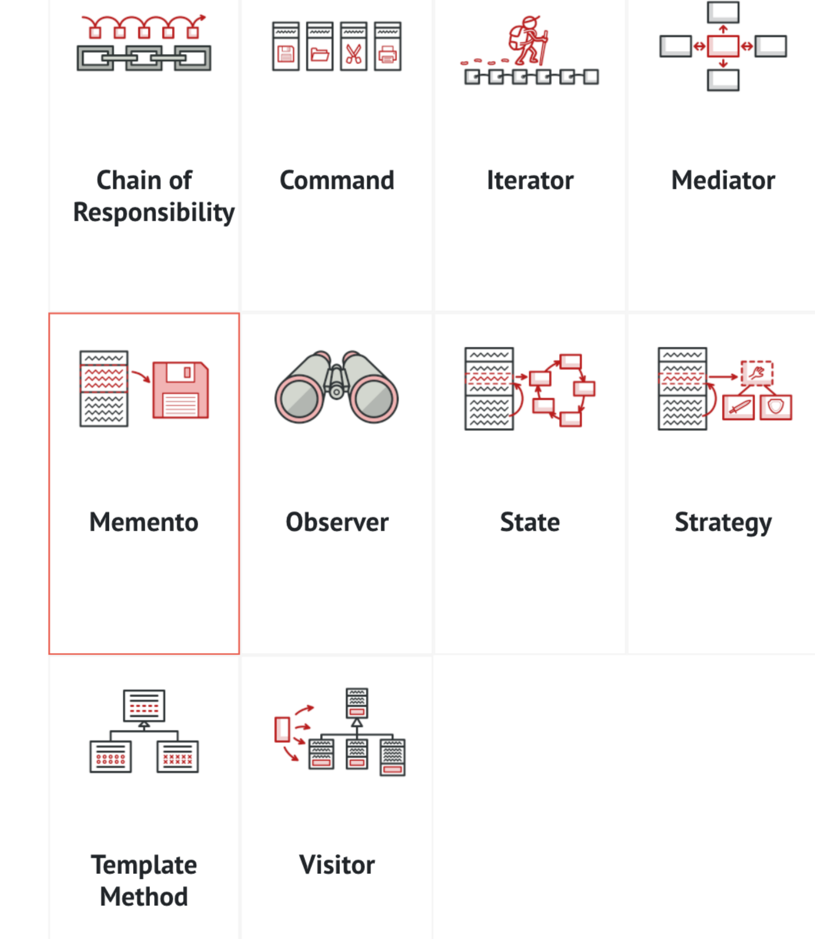
Adapter – Pros and Cons

- ✓ *Single Responsibility Principle.* You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle.* You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

- ✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.







https://refactoring.guru/design-patterns/catalog

Criticism of Design Patterns

- **Kludges for a weak programming language**

Usually the need for patterns arises when people choose a programming language or a technology that lacks the necessary level of abstraction.

- **Inefficient solutions**

Patterns try to systematize approaches that are already widely used.

- **Unjustified use**

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

Cargo cult programming

Are SOLID principles Cargo Cult?

It looks like a plane, but will it fly?

https://blog.ndepend.com/are-solid-principles-cargo-cult/