
The
WAR
Scripting Language

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East
Telefon 99 40 99 40
http://cs.aau.dk

Title:

The *WAR* Scripting Language

Theme:

Tools, Languages and Compiler

Projectperiod:

DAT2, spring semester 2011

Projectgroup:

D204A

Group members:

Mads Vestergaard Carlsen
Rasmus Søgaard Jacobsen
Kristian Pilegaard Jensen
Dan Duus Thøisen

Supervisor:

Paolo Viappiani

Circulation: 7

Pages: 85

Appendix Number: 7

Finished on 27-05-2011

Synopsis:

This report documents The *WAR* scripting language. This documentation covers various aspects of the design process and implementation process, in details such as Design Decisions, Syntactical Analysis, Contextual Analysis, and the documentation for using the language in itself. A simulation engine was developed for the script to execute on and is also documented in the report. The language in the report is implemented through interpretation.

Preface

This report is about the project that, was developed in the Spring semester of 2011, by the group D204A at the Department of Computer Science at Aalborg University. The group consisted of students attending their fourth semester of Computer Science. The project was undertaken in under the supervising of Paolo Viappiani.

In the report, the reader is assumed to have basic knowledge of programming languages, and a certain insight into the overall structure of the implementation of a programming language and the meaning of compilers and interpreters. The topic of the report is development of a programming language, and thereby the implementation of an interpreter for this language.
The language is developed in C♯ which assumes the reader is familiar with basic object-oriented programming paradigm [1].

In addition to the report itself, a disc is included containing the source-code and the binaries for further examination.
In order to run or compile this code a .NET Framework needs to be present and functioning on the machine.

The report documents the process of the language design, and details how the language is implemented. By doing so it also describes usage cases and testing along with an overall description and images of the application itself.

References are cited using the Vancouver style and are found in the bibliography at the very end of the report. Figures and tables are numbered in accordance with Chapter and section numbers throughout the report.

Contents

Chapter 1

Introduction

Designing a programming language is a task that is not too easily undertaken, not only is the implementation quite a unique process, decision making is also critical in order to create a properly functioning and well-rounded language. Today, many major implementations of multi-paradigm languages exist, which are all extremely thoroughly developed and is in widespread use in the world. Extensions and libraries for major language implementations make them extremely powerful tools, along with the support of the users of that. Reigning supreme is the ever so popular *C* and Java, due to their widespread use an excellent compatibility. But not all languages are necessarily designed with the purpose, sometimes it might be practical to implement a minor language, or maybe even a major one for a specific purpose. In computer science having the right tools at hand is crucial, and if the tools are not available, you will have to develop them yourself.
A purpose for such a specific language is in this case a scripting language for a simulation. Such a specific language will have to be tailored to very specific needs of the simulation in question.
Similar applications of scripting languages such as this do exist, as contests are held amon robotics students to see who can develop the best behaviour of a machine. Developing the best script to defeat an opponent, which may or may not be dependant on the script your opponent has set forth is an interesting approach to see how basic dynamic state-handling can govern a unit and imitate intelligent behaviour.

1.1 Purpose

The purpose of the language in itself is not to do anything revolutionary in regards of language design, but rather to plug in to a hosting application. The

script will then alter the flow of the hosting application entirely dependant upon the script set down by a programmer. To provide a meaningful context for this language implementation we would create a simulation 'game'. Strategy games are a very popular popular genre within the modern game-industry. In strategy games, your victory or defeat depends primarily on the applied tactics. However, most modern strategy games leave room for error through inferior control or attention on the battlefield. The purpose of our language is to define your tactics in advance and have the game play out as predicted by the best abilities of the scripters. As such the main and overall purpose is the implementation of a minor language.

1.2 Problem Statement

During the course of this project our goals were to:

- Apply concepts of programming language design
- Determine important design choices and argument for these choices
- Provide a meaningful context for an implementation of a simple language
- Implement our own interpreter or compiler
- Build a running simulator

1.2.1 Outline of the report

In the following, we give an overview of the structure of the report.

- **Chapter 2** describes the problem domain.
  In this Chapter we describe the game in details, giving an internal semantics to the possible actions and states(situations) that can occur during the game, as well a define certain terms and rules of the game.
- **Chapter 3** presents our high level design choices with respect to the language that constitutes the 'core' of our system.
- **Chapter 4** presents the language in detail from the point of view of a programmer that needs to program a regiment for our system. The specification of the language, is presented, documenting the principle elements of the language.

- **Chapter 5** describes the actual design of the language *WAR* representing it in a formal way with the known Backus-Naur Form, and later the Extended Backus-Naur Form. This chapter also represents how the scope-rules and type-rules of the language works.
- **Chapter 6** this chapter is about the implementation of all the different parts of the language. It describes in details how the 4 parts; scanner, parser, contextual analyzer and the interpretation are implemented.
- **Chapter 7** this chapter is about tests of the simulator. Scripts for the simulator and screenshots from the simulator are shown here.
- **Chapter 8** this chapter represents the ideas that was not implemented, but could have been fulfilled in later projects.
- **Chapter 9** this chapter is a conclusion, where we conclude on our design criteria found in chapter 3. This is also where we conclude on the problems we might have experienced.

Chapter 2

Game Description

In this chapter we describe the general setting of the project.
The problem we want to solve is the following: we want to make a war simulation where competitors, instead of controlling the armies directly, are controlling them by scripting their behaviours. The winner is the scripter who makes the best script. This is an interesting and fun problem, but also challenging because it requires a lot of expertise in language design and compiler techniques. The simulation or 'game' based on our scripting language is described in detail in this chapter.

2.1 The Idea

The main idea of the game was to create a generic war game simulator, where through programming one could make many different types of simulated battles. We drew inspiration from many different sources, in order to come up with an aspect that was interesting, while being feasible with our current technical abilities. The framework and setting is mainly inspired by tabletop games, real-time strategy games and chess, coming together in a rather simple package to provide with a somewhat interesting abstraction for a simulation which is implemented in our own programming language. During the process we considered many different concepts, and finally we decided to create a war simulation, with two or more opposing forces spawning in a grid like fashion.

Interesting aspects of a simulation, of this sort, is to establish basic ground rules, and see how the simulation proceeds on its own, given only basic instructions on how to start the simulation.

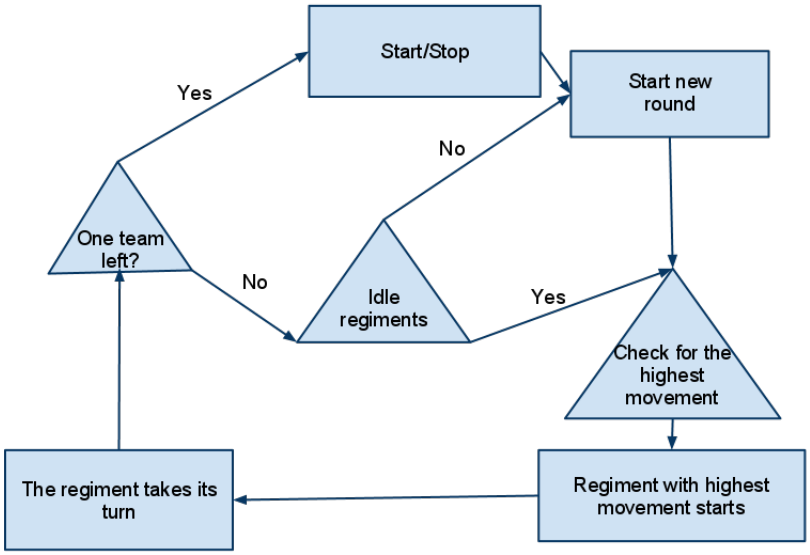


Figure 2.1: Flowchart of the simulation

2.2 Description of The *WAR*-Game

As mentioned earlier the *WAR*-Game is a simulation where the players will script the behaviour of their armies, and let the 'game' execute without further interaction. The winner of the simulation will be the one with the best script, as decided by the simulation engine.

2.2.1 The simulation

To run a simulation in the *WAR* simulator, a configuration file and at least two team files has to be included into the simulation upon start up. These files are written in the *WAR* scripting language. The configuration file is used to establish basic rules for the war simulation i.e. the size of the battlefield, the maximal values of properties and the standard properties of regiments. This allows for some flexibility in altering the simulation from game to game. Each participant will make a script file over their team of regiments, which will be provided to the simulator, which will then interpret the script and simulate the battle. These script files contain how many units and what kind

of units the participant would like to use. They may also contain information regarding the behaviour of specific regiments. The one who makes the best script will win the game, so every participant should strive to make the best script, and place their regiments in the best locations on the grid.

2.2.2 The Language

We designed the WAR language as a simple imperative scripting language. Indeed we wanted the language to be easy to program, so that inexperienced scripters could be able to control a simulation. Thus the imperative paradigm is what we felt best supported the idea of a simple programming language. We do not need the powerful abstractions of object oriented programming nor the task-solving capabilities of the functional languages. At the same time, the language should provide enough support, that more experienced scripters have an advantage over inexperienced scripters - the winner should be determined by their scripting skills.

2.2.3 Game Specification

In this section we define the terminology we use in the description of the game. These correspond with the terms used in the configuration- and team file.

Terms of the simulation

Team A team consists of 1 or more regiments. In the simulation two or more teams will battle to win the simulation. Here, a team is synonymous to an army.

Turn Each of the regiment will have a turn each round where they can perform a limited number of actions. The regiment have a *movement* constant declared which declares the maximum of movements the regiment can make. If the regiment have to attack, it can attack once and then its turn is over. The actions a regiment is allowed to do is described in 2.2.3. Further description of events during a turn, will be described in the following section.

Round A simulation consists of 1 or more rounds. When a round begins each regiment will be given a turn. The teams take turns performing the behaviour of their regiments. A new round will start when every regiment have used their turns. Further description of rounds are found in the following section.

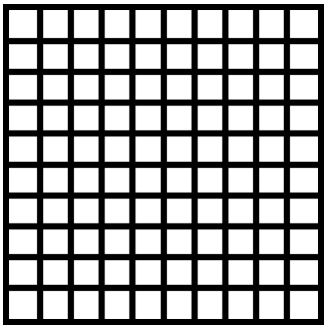


Figure 2.2: A 10x10 grid

Behaviour Behaviour allows the players to define how each regiment behaves. To control how a regiment behaves control statements can be used. As an example of a behaviour you may instruct a ranged regiment to behave in such a way, as to not approach another regiment, and only open fire if that regiment is within the range of your archer regiment.

Grid The grid is a 2 dimensional map where the teams will battle. The grid, might be seen as a map, table, or battlefield where the 'battle' will take place. Figure 2.2 shows how a grid with the size of 10 by 10 would look like, this grid is without regiments.

Terms in the team file

Size The size of the regiment is, in abstraction, the number of soldiers in the current regiment. The size is closely connected with the total health of the regiment, as well as total damage done by the regiment.

Health Health describes the initial health of each unit in the regiment. The total health is calculated from the size of the regiment, by using the equation $Total\_Health = Size \times Health$.

Attack Type The Attack *Type* defines how a regiment attacks. A regiments Type can only be *melee* or *ranged*. *Melee* regiments can only attack regiments adjacent to their position, while ranged regiments can attack any regiment within their specified range.

Range Range is an integer value, which defines how far a regiment of type *ranged* can attack. If a *range* is defined for a regiment of type *melee*, this is simply ignored.

Damage The damage of a regiment is an abstraction of the damage done by each unit in the regiment. The total damage done by one regiment, is calculated by the equation $Total\_Damage = Size \times Damage \times AttackSpeed$

Movement Movement defines how many tiles a regiment is allowed to move.

Attackspeed *Attackspeed* governs how many attacks are performed by regiment each turn, giving units further interesting properties. However, in the context of the simulation, the *attackspeed* is quite simply a multiplier of the damage done by the regiment.

Action

When a regiment has its turn they are allowed to do the actions, *attack*, *MoveTowards* or *MoveAway*. For describing these action we set a precondition(what is required to do the action), a failure(what happens if the condition is not met) and result(what happens if the condition is met).

Attack action

Usage: Attack(Regiment)

Precondition:

- It is the turn of this regiment to move.
- For a regiment of *attacktype* ranged the regiment must be in range.
- For a regiment of *attacktype* melee the enemy regiment must be 1 tile away(diagonals excluded).
- The Regiment must not have had attacked this round.

Failure: Nothing happens

Result:

- The enemy regiment current total health pool is calculated seen 2.2.3.
- The total damage of the attacking regiment is calculated seen 2.2.3.
- The damage is subtracted from the total health pool of the enemy regiment.

- The enemy regiments size is reduced by a number corresponding to the number of units that would die, in correspondence to the damage taken.
- The equation for calculating the new size of the attacked regiment is given by

  $$New\_Size = Size - (Damage\_taken/Health).$$

- If we assume, an attacking regiment that deals $2000 = 200 \times 10 \times 1$ damage, and a defending regiment that have $Health = 50$ and $Size = 200$, the resulting size of the defending regiment, after the attack, would be $200 - (2000/50) = 160$.

MoveTowards action

Usage: MoveTowards(Regiment)

Precondition:

- The regiment has its turn and the *tiles*(Available grid-movements - left,right,up down) which brings the regiment towards the target regiment is not blocked.
- The Regiment must be eligible to perform additional actions in this turn.

Failure: Nothing happens

Result:

- The regiment will move a number of tiles equal to its movement, in the direction of the specified regiment.

MoveAway action

Usage: MoveAway(Regiment)

Precondition:

- The regiment has its turn and the tiles (Available grid-movements - left,right,up down) which brings the regiment away from the target regiment is not blocked.

Failure: Nothing happens

Result:

- The regiment will move a number of tiles equal to its movement, in the opposite direction of the specified regiment.

Configuration of the Playing Field

The simulation will need basic instructions which is a common limitation of both teams, to ensure even 'armies'. As mentioned earlier this is defined in a configuration file.

Width Width defines the width of the grid measured in tiles. Defined with an integer value.

Height Height defines the height of the grid measures in tiles. Defined with an integer value.

Standards block This block will contain standard properties for regiments. These will be used if the necessary information for a regiment is not entered in the main script, the regiment will use these standard properties.

Maxima block This block sets the limitations for the properties of the regiments, e.g. regiments must be of size 200 or less.

2.3 Running a simulation

When the simulation engine has been provided with a configuration file and two or more team files, the simulation will run. During a simulation, a lot of things happen, that are not defined in the WAR code. This section will explore these events and provide some insight in what happens during a simulation.

Beginning the simulation

When beginning the simulation, the configuration file and provided team files are read and interpreted. The information found in these files is used during the simulation.

When the simulation is about to start, a team is picked at random. The chosen team is the first to evaluate its regiments turns. If there are more than two teams, the next team to go, is also chosen at random. The sequence of the teams will be consistent throughout the rounds of the simulation. All teams are added to a list of playing teams. Only teams on this list can participate in a round. When a team is initialized, each regiment related to the team, is stored in list, private to the team. This list contains all regiments, that are eligible to perform actions in a turn.

The events of a Teams round

At the beginning of a teams turn, the simulation engine checks if the team is still allowed to play. To be this a team must have a regiment left. Regiments are removed from a given team when their size is 0. If the team is allowed to play, each of the regiments on the team will have their behaviour performed.

2.3.1 The events of a regiments turn

If a given regiment is eligible to take a turn, the behaviour of the regiment is performed, according to the WAR script written. If the behaviour contains one of the methods; Attack, *MoveTowards*, *MoveAway*, they are performed, as described in section 2.2. If one or more of the three methods are performed, the grid is updated to reflect the change - be it the size of an regiment or the position of a regiment.

Chapter 3

Overview of the design choices

The last chapter is about the description of the WAR language and now in this chapter the language criteria will be discussed. Later in this chapter the language paradigm will be presented, why and how we made our mind on paradigms and lastly tombstone diagrams will be presented, both how the WAR language get interpret and how the interpreter gets compiled.

3.1 Design Criteria

When designing a new language one can use design criteria.A design criterion is a guideline for how the language will be. We have valued the main criteria below in table 3.1 to create an overview of which criteria are the more important ones, and which we can agree on not being important.

Criterion	Importance	Description
Write-ability	Very high	How easy and how fast can a program be written
Readability	High	How easy is already written code to read
Reliability	Medium	The program will not behave unexpectedly
Orthogonality	Low	A relatively small set of primitive constructs can be combined in a relatively small number of ways
Uniformity	High	Similar features should look and behave similar
Maintainability	Very high	How fast can an error be found and corrected
Generality	Medium	No or few special cases, combine the special cases and construct a more general one
Extensibility	Low	Able to add new constructs to the language
Standard-ability	Medium	Able to transport the language to other computers
Implementability	High	Ensure a translator or interpreter can be written

Table 3.1: Design criteria [2]

Table 3.1 lists an overview of the design criteria we considered when designing the WAR language, and the associated level of priority.

Read- and Write-ability These criteria are very important since this is a scripting language for designing a virtual game, and non-programmer should be able to script their regiments, and their behaviour, and even make scripting fun. This is also our main focus and our goal to fulfil an easy written script, and of course easy to read as well.

Reliability This is rated medium, because if the interpreter or simulator behaves different than a user might expect it to do, the user will not trust his own script and then the write-ability will not be as fulfilled as we would like. The reliability is not something we have used a lot of strength to fulfil, but after the tests we have made, we think this is easily fulfilled.

Orthogonality When talking about orthogonality we mean to consider how much one component in a system affects the other. If orthogonality is important to the software, we want to be able to change components of the system without affecting the rest of the system too much. Since this is a very specific language with a very specific purpose this is not something that is valued very highly.

Uniformity This is valued high, because of the write-ability and readability which are valued high too. The way of assigning constants is done in the same way to all of the different variables scripters can define.

Maintainability This is valued very high, because we still have the 'game' in mind, and a new inexperienced scripter would give it less time than an experienced one, so debugging must be easy to perform. Since the simulation might fail with a faulty script, it is important to be able to quickly determine where the error lies, or else the language would not be very practical.

Generality This is valued medium, we think this is easily fulfilled, since the WAR language is a very simple language, so we have none of very few special cases.

Extensibility The ability to extend the language in itself. This is not very important for a purpose-directed language.

Standard-ability The interpreter is written in the language C♯ which runs in the virtual machine .NET just like Java do in JVM. If the computer can run the .NET Framework the interpreter can run on the machine as well. This means that the standardabiity is dependant on .NET implementations, such as *Mono* [3].

Implementability The implementability of the language is of high, because of this project's purpose. In this case it is more practical to use an interpreter, but it should also be possible to use a compiler.

The primary focus of this language, has been on *write-ability* and somewhat readability which is easily fulfilled when *write-ability* is the main goal. By having the *write-ability* as a very important goal we think that beginners in programming should be able to write their own scripts, and the *maintainability* is also rated very high, so the programmer of the language easy can debug the script, and find the errors that might have occurred and rapidly fix them. The *uniformity* of the language is of a high importance since this helps the

beginners to rapidly learn the language and understand how the different part of the language works.

3.1.1 Paradigm

The design criteria of the WAR language set up the main characteristics of the language, how advanced or how simple it should be e.g. the main focus area and what sort of paradigm it should be. We want the language to be simple, which we feel fit the imperative programming paradigm the best, because e.g. one of the strong point object-oriented programming paradigm is the ability to create powerful abstractions. Another paradigm choice could be the functional paradigm, but one of the strong point of this paradigm is for problem solving, which neither is important.

3.2 Tombstone diagram

A tombstone diagram shows how a interpretation or compilation is done [4]. Interpreter tombstone diagrams are made with a spade looking figure, with the grip in the top, and the spadestick in the bottom. In the top of the diagram one must put what program is developed, and what language the program is written in. The middle part one must put the interpreter and the machine which is should run on. The interpreter and the language which the program is written in, must be the same, since this is an interpreter. And the spadestick, or the bottom part states the machine which the program should run on.

In figure 3.1 one can see that from a program developed in the scripting language WAR, it will be interpret from the language WAR to the machine M and the interpreter will run on the same machine M. By the machine M it that is can run on every computer with the .NET framework.

Compiling the interpreter written in C♯ is done by the following:

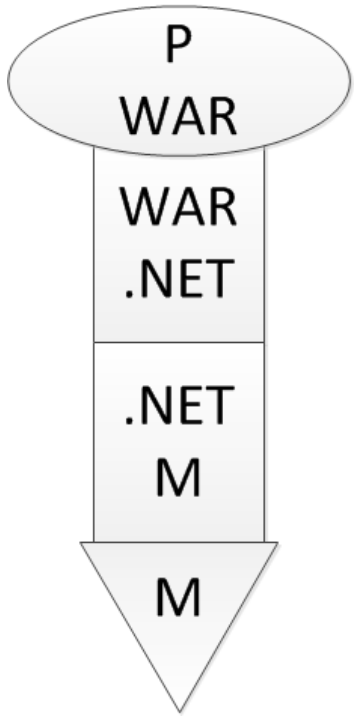


Figure 3.1: Tombstone Diagram for the WAR language
Interpreting a program P expressed in language WAR, which is written in the .NET framework. The WAR interpreter running on machine M

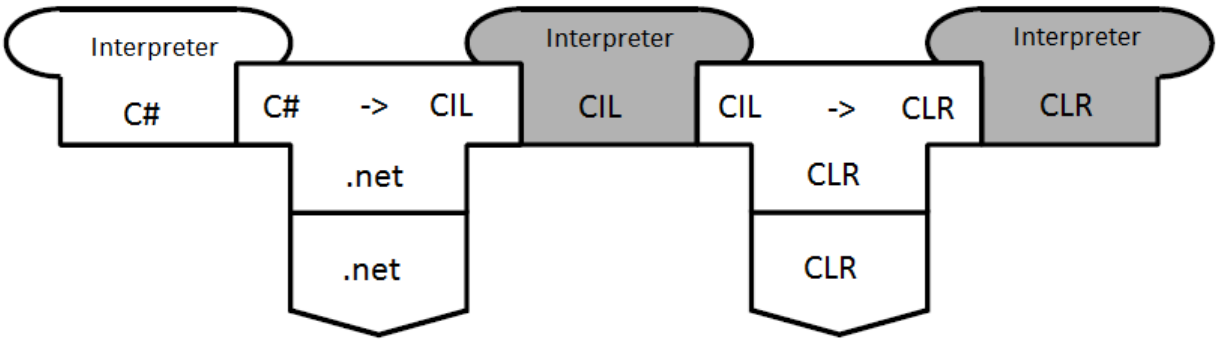


Figure 3.2: Tombstone Diagram of the interpreter

In figure 3.2 you can see how the .NET Framework is working by combining different languages and their matching compilers, and compiling all the languages into a new assembly like language called CIL (Common Intermediate Language) [3] which is very similar to the Java Virtual Machine. This is done with all the .NET compatible languages and then the platform specific CLR (Common Language Runtime) compiles the CIL to machine code for execution on the current machine.

Chapter 4

Language documentation

This chapter will show how to program scripts for the simulator. Scripts written for the simulator is written in the *WAR language*. When a term is defined we use <Term> to refer to the given term. When a multiple terms are used we illustrate it like this:

```
1 <Term-1>
2 ..
3 .
4 <Term-n>
```

Listing 4.1: Scripting

This illustrates that n terms can be written here.
The *WAR* Simulation framework consists of a relationship between the simulation engine of *WAR*, the interpreter for the script - which is effectively the implementation of the language itself, and the script files needed to describe the simulation.
To begin this simulation a *Config File* (See section 4.11) with the file extension *.cfg* and at least two *Team Files* (See section 4.10) with the file extension *.war*.

4.1 Identifiers

An *Identifier* is a word which starts with a letter and then follows a sequence of letters and/or digits. Identifiers are used for naming *Blocks* (See section 4.8) and for *Regiment Assignment* (See section 4.6).

4.2 Reserved Keywords

Table 4.1 shows reserved words of the WAR language. Reserved words cannot be used as identifiers.

Team	Regiment	Rules
Maxima	Config	Behaviour
if	else	while
Attack	SearchForEnemies	SearchForFriends
Distance	Size	Type
Melee	Movement	Position
Ranged	Range	Damage
Health	Movement	AttackSpeed
RegimentPosition		

Table 4.1: Reserved keywords

4.3 Data Types

WAR has the following data-types:

- Integer
- Boolean
- Position
- Regiment
- AttackType

To show the need for an element for a given data typeuse <DataType> is used. To show that an identifier is of a specific data type we write <Identifier:DataType>.

4.4 Unitstats

Unit stats define the stats of a regiment. The unitstats are used in both the config and team file. Here are the unitstats used in WAR

Size Sets the size of the regiment, expressed as an integer. If the size of the regiment is larger than one it will occupy multiple tiles.

```
1 Size = <Int>;
```

Listing 4.2: Size of a regiment

Position Sets the position of the regiment. If the size of the regiment is bigger than 1 them only 1 tile of the regiment touches the position. Usage:

```
1 RegimentPosition = Position(<Int>,<Int>);
```

Listing 4.3: Position of the regiment

Range sets from how many tiles away the regiment may attack another regiment.

```
1 Range = <Int>;
```

Listing 4.4: Range

Type defines what type of attack a regiment uses. The type can either be Melee or Ranged. Melee means that the regiment can only attack other regiments which it touches. Ranged means that the regiment can attack any regiment where the distance in number of tiles is less than the Range.

```
1 Type = <AttackType>;
```

Listing 4.5: Type of regiment

Damage defines how much Health an opposing regiment will lose if attacked.

```
1 Damage = <Int>;
```

Listing 4.6: Damage of the regiment

Movement means how many tiles the regiment can move.

```
1 Movement = <Int>;
```

Listing 4.7: Movement of the regiment

AttackSpeed defines how many attacks a regiment can perform in a round.

```
AttackSpeed = <Int>;
```

Listing 4.8: AttackSpeed of the regiment

An example of the above UnitStats.

```
Regiment PirateWarrior
{
        Size  = 10;
        Type = Melee;
        Damage = 8;
        Health = 4;
        Movement = 1;
        AttackSpeed = 1;
        RegimentPosition = Position(2,3);
        ...
}
```

Listing 4.9: Example: Using the UnitStats

To set up a regiment you need to name your regiment block. This is done by writing out Regiment and then your regiment name. Enclose your regiment in curly brackets, and assignment of the unitstats is now possible. In the case of missing unitstats assignments default standards from the configuration will be assigned.

4.5 UnitStatType

A *UnitStatType* is any of the UnitStats, except for Position, which is replaced by *Distance*. We use the UnitStatTypes in *Regiment Assignment* and *RegimentStat*.

Distance is an integer, which expresses how many tiles a given regiment is away from a certain point. The distance is expressed in the number of grid tiles two given regiments might be away from each other. It can be noted that distances are expressed in integers, and might be subject to truncation.

4.6 Regiment Assignment

You can assign an identifier to any given regiment - this allows you to use the identifier when attacking or moving to steer towards that particular regiment.

```
Regiment <Identifier > = <RegimentSearchFunction>
```

Listing 4.10: Regiment Assignment

```
Regiment PirateWarrior
{
...   //Here comes UnitStats
}
```

Listing 4.11: Example of a regiment assignment

4.7 RegimentStat

A *RegimentStat* is used to get the stats of a regiment. A RegimentStat can be used in Expressions.

```
<Identifier :Regiment>.<UnitStatType>
```

Listing 4.12: Regiment Stat

Please note that the identifier must be from a Regiment Assignment.

```
Regiment PirateArcher
{
        Size  = 10;
        Type = Ranged;
        Range = 2;
        Damage = 4;
        Health = 4;
        Movement = 1;
        AttackSpeed = 1;
        RegimentPosition = Position(3,2);
        Behaviour SilentMonkeysBehavoiur1
        {
                Regiment enemy = SearchForEnemies();
                if  (enemy.Distance <= Range)
                {
                        Attack(enemy);
```

```
17                          if (enemy.Distance < 2)
18                          {
19                                  MoveAway(enemy);
20                          }
21                  }
22                  else
23                  {
24                          MoveTowards(enemy);
25                  }
26          }
27  }
```

Listing 4.13: Example of an assignment in use

In listing 4.13 line 5 we see that the range is getting assigned the value of 2, and later in line 13 the range is getting used in an *if* statement, whether the enemy is in range the PirateArcher will attack.

4.8 Blocks

In WAR there are 5 different types of blocks: Regiment, Grid, Standards, Behaviour and Maxima blocks. These blocks are being used in both Config (See 4.10) and Team file (See 4.10).

4.8.1 Grid block

A grid block contains the *Height* and the *Width* of the map.

```
1  Grid <Identifier >
2  {
3          Width = <Int>;
4          Height = <Int>;
5  }
```

Listing 4.14: Grid Block

The width of the grid determines the size of the grid on the x-axis and the height determines the height y.

Behaviour Block The behaviour block controls how a regiment will behave in the simulation. To control regiments it is possible to use conditional statements such as *if-else if-else* and *while loops* and using unit functions.

Note that conditional statements and expressions can only be used in behaviour blocks.

if-else if-else Statement Example of the structure of an *if-else if-else* statement:

```
1  if (<Expression>)
2  {
3          //Code
4  }
5  else  if  (<Expression>)
6  {
7          //Code
8  }
9          else
10 {
11         //Code
12 }
```

Listing 4.15: if and else if statements

If the expression inside the parenthesis after if or else if returns true then the code inside the block will be executed. If none of the expressions return true the the code inside the else block will be executed. Please note that only the *if* block is required, there can be one to multiple *else if* blocks and only one else block which has to end the statement.

while Loop Structure of a *while* loop:

```
1  while(<Expression>)
2  {
3  //Code
4  }
```

Listing 4.16: While loop

If the expression returns true the code inside the block is executed. After execution the expression will be checked again.

Unit Functions Unit functions are used to move the regiment or attacking with the regiment.

```
MoveTowards(<Regiment>);
MoveAway(<Regiment>);
Attack(<Regiment>);
```

Listing 4.17: Unit functions

Standards block A standards block consist of unit stats and a behaviour block.

```
Standards
{
        <Unit-Stat-1>
        ..
        .
        <Unit-Stat-6>
        <Behaviour-Block>
}
```

Listing 4.18: Standards block

If a regiment in one of the team files are missing declarations or behaviour definitions, the missing values will be assigned with the values from the standards block.

4.8.2 Maxima Block

The Maxima block determines what the limits are for the unit stats. It also sets the maximum number of teams and regiments allowed. In the block the declarations UnitStat and MaximaStat are possible. MaximaStat is one of these:

```
Teams = <Int>;
Regiments = <Int>;
```

Listing 4.19: Maxima block

The Maxima Block is written as:

```
Maxima
{
        <UnitStat-1> or <MaximaStat-1>
        ..
        .
```

```
6         <UnitStat-6> or <MaximaStat-2>
7 }
```

Listing 4.20: Maxima stat

4.8.3 Regiment Block

A Regiment block contains unit stats and a behaviour block.

```
1 Regiment <Identifier >
2 {
3 <Unit-Stat-1>
4 ..
5 .
6 <Unit-Stat-6>
7 <Behaviour-Block>
8 }
```

Listing 4.21: Regiment block

4.9 RegimentSearch Functions

To get information about regiments on the map use *RegimentSearch Functions*. There are two RegimentSearch Functions in *WAR*.

```
1 SearchForEnemies(Parameters);
2 SearchForFriend(Parameters);
```

Listing 4.22: RegimentSearch Functions

Parameters is one or more expressions made with UnitStatType. Between two expressions there has to be a comma (,). Example:

```
1 SearchForEnemies(Distance > 5,Size == 200);
```

Listing 4.23: Functions with parameters

This will return all the enemy regiments, which are more than 5 tiles away and is of Size 200.

4.10 Team File

A team file is written this way:

```
Team <Identifier>
<Regiment−Block−1>
...
..
.
<Regiment−Block−n>
```

Listing 4.24: Regiment Assignment

4.11 Config File

The config file sets the rules for the simulation, it can be limitations on the regiments and deciding the size of the map. A config file is written this way:

```
Config
<Grid−Block>
<Standards−Block>
<Maxima−Block>
```

Listing 4.25: Config file

4.12 Code Example

In this section we demonstrate some examples of what the scripts in *WAR* looks like. In the following example it is shown how team and config files are written, assignment of regiment stats and the definition of their behaviour.

4.12.1 Code

Team file

```
Team NinjaMonkeys

Regiment SilentMonkeys
//Archers
```

```
5  {
6          Size = 200;
7          Type = Ranged;
8          Range = 120;
9          Damage = 2;
10         Movement = 30;
11         AttackSpeed = 1;
12
13         //Defining the behaviour of the regiment
14         Behaviour ArchSearchAndDestroy
15         {
16                 Regiment enemy = SearchForEnemies();
17                 if (enemy.Distance <= Range && enemy.Type == Melee)
18                 {
19                         Attack(enemy);
20                 }
21                 else if (enemy.Distance <= Range+Movement)
22                 {
23                         while(enemy.Distance < Range)
24                         {
25                                 MoveTowards(enemy);
26                         }
27                         Attack(enemy);
28                 }
29                 if (enemy.Distance <= Range)
30                 {
31                         MoveAway(enemy);
32                         if (enemy.Distance <= Range)
33                         {
34                                 Attack(enemy);
35                         }
36                 }
37         }
38 }
39
40 Regiment HeroMonkey
41 {
42         Type = Melee;
43         Movement = 50;
44         Damage = 30
45         AttackSpeed = 5;
46         Behaviour MonkeySeeMonkeyDo
47         {
```

```
48                  Regiment enemy = SearchForEnemies(Range >= 5);
49                  if (enemy.Health > 0)
50                  {
51                          Attack(enemy);
52                  }
53                  else
54                  {
55                          enemy = Search_for_Enemy();
56                          MoveTowards(enemy);
57                  }
58          }
59  }
```

Listing 4.26: Team file

Config file

```
1   Config
2
3   //Definition  of the grid
4   Grid Banana Island
5   {
6           Width = 512;
7           Height = 128;
8   }
9
10  Rules
11  {
12          Standards
13          {
14                  Size = 200;
15                  Type = Melee;
16                  Behaviour
17                  {
18
19                  }
20          }
21          Maxima
22          {
23                  Size = 2
24                  Regiments = 4;
25          }
26  }
```

Listing 4.27: Config file

Chapter 5

Language Design

In this chapter the syntax and the semantics of the WAR language will be defined. The syntax will be defined by a BNF, which will be extended to an EBNF - both can be found in the Appendix. The semantics (see section 5.2 will be defined by scope rules and type rules.

5.1 Syntax

The first step in writing our interpreter was to construct a BNF of our language. BNF is an acronym for *Backus Naur form*, which is a way of structuring the language in a formal way, to further ease implementation. A BNF consists of a set of *production rules* and each production rule consists of:

- Terminals: A *terminal* is a literal string or character as it appears in the code.
- Non-terminal: A *non-terminal* may consist of an arbitrary number of Non-terminals and Terminals, used for structuring the language.

Production Rule

A production rule has the form:

$$N ::= X \tag{5.1}$$

where X is a Non-terminal or a Terminal.

The elements in a production rule can be identified by: Terminals are written in **bold** and Non-terminals are just written in plain text.

Grouping of production rules

If we have two production rules on the form:

$$N ::= X \tag{5.2}$$

$$N ::= Y \tag{5.3}$$

we are allowed to make a transformation:

$$N ::= X|Y \tag{5.4}$$

Meaning N is X or Y

BNF

The BNF was made by looking at an example code - a code we wrote ourselves as an example of how we would like the final code to look. This example code can be seen in Listing 4.26. The BNF can be seen Appendix A.
Here is a production rule from the BNF:

```
Identifier  ::=  Letter
             |   Identifier Digit
             |   Identifier Letter
```

5.1.1 EBNF

EBNF is an acronym for *Extendend Backus Naur Form* and is, as the name refers to an extension of BNF. The EBNF allows us to use regular expressions to express production rules. This makes our rule set more compact and easier to implement.

Regular expressions

A regular expression is a convenient way of expressing strings. We use the following regular expressions:

- *(star): States that the terminal or non-terminal must be used 0 or more times.
- Parentheses: Used for grouping of non-terminals and terminals.
- ϵ: Represents the empty string.

Please note that these regular expressions can only be used on the right side of the production rules.

A transformation of a BNF is done systematically by applying *left factorization* , *elimination of left recursion* and *substitution of non-terminals* .

Left factorization

If we have a production rule on the form:

T ::= AB|AC

We can do a left factorization:

T ::= A(B|C)

This can be done because T will always start with an A and end with either a B or C.

Elimination of left recursion

If we have a production rule on the form:

T ::= A|TB

We can do an elimination of left recursion:

T ::= A(B)*

To understand how we can do this look at the first production rule. To terminate the production rule we need to put an A, so T will always consist of an A. When we are not selecting an A(, and terminating) we are only making B's, which is the same as B*.

Substitution of non-terminals

If we have two production rules on the form:

T ::= U | AUC
U ::= **ab** | **ba**

We can do a substitution of the non-terminal U for every production rule. This means that we substitute any occurrence of U in a production rule with what is on the right hand side of U like this:

T ::= (**ab** | **ba**) | A (**ab** | **ba**)B

5.1.2 The EBNF of the WAR Language

To obtain the EBNF we applied left factorization, elimination of left recursion and substitution of non-terminals on the BNF and can be seen in Appendix B. To illustrate how we transformed our BNF to an EBNF we will look at an example.

Transformation of production rule of Identifier

Here is the production rule of Identifer:

```
Identifier  ::=  Letter
             |   Identifier Digit
             |   Identifier Letter
```

We can first apply Left factorization giving us the new production rule:

```
Identifier  ::=  Letter
             |   Identifier (Digit | Letter)
```

Then we can apply Elimination of left recursion giving us the new production rule:

```
Identifier  ::=  Letter(Digit | Letter)*
```

5.2 Semantics

This section will define the scope rules and type rules of WAR.

5.2.1 Scope rules

Scope rules dictate the accessibility of identifiers from one scope to another. A scope placed inside another scope is called a *nested scope*. A *scope level* denotes how nested a given scope is, e.g. identifiers at scope level 0 is outside any scope, identifiers at scope level 1 is inside one scope etc.

```
//Scope level 0
Scope
{
        //Scope level 1
        Scope
        {
```

```
                //Scope level 2
        }
}
```

WAR has a nested block-structure which means that it allows more than 2 levels of nested scoping. The following scope rules applies for nested block-structures [5].

- No identifier may be declared more than once within the same block (at the same level)
- For any applied occurrence there must be a corresponding occurrence, either within the same block or block which is higher up in the nesting.

5.2.2 Type rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language [6]. The following types are present in *WAR*:

- Integer
- Boolean
- Position
- Regiment
- AttackType

Type rules of *WAR*

This type rule is used in expressions or when assigning a constant to a integer value.

Type Rule 1 $I1$ *o* $I2$ *is type correct and of type Integer*
if $I1$ *and* $I2$ *are type correct and of type Integer*
and o is an operator where $o \in \{-, +, /, *\}$

This type rule is used for checking if conditional statements are type correct.

Type Rule 2 $C(E)\{D\}$ *is type correct*
if E *is of type Boolean and* D *is type correct*
and C *is a control structure where* $C \in \{else\ if, if, while\}$

This type rule is used for assignments.

Type Rule 3 $ID = A$ *is type correct*
if ID *and* A *are type correct and of the same type.*

This type rule is used for *Position(x,y)*, which is used when assigning a position to a regiment.

Type Rule 4 $Position(I1, I2)$ *is type correct and of type Position*
if $I1$ *and* $I2$ *are type correct and of type Integer*

This regiment is for boolean binary expressions.

Type Rule 5 *E*1 *o E*2 *is type correct and of type Boolean*
*if E*1 *and E*2 *are type correct and of type Integer*
and o is an operator where $o \in \{==, >=, <=, <, >\}$

All these type rules makes the scripting more secure, because if a argument or types of returned values is not evaluated to type correct using these type rules then an error is present.

These definitions are the end of the section on semantics of WAR. With the syntax and semantics written a documentation for the language can be written, which will be the topic of the next chapter.

Chapter 6

Implementation

This chapter will describe how we implemented the simulator. The simulator consists of multiple parts:

1. Parsing
2. Contextual Analyzing
3. Simulation

The following figure shows the process from script files to output on the screen.

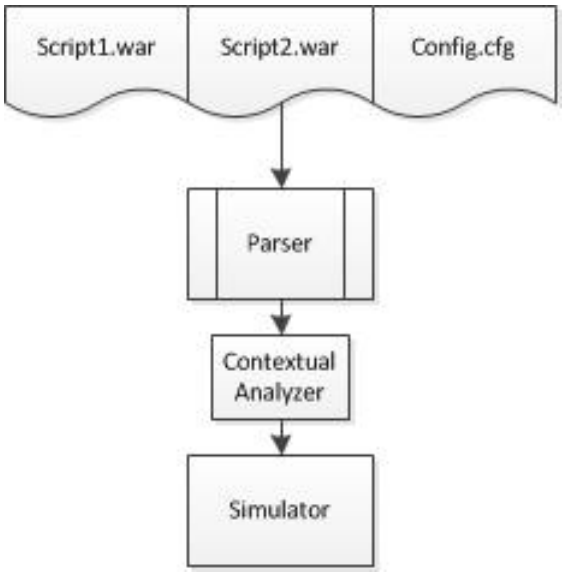


Figure 6.1: From scripts to simulation

6.1 Parser construction

The job of a parser is to check if a program is syntactically correct and to determine the structure of the program (Here we constructed an *abstract syntax tree* - see section 6.1.3). A parser can only read *tokens*, which is terminals as seen in the EBNF. So before we are able to parse we need to convert the program code from characters to tokens, this is done by using a scanner, that will be explained next. Beside having a scanner it is also important to choose a strategy for parsing, which we will look at in 6.1.2.

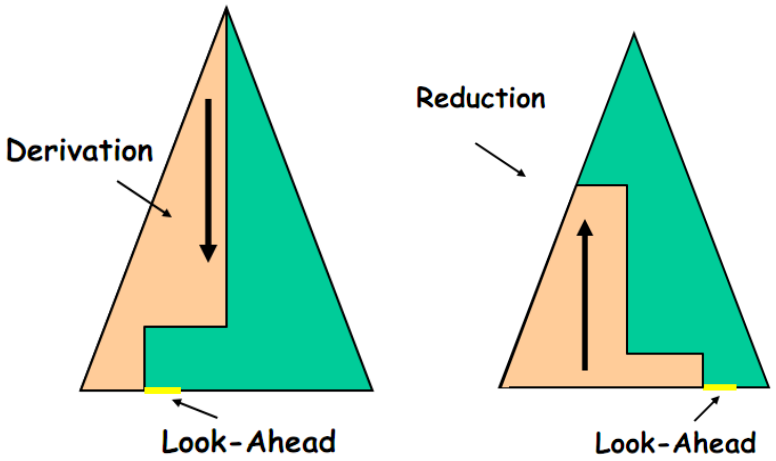
6.1.1 Scanner

The scanner, also called a lexer, is used to convert text, in the text files, to tokens. It reads a text file as a stream of characters and tries to identify which token it is currently reading. To identify tokens a table with the correct spellings of tokens is used.

6.1.2 Parse Strategies

A parse strategy is, beside checking that the code is syntactically correct a way to build our AST. To determine which production rule the parser is currently reading, a parser is using *lookahead*, which is a number representing how many tokens we have to look at, at a time to determine the production rule. There are two common strategies for parsing code, called *Bottom-up Parsing* and *Top-Down Parsing*, that are explained next [7].

Bottom-up Parsing This way of parsing takes simple structures and combines them into more complex structures. This type of parsing is commonly called *LR parsing* because we read the text from the left and reduces to the right. A LR parser with k lookahead is called a LR(k) parser.



(a) Top down (b) Bottom up

Figure 6.2: Two different parse strategies:Top-down(LL) on the left and bottom-up(LR) on the right

Top-down Parsing Top-down parsing figure 6.2a starts from the complex structures, and breaks them down into smaller parts. This type of parsing is called *LL parsing* because we read the text from the left and make derivations to the left. A LL parser with k lookahead is called a LL(k) parser.

Strengths and weaknesses The most commonly used parsing strategy of the two is LR parsing, the reason for this is because it is faster than LL parsing. In this project, however we have chosen to make an LL parser because it is easy to implement, furthermore speed is not important to us. The type of LL parser we made is called a recursive descent parser, which can parse LL(1) grammars.

6.1.3 Abstract Syntax Tree

The structure of the parsed program is stored as an AST. The tree structure is called abstract because it does not contain the concrete structure of the EBNF, but rather an abstract representation of the source code. Because of this there is not a straightforward way to construct such a tree. We constructed the AST-structure by making a class for each of the node we wanted represented i.e. we have classes for expressions, declarations and different datatypes.

Compared to the EBNF the AST is more specific i.e. an *expression* is not simply an *expression*, but it can be an *IntegerExpression* or a *RegimentStatExpression*.

To make parsing easier we use polymorphism so that all variants of an expression will inherit from the class *expression* and likewise for similar constructs. Furthermore all classes which are not a subtype of another, inherit from a class called *AST*, the reason for this will be explained in section 6.4, which is the only place we directly use it.

Small example Here is a small example of how an AST could look like when parsing our code.

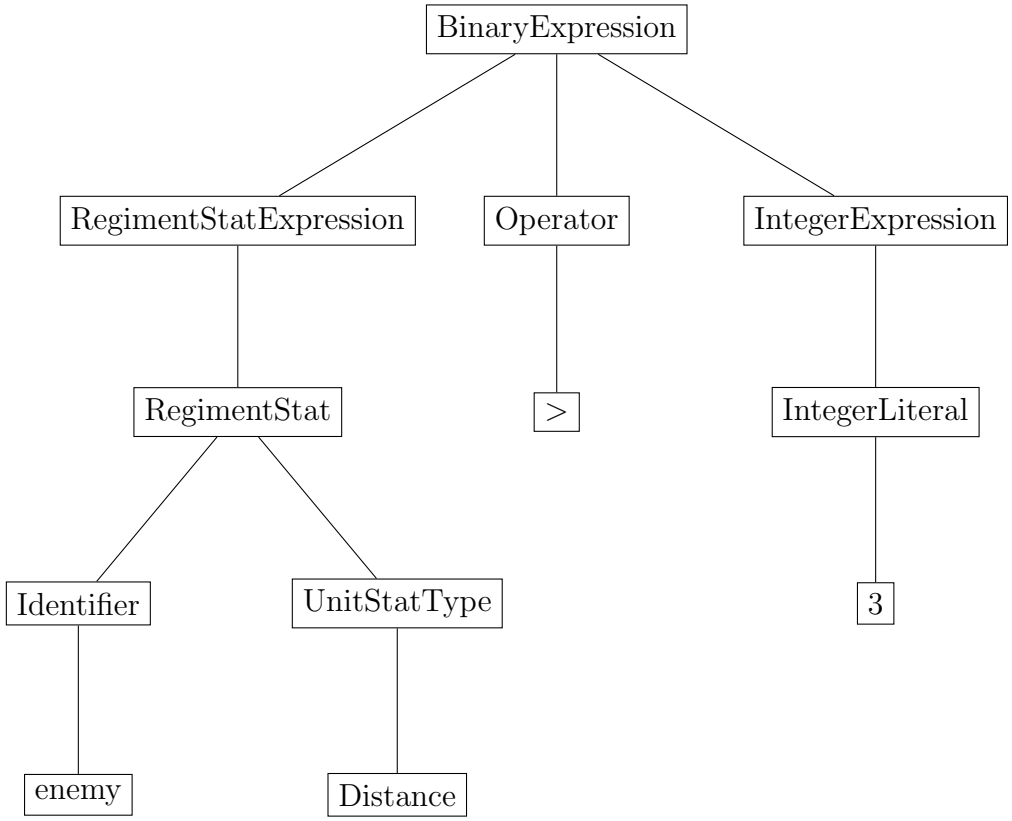


Figure 6.3: AST of a binary expression

The tree shows an AST of a *BinaryExpression*. The tree is equivalent to the code:

```
enemy.Distance > 3
```

where *enemy* is an identifier, which have been assigned to a regiment. How we constructed such a tree in the code is explained in 6.1.4.

6.1.4 Recursive descent parser

A recursive descent parser is a LL parser which works by recursively going through the program code. The parser we made is a LL(1) parser. To construct this type of parser the following is required:

1. EBNF
2. A scanner for reading chars and identifying them as terminals.
3. A parse method for each non-terminal in the EBNF.
4. A method or multiple methods that can accept terminals.
5. In each of the parse methods accept tokens and call other parse methods, as required.

Accepting terminals

The parser uses an accept method to check if the current token is the token that was expected, if not an error is reported. The following example shows the *Accept method* from our Parser.

```
1  private void Accept(Token.TokenType expectedType)
2  {
3          if (currentToken.type == expectedType)
4      {
5                  currentToken = scanner.Scan();
6      }
7      else
8      {
9          errorReporter.ReportParserError("Expected " +
10                 expectedType + ", but got " + currentToken.type, currentToken);
11     }
12 }
```

Listing 6.1: Accept method for accepting tokens

This method checks if the expected token matches the current token, and then retrieves the next token from the scanner, if it was a match. If there is a mismatch, an error will be reported.

Parse methods

The parse methods perform in a mutually recursive way, where a parse methods representing the higher level nodes in an AST calls parse methods representing its children. When a lower level node is done parsing it returns this node to a higher level node. This way an AST will be constructed.
Here is a parse method from our code:

```
1  private  SingleCommand ParseRegimentDeclarationCommand()
2  {
3      Accept(Token.TokenType.Regiment);
4          Identifier   i  = ParseIdentifier  ();
5          Accept(Token.TokenType.Assignment);
6      RegimentSearch rs = ParseRegimentSearch();
7      Accept(Token.TokenType.SemiColon);
8          RegimentDeclaration rd  = new RegimentDeclaration(i, rs );
9          return  new RegimentDeclarationCommand(rd,previousTokenPosition);
10 }
```

Listing 6.2: Parse method for RegimentDeclarationCommand

The method is for parsing *RegimentDeclarationCommand*s, which consists of a *RegimentDeclaration node*. The method starts by trying to accept the token *Regiment* because a *RegimentDeclaration node* must start with this. After this it expects there to be an *Identifier node*, which it tries to parse. After the rest of the accepting and parsing the new *RegimentDeclarationCommand node* is returned.

6.2 Contextual Analyzer

This section will explain how type checking and scope checking is performed on the AST which was constructed from the parser. To do this, a way of traversing the tree is needed, in which we chose the *visitor pattern*.

6.2.1 Visitor Pattern

The Visitor Pattern is a design pattern used to traverse tree structures. The design pattern is a smart way of separating traversal of the tree and the code that has to be executed at the tree nodes.
This is done by making a *Visitor interface* that has a *Visit method* for each of the nodes that can be visited (See figure 6.4). Classes that wish to traverse the tree can then implement the interface and its methods.

```
 8    public interface Visitor
 9    {
10        #region Blocks
11        Object VisitBehaviourBlock(BehaviourBlock ast, Object obj);
12
13        Object VisitGridBlock(GridBlock ast, Object obj);
14        Object VisitMaximaBlock(MaximaBlock ast, Object obj);
15        Object VisitRegimentBlock(RegimentBlock ast, Object obj);
16        Object VisitRulesBlock(RulesBlock ast, Object obj);
17        Object VisitStandardsBlock(StandardsBlock ast, Object obj);
18        #endregion
19
20        #region Control Structures
21        Object VisitIfCommand(IfCommand ast, Object obj);
22        Object VisitElseIfCommand(ElseIfCommand ast, Object obj);
23        Object VisitWhileCommand(WhileCommand ast, Object obj);
24        #endregion
25
26        #region Expressions
27        Object VisitBinaryExpression(BinaryExpression ast, Object obj);
28        Object VisitIntegerExpression(IntegerExpression ast, Object obj);
29        Object VisitRegimentStatExpression(RegimentStatExpression ast, Object obj);
30        Object VisitUnaryExpression(UnaryExpression ast, Object obj);
31        Object VisitUnitStatVNameExpression(UnitStatVNameExpression ast, Object obj);
32        #endregion
```

Figure 6.4: Screenshot of some of the Visitor interface

The class nodes all have a *Visit method*, which is overridden from the AST class, that calls the method from the visitor class that corresponds to the node (See figure 6.3).

```
1  class TreeNode:AST
2  {
3          public Node a,b;
4          public TreeNode(Node a, Node b)
5          {
6                  this.a = a;
7                  this.b = b;
8          }
9          public Visit (Visitor v, object arg)
10         {
11                 //Calls the right visit method from the visitor class
12                 v.VisitTreeNode(this, arg);
13         }
14 }
```

Listing 6.3: Example of a node class with a Visit method

6.2.2 Checker class

The *Checker class* implements the Visitor interface seen in 6.4. In its visit methods it checks for type rules and scope rules. To keep track of the current scope level and of declarations we use an *Identification table*.

Identification table

The identification table is a class named *IdentificationTable*(see listing 6.4), which contains a list with entries that holds the information declarations made and an integer *level* that represents the current scope level. For adding and retrieving entries *EnterEntry* and *RetrieveEntry* are used. For opening and closing the current scope *Open* and *Close*. Open increases the *level* and Close removes all the declarations made on the current level and *level* decreases it by one.

```
1  class IdentificationTable
2  {
3          private int level =0;
4          private List <IdEntry> entries = new List<IdEntry>();
5
6          public void Open(){}
7          public void Close(){}
8
9          public bool EnterEntry(Declaration declaration , string id ){}
10         public Declaration RetrieveEntry (string id ){}
11 }
```

Listing 6.4: The IdentificationTable class

Scope Rules checking

For opening and closing the current scope we use the *IdentificationTable class* just described. When we are entering a new scope the *Open method* is called and when we are leaving the scope the *Close method* is called. The *VisitRegimentBlock method* (see listing 6.5 below) which needs to open and close scopes:

```
1  public Object VisitRegimentBlock(RegimentBlock ast, Object obj)
2  {
3          //Opening scope
4          idTable. Open();
```

```
5
6           //Visits children
7       ast.bn.Visit (this , null );
8       ast.usds.ForEach(x => x.Visit(this , null ));
9       ast.bb.Visit (this , null );
10
11      //Closing scope
12      idTable.Close();
13
14          return null ;
15  }
```

Listing 6.5: The IdentificationTable class

Type checking

When the checker visits a node where type checking is needed (e.g. RegimentStat), the checker visits the nodes in need of identification. Here the datatype of the node is retrieved from the identification table. If the datatype matches the expected datatype, the analysis proceeds, otherwise an error is reported then the analysis continues.

```
1   public Object VisitRegimentStat(RegimentStat ast, Object obj)
2   {
3           ast.i.Visit (this , null );
4           Declaration declaration = idTable.RetrieveEntry(ast.i.spelling );
5           if (declaration == null)
6           {
7           reporter.ReportCheckerError("Regiment % wasn't declared",
8                       ast.i.spelling , ast.i.position );
9           }
10          else if (ast.i.type != DataType.Regiment)
11          {
12          reporter.ReportCheckerError("% was not of type Regiment",
13                      ast.i.spelling , ast.i.position );
14          }
15                  ast.ust.Visit (this , null );
16                  return null ;
17      }
18  }
```

Listing 6.6: Type checking of RegimentStat

This code snippets shows type checking for *RegimentStat*. The production rule of RegimentStat is:

$$RegimentStat ::= Identifier.UnitStatType \tag{6.1}$$

This means that we have to check if the identifier has been declared and we have to check if the identifier is of type Regiment. These two checks can be seen above in the *if statement* and the *if else statement*.

The type checking also serves another purpose called decoration. When a variable/constant is declared the data type and the value of the variable/constant is saved. Each time the checker visits a node which references a declared variable/constant it decorates it with its data type and value. When the type rules checking has finished the AST that is traversed is fully decorated.

6.3 Simulator

This section will explain how the simulator is implemented, the diagram below 6.5 shows the process.

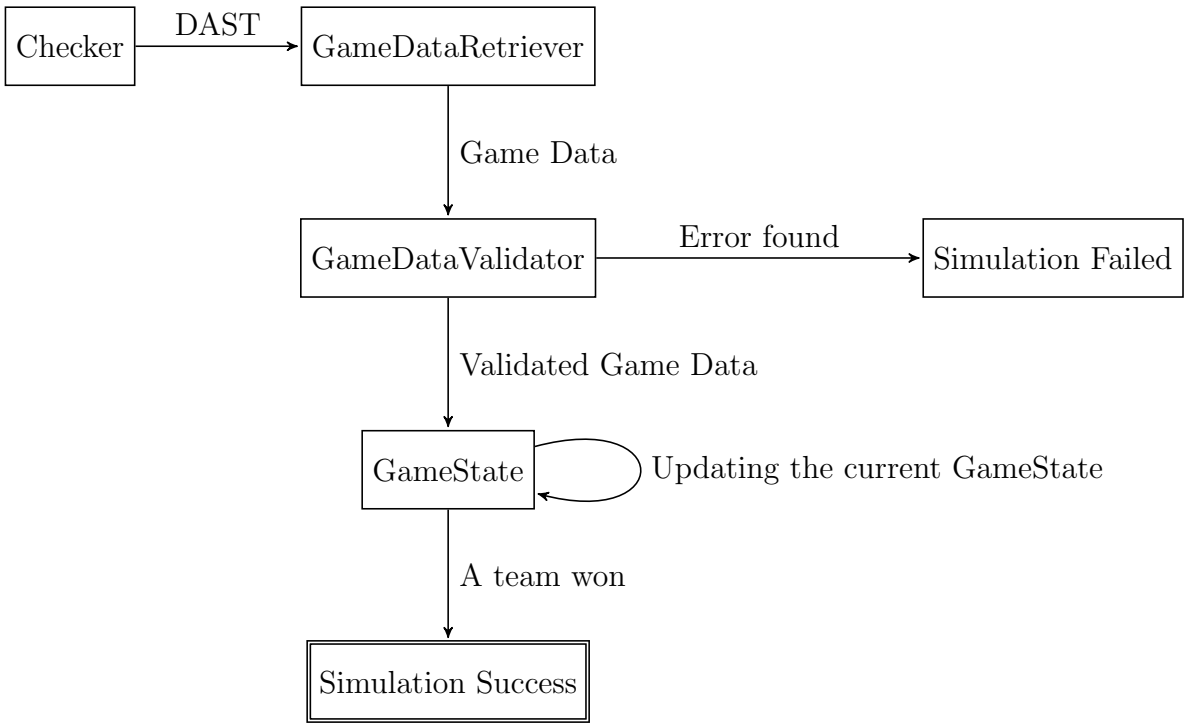


Figure 6.5: Diagram showing the simulation

The following subsections will explain what happens at the phases, but first an explanation of how the game world is emulated.

6.3.1 Game Classes

To emulate the game we have to create objects, which represent what is seen in the game such as regiments, the grid and the teams. This was simply done by making a class called *Regiment*, *Grid*, *Team* and *Tile*. Team contains a list of regiments, Regiment have methods for the three *UnitFunctions*: *Attack*, *MoveTowards* and *MoveAway* (see section 2.2.3), Tile has a boolean to indicate that it is occupied and *Grid* contains a 2-dimensional array of *Tile*s. We call the game environment at a specific point in the simulation for a *GameState*.

6.3.2 GameDataRetriever

The job of the *GameDataRetriever* is to retrieve the game data from the *DASTs*. The class implements the visitor pattern (see section 6.2.1) to traverse the decorated abstract syntax tree and retrieve the data. It starts by visiting each of the *Team file*s *DASTs* and when it visits the *Regiment node* it instantiates a new Regiment object, with the *UnitStat*s is contains, after this it is added to a Team object. When game data for every team have been retrieved the *Config file DAST* is visited. The data of the *Standards block node* is simply saved as a Regiment Object, because it is so similar to Regiment retrieving. The *Grid* data is retrieved from the Grid block. The *Maxima stats* is saved to a Maxima object, which contains a field for any of available Maxima stats. After all data is retrieved it is passed to the *GameDataRetriever*

6.3.3 GameDataValidator

After all the data has been retrieved from the DAST we have to validate whether or not the entered the data is correct. By correct we mean that every regiment has all the *UnitStats* or that the *UnitStat* of a regiment does not exceed the *MaximaStat*s. The class *GameDataValidator* is started by using the Team objects, standards regiment and maxima object passed from the *GameDataRetriever*. The process of the validation:

1. Check if there are more teams than allowed according to the config file

2. Check if the teams have more regiments than allowed according to the config file

3. Check if every regiment does not exceed Maxima Stats

4. Check if every regiment is missing UnitStats, if so the UnitStat from the standards regiment is given

5. Check if any of the regiments have the same positions

If any of the steps fail, an error is given and the simulation will not start. If it succeeds it returns a new *GameState* containing the teams and the grid.

6.3.4 Simulator

The simulator class starts by using the *GameDataRetriever* and *GameDataValidator* classes. The *GameDataValidator* returns the first *GameState* if it succeeded. The simulator reads and modifies a *GameState* in every round of the game. This is done by iterating through all regiments from the teams and interpreting their *Behaviour DAST*. This main loop is showed here:

```
1  foreach (Regiment regiment in regimentTurnOrder)
2  {
3          if (regiment.currentSize > 0)
4          {
5                  currentGameState =
6  behaviourInterpreter .InterpreteBehaviour (regiment, currentGameState);
7          }
8          else
9          {
10         currentGameState.teams[regiment.team].regiments.Remove(regiment);
11                 if (currentGameState.teams[regiment.team].regiments.Count <= 0)
12                 {
13                         teamsLeft--;
14                 }
15         }
16 }
```

Listing 6.7: Main loop of the simulation

behaviourInterpreter is an object of the class *BehaviourInterpreter*, which is able to interpret *Behaviour DAST*s.

6.3.5 BehaviourInterpreter

This class is the third and last class that implements the *Visitor interface* (see section 6.2.1), which it uses to interpret the DAST, this type of interpretation is called *Recursive Descent Interpretation*. The interpretation starts with the call of the method called *InterpreteBehaviour*, which can be seen below in 6.8.

```
1  public GameState InterpreteBehaviour(Regiment regiment, GameState gameState)
2  {
3          regimentAssignments.Clear();
4          currentGameState = gameState;
5          currentRegiment = regiment;
6          currentRegiment.currentMovement = currentRegiment.movement;
7          currentRegiment.hasAttacked = false;
8          currentRegiment.behaviour.Visit (this, null);
9          return currentGameState;
10 }
```

Listing 6.8: The method InterpreteBehaviour

The methods starts with resetting the list regimentAssignments, which is a list of identifiers that have been assigned to regiments. To retrieve a regiment from this list the method GetRegiment can be used, which takes a string as an argument. After this the method resets the current regiments movement and ability to attack and visits the *Behaviour DAST* of the regiment. When class visits a node which needs interpretation, e.g. the *UnitFunction node* they will be interpreted. To show how this is actually done we will now show interpretation of conditional statements, *UnitFunctions* and *RegimentSearch*.

Conditional Statements

Conditional statements consists of code that has to be executed and a condition, in our DAST the code that needs execution is a *SingleCommand node* and the condition is an *Expression node*. The code example 6.9 shows how a visit of the *IfCommand node* looks like.

```
1  public Object VisitIfCommand(IfCommand ast, Object obj)
2  {
3          //Retrieves the boolean value of the expression
4          BoolValue b = (BoolValue)ast.e.Visit (this, null);
5
6          //Checks if the expression evaluates to true
7          if (b.b)
```

```
8           {
9                   ast.sc1.Visit (this, null);
10          }
11          //Checks if there is any else if commands and visits them if there is
12          else if (ast.eifc.Count > 0) { ast.eifc.ForEach(x => x.Visit(this, null)); }
13          //Checks if there is any else command and visits it if there is
14          else if (ast.sc2 != null) { ast.sc2.Visit (this, null); }
15          return null;
16  }
```

Listing 6.9: VisitIfCommand method from the class BehaviourInterpreter

The evaluation of the condition is done by visiting the expression node, which is tied to the current conditional statement. The simple expressions (IntegerExpression or RegimentStatExpression) will simply return their value, but the *BinaryExpression* has to evaluate what the expression is by using a method called *CheckBinaryExpression*, which returns the value of the expression.

```
1   if (v1 is IntValue)
2   {
3           int i1 = ((IntValue)v1).i;
4           int i2 = ((IntValue)v2).i;
5           BoolValue b = new BoolValue();
6           IntValue i = new IntValue();
7           switch (op)
8           {
9           case "<": if (i1 < i2) { b.b = true; } else { b.b = false; } return b;
10          case ">": if (i1 > i2) { b.b = true; } else { b.b = false; } return b;
11          case "==": if (i1 == i2) { b.b = true; } else { b.b = false; } return b;
12          case ">=": if (i1 >= i2) { b.b = true; } else { b.b = false; } return b;
13          case "<=": if (i1 <= i2) { b.b = true; } else { b.b = false; } return b;
14          case "!=": if (i1 != i2) { b.b = true; } else { b.b = false; } return b;
15
16          case "+": i.i = i1 + i2; return i;
17          case "-": i.i = i1 - i2; return i;
18          case "/": i.i = i1 / i2; return i;
19          case "*": i.i = i1 * i2; return i;
20          }
21  }
```

Listing 6.10: Code snippet from CheckBinaryExpression

The code snippet 6.10 shows how the method handles binary expressions where both the expressions are integers, here *op* is a string which represents

the operator of the binary expression. CheckBinaryExpression has similar handling for expressions with type expressions.

UnitFunctions

UnitFunctions are the three actions (see section 2.2.3) that a regiment can perform.

```
1  public Object VisitUnitFunction(UnitFunction ast, Object obj)
2  {
3          //spelling of the regiment identifier
4          string spelling = (string)ast.i.Visit(this, null);
5
6          //spelling of the function name
7          string functionName = (string)ast.ufn.Visit(this, null);
8
9          //Gets the regiment by using the method GetRegiment
10         Regiment regiment = GetRegiment(spelling);
11         if (regiment != null)
12         {
13 //Checks which UnitFunction we are trying to call and calls it from the regiment class
14                 switch (functionName)
15                 {
16                 case "Attack": currentRegiment.Attack(regiment); break;
17                 case "MoveTowards": currentRegiment.MoveTowards(regiment); break;
18                 case "MoveAway": currentRegiment.MoveAway(regiment); break;
19                 }
20         }
21         return null;
22 }
```

Listing 6.11: VisitUnitFunction from the class

The code snippet 6.11 shows how a visit of the *UnitFunction node* looks like. First the identifier representing the regiment is retrieved and then the UnitFunction type is retrieved. The identifier, which was retrieved is then used in the function GetRegiment that returns a regiment that matches the identifier. Using a switch case the correct action of the current regiment is performed.

RegimentSearch

The *RegimentSearch methods* is used for finding either enemy or friendly regiments on the grid (see usage at section 4.9). We will here explain how

the the visit to a *RegimentSearch node* appears in multiple code snippets.

```
1  public Object VisitRegimentSearch(RegimentSearch ast, Object obj)
2  {
3          //Retrieves the type of regimentsearch we are doing
4          string regimentSearchSpelling = (string)ast.rsn.Visit(this, null);
5
6          //A list for storing all the regiments we find
7          List<Regiment> regimentsFound = new List<Regiment>();
8          if (regimentSearchSpelling == "SearchForFriends")
9          {
10                 //Looks for regiments which are friends
11         regimentsFound = currentGameState.teams[currentRegiment.team].regiments;
12         }
13         else
14         {
15                 //Finds all the regiments which is not friendly
16                 foreach (Team team in currentGameState.teams)
17                 {
18                         if (team.number != currentRegiment.team)
19                         {
20                                 regimentsFound.AddRange(team.regiments);
21                         }
22                 }
23         }
```

Listing 6.12: 1. part of visit of RegimentSearch node in the class BehaviourInterpreter

The first part of the method tries to find out which regiments the search is for where *regimentsFound* holds this collection. If we are searching for friends then only the regiments matching the team of the current regiment will be added to *regimentsFound*. Else we add all the regiments, which is not matching the current regiments team.

```
1          if (ast.p != null)
2          {
3                  foreach (Parameter p in ast.p)
4                  {
5                          //Retrieves the parameter from the ast
6                          Parameter parameter = (Parameter)p.Visit(this, null);
7
8                  //Gets all the regiments which matches the parameter
9                  regimentsFound = GetRegiments(regimentsFound, parameter);
10                         if (regimentsFound.Count == 0) { break; }
```

```
11                  }
12          }
13          if (regimentsFound.Count == 0)
14          {
15                  return null;
16          }
17          return GetClosestRegiment(regimentsFound);
18  }
```

Listing 6.13: 1. part of visit of RegimentSearch node in the class BehaviourInterpreter

This part of the method is about applying the parameters passed to the *RegimentSearch function* on the list of regiments we just found. The parameters are applied one of a time using the method *GetRegiments*, which returns the regiments for which the parameters are true. If no regiments was found that fits the parameters the method returns null, otherwise we get the regiment which is closest to the current regiment using the method *GetClosestRegiment*.

Chapter 7

Use cases and Test Scenarios

This chapter will show use cases showing how the simulator can be used.

7.1 Use case 1

2 Versus 2

In this first usecase, we demonstrate a simple battle between two teams, each with two regiments. For this simple battle, a 4×4 grid is provided by the config file, as can be seen by the Width and Height definitions. The additional standard-values and maxima-values can also be seen in the configurations file. As mentioned above, each team use the same two regiments - the only difference is the RegimentPosition, i.e. the start position of the regiments.
The first regiment of PirateArchers have the following stats; The regiment consist of 10 ranged units, with 2 range, dealing 4 damage with an attackspeed of 1. They have 4 health points an can perform 1 move action per turn. Their behaviour is quite simple; if the enemy is within range, they attack the enemy. if the enemy is within 2 tiles of the regiment, the regiment retreats. If the regiment is not within range of an enemy, the regiment moves towards the nearest enemy.
The other regiment consist of 10 melee warriors with 8 damage and 1 attack speed. These warriors have 8 health points and can perform 1 movement action per turn. This regiments behaviour is also quite simple; If able, they attack the enemy - otherwise they move towards the enemy and, if possible, attack the enemy.
The fight was quite even, but in the end, the blue Ninjas won. Three screenshots from the battle can be seen in figure 7.1, 7.2 and 7.3.

Config

```
Grid BananaIsland
{
        Width = 4;
        Height = 4;
}
Rules
{
        Standards
        {
                Size = 200;
                Type = Melee;
                Range = 90;
                Damage = 2;
                Movement = 10;
                Health = 50;
                RegimentPosition = Position(0,0);
                Behaviour Agressive
                {
                        Regiment enemy = SearchForEnemies(Size>0);
                        Attack(enemy);
                }
        }
        Maxima
        {
                Size = 500;
                Regiments = 4;
                Range = 100;
                AttackSpeed=1;
                Damage=100;
                Movement = 20;
                Health = 1000;
        }
}

Regiment PirateArcher
{
        Size = 10;
        Type = Ranged;
        Range = 2;
        Damage = 4;
```

```
        Health = 4;
        Movement = 1;
        AttackSpeed = 1;
        RegimentPosition = Position(4,5);
        Behaviour ArcherBehaviour
        {
                Regiment enemy = SearchForEnemies();
                if (enemy.Distance <= Range)
                {
                        Attack(enemy);
                        if(enemy.Distance < 2)
                        {
                                MoveAway(enemy);
                        }
                }
                else
                {
                        MoveTowards(enemy);
                }
        }
}
Regiment PirateWarrior
{
        Size = 10;
        Type = Melee;
        Damage = 8;
        Health = 4;
        Movement = 1;
        AttackSpeed = 1;
        RegimentPosition = Position(2,5);
        Behaviour WarriorBehaviour
        {
                Regiment enemy = SearchForEnemies();
                if (enemy.Distance <= Range)
                {
                        Attack(enemy);
                }
                else
                {
                        MoveTowards(enemy);
                        Attack(enemy);
```

```
                    }
            }
}
```

Screenshots from use case 1

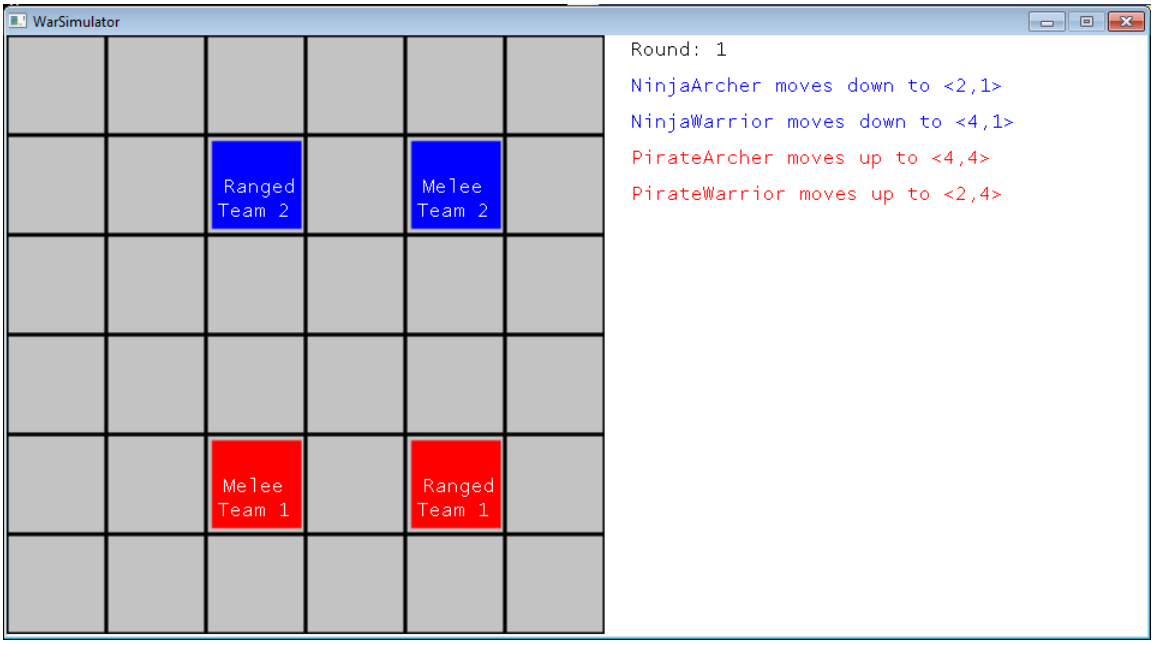


Figure 7.1: Both teams take up an aggressive position in front of the opposing teams army.

Figure 7.2: The Warriors chase the archers, while the archers desperately try to escape

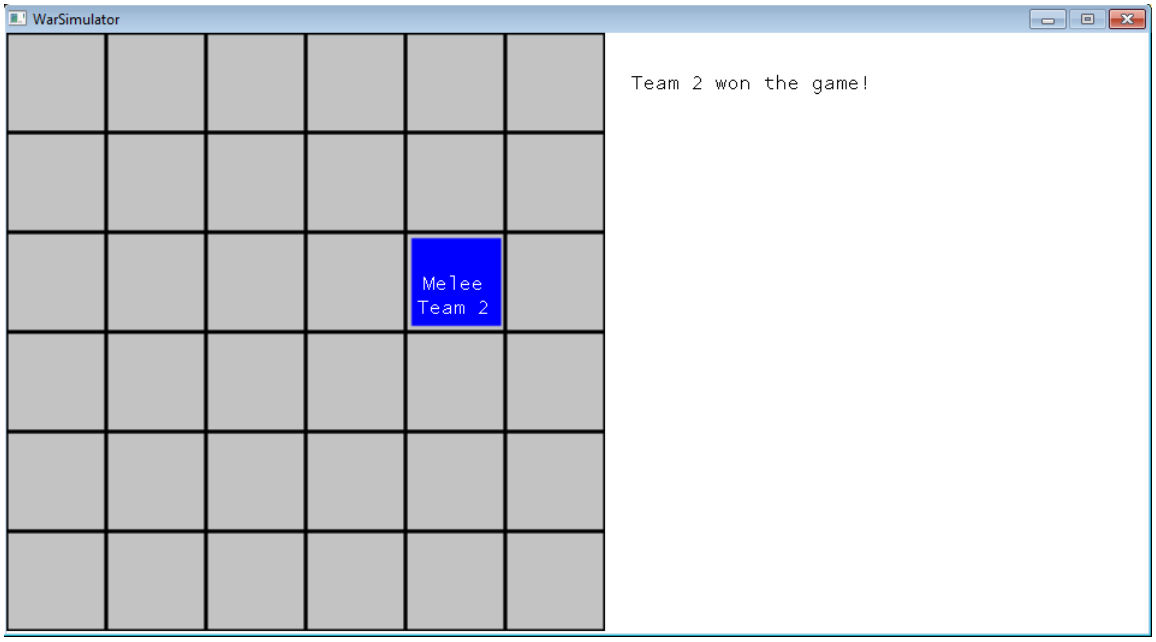


Figure 7.3: Team 2 (The Ninjas) won!

7.2 Use case 2

Big battle

Using the same regiments and config-file as seen in section 7.1, with the only alteration being the dimensions of the grid, this use case demonstrates a great battle between four opposing forces. In this scenario, the dimensions of the grid are increased to 10. As seen on figure 7.4 we see how all regiments engage the nearest enemies. The purple pirates seem to have taken refuge in a corner of the map. This imitates a common strategy - let the enemies fight amongst themselves, and then swoop in for the killing blow. This is also illustrated very well in figure 7.5, where the red, blue and green teams are at each others throat, while the purple pirates have yet to land a single strike. In figure 7.6 we see the result of the purple pirates patience - their team won, and two regiments survived!

Screenshots from use case 2

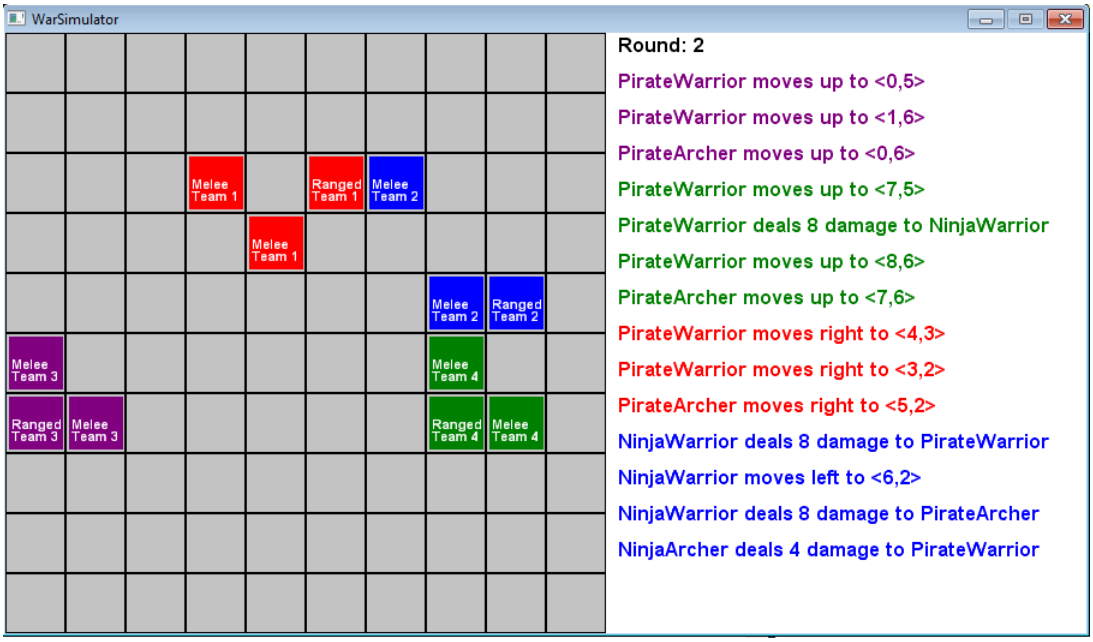


Figure 7.4: The battle begins

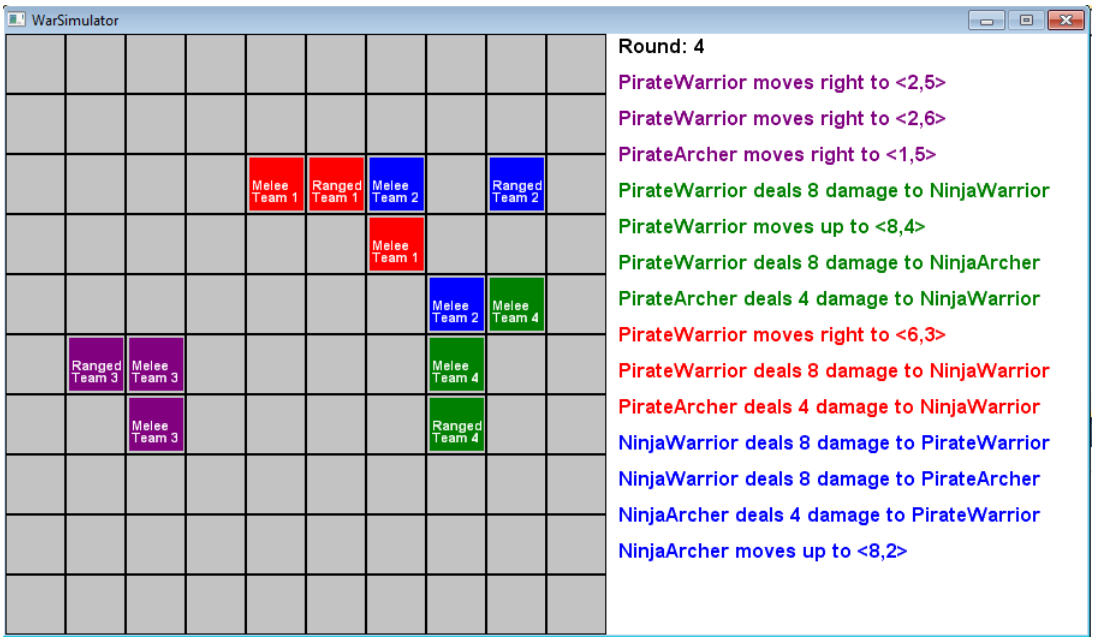


Figure 7.5: The Purple Pirates are passive

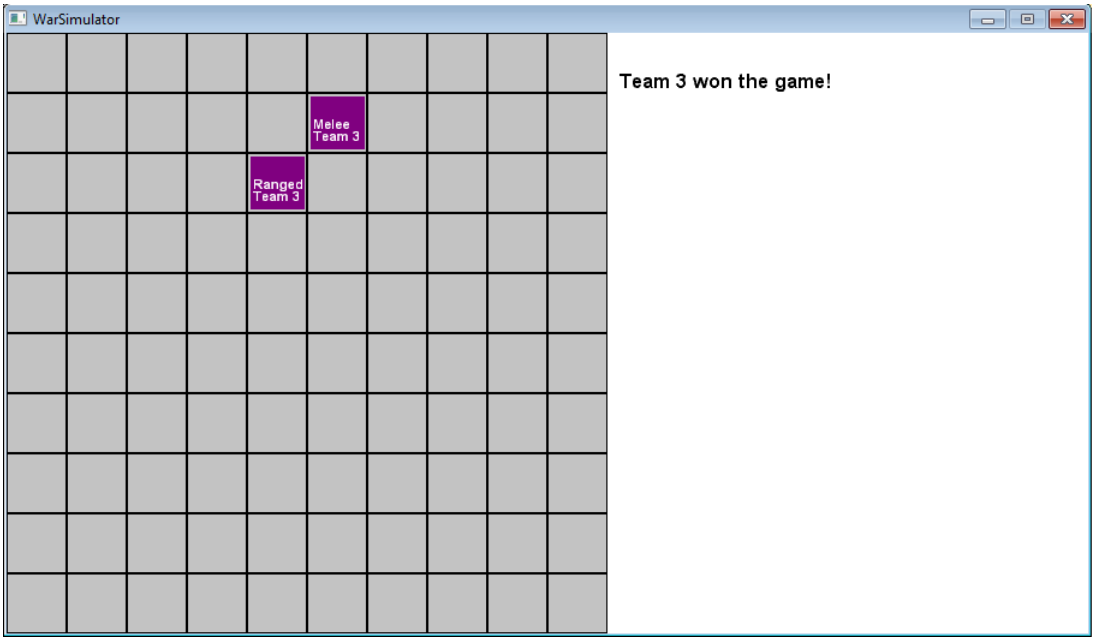


Figure 7.6: The Purple Pirates won!

7.3 Usecase 3

In this usecase, we will explore the scripts of an experienced scripter versus an inexperienced scripter. The experienced scripter wrote Team 1 and the inexperienced scripter wrote Team2. The purpose of the use case is to show that the better scripter will win.
Team 2 consist of one slow, but dangerous melee regiment called KlamBymilits. The scriptwriter behind team 2 in convinced this regiments high damage will make it possible for him to take out both of the opponents regiments. The regiment's behaviour is extremely simple - attack any nearby enemies, and if no enemies are within range, start moving towards the enemy.
The scripter behind team 1 has two regiment of archers with considerably less *Health*. These regiments are required to use their speed and positions to defeat their enemy. The archers will spawn on each side of the enemy regiment. The behaviour of the archer regiments is designed to take advantage of the slow movement and simple tactic of the enemy. When the distance between one archer regiment and the enemy, is less than the distance between the enemy and the other archer, the archer regiment moves away from the enemy regiment. If the distance is greater, than the distance between the enemy and the friendly archer regiment, the archer regiment attacks and moves towards the enemy.
The result is quite spectacular - the archers successfully kite the enemy regiment in circles, and win without taking any damage.

```
Team Team1

Regiment Archers1
{
        Size = 50;
        Range = 10;
        Damage = 10;
        Movement = 5;
        AttackSpeed = 2;
        Health = 20;
        Type = Ranged;
        RegimentPosition = Position(5,5);
Behaviour HitnRun
{
        Regiment enemy = SearchForEnemies();
        Regiment friend = SearchForFriends();
        if(enemy.Distance < (friend.Distance-enemy.Distance))
```

```
        {
                MoveAway(enemy);
                MoveAway(enemy);
        }
        else
        {
                if(enemy.Distance < Range)
                {
                        Attack(enemy);
                }
                else
                {
                        MoveTowards(enemy);
                        MoveTowards(enemy);
                }
        }
        Attack(enemy);
}
}


Regiment Archers2
{
        Size = 50;
        Range = 10;
        Damage = 10;
        Movement = 5;
        AttackSpeed = 2;
        Health = 20;
        Type = Ranged;
        RegimentPosition = Position(15,15);
Behaviour HitnRun
{
        Regiment enemy = SearchForEnemies();
        Regiment friend = SearchForFriends();
        if(enemy.Distance < (friend.Distance-enemy.Distance))
        {
                MoveAway(enemy);
                MoveAway(enemy);
        }
        else
```

```
        {
                if(enemy.Distance < Range)
                {
                        Attack(enemy);
                }
                else
                {
                        MoveTowards(enemy);
                        MoveTowards(enemy);
                        MoveTowards(enemy);
                }
        }
        Attack(enemy);
}

}

Team Team2

Regiment KlamBymilits
{
        Size = 20;
        Damage = 20;
        Type = Melee;
        Health = 50;
        Movement = 2;
        AttackSpeed = 1;
        RegimentPosition = Position(10,10);
Behaviour StupidBehaviour
{
        Regiment enemy = SearchForEnemies();
        if (enemy.Distance > 1)
        {
                MoveTowards(enemy);
        }
        else
        {
                Attack(enemy);
        }
}
}
```

Screenshots from use case 3

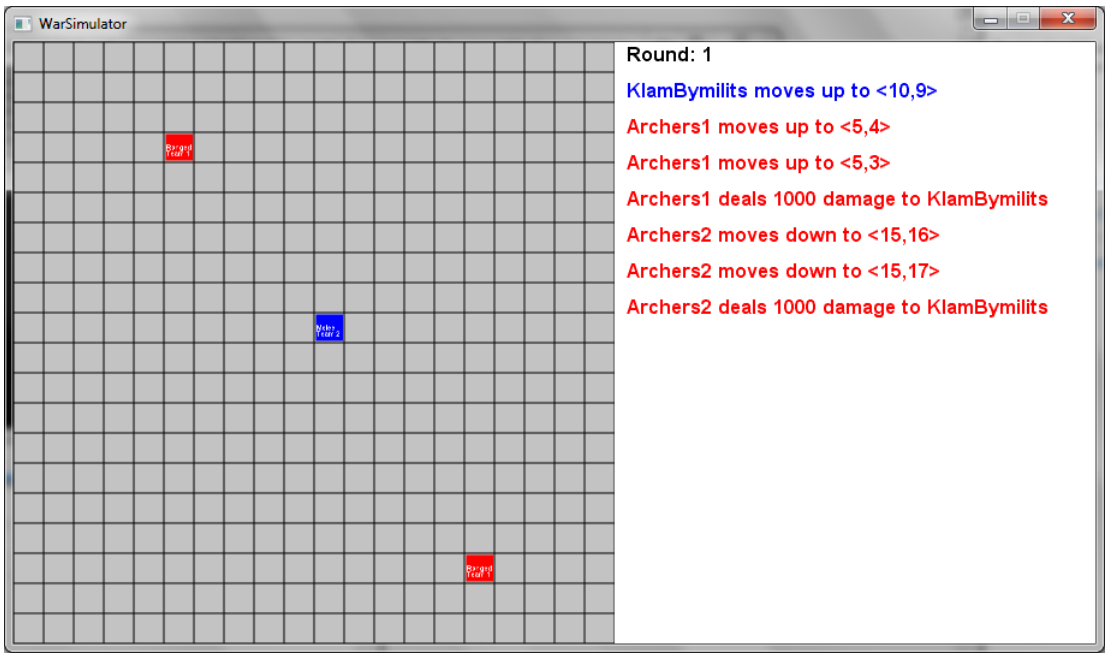


Figure 7.7: The battle begins.

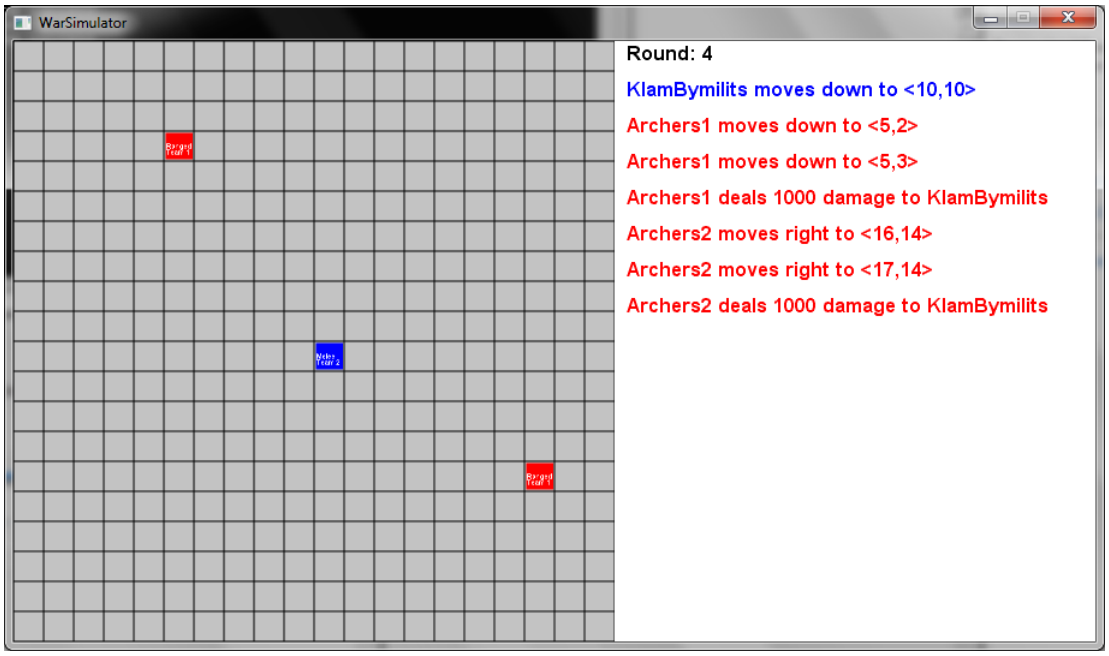


Figure 7.8: The blue team is hesitant to attack either of the red archers.

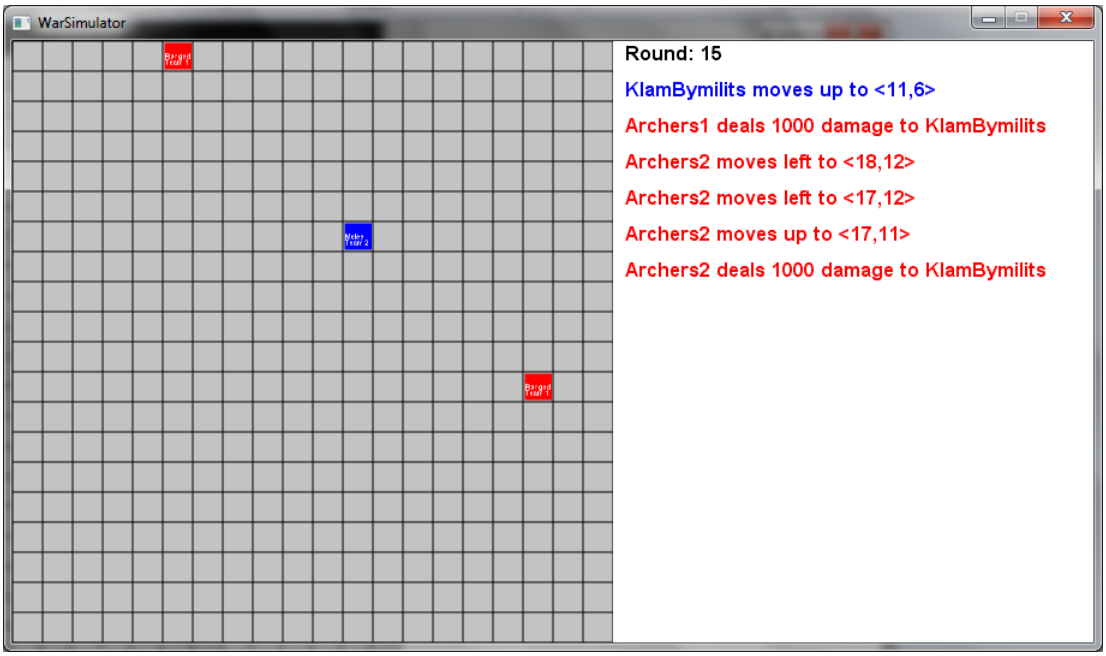


Figure 7.9: Even after 15 rounds, the blue team is only a little better off.

7.4 Error handling

This section will explain and show how the simulator will handle error reporting.

7.4.1 Error handling on input

When inputting files for the simulator different checks are made to ensue the right files are given. The first check that is made is to check if less than 3 files was given as arguments, because the simulator needs at least 1 config file and 2 team files. After this we check if the files have the correct file endings (.war for team files and .cfg for config files). The code for this is shown below in listing 7.1.

```
1 for (int i = 1; i < args.Length; i++)
2 {
3         if (!args[i].EndsWith(".war"))
4     {
5         Console.WriteLine("Incorrect filename of teamfile at {0} - Correct usage: .war", (i+1));
6         correctUsage = false;
7     }
8     teamFilesArgs.Add(args[i]);
9 }
```

```
10      if (!args[0].EndsWith(".cfg"))
11      {
12          Console.WriteLine("Incorrect filename of configfile – Correct usage: .cfg");
13          correctUsage = false;
14      }
```

Listing 7.1: Checking of file endings

7.4.2 ErrorReporter class

This class is borrowed from Bent Thomsen's [8] C♯ implementation of the *ErrorReporter* class found in the Java version of MiniTriangle compiler and interpreter. The class keeps track of how many errors that is currently found and can be used for printing errors to a console. See figure 7.10 for an example of how the *ErrorReporter* prints errors to the console.

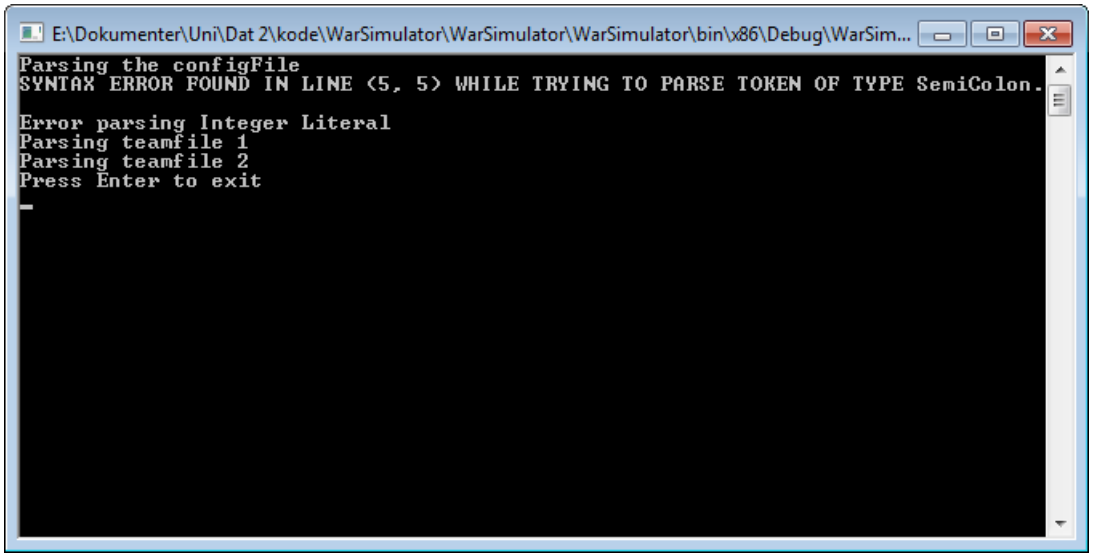


Figure 7.10: Printing of errors to the console

7.4.3 GameDataValidator

The *GameDataValidator* class (see section 6.3.3) prints errors to the console if it is unable to validate some of the data. The error messages are very simple, which can make it difficult to find out what is wrong in the config file or team files. The messages are:

- Maxima exceeded - This message is given when a UnitStat exceeds a Maxima

- Coordinates out of range - This message is given when a position is outside the *Grid*
- Positions overlap - This message is given when two regiments stand on the same position

To make debugging easier these messages could be more specific by referencing the regiment who gives the error.

Chapter 8

Language Extensions

In this chapter we will discuss some possibilities and considerations for future extensions to the simulation and to the language to support it. Since this is just a basic version of the language it has stripped down functionality - however future extensions are possible and we will cover what intentions we could have for future work. Through further scripting possibilities the simulation should come to support certain interesting aspects of unit properties, such as morale for regiments and thereby units.

8.1 Team Scripting Extensions

Improvements to the scripting of a team, and thereby the army definitions and behaviours, would make the simulation more dynamic. In doing so, more methods might be introduced to be used to make more advanced and realistic behaviours. Making more advanced behaviours would allow the armies to move in a more interesting manners, such as flanking enemy regiments.

8.1.1 Leaders

The notion of having a special unit being responsible for different parts of a team.

General

An interesting aspect to have, was having a leader or a 'General' leading the army. It is up to the script to define a behaviour for your army which best protects your general. Your General would serve as a large power of the army, and victory conditions could be to defeat the enemy general. The general could also serve the purpose of a moral change of the entire regiment,

which could empower this regiment, so if the general is near the death, the regiment will gain a higher moral, and if the general dies the regiment might loose some moral, and fight less powerful. This would mean an addition to the language, enabling the possibility of specifying a General in your team configuration.

```
Team MonkeyNinjas
{
        General
        {
        Movement =  50;
        Type = Melee;
        AttackSpeed = 1;
        Health = 150;
        }
}
```

In doing so, we would also need to allow the script to establish boundaries for the generals in the configuration file.

Regiment Leader

The idea of having a regiment leader is to make battles more dynamic. Each regiment would be assigned a leader which would inherit some of the properties of the regiment. The leader of a regiment would be a single unit, but very strong. A regiment leader would decide the behaviour of a regiment and as such the regiment might behave in another way with or without a regiment leader. A regiment leader inherits the properties of the regiments - for example a ranged regiment of archers would have a stronger archer leader with more range, damage, and what other properties that regiment might have. Depending on the type of the regiment leader the positioning of the regiment leader may vary, it makes sense to have a strong melee leader in the front of a regiment, while a strong ranged one might stay in the back.

8.1.2 Unit Orientation

To introduce further complexity of movement and of tactics, functions could be added to control the orientation of a regiment. Not only to be able to control whether or not a regiment has its back turned to an opposing regiment, but also to be able to flank opposing regiments. A flanked regiment would have to use an action to reorientate itself, in order to fight a flanking regiment.

8.1.3 Unit Management

Currently the engine of the simulation handles an entire regiment on its own - but it would be entirely more interesting through visualization to have each unit interact with each other on a individual basis. Movement would be handled by regiment, but attacking, damage, and deaths of units would be handled by the units of a regiment interacting with each other. As such only the front lines of a melee regiment would be able to attack at a time, and the units of a regiment could take on some interesting attributes. To handle the positioning of units, what the position of a regiment, would serve as the center point of the regiment, and the positioning of units would be placed from this center. The scripter would have the ability to design the desired formation of the regiment in the team-file. This could be further extended with different special formations - e.g. scatter formation, ball formation, single file formation.

8.1.4 Unit Morale

This is an attribute of a unit, which would be inherited from the regiment which the unit belongs to. Overall a regiment with a Regiment Leader would have a higher morale as mentioned earlier 8.1.1. Having a higher morale may boost certain abilities of the regiments, such as the chance for an attack to hit. Morale could also simply adjust the number of actions a regiment can perform in its turn. This attribute would be treated in different ways, and be dependant upon many other factors in the simulation. It would be impossible to assign a specific morale value to a regiment in the WAR-script, but it would be possible to adjust behaviour according to a level of morale. Morale would be an integral part of how well a regiment functions, and how it behaves. It would be event dependant and the importance of it configurable in the configuration script.

8.2 Configuration Enhancements

Enhancements to the setup of the game. This would expand the language to have some more interesting properties in the configuration and thereby the initial state of the game. To achieve this the language would require some modifications and additions, to allow for a more flexible configuration script.

8.2.1 Boundaries

Currently the only thing governing the limitations of the regiment and team properties is the maxima that are specified. This is open for abuse, because nothing prevents simply just maxing out to all the maxima - this would leave all regiments to be identical and thereby very uninteresting. This brings up several extensions that are needed for the language and the framework in it self. Boundaries are not the same as maxima, we want to achieve regiments that are well balanced, for example, a unit with high damage might not have as high movement speed or attack speed. A way to do this is to set up a maximum for a team. This maximum is defined in the configuration as a simple point value, different unit-stats would weigh differently with the points, as would the size of a regiment. This would ensure that each team is evenly balanced - and the teams would be able to have any number of regiments as long as the army itself is within the point limitation. This presents some issues -error reporting would need to be improved to recognize the boundaries of any single regiment, and report an error before launching the simulation if anything is out of boundaries. Additionally it would need to be able to specify what has exceeded the maxima and by how many points.

8.2.2 Winning Conditions

The language should support the possibility of defining specific winning conditions through the configuration script. Choosing from preset rules or setting up conditions through control statements. An example of setting up such a condition could be something similar to the following:

```
Rules
{
        Conditions
        {
        //Team loss if General is destroyed
        Defeat = General; // 'General' would be a pre-set rule.
        }
}
```

Other examples of a preset rule might be Last-Man-Standing, where the armies would battle until only one single unit remains.

It should also be possible to set up the conditions with a control statement:

```
Rules
{
        Conditions
```

```
        {
                if(Regiments == 0)
                {
                Defeat;
                // Defeat event - means a loss for the team.
                }
        }
}
```

The intention would be to set up conditions easily, while also allowing for much more specific conditions, through more advanced scripting. The conditions would not be limited to regiment attributes - conquering a special field on the grid could also be a viable condition for victory or defeat.

8.2.3 Grid Obstacles

Not every battlefield is a clear square - support for adding obstacles to the grid would be interesting. This could be achieved by giving the configuration file some data from which to generate random or specific obstacles. One might be to determine the density of obstacles on a grid, and another the size of the obstacles. Obstacles could be placed randomly on the grid according to the configuration file, or in designated tiles. This would also allow for some further interesting specifications in the behaviour definitions for a regiment. A scripter should be able to make a regiment avoid obstacles and take the most efficient route or use the obstacles as choke-points to deal with a superior enemy.

8.3 Engine and GUI Improvements

8.3.1 Event Handling

When certain events in the simulation takes place, we would like to handle these events, and have some consequences for these which affects the flow of the game.

Regiment Leader Death

Should the leader of a regiment perish, this could affect the remaining units of a regiment somehow. Maybe the regiment would lose morale, maybe the units of a regiment would become scattered.

General Death

Depending on the conditions set forth in the configuration of the game, a team may be declared defeated if their general dies. However if that condition is not set in the event of the death of a General this could be set to affect the entire army in some way.

8.3.2 Finer Simulation Control

During the course of the simulation, a player might have regretted some of the behaviours of a regiment. In such a case it could be interesting to allow the engine to ask for a new team script from both players at a set time. Such a thing would require no changes to the language itself, as the flexible nature of the script interpreter alone would allow for things such as behaviours on the fly.
For instance the simulation may simply ask for a new team file at a set interval, numbered by the time passed. Of course time is measured in the context of rounds in this simulation.

8.4 Language Additions

The language could have some more basic functionalities, which would not affect the structure of the simulation itself.

8.4.1 Libraries

The possibility of including libraries to your script, and calling certain things from those libraries. This would open many possibilities of saving your behaviours, or developing more and more complex behaviour packages over time. This would mean including a certain library in your script, enabling you to call behaviours or functions from that library. Calling such a library behaviour would appear like this:

```
Team Ninjutsu
#include AssassinationTechniques

Regiment Assassins
{
        Size = 10;
        Type = Ranged;
        Damage = 10;
```

```
        Movement = 70;
        AttackSpeed = 1;

        Behaviour AssPlan
        {
                Regiment enemy = SearchForEnemies(General);
                if(enemy.Distance <= Range)
                {
                        //This is a library function
                        AssasinateGeneral();
                }
                else
                {
                        SneakToGeneral();
                        //This is a library function
                }
        }
}
```

8.4.2 Configuration Variables

The possibility of setting a variable which changes throughout the simulation. These variables could be set purely to be used in the scripting of each team, or make regiments on the grid dependant on them. An example of this could be setting a global morale modifier in the configuration script. Some event might call for this morale modifier to be reduced - making every regiment on the grid lose morale. This would bring in the need to make a regiment less dependant on morale, as a team might gain an advantage over the other in that case.

Chapter 9

Conclusion

In this final chapter of the report we summarize the work done, and how well the goals of the project were fulfilled.

9.1 Conclusion

The goal of the project was to create a language, that when interpreted could emulate a battle between multiple armies. This goal has been fulfilled by programming a simulator, which does syntactical analysis, semantic analysis and interpretation. Syntactical analysis was done by recursive descent parsing the scripts written in our own scripting language - *WAR*. The semantic analysis was applied by using a visitor pattern that could check for type and scope rules. The simulator contains an interpreter interprets the Decorated Abstract Syntax Tree, provided by semantic analysis. The most challenging part of the interpretation, was to interpret a regiments behaviour.

Looking back at chapter 3.1 and our design criteria described in table 3.1, we see that not all of our design criteria were fulfilled.
The scripting language has a very high writeability and readability - we assume that most amateur programmers should be able to write their own unique script only using another script as reference. Unfortunately, there was no time to test the readability and writeability of the language.
The reliability of the program is also well established, but some scripters may be surprised when not all their planned behaviours are performed in a turn, due to the limited actions per turn. The straightforward composition of the language, should provide very few misunderstandings to the functionality of the various commands provided by the language.
The criterion of low Orthogonality of the scripting language is fulfilled implicitly,

due to the rigid nature of the team files and configuration files.
Because each regiment-declaration bears great resemblance to the declarations in the configuration file, it is safe to say that the scripting language is designed for high uniformity.
It is relatively easy to find errors in a script, when trying interpret it, because the syntactical and semantic analysis is implemented with an error reporter, which provides error messages with reference to the line containing the error. However, the error reporter is not very specific in explaining which file the error occurred in. In addition, some of the error messages are of debatable precision and quality. As such, the design criterion of very high maintainability is not fully satisfied.
Generality was estimated to be of mediocre importance as a design criterion, but the aforementioned structure of the team files and configuration files provide great generality for our language.
Providing extensibility for our language is simply impossible - there is no command or function provided by the language, which could extent the language in any way. To extent the language, extensions to the EBNF, parser, interpreter and simulator would be required.
The language can be transported to any other computer, provided it has an implementation for the .NET-framework. This provides the mediocre standardization we desired in our design criteria.
Because we were able to implement an interpreter for the language, the design criterion implementability, is surely fully fulfilled.

There was no design criterion related to the speed of the interpretation, but because the language is so simple, the syntactical and semantic analysis is very fast, as well as the interpretation.

Appendix A

BNF

Notes

- Digit represents one of the digits 0 through 9.
- Graphic represents a space or visible character.
- Letter represents one of the lower- or upper-case letters 'a','b',.....,or 'z'.
- eol represents an end-of-line 'character'.

Team-File	::=	**Team** Identifier Regiment-Block
Identifier	::=	Letter
	|	Identifier Digit
	|	Identifier Letter
Block-Name	::=	Identifier
Regiment-Block	::=	**Regiment** Block-Name **{** UnitStat Behaviour-Block **}**
	|	Regiment-Block **Regiment** Block-Name **{** UnitStat Behaviour-Block **}**
Behaviour-Block	::=	**Behaviour** Block-Name **{** Single-Command **}**
	|	**Behaviour** = Block-Name**;**
Regiment-Declaration	::=	**Regiment** Identifier = Regiment-Search**;**
	|	**Regiment** Identifier = Block-Name**;**
Regiment-Search	::=	**SearchForEnemies** (Parameters)
	|	**SearchForFriends**(Parameters)

RegimentStat	::=	Identifier **.** UnitStat-Type
Parameters	::=	UnitStat-Type Operator Integer-Literal
	|	Parameters**,** UnitStat-Type Operator Integer-Literal
Single-Command	::=	Control-Structure Single-Command
	|	Regiment-Declaration Single-Command
	|	UnitFunction Single-command
	|	Control-Structure
	|	UnitFunction
	|	Regiment-Declaration
ElseIf-Structure	::=	**else if(** Expression**)** **{** Single-Command **}** ElseIf-Structure
	|	**else if(** Expression**)** **{** Single-Command **}**
Control-Structure	::=	**if(** Expression**)** **{** Single-Command **}**
	|	**if(** Expression **)** **{** Single-Command **}** **else** **{** Single-Command **}**
	|	**if(** Expression **)** **{** Single-Command **}** ElseIf-Structure **else** **{** Single-Command **}**
	|	**if(** Expression **)** **{** Single-Command **}** ElseIf-Structure
	|	**while** **(** Expression **)**{ Single-Command **}**
Expression	::=	Primary-Expression
	|	Expression Operator Primary-Expression
Primary-Expression	::=	(Expression)
	|	Integer-Literal
	|	UnitStat-Type
	|	RegimentStat
Operator	::=	+
	|	−
	|	*
	|	/
	|	<
	|	>

	|	<=
	|	>=
	|	==
	|	&&
	|	||
UnitFunction	::=	**Attack(** Identifier **);**
	|	**MoveTowards(**Identifier **);**
	|	**MoveAway(** Identifier **);**
UnitStat	::=	UnitStat-Declaration UnitStat
	|	UnitStat-Declaration
UnitStat-Declaration	::=	**Size** = Integer-Literal**;**
	|	**Type** = AttackType**;**
	|	**Range** = Integer-Literal**;**
	|	**Damage** = Integer-Literal**;**
	|	**Movement** = Integer-Literal**;**
	|	**AttackSpeed** = Integer-Literal**;**
	|	**Health** = Integer-Literal**;**
	|	**RegimentPosition = Position(** Integer-Literal **,** Integer-Literal **);**
UnitStat-Type	::=	**Size**
	|	**Type**
	|	**Range**
	|	**Damage**
	|	**Movement**
	|	**AttackSpeed**
	|	**Health**
	|	**Distance**
AttackType	::=	**Melee**
	|	**Ranged**
Integer-Literal	::=	Digit
	|	Integer-Literal Digit
Comment	::=	// Graphic-Literal eol
Graphic-Literal	::=	Graphic Graphic-Literal
	|	Graphic
Config-File	::=	**Config** Grid-Block Rules-Block
Grid-Block	::=	**Grid** Block-Name **{** GridStat **}**
GridStat	::=	GridStat-Declaration GridStat
	|	GridStat-Declaration
GridStat-Declaration	::=	**Width =** Integer-Literal;

	|	**Height** = Integer-Literal;
Rules-Block	::=	**Rules { ** Standards-Block MaximaBlock **}**
Maxima-Block	::=	**Maxima { ** MaximaStat **}**
MaximaStat	::=	MaximaStat-Declaration MaximaStat
	|	MaximaStat-Declaration
MaximaStat-Declaration	::=	**Regiments** = Integer-Literal**;**
	|	**Teams** = Integer-Literal**;**
	|	**Size** = Integer-Literal**;**
	|	**Range** = Integer-Literal**;**
	|	**Damage** = Integer-Literal**;**
	|	**Movement** = Integer-Literal**;**
	|	**AttackSpeed** = Integer-Literal**;**
	|	**Health** = Integer-Literal**;**
Standards-Block	::=	**Standards { ** UnitStat-Declaration Behaviour-Block**}**

Appendix B

EBNF

We applied left factorization, substituion of non-terminals and elimination of left recursion to transform the BNF to an EBNF. Substituion of non-terminals have removed the following non-terminals from the BNF:

- Else-If
- UnitStat
- Graphical-Literal
- GridStat
- MaximumsStat

Left factorization and elimination of left recursion was applied to make the EBNF:

Team-File	::=	**Team** Identifier Regiment-Block*
Identifier	::=	Letter (Letter | Digit)*
Block-Name	::=	Identifier
Regiment-Block	::=	**Regiment** Block-Name **{** UnitStat-Declaration Behaviour-Block **}**
Behaviour-Block	::=	**Behaviour**(Identifier**{** Single-Command **}** | = **Identifie**
Regiment-Declaration	::=	**Regiment** Identifier = Regiment-Search**;**
	|	**Regiment** Identifier = Block-Name**;**
Regiment-Search	::=	**SearchForEnemies** (Parameters)
	|	**SearchForFriends**(Parameters)
RegimentStat	::=	Identifier**.**UnitStat-Type

Parameters	::=	UnitStat-Type Operator Integer-Literal (**,**UnitStat-Type Operator Integer-Literal)*
Single-Command	::=	(Control-Structure | UnitFunction | Regiment-Declaration)* (Control-Structure | UnitFunction | Regiment-Declaration)
Control-Structure	::=	if(Expression) **{Single-Command }** (**else if(Expression){ Single-Command })*** (ϵ | else **{ Single-Command }**
	|	while(Expression)**{ Single-Command }**
Expression	::=	Primary-Expression (Operator Primary-Expression)*
Primary-Expression	::=	(Expression)
	|	Integer-Literal
	|	UnitStat-Type
	|	RegimentStat
Operator	::=	+
	|	−
	|	*
	|	/
	|	<
	|	>
	|	<=
	|	>=
	|	==
	|	&&
	|	||
UnitFunction	::=	(**Attack** | **MoveTowards** |**MoveAway**)(Identifier **);**
UnitStat-Declaration	::=	(**Size**|**Range**|**Damage** |**Movement** | **AttackSpeed**|**Health**) = Integer-Literal**;**
	|	**RegimentPosition =** **Position(**Integer-Literal**,**Integer-Literal**);**
	|	**Type** = AttackType**;**
UnitStat-Type	::=	(**Size**|**Range**|**Damage** | **Movement**| **AttackSpeed**|**Health**|**Distance**)
AttackType	::=	**Melee** | **Ranged**
Integer-Literal	::=	Digit Digit*
Comment	::=	// Graphic* eol
Config-File	::=	**Config** Grid-Block Rules-Block
Grid-Block	::=	**Grid** Block-Name **{** GridStat-Declaration* **}**
GridStat-Declaration	::=	(**Width** | **Height**)= Integer-Literal**;**
Rules-Block	::=	**Rules {** Standards-Block MaximaBlock **}**
Maxima-Block	::=	**Maxima {** (MaximaStat-Declaration)* **}**

MaximaStat-Declaration	::=	(**Regiments** |**Teams**|**Size**| **Range**|**Damage**|**Movement**| **AttackSpeed**|**Health**) = Integer-Literal**;**
Standards-Block	::=	**Standards {** UnitStat-Declaration* Behaviour-Block**}**

Bibliography

[1] Overview of the four main programming paradigms; 2010. Available from: http://www.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html.

[2] Thomsen B. Programming Languages and Compilers - Lecture Slide 1; 2011. Available from: https://intranet.cs.aau.dk/uploads/media/SPOF11-1.pdf.

[3] International E. Common Language Infrastructure (CLI) Specification. ECMA International; 2010. Available from: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[4] Thomsen B. Programming Languages and Compilers - Lecture Slide 2; 2011. Available from: https://intranet.cs.aau.dk/uploads/media/SPOF11-2.pdf.

[5] Watt DA, Brown DF. Programming Language Processors in JAVA. Pearson Education Limited; 2000.

[6] Thomsen B. Programming Languages and Compilers - Lecture Slide 6; 2011. Available from: https://intranet.cs.aau.dk/uploads/media/SPOF11-6.pdf.

[7] Alfred V Aho RS Monica S Lam, Ullman JD. Compilers: Principles, Techniques, and Tools. 2nd ed. Pearson Education Inc.; 2006.

[8] Mini Triangle implemented in C♯; 2011. Available from: http://dw3.cs.aau.dk/~bt/TriangleInCSharp.