Session 5 : Design Patterns

sourcemind

What is a Design Pattern?

Design patterns are solutions to general software development problems.

- Well-proved solution for solving the specific problem;
- Reusability and extensibility of the already developed applications;
- Use object-oriented concepts like decomposition, inheritance and polymorphism;
- Provide solutions that help to define the system architecture.

Types of design patterns

- Creational patterns
- Structural patterns
- Behavioral patterns

Types of design patterns

- **Creational design pattern**s provide solutions to instantiate an Object in the best possible way for specific situations.

- **Structural design patterns** provide different ways to create a Class structure (for example, using inheritance and composition to create a large Object from small Objects).

- **Behavioral design patterns** provide a solution for better interaction between objects and how to provide loose-coupling and flexibility to extend easily.

Creational patterns

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

Structural patterns

- Adapter Pattern
- Composite Pattern
- Proxy Pattern
- Flyweight Pattern
- Facade Pattern
- Bridge Pattern
- Decorator Pattern

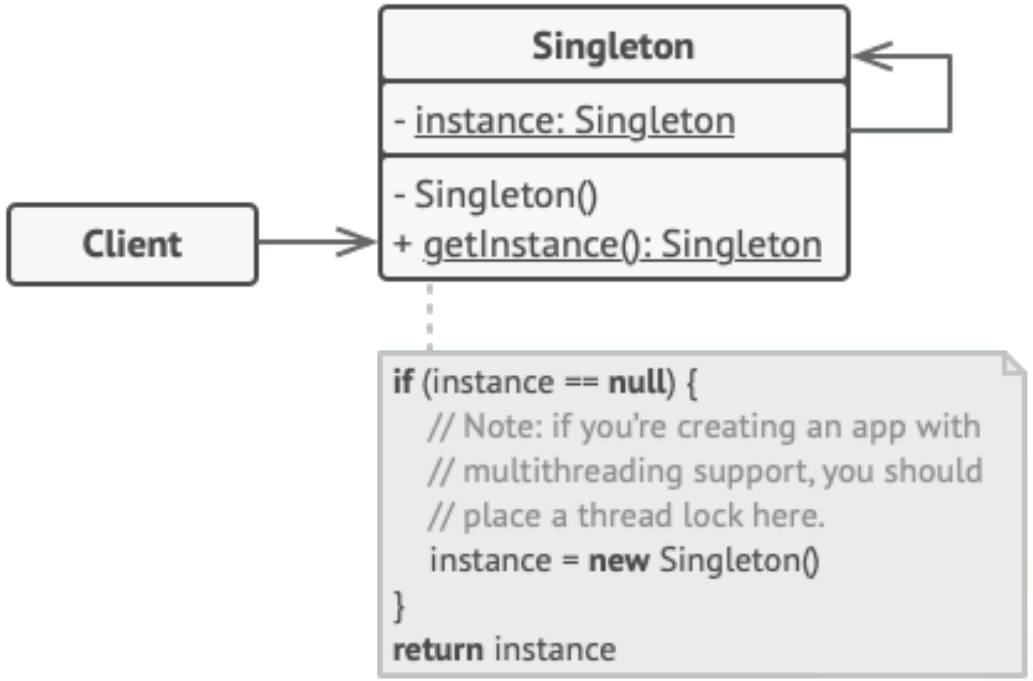
Behavioral patterns

- Observer Pattern
- Mediator Pattern
- Visitor Pattern
- Strategy Pattern
- Iterator Pattern
- Chain of Responsibility Pattern
- Template Method Pattern
- Command Pattern
- State Pattern
- Interpreter Pattern
- Memento Pattern

Creational patterns

Creational patterns – Singleton Pattern

The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created.



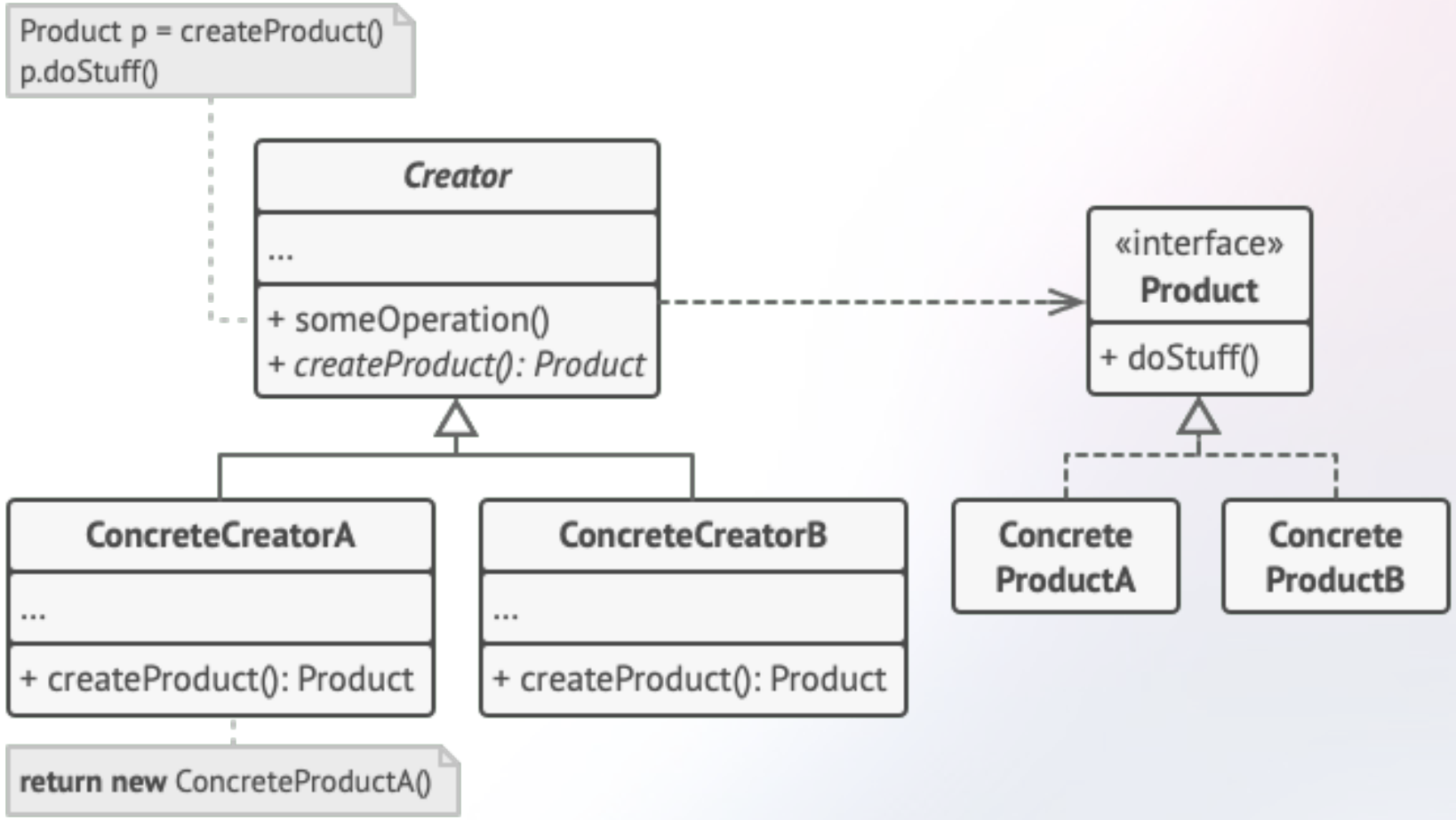

```
public class Singleton {

    private static Singleton instance;

    private Singleton(){}

    public static Singleton getInstance(){
        if(instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

sourcemind

Creational patterns – Factory Pattern

The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

Creational patterns – Factory Pattern



Exercise

Create a Payment Factory

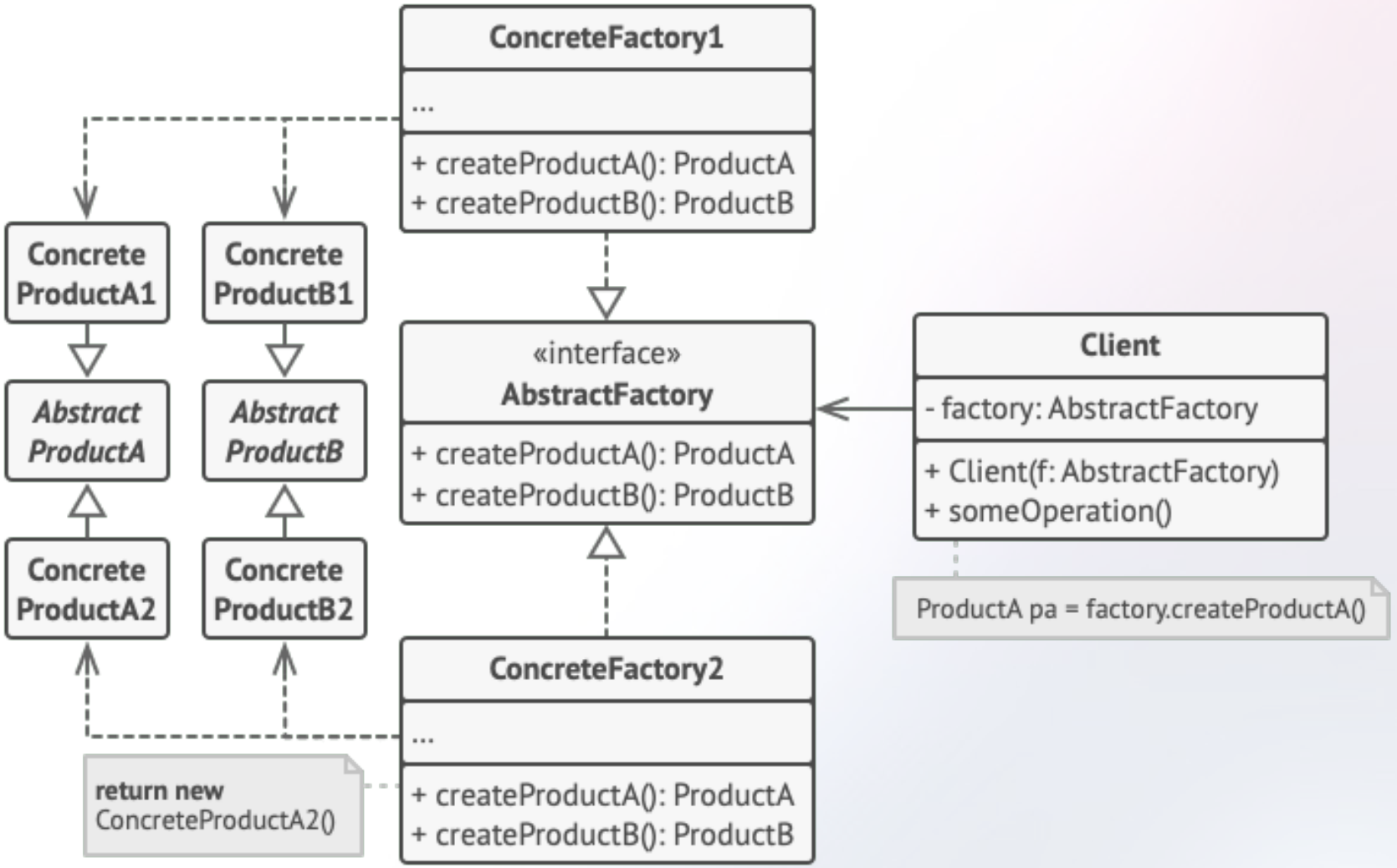
The factory creates and returns a Payment instance.

```
PaymentFactory factory = new CommandLinePaymentFactory();
Payment payment = factory.create(); // should use a Scanner internally

// Payment object is ready
if (payment.verify()) { ... }
else { ... }
```

Creational patterns – Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Creational patterns – Abstract Factory

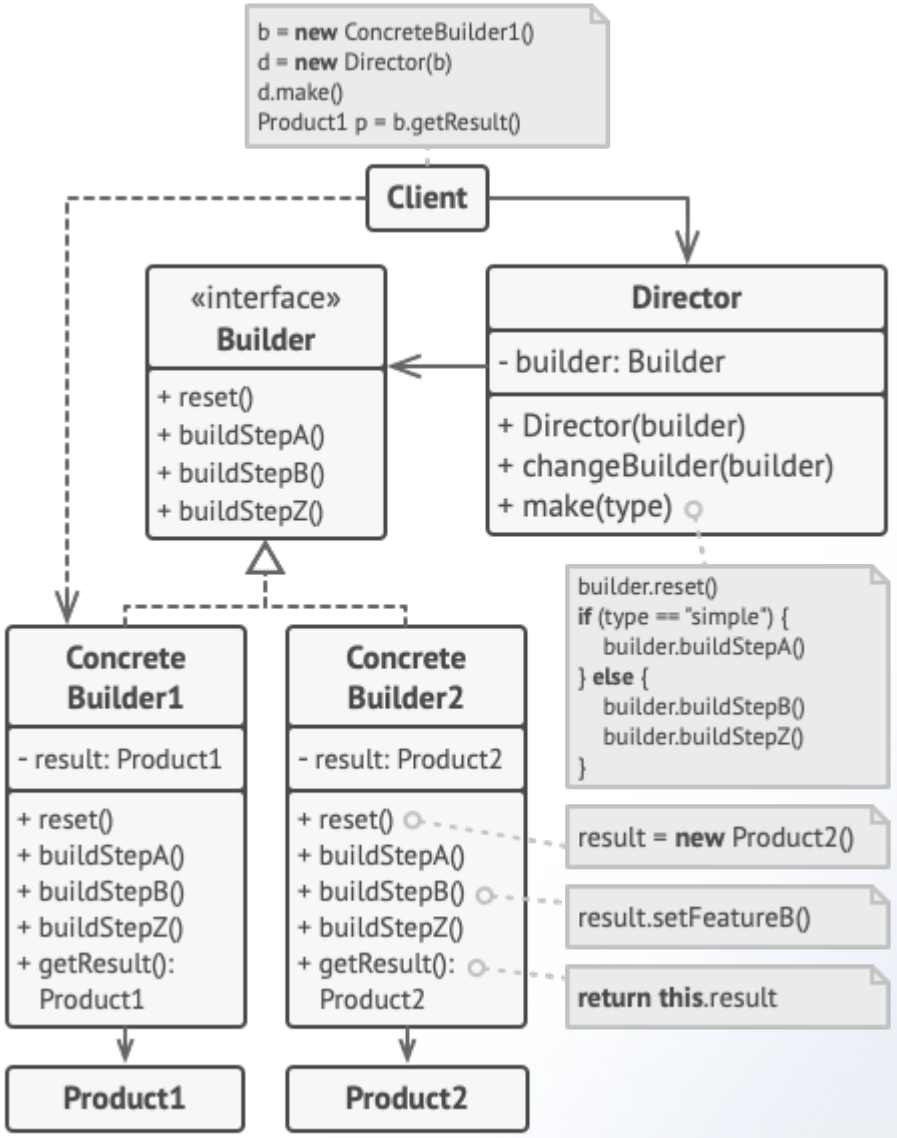


sourcemind

Creational patterns – Builder

The builder pattern was introduced to solve some of the problems with factory and abstract Factory design patterns when the object contains a lot of attributes. This pattern solves the issue with a large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

Creational patterns – Builder



sourcemind

Creational patterns – Prototype

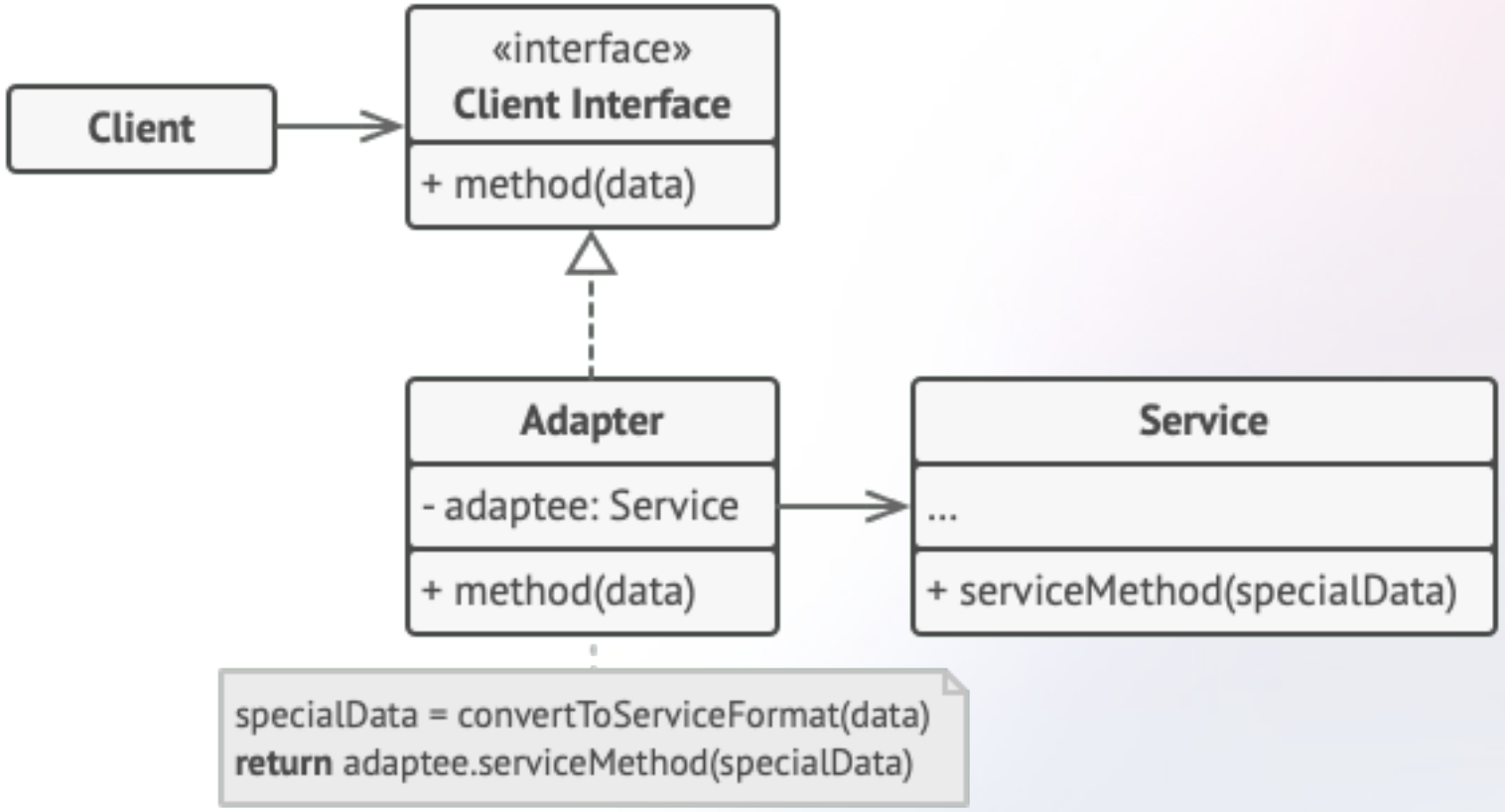
Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

Structural patterns

Structural patterns – Adapter

The adapter design pattern is one of the structural design patterns and is used so that two unrelated interfaces can work together. The object that joins these unrelated interfaces is called an adapter.

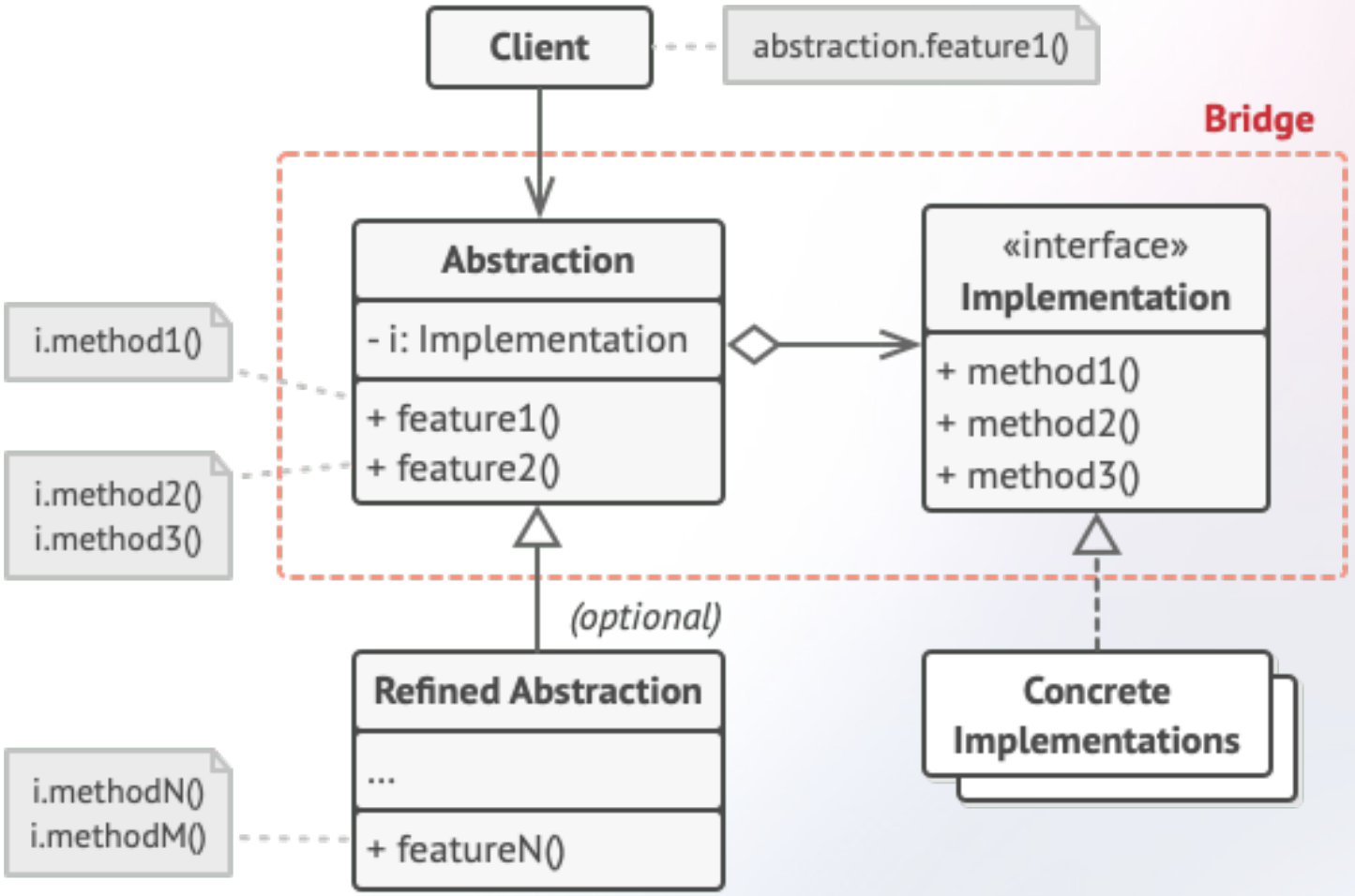
Structural patterns – Adapter



Structural patterns – Bridge

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

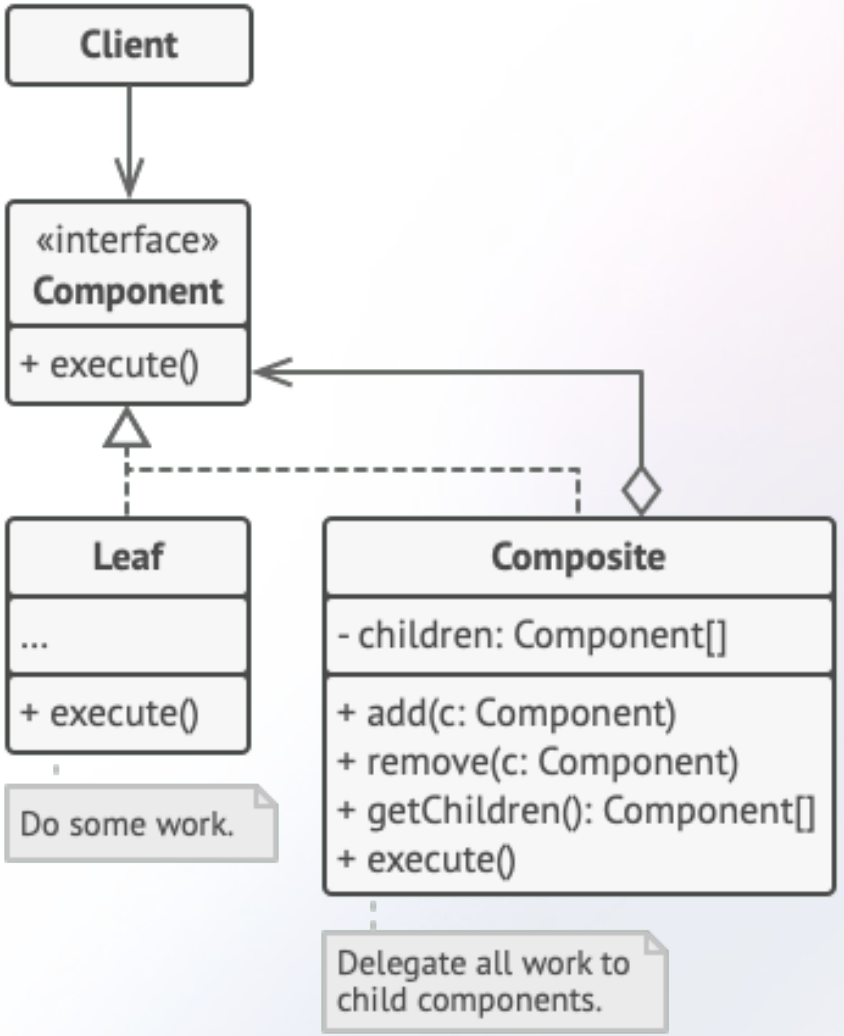
Structural patterns – Bridge



Structural patterns – Composite

When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern. Lets understand it with a real life example - A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc. When we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Structural patterns – Composite

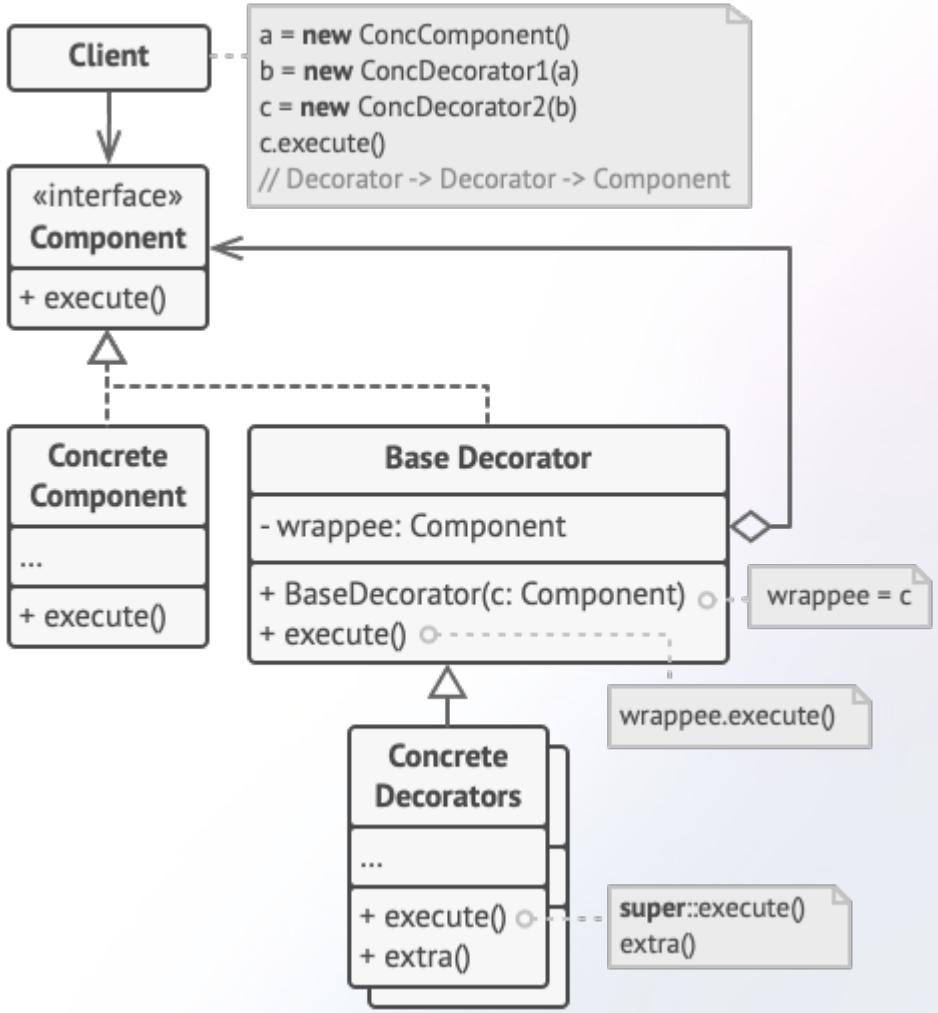


sourcemind

Structural patterns – Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Structural patterns – Decorator



sourcemind

Structural patterns – Facade

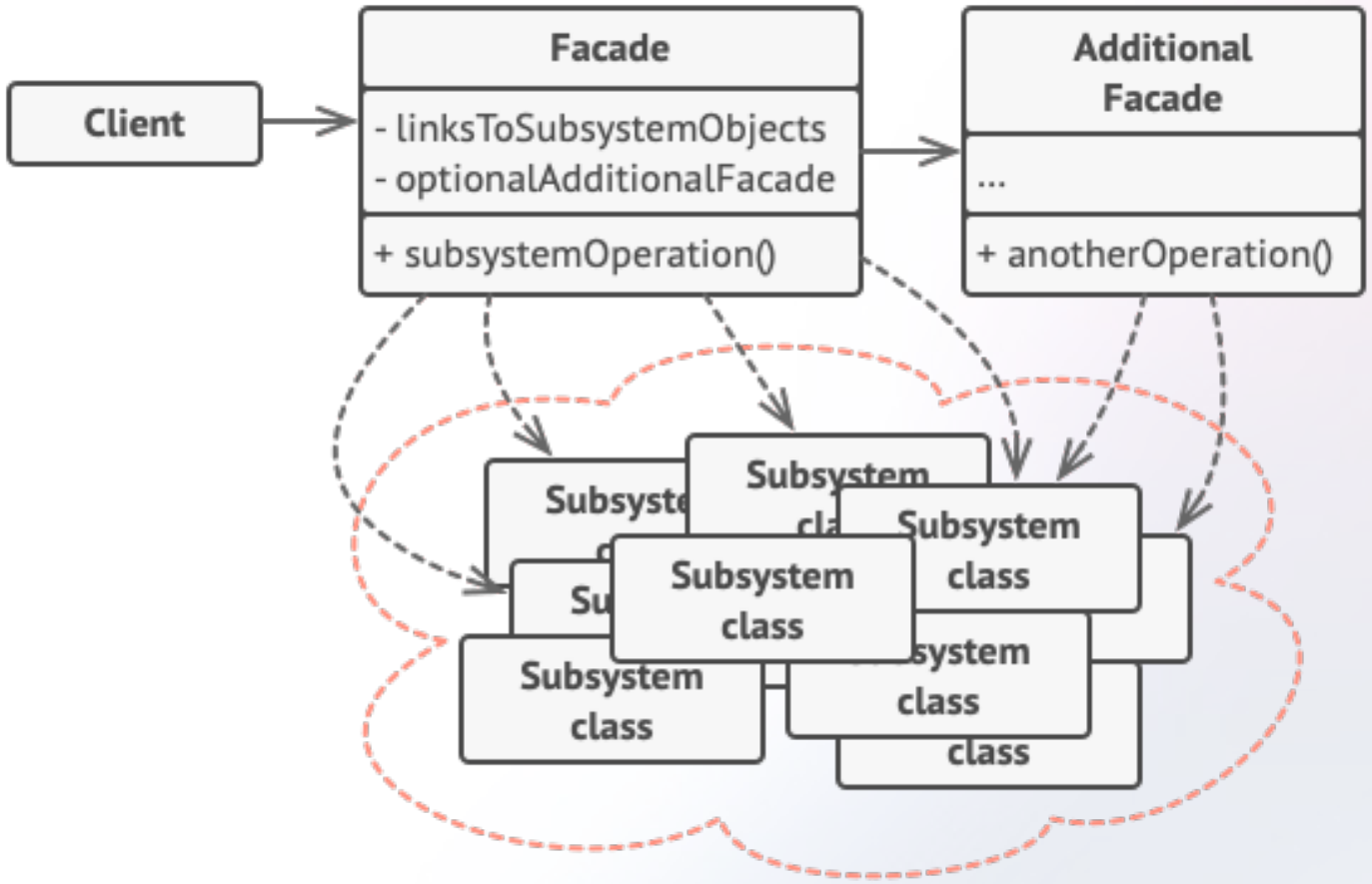
Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

According to GoF Facade design pattern is:

- Provide a unified interface to a set of interfaces in a subsystem.

- Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

Structural patterns – Facade



Structural patterns – Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Structural patterns – Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Behavioral Patterns

Behavioral Patterns – 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.

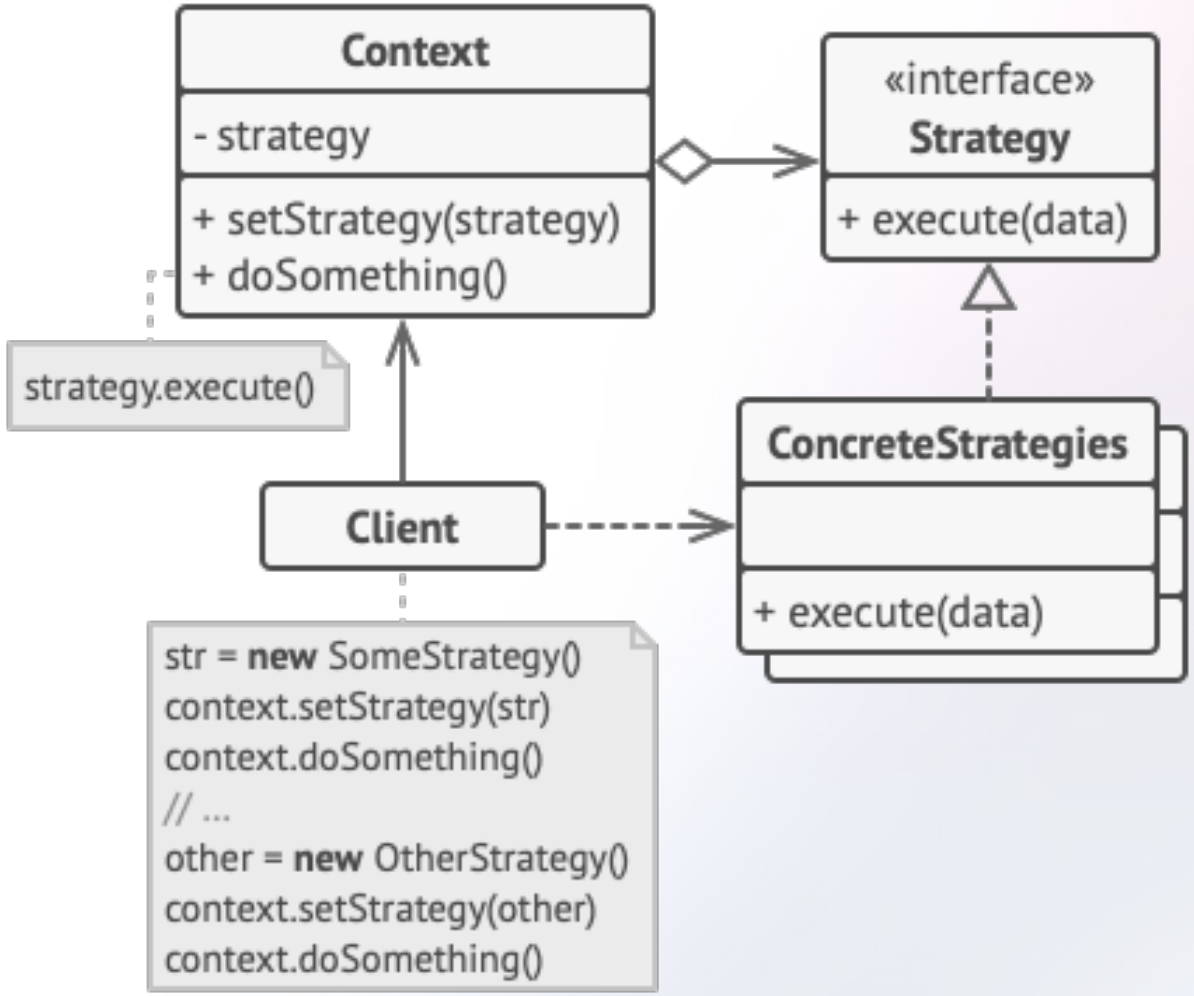
Payment is an example of strategy in our example

- Cash Payment
- Credit Card Payment
- …

All payment strategies support the same basic operations

- accept(), verify()
- Each strategy achieves the goal in a different way

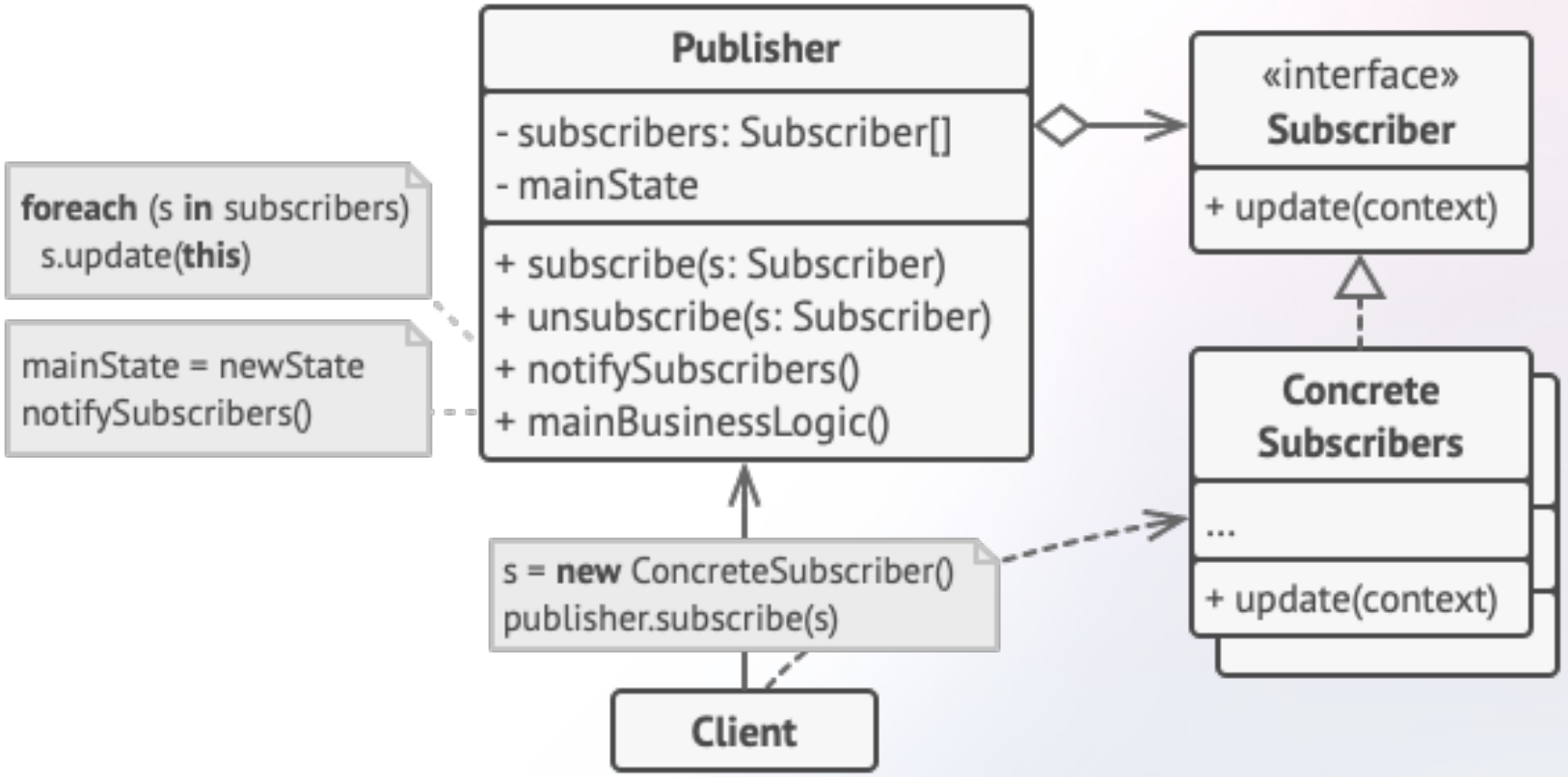
Behavioral Patterns – Strategy



Behavioral Patterns – Observer

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.

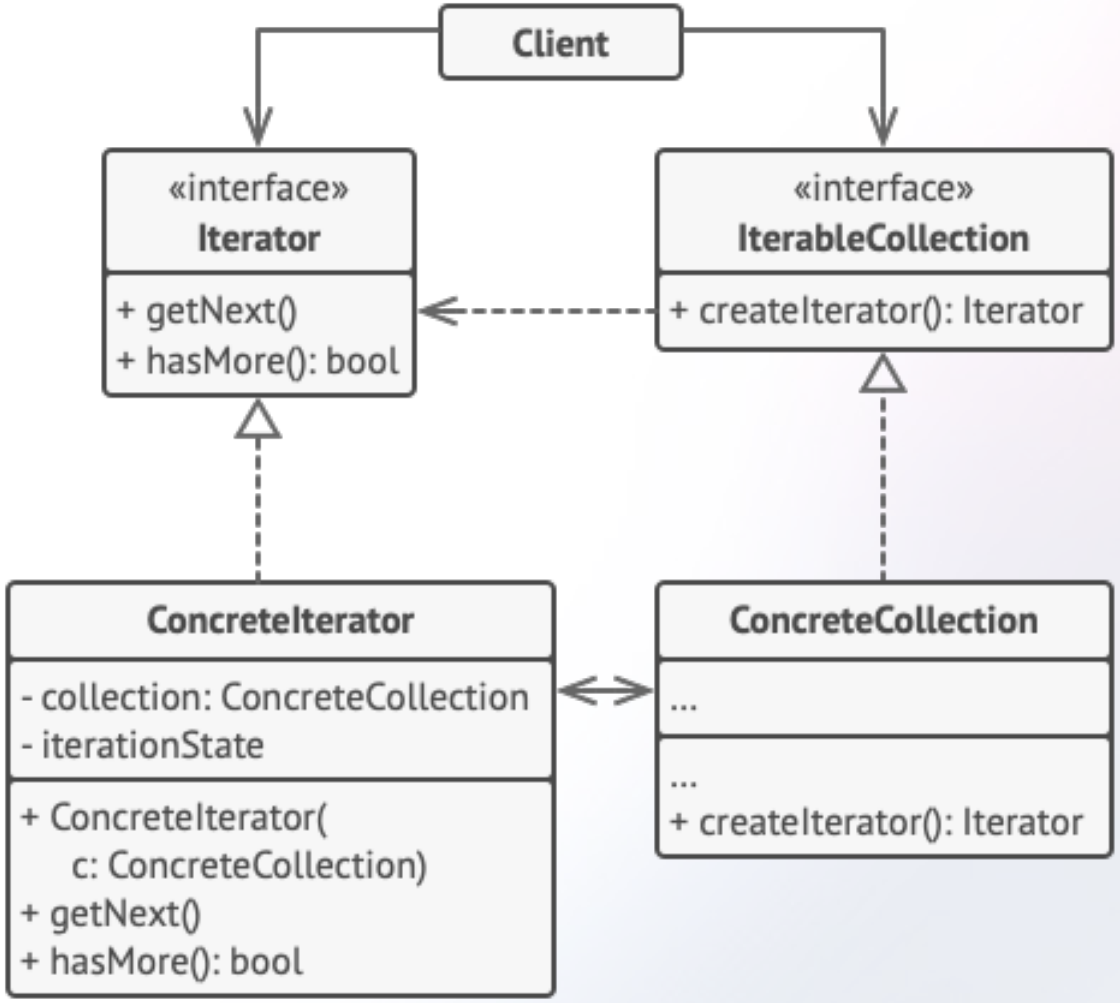
Behavioral Patterns – Observer



Behavioral Patterns – Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Behavioral Patterns – Iterator



sourcemind

Behavioral Patterns – Others

- Chain of Responsibility
- Command
- Mediator
- Memento
- State
- Template Method
- Visitor

End

sourcemind