Testing Plan
Donut Group

By Hellen Shi z5360603,

Christian Politis z5388072,

Anthony Chan z5361215,

Nicholas Zhou z5362402,

Danacha Lark z5210093

Contents

Buildable Entities

Buildable entities are all tested for the exception that may arise for where there is insufficient materials or invalid item is specified to be built. Their durability is integration tested with battle. Where after x battles, the x durability is completely depleted and removed from inventory.

Bow

Bow is tested to ensure that its single recipe works as expected.

Sceptre

All four variations of its recipe (excluding sunstone replacement of treasure/key) are tested. The use testing involved checking that all mercenaries were in their ‘bribed’ state when mind controlled. We also tested those mercenaries and assassin went to hostile after the duration of sceptre usage ran out.

Midnight Armour

The behaviour of midnight armour not being buildable when there is a zombie toast, despite having sufficient material is tested. The testing for the battle effects involved a dungeon where the player will get killed without the armour but will win with the armour, with the tests checking both cases.

Shield

Shield has testing to ensure both recipes work as expected. And that if materials are available for both recipes, the recipe with treasure is prioritised (an assumption that was made).

Collectible Entities

Collectible entities are all integration tested with player movement to see whether it can be picked up, appears in inventory, and removed from the map. And where applicable, if they are used in some way, they are removed from inventory. Also, they are tested for exceptions for when invalid item is used, or the item is not in inventory when used.

Key

Unit testing for keys should revolve around the collecting of such objects. Such as picking them up from places around the map. However, keys do interact and are used in many other aspects of the project such as doors and crafting. These interactions should be tested by their respective tests rather than relying on key to perform these tests.

On a system level the interaction between keys and relying on objects should be tested (such as crafting) and entering through doors.

Since key is a simple collectable and is a requirement for many other aspects, key should be early in the development pipeline to allow for more complex aspects to use it.

Invincibility + Invincibility Potion

Potions are unit tested to ensure that their duration and queueing functions as expected, and this is done by accessing the current portion of the queue inside the player. In integration testing, the effect of each potion is observed in battles and movement of mercenaries and zombies. Lastly, to make sure it works on the front end, a usability test is done to see if the movement is truly affected by consumption of potion.

Treasure + Wood + Arrow

Wood, treasure, and arrow are used in integration testing to see if they are removed from inventory when used in building items. The case of where there is a surplus of materials available to build an item is tested to ensure the correct number of materials is removed from inventory. In the case of treasure, it is also tested to see if it is removed from inventory when used in bribing.

Bomb

As bomb is a collectible entity, the first thing that had to be tested was whether it can be picked up and removed from the map, used, and removed from inventory. Then it is integration tested with switches and boulders to ensure that it can explode, and the range of the explosion. Further cases such as activating a switch later, deactivating a switch, inactive switch and switch out of range is tested so that bomb works appropriately in all cases. Finally, a usability test is conducted for bomb, switches, and boulders on the frontend to ensure entities are visibly removed from the frontend when bomb explodes.

Sword

Bomb being a collectible entity, we need to assess the following: can a player pick up the weapon and store it in the inventory which can be used for later, is it removed from that map, can it be used within battle? These all fall under system testing being reliant on several external sources to complete overall functionality. Internal testing is not required for this specific kind of entity as it serves no sole purpose at the end.

Sun Stone

Most of the sunstone unit tests can be derived from the tests from keys. As a collectable entity the collection of such objects should be tested. Moreover, as an object both used in opening doors and crafting items, the ability of sun stone to do these should be tested. As an extra requirement in the case of unlocking doors and crafting, the sun stone should always be retained after use. Since sunstone is an extension of key and treasure, the test for the extra functionality mostly falls upon sunstone. Sunstone must also count towards the treasure goal.

On a system level the above interactions should be tested in conjunction, that is opening a door, then crafting an item, all while checking that sun stone is never consumed.

Moving Entities

Moving entities are the other entities on the dungeon that can move by itself. It does not require player input however some of the movements may depend on it. Some of these entities will also move in a different behaviour depending on the player's potion and positions. For all of these entities, player can be in combat with them. The battle behaviour would be tested in Battles.

Zombie Toast

Unit testing for Zombie Toast is how it moves around. It should move randomly by default and move away from player under certain conditions while not being able to walk on walls, boulders, closed door etc. We would track the position the Zombie Toast is moving to determine which movement state it is currently assigned to.

System testing for Zombie Toast is how it is affected by external conditions. The requirement states that when a player consumes an invincible potion, the Zombie Toast's movement behaviour would change. To test this, we would have player consume a potion and check that the Zombie Toast is moving away from the player for the duration of the potion.

Spider

Unit test us to test spawning a spider we need to randomly choose one of the four spider spawns as stated in the spec. When the spawn location is implemented, it will then spawn a spider on top of the spawn location which then proceeds to complete its movement requirements. For us to test this, we will need to loop a certain amount of movement ticks until the tick rate equals the spawn rate, we then can see if the spider does in fact spawn or not. For the system level we need to make sure that it does not spawn on a boulder as a result it will not move.

Assassin

Integration tests were used to test the multiple different features for assassin. Most of the testing involves similar testing as mercenary's behaviours such as moving towards the player and following the player when bribed. For the bribe testing, we had three test to test the randomness of the bribe. Since we cannot guarantee a result of the bribed behaviour through movement, we would individually test the edge cases first where assassin can be bribed, and assassin cannot be bribed. We would alter the value of assassin_bribe_fail_rate to either 0 or 1. We would then test that it can differ as we would have an assassin_bribe_fail_rate that is not 0 and 1 and check that it can and cannot be bribed at certain times.

Hydra

Hydra's testing and behaviour, in terms of movement, is exactly the same as zombie. We would use the same test as zombies to test hydra's behaviours.
In battle hydra has a chance of regenerating health, and while doing so they won't take damage. This will be tested in battle where the seed is set to a known number to ensure health increase behaviour can be predicted. Then for a small selection of rounds, the test needs to assert that if the random number is below the rate, the enemy health delta is positive.

Advanced Movement

Advanced movement has a section for implementing Dijkstra's algorithm to find the shortest path to player. We would test this with assassin and mercenary as both of these entities are required to reach player. Dungeons were made to create a scenario where there are multiple paths to the player, but the entity has to go through the shortest path there is. To reduce the runtime of Dijkstra, there is a limitation to how far the mercenary or assassin could be relative to the player. This was implemented to restrict the number of computations that was required to compute during the auto test.

Mercenary

Unit testing for mercenary involves the default movement pattern for mercenary. It should be moving towards the player at all times. We also have to consider that there may be no path towards player i.e., player is surrounded by walls or mercenary is surrounded by walls. Each of these movements can be tested by tracking the position of the entity relative to the player's position or previous position.

System testing for mercenary is more complicate. Mercenary exhibits plentiful behaviour changes and interaction with other entities in the dungeon. For example, mercenary can be bribed by player for a certain sum of treasure within a certain range. We would not test this here as it is more on the player's end since it is a player's interaction. Mercenary can also switch movement behaviours depending on external conditions.

The switching of movement behaviours can further depend on whether mercenary is bribed or not. To test all these behaviours and movement changes, we would have to test all the possible different movement states mercenary could undertake in different scenarios such as bribed and invisible potion, not bribed and invisible potion, etc. Mercenary can interact with portal similar to a player, thus we have to test that the mercenary would interact the portal alike a player would give certain scenarios regarding portal's teleportation conditions.

Static Entities

Swamp Tile

Swamps was the other section to advanced movement. Here, we had to test the how each individual entity interact with swamp. For example, we had to test how zombie, assassin, mercenary, and spider interact with swamp. We need to consider how swamp will affect advanced movement of mercenary and assassin. Thus, there is a test that would ensure assassin and mercenary will move around swamp if swamp's movementFactor causes the seemingly shorter path to be longer. For each test, we test that the entity is stuck of the same duration as the movementFactor. Here, the movementFactor is also limited to reduce how long they are stuck for to reduce further runtime in the testing as movementFactor could be in the billions which would cause the testing to take extremely long.

Wire

Wire itself does not specifically require too much testing. The unit testing involved is simply seeing if a switch will trigger a wire when a boulder is on top of the switch. The system testing that is involved with the project is whether a series of wires can trigger a certain sequence that can satisfy a logic type either AND, OR XOR, COAND.

Switch Door

Switch doors inherit most of its properties from normal doors. To test switch doors its main properties from doors should be tested first. This includes opening the switch door using normal keys, and that switch doors remain open. New tests involve activating switch doors using different kinds of logic. That is testing that or, and xor and co_and logic work as described. The logic should be tested using both wires and switches as they both carry signals.

Further if a door is open using logic it will not consume a key, moreover the door will only remain open as long as it receives a signal or is opened by a key. System tests involve different kinds of keys (sunstone and key) and logic. It should test that a switch door will remain open if unlocked by a key even if a signal pulse was passed through.

This will be created after logic has been implemented.

Light Bulb

The majority of testing for lightbulb involved testing being triggered under the various logic gates as well as checking for an untriggered state without a valid activated logic. These were accomplished by setting up wires and lightbulbs such that they will trigger when a player pushes a boulder on top of the switch

Wall

Wall is integration tested with movement of player, moving entities and boulders to ensure there is collision where applicable.

Boulder

Boulder is integration tested with player movement to ensure it can be pushed in the direction the player is going. And tested in cases where there is a wall or another boulder, it is stopped.

Floor Switch

Floor switch is integration tested with boulders and bombs to see whether it is triggered as expected when there is a boulder on top. And deactivated when the boulder is removed.

Door

Unit tests involving doors should centre around the state the door is in, that is, whether the door is locked or unlocked. In the state where the door is locked, the player should be prevented from going through the door if the corresponding key is not in their inventory.

Furthermore, the transition from the unlocked state to the locked state should be tested, that is when the player has the correct key and is attempting to go through the door. Finally, the unlocked state should be tested as well, where the door should remain unlocked after passing through. For system testing the above unit cases should be tested in a similar way. This component will be developed after key has been implemented, and no components depend on this. Thus, this should be coded as an individual component.

Portal

Unit tests involving portals revolve around the cases where the exit portal is fully or partially blocked, as well as the position of the player when they go through the portal.
When the player goes through the portal which is not blocked by solid impassable objects (in any cardinally adjacent square of the exit) then the teleport will fail, and the player should take the position of the entrance portal.
This is the same when all cardinally adjacent squares are blocked, and the exit portal is fully inaccessible.

Zombie Toast Spawner

The unit tests for the Zombie Toast Spawner will spawn a Zombie Toast on an open adjacent position. T This will need to make the number of ticks required to spawn the entity. The spawner can check all the open positions as a result it will choose an open position for the Zombie Toast to spawn on. For the System testing it will be reliant on other entities blocking the way to spawn an entity as well as the specific tick rate. Another thing for system testing is when a player destroys the spawner, they must be cardinally adjacent to the spawner with a weapon.

To assess both of these we will need the player to move in a certain direction until enough ticks occur to cause the Zombie Toast to spawn or they are adjacent to the player with a weapon. This can call an exception, if need be, for weapon or no weapon.

Time Travel

Time travel involves many complex cases where the new and old player will interact with the current dungeon, as well as new player and old player interacting with each other.
The first concept to test is to test the rejection of illegal ticks, that is any tick other than 1, 5 or 30, with ticks 1 and 5 coming from the frontend, while ticks 30 coming from the time travelling portal.

Next the rewinding of the dungeon to its state to the requested number of ticks back needs to be tested, that is all collected entities collected between the previous dungeon and now will be back on the map, while all predictable moving entities will be on the same tile (with the same movement).
Further older_player should be spawned during the time travel at the position the player was in at the previous dungeon.

Older player also must be tested so that it follows the route the player performed during the original run, older player must also be able to collect collectable entities and interact with elements on the map. As according to the spec older player must perform all inputs the original player has performed and disappear thereafter.

When time travelling the inventory should be tested to remain and retain all previous pickups. Now when the player is in a time travelled state the player should be able to interact with all entities as usual, however if they were to encounter their older self a battle will ensue. However, the battle will not happen if the player is carrying a sun stone, midnight armour, or is invisible.

In a system test, the player should perform a variety of actions, such as picking up items, destroying spawners, then time travel to check that the older player also performs these actions. Battles should be tested between the new player and the entities in the dungeon, that is for predictable entities and older player. This will be coded before time travelling portal and is of moderate priority.

Time Turner

Time turner as a time travel device relies on time travel. Most of the tests (in the aspect of time travel) are handled by the core logic tests. Thus, for time turner, unit tests should revolve around the collection of time turner and its ability to rewind the game. So, time turner should be tested to check if it can be picked up, and that it will allow the user to rewind time. System tests involve collecting the time turner and achieving a successful rewind.

Time Travelling Portal

We have previously tested edge cases and certain properties of time travel when testing time travel. To further extend the testing to time travelling portal, we need to test the 30 tick rewind. Test will be required to check that it can successfully time travel back to 30 ticks ago upon coming in contact with the portal. We can check that it was successful by checking that certain features on the dungeon that were picked up still exist, or we can check the positioning of old player where it should be equivalent to the player’s position 30 ticks ago. We can also test that the portal does not constantly teleport the player when player is sitting on the same position of the portal. Another integration testing would involve the old player disappearing upon contact of the time travelling portal the player took.

The teleportation will only succeed in the case where the exit portal is fully open, in which the player will take the square in the direction the player entered the portal in.

For system testing all above cases should be tested with further tests for more advanced situations where the player exits a portal onto another portal. In this case the player should travel through the next portal until they reach an empty/passable space.
It is also required that mercenaries should teleport. This aspect should be handled by mercenaries and system testing is given by them.
Portal being mostly self-contained (and more advanced) is to be implemented in the middle of the development cycle where basic movements have been handled and is stable. Since mercenaries also teleport, portals need to be finished before the full completion of the mercenary.

Battles

A Lot of the testing when it comes to Battles are system level due to the fact that multiple entities and functions are required to take part when a battle state occurs. The system level testing involves checking the stats on weapon damage, durability, player damage, player health, enemy entity damage, enemy entity health and how many rounds there are. When player’s attack they lower the enemy's health and vice versa. When a weapon has been used it will lower the durability until it becomes at a certain point unusable/destroyed. By the end of the battle either the enemy dies or the player dies which can trigger one of the goals implemented.

Goals + Exit

All goal tests are system level because it assumes multiple functionalities work as expected and incorporates them.

Simple Goals

For simple goals, we will test that for each of the goal strategies (exit, enemies, boulders, and treasure) are read in correctly from the json, by testing if the string does contain the objective, e.g., contains exit. And then carry out the steps to complete the goal and test the goal string is empty and assert that the steps are carried out correctly. For the enemies' goal, we had two tests, one for when there is a spawner, which meant that the test needs to check that killing enemies without destroying spawner does not complete the goal. And the other test where there isn’t a spawner.

Complex Goals

We will test that the complex goals are read in correctly. Then test the two Boolean operators ‘AND’ and ‘OR’. And for ‘OR’ we need to make sure completing either of the goals will result in a win. With complex goals it is not possible to achieve a goal, such as pushing a boulder off a switch or moving off the exit. So, we tested not achieving goal works as unexpected.

Dungeon Builder

Tests will need to ensure that player and exit location are created correctly according to specified start and end coordinates, and that the goal is exit. Tests will also need to ensure that there is an enclosing wall around the maze and there is a path from player starting location to the exit. Further testing can consist of randomised start and end coordinates to ensure dungeon generation works for a wide variety (positives and negatives) of coordinates.

Persistence

All persistence tests consisted of creating a new game, doing something in the game, saving the game, making another new game, and then trying to load the saved game. Simple persistence tests ensured that type, id, and position of all entities are preserved in the loaded game. And also, other aspects such as battle response and inventory. Complex persistence tests ensured that bribed entities remain bribed after loading and potion effects remain and will last for the expected remaining duration after loading.