CAB201 Programming Principles

Assignment: Major Project – Space Race

Part A: Console Implementation

Semester 2, 2018

Due Date: 28/10/2018 20:00

Weighting: 40% Part A - 20%; Part B - 20%;

Assessment type: Individual, with optional pairs

Specification version: Version 1.0 (5th September 2018)

In this assignment you have the option of working in pairs. If you work alone, you will not receive any special consideration. Programming pairs need to be registered with the CAB201 teaching team by email (cab201admin@qut.edu.au) before ***21/09/2018 23:59.***

INTRODUCTION

This assignment aims to give you a real problem-solving experience, similar to what you might encounter in the workplace. You have been hired to complete a partially implemented prototype for a project that has not been fully specified at this stage. There is supporting documentation to the project in addition to this document. You are required to deliver a working prototype by the required date.

THE TASK

For this assignment your task is to develop a program for an on-line board game with the working title, **Space Race**. In Part A you will develop a Console application and in Part B a Windows Form GUI (Graphical User Interface).

The reason for implementing the Console application is so that you can develop the logic of the game correctly and test your program thoroughly without the task being complicated by trying to develop the GUI. You will see how to create Window Forms programs in Lectures 7 & 9 and the associated weekly worksheets.

If you work on this assignment with a partner, you should work together on each version, rather than one doing the Console version and the other doing the GUI version. This is because the assignment is designed so that the GUI version builds on top of the game play logic of the Console version, i.e. you can't develop each version independently.

The requirements for the GUI version will be released as Part B of this assignment specification, once Lecture 9 has taken place. Do not try to work on the GUI version until you see Part B of the specification, i.e. you can't just make up a GUI of your own design.

Read up to the end of the first paragraph in the section with the Heading "WHERE TO START" before attempting to write any code.

THE SPACE RACE GAME

This is entirely fictitious board game and bears no resemblance to any actual on-line game of the same name. It is similar to the popular ancient Indian board game, **Snakes and Ladders**. If you are unfamiliar with **Snakes and Ladders**, refer to the entry in Wikipedia.

The game is a simple race contest based on sheer luck. It is played between 2 or more players **(limited to 6 for this assignment**) on a board consisting of 56 squares and a pair of six-sided dice. The object of the game is for each player to move their token according to the roll of the dice from the Start square to the Finish square, helped or hindered by landing on certain select squares. Players take turns rolling the pair of dice once and moving their token the required number of squares. When all players have had a turn, that **round** is completed.

The squares are numbered 0 to 55, with square 0 the "Start" square and square 55 the "Finish" square. Each square from 1 to 54 is either an "ordinary", "wormhole", or "blackhole" square. Landing on a "wormhole" or "blackhole" square will result in the player being transported to another square as part of their turn. Landing on a "wormhole" square moves the player closer to the "Finish" square whilst landing on a "blackhole" square moves the player further away from the "Finish" square.

- Squares 2, 3, 5, 12, 16, 29, 40 and 45 are "wormhole" squares
- Squares 10, 26, 30, 35, 36, 49, 52, 53 are "blackhole" squares

Each player starts the game with a limited amount of fuel and landing on any square, at the end of their turn, results in the consumption of a specified amount of fuel. Should a player run out of fuel that player will take no further part in the game. It is a possibility, though very unlikely, that all players may run out of fuel before reaching the Finish square in which case no one would win the game.

Unlike most board games where the first player to reach the finish, wins and the game is over, in this game all players still in the game will complete their turn in that round. This gives the possibility that more than one player may reach the "Finish" square in a round and so there would be multiple winners. See document titled **Screenshots of Console Play** which shows various aspects of the game play. Your console interactions should be identical to these screenshots, though the actual output values will differ.

PROTOTYPE IMPLEMENTATION NOTES

The supplied prototype consists of four (4) projects within the Solution file, **Space Race.sln**

- The **Console Class** will contain the high-level code for running the Console version of the game in ***Main***. This class currently has two trivial methods which output various messages to the **Console,** other methods will need to be added**. *Main* contains a suggested high-level algorithm for playing the game once.**

- The **Game Logic Class** will contain code that will be used by both the Console application and the GUI. Extra private instance variables, public properties and methods will need to be added to this class. It currently contains two empty methods ***PlayOneRound*** and ***SetUpPlayers.***

- The **GUI Class** contains incomplete code for creating the GUI version of the game. No code in this class will be used by Console application and should be ignored until Part A is completed. ***Do not edit the current contents of this class until starting Part B.***

- The **Object Classes** contains the low-level objects used by both versions of the game.

  - The **Board Class** – Board.cs

    *This class models the board used in this game. I.e. this class contain all the squares that make up the board in the array named **squares**.*

    ***This class is incomplete**. While no other instance variables or properties are required to implement the board, code must be added for initializing the 54 squares in the method **SetUpBoard.** Also, the method **FindDestinationSquare** needs to be completed. No additional pubic methods or properties are required in this class. You can define additional constants or variables if you require them.*

  - The **Die class** – Die.cs

    *This class represents a many-sided die (commonly called a "dice") with each face having a distinct and unique value between 1 and the number of faces. **This class is complete**, i.e. no other variables, properties or methods need to be added to this class.*

    *Note that the Class variable **random** is initialised using the constructor which takes a parameter (seed). This is so that when testing your code, the same sequence of "random" values will be generated. You can change the seed when testing your code.*

  - The **Player Class** – Player.cs

    *This class represents a player of the game. A player has a name, an amount of fuel, knows which square they are currently on (**location**) as well as the number of that square (**position**). (Although not important for the Console game, a player has two other properties that will be used in the GUI game: a token-colour and a token-image.)*

    *Note the constructor only initialises the **name** of the player, other instance variables are set by the **Game Logic Class** using the various public properties, Position, Location, and FuelLeft that are available in this class.*

    ***This class is incomplete**. The bodies of the following two methods need to be completed: **Play** and **ReachedFinalSquare**.* You may need additional private instance variables and/or private methods in this class to play the game according to the assignment specifications . However, make a note of these additions as you will need to mention these additions in your final project report.

  - The **Square Class** – Square.cs

    *This class represents an ordinary square on the board, including the Start and Finish squares. It is also the base class for the **Wormhole Square** class and the **Blackhole Square** class.*

    Each square has a number which is the position (0 ... 55) of the square on the board, with the Start square's number is 0 and the Finish square's number is 55. Each square also has a name which is simply the string version of its position on the board except the Start square's name is "Start", and Finish square's name is "Finish".

    *The **NextSquare** property is only used by Wormhole and Blackhole squares to "jump" to their respective destination square.*

*The method **LandOn** for an Ordinary square uses a constant amount of fuel regardless of the number of squares traversed to arrive at that square. For Wormhole and Blackhole squares this method consumes a specified amount of fuel as well transporting the player to another square on the board.*

***This class is complete**, i.e. no other variables, properties or methods need to be added to this class.*

- The **Blackhole Square Class** - BlackholeSquare.cs

  *A subclass of Square which represents a Blackhole square on the board. **This class is complete,** , i.e. no other variables, properties or methods need to be added to this class.*

- The **Wormhole Square Class** – WormholeSquare.cs

  *A subclass of Square which represents a Wormhole square on the board. **This class is complete**, i.e. no other variables, properties or methods need to be added to this class.*

WHERE TO START

Tackling a large project such as this assignment may initially seem daunting, however breaking it down into smaller tasks that can be implemented and tested will make the task much more manageable and is an important skill for professional practice. You will develop this assignment as a bottom-up implementation exercise. This means that you will work with the supplied code in **Space Race.sln** that has already been written and implement methods, one by one, starting with various low-level functionality and incrementally implementing the body of ***Main*** in the ***Console Class*** to test the low-level functionality.

Become familiar with the six classes in the **Object Classes** project and, the Constructor methods for ***Square***, ***WormholeSquare*** and ***BlackholeSquare***. Starting with the ***Board Class*** implement the two incomplete methods, ***SetUpBoard*** and ***FindDestinationSquare***. When compiling (building) the ***Board Class,*** select ***Build Object Classes*** from the ***Build menu.*** This will restrict any compiler errors to just this class.

To test if these methods are correct, call ***SetUpBoard*** from ***Main*** in the ***Console Class.*** A ***Breakpoint*** has been inserted on the last line of ***SetUpBoard*** so that you can examine the contents of the array ***squares***. **Do not proceed any further until you can initialise each element of *squares* correctly.**

In the completed version of the Console implementation, the program will ask the user the number of players in a particular game and will check that the number of players is between 2 and 6 inclusive. Start with two players only for testing purposes as per the declaration in ***SpaceRaceGame.cs*** before

In ***Game Logic Class*** implement the method ***SetUpPlayers***. The players are stored in a special **List** type, called a **BindingList**, which is just like an array except the **List** can grow as elements are added to it. For the Console version use the default names defined in the string array ***names.*** When compiling (building) the ***Game Logic Class,*** select ***Game Logic Class*** from the ***Build menu.***

To test this method, place a ***Breakpoint*** on the line containing the closing brace (curly bracket) of whatever loop construct you are using to initialise the player's instance variables. Run the program and examine the contents of the list ***Players.*** **Do not proceed any further until you can initialise each element of *Players* correctly.**

When you start to play the game, you will become aware of the need for additional instance variables and/or properties in this class which will need to initialize in this method.

Do not cut and paste or copy code from either the Game Logic Class or Object Classes into the Console Class.

You are now ready to implement the loop to play a round of the game in ***Main***. In this game, a player does not need to roll the exact number to reach the Finish square. For example, a player may be on Square 50 and roll a total of 7, this player will finish on Square 55, the Finish square. However, be carefully that you code does not attempt to place the player on the non-existent Square 57.

When you believe your program is working correctly for any number of players, add to your code so that the user can choose to play another game or exit the program.

There are to be no explicit ***Write or WriteLine*** calls within the body of ***Main,*** you should use trivial methods similar to ***DisplayIntroductionMessage*** method in the ***Console Class*** to output any messages to the screen.

Any numeric input will need to be checked that it is the correct type and that it is within the range of allowable values for that input.

When asking the user to play another game, your code will accept either **"Y"** or **"y"** as **yes** and all other inputs will mean **they wish to exit the program.** If the user decides to play another game, they will have the option of changing the number of players in the new game.

To test the possibility of one or more players running out of fuel you will need to change the ***INITIAL_FUEL_AMOUNT*** in the ***Player Class*** to a much smaller value. However, be sure to change the constant back to the original value of 60 before submitting the assignment.

ACADEMIC INTEGRITY

Please read and follow the guidelines in QUT's Academic Integrity Kit, which is available from the Blackboard site on the Assessment page.

Students are reminded of the following MOPP statement C/5.3.7 which I have paraphrased as follows:

"***To assist in identification of potential breaches, unit/course coordinators may require students to authenticate their learning on the assessment item (for example, by showing notes/drafts/resource materials used in the preparation of the item, or by undertaking a viva or practical based exercise***)".

Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (http://theory.stanford.edu/~aiken/moss/).

FINAL COMMENT

Though all care has been taken in the production of this specification and related documentation, there may be a need to notify by email any alterations/clarifications to this specification and related documentation. **SO, CHECK YOUR QUT EMAIL DAILY**