Model-View-Controller and Observer/Observable

This tutorial is aimed at introducing the concept of the Model-View pattern/framework. This is a technique of separating, or de-coupling the user interface from the underlying data model. This has several advantages such as producing more reusable components and more flexibility in the software produced.

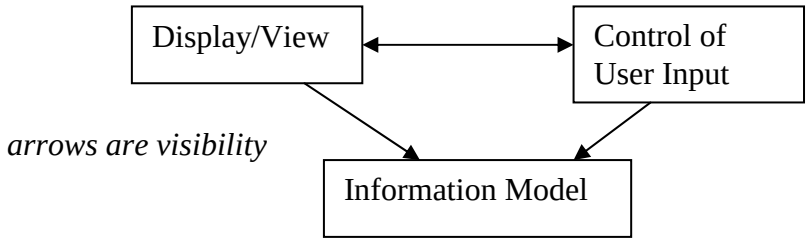
The idea of Java Observers and Observables will be introduced and how they can be used to implement the Model-View idea.

1. What are Patterns and FrameWorks and the Model-View-Controller?
A **framework** is a reusable design of a program or part of a program that is expressed as a set of classes. It can be thought of as a set of pre-fabricated software building blocks that can be extended, used or customised for a more specific solution. When you use a framework, there is no need to start from scratch each time you need to write another application, the framework can be used as a basis on which to build.

It is not just the code that can be reused, but also the design **idea**. The idea of a design that can be reused is called a **pattern**.

Originally MVC was a pattern. Now it is also a Framework in that it can be implemented using Observer/Observable Java classes/interfaces.

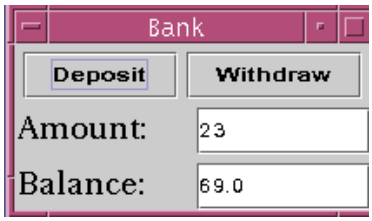
The **Model-View-Controller (MVC)** pattern and framework is the separation of the user display from the control of user input and from the underlying data model or representation.



This is considered to be useful for a number of reasons :
_ resusability of the application and or user interface components
_ ability to develop the application and user interface separately
_ ability to inherit from different parts of the class hierarchy
_ have multiple views of the same data model
_ build combinations of views and models in the same frame / window easily

This means that different interfaces can be used with the same application, without the **application** knowing about it. Additionally, any part of the system can be changed without effecting the operation of the other.

2. Example Code Without MVC – the bank program



Key points to note (no particular order) :

- BankSwing contains a number of subclasses used for creating button and text_fields etc.
- The data model and operations on it, the balance variable , the withdraw and deposit methods are contained within AccountInterface. There is no separate data model.
- The main class handles all event situations by implementing the ActionEvent interface as well as the WindowEvent interface. There is no separate control mechanism.
- Its relatively easy to follow what's going on. i.e. its not too large and complicated.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** A very basic GUI to handle a bank account
 ** My goal was to get something useful that shows the basics,
 ** but is very short - this is far from great programming!
 ** In particular the Model is simply a double variable (balance)
 ** it and the View/Controller are somewhat mixed up!
 ** @author Lynda Thomas
 */

public class BankSwing extends JFrame implements ActionListener{
////////////////////////////////////////////////instance variables
    private double balance = 0;
    private TextPanel textPanel;
    private ButtonPanel buttonPanel;

    /**
     * constructor builds window
     * puts in a TextPanel and a ButtonPanel (see below)
     */
    public BankSwing() {
        setTitle("Bank");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        textPanel = new TextPanel();
        add (textPanel, BorderLayout.CENTER);

        buttonPanel = new ButtonPanel(this);
        add (buttonPanel, BorderLayout.NORTH);

        pack();
        setVisible(true);
    }
```

```
    /**
     * This method is called when a button is pressed
     * This is because 'this' is listening to the buttons
     */
    public void actionPerformed(ActionEvent e) {
        try{
            double amount
=(Double.valueOf(textPanel.inputField.getText())).doubleValue();
            if (e.getSource() == buttonPanel.withdrawButton)
                withdraw(amount);
            else //deposit
                deposit(amount);
        }
        catch (NumberFormatException n) {
            JOptionPane.showMessageDialog(null, "Enter an amount");
        }
    }

    //housekeeping methods to model the Bank Account
    private void withdraw(double amt) {
      balance -= amt;
      textPanel.balanceField.setText(new Double(balance).toString());
    }

    private void deposit(double amt) {
       balance += amt;
      textPanel.balanceField.setText(new Double(balance).toString());
    }


////////////////////////////////////////////////////////////////
//////// INNER CLASSES /////////////////////////////////////////
    class TextPanel extends JPanel {
        JTextField inputField, balanceField;
        //Q. why not private?        //A. ............
        TextPanel() {
            setLayout(new GridLayout(2,2,5,5));
            add (new JLabel("Amount: "));
            inputField = new JTextField(8);
            add (inputField);

            add (new JLabel("Balance: "));
            balanceField = new JTextField(8);
            balanceField.setEditable(false);
            add (balanceField);
        }
    }
    class ButtonPanel extends JPanel {
        JButton depositButton;
        JButton withdrawButton;
        ButtonPanel (ActionListener control) {
            depositButton = new JButton("Deposit");
            depositButton.addActionListener(control);
            add(depositButton);
            withdrawButton = new JButton("Withdraw");
            withdrawButton.addActionListener(control);
            add(withdrawButton);
        }
    }

}
```

So what's wrong ???
On the surface, this kind of implementation is OK. Certainly for the size and complexity of the program this is an adequate solution.

However, what if the situation were slightly different. What if more operations needed to be added, what if you were developing an accounting system for a whole bank (a more complicated one). This would lead to possible different interfaces for different users, more people working on the project at the same time, the underlying data model may need changing from time to time and so on.

Using the above solution is OK for small scale, but the solution does not scale well and does not lend itself to any useful reuse of components. This is for a number of reasons :
- There is an inherent coupling between the user interface and the underlying model, you can not change one without changing the other in some way, especially if its a big change
- There is no way to develop the data model separately using this way as it is all tied up in one class
- It would be difficult to extend the functionality of the data model. Presently it records the balance level, you may want to add features such as inform manager when it becomes overdrawn, notify the customer and so on. Presently, the only access to the data model is via this simple interface, it would be nice to be able to use the model in different places for different things. This is also true for updating the model.
- Reuse is not possible really as everything is so tied together and there are no separate components to use, only the one main class which controls everything.

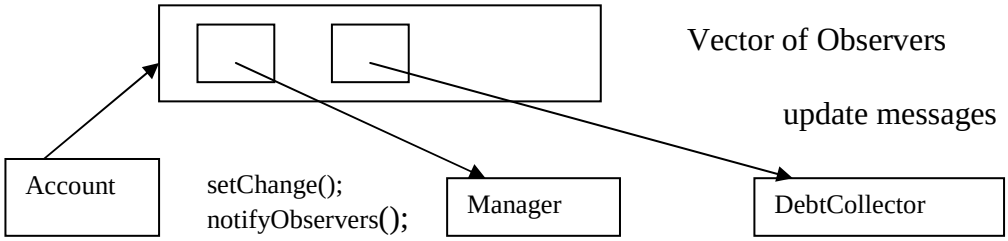
That’s enough to be going on with, but I'm sure you could possibly think of some more.

What we would therefore like is a solution to allow us to do the following :
- Allow reuse of components, by actually creating some
- Allow for a scaling of the system for functionality extensions and so on
- Decouple the data model from the user interface so the two can be as separate as possible, thus increasing their usefulness to other parties

3. Implementing the MVC pattern
In order to implement the MVC idea, Java provides a few useful classes, these being the Observer and Observable classes in the java.util package and the delegation event model.

Introducing Java Observers and Observables
In Java, one object can be said to be dependent on another if the state of one of the objects changes, the state of the other automatically changes in an appropriate manner. This dependency can be represented in Java using the Observer and Observables classes in the java.util package. The object on which the other depends on is called the observable object, whilst the other is the observer . The observable object allows the other object to observe its current state. An observable can have more than one observer, which are all notified when the state of the observable changes, via the notifyObservers() method.

Each observable stores 'who can observe it' in a vector (that Java takes care of), using the addObserver() method record them. When the notifyObservers() method is called, each observer on the list is informed of the changes via the update() method.
Below is a simple example. The account class has two observers, the manager and the debt collector. When the account class is changed, both the observers are informed there has been a change and the observers then know to interrogate the model for the changes. It is also possible to pass data to the update() method from the observable object if desired.



4. Example with MVC and Observer Observable
Here is the same Bank program using MVC/Observer Observable. As happens quite commonly the View and the Controller are put together – getting the Model separate is the most important thing. Here it is:

```
import java.util.*;
public class Account extends Observable{
    protected double balance =0;

    public void withdraw(double amt) {
        balance -= amt;
        setChanged();
        notifyObservers();   //can have an object in brackets
    }
    public void deposit(double amt) {
        balance += amt;
        setChanged();
        notifyObservers();
   }
    public double getBalance() {
        return balance;
    }
}
```

Now we have encapsulated the Model, and the View just references the relevant methods of the model as needed.

Note that we need a new method update() in the view that is called whenever a change happens in the Model.

This code is the panel that is in the BankAccountView frame

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * This class implements Observer, picks up what has happened to account and
displays
 */
public class TextPanel extends JPanel implements  Observer{
    private JTextField inputField;
    private JLabel balanceField;
    public TextPanel() {
        setLayout(new GridLayout(2,2,5,5));

        add (new Label("Amount: "));
        inputField=new JTextField("0",8);
        add (inputField);

        add (new Label("Balance: "));
        balanceField=new JLabel("0");
        add (balanceField);
    }

    /**
     * This method is needed by the listener to get what is typed
     */
    public double getInputField() {
        return Double.valueOf(inputField.getText());
    }

     /**
     * gets called when model is updated
     */
    public void update(Observable acc,Object blank) {//must have an OBject
in params too
        double balance=((Account)acc).getBalance();
        balanceField.setText(new Double(balance).toString());
    }
}
```

The Controller was written separately this time:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

/**
 * This is the thing that listens to Button presses.
 * It and ButtonPanel are the control part of MVC
 */
public class AccountListener implements ActionListener{
    private TextPanel textPanel;    //needs a link to here so can get what
they type
    private Account account;        //model
    public AccountListener(TextPanel tp, Account acc) {
        textPanel=tp;
        account=acc;
    }

    /**
     * called when button pressed
     * gets amount and updates the model
     */
```

```
    public void actionPerformed(ActionEvent e) {
      String action=e.getActionCommand();
      try{
        double amount=(textPanel.getInputField());
        if (action.equals("Withdraw"))
            account.withdraw(amount);
        else //deposit
            account.deposit(amount);
      }
      catch(NumberFormatException n) {
        JOptionPane.showMessageDialog(null, "Enter an amount");
       }

    }
}
```

So what's right ???

- Neither the display nor the controllers hold onto the balance. It is obtained from the account whenever it is needed. This means the account model can easily be changed without any effect on the other classes.
- When the controller asks the model to change it does not tell the display- the display finds out about the change through the observer/observable mechanism.
- The account is unaware as to where the messages (deposit etc) come from. This means any object can send a deposit message and the account would still inform its dependents about the change. This allows for other interfaces to the same model, at the same time - all being updated when a change is made.

We can add another View!

```
* This class provides an absolutely bare bones view of account object
 * done with observer/observable so automatically updated
 */
public class ManagerView extends JFrame implements Observer{
    private JLabel label;

    public ManagerView() {
        setTitle("Bank Manager's view of the account");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocation(100,100);

        label=new JLabel("nothing happened on account yet");
        add ("Center",label);

        pack();
        setVisible(true);
    }

    /**
     * gets called when model is updated
     */
    public void update(Observable acc,Object blank) {
            //must have an Object in params too
        double balance=((Account)acc).getBalance();
        label.setText(new Double(balance).toString());
    }
}
```

Run them both at once:

```
public class Demo{

    public static void main(String args[]) {
        Account account;            //model

        BankAccountView view1;      //frame contains view 1
        ManagerView view2;          //frame is view 2

        account=new Account();

        view1=new BankAccountView(account);
        view2=new ManagerView();

        account.addObserver(view2);
    }
}
```