Memento

A behavioral pattern

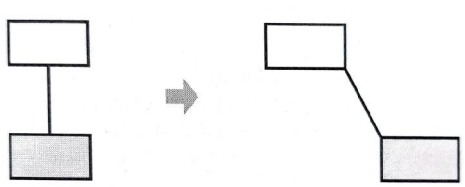
Learning goals

1. Learn the idea, structure, and Java implementation of the Memento design pattern.
2. Learn to apply the Memento DP in your own programming.

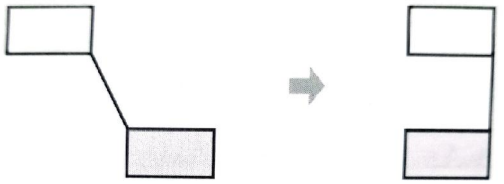
Idea of Memento

- The purpose of the Memento DP is to provide an easy way to implement the **undo operation**.
  - i.e. reversal of the object's state to an earlier state in history.
- Each state of an object is saved as a memento object.
  - A memento is like a snapshot of an earlier state.
- The sequence of snapshots makes the **history** of the object.
- The Memento DP offers a well-encapsulated way to manage the history, create saved states, and retrieve them.

Example



Original operation

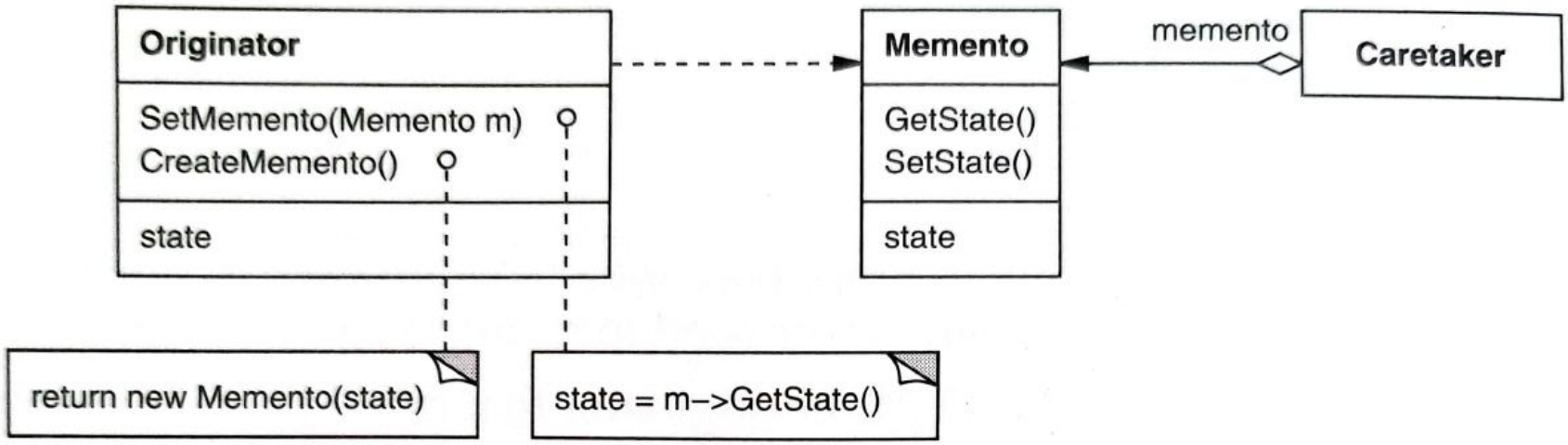


Anomaly after undo

- A vector graphics image has rectangles and a connecting line. The line updates automatically as one of the rectangles is moved.
- The implementation of the undo operation may not be straighforward.
  - For individual operations, the reverse operations (such as moving the rectangle back) may be cumbersome to create and hard to keep track of.
  - Anomalies may occur (such as the connecting line being drawn differently).

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), pp. 283-284

General structure



Note: setMemento() means setting the object's state based on the given memento

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 285

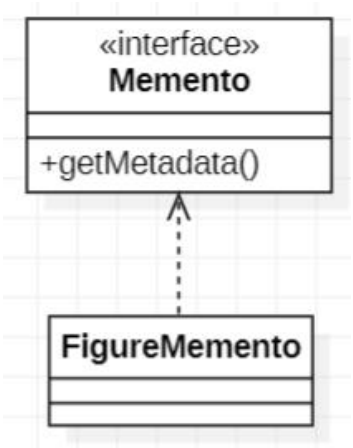
Roles

- **Originator**: Creates a memento based on its own data. Updates its data based on the retrieved memento.
  - The originator has full access to the retrieved memento.
- **Memento:** Stores the state of the Originator object.
- **Caretaker**: Keeps track of the available mementos. It can ask the originator to create a memento, and it can retrieve a memento from the history.
  - The retrieved memento will always be passed to the originator without further tampering.

Addon: Metadata interface

- By default, the Caretaker just handles the sequence of mementos (undo list).
- It can iterate the list, and retrieve a memento, but it should not have direct access to the content of the memento.
- The Caretaker may still need some information about the mementos for management. For example:
  - getTimestamp()
  - getAuthor()
  - Additional methods that the application may need.
- To achieve this, a memento interface is declared.

Addon: Metadata interface



- A metadata interface makes the necessary metadata information accessible to the Caretaker.
- The caretaker can store the mementos into the history as references to the Memento interface.
- This also encapsulates the memento, as the caretaker has no more access to the other methods of the memento (e.g. getters and setters).
  - This alone may be a valid reason for declaring the interface, even if it was empty.
- The Originator, after receiving the memento, will do the typecast to the memento's concrete class and have access to its state.

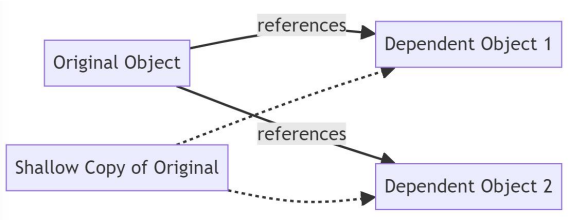
Managing the history

- ArrayList is usually a good choice for storing the mementos, as it is indexed.
  - Alternatively, a linked list or a stack can be used, if the purpose is just to implement undo of the last operation efficiently.
- If the stored objects are large, the memory consumption of the mementos can become huge.
- This can be controlled by adding the limit for the list length.
- Managing the list length is the Caretaker's responsibility.
  - Add a check before adding a memento to the list. If the list size turns out to be higher than the limit, remove the first list item.
- If the history size is limited, and no dynamicity is needed, an array can also be used for holding the references to the mementos.

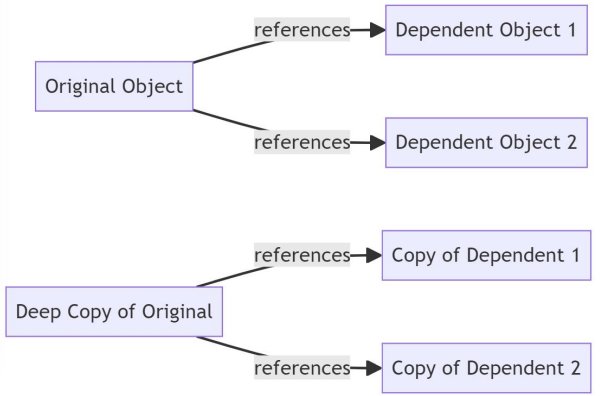
Deep and shallow copying

- A memento may have references to different objects.
- In shallow copying, only references to those object are copied.
  - Use it when dependent objects are known not to change, or when the undo is by design supposed not to affect dependent objects.
- In deep copying, the states of the dependent objects are copied into new objects.
  - Use it when the full state of the application needs to be undone.

Shallow-copied memento



Deep-copied memento



Full vs. incremental changes

- Usually, the memento stores the objects full state.
- For a large object, even a tiniest change causes a new, full memento to be created.
  - This can significantly drain resources.
- Alternatively, an incremental approach can be applied:
  - Only the changed data is stored in the memento, not the full data
- This requires changing the structure of the memento class.
  - The changed data can be stored, for example, as key-value pairs.
  - This complicates the implementation of both the save and the retrieve operations.