CS 308: Software Design and Implementation

Cell Society Analysis

Brian Bolze - Group 08

Contents

1 Introduction

For this project, we developed a robust application that would simulate many different versions of Cellular Automata (CA). Throughout the code's planning and development stages, our group focused specifically on making sure the code was as readible, flexible, and extensible as possible. We also focused on minimizing the amount of repeated code, and keeping the code "Shy".

After looking back on the last three weeks, I learned a lot about proper code design practices, but even more so about effective project management and teamwork. Although I am not completely satisfied with our final product, I am pleased with the significant progress that my teammates and I have made.

2 Project Journal

2.1 Time Review

During the first week of this assignment, our group met only once on Wednesday (the day before the due date) to meet and discuss the design plans. We met for roughly 6 hours and drew up high level class hierarchies and methods on the whiteboard, and then put together a formal plan using Google Drive. In the future, I would like to meet with the group earlier on, and actually try to write some basic code to perform a sanity check on the design. What we found later on was that the design we came up with had a score of dependencies that made us unable to get much done on an individual basis without consulting the other group members heavily. Moving forward, I would like to see a more formal schedule put in place up front to assure that the group gets as much face time as needed.

The basic code phase was extremely time intensive. I began working on the code on Sunday for about 5 hours, and then every day of the week for around 5 hours each day. Thursday night however, our group met up and we all spent roughly 12 hours through the afternoon and night. Within this time, I would say that most of the time can be contributed to two main areas of design and development - unexpected dependencies and debugging. After fleshing out our group's plan, we all thought that most of the code would be independent, and therefore we could all work on our own when we could. However, this ended up not being the case and therefore a lot of time was spent looking through each others code and trying to reach other teammates for explanation and discussion. As for debugging, most of the problems we ran into actually came from within each of the different simulation `RuleSets`. For example, in the `PredatorPrey` Class, I spent over 12 hours trying to figure out how to move cells to other spots in the board.

In the final implementation, because of the way we designed the interface between the GUI and our backend logic for each simulation type, the new shapes were not terribly difficult to encorporate into our design. On Sunday, I finished both Triangles, Hexagons and Rectangles (in the form of a polygon) in just 6 hours. However, the different grid types were a lot more difficult to figure out. Our original design stored all of the cells in a double array, which would not allow us to expand the grid in the case of an infinite grid type. While Toroidal only involved changing the `RuleSet` superclass's `getNeighbors()` functions, the infinite grid was detrimental to our design and cost me over 10 hours of work. However, because of some minor bugs, I was not able to fully encorperate the infinite capability into the final product.

2.2 Teamwork

Overall, our team spent about 50-60 hours each on the project. Within that time, about 15 of those hours were spent together as a group. As stated earlier, about 6 hours was spent coming up with a plan for the project. After the plan was set, we usually met up twice a week to work on the code as a group. Although communication was fairly consistent and effective, I think it would have been more effective to meet more often in person to work together and bounce ideas off of each other.

According to our initial plan, I was mainly responsible for the `RuleSet` hierarchy and the implementation of the four simulations. As for the other members, Mike was mainly responsible for the Front-End including the GUI and the XML reading. Justin was mainly responsible for the original `Grid` class and `Cell` class. Ultimately, however, due to the dependencies between the different `RuleSet` subclasses and the `Grid`, I ended up contributing a lot to the `Grid` class and `Cell` classes. Later on in the process, Justin also helped me with some of the specific `RuleSet` subclasses.

In the final implementation, I was primarily responsible for different shapes and grid types, Justin was responsible for the new simulations, and Mike was responsible for the new XML and GUI features. However, we all ended up helping each other with each part. For example, I wrote the code for the Parameter Menu in the GUI and I wrote most of the `SugarScape` Class.

What I learned from this project is that it can be very difficult to split up work entirely, and therefore work independently on everyone's own timelines. Also, I found out that face time and pair coding is invaluable because it offers a fresh perspective and a new set of eyes to a problem that one could be working on for multiple hours.

2.3 Commits

According to GitHub, I pushed a total of 49 Commits to the Repo, consisting of 4,993 additions and 4,019 deletions, making up an average commit size of about 100 lines. However, the variance of the size of the commits was quite large. While on the Master branch (which was most of the time), I often commited and pushed very frequently in hopes to avoid merge conflicts with others and make sure everyone else had the most updated version of the code. When I was adding a larger feature, howver, I would usually create a branch and then work on it independently and merge it at the end.

The titles of the commits at the beginning of the project were not as helpful, mainly because we were doing them so often. However, the larger commits had concise and informative names.

In particular, I felt that the three commits below were significant:

- **Implemented Parameter Control in GUI**
- **Implemented Hexagonal Shapes**
- **Finished first draft of PredatorPrey**

The first two commits above were done on a branch, and then merged back to the Master after completed. Neither of which caused any merge conflicts with other members. The third commit, **Finished first draft of PredatorPrey**, was helpful because I was able to stage an initial commit of a simulation that was not yet finished, which we could then work on together as a group.

Many of my other commits, especially the ones that consisted of refactorings and code clean-ups, were not named very well, and ended up causing a lot of merge conflicts with other members. In the future, I plan on consolidating these commits into fewer, more significant edits.

2.4 Conclusions

Overall, I think our group performed very well. Although our plan was not perfect, and the responsibilities ended up blending together very quickly, we were all quick to help each other out and get things done. In the future, I would like to come up with a more formal schedule up front and have people complete specific tasks to spread the work out and keep track of the overall group progress. I would also like to test out high level design issues at the beginning before developing code that is dependent on a suboptimal module. I feel that I put in a bit more work than some others on the team, but part of that may be because of the nature of the tasks that I chose to take on. Lastly, I believe that the methods for communication between group members on design issues will be something that will continue to be fleshed out for the rest of the semester.

3 Design Review

3.1 Status

The code that we have developed may not implement all of the features requested in the final assignment, but I am mostly satisfied with our overall design. After reading through all of the code, I noticed that it is fairly easy to read and the naming conventions are fairly standard. I also believe our methods are appropriately named, and are, for the most part, fairly short. For example, during the final implementation we collaborated across many of the different sections of the code, and none of us had much trouble figuring out how the other parts work. When I built the parameters menu and the different shapes, it only took me a few minutes to figure out where I needed to add the new changes, and neither feature entailed changing much of the original source code.

Besides these strong points, our code also has a number of flaws. For example, we do not have comments and documentation in all of our classes, making it hard for someone new to figure out what all of the classes and methods do and how they interact. Also, the way we implement different shapes, simulations, and grid types involves a lot of switch statements. As we have it now, it significantly takes away from the code's readability and also leads to a lot of repeated code. In the future I would like to use reflection to choose the specific instantiation of the superclasses.

Lastly, I believe that two of our classes could be removed or consolidated. As it's implemented now, `Cell` and `CASettings` are both essentially just data objects with getters and setters without any real behavior of their own. I think it is nice that `CASettings` is abstracted to make the code a bit more readable, but we may want to put some more functionality into the class. As for `Cell`, I believe that it can be collapsed into the `Patch` and `State` classes because right now it just acts as an unnecessary level of indirection, ie. another layer of dynamic coupling.

3.2 Design

In our current design, the `Main` class sets up all of the GUI elements, sets up the `AnimatorLoop`, and starts the animation. The `AnimatorLoop` then calls multiple other classes to help with getting the simulation set up. It calls `XMLHandler` to read in the XML file and get information about the layout of the grid and its parameters. The parameters specified in the file are then passed into a `CASettings` object that can be accessed by any other class. After the `AnimatorLoop` gets the information about the grid, it calls the appropriate `ShapeFactory` to build the shapes and place them on the grid. Once this is all setup, the simulation can start. The `AnimatorLoop` will select the simulation type from a map of different `RuleSet` objects, and then it will call the `update()` method to change the color of the shapes in the grid.

In terms of class interaction, there are only a few things that cannot be easily recognized from a quick sweep through the code. Firstly, the `CASettings` object contains a static map of parameters, so that any `RuleSet` can access the most recent parameters that the program is using. Also, the only thing that the `AnimatorLoop` accesses across classes is the color of each patch. Right now, when the `AnimatorLoop` updates its GUI cells, it calls on the `RuleSet` to update each `Patch` with its next `State`, and then it dynamically calls the `Patch` and `Cell`, in order to retrieve the color from the cell's current `State`. For this reason, I think our design would improve if we removed the Cell class indirection.

The `RuleSet` hierarchy that I built actually allows a new simulation to be built very easily. One would start by creating a new Class that extends `RuleSet`, and then simply overriding its `update()` function with its specific set of rules and behavior. The new implementation could also use many of the superclass's built in methods like `getNeighbors()` and `moveCell()`.

One design issue that we struggled with from the planning stage all the way into the final implementation was the (late) `Grid` class. At first, we thought that a `Grid` class that would store all of the cells and run through all of them, calling the `RuleSet` to update each cell one-by-one. However, this caused many issues when we needed to move cells to different patches on the grid because the `RuleSet` cannot see any other places on the grid to move to. This decision also abstracted a lot of behavior from the Cell class, rendering it a passive data container. At the start of the final implementation, I decided to remove the `Grid` class and move its structure and behavior into the `RuleSet` class, which significantly improved the performance and extensibility of our code.

One example of code that I designed specifically was the `ShapeFactory` hierarchy. For the initial implementation, we had the `AnimatorLoop` build a `Rectangle` object for each patch, and we placed it in a `GridPane`. However, this design obviously did not work for other shapes. Therefore I decided to the GUI "grid" a `Group` of `Polygon` objects, that would be built and placed by a `ShapeFactory` class. This way, after the `AnimatorLoop` reads the XML file and gets the parameters for the grid shape type, it can choose the specific `ShapeFactory` subclass from a map, just like for choosing a `RuleSet`, and pass it the size of the window and grid to make the specific shapes.

If one wanted to build a new shape, one would simply need to build a new class that extends `ShapeFactory` and use simple geometry to calculate the coordinates of each point on the shape. The user would then enter the dimension information in the `setDimensions()` method, and then set its coordinates in its `setCoords()` method. Lastly, the user would need to specify how the shape would be oriented and placed in a grid, and put that information in the `move()` method. After that, the superclass does the rest of the work!

Although there are many strengths of our current design, I currently see a lot of room for improvement. Beyond basic commenting and styling issues, I think that our code is still not "Shy" enough. For example, we decided to make the grid of patches a 2D array, which makes expanding grids impossible. In the future, I think it would be more appropriate to use an ArrayList of ArrayLists to store the grid. I would also like to remove the `Cell` class, as it is currently just a passive data object and the history of past states is never actually used. I would also like to abstract a lot more functionality out of some of the `RuleSet` subclasses so that one could build a new simulation quicker and easier.

3.3 Ideal Design

With our current design, it is relatively easy to add new simulations, but adding new functionality like different grid types and changing parameters is rather difficult. In an ideal design, I would begin by abstracting as much functionality up from the specific `RuleSet` subclasses so that new simulations could be built more quickly. I would also change the grid storage object into an

ArrayList of ArrayLists to be able to implement an infinite grid. Also, although the front end was not formally my responsibility, I might put the XML file reading and writing in the `Main` class as opposed to the `AnimatorLoop` so that the behavior's of each are more consistent.

Also, although we have not been taught exactly how to use reflection to specify subclasses, I think this would greatly improve the readability of our code and help minimize repeated code. I would also collapse the `Cell` class into the `Patch` and `State` classes to remove the dynamic coupling while updating.

Besides these changes, I think our code styling and documentation practices need to be improved significantly. I would like my teammates to cite their work where it is due both so I can recognize who did what and so that I know who to contact if I have questions about a particular class or method. I also think there is a lot of room for method shortening and abstraction.

4 Code Masterpiece

For my code masterpiece, I chose the `ShapeFactory` class hierarchy that I built to implement multiple shape types. I believe that it is good code for a few reasons. Firstly, no other part of the program cares about what type of `ShapeFactory` is being used, and what exact shape it is building. The `AnimatorLoop` simply gives the `ShapeFactory` a grid size and window size, and then the `ShapeFactory` does the rest. Also, all of the methods in the superclass and its subclasses are all under 10 lines. Additionally, I took as much functionality as I could out of the subclasses and pulled them up to the superclass, making implementing a new shape even easier.

If one wanted to build a new shape, one would simply need to build a new class that extends `ShapeFactory` and use simple geometry to calculate the coordinates of each point on the shape. The user would then enter the dimension information in the `setDimensions()` method, and then set its coordinates in its `setCoords()` method. Lastly, the user would need to specify how the shape would be oriented and placed in a grid, and put that information in the `move()` method. After that, the superclass does the rest of the work!

The below code shows the `ShapeFactory` superclass which is an abstract class. The main method that this class performs across subclasses is its `makeShape()` method, which the `AnimatorLoop` calls to get a Polygon to put in the grid.

```java
/**
 * @author Brian Bolze
 * Inheritance hierarchy used to build shapes for the grid to use
 */
public abstract class ShapeFactory {

  private String myDescription;
  protected double myWidth;
  protected double myHeight;

  public ShapeFactory(String description) {
    myDescription = description;
  }

  public Polygon makeShape(double paddedHeight, double paddedWidth,
      double rows, double cols) {
    setDimensions(paddedHeight, paddedWidth, rows, cols);
    Polygon shape = new Polygon();
    shape.getPoints().setAll(setCoords());
    return shape;
  }

  protected abstract void setDimensions(double paddedHeight,
      double paddedWidth, double rows, double cols);

  protected abstract Double[] setCoords();

  public abstract void move(Polygon shape, int row, int col, double vPad,
      double hPad);

  public String getDescription() {
    return myDescription;
  }

}
```

The below code shows an example of a subclass that has been implemented for hexagons. Notice that much of the work has been done already by the superclass, greatly reducing the amount of repeated code and the chances for inconsistencies.

```java
public class Hexagon extends ShapeFactory {

  private static final String HEXAGON = "Hexagon";

  public Hexagon() {
    this(HEXAGON);
  }

  public Hexagon(String description) {
    super(description);
  }

  protected void setDimensions(double paddedHeight, double paddedWidth,
      double rows, double cols) {
    myWidth = paddedWidth / (cols / 2. + 0.5);
    myHeight = (2./3.) * paddedHeight / rows;
    myWidth += 2. * myWidth / cols;
  }

  protected Double[] setCoords() {
    double edge = myHeight / 2.;
    double z = myHeight / 4.;
    Double[] points = new Double[] { 0., 0., // top
        myWidth / 2., z, // top right
        myWidth / 2., z + edge, // bottom right
        0., myHeight, // bottom
        -myWidth / 2., z + edge, // bottom left
        -myWidth / 2., z // top left
    };
    return points;
  }

  public void move(Polygon shape, int row, int col, double vPad, double hPad) {
    shape.setTranslateX(hPad + (double) col * myWidth / 2.);
    shape.setTranslateY(vPad + (3. * myHeight / 2.) * (double) row);
    if (col % 2 == 1)
      shape.setTranslateY(shape.getTranslateY() + (3. * myHeight / 4.));
  }

}
```

5 Testing

If one wanted to test my code masterpiece, one could test a few different features. Firstly, one could test that all of the shapes have been implemented. After that, one test that may be useful is to check if all of the shapes are aligned properly in the grid. To do this, I have written test code that will perform both of these checks. The code for this is contained in the shapeFactories package and is titled `ShapeFactoryTest.java`.