

Phase 3 Report

Discuss which test case/class covers which feature/interaction in your report:

ControllerIT.java covers the interactions between the Controller and Entity classes. These interactions include entity initiation, movement, and collision handling.

- **Entity initiation:** Inside Controller, entities are added to entity maps randomly or with hardcoded locations. The test methods `controllerInitEnemyTest()` and `controllerInitRegTest()` assert that these hardcoded locations match the retrieved locations of initiated enemies and regular rewards. Meanwhile, `controllerInitItemTest()` asserts that bonus rewards and punishments lie within the valid board size.
- **Main Character Movement:** `controllerMoveCharacterTest()` initiates a main character and assigns it various movements, checking that their new point matches the expected point in the board.
- **Entity collision handling:** There are two main collision-checking situations: when we want to check if a main character can collect an item or is going to be killed by enemies, or when we want to spawn entities in unique places. For the former, we want to remove the item or enemy and change game score. For the latter, we do not want to remove the overlapping entity. So our collision handling methods must only remove when necessary. Our test method `controllerCheckCollisions()` uses four test cases to represent all possible mixes of yes/no collision and yes/no removal.
- **Controller reset:** For reasons given in phase 2, Controller uses the Singleton design pattern. Setting up a new game requires us to empty all maps within the Controller and spawn new entities, so we chose to "reset" the Controller by calling its private constructor. Testing this required us to call Controller, change the main character's position, reset, and assert that the main character's position had also been reset.

BoardIT.java & CellTest.java covers the generation of the game's board from the board.txt file.

- **Cell creation:** Cells can be either ground or wall cells, so the test instantiates a wall and ground cell. We assert that the wall cell contains the wall sprite and it is solid so the entities cannot be on it. We also check that the ground cell contains the ground sprite and that it is not solid.
- **Board loading:** Instantiates the board from the board.txt file. It checks every cell on the board if it's solid or not. If it's solid, it should have an ID of 1, which indicates that it's a wall cell. It also checks if the sprite of the cell is the wall sprite. If the cell checked is not solid, the ID must be 0 and the sprite is the ground sprite.

RegularRewardsTest.java, BonusRewardsTest.java & PunishmentTest.java covers the items in the game.

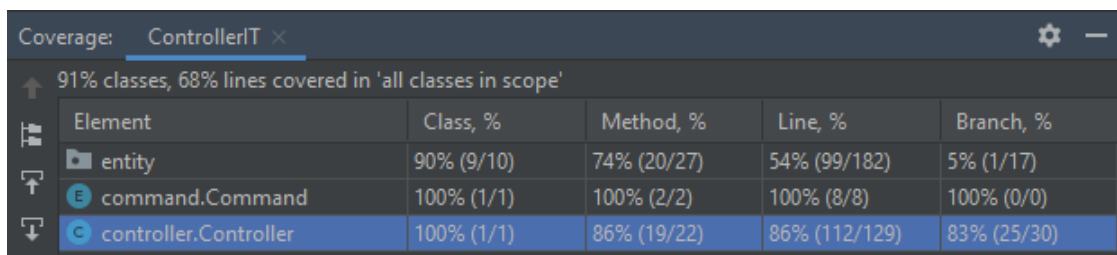
- **Constructor:** All items are constructed by taking a point object. We tested it by asserting the position, bound and collected parameters matched the expected result.
- **setPosition:** Items can be set to a new position. We tested it by asserting the position and bound parameters matched the expected result after the method is called.
- **countDown:** Bonus rewards disappear after a certain amount of ticks. We tested it by asserting the collected parameter is true after the amount of ticks for it to disappear occurs.

Phase 3 Report

Discuss the measures you took for ensuring the quality of your test cases in your report.

ControllerIT: First, I narrowed down which methods should be tested by excluding private, static, or simply untestable methods (ones that could not be tested automatically). For each eligible method, I carefully examined its purpose, parameters, and return value. I refactored methods that seemed to have too many purposes and made some methods private if they were never called externally. This process resulted in an even narrower set of methods, and a set of specifications for each method. From these, and inspired by the lecture slides in class, I chose test cases that covered each unique category of purpose and corresponding parameters/return values. For example, `controllerCheckCollisions()` uses four test cases to cover all collision check and handling scenarios (as described above). This helped ensure that all public, bug-prone code was automatically tested with realistic scenarios, but not in a purely ad-hoc manner.

Code coverage for ControllerIT is as follows:



| Element | Class, % | Method, % | Line, % | Branch, % |
|-----------------------|------------|-------------|---------------|-------------|
| entity | 90% (9/10) | 74% (20/27) | 54% (99/182) | 5% (1/17) |
| command.Command | 100% (1/1) | 100% (2/2) | 100% (8/8) | 100% (0/0) |
| controller.Controller | 100% (1/1) | 86% (19/22) | 86% (112/129) | 83% (25/30) |

As you can tell, a solid chunk of Controller and Entity lines were covered. Scrolling through the Controller class in IntelliJ after running ControllerIT with coverage, I was also able to visually confirm that all intended Controller-Entity interaction methods had been called. As we learned in class, line coverage is not the best indicator of test quality because if all your code is put on a single line, coverage would be 100%. Line coverage does not take into account decisions. Nevertheless, since our code is not all clumped together on the same line and extraneous statements have been minimized, this high level of line coverage is a helpful indicator of basic test quality.


Branch coverage is a better indicator. As expected, very few Entity branches were covered, which is fine because those branches do not fall within the purpose of this Controller-Entity integration test. Meanwhile, a solid majority of Controller branches were covered (the minority were from less important methods or methods that could not be easily tested automatically). This gave us confidence that important Controller responsibilities like collision handling were being properly handled in various scenarios.

Phase 3 Report



CellTest & BoardIT:

To verify that the correct sprites and cell IDs were being used, getter methods had to be implemented in the Cell class as those variables are protected. For the cell, the only method unable to be tested is the render() method since it is visual.

Code coverage of Cell unit test:

| 100% classes, 84% lines covered in 'all classes in scope' | | | |
|--|------------|-----------|-------------|
| Element | Class, % | Method, % | Line, % |
|  cells.Cell | 100% (1/1) | 83% (5/6) | 84% (11/13) |

Code coverage of Cell while running from Main:

| 97% classes, 95% lines covered in 'all classes in scope' | | | |
|---|------------|------------|-------------|
| Element | Class, % | Method, % | Line, % |
|  board | 100% (1/1) | 100% (5/5) | 96% (28/29) |
|  cells | 100% (3/3) | 77% (7/9) | 88% (16/18) |


Because getter methods were implemented to assist in the cell unit testing, the code coverage for main was decreased, but it was necessary to keep those variables protected rather than making them public.

The integration test using BoardIT had less code coverage because of the render method again, and one method to make Rectangles for bound checking walls. Though, those two methods aren't necessary to test if the cells loaded from board.txt were correct.

RegularRewardTest, BonusRewardTest & PunishmentTest:

The constructors are tested by asserting the parameters are as expected after the constructors are called. Setters and getters of a parameter are tested by one corresponding test by calling the setter first and then asserting the return value of the getter is as expected.

Code coverage:

| 100% classes, 91% lines covered in 'all classes in scope' | | | |
|---|------------|------------|-------------|
| Element | Class, % | Method, % | Line, % |
|  entity.Item | 100% (4/4) | 90% (9/10) | 91% (32/35) |

As shown, the lines and methods that weren't covered by the test is the render() method as it is visual.

Phase 3 Report

Discuss whether there are any features or code segments that are not covered and why.

- GUI could not be tested automatically in a convenient way. To confirm that the correct images have been rendered, colours are showing up right, etc. involves visual confirmation.
- User input was not covered because our user input method was through the keyboard, and this could not be automated easily.
- The Game and GameState classes were not tested because their methods were largely private for the sake of encapsulation. For example, one of Game's main responsibilities is to manage state switching, which requires user input and cannot be tested automatically to prevent states from being ended inappropriately. Likewise, a lot of the GameState functionality is to manage private game information and to call Controller methods.
- The Character Entities classes were not tested with unit tests as their move method depends on the checkWall method of the controller. This is a demonstration of how high coupling hurt testability of codes.
- For the board, the player and enemies haven't been tested to ensure they can't go on walls because the user input can't be automated.

Findings: Briefly discuss what you have learned from writing and running your tests.

As said above, the process of designing and implementing the tests helped us refactor our code. By pinpointing each method's purposes, parameters, and return value, we could observe which methods could be simplified, split into two, made private, moved, or removed. For example, at the end of Phase 2, we had a complicated collision checking method inside Controller which directly checked all collisions. It was long, repetitive, and used GameState code. Examining this method to help design the test revealed ways to break this method down. In Phase 3, we split this method into two parts, with one part in GameState and one part in Controller. This allowed us to reduce coupling by removing GameState from Controller's code. It also allowed us to remove other methods that did the same thing, but in a slightly different scenario. Looking at the code coverage helped identify methods that were unused or could be moved to more relevant directories. For instance, in the Board and SoundLoader classes, they each had the same method to format paths. After seeing that was the case, the path editing method was moved to the Utils class, where it should've been in.

While we were trying to implement unit tests for the MainCharacter class and the Enemy class, we discovered that as the character movements depend on the checkWall method in the Controller class. Hence unit tests for these two classes weren't implemented. Which is fine for this case as they were tested by the controllerIT test. But classes with lower coupling would have been more favorable..

Running the tests also contributed to finding and fixing bugs. We made lots of changes in phase 3, like changing access modifiers, deleting or rearranging methods, changing board size, and randomizing punishment spawns. Running the tests after making those changes helped us check that the updates did not negatively impact our app. One interesting moment was when ControllerIT failed. We had just randomized punishment spawning, but the test

Phase 3 Report

thought that punishments were still supposed to spawn at hardcoded points. This test failure was actually a helpful confirmation that our randomization update was successful and that punishments no longer spawned at those exact hardcoded points.