[PHASE 3 REPORT - GROUP 25]

◆COSMIC ESCAPE◆

Ananga Bajgai, Annie Boltwood, Mohamed Mustafa, Austin Yu

GAME TESTING
[Milestone 3 - Testing and Quality Assurance]

[1] MILESTONE 3 - TESTING PHASE
The focus of this milestone is the comprehensive testing of the game, which includes writing unit and integration tests, ensuring test coverage and quality, and refining the production code based on the test findings.

For the testing phase, our group hypothesized that most of our functional tests based on the specifications core logic of the game, such as the creation of entities, key collection, and entity interactions, would pass the tests due to their well-defined behaviors. We anticipated potential issues with testing graphical components of the game, such as draw methods inside various classes which need to be drawn on the game panel. We also expected to find bugs that were hidden, as we are in the debugging phase and are uncovering yet unknown problems.

[1.1] Test Development
This section describes the initial steps taken to develop a comprehensive testing system for Cosmic Escape.

- ➢ Our team strategized effectively regarding our approach to the project. We worked on a testing class collaboratively as a starting point to understand the setup for initialization and different types of tests.
- ➢ We then partitioned each section of the code base by its interactions with different classes, and divided the tests to each group member to complete.
- ➢ We began with functional tests to ensure each class behaved as expected. These tests were designed to closely align with the specifications of the game.

[1.2] Testing
This section examines the quality and coverage of the tests conducted. In each class, test coverage, functional tests, and structural tests are discussed, along with our testing process regarding other tests part of the testing pyramid, such as unit testing, integration testing, system testing, and manual testing.

[Game Class]

- ➢ In summary here we tested whether instances of the Game classes were not null, whether the frame object only was created after the “start()” method was created, and tested key presses and whether it was correctly interacting with the game state variables. The Game State variables, signals on which part of the game should be drawn, such as the start screen, game over screen, and play screen. The Game class’s overall main task is to input in keyboard presses, and appropriately make calls to other classes to manage it. So, by checking the boundary cases of game state transitioning (such as from Start Screen to Play Screen, or Game Over Screen to Start Screen) and whether the player is truly inhibited from making movements, we have done an adequate job of testing this class. Also, while testing this class, we refactored the parts that was previously inside the " public void keyPressed(KeyEvent e) " by moving it to an external method, and this change made testing easier, since KeyPressed is inside a "new KeyListener()" object.

- ➢ The game class itself is a class with interactions with most other classes, and so testing the game state changes involves various other classes and so certainly is a significant integration test.

[Game Panel Class]

- ➢ This class's main purpose was to just draw on the screen all the different entities, rewards, barriers and enemies on the screen. The only logic in this class was to do with drawing the objects, dependent on which Game State it's currently in, However, this variability of game states testing is already done as part of testing for the Game Class. Hence, we did not have an individual GamePanelTest Junit class, since the relevant logic is already being tested in other test classes. The bulk of the class is just the PaintComponent method and getters and setters. During this phase, we did however refactor a bit of Game Panel, to improve code quality and readability. Primarily this involved the large blurbs of code inside " paintComponent(Graphics g)" and breaking it up into smaller components.

[Main Character Class]

- ➢ Our testing approach for the Main Character class was initiated by testing the fundamental movement functions according to the specifications of the project, ensuring that the main character could move in all four cardinal directions (left, right, up, and down) as expected. The tests for moveLeft(), moveRight(), moveUp(), and moveDown() were designed to confirm that the character moved correctly after keyboard input from the player. The isMovingEnemyContact(), isPunishmentContact(), isRewardContact(), and isSpaceStationContact() tests ensured proper interaction between the integration of the units by checking the main character's collision with barriers, moving enemies, punishments, rewards, and the space station entities, respectively.
- ➢ The getKeys_tracker() test confirmed the accuracy of the tracking of the keys collected by the main character. We also tested the integration of the score updating mechanism by asserting the correct values of the score in situations such as punishment contact and reward collection. These tests further exemplified our group's extensive structural testing. Our tests achieved 79% line coverage and 70% branch coverage after iteratively implementing structural tests, with the remaining lines and branches consisting of graphics draw and exception cases.

[Space Station Class]

- ➢ In the Space Station class, our testing mainly focused on the key functionalities that are an integral part of the game's process of progression - specifically, the opening of the space station door as the player collects the uniquely coloured keys. The class consists of methods to update the door's open or closed status based on the number of uniquely coloured keys that are collected, open the door once all the unique keys are acquired, and visually open the door accordingly.
- ➢ As per the specifications of the game, our initial approach to testing the Space Station class was to confirm the door opening and closing states based on the unique keys collected requirements. We created tests to ascertain that the space station door remained closed at the start, aligning with

the initial requirements of the game. Subsequently, we examined the updateDoor unit of the Space Station class, which tests the ability to correctly update the status of the space station door based on the number of uniquely coloured keys collected. We tested the number of uniquely coloured keys at which the Space Station door opens, 4, and also conducted boundary testing by testing off values, in values, and out values, such as 3. As part of the unit, the openDoor method uses updateDoor's confirmation to open the door once four uniquely coloured keys have been acquired. This accordingly also covered edge cases such as when openDoor results in false when checking if the coloured keys are equal to four, contributing to the extensive branch coverage.

➢ Considering we are following an iterative testing model for this project, we refactored our codebase following the results of the initial test. The class underwent modularization, specifically in the Bufferedimage loading functionality and the door's state updates. This fortified the unit against resource loading exceptions and improved the code by having one functionality per method. As with other classes, we found that the catch exception branch in SpaceStation wasn't covered by SpaceStationTest, as that would require the installation of a mock test framework to test the exception.

[Reward Class]

➢ In the reward testing class we first began by testing the getting and setting methods inside the Reward class, getting the reward type in the first test, then setting the reward type to "bonus" in the second test. Then in the setScoreAmt() test, we verify the score amount of the reward is able to be modified. Test setActivateTime() and setDeactivatetime() made sure that the activation and deactivation time could be set as intended. The final test resetToRandomCoordinate() tests the boundaries and range that the random values can fall within. We could not use assertEquals() to verify the value of random variables, so instead we chose to test if those random values fell within the intended range.

[Regular Reward Class and Bonus Reward Class]

➢ Since the RegularReward and BonusReward classes were both child classes of the Reward class, there were few methods within them, meaning there were not very many tests to build. RegularRewardTest contains the test getColor() to make sure that the method correctly returns the color. And BonusRewardTest had a test GetDuration() to set and get the duration of the bonus reward.

[Punishment Class]

➢ The first 4 tests in the PunishmentTest class cover the get and set methods for the coordinates inside the Punishment class. Then additionally there is a getType() test to also get the type of punishment since Punishment is a parent class and its children have different types. The test testKeepDealingDamage() sets the keepDealingDamage variable to true, then false, and checks that it was correctly updated. The test testSetDamageStartTime() verifies that the damage timeout start time was set properly. And manageDamageDealingTimeout() makes sure to cover the various cases related to managing the damage dealing time out. Finally, getDamageAmt() to get the correct damage amount.

[EntityFactory Class]

- ➢ The tests involved creating the entities and setting them onto the map object then asserting that both the created entity and entity on map object are the same**.** This verifies that the entity positions are spawned correctly onto the map. Only stationary entities were tested with the map class since only stationary entities would be set on the map.

[Map Class]

- ➢ The tests involved setting an entity (barrier) on the map and getting the entity from the map using the getEntityAt() method. Then asserting that the created entity and entity on the map are the same. This verifies that the setter and getter methods are properly operating.

[Integration Testing]

- ➢ This portion was covered by our testing that occurred for Game class and Main Character class. Both classes interact heavily with other classes, and so, as described above in the Main Character and Game class section, we have appropriately handled these multi-class interactions.

[1.3] Quality Assurance

This section discusses the strategies that were implemented to maintain and guarantee the utmost quality of the code base. It provides examples and reasonings as to how and why different tests were chosen.

- ➢ **Use of Black-Box and White-Box Testing:** When creating our tests, in order to ensure that we could cover all potential and reasonable test scenarios, we began by functional testing of the classes. After, we began using structural testing; looking at the implementation of the code to try and find errors that functional testing could not cover, ie. testing using the developers point of view.
- ➢ **Following Practices:** We made sure to follow the methods learned in class in order to implement the tests up to standard.
- ➢ **Documentation of Test Rationale:** We made sure to have clear explanations in our report about what the test classes did, as well as leaving comments inside the more complicated tests to ensure that the functionality was easy to follow.
- ➢ **Boundary Testing:** Inside of our tests we were strategic in choosing which values to use for our tests so that they could test the boundary limits and range of values. We also included tests to make sure random values fell within the given range/boundary.
- ➢ **Independent Tester:** Since developers understand the system but might test gently, we had a friend test our program to find bugs that we have not found. We taught them about the system and they were driven to find bugs and break the program. This lead us to find a bug in the program where the player's small screen would lead to an uncaught exception occurring, whereby our team adequately addressed this issue and implemented a solution to this problem.

[1.4] Coverage

- ➢ **Figure 2** shows our code coverage. Overall, for most classes our tests had alright line coverage % and branch coverage %, which would have been higher, if not for the challenges we have faced. For line coverage, there were some classes with less than 20 % line coverage, however that is justified by how we weren't able to directly test some of the draw methods the classes had and the classes with low line coverage had the bulk of its lines for the draw portion. (explained in the Challenges section below). Although the line coverage might have been low for some classes, it does not take away from how we have still thoroughly tested import logics in the code, which is necessary before drawing items into the screen.

- ➢ On the other hand, for branch coverage %, there are some percentages that are of the form “100% (0/0)”. This is likely because these classes do not have true branches, and mostly linear logic (with no if/else statements or other similar statements). Excluding the “0/0” case, most of our branch lengths are over 50%, and would have certainly been higher, if we were able to test the draw methods. We have attempted to cover as much as possible of branches, outside of draw methods.

[1.5] Challenges

This section discusses challenges faced during the testing phase.

- ➢ One challenge that occurred during was that we were unable to test the methods that drew the images onto the screen. These methods depend on the Graphics class, which interacts directly with the graphical system of the game. Testing these would require installing and implementing a mock testing framework such as Mockito to simulate the drawing tests. Testing methods responsible for drawing was proven to be difficult because it meant dealing with the graphical user interface directly. The methods depended on visual output and we could not create an automated way to test the successful load of an image and could not confirm if the visual output ‘looked right” after load. Assertions assertTrue(), assertFalse(), and assertEquals() could not compare the actual visual components. Therefore, in terms of the draw methods, we were unable to test how the visual output looked. This led to reduced branch coverage, because often the draw functions contained conditional statements inside. An example of the code that we were unable to test is referenced in **Figure 1**. In **Figure 1**, the draw method calls the graphical operations, for example g.drawImage. This operation directly interacts with the graphics environment. And since this operation is tightly coupled with graphical libraries and hardware, we could not isolate and test the method independently, outside of the graphical context.

[1.6] Findings

- ➢ This section discusses lessons learned from writing and running the tests. Writing the functional tests required a well defined set of specifications and for that we were able to review the specifications provided in phase 1 as well as the phase 2 submissions. For unit testing, it was challenging but worthwhile thinking about ways to test our classes separately since it not only provided us the peace of mind from verifying the code but also a greater understanding of the control flow of each class.

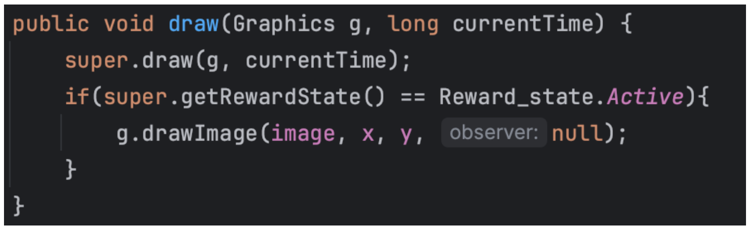
[1.7] Summary

- ➢ Overall, our initial hypothesis that “most of our functional tests which were based on the specifications core logic of the game, such as the creation of entities, key collection, and entity interactions, would pass the tests due to their well-defined behaviors”, was semi-correct. While we also anticipated issues with testing graphical components of the game, such with testing the draw methods, the scale at which it affected our coverage was not expected. At the same time, we note that we have still tested the important logic outside of the draw methods as much as possible, and this important logic is critical for the draw methods to properly work. As we were testing, we also put emphasis on refactoring the code, wherever possible, in order to improve code structure and readability.

[1.8] Appendix

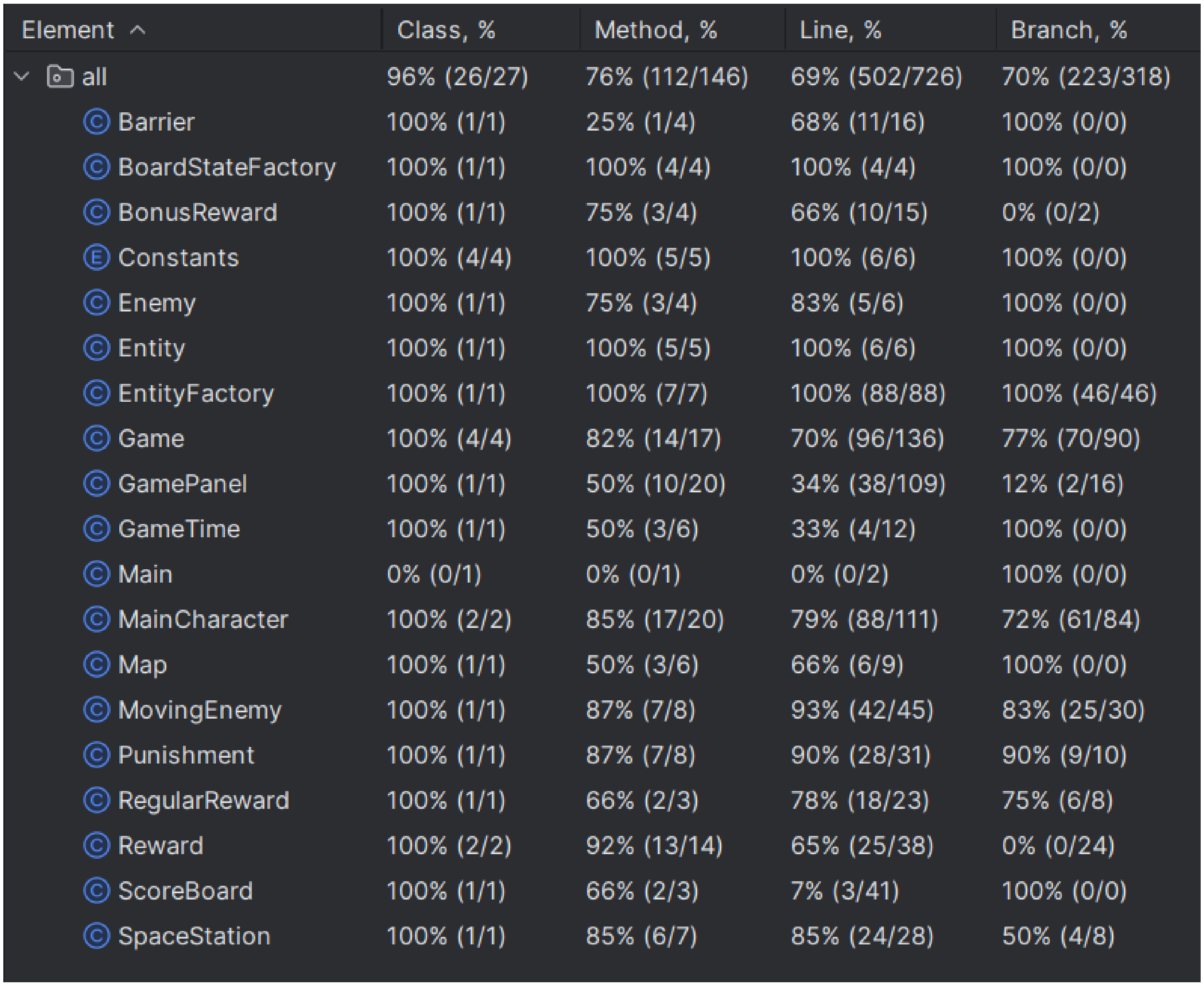
This section contains the coverage reports, samples of code and and test cases referenced in this report

Figure 1



```
public void draw(Graphics g, long currentTime) {
    super.draw(g, currentTime);
    if(super.getRewardState() == Reward_state.Active){
        g.drawImage(image, x, y, observer: null);
    }
}
```

Figure 2



Element	Class, %	Method, %	Line, %	Branch, %
all	96% (26/27)	76% (112/146)	69% (502/726)	70% (223/318)
Barrier	100% (1/1)	25% (1/4)	68% (11/16)	100% (0/0)
BoardStateFactory	100% (1/1)	100% (4/4)	100% (4/4)	100% (0/0)
BonusReward	100% (1/1)	75% (3/4)	66% (10/15)	0% (0/2)
Constants	100% (4/4)	100% (5/5)	100% (6/6)	100% (0/0)
Enemy	100% (1/1)	75% (3/4)	83% (5/6)	100% (0/0)
Entity	100% (1/1)	100% (5/5)	100% (6/6)	100% (0/0)
EntityFactory	100% (1/1)	100% (7/7)	100% (88/88)	100% (46/46)
Game	100% (4/4)	82% (14/17)	70% (96/136)	77% (70/90)
GamePanel	100% (1/1)	50% (10/20)	34% (38/109)	12% (2/16)
GameTime	100% (1/1)	50% (3/6)	33% (4/12)	100% (0/0)
Main	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
MainCharacter	100% (2/2)	85% (17/20)	79% (88/111)	72% (61/84)
Map	100% (1/1)	50% (3/6)	66% (6/9)	100% (0/0)
MovingEnemy	100% (1/1)	87% (7/8)	93% (42/45)	83% (25/30)
Punishment	100% (1/1)	87% (7/8)	90% (28/31)	90% (9/10)
RegularReward	100% (1/1)	66% (2/3)	78% (18/23)	75% (6/8)
Reward	100% (2/2)	92% (13/14)	65% (25/38)	0% (0/24)
ScoreBoard	100% (1/1)	66% (2/3)	7% (3/41)	100% (0/0)
SpaceStation	100% (1/1)	85% (6/7)	85% (24/28)	50% (4/8)

ReadMe File: (In the github repository at CMPT276F23_group25/Cosmic-Escape/README.md)