



University
of Glasgow | School of
Computing Science

Darwin: A Genetic Programming Framework for Cloning Web Services

George Kouzmov

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 31 January 2014

Contents

1	Introduction	5
1.1	Genetic Programming	5
1.2	Web Services	7
1.3	Problem	8
2	Context	9
2.1	Related Work	9
2.2	Organization	10
2.2.1	Agile Development	10
2.2.2	Requirements	11
2.2.3	User Stories and Sprints	12
2.2.4	Meetings and Minutes	13
3	Design	14
3.1	Multiple Technologies	14
3.1.1	RESTful Web Services	14
3.1.2	Genetic Programming Framework	15
3.1.3	Code Inspection and Generation	15
3.1.4	Graphical User Interface and XML	16
3.2	Architecture	16
3.2.1	Overall Design	17
3.2.2	Client	19
3.2.3	Evaluator	19

3.3	GUI Design and XML	20
3.3.1	XML Configuration	20
3.3.2	Purpose of the GUI and Iterations	20
4	Implementation	23
4.1	Technologies	23
4.1.1	DEAP	23
4.1.2	CherryPy	23
4.1.3	Inspect	24
4.1.4	Requests	24
4.1.5	wxPython	24
4.2	Implementation	25
4.2.1	DEAP Set Up	25
4.2.2	Generation of Individuals	27
4.2.3	Evaluation Over a Network	27
4.2.4	Code Generation	28
4.2.5	GUI	30
5	Evaluation	31
5.1	Unit Testing	31
5.2	System Testing	32
5.3	UI and Usability Testing	32
6	Problems Encountered and Future Development	34
6.1	Problems	34
6.2	Future Development	35
6.2.1	Improvements	35
6.2.2	New Functionality	35
7	Conclusion	36
7.1	Contribution	36

List of Figures

1.1	An example of a genetic programming individual that represents $x+y+z$	6
1.2	An example of a crossover between two individuals and their children as result	6
1.3	An example of a mutation applied to an individual	7
1.4	Simple diagram explaining how clients communicate with webservices	8
2.1	Velocity graph showing the sprints and the amount of work based on tickets completed in different iterations during first and second sprint	10
2.2	PivotalTracker and the way it is used by the project	11
2.3	The minutes format and what was discussed in them	13
3.1	Comparison between five of the chosen webservices	15
3.2	Top level view of sprint one and two. Sprint one is focused on evaluating an individual over the web and sprint two focuses on cloning a web services	16
3.3	Lower level component diagram showing the functionality and usage of each class/component .	18
3.4	Example of a dialogue used for configuration of projects used in Eclipse editor	21
3.5	One of the iterations of the GUI showing all the possible parameters	22

List of Tables

2.1	Sprint one and sprint two of the framework	11
2.2	Basic requirements for the project’s technologies	12
2.3	Sprint one and sprint two of the framework	12

Chapter 1

Introduction

In recent years web services have become a common method of communicating between applications and accessing needed information, creating distributed environment for many developers. Due to the standardized communication used by the web services they are expected to be used even more in the future. The nature of web services is a black box system which means we are unfamiliar with its contents and we only know how to access it and what it returns.

The central goal of this project is to produce a tool for copying a web services using genetic programming, a proof of concept project that reached very far. Genetic programming involves generating programs based on a set of functions and then evaluating the quality (correctness) of the program via a fitness function. To prove that cloning of a web service can be achieved using genetic programming we used a symbolic regression problem where the genetic program needs to find a mathematical expression that yields the same result as a given one from a fitness function. But instead of using our own fitness function we use a real web service hence generating a program that has the same functionality as the web service. Proving this meant that theoretically, by increasing the complexity of our function set we can achieve copying of an even more complex web service.

1.1 Genetic Programming

Genetic programming is a branch of artificial intelligence inspired by biological evolution. It is a machine learning technique that optimizes a population of computer programs until they start performing the user specified functionality. Like the more widespread genetic algorithms (GA), genetic programming uses an evolutionary algorithm based methodology to iterate through solutions until a correct one is found. In genetic programming a predefined size **population** is iterated through a user defined number of **generations**. The population consists of **individuals** which are programs represented as a tree structure. This tree structure is constructed from a predefined **primitive set** of functions and **terminals**. The terminals represent the bottom nodes of the tree, they can be both arguments and constants. An individual can be seen in figure 1.1 A genetic program needs a **fitness function** through which each individual is evaluated and compared. This way the correctness of the program can be calculated. Usually there are two type of fitness function minimum and maximum where minimum is looking for individuals with fitness values closer to 0 (where 0 usually means that a solution is found) and maximum for the individuals with highest fitness result.

Genetic program works by randomly generates individuals for the first generation evaluating them through the fitness function and creates new programs through **crossover** and **mutation**. Crossover is used to combine the

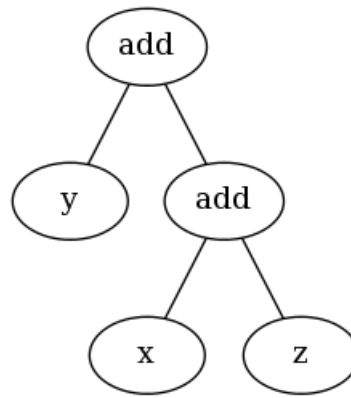


Figure 1.1: An example of a genetic programming individual that represents $x+y+z$

genetic information of two individuals which means switching the nodes of a tree of one individual with another as seen in figure 1.2. With a tree-based representation replacing a node means replacing the whole branch. When mutation is applied to an individual a single node is replaced with a new randomly generated one again changing a single node can mean changing the whole branch shown in figure 1.3. After crossover and mutation are applied to the population a new generation that consists of the children of the old population, individuals from the old population and mutated individuals is created hence the iterations continue until the generations are completed. A result may not be found in the end of the generations. The result of a genetic program is a heuristic, a stochastic method - no guarantee of success.

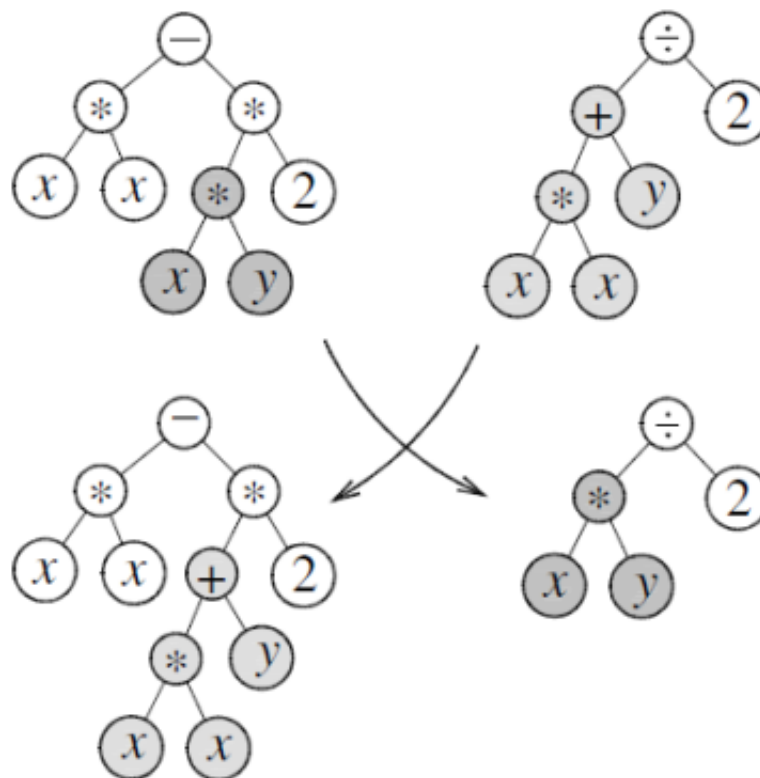


Figure 1.2: An example of a crossover between two individuals and their children as result

One of the problems with genetic programming is searching the correct parameters to solve a specific problem. For each problem a specified number of populations, generations needs to be provided also the percentage of the population to which crossover and mutation are going to be applied. In more advanced cases the types of

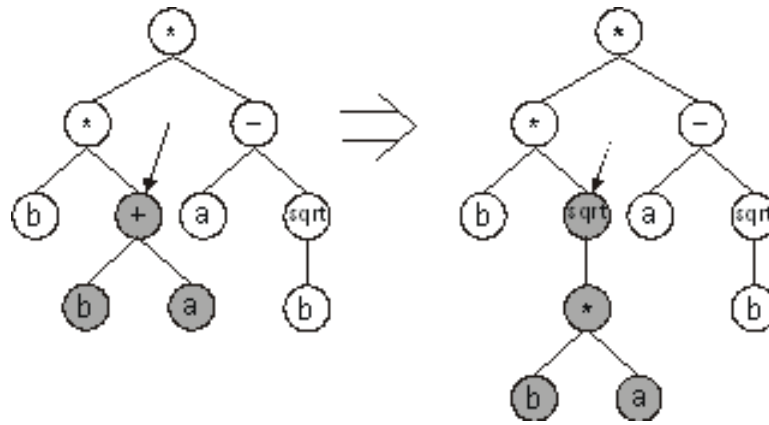


Figure 1.3: An example of a mutation applied to an individual

mutation, crossover and fitness function algorithms used need to be specified. Another complication is choosing the correct fitness function to evaluate the correctness of the individuals. If an incorrect fitness function is defined an individual with seemingly good fitness value may be far from correct. The primitive and terminal sets need to be chosen correctly as well. If too many primitives are chosen the generations might not be enough to find a solution to the problem because we are expanding the search boundaries. However if small sets are chosen we are limiting the search boundaries or the individuals won't have a sufficient primitive set to satisfy the required functionality.

Using this methodology multiple problems can be solved. A common and a simple one is symbolic regression. In the case of symbolic regression an individual represents a mathematical expression and is evaluated against another expression specified in the fitness function using the primitive set of operators available. In this project we are using this technique as a proof of concept for the capabilities of the framework.

1.2 Web Services

Web services are one of the main methods to provide utility computing. A definition for a web service is

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format. [17]

They represent an easy way for developers and clients to access and compute information. Usually through an API users are able to communicate with a web service although sometimes they need to be accessed with raw URL or HTTP request as seen in figure 1.4. Unlike traditional client/server models, such as a Web server/Web page system, Web services do not provide the user with a GUI. Web services instead share business logic, data and processes through a programmatic interface across a network. The applications interface, not the users. Developers can then add the Web service to a GUI (such as a Web page or an executable program) to offer specific functionality to users. Web services hold many benefits for developers. They represent components on the web that can be plugged in any project or software so its functionality can be used. It improves communication between companies and users because information is distributed much faster. A good example of a web service is of Google who provide an API to their GPS maps which when given longitude and latitude return a map with the location of the user. This service has improved the user experience for many devices and because of its good implementation has made the implementation of other similar services almost obsolete.

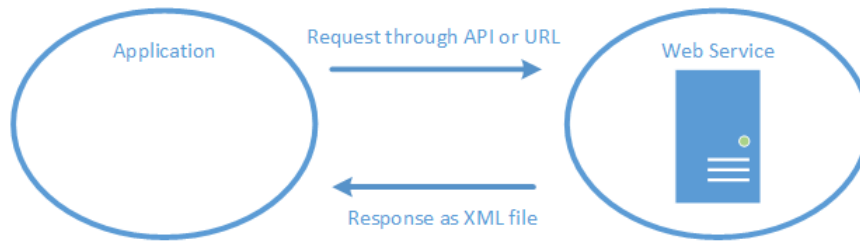


Figure 1.4: Simple diagram explaining how clients communicate with webservices

A problem with web services are that if they go offline or they have a bug, they affect the whole network and every user using them. Every non-open sourced framework is essentially a black box and the way it uses its parameters is unknown. However these problems are minor to the overall benefit of the web services. Because of the way the internet works the amount of web services has essentially created a distributed system with multiple components. Developers can integrate complex systems easier than before and can easily plug in additional functionality as long as the right web service is there

1.3 Problem

The purpose of the project is to develop a framework that can clone a web service. Theoretically it is possible to complete this through genetic programming. Initially the project was separated into three steps - evaluate individuals over a network, clone and existing web service and improve a web service. The project aims to clone an external web services and provide a GUI to generate the configuration files needed for the framework to work.

When using an external web service we know the parameters we are giving to it, we have an idea of the way it works and we know the results. Combining this knowledge meant it was possible to use an external web service as a fitness function to our genetic program and a way to specify the primitive set used. After part two of the project was completed it was proved that it is possible to clone the functionality of a web service. Because it is a proof of concept project a simple web service was used in order to verify the expected result. However the project provides evidence that with the right choice of primitive set it is possible to clone the functionality of even more complex web services.

Chapter 2

Context

A key part of the project was organization and project management. There were many factors and issues that needed to be considered before development could start. Research had to be done to identify other similar projects such as *Gen-o-fix: an embeddable framework for dynamic adaptive genetic improvement programming* [8]. Technologies had to be considered since multiple approaches to the implementation were available. Frequent meetings and iterations over requirements were key so the goals could be identified. Software development methods including agile development had to be used to help structure the project, focus on the key goals and important features so sprints could be set up and development could be as efficient as possible.

2.1 Related Work

Before finalizing the idea of the project, research had to be done so previous related work can be found and compared. The project has a very specific scope and there aren't many projects working towards a similar goal using genetic programming. The projects were *Gen-o-fix: an embeddable framework for dynamic adaptive genetic improvement programming* and *A Genetic Programming Approach to Automated Software Repair* [16].

A Genetic Programming Approach to Automated Software Repair makes project to repair bugs in off-the-shelf legacy C programs. To achieve that the framework does three main modifications to the standard GP technique:

1. genetic operations are localized to the nodes along the execution path of the negative test case;
2. high-level statements are represented as single nodes in the program tree;
3. genetic operators use existing code in other parts of the program, so new code does not need to be invented.

Initially *Darwin* was supposed to include similar functionality as well but it was later excluded and kept for future development.

Gen-o-fix explores the concept of genetic improvement programming (GIP); it's goal is to automate software maintenance. *Gen-o-fix* is a GIP framework which allows a software system hosted on the Java Virtual Machine to be continually improved (e.g. make better predictions; pass more regression tests; reduce power consumption). Similarly to *Gen-o-fix*, *Darwin* is also focused on creating a user-centric tool rather a research-centric one.

Both of the frameworks focus on post-development processes like bug fixing, code improvement and code maintenance. These ideas are different from *Darwin*'s one of cloning web services however the concept for creating and modifying existing code through genetic programming is observed in all three of the project. This made the papers a good source of information and approaches used for the development of *Darwin*.

2.2 Organization

Frequent team meetings, proper software development method and good project management were key for the success of the project. There were many unknowns in the beginning and as an inexperienced developer into the field of genetic programming there were many basics that needed to be covered. Through this basics it was later possible to set the requirements, sprints and organize a project around them.

2.2.1 Agile Development

Agile development was chosen for the project. In order to maintain a good project management a tool for agile development was needed. Github [7] was the version control used and PivotalTracker [12] was the agile tool of choice.

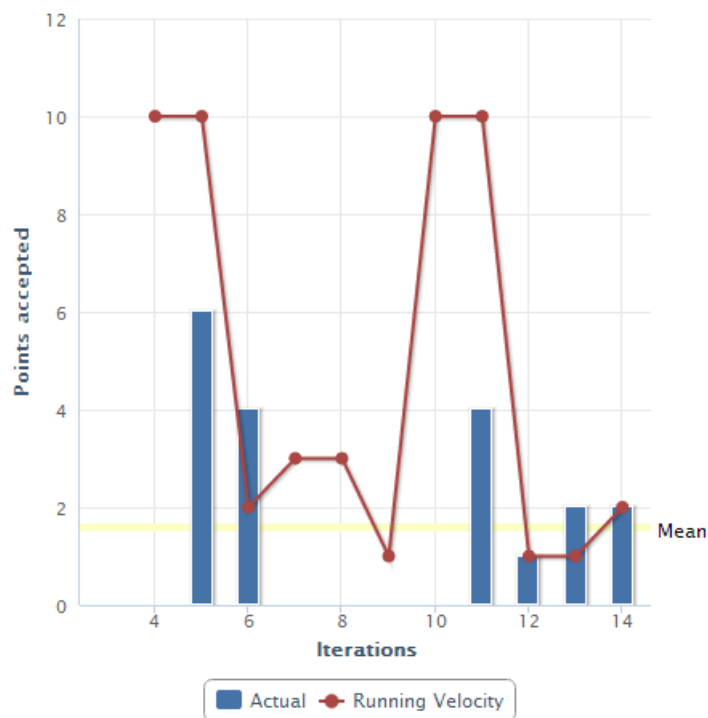


Figure 2.1: Velocity graph showing the sprints and the amount of work based on tickets completed in different iterations during first and second sprint

PivotalTracker is a web based tool where all the development process was stored, place to put all the user stories, functional and non-functional requirements, organize them and modify them easily. In the system it was easy to see when were the development bursts for the sprints and when was the post development time, when bugs were fixed, code was re-factored as seen in figure 2.1. This way the client can monitor how the actual

development is going when was I working and on what feature. Pivotal tracker was used to monitor the progress of the project as seen in figure 2.2.

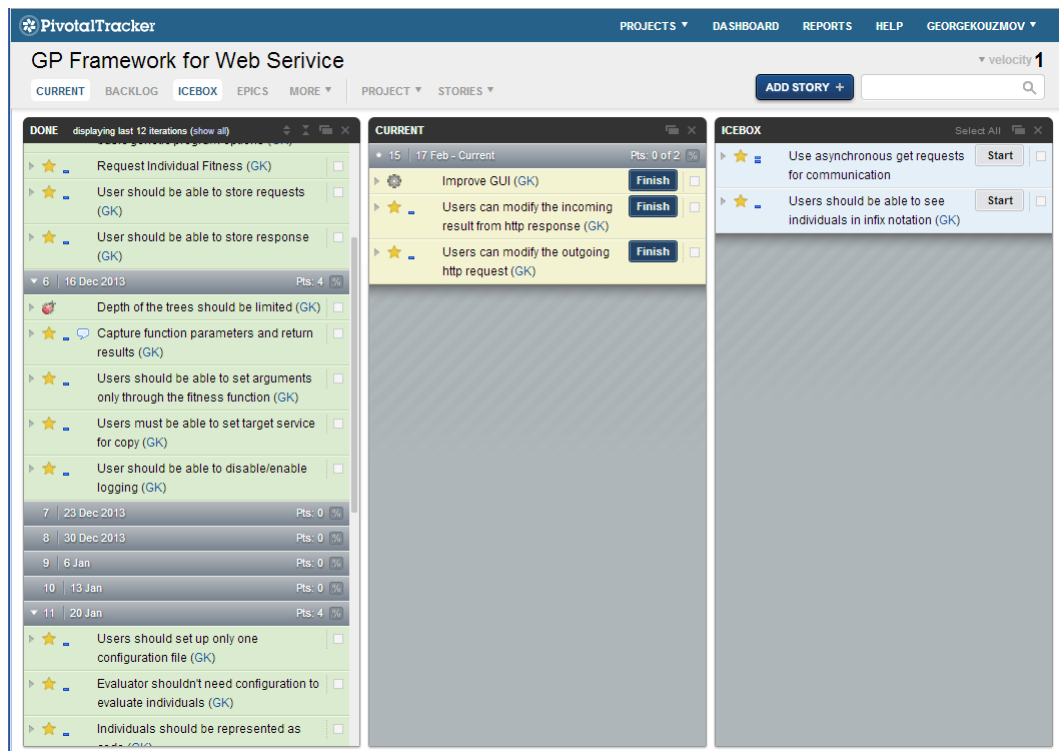


Figure 2.2: PivotalTracker and the way it is used by the project

Git hub is the version control tool used for the project. As a public code repository it gave the opportunity to present the code easily to the client and others as well. Frequent commits were done during the sprints. This way when the code broke during a sprint it could be reversed back to a working version.

2.2.2 Requirements

For requirements engineering an iterative approach was chosen. Through the first several meetings different ideas and goals were considered until requirements were finalized. The MoSCoW [5] method was used to reach a common understanding with the client about the importance of each requirement and how relevant it was to the project. MoSCoW method involves grading each requirement for how important it is for the project using the technique: (M - must, C - could, S - should, W - won't).

Table 2.1: Sprint one and sprint two of the framework

Sprint One GP to create a web service	Sprint Two - GP to clone a web service
1. Evolve a web service - M	1. Integrate with DEAP - M
2. Fitness evaluation over REST - M	2. Incorporate unit tests - C
3. Basic GP configuration - M	3. Start from existing web service - S
4. Define function set - M	4. Test through interpreter - S
5. Pre-set function set - S	5. Use original web service as an "Oracle" - M
6. Generate python code of the best solution - M	-
7. Manipulate python code - S	-

However later on because of the project complexity and the time frame the third sprint was changed from

improving a web service to creating a graphical user interface for easier configuration of the framework. Even though requirements were set, small changes were done through out the development process to suit the current situation and improve the software product. Requirements are displayed in table 2.1

Table 2.2: Basic requirements for the project's technologies

Basic Requirements
1. Must be implemented in python
2. Must use a REST framework (CherryPy)
3. Develop or choose a genetic programming framework (DEAP)

Before the requirements for each sprint were built there were basic requirements that needed to be identified as seen in table 2.2. Framework for genetic programming and a REST framework had to be chosen. Weeks of research were dedicated to find the most suitable frameworks. The only initial requirement was that the project should be done in python and later on after the client agreed with the technologies used they were added to the basic requirements. Knowing and understanding the technologies that would be the building blocks of the project helped guide the writing of the sprint requirements later on.

2.2.3 User Stories and Sprints

As mentioned the project is focusing on a user-centric tool. A list of user stories was created for the client explaining the way he and any potential user was going to interact with the framework. The list of stories was later iterated until there were set of user stories defining each of the sprints as seen in figure 2.3.

Table 2.3: Sprint one and sprint two of the framework

Sprint One
User should be able to disable/enable logging
User should set up only one configuration file
Users should be able to see individuals in infix notation
Users should be able to set arguments only through the fitness function
Sprint Two
User must be able to set target service to copy
User should be able to graph individuals
User should configure the framework through XML
Sprint Three
Users should chose from pre-set primitive functions
User should be able to generate XML file through a GUI
User should modify the incoming result from http response
User should modify the outgoing http request

1. **Sprint One** was focused on evaluating individuals remotely
 - (a) Configuration of basic genetic programming parameters.
 - (b) Create a class for defining and reading of primitive set.
 - (c) Converting DEAP individual to python code for easier evaluation.
 - (d) Create a base for cloning webservices.

2. **Sprint Two** was focused on cloning a web service.
 - (a) Use existing web service as a fitness function.
 - (b) Automation of configuration.
 - (c) Use a single URL for selecting the target webservice.
3. **Sprint Three** aimed for creating a more interactive and easy way to configure the system.
 - (a) Use of XML file to configure the framework.
 - (b) Create GUI for generating the XML file.
 - (c) Providing request and response handlers for handling communication with webservices.

2.2.4 Meetings and Minutes

The core purpose of the meetings was to iterate through the development process with the client. Initially it was only explanation of the project, genetic programming, possible technologies and identify what can be developed for the time frame set for the project. Every week since week two there has been a project meeting. They were helpful for me and for the client so we can clear the ideas that we had, form them into one and work towards the same goal. The client's expertise in genetic programming and frequent meeting helped me understand how it works in little time so it was possible to begin development. Since day one minutes were introduced to follow

```
Done:
Create Diagram
Write User Stories
Look at AST
Look at SymbRegression Problem

TODO:
-Send code
-Fitness evaluation is it max or min function
-Look at PyDev
-send minutes
-Improve Diagram
-Improve User Stories
Draft Report
Blogging
  -Write about DEAP
  -Write about architecture
  -Write about problems with ast
```

Figure 2.3: The minutes format and what was discussed in them

the development of the project. This included reading papers, learning concepts, testing technologies. Minutes became the main way of following the project progress and a way for setting small goals for the week. Minutes also shaped the meetings by being able to discuss step by step the goals set from previous time. Every meeting had the same structure, it was discussed - what was done, what were the problems, new issues were addressed and possible solutions. The format of the minutes is shown in figure 2.3

Chapter 3

Design

When designing the framework the key principle was to provide a simple way to control the key parameters in a genetic program. The scope of the project is quite big and so is its complexity so a key design feature was modularity. Component based software engineering practices were used throughout the project, helping to organize the multiple technologies used.

3.1 Multiple Technologies

Among the key issues in the development was combining all the technologies used in the project.

- genetic programming framework
- REST based web service
- XML parser
- GUI
- code inspection
- code generation

For each component that was needed there were multiple options available so the best ones needed to be chosen which meant trying out pack of frameworks and libraries.

3.1.1 RESTful Web Services

RESTful web services were needed for creating a service that can evaluate individuals over a network. Simplicity was important for the framework we are choosing. WebPy [18], CherryPy [2], Flask [6], Bottle [1] and Django [4] are commonly used frameworks. Each was compared as seen in figure 3.1 to identify the best candidate. However CherryPy and Django are the only frameworks that handle POST and GET request correctly, without any problems. In order to fix this issues with WebPy, Flask [6] or Bottle [1] additional libraries would have been needed. It was desired to avoid more dependencies because they would just make the project more complex. That meant the choice was between two. The more lightweight choice was CherryPy but that wasn't the only reason why the project was chosen. A potential deployment platform was the Glasgow Raspberry Pi Cloud [10]. Because CherryPy provided support for Raspberry Pi [11] it was the obvious choice.

Webservice	Quality Documentation	Lightweight	Correct POST/GET Handling	Raspberry Pi Support
Flask	✓	✓		
Bottle		✓		
Django	✓		✓	
WebPy		✓		
CherryPy	✓	✓	✓	✓

Figure 3.1: Comparison between five of the chosen webservices

3.1.2 Genetic Programming Framework

Genetic programming frameworks didn't have as many choices as RESTful web services however there was the option of implementing one. Pyevolve [13], pystep [15], pygene [14] and DEAP [3] were the only choices for genetic programming frameworks in python. None of them was minimalistic and simple enough, however implementing a genetic programming framework seemed a more complicated task that could slow down the project and its final result. The most common and supported one was the DEAP framework that also had really good documentation, tutorials and examples making it the correct choice. However as a future development option for the project remains creating a genetic programming framework related to the project.

3.1.3 Code Inspection and Generation

Code inspection and generation was needed in order to achieve automation. Code inspection is used to extract source and information from code objects and generation is used to create new code objects and execute them in the scope of the program. For inspecting code the best solution was the *inspect* library which provided a way to extract source and other parameters from functions. It is a library that comes with python, has great documentation, examples and tutorials which proved positive for the speed of development since not a lot of time was spend learning the inspect library.

Among one of the problems with using DEAP was it didn't support conversion between individuals and python source code or AST (Abstract Syntax Tree). Representing an individual as raw code or an AST meant that the evaluating web service can be simplified even more. Using AST was very compatible with the way individuals are represented in genetic programming and it was more robust and safe way of doing code generation rather than using raw string. However the more simple choice = raw strings, was selected due to pressure of time. This suggested the idea of an open source contribution to the DEAP framework for generating python source and AST for a DEAP individuals.

3.1.4 Graphical User Interface and XML

Since the system is working with many parameters configuring them needed an abstraction. Using XML files for parsing parameters was the choice of abstraction. A parser was created using lxml [9] and connected with the graphical user interface. Graphical user interface was the next step in project simplification. Its role was to generate XML files for the user with the ease of controlling all the parameters needed. For creating GUI in python there are several good choices Tkinter, PyQt, wxPython. I've had prior experience with Tkinter and I wasn't excited about working with it again. It looked much more complex compared to the other two. In sites such as StackOverflow the community was supporting wxPython. An additional benefit to using wxPython was the fact that there were a lot of good tools for generating layout. However, my previous experience with GUI frameworks like Swing and its similarities with wxPython, gave me the advantage of understanding wxPython quickly and writing a decent GUI in no time. In the end I didn't use any of the layout generating tools like wxGlade, wxDesigner or DialogBlocks because learning to use them was going to cost almost the same time as learning wxPython.

3.2 Architecture

A top down approach was applied to identify the major components in the system. The client's requirements were the starting point for identifying those components. They were clear and simple enough to create a high level view of the framework, with which to explain the framework's basic behaviour and how it's going to work see figure 3.2.

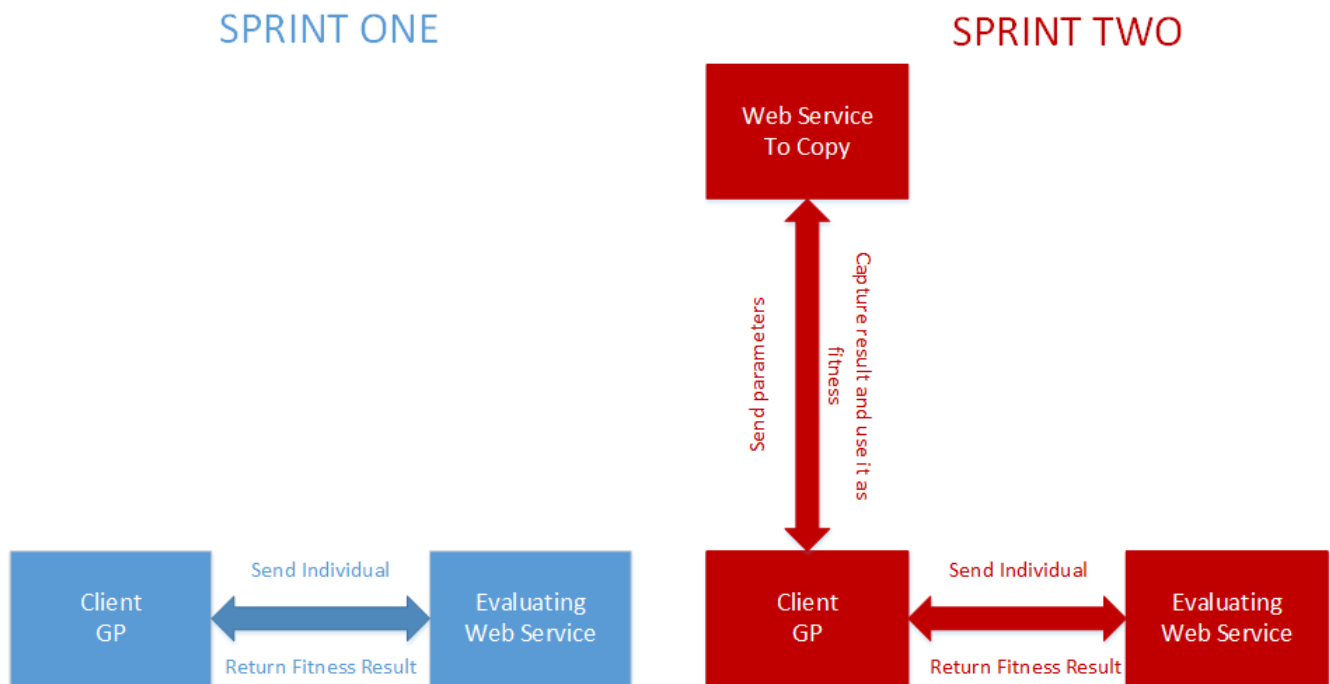


Figure 3.2: Top level view of sprint one and two. Sprint one is focused on evaluating an individual over the web and sprint two focuses on cloning a web services

By displaying the way the high level components communicate it was later possible to identify their functionality and split them into lower level components or classes. This way high cohesion and loose coupling was achieved at every level.

3.2.1 Overall Design

After the high level components were designed, the user stories helped create a more detailed view of them. At that point of time the project focused on sprint one, which meant focusing on evaluating individuals over a network. User stories helped identify how user was going to approach the framework hence it was able to create components with known input and return parameters as seen in figure 3.3

Even though Python2.7, unlike Python3.3 is a semi object language, for better use of software engineering practices and better ordering of the project, objects were used to represent the different components. The diverse capabilities of python and large set of libraries helped design a complex framework with reliable simple architecture. The prime design principle followed was loose coupling and high cohesion since it allows modularity and helps for the management of a complex systems. This design helped insert the technologies that are used at the correct places. Proceeding in a top-down manner helped us clearly identify the components.

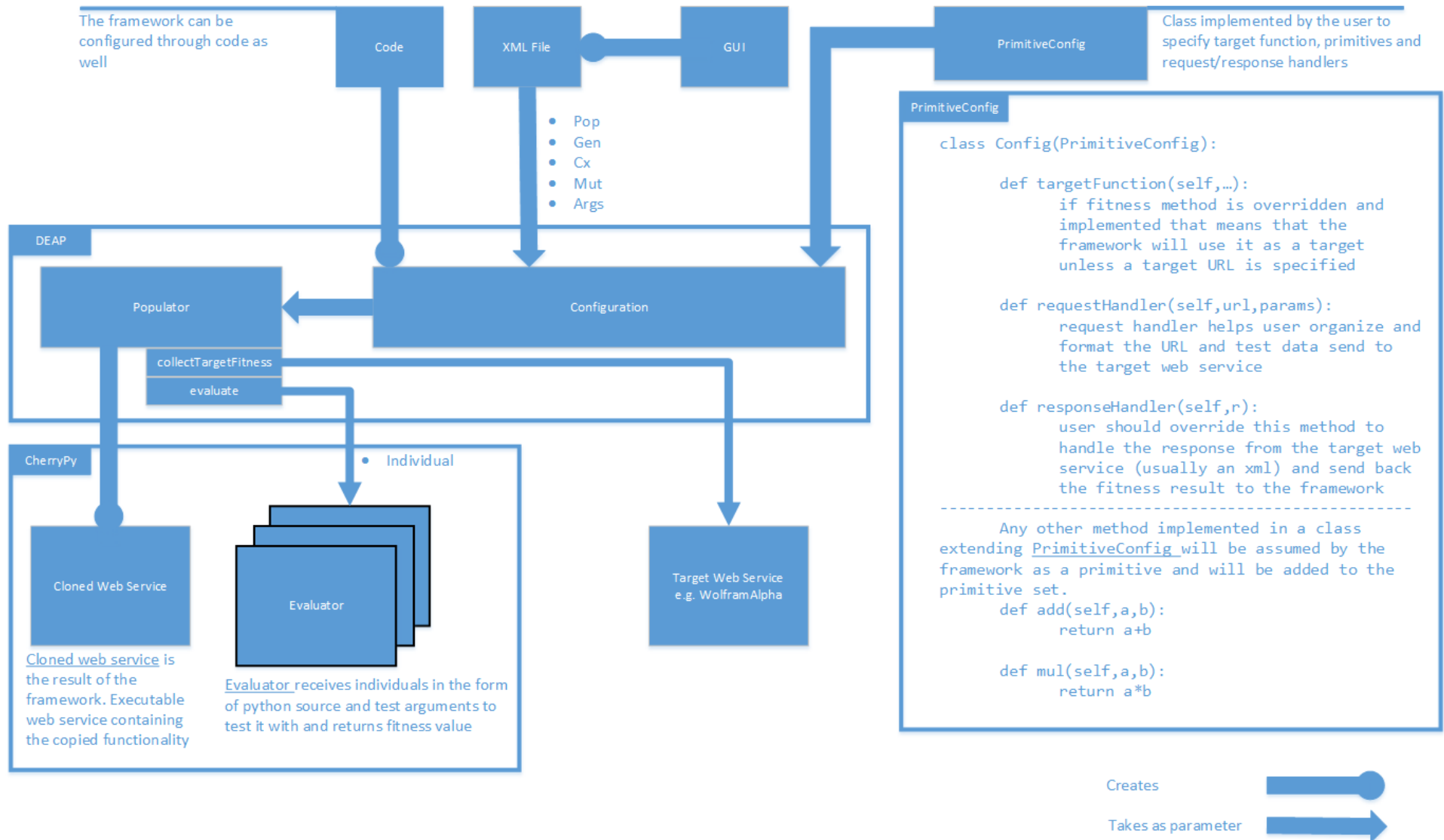


Figure 3.3: Lower level component diagram showing the functionality and usage of each class/component

3.2.2 Client

Before explaining how the client works, the DEAP framework needs to be explained. It is configured through a set of method calls to a *toolbox*; this *toolbox* is prior configured by a *primitive set* where all the primitives and terminals that the genetic program is going to use are defined. The toolbox give easy API for accessing and controlling the individuals, generations and population as well as methods that invoke mutation, crossover and evaluation over the population.

Knowing how the DEAP framework works, it was later obvious that the same parameters needed for DEAP's configuration were going to be needed for Darwin's as well. Through the *PrimitiveConfig* class shown in figure 3.3 it was possible to set all the primitive functions and the fitness function needed for the program. Later on the client decided that there should be a pre-set number of primitives and the user would choose through the GUI which to enable. However in the case of an advanced user if additional primitives need to be added the class can simply be extended and add them as normal methods. No other configurations regarding the primitives need to be done since Darwin is handling everything else.

In figure 3.3 you can see that for the configuration of the genetic programming algorithm other parameters are needed which are not specified in the *PrimitiveSet* class; this is done in the *Configuration* component. When setting up the framework they can be set manually or configured by an XML file that carries all the information needed. There are only two parameters that are mandatory and the framework won't work without them - the address of the evaluating web service and the target service that needs to be cloned. Any other parameter is automatically set up with a standard value for solving the symbolic regression problem. The class that configures the framework acts like a data model since all the parameters needed through any other step of the framework is instantiated there. This data model is later used by the *Populator* component.

The *Populator* class is basically a wrapper around the DEAP framework. The class creates the first generation by randomly generating individuals. Each individual is send over the network to the evaluating webservice. The evaluating webservice measures the fitness of an individual and returns it. After each individual's fitness value is updated it repeats the process until the number of generations are reached or an individual with 0.0 fitness is found. Besides executing the actual genetic program the (Populator) logs the traffic between the services and handles the output of the genetic program - either displaying the best individual or generating a web service working with the best individual.

3.2.3 Evaluator

The final big component is the *Evaluator* 3.3. It is a CherryPy service that can be executed on any machine running python and has only one method - evaluate. It takes an individual, test arguments in the form of a dictionary and fitness results evaluates the individual compares with the fitness function and return the result. Initially the individual was in a DEAP represented form however that meant that in order to evaluate it the same DEAP configuration needed to be set up on the evaluating side as on the client one. This was impractical, slower, it was introducing many dependencies and much harder to set up. The Evaluator needs to be easy to set up, easy to modify because it is expected to be ran on multiple machines including RaspberryPis. Because of that the individuals was later represented as source code which meant it was able to run on any machine that runs python. This way simplicity was achieved for this component by moving most of the overhead work to the client.

3.3 GUI Design and XML

A graphical user interface for generating the XML configuration file had to be designed for the third sprint. Ease of configuration had to be provided for configuring the multiple parameters the framework provides. Because it's a framework software project GUI is not as important as the more algorithmic parts of the project, but usability and simplicity are design aims of the project.

3.3.1 XML Configuration

The XML file that is parsed between the GUI and the framework needed to be readable and editable by users. That meant that the markup used had to follow standard design so it can be modified by users. After several iteration the XML file took the look of 3.1

Listing 3.1: XML file used by the framework for configuring the needed parameters

```
<config>
  <mut>0.2</mut>
  <arguments>
    <arg name="r" number="0">1</arg>
    <arg name="r" number="1">0</arg>
    <arg name="r" number="2">1.5</arg>
    <arg name="r" number="3">5</arg>
    <arg name="r" number="4">15</arg>
    <arg name="r" number="5">13.4</arg>
    <arg name="r" number="6">20.1</arg>
    <arg name="r" number="7">132.2</arg>
  </arguments>
  <pop>300</pop>
  <terminals>
    <terminal>3.14</terminal>
  </terminals>
  <cx>0.8</cx>
  <copyUrl>http://api.wolframalpha.com/v2/query</copyUrl>
  <basicPrimitives>
    <primitive>mul</primitive>
    <primitive>pow</primitive>
  </basicPrimitives>
  <evalUrl>http://localhost:8844</evalUrl>
  <gen>300</gen>
</config>
```

This designed provided a backup plan in case GUI implementation failed. However the GUI was created which meant a parser needed to be implemented which was done with the help of *lxml* library.

3.3.2 Purpose of the GUI and Iterations

While iterating through design decisions the idea was to make the GUI as common as possible. In many dialogs and simple graphical user interfaces a similar layout is encountered - menubar on top, left to right and top to bottom reading, bottom right buttons for completing the task see figure 3.4. Because of its purpose and the way

its used the GUI looks more as a dialogue window - simple window that helps generate a file/configuration and its used for a short amount of time.

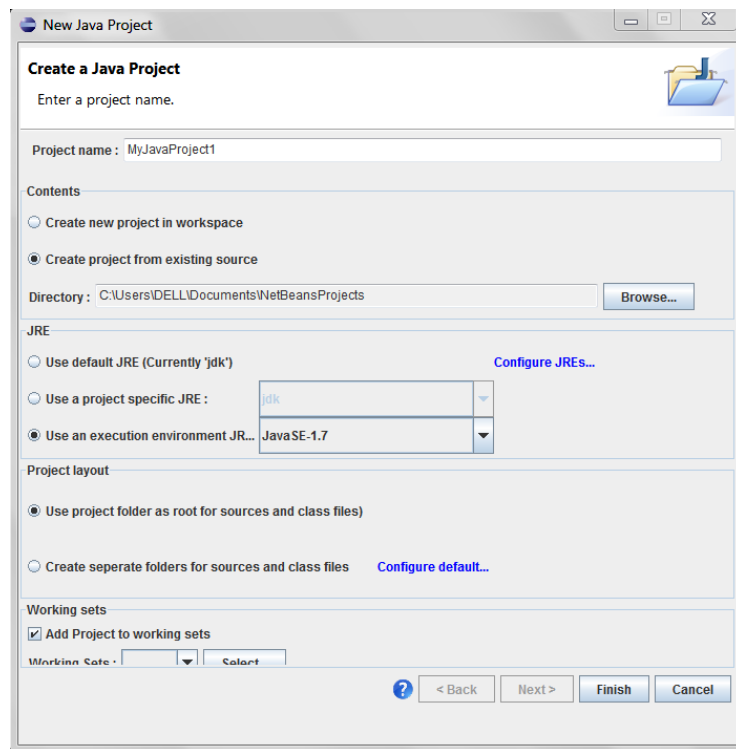


Figure 3.4: Example of a dialogue used for configuration of projects used in Eclipse editor

The most important parameters - target IP address and client IP address, are located at the top left corner of the window. Other parameters text fields are located from left to right top to bottom based on their relevance. That's why bottom left is located the button for generating the XML configuration file hence finishing the process.

In the first iteration of the design additional features like adding custom primitives and selecting imports for those custom primitives were added as you can see in figure 3.5. However after discussion with the client it was later decided that these features are part of the more advanced functionality of the framework. It was unnecessary to implement them in the GUI since they were available in the code where they would be normally reached and modified from a more experienced users that would use this functionality. The final iteration of the GUI design looked as displayed in figure ??

Darwin File Generator

Client XML Generator

Evaluator IP Address / Hostname

Target IP Address / Hostname

Basic Primitives

☒ Add

☒ Mul

☒ Div

☒ Sub

☒ Log

Population

▲

▼

Generations

▲

▼

Crossing Rate

▲

▼

Mutation

▲

▼

Arguments

1,2,0,3,2,0,1,4,0

Terminals

1,3,4,5

Imports

Operators,system

Mutation

▲

▼

Custom Primitives

Custom1(x,y)
Custom2(x,y,z)
Custom3(z)

Clear

Generate

Figure 3.5: One of the iterations of the GUI showing all the possible parameters

Chapter 4

Implementation

4.1 Technologies

4.1.1 DEAP

DEAP stands for distributed evolutionary algorithms in python. It is a framework that supports both genetic algorithms and genetic programming. It seeks to make algorithms explicit and data structures transparent. Hence making genetic programming easy by providing easy way for primitives to be created and access to pre set crossover and mutation algorithms. Each of its features can be fine tuned through multiple parameters so it's not creating heavy constraints.

As similar to Darwin, DEAP is emphasizing on loose coupling in its design, it uses multiple components that provide the necessary functionality for the framework. The *pset* is a class through which all the configuration related to the primitive set is done. Primitives are defined through mapping any defined function or method and declaring the amount of arguments they are taking. In addition the terminals are defined where there is a pre-set of constants that can be utilized. Arguments for the genetic program are also defined in through the *pset* hence separating it from other components so it can be extracted and manipulated individually.

Another core component of the DEAP framework is the *toolbox*. This is the component where the genetic programming is configured. Parameters like maximum depth of an individuals are specified, the algorithms for crossover and mutation. The framework gives the flexibility to define your own mutation and crossover algorithms as well as manipulate their rates and the size of the population and generation. Fitness function is specified in the toolbox and later maps to the toolbox. After all the configuration of the primitive set and the genetic programming parameters is done the population is generated. There is an automatic way of doing it through special DEAP functions however for Darwin a more complex approach was chosen which will be later explained. In the end there is the statistics module that takes the generations and creates set of statistics that can be displayed (e.g. mean of a generation, best and worst individual).

4.1.2 CherryPy

CherryPy is a minimalistic python object-oriented framework which focuses on flexibility and has a reliable HTTP/1.1-compliant, WSGI thread pooled webserver. It requires small amount of source code to run it, making

it very useful to take the purpose of an *evaluator*. The *evaluator* needs to be light in order to run either with multiple instances on a single server or run on a small devices like a RaspberryPi.

Listing 4.1: Example CherryPy webservice that has the method "hello"

```
class HelloWorld( BaseCherryPy ):

    def hello( self ):
        return "Hello_world!"
    hello.exposed = True
```

The way CherryPy works is it maps each method written in a class by adding it to the server URL name so it can be called. Each of those methods has to have an *exposed* flag set to true as shown in 4.1 so CherryPy can see it. This class is ran through the CherryPy which handles all the URL parsing. For example if we have a method as shown in 4.1 and the hostname is *http://localhost* then when accessing the server with URL *http://localhost/hello* the URL and it's parameters if existing will be forwarded to the method *hello*. After that the hello method will return the contents and CherryPy will format the response in a correct HTTP format.

4.1.3 Inspect

Inspect is a python library for getting information from live objects (e.g. modules, classes, methods, functions, tracebacks, frame objects, code objects). This is the primary tool used for automating the Darwin framework and the main tool used in code generation as well. Inspect library provides multiple functions for gathering information about live objects. In the case of Darwin functions, methods and arguments were the inspected live objects from which the source code, the signature or variable information was extracted.

4.1.4 Requests

Requests is a simple python library that provides easy API for sending HTTP requests and receive responses. It is an abstraction over the standard python library *urllib2*. Requests library works by creating a requests object which is filled in with the data it needs - URL as string, parameters in the form of dictionary. This object has methods like *get* and *post* which send respectively GET or POST request to the target URL with the parameters. The *get* and *post* method return a response object that contains the result in text format which in the case of web services is normally XML.

4.1.5 wxPython

WxPython follows standard graphical library design pattern -MVC (Model View Controller) . This design is focused on separation of concerns making it perfect for building a GUI for creation of XML files. The model or the data is the configuration parameters used, the view is the wxPython's frame and interface and the controller is the XML parser making use of data.

The library provides easy to understand and use API. The main component is a frame, each secondary component like a pane, button or a menubar is added to it. The framework uses different types of sizers to organize the components in the frame. To create more complex layouts sizers can be stacked. WxPython has large amount of pre-set and easy to configure dialogue boxes for almost any case that might be needed. More complex components were later created because of framework's versatility and modularity.

The main challenges in creating the GUI wasn't creating the functionality and linking it to the graphical interface but making the interface user friendly

4.2 Implementation

4.2.1 DEAP Set Up

The way DEAP is used in Darwin framework by utilizing the loose coupling used in it's design. Each of the DEAP components is configured separately. This way the processes of Darwin framework that are dependant on DEAP can be split into multiple functions so loose coupling can be achieved.

The *Configuration* class as seen in figure 4.2 consists of the *pset* and *toolbox* components from the DEAP framework. In order to set the needed information for the genetic programming an abstraction layers is added to the DEAP framework. Because the purpose of Darwin is known - solve symbolic regression problems, there are many pre-set parameters used in the configuration, hence there are fewer parameters to be specified for Darwin. This level of abstraction both simplifies the use of the framework by lowering the amount of parameters needed and creates a link between Darwin and DEAP. *Configuration* provides a *configure()* method that organizes the Darwin variables and sets up the DEAP framework underneath. This *configure()* method does four things as seen in [ref fig configure method]:

1. *setPrimitiveFunctions()* extracts the primitive functions from the user defined class that extends *Primitive-Config* . It takes an instance of each of the methods and maps them to the DEAP framework.
2. *setTerminals()* gathers a list of all defined terminals and instantiates them in the DEAP framework.
3. *configureToolbox()* is a more complex method where DEAP's *toolbox* is configured.
 - (a) Creates an individual by specifying the type of structure it is going to use (PrimitiveTree in the case).
 - (b) Checks if the individual is aiming for minimum or maximum fitness function; this sets the individual to either look for lowest or 0 fitness value or the maximum among individuals.
 - (c) Defines the initial size of the individuals, the size of the population and generation based on the user parameters.
 - (d) Maps all the mutation and crossover methods to pre-defined DEAP algorithms which are good for the purpose of the genetic program
 - (e) Since DEAP uses decorators to modify the generated individuals, decorators need to be defined to limit the size of an individual.
4. *configureArgumetns()* takes the test arguments given by the user and based on their names maps them to DEAP so the individuals can take arguments.

Listing 4.2: The core method in Configuration class that configures the DEAP framework

```
def configureToolbox( self ):
    if not self.isMax      : creator.create("FitnessMin", base.Fitness , weights=(-1.0,))
    else                  : creator.create("FitnessMax", base.Fitness , weights=(-1.0,))

    creator.create("Individual", gp.PrimitiveTree , fitness=creator.FitnessMin , pset=self.pset)
    self.toolbox.register("expr", gp.genRamped, pset= self.pset , min_=self.depthInitialMin , max_=self.depthInitialMax)
    self.toolbox.register("individual", tools.initIterate , creator.Individual , self.toolbox.expr)
    self.toolbox.register("population", tools.initRepeat , list , self.toolbox.individual)
    self.toolbox.register("select", tools.selTournament, tournsize=3)
    self.toolbox.register("mate", gp.cxOnePoint)
    self.toolbox.register("expr_mut", gp.genFull , min_=0, max_=2)
    self.toolbox.register('mutate', gp.mutUniform, expr= self.toolbox.expr_mut)

    self.toolbox.decorate('mutate',gp.staticDepthLimit(self.maxDepthLimit))
    self.toolbox.decorate('mate',gp.staticDepthLimit(self.maxDepthLimit))
```

4.2.2 Generation of Individuals

Individuals are created and generated in the *Populator* class. It is a component of the framework where the genetic programming is done and handles output. It also contains functionality for communicating with other web services, either an evaluator or a target web service for cloning.

Unlike the *Configuration* class the *Populator* doesn't work with the *pset* and *toolbox* DEAP components directly. It uses them to generate individuals and apply mutation or crossover algorithm to it. Instead of using the standard genetic programming algorithm that DEAP provides by calling a single function, the algorithm had to be implemented manually. As seen in 4.4 the offspring is initialized by copying it from the initial population. This first population is a generation of random individuals with depth between 1 and 3.

Listing 4.3: Populate function responsible for generating individuals

```
def populate(self):
    for gen in range(self.configuration.gen):
        self.offspring = self.toolbox.select(self.population, len(self.population))
        self.offspring = map(self.toolbox.clone, self.offspring)
        self.crossover()
        self.mutation()
        invalid_ind = self.nonEvaluated(self.offspring)
        fitnesses = self.toolbox.map(self.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
        self.population = self.offspring
        self.hof.update(self.population)

    self.outputIndividuals()
```

To each generation crossover and mutation functions are applied. However both of the functions are modified so each individual is evaluated separately. The whole avoiding of DEAP standard genetic programming function is because evaluation is not done in the *Populator* but by the *Evaluator*. In order to make use of the decorators the crossover and mutation function a manual substitution of the generation is done as seen in ??.

Listing 4.4: Modified mutation method to substitute each parent with its children

```
def mutation(self):
    for i in range(len(self.offspring)):
        if random.random() < self.configuration.mut:
            self.offspring[i], = self.toolbox.mutate(self.offspring[i])
        del self.offspring[i].fitness.values
```

4.2.3 Evaluation Over a Network

Evaluation over the network is done using the requests python library creating HTTP requests to the *evaluator* webservice that is created using the CherryPy Framework.

If no target function is specified and a target webservice is, then requests library is used to send the test arguments to the webservice so it can get the correct result. This result is put with every set of test arguments in a dictionary and send to the evaluator together with an individual. Initially the idea was to have multiple evaluators processing the individuals which meant making asynchronous requests to multiple webservices. However the complexity of the task was too high and the development time for such a component would of been too long, so there is only a single evaluator used.

The easy parsing of parameters in CherryPy made it comfortable to send individuals as parameters to the web service and their test arguments so they can be evaluated. Initially the web service had to set up additional DEAP configuration in order to evaluate an individual and return the result. However there was a lot of complexity and dependencies were too many and so it was later re-factored to parse the source code of an individual to the *evaluator*. This resulted into simplification of the evaluator to a single simple method.

Listing 4.5: The source of the evaluator

```
class BaseCherryPy:

    def evaluate(self, individual, arguments):
        arguments = eval(arguments)
        diff=0
        for i in arguments:
            sys.argv = i[0]
            with stdoutIO() as s:
                exec(individual) in {}
            try:
                res = float(s.getvalue())
                diff += (res - i[1])**2
            except:
                diff = sys.float_info.max
        return str(diff)

    evaluate.exposed=True
```

In order to run the evaluator the class that is ran by CherryPy needs to be either *BaseCherryPy* or a class extending it. What *BaseCherryPy* contains is a method that has the exposed flag set to true that handles the sent individual. The client is always calling the method called *evaluator* see ?? with two parameters. One is an individual in python source code and the second is a dictionary with test arguments and fitness values from target. What the method does is it sets the system arguments of the python interpreter to the test arguments and after that executes the source code into a different context. Then by overwriting the *contextmanager*¹ the python standard output is parsed back to the original context (the one of the evaluator).

4.2.4 Code Generation

Code generation is the main technique for automation used in Darwin framework. It is using the inspect library to gather the data and code needed for code generation. The main methods for extraction of code are located in *PrimitiveConfig*. The methods that use the inspect library are extracting the code of the functions chosen as primitives as seen in fig 4.6. The method *functionArgs* extracts the arguments hence the signature from target primitive function given it's name.

¹Note that the overwriting of contextmanager wasn't done by me but the stackoverflow user isedev in the thread <http://stackoverflow.com/questions/14708216/showing-current-output-from-exec-in-python> handling this issue

Listing 4.6: Function extracting the arguments of a primitive based on it's name

```
def functionArgs( self , funcName ):
    func = getattr( self , funcName )
    args = inspect . getargspec ( func ) [ 0 ]
    del args [ 0 ] # removes ' self '
    return args
```

The method for generating DEAP individual into source is located in PrimitiveConfig as well. In the *helper* module there are multiple functions constructing different parts of the individual source as seen in figure 4.7.

Listing 4.7: Function in PrimitiveConfig class that gathers source code parts of the individual

```
def getSource( self , imports , individual , arguments ):
    source = generateImports ( imports )
    source += generateFunctions ( self , False )
    source += generateMain ( arguments , individual , False )
    source += "print _main(**sys . argv)\n"
    return source
```

The *getSource* method collects source from three main methods located in the *helper* module.

1. *generateImports* based on a list of module names is generating the import statements.
2. *generateFunctions* extracts the source and the signature of a primitive by removing the self argument in the method.
3. *generateMain* creates a main function that is taking as parameters the arguments of an individual. Later the DEAP individual that is represented as a stack of functions is ran through a lambda expression that yields and returns the result
4. the main method is called using system arguments as a dictionary. The result is printed to the standard output so it can be forwarded to the original context when executed

The same methods are used when generating a webservice as seen in figure 4.8. The difference is the code is wrapped into a class that is ran by CherryPy meaning the primitive signature and the lambda expression need to include the *self* notation. The webservice has an index method that is called only by accessing the URL that returns the result of the individual.

Listing 4.8: A generated web service that is the result of the Darwin framework

```
class ClonedWebService( BaseCherryPy ):

    def mul( self , a , b ):
        return operator . mul ( a , b )

    def main( self , r ):
        r = float ( r )
        ind = lambda r : self . mul ( self . mul ( 3.14 , r ) , r )
        return ind ( r )

    def index( self , r ):
```

```
        return self.main(r)
index.exposed = True

cherry.py.quickstart(ClonedWebService())
```

4.2.5 GUI

Chapter 5

Evaluation

Evaluation is an important part of system development. *Darwin* being a complex framework extensive tests needed to be ran over each of it's component. Unit tests were used to verify the functionality of the core components, system testing was used to evaluate the overall functionality and stability of the system. User evaluation was used to test the usability of the framework and the graphical user interface.

5.1 Unit Testing

By separating the framework into several base components it was easier to split each of the components into it's core functions and apply unit tests to verify their correctness. However due to the complexity and dependencies in the framework test writing was complicated. Many of the functions depended either on the DEAP toolbox or on the primitive set which was responsible for generating DEAP individuals. They couldn't be reproduced manually for the tests, so multiple components had to be ran first in order to verify the unit tests. Since DEAP was an external framework with a stable release it was assumed that it is functioning correctly so no unit tests were ran on it specifically however components using it were extensively tested. Each of the components had testing aims however not all functions were covered since their complexity or dependency was too high, thats why they were left for system testing later on. The strategy for unit testing looked like this:

1. Unit tests on each of PrimitiveConfig's methods
 - (a) Test pre-set primitives
 - (b) Test code generation
 - (c) Test utility functions
2. Test all of Populator's methods without the configuration methods or the more complicated ones which would be part of the system testing.
 - (a) Test methods related to individual manipulation
 - (b) Test code generation
 - (c) Test fitness gathering functions
3. Handle all the functions in the helper module excluding Proxy class that is going to be aprt of the system testing.
 - (a) Test utility functions

(b) Test code generation

Through the unit tests many errors in the code generation functions were found so they had to be re factored. The imports used for the custom primitive set weren't working properly so they had to be fixed. After the unit tests were created initially they were used for verification of code correctness after a change in the code. If the unit test didn't work that meant there was an error in the code.

5.2 System Testing

There were many dependencies in the framework, which made testing a very difficult task. So instead of spending a lot of time trying to write complicated unit tests for each function/method in the code it was decided it's easier to do system testing. In the case of methods like *evaluate* where individuals were send to the evaluating web service unit tests weren't an option because in order to test them individuals had to be generated manually. Generating individuals was easy but generating the same ones every time so the test data can be robust was a problem. That's why after other framework functionalities were verified system testing was done to prove the correctness of the more complex components.

For system testing the goal was to use a generated web service for cloning and a real world one. The first step of the system testing was easy and simple enough. Testing proved successful and useful results were pulled out of it. For example a bug was detected in which DEAP individuals were generated with depth above 90. This meant that the python stack was breaking and couldn't evaluate the individual properly. The bug was fixed by modifying the way DEAP's decorator methods were used in the individual generation. This helped clear a major issue and second stage of system testing could being. In the second stage Wolfram Alpha[ref] was used as an external web service targeted for cloning. The first issues was the way Wolfram Alpha was receiving parameters and the responses in XML format. Since the framework didn't have functionality to handle the issue system testing was stopped and development over request and response handlers was started. After completion of the tasks system testing was initiated again. Framework passed through first and second stage of system testing. Second stage was tested with Wolfram Alpha. The framework succeeded to generate a web service that had the same functionality as the expression used as input for Wolfram Alpha. The mathematical formula used was measuring the face of a circle. It was a formula that was simple enough so cloning time can be short but complex enough to show the frameworks functionality.

5.3 UI and Usability Testing

Important feature for a framework is usability. Participants were used to evaluate the quality of the framework and check if the goals to create an easy to use and simple framework were achieved. The participants were presented with tutorial for the framework and it's documentation each of them had to have basic knowledge in programming. The choice of first to fourth year students presented a good study group since it presented people from different programming experience. The participants had to asses the quality of the tutorial and documentation and answer the following questions:

1. How helpful was it to understand the basics of the framework?
2. How complex it was to create the first examples?
3. How confident are you understand the framework basics?

4. Were you given enough information to continue working with the framework on your own?

In another part of the evaluation they had to assess the graphical user interface. They had to answer the following questions:

1. How intuitive is it?
2. How simple is it?
3. Were there enough options presented?
4. Would you think tips would be helpful above each field?
5. Can you give additional feedback that can help improve the GUI

In addition through the evaluation the participants were observed and notes on their actions were taken to compare with their feedback. This way more accurate conclusions could be drawn so the correct features could be improved and re-worked.

The studies helped spot problems in...[wait for results]

Chapter 6

Problems Encountered and Future Development

It has been a great journey developing this project and one that will hopefully continue further. Since the beginning there have been many difficulties with the project mostly related with the meta-programming field it's focused on. Finding sources and doing research has been problematic because development in the field of genetic programming isn't on the same scale as other more popular fields in programming (e.g Machine Learning, Artificial Intelligence). However the result from the project has motivated me to continue explore genetic programming even more and expand the framework.

6.1 Problems

Multiple Technologies - among the main issues with the project is the amount of technologies used. This meant each of them had to be understood so quality development in the area can be done. The framework was related to web services, genetic programming, code generation, GUI which meant extensive research in each area so the correct libraries or frameworks can be chosen. The learning curve was steep for the project however each of the fields became easier to work and understand after the basics were clear.

Proof of Concept - before the project reached any results it had a *proof of concept* label. Exploring new fields and expanding your knowledge in specific fields can be fun but it's hard when not a lot of information is presented. Three times during the project I had to reach out to *stackoverflow* for help related to issues I couldn't find anywhere on the web resolved. The first time it was related to the way CherryPy processed its requests and responses. The second one was focusing on the problem of converting DEAP individuals into executable code or AST. Later on I had to tackle the problem on my own since such a functionality wasn't introduced to DEAP. However the developers from DEAP were kind enough to provide support for further questions and the code generation has spawned the possibility for an open source contribution to the DEAP framework with an extension that converts DEAP individuals to AST's. The third time was related to code execution in python and the way arguments were passed to a sub process.

6.2 Future Development

Since the beginning of the project the development has been very ambitious and all the goals that were set were accomplished. Which gave the opportunity to think a lot about improvements for the framework - anything from re-factoring, improving features to adding additional functionality.

6.2.1 Improvements

Expand symbolic regression - throughout the project symbolic regression was used as a proof of concept so the framework functionality can be showed. However non mathematical web services are harder to implement using the framework and this can be improved so cloning can be more automated. This can be done by adding a better method for overriding at the evaluating web service and tweaking the basic primitives sets by adding primitives for control flows.

AST for code generation - currently all the code that is generated is in string form and through unit tests it was proved that it's working well however using string manipulation rather than manipulating ASTs is not as robust. A good re-factoring plan is to do all the code generation with ASTs and keep generation from strings as little as possible. This would also help for creating the base for an open source contribution to the DEAP framework that adds the functionality for individuals to be converted to ASTs and/or executable python code.

Asynchronous requests - among the initial requirements of the project was using a cloud of evaluating web services to evaluate individuals over the web hence improving the speed especially for more complex individuals. However that idea is a small project on it's own and it was going to bring a lot of complexity to the project. Since the core of *Darwin* is completed it's a solid place to start from however for dealing with threads and asynchronous requests python is not the best language to handle the issues, implementing it in C would be much better

6.2.2 New Functionality

Improvement of web services - improving web service performance using the framework was an initial requirement in the project but due to complexity and time issues it was opt-out from the project. The concept can still be implemented and used for improving current web services that the developer has created so improved web service can be compared by performance speed, memory usage and stability to the target one. After certain threshold of improvement is reached there can be automatic deployment of the generated improved web service.

Using unit tests as fitness - a way of measuring fitness can be a set of unit tests the individual needs to pass. This way individual can be tested if certain functionality exists and hence force the genetic program to evolve towards an individual with the required functionality. From that point a dynamic addition of unit tests can be included so functionality can be introduced through the evolution of a program. Comparing it to biological evolution its like adding a new threat for an organism in its environment hence the organism needs to adapt.

Chapter 7

Conclusion

7.1 Contribution

Bibliography

- [1] Bottle. Bottle. <http://bottlepy.org/docs/dev/index.html>.
- [2] CherryPy. Cherrypy. <http://www.cherrypy.org/>.
- [3] deap. deap. <http://deap.gel.ulaval.ca/doc/default/index.html>.
- [4] Django. Django. <https://www.djangoproject.com/>.
- [5] Stephanie Famuyide. Moscow: Requirements prioritization technique.
- [6] Flask. Flask. <http://flask.pocoo.org/>.
- [7] GitHub. Github. <https://github.com/>.
- [8] Michael G. Epitropakis Jerry Swan and John R. Woodward. *Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming*. PhD thesis, Department of Computing Science and Mathematics; University of Stirling, 2014.
- [9] lxml. lxml. <http://lxml.de/>.
- [10] The University of Glasgow's Raspberry Pi Project. The university of glasgow's raspberry pi project. <http://raspberrypicloud.wordpress.com/>.
- [11] Raspberry Pi. Raspberry pi. <http://www.raspberrypi.org/>.
- [12] PivotalTracker. Pivotaltracker. <http://www.pivotaltracker.com/>.
- [13] PYevolve. Pyevolve. <http://pyevolve.sourceforge.net/>.
- [14] PyGene. Pygene. <http://freenet.mcnabhosting.com/python/pygene/>.
- [15] pySTEP. pystep. <http://pystep.sourceforge.net/>.
- [16] Westley Weimer Claire Le Goues Stephanie Forrest, ThanhVu Nguyen. *A Genetic Programming Approach to Automated Software Repair*. PhD thesis, Dept. of Computer Science of University of New Mexico and University of Virginia, 2009.
- [17] W3. W3. <http://www.w3.org/TR/ws-gloss/>.
- [18] WebPy. Webpy. <http://webpy.org/>.