# Company Defense Documentation

## 1. Personal information

Company Defense, Olli Glorioso, 100098954, Bachelor of Science And Technology (Computer Science), 2022-2023, 19.4.2023

## 2. General description

Company Defense is a popular small game format, tower defense, in which a group of enemies aim for a target, usually through a predetermined route. The player must stop the enemies by building towers next to the path that shoot or in other way defend against the enemy. With time, there will be more and more enemies coming, and the enemies might be more durable and faster than previously. The game ends when there have been too many enemies getting through the map. This project is was planned to be done with the difficulty level of demanding, and I think I achieved that goal.

## 3. User interface

The game is launched by going to the src/main/scala/UI/MainUI.scala-file and running the start-function from there, via the UI of your IDE perhaps. Also, you can just run it with sbt and it should automatically recognize the start-function. Thus, just run `sbt run` in the terminal of your choice.

After launching the project, the user is redirected to the main menu (picture 1). In this simple scene, there are three buttons available:

1. Start new game: with this button, the user can start a new game with the currently set difficulty setting.
2. Continue saved game: with this button, user can continue a game that is saved from last game time. Saved game will continue with towers and health that were in effect during the last finished wave.
3. Settings: open settings menu.

In addition, as in the picture 1 is shown, the “hard” errors are shown in the main menu as error dialogs, if the gameplay is not able to start for some reason. Dialog contains a title, a description of the error and an OK-button to continue. I ended up showing the dialog for crucial errors as I had to be sure that the user will notice the error message.

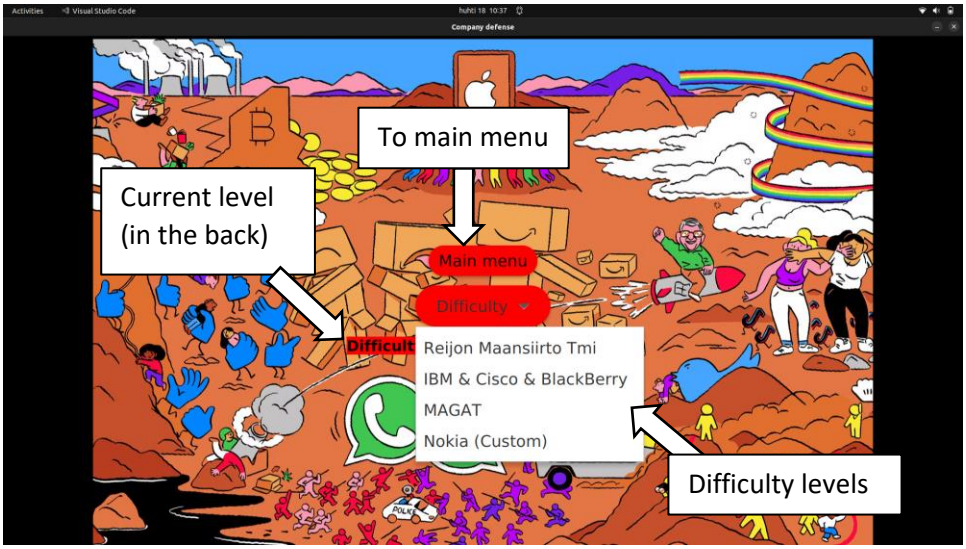
Suppose the user wants to first choose their wanted difficulty level. The user clicks the Settings-button and hops to the settings menu (picture 2). In there, there are two buttons:

1. Main menu: open main menu.
2. Difficulty: a dropdown which allows user to choose the difficulty level. Difficulty levels basically change the default map and wavedata-file, which configures the waves and their enemies (thus, making waves harder or easier depending on the amount of enemies and the ratios of enemy types). There are four levels:
    a. Reijon Maansiirto Tmi: easy difficulty.

b. IBM & Cisco & BlackBerry: medium difficulty.
c. MANGA: hard difficulty.
d. Nokia (Custom): custom difficulty, thus the user specifies the map in Resources/Maps/CUSTOM_MAP.txt and the wave data in Resources/WaveData/CUSTOM_data.txt. More on this later.

After clicking on the wanted difficulty level in the dropdown, the user presses the Main menu –button and returns to main menu. Let's say in this case that the user decides to select the difficulty level "Reijon Maansiirto Tmi". After this, the user presses the Start new game –button, and hops to the gameplay scene (picture 3).


Picture 1: Main menu.




Picture 2: Settings menu.

The gameplay scene is divided into two sections which is quite classical for tower defense games:

1. Sidebar, which contains following features:

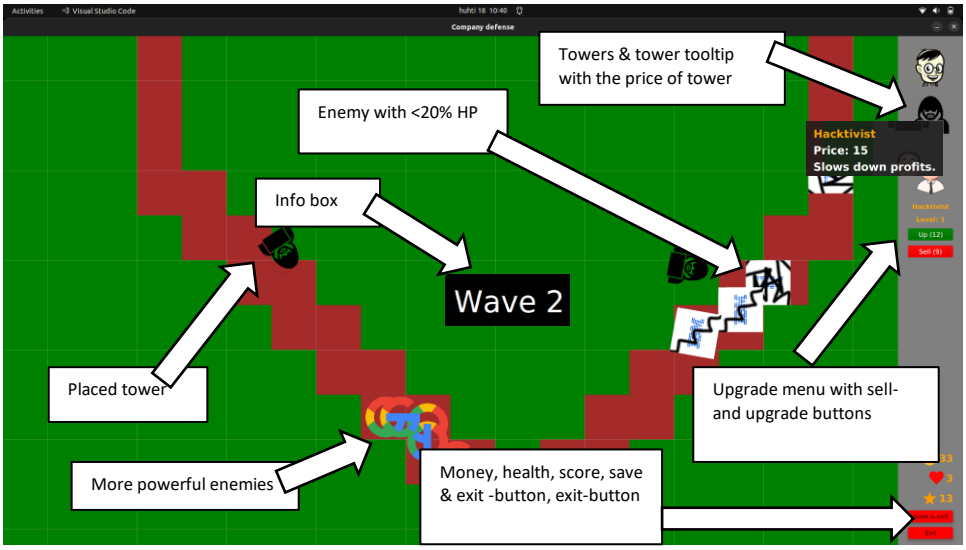
a. Tower buttons – by hovering mouse over these, one can get the description and price of these towers without losing space in the screen for a long time. Then, by dragging them to the map, one can try to place them to it, and money is automatically decreased. While dragging the tower, the range of the tower is shown in the screen with a circle around the tower. The color of the circle depends on the position of the current drag – if it's red, the current position is illegal for a tower to be placed on. If it's green, one can place a tower to the current position.
b. Upgrade menu – this is shown, when an already placed tower is clicked on the map. And also when a tower is freshly placed, in case the user wants to immediatelly upgrade it. After the click, this upgrade menu shows up in the sidebar, just under the tower buttons. It shows the level of the clicked tower, and two buttons:
    i. Up ($cost): with this button, the user can upgrade the features (such as damage, shooting speed) of the clicked tower. This is only possible if the tower has more levels to be upgraded on, otherwise the button is disabled. Max level is five for most of the towers. Money is automatically transferred. The cost of the upgrade will be higher with higher levels of towers.
    ii. Sell ($sellprice): with this button, the user can sell the currently clicked tower with the specified price. This price will increase with the level.
c. Other stuff:
    i. Money, health, and score: they are shown in this order, just on top of the Save & exit –button. The start values of these will vary depending on the difficulty level.
    ii. Save & exit: save current game to a file and exit. In the save, the position of the towers **from last defeated wave** will be saved. Also with the same rule, score, health and money will be saved.
    iii. Exit: exit without saving current gameplay.

2. Map, which has the following elements:
    a. Path tiles, which have the color brown as in the picture 3. The first path tile of the map is determined in the map files, so the enemies will always start from a certain end of the path. Towers cannot be placed on the path tiles.
    b. Background tiles, which have the color green. Towers can be placed onto these tiles.
    c. Enemies, which are basically company logo images. They roll out in "waves", and between these waves is a small break. Enemies don't have health bars, but their logos become increasingly broken when they lose more and more health. In the end, after losing all of their health, they will disappear from the screen. There exist following kinds of enemies:
        i. Basic enemy (IBM): no special abilites, just a basic amount of health and speed.
        ii. Splitting enemy (Google): lots of health, basic speed. Splits into three basic enemies after being defeated.
        iii. Camouflaged enemy (TSMC): regular amount of health and speed. Currently no towers can "see" it, thus deliberately shoot towards it. Bomb tower can destroy them by shooting to nearby enemies (bullets have big range & damage).
        iv. Apple enemy (Apple): even more health than splitting enemy has, slow speed. Splits into three splitting enemies (Google) after being defeated.

d. Towers, which are images of different kinds of tech-related people. Towers are placed on the background tiles. Following towers exist currently:
    i. Regular tower (Nerdinator): just basic tower, regular bullet damage and regular shooting interval.
    ii. Slow down tower (Hacktivist): same features as regular tower, but slows downs enemies until a certain point. More expensive than regular tower.
    iii. Bomb tower: slow shooting interval, but big bullet damage & range.
    iv. Money tower: generates a little amount money every once in a while and rotates every time when money has been generated.

Towers have certain ranges, and they shoot towards enemies in a predetermined speed when the enemies come to their range.

e. Info box: this box shows information about the process of the game, for example when a new wave is beginning and its number
f. Error text: is shown on top of the screen with a red color. Displays an error message if an error or a misuse occurs. For example, if the user is trying to place a tower onto an illegal position, an error text is shown here.

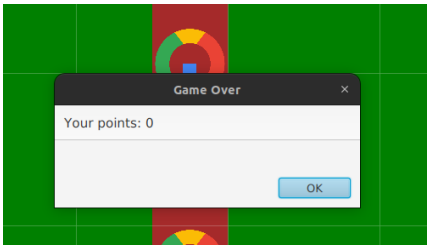



Picture 3: Gameplay scene.

Whit these rules and information, user starts to play the game by placing towers when they have enough money. When towers destroy enemies, a certain amount of money is added to the player's resources. In addition, when the user has defeated a wave, a bigger amount of money is added to the resources. Also score is increased by one when an enemy is destroyed.

Eventually, when the waves become increasingly harded, some of the enemies reach the end of the path. When this happens, the user's health is decreased by one. The amount of health in the beginning of the game depends on the difficulty level. For instance, in the easy mode, the health is ten in the beginning.

When more and more enemies reach the end, the health eventually reaches zero. The player loses, and a game over –dialog will be shown with an OK-button (Picture 4). Also the gathered point amount is shown. After pressing OK, the user is redirected to the main menu.




Picture 4: Game Over -dialog.

An alternative end to the session is the situation, where the user presses Save & exit –button in the gameplay scene. This saves the game data with the previously mentioned rules, and uses is redirected to the main menu. After this, user can, for example, exit the game and do something else. When the user restarts the game software and presses the Continue saved game –button in the main menu, this data is used to initialize the game again, and user will continue from the last wave they were trying to defeat.

# 4. Program structure

When it comes to file structure of the program, the files are divided into four folders (test-folder excluded):

1. Logic: includes all the classes that are related to the main game logic, such as enemy-classes, map-class and tower-classes. Even though, for instance GameObject-class and thus also Tower-class, extend a UI-component, ScalaFX's Rectangle, they are still included in this folder.
2. UI: includes all the classes that are clearly related to rendering the user interface, such as SidebarUI-class and SettingsUI-class. Even though GameplayUI-class has a major part of the game's logic inside it, it's included in this folder, as it does lots of essential rendering as well.
3. Util: includes util classes, constants and global state class of the game. These are separated from the other code to allow clear reuse of functions and constants.
4. Resources: includes all the static resources, such as images, and I/O-related things, such as map- and wavedata-files. Divided into several subfolders accordingly.

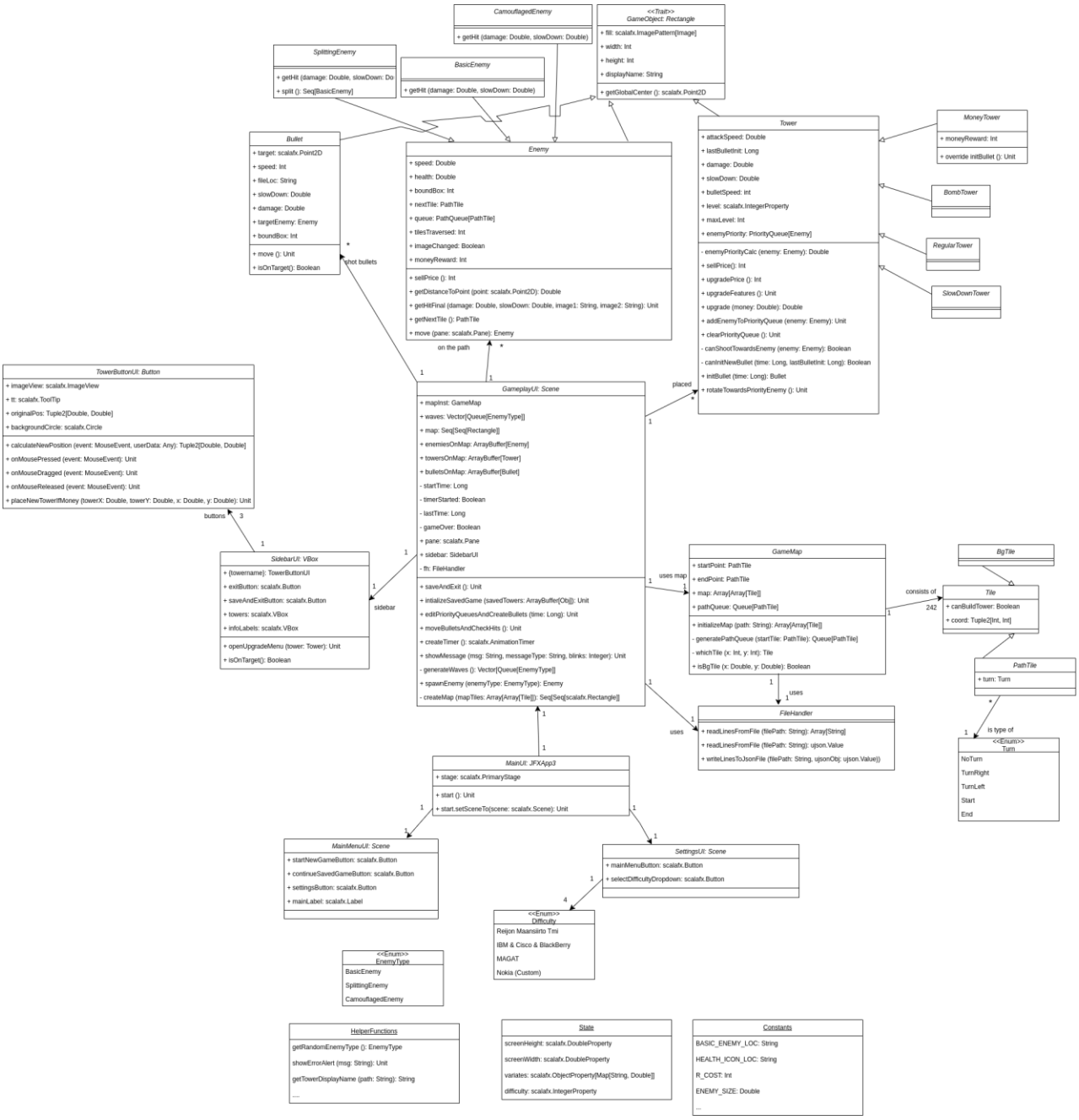
ScalaFX's classes are heavily utilized in the project. The smartest way to implement most of the classes was to extend the ScalaFX-classes, so that it was possible to integrate them to the ScalaFX environment conveniently. See the table below to see which classes just extend ScalaFX classes. Another way would have been to create entirely separate classes for Logic and UI but this would have created lots of unnecessary files and structures. Current solution is a bit of mix: the classes are extensions of UI-components but at the same time they include the required logic.

Here short descriptions of the most essential classes and their methods (see picture 10 for a UML-diagram):

| Class | Description | Functions and class variables |
|---|---|---|
| **MainUI** | Highest-order class, starts the game and initializes | Start() = start the program, initialize stage |

| | | |
|---|---|---|
| | the main menu & other scenes. ScalaFX's JFXApp3. | |
| **MainMenuUI** | Scene for main menu and its buttons & elements. ScalaFX's Scene. | StartNewGameButton = button to start completely new game with current settings.<br>SettingsButton = button to move to the settings scene. |
| **SettingsUI** | Scene for settings scene and its buttons & elements. ScalaFX's Scene. | SetDifficultyDropdown = dropdown which allows to set the difficulty state of the game. |
| **GameplayUI** | Handles the main gameplay. ScalaFX's Scene. | MapInst = created Map-instance saved in this.<br>Waves = saves wave-vector created with generateWaves().<br>Map = saves 2D map sequence for rendering, created with createMap().<br>{something}onMap = saves the currently displayed game objects.<br>Sidebar = saves created SidebarUI-instance.<br>SaveAndExit() = save the gameplay to files and exit the game.<br>InitializeSavedGame (savedTowers) = initialize saved game based on given array that was retrieved previously from a file.<br>EditPriorityQueuesAndCreateBullets (time) = run on every relevant clock tick, edits the priority queues of the towers and initializes bullets if possible.<br>MoveBulletsAndCheckHits () = run on every relevant clock tick, moves bullets and check if they hit enemies -> make damage.<br>CreateTimer () = initializes the internal clock of the program.<br>ShowMessage (msg, messageType, blinks) = shows the gameplay messages (blinks of texts), such as error messages.<br>SpawnEnemy (enemyType) = spawns a given enemy type, is run in certain periods in the clock. |
| **Enemy** | Represents the enemies in the game, has several subclasses for different kind of enemies. ScalaFX's Rectangle. | NextTile = the tile the enemy is going to enter next.<br>Queue = the queue of path tiles that enemy is going to traverse through.<br>GetDistanceToPoint (point) = get distance from enemy to the desired point.<br>Move (pane) = move for the amount of the speed of the enemy. |
| **Bullet** | Represents the bullets of towers in the game. ScalaFX's Rectangle. | Target = target of the bullet. The bullet will be removed in this point.<br>Move () = move for the amount of the speed of the bullet.<br>IsOnTarget () = checks if the bullet is on target and returns a boolean value accordingly. |

| | | |
|---|---|---|
| **GameObject** | Represents all the moving/rotating objects (towers, bullets, enemies), and they extend this class. ScalaFX's Rectangle-class. Includes the image of the object. | GetGlobalCenter () = gets the absolute, global center coordinate of the object, related to the map. |
| **Tower** | Represents a tower in the game. Has several subclasses for different kind of towers. ScalaFX's Rectangle. | EnemyPriority = saves the queue which has the enemy priorities (a priority of enemy depends on different factors, such as its distance to the end of the path and its health). The tower shoots and rotates towards to the first enemy in the queue.<br>EnemyPriorityCalc (enemy) = determines the equation for a certain enemy's priority.<br>AddEnemyToPriorityQueue (enemy) = adds an enemy to the previously described priority queue.<br>CanShootTowardsEnemy (enemy) = determines whether the tower can shoot towards the given enemy. Usually depends on the range of the tower and the distance of the enemy regarding to the tower.<br>InitBullet (time) = shoot. |
| **GameMap** | Logical representation of a map in the game. Is used to fetch, save and work with map files. | InitializeMap (path) = create a representation of a map based on a given file.<br>GeneratePathQueue = generate a queue of path tiles which enemies are supposed to move through.<br>WhichTile = gives the type of the tile in a given coordinate. |
| **Tile** | Superclass for different kind of map tiles. | |
| **FileHandler** | Util class for I/O. | |
| **HelperFunctions** | Util functions. | |
| **State** | Global state of the game. | |
| **Constants** | Some constants for reuse. | |




Picture 10: UML-diagram of the project class structure.

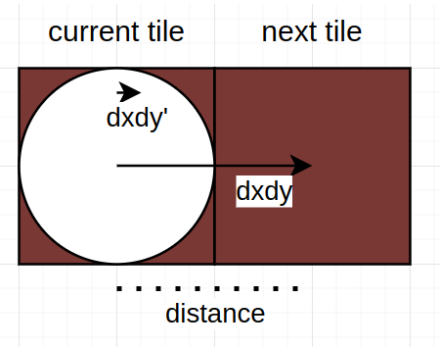
As seen in UML diagram, classes HelperFunctions, State, Constants and EnemyType are not particularly related to any other classes. The first three are utilized in many classes but they are just used just for their functions and values, the classes themselves are not used for any particular reason. That's why they are not connected to other classes in the UML diagram, and also because otherwise we would have a messy diagram with too many arrows. EnemyType either isn't saved in any instance variables.

# 5. Algorithms

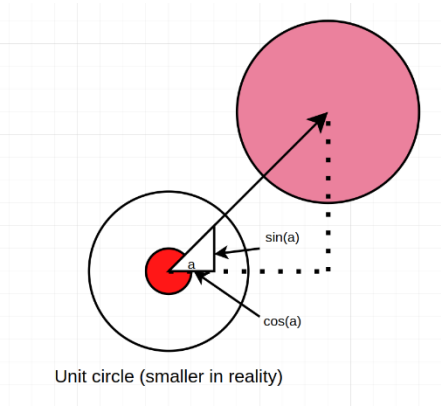
Algorithms of the game are briefly described in the table below.

| Title | Description |
|---|---|
| **Enemy movement.** | 1. Enemy has a pathQueue as a parameter. It's a previously created queue of path tiles in the order they are in the map.<br>2. The algorithm (see picture 5 also):<br>a. Save current coordinates of the enemy, current tile where the enemy is, and the next time from the queue.<br>b. Create a 2D vector from current position to the next tile's position, let's call it dydx.<br>c. Calculate the length of dydx vector by taking square root of the squares of its components, $\sqrt{x^2 + y^2}$.<br>d. Then, if length > 0, create new 2D vector, which is dxdy's unit vector. Now we have a vector that has the correct direction and a length of one. Let's call it dxdy'. If the length <= 0, make this same vector a zero vector (no direction nor length).<br>e. Multiply the dxdy' by the speed of the enemy, to add different speeds.<br>f. If the length of dxdy > speed, add the dxdy's x-component to enemy's x-position on the map, and same for y-component and y-position. If length of dxdy <= speed and there is still next tile available (enemy has reached the next tile), fetch the next tile from pathQueue and return enemy's current position.<br>g. If length of dxdy was > speed, calculate the angle of the enemy with the following equation: $\tan(\alpha) = \frac{x}{y}$ and set enemy's rotation to it (adjusted). |
| **Distance between a tower and an enemy.** | 1. This is done with the distance-method of the ScalaFX's Point2D-class. But under the hood, it's quite simple, just uses the following equation: $\sqrt{(x_{tower} - x_{enemy})^2 + (y_{tower} - y_{enemy})^2}$. |
| **Initialize map from a file.** | 1. Initialize an Array[Array[Tile]]-2D-array (let's call it map), and fetch the characters in each line so that we get 2D-array of lines and their characters (let's call it lines).<br>2. The algorithm (iterating through the lines-array):<br>a. If the character is zero -> create a new background tile to the same position in the map.<br>b. If the character is one -> first check the characters on the top, down, left and right sides of the current character, and based on that either create a path tile with the correct turn.<br>c. If the character is two -> create a path tile with the type 'Start'.<br>d. If the character is three -> create a path tile with the type 'End'. |
| **Generate a path queue for the enemies to use.** | 1. Start from the start path tile, determined already when the map has been created.<br>2. Get the current tile's coordinates.<br>3. Go through all the possible positions of the next path tile: up, down, left and right (in the code all the possible directions are searched, |

| | |
|---|---|
| | this could be changed as the next tile can only be in the abovementioned positions regarding to the current tile).<br>4. In every one of the possible positions, get the prospective tile from the map class variable.<br>5. If the tile is an instance of path tile, add the tile to the path queue.<br>6. Move the iteration to the next tile that was gotten from the map clas variable. If it's turn class variable is 'End', end the iteration. Otherwise continue in the same way. |
| **Shooting of the enemies.** | 1. First phase is to create a new bullet to the center of the tower.<br>a. First, check if it is even possible to shoot an enemy. This is done by taking the first enemy in the priority queue of a tower, and checking if the distance to the enemy is less than the range of the tower.<br>b. If shooting towards an enemy is possible, create a new bullet to the center of the tower, and save the target enemy to the bullet-class.<br>c. Save the bullet initilization time. Next time a bullet can be shot, if the last bullet initialization happened a certain time ago.<br>2. Next phase is to move the bullets towards the destination until they hit the target.<br>a. With each tick of the game timer, go through all the bullets and move them:<br>b. First, calculate the distance to the target coordinates.<br>c. If the distance > speed of the bullet, calculate the angle between the bullet current coordinates and the coordinates of the target, with respect to the positive x-axis, with the Java's math.atan2-method, which converts the specified rectangular coordinates into polar coordinates.<br>d. Then, we take the cosine of the angle (thus we get the x-component of the angle in an unit circle) and move the bullet's x-position to according to this x-component multiplied by speed.<br>e. Same for y-values.<br>f. If distance <= speed, bullet is on target and is removed.<br>3. With each tick, we also go through all the enemies, measure the distance between the enemies and the bullets. If the distance is less than the boundbox radius of the enemy/bullet, the enemy takes damage and some things happen accordingly (for example, if enemy has lost all of its health, remove it). |
| **Tower's priority queue.** | Tower's priority queue is just a queue of enemies, and the elements in the queue are ordered based on a certain rule. The rule is (the enemy with the highest value is the first element in the priority queue):<br>Enemy's health * 0.4 + enemy's speed * 0.01 - distance between tower and enemy * 0.25 (negative, because the bigger the distance the lower the priority) and if distance to enemy is less than range of tower, + enemy's traversed tile amount * 100. |




Picture 5: Visualization of the components in the enemy movement –algorithm.




Picture 6: Visualization of the angle between the enemy and the bullet, with respect to the positive x-axis.

# 6. Data structures

Below some of the most essential data structures of the program explained:

| Data structure title | Type | Mutable | Description |
|---|---|---|---|
| **pathQueue** | Scala.collection.immutable.Queue | No | PathQueue is a queue that includes all the path tiles in an order in which they are supposed to be traversed. Queue was chosen to be the data structure, because queue is the most natural decision for traversing through collections with only the top element needed every time, and after which it is not needed anymore. Immutability was chosen, because there is no particular reason to mutate original queues. |
| **enemyPriority** | Scala.collection.mutable.PriorityQueue | Yes | Scala's PriorityQueue-collection is good for this cause, because it allows us to define the way of ordering the queue. Thus, at every tick, we create a new priority queue for every tower in the map, |

| | | | |
|---|---|---|---|
| | | | and just add every enemy into it, and the built-in data structure takes care of ordering the queue based on the enemy features and the way we want. See Algorithms-part of this documentation as well. Priority queues are mutable. |
| **waves** | Scala.collection.immutable.Vector[Scala.collection.mutable.Queue]] | No | When it comes to waves, we just create them in the beginning and then traverse through them, so we don't mutate them. This is why an immutable collection was the right choice. In addition, because we save the wave number in a game state (for saving games and UI for example), it's also beneficial to be able access certain indexes within this collection. Vectors allow these both features, hence was the best possible option. |
| **map** | Array[Array[Tile]] | Yes | With the map we use arrays. Because the map's size is always a constant, and this data structure stores **fixed-size** sequential collections, it's a good option to use. Also, it gives us an easy and fast access to its indices. |
| **{something}OnMap** | Scala.collection.mutable.ArrayBuffer | Yes | TowersOnMap, bulletsOnMap and enemiesOnMap are all built with this same data structure, array buffer. In these variables, we have to store the enemies, bullets and towers that are currently drawn on the map. We need to iterate through all of these arrays, and mutate the elements in them (for example, positions or health of enemies), and thus mutable array buffer is the best option for this use case. |
| **variates** | ObjectProperty[Map[String, Double]] | Yes | Because we have only four variables that are shown in the screen (money, health, waveNo, score), in terms of efficiency, it doesn't really matter which data structure we use. We use Maps, because with it it's easy and clear to get values from the structure. In addition, it make code clearer to use one data structure to store all these four values. This is wrapped in ScalaFX's ObjectProperty-class, which allows us to easily see the changes of values in this Map. |

# 7. Files and internet access

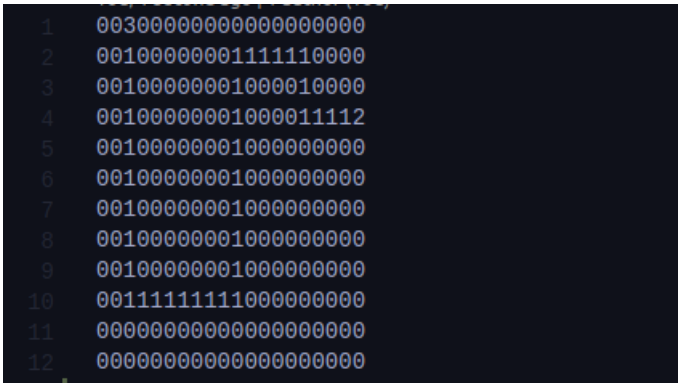
Below described the files of the program. All the files are located inside Resources-folder. All the file names have to be as they are, so please do not change the them. Bullet-, Enemy-, Other- and Tower-folders have static files. On ther other hand, WaveData-, Saved- and Maps-folders save data.

| Location | Files | Description |
|---|---|---|
| **Bullets** | {bullet_type}Bullet.png | Images of bullets. |
| **Enemies** | {enemy_type}Enemy.png | Images of enemies. |
| **Towers** | {tower_type}Tower.png | Images of towers. |
| **Other** | {otherf_type}.png | Images of some other resources, such as health icon or money icon. |
| **Saved** | LATEST_saved.json | Includes LATEST_saved.json-file, which includes the saved data from last save. It's in JSON-format, and has the following keys and values (description in the comments):<br>health: Int // Health<br>money: Int // Money<br>score: Int // Score<br>towers:<br>type: 'S' \| 'R' \| 'B' \| ´M´ // S stands for SlowDown, R for Regular, B for Bomb, M for Money<br>globalX: Int // Global x coordinate of its center<br>globalY: Int // Global y coordinate of its center<br>level: Int // Tower level<br>See the picture 7 for an example. |
| **Maps** | CUSTOM_map.txt.<br>EASY_map.txt<br>HARD_map.txt<br>MEDIUM_map.txt<br>{test_map_name}.txt | This folder contains all the maps of the game. EASY_map.txt is a ready-made map for easy difficulty level, and same goes for HARD_map.txt and MEDIUM_map.txt. CUSTOM_map.txt is meant to be customized by the user, and it will be used when the Nokia (Custom) difficulty is set.<br><br>Guide for customization:<br>- Please do not change the amount of rows and columns (12x20) in the map, these have to stay the same.<br>- Also, the map is a 12x20 quadrangle consisting of zeros, ones, one two and one three. Zeros are background tiles (towers can be placed on these), ones are path tiles where the enemies are supposed to move. Two is the starting point and three is the ending point.<br>- Mark 2 to the coordinate you want the path to start. Mark 1's to determine the path, and in the end, mark 3 (this has to be in the beginning/end of a column or row).<br>See example in picture 8. |

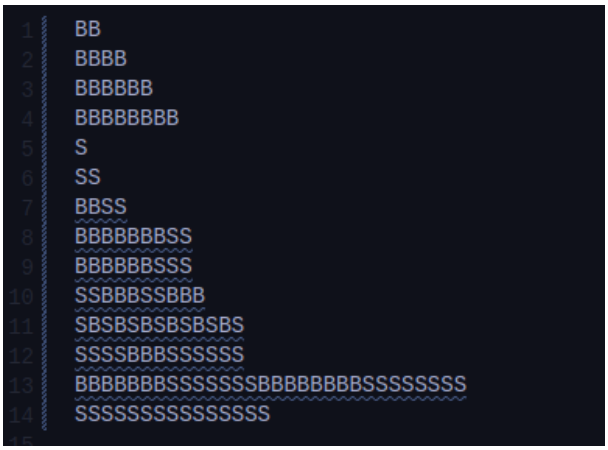
| | | |
|---|---|---|
| **WaveData** | CUSTOM_data.txt<br>HARD_data.txt<br>MEDIUM_data.txt<br>EASY_data.txt<br>{test_wave_data_name}.txt | These files include the enemy wave data for different levels, as in the map-files as well: HARD_data.txt for hard difficulty level, EASY_data.txt for easy and MEDIUM_data.txt for medium. Custom data is used when Nokia (Custom) difficulty is set. Thus CUSTOM_data.txt is meant to be customized by the user.<br><br>Guide for customization:<br>- Every line in a wavedata-file represents one wave. And every character in one line represents one enemy. For example.<br>- Characters represent enemies with the following rules:<br>a. S = Splitting enemy (Google)<br>b. C = Camouflaged enemy (TSMC)<br>c. B = Basic enemy (IBM)<br>So, for example, the following text inside file:<br>SSS<br>BBS<br>Would represent two waves, three splitting enemies in the first, and two basic enemies and one splitting enemy in the second wave. See picture 9 for an example. |
| | | |


```
{
    "health": 2,
    "money": 49,
    "difficulty": 2,
    "waveNo": 1,
    "score": 14,
    "towers":
        [
            {
                "type": "S",
                "globalX": 592,
                "globalY": 415,
                "level": 1
            },
            {
                "type": "S",
                "globalX": 686,
                "globalY": 516,
                "level": 1
            }
        ]
}
```


Picture 7: Saved/LATEST_saved.json example data.




Picture 8: Maps/CUSTOM_map.txt example map.




Picture 9: WaveData/CUSTOM_data.txt example wave data.

For testing purposes: one does not have to make any edits to the files. If one would like to make some edits and try different kinds of maps, please edit the Maps/CUSTOM_map.txt for maps and WaveData/CUSTOM_data.txt for wave data, and choose Nokia (Custom) as a difficulty.

# 8. Testing

The tests of the game are separated into two separate test entities (also files): GUITests and LogicTests (Unit tests). LogicTests has basically tested all the logic that doesn't require any ScalaFX logic or classes, thus it tests the Map and FileHandler-classes. In the beginning of logic tests, instances of both classes are initialized. The reason for this kind of separation of tests is the fact that in order to test any entities that use or extend ScalaFX classes, a GUI must be started.

GUITests then has more tests than LogicTests as ScalaFX classes are used in many parts of the program code. This test class first creates the GUI and sets the main menu scene as the current scene, also in the beginning a BasicEnemy-instance and a Map-instance are created. In addition, a class specifically for testing is created: TowerExtended. This just extends Tower-class and adds a private method that is part of the Tower. This is because we are also going to test this private method in the tests. TowerExtended-instances are created within tests because in many tests we mutate them, and in many tests non-mutated instances are needed. Then, the GUI is started and the tests themselves are run. Tested are, for instance, Tower-class, Enemy-class, Bullet-class, the most important methods of HelperFunctions-class.

The testing process was a bit different to the original testing plan, as I had to do a big portion of the tests during the end of the project. This is because I first didn't get the tests using GUI to work and I wanted to be sure first that I am able to finish the game itself on time.

Most of the classes are tested well. What is missing are the tests that check the validity of GUI responses to user input. Had I more time, would I focus on creating these tests.

# 9. Known bugs and missing features

The program is not perfectly balanced when it comes to the difficulty levels and the experience in general. For example, a wave might be too easy for a certain difficulty level sometimes, or too much money is given for destroying a certain enemy. It's trivial but time-consuming to balance the game and that's why it was not the priority when developing the game.

Another trivial thing that the game lacks is more enemy and tower types. As it is trivial to add simple enemies or towers (just adjust their properties, behavior and properties), I did not want to use too much time for it as it does not show my skills anymore. Instead, I used some more time to create essentially different types of enemies (for example, splitting enemy), not just new enemies with different kinds of adjusted properties.

I am not aware of major bugs in the game.

# 10. 3 best sides and 3 weaknesses

One of the best sides of my game is probably customization and automatization options. One can choose between three difficulty levels and even choose the custom difficulty. Maps are straightforward to create, also the waves are very easy to create and use in the custom mode. The program automatizes the rest – creates the map and spawns the new enemies seamlessly. And if the program runs out of data from wavedata-file, it just starts to spawn random enemies with an increasing difficulty.

User interface is quite easy to use and straightforward as well. Towers are easy to just drag to the map and the money is automatically decreased – no need to press buttons. Also, the tooltip saves lots of room, because they can be shown on demand by hovering mouse over the buttons.

Error handling is another good side in my game. Breaking errors, such as an illegal map file, don't break the application – they show a very informative error dialogs in the screen. Not only the user's experience with the error handling is good, but also the developer's - it's easy to throw errors and the error dialogs are given with functions. Also, the gameplay error messages are easy to add for developer just by using a dedicated function.

One weakness is the movement of enemies regarding to each other. Currently, if they are in different speeds, they can stack on top of each other and that probably doesn't so clean. I would fix this according to my original plan – use some group simulation. Thus, enemies that are faster than the ones in front of them, would wait behind the slower enemies.

Another weakness is probably testing, as the tests could be more comprehensive and there should be more UI tests. I had some issues with getting the UI input recognized in the tests, which decreased the number of tests that I was able to create. This affects especially tests that require some input from the

tests, such as clicking buttons or dragging towers to the map. The methods and features of classes without inputs are well covered on the other hand.

# 11. Deviations from the plan, realized process and schedule

Project was implemented in the following cycles:

| Week | Time used (h) | Description | Focused on classes |
|---|---|---|---|
| **2** | 15 | During these two weeks, project was initialized and formatters & linters installed. Project's file structure was initialized with folders for Logic, UI and Util. Some base files for future classes were created.<br><br>Then, I started with MainUI-file and just tried to get the UI to start and run. This took quite a lot of time, since I ran into many problems and error messages. After getting the UI to work, I initialized the main menu and game play view without map. I also finished with the logic of Map-class, also creating maps based on files.<br><br>Within these two weeks, there were no deviations from plan. I used time to every aspect of the game I was planning (see technical plan). | MainUI, GameplayUI, Map |
| **4** | 25 | Some tests (especially for Map-class) were initialized and made.<br><br>Movement and spawning of enemies was finished, though some bugs still existed. Especially the algorithms for enemy movement were quite consuming, their animations not that much.<br><br>Wave creation based on a file was created and finished. Now we were able to specify waves in a file and they were automatically used by GameplayUI. No deviations from plan. | GameplayUI, Enemy |
| **6** | 5 | Research on dragging towers to the map, no results yet. Some bugs fixed and error handling. Tooltip for tower buttons. Big devations from plan as I didn't use no where near the time I planned for these two weeks. I also heavily concentrated on the UI rather than logic, in the plan I should've done both UI and logic. | SidebarUI, TowerButtonUI |
| **8** | 47 | Tower dragging, rotating, enemy priorities and shooting implemented. Bullets implemented, new enemy and tower type implemented, priority queue. Enemy deaths. Health, money and score implemented and automatically increased / decreased on certain occasions. Upgrade and sell features of towers implemented. Deviations from the plan only when it comes to the used time. | GameplayUI, Bullet, Tower, TowerButtonUI |
| **9** | 23 | Tests, bug fixes, refactoring and final testing. Documentation. No deviations from the plan, though in the plan I didn't take the time used for documentation into consideration. | Everything |

I think the total time estimate was about ten hours too much, but about correct still. I deviated heavily from the weekly hours plan because I concentrated much of the work to the last weeks before deadline and to the beginning of the project. In the middle of the project, I didn't use as much time as planned initially due to workload on other courses.

During the progress I learnt that the planning is an essential part of a software project. In my previous projects I have not really done any planning, just documentation to the existing projects. I found it very convenient to just check some things from the plan and implement them. For example, compiling the class structure was very easy with the plan. Naturally, I learnt many neat coding tricks with Scala and even though I initially didn't utilize ScalaFX's features and data structures (Point2D, ObjectProperties), I ended up with using many of those and found them great.

# 12. Final evaluation

In overall, I think the game is a very well-working basic tower defense game. The core mechanics work seamlessly, and the UI is easy to use. There are no major bugs, and everything just works without problems. Maybe that's the best side of the game: it perfoms well in every subarea, from good UI to good logic. Other good aspects worth mentioning would be the modularity of my code – everything is well organized in the program code and separate entities are in separate files or folders.

There are still many things to improve in the program. First, I would improve the enemy movement regarding to each other. As mentioned, using group simulation with the movement would make the enemy movement way better, so that they don't stack on top of each other. Another good step would be optimizing the data structures even more to make the game less buggy with lots of enemies and bullets on the map, because this issue emerges also when there are lots of game objects on the screen. I was especially thinking to get rid of arrays enemiesOnMap, towersOnMap and bulletsOnMap. It's quite time consuming to iterate through these arrays. I would replace them by just iterating through GameplayUI's children's nodes, obviously skipping the map tiles and other unnecessary nodes.

Also creating new types of enemies and towers would be a good idea, especially towers that can counter the camouflaged enemy, for example the tower that shoots in the direction of the current mouse position in the screen. After these updates, only minor enhancements would be required. Thankfully the program structure is quite modular, so it's pretty easy to make extensions to the game. Although tower and enemy types could be added more easily creating an array of all the types and then in every place iterate through these arrays. Currently, if a tower type is added, it has to be added manually everywhere to the code.

If I started the project again from the beginning, I would immediatelly create all the classes by extending existing ScalaFX classes and not really create any classes from scratch. For example, the Map-class was a bit unnecessary, I could have just created all the tiles based from the file and add them to GameplayUI's root plane, instead of creating a 2D array first in the Map-class and then iterating through it again. Also I would utilize more the Integer-, Double-, ObjectProperties etc. with all the UI-related variables.

# 13. References

https://www.scala-lang.org/api/2.12.6/scala/collection/mutable/PriorityQueue.html

https://www.tutorialspoint.com/scala/scala_arrays.htm

https://github.com/TestFX/TestFX

https://www.scalafx.org/

https://www.wartremover.org/

https://openjfx.io/

https://www.scalatest.org/

https://scalameta.org/scalafmt/

https://com-lihaoyi.github.io/upickle/

https://www.baeldung.com/scala/vector-benefits

https://github.com/scalafx/scalafx

https://scalameta.org/metals/

# 14. Appendixes

Source code:

https://version.aalto.fi/gitlab/glorioo1/company-defense

Program window when it is launched: