Project Journal

Time Review

* How many hours I spent – Individually, I spent around 50 hours working on this project – ~20 hours for the basic implementation and ~30 hours for the complete implementation (this includes all team meetings as well as individual work). I began work on September 8th and finished on September 26th.
* How I spent my time – Most of the time in the project was spent working with the other teammates on primarily integration of different components, design reevaluation, and refactoring of existing code. Each meeting we would pull together what we worked on individually or continue to develop on pieces that needed work.   
  Most of my time individually was spent implementing a specific new feature that was decided upon at a group meeting. After a meeting, we would make delegations of what needed to be accomplished by the next meeting. For example, between meetings, I would focus on drafting and implementing the logic for the update function for one specific type of simulation. In the first week of the project, a lot of my individual work time was spent on implementing XML parsing and the specific update functions for each simulation. In the second week of the project, the biggest chunk of my time was spent on changing our rigid Cell definition to Cells and Patches in the simulations.
* What tasks were easy, what were hard – The tasks that were easiest were building features/writing code from scratch. In terms of specific features, writing the logic for the update functions of the simulations was easy. I found the XML parsing and error detection to be moderately challenging, but still quickly accomplishable. The only challenge I ran into here was dealing with non-standardized inputs and reacting properly to these situations (such as null pointer exceptions) without stopping the entire CellWorld program execution. What became difficult was either integrating my approach with what was already done, or to build on existing code. One example of this was changing the cells to cells and patches. In doing this, the model portion of the project (the backend logic and data) became incompatible with the front end, which was primarily looking at only the cell states. This led to the second example, which involved me having to go into the front end code and make multiple changes in order to display cell or patch states whenever appropriate.

Teamwork

* How many hours the team spent – The team spent overall about the same amount of time as I did individually, as most of the time we were working together in meetings.
* The roles were split as such:
  + Frontend – Rahul Harikrishnan
  + Backend and cell hierarchy – Kevin Li
  + Backend and inputs (XML, error checking) – Abhishek Balakrishnan
  + We received some guidance from our TA (John Godbey) as well.
* Team Plan and Stability – For the basic implementation, we had a very solid plan that we followed. The class diagrams gave us a good framework to go off of and made the initial implementation run very smoothly. Our plan was to meet at least once every two days so that we were keeping up to date with integration, each other’s progress, switching roles in case something became overwhelming for an individual or another individual had a better idea for how to complete a feature. To keep our working from conflicting, we each worked in our own branch. Upon completing an individual portion of the work, that branch was merged to the next closest part, and so on until we were all working on the same components or just refactoring. This really reduced merge conflicts over the project. Overall, the plan was quite stable and helped us submit a well-coded design by the end.
* Response to extensions – We were not too surprised by the requirement extensions, so our response was to continue with our general design framework. The only thing that we found overwhelming was the number of additional features to implement, and it took a few days for us to figure out what we needed to prioritize. Our team responded by adding more hierarchies, which made it easier to clean up our existing code/design as well as extend to the new features.
* How did I individually manage code – I managed code by starting small and separating into new classes wherever possible. I pushed as soon as a feature was done or if I had done some key refactoring.
* How was group communication – Communication was just right. We were regularly in contact and it was easy reaching others in the group. There would be times when we decided it was okay to not meet so that we could focus on our individual work and implementations more.

Commits

* How many commits – I made 50 commits throughout the project to master branch.
* Yes, the commits accurately reflect my contributions – especially in the branches outside of master (these branches are Inputs, Patches, and Gui4).
* Commit 1 - “Randomized Eco cell, made sharks starve so they actually die”
  + Time: September 18, 2014
  + Branch: Master
  + Purpose: Corrections for new features.
  + Merge Conflicts: No
  + Commit done in timely manner: Yes
* Commit 2 – “almost all original cell->patch conversions done”
  + Time: September 25, 2014
  + Branch: Patches
  + Purpose: I spent a lot of time separating Cells into Cell and Patch classes. This involved changing the hierarchies and all of the logic for each cell/patch, as well as the Simulation classes that are running the update function for each unit in the grid.
  + Merge Conflicts: Yes
  + Commit done in timely manner: could have been timed better, as I waited to do all the logic and conversions from Cell to Patch (which affected code in numerous classes) before pushing. This took some time to resolve merge conflicts when trying to integrate with other code, and eventually we decided that we would just recode some of the pieces based on this template because of the number of conflicts and inconsistencies.
* Commit 3 – “Finished XML Parsing Code”
  + Time: September 13, 2014
  + Branch: Inputs
  + Purpose: This was wrapping up a new feature via debugging. Since this was before Junit tests were introduced, my debugging was done with print statements and by also verifying that data extracted from the files were properly stored in the game parameters class.
  + Merge Conflicts: No, commit was done in timely manner and was completely separate to what the others worked on.

Conclusions

* Our team underestimated the number of extension requirements, and while we had a basis of extensibility and an idea for how to accommodate for broader requirements, we should have taken more time to make our initial design more extensible and have more class hierarchies so that the code was manageable from the start and we could tackle the new problems from small classes rather than having to make changes to very large classes.
* I think I took on enough responsibility in the team, as I handled the XML data process as well as worked heavily on the backend logic and class hierarchies. One of the biggest responsibilities I took on was handling the separation into Cell and Patch classes at the end. By laying out the template for how to do this, we were able to quickly code in the links and finalize the update logic for cell/patch combos in each simulation mode.
* The parts of the code that required the most editing (from my perspective) were the Cell definitions, switching them to Patches; and the neighborhood definitions, moving from one standard 4 or 8 definition to something that varied with different parameters and completely changed depending on the simulation. This is because they accommodated for very simple cases in the basic implementation and had to be extended for much broader cases for the complete implementation. Therefore, we had to add a new hierarchy for both.
* To be a better designer, I should continue to use good coding conventions from the start. One of the practices I think I do well is writing small methods as I go. If a method I am working on is too long, I try to separate it immediately rather than pushing it off to refactoring later. What I should start doing is writing tests for my code before I write the code, as this will speed up crafting the code and make sure I hit the cases I need to. I should also start fleshing out designs on paper/whiteboard more before jumping straight into the code.
* To be a better teammate, I should keep being up to date with communication. I think I could improve at getting more done on my own. There were a lot of times that I would get stuck and have to wait till a group meeting to flesh out details for how I should proceed or what design decision we should make. Something else I could improve as a teammate is the timing of my commits. As mentioned earlier, I did try to commit once a day, but one bad commit I made was a very massive change all at once. I think committing after steps of a feature have been implemented is valuable so that people can get the progression of the feature, not the whole thing at once. Lastly, I think doing more extensive testing on my code before committing is useful too because putting buggy code up can cause problems for others.
* If I could work on one part to improve my grade, it would be to refactor the CellViewer class. I think the class encompasses a lot of the project and that a lot of the features need to be separated from it into smaller classes. I think that the CellViewer’s length is the one thing that is making the code not as readable. Some improvements that could be made to this class would be the organization and obviously extracting some of the grouped functions into separate classes. There are many ways of doing this – for example, some of the GUI elements could be set up in another class. Right now, XML Parsing and Error Correction are both happening in this class, and they could be separated and put together somewhere else. Since I think our design is strong at this point, I think this would be the last step to solidifying it as a whole, and that would make adding the remainder of the features very easy for us.

Design Review

Status

* The code is generally consistent in its layout. We followed the naming conventions generally mentioned in class - with class variables named with my\_\_\_, camel casing, method names (just to name a few examples). Each of our methods is commented with a clear description at the top, and almost all of our classes have the Javadoc title, description, and author named there.
* The code is very readable. This is helped by the fact that most of our classes are short and our methods to the point. Where there is tedious work, that is pushed away into new classes (i.e. neighborhood setting). Also, since a lot of subclasses for specific simulations follow the same overall template, this makes reading and understanding the logic a lot easier. Comments are including outside each method and at the top of each class for ease of understanding the general functions, but these comments are not needed to understand the code.
* Dependencies are clear and easy to find.
* Implemented features are easy to extend because they are mostly done through hierarchies. If a new simulation wants to be added, this requires just extending the existing Cell, Patch, and Simulation classes. If a new shape should be displayed in the GUI, this requires extended the ShapeCell class in the frontend package. For the tessellated requirement which we were unable to do for this, we know that the implementation of this would mean simply using the existing ShapeCells and having our ShapeFactory instantiate different combinations of these than just the same one over and over.
* Testing individual Cell or Patch subclasses would be easy because there are various helper methods that can run on any input data such as lists or state variables. These subclasses do not have much data individually, and so we could easily set neighbors and initial states and test the update function to make sure the state changes where appropriate based on conditions. Testing the Simulation subclasses would be a little bit more difficult since these classes are primarily focused on running the update function for every Patch in the grid of patches, so testing this would go back to testing the Cell or Patch subclasses. Testing the Viewer class would be the hardest since I am not too sure of effective ways of testing GUI code. At the moment, our Viewer class is also very large and is doing too many different things one after the other, so it would be hard for me to gauge where to start testing.
* Found minor bugs – one related to XML parsing when it was being used outside, in that there were extra whitespaces in the code, which led to bugs later. Another bug I remember finding in another person’s code was related to the specific logic in Eco simulation. This was related to a misunderstanding of the logic and leaving out some parameters. At the end, the biggest bug I found was a mix up in the coordinate system. All of a sudden, even though we had been following a standard for the model, the view mixed the x and y coordinates up for the 2D arrays, and this led to all of the logic running incorrectly for the games. By identifying this as the problem and verifying by comparing the data in the grid object to what was displayed on the screen, I was able to point my teammate to make the quick fix that could have taken a while to find.
* Learned something good from the Cell hierarchy that was written at the beginning – this honestly helped me grasp inheritance better. Learned something bad from the CellViewer class that was also written in the first half of the project – this helped me realized the consequence of not refactoring early, as code can pile on itself and become unmanageable.
* Aside from commenting, we could work on even better naming conventions to differentiate between Cells and Patches, as their variables and methods have the same names. I think we can be more descriptive as a whole with our variable/method names, and this will reduce the number of comments we have to insert. Lastly, I think across the board keeping our methods short (not just in my code but in the whole group’s) will make our code a lot more to the point and clear.

Design

* Overall design: our implementation of CellSociety is split up into several different parts. The front end is primarily focused on displaying the data. The backend involves both the input files as well as the simulation hierarchies.
  + The highest level is CellWorld, which sets up the screen and launches the CellViewer, which is very similar to the Game Loop from Assignment #1.
  + Visualization – CellViewer talks directly to the front end (as in this class takes the data and displays it every loop). CellViewer launches the simulation initially by loading and parsing the XML file. The GUI is set up as follows: a number of Button objects are added to a scene which is returned for the GUI. Each of these buttons is tied to a specific function for the animation (i.e. start plays the animation and continues looping, stop pauses the animation, step updates the animation once). Then, the grid is constructed via a hierarchy based on the input data from the XML file. Each extension of the shape hierarchy calculates the positions of the vertices of the specific polygon for a grid unit. These are packaged together into an overall grid of shape cells via a Shape Factory class, which returns this grid to the view to be displayed in a BorderPane. Lastly, mouse events are handled by tracking a mouse click on a specific unit of the grid, and translating this to a change of state. The click will iterate through a list of the states for the cell occupying that grid unit, and this will change in the view by switching to the color in a list that matches the cell’s states.
  + Simulation – The specific simulation object is matched with the input from the XML file and then sets up its grid of patches and cells with all of the necessary parameters. In doing so, this launches the NeighborSetter hierarchy, which hides the specifics of calculating the neighbors based on the simulation mode, unit shape, and edge type in the grid. Once all of the backend has been set up, each frame, the simulation runs an update routine for each patch in the grid of patches.
    - Patches and Cells are both hierarchies with a one-to-one mapping – there is always one cell/patch and one patch/cell. These classes contain pointers to each other as well as a swap method for Patches to switch which Cell they are pointing to. This makes jumping around the grid easy for the Cell to do. The grid stores patches, so the list of neighbors is stored in Patch. Both Patch and Cell classes have their own update methods. Based on the given simulations, the Cell’s update method is always defined, while the Patch’s update method might do nothing more than call the Cell’s update. This is because in some simulations the Patch and Cell are acting separately and have their own states, but in others all of state and logic is accounted for in the Cell. One of the extreme benefits of separating the Cell into both Patch and Cell is that now these classes are much simpler, and there are much fewer if statements. Both the Patch and Cell have a state representing whether they have been updated in a round, which is valuable because sometimes there needs to be a priority in the order to examine the units or a cell that moves might be viewed multiple times in the simulation.
    - Simulation is a hierarchy that constructs the model grid of patches and runs the update function every round of the simulation for each Patch in the grid. After calling the update on each Patch, this class resets each Patch and Cell by setting their updated state to not updated.
    - NeighborSetter is another hierarchy that fills the list of neighbors for each Patch based on what inputs come from the XML file. This design allows a number of generic neighborhood patterns to be coded in the parent class, a default pattern to be chosen, and in the case of a simulation that does not have a standard neighborhood definition (such as Sugarscape), the appropriate pattern can override the default pattern. The biggest benefit of this hierarchy is that the complexities of the neighbor setting process are hidden from the user.
  + Configuration – the other portion of the backend is the input file process. The XML parsing happens upon initialization of the CellViewer loop. The parsing happens in a separate class, and the data is packaged and sent back to the viewer in InitialGameParameters. Within the viewer class, error checking happens on this data. Error checking occurs here so that messages can be sent directly to the user in the loop, and so that the loop can stop in the case of an error.
  + Regarding dependencies, the frontend depends on the Simulation class for the updated grid, which is translated directly into the GUI. Since Simulation creates Patches which in turn create Cells, the functional dependency is for Simulation to run Patch and Patch to run Cell for each frame. During updates, Patches and Cells depend on each others’ states. Cells depend on the Patches for the neighboring cells in order to properly update during each round.
* To add a new simulation, an extension to the Cell, Patch, and Simulation classes need to be defined for that simulation.Then, the neighbors need to be defined as a new extension to the NeighborSetter class. Lastly, they need to add to line 157 in CellViewer to define that new Simulation object. Our design hides a lot of the initialization process from the viewer class, and also iterates through a list of Simulation objects to compare to the input from the XML file. From there, the Simulation class takes care of setting up the model logic and data.
* As a team, we made several different design decisions. First, we decided to have the Cells concerned only with their own update logic and their own state rather than have to store the entire grid and make decisions all at once. This led to our Simulation classes which keep track of the grid and run the Cells’ update functions. This was a key aspect of the first half of our design, and it helped us a lot in compartmentalizing functions and keeping the data to a minimum within the Cell and Patch classes.  
  During the first part of the project, a design decision that came up with the TA with regards to how to handle a case where a Cell needed to know grid area outside of its neighbor definition was to allow the Cell to have knowledge of the Simulation, even though the Cells were being created by the Simulation. At first, this seemed like a strange design decision, but this proved to not be burdensome in any way for the Cell as all it had to do now was access new data from the Simulation (not take the whole grid from the Simulation).  
  At the second part of the project, our team decided to switch to a Patch and Cell definition because we realized that the Cell logic would become very complicated and convoluted if everything stayed in just one class. There were also benefits to separating Cells from the grid unit, in that now Cells would truly represent mobile units. While this change was tedious and required all of our heads to flesh out at the end, this was a very good decision we made in that it simplified our Cell subclasses and also allowed us to easily integrate a new simulation at the end for Sugarscape.
* Some of the key portions I worked on were the XML Parsing and the Patch hierarchy. One design change I made from the existing framework with just a Cell hierarchy were to involve minimal data for the Patch in its constructor. Another change I made in terms of how data was being passed from the Simulation subclass to the Patch/Cell subclasses was to remove constants from constructors. I realized that there was no reason to have additional parameters in a constructor if they were always going to be set to the same value. Therefore, I introduced an idea in our code of having more than just the input parameters to constructors be initialized in the constructor. This became very helpful in extending the Patch class to subclasses where a particular simulation might require more parameters for the Patch than others. This same concept was applied to the Cell class as well, and it improved extensibility a lot. Otherwise, my design for Patches was useful in reducing duplicated code in that a lot of similarities between subclasses of Patch and Cell were pulled up to the parent class. A particular instance of this was with accessing neighbors from the Cell’s perspective.
* The feature that I will describe in detail is the Error Detection and Recovery for bad input data. There are many cases where the XML file might be missing a tag for a key parameter, such as the grid size or the edge type. The XML file might also have bad data, such as cells being defined out of bounds or the grid size being negative or a non-integer. In any case, these inputs need to be accounted for in some way. After returning an InitialGameParameters object which contains all of the data read from the XML file, this data is passed through a checkValid method, which raises a number of potential problems that could be there with the file. The code at hand can be improved by grouping these into and acting based on categories.
  + If there is no simulation mode specified, this is an immediate stop.
  + If the grid size (either dimension) is < 1, this is an immediate stop.
  + If the threshold value for a simulation mode’s logic is < 0, this is an immediate stop. This could also be accommodated for by setting that value to a default value such as 0.5.
  + If any of the Initial Cells are out of bounds, this is an immediate stop. This could also be accommodated for by removing that cell from the view and the list.
  + If there is no unit shape or edge type specified, these are defaulted to SQUARE and FINITE.
  + There are many extensions to this concept of Error Checking. For example, based on the simulation, there could be a default setting which would easily be set here. If no cells are specified or there is no grid size listed, then the Initial Cells could be populated here. Furthermore, if ambitious, the user could also be prompted to select which option he/she wants if there is an error. This would involve a popup notification that tracks a response. The easiest would be for the user to select between options.
  + For most of the errors, they are displayed to the user via a popup notification, which is generated in the method popupNotification(final Stage stage, final String message) { }. This method takes the stage that was provided from the file choosing window and displays a message there based on the issue that the code found.
* The code is designed as an overall loop, with higher classes creating instances of smaller classes to do processing and return data that will eventually return to the view. Our design does not rely on some classes to pipe data back and forth from the Cells, nor does it have a bunch of smaller “dumb” classes get generated simply to return values, as each class is doing a fair amount of work.

Ideal Design

* The original design handled some of the project extensions well, such as dealing with simulations where neighbors were not just the ring around the cell and dealing with simulations where patches and cells had different states. In regards to the different shapes, it was not too difficult design-wise to switch from GridPanes to specific ShapeCells to be rendered in JavaFX. One of the biggest shortcomings of the original design we realized was the static grid definition we employed. Our grid is a 2D array, and this limited us for dealing with the infinite edge type.
* The ideal design would use a dynamic data structure to store the grid of cells. In doing so, the grid could expand once cells reached the edge. The ideal design would keep a class to set the neighbors, and separate out the model into Simulation, Patch, and Cell classes. It would also have separate classes for XML Parsing, Error Detection and Recovery, and Configuration/Styling. This design would heavily rely on hierarchies in order for extensibility in each component, whether it’s a new visualization technique, a new type of simulation, or a new input parameter to the simulation.
* To add a new simulation would be the same process as before. Even though it may seem like a lot of pieces need to change, it’s better to have the logic separated into many classes than to have to write all the details into one class.
* The current design is deficient in its static grid definition because this prevents us from being able to deal with different grid shapes (not a requirement but a cool extension) or infinite edge types where the grid could expand as the simulation went on.
* The current design matches the ideal design in the various hierarchies described. I think the hierarchies we have laid out for Patch vs Cell vs Simulation vs Neighbors spread functionality out. This will make integrating more complicated simulations a lot easier in the long run.

Code Masterpiece

The purpose of this component is to read and parse information in an XML file for the simulation. This code is well designed for a few reasons. First of all, the entry point into this is a get method, but the get method does more than simply return the parameters – it does some calculation, loads the data into an instance of the InitialGameParameters class, and finally returns this object. The get method clearly calls three significant methods – first, it creates the document; second, it parses the generic simulation parameters (the static data); and lastly, it parses the list of cells (the dynamic data). The list of cells follows a hierarchy in the XML file, so having a separate section for parsing this is valuable in that each element can be examined and parsed in a new method.

Instead of passing around the InitialGameParameters object as done before, this class now has the IGP class object as a private class variable, so that the data can be populated from any method.

Another of this code masterpiece are that methods are very short and to the point. Also, every method has some key functional purpose, and nothing is serving as just piping data from one side to another or doing redundant work.