Testing Document

TEAM HEC

1. Introduction

1.1 Overview

This document outlines the testing strategy for the first stage of development of Locomotion Commotion. It includes descriptions of all of the automated unit tests, our interface testing strategy and our requirements verification testing. If you have taken on our code you should easily be able to replicate our results.

1.2 People

Unit Testing was predominantly done by Callum Hewitt, the project manager. However, for more specialised parts of the code unit tests were created by their respective authors, including Matthew Taylor, Elliot Bray and Sam Anderson.

Interface testing was generally informal with final checks devised towards the end of the project to ensure continued quality. This document defines exactly how the interface should work and some tests that can be performed for each aspect of to ensure it still works correctly. These inspection tests were performed by Elliot Bray and Rob Precious.

Our verification tests were done via inspection and are used to ensure that our Game delivers on the user requirements. Any requirements which haven't been delivered in this iteration have been justified in the architecture document.

1.3 Testing History of the Project

For about 4 weeks during the project we had a problem where we could not run any tests. This lead to some unstable development, particularly on the backend whilst the error was being tracked down. The problem was eventually solved by mocking OpenGL and other LibGDX features in a class "GdxTestRunner". The solution was found online. Read carefully the section about adding new tests to see how to avoid this problem for new tests.

1.4 Testing Quality

It is impossible to test every possible scenario and some parts of our code were impossible to test at all, mainly due to continued problems with mocking graphics libraries. However, as a team we feel that our tests cover as much code as is necessary to be confident that the software will perform as expected. Our inspection tests backup our unit tests well and should cover most problems.

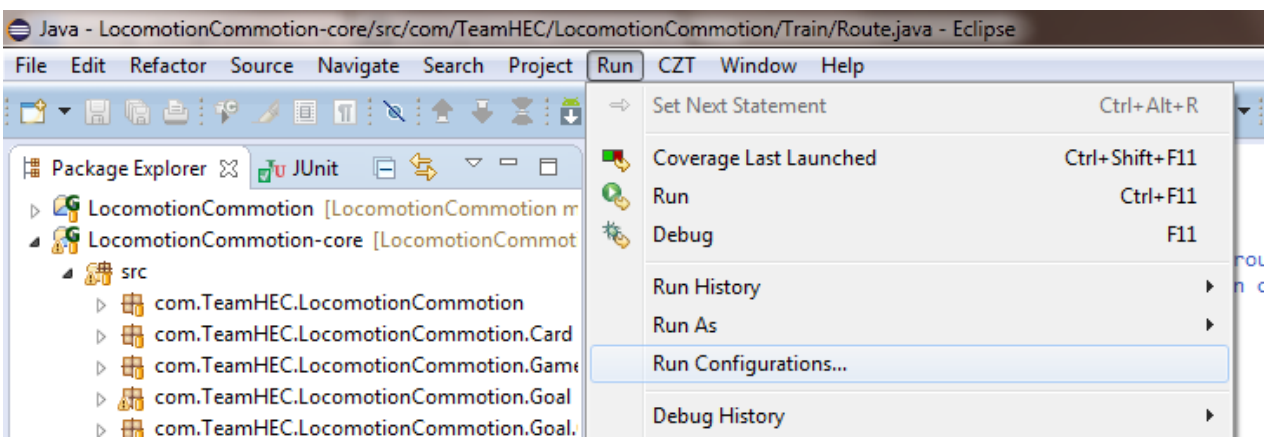
2. Test Procedures

2.1 Test Bed Setup

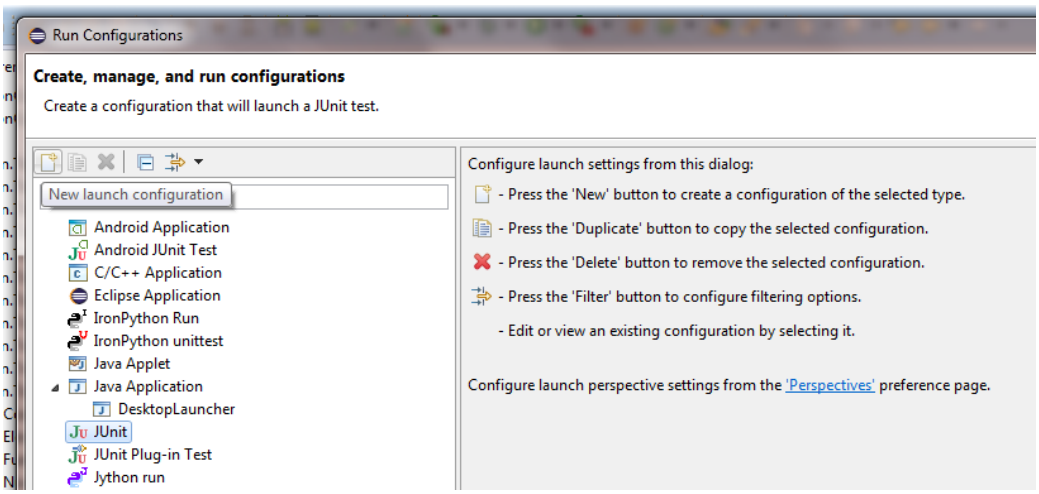
All tests for our code are declared in the same package as the code they test. However, the files are stored in the testsrc folder, which can be seen in the package explorer under LocomotionCommotion-core.

To set up the test bed you will first need to make sure you have loaded the project correctly. Visit this wiki or read the user manual to see how to get started with the Locomotion Commotion repository. After Eclipse has been set up follow these steps:

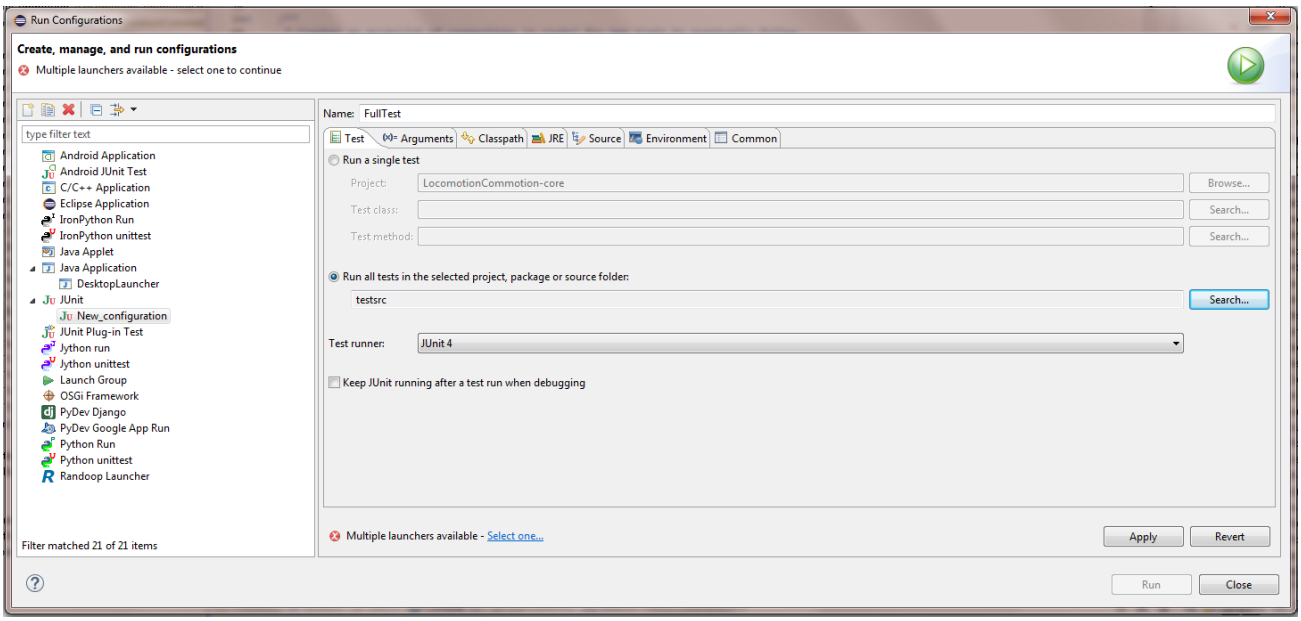
1) Select Run -> Run Configurations…



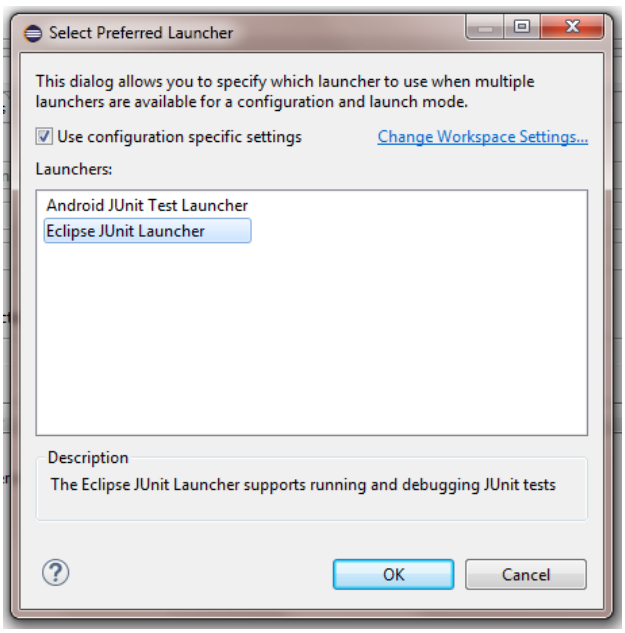
2) Select JUnit from the left hand list and then select the new launch configuration button at the top.



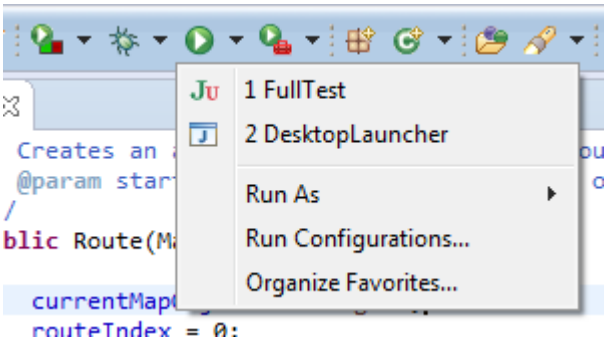
3) In the new launch configuration window. Select the Test Runner to be JUnit 4 and the click “Run all tests in the selected project, package or source folder”. Press “Search…” and look for the testsrc source folder. You can rename this configuration to FullTest at the top.



4) At the bottom you might see an error saying “Multiple launchers available”. Click “Select one...”, Eclipse JUnit launcher and then OK.



5) Click Run to run the configuration. The configuration should now be saved in the quick run menu so you can use it at any time.

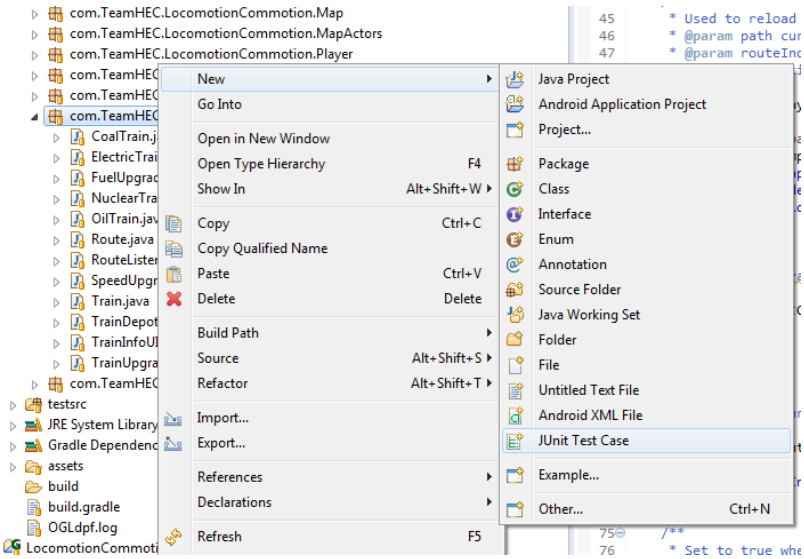


6) Add more configurations for individual test packages by adding new run configurations in the same way and change “testsrc” to one of the individual test packages in the testsrc folder.

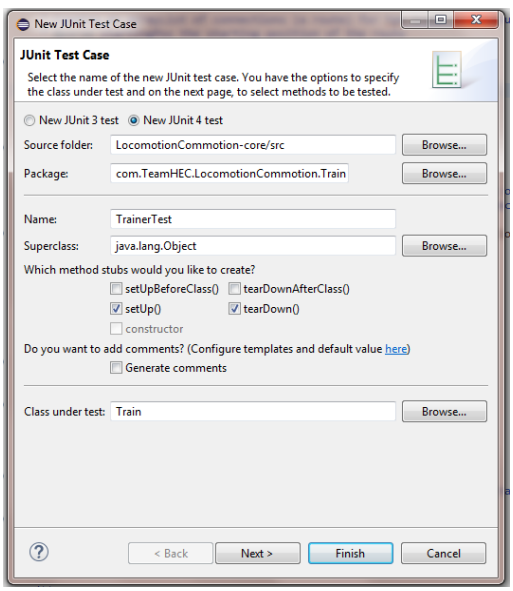
2.2 Adding new tests

When writing new tests ensure that you create a package in testsrc of the same name as the package the class you want to test is under.

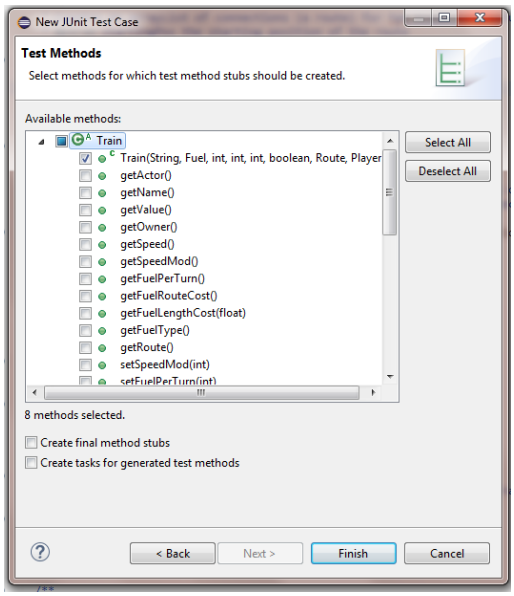
1) Right click the package the testcase needs to be part of in Package Explorer. Select New -> JUnit Test Case



2) Name your test class using the convention: “ClassToTest” + “test”. Use the browse button next to “Class under test:” to find the class you want to test. Then press next.



3) Select the methods you want to test and then click Finish. You shouldn't need to test any getters or setters.



2.3 Mocking

To run new tests you will need to run them with the GdxTestRunner class. This initialises OpenGL and libGDX features correctly as well as the file system. Tests will fail without this class!

To run with GdxTestRunner make sure that your test class has imported the following:

org.junit.runner.RunWith;
com.TeamHEC.LocomotionCommotion.Mocking.GdxTestRunner;

Then above the class declaration add: @RunWith(GdxTesrtRunner.class). This should fix most problems with graphics during testing. However, some classes which instantiate more complex UI structures won't be able to be tested, even with this solution. These should be tested using Inspection.

```
22
23 @RunWith(GdxTestRunner.class)
24 public class CardFactoryTest {
25
```

3. Unit Testing

3.1 Introduction

Unit tests are the main testing method for the back end. We used JUnit to create dynamic tests which execute code in a testing environment. We test that outputs are generated correctly for certain methods (Black Box) and that certain methods change their objects in the right way (White Box) . These testing types are distinguished in the table below. All of the assertions have an error message set to print if it a test fails. This helps us find testing problems more quickly.

The unit tests are all functional tests and test actions of objects as opposed to non-functional tests which check how something feels and are more closely aligned to user experience. All of these tests can be run very quickly and are designed for regression testing. They should be run after significant changes are made to the codebase, ensuring that the project is still stable. If they fail then you know that your recent changes have broken some features.

The testing approach for unit tests has been to add tests to methods which could break easily. For tests with inputs we have only tested valid inputs. Invalid cases have not been considered. This is due to time constraints on tests. We felt that the methods we test should never be in a situation that they receive invalid values. So, at least for now, it was better to direct our attention to increasing test coverage for a large proportion of the backend than test more inputs for a smaller number of methods.

3.2 Tests

3.2.1 CoreGameTest

This section is responsible for testing CoreGame. This is a very key class that ties most of the back end together and is referenced heavily by the front end game screen. For this reason we used reflection to access the private fields so we could check that the getters were working correctly. This is the only test that does this.

These tests have helped catch some key bugs. Running these tests we have fixed the following problems:

1. Multiple problems with Player's resources not being set correctly at the beginning of the game. Players were either missing large amounts of fuel and gold or had much more than expected. This was solved when we changed how the constructors of Player worked and when we changed which classes were liable for which parts of the initialisation game process. (eg. CoreGame generates start resources, Player generates starting station and train. However, all is controlled by CoreGame).
2. Problems with connections between the UI and CoreGame. We realised that the UI was doing a lot of work the back end was supposed to handle when we discovered that the UI (through Interface testing) and CoreGame start values were very different
3. Saving the game was a very difficult method to create due to the amount of classes and features that have to be accessed and formatted correctly. Testing was the only way to ensure that the JSON was valid and could be interpreted by the JSON-simple library. It also allowed us to check that the save game did represent the object saved.

Due to the central nature of the class many of the bugs found had their roots in classes other than CoreGame. These bugs are explained in the sections below.

ID	Description	Category	Author	Status
Core .tCoreGame	Check's that CoreGame initialises all values correctly.	White Box	Callum Hewitt	Pass
Core .tFlipCoin	Checks that FlipCoin only generates values between 0 and 1.	Black Box	Callum Hewitt	Pass
Core .tEndTurn	Checks EndTurn correctly transfers control and increments turn count.	White Box	Callum Hewitt	Pass
Core .tStartTurn	Checks that StartTurn correctly increments player fuel based on owned station	White Box	Callum Hewitt	Pass
Core .tGetBaseResources	Tests that getBaseResources generates the correct resources	Black Box	Callum Hewitt	Pass
Core .tGetGameMap	Tests that CoreGame has access to the WorldMap.	White Box	Callum Hewitt	Pass
Core .tGetPlayer1	Tests that getPlayer1 returns the right player	White Box	Callum Hewitt	Pass
Core .tGetPlayer2	Tests that getPlayer2 returns the right player	White Box	Callum Hewitt	Pass
Core .tGetTurnCount	Tests that getTurnCount returns the turn count	White Box	Callum Hewitt	Pass
Core .getTurnLimit	Test that getTurnLimit returns the turn limit	White Box	Callum Hewitt	Pass

Core .tGetPlayerTurn	Tests that getPlayerTurn returns correctly	White Box	Callum Hewitt	Pass
Core .tSaveGameJSON	Tests that saveGameJSON generates a valid JSON file that represents the CoreGame object	Black Box	Callum Hewitt	Pass

3.2.2 ShopTest

This section is responsible for the Shop class. This is quite an important class as much of the resource management parts of the game hinge on shop being correct. Errors found using these tests include:

1. Problems with selling Electric fuel. Rounding errors kept giving recurring decimals. This was fixed by using a maths library.
2. Some fuels were selling for the wrong price. We discovered that they were working from Coal Price instead of the correct one for their fuel. This was caused by a copy paste error and was quickly amended.
3. Customers could buy too many cards. We added a limit so that the UI wouldn't crash when it had too many cards in it's card hand to display.

ID	Description	Category	Author	Status
Shop .tShop	Tests that Shop initialises correctly	White Box	Callum Hewitt	Pass
Shop .tBuyFuel	Tests that the correct amount of fuel is bought for the right price and can not be bought when the customer has no money.	White Box	Callum Hewitt	Pass
Shop .tSellFuel	Tests that the correct amount of fuel is sold for the right price and can not be sold when the customer doesn't have the correct fuel	White Box	Callum Hewitt	Pass
Shop .tBuyCard	Tests that a customer can buy up to 7 cards only when they have the right amount of Gold.	White Box	Callum Hewitt	Pass

3.2.3 Train Tests

3.2.3.1 TrainTest

Trains are important structures in our game and interact heavily with the UI. They must be tested thoroughly.

1. We didn't find many bugs using these tests. Most of the problems we had with trains were UI and were fixed using the interface tests or were related to route and were fixed below. These are mainly here for regression testing.

ID	Description	Category	Author	Status
Train .tTrain	Tests that all the different types of trains are initialised correctly	White Box	Callum Hewitt	Pass
Train .tRemoveUpgrade	Tests that removal of SpeedUpgrade works correctly	White Box	Callum Hewitt	Pass
Train .tAddUpgrade	Tests that addition of SpeedUpgrade works correctly	White Box	Callum Hewitt	Pass

3.2.3.2 RouteTest

This section is responsible for testing the routing of trains through the map. It is a key class for the backend of the Trains position and Goal completion testing, but also at the front end for displaying valid routes in the GUI and allowing valid connections to be added and removed from the path accordingly.

Bugs found include unknown properties of Vector2 vector implementation - a key feature for scaling and updating the trains position, it was important that they were used correctly.

1. It was discovered on creating a new Vector2 instance from an existing connection vector that any changes to made to the new vector, such as scaling or normalising a vector would also change the original copy of the vector within the **Connection** class.

   This meant that scaling a vector in the update method would permanently scale the original vector, losing the original normalised vector used to calculate other important features such as the position of the train.

   This was fixed using the .cpy() method when initialising a Vector2 which the values of another as such:

   Vector2 vect = path.get(routeIndex).getVector().cpy();

2. If the route was empty when calling the **inStation()** and **getStation()**, there was no check to see if the existing path was empty, so when a connection hadn't been added to a route, the code attempt to access an index of an ArrayList that was empty, throwing an exception.

   This was fixed with adding a .isEmpty() if statement in inStation() which returned true if empty.

ID	Description	Author	Status
Route .tGetTrainPos	Tests that the coordinates of a train within a route are correct after moving the train through a route.	Matthew Taylor	Pass
Route. tRouteReloadConstructor	Tests the second Route Constructor used to implement an existing route is correct	Matthew Taylor	Pass
Route. tGetTotalLength	Adds connections to a route and checks if the total length of the path is correct	Matthew Taylor	Pass
Route. tGetLengthRemaining	Moves a train through it's route, checking if the length remaining is correct	Matthew Taylor	Pass
Route. tGetStation	Checks if a train is currently in a station within the route	Matthew Taylor	Pass
Route. tUpdate	Update/moving the train through it's route, checking it's position is correct	Matthew Taylor	Pass

3.2.4 Card Tests

3.2.4.1 CardTest

Cards are one of the more interesting features of the game. It was important that they worked correctly. No bugs were found using these tests but we implemented them for the purpose of regression testing.

ID	Description	Category	Author	Status
Card .tCard	Tests that an instance of Card is initialised correctly. Uses coalCard.	White Box	Callum Hewitt	Pass
Card .tSetOwner	Tests that a Card's owner can be set correctly.	White Box	Callum Hewitt	Pass

3.2.4.2 GoldCardTest

These tests check that the **GoldCard** does what it should do and the attributes match up to what the should be.

ID	Description	Category	Author	Status
Card .tImplementCard	Tests that the player's gold increases when the card was implemented	White Box	Callum Hewitt	Pass
Card. tGoldCard	Tests that a valid GoldCard has been created including correct name, image and has an owner.	White Box	Callum Hewitt	Pass

3.2.4.3 ResourceCardTest

The Resource cards are the more boring and simple cards that we reward but it is important that the correct resource is given with the correct image and owner.

1. We found a bug during the implementation of these Cards where the wrong fuel type was increasing during implementation of the card. This was again due to a copy/paste error where the variables had not been changed properly

ID	Description	Category	Author	Status
Card .tImplementResourceCard	Tests that the player's coal increases when the card was implemented and the other resources were not affected.	White Box	Callum Hewitt	Pass
Card. tResourceCard	Tests that a valid ResourceCard has been created including correct name, image and has an owner.	White Box	Callum Hewitt	Pass

3.2.4.4 TeleportTest

Used to test the idea behind the TeleportCard in which a train is teleported from one position to another. Only partially implemented due to a lack of UI selecting support for choosing the Train and Station to use. The players first train and the teleport location LONDON were chosen as default tests.
No errors were found but it was implemented for regression testing as when this Card starts to take parameters it will become important to test it properly.

ID	Description	Category	Author	Status
TeleportCard. tImplementCard	Checks the new position of the train matches that of London	White Box	Callum Hewitt	Pass
TeleportCard. tTeleportCard	Checks the constructor initialises correctly	White Box	Callum Hewitt	Pass

3.2.4.5 GoFasterStripesCardTest

Used to increase the players Train speedMod by 10, currently uses the players first train due to lack of UI implementation.
Again this has mainly been implemented for regression testing for the same reasons as TeleportCard.

ID	Description	Category	Author	Status
GoFasterStripesCard.. tImplementCard	Checks the trains speedMod is increased by 10	White Box	Callum Hewitt	Pass
GoFasterStripesCard. tGoFasterStripesCard	Checks the constructor initialises correctly	White Box	Callum Hewitt	Pass

3.2.4.6 CardFactoryTest

Used to create new Card Instances on demand:

1. We accidentally mixed up CreateMagicCard and CreateResourceCard so each would try and create the other. We were getting IndexOutOfBounds errors however as CreateMagicCard would try to generate a card from the MagicCardList in range 0 to ResourceCardList.size() which often returned out of bounds. This was fixed by switching the methods

ID	Description	Category	Author	Status
CardFactory. tCreateAnyCard	Creates 5000 random cards and checks that at least 1 instance of each exists	White Box	Callum Hewitt	Pass
CardFactory. tCreateMagicCard	Creates 500 magic card and checks at least 1 instance of each type exists, with no resource cards present	White Box	Callum Hewitt	Pass
CardFactory. tCreateResourceCard	Creates 5000 resource cards and checks there is at least one instance of each and none of them contain a MagicCard	White Box	Callum Hewitt	Pass

3.2.5 MapTest

3.2.5.1 StationTest

The Station class in itself does not have much complexity but is used by a variety of other classes so it is important that the values it receives it assigns correctly. No errors were found here except in relation to Player.

ID	Description	Category	Author	Status
Station.tStation	Tests that the Station initialises all values correctly	White Box	Callum Hewitt	Pass

3.2.5.2 MapObjTest

The Map is built up of MapObj it is important that the MapObj's are created correctly and can be found using there correct names. A map object does not have to be a station. No errors were found here.

ID	Description	Category	Author	Status
Map. tMapObj	Test that the name is the assign one and test that it does not have a station	White Box	Callum Hewitt	Pass

3.2.5.3 ConnectionTest

Connections are the the routes between two map objects. They are very critical map objects. We found no errors but kept the tests for regression testing.

ID	Description	Category	Author	Status
Map .tConnection	Test that connection between two map object is valid	White Box	Callum Hewitt	Pass
Map. tIsReverseof	Checks that the connection recognises its inverse and does not recognise some connection that is not its inverse	White Box	Callum Hewitt	Pass

3.2.6 Dijkstra

3.2.6.1 DijkstraTest

Dijkstra is used to compute the minimum path length between two station objects.

1. The main bug we found with this was where Comparable hadn't been implemented for Node and the priority queue would fail to add Nodes as they could not be compared. This was ammended by implementing the interface.

ID	Description	Category	Author	Status
Dijkstra. tDijkstra	Tests constructor is intiliased correctly by comparing the	White Box	Callum Hewitt	Pass
Dijkstra.testComputePaths	Tests that the calculated res a value in between 0 and 2000. Alo tests that reward is a non negative number	White Box	Sam Anderson	Pass
Dijkstra.testLookUpNode	Ensures the returned node is a Node in nodeList. Ensures the returned node is not an empty node.	White Box	Sam Anderson	Pass
Dijkstra.testInitialiseGraph	Ensures that graph initialisation is correct. Asserts that every node in staionList is in node list.	White Box	Sam Anderson	Pass

3.2.6.2 EdgeTest

Edges are how Dijkstra represents Connections in a form easier to perform the algorithm. No errors were found here.

ID	Description	Category	Author	Status
Map. tEdge	Test the edge object is not null and the weight matches with the assign one.	White Box	Callum Hewitt	Pass

3.2.6.3 NodeTest

Nodes are how Dijkstra represents MapObjs in a form easier to perform the algorithm. No errors were found here.

ID	Description	Category	Author	Status
Map. tNode	Tests: The mapobj equals the one assigned. The edges.size() is correct. The minDistance is correct. The next node is not null.	White Box	Callum Hewitt	Pass

3.2.7 Goal Tests

3.2.7.1 GoalTest

Goals are very critical to the game. They represent how the player will score.

1. We found an error where Goals wouldn't create properly. They would be half formed. We were missing quite a lot of the important variables. We made sure they were instantiated correctly.
2. We had lots of problems converting the results from Dijkstra's algorithm into a form that could be given as a reward. This took some fiddling but we got it working eventually.

ID	Description	Category	Author	Status
Goal. tGoal	Tests that the goal has a start and end station	White Box	Callum Hewitt	Pass
Goal. tAssignTrain	Test that the goal has a train assigned to it.	White Box	Callum Hewitt	Pass
Goal. tIsSpecial	Test that the goal is not special. (When it shouldn't be)	White Box	Callum Hewitt	Pass
Goal. tgetReward	Test that the reward is greater than zero.	White Box	Callum Hewitt	Pass
Goal. tgetStartDate	Test that the start date is not null.	White Box	Callum Hewitt	Pass

3.2.7.2 GoalFactoryTest

GoalFactory is the only place Goals are generated outside of tests. It is important that they are right.

1. We fixed a bug where StartStation and EndStation were the same.

ID	Description	Category	Author	Status
GoalFactory. tCreateRandomGoal	Test that the goal factory returns a fully valid goal.	White Box	Callum Hewitt	Pass

3.2.8 PlayerTest

This section is responsible for testing the Player class it is an important class as it interacts with many other classes to allow the players actions to be recorded.
SellStation is not currently implemented due to not being supported on the UI as a result the contents of sellStation and its test have been commented out but were both fully functional.
Errors found using these tests include:

1. Route did not check if the route was empty, causing an error where trains were not considered to be in the station preventing the player from buying stations on occasion.

ID	Description	Category	Author	Status
Player.tPlayer	Tests that the Player initialises all values correctly	White Box	Elliot Bray	Pass

Player.tPurchaseStaton	Tests that stations adds the station to the players list of stations, correctly adds the line information and reduces the players gold correctly only when it is a valid purchase (station is unowned, player has a train in the station and has enough gold)	White Box	Elliot Bray	Pass
Player.tSellStation	Tests that the station is removed from the players list of stations and the correct amount of gold is given to the player only when it is a valid sale (the player owns the station)	White Box	Elliot Bray	Not activated
Player.tLineBonuses	Tests that line bonuses for stations are calculated and applied correctly	White Box	Elliot Bray	Pass

4. Interface Testing

Interface testing was carried by inspecting the various UI elements in the game itself to confirm if they are working/appearing as intended. This is static testing and generally covers what is not tested by Unit tests.

The Table below lists each interactive UI element, with a description of how the element can be recognised in the game and what action it should perform (if it does not currently perform an action, the action has been listed as "Nothing yet"). It also lists any errors discovered when testing the UI elements and if they were fixed, along with if the feature is functional (alters the state of the game) or non-functional (alters purely the UI) (UI elements that do not yet have their functionality listed are described as "Non-functional (currently)" as upon implementation they should become functional but as they stand are non-functional).

Feature	**Description of element**	**Description of action**	**Errors discovered**	**Fixed**	**Functional or Non-functional**
Start Menu					
newGameButton	The new game button on the main menu	Move screen (up) to the new game menu	Doesn't move to correct position	Yes	Non-functional
loadGameButton	The load game button on the main menu	Move screen (right) to the load game menu	Doesn't move to correct position	Yes	Non-functional

preferencesButton	The preferences button on the main menu	Move screen (down) to the preferences menu	Doesn't move to correct position	Yes	Non-functional
howToPlayButton	The how to play button on the main menu	Move Screen (left) to how to play menu	Doesn't move to correct position	Yes	Non-functional
exitButton	The exit button on the main menu	Exits the game			Functional
newGameBackButton	Back button on the new game screen	Moves screen (down) to the main menu	Doesn't move to correct position	Yes	Non-functional
turnTimeoutButton	Turn timeout button on the new game screen	Toggles the highlight of the turn timeout texture			Non-functional (currently)
stationDomButton	Station domination button on the new game screen	Toggles the highlight of the station domination button			Non-functional (currently)
newGameGoButton	Go button on the new game screen	Starts core game (changes screen to game screen)	Doesn't open game correctly	Yes	Functional
turn50Button	50 button on the new game screen	Toggles the highlight of the turn 50 button			Non-functional (currently)
turn100Button	100 button on the new game screen	Toggles the highlight of the turn 100 button			Non-functional (currently)
turn150Button	150 buton on the new game screen	Toggles the highlight of the turn 150 button			Non-functional (currently)
loadGameBackButton	Back button on the load game screen	Moves screen (left) to the main menu	Doesn't move to correct position	Yes	Non-functional
preferencesGameBackButton	Back button on the preferences screen	Moves screen (up) to the main menu	Doesn't move to correct position	Yes	Non-functional
howToPlayGameBackButton	Back button on the how to play screen	Moves screen (right) to the main menu	Doesn't move to correct position	Yes	Non-functional
settingsButton	Game settings button on the preferences screen	Nothing yet			Non-functional

displayButton	Display settings button on the preferences screen	Nothing yet			Non-functional
soundButton	Sounds settings button on the preferences screen	Nothing yet			Non-functional
controlButton	Control settings button on the preferences screen	Nothing yet			Non-functional
homeButton	Button between previous and next on the how to play screen	Nothing yet			Non-functional
nextButton	Next button on the how to play screen	Nothing yet			Non-functional
prevButton	Previous button on the how to play screen	Nothing yet			Non-functional
Card					
card(1-7)	Any of the possible 7 cards the player has	Raises the card up, shows the new card button and lowers any other raised cards	1. Doesn't raise 2. Doesn't lower others 3. Doesn't show used card button	1. Yes 2. Yes 3. Yes	Non-functional
useCardButton	The use card button above a card	Calls use card method passing the selected card	1. Wrong index sent 2. Not clickable due to bounds issue	1. Yes 2. Yes	Functional
Goals					
goalActor (actor for player goals and goal menu goals)	The tickets goals are displayed on	If its a player goal hover will show the plan route button, if its a menu goal it will show add goal button.	Plan route didn't hide when leaving goal bounds	Yes	Non-functional
RemoveButton(1-3)	Up and right of a goal in the goal window if you have selected it in the goal screen	If goal has just been chosen from goal menu remove button will undo that choice otherwise it will remove	Wrong texture showing	Yes	Functional

		the goal entirely.			
planRouteButton	The button on the goal ticket in the players goal window	Open routing mode	Plan route didn't hide when leaving bounds	Yes	Non-functional
addGoalButton	The button on the goal ticket in the goals window	Adds selected goal to player goals	Not passing the right index	Yes	Functional
backButton	The back button on the goal screen	closes goal menu			Non-functional
refreshGoalButton	Not implemented on the UI	Nothing yet			Non-functional
Map					
station	The station blips on the map	Opens the station info window			Non-functional
stationSelect	The select button on the station info window	If in starting sequence it selects player 1 and 2 stations otherwise it purchases stations	Not recognising trains being in the station	Yes	Functional
confirmRouteButton	Confirm button in routing mode	Closes the routing mode			Functional
undoLastRouteButton	Undo button in routing mode	Undoes the last connection			Functional
abortRouteButton	Abort button in routing mode	Undoes all connections made			Functional
cancelRouteButton	Cancel button in routing mode	Cancels the route			Functional
trainActor	The train blips that moves around the map	Moves around the map, if you click on it, it opens relevant information for that train depending on current state	Covers up stations, preventing them from being clicked/selected for purchasing	Yes	Non-functional
planRoute	Plan route button on the train info window	Enters routing mode			Non-functional
Pause menu					

game_pause_resume	Resume button on the pause screen	Closes the pause menu			Non-functional
game_pause_save	Save button on the pause screen	Nothing yet			Non-functional
game_pause_settings	Settings button on the pause screen	Nothing yet			Non-functional
game_pause_mainmenu	Main menu button on the pause screen	Changes the screen to start menu			Non-functional
Shop					
game_shop_backbtn	The back button on the main shop window	Closes the shop			Non-functional
back	Back button on the buy and sell windows	Goes to shop start screen			Non-functional
shopBuyButton	Buy button on the main shop window	Changes the shop screen to buy	Does not update prices label	Yes	Non-functional
shopSellButton	Sell button on the main shop window	Changes shop screen to sell			Non-functional
train	Train screen (not implemented yet)	Nothing Yet			Non-functional
buyButton(coal, oil, electric, nuclear and card)	The buy button for coal, oil, electric, nuclear and card to purchase the chosen item	Buys the corresponding item * quantity (if valid) (card does not have quantity)			Functional
addButton(coal, oil, electric and nuclear)	The plus button for increasing quantity for coal, oil, electric and nuclear	Increases quantity	Does not update label correctly	Yes	Functional
minusButton(coal, oil, electric and nuclear)	The minus button for decreasing quantity for coal, oil, electric and nuclear	Decreases quantity	Does not update label correctly	Yes	Functional
UI_Elements					
getStartedWindow	The window that appears when a game starts	The notification window for the starting sequence			Non-functional

game_menuobject_tickettoggle	Ticket symbol in the top left	Toggles side menu that shows the players goals			Non-functional
game_menuobject_goalscreenbtn	Goal screen button in the top left	Toggles goal screen			Non-functional
game_menuobject_menubtn	The pause button (three horizontal bars) in the top right	Toggles pause menu			Non-functional
game_menuobject_infobutton	The i button in the bottom right	Toggles map info image			Non-functional
game_menuobject_shopbtn	The dollar symbol button in the bottom right	Opens shop			Non-functional
game_menuobject_traindepotbtn	The train symbol button in the bottom right	Opens train depot			Non-functional
game_menuobject_endturnbutton	The end turn button	Ends turn and calls associated methods			Functional
game_card_togglebutton	The show cards button at the bottom	Toggles the visibility of cards			Non-functional
game_resources_togglebutton	Three horizontal bars button in the bottom left	Toggles the expansion of the resources bar			Non-functional
window	The window used for all the warning messages	The window used in all warning messages			Non-functional

5. Requirements Validation

We wanted to ensure that we were meeting the requirements we had set ourselves so for validation we inspected our game to ensure that we have features implemented to satisfy our used requirements. User requirements are all backed by system requirements so we felt like we didn't need to test validation for those. Any requirements which haven't been satisfied are featured at the end of our Architecture Document where we explain why that requirement is not satisfied.

This is testing by inspection. The 'Evidence for Satisfaction' column describes what feature can be inspected in game, and how to inspect it to ensure that the requirement has been met.

Req ID	Description	Evidence for Satisfaction
User.GP.1	Game MUST be turn based.	We have an End Turn button which when pressed switches control between two separate Player

		entities. Can be checked by playing a game and checking that before pressing End Turn you can control one set of Trains and Stations and have access to one list of goals and after you cannot access that set but you have access to a different set. Pressing End Turn again restores access to the previous set of controls.
User.GP.2.1	Players MUST be provided with goals.	Goal Menu. Mousing over a ticket icon in the GoalMenu and pressing add to Goals will assign a goal to a player.
User.GP.2.2	Goals MUST be based around sending trains from city to city.	Goals specify a start station and end station and will complete and reward players after they route a train through the start station and then the end station after having assigned a goal to a train.
User.GP.2.3	Game MUST support at least 10 different goals.	Game randomly generates goals. We have 20 stations so we have 190 different combinations of Start and End stations. (20 choose 2).This isn't even accounting for cargo being one of two options either, which varies across Goals.
User.GP.2.4	Each goal MUST have an associated number of points a player can score for completing it.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.2.5	Goals MUST be completable.	If you assign a goal to a train and route it through the Goal's start station and end station (start before end) the Goal will complete and reward the player when the train arrives at the end station.
User.GP.2.6	Users MUST be able to accept or reject goals.	The user has a choice of 9 goals in goal menu. They can choose to accept three goals and refuse to take any they do not wish to complete.
User.GP.3.1	Players MUST be able to obtain resources.	Stations will generate resources for the player. Upon pressing End Turn a Player's resources will increase based on their owned stations' resource type and resource output parameters. Players can also obtain resources by purchasing them in the shop. Players can gain gold by selling fuel in the shop.
User.GP.3.2	Players MUST be able to deploy resources.	Players 'deploy resources' by routing trains. This takes some of the player's resources as fuel for the train. They can also use resources in a shop either by selling fuel or buying fuel with gold. Player's can also use Cards from the card dock at the bottom of the screen by clicking "SHOW CARDS" and clicking one of the card icons that appear, then pressing "Use Card".
User.GP.3.3	Game MUST have only 7 different types of deployable resource.	The game's 7 resources are: 1. Gold 2. Coal 3. Oil 4. Electric 5. Nuclear 6. Cards

		7. Trains All are deployable albeit in different ways. Cards get used up for special bonuses, fuel is used by trains, gold is used to buy items in the shop and trains are deployed to the map and used to gain more resources. How these are each deployed has been explained in other rows of this table.
User.GP.3.4	Game SHOULD have a series of wild cards which cause random effects.	We generated a couple of 'magic' wild cards which teleport a train (currently only to London) when used and another which increases the base speed of a train. We also have resource cards which give the player free resources when used. These cards are generated randomly so you never know what you're going to get. You can see this when you purchase cards from the shop. You cannot specify what card type you receive.
User.GP.4	Game MUST have random events which affect certain routes, cities or other gameplay features.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.5.1	Players MUST be able to send trains between different European cities (fictional or nonfictional).	Player's can route trains between cities by selecting them and pressing PLAN ROUTE. They can then select a Station (representing European cities) adjacent to their location and, assuming they have enough fuel, select confirm to send the train to that city over the next few of the Player's turns.
User.GP.5.2	Players MUST be able to plan non-direct routes via other cities.	Player can continue to plan the route after selecting an adjacent city such that the route extends from the adjacent city to another one. They click and confirm in the same as in User.GP.5.1.
User.GP.5.3	Players SHOULD be able to abort routes. There SHOULD be a penalty for this action.	Player can abort a route at any time however they will not recover the fuel it took to get them to the next station. They click the abort button in route panning mode.
User.GP.5.4	Players SHOULD be able to halt and restart trains whilst on their routes.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.6.1	Game MUST include at least 5 different cities.	Game includes 20 different cities. As can be seen when the game is launched.
User.GP.6.2	There SHOULD be at least two junctions (i.e., train routes that intersect).	There are two junctions where lines cross located between Warsaw and Moscow and between Prague and Paris. These act like stations but do not perform the same actions.
User.GP.6.3	There MUST be at least two obstacles in the game.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.7.1	User MUST be able to score points, such that the player with the most points will win the game.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.7.2	User's score MUST be based on their achievement of goals.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.8	The system MUST have a method in which the game ends and a winner is declared (or a draw is	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.

	declared).	
User.GP.9	The game COULD have multiple game modes.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.10	Game MUST support exactly two players on one computer.	There are two distinct Player entities upon starting the game with distinct sets of Trains, Stations and Goals. Players switch between them by pressing End Turn on ONE computer.
User.GP.11.1	Game COULD support the purchasing and upgrading of stations to provide benefits to the owner. These benefits could be more train slots, charging a use fee when the other player passes through etc.	Partially implemented. Players can purchase a station at the beginning and during the game which will generate resource for them based on it's type. Rent features are available on the back end but are not fully implemented. See Architecture Doc for more.
User.GP.11.2	Stations COULD belong to a Line providing benefits if a Player owns multiple stations on a Line.	Partially implemented. Stations belong to lines and line bonuses are generated on the back end but are not fully implemented. See Architecture Doc for more.
User.GP.12.1	User COULD have the ability to upgrade trains to make them go faster, more efficient or support more carriages.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.GP.12.2	Game SHOULD support more than one kind of train.	Game supports four different types of train: - Coal - Oil - Electric - Nuclear These can all be obtained by the player by selecting a station of the same type at the start.
User.GP.13	There MUST be an in game currency.	We use Gold in the Shop as a currency for purchasing fuel and cards. You also recieve Gold for selling your items.
User.GP.14	Users SHOULD be able to purchase resources.	Players have access to a shop which sells the four fuel types and Cards in exchange for Gold.
User.UI.1	The user's current goals MUST clearly be shown.	The user's Goals are displayed by selecting the ticket in the top left hand corner of the screen.
User.UI.2	The user MUST be able to track the progress of their trains.	The Map represents the User's trains as blue and orange blips showing their locations in relation to the stations.
User.UI.3	MUST clearly show both players' scores.	Sort of implemented. The score display is available but it is not linked to a Player object as Players currently don't have scores.
User.UI.4	MUST clearly differentiate between different player's trains.	Both Player trains have different textures.
User.UI.5	COULD display both players names.	Names are displayed as labels next to the Player's scores at the top of the game screen. The current player's turn is shown on top of the End Turn button.
User.UI.6	Users MUST have access to a start menu system to start games, load games and quit the program. This should be the first screen the users see.	The interface is designed. You can currently start games, and quit the program and load games

		interface is set up but it doesn't perform an action. At launch this is the first screen the player's see.
User.UI.7	Users MUST have access to an in game menu system/ pause screen that allows user to save games and exit to the start screen.	Pressing the button in the top right corner opens the pause menu. It has the options to save the game and exit to the start screen. Save game is not linked to the button.
User.UI.8	The start and pause screens SHOULD also feature controls for a preferences screen.	The buttons are ready for a preferences screen in the pause menu and in start a preferences screen is available. However, the pause screen button does not lead to a preferences screen and at the start menu the preferences screen currently does nothing.
User.UI.9	MUST display trains on screen.	We have textures representing the Player's trains.
User.UI.10	MUST display hazards on screen.	NOT IMPLEMENTED. SEE ARCHITECTURE DOC.
User.UI.11	MUST display stations on screen.	Stations have textures and are labelled with their associated city.
User.UI.12	MUST display routes on screen.	Routes are represented with lines which are visible on the map.
User.UI.13	MUST display player resources on screen.	Player fuels and gold are located at the bottom of the screen on the left hand side. Selecting show cards will reveal user cards and player trains are displayed on the map.