CS-230 Partial Design

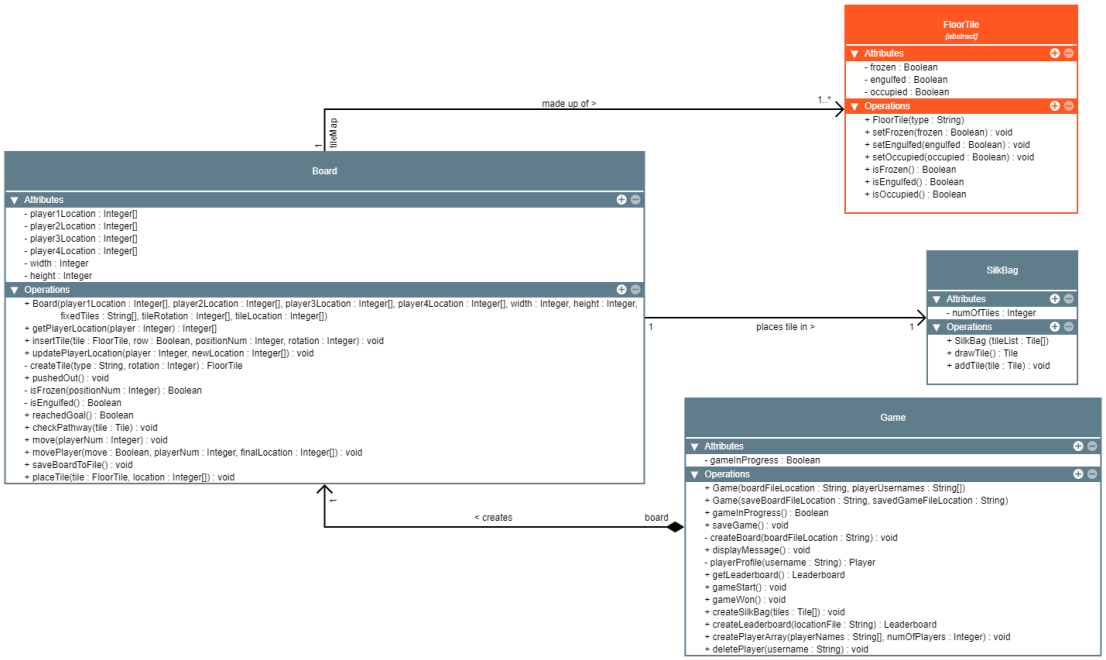
Group 16

2nd November 2020

1 Candidate Classes, Responsibilities & Class Diagrams (UML)

1.1 Board

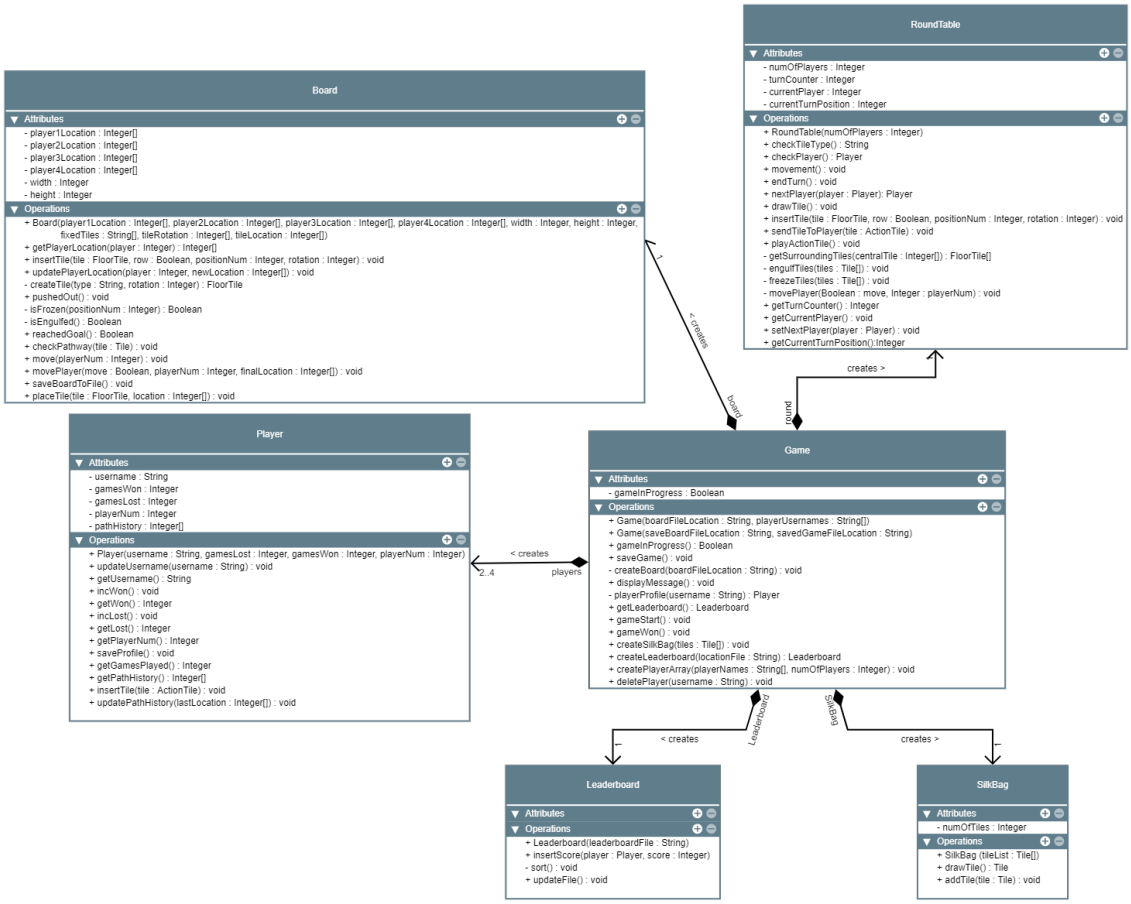
Board	
The main board of the game. Holds tiles and player locations. Moves player on board	
Bartosz Kubica & Marius Antemir	
Superclass	**Subclasses**
None	None
Responsibilities	**Collaborations**
Create board with fixed tiles	Game
Store tiles currently in play	Tile
Insert new tile board and discard to SilkBag	Tile, SilkBag
Store Player Locations	Player
Move player on board	RoundTable



The Board class has collaborations with the abstract class FloorTile, the SilkBag Class, and the Game class. The association between the FloorTile class is that the Board is made up of FloorTiles. The multiplicity is 1..* (1 to many) because the board needs to have at least 1 FloorTile. The association between the Board class and SilkBag is that the Board places the discarded tiles back into the SilkBag, therefore the SilkBag class does not need access to the Board class. This multiplicity is 1. There's an association and composition relationship between the Board class and Game class. This is because the Game creates the Board for the game and if the Game is deleted then the Board is deleted too.

1.2 Game

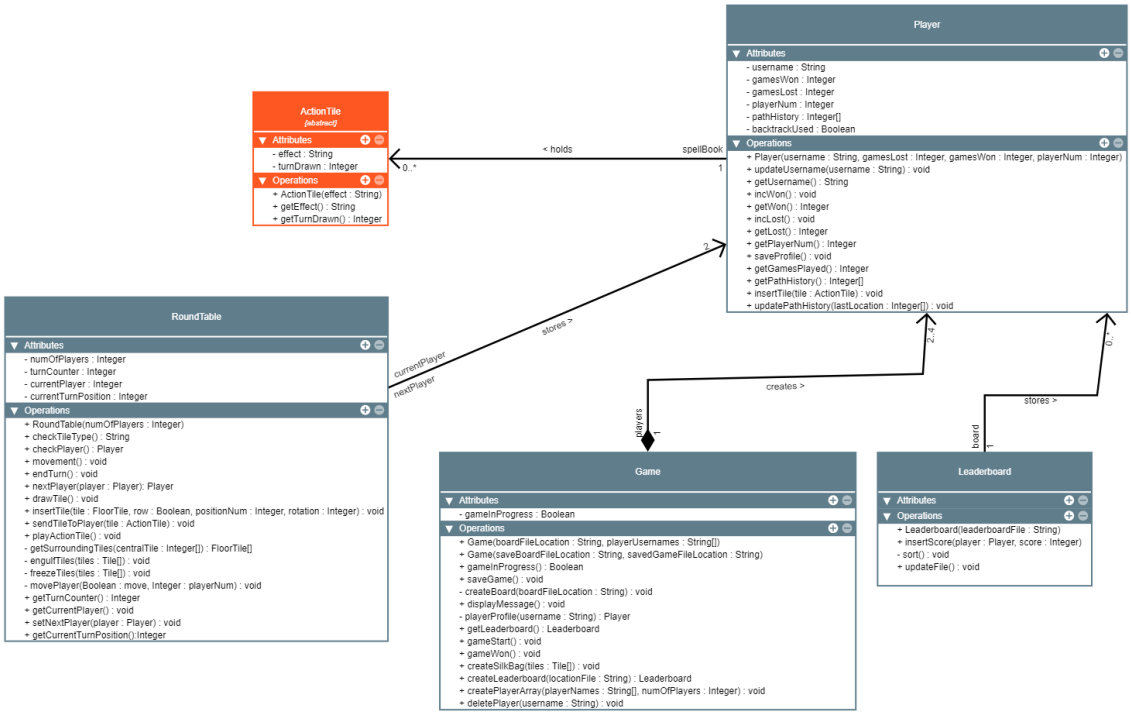
Game	
Initialises the game. Reads board file, player profile, leaderboard and starts game.	
Bartosz Kubica & Marius Antemir	
Superclass	**Subclasses**
None	None
Responsibilities	**Collaborations**
Initialises board from selected file	Board
Gets and creates player profiles	Player
Gets and creates leader board file	Leaderboard
Starts the game	RoundOrder



The Game class has composition and association between the Board, Leaderboard, SilkBag, RoundTable, and Player classes. This is because it creates instances of all the classes necessary for the gameplay. The multiplicity of the collaborations is 1 because the Game class creates 1 instance of the classes and needs access to only 1 version of them. This is an exception with the Player class as the multiplicity of that relationship is 2..4 because there can be between 2 to 4 players in the game.

1.3 Player

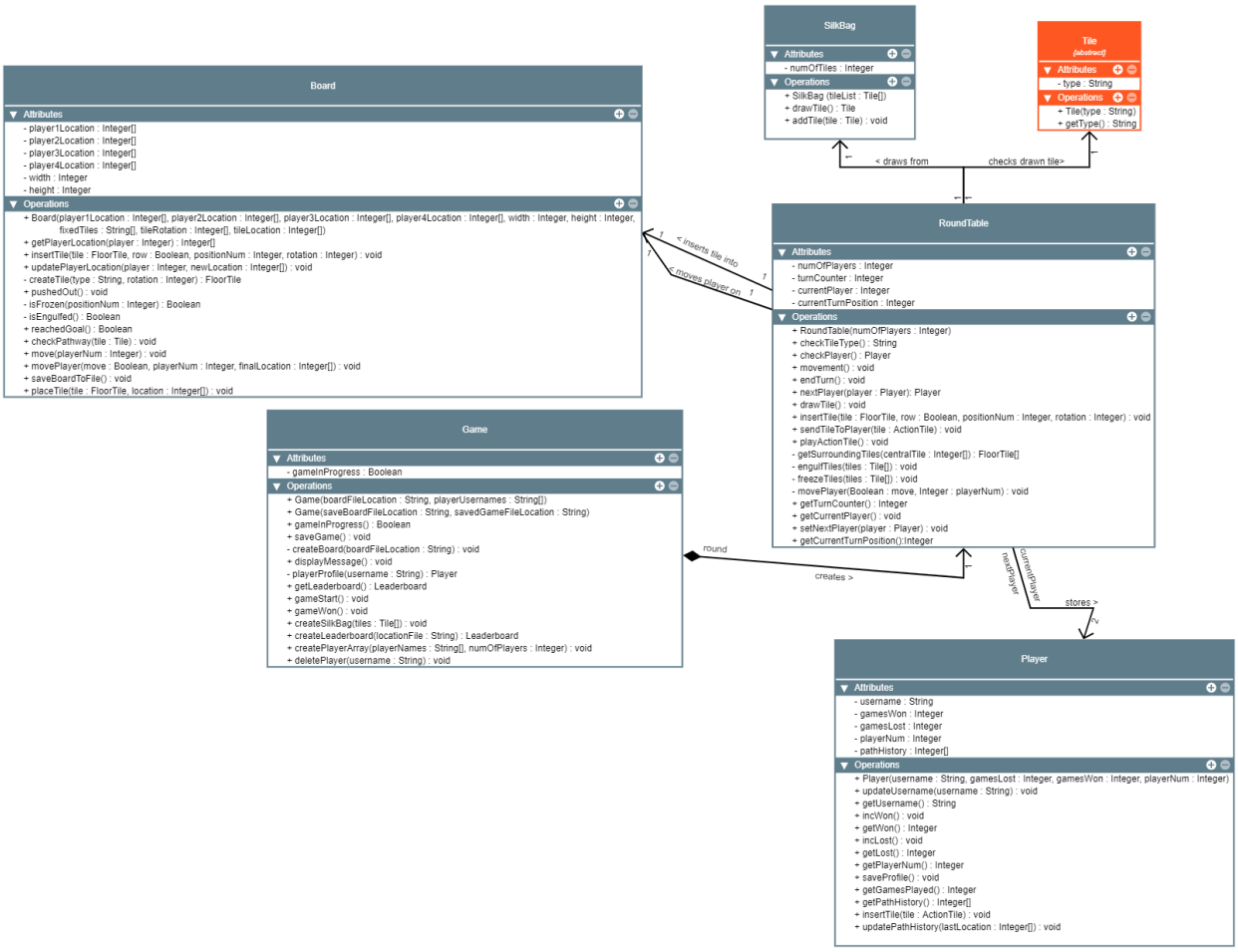
Player	
Creates a profile for player	
Chloe Thomas & Michelle Bhaskaran	
Superclass	**Subclasses**
None	None
Responsibilities	**Collaborations**
Creates player profile	
Updates player profile	
Saves player profile	
Stores action tiles	ActionTile
Stores player path history	



The Player class has collaborations with several different classes. It has an association with ActionTile. This is because the Player class must hold objects of any of the subclasses of ActionTile in the player's hand (spellBook). The navigability is from Player to ActionTile as the ActionTile objects do not need access to the Player class. The multiplicity is 0..* (zero to many) because the player would initially have no ActionTiles and could get many during the course of the game. It also has an association with the RoundTable class because it stores Player objects. The navigability is from the RoundTable class to the Player class because the Player class does not need to have access to any objects of the RoundTable class. The multiplicity is 2 because the RoundTable will store exactly 2 objects of Player, currentPlayer, and nextPlayer. There's an association and composition between the Player class and the Game class. This is because the game cannot exist without the players in the game. The players are initialized in the Game class so if the game stops the player objects are destroyed. The collaboration creates new Player objects. The multiplicity is 2..4 because the game is played by a minimum of 2 players and a maximum of 4 so the Game class stores any number in that range. Finally, there is a collaboration with the Leaderboard class as it stores Player objects. The association is from the Leaderboard class to the Player class because the Player objects don't need access to the Leaderboard and the multiplicity is 0..* (zero to many) because the Leaderboard can be empty in the first instance and then can hold many entries.

1.4 Round Table

RoundTable	
Each players turn. Draw a tile, insert tile, play an action tile and move along board	
Jimmy Kells & Surinder Singh	
Superclass	**Subclasses**
None	None
Responsibilities	**Collaborations**
Allow user to draw a tile	SilkBag
Allow user to insert floor tile	Board
Allow user to play an action tile	ActionTile & Board
Store current and next player	Player
Send action tile to player	Player
Allow user to move along board	Board



The RoundTable class has collaborations with several classes. It has an association with the Tile abstract class where it can check the tile type, the SilkBag class where it can draw a new tile, and the Board class where it can insert a new tile and move the player. All these collaborations have a multiplicity of 1 because they only work with one object at a time. There is an association and composition with the Game class as it creates the RoundTable. This is because the Game cannot exist without the RoundTable and if the Game is deleted then the RoundTable is deleted too. An association with the Player class allows the RoundTable to store the current and next player which is a multiplicity of exactly 2.

2 Five Complex Behaviours

From the classes RoundTable, Board, Game and Player we have chosen 5 complex behaviours and explained them below.

The drawTile behaviour in the RoundTable class gets a tile from the silk bag and determines whether the tile should be placed onto the board or into the player's spellBook. This means it calls drawTile function in the SilkBag class to get a random tile from tileList. The function then checks the tile type by calling checkTileType. If the tile chosen from the SilkBag is an action tile, then it will call sendTileToPlayer which will insert the tile to the spellBook of the player. If the tile is a floor tile it will prompt the user to insert the tile to a row or column and send the tile along with the location of the tile to the Board class ready for insertion. Once this is complete it will run the playActionTile function prompting the player to either skip or play an action tile. It has access to the data through the SilkBag class. It can call the drawTile function in the SilkBag class which will get a tile at random from the tileList. Once it gets a tile it will be able to get the type of that tile and determine where to go next via the main Tile class. This behaviour can be implemented because it only calls other functions within it depending on whether the tile is an action tile or not.

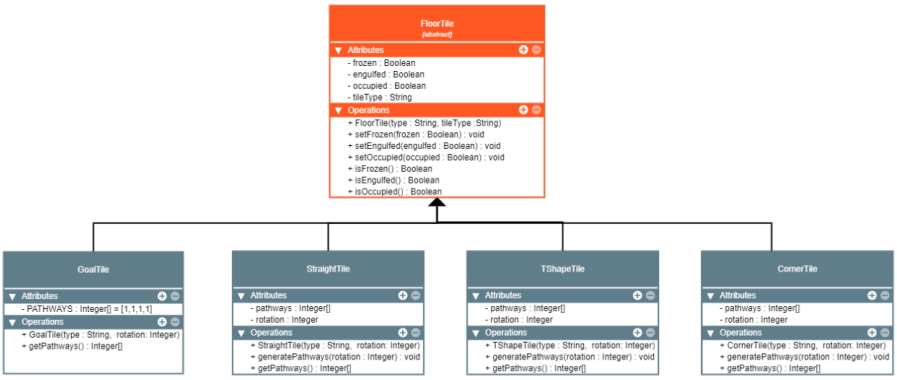
The saveProfile behaviour in the Player class saves the players current statistics and other information to a file on the system. It does this by gathering all the information from the player object. It does not need access to any other classes for this behaviour. The function will then write to a text file the information it has collected. This means that the profile can be accessed later. This can be implemented using the FileWriter library. It accesses data such as username, games played, games won, and games lost from itself. The object holds all this information, so it does not need any collaboration between other classes. This can be implemented as it's just taking the current objects information and writing it to a file.

The displayMessage behaviour in the Game class will display the daily message using the API specified in the functional requirements. The function will use the prebuilt class HttpUrlConnection. It will request the message from the website and then interpret it so that it can be displayed on the screen. The data necessary can be accessed via the API for the assignment. This can be realised when creating the system because most members of the group have background knowledge on HTTP requests in Java.

The saveGame behaviour in the Game class will save the current game state. It will do this by first collecting all the information required to save the game. It will call the function saveBoard in the Board class which will save the current board state in a file. The function will get all the players, their spellBooks, their path history, the Silk Bag contents and the round table iteration. All this information will then be written to another file using the FileWriter library. These two files will hold all the information needed to reload the game. The function has access to the Board, SilkBag, RoundTable and Player classes meaning it can pull all the necessary data. This can be coded in the system as it only needs to collate the data and write it to a file.

The behaviour move in the Board class has the responsibility of moving the player. It gets called from the movement function in RoundTable. It will first check whether any of the north, east, south or west tiles are engulfed by calling the function isEngulfed. It will then check whether there is a pathway to the next tile that is not on fire. It will do this using the checkPathway function. If there is a pathway it will display it to the user to choose which one to go to. Once the user chooses where to go the move function will move the player and update their location. To check whether the tile is engulfed it has access to the FloorTile class where an attribute engulfed stores whether the tile is engulfed. This can be implemented as it only calls other functions and then moves the player to another tile.

3 Hierarchy Descriptions



These are some of the classes we have decided to put into a hierarchy. We have decided to do this because it would make the smaller floor tile classes more manageable. In the Floor Tile class, we have all the

parameters that are the same across all the floor tile pieces. This class is abstract because we do not want to make instances of the FloorTile class but we still have attributes and behaviours that all floor tiles share. This lets us have less redundant data across all the tiles and makes us less likely to make errors. By doing this we can also reuse any code from the parent superclass which we will need across all the subclasses.

An example of this would be the setFrozen method in the FloorTile abstract class. This is in the superclass because all floor tiles can be set on fire by the fire action tile. When the action tile is played that method can be called using any FloorTile. This is because the playActionTile method does not need to know what type of floor tile it is but instead just needs to know that it is a floor tile.

The subclasses then store attributes and methods that allow them to be the more specialised tiles. For example, the corner tile stores the orientation so that we know which 2 sides are available pathways for players to go through. The goal tile however does not need this method, so we have decided to only have it in the other subclasses. The goal tile does not need this method as no matter which orientation it is in you can enter from the same side as you can enter it from all 4 cardinal directions no matter the rotation.

Initially, we decided not to have any inheritance for this section. This meant that all the behaviours and attributes of the action and floor tiles were all in one class. This caused some issues when designing the UML because the freeze behaviour on the action tile had to call the setFreeze behaviour on the floor tile. This inheritance worked because all the floor types do slightly different things and store different information.

4 Level File Format

The level file needs a specified format to be able to create level layouts. The design needs to be in a basic ASCII text file. There will be a wide selection of level files that the user will be able to choose from. The files will all have the same format so that the game can read it efficiently. The format of the file will be:

- Line 1: width of the board, the height of the board
- Line 2: number of fire tiles, number of ice tiles, number of double move tiles, number of backtrack tiles
- Line 3: number of straight tiles, number of T junction tiles, number of corner tiles
- Line 4: number of fixed tiles
- Lines 5 to x: describe a tile in the specified format: posx, posy, tile type e.g. CORNER, rotation 0 = north, 1 = east, 2 = south, 3 = west
- Line x + 1: goal tile position x, y
- Line x + 2 : number of players
- Line x + 3: Player 1 location x, y
- Line x + 4: Player 2 location x, y
- Line x + 5: Player 3 location x, y
- Line x + 6: Player 4 location x, y

Line 1 will show the width and height of the board. This allows the tileMap to be initialised and created ready for the tiles to be inserted. If the user tries to move beyond the board or a player is pushed out due to an inserted tile it can move the player back on the board.

Line 2 has the number of action tiles split up into the 4 different types, Ice, Fire, Backtrack and Double Move.

Line 3 is the floor tiles that can be placed by the players, split up by the different types, straight, T junction and corner tile. These are the tiles that will go to the silk bag. The lines will hold the number of these tiles in that order. This will mean the system can read each number separated by commas and initialise each action tile.

Line 4 is just the number of fixed tiles. This will be used to help read the rest of the file. It allows us to see how many lines below will be used to describe the fixed tiles. The next lines are dedicated to a fixed tile per line. The fixed tiles lines are stored in a specified format separated by commas which is the position of the tile in x axis, position of the tile in y axis, tile type and rotation. The line below all of the fixed tiles will be the goal tiles coordinates. It does not need a rotation because it has paths on all sides.

The next 2 to 4 lines are each player's starting location. They will be in the cartesian format of the x coordinate first and y coordinate second. This will allow the game to insert each player on the board and make each player move.