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

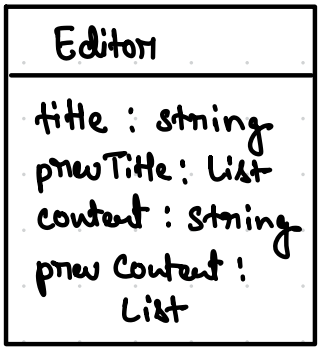
Memento Design Pattern

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.

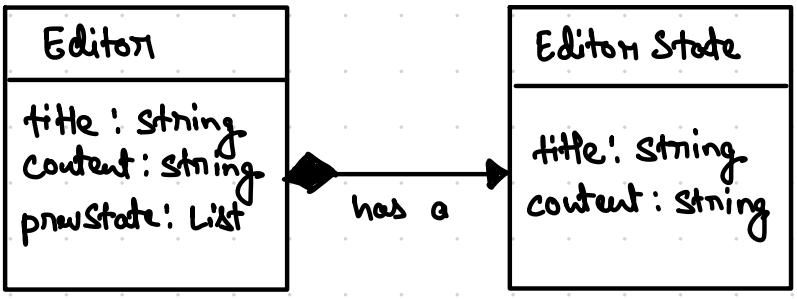
- simple way to implement
  - create a single Editor class and have a field for title and content, and also have a field that stores each of the previous value for each field in some list.



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.

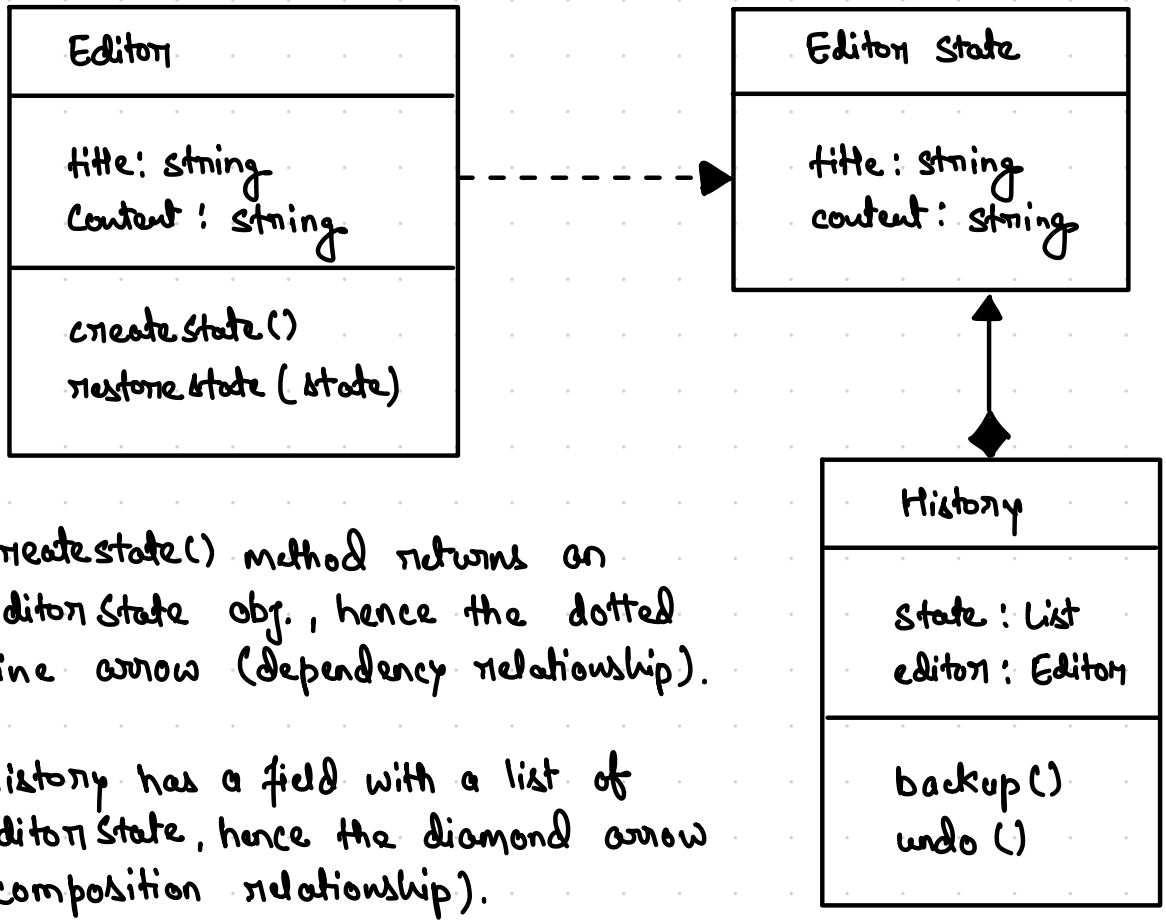


composition relationship:
Editor is composed of, or has a field of the EditorState class.

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

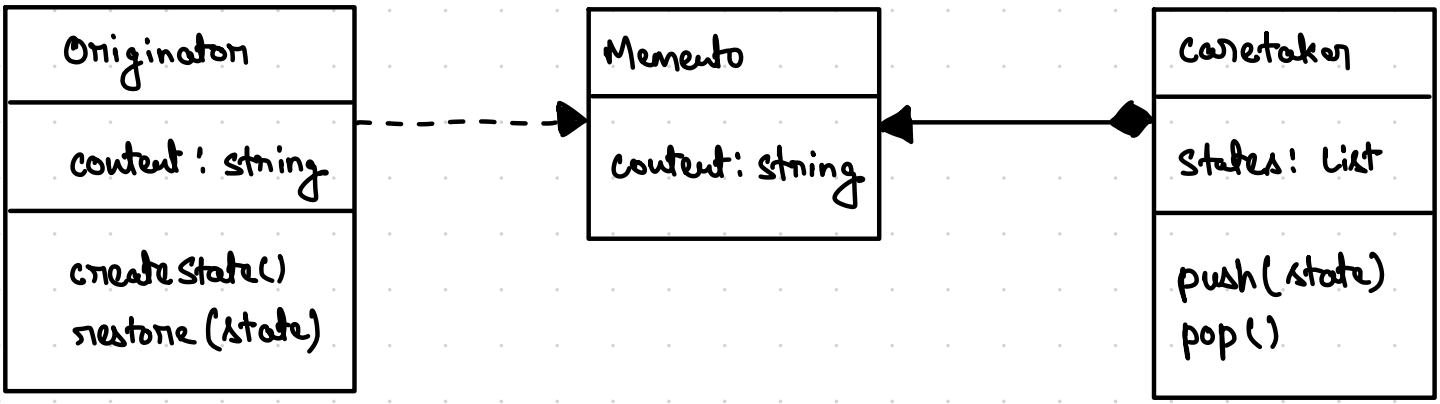
Problem: violating the SRP, as our Editor class has multiple responsibilities
1. State management
2. Providing the features that we need from an editor.

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.

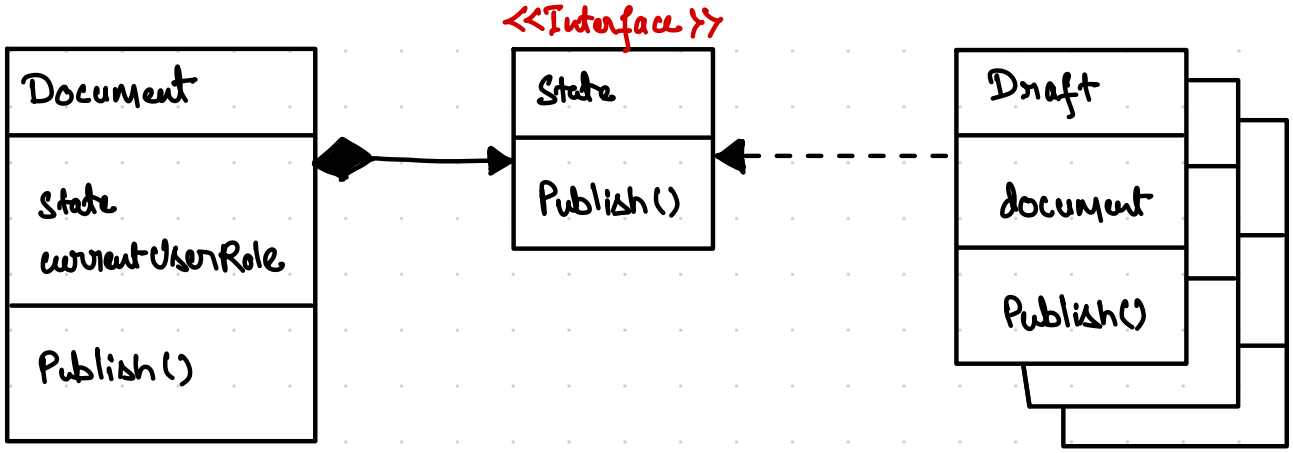
- Simple Solution :-
using if/else statements to check the current state of the post to see whether the state of the document should be upgraded.

- 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.

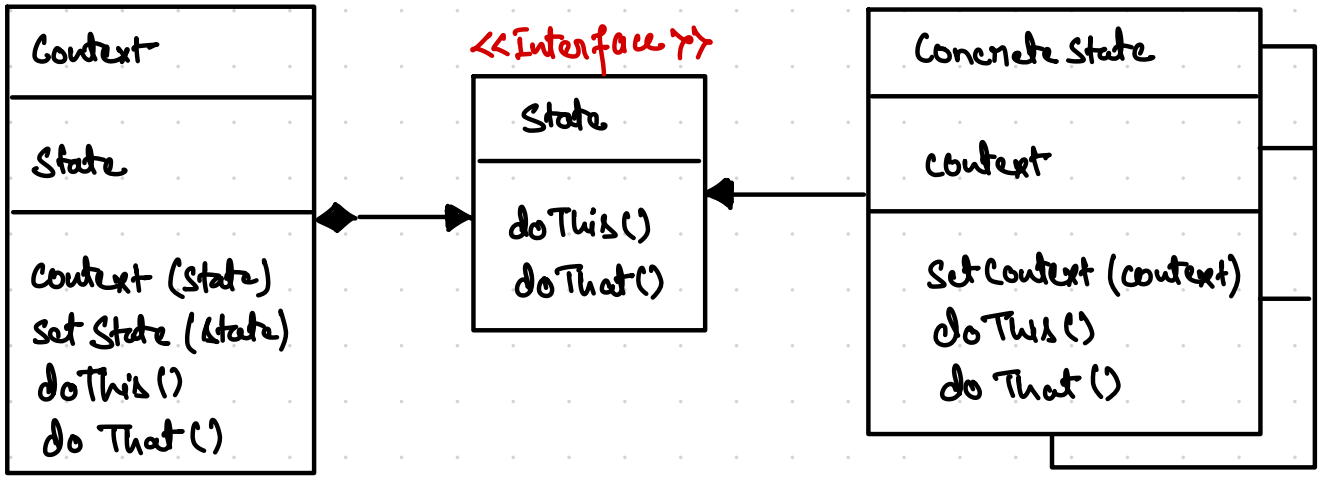
- We are using polymorphism, as the state field can be any one of the concrete state classes (Draft, Moderation, Published), as we are coding to an interface, not concrete classes.

- 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

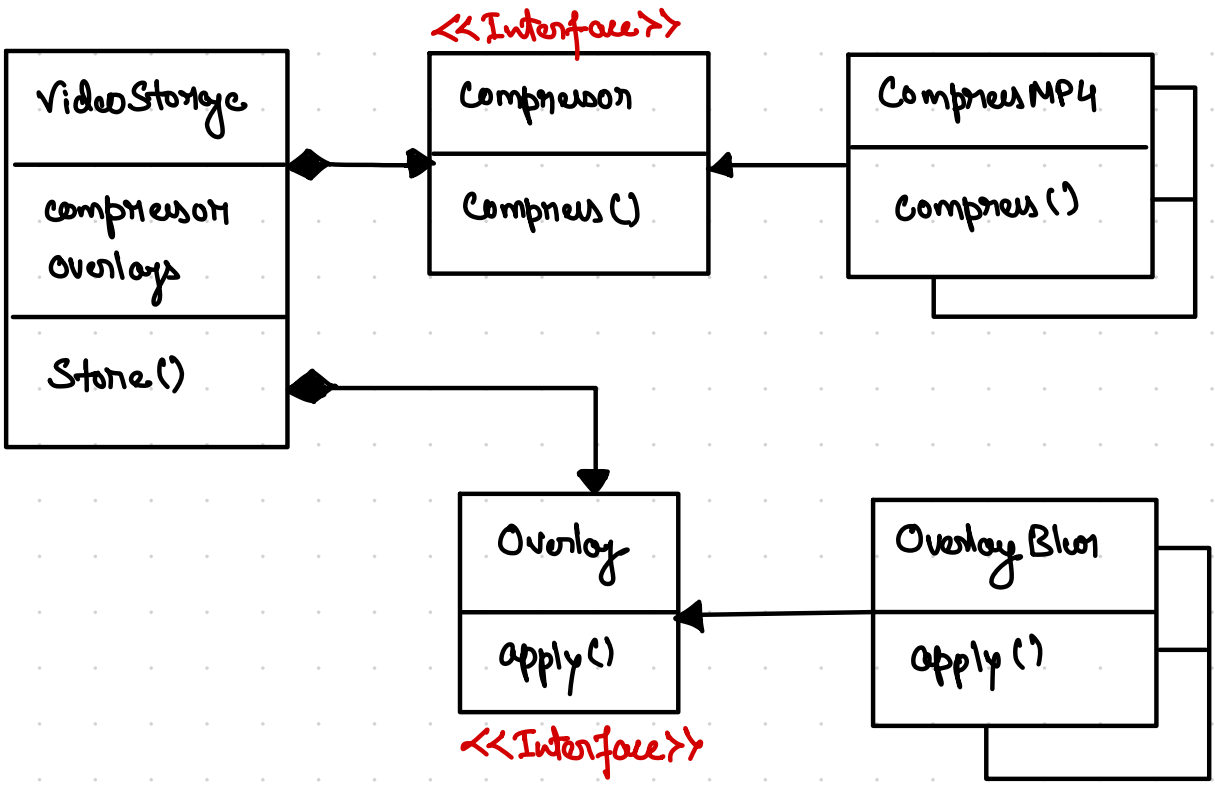
VideoStorage(compresson, overlay)
Set Compresson (compresson)
Set Overlay (overlay)
Store (filename)
↳ //compress
//apply overlay
// store

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

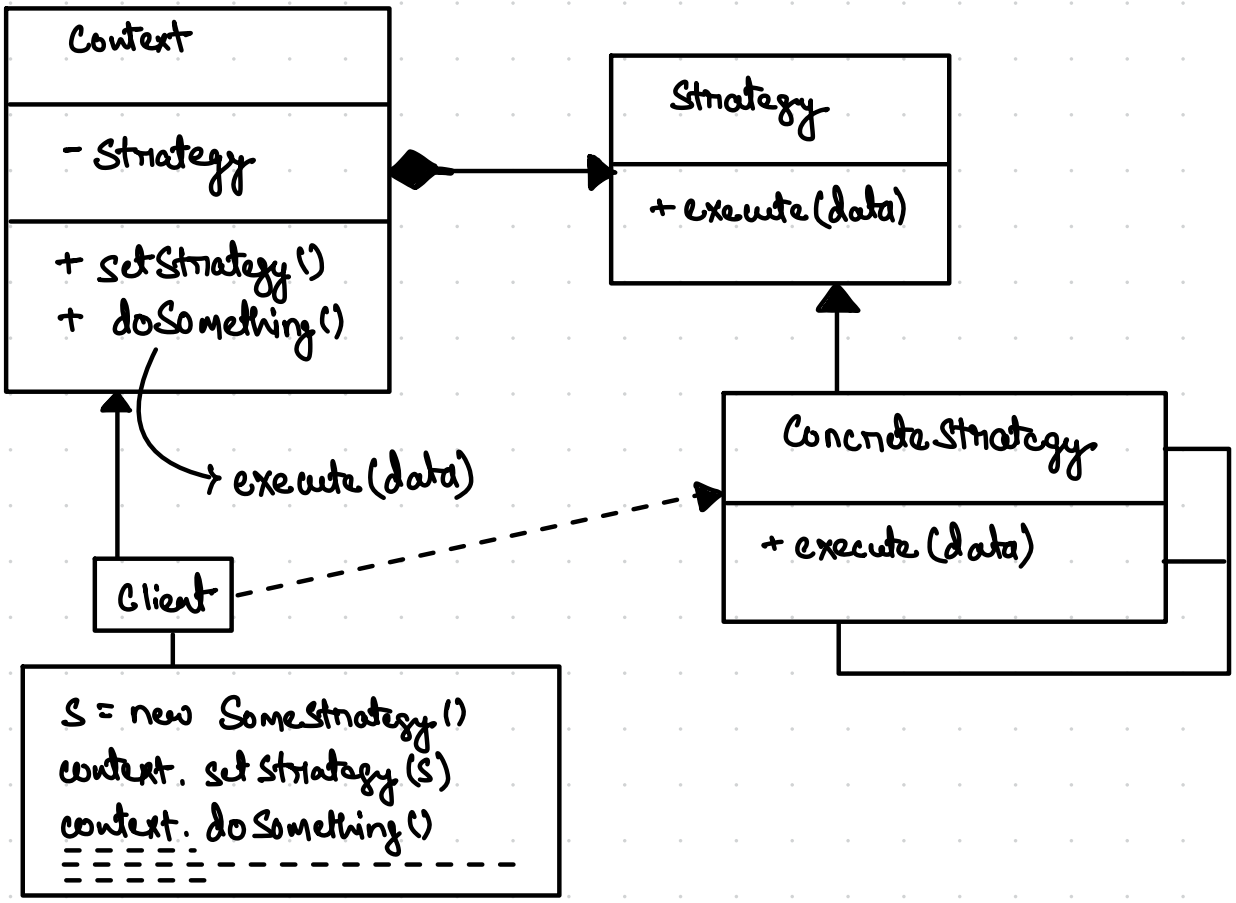
- We can create a Video Storage object, we pass it the concrete compresson and overlay objects that we want to use.

- 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.