1.00/1.001 Introduction to Computers and Engineering Problem Solving

Fall 2002

Problem Set 6

Due: Day 21

Problem 1. Simple Drawing (20%)

Create a class called `SampleGrid`, which extends `JPanel`. Write the following method within this class
`public void paint(Graphics g)`
First, use `fillRect` to draw the rectangle and fill the rectangle with the desire color (call `setColor` before `fillRect`). Make the rectangle to have equal length and width as 99 pixels. Then use another color to draw lines. Two horizontal lines and two vertical lines will separate the square into 9 even smaller ones. You would find the `drawLine` function useful for this part. All three of these functions, you could find them in the `Graphics` class.

Write another class called `SampleDrawing`, which extends `JFrame`. This class has one data field called `grid` (type `SampleGrid`). In the constructor, instantiate grid and add it to its content pane using `BorderLayout` as its layout manager. In `main()`, create a `SampleDrawing` object, set the size of this object to be big enough so that you can view the entire 3x3 grid, then disable the resizable frame by calling `setResizable(boolean)` function. Remember to make the execution of the program ends when clicking on the "x" button on the frame window.

Think about what you will do if you need to draw 10 horizontal lines and 10 vertical lines to make 100 smaller squares. This will be helpful in the next question.

Problem 2. Conway's Game Of Life (80%)

The Game of Life was invented by Cambridge Mathematician John Conway and became widely known in the 1970s. It is not a typical computer game, but rather a "cellular automaton."

The game setting is similar to that of solitaire—a collection of cells. A cell can be alive or dead. Depending on the initial conditions, the cells form various patterns. The rules of the game are as follows:

A cell with three "live" neighbors becomes or stays alive.

A live cell with two live neighbors stays alive.
In every other situation, the cell dies (of overpopulation or by loneliness) or remains dead.

Note: At generation (t+1), the state of a cell depends on the state of its neighbors at generation t.

In this problem, to simplify the coding, for every cell, it will have 8 neighbors. We achieve this by making the top edge connects with the bottom edge and the right edge connects with the left edge. If we have a 5x5 grid like the following figure, most of the cells will be like (1, 1) and its neighbors will be those cells in the 3x3 grid with (1, 1) as the center cell.

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)

But for cells that are on the corner, like (0, 0) for example, its neighbors will be (1, 0), (1, 1), (0, 1), (0, 4), (1, 4), (4, 0), (4, 1), (4, 4). For the cells on the edge like (4, 2), its neighbors will be (3, 1), (3, 2), (3, 3), (4, 1), (4, 3), (0, 1), (0, 2), (0, 3).

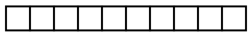
For this problem, you are going to write three classes, they are:

`Cell.java`, which is a class that represents a single cell. It has two data fields: one is a boolean that determines the state of the cell, and the other is an array that keeps tracks of its neighbors' states.
`GridPanel.java`, which is a class that extends JPanel. It has five data fields. They are three integer data fields which keeps the number of rows and the number of columns of cells, the size of the cell (we will assume here all cells are squares, and the size is the length of the square measured in pixels), and one Cell array (a two dimensional array) to keep the current state of all cells, and another Cell array to keep the previous state of all cells.
`GameOfLife.java`, which is a class extends JFrame. The **`main()`** program runs the life game.

In the **`Cell`** class, besides the constructor, also write the methods that get the data fields and set the data fields. Write a method that determines if a cell is going to be alive in the next generation using the game rules.

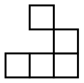
In **`GridPanel`**, write a constructor that takes in three integers that will set the value for the three integer data fields. The constructor also instantiates and initializes the two Cell arrays. Change the **`paint`** method that you wrote in problem 1 to fit this problem. Next write a method that uses the game rules to determine what the next generation grid will look like. Write four possible initial states that will have the following different figures(they would look prettier if you put these initial figures around the center of panel):

1. a 10 cell row



2. a blinker: a 3 cell row or 3 cell column

3. a glider



4. random

   make each cell to have 50% chance of being alive (you will find the **`Math.random()`** function useful)

In the **`GameOfLife`** class, put in the buttons/labels/text fields necessary for this problem. Include a "Next" button that when you click on it, it will display the next generation. Also include a **`JComboBox`** that allows you to choose the different initial states. You could also choose to display the generation number(this is optional). Create a grid of 50x50 with each cell size as 10 pixels. Call on the **`setDefaultCloseOperation`**, and make the program exit when the "x" button on the window is clicked.

In short, your classes should look like the following, but you don't need to make or name your data fields or methods the same.

```
//Cell.java
public class Cell {
    private boolean alive;
    private Cell[] neighbors;

    public Cell() { … }
    public Cell[] getNeighbors() { … }
    public void setNeighbors(Cell[] c) { … }
    public boolean isAlive() { … }
    public void setAlive(boolean b) { … }
    public void liveOrDie(Cell prevGen) { … }
}

//GridPanel.java
public class GridPanel extends JPanel {
    private int cellSize;
    private int gridCol;
    private int gridRow;
    private Cell[][] cells;
    private Cell[][] prev;

    public GridPanel(int row, int col, int size) { … }
    public void setTenCellRow() { … }
    public void setBlinker() { … }
    public void setGlider() { … }
    public void setRandom() { … }
    public void paint(Graphics g) { … }
    public void nextGeneration() { … }
}

//GameOfLife.java
public class GameOfLife extends JFrame {
    private Container conPane;
    private JLabel select;
    private JPanel buttonHolder;
    private JButton nextButton;
    private JComboBox choice;
    private JLabel genLabel;
    private JTextField genField;
    private int generation;
    private GridPanel grid;
    private static String[] fig = { … };

    public GameOfLife() { … }
    public static void main(String args[]) { … }
}
```

Want Some Challenge? Do the Extra Credit…

Make a cool addition to the life program or interface. The best addition will be demonstrated in class. This extra credit problem is not worth any points, but will get you glory.

Turnin

Turnin Requirements

- Hardcopy and electronic copy of ALL source code (all .java files). Remember to comment your code, points will be taken off for insufficient comments.
- Place a comment with your name, username, section, TA's name, assignment number, and list of people with whom you have discussed the problem set on ALL files you submit.
- Do NOT turn in electronic or hardcopies of compiled byte code (.class files).

Collaboration

For this problem set, you may work together with at MOST one other person in the class. If you choose to work with a partner, you must include both of your names on your submission. (If you have different TAs, be sure you write both TAs' names on your PS.) You will not be allowed to add the name of your partner after submitting your problem set. Only submit the problem set ONCE (choose either person). Both you and your partner will get the same grade. Your partner can be your official partner or someone else officially n the class(including listeners). Only 1 or 2 member teams, no exceptions.

Electronic Turnin

Use *SecureFX* (or another secure ftp or secure shell program) to upload your problem set to your 1.00 homework locker.

Detailed instructions of how to upload are on the course website.

Since your problem set is due at the beginning of lecture, your uploaded problem should have a timestamp of no later than morning on the due date.

Penalties

- Missing Hardcopy: -10% off problem score if missing hardcopy.
- Missing Electronic Copy: -30% off problem score if missing electronic copy.
- Late Turnin: -30% off problem score if 1 day late. More than 1 day late = NO CREDIT.

If your problem set is late, or if a professor has granted you an extension in advance, do not submit a printed copy of your problem set.