Memento

20

Encapsulation is a central theme of object-oriented programming. Every object has attributes that it must not make known to the outside world. Instead, it offers the narrowest possible interface. In the last chapter, with the Visitor pattern, I addressed this conflict – there, access methods had to be created specifically for the Visitor. Encapsulation is also an issue for the Memento Pattern.

20.1 Task of the Memento Pattern

The starting point is to be any object A whose state you want to store externally in order to be able to restore it later. How can the state be determined and stored? First, it is obvious to take an object B that reads the data from A and stores it. But how could the storing object B get at the data of the object A to be stored?

20.1.1 Public Data Fields

A first approach could be to make the data fields publicly available.

```
public class Memento {
    public int answer = 42;
    public String password = "Ken sent me";
}
```


O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_20

Now another object could read and store the state. But why won't you pursue this approach? The data fields are publicly viewable and modifiable, which goes against the principle of data encapsulation, information hiding.

20.1.2 The JavaBeans Pattern

Most Java implementations implement the principle of data encapsulation by having methods that can read and write to the data fields, which is more or less like the JavaBeans specification.

```
public class Memento {
    private int answer = 42;
    private String password = "Ken sent me";

    public int getAnswer() {
        return answer;
    }

    public void setAnswer(int answer) {
        this.answer = answer;
    }

    // … abridged
}
```

Now the data is encapsulated, but nothing has changed compared to the approach with the public fields – anyone can still read and write to the data, albeit via the detour of the access methods.

20.1.3 Default Visibility

Conceivably, you could set the fields – or the access methods – to default visibility (package-private); then only classes in the same package could access the data. Consequently, an object that is to store the state of another object must be defined in the same package. But even this approach is not optimal. Packages should help to group classes into meaningful units. For example, all classes that display data on the screen go into one package, while data that describes the data model is defined in another package. In my opinion, it would be excessive to create a package just to store a class and its data store in it.

The previous suggestions were not convincing – it seems like a bad idea to try to read the data. So, take a different approach: the object whose state is to be stored is itself made responsible for creating its data store.

20.2 A Possible Realization

Before I introduce you to the project, I would like to define a few terms. The object whose state is to be stored is the *originator*; in the German translation of the GoF book it is also called "Urheber". The object that stores the state of another object is the *memento*. Finally, the *caretaker* maintains a list of mementos. The example project Memento_Simple implements the pattern in a very clear way. The caretaker is the class `Stack`. It stores the different memento objects. As stack I didn't take the Java implementation, but developed my own class. The memento is defined as an empty marker interface.

```
public interface Memento {
}
```

You want to save the state of a car object. It has the attributes `speed` and `currentFuelConsumption`. The speed can be influenced; there are corresponding access methods for this. The fuel consumption can only be influenced indirectly via the speed.

```
public class Car {
    // … abridged

    private int speed = 0;
    private int currentFuelConsumption = 0;

    public void driveFaster() {
        speed++;
        calculateFuelConsumption();
    }

    public void driveSlower() {
        if (speed > 0) {
            speed--;
            calculateFuelConsumption();
        }
    }

    private void calculateFuelConsumption() {
        // … abridged
    }
}
```

Now I am looking for a way to be able to store all the data fields. For this, I create an inner class AutoMemento which is suitable to copy the data fields. Every time you create an instance of the AutoMemento class, the data fields of the AutoObject are copied.

```
public class Car {
    private class CarMemento implements Memento {
        private final int tempo = speed;
       private final int thirst = currentFuelConsumption;
    }

    private int speed = 0;
    private int currentFuelConsumption = 0;

    // … abridged
}
```

You pass the responsibility of creating a memento to the car class. There you request a memento; moreover, the car is also responsible for restoring an old state from a given memento.

```
public class Car {
    // … abridged

    public Memento createMemento() {
        return new CarMemento();
    }

    public void setMemento(Memento memento) {
        var myMemento = (CarMemento) memento;
        this.currentFuelConsumption = myMemento.thirst;
        this.speed = myMemento.tempo;
    }
}
```

The client creates a stack, the caretaker, and an auto object. Then it performs some operations that affect the speed and thus the fuel consumption, and saves the state in a memento after each change. The memento is placed on the stack; any change can now be undone. The main method of the `ApplStart` class demonstrates this procedure. Please analyze the code on your own.

How do you evaluate this solution? On the one hand, the attributes of neither the originator nor the memento can be read – so they are very consistently encapsulated. However, you buy the encapsulation at the price that the class itself is responsible for creating and restoring the memento; the code is "polluted" in this respect. If this solution appeals to you, please note that the example is extremely simple. The attributes are of the primitive type int. If you have mutable objects, they must be cloned – the problem of copied references, which you learned about in the Prototype chapter on cloning, is encountered again here.

In the next example we will use the graphic editor of the Prototype Pattern again.

20.3 A Major Project

In the second expansion stage of the GraphicEditor in the chapter about the Prototype Pattern, you cloned a diagram by serializing and deserializing it. Exactly this procedure can also be interesting for the Memento. For the following example project Memento, I took the `GraphEditor_2` as a template and modified it. The advantage now is that in the class `PanelCanvas` the method `getMemento()` is defined – it creates and returns a cloned diagram as a deep copy, the Memento.

```
public Diagram getMemento() {
    Diagram clone = null;
    try {
        ObjectOutputStream oos;
        ByteArrayInputStream bais;
        ObjectInputStream ois;
        try ( var baos = new ByteArrayOutputStream()) {
            oos = new ObjectOutputStream(baos);
            oos.writeObject(diagram);
            bais = new ByteArrayInputStream(baos.
                                        toByteArray());
            ois = new ObjectInputStream(bais);
            clone = (diagram) ois.readObject();
        }
        oos.close();
        bais.close();
        ois.close();
    } catch (IOException ex) {
        new ErrorDialog(ex);
    } finally {
        return clone;
    }
}
```

The `setDiagram()` method takes a Diagram object and fills the drawing area with its data.

```
public void setDiagram(Diagram clone) {
    selected = null;
    this.diagram = clone;
    repaint();
}
```

In the main method of the `ApplStart` class, the `undoAction` action is defined, which requests the last diagram from the caretaker and passes it to the drawing area. This action is passed to a MenuItem. The caretaker is an instance of the `Stack` class, which you know from the last example. You pass the same instance to the drawing area.

```
private final PanelCanvas canvas =
                            new PanelCanvas(caretaker);
// … abridged

private final Action undoAction =
                             new AbstractAction("Undo") {
    @Override
    public void actionPerformed(ActionEvent e) {
        myFrame.setTitle("Memento Demo");
        if (!caretaker.empty()) {
             Diagram diagram = caretaker.pop();
             canvas.setDiagram(diagram);
        }
        else
             myFrame.setTitle("…");
    }
};
```

You could trigger most actions via mouse clicks: Moving circles, creating lines or circles, and selecting graphic elements. You deleted graphic elements via the menu. All the methods you called this way were defined in the PanelCanvas class – either as a standalone method or as an EventListener. These methods are supplemented so that a snapshot is created and saved at the same time as the defined action.

```
public void deleteSelectedGraphic() {
    if (selected != null) {
        createMemento();
        diagram.deleteSelectedGraphic(selected);
        selected = null;
        repaint();
    }
}

private void createMemento() {
    var tempDiagram = getMemento();
    caretaker.push(tempDiagram);
}
```

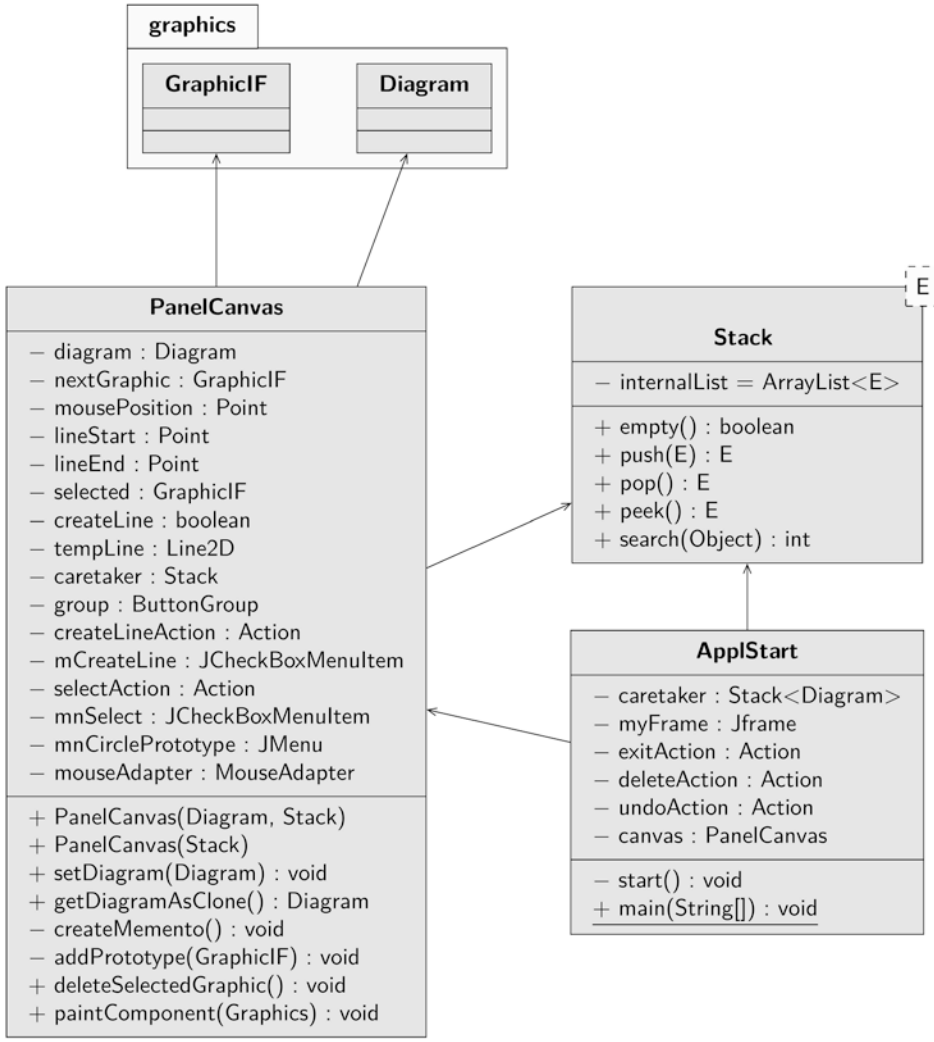

Fig. 20.1 UML diagram of the Memento Pattern (example project Memento)

In this realization, you store an object of type `Diagram` in the caretaker. Alternatively, you could have stored the serialized object in the caretaker.

20.4 Memento – The UML Diagram

In Fig. 20.1, I show you only some of the classes involved in the Memento sample project. I have not shown the graphics and prototype packages again here. You can find their representation in the UML diagram for prototype in Sect. 17.3.

20.5 Summary

Go through the chapter again in key words:

- The goal is to store the state of an object externally.
- The attributes of the object must be accessed.
- The encapsulation should not be weakened if possible.
- To implement the pattern, have the originator create a memento.
- The Caretaker stores the mementos.

20.6 Description of Purpose

The Gang of Four describes the purpose of the pattern “Memento” as follows:

> Capture and externalize the internal state of an object without violating its encapsulation, so that the object can later be restored to that state.