Software Design Final Presentation

By:
Alexander Finch
Colin Woods
Mariah Moore
Peter Harris
Stefan Emmons
For:
Software Design, COSC 3011-01
To:
Dr. Buckner
Date:
5/12/2020

Introduction

The journey of this project began like many other journeys, strangers becoming friends, a daunting task ahead of them, and no idea where to start. We knew that we would be developing a maze game, but how? Many members of our group had never touched GUI design, and all of us needed to refresh our Java skills. Armed only with our wits, and Dr. Buckner's words in our hearts and minds, “read the docs”, we set forth on our journey with a few principles.
First, we must establish meeting schedules that work with everyone, and split up the work evenly. This would allow all members to observe their own skill level while working with something new, along with getting a good idea of who is more suited to work with specific parts of this project, such as coding, documenting, and coordinating. You can observe this brainstorming and planning process from the meeting document below:

Peter suggested that we all take a look at Main.java and determine what was being called, and what was incomplete. Stefan noted that exceptions in this file need to filled in at a later time.

Mariah said that we need to carefully look into all comments placed before each method in GameWindow.java. If we are told to research something we should likely do it. We all agreed with this. Colin suggested that we break up sections that require writing code between all five of us. We delegated the sections of GameWindow.java that need to be working on accordingly. Some of us will be working with another person in the same section. We have agreed to combine the efforts of both members into a single submission/push when the time to submit is a more prominent concern.

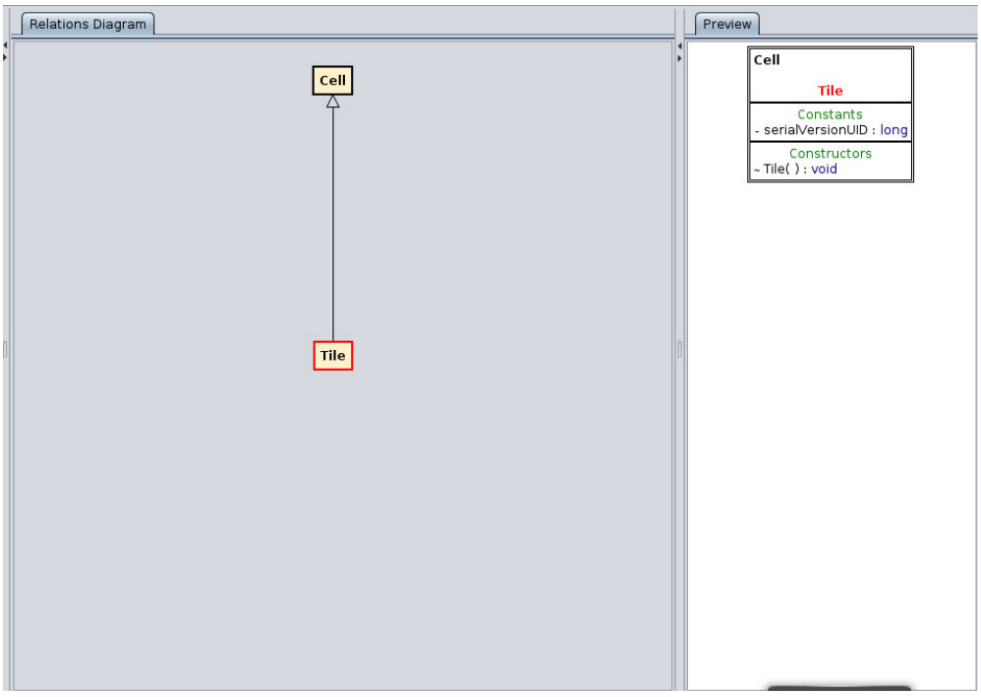
Stefan delegated the following lines of code in GameWindow.Java to the following members:
setUp() function, lines 75-90, Stefan and Alex
setUp() function, lines 90-104, Colin and Mariah
addButtons() function, Peter
Everyone seemed on board with these delegations.

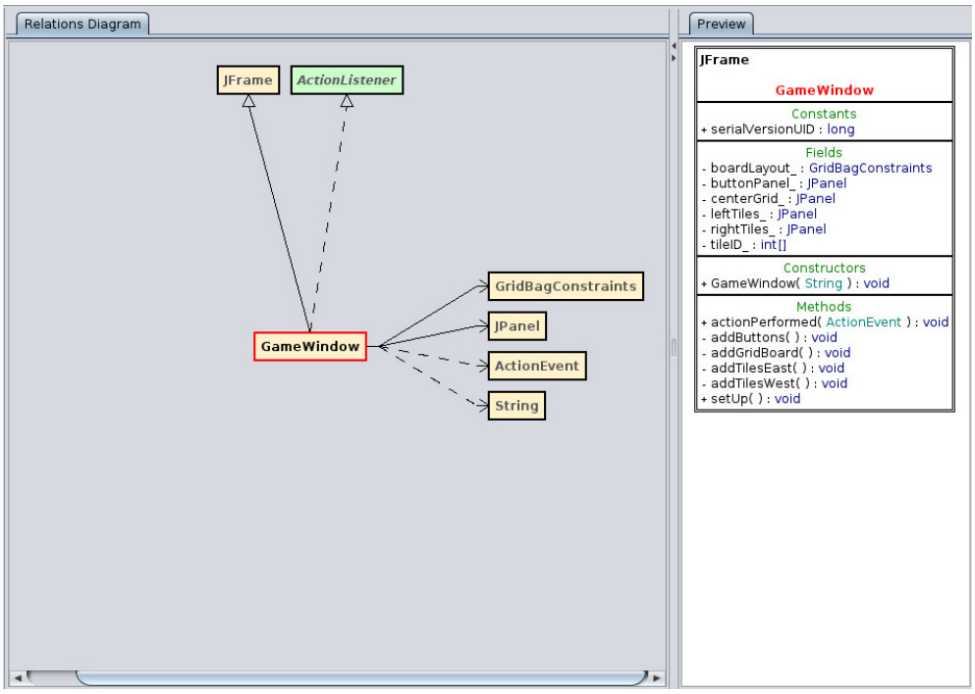
Stefan also agreed to set up a MarkDown document that allows for members to sign off on specific milestones.

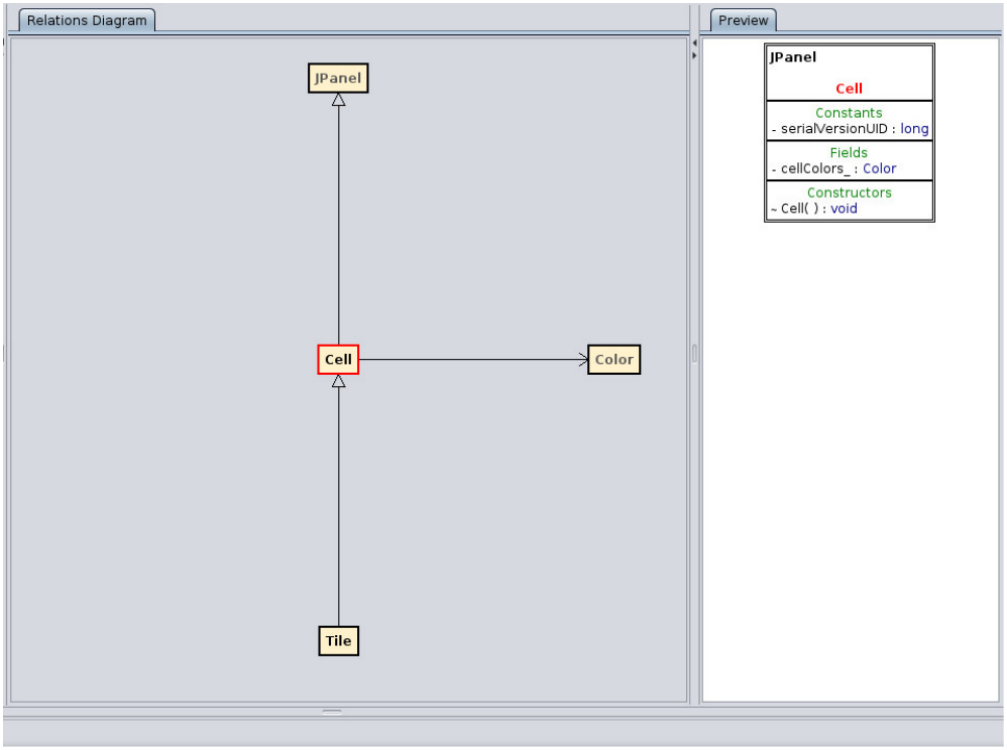
We finished up by looking at a few of the recommended interfaces and methods. We ensured that we will always contact each-other when a serious problem arises, or when we are about to make a push to the repository. Text or email was preferred.

Meeting ended: 3:45pm

Ultimately, before we began with our drafting of source code and documentation, we needed to have UML in place, to give us a good idea of where to start:



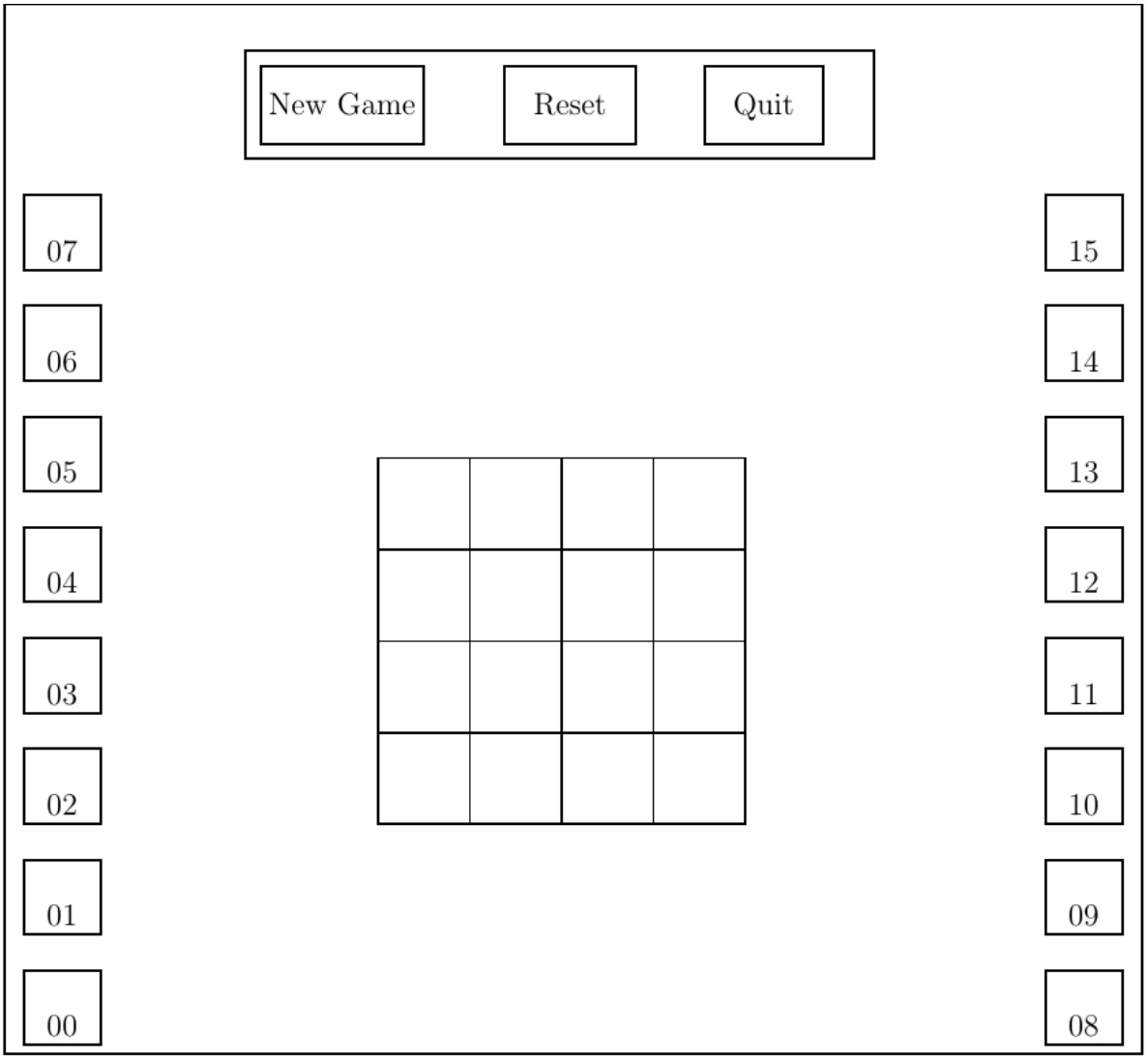




With this framework in mind, and assignments delegated, we journeyed forward to Program Iteration #1.

Program 1

For the first program, our objective was to establish the skeleton of our game within a JFrame, conveniently termed “GameWindow”. Our team was given the directive to establish a game view that looked similar to the following:



Our team was given some basic Java code with hints and suggestions, but the majority of the work was left to us. Luckily, based on the UML provided above, we had a good place to start. We needed to create a grid of “Cells” as the game board in the center of the window, as well as two columns of eight cells, each housing a “Tile”, on the sides of the grid. We also had to implement a button panel containing three buttons on the top center of the window. One was for a “new game”, another for “reset” and the last was a “quit” button. The only button that featured any functionality on this iteration of the project was the “quit” button. We were also given a friendly suggestion to use gridbag layout in the construction of all of these components.
The construction of the buttons was simple, and endowing the quit button with “System.exit” functionality was trivial. The main challenge here was to experiment with layout managers and

find out why GridBagLayout was so highly regarded as the optimal choice. We quickly found that the challenge was not to generate the Cell and Tile objects, but rather on how to position them within the JFrame backbone that we had been provided. Ultimately, we found a setup that combined GridLayout, and GridBagLayout, that gave us a game view that directly mirrored the image provided above.

Program 2

After establishing our game view, it was time to add some interactivity to the pieces. We had a board ready to accept moves, and we had columns that could also accept moves. We were given two options on how to include mouse events, point-and-click, or drag-and-drop. This initial choice divided our team into two schools of thought, those who liked the idea of drag-and-drop, and those who liked point-and-click:

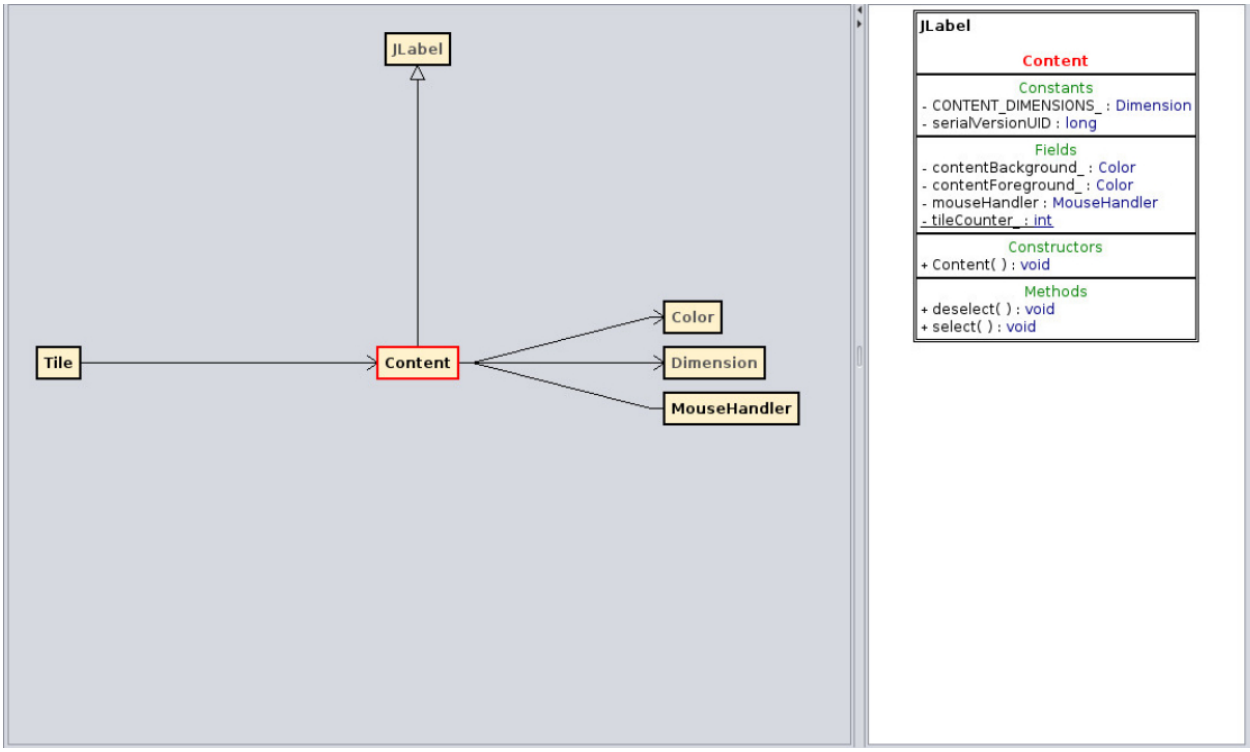
Once the meeting began, it became clear that duties for code completion would not be so clear cut as compared to the last iteration. There are likely many different methods of accomplishing panel interactivity, which includes mouse event listeners, data transfer methods, and pure select-click boolean check statements. With so many options available to us, each member agreed to complete their own version of interactivity within the time-frame of a week.

Each member agreed that if we pursue the click, and click again method, we will need to pursue a clear highlighting system that lets the user know when a tile has been selected, and when it has been deselected. Peter showed great interest in using this method, and will likely be able to present a physical code example by next week.

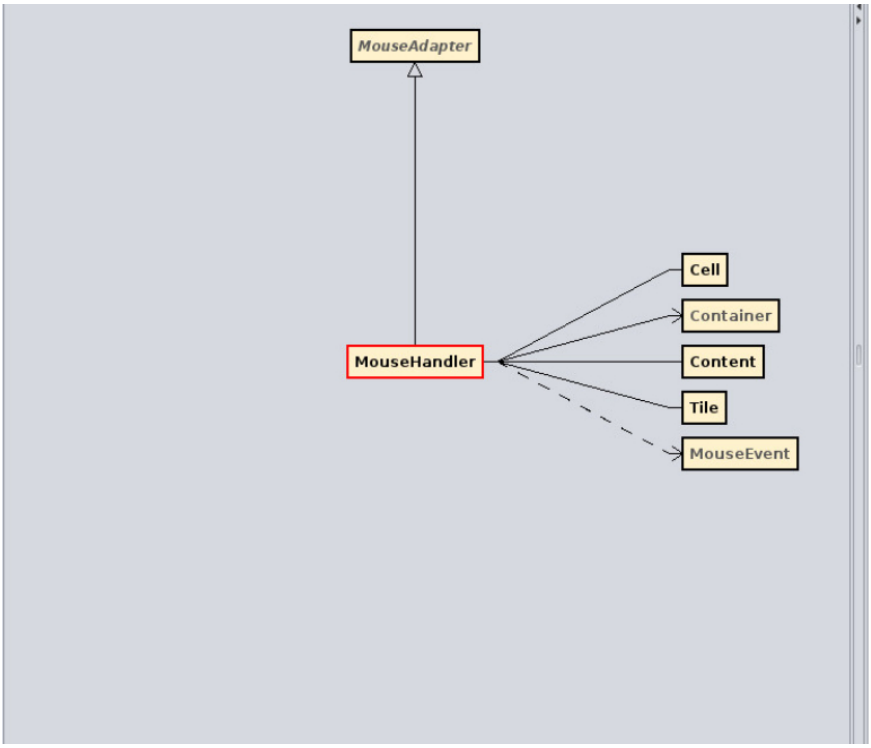
Stefan also liked the idea of excluding drag and drop, but liked the idea of a challenge when it came to building event handlers that could potentially handle the drag and drop method. He ultimately decided to go down the drag-N-drop path, along with the point-N-click path for an extra challenge.

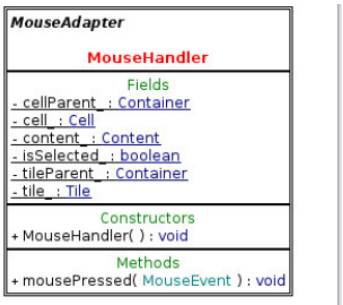
Colin and Mariah wanted to do some more research on the two primary methods of data transfer (point and click, or drag and drop), before they decided to commit on a build path that suited them.

Since mouse event handlers composed such a large part of this iteration, it seemed fair to have each team member build their own version of event handling, and present it such that all members could vote on what they liked best.
Over the course of that week, the team had come to a few realizations. First, they must generate a movable object that could be held by both Cells and Tiles. This object was dubbed “Content”, and was a perfect solution that fulfilled a “placeholder” requirement (an object must remain in the columns, regardless of user decisions, this object was now a transparent Tile). The structure of this object was the following:



They also realized that drag-and-drop was substantially more difficult. The team had developed a version that worked, but each movable object would not format itself properly each time it was moved to a parent container. The objects were fighting a layout manager issue, and no solution could be found. As a result, they abandoned the idea, and moved forward with the point-and-click version. This turned out to be much simpler, and required the overriding of only one function in the MouseAdapter interface, “mousePressed”. The new class was to be named “MouseHandler”, and interfaced with all JComponents like so:





Since this new class was much simpler, the team had time to adjust layout manager settings. They found a way to dynamically adjust the internal layout manager of a Tile or Cell object that had a Content object moved into it. In addition to this, the team developed a handy border highlighting system to let the user know what object they had just selected. Red for currently selected, and black for deselected.

After some documentation and fine tuning, the team concluded that point-and-click was the right choice for mouse handling, and moved forward with submission.

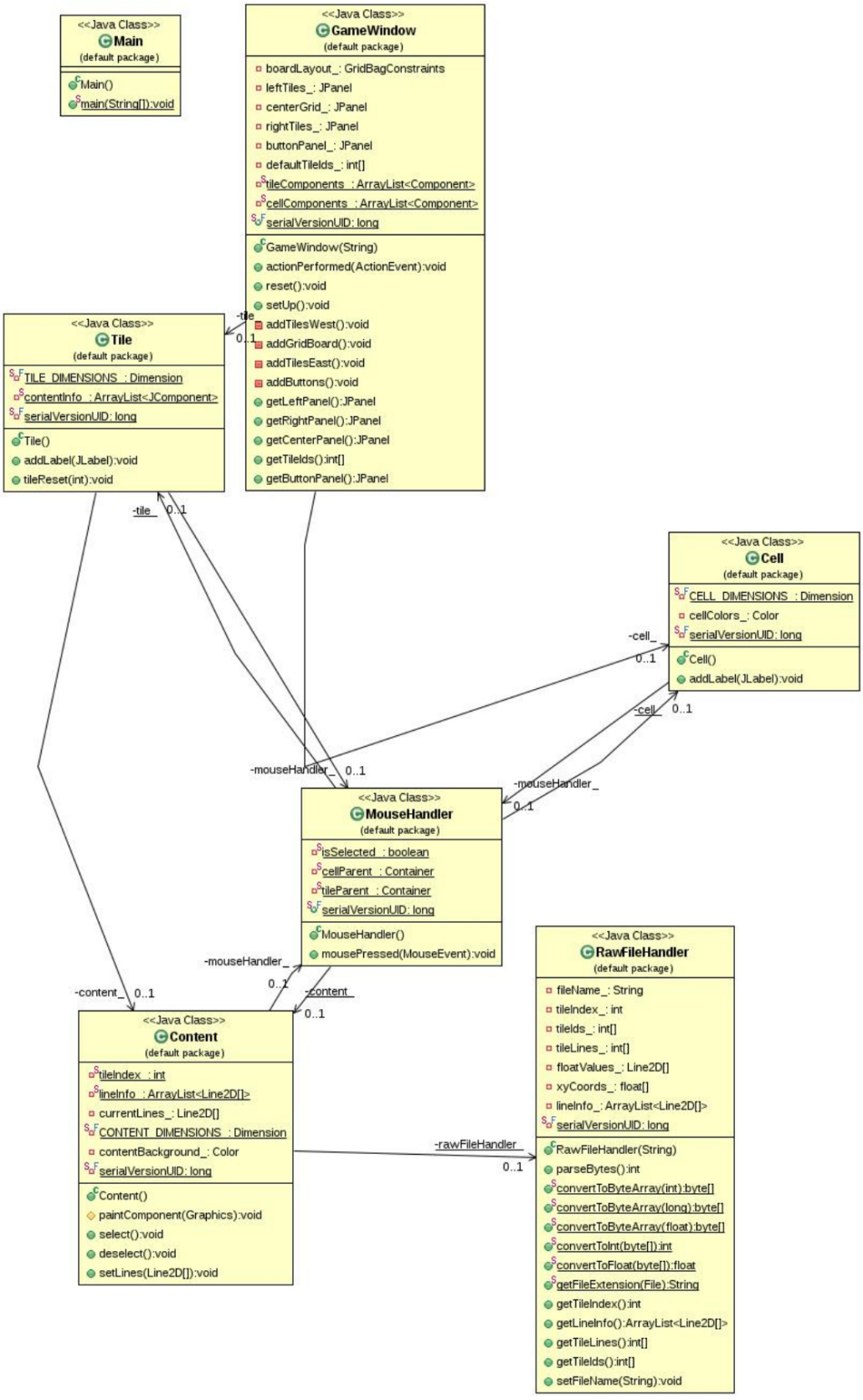
Program 3

After a successful execution of the last program iteration, it was time to work with one of the most difficult aspects of this game so far, loading from a file. Not just any file, a binary file formatted in a fashion that many of us had never seen, using a .MZE extension. It was noted that this particular file was chosen because many of us had not touched on it before. We had movable tile objects, but how could we establish custom graphics on them from this format? To make matters more difficult, this was during the time that many of us had been displaced due to the growing pandemic concerns. Despite these challenges, the team set forth, and laid out the following plan:

Our planning and meeting for version three of this project was focused around decomposing the responsibilities of the RawFileHandler class, which will get messy, and what can be brought in from this class to other objects. In terms of the reset button, we only need to slightly modify the GameWindow and Tile Class to "add" movable objects to their original locations..
The RawFileHandler will need to check for edge cases before even beginning to parse the binary file, such as checking if the file exists, or if it has the correct extension. From there, information will likely be read in sections, just as the notes/program pdfs indicate. As the file is "read", information will be read into several different data structures, such that all information from the file can be exported to other classes. For now, all we care about is line data.
The Contents class will be the main candidate to call upon the RawFileHandler's capabilities, and for the time being, should only be importing line coordinate information. It is important to override paintComponent in this class only, as this function is called upon by every opaque Jcomponent in the Jframe. Thankfully, the override tag makes it clear that only these objects need custom graphics information.
The GameWindow class and the Tile class will be responsible for reset functionality. The GameWindow class houses the buttons and the overall game setup, and so it will need to initialize the reset sequence. The Tile class is where our Content objects are original called upon and generated with a factory method, and so the original spawn positions will need to be tracked from there. Once moves have been made, a data structure can keep track of where each movable component "belongs" once a reset function is called.

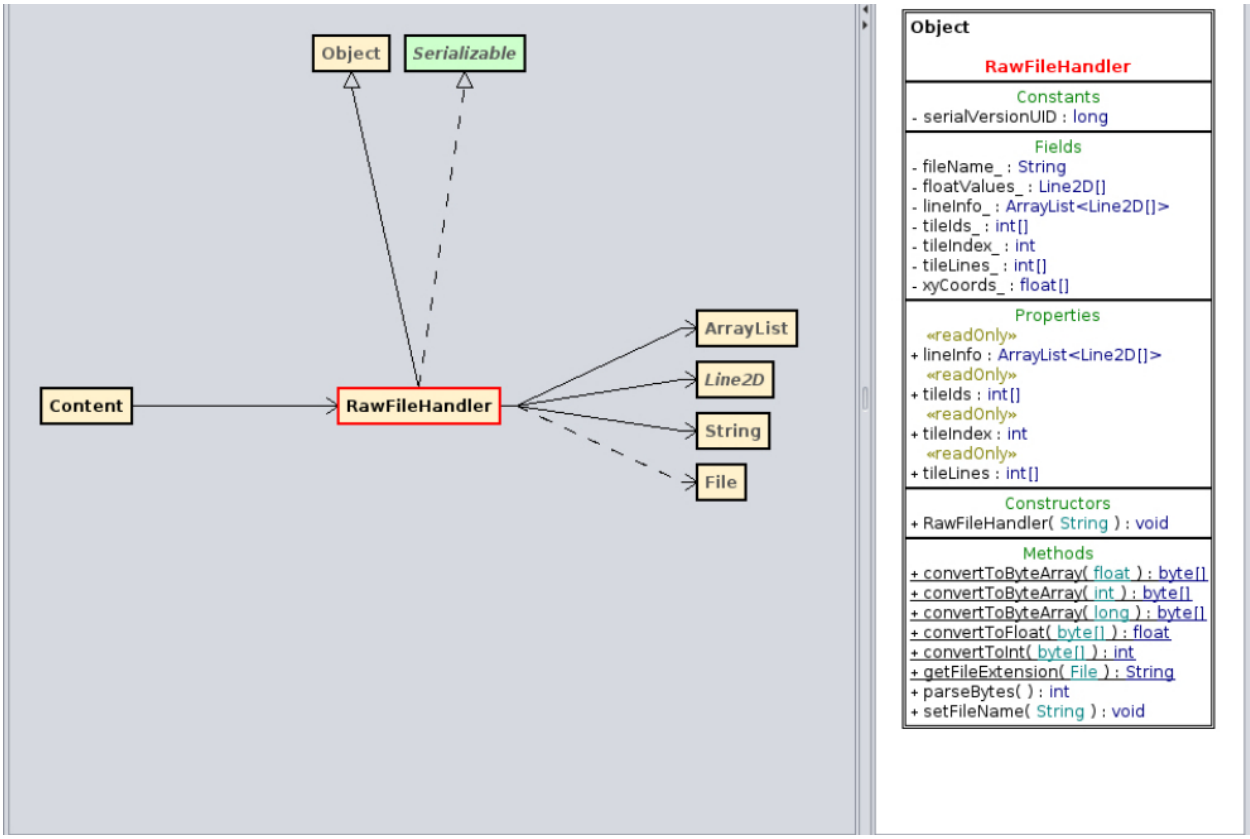
Tiles: These objects are now primed for different types of data transfer, and allow for basic stages of interactivity. We have solidified the type of JComponent that these objects will be, JLables that are hosted by JPanels, and will likely not step away from this. This is because these labels can easily be housed and manipulated within parent containers such a JPanels, without messing with the original constraints of the game board. In addition, the paintComponent and other graphic methods associated with Jlabels makes them prime candidates for future image upload and rotation, which is something that will need to be dealt with repeatedly.

We knew that a new class would need to handle all of this raw binary data, hence the name "RawFileHandler". We also knew that this class would need to load several different data structures with information that could be accessed through getters by other classes, such as Tile and Content. With this mental plan in place, we got to work putting together the following, theoretical at best, source code plan:



Based on the UML model provided above, we moved forward and ensured that RawFileHandler

was only accessed by the class that needed it the most, Content. From there, RawfileHandler would provide the most important part of this iteration, line data. This data was in the form of floating point values, and needed to be utilized through the overriding of the paintComponent() function. This overriding was done specifically for all Content objects, as they carried these custom graphics with them no matter where the user would place them. RawFileHandler used several other variables to store information such as Tile Ids and Tile indices, but these weren't of much use to us at this stage. The finished RawFileHandler class had the following structure:



Once the parsing of this .MZE file was completed, we had to adjust how our lines were displayed on our Content objects, as certain line sizes looked odd, or weren't colored properly.

On top of this, we needed to give our "reset" button some functionality. This was achieved by manipulating an Array List that was populated with all Content objects upon initial startup. When the reset button was pressed, we would add these Content objects back to their original parent container. This would "reset" their positions based on how they were originally generated.

Once this tweaking stage was done, we merely needed to document our changes, and move forward with the next iteration.

In terms of meetings and coordinating, the pandemic actually made our meeting times more flexible, as we could Zoom conference each other at almost any time.

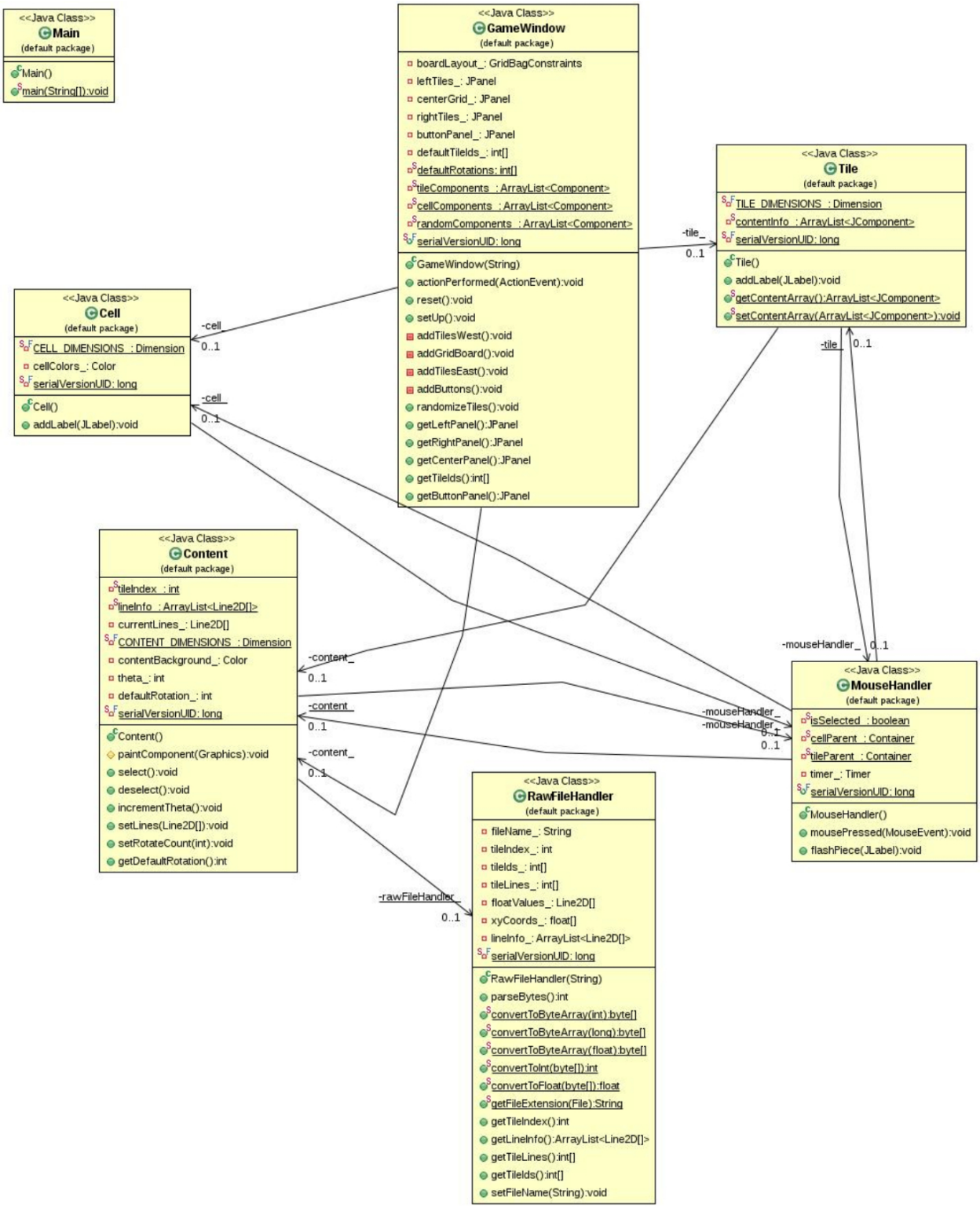
Program 4

In the fourth iteration of the program, we loaded the previously constructed tiles in a random

order, adding difficulty to the game. There would be no point in playing if the game was too easy. Along with the random order, we added a random rotation in increments of 90°. Since the player would need to change the rotation of the tiles to get them back in the correct alignment, an option was added to manually change the rotation. With a click of the right mouse button the tile would rotate 90° regardless of their location in the screen. The number of rotations would be unrestricted but limited to 90° increments. The plan for implementing these new features looking something similar to the following:

> Our planning and meeting for version four of this project was focused around selecting individual features to work with, and do a little communication in the process. Most work will be done in the GameWindow, MouseHandler, and Content classes.
> MouseHandler will likely focus on flashing/tile rotation features, as they are direct results of a mouse event. Tile rotation will need to interact with the Content class, as it has already overridden the paintComponent() function, and can easily leverage Graphics2D functions such as rotate(). Flashing will be handled with a single function call that changes to background of an offending Tile to a different color, and back.
> The Content class will be slightly modified to allow access from both GameWindow and MouseHandler. This modification should be simple, likely one or two variables associated with rotation that can be changed from external stimuli.
> GameWindow will need to be modified to accommodate tile and rotation randomization at spawn time. This should require the use of one function that is called from setUp(). This function will need to load data structures with rotation and position information, so that they may be accessed when it comes time to reset.

Our program opens a default.mze file that is found in an input folder. If the file is not present, it exits. Now, the game can exit by way of the button and if it doesn't find the required file. In addition to the exit button, functionality was added to the reset button. The reset button puts all of the tiles back to the position and rotation that they were loaded in.
There were difficulties and challenges encountered along the way. We all had to look things up and learn new things about Java. However, a unique opportunity presented itself in this iteration in the form of not needing to create any extra classes. We simply needed to expand the functionality of existing game classes. This expansion did include giving classes access to others that previously had no contact with each other, as can be seen from the overall UML plan for this iteration:

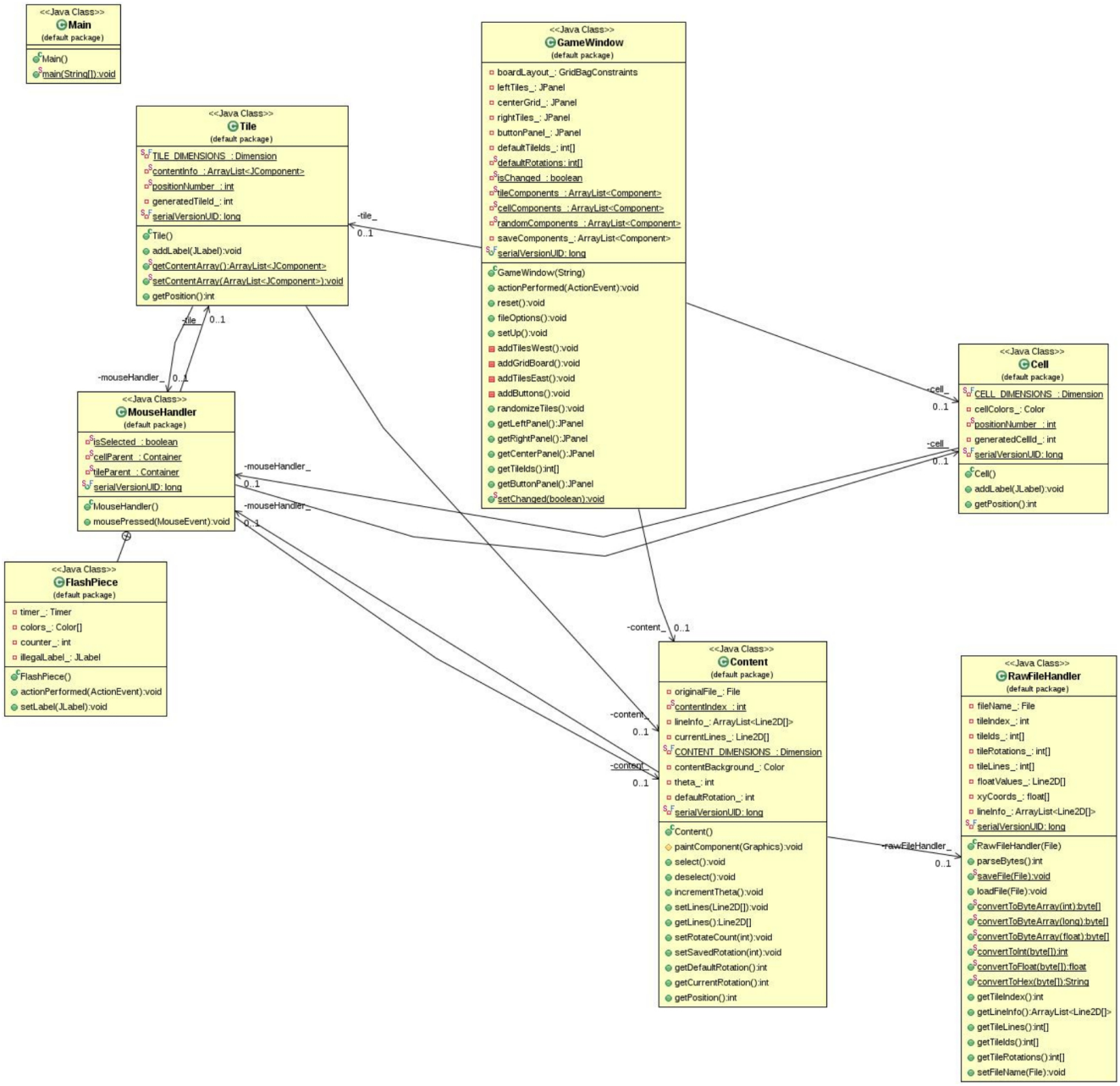


Communication with members of the team had become more difficult due to the closure of the campus. We found ways to work through all of the difficulties and still work together to complete the necessary work.

Program 5

After incorporating the new features of the previous program iteration, it was time to move forward with the next step of game development, file options. Instead of having a button labeled “new game”, the team was instructed to rename this button “file”, and give this button the functionality of allowing the user to save, or load their game. In addition to these new file changes,our team was tasked with removing any anonymous classes that were present in our program from the previous versions. This meant that the only section of our last program that needed revision was our “flashPiece” operation. Now an inner class, this operation handled all illegal moves pertaining to our “Content” objects. These moves would consist of attempts to put one movable piece on top of another. If this was attempted, the offending piece would “flash” at the user.

Our team knew that saving and loading files could be done in much of the same manner and structure as the original default file parser, with some subtle changes. The biggest changes that would need to be made, would be to integrate GameWindow with RawFileHandler, along with all the getters and setters related to Content objects. We would need to effectively export and import position information, line information, and rotation information, all in the form of primitive integers and floats. Most of this information was easily accessible, and readily developed. The biggest problem was tracking the position of all movable pieces at save time. Luckily, a clever function was developed in all Content objects that could track the immediate ID of any parent container it immediately resided in. The rest of the program was planned out like so:



As you can see, this iteration did not require the creation of any new classes (beyond FlashPiece). Instead, the team focused on expanding the integration and functionality of existing classes to fulfill the new requirements. Along with the previously mentioned file options, this program version needed to handle “bad” file loads, new reset positions based on previous file loading, and accommodating for the situation of when a default.mze file is not present. Unfortunately, some of these features were overlooked by team members, and so this version did not accommodate well to a default file being not present, and did not reset piece positions based on new file load information. The team took these mistakes in stride, and moved forward with the last iteration.

Program 6

This was it. This was the last step of project completion for the team. In addition to accommodating for new features, such as a timer, and a win condition check, the team needed to fix issues with the previous game version. Most of these fixes pertained to file loading scenarios, which required a great deal of piece adjustment and error catching. The team was determined to create the most robust, beautiful version of their game they could. To achieve this, the team laid out the following plan for themselves:

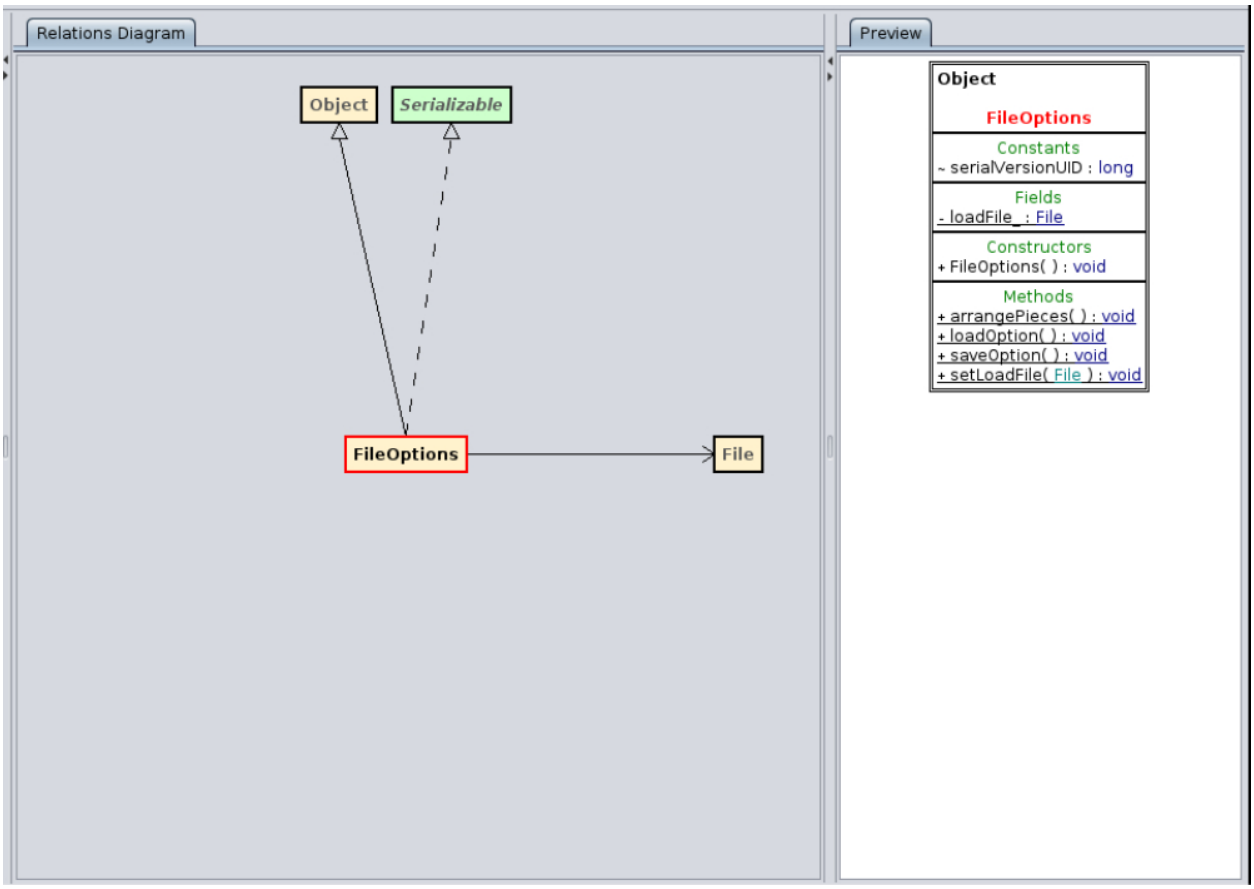
Stefan began the meeting by going over some of the file option issues that had been brought up from the previous submission. He agreed to get them fixed, as most of the issues required some minor tweaking. Colin noted that file options should be separated into its own class because of the fact that it contains so many catches and operations unique to only file options.

All members agreed that this was a good idea, and Stefan agreed to shift what he could to another class. All members seemed confused as to how a winning solution could be read from the file, especially if user rotations need to saved, yet they must be set to zero at save time. They agreed to look away from the file solution, and instead use the piece ID tactic that had worked well for saving positions.

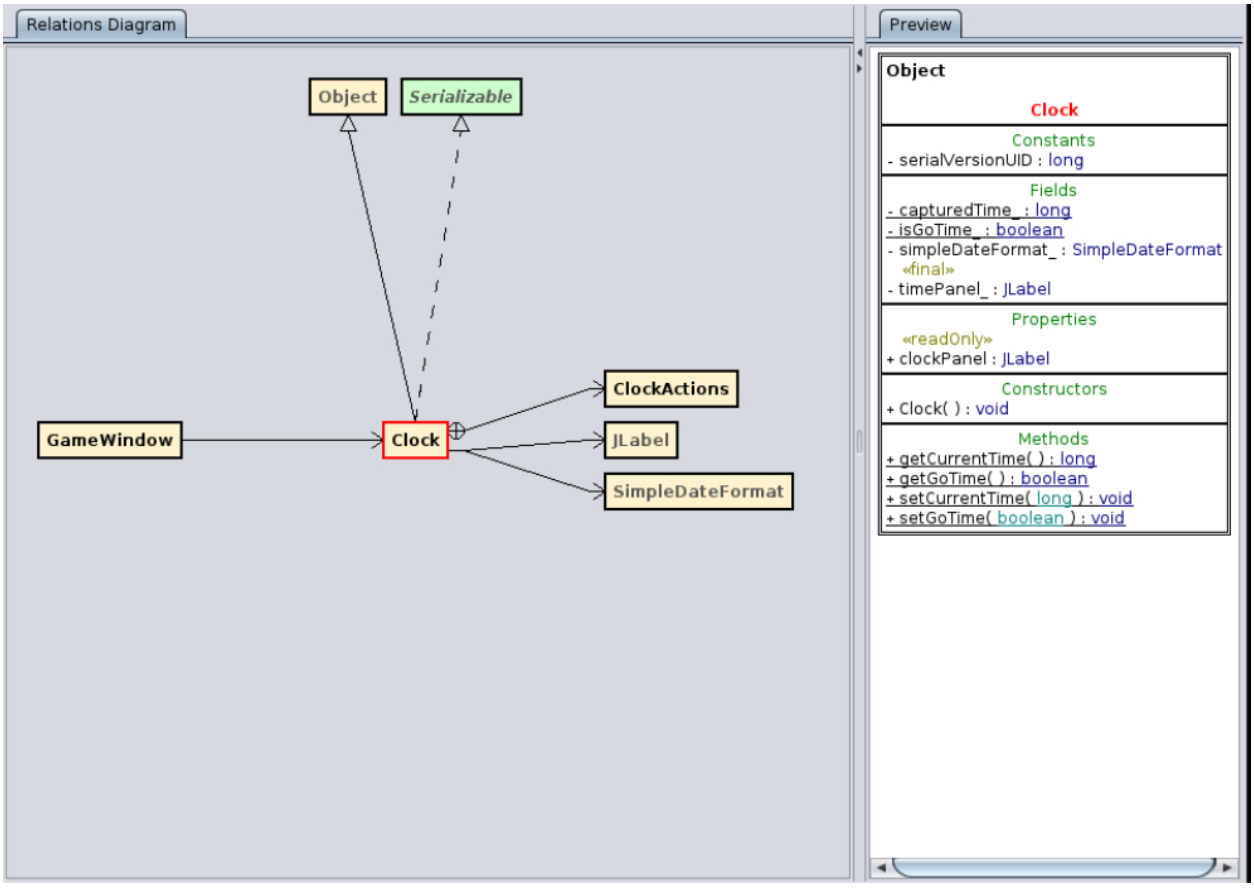
Without randomization, all members noted that the Content pieces are generated in order of a solution, that is, the top left Content object is the first piece of the maze, and the bottom right is the last piece. Keeping this in mind, they also knew that all cell objects are generated in the same sequential order. So, the top left Content piece should go in the top left Cell object, rinse and repeat. Upon trying this tactic, the members found that all pieces lined up to form a maze with one entrance and one exit, with no weird overlapping. This must constitute a solution, and so a win condition check that takes a look at rotation, and ID values of both a Content object and Parent object was decided upon.

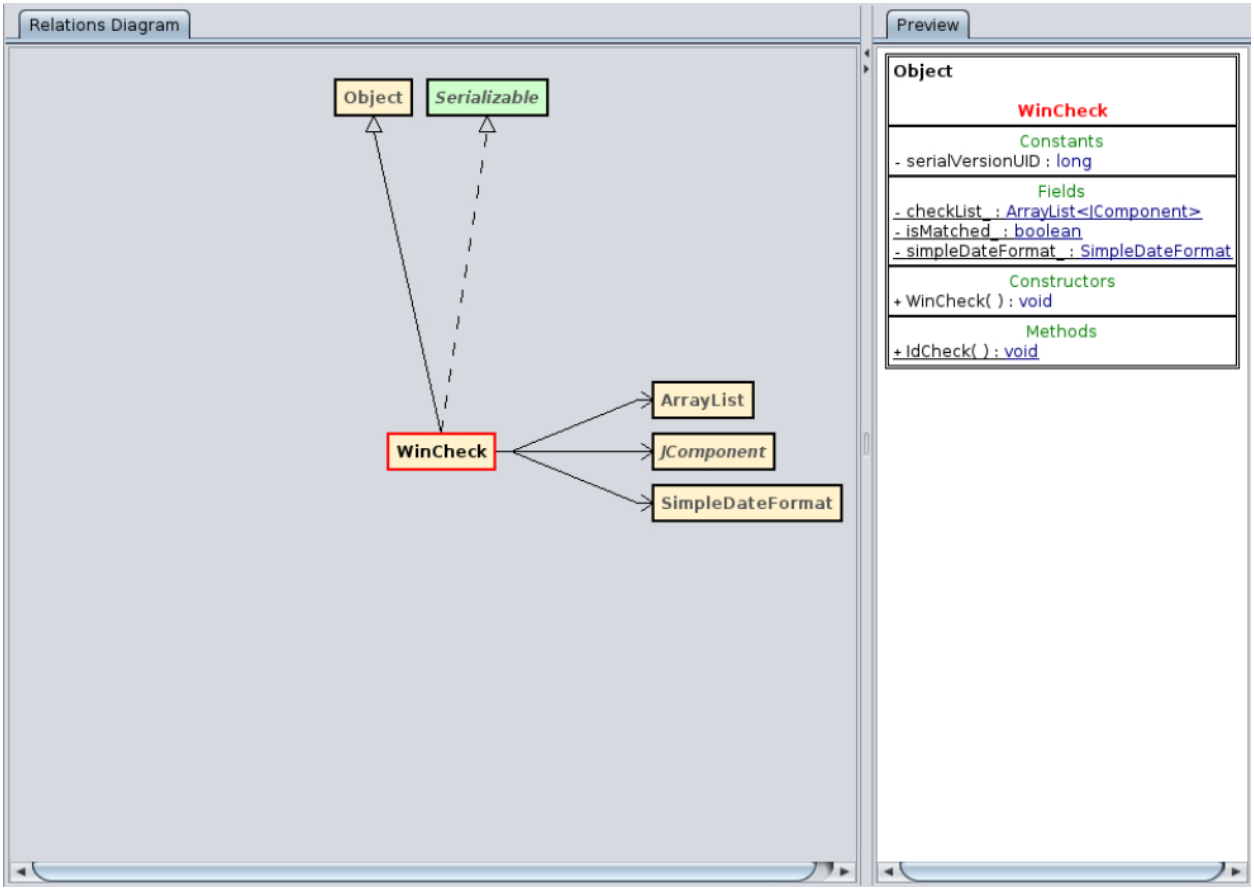
Stefan and Alex agreed to work on the Timer, as it would likely use a Swing Timer similar to what was used in Flash Piece.

In an effort to encapsulate file options, and all their related catched and operations, a new UML diagram was created for a “FileOptions” class:



In addition, two new classes would need to be created for a game timer, and a win condition check that is to be triggered each time a mouse event is executed:


Relations Diagram
Object
Serializable
GameWindow
Clock
ClockActions
JLabel
SimpleDateFormat
Preview
Object
Clock
Constants
- serialVersionUID : long
Fields
- capturedTime_ : long
- isGoTime_ : boolean
- simpleDateFormat_ : SimpleDateFormat
«final»
- timePanel_ : JLabel
Properties
«readOnly»
+ clockPanel : JLabel
Constructors
+ Clock( ) : void
Methods
+ getCurrentTime( ) : long
+ getGoTime( ) : boolean
+ setCurrentTime(long) : void
+ setGoTime(boolean) : void



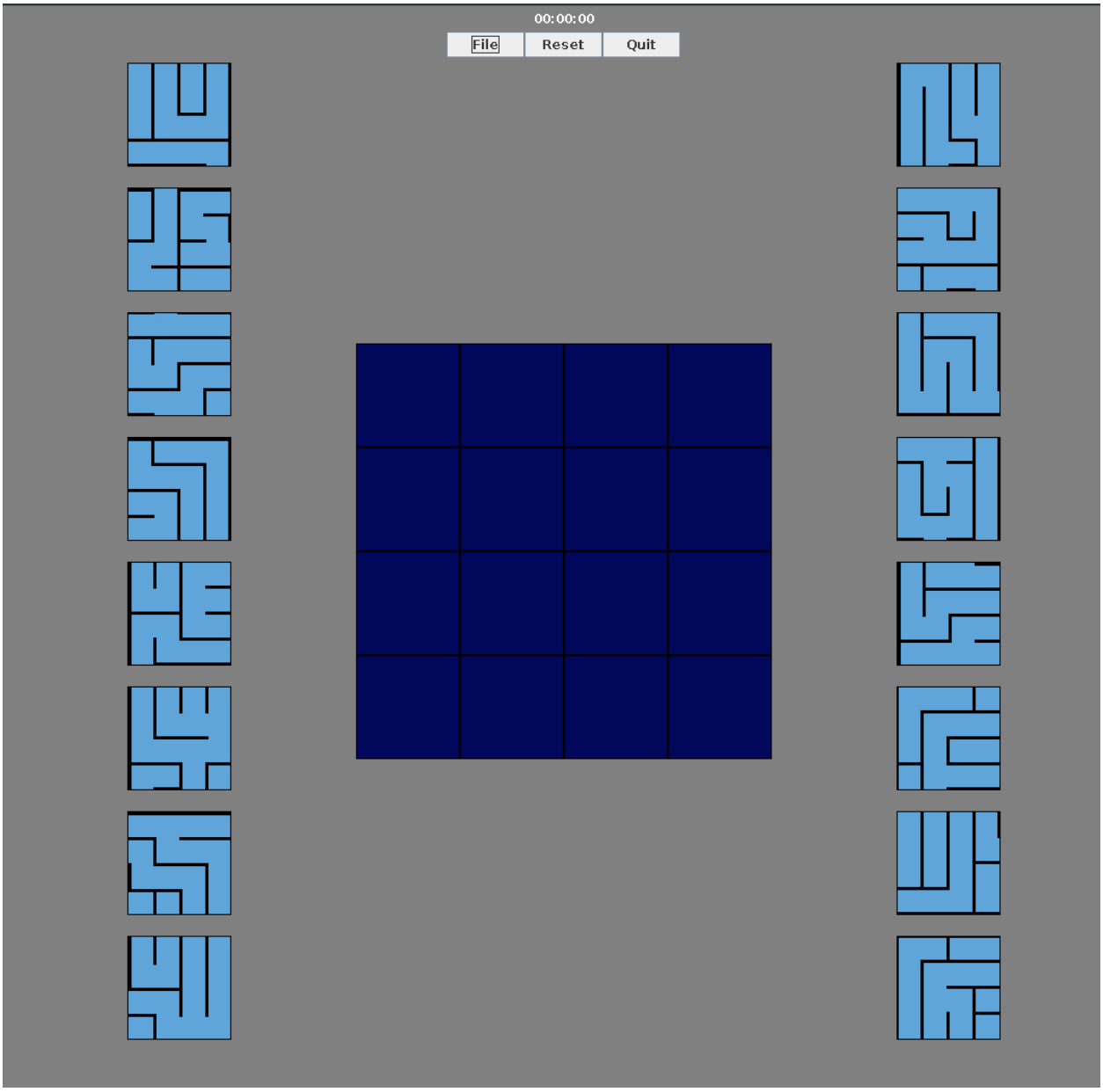
Overall, the game structure was gaining a new inner class, and three new public classes. These would be the last class additions, and were fairly simple in comparison to some of the other classes that had been present since version one.

The new features of this game version were fairly simple to implement, and took little time to complete. The time played so far was stored as a long float value, and was easily loaded or saved with the rest of the game information. The biggest task was to ensure that the file options for this game were as robust as possible, especially for loading edge cases and exceptions. The team ensured that the user would always be notified if a bad file was loaded, and that reset positions would be modified upon a new file load. In addition, if a default file was not present, the user would be notified, and allowed to load any saved game, as long as the hex values matched a valid file format. After these edge cases were accounted for, team members performed extensive testing to ensure that they could not find any hidden issues. Some members even had students from other departments test the game to ensure that it made sense, and was completely solid, even to a complete stranger. The resulting submission was something that they could all be proud of, and ensured that it was a good representation of a semesters' worth of work.

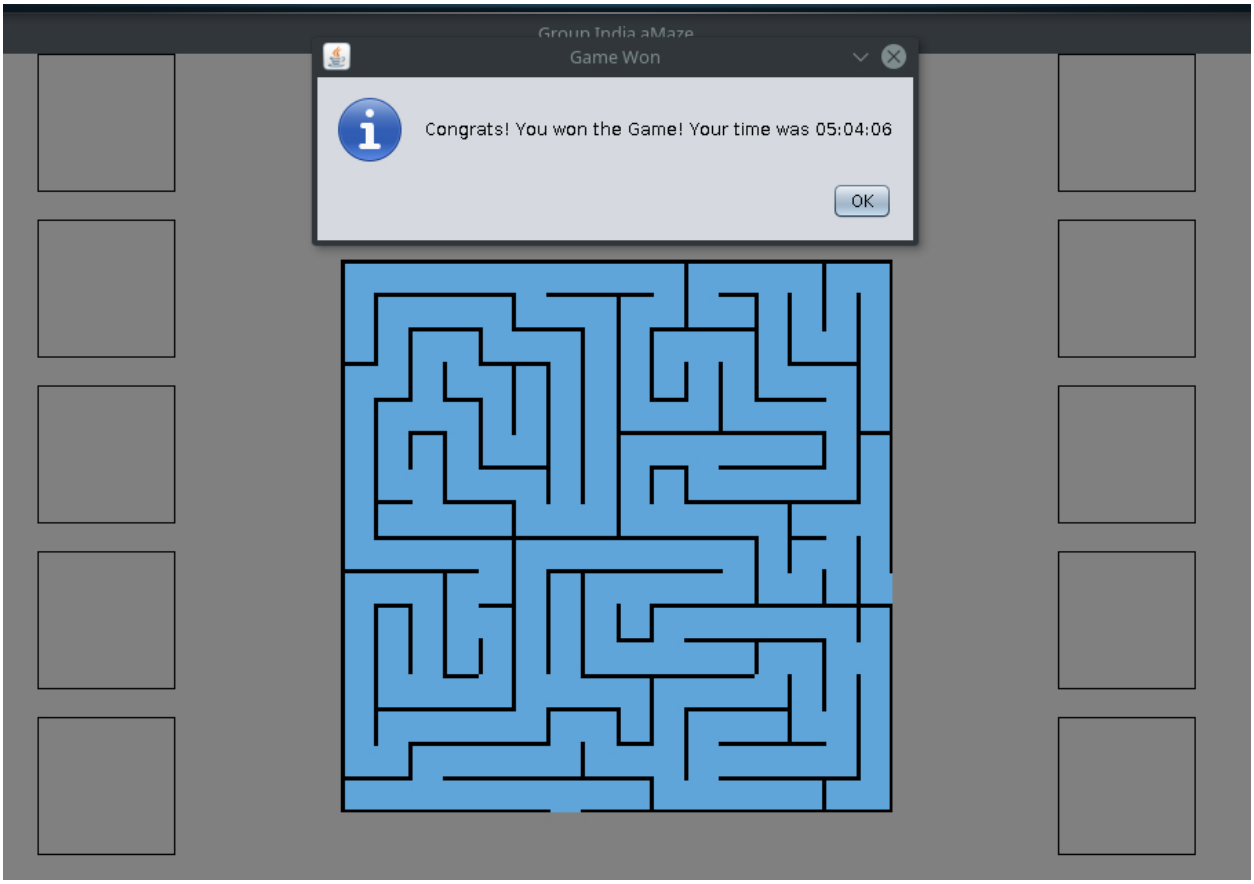
Conclusion

Over the course of the semester, this team went from being Java GUI novices, to modestly skilled game builders. Through many months worth of researching, coding, documenting., and

juggling other classes, our team successfully built a fully functioning game, closely tailored to Dr. Buckner's standards with relatively few hiccups along the way. The game looks excellent:



It has versatile file options, along with features such as reset capabilities, save catches, and interactive Content pieces that can be rotated, moved, and flashed. It can be won with time and patience, and the user is immediately congratulated when they complete the puzzle:



Overall, the game was a joy to make, and stands as a team accomplishment, especially considering the fact that many members were displaced during the semester because of the current pandemic. We hope others enjoy playing this game as much as we did, and in the meantime, we will be looking forward to creating more GUI's in the future.