SIMON FRASER UNIVERSITY

ENGAGING THE WORLD

Phase 3 Report

Group 15

CMPT 276

Testing:

Test Class Name(s)	Functionalities to be tested

BunnyTest (integration)	- Bunny setup/creation
	  - setupBunny()
	- Bunny movement and position
	  - testSetBunnyBasic()
	- Bunny death/lose condition
	  - testBunnyDie()

WolfTest (integration)	- Wolf setup/creation
	  - setupWolf()
	- Wolf movement and position
	  - testSetWolfBasic()

Path finding test (for node validation)	- WIP (work in progress)

CarrotTest, DoorTest, MedkitTest, SpoiledCarrotTest (Unit)	- Variable naming
	  - name()
	- Collision checking
	  - collision()
	- Retrieval of appropriate .png
	  - image_load()

CollisionCheckTest (Integration)	- Collisions with rewards (carrot, medkit)
	  - reward_collision()
	- Collisions with enemy and position of enemy (wolf)
	  - wolf_collision()

ControlTest (Unit)	- User keystroke inputs - key_release()

PlaceSetterTest (Unit)	- Initialization of reward position
	  - object_placement()
	- Initialization of enemy position
	  - wolf_placement()

TileTest (Unit)	- Map and wall generation - test_Map_TileLoad() - Testing object identities (checking if the map created is what we think it is)

	- test_read_map()

UITest (Unit)	- Gamepanel generation
	  - setup()
	- Message displays
	  - test_showMessage()
	- Loading UI images
	  - test_imageLoad()

1.Discuss the measures you took for ensuring the quality of your test cases in your report. Code coverage has been traditionally used as a popular test adequacy criterion. Measure, report, and discuss line and branch coverage of your tests. Document and explain the results in your report. Discuss whether there are any features or code segments that are not covered and why.

We decided to emphasize very common interactions with the program that were almost always going to take place if not always (eg. map generation and object generation). While user collisions with objects may not always happen (eg. the player may not always collide with the enemy), they are likely and we wanted to verify that the interaction was working as intended.

Our line and branch coverage was relatively high, naturally, as we focused on simulating the most common interactions which also represented most of our original source code. However, we may have omitted some interactions that are either highly unlikely or impossible (such as the timer running for an extremely long time) to replicate in our program, as testing that specific behaviour would not help us much in validating our program's intended interactions. In addition, we could have (unknowingly) missed some unlikely edge cases in our testing as testing every single possible condition may have been impractical in the interest of both efficiency and time. We also decided against some tests that we believed to be redundant for testing functionality.

2.Briefly discuss what you have learned from writing and running your tests. Did you make any changes to the production code during the testing phase? Were you able to reveal and fix any bugs and/or improve the quality of your code in general? Discuss your more important findings in the report.

- We learned the difference between unit tests and integration tests
- We learned how to write Maven test cases using JUnit.

- We enhance the quality of the enemy class's code by adding in more features and fixing some problems.
- We learned to prioritize testing in a way that would emulate how the user would interact with the program realistically.
- We tried to moderate the amount of tests in order to reduce redundancy in our test code and to also make the test code look cleaner.