CPS2004 — Object Oriented Programming

Assignment

Juan Scerri

B.Sc. (Hons)(Melit.) Computing Science and Mathematics (Second Year)

December 28, 2022

Contents

List of Figures

Listings

1 Plagiarism Declaration

Plagiarism is defined as *"the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines"* (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include 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.

Juan Scerri	CPS2004	December 28, 2022
Student's full name	**Study-unit code**	**Date of submission**
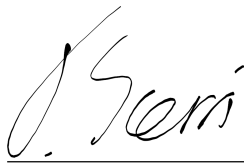
Title of submitted work: Object Oriented Programming Assignment

Student's signature

2 Village War Game

2.1 Language Choice

Java was chosen for the Village War Game because it has more complicated entity lifetimes. Programming the game in C++ would have required dealing with pointers to manage lifetimes. Obviously, dealing with pointers brings the possibility of memory leaks. Since Java has a Garbage Collector (GC) there is no need for manual deallocation of memory.

2.2   User Guide

2.2.1   Download, Compiling & Running

1. Clone the repository.

```
$ git clone https://github.com/JuanScerriE/village-war-game
```

2. Compile the game.

```
$ cd village-war-game ; ./compile.sh
```

3. Run the game.

```
village-war-game $ ./run.sh
```

2.2.2   Playing

```
Enter the number of human players
> 1
```

Figure 1: Picking the number of human players

Starting the game the player is asked to input the number of human players.

```
Enter the number of AI players
> 2
```

Figure 2: Picking the number of AI players

Then the player is asked to input the number of AI players.

```
Human 1
Info:
1.  Print village stats          2.  Print stationed troops stats
3.  Print building stats         4.  Print costs
5.  Print enemy villages         6.  Print armies
Actions:
7.  Build                        8.  Upgrade
9.  Train                        10. Attack
11. Pass
> ▌
```

Figure 3: Menu for the human players

The game loop will start. Each human player will be prompted with a menu of options. There are two categories: **Info** and **Actions**.

The **Info** category contains options which provide the player with information about his own village, enemy villages, armies and costs. Every option in the **Info** category is self–explanatory.

The **Actions** category contains options which affect the state of the game such as *building*, *training* and *attacking*. Finally, the player can decided to pass the turn to the next player.

All players by default will start with 50 "food", "metal" and "mana". The player can use these resources to build or upgrade buildings and train troops.

The following types of building can be built:

1. Academy (to generate wizards)

2. Foundation (to generate scouts)

3. Arena (to generate brawlers)

4. Farm (to generate food)

5. Mine (to generate metal)

6. Mana Tower (to generate mana)

As described in the above list there are three types of troops:

1. Wizard (high attack, low health, medium speed, low carrying capacity)

2. Brawler (high attack, high health, slow speed, medium carrying capacity)

3. Scout (medium attack, medium health, high speed, high carrying capacity)

Building

```
Human 1
Info:
1.  Print village stats        2.  Print stationed troops stats
3.  Print building stats       4.  Print costs
5.  Print enemy villages       6.  Print armies
Actions:
7.  Build                      8.  Upgrade
9.  Train                      10. Attack
11. Pass
> 7

Cost to build Farm: [Food: 10, Metal: 20, Mana: 5]
Cost to build Mine: [Food: 15, Metal: 20, Mana: 5]
Cost to build Mana Tower: [Food: 10, Metal: 25, Mana: 5]
Cost to build Academy: [Food: 10, Metal: 15, Mana: 10]
Cost to build Arena: [Food: 10, Metal: 15, Mana: 10]
Cost to build Foundation: [Food: 10, Metal: 15, Mana: 10]
1. Build Academy
2. Build Foundation
3. Build Arena
4. Build Farm
5. Build Mine
6. Build Mana Tower
7. Go Back
> 1
```

Figure 4: Building an Academy in the player's village

As depicted above in figure 4, the player is prompted with the different costs of the different buildings. If the player does not have enough resources for building he/she is notified.

```
Info:
1.  Print village stats        2.  Print stationed troops stats
3.  Print building stats       4.  Print costs
5.  Print enemy villages       6.  Print armies
Actions:
7.  Build                      8.  Upgrade
9.  Train                      10. Attack
11. Pass
> 3
Resource Buildings:
No buildings!
Troop Buildings:
— Academy, Level 1
```

Figure 5: Info about buildings in the player's village

As shown in figure 7, the buildings can have different levels of productivity. The initial level of productivity for all buildings is 1.

Upgrading

```
Info:
1.  Print village stats           2.  Print stationed troops stats
3.  Print building stats          4.  Print costs
5.  Print enemy villages          6.  Print armies
Actions:
7.  Build                         8.  Upgrade
9.  Train                         10. Attack
11. Pass
> 8

Cost to upgrade Farm: [Food: 5, Metal: 25, Mana: 5]
Cost to upgrade Mine: [Food: 10, Metal: 20, Mana: 10]
Cost to upgrade Mana Tower: [Food: 5, Metal: 20, Mana: 15]
Cost to upgrade Academy: [Food: 5, Metal: 10, Mana: 20]
Cost to upgrade Arena: [Food: 5, Metal: 10, Mana: 20]
Cost to upgrade Foundation: [Food: 5, Metal: 10, Mana: 20]
1. Upgrade Academy
2. Upgrade Foundation
3. Upgrade Arena
4. Upgrade Farm
5. Upgrade Mine
6. Upgrade Mana Tower
7. Go Back
> 1
```

Figure 6: Upgrading an Academy in the player's village

Upgrading the building increases the building's level of productivity. This has the effect of generating more resources or troops per round. Similarly, if the player does not have enough resources for upgrading he/she is notified.

Additionally, the building which is upgraded first is the oldest building.

Training

```
Info:
1.  Print village stats          2.  Print stationed troops stats
3.  Print building stats         4.  Print costs
5.  Print enemy villages         6.  Print armies
Actions:
7.  Build                        8.  Upgrade
9.  Train                        10. Attack
11. Pass
> 9

Cost to train Wizard: [Food: 2, Metal: 0, Mana: 3]
Cost to train Brawler: [Food: 2, Metal: 3, Mana: 1]
Cost to train Scout: [Food: 2, Metal: 2, Mana: 1]
1. Train Wizards
2. Train Brawlers
3. Train Scouts
4. Go Back
> 1
Input number of troops
> 2
```

Figure 7: Training Wizards in the player's village

Training troops increases the overall stats of the chosen troops. Similarly if the player does not have enough resources or troops he/she is notified.

Additionally, the training scheme is the same as for buildings i.e. the oldest troops are trained first, and each troop has an initial level of 1 and a maximum level of 3.

Attacking

```
Info:
1.  Print village stats          2.  Print stationed troops stats
3.  Print building stats         4.  Print costs
5.  Print enemy villages         6.  Print armies
Actions:
7.  Build                        8.  Upgrade
9.  Train                        10. Attack
11. Pass
> 10
Enemy Villages:
1. AI 1
Location: (5, 9)
Health: 1000
Resources: [Food: 30, Metal: 10, Mana: 51]
2. AI 2
Location: (27, 4)
Health: 1000
Resources: [Food: 15, Metal: 8, Mana: 34]
Input village number
> 1
<Station>
Number of Wizards: 9
Number of Brawlers: 0
Number of Scouts: 0
Total Attack Power: 184
Total Carrying Capacity: 47
Slowest Movement Speed: 2
Input number of wizards
> 9
Input number of brawlers
> 0
Input number of scouts
> 0
```

Figure 8: Attacking another village with Wizards in the player's village

To attack another village the player needs to pick a enemy village to attack. Then the player must specify the composition of the army.

Again if the player does not have enough troops he/she is notified.

The troops will be moved from the village station into the army. This means that the village will have less defensive power making it more vulnerable to attack.

```
Info:
1.  Print village stats          2.  Print stationed troops stats
3.  Print building stats         4.  Print costs
5.  Print enemy villages         6.  Print armies
Actions:
7.  Build                        8.  Upgrade
9.  Train                        10. Attack
11. Pass
> 6
<Army>
Sent By: Human 1
Attacking: AI 1
Location: (27, 35)
Number of Wizards: 9
Number of Brawlers: 0
Number of Scouts: 0
Total Attack Power: 184
Total Carrying Capacity: 47
Slowest Movement Speed: 2
```

Figure 9: Viewing an army marching towards an enemy village

Additionally, when the army arrives at it destination the player will be notified of whether the attack was successful or unsuccessful.

Incorrect Player Input

In all instances where the player input is unexpected or incorrect, the player is notified. All error messages are handled in the `HumanPlayer` class.

Moreover, error handling was performed by returning a `Status` enum which contains a number of possible errors which the program might encounter.

Note: Exceptions were avoided at all costs as they obscure control flow. Moreover, most errors do not arise because of exceptional circumstances. Exceptions were handled only when thrown from the standard library.

Quitting

Any human player can quit by pressing Ctrl-C.
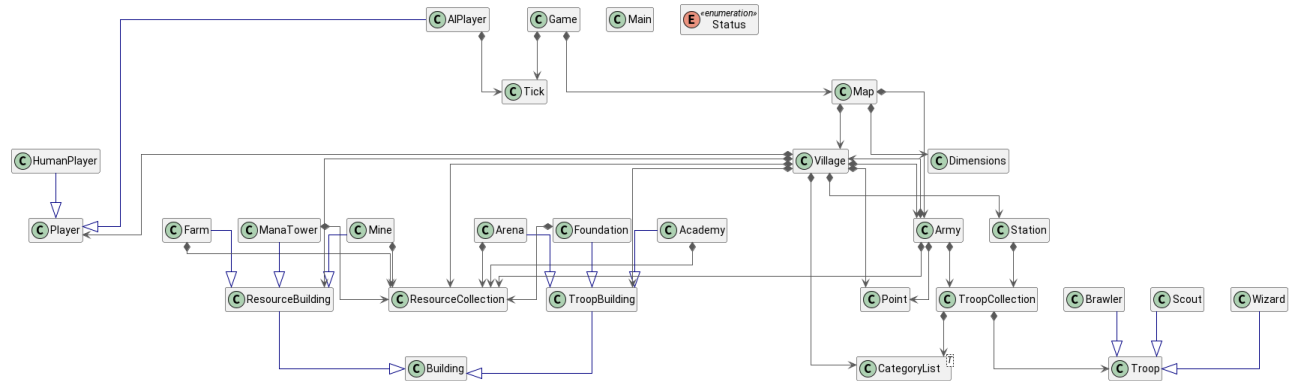
2.3   Design

Figure 10: A UML diagram (without members) of the game

Note: The full class diagrams will not be displayed as they would consume to many pages.

In terms of design the specification was followed closely.

2.3.1   The Game Class

The Game class is essentially the entry point to our program. The main method will create a new instance of the Game class and it will call the `run()` method to start the game. The Game class houses a Map and has two main phases, the setup phase and the game loop phase. The setup phase essentially creates all the objects and which are required at the start.

Note: The size of the Map varies depending on the number of players. Moreover, it tries to spread out all the villages evenly on the map to ensure that no villages are to close to each other.

The game loop phases is essentially, where all the game play start and operations which need to occur everything every round of the game are done. Also, every round the game tick is increased. This allows us to keep track and try to quantify at what stage the game is.

2.3.2   Map & Village Design

The Map class will only contain two lists. These are a list of villages and a list of armies. This approach instead or actually storing everything in a 2D array has a couple of benefits. Firstly, this makes dealing with objects a lot easier since there is no need to move objects from one location in to another in the array. Also, it is easier and more efficient to just store the location of the entity directly with the entity. This means that getting the location of an entity is fast. Moreover, it also means that you can have multiple objects which occupy the same location without interfering with each other. This is obviously useful since we do not have to worry about intersecting armies.

The Village of each player is essentially the hub where all actions are performed. This also unfortunately means that the village has to be polluted with a lot of action methods. This is only the case due to the limitations present in Java's generics and reflection capabilities.

Note: This is ideally circumvented by actually, dropping all OOP design and instead using singular entities like: Troop and Building, then providing a builder which constructs Troops and Buildings with specific statics and identifiers remove the need for reflection and generics to deal will all types of troops and buildings, whilst retaining the possibility of quickly extending the types of troops and buildings very easily.

2.3.3   Player Design

For the player, there is a main abstract class and two subclasses which implement the interface defined in the abstract class. The two subclasses are AIPlayer and HumanPlayer. Then each Village has a Player field which we can populate with a player. Since both types of players have the same interface and they can interact with the village in the same way. Also it makes swapping between an AI player and human player easy.

Note: A simple reference to the Tick object present in the game object is also present in all AI players. This can allow for more complicated AI behaviour because it allows for the artificial creation of game stages similar to chess for example: the opening, mid-game and the end-game.

2.3.4   Resource Design

In particular in terms implementation since we need a substantial amount of resources to almost anything in the game it simply does not make sense to create a Resource class and then subclasses of specific types of resources. The issue with doing this is that handling the resources would have been very difficult. Additionally, it would have required the continuous creation of resources which would have definitely taken time since allocating memory is an expensive operation. Also as objects they would have carried no internal state what so ever and only the presence of that type of which add to the cumulative sum of resources makes has meaning. Hence, everything was design around having one ResourceCollection which holds counters of the amount of resources of all the different types.

This substantially improves performance since there is no need for a lot of allocations and it means that overall handling resources is less cumbersome.

2.3.5   Troop Design

Again there is a Troop abstract class which which specifies the behaviour of all troops and then there are subclasses which initialize their fields with specific values according to the strength and weaknesses each Troop type (as described above in the player guide.)

Further more an important data structure built on top of the CategoryList is the TroopCollection.

This class facilitates the grouping of all troops even of different class types into one object. This object is able to send and receive troops, calculate the total attack power etc.

Additionally, we have to other classes which compose a TroopCollection. These are Army and Station. The Army as the name implies is able to roam the map whilst the Station is part of the village.

The difference between these two is that the Army is able to move and exist separately of the village. Additionally, it has a lot of extra methods which facilitate combat with a village.

2.3.6   Building Design

This is the biggest inheritance hierarchy. There is a main abstract class called Building and two abstract subclasses which are TroopBuilding and ResourceBuilding which differentiate between troop generating buildings and resource generating buildings. Initially, the reason for the split was done under the assumption that the troop buildings require an additional method to train troops. However, this was no longer needed later on in development. However, changing would have resulted in changing quite a few parts of the code to deal with a single level of inheritance. So, this system was kept. Since training is not actually done by the building but by a method in the troop, to follow game specifications a check to ensure that at building of the appropriate type exists was put in place.

Moreover, the Village class contains two category lists of type TroopBuilding and type Resource-Building because the CategoryList class is not powerful enough provide a mechanism where you can group any collection of objects by the respective category even if it may be a super class. Also using such a data structure is either guaranteed to be slow due to requirement of iteration or memory hungry requiring separate list for all the classes in the inheritance hierarchy to accommodate searching with any possible class in the hierarchy.

2.4   Technical Aspects

In general, give the above design which is very technical it meant that most of the coding was simply routine coding with nothing out of the ordinary. The only notable mentions is the extensive use of design patterns specifically the builder, the singleton and the observer and the CategoryList.

```
public static class Builder {
    private int _food = 0;
    private int _metal = 0;
    private int _mana = 0;

    public ResourceCollection build() {
        return new ResourceCollection(_food, _metal, _mana);
    }
}
```

```java
    public Builder setFood(int food) {
        if (food >= 0) {
            _food = food;
        }

        return this;
    }

    public Builder setMetal(int metal) {
        if (metal >= 0) {
            _metal = metal;
        }

        return this;
    }

    public Builder setMana(int mana) {
        if (mana >= 0) {
            _mana = mana;
        }

        return this;
    }
}
```

Listing 1: Builder pattern used for the ResourceCollection

As we can see the above code is implementing the builder pattern. This was helpful because it allowed for the easier creation of resource collection as sometimes they are also used to denote cost for example upgrading a building.

```java
public class Tick {
    private static Tick _instance = null;

    private BigInteger _tick = BigInteger.ZERO;
    private Tick() {}

    public static Tick getInstance() {
        if (_instance == null) {
            _instance = new Tick();
        }

        return _instance;
    }

    // ...
```

```
}
```

Listing 2: The game Tick class is a singleton

Again similarly, this was useful to ensure that one can use the Tick object where ever it useful and ensure that there is only one instance per game.

```java
public abstract class Player {
    private final String _name;
    public Player(String name) {
        _name = name;
    }
    public String getName() {
        return _name;
    }
    public abstract void actions(Village village);
    public abstract void notify(String text);
}
```

Listing 3: The Player class (contains the notify for the observer pattern)

```java
public Village enemyTroopArrival() {
    List<Army> clonedArmies = new LinkedList<>(_armies);

    for (var army : clonedArmies) {
        if (army.isEnemy(this) && army.arrived()) {
            if (isDestroyed()) {
                army.goBack();
            } else {
                if (!army.attack().isSuccessful()) {
                    army.getSender().getPlayer().notify("The below army
  has been defeated\n" + army);
                    _armies.remove(army);
                }
            }
        }
    }

    return this;
}
```

Listing 4: `notify()` being used to notify the player that a army has been defeated in Village.java

The observer pattern is used because it is useful for notifying players of events in game. Specifically, it is used when armies are defeated or when they arrive back. This helps the player gain useful information which he/she wouldn't have since the player's village will never actually interact with the defeated army again.

2.5   Testing

```
Enter the number of human players
> -1
Invalid: must be a greater than or equal to 1
Enter the number of human players
> a
Invalid: must be a integer
Enter the number of human players
> 1
Enter the number of AI players
> 2
```

Figure 11: Demonstrating invalid option (out–of–bounds) and failure to parse repectively

So, there are three types of failures which our application has to cater for:

1. Incorrect input i.e. failure to parse input into correct type.

2. Failure due to invalid option or out of bounds e.g. option 12 does not exist.

3. Failure due to insufficient resources or troops.

```
Info:
1.  Print village stats         2.  Print stationed troops stats
3.  Print building stats        4.  Print costs
5.  Print enemy villages        6.  Print armies
Actions:
7.  Build                       8.  Upgrade
9.  Train                       10. Attack
11. Pass
> 8

Cost to upgrade Farm: [Food: 5, Metal: 25, Mana: 5]
Cost to upgrade Mine: [Food: 10, Metal: 20, Mana: 10]
Cost to upgrade Mana Tower: [Food: 5, Metal: 20, Mana: 15]
Cost to upgrade Academy: [Food: 5, Metal: 10, Mana: 20]
Cost to upgrade Arena: [Food: 5, Metal: 10, Mana: 20]
Cost to upgrade Foundation: [Food: 5, Metal: 10, Mana: 20]
1. Upgrade Academy
2. Upgrade Foundation
3. Upgrade Arena
4. Upgrade Farm
5. Upgrade Mine
6. Upgrade Mana Tower
7. Go Back
> 2
Not enough resources!
```

Figure 12: Demonstrating "not enough" resources failure

Each input was manually tested to ensure that it is not possible to plunge the game into an invalid where behaviour is not properly defined.

```
The below army has been defeated
<Army>
Sent By: Human 1
Attacking: AI 1
Location: (3, 26)
Number of Wizards: 0
Number of Brawlers: 0
Number of Scouts: 0
Total Attack Power: 0
Total Carrying Capacity: 0
Slowest Movement Speed: 0
AI 1
AI 2
Human 1
Info:
1.  Print village stats          2.  Print stationed troops stats
3.  Print building stats         4.  Print costs
5.  Print enemy villages         6.  Print armies
Actions:
7.  Build                        8.  Upgrade
9.  Train                        10. Attack
11. Pass
```

Figure 13: Demonstration of a army defeat notification being manually tested

Additionally, the game was further manually tested in a form of white-box testing to ensure that the game behaviour is as expected.

2.6 Limitations & Improvements

Limitation: The user interface is very hard to navigate around and not intuitive. Additionally, crucial information for the player to play strategically, is hidden behind info options which is decision taken to reduce screen clutter.

Solution: Creating a GUI which the user can use would make the overall experience of playing the game better. This is because the user interface can be better to provide the user with important information quickly and easily accessible actions.

Limitation: Dealing with different object types which all have identical features just different values is not ideal. This is because it increases complexity due to the restrictions of the type checking in a language. To circumvent this issue facilities such as generics and reflection are used. However, in Java these are not really first-class features with the same power as in other languages.

Solution: Settling on a simpler entity system where inheritance is a avoided as much as possible reduces issues related type checking enforced by the compiler and hence there is no need for dealing

with generics or reflection making the overall code significantly simpler.

Limitation: The amount of options and choices the user can make is limited. This is mainly due to the how clunky the user interface it.

Solution: Changing to GUI can allow the application to grow in terms of complexity

3 Minesweeper

3.1 Language Choice

C++ was chosen for Minesweeper because it has a fixed board size of 16×16. This means that it is possible to stack allocate every object removing the need for dynamic memory allocation. This is facilitated by `std::array` from the Standard Template Library (STL) which allows for the creation of fixed size arrays on the stack.

3.2 User Guide

3.2.1 Download, Compiling & Running

1. Clone the repository.

```
$ git clone https://github.com/JuanScerriE/minesweeper
```

2. Compile the tests and the game.
   Note: Make sure that `gtest` and `ncurses` are installed for the tests and the game, respectively.

```
$ cd minesweeper ; ./compile.sh
```

3. Run the tests.
   Note: Some tests might fail. This is because the implementation of `srand` and `rand` differ between platforms (specifically macOS and Linux).

```
minesweeper $ ./tests.sh
```

4. Run the game.

```
minesweeper $ ./run.sh
```

3.2.2   Playing

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



 Pos:       (00, 00)
 q:         Quit
 h,j,k,l: Left, down, up, right
 SPACE:    Reveal hidden cell
 r:         Reset board
```

Figure 14: Playing Minesweeper

The general guide to playing Minesweeper is described above in figure 14.

Note: vim motions are used to move the cursor.

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


 YOU HAVE HIT A MINE! (Press r to retry)

 Pos:      (07, 02)
 q:        Quit
 h,j,k,l: Left, down, up, right
 SPACE:    Reveal hidden cell
 r:        Reset board
```

Figure 15: Hitting a mine

If the player hits a mine the above message as in figure 15 will be displayed.

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

YOU HAVE CLEARED THE BOARD! (Press r to retry)

Pos:      (03, 01)
q:        Quit
h,j,k,l:  Left, down, up, right
SPACE:    Reveal hidden cell
r:        Reset board
```

Figure 16: Finishing a game of Minesweeper

If the player manages to clear all the cells without hitting a mine the above message as in figure 16 will be displayed.

Finally, there is an additional *secret* command to automatically complete the board without hitting a mine. The user needs to press W (`shift + w`).

Note: This only works if the user has at least revealed one cell. This is because the board is populated with all the mines after the first reveal to ensure a player never hits a mine on his first try.
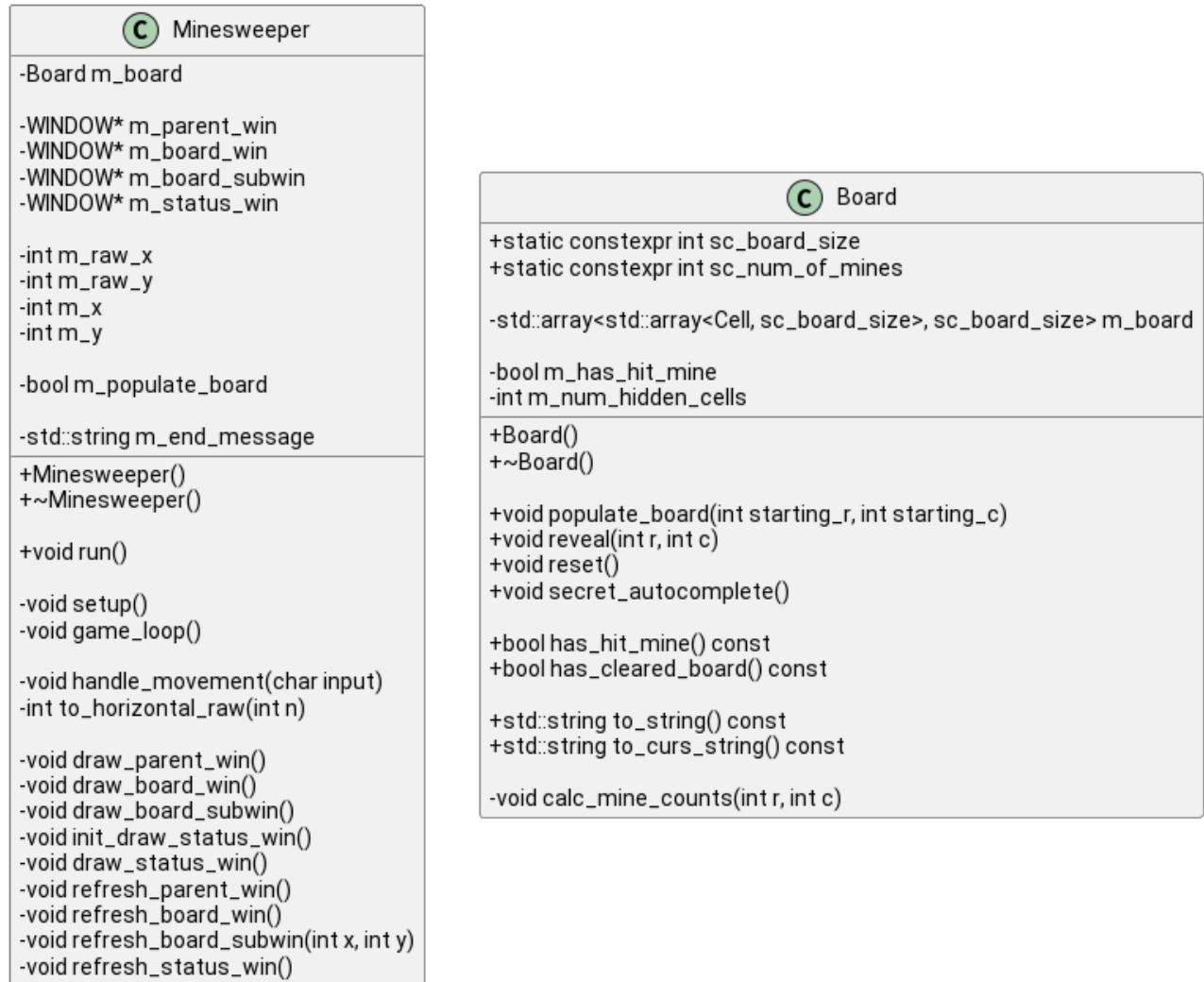
3.3   Design

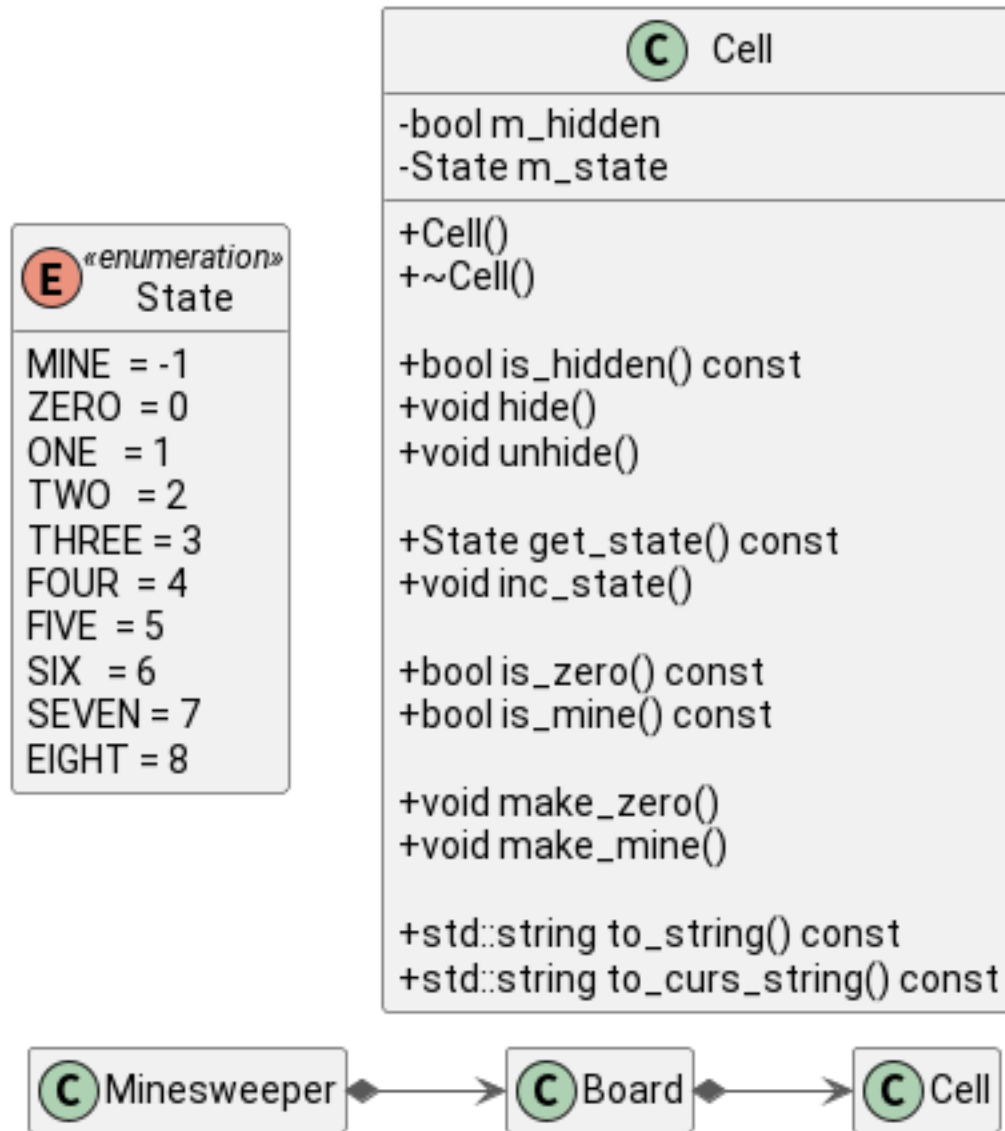Figure 17: Class diagrams of the Minesweeper class and the Board class

Figure 18: A class diagram of the Cell class State enum and a UML diagram (without members) of the game

The Minesweeper class contains the user interface code, that is it contains all `ncurses` specific code. The class also handles user input.

The Board class contains the majority of the game logic. It is a part of the Minesweeper class, that is if a Minesweeper object ceases to exist so does the Board object. Further more the actual board `m_board` is a grid of Cell objects. Also the lifetime of the objects is managed by the Board since when the board ceases to exist so do the cells.

3.4   Testing

```
[       OK ] CellTest.ZeroCellToString (0 ms)
[----------] 11 tests from CellTest (0 ms total)

[----------] 7 tests from BoardTest
[ RUN      ] BoardTest.PopulateBoard
[       OK ] BoardTest.PopulateBoard (0 ms)
[ RUN      ] BoardTest.HasHitMine
[       OK ] BoardTest.HasHitMine (0 ms)
[ RUN      ] BoardTest.HasNotHitMine
[       OK ] BoardTest.HasNotHitMine (0 ms)
[ RUN      ] BoardTest.HasNotClearedBoard
[       OK ] BoardTest.HasNotClearedBoard (0 ms)
[ RUN      ] BoardTest.RevealZeroCellAndNeighbouringCellsRecursively
[       OK ] BoardTest.RevealZeroCellAndNeighbouringCellsRecursively (0 ms)
[ RUN      ] BoardTest.HasClearedBoardWithoutHittingMineManually
[       OK ] BoardTest.HasClearedBoardWithoutHittingMineManually (0 ms)
[ RUN      ] BoardTest.HasClearedBoardWithoutHittingMineWithSecretAutocomplete
[       OK ] BoardTest.HasClearedBoardWithoutHittingMineWithSecretAutocomplete (0 ms)
[----------] 7 tests from BoardTest (0 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 18 tests.
```

Figure 19: Unit tests for Minesweeper (on macOS Ventura 13.1)

Black-box Testing and Unit Testing were used to test the application. For unit testing `gtest` is required.

```
% sudo leaks -atExit -- ./minesweeper
Password:
minesweeper(23642) MallocStackLogging: could not tag MSL-related memory as no_footprint, so those pages will be included in process footprint - (null)
minesweeper(23642) MallocStackLogging: recording malloc and VM allocation stacks using lite mode
Process 23642 is not debuggable. Due to security restrictions, leaks can only show or save contents of readonly memory of restricted processes.

Process:         minesweeper [23642]
Path:            /Users/USER/Desktop/*/minesweeper
Load Address:    0x1005b4000
Identifier:      minesweeper
Version:         0
Code Type:       ARM64
Platform:        macOS
Parent Process:  leaks [23641]

Date/Time:       2022-12-24 16:37:32.414 +0100
Launch Time:     2022-12-24 16:37:16.228 +0100
OS Version:      macOS 13.1 (22C65)
Report Version:  7
Analysis Tool:   /Applications/Xcode.app/Contents/Developer/usr/bin/leaks
Analysis Tool Version:  Xcode 14.2 (14C18)

Physical footprint:        3985K
Physical footprint (peak): 3985K
Idle exit: untracked
----

leaks Report Version: 4.0, multi-line stacks
Process 23642: 599 nodes malloced for 553 KB
Process 23642: 0 leaks for 0 total leaked bytes.
```

Figure 20: Testing for memory leaks (on macOS Ventura 13.1)

Furthermore, to test for memory leaks, on macOS, `leaks` was used and, on Linux, `valgrind` was used. `leaks` reported not memory leaks whilst `valgrind` reported leaks from `ncurses`.

3.5   Limitations & Improvements

Limitation: The unit tests are not cross-platform; four of the unit tests fail on Linux. This is mostly due to differing implementations of `srand()` and `rand()` on different platforms.

Solution: Create a custom random number generated. This guarantees the same result across different platforms.

Limitation: The ncurses library leaks memory on Linux whilst on macOS it does not.

Solution: This seems to be intended behaviour from ncurses. Read the man page at https://man7.org/linux/man-pages/man3/curs_memleaks.3x.html