Software Testing

Megan Miles
Alistair Foggin
Kajol Dodia
James Wild
Isselmou Boye
Matt Fitzpatrick

Part A

When creating a new system, it is vital to thoroughly test the system as a whole to ensure that there are no faults on release and that the system successfully achieves the requirements. One method of testing would not be enough since it would not give a good enough picture of the finished system. As a result, there have been a few testing methods completed: JUnit testing, white box testing and black box testing.

Before jumping right in, it is important to ensure that tests exist for each part of the system. This required a traceability matrix where we kept track of all the tests that were created and the corresponding requirements that the test fulfils. Before submitting the project, we ensure that each requirement has been tested by one method or the other. This ensures that we get the most coverage that we possibly can, such that we can find any faults in the code.

Unit testing is the most appropriate for this project since it evaluates whether a function has its desired result. This type of testing is known as a narrow test since it only focuses on a small part of the code and not at the bigger picture. However, it is important to ensure that all parts (no matter how small) do what they are supposed to. To start off testing, JUnit tests were created to test the code from assessment 1. Therefore, in the event where we continue coding and the system fails, it will be easier to identify the fault and debug. After completing the unit tests for the previous code, the method in CI new feature, new test. Thus, the code will always have appropriate testing available for when any failures occur. Additionally, there would be less time at the end of the project needed to be dedicated to software testing.

However, as you can see there are some requirements and classes and methods that cannot be tested using JUnit. To amend this, White box and Black box were used. Although both of these testing methods are manual and can take time, they both have their benefits. Since the project is a game, it is important to consider the experience that the user will have whilst playing. Black box testing is perfect for testing the playability of a system and the best way to test against the requirements. White box testing is included in the testing plan, as there are methods and classes that are not covered in the JUnit tests and therefore there needs to be an additional method of testing to ensure that there is more coverage.

Since testing is a large amount of work and time needed, 4 team members were assigned to the JUnit Tests to ensure that risk R4 isn't being broken and 1 member was assigned to the manual tests. Each of the testing units that needed to be generated were added to the GitHub Backlog on the repository and each team member working on the software testing were assigned tests for completion. It is important for each member to update the backlog for when tests are in progress, in review and completed, such that all members are on the same

page. Once these are complete for the assessment 1 features, for any additional features that are added, the member of the team who is responsible for the feature must include the tests (if applicable). This was the general rule that was followed, which is further described in the Continuous Integration Report.

Whilst implementation and JUnit Tests are being completed, the testing scripts for the manual tests can be generated, ready to actually complete the tests when the implementation is complete. Each of the testing scripts includes the following: Identifier, short description, related requirement, author, steps to be followed (including input data) and expected outcome. Once the implementation is complete, another table is generated which includes: identifier, tester,, actual outcome and status (pass / fail). Splitting up the scripts and the actual results ensures that the member working on the manual test can spread the workload evenly.

Part B

The AssetTest tests whether all images, fonts or maps that are used in the code exist in the asset folder, this includes images for the tutorial page, different customers and cooks whether they are holding an item or crate. This is more of a test to ensure that the presentation of the game is correct. This test passed successfully.

The following tests are to check the components of the game and are to ensure their correctness. This has a high coverage as it is very easily testable and no obstructions preventing it. Both the CookingComponentTest and CustomerComponentTest check to ensure that all default values of the corresponding components are correct. The FoodComponentTest tests whether the ID and Food Types are as expected. The three tests in this package all passed with a 100% pass rate.

The tests below are used to ensure that the component systems are operating as expected. To ensure the cook moves in the correct direction, the CarryItemsSystem test checks that when the rotation of an item is changed the position of the transform matches with the corresponding vector for all directions. The CustomerAISystemTest tests that the customers are appearing correctly from frame to frame as well as ensuring that the reputation of the player decreases as expected. It also checks that customers are destroyed as expected and that the cook interacts with the customer correctly. As well as this it tests the steering behaviour of the ai component and that orders are fulfilled properly by ensuring the food it receives is the same as the food it asked for. The InventoryUpdateSystemTests checks that the array of a cook stores the correct ingredients in the correct order and that the cooks behave independently so their inventory does not overlap. PhysicsSystemTest tests that when a force is applied to a body its position is changed accordingly in the x and y for positive and negative values. The PlayerControlSystemTest tests that the chef chosen by the player has the player component and has the correct settings so it can pick up and put down items or move around the map and the other cooks are not being controlled unless the player changes cook. PowerUpSystemTest is used to test that when the player has purchased a power up that the corresponding attribute has been adjusted correctly, for example once the power up for chopping speed has been bought it should have increased by 1.2. We use the StationSystemTest to check that cooks interact with the stations correctly, meaning that when an item is placed on a station it is no longer in the cook stack and they can only place compatible ingredients on the station. It also tests that when a station is full it can no longer accept any additional ingredients and stations that require processing time, such as the grill station, are doing so and the result of these processes are correct, for example a patty turning into a grilled patty. Some of the classes in this package struggle with coverage for example the LightingSystem and RenderingSystem as OpenGL does not work well with unit testing. All tests in the package above successfully passed their tests with 100%.

This test is for the user input of the game and to ensure it is operating correctly and reflecting what the user is instructing. The KeyboardInputTest checks that when a key is pressed or not pressed then the corresponding variable should reflect that using a boolean value, for example when the ‘e’ key is pressed then the ‘changeCooks’ variable should be true. This test passed successfully.

These next tests are to check the box2d utilities of our game are operating correctly. Box2DLocationTest is used to test that vectors and angles that are expressed in radians are translated between each other correctly and that when we create a new location it is in the right position. Box2dRadiusProximityTest creates two physics bodies and checks to see whether the owner proximity should accept the other body when it is within range. In the Box2dSquareAABBProximityTest, the testFindNeighbors functions create two different physics bodies, one of them belonging to a steering body which has the proximity. The proximity is used to detect if other bodies are within range using either a rectangle for the AABB proximity or a circle for the radius proximity. The position of the bodies are set to define whether they are in range and should be detected or not. If they should be, then the findNeighbors function returns 1 corresponding to the number of bodies it sees. Box2dSteeringBodyTest is used to check that the vectors and angles in radians can be translated correctly for a steering body and ensures that the setZeroLinearSpeedThreshold throws an error for when the threshold is below what the linear speed can be considered as zero. The CollisionCategoryTest tests that the boundaries, entities and lights provide the correct corresponding value so then it can tell the physics engine which objects to collide with, for example a light will collide with different obstacles than a cook or customer. The LightBuilderTest checks that point or room lights are created correctly so they are in the right position, have the correct colour, if the distance it lights is correct and if they are soft or have x-ray turned on. We use the MapBodyBuilderTest to test that when we load the map the correct number of objects are detected and that objects like rectangles, circles and polygons get the correct parameters and instantiate them properly. WorldContactListenerTest is used to ensure that if a station and cook physics object are touching, then that is identified to allow interaction between the objects. This also applies between cooks and customers. This package struggles with coverage especially in the Box2dSteeringBody, LightBuilder and WorldContactListener. However all of the parts that could be tested were done so with a 100% pass rate.

The following tests are used for the saving part of our game to ensure correctness and the user can safely use the save function that was a requirement. The GameStateTest is used to check that when loading a saved game the everything is correctly retrieved from the engine. We use the SavableCookTest to test that the cook has the correct coordinates and any items in their inventory should be the same when a saved game has been loaded. SavableCustomerAISystemTest is used to ensure when loading a saved game there is the correct number of customers and groups and that any powerup attributes are correct. The SavableCustomerTest is used to test that after loading a saved game the customers have the same food order and coordinates as before. We use the SavableFoodTest to check whether the food components have the same transform and are actually the same component after loading a saved game. The SavablePowerUpSystemTest is used to ensure that there are the correct power ups applied when loading a game. We use the SavableStationTest to check whether the powerups on the stations are correct and if they have the correct ingredients on them. The SavableTimerTest is used to test if the elapsed time is the same as it was when the previous game was saved and then reloaded. 100% of the tests in the package above successfully passed.

These tests check the utility functions of our game and ensure correctness while operating the game. We use the EntityFactoryTest to test whether all entities including cooks, food, stations and customers are created properly. For all of these we first check that there is only one entity with the correct component and is the same as what is in the engine. For the cook, the stations and the customer we then see if it has the correct fixture and the component is added as the user data. For the food we make sure it is the food type we instantiated it as, we then test that we can load all 13 food textures. The FoodStackTest is used to check that the food item is added on the stack of the cook correctly , by checking that the stack is the correct size, that the food item is actually in the stack and has the right transform. A similar process is done to check that the food is pushed and popped on the stack correctly and that the isHidden attribute is updated. It also tests the visibility so that the isHidden attribute is changing in the manner that is expected as the stack is changing. We use the GdxTimerTest to check that the timer is incrementing correctly and resets, stops and starts when required.

MapLoaderTest uses a simplified version of the map to see if all objects, including three ai objectives with different keys, are built correctly as well as all checking all the stations required exist. The StationTest tests that ingredients or a combination of ingredients are mapped to the correct next step for example formed patties are mapped to grilled patties or when the correct ingredients are in the stack it is mapped to a salad. It also checks that each station is mapped to the correct value and vice versa. We use the WalkAnimatorTest to test that when given a rotation it is converted to the correct direction, for example when given a rotation of 90 it will give the up direction. The YComparatorTest and the ZComparatorTest both work in similar ways and both test that when the same transform is applied to two entities they have equal coordinates and then the opposite if they are given two different transforms. Certain classes of this package are unable to be covered such as WordlTilemapRenderer as unit testing does not work well with rendering systems. The tests in this package all have a 100% pass rate.

All of the tests that were listed above are automated tests using JUnit 4 that can be found in our code and have a 100% pass rate for all Operating Systems that we tested on which include the latest versions of Windows, macOS and ubuntu. These tests are all carried out after every push or pull request that is completed on the main branch due to the continuous integration we implemented and are analysed each time to ensure that our tests are passing successfully. These tests were shared between Alistair Foggin, Matt Fitzpatrick, James Wild and Isselmou Boye and can be seen who has written each one on our GitHub.

Below you can find the coverage report from the automated tests we have carried out:

Package	Class, %	Method, %	Line, %
All Packages	60.2% (68/113)	53.3% (223/418)	53.6% (1348/2517)
com.devcharles.piazzapanic	5.9% (1/17)	2% (1/50)	7.8% (23/293)
com.devcharles.piazzapanic.components	100% (15/15)	95.8% (23/24)	91.3% (95/104)
com.devcharles.piazzapanic.componentsystems	76.9% (10/13)	74.7% (56/75)	69.3% (427/616)
com.devcharles.piazzapanic.input	100% (1/1)	40% (4/10)	57.1% (56/98)
com.devcharles.piazzapanic.scene2d	0% (0/20)	0% (0/66)	0% (0/463)
com.devcharles.piazzapanic.utility	80% (24/30)	77.5% (62/80)	81.6% (417/511)
com.devcharles.piazzapanic.utility.box2d	100% (9/9)	66.2% (51/77)	65.9% (170/258)
com.devcharles.piazzapanic.utility.saving	100% (8/8)	72.2% (26/36)	92% (160/174)

As mentioned earlier, there are certain parts of the code that cannot be automatically tested due to OpenGL not working well with unit testing which is the reason why some of the packages above such as scene2d and box2d have a lower coverage. These include the rendering system, lighting systems, base game screen, endless game screen and scenario game screen. Although, these systems are entirely visual so can be tested through black box testing rather than using JUnit.

PART C:

Coverage report link : https://eng1-32.github.io/assessment2/htmlReport/index.html
Manual test cases: https://eng1-32.github.io/assessment2/docs/ManualTestingScripts.pdf
Test results: https://github.com/eng1-32/piazza-panic-2/actions/runs/4863340485/jobs/8670933864