Design patterns

Design patterns are solutions to recurring problems, **guidelines on how to tackle certain problems**. They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations.

Design patterns

- You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.
- Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.
- pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Why we learn patterns?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

- Design patterns are 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.
- Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

Be careful!

- Design patterns are not a silver bullet to all your problems.
- Do not try to force them; bad things are supposed to happen, if done so.
- Keep in mind that design patterns are solutions **to** problems, not **solutions finding problems** so don't overthink.
- If used in a correct place in a correct manner, they can prove to be a savior, or else they can result in a horrible mess of a code.
- Do no apply them everywhere, even in situations where simpler code would do just fine.

Types of Design Patterns

- Creational patterns
- Structural patterns
- Behavioral patterns

Creational Design Patterns

- Creational patterns are focused towards how to instantiate an object or group of related objects.
- These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Structural Design Patterns

- These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

- In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

Creational pattern examples

- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural Pattern examples

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

Behavioral Design Patterns

- In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

- It is concerned with assignment of responsibilities between the objects. What makes them different from structural patterns is they don't just specify the structure but also outline the patterns for message passing/communication between them. Or in other words, they assist in answering "How to run a behavior in software component?"

Behavioral Pattern example

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

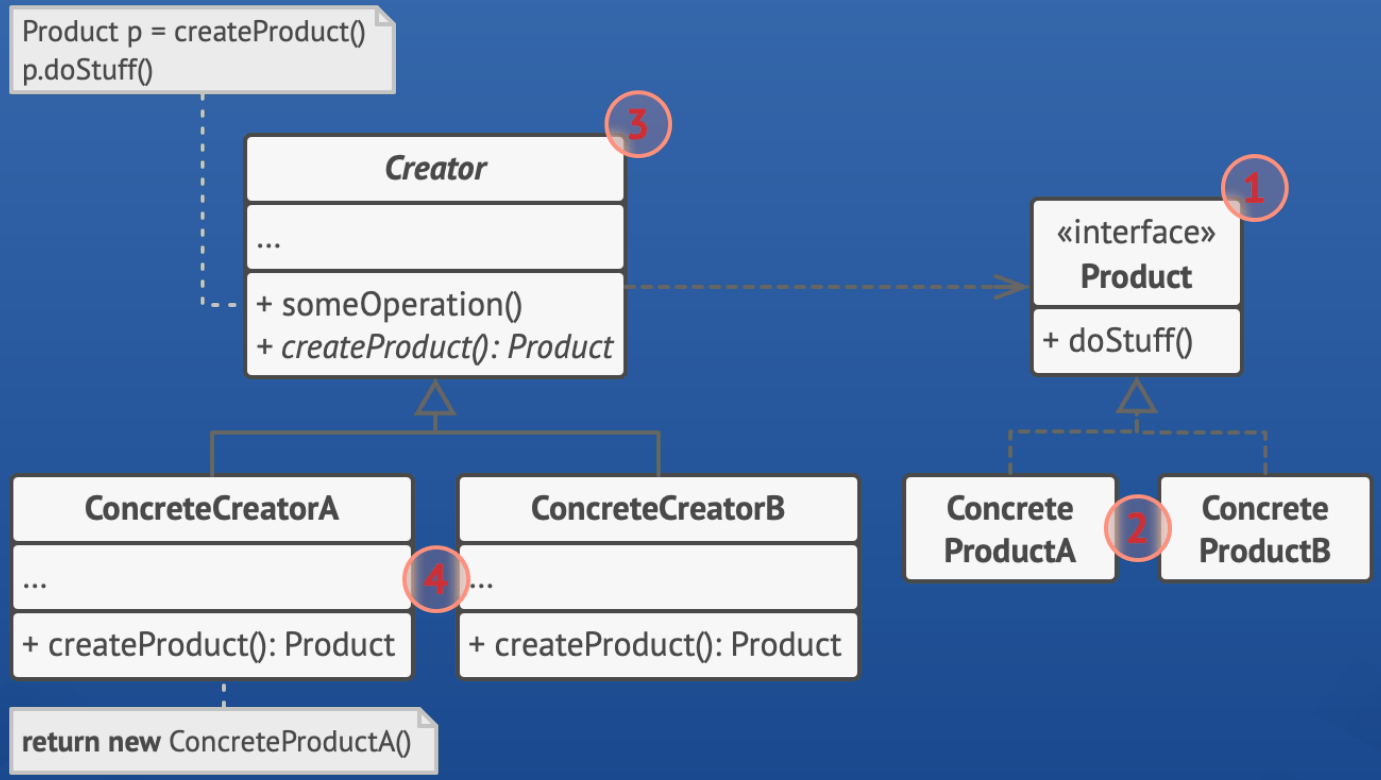
Factory Method

Example1 : Consider the case of a hiring manager. It is impossible for one person to interview for each of the positions. Based on the job opening, she has to decide and delegate the interview steps to different people.

Example2 : Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

Factory Method



Singleton

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

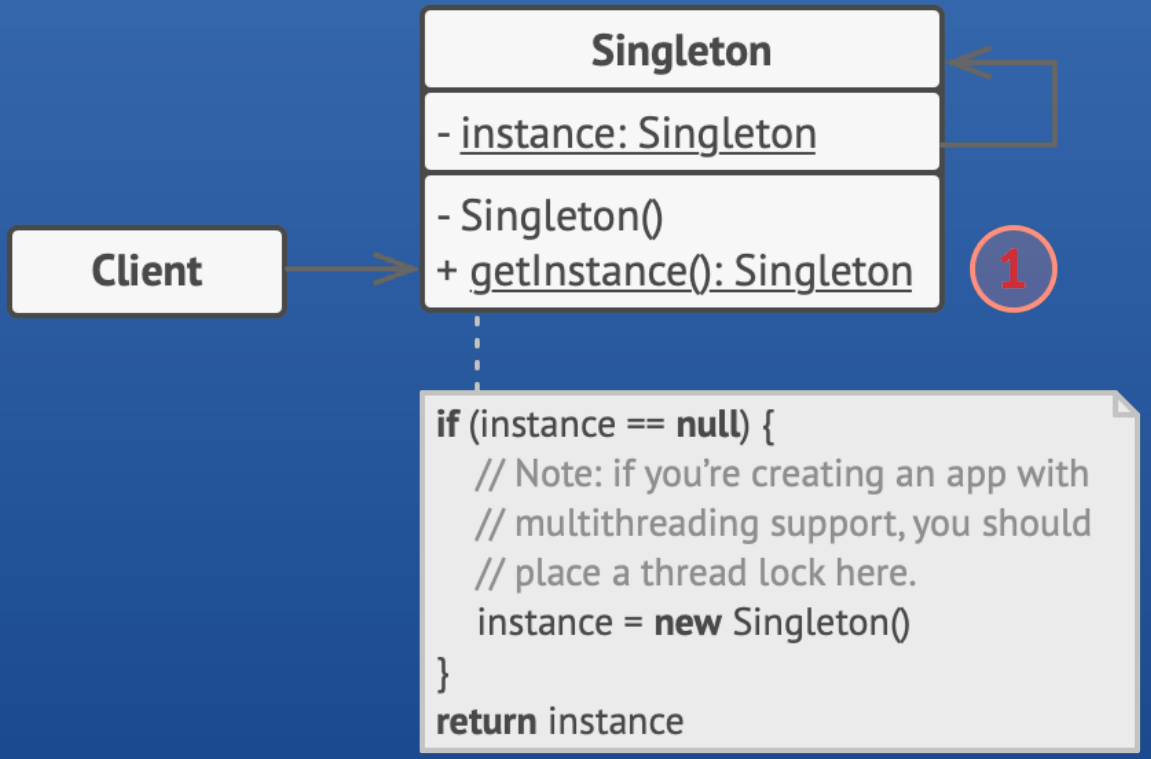
1- Ensure that a class has just a single instance.

2- Provide a global access point to that instance

Singleton

Singleton pattern is actually considered an anti-pattern and overuse of it should be avoided. It is not necessarily bad and could have some valid use-cases but should be used with caution because it introduces a global state in your application and change to it in one place could affect in the other areas and it could become pretty difficult to debug. The other bad thing about them is it makes your code tightly coupled plus mocking the singleton could be difficult.

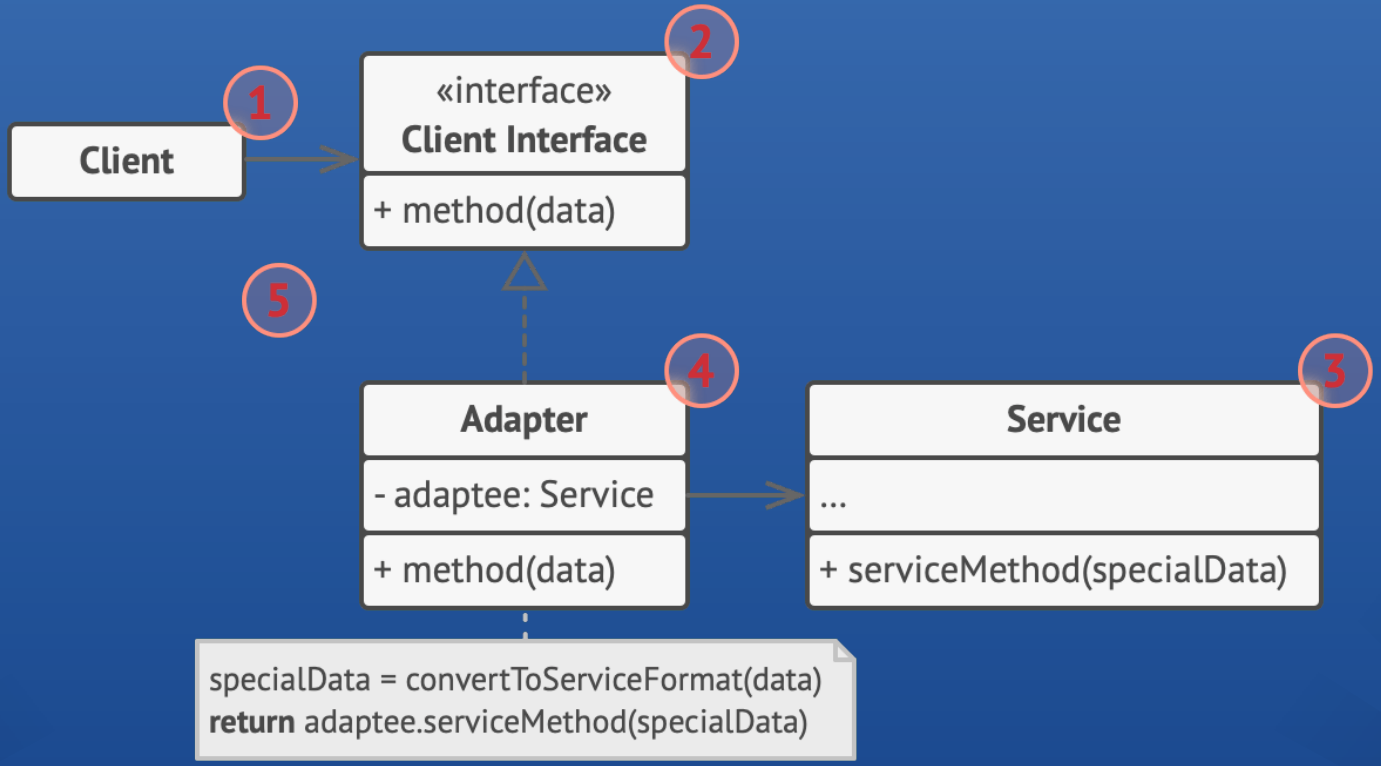
Singleton


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
1
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

Adapter

Adapter or Wrapper is a structural design pattern that allows objects with incompatible interfaces to collaborate.

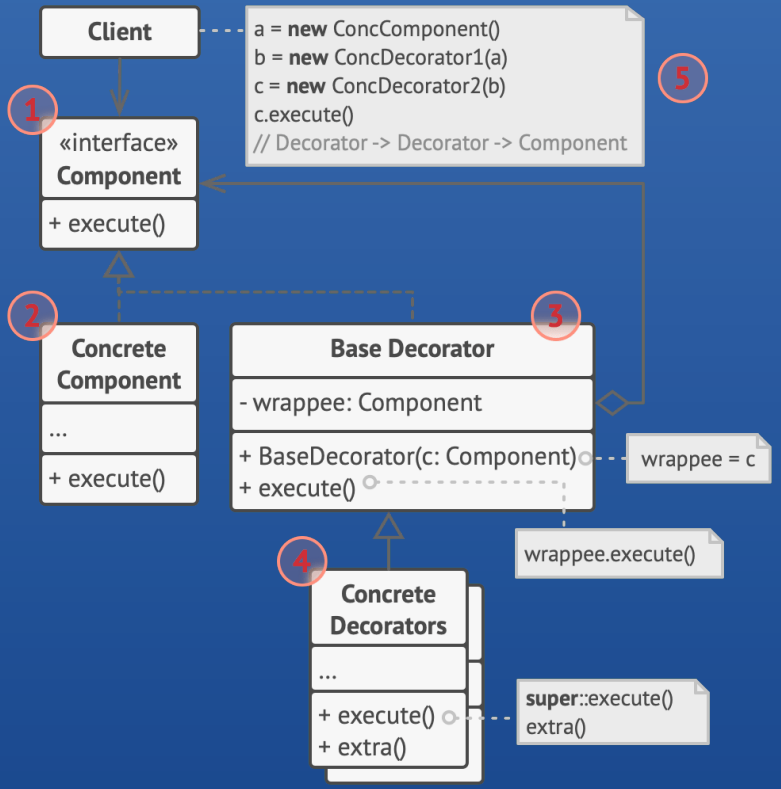
Adapter


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

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.

Decorator


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
5
1
«interface»
Component
+ execute()
2
Concrete
Component
...
+ execute()
3
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
4
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()

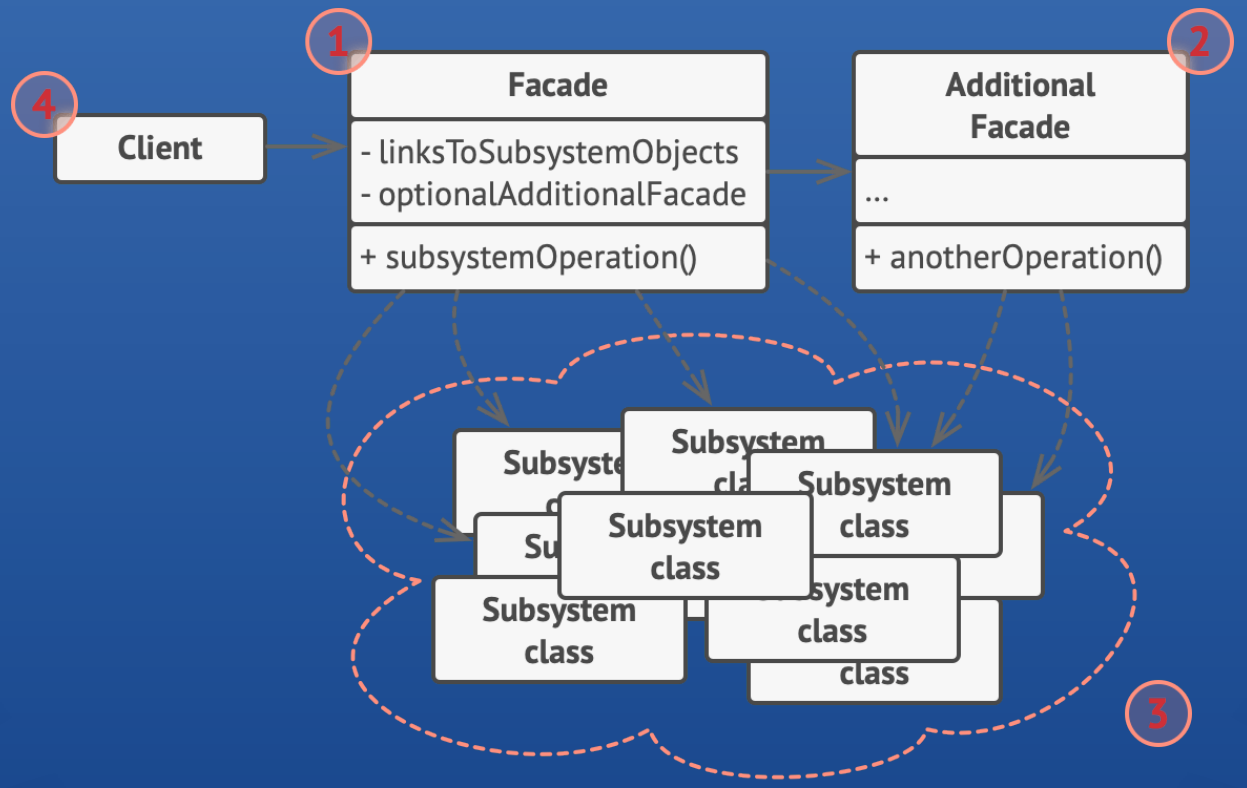
Facade

Facade pattern provides a simplified interface to a complex subsystem.

Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

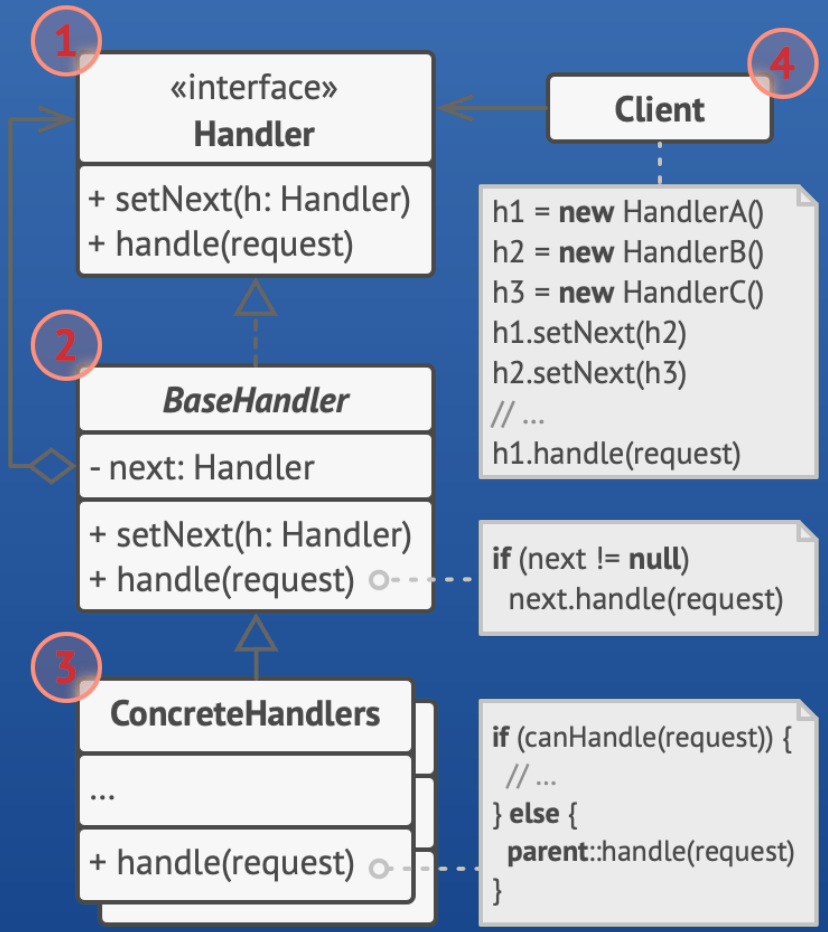
Facade


1
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
2
Additional Facade
...
+ anotherOperation()
4
Client
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class
3

Chain of Responsibility

Chain of Responsibility: is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Chain of Responsibility

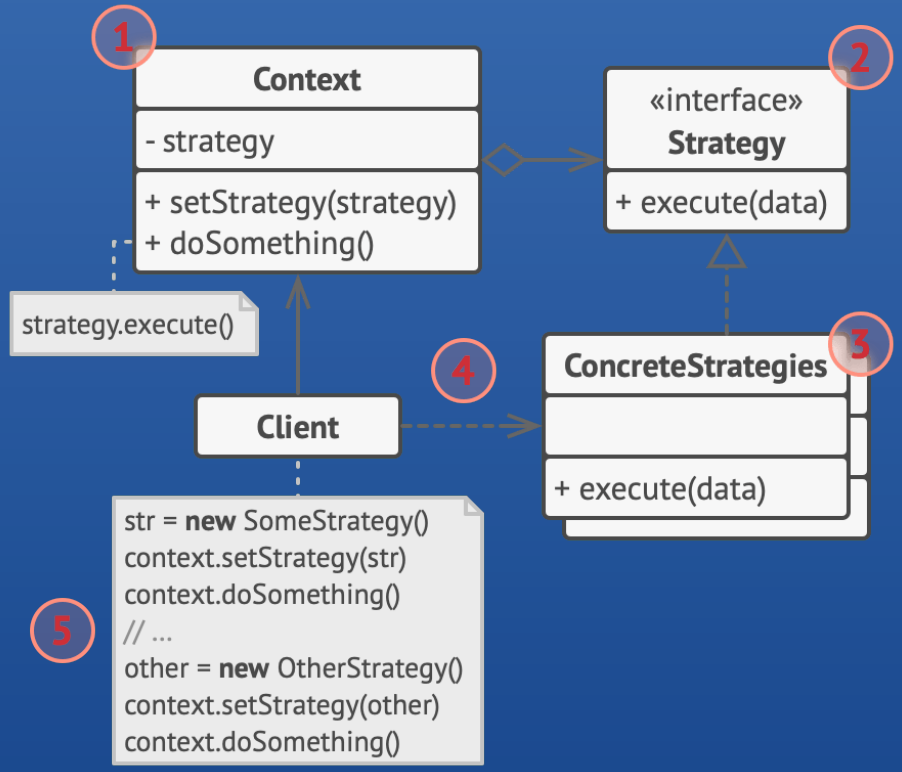


Strategy

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 pattern allows you to switch the algorithm or strategy based upon the situation.

Strategy

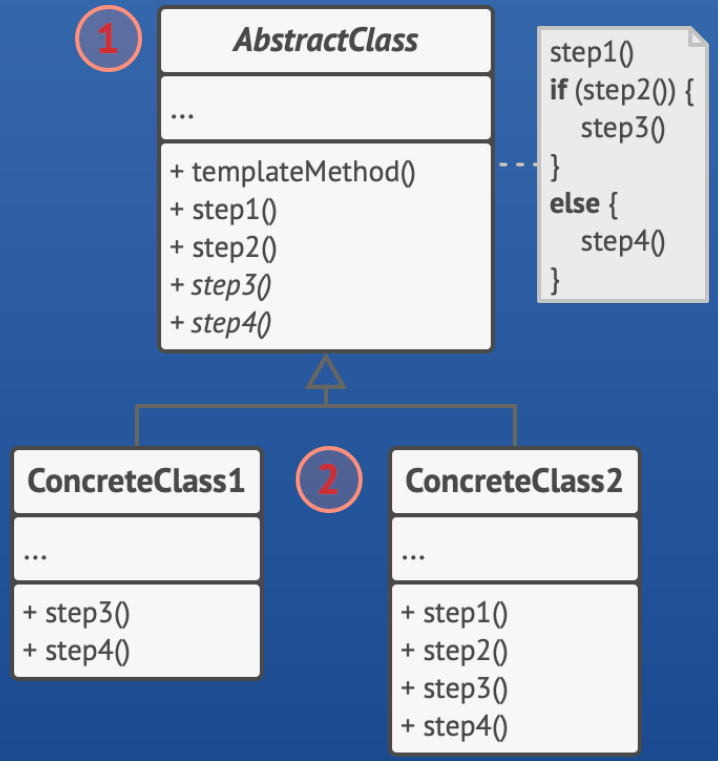

1
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
2
«interface»
Strategy
+ execute(data)
strategy.execute()
4
Client
3
ConcreteStrategies
+ execute(data)
5
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

Template method

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the super class but lets sub classes override specific steps of the algorithm without changing its structure.

Template method defines the skeleton of how a certain algorithm could be performed, but defers the implementation of those steps to the children classes.

Template method

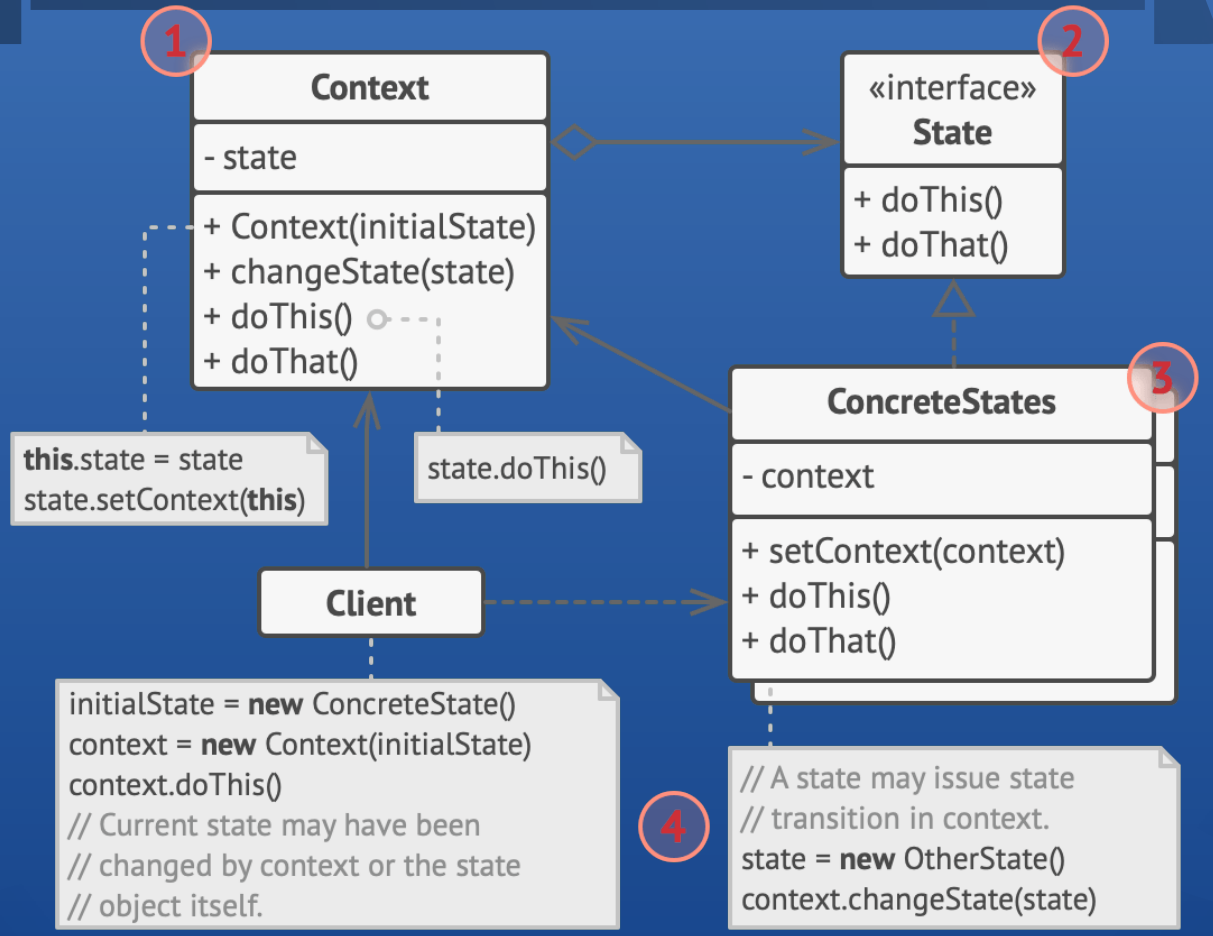


State

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

It lets you change the behavior of a class when the state changes.

State


1
Context
- state
+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()
2
«interface»
State
+ doThis()
+ doThat()
3
ConcreteStates
- context
+ setContext(context)
+ doThis()
+ doThat()
this.state = state
state.setContext(this)
state.doThis()
Client
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
4
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)