DESIGN DOCUMENT

for

Java GameProject Version 1

Adam Ryan (14395076)

COMP30820

May 12, 2021

Contents

1 Introduction

1.1 Purpose

As part of the COMP30820 Module, students had to complete a project to design a Game Project system capable of playing some basic number guessing games through the terminal. In addition to submitting the working code for that project, students are required to submit a report supporting the details of their design, examining it from the perspective of topics covered during the Java module and explaining some of the key design decisions and architecture.

1.2 Overview

As part of this project, there were four key elements which were present:

- The Project Required a multi-tiered menu system comprising of:
  - An initial menu to choose to start or leave the app.
  - A menu to choose your player type.
  - A menu to choose the game.
- Two number guessing games, with examples provided consisting of:
  - Blackjack.
  - Roulette.
  - Rock paper scissors.
  - Coin-flipping.
- A player and specialisation of a player which lasted for the player's session.
- A persistent global leaderboard.

Based on these high level requirements, I elected to implement the following:

- A menu with various subclass menu instances with their own options.
- A token-based betting version of CoinFlip and Rock Paper Scissors.
- A persistent leaderboard.
- A player and a 'hardcore' player; a 'hardcore' player being one which introduces a higher degree of randomness into the games.

2 System Design and Design Decisions

2.1 Class Architecture and Design Decisions

This section details the classes and some of the key items present in relation to lecture material. Almost all classes feature private variables accessed via methods and public methods to allow for various components to be accessed. This is not problematic in the app's current design as the key interaction with the components if via wrappers and the getPlay section of games for which it represents the largest risk. Overall, to keep the application modular and relatively devoid of cross-dependencies, I have generally tried to avoid passing self defined classes into other self-defined classes which would intertwine the operations of those classes. Although that could, in some cases, lead to a lot nicer design, the burden has instead been shifted to the run flow which call the components and serves as a controller between them. The only exception to the usage of this convention is in the Leaderboard class, where setting the player list to a public variable ended up neater in being able to call certain ArrayList methods which resulted in easily clearing some aspects of the Leaderboard that would otherwise result in a little bit less readability if pushed solely through class methods (e.g. the sorting of players and inserting of new player values) albeit still achievable.

2.1.1 Menus

One element of note in the Menus is that as they're largely compartmentalised outside of the Game Choices, there are some instances where it would have been reasonable to e.g. pass in a list of players or games, however for the purpose of keeping the code modular and non-intertwined, this design approach was avoided and, in general, I drive this via the Run method. Within the Menus section, the advantage of using some of the data structures discussed in the module were of particular use as HashMaps allowed for a neat way to display menu options to the user while ensuring a generalised solution could be implemented for showing options without much need for reconfiguration.

Menu This is the primary menu class containing a title, a question and options to choose from. For compartmentalisation, the variables are accessible through getters and setters. A to string method is implemented for easy display, and a StringOptions method is added to help the toString method. This is public to allow easy access and view as to what is in the hashmap in a visual manner.

StartMenu Largely an instance of a Menu but with specific private variables relating to the options, menu title, and menu question.

PlayerTypeMenu Largely an instance of a Menu but with specific private variables relating to the options, menu title, and menu question. It contains a callAction method which is used to choose the inputted option to return a player. One element of note is that the more modern design choise would neglect this OOP structure and instead go with an Interface based structure. In such an instance, this would comprise an Interface called Actionable and this menu would implement the Actionable interface.

gameMenu This is largely a holdover from when I initially thought each game was meant to have an individual leaderboard. Once that was not present, this became largely redundant, however because it could end up interesting at some point I left it in as it allows for room for expansion with regards to the Leaderboard per game element.

2.1.2 Games

This was added as although not a requirement, I thought there were some elements of the games which would be shared. As previously discussed, I initially thought that each game was meant to have a leaderboard, so I created the game method largely to hold which menu should be linked and what should be displayed in the menu, and the subclasses would then leverage the individual logic for their games. As it turned out, since the leaderboard needed to be global, this could have potentially been better served as a superclass, with a subclass of a 'Bettable Game' which contains the logic for betting games (which both games I implemented are) however as the intiial design was not going to feature as heavy an overlap and was to look at win streaks this was not implemented. Regardless, this is one area of the design structure which could be tidied but as the games were meant to be method-based this was not viewed as critical. The methods were kept as public for these games as the games are accessed solely via the Playable Game method. as with the other classes, the working variable are private. As the report needs to be concise, I will avoid going into the details of specific methods within classes and focus more on the broad details of the class.

Game This is the primary menu class containing a title, a question and options to choose from. For compartmentalisation, the variables are accessible through getters and setters. A to string method is implemented for easy display, and a StringOptions method is added to help the toString method. This is public to allow easy access and view as to what is in the hashmap in a visual manner. One key design decision to note is that I initially had considered that a game has a player, however because the Player is persistent to the session rather than the game I decided that the Player should be decoupled from the games and that's why the only imeplementation of Player in the Game classes is in setting the hard mode. I decided based on how the session element works, it is more sensible for that to be handled via the menuing and leaderboard system, rather than something the games should pay particular attention to even if it may limit some functionality (such as calling out player names).

CoinFlip following from the description of how the game operates in **??**, the key variables in this class relate to the minimum stake, the modifier, starting points, current points, and winning streak. Although some of these may seem unnecessary (e.g. starting stake) the key purpose was to allow for a later degree of expansion on what the games consist of while also making it much easy to turn into a 'proper' bettable games interface which coinflip (and rockpaperscissors) implement.

RockPaperScissors This element is almost identical to CoinFlip with the exception of the logic in how the game operates (guessing rock paper scissors instead of heads or tails),

2.1.3 Players

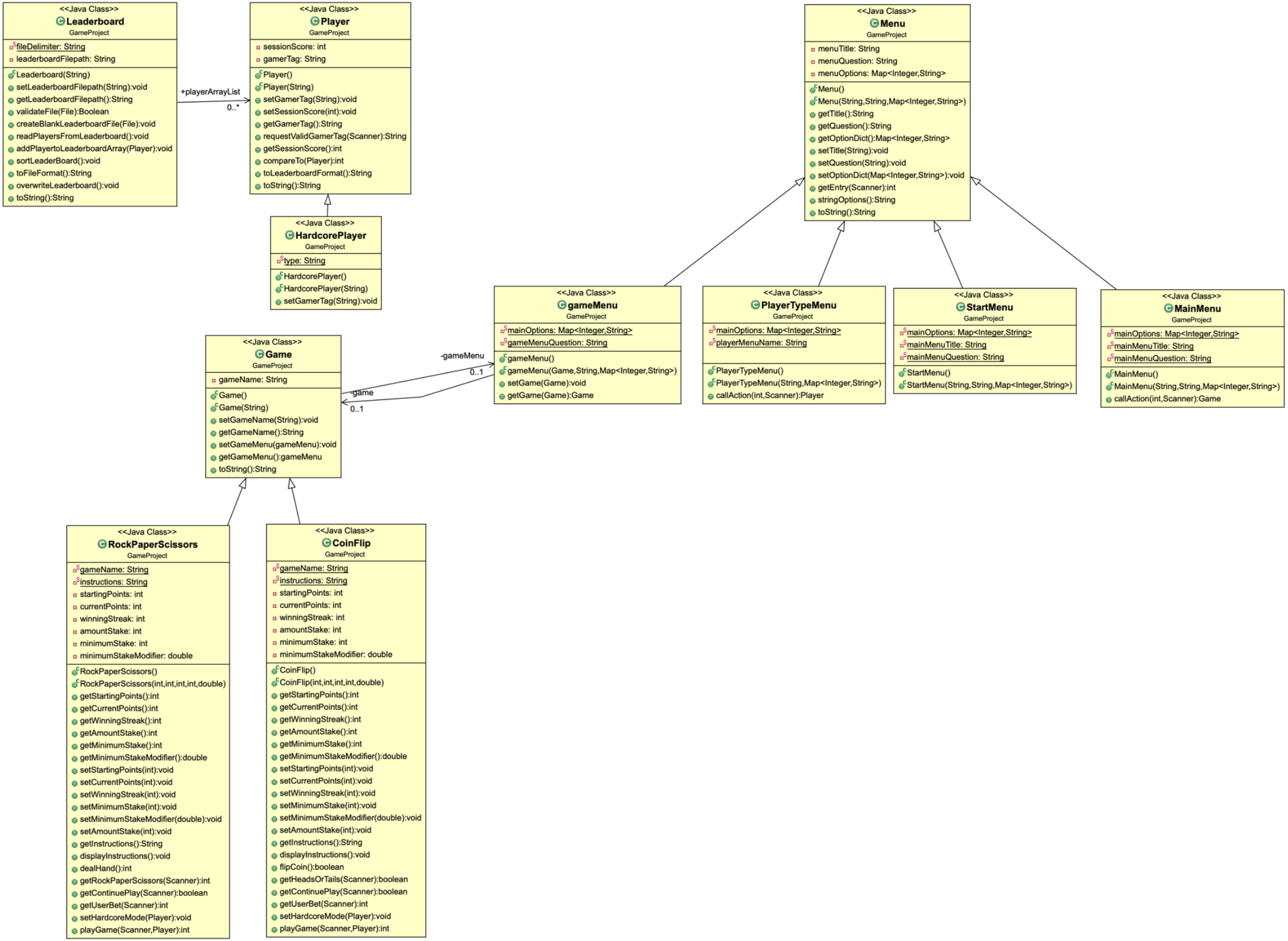
Players are primarily used as a way to keep track of the score over the session for the leaderboard, and to set the difficulty modifier on games. One of the most important elements in implementing the player class was the value of the Comparable interface as it allowed a very easy way to sort the leaderboard values.

Player This is the key player class implemented in the project, and the default player. It consists of a gamertag and score for the session.

harcorePlayer Based on the need to have multiple player types, I elected to add a Hardcore player type which would add a degree of randomness to the setting of the minimum bet volume within the game types, and also affect how the user appears in the leaderboard by specifically modifying their gamertag with a hardcore prefix.

2.1.4 Leaderboards

The Leaderboard consisted of a single class, so will not be broken up into further bullet points. Outside of the game operations and the menu connectivity, some elements of the leaderboard served as tricky components in the initial design, but thankfully the usage of some of the data structures discussed in the module were very easy ways to avoid this as the usage of ArrayLists and Comparators made sorting the leaderboard a very easy task without having to develop complicate methods of sorting and inserting into the leaderboard. This class contains details on the input file and the players. This broadly works by reading in an existing file if it exists, and if not creating one. It splits all of the players in the file by the delimited and adds them to a player array. When saving our players, we add to the array list and then can simply sort the players and overwrite the leaderboard with the headers added to the top.


<<Java Class>>
Leaderboard
GameProject
fileDelimiter: String
leaderboardFilepath: String
Leaderboard(String)
setLeaderboardFilepath(String):void
getLeaderboardFilepath():String
validateFile(File):Boolean
createBlankLeaderboardFile(File):void
readPlayersFromLeaderboard():void
addPlayertoLeaderboardArray(Player):void
sortLeaderBoard():void
toFileFormat():String
overwriteLeaderboard():void
toString():String
+playerArrayList
0..*
<<Java Class>>
Player
GameProject
sessionScore: int
gamerTag: String
Player()
Player(String)
setGamerTag(String):void
setSessionScore(int):void
getGamerTag():String
requestValidGamerTag(Scanner):String
getSessionScore():int
compareTo(Player):int
toLeaderboardFormat():String
toString():String
<<Java Class>>
HardcorePlayer
GameProject
type: String
HardcorePlayer()
HardcorePlayer(String)
setGamerTag(String):void
<<Java Class>>
Menu
GameProject
menuTitle: String
menuQuestion: String
menuOptions: Map<Integer,String>
Menu()
Menu(String,String,Map<Integer,String>)
getTitle():String
getQuestion():String
getOptionDict():Map<Integer,String>
setTitle(String):void
setQuestion(String):void
setOptionDict(Map<Integer,String>):void
getEntry(Scanner):int
stringOptions():String
toString():String
<<Java Class>>
gameMenu
GameProject
mainOptions: Map<Integer,String>
gameMenuQuestion: String
gameMenu()
gameMenu(Game,String,Map<Integer,String>)
setGame(Game):void
getGame(Game):Game
<<Java Class>>
PlayerTypeMenu
GameProject
mainOptions: Map<Integer,String>
playerMenuName: String
PlayerTypeMenu()
PlayerTypeMenu(String,Map<Integer,String>)
callAction(int,Scanner):Player
<<Java Class>>
StartMenu
GameProject
mainOptions: Map<Integer,String>
mainMenuTitle: String
mainMenuQuestion: String
StartMenu()
StartMenu(String,String,Map<Integer,String>)
<<Java Class>>
MainMenu
GameProject
mainOptions: Map<Integer,String>
mainMenuTitle: String
mainMenuQuestion: String
MainMenu()
MainMenu(String,String,Map<Integer,String>)
callAction(int,Scanner):Game
<<Java Class>>
Game
GameProject
gameName: String
Game()
Game(String)
setGameName(String):void
getGameName():String
setGameMenu(gameMenu):void
getGameMenu():gameMenu
toString():String
-gameMenu
0..1
-game
0..1
<<Java Class>>
RockPaperScissors
GameProject
gameName: String
instructions: String
startingPoints: int
currentPoints: int
winningStreak: int
amountStake: int
minimumStake: int
minimumStakeModifier: double
RockPaperScissors()
RockPaperScissors(int,int,int,int,double)
getStartingPoints():int
getCurrentPoints():int
getWinningStreak():int
getAmountStake():int
getMinimumStake():int
getMinimumStakeModifier():double
setStartingPoints(int):void
setCurrentPoints(int):void
setWinningStreak(int):void
setMinimumStake(int):void
setMinimumStakeModifier(double):void
setAmountStake(int):void
getInstructions():String
displayInstructions():void
dealHand():int
getRockPaperScissors(Scanner):int
getContinuePlay(Scanner):boolean
getUserBet(Scanner):int
setHardcoreMode(Player):void
playGame(Scanner,Player):int
<<Java Class>>
CoinFlip
GameProject
gameName: String
instructions: String
startingPoints: int
currentPoints: int
winningStreak: int
amountStake: int
minimumStake: int
minimumStakeModifier: double
CoinFlip()
CoinFlip(int,int,int,int,double)
getStartingPoints():int
getCurrentPoints():int
getWinningStreak():int
getAmountStake():int
getMinimumStake():int
getMinimumStakeModifier():double
setStartingPoints(int):void
setCurrentPoints(int):void
setWinningStreak(int):void
setMinimumStake(int):void
setMinimumStakeModifier(double):void
setAmountStake(int):void
getInstructions():String
displayInstructions():void
flipCoin():boolean
getHeadsOrTails(Scanner):boolean
getContinuePlay(Scanner):boolean
getUserBet(Scanner):int
setHardcoreMode(Player):void
playGame(Scanner,Player):int

3 Discussion

3.1 Design Discussions and Module Impact

As I have broadly discussed some of the key design decisions in the previous sections, this section will be primarily to discuss the usage of the material within the module, and some of the key aspects which ended up impacting the project.

One of the key areas which would add significant value in how this application is designed would be in largely foregoing the object inheritance structure and instead in the implementation of a interface-first design structure. Within the design of my application, there were some initial areas where I was uncertain as to what the requirements were (most notably a leaderboard per game vs a global leaderboard) which impacted some of the design decisions I had initially made regarding the implementation of the game menu, and similarly as I had not intended considered that both games would be betting games yet both inherit from a generic Games class, it would be more sensible for some of the methods shared in common into either a superclass, or preferably an interface. By going with an interface-first approach, there's potentially a greater degree of flexibility which would have helped enable some of these later changes whereas that was not as 'easily' possible when the objects and relation been set.

Another area where module material had a key impact on design was in the usage of getters and setters almost universally to access the variable for objects, and likewise setting the instance variables as private only accessible via getters and setters to allow for an easy ability to ensure consistency in how the values are set and retrieved. The usage of certain data structures to enable a single solution for multiple scenarios without the need to refactor the code, such as the menu options hashmap or the leaderboard arraylist, also helped significantly in creating an 'easy to follow' run method.

Finally, creating modular and insulated classes had significant implications in terms of how few dependencies I built into some classes (e.g. games and Players, and menus and Players) to help allow code to be less intertwined and dependant on specific components of other classes.