William Chang
netID: wkc10
Team 4
Analysis: Cell_Society

Time Review

For this project, I estimate personally having invested around 45 to 50 hours of my time, beginning 9/9/2014 and ending 9/26/2014. I splitting my time around 15% reading directions, code documentation, tutorials, and forum posts to understand how to code what I wanted to code; 25% thinking, planning, meeting and discussing the project design and implementation details with teammates; 45% coding new features; and 25% refactoring and debugging.

My time spent communicating ideas with my teammates was well spent, as well as time researching details on how to code things (such as implementing LineChart as well as using design paradigms such as)

The easiest task for me was working and communicating ideas with Davis and Wenjun: it was easy to get our team motivated to work regularly and to work together, and for the most part, I am very thankful to have been paired with such great, productive teammates. We made sure that everyone was in contact with everyone else (ie, everyone had everyone else's contact information). The tone for this group dynamic was set very early on when we met for 3 hours, two consecutive nights in a row to make sure we had the best initial design possible, and that everyone in the team was clear about how we wanted our design to be implemented. Further, responses to communication of logistical details happened in constant and quick fashion. These were all great uses of our time.

However, there were some hiccups during our meeting time, such was during discussion of ideas, as well as communicating to make sure we were not causing conflicts for each other after pushing. However, as we got to know each other's work habits and styles better, this initial problem became a nonissue, and we became more efficient with out time.

At the beginning of our work, using Git in a team setting was slightly difficult (to avoid conflicts). We all caused conflicts for each other at certain times from our pushes and commits, and it took a little bit for use to figure out a way to prevent problems from recurring (we would check to make sure everyone's branches were up to date before performing a push).

At other times, it was slightly difficult to code features in such a way as to match the implementations of my teammates. At times, none of us necessarily knew all the _how_ the others were coding the features (although we knew all the _what_ each team member was coding), especially at the start of the project since it was the first time we had coded together in a team setting. Specifically, we had split the Controller classes and the Cell

hierarchy equally between the three of us, and each of those classes worked closely with the others. However, this difficulty was remedied by the fact that we worked together as often as possible, and thus in the long run, everyone gained an understanding of how we implemented our features by maintaining constant, open communication about each feature's functionalities.

The most difficult things I had to do was to convince my teammates that design changes were necessary after work on our project was already underway. I elaborate more on this in the design considerations section below.

Teamwork

My teammates and I regularly put in time every day of the week, having a set meeting time every night in the Link. We averaged a regular 2.5 to 3 hours per day (with a few days of break for weekends to give everyone a chance to breathe and see daylight). Adding all our man-hours together, we clocked somewhere between the 120-150 hour range.

Our team was cohesive overall and open to exchange of ideas. From the start, we were all committed to investing as much time as possible, as well as creating the best simulation possible. From the first meeting, it seemed that we all conveyed this similar attitude to each other, and shared a common vision for the project. We made sure everyone was equally informed and contributing his equal share, planning, designing, and writing our plan together, at the same time at night, working in the same place in the Link, sitting at the same table.

Davis

From speaking casually with both Davis and Wenjun, it was evident to me that Davis had the most coding of the three of us, having worked a software internship the previous summer. Thus, I believe Davis actively served as a source for knowledge and reference that we were heading in the right direction for our project. This role was especially evident early on as Wenjun and I were learning to use Git in a team setting, and Davis would step in and guide us through the process, helping us resolve conflicts. I also believe Davis's prior experience gave our team as a whole extra confidence that we could not just complete our project in time, but also complete our project in a stellar fashion (I understood that there were a few people in the class who had extensive prior coding experience, and thus my team would have been at a disadvantage had we not been more evenly split in terms of experience). The counterpoint to this confidence was convincing Davis of redesigns down the road (the Patch), which really required solid reasoning. Regardless, having an experienced member also meant that Wenjun and I would be able to learn more from him, coding in the team (as evidenced by the pair programming strategy Professor Duvall presented and had us do in class). One of the major design considerations Davis contributed was the idea of having cells "preparing to update" their

states, before actually updating, which we were able to develop further to allow greater extendability for part 2 of our project). As well, Davis contributed greatly to the core functionality of the project, such as the controller (for initializing the simulation) as well as the userinterface, as well as a simulation (Segregation).

Wenjun

As a senior, Wenjun has considerable experience in other ways than Davis, such as more life experience, more computer science classes under his belt, and more experience fulfilling responsibilities. I believe Wenjun (like me) also understood that he had less coding experience than Davis, and thus checked with Davis for his coding implementations. With that in mind, Wenjun served in implementing all the things that entailed nitty gritty details, such as coloring the UI and the xml parsing; which were on the more time consuming side; these implementations allowed our project to reach the quality and extra functionality that we were aiming for from the start of the project, and Wenjun got it down for us. Wenjun contributed in our discussions, however in a less active way; usually affirming ideas rather than offering suggestions or discussing the feasibility or reasonability of a design idea. Wenjun also operated a little more independently in his implementations and communicated less about what he was doing than Davis or I did, which is not necessarily a bad thing, but would have contributed better to our overall group understanding of what was happening with our project, as well as maintain better design. What surprised me about Wenjun, was that in the two days before our final implementation was due, Wenjun cut off all contact with both Davis and I, and did not meet with us to work on the project, nor did he tell us why. He later revealed early in the morning our project was due, that he was overloaded with work for other classes, but had actually implemented an additional simulation to our project (which worked great). This revelation both impressed, but also frustrated me because better communication on Wenjun's part could have kept Davis and I from wondering what he was doing, and wondering whether his code would actually be refactored, or if we would need to refactor his code for him before our deadline. Regardless, Wenjun contributed greatly to adding detailed features to our simulation (colorpicking, forest and Sugar), as well as the core functionality of the xml parser.

Me

I served as the facillator/organizer for our group, bringing up design points and considerations when needed for discussion, and making sure everyone was in agreement (or at least had a consensus), about how our design would work. This consensus is evident in our efficient division of classes into both Cells and Patches, as well as our Controller class divisions. Furthermore, I sent out and set up meeting times and locations, made sure our group was communicating as much as possible and updating each other on progress (by Thursday night before our first deadline, we had already finished all functionalities, were focused instead on refactoring and making the code as good as it could be, and we knew we would be wrapping up well before midnight). I took on this

role because of the void I felt within our team structure; none of us knew either of the other members before working on this project and we seemed slightly hesitant to begin communicating ideas with each other, so I understood that building chemistry and rapport early was important in order for us to be open about our ideas, problems, and work efficiently together in the long run (which I think was successfully achieved because I enjoyed, to a certain extent, our productive, daily, late-night coding binges by the end of our two weeks together, and I believe the others did as well). Further, I maintained intermittent contact with other teams (Specifically 1, 5, 6, 8, 14, 18, 19, 20, and 21), comparing progress in feature development and design style as a checkpoint to how our group was doing feature and design-wise. Feature-wise, I implementated our Patch hierarchy, a fair amount of our Cell hierarchy (including behavior interactions with the Patch hierarchy, as well as the Grid), and our Grid behavior and functionality. The Grid serves as the third pillar in our package of controller classes, organizing and updating information for all the cell and patch classes.

Note:
In our original plan, we gave everyone primary and secondary responsibilities for each of our classes, which we maintained through the course of the project. Further, as our simulation became more complex, it became increasingly important for each team member to understand the code/classes of the other members, thus each of us contributed to the design and implementation of the other classes in some ways (i.e. most of the simulations and the main controller interacted with the Grid, thus both Davis and Wenjun contributed to some of its functionalities; likewise, simulations and patches required interaction with the MainController and UI, and Wenjun and I contributed functionalities to those classes).

Our plan for completing the project involved implementing a core set of controllers simulating the behaviors of any simulation defined through our cell and patch classes. The core idea of our plan remained stable: we maintained a package of controller and and simulation object classes which would be closed to modification but open to extension. Through the course of our project though, the implementation details evolved as we better understood how our classes could work together (i.e. adding a parser class, beginning to create factories to relieve the MainController of that responsibility). In terms of simulations, we maintained the original idea of easily extending the breadth of possible simulations by simply creating a Cell (and a Patch class if desired) which hold the behaviors of that simulation. By separating our simulation into these two main grouped component packages, we could ensure that progress could be made regardless of whether or not one of the packages was complete vs the other. In making this division, we attempted to make testing and updating code easier; that is to say, we understood that if a behavior was not performing correctly, it was a problem with the simulation objects, and not with the any of the controller (such as if the Shark cells were not replacing any of the Fish cells by eating them); however, if something was not loading properly, or if users could not interact with the simulation properly (such as clicking a button or changing the state of a cell). Thus our preliminary efforts in categorizing our class components served our purposes well. However, we perhaps did not go far enough in classifying our

components, and our resulting Controller vs Simulation Object packages have become unwieldy in size as we added more classes and functionalities.

Our team quickly divided up the extensions and began work on them after the details were released. However, it became slightly less obvious who would be in charge of doing what the new requirements made our original roles obsolete, and we struggled a little bit to assign features to the best possible party to develop (i.e. Davis worked on Patch shapes, I worked on Grid rules and a graphical display, while Wenjun worked on cell colorings and the look of the the simulation overall, vs. our original roles). Ultimately however, we managed to

I developed my code in bits and pieces, developing the core functionality first, testing said functionality, refactoring it into a hierarchy, and then pushing it. For example, in creating Toroidal functionality for the Grid, I created the methods to implement the behavior of Neighborfinding; then I updated the MainController, to allow for setting the Grid to have Toroidal rules; next I created radio buttons to allow for realtime graphical change of Grid rules; and lastly, I refactored the functionality into a class hierarchy. Sometimes I would push before my code was functional, which caused errors for my teammates, and I learned to stop doing that.

Overall, I erred on the side of more communication to balance any potential loss of communication and make sure everyone understood what was going on in all aspects of the project. From our fairly quiet initial meeting, I was worried that Davis and Wenjun would not be as active in communicating, as would be evidenced in the final couple days of the project when neither Davis nor I was sure what Wenjun was doing. Further into working together, Davis communicated as much as necessary and so there were no problems on that front. However, Wenjun's lack of responsiveness served to hinder our progress a little late into the final project implementation, although not detrimental to functionality as he ultimately contributed an additional simulation. However, design wise, Wenjun's lack of communication and discussion for Davis's and my feedback on how he was implementing the XML and Parser, as well as the GridInfo classes caused a fair amount of design flaws and extendability issues in the long run (implementing new simulations was problematic as it required a tri-fold change: implementing the Cell and Patch as required, updating the GridInfo–a class of only getters and setters–with more getters and setters, and updating the XML Parser with more case fields). After reading through Wenjun's implementations near the end of the first deadline, I spoke with him about the potential design issues his implementations might bring up in the future. Wenjun responded in recognition of said design flaws, and said he would refactor things for the second deadline, however this refactoring never happened. These design flaws also led me to hesitate adding more functionality, as I would only be able to extend the project with additionally poorly designed code based on the dependencies I would be forced to work with (i.e. the Ant simulation would require increased XML parameter parsing, which was not supported, and would require adding getters and setters to multiple classes in order to implement).

Commits

In total, GitHub reports that I committed 99 times to the project in the span of 2 weeks, averaging around 7 commits per day. I would merge and push least once per day to minimize the potential conflicts, and to let everyone know what I was doing/had done that day. Some of my commits were really big (new/deleted classes, refactored hierarchies), while others were tiny (such as documentation and null checks). In the future, it would probably be better to commit somewhere between those two extremes to keep updates and changes more consistent each day, rather than in big or little steps.

I would usually push code as soon as I committed it, to make sure everything was the same for all branches (frequency is thus at least 1 time per day, as with commits above). Doing this caused conflicts for my teammates at certain times during the project, and thus I learned to only push to Master after communicating to everyone that I was about to do so, and making sure that everyone had up to date branches themselves.

My commit messages accurately represent my contribution to the project.

Three commit examples:

Commit Description: Created Cell, GridManager, and PredatorCell Classes. Added method
commit 940c2fef647105ea838b96cdba1845d8eda52b7c

Purpose: To begin creating functionality of the project. Initialized the classes I was working on.

Consequences: None.

Timeliness: This was the first commit of the project.

Commit Description: Resolved conflicts.
commit 05eb627f5824e1da7548db01f9f62c0f28515388

Purpose: Resolve merge conflicts with classes I had updated.

Consequences: Some spacing conflicts with my teammates upon pushing. However, nothing major. Most of our conflicts were spacing, naming, or method movement conflicts.

Timeliness: This commit was timely as we began adding extra functionality to our project for the second deadline.

Commit Description: Undid moves. Created GridEdgeRules Hierarchy.
commit 3ed2a4832ef988f21a893ee16dea502a69160170

Purpose: Refactoring functionality into a hierarchy, and resolving conflicts.

Consequences: None, fixed a problem I introduced earlier when attempting to refactor classes into more packages; we introduced a dependency in our code that necessitates the existance of two packages only, and no more (a dependency I was unaware of at the time).

Timeliness: My commit was done in a timely manner, resolving conflicts as soon as possible when my teammates informed me about it.

Conclusions

Overall, I believe our team worked very well together. While there were certain small issues in our time together, I believe overall we were effective at creating a functionally mature, and fairly well-designed simulation. We definitely underestimated the size of this project when we first began planning; we had some foresight into what the simulation might require in the future, however we mostly stuck to fulfilling the basic stipulations of the first deadline. In the future, I believe it to be better to overestimate than underestimate, as overestimating leads to more extendable code (regardless of whether we extend it or not), as well as better design overall.

I do believe I took on enough responsibility within the team, although I could have taken on more responsibility. I continued to think about how well our design was progressing or regressing as we continued to implement new features, and thus very cautiously addednew components and features. However, in hindsight, our design was good enough to the point where I should have just implemented all the features first (which would not have taken as much time as continually analyzing our design), and then refactored aggressively in the days leading up to the deadline. Throughout the project, I made sure my team was informed about what I was doing, from commits and pushes, to Facebook messaging and discussions.

The Parser, GridInfo, Properties, User Interface classes and Cell hierarchy required the most editing. The first four required a lot of editing because our original XML design was not very extendable, and adding new parameters meant we had to update the Parser, GridInfo, Properties, and potentially the User Interface classes due to the inherent dependencies among them. Lastly, the Cell hierarchy required a fair amount of editing, just because those classes contained the bulk behavior of our simulations, and we had to update them as we developed the controller classes.

To be a better desinger, I should start creating more hierarchies and classes to do single behaviors as I code. I should continue to communicate with my teammates to come to a

consensus on a best design, and I should stop worrying as much about the final design, and more on implementing more components.

To be a better teammate, I should be more clear in communicating my ideas, continue keeping lines of communication open, and I should stop being as nitpicky about implementations as the final product will potentially look drastically different when compared with the initial plan.

If I could work on one part right now to improve my grade, I would work on being a better designer and refactor all my code twice over. While our component designs are good in their extendability and degree independence in interactions, there are definitly unnecessary dependencies that still exist that need to be resolved. Also, I would implement the last two simulations, which our design is easily capable of doing.

Design Review

Status

Our code overall is generally consistent in layout, naming conventions, and style. For our code and method layouts, we follow normal coding conventions: instance variables at the tope, defining constructors, and then grouping methods and their helper methods together in blocks. For example, our Patch class contains its defined enumerations and protected instance variables at the top, then two Patch constructors, followed by its update methods, as well as other function calls. In terms of naming conventions, we tried our best to maintain the same instance variable names across all classes. We referred all pointers to GridInfo as the _infoSheet_, to the Grid as _grid_, and to Cell and Patch instances as _cell_ and _patch_. However, there are some hiccups that we missed, such as the one instance of Grid in the UserInterface that still refers to its Grid as myGrid. Each class contains similar styled javadoc documentation. There are certain classes, on which we each worked more exclusively (such as my SimulationChart class, Wenjun's Parser, GridInfo, and ColorPicker classes, and Davis's UserInterface classes), which exhibit code that is perhaps more personal in style. For these classes, we did not discuss the design as a group, and thus slight inconsistencies in coding style may be seen. For future projects, we will try to be more transparent about our code for classes others are not in charge of, and attempt to code to a similar level of design quality.

For most of the classes, the method's name is self evident in declaring its purpose; methods such as prepareToUpdate and update in the Cell class, applyRulesAndGetNeighbors in GridEdgeRules, and createRadioButton in the UserInterface classes do not need further explanation. There are some instances where better naming conventions, and creation of additional methods could help make the purpose of the method more clear: the initializeSimulationObjects method in the MainController is a bit too unwieldy for its own good, as well as the attributeParse method in the Parser class and the initializeChart method in my SimulationChart class.

Are the dependencies in the code clear and easy to find or do they exist through "back channels" (e.g., global variables, order of method call, type requirements, or get methods instead of parameters)?

While most of the dependencies are clear and easy to find, such as the update order from Grid to Patch to Cell; unfortunately, there are a couple different dependencies which exist through back channels. These include the references to the GridInfo class, the structure of the XML file read through the Parser, as well as the Properties file. Because many classes rely on these hidden dependencies to work properly, reorganizing methods, files, or hierarchies, as well as adding new features requires that we check whether or not they disrupt said dependencies.

Our design overall allowed us to easily add new features to our existing working simulation. Different grid tiles were easily added as an extended feature of the Patch class (in a separate hierarchy); other GUI features were easily added to the UserInterface class; different edge rules were also easily added with a simple class hierarchy and shift in method location. Additional simulations were also easily added, as we had dual extendability through our Patch and Cell combo classes. Specifically, adding the Patch made implementing location based simulations, such as GameOfLife and Segregation Cell, easy (as the location Patches are permanent and never null). Because of our thoughtful design, our Simulations (Grid, Cell, Patches) could operate independently from any of the Controller classes, as long as the stepSimulation method was called from the MainController. Further, our design allows for the easy implementation of behavior for more complex simulations such as the Sugarscape simulation (complete) and the AntForaging simulation (incomplete). However, because of the way in which our XML file, Parser, and GridInfo classes operated, additional simulations were also problematic in another sense, as updating specific attributes of all the afore mentioned classes is necessary, as our XML and Parser only handle very specific tags, thus leading to creation of addition cases in the Parser, as well as additional getters and setters for new fields in the GridInfo class.

The SimulationObjects package and the Grid would be easily tested for correctness, as they operate based on local parameters and take on definite states which we could print and/or compare in a tester (which is possible even without displaying any of the SimulationObjects). The MainController and the Parser may be difficult to test, as extensive test cases must be created as inputs (various XML files for the Parser and various initializations for the MainController). Additionally, the ColorPicker may be difficult to test by itself, as chosen colors are arbitrary on any given simulation. Lastly, the GridEdgeRules and the PatchBody classes may be difficult, necessitating extensive in puts (a grid, patches, an a cell object for the rules, and patches for the PatchBody).

Checking through our project, I found no bugs in others' code.

In order to implement Patches and make the WaTor World/Predator Prey simulation work, I had to read up on the MainController, Parser, and GridInfo classes, as well as apply changes. Thus, I learned and understood how they worked. The Parser class took a

little bit to understand, as Wenjun consolidated most of the functionality into one very large method–attributeParse. However, I was able to implement all of my desired features having read and understood its implementation. Similarly, in order to implement Patches, I had to read and understand the MainController, and how it initializes Cells and Patches. Davis later refactored my method into a Patch Factory class after we decided to try implementing factories into our project and improve our design. Further, I had to refactor Wenjun's ColorPicker class, because unfortunately he decided to make the Grid in the MainController a _static_ variable. The ColorPicker class took a fair amount of time longer to understand than the the previously mentioned classes, and unfortunately I still do not understand why the ColorPicker is itself given an instance of a ColorPicker.

In order to make our code more clear and maintainable, as well as more elegant, extensive refactoring needs to be applied. First, we need to make sure we follow the principles of SOLID, especially the principle of Single Responsibility and Interface Segregation. Most of our classes contain too much functionality, and need to be refactored into other classes to clarify our design, and relieve our main controller classes of handling too much logic and information. For example, our Grid class should just hold the elements of the grid and the data related to it, while we should delegate the Grid's additional behaviors to other classes, to clarify the role of the grid. Furthermore, button classes as well as factories could be created to generate individual buttons for placement into the UserInterface class; the same could be said for Cell and Patch factories in the MainController. Lastly, completely revamping our XML and Parsing design could greatly improve the design of our Simulation overall, as much of the dependencies between our controller and SimulationObject classes are due to these dependencies.

Design

Our overall design was split into two main groups: the Simulation Objects and the Controllers (however aditional subpackages and interacitons may be classified). Our simulations were controlled mainly by the Grid class keeping instances of all SimulationObjects and neighborfinding, and the Patches and the Cells implementing all the behavior independently each step of the program. In designing our Simulations this way, we could maintain a closed design of the Grid, which served only as a container and location search class, while the Patches and Objects implemented their own behavior through status changes. Through this implementation, we attempted to follow the Dependency Inversion principle of design. We included both a Patch and and a Cell because we ultimately realized that keeping Patches as a permanent object in the Grid allowed for greater extendability of the Simulation in the long run (if we wanted the simulation environment to have its own indpendent behavior in interacting with the Cells). Further, in having Patches serve as location objects, we would never have a null location in the Grid, and we could even extend the Grid to have an Infinite Edge by creating and adding more Patches to the Grid. Lastly, Patches allow for further extendability, especially in the case of implementing a hexagonal and triangular grid: our Patches extend the Polygon class, and thus can easy take on whatever shape with

whatever neighbors we would like. The SimulationObjects, however, greatly depend on the data on other SimulationObjects which the Grid contains.

Our Controller Package contains the bulk of the configuration and visualization of our program.

The configuration for each simulation is loaded through an XML file into the Parser, which parses and saves a copy of all the information from the XML file into the GridInfo class, which holds initialization information. The MainController then takes this information and initializes the Grid, the Patches, and the Cells (all the Simulation Objects and Settings). We separated the Parser from the MainController because the Parser itself had enough specialized functionality that it needed to be differentiated. The MainController serves mainly to initialize the Simulation, and thus the bulk of the instantiation happens there. An improvement on this design would be to have a specialized Factory class to specially instantiate all of the Cell and Patches, and thus simplify the code logic within the MainController class. The Main dependency here is between the MainController and the GridInfo and Parser classes; the bulk of the data used for simulation initialization in the MainController comes from those classes.

The visualization of our program involves the UserInterface class, the Parser and GridInfo classes, the SimulationChart class, the Grid class, the ColorPicker class, and the SimulationObjects in general. Unfortunately, this part of our project involves the greates number of dependencies, as the colors of our Simulations are defined within our Simulations (which we were going to refactor). Otherwise, visualization is initialized through the UserInterface class, which both displays everything to the user, and also take user input (through buttons, etc) to customize a simulation (for speed, simulation type, edge type, and patch shape). Increating these dependencies, we allow the user to update simulation parameters in realtime (the grid shape can be changed real time, while maintaining the integrity of the Cell and Patch rules of the simulation). The SimulationChart gives a realtime visualization of the different cell populations being simulated by taking data on all the Cells and Patches in the Grid, while the ColorPicker class uses the Grid to traverse all Cells and Patches during its update method calls to change the color of anything in the Grid.

Adding a new simulation the the program is simple (although could be simpler with XML style, Parser, and GridInfo refactoring). First, the rules of the simulation, in terms of environment parameters for the Patch and Cell behaviors must be programmed as a new Cell and/or Patch type. Next, an XML file, matching the style we designed for the simulation must be created to initialize the simulation. Lastly, the Parser and the GridInfo classes must be updated in case specific additional parameters must be defined and saved for the simulation. Nothing needs to be changed with the Grid, MainController, or UserInterface (as we desired in our Open/Closed design). Having added/changed these things, launching the application with the new XML file will give the working simulation of whatever Cellular Automata the user desires.

We made sure to discuss as many design considerations as possible, in order to make sure everyone was on the same page, and capable of working independently at the start of our project.

One of the biggest design considerations we decided on was to include Patches. Originally, we implemented the CA simulations without Patches, having our Cell class hierarchy implmement and hold all location information and simulation rules. This former implementation seemed to be a quick and easy solution to fulfill the initial project requirements, and it was difficult to convince my teammates to include an implementation that was not explicitly stated as a requirement of the project. At the time, I pushed for Patches because I believed it would lead to greater extendability of our design, provide a constant state in our Grid (to avoid null pointer exceptions), and relieve our cell hierarchy from becoming overstuffed with funcitonality; however, at that point I could not provide adequate concrete evidence to support the need of the extra extendability. However, my idea gained traction after further discussion of Patches was made in class, and the Game of Life simulation was added to the list of requirements. Ultimately, we successfully implemented Patches, to the benefit of our design and functionality into the second part of the project.

Another big design consideration was to have Cells (and then the Patches) perform and hold the actual behavior and rules. It made the most sense to define the specific cell class with the specific behavior it represents (i.e. a Shark cell should act like a shark, and a Fish cell like a fish). We considered other implementations where we extended a "Rules" hierarchy instead of the Cell hierarchy, or a "Simulation" hierarchy, however the abstraction did not fit as appropriately as just defining a specific cell with behavior. Further, it

Another big design consideration was whether or not to implement a one stage update method, or a two stage update method: the first stage saving the states on the board/setting up for the updating stage, while the second stage doing the actual work of updating. The argument for the one stage implementation involved the lack of initial necessity for the two stage implementation: since everything is updated in a linear fashion (array traversal). However, after seeing more complicated simulations, such as the Predator simulation, save state on each update, it became clear that a two stage update method would allow greater flexibility and extendibility for future simulation implementations.

Next, we also discussed the style of the XML file and Parser at length, and how best we could create tags that could read state types and parameter lists of any kind. Ultimately we were unable to implement the changes in time, however had a quick fix by reading in a list of integers as all of the parameters, which is not particularly robust, but works for simple Simulations.

/////////

I just realized how much I actually wrote for this, if you, the awesome grader, actually read to this point, I give you mad props, and would offer you the highest of fives. For you, either Kevin or Professor Duvall or other TA or UTA, are Legen–wait for it because I just realized I need to buy some more–dary.

////////

My code was designed with a minimalism and hierarchies in mind, that is, what is the most functionality I can obtain from the least amount of code. This is evident in my Patch design as well as my Grid, and GridEdgeRules design.

To choose a feature and talk about it in detail, I choose my GridEdgeRules Hierarchy. GridEdgeRules is a simple, yet powerful solution to defining specific boundary conditions for the Grid. By giving the Grid a specific set of GridEdgeRules, the user may change a huge aspect of the simulation, with relatively little work.

Ideal Design

Our original design was actually able to handle most of the project's extensions. One of the few main changes we had to implement was simply changing Patches from extending the Rectangle Class to implementing the Polygon class, which was a fairly small change. From edge types, to additional graphical data, to grid "tile" shape changes, to more complex simulations, our design could handle everything. We could probably implement Tesselations and an Infinite Edge as well, and potentially even realtime saving of an XML file, however we lacked the time constraints to actually do so.

Our ideal design would be similar to our current one, except that our XML design would be able to handle all potential parameter types, as well as have a Parser and GridInfo that was more extendable to additional changes. Currently, we could have also implemented more design patterns according to the SOLID principles in order to make our code more readable: creating more subclasses, as well as creating Factories to generate any type of Cell and Patch hierarchy with any type of input.

In this ideal version, a minimally specialized Cell and/or Patch class would be the only things added to the Program, as well as a specific XML file. However, users would also be able to create their own Cells and Patch paramaterizations from a GUI that pops up, letting users define their own Cell behaviors. (abstracting away the need for actual coding on the user's part). In the end, it would serve as an actual product that could be marketable to the public.

Our current design is deficient because of the XML, Parser, and GridInfo dependencies, which greatly restrict the extendability of our program. In order to add to our program, we must always make sure those dependencies are not altered so as to cause the rest of our simulation to fail upon starting up.

In terms of only having the user need to update a Cell, Patch, and XML file, that design is only so good as to assume the needs of the user and the user's ability to code. If those abilities and needs are not assumed, then a completely graphical, and real time, functionally programmable Cell, Patch, and Simulation file should be generatable by the user. This design would be the ultimate in abstraction from the user needing to see how the code behind the program works at all, as well as simplifying simulation creation as much as humanly possible.

Code Masterpiece

```
package controller;

import java.util.List;

import simulationObjects.Patch;

/**
 * Parent class for the grid's edge rules
 *
 * @author Will Chang
 *
 */
public abstract class GridEdgeRules {

    protected int myXBound;
    protected int myYBound;
    protected Grid grid;

    /**
     * Constructor for the grid's edge rules
     *
     * @param x
     *          upper bound
     * @param y
     *          upper bound
     * @param g
     *          grid reference
     */
    public GridEdgeRules(int x, int y, Grid g) {
        myXBound = x;
        myYBound = y;
        grid = g;
    }

    /**
```

```
    * Method to apply each subclass's specific set of rules
    *
    * @param nextX
    *            coordinate to check for a neighbor
    * @param nextY
    *            coordinate to check for a neighbor
    * @param neighbors
    *            list of all potential neighbors
    */
   public abstract void applyRulesAndGetNeighbors(int nextX, int nextY,
            List<Patch> neighbors);

   /**
    * Checks if a location is not in the grid
    *
    * @param xCoord
    *            in grid
    * @param yCoord
    *            in grid
    * @return true if out of bounds, false otherwise
    */
   public boolean isOutOfBounds(int xCoord, int yCoord) {
         return xCoord > myXBound - 1 || xCoord < 0 || yCoord > myYBound - 1
                  || yCoord < 0;
   }

}




// This entire file is part of my masterpiece.
// Will Chang
package controller;

import java.util.List;

import simulationObjects.Patch;

/**
 * Default grid rules.
 *
 * @author Will Chang
 *
 */
public class DefaultEdgeRules extends GridEdgeRules {
```

```
    public DefaultEdgeRules(int x, int y, Grid g) {
        super(x, y, g);
    }

    /**
     * Default rules for a finite grid
     */
    @Override
    public void applyRulesAndGetNeighbors(int nextX, int nextY,
            List<Patch> neighbors) {
        if (!isOutOfBounds(nextX, nextY)) {
            neighbors.add(grid.getPatchAtPoint(nextX, nextY));
        }
    }

}




// This entire file is part of my masterpiece.
// Will Chang
package controller;

import java.util.List;

import simulationObjects.Patch;

/**
 * Toroidal grid rules
 *
 * @author Will Chang
 *
 */
public class ToroidalEdgeRules extends GridEdgeRules {

    public ToroidalEdgeRules(int x, int y, Grid g) {
        super(x, y, g);
    }

    /**
     * Sets up the Toroidal edge rules
     */
    @Override
    public void applyRulesAndGetNeighbors(int nextX, int nextY,
```

```
            List<Patch> neighbors) {
        int xWrapped = nextX;
        int yWrapped = nextY;
        if (isOutOfBounds(nextX, nextY)) {
            xWrapped = wrapCoordAround(nextX, myXBound);
            yWrapped = wrapCoordAround(nextY, myYBound);
        }
        neighbors.add(grid.getPatchAtPoint(xWrapped, yWrapped));
    }

    /**
     * Wraps a coordinate around the edges of the grid.
     *
     * @param coord
     *            out of bounds coordinate to wrap around
     * @param max
     *            boundary reference
     * @return the wrapped around coordinate
     */
    private int wrapCoordAround(int coord, int max) {
        int wrappedCoord = coord;
        if (coord > max - 1) {
            wrappedCoord = 0;
        } else if (coord < 0) {
            wrappedCoord = max - 1;
        }
        return wrappedCoord;
    }
}
```

My GridEdgeRules Hierarchy. GridEdgeRules is a simple and minimalistic, yet intuitive and powerful solution to defining specific boundary conditions for the Grid. By giving the Grid a specific set of GridEdgeRules, the user may change a huge aspect of the simulation behavior, with little effort. As well, I had refactored my original code to create this hierarchy as a way to consolidate similar functionality into other classes, as well as eliminate switch/case and if/else statements.

The design is good because each subclass in GridEdgeRules only has one behavior: it defines and applies a certain set of boundary conditions for a cellular neighborhood, in accordance with the Single Responsibility principle of SOLID. As well, it also follows from our open/closed controller design principle; this hierarchy leaves the GridRules as a necessity in Grid functionality, thus it is closed to modification, but as GridEdgeRules may be subclassed, it is open to extension. Lastly, it is an example of good Liskov substitution, in that I have created an abstract inheritance hierarchy, and

thus, the implementation details of the GridEdgeRules are hidden from the Grid, and only known by the specific subclasses themselves.

Potential JUnit Tests:

- Which parts of the project you would test
    - My grid
    - GameOfLife simulation
    - Predator/Prey simulation
    - Sugarscape simulation
    - Cell Classes, Patch classes
- what tests you would provide for each of these parts
    - I would test whether the x and y Coordinates placed into wrapCoordAround actually wrap the coordinates around to the other side of the grid for a boundary condition.
    - I would test whether applyRulesAndGetNeighbors actually applies the rules and gets the neighbors.
    - I would test if a new GridEdgeRules instance is constructable.
    - I would test if the isOutOfBounds condition returns true or false for the right conditions.
- what bugs you want to detect especially
    - If a Cell is not going out of bounds, if the GridEdgeRules would still be applied