TaxE Test Plan

by

SEPR Group GVA

Introduction

The purpose of this master level plan is to formulate a strategy to test future game code and to discuss the: risk, scope, approach and resources used, of the current codes testing activities.

Risk of object classes and functions will be assessed whether they are tested or not, this is to assess the overall risk associated with our program. The level of risk chosen will be an estimation based on likelihood of code fault and damage caused by such a fault. However because of the mild level of complexity of the game we have a general high level of confidence in our code. So although we label some errors as high risk this is relative to a fairly simple project. Therefore damage caused by the fault will be valued as more risk than the likelihood of a fault.

The table below is a probability vs impact matrix describing how we estimate risk.

Probability Impact	high	medium	low
high	high	high	medium
medium	medium	medium	low
low	low	low	low

Perhaps with more time we could plan and execute more thorough testing however with a tight schedule the scope of our testing will be reduced. Scope of testing will be limited to what is expected to happen within the game as described in the use cases. Due to the hot seat nature of our game the idea of ‘cheating’ will not be tested as this is outside of our control and beyond the user requirements; it will be up to players to play fair.

Test items

The objects and functions in our game vary in degrees of cohesion, so unit testing all of them alone will not be sufficient; integration tests must therefore be made. For time efficiency, higher precedence for the integration tests will be given to objects and functions that have low cohesion because they make a larger impact on the rest of the software.

Here are lists of items from our code (functions and objects) that could be tested.

Objects: Train, City Tile, Tower, Blank Tile, Rail Tile, Map, MainMenuScreen, GameScreen, CustomisationScreen, GameButton, GeneralButton

Functions: createRoute, routeGeneration

Features to be Tested

High risk
Train movement on rail, junctions and cities - it is a user requirement for the trains be able to move along rail between cities to transport passengers and be able to cross junctions along the way if needed. Thus, this is a high damage risk. It is also quite likely to happen because the path a train takes depends on the tile it is on and the next tile it is moving to. For curved rail tiles, the two tracks are significantly different in length, which is likely to causes problems. Junctions are also difficult because there are many ways in and out of the junction.

Medium risk
MenuButton and GameButton - these buttons are necessary for navigating the menu screen and in-game UI. If they fail to work properly many elements of gameplay might not be possible such as moving between different game screens.

Low risk
Ending the game - after a winner has been chosen gameplay should end, although this is no risk to the software. This is unlikely to happen because there is a simple boolean condition that cause the game to end when both trains have been destroyed. This is also a robust condition, as even if the trains are somehow destroyed at the exact same time frame the boolean will still return true.
Selecting a victor - it is a user requirement that the game be able to select a victor, although this poses no threat to the game as a whole. It is an unlikely threat because the victor is decided by who delivered the most passengers which is a simple comparison between two integers. The only special case is a draw which is unlikely because there a many variables which will change the number of passengers delivered in a given journey.

Features not to be Tested

Ending turns - we have a working implementation of a turn phase cycle that has been trial tested
ChooseRailTextures()

Where applicable if a test is made, which covers a set of potential bugs, and encounters errors the set of bugs it tests will be split into many individual tests to find the cause(s) of the error(s) For functions where the type of output was not important and we wanted to check code correctness quickly we used print statements as weak unit tests, for example:

```
System.out.println("Turn no: " + turnNo + ". Phase: " + phaseNo);
```

print outs in the console of the Eclipse IDE the result of taking a number of test turns. In this case we observed integers in place of ‘turnNo’ and ‘phaseNo’ .

However for thorough unit testing we have decided to take a black box approach. By making a set of assertions about the code under test we can guarantee that those initial conditions were met before a function.

Buttons in our game have been implemented by GeneralButton class, as such it is fair to assume that all instances of GeneralButton act in fairly the same way, therefore testing only one out of the many buttons there are in the game, should represent how they all function correctly. However the test is simple enough that it can be done to a few buttons without wasting time.

To test buttons we will observe the effects of left and right, clicks and drags, inside and outside the region the button listener occupies.

Keyboard input is only used on two occasions: when typing a company name and when scrolling the map. We will test the outcome of an exceedingly large text input (100 characters) for the company name input. As the game assumes use of a standard keyboard for play, the full range of keyboard inputs (letters, numbers and special characters) will be put into a test string.

Types of Tests

Unit Testing

It is difficult to undertake unit testing on our game because testing a class and/or any of a class’ functions you first need to create an instance of that class. This is difficult to do in our code because most classes rely a lot on the game itself being created, and so we would have to pass an instance of the game into the Junit test case. Each class also relies heavily on other data structures that are created when the game is first initialised due to the dynamic nature of the way our game is built. All of these factors meant that unit testing would have been unsuitable for this code. On hindsight we should have designed and carried out our unit tests before writing the code, in a test driven development style. This would have meant that we would have had a large amount of unit tests which could be reused for things like regression testing.

Acceptance Testing

Our code development relies heavily on acceptance testing because we can only interact with the data structures once they have been created, which is only possible when the game is running as they are all dynamically created at the beginning of each game instance.
This allowed us to spot problems such as the fact that once a tower has been built, when the phase changes (and thus the game screen is redrawn) the tower disappears. This is because we have not properly stored the towers in a data structure that belongs to the game, unlike the trains and tiles which are all stored in arrays created in the game class.
It also allowed us to visualise the whether the chooseRailTextures() class worked by adding routes of track and watching to see if the track textures were all aligned in the game. We were then able to perfect this and use the textures to test train movement.
It also allowed us to test the way that routes are specified, we could initialise trains in different cities and use the isReachable() function to see whether the appropriate cities would be highlighted. When they were not, such as in the case of the junction, where the algorithm stopped at the junction and therefore did not highlight any cities beyond. This error allowed us to work through the code systematically to find out where the issue was and correct it.

Systems Testing
To try and find issues with the code (of which there may be one or two) we gave the system values which should cause errors, such as defining routes that don't go from city to city or creating a train in a tile that is not a city. This allowed us to, whilst making sure we know exactly what needs to be passed into a function or class constructor, to test the limits of our program.