Scotty Shaw (sks6)
Cell Society – Team 19
Computer Science 308

Project Journal

For Cell Society, I estimate having worked 60 hours, with 40 focusing on specifically coding, sometimes at our team meetings. Implementing simulations and hierarchies was easier, whereas refactoring was a significant learning experience. Office hours with Yu Zhou Lee and Dr. Duvall helped me understand and implement good design, which also offset the days I lost trying to find a TA who was available to help, despite being assigned to other teams.

Collectively, our team worked seamlessly and even peer programmed before we learned about it. As a result, we were often involved in each other's commits. We met for around 4 hours every school night and probably worked over 100 total hours, if each team hour counts as 3. We quickly conceptualized a code design, but unfortunately, it was too simple because we underestimated the size of this project. As a result, our original design document does not accurately describe our final submission. However, our original model of using three modules (reading input, simulating games, and a Cell hierarchy) served as an initial basis for our last commit.

Chris programmed most of our front end, such as the GUI and its various features. He was also extremely important in our refactoring process. When we implemented a Cell Factory, he accidentally used reflection. Marcus focused mostly on the XML-related files, and we often worked to integrate information written in the XML files into our cells.

My responsibilities were to implement the various simulations, which led to the Cell, Edge, Rules, and Grid hierarchies and a Cell Factory when I led our massive refactoring efforts. Combined with guidance from Dr. Duvall and Yu Zhou Lee, I was able to significantly reduce our amount of repeated code and create what I consider to be a quite sophisticated program. If our team were to have more time, we are confident that we would have implemented all features other than tessellation and an infinite grid. Since most of our second week focused on extracting hierarchies and refactoring our code, I feel that I learned a lot about maintaining good design by working on our cellTypes and edgeTypes packages.

As for non-coding responsibilities, I served as our Git Master since I was already familiar with Github, and I smoothly and easily resolved all Github-related issues. This is definitely a credit to our teamwork. We communicated frequently and did not have any issues involving teamwork. Other than occasional merge conflicts, some of which were intended, we worked seamlessly and had no problems. Chris and Marcus are excellent teammates, and I highly recommend them to anyone.

Regarding my individual Github efforts, my account lists nearly 60 commits out of 132 on our master branch. This does not include the 4 branches I created, nor does it fully reflect on any of our contributions. I also participated in all commits, regardless of the associated account, involving any simulations, hierarchies, and refactoring through peer programming. But I would not hesitate to crown Chris as our MVP. He was our strongest programmer and very capably led our team.

As for my individual contributions on Github, I always tried to commit only if our code ran properly after some modification and maintenance. As a result, my commits range from the simple (editing our code for readability, spelling, and neatness) to the massive, such as implementing the entire Edges hierarchy in a separate package or the full Game of Life simulation. The majority of my commits focused on eliminating some form of repeated code (such as blocks of if statements or override methods common to multiple subclasses) and resolving all Github issues so that Chris and Marcus could focus solely on their aspects of Cell Society.

As for commits of note from my account, there are three that stand out in my mind. The first one, also from September 16, is the earliest commit I know of from our team that was completely individual. Inspired by Austin Kyker's account of Zach Bears implementing Conway's Game of Life in 75 minutes, I decided to copy his feat on my own. It took me 5 hours. But I suppose I am 1/4 Bear. No merge conflicts resulted because I went to work during the day (we met late at night), and it helped us move forward quickly because it was early in our project.

My second commit of note, from September 25, is a culmination of my refactoring efforts. After pushing many methods (such as the calculateNeighbors() method) to the Cell superclass, creating calculateNewCoordinate(), and rewriting simulations, I created an Edge interface hierarchy in the edgeTypes package. I then implemented the ability to read XML files for the user's desired edge strategy (finite, toroidal, or future edge strategies). Finally, I created a Grid hierarchy. This single commit built on a series of previous commits, happened during a team meeting, and even pushed along our XML work. Again, no merge conflicts happened, but most importantly, this successful solo effort helped set up my third commit of note.

This commit, however, highlights a failure. After implementing Game of Life and cutting hundreds of lines of repeated code, I aimed to reduce PredPreyCell.java by 2/3 down to 60 lines. I implemented a Rules interface hierarchy with FishRules and SharkRules on September 26. Yu Zhou mentioned state patterns could maintain Chris' implementation of allowing the user to manually alter a cell's state by clicking. I pushed calculateNeighbors() and doAction() into the Rules superinterface, which is implemented by the Cell superclass and Rules subinterfaces, to reduce repeated code in PredPreyCell.java and achieve DRY hierarchies of Cell, Edge, and Rules. Despite this success, I was forced to give up and, for the first time, submit a program that is not fully functional in the interest of time and good design.

Although we completed every objective in Week 1, many problems in Week 2 came from an inadequate design. Each member of Team 19 worked responsibly and communicated smoothly, but I should improve in evaluating and implementing good design since I had to edit the Cell hierarchy extensively. I should also peer program with teammates outside of team meetings to increase our mutual understanding of the code. Learning from my teammates will probably most improve my computer science skills and performance in this class. As for this project, I wish I had time to implement a fully functional PredPreyCell.java since I have refactored all my code to the limit of my current abilities.

Design Review

After reviewing our code, I have realized that there is a massive distinction between the codes each of us wrote. Overall, the layout is okay, and the naming conventions and descriptiveness are bad, even atrocious in some sections. As a result, most of the code is nearly indecipherable without comments, which are inconveniently absent in several areas. Many dependencies in our Cell hierarchies rely on getters and setters, but doing so did allow me to preserve the protected status of some variables and methods. Theoretically, I understand that variables and methods that are public are accessible to anything in the program, those that are protected are only accessible within a hierarchy, and those that are private are only accessible within the specified scope, which is usually a single class. Even so, my lack of previous practice may have resulted in incorrectly designating many variables and methods as public. The same issue also affected the final and static statuses of several other variables and methods.

However, there are some bright spots. I frequently edited our entire program to maintain a highly consistent quality of style and, from trying to understand my partners' codes, learned quite a bit about layout and other issues regarding the readability of code. By splitting our program into packages and hierarchies, some classes were very manageable in size, which resulted in great layout. Also, my difficulties in reading our code led me to try and learn various layouts, naming conventions, and descriptions. Most importantly, I was able to build hierarchies that allow users to extend features and test our code relatively easily.

Of course, testing is not always simple because one simple edit in back end code could potentially remain hidden until being exposed by extensive front end testing. Until we understand JUnit tests better, it will be difficult to efficiently test our code. Other than my PredPreyCell bug, our project functions exactly as intended because we were able to test our program effectively when we had the necessary time, which helped me expand my understanding of code relations from a method-to-method level to that of classes and packages. From the code I did not write, I also learned some about GUI and XML because I had never experienced either one.

To improve the clarity and maintenance of our program, we would have to rewrite much of our code to delineate the GUI, graphics, XML, and other functions. Our SimulationLoop.java is currently 575 lines, whereas our XMLReader.java seems overly simplistic. The Grid hierarchy would also need to expand greatly to carry some of the burden that SimulationLoop.java carries. The hierarchies may also be improved, but my current skill level may not be enough to do so, since I wrote them as well as I could.

The overall design of the complete program is simple. XMLReader.java reads in user files and passes information to the Cell Factory. It relies on the hierarchies to create various cells for various simulations on a grid, which is then displayed on a GUI that tracks information and allows the user to view, interact, and even modify the simulations. To add a new simulation to the program, the user would only need to create a new Cell subclass that the Cell Factory is able to produce. This would require a few lines in CellFactory.java and an XML file, but the hierarchies have

eliminated the need to do much more. However, if the user wants more complicated simulations, then new Rules subinterfaces could become necessary.

This is attributable to our initial team meeting, during which we determined we would create three modules for XML, Cells, and GUI. Unfortunately, without any guidance until late in our project, SimulationLoop.java has not been refactored at all, and our XML class is overly simple. Only the Cells module saw extensive evolutions during the refactoring process, which started very late into the assignment due to the mysterious absences of both our assigned TA and the grad TA. In the end, I had to seek help from Yu Zhou Lee, who is the TA for another team, and Dr. Duvall. This allowed me to significantly improve our hierarchies at the last moment.

As a result, my code creates various Cells with user input, which becomes the necessary traits for each specified simulation and the abilities to evaluate and interact with their environments based on specified Edges and Rules. With the hierarchies, the Cell Factory is quite simple. A constructor acts as a blueprint that directs the assembly line method createCell() because it holds the information needed to build Cells. The createCell() method starts by declaring a Class object and initializing it as a specific type of cell, then declaring a Constructor. With reflection, the try statement attempts to designate the Constructor as the specified Cell's constructor, and another try statement returns a new instance of the Constructor, cast as a Cell and filled with information from the parameters and the delta maps created by the setUpDeltas() method.

Otherwise, our overall code is not designed well since it fails to uphold the values of DRY, SHY, and Don't Tell The Other Guy. We simply ran out of time because we underestimated the complexity of this project and did not split our code into classes based on function. For one, our GUI allows the user to choose an XML file, but that section of code could have been placed in XMLReader.java and called by SimulationLoop.java when needed. A GUI Factory would probably be an excellent addition to our program, since it could handle the creation of multiple buttons and reduce a significant amount of repeated code. Instead, Week 2 only produced a program that handles the objectives of Week 1 with good design.

Other than our hierarchies, I believe that our program's design is very poor and does not resemble my ideal design. Preferably, everything involving XML files would be in a package of XML classes and only called by the GUI classes when it is needed. As described before, the information would pass through the Cell Factory and the Cell, Edge, Rules, and Grid hierarchies to set up the simulations, which would then be very simply displayed by the GUI, created by a GUI Factory that can create multiple buttons and sliders for the user. I feel that robust XML classes would simplify much of our necessary design, since I believe very strongly in building a strong foundation. In the case of code design, this would mean a powerful back end that allows the front end to simply focus on displaying simulations and allowing user interaction. With this ideal version of our program, the user can quickly create many new simulations with new behaviors on new shapes as described previously. and display a GUI with new buttons and new sliders. But our current XML module does not handle all XML responsibilities, and our Grid hierarchy is underdeveloped. As a result, SimulationLoop.java takes on too many burdens.

Otherwise, I consider the hierarchies to be very well designed code. I reduced, if not completely eliminated, all repeated code in our Cell, Edge, and Rules hierarchies. With Chris' help, our Cell Factory uses reflection and adds a layer of SHYness to go along with the high degree of DRYness.

Code Masterpiece

For my code masterpiece, I have elected to improve on my own module some more. One annoying feature I was unable to eliminate was the need for getters and setters to maintain the protected status of several variables and methods. But even worse, I was not consistent about the modifiers. As a result, I refactored Cell.java and CellFactory.java to improve the design and better protect my information. Combining that with refactoring that would be done in SimulationLoop.java lets me delete all getters and setters while maintaining my protected status. Some methods in other files could even go on to become private. This adds to the open-close design that we have discussed in class.