Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

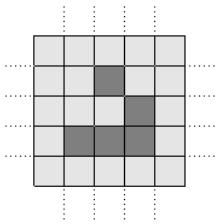
Lab Handout

Outline

The purpose of this lab is to use inheritance as a mechanism for extending existing code bases, and to implement code which makes use of subtyping. You will also learn about the `javadoc` tool. Before the end of the lab, you should submit your solutions via the *online submission system* which will automatically mark it. You may submit as many times as you like in order to improve your mark and the final deadline will be Friday @ 23:59.

Conway's Game of Life

Conway's *Game of Life* is a simple cellular simulation devised by John Horton Conway (a British mathematician) in 1970. The Game of Life has been studied extensively since then for both scientific interest, and also just for fun. For example, it has been shown that the problem of deciding whether a given state of the game will ever reach the empty board is *undecidable*. A simple example is:



Here, cells marked in black are "alive" whilst those in white are "dead". Conway identified four rules for governing the transition between life and death in the Game of Life:

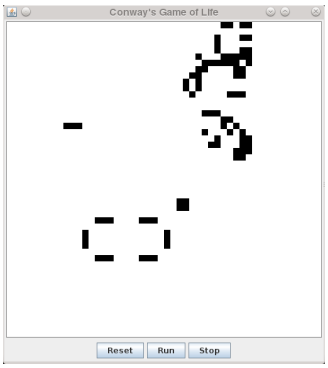
1. Any live cell with fewer than two live neighbours dies, as if caused by *under-population*.
2. Any live cell with exactly two live neighbours continues on to the *next generation*.
3. Any live cell with more than three live neighbours dies, as if by *over-population*.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by *reproduction*.

Here, the neighbours of a cell are the *eight* surrounding cells (i.e. the four cells at *North*, *South*, *East*, and *West* and the four diagonals at *NorthWest*, *NorthEast*, etc). See `en.wikipedia.org/wiki/Conway's_Game_of_Life` for more.

In this lab, we are going to play with an extensible implementation of the Game of Life. The implementation is extensible because new rules can easily be written to control the game in different ways. The implementation also supports more complex forms of the Game, as it allows for more than just cells which are "alive" or "dead". Instead, each cell is in a state between 0 and 9, where 9 corresponds roughly to "dead" and the other states to varying forms of "alive". The meaning of the intermediate states is left up to the rule implementor, but can be used to simulate *age*, *sickness*, *happiness*, etc.

Getting Started (approx 20 mins)

To get started, download the `conway.jar` file from the course website and import this into an Eclipse project. The Game of Life is provided with a Graphical User Interface, which you can run by right-clicking on `swen221.conway.GameOfLife` and selecting "Run As⟶Java Application" (ignore `CellDecayGameOfLife` for now). You should see a window pop up like this:



You can play the game by clicking on the window to place cells and the starting the simulation by pressing "run". Take a few minutes to play around with the game. You should notice that not all rules are implemented. In Activity 2, you will return to complete the rules for the game.

Activity 1: JavaDoc (approx 10mins)

In this activity, you will use the `javadoc` tool to generate documentation for the Conway program. Begin by reading the `javadoc` tutorial available from here:

`http://ecs.victoria.ac.nz/Support/TechNoteJavadoc`

At this point, start by generating the `javadoc` documentation for the Conway program as given out. To do this in Eclipse, select the project and then choose "Project⟶Generate Javadoc..." from the menu. Once the tool has completed, you can open the documentation to view in your web browser.

To complete the activity, you will write some `javadoc` documentation yourself. Observe that the class `swen221.conway.rules.ConwaysUnderpopulationRule` has no `javadoc` comments. Add a `javadoc` comment for the class which explains that it implements Conway's underpopulation rule for cell's with fewer than two neighbours (see above). Add an appropriate comment for the `apply()` method or simply mark it with `@Override` (in which case `javadoc` pulls any documentation for the overridden method). Again, rebuild the documentation by running `javadoc` and check the results in your browser.

Activity 2: Conway's Missing Rules (approx 30mins, worth 50%)

The implementation of the Game of Life provided does not properly implement rules (3) and (4) from above. The goal of this activity is to complete the following classes:

`swen221.conway.rules.ConwaysReproductionRule`
`swen221.conway.rules.ConwaysOverpopulationRule`

The current code is just a stub and in order to complete this part you can modify it as you need. You should find these classes are essentially empty and do nothing useful. The current code of those rules implements the `Rule` interface. On the other side, fully implemented rules like `ConwaysUnderpopulationRule` extends `ConwayAbstractRule`: an utility class that helps to implement rules. Indeed, the given implementation of `ConwaysUnderpopulationRule` should provide a useful guide which you can follow. Having done this, you should find that all tests in `GameOfLifeTests` now pass.

Activity 3: Cell Decay (approx 30mins, worth 50%)

The goal now is to implement a variation on the basic rules for the Game Of Life which models the age a cell. Our updated rules for the game are:

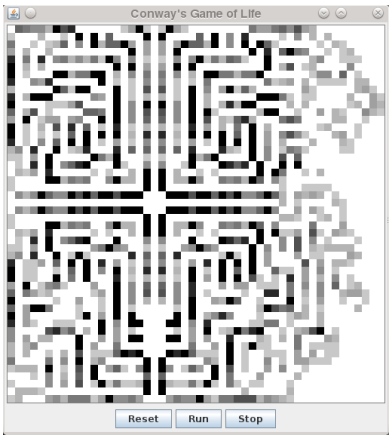
1. Any live cell with fewer than two live neighbours gets *older*, as if caused by *under-population*.
2. Any live cell with two live neighbours continues *in stasis* to the *next generation*.
3. Any live cell with more than three live neighbours gets *older*, as if by *over-population*.
4. Any cell with exactly three live neighbours gets *younger*, as if by *happiness*.

The age of a cell is determined by its current state which, if you recall, is a value between $0 \ldots 9$. Here, 0 is youngest whilst 9 is oldest. We consider that cells of age 9 are *dead*, whilst all others are varying forms of *alive*.

At this point, we will now concentrate on using the variation `CellDecayGameOfLife` and the accompanying test cases, `CellDecayTests`. You can run the simulation as a Java Application with `CellDecayGameOfLife`, though at this stage you should find that it does *nothing*.

What to do. Your aim is to develop one or more instances of `Rule` which implement the above rules for cell decay. You should add any rules you create to the `CellDecayRules` array in `CellDecayGameOfLife`, in order that `CellDecayGameOfLife` and `CellDecayTests` can use them. Having done this, all tests in `CellDecayTests` should now pass.

Having completed this part, you should find that the balance has tipped and cell growth expands *quickly*. This is because cells stay "alive" for much longer than before. For example, the system will often produce something like this:



Submission

Your lab solution should be submitted electronically via the *online submission system*, linked from the course homepage. The minimum set of required files is:

```
swen221/conway/rules/ConwaysUnderpopulationRule.java
swen221/conway/rules/ConwaysOverpopulationRule.java
swen221/conway/rules/ConwaysReproductionRule.java
swen221/conway/util/ConwayAbstractRule.java
swen221/conway/model/Board.java
swen221/conway/model/BoardView.java
swen221/conway/model/Rule.java
swen221/conway/model/Simulation.java
```

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.