CSCM94 Assignment 1
Partial Design Document

Group 9

Swansea University
Prifysgol Abertawe

Group Members: *Mohammed Abdirizaq (914605), Connor Atkins (2005348), Gibson Dzimbiri (920768), Hamish Lawson (864254), Todd Mpeli (849944), Manikodi Prince Christopher (2024860), Bin Xiao (2003795)*

1. CRC Cards

Player	
Author: Connor	
SuperClass: User	SubClasses: -
Responsibilities	Collaborations
Player ID Played Games Number of Wins Number of Losses Win Percentage Favourite Players Add Favourite Remove Favourite	System Game Arcade Leaderboard Data

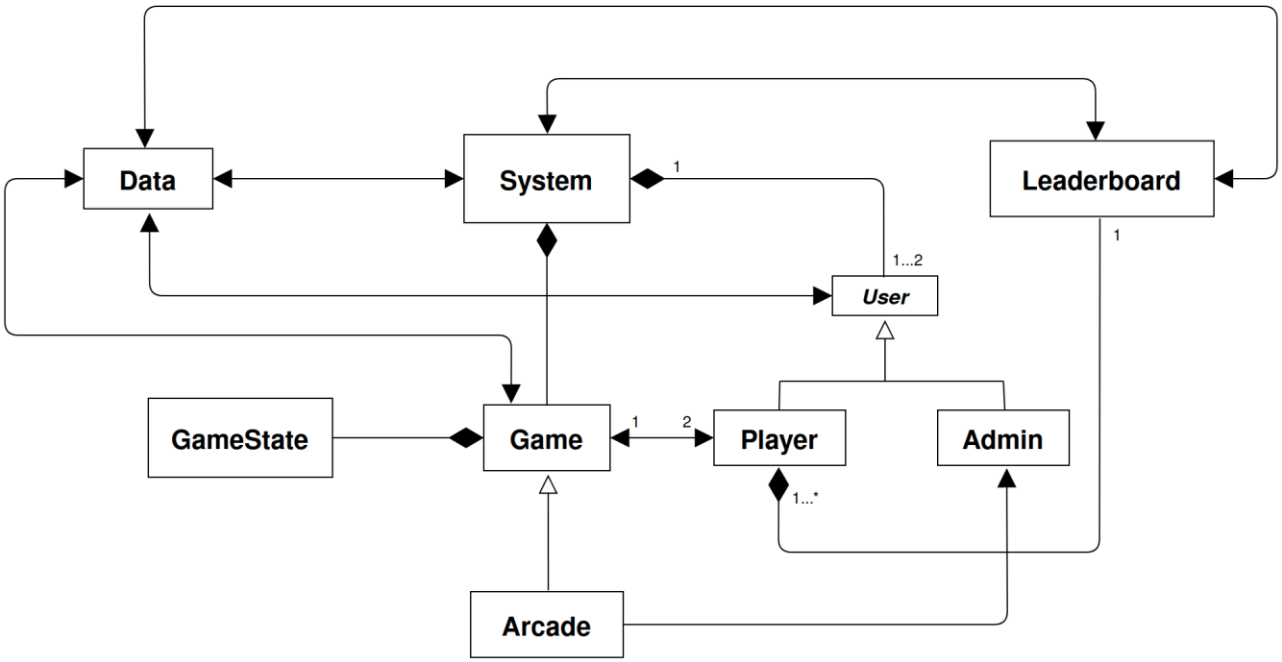
Game	
Author: Todd and Manikodi	
Superclass: -	SubClasses: Arcade
Responsibilities	Collaborations
Current State Players Datetime Choose Starting Player Current Players Turn Update State Is Game Finished Who is Winner	Player System GameState Leaderboard Data

GameState	
Author: Mohammed and Gibson	
SuperClass: -	SubClasses: -
Responsibilities	Collaborations
Player1 Side Player2 Side Player1 Score Player2 Score Is Side Empty	Game

Leaderboard	
Author: Hamish and Bin	
SuperClass: -	SubClasses: -
Responsibilities	Collaborations
Player Ranks View Player Profile Update Leaderboard	Player System Game Data

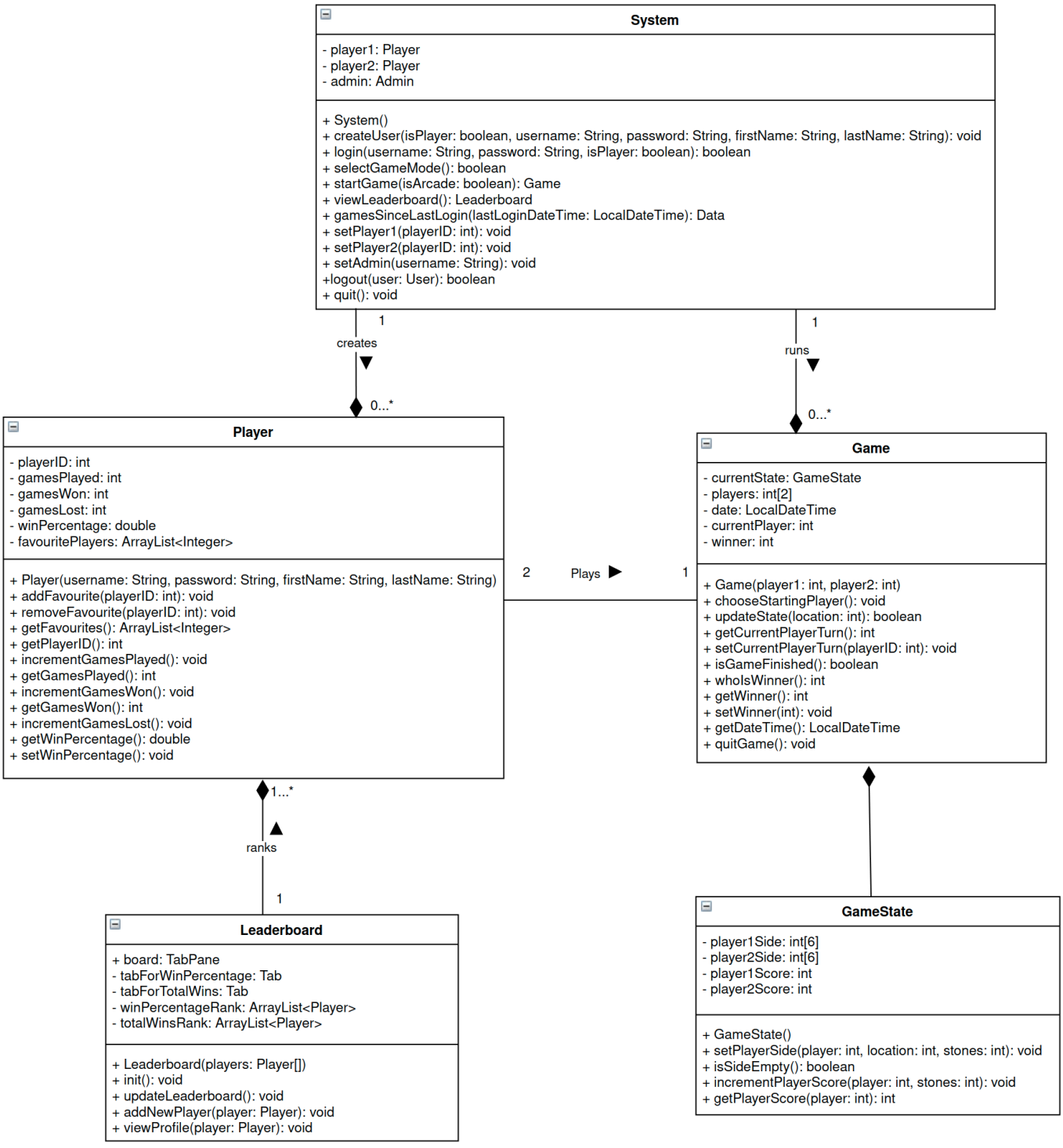
System	
Author: Connor	
SuperClass: -	SubClasses: -
Responsibilities	Collaborations
Create User Login Select Game Mode Start Game View Leaderboard Games Since Last Login	Player Admin Game Arcade Leaderboard Data

2. Class Collaboration Diagram



The above class collaboration diagram shows the high-level overview of the whole application with all the classes and the collaborations between them. The CRC cards in Section 1 show the responsibilities of a selection of the most important of these classes. The System class is at the centre of the application, allowing users to log in to the system and then select which game mode to play, as well as being able to view the Leaderboard. On which they will have access to the other players' profiles, including the ability to favourite these players and view a list of the games that have been played since the last time they logged in. The Data class will be responsible for storing all the information about users and the games that have been played, to prevent this information from being lost when the application is closed. The Game class is responsible for initialising and playing out a traditional game between two players. It will return the winner of the game and update the stats in the players' profiles that took part in the game. The Game class is composed of the GameState class, where GameState is a representation of the state of the board at any given time in the game and will get updated with every move that a player takes. This is a composition as GameState will not exist outside of any game. The Player class will be responsible for storing the stats of the games that a Player has taken part in. The Leaderboard class is a composition of Player objects, as there must be at least one player to have a Leaderboard, hence the one-to-many multiplicity. It will rank these players by both win percentage and total wins. It also has an association with System so that the Leaderboard can be viewed from the main menu. As well as an association with the Data class, in order to initialise and rank the players from games that have be played previously. Finally, Arcade is a subclass of Game and will inherit a traditional game's functionality, then extending it to implement the different rules and special moves of that mode. It has an association with the Admin class, so an admin can track the statistics of the special moves and stones that have been used. Admin will also have to approve any new player created in the System class before that player can login and play the game.

3. Class UML Diagrams



From the UML diagrams of the classes chosen to be the focus of the partial design report, it can be seen in more detail the attributes and methods that these classes will have to implement to meet the responsibilities they have according to the CRC cards in Section 1. The following section will explain how some of the most complex and interesting methods will work to achieve this functionality.

3.1 System – startGame(isArcade:boolean) and gamesSinceLastLogin(lastLoginDateTime: LocalDateTime)

Starting with the System class, a user will need to log in upon starting the application and will do so by entering their username and password. Once this has been done, they can select a game mode and will then be asked if they wish to play against the computer or another player. If another player is selected, that player will be asked to log in. The startGame() method will take in the choice of game mode as a boolean, True if Arcade mode has been selected or False if Traditional. startGame() will start the game by either calling the constructor for the Game Class or Arcade class, depending on the boolean argument. It will be able to do this through the composition collaboration it has with Game and provide the Game constructor with the PlayerID's of the two players that will be playing the game that it has access to from the composition collaboration it has with Player. A player that is logged into the system can also see the list of games that have been played since they last logged in through the gamesSinceLastLogin() method. The method will retrieve this list by querying the data class using the lastLoginDateTime attribute of Player to return the games played since that time, which will be possible as all games will have a timestamp from the date attribute in the Game class. The data class will return all games with a timestamp after this, with the two players that played that game and the winner for each of these games that have been played.

3.2 Game – updateState(location: int) and whoisWinner()

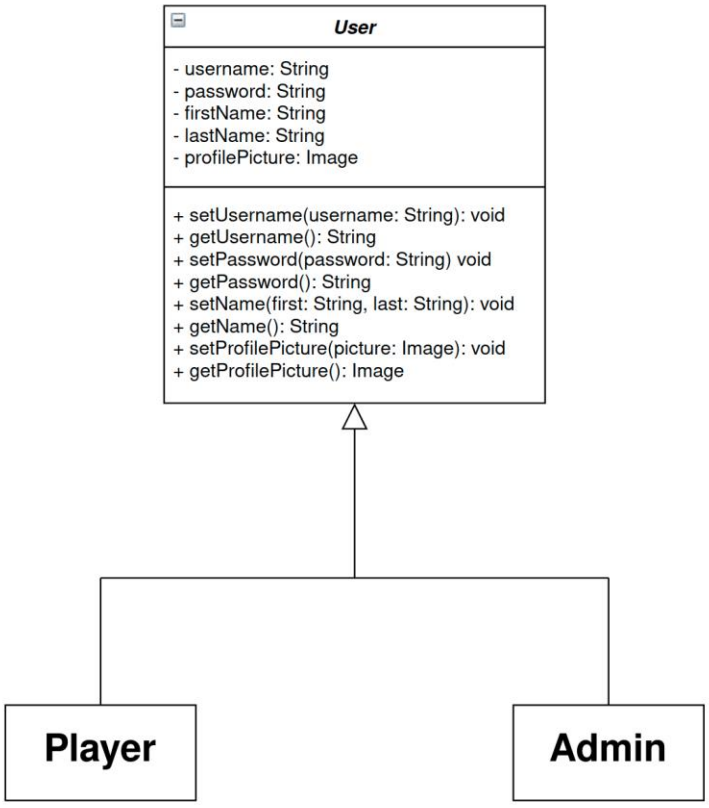
The most interesting method in the Game class is the updateState() method that updates the currentState attribute to change the board's state after a player has made a move. It takes one argument: an integer representing the hole on that player's side of the board that they would like to make their move from. The currentState will then be accessed via the composition collaboration that Game has with GameState, and that playerSide array will be accessed at the location provided. Firstly, it will be checked that the value is not zero to determine that it was a valid move. If it were not valid, an exception would be raised, and the player will be asked to make another choice. If the move were valid, the number at that location in the array would be set to zero to represent all the stones being taken from that hole. The number of stones present in that hole will be stored in a stones variable that will then be used to loop over the two arrays and that players' score attribute (which represents their mancala hole), distributing one stone into each hole. When the stones variable reaches zero, it will check if the last stone had been added to the players' score. If this is the case, the current player would not be changed, allowing that player to make another turn as per the rules. Alternatively, if the last stone had been added to a hole that now has a value greater than one (which would represent a hole that already had stones in), the loop would be reentered at the next position where there are stones and run again. Finally, after the turn is completed, it will call the isGameFinished() method to check if the game is over. It does this by calling the isEmpty() method from GameState to check if either of the sides of the board is empty, which will determine that the game is over. If so, the side of the board that is not empty will have the stones added to that players' score, and then the updateState() method will return True to indicate the game is over. When the game is over the

whoIsWinner() method will be used to return the player with the highest score, the winner attribute will then be set to the PlayerID of this player using the setWinner() method. When a game is over, the IDs of the two players and the winner and date of the game will be sent to the Data class to record the game. The two players who took part in the game will have their stats updated, and a call will be made to the updateLeaderboard() method to update the ranks in the Leaderboard.

3.3 Leaderboard – updateLeaderboard()

The updateLeaderboard() method in the Leaderboard class will be used after any game has been played to re-rank the players after the stats in their profile will have changed. It will have access to the gamesWon and winPercentage attributes of the Player objects that make up the Leaderboard through the composition collaboration it has and will sort the players by each of these and reorder the totalWinsRank and winPercentage ArrayList attributes accordingly.

4. Inheritance Hierarchy



In the application, we have decided to use an inheritance hierarchy for the users of the system. There is an abstract User class that Player and Admin will inherit from. We chose to have this as an inheritance

relationship as both players and admin for the system have common attributes such as username, password, firstname, lastname and profile picture. Using inheritance means both Player and Admin can inherit these attributes and the methods related to these attributes and then implement the responsibilities that differ between them in their own classes. User is an abstract class as there will never be a user of the system that is not either a Player or an Admin.