ASSIGNMENT 2

Due date: 5pm, Wednesday 4th of November 2015

Assignment marks: 20% of overall unit marks.

Objective: Develop your skills of working with a shared code base by refactoring and extending existing code. Practice writing code which conforms to a specific coding and documentation style. Come to grips with some simple threaded animation code.

This assignment primarily assesses your progress relative to learning outcomes 1 - 7 of the unit guide. We will be placing emphasis on learning outcome 3, so when marking this assignment we will be paying particular attention to:

- the quality and clarity of the code you write,
- the extent to which you have applied the coding and variable naming standards of the existing code base you will be working on,
- the quality, clarity and completeness of your Javadoc commenting, and
- the extent, relevance and quality of the (automated) tests that you have applied to your code.

To encourage you to start working on this assignment in good time, and to help by providing you with feedback along the way, we have already asked you to develop a solution to an intermediate problem (Conway's Game of Life) in the mixed classes of weeks 10 and 11. If you have not already completed this exercise then you should do so before attempting this assignment.

Please note: This assignment specification aims to provide as complete a description of this assessment task as possible. However, as with any specification, there will always be things we should have said that we have left out and areas in which we could have done a better job of explanation. As a result, you are strongly encouraged to ask any questions of clarification you might have, either by raising them during a lecture or a mixed class or by posting them on the iLearn discussion forum devoted to this assignment.

Investigating Cellular Automata for Fun and Profit.

A *cellular automaton* consists of a regular (finite or infinite) grid of cells laid out as a multi- (often 2-) dimensional array. Each cell can contain one of a finite number of different values, such as `off` and `on` of the numbers from `0` to `15` for example. Furthermore, each cell is surrounded by a finite set of cells, called its *neighbourhood*, defined relative to (and possibly including) that cell.

A cellular automaton *evolves* through time, which we think of being broken up into a sequence of discrete time steps or *generations* `t = 0, 1, 2, ....`. It is initialised with a given starting state or *configuration*, specifying the initial state of the automaton at generation `0`. It then evolves from one generation to the next according to some simple *updating rules*, which specify the contents of a cell in generation `t+1` in terms of the contents of its neighbourhood in generation `t`.

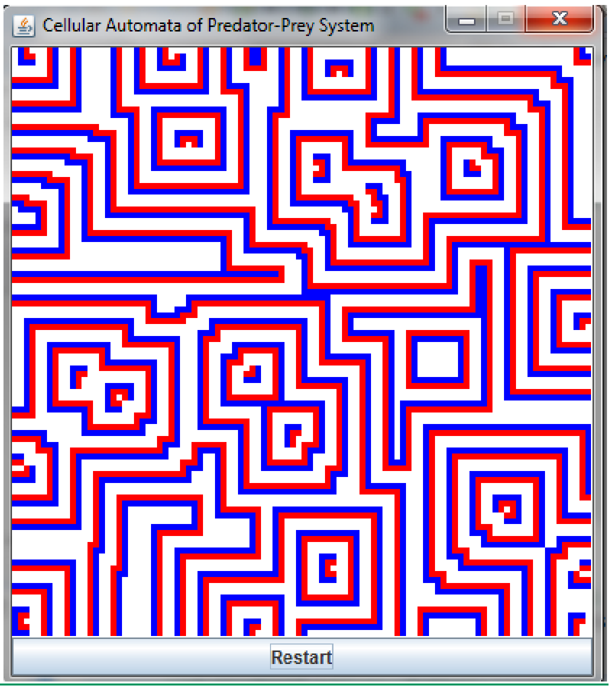
Cellular automata probably started life as a *recreational* pass-time amongst mathematicians. However, in recent years they have been used to model a wide variety of important physical phenomena and have even been discussed as a possible foundation for physics itself. Some people have even suggested that the universe itself may be a cellular automaton ;)

You can find out more about cellular automata, you might like to consult the Wikipedia article on that topic.

Spiral formation in cellular automata of predator-prey systems

Cellular automata are "games" (or "simulations") in which various entities in the squares of a grid are in each turn of the game updated according to fixed rules, such that the new state of a square depends only on the state of the square plus a set of neighbouring squares in the previous turn. The famous "Life" game of Conway is an example.

In various types of cellular automata systems, it is found that "spiral" patterns appear in the simulated entities, where these spirals rotate around points that pretty much remain in the same place, and where a rotating spiral pattern, once formed, remains "alive" during a large number of turns. The picture bellow shows an example of such spirals. This is a predator-prey model from population ecology. The prey are blue, and the predators are red. The prey and predators each reproduce; prey reproduce and die at



certain rates. Predators consume prey.

To give you an idea of what we expect, we've written an implementation of the kind of thing we are looking for. You can download (the executable version of) this sample solution from the COMP229 iLearn site. It is contained in a "jar" file called Spiral.jar.

Implementation predator pray cellular Automaton

These are the list of classes we want you have in your assignment.

Animatable Interface

This interface should be implemented by any class which is intended to represent the state of an animation as it evolves from frame to frame.
 Note that the step() and paint() methods should generally be synchronized, because they are called from different threads. Specifically, step() is called from the animation thread and paint() is called from the event dispatch thread.

```
 * only be called from within the event dispatch thread.</p>public interface
Animatable {

        /** Step the animation state forward by one frame.*/

        public void step();

        /** Paint the current state of the animation onto the graphics canvas.*/

        public void paint(Graphics pGraphics);

        /** Get the width of this animation in pixels.*/

        public int getWidth();

        /** Get the height of this animation in pixels.*/

        public int getHeight();

}
```

Animator Class

The Animator class provides the basic functionality to display the state of an animation and to step that animation forward at a regular frame rate. This class extends JPanel, to provide a component upon which the animation is drawn. It also implements Runnable in order to provide a thread which periodically steps the state of the animation to the next frame and calls repaint() to persuade the event dispatch thread to paint that frame.

The state of the animation is actually managed by an aggregated object of a class which implements the Animatable interface. This provides methods by which the state may be stepped or painted.
To ensure thread safety, the methods of this class, except for run() which shouldn't be called directly anyway, should only be called from within the event dispatch thread.

Cellular Automaton

This class implements Animatable and it provides an abstraction of a cellular automata as a two dimensional array. Each elements of the board denotes a cell which can be Blue,Red or White.

Main Pannel

A panel of this type encapsulates the contents of the top level window of this application. This includes the animator panel itself and a button for restarting the animation. Such a panel may either be used as the content pane of a JFrame, if our game is to run as an application.

Position
Objects of this class describe positions on a life board as a pair of a row and a column number.

Main
The main class of this application, sets up the contents of the main window and starts the application running.
This class allows this program to be run as a standalone application, started via the main(String[])method. The actual hard work of constructing and laying out the GUI is delegated to the MainPanel class.

Starting and Stepping the Automaton

- You should choose a grid size, my implementation has 100 * 100 cells.
- Each square contains either a prey, or a predator, or is empty. In the above picture you can see predators as red cells, prey as blue cells and empty (dead) cells as white cells.
- Initially, fill each square with probability $\text{Pinitial}_{\text{prey}} =$ 1/4 with a prey, with probability $\text{Pinitial}_{\text{predator}}$ = 1/4 with a predator, and with $\text{Pinitial}_{\text{dead}}$ = 1/2 leave it empty.
- As in very many cellular automata, the "neighbours" of a cell are defined as the 8 neighbouring cells.
- As in all cellular automata, the simulation proceeds in turns. Each turn, apply the following algorithm on a separate, initially empty, grid the new generation from the current generation.
- At each step of the automaton you should scan through the board and update its cells using the following algorithm (expressed in pseudo code):

```
for each location (r, c) on the board
    in generation t
        1. if (r,c) containing a prey:
                1a. fill the corresponding cell in the new generation (t+1)
                    with a prey.
                1b. for all neighbouring cells of (r,c) that are EMPTY in
generation t
                        With probability PPrey, fill this empty cell in the
                        new generation (t+1) with a new prey.  (And with
                        probability 1-PPrey leave it empty).
                        //(Host offspring grows with “speed” PPrey into
                        //neighbouring empty cells.)
```

```
for each location (r, c) on the board
    in generation t
        2. if (r,c) containing a predator:
                2a. fill the corresponding cell in the new generation (t+1)
                    with a predator.
                2b. for all neighbouring cells of (r,c) that are prey, do :
                        With probability PPredator, replace the contents of that
                        neighbouring square in the new generation with a
                        predator.  (And with probability 1-PPredator leave the
                        cell intact.)
                        //(Parasite eats host and multiplies at host's
                        //expense, with ''speed'' PPredator).
```

```
for each location (r, c) on the board
        in generation t+1:
        3. Remove all predators that do
           not have a prey in at least one neighbour cell.
           (Predators not directly in contact with a prey on which they
           feed, die.)
```

To get clearly pronounced spirals, PPrey and PPredator must be close to 1, say between 1 and 0.85. As PPrey and PPredator get smaller, the spirals become more "fuzzy", and for very small PPrey and PPredator the visual appearance of the evolving simulation approaches the "turbulence".

For PPrey =PPredator = 1, when the computation of the next state from the old state involves no probabilities and is therefore entirely deterministic (and the only random thing is the initial situation), we definitely get very clearly pronounced spirals. **The image above is produce by the PPrey =PPredator = 1.**

If you want to find the reason for spiral formation for this system please check this site.

This assignment is quite easy if you have the game of life working already, so focus on getting that working and then start on the changes for this assignment.

What you should do for a PASS / CREDIT grade.

To obtain a PASS / CREDIT in this assignment you should:

Start with either of:

- Game of Life, which will be explained in the lecture and tutorial classes.
- Your own Game of Life code, which was probably adapted from the Bouncing Head code base.

You'll probably want to keep your automaton board quite small (maybe no more than 100x100) otherwise you code may run slowly.

Write a small suite of automated JUnit tests to test a few aspects of the code you have written.

What you should do for a DISTINCTION grade.

Provide the user with buttons to run, pause and step the simulation from one generation to the next, just like Game of Life does.

Provide input facilities which allow users to specify the probability of the predator and prey.

What you should do for a HIGH DISTINCTION grade.

You will discover that if your automaton has many cells then it will take a long time for a single step of the automaton to be calculated and displayed. The sample application handles this situation by doing the actual updating of the automaton board in a separate thread. This executes a loop which randomly selects cells on the board and updates them in the background. That worker thread also updates a "back-buffer" image (of type java.awt.image.BufferedImage) as it goes, so that when the automaton's `paint(Graphics)` method is called all it need do is draw that back-buffer image onto the animators graphics pane.

Of course, to ensure thread safety this painting has to be `synchronized` with the image updating undertaken in the worker thread. But hardware accelerated image blitting is so fast that this doesn't tend to hold up the worker thread much.

If you are aiming for a HIGH DISTINCTION grade you should work to expand the size of your automaton board while still having a responsive animation. Do this by introducing an extra incremental / asynchronous "worker" thread to handle cell updates and to incrementally draw the state of the animation to a back-buffer.

Alternatively, you might like to come up with some further interesting bells and whistles of your own to enhance your application. You may even want to experiment with other kinds of cellular automata you've read about on the web.

Recall that Macquarie's assessment policy says that HIGH DISTINCTIONs are awarded when individuals show *"substantial originality and insight in identifying, generating and communicating competing arguments, perspectives or problem solving approaches; critical evaluation of problems, their solutions and their implications; creativity in application as appropriate to the discipline"*. So to a great extent we're looking to you to come up with an idea for your own HD.

Expected effort.

In general, a 3cp unit of study at Macquarie is rated as about 150 hours of work (including lectures attendance, exam revision, assignment and tutorial work...) scattered throughout the semester (17 weeks). This assignment is worth 20% of the overall unit marks and so it is probably worth expending about 20% of your total COMP229 study time on. That means that you should expect to spend around a total of 30 hours on writing your code and documenting your work in a report.

You are strongly advised not to attempt to solve all of this assignment in one go. Rather, you should solve each part and get it working before you go on to the next part. It is also important to start working on this assignment *right now*, even if you only devote an hour or three over the first 10-14 days to thinking carefully about what this task might entail and marshalling the resources you will need to complete it.

Mark allocation

Within each part, the marks for this assignment will roughly be allocated in the following proportions:

- 60% for code functionality and the quality of your code, in terms of how closely it matches the standards of the original code skeleton and those discussed in the lectures.
- 40% for automated testing of the code you have written and the quality and thoroughness of your javadoc documentation.

Mark modifiers

If any of the following statements is true of your code, you will lose marks (for each):

- nonsensical architecture
- single-letter variable names for anything but iteration variables
- program crashes during testing (it will be re-run but you will have a penalty).
- Not including a reference to code you have found elsewhere (except course texts/readings).

Double-check your submission because if it does not compile, you will receive zero marks.

Submission Instructions

- You must submit a zipped Eclipse project containing all the files needed to compile and run your program in Eclipse. Your submitted files must have your student number as it's name and it must be a .zip file (not a tar file or a rar file, or any other type of archive, zip only.)
- To test if your submission will act this way, try importing it into a clean version of Eclipse on the lab computers before you submit it.