

Design Patterns

Design patterns are Structured by three Categories :

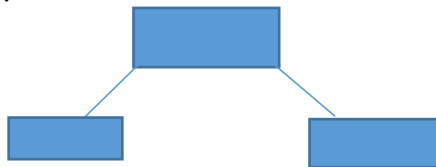
- 1) Creational Design Patterns
- 2) Structural Design Patterns
- 3) Behavioral Design Patterns

1) Creational Design Patterns

Creational Patterns Provide Various object Creation mechanism , Which increase flexibility and reuse of existing Code.

Creational Design Patterns are :

A) Factory Method



Provides an Interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass , but allows subclasses to alter the type of objects that will be created .

B) Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

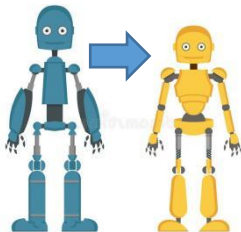


C) Builder



Lets you construct complex objects step by Step. The Pattern allows you to produce different types of an object using the same construction code.

D) Prototype



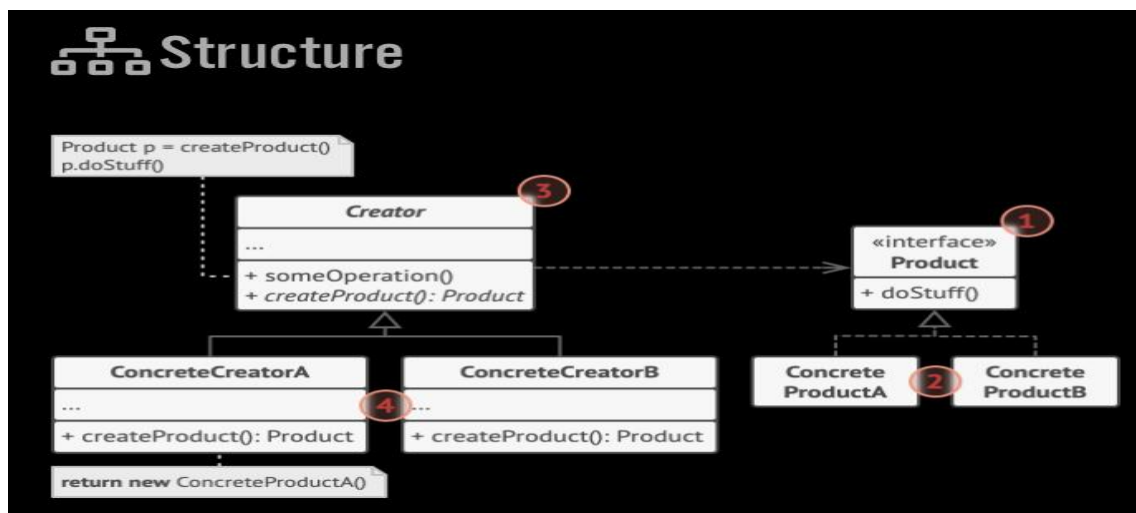
Let's you copy existing objects without making your code dependent on their classes.

E) Singleton



Let's ensure that a class has already one instance, while providing a global access point to this instance.

A) Factory Method ==> Structure With Pros & Cons



Pros & Cons:

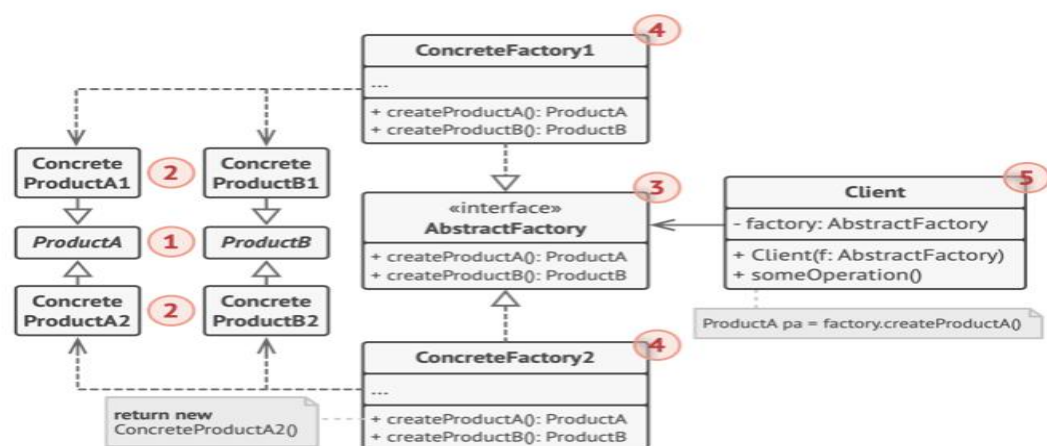
- ✚ a) You Avoid tight coupling between the creator and the concrete products.
- ✚ b) *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✚ c) *Open/Closed Principle*. You can introduce new types of products into the program without breaking the existing client code.

✘ The Code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The Best case Scenario is when you're introducing the pattern into an existing hierarchy of creator classes.





B) Abstract Factory ==>> Structure With Pros & Cons

Abstract Factory is a creational Design Pattern that lets you produce families of related objects without specifying their classes.

Structure

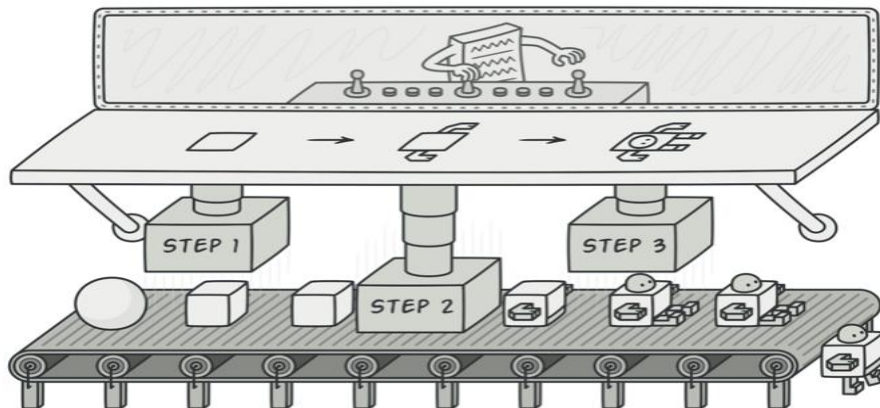


Pros & Cons:

-  a) You can be sure that the products you're getting from a factory are compatible with each other
-  b) You avoid tight coupling between concrete products and client code.
-  c) *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
-  d) *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

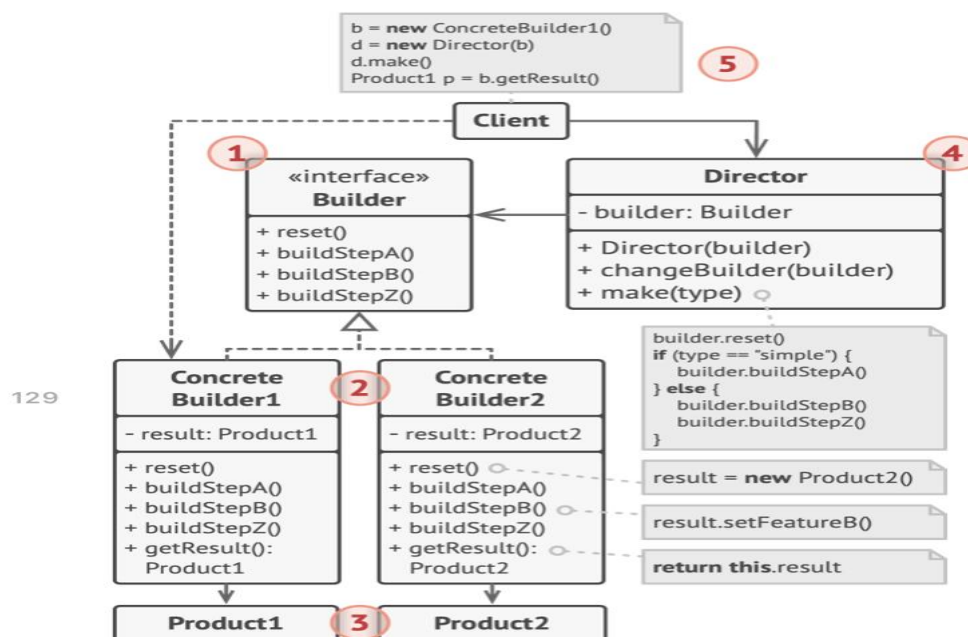
X: The Code may become more complicated that it should be, since a lot of new interfaces and classes are introduced along with the pattern.

C) Builder ==> Structure With Pros & Cons



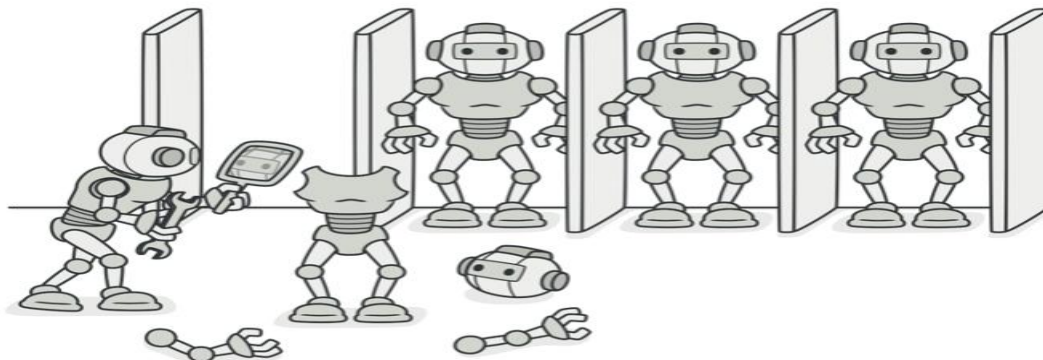
Builder is a creational Design Pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and presentation of an object using the same construction code.

Structure



Pros & Cons:

- + a) You can construct objects Step by Step defer construction steps or run steps recursively.
- + b) You can reuse the same construction code when building various representations of products.
- + c) *Single Responsible Principle*. You can isolate complex construction code from the business logic of the product.
- X) The overall complexity of the code increases since the pattern requires creating multiple new classes.



D) PROTOTYPE

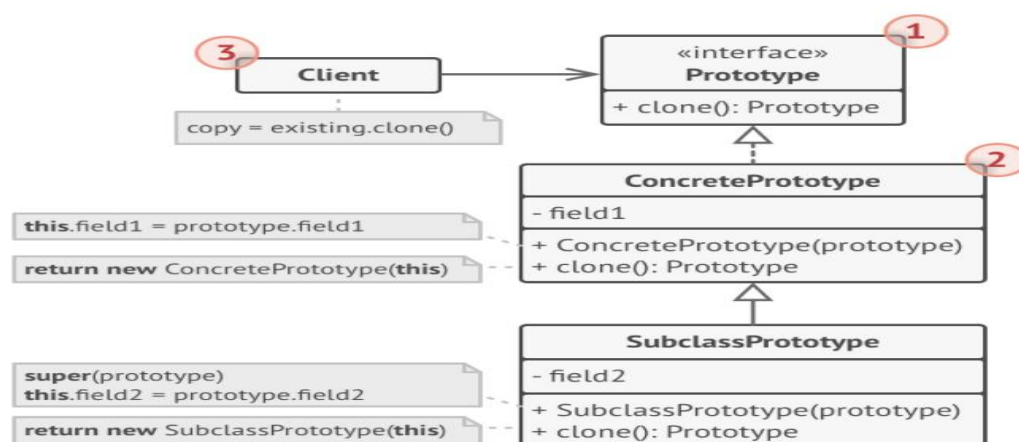
Also known as: Clone

=> Structure With Pros & Cons

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Structure

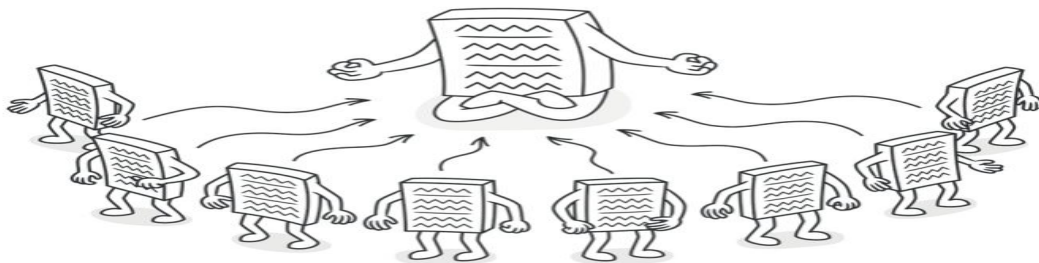
Basic implementation



Pros & Cons:

- ✚ a) You can clone objects without coupling to their concrete classes.
- ✚ b) You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- ✚ c) You can produce complex objects more conveniently.
- ✚ d) You get an alternative to inheritance when dealing with configuration presets for complex objects.

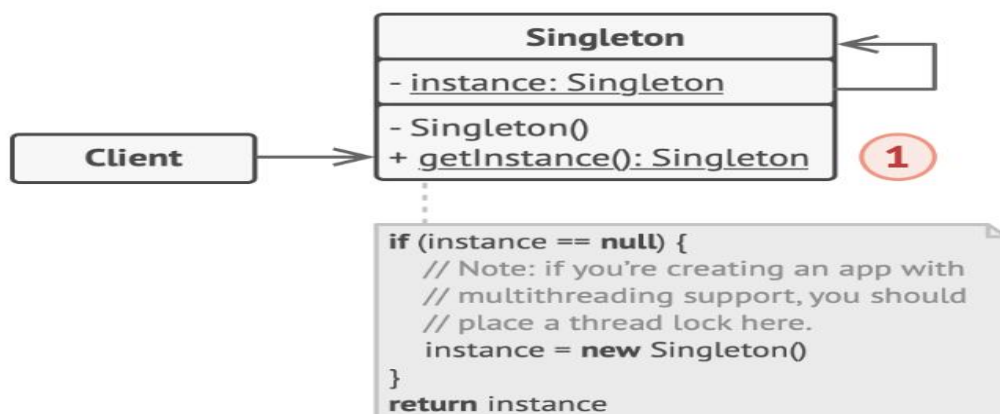
✗) Cloning complex objects that have circular references might be very tricky.






E) SINGLETON ==> Structure With Pros & Cons


Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.


Structure





Pros & Cons:

-  a) You can be sure that a class has only a single instance.
-  b) You gain a global access point to that instance.
-  c) The singleton object is initialized only when it's requested for the first time.

 Violates the Single Responsibility Principle. The pattern solves two problems at the time.

 The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.

 The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.

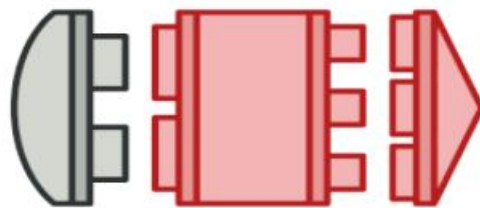
 It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

2) Structural Design Patterns

Structural patterns explain how to assemble objects and classes into larger structures, while keeping this structures flexible and efficient.

Structural Design Patterns are Dived as Follow:

A) Adapter



Allows objects With incompatible interfaces to collaborate.

B) Bridge



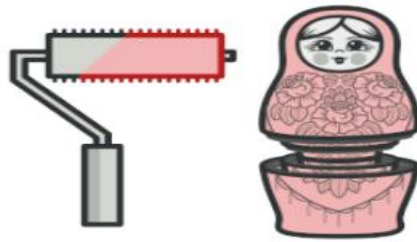
Lets you Split a Large class or a set of closely related classes into two separates hierarchies-abstraction and implementation which can be developed independently of each other.

C) Composite

Lets you compose objects into tree structure and then work with these structures as if they were individual objects.



D) Decorator



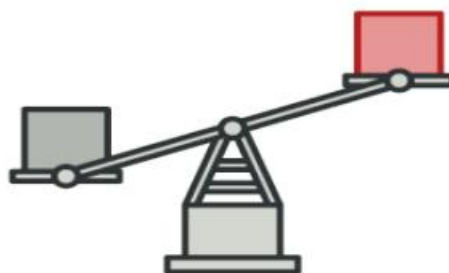
Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

E) Facade



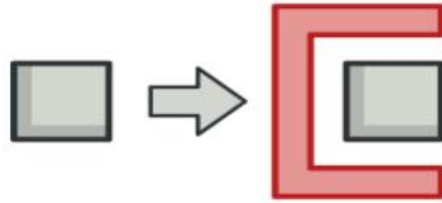
Provides a simplified interface to a library, a framework, or any other complex set of classes.

F) Fly-Weight



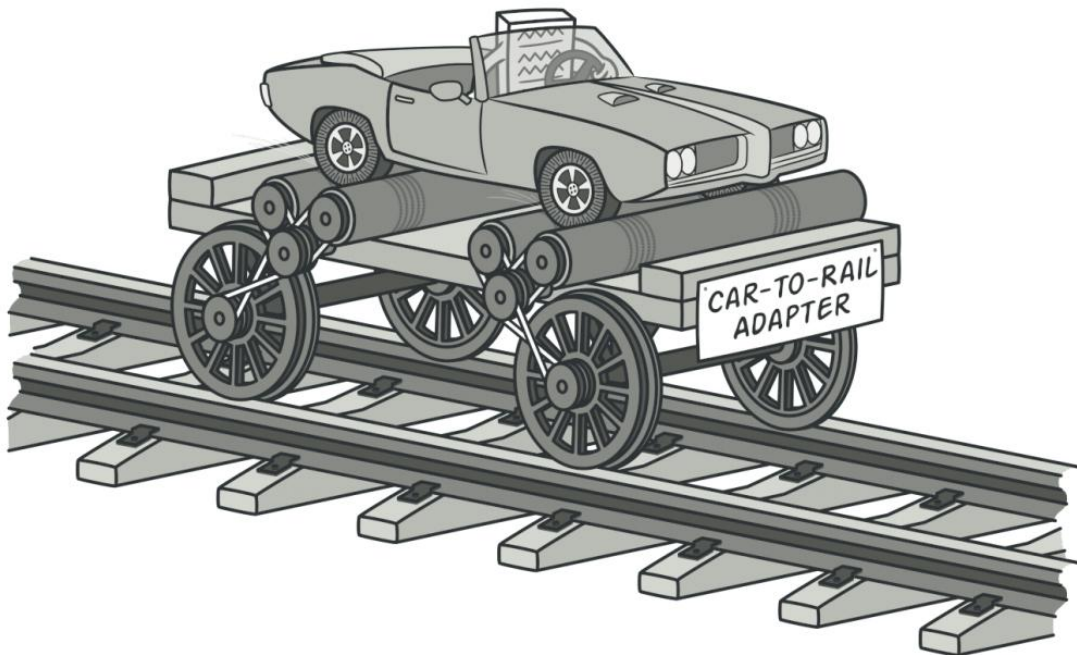
Lets you fit more objects into the available amount of RAM by Sharing common parts of state between multiple objects instead of keeping all of the data in each object.

G) Proxy



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.

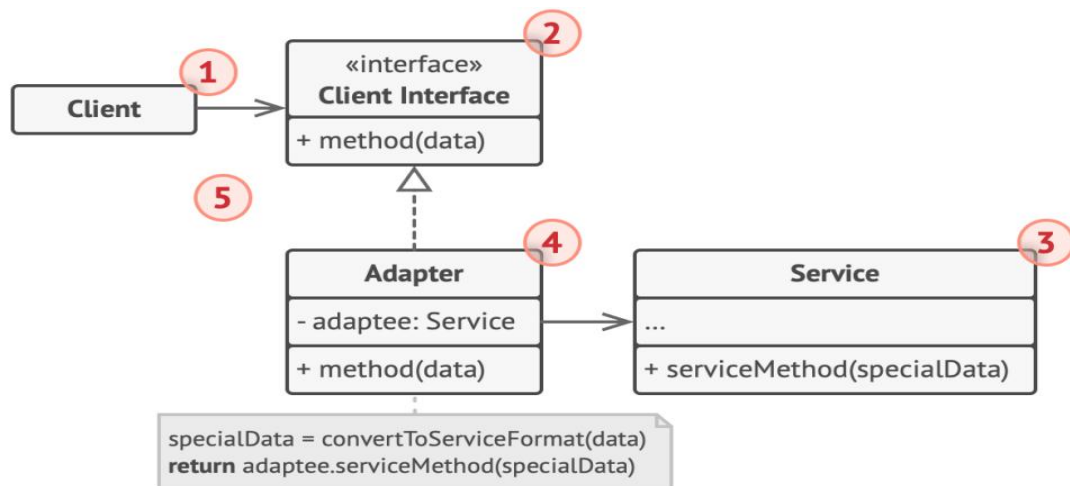
Structure of each of the pattern with Pros and Cons for the usability :



A) ADAPTER

Also known as: Wrapper

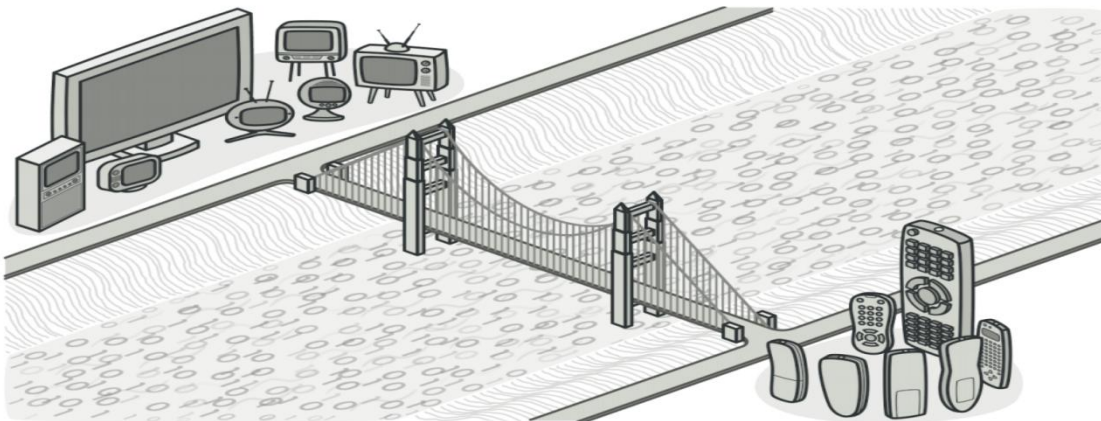
Structure:



Pros & Cons:

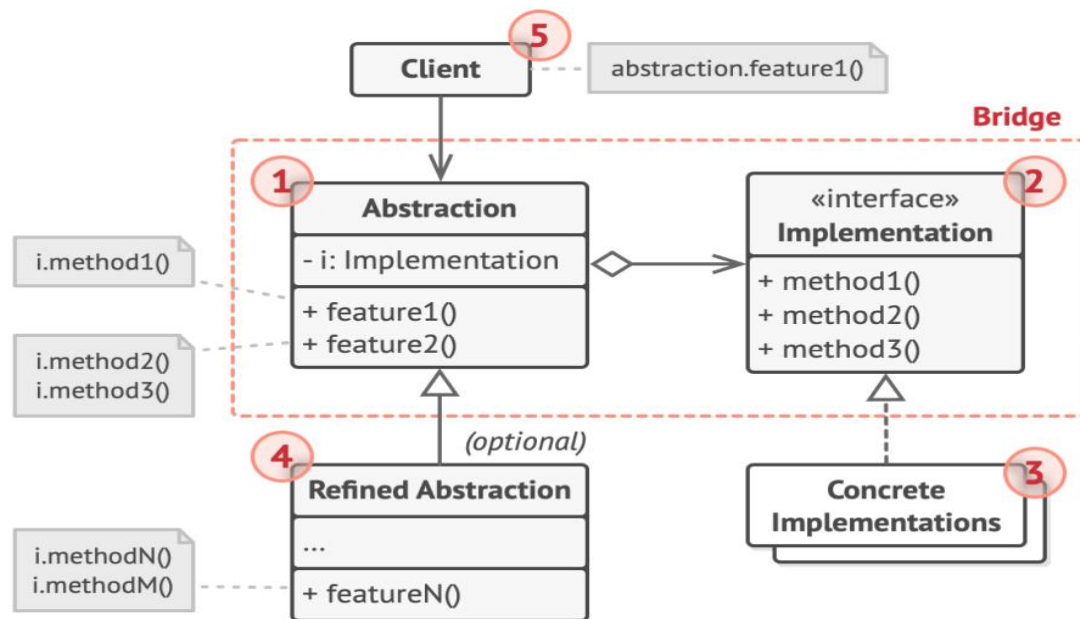
- ✚ a) Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.
- ✚ b) Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the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



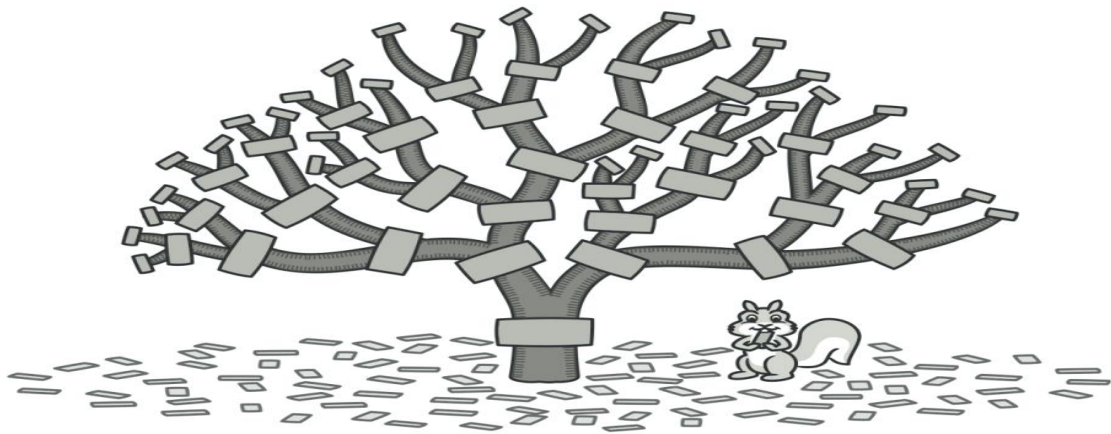
B) BRIDGE

Structure:



Pros & Cons:

- ☑ a) You can create platform-independent classes and apps.
 - ☑ b) The client code works with high-level abstractions. It isn't exposed to the platform details.
 - ☑ c) Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
 - ☑ d) Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.
- ✗ You might make the code more complicated by applying the pattern to a highly cohesive class.

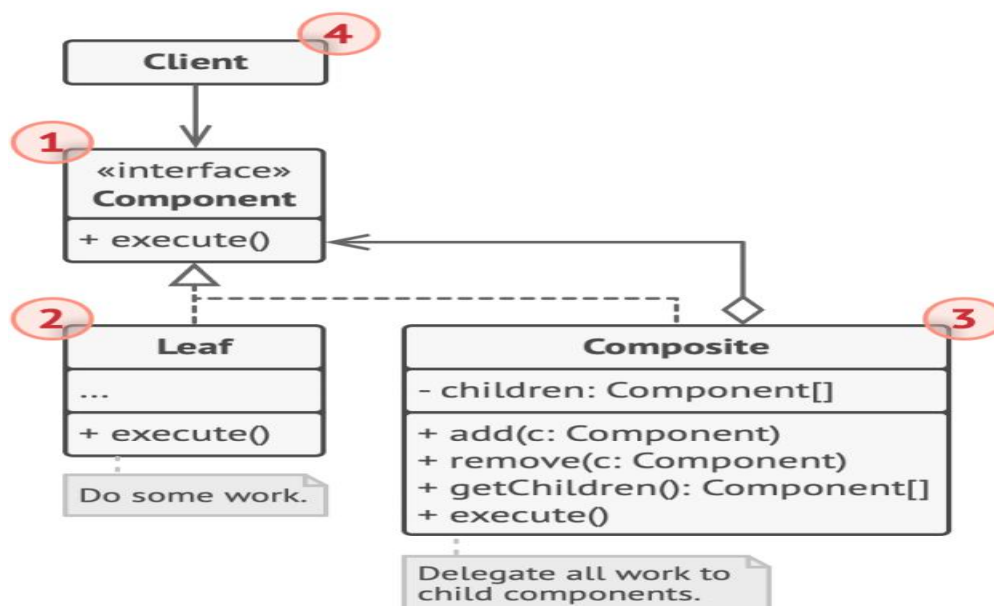


C) COMPOSITE

Also known as: Object Tree

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

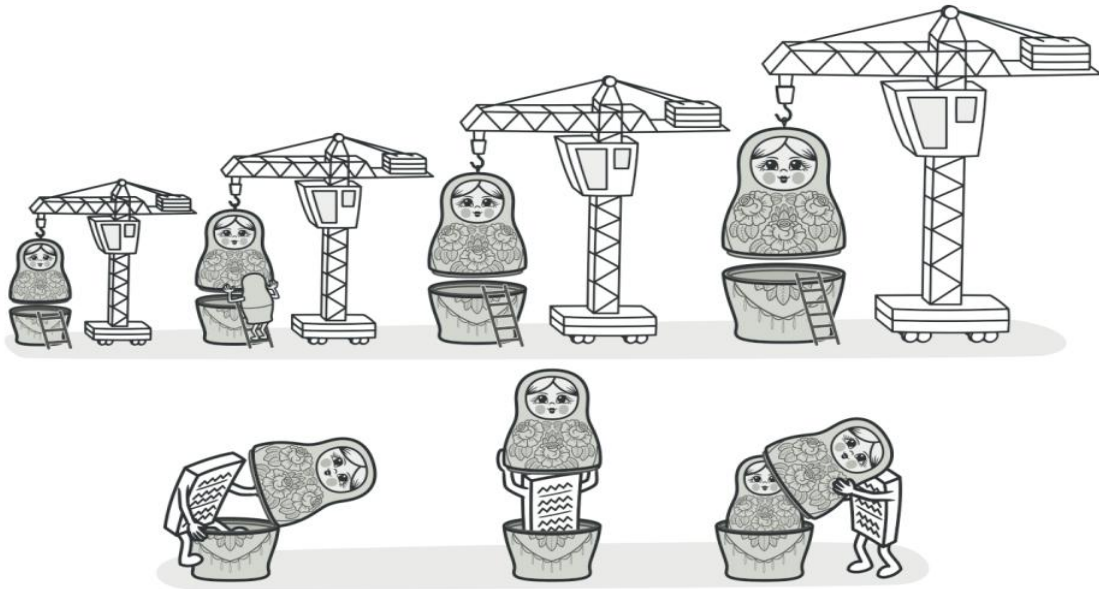
Structure:



Pros & Cons:

- ✚ a) You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✚ b) Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

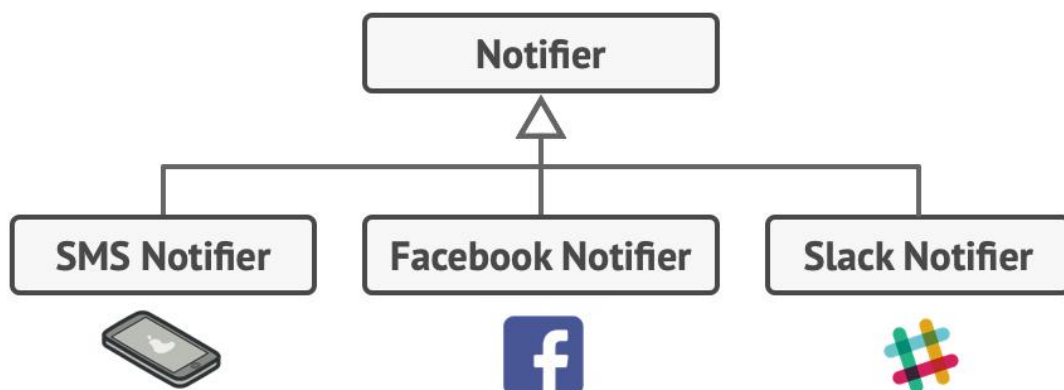
✗) It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.



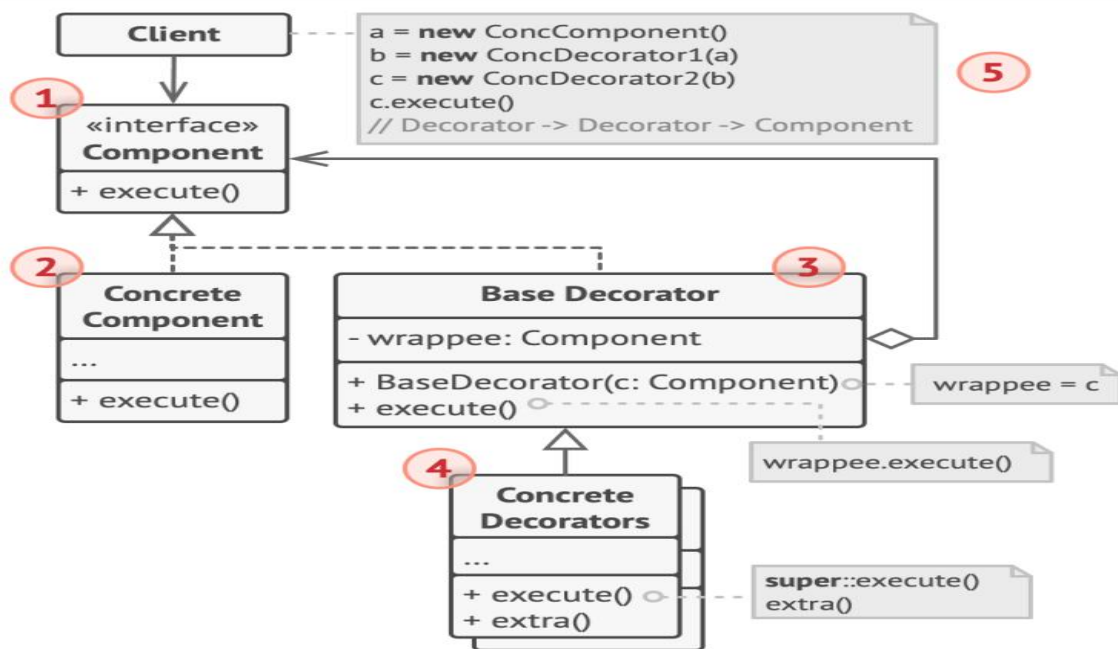
D) DECORATOR

Also known as: Wrapper

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.



Structure:



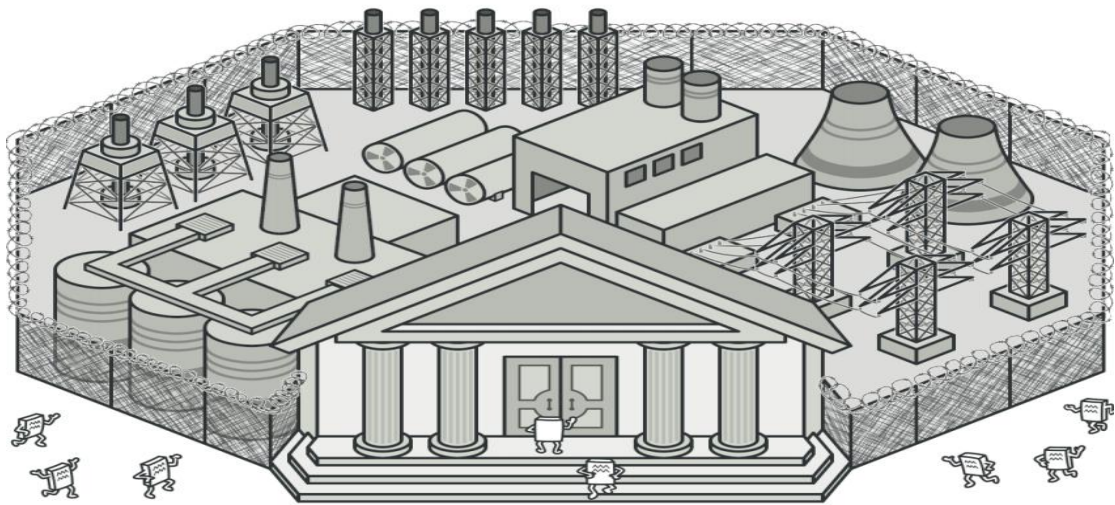
Pros & Cons:

- + a) You can extend an object's behavior without making a new subclass.
- + b) You can add or remove responsibilities from an object at runtime.
- + c) You can combine several behaviors by wrapping an object into multiple decorators.
- + d) Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

X) It's hard to remove a specific wrapper from the wrappers stack.

X) It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.

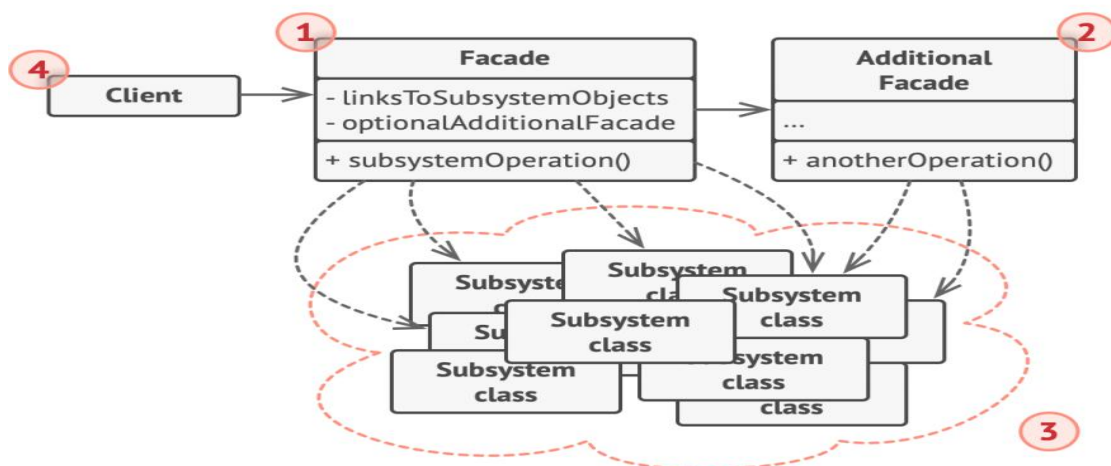
X) The initial configuration code of layers might look pretty ugly.



E) FACADE

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

Structure:



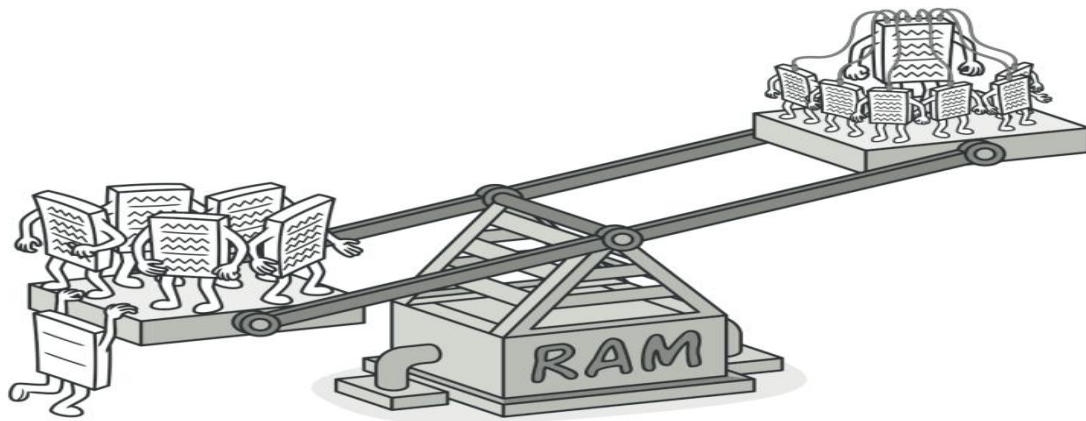
Pros & Cons:



You can isolate your code from the complexity of a subsystem.



A facade can become a god object coupled to all classes of an app.

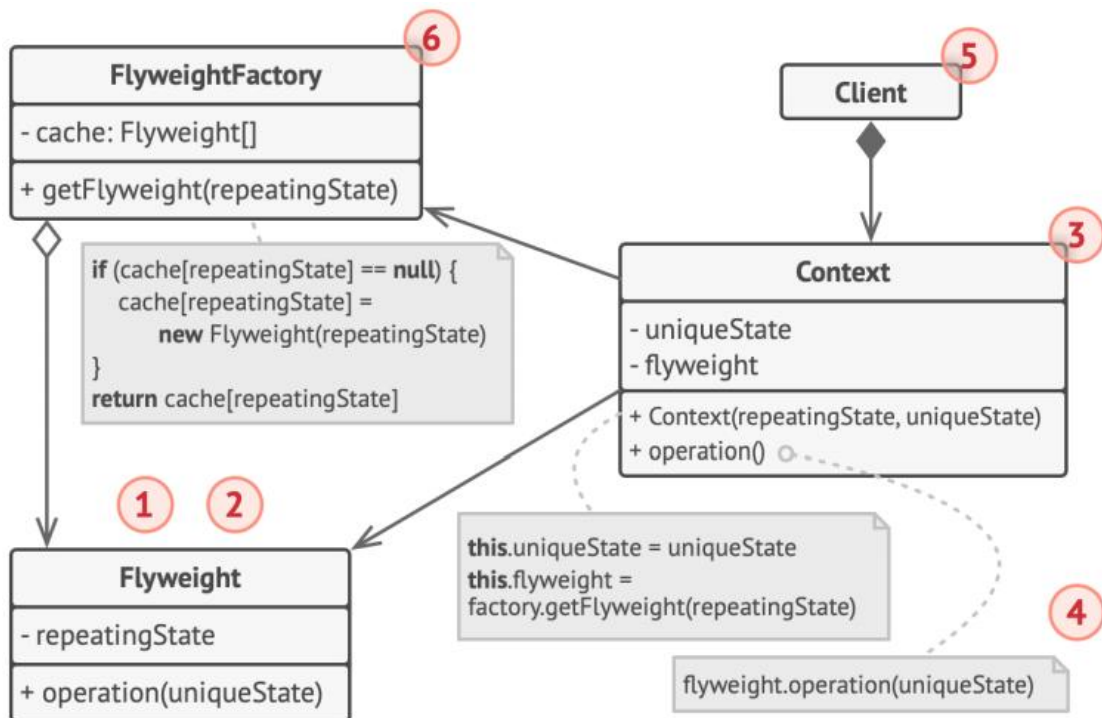


F) FLYWEIGHT

Also known as: Cache

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.

Structure:

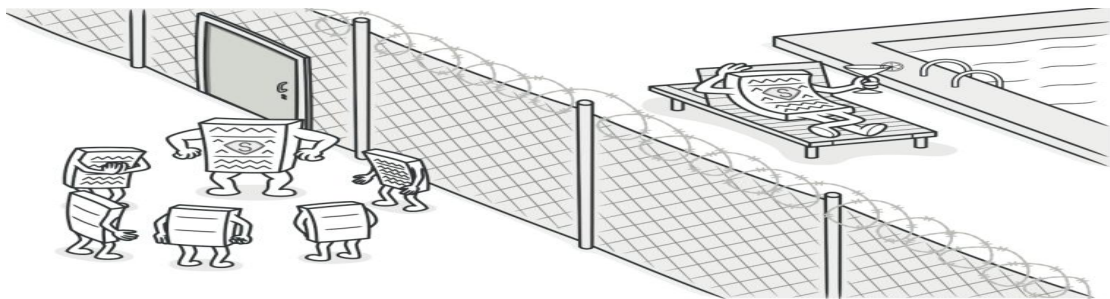


Pros & Cons:

✚ You can save lots of RAM, assuming your program has tons of similar objects.

✗ You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.

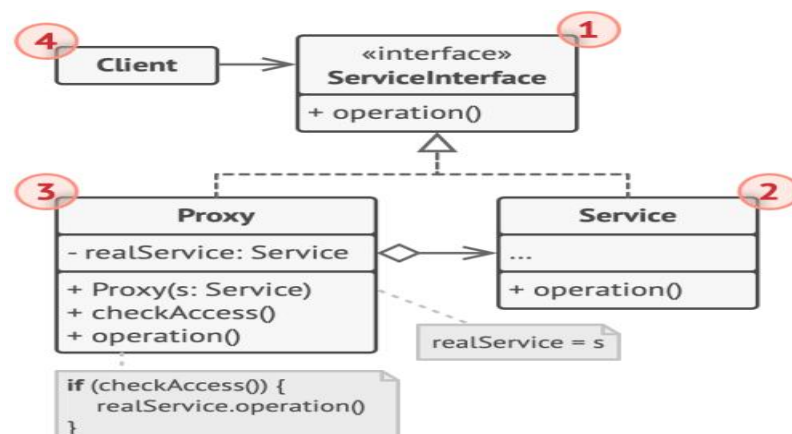
✗ The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.



6) 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.

Structure:



Pros & Cons:

- ✚ a) You can control the service object without clients knowing about it.
- ✚ b) You can manage the lifecycle of the service object when clients don't care about it.
- ✚ c) The proxy works even if the service object isn't ready or is not available.
- ✚ d) Open/Closed Principle. You can introduce new proxies without changing the service or clients.

✗) The code may become more complicated since you need to introduce a lot of new classes.

✗) The response from the service might get delayed.

3) Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral patterns are Divided as Follow:



A) Chain of Responsibility

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 handler in the chain .

B)Command



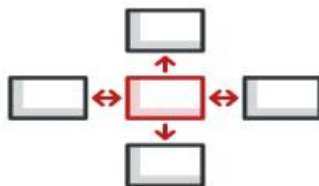
Turns a request into a stand alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

C)Iterator



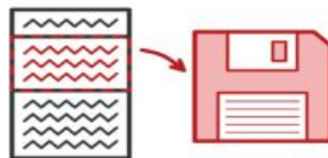
Lets you traverse elements of a collection without exposing its underlying representation (list, stack, Tree, etc).

D)Mediator



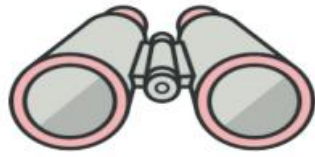
Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

E) Memento

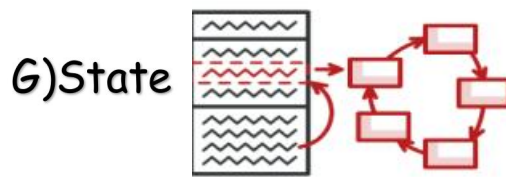


Lets you save and restore the previous state of an object without revealing the details of its implementation.

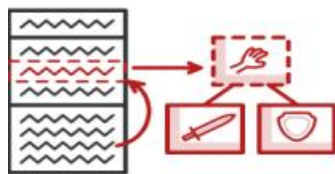
F) Observer



Lets you Define a subscription mechanism to notify multiple multiple objects about any events that happen to the object they're observing.



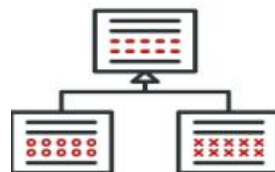
Lets an object alter its behavior when its internal state change. It appears as if the object changed its class.



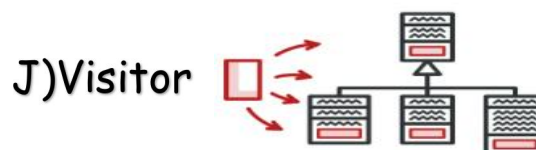
H) Strategy

Lets you Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

I)Template Method

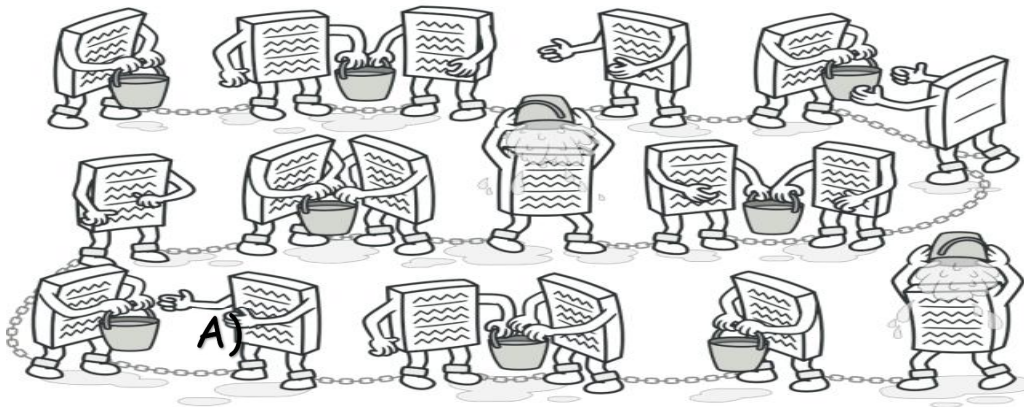


Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing it's structure.



Lets you separate algorithms from the objects on which operate.

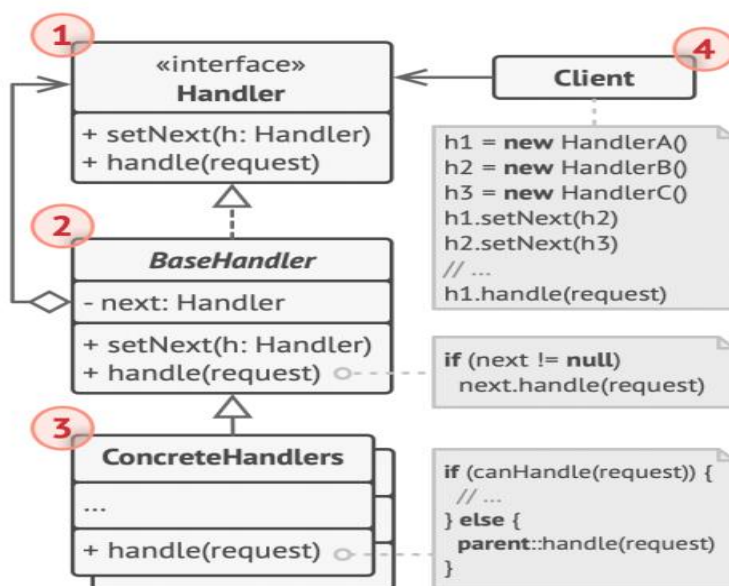
Structure of each of the pattern with Pros and Cons for the usability :



CHAIN OF RESPONSIBILITY

Also known as: CoR, Chain of Command

Structure:



Pros & Cons:

- ✚ a) You can control the order of request handling.
- ✚ b) *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform operations.
- ✚ c) *Open/Closed Principle*. You can introduce new handlers into the app without breaking the existing client code.

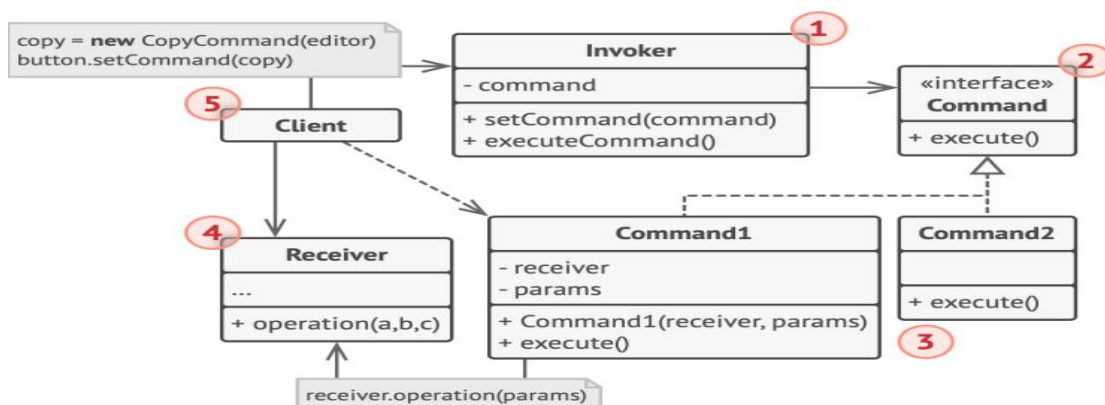
✗) Some requests may end up unhandled.



B) COMMAND

Also known as: Action, Transaction

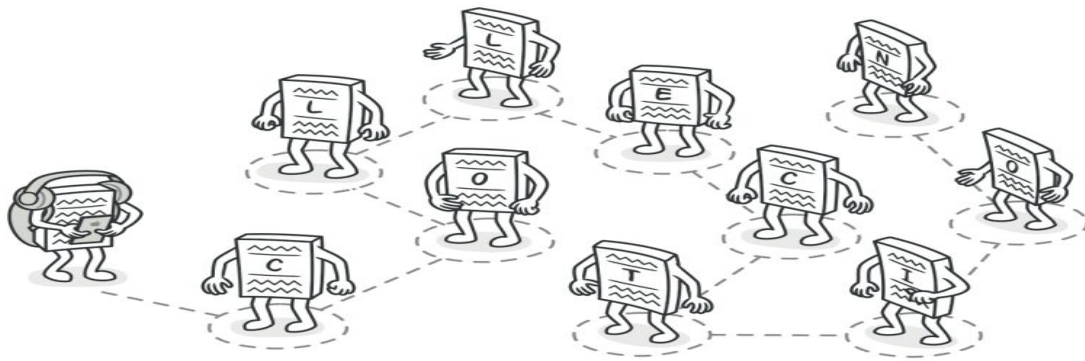
Structure:



Pros & Cons:

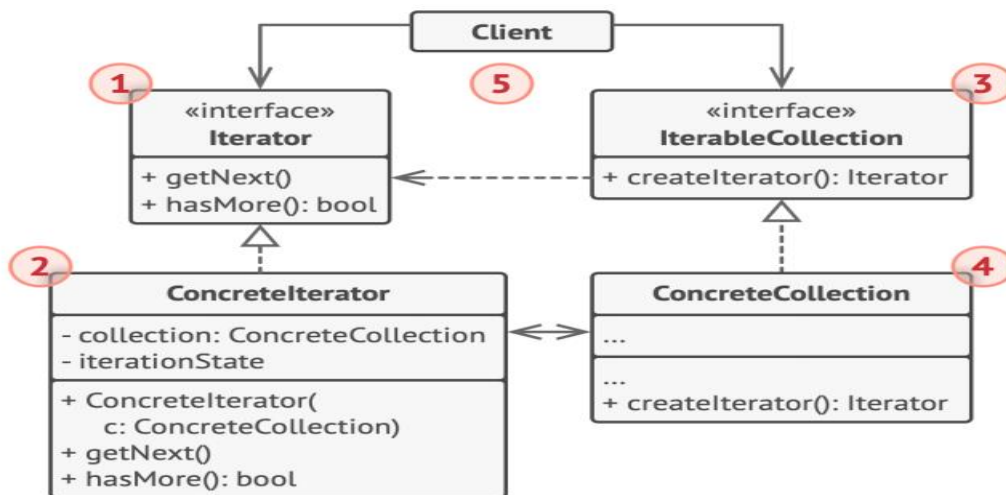
- ✚ a) Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations.
- ✚ b) Open/Closed Principle. You can introduce new commands into the app without breaking existing client code.
- ✚ c) You can implement undo/redo.
- ✚ d) You can implement deferred execution of operations.
- ✚ e) You can assemble a set of simple commands into a complex one.

✗) The code may become more complicated since you're introducing a whole new layer between senders and receivers.



C) ITERATOR

Structure:

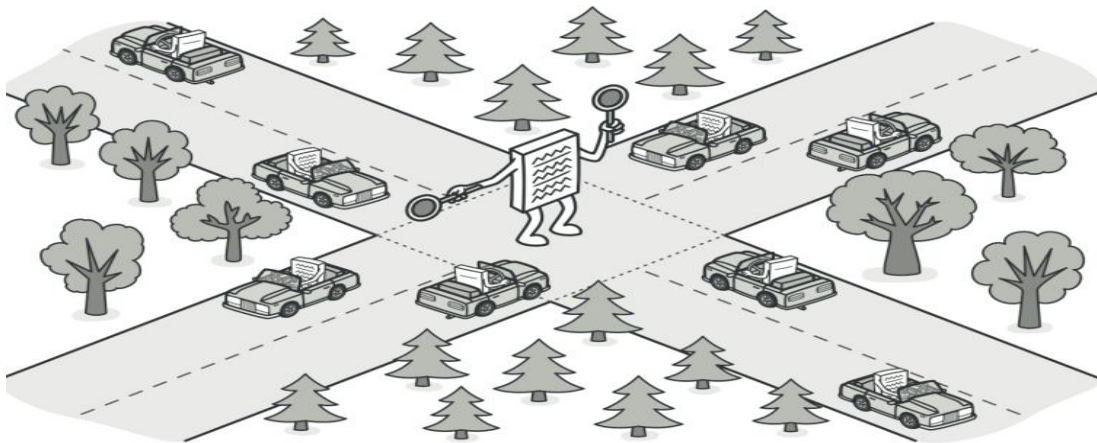


Pros & Cons:

- ✚ a) Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- ✚ b) Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.
- ✚ c) You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- ✚ d) For the same reason, you can delay an iteration and continue it when needed.

X)Applying the pattern can be an overkill if your app only works with simple collections.

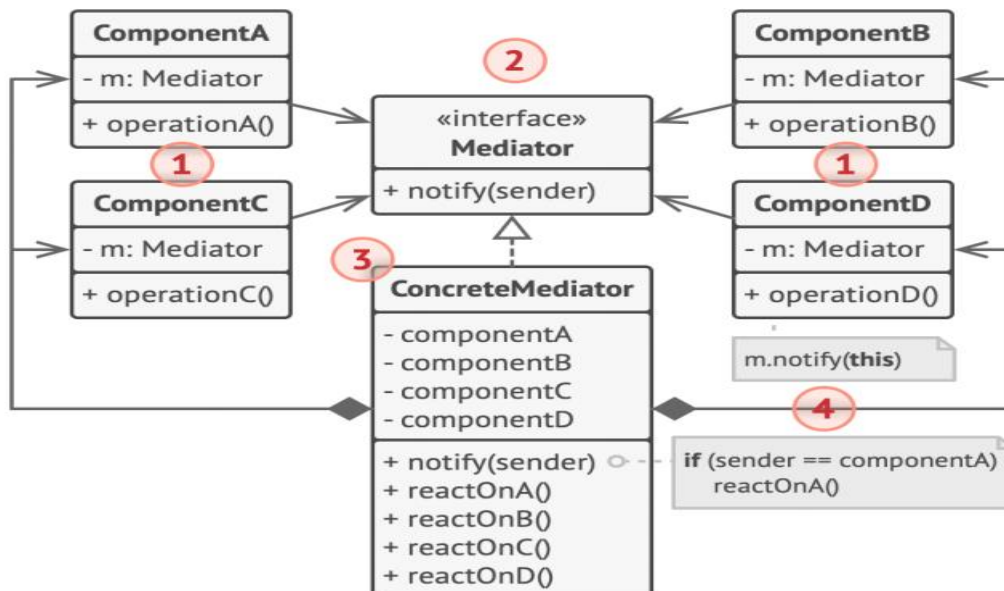
X)Using an iterator may be less efficient than going through elements of some specialized collections directly.



D) MEDIATOR

Also known as: Intermediary, Controller

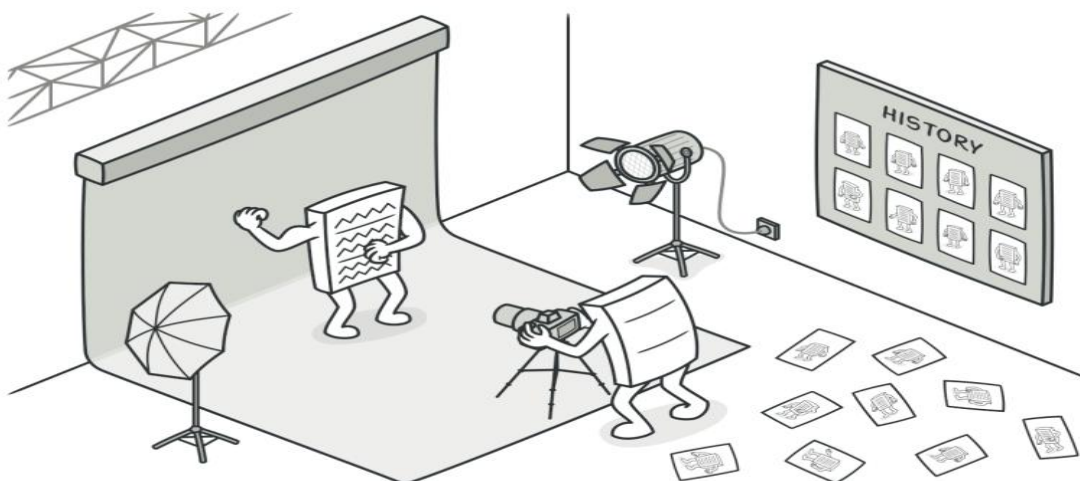
Structure:



Pros & Cons:

- ✚ a) Single Responsibility Principle. You can extract the communications between various components into a single place, making it easier to comprehend and maintain.
- ✚ b) Open/Closed Principle. You can introduce new mediators without having to change the actual components.
- ✚ c) You can reduce coupling between various components of a program.
- ✚ d) You can reuse individual components more easily.

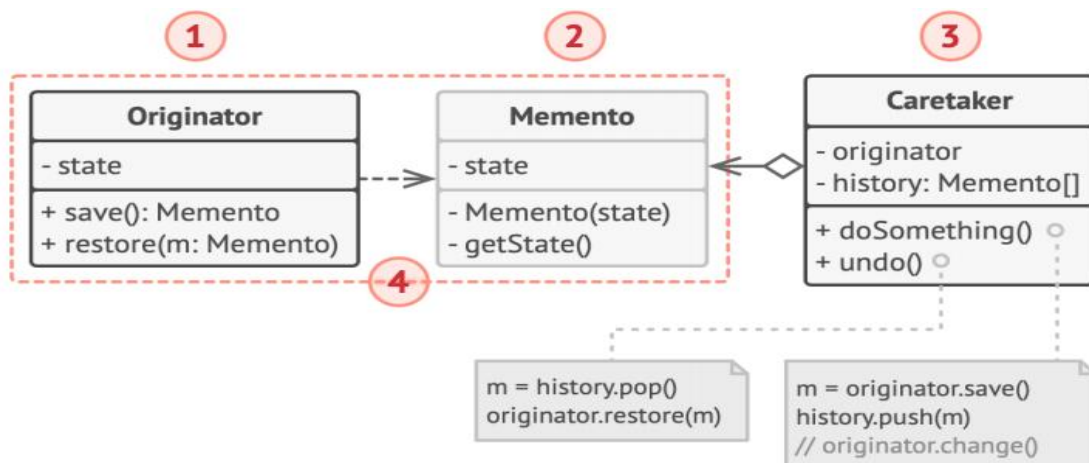
✗) Over time a mediator can evolve into a God Object.



E) MEMENTO

Also known as: Snapshot

Structure:



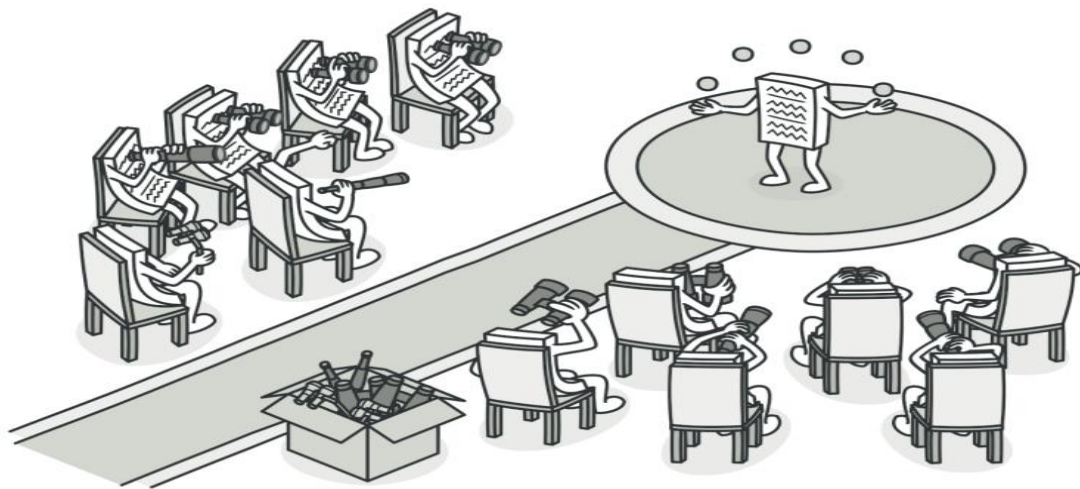
Pros & Cons :

- + a) You can produce snapshots of the object's state without violating its encapsulation.
- + b) You can simplify the originator's code by letting the caretaker maintain the history of the originator's state.

✗) The app might consume lots of RAM if clients create mementos too often.

✗) Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.

✗) Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays untouched.

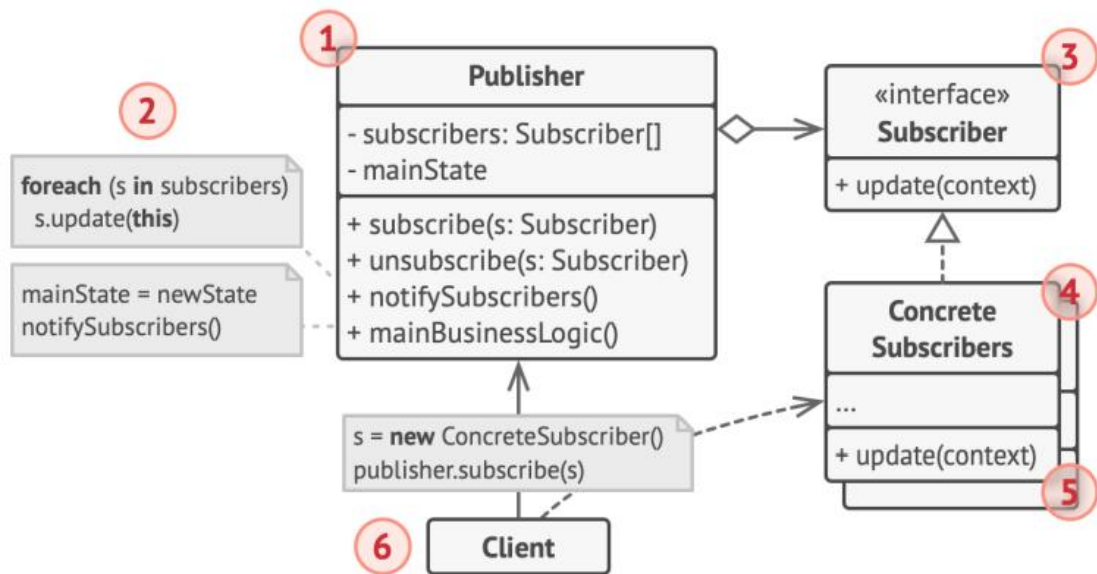


F) OBSERVER

Also known as: Event-Subscriber, Listener

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.

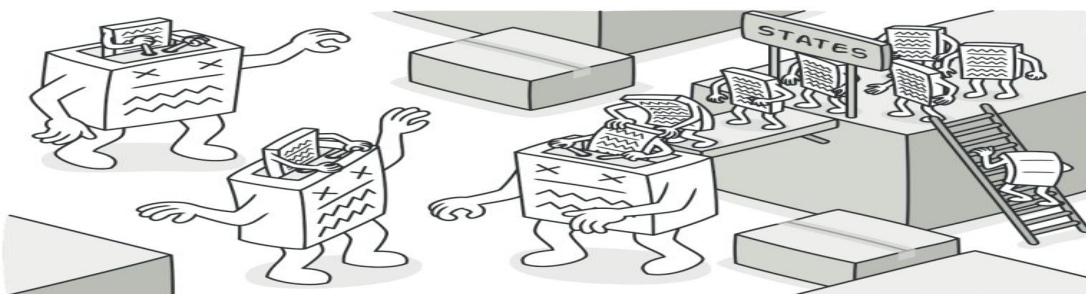
Structure:



Pros & Cons :

- ✚ a) Open/Closed Principle. You can introduce new subscriber classes with-out having to change the publisher's code (and vice versa if there's a publisher interface).
- ✚ b) You can establish relations between objects at runtime.

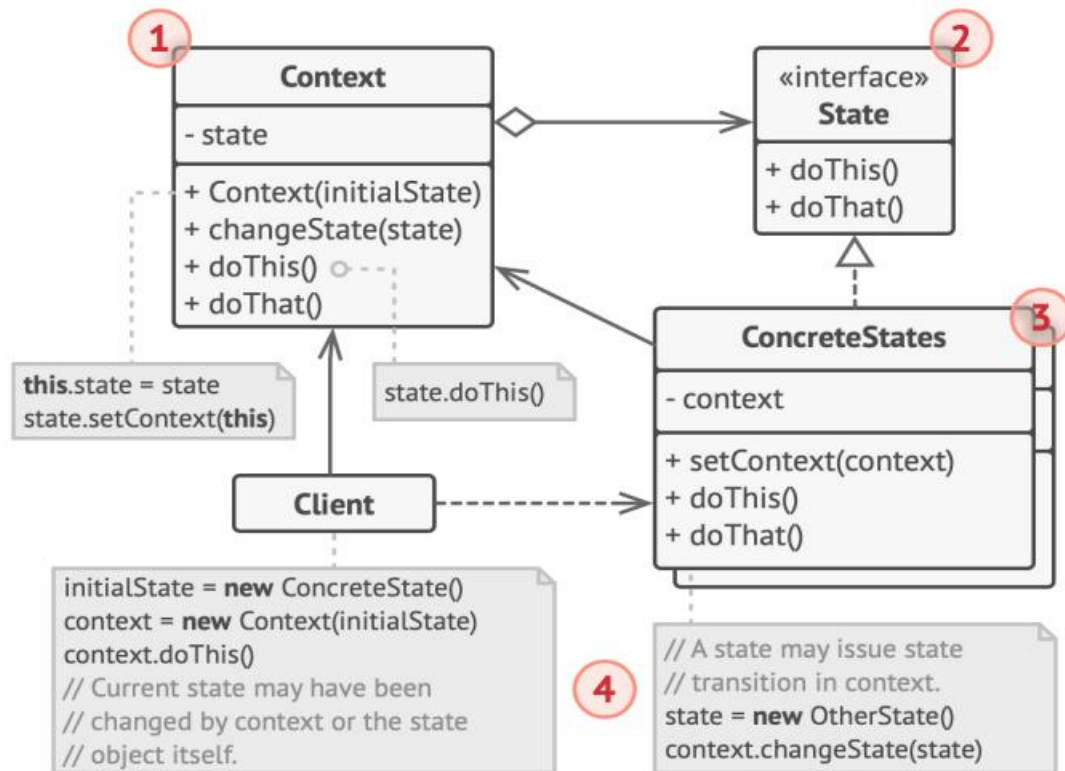
✗) Subscribers are notified in random order.



G) 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.

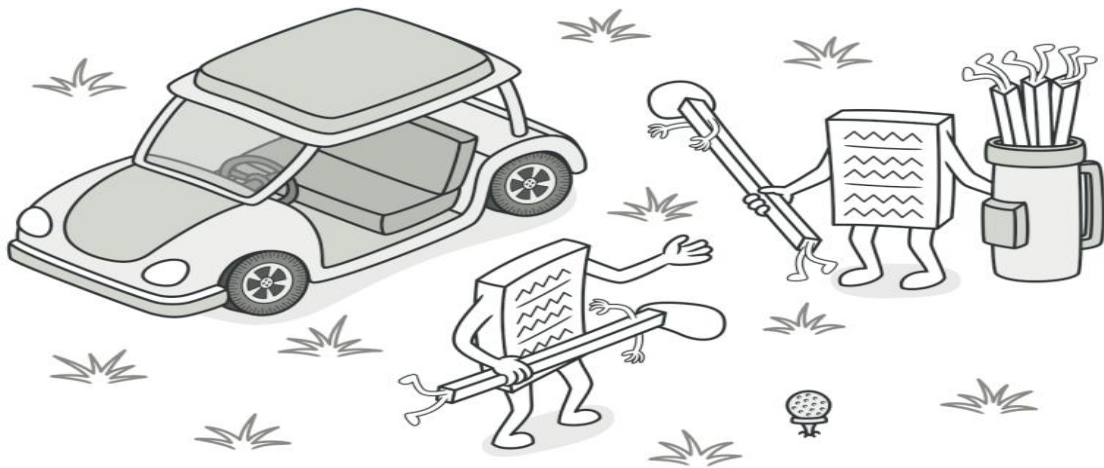
Structure:



Pros & Cons :

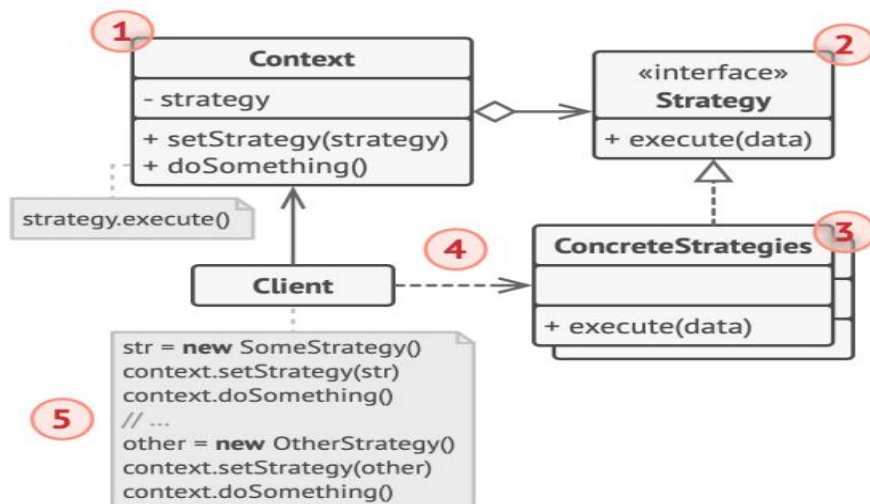
- ✚ a) Single Responsibility Principle. Organize the code related to particular states into separate classes.
- ✚ b) Open/Closed Principle. Introduce new states without changing existing state classes or the context.
- ✚ c) Simplify the code of the context by eliminating bulky state machine conditionals.

✗) Applying the pattern can be overkill if a state machine has only a few states or rarely changes.



H) STRATEGY

Structure:



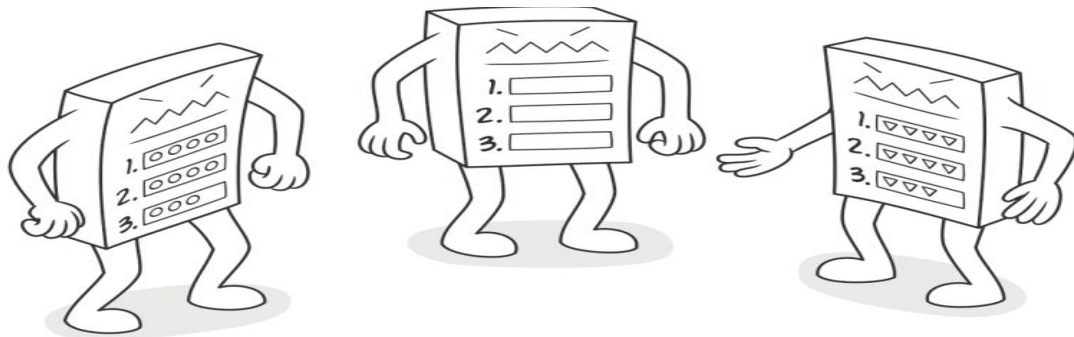
Pros & Cons:

- + a) You can isolate the implementation details of an algorithm from the code that uses it.
- + b) You can replace inheritance with composition.
- + c) Open/Closed Principle. You can introduce new strategies without having to change the context.

X) 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.

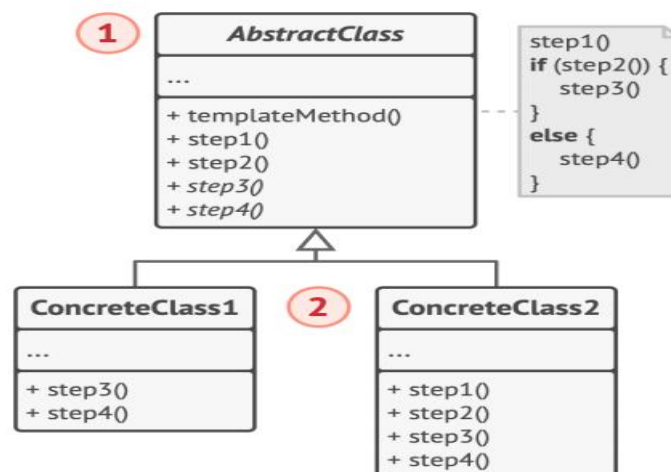
X)Clients must be aware of the differences between strategies to be able to select a proper one.

X)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.



I) TEMPLATE METHOD

Structure:



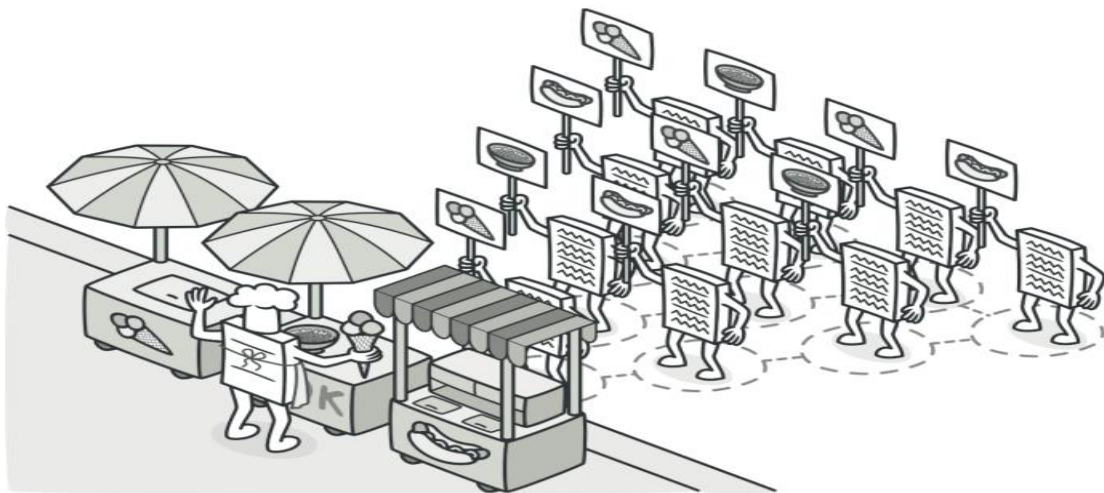
Pros & Cons :

- ✚ a) You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- ✚ b) You can pull the duplicate code into a superclass.

✗) Some clients may be limited by the provided skeleton of an algorithm.

✗) You might violate the *Liskov Substitution Principle* by suppressing a default step implementation via a subclass.

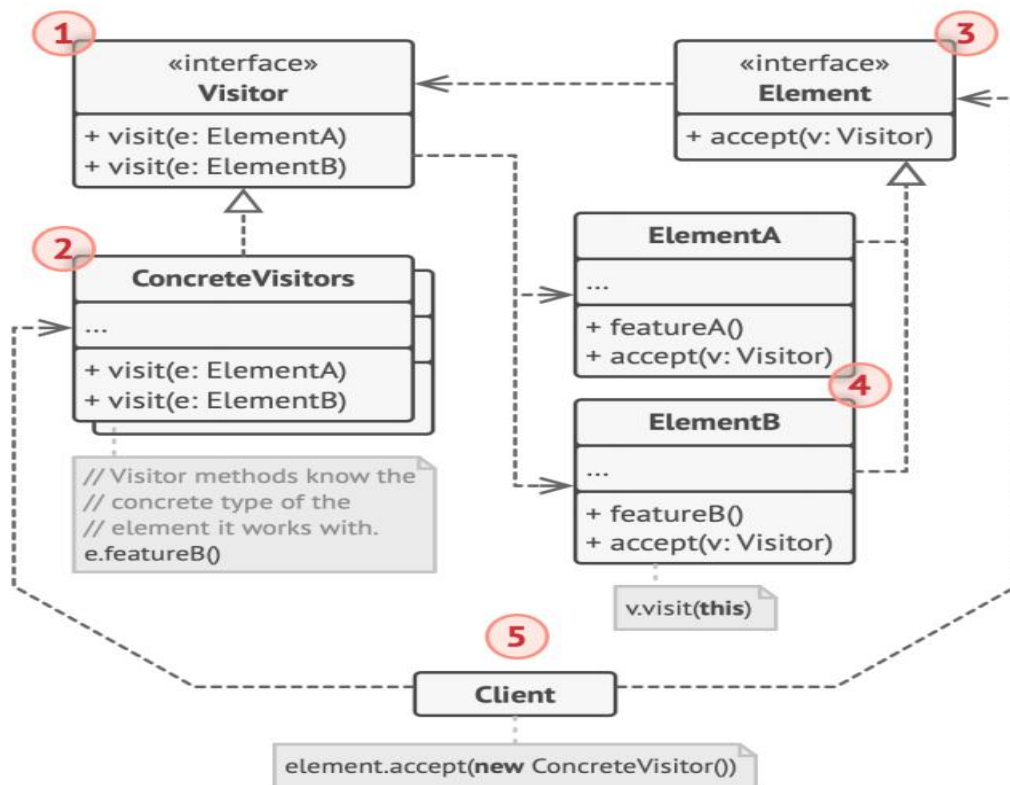
✗) Template methods tend to be harder to maintain the more steps they have.



J) VISITOR

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Structure :



Pros & Cons :

- ✚ a) *Open/Closed Principle*. You can introduce a new behavior that can work with objects of different classes without changing these classes.
- ✚ b) *Single Responsibility Principle*. You can move multiple versions of the same behavior into the same class.
- ✚ c) A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.
- ✗ d) You need to update all visitors each time a class gets added to or removed from the element hierarchy.
- ✗ e) Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.