Software Engineering Group Project
Maintenance Manual

Author:	Ash Bagnall [asb20], James Green [jag77]
Config Ref:	MaintainanceMaunalGP11
Date:	11th May 2022
Version:	1.0
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

This document answers any questions that may arise about the project for installers and maintainers.

1.2 Scope

This document supports the project by answering any likely questions that maintainers may have about the project.

1.3 Objectives

The objective of this document is to provide the accurate information about the project.

2. THE MAINTENANCE MANUAL

2.1 Program Description

The program is a remake of the famous board game Buccaneer with slight adaptations to make the implementation much easier. It uses a turn-based system to allow up to four players to play at once, allowing each player to execute a given action each round such as attack, trade, move or change direction. The game has been developed using a MVC (Model-View-Controller) design pattern that divides the program logic up into three separate interconnected elements.

2.2 Program Structure

The program structure is fully documented in the Design Specification, displayed using case diagrams, Java outline classes and UML diagrams. These diagrams show all the methods in the project, with a brief description of each and the parameters it takes along with the values it returns.

2.3 Main Data Areas

The main data structures in the program are ArrayList and arrays that are stored in the main Game class. The Game class contains an ArrayList for all the players in the game, ports, crew cards and chance cards. Furthermore, the Game class also contains an array of Tile[20][20] that has all the data in each cell of the board. This can be seen under the Game Class in the Design Specification.

2.4 Algorithms

In the program, there are a few significant algorithms that are used to help the game run smoothly and efficiently.

To start with, one of our biggest collection of algorithms would be the path finding algorithms that allow for movement on the 20x20 tile board. The main algorithm checks that there is a valid path from the player and is called every single time any movement is required or even for usage in the chance cards. The method containing the algorithm is shown below:

```
public boolean pathUpToTileFree(int toCol, int toRow, Tile[][] gameBoard) {
    int [] moveDir = directionalMovement.get(direction);
    int tempCol = col, tempRow = row;
    if (inlineWithPlayer(toCol,toRow) && withinMovingDistance(toCol,toRow)){
      while (tempCol != toCol && toRow != tempRow){ // will intersect eventually
        tempCol += moveDir[0]; tempRow += moveDir[1];
        Tile tempTile = gameBoard[tempCol][tempRow];
        if (!tempTile.isTraversable()){
          return false;
        }
      }
    }
    return true;
  }
```

This algorithm is extremely useful for both movement and allowing players to effectively ‘teleport’ across the board for when chance cards influence a player. As seen above, the algorithm first collects the player’s current compass direction by returning the coordinates of the first tile that the player is facing to start the path finding algorithm. It then stores temporary column and row values that it will reference later in the algorithm. Using other ‘path finding/verifying’ algorithms written by us, it first checks if the point it wants to move to is both inline with the player (otherwise it’s an invalid move) and checks if it is within moving distance of the player- using the total value of the player’s crew cards. It then iterates until it reaches the point in which it wants to access. As the tiles are of an instance Tile, the algorithm then checks if the tile the player wants to move to is traversable, if not it returns false. This is a significant algorithm used for movement throughout the game and provides a basis for the controller of the main board to highlight cells when the player wants to move (UI).

Furthermore, our biggest algorithm is contained in our ‘handlePlayerMovement’ method under our Game class. This algorithm does what is implied in the title, it handles all player movement in the game. The algorithm first grabs the current player and checks if the player can first move in any direction, although this does not seem like a valid move, after an attack, the loser is required to move in any direction, thus why we check it here before any other checks. The algorithm then proceeds to ensure that the player has not already rotated their ship or moved during this same turn as that would be an invalid turn.

After doing a few small checks, it checks if the tile that the player wants to move to is first within the bounds of the board (0-19, 0-19), if so, we check use our algorithm listed above to check if the path directly to the point that the player wants to move to is free, if not the move is therefore invalid and the UI displays the tiles leading up to that point as red.

We then check that the player can move in a straight line to that point. Once these checks are done, we then check the details about the actual point the player wants to move to. If it’s a PlayerTile (that implements Tile), this indicates that the player wants to attack. In which case, the player is offered the option to attack this player, if so then our attacking algorithms take over and deal with any outcomes or transfer of cards/treasure. Additionally, if this Tile is a Port, then we interact with the Port and the port then handles all the interaction from there.

Moreover, if this is just a normal OceanTile, we allow the movement and place the player within this tile on the 20x20 board and the movement is complete.

2.5 Files

The program no longer supports Java Persistence so there is not further need for us to save data to files. Previously in other versions, there were multiple JSON files that contained data which helped to save the current game state. These files contained the Game object which held all information regarding the 20x20 board and what was contained in each cell, along with the players in the game and all the ports.

2.6 Interfaces:

The user must have access to a mouse, and preferably a keyboard. However, there are no external devices required for the program to run.

2.7 Suggestions for improvement

Some basic and obvious improvements would be to finish the functional requirements (namely, implementing object persistence and implementing all of the chance cards. To improve maintenance a refactor of the chance cards would be advised as the current implementation involving a switch statement is not ideal nor is it very object orientated. Furthermore, the treasurehand and crewhand share functionality and instead merging that functionality into a superclass or interface could be ideal. Especially for the implementation of the chance cards that the user holds (leading to a chancecardhand). Furthering this idea of implementing an interface to join similar classes, a lot of classes use treasurehands and crewhands (flat island, player, ports etc) so having all of these under an interface (an "entity" interface perhaps?) so that accessing similar functionality would be named and accessed in a consistent way throughout the program, and would mean that in certain cases in the program, more variables could be made private consistently. This is only mentioned as some variables in classes are left as public for ease of access. Furthermore, expanding on the Model View Controller system that is used throughout the program would be a good idea as at some points this is not properly integrated (basically anything outside of the gamescreen-gamescreencontroller-game connection does not contain a "model". For example, the attack screen (controller) should implement a class that deals with attacks as the "model" to effectively separate functionality into clear sections. Whilst we also made an effort to have methods be in appropriate locations, this is not always the case. So a refactoring of the location of code to make more sense may be useful for the overall maintenance of the code.

2.8 Things to watch for when making changes

Most things are passed through the "game" class and the "game screen controller" so any changes to these two classes will affect the program a lot. Furthermore, the "app" class acts as a connection point between the different screens as we wanted to reduce coupling between the different classes by having them create the screens within their class. So any new screens added to the code should be implemented in a similar way, by setting up the screen and loader within app and implementing getter methods for each.

2.9 Physical limitations of the program

Whilst we are not aware of any specific numbers, we would assume that this program would run on a relatively low spec computer made within the last decade, possibly even more as long as it has a relatively up to date operating system. Of course, as mentioned in interfaces, a mouse (and preferably keyboard if they wish to type names) are required to play the game.

2.10 Rebuilding and testing

The program is built from "\gp11\src\Code\gp11_project_src_code" and all relevant files are already within this directory. The project is built using maven so (as long as the programmer has access to the latest maven update) there should be no need to download any jar files used in the project (javafx, and possibly Gson when object persistence is implemented). When it comes to testing, we have made a testing directiory where all of the tests for each appropriate class are stored. Junit was used for testing throughout the program, so any new tests would be done in this directory.

REFERENCES

[1] QA Document SE.QA.05 – Design Specification Standards

DOCUMENT HISTORY

Version	*Issue No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	11/05/2022	N/A - original version	BHW