cs205: engineering software
http://www.cs.virginia.edu/cs205/ps/ps1/
23 August 2006

Problem Set 1

Game of Life

Out: 23 August
Due: **Monday, 28 August**
(beginning of class)

Collaboration Policy - Read Carefully

For this problem set, you should work alone, but feel free to ask other students for help and offer help to other students.

Purpose

- Learn basics of programming in Java
- Get familiar with the Eclipse integrated development environment (IDE)
- Provide some exposure to class interactions and object oriented design

Background

Cellular automata are computer simulations inspired by biological cells. An automata simulation starts with a grid of cells, where each cell is initialized to a particular state. At each step in a simulation, the next state of each cell is determined based on simple rules that depend on its current state and the current states of its neighbors.

Despite their simplicity, cellular automata can produce complex and interesting patterns. We have provided you with code that implements a cellular automata simulator. You are encouraged to look over the code, but you do not need to understand this code now and should not be concerned if it doesn't all make sense yet as it uses concepts that we will not encounter until later in the course.

The cellular automata simulator takes a Java class that defines the rules for each step, and simulates a grid of cells. Each cell can be in one of two states: *dead* or *alive*. A dead cell is shown as a white square and a live cell is shown as a green square. Each cell has neighbors which are the eight cells that surround it.

Although this is a fairly small program, there are many different ways it might be done. The provided implementation uses many programming techniques and Java features that we will not cover until later in the class including subtyping, parameterized types, interfaces, inner classes, exceptions, event handlers, and threads. It is not necessary (or expected) for you to understand all of the provided code for this assignment. This document will explain the most important parts of the code you need to understand for this assignment. You are encouraged to be curious about the rest of the code, and feel free to explore it and make changes to it, and think about better ways to implement this program.

Running the Simulator

Download *`http://www.cs.virginia.edu/cs250/ps/ps1/code/ps1.zip`* to your home directory. Create a `J:\cs205` directory and extract all files in the zip file to it (this will create the `cs205\ps1` subdirectory).

The following steps will create an Eclipse project containing the provided source code for Problem Set 1:

1. Open Eclipse by clicking on `Start | SEAS | Java Apps | Eclipse`
2. Inside the Eclipse window, click on `File | New | Project`. The *New Project* dialog box will appear.
3. Select the first option: `Java Project` and click `Next`. The *New Java Project* dialog box will appear.
4. In the `Project name:` box enter, `ps1`. In the `Contents` pane, select `Create project from existing source`, and select the directory where you extracted the ps1 code (`J:\cs205\ps1`). Click `Next` and then `Finish` on the next dialog box.

Now, you should see the *Package Explorer* view, containing the `ps1` folder. Click on the folder to explore the files.

Eclipse automatically compiles Java projects when they are imported. Also, whenever you make a change to a Java source file and save it, the changes are automatically compiled. If you want to force Eclipse to rebuild the entire project, select `Project | Clean`.

Try running the cellular automata simulator using the provided `ExtremeLifeCell` class to model each cell. To do so:

1. Select `Run | Run` from the Eclipse menu bar.
2. Select `Java Application` as the configuration on the left. Select `CellAutomata` for the main class.
3. Click on the `Arguments` tab and enter `ExtremeLifeCell` into the program arguments text window.
4. Click on the `Run` button on the bottom right of the window.

Eclipse will run the Java virtual machine, which starts by calling the `main` method of the `CellAutomata` class. The command line arguments (`ExtremeLifeCell`) are passed as a parameter to the `main` method, which loads the `ExtremeLifeCell` class and initializes the grid to contain an `ExtremeLifeCell` object in each square.

When the simulator starts, click on a cell and wait for it to turn green. Cells that are *alive* appear green; non-alive cells are displayed in white. Then press start and observe what happens. You can also use `Step` to run the simulation one step at a time. Try to figure out what rules the `ExtremeLifeCell` is following just by observing its behavior. (You'll see the code later.)

The Cellular Automata Simulator

The implementation of this cellular automata contains modules for providing the graphical user interface, maintaining a grid of cells, and implementing the rules for a given cell. Here we provide a description of some of the most important classes, but you do not need to understand the details of how these classes are implemented.

`CellAutomata.java`

> This is the main class for the implementation of the cellular automata. It sets up the user interface using Swing. Swing provides classes for displaying windows and handling input events.
>
> The `CellAutomata.main` method initializes a grid of cells, each containing an object instance of the class passed in as the first parameter to main. Then, it displays the grid. The `actionPerformed` method makes the appropriate things happen when the user clicks on the `Start`, `Stop`, and `Step` buttons.
>
> To load a specific cell class, you specify the cell class as a program argument when running the Java application. To test the class your create, replace `ExtremeLifeCell` with the name of your cell class. (Note that the `.class` extension should not be included.)

`CellGrid.java` — This class keeps track of all the states of the cells on the board. The most important method of the `CellGrid` class is `step` which excutes one step in the simulation:

```
public void step() {
   // Because we need to update all cells synchronously, we
   // first calculate the next state for each cell, and store it
   // in a temporary array:

   CellState[][] nextStates = new CellState[x_max + 1][y_max + 1];
   for (int x = 0; x <= x_max; x++) {
      for (int y = 0; y <= y_max; y++) {
         Cell cell = (Cell) getObjectAt(x, y);
         nextStates[x][y] = cell.getNextState();
      }
   }

   // Then, we update all the cells:
   for (int x = 0; x <= x_max; x++) {
      for (int y = 0; y <= y_max; y++) {
         Cell cell = (Cell) getObjectAt(x, y);
         cell.setState(nextStates[x][y]);
      }
   }
}
```

`CellState.java` — implements a datatype for recording the state of a cell:

```
import java.awt.Color;

public class CellState {
   // OVERVIEW: A CellState is an immutable object that represents
   //    the state of a cell, either alive or dead.

   private boolean alive;

   private CellState(boolean isalive)
   // EFFECTS: Initializes this to alive if isalive is true,
   //   otherwise initializes this to the dead state.
   {
      this.alive = isalive;
   }

   static public/* nonnull */CellState createAlive()
   // EFFECTS: Returns an alive cell state.
   {
      return new CellState(true);
   }

   static public/* nonnull */CellState createDead()
   // EFFECTS: Returns a dead cell state.
   {
      return new CellState(false);
   }

   public Color getColor()
   // EFFECTS: Returns the display color for this state
   {
      if (alive) return Color.green;
      else return Color.white;

   }

   public boolean isAlive()
   // EFFECTS: Returns true iff this is alive.
   {
      return alive;
   }
}
```

The methods with `static` modifiers are class methods, used to create new states. They are not invoked on an object (there is no `this`). Instead, invoke them using `CellState.createAlive()` and `CellState.createDead()`.

`Cell.java` — This class provides a basic Cell implementation. It includes the methods specified below:

```
public class Cell extends SimObject {
   private/* nonnull */CellState state;

   public Cell()
   // EFFECTS: Initializes this to a dead cell.

   public/* nonnull */java.awt.Color getColor()
   // EFFECTS: Returns the display color for this.

   public boolean setState(/* nonnull */CellState s)
   // MODIFIES: this
   // EFFECTS: Sets the cell's current state to s. Returns true iff the new
   //    state is different from the current state.

   public/* nonnull */CellState getState()
   // EFFECTS: Returns the cell's current state.

   boolean isAlive()
   // EFFECTS: Returns true if the cell is alive, false otherwise.

   public/* nonnull */CellState getNextState()
   // REQUIRES: init has previously been called for this.
   // EFFECTS: Returns next state value for this.
   // Note: For the underlying Cell class, the next state is always the
   //   current state.  Override this method to implement
   //   different cell automata rules.
}
```

Producing Cell Behaviors

To produce cellular automata with interesting behavior, you will create a subtype of the `Cell` class and override the `getNextState` method to do something more interesting. The `ExtremeLifeCell.java` class implements a cell that follows the Extreme Life rules. We use `extends Cell` to indicate that the `ExtremeLifeCell` class is a subtype of `Cell`. This means it inherits the non-private methods from the `Cell` class. To implement a new cell behavior, we need to provide a new implementation of the `getNextState` method. Here is the `ExtremeLifeCell` class:

```
     public class ExtremeLifeCell extends Cell {
        public CellState getNextState()
        // EFFECTS: Returns the next state for this cell.
        //          The next state will be alive if this cell or any of its neighbors
        //          is currently alive.
        {
           Enumeration neighbors = getNeighbors();
           while (neighbors.hasMoreElements()) {
              SimObject neighbor = neighbors.nextElement();
              if (neighbor instanceof Cell) {
                 Cell cell = (Cell) neighbor;
                 if (cell.isAlive()) {
                    // If the cell has at least one neighboring cell
                    // that is alive, this cell should become alive.
                    return CellState.createAlive();
                 }
              }
           }

           // No alive neighbor found, next state is current state
           return getState();
        }
     }
```

The `getNextState` method replaces the `Cell.getNextState` method. The `while` loop goes through the cell's neighbors (returned by the call to `getNeighbors()` (which is defined in `SimObject`, and inherited through the `Cell` class). If a neighbor is found that is in the alive state (`cell.isAlive()`), it returns immediately with the result an alive state. If the loop finishes without finding an alive neighbor, the cell maintains its current state.

Conway's Game of Life

John Conway, a Cambridge mathemetician, invented the game of life with these simple rules:

1. If a cell is currently dead cell and has three live neighbors, then it becomes a live cell.
2. If a cell is currently alive and has two or three live cells it remains alive.
3. Otherwise, the cell dies.

1. Using the `ExtremeLifeCell.java` code as an example, produce a `ConwayLifeCell.java` file containing a class that implements the Conway's Game of Life cellular automaton. To create a new class in Eclipse, select `File | New | Class` fron the Eclipse main menu, and enter the class name (`ConwayLifeCell` in the `Name` field. Change the superclass to `Cell`. Eclipse will generate a stub class file for you. You will need to provide an implementation of the `getNextState` method that implements the Conway's Game of Life rules.

With these simple rules, it is possible to produce many interesting patters. For example, if you start with three cells in a row it should produce a blinker that alternates between three horizontal and three vertical live cells. Experiment with different initial conditions to see what happens.

2. Modify the program so that instead of dying instantly, cells go through a *dying* state where they are displayed in a different color for one step before they die. (If you are more ambitious, feel free to make additional modifications and improvements.)

Turn-in Checklist: You should turn in your answers on paper at the beginning of class on Monday, 28 August. For question 1, turn in your `ConwayLifeCell.java` code. For question 2, describe the changes you made and turn in all the code you changed.

Related links:

- Wonders of Math - The Game of Life
- Scientific American article (October 1970): Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"

Credits: This problem set was originally developed for UVa CS 201J Fall 2002 by Mike Peck, Tiffany Nichols and David Evans, revised for UVa CS201J Fall 2003 by Mike Peck, John Franchak and Leonid Bolotnyy, and revised for UVa CS205 Fall 2006 by David Evans.

evans@cs.virginia.edu