

Patterns : Memento

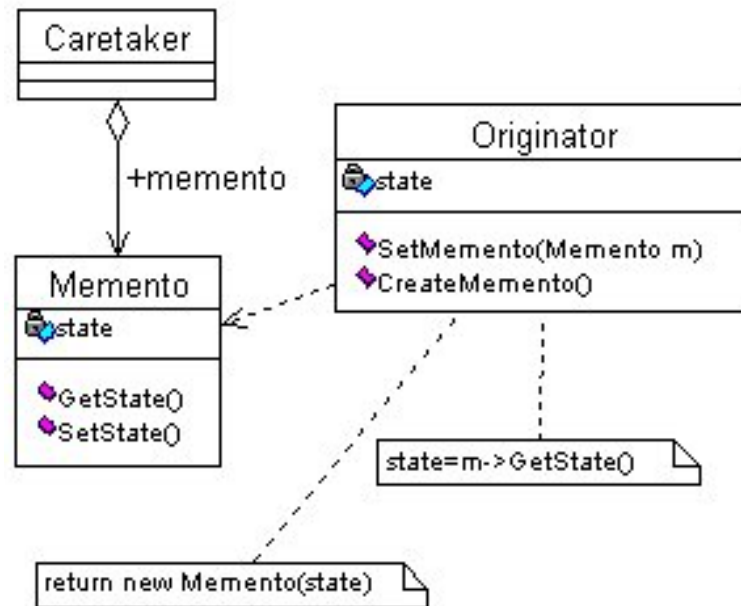
Byungjoon Lee @ NCP Team . ETRI

bjlee@etri.re.kr

<http://www.buggymind.com/>

Memento Pattern (1)

- Useful when application-level rollback feature should be added to a specific module



Template Method Pattern (2)

- Basic Idea
 - “Originator” has a state
 - “Caretaker” does not know how a state is represented
 - State representation is Originator-private
 - Caretaker can make a snapshot of the state of the Originator
 - By this snapshot, Caretaker can request the Originator to restore the internal state to that of the snapshot
 - The responsibility of maintaining the snapshot-ed states?
 - Caretaker maintains the states
 - » That is, the decision about the states maintenance is up to the implementation of the Caretaker

Example (1)

```
class Originator {
    private String state;
    /* lots of memory consumptive private data that is not necessary to define the
    * state and should thus not be saved. Hence the small memento object. */

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

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

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

    private static class Memento {
        private String state;

        public Memento(String stateToSave) { state = stateToSave; }
        public String getSavedState() { return state; }
    }
}
```

Example (2)

```
class Caretaker {
    private ArrayList savedStates = new ArrayList();

    public void addMemento(Object m) { savedStates.add(m); }
    public Object getMemento(int index) { return savedStates.get(index); }
}

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) );
    }
}
```

Benefit

- The “Caretaker” needs not to pay attention to the representation of the “State” of the “Originator”
 - The representation of the “State” can be freely modified

Disadvantages

- The maintenance algorithm of the “States” can be very inefficient by not considering the representation of the States at all
 - This little disadvantage can be easily fixed by introducing the super class of every “State” classes