Software Testing Report

Team 16

CatepillaDevelopment

Yousif Al-Rufaye
Joe Fuller
William Gracie-Langrick
Jack Hardy
Bailey Uniacke
Ben Young

a) Methods and approaches

Our testing method involved automated unit testing using JUnit. Automated testing is very useful as it can be run quickly after each new change to establish whether the recent changes have caused problems.

JUnit testing also offers us a framework that gives us clear structure to our tests which makes it easier to build and grow our test base over time. Keeping it clean and organised at all times. Providing that we have used valid test names for our methods and clear error messages that illustrate when things go wrong. JUnit testing also provides useful methods for checking values; with its assertEquals functionality which is used for the majority of our testing.

Our approach to testing focuses on verification through inspection rather than validation. The style of the project provides and allows us to test for key values which will tell us if the way the program is correct. Rather than having to create our own environment to test the logic of the game. To do this would create unnecessary repetitions of code as we check the test process is the same as the process in the project. We already know the ideal way to build the game.

Test sizes are likely to be small in size. Usually testing only 1 statement at a time. Unit tests are usually narrow in scope as they focus on key values. Large tests tend to be more complicated and contain more moving parts, this is an opportunity to introduce errors. If we have to test on a larger scale, it is more effective to simulate the larger program while only using a smaller part of the system. This can reduce the problems introduced, but it is important that this is accurate and doesn't hide problems in the overall program.

Coverage of the project will be based on structural coverage, where each method is decided on whether it can or needs to be tested, rather than each and every statement/line. It is not expected that all methods will be covered. Including those methods that are private or utility methods only. A good coverage may expect to see upwards of 80% of methods covered.

Testing inputs do not need to be validated as the interface of our game allows only very restricted user inputs and the library we are using to process these user inputs already has validation included within it. Therefore, when testing user inputs, we will not use a range or random and close to error values. We will instead use single example inputs and test only the logic of our system through expected outputs.

This methodology should provide a well rounded and covered testing check for our system, while not over testing and creating unnecessary work for ourselves.

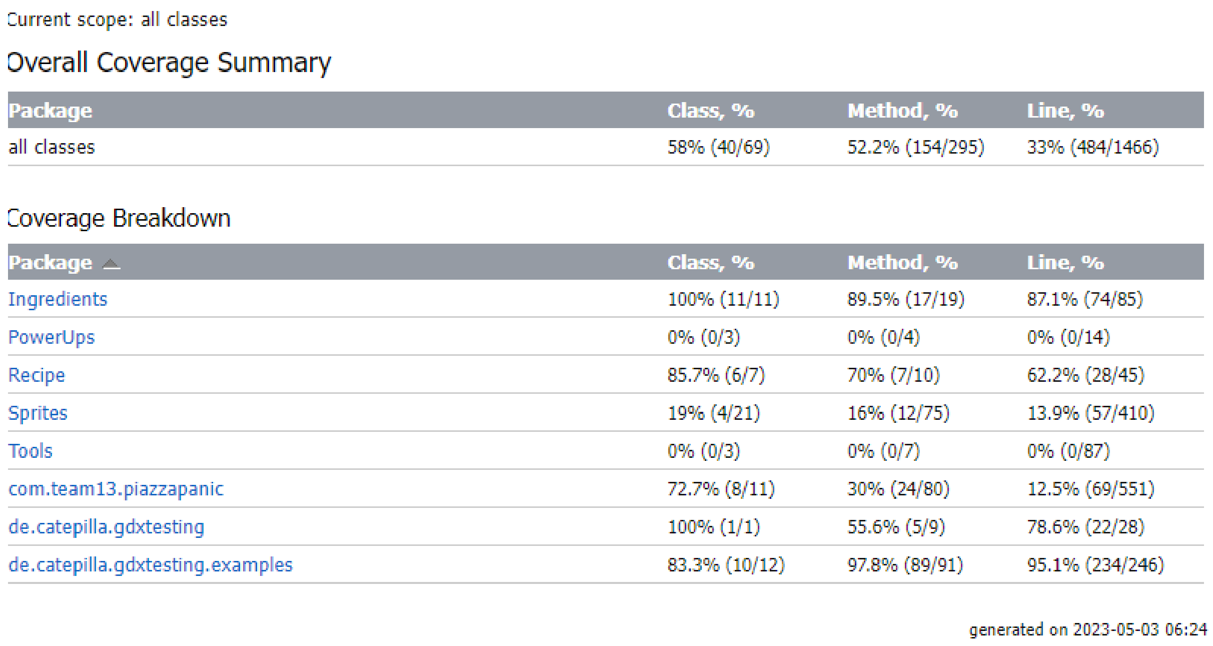
b) Testing Report

Test traceability

Where related and necessary, tests are linked back to their requirements by the requirement ID in the tests method as a comment. We always attempted automated testing on each part of the system, however this was not always possible. So sometimes manual testing had to be used. These tests links to the requirements are stated below.

Automated testing

The original plan for testing in our project was to achieve majority coverage with automated testing. Conclusively, this was not possible due to a variety of reasons explained here. Below is a table of the coverage produced by IntelliJ.



Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	58% (40/69)	52.2% (154/295)	33% (484/1466)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
Ingredients	100% (11/11)	89.5% (17/19)	87.1% (74/85)
PowerUps	0% (0/3)	0% (0/4)	0% (0/14)
Recipe	85.7% (6/7)	70% (7/10)	62.2% (28/45)
Sprites	19% (4/21)	16% (12/75)	13.9% (57/410)
Tools	0% (0/3)	0% (0/7)	0% (0/87)
com.team13.piazzapanic	72.7% (8/11)	30% (24/80)	12.5% (69/551)
de.catepilla.gdxtesting	100% (1/1)	55.6% (5/9)	78.6% (22/28)
de.catepilla.gdxtesting.examples	83.3% (10/12)	97.8% (89/91)	95.1% (234/246)

generated on 2023-05-03 06:24

We have achieved over 50% method coverage, this is obviously lower than expected for automated testing coverage. But some testing is to be done manually after some problems. Here is how testing is conducted for each part of the package.

- Ingredients, this class and children were easy to test after implementing getter and setter methods. The rendering functionality was already supered in the parent class so it was possible to just not test this. All possible ingredients are tested being instantiated.
- Powerups and Saving were new functionalities that were implemented too late in development to be tested properly. This fault here lies with a lack of communication between testing and implementation teams, we should have suggested a timeline that allowed us to test these features.
- Recipe; there are tests written for each child in the recipe class. However these do not pass due to a problem with the .isEquals() function discussed below.
- Sprites were integrated too strongly with the rendering methods of LibGDX that caused us problems. One Class within Sprites; Chef has been refactored with tests.
- Tools, this is part of the LibGDX library and does not need testing.
- com.team13.piazzapanic; Contains the main files with logic for the game. The GameState file has had its logical aspects tested for the most part. And some features of MainGame, such as Chef Count But the other files have not been tested.

- The testing packages are automatically included in the testing scope, but contain the tests and do not need testing.

Problems with testing

The dominating problem was the incompatibility between the UnitTesting software and the LibGDX rendering and world loading methods. As guided by documentation and the lectures, we created a headless backer to function as the backend of a running gradle application. This ran many of the gradle functions in the backend and simulated many required functions. This allowed for some tests, but was not enough to simulate a game window. Therefore all functions that included a draw() function, or similar, were not possible to run during a testing build.

Normally, if this is known during production, then the rendering methods can be kept separate from the logic and physics code. With the rendering drawing values from these objects and calculations in the same way that our tests would. However this was not initially possible as the rendering was integrated fully with objects methods. Often, objects were rendered first and then values for interactions were drawn from the rendered sprites class rather than the other way around.
In order to write tests for this code therefore refactoring of the original code is needed. Using lazy execution, we can check whether a test is running the method and change the function to only instantiate statements that we know we can run. We have completed this for the Chef class to some success. This change was complicated and involved changes in multiple files that had dependencies across the repository. This took us a long time so unfortunately we have not been able to completely implement this test structure on other methods or classes. Now we understand this, it may be possible to complete in a further sprint on the project.

Additionally, many classes were missing functionality that would allow us to test them. Most classes did not include getter and setter methods unless they were used elsewhere in the program. There were many functions that called on private methods or values, which could not be called when testing or caused unknown errors when made public.

Some classes stored objects as LibGDX defined classes. These did not include useful (==) or .isEquals() functionality. When attempting to compare Textures for example, the same "Bun" file path would be used, but had a differing memory ID so the comparator returned that the values were not the same. This was also a problem for some user defined classes, where this function could have been implemented by the testing team. This didn't happen as there was confusion as to how the equivalency should be defined.

Many features of these could be implemented with further time spent on the project; a majority of effort towards testing was spent familiarising ourselves with the project, how unit testing can and can't be used with the software and libraries we had. Both team members had been assigned to documentation in the previous sprint. Though we both had a good understanding of the overall architecture and structuring of the project, neither of us had familiarities with the intricacies of the LibGDX library, which increased the time it took to start testing as we had to understand this as well. This problem can be attributed to our role assignment at the beginning rather than our testing implementation.

Failed tests

In our tests, a total of 13 tests have failed.

- ChefMovementTest - this results in an error due to there being no methods to run in the class, this was due to the test being commented out because it was not failing due to the gamestate variable being private.
- TestChefIndex - this test fails due to it being incomplete.
- PowerUpTests - this test results in an fails and results in an error due to it requiring gamestate to be initialized which is a private variable.
- SaladRecipeTest, CookedPizzaRecipeTest, JacketPotatoRecipeTest, RawPizzaRecipeTest, BurgerRecipeTest - These tests failed due to not being able to compare the textures of the ingredients using the available comparators.
- CanLoad, CanSave, TimeLoadCorrectly - These tests fail and result in an error due to them not being implemented fully.
- zeroOrdersRemainingTest, OrdersRemainingTest - These tests fail and result in an error due to them requiring MainGame to be initialized, which needs the spritebatch to be intitialised which cannot be due to it being part or the rendering of the game.

c) Website Material URLs.

https://lucky13dev.github.io/files/assessment2/Test_Results-20230503T080513Z-001.zip

https://lucky13dev.github.io/files/assessment2/Coverage-20230503T080515Z-001.zip

Manual Testing

For those parts of the code that we were unable to test automatically, manual testing allows us to check for any problems. This has been especially helpful for testing user interface requirements as these have not been testable due to restrictions with LibGDX, but also for testing features that may have been complicated to create tests for but can be viewed simply by running and quickly creating scenarios in the game.

The following requirements were tested using manual testing:

- FR_FINISH - There are tests for this but they consistently fail as the testing environment is inconsistent. Therefore, manual testing is used. This test can take a while to complete depending on the skill level of the user participating in the game.
- FR_MOVEMENT; FR_CONTROLS; FR_SWITCH - These can be visually completed on opening the game.
- FR_INTERACT - More difficult to complete a thorough test for this when only using vision in manual testing. Especially for edge cases.
- FR_SWITCH_INFORM_USER; FR_RP_LOSS_INFORM_USER; FR_DEMO - Clear notifications can only be tested through manual testing.
- FR_CONTINUE_ACTION; FR_MONEY; FR_COLLISION - Scenarios where these can be completed can be less consistently set up which can make manual testing difficult.
- FR_POWERUPS - There are tests for this but due to them failing because of certain aspects in the environment being inconsistent, they have been commented out with comments explaining the reasons, and we resorted to manual testing for the powerups, which could prove to be difficult to test manually due to the way powerups are obtained in the game.

Some Requirements in the requirements table were not fully implemented, so they cannot be tested.