Assessment 4 Test Plan

Software Engineering Project (SEPR)

Team “EEP” for Team “ADB”

Richard Cosgrove, Yindi Dong, Alfio E. Fresta, Andy Grierson, Peter Lippitt, Stefan Kokov
Department of Computer Science
University of York

1) Test Plan Identifier - SEPRADB V3

2) Introduction

The following Master Test plan was created for testing team ADB’s Trains Across Europe (TaxE) game as part of the SEPR module for team EEP in assessment 4 [1]. The plan will target specifically the testing of all modules, classes and elements that affect the TaxE game after implementing the requirements based around assessment 4 [1], but may go beyond this if chosen to based on the correctness and reliability of the previous implementation. The reason for the plan is to verify and validate that the TaxE game works efficiently and correctly for users of the system and for programmers that will be maintaining or implementing features into the game in the distant future.

The testing will involve five levels: Unit testing, Regression testing, Integration testing, System testing and Acceptance testing. Each level is explained further in the Approach section and detailed in each of their respective sections. This enables our plan to follow industrial procedures to create the highest quality of software and test all parts of our system rigorously, ensuring the system is reliable.

The timeline for this project is defined in the project plan [2], starting from 23rd February (officially 18th February but new requirement criteria not given until this date) and will take up to the 29th April 2015 (about 7.5 months). The deadline for this section is 29th February 2015 with enough time to make a presentation for the following week, hence any delays to the project will result in not completing all specified tasks and losing marks for the test plan and the other specified sections. The testing is expected to take up to four weeks and is to be done in parallel with the implementation of the TaxE game requirements stated in assessment 4 [1] and team ADB’s requirements specification. The agile development technique of Scrum will be used for the purpose of sprints to ensure final deadlines are met by each team member being allocated tasks and tracking their progress. Sprints will last for one week each.

The Master Test plan will be following the IEEE 829 format for software and system test documentation [13]. We have chosen to follow this document format due to the document being based on software testing and being of a rigorous industrial standard. As it is a format from IEEE, an established institution for electrical and electronics engineers, it will be of the highest quality and help create a familiar format for readers to understand and easily identify where a piece of information is held.

Ideally, the documentation will be completed before the testing commences to show how our system would ideally be created, however, it is not realistic as the requirements may change overtime and certain documentation can not be verified until after the feature has been implemented or is in the process of being made. Our documentation will hence follow the findings of Parnas and Clements paper [4] in the sense that if some information in the documentation is unknown at the time, then it will be implemented at a later date as, according to their findings, it results in a high quality product still.

3) Test Items

The following is a list of items to be tested

- ADB’s TaxE game v1.2 (the back-end logic of the game, i.e. the model part of MVC)
- ADB’s TaxE game GUI v1.2 (the visuals to the user, i.e. the view part of MVC)
- ADB’s TaxE game interactions v1.2 (the manipulation of inputs, i.e. the controller part of MVC)

4) Software Risk Issues

The following areas within the software are potential risks that may occur:

1. Understanding the libGDX framework and Gradle

   Team ADB had previously used the libGDX framework and Gradle, but nobody in our team has adept knowledge in using these, having only recently used them for the first time in the previous assessment. In the previous assessment we made use of Java SWING. Therefore there could be delays due to our lack of knowledge and our need to spend time familiarising ourselves with the framework. Unless there is available time, we do not plan on making any substantial changes to the GUI that are not absolutely essential for us the meet the requirements of assessment 4. To reduce the chances of this occurring, our team will research into the software further and understand how the previous implementation used libGDX and Gradle and, if it does occur, the highest prioritised tasks will be completed first.

2. Meeting the Requirements of assessment 4

   Assessment 4 [1] indicated that the game must now include a replay system and track modification (in-game) at the bare minimum, with any additional features at the request of the clientele. It is clearly stated that "you must not exceed this brief" [1]. This could be a risk if our team implements the wrong features, going beyond the brief ('scope creep') or not meeting the requirements. The impact would be that the assessment deadline is not met or the required parts are not included, which would mean a lower quality system and dissatisfied clientele, hence resulting in a lower assessment mark. It would also make it less attractive within the final presentation and difficult to improve and maintain in the future if so desired. We will mitigate this by negotiating with the client which of our stated requirements he expects completed by the deadline for assessment 4.

3. Understanding ADB's documentation

   As we are continuing a TaxE game developed by team ADB, we will need to be provided with and understand their documentation. This will outline the architecture, the GUI, game interactions and test plan to enable EEP to plan and extend the game to meet assessment 4's [1] criteria with ease. If team ADB's documentation is not understandable then it would take substantially longer to develop the system to meet assessment 4's [1] requirements. This may delay the project by a few days, resulting in the team becoming more pressured in sticking to the deadline or a lower quality mark. In either case, the most likely result will be losing marks for the assessment. We can lower the chance of this occurring by going through the documentation as a team to enable everyone to discuss the documentation together or by contacting team ADB in the event that something is not understandable to help clarify the documentation. The same can be done in the event there is a delay, but reading at a quicker pace or only looking at the information required to reduce the length of the delay.

4. Clashes with ADB's code (same applies to our developers coding in different branches)

   As we are continuing a TaxE game developed by team ADB, we will need to understand their coding. Their coding would have been implemented in a particular architectural style and in a coding style they are familiar with, hence our team will need to understand this. If we do not, we could end up with high numbers of clashes against their coding, which would take a long period of time to fix and delay the project. Technically, another architectural style could be implemented to suit our team, however, this would take a considerable amount of time to refactor the entire code, which we do not have, and would be similar to coding the system from scratch. To reduce the chances of this occurring, we can spend a day understanding the code and/or request a programmer from team ADB to assist us before extending the code base. If a delay does happen, then the team can request a team member from ADB to assist in understanding the problem or by simply trying to fix the problem by researching into the error and fixing it as quickly as possible.

5) Features to be Tested

The following is a list of features to be focused on during testing. These are features are necessary for the game requirements covered by assessment 4 [1] to be met. They can also be found in the changes log [Appendix K]

- Newly implemented features coincide with old system

- Replay system - User Req Replay.1.1 , User Req Replay.1.2 , User UI Req Replay.4.1 , User UI Req Replay.4.2 , Func. Req Replay.1.1 , Func. Req Replay.1.2 , Func. Req Replay.1.3 , Func. Req Replay.1.4

- Track Replacement - User Req Modify.2.1 , User Req Modify.2.2 , User Req Modify.2.3 , User UI Req Modify.5.1 , User UI Req Modify.5.2 , User UI Req Modify.5.3 , Func. Req Modify.2.1 , Func. Req Modify.2.2 , Func. Req Modify.2.3 , Func. Req Modify.2.4

- Any further features the development team wishes to implement (beyond the requirements and specification, but have been agreed to by the clientele) - User Req CP.3.1 , User UI Req CP.6.1 , Func. Req CP.3.1

6) Features not to be Tested

The following is a list of the features that will not be tested. These are mainly the requirements not stated in section 5 as there are too many requirements to list from team ADB's requirements specification (contained in Appendix) [7]. System requirements have been stated though:

- Checking the user has the appropriate system specifications and software to run the game.

  The user manual will detail which software and system specifications will be required to run the TaxE game. The user will be responsible for ensuring their system can meet the software and system specifications listed in the manual as the developers will have no control over their system specifications. The developer can only keep these specifications as low as possible to ensure users can meet it without purchasing a new system, expanding the target audience range.

- Any feature which is not specified in the assessment 4 [1] requirements

  Assessment 4 [1] stated certain features that should be included at this stage of the software development, covered in the previous section. It is therefore not required and not the team's responsibility to go beyond the requirements, however, if time allows it then extra features will be implemented based on the priority level of the client's requirements and the requirements specification first defined by team FVS in assessment 1, team FVS in assessment 2, ADB in assessment 3 and ADB in assessment 1. Unfortunately, we could not gain access to the original requirements specification of FVS for assessment 1 and FVS's assessment 2 and ADB's assessment 3 did not contain any formal documentation about new requirements, hence the requirement identifiers of the features not to include are not written down. For any additional features, the team went straight through the clientele to prioritise what requirements should be added and included or if it should not be added at all to ensure the clientele were satisfied with their game and defined according to them. The requirement identifiers for assessment 3 features could not be used as the team could not retrieve ADB's or FVS's original requirement specifications and may have changed since then anyway.

- Features defined in assessment 3 [10]

  Assessment 3 stated that specific features were required to be implemented before this software development stage. Providing all stated features were implemented and no tests had failed, then the previous work does not need to be altered or touched in any way because it would fully function as intended. In the event that a previous feature provides a challenge when implementing a current feature, the team will document, alter and test the feature as required to provide future development teams with a note as to how the software has changed over time. The requirement identifiers for assessment 3 features could not be used as the team could not retrieve ADB's or FVS's original requirement specifications and may have changed since then anyway.

7) Approach

7.1 - IDE Choice: IntelliJ or Eclipse

To code and test the system effectively in Java, a suitable Java IDE (Integrated Development Environment) was required. It was recommended to all use the same IDE as there would be a reduced chance in any errors occurring when pushing and pulling branches to and from GitHub. There were two options to choose from: return to using eclipse as our team did in Assessment 2 or carry on using IntelliJ as our team did in Assessment 3.

IntelliJ IDEA is a Java integrated development environment (IDE) by JetBrains. The previous development team used IntelliJ to create and modify their Java project through the use of Gradle (open source build automation system) while allowing easier management of branches with our git repository (version control). In response to this, our team decided that the best option would be to continue using IntelliJ as it would mean less confusion when trying to switch the project to another IDE and was recommended by our lead programmers as a higher quality developer environment that was more efficient in managing our GitHub repository. Initially, this seemed a good decision, and perhaps still good, however, there were a number of problems that occurred during the installation of IntelliJ. For example, IntelliJ required Gradle to be downloaded and linked to the GitHub cloned project, but some developers had issues where it could not be found. There are many more issues such as the Java SDK (source development kit) not linking in correctly, time being scheduled to train in using IntelliJ and having to use our own laptop systems because it is not installed on the computer science laboratory computers (and requires you to have a nearly empty user area to have it installed). This has impacted our remaining time as many developers have been delayed in completing their tasks, which may result in a lower quality end product for the client. In response to this, our team has altered the scrum and coding deadlines and distributed our resources differently to accommodate this (i.e. project leader is now dedicated to the documentation and will help in coding when required rather than the other way round).
Eclipse is another IDE by the Eclipse Foundation which can run many programming languages, in this case it is Java. Eclipse was considered as another IDE option as our group used it in the previous development stages, hence we are highly familiar with its environment.This means that less time would be required in training the developers to use Eclipse (not none as they will need training and time to get used to the new project and any additional add ons that require downloading, such as Gradle). Eclipse can be linked with GitHub for managing versions and branches between team members in a more efficient style, but other IDEs can perform this too. Eclipse is also available on all computer science laboratory computers, which comes in handy for increasing our available resources by the availability of the IDE in the event that a team member may not have access to their own computer system. The problem with Eclipse is that some functions take longer to perform in comparison to IntelliJ, such as switching between repository branches. The time saved would be negligible, but can build up and help enable the team to have more time in allocating it and resources elsewhere to reach the deadline.
Overall, each of the two IDEs can fulfil our desired programming requirements, hence our team has decided that each member may use either of the IDEs. IntelliJ was recommended as, on paper, it is a much better IDE than Eclipse, however, people have had more experience with Eclipse and is installed on all the computer science laboratory computers. It does not matter what IDE our team members uses as long as the work is completed on time and they are comfortable on what they are working on.

7.2. Overall approach - Test-Driven Development

In test-driven development, test cases were formed from the requirements before the creation of source code to exercise the interface, finding errors in the functionality and data structures of these components [9]. It started by creating a test case that attempted to make the code fail, writing new code to satisfy the test and then executing the new test and any previously existing ones. If the tests passed, a new test case was written for the next piece of code and so on until the component was completed and all tests were satisfied. If the test failed by finding an error (after checking that the test was correctly written and not producing an incorrect value), then the code was refactored (corrected) and all tests that were created to the current point were rerun [9]. Each iteration here had one or more tests associated with the code, which were added to a regression test suite to validate existing functionality as well as new functionality. Code was written in small components, allowing the production of an extensively tested and working piece of code before it was merged into the master branch to stop bugs following through into more complex and larger coding areas. Unit tests were conducted by the developer for their respective code because they would understand it best) and each documented it as part of the testing evidence, which allowed the team to manage and verify what features were working and what features needed fixing. These tests were reviewed by the project leader who consistently checking the progress of each developer to ensure all were participating fairly, work was distributed fairly and that everyone had a role to make use of the team resources available. Additional support (personnel) was allocated to a developer if required. Test cases were based on reading team ADB's and FVS's previous documentation and new requirements of the system (given by the clientele) to understand what tests the team had expected to include and test in the system. The team decided to continue with the test-driven approach as it worked effectively and efficiently in the previous development stages and seemed to come naturally to all the developers again, hence the transition of the method onto a different project was minimal and did not require additional training or time to adjust to. The development method also helped to improve the overall quality of the code by locating and fixing the majority of errors during the coding stages in separate classes before they were merged due to being a smaller code base each time to search through, otherwise it would have wasted time unnecessarily. It could also have cascaded the errors through to cause errors in other classes that may have been previously functioning correctly, which would have made it considerably harder to find the error. The test-driven development was applied throughout the entirety of this stage while coding.

7.3 Testing Levels

The testing for the project will be split into five parts: Unit testing, Regression testing, Integration testing, System testing and Acceptance testing. The tests will be conducted by each member of the SEPR team, however, the lead programmers will be the main contributors towards the testing. The team will follow a test-driven development approach where each developer will create unit test cases before coding, write their piece of code, conduct their unit test cases and then implement the necessary improvements if required on flagged coding.

Before initial testing commences, ADB’s previous implementation will be refactored to create a higher quality system by improving efficiency (both memory and time), security, readability and maintainability if the code in question if possible. The refactoring period will act as the first scrum sprint of the assessment, lasting one week in which all, or most, of the refactoring required will be completed. Each developer will be working on the refactoring task. The workload and time frame spent will be based upon the coding quality of both FSV and ADB’s work (FSV’s assessment 2 and ADB’s assessment 3 implementation) as these projects were used in creating the current implementation. If there is too much refactoring to complete, then the team will reallocate time and resources to start the essential requirements where all refactoring will stop, unless a developer is close to finishing a class of code. If there is very little refactoring to complete, then work on the essential requirements will start earlier to ensure the final deadline is met.

Unit testing will be conducted by the developers of the unit on separate Git branches and will report back to the rest of the team with further assistance from the lead programmers if required. The developers will follow a white-box testing approach to ensure that paths in the unit link correctly and will provide proof of unit testing in the form of a test case list with evidence of the output, any errors and the fixed code if applicable. This will be split further into functional and non-functional testing to distinguish the requirements apart. All possible data entry types will be used if applicable to the unit. Each developer will use JUnit to test their coding. All unit testing information will be reported to the team leader before moving onto the system/integration testing. Unit testing is done to ensure all code is functioning to its intended purpose with no errors. Refactoring of code (i.e. improving the internal structure of code without altering the external behaviour) will occur after each unit test providing time allows and is necessary. The aim is to improve the efficiency, readability and maintainability of the code base, enabling future development teams who take on our project to understand it quicker. This would also make a strong impression during the final presentation. The danger here is that making any further refactoring may break existing, functioning code and may not be required if the team produces high quality code first time around.

Regression testing will involve locating bugs in existing areas of a system after implementing a change in some form. These may be functional or non-functional areas of the system, reflecting on past functional and non-functional requirements of the system. There is no distinct minimum number of tests to perform each time, rather it is based on the number of old unit tests added to the number of new unit tests required to validate the new function and the integration. Regression testing will be performed by the developer of each feature after creating the feature by running the old and new unit test cases against the recently installed feature. This testing level will be continuous throughout the project from after the unit testing level and onwards as modifications will be made throughout the entire development cycle. If an old unit test returns false to show it has failed, then the developer will need to fix the bug and repeat the process of running all the unit tests again before progressing onto the next testing stage. If an old unit test returns true, the developer can note down that the feature has passed the regression testing stage. The process repeats until all new features at this development stage have passed the regression testing level, upon which integration testing can start. The aim here is to discover any new faults in other areas of the system based on changes in one particular area, e.g. a new feature such as the track modification being added or a change to existing code to make it more efficient. This will reduce the chance of errors being carrying forward into the later testing levels where it should have been caught earlier on to aid in tracking and debugging errors. As a result, the system will be more efficient and maintainable for future developments or as a final product, hence the software quality will be increased.

Integration testing will occur after the unit testing, combining set of modules/features and testing that each links successfully without flagging any problems. This will be done by making a pull request on a fully functioning and tested Git branch and merging it with the master branch, whereby Travis-CI [5] will help, flagging issues that occur. Integration will be completed by the lead programmers using a white-box testing approach and by using regressing testing. The integrations will be documented by noting the name branch to be merged, whether the new features have been implemented, whether the new unit tests were all written and if all unit tests have been passed before making the merge. The merge will also be noted as to whether it was successful or not (with the errors) to show that there are still issues to be resolved before merging. There is no distinct minimum number of tests to perform each time, rather it is based on the number of old unit tests added to the number of new unit tests required to validate the new function and the integration. Any additional errors discovered at this stage will be immediately flagged by the lead programmers, who will fix it themselves or direct it to other developers. Elements of black-box testing will be used to test GUI features and how they are displayed on screen when linked with particular units to get a feel of what the user would see. This will help improve the GUI to be of a professional quality while ensuring that information is portrayed correctly to the user, otherwise it may discourage users.

System testing will be performed when all Git branches have been merged together into the master branch. This will be done by the lead programmers and project leader with the assistance of all developers if required. The method will be conducted using a black-box testing approach (developer perspective in relation to system requirements to implement) and using the GUI to validate the software after the integration testing. This stage will be entered when all unit and integration testing is completed and no more errors are discovered. There should be no critical errors entering this stage because of the previous levels of testing. In the event that a critical error is observed, it will be flagged and noted by the lead programmers who will deal with the error. If a backlog of errors occurs, i.e. more than 1, additional developers will be utilised.

Acceptance testing will be completed from the potential end users (other SEPR students) and the clienteles perspective as a form of formal testing to determine whether they accept the system [8]. Acceptance testing is to verify whether or not the requirements have been met, i.e. those found in the assessment 4 briefing [1]. The testing will involve a grey-box approach where the potential end users and the client will be under the supervision of the EEP team to conduct a black-box approach (user/clientele perspective in relation to new scenario cases) and the developers taking a white-box approach. New scenario cases were created as team ADB's originally had use cases did not cover any of the clientele requirements of assessment 4, hence it would be very hard for us to complete the acceptance testing stage. Scenarios were used in place of use cases as they are better suited towards the agile development process as, based on our time schedule, are quicker to implement and easy to adapt to the clientele's potentially volatile requirements. User's experiences will be compared to scenarios we will write to test new functionality we are implementing.. No major errors should be entering this testing stage. Any major errors discovered at this stage will be immediately flagged and fixed by the lead programmers. If a backlog of errors occurs, i.e. more than 1, additional developers will be utilised.

After acceptance testing, the program should contain no more major bugs. One final test will be conducted by the team leader and, if no more errors are discovered, can be distributed appropriately.

7.4 Change Levels

The testing for the project at this development stage will be split into three parts based upon the type of changes made: Perfective changes, Corrective changes and Additive changes. The changes will be conducted by each member of SEPR EEP's team, noting down who made the change as they would understand the changes they made and be able to explain it in an understanding manner. The team will follow a test-driven development approach where each developer will create their unit tests, write their piece of code, conduct the unit test cases on the code and then implement the necessary improvements if required on flagged coding. This will enable the program to be consistently tested against the current JUnit test cases previously implemented and those that are newly implemented by our development team.

Perfective changes are defined as changes that optimise the system, improving the efficiency of the previous version code (space and time) and the architecture of the system, making it more understandable and simpler to follow . Ultimately, these will provide a higher quality system for the client, leading to more users, and help future development teams maintain the system. The changes will be tested by the developers who made the change, reporting back to the rest of the team whether they passed or if errors were found, in which case it will be flagged and noted in an error log. The reason for this is to document all errors that were fixed in the event that if the same error occurred, it could be fixed quickly and be tracked to ensure that all errors are fixed before the deadline date, increasing the quality of the software. The developers will follow a white-box testing approach to ensure that modules in the changes still link correctly without any errors. Developers will provide proof of the testing in the form of a test case list with evidence of the output, any errors and the fixed code if applicable. All possible data entry types will be used if applicable to the test to ensure all possible entries are considered. Each developer will use JUnit to test their coding because the development team has a good understanding of JUnit and can be an extended plugin used with Eclipse or IntelliJ. All testing information will be reported to the team leader before moving onto another feature. Refactoring of code (i.e. improving the internal structure of code without altering the external behaviour) will occur after each test case to improve readability and maintainability of code to enable future development teams who take on our project to understand it quicker.

Corrective changes are defined as changes that fix faults and errors within the current version of the program, most likely stemming from previous versions of the system. As a result, the program will contain few, if any, errors and result in a higher quality system to present to the client, which may increase the potential users of the system. It will be tested by the developers who made the change, reporting back to the rest of the team whether they passed or if errors were found, in which case it will be flagged and noted in an error log. The reason for this is to document all errors that were fixed in the event that if the same error occurred, it could be fixed quickly and be tracked to ensure that all errors are fixed before the deadline date, increasing the quality of the software.The developers will follow a white-box testing approach to ensure that modules in the changes still link correctly without any errors. Developers will provide proof of testing in the form of a test case list with evidence of the output, any errors and the fixed code if applicable. All possible data entry types will be used if applicable to the test to ensure all possible entries are considered. Each developer will use JUnit to test their coding because the development team has a good understanding of JUnit and can be an extended plugin used with Eclipse or IntelliJ. All testing information will be reported to the team leader before moving onto another feature. Refactoring of code (i.e. improving the internal structure of code without altering the external behaviour) will occur after each test case to improve readability and maintainability of code to enable future development teams who take on our project to understand it quicker.

Additive changes concentrate on changes that add new functionality or features to the system. In this development stage, the main additive changes will focus on implementing the main requirements stated in assessment 4 as these must be implemented. Any spare time and resources will be allocated to making additional features after all the main documentation has been implemented to verify the minimum documentation is completed by the deadline. As a result, the system will provide a higher gameplay experience for users while meeting the client's requirements, resulting in higher satisfaction for both the client and users. The reason for this is to document all errors that were fixed in the event that if the same error occurred, it could be fixed quickly and be tracked to ensure that all errors are fixed before the deadline date, increasing the quality of the software.The developers will follow a white-box testing approach to ensure that modules in the changes still link correctly without any errors. Developers will provide proof of testing in the form of a test case list with evidence of the output, any errors and the fixed code if applicable. All possible data entry types will be used if applicable to the test to ensure all possible entries are considered. Each developer will use JUnit to test their coding because the development team has a good understanding of JUnit and can be an extended plugin used with Eclipse or IntelliJ. All testing information will be reported to the team leader before moving onto another feature. Refactoring of code (i.e. improving the internal structure of code without altering the external behaviour) will occur after each test case to improve readability and maintainability of code to enable future development teams who take on our project to understand it quicker.

7.5 Configuration Management/Control Management

Configuration management is focused on the maintainability of the code while tracking and controlling changes in the software, making sure it is traceable. This will be done by using GitHub to store the latest and previous versions of the system through commits made by the development team. The configuration management team will verify that all developers are working on the latest version of the system to stop the team wasting resources (developers and time) on older problems that may have been fixed in the latest version. This can be done by asking the developers what version of the system they are using.

The configuration management team will report the changes made per commit to the project leader and will receive a report from the developers to indicate each stage of the testing being completed. This will enable easier traceability of problems solved and the unit tests completed to meet their passing criteria which can be shown through a traceability matrix. If all the testing has been verified with no errors, the system can move onto the next phase of testing to stop errors concatenating through. The configuration management team will indicate this change by switching the unit testing status to complete from incomplete and inform the project leader and development team. The process will repeat until all phases are completed in which the configuration management team will conduct a final review to check all documentation describes the system and versions appropriately.

7.6 Test Tools

The following test tools will be used:

1. **JUnit**
   JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks [12] and is included in both IntelliJ and Eclipse. JUnit will enable the team to create unit test cases for the modifications made in this development stage, testing whether or not our modifications are successfully integrated into the existing system or not. The framework will locate the errors by indicating which tests fail and flag them to allow us to log them in a table and get round to fixing before the final deadline. This will improve the overall quality of the system for the clientele, which should result in an increased user gameplay experience by containing less bugs. All JUnit tests created can remain within the code and run simultaneously when executed to ensure the system is fully functional. JUnit is our prefered method of testing the system as the team has used JUnit in the previous development stage, therefore the team are highly experienced and comfortable in using it which will help keep confidence high and reduce any additional training that what is necessary (time and resources can be allocated elsewhere). In addition, team ADB used JUnit to conduct their testing, so using JUnit will help to keep a familiar environment for both the system and team ADB incase they take on the project in the final development stage. JUnit tests will be conducted by each developer on their own piece of coding as they will understand their coding more and will want to constantly test their code to ensure it is bug-free and mergeable. This will be used to support the test driven development method adopted by the team.

2. **Eclipse and IntelliJ programming environments**
   Developers require a Java programming environment for programming in a more efficient manner for editing, compiling and debugging Java code. More information can be found in the extension report [6]

3. **Travis-CI**
   Integration testing will be aided by the continuous integration service Travis-CI [5] that is embedded into our repository to test if the integration between branches can occur. It is used for testing and building projects in Github [5] and will help ease integration testing by verifying if the project can be built when a commit has been made where, if it fails, will not merge the branches together. This will be reported in a desired outcome (e.g. email) and shown in the GitHub repository with a pass/fail, indicating how the merge failed by flagging the issues. We are using Travis-CI because the service will help to prevent any errors being merged from one branch to another that have not been extensively tested which may otherwise harm the GitHub repository, causing it to break. If this happened, the team would have to spend unnecessary amounts of time fixing the problem which should not be happening, especially when our deadline is as short as it is. It also provides evidence for our integration testing for it to be documented.

4. **JavaDocs**
   Javadoc is a tool for generating API documentation in HTML format from doc comments in source code [4]. The primary use is to clarify the meaning behind the code for developers of the system to make the program easier to understand without spending large amounts of time. For example, explaining what a method does and the parameters that need passing to show developers what they need to call if it is already implemented rather than creating an entirely new method. Producing a JavaDoc will create a reference point for any queries a developer may have about the code in the event that they need support without calling on support of the previous developers as they might not be available to contact. The team and team ADB have used JavaDoc in the previous development stage to great effect to improve the readability and maintainability of the code while also being familiar with the tool, hence it was decided to continue using JavaDoc and simply extend the existing JavaDoc by team ADB. JavaDocs will be maintained by our one of the lead programmers, but developers will be responsible for documenting their own code to save time for the lead programmer and as the developer will understand their coding more.

7.7 Meetings
By following the Scrum methodology, the SEPR team will perform sprints by scheduling meetings at least once per week up until the deadline to identify and report any errors in addition to the overall progress of the testing. More meetings may be scheduled closer to the deadline if required and additional meetings may be called upon in the event of an emergency. Testing will be continuous through the entirety of assessment 4.

7.8 Measures and Metrics

The developers of each unit will collect information from test cases regarding their unit testing. These will be documented and reported back to the team at the end of a group development session (around 4 hours) or on a weekly basis. The following information will be shown in a table format:

1. Test ID (format u.t where u is number of unit and t is number of test)
2. Location within code (module)
3. Description of test conducted (purpose, test data, expected result)
4. The requirement that the test case relates to
5. Category of test case
6. Author of test
7. Status of test (pass/fail)

Evidence of appropriate testing will be provided per test case and a traceability matrix will be provided to show the relation between the requirement and the test case.

8) Item Pass/Fail Criteria

The Master test plan will be completed once all levels of testing are completed without detecting any errors and all errors detected have been fixed. Each test case will be completed if it passes the specified criteria set without detecting an error. Once all developers have completed their unit testing, the developer will report to the team leader and distribute to another task accordingly, such as another developer requiring help with their test cases.

The test plan will be finished when all unit testing, integration testing, system testing and acceptance testing have been completed without any bugs. If the deadline is within a day or two from being met, a few minor errors may be left in the final implementation code.

9) Suspension Criteria and Resumption Requirements

If any of the fundamental game components (e.g. completing a goal) do not pass their unit tests, i.e. are not functioning, then the development team will not attempt to integrate the units together until they are fully functioning as separate units. To reduce the risk of a potential delay occurring to the project, the development team will distribute another member of the team to work on the unit that contains the problem if other units depend on it.

10) Test Deliverables

- Traceability matrix (relation between tests and requirements)
- Unit test cases (will be found in code)
- Unit test logs
- Integration test plan
- System test plan
- Acceptance test plan
- The Master Test plan as a combination of the unit, integration, system and acceptance test plans

11) Remaining Test Tasks

The remaining test tasks will be completed by a third party, i.e. another SEPR team, that decides to take on our project in the next assessment task. This will create a new version of the test plan that covers the specified areas in assessment 4 [1], however, we will have to replicate the same procedure on the next assessed task too for another group.

12) Environmental Needs

The following elements are required to support the testing phase of the project:

1. Installed versions of Eclipse or IntelliJ to run Java on all developer machines. We will use the Luna version of Eclipse, as this is the version installed on PCs in the software lab, and/or the IntelliJ IDEA Community Edition 14.0.3 as it is the latest version and should contain less bugs.

2. Installed JUnit in the Eclipse and/or IntelliJ environment on all developer machines.

3. Possible end users, as part of acceptance testing

13) Staffing and Training Needs

There are some team members who have not programmed in Java before, hence a crash course in Java will be required. The training will include the fundamentals of the language and any additional libraries or softwares that may not have been used before, such as libGDX for the GUI, and will take up to three days to complete or a total of 15 hours. This timeframe is specified to provide enough Java experience for team members to start programming the game while gaining further experience through implementation.

As all developers will be conducting their own unit tests, each developer will need to be trained in using the JUnit framework to write test cases in an effective manner for maintainability of the software. In the event that one or more developers are unable to be present, the project leader will assume the role and continue with their work in their absence. The training will take one day.

If a developer has requested the assistance of another developer, then the other developer will need a brief overview of the problem and the unit, which may take up to thirty minutes.

All team members will need to be trained on how to use the software prior to the deadline date. This will enable each team member to check the quality of all coding and documentation before being submitted.

14) Completeness/Coverage Criteria

As a team decision, the test plan will mainly cover the extent of features demanded by the clientele of assessment 4 and the features to be tested (section 5), which includes the replay system, the track modification system and any additional features we implemented, i.e. the cross-platform and portable capabilities of the game into the Android and iOS operating systems. This coverage includes any modules and classes which will directly impact in how the features are implemented. The reason is to meet only the requirements that were stated by the clientele as they did not ask for anything else, however, testing other components previously implemented will be conducted to ensure previous functionality does not intervene or disrupt new functionality. Adding additional features may over complicate the system, cause the team to get distracted from the clientele's requirements and increase the chances of the final product containing bugs, which would reduce the overall quality of the system and not fit the needs of the clientele. If there are some noticeable small areas that could be improved and are not related to the requirements of assessment 4, then the developer may spend up to one day sorting the problem out, otherwise they may get distracted for a long period of time from the main goal. If time allows or spare resources are available, a developer may implement extra features if the clientele demands it.

The test plan will be completed once it has been approved by the clientele and development team and when acceptance testing has been completed. Acceptance testing is completed when all previous levels of testing have been completed, when team ADB's, team FVS's and our team's (EEP) requirements that relate to assessment 4 have been completed and satisfy their use cases and finally when all of the original clientele's requirements (located at the very end of the assessment briefing [1] document) have been met. This will ensure the system is fully operational and fits the clientele's needs as best it can to provide a high quality, fit for purpose system for users to enjoy. It will also indicate to future developers that the software is available to them at its peak quality in terms of coding efficiency, readability, maintainability, security and memory size.

15) Responsibilities

Task	Developers	Lead Programmers	Configuration Management	Project Leader	End User / Client
Unit Testing Documentation and Execution	X	X	X	X	
Regression Testing Documentation and Execution	X	X	X	X	
Integration Documentation and Execution	X	X	X		
System Documentation and Execution	X	X	X		
Acceptance Documentation and Execution	X	X	X		X
Final Test		X			X
Verify final versions of documents	X	X	X	X	
Submit the documents				X	

The project leader and head programmers will ensure all documentation is completed, with the review of the documents being completed by all team members.

The head programmers will verify the software with a final test to check the coding for any final errors before being submitted. The client, if available, will be allowed to use the software as part of the final test to see if any additional errors are discovered and that the system meets their requirements.

The project leader will submit the final documents to the client for the assessment.

16) Schedule

Time had been allocated appropriately in the first assessment task [1] using a gantt chart to track the progress of our team and ensure the final deadline is met. Each task has had a team member allocated and their responsibilities have been recorded so that there are no wasted resources otherwise a delay may occur. These allocations are made available on GitHub, where assistance can be offered if a member needs support on their task.The whole testing process should take four weeks, leaving at least a day free before the deadline of 17th February for a final review for any poor coding or code documentation. Any communication of tasks are handled by the project leader and head programmer with the assistance of the collaboration and version control tool GitHub.

17) Planning Risks and Contingencies

There are a list of potential risks in the Risk Assessment document for Assessment 1 [3] [11], however, the following risks have more of an emphasis on the testing process:

- Team members have gained experience in creating unit tests from the previous assessment, but there still remains the risk that tests are poorly written. The consequence of this could be tests being written that do not detect major problems with the functionality they are testing. This will lead to much bigger problems at a later stage of development when these major problems become apparent. This can be mitigated by:
    - Ensuring sufficient time is allocated to teaching/building experience in using JUnit such that all programmers feel confident in efficiently using it.
    - Ensuring all unit tests are regularly shared using our version control system and a testing log is also kept so all members can scrutinise these tests.
    - A traceability matrix will help to ensure that there exists tests for all functional requirements that are being implemented.
- Time allowed for development and testing of the game is constrained by the assessment deadline. Therefore there exists a risk that testing begins to be rushed or ignored due to development running over-schedule. The possible consequence of this is that the game will be released with major defects and problems that were not detected by previous testing. This can be mitigated by:
    - Creating a schedule that is flexible and allows for tasks to take longer than originally expected will reduce the risk that not enough time is allowed for testing.
    - Our test-driven development approach means that much development is done along-side testing, rather than testing being done last minute.
    - All requirements have been given a priority level (Essential, Preferable, Optional), therefore should development time become sparse, the project leader will decide to exclude less crucial requirements from the working implementation so that more time can be allocated towards ensuring essential requirements have been tested.
        - Essential requirements are those that must be completed by the deadline to meet clientele's original set requirements;
        - Preferable requirements are those that would be advised to complete by the deadline to improve the product as a whole (requirements that may have been asked by the clientele later on or recommended by the clientele after being put forward by our team);
        - Optional requirements are those that do not really need to be implemented, but may add some kind of extra dimension or new feature to the game that are not essential towards the core gameplay. These requirements may have been asked by the clientele or our team but may have been asked not to prioritise by the clientele or the development team have said to the clientele that it would take too much time to implement or there are not enough resources available to attempt the requirement.

18) Approvals

Name	Main Role	Signature
Richard Paige	Client and direct stakeholder	
Fiona Polack	Client	
Tim Kelly	Client	
Andrew Grierson	Project Leader	
Alfio Fresta	Technical Lead and Lead Programmer	
Stefan Kokov	Designer and Configuration Management	
Peter Lippitt	Lead Programmer	
Richard Cosgrove	Quality Assurance	
Yindi Dong	Lead Artist	

19) References

1. Assessment 4 briefing
2. Team EEP's Project Plan
3. Team ADB's Risk Assessment
4. D. Parnas, P. Clements, "A Rational Design Process: How and Why to Fake it", IEEE Transactions on Software Engineering, 1986, pp. 251-257
5. Travis-CI, "Travis-CI - Free Hosted Continuous Integration Platform for the Open Source Community", Internet: https://travis-ci.org/recent , [Feb. 13th 2015]
6. Team EEP's extension report
7. Team ADB's Requirements Specification
8. IEEE, "610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology", Internet: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=159342 , [Feb. 13th 2015]
9. S. R. Pressman, Software Engineering: A practitioner's approach, Boston ; London : McGraw-Hill Higher Education, 7th edition, 2010.
10. Assessment 3 briefing
11. Team EEP's Risk Assessment
12. JUnit, "About JUnit", Internet: http://junit.org/ . [Feb. 12th 2015]
13. IEEE, "IEEE Standard for Software and System Test Documentation", Internet: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4578271 , [Nov. 25th 2014]