Object Oriented Programming Report

Giorgio Grigolo

Semester 1, 2022

Abstract

This report is a summary of the Object Oriented Programming course.

Contents

1 Village War Game — A Java Implementation **2**
1.1 Introduction . . . 2
1.2 Implementation . . . 2
1.2.1 UML . . . 2
1.3 Design Choices . . . 3
1.4 Testing . . . 8

2 Minesweeper — A C++ Implementation **9**
2.1 Introduction . . . 9
2.2 Implementation . . . 9
2.2.1 UML . . . 9
2.3 Design Choices . . . 10
2.4 Testing . . . 10
2.5 Critical Evaluation and Limitations . . . 10

1 Village War Game — A Java Implementation

1.1 Introduction

My village war game, code-named `Clash of Clubs` is an offline multiplayer turn-based strategy game, where the player is in charge of a village, and must defend it from the attacks of the enemy. The player must collect resources in order to build new buildings and upgrade existing ones, through which he can train troops, collect more resourcse and even attack the enemy.

The game also features an AI with configurable levels difficulty, which can be used to play against. It will take turns with the player, and will try make a fixed number of actions per turn, depending on the difficulty level. Note that the AI is quite smart, meaning that it will not make any invalid choices, like trying to purchase a building that is too expensive, or train a troop for which no training hut is available.

1.2 Implementation

1.2.1 UML

For simplicity, the top level UML only contains unidirectional dependencies, aggregations and compositions. More detailed subsections of this UML will be discussed later, where all the dependencies will be shown.

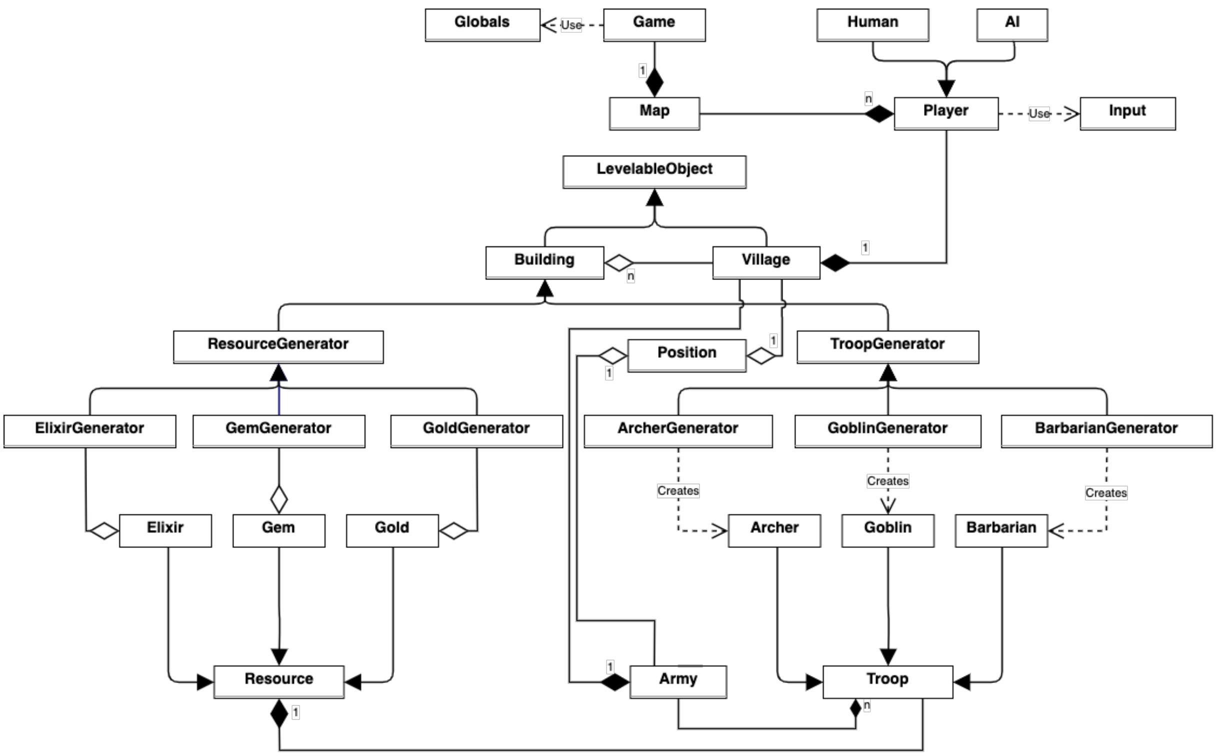

Figure 1: High level UML Diagram

1.3 Design Choices

In view of the fact that the game intrinsically exhibits multiple object oriented design patterns, it was noted that the design stage was crucial to the development of the game. Structurally, the project was split into a number of packages, which are the following:

- `game` — contains the main class, which is the entry point of the game, and the `Game` class, which is the main game loop.
- `players` — contains the `Player` class, which is the base class for a `Player`, and the `Human` and `AI` classes, which inherit the `Player` class. This package also contains the `Village` given that its ownership is tied to a single `Player`.
- `buildings` — contains the `Building` class, which is the base class for all buildings, and the respective building type subclasses, namely the `ResourceGenerator` and the `TroopGenerator` classes.
- `resources` — contains the `Resources` class, which is the base class for all troops, and the respective troop type subclasses, namely the `Gold`, `Elixir` and `Gem` classes.
- `troops` — contains the `Troop` class, which is the base class for all troops, and the respective troop type subclasses, namely the `Archer`, `Goblin` and `Barbarian` classes.
- `utils` — contains various helper singleton helper classes, to facilitate certain common tasks, as well as any miscellaneous classes that did not fit in the previous packages.
- `exceptions` — contains all of the exceptions that could be thrown by any action in the game.

In the following pages, I will go through most of the packages, and explain the interactions between them as well as the reasoning behind it.

All of the class methods will be omitted from the following UML as the sheer amount of them makes it unfeasible for them to be displayed neatly. However, any relevant function will be quoted and briefly explained when needed.

Game

As can be seen in the diagram hereunder, `Village`s are owned by `Player`s, and `Player`s are the ones that can perform actions that affect their Villages.

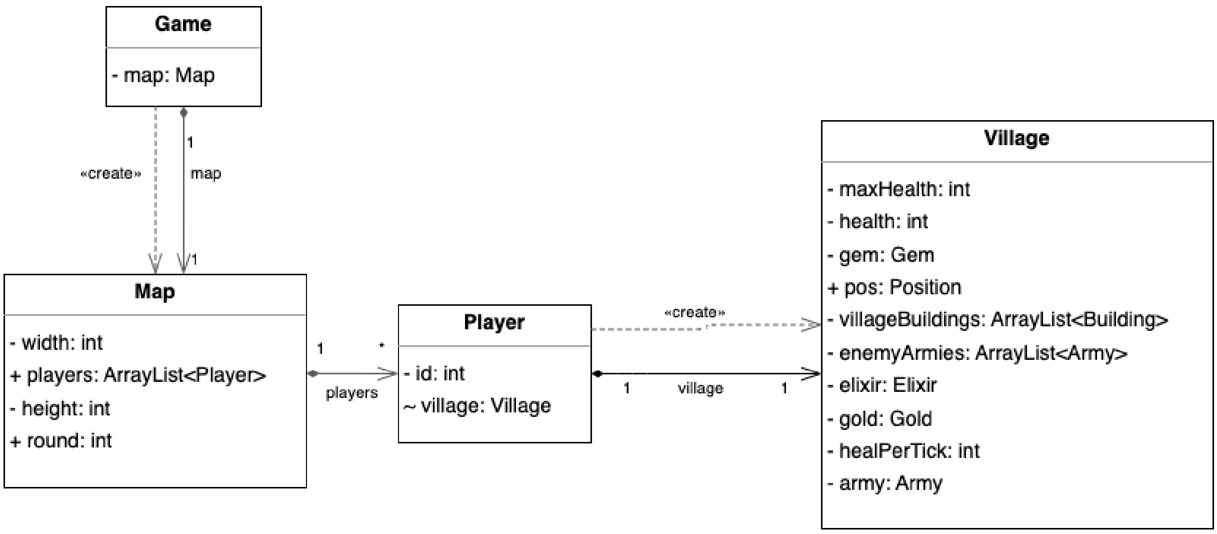

Figure 2: Game ↔ Player interactions.

The main game loop, called `doRound()` will recursively execute itself, until the win condition has been met. Every round, the following things happen:

1. Dead players are removed from the `players` array in the `Map`.
2. Each `Player` will have its `doTurn()` method executed.
3. The win condition check is executed.
4. Each player's (`Village`'s) `Army` will march to the set destination.
5. The round count will be incremented.

Player

We will now define the `doTurn()` method in the `Player` class, which, as explained in the previous section, is executed once for every player each round.

As per specification, a player's turn consists of the following stages

1. Friendly troop arrival — if the `Player`'s `Village`'s army has a distance of 0 ticks to cover, and its position is that of the village, we can safely conclude that it's at home, and that therefore each troop must empty their inventory in the appropriate village resource pools.
2. Enemy troop arrival — if the `Player`'s `Village`'s `enemyArmies` contains any armies that have no distance to cover, and their position is equal to that of the village, we can safely conclude that these armies are allowed to attack the `Village`. This attack phase will be discussed in more detail later.
3. Resource Earning — the `Player`'s `Village`'s `villageBuildings` array will be iterated through, and each `Building` will be asked to execute its `doTick()` method. For `troopGenerators`, this does nothing, as they need to be interacted with voluntarily. If the `Building` is a `resourceGenerator`, it will generate the appropriate amount of resources, and add them to the `Village`'s resource pool accordingly.
4. Player actions — according to the player type (`Human` or `AI`), the appropriate menu loop will be displayed, or the `Player` will take a fixed preconfigured number of random actions and pass it's turn. This is defined through overriden methods in the `Human` and `AI` classes. Furthermore, since we know that a `Player` has to be either a `Human` or an `AI`, we define the `playerInput()` method as abstract, to be later implemented in the respective subclasses.

Levelable Objects

Before we introduce the building types, we must first define the root object that they inherit. Given the similar system that entities must follow when leveling up, I thought it would be a good idea to define a base class for them, as they all would have to have a `level`, a specific `Resource` needed to level up, and a specific amount of that `Resource` needed to level up.

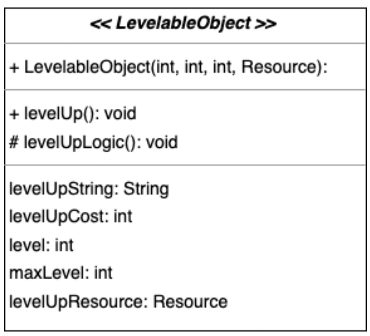

Figure 3: `LevelableObject` UML

The two methods that are defined in the `LevelableObject` class, are helper functions, to make sure that there is enough of the required `Resource` to level up and to actually level up the object.

However, each `LevelableObject` must implement the `levelUpLogic()` method, as different attributes will change when leveling up, always according to the inherited class type.

Buildings

All building types, inherit their main attributes from the `Building` super class, most notably the `buildCost` and all the interfaces to implement.

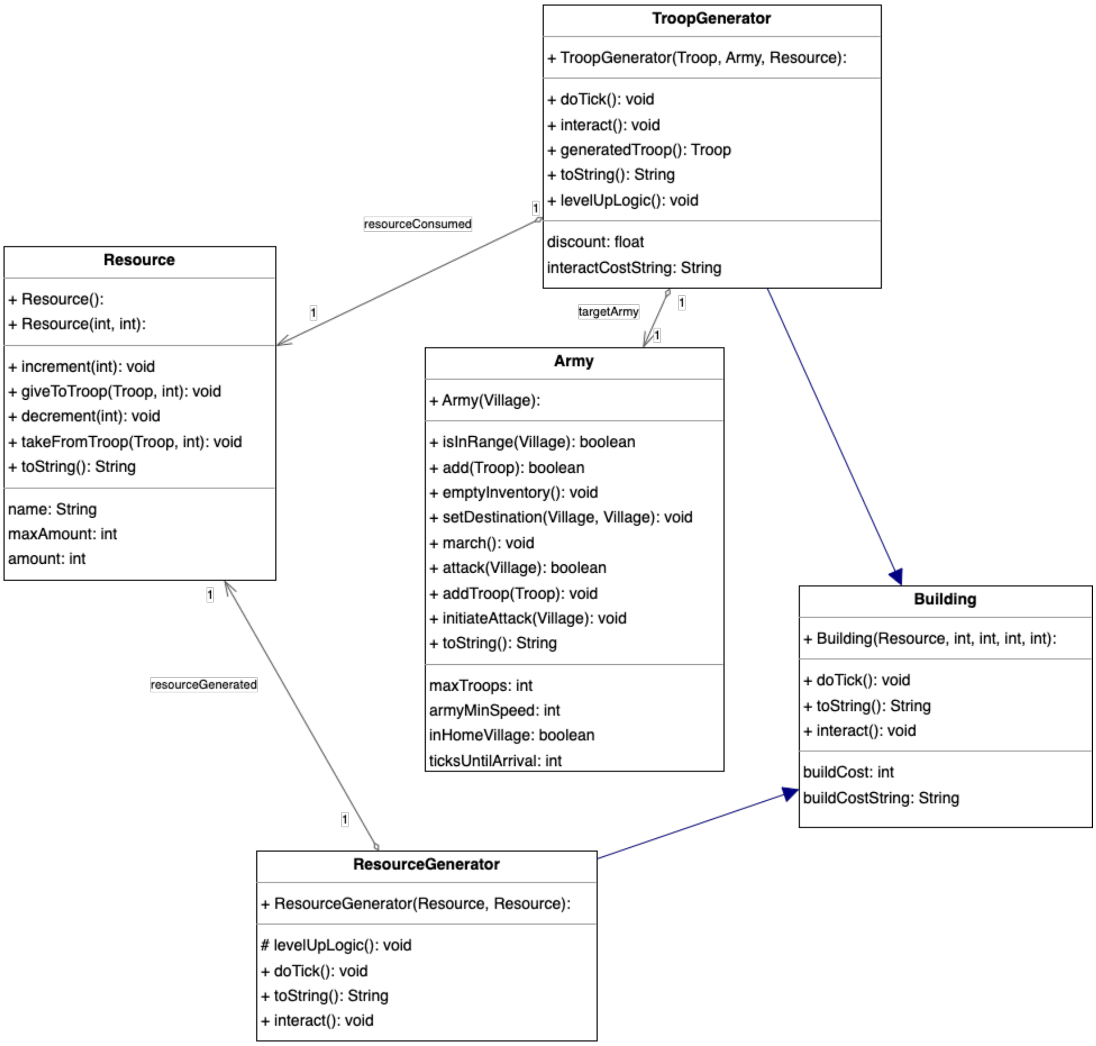

Figure 4: Building ↔ Resource interactions.

As we can see from the figure above, `troopGenerators` consume a specific type of `Resource` to generate troops, so that they can be assigned to the `targetArmy`. The `resourceGenerators` on the other hand, generate a specific type of `Resource`, which is then transferred to the `Village`'s resource pool.

It should be noted that the Java `toString()` method is overriden in each of the subclasses, to display the appropriate information for each building type in the user interface.

Resources

Given all of the above game-mechanics, we are still missing for a way to purchase all of the buildings and troops. This is where the `Resource` class as well as it sub-types come into play.

The two construtors in each of the `Resource` subclasses, are used to create a storage for that specific `Resource` type, where the two integer paramaters define the initial amount of that `Resource` type, and the maximum amount of that `Resource` type that can be stored which can be levelled up (only if it's the `Resource` owned by a `Village`).

We have seen how we can get some resources from the `resourceGenerators`, but we can get more `Resource`s by attacking other `Village`s, and brought to us by the `Army` that we send to attack.

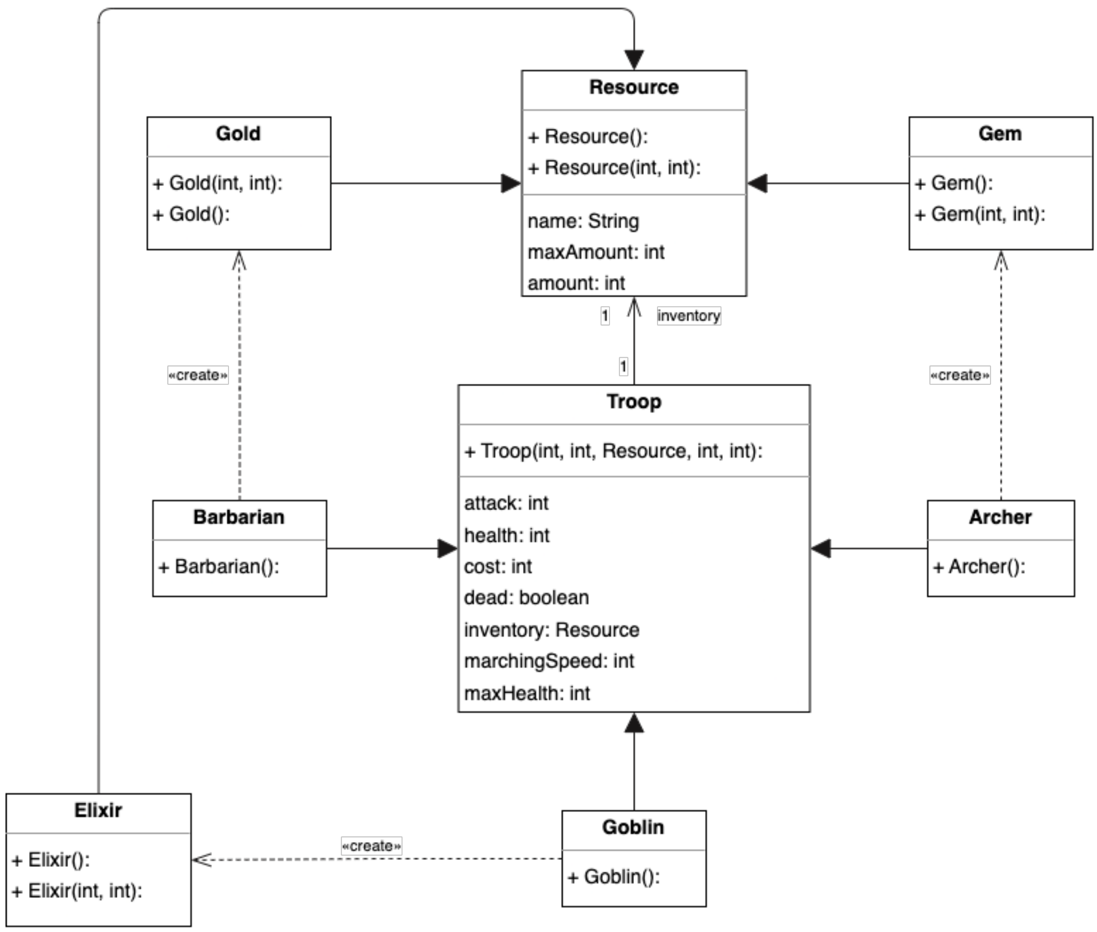

Figure 5: Resource ↔ Building interactions.

As can be seen above, each `Troop` in the `Army` will have an inventory that is only able to carry a specific type of `Resource`.

- `Gems` can only be carried by an `Archer` and is used to purchase a `Goblin`,
- `Elixir` can only be carried by a `Goblin` and is used to purchase a `Barbarian`,
- `Gold` can only be carried by a `Barbarian` and is used to purchase an `Archer`,

Note: All troops form a resource dependency cycle, as to equally distribute the types of `Resources` among expenses thus making the game more balanced.

Armies

To be able to attack other `Villages`, we need to have an `Army`, that is, a collection of troops that resides in our `Village`. The `Army` class extends the native Java `ArrayList` class, overloading some of its core methods, to make sure that the `Army` is always balanced, and that it can only contain a specific number of troops (depending on the level of the `Village`).

Figure 6: Army UML Diagram

When a user selects the attack option from the menu, they are greeted with a list of all the other `Villages` in the game, and are asked to select one of them. Once they have selected a `Village`, the `initiateAttack()` method is called, which will set the user's `Army`'s destination, and add it to the target `Village`'s encumbent `Army` queue.

A variety of helper functions, such as `isInRange()` and `inHomeVillage()` are used to quickly determine the position of an `Army`.

An attack follows a set of rules, whereby the specifications had to be slightly modified to make an attack possible. More particularly, step 3 of the Resolve Combat section of the specifications, had to be modified, as killing `Troops` until the total health of the `Troops` killed is equal to the total attack of the opposing `Army` might result in an infinite loop, in case the amount is not reached exactly, which is very improbable.

Instead, each `Troop` in the attacking `Army` will choose one random defending troop to attack (if any are present) and calls the `fight() Troop` method on it, until either the attacking troop is dead, or the defending army is dead.

The `fight()` method will bi-directionally deal damage to the two `Troops`, and if the defending `Troop` is dead, it will be removed from the defending `Army`.

An `Army` will stay in the `Village` it is attacking, until it is either defeated, or it has killed all of the `Troops` in the defending `Army`, at which point it will steal all of the `Resources` from the `Village` it is attacking, and start its trip back to its home `Village`. This game mechanic is used to make sure that the game is not too easy, and that the player has to be careful when attacking other `Villages` as it leaves them vulnerable to attacks from other players.

Position

The `Position` object is given to those entities that are expected to be able to move around the game map. In essence, the `Position` object is a wrapper around a pair of integers, that represent the x and y coordinates of the entity on the game map. This allows us to override the `equals()` method, and compare two `Positions` by their coordinates, instead of their memory addresses.

Furthermore, the `Position` object has a `distance(Position p)` that calculates the distance between two `Positions`, taking into account the grid-like structure of the map. In fact, this distance is defined over the taxi-cab metric, which is equivalent to

$$d(p_0, p_1) = |x_1 - x_0| + |y_1 - y_0|$$

Exceptions

Of course, a `Human` makes mistakes, and therefore so does our code. To make sure that our code is as robust as possible, we have implemented a variety of custom exceptions, that are thrown when an error occurs. These exceptions are then caught by the `Player` class, which displays nicely formatted error messages to the user, and allows them to try again the action they were trying to perform.

In this game, the following exceptions are thrown:

- `ArmyAwayException` is thrown when the user tries to perform an action on an `Army` that is not in their `Village`.
- `ArmyEmptyException` is thrown when the user tries to send an empty `Army` to attack.
- `ArmyFullException` is thrown when the user tries to add a `Troop` to an `Army` that is already full.
- `InsufficientResourcesException` is thrown when the user tries to perform an action that requires more `Resource`s than they have.
- `MaxLevelException` is thrown by a `LevelableObject` when the user tries to upgrade it to a level that is higher than the maximum level.

Globals

In order to allow maximum flexibility, we have implemented a `Globals` class, that contains all the constants that are used throughout the game. This allows us to easily change the values of these constants, without having to search through the code for all the places where they are used.

The statically loaded variables in the `Globals` class, define the default values of a myriad of properties of objects. Since I am not a game designer, I did not have the time to come up with balanced values for these properties, and therefore I have decided to make them configurable, so that they can be tweaked by the user.

1.4 Testing

Due to the sheer number of interactions and possible states that the game can be in, it is very difficult to test the game manually. Therefore, we have decided to implement a series of automated unit tests, that test the functionality various parts of the game in an isolated manner, making sure that they work as expected.

These can be found in the `tests` directory under the main package, and are written using the `JUnit` framework.

2 Minesweeper — A C++ Implementation

2.1 Introduction

Minesweeper is a logic puzzle video game genre generally played on personal computers. The game features a grid of clickable squares, with hidden "mines" scattered throughout the board. The objective is to clear the board without detonating any mines, with help from clues about the number of neighboring mines in each field. In this section (2), we will implement Minesweeper with the help of ncurses[1] and C++.

The game is played on a board of tiles, each of which is either a mine or empty. The player is initially presented with a board of tiles, and must use logic to deduce the locations of the mines. The player can click on a tile to reveal it. If the tile is a mine, the player loses. If the tile is empty, the tile will be revealed, and if it has no neighboring mines, all of its neighboring tiles will be revealed as well. If the tile has neighboring mines, the number of neighboring mines will be displayed on the tile. If the player marks all of the mines, the player wins.

2.2 Implementation

2.2.1 UML

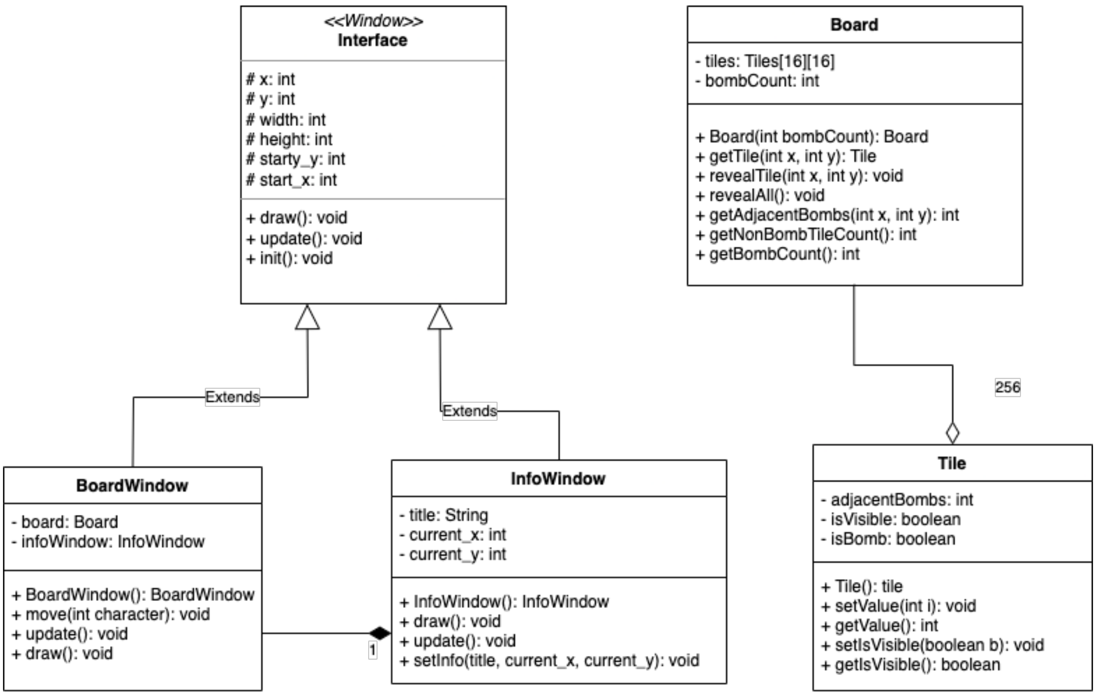

Figure 7: Minesweeper UML Diagram

[1]Ncurses is a programming library providing an application with a terminal-independent screen-painting and keyboard-handling facility in a text-mode environment.

2.3 Design Choices

The user interface, to be preferably displayed on a full screen terminal, is usable through the traditional 'wasd' keys, or the vim[2]-oriented 'hjkl' keys, and the 'Space' key to reveal a tile.

The user interface uses the ncurses library. Being by far the most difficult part of this program that I had to implement, it was the one that required most of my attention and time. Even though it originated in the form of a C library, I have created an abstract handler class `Window`, in such a way that I thought would best fit the needs of this program. The window class is inherited by the two windows that are shown to the user: the `BoardWindow` and the `InfoWindow`.

The `BoardWindow` will tile by tile, get the properties of the tile, and display them accordingly; either hidden or revealed. In case of the latter, the number of adjacent mines will be shown, in an appropriate color. The `InfoWindow` will display the current selected coordinates, mostly for debugging purposes. When the game is over be it won or lost, it will display the final message respectively.

Of course, each window type update and draw methods are different, as they have different properties to display. Whilst inheriting a common interface, they are still different classes.

As for the board, it made sense for it to act as a storage for the tiles, as well as a handler for the different tile states. One could say that the `Board` is a list-like data structure, with utility functions modifying the internal state for every atomic object inside of it, the `Tile`.

A `Tile` is just a POCO (a plain old C++ object), which means that the class only contains privated class variables, together with their public getters and setters, thus making it encapsulated and low-coupled.

2.4 Testing

The game, being simple in nature, is easy to test. I have tested the `Board` class by creating a board with a given number of mines, and then checking if the number of mines is correct. I have also tested the `Board` class by checking if the number of adjacent mines is correct.

This was done through manual inspection, together with a hidden keybind set to 'C' that would reveal all the tiles. A second debug keybind was set to 'B' which would turn the currently selected tile into a mine, to see that the adjacent mines were correctly calculated.

The win condition was tested by generating a board with all mines except for one tile, and clicking on it which is known to us given the `revealAll` function, which is called when 'C' is pressed.

A lot of care has been put in such a way that on quitting, (keybind: 'Q'), the windows are destroyed through the pipeline provided by ncurses `endwin()`, as to reduce as many memory leaks as possible.

2.5 Critical Evaluation and Limitations

The game is fully functional, and it is playable. It is also easy to use, and it is very intuitive. The game was developed with scalability in mind, and one can set the board size and the number of mines in the `Board` class.

However, the game is not perfect, and it has some limitations. One of these is the fact that user interface is not resizable, and it will look unplayable on a terminal smaller than 29x102 characters.

Also, since the mines are randomly generated, the game can be lost in the first move, which is not the way the original minesweeper works. The original minesweeper would randomly seed the board mine position randomization process after the first move, in such a way that your first move is never a mine.

[2] Vim is a free and open-source, screen-based text editor.