Design Patterns

DEVASHISH ROY
[GitHub : roydevashish]

Design Patterns

DESIGN PATTERNS

CREATIONAL

diff. way to create objects

STRUCTURAL

relationship b/w objects.

BEHAVIORAL

interaction or communication b/w objects.

Behavioral Design Pattern

Behavioral Design Pattern

- focus on how objects interact with each other and they communicate to accomplish specific tasks.
- address communication, responsibility, and algorithmic issues in object oriented software design.
- help in defining clear and efficient communication mechanisms b/w objects and classes.
- help in making the design more flexible, extensible, and maintainable by promoting better communication and separation of concerns b/w objects and classes in the system.
- each pattern addresses specific design issues and provides a standardized solution to common problems encountered in software development.

1. Memento
2. State
3. Strategy
4. Iterator
5. Command
6. Template Method
7. Observer
8. Mediator
9. Chain of Responsibility
10. Visitor
11. Interpreter

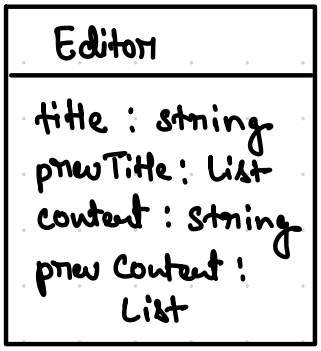
Memento Design Pattern

Memento Design Pattern

- used to restore an object to a previous state.

- common use case is implementing an undo feature. Ex - most text editor have undo feature where you can undo things by pressing some commands.

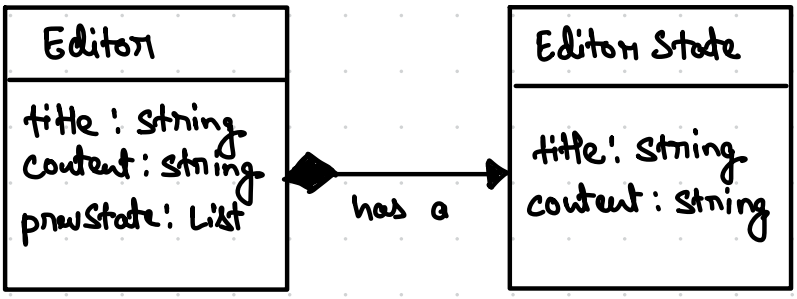
- simple way to implement
  - create a single Editor class and have a field for title and content, and also have a field that stores each of the previous value for each field in some list.



Problem:
- every time we add a new field, e.g. author, date, isPublished, we have to keep storing list of prev states (all the changes) for each field.

- how we implement the undo feature?
  If the user changed the title, then changed the content, then pressed undo, the current implementation has no knowledge of what the user last did - did they change the title or the content?

- Instead of having multiple fields in the Editor class, we create a separate class to store the state of our editor at a given time.

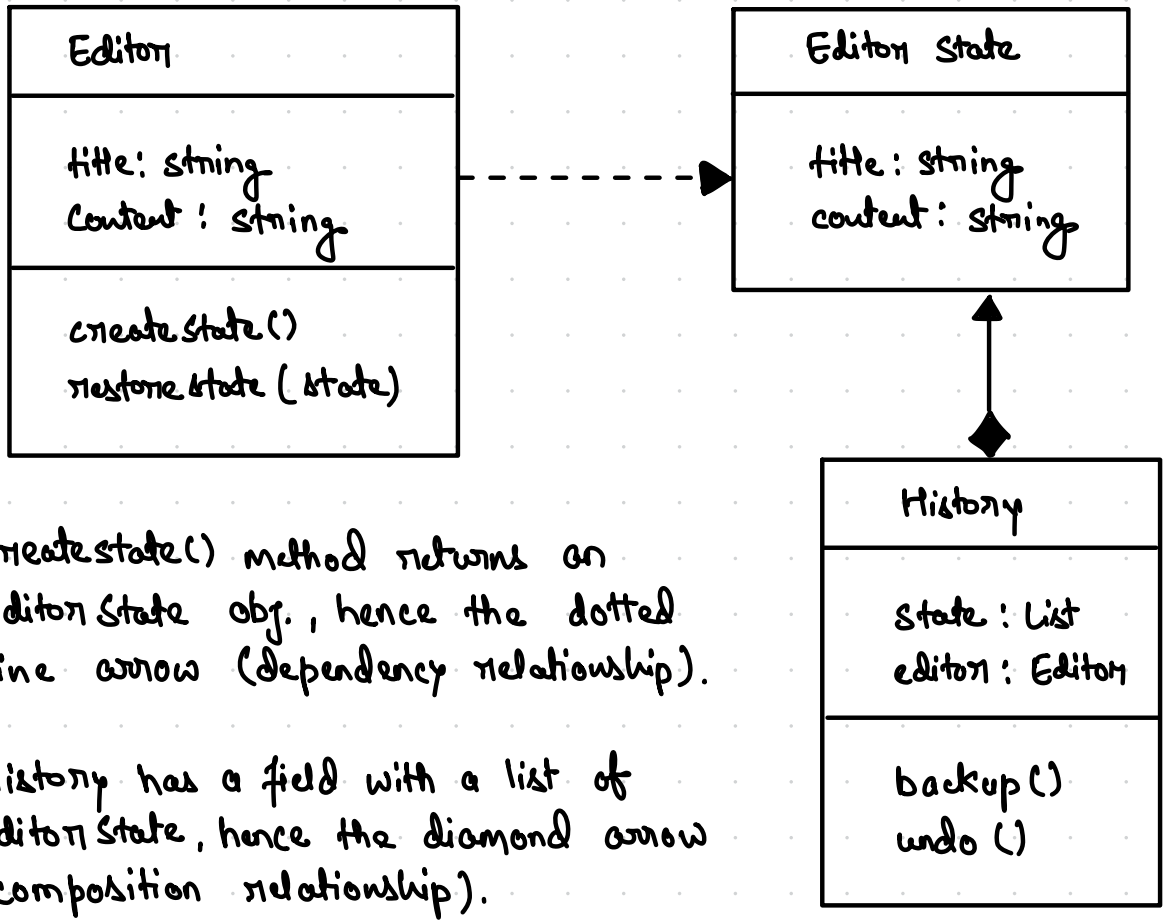


composition relationship:
Editor is composed of, or has a field of the EditorState class.

This is a good solution as
- we can undo multiple times and
- we don't pollute the Editor class with too many fields.

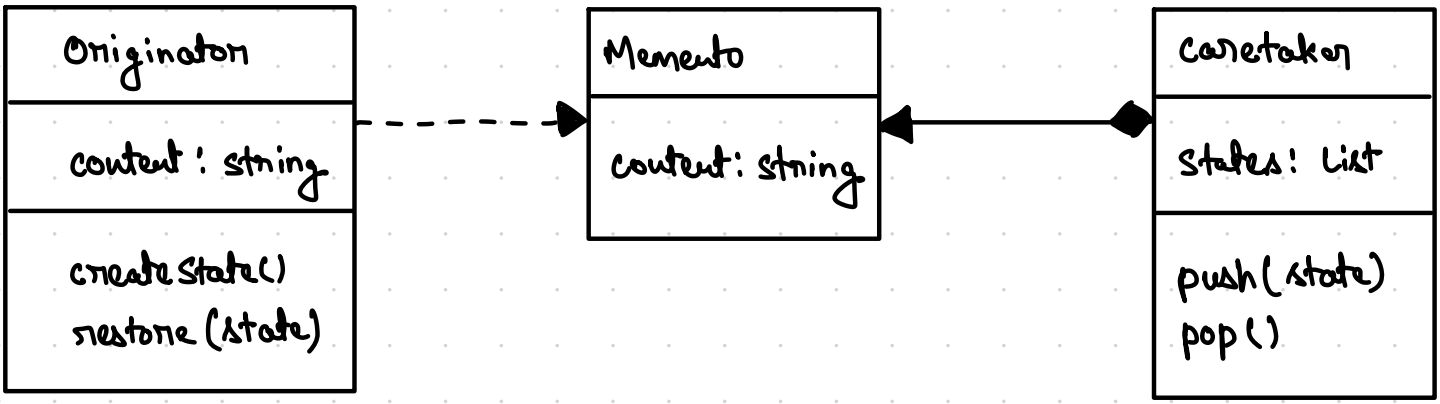
Problem: violating the SRP, as our Editor class has multiple responsibilities
1. State management
2. Providing the features that we need from an editor.

Solution: We should take all the state management stuff out of Editor and put it somewhere else.



- createState() method returns an EditorState obj., hence the dotted line arrow (dependency relationship).
- History has a field with a list of EditorState, hence the diamond arrow (composition relationship).

Memento Pattern with abstract names that each class would be in the memento pattern:



When to use: when you want to produce snapshots of an object's state to be able to restore the object to a previous state.

+ Can simplify the originator's code by letting the caretaker maintain the history of the originator's state, satisfying the SRP.

- App might consume a lot of RAM if lots of mementos are created. Eg. if we have a class that is heavy on memory, such as a video class, then creating lots of snapshots of video will consume lots of memory.

State Design Pattern

Stat Pattern

- State pattern allows an object to behave differently depending on the state that it is in.

Ex :- Writing a blog post, the post can be in one of three states :-
1. Draft 2. Moderation 3. Published.

There are three types of user roles :-
1. Reader 2. Editor 3. Admin
* Only admin can publish post.

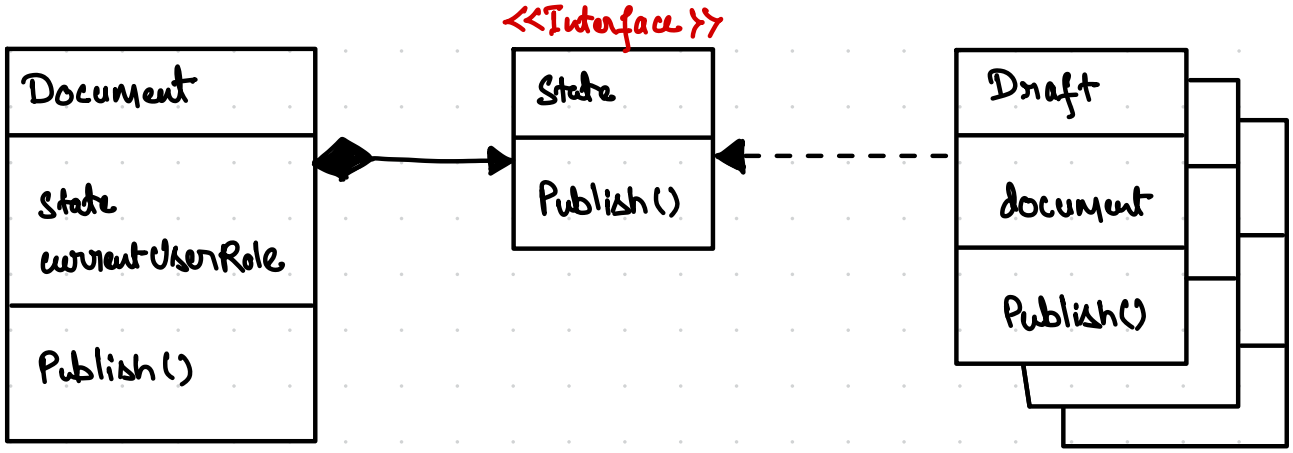
- Simple Solution :-
using if/else statements to check the current state of the post to see whether the state of the document should be upgraded.

- Solution with State Pattern.

It suggest that we should create state classes for each possible state of the Document object, and extract all state-specific logic into these classes.

The Document class will store a reference to one of the state classes to represent the current state that it is in.

Then, instead of Document implementing state-specific behaviours by itself, it delegates all the state-related work to the state object that it has a reference to :



- Document keeps reference to (is composed of) a State object.

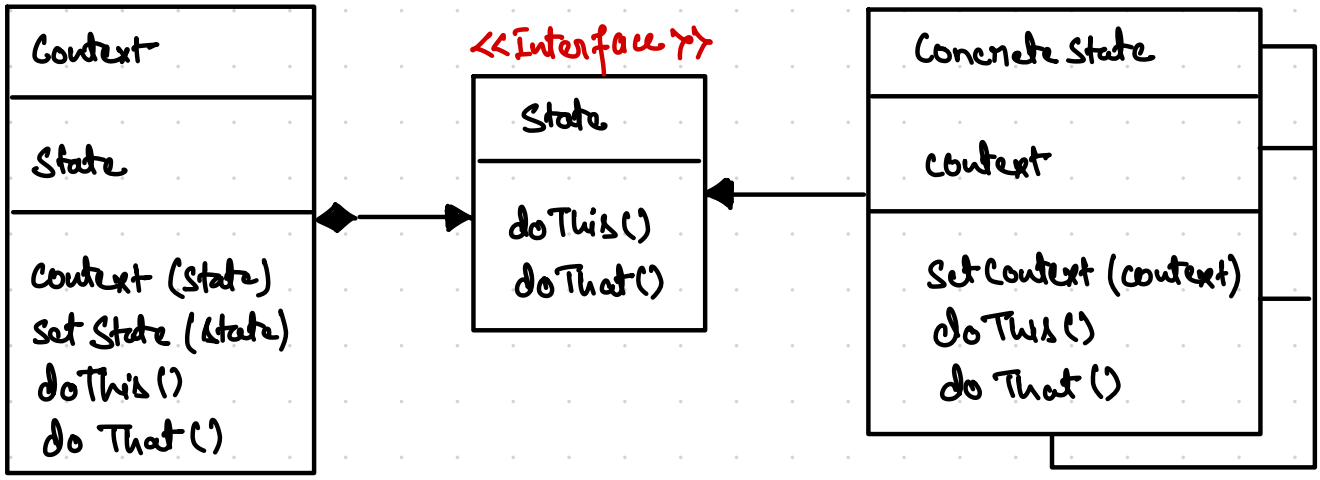
- We are using polymorphism, as the state field can be any one of the concrete state classes (Draft, Moderation, Published), as we are coding to an interface, not concrete classes.

- In Document, the Publish() method calls state.Publish() - it delegates the work to the concrete state object.

+ Our solution now satisfies the Open/Closed Principle:
if we want to add a new state, we create a new concrete state class that implements the State interface.

We extend our codebase (add new classes) without having to modify any current classes (Document in our case).

- State Pattern in GOF book:-



Strategy Design Pattern

Strategy Pattern

- Strategy Pattern is used to pass different algorithm, behaviours to an object.

- Consider an application that stores video. Before storing a video, the video need to be compressed using a specific compression algorithm, such as MOV or MP4, then if necessary, apply an overlay to the video, such as black and white or blur.

VideoStorage
compresson overlay

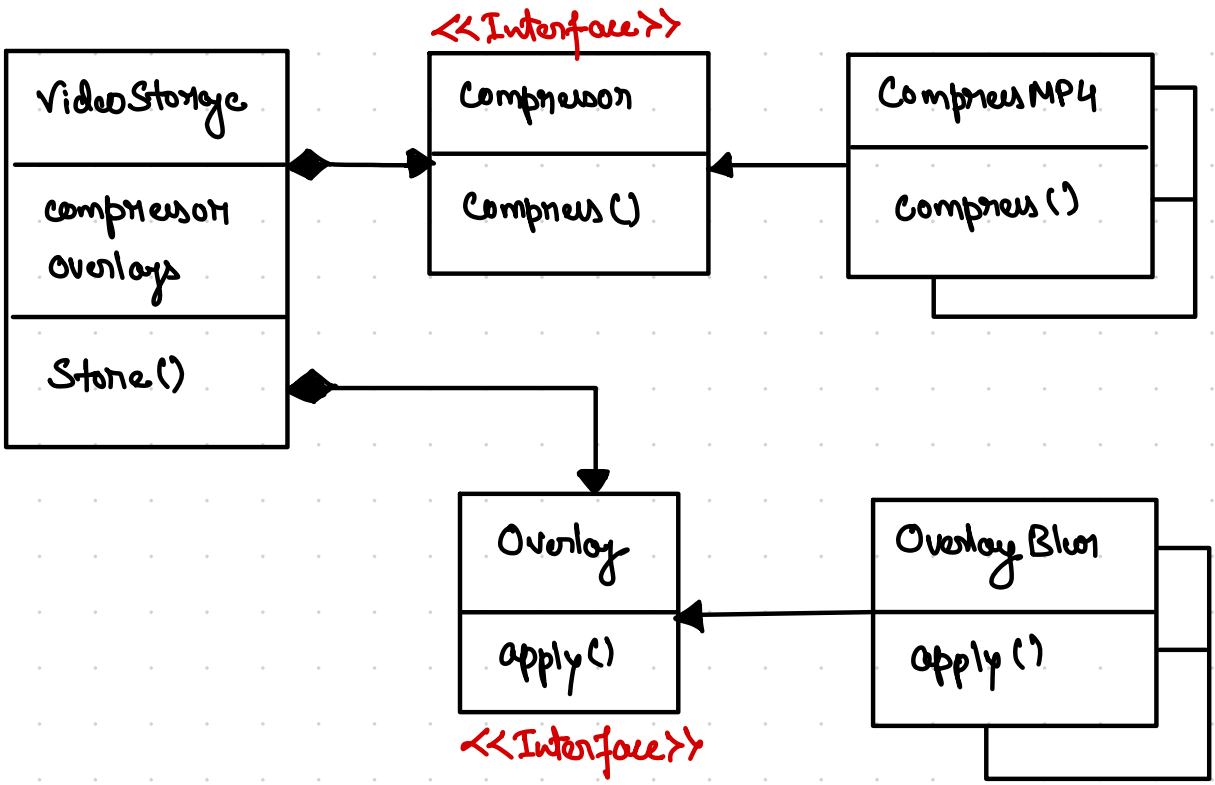
VideoStorage(compresson, overlay)
Set Compresson (compresson)
Set Overlay (overlay)
Store (filename)
↳ //compress
//apply overlay
// store

- violates open / close principle as we need other compression also or other overlays it will bloat with if-else.

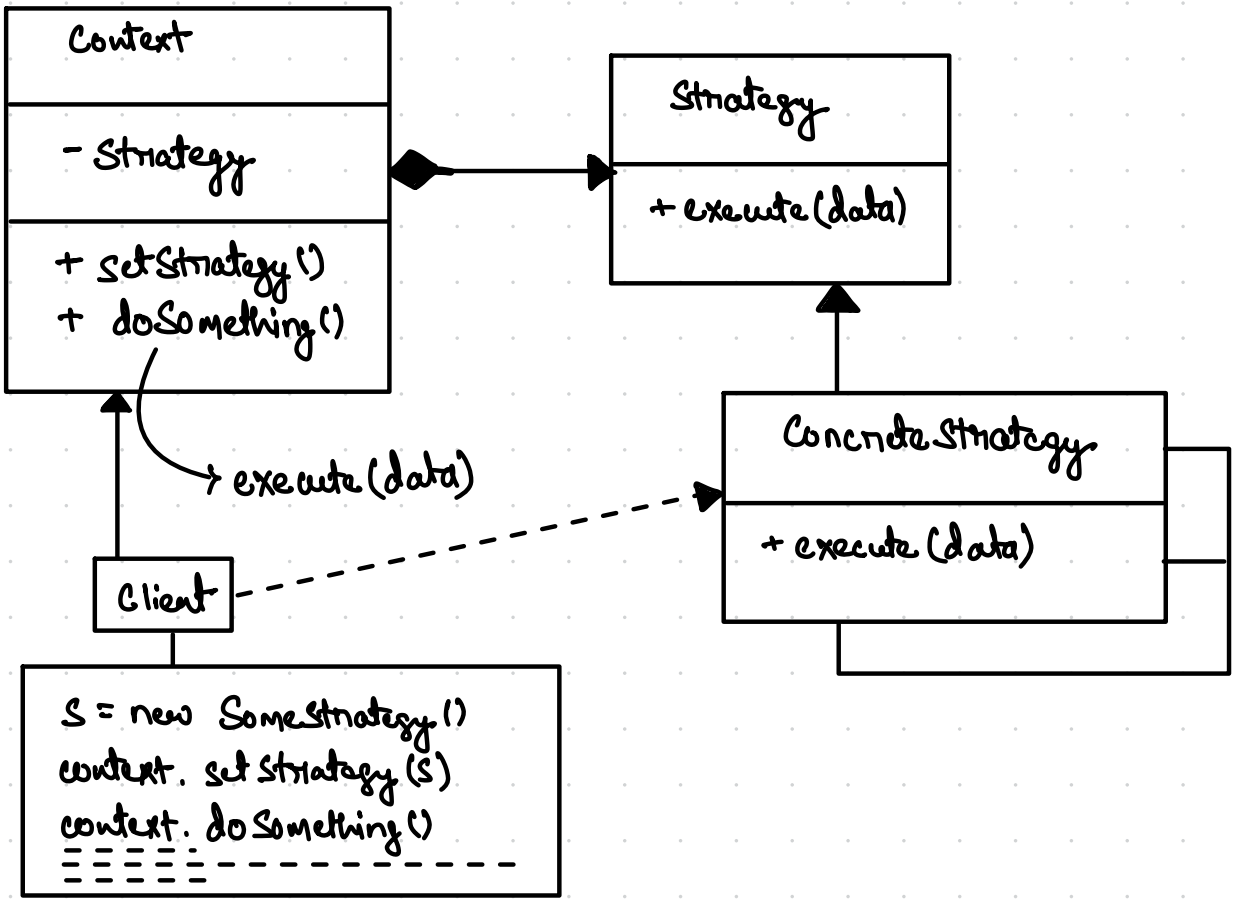
- We can create a Video Storage object, we pass it the concrete compresson and overlay objects that we want to use.

- This is polymorphism : VideoStorage can accept many different forms of compresson and overlay objects.

- VideoStorage is composed of Compresson and Overlay objects, and there are multiple concrete compresson and overlay implementation that extends Compresson and Filter / Overlay respectively.



- Strategy Pattern and its abstract names in GoF :-



Difference b/w Strategy and State.

- Two patterns are similar in practice, and the difference b/w them varies depending on who you ask:-
  - State store a reference to the context object that contains them, strategy do not.
  - State are allowed to replace themselves (to change the state of the context object to something else), while Strategies are not.
  - Strategies only handle a single, specific task, while State provide the underlying implementation for everything (or most everything) the context object does not.

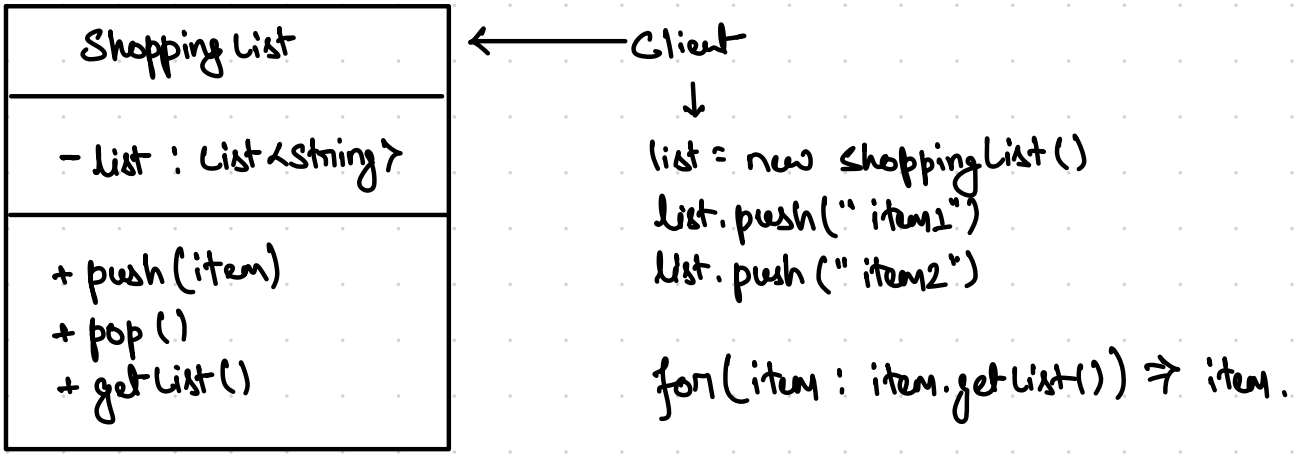
When to use:-
- When you have a class with a large no. of conditional statements that switch b/w variant of the same algorithm.
- The algorithm logic can be extracted into separate class that implement the same interface.
- The context object then delegates the work to these classes, instead of implementing all algorithm itself.

+ Satisfies open/close principle: can add new strategies without modifying the context.
+ Can swap algorithms used inside an object at runtime.
- Client have to aware of the different algorithms and select the appropriate one.
- Having only few algorithms that rarely changes, then using the Strategy Pattern may be over-engineering.

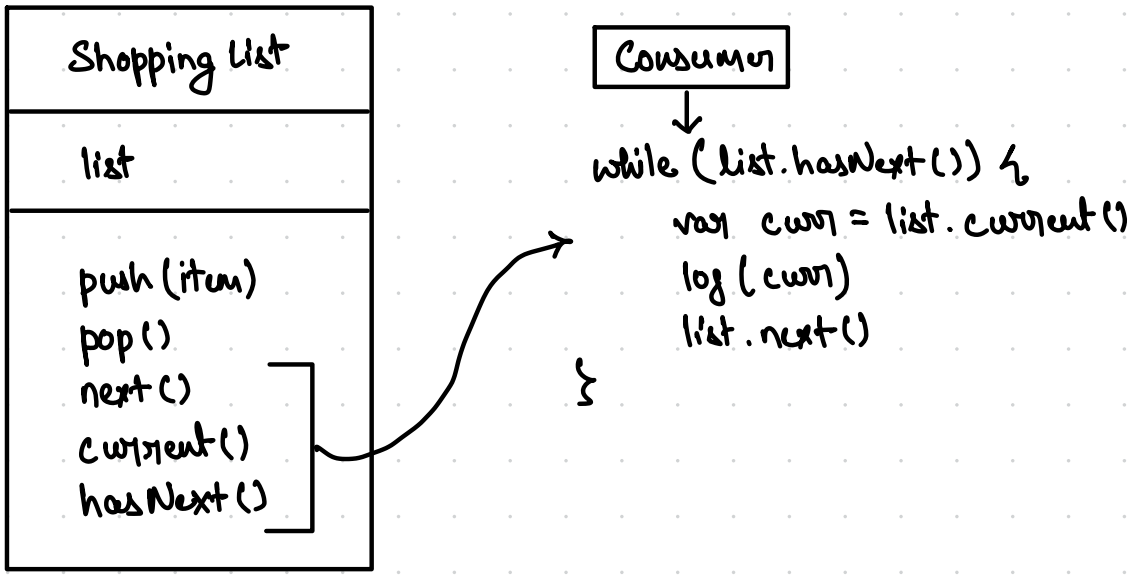
Iterator Design Pattern

Iterator Pattern

- Provides a way of iterating over an object without having to expose the objects internal structure, which may change in the future.
- Changing the internals of an object should not affect its consumer.
- Example: Shopping List contains the items in string.



- We can use the Iterator Pattern to ensure that changing the internals (changing the list data structure) doesn't affect consumers. We can add some methods to ShoppingList to allow iterating over a shopping list object, without knowing its internal representation.



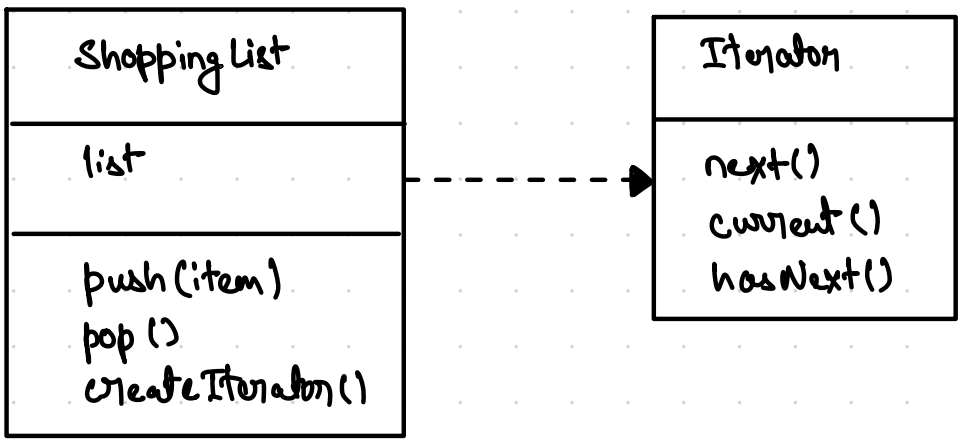
- added three new methods to help consumer to iterate over the object, without knowledge of internal data structure.

  - next() : goes to the next item
  - current(): returns the current item
  - hasNext(): checks if there is another item.

- With this, we don't know the internal representation of the list object, so, if we changed the data structure used in shopping list to store items, its consumers wouldn't break on need to be changed. We'd just have to perhaps update the iterator methods to account for the new data structure.

- Problems: the above class violates the SOLID single responsibility principle
  1. It's responsible for list management, using push() and pop().
  2. It's responsible for iteration, using next(), current() and hasNext().

- Fix: To follow the SRP, we can put the iterator methods into a new class.

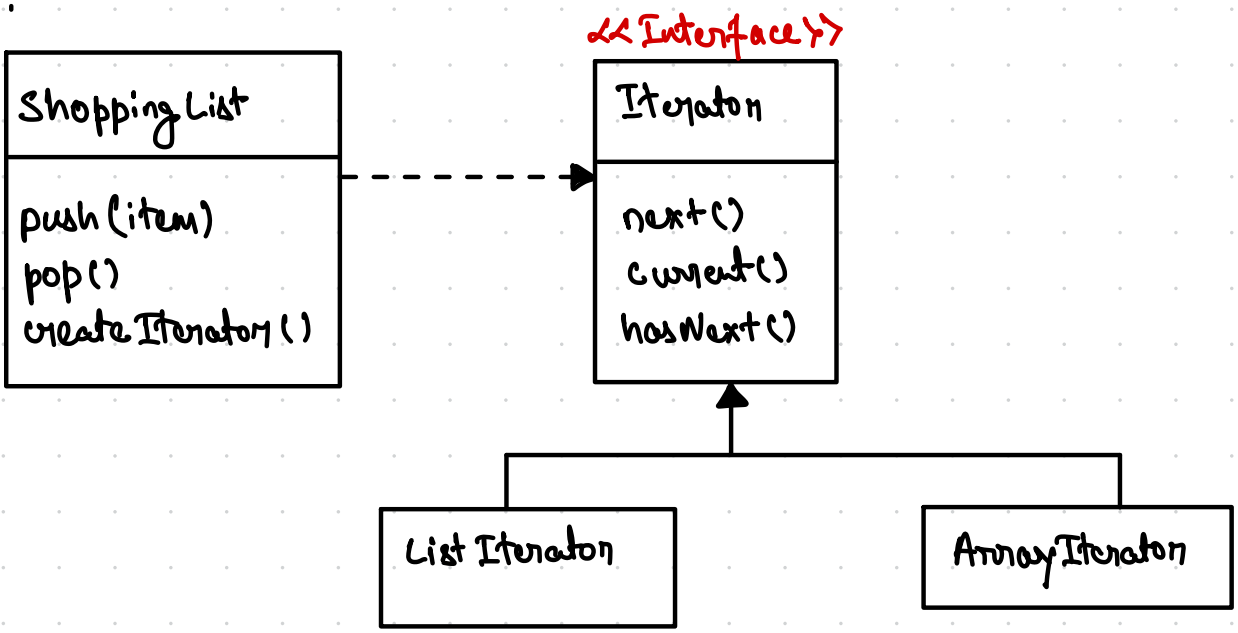


- The createIterator() method returns an instance of Iterator that allows consumer to iterator over shopping lists without knowing internal details.

Problem: If the data structure in ShoppingList changes, then we will need a different Iterator to manage it. So, Iterator needs to be an interface, and then we can have concrete classes for each data structure that Implement Iterator to ensure the contain the iterator methods.

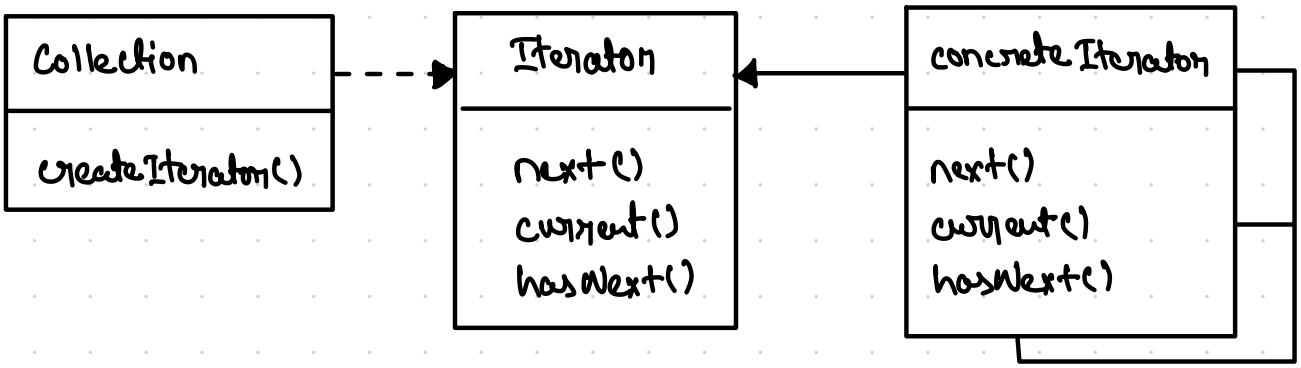
The interface determines the capabilities we need from a real concrete iterator. The data structure could be Array, List, Stack etc.

Solution:



- ShoppingList has a dependency to the Iterator interface, as createIterator() returns an object of type Iterator. The concrete iterator classes extends Iterator and implements its methods.

- Abstract Class Name in GoF.



When to use:

- when your collection possesses a complex internal data structure, or a data structure that is likely to change, so that the client can iterate over the collection without any knowledge of the data structure.

+ Satisfies SRP: traversal logic is abstracted into separate classes.

+ Satisfies Open/Close principle: you can create new collections and iterators without breaking the code that uses them.

- Can be overengineering if your app only works with simple collections.

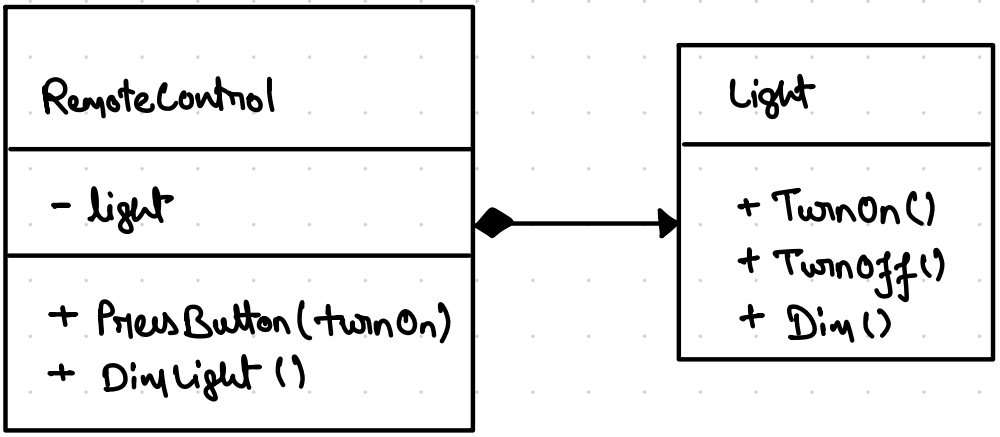
Command Design Pattern

Command Pattern

- It is a behavioral design pattern that encapsulates a request as an object, allowing you to parameterize clients with queues, requests, or operations.

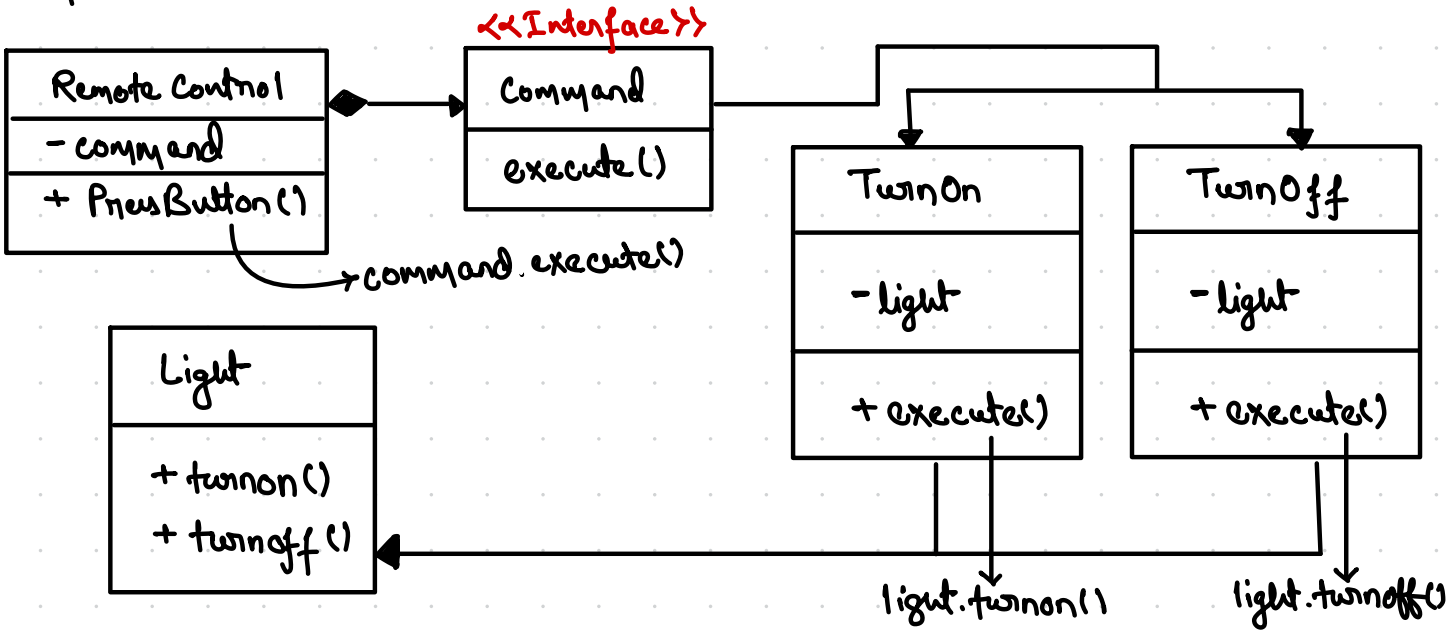
- It decouple the sender from the receiver, providing flexibility in the execution of commands and supporting undoable operations.

Example: Light (receiver) that can be controlled by Remote (invoker).



- Light is tightly coupled to RemoteControl. If we want to add new features then we have to modify RemoteControl.

- Command Pattern UML:-

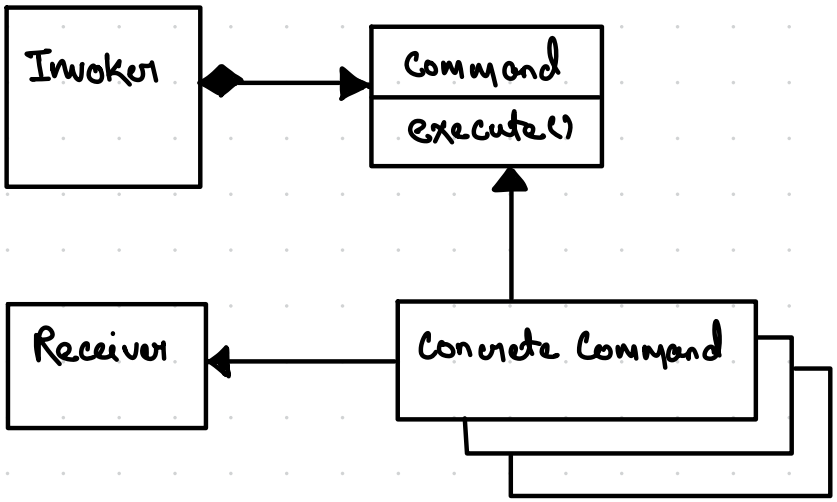


- Using the Command Pattern, we decouple RemoteControl (the sender) from Light (the receiver).

- To add new functionality, such as dimming the light, we can extend our codebase by adding a new Dim command, without having to modify RemoteControl.

- The RemoteControl is composed of Command. The concrete commands, TurnOn, TurnOff, implement Command, store a reference to Light.

- GoF UML

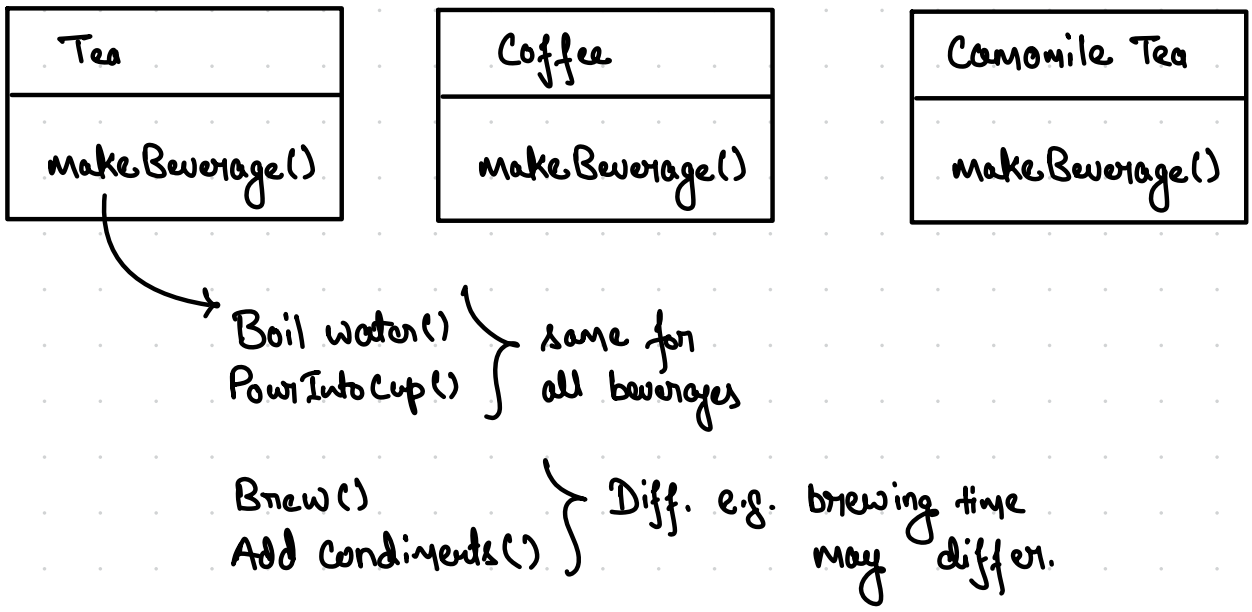


Template Method Design Pattern

Template Method Pattern

- It allows to define a template method, an skeleton, for an operation. The specific steps can then be implemented in subclasses.

- Ex: We have a machine that makes hot beverages. At the beginning we just had tea and coffee. But after some time, we need to add some more beverages, such as camomile tea:



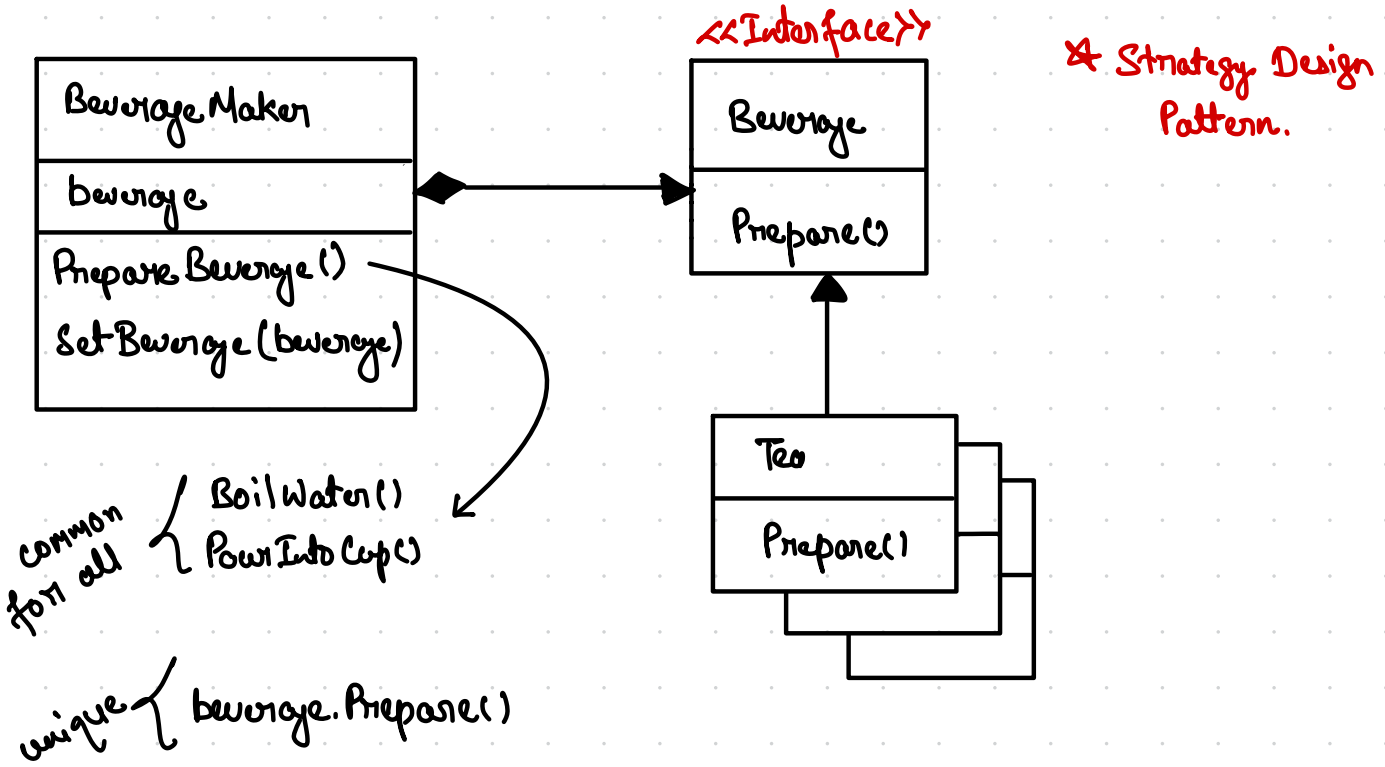
- We started out simple, making a separate class for each hot beverage.

- But as the no. of beverages grows, we see a lot of code duplication.

- We also have no way of ensuring that each class follows a particular structure, which means that the client code will have lots of conditionals that pick the proper course of action depending on the particular beverage class.

- There are two good ways to solve this issue of code duplication
  1. Polymorphism
  2. Inheritance.

1. Polymorphism



* Strategy Design Pattern.

- We provide a common Beverage interface to force all Beverage to follow a specific structure.
- We then have a BeverageMaker class that manages preparing different beverages. This class includes the common operations for making all beverages, such as boilWater and pour into cup, and also call the operations specific to each beverage, which is delegated to Beverage.
- Now we can create a new beverage, we only have to include code specific/unique to that beverage.

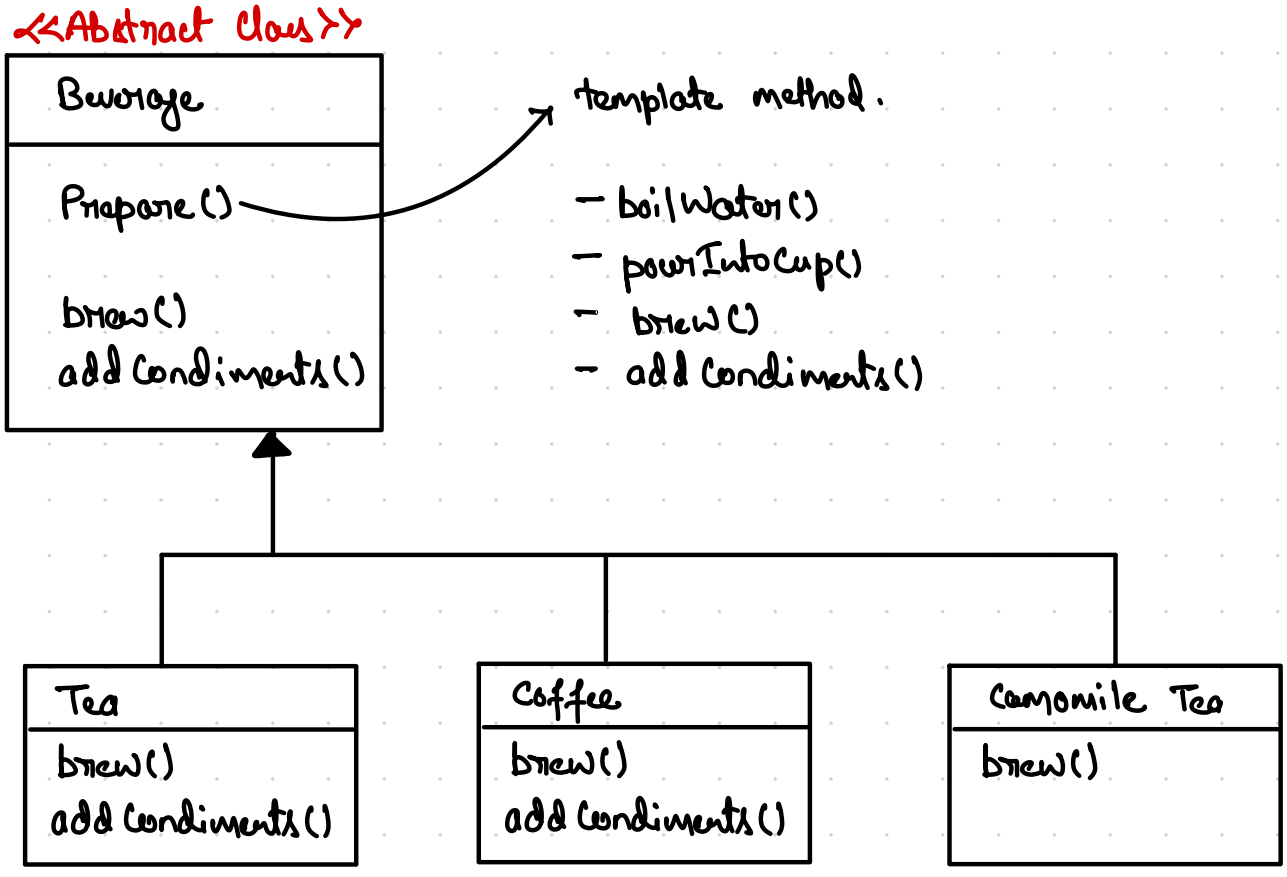
2. Inheritance

- Tea, Coffee and Camomile Tea have things in common, so we can create an abstract Beverage class to implement the Prepare() method.

- After we boiled the water and poured it into a cup, what happens next is unknown in the abstract Beverage class, as it depends on the particular beverage.

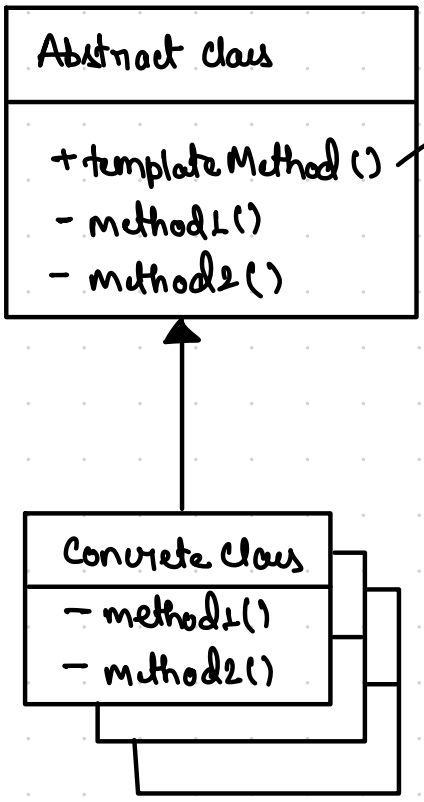
- These beverage-specific steps will be determined later on, when the beverage class is extended.

- We can provide a base abstract class called Beverage that contains all common operations for making a beverage, and we can provide methods, brew() and addCondiments(), which can be implemented / overridden in the concrete beverage classes.



- This is the template method pattern:
  The Beverage class has a template methods that provides the common setup and structure for preparing a beverage.

- Template Method Pattern in GoF book:-



this.method1()
this.method2()

- we have a abstract class with a concrete implementation of the common/shared templateMethod().

- the abstract methods that will be implemented within the concrete classes can be used to alter the behaviour of the template method.

- we can also give the primitive operations a default implementation, and leave it up to the subclasses to either take them as they are, or override them.

- In this case, we refer to these methods as "hooks", or "hook operations".

Template Method Vs. Strategy Pattern

- Template Method

  - When you have an algorithm with a fixed structure but with certain steps that need to be customized or implemented differently by subclasses.

  - Ideal when you want to define a common algorithm skeleton (template method) in a base class and allow subclasses to selectively override specific steps to provide their own implementations.

  - Suitable when the overall algorithm structure remains consistent, but specific parts of the algorithm can vary based on different requirements or context.

- Strategy Pattern

  - When you want to define a family of interchangeable algorithms or behaviors and encapsulate each algorithm into its own class.

  - Ideal when you want to dynamically select and switch b/w different algorithms at runtime, depending on the situation or context.

  - Suitable when you want to decouple the client code from specific algorithm implementations, allowing greater flexibility and extensibility.

Summary :-

* If you primarily need to customize or override specific steps of an algorithm while keeping the overall structure intact, the Template Method Pattern is a good choice.

* If you need to encapsulate entire algorithms or behaviours as inter-changeable components that can be dynamically selected or replaced, the Strategy Pattern is more appropriate.

When to use:-

- When you want to allow clients to implement only particular steps in an algorithm, and not the whole algorithm.

- When you have a bunch of classes with the same logic, or algorithm, but with difference in a few steps.

- So, if the algorithm changes, it only has to be modified in one place - the base class.

+ Reduce code duplication.
+ Clients are only allowed to modify certain steps in an algorithm, reducing the risk of breaking clients if the algorithm changes.

- Some clients may be limited by the provided template.
- Template methods can be hard to maintain if they have lots of steps.

Observer Pattern

Observer Pattern

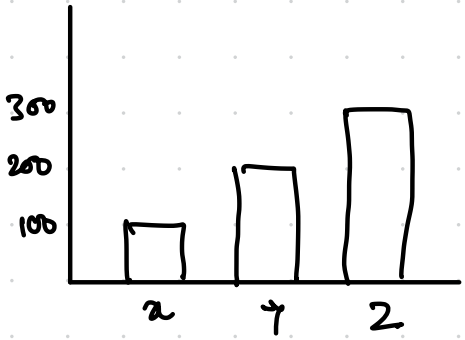
- The observer pattern involves an object, known as the subject, maintaining a list of its dependent objects, called observers, and notifying them automatically of any state changes.

- Situation

Data Source

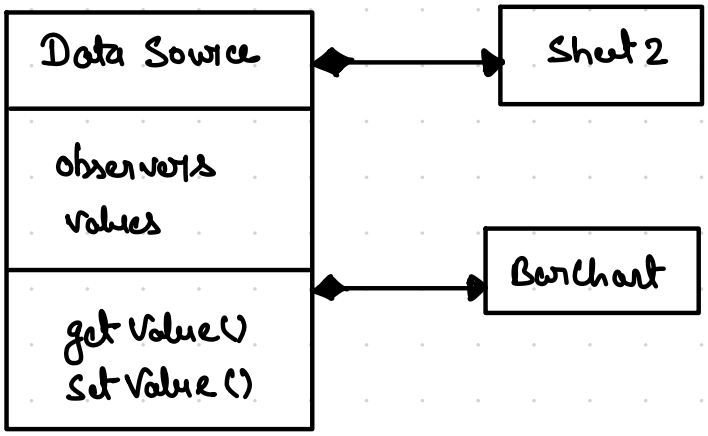
City	No. of dogs.
x	100
y	200
z	300

BarChart



Sheet 2

Total Dogs : 100 + 200 + 300 = 600



Problems:-
tight coupling to concrete observers.
i.e. we need conditions to check the object type before updating.

- SRP : DataSource has two responsibilities : Storing data and managing dependent observer objects.

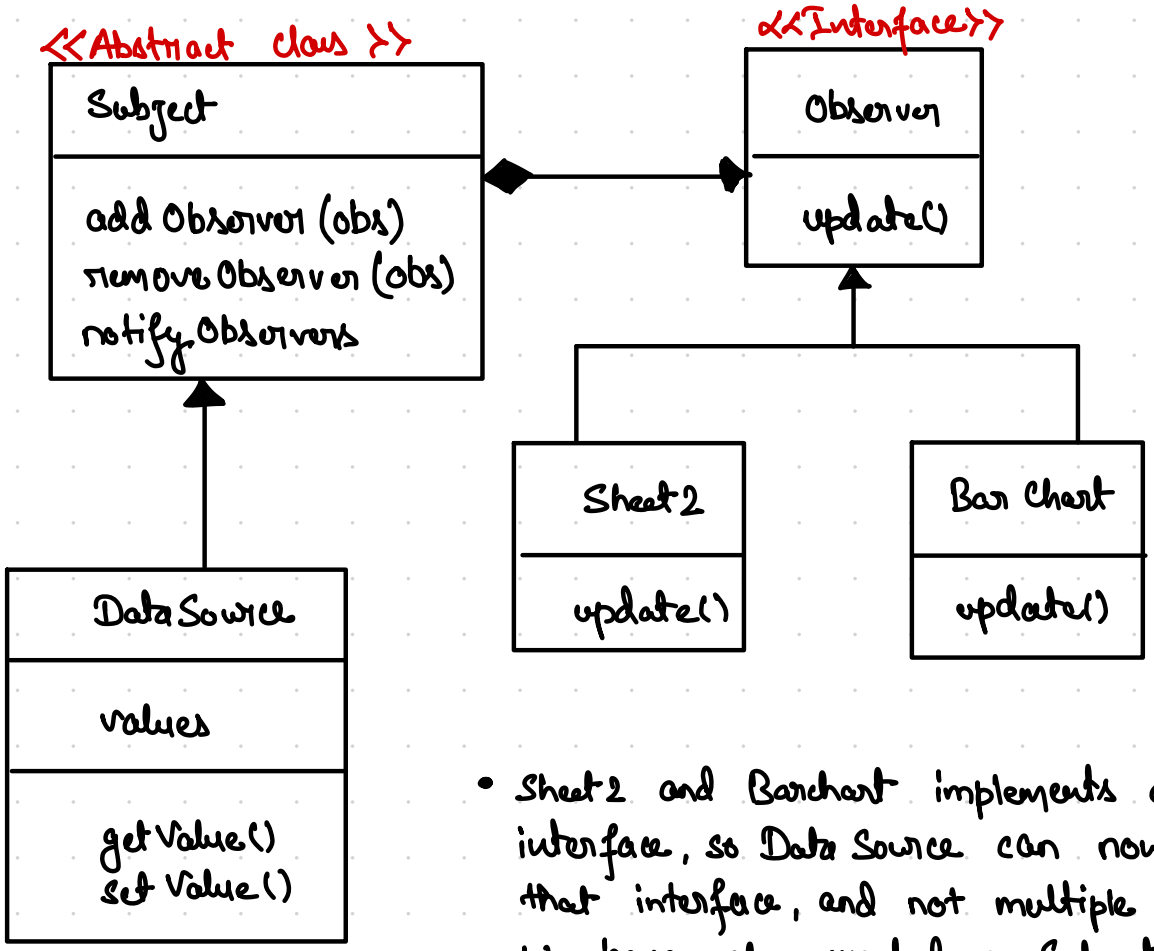
- OCP : every time we create a new observer object, we have to modify DataSource. This is because we are programming to concrete objects, rather than to a generic interface.

Solution.

- To solve SRP violation, we could create a separate class for managing the dependent observer objects.

- To solve the OCP violation, we can ensure that all observer object implement a common interface so that they provide consistent methods, allowing us to use polymorphism in DataSource.

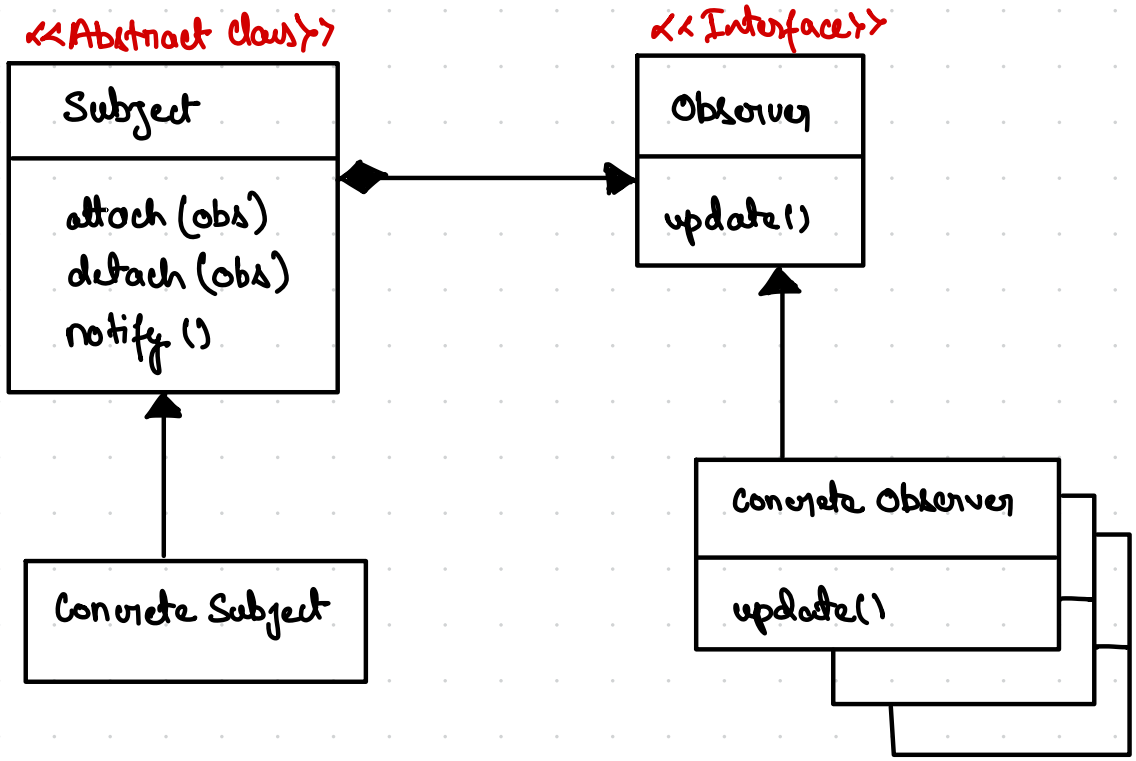
Observer Pattern.



- Sheet2 and Barchart implements a common interface, so DataSource can now talk to that interface, and not multiple concrete classes. We have also created a Subject class to provide the methods for managing observers.

- setValue(values) will loop through all of its observers and call update() on each. This is polymorphic behavior: a different update() method will be called depending on the observer but DataSource doesn't need to know what the specific concrete observers are. Each concrete implementation fig. out how to update themselves.

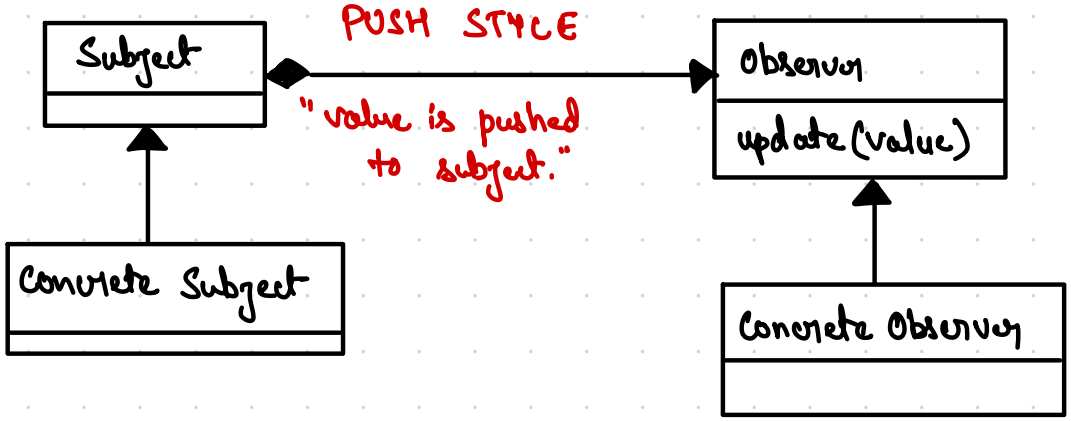
- GoF Observer Pattern



- The observer Pattern is AKA the pub and subscribe pattern: the subject (publisher) publishes changes in its states, and the subscribers (observers) subscribe to those events.

Communication Styles

- Above, the observers get notified of a change, but they don't know what has changed. One solution is to add a parameter to the observer update() method. This is known as a "push" style of communication, as the subject pushes the changes to the observers.

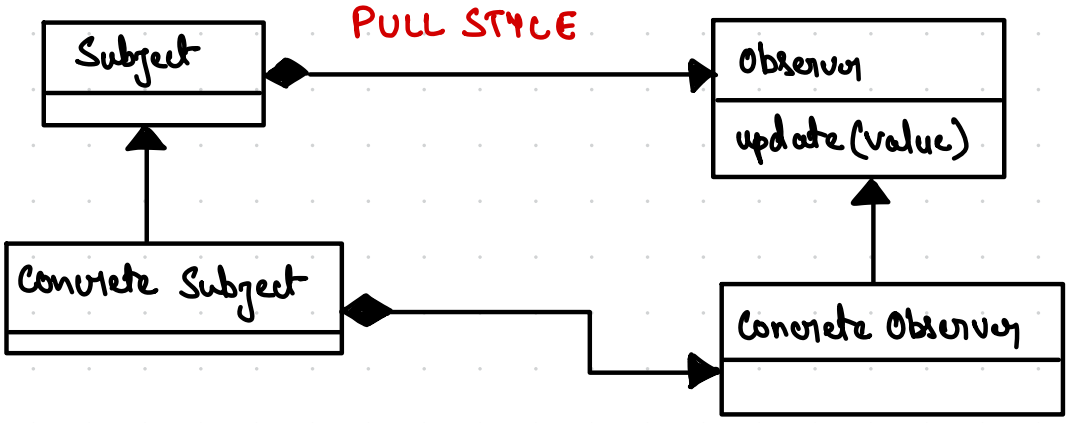


- Value could be any object or generic type.

- Advantage that Concrete observer doesn't depend on (has no knowledge or coupling to) the concrete subject.

Problem: What if each observer needs a diff set of values?

- We can use Pull style of communication, where the observer stores a reference to the concrete subject, then whenever it is notified of a change, it pulls, or queries, the data it needs from the concrete subject.



Value is "pulled" from concrete subject.

- Concrete observer store a reference to the concrete subject.
- We give concrete subjects a getValue() method, so a concrete observer can get the data it needs.

- This gives more flexibility, however, we have coupling b/w Concrete Subject and ConcreteObserver, because these observers could change in the future, and we may introduce more observers -- and ConcreteSubject would have to keep reference to them all. We don't want to change our concrete subject class (Data Source in our example) every time there is a new observer.

- In reality, we never have zero coupling in software. What matters is the direction of the relationship.

- With pull style communication, we pass the concrete subject to the observer object.

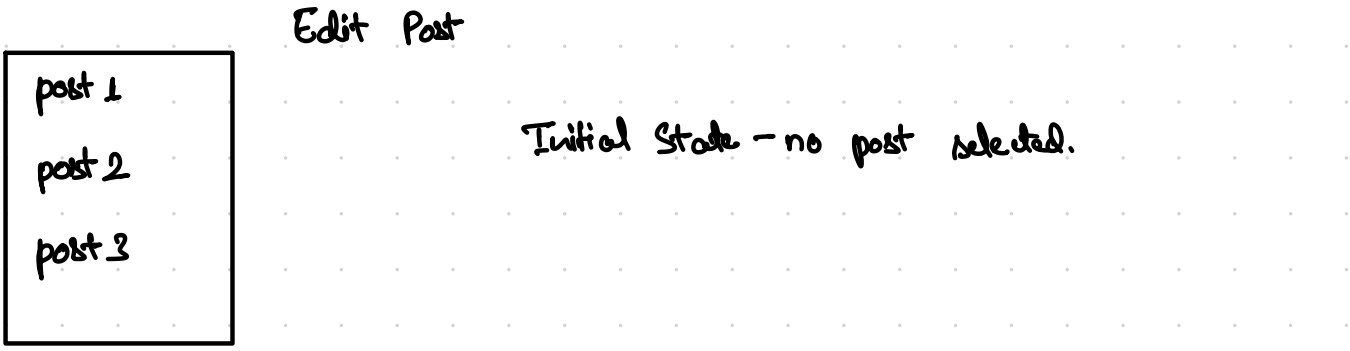
Mediator Pattern

Mediator Pattern

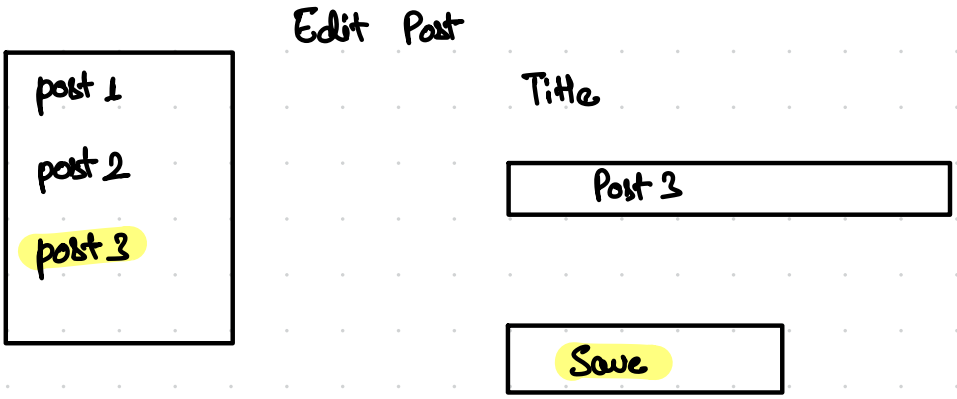
- It defines an object (the Mediator) that describes how a set of objects interact with each other, therefore reducing lots of chaotic dependencies b|w those objects.

Example: We have a blog that lists all of your posts, and you can select a post and then edit that post title.

State 1:

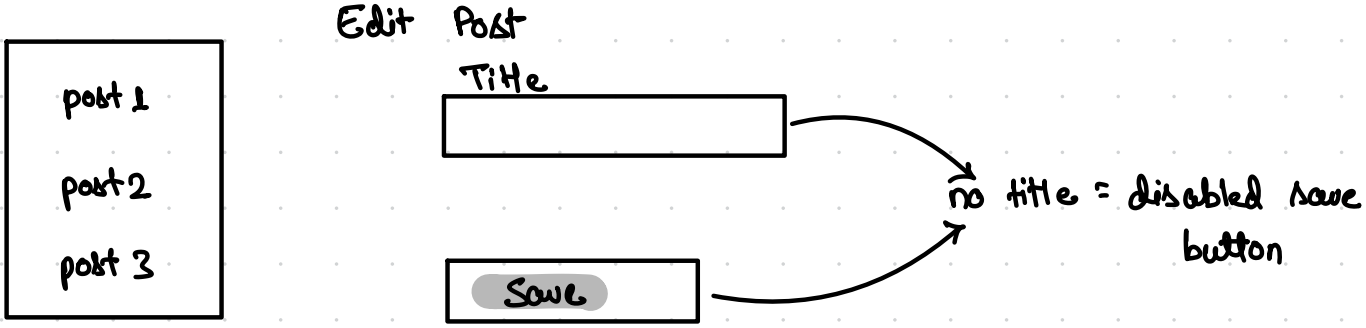


Select an article from the postsContainer on left and the input is populated on right with the title.



The save button is disabled if no title provided, or no article selected.

State 3



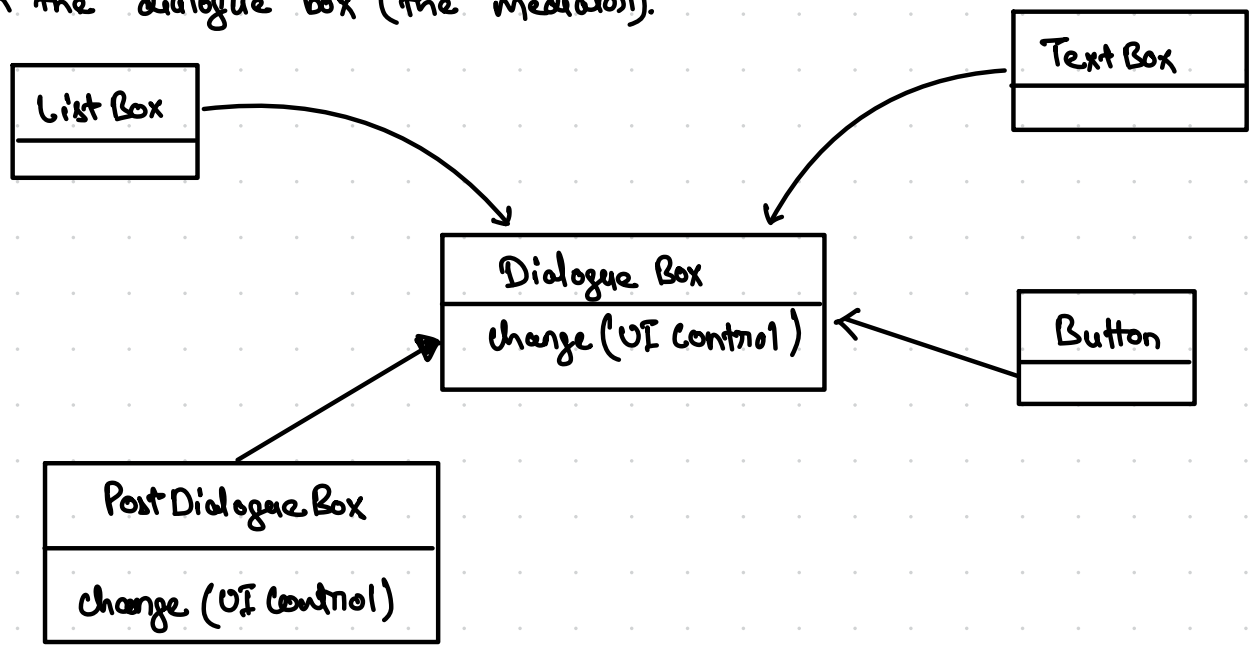
Components (classes) that we need.

- ListBox that contains the posts
- TextBox for editing title
- Button that can be disabled on enabled.

The above classes will come from a UI framework, so we do not have access to the source code.

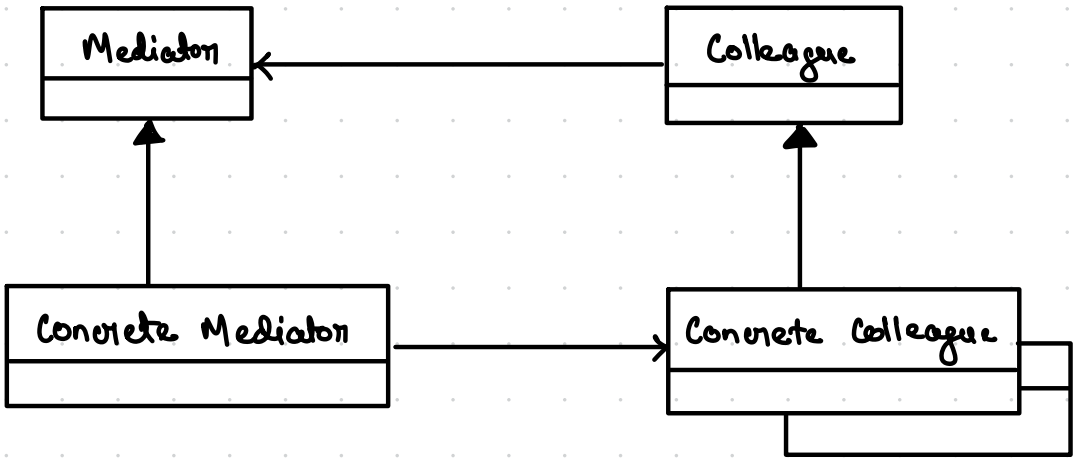
When an article is selected from the list box, the text box should be populated, and the button enabled. When we clear the text box, the button should become disabled.

Mediator Pattern:
The UI component don't know about each other, and all interaction logic is in the dialogue box (the mediator).



Whenever a UI component changes, it notifies its owner, the dialogue box, by calling the change(UIControl) method and passing itself as argument, which then handles updating other components.

GoF UML:



Abstract names for our previous post-title-editing app:

- Mediator = Dialogue Box
- ConcreteMediator = PostDialogueBox
- Colleague = UIControl
- Concrete Colleague(s) = our concrete UI classes (Button, TextBox, ListBox).

The concrete colleagues are all unrelated/uncoupled from each other. They talk to each other indirectly via a mediator, allowing them to be reused in different contexts - e.g. we are not coupling a list box to a text box or button.

The only coupling we have is b/w ConcreteMediator and Concrete Colleague. This is fine, because in our example the PostsDialogueBox needs to know about all of its UI components so they can interact with each other.

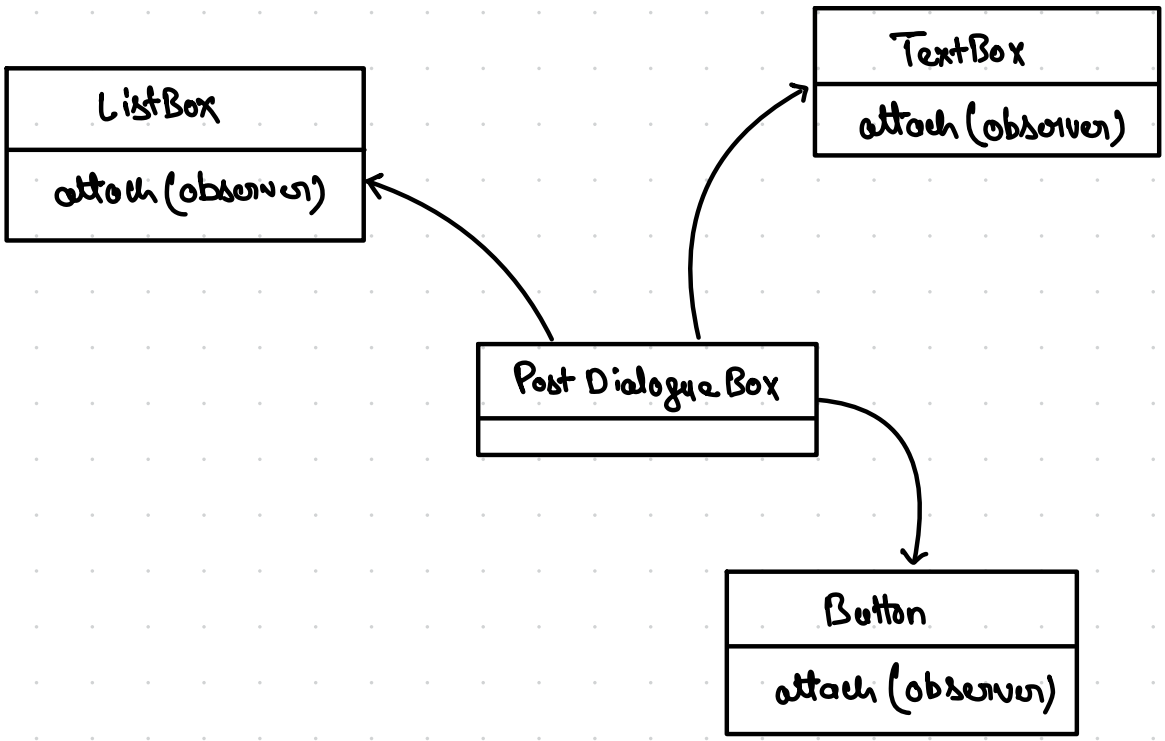
Mediator Pattern with Observer Pattern

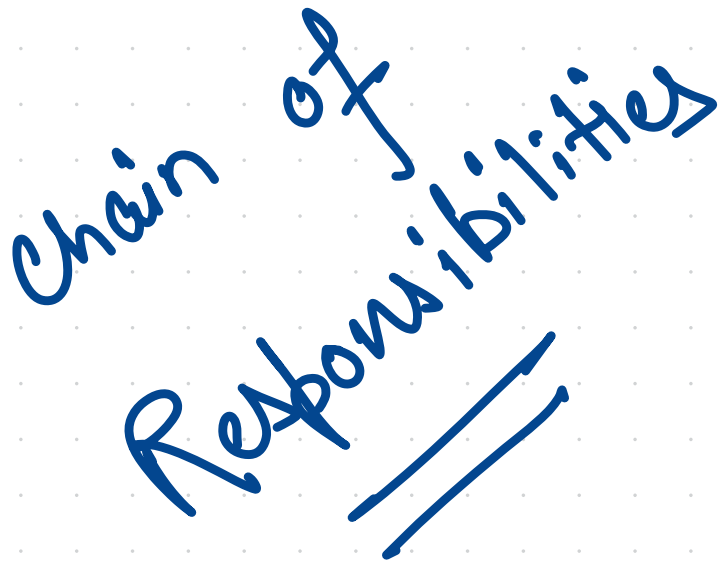
One problem with our previous solution is that the change() method on PostDialogueBox can get bulky as we add more UI components - lots of if/else to see what component has changed.

To solve this, we can implement the Mediator pattern using the Observer pattern.

The subject notifies the observer when any change happens.

The UI controls are the subjects, and the PostDialogueBox is the observer. When a UI control changes, PostDialogBox gets notified.




Chain of
Responsibilities

Chain of Responsibility

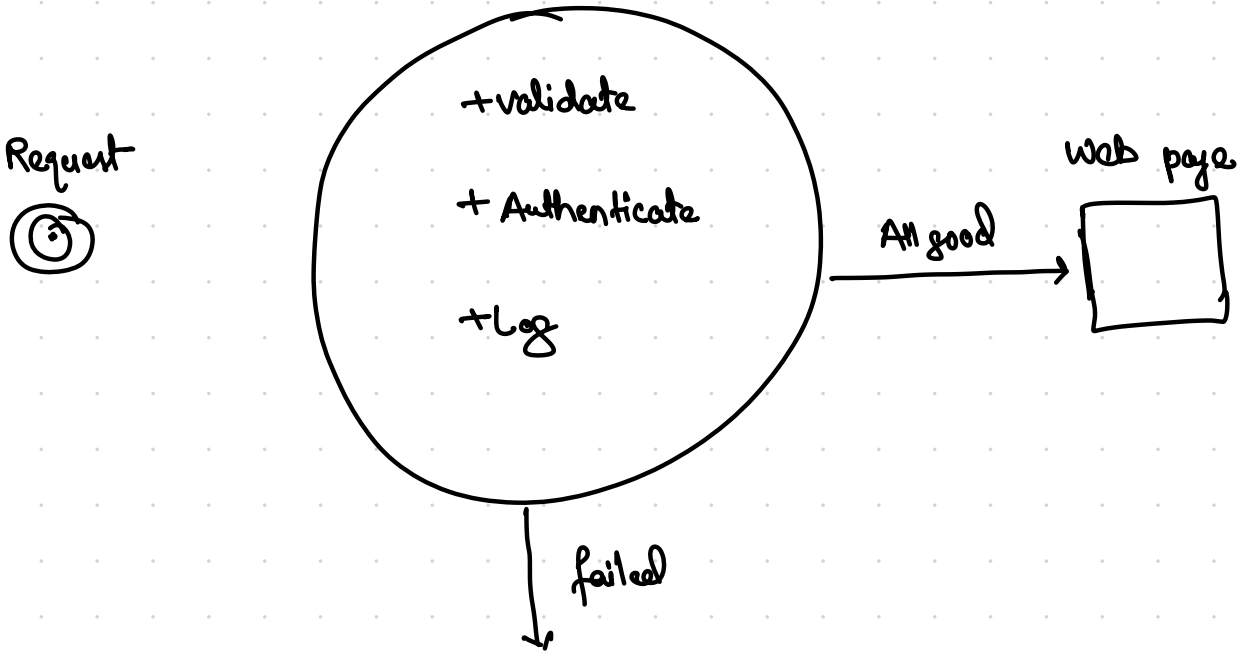
- The chain of responsibility pattern allows building a chain of objects to handle a request.

- A request is passed through a chain of handlers, each capable of either handling the request or passing it to the next handler in the chain.

Example: We have a web page that contains some information that only admins of this website can access, such as a page that allows an admin to manage the website's users - eg. create new user, get information, update user information, etc.

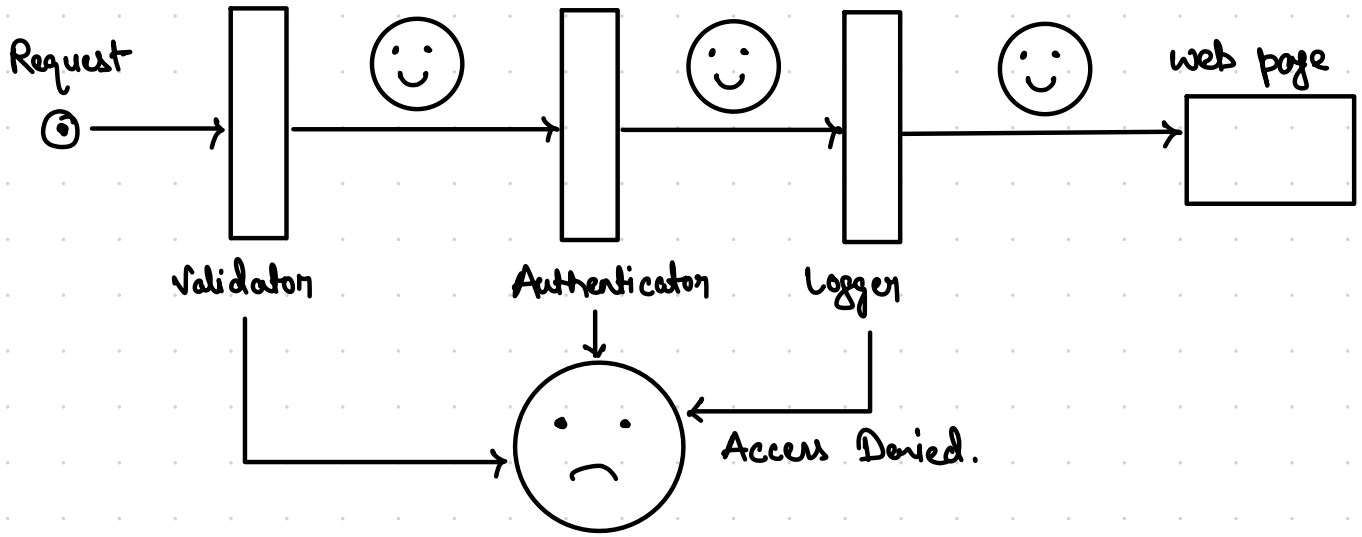
Let the user makes a request to the website server, but before returning the web page, the user's data must be validated (eg. trim any whitespace), authenticate the user, and then log some information onto the server about the request.

If any of those steps fail, then "access denied" is returned to the user.



Chain of Responsibility solution.

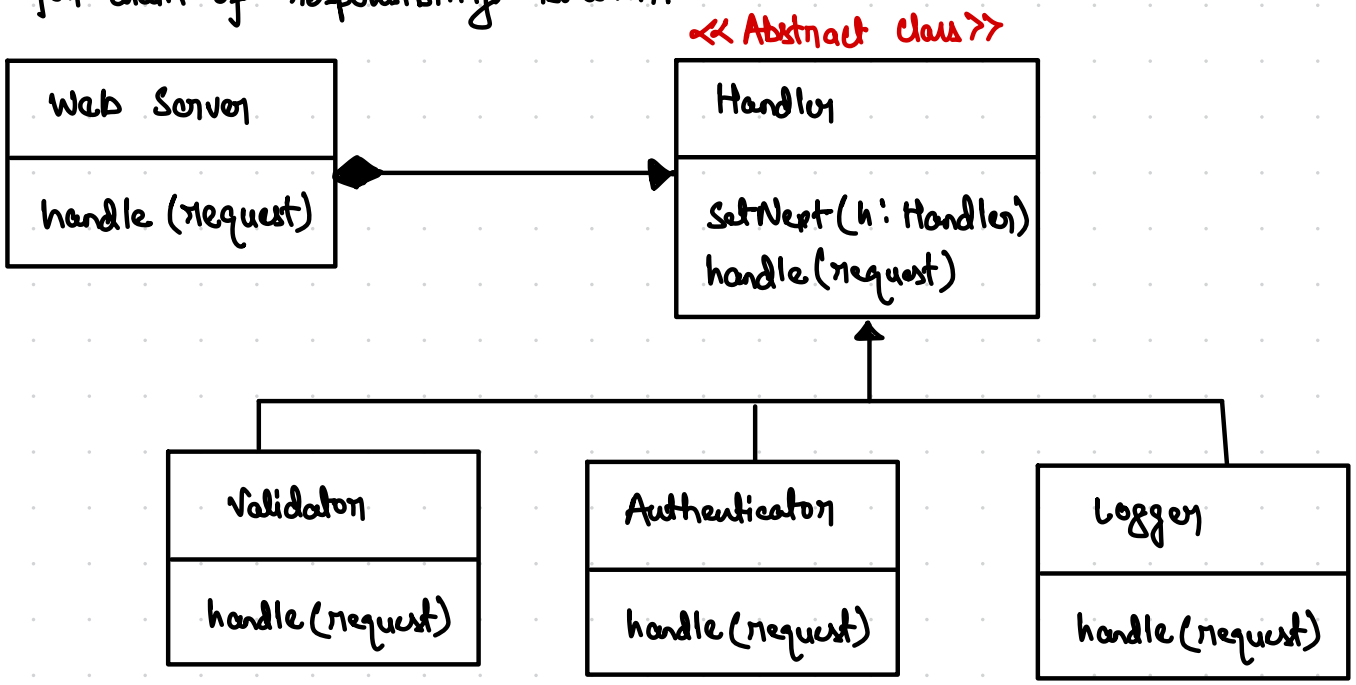
- Instead of having all our request processing logic inside of the one method, we can create a processing pipeline – a chain of objects.



- Each object only knows about the next object in the chain.

- First, a request is passed to the first object in the chain (Validaton). If this request is succenful, it will stop processing right there, so the other object aren't used.

UML for chain of responsibility solution:-



- We have a abstract class called Handler that has a reference to itself -- it has a field called next of type Handler.

- With this, each handler can know about the next handler in the chain (a linked list). handle() is an abstract method, because at the time of implementing the class we don't know how to handle a request -- we determine/implement this in our concrete handlers (validation, Authentication, and Logger).

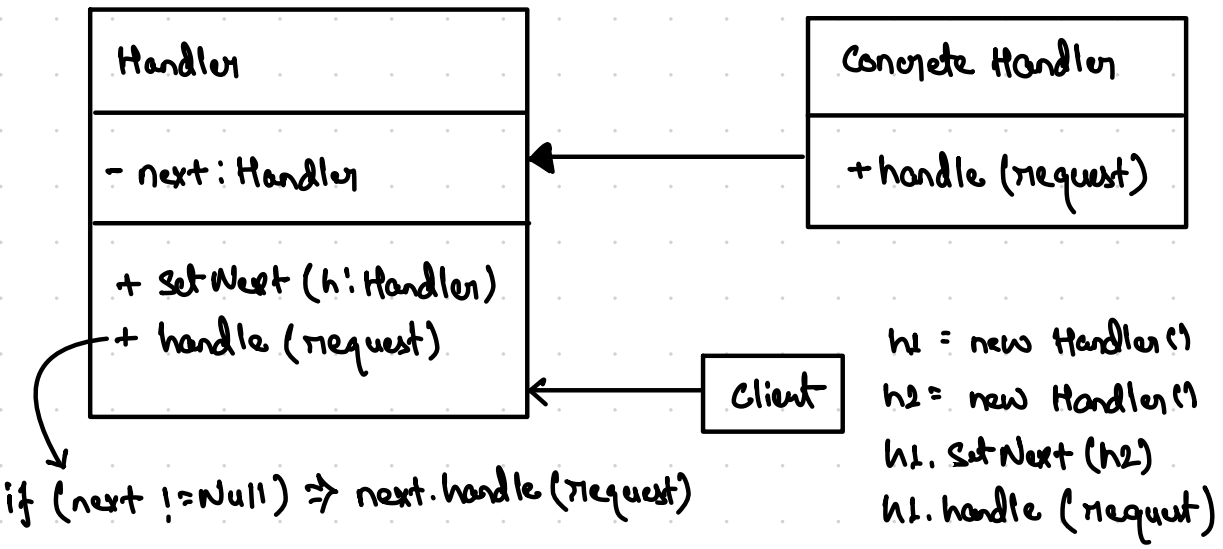
- Web Server has the reference to the first handler in the chain.

- Note: Web server is not talking directly to the concrete handler, it is talking to the handler interface. So, it's completely decoupled from the concrete implementation.

- This satisfies the open/close principle -- if we want to remove logging, we don't have to go to the handle() method on Web Server and change its implementation.

- And, if we want to add a new process, we can create a new class that extends Handler, then add it to our chain - we extend the code, but don't modify any existing implementations.

GoF implementation of Chain of Responsibility



Visitor Pattern

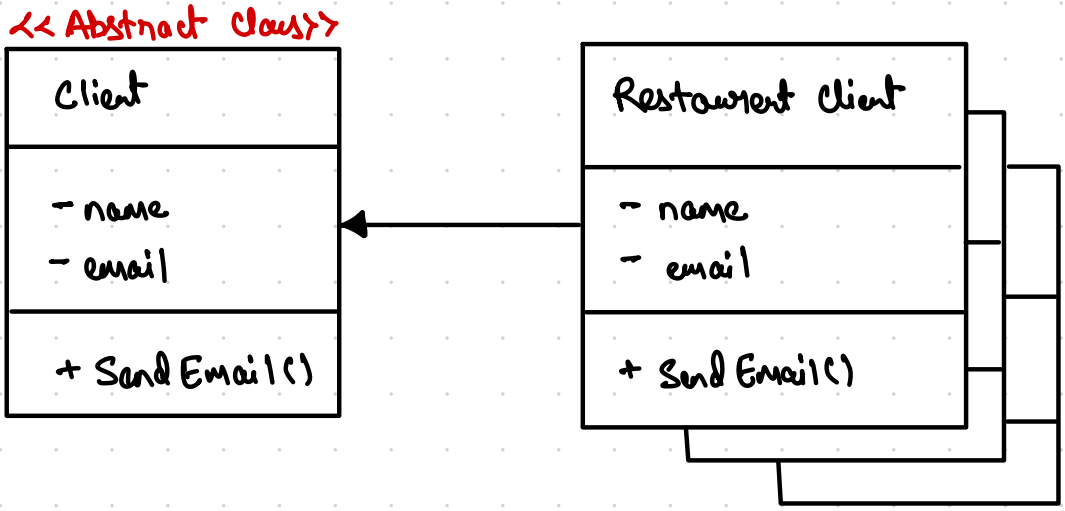
Visitor Pattern

- The visitor pattern separates the algorithm, or behaviors, from the objects on which they operate.

Scenario : We have different types of client

i. Restaurants
ii. Law firms
iii. Retailers.

and have to send customized email to each client.



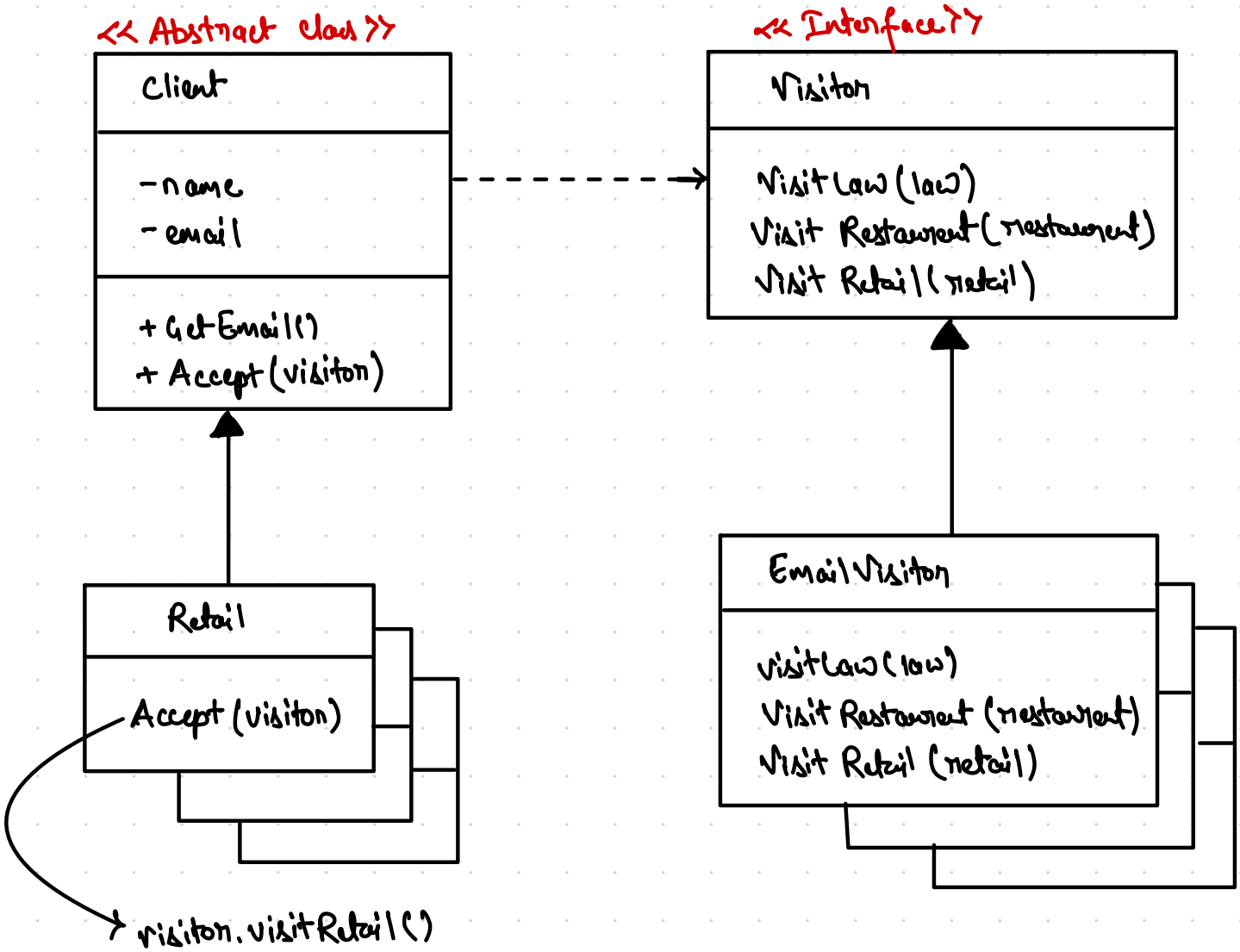
Problem : If we have to implement new features like export client as PDF or XML.

We have to modify abstract client class and its concrete classes.

It violates open/close principle and SRP.

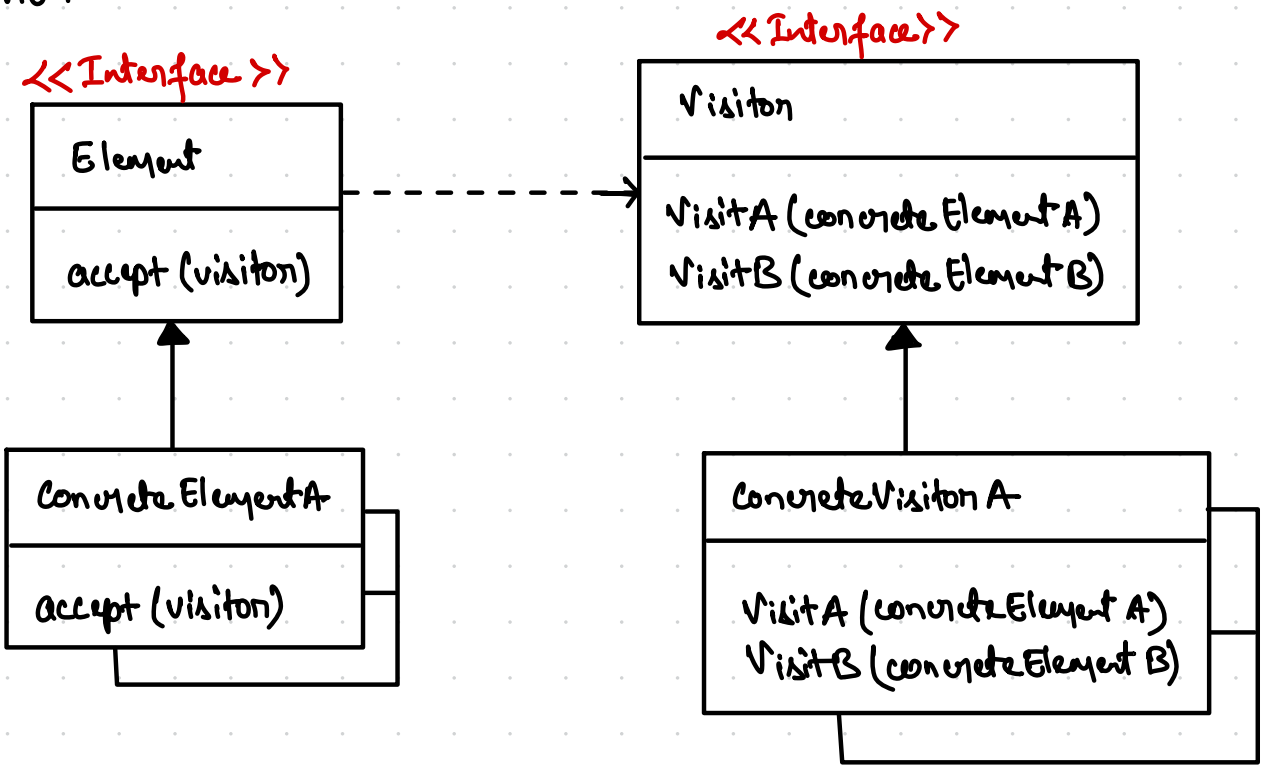
- To solve these issues, we extract these behaviours outside of the client classes on which they operates.

UML of Visitor pattern solution:-



- The behaviours have been abstracted into the concrete visitor classes, which can be passed to the objects that they operate on.

GoF UML :-


<<Interface>>
Element
accept (visitor)
<<Interface>>
Visitor
VisitA (concreteElementA)
VisitB (concreteElementB)
ConcreteElementA
accept (visitor)
ConcreteVisitorA
VisitA (concreteElementA)
VisitB (concreteElementB)

Interpreter Pattern

Interpreter Pattern

- Interpreter pattern defines a way to represent and evaluate sentences in a language by using an abstract class for expressions, which concrete subclasses implement to interpret specific parts of the language grammer.

Example Use case:

- Parsing and executing SQL queries, where the Interpreter pattern help parse the query syntax and execute it against a database.

- Calculators or scientific software that interpret and evaluate complex mathematical formulas entered by users.

- Web frameworks that render HTML templates with embedded expressions or directives – i.e. template – (e.g. {{ variable }} in Django or <%= expression %> in JSP).

Example: We need to build a calculator app, that takes some user input string and calculates the result.

User Input: "1 + 2 + 3"

- An Interpreter will convert/parse this input, or "expression", into an Abstract Syntax Tree (AST), or "Expression tree".

- We are converting, or parsing, an input string into a tree of objects according to the grammar rules that we specify.

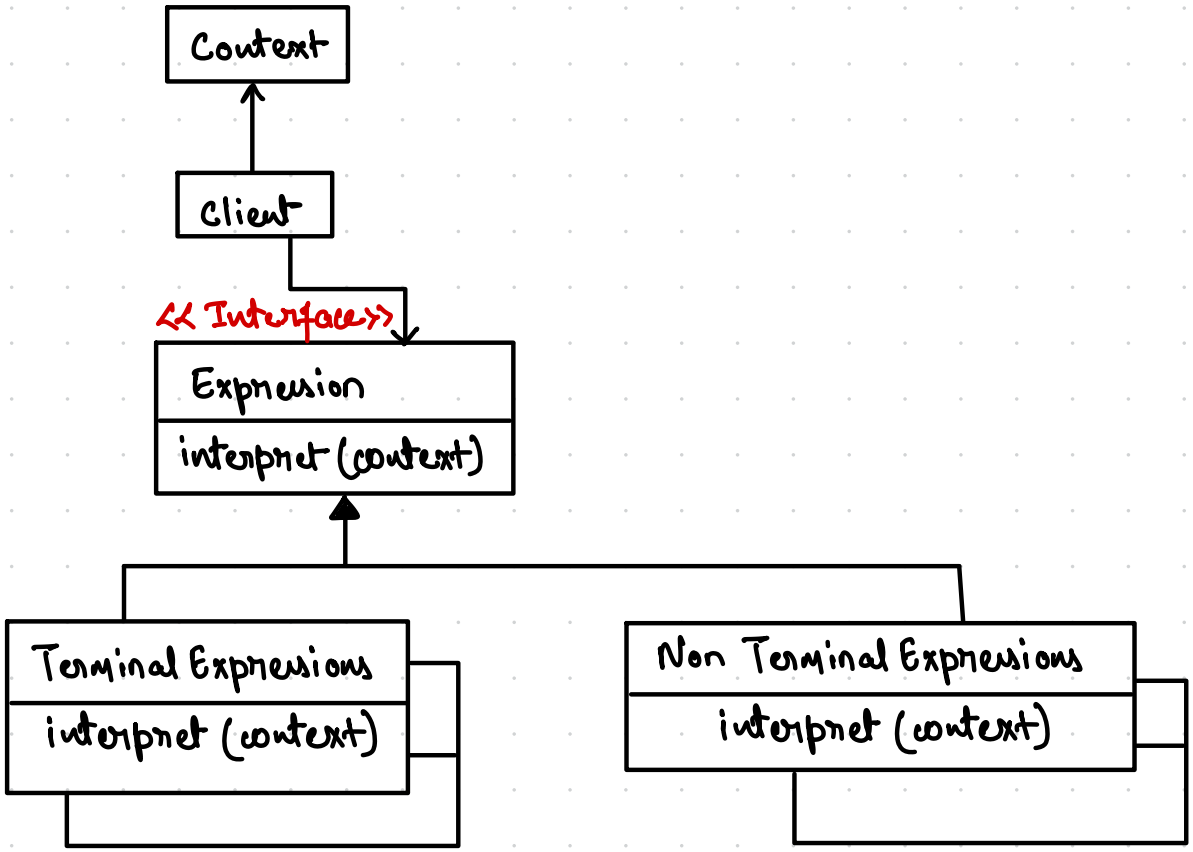
- The result of this expression can be found by calling its interpret method.

- We can use the interpreter pattern to get from the user input to the AST.

Components of the Interpreter Pattern

- Abstract Expression : Establishes the interface for all expressions within the language.
- Terminal Expression : Represents the fundamental components of the lang. such as numbers on variables.
- Non-Terminal Expression : Represents more complex expressions that are composed of other expressions using operators on functions.
- Interpreter : Implements the logic for interpretation and determines how to evaluate different types of expressions.
- The context class contains any global information needed for interpretation.

GoF UML:



When to use, and when to not use

- If you need to interpret and execute expressions or commands in a domain-specific language (DSL), the interpreter pattern offers a flexible and extensible methods for implementing the language's grammar and semantics.

- If your task only involves calculating simple operations that can be handled by a general-purpose programming lang or a library, then using the Interpreter pattern adds a lot of unnecessary complexity.

- If the grammar of the lang is complex, then the interpreter pattern could lead to having a large number of classes and increased code complexity. In this case, a dedicated parser generator or compiler could be a better option.

Structural Design Pattern

Structural Design Patterns.

- Structural design patterns focus on the composition of classes and objects to form larger structures and systems.

- These patterns primarily deal with how classes and objects can be combined to form larger, more complex structures while keeping these structures flexible and efficient.

- The key objective of structural design pattern is to provide solutions to design problems related to object composition and structure, allowing for better organization and management of code.

Structural design patterns help to achieve several important goals in software development.

- Promote code reusability and modularity by defining clear and standardized way to compose and organize classes and objects. This make the codebase more maintainable and scalable over time, as changes or additions to the system can be made more easily.

- Enhance flexibility and extensibility by allowing the system's structure to evolve without requiring major changes to the existing code. This is achived by decoupling the components of the system and promoting loose coupling b/w different parts.

- Improve performance and resource utilization by optimizing the way objects interact and collaborate within the system, thereby enhancing overall system efficiency.

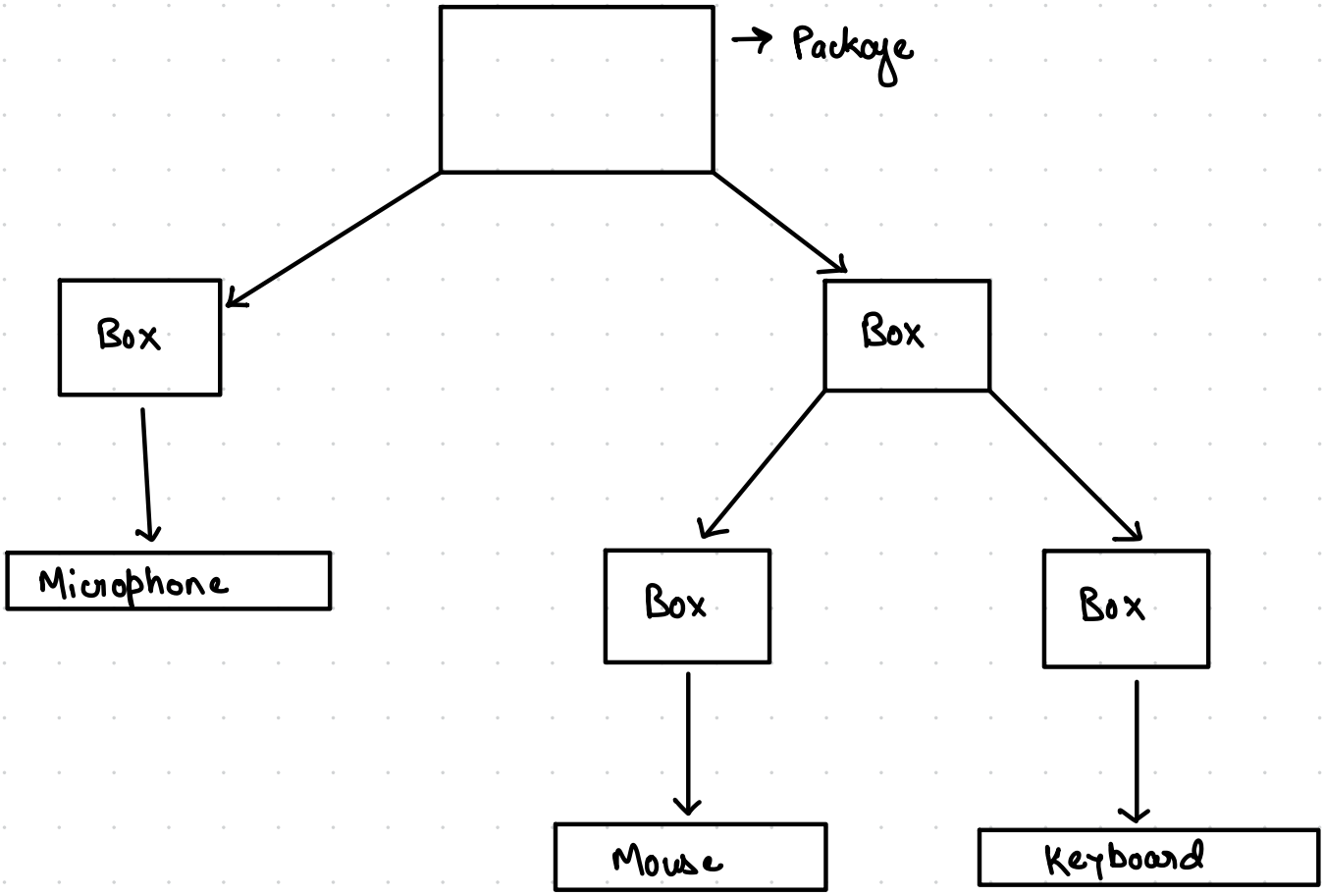
These patterns contributes to building robust, adaptable, and well-organized software systems that are easier to understand, maintain and extend over time.

Composite Pattern

Composite Pattern

It enables the creation of tree like structure to represent collection of objects, where both individual objects and groups of objects are treated in a unified manner.

Example: An amazon delivery of a large package that contains multiple items.



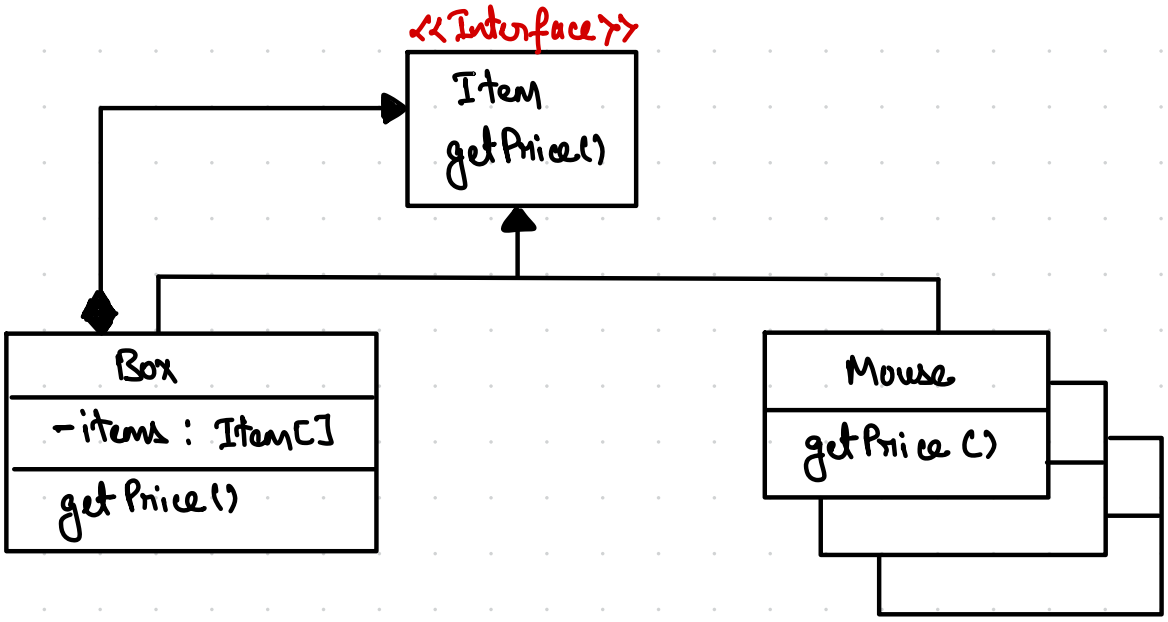
- Boxes can contain groups of other boxes and items. The above diagram shows how a package of items can be represented as a tree structure.

- Simple solution: Create an array of boxes and items, then loop through them recursively to find the total price of the package.

- Problems: Contains lots of conditions. This is hard to read.
  Open/close principle is violated: whenever we add a new item, we had to modify Box.

Whenever we have lots of conditionals that are checking, and casting it to another type, it's a good sign that we need to use polymorphism.

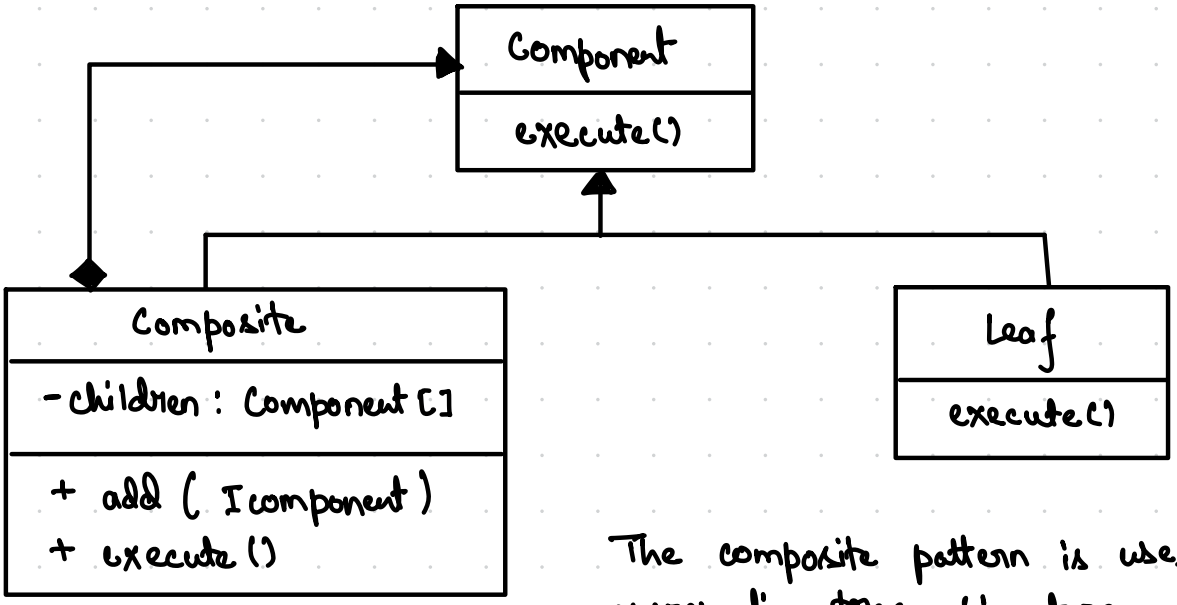
We can use polymorphism by creating an interface, called Item and extracting common methods, or logic, b/w the objects into interface, like this:



- A box is essentially composed of a group of items (represented by the diamond arrow), but is also an item itself (represented by the arrow). A box is simply just a group of items.

- Now, a box and its contains can be treated the same way - as items - thanks to polymorphism.

- GoF UML for the composite pattern :



The composite pattern is useful for representing tree structure.

- files and folders.
- graphical editor, that allows to group shapes together, and group groups of shapes together.

Adapter Pattern

Adapter Pattern.

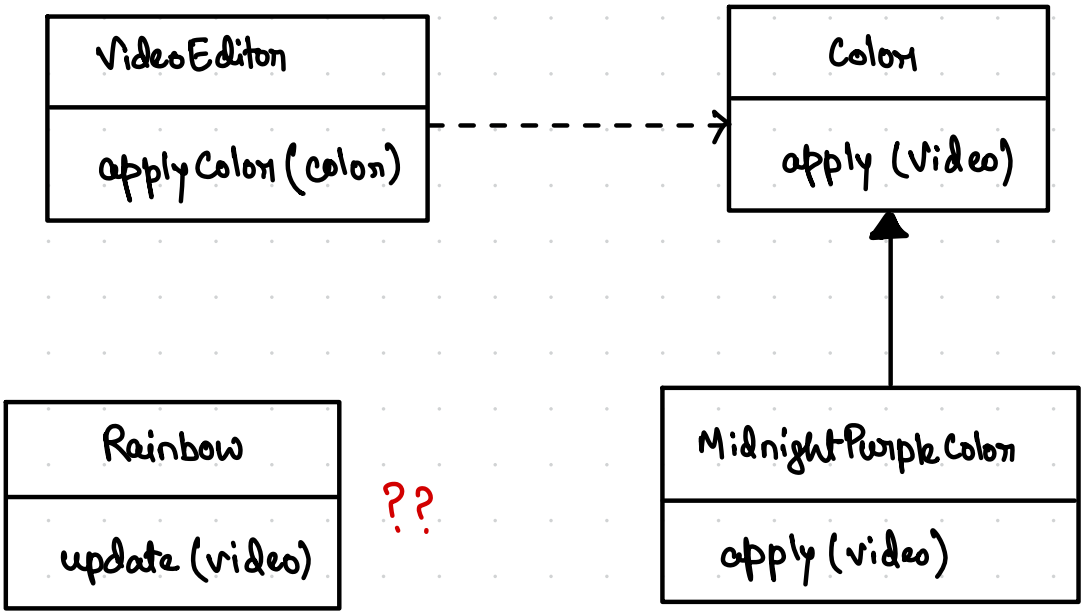
It is a structural design pattern that allows incompatible interfaces b/w classes to work together by providing a wrapper that translates one interface into another.

Example: We have a video editing application that allows users to upload a video and change the color of the video. The application provides preset color options for the user to select, such as black and white, or midnight purple.

- The classes that change the color of the application are built by us, and implements a Color interface.
- We then decided to install a 3rd-party library into our application that allows user to apply more types of colors to their videos.
- The problem is that all concrete color classes are expected to implement our Color interface and have an Apply() method. But, the concrete color classes from the installed library do not, meaning that we can't pass them to our videoEditor.ApplyColor() method, like

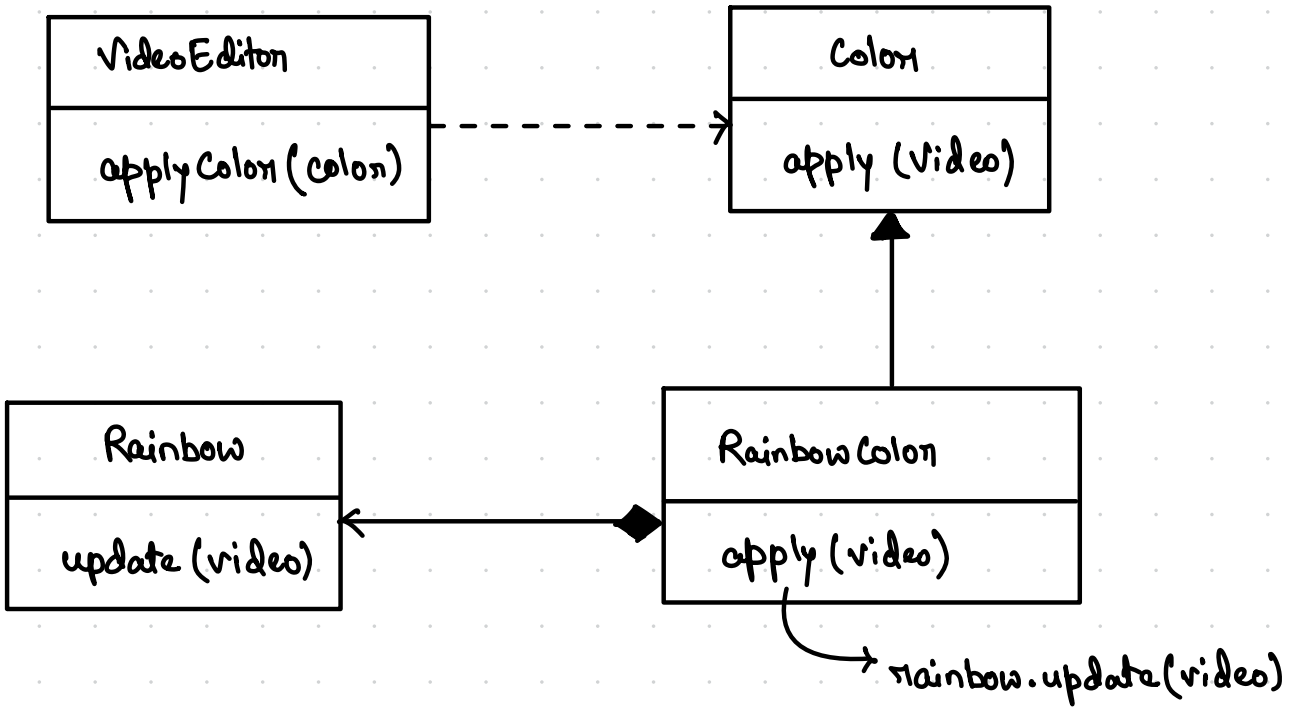
```
videoEditor.ApplyColor(new Rainbow());
```

- We cannot modify the 3rd-party library code to make the color classes implement our Color interface.
- We can solve this by converting the interface of the 3rd party color classes to a different form, using the Adapter pattern.
- We can't use Rainbow because it doesn't implement Color, and we can't make it implement Color because it's inside a 3rd party package



Solution

We create a RainbowColor class that implements Color and is composed of 3rd party Rainbow class. We can then call the Apply() method in RainbowColor, and, inside it, call whatever methods we need to call from Rainbow to apply the filter. We adapt the class to a different form:



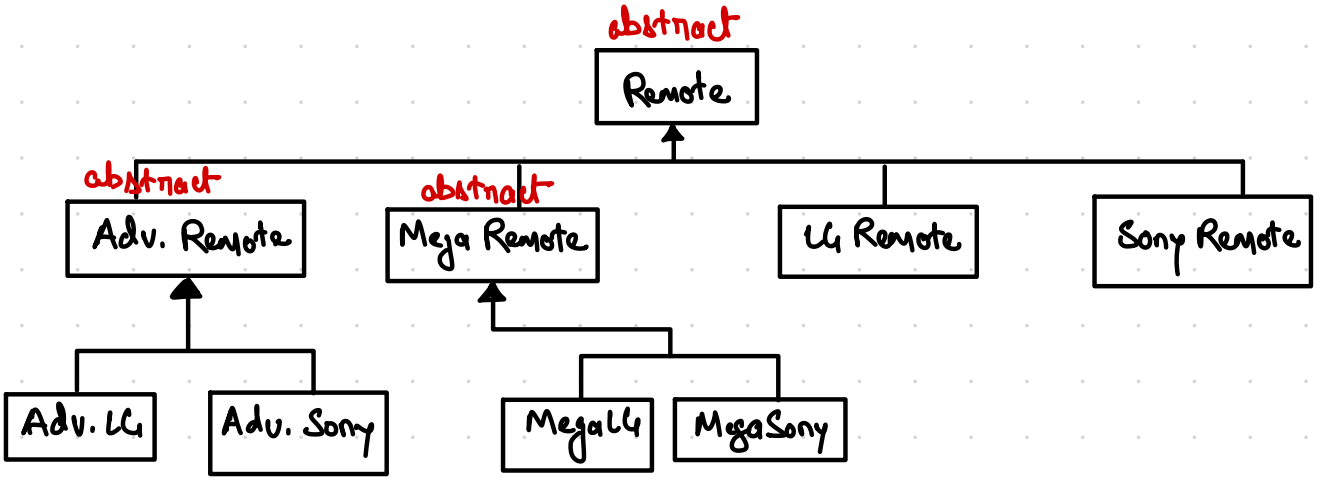
- RainbowColor is the adapter: it's converting the interface of the Rainbow ("adaptee") class into a different form (Color).

Bridge Pattern

Bridge Pattern.

This pattern separates a large class, or a set of related classes into two separate hierarchies so they can be developed independently from each other.

Example:
We have a remote for controlling a radio. There are multiple different brands of radio, and there are different types of remote.



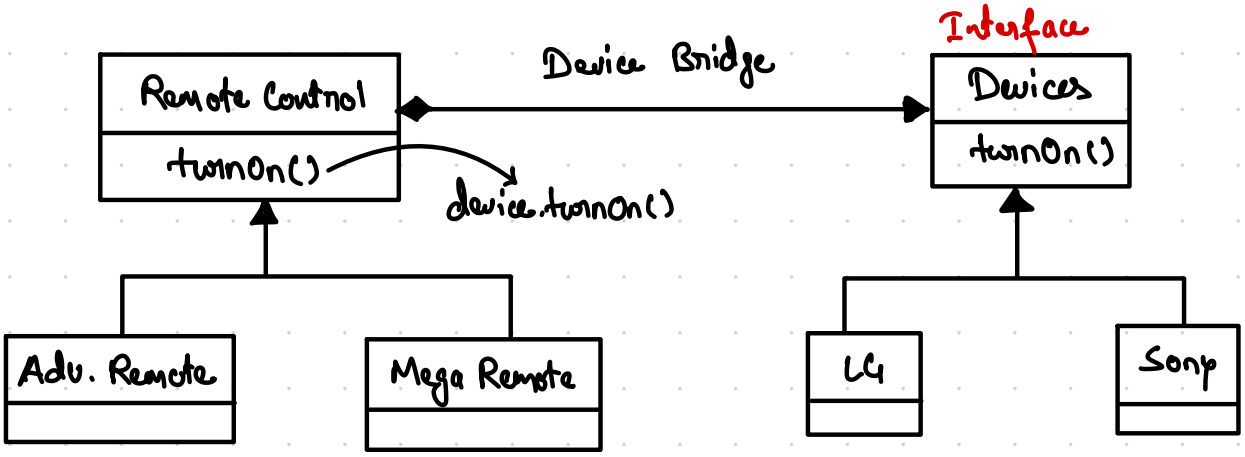
Problem:
Every time we add a new brand, e.g. Samsung, we'd have to create three new classes: SamsungRemote, Adv. SamsungRemote and MegaSamsungRemote.

And if we create a new type of remote, e.g. RadioAndTVRemote, then we have to create a new class for every brand, so RadioAndTVLG, RadioAndTVSony, RadioAndTVSamsung.

Which is not maintainable.

Reason:
We ended up with this structure is because our hierarchy is growing in 2-dimensions: the abstract dimension (remote type) and an implementation dimension (the brand/device).

To simplify this hierarchy. we can break it down into two separate hierarchies:



Whenever we have a hierarchy growing in two separate dimensions, we need to split them in half and connect them using a bridge.

Two hierarchies can grow independently from each other.

Proxy Pattern

Proxy Pattern

It is a structural design pattern that provides a proxy, or agent, object to control access to another object, allowing for additional functionality such as caching, logging, lazy loading on access control, without changing the client code.

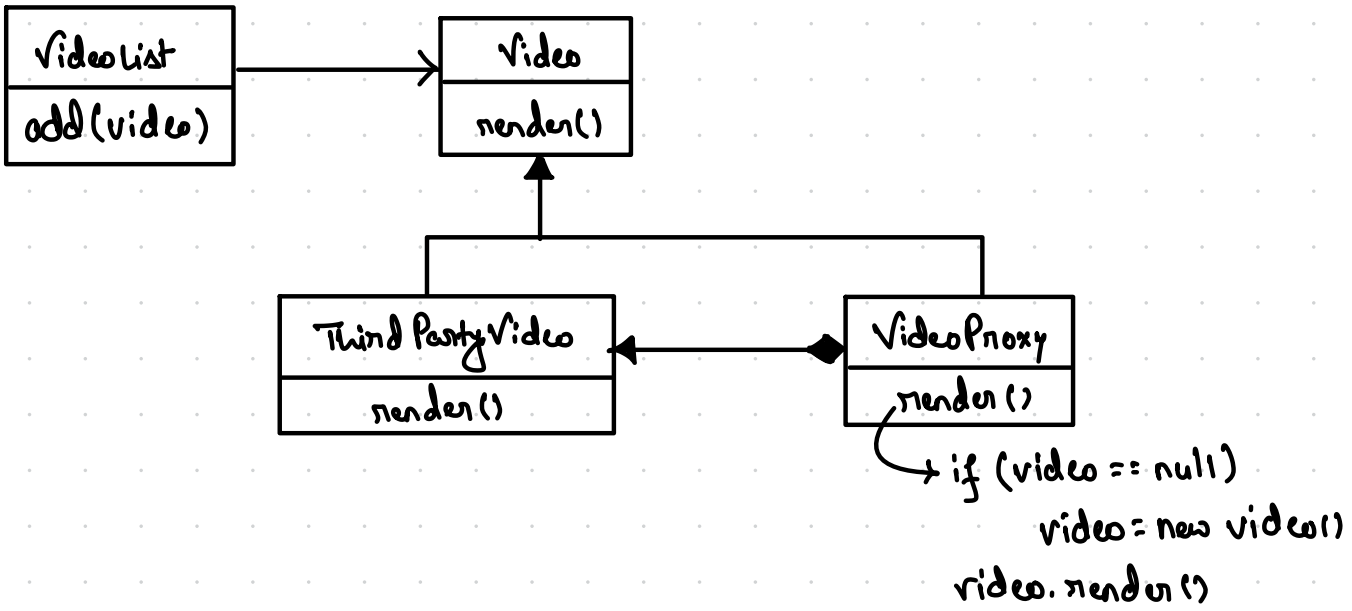
Example:
We have an application that fetches a list of Youtube video from Youtube API and displays them in a list.

In our application we are using a 3rd-party YouTube package to handle fetching Youtube videos from the API, and then rendering the video on the screen with the video controls.

The problem is that every time a request is made to our application, our server has to re-download the video from the Youtube API. This takes long time, especially if lots of requests are made to our application at once.

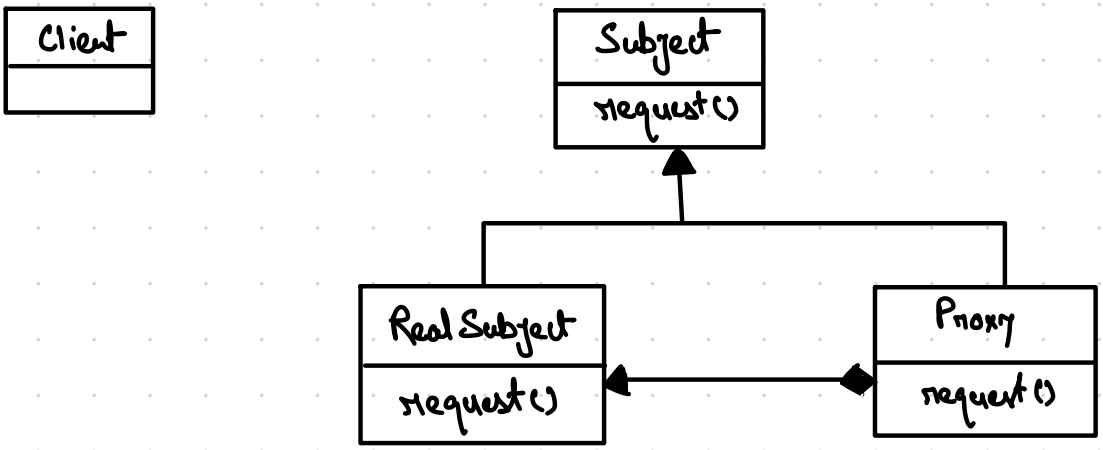
The YouTubeVideo class is from a third-party library, so we are unable to modify it's code. The solution is to use a proxy object.

Our VideoList object will now talk to VideoProxy objects.



The above UML shows that our VideoProxy object will implement the third-party libraries Video Interface, meaning that we can add these video proxy objects to our VideoList, and VideoList won't care weather its a ThirdPartyVideo or a VideoProxy, as long as it's a Video.

GoF UML:

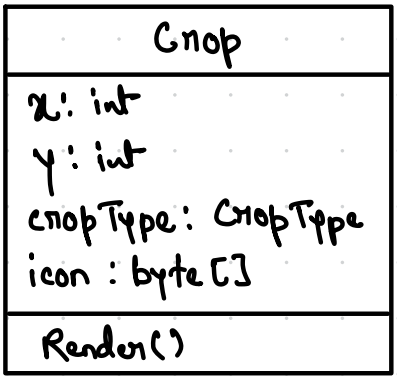


Flyweight Pattern

Flyweight Pattern

Structural design pattern that aims to minimize memory usage by sharing common state b/w multiple objects, allowing efficient handling of large numbers of lightweight objects with shared characteristics.

Example: We have a farming game that includes different types of crops, such as potatoes, carrots and wheat. Each crop is represented by a crop object, that includes its x and y coordinates, the type of crop, and an icon:



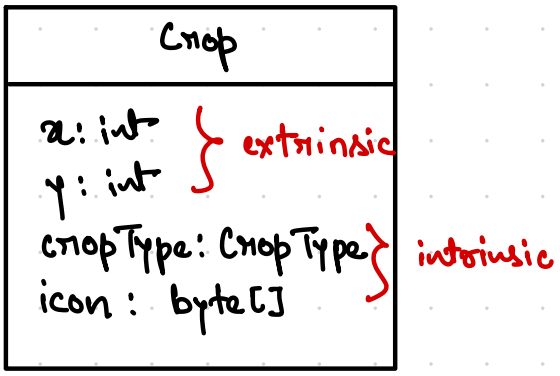
Problem: For every carrot created, we are storing a new object with all of the information about a carrot stored within that object.

This means that if we create 1000 carrots, then we will be storing 1000 carrot icons in RAM - that is going to take up lots of memory, and many mobile devices will struggle to handle that.

What if we could share icons b/w crop objects of the same type...

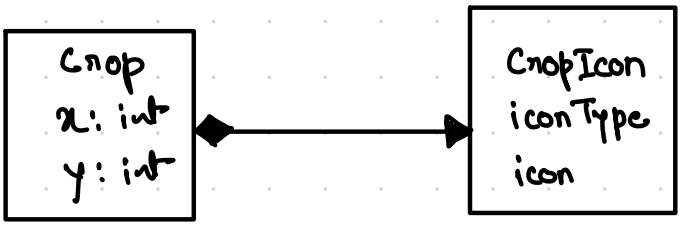
If a carrot is created with the Crop class, then its icon and CropType fields will remain constant for the lifetime of the object, but its x and y coordinate will vary, as the crop can be harvested and moved around.

State that remains the same throughout the objects life is called internal state. State that can change is called extrinsic state.



We can extract the intrinsic state out of Crop, and place it into a new object, called CropIcon, the we would only need to create three CropIcon objects (for potato, carrot and wheat) in our application, even if there are 1000s of crops in the game.

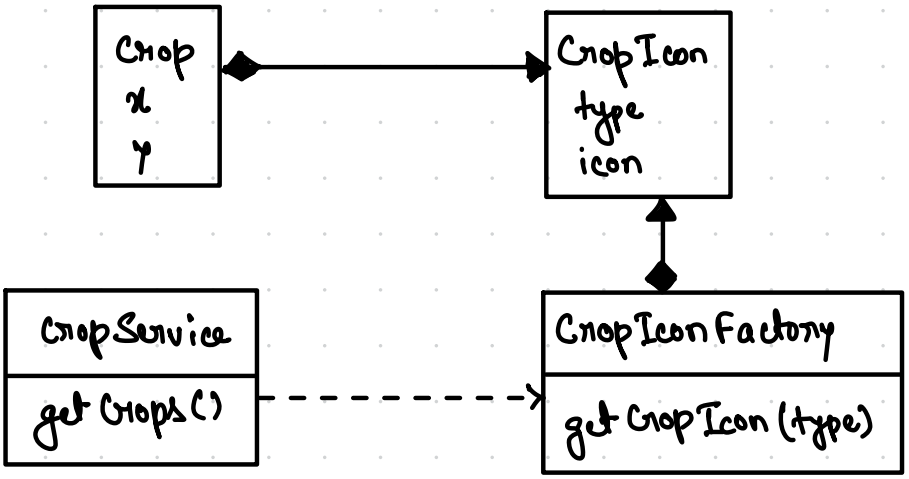
We can then, for example, have just one carrot icon object stored in memory, then all crops of type carrot can reference, or reuse, that carrot icon object throughout the game.



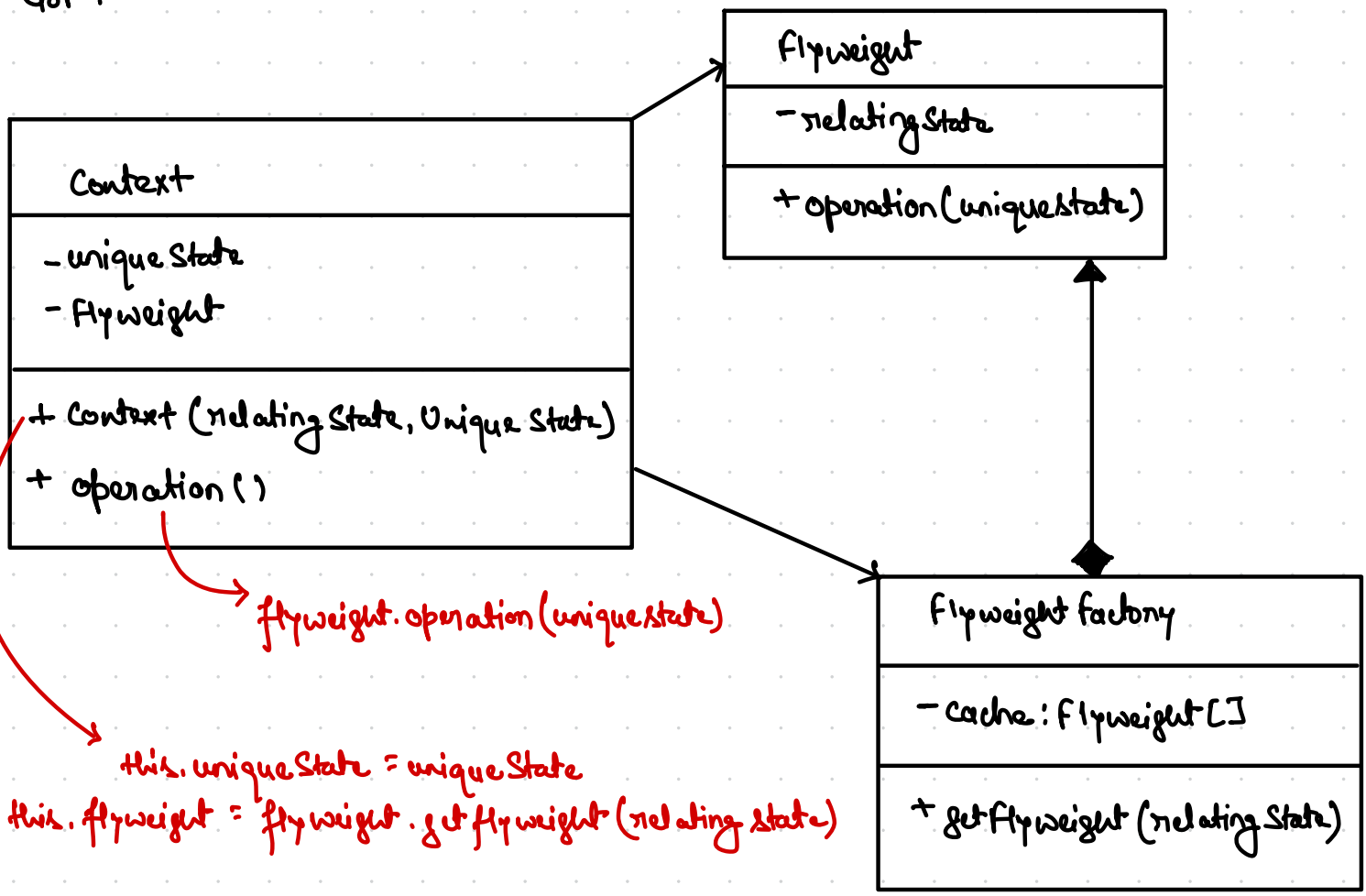
- An object that contains only intrinsic state is called a flyweight.

- We shouldn't create CropIcon objects directly. We can create a factory class that creates an icon, depending on the icon type, and caches that icon in memory, ensuring that it's only stored in one place.

Overall Structure:



Gof :

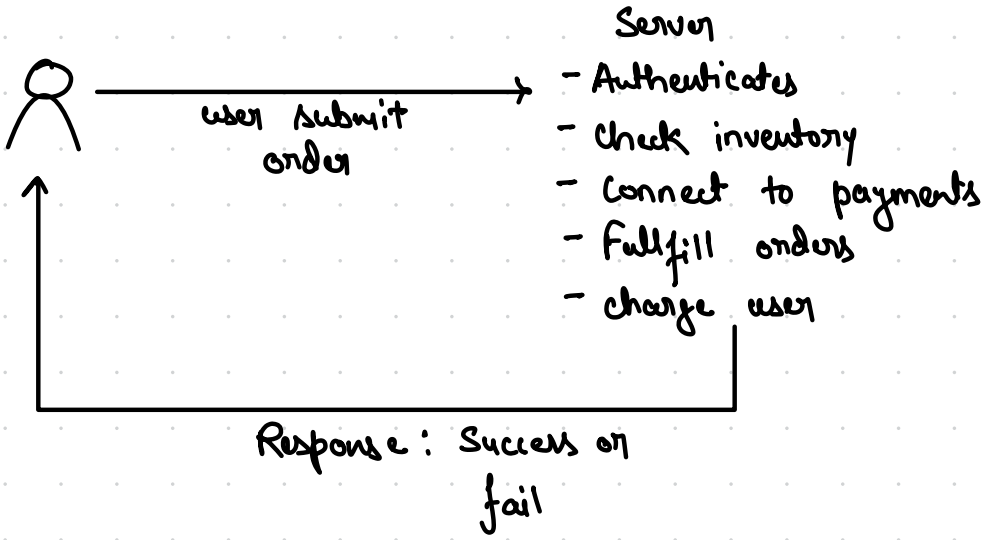

Flyweight
- relatingState
+ operation(uniqueState)
Context
- uniqueState
- Flyweight
+ Context (relatingState, UniqueState)
+ operation()
Flyweight factory
- cache : Flyweight[]
+ getFlyweight (relatingState)
Flyweight.operation(uniqueState)
this.uniqueState = uniqueState
this.flyweight = flyweight.getFlyweight(relatingState)

Facade Pattern

Facade Pattern

The facade pattern is a structural design pattern that provides a simplified interface to a complex system, encapsulating the complexities of multiple subsystems into a single unified interface for client.

Example: We have an eCommerce application that allows users to submit orders. Here are the steps involved :-

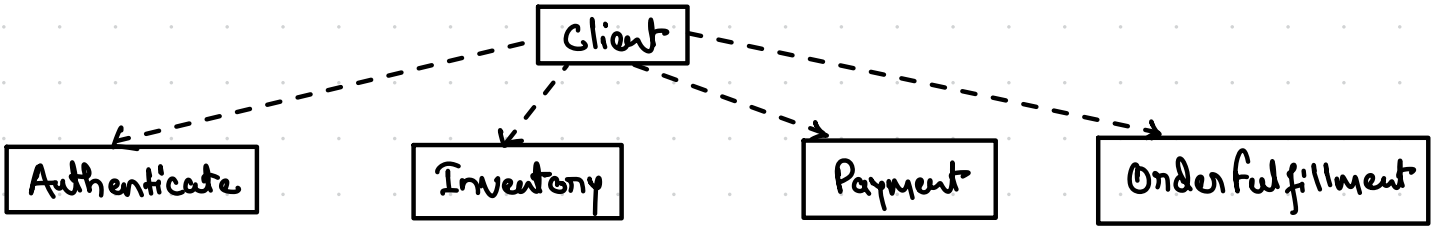


Every client that needed to make an order has to go through all these steps to make a single order. Every client becomes coupled to, or dependent on, four classes :-

Authenticate, Inventory, Payment and OrderFulfillment.

If we had ten classes that needed to make an order, then we'd have ten classes dependent on these four classes - that's a lot of coupling. If one of these four classes changes, that's ten classes that may need updating. Not good.

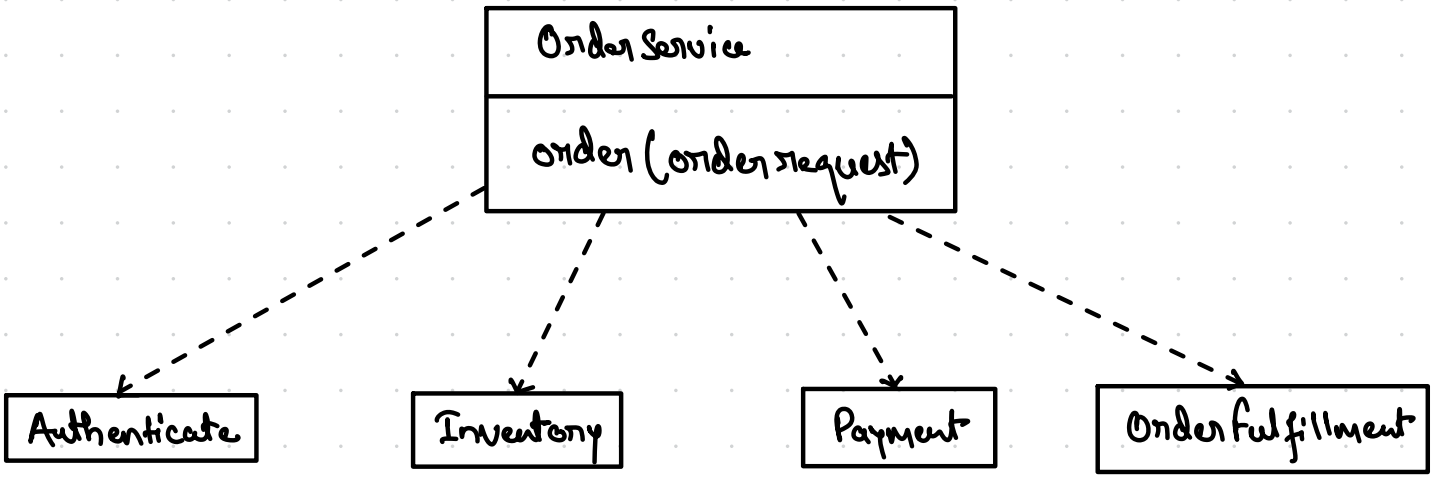
Each client that needs to make an order will be dependent on four classes :-



Facade pattern Solution :-

Clients making an order don't care about, or don't need to know about, the steps involved in making an order, they just want to make an order.

Introduce a new class, OrderService, with a single method, order(), that abstracts all the logic, so all other classes that needs to make an order only have to depend on this one class :-



Now, all classes that need to be able to make an order only need to depend on the on OrderService class.

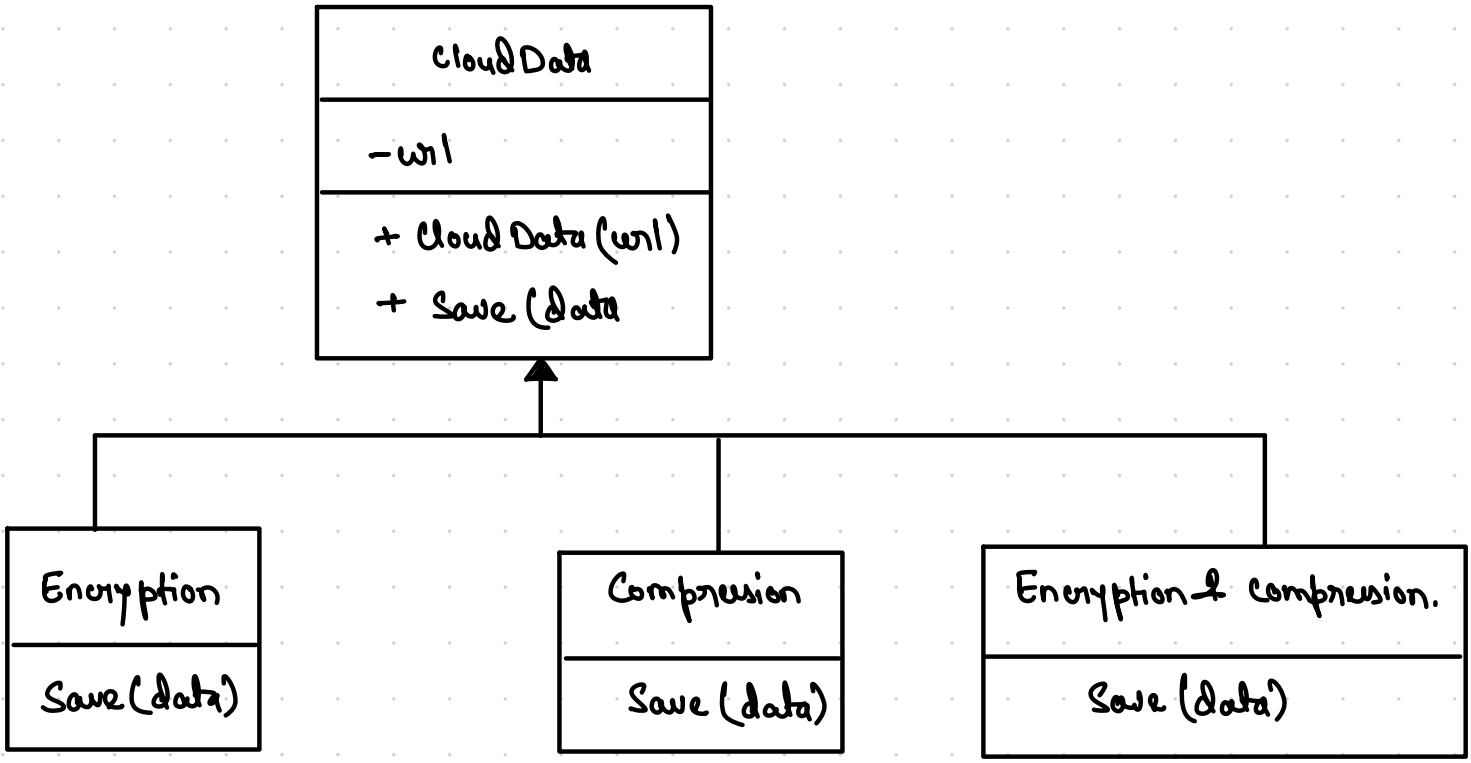
Decorator Pattern

Decorator Pattern

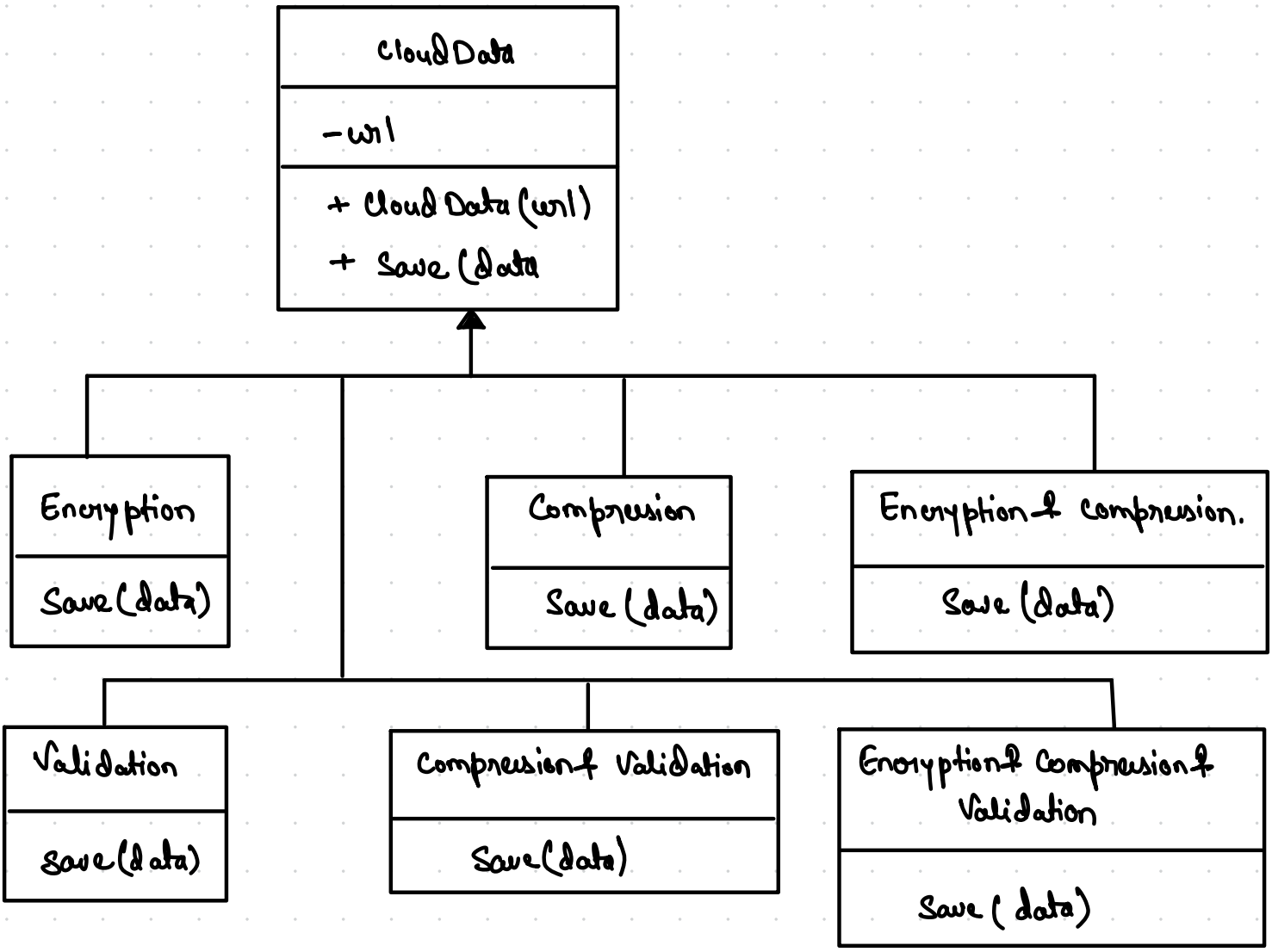
It is a structural design pattern that allows behavior to be added to individual objects dynamically, enhancing functionality without altering the object's structure, and it's used to extend or modify the behaviour of objects by wrapping them with additional functionality through composition.

Example: We have an application that allows users to store data in the cloud. The data can be sent to the cloud as it is, without any processing and it can also be compressed and/or encrypted before it is saved to the cloud.

UML Diagram:-

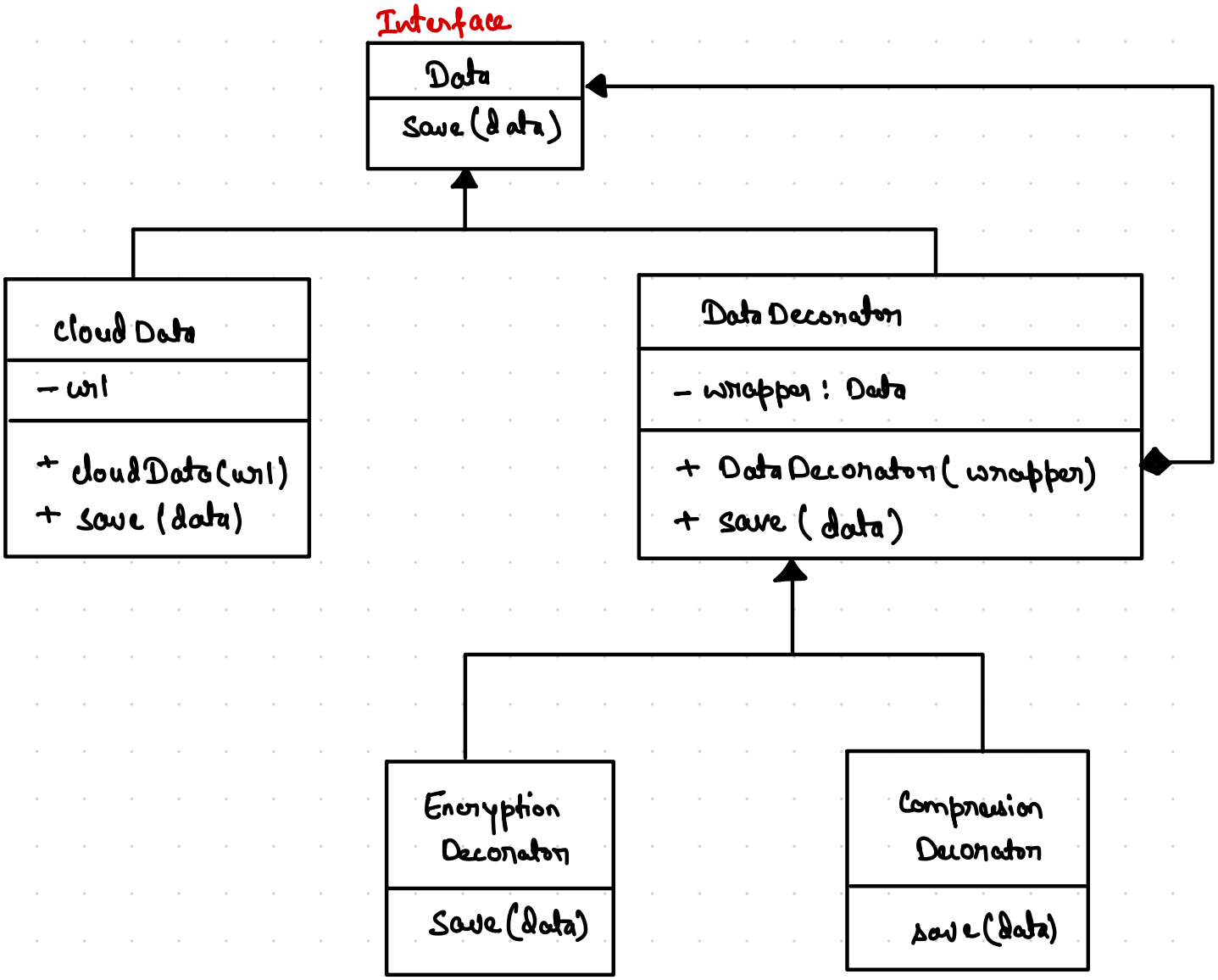


Say we need to create a new validation class to validate and clean the data before it is sent to the cloud. Our code starts to look bloated as we need to make lots of new classes just to add an extra feature:-



To add one new feature, we have to create four new classes. Our class library is growing exponentially - not good, not maintainable, not flexible!

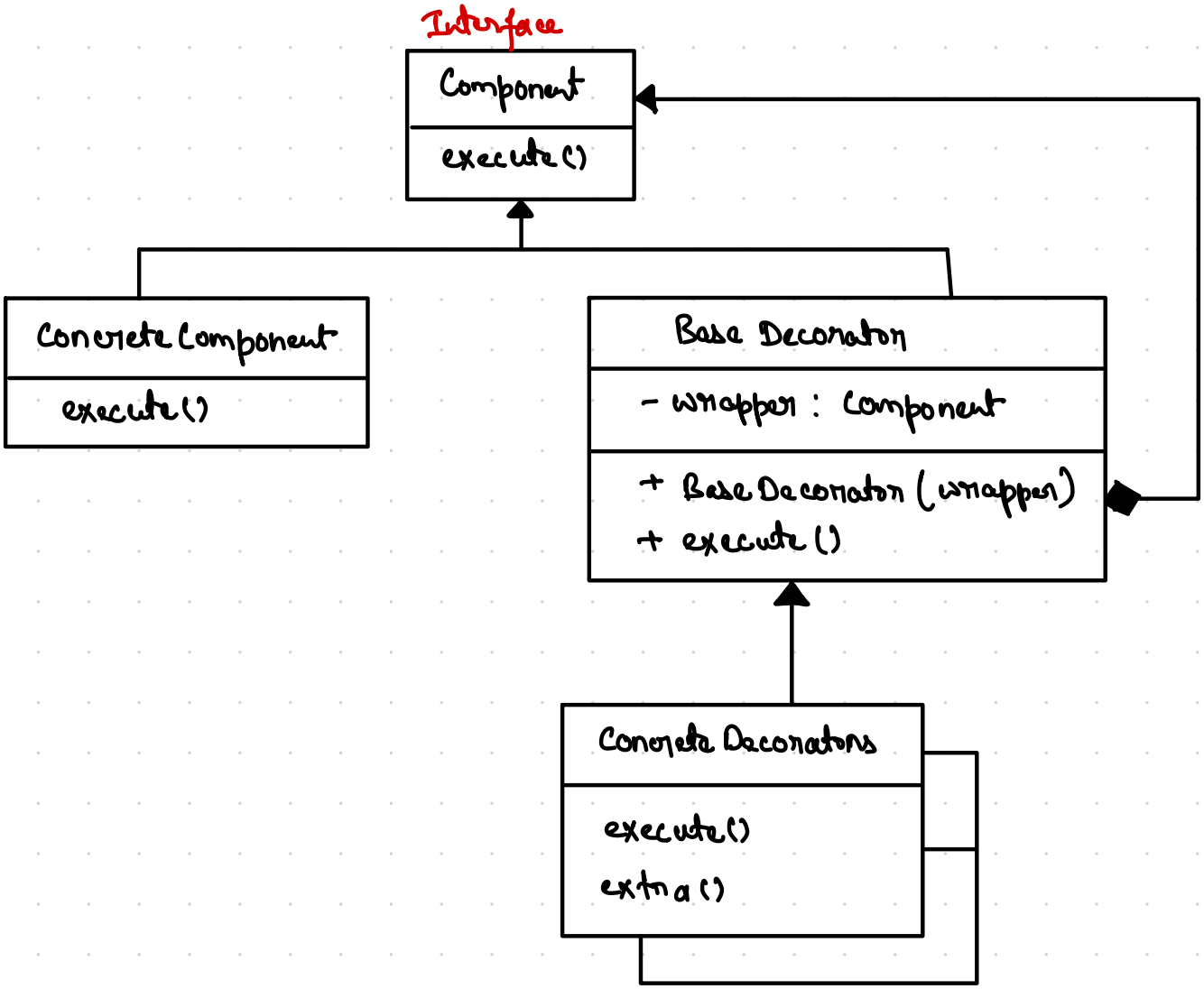
Solution! Move away from inheritance and use composition instead.



We make the encryption and compression objects decorators, because they are decorating the CloudData object with some additional behaviour. Because decorator classes have some common logic - e.g. referencing a CloudData object - we have created a DataDecorator class where this logic can be inherited to prevent code repetition.

Our decorator classes are composed of (or are decorating/wrapping) a CloudData object (or any object that implements the Data interface).

Gof UML :


Interface
Component
execute()
ConcreteComponent
execute()
Base Decorator
- wrapper : Component
+ BaseDecorator (wrapper)
+ execute()
Concrete Decorators
execute()
extra()

Creational Design Pattern

Creational Design Pattern

It is a category of design patterns that focus on object creation, dealing with the best way to create objects while hiding the creation logic and making the system independent of how its objects are created, composed, and represented.

Benefits :

- Encapsulation of Object creation.
- Enhanced Flexibility and Extensibility.
- Improved Code Reusability.
- Promotion of Separation of Concerns.
- Support of Dependency Injection.
- Centralized control over object creation.
- Enforcement of Design Principles.

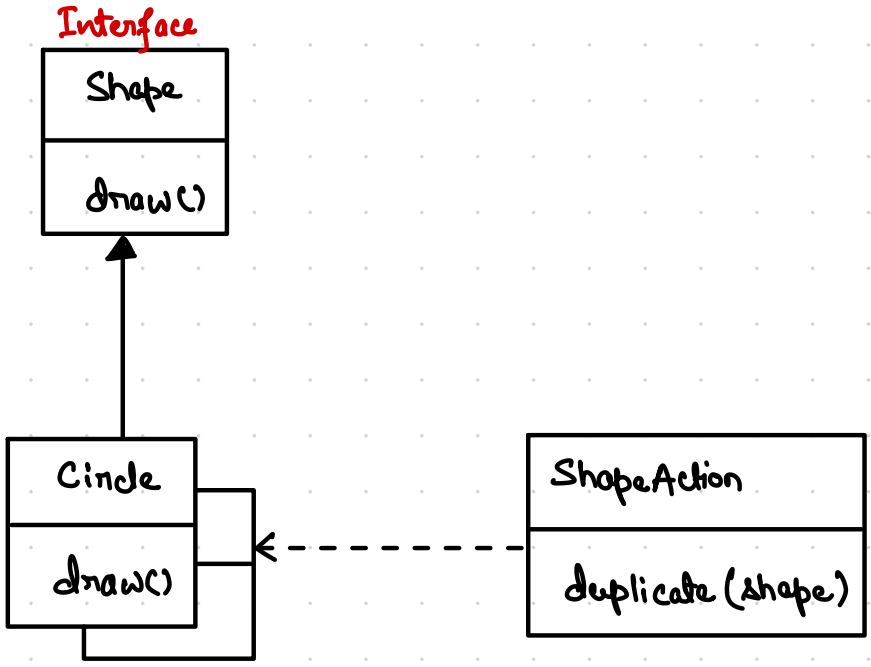
Prototype Pattern

Prototype Pattern

Creational design pattern that allows objects to be copied or cloned, providing a mechanism to create new instances by copying existing objects without explicitly invoking their constructors, and it is used to efficiently produce new instances with identical properties to existing objects.

Example: We have a GUI that allows the user to create new shapes on the screen, such as circles and rectangles. When the user right-clicks on a shape, an actions menu opens up. The user can then select "duplicate" to clone the shape.

Solution:

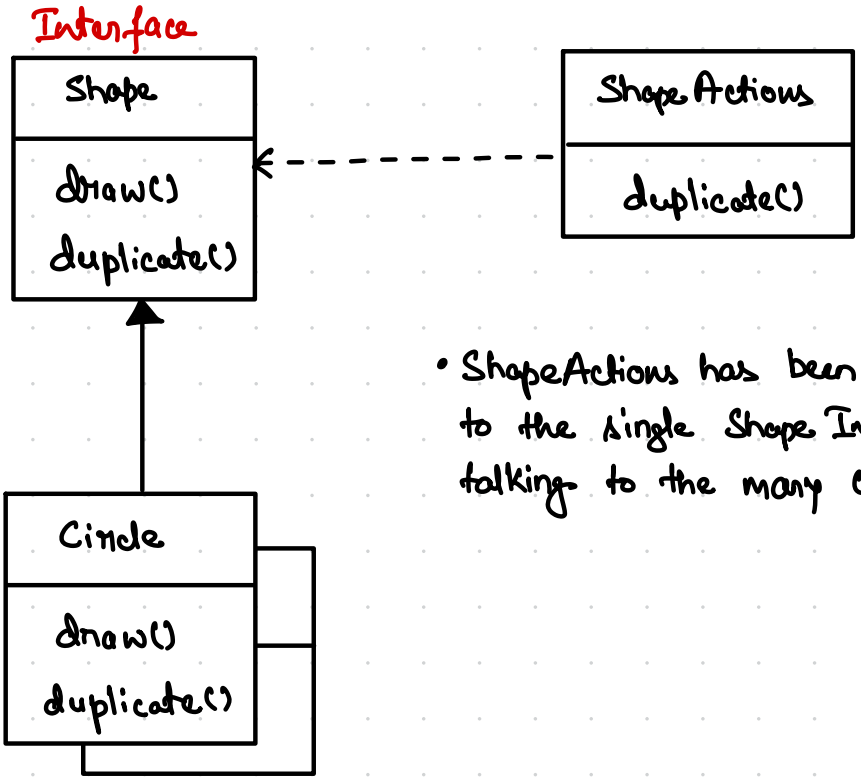


We have concrete shape classes that implement the Shape Interface. The ShapeActions class contains the logic for duplicating each shape, and is dependent on all of the shapes that can be duplicated.

ShapeActions has to know about all the kinds of shapes. Not a flexible solution. It currently depends on concrete implementations of shapes.

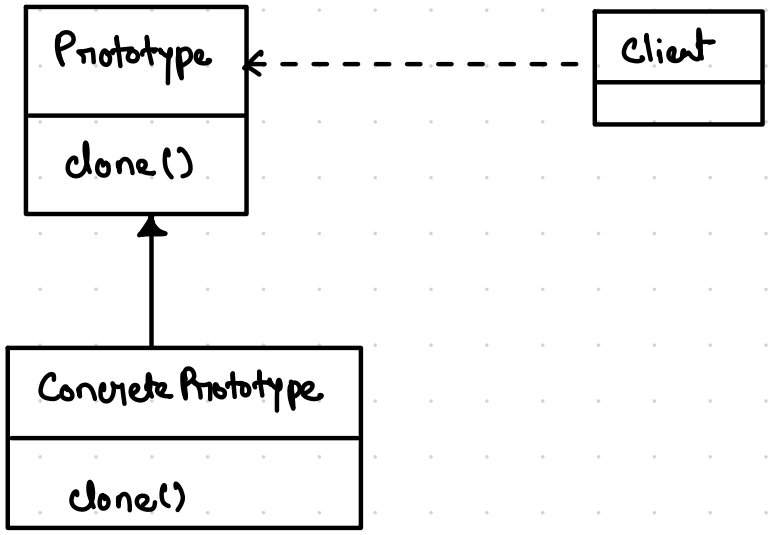
Good Solution:

The logic for duplicating a shape can be moved to each concrete shape class, rather than having it all in ShapeActions.Duplicate(). We can then decouple ShapeActions from all of the concrete shapes, and have it talk to the Shape interface. Flexible and maintainable.

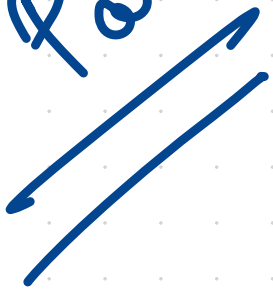


- ShapeActions has been lifted up to talk to the single Shape Interface, rather than talking to the many concrete classes.

GoF:



Singleton Pattern



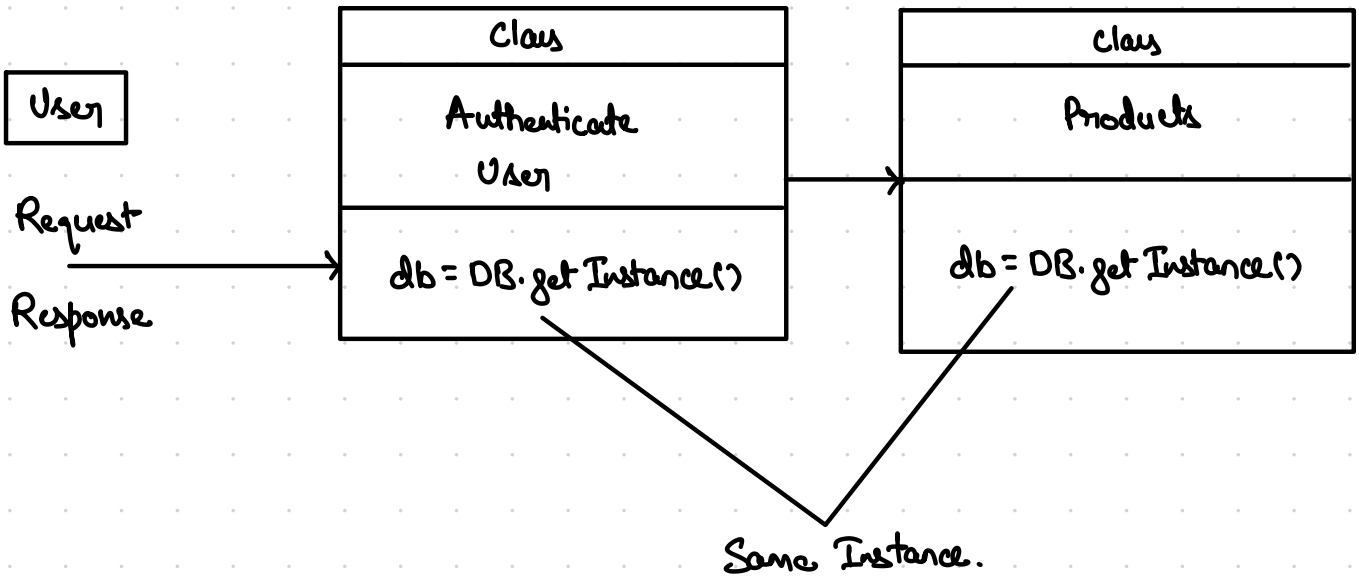
Singleton Pattern

Creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. The single instance is commonly used for managing shared resources, configuration settings, or logging functionality with in an application.

A common use case of the Singleton pattern is to use a single global instance of a database object throughout an application. This means that all clients that need to connect to a database will retrieve the same database object, and not to be creating new, separate ones. The database object is only created once, the first time it is needed, and then all other client that need to connect and query the database will use this same object.

Example: User making a request to fetch some products. There are two classes - User Authentication and Products. Both require fetching things from the same database.

We are not using the new keyword to get the database object, we are using a method called getInstance() that returns the exact same database object throughout the application, ensuring that we always use a single database connection:



- Why we can't just create a new database object every time we need to connect to the database. Some good reasons to use a single global database object in all client.

1. Resource Efficiency: Database connections and resources are typically limited and can be expensive to establish. By using a single instance of a database object, you minimize the overhead of creating and managing multiple connections, optimizing resource utilization.

2. Consistency and State Management: Having a single database instance ensure consistent state management and transaction handling across different parts of the application. Changes made to the database state are visible universally within the application, avoiding inconsistencies that could arise from multiple database instances.

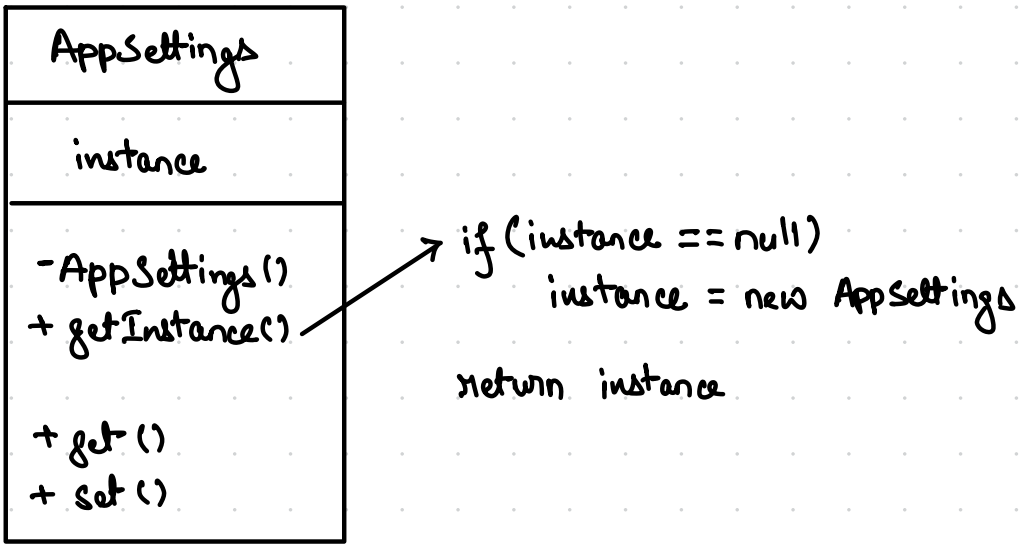
3. Simplified Configuration and Management: With a singleton database instance, configuration settings such as connection parameters, credentials, and initialization logic are centralized and managed in one place. This simplifies application setup and maintenance.

4. Performance Optimization: By reusing a single database instance, you can optimize database query performance and reduce latency associated with establishing new connections on reinitializing database resources.

The singleton pattern is great for storing app configuration settings, logging configuration, session information, authentication tokens - and making this information available globally via a single instance, ensuring that it is the same throughout the app.

Example: We need to keep an AppSettings object, that stores global variables such as the name of the app, the database configuration and logger settings. We need to create only a single instance of this object throughout our app to ensure that it only needs to be configured once in one place, and to ensure consistency throughout the app.

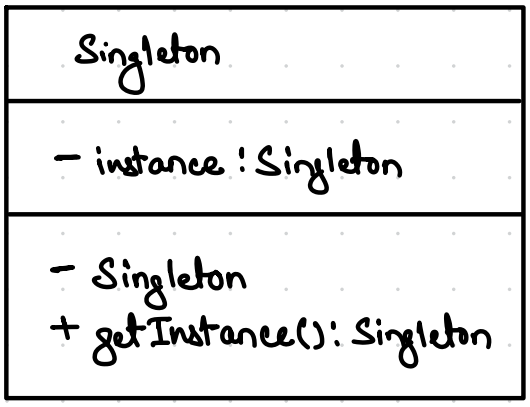
Solution: Singleton pattern ensure we have only a single instance of this class, first we have to make the construction private (notice -ve UML symbol) so we can't use the new operation with this class.



We also add a private static (symbolized by underlining) instance field that holds an instance of the AppSettings class - i.e. the class is responsible for maintaining a single instance of itself.

getInstance() is a static method for getting that single instance. Static because static fields are only visible to static methods.

GoF:



1. Make the construction private, so the new keyword can't be used to create multiple instance of the class.
2. Create a private static instance field to keep reference to the single instance.
3. Create a public static getInstance() for creating that single instance the first time the method is called in the application, then returning that same instance every time the method is called.

Factory Method Pattern

Factory method pattern

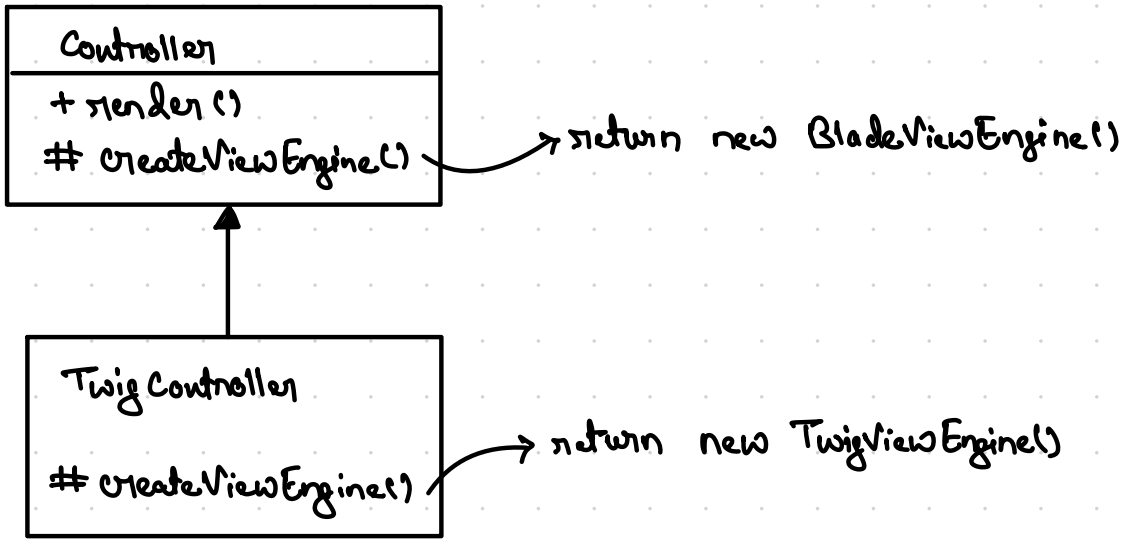
The factory method pattern is a creational design pattern that defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created, providing a way to delegate the instantiation logic to subclasses, enabling flexibility in object creation without changing the client code.

Example: We are developing a new Model-view-Controller backend framework, to rival the popular PHP framework, Laravel.

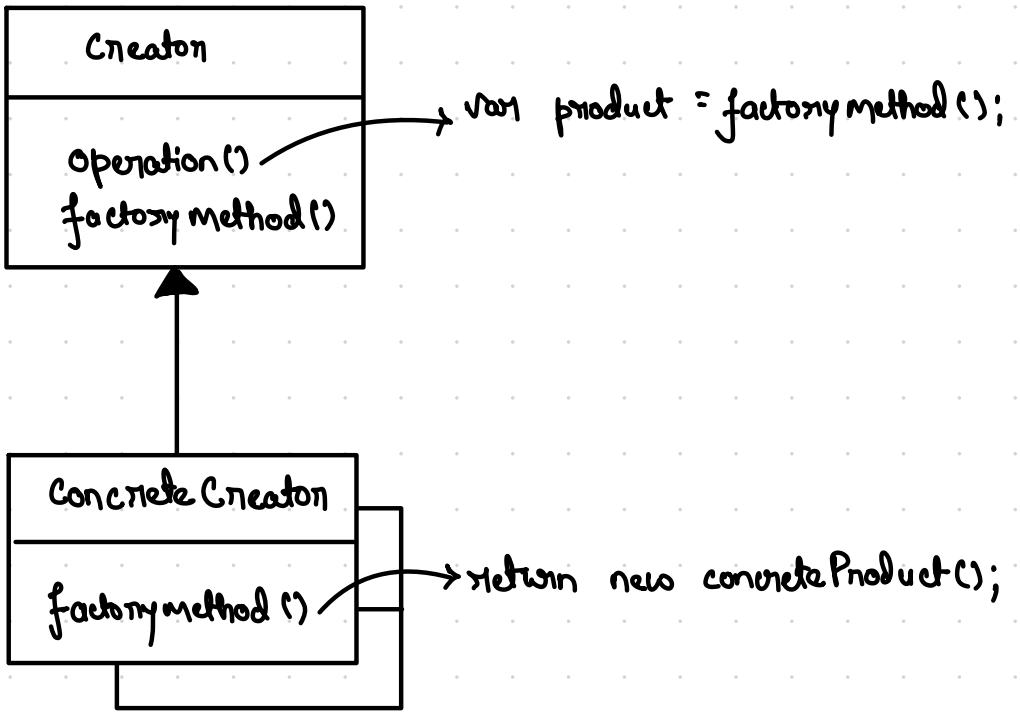
- We create a base Controller class to handle requests made to our application:
- Then, developers can extend the base controller class to create their own controllers to handle requests to their application - such as a controller that deals with order-related requests.

Solution: Create a factory method

We can add a createViewEngine() template method in the Controller class. By default, we return the Blade view engine. But if the developer wants to switch to a different view engine, such as Twig, then they can use our alternative controller class, TwigController, that overrides the createViewEngine() factory method:



GoF:



- The object returned from factory methods are referred to as "products".
- The Factory Method Pattern is often misunderstood. It relies on inheritance and polymorphism to add flexibility to the design. Inheritance allows methods to be overridden in subclasses, and polymorphism allows different objects to be returned from the overridden methods.
- Many people implement the Factory Method Pattern incorrectly, e.g. using a static method, such as:

```
var engine = ViewEngineFactory.createViewEngine();
```

- But static methods cannot be overridden, so there is no flexibility with this approach, and it isn't correct. We cannot change the implementation of the createViewEngine() method.
- Using the factory method pattern, we can defer the creation of an object to subclasses, and this is possible through inheritance.