MVC in Java ME

This article discusses how to implement a generic Model-View-Controller (MVC) architectural pattern using Java

Model-View-Controller

Applications built to the Model-View-Controller (MVC) architectural pattern contain three kinds of component:

- Models (contain the data and the functionality for manipulating it)
- Views (retrieve data from the Models and display it)
- Controllers (attach to the user interface so that updates can be sent to the Models)

Every time a Model is changed, update notifications must be sent to all Views that are interested in that Model, so that they can retrieve the updated information and refresh themselves.

The View

Views need to implement an interface like the following, so that they can receive update messages from Models.

```
/**
 *  A class must implement the View interface, in order to be notified when a Model
changes.
 *  Roughly equivalent to java.util.Observer.
 *  @author Graham Hughes
 */
public interface View {
    /**
     *  Called when "model" is changed.
     *  @param model the Model that has changed
     *  @param arg a model-specific notification object (passed to the model's
notifyViews() method)
     */
    public void update(Model model, Object arg);
}
```

The Model

Models need to extend a common super class (as below), which defines how they interact with Views. All Models need to maintain a list of interested Views, need a "changed" status, and need the ability to inform all interested Views when they have changed.

```
import java.util.Vector;
import java.util.Enumeration;

/**
 *  The abstract super class for any Model to which a View can attach.
 *  Roughly equates to java.util.Observable.
 *  @author Graham Hughes
 */
public abstract class Model {
    private Vector views;
    private boolean changed;

    public Model() {
        views = new Vector();
    }
```


(C) Copyright Nokia 2014. All rights reserved.

```
    /** Add a View to this Model's set of views, if it is not already.
     *  @param v View to add
     *  @throws NullPointerException if v is null
     */
    public synchronized void addView(View v) {
        if (v == null) {
            throw new NullPointerException();
        }
        if (!views.contains(v)) {
            views.addElement(v);
        }
    }

    /** Delete a View from this Model's set of views. */
    public synchronized void deleteView(View v) {
        views.removeElement(v);
    }

    /** Remove all Views from this Model. */
    public synchronized void deleteViews() {
        views.removeAllElements();
    }

    /** Notify this Model's Views if it has changed, and clear the changed status.
Views are notified in no specific order. */
    public synchronized void notifyViews(Object arg) {
        if (hasChanged()) {
            for (Enumeration en = views.elements(); en.hasMoreElements();) {
                ((View) en.nextElement()).update(this, arg);
            }
            clearChanged();
        }
    }

    /** Equivalent to notifyViews(null). */
    public void notifyViews() {
        notifyViews(null);
    }

    /** @return true if this Model has been changed. */
    public synchronized boolean hasChanged() {
        return changed;
    }

    /** Set this Model into the "changed" state. */
    protected synchronized void setChanged() {
        changed = true;
    }

    /** Clear this Model's "changed" state. */
    protected synchronized void clearChanged() {
        changed = false;
    }
}
```


(C) Copyright Nokia 2014. All rights reserved.

The Controller

Since Controllers only *send* messages to Models, and don't receive them, they require no special code as part of this framework.

Example

```
public class MVCExample {
    public static void main(String[] args) {
        // create the Model
        MyModel m = new MyModel();
        // create and attach a View
        MyView v = new MyView(m);
        // update the Model (act as a Controller)
        m.setMessage("fish for dinner!");
    }
}

// The Model
class MyModel extends Model {
    private String message;

    public void setMessage(String newMessage) {
        message = newMessage;
        setChanged();
        notifyViews();
    }

    public String getMessage() {
        return message;
    }
}

// The View
class MyView implements View {
    public MyView(MyModel model) {
        model.addView(this);
    }

    public void update(Model m, Object arg) {
        System.out.println(((MyModel) m).getMessage());
    }
}
```


(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.