Workbook 6

Introduction

Last week you applied some basic software design principles to your program. You used an interface to provide abstraction, allowing several alternative storage implementations to be written and easily substituted for one another. In addition, you used inheritance to share code between those implementations. This week you are going build your own graphical interface with Java Swing in order to make your program more accessible to other users.

Important

You may find Sun's Swing Tutorial helpful:

http://java.sun.com/docs/books/tutorial/uiswing/

Remember to check the course website regularly for announcements and errata:

http://www.cl.cam.ac.uk/teaching/current/ProgJava

Java Swing

Java Swing is a graphical user interface supported by the Java standard library which supports a vast range of graphical components. Components include *widgets* such as buttons, menus and text boxes, as well as *containers* which may contain (and control the layout) of any widgets or containers within them. Because containers may themselves contain containers, a typical graphical interface in Swing is built by recursively nesting containers and widgets. You will use this technique in this workbook to layout a graphical interface for the Game of Life.

To create a new Swing application you will need to create a window to display your graphical components. A window in Swing is created by creating an instance of `JFrame`. Here is a simple example:

```
package uk.ac.cam.your-crsid.tick6;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloSwingWorld extends JFrame {
 HelloSwingWorld() {
  super("Hello Swing");                    //create window & set title text
  setDefaultCloseOperation(EXIT_ON_CLOSE); //close button on window quits app.
  JLabel text = new JLabel("Hello Swing"); //create graphical text label
  add(text);                               //associate "text" with window
  setSize(320,240);                        //set size of window
 }
 public static void main(String[] args) {
  HelloSwingWorld hello = new HelloSwingWorld(); //create instance
  hello.setVisible(true);                        //display window to user
 }
}
```

1. Type in `HelloSwingWorld`, compile it and run it.

As you can see, writing a basic graphical interface is quite simple. The `import` statements tell Java that the definitions of `JFrame` and `JLabel` are inside the package `javax.swing`. You might like to look at the documentation for these widgets in the Java documentation.

In the example above, `HelloSwingWorld` extends `JFrame` to create a new window. The use of inheritance used here, whilst optional, is convenient since it makes it easy to invoke the methods `setDefaultCloseOperation`, `add`, `setSize` and `setVisible`.

An instance of `JLabel` represents a string on the graphical interface; note however that the `add` method must be called to attach the instance called `text` to the instance of `HelloSwingWorld` created in the constructor. The `setSize` operator configures the default size of the window when it is created, although the user can change the size at run-time (try resizing the window yourself to see what happens). Finally, `setVisible` is called to display the window to the user; if you forget to call `setVisible` with the argument `true`, your graphical interface will not be displayed and your program will terminate instead.

In order to create more complicated graphical interfaces, you will need to tell Swing how to layout the elements in the window. In order to get a visual impression of the layout options, take a look at the layouts described in the Swing tutorial: http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html. You do not need to work through the tutorial now, although you might like to look at it if you get stuck or you want to know more. There are eight layout types: `BorderLayout`, `BoxLayout`, `CardLayout`, `FlowLayout`, `GridBagLayout`, `GridLayout`, `GroupLayout` and `SpringLayout`.

In order to create a graphical interface for the Game of Life, the screen should be divided up as shown in Figure 1, “Layout of GUI for the Game of Life; the left shows the GUI, and the right shows the names of the main interface containers”. Controlling the size of each of the components as the window is resized is important: the window should make good use of the display area whatever the size of the window. It is a bad idea to assume that the window is of a fixed size, since the size of screens varies between computers, and some users may prefer the window to occupy only a portion of the desktop in order to view multiple applications simultaneously. The decision of how best to use the space available in the window is application specific.

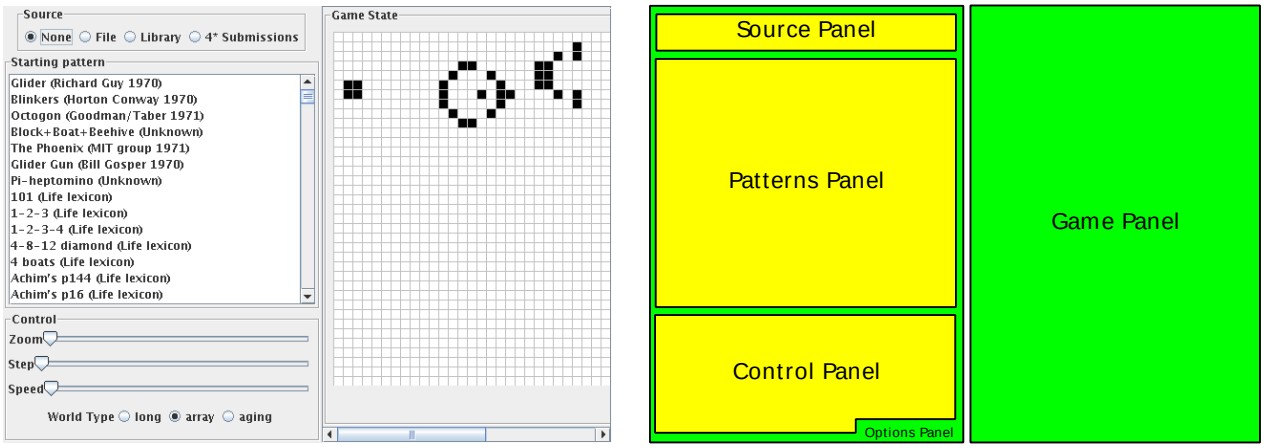

Figure 1. Layout of GUI for the Game of Life; the left shows the GUI, and the right shows the names of the main interface containers

When resizing the graphical interface for the Game of Life, the main window should grow or shrink the Game Panel depending on the space available; in contrast the size of the Options Panel should remain fixed. The Options Panel (which contains the Source Panel, Pattern Panel, and Control Panel) should take up only the space which is necessary to display the widgets within it. This kind of control over resizing can be achieved using `BorderLayout`. Below is an example program which demonstrates how to use this layout type.

```
package uk.ac.cam.your-crsid.tick6;

import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GuiLife extends JFrame {
 public GuiLife() {
  super("Coloured Boxes");
  setSize(640,480);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setLayout(new BorderLayout());

  JPanel optionsPanel = createOptionsPanel();
  add(optionsPanel,BorderLayout.WEST);

  JPanel gamePanel = createGamePanel();
  add(gamePanel,BorderLayout.CENTER);
 }

 private JPanel createOptionsPanel() {
  //TODO
 }

 private JPanel createGamePanel() {
  JPanel result = new JPanel();
  result.setBackground(Color.GREEN);
  return result;
 }

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

2. Type in `GuiLife` and complete the method `createOptionsPanel` which should create a new instance of `JPanel`, set the background colour to `Color.BLUE` and return a reference to the new instance of `JPanel` you created to the caller.

If you run the program `GuiLife` and resize the window, you should find that the green area (a placeholder for the Game Panel) takes up all the additional space made available through expanding the size of the window, and the size of the blue area (a placeholder for the Options Panel) remains fixed. This is just the behaviour required for the graphical interface for the Game of Life. The window behaves in this way because the behaviour of `BorderLayout` is to resize the "center" panel in preference to any other panels which might exist.

Nested components

In the previous section you created a window which used `BorderLayout` to control the layout of the window. We will now add further components to the Options Panel to hold the Source Panel, Pattern Panel and Control Panel. Below is a program which, when completed, will create space for all the key parts of our graphical interface.

```
package uk.ac.cam.your-crsid.tick6;

import java.awt.BorderLayout;
import javax.swing.border.Border;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.border.EtchedBorder;

public class GuiLife extends JFrame {

 public GuiLife() {
  super("GuiLife");
  setSize(640, 480);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setLayout(new BorderLayout());
  JComponent optionsPanel = createOptionsPanel();
  add(optionsPanel, BorderLayout.WEST);
  JComponent gamePanel = createGamePanel();
  add(gamePanel, BorderLayout.CENTER);
 }

 private JComponent createOptionsPanel() {
  Box result = Box.createVerticalBox();
  result.add(createSourcePanel());
  result.add(createPatternPanel());
  result.add(createControlPanel());
  return result;
 }

 private void addBorder(JComponent component, String title) {
  Border etch = BorderFactory.createEtchedBorder(EtchedBorder.LOWERED);
  Border tb = BorderFactory.createTitledBorder(etch,title);
  component.setBorder(tb);
 }

 private JComponent createGamePanel() {
  JPanel holder = new JPanel();
  addBorder(holder,Strings.PANEL_GAMEVIEW);
  JPanel result = new JPanel();
  holder.add(result);
  return new JScrollPane(holder);
 }

 private JComponent createSourcePanel() {
  JPanel result = new JPanel();
  addBorder(result,Strings.PANEL_SOURCE);
  return result;
 }

 private JComponent createPatternPanel() { /*TODO*/ }
 private JComponent createControlPanel() { /*TODO*/ }

 public static void main(String[] args) {
  GuiLife gui = new GuiLife();
  gui.setVisible(true);
4}
}
```

In the example program, the Options Panel is created as a container with a vertical `BoxLayout` layout; such a component places all the components within itself vertically. The layout of widgets inside a container such as the Options Panel is quite flexible in Swing. Widgets have three sets of dimensions: a minimum size, a preferred size, and a maximum size. In the case of a vertical `BoxLayout` as used in the Options Panel, Swing attempts to provide space for widgets at their preferred size. More information on the resizing of widgets using a particular layout with Swing can be found in Sun's documentation for the appropriate layout class.

In the example above, you may have noticed there is a new object called `Strings`. This is in fact another class in the same package as `GuiLife` which contains all of the text data used in the graphical interface. It's a good idea to collect together all the strings used in the interface in one place so that they are easy to modify; this is especially useful if you wish to update the application so that it supports multiple languages. The contents of `Strings` should be as follows:

```
package uk.ac.cam.your-crsid.tick6;

public class Strings {
 public static final String PANEL_SOURCE = "Source";
 public static final String PANEL_PATTERN = "Starting pattern";
 public static final String PANEL_CONTROL = "Control";
 public static final String PANEL_GAMEVIEW = "Game State";
}
```

3. Place the contents of `Strings` as defined above into a suitable file and directory structure.

4. Replace the your copy of `GuiLife` with the new one provided in this section.

5. Complete the methods `createPatternPanel`, and `createControlPanel` in `GuiLife`. Your implementations should create a new instance of `JPanel`, add a border to the panel using `addBorder`, and return a reference to the instance of `JPanel` which you created; you should use a suitable field from `Strings` for the border text. You might find reading the body of `createSourcePanel` helpful.

6. Run your improved version of `GuiLife`. You should see that the screen is now split into four areas, although the text associated with the etched borders will not be entirely visible for the three panels in the left half of the screen. This is okay at the moment.

Source Panel

In the previous section, you created space for the four main elements of the final graphical display. In this section you will complete the body of the Source Panel. The Source Panel will eventually allow the user to select the source of the patterns for the initial state of the world. Possible sources could include a file in the file system, or a URL. This week you will add support for four possible input methods on the display, none of which will actually function! Next week you will implement the code to make them work.

In your application you will only ever have one source at a time, and therefore a user interface component which only allows the user to select a single source is required. One such user interface component is the *Radio Button* which provides the user with a selection from one of several alternatives.

To use a Radio Button in your implementation of Source Panel you should create a separate class to contain the details of the widgets used. Below is a partially complete implementation of `SourcePanel` which uses `JRadioButton` widgets to control user selection of the source of patterns for the Game of Life.

```
package uk.ac.cam.your-crsid.tick6;

//TODO: specify the appropriate import statements

public class SourcePanel extends JPanel {

 public SourcePanel() {
  super();
  setLayout(new BoxLayout(this,BoxLayout.X_AXIS));
  JRadioButton none = new JRadioButton(Strings.BUTTON_SOURCE_NONE, true);
  JRadioButton file = new JRadioButton(Strings.BUTTON_SOURCE_FILE, true);
  JRadioButton library = new JRadioButton(Strings.BUTTON_SOURCE_LIBRARY, true);
  JRadioButton fourStar = new JRadioButton(Strings.BUTTON_SOURCE_FOURSTAR, true);
  //add RadioButtons to this JPanel
  add(none);
  add(file);
  add(library);
  add(fourStar);
  //create a ButtonGroup containing all four buttons
  //Only one Button in a ButtonGroup can be selected at once
  ButtonGroup group = new ButtonGroup();
  group.add(none);
  group.add(file);
  group.add(library);
  group.add(fourStar);
 }
}
```

The class `SourcePanel` extends `JPanel`, creating a new interface component which you can add to an existing `JFrame` or `JComponent`. The constructor for `SourcePanel` first calls the default constructor of the super class, and then configures the layout for this component as a `BoxLayout` with components laid horizontally. Four new instances of `RadioButton` are created, and these are then added to the current instance of `SourcePanel` under construction using the `add` method supplied by the parent class. The remainder of the constructor creates a new instance of `ButtonGroup` and then adds the instances of `JRadioButton` to it; `ButtonGroup` prevents more than one `JRadioButton` instance in the group from being selected at the same time.

7. Place the code above for `SourcePanel` inside a suitable class and package structure, and use Sun's Java documentation to determine the appropriate import statements for the class.
8. Create four new entries inside your `Strings` class to provide textual data for the Java `String` literals used in the `JRadioButton` objects created in `SourcePanel`. The `String` literals should be `"None"`, `"File"`, `"Library"`, and `"4* Submissions"`.
9. Update your implementation of the method `createSourcePanel` inside `GuiLife` to create a new instance of `SourcePanel` instead of `JPanel`.
10. Run `GuiLife`. You should see four radio buttons inside the Source Panel with appropriate text labels. It should only be possible to select a single radio button at a time. Compare your implementation of the Source Panel with that shown in Figure 1.

Loading a list of patterns

In this section you will load a list of patterns into the Pattern Panel. To do so, you will need to use the `JList` object to contain the list of patterns. The `JList` component should be placed inside a

`JScrollPane`, which adds scroll bars to the list; this is required since the number of patterns may be quite large and you will not want the window to grow too large. Here is an outline of the code needed to implement the Pattern Panel:

```
package uk.ac.cam.your-crsid.tick6;

//TODO:  specify the appropriate import statements

public class PatternPanel extends JPanel {

 private JList guiList;

 public PatternPanel() {
  super();
  setLayout(new BorderLayout());
  guiList = new JList();
  add(new JScrollPane(guiList));
 }

 public void setPatterns(List<Pattern> list) {
  ArrayList<String> names = new ArrayList<String>();

  //TODO: Using a for loop which iterates over the items
  //      in "list" and adds the pattern name and pattern
  //      author to "names". For example, if the pattern
  //      name and author is "Glider" and "Richard Guy 1970"
  //      then you should add the string:
  //
  //               "Glider (Richard Guy 1970)"
  //
  //      to "names" using the method "add" on "names".

  guiList.setListData(names.toArray());
 }
}
```

11. Copy across your implementation of `Pattern`, `PatternLoader`, `PatternFormatException`, `WorldImpl`, `PackedWorld`, `ArrayWorld`, `AgingWorld` and `PackedLong` from last week into the package `uk.ac.cam.your-crsid.tick6`; remember to update the package statements at the top of these files accordingly.

12. Place the code above for `PatternPanel` inside a suitable class and package structure, and use Sun's Java documentation to determine the appropriate import statements for the class.

13. Complete the body of `setPatterns` to add a string for each of the patterns found in `list` into the local array of strings called `names`.

14. Create a new field called `patternPanel` of type `PatternPanel` inside the class `GuiLife`.

15. Update your implementation of the method `createPatternPanel` inside `GuiLife` to create a new instance of `PatternPanel` instead of `JPanel`. Your implementation should ensure that the field you created in the previous step references the new instance you create in this method.

16. Add the following lines of code immediately after the definition of the variable `gui` in the `main` method of `GuiLife`:

```
try {
 String url="http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt";
 List<Pattern> list = PatternLoader.loadFromURL(url);
 gui.patternPanel.setPatterns(list);
} catch (IOException ioe) {}
```

and add an import statement to import the exception `IOException` and the class `List` into `GuiLife`.

17. Run `GuiLife`. You should find that the Pattern Panel is populated with the list of pattern names available from the course website.

Control Panel

The Control Panel part of the interface shown in Figure 1 looks quite complex. The method of laying out the elements you should use is to create a vertical `BoxLayout` inside the Control Panel with four elements, each element of which contains a horizontal `BoxLayout`; the first, second and third inner horizontal `BoxLayout` objects will contain a `JLabel` object followed by a `JSlider` object. The last element is a `JLabel` followed by a group of three `JRadioButton` objects. It's perhaps best to view this layout visually by looking back to Figure 1 and then reading the code which lays out the Control Panel as just described:

```
//TODO: Write a suitable package statement and import statements
public class ControlPanel extends JPanel {

 private JSlider zoomSlider;
 private JSlider stepSlider;
 private JSlider speedSlider;
 private JRadioButton longButton;
 private JRadioButton arrayButton;
 private JRadioButton agingButton;

 private JSlider createNewSlider(int min, int max, int start, String s) {
  Box panel = Box.createHorizontalBox();
  add(panel);
  panel.add(new JLabel(s));
  JSlider slider = new JSlider(min,max,start);
  panel.add(slider);
  return slider;
 }

 private JRadioButton createNewButton(String s, ButtonGroup g, Box b) {
  //TODO: create a new instance of JRadioButton with text "s"
  //TODO: add the new instance to the ButtonGroup referenced by "g"
  //TODO: add the new instance to Box "b"
  //TODO: return a reference to the new instance
 }
```

```
 public ControlPanel() {
  super();
  setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));

  zoomSlider = createNewSlider(1,20,1,Strings.CONTROL_ZOOM);
  add(Box.createVerticalStrut(10)); //add 10px of extra space
  stepSlider = createNewSlider(0,10,0,Strings.CONTROL_STEP);
  add(Box.createVerticalStrut(10)); //add 10px of extra space
  speedSlider = createNewSlider(0,100,0,Strings.CONTROL_SPEED);
  add(Box.createVerticalStrut(10)); //add 10px of extra space

  Box worldPanel = Box.createHorizontalBox();
  add(worldPanel);
  worldPanel.add(new JLabel(Strings.STORAGE_WORLD_TYPE));
  ButtonGroup group = new ButtonGroup();
  longButton = createNewButton(Strings.STORAGE_LONG,group,worldPanel);
  arrayButton = createNewButton(Strings.STORAGE_ARRAY,group,worldPanel);
  agingButton = createNewButton(Strings.STORAGE_AGING,group,worldPanel);
  arrayButton.setSelected(true);
  add(Box.createVerticalStrut(10)); //add 10px of extra space
 }

 public World initialiseWorld(Pattern p) throws PatternFormatException {
  World result = null;
  if (longButton.isSelected()) {
   result = new PackedWorld();
  } else if (arrayButton.isSelected()) {
   result = new ArrayWorld(p.getWidth(),p.getHeight());
  } else if (agingButton.isSelected()) {
   result = new AgingWorld(p.getWidth(),p.getHeight());
  }
  if (result != null)  p.initialise(result);
  return result;
 }
}
```

18. Place the code above for `ControlPanel` inside a suitable class and package structure.

19. Complete the items marked `TODO` in `ControlPanel`.

20. Create seven new entries inside your `Strings` class to provide textual data for the Java `String` literals used in the `ControlPanel`. The `String` literals should be `"Zoom"`, `"Step"`, `"Speed"`, `"World type"`, `"Long"`, `"Array"`, and `"Aging"`.

21. Create a new field called `controlPanel` of type `ControlPanel` inside the class `GuiLife`.

22. Update your implementation of the method `createControlPanel` inside `GuiLife` to create a new instance of `ControlPanel` instead of `JPanel`. Your implementation should ensure that the field called `controlPanel` you created in the previous step references the new instance you create in this method.

23. Run `GuiLife`. You should find that the Control Panel now displays the labels, sliders and radio buttons as shown in Figure 1.

Game Panel

The Game Panel should display the state of the game board. The code for the Game Panel is:

```
//TODO: Write a suitable package statement and import statements
public class GamePanel extends JPanel {

 private int zoom = 10; //Number of pixels used to represent a cell
 private int width = 1; //Width of game board in pixels
 private int height = 1;//Height of game board in pixels
 private World current = null;

 public Dimension getPreferredSize() {
  return new Dimension(width, height);
 }

 protected void paintComponent(Graphics g) {
  if (current == null) return;
  g.setColor(java.awt.Color.WHITE);
  g.fillRect(0, 0, width, height);
  current.draw(g, width, height);
  if (zoom > 4) {
   g.setColor(java.awt.Color.LIGHT_GRAY);
   //TODO: Using for loops call the drawLine method on "g",
   //      repeatedly to draw a grid of grey lines to delimit
   //      the border of the cells in the game board
  }
 }

 private void computeSize() {
  if (current == null)  return;
  int newWidth = current.getWidth() * zoom;
  int newHeight = current.getHeight() * zoom;
  if (newWidth != width || newHeight != height) {
   width = newWidth;
   height = newHeight;
   revalidate(); //trigger the GamePanel to re-layout its components
  }
 }

 public void display(World w) {
  current = w;
  computeSize();
  repaint();
 }
}
```

24. Place the code above for `GamePanel` inside a suitable class and package structure.

25. Complete the items marked `TODO` in `GamePanel`. Remember to look at Figure 1 to see how the gray borders between the cells in the game board should be drawn.

26. Create a new field called `gamePanel` of type `GamePanel` inside the class `GuiLife`.

27. Update your implementation of the method `createGamePanel` inside `GuiLife` to create a new instance of `GamePanel` instead of `JPanel`. Your implementation should ensure that the field called `gamePanel` you created in the previous step references the new instance you create in this method.
28. Add the following two lines of code immediately after the call to `gui.patternPanel.setPatterns` in the `main` method of `GuiLife`

    ```
    World w = gui.controlPanel.initialiseWorld(list.get(0));
    gui.gamePanel.display(w);
    ```

    The first line of code above may throw an instance of `PatternFormatException`. You should write an additional `catch` block to handle this type of exception and print a suitable error message.
29. Run `GuiLife`. You should find that the Game Panel now contains a graphical depiction of a "Glider" by Richard Guy.
30. Modify your program so that it displays the second pattern available from the website, called "Blinkers" by Horton Conway.

Graphical world (Optional)

The work described in this section is optional; you may skip the work described here and proceed directly to the last section of the Workbook. In the previous sections, you have displayed all the widgets needed to support a graphical interface for the Game of Life. Unfortunately none of the widgets actually do anything! For example, changing the zoom slider does not actually change the zoom level of the Game Panel. Next week we will investigate how to support *event handlers* in Java to update parts of the program in response for user input and make the graphical interface fully functional.

This week you will finish your implementation by adapting code from the `main` method you wrote for `RefactorLife` last week. The code you wrote last week parsed options, a URL or file, and an integer the user provided to the program on the command line. Many of you produced a somewhat lengthy and messy implementation! A better solution is to create a new class to store the pertinent details provided by the user on the command line. Here is a skeleton structure of such a class:

```
package uk.ac.cam.your-crsid.tick6;

public class CommandLineOptions {

 public static String WORLD_TYPE_LONG = "--long";
 public static String WORLD_TYPE_AGING = "--aging";
 public static String WORLD_TYPE_ARRAY = "--array";
 private String worldType = null;
 private Integer index = null;
 private String source = null;

 public CommandLineOptions(String[] args) throws CommandLineException {
  //TODO: parse "args" to update the private fields "worldType",
  //      "index" and "source" with the correct values, if any.
 }
 public String getWorldType() {return worldType;}
 public Integer getIndex() {return index;}
 public String getSource() {return source;}
}
```

Notice that the constructor for `CommandLineOptions` may throw an exception of type `CommandLineException`, which you should define as follows:

```
package uk.ac.cam.your-crsid.tick6;
public class CommandLineException extends Exception {
 public CommandLineException(String message) {
  super(message);
 }
}
```

In the constructor of `CommandLineException` you should pass in a message of type `String`. Since `CommandLineException` inherits from `Exception` you can retrieve the message using the `getMessage` method on `CommandLineException`. You will find the following implementation of `TextLife` helpful in writing and testing your implementation of `CommandLineOptions`:

```
//TODO: Write a suitable package statement and import statements
public class TextLife {

 public static void main(String[] args) {
  CommandLineOptions c = new CommandLineOptions(args);
  List<Pattern> list;
  if (c.getSource().startsWith("http://"))
   list = PatternLoader.loadFromURL(c.getSource());
  else
   list = PatternLoader.loadFromDisk(c.getSource());
  if (c.getIndex() == null) {
   int i = 0;
   for (Pattern p : list)
    System.out.println((i++)+" "+p.getName()+" "+p.getAuthor());
  } else {
   Pattern p = list.get(c.getIndex());
   World w = null;
   if (c.getWorldType().equals(CommandLineOptions.WORLD_TYPE_AGING)) {
    w = new AgingWorld(p.getWidth(), p.getHeight());
   } else if (c.getWorldType().equals(CommandLineOptions.WORLD_TYPE_ARRAY)) {
    w = new ArrayWorld(p.getWidth(), p.getHeight());
   } else {
    w = new PackedWorld();
   }
   p.initialise(w);
   int userResponse = 0;
   while (userResponse != 'q') {
    w.print(new OutputStreamWriter(System.out));
    try {
     userResponse = System.in.read();
    } catch (IOException e) {}
    w = w.nextGeneration(0);
   }
  }
 }
}
```

The class `TextLife` uses many of the classes you wrote in previous weeks; you should have already copied across the relevant classes to successfully complete previous exercises found in this workbook. You should use the class `TextLife` above to help you test your implementation of `CommandLineOptions`. For example, `TextLife` should function as shown in the following examples:

```
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife
Error: No arguments found
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife \
http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt | head -n 2
0 Glider Richard Guy 1970
1 Blinkers Horton Conway 1970
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife \
http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt -3
Error: Index out of bounds
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife \
--long http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt | head -n 2
0 Glider Richard Guy 1970
1 Blinkers Horton Conway 1970
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife \
http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt 0
-
________
___#____
_#_#____
__##____
________
________
________
________
q
crsid@machine:~> java -cp world.jar:. uk/ac/cam/your-crsid/tick6/TextLife \
--aging http://www.cl.cam.ac.uk/teaching/current/ProgJava/life.txt 0
________
___#____
_#_#____
__##____
________
________
________
________
-
________
__#1____
_1_##___
__##____
________
________
________
________
```

31. Write an implementation of `CommandLineOptions` which parses the command line options in the same way as your implementation of `RefactorLife` did last week.

32. Adapt the definition of `TextLife` as shown above so that, when combined with your implementation of `CommandLineOptions`, it produces the same output as shown in the test cases shown above. Hint: You will need to catch and handle several exceptions which are thrown by methods used in `TextLife`; they aren't currently caught so try compiling `TextLife` as is to see which exceptions must be caught. To handle the errors, you might like to print out the messages associated with the exception using the method `getMessage`.

33. Update your implementation of `GuiLife` to use the implementation of `CommandLineOptions` to interpret the options provided to the program on the command line. If the options provided are incorrect, an error message should be printed out and the program should terminate; if the options are well formatted and a specific starting pattern is specified, then the graphical interface should load with the Game Panel configured with the specified pattern.

Tick 6

To complete your tick you need to prepare a jar file with the contents of all the classes you have written in this workbook and email it to `ticks1a-java@cl.cam.ac.uk`. Your jar file should contain:

```
uk/ac/cam/your-crsid/tick6/HelloSwingWorld.java
uk/ac/cam/your-crsid/tick6/HelloSwingWorld.class
uk/ac/cam/your-crsid/tick6/CommandLineOptions.java  <-- optional
uk/ac/cam/your-crsid/tick6/CommandLineOptions.class <-- optional
uk/ac/cam/your-crsid/tick6/TextLife.java            <-- optional
uk/ac/cam/your-crsid/tick6/TextLife.class           <-- optional
uk/ac/cam/your-crsid/tick6/GuiLife.java
uk/ac/cam/your-crsid/tick6/GuiLife.class
uk/ac/cam/your-crsid/tick6/GamePanel.java
uk/ac/cam/your-crsid/tick6/GamePanel.class
uk/ac/cam/your-crsid/tick6/Strings.java
uk/ac/cam/your-crsid/tick6/Strings.class
uk/ac/cam/your-crsid/tick6/ControlPanel.java
uk/ac/cam/your-crsid/tick6/ControlPanel.class
uk/ac/cam/your-crsid/tick6/PatternPanel.java
uk/ac/cam/your-crsid/tick6/PatternPanel.class
uk/ac/cam/your-crsid/tick6/SourcePanel.java
uk/ac/cam/your-crsid/tick6/SourcePanel.class
uk/ac/cam/your-crsid/tick6/WorldImpl.java
uk/ac/cam/your-crsid/tick6/WorldImpl.class
uk/ac/cam/your-crsid/tick6/ArrayWorld.java
uk/ac/cam/your-crsid/tick6/ArrayWorld.class
uk/ac/cam/your-crsid/tick6/PackedWorld.java
uk/ac/cam/your-crsid/tick6/PackedWorld.class
uk/ac/cam/your-crsid/tick6/AgingWorld.java
uk/ac/cam/your-crsid/tick6/AgingWorld.class
uk/ac/cam/your-crsid/tick6/Pattern.java
uk/ac/cam/your-crsid/tick6/Pattern.class
uk/ac/cam/your-crsid/tick6/PackedLong.java
uk/ac/cam/your-crsid/tick6/PackedLong.class
uk/ac/cam/your-crsid/tick6/PatternLoader.java
uk/ac/cam/your-crsid/tick6/PatternLoader.class
uk/ac/cam/your-crsid/tick6/PatternFormatException.java
uk/ac/cam/your-crsid/tick6/PatternFormatException.class
```

If you completed the optional work, you should be able to run your program in one of three ways:

- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.your-crsid.tick6.TextLife [url/file]`
- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.your-crsid.tick6.TextLife [url/file] [index]`
- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.your-crsid.tick6.TextLife [worldType] [url/file] [index]`