Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Abstract

The purpose of our work during this phase was to extensively test the functionality essential to the internal mechanisms of the game - primarily, types like Board and Entity, the tick system, and changes to the game state from in-game events. Our testing was done using the JUnit framework, running through individual IDEs, as well as through Maven.

Unit Test Focuses

We had numerous different classes / features we needed to extensively unit test, which we will outline in the following list (from most important to least important):

Board System

The Board system is the underlying core of the game. It is what manages the placement, movement, spawning/despawning, etc. of *all* entities that exist in the game world. It was of utmost importance for us to achieve full branch and line coverage for the Board class, since everything from the Player, to a Reward, to the exit Gate was routed through its functionality.

Entity System

Entities were of second-highest importance. They are the game objects that the player actually cares about, therefore once it was confirmed that they were properly represented on the board, we unit tested their spawning/despawning, collisions, etc.

- **Player:** The player is probably the most important Entity in the game. After making sure all the base methods were working properly, we subsequently unit and integration tested movement, the collection of rewards, being touched by enemies, touching punishments, etc.

Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

- **Enemy:** The enemy class is what provides the core challenge of the game, and also encapsulates the most complex behaviour. Our unit tests on Enemy mainly focus around ensuring that their pathfinding works correctly, and there are no issues with them colliding, obstructing each-other, etc.
- **Rewards & Punishments:** Rewards and punishments are next up, being the next most important aspects of the game as they can directly change the state of the game and dictate its flow. Ensuring punishments properly deducted score and/or ended the game, rewards added score and opened the gates, and similar were the focus of these tests.
- **Gate:** Making sure the exit gates open properly was something we needed to make sure worked correctly (if a player went through the whole challenging ordeal of traversing the maze and collecting all the rewards, a bug that made it all worthless would be devastating).

Pathfinding

Last but certainly not least was pathfinding. Now that we were sure Entities spawned in reachable locations, and enemies could successfully reach the player, we devoted a final few tests to ensuring the common Pathfinding algorithm they used was fully functional. This mostly involved testing edge cases like the start position being encased, there being no open spaces, etc.

Integration Test Focuses

There were a few important things to test interactions between, including:

Game-Events and Game-State

The state of the game (i.e., the game being over, active, etc.) is dependent on what happens to the player during gameplay (collecting rewards, hitting punishments, etc.). Making sure the game state properly reflected this was important.

Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Entities and AI

Using a decoupled class for pathfinding means it is important to test the interaction between the abstracted algorithm and the intended use-case. We mostly used brute force to test if it was working properly.

Game-Variables, Game-State and UI

Making sure variables such as time, score, and rewards remaining were all properly updated and displayed in the UI, and ensuring the correct screen was shown at the right time was the next most important interaction to test.

Test Classes

Below is a list of the test classes we implemented:

BoardTest

- **Purpose:** Encapsulates the logic for testing the placement, removal, relocation, and interaction of Entities, and the management of game ticks. As this logic is the same across all Entities, this allowed us to effectively cover a surprising amount of common functionality.
- **Coverage:** 100% Branch / Line coverage of Board.java, including:
  - Board Constructor, GetBoardDimensions, WithinBounds
  - SpaceAvailable, GetEntitiesAt, IsEmpty, PlaceEntityAt, PlaceEntity
  - RemoveEntity, RemoveAllEntitiesAt, MoveEntity, RandomEmptySpot
  - RandomReachableEmptySpot, Tick

Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

EntityTest

- **Purpose:** Encapsulates remaining logic for Entity testing not done by BoardTest.
- **Coverage:** 100% Branch / Line coverage of Entity.java (when combined with the unit tests found in *BoardTest)*, including:
  - Get/SetCoordinates, Get/SetIsBlocking, SpawnEntity, RemoveEntity
  - OnEntityTouch, OnEntityWasTouched, OnTick

PathfindingTest

- **Purpose:** Contains integration tests between pathfinding, procedural generation, and enemy AI. Mainly relies on using brute-force random generation to run through procedural test cases.
- **Coverage:** 100% Branch / Line coverage of Enemy.java, and AI.java, including:
  - Enemy.OnEntityTouch, Enemy.OnTick, AI.Pathfind

LogicIntegrationTest

- **Purpose:** Primarily integration tests of interactions between changes to the Player (e.g., changes in score, being touched by enemies), and the state of the game.
- **Coverage:** 100% Branch / Line coverage of Punishment.java, Reward.java, RewardBonus.java, including:
  - Constructors, Punishment/Reward.OnEntityWasTouched

LiveTests

- **Purpose:** Consists of integration tests for everything that has to do with simulating runtime (i.e UI, game.java, etc)
- **Coverage:** ~% Branch / Line coverage. Our library, libGDX, made this section of our testing suite unable to function properly. Commented-out code and a more technical explanation of the issue is kept in this file for reference.

Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Test Quality

To ensure a high level of test quality, we:

- Clearly partitioned our test classes to make sure tests were in the correct place
- Bundled common test functionality in BoardedTestCore.java to avoid duplicate code
- Included frequent assertions and monitored states throughout testing
- Provided comments which described a tests' purpose, important details, etc.

Coverage

For our most important classes, we obtained near 100% branch and line coverage, as was shown before.

From our analysis, it seems that the code that has been fully or almost fully covered is:

- Board.java, Player.java, Entity.java
- Gate.java, Obstacle.java, Punishment.java, Reward.java,
- RewardBonus.java, Punishment.java, Enemy.java, AI.java

Code that only received partial coverage is:

- Game.java, due to it being used mainly at runtime (making it harder to fully test)
- Board/MazeBuilder.java, as it was mostly tested through repeated iterations in integration tests.

And code that received nearly no coverage is:

- Input/RenderHandler, as these require hooks to an application and would always fail to work properly during unit tests
- *Screen.java & UITable.java, as they also require hooks to an application - and cause stalls/crashes during the unit/integration tests we attempted.
- BoardRenderer, for the exact same reason as above
- Leaderboard.java, due to being a non-essential feature, and there being bigger priorities.

Phase 3 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Findings

From our tests, we found that there were few to no errors, with the majority of failed tests resulting from improper implementation of the tests themselves. There were some NullReferenceExceptions that we fixed, but other than that, the system seemed to handle repeated, thorough testing as well as could be expected.

We did, however, decouple the procedural generation functionality from Game.java, moving it into its own separate file called BoardBuilder.java. This was done not only to improve the quality of the code (by avoiding duplicate generation code in our test suite), but also to make it easier to test our code and ensure our map tests accurately reflect in-game generation if things are ever changed around.

We also found (a bit late, unfortunately) that our primary library (libGDX) makes it particularly difficult to perform integration tests. If we were to do it over again, we would have accounted for this in our design by making things like runtime Game logic and UI more decoupled from the library in order for them to be properly tested.