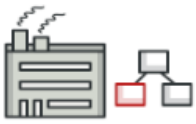
Creational



Create object without specifying its concrete class
(Logistic app which create Truck or Ship as transporter)

Factory Method

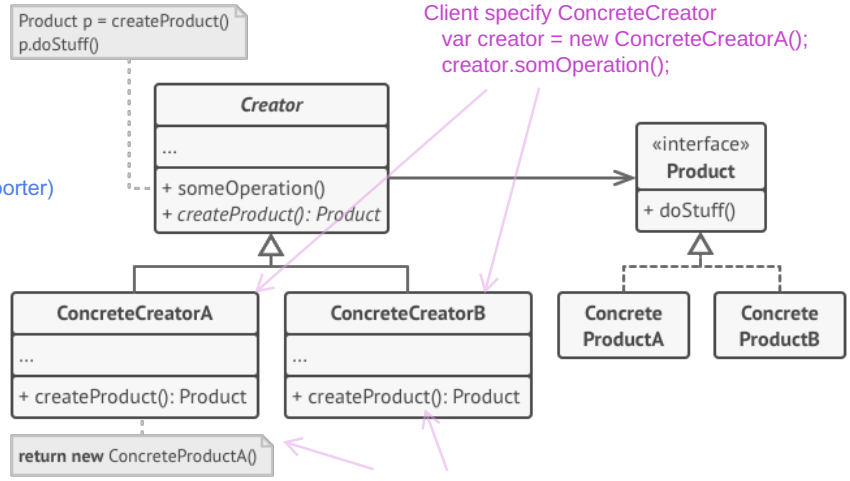
(Virtual Constructor)

Complexity: ★☆☆

Popularity: ★★★

- Avoid coupling of Client and ConcreteProduct
- Easy to introduce new types of products.

x The code may become more complicated because of many subclasses



Create objects of an entire family without specifying their concrete classes
(Furniture factory with multiple
product families and their variants)

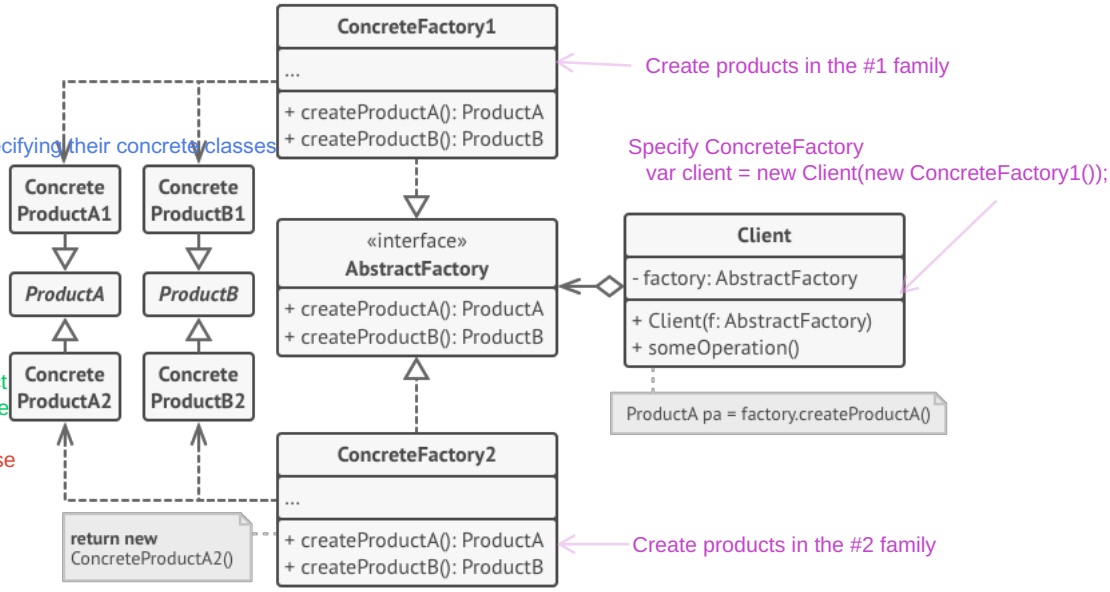
Abstract Factory

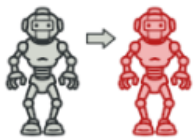
Complexity: ★★☆

Popularity: ★★★

- Loosely couple of Client and ConcreteProduct
- Products getting from a factory are compatible
- Easy to introduce new variants of products

x Code may become more complicated because of many interfaces and classes





Clone object

Prototype

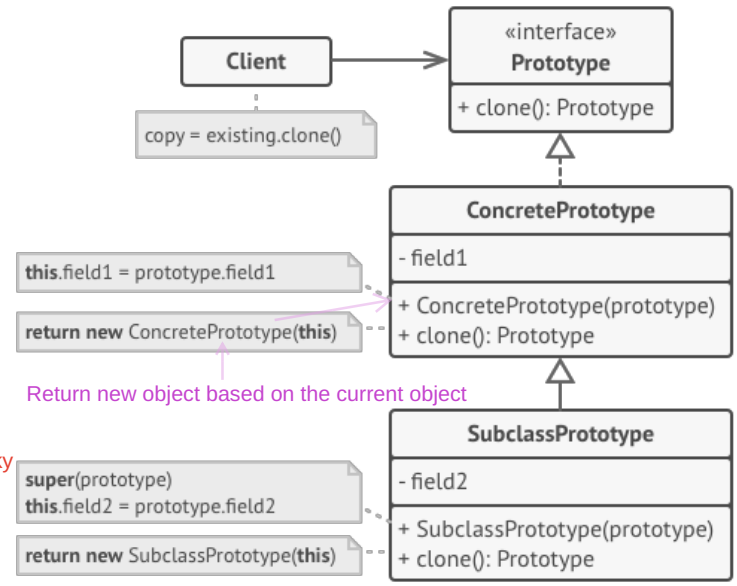
(Clone)

Complexity: ★☆☆

Popularity: ★★☆

- Can create object with configuration presets
- DRY

x Cloning objects having circular reference may be tricky



Lets you ensure that a class has only one

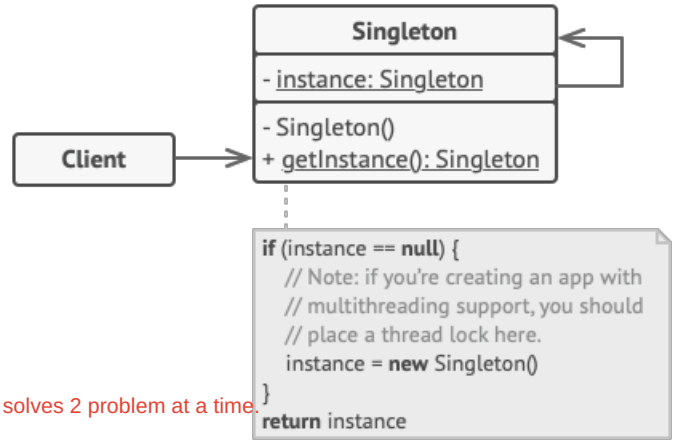
Singleton

Complexity: ★☆☆

Popularity: ★★☆

- Gain global access point to that instance
- Object it initialized when it's requested for the fist time

x Violates the Single Responsibility Principle. Its constructor solves 2 problem at a time.
x The pattern require special treatment in a multi-threading.
x It may be difficult to unit test Client code of singleton since constructor of singleton is private and difficult to mock



Construct complex objects step by step
(Houses contruction)

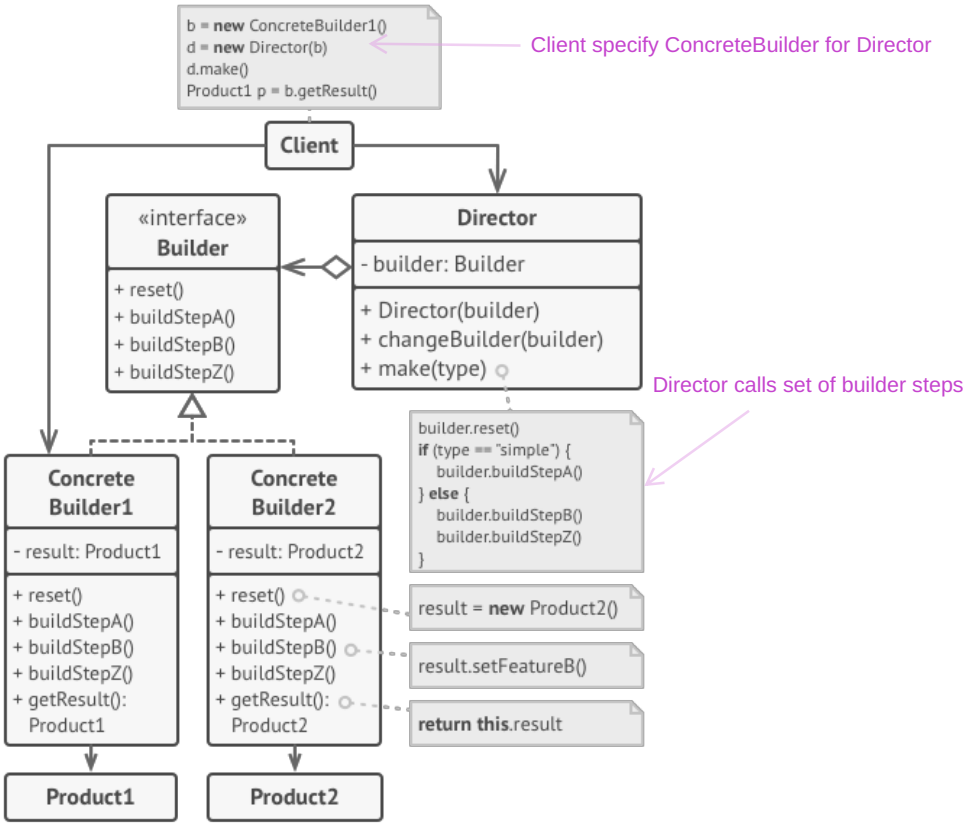
Builder

Complexity: ★★☆

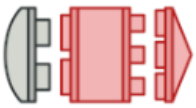
Popularity: ★★★

- Isolate complex construction code from business logic of the product

x Code will be more complex a bit.



Structural



Collaborate objects with incompatible interfaces.
(Electricity adapter)

Adapter

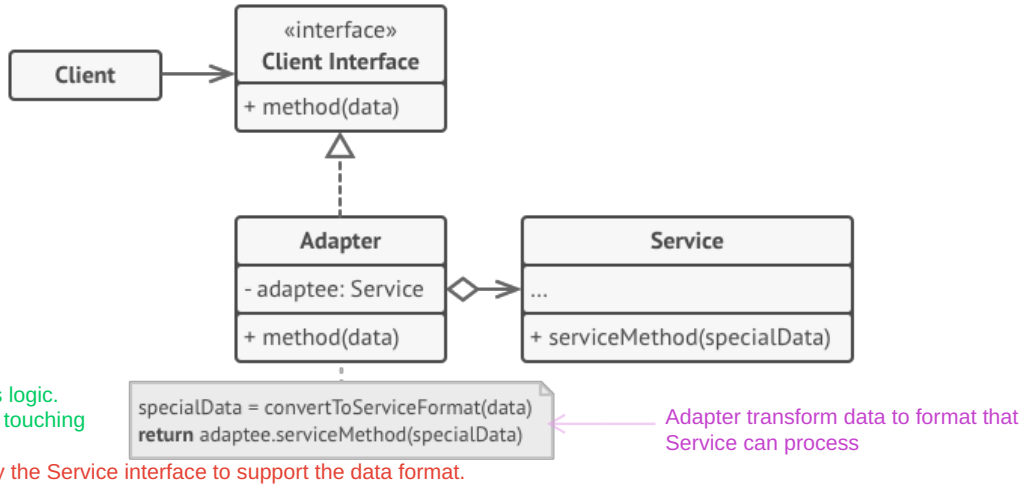
(Wrapper)

Complexity: ★☆☆

Popularity: ★★★

- Isolate data conversion code from core business logic.
- Easy to introduce new types of adapters without touching the Service or Client code.

x It may over-complicated since we can just modify the Service interface to support the data format.



Allows adding new behaviors to objects dynamically
(Notifier system that we can keep adding new channels to send noti)

Decorator

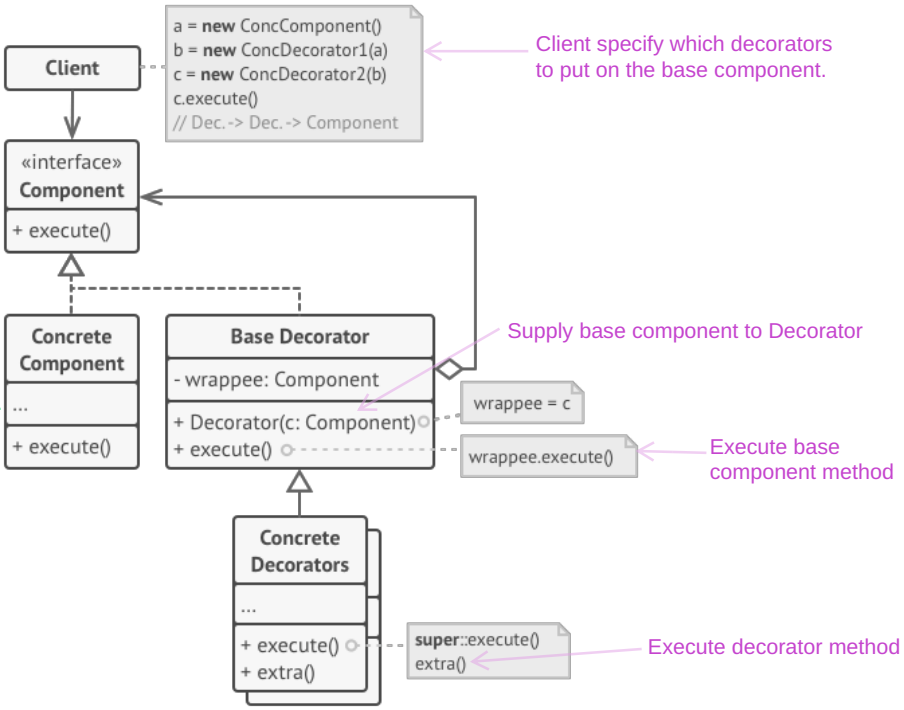
(Wrapper)

Complexity: ★★☆

Popularity: ★★☆

- Can extend object's behavior without making a new subclass.
- Can add or remove responsibilities from an object at runtime.
- Can combine several behaviors to an object.

x It's hard to remove a specific wrapper from wrappers stack.
x It's hard to implement decorator that its behavior doesn't depend on the order of wrappers stack.
x The initial configuration code of layered decorators might be



Provides a simplified (but limited) interface to a complex system of classes, library or framework.

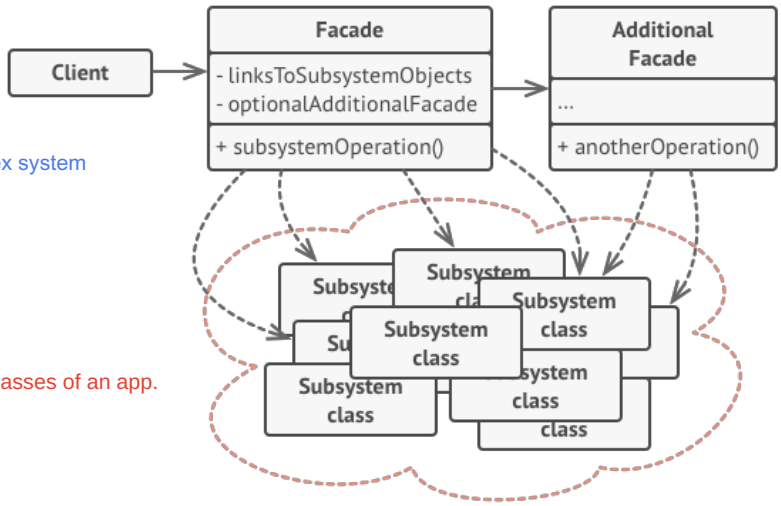
Facade

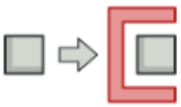
Complexity: ★☆☆

Popularity: ★★☆

- Isolate your code from the complexity of a systems.

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





Proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object

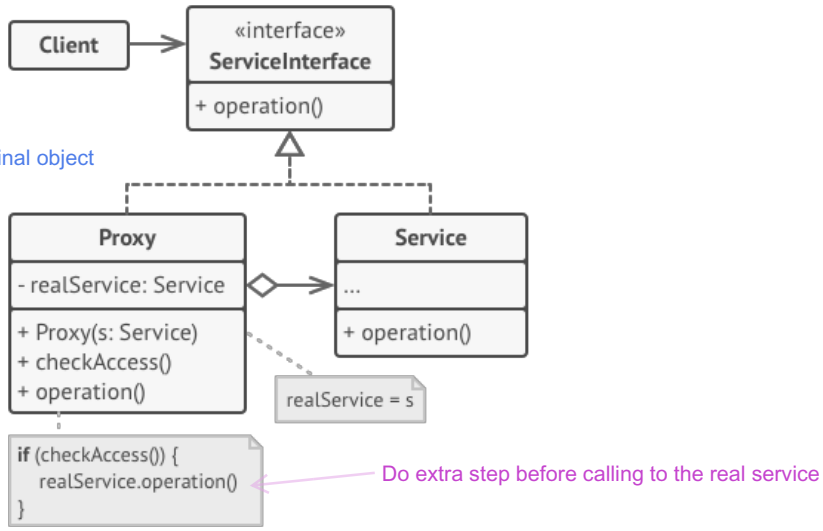
Proxy

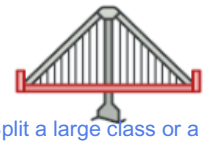
Complexity: ★★☆

Popularity: ★☆☆

- Can control service object without Client knowing.
- Can manage service lifecycle without Client knowing.
- Proxy works event if the service isn't ready.
- Easy to introduce new proxy.

x The response from the service might get delayed.





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.

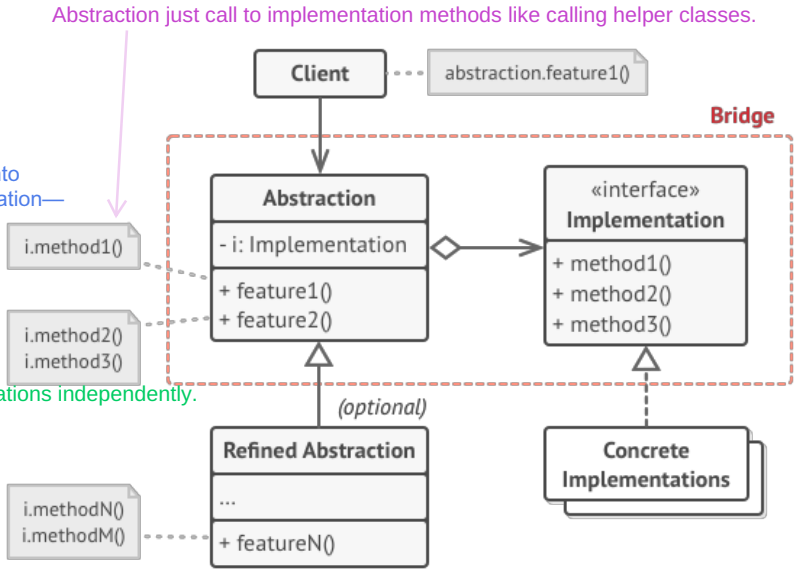
Bridge

Complexity: ★★★

Popularity: ★☆☆

- Easy to introduce new abstractions and implementations independently.
- Client code work with high-level abstractions.

x Code may become complicated by applying the pattern to a highly cohesive class.



Lets you compose objects into tree structures and then work with these structures as if they were individual objects. (Your shopping box un-boxing)

Composite

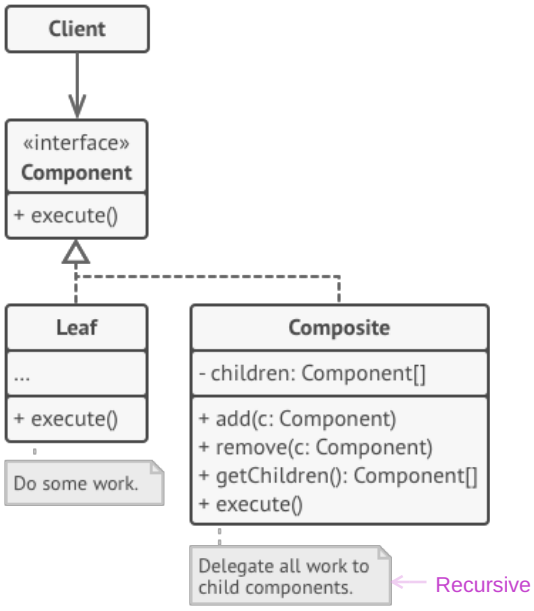
(Object Tree)

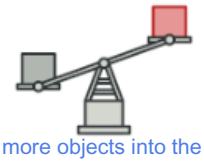
Complexity: ★★☆

Popularity: ★★☆

- More convenient to work with complex tree
- Easy to introduce new element types.

x You'd need to overgeneralize the component interface.





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.

Flyweight

(Cache)

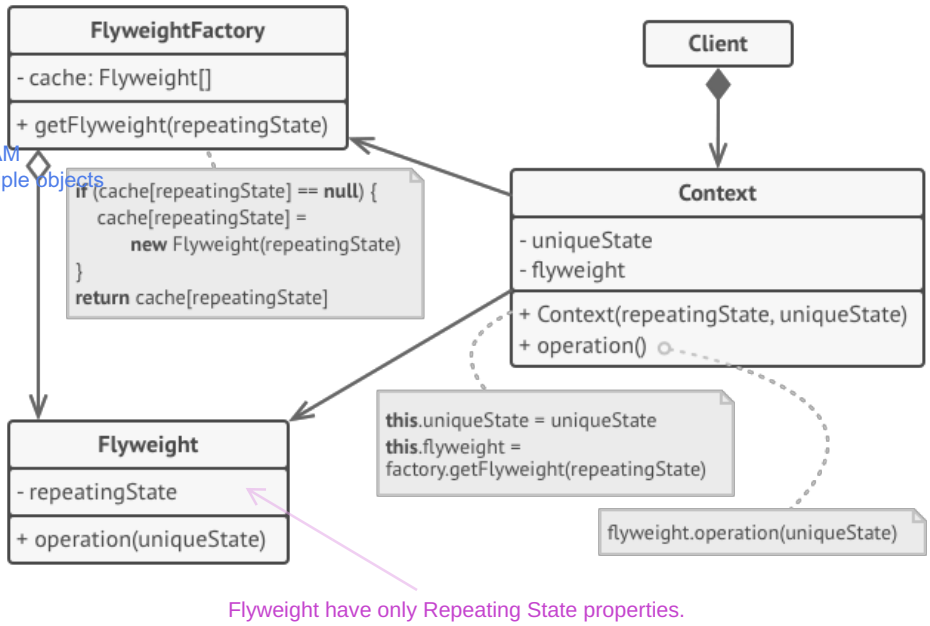
Complexity: ★★★

Popularity: ★☆☆

- Can save lots of RAM if your program has lots of similar objects.

x Might be trading RAM with CPU cycles when some of the context data recalculated each time calling a flyweight method.

x New team members will always wondering why state of an entity was exasperated.



Flyweight have only Repeating State properties.

Behavioral

Pass requests along a chain of handlers

Chain of Responsibility

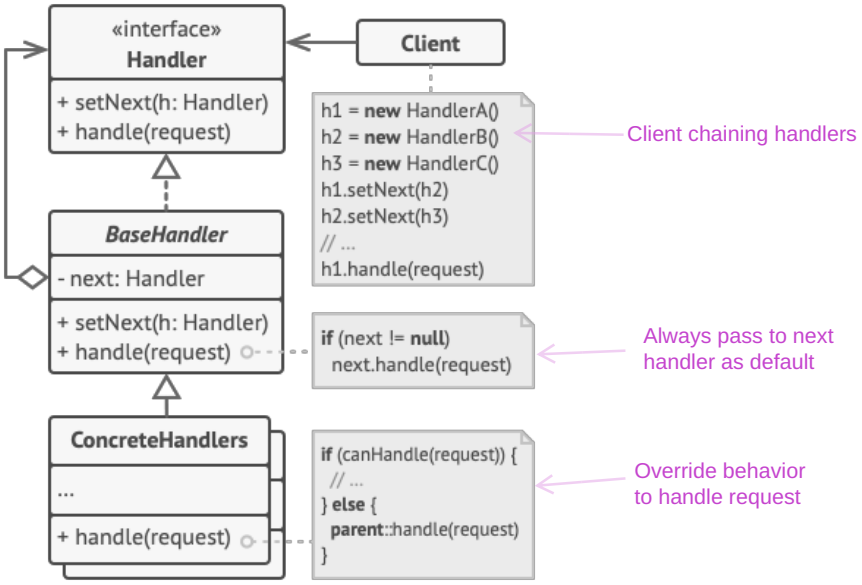
(CoR, Chain of Command)

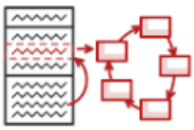
Complexity: ★★☆

Popularity: ★☆☆

- You can control the order of request handling.
- Decouple classes that invoke and perform operations.
- Easy to introduce new handlers

x Some request may end up unhandled.





Lets an object alter its behavior when its internal state changes (Document approval)

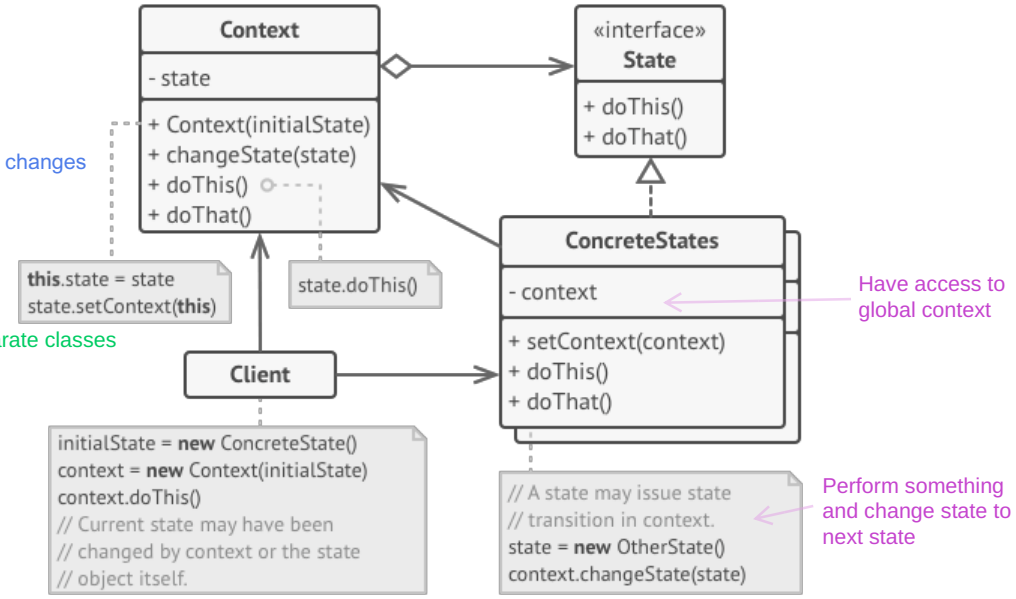
State

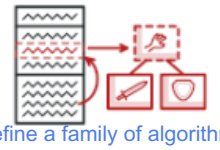
Complexity: ★☆☆

Popularity: ★★☆

- Organize code related to a particular states into separate classes
- Easy to introduce new states
- Eliminate bulky state machine conditionals

x May be overkill if the state machines has only a few states or rarely changes.





Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
(Picking route type on Navigation App)

Strategy

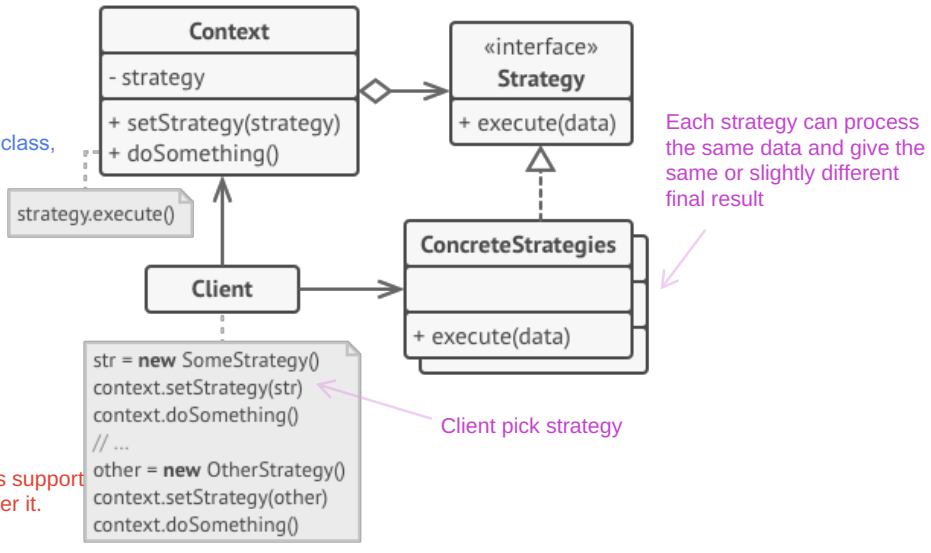
Complexity: ★☆☆

Popularity: ★★★

- You can swap algorithm used at runtime.
- Isolate implementation of algorithms from the code using it.
- Can replace inheritance composition.
- Easy to introduce new strategies.

x Client must be aware of the different between strategies.

x A lot of modern programming languages have functional types support that lets you implement different version of an algorithm, consider it.



Command handle, delay or queue a request's execution.
(GUI button actions)

Command

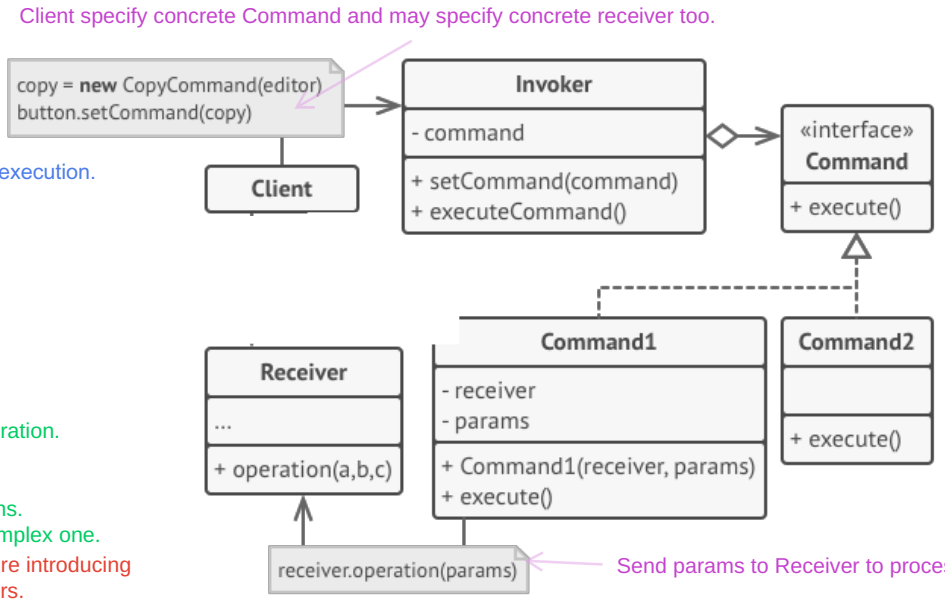
(Action, Transaction)

Complexity: ★☆☆

Popularity: ★★★

- Decouple classes that invoke and perform operation.
- Easy to introduce new commands
- Can implement undo/redo.
- Can implement deferred execution of operations.
- Assemble a set of simple commands into a complex one.

x The code may be more complicated since you're introducing a whole new layer between senders and receivers.



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

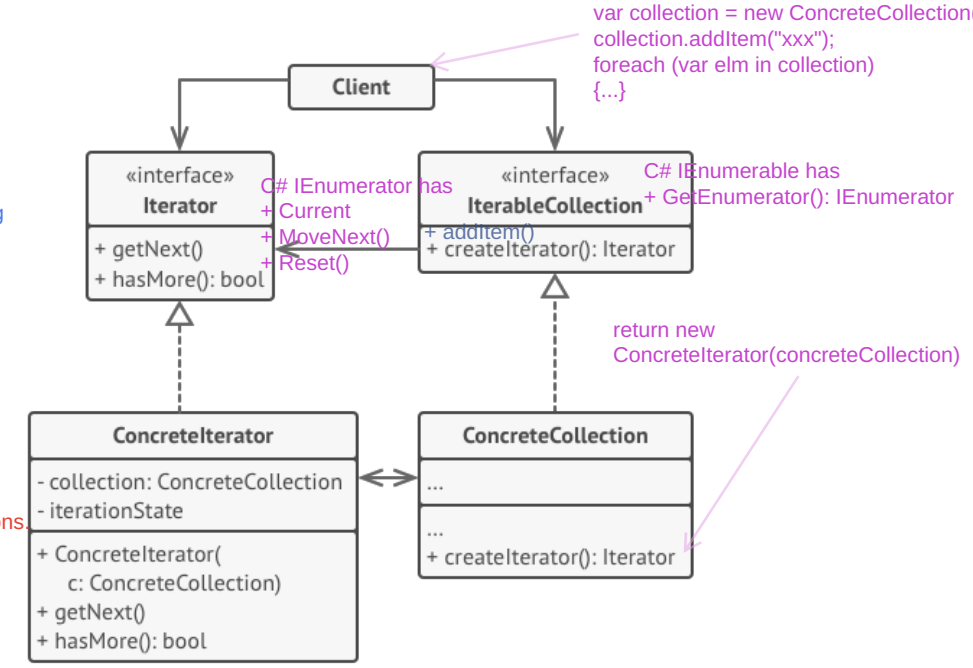
Iterator

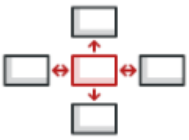
Complexity: ★★☆

Popularity: ★★★

- Extract bulky traversal algorithms into separate class.
- Easy to introduce new collection and iterator.
- Can iterate over the same collection in parallel becaz each iterator have it own state.

x May be overkill if your app only works with simple collections.
x Accessing element via iterator may be less efficient than direct access.





Reduces coupling between components of a program by making them communicate indirectly, through a special mediator object

Mediator

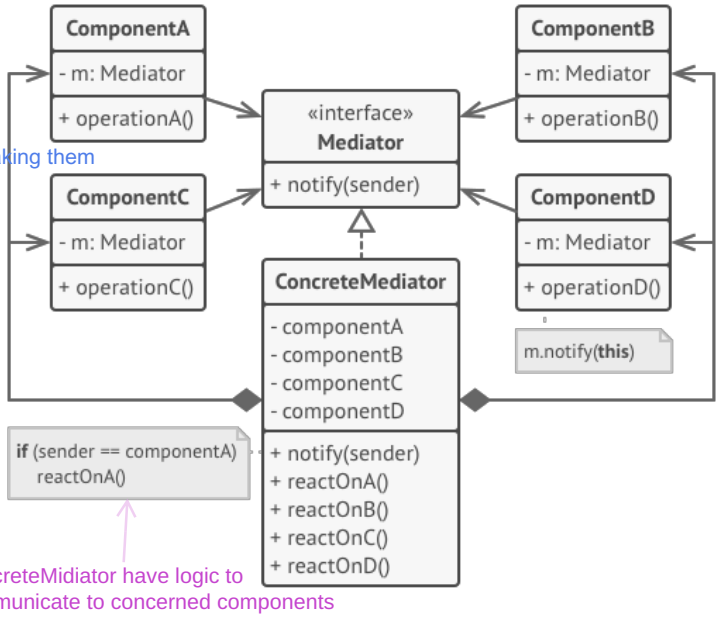
(Intermediary, Controller)

Complexity: ★★☆

Popularity: ★★☆

- Extract communications between components into a single place
- Easy to introduce new mediators
- Decouple components

x A mediator can become a god object coupled to all components.



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

Memento

(Snapshot)

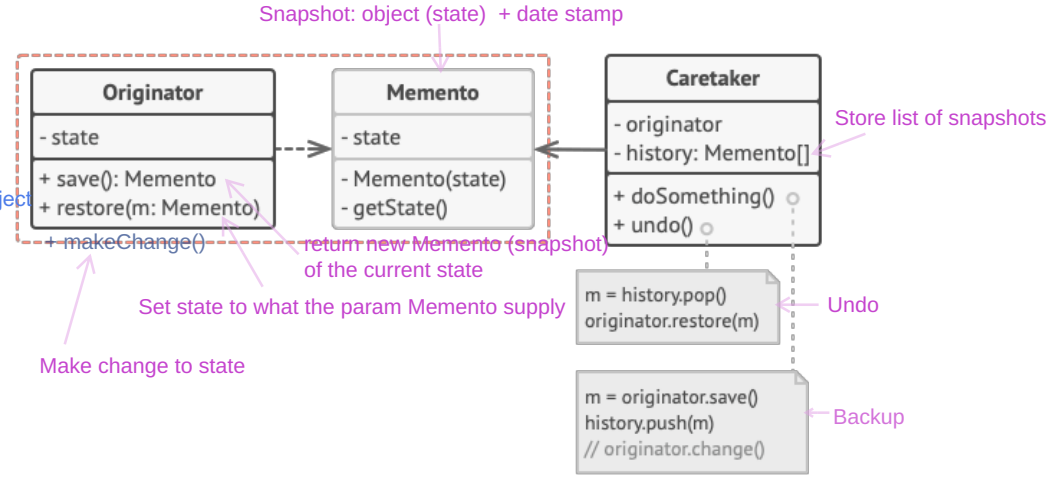
Complexity: ★★★

Popularity: ★☆☆

- Can produce snapshots of object's states without violating its encapsulation
- Can simplify Originator code by letting CareTaker maintain the history records.

x The app may consume RAM if clients create momento to much.

x Caretaker should track the Originator lifecycle to manage obsolete shapshots.



Publisher + Subscriber

Observer

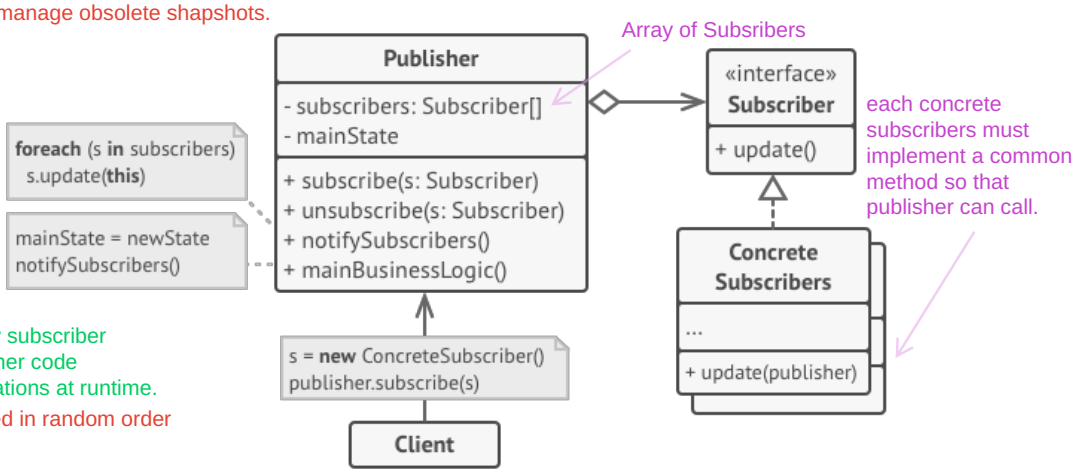
(Event-Subscriber, Listener)

Complexity: ★★☆

Popularity: ★★★

- Easy to introduce new subscriber without touching publisher code
- Can control object relations at runtime.

x Subscribers are notified in random order



The skeleton of an algorithm in the SUPERclass but lets SUBclasses override specific steps of the algorithm without changing its structure.

Template Method

Complexity: ★☆☆

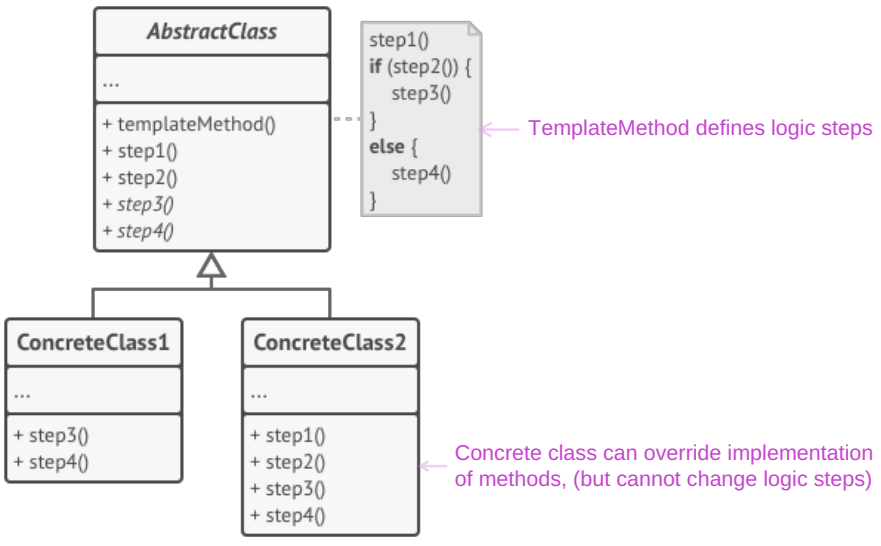
Popularity: ★★☆

- Can let clients override only certain part of a large algorithms.
- Can pull a duplicated code into a superclass

x Some clients may be limited by the provided skeleton.

x Might violate Liskov Substitution Principle by suppresing steps implementation via subclass

x Template method then to be harder to maintain when having more steps.



Allows adding new behaviors to existing class hierarchy without altering any existing code.

Visitor

Complexity: ★★★

Popularity: ★☆☆

- Easy to introduce new behavior to a class
- Can enhance and introduce new version of a behavior to the same class
- Visitor object can accumulate some useful information while working with various objects

x Need to update all visitors when a concrete element class added or removed.

x Visitor might lack access to private fields of elements.