UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Practical 6 Specifications

Release Date: 29-09-2025 at 06:00

Due Date: 03-10-2025 at 23:59

Late Deadline: 04-10-2025 at 00:59

Total Marks: 115

Read the entire specification before starting with the practical.

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you begin coding.***
- This assignment should be completed individually.
- **Every submission will be inspected with the help of dedicated plagiarism detection software.**
- Be ready to upload your assignment well before the deadline. There is a late deadline which is 1 hour after the initial deadline which has a penalty of 20% of your achieved mark. **No extensions will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or structure).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use C++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software to generate submitted code is strictly forbidden and will be considered as plagiarism.

2 Overview

In this practical, players in a game show are involuntarily placed into an arena with one entrance and one exit. Some tiles may be dangerous and some players may sabotage other peoples' progress.

3 Your Task:

You are required to implement the following class diagram illustrated in Figure 1. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

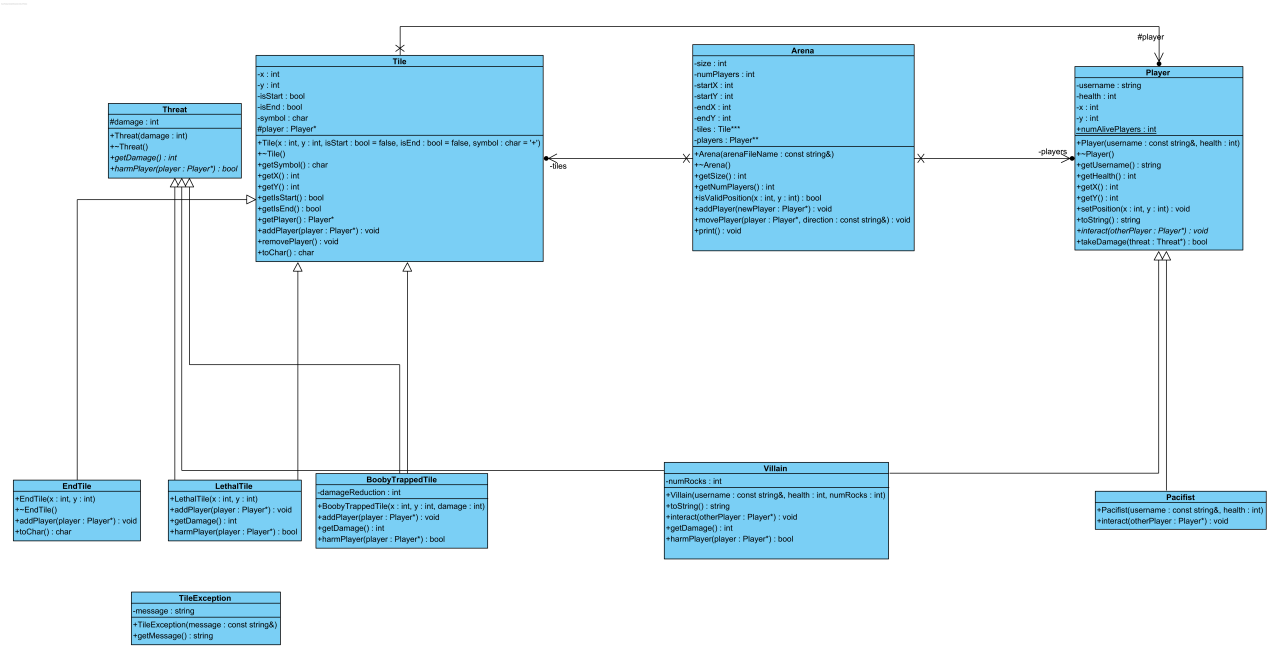

Figure 1: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214. The member functions written in italics are pure virtual functions. Member variables which are underlined are static variables.

3.1 TileException

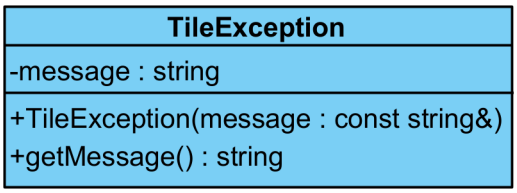

Figure 2: UML Diagram of TileException

3.1.1 Members

- TileException Member 1
  - This member variable will contain a message to provide further details on what caused the Exception object to be thrown.

3.1.2 Functions

- TileException Function 1
  - This function should set the *message* member variable to the passed in parameter.
- TileException Function 2
  - This is a constant function.
  - This function should return the *message* member variable.

3.2 Threat

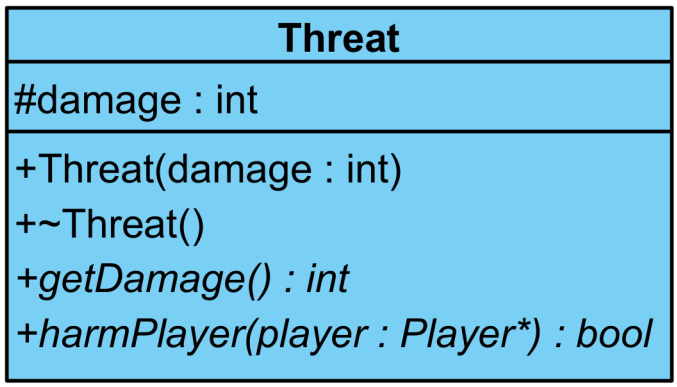

Figure 3: UML Diagram of Threat

3.2.1 Members

- Threat Member 1
  - This member variable indicates how much damage a threat inflicts on its victims.

3.2.2 Functions

- Threat Function 1
  - This function should initialise the damage member variable appropriately.
- Threat Function 2
  - This is a virtual function.
  - This function should deallocate memory appropriately.
- Threat Function 3
  - This is a pure virtual function.
  - Classes that inherit from this class will have different logic for determining their damage.
- Threat Function 4
  - This is a pure virtual function.
  - If the Threat was able to kill the player (passed in parameter), this function should return true. Otherwise, this function should return false.

3.3 Player

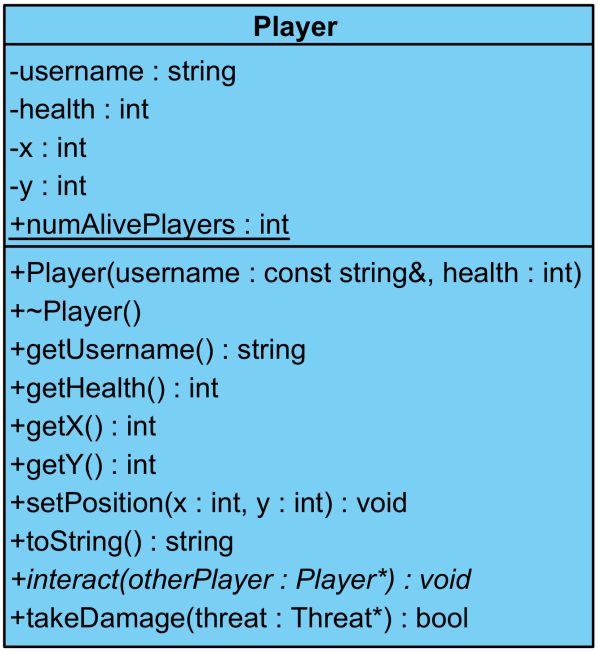

Figure 4: UML Diagram of Player

3.3.1 Members

- Player Member 1
  - This member variable will be used to identify players.
  - It must begin with a capital letter and consist of at least 2 characters.
  - Usernames do not have to be unique to players.
- Player Member 2
  - This member variable will store the player's health.
  - It should never exceed a value of 100.
- Player Member 3
  - This member variable indicates the row where the player is located.
  - A value of -1 means the player is not in the arena.
- Player Member 4
  - This member variable indicates the column where the player is located.
  - A value of -1 means the player is not in the arena.
- Player Member 5
  - This is a static variable.
  - It must be initialised in the ".cpp" file with a value of 0;
  - It should increase by 1 every time a Player is created.
  - It should decrease by 1 every time a Player is killed.

3.3.2 Functions

- Player Function 1
  - The username should be set correctly.
    - You may assume that a valid username will always be passed to this function.
  - The player's health should be set correctly.
    - The passed-in parameter must be a positive integer value.
    - If the passed-in value for health is not positive, the player's health should be set to 50.
    - If the passed-in value for health is greater than 100, the player's health should be set to the highest valid value (100).
- Player Function 2
  - This is a virtual function.

  - This function should deallocate any dynamic memory.

- Player Function 3
  - This is a constant function.
  - This function should return the player's username.

- Player Function 4
  - This is a constant function.
  - This function should return the player's health.

- Player Function 5
  - This is a constant function.
  - This function should return the row where the player is located.

- Player Function 6
  - This is a constant function.
  - This function should return the column where the player is located.

- Player Function 7
  - This function should set the player's position (row and column values).

- Player Function 8
  - This is a constant function.
  - This is a virtual function.
  - This function should show information related to the player.
  - Each property will be displayed on a **separate line**.
    * Firstly, the player's username will be displayed in the format.
      "Username: <username>"
    * Next, the player's health will be displayed in the format.
      "Health: <value>"
    * Lastly, the player's position will be displayed in the format.
      "Position: (<x-value>,<y-value>)"
  - Example of output:

```
Username: Jester
Health: 80
Position: (3,4)
```

- Player Function 9

  - This is a pure virtual function.
  - This function allows players to interact with one another.

- Player Function 10
  - This is a virtual function.
  - This function must reduce the player's health when they are harmed by a threat.
  - A player is killed when their health drops to zero.
  - The player's health must reduce by the Threat's damage value.
    * If the inflicted damage by the threat is greater than the player's health, the player's health must be set to zero.
    * Whenever a player's health reaches zero, the *numAlivePlayers* static member variable must be reduced by one.
    * The following must be output when a player dies (ending with a newline).
      "<Username> has perished"
    * Example of output:

```
Jester has perished
```

  - If the player is harmed but not killed:
    * The following must be output (ending with a newline):
      "<Username> took <x> damage and now has <h> health"
    * Example of output:

```
Jester took 10 damage and now has 40 health
```

  - This function should return `true` if the player that takes damage is killed; otherwise, it should return `false`.

3.4 Pacifist

This class publicly inherits from the Player class. Pacifists simply wave at other players throughout the game.

Figure 5: UML Diagram of Pacifist

3.4.1 Functions

- Pacifist Function 1
  - This function should call the parent class constructor with the correct arguments.
- Pacifist Function 2
  - This function should handle interactions by Pacifists.
  - It should do nothing if the other player is **NULL** or a player tries to interact with itself.
  - The function should output the following (ending with a newline):
    "<Username> wave at <OtherUsername>"
  - Example of output:

```
Jester waved at Tom
```

3.5 Villain

This class publicly inherits from both the Player and Threat classes.

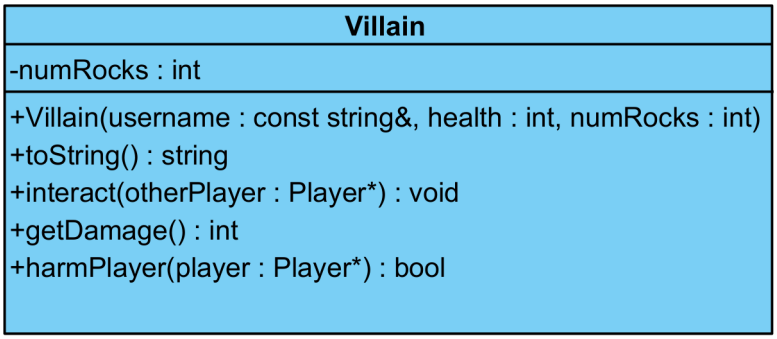

Figure 6: UML Diagram of Villain

3.5.1 Members

- Villain Member 1
  - This member variable determines how many rocks the player possesses.

3.5.2 Functions

- Villain Function 1
  - This function should call the appropriate parent constructors with the relevant parameters.
  - Villains do 10 damage.
  - The player's number of rocks member variable must be initialised correctly.

- Villain Function 2
  - This is a constant function.
  - This function should call the parent class's version of the `toString function` and append additional information about the player.
  - This function should include the player's number of rocks and damage in the result.
  - The additional information should be displayed in the following format
    (on separate lines):
    "Rocks: 5"
    "Damage: 10"
  - Example of output:

```
Username: Jester
Health: 80
Position: (3,4)
Rocks: 5
Damage: 10
```

- Villain Function 3
  - This function should call the `harmPlayer()` function, passing the *other* player pointer.
  - If the passed-in parameter is **NULL** or the parameter is equal to *this*, this function should do nothing.
- Villain Function 4
  - This function should return the player's damage.
- Villain Function 5
  - Villains should be able to harm, but **not kill** other players.
    - This member function should not inflict harm upon a player if the player has 10 health or less.
    - It should return *false* in this case.
  - A villain's rocks cannot be retrieved or replenished
    - Villains throw one rock at a time and never miss.
    - Once they have run out of rocks, a villain cannot harm other players any more.
  - This function should return *true* if the other player was successfully hit; otherwise, it should return *false*.

3.6 Tile

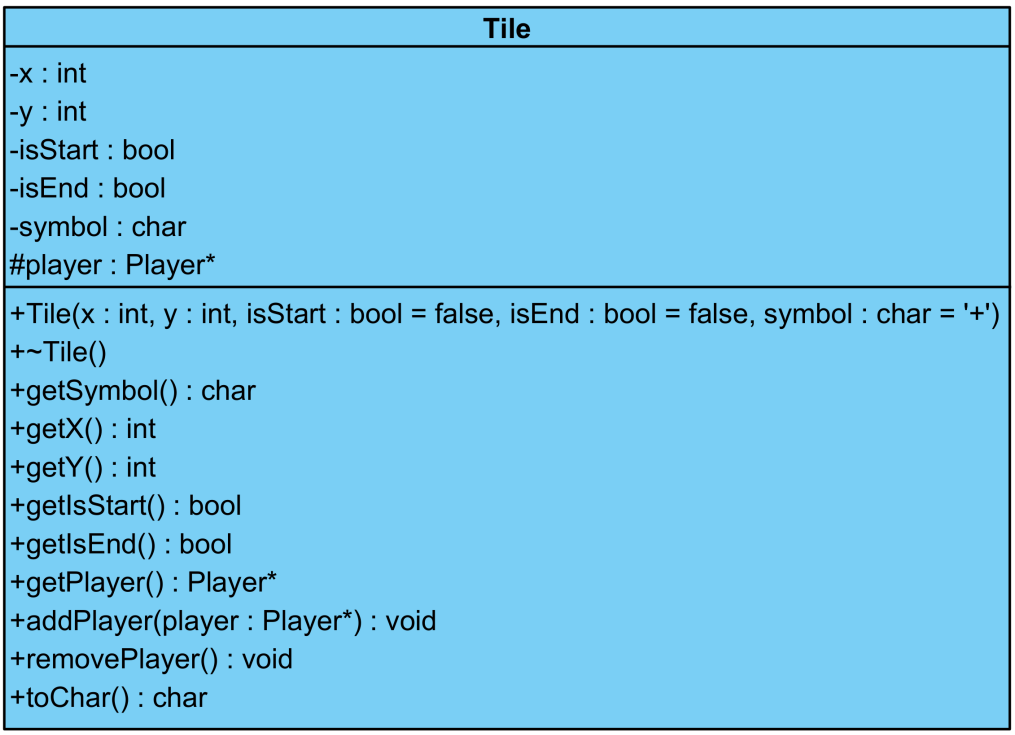

Figure 7: UML Diagram of Tile

3.6.1 Members

- Tile Member 1
  - This member variable indicates the row where the tile is located.
- Tile Member 2
  - This member variable indicates the column where the tile is located.
- Tile Member 3
  - This member variable indicates whether a tile is the starting tile for the arena.
- Tile Member 4
  - This member variable indicates whether a tile is the ending tile for the arena, where players can escape.
- Tile Member 5
  - This member variable is used to display the tiles when printed.
  - Regular tiles have a '+' symbol.
- Tile Member 6
  - This member variable refers to the player that is currently standing on the tile.

3.6.2 Functions

- Tile Function 1
  - This function should initialise the properties of a tile according to the passed parameters.
  - In your '.h' file, you should specify the following default values:
    - isStart: false
    - isEnd: false
    - symbol: '+'
  - If the tile is a starting tile, the symbol should be set to 's'.
  - Player must be set to NULL.
- Tile Function 2
  - This is a virtual function.
  - This function should perform memory management for Tiles.
  - If the tile has a player on it, the player must be deleted.
- Tile Function 3
  - This is a constant function.
  - This function should return the tile's symbol.
- Tile Function 4
  - This is a constant function.
  - This function should return the tile's row value.
- Tile Function 5
  - This is a constant function.
  - This function should return the tile's column value.
- Tile Function 6
  - This is a constant function.
  - This function should state whether or not the tile is a starting tile.
- Tile Function 7
  - This is a constant function.
  - This function should state whether or not the tile is an ending/exit tile.
- Tile Function 8
  - This is a constant function.

  - This function should show return the player currently on the tile.

- Tile Function 9
  - This is a virtual function.
  - This function should add a player onto the tile.
  - The function should also call `setPosition` on the player, passing the tile's position-values as parameters.
  - If passed-in parameter is NULL, the function should throw a `TileExcecption` with the message "Player is NULL".
  - If tile already has a player on it, the function should throw a `TileExcecption` with the message "Tile occupied".

- Tile Function 10
  - This is a virtual function.
  - This function should set the *player* property of the tile to NULL.

- Tile Function 11
  - This is a constant function.
  - This is a virtual function.
  - This function should return the tile's *symbol* if there is no player on the tile.
  - Otherwise, the function should return the first letter of the player's username.

3.7 EndTile

This class inherits publicly from the Tile class.

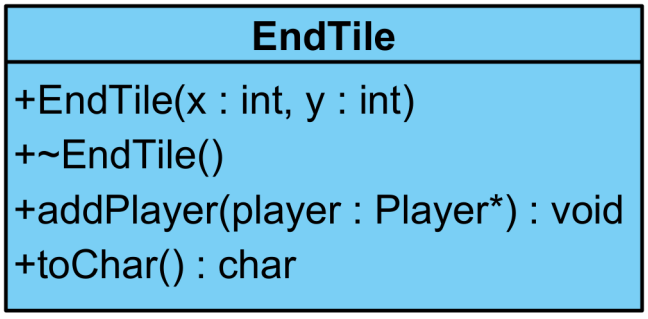

Figure 8: UML Diagram of EndTile

3.7.1 Functions

- EndTile Function 1
  - This function should call the parent constructor with the corresponding x and y values.
  - End Tiles cannot also be start tiles.
  - The symbol for these tiles is 'e'.
- EndTile Function 2
  - This function should do memory management for the tile.
- EndTile Function 3
  - This function should call the parent class's version of the `addPlayer` function.
  - The player's position must then be reset to *(-1,-1)* to indicate they are no longer in the arena.
  - The following should be output (ending with a new line):
    "<Username> has escaped"
  - The tile's player member variable must then be set to *NULL*.
- EndTile Function 4
  - This is a constant function.
  - This function should always return the tile's symbol.

3.8 BoobyTrappedTile

This class inherits publicly from both the Tile and the Threat classes.

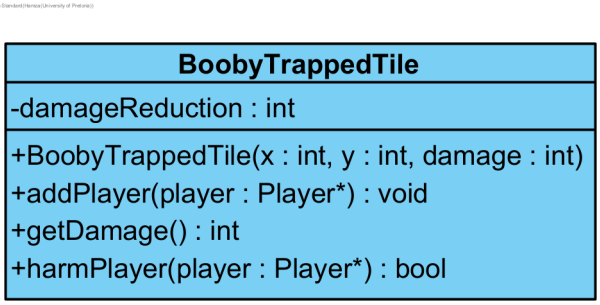

Figure 9: UML Diagram of BoobyTrappedTile

3.8.1 Members

- BoobyTrappedTile Member 1
  - This member variable indicates how much the tile's damage is reduced.

3.8.2 Functions

- BoobyTrappedTile Function 1
  - This function should call one parent constructor with the corresponding x and y values.
  - *isStart* and *isEnd* should be false
  - These tiles are represented with the 'x' symbol.
  - It must also call its other parent constructor to initialise the damage.
  - The tile's damage reduction must initially be 0.
- BoobyTrappedTile Function 2
  - This function should call the parent class's version of the `addPlayer` function.
  - It must then call the `harmPlayer` function with the player.
  - If the tile managed to kill the player, this function should call `removePlayer`
  - If a TileException is thrown at any point in this function, the text "Failed to harm player" must be output (ending with a newline)
- BoobyTrappedTile Function 3
  - This function should return the effective damage of the tile.
  - Effective damage = $damage - damageReduction$.
  - After performing the calculation, the *damageReduction* must always increase by 10.
- BoobyTrappedTile Function 4
  - This function call player's `takeDamage` function, passing *this* as an argument.

3.9 LethalTile

This class inherits publicly from both the Tile class and the Threat class.

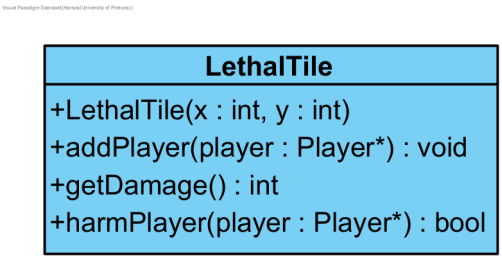

Figure 10: UML Diagram of LethalTile

3.9.1 Functions

- LethalTile Function 1
  - This function should call one parent constructor with the corresponding x and y values.
  - *isStart* and *isEnd* should be false.
  - These tiles are represented with the '#' symbol.
  - It must also call its other parent constructor to initialise the damage with a value of 200.
- LethalTile Function 2
  - This function should call the parent class's version of the `addPlayer` function.
  - It must then call the `harmPlayer` function with the player.
  - If the tile managed to kill the player, this function should call `removePlayer`
- LethalTile Function 3
  - This function should return the damage of the tile.
- LethalTile Function 4
  - This function call player's `takeDamage` function, passing *this* as an argument.

3.10 Arena

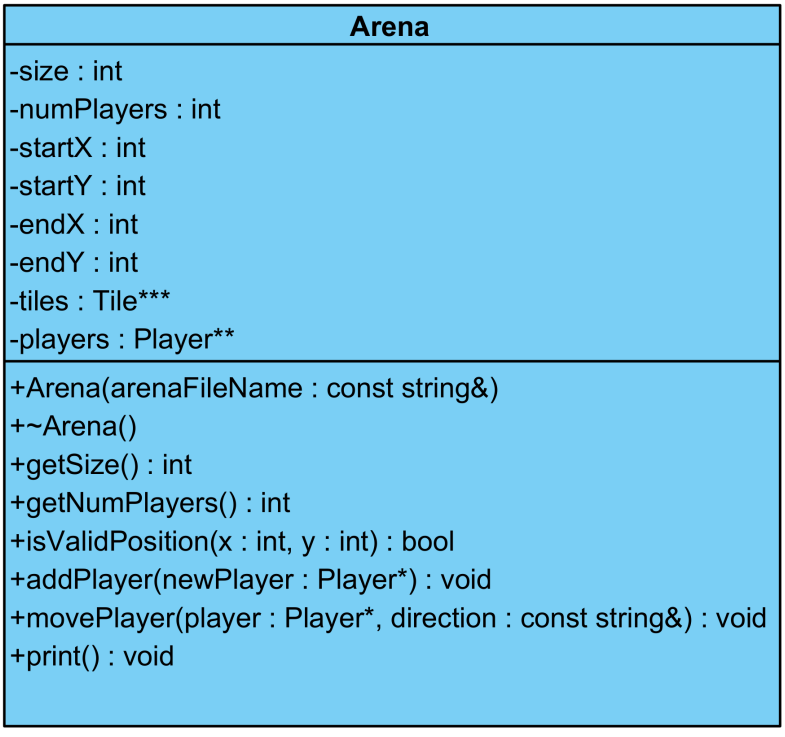

Figure 11: UML Diagram of Arena

3.10.1 Members

- Arena Member 1
  - This member variable stores the number of rows and columns in the Arena.
- Arena Member 2
  - This member variable stores the size of the *players* array.
- Arena Member 3
  - This member variable stores the x position for the entrance/starting tile.
- Arena Member 4
  - This member variable stores the y position for the entrance/starting tile.
- Arena Member 5
  - This member variable stores the x position for the exit/ending tile.
- Arena Member 6
  - This member variable stores the y position for the exit/ending tile.
- Arena Member 7
  - This member variable stores a 2D array of pointers to Tiles for the arena.
- Arena Member 8
  - This member variable stores an array of Player pointers.

3.10.2 Functions

- Arena Function 1
  - This function should handle creation of the arena using a textfile.
  - You may assume that that the textfile will always be present and that it already include the file extension.
    - The first line in the textfile will contain a number. This number represents the number of rows and columns in arena.
    - The remainder of the textfile contains the symbols for the tiles in the arena.

    - Regular tiles are represented with a '+'.
    - The start tile is represented with an 's'.
    - The end tile is represented with an 'e'.
    - BoobyTrapped tiles are represented with a 'x'.

    - Lethal tiles are represented with a '#'.
  - This function should create the correct types of tiles base on the read characters in the textfile.
  - *numPlayers* should be set to 0.
  - The *players* array should be initialised an n array of size 0.
- Arena Function 2
  - This function should perform memory management for the entire arena.
  - It should delete all tiles and the pointer array member variables within the class.
- Arena Function 3
  - This is a constant function.
  - This function should return the size of the arena.
- Arena Function 4
  - This is a constant function.
  - This function should return the number of players in the arena.
  - Note: this is different from the total number of players that have been created.
- Arena Function 5
  - This is a constant function.
  - This function should ensure that the x and y positions are valid for the size of the arena.
- Arena Function 6
  - The function should do nothing if the passed-in parameter is **NULL** or the *newPlayer* is already in the arena.
  - Since only one player can occupy a tile and there is only one start tile, the *newPlayer* will not be able to be added into the game if the starting tile has a player on it. This function should have a try-catch block, with a catch block for the TileException class.
    - If the exception is thrown, you should output the message from the thrown exception. (Ending with a newline)
  - Once the player has been successfully added, remember to update the *players* array and *numPlayer* to reflect this.
- Arena Function 7
  - This function will handle player movement on the arena.
  - This function should do nothing if the passed in *player* parameter is NULL or the player is not in the arena.

- The *direction* parameter will be one of 4 options: "up", "down", "left", "right"
  - If the direction is anything other than the above-mentioned options, the function should do nothing.
- If the position where the player is trying to move is invalid, output the following:
  "Invalid Position"
  This string must end with a newline.
- This function should call *removePlayer* and *addPlayer* appropriately to move the player.
  - If a TileException is thrown while trying to move a player, this function should output the message (ending with a newline).

- Arena Function 8
  - This is a constant function.
  - This function should output information about the arena and the players.
  - Firstly, the function should output the number of players currently in the arena. It should adhere to the following format:
    "<x> players entered the Arena". (ending with a newline)
  - Next, the number of living players should be displayed. It should adhere to the following format:
    "<y> alive players" (ending with a newline and a blank line)
  - After this, the players' information should be displayed. It should adhere to the following format:
    "-Player info-"
  - Each player in the arena should then be output as well, with a blank line after each player.
  - Finally, a string representation of the Tiles in the arena should be output.
  - Example of complete output:

```
2 players entered the Arena
6 alive players

-Player info-
Username: Aaron
Health: 60
Position: (4,3)

Username: Brett
Health: 90
Position: (4,1)
Rocks: 5
Damage: 10

```

```
++e++
+xxx+
+#+x+
#++x+
+BsA+
```

4 Memory Management

As memory management is a core part of COS110 and C++, each task on FitchFork will allocate approximately 10% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocates *all* the memory that was allocated.

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- You may only use the following libraries:
  - iostream
  - sstream
  - fstream
  - string
- You are supplied with a **trivial** main demonstrating the basic functionality of this assessment.

7 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Arena.cpp
- Tile.cpp
- TileException.cpp
- Threat.cpp

- BoobyTrappedTile.cpp
- LethalTile.cpp
- EndTile.cpp
- Player.cpp
- Pacifist.cpp
- Villain.cpp
- `main.cpp`
- Any textfiles used by your `main.cpp`
- `testingFramework.h` and `testingFramework.cpp` if you used these files.

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ -Werror -Wall *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your best mark will be your final mark. Upload your archive to the Practical 6 slot on the FitchFork website. If you submit after the deadline but before the late deadline, a 20% mark deduction will be applied. **No submissions after the late deadline will be accepted!**