Assignment 2 for COMPSCI 230

Submission deadline: 4pm Friday, August 29. (End of 6^{th} week)

Version 1.1

Assignment Information

Weighting. This assignment has 40 marks and is worth 4% of your total result for COMPSCI 230.

Length. *This assignment is longer than the previous assignment.* You should read the entire assignment before you start working on the first question, so that you can allocate your time appropriately, and so that you understand how each question will prepare you for the next one.

Learning goals. Completing this assignment will give you experience in aspects of GUI design and event-driven GUI programming with Java's Swing framework. You will also gain some experience in working with larger code bases developed by other people.

Working independently. You are **not** allowed to receive assistance from any other person when you are answering questions in this assignment, nor when you are designing and writing your class diagrams and code for this assignment. This must be your own work, done independently. You may gain design ideas or examples from the internet or a textbook. Anyone who gives you debugging assistance should not tell you how to fix your bug; however they might help you learn how to import code into your development environment, or how to use its debugger.

English grammar and spelling: You will not be marked down for grammatical errors, nor for spelling errors in words, as long as the markers are confident they understand your intended meaning. However, if your meaning is not readily apparent to the marker, you will lose some marks.

Submission instructions. Submission will be via the Assignment Drop Box (`https://adb.auckland.ac.nz`). Your completed assignment will consist of a report, which must be submitted as a single (PDF, docx or odt) document. All figures must be included in this document.

Bug reports. If you find an error in this assignment, please send an email to `tgvaughan@gmail.com` with the string "COMPSCI 230 ASSIGNMENT BUG" in the subject.

Question 1 [10 marks]

Read the tutorial on "Swing and the MVC" (appended). Build the StringIntegerTableModel and TestFrame classes, as described in this article, and carefully consider how the ActionListener, TableModelEvent and JFrame classes are being used.

1. Draw a class diagram showing the relationships between StringIntegerTableModel, TableModel, TestFrame, JFrame and TableModelListener. Remember that TableModel and TableModelListener are both interfaces. (You do not need to include methods in this class diagram.)
2. Briefly outline the steps that occur when the "Add" button is clicked until the entry appears on the screen. Do not get too detailed here—there should only be around 6 steps in your list. (Hint: The first two entries on your list might be 1. User clicks button, 2. button fires ActionEvent to its listeners, ...)
3. In two or three sentences, describe how the inversion of control principle applies between the application code you put together from the tutorial and the Swing framework.

Your report should include: Your class diagram from Q1.1, your sequence of steps from Q1.2 and your discussion of IOC from Q1.3. You do *not* need to include any code here.

Expected completion time: 2 hours: 1 hour for the tutorial/programming, 1 hour for your analysis and write-up.

Question 2 [5 marks]

Download `SpaceInvaders.jar` from
https://github.com/tgvaughan/SpaceInvaders/releases/latest
and confirm that you can run this application on your computer.

1. Play the game a few times and interact with the items in the menus. Briefly describe how the game's UI follows (or does not follow) the following interface design principles:

   User familiarity: The interface should use terms and concepts which are drawn from the experience of the people who will make use of the system.

   Minimal surprise: Users should never be surprised by the behaviour of the system. The user should be able to predict the outcome of each command from its documentation.

User diversity: Interaction facilities for different types of user (eg. different levels of experience) should be supported.

You should aim to write only one or two sentences per rule. Pay particular attention to the menus and keyboard shortcuts for different commands.

2. Consider the problem of using the code from the previous question to add high-score table to the game. On the UI side, this will require:

   - the addition of a small input dialog (e.g. the last example on the Java tutorial page) following the "game over" dialog box to collect the player's name,
   - a menu item through which players can access the high scores table, and
   - the high score table itself, which will have exactly the same layout as the TestFrame class, but with a single "Ok" button at the bottom of the frame in place of the existing text fields and "Add" button.

   Construct a simple screen flow diagram modelling the process of getting a new entry into the high score table and accessing the high score table itself. (It should be possible to determine the functionality of the completed program from the diagram alone.)

Your report should include: Your discussion of the design principles from Q2.1 and your screen flow diagram from Q2.2. (You are welcome to submit a scan of a hand-drawn diagram for Q2.2 in your report, but it must be legible.)

Expected completion time: 1 hour

Question 3 [15 marks]

Download the Java source code for the Space Invaders game from its Github repository. The easiest way to do this is to download the zip file from `https://github.com/tgvaughan/SpaceInvaders/archive/master.zip` and extract into a new folder. (If you're feeling adventurous, or are already familiar with Git and Github, cloning the repository which is found here is also an option.)

Once you have obtained the source, create a new Eclipse project and build the game. The main class is SpaceInvadersApp. Feel free to ask your tutors or classmates for help with this step.

1. Study the classes present in the package `spaceinvaders`. Construct a class diagram representing their relationships. (Don't include methods or fields in this diagram.)

2. Create a new package named `spaceinvaders.highscores` and to it add a copy of the StringIntegerTableModel and TestFrame classes you assembled in the MVC tutorial. Rename them HighScoreTableModel and HighScoreDialog respectively.

3. Read the Java tutorial page "Using Top-Level Containers" found at http://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html. Note that JDialog is an alternative top-level container to JFrame that is intended for displaying dialog boxes. Modify HighScoreDialog so that it extends JDialog, and remove its main() method. (You will also need to remove the call to setDefaultCloseOperation(), as JDialog does not allow EXIT_ON_CLOSE.)

4. Replace the bottom JPanel containing the text fields and the "Add" button with a single "Ok" button. Give this button an anonymous ActionListener that calls `setVisible(false)`.

5. Modify the HighScoreDialog constructor so that it takes a single HighScoreTableModel argument. Modify the rest of the method to use this model rather than creating a new one.

6. Add a new private field to the SpaceInvadersApp class of type HighScoreTableModel, and initialise this field at the beginning of the SpaceInvadersApp constructor. Use appropriate names for the table header, e.g. "Name" and "Score".

7. Add a new menu item to the Game menu that displays the high score dialog by creating a new HighScoreDialog instance (passing the HighScoreTableModel object to the constructor) and calling its setVisible(true) method.

Your report should include: The class diagram from Q3.1, a *full* listing of your completed HighScoreDialog and a *partial* listing of your modified SpaceInvadersApp *showing only the additional code you have introduced.*

Expected completion time: 2–3 hours

Question 4 [10 marks]

Read the Java tutorial page "How to Make Dialogs" found at http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html.

1. Within SpaceInvadersApp.gameEnded(), use JOptionPane.showInputDialog() to display a dialog following the initial Game Over message dialog to users who have won the game with a score greater than zero. (Refer to the existing code in the gameEnded() method to see how to access this information.) The new dialog should inform the user that they have achieved a high score and request their name for the high score table.

2. Use the HighScoreTableModel.addRow() method to add the name and score to the table model. (Make sure you do something sensible when the user presses "Cancel" instead of "Ok"!)

3. Test your implementation to make sure it works! Complete the following table by stating what you *expect* to see if your program is functioning correctly and what you actually *observe* when you carry out the test.

   Test Case 1

   Test procedure : User clicks Cancel button when asked to add name to high score table.

   Expected behaviour : ?

   Observed behaviour : ?

   Design another (equally simple) test of your own and again complete the table by stating what you expect to see and what you observe:

   Test Case 2

   Test procedure : ?

   Expected behaviour : ?

   Observed behaviour : ?

   You may include multiple sentences for each entry in the table, but your expectations and observations should be concise. Note that even if your program does not function as expected, you will still receive marks for good testing.

Your report should include: A *partial* listing of SpaceInvadersApp consisting of only the modifications you made to the gameEnded() method as part of Q4.1 and Q4.2, as well as the completed test case descriptions and results from Q4.3.

Expected completion time: 2 hours: 1.5 hours for the programming, 0.5 hours for the test case design and evaluation.

Tutorial: Swing and the MVC

(Adapted from http://compsci.ca/v3/viewtopic.php?t=11199.)

Introduction

This tutorial is designed to help the Swing0 programmer better understand the Swing framework, by introducing the programmer to the foundation of Swing and many other user interaction framworks, i.e. the Model View Controller architecture – MVC.

Before I continue, I recommend you to use the online javadocs throughout this tutorial. I've quoted them wherever necessary, but I strongly suggest your use the online docs, as it will help you get a better understanding on how-things-work.

MVC is a philosophy about the separation of the three major concerns when dealing with user interaction: the model, the view and the controller. This separation, leads to better maintainability, scalabitlity, reusability and ease-of-development.

The View

So what is the view? Well, this is easy to answer. Anything you see on your screen, is basically your View. So a JButton is a View, a JTextField is a View, a JTable is also View, etc. Basically anything the user can finally see or interact with, is your View.

The Model

This may not be so easy to answer, but a Model is basically the underlying data of your program. It is the collective state of your program that is essential to providing the view with useful information (the data relevant to the purpose of your program). The model is usually (but not necessarily) serializable. This means that the Model can be saved as a file, into a database, or is transferable over the network.

The Controller

It is the means by which the View can communicate with the Model and visa-versa. Any change in the View that needs to update the model, will be carried out through the controller. Any changes in the model that need to notify the View of an update, is also carried out through the controller. It is within the controller that your decision logic is placed. Hence the controller, controls the flow of information within your program.

MVC by example

It is said that the MVC in Swing is really MV/C. What this means it that most Swing components (Views) are pre-coupled with a default Model. The programmer only needs to add a Controller (EventListener) to it, to get it to a workable state.

While this is true for most cases, but usually only for simpler components, like JButton etc. The more complex components, like JTable, JTree or JList, require you to provide them with your own custom Model to get them to a fully functional state.

For the sake of keeping things simple, we're going to look into the just one of these components: JTable. In our first example we are going to create a simple two-column JTable that is dynamically updatable. We will create our custom Model to do this, but we will not make is Serializable, since this isin't necessary.

The TableModel Interface

Every JTable Model *must* implement the TableModel interface. First let's take a look at the javadocs to find out what this interface declares:

Method Summary

Methods

Modifier and Type	Method and Description
void	**addTableModelListener**(TableModelListener l) Adds a listener to the list that is notified each time a change to the data model occurs.
Class<?>	**getColumnClass**(int columnIndex) Returns the most specific superclass for all the cell values in the column.
int	**getColumnCount**() Returns the number of columns in the model.
String	**getColumnName**(int columnIndex) Returns the name of the column at columnIndex.
int	**getRowCount**() Returns the number of rows in the model.
Object	**getValueAt**(int rowIndex, int columnIndex) Returns the value for the cell at columnIndex and rowIndex.
boolean	**isCellEditable**(int rowIndex, int columnIndex) Returns true if the cell at rowIndex and columnIndex is editable.
void	**removeTableModelListener**(TableModelListener l) Removes a listener from the list that is notified each time a change to the data model occurs.
void	**setValueAt**(Object aValue, int rowIndex, int columnIndex) Sets the value in the cell at columnIndex and rowIndex to aValue.

Let's create a new class StringIntegerTableModel that implements this interface to represent a table with exactly two columns: the first of Strings and the second

of Integers. We want this model to allow for the dynamic addition and removal of rows from the table. Note that we need to implement the facility to add Controllers which need to be notified when the model changes.

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

public class StringIntegerTableModel implements TableModel {

    private List<String> col1StringList = new ArrayList<String>();
    private List<Integer> col2IntegerList = new ArrayList<Integer>();
    private String col1Name, col2Name;

    private List<TableModelListener> listenerList = new ArrayList<TableModelListener>();

    public StringIntegerTableModel(String col1Name, String col2Name) {
        this.col1Name = col1Name;
        this.col2Name = col2Name;
    }

    public int getRowCount() {
        return col1StringList.size();
    }

    public int getColumnCount() {
        return 2;
    }

    public String getColumnName(int columnIndex) {
        switch (columnIndex) {
            case 0:
                return col1Name;
            case 1:
                return col2Name;
            default:
                return null;
        }
    }

    public Class<?> getColumnClass(int columnIndex) {
        switch (columnIndex) {
            case 0:
                return String.class;
            case 1:
                return Integer.class;
            default:
                return null;
        }
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        if (rowIndex>=getRowCount() || rowIndex<0)
```

```
            return null;

        switch (columnIndex) {
            case 0:
                return col1StringList.get(rowIndex);
            case 1:
                return col2IntegerList.get(rowIndex);
            default:
                return null;
        }
    }

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return false;
    }

    public void setValueAt(Object arg0, int arg1, int arg2) {
        throw new UnsupportedOperationException("Not supported.");
    }

    public void addTableModelListener(TableModelListener l) {
        listenerList.add(l);
    }

    public void removeTableModelListener(TableModelListener l) {
        listenerList.remove(l);
    }
}
```

Here we have implemented the bare minimum. We implemented adding/removing of TableModelListeners, the ability to retrieve data from the model, as well as the ability to inquire the size of the table model. This is *almost* everything needed to get our TableModel functional, but not quite!

Remember that we were going to dynamically add rows to the JTable? Well the model needs to reflect that, so we need to move beyond the bare minimum. Let's implement an "addRow" method:

```
    public void addRow(String name, Integer value) {
        col1StringList.add(name);
        col2IntegerList.add(value);
    }
```

Let's also implement "removeRow" and "update" methods as well:

```
    public void removeRow() {
        int lastRow = getRowCount()-1;
        if (lastRow>0) {
            col1StringList.remove(lastRow);
            col2IntegerList.remove(lastRow);
        }
```

```
    }

    public void update() {
    }
```

We'll flesh out the "update" method later, but the rest should work shouldn't it?

No. As mentioned before, we need to notify the View when the Model changes. At the moment our TableModel does not do that. This is done through interacting with the Controllers. *Never modify the View directy!*

So how do we proceed? Well, it's rather simple really. First we look up the javadocs for TableModelListener class, since it is the controller that our Model needs to interact with. Here's what it has to say about it:

Method Summary

Methods

Modifier and Type	Method and Description
void	`tableChanged(TableModelEvent e)` This fine grain notification tells listeners the exact range of cells, rows, or columns that changed.

Here we see only a single method to be *called* from our model. What's also interesting is that we need to *pass* these methods a TableModelEvent object. Let's look up the javadocs for this:

Constructor Summary

Constructors

Constructor and Description
`TableModelEvent(TableModel source)` All row data in the table has changed, listeners should discard any state that was based on the rows and requery the TableModel to get the new row count and all the appropriate values.
`TableModelEvent(TableModel source, int row)` This row of data has been updated.
`TableModelEvent(TableModel source, int firstRow, int lastRow)` The data in rows [*firstRow*, *lastRow*] have been updated.
`TableModelEvent(TableModel source, int firstRow, int lastRow, int column)` The cells in column *column* in the range [*firstRow*, *lastRow*] have been updated.
`TableModelEvent(TableModel source, int firstRow, int lastRow, int column, int type)` The cells from (firstRow, column) to (lastRow, column) have been changed.

It seems that these constructors are all we actually need. (Note how useful the javadocs are. They tell us much of what we need to know to go about our work in Java.)

On a side note, since it's not apparent here, but it is if you are reading the docs directly: the int type can be any of the values defined by the static TableModelEvent fields INSERT, UPDATE or DELETE.

Now that we know what we need to do and how to do it, we will write three new private methods for the three respective events:

```
private void fireContentsChangedEvent() {
    TableModelEvent tme = new TableModelEvent(this);

    for (TableModelListener l : listenerList)
        l.tableChanged(tme);
}

private void fireRowAddedEvent() {
    TableModelEvent tme = new TableModelEvent(this,
            getRowCount()-1,
            getRowCount()-1,
            TableModelEvent.ALL_COLUMNS,
            TableModelEvent.INSERT);

    for (TableModelListener l : listenerList)
        l.tableChanged(tme);
}

private void fireRowRemovedEvent() {
    TableModelEvent tme = new TableModelEvent(this,
            getRowCount(),
            getRowCount(),
            TableModelEvent.ALL_COLUMNS,
            TableModelEvent.DELETE);

    for (TableModelListener l : listenerList)
        l.tableChanged(tme);
}
```

Well, I hope what I've written there is pretty self-explanatory, but basically what we do is send each TableModelListener the TableModelEvent object describing the event referred to in the method name. Simple as that.

Now we need to call these methods at the appropriate times. To do that we need to modify our mutative methods to something like this:

```
public void addRow(String name, Integer value) {
    col1StringList.add(name);
    col2IntegerList.add(value);

    fireRowAddedEvent();
```

```
    }

    public void removeRow() {
        int lastRow = getRowCount()-1;
        if (lastRow>0) {
            col1StringList.remove(lastRow);
            col2IntegerList.remove(lastRow);
        }

        fireRowRemovedEvent();
    }

    public void update() {
        fireContentsChangedEvent();
    }
```

Now we have a functional TableModel! Let's try it out by creating a JFrame and adding a JTable based on our model, a pair of JTextFields and a JButton to it:

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

public class TestFrame extends JFrame {

    public TestFrame() {
        setTitle("Test Application");
        setPreferredSize(new Dimension(200, 400));
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container cp = getContentPane();

        final StringIntegerTableModel tableModel =
            new StringIntegerTableModel("String", "Integer");
        JTable table = new JTable(tableModel);
        cp.add(table.getTableHeader(), BorderLayout.NORTH);
        cp.add(table, BorderLayout.CENTER);

        final JTextField nameField = new JTextField("A string");
        final JTextField scoreField = new JTextField("0");
        JButton button = new JButton("Add");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                tableModel.addRow(nameField.getText(),
                        Integer.valueOf(scoreField.getText()));
            }
        });

        JPanel panel = new JPanel(new GridLayout(1, 3));
        panel.add(nameField);
        panel.add(scoreField);
        panel.add(button);
```

```
        cp.add(panel, BorderLayout.SOUTH);

        pack();
    }

    public static void main (String[] args) {
        (new TestFrame()).setVisible(true);
    }
}
```

Although we did not actually make any use of the "removeRow" and "update" methods, you can make use of them in similar ways.

Once again notice how the actual interaction is done only through Controllers. Swing is designed from the ground up using the MVC architecture. In fact, JButton and JTextField both have their own Models, but for the most part the default one is usually the best choice. This is obviously not always the case, since JTable required a more flexible Model (StringIntegerTableModel) to make it do something useful.