Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

 abdulrahman@nu.edu.pk

 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786

 http://alphapeeler.sf.net/me

 alphapeeler#9321

 *reddit.com/user/alphapeeler*

 www.flickr.com/alphapeeler

 http://alphapeeler.tumblr.com

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 pinterest.com/alphapeeler

 www.youtube.com/user/AlphaPeeler

Memento Design Pattern

What is Memento pattern?

Memento is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

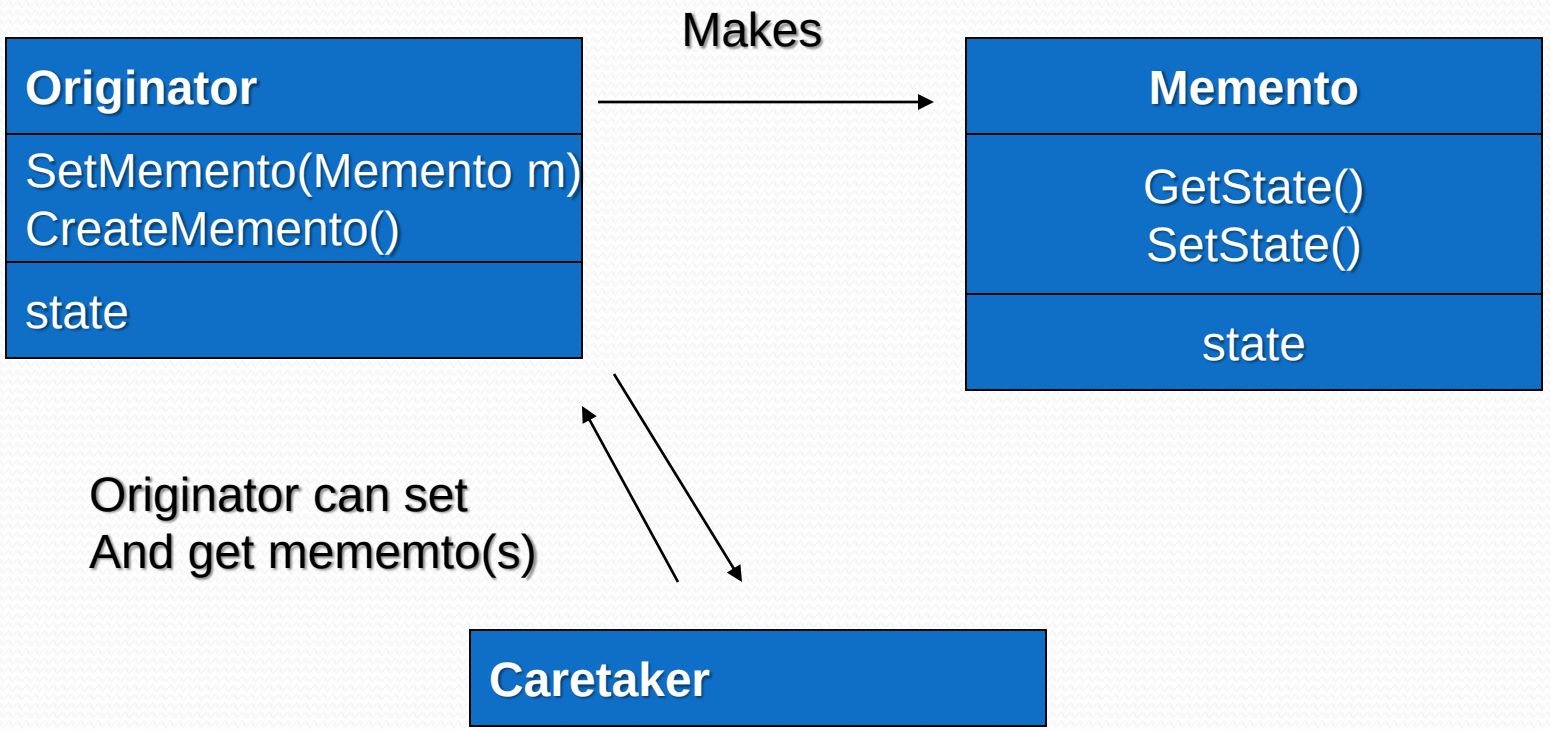
Your Situation

- You want to provide a way to save the state of an object, without violating encapsulation
- Delete from cart and Undo
- Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

Who's involved?

- **Originator:** The object whose state we are going to preserve. Can use memento to restore his state.
- **Memento:** The object who saves the state of the Originator. Can hold a bit or all of originator's data. There is stuff only caretaker can see (the narrow interface), and stuff that originator can see (wide interface).
- **Caretaker:** The object who holds the Memento and possibly returns it to the originator. Doesn't look into the memento.

How do these guys work together?

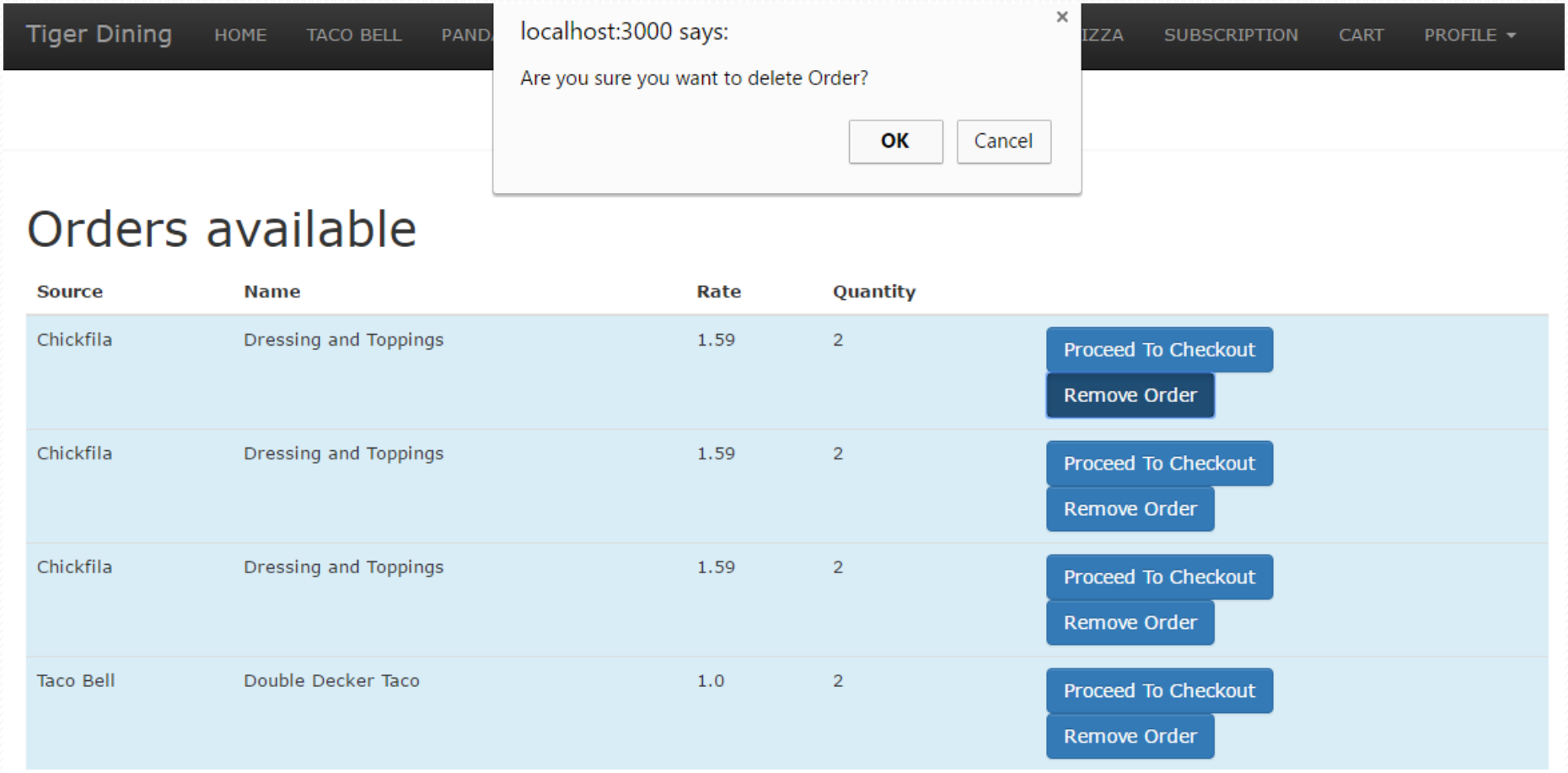


When do I use this again?

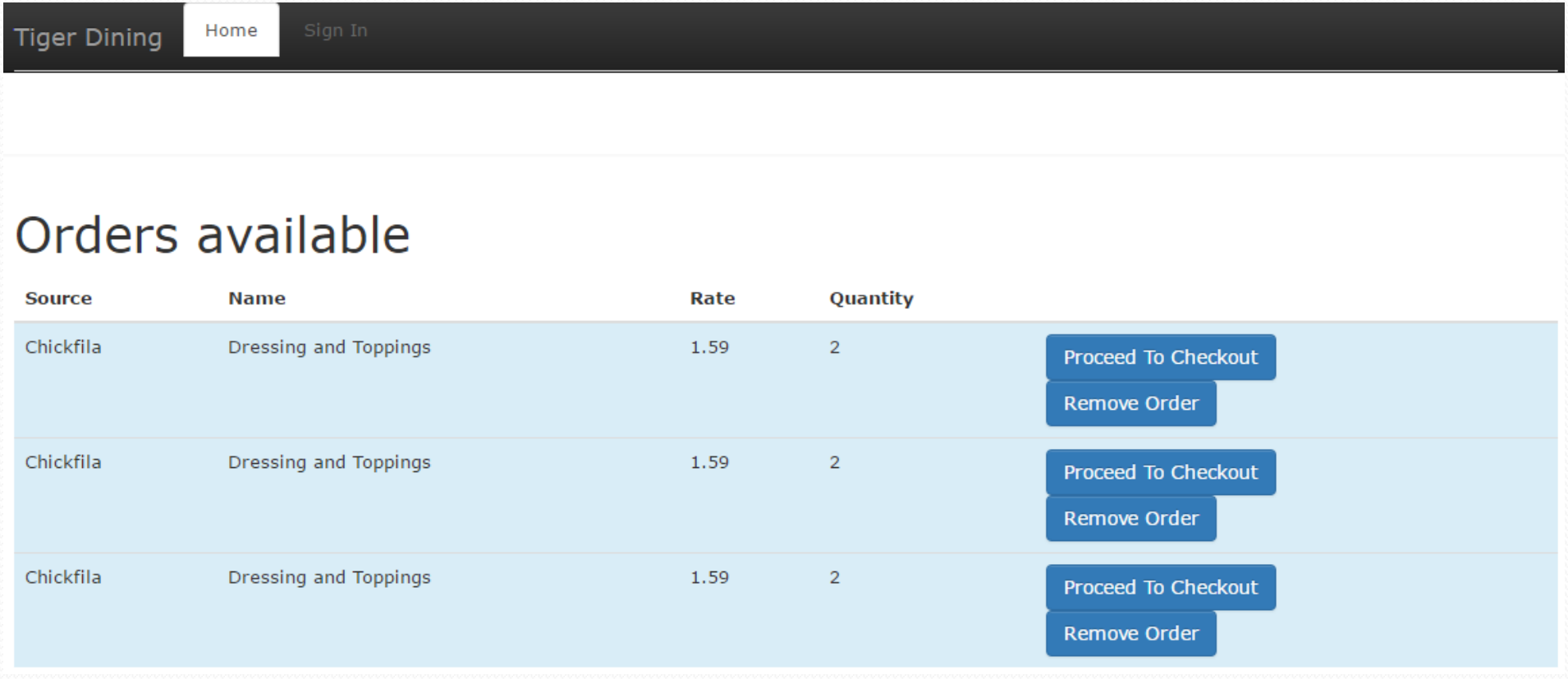
you want to save a snapshot of an object's state - so it can be restored later

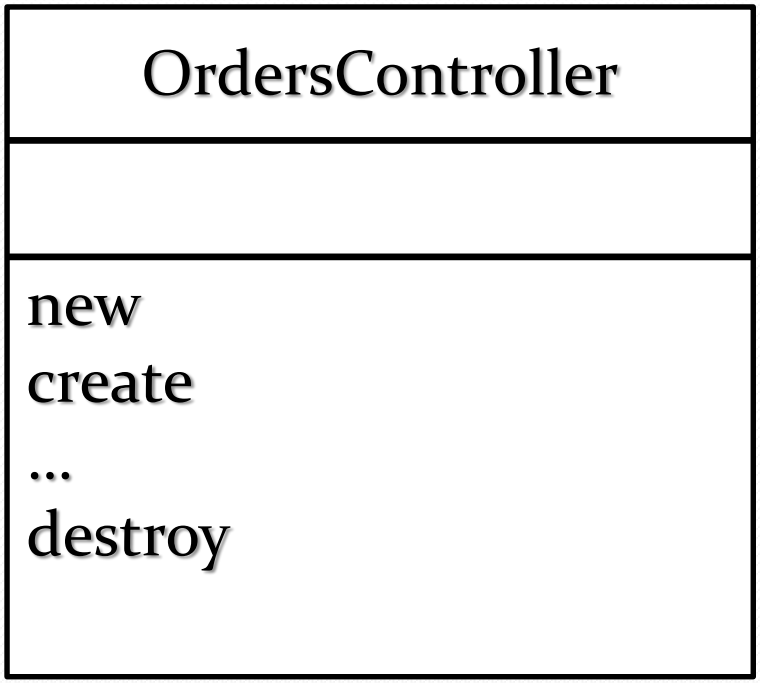
AND

directly exposing those values would expose implementation details - violating encapsulation.


Tiger Dining
HOME
TACO BELL
PAND
IZZA
SUBSCRIPTION
CART
PROFILE
localhost:3000 says:
Are you sure you want to delete Order?
OK
Cancel
Orders available
Source
Name
Rate
Quantity
Chickfila
Dressing and Toppings
1.59
2
Proceed To Checkout
Remove Order
Chickfila
Dressing and Toppings
1.59
2
Proceed To Checkout
Remove Order
Chickfila
Dressing and Toppings
1.59
2
Proceed To Checkout
Remove Order
Taco Bell
Double Decker Taco
1.0
2
Proceed To Checkout
Remove Order

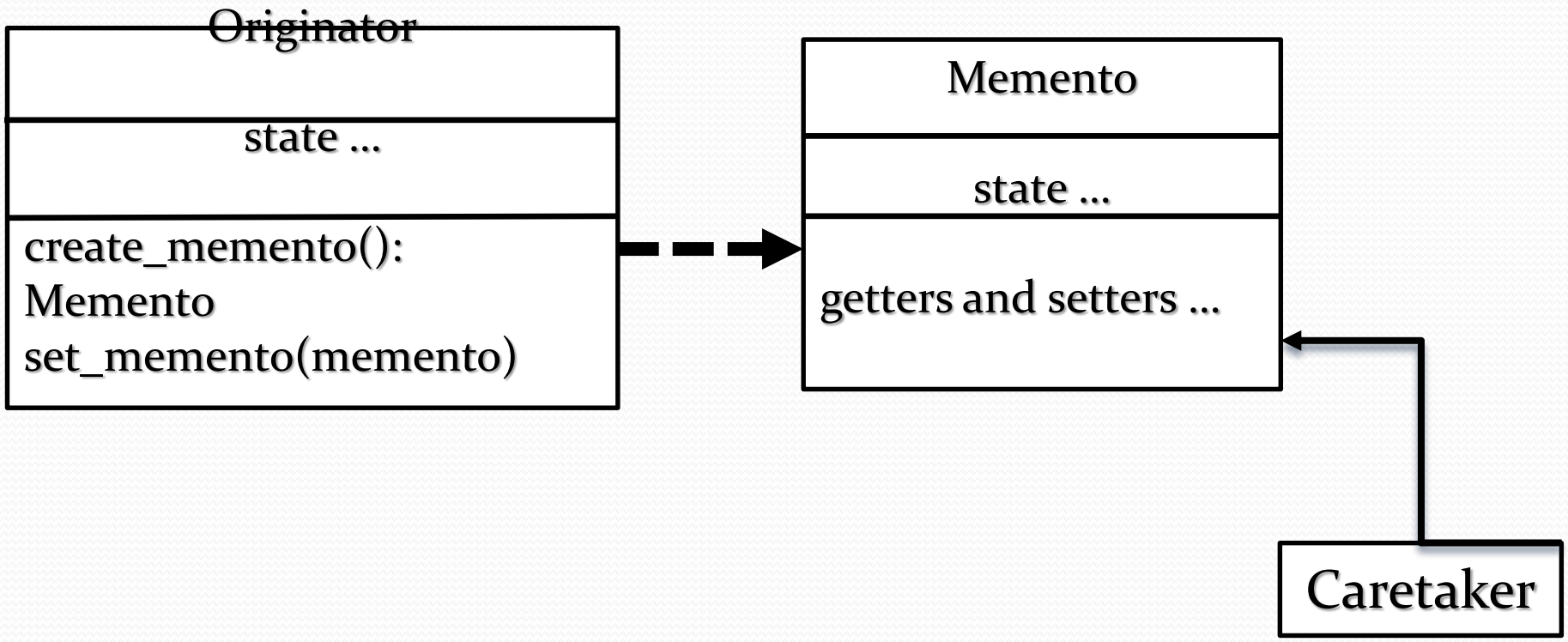
Cannot retrieve back

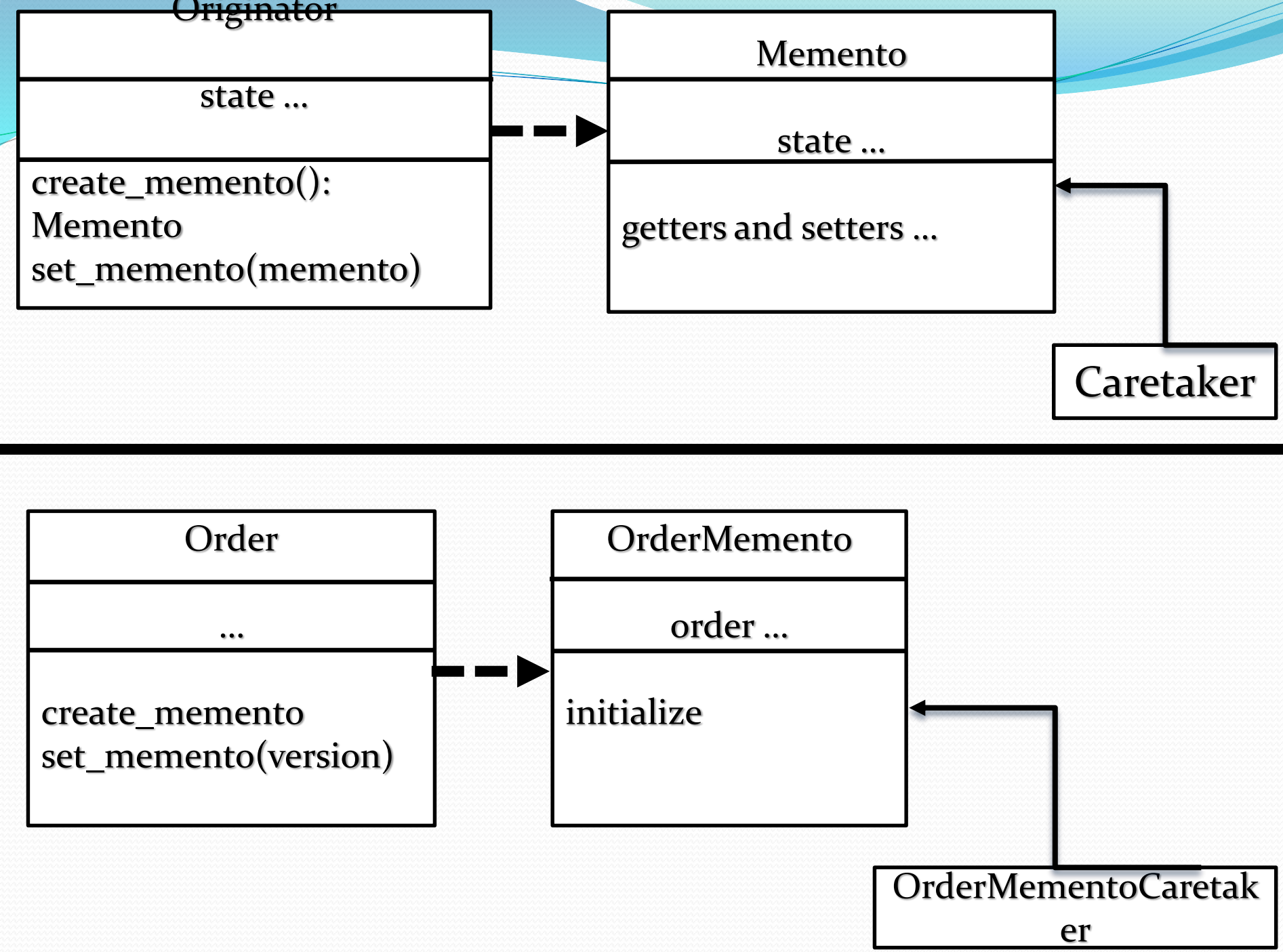



OrdersController
new
create
...
destroy

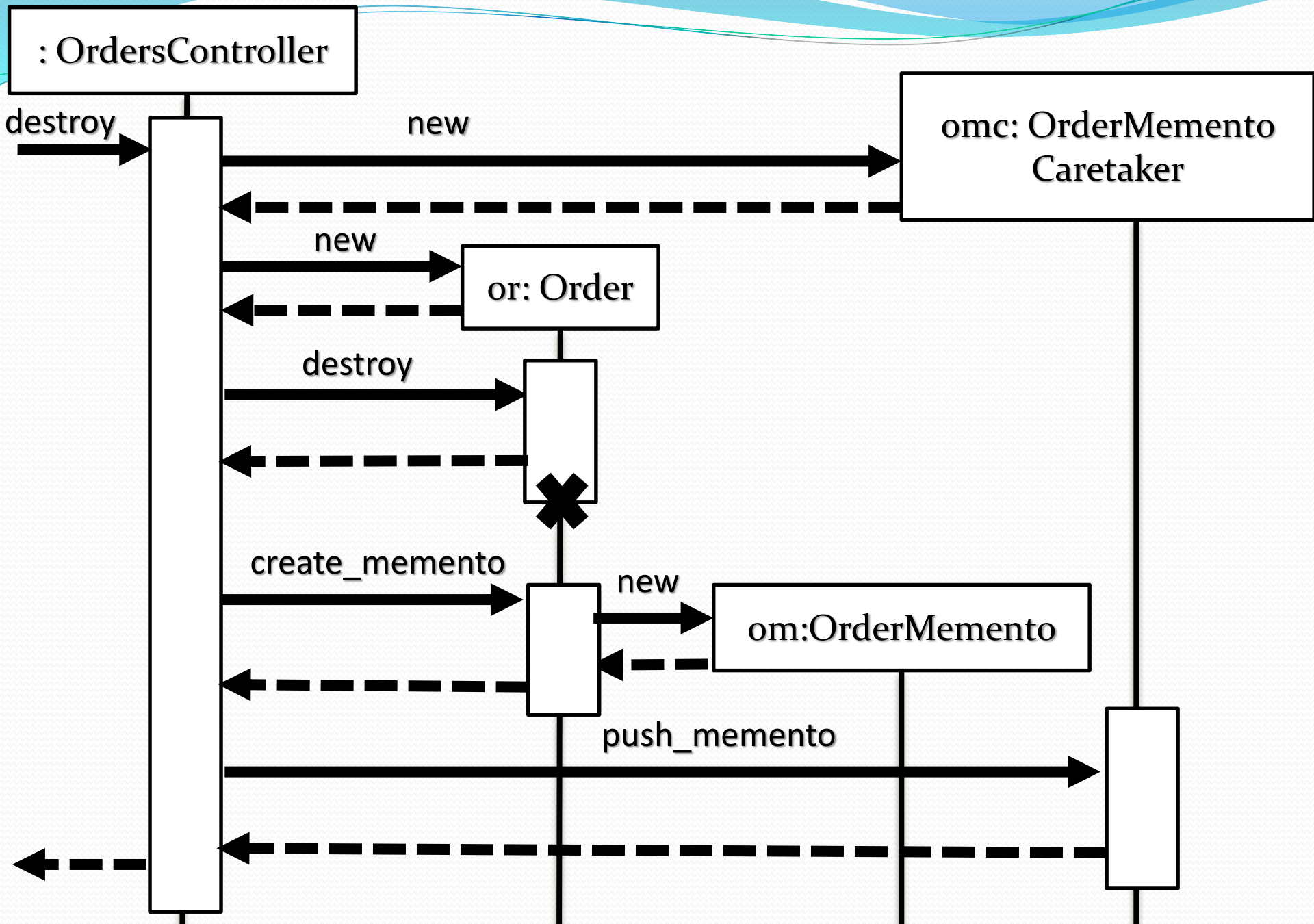
Solution: Memento

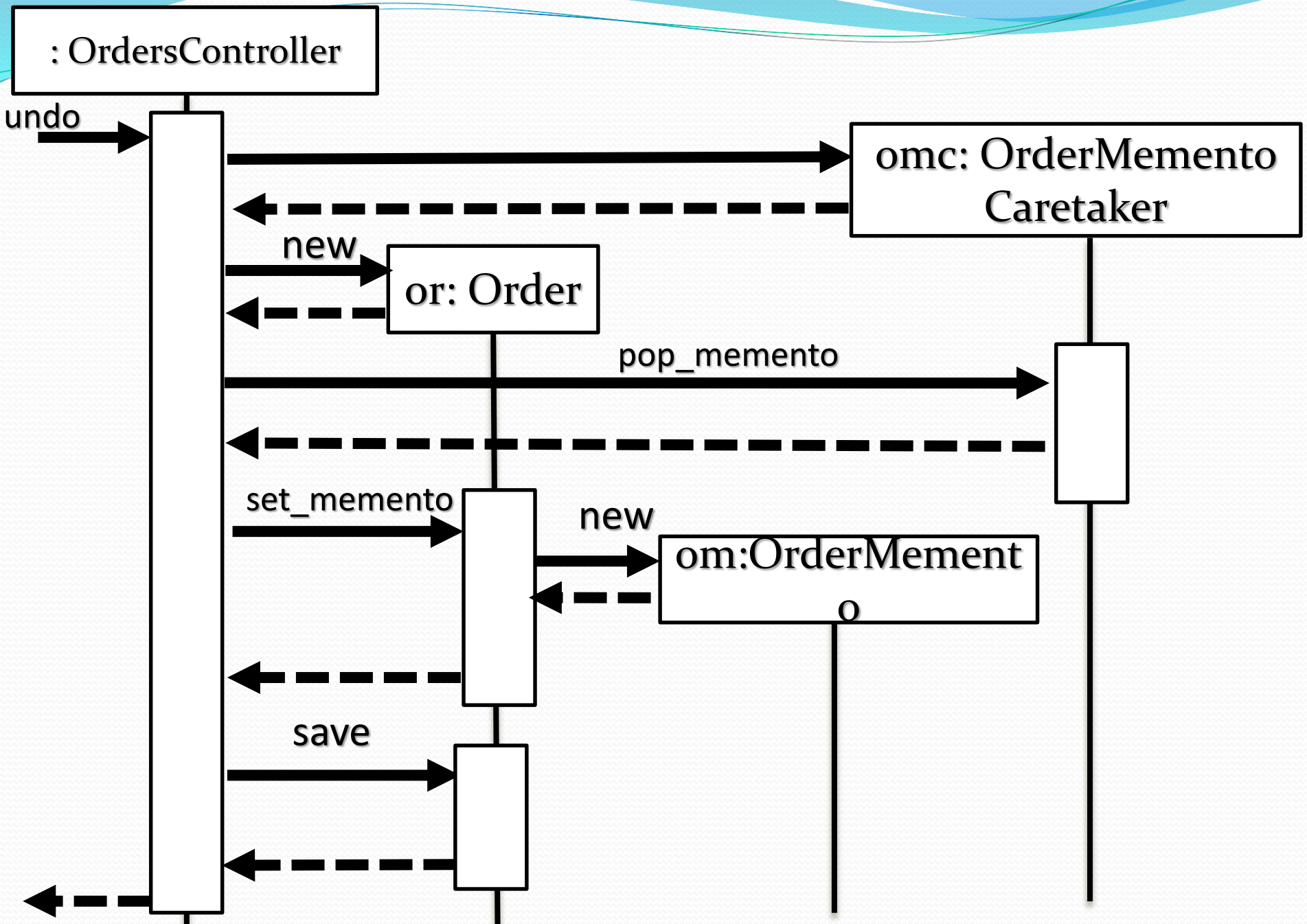
- Needs to store the state of the order




Originator
state ...
create_memento():
Memento
set_memento(memento)
Memento
state ...
getters and setters ...
Caretaker
Order
...
create_memento
set_memento(version)
OrderMemento
order ...
initialize
OrderMementoCaretaker

Control Flow: destroy




: OrdersController
undo
omc: OrderMemento Caretaker
new
or: Order
pop_memento
set_memento
new
om:OrderMemento
save

Goal: Implement memento pattern

- Order
  - “create_memento” and “set_memento”
- OrderMemento
  - initialize
- OrderMementoCaretaker
  - “push_memento” and “pop_memento”

Consequences

- Preserving encapsulation boundaries. (good)
- Simplifies originator (good)
- Could be expensive (bad)
- Defining narrow and wide interfaces (hard)

Problem: Narrow and wide interfaces

- Defining narrow and wide interfaces refers to the granularity of the interfaces provided by the Memento objects for interacting with the Originator object.

Problem: Narrow and wide interfaces

- **Narrow Interface**:
  - A narrow interface provides limited access to the internal state of the Originator object.
  - Advantages:
    - **Encapsulation:** It encapsulates the state, preventing direct access from external objects, thereby enhancing the encapsulation of the Originator.
    - **Security:** It restricts access to sensitive data, improving security.
  - Disadvantages:
    - **Limited functionality:** Since it only exposes essential methods for restoring state, it may not provide enough flexibility for certain use cases where more comprehensive access is required.

Problem: Narrow and wide interfaces

- **Wide Interface**:
  - A wide interface provides more extensive access to the internal state of the Originator object. It may expose methods for inspecting or modifying the state.
  - Advantages:
    - **Flexibility**: It allows for more versatile interactions with the Originator, enabling a wider range of operations.
  - Disadvantages:
    - **Reduced encapsulation:** Exposing more of the internal state can compromise encapsulation, potentially leading to unintended modifications or dependencies on internal implementation details.
    - **Security risks:** With greater access, there's an increased risk of unauthorized manipulation of the state, potentially compromising system integrity.

Disadvantages of Defining both Narrow & Wide Interfaces:

- **Complexity**: Maintaining two interfaces (narrow and wide) can increase the complexity of the Memento pattern implementation.
- **Maintenance**: Changes in requirements or the internal structure of the Originator may require adjustments to both interfaces, increasing maintenance overhead.
- **Confusion**: Developers may face challenges in determining which interface to use in different scenarios, leading to confusion and potential misuse.
- In summary, while a narrow interface enhances encapsulation and security but may lack flexibility, a wide interface provides more flexibility but may compromise encapsulation and introduce security risks. **The choice between narrow and wide interfaces depends on the specific requirements and trade-offs of the application.**

Related Classes

- Command: can use mementos to maintain state for undoable commands
- Iterator : Mementos can be used for iteration

Example 1

```
import java.util.List;
import java.util.ArrayList;
class Life {
    private String time;
    public void set(String time)  {
        System.out.println("Setting time to " + time);
        this.time = time;
    }
    public Memento saveToMemento()  {
        System.out.println("Saving time to Memento");
        return new Memento(time);
    }
    public void restoreFromMemento(Memento memento) {
        time = memento.getSavedTime();
        System.out.println("Time restored from Memento: " + time);
    }
    public static class Memento  {
        private final String time;
        public Memento(String timeToSave) {
            time = timeToSave;
        }
        public String getSavedTime() {
            return time;
        }
    }
}
```

Example 1

```
class Design {
    public static void main(String[] args)  {
        List<Life.Memento> savedTimes = new ArrayList<Life.Memento>();
        Life life = new Life();
        //time travel and record the eras
        life.set("1000 B.C.");
        savedTimes.add(life.saveToMemento());
        life.set("1000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("2000 A.D.");
        savedTimes.add(life.saveToMemento());
        life.set("4000 A.D.");
        life.restoreFromMemento(savedTimes.get(0));
    }
}
```

Output:

```
Setting time to 1000 B.C.
Saving time to Memento
Setting time to 1000 A.D.
Saving time to Memento
Setting time to 2000 A.D.
Saving time to Memento
Setting time to 4000 A.D.
Time restored from Memento:
1000 B.C.
```

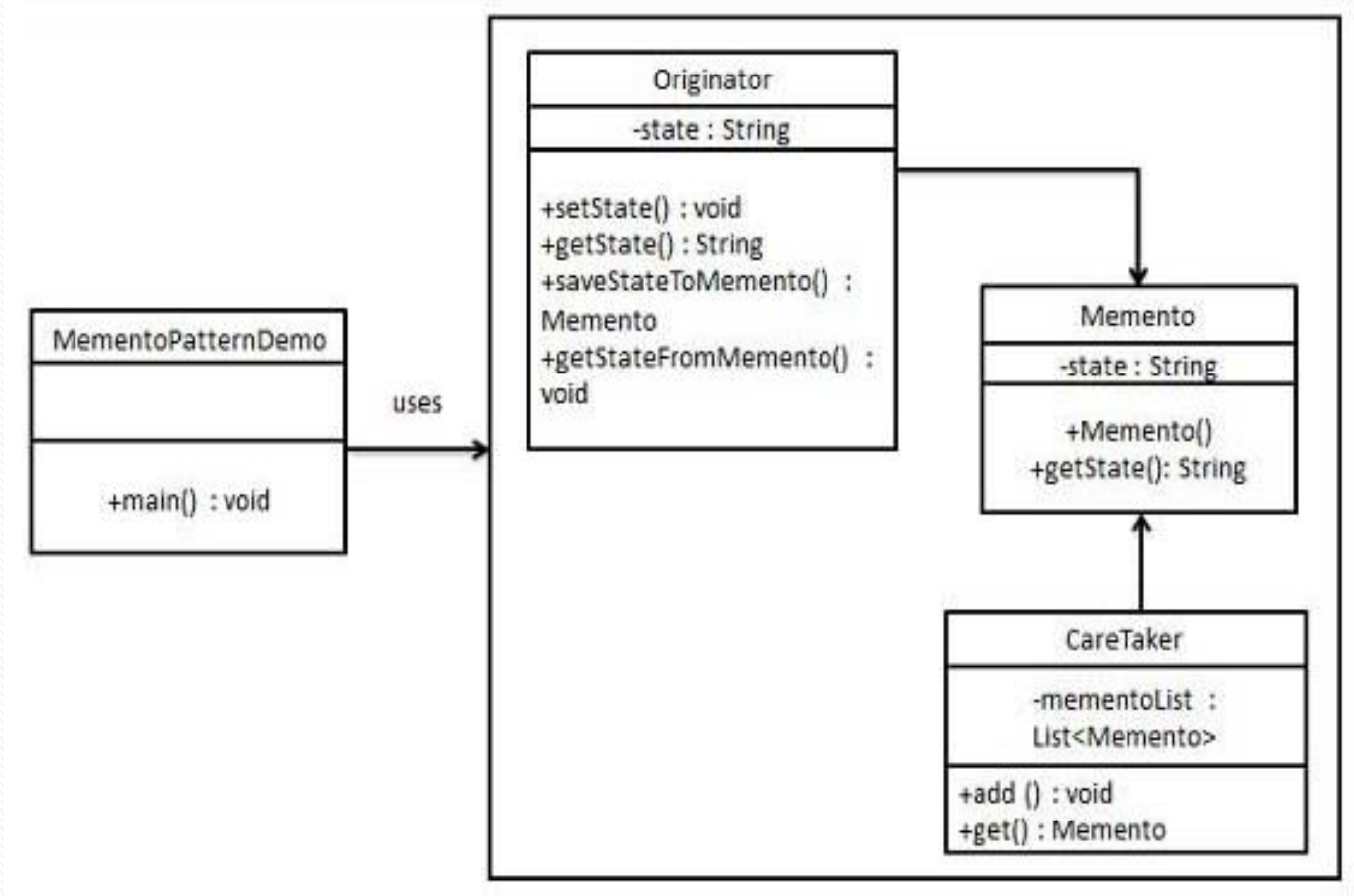
Memento

- **Advantage**
- We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.
- **Disadvantage**
- If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

Example 2:

- Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.
- **Implementation**
- Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. We have created classes *Memento*, *Originator* and *CareTaker*.
- *MementoPatternDemo*, our demo class, will use *CareTaker* and *Originator* objects to show restoration of object states.

Example 2:


MementoPatternDemo
+main() : void
uses
Originator
-state : String
+setState() : void
+getState() : String
+saveStateToMemento() : Memento
+getStateFromMemento() : void
Memento
-state : String
+Memento()
+getState(): String
CareTaker
-mementoList : List<Memento>
+add () : void
+get() : Memento

Example 2:

- **Step 1**
- Create Memento class.
- *Memento.java*

```
public class Memento {
   private String state;
   public Memento(String state){
      this.state = state;
   }
   public String getState(){
      return state;
   }
}
```

Example 2:

- **Step 2**
- Create Originator class
- *Originator.java*

```
public class Originator {
   private String state;
   public void setState(String state){
      this.state = state;
   }
   public String getState(){
      return state;
   }
   public Memento saveStateToMemento(){
      return new Memento(state);
   }
   public void getStateFromMemento(Memento memento){
      state = memento.getState();
   }
}
```

Example 2:

- **Step 3**
- Create CareTaker class
- *CareTaker.java*

```
import java.util.ArrayList;
import java.util.List;
public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();
   public void add(Memento state){
      mementoList.add(state);
   }
   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

Example 2:

- **Step 4;** Use *CareTaker* & *Originator* (*MementoPatternDemo.java*)

```
public class MementoPatternDemo {
   public static void main(String[] args) {
      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();
      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

Example 2:

- **Step 5**
- Verify the output.

Current State: State #4
First saved State: State #2
Second saved State: State #3