June 2008

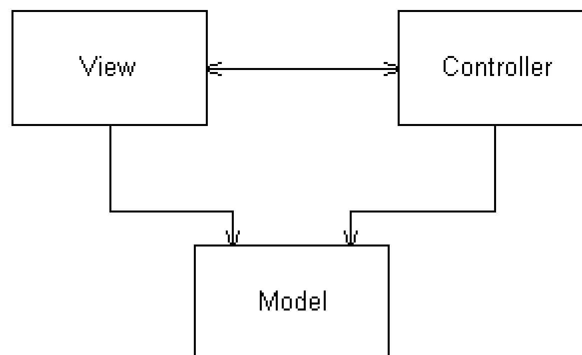
Model View Controller

with Java Swing

Paul Grenyer

Introduction

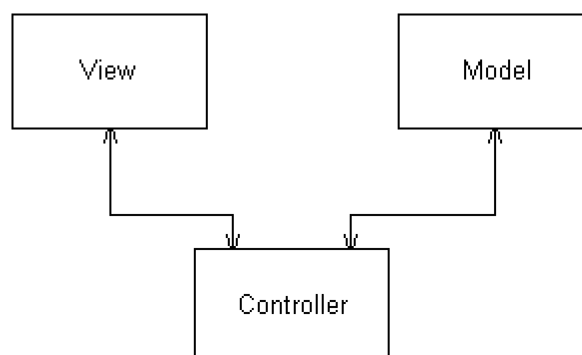
In Patterns of Enterprise Application Architecture [PEAA] Martin Fowler tells us that the Model View Controller (MVC) splits user interface interaction into three distinct roles:



- **Model** – The model holds and manipulates domain data (sometimes called business logic or the back end).
- **View** – A view renders some or all of the data contained within the model.
- **Controller** – The controller takes input from the user and uses it to update the model and to determine when to redraw the view(s).

MVC is all about separating concerns. The model and views separate the data from the views and the controller and the view separate user input from the views.

Another version of the MVC pattern employs the controller as a mediator between the views and model:



The controller still takes user input, but now it passes it on to model. It also passes commands from the view to the model and takes events from the model and passes them on to the view. This version provides greater separation as the model and view no longer need to know about each other to communicate.

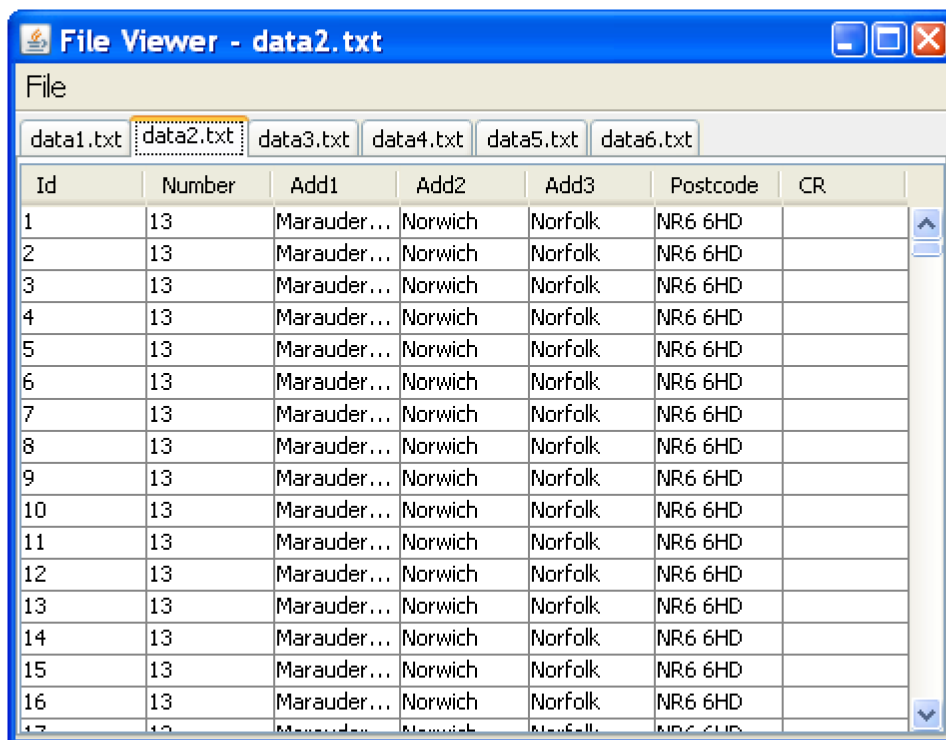
In this article I am going to describe a real problem I had and demonstrate how I solved it in Java with MVC. I am going to assume familiarity with both Java 6 and Swing.

It is very important to implement MVC carefully with a good set of tests. Benjamin Booth discusses this in his article The M/VC Antipattern. [MVC Antipattern].

The Problem

As I have mentioned in previous articles (and probably to everyone's boredom on accu-general) I am writing a file viewer application that allows fixed length record files in excess of 4GB to be viewed without loading the entire file into memory. I have had a few failed attempts to write it in C# recently (although recent discoveries have encouraged me to try again), but it was not until I had a go in Java with its `JTable` and `AbstractTableModel` classes that I really made some progress. These two classes are themselves a model (`AbstractTableModel`) and view (`Jtable`). However, in the example I'll discuss in this article they actually form part of one of the views.

The file viewer application needs to be able to handle multiple files in a single instance. The easiest and most convenient way to do this is with a tabbed view:



I have completed the back-end business logic which models the file and its associated layout (which describes how records are divided up into fields) as a project with the following interface:

```
public interface Project
{
            public abstract RecordReader getRecordReader()
                        throws RecordReaderException;

            public abstract Layout getLayout() throws LayoutException;

            public abstract String getDataDescription();
```

```
        public abstract void close();
    }
```

I will look in a bit more detail at most of the methods in the interface as the article progresses, but for now the important methods are `getRecordReader` and `getLayout`. `getRecordReader` gives the `AbstractTableModel` random access to the records and fields in the file and `getLayout` gives access to the layout which allows the view to name and order its fields.

The table model implementation I have looks like this:

```
public class RecordGrid extends AbstractTableModel
{
    private final Project project;
    private Record record = null;

    public RecordGrid(Project project)
    {
        this.project = project;
    }

    public Project getProject()
    {
        return project;
    }

    @Override
    public int getRowCount(  )
    {
        return (int) project.getRecordReader().getRecordCount();
    }

    @Override
    public int getColumnCount(  )
    {
        return project.getLayout().getFieldCount();
    }

    @Override
    public Object getValueAt(int row, int column)
    {
        if (column == 0)
        {
            record = project.getRecordReader().getRecord(row);
        }

        return record.getField(column);
    }

    @Override
    public String getColumnName(int column)
    {
        return project.getLayout().getFieldName(column);
    }

    @Override
    public Class<? extends String> getColumnClass(int column)
    {
        return String.class;
    }
}
```

I've omitted exception handling and a few other bits and pieces that are not relevant to the example. Basically a RecordGrid object holds a reference to a Project object and uses it to populate the table cells and column titles on request. Rows are populated one at a time from column 0 to column x, so every time column 0 is requested a new record is loaded. This reduces the amount of file access that would be required if a record was loaded every time a cell was requested.

Every time a new project is created the following code is used to create a tab for the file:

```
public class DataPane extends JTabbedPane
{
          public void addProject(Project project)
          {
                    RecordReader recordReader = project.getRecordReader();
                    TableModel model = new RecordGrid(project);
                    JTable table = new JTable(model);
                    this.addTab(       recordReader.getDataDescription(),
                                       new JScrollPane(table));
                    this.setSelectedIndex(this.getTabCount()-1);
          }

          ...
}
```

Again, exception handling has been omitted. A reference to the RecordReader is created in order to get a name for the tab by calling the getDataDescription method. The new Project object is passed to a new RecordGrid object, which is then used as a JTable model. A new scrollable pane is created using the table and in turn used to create a new tab. Finally the new table is made the currently selected tab. All straight forward and not particularly complicated or problematic.

The problem comes when you want to get the Project object out of the current tab so that you can, for example, call close on it or use it to set the main window title bar. The code below shows one way it can be done:

```
JScrollPane pane = (JScrollPane) tabbedPane.getSelectedComponent();
Component coms[] = pane.getComponents();
JViewport viewPort = (JViewport) coms[0];
coms = viewPort.getComponents();
JTable table = (JTable) coms[0];
RecordGrid grid = (RecordGrid) table.getModel();
project = grid.getProject();
```

It relies on the fact that every object is a component of another object. Sometimes, as shown above, requesting a component's child component returns an array of components and, although this code doesn't show it, the required component needs to be found within the array. This is messy and potentially unreliable. After writing this code I felt there had to be a better way. So I asked some people. Roger Orr came up with a much simpler solution:

```
JTable table = (Jtable)pane.getViewport().getView();
```

Something still didn't feel right though. Tthe code is querying the GUI components to navigate the object relationships. This breaks encapsulation as changing the GUI layout would break this code. There are also other ways, but none of them seemed to be the right solution either.

The Solution

The general consensus of opinion was that I was mad to have a user interface component (the tabs) managing a data component (the project) and that I should be using the mediator version of the Model View Controller (MVC) described above.

I thought I pretty much had how MVC worked nailed down because of the MFC document view model stuff (basically just a model and views) I had done early in my career, but reading up on MVC and Swing in the various books I had just left me confused in terms of implementation. Then I googled and found Java SE Application Design With MVC [JADwMVC] on the Sun website. Suddenly everything was much clearer and I set about knocking up a prototype.

The main concern I had was keeping the tabbed view in sync with the model, so that when I selected a tab the currently selected project in the model was changed to reflect it correctly and when a new project was added to or deleted from the model it was also added or removed from the tabbed view. Should I remove every tab from the tabbed view and redraw when the model was updated or try and add and remove tabs one at a time in line with the model?

I decided to start developing my MVC prototype and cross that bridge when I came to it. The beauty of the mediator MVC patterns is that each component can be worked on and changed individually to a greater or lesser extent. The concerns are well separated.

The Model

The file viewer model:

- Needs to handle multiple projects.
- Needs to have a mechanism for adding projects.
- Needs to have a mechanism for deleting projects.
- Needs to have a mechanism for setting the current project.
- Needs to have a mechanism for getting the current project.
- Should have a mechanism for getting the number of projects for testing purposes.
- Needs to fire events when properties change (e.g. a project is added, deleted or selected).

The controller will have to mediate a number of these operations from views and menus to the model. So both will have similar interfaces for organizing projects within the model. The model interface looks like this:

```
public interface ProjectOrganiser
{
    public abstract void addProject(Project prj);

    public abstract int getProjectCount();

    public abstract void setCurrentProjectIndex(int index);

    public abstract int getCurrentProjectIndex();

    public abstract Project getCurrentProject();

    public abstract void deleteCurrentProject();
}
```

New projects will be created outside of the controller, so the `newProject` method takes a `Project` reference. Fully unit testing a model like this is relatively easy, but beyond the scope of

this article. The other methods perform the other required operations, with the exception of firing events.

Implementing the model is straight forward. I'll go through the data members first and then each method in turn.

```
public class Projects implements ProjectOrganiser
{
        private final List<Project> projects =
                                new ArrayList<Project>();
        private int currentProjectIndex = -1;
        ...
}
```

The model is essentially a container for storing and manipulating projects, so it needs a way of storing the projects. An `ArrayList` is ideal. The model also needs to indicate which project is currently selected. To keep track it stores the `ArrayList` index of the currently selected project. If the `ArrayList` is empty or no project is currently selected then `currentProjectIndex`'s value is `-1`. Initially there are no projects.

The `addProject` method adds the new project to the `ArrayList` and sets it as the current project:

```
public void addProject(Project prj)
{
        projects.add(prj);
        setCurrentProjectIndex(projects.size() - 1);
}
```

The `getProjectCount` method simply asks the `ArrayList` its size and returns it.

```
public int getProjectCount()
{
        return projects.size();
}
```

The `setCurrentProjectIndex` method checks the index it is passed to make sure it is within the permitted range. It can either be `-1` or a valid index within the `ArrayList`. If the index is not valid it constructs an exception message explaining the problem and throws an `IndexOutOfBoundsException`. If the index is valid it is used to set the current project.

```
public void setCurrentProjectIndex( int index )
{
        if (index >= getProjectCount() || index < -1)
        {
                final StringBuilder msg = new StringBuilder();
                msg.append("Set current project to ");
                msg.append(index);
                msg.append(" failed. Project count is ");
                msg.append(getProjectCount());
                msg.append(".");
                throw new IndexOutOfBoundsException
                                        (msg.toString());
        }

        currentProjectIndex = index;
```

```
}
```

The getCurrentProjectIndex simply returns currentProjectIndex.

```
public int getCurrentProjectIndex()
{
        return currentProjectIndex;
}
```

The getCurrentProject method relies on the fact that the setCurrentProjectIndex method has policed the value of currentProjectIndex successfully. Therefore it only checks to make sure currentProjectIndex is greater than or equal to 0. If it is it returns the corresponding project from the ArrayList, otherwise null.

```
public void getCurrentProject()
{
        Project project = null;

        if (getCurrentProjectIndex() >= 0)
        {
                project = projects.get(currentProjectIndex);
        }

        return project;
}
```

The deleteCurrentProject method is by far the most interesting in the model. It is also the most important method to get a unit test around. It checks to make sure there are projects in the ArrayList. If there are then it calls close on and then deletes the current project from the ArrayList and calculates which the next selected project should be. If, following the deletion, another project moves into the same ArrayList index it becomes the next selected project. Otherwise the project at the previous index in the ArrayList is used. If there are no longer any projects in the ArrayList the current index is set to -1.

```
public void deleteCurrentProject()
{
        if (getCurrentProjectIndex() >= 0)
        {
                final int currentIndex
                        = getCurrentProjectIndex();

                getCurrentProject().close();
                projects.remove(currentIndex);

                int nextIndex = -1;
                final int projectCount = getProjectCount();

                if (currentIndex < projectCount)
                {
                        nextIndex = currentIndex;
                }
                else if (projectCount > 0)
                {
                        nextIndex = projectCount - 1;
                }

                setCurrentProjectIndex(nextIndex);
        }
}
```

As you can see, manipulating projects within the model via the `ProjectOrganiser` interface is very straight forward. However, there is currently no way of notifying the controller when a property changes. The Java SE Application Design With MVC article recommends using the Java Beans component called Property Change Support. The `PropertyChangeSupport` class makes it easy to fire and listen for property change events. It allows chaining of listeners, as well as filtering by property name.

To enable the registering of listeners and the firing of events a few changes need to be made to `ProjectOrganiser` and `Projects`. First, event names and a change listener registering method need to be added to `ProjectOrganiser`:

```
public interface ProjectOrganiser
{
	public static final String NEW_PROJECT_EVENT
				= "ProjectOrganiser.NEW_PROJECT";

	public static final String DELETE_PROJECT_EVENT
			= "ProjectOrganiser.DELETE_PROJECT";

	public static final String
		CURRENT_PROJECT_INDEX_CHANGED_EVENT
		= "ProjectOrganiser.CURRENT_PROJECT_INDEX_CHANGED";

	public static final String CURRENT_PROJECT_CHANGED_EVENT
		= "ProjectOrganiser.CURRENT_PROJECT_CHANGED";

	public abstract void addPropertyChangeListener(
				PropertyChangeListener listener);
	...
}
```

Later versions of Java have support for enums. However, property change support does not, but uses strings instead. Enums could be used in conjunction with the `toString` method, but this makes their use overly verbose and does not give any clear advantages.

Then the `Projects` class needs a `PropertyChangeSupport` object and the methods to add listeners and fire events:

```
public class Projects implements ProjectOrganiser
{
	...
	private final PropertyChangeSupport propertyChangeSupport
			= new PropertyChangeSupport(this);

	...

	@Override
	public void addPropertyChangeListener(PropertyChangeListener listener)
	{
		propertyChangeSupport.addPropertyChangeListener(listener);
	}

	private void firePropertyChange(	String propertyName,
				Object oldValue,
				Object newValue)
	{
		propertyChangeSupport.firePropertyChange(	propertyName,
					oldValue,
```

```
                                                        newValue);
        }
}
```

The PropertyChangeSupport object needs to know the source of the events it is firing so this is passed in. The addPropertyChangeListener method simply forwards to the same method of the PropertyChangeSupport object.

Any class that implements the PropertyChangeListener interface can receive events. The firePropertyChange method takes the name of the property that has changed, the property's old value and its new value. All of these, together with the source object, are passed to the event sink as a PropertyChangeEvent. If the old and new values are the same the event is not fired. This can be overcome by setting one or more of the old and new objects to null. We'll look at the implications of this shortly.

The Projects class now has the methods to fire events, but is not actually firing anything. The controller needs to be notified every time a project is created, deleted or selected. This means the addProject, deleteProject and setCurrentProjectIndex methods must be modified:

```
        public void addProject(Project prj)
        {
                Project old = getCurrentProject();
                projects.add(prj);
                firePropertyChange(NEW_PROJECT_EVENT,old,prj);
                setCurrentProjectIndex(projects.size() - 1);
        }
```

The addProject method now stores a reference to the current project prior to the new project being added. The old project and the new project are both passed to the event as properties. This means that the event sink can perform any necessary clean up using the previously selected project and update itself with the new project without having to query the controller. Also, the old project and new project references will never refer to the same project, so the event will never be ignored.

The setCurrentProjectIndex method fires two events. The CURRENT_PROJECT_INDEX_CHANGE_EVENT event is fired when the current project *index* changes and the CURRENT_PROJECT_CHANGE_EVENT is fired when the current project changes. These are deceptively similar events. Consider when a project is deleted. If the project is in the middle of the ArrayList the project in front of it moves into its index. The project changes, but the index stays the same. The CURRENT_PROJECT_INDEX_CHANGE_EVENT event passes both the old and new indexes. The CURRENT_PROJECT_CHANGE_EVENT is only passed the new project. The old value is always null. This is because, following a project deletion the old project no longer exists.

```
        public void setCurrentProjectIndex( int index )
        {
                ...

                final int oldIndex = currentProjectIndex;
                currentProjectIndex = index;
                firePropertyChange(CURRENT_PROJECT_INDEX_CHANGED_EVENT,
                                        oldIndex,
                                        currentProjectIndex);

                firePropertyChange(CURRENT_PROJECT_CHANGED_EVENT,
```

```
                                        null,
                                        getCurrentProject());
        }
```

The `deleteCurrentProject` method requires some significant changes:

```
public void deleteCurrentProject()
{
        if (getCurrentProjectIndex() >= 0)
        {
                final int currentIndex = getCurrentProjectIndex();
                final Project currentProject = getCurrentProject();

                final NextProject nextProject
                        = new NextProject(projects,currentIndex);

                firePropertyChange(DELETE_PROJECT_EVENT,
                                        currentProject,
                                        nextProject.getProject());

                currentProject.close();
                projects.remove(currentIndex);
                setCurrentProjectIndex(nextProject.getIndex());
        }
}
```

Ideally, when a project is deleted the old value and the next project to be selected are be passed as the old and new values. The event needs to be fired before the project is actually removed so that any thing using it can clean up. This can make it difficult to work out which project is which. One way to be sure is to make a copy of the project `ArrayList`, remove the project to be deleted from it and then work out which the next selected project will be. To do this I wrote a helper class called `NextProject`. Overall it is more code, but it makes for a much neater solution to the `deleteCurrentProject` method and means the next project index only needs to be calculated once. Again, the deleted project and the next selected project will never be the same, so the event will not be ignored.

That completes the fully unit testable model. The model is by far the most complex and difficult part of the MVC to implement and get right. A good set of unit tests is essential. Once you have it right the view and controller follow quite easily.

The Controller

The controller is the mediator between the model and the views. Therefore it makes sense to develop it next; otherwise the model and view could end up so incompatible that writing a controller would be very difficult.

The controller maintains a reference to the model and list of references to registered views. If the model changes it passes the event onto the views via the `ProjectOrganiserEventSink` interface.

```
        public interface ProjectOrganiserEventSink
        {
                public abstract void modelPropertyChange(
                                        final PropertyChangeEvent evt);
        }
```

Menu items and registered views all maintain a reference to the controller and use it to pass actions

to the model. Therefore the model has a number of methods that just forward to the model.

The code below shows the `Controller` properties and constructor:

```
public class Controller implements PropertyChangeListener
{
	private final ProjectOrganiser model;
	private List<ProjectOrganiserEventSink> views
				= new ArrayList<ProjectOrganiserEventSink>();

	public Controller(ProjectOrganiser model)
	{
		this.model = model;
		this.model.addPropertyChangeListener(this);
	}
	...
}
```

The Controller implements the `PropertyChangeListener` interface. The `PropertyChangeListener` interface allows the controller to receive events from the model. The `Controller` takes a reference to the model as a constructor parameter and uses it to register itself with the model. Views register themselves via the `addView` method:

```
	public void addView(ProjectOrganiserEventSink view)
	{
		views.add(view);
	}
```

Events are passed on to registered views via the overridden `propertyChange` method from the `PropertyChangeListener` interface:

```
	@Override
	public void propertyChange(PropertyChangeEvent evt)
	{
		for (ProjectOrganiserEventSink view : views)
		{
			view.modelPropertyChange(evt);
		}
	}
```

The model forwarding methods do just that, with the exception of the `newProject` method:

```
	public void deleteCurrentProject()
	{
		model.deleteCurrentProject();
	}

	public Project getCurrentProject()
	{
		return model.getCurrentProject();
	}

	public int getCurrentProjectIndex()
	{
		return model.getCurrentProjectIndex();
	}

	public int getProjectCount()
	{
		return model.getProjectCount();
```

```
    }

    ...

    public void setCurrentProjectIndex(int index)
    {
            model.setCurrentProjectIndex(index);
    }
```

The newProject method is different because it has to create a project to pass to the model. The idea behind the Project interface is that it can be used to reference implementations for different file type and layout type combinations. Therefore I have written the NewProjectDlg class to allow the user to select the type of project they want. It then calls createNew on the project to do some project specific creation. A reference to the project can then be queried and passed to the model:

```
    public void addProject(JFrame owner)
    {
            NewProjectDlg d = new NewProjectDlg(owner,true);
            d.setVisible(true);
            Project prj = d.getProject();
            if (prj != null)
            {
                    model.newProject(prj);
            }
    }
```

The internal workings of the NewProjectDlg class are beyond the scope of this article.

The Controller is almost fully unit testable. The fly in the ointment is of course the NewProjectDlg. You just don't want it popping up in the middle of a test. There are a number of easy ways of getting round it, but they are beyond the scope of this article also. However, Controller is so simple that it hardly requires a unit test. Under normal circumstances I would write one anyway, but it would require some non-trivial mock objects that just do not seem worth it.

The View

Getting a view to receive events from the controller is very simple. It only requires the implementation of the ProjectOrganiserEventSink interface and then registering with the controller. The complexity comes with what you actually do with the view. I'm going to explain two examples. One that just updates the title of the main window when the current project changes and one that keeps tabs in sync with the model (that was the method I decided to try and implement first. It worked as you'll see!).

Main Window View

As I hinted at earlier, I come from an MFC background. Writing GUIs in Java with Swing is therefore an absolute dream by comparison. Instead of having to rely hugely on the wizard and get the linking right, with Swing a window can be created from a main function and a few simple objects.

The main window is a good place to create and reference the controller. In order to receive events from the controller the view must implement the ProjectOrganiserEventSink interface and

register itself with the model:

```
public class MainWindow  extends JFrame
                              implements ProjectOrganiserEventSink
{
          private final Controller controller;

          public MainWindow()
          {
                    controller = new Controller(new Projects());
                    controller.addView(this);

                    this.setDefaultCloseOperation(
                                        JFrame.EXIT_ON_CLOSE );
                    this.setSize(1000, 600);
          }

          @Override
          public void modelPropertyChange(PropertyChangeEvent evt)
          {
                    ...
          }

          public static void main(String[] args)
          {
                    javax.swing.SwingUtilities.invokeLater(new Runnable()
                    {
                              public void run()
                              {
                                        new MainWindow().setVisible(true);
                              }
                    });
          }
}
```

When the current project changes, due to a project being added or deleted or the user selecting a different tab, the description of the project should be updated in the main window's title bar. This is done by handling the project changed event:

```
public class MainWindow  extends JFrame implements ProjectOrganiserEventSink
{
          private static final String TITLE = "File Viewer";

          ...

          @Override
          public void modelPropertyChange(PropertyChangeEvent evt)
          {
                    if (evt.getPropertyName().equals(
                              ProjectOrganiser.CURRENT_PROJECT_CHANGED_EVENT))
                    {
                              StringBuilder builder = new StringBuilder(TITLE);

                              Project prj = (Project) evt.getNewValue();
                              if (prj != null)
                              {
                                        builder.append(" - ");
                                        builder.append(prj.getDataDescription());

                              }
                              setTitle(builder.toString());
                    }
```

```
        }

        ...
}
```

When the view receives the `PropertyChangeEvent` it checks to see what type of event it is. If it is a CURRENT_PROJECT_CHANGED_EVENT it gets the project from the event object. If the current project is `null`, for example if there are no projects in the model, it sets the default title, otherwise it gets a description from the project and concatenates that to the default title.

So far we have a main window that creates and handles events from a controller, but nothing that actually causes an event to be fired. To get events we need to be able to create and delete projects. One of the easiest ways to do this is via a menu. Menus are easy to create and anonymous classes give instant access to the controller.

```
public class MainWindow  extends JFrame implements ProjectOrganiserEventSink
{
        private final Controller controller;

        public MainWindow()
        {
                controller = new Controller(new Projects());
                controller.addView(MainWindow.this);
                buildMenu();
        }

        private void buildMenu()
        {
                JMenu fileMenu = new JMenu("File");
                fileMenu.setMnemonic(KeyEvent.VK_F);

                JMenuItem newItem = new JMenuItem("New");
                newItem.setMnemonic(KeyEvent.VK_N);
                newItem.addActionListener( new ActionListener()
                {
                        @Override
                        public void actionPerformed(ActionEvent e)
                        {
                                controller.newProject(MainWindow.this);
                        }
                });
                fileMenu.add(newItem);

                JMenuItem closeItem = new JMenuItem("Close");
                closeItem.setMnemonic(KeyEvent.VK_C);
                closeItem.addActionListener( new ActionListener()
                {
                        @Override
                        public void actionPerformed(ActionEvent e)
                        {
                                controller.deleteCurrentProject();
                        }
                });

                fileMenu.add(closeItem);

                JMenuBar menuBar = new JMenuBar();
                menuBar.add(fileMenu);
                this.setJMenuBar(menuBar);
        }
```

Projects can now be added to and deleted from the model via the main window's file menu. The main window handles an event from the controller that allows it to set it's title based on the currently selected project.

Tabbed View

Projects are not much use if they cannot be viewed or selected, so what is needed is a tabbed view capable of displaying projects:

```
public class DataPane extends JTabbedPane
                    implements ProjectOrganiserEventSink
{
        private final Controller controller;

        public DataPane(Controller controller)
        {
                this.controller = controller;
                initialise();
        }

        private void initialise()
        {
                controller.addView(this);
        }

        @Override
        public void modelPropertyChange(PropertyChangeEvent evt)
        {
                ...
        }
}
```

Swing has the `JTabbedPane` class that will do the job perfectly. Subclassing, as shown above, gives a better level of encapsulation and control of the view's functionality. The view must also implement the `ProjectOrganiserEventSink,` maintain a reference to the controller, which must be passed into the constructor, and register itself with the controller. In order to be seen and used the `DataPane` also needs to be added to the main window:

```
public class MainWindow  extends JFrame implements ProjectOrganiserEventSink
{
        ...

        public MainWindow()
        {
                ...
                this.add(new DataPane(controller));
                this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
                this.setSize(1000, 600);
        }
        ...
}
```

As well as responding to events fired by the controller, the view also needs to notify the controller when a user has selected a different tab. This is done by writing a change listener:

```
        public void initialise()
        {
                controller.addView(this);
```

```
            addChangeListener(new ChangeListener()
            {
                  @Override
                  public void stateChanged(ChangeEvent arg0)
                  {
                        controller.setCurrentProjectIndex(
                                          getSelectedIndex());
                  }

            });
      }
```

Every time the tabbed control's state changes, for example the user selects a different tab, the change listener's `stateChanged` method is called. As the change listener is implemented as an anonymous class within `DataPane` it has access to `DataPane`'s properties. Therefore it can query the tabbed view for the current index and pass it to the controller.

The `modelPropertyChange` override handles events from the controller:

```
      @Override
      public void modelPropertyChange(PropertyChangeEvent evt)
      {
            if (evt.getPropertyName().equals(
                  ProjectOrganiser.NEW_PROJECT_EVENT))
            {
                  addProject((Project)evt.getNewValue());
            }
            else if(evt.getPropertyName().equals(
                  ProjectOrganiser.CURRENT_PROJECT_INDEX_CHANGED_EVENT))
            {
                  setSelectedIndex((Integer) evt.getNewValue());
            }
            else if(evt.getPropertyName().equals(
                   ProjectOrganiser.DELETE_PROJECT_EVENT))
            {
                  deleteCurrentTab();
            }
      }
```

As before, the event's name is used to determine what sort of event it is. The `DataPane` handles the new project, project deleted and project index change events, passing where appropriate, the event's new value to another method for handling. The new value is a new project that has been created, the newly selected project following a project deletion, or the new index following a newly created or deleted project. `setSelectedIndex` is a method inherited via `JTabbedPane` and does exactly what you would expect. The `addProject` method is implemented as follows, with exception handling omitted for clarity:

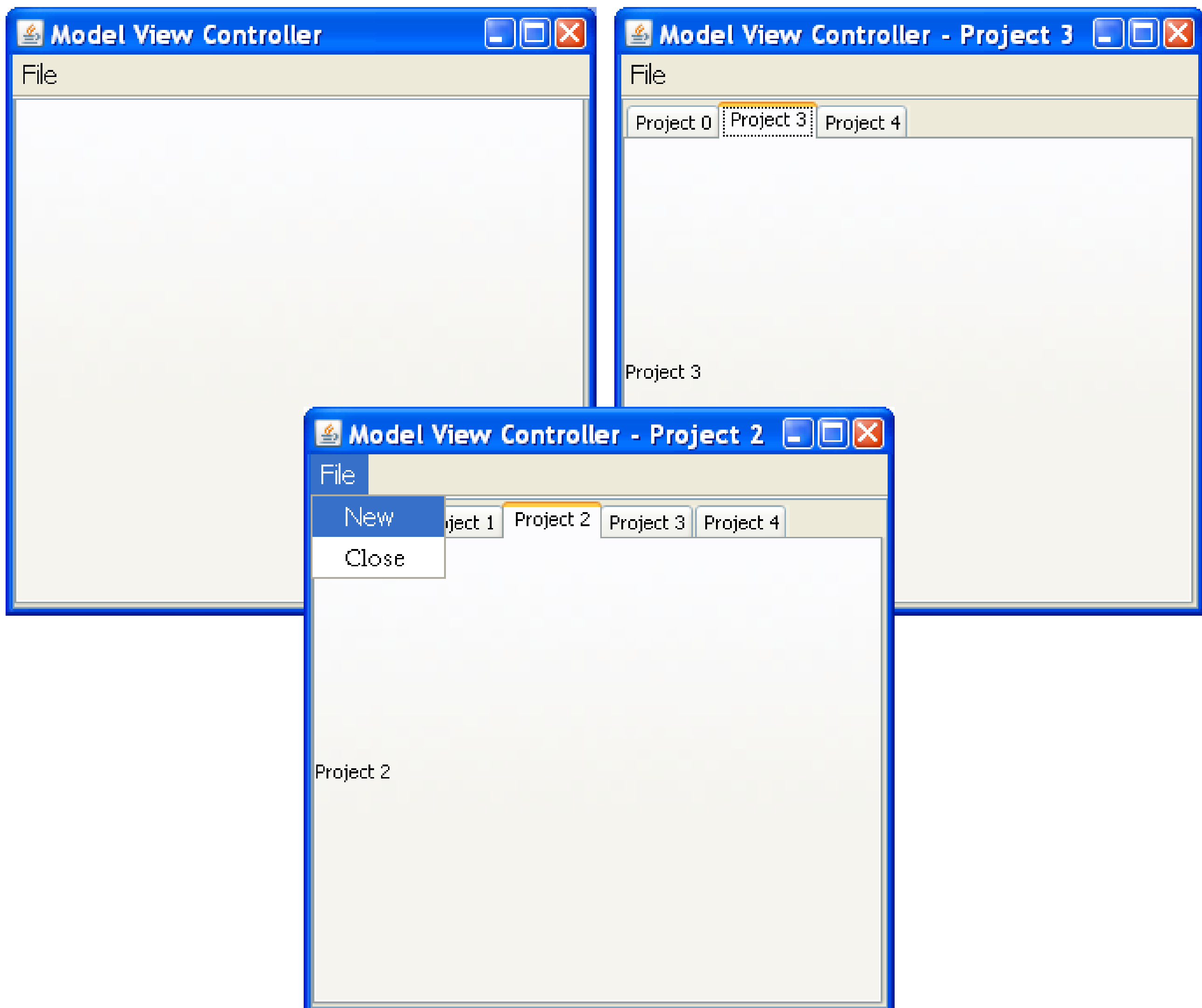
```
      private void addProject(Project project)
      {
            if (project != null)
            {
                  RecordReader recordReader =
                              project.getRecordReader();

                  TableModel model = new RecordGrid(project);
                  JTable table = new JTable(model);
                  this.addTab(recordReader.getDataDescription(),
                                    new JscrollPane(table));
            }
      }
```

The code should look familiar. It is almost identical to part of the code in The Problem section above. The only difference is that the newly added tab is not selected as the current tab. That is now set following a project index changed event.

```
private void deleteCurrentTab()
{
        if (getTabCount() > 0)
        {
                remove(this.getSelectedIndex());
        }
}
```

The `deleteCurrentTab` method checks to make sure there is at least one tab to delete, gets the index of the current tab and deletes it.

That completes the implementation of the `DataPane` view. Projects can now be added to and deleted from the application, tabs can be changed and the changes reflected in the main window title and the client area tabbed view:



Conclusion

By implementing MVC and demonstrating how easy it is to manipulate, control and display the projects, I believe I have demonstrated the advantages of MVC over the original design I had, where the model was effectively buried in the view and difficult to get hold of when needed.

The mediator pattern keeps the model and views nicely decoupled. There is slight coupling of the view to the `ProjectOrganiser` interface as the event names are part of the interface. If this became an issue it would be simple to move the event names to their own class. I believe this is unnecessary at this stage.

I was also concerned about keeping the projects in the model and the associated tab in the view in sync. However, this problem was easily overcome:

- By relying on the fact that new tabs are always added at the end of the tabbed view and new projects are always added at the end of the model `ArrayList`.
- Using an event from the model to set the currently selected project in the tabbed view.
- Using an event from the model to remove the tabs for deleted projects from the tabbed view.

Finally I'll leave you with a comment from a colleague of mine:

> “I really like the idea of Model View Controller, but it takes so much code to do very little.”

I think I've shown here that there is quite a lot of code involved in the MVC design compared to the model within the view design, but the separation and the ease with which the data can be manipulated is a clear advantage.

Acknowledgments

Thanks to Roger Orr, Tom Hawtin, Kevlin Henney, Russel Winder and Jez Higgins for general guidance advice and patience and to Caroline Hargreaves for review.

References

[PEAA] Patterns of Enterprise Application Architecture by Martin Fowler. ISBN-13: 978-0321127426

[MVC Antipattern] The M/VC Antipattern
http://www.benjaminbooth.com/tableorbooth/2005/04/m_c_v.html

[JADwMVC] Java SE Application Design With MVC
http://java.sun.com/developer/technicalArticles/javase/mvc/