Design Patterns

Dainish Jabeen

August 15, 2023

1 Introduction

Patterns describe evolved methods through trail and error, of commonly occurring problems.

2 SOLID

2.0.1 Single Responsibility Principle

A class should have just one reason to change.

2.0.2 Open/Close Principle

Classes should be open for extension but closed for modification.

2.0.3 Liskov Substitution Principle

Extending a class, should be able to use objects of the subclass instead of the parent class.

2.0.4 Interface Segregation Principle

Clients shouldn't be forced to depend on methods they dont use.

2.0.5 Dependancy Inversion Principle

High level classes should not depend on low level classes. Details should depend on abstractions.

3 Composition

Preferred to inheritance, the concept is to use interfaces to expand behaviours easily (in a plugin method).

4 Types of patterns

Creational Patterns: Increase flexibility and reusability.

Structural Patterns: Assemble classes to large structures while keeping them flexible and efficient.

Behavioural Patterns: Effective communication and responsibility between modules.

5 Creational Patterns

5.1 Factory

Create objects in superclass, alter in subclass.

The creation of objects is done using the factory method, that changes the object returned based on context (normally conditional statements) which is the subclass of a 'Creator' class . The return type of the function is an abstract so the client receiving does not know the details of the object.

This abstract type is an interface, which is detailed in concrete implementations that specify which object is created. Which are directly called via a subclass of the creator.

This eases the creation of new objects to be used, and is useful if it is context dependant which object should be created and used.

Can also create a pool of objects, which the factory method checks before creating a new object.

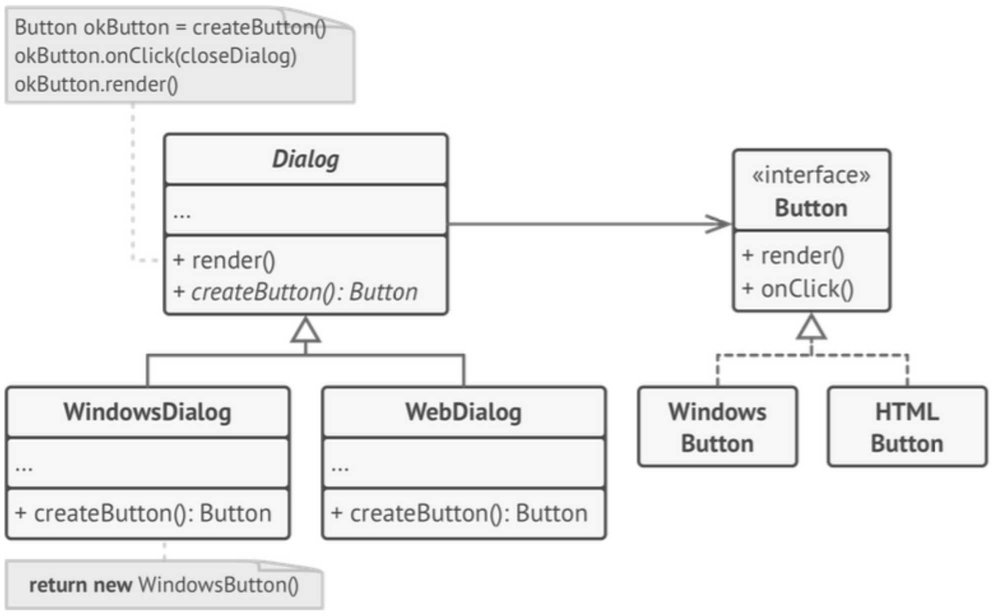


Figure 5.1: Factory Structure

Diagram of structure Figure:5.1 on Page:2.

5.2 Abstract Factory

Create families of objects without specifying its implementation.

If there matrix of implementations ie feature 1 x feature 2 (furniture x style). An interface is created for each characteristic on one dimension (interface for each furniture type) and an implementation of each the second dimensions characteristic on each interface (implementation for each style type).

The abstract factory is the creator interface with an implementation of each of the second dimensions characteristics (style types) that create each of first dimensions object with that chosen second dim. The return type is an interface of the first dimension allowing to use any first dim methods without the client knowing the second dim. Allowing easy additions of either dims. Extension method for first dimension and plugin method for second dim.

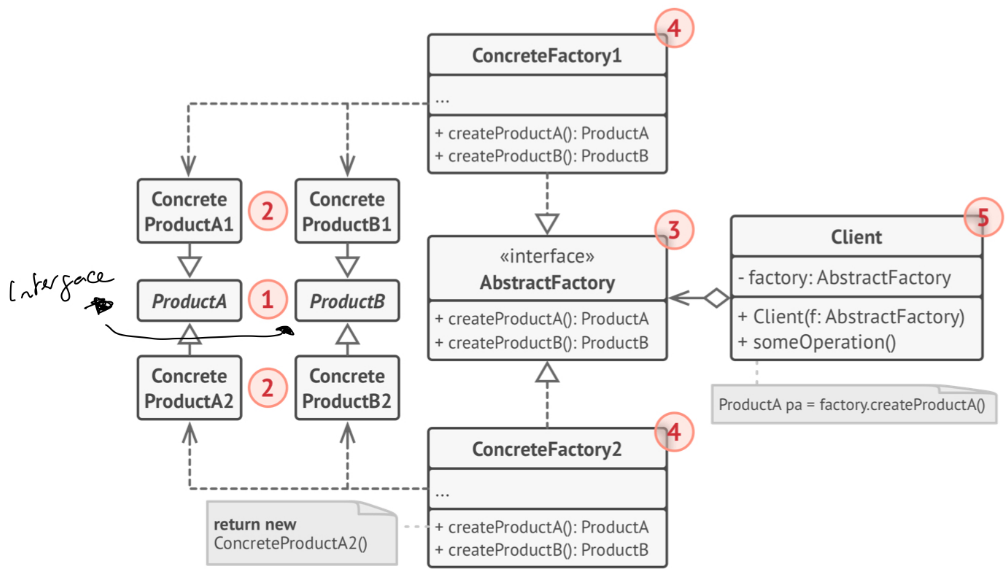

ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
Interface
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()
1
2
2
3
4
4
5

Figure 5.2: Abstract Factory Structure

Diagram of structure Figure:5.2 on Page:3.

5.3 Builder

Using the same construction code to build complex objects.

Solving the issues of classes with many parameters, create a builder modules thare implementations of an interface for the variations in the object. In each builder the parameters will be set, you dont need to use all parameters.

For easier construction reuse a director module with coded methods of using the builders. The client code will use the director via a concrete builder to get the object.

This gives you an easy way to extend variations of an object.

Diagram of structure Figure:5.3 on Page:9.

5.4 Prototype

Copy existing objects without dependancy.

Create a cloning interface common to objects that are to be duplicated. The implementation of the interface will have a method to clone all fields of that object.

Good for when many objects are storing state, easy way to get an object with a basic configuration. May not have access to all fields of an object to clone.

Diagram of structure Figure:5.4 on Page:10.

5.5 Singleton

Uses only one instance that is globally available. Help protect from threading issues a shared resource.

Make constructor of object private, used via method in module that creates ans stores object. When called will return created object.

Diagram of structure Figure:5.5 on Page:10.

6 Structural Patterns

6.1 Adapter

Communication between incompatible interfaces.

Using an interface when implemented will convert the type of one object to another by wrapping it (referencing it). Access to the methods of the object are then via the interface. Allowing a plugin model for adapters.

Diagram of structure Figure:6.1 on Page:11.

6.2 Bridge

Separate modules that are the implementations and abstract.

Modules write high level methods that uses an interface for details to create a plugin nature. The abstract methods use the implementations via the interface.

Diagram of structure Figure:6.2 on Page:11.

6.3 Composite

Objects in tree structure.

Deal with all bottom leaves and higher leaves with children in the same, allowing recursion to navigate through the tree and provide a result. A common interface is used to access each leaf, allowing an implementation be for a bottom leaf or an adult leaf, for the client its does not matter.

Diagram of structure Figure:6.3 on Page:12.

6.4 Decorator

Add new behaviours to objects via wrappers.

With a common interface between current object and the wrapper/decorator, a base decorator is created which is extended and has access to the base object. Allowing use of either base object or decorator without the client being aware.

Diagram of structure Figure:6.4 on Page:13.

6.5 Facade

Simple interface to complex objects.

Use an intermediary module handling all calls to the subsystem, so the client doesn't call it directly.

Diagram of structure Figure:6.5 on Page:14.

6.6 Flyweight

Save RAM by sharing objects.

Create objects "flyweights" that hold all intrinsic (not changing/identical across objects) data of objects, which is referenced to by objects that hold the extrinsic data. The flyweights are immutable, another module can be used to manage pools of flyweights.

Diagram of structure Figure:6.6 on Page:14.

6.7 Proxy

Substitute or placeholder for object, controls access to objects so can perform methods before/after.

Using an interface for both the base object and proxy, with proxy linking to the service.

Diagram of structure Figure:6.7 on Page:15.

7 Behavioural

7.1 Chain of responsibility

Pass request along chain of handlers, each handler deciding if to use it then passes it on.

A handler is a single module with a single method (execute) and a ref to the next handler in chain . All handler implement the same interface, this allows the chain to be dynamically created at run time.

The handler will make a decision on if the data should be passed on or if the chain can stop, therefore only executing the number of steps required. For a cleaner setup can have a base handler which is extended by all handlers.

Diagram of structure Figure:7.1 on Page:16.

7.2 Command

Turn a request into an object for easier handling; can parametrise, delay or queue requests.

Using a 'command' module implementing an interface between procedure and low level details with a single execute method. Allowing the command of a interface to become a plugin. The interface will simply alert the command, the command will then gather the data needed on its own to further decouple them.

If an invoker module is used to setup the required command modules and execute them when receiving a signal from the interface can stack commands and reuse the objects.

Diagram of structure Figure:7.2 on Page:17.

7.3 Iterator

Traverse elements without exposing its details.

Storage patterns may require different traversal methods, this can be separated into modules that implement an interface. To give the traversal a plugin nature. Can use a interable collection interface so grouping of storage method and iterators is easy.

Diagram of structure Figure:7.3 on Page:17.

7.4 Mediator

Reduce dependencies between objects and by controlling communication.

Components that are independent should not directly communicate, better to go via a mediator that redirects calls based on needs. So the components depend on the mediator and not on each other. Can use an interface for using multiple mediators, so that each component use the interface further decoupling further the interfaces.

Should be careful of the size of the mediator.

Diagram of structure Figure:7.4 on Page:18.

7.5 Memento

Save and restore previous state of an object.

Best method is to allow the object to create a copy of its own state to not void the encapsulation. The copy of the state is a special object called memento. A separate Caretaker can then store this object.

The memento can be created by the original object via an interface, allowing multiple memento to be used polymorphically.

Diagram of structure Figure:7.5 on Page:18.

7.6 Observer

Notify events occurring of an observed object.

Via an interface a publisher module will communicate to subscribers. The publisher will hold a list of all subscribers, with methods to add or remove a sub. Can also have a interface for the publisher.

Diagram of structure Figure:7.6 on Page:19.

7.7 State

Change behaviour based on state.

Finite state machine are normally based on conditionals, but this lacks evolvability. Each state has its own module, with the original object holding a ref to its current state. And a method to change states with the state implementations holding the state specific behaviour which is accessed via the interface.

The states are implemented via an interface, easy adding of states.

Diagram of structure Figure:7.7 on Page:19.

7.8 Strategy

Define a family of algorithms with each object interchangeable.

Split modules an implement via an interface an algorithm that does something specific in many ways. Use a context module to keep track of the strategies and allow use via interface. The client will then decide which strategy to use. Context module used for cleanliness.

Can use a set of anonymous functions instead of this interface pattern; functions without an identifier. This allows you to set the different functions in the context module at compile time to be used and easily add.

Diagram of structure Figure:7.8 on Page:20.

7.9 Template Method

Define structure of procedure and then adjust via subclasses.

The high level module will have abstract methods which have to be implemented by extensions and optional methods.

Diagram of structure Figure:7.9 on Page:21.

7.10 Visitor

Separate algorithms from objects on which they operate.

Create a separate module for the algorithms via an interface and choose which method is require for the current object via double dispatch. Double dispatch is when the object has a method that will use the correct method in the algorithm module accessed via the interface.

Diagram of structure Figure:7.10 on Page:22.

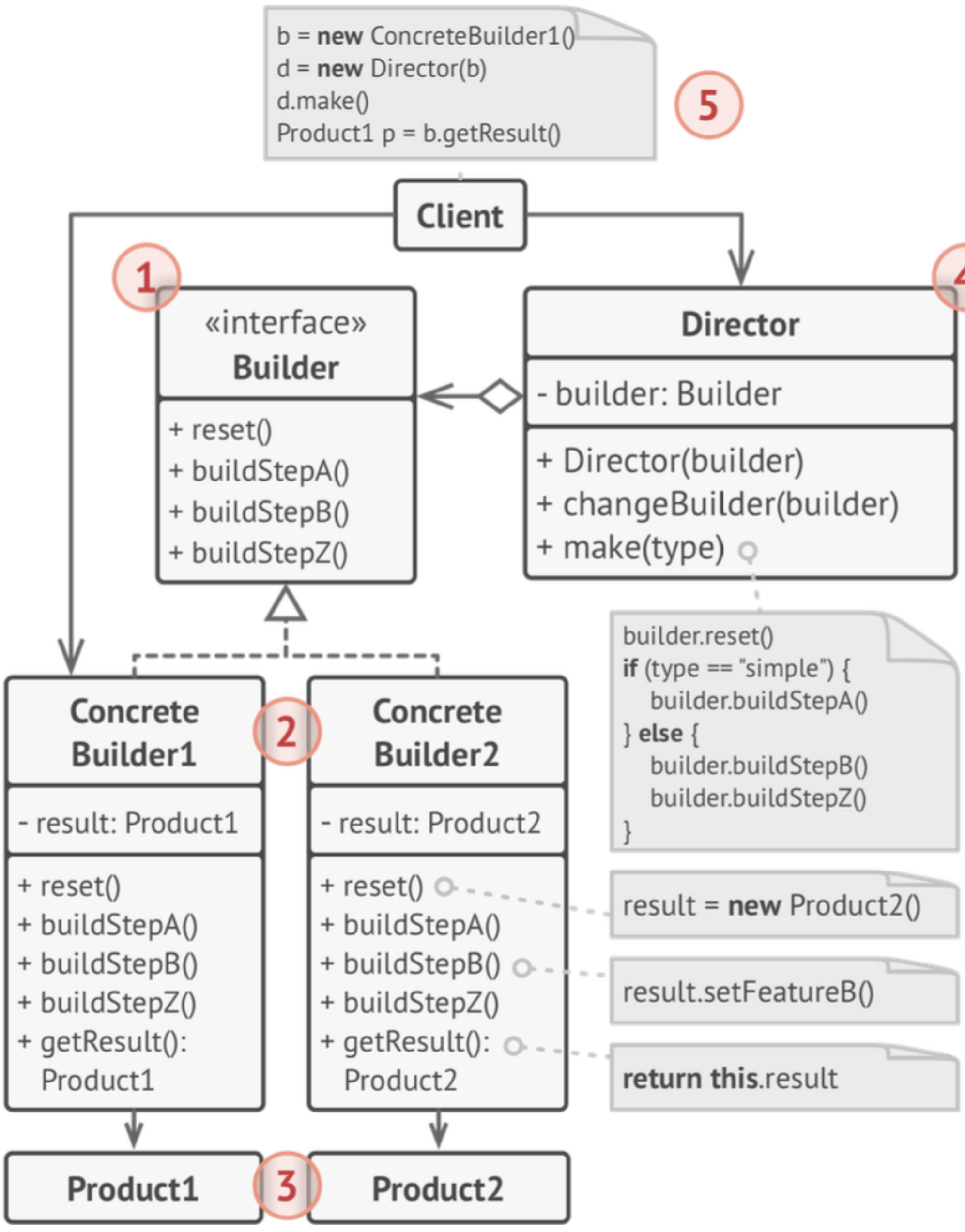


Figure 5.3: Builder Structure

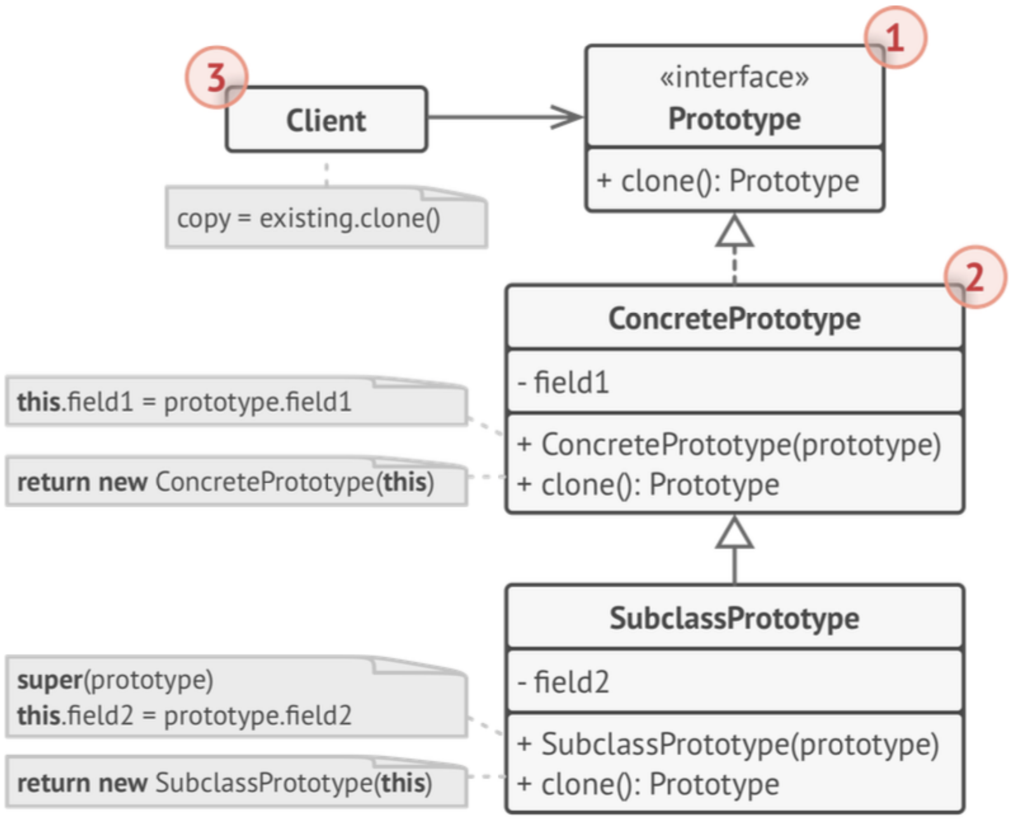
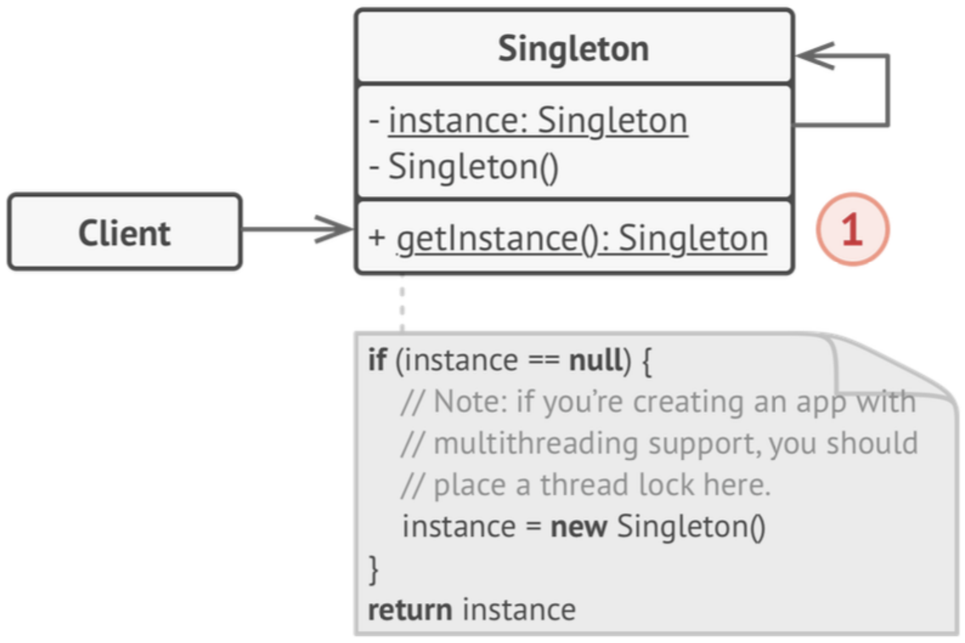

3
Client
copy = existing.clone()
1
«interface»
Prototype
+ clone(): Prototype
2
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)

Figure 5.4: Prototype Structure

– – –



```
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
```

Figure 5.5: Singleton Structure

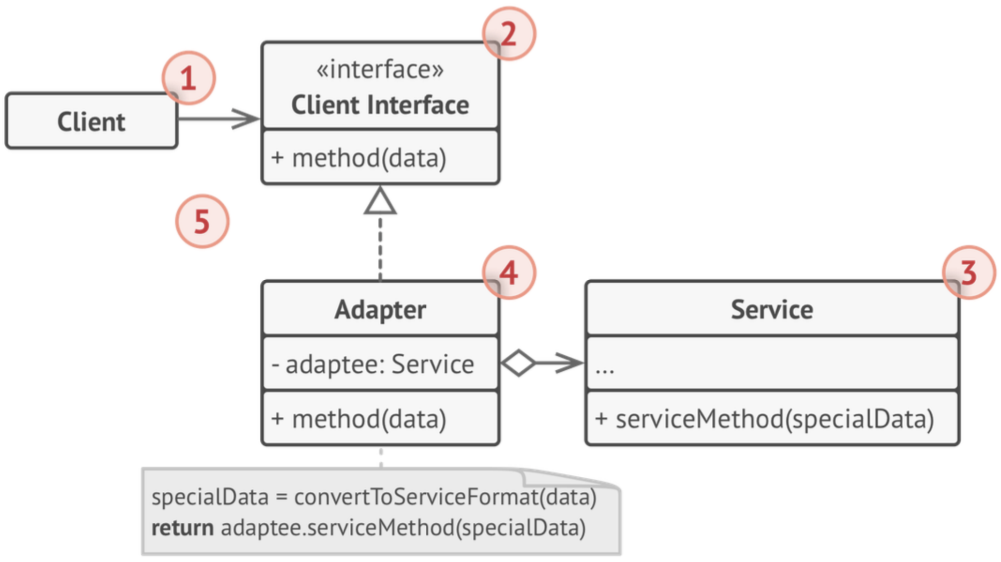

1
2
3
4
5
Client
«interface»
Client Interface
+ method(data)
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

Figure 6.1: Adapters Structure

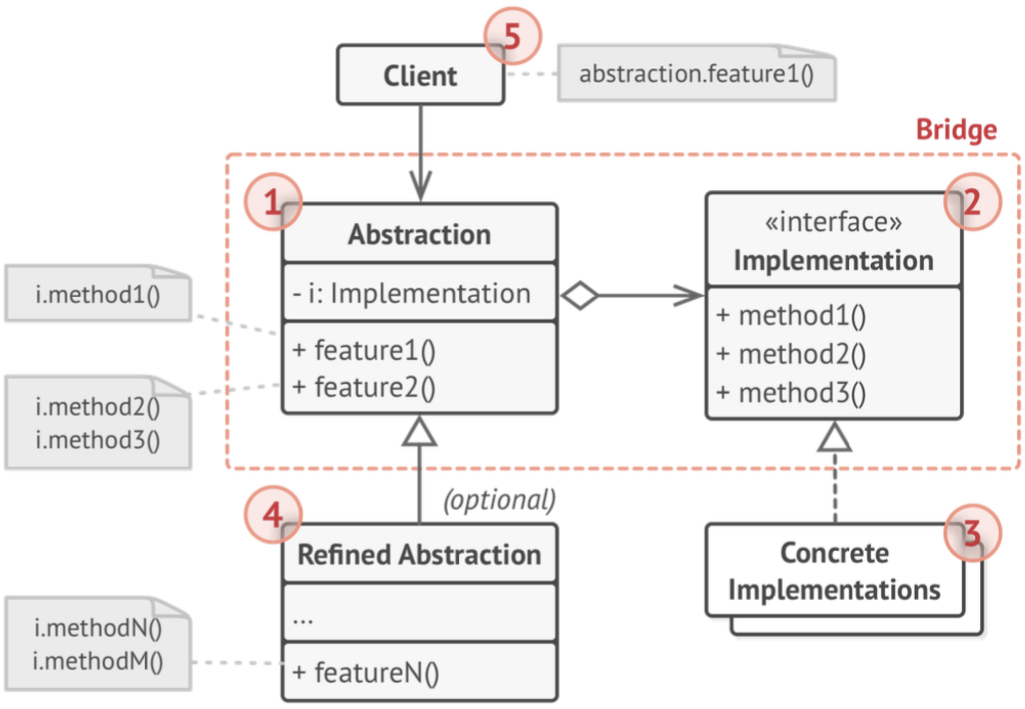


Figure 6.2: Bridge Structure

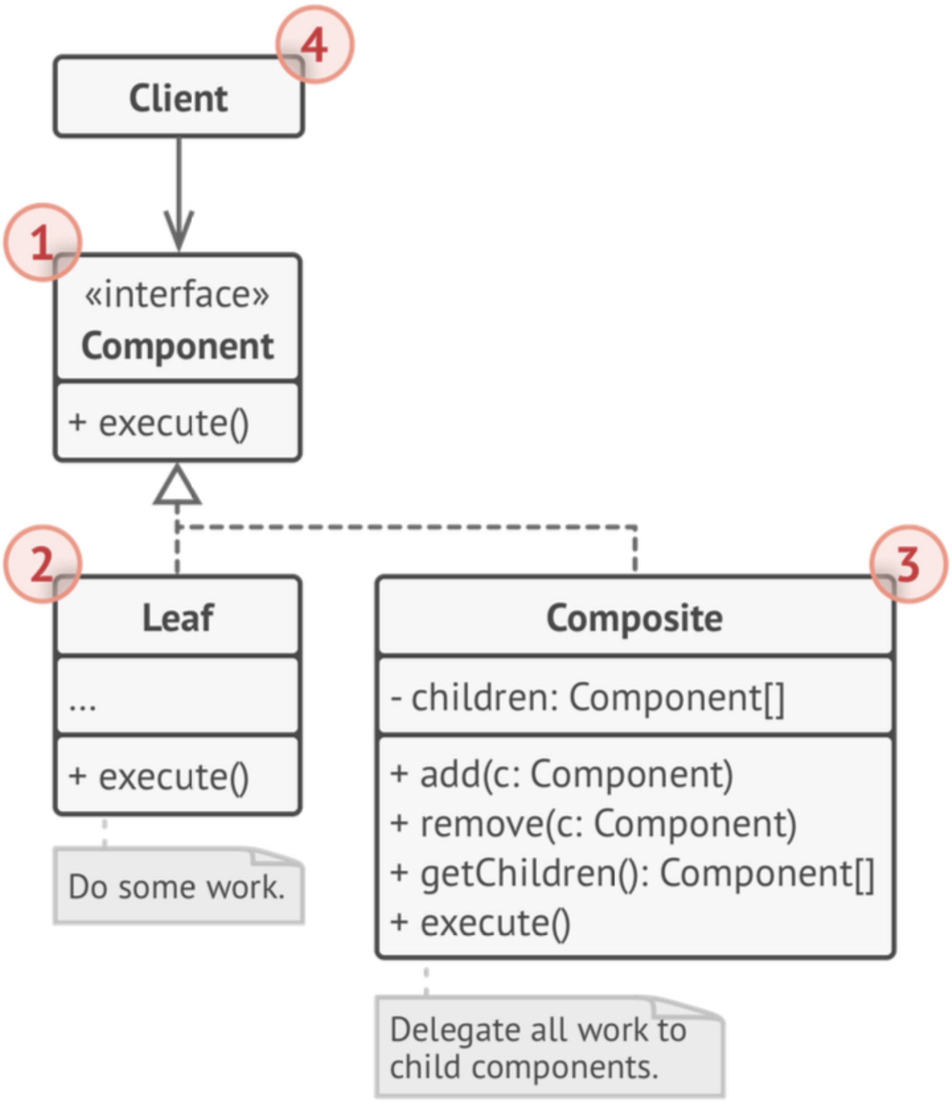


Figure 6.3: Composite Structure

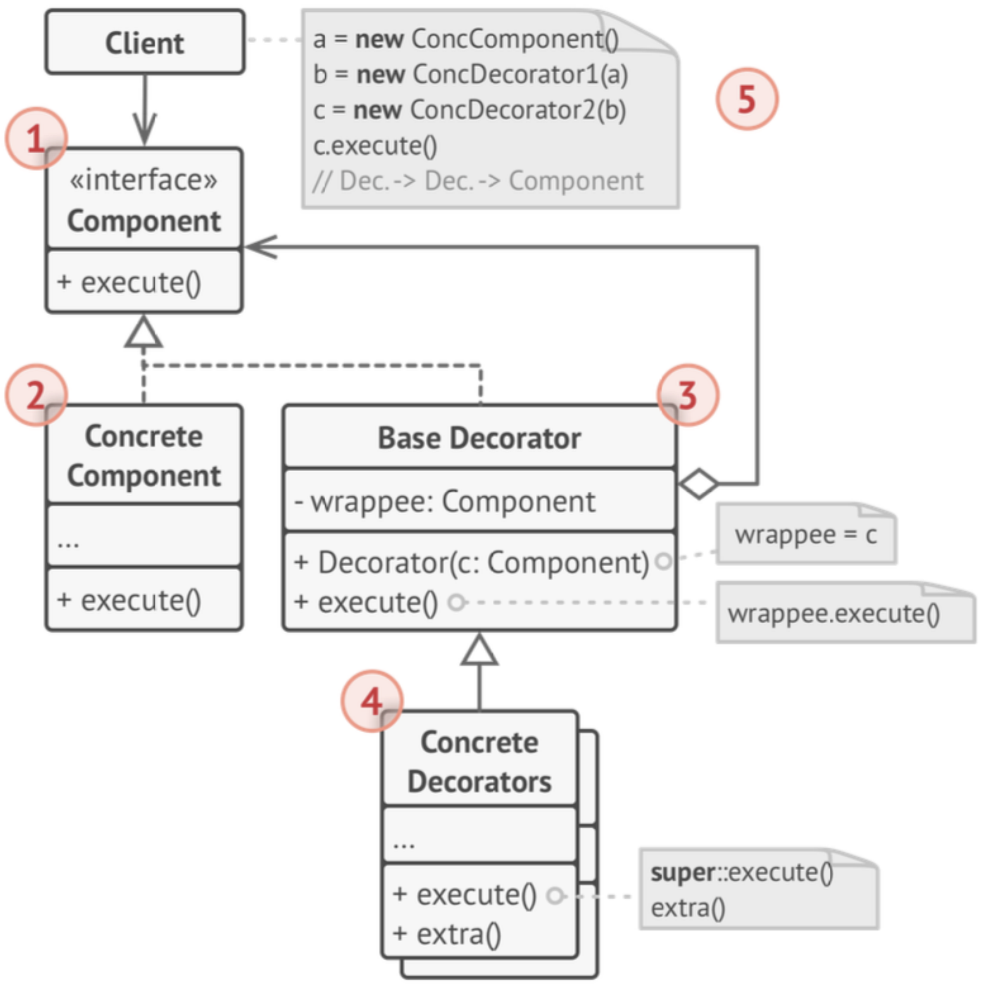


Figure 6.4: Decorator Structure

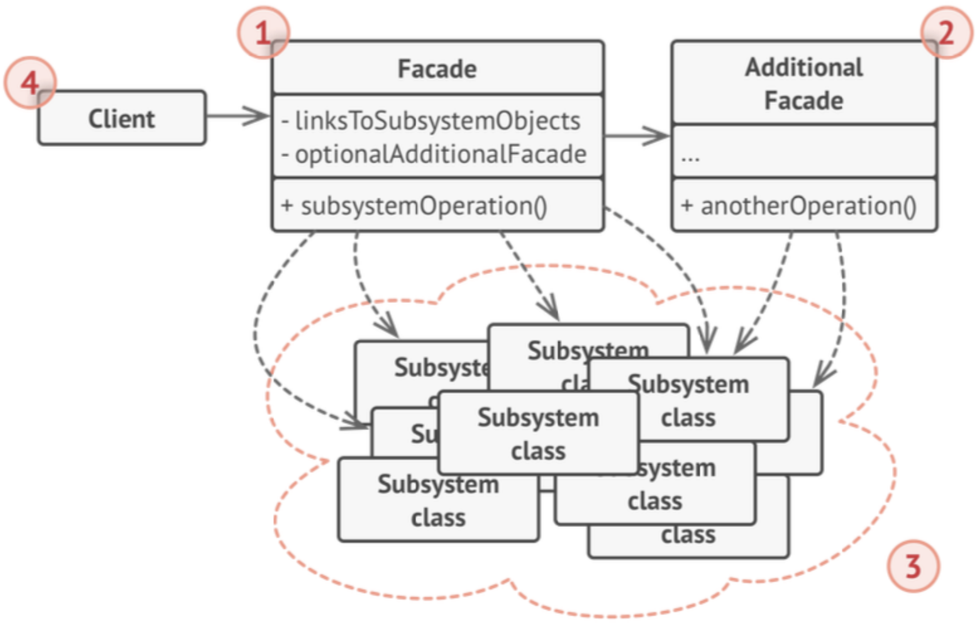
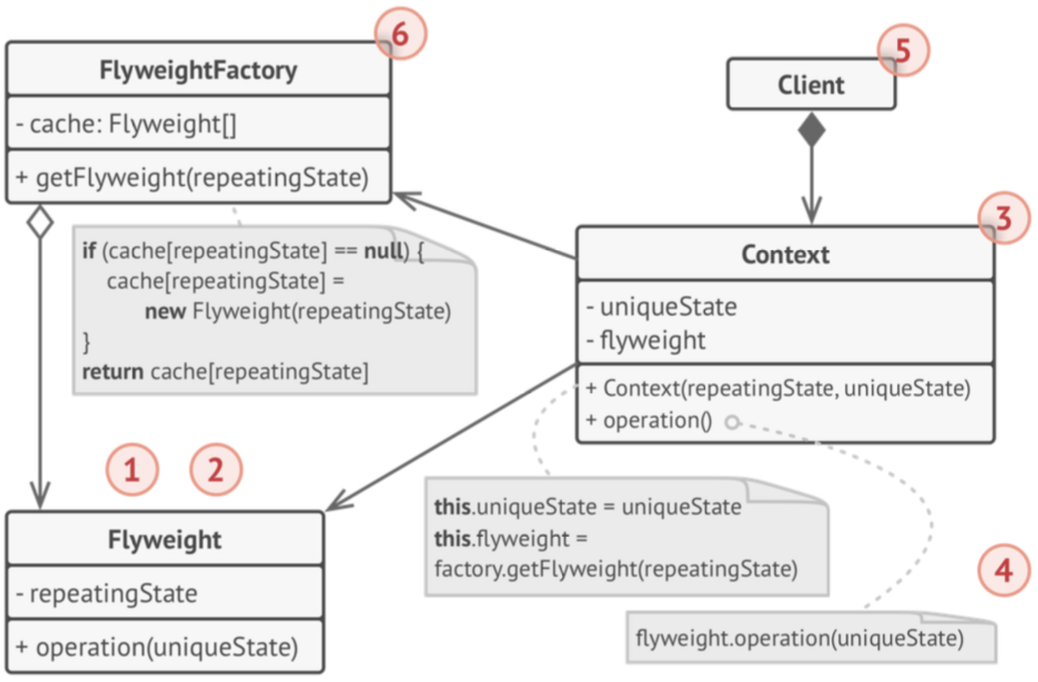


Figure 6.5: Facade Structure



6
FlyweightFactory
- cache: Flyweight[]
+ getFlyweight(repeatingState)
if (cache[repeatingState] == null) {
cache[repeatingState] =
new Flyweight(repeatingState)
}
return cache[repeatingState]
5
Client
3
Context
- uniqueState
- flyweight
+ Context(repeatingState, uniqueState)
+ operation()
1
2
Flyweight
- repeatingState
+ operation(uniqueState)
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
4
flyweight.operation(uniqueState)

Figure 6.6: Flyweights Structure

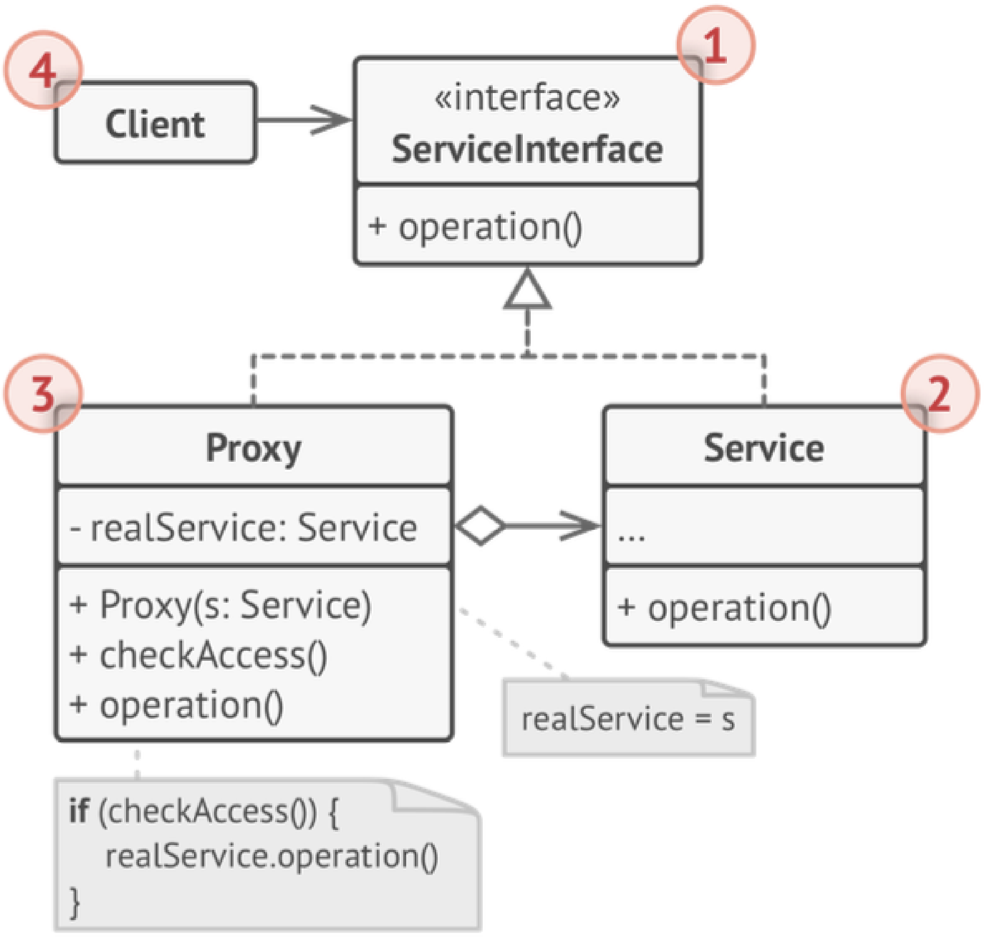
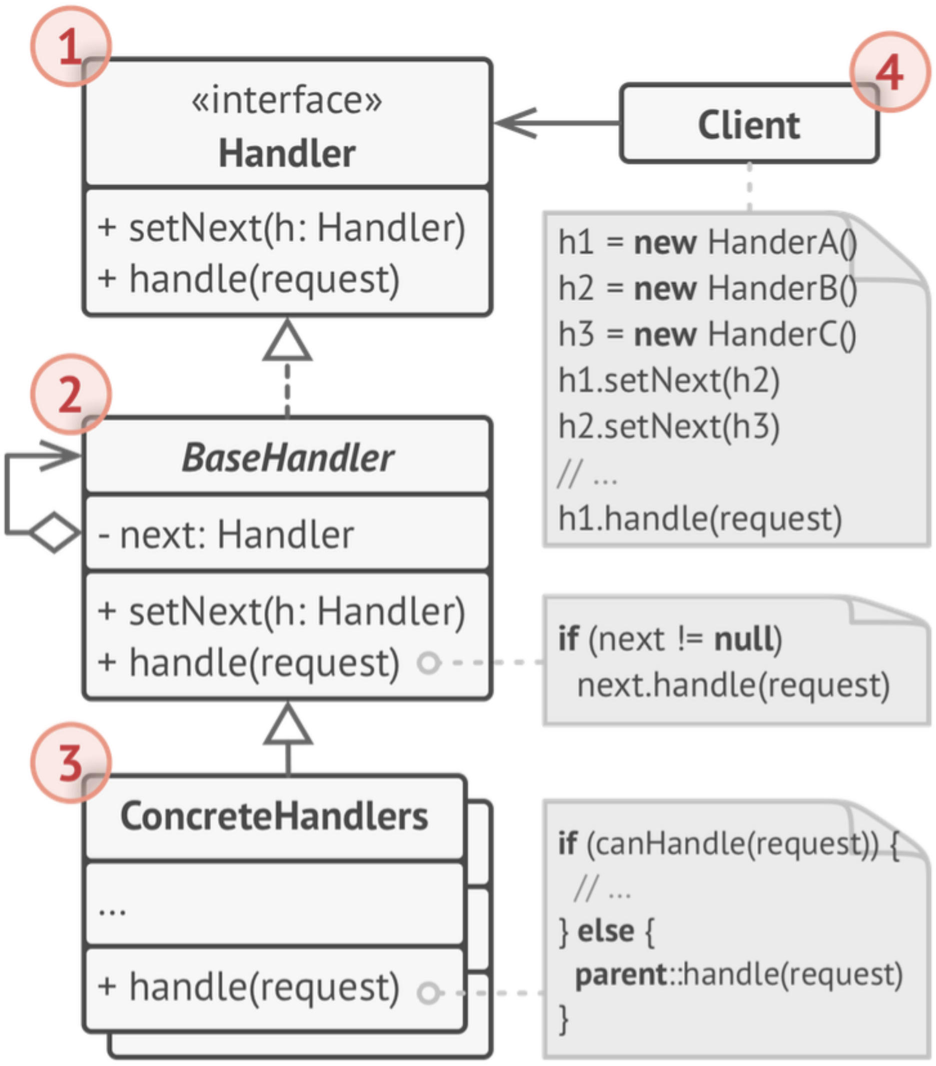


Figure 6.7: Proxy Structure



```
h1 = new HanderA()
h2 = new HanderB()
h3 = new HanderC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

Figure 7.1: Chain of Responsibility Structure

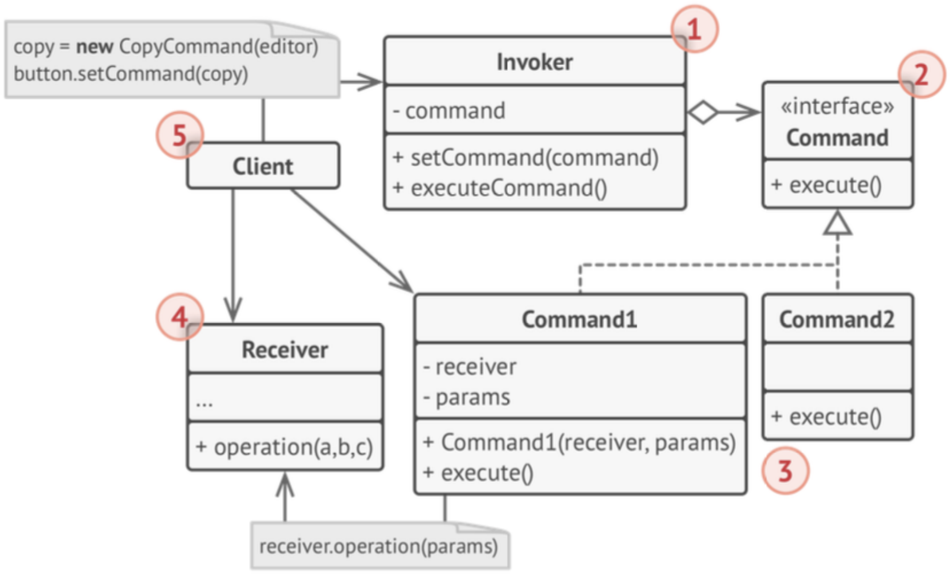


Figure 7.2: Command Structure

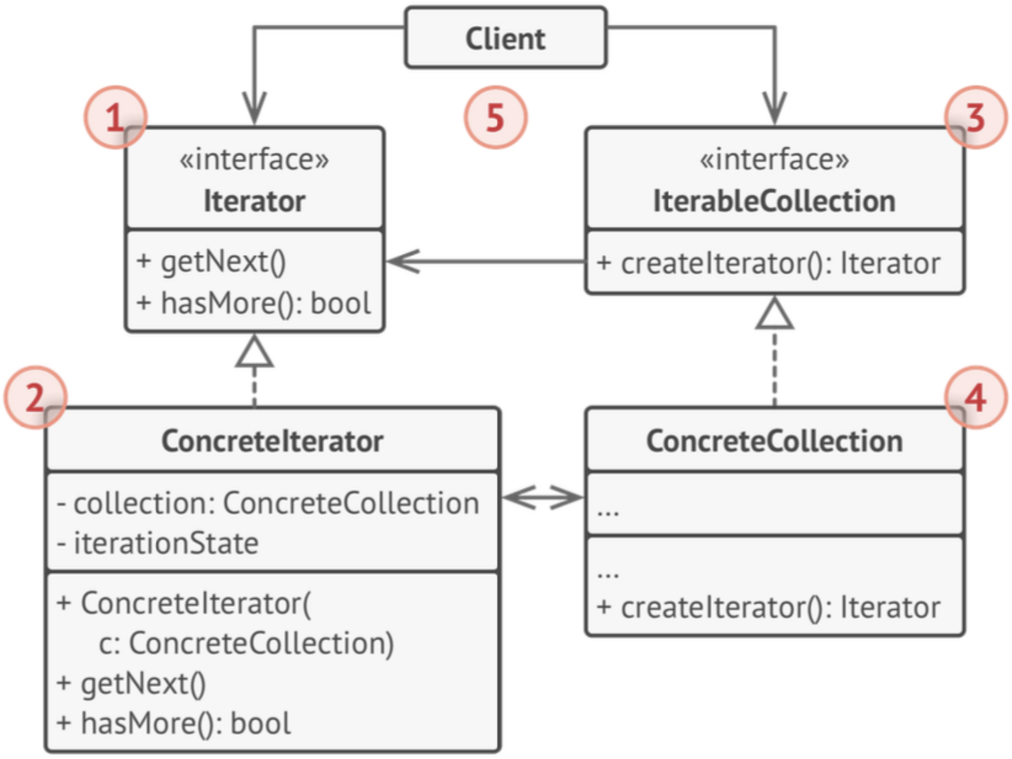


Figure 7.3: Iterator Structure

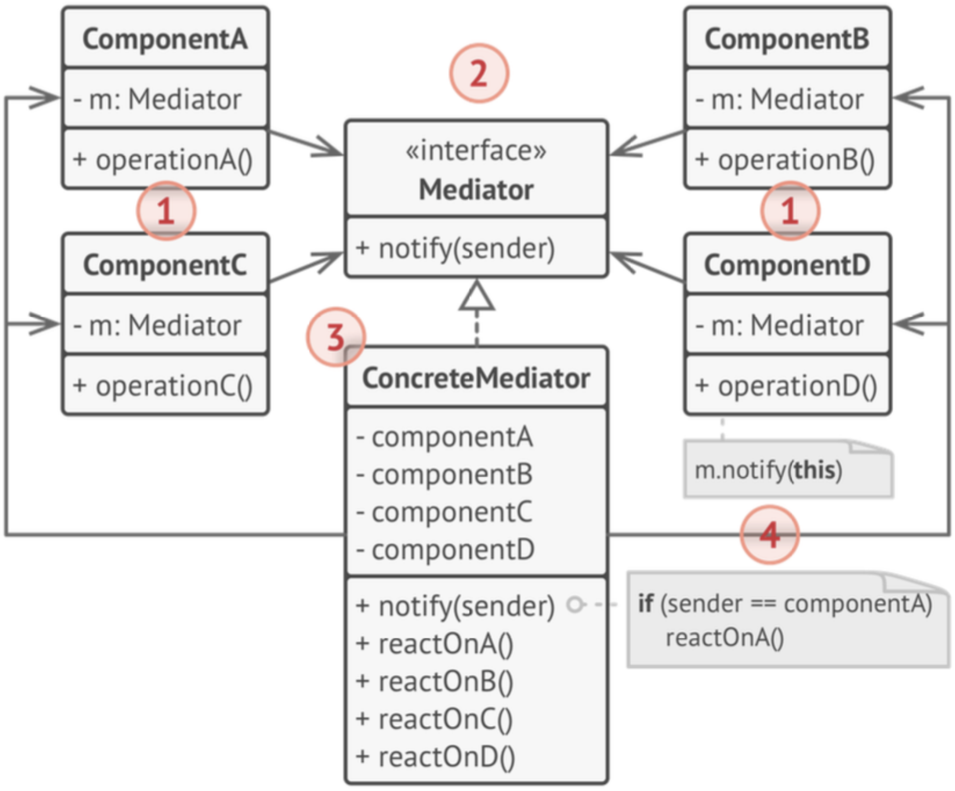


Figure 7.4: Mediator Structure

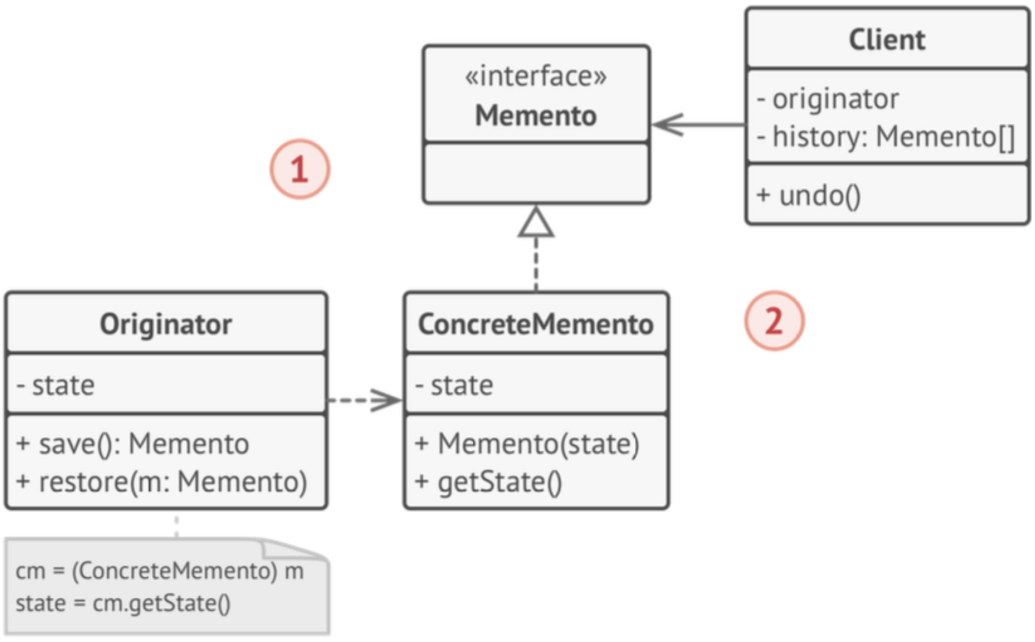


Figure 7.5: Memento Structure

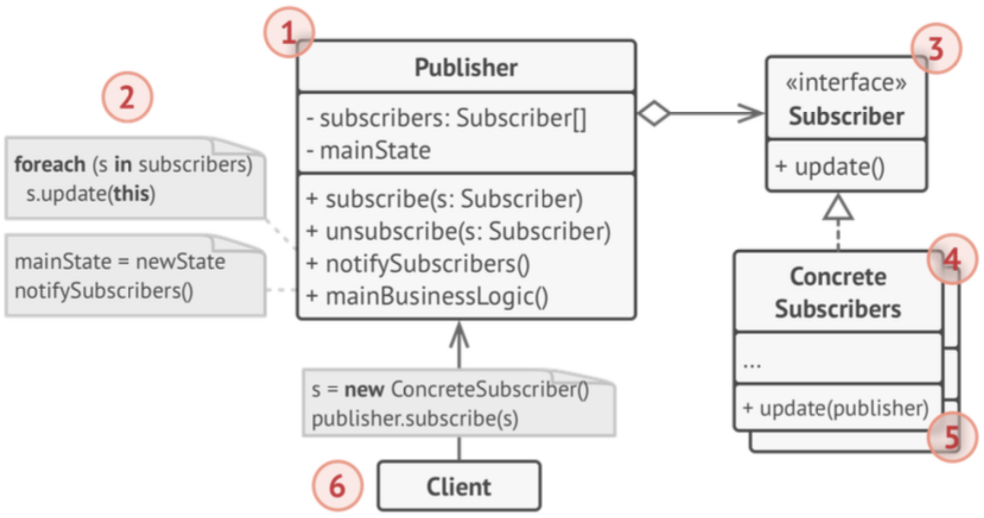


Figure 7.6: Publisher Structure

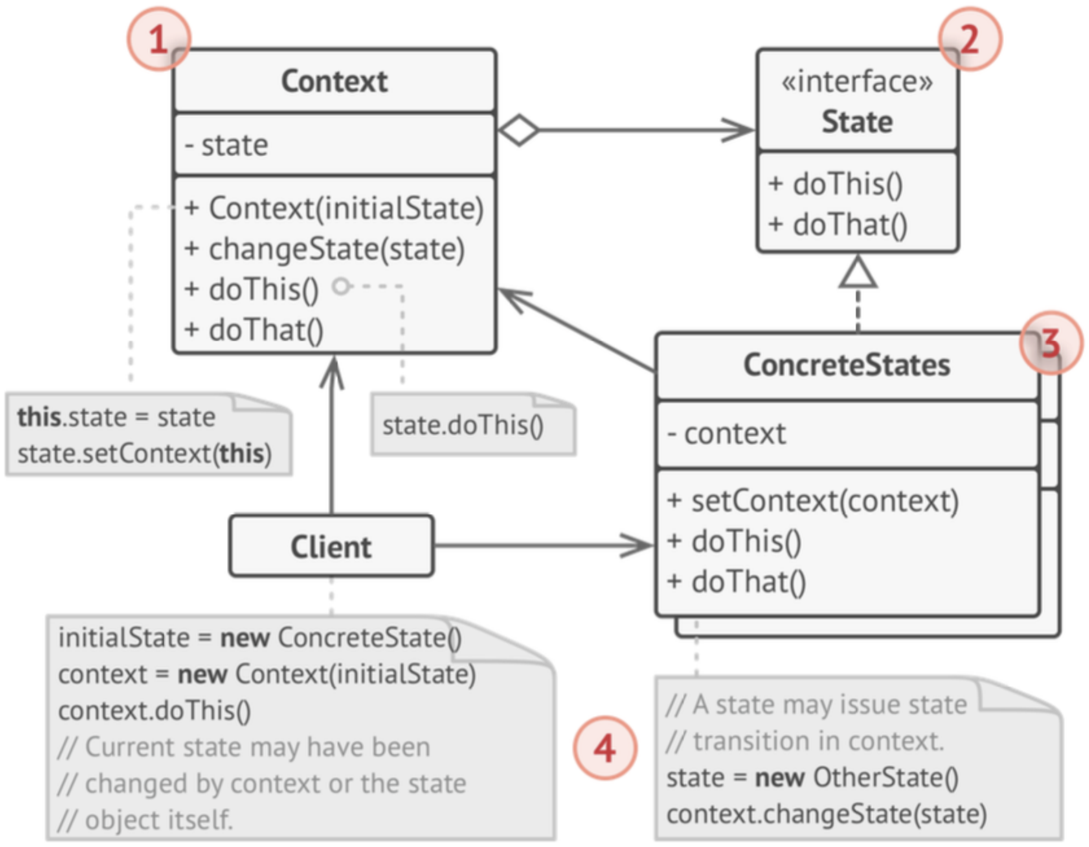


Figure 7.7: State Structure

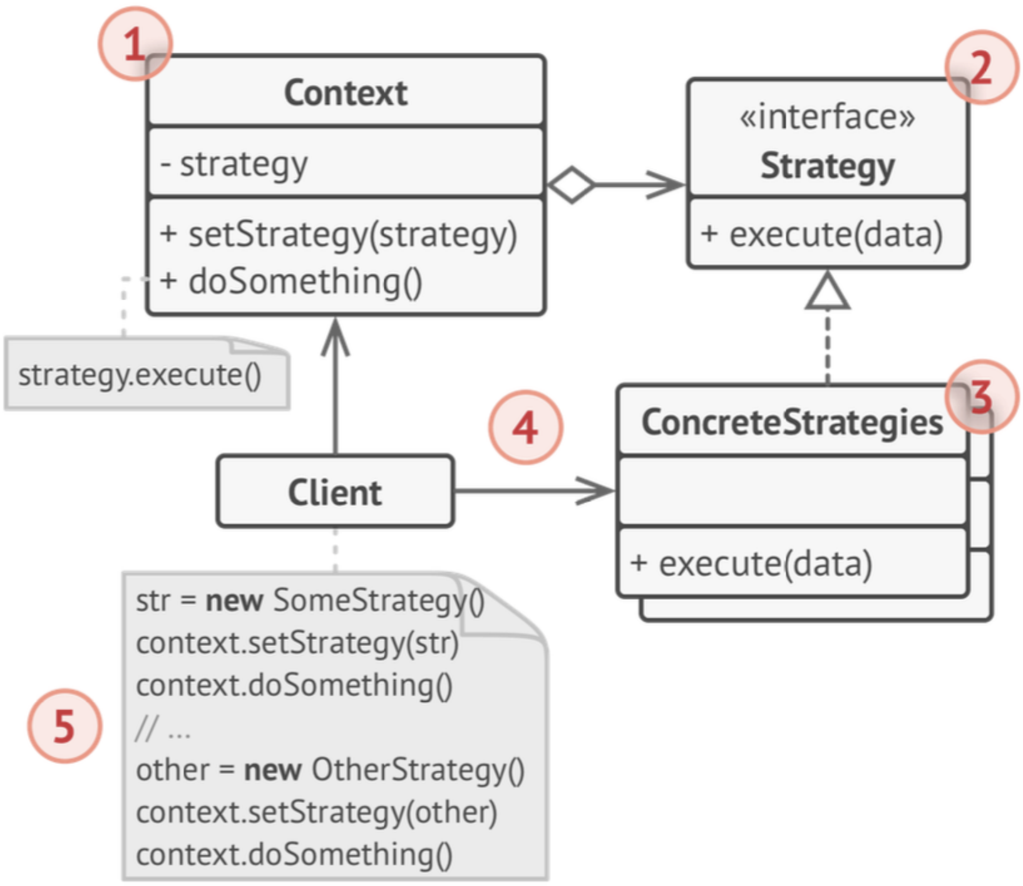


Figure 7.8: Strategy Structure

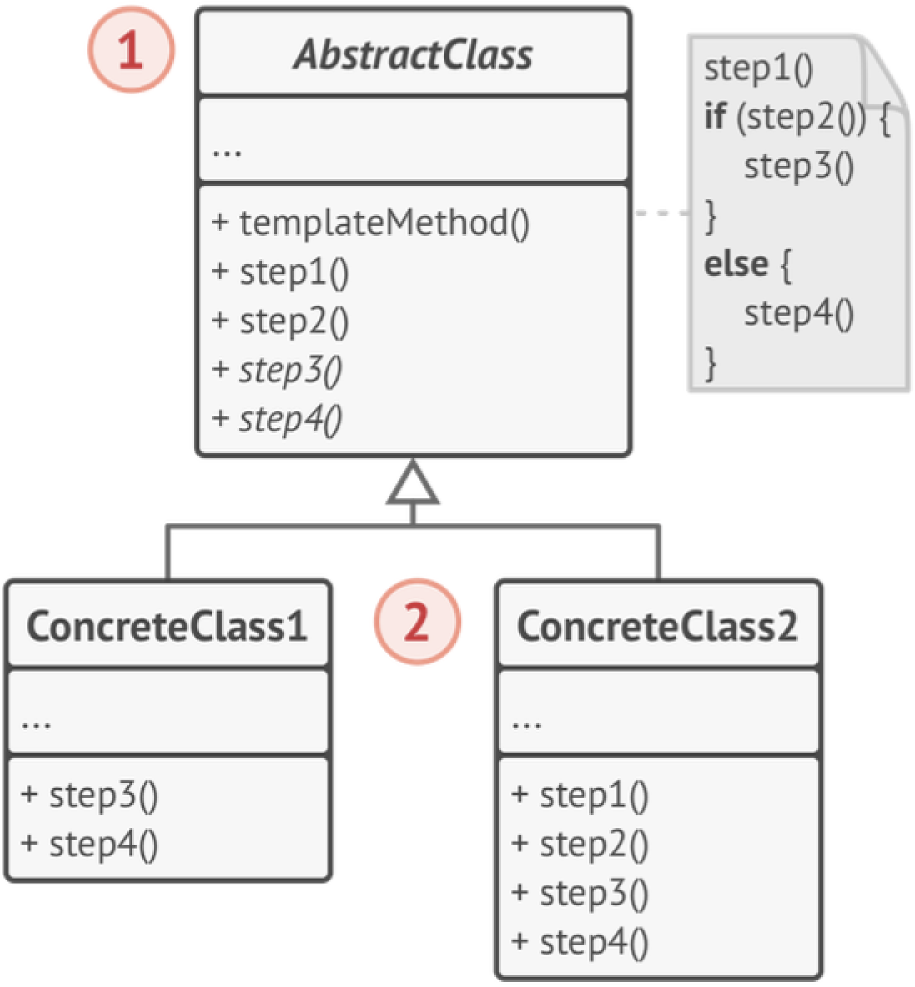


Figure 7.9: Template Method Structure

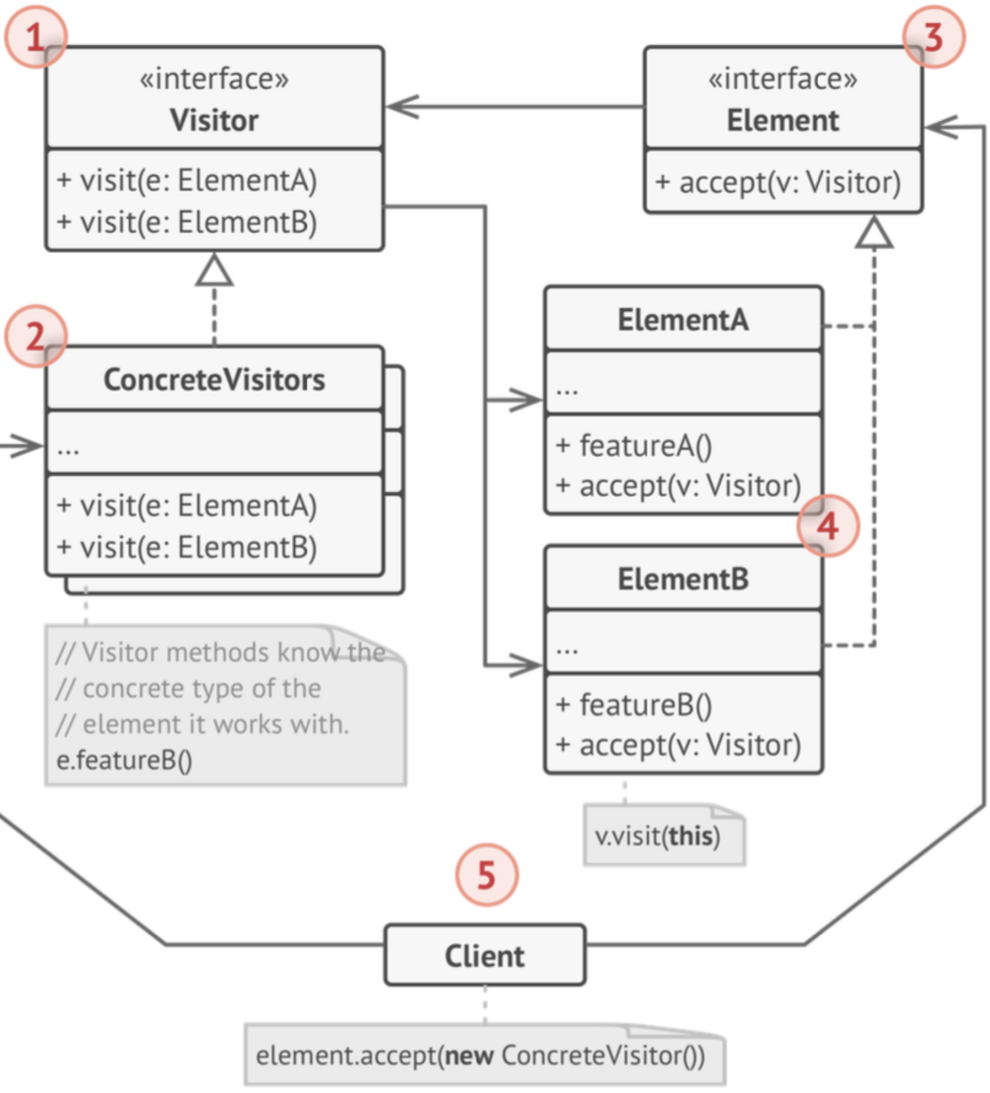


Figure 7.10: Visitor Structure