Miro Keimiöniemi (100787595)

Data Science, BSc, 04.05.2023

Strategy Game

Programming Studio A course project

Final Report

General description

The untitled strategy game is a probabilistic battle simulator, aiming to capture the uncertainty of traversing and engaging in combat on unknown territory against unknown enemies. The game mode is conquest, where two players compete for dominance over the objective tiles (dark grey dotted ones in the middle of the GUI picture) until either one has controlled them for a total of 100 turns with various types of units such as foot soldiers, tanks and snipers with different attributes such as damage, weight, size, range and health. These attributes determine the probabilities of a given unit being able to enter a given terrain and triumphing over another in battle.

To give the player some agency and a chance to show and develop their strategic competency as the commander of their troops, at the very core of the game is the idea of conditional actions consisting of primary and secondary actions and their respective targets. Each action target has an associated probability calculated based on the type of action and attributes of both the agent and the target. (These will be discussed in more detail in the algorithms section) The player may select any of their battle units by left clicking it with their mouse and choose a primary action for it from the dropdown menu on the right GUI pane, that gets highlighted in grey on the grid as shown in the picture below. This instantly prompts the player to choose a secondary action as well, which is highlighted in a light brown shade.

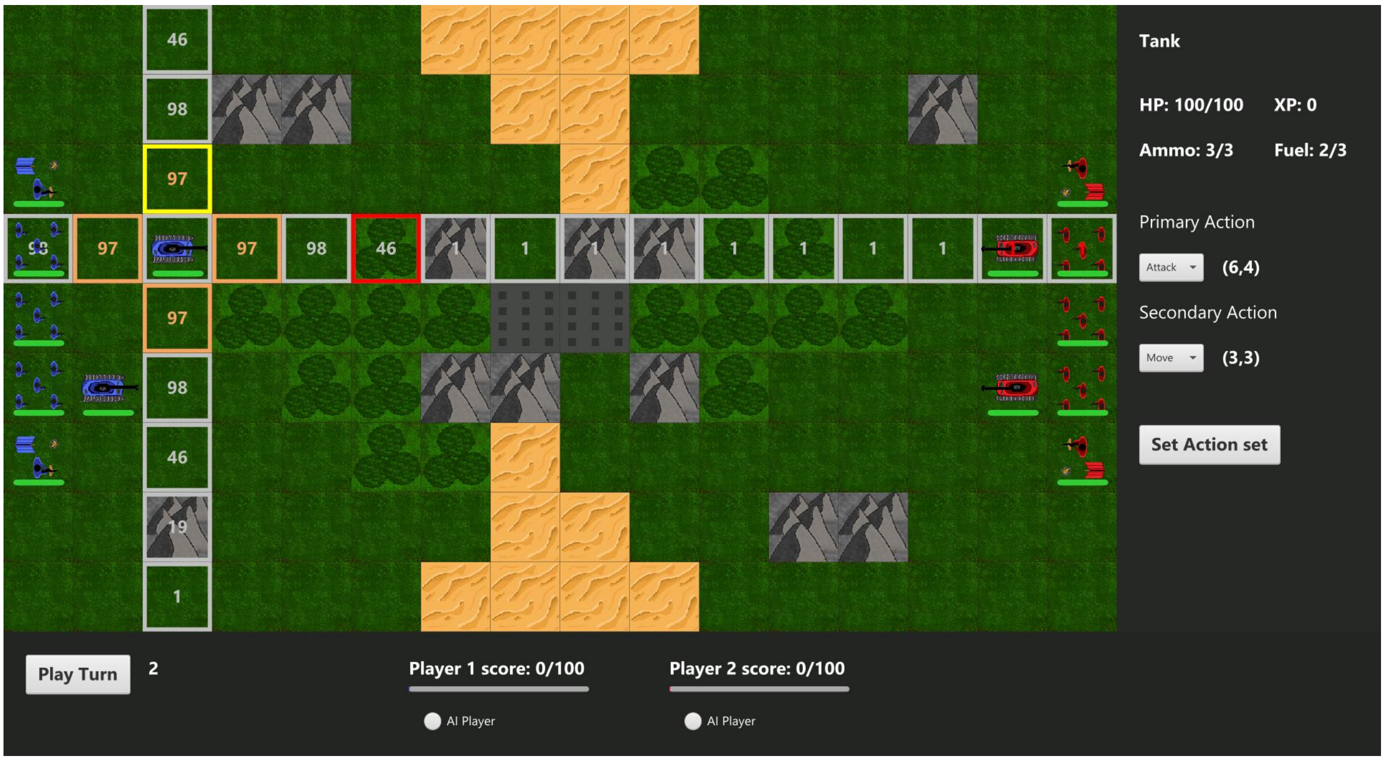
Both actions are by default “Move” but may be mixed and matched in any combination. Target selection for the primary action is highlighted in red and secondary target selection is highlighted in yellow. Once both actions and targets (coordinates next to selected actions) are selected, the “Set Action set” button can be clicked to save the action set for the selected battle unit. Once action sets are set for all desired units, “Play Turn” calculates the outcomes and updates the game state accordingly. Only if the primary action fails is the secondary action even attempted. It is entirely possible that both actions fail, whereas only one can ever succeed at once. Every unit can be assigned an action set during a turn so that with the default game launch configuration at most seven action sets will be excited at once.

Attacking an enemy battle unit always launches a duel where, based on the unit types, their experience and damage gradients (distance between), the loser suffers all the damage, whereas the winner stays unscathed. Attacking a terrain tile on the other hand degrades it so that its attributes change, making it easier or harder to traverse. If sufficient explosive damage is dealt to, for example, a rock, it will turn into gravel. The environment is therefore destructible. However, only tanks can degrade rocks.

Moving simply places the battle unit in the new coordinates within its range. However, if these contain another battle unit, should the move be successful, the moving unit will ram them and be placed on top of the now destroyed enemy unit.

Defend activates a temporary effect that reduces the damage taken by the defending battle unit to half in case it loses a battle while defending. The effect will persist until it moves or attacks something proactively.

The other two actions are Rest and Reload, replacing the supply chain mechanic, which was foregone due to lack of time and potentially resulting in overcrowding of the GUI. Rest restores a battle unit's fuel to its maximum while staying still for one round and Reload restores its ammo to its maximum such that there is an aspect of resource management as well.



User Interface

The user interface consists of three sections: a GridPane starting from the top left corner displaying the game map and battle units, a VBox on the right containing all battle unit related information and controls and an HBox on the bottom containing the game state information and controls.

As already mentioned above, selected actions are highlighted in the game map upon unit selection. First, only the primary action in light grey and after selecting its target, highlighted in red, the range of the secondary action is highlighted in light brown as well and its target is highlighted in yellow upon selection.

In the battle unit pane on the right, there are five labels on the top displaying the selected unit's type, health, experience, ammo and fuel in that order. Ammo or fuel running out is highlighted in red. Primary and secondary actions are selected using the dropdowns with the second one becoming accessible only after primary action has been set. Their target coordinates are shown next to them and “Set action set” saves the current selection to the battle unit such that it can be overwritten any time by just selecting new actions and pressing it again. Changing between the action currently being selected can be done by clicking on the dropdowns with or without selecting a new action or by clicking the battle unit twice to reset the selections.

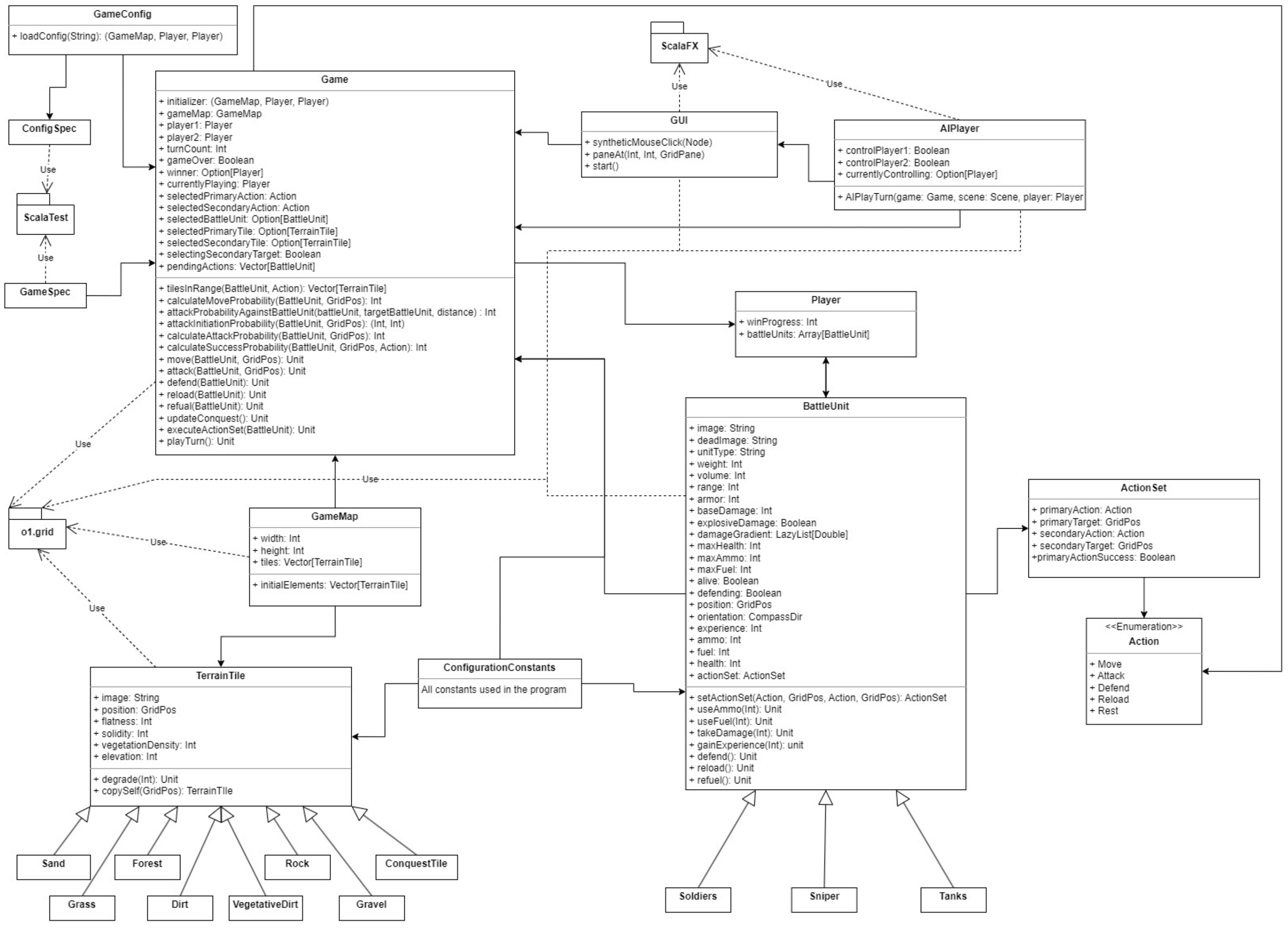
In the bottom pane, there is a “Play Turn” button that, upon clicking, processes all the set action sets and updates the GUI and game state. Next to it is a turn counter followed by both player’s current scores and their visualizations in the forms of bars filling up with the same color as their battle units. Below them are radio buttons whose states signal to the “AI” Player whether it is expected to move that player’s units. This means that the player may choose which units they want to control or if they just want to observe the AI Player play against itself. This also makes intervening possible at any point by simply unchecking one of them and continuing the game as normal.

The GUI is dynamically scalable when the map is sufficiently wider than it is tall. This is due to a very simplistic and quick implementation to mostly just scale up to larger monitors without considering scaling down. However, one visual bug that I did not have time to fix remains, where the health bar margins only get updated after pressing “Play Turn” instead of immediately upon rescaling. This is not much of an issue scaling up because they are just a bit off-center and will get fixed rather quickly but majorly scaling down, the too large lingering margins can cause the neighboring tiles to be pushed away due to overflow. This is also fixed within one turn but is significantly more disturbing than the other way around.

Program structure

In the end, I ended up implementing key mechanics 1 to 5 from the list of 8 mentioned in the general plan, landing me comfortably in the interesting and potentially fun game category and still landing in the course grade 5 range according to my assessment when writing the technical plan. However, most cut features have justifications different to simply the lack of time. Sure, this did not enable experimenting with them in the first place and trying to make them work but mainly, a random generated map was always just a bonus, simultaneous turns would be very difficult to execute so that it is completely fair to both players in all cases and supply chains turned out be cluttering and away from the main focus of the game after all. The lack of a zoom feature and therefore smaller map sizes were also deciding factors and thus, overall I am quite happy with what was actually implemented in the end.

Below is the final UML illustrating the final structure of the program.


GameConfig
+ loadConfig(String): (GameMap, Player, Player)
ConfigSpec
Use
ScalaTest
Use
GameSpec
Game
+ initializer: (GameMap, Player, Player)
+ gameMap: GameMap
+ player1: Player
+ player2: Player
+ turnCount: Int
+ gameOver: Boolean
+ winner: Option[Player]
+ currentlyPlaying: Player
+ selectedPrimaryAction: Action
+ selectedSecondaryAction: Action
+ selectedBattleUnit: Option[BattleUnit]
+ selectedPrimaryTile: Option[TerrainTile]
+ selectedSecondaryTile: Option[TerrainTile]
+ selectingSecondaryTarget: Boolean
+ pendingActions: Vector[BattleUnit]
+ tilesInRange(BattleUnit, Action): Vector[TerrainTile]
+ calculateMoveProbability(BattleUnit, GridPos): Int
+ attackProbabilityAgainstBattleUnit(battleUnit, targetBattleUnit, distance) : Int
+ attackInitiationProbability(BattleUnit, GridPos): (Int, Int)
+ calculateAttackProbability(BattleUnit, GridPos): Int
+ calculateSuccessProbability(BattleUnit, GridPos, Action): Int
+ move(BattleUnit, GridPos): Unit
+ attack(BattleUnit, GridPos): Unit
+ defend(BattleUnit): Unit
+ reload(BattleUnit): Unit
+ refual(BattleUnit): Unit
+ updateConquest(): Unit
+ executeActionSet(BattleUnit): Unit
+ playTurn(): Unit
ScalaFX
Use
Use
GUI
+ syntheticMouseClick(Node)
+ paneAt(Int, Int, GridPane)
+ start()
AIPlayer
+ controlPlayer1: Boolean
+ controlPlayer2: Boolean
+ currentlyControlling: Option[Player]
+ AIPlayTurn(game: Game, scene: Scene, player: Player
Player
+ winProgress: Int
+ battleUnits: Array[BattleUnit]
BattleUnit
+ image: String
+ deadImage: String
+ unitType: String
+ weight: Int
+ volume: Int
+ range: Int
+ armor: Int
+ baseDamage: Int
+ explosiveDamage: Boolean
+ damageGradient: LazyList[Double]
+ maxHealth: Int
+ maxAmmo: Int
+ maxFuel: Int
+ alive: Boolean
+ defending: Boolean
+ position: GridPos
+ orientation: CompassDir
+ experience: Int
+ ammo: Int
+ fuel: Int
+ health: Int
+ actionSet: ActionSet
+ setActionSet(Action, GridPos, Action, GridPos): ActionSet
+ useAmmo(Int): Unit
+ useFuel(Int): Unit
+ takeDamage(Int): Unit
+ gainExperience(Int): unit
+ defend(): Unit
+ reload(): Unit
+ refuel(): Unit
ActionSet
+ primaryAction: Action
+ primaryTarget: GridPos
+ secondaryAction: Action
+ secondaryTarget: GridPos
+primaryActionSuccess: Boolean
<<Enumeration>>
Action
+ Move
+ Attack
+ Defend
+ Reload
+ Rest
Use
Use
o1.grid
Use
Use
GameMap
+ width: Int
+ height: Int
+ tiles: Vector[TerrainTile]
+ initialElements: Vector[TerrainTile]
ConfigurationConstants
All constants used in the program
TerrainTile
+ image: String
+ position: GridPos
+ flatness: Int
+ solidity: Int
+ vegetationDensity: Int
+ elevation: Int
+ degrade(Int): Unit
+ copySelf(GridPos): TerrainTIle
Sand
Forest
Rock
ConquestTile
Grass
Dirt
VegetativeDirt
Gravel
Soldiers
Sniper
Tanks

Most notably, it lacks the SupplyChain class and the Condition enumeration as the former was cut and the latter turned out to be redundant. Other than more dependencies being explicitly show, along with the unit tests, a few more TerrainTiles and a ConfigurationConstants file, besides the name change and decrease in scope of the Configurator that is now GameConfig, the structure of the entire program is more or less fully consistent with the original UML draft. Battle units duplicate array was removed from map and so GameMap and BattleUnit are no longer directly connected.

The game class works as the central processing unit of the program, connecting all classes to each other to the least degree necessary so that maximal independence is retained. Most importantly, the GUI communicates exclusively with the game class and is so kept isolated from all others, although game and GUI are highly interdependent.

The game class stores the game state both on an overall level as well as per-turn basis and handles all changes to it. As the central component of the program, connecting all others, this is where the probabilities for all actions are calculated and the game logic is executed.

The GUI is a single, massive, 900-line object responsible for everything the user sees on screen starting from the window itself and is thus where the game is launched. This was the least planned part of the entire project, as can be seen in the original draft UML but it was the largest time-sink and source of issues by far, perhaps because of that exact reason. However, having not known any ScalaFX or much of any user interface development besides some HTML and CSS prior, I do not believe that I could have done much better with that knowledge. In hindsight, however, I would probably modularize the GUI by splitting it into smaller classes based on layouts / components and functionality by having, for example, a render engine be a purely functional class that handles all changes automatically instead of the now and then repetitive and self-interdependent spaghetti that it currently is. Spending countless hours on debugging, I certainly learned a lot about how the user interface could be built in a more coherent and sensible way, but it was unfortunately a bit too late for major changes so late to a long file whose number one top priority trumping all else was user experience for the player instead of the developer. With the knowledge and experience that I had, however, I am happy to have created something that looks decent and is very intuitive, seamless and fun to interact with even if the code is suboptimal.

All other classes are fairly simple and straightforward both conceptually and in practice. The GameMap simply holds the TerrainTiles, which contain the images and properties as well as the degrade method for each tile on the map. BattleUnit does the same for battle units with a few more variables and methods and ActionSet is essentially a container for actions and their targets for one battle unit to attempt during a turn. Player class holds the battle units and is used identify the winner and whose turn it is. GameConfig loads a configuration file and creates appropriate objects as a result and GameSpec and ConfigSpec contain unit tests for Game and GameConfig respectively.

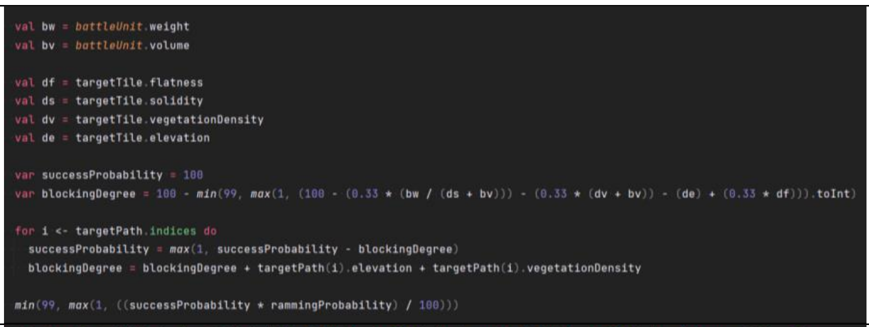
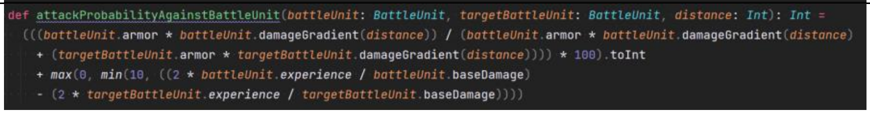
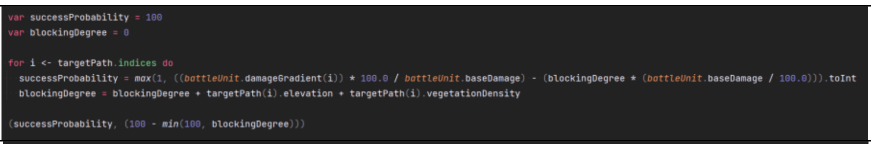
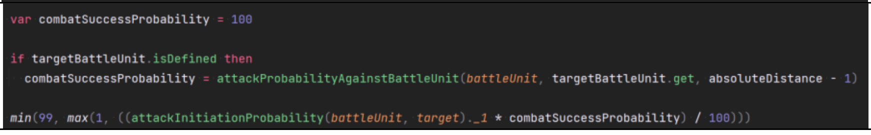
As I wanted to optimize the game for real human players, playing against each other, it made sense for the AIPlayer to simulate one by having it only interact with the GUI just like a human would but with synthetic MouseEvents. However, for some reason this introduced issues that I could not resolve quickly enough in the span of a single night and I had to therefore occasionally manipulate the Game class directly too, in addition to it providing the vision for the AIPlayer so to say.

The project has three dependencies: ScalaFX used for building the GUI, ScalaTest used for unit tests and o1.grid from the Programming 1 course that mostly just serves the purpose of representing coordinates in a non-tuple form. GridPos classes unique methods are used more extensively by AIPlayer and by the private fovTiles method inside Game. It could therefore easily be replaced by a similar custom class but it was a decent basis to build on top of and so made sense to use accordingly to the programmer ethic of never coding anything twice as only very trivial functionality was required.

Algorithms

The most important algorithms of the game are the methods for calculating success probabilities for actions and the "AI" Player's behavior. Everything else is rather trivial and/or non-unique or non-essential in terms of what makes the game unique.

The probabilities are calculated with the following formulas:

Move probability	
Attack probability Against another battle unit	
Probability for successful initiation of a duel	
Combined attack probability	

```
val bw = battleUnit.weight
val bv = battleUnit.volume

val df = targetTile.flatness
val ds = targetTile.solidity
val dv = targetTile.vegetationDensity
val de = targetTile.elevation

var successProbability = 100
var blockingDegree = 100 - min(99, max(1, (100 - (0.33 * (bw / (ds + bv))) - (0.33 * (dv + bv)) - (de) + (0.33 * df))).toInt)

for i <- targetPath.indices do
  successProbability = max(1, successProbability - blockingDegree)
  blockingDegree = blockingDegree + targetPath(i).elevation + targetPath(i).vegetationDensity

min(99, max(1, ((successProbability * rammingProbability) / 100)))
```

```
def attackProbabilityAgainstBattleUnit(battleUnit: BattleUnit, targetBattleUnit: BattleUnit, distance: Int): Int =
  (((battleUnit.armor * battleUnit.damageGradient(distance)) / (battleUnit.armor * battleUnit.damageGradient(distance)
    + (targetBattleUnit.armor * targetBattleUnit.damageGradient(distance)))) * 100).toInt
    + max(0, min(10, ((2 * battleUnit.experience / battleUnit.baseDamage)
    - (2 * targetBattleUnit.experience / targetBattleUnit.baseDamage))))
```

```
var successProbability = 100
var blockingDegree = 0

for i <- targetPath.indices do
  successProbability = max(1, ((battleUnit.damageGradient(i)) * 100.0 / battleUnit.baseDamage) - (blockingDegree * (battleUnit.baseDamage / 100.0))).toInt
  blockingDegree = blockingDegree + targetPath(i).elevation + targetPath(i).vegetationDensity

(successProbability, (100 - min(100, blockingDegree)))
```

```
var combatSuccessProbability = 100

if targetBattleUnit.isDefined then
  combatSuccessProbability = attackProbabilityAgainstBattleUnit(battleUnit, targetBattleUnit.get, absoluteDistance - 1)

min(99, max(1, ((attackInitiationProbability(battleUnit, target)._1 * combatSuccessProbability) / 100)))
```

Where target path contains the tiles between the battle unit and the tile for which the probability is being calculated. Shown in code because translating into mathematical equations would be very laborious.

Especially move probability is a very inefficient implementation but it was the first that came to mind and the responsiveness of highlights when selecting a battle unit on the map did not suffer at all, so I decided to focus my efforts elsewhere.

The formulas themselves are not special in any way. I came up with them by just experimenting with the ranges of values that the BattleUnit and TerrainTile attributes would take and fine tuned their sensitivity to each attribute by pure intuition.

Codifying the multiple different probabilities associated with each action into single figures, however, was one of the hardest tasks because I had to somehow convey the effects of distance, type of tiles in between and the relative strength of one battle unit with respect to another in a single number, not to overwhelm the player. For relative strength, the value should be 50 for battle units of the same type and more or less for different types whereas if it is on the other side of the map or there is a rock or a forest in the way, it should be less. For the case of the attack probability, I settled on a solution where the relative success probability in combat is masked by the probability of successfully initiating the attack by multiplying them together and dividing by 100 inside min and max that make sure that the probability stays within the interval 1 – 99 because I do not want anything to ever be certain or entirely impossible. This might require the player to develop some intuition about the relative strengths of the units, their damage gradients and different terrain properties. For example, two tanks on a desert are shown only 49% probability of winning even though it is really 50%. That 1% is taken away by the possible sand dunes in between.

This also closely follows the progression of a duel, for which, first, the probability of successfully initiating the duel, i.e. actually being within hitting distance of the attacked, is calculated. If this condition is passed, the potential winner is selected next by comparing if a randomly generated integer to the probability of beating the other unit in combat. If it is less, the attacker is considered for winning and if it is more, the attacked is considered for winning. What this means is that the attack initiation probability is calculated again for the winner as if they were the attacker. Only then, the loser takes damage equivalent to evaluating the winner's damage gradient at that distance. This enables tanks picking fights with snipers at a distance and dealing little damage upon success whereas they can get destroyed by the sniper that does a lot more damage from distance a lot more certainly.

The damage gradients convey information about how much damage its possessor can deal at certain distances. This is the amount of damage dealt by the winner but is also used in calculating probabilities of winning as a soldier with short distance weapons has very little chance to hit a far away, accurate sniper. These are created in the configuration files as lazylists that iterate through the distance using a base damage value and a function.

For soldiers this is a very simple halving function such that they deal maximum damage at distance one and the drop off is half of that for every consecutive tile:

$$damage\ at\ distance\ n = \frac{1}{2}^{n-1} * base\ damage$$

For tanks it is:

$$damage\ at\ distance\ n = \begin{cases} 10\ for\ i = 1 \\ \frac{2base\ damage}{2^{i-1}}\ for\ i > 1 \end{cases}$$

And for snipers it is an even shallower gradient that can be found from the code.

The “artificially intelligent” opponent is by far the weakest link of the entire project simply due to how little time I had for implementing it. The general gist of it is to iterate through all alive battle units of the player it controls and either attack, move, refuel or reload depending on its immediate state and surroundings. The algorithm is in this sense selfish on the level on the individual battle units that it only considers each one at a time, viewing others from the same team simply as obstacles.

First it checks whether the battle unit is out of fuel or ammo and if this is the case, refuels or reloads accordingly. Then, if there is a sufficient chance of winning a duel that is greater than the chance of simply ramming the opponent over if it is within move’s reach, the battle unit will attempt to attack that enemy in its range that it is most likely to beat. If it is confident about winning the duel, it attempts it twice but if not, it simply escapes if the attack fails. If there is a battle unit in range for which ramming it has a higher probability than attacking it, it should attempt to run it over but something must be wrong with that or the previous part of the code as this is not observed in reality. If the battle unit is next to a conquestTile, it attempts to move onto it or, if it is further away, it attempts to move towards it or another enemy battle unit if all conquestTiles are occupied by friendlies. If the way is blocked, it attempts to move left or right of that direction to keep the overall course and as a catch-all-situation it finally tries to move just on random. Injecting this bit of randomness in the end and in some secondary actions of the other move attempts improved it significantly.

The one and only remaining issue with the stupid, selfish AI is that despite the description, it is way too polite. Everything works as expected, except that the AI refuses to move BattleUnits to tiles with dead opponent units on them. It has no problem running over its own dead companions but it simply refuses to ram to opponents or stand on top of them even if they are dead, although this can easily be done by a human player. Therefore, the current best way to win the AI opponent is simply to just block it off from the conquest objective using your BattleUnits' corpses.

Data structures

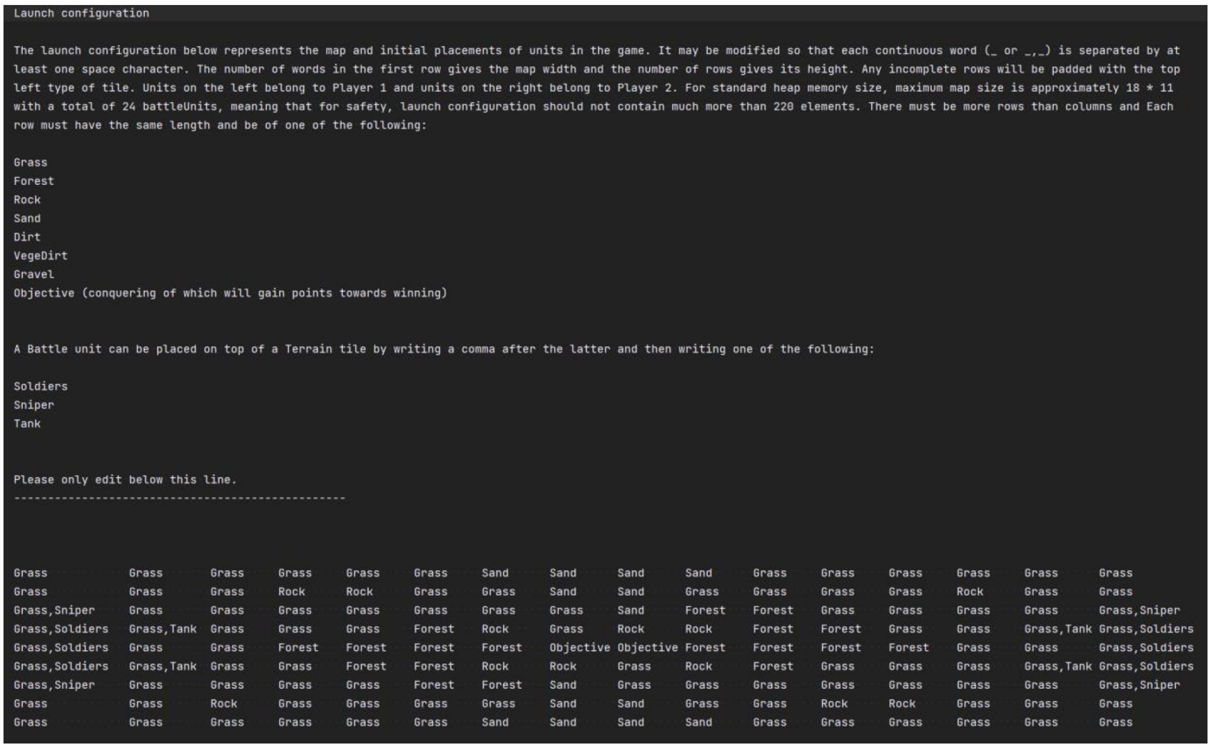
Data structures used were almost exclusively immutable vectors, often stored in variables for editing, due to their high versatility, ease of use and decent operations efficiency. LazyLists were utilized for the damage gradients to accommodate for variable map sizes and everything else is just standard Ints, Strings and Options. I must add a note about the use of constants and options in the program. Every meaningful constant is stored and globally editable and accessible from the ConfigurationConstants file but there are various 1s, 0s and 100s and even some 2s around for index access and handling percentages, the changing of which would only break the project and thus I do not consider them to be magic numbers. I am also going against the best practices of Option usage in almost exclusively using .get to access their values. I find that if just tested for every time with an if statement, their use is far more convenient and natural. This is basically the same as using a match statement but more natural in my opinion when the code blocks between grow in size.

My only custom data structure is the ActionSet class, which holds the primary and secondary actions and their respective targets and a state variable indicating whether the first action was successful or not. It was an eloquent and easy way to package and process control information but was rather simple and straightforward to implement.

Additionally, various ScalaFX object properties and observable buffers were used to automatically update GUI elements more out of necessity than by choice.

Files

Initially, I intended to make the entire game fully configurable down to the most minute constant but at the approach of the deadline, even though this would be rather quick and easy addition, I opted for a more simple, custom human readable launch configuration text file that configures the map layout and battle unit formations for each player (player 1 to the left of center and player 2 to the right), which can be seen open in IntelliJ IDEA below.



```
Launch configuration

The launch configuration below represents the map and initial placements of units in the game. It may be modified so that each continuous word (_ or _,_) is separated by at
least one space character. The number of words in the first row gives the map width and the number of rows gives its height. Any incomplete rows will be padded with the top
left type of tile. Units on the left belong to Player 1 and units on the right belong to Player 2. For standard heap memory size, maximum map size is approximately 18 * 11
with a total of 24 battleUnits, meaning that for safety, launch configuration should not contain much more than 220 elements. There must be more rows than columns and Each
row must have the same length and be of one of the following:

Grass
Forest
Rock
Sand
Dirt
VegeDirt
Gravel
Objective (conquering of which will gain points towards winning)


A Battle unit can be placed on top of a Terrain tile by writing a comma after the latter and then writing one of the following:

Soldiers
Sniper
Tank


Please only edit below this line.
------------------------------------------------



Grass            Grass      Grass     Grass     Grass     Grass     Sand      Sand      Sand      Sand      Grass     Grass     Grass     Grass     Grass      Grass
Grass            Grass      Grass     Rock      Rock      Grass     Grass     Sand      Sand      Grass     Grass     Grass     Grass     Rock      Grass      Grass
Grass,Sniper     Grass      Grass     Grass     Grass     Grass     Grass     Grass     Sand      Forest    Forest    Grass     Grass     Grass     Grass      Grass,Sniper
Grass,Soldiers   Grass,Tank Grass     Grass     Grass     Forest    Rock      Grass     Rock      Rock      Forest    Forest    Grass     Grass     Grass,Tank Grass,Soldiers
Grass,Soldiers   Grass      Grass     Forest    Forest    Forest    Forest    Objective Objective Forest    Forest    Forest    Forest    Grass     Grass      Grass,Soldiers
Grass,Soldiers   Grass,Tank Grass     Grass     Forest    Forest    Rock      Rock      Grass     Rock      Forest    Grass     Grass     Grass     Grass,Tank Grass,Soldiers
Grass,Sniper     Grass      Grass     Grass     Grass     Forest    Forest    Sand      Grass     Grass     Grass     Grass     Grass     Grass     Grass      Grass,Sniper
Grass            Grass      Rock      Grass     Grass     Grass     Grass     Sand      Sand      Grass     Grass     Rock      Rock      Grass     Grass      Grass
Grass            Grass      Grass     Grass     Grass     Grass     Sand      Sand      Sand      Sand      Grass     Grass     Grass     Grass     Grass      Grass
```

The file is structured such that the instructions for its safe modification are written on top and the actual space-separated configuration is parsed such that the first non-space character after the dashed line will be interpreted as the first character of the first map tile. This is then read one row at a time into a map with width equal to the number of words on the first line and height equal to the number of lines.

The launch configuration represents the map and initial placements of units in the game with units being specified on top their starting tiles, separated from them only by a comma (no spaces). It may be modified so that each continuous word representing a TerrainTile and a possible BattleUnit (_ or _,_) is separated by at least one space character. The number of words in the first row gives the map width and the number of rows gives its height. Any incomplete rows will be padded with the top left type of tile. Units on the left belong to Player 1 and units on the right belong to Player 2. For standard heap memory size, maximum map size is approximately 18 * 11 with a total of 24 battleUnits, meaning that for safety, launch configuration should not contain much more than 220 elements. There must be more rows than columns and Each row must have the same length with its words each being one of the following:

- Grass
- Forest
- Rock
- Sand
- Dirt
- VegeDirt
- Gravel
- Objective (conquering of which will gain points towards winning)

A Battle unit can be placed on top of a Terrain tile by writing a comma after the latter and then writing one of the following:

- Soldiers
- Sniper
- Tank

The file is read out of a directory ("src/main/resources/launch-configuration/" by default) that is assumed to only ever contain a single file, so that the file's name does not matter. The project is shipped with two launch-configuration files that are meant to be swapped in and out of the default folder for the game to be initialized with its contents. The configurations folder is meant for storage of custom launch configurations and launch-configuration is meant to host and use the currently active one. If the launch-configuration folder is empty or contains an illegally formatted file, the game defaults to the launch configuration defined in ConfigurationConstants.

Testing

Testing was mostly done through the GUI especially in the beginning and the end. In the beginning when the graphical user interface was just being built, it consisted simply of visually inspecting that each GUI component was indeed visible and functioning as intended but later when implementing the artificially intelligent opponent, it consisted mostly of making it play against itself and observing its behavior as well as any weird quirks, bugs or glitches emerging from it, of which there were plenty initially.

I wrote some unit tests for the Game class to make sure its methods were functioning correctly at all times, although to be honest, this was more for the optics than me actually getting real value out of them for development as I felt that it mostly just slowed me down. By definition, fully testing a method requires multiple tests and therefore I would have been writing more tests than program, the benefit of which is debatable in such a small-scale, non-critical application with a tight deadline. The game does not allow for variable input but is entirely self-contained and therefore it was pretty easy to just write it to do what was specified.

I did, however, get to try test-driven development as well with the loadConfig function, which I decided to test as thoroughly as possible as for reading input it makes a lot of sense to cover as many cases as possible. I created every possible type of file I could imagine and defined what kind of behavior I wanted from loadConfig function in each case, wrote corresponding tests and then shaped the function to comply to all of them one at a time. The test coverage of that method is nearing probably 95% and I can thus be quite confident in it working according to specification in most situations and even if a new type of error or exception occurs, every throwable is caught in game initialization and default configuration is used instead so all in all, file configuration should be very safe. The number of tests necessary to achieve that just proves my point about it potentially being a major weight in light non-critical applications. However, I do also understand very well why it is advocated.

The program passes all tests, both unit and manual. The testing worked out a bit differently in the end as I did not have enough time to test even every game method once. Also, the artificially intelligent opponent was implemented way later than anticipated and therefore manual testing was used even a little more extensively than initially thought. In summary, GameSpec is me mostly just learning about ScalaTest and trying to get used to it whereas ConfigSpec represents something actually useful.

Known bugs and missing features

As already discussed above 5/8 of the planned game mechanics were implemented and thus those 3 could be called missing but after making the decisions to cut them, I feel that the game is whole as is and they cannot therefore be called missing per se.

In terms of the user experience, the game is as polished for mouse input as it can be but keyboard control might be more attractive for more seasoned strategy players as the mouse is just fundamentally slower. This is thus something that could be added but is more of a bonus than a missing feature.

As also already mentioned above, the “artificially intelligent” opponent is far from what it could be and still a bit broken. It does not implement any kind of strategy other than “go and kill” and for some strange reason I could not figure out at this time, it refuses to place units on top of dead opponents, which is often necessary for changing the current conqueror.

As also detailed above in the user interface section, there are issues with scaling health bars up but especially down as they might overflow and push other elements in the grid, which I classify as a bug. Proper, safe dynamic scaling both up and down might be said to be a missing feature. The components have absolute minimum dimensions and will therefore not scale down but will instead just disappear.

Outside of the GUI and the AIPlayer, there are not too many recorded bugs remaining. Again, related to the AI, if the AI Player is playing against itself, and wins the game. The winner message will always be red, regardless of who won. This is not the case with human players.

The most critical issue is that of heap memory running out if too large of a map is loaded, which fails to open the game entirely until the configuration file is modified. According to quick testing, 18 * 11 seems to be the largest safe map configuration. It is also not tested for before opening the file and therefore does not cause any catchable exceptions, meaning that encountering it will be fatal every time. In the future, I might add a check for this or inspect how to increase the allocated heap size for larger maps but for now it is a limitation of the launch configuration files that must be obeyed if you want the game to open. Maps that are not sufficiently wide, may also push the bottom pane out of out of the window, making full screen unfeasible.

3 best sides

- Seamless and intuitive user experience
  - It feels effortless and good to use
  - Most GUI controls do exactly what you would expect them to do
  - GUI does not look super terrible and the layout is very logical and clean

- Well-rounded demo optimized for human against human local games
    - Semi-original, interesting idea and novel, somewhat enjoyable gameplay
- Launch configuration file and the design of BattleUnit and TerrainTile classes
    - The launch configuration file is well designed in that it is very readable and intuitive to edit as well as easy to load by the thoroughly tested loadCondig() method of the game – the game always reliably loads
    - BattleUnit and TerrainTile classes are clear, coherent and highly extensible.
    - New types of units and tiles can very easily be added in code with any types of attributes as long as integer values range from 0 to somewhere a bit above 100 and the game adapts to them well.
    - Calculating probabilities from unit attributes allows for easy and very fine-tuned balancing of the game.

3 weaknesses

- "Artificially intelligent" player
    - Is too polite and does not ram or move on top of opponent battle units, whether they are dead or alive
    - May often get stuck because of the above
    - Has a very crude, predictable and selfish algorithm that does not create any too interesting behavior such as actual strategies or cooperation between battle units, although sometimes they might amusingly shoot at nothing or turn away at odd times (not verified, simply by the real time looks of it)
- Thread handling is very crude
    - The threads are never synchronized and timing is controlled simply by duration of sleep
    - As a consequence, when the AI player plays against itself, the turns played by it do not take constant time and have even a risk of overlapping perhaps especially on slower computers
- Inefficiency of many algorithms
    - Not too big of an issue in this scale but prevents the seamless scaling up of maps and numbers of BattleUnits
    - May be behind some of the heap memory issues

Deviations from the plan, realized process and schedule

The realized schedule ended up being quite rear-heavy, resulting in a total of approximately 8 hours of sleep in the past five days as of the writing of this document.

The game is first and foremost optimized for local human versus human gameplay, for which it works flawlessly. Because of this, however, file reading and the “AI” player were implemented just this week as refining the gameplay and associated bug hunting, each of which often ended up taking an entire day or two, took so much time. It was a common occurrence for an otherwise ready commit be delayed up to two days because of doing some manual testing and finding some bug, sometimes harmless, sometimes very consequential, right before closing the GUI and clicking commit. This is often the explanation in the gaps in the progress log entries, especially in the past couple of weeks, that were often mostly written the next day after the last commit but were only submitted a day or two later after the bug had finally been resolved. I documented and mentioned many of these but not all.

The process, expectedly and still, unexpectedly, took way longer than initially planned. The beginning, which already happened later than it should have because of other courses, was a bit slower with me trying to get used to ScalaFX and ScalaTest, the learning of which was greatly helped by ChatGPT in place of documentation that would be geared towards a first timer newbie. The pace picked up once I figured the way ScalaFX is structured and gained familiarity with how it is supposed to be used but learning it was definitely one of the biggest speed bumps, in addition to the six or seven additional courses I was studying for in the first period, because making GUIs is easily the most neglected aspect of the basic programming courses. Other slowing factors were probably my perfectionism and commitment to only clean commits, the benefit of which is difficult to weigh at the moment because it likely saved me from a ton of future work and debugging but the time saved cannot be compared to the cost of staying up nights, just staring at the screen, wondering why something does not work and will therefore remain indetermined.

The order of implementation also changed a bit such that the GUI was functionally implemented to a much further extent than I thought would have been necessary at each step. This was because I realized that it makes most sense in a project of this scale with this timetable to just build each component as close to its final form as possible because there is no time for multiple iterations on all of them. This, however, might also have introduced some rigidities that may have made the implementation of other components more difficult.

The configurability of the game was also pushed from the earlier half to the very last days because I somehow thought that it would just be reading some JSON file quickly, for which there should be close to end-to-end solutions in libraries such as Circe, as long as I keep everything nicely packaged in the ConfigurationConstants file. When it came to it, however, I realized that this task was less straightforward than initially thought because the map and battle unit formations consisted of custom objects and their representation would be a bit more difficult and very unintuitive in a JSON file. Therefore, I decided to create a custom text file after all, where the map and battle units are represented visually in the form that they will be drawn in game. Now, for the remaining constants an additional JSON configuration files would likely be a very quick addition, but I must allocate time for other closing deadlines already and therefore decided to forego it now as modifying the ConfigurationConstants file does the same trick and according to my TA, it was enough to have the layouts be loaded from external files. Besides these the order was mostly followed. Only their required development times and the adapting of other classes to support the features is not represented by how much space they took in the time estimate table of the technical plan.

For more details and low level, technical breakdowns of each decision, read the progress_log.txt in the root folder of the source code.

Final evaluation

This was quite a rollercoaster. Any smoother patch was met with a deep ditch of great difficulties, but I certainly learned a lot from both. This was definitely a formative experience in software creation overall for me, demonstrating how vague ideas must be formalized into detailed, concrete plans, the implementation of which can then bring them alive but definitely not immediately nor without hardship and difficult choices whose effects remain to be confronted for the rest of the project.

Overall, I am quite satisfied with how the game turned out. The human versus human gameplay, which was the main focus from day one, and interacting with the GUI is very seamless, intuitive and effortless such that the player should only be frustrated about the opponents moves rather than the GUI or the game itself. It is a very polished experience and has a decent look, considering it has not been written with CSS and the all map tiles and battle units were haphazardly self-drawn during the lectures.

However, the "artificially intelligent" opponent, although significantly better than an agent acting at random, is subpar and deficient in almost every aspect. Its strategy is a very simplistic and selfish "go and kill" type of approach without any coordination with other units. Also, for some reason, without programming in this respect, it chooses not to overrun opponent battle units ever, even though it is entirely content doing so to its own, and is therefore a very easy opponent to beat, making single player game sessions quite easy, predictable and boring. However, this gives it a quirky personality and me a story to tell fellow programmers.

The game is also not quite as scalable in terms of map size as I would like, maybe because the images are too large and there is not any kind of file size optimization, or the game gets launched with too small heap memory or there are some issues with garbage collection. As this issue occurs upon loading already, I highly doubt the latter though. The very inefficient implementation of the probability calculation methods may also affect this eventually and therefore these could be improved.

These could be improved just by spending more time on the AI and learning more about the way ScalaFX handles images as well as various other, perhaps even quite minor optimizations but they do not interfere with the central action that is the human player versus human player action on a non-zoomable map of sufficient size for interesting games but also visibility.

These would be obvious improvements for the future along with additional configurability, such as setting the conquest target to any value. Additionally, if the scaling issue is fixed, the map could be made zoomable and navigable for larger battlefields. Also, more mechanics such as the supply chains and their management as well as the simultaneous turns as well as random generated maps could be added.

The structure of the program should accommodate these quite well other than the bit spaghettified GUI, where potentially a lot more code and editing might be required for displaying the supply chains. It could be rewritten such that it is split into multiple different component classes and a render engine so that the map is explicitly rendered in multiple layers with varying number of items such that the tiles are on the bottom, dead battle units are on the second layer, supply chains on the third, alive battle units on the fourth and probabilities and highlight squares on the fifth and above, which would make it significantly easier to just add layers if one wants to add, for example effects. Thus, the GUI is unfortunately not very easily extendable at its current state other than for new or different components. Its look is easy to modify but its functionality is not.

For any other changes or extensions, the game is highly accommodating. Simultaneous turns would simply require an advances algorithm in the simple PlayTurn method of Game, randomly generated maps could be added by simply adding one more generator method to the map class that may or may not use symmetricTileUpdated with desired type of symmetry. New terrain tiles can be added simply by adding a new class with different properties and degrade function and same goes for battle units, although they need a bit more values and a damage gradient and the properties of all of these can be modified in the ConfigurationConstants. Even new actions can be added by simply adding it to the Action enumeration and Game’s executeActionSet method and linking it to its own method that may use one of the existing probability functions, the behavior of which can be modified often by just adjusting weightings, or by creating a new one.

References

CS-C2120 Ohjelmointistudio 2 (2023) *A+*. Aalto University. Available at: https://plus.cs.aalto.fi/studio_2/k2023/ (Accessed: May 5, 2023).

(2022) *O1 Library Docs*. Aalto University. Available at: https://gitmanager.cs.aalto.fi/static/O1_2022/modules/given/O1Library/doc/o1/grid/GridPos.html (Accessed: May 5, 2023).

ScalaFX • simpler way to use javafx from scala (no date) *ScalaFX*. Available at: https://www.scalafx.org/ (Accessed: May 5, 2023).

Scalatest (2023) *ScalaTest*. Artima. Available at: https://www.scalatest.org/ (Accessed: May 5, 2023).

(2023) *ChatGPT*. OpenAI. Available at: https://chat.openai.com/ (Accessed: May 5, 2023).