EXTENSION REPORT

Introduction

This is a documentation of the extension of the software "Don't crash" by team WAW. The software to be extended is an s-type program i.e. Its functionality and design are influenced by an external source i.e. Tim Kelly who is the customer. As this is an s-type program, evolving the software requires a software engineering approach which introduces change at a specification level; hence modification of the initial requirements (corresponding to assessment 2) is needed before implementing any changes.

Tasks are divided as follows:

- Extension report (Aishat)
- Implementation (Radostin, Tim)
- Updated user manual (Chris)
- Test plan/report (Katie, Valentine)

Software Engineering Approach

The software change management approach to be implemented in this project is a process consisting of various important stages; each stage will be managed closely by splitting the process into tasks amongst the team members and working accordingly. The various stages in the software engineering approach chosen are well-defined below:

- **Preliminary testing:** This is the first stage of the process and the aim is to test the software against the requirements collected by our team (BHD) for assessment two. This is needed to ensure that the software we have received is up to date and satisfies all of the customers' requirements listed before now. This stage will also help us discover any loopholes and be more familiar with the software's code and structure, this is very essential for when we begin to modify its design, architecture, functionality and code. Details of the tests carried are documented in the test document which is submitted as a different file.
- **Modification of initial Requirements:** This is the second stage of the process and the aim is to compare the requirements collated for the previous assessment with the customers' new requirements and make adjustments accordingly. It is important this is done in order to monitor that all of the customer's requirements listed in the specification document (from assessment two and three) are met. It is also at this stage that we are able to clarify our understanding of what the customer needs and what the finished product should do. This stage has a major impact on all the other stages as it is the yardstick and foundation for making changes to the product that satisfy the customer's requirements. For this project, the new requirements for assessment three will generally be an extension on the requirements from assessment 2.
- **Impact Analysis on the Initial Software Architecture:** This is the third stage of the process and the goal is to determine the ideal architectural pattern/ style for the system. At this stage we will be analysing the software architecture to see if implementing the new requirements calls for a different architectural pattern/style.
- **Impact Analysis on the Initial Software Design:** This is the fourth stage of the process and the aim is to examine the status of the software's current design and then, to analyse the impact of the modified requirements on its design. Extending the software according to the modified requirements may require the addition or removal of classes, modules, components, functions etc. therefore it is at this stage all these options are determined.

  Impact Analysis on the Initial GUI: This is the fifth stage of the process and the aim is to analyse how the modification of the requirements affects the GUI that is already being implemented in the software. This is especially important if the new requirements require new functionality or interaction between the software and the user, since users generally interact with software via the GUI.

- **Impact Analysis on Testing:** This is the sixth stage of the process and it involves of analysing existing tests and deciding on tests that need to be conducted to ensure that the finished product satisfies the customer. This will generally be a concatenation of tests used in assessment two and additional tests generated and added to the test plan to check that all the changes, features and functionalities requested by the client are met.
- **Implementation of new design:** This is the seventh stage of the process, and it is at this stage where the implementation team implement the new design we have arrived at, after the several analyses conducted to ensure the resulting software conforms to the changes in the customer's requirements.
- **Implementation of new GUI:** This is the eighth stage of the whole approach and the section stage of the implementation process. The aim of this stage is to update the GUI to match the modifications that have occurred in the software due to the change requirements and to ensure that the GUI completely fulfils its purpose, even with additional features implemented.
- **Tests:** This is the second to the last stage of the approach being used for this project, and the aim is to test the resulting software against the modified test plan created in a previous stage. This is done to create evidence that the software satisfies all the customer's requirements whilst being an efficient and effective solution.
- **Debugging:** This is the last stage and the goal is to find and completely remove or reduce the number of bugs or defects in the software.

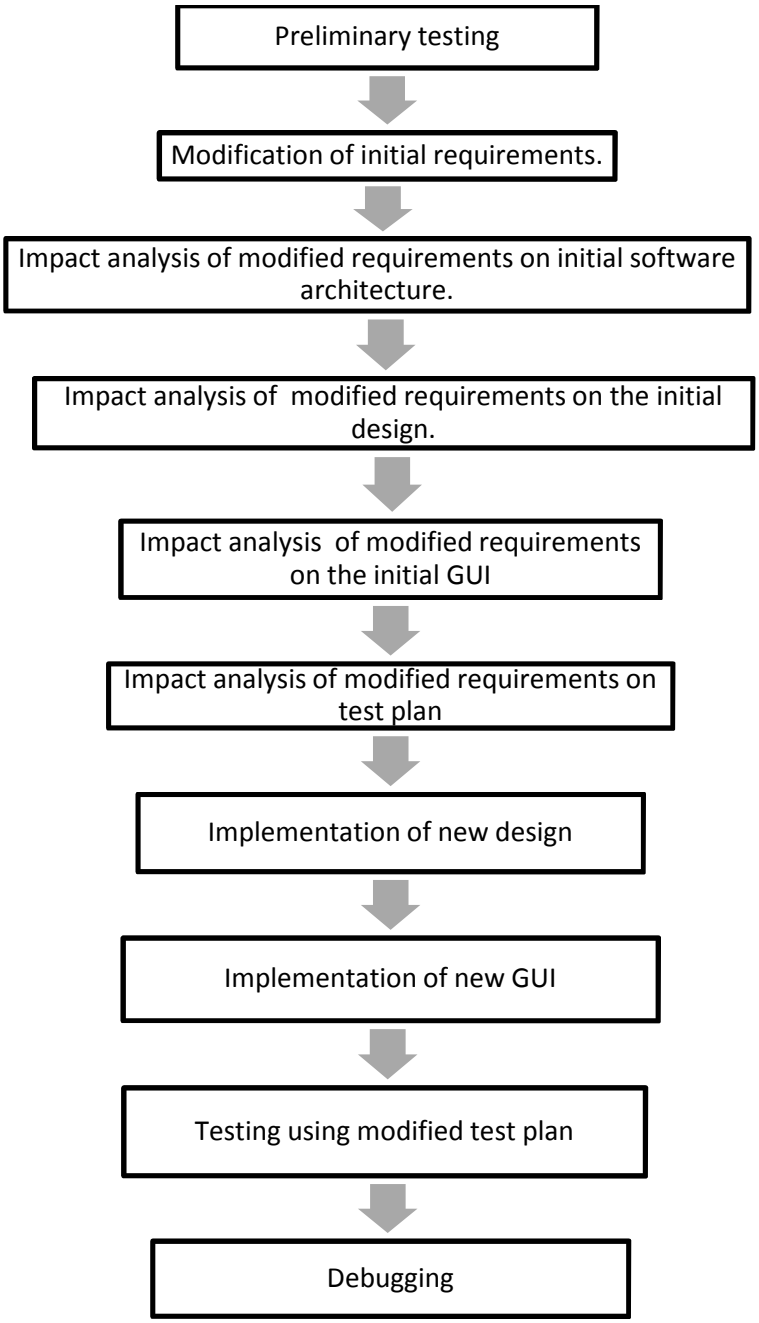

Figure 1 Software approach diagram

Preliminary Testing

For the preliminary testing, we conducted black box testing using our requirements to check which of our requirements gathered for assessment two were satisfied in the game. In addition, we also ran the other team's unit tests [2] to ensure that the software was in good working order.

Overview of results:

Most of our initial requirements were implemented in the software however; a few were not implemented because they were not necessarily required for fulfilling the specification for assessment two. The benefit of conducting these preliminary tests using our initial requirements is that we're able to determine which requirements need to be implemented to fulfil the specification for assessment two. Conducting the unit tests using the other teams' unit tests was useful in determining that all interfaces, classes, methods, modules and functions in the source code are fit for use. Generally, all the unit tests conducted on the source code at this stage were successful i.e. no problems were found with the source code and we could progress on making suitable changes to the code.

Modification of Initial Requirements: Functional Requirements

The requirements gathered for this project are a concatenation of the initial requirements used for assessment two and new requirements. These requirements were derived from the client brief and consistent communication with the client for clarity. Requirements from the previous assessment are included because they serve as a firm foundation for satisfying the additional requirements; it is difficult to satisfy the clients' new requirements without satisfying the ones previously stated. All the requirements are stated below:

ER= Extension Requirement

UR= User Requirement

SR= System Requirement

Extension requirements:

1. An aircraft can land at an airport - **ER1.**
2. An aircraft can take off from an airport - **ER2.**
3. Airports must exist and be visible to the user - **ER3.**
4. There must be constraints on how many aircraft can land or take off at any one time - **ER4.**
5. Flight plans must be able to enter and land at an airport or take off from an airport and exit - **ER5.**
6. The score must be displayed for a particular game while it is being played - **ER6.**
7. When a game has finished, the score should be added to a list of high scores - **ER7.**
8. The list of high scores should be available for the user to see - **ER8.**
9. At least ten flights should be allowed in the airspace at any one time - **ER9.**

Initial requirements:

10. Have access to an in-game "help" screen - it is important for new users to be able to understand the mechanics of the game - **UR1**.
11. See airspace - the game is simply meant to emulate the daily work of an air traffic controller - **UR2.**
12. Start the game after examining the airspace - since the airspace in the game might be new and complicated for the player, they should be able to examine it before getting into the game - **UR3**.
13. Alter the course of flights that enter the airspace - without this ability, the game will lack any gameplay whatsoever - UR4
14. Direct flights towards waypoints - the simplest way to direct flights - **UR5.**
15. Have flights land and take off - a "landed" flight is a possible way to score points - **UR6**
16. Know how much time they have spent in the game - a way to help the player score and weigh down their own performance without necessarily ending his game **- UR7.**
17. See a score, which will be a way for the player to assess their own performance – **UR8.**
18. Have a feeling for "challenge" - provide an online leader board to create an impetus for users to play the game - **UR9.**
19. Be able to "lose" - without a challenge and a risk of losing, most games are perceived poor by critics and players alike - **UR10.**
20. Be able to quit and see a high score without "losing" - a simple preventive option, meant to be used if the player would want to quit playing before actually losing the game - **UR11.**
21. Be able to pause and restart the game "at will" without exiting it - **UR12.**
22. Have a main menu - the easiest and most common way for interfacing between the player and the game prior to the launch of a game session - **SR1.**

23. Have a help/instructions screen - in direct relation to UR1 - **SR2.**
24. Have a "start" button for the user to start the game - in direct relation to UR2 - **SR3.**
25. Generate an airspace - without having the system create a random or predefined airspace, no gameplay can be conducted - **SR4.**
26. Simulate airspace graphically - the best and most engaging way to present the game to the players - **SR5.**
27. Populate the airspace with flights - without flights, the game will be exempt of gameplay -**SR6.**
28. Have a GUI with a score and a timer - a GUI allows for easy and engaging control of the game by the user - **SR7.**
29. Have varied characteristics for flights - variety is a great way to improve the player experience - **SR8.**
30. Display score - in direct relation to UR8 - **SR9.**
31. Save score - in direct relation to UR9 - **SR10.**
32. Upload score to a master repository - in direct relation to UR9 - **SR11.**
33. Monitor separation rules - separation rules are an authentic way to provide a challenge for the players and challenge is very important, as previously stated - **SR12.**
34. Generate and remove flights from the airspace via entry and exit points - the game must persistently give the player flights to manipulate - **SR13.**
35. Pre-set a flight plan - when flights come into the airspace they should have a sensible flight plan - **SR14.**
36. Alter a flight plan - players should be able to manipulate the flights in the airspace - **SR15.**
37. Change course - in direct relation to UR4 - **SR16.**
38. Change altitude - in direct relation to UR4 - **SR17.**
39. Land at airport - in direct relation to UR6 - **SR18.**
40. Take off from airport - in direct relation to UR6 - **SR19.**
41. Turn left or right by particular degree - in direct relation to UR4 - **SR20.**
42. Display updates regularly - if updates are not regular enough, the game will feel unresponsive and "stutter"-y, which will decrease player immersion and enjoyment - **SR21.**
43. Have waypoints - in direct relation to UR5 - **SR22.**
44. Occasionally set flights on crash course/near miss - without this feature, the game will mostly lack challenge - **SR23.**
45. Simulate bad weather - a feature that will provide greater challenge to players - **SR24.**
46. Simulate equipment failures - a feature providing greater challenge to players - **SR25.**
47. Have a quit button - the player must be able to exit the game seamlessly - **SR26.**
48. Game becomes harder as the timer goes up - to provide an increasing challenge arc and keep the player engaged - **SR27**.
49. Have a pause /resume functionality - **SR28.**

Modification of Initial Requirements: Non-Functional Requirements

These aspects of the game will increase player enjoyment and immersion, but all these requirements need not be satisfied in order to have a working product:

- **Efficiency** (reflect changes within 30ms) - Faster updates will provide for better and more agile decision making on the users' part.
- **Documentation** - Made even more important due to code switching.
- **Testability** – i.e. can we test that we have fulfilled our requirements?
- **Stability** - a game filled with bugs is not an enjoyable one.
- **Reliability** - The game must behave consistently, to stop players from feeling that the game has "artificial difficulty".
- **Maintainability** - code must be maintained for all 3 assessments.
- **Extensibility** - code will be built up and improved upon in 3 distinct steps, therefore must be easily extensible.
- **Accessibility –** features such as colour blind mode or large font to allow for a wider audience to enjoy our games.
- **Open source** - the code will be shared between groups, thus being open source, though not necessarily free software, is absolutely necessary.
- **Time-to-market** - our team works on a tight schedule and must adhere to it.
- **Immersion** - features such as advanced graphics and/or sounds can increase player enjoyment.
- Update website

Impact Analysis on the Initial Software Architecture [1]

The initial architectural style of the software is object-oriented. This means that the software's architecture follows a design paradigm based on the division of responsibilities for the game into individual reusable and self-sufficient

objects, each containing the data and the behaviour relevant to the object. This is obvious in the way the code is split into different classes and modules, each containing constructors, methods and functions that characterise the purpose of the class/module within the game. For example, the class, 'Airspace' in the software has a constructor and resetAirspace() as one of its methods.

The architecture of the software has been designed effectively to have a series of objects working together cooperatively. These objects are distinct, independent and loosely joined, communicate through interfaces by calling methods or accessing properties in other objects, and by sending and receiving messages. The architectural style of this code demonstrates abstraction, composition, inheritance, encapsulation, polymorphism and decoupling, all of which are key principles of object-orientated architecture.

Based on our understanding of the architecture of the code we have received, and our understanding of the requirements that need to be satisfied by the software, we have decided to extend this code using the same architectural style- Object orientated architecture pattern.

We have decided to stick with the object orientated architecture pattern that was initially used for this code because, that it makes the process of extending the code easier and quicker (especially since there is a short time limit) since we are not meddling with the architecture of the initial code and trying to rebuild it to conform to a new architectural style. Another reason why we're sticking to this architectural pattern is because it has very useful and needful advantages; the object orientated architectural style is understandable, reusable, testable, extensible and highly cohesive.

In conclusion, after identifying and analysing the architectural style pattern used in the base code, we have decided to stick with the architecture pattern already implemented. This is the object-orientated architectural style pattern.

Impact Analysis on the Initial Software Design

Implementing the new features required by the customer demands making changes to the design of the base code i.e. adding, editing and/or removing classes, methods, modules etc. Analysing the base code which is fully up to date based on the client's previous requirements, we're able to arrive at rational changes to the design that need to be implemented in order to satisfy both the initial and additional requirements. The analysis of the impact of the requirement changes on the code should serve as a narrow scope of what actually ends up being done in the base code because during implementation, trial and error means that it is difficult to actually completely foreshadow what is going to be done in the code to ensure it satisfies the customers' requirements. However it is possible to envisage that a new class, airport will be added to implement the "locations" for landings and take-offs also, new methods, fields and parameters will be added to support the functionality of the new class and the scoring system.

Impact Analysis on the Initial GUI

With a view to the marketability of the game in the swap phases, we decided that it would be necessary to rebrand the menu and title screens: Two other groups chose the same (WAW) codebase for this assessment, so we felt that keeping the present branding would potentially make it difficult for us to differentiate our product, especially if Team WAW also re-used their Assessment 2 branding with their own Assessment 3 game.

Impact Analysis on Testing

Based on the changes in the customer's requirements, we have decided to make changes to our test plan. These changes will be effective in the plan for the requirements, regression, unit and acceptance testing to be conducted on the software to ensure it satisfies all the requirements. The effect of the changes on the various test plans are outlined below:

- **Requirements testing:** A requirements test will be conducted in order to test the requirements which have been written for the extension of the game, this test will be conducted using a black-box approach. This test approach is based on the requirements collected after analysing the impact of the change in the customer's requirements and is a needful way for ensuring all requirements are meant. It is to be done manually and carried out using a test plan table describing the action taken, expected result, actual result and the requirement that is being tested.
- **Regression and Unit testing:** This is going to be done by first re-running the unit tests that came with the base code to ensure that the initial required functionality of the code is not affected by any alterations made to satisfy the new requirements. To do this effectively, suitable changes will be made to unit tests if the changes made to the part of the code their testing have required functions to have additional parameters, fields etc. This test is to be done automatically within the code to test the newly implemented functionalities. As seen in the test report[2], new unit tests for the game are written in the same java files as the old unit tests. The additional unit tests used and reasons why they have been added or particular unit tests have been altered are documented in the test report [2].
- **Acceptance testing:** This will be done once all the other test have been done, it will be used to see if the client accepts that the changes/extension made to the base code satisfies his requirements. These are to be partially based on the acceptance tests for the original version of the game as well as new tests to allow the client analyse the new functionality of the game. These tests are to be mainly scenario based, testing several requirements at a time.

Implementation of the New Design

Implementing the new design created as a result of analysing the change in the customer's requirements, (Impact analysis on the initial software design) allows for more light to be shed on the actual changes being made in the base code. As touched before in the section describing the impact analysis on the design, to satisfy the new requirements by the customer, we will be adding, editing and/or removing classes, methods modules etc. in the base code we have received. Below are the changes and justification of changes made to the base code, categorised into the classes implemented:

- **Airport.java**

  An airport class has been added. The airport class is an extension of the Point class. The init and render methods are used- init sets up the parameters of the airport; render puts the airport image on the screen.

- **Airspace.java**

  Field and Parameters added:
  - private int numberOfGameLoopsSinceLastFlightAirport : This keeps record of the number of the game loops since the last use of the airport. It is used to determine a "hold-up" state during which the player cannot land a flight and a flight cannot be generated at the airport. It is initialized to 500.
  - private boolean AirportAvailable : A Boolean that sets up the aforementioned "hold-up" state. It is initialized to true.
  - int maximumNumberOfFlightsInAirspace : This has been changed to 10 to reflect the assessment 3 brief- at least 10 planes should be in the airspace.
  - Airport parameter: An airport parameter has been added, to keep record of the airport's location.

  Methods changed:
  - init now sets the airport location.
  - resetAirspace now also resets the new parameters – the AirportAvailable, numberOfGameLoopsSinceLastFlightAirport and Airport.
  - newFlight now can generate a flight at the airport, if it is available, with 20% probability. Since having the airport as an entry (or exit) point dramatically changes a flight's flight plan, we also pass on a parameter specifying if the flight starts at the airport to the flight class constructor. This is because we want to prevent a situation where a flight has the airport set up as both the entry and exit point. Also, flights starting at the airport should be initialised stationary.
  - update now also increments the numberOfGameLoopsSinceLastFlightAirport variable. It sets AirportAvailable depending on this variable. If it is above 500, we set airportavailable to true.

- render now also renders the airport.
- removeSpecificFlight now remove flights that have had the "landing" flag set to true for the last 250 game loops. The method also checks the specific flight's flight plan. If the flight plan is empty (the flight has flown through all designated waypoints), the methods increments the score by 200. If not, score is decremented by 200.

Methods added:

- newAirport sets up a position for the airport within the game space.
- A getter for the new Airport and AirportAvailable variables.
- A resetter for the numberOfGameLoopsSinceLastFlightAirport variable.
- An addAirport method. This is because in the previous team's implementation each waypoint has to be manually added to the list of points that the airspace contains.

- **Controls.java**

Fields and Parameters changed or added:

- Changed Maximum/Minimum altitude and velocity to reflect new design: We need flights to start at a low altitude to allow for quick descent to the airport and we need to be able to halt or start immobile at the airport.

Methods changed:

- init : Previously, the game used the same images for most buttons, only named them differently. Now, we have renamed the image names to be similar.
- handleAndUpdateButtons : There is a check that a flight has taken off and not landed that renders buttons unusable if the conditions are not satisfied. The method now handles the buttons for taking off, landing, accelerating and decelerating.
- giveHeadingWithMouse: Likewise to handleAndUpdateButtons, there is a check that the flight has taken off and not landed in order to enable the mouse controls.
- render: There are now buttons for taking off, landing, accelerating and decelerating. The navigation panel to the right has also been made more congested due to the need to add 4 new buttons.

- **Flight.java**

Fields and Parameters:

- The flight class now has two additional boolean fields, takenoff and landing, that represent the states in which the aircraft can be. Those are initialized to true and false, respectively.
- The constructor takes an "entry" parameter that points out the point of origin of the flight. This parameter is used to distinguish between flights starting at the airport and flights starting at the entry points.

Methods Changed:

- generate_altitude now returns 0 in case that we create the flight at the airport.
- update now also calls updateVelocity.
- checkIfFlightAtWaypoint now increments the score by 100 if it returns true.

Methods Added:

- LandFlight: This method does a number on checks on the flight's velocity, altitude, position and the status of the airport. Then, if all the checks are passed, the landing flag is set to true and the takenoff flag is set to false (thus disabling controls). The flight's target altitude and velocity are then both set to 0.
- TakeOff : This method checks if the flight is stationary, on the ground and at the airport. If the checks are passed, it sets the takenoff flag to true (thus enabling controls) and sets the flight's target altitude and velocity to the lowest values obtainable via user controls - 1000 and 25, respectively.
- changeVelocity : This method calls the setTarget method in the flight plan, provided that the velocity is within the allowed limits [0;400]
- checkIfAtAirport : Likewise to checkIfFlightAtWaypoint, only checks in an area of a different size, compared to the waypoint check - a rectangle of 140:30 pixels around the airport exit point.
- updateVelocity : This speeds up or slows down the flight at each update (from the user) at a 0.25 step, depending on the difference between the actual current velocity and the target speed.

- **FlightPlan.java**

  Fields and Parameters:
  - We now pass on the entry point of the flight to the constructor as a parameter, in order to differentiate between flights chosen to start at the airport and those starting at the edge waypoints.
  - New targetVelocity variable to keep in store the velocity the flight strives to achieve. Initialized as equal to the current velocity. Comes complete with setter and getter.

  Methods Changed:
  - generateEntryPoint: If we pass on a signal that the flight starts at the airport via the new "entry" parameter, the airport is initialized as the first point in the flight plan.
  - buildRoute: This method now takes an "entry" parameter. If we have started at the airport, we cannot have the airport as an exit point - it is only possible to get it as an exit point, with a 20% probability, if we enter via the edge entry points.
  - generateVelocity: This method now generates velocities at a multiples of 25.

- **PlayState.java**
  - The score is now indicated on screen.
  - The waypoint positioning has been changed slightly, since waypoint F was deemed to be too close to the ariport.

- **Unchanged Classes:**
  - EntryPoint.java
  - ExitPoint.java
  - Point.java
  - SeparationRules.java
  - Waypoint.java
  - Game.java

Implementation of new GUI

ControlsState.java MenuState.java GameOverState.java PauseState.java PlayState.java

Upon investigating the implementation of the menus, we discovered that the backgrounds and the title text were merged into a single PNG image, which in the absence of the source files and with no knowledge of the fonts used, would be nearly impossible to edit cleanly. Instead of repeating this implementation decision ourselves, we decided instead to render all of our title, help, and menu elements as text rather than as images. This approach has the advantages of making it much easier to alter the branding and other text, especially as we ensured that the hover- and click-event handlers used generated widths from the actual strings, rather than, as previously, hard-coded pixel positions. This will, we think, be a significant plus-point for our project in the Swapping Phase, as it will make it much easier for other teams to add interface options, and explanatory text in the help screen, to match the additional requirements to be announced in Assessment 4.

The main changes to the GUI are new background images and fonts. Originally, the background images had the text as part of the image, now we draw them as text from strings in the code.

Relative to the menu screens, we made comparatively few changes to the gameplay screens themselves: The only changes we made were those needed to implement the Assessment 3 requirements. We added buttons to the left-hand side of the screen, christened the "control hub" by Team WAW, to support the requirements for acceleration and deceleration functionality. Similar to the existing altitude buttons, the new buttons follow the "Accelerate to 125mph" / "Decelerate to 75mph" format and are disabled when the selected flight is at its minimum or maximum altitude.

We also added a single button for the land and take-off functionality. When the selected flight is on the ground, it will perform the take-off function, and when it's in the air it will perform the land function. Since the two states are mutually exclusive, we felt that it would be a good idea to save valuable screen space by combining the two functions. All other controls are hidden when the flight is on the ground, so as to avoid presenting the user with a sea of non-functional options.

The final change we made was to add a display of the current score in the empty space next to the clock icon at the top of the left-hand column.

Testing

Overview of requirements testing results:

The results from the requirements testing conducted (documented in test report[2]) on the extended software indicate that the essential requirements i.e. The clients' requirements have been satisfied. This testing was done based on only the extension requirements listed for the changes that need to be made, these extension requirements are listed above (Impact analysis on requirements section). This result is easily observed in the way the actual result from each test match the expected result, hence satisfying the each associated requirement.

However, it is also visible in the test report[2] that the extended software does not pass tests 14-16, this is because the associated functionality of a high scores list has not been implemented. This functionality has been intentionally left out due to time constraints and since implementing it is not a priority; this functionality is not part of the client's requirement but was to be added as an additional feature (ER7 and ER8).

In conclusion, this testing was successful as it proves that required features such as the functionality of controls and the score used in the actual game are working effectively therefore, all the client's requirements are satisfied.

Overview of regression and unit testing results:

The regression and unit tests were used to test that each of the independent objects in the software (e.g. modules, classes, methods etc.) work as intended. This testing was carried out using previously created tests(i.e. tests that came with the original software) and newly created ones to ensure that all the separate functionalities needed to ensure that the game works effectively are implemented.

The results from this testing proves that the individual functionalities have been properly implemented. There were 62 unit tests carried out on the software (including old and new tests) and all were successfully passed (Details and evidence of this in the test document).

Overview of acceptance testing results:

Initial acceptance testing:

The first acceptance testing done with the client wasn't successful however, we were able to get feedback. Our client's comments/ views from the first acceptance testing are listed below:

- Score - "pass worth, considering score penalty for changing waypoints"
- Landing - "but how do we (visually) know that the plane is close enough to land?"
- Taking off - he seems to be okay with how it's implemented
- High scores - "agree that nonessential in this phase"

During this acceptance testing, the client pointed out that changing the flight plan via plan mode might reflect on the score. However, during playtests, it was observed that players tend to easily get a negative score due to the punishing difficulty of the game. Therefore, we (including the client) decided that there is no reason to further penalise the players. The result of the initial acceptance testing meant that several changes needed to be made to the software to ensure that his requirements for it are satisfied. Solving the problems that seem to have surfaced from the first acceptance testing required another acceptance testing form to be made, this was used for the second acceptance testing.

Final acceptance testing:

The second acceptance test done with the client turned out to be the last one because, the client was able to verify that all his requirements for the software have been satisfied. Details of the acceptance tests used are documented in the test report [2].

Debugging/Challenges

- Feedback on the gameover screen: a challenge we faced when extending the game is trying to ensure feedback is given on the game over screen. This challenge is a as a result of the fact that, Slick (the library used by the previous team) uses StateBasedGame. We inherited this from the previous team. As it turned out during development, it is complicated to pass on data between the different game states. The solution to this would have been to create a singleton class (that is how Java calls global classes) to carry over the global information. However, such a solution

would have required a number of changes across all the other classes and it was decided that it is not worthwhile. Another solution would have been to serialize the data in an external file, but such a solution would be very system/user-specific and, therefore, hard to maintain.
In summary, the game consists, by inception, of a number of states that are independent. There is no easy way to pass on data between the gameover state and the play state.

- One challenging aspect of implementing the change in the GUI was that the methods for dealing with fonts in the Slick2D do, unfortunately, tend to be somewhat confusing and arcane. Originally, we decided to use the TrueTypeFont class to draw our menu text, mainly because it was already in use in the game itself for drawing, amongst other things, the control buttons and flight statistics. For smaller font sizes (such as the in-game text or the smallest size of menu buttons) this seemed to work fine, though loading several different sizes, in several different game states, did slow down the initial game loading (when the init function of each state is called) somewhat. It was, however, more problematic with larger font sizes: graphical corruption when rendering some characters left screen titles unreadable beyond a certain point size. This issue was somewhat challenging to debug, especially as the font-size threshold beyond which things started varied even between different computers with the same operating system and java version. Eventually, and after spending a fair amount of time trawling through various StackOverflow threads, we came to believe that this was due to the TrueTypeFont class interacting at too low a level with the graphics card, and the larger glyphs overflowing the card's default texture size.

References

[1] http://msdn.microsoft.com/en-us/library/ee658117.aspx , Feb. 2014.

[2] Team BHD, Assessment 3 “Test report”, Department of Computer Science, University of York, Feb.2014