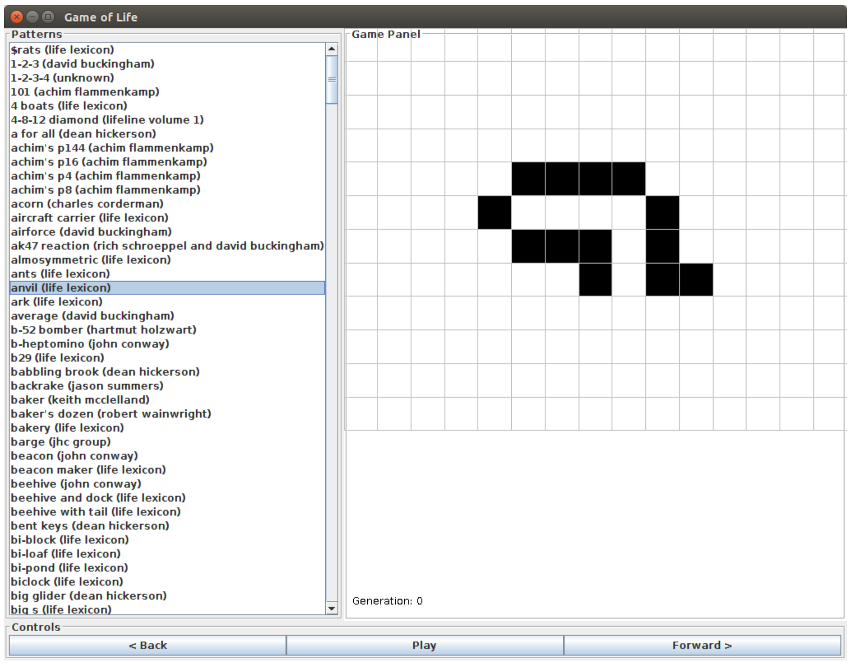
American University of Armenia
CS 120 Intro to OOP
Spring 2019

Homework Assignment 6, Part 2

In this part of the homework you will move away from relying on the command line for all inputs and develop a more modern Graphical User Interface (GUI) using the Swing libraries supplied as part of Java. Along the way you will observe the use of various design patterns to allow the visual structure and dynamic behaviour you would expect. Your final application will look like the screenshot below. Note there is a video of how it should *behave* as part of the assignment on Moodle.

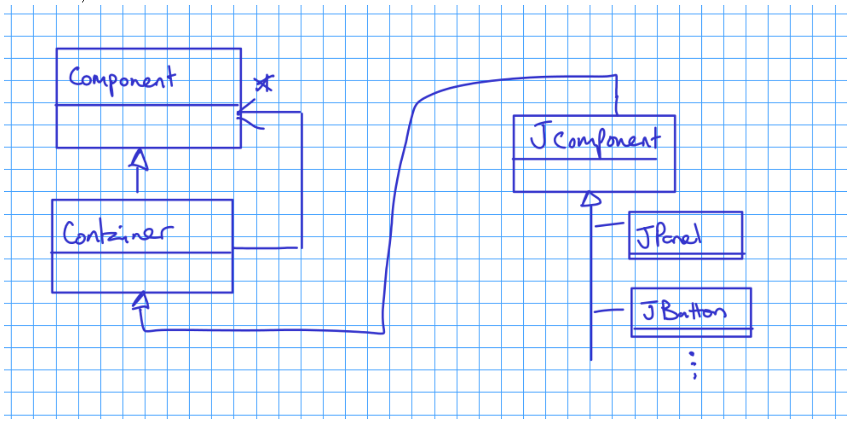

Game of Life
Patterns
$rats (life lexicon)
1-2-3 (david buckingham)
1-2-3-4 (unknown)
101 (achim flammenkamp)
4 boats (life lexicon)
4-8-12 diamond (lifeline volume 1)
a for all (dean hickerson)
achim's p144 (achim flammenkamp)
achim's p16 (achim flammenkamp)
achim's p4 (achim flammenkamp)
achim's p8 (achim flammenkamp)
acorn (charles corderman)
aircraft carrier (life lexicon)
airforce (david buckingham)
ak47 reaction (rich schroeppel and david buckingham)
almosymmetric (life lexicon)
ants (life lexicon)
anvil (life lexicon)
ark (life lexicon)
average (david buckingham)
b-52 bomber (hartmut holzwart)
b-heptomino (john conway)
b29 (life lexicon)
babbling brook (dean hickerson)
backrake (jason summers)
baker (keith mcclelland)
baker's dozen (robert wainwright)
bakery (life lexicon)
barge (jhc group)
beacon (john conway)
beacon maker (life lexicon)
beehive (john conway)
beehive and dock (life lexicon)
beehive with tail (life lexicon)
bent keys (dean hickerson)
bi-block (life lexicon)
bi-loaf (life lexicon)
bi-pond (life lexicon)
biclock (life lexicon)
big glider (dean hickerson)
big s (life lexicon)
Game Panel
Generation: 0
Controls
< Back
Play
Forward >

1. **(5 points)** A Swing application starts with a desktop window. This functionality is provided by the `JFrame` class, and the easiest way to use it is to extend it. Here's your starting point:

```java
import javax.swing.JFrame;

public class GUILife extends JFrame {

  public GUILife() {
    super("Game of Life");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(500,400);
  }

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

  }
}
```

Copy, compile and run this code. You should see an empty window entitled "Game of Life" that can be minimised, maximised, resized or closed (check each of these works as expected).

The window is a container for other components, which can themselves be containers for other components, and so on. To achieve this the Swing libraries apply the *Composite* design pattern, as discussed in lectures:



It is therefore possible to add anything that derives from `Container`, you just need the `add(...)` method. Try adding this to the bottom of your `GUILife` constructor and running it again:

```
add(new JButton("Button 1"));
```

You should see a button labelled "Button 1" that fills the window. Now add another button so you have:

```
add(new JButton("Button 1"));
add(new JButton("Button 2"));
```

You will find button 2 has replaced button 1 (which is nowhere to be seen). This is because each `Container` is associated with a `LayoutManager` that tells it what to do if it finds itself with multiple 'children'. In this case, the default is a `BorderLayout`. This has one central component and four satellite components above, below, left and right of it respectively. When you don't specify where you want a new component to go, it assumes you mean the central bit and puts it there, displacing whatever was there before. To understand this layout better edit your constructor to contain:

```
add(new JButton("Centre"));
add(new JButton("North"), BorderLayout.NORTH);
add(new JButton("South"), BorderLayout.SOUTH);
add(new JButton("West"), BorderLayout.WEST);
add(new JButton("East"), BorderLayout.EAST);
```

Play this and you will be presented with all five buttons laid out as described. Try resizing the window and you should start to see why the `BorderLayout` is named. The border elements only stretch along the direction that touches the window, while the central position scales fully.

There are other layouts available, including `BoxLayout`, `CardLayout`, `FlowLayout`, `GridBagLayout`, `GridLayout`, `GroupLayout` and `SpringLayout`. They are all useful for different types of application and you can find out more by looking here

In this homework you will be using a `BorderLayout` for the main interface, with a visualisation of the board as the central component, and controls in the WEST and SOUTH layout placement.

2. **(5 points)** Your `GUILife` class should replace your `GameOfLife` class from Part 1 of this homework, keeping the key functionality.

   Copy across the `store`, `world` and `cachedWorld` state and the `copyWorld` method from `GameOfLife` to `GUILife`. Update the constructor to say:

   ```
   public GUILife(PatternStore ps) {
     super("Game of Life");
     store=ps;
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setSize(1024,768);
   }
   ```

   Add two private methods `void moveBack()` and `void moveForward()`. Their functionality should be the same as pressing "b" or "f" in Part 1 (i.e. they need to move through the cache or generate new worlds as appropriate).

   In the `main()` method, create a `PatternStore` that reads in `https://bit.ly/2FJERFh` and create a new `GUILife` object, setting it to visible as before. Check your application still runs (albeit without doing anything interesting).

3. **(10 points)** There are three main sections to the interface: the game panel, the patterns panel and the control panel.

   Add the following to `GUILife`:

   ```
   public GUILife(PatternStore ps) {
     super("Game of Life");
     store=ps;
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     setSize(1024,768);

     add(createPatternsPanel(),BorderLayout.WEST);
     add(createControlPanel(),BorderLayout.SOUTH);
     add(createGamePanel(),BorderLayout.CENTER);

   }

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

   private JPanel createGamePanel() {
     // TODO
     return new JPanel(); // temporary return
   }

   private JPanel createPatternsPanel() {
     JPanel patt = new JPanel();
     addBorder(patt,"Patterns");
     // TODO
     return patt;
   }

   private JPanel createControlPanel() {
     JPanel ctrl =  new JPanel();
     addBorder(ctrl,"Controls");
     // TODO
   ```

```
  return ctrl;
}
```

Complete `createControlPanel()` to create three `JButtons` with the labels "< Back", "Play", "Forward >" respectively. They should appear in the SOUTH part of the interface, and should run horizontally, each taking 1/3 of the panel width as per the screenshot above.

Complete `createPatternsPanel()` to produce the WEST panel containing the list of patterns. This is actually a `JPanel` that contains a `JScrollPane` that contains a `JList`: you are expected to use the Java API documentation to work out how to use these elements. The `JScrollPane` automatically provides scroll bars whenever the components it contains cannot be displayed. The `JList` it contains should be initialised using `store` and should present the patterns sorted by name. When you run `GUILife` now, you should see the list of patterns down the left, it should be vertically scrollable and the three buttons should be along the base of the window.

You will find that the `JList` presents each `Pattern` as a string containing a type and memory reference. This is because it doesn't know how to print a `Pattern` object. To fix this, override the `String toString()` method in `Pattern` to print e.g. "pattern-name (author)".

4. **(10 points)** The game panel will be composed of an extended `JPanel`, modified to draw the board. Here is a starting point:

```
import java.awt.Color;
import javax.swing.JPanel;

public class GamePanel extends JPanel {

  private World world = null;

  @Override
  protected void paintComponent(java.awt.Graphics g) {
    // Paint the background white
    g.setColor(java.awt.Color.WHITE);
    g.fillRect(0, 0, this.getWidth(), this.getHeight());

    // Sample drawing statements
    g.setColor(Color.BLACK);
    g.drawRect(200, 200, 30, 30);
    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(140, 140, 30, 30);
    g.fillRect(260, 140, 30, 30);
    g.setColor(Color.BLACK);
    g.drawLine(150, 300, 280, 300);
    g.drawString("@@@", 135,120);
    g.drawString("@@@", 255,120);
  }

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

The `paintComponent()` method is called whenever the system wants to draw the component. It is overridden here to draw something other than the usual block of colour for a panel. The code above includes some example statements that you will need to remove later.

Create `GamePanel.java` and copy the code above into it. Add the following definition of `createGamePanel()` to `GUILife` and run `GUILife`.

```java
private JPanel createGamePanel() {
  gamePanel = new GamePanel();
  addBorder(gamePanel,"Game Panel");
  return gamePanel;
}
```

Change `paintComponent()` to draw the world represented by `world` as shown in the screenshot above. Note the following requirements:

- if `world` is `null` a blank white panel should be displayed
- Live cells should be drawn as black squares; dead remain white
- Cell outlines (perimeters) should be in light gray
- The cells should *always appear square* (not stretched to be rectangular). When resizing the window, the squares should fill as much of the panel as possible *while remaining square*. Note that constraining to perfect squares is only possible if the grid dimension divides exactly into the respective panel dimension. If it does not, you either get a gap (not aesthetically pleasing) or you have to use near-square cells (where you might be out by a pixel in one dimension due to quantisation error). The latter is generally more pleasing to the eye and is what you see in the video.
- The current generation should be printed near the bottom left of the game panel (see image above) as e.g. "Generation : 6".

In order to test your implementation you can create a World object in the `GUILife` constructor and pass it to `GamePanel`'s `display()` method.

5. **(10 points)** The remaining task for you is to handle interaction (e.g. clicking buttons or patterns)

   `JButtons` generate `ActionEvent` objects when clicked. Anything that implements the `ActionListener` interface can register with a `JButton` to receive these event objects. This is an application of the Observer design pattern.

   There are three ways to achieve this:

   - Implement `ActionListener` explicitly. Make your class implement the interface, then register with the button directly. This is a pain if you have multiple buttons since the handler needs to check which button was clicked. E.g.

```java
public class ButtonDemo extends JFrame implements ActionListener {

  public ButtonDemo() {
    JButton b = new JButton("Click me!");
    b.addActionListener(this);
    add(b);
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("Button was clicked!");
  }
}
```

- Use an anonymous class to avoid having to spread the code out and make handling multiple buttons easy:

  ```
  public class ButtonDemo extends JFrame  {

    public ButtonDemo() {
      JButton b = new JButton("Click me!");
      b.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
          System.out.println("Button was clicked!");
        }
      });
      add(b);
    }
  }
  ```

- Use a lambda function to make this all look neater:

  ```
  public class ButtonDemo extends JFrame  {

    public ButtonDemo() {
      JButton b = new JButton("Click me!");
      b.addActionListener( e -> System.out.println("Button was clicked!"));
      add(b);
    }
  }
  ```

Using a method of your choice, make the back and forward buttons call the appropriate methods in `GUILife`. Leave the middle “Play” button for now.

6. **(5 points)** You need to make it so that clicking on a pattern in the `JList` causes that pattern to be loaded in at generation zero. This is very similar to connecting up buttons, except you need to use a `ListSelectionListener` to receive `ListSelectionEvents`.

   Adapt `GUILife` to implement `ListSelectionListener`. Register the current `GUILife` object with the `JList` just after you create it (hint: use `addListSelectionListener`). Here is a skeleton `valueChanged()` method:

   ```
   @Override
     public void valueChanged(ListSelectionEvent e) {
       JList<Pattern> list = (JList<Pattern>) e.getSource();
       Pattern p = list.getSelectedValue();
       // TODO
       // Based on size, create either a PackedWorld or ArrayWorld
       // from p. Clear the cache, set world and put it into
       // the now-empty cache. Tell the game panel to display
       // the new world.
     }
   ```

7. **(5 points)** The final task is to enable the “Play” button to start animating the current world. You do this using a `Timer` to call `moveForward()` at regular intervals.

   In `GUILife`, add member variables `playButton` referencing the “Play” `JButton`; `timer` of type `java.util.Timer`; and `playing` of type `boolean`. Then connect the following method to `playButton`:

```
private void runOrPause() {
  if (playing) {
    timer.cancel();
    playing=false;
    playButton.setText("Play");
  }
  else {
    playing=true;
    playButton.setText("Stop");
    timer = new Timer(true);
    timer.scheduleAtFixedRate(new TimerTask() {
      @Override
      public void run() {
        moveForward();
      }
    }, 0, 500);
  }
}
```

Test that the button works as expected on a few patterns.

Correct the functionality such that clicking a new pattern, or the back or forward buttons when already animating causes the animation to stop.