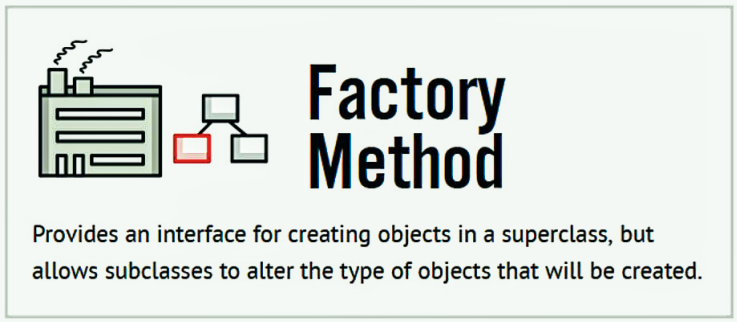
- Design Patterns
  - Creational Patterns
    - Factory Method
    - Abstract Factory
    - Builder
    - Prototype
    - Singleton
  - Structural Patterns
    - Adapter
    - Bridge
    - Composite
    - Decorator
    - Facade
    - Flyweight
    - Proxy
  - Behavioral Patterns
    - Command
    - Chain Of Responsibility
    - Iterator
    - Mediator
    - Momentam
    - Observer
    - Strategy
    - State
    - Template Method
    - Visitor

- Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.
- Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.
- Creational patterns = Factory Method + Abstract Factory + Builder + Prototype + Singleton

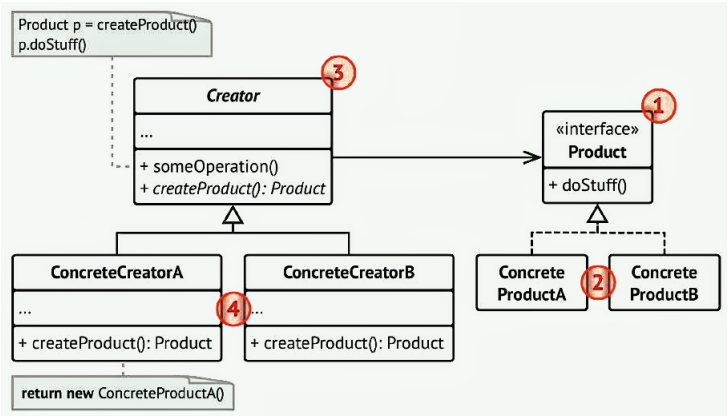
စက်ရုံ ၂ခု၊ တည်ဆောက်သူ၊ ပုံတူ၊ တစ်ဦးတည်း



FACTORY METHOD

Also known as: Virtual Constructor

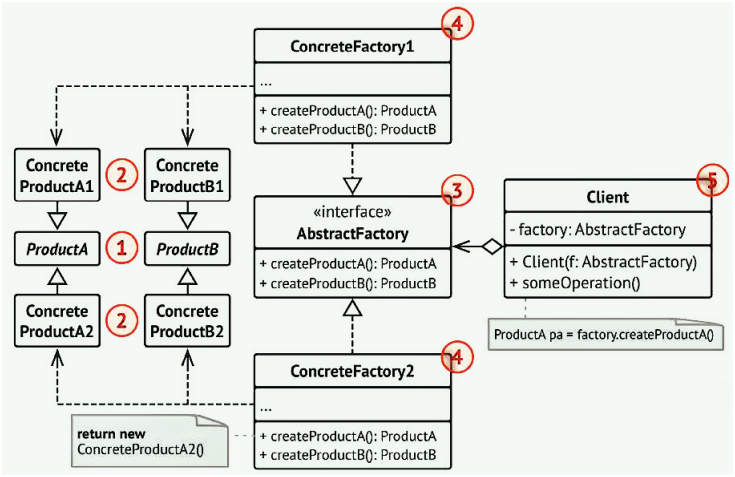
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



- Template Method < Factory Method < Abstract Factory
- Factory Method vs Abstract Factory vs Prototype vs Builder
  (less complicated) (more flexible but more complicated)
- Factory Method vs Abstract Factory
  (a set of Factory Methods, which can be composed by Prototype)
- Factory Method + Iterator (collection subclasses)
- Factory Method vs Prototype
  (inheritance) (not based on inheritance)
- Factory Method vs Template Method
  (specialization of Template Method)

ABSTRACT FACTORY

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



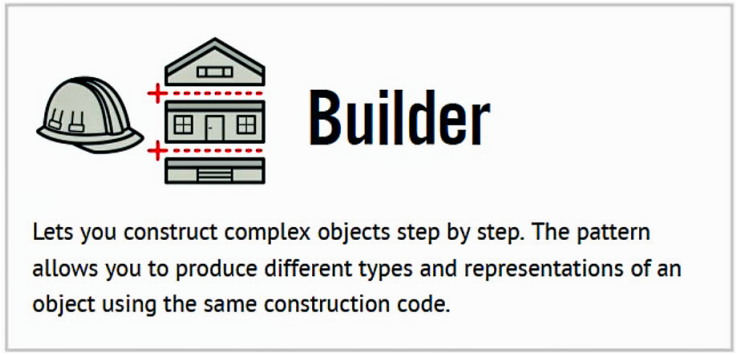
- Builder (construct complex objects step by step) vs Abstract Factory (specializes in creating families of related objects)

- Abstract Factory (a set of Factory Methods) vs Factory Method, + Prototype (can use Prototype to compose the methods)

- Abstract Factory (if only want to hide the way the subsystem objects from the client) vs Facade

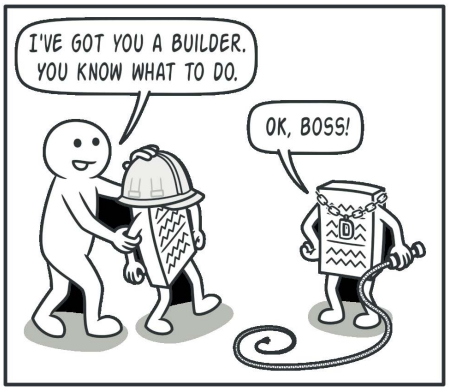
- Abstract Factory + Bridge
(when some abstractions defined by Bridge can only work with specific implementations)
(Abstract Factory can encapsulate these relations and hide the complexity from the client)

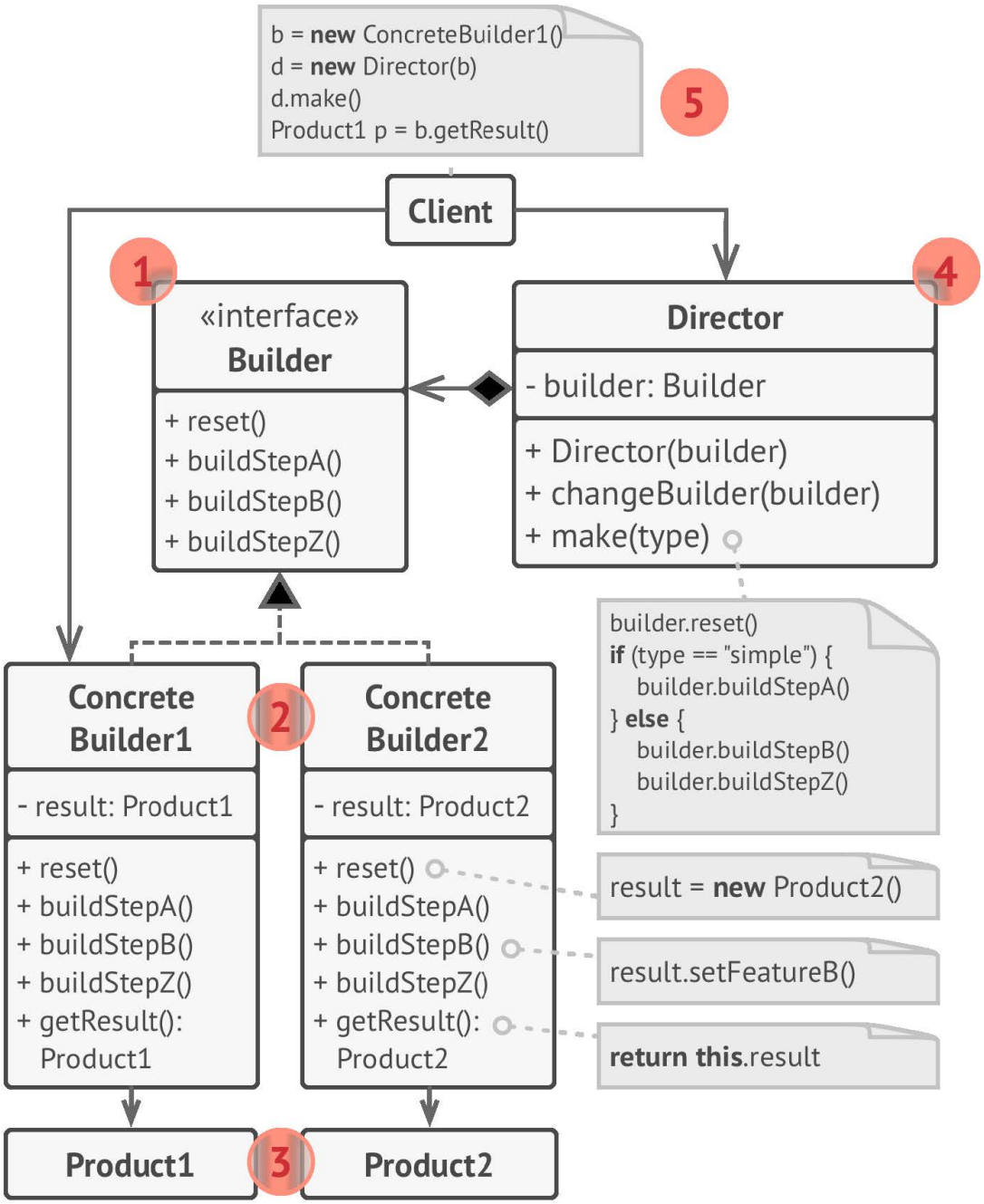
- Abstract Factory, Builder, Prototype, vs Singleton
(they can implemented as Singleton)



BUILDER

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


I'VE GOT YOU A BUILDER. YOU KNOW WHAT TO DO.
OK, BOSS!


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
5
Client
1
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
4
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
2
Concrete
Builder2
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2()
result.setFeatureB()
return this.result
Product1
3
Product2

• Builder pattern is to get rid of using constructor (such as with ten optional parameters)

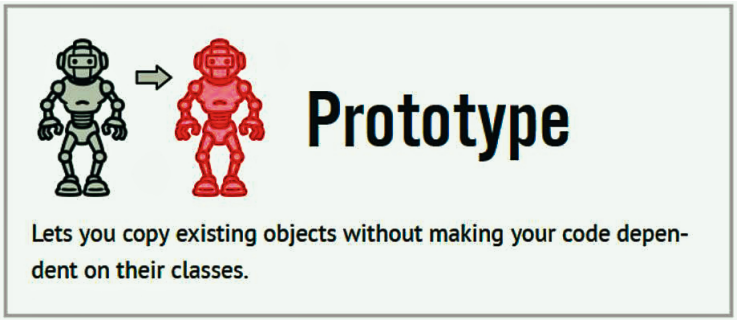
• Builder pattern is able to create different representation of some product (such as stone and wooden houses involves similar steps that differ only in the details)

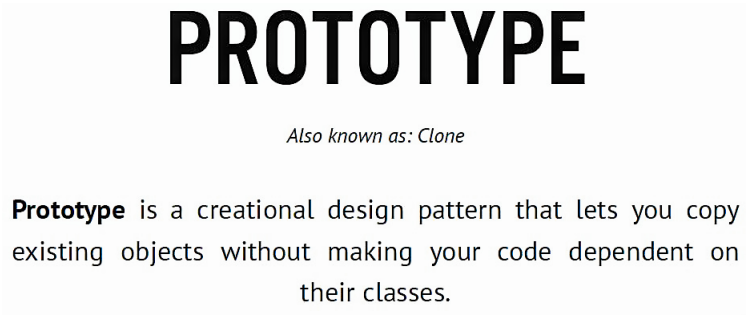
• The base builder interface defines all possible construction steps

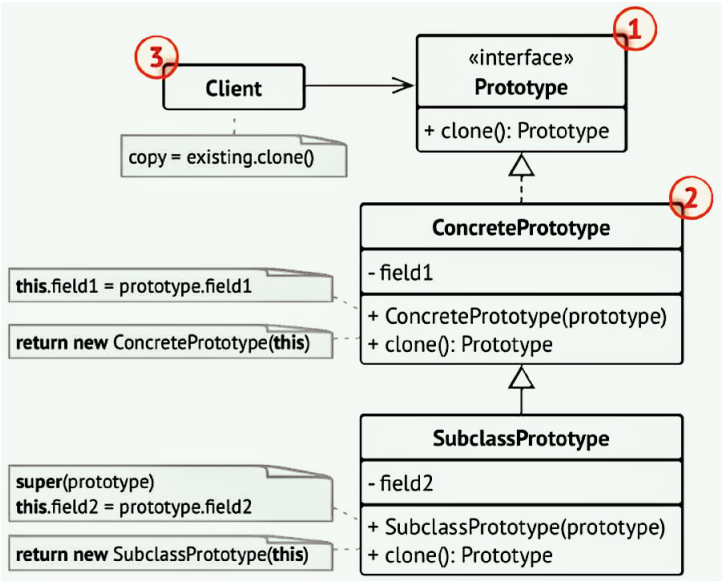
• The director class guides the order of construction

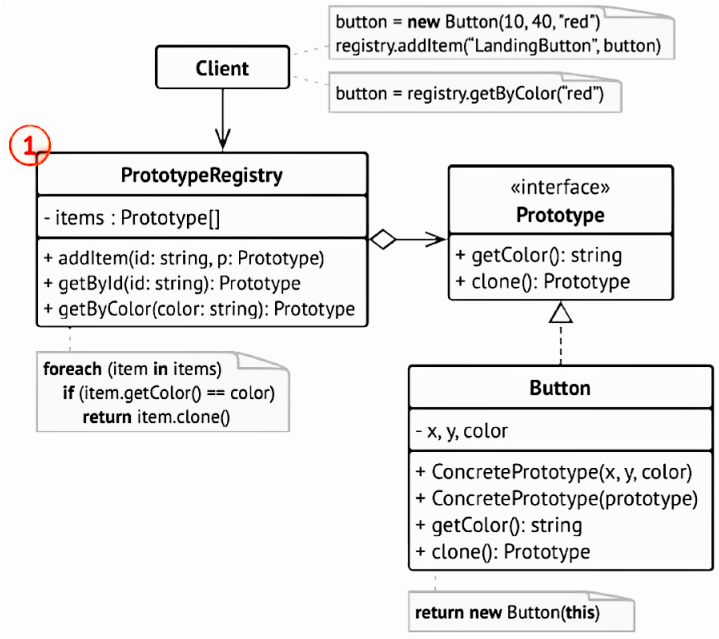
• Builder pattern can be used to construct Composite trees or other complex objects

• Factory Method vs Abstract Factory, Prototype, Builder
(less complicated and more customizable via subclasses) (more flexible but more complicated)









- Prototype pattern can be used when not depend on the concrete classes of objects that you need to copy

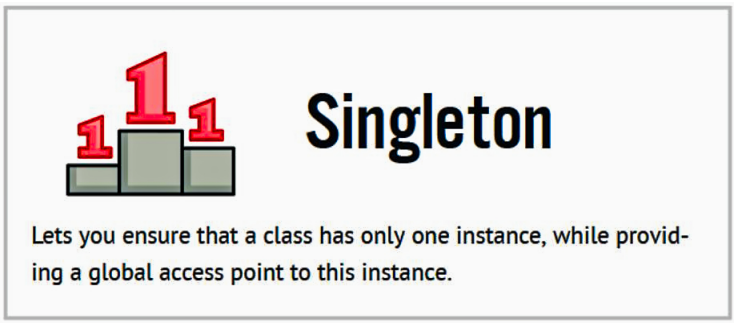
- Prototype pattern lets us use a set of pre-built objects configured in various ways as prototypes

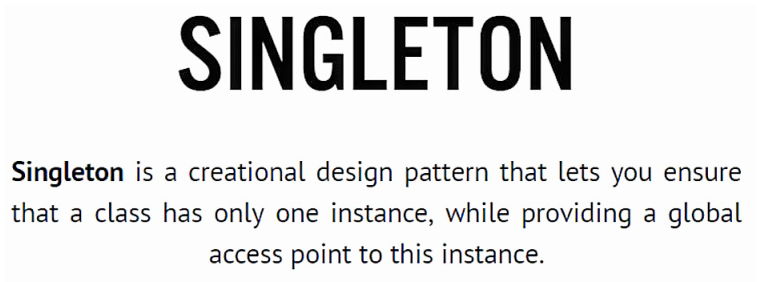
- Prototype can help to save copies of Commands into history

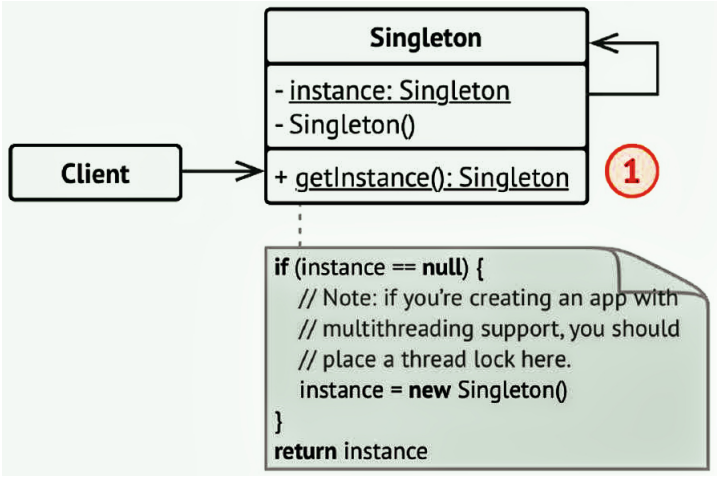
- Composite + Decorator + Prototype
(lets us clone complex structures instead of re-constructing them from scratch)

- Prototype vs Memento
(when want to store in the history)

- Abstract Factory, Builder, Prototypes, vs Singleton
(they all can be implemented as Singleton)







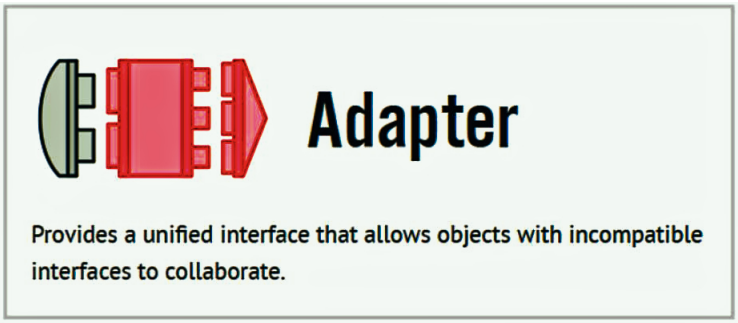
- Singleton pattern can make sure to have a single instance available to all clients

- Singleton pattern can be used to have stricter control over global variables

- Facade vs Singleton
(It can transform into a Singleton)

- Flyweight vs Singleton
(It would resemble Singleton when can reduce all shared to just one flyweight object)
(Flyweight = multiple instances with different intrinsic states, immutable)
(Singleton = only one instance, mutable)

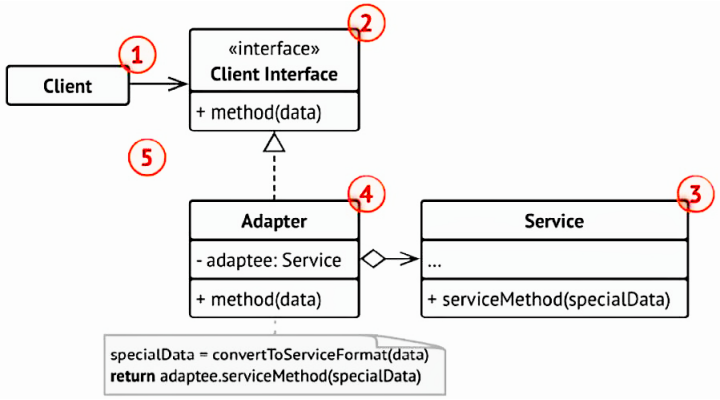
- Structural patterns = Adapter + Bridge + Composite + Decorator + Facade + Flyweight + Proxy
  (A,B,C,D,2F,P)



ADAPTER

Also known as: Wrapper

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



- Adapter pattern is able to use when we want to reuse several existing subclasses

- Bridge vs Adapter
  (develop each part independently) (with existing app to work together nicely)

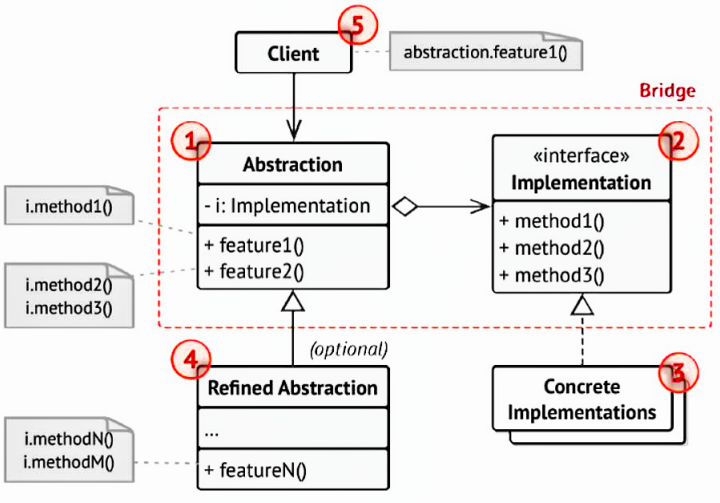
- Adapter vs Decorator vs Proxy
  (change existing interface) (enhance interface) (same interface)

- Facade vs Adapter
  (new interface) (tries to make the existing interface usable

- Bridge, State, Strategy, vs Adapter
  (based on composition, which is delegating work to other objects)

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.



• Bridge pattern can be used when we want to divide and organize some functions

• Bridge pattern can be used when we need to extend a class

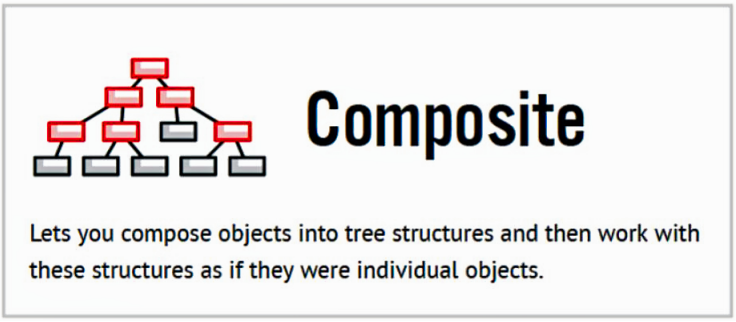
• Bridge pattern can be used when we need to switch implementations at runtime

• Bridge vs Adapter
(when develop each part independently) (work existing app together)

• Bridge, State, Strategy, vs Adapter
(similar structures)

• Bridge + Abstract Factory
(encapsulate relations and hide the complexity from the client)

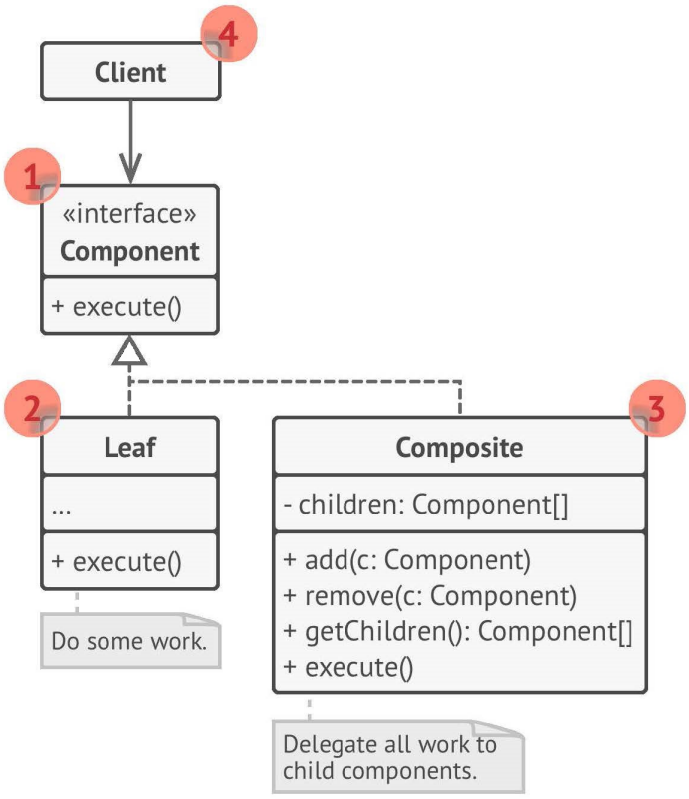
• Bridge + Builder
(director = the role of abstraction)
(builders = the rolw of implementations)



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.



• Composite pattern can be used when we need to implement a tree-like object structure

• Composite pattern can be used when we want to treat both simple and complex elements uniformly

• Composite + Builder
(when need contruction steps to work recursively)

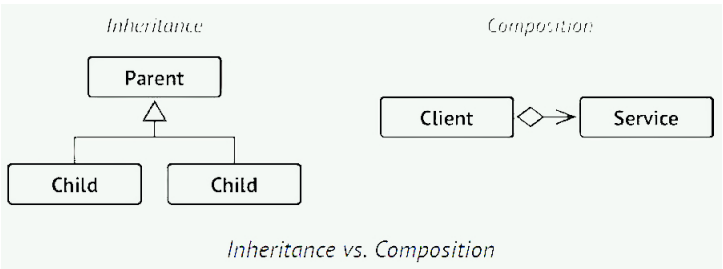
• Composite + Chain of Responsibility
(when need through the chain of all parent components to the root of object tree)

• Composite + Iterator
(when traverse Composite trees)

• Composite + Visitor
(when need to execute over entire Composite tree)

• Composite + Flyweight
(when need to save some RAM)

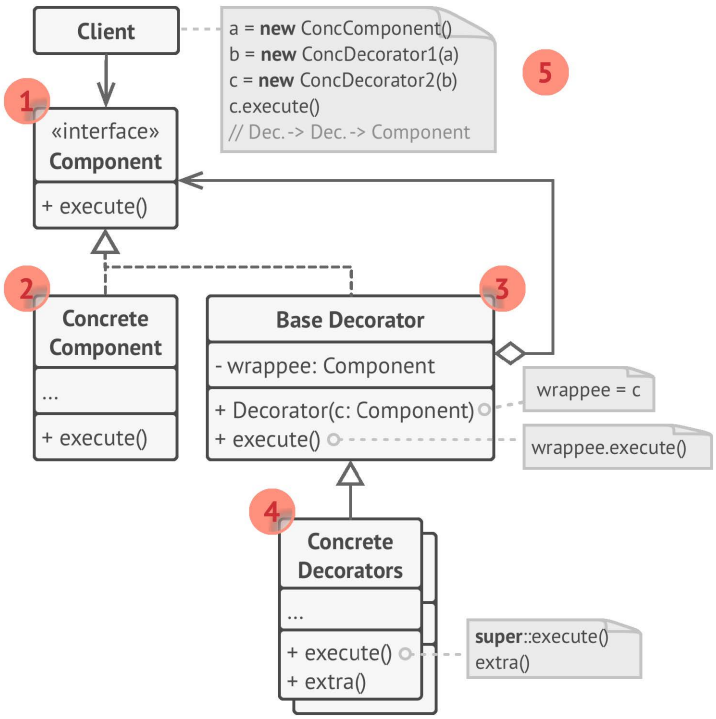
• Composite + Decorator + Prototype
(similar structure) (when need to clone complex structures instead of re-constructing)

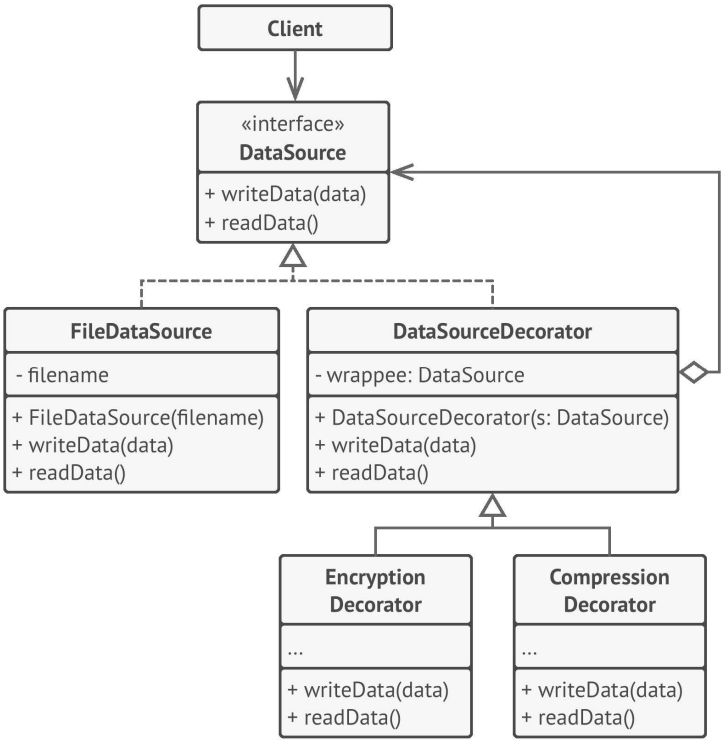


DECORATOR

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.





- Decorator pattern can be used when we need to assign extra behaviors to objects at runtime

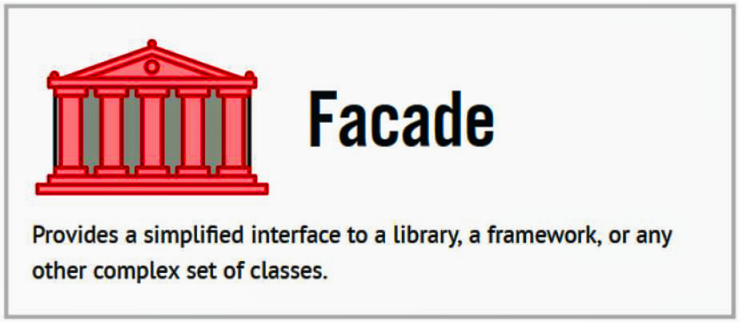
- Decorator vs Adapter vs Proxy
  (enhance existing interface) (change existing interface) (with same interface)
  (supports recursive composition) (doesn't support recursive composition)

- Decorator vs Chain of Responsibility
  (base interface) (operations independently each other)
  (not break the request flow) (can stop passing the request flow further)

- Decorator vs Composite
  (add additional responsibilities) (sums-up its children's results)

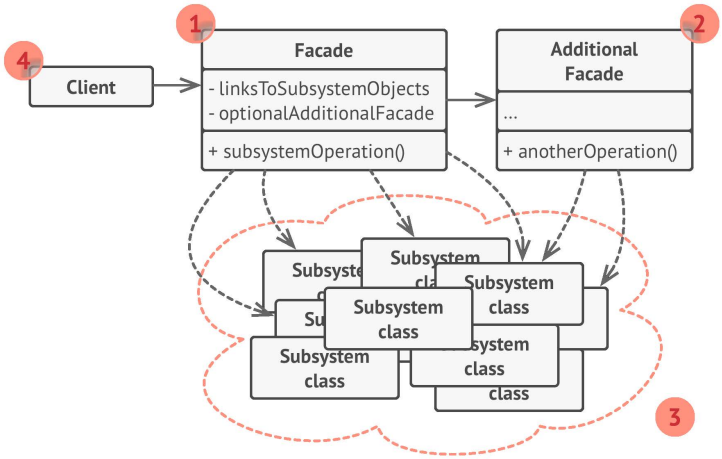
- Decorator vs Strategy
  (change the skin) (change the guts)

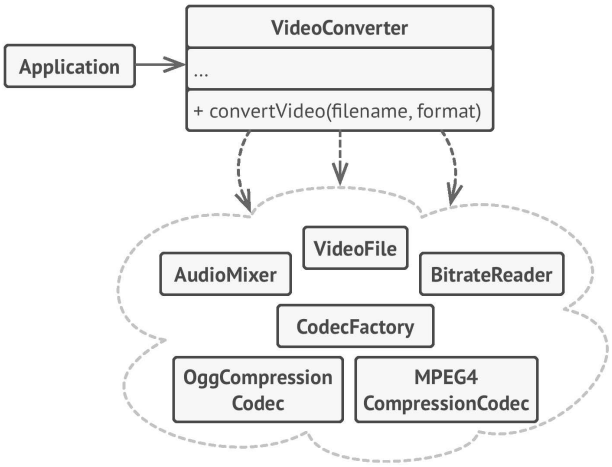
- Decorator vs Proxy
  (controlled by the client) (manages the life cycle of its service object on its own)



FACADE

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





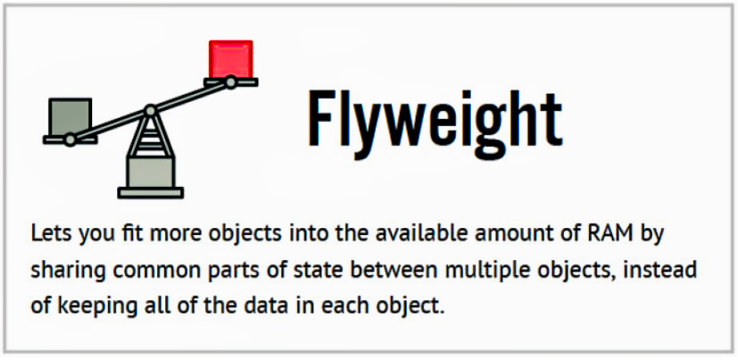
- Facade vs Adapter
  (new interface) (existing interface usable, and usually wraps just one object)

- Facade vs Abstract Factory
  (similar when we want to hide complexity from the client)

- Facade vs Flyweight
  (shows a single object represent an entire subsytem) (shows how to make lots of little objects)

- Facade vs Mediator
  Facade = hide complex from the client and subsystems can communicate with each other directly
  Mediator = hide complex from the client and they don't communicate directly

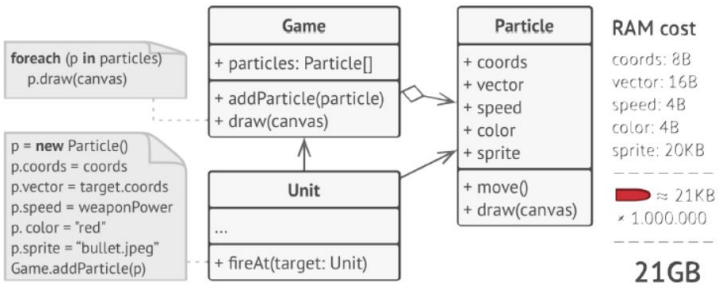
- Facade vs Proxy
  (both buffer a complex entity and initialize it on its own)
  Proxy = the same interface as its service object, with makes them interchangeable

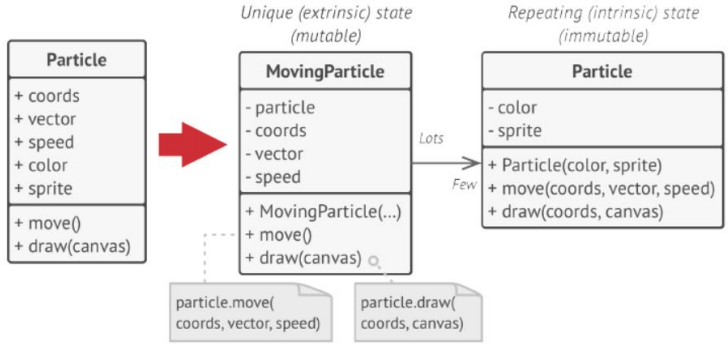


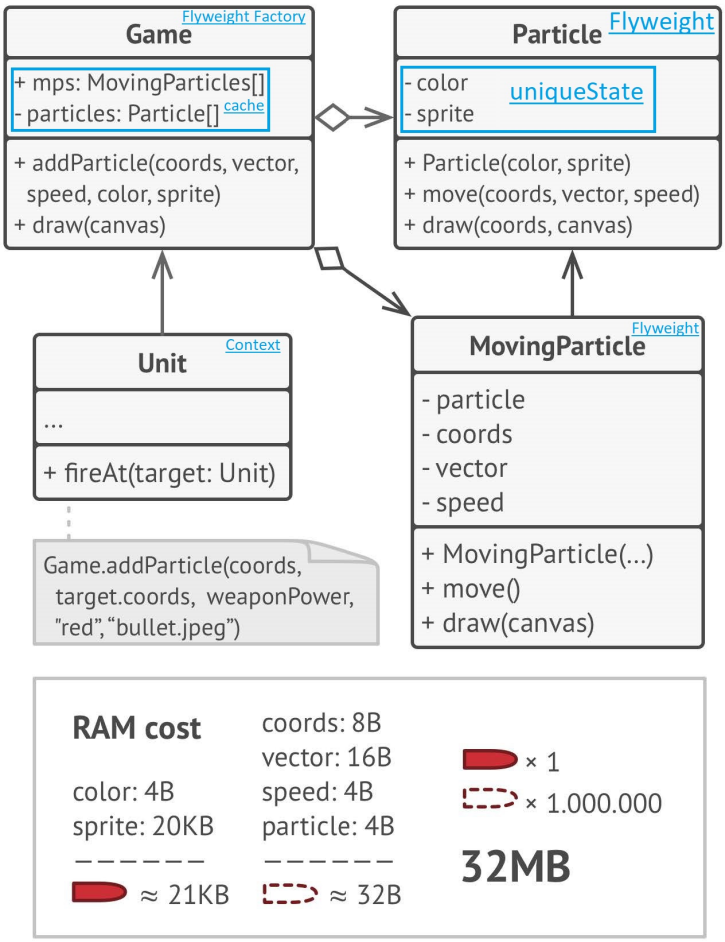
FLYWEIGHT

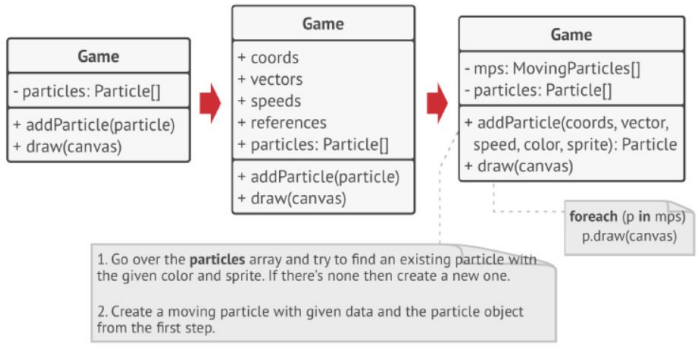
Also known as: Cache

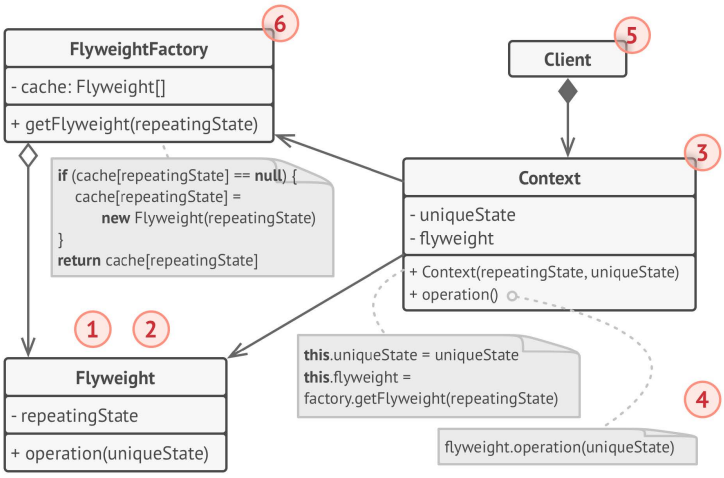
Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.









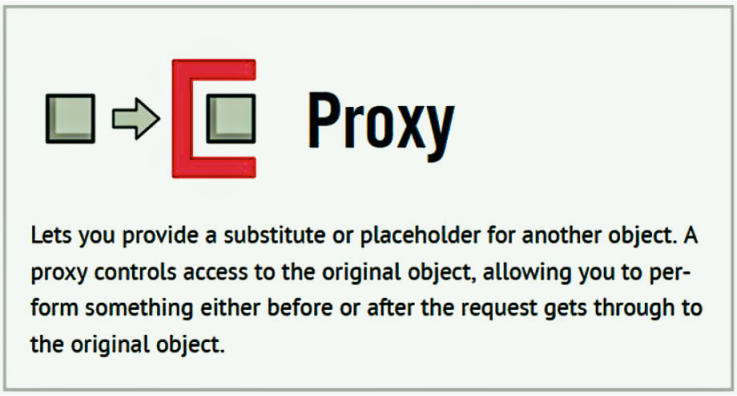


- Flyweight pattern can be used when we need to support a huge number of objects

- Flyweight + Composite
(when implement shared leaf nodes of Composite and need to save some RAM)

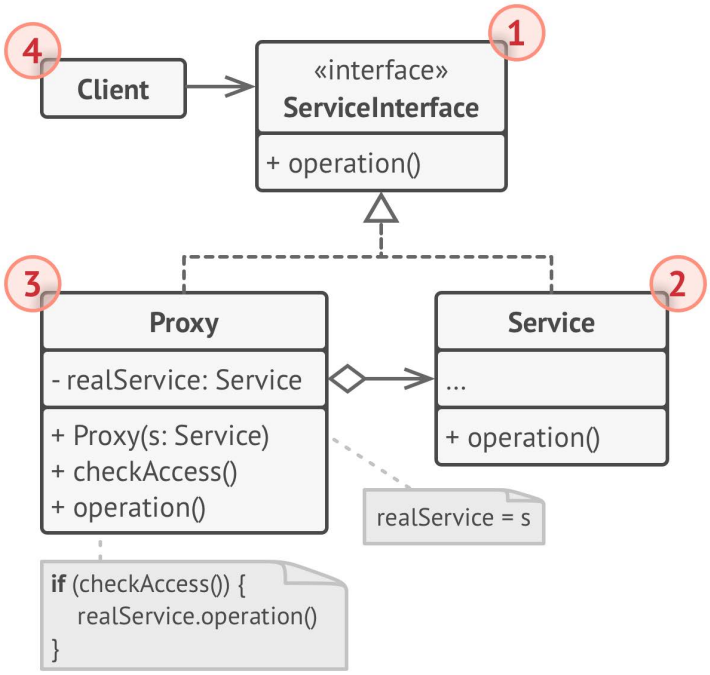
- Flyweight vs Facade
(shows how to make lots of little objs) (shows how to make a single obj that represents an entire subsystem)

- Flyweight vs Singleton
(multiple instance) (single object)
(immutable) (mutable)



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.

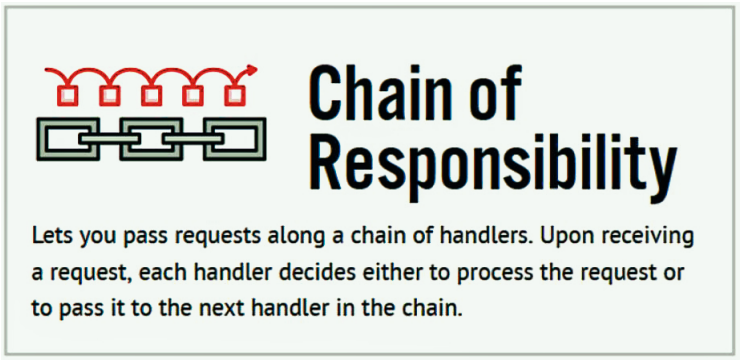


- Proxy (with same interface) vs Adapter (with different interface) vs Decorator (with enhanced interface)

- Proxy vs Facade
(with same interface which makes interchangeable)

- Proxy vs Decorator
Proxy = with the life cycle of its service object
Decorator = controlled by the client

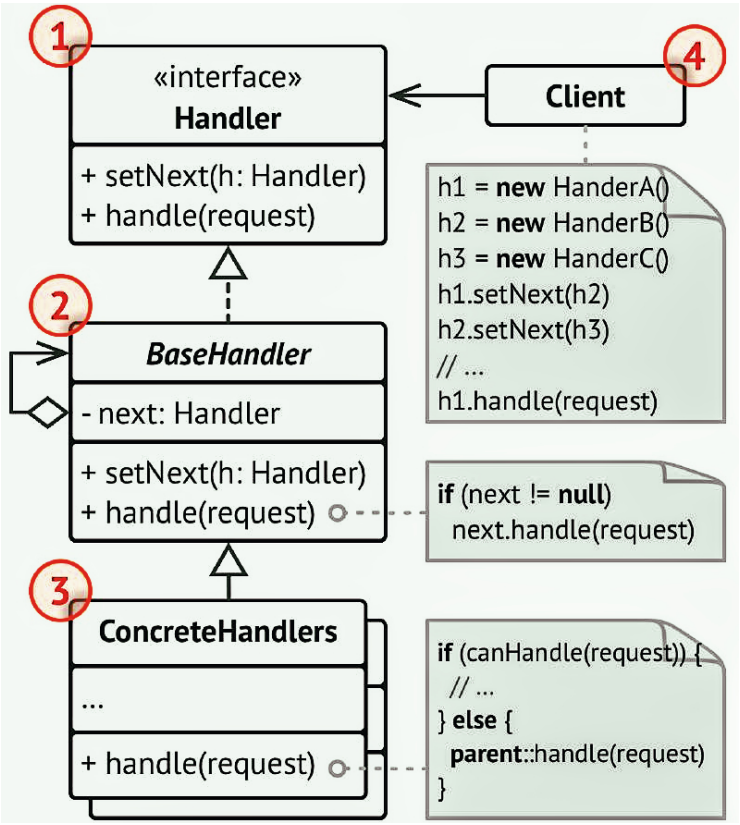
- Behavioral patterns = Chain of Responsibility + Command + Iterator + Mediator + Memento + Observer + State + Strategy + Template Method + Vistor
(C2,I,M2,O,S2,T,V)



CHAIN OF RESPONSIBILITY

Also known as: CoR, Chain of Command

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 vs Command vs Mediator vs Observer
  CoR = dynamic chain of potential receivers
  Command = unidirection between senders and receivers
  Mediator = indirect connections between senders and receivers via a Mediator
  Observer = dynamic subscribe and unsubscribe

- Chain of Responsibility + Composite

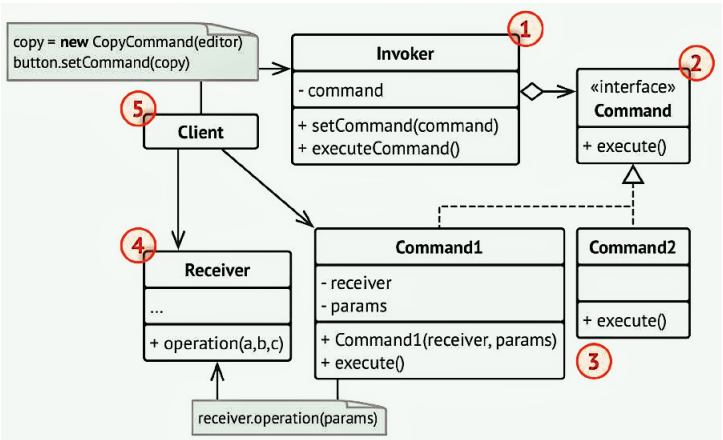
- Chain of Responsibility + Command

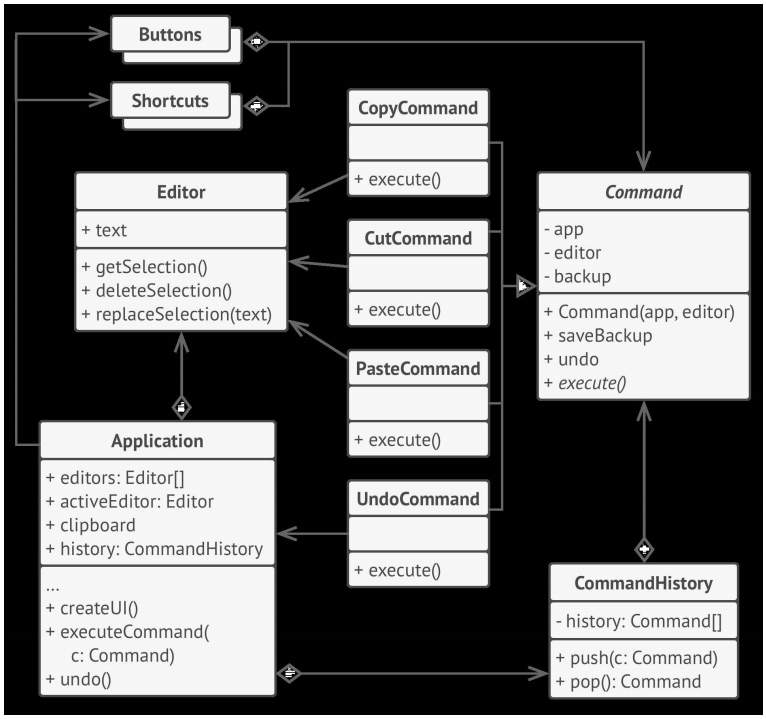
- Chain of Responsibility + Decorator

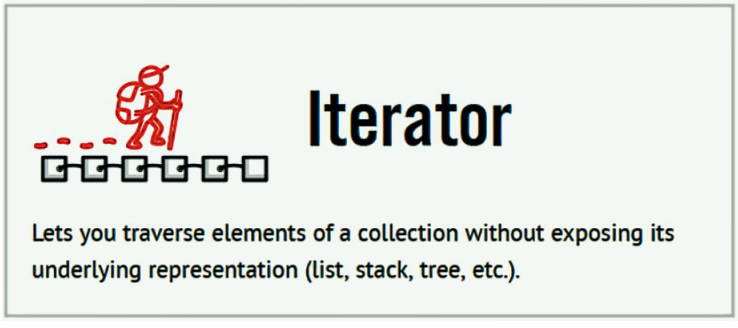
COMMAND

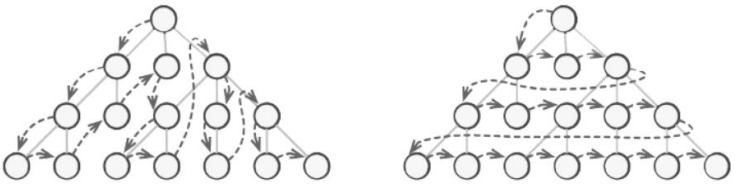
Also known as: Action, Transaction

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.





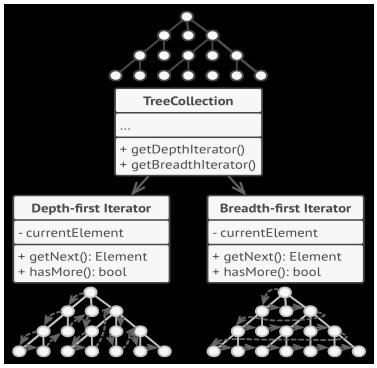


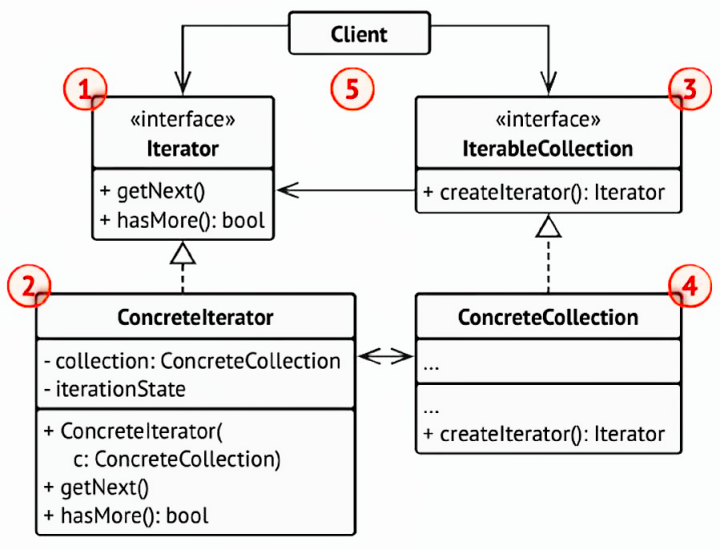


The same collection can be traversed in several different ways.

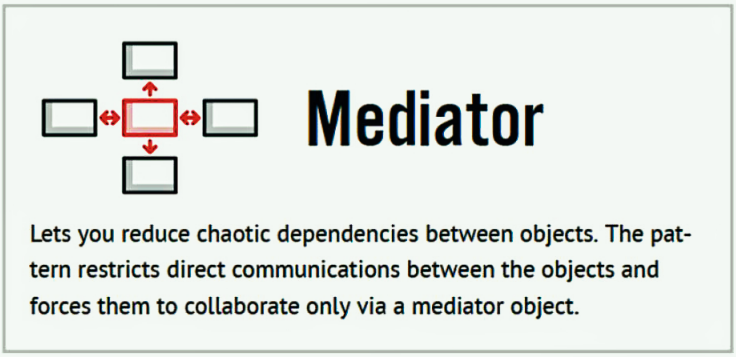
ITERATOR

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





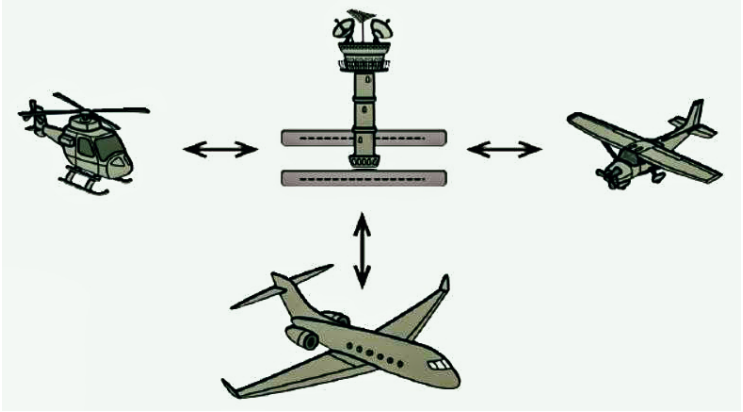
- Iterator + Composite
- Iterator + Factory Method
- Iterator + Memento
- Iterator + Visitor

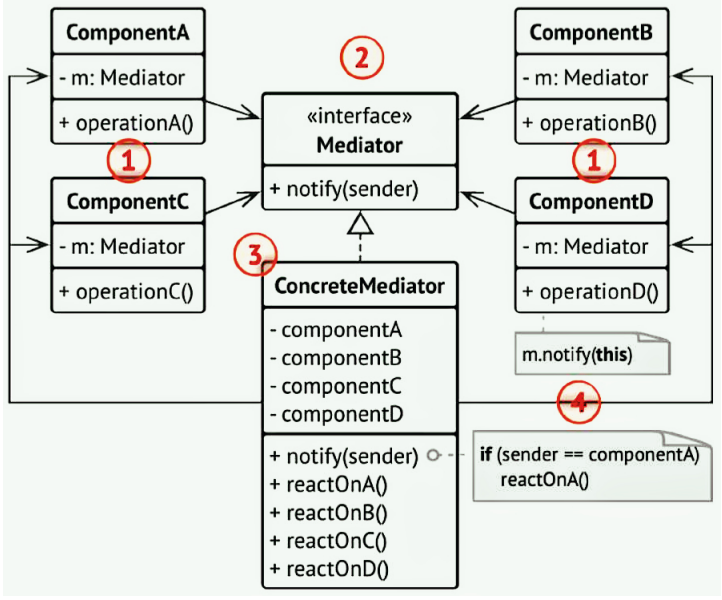


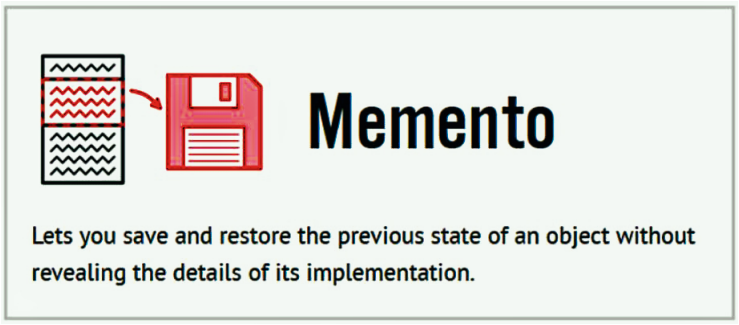
MEDIATOR

Also known as: Intermediary, Controller

Mediator is a behavioral design pattern that 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.



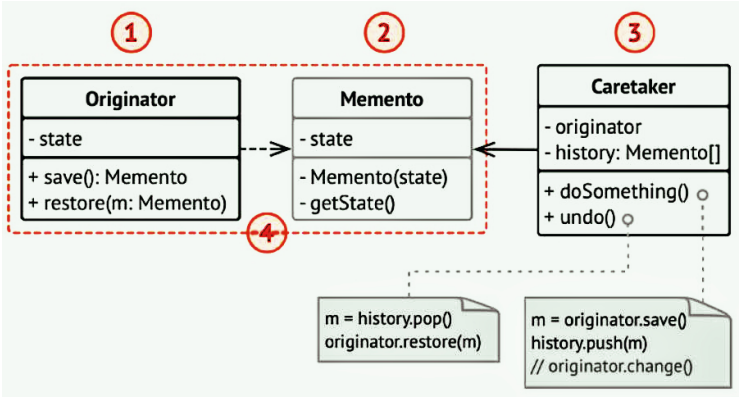




MEMENTO

Also known as: Snapshot

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

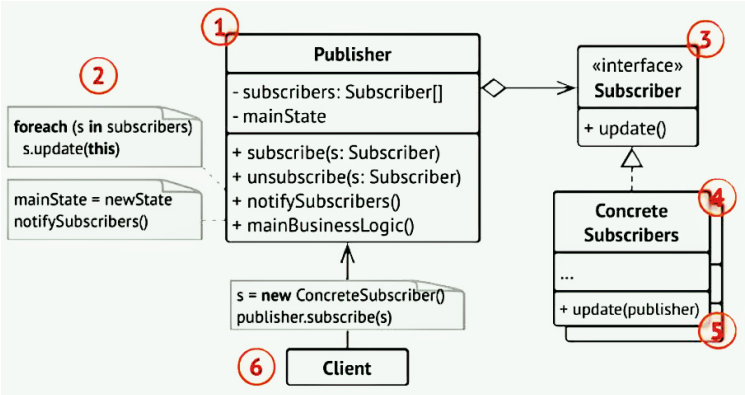


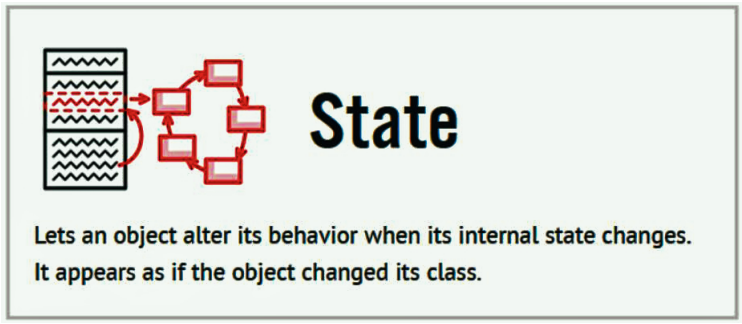
- Memento + Command
- Memento + Iterator
- Memento vs Prototype

OBSERVER

Also known as: Event-Subscriber, Listener

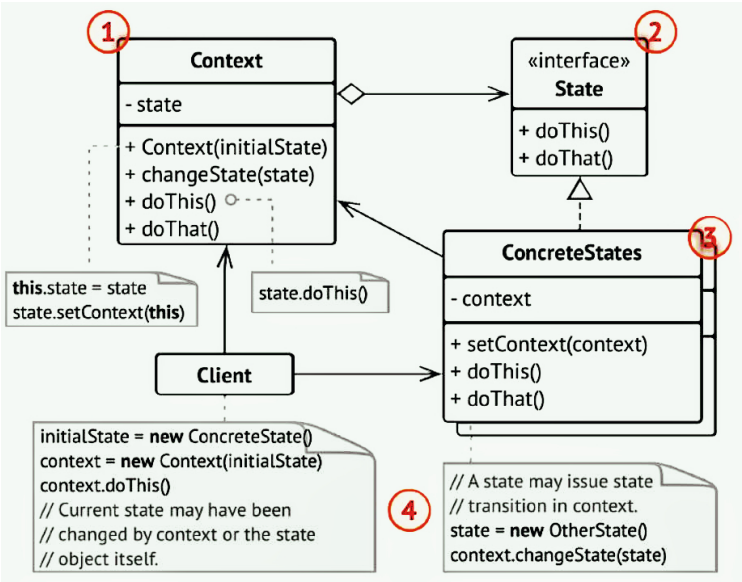
Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.





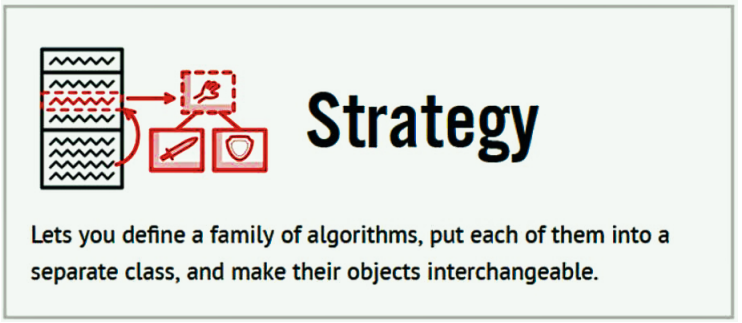
STATE

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.



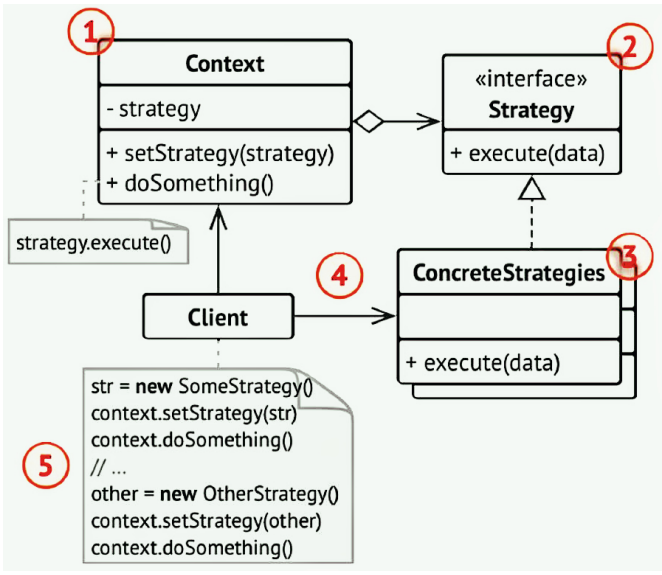
- Bridge vs State vs Strategy
  (similar structures)

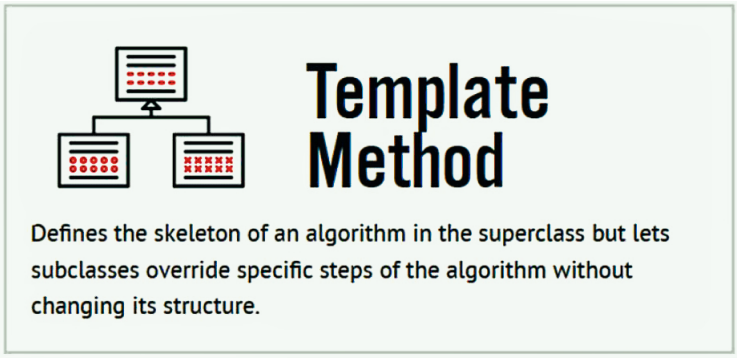
- State vs Strategy
  (State can be considered as an extension of Strategy)



STRATEGY

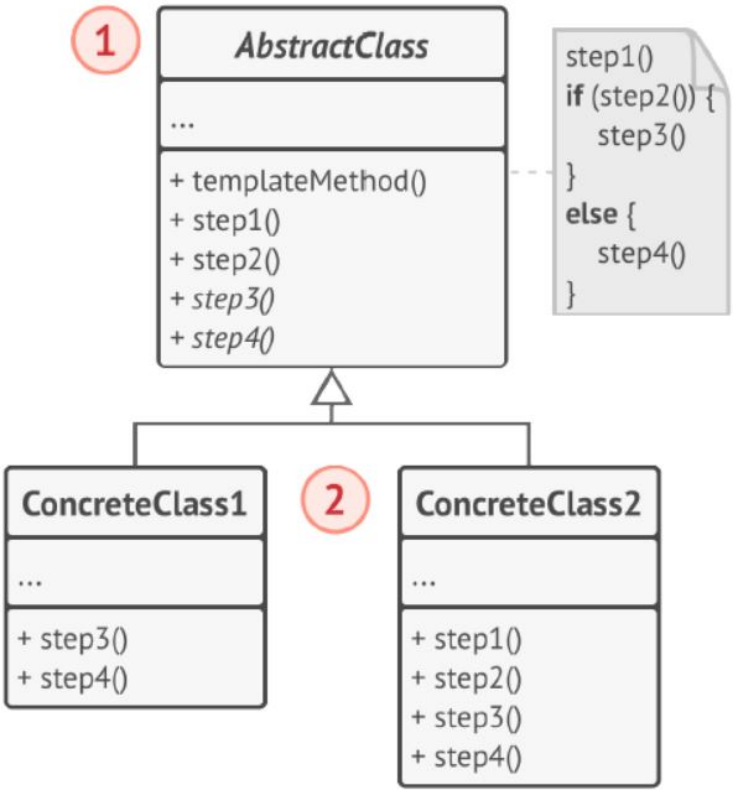
Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

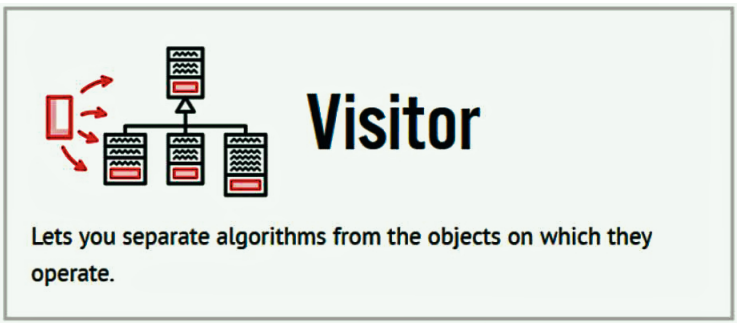




TEMPLATE METHOD

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





VISITOR

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