Greg Lyons
CS 308
Cell Society – Team 02

CELL SOCIETY ANALYSIS

<u>Project Journal</u>

This project was significantly more complex than the Game project and required much more planning and coding. We started the project on September 10th and finished early in the morning on September 26th. It is difficult to estimate the amount of total time I spent on the project, but I would guess approximately 30 hours. We tended to work in large chunks as a whole group rather than working a little bit individually each day – these group sessions usually lasted approximately 3-5 hours (10 hours the last day/night). The easiest tasks were designing the model/view system and writing the logic for the most basic simulations, while the hardest tasks were coming up with a flexible and robust design and adding the more complicated extended features (graph, dynamic user interaction, etc).

Although we put much thought and time into our initial design plan and implemented it for the basic implementation deadline, we realized that our design was not very well suited for extending and adding the new features for the complete implementation. Thus, while the first week was spent planning and implementing the planned design, the second week was mostly spent on an overhaul of our design (in reality, two overhauls of our design, but the second transition was not as major). It took longer than anticipated to get all of our basic features up to date with our new design, and we were not able to implement new features until the last two days of the project. As we were coming up with our new design, we were able to think about how we would implement the new features, so although it took a long time to overhaul our design, it made extension easier because we built that capability into our new design. Initially, we coded the abstract design together and then we each worked on implementing one of the basic simulations. When we realized we needed to make a big shift from a grid-based hierarchy to a cell and patch-based hierarchy, we spent much time together coming up with the new design and mostly worked together on implementing the new features. Meeting up as a whole group as much as possible made it easier to avoid conflicts and to make sure all of our components worked together. Although we often were working on individual tasks while we were together, it made it easy to keep each other updated on modifications and to fix merge conflicts, and we did take advantage of pair programming when adding the more complicated features. It is difficult to come up with a figure for how much time the whole team spent, but I would estimate that we spent approximately 20-25 hours as a group that we each spent approximately 5-10 hours individually. I spent a good portion of my time working on the Predator-Prey simulation (using the PredPreyCell, SharkCell, and FishCell classes). I had to implement it three different times for our three different designs, and it was significantly different for each design (the other basic simulations were not as hard to change from design to design). My other individual roles were designing the scrapped abstract Grid class (as part of our original design), implementing the process for getting resource labels, and adding toroidal movement capability. Chase and I combined to be the driving force for the design decisions of the program, and while Kevin attended our group sessions and made some individual contributions, he didn't add much at all to our discussions of overall design and how to implement new features. Chase put in a lot of individual effort in creating the XML parsing system and adding extended features (triangular grid, different initial configurations, and error checking). Chase was also able to implement the Segregation, Game of Life, and Sugarscape simulations. Chase and I utilized pair programming for creating most of the design, including the GridModel, GridView, Cell, and Patch classes. Kevin implemented the Spreading Fire simulation and added the graph feature to the final implementation, but did not contribute much to our pair programming sessions. Chase and I also did all of the commenting for the program. For our individual work, we each had our own branches that we made changes too, and we generally held off on merging these branches with master until we were all together as a group (to deal with conflicts in the best way possible). I thought the communication within the team was very good, in that we were mostly together and were able to look at and make suggestions to other group members' code. The communication was pretty consistent throughout the project and was mostly face-to-face.

As a group, we had 119 commits to the master branch. It is difficult to come up with an average figure for the size of each commit. The majority of the commits were small changes (<100 lines changed), although several commits (after long group sessions) had hundreds of lines changed. On average, we had more smaller commits than large commits. Our commit messages were fairly lazy in general – a lot of them are simply "Update to [insert class]". Going back now, it is hard to see exactly how the program progressed without more specific commit messages.

One of my commits in the early stages of the progress set up an initial framework for our original design. For this commit, I used the message "Started work on Grid class, created Cell class." This commit set up our abstract Grid class so that we could each implement our given simulations using that design (a design that was later changed in favor of a better version). It did not cause any merge conflicts because it created two new files, and it was done very early in the project so that our team was able to move forward with the simulation implementations. Another commit, which I titled "Started working on cell version of program", was the initial commit in our large-scale design shift. We switched from an abstract Grid to an abstract Cell. In this commit, we also set up a Model/View system that was much design much better than our initial Grid class, which served as both model and view. This commit was one of our largest and also created Patches and updated our XML parser. It did not cause merge conflicts because we made sure everyone was up to date with master before we made the big changes (using pair programming). Finally, one of my big commits towards the end was titled "Updated PredPrey simulation." This commit re-implemented my working Predator Prey simulation using Cell classes instead of Grid classes. I created PredPreyCell and its subclasses, SharkCell and FishCell. This update was later in the project than we would have liked, but we had to make adjustments to our design on the fly. This update caused merge conflicts because it modified our existing Predator Prey simulation, though we were able to work out these conflicts easily as a group (we had been meeting together at the time).

In conclusion, our team worked together well, although the project itself was quite a behemoth to tackle. We put in a lot of time and effort, though changing our design twice in the second week of the project caused a lot of stress. In the future, we need to take these possible large-scale design changes into consideration when planning. I think I took good responsibility on the team, although perhaps I could have done a little more work on my own if I had not spent so much time on the Predator Prey simulation. The parts of the code that required the most editing were the GridModel and GridView, as it was difficult to keep track of and update the simulation data while also displaying it in a sensible manner. To be a better designer, I need to think in terms of further implementation and extension rather than simply choosing the simplest and best option for a basic implementation. If I want to be a better teammate, I should continue my good communication, and also work on making my code more adaptable and flexible to changes by others. If I could work on one part to improve my grade, it would be improving the way that we update our simulation data (modifying objects that we pass in as parameters). Our visualization could also be improved significantly, as right now our display is pretty bare and simplistic.

Design Review

The code is fairly consistent in its layout and style. We spent much of our time coding together as a group, which allowed us to provide suggestions to each other for designing and naming methods. Chase and I split up the commenting and did so at the last minute, and perhaps we could have done a better job at making our comments in a similar style. Some of the classes are clearer than others, though we tried to do our best at making the important classes clear. Some of the logic gets a little confusing in the simulation rules (the subclasses of Cell), though we did our best to eliminate magic values and to comment in confusing parts. Most of the larger classes (GridModel, GridView, Cell, Patch) are designed to be easy to understand and are generally readable, though we were very thorough in commenting our methods. We tried to eliminate back channels as much as possible, although we did run into some issues with casting. In my Predator Prey simulation, I created a PredPreyCell class that had SharkCell and FishCell as its subclasses (to take advantage of similarities between the two species and their movements). However, as much of the simulation's logic required on methods specific to these classes while iterating through more generic collections, I was forced to cast some of the Cells in order to access the methods I needed. We also used a lot more get methods than we would have liked, though we had trouble finding ways around this. The main issue that we struggled with in our design dealt with sharing of information between objects in classes. Originally, we had a Grid class that was large and had access to all of the simulation's data. We decided that it was better to break the system up into smaller Cell and Patch classes as part of a Model and View. However, this required extensive getters and setters for the Cells and Patches so that their data could be accessed by the larger classes. It was difficult to create a useful inheritance hierarchy while keeping getters and setters to a minimum. The extendibility of implemented features varies depending on the feature. For example, we were able to extend from a rectangular grid to a triangular grid, but implementing hexagonal and tessellated grids would have required a fair amount of extra effort and thinking. However, extending features like error-checking and grid styling is not too difficult, as it does not affect the data and logic of the simulations very much. Parts that would be easy to test for correctness are the XML parsing (i.e. setting initial configurations), the behavior of individual cells (depending on the simulation), and the functionality of buttons. Parts that are more difficult to test for correctness are the updating of the whole grid and the proper implementation of all rules for a simulation's logic. I was able to find a bug in Chase's code that was preventing him from showing the initial display before starting the simulation's loop. He was updating the model before displaying the grid, and I made sure that we showed an initial display of the grid before making the first update. I was able to learn from Chase's XML parser about how to handle the reading of data from an XML file. To make the code more maintainable and elegant, it would be nice to consolidate some of the iterating we do (i.e. combining for loops), ideally we could eliminate some of our getters and setters, and work on updating our data in a better way than modifying a Patch array that is passed as a parameter.

The overall design of the program is a Model-View setup. The Model stores the simulation's data using an array of Patches and their corresponding Cells, and it updates the data with each step. The View sets up the user interface and visualization of the simulation, and makes calls to the model according to user actions. The XML Parser is the key component of setting up the Model – it reads configuration settings and initial states of the simulation from an XML file and then passes the initial data to the Model. The abstract classes used are Patch and Cell – the Patch is static in location but can also store and update its own state, and it holds a current and future Cell. The current Cells represent the current state of the simulation – used when checking neighbors, finding move locations, etc. The future Cells represent the next state of the simulation – storing what changes will be made before they are set to be the current state. The Model iterates through an array of Patches and updates the Patches, and each Patch updates its future Cell as part of its own update. This is accomplished by passing the whole array of Patches to each Patch (and Cell), which is not ideal for design purposes. Ideally we would have been able to pass only the Cells and Patches that were necessary, like neighbors, however the Segregation simulation required access to all locations on the grid for its random movement. Once the Cells and Patches are updated, the View utilizes the Draw class and updates the grid that is shown as the display.

Adding a new simulation to the program is fairly simple. The new simulation will need a subclass of Cell that implements its logic and rules, and may also need a subclass of Patch (if not, the GenericPatch class can be used, which serves as a simple location without any useful state). The simulation's necessary parameters will be parsed from an XML file and stored in a parameter map, which is the same process for all simulations (and the Cells and Patches can access and use for whatever purpose they need). Aside from making subclasses of Cell and Patch, the only other necessary steps for adding a new simulation are writing the XML file for its initial configurations and parameters (which have default settings and can use randomized or probability-based initial states) and adding another case to the switch-case system in the Factory class. The Factory will take a String from the XML file representing the type of the simulation, and will create its corresponding Cells and Patches accordingly.

We had many discussions and decisions regarding our design. As stated, we started off with an initial system of a large abstract Grid that had access to all data and served as both model and view. In this design, we didn't even have a Cell or Patch class, and we simply stored all of the data in integer arrays of current and future states and updated an array of colored Rectangles based on the states. This worked for our basic implementation of the first four simulations, although it got very complicated for Predator Prey (with three additional integer arrays that needed to be updated). We discussed this design with our TA, and he agreed that although it was a neat and fairly unique way of doing it, it was not likely very extendable to new features. Also, the idea of containing all of the data and the display within a single Grid class was not very good practice, as the entire Grid had access to everything in the simulation and could make changes wherever they were needed. The benefits of this were that we were able to avoid getters and setters, as the Grid had access to all of the data, and we did not have to make things public or worry about passing parameters between classes. The drawbacks to this design were that we felt that we were making the Grid class too powerful and giving it too much responsibility, and the abstract Grid didn't provide much of a framework to be extended (it simply left the class open for each simulation to write its own entire functionality in a single class). Also, using many integer arrays to store states and data was not good practice and had a lot of magic numbers. We decided to switch directions to base our design on abstract cells and patches, and we decided to use a model/view implementation that separated visualization from simulation data. Initially, we decided to make an array of Cells and an array of Patches, and each Cell and Patch had access to both arrays to manipulate and make changes. We were iterating over both all of the Cells and all of the Patches, and we decided that perhaps this wasn't the best design either. We settled on an array of Patches, which serve as static locations that can change and update their state. Each Patch has a corresponding Cell (stored as current and future cells, as explained above), and these Cells can be passed around across the Patch array (for simulations like the Predator-Prey one that have entities that move and evolve over time). The benefits of this design are that the responsibilities of running the simulation are divided between several classes (Model vs. View, and within the Model: Cells and Patches) rather than doing it all in one class, the data is stored in Cell and Patch objects rather than several vague integer arrays, and that the common shared features of running the simulations can be pulled out and passed with inheritance rather than making a whole entire new Grid subclass for every simulation. The drawbacks of this design are that getters and setters are required and used extensively to access the data stored by Cells and Patches, and that the entire Patch array is passed around and modified in many parts of the program. This was necessary for the Segregation simulation, which requires moving to a random location anywhere on the grid. Because SegregationCell was a subclass implementation of Cell, it was difficult to restrict the amount of data given to the abstract Cell while allowing the SegregationCell to have all of the data. We did our best to treat the Patch array as a sort of unmodifiable data structure, by updating the Cells and Patches using set methods for their data rather than directly modifying the Patches and their locations in the grid. However, the nature of this project is modification of data, and that data has to be accessed and changed in some way.

I personally coded the Predator Prey simulation, much of the Model/View interaction, and other various elements in the abstract classes. I struggled with how to best handle the Predator Prey simulation – in our initial design, I had a cumbersome amount of integer arrays that were updated in a mass of confusing code. The simulation worked, but I knew it was not a good way to implement it, and that treating the fish and sharks as objects would be better than storing their data in unrelated arrays. When we made the switch, it made my Predator Prey code a lot clearer. However, I struggled initially with how to best treat the sharks and fish as separate objects with similar but slightly different behavior. I settled on creating an intermediate PredPreyCell class that could implement the shared behavior of the fish and sharks while also serving as a blank water cell for the empty spots on the grid. I was able to use a mini-factory within this class to handle the initial instantiating of sharks and fish as part of XML parsing. My code masterpiece and its analysis explain this PredPreyCell class and its subclasses more extensively (see further on in this analysis).

One feature that was implemented successfully was allowing a choice for setting up the initial configuration of the simulation – either completely randomly, based on a specific list of states, or randomly based on probability distributions. This choice is provided in the simulation's XML file as an attribute with the tag "config". The XML parser reads the value of the attribute (a String) and stores it as an instance variable, myConfig. This instance variable is used later when it is time to construct the initial arrays of Cells and Patches (the Cell array is only needed to assign each Cell to a Patch in the beginning – the Cell array is not used later, and the Cells are instead accessed through their corresponding Patches). Both cells and patches are created using the constructList method, which takes either "cell" or "patch" as a parameter. The constructList method is a sort of factory that has switch-case statements to create the arrays based on myConfig, and calls the appropriate method (either doGiven, doRandom, or doProbability). The doGiven method is associated with XML files that have a specific list of locations and states. doGiven parses the row, column, and state for each Cell and Patch that is specified, and then adds an appropriate new Cell or Patch object (returned from the Factory class) to its spot in the array. When all of the specified locations have been filled, a method called setNullState fills in all of the empty locations with a default blank Cell and Patch for the correct simulation type. For a completely random assignment of initial states, the doRandom method fills the initial Cell and Patch arrays with random Cells and Patches by calling Factory's makeRandomCell or makeRandomPatch to make a new Cell and Patch for each spot. The makeRandomCell and makeRandomPatch methods work by selecting a random integer state from the number of possible Cell states or Patch states (which are specified in the XML as attributes called "numStates" and "numPatches"). Finally, the probability distribution of initial states works using the doProbability method, which makes a Map of probabilities based on parameters provided in the XML file. Then, each spot gets a new Cell and Patch by calling the Factory's makeProbCell and makeProbPatch methods, which each take a probability map as a parameter. These methods use a double called "current", which is initially set as one of the probabilities. It is compared to a random double ("value"), and if value is less than current then the corresponding type for the given probability is used to make the Cell/Patch. If value is not less than current, then the next probability is added to current, and value is again compared. If all of the possible types have been exhausted, the method will return null (not possible, however – the probabilities sum to 1 and the random is between 0 and 1). These three ways of setting the initial configuration can be used by all of the simulations, because they all call the Factory class (which insulates the handling of creating the correct Cell/Patch implementation). This design would likely need to be modified for horizontal or tessellated grids, for which a 2-D array might not be the best way to store Cells and Patches. It is possible to use a 2-D array for a horizontal grid, but it is not as intuitive. However, using the array works well for our rectangular and square grids, and it allows these three configurations to be used by any simulation that we implement (as long as the Factory class creates its Cells and Patches in the correct manner). We used the 2-D array as an assumption that made our program work a lot easier.

Overall, the main issue for us in this project was the conflict between containing everything in large classes vs. spreading the responsibility between smaller classes. Our initial design took the former approach, and avoided getters and setters by allowing the whole class to access all of the data and act accordingly. This design was not ideal for extending the project to the new specifications, as it required a lot of coding work done in each grid class rather than taking advantage of similarities in the simulations. Our final design took the latter approach, allowing for a clearer distinction of each class's responsibility and the storage of data in useful objects rather than unwieldy integer arrays. This design was better for handling the extensions, but relied a lot on getter and setter methods and didn't modify the data in a great way. Our ideal design would be more flexible and would use aspects of both design strategies. Our ideal design would utilize a Model/View interaction to split up the responsibilities of handling and displaying data, but instead of giving each Cell and Patch access to the entire list of data, we would only give it what it needs. The ideal design would allow the Cells and Patches to perform actions and update within themselves rather than using getters and setters to modify all of the Cells and Patches from the outside. Adding a new simulation to this ideal version would be similar to our current way in that it would need to implement the abstract Patch and Cell classes and have a way to make them in the Factory. Overall, it was difficult to decide on a design that was perfect for satisfying every aspect of this project, but perhaps we could have done a better job in putting together the best of both worlds for a flexible and extendible design.

Code Masterpiece

My code masterpiece consists of my PredPreyCell class and one of its subclasses, FishCell. The PredPreyCell class has a dual function: it serves as a default empty cell for the Predator-Prey simulation (a blank water cell that has no behavior), but it also serves as a superclass for the FishCell and SharkCell classes. I think the superclass is well designed because it helps avoid repeated code (sharks and fish have similar movement and breeding behavior), and it requires only a small amount of implementation by the subclass. It also makes use of a small pseudo-factory, which helps hide implementation from larger classes that are not concerned with implementation. The main Factory class creates the cells for each grid, and it is better to call the makeCell() method from the PredPreyCell class than to have separate if-statements within the Factory class to create the various implementations needed. Leaving the breed() and updateStateandMove() methods as blank in the superclass allows for a water cell to have no behavior (as intended), while leaving it up to the FishCell and SharkCell to override and provide the basic implementation needed for these two methods. The pickMove and checkBreed methods are common to both subclasses, and bringing them up into the superclass allows both subclasses to access the methods without repeating the code. Additionally, utilizing an inheritance hierarchy helps get rid of unnecessary chains of if-statements. My original implementation of the Predator Prey simulation was all within one class and utilized integer arrays and countless if-statements (to implement shark vs. fish logic), rather than putting the different behaviors into subclasses. In this design, the two subclasses are reduced to the smallest possible amount of code that they need to function – there is no need for a FishCell to run through if-statements to see if it is a fish, it can simply implement the fish's behavior.

It is difficult to test my code masterpiece in isolation without running it as part of the game (and taking advantage of XML parsing, etc), although it is possible to design a few tests. Primarily, I would test the ability of the FishCell to move, breed, and update properly, by calling its update function and checking parameters. I would also test the ability of the SharkCell to limit the FishCell's movement. Bugs that I would look out for are invalid moves by the FishCell and improper timing of breeding. Please note that in creating my J-Unit tests, I had to make some methods public that were previously private or protected (they are commented in the file). For the purposes of running the program, they should be protected.