Design Patterns

Memento Pattern

ebru@hacettepe.edu.tr
ebruakcapinarsezer@gmail.com
http://yunus.hacettepe.edu.tr/~ebru/
@ebru176
Kasım 2017

Intent

- Capture and externalize an object's state without violating encapsulation.
- Restore the object's state at some later time.
  - Useful when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
  - Entrusts other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Forces

- Application needs to capture states at certain times or at user discretion. May be used for:
  - Undue / redo
  - Log errors or events
  - Backtracking
- Need to preserve encapsulation
  - Don’t share knowledge of state with other objects
- Object owning state may not know when to take state snapshot.

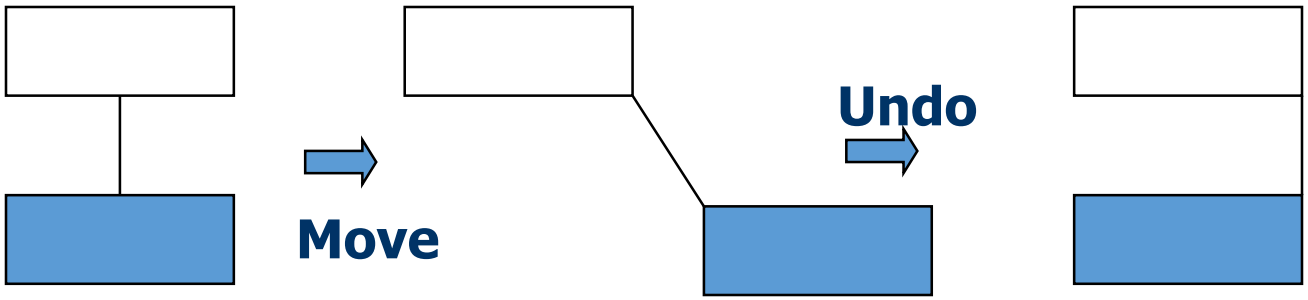
Motivation

- Many technical processes involve the exploration of some complex data structure.
- Often we need to backtrack when a particular path proves unproductive.
  - Examples are graph algorithms, searching knowledge bases, and text navigation.

Motivation

- Memento stores a snapshot of another object's internal state, exposure of which would violate encapsulation and compromise the application's reliability and extensibility.

- A graphical editor may encapsulate the connectivity relationships between objects in a class, whose public interface might be insufficient to allow precise reversal of a *move* operation.



Motivation

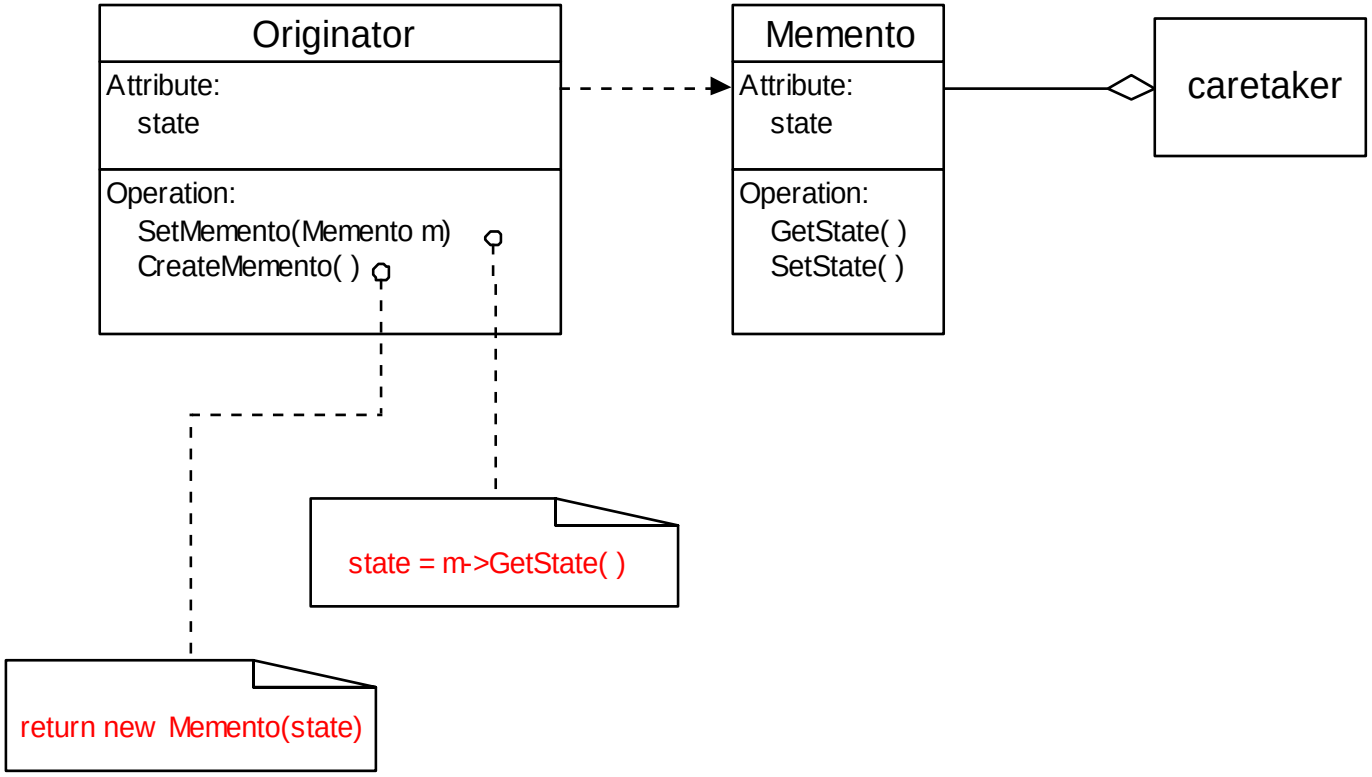
Memento pattern solves this problem as follows:

- The editor requests a memento from the object before executing *move* operation.
- Originator creates and returns a memento.
- During *undo* operation, the editor gives the memento back to the originator.
- Based on the information in the memento, the originator restores itself to its previous state.

Applicability

- Use the Memento pattern when:
  - A snapshot of an object's state must be saved so that it can be restored later, and
  - direct access to the state would expose implementation details and break encapsulation.

Structure


Originator
Attribute:
state
Operation:
SetMemento(Memento m)
CreateMemento( )
Memento
Attribute:
state
Operation:
GetState( )
SetState( )
caretaker
state = m->GetState( )
return new Memento(state)

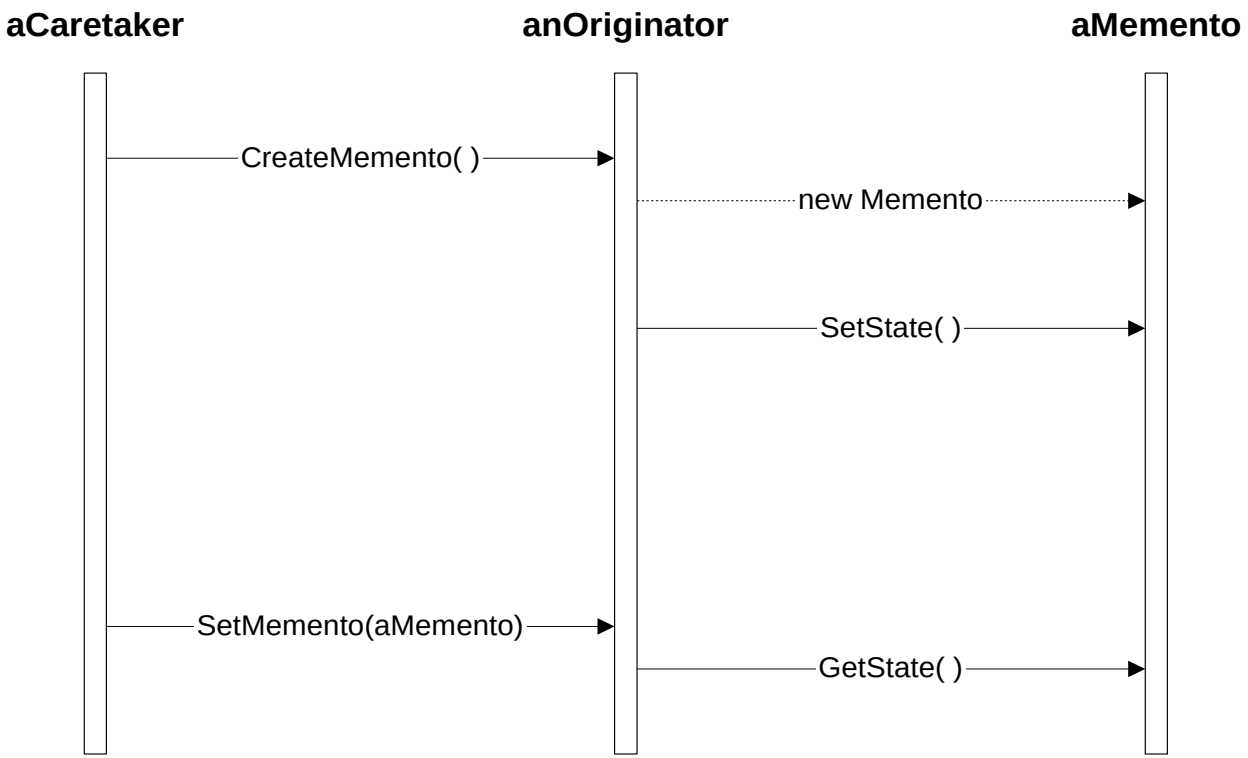
Participants

- **Memento**
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

- **Originator**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.

- **Caretaker**
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento.

Event Trace



Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

Consequences

- Memento has several consequences:
  - Memento avoids exposing information that only an originator should manage, but for simplicity should be stored outside the originator.
  - Having clients manage the state they ask for simplifies the originator.
- Using mementos may be expensive, due to copying of large amounts of state or frequent creation of mementos.
- A caretaker is responsible for deleting the mementos it cares for.
- A caretaker may incur large storage costs when it stores mementos.

Implementation

- When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just incremental changes to originator’s state.

Memento Implementation

```
// "Originator"
class AdressBook
{
    private string name;
    private string surname;
    private string tel;

    public string Name
    {
            get { return name;  }
            set { name = value; }
    }
    public string Surname
    {
            get { return surname;  }
            set { surname = value; }
    }
    public string Tel
    {
            get { return tel; }
            set { tel = value; }
    }
    public Memento SaveMemento()
    {
            return new Memento(name, surname, tel);
    }
    public void RestoreMemento(Memento memento)
    {
            this.Name = memento.Name;
            this.Surname = memento.Surname;
            this.Tel = memento.Tel;
    }
}
```

```
// "Memento"

class Memento
{
    private string name;
    private string surname;
    private string tel;

    public Memento(string name,
                   string surname,
                   string tel)
    {
            this.name = name;
            this.surname = surname;
            this.tel = tel;
    }
    public string Name
    {
            get{ return name; }
            set{ name = value; }
    }
    public string Surname
    {
            get{ return surname; }
            set{ surname = value; }
    }
    public string Tel
    {
            get{ return tel; }
            set{ tel = value; }
    }
}
```

Memento Implementation

```
// "Caretaker"
class Caretaker
{
    private Memento memento;

    public Memento Memento
    {
            set { memento = value; }
            get { return memento;  }
    }
}
```

```
class MainApp
{
    static void Main()
  {
            AdressBook book = new AdressBook();
             book.Name = "Ad_1";
             book.Surname = "Soyad_1";
             book.Tel = "Tel_1";
            // book.Display();

            // Store internal state
            Caretaker careTaker = new Caretaker();
             careTaker.Memento = book.SaveMemento();

            // Continue changing originator
             book.Name = "Ad_2";
             book.Surname = "Soyad_2";
             book.Tel = "Tel_2";
            // book.Display();

            // Restore saved state
            book.RestoreMemento(careTaker.Memento);
            //s.Display();
            }
}
```

Implementation in C++

```
class State{..};

class Originator {
public:
    void setMemento(Memento*);
    void createMemento();
   void setState(const State);
   const State getState() const;
    // ...
 private:
    State _state;
}
```

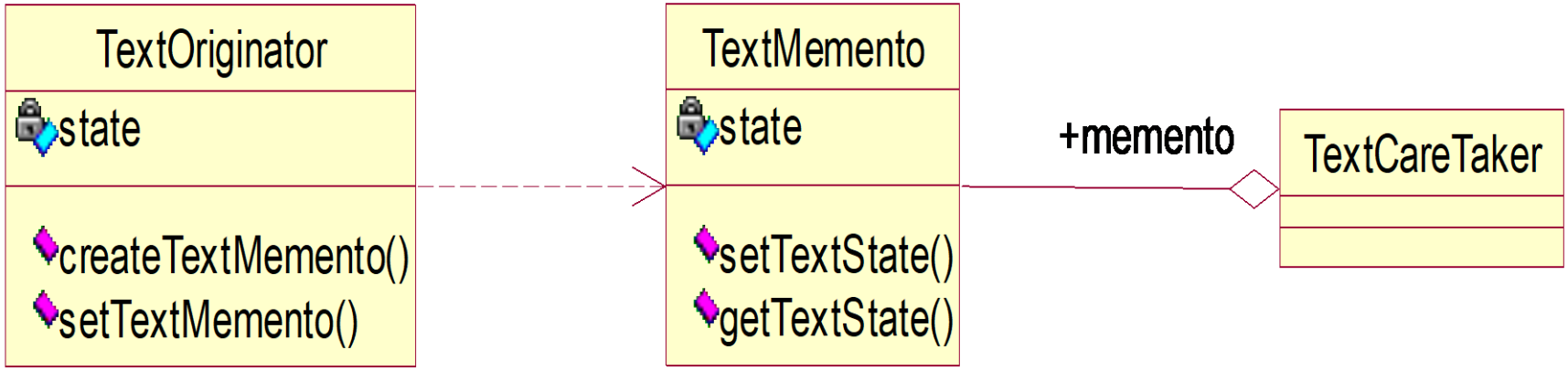
```
class Memento {
public:
    // Small Intercafe for other classes
    virtual ~Memento;
private:
    // Large Interface for the Originator
    friend class Originator;
    Memento();
    void setState(State*);
    const State* getState() const;
   //....
private:
    State* _state;
}
```

Implementation in C++ (Cont.)

```
class CareTaker
{
public:
    void setMemento(Memento*);
    const Memento* getMemento() const;
private:
    Memento* memento;
}

//for one memento copy
```

```
int main(int argc, char* argv[])
{
    Originator origin;
    origin.SetState(ON);
    cout << origin.GetState().c_str() << endl;

    CareTaker careTaker;
    careTaker.setMemento(origin.CreateMemento());


    origin.SetState(OFF);
    cout << origin.GetState().c_str() << endl;

    origin.SetMemento(careTaker.getMemento());

    cout << origin.GetState().c_str() << endl;

}
```

Another Example



Another Example

```
public class TextOriginator{
    private Cursor cursor;
    private String text;
     private int size=10;
    private Boolean italic=false;
    private Boolean bold=false;
    //  getters&& setters
    public void setMemento(Memento m){
            m.getState(this);
    }
     public Memento createMemento(){
            return(new Memento().setState(this));
    }
    public void setText(String text){
            saveMemento
            this.text=text;
    }
    public String getText(){
             return this.text;
    }
}
```

```
    public class Memento{
            private String mementoText;
            private int mementoSize;
             private Boolean mementoItalic=false;
             private Boolean mementoBold=false;
            //getters and setters
            ................

      //save the state of originator into memeto
            public void setState(TextOriginator t){
                    mementoText=t.getText();
                    mementoSize=t.getSize();
                     mementoItalic=t.getItalic();
                     mementoBold=t.getBold();
            }
      //restore the originator;
            public void getState(TextOriginator t){
                     t.setText(mementoText);
                     t.setSize(mementoSize);
                     t.setItalic(mementoItalic);
                     t.setBold(mementoBold);
            }
}
```

Another Example

```
public Class CareTaker{
        Stack <Memento> s = new Stack[10,3];
        public void pushMemento(Memento m)
        {
                s.push(m);
        }
        public Memento popMemento()
        {
                return (s.pop());
        }
 }
```

```
public class ClientClass{
public static void main (String args[]){
        Caretaker careTaker;
        TextOriginator demo=new TextOrginator();
        demo.setText("Memento deneme");
        demo.write();
// save the current
        careTaker.pushMemento(demo.createMemento( ));
// new state
        demo.initializeCursor(int x,int y);
        demo.setSize(16);
        demo.setBold(true);
        demo.setItalic(true);
        demo.write();
// undo
        demo.setMemento(careTaker.pop());
        demo.write();
} }
```

Known Uses

- ***Memento*** is a 2000 film about Leonard Shelby and his quest to revenge the brutal murder of his wife. Though Leonard is hampered with short-term memory loss, he uses notes and tatoos to compile the information into a suspect.