Phase III Report

1.Project Details:

Project: 2D maze game in Java - Survive in the end

Development team: CMPT 276 - Group16

Team Members:
- Risa Kawagoe,
- Rongsheng Qian,
- Anika Sheikh,
- Sibei Zhou

Development Tools:
- intelliJ IDE with JDK
- GitLab
- Maven
- Junit

Project Phase 3 Deadline: April 4th, 2022

2.Features that needed unit testing

When writing tests, we found a lot of set/get functions for private variables to be the main client for unit testing. Since many of the final features were created as a result of interactions between components, we found more individual methods needed unit testing in the code rather than individual features of the game. These methods include functions to set and get variables, increment timer clock, check collisions between objects, modify characteristics for static and dynamic objects, import image and music for GUI, and class constructors.

3.Important interactions between different components of the system

The project uses the GameFrame class as a platform for all different classes to interact with each other and create the final GUI of the game. Therefore, The most significant interaction in our code is the interaction between GameFrame class and all other classes, specifically the Static and Dynamic classes including their child classes, GameObject class including its child classes, and Music and GameMap class.

The checkCollision is a class to check for collisions of objects with walls. The class contains significant interactions between itself and classes with dynamic objects such as MainCharacter, and Zombie. The class further interacts with the GameFrame class to acquire the game board with layout of the wall location.

Additionally the inputKey class contains important interactions with the GameFrame class. The class is used to obtain user keyboard inputs and determine further actions to perform according to user inputs. The class passes the appropriate actions to take the GameFrame class who then performs them accordingly.

4.Test case/class - feature/interaction correspondence

4.1 FoodTest, GameObjecTest, RewardTest, VaccineTests

Apart from set and get functions, the test cases contained within FoodTest, GameObjecTest, RewardTest, VaccineTests performs tests to check for correct initial positions in the frame for the rewards(food), and vaccines. They further test for correct increment of main character's HP and correct appear and disappear time when colliding with food objects. These tests also cover checking for collisions between reward and vaccine objects with the main character object.

4.2 BadSurvivorTest, KindSurvivorTest, MainCharacterTest, and ZombieTest

Additionally the test cases contained within BadSurvivorTest, KindSurvivorTest, MainCharacterTest, and ZombieTest perform tests to check for correct initial positions in the frame for the static and dynamic character objects. It further tests for correct import of images to display these objects. For the dynamic objects, Zombie and Main character, it covers their movement according to user key inputs, and their collisions with walls. For the static character objects, BadSurvivor and KindSurvivor, it further tests their speak() functions which display a text box, and their appearance and disappearance time.

4.3 MusicTest, inputKeyTest

The MusicTest tests the basic music feature of the game, which is used in the GameFrame class to be controlled in the entire game frame. The inputKeyTest tests the behavior of different game states in the GameFrame class. Depending on the inputs from the user, we check for the gameState variable which controls the entire display of the game.

4.4 GameMapTest, TimerClockTest, GameFrameTest

The GameMapTest primarily contains test cases to cover correct return of the game map layout with appropriate image import for each tile. The class also tests for correct initial positioning of the main character/player. The return of these game boards and initial positioning is dependent on the game level.
The test cases contained in the TimerClockTest covers the correct display of time when the timer initially begins, and when second is increased less than 60 times. It also covers display of minute when second is increased greater than 60 times,
The GameFrameTest contains test cases to cover interactions between classes listed above and between different methods contained within the classes as well as data flow between them. The same is done for the Main class to cover interaction between itself and the GameFrame class.

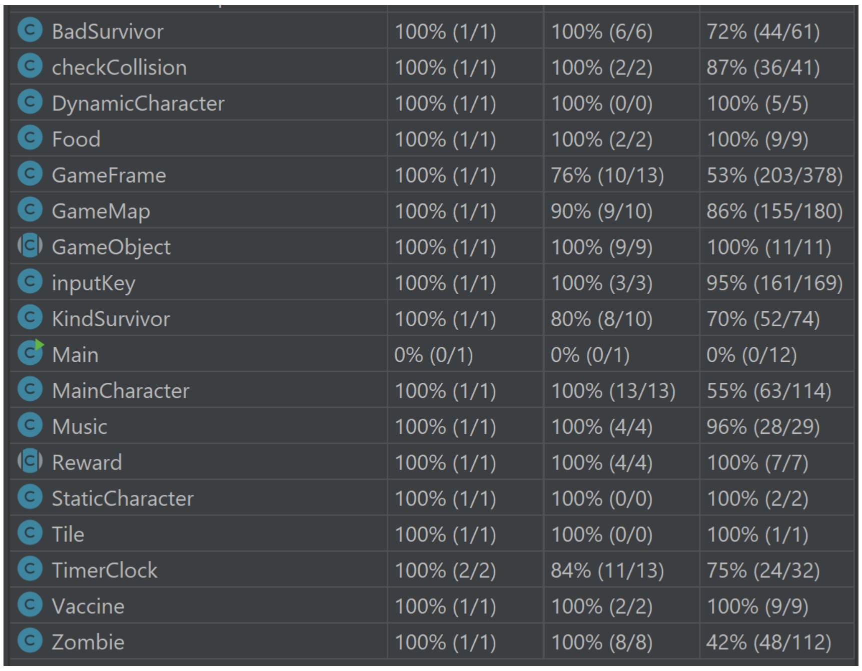
5.Measures taken to ensure the quality of our test cases

Initially we divided the classes between team members such that we write test cases for classes we worked for the majority of. This was done to ensure we completely understand the code we wrote such that we can cover all of its combined conditional and branch test cases. Before developing the test cases, we wrote skeleton code to ensure good communication between team members about tests one writes, and to prevent any lack of code coverage. Once we completed our initial test cases we reviewed other team member’s test cases for an additional pair of eyes. When reviewing each other’s test cases we kept in mind developer bias and tried to implement additional test cases. To achieve this we tried to break the code for which the test cases were written for. This gave us the capability to discover any boundary coverage that we might have missed initially.

6.Line and branch coverage

6.1 Measure

GameobjectTest: line 100%;
RewardTest: line 100%;
VaccineTest: line 100%;
FoodTest: line 100%;
BadSurvivorTest: line 72%;
KindSurvivorTest: line 70%;
MainCharacterTest: line 55%;
ZombieTest: line 42%;
CheckCollisionTest: line 87%
inputKeyTest: line 95%;
TimerclockTest: line 71%;
GameFrameTest: line 51%;
GameMapTest: line 83%;



BadSurvivor	100% (1/1)	100% (6/6)	72% (44/61)
checkCollision	100% (1/1)	100% (2/2)	87% (36/41)
DynamicCharacter	100% (1/1)	100% (0/0)	100% (5/5)
Food	100% (1/1)	100% (2/2)	100% (9/9)
GameFrame	100% (1/1)	76% (10/13)	53% (203/378)
GameMap	100% (1/1)	90% (9/10)	86% (155/180)
GameObject	100% (1/1)	100% (9/9)	100% (11/11)
inputKey	100% (1/1)	100% (3/3)	95% (161/169)
KindSurvivor	100% (1/1)	80% (8/10)	70% (52/74)
Main	0% (0/1)	0% (0/1)	0% (0/12)
MainCharacter	100% (1/1)	100% (13/13)	55% (63/114)
Music	100% (1/1)	100% (4/4)	96% (28/29)
Reward	100% (1/1)	100% (4/4)	100% (7/7)
StaticCharacter	100% (1/1)	100% (0/0)	100% (2/2)
Tile	100% (1/1)	100% (0/0)	100% (1/1)
TimerClock	100% (2/2)	84% (11/13)	75% (24/32)
Vaccine	100% (1/1)	100% (2/2)	100% (9/9)
Zombie	100% (1/1)	100% (8/8)	42% (48/112)

6.2 report - results found from coverage

For GameObject’s test including all Character test and Reward test, there is a draw function that push the picture represent the Object show on the GameFrame, but it is impossible for testing if it is work thought Assert because there are not values to be compared, so we just write a local JFrame and Jpanel in each test and let them show the images of the objects for test. However, there are still so many conditions we were not able to simply change so that the local value fits, since the whole program is running with a Thread, and the value (such as keyboard input, InputKey) changes every time. Because of this, some conditions could not be matched and some lines were ignored (especially in MainCharacter class and Zombie class).

6.3 Discussion - Features or code segments that are not covered and why

Code for main character and object collisions is not covered because it is hard to test the implementation of the collisions between the mc and the object, which means we have to implement the whole game to test this branch.

Image is also a problem. We can test if the character's pictures in different directions display properly, but we can't test which picture will display under which circumstances, because that means we have to add a listener to test.

As per the line coverage Measurements above, it can be seen that GamFrame’s line coverage in our testing is significantly low. This is due to most methods and variables being set to private. We have also deducted methods that use primarily built-in java functions as draw, and clock. This is to ensure our tests are conducted towards code that we developed ourselves.

7. What we have learned from writing and running tests - Changes to the production code

Before developing the tests we did not realize how coupled our code was. This had caused us some time to fix when beginning developing our unit tests. We had to go through our code and discover places where it is possible to make independent code blocks so it is easier to write unit tests on them. For classes and methods we could not decouple, we shifted towards integration tests.

Additionally many of our code blocks were integrated within the same method with built-in Java functions such as draw, start thread, and timer functions. This made it harder to write tests as we wanted to avoid testing the built-in Java functions and keep our tests primarily focused on methods we created. To fix this, we added additional methods to separate our methods from built-in Java methods. We then tested those additional methods to ensure test coverage for the code we produced.

8. Revealing and fixing bugs and improving quality of code

8.1 General

When running our test cases we were able to reveal certain bugs that were caused due to the lack of boundary checks. Majority of these errors contained nullPointer errors. To improve this aspect of our code, we added additional conditions to check for null and boundary cases where applicable.

8.2 Bugs found in resetting the game

Furthermore, we found bugs related to the reset function behaviors of our game board when the game restarts.This includes disappearance of the previously collected vaccines when the player restarts the game. We refactored the resetGame function in the GameFrame class. We cleared the original LinkList of the reward (vaccine and food) and re-added new elements in this function.

Also, in order to distinguish the behavior between restart(restarts the game with the same game level) and return(returns to the title screen thereby resetting all features to the initial state when the game application is opened) in the end screen(after the user has lost/won the game), we added a parameter to notify the function whether to partially reset the game or fully reset the game state. In the resetGame function, all previous information about reward is cleared and a new one is created.

8.3 Background music bug

Additionally when writing tests for the Music class, we found a bug concerning our background music where there were multiple files playing at the same time. The more times we restarted the game, it would keep adding another layer of (the same) background music which led us to notice the bug.
We fixed this bug by rewriting the attributes of the Music class. In the original version, Music class had a variable(clip) of type Clip which could contain only one file. We used this variable to contain the currently playing music (bgm) having javax.sound libraries functions such as start() and stop() control the music state.
However, when it switched to another music file, the clip was copied to the new file leaving the previous file undealt. So we used Clip[] (an array of Clip) instead of a single Clip, and passed the integer parameter as the index of the music file. After that we were able to play and stop the specified music.

8.4 User keyboard inputs

As user-interaction is a crucial part of this game application, we improved the inputKey class by adding extra conditions where invalid command code(commandNum) would be handled as exceptions.

9. Discuss your more important findings in the report.

Since our testing was all done by ourselves who were the developers of this game application, we acknowledge there could have been much more bias than we think. Despite the fact that we checked each other's test code, it does not change the fact that we all knew the make of this program and how it works to a certain degree. There could be missing test cases or considerations that we subconsciously did not notice. In this case, our project is small and bound to the initial members who developed the program, however, we realized how meaningful it would be to have completely separate members for the sole purpose of testing the program.