Object Oriented Programming Assignment

Matthias Bartolo* (0436103L)

*B.Sc. It (Hons) Artificial Intelligence (Second Year)

Study-unit: **Object Oriented Programming**

Code: **CPS2004**

Lecturer: **Dr Clyde Vassallo**

Link to GitHub: https://github.com/mbar0075/CPS2004-Assignment

Last Commit Hash: `37ab8126de71dfc32cf5059539f43c2e750d55bd`

Task 1

UML Diagram

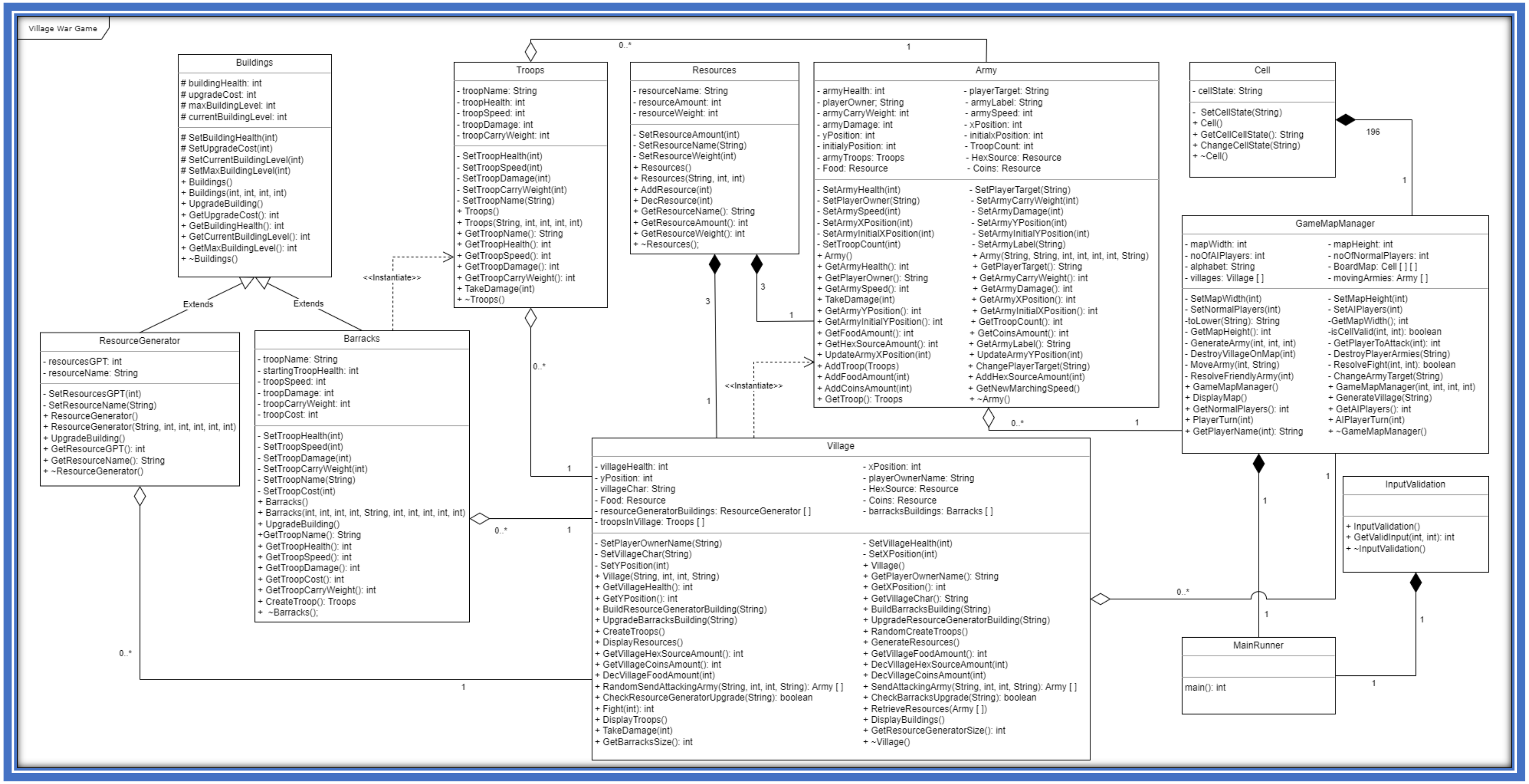


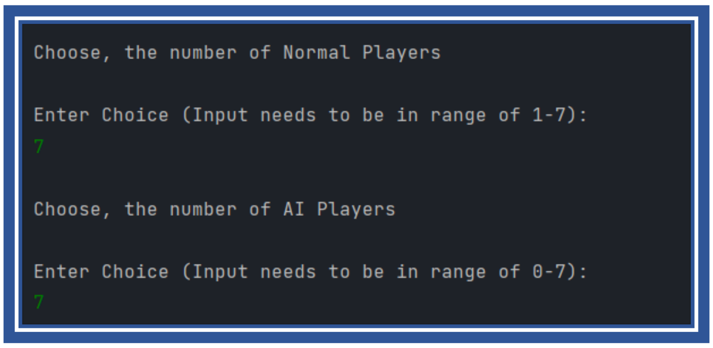
Figure 1: Village War Game UML

Problem solution and User guide

Task 1 entailed the creation and implementation of a Village War game; such problem was resolved through partitioning said problem in different classes as can be seen in Figure 1. Furthermore, the mentioned classes include the following:

1. **Resources** – is the class which specifies, the Resource entity in the Village War Game, as well as contain all operations performed on such Resource. Instantiation, of such class will be used, to create the three types of resources utilised by said game: Coins, Food and HexSource resources.
2. **Troops** – is the class which specifies the Troop entity in the Village War Game, as well as contain all operations performed on such Troop.
3. **Buildings** – is the class which specifies the building entity in the Village War Game, as well as contain all operations performed on such Building.
4. **ResourceGenerator** – is the class which specifies the Resource Generator Building entity in the Village War Game, as well as contain all operations performed on such entity. Furthermore, such Building, is also responsible for generating resources, and inherits properties from the building class.
5. **Barracks** – is the class which specifies the Barracks Building entity in the Village War Game, as well as contain all operations performed on such entity. Furthermore, such Building, is also responsible for creating/training Troops, and inherits properties from the building class.
6. **Army** – is the class which specifies the Army entity in the Village War Game, as well as contain all operations performed on such entity. An Army is composed of several Troops and contains Resources.
7. **Village** – is the class which specifies the Village entity in the Village War Game, as well as contain all operations performed on such entity.
8. **Cell** – is the class which acts as a data type for each of the individual cells in the Village War Game Map.
9. **GameMapManager** – is the class which contains the Village War Game Map, and all the operations performed on such map, as well as connecting the logical aspect of the game, to the visual aspect of the game.
10. **InputValidation** – is the class which is responsible for processing and validating the user input.
11. **MainRunner** – is the class which runs the Village War Game.

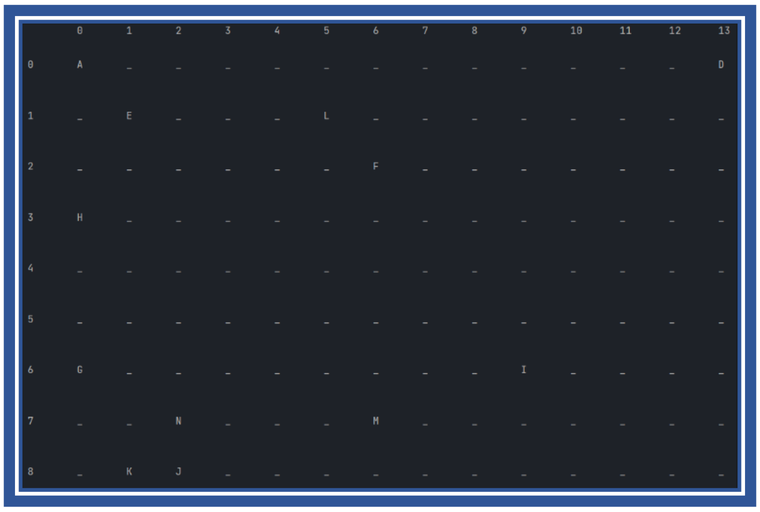
The solution presented prompts the user at the start of the game to input, the number of Normal Players and AI Players, as can be seen in Figure 2.



```
Choose, the number of Normal Players

Enter Choice (Input needs to be in range of 1-7):
7

Choose, the number of AI Players

Enter Choice (Input needs to be in range of 0-7):
7
```

Figure 2: User Input, for number of players

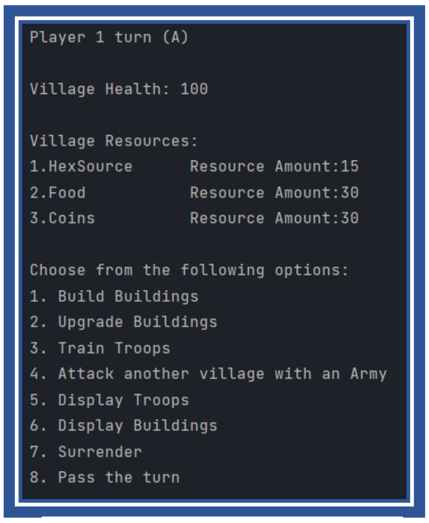
Depending on the user input, the program proceeds to create a GameMapManager, which will in turn generate Villages for each of the respective Players. Subsequently, the program will display a Map, containing the respective Villages. Each Village on the Map is represented, by a distinct Capital letter, as can be seen in Figure 3.



```
     0   1   2   3   4   5   6   7   8   9   10  11  12  13

0    A   _   _   _   _   _   _   _   _   _   _   _   _   D

1    _   E   _   _   _   L   _   _   _   _   _   _   _   _

2    _   _   _   _   _   _   F   _   _   _   _   _   _   _

3    H   _   _   _   _   _   _   _   _   _   _   _   _   _

4    _   _   _   _   _   _   _   _   _   _   _   _   _   _

5    _   _   _   _   _   _   _   _   _   _   _   _   _   _

6    G   _   _   _   _   _   _   _   _   I   _   _   _   _

7    _   _   N   _   _   _   M   _   _   _   _   _   _   _

8    _   K   J   _   _   _   _   _   _   _   _   _   _   _
```

Figure 3: Showing Villages, on the Map

The program, then proceeds to process the first Game Round where, every Normal Player is given the following options menu: Visualisation of such Menu can be seen in Figure 4.



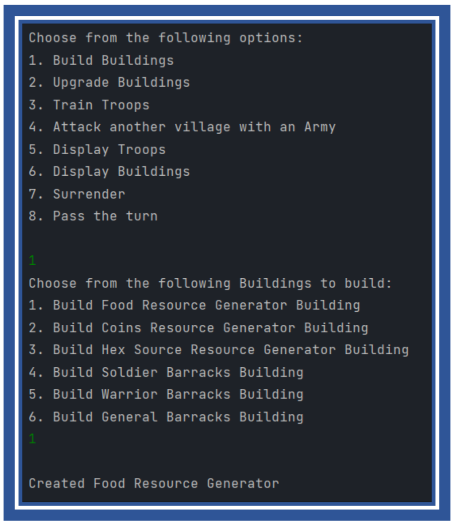
```
Player 1 turn (A)

Village Health: 100

Village Resources:
1.HexSource       Resource Amount:15
2.Food            Resource Amount:30
3.Coins           Resource Amount:30

Choose from the following options:
1. Build Buildings
2. Upgrade Buildings
3. Train Troops
4. Attack another village with an Army
5. Display Troops
6. Display Buildings
7. Surrender
8. Pass the turn
```

Figure 4: Player Options Menu

1. **Build Buildings** – Once a player chooses this option, a new menu is displayed enabling the player to choose from six different buildings to erect. If the player does not have sufficient Coins resource, the player will be unable to construct the chosen building. In such case, a relevant message is shown. On the other hand, if the player has enough Coins, then construction will be finalised and the Player will also have the facility to choose other options from the main menu, such as train troops or upgrade existing buildings. The construction of new buildings will increase the village's health, and chance of survival in case of attacks. Choosing such option can be seen in Figure 5.
2. **Upgrade Buildings** – Once a player chooses this option, a new menu is displayed enabling the player to choose from six different structures to upgrade. Moreover, the player is only able to upgrade the building if the existing structure had been already built, and the player has enough HexSource resource to continue with such improvement. A relevant message will be displayed if the player has insufficient resources and/or building is not yet erected. The player needs to be wise in choosing the building to upgrade, since upgrading a resource generator building will directly increase the number of resources generated per turn, whilst if a barrack building is upgraded this will directly imply stronger troops. If there are more buildings of the same type, the program will automatically upgrade the building with the lowest building level. Consequently, once a building level reaches the maximum level, the player will be unable to upgrade said building, and a respective message is shown. Furthermore, the upgrading of new buildings will add to the village's health and strengthen the survival rate in case of an attack. Choosing such option can be seen in Figure 6.
3. **Train Troops** – When the player chooses this option, the program will go through the troop barracks in the village and will seek the player's input whether he would want to create a troop from that barrack. In order that the player may train troops, there must be sufficient Food resources. If this is not the case, a relevant message is displayed. Training of troops is critical for the village defence from potential attacks from other Armies. Choosing such option can be seen in Figure 7.

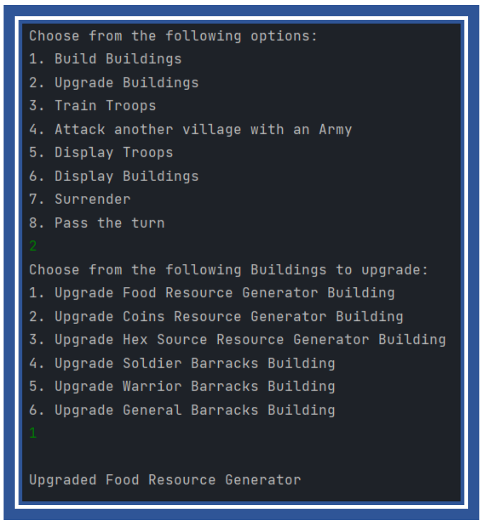
4. **Attack another village with an Army** – If a player chooses this option, the program will first ask the player to identify a village to attack. Consequently, the player is unable to choose to attack his own village. Once, a choice is made by the player, the program will iterate through all the village troops, and will ask the player whether to add that troop to the army, which will be attacking the other village. The creation and deployment of such attacking army will update the Map with the village's label in lower case, in one of the village's adjacent cells. Consequently, if all the adjacent cells are occupied, the player will not have the facility to create an army and the relevant message is shown. Choosing such option can be seen in Figures 8 and 9.
5. **Display Troops** – The displaying of the troops will give a full view of the strength of the troops stationed within the village, who will defend the village upon any attack from outsider armies. This option is given to the player to strategically plan his moves.
6. **Display Buildings** – The displaying of buildings within the village will show the internal structures within the village. By choosing this option, the player could strategically plan his next moves.
7. **Surrender** – The player has the option to give up the fight or the protection of the village by surrender. This will imply that the player will be exiting the game, whereby all the village's structures, and army will be destroyed and consequently removed.
8. **Pass the turn** –This signifies the end of the player's turn, and the program will execute the next player turn.

The game loops through several game rounds, until the winning condition is satisfied, whereby only one player's village remains. Furthermore, each game round is composed of player's turn, and each turn is comprised of different turn phases. For every player turn the program, first resolves friendly troop arrival in the player's village, whereby armies which were sent to attack other villages are now considered stationed in the village, and the resources they have acquired will be added to the village's resources. In addition, the army's troops would also be added to the village's troops. Secondly, the program, then proceeds to resolve fights between the player's village, and enemy troops that have arrived at the village's location. Combat between the two entities is resolved by invoking the total damage of the troops in the village on the army health, and the total damage of the troops in the army on the village health. If the village health is smaller than or equal to 0, the village, and all its armies, on the board, will be destroyed. Consequently, if the attacking army's health is smaller than or equal to 0, the attacking army is demolished. Notwithstanding, if the attacking army is not destroyed, it will take some of the village's resources, and then return to its home village. The program then proceeds to generate the resources from the resource generator buildings, for the player's turn. Subsequently, the generated resource amount will be added to the village's resources. As part of the player's turn, the program will then proceed to update the player's village, attacking armies' positions on the Map. To conclude the player's turn phase, the program displays the player's options menu to the player. Note that, if either a player surrenders, or his village is destroyed by an attacking army, such player's turn is skipped indefinitely, and all game objects owned by the player will be removed.



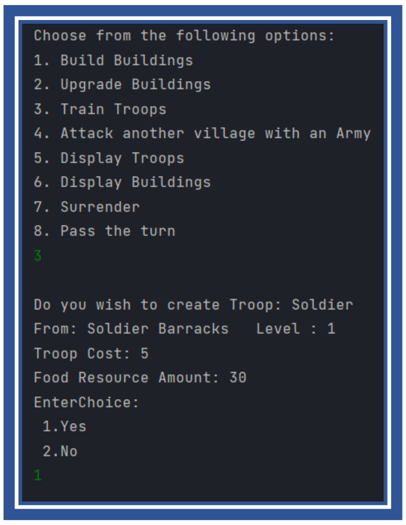
```
Choose from the following options:
1. Build Buildings
2. Upgrade Buildings
3. Train Troops
4. Attack another village with an Army
5. Display Troops
6. Display Buildings
7. Surrender
8. Pass the turn

1
Choose from the following Buildings to build:
1. Build Food Resource Generator Building
2. Build Coins Resource Generator Building
3. Build Hex Source Resource Generator Building
4. Build Soldier Barracks Building
5. Build Warrior Barracks Building
6. Build General Barracks Building
1

Created Food Resource Generator
```

Figure 5: Choosing Build Building option.



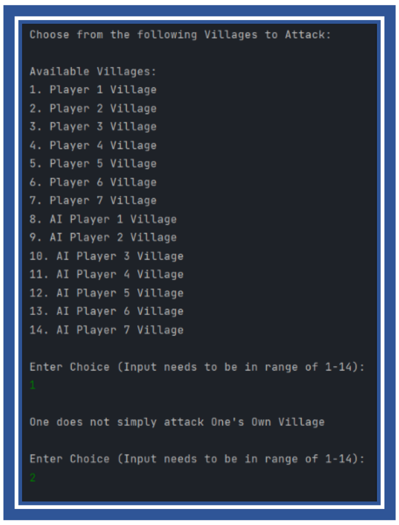
```
Choose from the following options:
1. Build Buildings
2. Upgrade Buildings
3. Train Troops
4. Attack another village with an Army
5. Display Troops
6. Display Buildings
7. Surrender
8. Pass the turn
2
Choose from the following Buildings to upgrade:
1. Upgrade Food Resource Generator Building
2. Upgrade Coins Resource Generator Building
3. Upgrade Hex Source Resource Generator Building
4. Upgrade Soldier Barracks Building
5. Upgrade Warrior Barracks Building
6. Upgrade General Barracks Building
1

Upgraded Food Resource Generator
```

Figure 6: Choosing Upgrade Building option.



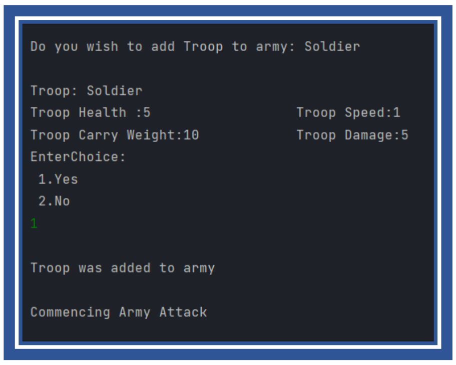
```
Choose from the following options:
1. Build Buildings
2. Upgrade Buildings
3. Train Troops
4. Attack another village with an Army
5. Display Troops
6. Display Buildings
7. Surrender
8. Pass the turn
3

Do you wish to create Troop: Soldier
From: Soldier Barracks   Level : 1
Troop Cost: 5
Food Resource Amount: 30
EnterChoice:
 1.Yes
 2.No
1
```

Figure 7: Choosing Train Troops option.



```
Choose from the following Villages to Attack:

Available Villages:
1. Player 1 Village
2. Player 2 Village
3. Player 3 Village
4. Player 4 Village
5. Player 5 Village
6. Player 6 Village
7. Player 7 Village
8. AI Player 1 Village
9. AI Player 2 Village
10. AI Player 3 Village
11. AI Player 4 Village
12. AI Player 5 Village
13. AI Player 6 Village
14. AI Player 7 Village

Enter Choice (Input needs to be in range of 1-14):
1

One does not simply attack One's Own Village

Enter Choice (Input needs to be in range of 1-14):
2
```

Figure 8: Choosing Attack another Village option.



```
Do you wish to add Troop to army: Soldier

Troop: Soldier
Troop Health :5                    Troop Speed:1
Troop Carry Weight:10              Troop Damage:5
EnterChoice:
 1.Yes
 2.No
1

Troop was added to army

Commencing Army Attack
```

Figure 9: Appending Village Troop, to Army

Design and Implementation

Special care was taken, to design such large program, to adhere and implement a multitude of OOP principles. The principles which were implemented include:

1. **Modularity** – The program was split, between eleven classes, whereby every class represents a different game entity in the Village War Game. In doing so, the code utilised to tackle such problem was made more readable and robust.
2. **Inheritance** – The program, made use of Inheritance with respect to the Resource Generator, and Barracks Building. This was done, as to reuse code, and make proper use of the OOP principle.
3. **Instantiation** – Instead of creating multiple classes, to represent the different Resources and Troop types, Instantiation was utilised to avoid repetition of code. To illustrate, instead of creating two different classes, which will have the same methods, whereby one class would represent the HexSource resource, and the other class would represent the Food resource, a single Resources class was created. This facilitated the distinction between resources through the resource name property of the Resources class. Similarly, such concept was also adopted when creating different Troops, ResourceGenerator buildings, and Barracks buildings.
4. **Encapsulation** – It was made sure that the program utilised proper Encapsulation principles, whereby all class variables were set to private, and the only access to such variables would occur only through setters, and getters.
5. **Code Scalability** – In continuation with the previous point, setters, and getters were created for all class variables. Although, a few of these methods were not utilised, they were included with the aim that the program could be extended upon in the future, and thus, access to such variables through setters and getters would be available.
6. **Proper Input Validation** – The program included proper input validation for user input, whereby every time a user would enter an incorrect input, a relevant message would be shown, and the program, would not terminate abruptly. To facilitate this, an InputValidation class was constructed, as well as utilising various String input, to avoid the program crashing.
7. **Proper Memory Management** – As highlighted during, the OOP lectures, Proper Memory Management, was utilised through unique pointers, whilst avoiding raw pointers. Nevertheless, it was made sure that every time a unique pointer was created, a reset pointer was also implemented to avoid any memory leaks. Furthermore, the program made extensive use of C++'s dynamic lists structure. In doing so, only the portion of required memory on the heap, was utilised.
8. **Game Customizability** – An extensive list of constants was utilised in the header file, to hold the default game entities' values, as well as the map specifications. Furthermore, this provides a vast array of customizability, whereby one can easily change a specific Troops' health constant, or change the map width constant, and the change would be reflected once a rerun of the game occurs. The game was specifically built to be robust in handling this customization feature.
9. **Implementation of Normal Player and AI Players** – Implementation of the player entities was facilitated through the creation of a Normal Player Turn method, and an AI Player Turn method. Both methods were included in the GameMapManager class. This was done with the aim of avoiding the duplication of code, whereby instead of creating two new player classes, which would ultimately utilise the same code, a single class could be utilised to represent both players' functionality. Unfortunately, a backlash of such method was that the GameMapManager class would be of a large size. However, the benefit of this design pertained to code reusability.
10. **Comments** – The program was properly commented. Attention was given to ensure that the code would be readable.

Moreover, the following **Assumptions** were made:

1. Every time a Building was created or upgraded; the Village's health was also increased. Nonetheless, every time the Village takes damage, buildings were not destroyed, however, all buildings would be destroyed, when the Village is destroyed i.e., the village's health is smaller or equal to 0. In addition, the player could also erect multiple buildings of the same type; such feature was implemented to facilitate prolonged strategic matches, between villages of large health.
2. The armies on the Map were moved in accordance with each Player's Turn, rather than moving all the armies for every Game Round. For example, when the player turn of Player 1 occurs, the armies of Player 1 would move on the Map, rather than moving all the players' armies, after the last player's turn. Although this assumption deviates from the project specifications, a lot of critical thinking was done, in factor of taking such assumption. The main reasons for choosing such assumption included that, new players could be overwhelmed, if multiple armies would move on the board at the same time, and thus the player would have a hard time to keep track of what is happening. Furthermore, this feature would also increase player engagement whilst, motivating a higher strategic thinking, with respect to the game flow, as the player would have another chance to defend his village prior to an army attack.
3. In case that the number of normal players is set to 1, and the number of AI players is set to 0, Player 1 will automatically win the game, as the game's winning condition whereby a single Player is still left standing, would be satisfied. Moreover, the game was designed with the scope of deploying armies to destroy other villages, rather than letting Player 1 continuously, build buildings, and train troops, which can only be stationed in the village.

As can be seen in Figure 1, and previously mentioned, the enormous size of the program created, required the development of numerous methods. However, the most prominent methods, which enable the main Village War Game logic, are shown hereunder:

The following methods form part of the GameMapManager class:

1. DisplayMap() – This method is used to display the Map, along with the row and column numbers.
2. GenerateVillage() – This method is used to generate a Village, on the Map, as well as create a Village instance, and add said instance to the villages dynamic list in the aforementioned class. Moreover, spawning of such Villages, on Map was done such that the first four villages would be spawned at the Map's corners, and the remaining are spawned at a random position.
3. Generate Army() – This method is used to generate an Army on the Map, as well as creating an Army instance given a player index, and an opponent index. Created army instance will be added to the movingArmies list.
4. DestroyVillageOnMap() – This method is used to loop, through all the Cells in the Map, removing all the Village and Armies on the Map, for a given player index. This process was achieved by parsing through all the characters, in every Cell, and removing the specified characters in turn.
5. MoveArmy() – This method is used to update the positions of all the armies of a given player index. This was achieved by iterating through all the armies in the movingArmies list, and if that army belonged to the requested player, then the army would move by its army speed closer to the opponent village. The army would first try to move in the horizontal direction, and then in the vertical direction.
6. ResolveFight() – This method is used to resolve a fight between an attacking army, and a village, which share the same position. In this method, both attacking army, and defending village show the damage imposed by the opposing side. Namely, if the village is destroyed,

then the player is removed from the game, whereas, if the army survives the attack, then the army steals a portion of the village's resources, proportional to the army's total carry weight before heading back to its home village. The approach taken for the surviving army to head back to its home village, was that of changing the army's target to its home village, and in doing so facilitating multiple instances of code reutilisation.

7. ResolveFriendlyArmy() – This method is used to resolve friendly army arrival to its home village. If the friendly army shares the same position as its home village, then the army's resources will be added to the village's resources, and the army's troops will be added to the village's troops. Consequently, after the army has transferred all its troops, and resources to the village, the empty army is removed.
8. PlayerTurn() – This method is used to replicates a normal player's turn phases, given a particular player index.
9. AIPlayerTurn() – This method is used to replicates an AI player's turn phases, given a player index. This method primarily works based on random indexes which are used to replace a normal player's choices. Nevertheless, such AI player, should not be taken lightly as one could easily lose, to it.

The following methods form part of the Village class:

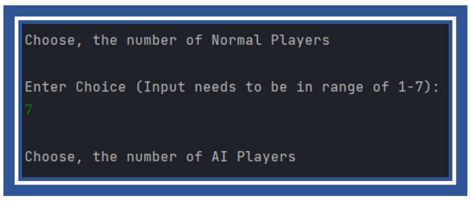
1. BuildResourceGeneratorBuilding() & BuildBarracksBuilding() – These methods ensure that first checks are undertaken to determine whether the Village has enough Coins Resource to build. Insufficient amount of Coins Resources will not allow the creation of such building, and a message is displayed. On the other hand, if Village has Sufficient Coins, then cost to build the respective building, will be taken from Village Coins Resource, and structure would be erected.
2. UpgradeResourceGeneratorBuilding() & UpgradeBarracksBuilding() - These methods ensure that first checks are undertaken to determine whether the Village has enough HexSource Resource to upgrade. Insufficient amount of HexSource Resources will not allow the upgrading of such building, and a message is displayed. On the other hand, if Village has Sufficient HexSource, then cost to upgrade the respective building, will be deducted from Village HexSource Resource, and the building with the lowest level, and which type matches the structure would be upgraded.
3. CreateTroops() & RandomCreateTroops() – The first method is used to create and/or train troops from a particular barrack by iterating through all the barracks, and asking the player whether to create a troop from that barrack. Relevant message is shown if player does not have insufficient Food Resource to create troop. The second method works similarly to the first method, however instead of user input, it replicates the user's input by a random choice.
4. GenerateResources() – This method will iterate through all the village's ResourceGenerator buildings, and based on building type, the method will append the Generated Resource Amounts, to the village's resources.
5. SendAttackingArmy & RandomSendAttackingArmy() – The first method is used to append troops to an Army based on user input, and send an assembled army to attack another village. The second method works similarly to the first method, however instead of user input, it replicates the user's input by a random choice.
6. Fight() & RetrieveResources() – The first method simulates a fight between an attacking army, and a village. The second method allows the player to retrieve Resources and Troops from the friendly Army.

<u>Testing</u>

Testing Proper Input Validation for integer input:

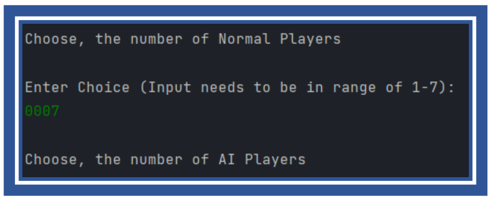
Please note that the same input and validation method, was used for both number of normal players input, and number of AI players' input. In the light of this, the testing was only performed on the number of normal players input, since the functionalities of the normal players input are replicated for the AI players input, thus any problems with the game would easily be identified when testing with normal players input. The program also utilised multiple instances of Sting input, as such input would never raise any exceptions, and avoid the program from terminating abruptly.

Input Type	**Input**	**Actual Output**	**Targeted Output**	**Figure**
Valid Input	Int x | 0<=x<=15	Accepted	Accepted	Figure 10
Valid Input	0007	Accepted	Accepted	Figure 11
Valid Input	Double d	Accepted	Accepted	Figure 12
Invalid Input	Int x | x<0 ^ x>15	Not Accepted	Not Accepted	Figure 12
Invalid Input	String s	Not Accepted	Not Accepted	Figure 13



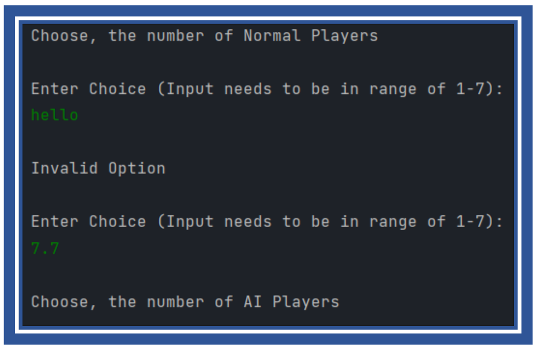
```
Choose, the number of Normal Players

Enter Choice (Input needs to be in range of 1-7):
7

Choose, the number of AI Players
```

Figure 10: Valid Input 1



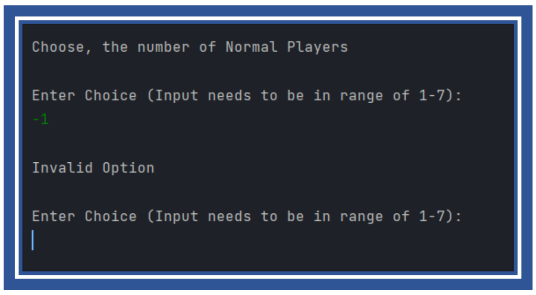
```
Choose, the number of Normal Players

Enter Choice (Input needs to be in range of 1-7):
0007

Choose, the number of AI Players
```

Figure 11: Valid Input 2



```
Choose, the number of Normal Players

Enter Choice (Input needs to be in range of 1-7):
hello

Invalid Option

Enter Choice (Input needs to be in range of 1-7):
7.7

Choose, the number of AI Players
```

Figure 12: Valid Input 3 and Invalid Input 1

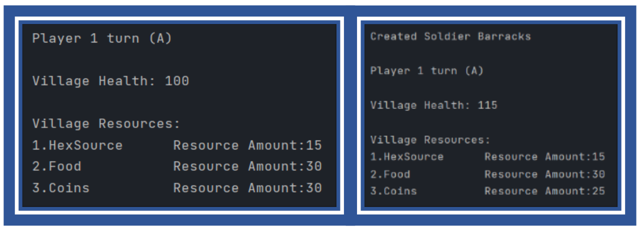


```
Choose, the number of Normal Players

Enter Choice (Input needs to be in range of 1-7):
-1

Invalid Option

Enter Choice (Input needs to be in range of 1-7):
|
```

Figure 13: Invalid Input 2

Testing Village functionality:

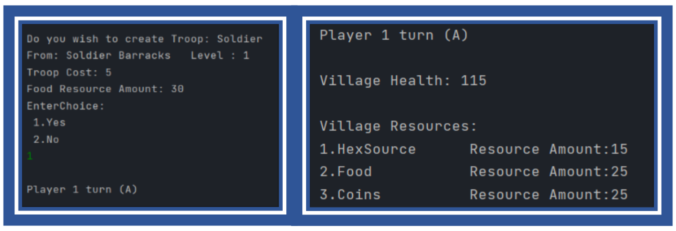
Test Case	**Figure**
Creating Building (Checking for correct reduction of Village resources)	Figure 14
Creating Troop (Checking for correct reduction of Village resources)	Figure 15
Creating Army (Checking that Player cannot choose his own Village, to attack)	Figure 16
Creating Army (Checking for correct reduction of Troops from Village Troops)	Figure 17
Checking for correct Army movement between Player Turns	Figure 18
Checking for reduction of Village resources, after an attack, from an attacking Army	Figure 19
Checking for Army returning to its home Village (correct transferring of resources, and Troops)	Figure 20
Checking for Army destruction if Village Troops destroyed Army Troops	Figure 21
Checking for Village destruction if Army Troops destroyed Village	Figure 22
Checking Correct Functionality between multiple AI Players. This was done by creating 1 normal player and 7 AI players, making the normal player surrender, and checking whether one of the AI players win the game	Figure 23



```
Player 1 turn (A)

Village Health: 100

Village Resources:
1.HexSource        Resource Amount:15
2.Food             Resource Amount:30
3.Coins            Resource Amount:30
```

```
Created Soldier Barracks

Player 1 turn (A)

Village Health: 115

Village Resources:
1.HexSource        Resource Amount:15
2.Food             Resource Amount:30
3.Coins            Resource Amount:25
```

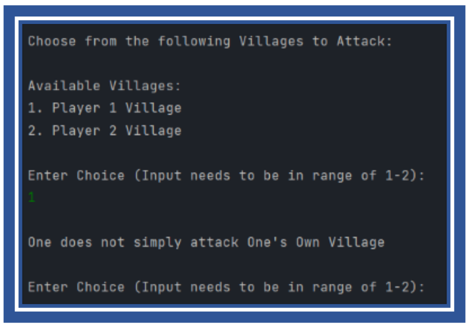
Figure 14: Resources reduction, upon creation of Building



```
Do you wish to create Troop: Soldier
From: Soldier Barracks   Level : 1
Troop Cost: 5
Food Resource Amount: 30
EnterChoice:
 1.Yes
 2.No
1

Player 1 turn (A)
```

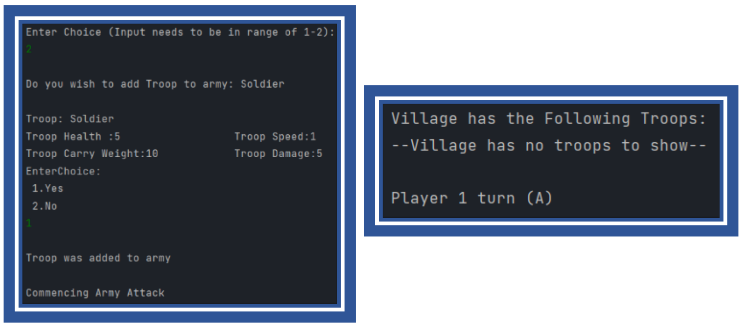
```
Player 1 turn (A)

Village Health: 115

Village Resources:
1.HexSource        Resource Amount:15
2.Food             Resource Amount:25
3.Coins            Resource Amount:25
```

Figure 15: Resources reduction, upon creation of Troop



```
Choose from the following Villages to Attack:

Available Villages:
1. Player 1 Village
2. Player 2 Village

Enter Choice (Input needs to be in range of 1-2):
1

One does not simply attack One's Own Village

Enter Choice (Input needs to be in range of 1-2):
```

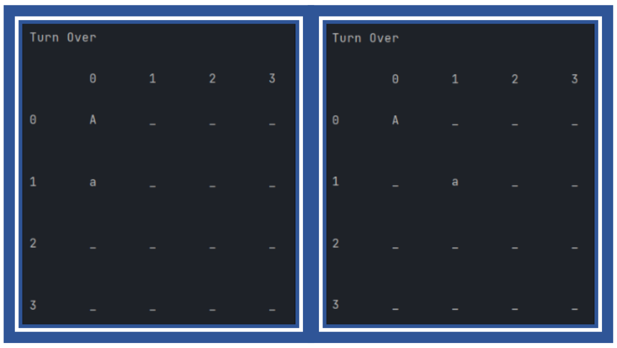
Figure 16: Player cannot choose to attack his own Village



```
Enter Choice (Input needs to be in range of 1-2):
2

Do you wish to add Troop to army: Soldier

Troop: Soldier
Troop Health :5                   Troop Speed:1
Troop Carry Weight:10             Troop Damage:5
EnterChoice:
 1.Yes
 2.No
1

Troop was added to army

Commencing Army Attack
```

```
Village has the Following Troops:
--Village has no troops to show--

Player 1 turn (A)
```

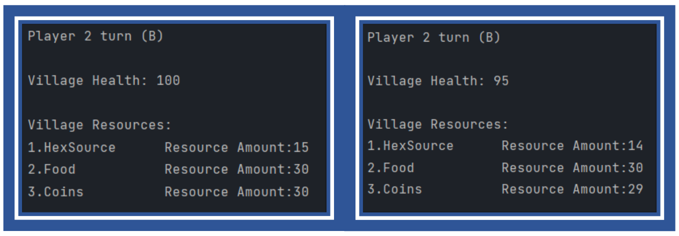
Figure 17: Village Troop reduction, upon initiating Army attack



```
Turn Over

        0    1    2    3

0       A    _    _    _

1       a    _    _    _

2       _    _    _    _

3       _    _    _    _
```

```
Turn Over

        0    1    2    3

0       A    _    _    _

1       _    a    _    _

2       _    _    _    _

3       _    _    _    _
```

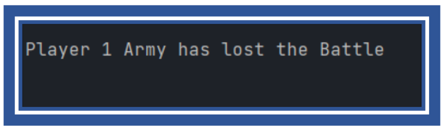
Figure 18: Army movements between Player Turns



```
Player 2 turn (B)

Village Health: 100

Village Resources:
1.HexSource        Resource Amount:15
2.Food             Resource Amount:30
3.Coins            Resource Amount:30
```

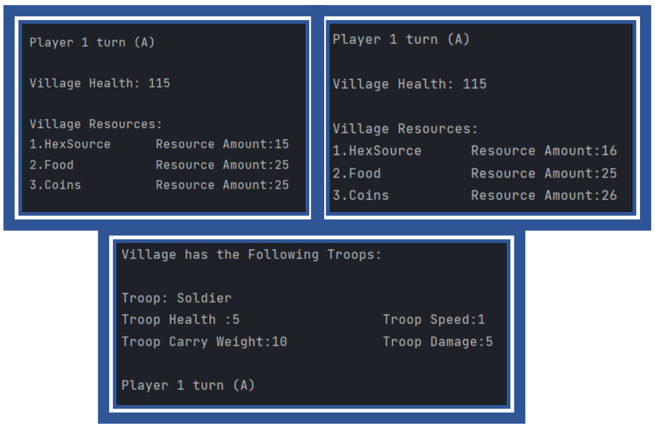
```
Player 2 turn (B)

Village Health: 95

Village Resources:
1.HexSource        Resource Amount:14
2.Food             Resource Amount:30
3.Coins            Resource Amount:29
```

Figure 19: Village Resource reduction, upon Army attack



```
Player 1 Army has lost the Battle
```

Figure 21: Attacking Army destruction, after attack

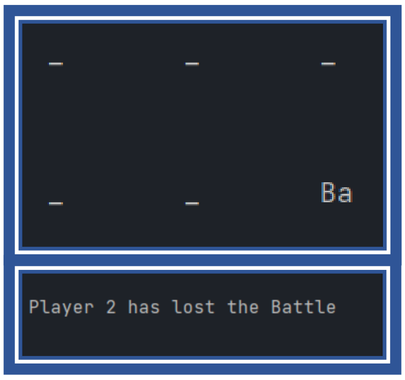


```
Player 1 turn (A)

Village Health: 115

Village Resources:
1.HexSource        Resource Amount:15
2.Food             Resource Amount:25
3.Coins            Resource Amount:25
```

```
Player 1 turn (A)

Village Health: 115

Village Resources:
1.HexSource        Resource Amount:16
2.Food             Resource Amount:25
3.Coins            Resource Amount:26
```

```
Village has the Following Troops:

Troop: Soldier
Troop Health :5                   Troop Speed:1
Troop Carry Weight:10             Troop Damage:5

Player 1 turn (A)
```

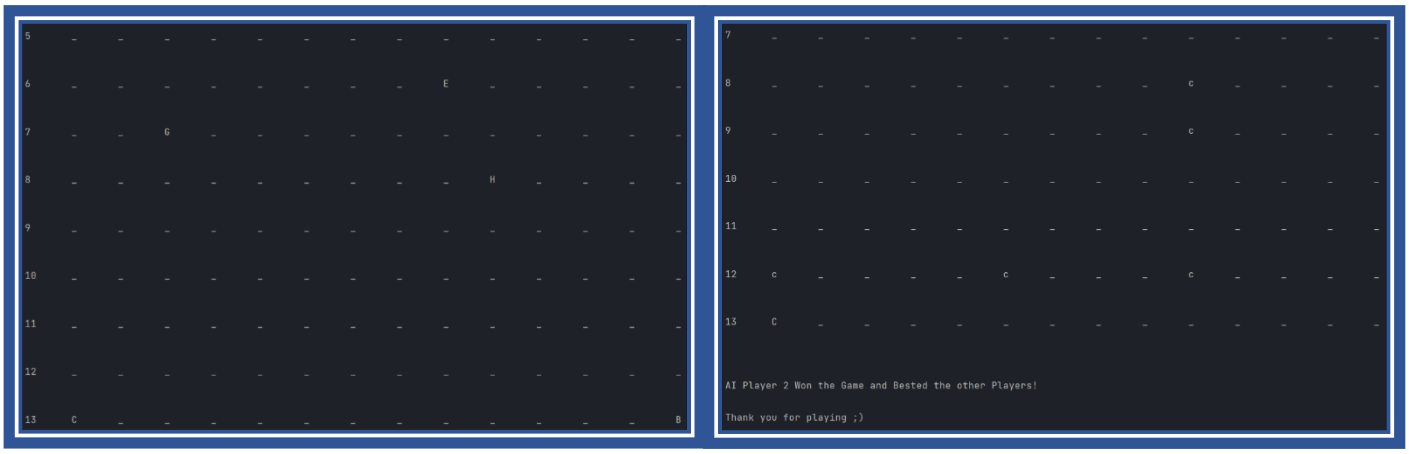
Figure 20: Transferring of friendly Army Troops and Resources, to its Home Village



```
_        _        _


_        _        Ba
```

```
Player 2 has lost the Battle
```

Figure 22: Village destruction, after attack



```
5     _   _   _   _   _   _   _   _   _   _   _   _   _   _

6     _   _   _   _   _   _   _   _   E   _   _   _   _   _

7     _   _   G   _   _   _   _   _   _   _   _   _   _   _

8     _   _   _   _   _   _   _   _   _   H   _   _   _   _

9     _   _   _   _   _   _   _   _   _   _   _   _   _   _

10    _   _   _   _   _   _   _   _   _   _   _   _   _   _

11    _   _   _   _   _   _   _   _   _   _   _   _   _   _

12    _   _   _   _   _   _   _   _   _   _   _   _   _   _

13    C   _   _   _   _   _   _   _   _   _   _   _   _   B
```

```
7     _   _   _   _   _   _   _   _   _   _   _   _   _   _

8     _   _   _   _   _   _   _   _   _   c   _   _   _   _

9     _   _   _   _   _   _   _   _   _   c   _   _   _   _

10    _   _   _   _   _   _   _   _   _   _   _   _   _   _

11    _   _   _   _   _   _   _   _   _   _   _   _   _   _

12    c   _   _   _   _   c   _   _   _   c   _   _   _   _

13    C   _   _   _   _   _   _   _   _   _   _   _   _   _


AI Player 2 Won the Game and Bested the other Players!

Thank you for playing ;)
```

Figure 23: Multiple AI Players

Critical Evaluation

Although the program was implemented in a proper, and optimised way as specified in the assignment brief, the program could still be extended upon through the implementation of a proper GUI interface. Moreover, such interface will greatly aid the Players, participating the game, whilst increasing the engagement factor of the game. Furthermore, the game could also be extended to include the "Save Game" functionality, by saving, and loading player data to/from a file.

Task 2

UML Diagram

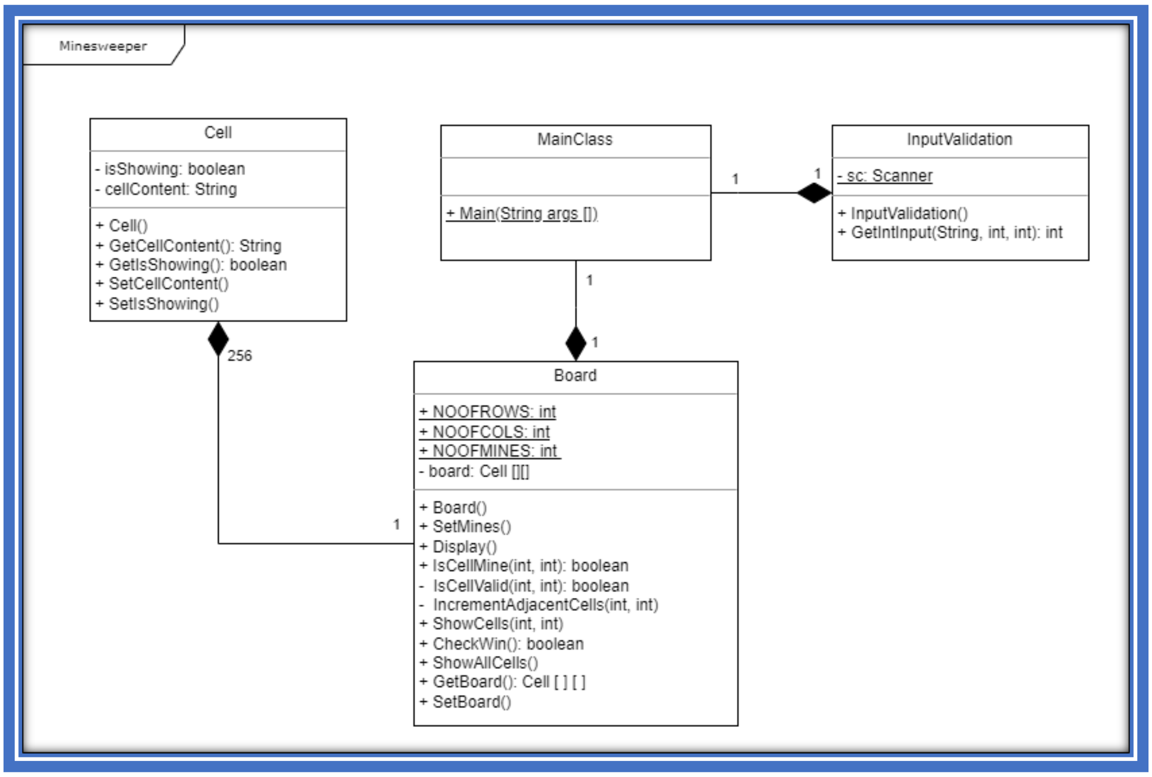


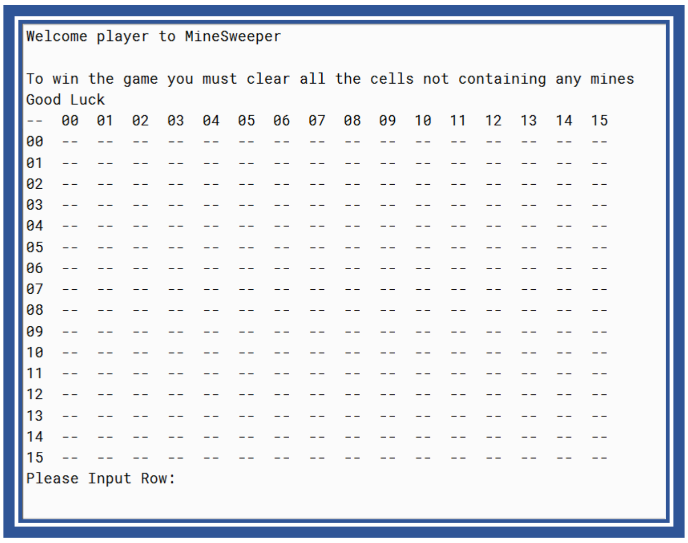
Figure 24: Minesweeper UML

Problem solution and User guide

Task 2 entailed the creation, and implementation of a Minesweeper game. This was addressed through the partitioning and the creation of different classes as can be seen in Figure 24. Furthermore, the mentioned classes include the following:

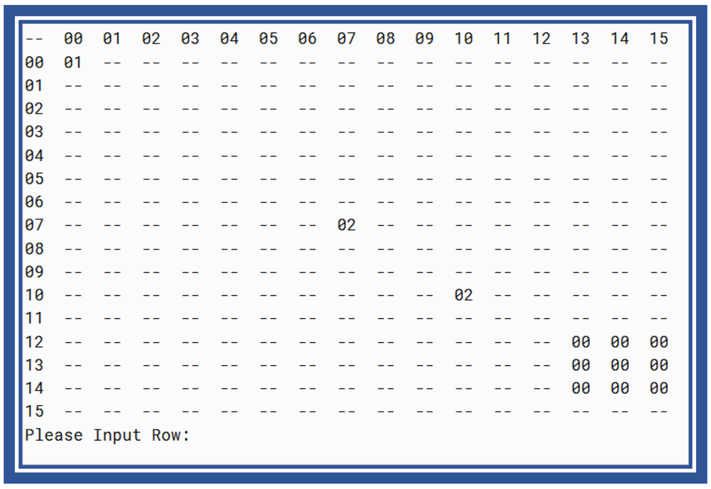
1. **MainClass** – is the class which runs the Minesweeper game.
2. **Board** – is the class which contains the Minesweeper board, and all the operations performed on such board.
3. **InputValidation** – is the class which is responsible for processing and validating the user input.
4. **Cell** – is the class which acts as a data type for each of the individual cells in the Minesweeper board.

Moreover, the solution presented prompts the user at the start of the game with a Menu as can be seen in Figure 25.



```
Welcome player to MineSweeper

To win the game you must clear all the cells not containing any mines
Good Luck
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
01  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
02  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
03  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
04  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
05  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
06  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
07  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
08  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
09  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
11  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
12  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
13  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
14  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
15  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
Please Input Row:
```

Figure 25: Initial Menu presented to the user



```
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  01  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
01  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
02  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
03  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
04  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
05  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
06  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
07  --  --  --  --  --  --  --  02  --  --  --  --  --  --  --  --
08  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
09  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10  --  --  --  --  --  --  --  --  --  --  02  --  --  --  --  --
11  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
12  --  --  --  --  --  --  --  --  --  --  --  --  --  00  00  00
13  --  --  --  --  --  --  --  --  --  --  --  --  --  00  00  00
14  --  --  --  --  --  --  --  --  --  --  --  --  --  00  00  00
15  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
Please Input Row:
```

The program, then waits for the user to input a valid row number, and a valid column number before proceeding to check whether at the positions of the cell entered, a mine was located. In case the cell was a mine, the program would display a termination message to the user and show all cells. If, however, the cell was mine free, but surrounded by adjacent mines, the current cell is shown. On the other hand, if the cell did not have adjacent mines, both the current cell, and all adjacent cells were shown. This can be seen in Figure 26.

Figure 26: Game Progression

Design and Implementation

As previously highlighted in the earlier sections, design of such program included the creation of four classes. This method was used to adhere to OOP principles, in splitting and tackling various small parts of the problem at a time, whilst making the code more readable. This also ensure proper encapsulation, through setting all variables private, and accessing them only through setters, and getters. The approach taken to solve such problem included the presentation of a fake board to the user, whilst hiding the contents of the real board. Furthermore, the cells of the real board, selected by the user would be shown at each turn.

This approach was utilised as it proved to be the most memory efficient approach on comparison with other approaches. Consequently, this led to the creation of a new data type called Cell, which was used by all the individual cells in the Minesweeper board. The class Cell included two class variables; a Boolean isShowing to denote whether the respective cell should be shown to the user, and a String cellContent, which denotes the content of the cell which will be displayed. The default Constructor for the class Cell is used to initialise the respective values to false and “00” respectively.

Moreover, the board class which describes the Minesweeper board and operations on such board, is composed of 256 of the previously mentioned Cells. The board class included three variables which hold the values for the number of rows, columns, and mines. These variables were set to constants, and under no circumstances they could be changed throughout execution. The board class also holds another variable consisting of 2d array of cells, which acts as the Minesweeper grid. Operations on such grid include:

1. Board() – This method acts as the default Constructor which enables the initialisation of the Cell objects in the 2D Cell array (board).
2. SetMines() – This method is used to randomly set the position of the mines on board. Furthermore, the cellContent of the randomly selected cell is changed to “XX”
3. Display() – This method is used to display the board, showing the row numbers and column numbers, as well as checking every cell whether it’s Boolean variable is set to true, if not “--” is shown instead of the cellContent.
4. IsCellMine(int row, int col) – This method is used to determine whether the current position of the cell is a mine or not, and respective return statement is invoked.

5. IsCellValid(int row, int col) – This method is used to determine, whether the current position of the cell falls within the valid board range of rows and columns. Respective return statement is invoked.
6. IncrementAdjacentCells(int row, int col) – This method is used to increment all the adjacent cells of a mine by 1, such method is invoked in the SetMines() method, whereby for every time a mine is mapped on the board, the adjacent cells are incremented. This approach was used to limit the computation of counting the number of adjacent mines of a cell during runtime. This method also utilises the IsValid() and IsMine() methods, to validate the adjacent cells.
7. ShowCells() – This method is used to show the 8 adjacent cells and current cell , by looping through the positions of the cells and setting isShowing to true.
8. CheckWin() – This method is used to determine whether the player has won the game, by counting the number of cells which are showing, and determining whether it is less than the grid size subtracted by the number of mines. If the size of the grid subtracted by the number of mines is larger than the number of cells counted, then the method returns false.
9. ShowAllCells() – This method is called at the end of game, which shows all the cells in the grid. This was achieved by setting all the cells' is Showing to true, and calling the Display() method.

Proper Input Validation was achieved through the InputValidation class, which included a Scanner instance, the default constructor,and a method called GetIntInput(String, int, int). Subsequently, the latter parameter takes a String and two integer inputs; the string is used to guide the user on what type of value should be inputted. Additionally, the two integers specify the range of which a correct int input should fall in. This method makes use of an infinite loop, and a try catch to determine any input mismatch exceptions the user may invoke, which it keeps on repeating until a correct input in entered. The inclusion of the mentioned parameters facilitates the invoking of the same method for both the inputting of rows and columns, hence adheres to OOP practices, through reuse of code.

The MainClass is the class, which is responsible for connecting all the previously, mentioned classes together. The class creates instances of the Board and InputValidation classes, then proceeds to initialise the board, set the mines, whilst displaying a menu and the Minesweeper board to the user. The program loops until the game is won. For every turn it requests the user to input a valid position of a cell on the board. The program checks whether current cell is a mine. If being the case it will terminate the game, otherwise it will proceed to check whether the current cell has the cellContent of "00", which indicates that no adjacent cells contain a mine. In doing so, it thus display's the current cell, and its adjacent cells. In case the cell is not "00" only the current cell is shown. The program then checks whether the player has won the game, and if not will proceed to execute the next round.

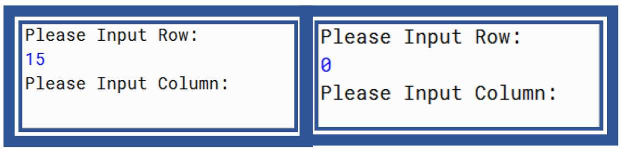
Testing

Please note that the same input and validation method, was used for both rows and columns. In the light of this, the testing was only performed on the row input, since the functionalities of the row input is replicated for the column input, thus any problems with the game would easily be identified when testing for row input.

Input Type	Input	Actual Output	Targeted Output	Figure
Valid Input	Int x | 0<=x<=15	Accepted	Accepted	Figure 27
Valid Input	0007	Accepted	Accepted	Figure 28
Invalid Input	Int x | x<0 ^ x>15	Not Accepted	Not Accepted	Figure 29
Invalid Input	String s, double d	Not Accepted	Not Accepted	Figure 30
Invalid Input	Cell which is showing	Do Nothing	Do Nothing	Figure 31
N/A	Winning condition*	Accepted	Accepted	Figure 32
N/A	Losing condition**	Accepted	Accepted	Figure 33

* For the purpose of this test the number of mines was set to 255 and all mines were set to isShowing=true.

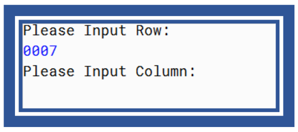
**For the purpose of this test the number of mines was set to 256 i.e., all the cells in the board.



```
Please Input Row:
15
Please Input Column:
```

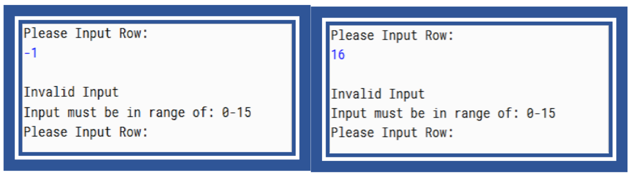
```
Please Input Row:
0
Please Input Column:
```

Figure 27: Valid Input 1



```
Please Input Row:
0007
Please Input Column:
```

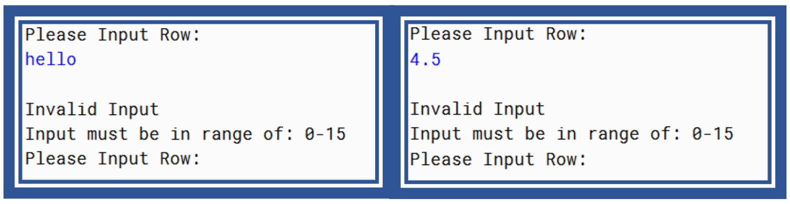
Figure 28: Valid Input 2



```
Please Input Row:
-1

Invalid Input
Input must be in range of: 0-15
Please Input Row:
```

```
Please Input Row:
16

Invalid Input
Input must be in range of: 0-15
Please Input Row:
```

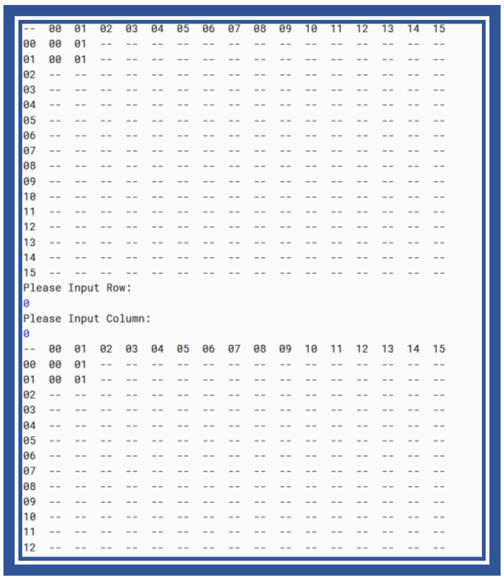
Figure 29: Invalid Input 1



```
Please Input Row:
hello

Invalid Input
Input must be in range of: 0-15
Please Input Row:
```

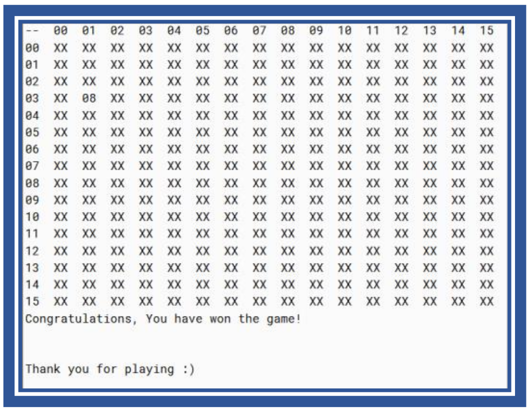
```
Please Input Row:
4.5

Invalid Input
Input must be in range of: 0-15
Please Input Row:
```

Figure 30: Invalid Input 2



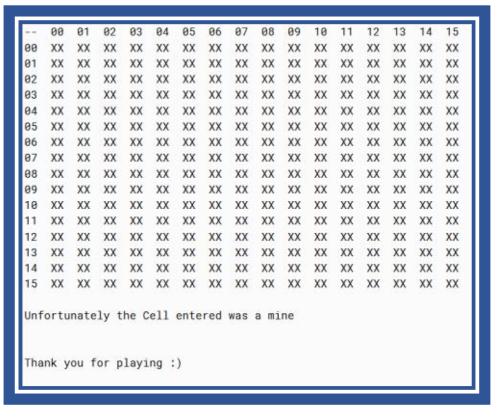
```
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  00  01  --  --  --  --  --  --  --  --  --  --  --  --  --  --
01  00  01  --  --  --  --  --  --  --  --  --  --  --  --  --  --
02  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
03  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
04  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
05  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
06  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
07  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
08  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
09  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
11  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
12  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
13  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
14  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
15  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
Please Input Row:
0
Please Input Column:
0
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  00  01  --  --  --  --  --  --  --  --  --  --  --  --  --  --
01  00  01  --  --  --  --  --  --  --  --  --  --  --  --  --  --
02  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
03  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
04  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
05  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
06  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
07  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
08  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
09  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
11  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
12  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

Figure 31: Invalid Input 3



```
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
01  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
02  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
03  XX  08  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
04  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
05  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
06  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
07  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
08  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
09  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
10  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
11  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
12  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
13  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
14  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
15  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
Congratulations, You have won the game!


Thank you for playing :)
```

Figure 32: Testing winning condition



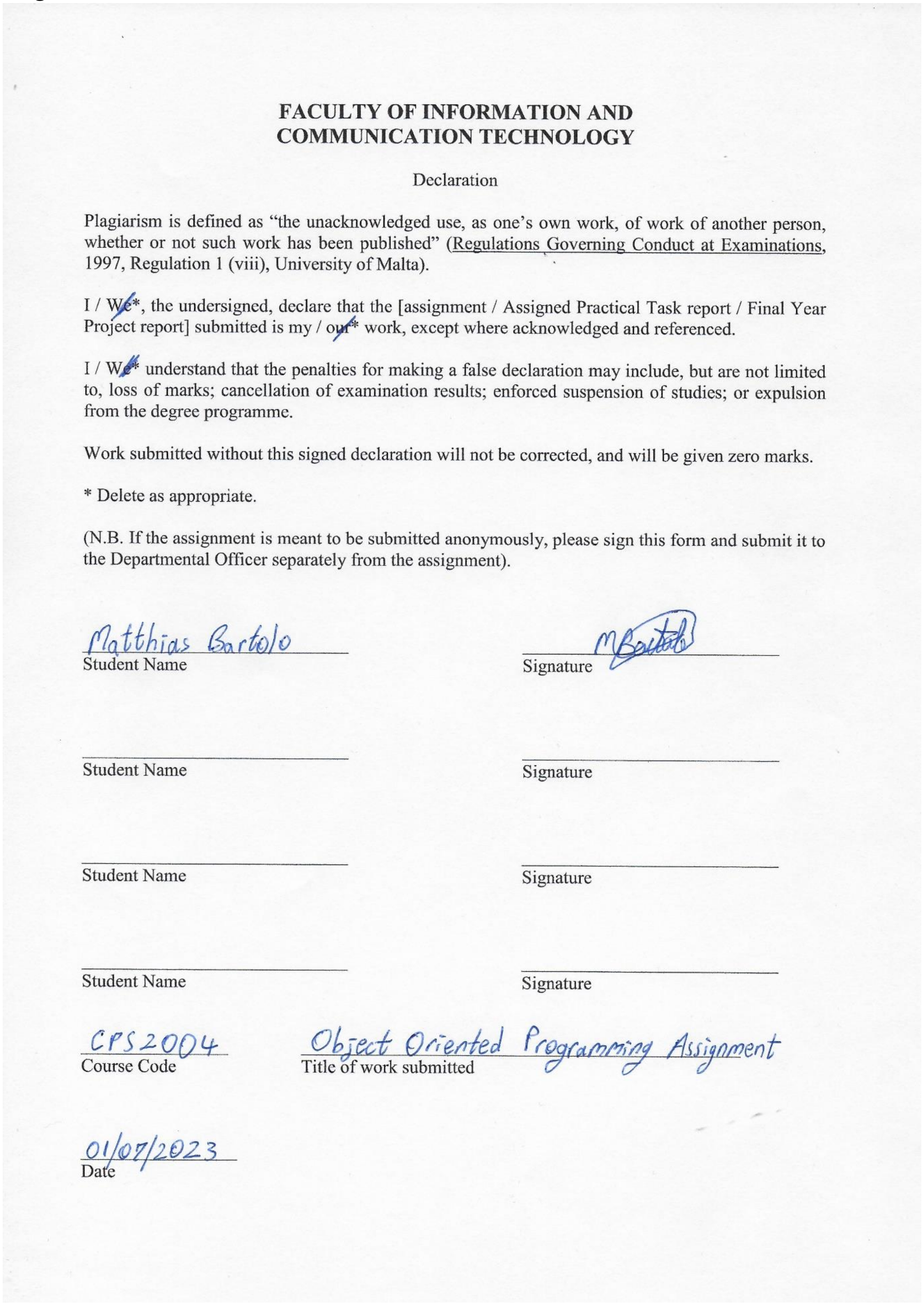
```
--  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
00  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
01  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
02  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
03  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
04  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
05  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
06  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
07  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
08  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
09  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
10  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
11  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
12  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
13  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
14  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX
15  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX  XX

Unfortunately the Cell entered was a mine


Thank you for playing :)
```

Figure 33: Testing losing condition

Critical Evaluation

Although the program was implemented in a proper and optimised way, as specified in the assignment brief, it omits some features present in a full Minesweeper game. The implementation mentioned above, could be extended to clear all the adjacent cells which contain a 0, rather than just the limitation of the 8 adjacent cells. This extension could be facilitated through the implementation of a recursive method and a stack.

Plagiarism Declaration Form



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / ~~We~~*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / ~~our~~* work, except where acknowledged and referenced.

I / ~~We~~* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthias Bartolo	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

CPS2004	Object Oriented Programming Assignment
Course Code	Title of work submitted

01/07/2023
Date