Testing Approach

Throughout the course of this phase, we decided to alternate between which group members are developers and testers. Before we started testing, we held a scrum meeting to discuss the verification (whether we are building the correct game) and validation (whether we are building the game correctly) of our design. The meeting allowed us to reflect and look back at some of the flaws in our designs and the overall cohesiveness and coupling of our classes. In order to access the overall quality of our code, we first began by creating unit tests for every function of our design. As a group, we had the goal of writing unit tests that test different features of our code in isolation. In the second phase, we tried our best to implement a modularized, divide and conquer approach to dealing with any concerns we had with our code. Within our program many of the classes were responsible for doing multiple things, overall truly hindering the cohesiveness of the classes, this became evident in the testing process of our code.

Another aspect that our program lacked was information abstraction and using access modifiers. Because of our lack of information hiding, we decided to make most of our fields private and created getters and setters for every field that we need to modify and or access. The lack of access modifiers truly forced us to refactor our code as finding the errors became very difficult. Within this phase, we followed a test-driven design (TDD) where we first attempted to create the tests, fail the test, and refactor the design to pass the tests. Leading up to this phase there were certain aspects of the functionality of our code that did not work and we attempted to fix these prior to testing. We first attempted to test all the getters, setters, and boolean values. We first began writing our tests for the inmate class, where we tested the direction, position, number of keys and whether they collected them all, score, timer, and whether the player reached the main door or not.

For this phase, we were required to only write unit and integration tests, in terms of the testing pyramid we made sure to have more unit tests than integration tests (following the general shape of the pyramid). The integration tests were harder to implement; however, they proved to be more useful in detecting different types of errors with the code and how different things interact with one another. Throughout our testing, we realized the advantages of doing unit tests as they tended to be very easy to implement; however, they were not very real and certain bugs were not possible to find using this approach. With our unit tests, we attempted to test as many different methods as we would. We would try to test different edge cases, out-of-bounds values, and middle values. The implemented getters (for the class fields) helped with getting the value of a particular field after calling a method that changes its value. With our integration tests, we had the goal of testing the important interactions that occur within our system. We tested how different collisions worked with one another, how our UI system would work if we were to select different sorts of buttons, and overall how different

components of our game functioned The important interactions we wanted to test were the different types of collisions that can occur between entities like when the main player collects all the keys and is able to open up the gate or when the player collides with the guard and then the game should end with the user seeing the end game menu.

With this phase, we tried to the best of our abilities to obtain as much coverage as possible with how we implemented this game. This task was difficult because with our implementation of the game, there were obviously certain lines of code that were not testable and certain segments of the classes that we were testing that were not possible for us to test. We did however try to use line coverage as a measure of how well we test the MovingEntity, Inmate, Guard, and Menu (and its subclasses) classes. We strived to achieve a greater than %50 line coverage with the unit test that we wrote. Although this was not always achievable, it was a goal we came up with as a team and decided to stick with.

Throughout the course of the development, we played the game many times to detect different bugs, in other words, manual testing has been extremely beneficial and useful to the development of our game. The next section lays out all the different bugs that we encountered regarding our game.

Finding & Handling Bugs

Most bugs were discovered earlier in the prior phases but the major bugs unveiled in testing included bugs related to resetting variables to default values after every level. Mainly, resetting the score, timer values and the inmate and guard positions after the completion of a level was key to maintaining the game-flow of our project. Another related bug caused the time limit for level 1 to be continually used for level 2 instead of its own time limit, especially after completing level 1, returning to the main menu and then continuing to level 2 from the menu respectively in order. Handling such bugs required us to test if the reset function was being implemented correctly by writing unit tests for it. This helped us pinpoint the problem at hand and fix it with minimal change with respect to the structure of the code itself. Another bug discovered was the flashing door in level 3, indicating that the escape door is open when in fact it was still closed. Debugging this error is under work and should be solved relatively soon. Lastly, the initialization of inmate and guard objects introduced novel error behavior to our game which included failed inmate-guard collision checks, failed guard-wall collision checks and failed inmate-wall collision checks. As a result of this, our inmate and guard could move through barriers and walls and the game would not end when the guard collided with the inmate. The solution was uncovered during a detailed code review of the classes involved in the collision process which brought to light the underlying issue at hand. Essentially, the initialization of our inmate and guard objects was being done incorrectly, owed partly to the

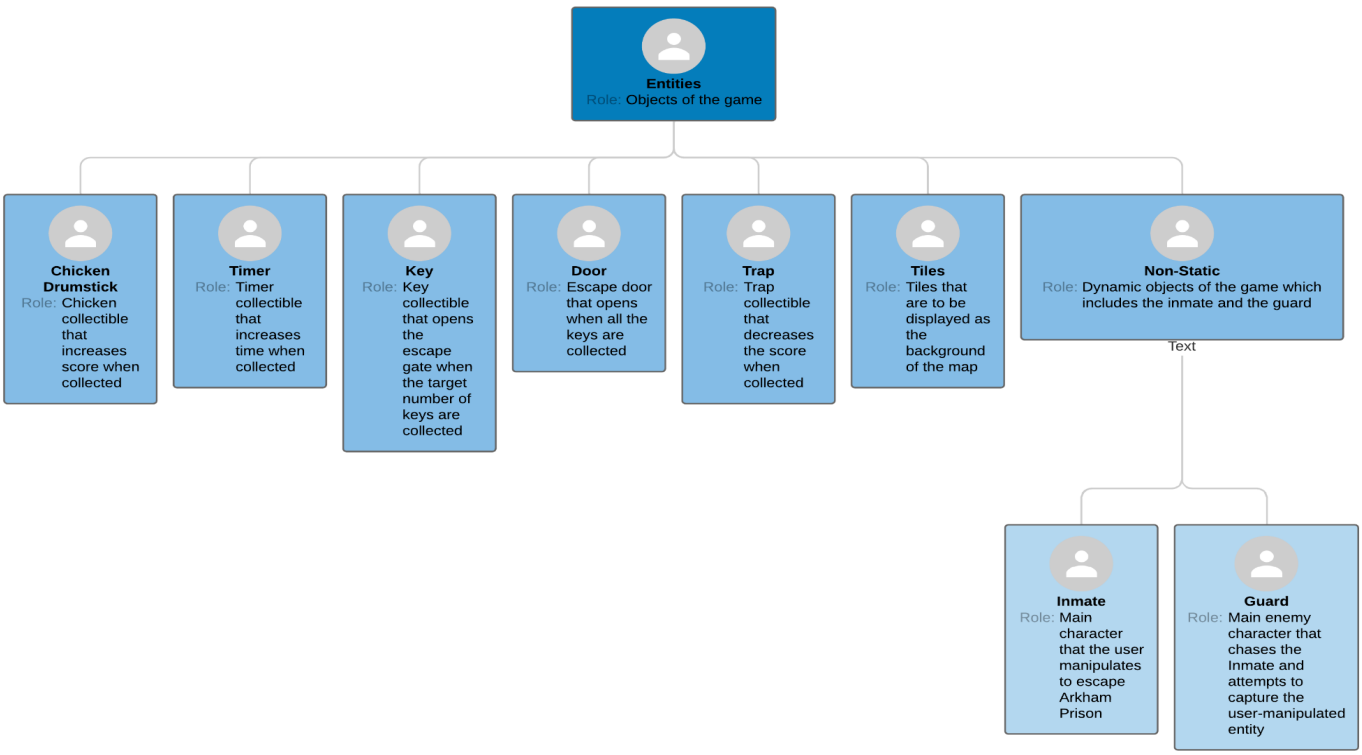
refactoring process. This needed us to initialize our objects within GamePanel using a GamePanel object as opposed to simply initializing them on their own whenever required.

Refactoring Difficulties

The task of refactoring was very difficult to implement because of the poor design we had for our code in phase 2 of the project. Our implementation lacked coherence and truly needed to be redesigned. This was a very costly lesson to learn at this phase in our implementation because of how difficult it is to fix when we are working with such tight deadlines. This process was difficult because of the high coupling we had with our classes (as many of the classes were very dependent on other ones). Along the way, we noticed that we contained spaghetti code that wasn't very Object Oriented. If we needed to do modifications, it would result in a lot of things getting affected and it was worse when we attempted to find bugs with the code.

Modifications and Improvements Approach

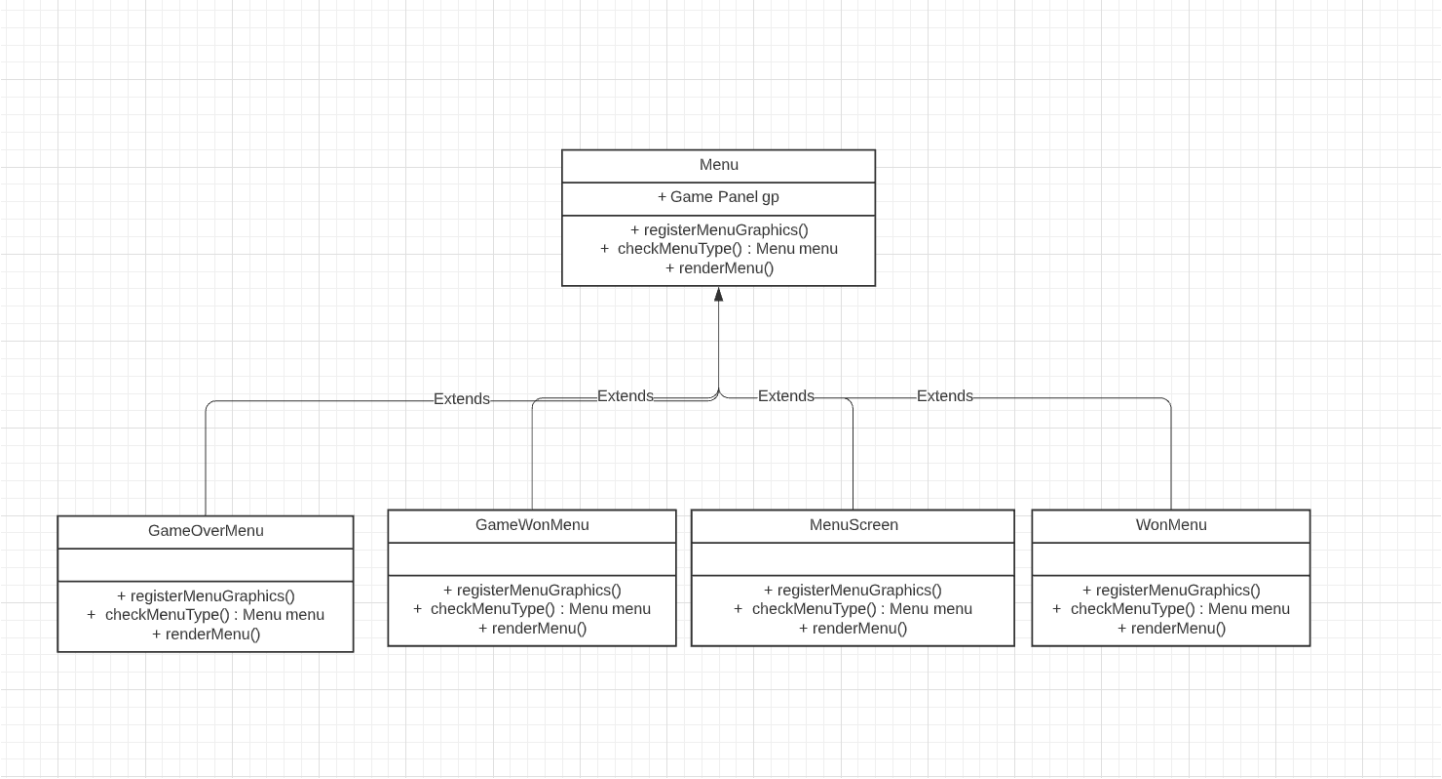
In this phase, we focused on clearly defining the relationships between our classes which represent the various entities of our game. Primarily, we moved code blocks with counterparts in other similar entities to the parent class of both the entities. We implemented this for the inmate and the guard classes. We decided to move the update, draw, and sprite counter methods that were being shared between the two classes to a new superclass called MovingActor. This allowed for our program to have a better overall coherence. This allowed us to remove the long update method shared between the two classes and move them to the superclass. We decided to opt for a different structure that essentially divides the entities of the game into two main types, namely, static entities and non-static or dynamic entities. Our static entities comprise the various stationary collectibles available for the player to collect, on the map. This includes traps, rewards such as the chicken drumstick, timer and keys, and other collectibles such as the escape door and tiles. Next, the non-static or dynamic entities encapsulate the inmate and guard characters. The inmate character is our main user-manipulated entity that moves according to the key inputs of the user, aiming to maximize rewards and ultimately escape. On the other hand, our guard character is the game’s main enemy entity which moves toward the inmate character at all times and attempts to capture and thus end the game.



Additionally, with our tile manager and entity manager classes, we had many different initializations that resulted in long methods. For the tile manager class, we had a register image class that consisted of many different calls to the setup function. The method took three different values (index number, name, and a boolean value). We decided to fix this problem by adding all the values to a text file called tileSetUp.txt which contains all the values/booleans associated with each function call. This allowed the method length to be reduced. Additionally, we did something very similar to this for our EntityManager class where we were containing many createObj() method calls in the setEntityLevel classes. Originally we had three of these classes (setEntityLevel1, setEntityLevel2, setEntityLevel3) where all three of these methods were doing very similar things (creating the objects associated with each of the three maps in our game). Similarly, with what we did with the tileManager class, we decided to move all these significant values needed for the createObj() function call into a text file called EnetityLevel1.txt, EnetityLevel2.txt, and EnetityLevel3.txt. Furthermore, this allowed us to merge all the setEntityLevel functions into 1 more shortened and more cohesive method. As a group we wanted to make our levels look better by adding more objects like different tiles and such and the refactoring process we did for the tileManager and the EntityLevelManger made our job much easier as all we had to do was change the textfiles associated with the new objects and the with each level.

Previously, in our UML design we had a Menu superclass, where all the menu subclasses extended it. We noticed the 4 menu states (GameOverMenu, MenuScreen, GameWonMenu, and PauseMenu) all contain similar methods that both register the graphics of the menus and

render them. We decided to have a sort of interface implementation for the Menu superclass that contains the method for menu registration and rendering and had the menu subclasses override them with their own implementations. Furthermore, this allowed us to clear up the long paintComponent() method in our GamePanel class to a system that takes advantage of the superclass implementation to know what type of menu is being created.



Before none of the fields for the classes were implementing any sort of data abstraction. This was problematic during the testing phase as this made it very difficult to test different methods and interactions and get the values associated with these fields. We ended up changing the access modifiers for most of the fields we have implemented to private for better encapsulation.