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

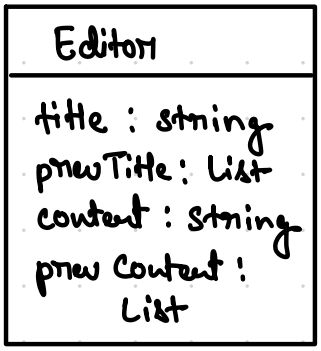
Memento Design Pattern

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.

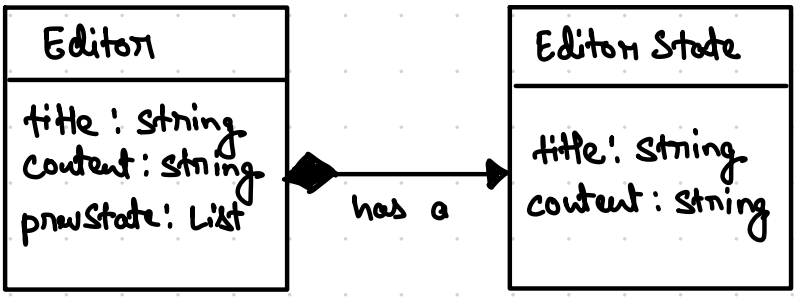
- simple way to implement
  - create a single Editor class and have a field for title and content, and also have a field that stores each of the previous value for each field in some list.



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.

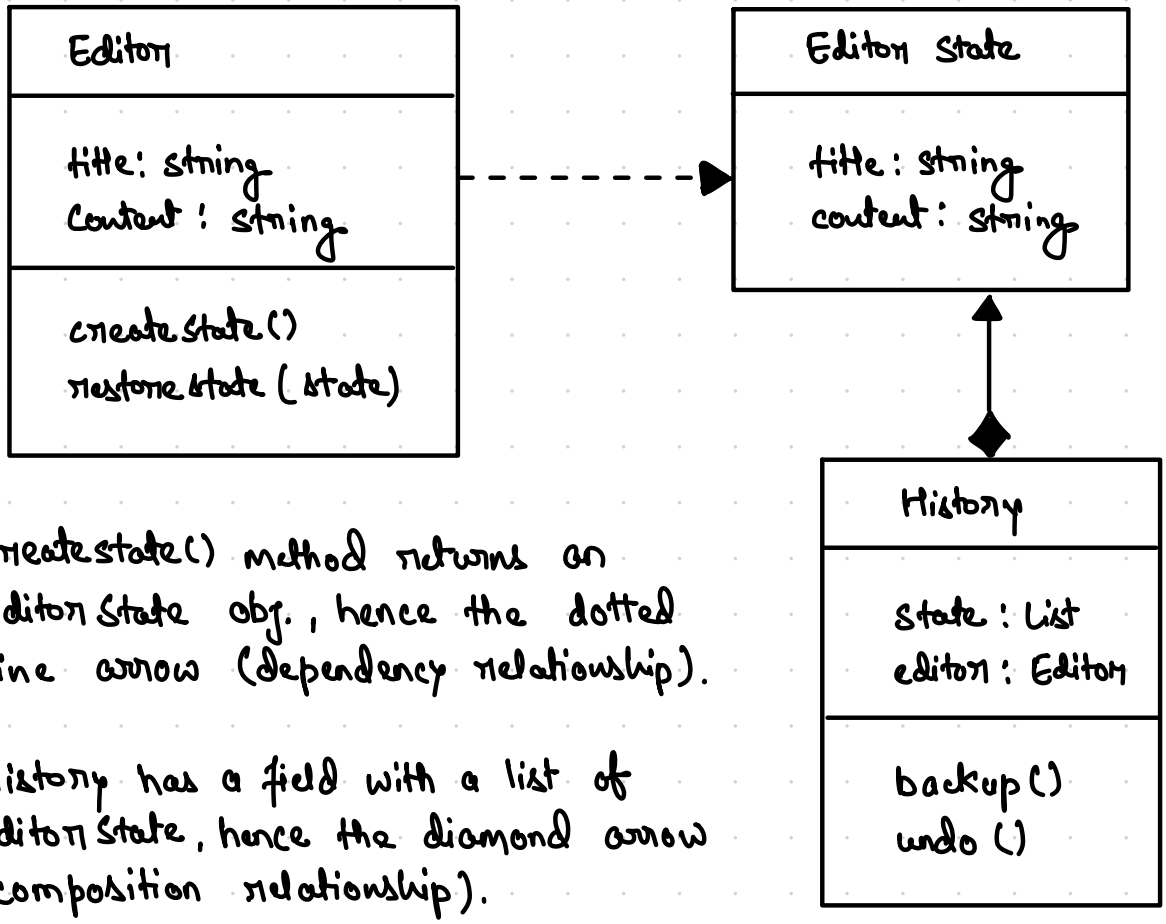


composition relationship:
Editor is composed of, or has a field of the EditorState class.

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

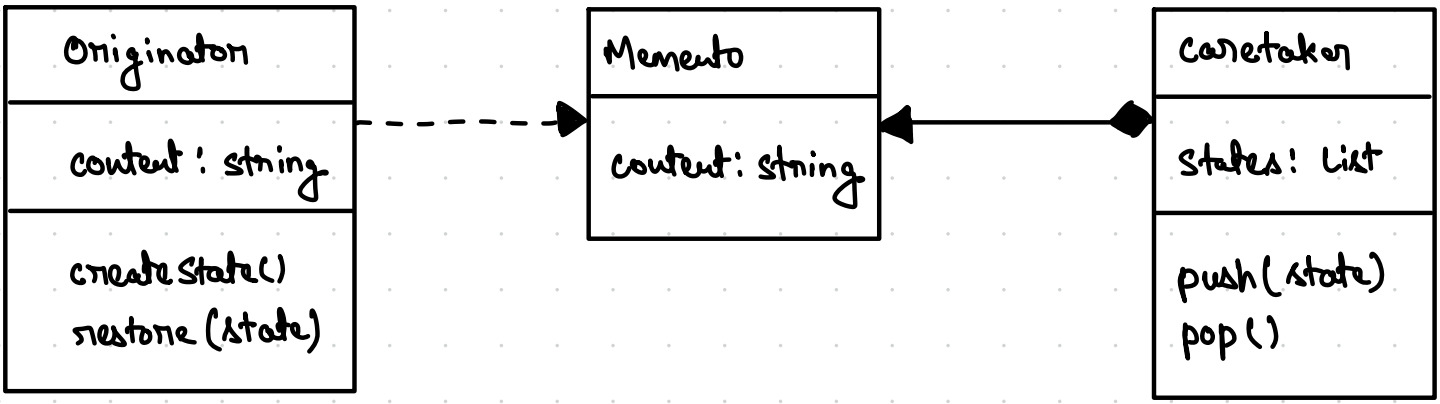
Problem: violating the SRP, as our Editor class has multiple responsibilities
1. State management
2. Providing the features that we need from an editor.

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.

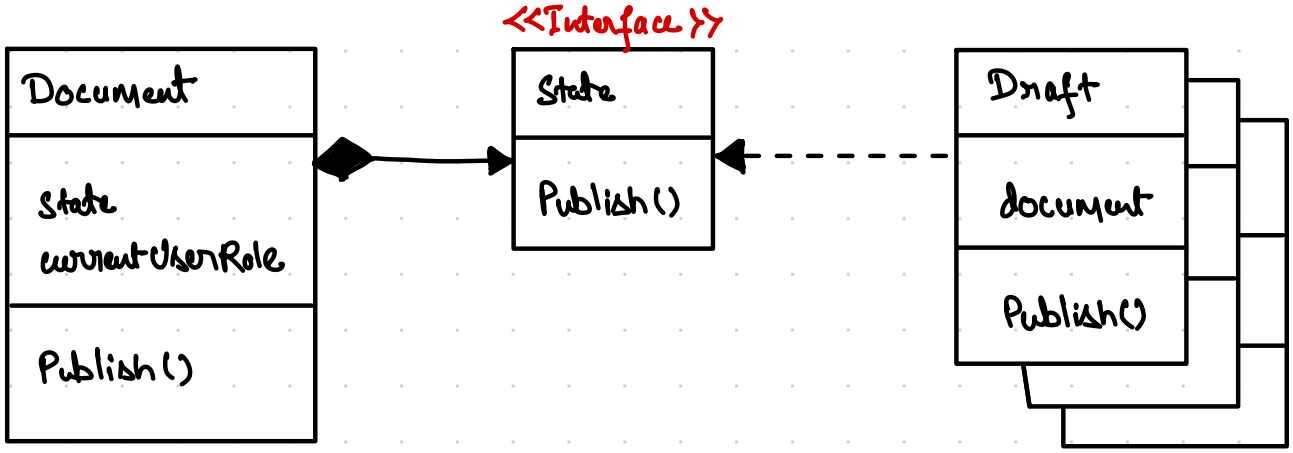
- Simple Solution :-
using if/else statements to check the current state of the post to see whether the state of the document should be upgraded.

- 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.

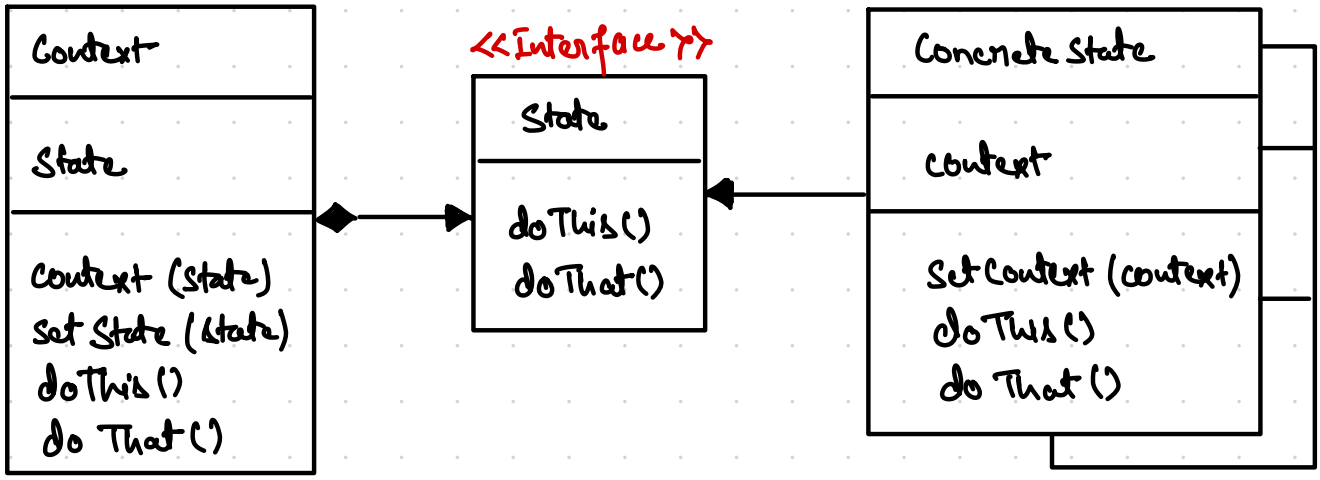
- We are using polymorphism, as the state field can be any one of the concrete state classes (Draft, Moderation, Published), as we are coding to an interface, not concrete classes.

- 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

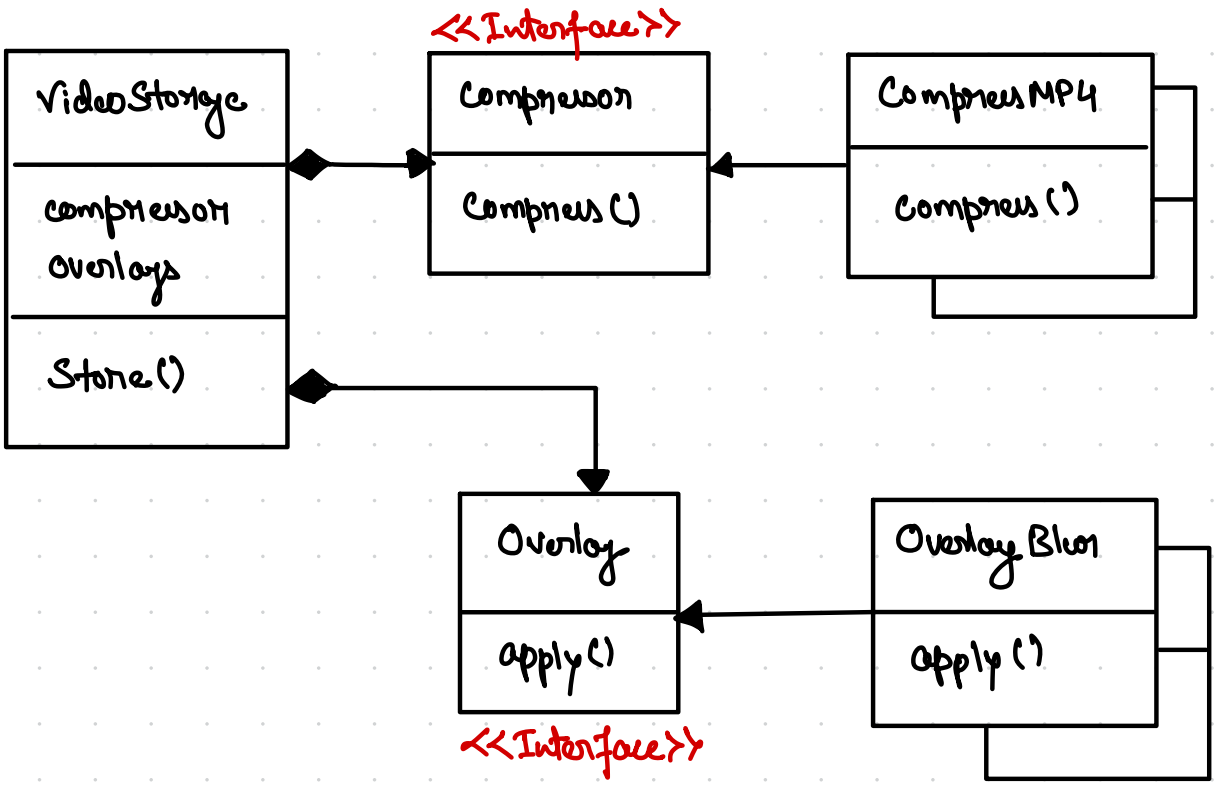
VideoStorage(compresson, overlay)
Set Compresson (compresson)
Set Overlay (overlay)
Store (filename)
↳ //compress
//apply overlay
// store

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

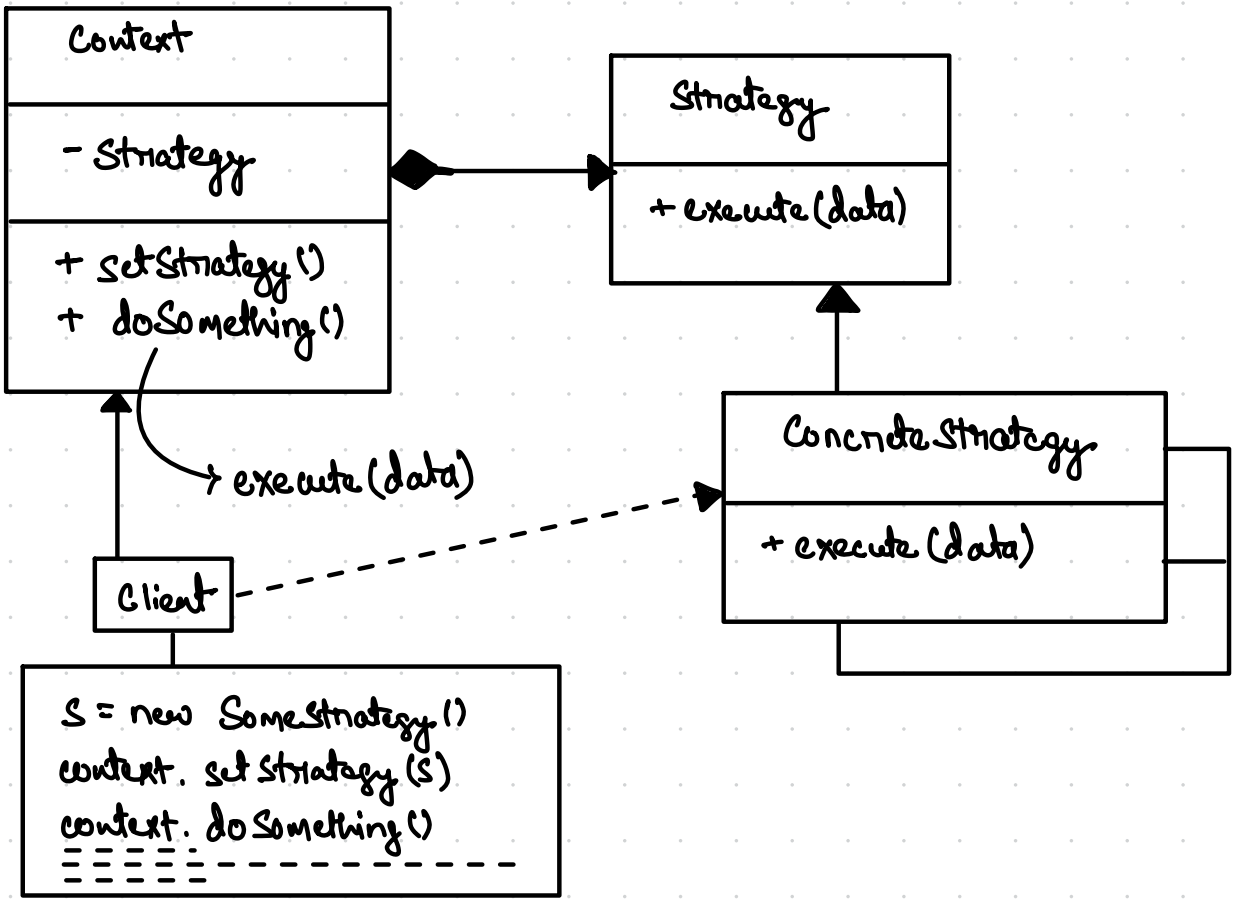
- We can create a Video Storage object, we pass it the concrete compresson and overlay objects that we want to use.

- 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.

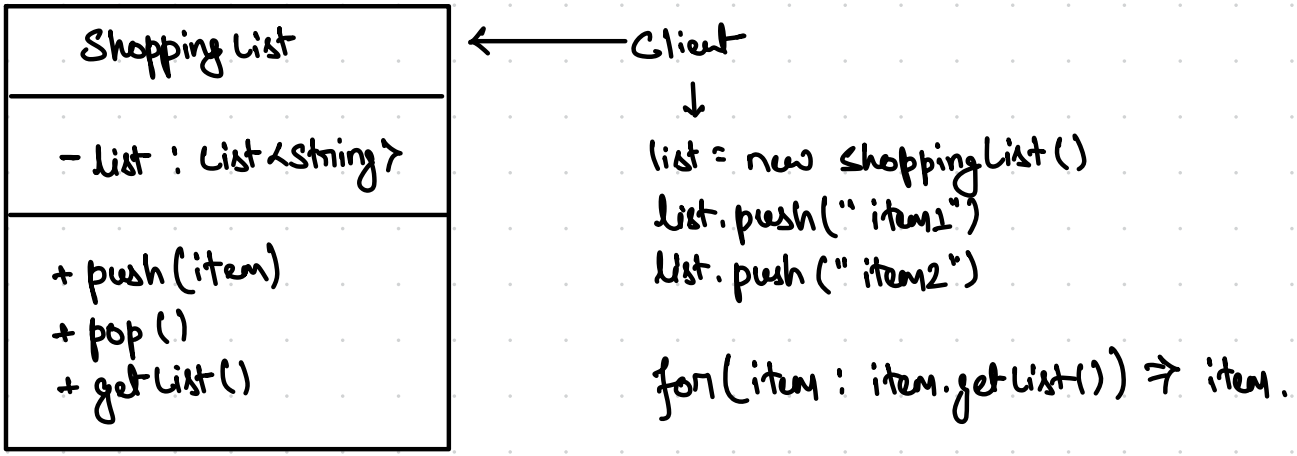
When to use:-
- When you have a class with a large no. of conditional statements that switch b/w variant of the same algorithm.
- The algorithm logic can be extracted into separate class that implement the same interface.
- The context object then delegates the work to these classes, instead of implementing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

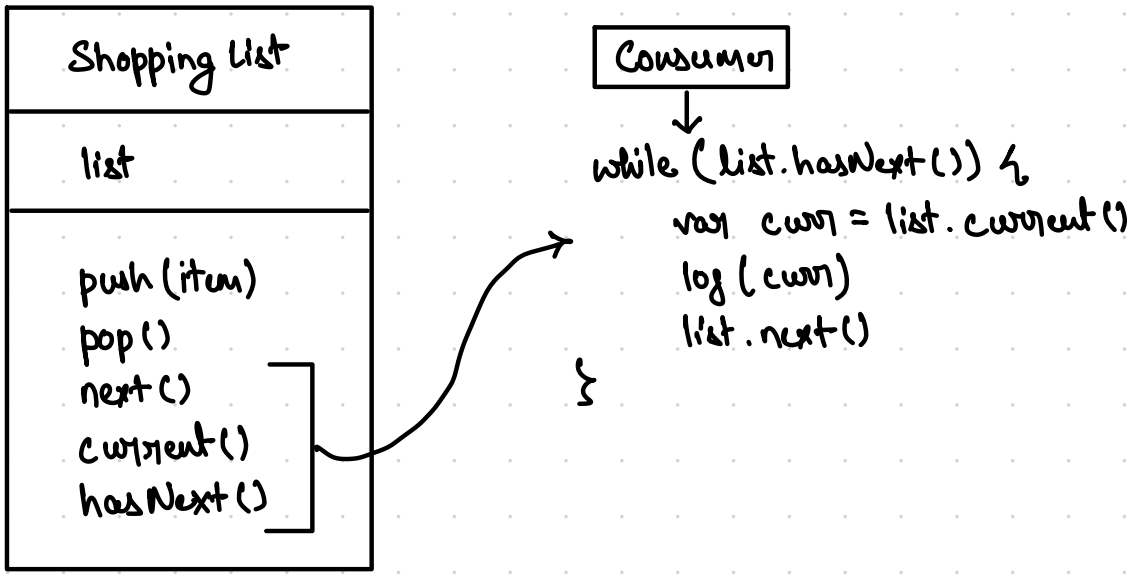
Iterator Design Pattern

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.



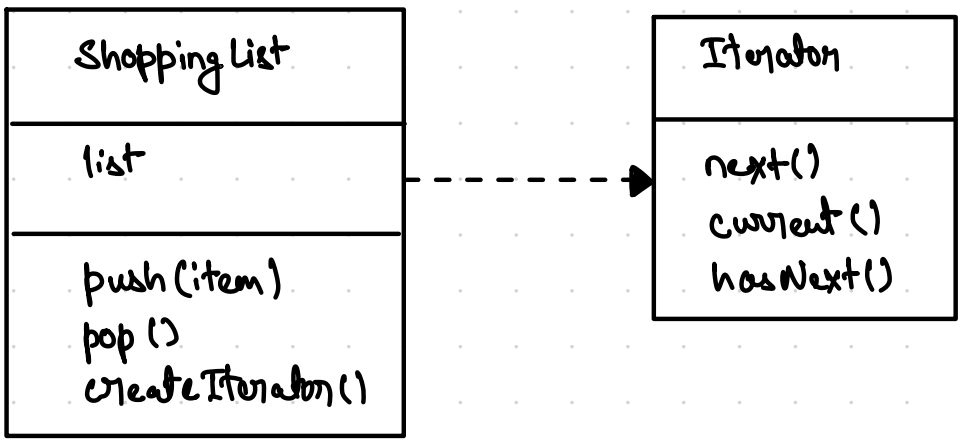
- added three new methods to help consumer to iterate over the object, without knowledge of internal data structure.

  - 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.

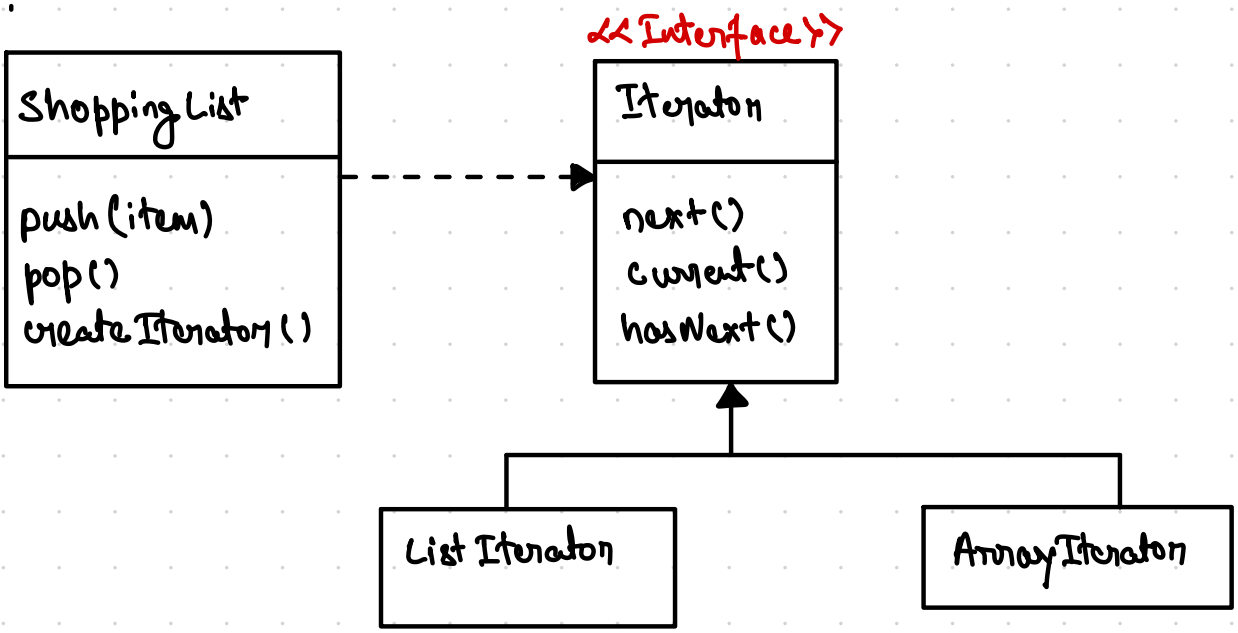


- The createIterator() method returns an instance of Iterator that allows consumer to iterator over shopping lists without knowing internal details.

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.

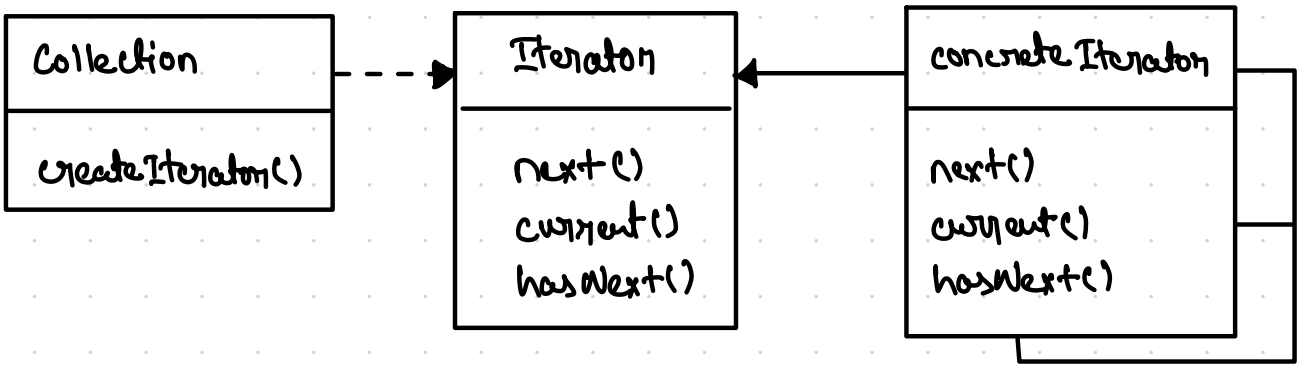
The interface determines the capabilities we need from a real concrete iterator. The data structure could be Array, List, Stack etc.

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.

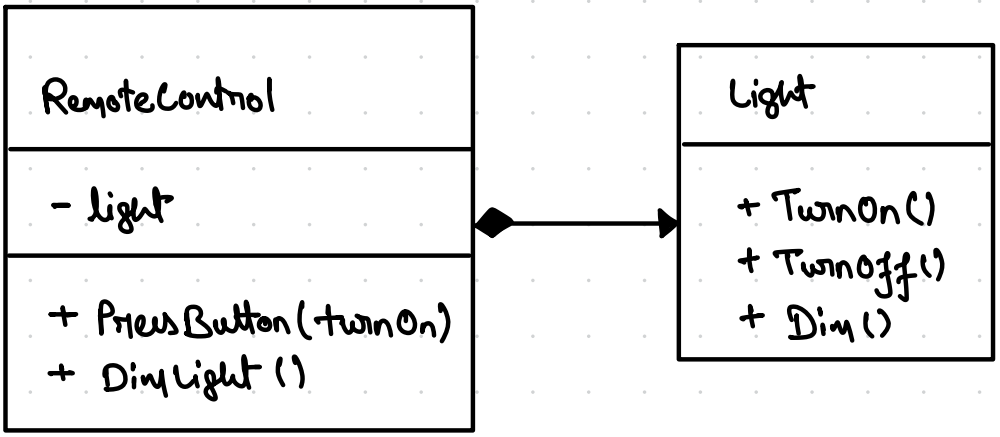
Command Design Pattern

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.

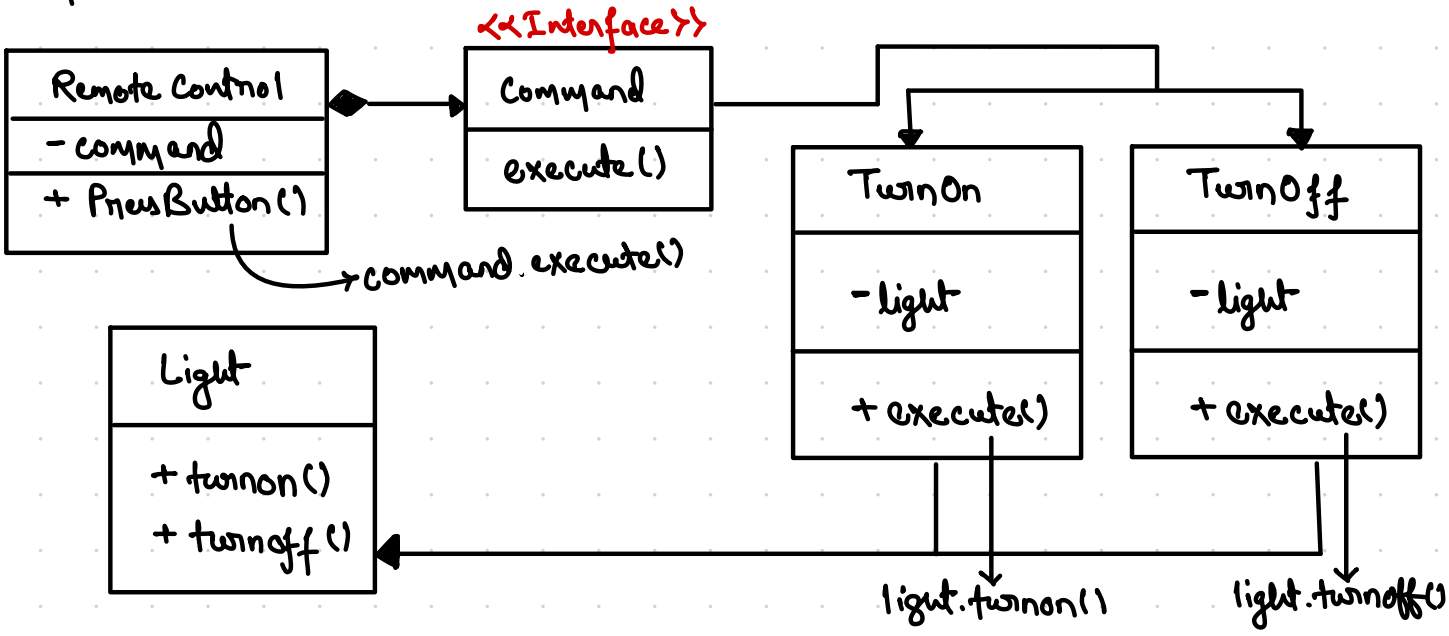
- It decouple the sender from the receiver, providing flexibility in the execution of commands and supporting undoable operations.

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:-

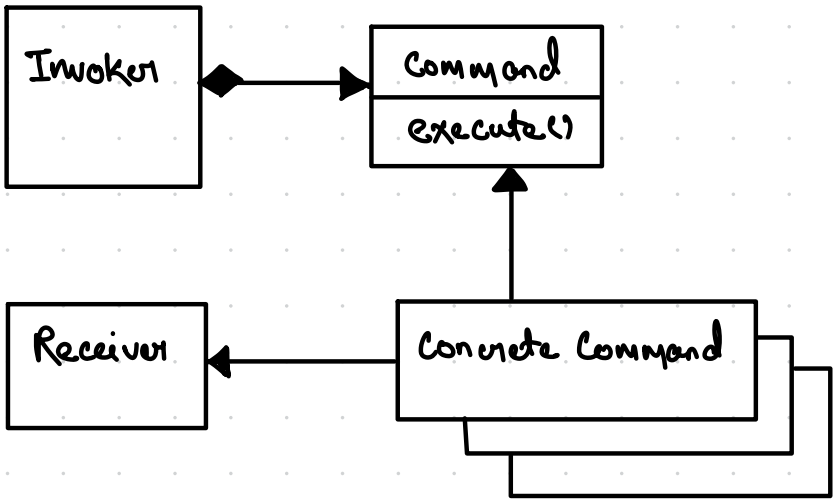


- Using the Command Pattern, we decouple RemoteControl (the sender) from Light (the receiver).

- 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

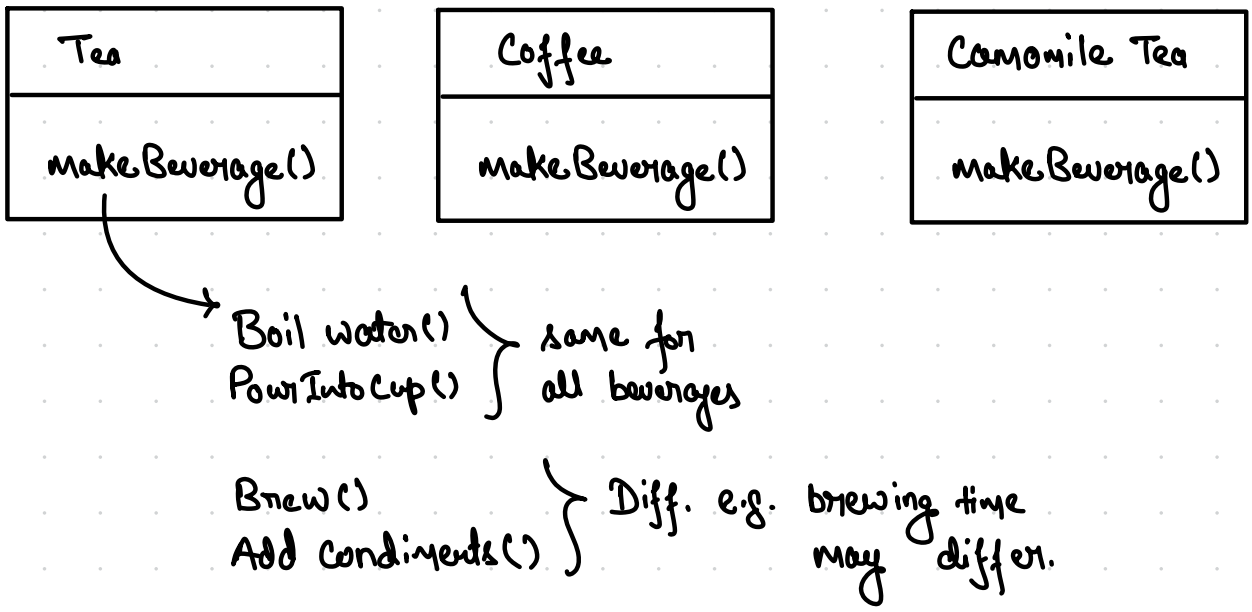


Template Method Design Pattern

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:



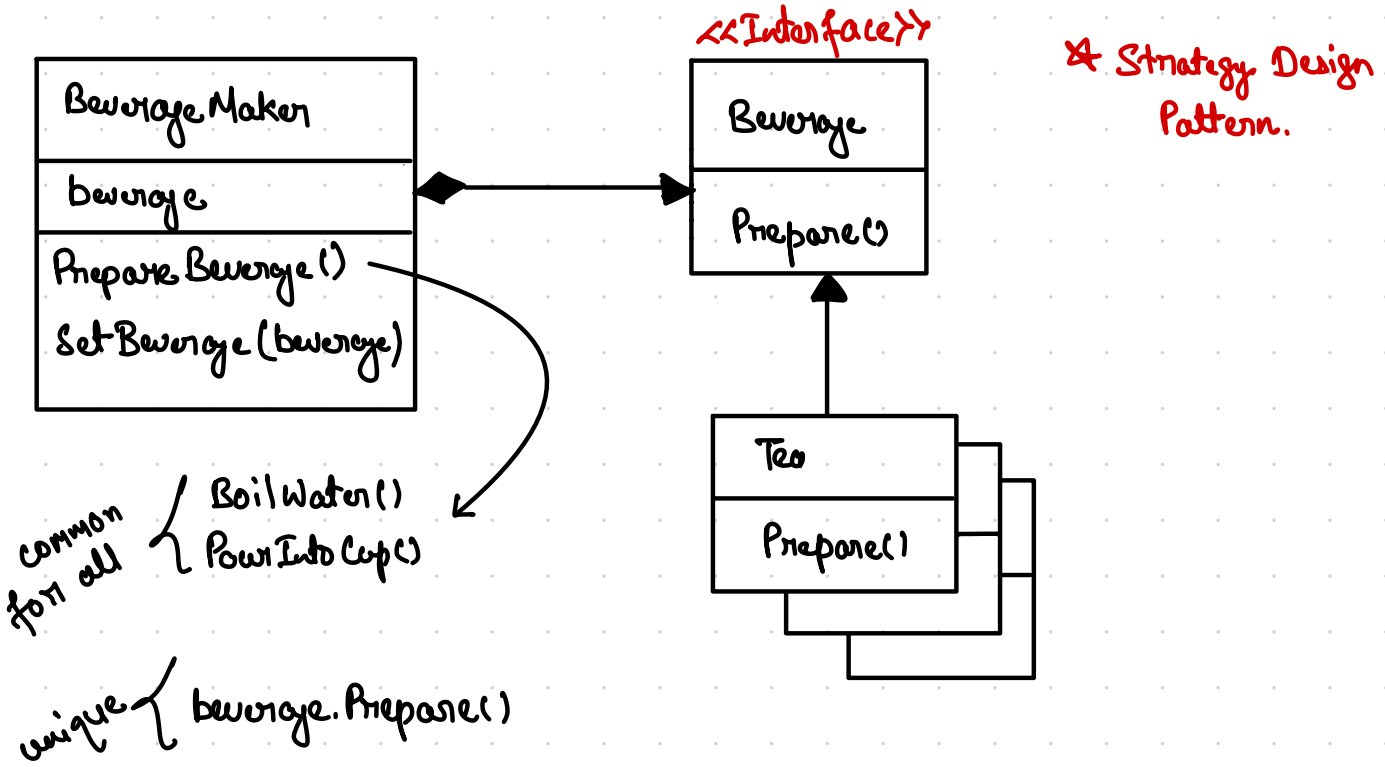
- We started out simple, making a separate class for each hot beverage.

- 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.

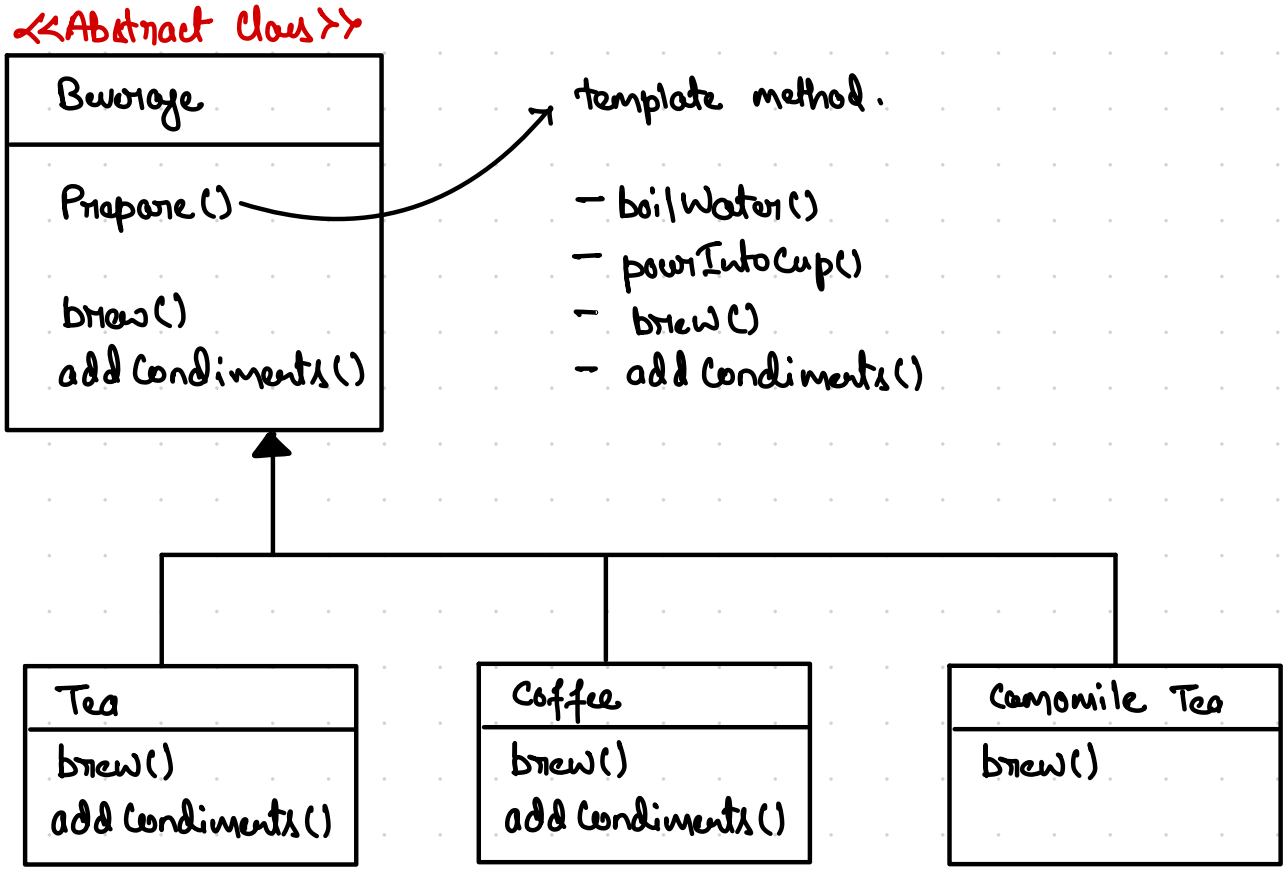
2. Inheritance

- 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.

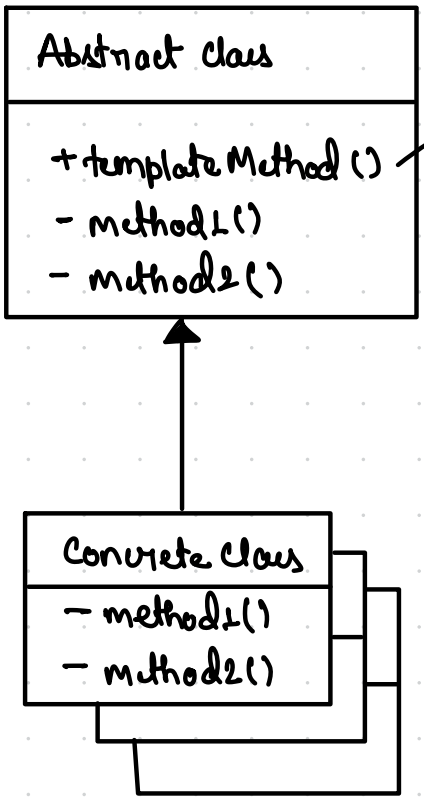
- These beverage-specific steps will be determined later on, when the beverage class is extended.

- 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

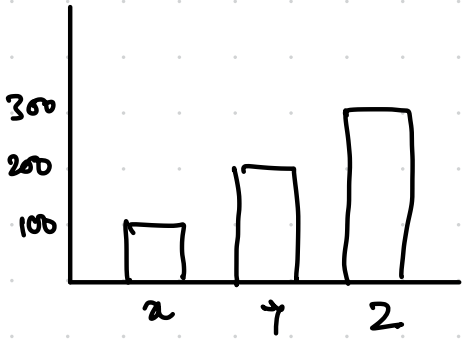
- The observer pattern involves an object, known as the subject, maintaining a list of its dependent objects, called observers, and notifying them automatically of any state changes.

- Situation

Data Source

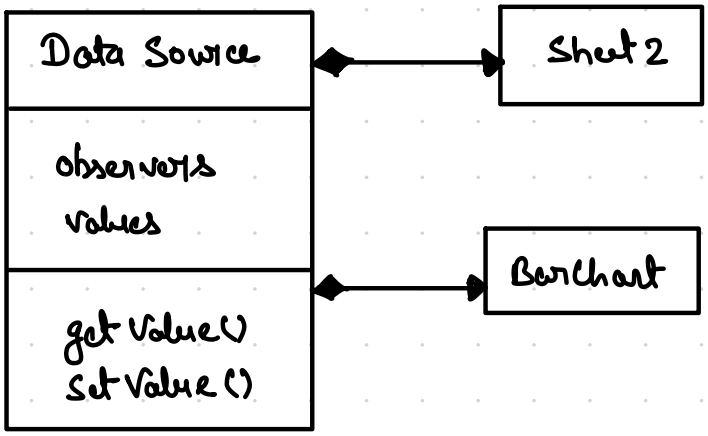
City	No. of dogs.
x	100
y	200
z	300

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.

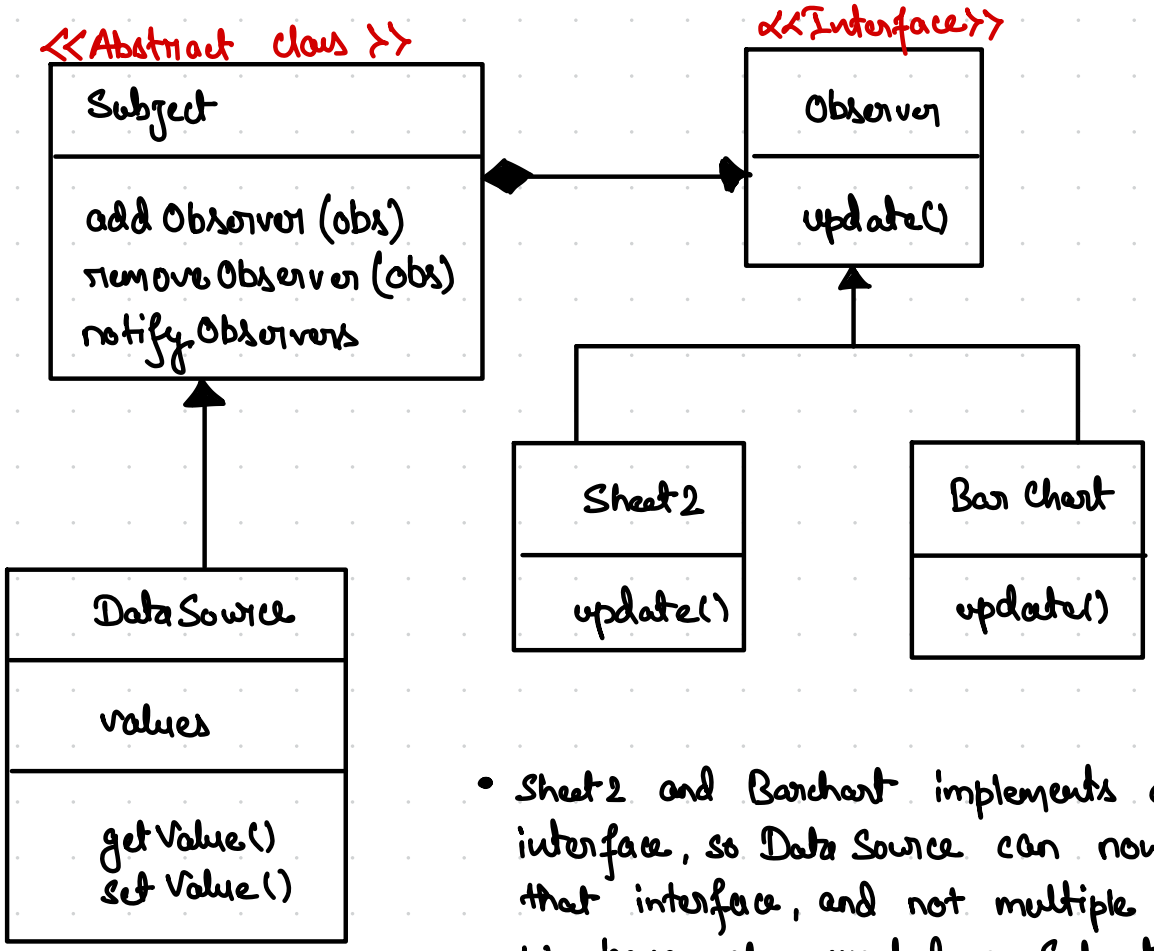
- OCP : every time we create a new observer object, we have to modify DataSource. This is because we are programming to concrete objects, rather than to a generic interface.

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.

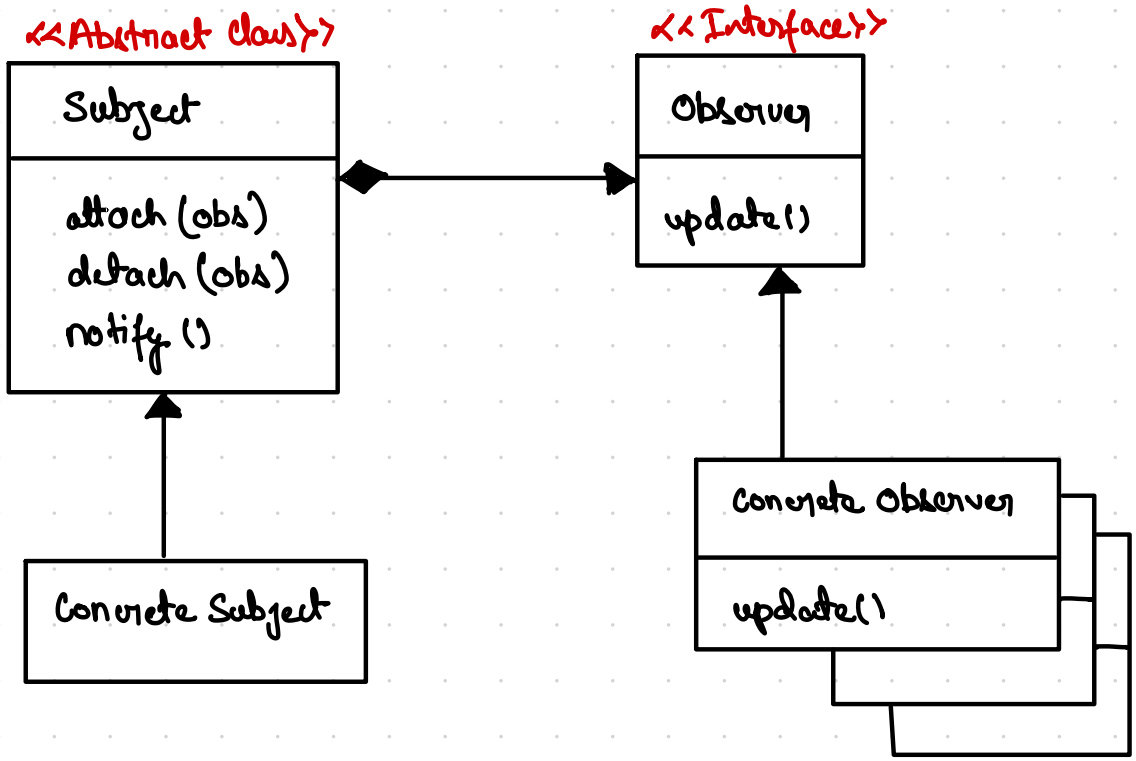
Observer Pattern.



- 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.

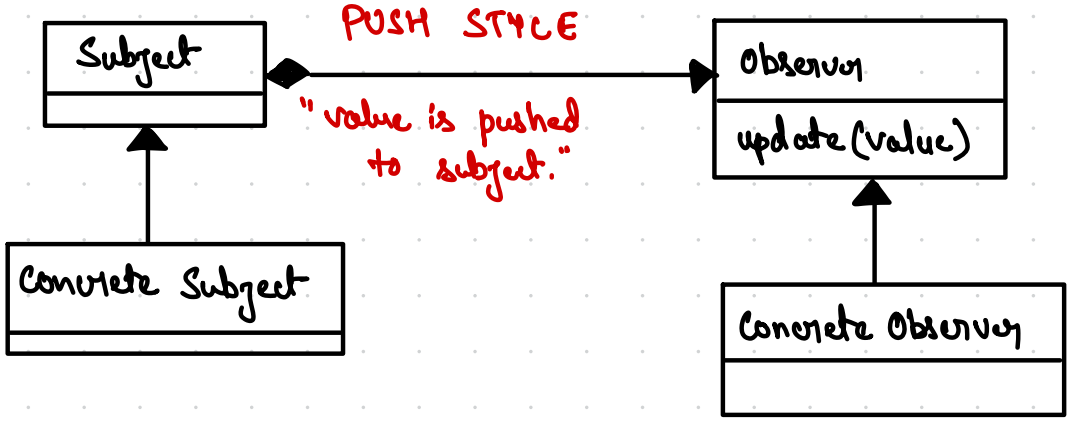
- GoF Observer Pattern



- 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.

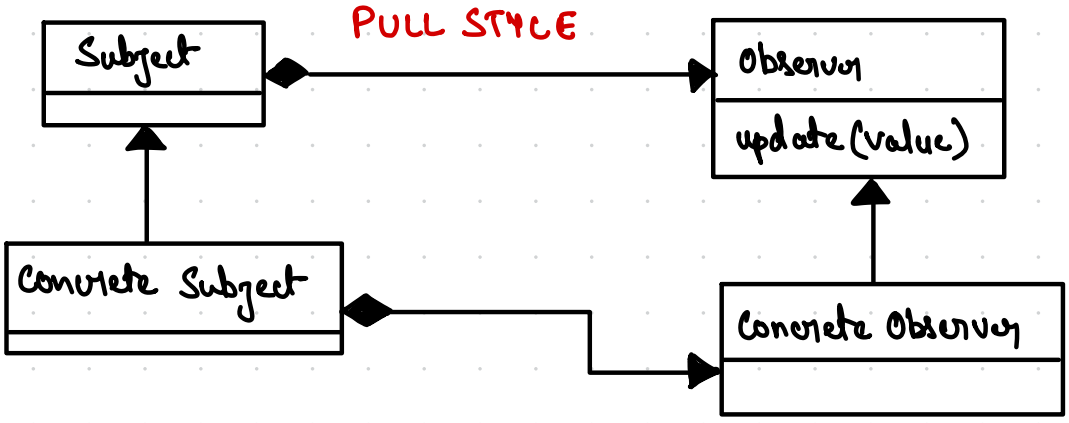


- Value could be any object or generic type.

- 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.

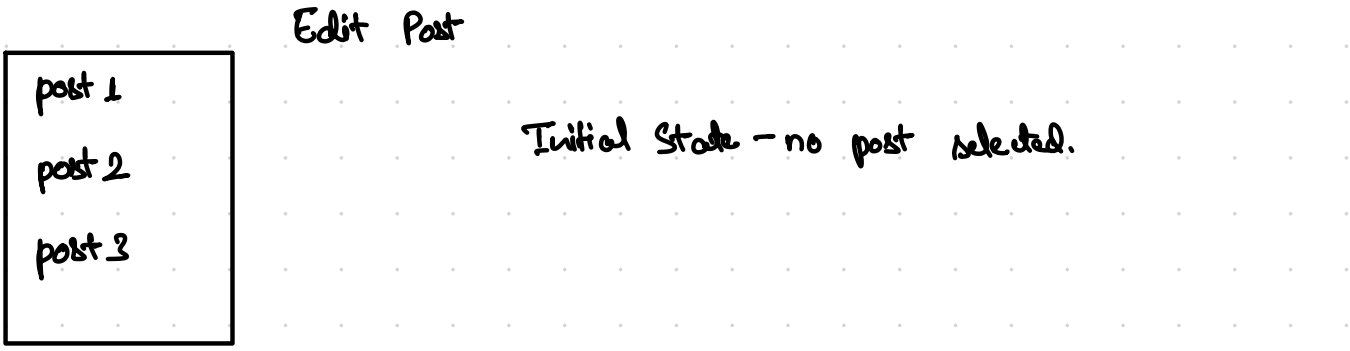
Mediator Pattern

Mediator Pattern

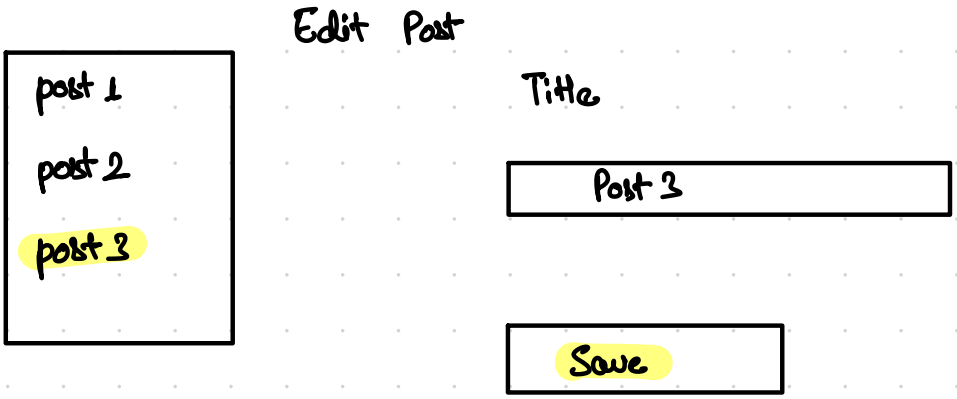
- It defines an object (the Mediator) that describes how a set of objects interact with each other, therefore reducing lots of chaotic dependencies b|w those objects.

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:

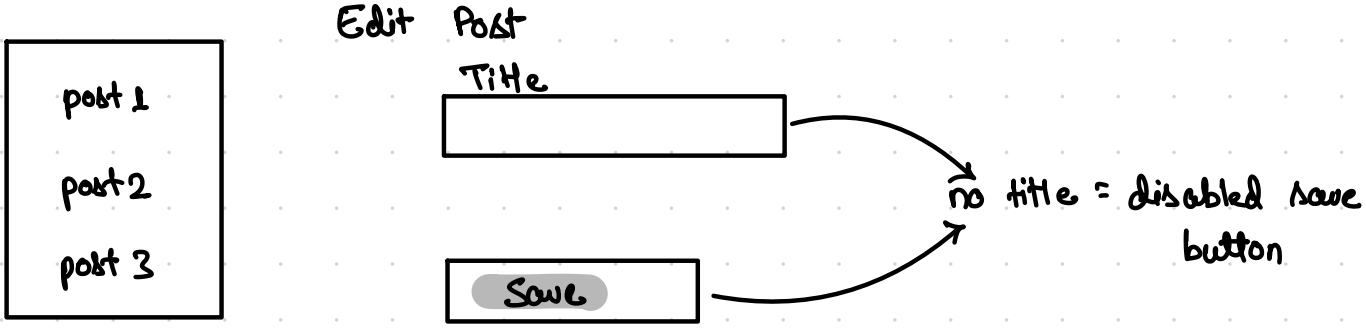


Select an article from the postsContainer on left and the input is populated on right with the title.



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

State 3



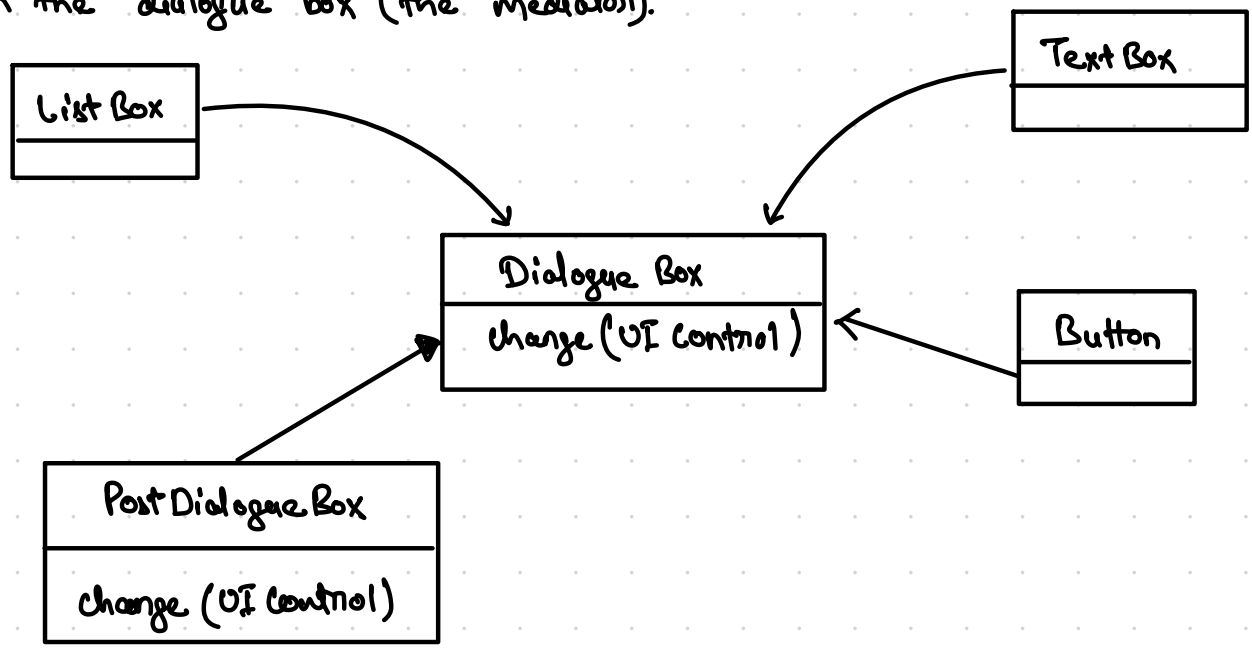
Components (classes) that we need.

- 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.

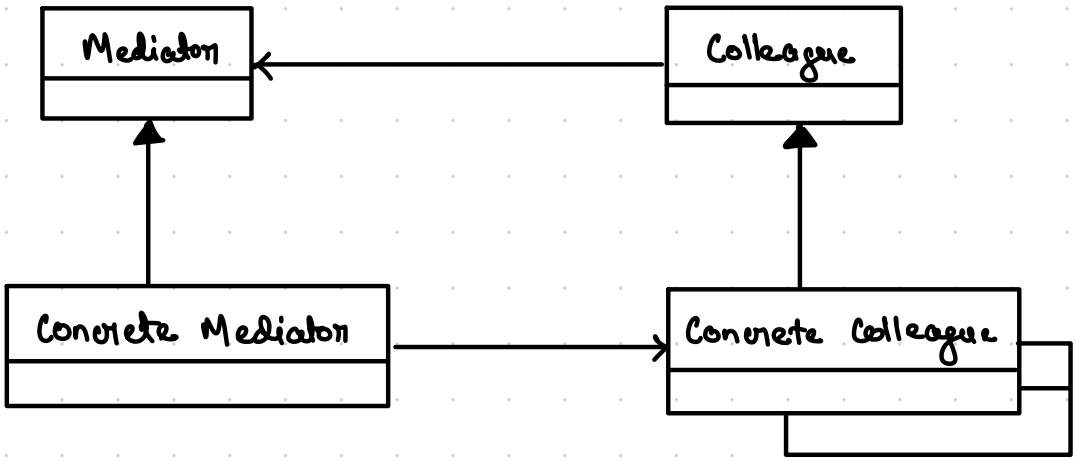
When an article is selected from the list box, the text box should be populated, and the button enabled. When we clear the text box, the button should become disabled.

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.

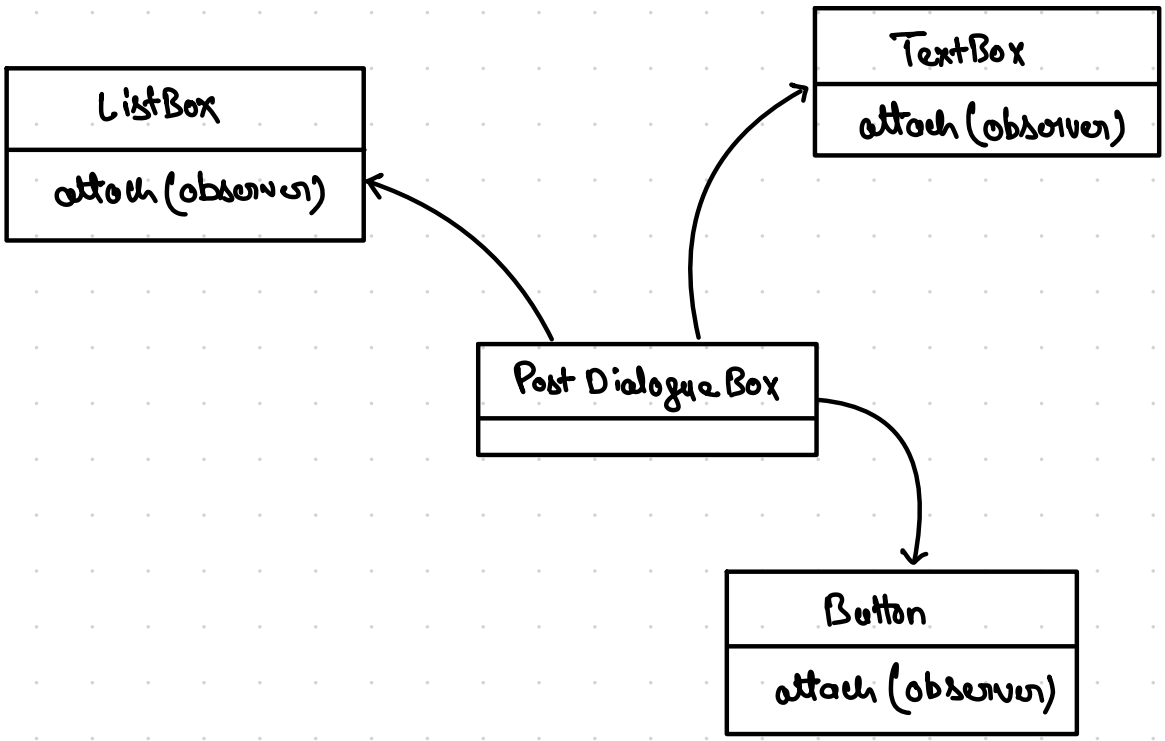
Mediator Pattern with Observer Pattern

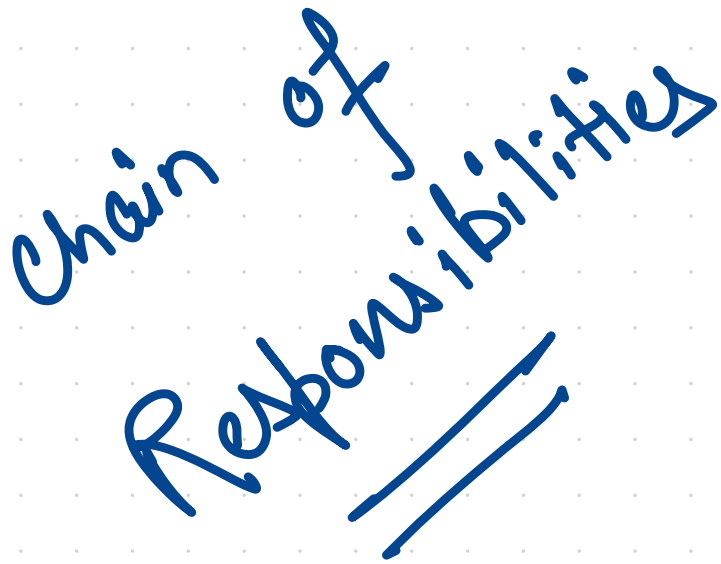
One problem with our previous solution is that the change() method on PostDialogueBox can get bulky as we add more UI components - lots of if/else to see what component has changed.

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

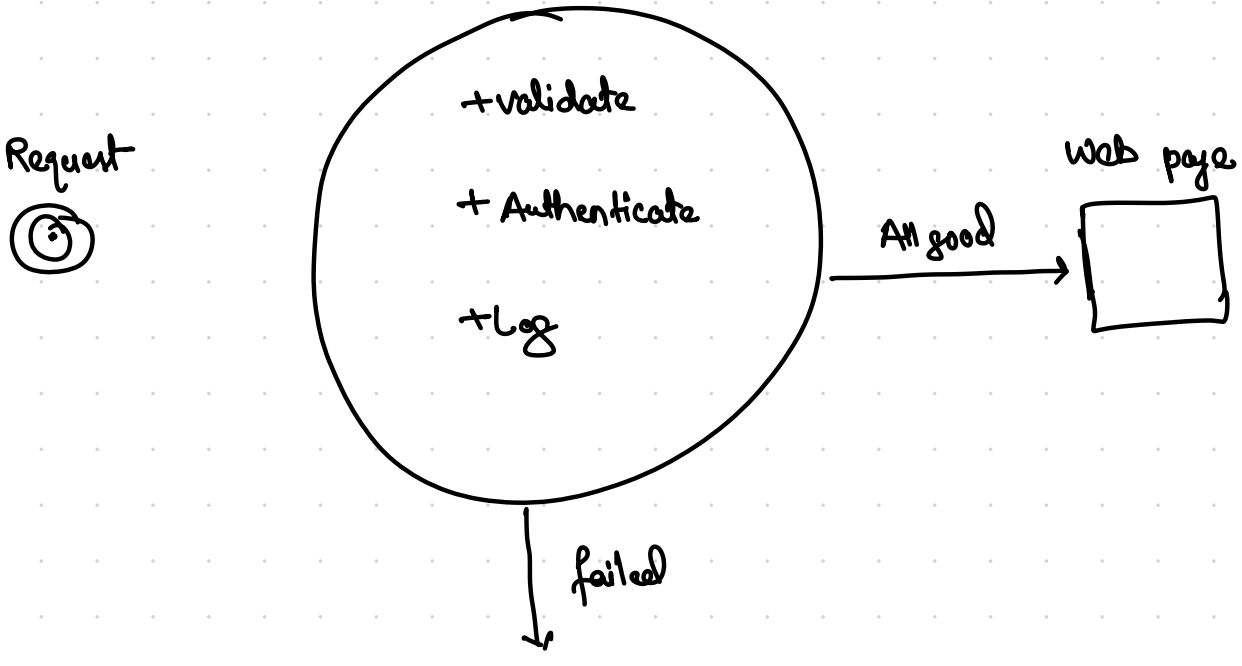
- The chain of responsibility pattern allows building a chain of objects to handle a request.

- 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.

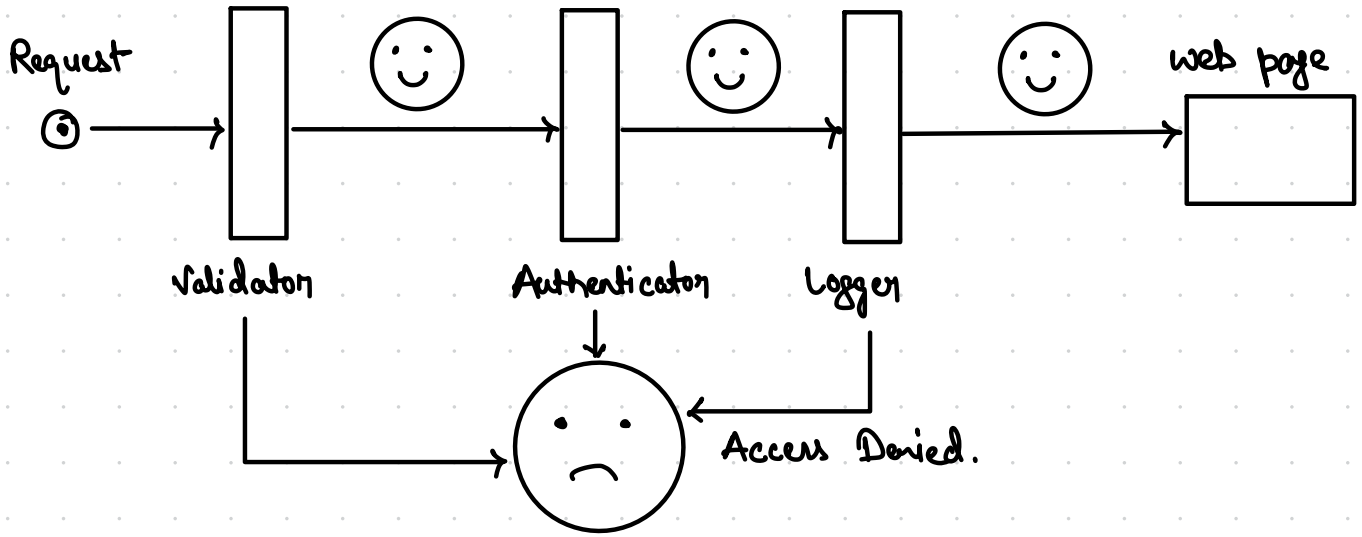
Let the user makes a request to the website server, but before returning the web page, the user's data must be validated (eg. trim any whitespace), authenticate the user, and then log some information onto the server about the request.

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



Chain of Responsibility solution.

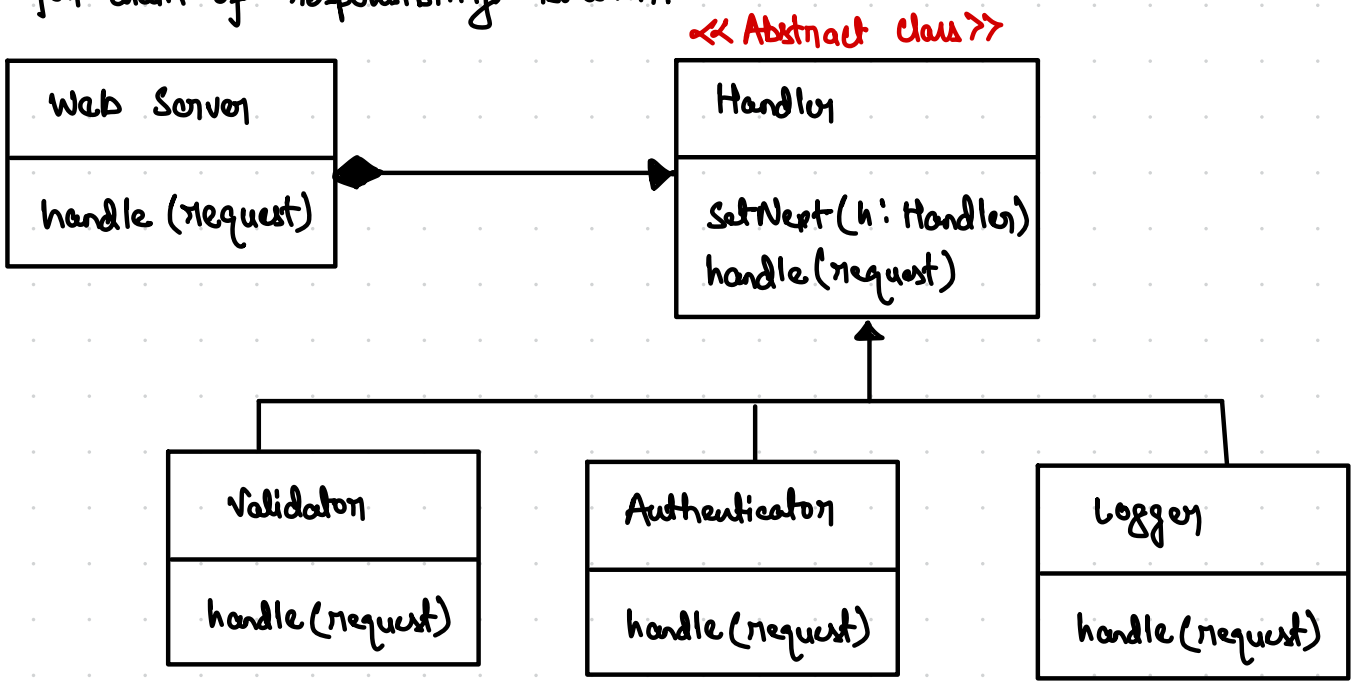
- Instead of having all our request processing logic inside of the one method, we can create a processing pipeline – a chain of objects.



- 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