SE 350 - Object-Oriented Software Development

Software Design Patterns: Memento

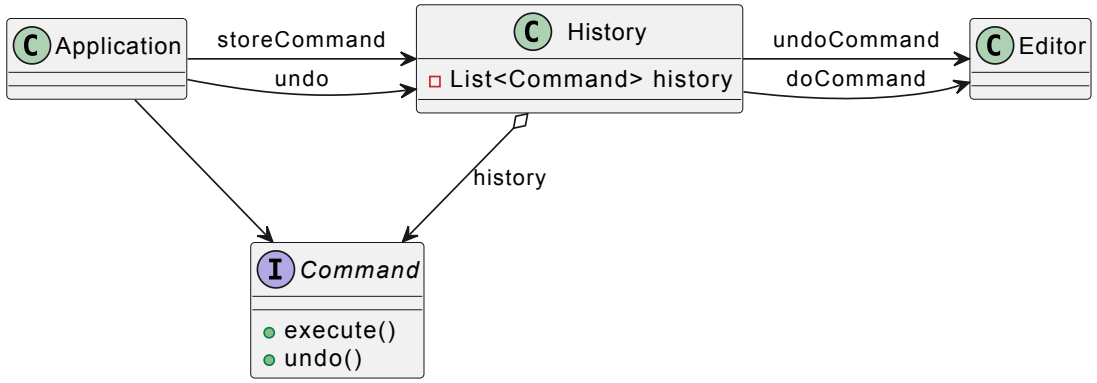
Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Memento
- Understand the consequences of using the pattern

Design Challenge

❓ Notification editor, save and undo text edits




```
public class Editor {
  private String text;
  private long cursorPos;
  // ...
}
```

⚠️ Reversible `Command`

⚠️ What if command is not easily reversible, such as "Replace All"?

⚠️ How to make a copy of an object's private state?

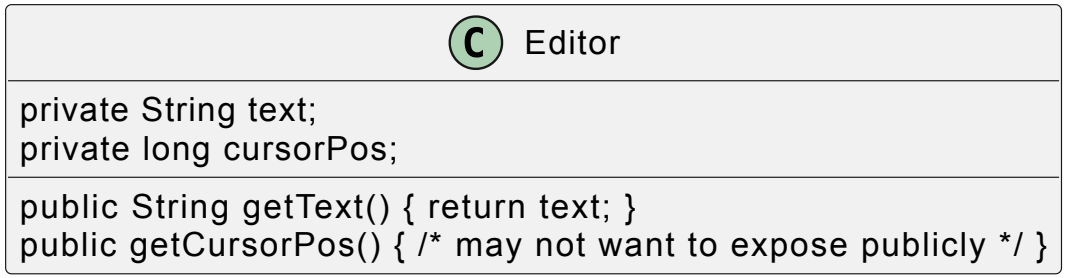
Memento Introduction

Memento takes a snapshot of an object's internal state

- Behavioral design pattern
- Store internal state without violating encapsulation

Problem illustration:

❓ How to store the internal state without breaking encapsulation?



Memento Overview

Problem

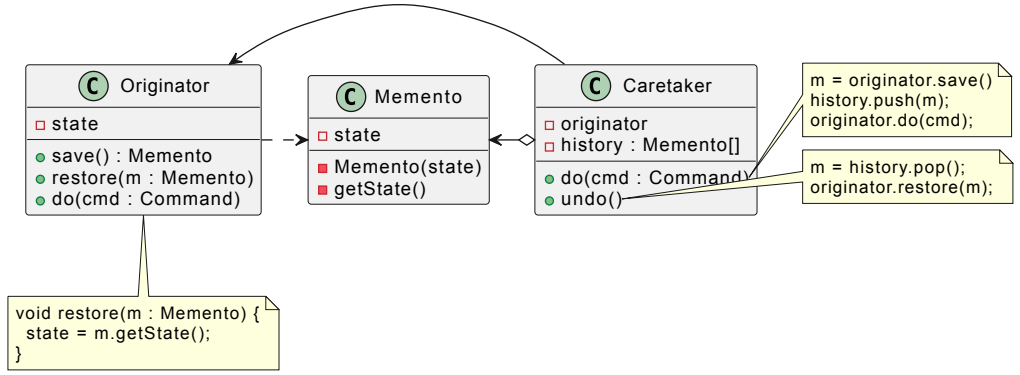
Want to store object's internal state without violating encapsulation

Intent

- Support undo operations
- Use when direct public getters/setters are undesired

Structure

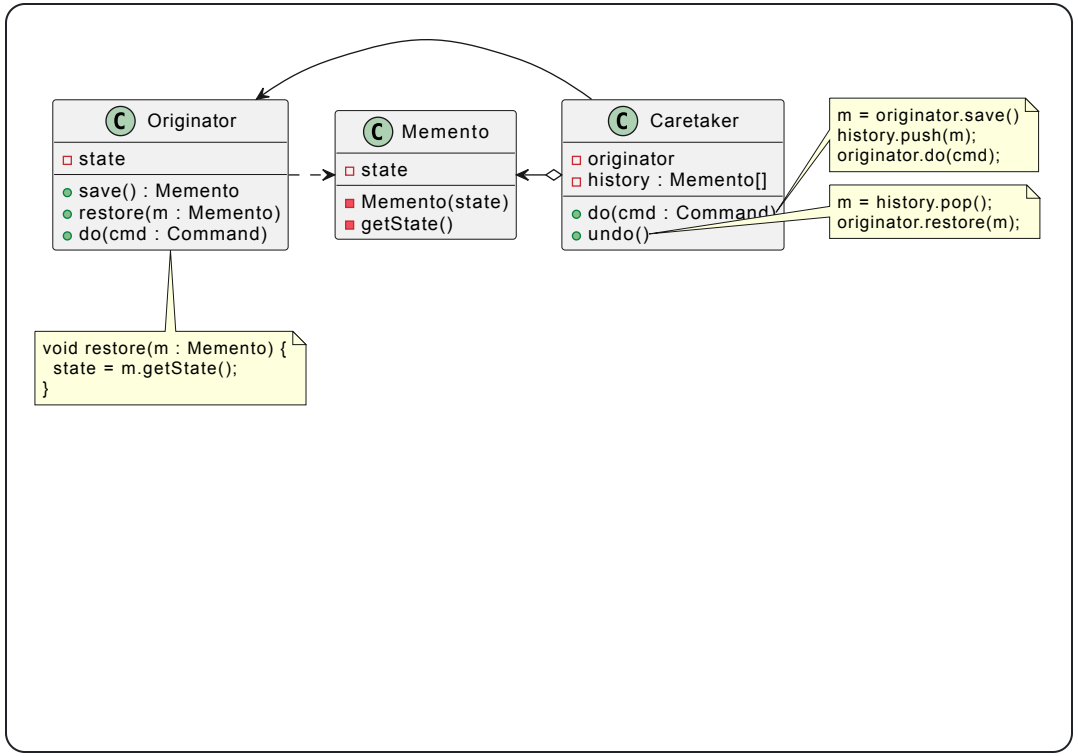
Based on nested classes



Memento Participants

1. The `Originator` class produces a snaptshot of its own state, and restores its state from snapshots

2. The `Memento` is a value object that has fields to hold the `Originator` state; a `Mememto` should be immutable and only be initialized through the constructor.



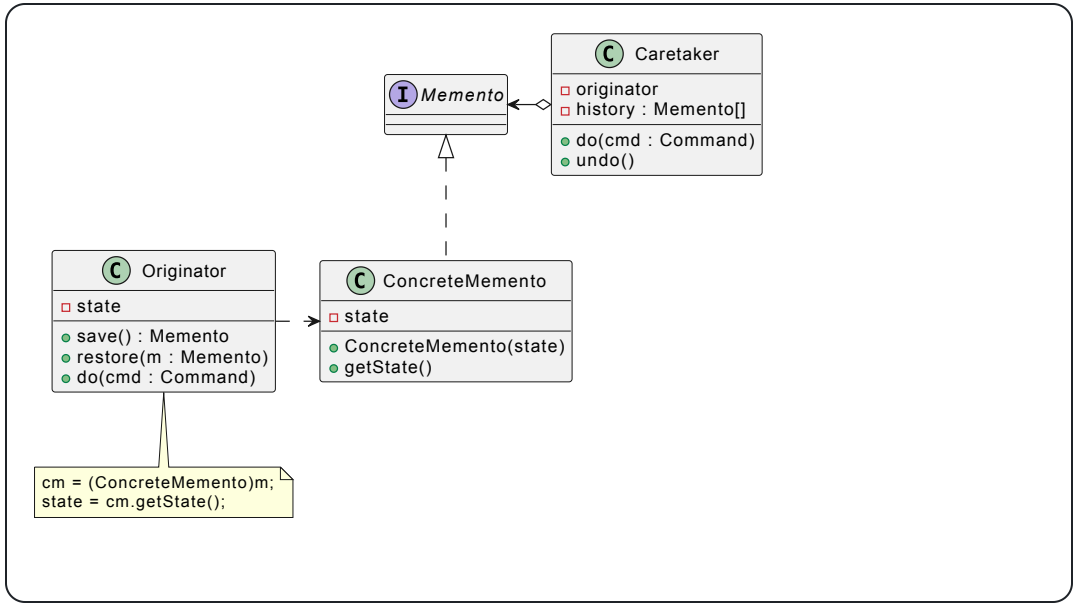
3. The `Caretaker` knows **when** to capture and restore the state of the `Originator`; the `Caretaker` can keep track of the history and pass previous snapshots to the `Originator` to restore its state.

4. In this implementation, the `Memento` is a nested class and has a `private` constructor, so that only the originator can create it; even though the `Memento` has access to the `private` fields of the `Originator`, the `Caretaker` has very limited access.

Memento Variation: Intermediate Interface

For programming languages without support for nested classes

1. The `Originator` class produces a snapshot of its own state by instantiating a `ConcreteMemento` class; all others work with the memento through the `Memento` interface

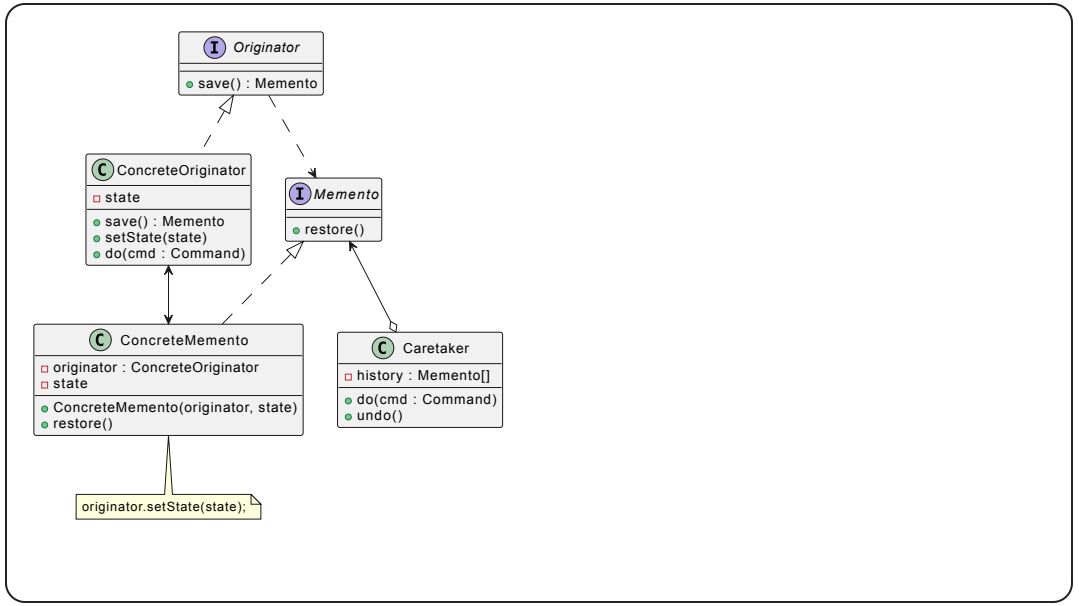


2. The `Memento` has all `public` getters and setters, so that the `Originator` can work with it

Memento Variation: Strict Encapsulation

For programming languages without support for nested classes

1. Multiple types of `Originator` and `Memento`; each originator works with a corresponding memento; neither one has `public` getters to expose their state to anyone



2. The `Caretaker` is now explicitly restricted from accessing state in the `Memento` and entirely decoupled from the `Originator`

3. Each memento is linked to the originator that produced it; the originator passes its state to the memento through its constructor; the memento knows how to set the state of the originator.

Memento Applicability

- **Want to produce snapshots and restore state**:
  Memento pattern can make full copies of an object's internal state, including `private` fields; snapshot can be stored separately from originator
- **Use when direct access to getters/setters violates encapsulation**:
  Memento pattern makes the originator responsible for creating snapshots; no other object needs access to the state

Memento Implementation

- Create the `Memento` class, mirror the originator's fields in the memento
- Make the `Memento` class immutable
- Use nested classes if supported by the language
- Add a method to the `Originator` for producing a `Memento`
- Add a method to the `Originator` for restoring state
- Implement a `Caretaker` that keeps track of mementos

```
public interface NumberCommand {
  int execute();
}
public class RandomNumberCommand {
  private Random r = new Random();
  public int execute() { return r.nextInt(); }
}

public class NumberStore {
  private int number;

  public static class Memento {
    private int n;
    private Memento(int n) { this.n = n; }
    private int getNumber() { return n; }
  }

  public Memento save() { return new Memento(number); }
  public void restore(Memento m) { number = m.getNumber(); }

  public void do(NumberCommand c) { number = c.execute(); }
}

public class History {
  private List<NumberGenerator.Memento> history = List.of();
  private NumberStore store;

  public History(NumberStore store) { this.store = store; }

  public void do(NumberCommand c) {
    history.add(store.save());
    store.do(c);
  }

  public void undo() { store.restore(history.removeLast()); }
}

public class Demo {
  public static void main(String[] args) {
    NumberStore s = new NumberStore();
    History h = new History(s)
    h.do(new RandomNumberCommand());
    h.do(new RandomNumberCommand());
    h.undo();
  }
}
```

Memento Discussion

Benefits

- Produce snapshots without violating encapsulation
- Simplify originator code by letting a caretaker maintain history

Drawbacks

- History may consume considerable amounts of memory
- Caretaker needs to track originator's lifecyle to destroy obsolete mementos
- Needs statically typed languages, most dynamically typed languages cannot guarantee immutable mementos

Relations with Other Patterns

- Combine with `Command` to implement undo