Testing

Group 12 - T12

Assessment 2 updated version

Members:
Ruslan Allahverdiyev (ra1354)
Billy Brudenell (bb1085)
Harry Erskine (hde501)
Usman Khan (muk509)
Adi Laskowski (akl532)
Ben Remmer (br894)
Ollie Stoole (os878)

4.a)

To make generating tests easier and ensure all requirements are met, we will use requirements based testing. This is because the people developing the test won't necessarily need to know how all parts of the game are integrated in order to write tests. As stated by IBM [1], analysing requirements to develop tests lets us learn information about intended system behaviours for a given scenario, performance expectation, which parts of the system or system features might need testing also when new system requirement are added test can be generated quickly for them, reducing reworks errors and defects in are game and testing procedures. Additionally using this method will allow us to test non-functional requirements which might be hard to test otherwise.

Another method we will incorporate is unit testing for lower level methods in the game. This involves the use of tools like Junit to execute tests and compare the actual results with expected results. We run the game in a headless environment using the gdx-testing environment. Testing methods like saving the game and if assets can be loaded. This will make for faster and more efficient testing as automatic testing can execute a large number of tests much faster and more efficiently than manual testing, which saves time and reduces costs. Additional benefits include consistent and repeatable testing, which reduces the likelihood of errors and improves the reliability of the testing process. It will also allow for early detection of defects this will also reduce the cost and time required to fix defects in the future.

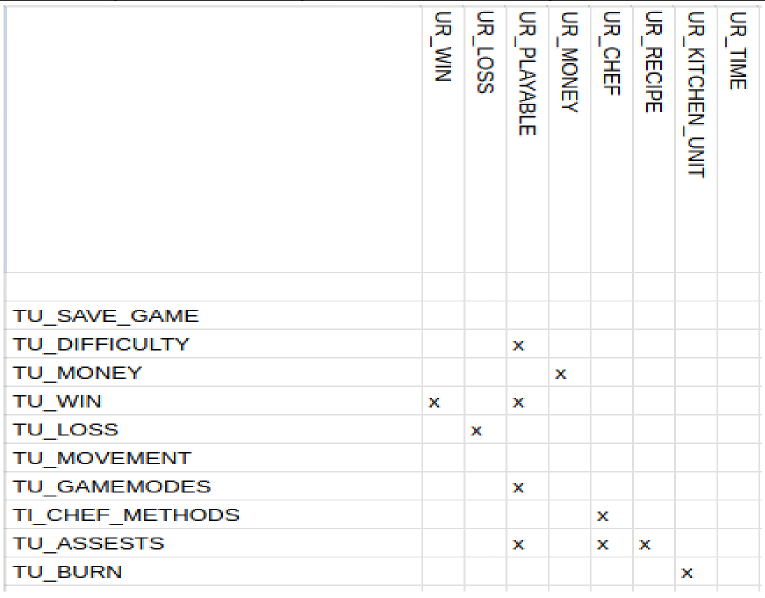
Once all the lower level methods are tested using unit tests. The next step is to make integration tests, this will be used to combine methods from different classes to make sure the functional user requirements have been met. Integration tests will be used when testing separate methods from different parts of the system working together as defined by the user requirements. This will make it easier to detect bugs as the game gets developed. Method that will be tested like this are checking recipes have correct ingredients and chef functionality.

When automatic tests are not possible we will use manual testing such as testing the GUI. This is a type of software testing that involves human intervention to manually execute tests and verify the expected results by testing the system as a whole, by running the game. This will allow exploratory testing. Allowing us to cover some parts of the software that we were unable to test through automated testing. Such as various non-functional requirements and UI layouts. Because manual testing uses human intuition, meaning the tester can use their own experience to identify potential issues. We can also incorporate context-specific testing, allowing the tester to use different personas and scenarios. For example testing is a game on a projector or on someone who has never played it before. However, when designing manual tests we have to consider it can be time-consuming, labour-intensive, and prone to human error, which is why we must automatically test as much of the application as possible.

4.b)

To aid the test development we have made a test full test report spreadsheet (link) and also a test matrix (link) they can also be found on our website. The test report covers in detail about each test and their status. The test matrix is used to make sure we cover all requirements with as little overlap as possible. Samples of these documents are shown below.

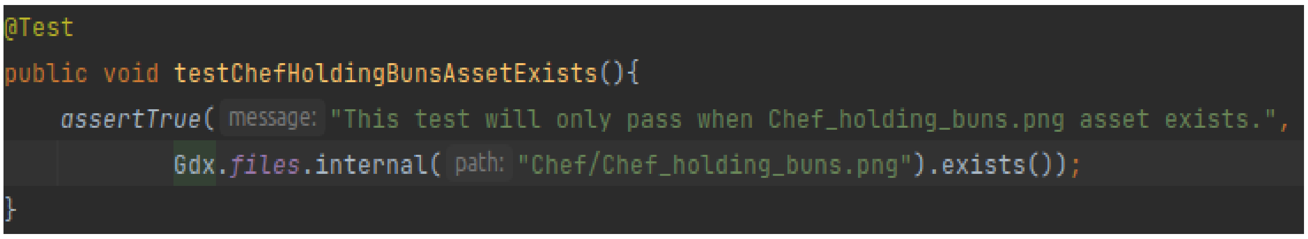
ID	Description	Requirements	Expected Outcome	Location	Status	Author
TM_SAVE_GAME	Make sure the saving and loading of the game functions create and load correctly	UR_SAVE_GAME, FR_SAVE_GAME	When saving a game state it should be written to file "game Save.text" and when loading a game state it should update the position of the chef the money and rep. points of the player	see manual testing document	PASS	B
TU_DIFFICULTY	There should be 3 difficulties (Easy, Medium, Hard) these must be selectable from a function	UR_PLAYABLE, UR_DIFFICULTY, FR_DIFFICULTY	Check that difficulty changes how much the order timer decrements for each of the different difficulties	OrderTimerTest.java	PASS	B
TU_MONEY	The player will receive money after completing an order	UR_MONEY, FR_MONEY	On completion of an order the money should increase by x	MoneyTest.java	PASS	A



	UR_WIN	UR_LOSS	UR_PLAYABLE	UR_MONEY	UR_CHEF	UR_RECIPE	UR_KITCHEN_UNIT	UR_TIME
TU_SAVE_GAME								
TU_DIFFICULTY			x					
TU_MONEY				x				
TU_WIN	x		x					
TU_LOSS		x						
TU_MOVEMENT								
TU_GAMEMODES			x					
TI_CHEF_METHODS					x			
TU_ASSESTS			x		x	x		
TU_BURN							x	

Automatic Testing

In the we cover as much of the code as possible with unit tests. For example, the asset test is shown below.



```
@Test
public void testChefHoldingBunsAssetExists(){
    assertTrue( message: "This test will only pass when Chef_holding_buns.png asset exists.",
            Gdx.files.internal( path: "Chef/Chef_holding_buns.png").exists());
}
```

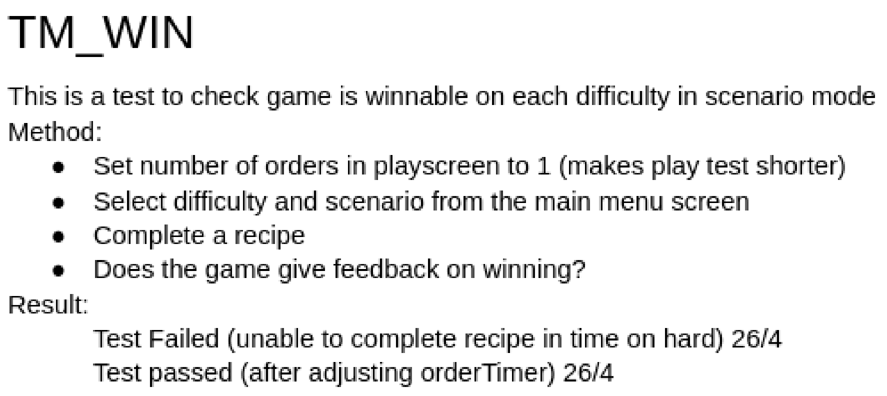
These tests can be run easily from inside the Intellij IDE, our group chosen IDE. So when adding new features, if it passes these tests the feature wouldn't have broken anything. All these tests can be found on our github. The unit tests are easy to design and cover the basic parts of the program in order to get it to run.

When designing integrated tests we encountered problems to do with batches and drawing texture, this isn't possible in the headless runner as it doesn't have a screen to draw to. A work around was by defining these classes in the show method and using lazy loading in the screen so it isn't called when instantiating the Screen class this allowed for us to access more code using unit testing however still made testing some classes impossible without completing changing the architecture.

The integrated test covers much more of the program than the unit tests, but some parts still can't be tested due to the problem mentioned above. The tests that we have written do work such as chef tests and make sure an order contains the right ingredients.

Manual Testing

Where we encountered problems and were unable to access certain methods we ensured we included the functionality of these classes in the manual tests. Along with these we made sure to test all non-functional requirements. Our manual tests are defined by a series of instructions the tester has to carry out inorder to make sure that the program works as expected, along with in some cases a persona or scenario
An example for how the game win is tested can be found below but a full manual test can be found here (link).



TM_WIN

This is a test to check game is winnable on each difficulty in scenario mode
Method:

- Set number of orders in playscreen to 1 (makes play test shorter)
- Select difficulty and scenario from the main menu screen
- Complete a recipe
- Does the game give feedback on winning?

Result:

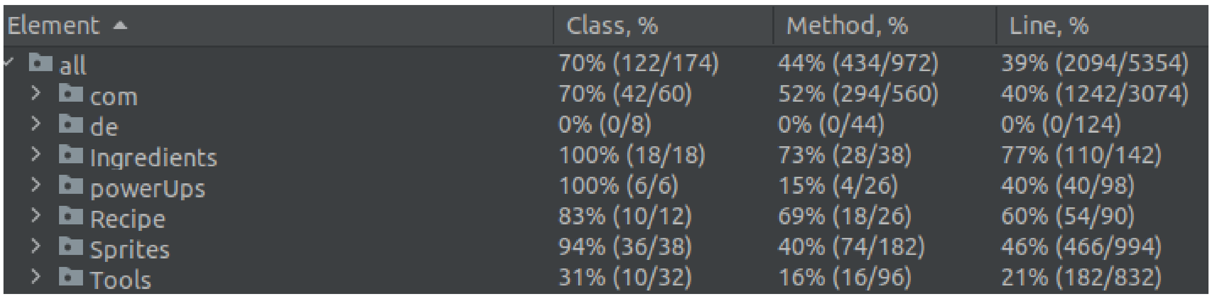
Test Failed (unable to complete recipe in time on hard) 26/4
Test passed (after adjusting orderTimer) 26/4

Here you can see that changing certain variables will decrease the amount of time to complete the test while also ensuring that the requirements are met. Saving time but also making tests easier to carry out.

Some of these tests failed as the game sometimes crashes. The reason for this is unknown but when the game doesn't crash the test is possible to complete. Also making note of when the game crashed during play testing will help us debug this problem in the future.

Coverage and Completeness of Tests

Using Intellij to generate test coverage we are able to generate the coverage of automatic tests as shown below. As you can see we cover a good proportion of our code, such as all ingredients classes and most recipe classes.



Element	Class, %	Method, %	Line, %
all	70% (122/174)	44% (434/972)	39% (2094/5354)
com	70% (42/60)	52% (294/560)	40% (1242/3074)
de	0% (0/8)	0% (0/44)	0% (0/124)
Ingredients	100% (18/18)	73% (28/38)	77% (110/142)
powerUps	100% (6/6)	15% (4/26)	40% (40/98)
Recipe	83% (10/12)	69% (18/26)	60% (54/90)
Sprites	94% (36/38)	40% (74/182)	46% (466/994)
Tools	31% (10/32)	16% (16/96)	21% (182/832)

The main class of code that hasn't been tested are the ones that involve sprite and textures and these are tested with the manual test. There will be a full coverage report on our website.

4. c)

Testing material on website: https://t12official.github.io/t12squared/#testing-documentation
Testing results and coverage report:
https://t12official.github.io/t12squared/files/htmlReport/index.html
Manual test cases: Manual Testing
Testing report and test matrix: Testing Report and Test Matrix

References:

https://www.ibm.com/docs/en/elms/elm/6.0.6?topic=requirements-based-testing [1]