CPS2004 — Object Oriented Programming

Assignment

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

February 10, 2023

Task 1: `https://github.com/JuanScerriE/village-war-game`
Commit: `482d6d3f2ba12d4530335e18c905ecfa30a1ae5e`

Task 2: `https://github.com/JuanScerriE/minesweeper`
Commit: `1867340ffdec99e1d9a60dae7a655757d523ffe0`

Contents

List of Figures

1 Village War Game

1.1 Language Choice

Java was chosen to avoid dealing with manual memory management since the Village War Game needs heap allocations to support manual entity lifetime management.

1.2 User Guide

1.2.1 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
```

1.2.2 Playing

Number of Human Players

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

Figure 1: Picking the number of human players

Number of AI Players

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

Figure 2: Picking 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: The menu for human players

Each human player is prompted with a menu of options having 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 is self–explanatory.

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

All players, by default, start with 50 "food", "metal" and "mana". The player can use resources or troops to perform the above described actions.

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)

And there six types of building:

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)

Note: It is useful to check the amount of resources you have using option 1 prior to performing any action.

Building

1. Use option 7 to get a list of all the different buildings and there build cost.
2. Pick from option 1 – 6 to build or 7 to go back.
3. You can view your new building by using option 3.

Upgrading

1. Use option 8 to get a list of all the different buildings and there upgrade cost.
2. Pick from option 1 – 6 to upgrade or 7 to go back.
3. You can view your upgraded building by using option 3.

Note: The player does not have granular control over which buildings to upgrade. Given the implementation, the oldest building of the specified type is upgraded first. However, buildings which have reached their maximum level cannot be upgraded further.

Training

1. Use option 9 to get a list of all the different troops and there training cost.
2. Pick from option 1 – 3 to train or 4 to go back.
3. Pick the number of troops to train.
4. You can view the contribution of your trained troops by using option 2.

Note: The user does not have granular control over which troops to train. Due to the movement of troops, they are essentially trained randomly. However, troops which have reached their maximum level cannot be trained further.

Attacking

1. Use option 10 to get a list of all enemy villages.
2. Pick a village, from 1 – n, to attack.
3. Pick the composition of the army i.e. the number of wizards, brawlers and scouts.
4. You can view the marching armies by using option 6.

Quitting

Any human player can quit by pressing `Ctrl-C`.

Player Notification

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

The most common notifications are caused by not having enough troops or not having enough resources.

1.3 Design

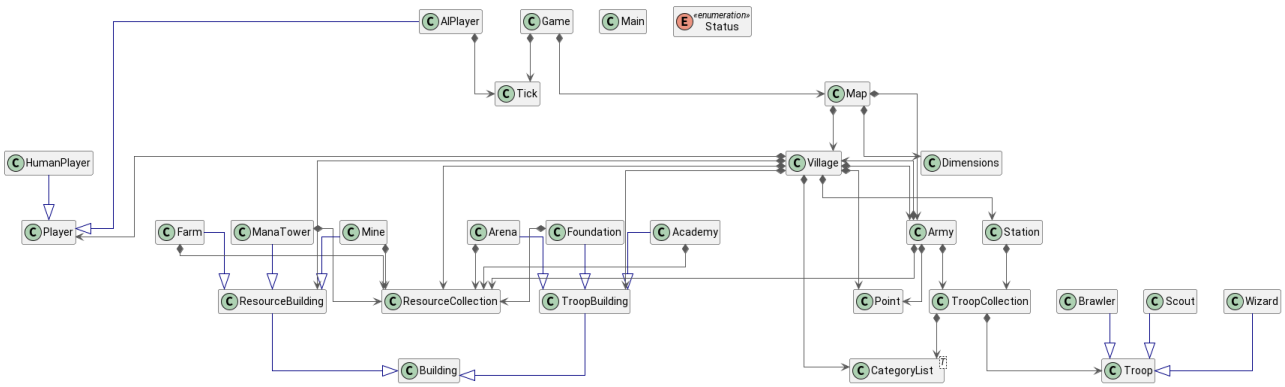

Figure 4: A UML diagram (without members) of the game

Note: The full UML diagram including members is not shown as it would take up too much space.

1.3.1 The Game Class

The `Game` class is the entry point of the game. It houses a `Map` object and has two main phases. The *setup* phase and the *game–loop* phase.

The *setup* phase creates and initialises all the objects required at the start.

The *game–loop* phase is where all the game play takes place. Each iteration of the loop is called a "round". The total number of rounds is tracked using a singleton `Tick` object.

Note: The size of the map varies depending on the number of players and the villages are spread out evenly across the map.

1.3.2 Map & Village Design

The `Map` class contains two lists: a list of villages and a list of armies, and each village and army contains a `Point` object to store its own location. This approach, instead of using a 2D array has the benefit of allowing multiple entities to exist at the same location. This is important since marching armies can have intersecting paths. Moreover, it makes managing entities significantly easier.

The village of each player is the hub where all actions are performed. Unfortunately this means that the village has to be polluted with a lot of action methods.

1.3.3 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`. Each village has a player and both types of players interact with the village using the same methods exposed by the `Village` class. This is because both players have an `actions()` method which takes in a `Village` object.

Note: A reference to the singleton `Tick` object is present in all AI players. This allows for more complicated AI behaviour because a whole game can be segmented into different stages for example: the opening, the mid-game and the end-game.

1.3.4 Resource Design

Since the resources carry no internal state and only the quantity of the resources matters there is no need for creating actual objects to represent the resources. Hence, everything was designed around having a `ResourceCollection` class which holds counters for each type of resource.

1.3.5 Troop Design

Again there is a `Troop` abstract class which specifies the behaviour of all troops and then there are subclasses which initialize their fields with specific values according to the strengths 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, `Army` and `Station` are tow other classes which compose a `TroopCollection`. Armies are different from stations because they are capable of marching across the map and engaging in combat with an enemy village. This is reflected in the additional methods and fields required.

1.3.6 Building Design

There is a main abstract class called `Building` and two abstract subclasses are called `TroopBuilding` and `ResourceBuilding`. These two subclasses facilitate differentiation between troop–generating buildings and resource–generating buildings. This split was done under the assumption that the troop buildings require an additional method to train troops. This was no longer the case later in development. Removing two subclasses would have required quite a few changes to the code. Consequently, the subclasses were kept.

Since training is not actually done by the building but by a method in the `Troop` class, to follow the game specification, a check was added to ensure that at least one building of the appropriate type exists.

1.4 Technical Aspects

1.4.1 Usage of Exceptions

Exceptions were avoided as they obscure control flow. Most errors do not arise because of exceptional circumstances. Exceptions were handled only when thrown from the standard library.

1.4.2 OOP & `CategoryList`

Using OOP and creating separate classes for the building and troop types added complexity to the code. This is because when it comes to generics and reflection Java is quite limited. This became an issue, because when trying to genericise the code to cater for multiple troop and building types, without needing to change the `Village` class, it proved to be very difficult.

It required the creation of a special data structure named a `CategoryList`. It is similar to a `HashMap`, however, the key is the type of the object. This data structure facilitates: getting a collection of objects by there type, iterating over all objects etc. However, it is still limited because it is not capable of handling multi–level inheritance hierarchies.

1.4.3 Singleton Design Pattern

Some objects such as `Tick` and `Map` require only one instance for for the whole game. Hence, the singleton pattern was used for these objects.

1.4.4 Observer Design Pattern

When an army is defeated, following the specification, the army is also destroyed. This requires that the player is notified of the defeat as it is information which can affect the decisions the player takes. Hence, the observer pattern is used to notify the player of the armies defeat.

Additionally, to decouple UI from the notification method used by the observer, it makes sense to add a message queue.

1.5 Testing

There are three types of failures which our application has to cater for:

1. Failure due to inability to parse player input.
2. Failure due to invalid or out–of–bounds option.
3. Failure due to insufficient resources or troops.

```
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 5: Demonstrating failure due to invalid option (out–of–bounds) and inability to parse respectively

```
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 6: Demonstrating failure due to insufficient resources

Each input was manually tested to ensure that the above three cases are handled properly.

```
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
```

Figure 7: Demonstration of a army defeat notification

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

1.6 Limitations & Improvements

Limitation: The user interface is very limiting. It is limited in the amount of information it can neatly provide to the player, it is limited in maneuverability etc.

Solution: Using a GUI would help provide a better UX since a GUI is very flexible.

Limitation: The amount of control the player has, is limited. This is mainly due to how clunky the UI is. Giving the user more granular control would require more clutter or more nesting in sub–menus.

Solution: Changing to GUI can allow the application to grow in terms of complexity without sacrificing UX.

2 Minesweeper

2.1 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.

2.2 User Guide

2.2.1 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
```

2.2.2 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 00 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 8: Playing Minesweeper

The general guide to playing Minesweeper is described above in figure 8.

Note: `vim` motions are used to move the cursor.

If the player hits a mine the message "YOU HAVE HIT A MINE! (Press r to retry)" will be displayed.

If the player manages to clear all the cells without hitting a mine the message "YOU HAVE CLEARED THE BOARD! (Press r to retry)" 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.

2.3 Design

Figure 9: Class diagrams of the Minesweeper class and the Board class

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.

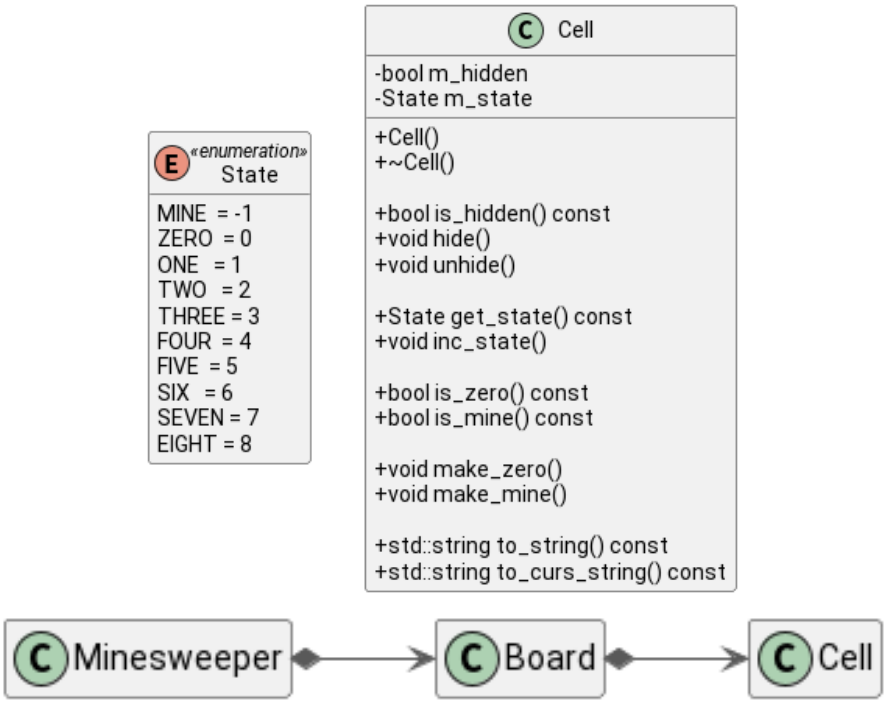

Figure 10: A class diagram of the Cell class State enum and a UML diagram (without members) of the game

Furthermore, the actual board `m_board` is a grid of Cell objects. Also, the lifetime of the objects is managed by the Board as when the board ceases to exist so do the cells.

2.4 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 11: Unit tests for Minesweeper (on macOS Ventura 13.1)

White-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 12: Testing for memory leaks (on macOS Ventura 13.1)

Furthermore, to test for memory leaks, `leaks` was used on macOS and `valgrind` was used on Linux. `leaks` reported no memory leaks whilst `valgrind` reported leaks from `ncurses`.

2.5 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 it does not not macOS.

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`