Measures Taken to Ensure Quality of Test Cases

To maintain the highest quality in our test cases we aimed to meaningfully invoke all units of functionality, comprehensively cover the maximum number of possible branches that could be taken by changes in control flow, and test the interactions between units of functionality to ensure that their integration resulted in the expected behaviour.

To invoke all units of functionality, we first had to divide the components of our game into independently testable units. We observed that the major objects that make up our maze game are Mouse, Cat, MouseTrap, Cheese, OrganicCheese, LevelOne, Tile and Game; Therefore, we identified the creation of these objects as our ITF's. We then created a test file called ObjectCreationTest.java where we exercised instantiating these objects and invoking all member functions. At the end of each function we placed meaningful assertions that would ensure that the operations performed had been done correctly.

The two functions that were the primary focus of our branch coverage were the MoveMouse function found in the Game.java class and the Chase function found in the Cat.java class. The MoveMouse function houses all of the control flow logic which allows the mouse to traverse the map. It covers cases such as what should be done when the new location of the mouse isEmpty, IsCheese, isOrgCheese, or isTrap. Each of these cases need to be checked when moving in any of the four directions (up,down,left,right). Our approach to covering these cases was implemented in the 'MouseMovementIntegrationTest.java' file where we essentially defined functions for each case and recreated the exact scenarios. Following the enactment of each scenario was a set of assertions which ensured that all objects involved were exhibiting the expected behaviour and that all values stored within the objects were correct. This independent testing of each individual case allowed us eliminate redundancy while attaining a high branch coverage.

After testing our individual units of functionality, we then turned our focus to integration and the state of several objects after an interaction. The primary focus here was the chase function found in the Cat.java file. The purpose of the chase function is to move the cats one cell in the direction that would bring them closest to the mouse. Here we can see that the two classes Mouse and Cat are interacting everytime the player moves the mouse so it is of utmost importance that we test whether the cats are moving correctly and consistently. To do this, we created a suite of integration tests in the CatMovementTest.java file where, similarly to the mouse movement tests, we recreated the possible interactions that could occur and placed each of these interactions into their own test functions. This allowed us to recreate scenarios like "the mouse is above the cat" or "the mouse is to the left of the cat", etc. After each interaction, we applied assertions targeting all attributes of both the mouse and the cat classes to ensure that the resulting behavior is correct.

Reporting on Line and Branch Coverage

To maximize our line and branch coverage during the testing phase, we broke down each of the scenarios that the player could encounter into individual test cases each invoked by one test function. Following this approach we were able to achieve 83% line coverage and 74% branch coverage. Our coverage was not closer to 90% or 100% due to lines and branches associated with UI, as well as some branches in the Cat class's chase function that would not be encountered without adding more barriers in specific arrangements on the map.

Functions like Win/Lose page and images' setter/getter for the objects are not part of the project's requirements, but rather act as additional UI features for the game. Hence, we decided to generate our test cases only toward functionalities of the game itself to ensure each object functions properly.

Additionally, based on the layout of the map, some of the cases within the chase function for the cats are not able to be covered by the tests. For example, if the mouse is at the bottom right direction with respect to the cats, they would check and move to a direction that does not have a barrier, starting from right, bottom, left, and top. However, because the designed layout does not have areas/corners in which the cats' right, bottom, and left directions are barriers, the condition in which the cats move to the top would not be reached by the test.

Learnings from writing and running the tests:

From writing our tests we identified two insights that we will benefit from when working on future projects: the importance of refactoring, and the benefit of regression testing.

The importance of refactoring became apparent to us when we were writing the MoveMouse function in the Game.java class. For our project we followed an incremental approach for development where we would implement one feature and manually test it before moving on to the next. With this approach we first made the mouse move without any checks, then we added a check for barriers, then we added a check for cheese and incrementally added checks for all objects that the mouse could encounter.

At this point we had not learned about refactoring, so with each increment we added five to ten lines of code and this code was repeated for each direction. By the time we had implemented all of the checks for the game, the moveMouse function was close to 500 lines long. Hence, testing this function was difficult since there was duplicated logic. Additionally, this cost us a lot of time to locate all the bugs in the code.

To improve the design of our function, and make it easier to follow, we found all occurrences of duplicated logic and extracted them to new methods. By the time we finished refactoring we had extracted 7 new methods and cut the length of function down to half. The best part about having extracted these methods is the fact that if we find that there is a bug with that functionality, we only need to apply the fix in one place instead of several.

The benefits of regression testing became apparent to us after we had written some of our tests for phase 3. As we were developing and refactoring our code, there would be several occasions where we would add a feature or restructure a function and a previously working feature would “break.” We would then have to try and figure out what aspect of the new feature created that fault by intuition. After writing our tests, we learned that, had we written tests for each feature as we implemented them, we could simply run the tests after we added a feature to see what assertion failed in which specific scenario. This would have saved us a lot of time in debugging and ensured the integrity of our application.

One of the bugs that the testing helped us identify was in the setters of the mouse, and cat classes. Initially, prior to making the jlabel associated with each object an attribute of its class, the setters only updated the x and y coordinates stored for the tilemap. When we set the location of the mouse adjacent to a cell containing a cheese or a mouse trap while writing our tests, we noticed that the jLabels weren’t actually moving to the location we had set. This made us realize that we had missing logic in the setters and promptly fixed this error.

Changes to the production code during the testing phase:

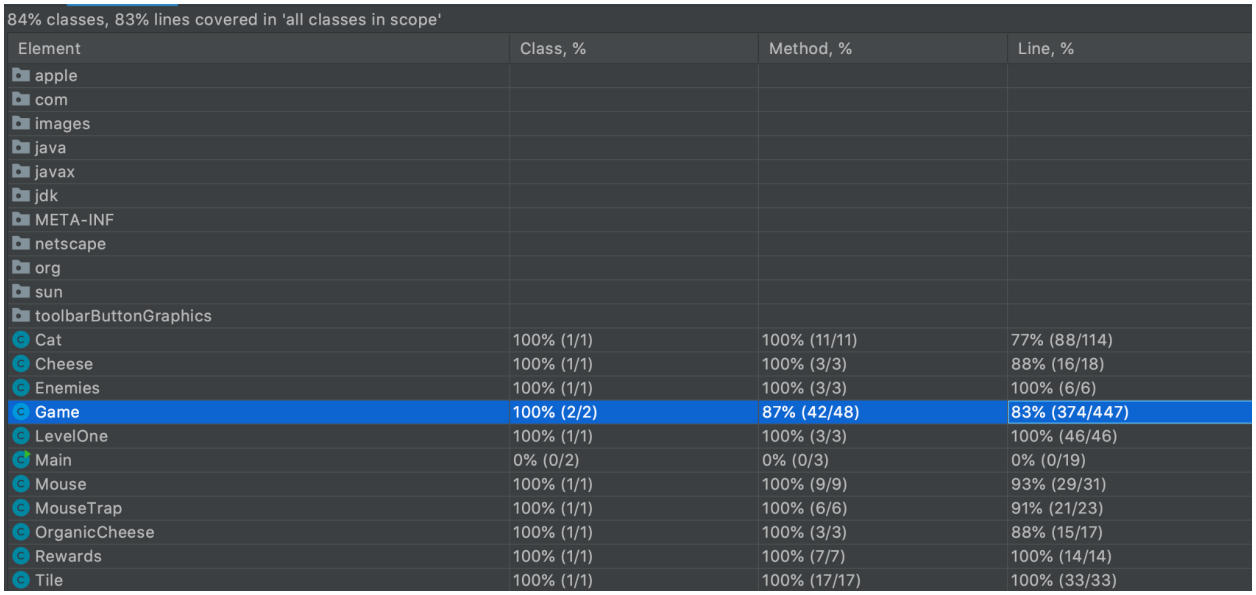
Initially, the Game class was used as the “God Class” to store most of the important functions of other objects within the program, including functions to generate the map’s layout, and to create the objects’ label. During the testing phase, we decided to do some refactoring so we either extracted functionality that should have been its own class or moved feature envy functions and attributes to the classes that they really wanted to be in. Two prime examples of these changes were the Jlabels for all objects and the LevelOne class.

Originally, the jlabels for each object were stored and created in the game class however, this seemed out of place because the jlabel is an integral component of the object so we made the jlabels data members of their respective class and the functions for their creation member functions.

In addition to the jlabels, the creation and storage of the tile map was originally done in the game class which was poor design because it meant any future maps would have to be manually created despite sharing common functionality. To remedy this, we extracted the creation of a tile map to the LevelOne class so that the game class would be less bloated and, in the future, if we wanted to create a new or specialized level, we could simply subclass the LevelOne class.

Screenshots of Coverage

Reporting on line Coverage



84% classes, 83% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
apple			
com			
images			
java			
javax			
jdk			
META-INF			
netscape			
org			
sun			
toolbarButtonGraphics			
Cat	100% (1/1)	100% (11/11)	77% (88/114)
Cheese	100% (1/1)	100% (3/3)	88% (16/18)
Enemies	100% (1/1)	100% (3/3)	100% (6/6)
Game	100% (2/2)	87% (42/48)	83% (374/447)
LevelOne	100% (1/1)	100% (3/3)	100% (46/46)
Main	0% (0/2)	0% (0/3)	0% (0/19)
Mouse	100% (1/1)	100% (9/9)	93% (29/31)
MouseTrap	100% (1/1)	100% (6/6)	91% (21/23)
OrganicCheese	100% (1/1)	100% (3/3)	88% (15/17)
Rewards	100% (1/1)	100% (7/7)	100% (14/14)
Tile	100% (1/1)	100% (17/17)	100% (33/33)

Reporting on Branch Coverage

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		85%		74%	45	141	73	443	6	46	0	1
Cat		82%		67%	16	43	26	114	0	11	0	1
Main		0%		n/a	2	2	16	16	2	2	1	1
Main.new ActionListener() {...}		0%		n/a	2	2	4	4	2	2	1	1
Mouse		97%		n/a	0	9	2	31	0	9	0	1
MouseTrap		96%		n/a	0	6	2	23	0	6	0	1
Cheese		96%		n/a	0	3	2	18	0	3	0	1
OrganicCheese		96%		n/a	0	3	2	17	0	3	0	1
LevelOne		100%		100%	0	10	0	46	0	3	0	1
Tile		100%		n/a	0	17	0	33	0	17	0	1
Rewards		100%		n/a	0	7	0	14	0	7	0	1
Enemies		100%		n/a	0	3	0	6	0	3	0	1
Game.new ActionListener() {...}		100%		100%	0	3	0	4	0	2	0	1
Total	545 of 4,217	87%	67 of 261	74%	65	249	126	767	10	114	2	13