**SCHOOL OF COMPUTING**

**UNIVERSITY OF TEESSIDE**

**MIDDLESBROUGH**

**TS1 3BA**

**Artificial Evolution**

**BSc Computer Science**

**Abelether Germa**

**11 April 2016**

**Supervisor: Eudes Diemoz**

**Second Reader:**

# Abstract

# Acknowledgements

I would like to thank Simon Linch for his help with all aspect of the project

**CONTENTS**

[Abstract ii](#_Toc448122248)

[Acknowledgements iii](#_Toc448122249)

[1 Introduction 7](#_Toc448122250)

[2 Methodology 8](#_Toc448122251)

[3 Research 9](#_Toc448122252)

[3.1 First iteration 9](#_Toc448122253)

[3.1.1 Java 9](#_Toc448122254)

[3.1.2 Clojure 9](#_Toc448122255)

[3.1.3 SQL 9](#_Toc448122256)

[3.1.4 Visual Studio 9](#_Toc448122257)

[3.1.5 Full Integration 9](#_Toc448122258)

[3.1.6 Choice of software development tools used 9](#_Toc448122259)

[3.2 Second iteration 9](#_Toc448122260)

[3.3 Hardware used. 9](#_Toc448122261)

[3.3.1 Netlogo 9](#_Toc448122262)

[3.3.2 Java and Clojure 11](#_Toc448122263)

[4 Design 12](#_Toc448122264)

[4.1 First iteration 12](#_Toc448122265)

[4.1.1 Netlogo and Clojure 12](#_Toc448122266)

[4.1.2 Between Java and Clojure 12](#_Toc448122267)

[4.1.3 Java to MS SQL Management Studio 12](#_Toc448122268)

[4.1.4 Visual Studio 12](#_Toc448122269)

[4.1.5 Full integration 12](#_Toc448122270)

[4.2 Second iteration 12](#_Toc448122271)

[4.2.1 Netlogo 12](#_Toc448122272)

[5 Implementation 13](#_Toc448122273)

[5.1 First iteration 13](#_Toc448122274)

[5.1.1 Communication Between Netlogo and Socket 13](#_Toc448122275)

[5.1.2 Communication between Java and Clojure 13](#_Toc448122276)

[5.1.3 SQL 13](#_Toc448122277)

[5.1.4 Clojure and Netlogo 13](#_Toc448122278)

[5.2 Second iteration 14](#_Toc448122279)

[5.2.1 Netlogo 14](#_Toc448122280)

[6 Testing 14](#_Toc448122281)

[6.1 Testing in Clojure 14](#_Toc448122282)

[6.2 Testing in Netlogo 14](#_Toc448122283)

[6.3 First Iteration 16](#_Toc448122284)

[6.3.1 Between Clojure and Netlogo 16](#_Toc448122285)

[6.3.2 Between Java and Clojure 17](#_Toc448122286)

[6.4 Second iteration 17](#_Toc448122287)

[6.4.1 Netlogo 17](#_Toc448122288)

[7 Evaluation 18](#_Toc448122289)

[7.1 Iteration one 18](#_Toc448122290)

[7.2 Iteration two 20](#_Toc448122291)

[7.3 Scope of project 20](#_Toc448122292)

[7.4 Provisional analysis 20](#_Toc448122293)

[7.5 Future improvements 20](#_Toc448122294)

[7.5.1 Code review 20](#_Toc448122295)

[7.5.2 Readability 20](#_Toc448122296)

[7.5.3 Reusability 20](#_Toc448122297)

[7.5.4 Structure 20](#_Toc448122298)

[7.5.5 Features 21](#_Toc448122299)

[7.6 Social implication 23](#_Toc448122300)

[7.7 Critical evaluation 23](#_Toc448122301)

[8 Recommendations 24](#_Toc448122302)

[9 Conclusion 25](#_Toc448122303)

[List of Figures 26](#_Toc448122304)

[References 28](#_Toc448122305)

[Appendix 29](#_Toc448122306)

[9.1 Appendix 0 29](#_Toc448122307)

[9.2 Appendix 01 – First iteration 29](#_Toc448122308)

[9.2.1 Implementation 29](#_Toc448122309)

[9.2.2 Testing 33](#_Toc448122310)

[9.3 Appendix 02 – From Iteration 02 36](#_Toc448122311)

[9.3.1 Implementation 36](#_Toc448122312)

[9.4 Appendix 03 37](#_Toc448122313)

# Introduction

## Problem definition

## Problem justification

Evolution is constantly changing the world, sometimes in a blink of an eye, other times millions of years but it is always working. What is suppressing is that this unintelligent, chaotic and completely random system produced complex and intelligent systems. Scientists have been researching over 100 years and even though not every little detail is known, we have all the steps that was taken to achieve complex life on earth. Can this process be replicated?

## System Requirements

### Initial Objectives

* Create a Netlogo environment where agents can move and interact with the environment.
* Agents must be able to die and be able to do something to prevent them self from dying.
* Create communication between, Netlogo, Clojure and java.
* Create a java application where agents DNA is mutated.

### Overall Objective

* Produce a database where agents DNA can be stored.
* Allow the application to perform switchover when requested.

### Advanced objective

* Create a data cube to analyse the database.
* Create a website to allow anyone around the world access to the results found.

## Products used for this project

* IntelliJ IDEA – for java and Clojure development
* Netlogo – Integrated modelling environment for Visulisation
* MS SQL Server 2014 – For the database
* Visual Studio 2013 – For data mining
* Github – For Version control
* Pandoc – Version Control for Word (works along with github)
* Teamviewer – Accessing Home computer while at the university.

# Methodology

# Research

## First iteration

### Java

#### Communication between Java and Clojure

Clojure run on the JVM as well as Clojure programs run in a JVM. What this mean is Clojure itself was created using java and is in fact an application that run on the JVM just like any other Java application. This also means Clojure programs gets compiled into Java bytecode which the JVM executes and sends instructions to machine as shown in the figure below. Due to this, a Clojure can be viewed just like another java class and Vis versa.

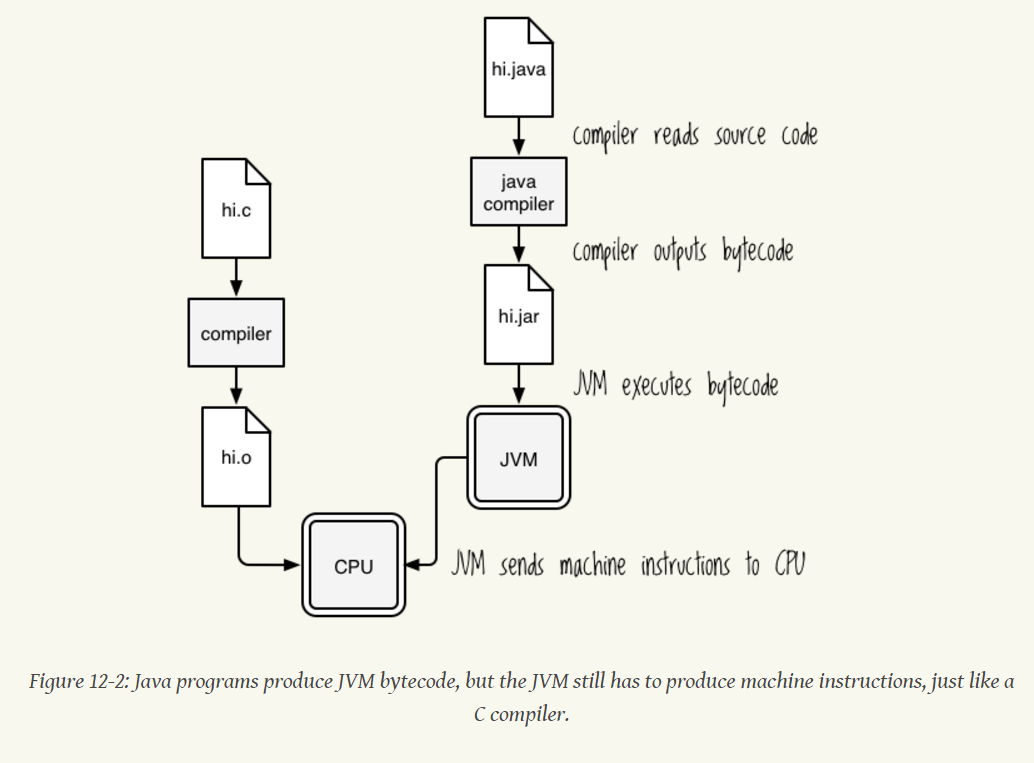


Figure 3.1‑1: Higginbotham, D. (2015) Clojure for the brave and true

Even though a Clojure and Java class are very alike, to access each other different syntax’s is used compared to accessing the same type of class.

For example in Clojure when referring another Clojure class, the following code is used.

(**require** [comm.wrapper :refer :all]

Require: loads the class

Refer : Adds specified item to the current name spaces.

Use : Require + User

### Clojure

Clojure main use was to create communication between Java and Netlogo.

#### Clojure to Netlogo

A Socket would be used to communicate with between Clojure and Netlogo, The socket is stack based and would be transferring data between the two components as a string. Therefore the data received from Java needs to be a string or needs to be converted into a string before sending to Netlogo. The beginning of the string should indicate what kind of message it is so Netlogo knows what to do with the set of string. E.g. create a new agent or change an environment variable.

#### Clojure to Java

Once information is received front Netlogo a few things need to happen before sending it to Java.

1. Split up the string – the socket is stack based and all words that is being sent from Netlogo are received one word at a time.

There is two ways to deal with this.

* First- Indicate when a sting of commands starts and ends by a certain words e.g. start and end.
* Second- send everything as one word and split it up once it is received by Clojure. Set of instructions can be split up by using a hyphen, comma or a plus sign

1. Understand where the information is going – Is the information going for evaluation, or straight to a database. This is a question that Clojure must ask every time a new set of strings arrive. This information can be embedded with in the first string, and this method can also work the same way when sending data to Netlogo.

For example if an agent has died and its information simply needs to be sent to a database, this can simply be indicated by putting the word “dead-agent” or “to-SQL” at the beginning of the string

1. Once broken down each set of string must be converted to its correct data types. Therefore when it needs to be sent to java an integer, the Clojure class can convert it to correct format and send it in as an object if needs be.

### SQL

Due to how this software will run there is going to be a vast amount of data that is going to be generated. By looking at other Netlogo projects it was decided that all results need to be stored Into A database. Within Netlogo an agent can be expected to die every few second, or A few agent every second and the more agents there are at a time it becomes very difficult to tell what is really going on. Plot charts exist within Netlogo but that can only show the big picture and is not able to answer any specific questions.

For example using plot charts in Netlogo, Questions such us What is the average ‘speed’ or other attributes of total agents can be asked but when asking about a specific agents, it becomes impossible. By storing all the data in an SQL database questions such us, who are this agents parents? Or at a certain time what was the average speed of agents? This kind of questions and many more become easy to answer once everything is within a database.

Using Sql, By designing the relationship diagram correctly the amount of data stored can be reduced dramatically by removing any repeating data.

### Visual Studio

After running the evolution system software for a few hours, a vast amount of data and it is going to require a team of people to go through it all and long amount of time, around 6 month to have any conclusion.

To tackle this issue, it was important that a data analysis tool was set up.

Visual studio, using a database allows the creation of multi-dimensional data structures and uses data mining to come to conclusion within hours what could take a person a few weeks.

It would be able to answer questions such us what are the probabilities of an agent surviving with a speed of 0.2?

### Full Integration

#### Java to SQL

#### SQL to Visual Studio

## Second iteration

## Hardware used.

### Netlogo

Within Netlogo Agents are referred as turtles. While speaking within Netlogo context the term turtle would be used to refer to a species of agents and the term agent will be used for one specific agent.

#### How successes is measured

Depending on the type of model created there is many ways successes can be measured. From their speed or from the level of energy gained after the session is finished. For this project the level of energy gained at the end of a session or when the agent dies. After the end of a session agents total energy can be compared to determine if they have evolved or not.

#### What do agents gain energy?

Energy can be awarded by doing anything from having to perform a complex task to just turning left. There are few models that revolve around energy. The rabbits and fox model created by Simon .Linch revolves around energy consumption for survival. There are two type of turtles, Rabbits and Foxes. The rabbits run around eating grass and gain their energy from grass while the foxes run around looking for rabbits and gain their energy from eating a rabbit. This model is very simple when developing in netlogo but it quickly become very complicated when the element of evolution is added to it. To simplify this for this project there would only be one type of turtles called rabbits and they would be required to go around eating grass.

#### How genes are stored

A lot of research has been done on Deoxyribonucleic Acid (DNA) and ribonucleic Acid. How it is copied, mutated. For over 100 years but there are still key elements we don’t understand. The most recent discovery was made by a Postdoctoral student named Tomomi Kiyomitsu in 2012, He figured out how one of the motor protein helps snap the chromosomes into line which comes to show that how little we know about DNA.

DNA is a long strand of (A) adenine, (G) guanine, (C) cytosine, and (T) thymine. It uses this 4 types of nucleotide to encode a certain amino acid which when put together in a certain sequence creates a protein, there are 20 different types of amino acids in a human cell and a gene is not exactly limited to a certain size, there for there could be almost un unimaginable about of combination of amino acids that will create a certain protein.

#### Numeral Base System

It was very difficult to choose which base system to use to best imitate evolution, as we know it. The base that was conceded were base one (Unary), base two (Binary), base four (Quaternary) and base ten (Decimal). After the following considerations, Binary was chosen as most appropriate.

##### Unary

Unary is a base one numeral system where only the symbol ‘1’ is used to represent all the numbers. To represent a Decimal in unary all that is needed is to repeat ‘1’ n times, for example the number 5 in Unary it will be present as ‘11111’.

The advantages of using a Unary is it returns a vast amount of digits, even for a relatively small number as 30. This is very good because just for a gene of 10 varieties, it returns 55 digits. The more digits there is the better and more accurately evolution can be mimicked because just a single human DNA contains over 4.2 billion pair of characters and events like mutations are rare in comparison to how much data is actually being rewritten or duplicated. As base one this will by far will give the most amount of digits.

Unfortunately base one is completely unusable for DNA manipulation for a few reasons. First it only has one set of character so a single digit cannot be mutated because there is nothing else to change the number two, the only type of mutation that can applied is insertion and deletion. The second problem is it is impossible to tell where one gene starts and ends therefore making it unpractical for this situation.

##### Binary--

Binary is a base two numeral system where two different symbols are used to represent all possible numbers. This symbols are normally ‘0’ and ‘1’ in circuitry referred to as ‘off’ and ‘on’.

Binary is both simple and effective for this project. When applying mutation to a sequence of DNA with binary it will either go from 0 to 1 or vice versa. Using it in base 4 will give the possibilities 16 combinations, if one set of combination is reserved to indicate where a gene starts then that gives 15 different possible of combinations, where in DNA there are 20 amino acids.

Just like gene in our DNA if the length of a gene is not pre-defined than using this method any numerical value can be stored.

### Java and Clojure

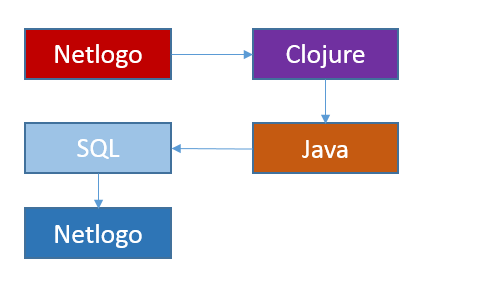
Further research was conducted on communication between Java and Clojure which led to the conclusion that the application need to be restructured to no longer user Java for data manipulation but as only as communication between Clojure and the SQL Database.

It was still possible to create an instance of a java object within Clojure therefore data to the database can be sent using java objects

# Design

## First iteration

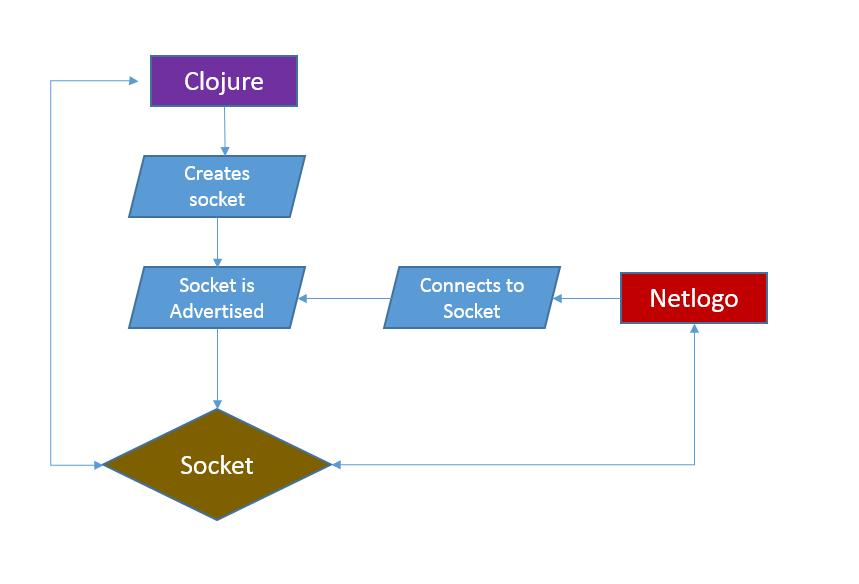
The main goal for this iteration is to stablish communications between all components. All together there will be five components Netlogo for visualisation, Java for data manipulation, Clojure for communication between Java and Netlogo, SQL for storing Data and Visual Studio for processing data to make sense of it.



### Netlogo and Clojure

#### How communication is established

Clojure first creates the socket with the port number and advertise it. Once the port is being advertised using the socket extension Netlogo connects to the socket using the port number. Once both of them are connected they can communicate with each other using the socket



#### Communication between Netlogo and Clojure

As researched the socket is stack based, which uses first in, first out. This means when a message is sent from Clojure to Netlogo, it will go on a stack and when Netlogo request to read from the socket the first message sent will be given to it and vis versa.

##### Clojure to Netlogo

The design decision was made to send data from Clojure to Netlogo as one set of string, with the first word being the method to call and the rest of the strings will contain the arguments for that method. This wad decided on the bases that it is minimal and removes the need to iterate through the string

##### Netlogo to Clojure

The same method as above will be used to read the socket. Due to the restriction that only one word can be read at a time from the socket, Netlogo would send all the words as one string combined with “:”. Once received Clojure would iterate through the string and split the string and execute them appropriately.

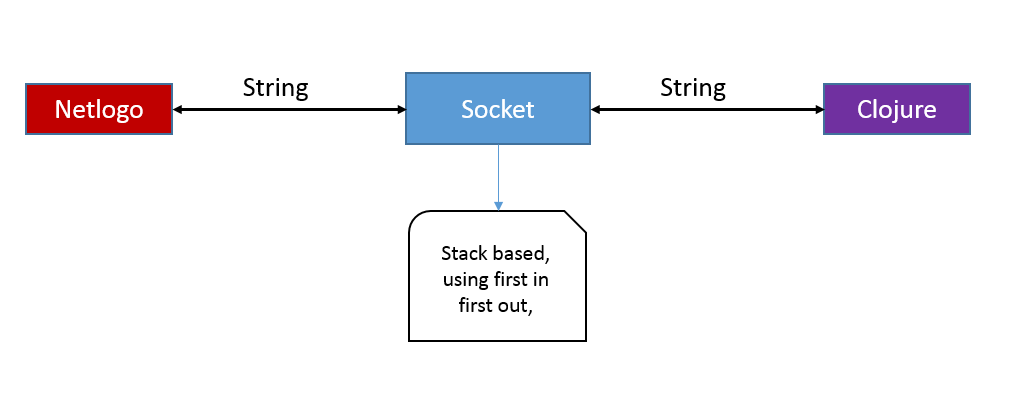


Figure 4.1‑1: Communication between Netlogo and Clojure

### Between Java and Clojure

Java and Clojure will communicate thorough java objects. Each object will have getters and setters and Clojure will be able to communicate with java using constructers.

When java need to send messages to Clojure it will simply call the corresponding method with the correct parameters, that will then be processed into a string and sent to Netlogo.

### Java to MS SQL Management Studio

Java will send data the SQL database the same way a web from communicates with a database.

SQL is a language used by MS SQL Management studio to make changes to a table, Using SQL Quires, A new table can be created, Altred, Inserted Into, Updated and Quires can be created to get specific data from users.

Two type of information will be stored into the database.

Invirmental data and agent data, and this will be connected by time.

A relationship diagram was created to show how each table is going to interact with each other

C:\Users\n3178194\Downloads\Untitled Diagram (1).png

Figure 4.1‑2: Relationship Diagram

#### Table names and content

* Agents
  + Id – contains agents Identification number
  + DNA – the agents DNA
  + Time\_start\_id – time agent was born
  + Time\_end\_id – time agent died
  + Energy at the end – energy of agent at death
  + Total\_engery\_gained – total\_energy\_gained
  + Speed- agents speed
  + Colour – colour of agent
  + Food- what the agent ate
  + Date\_time – date and time information was received
* Time
  + Time\_id – how time is identified
  + Tick – time within Netlogo
* Environment actual
  + Environment\_id – Identfication number
  + Time\_id – Netlogo time
  + Session\_id – what session it was
  + Num\_turtles – the number of turtles at that time
  + Other – other environmental factors

This will be used to store all information from a session, the actual date and time a data is submitted because depending on the processing power of the computer and the amount of factors that are at work the speed of Netlogo will vary from session to session, by recording the actual time we can calculate the amount of time it took between two events.

### Visual Studio

Visual Studio can connect to MS SQL Server and with the correct authentication can access the database. This means another copy of the database nor is the data required for datamining.

Before a multi-dimensional data mining cube can be design actual data need to be gathered. Multi-dimensional data mining works best when there is a lot of data. Once a lot of data is gathered,

### Full integration

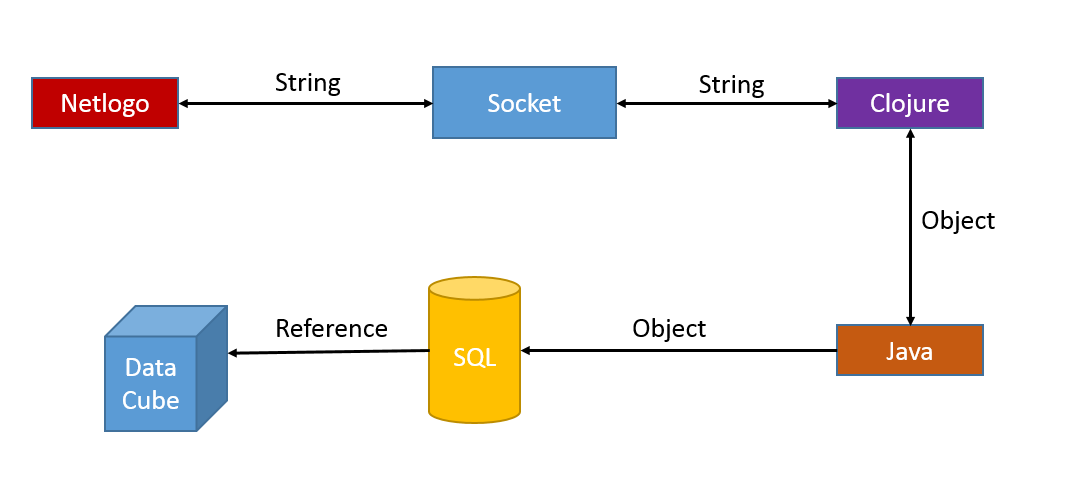


Figure 4.1‑3: Full integration Diagram

The above diagram shows all the connections between all component of this software and what data type is going to be used for the communication.

## Second iteration

### Netlogo

The Netlogo world will consists of two things, roads and grass. If agents can gain energy by eating grass then to force those to evolve there must be something negative in motion that will prefer one set of gene compared to another. Roads will get in the way of agents which means while they are on a road they cannot eat which means they cannot gain energy. Due to the surface area ratio between grass and road, this method is not going to effect the agents at all, therefore another set of factor is required to be added in.

To force the agents to evolve, when an agent is on a road a random number is generated that will determine the probabilities of that specific agent spontaneously dying as if it has been hit by a bar

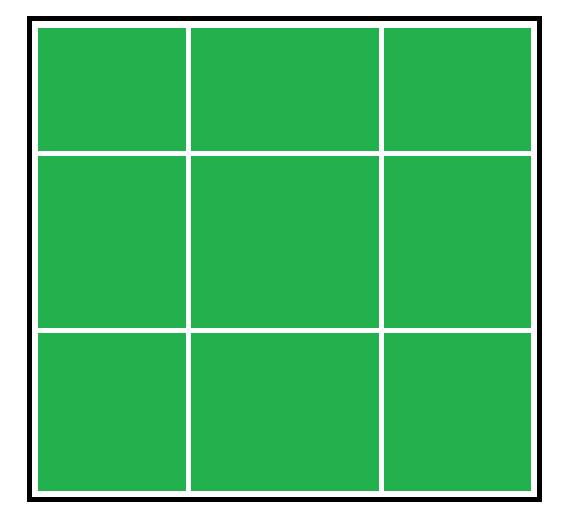


Figure 4.2‑1: Netlogo world design

# Implementation

## First iteration

### Communication Between Netlogo and Socket

The communication between Netlogo and Clojure was very simple to establish thank to the socket provided by Simon. Linch. With this socket information can be read or sent with just one line of code.

#### Reading from Socket



Figure 5.1‑1:Netlogo - Reading from socket

The above code simply reads the first thing on the stack and executes it as if a method is being called

e.g. if a method called crt.rabbits exisits and it takes in one number as an argument. When sending the message “crt.rabbits 3”, the crt.rabbits method is called with 3 as its argument.

#### Writing to socket

Reading from

Figure 5.1‑2:Netlogo- Writing to a socket

Writing to Socket is as simple as reading it. Whenever a message needs to be sent from Netlogo to Clojure the above method can be called with the message as an argument.

### Communication between Java and Clojure

After a lot attempts were done to achieve full communication between a java and a Clojure class, the mostly developed one being a project called ‘helpme’ where a Clojure class can call both static methods and methods on instance of an object.

Many different attempts were made by looking at suggested solutions from the website stackoverflow.com, reading Clojure for the brave and other websites on the subject ‘clojure – java interop’. As stated above only one way communication was established.

### SQL

While developing the database, a few changes were required to be made. Some of the spelling mistakes that was made in the design was corrected and unnecessary columns were removed. Full changes, including steps taken is available in (appendix: first iteration).

Once all the tables were created with appropriate data type, a relationship diagram was created assigning each tables with appropriate primary and foreign key

### Clojure and Netlogo

#### From Java to Clojure

Communication from Java to Clojure proved to be much more difficult than anticipated

#### From Clojure to Netlogo

To connect Clojure to Netlogo was very simple Thanks to S.Linch …

Using the wrapper was very simple and it was quickly tested using the REPL and Netlogo to confirm communication. There was some problems that occurred due to the Clojure environment.

##### Problem 1: Methods not recognised within REPL

A Clojure class called wrapper was created to handle all the communication between Clojure and Netlogo. All the necessary methods and imports were copied over to the class. In theory, this should have worked as it did on other project but when tested on the REPL, The REPL would not recognise the methods as shown in the following figure.

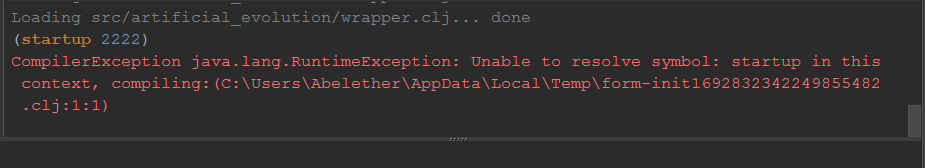


Figure 5.1‑3: Unable to resolve symbol Error when wrapper method was run.

Generally this would not be a problem when working on a Clojure project and can simply be solved by copying all the methods into the REPL and the REPL would recognise them but for this project it was not an option especially for it to be compiled and run as an application.

## Second iteration

### Netlogo

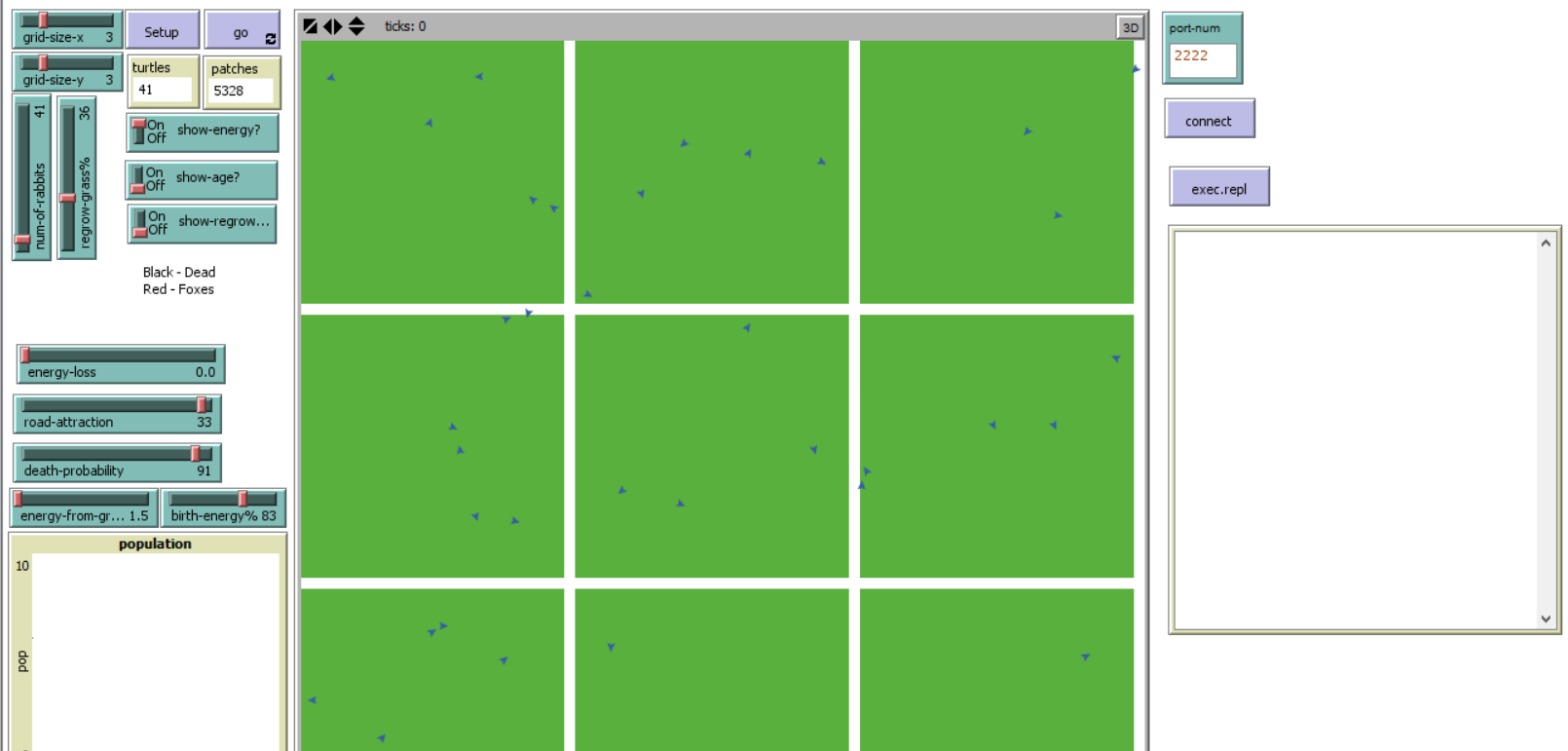


Figure 5.2‑1: Netlogo finished development

# Testing

## Testing in Clojure

Clojure has an REPL (read-eval-print loop) which makes testing interactive. All Clojure testing were done using the REPL. To first understand the use of REPL the website tryclj.com was used.

## Testing in Netlogo

For all Netlogo testing an Output Box was created, With in Netlogo messages can be printed in the output box just like print line in other programing languages.



Figure 6.2‑1: Output Box being used in Netlogo

In Addition to an output Box the command Centre is used to do additional tests. The command centre is used to run methods. The command centre is an REPL for Netlogo and certain quires can be run from the command centre, the figure below shows the result when ‘show count turtles’ is entered.



Figure 6.2‑2: Netlogo Command Centre

## First Iteration

You can see all evidence for Iteration one in Appendix 01

### Between Clojure and Netlogo

#### Establish connection

To test the communication between Netlogo, the REPL was used and within Netlogo an addition output window was created to assist in testing. And addition updates were made to the code within Netlogo.

The above code shows how communication is established between Netlogo and Clojure. The function (startup #portnumber) is used to create and advertise a socket. After A Netlogo model connects to the socket also using the port same port number, the message socket accepted with the object information is returned.

#### Sending message Between Clojure to Netlogo

Using the function (nlogo-send) a string is sent to Netlogo and no problem occurred while testing this. When sending a message the other way around as expected it was sent one word at a time, therefore from this point forward when sending from Netlogo to Clojure “:” must be used to separate words instead of space.

### Between Java and Clojure

The working part of the communication was tested using the project ‘helpme’ found in the appendix. Within the project helpme and the other projects, there are a number of classes created to understand how name spaces work.

The main classes that will be looked at are both in the helpme project and are called helpme.core.clj and jcore.java.

Existing Clojure libraries work when calling a function within them from java but once the name space is changed to one that was created in the projects scope it no longer works. Further research is required to understand how java and Clojure import classes and how packages and name spaces work to understand where the problem is coming from

## Second iteration

### Netlogo

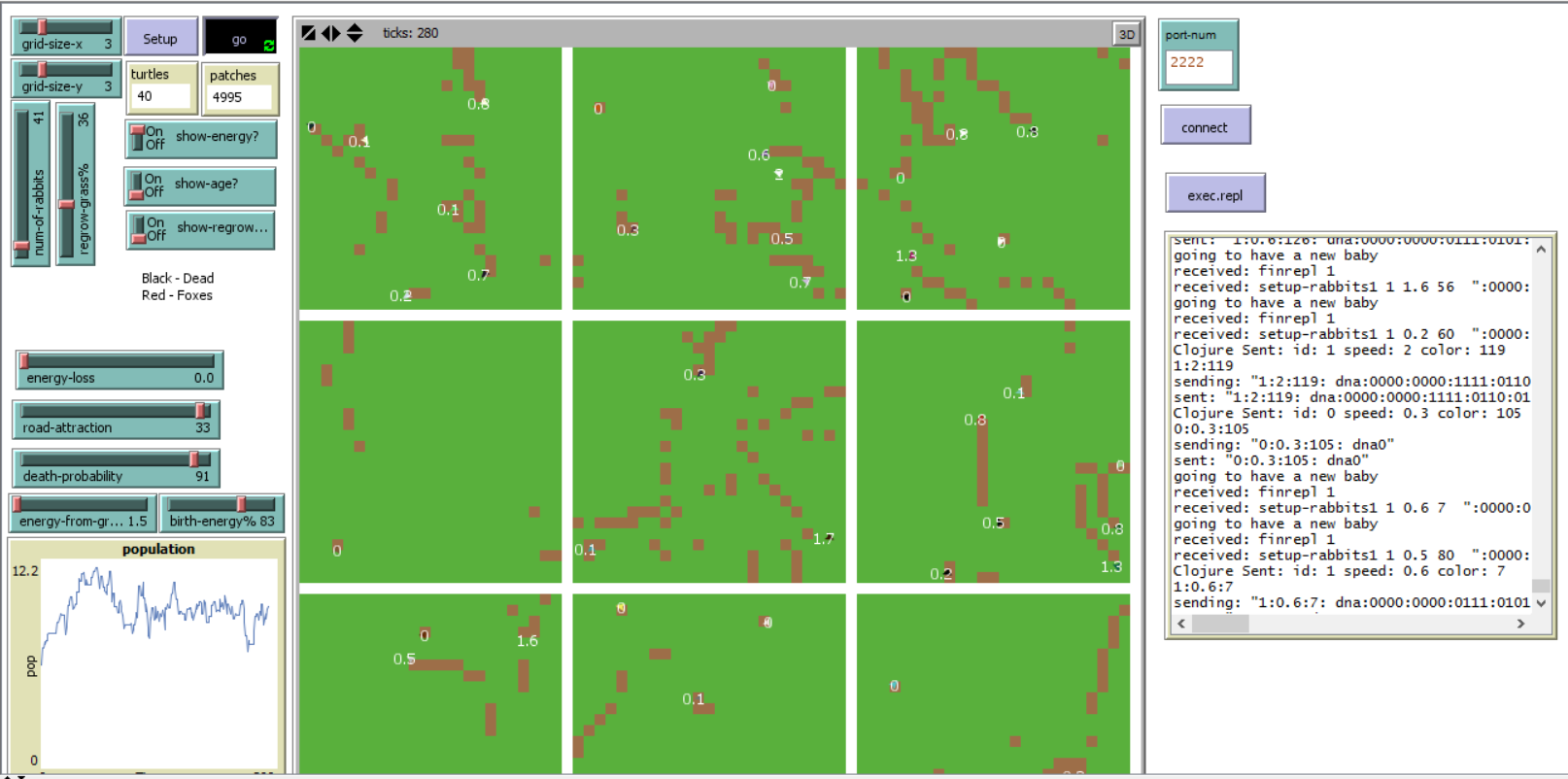


Figure 6.4‑1: Netlogo, Finished – testing

To test the Netlogo, it was imply run to see if it worked and if any errors occurred.

# Evaluation

## Iteration one

A day was allocated for Iteration one, which involved stablishing communication between java, Clojure and Netlogo. At the end this ended up taking over a week and the creation of the database was included in Iteration one to take my mind of the communication between Clojure and java.

Creating communication between Clojure and Netlogo was very simple because they connected thorough a socket and the socket was already provided by Simon. Linch in a previous project.

Communication between Clojure and java was proven to be impossible even though all the examples provided online seem to work. Unfortunately they all used pre-included libraries and it seem to work but when attempting with my own classes it didn’t. I believe this is heavily to do with my understanding of Clojure name spaces and how java import files.

Only after doing many research and understanding name spaces completely and how java worked was I able to call a java method from Clojure. By this point I have already moved on to creating the database and restructuring how different components will work with each other. I have also learned enough to start the project within Clojure which is what I did.

There was a few problems I came across while attempting to call a java method within Clojure. The first was that I can import a java class the same way I can on another java class but I could only access the static methods. This meant that I could not interact with any intense/objects. Serval projects were created to pin point where the problem was occurring but when it was run using the terminal the message I received is shown in figure 7.1-1.



Figure 7.1‑1: error when calling java class from Clojure

I asked Simon lynch if he has done this before and he suggested to pre compile the project and import it the same way. Looking at other forums a few of them suggested to pre compile, which I learned how to do using leiningen but the same problem still persisted.

Only after a week and a half I realised that all that was required was to create an instance of an object and assign it to a variable. I still don’t understand why because this was not required for pre-loaded java libraries nor was it required when using another java class.

At this point I was presented with two options, A. continue to work using Clojure, B. Start again using Java or C. Continue from where I was using Java. Ultimately I decided to go with option A because option B meant that all the work I had done was useless and time was not on my side as the project was due in almost in a week and a half. Option C was more reasonable but I would be sacrificing the structure of the project and it could result in the project quickly becoming over complicated and hard to understand what is happening. After many consideration I decided to continue using Clojure.

## Iteration two

## Scope of project

The scalability of the project was perfect due to the fact that it is easily scalable but I was a bit too ambitions with the level of work I wanted to produce given the amount of time I had.

## Provisional analysis

## Future improvements

### Code review

I was impressed by the results and what I managed to accomplish given the fact that I had almost no experience with Clojure

### Readability

### Reusability

### Structure

The structure of the class is one the first things I would change given more time. Current it only has basic structure created near the begging of the project. As the project progressed and the dead line time came close, I became more worried about the functionality than anything else. Because of this I ended up putting functions in a name space when it should be within a set of name spaces,

Due to lack of structure in some name spaces, a lot of files were imported in, this became a serious concern when a loop of name spaces was created.

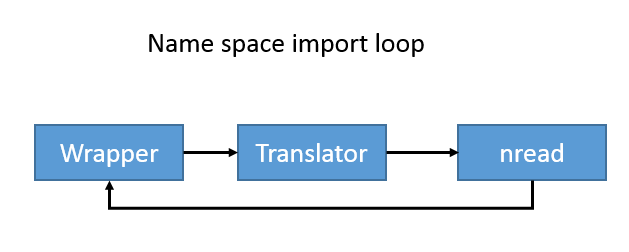


Figure 7.5‑1: Name Space Import loop

I was able to fix all name space import loops within the project but for next time steps need to be taken to avoid it by thoroughly designing the structure and when implementing a new future updating the structure first

### Features

#### Concurrency

Currently in Clojure there is only one thread that is created and each task is executed one after another. This is ok when there only a few agents running at a time but as the number grow the number of request coming per second grows with it. Also currently the product only perform mutation but when switchover is implemented the amount of processing power required triples. Once the data is also being added to a database, it is going to reduce the speed to a point where it is visible on Netlogo

#### Database

Even though the tables created are functional I believe it can still be developed more to increase efficiency and to be more abstract. The database is currently designed to store data for a specific simulation, if it was made to work more abstractly, a simple guide can be created to allow me or other people who are interested to create a completely different simulations based on evolution, using the same system, they would not be required to edit anything other than the database.

Depending on what framework is used to create the website the process might not be much different from the connection between java and the database. For example as long as I am using an object orientated programing language, most likely C# I would need to create an object that models the set of data I want to view and edit, and if using something like web forms I will be making changes by sending an SQL command over.

If using something like MVC all that is required is make a connection between the MVC application and the database and have the data model for each table. Entity framework will take care of everything else. It is even possible to recreate the database on the local sql server if a person would like to experiment with this system on their own machine.

Within this project scope a view is only created to test the connections between all the different tables but views can be expanded return most frequently requested sets of data. For example if a person were to be comparing the relationship between an agent and the total number of agents at the time, a view can be created a view can be generated to return agents with a total number of turtles in the same session, at the same time the agent existed. This would mean, this person would not have to create a query every single time they want to see those results, they can just use the view

#### Data Cube

#### Website for data

A basic website can be created to view results from the database. This is a simple feature that gives the results access to millions of people that are interested in this kind of data, it will a

lso raise awareness of the product communication process between the database and the website won’t be much different

#### Webserver to run simulations

#### Netlogo Model

The Netlogo model can be expanded to be more abstract to allow multiple type of environment to be used. For example currently the evolution system is reliant on this specific model. But if abstracted well a netlogo framework can be created where an environment can be designed on top. This would allow evolution on any type of agents or environments and even multiple type of agents at the same time.

## Social implication

Ethics issues

## Critical evaluation

Looking at the project and what I have manged to accomplish with in a space of three weeks using a completely new language. I am impressed at my accomplishment but I have found that some errors take me far too long simply because of lack of understanding of the subject, in the future instead of continually working trying to fix a single error, I will need to have a safe guard in place to make sure I am making continuous progress such as I could only try to fix an error for half an hour and if it is not fixed by then I must work on a different part of the project

### Initial Objectives

* Create a Netlogo environment where agents can move and interact with the environment.

Goal Achieved

Agents are free to move anywhere within the net logo world. The Netlogo environment consists of a field of grass with a grid of roads, the grid number can be adjusted. Agents interact with the road as if it was real and every second they stay on their the probability of them instantaneously dying increases

* Agents must be able to die and be able to do something to prevent them self from dying.
* Create communication between, Netlogo, Clojure and java.
* Create a java application where agents DNA is mutated.

### Overall Objective

* Produce a database where agents DNA can be stored.
* Allow the application to perform switchover when requested.

### Advanced objective

* Create a data cube to analyse the database.
* Create a website to allow anyone around the world access to the results found.

# Recommendations

# Conclusion

# List of Figures

[Figure 3.1‑1: Higginbotham, D. (2015) Clojure for the brave and true 8](#_Toc448122097)

[Figure 3.1‑2: Using Microsoft Word with git, Martin Fenner, (2014) 8](#_Toc448122098)

[Figure 4.1‑1: Communication between Netlogo and Clojure 11](#_Toc448122099)

[Figure 4.1‑2: Relationship Diagram 11](#_Toc448122100)

[Figure 4.1‑3: Full integration Diagram 11](#_Toc448122101)

[Figure 4.2‑1: Netlogo world design 12](#_Toc448122102)

[Figure 5.1‑1:Netlogo - Reading from socket 12](#_Toc448122103)

[Figure 5.1‑2:Netlogo- Writing to a socket 12](#_Toc448122104)

[Figure 5.1‑3: Unable to resolve symbol Error when wrapper method was run. 12](#_Toc448122105)

[Figure 5.2‑1: Netlogo finished development 13](#_Toc448122106)

[Figure 6.2‑1: Output Box being used in Netlogo 14](#_Toc448122107)

[Figure 6.2‑2: Netlogo Command Centre 15](#_Toc448122108)

[Figure 6.4‑1: Netlogo, Finished - testing 16](#_Toc448122109)

[Figure 7.1‑1: error when calling java class from Clojure 18](#_Toc448122110)

[Figure 7.5‑1: Name Space Import loop 20](#_Toc448122111)

[Figure 9.2‑1: Creating a new Database 27](#_Toc448122112)

[Figure 9.2‑2: Creating the agent table 28](#_Toc448122113)

[Figure 9.2‑3: Creating the agent session table 28](#_Toc448122114)

[Figure 9.2‑4: Creating the session table 28](#_Toc448122115)

[Figure 9.2‑5: Create a time' table 29](#_Toc448122116)

[Figure 9.2‑6: creating an environment table 29](#_Toc448122117)

[Figure 9.2‑7: creating keys and the relationship diagram 30](#_Toc448122118)

[Figure 9.2‑8: Relationship diagram with primary and forigen key view 31](#_Toc448122119)

[Figure 9.2‑9: Starting up Comunication between Clojure and Netlogo 32](#_Toc448122120)

[Figure 9.2‑10: Sending message from Clojure to Netlogo 32](#_Toc448122121)

[Figure 9.2‑11: Sending message from Netlogo to Clojure 32](#_Toc448122122)

[Figure 9.2‑12; Sending message from Netlogo to Clojure with ":" 33](#_Toc448122123)

[Figure 9.3‑1: update for database relationships 34](#_Toc448122124)

[Figure 9.3‑2: Database view created, to test connection 35](#_Toc448122125)

# References

(2006) Available at: https://www.nlm.nih.gov/visibleproofs/education/dna/dna.pdf (Accessed: 30 March 2016).

2016 (2000) *An overview of DNA functions*. Available at: http://www.exploredna.co.uk/an-overview-dna-functions.html (Accessed: 5 April 2016).

Abernethy, M. (2010) *Data mining with WEKA, Part 1: Introduction and regression*. Available at: http://www.ibm.com/developerworks/library/os-weka1/ (Accessed: 16 October 2015).

Chris Lam (2014) *How to use GitHub for beginners*. Available at: https://www.youtube.com/watch?v=E8TXME3bzNs (Accessed: 20 March 2016).

*Clojure: Call java method* (2014) Available at: http://xahlee.info/clojure/clojure\_calling\_java.html (Accessed: 31 March 2016).

*DNA structure and replication* (no date) Available at: https://www.khanacademy.org/partner-content/crash-course1/partner-topic-crash-course-bio-ecology/crash-course-biology/v/crash-course-biology-110 (Accessed: 5 April 2016).

Fenner, M. (2014) *Using Microsoft word with git*. Available at: http://blog.martinfenner.org/2014/08/25/using-microsoft-word-with-git/ (Accessed: 20 March 2016).

Grimes, A. and contributors, numerous (2011) *Try Clojure*. Available at: http://www.tryclj.com/ (Accessed: 10 April 2016).

Hickey, R. (2008) *Values and change: Clojure’s approach to identity and state*. Available at: http://clojure.org/about/state (Accessed: 3 April 2016).

Higginbotham, D. (2015) *Clojure for the brave and true: Learn the ultimate language and become A better programmer*. United States: No Starch Press,US.

*In Clojure 1.4 what is the use of refer within require?* (2016) Available at: http://stackoverflow.com/questions/10358149/in-clojure-1-4-what-is-the-use-of-refer-within-require (Accessed: 31 March 2016).

Katamreddy, S. (2014) *NetBeans IDE and Intellij IDEA from the eyes of a Long Time eclipse user - DZone java*. Available at: https://dzone.com/articles/netbeans-ide-and-intellij-idea (Accessed: 24 March 2016).

*Machine Learning Group* (no date) Available at: http://www.cs.waikato.ac.nz/ml/index.html (Accessed: 16 October 2015).

MathsIsFun (2014) *Combinations and Permutations calculator*. Available at: https://www.mathsisfun.com/combinatorics/combinations-permutations-calculator.html (Accessed: 5 April 2016).

*NetLogo* (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/NetLogo (Accessed: 20 March 2016).

Nick Bostrom & Carl Shulman (2012) *How hard is artificial intelligence? Evolutionary arguments and selection effects*. Available at: http://www.fhi.ox.ac.uk/ai-evolution-and-selection-effects.pdf (Accessed: 11 April 2016).

*Unary numeral system* (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Unary\_numeral\_system (Accessed: 4 April 2016).

Wilensky, U. (1999) *NetLogo*. Available at: http://ccl.northwestern.edu/netlogo/. . Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. (2003) *Netlogo Traffic Grid model*. Available at: http://ccl.northwestern.edu/netlogo/models/TrafficGrid. .

mybringback (2013) *How to use git and GitHub 1: Install git and basic commands*. Available at: https://www.youtube.com/watch?v=tRTckrrCME4 (Accessed: 20 March 2016).

**Citations, Quotes & Annotations**

Grimes, A. and contributors, numerous (2011) *Try Clojure*. Available at: http://www.tryclj.com/ (Accessed: 10 April 2016).

"Try Clojure REPL" (Grimes and contributors, 2011)

# Appendix

Appendixed files can be found in //Artificial-Evolution/artificial\_evolution/doc/Appendix

## Appendix 0

## Appendix 01 – First iteration

### Implementation

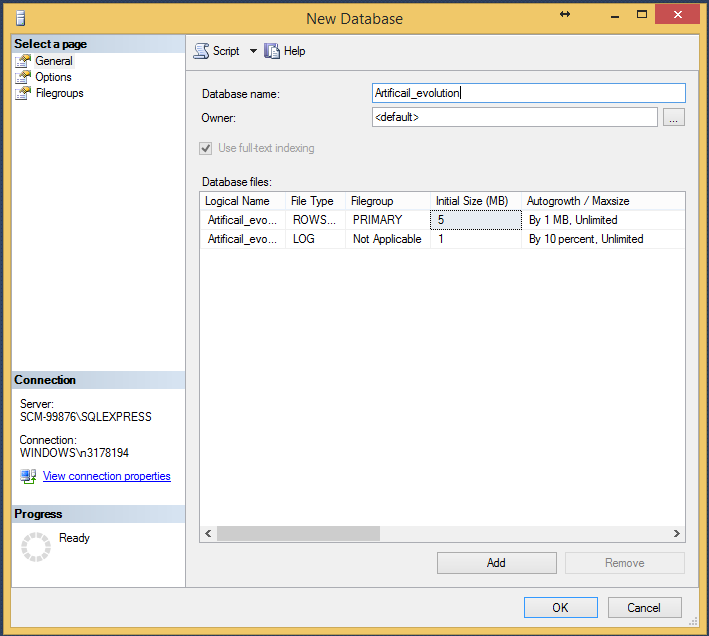


Figure 9.2‑1: Creating a new Database

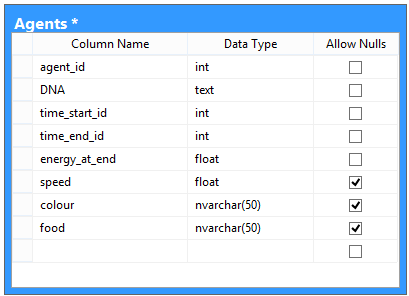


Figure 9.2‑2: Creating the agent table

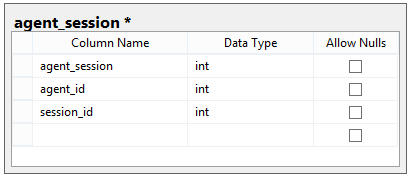


Figure 9.2‑3: Creating the agent session table

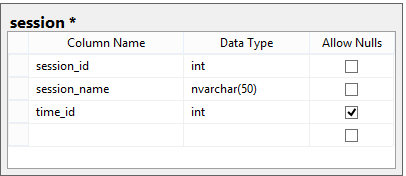


Figure 9.2‑4: Creating the session table

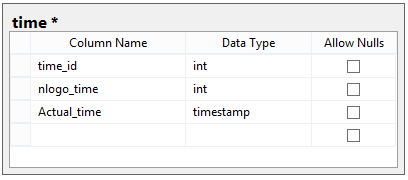


Figure 9.2‑5: Create a time' table

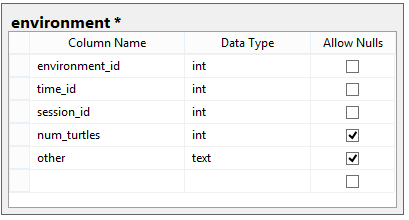


Figure 9.2‑6: creating an environment table

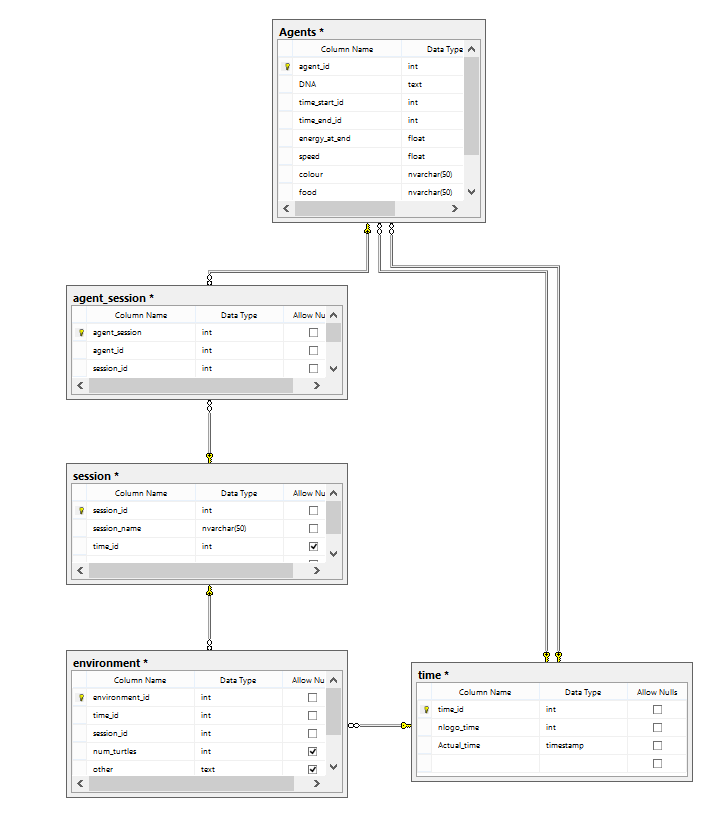


Figure 9.2‑7: creating keys and the relationship diagram

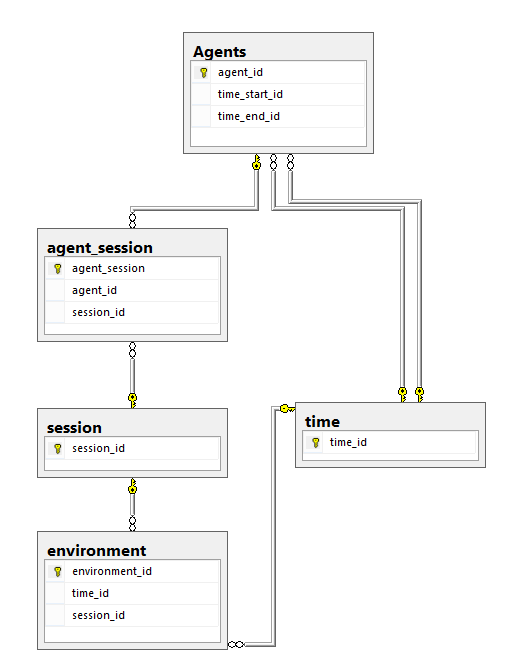


Figure 9.2‑8: Relationship diagram with primary and forigen key view

### Testing



Figure 9.2‑9: Starting up Comunication between Clojure and Netlogo



Figure 9.2‑10: Sending message from Clojure to Netlogo



Figure 9.2‑11: Sending message from Netlogo to Clojure



Figure 9.2‑12; Sending message from Netlogo to Clojure with ":"

## Appendix 02 – From Iteration 02

### Implementation

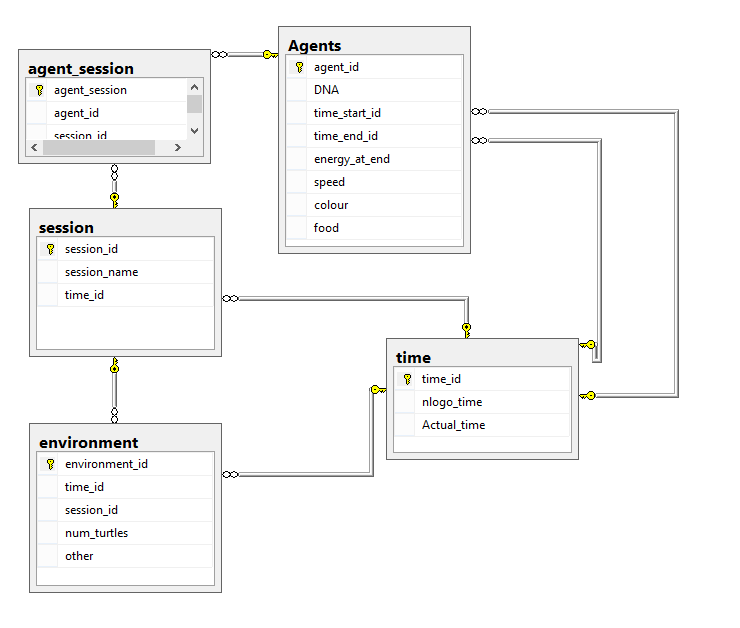


Figure 9.3‑1: update for database relationships

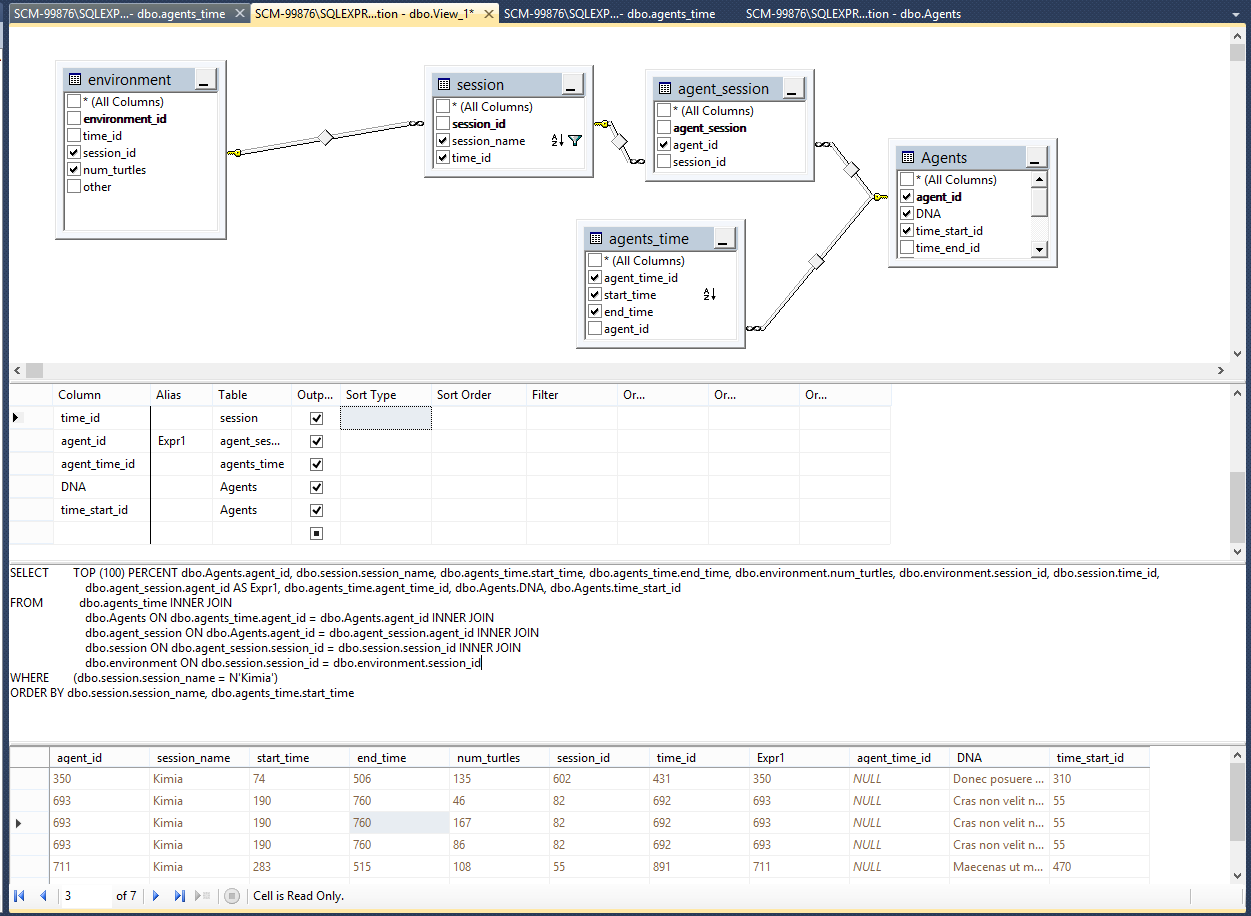


Figure 9.3‑2: Database view created, to test connection

## Appendix 03

//