Software Design Patterns

Lecture 11

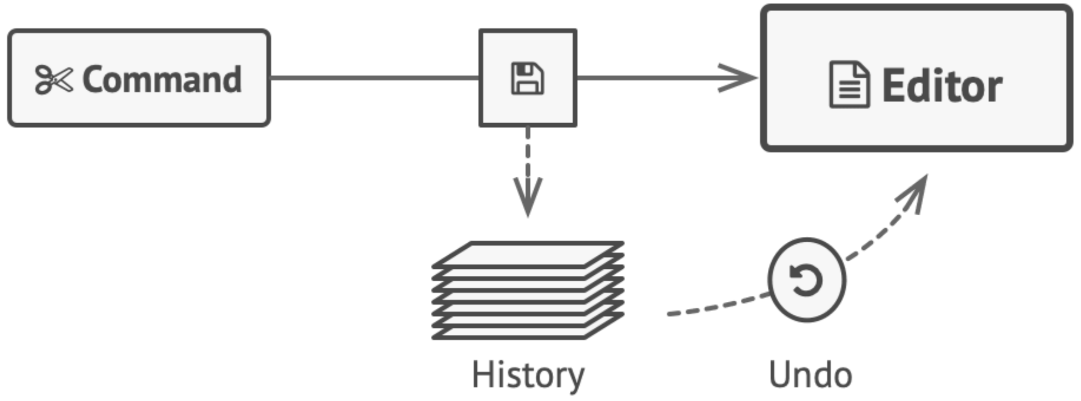
Memento
Observer

Dr. Fan Hongfei

14 November 2024

Memento: Problem

- **Example: supporting undo in a text editor app**

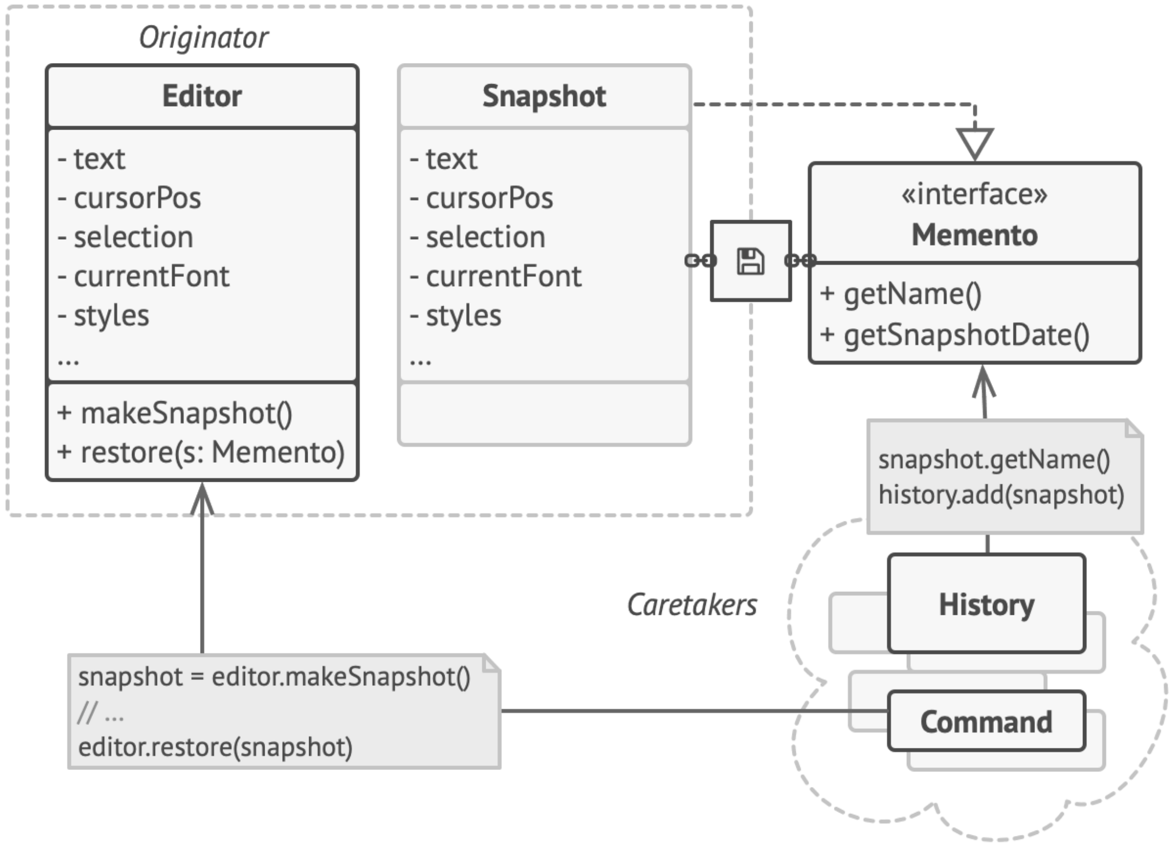


- **Problem 1:** private fields cannot be accessed
- **Problem 2:** when fields are made public, refactoring would be problematic
- **Problem 3:** the fields of the snapshot class need to be public, exposing all the editor's states

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/memento]

Memento: Solution

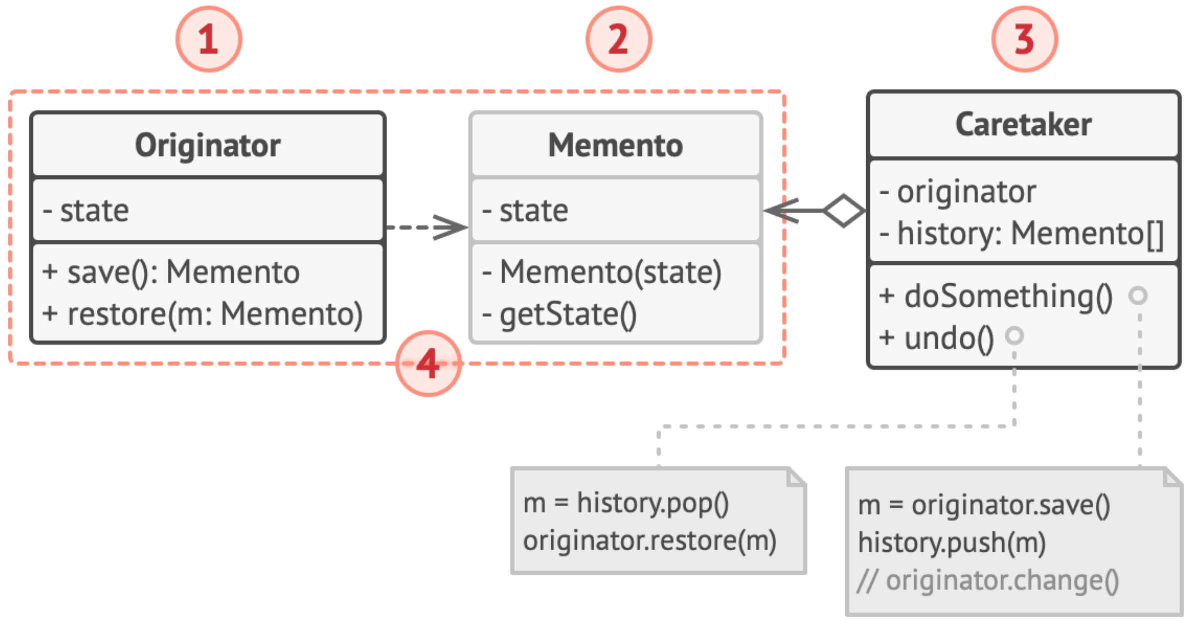
- Essence of the problems: **broken encapsulation**
- **Memento**: delegating the creation of state snapshots to the actual owner of the state – the **originator** object
- Storing the copy of state in a special object: **memento**
  - Contents are not accessible to other object, except the originator
  - Communication with mementos via a limited interface, fetching the snapshot's metadata
  - Stored inside caretakers



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/memento]

Memento: Structure

(a) Implementation based on nested classes

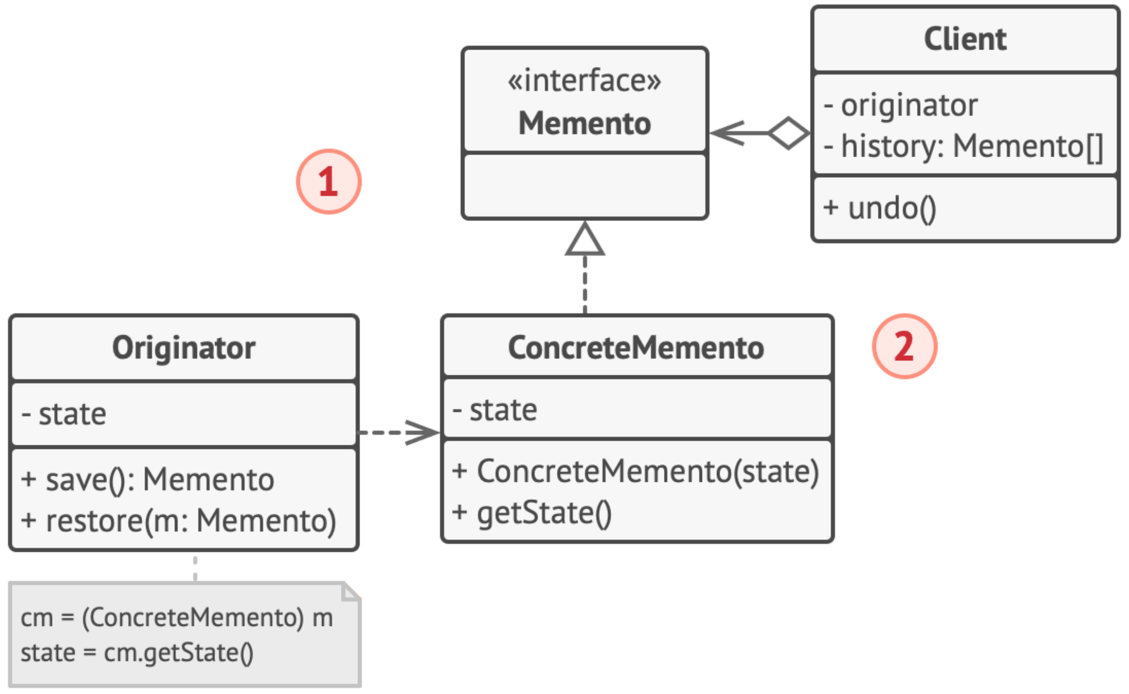


1. **Originator:** producing snapshots of its own states, and restoring its state from snapshots
2. **Memento:** a value object acting as a snapshot, commonly immutable
3. **Caretaker:** keeping track of the history by storing a stack of mementos
4. Memento is **nested** inside the originator

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/memento]

Memento: Structure

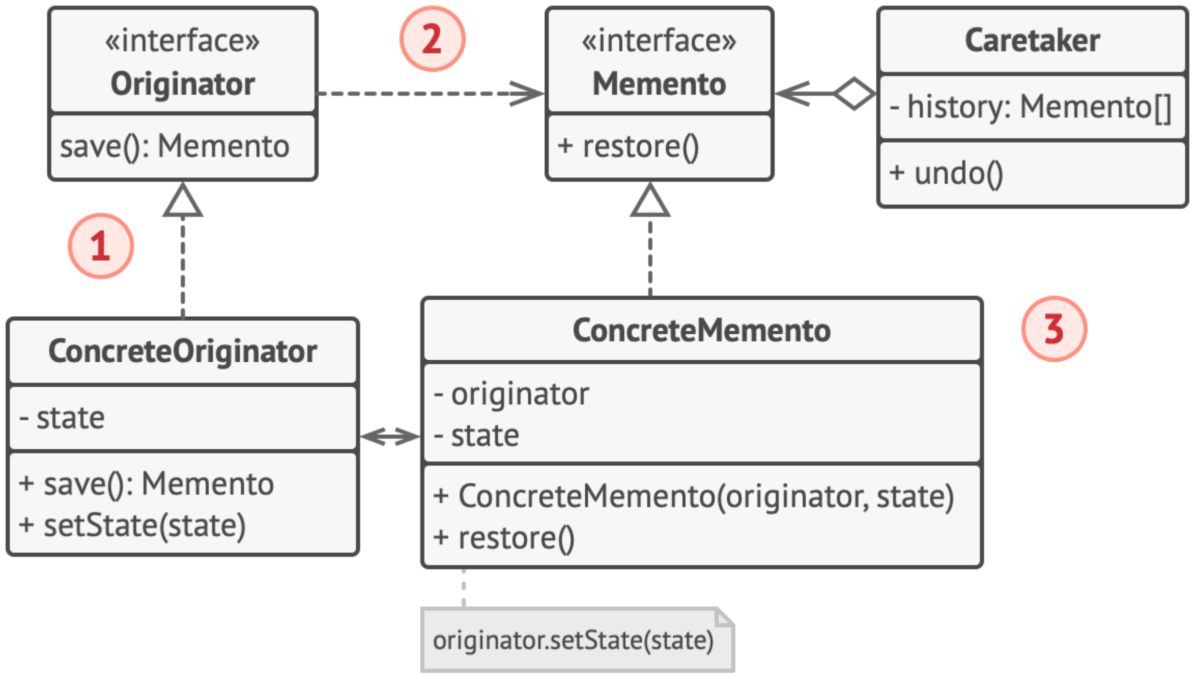
(b) Implementation based on an intermediate interface



1. In the absence of nested class, restricting access to the memento's fields: caretakers working with a memento only through an **intermediary interface**
2. Originators working with mementos directly
   – **Downside**: all members of the memento need to be public

Memento: Structure

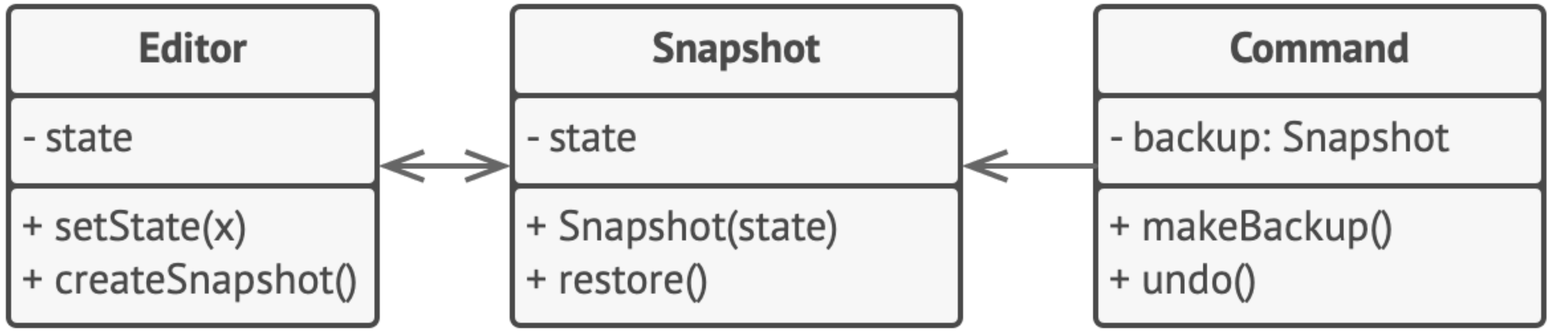
(c) Implementation with even stricter encapsulation



1. Multiple types of originators and mementos, and **none of them expose states to anyone**
2. Caretakers now explicitly restricted from changing the state in mementos
   - **More dependent** from the originator, because restoration is defined in the memento
3. Each memento linked to the originator that produced it

Memento: Example

- **Storing snapshots of the complex text editor's state, and restoring a state when needed**
- Memento + Command patterns
- Command objects: caretakers



Memento: Applicability

- To produce snapshots of the object’s state to be able to restore a previous state of the object
  - Make full copies of an object’s state, including private fields, and store them separately from the object
  - Undo, transactions
- When direct access to the object’s fields/getters/setters violates its encapsulation
  - The Memento makes the object itself responsible for creating a snapshot of its state

Memento: Implementation

1. Determine what class will play the role of the **originator**
2. Create the **memento** class, and declare a set of fields that mirror the fields of the originator
3. Make the memento class **immutable**
4. If nested class is supported, **nest** the memento inside the originator; otherwise, extract a blank **interface** from the memento and make all other objects use it to refer to the memento
5. Add a method for **producing mementos** to the originator class
   – The return type should be of the interface extracted in the previous step
6. Add a method for **restoring** the originator's state to its class, and accept a memento object as an argument
7. The **caretaker** should know when to request new mementos from the originator, how to store them and when to restore the originator with a particular memento
8. The link between caretakers and originators may be moved into the memento class
   – Make sense only if the memento class is nested, or the originator class provides sufficient setters

Memento: Pros and Cons

- **Pros**
  - Producing snapshots of the object’s state without violating its encapsulation
  - Simplifying the originator’s code by letting the caretaker maintain the history of the originator’s state
- **Cons**
  - Consumption of lots of RAM
  - Caretakers should track the originator’s lifecycle to be able to destroy obsolete mementos
  - Most dynamic programming languages (such as PHP, Python and JavaScript) cannot guarantee that the state within the memento stays untouched

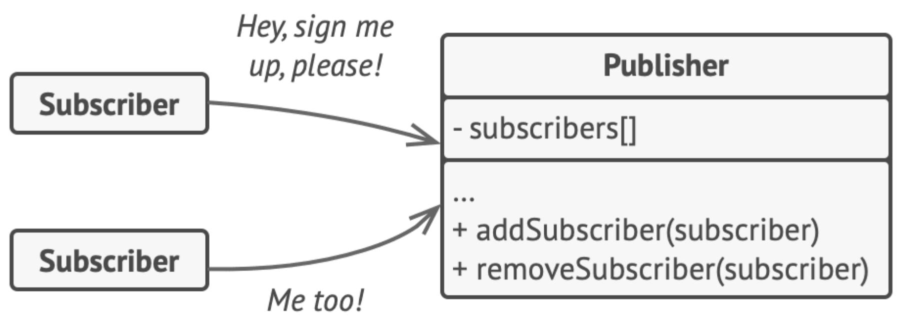
Observer: Problem

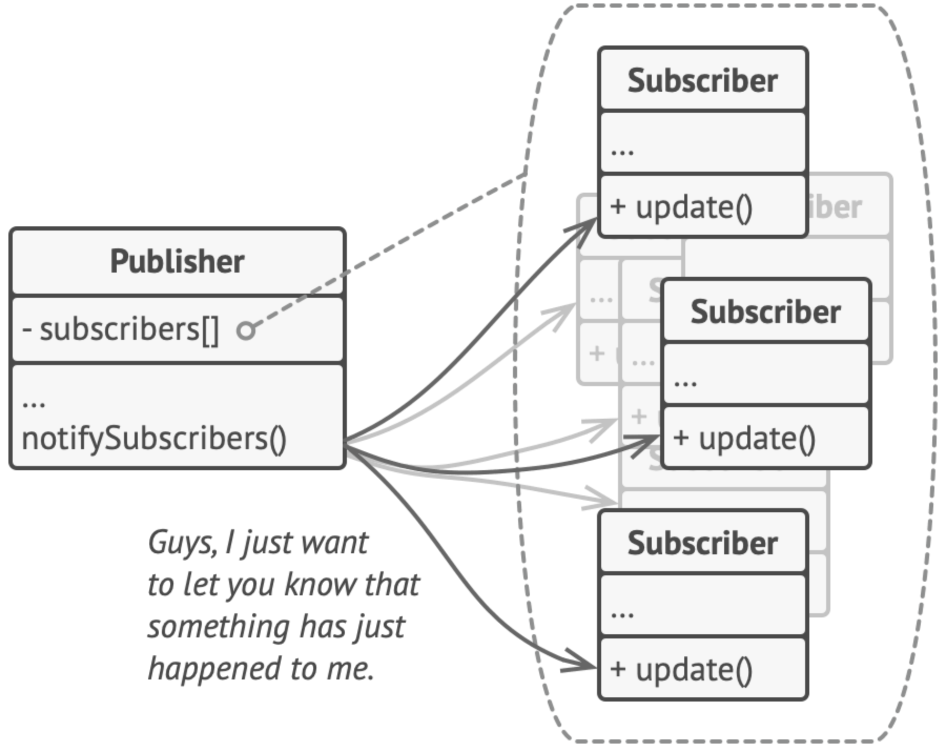
- Two types of objects: **Customer and Store**
- The customer is interested in a particular product which should become available soon

- **Problem 1:** the customer visits the store frequently and checks product availability
- **Problem 2:** the store sends tons of emails to all customers each time a new product becomes available

Observer: Solution

- **Publisher and subscriber**
- **Observer** (aka Event-Subscriber, Listener)
- Adding a **subscription mechanism** to the publisher class
  - A list of references to subscriber objects
  - Several public methods

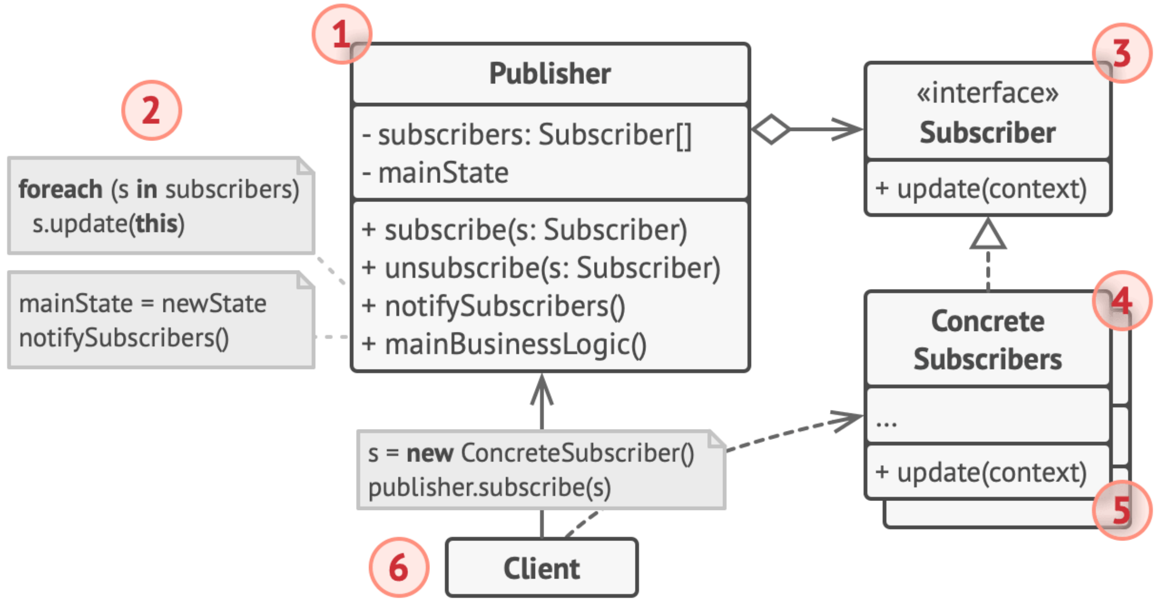




- All subscribers implement the same interface, and the publisher communicates with them **via the interface**
- **Further improvement:** make all publishers follow the same interface

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/observer]

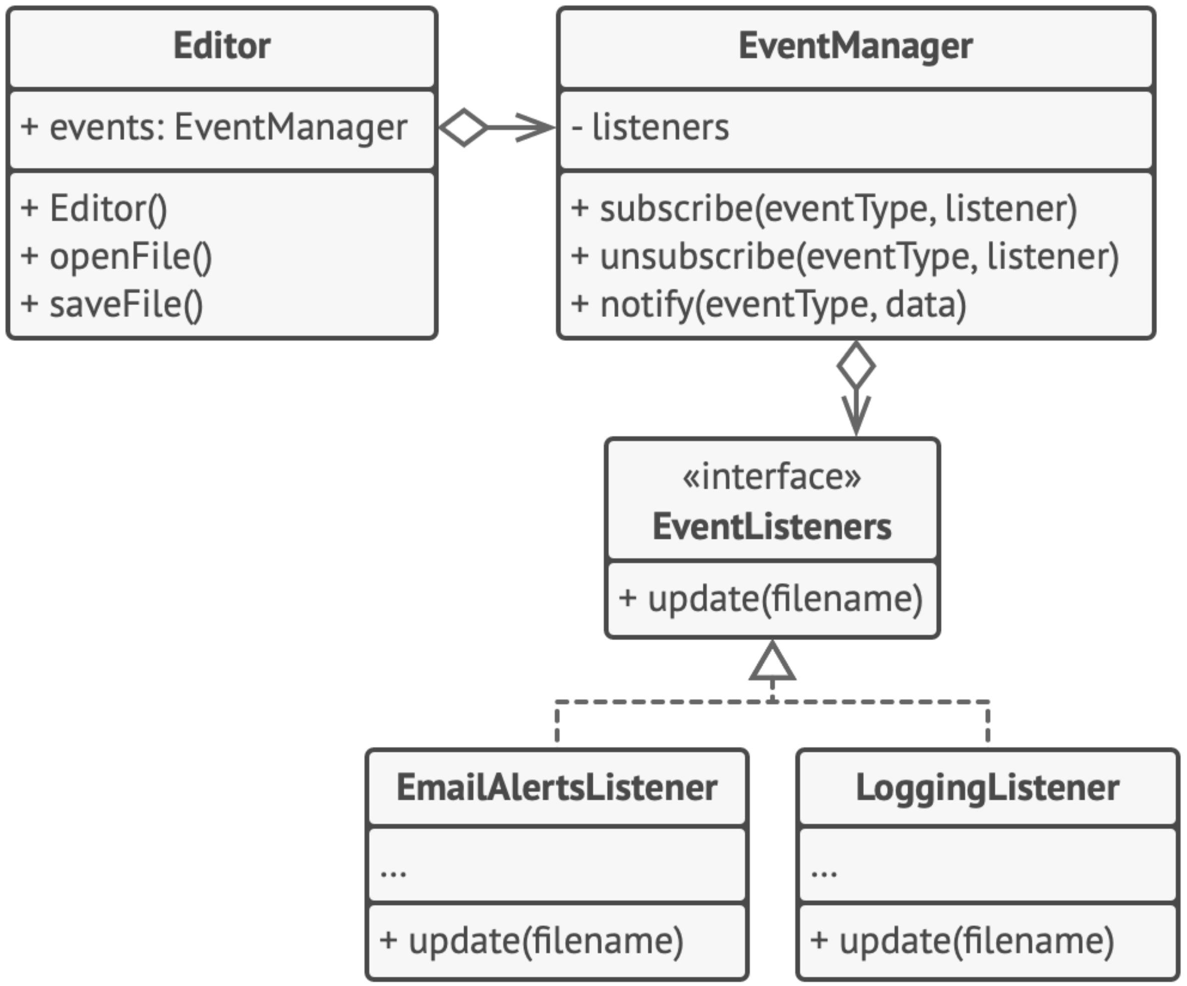
Observer: Structure



1. **Publisher:** source of events, occurring when the publisher changes its state or executes some behaviors
2. When a new event happens, the publisher **calls the notification method** on each subscriber object
3. **Subscriber:** the notification interface
4. **Concrete Subscribers:** actions in response to notifications
5. Subscribers need **contextual information** to handle the update
6. The Client creates both publisher and subscriber objects

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/observer]

Observer: Example



- Objects can start or stop listening to notifications at runtime
- The editor delegates the subscription management to a helper object
  - Could be upgraded as a centralized event dispatcher

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/observer]

Observer: Applicability

- When changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically
  - Common in graphical user interface systems
  - The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects
- When some objects must observe others, but only for a limited time or in specific cases
  - The subscription list is dynamic, so subscribers can join or leave the list whenever they need to

Observer: Implementation

1. Look over the business logic and try to break it down into **two parts**: the core functionality, as the **publisher**, and the rest as a set of **subscribers**
2. Declare the **subscriber interface**
3. Declare the **publisher interface** and describe a pair of methods for adding a subscriber object to and removing it from the list
4. Decide where to put the subscription list and the implementation of subscription methods
   - Usually, in **an abstract class** derived directly from the publisher interface
   - If applying the pattern to an existing hierarchy, consider an approach **based on composition**
5. Create **concrete publisher classes**
6. Implement the **update notification methods** in concrete subscriber classes
   - Context data can be passed as an argument
   - Another option: the subscriber can fetch any data directly from the notification
   - The less flexible option: link a publisher to the subscriber permanently
7. The client creates all necessary subscribers and register them with proper publishers

Observer: Pros and Cons

- **Pros**
  - Open/Closed Principle: introducing new subscriber classes without changing the publisher’s code
  - Establishing relations between objects at runtime
- **Cons**
  - Subscribers are notified in random order

Combinations and Comparisons

- **Command + Memento:** implementing undo
- **Memento + Iterator:** capturing the current iteration state and rolling it back if necessary

- **Chain of Responsibility, Command, Mediator, and Observer**
  - Chain of Responsibility: passes a request sequentially along a dynamic chain
  - Command: establishes unidirectional connections between senders and receivers
  - Mediator: eliminates direct connections between senders and receivers
  - Observer: lets receivers dynamically subscribe to and unsubscribe from receiving requests
- **Mediator and Observer**
  - Mediator: eliminates mutual dependencies among a set of components
  - Observer: establishes dynamic one-way connections between objects