Group-19 Phase 3

Unit Testing

When identifying features in our project to unit test, we looked at the core elements of each class. This would include interactions such as a food object disappearing if its active flag is set to false or a reward being marked collected if its interact method is called. We split these unit tests up further by checking phases of an object's lifetime. First, we confirmed that all variables and properties were correctly set by asserting them in an initial test method. Then, we would test specific interactions within the object as mentioned above with the reward and food examples. Testing destruction of objects isn't necessary for most classes as we wipe everything except the game when calling a new map.

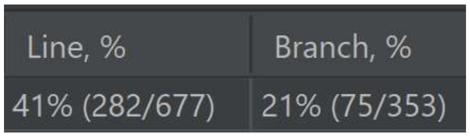
Integration Testing

Since interactions between different objects makes up for a large part of our game, integration tests were very important. They helped ensure our modular design approach didn't take away from overall functionality. These tests included all interactions between two or more classes. Key interactions within our system that we wanted to test include when a player picks up food, she should heal for 3 life and the food should disappear. When a player collides with a punishment, she should take 3 damage and the punishment should disappear after its animation finishes. Additionally, enemies or the player should not be able to interact with non-active rewards or punishments. All of these interactions rely on multiple code segments working together. Thus, we tested interactions by creating multiple instances of each object then called the methods that cause interaction between the objects and asserted the output.

Test Quality and Coverage

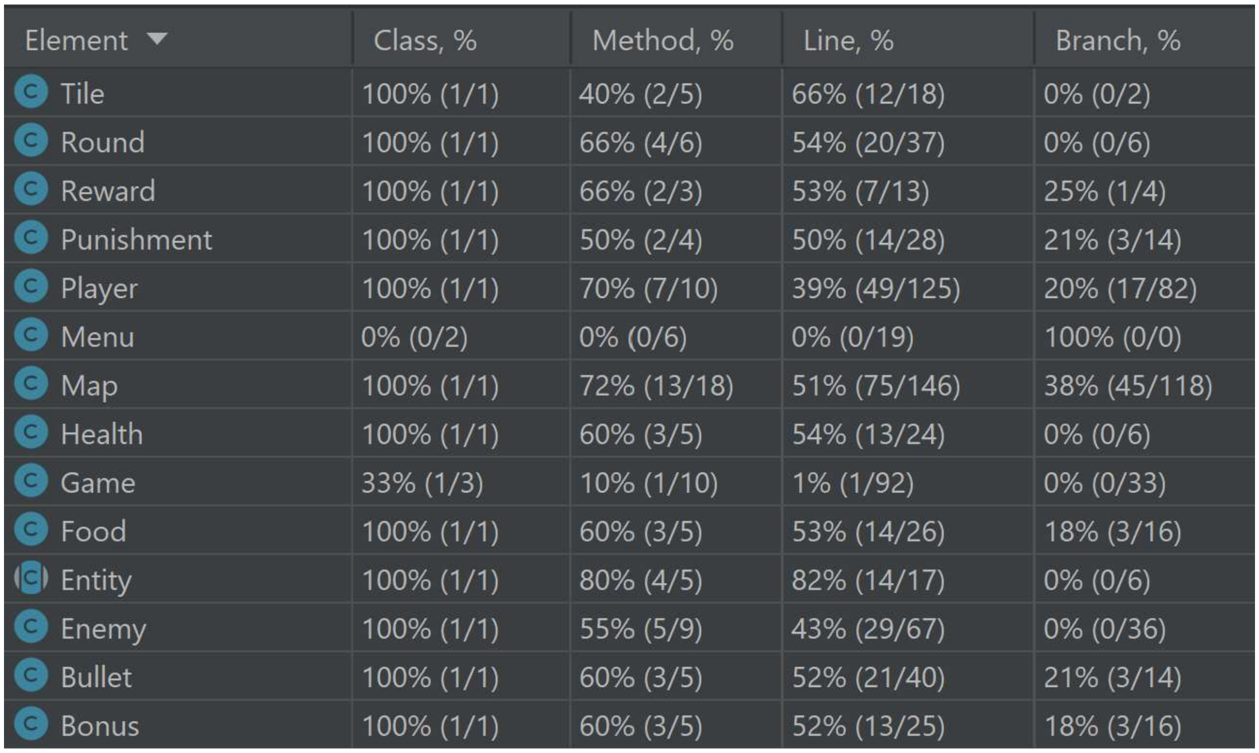
We tried to test as much as we could with unit and integration tests. Unfortunately, much of the code is about things such as animation that is not possible to test in this way. There are also classes such as Entity that cannot be tested directly because it is abstract and can only be tested through the use of the child classes. The exact test coverage numbers are listed below.

Overall coverage:



Line, %	Branch, %
41% (282/677)	21% (75/353)

A breakdown of coverage of each class:



Element	Class, %	Method, %	Line, %	Branch, %
Tile	100% (1/1)	40% (2/5)	66% (12/18)	0% (0/2)
Round	100% (1/1)	66% (4/6)	54% (20/37)	0% (0/6)
Reward	100% (1/1)	66% (2/3)	53% (7/13)	25% (1/4)
Punishment	100% (1/1)	50% (2/4)	50% (14/28)	21% (3/14)
Player	100% (1/1)	70% (7/10)	39% (49/125)	20% (17/82)
Menu	0% (0/2)	0% (0/6)	0% (0/19)	100% (0/0)
Map	100% (1/1)	72% (13/18)	51% (75/146)	38% (45/118)
Health	100% (1/1)	60% (3/5)	54% (13/24)	0% (0/6)
Game	33% (1/3)	10% (1/10)	1% (1/92)	0% (0/33)
Food	100% (1/1)	60% (3/5)	53% (14/26)	18% (3/16)
Entity	100% (1/1)	80% (4/5)	82% (14/17)	0% (0/6)
Enemy	100% (1/1)	55% (5/9)	43% (29/67)	0% (0/36)
Bullet	100% (1/1)	60% (3/5)	52% (21/40)	21% (3/14)
Bonus	100% (1/1)	60% (3/5)	52% (13/25)	18% (3/16)

Findings

We ended up making changes to our production code during this phase as a result of testing. The majority of the changes increased the robustness of our classes. If we were trying to test a certain interaction and needed access to a private variable, we would add a getter method in the class to be able to gain access to it. This resulted in more encapsulation throughout the program.

A problem that we had was that it was very difficult to test everything with JUnit because most of the logic and interactions can not be tested with simple function calls. Much of the system's behaviour can only be observed during run-time. However, this is not what JUnit is used for. On doing more research into automated testing in games, we found that people mostly do things such as automating user input -- something we spoke about in class but unfortunately fell outside the scope of this assignment.

An example of something that is difficult to unit test is pathfinding. Many of the individual functions employed for pathfinding were tested but pathfinding as a whole could not be tested without running the game. This limitation was supplemented by the use of a debug mode. To run debug mode, press the “G” key while the game is running and you will be able to see some of the tests we used during development.