Continuous Integration Report

Group 17

Team Loading

Joel Warren
Charlie Wilson
Jake Keast
Hari Bhandari
Ishrit Thakur
Joshua Hills

4. a) Testing method(s) and approach(es) + appropriateness

The overall testing approach has been continuous testing, in which we immediately started testing throughout the project as soon as we took ownership of the game's code. This meant that whenever new code/functions were implemented, we immediately wrote the automated test for that function. As we began implementing automated testing in our project, our first step was to introduce unit tests. Since our project used Java and LibGDX, we opted for JUnit to write the tests. Initially, we concentrated on creating tests for the most frequently updated parts of the codebase. Over time, we expanded test coverage to include most classes, emphasising numerous edge cases to ensure our game stays free from breaking changes and bugs.

This helped ensure bugs were detected early and sped up release rate. To enable testing after assuming ownership of the code, refactoring was done to allow for testability. To improve the quality, maintainability, and overall development experience, we decided to implement unit tests into our project using Java, LibGDX, and GitHub Actions.

Testing was then done through automated testing, as part of our continuous integration pipelines. We then decided to fully utilise GitHub Actions to enhance our testing process by incorporating additional features and tools from github, further improving our project's development workflow and automation. Importantly, we configured our GitHub Actions workflow to only run on the main branch, as we use main branch for stable code and dev branch for more experimental code. The workflow is triggered every time we make a commit to the main branch or merge other branches into it and we've also set up a discord webhook where it sends us messages if something goes wrong, so any of us could take care of it. Due to the time constraints, automated testing was not always possible, and we decided to do manual testing for UI components and for those components that are least frequently changed. Automated testing speeded up delivery cycles and expanded test coverage.

These approaches of continuous/automated testing have been suitable for our project because of our agile methodology. Continuously writing tests helped with iterative development by allowing for frequent bug free pushes, as well as ensuring that each newly implemented code worked before implementing new requirements, avoiding messy bugs with big pushes to the code.

Throughout implementation, as part of our agile software development, changes and new features were implemented frequently as part of our agile development. Unit testing was therefore appropriate by helping to support these updates by allowing any changes to be tested against the existing unit tests, ensuring that nothing breaks when implementing new features.

Our automated test functions were written using unit testing, specifically junit4. These were carried out by Hari, to ensure each individual component of the software was functional and working as intended. This was suitable because it was very simple to use and the project was coded in java.

4. b) Software testing report

Report

Testing was done to cover the different components of implementation, to ensure tests were provided for all requirements of the system. Unit tests were written to cover: GameScreen, MainMenuScreen, CreditsScreen, EndGameScreen, SettingsScreen, Cook, Money, Customer, powerups, food; pizza, burger, salad, potato, order, ingredient, food menu, baking, extra chef, pantry, extra time, cutting, serving, bin, frying, serving, savegame, salad/potato/tomato/patty/bun, unlock baking test. These tests cover all the basic components and functionalities of the game.

Respectively, these cover all the functional and user requirements [https://eng1-loading.github.io/assessment_2-website/pdf/ReqUpdated.pdf]
However, some requirements, were untestable because they require verbal feedback, which include: UR_USER_EXPERIENCE as well as all the nonfunctional requirements except NFR_PORTABILITY, which was tested correctly that the game does run on Windows, MacOS and Linux.

Statistics

A total of 149 tests were run, with a coverage of 77% (12794 of 14,124 instructions).
The following test elements had the following coverage:

Element	**Missed Instructions**	**Coverage**
Screens	2019 of 5999	66%
Clickables	584 of 2610	77%
game	418 of 1079	61%
ScreenTests	256 of 1316	80%
Powerups	178 of 611	70%
Food	126 of 830	84%
FoodTests	88 of 452	80%
ClickableTests	59 of 2218	97%
tests	26 of 1433	98%

Results

The results are as follows: 100% of the 12,794 tests run were successful.

Failed tests

For the tests written, no tests failed. The tests run are largely complete and correct. With a coverage of 77%, most components were able to be tested automatically, which covered all functional and user requirements where possible. Further adding to this, the completeness was improved by using manual tests to test where automated testing was not feasible, which is further explained below.

However, the testing is not 100% complete. Most non-functional requirements could not be tested with our testing methods, such as NFR_USABILITY and NFR_USER_ENGAGEMENT. This would require different testing methods such as player testing giving feedback on the game. This was not carried out because we ran out of time to arrange player testing. This is something that would be done in the future to ensure that the game meets the customer requirements and is suitable for a family friendly enjoyable game to be played at the University of York's open days.

Manual tests explanation

We achieved 77% coverage for our automated testing, the testing that was not covered is due to mainly the fact that they are ui components, such as render functions or functions whose primary job is to manipulate UI components. For example instead of writing automated tests for things like click events we went the much more efficient way of testing during development by running the game clicking the button and then iterating this process during any changes and then ultimately testing at the end of the functions development. This made more sense to use than an automated test due to the fact that everytime we tested a different function it was likely that we would have to click one of these clickables and that would tell us whether it was working as expected. To test the functional aspect of click components we refactored the click events to move the code used in the click to its own separate function, then we can write tests for the functionality and manually test that the actual click action works.

In the screen package, most notably the gamescreen class, we wrote automated tests for all functions whose sole purpose was something other than modifying how the game looks. Then for things that did modify the UI we observed what changes were present in the UI upon running and doing different actions. We also used print statements to see what was happening in the internal runnings. For example when implementing the third chef the station collision was broken, a critical part in finding the solution to this was using print statements to print out current stations this way we could see where things were going wrong and what we needed to modify to fix the bug.

There were some components of our project that we could not test due to our headless testing environment, these tests were all either methods used by the UI or UI components themselves. To get around this we used forms of manual testing to verify that what the components did is what was expected of them. One way we did this was using a mixture of Tiled to open our map, along with printing co-ordinates of cooks/customers to verify that they were going to the correct place. The process of this was as follows, we opened up our pixmap in Tiled, hovered our mouse where we expected the player to be, noted down the

coordinates of that place, made the action that would make the cook move to that place and then print out the cooks current coordinates to cross reference.

Another way we manually tested was to add a line to the code that would perform an action. An example of this would be to verify that the rep was being displayed correctly. We added a key listener that upon being clicked would decrement the rep counter variable and we looked to make sure that it was visually being decremented too.

We verified all the textures were being shown correctly by adding them in, bringing them to the front and making sure there were no visual bugs with it. Then moving them to where they needed to be, performing the action that should show them, and then verifying that they were there.

We also slightly refactored the gamescreen code to move functions that weren't manipulating the UI (like our customersToServe()) function to a helpers class that let us test the functionality of the function without having to interact with the UI.