Object Oriented Programming

Practical Assignment 2

Boris Motik, University of Oxford

1 Preliminaries

The estimated time needed for completion of this practical is three sessions (i.e., six hours). If you are unable to complete the assignment in that time, you will be able to continue working on the assignment on your own and submit the deliverables before the start of the following session.

You should take some time to think about the questions before attending the practical session as this will help you finish the task on time.

It is expected that you know how to use the Java programming environment before attempting this practical. It particular, you need to be able to use an editor, the Java compiler, and the Java run-time engine. If you need to refresh your memory of these things, consult any of the Java books available in your library.

You are encouraged to use an Integrated Development Environment for the development of your assignment. Eclipse (http://www.eclipse.org/) is currently a de facto industry standard for Java development, and it will have been preinstalled on all computers in the lab. You are encouraged to get acquainted with Eclipse before attending the session. In order to do so, you might consider installing Eclipse on your computer and writing a small application such as "Hello World". A number of Eclipse tutorials are available online; the one available at http://eclipsetutorial.sourceforge.net/ seems to be quite good.

2 Starting Point

The starting point for this practical is the version of the code accompanying Lecture 8. The assignment of this practical is largely concentrated on the part of code representing the domain model (the `layoutapp.model` package), which is summarized below.

The sample code implements the skeleton of an application for visual layout of pages in publications. The domain model of the application is contained in the `layoutapp.model` package, and is schematically shown in Figure 1. The main components of the domain model are as follows:

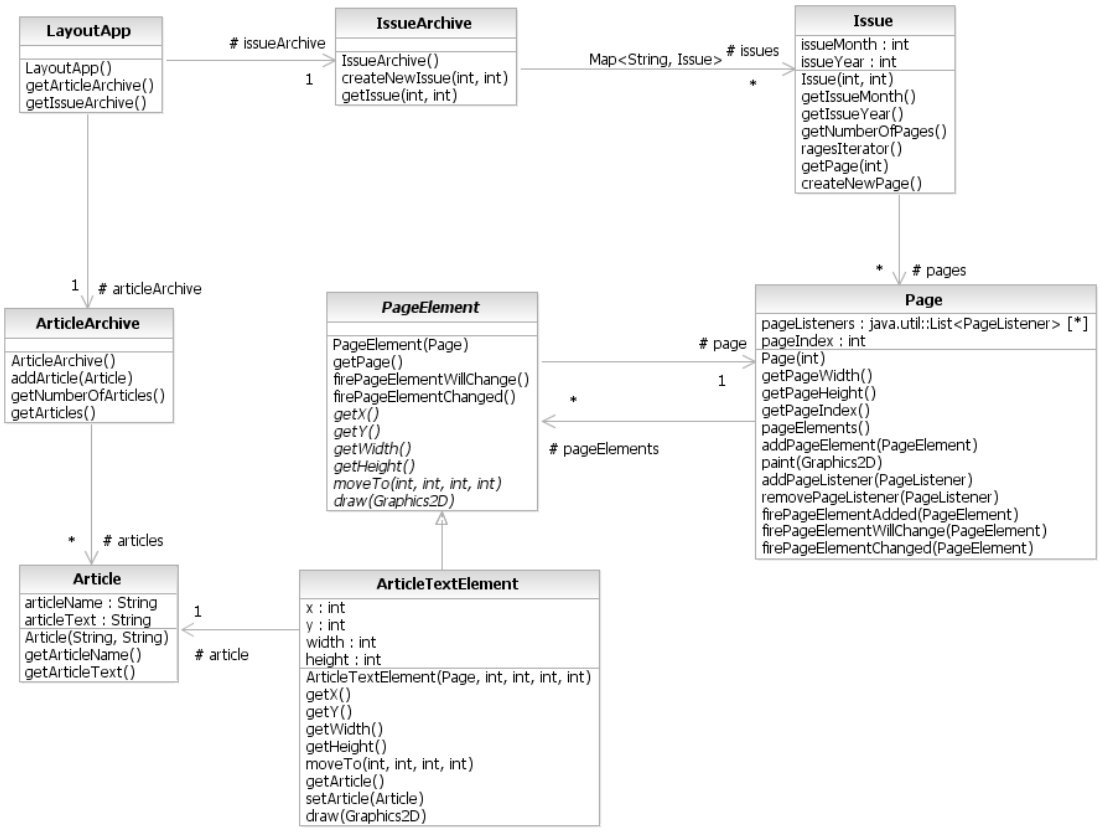

Figure 1. The Domain Model of the Sample Application

- `LayoutApp` represents the application and simply acts as a holder for an `IssueArchive` and an `ArticleArchive`.
- The main object in the application is `Issue`, which represents a concrete issue of the magazine being published. Different issues of the same magazine are stored in an `IssueArchive`. Thus, the collection of all issues of the "Scientific American" magazine would be stored in the `IssueArchive`, whereas the October 2008 issue of the magazine would be represented by an instance of the `Issue` class.
- Each `Issue` consists of a list of pages, and the latter are modeled by the `Page` class. Do not worry for the moment about the `PageListener` class (not shown in the diagram) and the `pageListeners` field of the `Page` class: the purpose of these is explained in detail in Lecture 8.
- A page consists of a set of page elements. The root of the element hierarchy is represented by the `PageElement` class. Currently, `ArticleTextElement` is the only kind of page element; however, new elements are introduced in the following lectures.
- An `ArticleTextElement` represents a region of page containing the text of an article. The text is not shown in the current application; this feature is imple-

mented only in Lecture 11. For the moment, article text elements are represented simply as rectangles.

The starting point of the application is the `ApplicationFrame` class in the `layout` package. This class represents the main application window (which is called in Java a frame). An `ApplicationFrame` is initialized with an issue in the constructor, and it then constructs the graphical user interface for the application. This is also done in the constructor of `ApplicationFrame` by creating a toolbar and an instance of a `PageView`. The latter is a class that implements the main area of the frame and shows the layout of a particular page.

3 Missing Functionality

The sample application currently allows users to work only on the October 2008 issue, which is clearly inadequate. A more realistic application would maintain a database of all issues, which would be persisted between application runs. Upon start, such an application would allow the use to choose an issue he wants to work on. Furthermore, the application would allow users to switch between different issues while the application is running: having to exit the application and then reopen it in order to work with a different issue is clearly suboptimal.

4 The List of Assignments

You must extend the application from Lecture 8 with the functionality outlined in Section 3. In particular, you are required to implement the following functionality:

- You must extend the application such that, when it starts, it maintains not just one issue, but a database of issues. This database does not been to be persistent; that is, it is acceptable for the database to be lost once the application terminates. In order to demonstrate that a database of issues exists, you should populate the issue database with several "dummy" hardcoded issues upon start.
- You must extend the application such that, when it starts, it maintains a database of articles. Again, this database need not be persistent: it can be hardcoded into the application.
- You must extend the application such that it allows the user to choose the issue from the database he wants to work with. Furthermore, the application should also allow the user to switch to a different issue in the database after the application has already started.
- (**optional feature**) You should extend the application such that it allows the user to create new issues. In doing so, use the model-view-controller pattern.

You should also provide a short report (half a page suffices) in which you will provide answers to certain questions outlined below.

Implementing the first three features is required in order to attain the S grade; implementing the optional feature in addition is required in order to attain the S+ grade. The rest of this document provides more details about each of these features.

5 Detailed Instructions

5.1 Maintaining Database of Issues and Articles

The classes `IssueArchive` and `ArticleArchive` from the domain model of the sample application implement the databases of issues and articles, respectively. Furthermore, the class `LayoutApp` acts as a holder for the two archives. To realize the first two requirements, you should ensure that an instance of `LayoutApp` gets created once the application starts, and that instances of `IssueArchive` and `ArticleArchive` are properly initialized as well. The latter step (i.e., the initialization of `IssueArchive` and `ArticleArchive`) should include creating a few dummy issues and articles. Feel free to invent data for these as you see fit.

An important question is where this initialization is done. Creating the domain model is part of application startup, so it may make sense to place this code into the `main()` method of the `ApplicationFrame` class.

Another important question is what to do with the instance of `LayoutApp` once it is created. Since this object provides the foundation for the application, it is likely that most other parts of the application will need access to it. For example, the part of the application that allows the user to choose the issue will need access to the `IssueArchive`. To enable this, a reference to `LayoutApp` should be stored somewhere in the application such that it is easy to access it from other parts of the application.

You should decide where you want to store this instance and explain the rationale for your choice in your report.

5.2 Allowing the User to Choose an Issue

In order to allow the user to choose an issue he wants to work with, you will need to use Java's Swing library for building graphical user interfaces. The rest of this document contains a rough description of the classes you should familiarize yourself with. This description, however, is intentionally not exhaustive: as a software engineer, you will be frequently required to acquire knowledge of new APIs and tools by yourself, and preparing you for such situations is one of the main objectives of this practical. **You are strongly advised to look at these classes before attending the first session of the practical.**

Java has a really good tutorial on Swing available online.[1] The entire tutorial is quite extensive, and you are not expected to cover all of it. You might want to take a look at the "Getting Started with Swing" lesson; furthermore, you will be pointed to specific parts of the "Using Swing Components" section of the tutorial.

5.2.1 Opening a Choice Dialog Box

In order to let the user choose an issue, the application should open a so-called *modal dialog*. Dialogs are windows that typically contain a bunch of options and the "OK" and "Cancel" buttons. The term "modal" refers to the fact that, while the dialog is open, I

[1] http://java.sun.com/docs/books/tutorial/uiswing/index.html

covers the underlying application window and prevents the user from interacting with it. In order to continue working with an application, the user must fill in the options and click "OK" or "Cancel"; only then can he again access the actual application.

In Swing, dialogs are top-level containers, which means that they need not be contained in other components. Such containers are described in the "Using Top-Level Containers" part[2] of the Swing tutorial, and you should go through it.

Your first step towards completing this task is modifying the application such that it opens a dummy dialog box containing an empty table and the "OK" and "Cancel" buttons. To accomplish that, you need to do the following.

You need to create a new class that will represent the dialog presenting the user with the choice of issues. Think about how you will name this new class; in the rest of this document, this class will be called `X`. The class `X` should subclass the `javax.swing.JDialog` class. The constructor of the class should receive a reference to the `ApplicationFrame` (the frame is said to be the *owner* of the dialog). The constructor of `X` should call the `JDialog(Dialog owner, String title, boolean modal)` constructor of `JDialog`, passing `true` for the parameter `modal`. The latter means that the dialog is modal—that is, the owner is disabled while the dialog is shown.

Class `X` should contain code for populating the dialog with controls such as buttons. Think about where to include this code in `X`. Hint: populating an instance of `X` with controls can naturally be seen as a part of instance's initialization, and OO languages provide a natural place to put such code.

In order to position controls within dialog boxes, Swing uses the notion of containers and layout managers. Most Swing controls are containers, which means that they can contain other controls. A most commonly used container is a panel, represented by an instance of the `JPanel` class. By using so-called layout managers, containers can arrange the controls in a desired layout. The "Laying Out Components Within a Container" part[3] of the Swing tutorial explains in detail how to arrange components in containers. This part is rather long, but you should be able to understand the basic ideas quickly.

The general principle for populating a dialog with controls is the following. You first divide the surface of your windows into rectangular areas. Each such area is represented by a `JPanel`. The layout of elements within the area is controlled by a layout manager, which must subclass the `LayoutManager` class. There are different types of layout managers. For example, the `FlowLayout` layout manager arranges its components in a horizontal or vertical sequence. Similarly, the `BorderLayout` layout manager divides its space into five different regions called `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`. You can see the effect of all these layouts in the "A Visual Guide to Layout Managers" part[4] of the Swing tutorial.

---

[2] http://java.sun.com/docs/books/tutorial/uiswing/components/toplevel.html

[3] http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html

[4] http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html

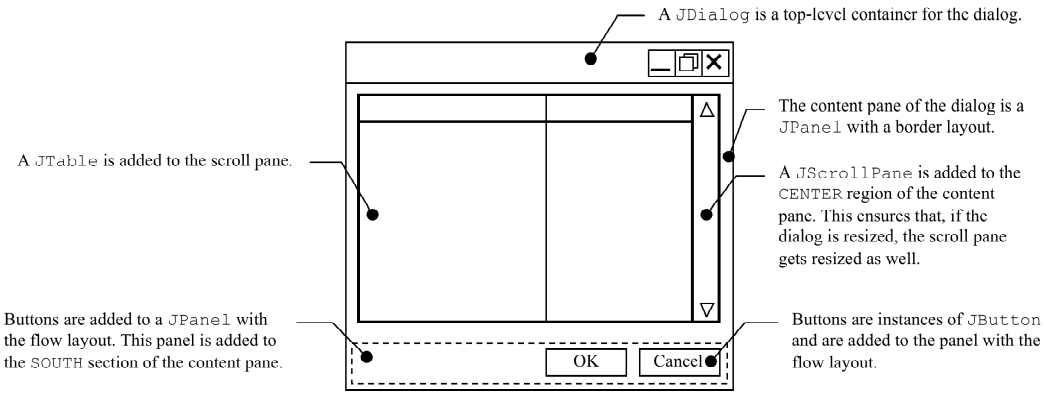

Figure 2. The Layout of the Dialog Box

Once you know how to break up the entire surface of a dialog into nested rectangles, you then create a tree of `JPanel` instances. You initialize each panel with the appropriate layout manager, and you add to its contents the child components. At the end of this process, you will have a panel representing the entire surface of the dialog, which you should then designate as a content pane of the dialog.

Your ultimate goal is to populate the dialog as schematically shown in Figure 2. The dialog should have a content pane which is a `JPanel` with a `BorderLayout`. The `CENTER` region of the content pane should contain a `JScrollPane`—a component that implements scrolling of another component. Using a `JScrollPane` is really simple and does not require work other than creating the component in the appropriate way; however, if you need more information, take a look at the "How to Use Scroll Panes" part[5] of the Swing tutorial. The scroll pane should contain a `JTable`—a component that will ultimately contain the list of issues. For the moment, simply create a `JTable` with two rows and columns and do not worry about populating it with issues. Creating buttons should be straightforward as long as you use the `JButton` class; if you need further help, take a look at the "How to Use Buttons, Check Boxes, and Radio Buttons" part[6] of the Swing tutorial. You can also take a look at the `ApplicationFrame` class to see how buttons are used. The buttons of the dialog should be added to another `JPanel` with a `FlowLayout`. This panel should be added to the `SOUTH` region of the content pane.

To ensure that the components are nicely spaced apart, you can add empty borders to various components; refer to "How to Use Borders" part[7] of the Swing tutorial.

You next need to ensure that the dialog has the proper size. To do so, simply set a preferred size for the scroll pane like this:

```
JScrollPane tableScroller = new JScrollPane(...);
tableScroller.setPreferredSize(new Dimension(400, 300));
```

---

[5] http://java.sun.com/docs/books/tutorial/uiswing/components/scrollpane.html

[6] http://java.sun.com/docs/books/tutorial/uiswing/components/button.html

[7] http://java.sun.com/docs/books/tutorial/uiswing/components/border.html

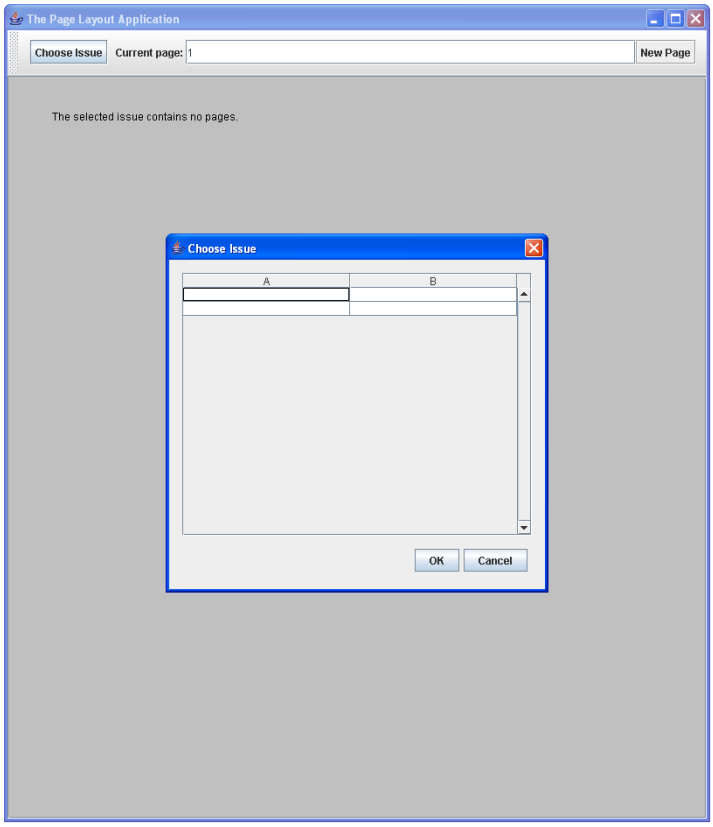

Figure 3. Application Screenshot

This will ensure that, when the layout managers compute the layout of the various components, they will try to ensure that the scroll pane is at least 400 pixels wide and 300 pixels high.

You need to tell the layout managers to compute the layout of the dialog. You can do this by invoking the `pack()` method in `JDialog`.

Next, you need to determine the position of the dialog box on screen by invoking the `setLocation()` method in `JDialog`. You should center the dialog box within its owner. This will involve some simple calculations involving the position of the owner, the size of the owner, and the size of the dialog box.

Next you need to ensure that the "OK" and "Cancel" buttons actually close the dialog. To do this, you need to register an action listener on the buttons which, when invoked, calls the `dispose()` method in `JDialog`. You can take a look at the `ApplicationFrame` class for an example of how to register an action listener to a button.

Finally, you need to extend the application with a "Choose Issue" toolbar button. This button should have an action listener registered which, when invoked, would create an instance of the dialog class `X` and show it to the user. The latter can be down by invoking the `setVisible()` method in `JDialog`.

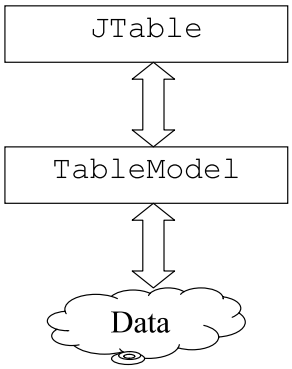

Figure 4. How a **`JTable`** Gets Its Data

After you have completed all these steps, your application should be similar to the one shown in Figure 3.

5.2.2 Populating the Table with Issues

In order to complete your assignment, you next have to actually show the list of issues in the table of the dialog. A straightforward way to do so would be to simply go through the issues in the `IssueArchive` and copy the information about the issues into the table. This, however, is a highly unsatisfactory solution because it duplicates information between the `IssueArchive` and the table. Whenever you duplicate information in software, you get into a problem of keeping the two pieces of information in sync. Very often, it happens that one piece gets changed but does not, which then leads to hard-to-detect bugs. Therefore, you are required to solve this assignment in a different way.

The idea is to display the information about issues without any duplication. The main problem with that lies in the fact that the `JTable` component does not know anything about the format of information in the `IssueArchive`. We are in luck, however, as the designers of the `JTable` component have anticipated such situations. They have decoupled the part of the component that draws the information in a table from the part of component that gets the information. The information shown in a `JTable` is modeled using the `TableModel` interface. By initializing a `JTable` with an appropriate table model, you determine which information is actually shown in the table.

A table model can physically contain the data. More often than not, however, a table model does not contain the data itself; instead, it acts as a mediator between the `JTable` and some external data source. This situation is schematically shown in Figure 4. The table model is then said to be an *adapter* for the data source. The data is accessed using the following sequence of calls. A `JTable` decides that it needs to paint something on screen. In order to obtain the data, it asks the `TableModel` what is actually located in a cell at a particular row and column. The table model then goes to the external data source, retrieves the information, and returns it to the `JTable`. Finally, the table paints this data. Hence, the data merely passes through the table model and is not stored there. The main purpose of the table model is to access the external source and reformat the data into the format that the `JTable` knows how to work with.

In order to keep the table model in sync with the table, the `TableModel` interface implements the model-view-controller (MVC) paradigm.[8] A `JTable` registers itself as a listener to the table model. If anything in the table model changes, the table model should fire the appropriate event. This allows the `JTable` to be notified of change and update its appearance appropriately.

The `TableModel` interface consists of the following methods:

- `addTableModelListener()` and `removeTableModelListener()` allow a component to register itself as being interested in changes in the data of the table model;
- `getColumnCount()` returns the number of columns;
- `getRowCount()` returns the number of rows;
- `getColumnClass(int columnIndex)` determines the type of objects occurring in column `columnIndex` (e.g., `String`, `Boolean`, `Integer`, etc.)
- `getColumnName(int columnIndex)` determines the name of the column `columnIndex`, and it is used to display the appropriate header in the table;
- `getValueAt(int rowIndex, int columnIndex)` should return the object occurring at row `rowIndex` and column `columnIndex` in the table;
- `isCellEditable(int rowIndex, int columnIndex)` should return `true` if the cell in row `rowIndex` and column `columnIndex` can be edited; and
- `setValueAt(Object aValue, int rowIndex, int columnIndex)` should change the object stored at row `rowIndex` and column `columnIndex`.

Thus, `getColumnCount()` and `getRowCount()` determine the layout of the table, and `getValueAt()` determines the value in a particular row. These are the most important methods of `TableModel`, since they are responsible for reformatting an external data source into a tabular format. In order to help you get started with an implementation of a table model, Swing provides the `AbstractTableModel` class, which provides the machinery for managing the list of registered listeners.

To display the issue list, you need to implement a table model that mediates between the `IssueArchive` and the `JTable`. Create a new class—it will be called `Y` in the rest of this document—that subclasses `AbstractTableModel`. Think about choosing an appropriate name for the class. It is common to implement `Y` as an inner class of the dialog class `X`. In this way, you are not polluting the global name space with classes that are doing little bits and pieces of work.

---

[8] You should not be confused by the fact that MVC is used as the architecture model for the overall application: it is possible to apply the MVC design pattern at several levels of granularity in an application.

Class `Y` should receive a reference to an `IssueArchive` it is wrapping. The table model should reformat the data into two columns. The first column should show the issue designator in the `month/year` format, whereas the second column should show the number of pages in the issue. Thus, your table model should have two columns. The cells in the first column will be of type `String`, while the cells in the second column will be of type `Integer`. The first column should be called "Issue Date", and the second column should be called "Number of Pages". Based on this information, you should be able to implement the following methods in class `Y`:

- `getColumnCount()`
- `getColumnClass(int columnIndex)`
- `getColumnName(int columnIndex)`

No cells should be editable; hence, `isCellEditable()` should always return `false` and you can simply introduce an empty implementation for `setValueAt()`. In fact, these methods are already implemented in the `AbstractTableModel`, so you need not worry about them at all.

The number of rows in the table model should be the same as the number of issues in an `IssueArchive`, and the `getValueAt()` method should return the appropriate value by looking it up in the `IssueArchive`. There is a problem, however: `IssueArchive` currently does not allow us to ask a question "how many issues are there", and "give me an issue with some index". Therefore, you need to extend the `IssueArchive` with two new methods:

- One method should simply return the number of issues in the archive. This information can be obtained from a `Map` storing the issues.
- Another method should accept an index of an issue and return an issue with the given index.

Implementing the latter method is currently not trivial because `IssueActhive` does not store issues in an ordered manner (a `Map` provides no assumptions about the order of elements stored in it). To make accessing an issue with a given index easier, you should extend `IssueArchive` such that it, in addition to the `Map`, it also stores the issues in an array. The `Map` is then used to access an issue by the month–year pair, whereas the list is used to access the issue by an index. Note that the latter change does not affect any of the clients of `IssueArchive`—that is, we can change the implementation details of the issue archive without affecting the clients of the class. This is one of the core ideas underpinning object-oriented programming.

Once you have done this, you should go back to the dialog class `X` and make sure that, when you create the `JTable`, you also create an adapter table model `Y` and pass it to the table. That's it!

5.2.3 Processing User's Choice

If the user clicks "OK", you should record his choice and reconfigure the application. In the sample code, this is made difficult by the fact that both the

ApplicationFrame and the PageView are initialized with a fixed issue that cannot be changed subsequently. Furthermore, both classes assume that the current issue is never missing. The last assumption is a problem: when the application starts, there is no issue until the user selects one.

Addressing these problems is not difficult. First, you should ensure that the current issue is stored only in one place. The natural place for doing so is the PageView class. Thus, should remove the final modifier for the member variable that holds the current issue and you should provide methods in PageView that allow the user to set the current issue. You should then modify the ApplicationFrame such that, if it needs the current issue, it can retrieve one from the PageView. In this way, the current issue is stored on one place, which makes updating it easier. You should also carefully examine the code of both classes and check whether it can happen that it accesses the current issue which can be null; if you detect such a situation, you need to decide what to do when there is no current issue. Hint: this will be necessary during painting of PageView. The paintComponent() method does not take into account that the issue can be null; in fact, if it is null, the application will crash. You should modify the painting routine such that, if there is no current issue, it shows the text "No issue has been selected".

Finally, you need to modify the action handler for the "OK" button such that, when it is invoked, it updates the current issue. First, it should determine how many rows have been selected in the table by invoking the getSelectedRowCount() method of JTable. If the result is different from one, the dialog box should remain open (since the user has not precisely specified the input). Otherwise, by invoking getSelectedRow() of JTable, the action handler can determine which row/issue has been selected, retrieve the issue from the IssueArchive, and set it to the ApplicationFrame.

5.3 Allowing the User to Create a new Issue

Creating a new issue is not difficult. The first step is to add a button "New Issue" to dialog X. When this button is clicked, a new dialog should open allowing the user to input the month and a year. In the rest of this document, this new dialog will be called Z.

The dialog Z should be implemented in pretty much the same way as X. The owner of Z should be X; in this way, while Z is being shown, the user cannot work with X. The main difference from X is in the layout of the components. You should create one panel whose layout manager is BoxLayout (with an Y_AXIS orientation). Then, you should add to this panel a label with the text "Month:", a text field, a label with the text "Year:", and another text field. You should set the preferred size of the text fields to be 100 by 20. You can then add the components panel to the CENTER region of the content pane. The buttons can be created in the same way as in X. You can take a look at the ApplicationFrame class to see how text fields are used.

The main problem is what to do when the user actually clicks "OK" in dialog Z. Clearly, one must add a new issue to the IssueArchive, but how to update the JTable showing all the issues?

A naïve solution is to explicitly tell the `JTable` to repaint itself. This, however, is not acceptable. What if there are other components that need to be notified when a new issue is added? You clearly do not want to keep track of all of them; rather, you would like all of them to magically update themselves.

To solve this problem, you should again apply the MVC pattern. You should modify the `IssueArchive` such that it notifies clients of issues being added. To do so, you should introduce the `IssueArchiveListener` interface which will receive these notifications. You should think about the methods that you should include into this interface. Hint: for the moment, one method should suffice. Think about when you would like for this method to be invoked.

You should then modify the `IssueArchive` such that it keeps a list of listeners and allow clients to register different listeners. Similar code is already available in the `Page` class. Finally, you should modify the `IssueArchive` such that the archive notifies all registered listeners after an issue has been added.

You should then modify the table model `Y` to register itself as a listener to the `IssueArchive`. You can take a look at the `Page` class to see how it registers itself as a listener to a `Page`. When the table model receives a notification that the issue archive has been updated, it should transform this notification into a notification of the `TableModel` interface. In this particular example, the easiest thing to do is to tell the `JTable` that all data in the table has changed; you can do this by invoking the `fireTableDataChanged()` method from the `AbstractTableModel` class.

In a more realistic application, you should inform a `JTable` of the row that has been changed, such that not entire table needs to be repainted. This might be too ambitions for this practical; however, if you feel eager, please do so.

A final **very important** point is about memory leaks. Currently, the table model will register itself as a listener to the `IssueArchive` **every time the "Choose Issue" dialog has been opened**. This means that, each time this dialog is shown, a new object gets registered with the `IssueArchive`. These objects are never deregistered from the `IssueArchive`, so the archive will keep the reference to them indefinitely and the objects can never be garbage collected. Thus, each time the "Choose Issue" dialog is opened, we loose a small amount of memory. If you opened the dialog often enough, the available memory would eventually be exhausted and your application would crash.

In order to prevent this, you need to **cleanly deregister** the table model from the `IssueArchive` each time a dialog `X` is closed. But how do you know when the window is closed? To get notified of that, you should register a `WindowListener` with the dialog; then, when the dialog is closed, your listener will receive a `windowClosed()` notification. While handling this notification, you should then ensure that the table model actually gets removed as a listener from the `IssueArchive`. The most convenient way for doing this is to introduce a `dispose()` method in the table model class `Y`. This method should be called from the `windowClosed()` handler and should unregister the listener from the `IssueArchive`.

The `WindowListener` interface contains seven methods in total. Since you must implement all methods in an interface, you would need to provide a dummy implementation for six uninteresting methods and just one implementation for the interesting method. This is clearly unsatisfactory, so Java provides a `WindowAdapter` class, which implements the `WindowListener` interface and provides a dummy implementation for all seven methods. Thus, instead of implementing the `WindowListener` interface, you can save yourself some work by subclassing the `WindowAdapter` class and overriding the `windowClosed()` method.