

Phase 3 Report

CMPT 276

Group 23

Nov. 26, 2022

Introduction

In this phase, we were tasked with identifying and writing unit and integration tests for our game and generating code coverage reports to help us see how much of our code has been tested. With this development, we also got a chance to fix our bugs from Phase 2 and add features that would make playing the game easier. The following sections detail our classification of which methods/classes belonged under unit tests and integration tests, the breakdown of our code coverage report, and the key takeaways.

Unit Tests

Our unit tests were mostly centered around what was happening on our map and the interactions between static and moving entities. The classes composed of unit tests included Entity, Moving Entity and its subclasses (Cat and Mouse), Cheese, Position, and Score.

Integration Tests

Our integration tests were written for all the classes that called our external libraries, mainly, User Input, Game, Game Timer, and Menu. These classes would interact with our resources folder to get the images for our map components and monitor for KeyEvents such as detecting mouse and key clicks to move the player or select an option from our menu.

Certain classes such as Map combined both unit and integration tests. We decided to keep these together and instead, denote which test was a unit or integration test in the comments.

Test Quality and Quantity

The general approach to writing our test cases was to think of all the scenarios that pertained to the object we were testing. This included making sure as many of our branches were covered without having redundancies and checking edge cases through our games states. The following list breaks down what we considered:

- Moving Entity, Cat & Mouse:
 - Moving out of bounds of the map

- Moving in x and y directions or not moving
 - Collisions and the change of game states
 - The collection of rewards/punishments and encounters with barriers
 - Key presses and the effects on movement
 - Win conditions for the Mouse
- Cheese:
 - Generation of object within map bounds
- Game
 - Start, stop, and restart of game
- Map:
 - Addition and removal of characters/objects present on map from their corresponding arrays
 - Spawning/despawning of cheese and movement timer calling on System time
 - Drawing of characters at updated positions
- Menu:
 - Detection of mouse clicks and the change in game states
- User Input
 - Detection of key press with the Java KeyEvent library
 - Checking if pressed key was valid to make the player move or not

Some classes such as Entity, Game Timer, Score, and Position just include getter and setter tests as they were the classes we used to initially help us understand how the JUnit tests ran. We later confirmed that we do not need to further test classes containing constructors, getters, and setters as those methods will be called through our other tests. Overall, our instruction coverage is 99% and branch coverage is 90%. Figure 1 depicts the breakdown of the coverage for each class.

default
















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Menu		90%		44%	9	15	10	86	1	6	0	1
Game		94%		95%	2	20	7	76	1	9	0	1
Mouse		90%		93%	3	15	5	34	2	7	0	1
Map		99%		93%	3	42	2	109	0	18	0	1
Cat		98%		100%	0	24	2	62	0	5	0	1
Cheese		96%		75%	2	6	2	19	0	2	0	1
MouseTrap		84%		n/a	0	1	2	7	0	1	0	1
Crumb		84%		n/a	0	1	2	7	0	1	0	1
UserInput		100%		100%	0	6	0	13	0	2	0	1
GameTimer		100%		n/a	0	4	0	16	0	4	0	1
MovingEntity		100%		100%	0	9	0	11	0	3	0	1
Score		100%		100%	0	6	0	12	0	5	0	1
Position		100%		n/a	0	6	0	11	0	6	0	1
Entity		100%		n/a	0	4	0	9	0	4	0	1
Game.STATE		100%		n/a	0	1	0	5	0	1	0	1
Window		100%		n/a	0	1	0	9	0	1	0	1
StaticEntity		100%		n/a	0	2	0	4	0	2	0	1
Cat.PositionStruct		100%		n/a	0	1	0	1	0	1	0	1
Total	82 of 11,892	99%	17 of 172	90%	19	164	32	491	4	78	0	18

Figure 1: Instruction and branch coverage for each class

For the classes that do not have 100% coverage, we will explain their absence and why we didn't include them in the sections below.

Cat

As part of the constructor for many object classes, we included a try-catch block for the reading of the corresponding image. Since the image was read successfully, the catch portion is ignored as shown in Figure 2.

```
/**
 * Instantiates this cat's position and its image on
 * the game map
 * @param x    Row coordinate on map
 * @param y    Column coordinate on map
 */
public Cat(int x, int y, Map m) {
    super(x, y, m);

    try{
        picture = ImageIO.read(new File("src/main/resources/cat.png"));
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

Figure 2: Catch block not being executed

Cheese

Inside Cheese, our most significant test was the generation of a cheese object at a random spot on the map. As part of the generation, we included a check to make sure the cheese does not appear on a wall, on another static object such as a crumb or a mouse trap, or on the player or a cat. However, since the x and y coordinates are randomly generated, our condition could not be explicitly checked for all possibilities and we are missing 2 out of 6 possible branches.

```
protected Position generatePosition(){
    int maxX = 1450;
    int maxY = 1025;
    int x = 0 ,y = 0;
    boolean posAvail = false;
    while (!posAvail){
        x = (int)(Math.random() * maxX)/25;
        y = (int)(Math.random() * maxY)/25;
        if (map.isWall(x,y) == 0 && map.getItem(getPos()) == null && map.getCharacter(getPos()) == null){
            posAvail = true;
        }
    }
    return new Position(x,y);
}
```

Figure 3: Missing branches in valid cell check

Game

The game class has missing coverage on the main function, where we create the new game object and start the game. Since we can successfully generate the game window and play it without the tests, we assumed that was enough confirmation that our main method works. Other missing coverage would be the stop() in running() since we usually stop the program by exiting.

```
    } while (programRunning);  
    stop();  
}  
  
/**  
 * Main method, start of the game  
 */  
public static void main(String[] args) {  
    Game game = new Game();  
    game.start();  
}
```

Figure 4: main method not called

Menu

In the first version of our game, we had plans to make the buttons on our menu change colour when the user hovered their mouse over them. Currently, our menu buttons do not change colour but regardless, still detect the user's mouse when hovered over and perform the correct action. This missing colour change is reflected in Figure 5.

```
    Font fnt1 = new Font(font, Font.BOLD, 30);  
    g.setFont(fnt1);  
    g.setColor(Color.black);  
    if (pHighlight) {  
        g.setColor(Color.white);  
    }  
    g.drawString("Play", playButton.x + 100, playButton.y + 30);  
  
    Font fnt2 = new Font(font, Font.BOLD, 30);  
    g.setFont(fnt2);  
    if (hHighlight) {  
        g.setColor(Color.RED);  
    }  
    g.drawString("Help", helpButton.x + 100, helpButton.y + 30);  
  
    Font fnt3 = new Font(font, Font.BOLD, 30);  
    g.setFont(fnt3);  
    if (qHighlight) {  
        g.setColor(Color.RED);  
    }  
    g.drawString("Quit", quitButton.x + 100, quitButton.y + 30);  
  
    g.setColor(Color.white);  
    g2d.draw(playButton);  
    g2d.draw(helpButton);  
    g2d.draw(quitButton);  
}
```

Figure 5: Missing conditions for changing the button colour

Figure 6 shows the intended overridden function that would be used to detect the hovering mouse.

```
@Override
public void mouseMoved(MouseEvent e) {

    Point p = e.getPoint();

    // determine if mouse is hovering inside one of the buttons
    pHighlight = playButton.contains(p);
    qHighlight = quitButton.contains(p);
    hHighlight = helpButton.contains(p);

}
```

Figure 6: Overridden mouseMoved function

Mouse

One of our most frequent and significant checks was to see if the player (mouse) has finished the game. Since our finish is a line rather than a cell, we need to iterate through the y values and check if our player is on any one of them. We found it sufficient if we only check for a single y value rather than all of them because the outcome will be the same. Figure 7 shows the condition that is responsible for the finish line check.

```
private void checkFinish() {
    ArrayList<Position> end = map.getEnd();
    if (map.crumbsCollect >= 4) {
        for (int i = 0; i < end.size(); i++){
            if (getPos().getX() == end.get(i).getX() && getPos().getY() == end.get(i).getY()){
                Game.State = Game.STATE.WIN;
            }
        }
    }
}
```

Figure 7: Finish line check

Findings

At the beginning of this phase, we discovered a lot of bugs in our logic that we fixed simply by doing a system test. This required re-writing some of our methods, with the most significant change being an entire rewrite of the Cat movement. Once the most obvious errors were solved, our unit and integration tests allowed us to see how our code reacts to edge cases and we were able to refine it further. We added some features, such as the cats not being allowed to overlap on the same cell and changing our previous finish cell to a finish line, that made the gameplay clearer and easier.

Keeping code quality in mind, we used any spare time we had to clean up unused or repeated functions from the previous phase. In the future, we hope to make our graphics nicer for our final presentation.

Conclusion

Throughout this phase, we learned about having a methodical way of writing test cases to ensure as much of our code and eventually, the gameplay was covered. The biggest takeaway for the team was moving past using only print statements to debug our code and learning how to read code coverage reports. Overall, we enjoyed playing our game and finding the bugs because it allowed us to understand the different types of tests we can write.