Design Patterns

MSc in Communications Software

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Memento

Design Pattern

Intent

- Encapsulate an object's state in such a way that no external entity can know how the object is structured. An external object (called a caretaker) can store or restore object.

Structure

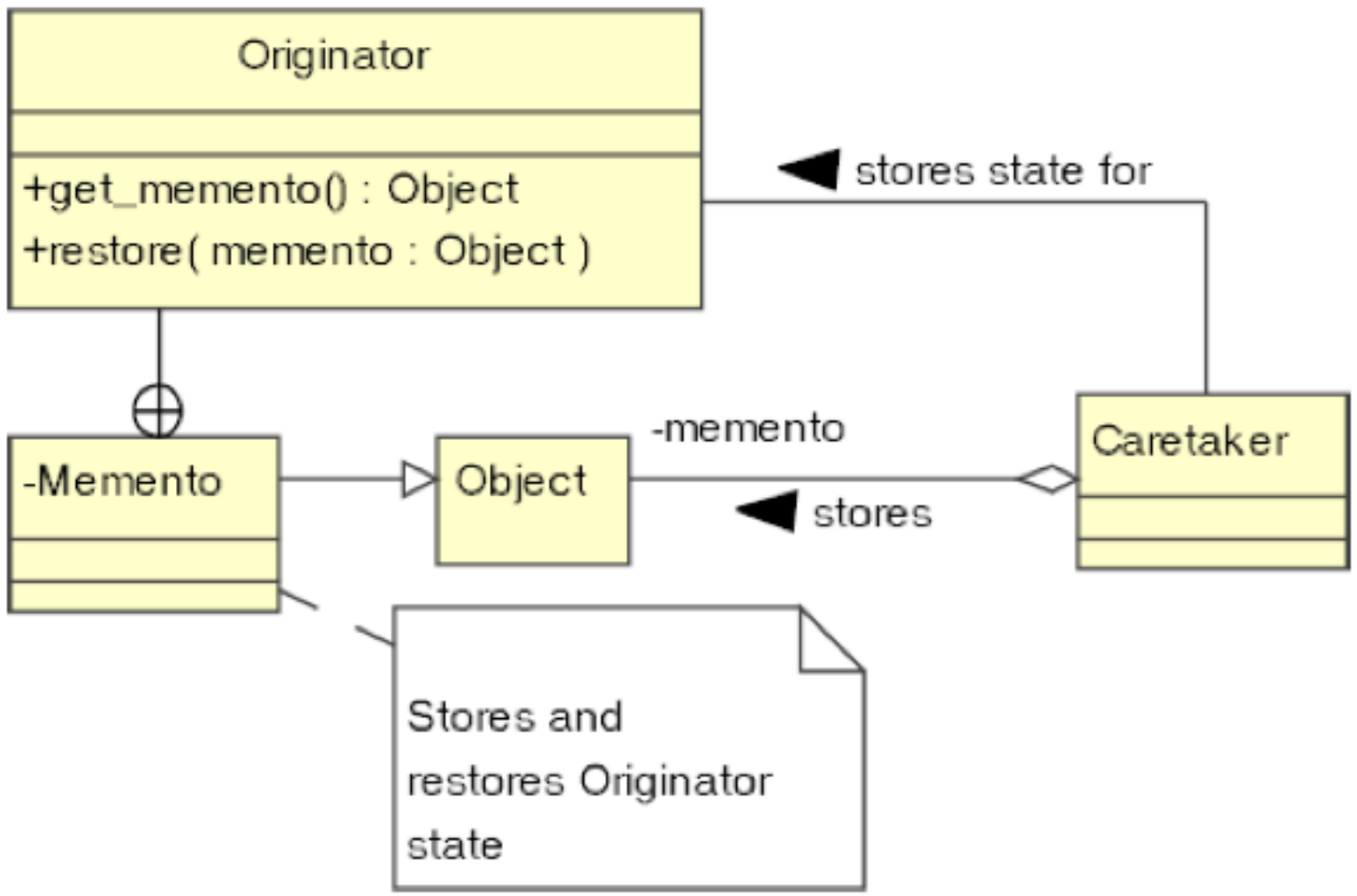
- Originator: Creates a memento that holds a “snapshot” of its current state.

- Memento: Stores the internal state of the Originator in a way that does not expose the structure of the Originator.

- Supports a “wide” interface used by the originator and a “narrow” interface used by everyone else.

- Caretaker: Stores the mementos, but never operates on them

Example




```
package scratch;

class Originator
{
  private String state;
  private int    more;

  private class Memento
  {
    private String state = Originator.this.state;
    private int more     = Originator.this.more;

    public String toString()
    {
      return state + "," + more;
    }
  }

  public Object get_memento()
  {
    return new Memento();
  }

  public void restore(Object o)
  {
    Memento m = (Memento) o;
    state = m.state;
    more = m.more;
  }
}

class Caretaker
{
  Object     memento;
  Originator originator;

  public void capture_state()
  {
    memento = originator.get_memento();
  }

  public void restore_yourself()
  {
    originator.restore(memento);
  }
}
```

Advantages

- Allows an object's state to be stored externally in such a way that the maintainability of the program is not compromised.

- Allows a "caretaker" object to store states of classes that it knows nothing about.

Disadvantages

- Versioning can be difficult if the memento is stored persistently. The originator must be able to decipher mementos created by previous versions of itself.

- It’s often unclear whether a memento should be a “deep” copy of the Originator. (i.e. should recursively copy not just references, but the objects that are referenced as well). Deep copies are expensive to manufacture. Shallow copies can cause memory leaks, and referenced objects might change values.

- Caretakers don’t know how much state is in the memento, so they cannot perform efficient memory management.

Another Example

```
class Memento
{
  private String state;

  public Memento(String stateToSave)
  {
    state = stateToSave;
  }

  public String getSavedState()
  {
    return state;
  }
}
```

```
class Caretaker
{
  private List<Memento> savedStates = new ArrayList<Memento>();

  public void addMemento(Memento m)
  {
    savedStates.add(m);
  }

  public Memento getMemento(int index)
  {
    return savedStates.get(index);
  }
}
```

```java
class Originator
{
  private String state;

  /*
   * lots of memory using private data that does not have to be saved. Instead
   * we use a small memento object.
   */

  public void set(String state)
  {
    System.out.println("Originator: Setting state to " + state);
    this.state = state;
  }

  public Memento saveToMemento()
  {
    System.out.println("Originator: Saving to Memento.");
    return new Memento(state);
  }

  public void restoreFromMemento(Memento memento)
  {
    state = memento.getSavedState();
    System.out.println("Originator: State after restoring from Memento: " + state);
  }
}
```

```java
class MementoExample
{
  public static void main(String[] args)
  {
    Caretaker caretaker = new Caretaker();
    Originator originator = new Originator();
    originator.set("State1");
    originator.set("State2");
    caretaker.addMemento(originator.saveToMemento());
    originator.set("State3");
    caretaker.addMemento(originator.saveToMemento());
    originator.set("State4");
    originator.restoreFromMemento(caretaker.getMemento(1));
  }
}
```

JDK Examples

Usage

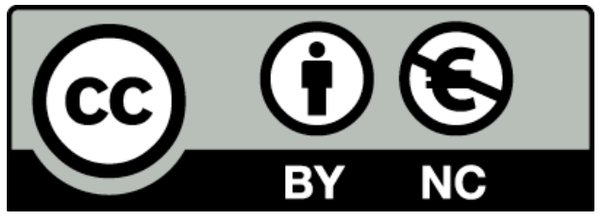
```
class Originator implements Serializable{ int x; }

ByteArrayOutputStream bytes = new ByteArrayOutputStream();
ObjectOutputStream out= new ObjectOutputStream( bytes );

Originator instance = new Originator();  // create
out.writeObject( instance );             // memento
byte[] memento = bytes.toByteArray();

ObjectInputStream in =                   // restore object
   new ObjectInputStream(                // from memento
         new ByteArrayInputStream(memento));
instance= (Originator) in.readObject();
```

A byte array is about as black as a box can be. *Decorator* is used, here, to produce a system of streams that manufacture the memento. This example also nicely illustrates a flaw in *Decorator*—that you sometimes have to access an encapsulated decorator to do work.



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit