

Team DUS -Testing of Assessment 2

The game must be tested in order to ensure that the system works as expected. It is important to confirm that the game performs to certain standard and reduces the chances of errors occurring due to poor coding or design. By testing small parts of code throughout development and then working up the hierarchy to finally test the game as a whole, we can reduce the time that must be spent finding the exact part of the code that is at fault if an error occurs during testing the whole system, as we can guarantee each level of the hierarchy is correct before testing the next level. The lower down the hierarchy a feature is located, the higher the chance it can automate with code. Most low level functions have predictable results and, hence, it can be checked whether the implemented function produces the result it is expected to produce. The higher a feature is in the hierarchy, however, the more likely it is that it must be tested by playing the game. These must be tested by inspection rather than by feeding data to a function. This includes many features of the GUI and the random generation of engines and upgrades. Most importantly, when tests fail, the code or general design of the feature in question can be amended, so that the code performs correctly. Nevertheless, as stated by Dijkstra [1]: “testing can only show the presence of errors, not their absence”.

Unit Testing Methodology

Our unit testing methodology follows the IEEE Standard for Software Unit Testing [2]. We chose to test in this way as unit testing is based on testing from the bottom up - testing the smallest possible classes first. Rather than testing large parts of the program, where it is not only difficult to debug but generate test data that only tests one component, unit testing ensures the smallest parts of the program works before it is used in a larger context.

Unit testing is based on testing the smallest part of code – e.g. a single function. Each test usually consists of a fragment of code which can be used to check if a unit produces the expected results. Some parts of the game cannot be tested in this way, however; the GUI, for instance, must be tested by manually clicking parts of the interface and ensuring that the program responds in the correct way though these tests can still be split into testing the smallest possible units.

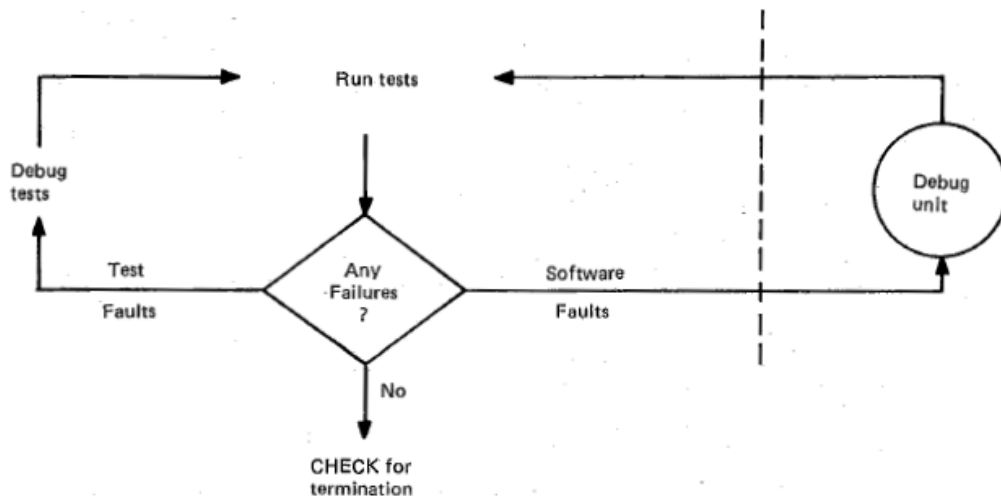
The first stage of testing is to plan the general approach, by taking the project plan and systems requirements documentation and analyse what must be tested. This involves identifying risk areas, possible inputs and ways of recording outputs. As the game will only take clicks as inputs, the possible inputs is limited to the ways in which each function may be called from other functions rather than user data entry. This reduces the amount of tests that must be done compared to a function that takes keyboard entry, for example, but tests must still be done to check how the system handles unexpected function calls (e.g. checking that if a function to create a new train is called when a player already has 3 that it will not generate a new train).

The second stage is to produce a more specific plan regarding the exact units to be tested and how each unit is to be tested by using the software architecture design documents and requirements. This stage also involves creating a schedule of unit testing based on when modules higher in the hierarchy are being developed according to the original project plan - i.e. plan to test the ‘player’ class before the development of the ‘Game’ class which makes use of the functions in ‘Player’. The ‘completeness’ (i.e.

which areas must be covered for each part of the code for testing to be considered complete) and 'termination' (what criteria should be met before the testing phase of the game's development should end) requirements should be determined while planning.

This is then followed by designing the sets of tests that should be done. This involves specifying each test case that should be implemented and working to a hierarchy where each lowest-level object can be tested directly by at least one test case. At this stage, predict any difficulties that could occur as part of the result recording/visualising - for example, tree data structure involved in the 'map' class. The implementation stage simply involves producing code that will test all features as laid out in the test plan. This consisted of using the JUnit 4 framework to set up each test as well as asserting certain conditions before, after and/or during tests.

Finally, each of these test classes was executed in order to get a returned list of successful or failed tests and any unexpected errors raised during the tests. The responses to test results follows the flowchart shown below.



For each test that fails, check where the fault is. The fault could occur in any of the following places either in the test of the class to be tested: test data, test implementation, test execution, unit implementation or unit design. Attempt to correct this fault and then rerun the test(s). Record any faults, how the code was amended to fix them and store this to reuse at a later stage. Finally, check whether the test termination criteria has been met to determine whether more testing must be done.

General Test Plan

The general test plan as described above sets out the general approach to testing. The two distinct goals of testing as described by Ian Sommerville [3] are:

1. "To demonstrate to the developer and the customer that the software meets its requirements".
2. "To discover situations in which the behaviour of the software is incorrect, undesirable, or does not conform to its specification".

These goals give rise to validation testing and defect testing (which we will split further into unit testing and inspection testing).

Test driven development was considered as it tends to produce very reliable code; however, it was considered that overall this would require so much time planning that it would be incompatible with the deadline of assessment 2 in January. It also requires very close collaboration with all members of the team which would be difficult to maintain over Christmas and revision periods. It was therefore concluded that the best method was to test after each class had been mostly completed.

Each class would be tested when it was completed. This meant classes lower down the hierarchy (i.e. by developing smallest units first) were tested before other classes which are linked to them would be developed. E.g. The connection class will be tested before development of the map class and the map class will be tested before development of the game class.

Problematic tests were also predicted. Testing the tree structure of nodes required some way of visualising this abstract data structure whilst the random generation of upgrades and engines could not be tested through the standard framework. Testing user interaction could also not be automated and would need to be manually carried out.

Detailed Test Plan

The test plan arose from designing tests that provide coverage of all features of the object as laid out by Sommerville [3]. "This means that [it] should:

- test all operations associated with the object;
- set and check the value of all attributes associated with the object;
- put the object into all possible states [i.e.] simulate all events that cause a state change."

These serve as the 'completeness' criteria for testing the system - i.e. when tests can be considered complete. The 'termination' criteria for testing the system - i.e. when to stop testing - is met ideally when the completeness criteria has been met and all tests pass; however, the deadline for assessment 2 must be taken into account meaning it must be accepted that it is unlikely that all possible conditions can be tested within the time constraints of the project.

The timetable for testing, as described above, is the same as the timetable of development so that the methods of each class are tested before development of other classes using those methods.

Each unit test was implemented using the automatic testing framework JUnit to provide general test classes that we expanded to form specific test cases.

To solve the problems identified in the general test plan we wrote a toString method for node (so that the tree of nodes could be visualised) and a code to make a list of the nodes in the tree. To test random elements, a loop was written to generate 50 engines and print the type of each to the console. This could then be analysed to determine if the engines are being generated with roughly the correct probabilities. Finally, the GUI will have to be tested by simply manually producing the conditions for each test as described in the test plan - e.g. clicking the mouse in a certain position.

The full test plan, is listed below for each class. This was used to implement the tests in JUnit or what actions to manually carry out where appropriate. For this document, results of each test and any changes made to the corresponding code are listed alongside.

Defect Testing

Defect testing is checking for errors as part of verification. Verification can be thought of as asking the question: are we “building the product right?”[4]. Therefore, these tests ensure that methods in each class respond as expected. Most of these tests can be implemented using JUnit though some (see Random Tests and Inspection Tests below) cannot. The JUnit test classes used to carry out the defect unit testing are included at the end of this document.

Route Class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)
Construct or, setTrain	Need to ensure that the is correctly created	Run the constructor, checked variables assigned in the constructor	index of current node = 0 distance = 0	index of current node = 0 distance = 0	Passed. No further action required.
updateDistanceAlongConnection()	Need to ensure that the speed of the train is right and that the train travels at the calculate speed	Moved the train, checked node and distance travelled	distance = distance travelled in a turn by a hand cart (0,25)	distance = distance travelled in a turn by a hand cart (0,25)	Passed. No further action required.
updateDistanceAlongConnection()	Need to ensure that train's position and the distance are updated when the train travels beyond a node	Moved the train beyond the next node, checked node and distance travelled	distance = 1 index of current node = 1	distance = 1 index of current node = 1	Passed. No further action required.
updateDistanceAlongConnection()	Need to ensure that train's position and the distance are	Moved the train exactly one node, checked node and	distance = 0 index of current node = 1	distance = 0 index of current node = 1	Passed. No further action required.

	updated when the train travels exactly on a node	distance travelled			
updateDistanceAlongConnection()	Need to ensure that train's position and the distance are updated when the train travels beyond more nodes	Moved the train exactly two nodes, checked node and distance travelled	distance = 0 index of current node = 2	distance = 0 index of current node = 2	Passed. No further action required.
updateDistanceAlongConnection()	Need to ensure that train's position and the distance are updated when the train travels beyond two nodes and more	Moved the train two nodes and another unit, checked node and distance travelled	distance = 1 index of current node = 2	distance = 1 index of current node = 2	Passed. No further action required.
updateDistanceAlongConnection()	Need to ensure that train's position and the distance are updated when the train travels beyond a node, and is near to reach the next one	Moved the train one node and for the distance to the next node minus one, checked node and distance travelled	distance = distance to the next node -1 index of current node = 1	distance = distance to the next node -1 index of current node = 1	Passed. No further action required.

Train class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)
Constructor	Need to ensure that trains start with the correct engine	Checked the engine	engine = "Hand Cart"	engine = "Hand Cart"	Passed. No further action required.
getSpeed()	Need to ensure that the speed is correctly	Checked the speed	speed = 15	speed = 15	Passed. No further action required.

	retrieved from the engine				
addUpgrade()	Need to check that upgrades are added to trains and that there are no upgrades on new trains.	Checked if "Double Speed" upgrade was present before and after applying it	train has double speed upgrade before adding it = false train has it after adding = true	train has double speed upgrade before adding it = false train has it after adding = true	Passed. No further action required.
addUpgrade()	Need to check that upgrades will not be applied twice	Added "double speed" to train, then tried to add again	Method throw exception	Error – no tests reapplied because use function of upgrade tests if upgrade is in train.upgrades (to prevent upgrades being added twice) This test also showed that the hasUpgrade() method was not working	Fixed by removing upgrade from train.upgrades, applying upgrade then reapplying. hasUpgrade() was fixed by adding comparator to upgrade class and using override added new compareTo function to upgrade class, then we changed this.upgrades.sort(); to Collections.sort(this.upgrades); (Using import java.util.Collections)
setSpeed()	Need to ensure that speed is updated	Speed set to 150	speed = 150	speed = 150	Passed. No further action required.
getEngine()	Need to ensure that current engine is returned	Set engine to Diesel check if engine is Diesel	engine = Diesel	engine = Diesel	Passed. No further action required.

setEngine()	Need to ensure that the new engine is applied	Set engine to Electric check if engine is Electric	engine = Electric	engine = Electric	Passed. No further action required.
setEngine()	Need to ensure that upgrades are preserved after changing the engine	Applied double speed, checked speed, applied Electric engine, checked speed	speed with double speed = 30, speed with Electric engine = 150	speed with double speed = 30, speed with Electric engine = 150	Passed. No further action required.
getUpgrades()	Need to ensure that all the applied upgrades are returned	Applied double speed and teleport	getUpgrades() = {"Double Speed", "Teleport"}	getUpgrades() = {"Double Speed", "Teleport"}	Passed. No further action required.
hasUpgrade()	Need to ensure that the method works	Checked that double speed is not present in new trains, applied double speed, checked if double speed is present	has double speed before adding it = false has double speed after adding it = true	has double speed before adding it = false has double speed after adding it = true	Passed. No further action required.

Upgrade Class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)
getName()	Need to ensure that the name is returned	Checked if the returned name was correct	name = double speed	name = double speed	Passed. No further action required.
reapply()	Need to ensure that upgrades will not be reapplied	Checked if reapply is false at the start	reapply = false	reapply = false	Passed. No further action required.
getDescription()	Need to ensure that the description is returned	Checked if the returned description was correct	description = "This upgrade doubles the speed of one of your trains!"	description = "This upgrade doubles the speed of one of your trains!"	Passed. No further action required.

Engine Class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)
getName())	Need to ensure that the correct name is returned	getName() called on a steam engine	name = steam	name = steam	Passed. No further action required.
getSpeed())	Need to ensure that the correct speed is returned	getSpeed() called on a diesel engine	speed = 50	speed = 50	Passed. No further action required.

Player Class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)
Construct or, addTrain(), removeTrain()	Need to ensure that the Player class works and that is possible to add and remove trains	Create a new player, checked the number of trains, added a train checked that the player had one train, removed a train checked that the the player had no trains	trainSize = 0 after player is created trainSize = 1 after one train is added trainSize = 0 after one train is removed	trainSize = 0 after player is created trainSize = 1 after one train is added trainSize = 0 after one train is removed	Passed. No further action required.
getTrain()	Need to ensure that trains are saved in the right way and that they are accessible	Added a single train, ensured that the train with index zero is in the first position	index of train in position 0 = 0	index of train in position 0 = 0	Passed. No further action required.

discardR andUpgrade()	Need to ensure that is possible to remove a random upgrade	Added 4 upgrades, removed a random one, checked that the player has 3	number of upgrades = 4 after adding 4 then = 3 after removing a random one	number of upgrades = 4 after adding 4 then = 3 after removing a random one	Passed. No further action required.
hasMaxUpgrades()	Need to ensure that a player can't have more than 4 upgrades	Added 3 upgrades, checked that hasMaxUpgrades() returned false. Then checked that hasMaxUpgrades() returned true after adding another one	hasMaxUpgrades() = false with 3 upgrades hasMaxUpgrades() = true with 4 upgrades	hasMaxUpgrades() = false with 3 upgrades hasMaxUpgrades() = true with 4 upgrades	Passed. No further action required.
hasMaxEngine()	Need to ensure that a player can't have more than 3 engines	Added 2 engines, checked that hasMaxEngines() returned false. Then checked that hasMaxEngines() returned true after adding another one	hasMaxEngines() = false with 2 engines hasMaxEngines() = true with 3 engines	hasMaxEngines() = false with 2 engines hasMaxEngines() = true with 3 engines	Passed. No further action required.
discardR andEngine()	Need to ensure that is possible to remove a random engine	Added 3 engines, removed a random one, checked that the player has 2	number of engines = 3 after adding 3 then = 2 after removing a random one	number of engines = 3 after adding 3 then = 2 after removing a random one	Passed. No further action required.

Map Class

Method tested	Why it was tested	How it was tested	Expected result	Actual result	Corrective action or evidence (if required or applicable)

Construct or	Need to ensure that map is correctly created inside the game from a file	Checked two random nodes and looked for adjacent nodes of one	11th node's name = Milan 9th node's name = Antwerp number of adjacent nodes to Milan = 5	11th node's name = Milan 9th node's name = Antwerp number of adjacent nodes to Milan = 5	Passed. No further action required.
findDistance()	Need to ensure that is possible to obtain distances between node	Checked distance between two nodes	Distance between Milan and Antwerp = 290	Distance between Milan and Antwerp = 290	Passed. No further action required.
giveRandomGoal()	Need to ensure that the game is able to create goals	Created three random goals and printed their description	Checked that three description are printed	Three description were printed	Evidence: Move a train from London to Durka Durka Stan Move a train from Paris to London Move a train from London to Durka Durka Stan

Randomness tests

As the game uses functions that generate weighted random data it is important to test the the output of this function is random.

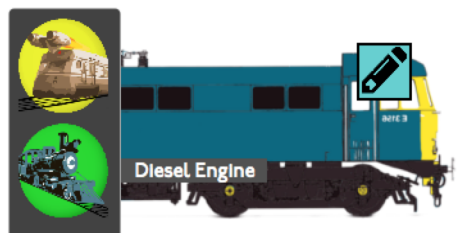
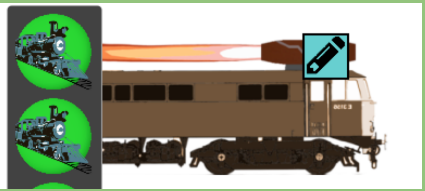
Engine	Weight	Count	Percent
Steam	40	19	38%
Diesel	30	17	34%
Electric	20	8	16%
Rocket	10	6	12%

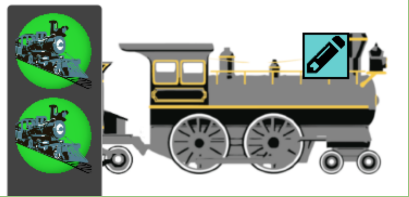
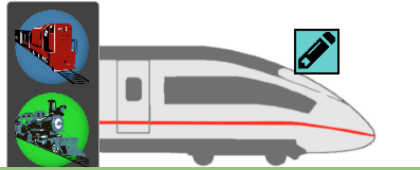

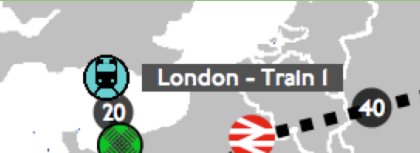
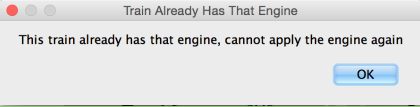
This table contains data from the creation of 50 random engines. Although the actual occurrences do not exactly match up with the weighting, the distribution appears adequately random for use in our game for balancing.


Inspection tests


Most features of the software that are higher level must be tested by inspection. The GUI is the most important feature cannot be tested with data input. Instead, someone will have to set up specific conditions in the game to be able to test certain features. For example, to check whether the inventory can display up to 7 resources correctly, the game will have to be played so that 7 resources build up and are not used. Some low-level features must also be tested by inspection too. The random generation of resources (upgrades and engines) must be checked so that all resources are being produced and are being produced in numbers roughly similar to those to be expected given the probabilities involved – however, as with all random functions, this will not always be the case even due to the nature of randomness. This will have to be tested by manually inspecting a list of types of generated objects and inspecting whether it can be assumed that the resources are (within reason) being produced to the correct probabilities.



These inspection tests have been documented in the following table. The table documents which element was tested, how and why it was tested and the result of the test. In some cases this result is a textual description, in others a screenshot is supplied showing evidence of the result. The table has also been colour coded green for pass and red for failure, if a test has failed then a solution has been included, the element is also retested.

Test Number	Element tested	Why it was tested	How it was tested	Result
1	Route creation	Have to ensure that the user is able to click the set route button to open the set route process	The set route next to a train was pressed	The route creation process successfully initiated
2	Add diesel engine to a train	Need to ensure that the diesel engine is installed on trains and that the icon is changed	A diesel engine was dragged on a train	
3	Add rocket engine to a train	Need to ensure that the rocket engine is installed on trains and that the icon is changed	A rocket engine was dragged on a train	

4	Add steam engine to a train	Need to ensure that the steam engine is installed on trains and that the icon is changed	A steam engine was dragged on a train	
5	Add electric engine to a train	Need to ensure that the electric engine is installed on trains and that the icon is changed	A electric engine was dragged on a train	
6	Stations Hover-Over s with no trains.	Need to ensure that names are correctly displayed in hover-overs that appears when the mouse hovers on a station	Mouse cursor was put on a station that did not have a train currently on it	
7	Station hover-overs with trains on	Need to ensure that the correct trains are displayed in the station tool tip	Mouse cursor was put on a station that currently had a train on it.	
8	End turn	Need to ensure that pressing the end turn button ended the turn	The end turn button was pressed.	The player's turn ended and the information for other player was displayed. All end turn methods were run correctly.
9	Add an engine to a train that already has that engine	We need to ensure that if a player tries to add an engine to a train that already has that type of engine the game the game will show a notice and prevent the user from wasting that engine	Applied a Steam engine to a Steam train	The engine was reapplied Solution: Additional code that checks whether the engine already exists on the train when the players let go the dragged engine
10	Add an engine to a train that already has that engine	We need to ensure that if a player tries to add an engine to a train that already has that type of engine the game the game will show a notice and prevent the user from wasting that engine	Applied a Steam engine to a Steam train	The engine was not reapplied. A message appeared warning the user that what they were trying to do was not permitted. 

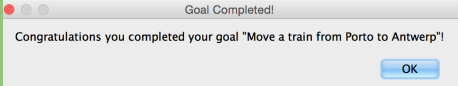
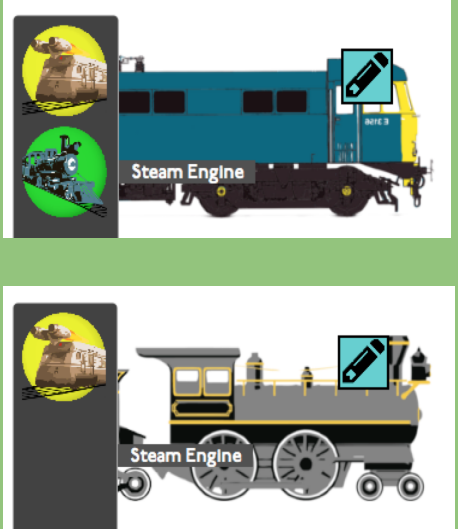
11	Add an upgrade to a train that already has that upgrade	We need to ensure that if a player tries to add an upgrade to a train that already has that type of upgrade the game will show a notice and prevent the user from wasting that upgrade	Applied a Double Speed upgrade to a train that already had Double Speed	<p>The upgrade was reapplied</p> <p>Solution: Additional code that checks whether the upgrade already exists on the train when the players let go dragged upgrade</p>
12	Start of the turn for both player	Need to ensure that both players start with three hand cart, one goal, one engine and one upgrade in the inventory	Started a new game, switched turn.	<p>Player 1 starts with three hand cart, one goal, one engine and one upgrade in the inventory. Player 2 starts with three hand cart, two goals, two engines and two upgrades in the inventory.</p> <p>Solution: An exception for the second turn was added so player 2 doesn't receive additional goals, upgrades and engines</p>
13	Start of the turn for both player	Need to ensure that both players start with three hand cart, one goal, one engine and one upgrade in the inventory	Started a new game, switched turn.	Both players started with three hand cart, one goal, one engine and one upgrade in the inventory
14	Discarding random upgrade, goals or engines	Need to ensure that the player is able to discard a random resource or goal to make space for a new one	After filling the inventory up, turn has been passed many time without using resources or completing goals	<p>The game showed a prompt each time, asking if the player wanted to discard something. Clicking "yes" resulted in discarding a random item and getting a new one, while clicking "no" left the game state unaltered</p>
15	Goal start hover-over	Need to ensure that the hover-over feature for the goal's start node functions correctly.	The mouse cursor was moved over the start of one of the player's goals.	<p>A targeting reticule was displayed over the node that matched up to the start node of that particular goal.</p> 
16	Goal end hover-over	Need to ensure that the hover-over feature for the goal's	The mouse cursor was moved over the	A targeting reticule was displayed over the node that matched up to

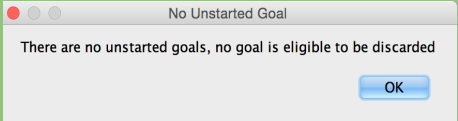

		end node functions correctly.	end of one of the player's goals.	the end node of that particular goal. 
17	Assigning multiple trains to complete the same goal on the same turn	Need to ensure that multiple trains can't be assigned routes that mean they are completing the same goal. This tested whether this occurred when the routes were assigned on the same turn.	Two trains were assigned the same route to complete the same goal on the same turn.	Both trains were assigned to complete that goal. The colour coding of the goal was assigned to the train that had its route most recently created. When the goal was completed, the user received two points and both trains were deleted as well as the goal. This clearly has failed the test. Corrective action: The game does not allow the user to select a node that is the start node for a goal with a train already assigned as the start of their route.
18	Assigning multiple trains to complete the same goal on different turns	Need to ensure that multiple trains can't be assigned routes that mean they are completing the same goal. This tested whether this occurred when the routes were assigned on different turns.	Two trains were assigned the same route to complete the same goal on different turns.	Both trains were assigned to complete that goal. The colour coding of the goal was assigned to the train that has its route most recently created. When the goal was completed by the first train, the user received and point and then the goal and both trains were removed from the player. This clearly failed the test. Corrective action: The game does not allow the user to select a node that is the start node for a goal with a train already assigned as the start of their route.
19	Assigning multiple trains to complete the same goal on the same turn -	Need to ensure that multiple trains can't be assigned routes that mean they are completing the same goal. This tested whether this occurred	Two trains were assigned the same route to complete the same goal on the same turn.	The game did not allow the user to select the start node of the already started goal as the start node of a different train's route.

	After solution applied.	when the routes were assigned on the same turn. This failed the first test so this was to ensure that the solution was effective at preventing the issue.		
20	Assigning multiple trains to complete the same goal on different turns - After solution applied.	Need to ensure that multiple trains can't be assigned routes that mean they are completing the same goal. This tested whether this occurred when the routes were assigned on different turns. This failed the first test so this was to ensure that the solution was effective at preventing the issue.	Two trains were assigned the same route to complete the same goal on different turns.	The game did not allow the user to select the start node of the already started goal as the start node of a different train's route.
21	Name entry - No value entered	Need to ensure that our game can handle the user entering no name on the initial screen and pressing 'OK'	No value was entered in the name textbox and 'OK' was clicked	<p>The game was run and the player's stored name was "", which looked strange on the UI but otherwise caused no issues. Tentative pass.</p> 
22	Name entry - Cancel clicked	Need to ensure that the game runs appropriately when the user presses 'Cancel'	The "Cancel" button was pressed.	<p>The player's name was stored as "Player 1" and the game ran appropriately, showing "Player 1's Turn" on player 1's turn.</p> 
23	Name entry - Name entered	Need to ensure that the game runs appropriately when the user enters a name and presses "OK"	The name "Matt" was entered and "OK" was pressed.	"Matt" was stored as the player's name and the game ran appropriately, showing "Matt's Turn" on player 1's turn.

				
24	Acquisition of new Goal - Accept	Need to ensure that the user is able to receive a new goal when they have the maximum number, if they want to.	When the game asked if a new goal was wanted, "OK" was pressed.	<p>A new goal was received and a random unstarted goal was discarded.</p> 
25	Acquisition of new Goal - Decline	Need to ensure that the user is able to reject receiving a new goal if they do not want one.	When the game asked if a new goal was wanted, "Cancel" was pressed.	<p>No new goal was received and the turn began as normal with the same goals as the previous turn.</p>
26	Acquisition of new Upgrade - Accept	Need to ensure that the user is able to reject receiving a new upgrade if they do not want one.	When the game asked if a new upgrade was wanted, "OK" was pressed.	<p>A new upgrade was received and a random upgrade was discarded from the inventory.</p> 
27	Acquisition of new Upgrade - Decline	Need to ensure that the user is able to reject receiving a new upgrade if they do not want one.	When the game asked if a new upgrade was wanted, "Cancel" was pressed.	<p>No new upgrade was received and the turn began as normal with the same upgrades as the previous turn.</p>
28	Acquisition of new Engine - Accept	Need to ensure that the user is able to receive a new engine when they have the maximum number, if they specify they want to.	When the game asked if a new engine was wanted, "OK" was pressed.	<p>A new engine was received and a random upgrade was discarded from the inventory.</p> 
29	Acquisition of new Engine - Decline	Need to ensure that the user is able to reject receiving a new engine if they do not want one.	When the game asked if a new engine was wanted, "Cancel" was pressed.	<p>No new goal was received and the turn began as normal with the same engines as the previous turn.</p>

30	Make route - Valid selections	Need to ensure that the route creation process works smoothly when a valid selection for the nodes is made.	The set route button was pressed for a train and a valid route was set by pressing adjacent nodes in sequence.	A route was successfully created and the train followed the route until it reached the goal node.
31	Make route - erroneous selections	Need to ensure that the route creation process does not support the selection of an invalid route	The set route button was pressed for a train and random nodes were pressed that were not adjacent to the start node.	It was impossible to create a route by just pressing random nodes, eventually just exited the route creation process as it was not accepting invalid inputs.
32	End game - Player 1 win	Need to ensure that after the set number of turns, the game ends and if player 1 has the highest number of goals completed then they are declared the winner.	The game was played out, with player 1 completing 1 goal.	After 10 turns each, player 1 was declared to be the winner. 
33	End game - Player 2 win	Need to ensure that after the set number of turns, the game ends and if player 2 has the highest number of goals completed then they are declared the winner.	The game was played out, with player 2 completing 1 goal.	After 10 turns each, player 2 was declared to be the winner. 
34	End game - Tie	Need to ensure that after the set number of turns, if both players have the same number of goals complete then a tie is declared.	The game was played out with no players completing any goals.	After 10 turns each, a tie was declared. 
35	Complete goal	Need to ensure that when a goal is completed, the appropriate train and goal are deleted from the player's inventory.	A route was created using the route creation process, end turn was pressed until the goal should complete.	At the beginning of the player's next turn, a message was displayed saying that their goal had been completed. The goal and the train had both been removed from the player's inventory.

				 <p>Goal Completed! Congratulations you completed your goal "Move a train from Porto to Antwerp!" OK</p>
36	Teleport function	Need to ensure that the teleport upgrade functions correctly. (i.e takes train from current node to end node)	The teleport upgrade was applied to a train. The train was then set upon a valid route, which took longer than 1 turn to complete normally.	<p>At the end of the turn the game crashed due to a "Out of Bounds Error" in the ArrayList. This was due to an error in the code where the train's currentNodeIndex was set to currentNodeList.size() rather than currentNodeList.size()-1.</p> <p>Solution: Changed the teleport method to have the correct value.</p>
37	Teleport function - Retested	Need to ensure that the teleport upgrade functions correctly. (i.e takes train from current node to end node). Tested again to ensure that solution worked correctly.	The teleport upgrade was applied to a train. The train was then set upon a valid route, which took longer than 1 turn to complete normally.	At the beginning of next turn, a message appeared notifying the player that the goal had been completed. Therefore the goal must have completed and teleport must have worked.
38	Edit Route	Need to ensure that the edit route functionality works correctly. It should update the current route to include the new nodes selected.	The edit route button was pressed on a train that had already travelled along some of its route.	The route was successfully edited and changed from the original. The train proceeded to instead travel along the new route.
39	Overwriting Engine	Need to ensure that the overwriting of the engine functions correctly.	An engine was applied to a train. A different engine was then applied to the same train.	 <p>The top image shows a blue steam engine with a pencil icon, labeled 'Steam Engine'. The bottom image shows a grey steam engine with a pencil icon, also labeled 'Steam Engine', indicating the engine has been overwritten.</p>

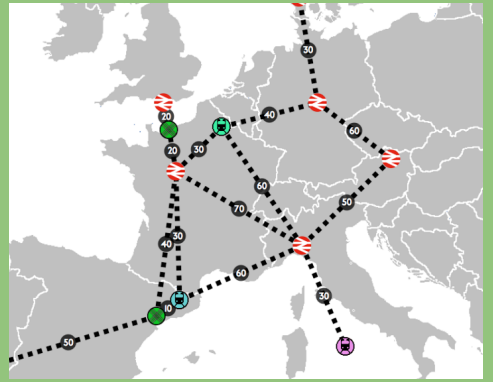
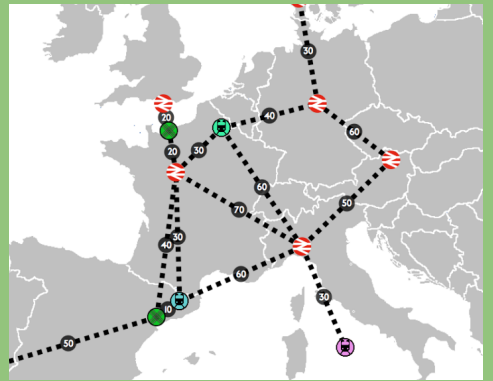
40	Discarding Goal - 3 started goals	Need to ensure that when a new goal is asked for, that there is at least one unstarted goal that can be discarded.	3 goals were obtained and a train assigned to each. At the beginning of the next turn a goal is asked for.	<p>A goal was discarded and a new one was given to the player. This should not have happened, so this test failed.</p> <p>Solution: Changed the way that <code>discardUnstartedGoal()</code> so that it does not return a goal if there are none that are unstarted.</p>
41	Discarding Goal - 3 started goals - Retested	Need to ensure that when a new goal is asked for, that there is at least one unstarted goal that can be discarded. This had to be retested after the fix was applied.	3 goals were obtained and a train assigned to each. At the beginning of the next turn a goal is asked for.	
42	Applying upgrade to train	Need to ensure that the method of adding upgrades to trains functions correctly	An upgrade was dragged onto a train.	
43	Double Speed upgrade	Need to ensure that the double speed upgrade functions correctly.	The double speed upgrade was dragged onto a train.	The train travelled at twice the speed that it was expected to with its given engine. This means that the double speed upgrade worked as intended.

Validation Testing

Validation is the process of evaluating whether a system or component satisfies the specified requirements set out at the beginning of the development process. It is considered by Boehm [4] as asking the question: are we “building the right product?” This includes providing evidence that the software satisfies the user requirements. Therefore, for each user requirement, the aspect(s) of the game which meet this requirement are listed and evidence is given of this. Where requirements have been missed, the justification for this has been provided in the System Architecture document.

High Priority Requirements


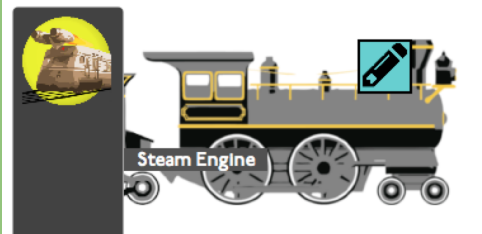
Requirement	Aspect of game that meets requirement	Evidence
1 -The user shall be able to specify the route that a particular train takes, including any intermediate stops.	The GUI allows the user to click nodes to create a route.	Inspection Test 1 Inspection Test 30 Inspection Test 31 Inspection Test 38
2 - The user shall be able to upgrade any of their base trains at the beginning of each turn, using obtained resources.	The GUI allows the user to drag resources from their inventory onto their trains in order to upgrade them.	Inspection Test 2 Inspection Test 3 Inspection Test 4 Inspection Test 5 Inspection Test 37
3 - The user shall be able to see certain information about the other player in the game, including their current goals, current score and trains.	The GUI shows the other player's trains. As scoring was not implemented they cannot see their score. Also it was decided that the player would not be able to see the other player's goals as it we were unable to find a suitable place to display this information.	N/A
4 - The user shall receive one goal at the beginning of each turn, up to a maximum of three.	At the beginning of each turn, the game gives the user a new goal if they have less than 3	Inspection Test 8
5 -The user shall receive one engine each turn up to a maximum of three, which can be applied to their current trains. If they are at the maximum then they can randomly replace an engine in their inventory.	The Game gives one engine per turn up to three, and if the max is reached the game will ask if the user want to discard a random one. If the user drags an engine onto the train then it is applied to the train.	Inspection Test 2 Inspection Test 3 Inspection Test 4 Inspection Test 5
6 - The user shall receive one upgrade each turn up to a	The Game gives one upgrade per turn up to four, and if the	

maximum of four, which can be applied to their current trains. If they are at the maximum number of upgrades then they can randomly replace an upgrade in their inventory.	max is reached the game will ask if the user want to discard a random one. If the user drags an upgrade onto the train then it is applied to the train.	
7 - The user shall be able to play the game with another person on the same machine.	The game has a turn system whereby the user is able to play with another person on the same machine.	The game supports two players in a game simultaneously, taking turns to input their commands to the game.
8 - The user shall be able to see the total distance of their selected route and how long it will take (in terms of turns and game time) at the speed of their selected train.	This requirement has not been met as we could not devise a way to appropriately display this information in a clear and concise manner that did not encroach on the rest of the GUI.	N/A
9 - The user shall be able to a view a map which shows all the nodes, the connections that link them and the locations of all trains.	The core of the game's GUI is a simple map, which displays all of the nodes, as well as the links that connect them. Trains are displayed on the nodes upon which they are currently located	
10 - User shall be able to see the progress that their train are making	The Game provides a map that shows the nodes and connections between them. The location of trains are indicated by icons on their respective nodes. It is easy to see where the start and end of each route by mousing over the goal, hence it easy to see how far the train has to travel.	
11 - The user shall be able to complete goals in order to earn points; when the user has enough points they win the game.	The game has removed the feature of earning points as explained in the System Architecture document. Instead the system declares the player with the most completed goals after 10 turns the winner.	Inspection Test 32 Inspection Test 33 Inspection Test 34
12 - The user shall be able to	This requirement has not been	N/A

see how much time is left of their turn.	implemented as explained in the System Architecture document.	
13 - Every goal must be completable	The game generates random goals in the form of "Move train from x to y" and as all nodes are connected every goal is completable.	N/A

Medium Priority

Requirement	Aspect of game that meets requirement	Evidence
1 - It shall always be possible for the user to complete a goal, regardless of any upgrades they do/do not possess.	As all goals in our game are absolute goals, i.e in the form of 'Move train from x to y', they are not upgrade or engine dependent, also all nodes are connected. Hence, they are always completable regardless of the game state.	Played the game 20 times, did not once have a goal that it was impossible to be completed. Difficult to provide comprehensive testing for this requirement, but by the way goals are generated it should always be a pass.
2 - The user shall be able to use their resources to complete optional bonuses for additional points.	The GUI allows the user to drag and drop resources onto trains. These resources improve the speed of trains which will allow the user to complete goals quicker. Optional bonuses were removed from the game as outlined in the System Architecture document due to the limitations imposed.	
3 - The user shall be able to create obstacles for the other player in order to slow them down.	This is not implemented as outlined in the System Architecture document.	N/A
4 - The user shall be able to see warnings about obstacles that occur along their current route and be presented with options to avoid them or remove them (if possible).	This is not implemented as outlined in the System Architecture document.	Inspection Test 38

5 - The user shall be able to change their train's current route mid-way through the journey, as long as the associated goal will still be met.	The game allows the user to edit the route of a train that is mid-way through it's journey by clicking on the edit route button and then selecting a suitable set of nodes.	N/A
6 - The user shall be able to see how many turns a particular obstacle will remain in place if there is no player interaction with it.	This is not implemented as outlined in the System Architecture document.	N/A
7 - The user should be able to overwrite upgrades on existing trains.	The GUI allows the user to drag an engine from their inventory onto a train which currently has an engine applied. If the engine is different to the one already on the train then it shall overwrite the current engine.	 

Low Priority

Requirement	Aspect of game that meets requirement	Evidence
1 - The user shall have the choice to randomly change an unstarted goal at the beginning of their turn if they are already at the maximum number of three goals.	The game asks the player if they wish to swap an unstarted goal for a new if they are at the maximum of three goals.	Inspection Test 24 Inspection Test 25 Inspection Test 41
2 - The user shall be able to crash their train into an enemy train, if the train is blocking their route.	This is not implemented as outlined in the System Architecture document.	N/A

3 - The user shall be able to cancel a route, hence losing progress towards their goal.	This is not implemented as outlined in the System Architecture document.	N/A
4 - The user shall be able to 'freeze' a train, at which point the train will not progress along its current route until unfrozen.	This is not implemented as outlined in the System Architecture document.	N/A

As a final test the game was compiled into a single JAR file and run on a PC in the University of York Computer Science software labs. The game ran without any issues and had the same experience as that of the development environment. This shows that our game is fit for purpose.

Conclusions

Following this thorough testing of our system, we feel that we can prove the reliability and functionality of our program. This means that when the system is extended, there can be full confidence that the platform upon which the extension built has no obvious bugs or issues that could hinder future development

References

- [1] B. Ercoli, "SOFTWARE ENGINEERING TECHNIQUES," in *NATO SCIENCE COMMITTEE*, Rome, Italy, 1969.
- [2] IEEE, "IEEE Standard for Software Unit Testing," IEEE, 1986.
- [3] I. Sommerville, *Software Engineering: 9th Edition*, Pearson, 2011, 1996.
- [4] B. Boehm, "GUIDELINES FOR VERIFYING AND VALIDATING SOFTWARE", IEEE, 1984.

JUnit Test Classes

To conduct unit testing we used the JUnit framework. Below are the test classes that we ran to conduct the testing:

EngineTest.Java

```
package com.dus.taxi;

import org.junit.Test;

import static org.junit.Assert.*;

public class EngineTest {

    //Test return name correctly
    @Test
    public void testGetName() throws Exception {
        Engine myEngine = new Engine(Engine.EngineType.STEAM);
        String name = myEngine.getName();
        assertEquals("Steam Engine", name);
    }

    //Test return speed correctly
    @Test
    public void testGetSpeed() throws Exception {
        Engine myEngine = new Engine(Engine.EngineType.DIESEL);
        int Speed = myEngine.getSpeed();
        int expected = 50;
        assertEquals(expected, Speed);
    }

}
```

MapTest.Java

```
package com.dus.taxi;

import junit.framework.TestCase;
import org.junit.Test;

public class MapTest extends TestCase {
    @Test
    public void testFindAdjNodes(){

        Map newMap = new Map();
        assertEquals(newMap.listOfNodes.get(11).getName(), "Milan");
        assertEquals(newMap.listOfNodes.get(9).getName(), "Antwerp");

        assertEquals(newMap.findAdjacentNodes(newMap.listOfNodes.get(11)).size(), 5);

    }
    @Test
    public void testFindDist(){
        Map newMap = new Map();
        assertEquals(newMap.findDistance(newMap.listOfNodes.get(11),
newMap.listOfNodes.get(9)), 290);
    }

    @Test
    public void testGetRandGoal(){
        Map newMap = new Map();
        Goal g1 = newMap.getRandomGoal();
        Goal g2 = newMap.getRandomGoal();
        Goal g3 = newMap.getRandomGoal();
        System.out.println(g3.getDescription());
        System.out.println(g2.getDescription());
        System.out.println(g1.getDescription());

    }
}
```

PlayerTest.Java

```
package com.dus.taxi;

import junit.framework.TestCase;
import org.junit.Test;

public class PlayerTest extends TestCase {
    @Test
    public void testAddTrain() throws Exception {
        Player newPlayer = new Player("Bla");
        assertEquals(0, newPlayer.trainSize());

        newPlayer.addTrain();
        assertEquals(1, newPlayer.trainSize());

        newPlayer.removeTrain(0);
        assertEquals(0, newPlayer.trainSize());
    }

    @Test
    public void testMoreTrain() throws Exception {
        Player newPlayer = new Player("Testie");

        newPlayer.addTrain();
        Train newTrain = newPlayer.getTrain(0);

        assertEquals(newPlayer.getTrainIndex(newTrain), 0);
    }

    @Test
    public void testUpgrades() throws Exception {
        Player newPlayer = new Player("Testie2");
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();

        assertTrue(newPlayer.hasMaxUpgrades());
    }

    @Test
    public void testDiscardUpgrade() throws Exception {
```

```

        Player newPlayer = new Player("Raluca");
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        assertEquals(4, newPlayer.upgradeSize());

        newPlayer.discardRandUpgrade();
        assertEquals(3, newPlayer.upgradeSize());
    }

    @Test
    public void testHasMaxUpgrades() throws Exception {
        Player newPlayer = new Player("Raluca");
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        newPlayer.giveRandomUpgrade();
        assertEquals(newPlayer.hasMaxUpgrades(), false);

        newPlayer.giveRandomUpgrade();
        assertEquals(newPlayer.hasMaxUpgrades(), true);
    }

    @Test
    public void testGiveRandEngine() throws Exception {
        Player newPlayer = new Player("Raluca");
        newPlayer.giveRandomEngine();
        newPlayer.giveRandomEngine();
        assertEquals(newPlayer.hasMaxEngines(), false);
        newPlayer.giveRandomEngine();
        assertEquals(newPlayer.hasMaxEngines(), true);
    }

    @Test
    public void discardRandEngine() throws Exception{
        Player newPlayer = new Player ("Raluca");
        newPlayer.giveRandomEngine();
        newPlayer.giveRandomEngine();
        newPlayer.giveRandomEngine();
        assertEquals(newPlayer.hasMaxEngines(), true);
        newPlayer.discardRandEngine();
        assertEquals(newPlayer.hasMaxEngines(), false);
    }
}

```

RandomTest.Java

```
package com.dus.taxe;

public class RandomTest {

    public static void main(String[] args) {

        for (int n = 0; n < 50; n++) {
            Engine myEngine = new Engine();
            System.out.println(myEngine.getName());
        }

    }
}
```

RouteTest.Java

```
package com.dus.taxe;

import junit.framework.TestCase;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

import java.util.ArrayList;

public class RouteTest extends TestCase {

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    Map map = new Map();

    Game game = new Game(map);

    Train train = new Train();

    public ArrayList<Node> listOfNodes;

    public ArrayList<Node> createList(){
        ArrayList<Node> arr = new ArrayList<Node>();
    }
}
```

```

        ArrayList<Node> a;

        arr.add(0,map.retrieveNode(1));
        a = map.findAdjacentNodes(arr.get(0));
        arr.add(1,a.get(0));
        a = map.findAdjacentNodes(arr.get(1));
        arr.add(2,a.get(0));
        a = map.findAdjacentNodes(arr.get(2));
        arr.add(3,a.get(0));
        a = map.findAdjacentNodes(arr.get(3));
        arr.add(4,a.get(0));
        a = map.findAdjacentNodes(arr.get(4));
        arr.add(5,a.get(0));
        return arr;
    }

    @Test
    public void testRoute() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        assertEquals(route.getDistanceAlongConnection(), 0);
        assertEquals(route.getCurrentNode(), 0);
    }

    @Test
    public void testLessDistance() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));
        route.updateDistanceAlongConnection();
        assertEquals(route.getDistanceAlongConnection(), (15/60));
    }

    @Test
    public void testMoreDistance() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));
        route.updateDistanceAlongConnection(distance + 1);
        assertEquals(route.getDistanceAlongConnection(), 1);
        assertEquals(route.getCurrentNode(),1);
    }
}

```

```

    public void testExactDistance() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));
        route.updateDistanceAlongConnection(distance);
        assertEquals(route.getDistanceAlongConnection(), 0);
        assertEquals(route.getCurrentNode(), 1);
    }

    public void test2NodeDistance() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));
        distance = distance +
map.findDistance(route.listOfNodes.get(route.getCurrentNode()+1),
route.listOfNodes.get(route.getCurrentNode()+2));

        route.updateDistanceAlongConnection(distance);
        assertEquals(route.getDistanceAlongConnection(), 0);
        assertEquals(route.getCurrentNode(), 2);
    }

    public void test2NodeDistancePlus1() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));
        distance = distance +
map.findDistance(route.listOfNodes.get(route.getCurrentNode()+1),
route.listOfNodes.get(route.getCurrentNode()+2));
        route.updateDistanceAlongConnection(distance + 1);
        assertEquals(route.getDistanceAlongConnection(), 1);
        assertEquals(route.getCurrentNode(), 2);
    }

    public void test2NodeDistanceMinus1() throws Exception {
        Route route = new Route(createList());
        route.setTrain(new Train());
        int distance =
map.findDistance(route.listOfNodes.get(route.getCurrentNode()),
route.listOfNodes.get(route.getCurrentNode()+1));

```

```

        distance = distance +
map.findDistance(route.listOfNodes.get(route.getCurrentNode()+1),
route.listOfNodes.get(route.getCurrentNode()+2));
        route.updateDistanceAlongConnection(distance - 1);
        assertEquals(route.getDistanceAlongConnection(),
map.findDistance(route.listOfNodes.get(route.getCurrentNode()+1),
route.listOfNodes.get(route.getCurrentNode()+2))-1);
        assertEquals(route.getCurrentNode(), 1);
    }
}

```


TrainTest.Java

```
package com.dus.taxi;

import junit.framework.TestCase;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

import java.util.ArrayList;

public class TrainTest extends TestCase {

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void testNewTrain() throws Exception {
        Train myTrain = new Train();
        assertEquals("Hand Cart", myTrain.getEngine().getName());
        //confirms trains start with hand cart
    }

    @Test
    public void testAssociateRoute() throws Exception {
        //This still must be written once Game and Route classes have
        //been finished
    }

    @Test
    public void testGetSpeed() throws Exception {
        Train myTrain = new Train();
        int speed = myTrain.getSpeed();
        int expected = 15;
        assertEquals(expected, speed);
    }

    @Test
    public void testAddUpgrade() throws Exception {
        Train myTrain = new Train();
        boolean beforeUpgrade = myTrain.hasUpgrade("Double Speed");
        assertFalse(beforeUpgrade);

        Upgrade myUpgrade = new
        Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);
        myTrain.addUpgrade(myUpgrade);
    }
}
```

```

        boolean hasUpgrade = myTrain.hasUpgrade("Double Speed");
        assertTrue(hasUpgrade);
    }

    @Test()
    public void testAddUpgradeWhenUpgradeAlreadyUsed() throws Exception
    {
        Train myTrain = new Train();
        Upgrade myUpgrade = new
Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);
        myTrain.addUpgrade(myUpgrade);
        boolean hasUpgrade = myTrain.hasUpgrade("Double Speed");
        assertTrue(hasUpgrade);

        Upgrade anotherUpgrade = new
Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);

        try {
            myTrain.addUpgrade(anotherUpgrade);
            fail("Should have thrown exception");
        } catch (UnsupportedOperationException e) {
            assertTrue(true); //exception raised correctly
        }
    }

    @Test
    public void testSetSpeed() throws Exception {
        Train myTrain = new Train();
        int expected = 15;
        assertEquals(expected, myTrain.getSpeed());
        myTrain.setSpeed(150);
        expected = 150;
        assertEquals(expected, myTrain.getSpeed());
    }

    @Test
    public void testGetEngine() throws Exception {
        Train myTrain = new Train();
        Engine myEngine = new Engine(Engine.EngineType.DIESEL);
        myTrain.setEngine(myEngine);
        assertSame(myEngine, myTrain.getEngine());
    }

    @Test
    public void testSetEngine() throws Exception {
        Train myTrain = new Train();

```

```

        Engine newEngine = new Engine(Engine.EngineType.ELECTRIC);
        myTrain.setEngine(newEngine);
        assertSame(newEngine, myTrain.getEngine()); //confirms new
engine added
    }

    @Test
    public void testReapplyingWithNewEngine() throws Exception {
        Train myTrain = new Train();
        Upgrade myUpgrade = new
Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);
        myTrain.addUpgrade(myUpgrade);
        int expected = 30;
        assertEquals(expected, myTrain.getSpeed()); //confirms double
speed applied correctly

        Engine myEngine = new Engine(Engine.EngineType.ELECTRIC);
        myTrain.setEngine(myEngine);
        expected = 150;
        assertEquals(expected, myTrain.getSpeed()); //confirms double
speed applied again after new engine

    }

    @Test
    public void testGetUpgrades() throws Exception {
        Train myTrain = new Train();
        Upgrade upgradel = new
Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);
        Upgrade upgrade2 = new Upgrade(Upgrade.UpgradeType.TELEPORT);
        ArrayList<Upgrade> myList = new ArrayList<Upgrade>();
        assertEquals(myList, myTrain.getUpgrades()); //confirms empty
arraylist for no upgrades

        myTrain.addUpgrade(upgradel);
        myTrain.addUpgrade(upgrade2);
        myList.add(upgradel);
        myList.add(upgrade2);
        assertEquals(myList, myTrain.getUpgrades()); //confirms
arraylist has two added upgrades
    }

    @Test
    public void testGetVisitedNodes() throws Exception {
        //This still must be written once Game class has been finished
    }

```

```

@Test
public void testHasUpgrade() throws Exception {
    Train myTrain = new Train();
    assertFalse(myTrain.hasUpgrade("Double Speed")); //confirms
return false when doesn't have upgrade

    Upgrade upgradel = new
Upgrade(Upgrade.UpgradeType.DOUBLE_SPEED);
    myTrain.addUpgrade(upgradel);
    assertTrue(myTrain.hasUpgrade("Double Speed"));
}

@Test
public void testGetRoute() throws Exception {
    //Wait for route and game classes to be completed.
}

}

```