The Memento Design Pattern

The intent of the Memento pattern is to provide storage and restoration of an object’s state.

• Some features
  – The memento pattern is used by two objects: the originator and a caretaker. The originator is some object that has an internal state.
  – The caretaker is going to do something to the originator, but wants to be able to undo the change.
  – The caretaker first asks the originator for a memento object.
  – Then it does whatever operation (or sequence of operations) it was going to do.
  – To roll back to the state before the operations, it returns the memento object to the originator.

• Some common applications – using a seeded random number generator or using a finite state machine

• In the GoF classification this is a behavioral design pattern
Acknowledgements

• Materials were borrowed from
  – *Design Patterns in Java* by Steven Metsker and William Wake (textbook for course)
  – *Head First Design Patterns* by Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra
  – Information on the previous slide came from the wikipedia article on the memento pattern
Using Undo in our Oozinoz Factory

- The engineers use a graphical user interface to move object between production lines; here is the interface at the start and after some changes.
What information should be saved?

• The state can be described by a list of the locations of the machines that the user has placed
  
  • Each time the user adds or moves a machine in the visualization, your code will create a memento of the simulated factory and add it to a stack.
  
  • Each time the user clicks the Undo button, your code will pop the most recent memento and then restore the simulation to the state stored at the top of the stack.

• This could be implemented using a simple stack, but if we want to make our design expandable and scalable, it is better to use a model-view-controller approach
The Basic Design

• We will first concentrate on the factory model
• Each time there is a change by adding a machine or moving a machine the factory will save a memento
• The visualization GUI must register to be updated on changes the user makes to the system
The Factory Model

- `mementos: Stack`
- `listeners: ArrayList`

- `add(loc: Point)`
- `drag(oldLoc: Point, newLoc: Point)`
- `getLocations: List`
- `canUndo: boolean`
- `undo()`
- `notifyListeners()`
- `addChangeListener(:ChangeListener)`

- A stack is used to maintain the mementos; the listeners are stored in a list
Some Code Snippets

• The constructor initializes the stack and arraylist
• When a machine is added then copy the last memento from the top of the stack, add the location of the new machine at position 0, push the new memento onto the stack and notify the listeners
• getLocations returns the latest configuration

```java
package com.oozinoz.visualization;

public class FactoryModel {
    private Stack mementos;

    private ArrayList listeners = new ArrayList();

    public FactoryModel()
    {
        mementos = new Stack();
        mementos.push(new ArrayList());
    }
    //...

    public void add(Point location)
    {
        List oldLocs = (List) mementos.peek();
        List newLocs = new ArrayList(oldLocs);
        newLocs.add(0, location);
        mementos.push(newLocs);
        notifyListeners();
    }

    public List getLocations()
    {
        return (List) mementos.peek();
    }
}
Implementing the Undo

• The undo action has to remove the current configuration from the top of the stack and notify listeners that the system has changed.

**CHALLENGE 19.1**

Write the code for the `FactoryModel1` class’s `undo()` method.
public boolean canUndo() {
    return mementos.size() > 1;
}

public void undo() {
    if (!canUndo()) return;
    mementos.pop();
    notifyListeners();
}
The MVC Design

- The visualization class creates the controls but a mediator is used to handle any events.
- The visualization changes factory events into GUI changes.
- The mediator translates GUI events into factory changes.
More Code Snippets

• The undo button constructor registers the mediator as a listener

```java
protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator.undoAction());
    }
    return undoButton;
}
```

• The mediator notifies the factory of the undo request

```java
private void undo(ActionEvent e) {
    factoryModel.undo();
}
```

• We have already seen how the factory pops the most recent memento from the stack and notifies listeners

```java
public Visualization(UI ui) {
    super(new BorderLayout());
    this.ui = ui;
    mediator = new VisMediator(factoryModel);
    factoryModel.addChangeListener(this);
    add(machinePanel(), BorderLayout.CENTER);
    add(buttonPanel(), BorderLayout.SOUTH);
}
```

• The visualization constructor creates the mediator and passes in a handle to the factory

• It also listens for any changes in the factory so it can change the display
The Message Flow for an Undo

- Here is the flow of events that occurs when the user clicks undo.
The stateChanged Method

• When the Visualization class is notified of any changes in the factory model, it must display the new configuration

**CHALLENGE 19.2**

Write the stateChanged() method for the Visualization class.
public void stateChanged(ChangeEvent e) {
    machinePanel().removeAll();

    List locations = factoryModel.getLocations();

    for (int i = 0; i < locations.size(); i++) {
        Point p = (Point) locations.get(i);
        machinePanel().add(createPictureBox(p));
    }

    undoButton().setEnabled(factoryModel.canUndo());
    repaint();
}
Memento Durability

• So far we have shown own mementos stored in a volatile data structure that disappears between program sessions
• It may be desirable to improve the “durability” of the mementos

CHALLENGE 19.3

Write down two reasons that might drive you to save a memento in a file rather than as an object.
Storing a memento as an object assumes that the application will still be running when the user wants to restore the original object. Reasons that will force you to save a memento to persistent storage include the following.

- The ability to restore an object’s state has to survive a system crash.
- You anticipate that the user will exit the system and will want to resume work later.
- You need to reconstruct an object on another computer.
Saving Mementos Across Sessions

• The screen below shows options to save the current configuration and associated mementos to a file and to, at a later time, restore the session

• What mechanism does Java provide for easily storing collections in a file and restoring the collections at a later time? (our actual code will not use this mechanism)
package com.oozinoz.visualization;

import javax.swing.*;
import com.oozinoz.ui.SwingFacade;
import com.oozinoz.ui.UI;

public class Visualization2 extends Visualization {
    public Visualization2(UI ui) {
        super(ui);
    }

    public JMenuBar menus() {
        JMenuBar menuBar = new JMenuBar();

        JMenu menu = new JMenu("File");
        menuBar.add(menu);

        JMenuItem menuItem = new JMenuItem("Save As...");
        menuItem.addActionListener(mediator.saveAction());
        menu.add(menuItem);

        menuItem = new JMenuItem("Restore From...");
        menuItem.addActionListener(
            mediator.restoreAction());
        menu.add(menuItem);

        return menuBar;
    }

    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame = SwingFacade.launch(
            panel, "Operational Model" );
        frame.setJMenuBar(panel.menus());
        frame.setVisible(true);
    }
}
public ActionListener saveAction() {
    return new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                VisMediator.this.save((Component)e.getSource());
            } catch (Exception ex) {
                System.out.println(
                    "Failed save: " + ex.getMessage());
            }
        }
    };
}

public void save(Component source) throws Exception {
    JFilechooser dialog = new JFilechooser();
    dialog.showSaveDialog(source);

    if (dialog.getSelectedFile() == null)
        return;

    FileOutputStream out = null;
    ObjectOutputStream s = null;
    try {
        out = new FileOutputStream(dialog.getSelectedFile());
        s = new ObjectOutputStream(out);
        s.writeObject(factoryModel.getLocations());
    } finally {
        if (s != null) s.close();
    }
}
The restoreAction method

• Assume the file is saved using the code given on the previous slide
• Your task is to write the restore method
  – Use JFileChooser to select the file to be opened
  – Create an object input stream
  – Read the objects one at a time and put into a list
  – Send the list to the appropriate method in the factory model

**CHALLENGE 19.4**

Write the code for the restoreAction() method of the VisMediator class.
public void restore(Component source) throws Exception {
    JFileChooser dialog = new JFileChooser();
    dialog.showOpenDialog(source);

    if (dialog.getSelectedFile() == null)
        return;

    FileInputStream out = null;
    ObjectInputStream s = null;
    try {
        out = new FileInputStream(dialog.getSelectedFile());
        s = new ObjectInputStream(out);
        ArrayList list = (ArrayList) s.readObject();
        factoryModel.setLocations(list);
    } finally {
        if (s != null)
            s.close();
    }
}
Should we use a textual representation?

- The Java methods XMLEncoder and XMLDecoder allow us to easily save collections to XML files
- XML files are textual and can be ready by anyone
- Is saving in XML files (as contrasted to binary files) a good idea?

**CHALLENGE 19.5**

In this case, we used Java serialization to write to a file in binary format. Suppose that we had written it to XML format (textual) instead. Write a short statement of whether, in your opinion, saving a memento in textual form would violate encapsulation.
Solution 19.5

To *encapsulate* is to limit access to an object’s state and operations. Saving an object, such as a collection of factory location points, in textual form exposes the object’s data and allows anyone with a text editor to change the object’s state. Thus, saving an object in XML form violates encapsulation, at least to some degree.

Violation of encapsulation through persistent storage may be a concern in practice, depending on your application. To address this threat, you might limit access to the data, as is common in a relational database. In other cases, you might encrypt the data, as is common when transmitting sensitive HTML text. The point here is not whether the words *encapsulation* and *memento* apply to a design but rather the real importance of ensuring data integrity while supporting the data’s storage and transmission.
**Memento Benefits**

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object’s data encapsulated.
- Provides easy-to-implement recovery capability.

**Memento Uses and Drawbacks**

- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system’s state.
A scenario

Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled “level 13.” As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably distressed when their character gets snuffed, and they have to start all over. The cry goes out for a “save progress” command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The “save progress” function needs to be designed to return a resurrected player to the last level she completed successfully.
The Memento at work

The Memento has two goals:

- Saving the important state of a system’s key object.
- Maintaining the key object’s encapsulation.

Keeping the single responsibility principle in mind, it’s also a good idea to keep the state that you’re saving separate from the key object. This separate object that holds the state is known as the Memento object.

```java
// when new level is reached
Object saved = (Object) mgo.getCurrentState();

// when a restore is required
mgo.restoreState(saved);
```

While this isn’t a terribly fancy implementation, notice that the Client has no access to the Memento's data.

```java
GameMemento
savedGameState

Client

// when new level is reached
Object saved = (Object) mgo.getCurrentState();

// when a restore is required
mgo.restoreState(saved);

MasterGameObject

gameState

Object getCurrentState() {
    // gather state
    return(gameState);
}

restoreState(Object savedState) {
    // restore state
}

// do other game stuff
```
Homework Exercise

- Expand this example and add new classes as appropriate to transform the given solution to use a MVC perspective
- Use Violet to draw a new class diagram
- Use Violet to draw two sequence diagrams, one for the saving of a game state and one for restoring a game state