Additions are marked in yellow, reductions are also ~~strike-through~~.

TESTING JUSTIFICATION

Traceability Matrix:
~~https://drive.google.com/file/d/1hVEtNUmXRG6uwWm0Vq5bAwHViH_dt31/view~~
https://jordan00789.github.io/SpaceKeyProjects/Assessment3/traceabilitymatrix.png

The team acknowledges the failure in generating robust unit testing material to support the game's architecture. The reasons behind this are multiple: miscalculation of time and effort allocation in ~~primis~~ the first place, followed by an insufficient preventive research of the necessary tools and technologies needed for developing such tests. The Project Manager takes full ownership of the responsibilities of the former, while sharing with the Tech Lead the liabilities of the latter. This is something which no team member takes lightly, and everyone will reflect on this experience, treasuring it as a lesson for the incoming assessments.

Despite a thorough understanding of JUnit tests and the differences and applications of white- and black-box testing from every member in the team, what refrained us from initially achieving the results expected was the presence of too many dependencies between classes and methods which could not be broken with external tools and libraries such as Mockito. For example, creating a new instance of the game Kroy would require the instantiation of a new MenuScreen and GameScreen, which would return ~~return~~ a NullPointerException that traces back to the SpriteBatch of the instantiated screen being null, therefore making it impossible to test any method or function. However most of this was changed in phase 3 of the project, a lot of the code was refactored to allow unit tests to be implemented easily.

Nonetheless, we employed multiple methods to test our game, we took wide reaching methods to testing including user play-testing and debug features. We also decided, for the sake of consistency throughout the assessments, to stick with the ISO/IEC/IEEE 29119 standards [1][2] for software testing design report and traceability matrix (URL). A debug feature was implemented, which can be controlled directly from the main menu, by setting the 'showDebug' variable to true, which draws on the screen various important aspects of the player's view, such as the hitboxes. This allowed both us and any further developers to ensure that these usually unseen objects are aligned, located and are moving correctly. It was also created as a separate class in order to facilitate the addition and ~~remotion~~ removal of further elements to draw.

For phase 3 of our project, we used JUnit testing for our automated unit tests. JUnit is simple and easy to use with java and we used JUnit testing to test the entities as a lot of the important game logic relies on entities. Therefore, to make sure our game works properly and the game logic is correct, we used JUnit testing. The **FireStationTest, FireTruckTest** and **FortressTest** were tested using Junit and Mockito. JUnit testing is useful when new features are added as it only tests specific parts of the programme under the condition the code doesn't have many dependencies.

Acceptance testing involves testing the software that is tested for acceptability. The purpose of this is to test whether the software has met the core requirements and fulfilled the fit criteria. The software is constantly compared the to fit criteria to ensure requirements are met.

Finally, we also used play-testing. While this is not the most robust testing method, it allowed us to emulate behaviour similar to our players. This showed us issues we had not expected, nor had been able to test for - such as an infinitely scrolling map past the boundaries, and sprites overlapping other sprites that should be further in the foreground. This was particularly important since we were using external libraries which we couldn't unit test ourselves - by using this playtesting we were able to recognise some of the quirks of how our code worked with these libraries.

Additions are marked in yellow, reductions are also ~~strike-through~~.

~~In retrospect, we believe we could have begun the testing process earlier in the development cycle than we did. While we had a rigid architecture set out, it would have allowed us to cement that architecture, as well as resolve certain issues much quicker than we did, and also allow us to control the amount of time we spent reworking sections of the code.~~

References [1] Software Testing Standard Website [Online] Available: http://softwaretestingstandard.org/ [2] Summary of IEEE Software Testing Standard [Online] Available http://www.cs.otago.ac.nz/cosc345/lecs/lec22/testplan.htm

Additions are marked in yellow, reductions are also ~~strike-through~~.

TESTING REPORT

~~The inability to produce unit tests for the game did not stop the team from coming up with alternative ways of testing the correctness of our code. On the other hand, using various types of testing methods provided a comprehensive analysis of the system and examined the functionality in multiple areas.~~
Unexpected errors started to rise early in the project, which the team managed to quickly rectify by implementing useful and straightforward debugging tools. Because of this direct approach, the team is confident that the testing is sufficient to prove that the code meets the requirements specifications in most areas.

Junit Testing

Alongside straightforward debugging tools, a large amount of the code was refactored to reduce dependencies between classes and methods, which allowed us to implement automated Unit testing in the form of J Unit tests. For our unit testing, each unit test was assigned a "Test ID", "Test function name", "function tested", "function use", "result of test" and "Test description". A ID was given so tests could be referred to easily within the report and the traceability matrix. The "Test function name" is given to the methods that are tested by each JUnit test. "Function tested" was given to describe what the method being tested does. We included a result to check if the test passes the test and a test description to describe what the test does. The **FireStationTest, FireTruckTest** and **FortressTest** were done using Junit and Mockito. Mockito is a add on for Junit testing which allows the creation of test double objects. This allowed us to test important logic for the game and not just get functions and set functions. For our JUnit testing all our tests passed. Our Junit testing made sure to get rid of any bugs present in the game and any problems with the game game logic.

The reason we chose to test **FireTruck, FireStation, Fortress, Goose** and **Pipe** for our automated Unit testing is because the testing team felt that testing the entities were the most important classes to test as a lot of the important game logic relies on the entities.
The Junit testing report can be found in the testing document. The link will be at the bottom of this document.

Play Testing

For our Black Box testing the team decided to use play testing for user requirements and functional requirements. We decided to adopt DicyCat's style of play testing because the testing team thought it was robust enough to get rid of most problems. However, the team added an additional six tests that we felt were necessary to test more requirements and to test the added requirements such as the minigame which were features implemented in phase 3. Some of the current tests were also appropriately added the correct requirement ID. We also appropriately fixed the game so that tests that failed before now pass.
The Play testing report can be found in the testing document. The link will be at the bottom of this document.

Acceptance Testing

We used acceptance testing to test the non-functional requirements for the game. Acceptance testing is a form of black box testing. Each test was assigned an "ID", so it could easily be reffered to in the report and traceability matrix, a requirement ID, which is the specific requirement that is being tested, a fit criterion which is the condition that requirement must meet so that it passes, a result to say if it passes or fails and the relevant evidence to say why a test failed or passed. For our

Additions are marked in yellow, reductions are also ~~strike-through~~.

acceptance testing the only test that failed was A_5, which was a test to check if there was a colour-blind mode for the game to improve accessibility. The reason this test failed is because this feature wasn't implemented. This was a requirement that the team felt wasn't necessary add as the user requirement associated with this was UR_COLOUR_ACCESSIBILITY, which was a low priority. Due to short turnaround for the project this was not implemented.
The Acceptance testing report can be found in the testing document. The link will be at the bottom of this document.

The biggest factor that conditioned the whole testing process, however, was time: towards the late stages of the development, features did not work as intended, and the team decided that given the limited time left it was not possible to safely make changes to the code without having the risk of upsetting the whole structure and functioning of the game. The team therefore decided to stick with the testing material that was produced so far and use this as a lesson for the next assessments.
~~Below there is a table with the functionality tests designed, and their results:~~

The results of the testing can be found here:

https://jordan00789.github.io/SpaceKeyProjects/Assessment3/a3utests.pdf