The Game of Life

In the game of life, cells exist in a 2-dimensional rectangular world under the following conditions:

- If a cell is alive:
    - It dies if it has fewer than 2 live neighbor cells
    - It dies if it has more than 3 live neighbor cells
    - It continues to live if it has 2 or 3 neighboring cells.
- If a cell is dead:
    - It comes to life if it has exactly 3 live neighboring cells.

These rules are applied simultaneously with each generation of the game.

Consider a GUI that uses JButton objects to represent cells in the world. These buttons will appear in a grid layout in the center region of the GUI. The GUI will also include a control panel with three buttons on it: Step (executes one generation of the game), Start (executes steps continuously until the button is clicked again) and Reset (restarts the game with all cells dead.

Cells can be brought to life by clicking them. Cell start out dead, but one click brings them to life. A second click kills the cell. This allows the user to create a pattern of living cells before the simulation is run.

Designing the project:

A Cell is a JButton. It has a state (living or dead) and this state is changeable, so the Cell class needs a setState method and an isAlive method. The Cell will display an "X" when it is alive and be blank when it is dead.

The LifeWorld is a JPanel that contains a grid of Cell objects. It can applies the rules of the game of life to the cell objects one generation at a time (step) or continuously (run) and must allow for resetting the world (reset).

The Controller is a JPanel that contains three buttons: Step, Start and Reset.

The LifeGUI is a JFrame with a LifeWorld object displayed in the center region of the frame and a Controller object in the south region of the frame.

The LifeWorld class:

There are four methods in the LifeWorld class:

- Reset – changes the state of every cell in the world to dead.
- Step – applies the rules of life to cells one time
- Run – uses a Timer object to invoke the step method repeatedly.
- Stop – stops the Timer object used in the start method.

The Timer object:

The Timer object issues an ActionEvent at regular intervals. Therefore the class that uses the Timer object must implement the ActionListener interface and therefore override the actionPerformed method.

The Timer class constructor takes two parameters: The time interval (in milliseconds) between ActionEvent objects being issued and the parent component that will handle the ActionEvents. That is, the class that will contain the actionPerformed method. So if this class implements the ActionListener interface and overrides the actionPerformed method, the Timer object might be crated as follows:

```
Timer t = new Timer(1000,this);
```

In the run method, the Timer object needs to be started:

```
t.start();
t.setRepeats(true);
```

In the stop method …

```
t.stop();
```

In the actionPerformed method:

```
public void actionPerformed(ActionEvent e)
{
     step();
}
```

The Controller class

The controller class has three buttons that control the LifeWorld object. Therefore, it must accept the LifeWorld object as a parameter of its constructor:

```
public Controller(LifeWorld world)
{
     this.world = world;
}
```

The button's ActionListeners must invoke the appropriate method of the LifeWorld object. For example, the actionPerformed method in the Step button's ActionListener is:

```
public void actionPerformed(ActionEvent e)
{
     world.step();
}
```

The step Method of the LifeWorld class

The step method applies the rules of the game of life simultaneously to the cells in the LifeWorld object. This is not a simple as it sounds. Consider this example:

	1	X	2	
	X	X	X	
	3	X	4	

According to the rules of the game of life, cells 1, 2, 3 and 4 should come to life (they have three live neighbors) and the middle cell should die (it has more than 3 live neighbors). However, if the rules are applied one after the other, the middle cell dies first …

	1	X	2	
	X		X	
	3	X	4	

… cells 1, 2, 3 and 4 now have only 2 live neighbors, so they do not come to life.

The problem is further complicated by the way object variables work. You might think a solution to the problem would be to create a copy of the world …

```
Cell[][] copy = world;
```

… and make the changes to copy, using world as a reference. However, since object variable contain only pointers to objects, the variable copy and the variable world will point to the same object. Thus changing one, changes the object.

You must create a second array of Cell objects, but each element of this array must be assigned a state that matches the state of the corresponding cell in the world array …

```
copy[r][c].setState( world[r][c].isAlive() );
```