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

Abstract

Web services have become a common way of communicating between devices. With the growth of the world wide web there are thousands software projects using webservices to exchange information.

Genetic programming is a branch of artificial intelligence inspired by biological evolution. Its creating a program that modifies programs. It uses an evolutionary algorithm based methodology to iterate through individuals(programs) and captures the best performing ones. The result of a genetic program is a heuristic, a stochastic method - no guarantee of success.

The project aims to prove that genetic programming can be used for cloning web services. It focuses on creating a user centric framework. Cloning is done through reverse engineering the functionality using the primitive set of the genetic program. One of many applications of cloned web services is software redundancy also known as N-version programming, that can increase the reliability of a system. The Darwin framework proves that such use of genetic programming is possible.

This is an innovative project, there are no prior attempts to use genetic programming to clone web services. Hopefully it would create a base for future development in the area.

Acknowledgements

I would like to thank David White for supervising the project and helping with building the requirements and successfully introducing me to the field of genetic programming. Many thanks to the DEAP developers Flix-Antoine Fortin, Marc-Andr Gardner and Franois-Michel De Rainville who helped me with understanding and modifying the DEAP framework.

Contents

List of Figures

List of Tables

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 a web service 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 main goal of this project is to produce a tool for cloning 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 a symbolic regression problem was used. To solve a symbolic regression problem the genetic program needs to generate an individual that yields the same result as a mathematical expression specified in the fitness function. But instead of defining the expression used in the fitness function, a real webservice is used hence generating a program that has the same functionality as the webservice. Proving this meant that theoretically, by increasing the complexity of our function set we can achieve cloning of an even more complex web service.

1.1 What is 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 pre-defined set of **primitives** and **terminals**. The primitives represent the functions from which the individual is constructed and the terminals represent the bottom nodes of the tree, they can be both arguments and constants. A tree representation of 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.

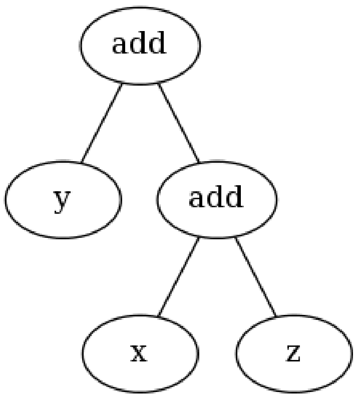


Figure 1.1: An example of a genetic programming individual that represents x+y+z

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

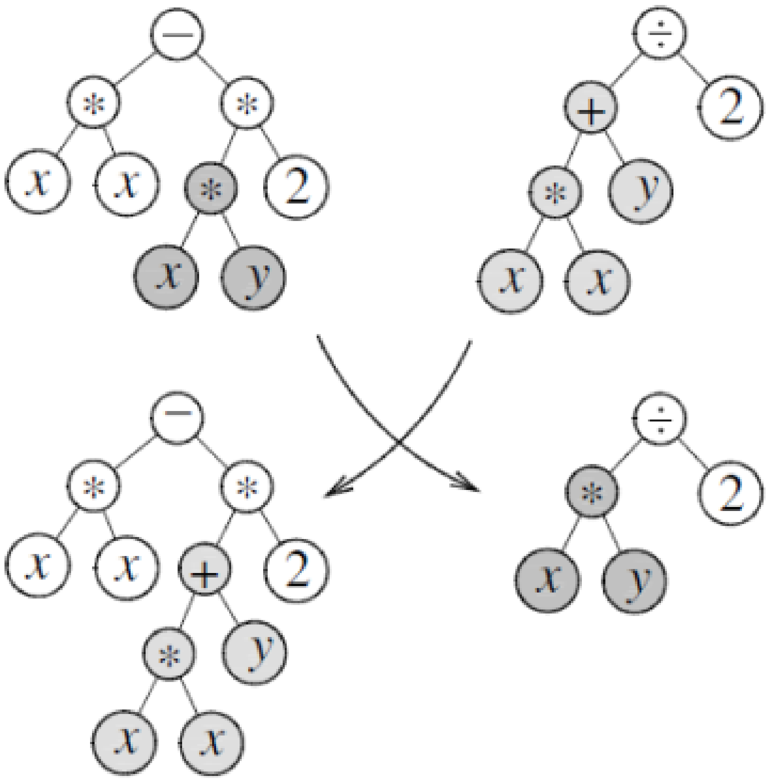


Figure 1.2: An example of a crossover between two individuals and their children as result

One of the problems with genetic programming is searching for the correct parameters to solve a specific

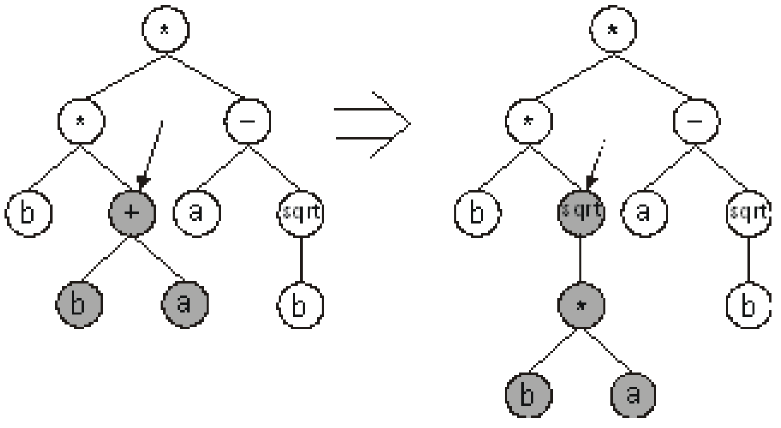


Figure 1.3: An example of a mutation applied to an individual

problem. For each problem a specified number of populations, generations needs to be provided e.g the percentage of the population to which crossover and mutation are going to be applied. In more advanced cases the types of mutation and crossover algorithms 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 wisely chosen as well. If too many primitives are chosen the generations might not be enough to find a solution to the problem because the search boundaries were expanded. 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. In this project this technique is uesed as a proof of concept for the capabilities of the framework.

1.2 What are 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. [1]

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. Instead they 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 their 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.

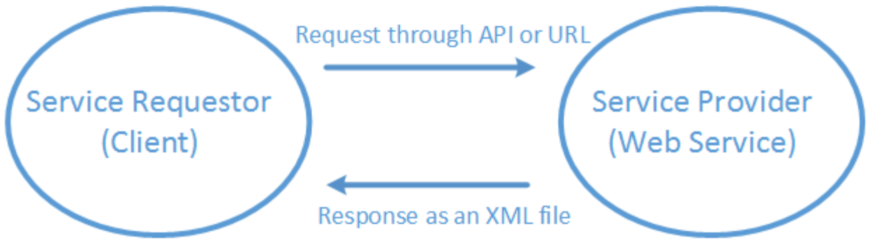

Service Requestor (Client)
Request through API or URL
Response as an XML file
Service Provider (Web Service)

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 application 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 Project's Aim

The purpose of the project is to develop a framework that can reverse engineer 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, however the third step was changed to creating a graphical user interface(GUI). 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 the target expression in the function. 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* [2]. 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* [3].

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); its 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.

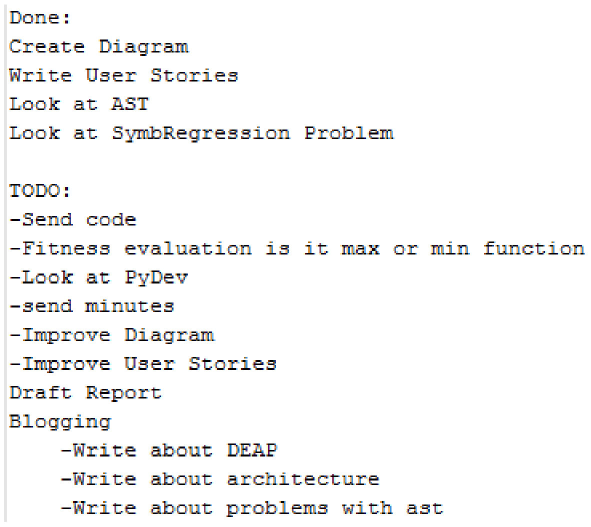
Both of the frameworks focus on post-development processes (e.g. bug fixing, code improvement, code maintenance). These ideas are different from *Darwin*'s one of cloning web services however the concept for creating and modifying existing code through genetic programming is observed in all three of 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.

2.2.1 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 meetings 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 2.1: 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 2.1

2.2.2 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 [4] was used for version control used and PivotalTracker [5] was the agile tool of choice.

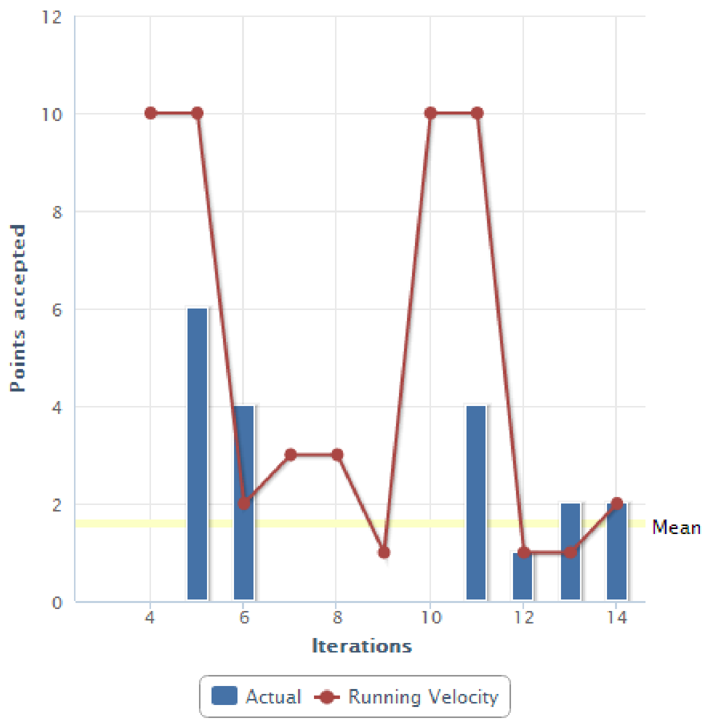


Figure 2.2: Velocity graph showing the sprints and the amount of work based on tickets completed in different iterations during first and second sprint

PivotalTracker is a web based tool where all the development process was stored, it was a 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 2.2. This way the client can monitor how the actual development was going. Pivotal tracker was used to monitor the progress of the project as seen in figure 2.3.

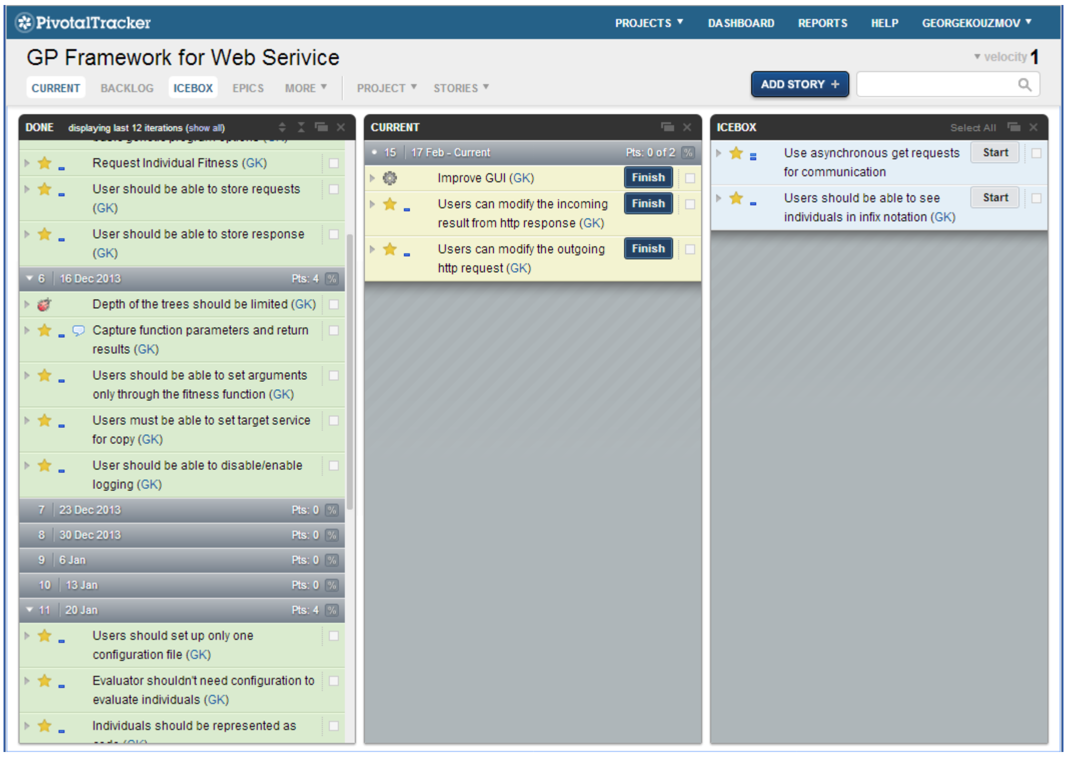
GitHub is the version control tool used for the project. As a public code repository it gave the opportunity to present the code easily to the client and others as well. Frequent commits were done during the sprints. This way when the code broke during a sprint it could be reversed back to a working version. The repository is easily accessible from everywhere making it useful for development from multiple machines.

2.3 Requirements Gathering

After creating project management environment and clarifying genetic programming concepts it was possible to specify the requirements. There were pre-set by the client, however with time they iterated and new ones were introduced. Requirements were key to establishing the same goal both for me and the client.

2.3.1 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 [6] method was used to reach



GP Framework for Web Serivice

DONE displaying last 12 iterations (show all)

Request Individual Fitness (GK)
User should be able to store requests (GK)
User should be able to store response (GK)

6 | 16 Dec 2013 Pts: 4

Depth of the trees should be limited (GK)
Capture function parameters and return results (GK)
Users should be able to set arguments only through the fitness function (GK)
Users must be able to set target service for copy (GK)
User should be able to disable/enable logging (GK)

7 | 23 Dec 2013 Pts: 0
8 | 30 Dec 2013 Pts: 0
9 | 6 Jan Pts: 0
10 | 13 Jan Pts: 0
11 | 20 Jan Pts: 4

Users should set up only one configuration file (GK)
Evaluator shouldn't need configuration to evaluate individuals (GK)
Individuals should be represented as

CURRENT

15 | 17 Feb - Current Pts: 0 of 2

Improve GUI (GK)
Users can modify the incoming result from http response (GK)
Users can modify the outgoing http request (GK)

ICEBOX

Use asynchronous get requests for communication
Users should be able to see individuals in infix notation (GK)

Figure 2.3: PivotalTracker and the way it is used by the project

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

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.

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)

2.3.2 User Stories and Sprints

As mentioned before, 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.

Chapter 3

Design

When designing the framework the key principle was to provide a simple way to control the main 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 Choice of 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 there were multiple options available.

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, because it was expected to run multiple instances on a machine or a single instance on a smaller device. A potential deployment platform was the Glasgow Raspberry Pi Cloud [7,8]. WebPy [9], CherryPy [10], Flask [11], Bottle [12] and Django [13] 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 or Bottle 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.

Table 3.1: Comparison between the considered webservices

Web service	Quality Documentation	Lightweight	Correct POST/GET Handling
Flask	✓	✓	x
Bottle	x	✓	x
Django	✓	x	✓
WebPy	x	✓	x
CherryPy	✓	✓	✓

CherryPy is a minimalistic python object-oriented framework which focuses on flexibility and has a reliable HTTP/1.1-complient, WSGI thread pooled web server. It requires a small amount of source code to run it, making it very useful to take the purpose of the project.

3.1.2 Genetic Programming Framework

Genetic programming frameworks didn't present as many choices as RESTful web services however there was the alternative of implementing one. Pyevolve [14], pystep [15], pygene [16] and DEAP [17] 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 live objects (e.g. modules, classes, methods, functions, tracebacks, frame objects, code objects). 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.

The main problem using DEAP was that it didn't support conversion between DEAP 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 an 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 strings. 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 ASTs for 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 [18] and connected with the graphical user interface. 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.

wxPython follows the 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 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 a large number of pre-set and easy to configure dialogue boxes for almost any case that might be needed. More complex components were created because of the framework's versatility and modularity.

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 see figure 3.1.

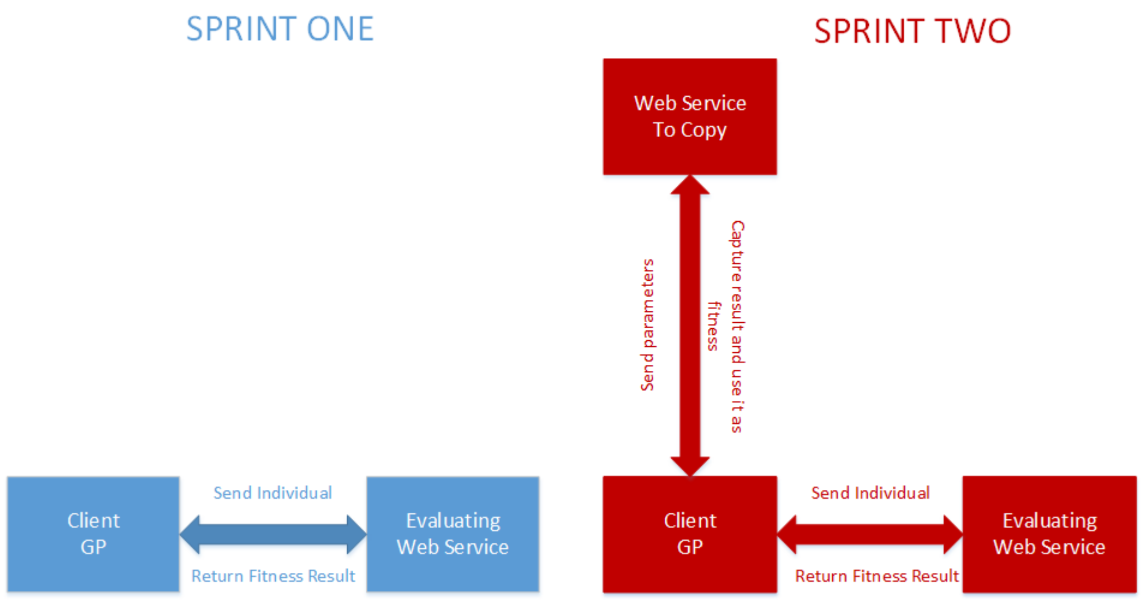


Figure 3.1: Top level view of sprint one and two. Sprint one is focused on evaluating an individual over the web and sprint two focuses on cloning a web services

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 GP individuals over a network. User stories helped identify how a user was going to approach the framework hence it was able to create components with known input and return parameters as seen in figure 3.2

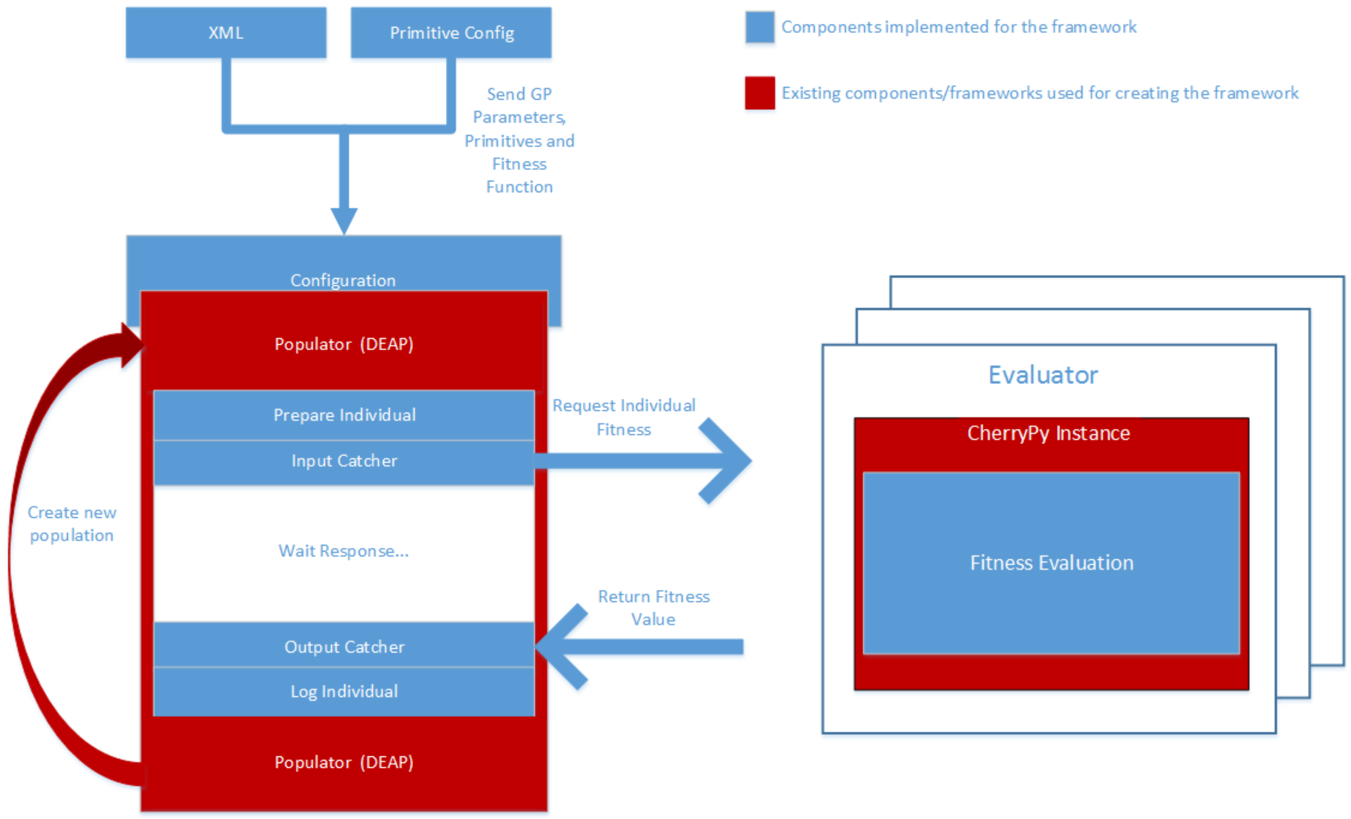


Figure 3.2: Lower level component diagram showing the functionality and usage of each class/component

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.

3.2.2 Client

Before showing how the client is designed, 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 gives an 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. So the idea was to create a *Configuration* class used as a data model for the framework. For easier configuration of the parameters an XML configuration file was introduced. It carried all parameters besides the primitives.

The *Configuration* component needed primitives to configure DEAP. To create modularity and automation a class was introduced where all primitives were captured. User would implement methods in the class that would be treated as primitives. To automate the configuration extraction of source had to be introduced to the class' functionality.

The *Populator* component would be a wrapper around the DEAP framework. The component would use it in a specific way to generate individuals and convert them to python code for easier transportation. Each individual is send over the network to the evaluating web service. The web service measures the fitness of the individual and returns it. After each individual's fitness value is updated it repeats the process until the number of generations are reached.

3.2.3 Evaluator

The final big component is the *Evaluator* as seen in 3.2. It is a CherryPy service that can be executed on any machine running python and has only one method - evaluate. It would receive an individual, compare it to results from the target return fitness as a result. Initial design involved using DEAP individuals. However to evaluate them the same DEAP configuration needed to be set up on the client and evaluating side. This was impractical,slower, it was introducing many dependencies. The Evaluator needs to be easy to set up, easy to modify because it is expected to be ran on multiple machines including Raspberry Pi. Because of that individuals were 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

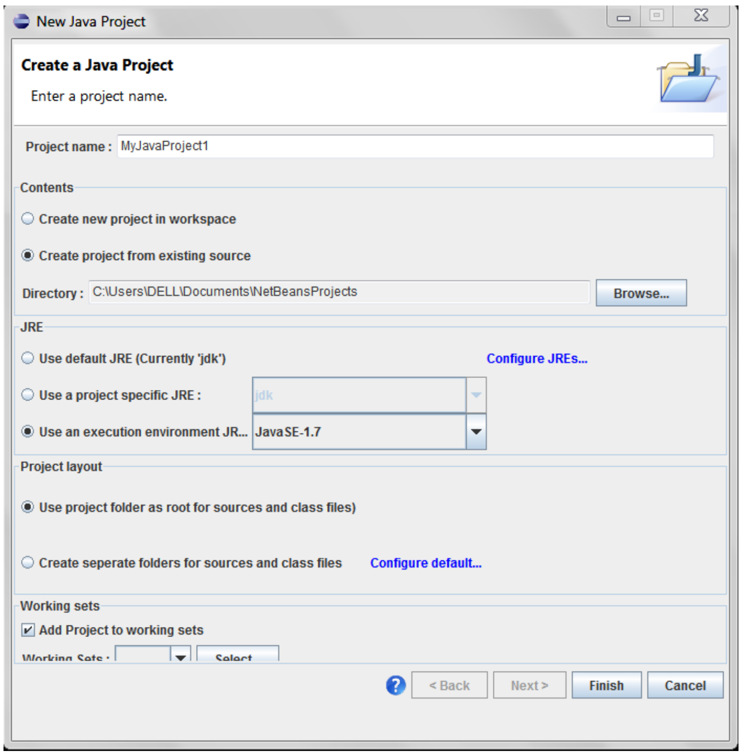
A graphical user interface for generating the XML configuration file had to be designed for the third sprint. Ease of configuration had to be provided for modifying the multiple parameters the framework provides . Because its a framework software project GUI is not as important as the more algorithmic 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. This design 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 familiar as possible. In many dialogues and popular 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 3.3. 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.


New Java Project

Create a Java Project

Enter a project name.

Project name : MyJavaProject1

Contents

Create new project in workspace

[✓ Create project from existing source]

Directory : C:\Users\DELL\Documents\NetBeansProjects

JRE

Use default JRE (Currently 'jdk')

Use a project specific JRE : jdk

[✓ Use an execution environment JR...] JavaSE-1.7

Project layout

[✓ Use project folder as root for sources and class files)]

Create seperate folders for sources and class files

Working sets

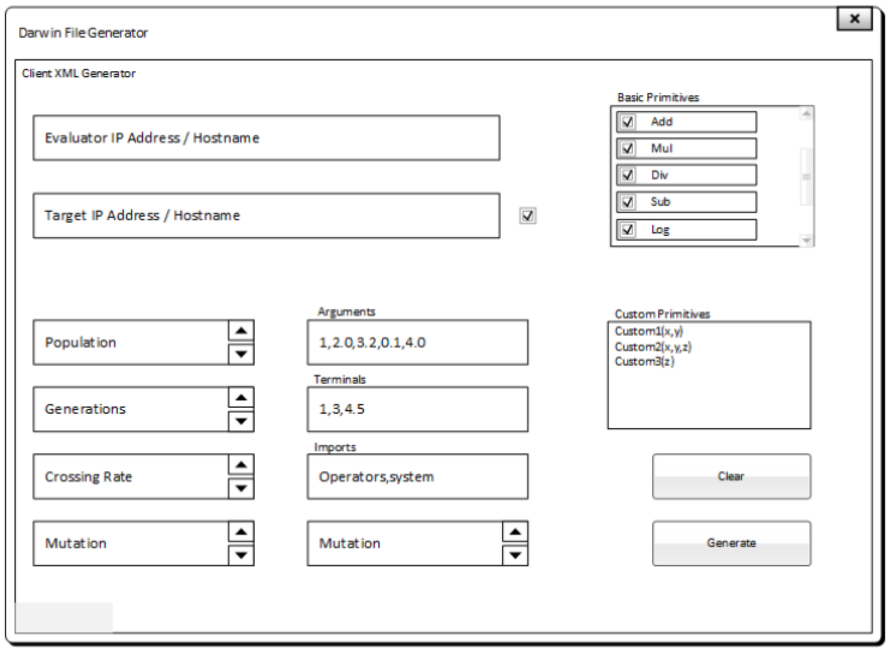
[✓ Add Project to working sets]

Working Sets :

Figure 3.3: 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 3.4. 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 3.4 it



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)

Figure 3.4: One of the iterations of the GUI showing all the possible parameters

Chapter 4

Implementation

During the first months of the project a lot of time was spend developing small components and testing the technologies. This created the base for productive sprints. There were difficulties in the implementation however the time spent testing out the technologies proved very useful in solving the problems quickly. The agile development proved very successful for implementing step by step the functionalities of the project. Following the architecture design strictly helped when modifications of components were done.

4.1 Use of Technologies

4.1.1 DEAP

DEAP (Distributed Evolutionary Algorithms in Python) 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 an easy way for primitives to be created and access to crossover and mutation algorithms. Each of its features can be fine tuned through multiple parameters.

Similar to Darwin, DEAP emphasizes 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 needed. Arguments for the genetic program are also defined 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 as well as 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 and mapped in the toolbox. After 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. The statistics module 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 Requests

Requests [19] 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. Requests library is used in the *Populator* component by giving it the functionality to access both the *Evaluator* and the target web service.

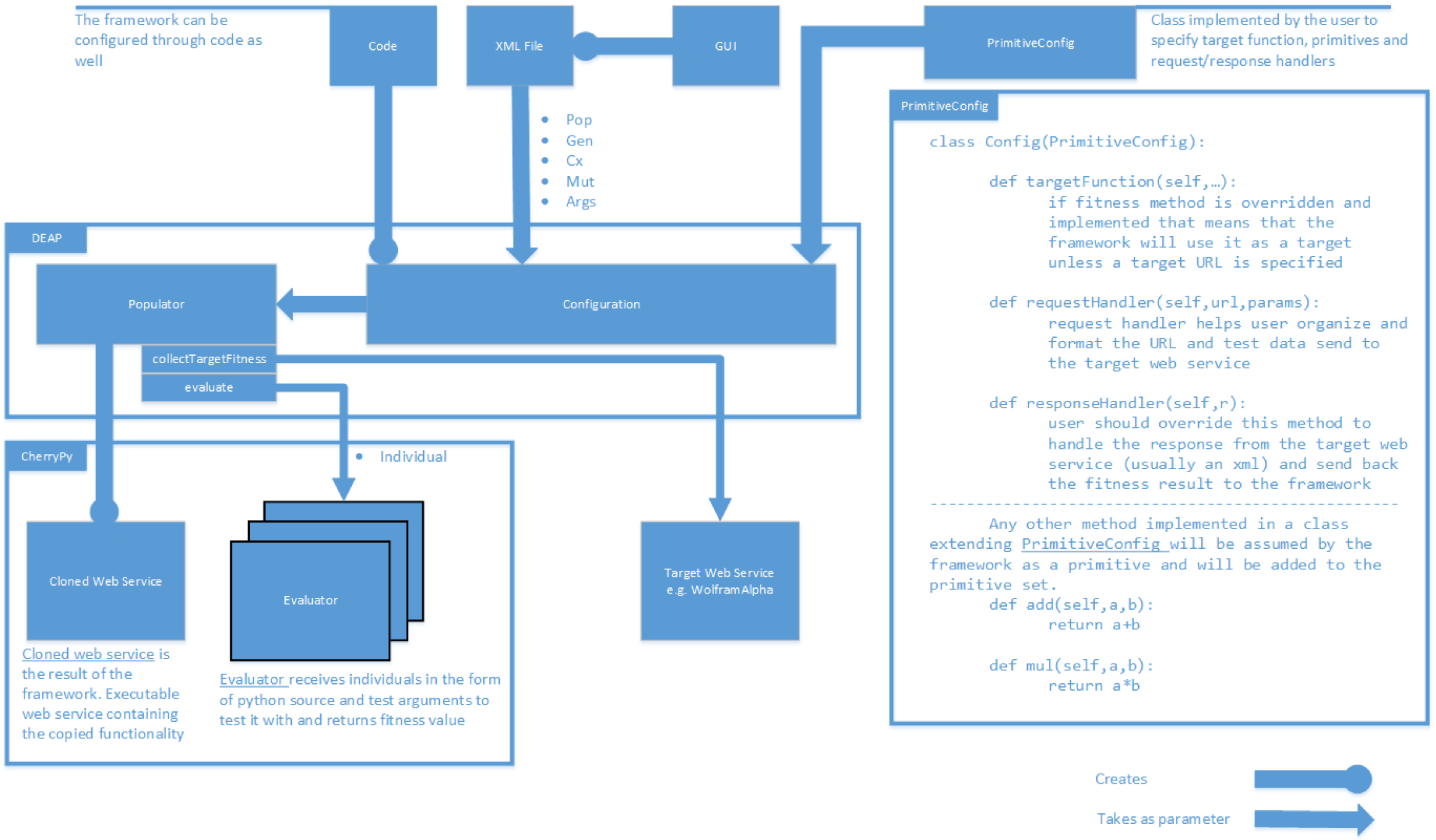
4.2 Core Components and Functionality

4.2.1 DEAP Set Up

DEAP is used in Darwin framework by utilizing its loose coupling 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 4.1 consists of the *pset* and *toolbox* components from the DEAP framework. In order to set the needed information for the genetic programming an abstraction layer 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 simplifies the use of the framework by lowering the amount of parameters needed. *Configuration* provides a *configure()* method that organizes the Darwin variables and sets up the DEAP framework underneath. This *configure()* method do four things as seen in 4.1:

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.


The framework can be configured through code as well
Code
XML File
GUI
PrimitiveConfig
Class implemented by the user to specify target function, primitives and request/response handlers
Pop
Gen
Cx
Mut
Args
DEAP
Populator
Configuration
collectTargetFitness
evaluate
CherryPy
Individual
Cloned Web Service
Evaluator
Target Web Service e.g. WolframAlpha
Cloned web service is the result of the framework. Executable web service containing the copied functionality
Evaluator receives individuals in the form of python source and test arguments to test it with and returns fitness value
PrimitiveConfig
Creates
Takes as parameter

```
class Config(PrimitiveConfig):

      def targetFunction(self,…):
            if fitness method is overridden and
            implemented that means that the
            framework will use it as a target
            unless a target URL is specified

      def requestHandler(self,url,params):
            request handler helps user organize and
            format the URL and test data send to
            the target web service

      def responseHandler(self,r):
            user should override this method to
            handle the response from the target web
            service (usually an xml) and send back
            the fitness result to the framework
---------------------------------------------------
      Any other method implemented in a class
extending PrimitiveConfig will be assumed by the
framework as a primitive and will be added to the
primitive set.
      def add(self,a,b):
            return a+b

      def mul(self,a,b):
            return a*b
```

Figure 4.1: Lower level component diagram showing the functionality and usage of each class/component (this is a copy for easier readability

Listing 4.1: 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 a 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 4.3 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 4.2: 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) #hall of fame storing the best
            individuals is updated

     self.outputIndividuals()
```

To each generation crossover and mutation functions are applied. However both of the functions are modified so each individual is evaluated separately. Avoiding of DEAP's standard genetic programming function is done because evaluation is moved to the *Evaluator* rather than the *Populator*. In order to make use of the decorators the crossover and mutation function, a manual substitution of the generation is done as seen in 4.3.

Listing 4.3: 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* web service that is created using the CherryPy framework.

If no target function is specified and a target web service is, then the requests library is used to send the test arguments to the web service so it can get the correct result. This result is put with every set of test arguments in a dictionary and sent 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 web services. However the complexity of the task was too high and the development time for such a component would of been too long, so only a single evaluator is 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 simplifies of the evaluator to a single simple method.

Listing 4.4: The soruce 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 used 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 4.4 with two parameters. One is an individual in Python source code and the second is a dictionary with test arguments and compare values from the 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 in a different context. Then by overwriting the *contextmanager*[1] the python standard output is redirected 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 uses 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 4.5. The method *functionArgs* extracts the arguments hence the signature from target primitive function given its name.

[1]Note 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 4.5: Function extracting the arguments of a primitive based on its name

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

The method for converting 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 4.6.

Listing 4.6: 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 accepts as parameters the arguments of an individual. Later the DEAP individual that is represented as a stack of functions is executed through a lambda expression that yields 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

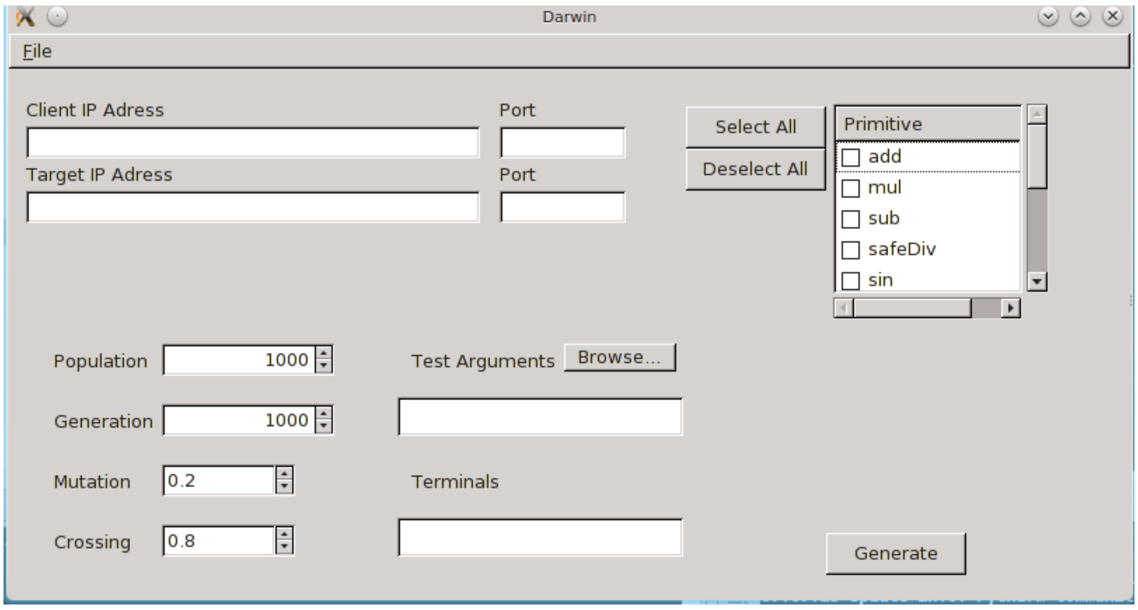
The same methods are used when generating a web service as seen in figure 4.7. The difference is the code is wrapped into a class that is used by CherryPy meaning the primitive signature and the lambda expression need to include the *self* notation. The web service has an index method that is called only by accessing the URL that returns the result of the individual.

Listing 4.7: 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

cherrypy.quickstart(ClonedWebService())
```

4.2.5 GUI

For python GUI development wxPython was a great choice because it offered simplicity and progress was easy. The framework supports loose coupling and high cohesion, it gives the opportunity to develop separate components and combine them later. This design feature was used to develop a component for each set of elements in the wireframe. Later on it was decided that advanced functionality should be opt out from the GUI and only basic configuration should be available. The result was the following GUI 4.2



Population 1000
Generation 1000
Mutation 0.2
Crossing 0.8

Figure 4.2: Graphical user interface for the Darwin framework that generates XML files

The GUI was split into five components 4.8.

1. *InitMenu()* initializes the menubar on top, creates each of the menu and sub-menu options and handles the functionality of the menu items.
2. *InitTexts()* creates a pane that has text input fields for the URLs of the target web service and the evaluator.
3. *InitAttributes()* returns a pane that has fields that specify the parameters used in the genetic program (e.g. crossing rate, mutation rate, population size, number of generations)
4. *InitPrimitives()* creates a primitives selector. The selector is created by combining several wxPython components like *scrollable panes*, *CheckListControl* and multiple sizers to organize the buttons and create a better feel.
5. *InitButtons()* simply creates a plane with the button *Generate* however it would be expected that later on more buttons would be added

Listing 4.8: The method in the GUI that initializes all the components used

```
def InitUI(self):
    self.InitMenu()
    textPanel = self.InitTexts()
    attrPanel = self.InitAttributes()
    primPanel = self.InitPrimitives()
    buttPanel = self.InitButtons()
```

When the *Generate* button is pressed the GUI is organizing all the data from the fields into a dictionary which is passed to the XML parser to generate the XML configuration file see listing 4.9

Listing 4.9: 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>
```

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 its components. 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 its 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 enabled 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. This was the unit testing strategy:

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.

Among the bugs found through unit tests was a code generation one. The python source was missing the imports specified by the user. When a simple system test was ran it couldn't be identified since the primitives used didn't include any external imports. However because of the modified configuration used for unit testing the bug was identified on time.

Another problem was with the pre-set primitives. Because of the mathematical nature of primitives division by 0 had to be avoided. In the simple system tests there were no examples with division by 0, the bug wasn't found until unit tests for each primitive were executed.

5.2 System Testing

There were many dependencies in the framework, which made testing a very difficult task. So instead of spending valuable time writing complicated unit tests for each function/method in the code, extensive system testing was initiated. 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 used 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.

Simple Testing involved running the Darwin framework against a simple, created for testing webservice. Using this simple webservice 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.

Wolfram Alpha Testing was used in the second stage. Wolfram Alpha [20] 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

An important feature for a framework is acceptance testing. Anonymous evaluation was done. 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 [21] for the framework, 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 with different levels of programming experience. The participants had to asses the quality of the tutorial, understanding of the framework and usability of the GUI. Each had to answer the following questions [22]:

1. How helpful was it to understand the basics of the framework?
2. How complex was it to create the first examples?
3. How complex was to create the advanced example?
4. How far on the tutorial did you reach in 30 minutes?

Second part of the evaluation participants had to asses the graphical user interface by answering the following questions:

1. How intuitive is it?
2. How simple is it?
3. Were there enough options presented?
4. Can you give additional feedback that can help improve the GUI?

SurveyMonkey was used for creating the survey. Through the evaluation the following problems in the GUI and the tutorial were identified and fixed:

Graphical User Interface

- Omitting *http* should still create a valid URL.
- If no arguments are provided an error message should pop up
- There should be help text for the *terminals* field to specify its format.

Tutorial

- There should be a link to the libraries/frameworks websites?
- The version of each library/framework should be specified?
- There should be more frequent code examples?

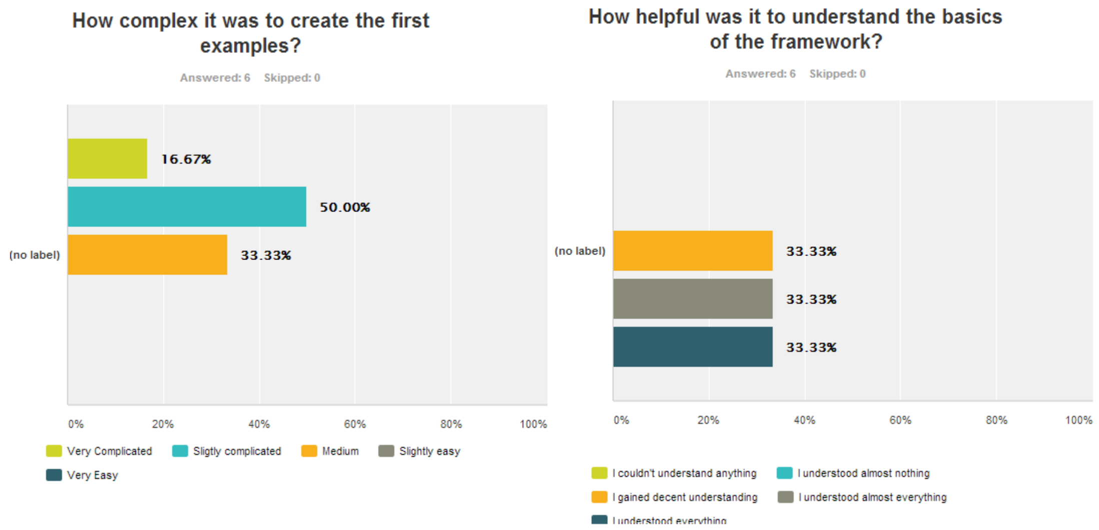


Figure 5.1: Plotted answers to the question: ”How complex it was to create the first examples?” and ”How helpful was it to understand the basics of the framework?”

As seen in the results from the survey, participants had hard time with running the first example as seen in fig 5.1. This was due to the multiple dependencies of the framework, lack of installation script and ambiguity in the tutorial. However in the end of the tutorial there was a high number participants that understood the basic framework functionality. Most of the participants managed to complete the tutorial in less than 30 minutes, however these 30 minutes didn’t include set up time.

For the GUI the results from participants were quite positive. The main problems with the GUI were the lack of feedback in the case of an error and lack of automation at the URL fields. Each of those problems was fixed by adding pop-up dialogues in case of a missing field, example for the *Terminals* field was added and the URL fields don’t require *http://* in front of the host name.

Chapter 6

Conclusion

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 its 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 motived 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 learned so quality development in the area could be done. The framework was related to web services, genetic programming, code generation, GUI which meant extensive research in each areas 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 solutions to on the web. 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 processed.

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. This 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 beyond 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 then 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 a 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.

6.3 Summary

The aim of the project was to develop a tool that uses genetic programming for cloning web services. After three sprints of development, system and usability testing, the project was a success. The tool was successful in satisfying the clients requirements and in cloning part of WolframAlpha's functionality. It managed to create a base for further development in the area. Hopefully Darwin will be used by the genetic programming community and help genetic programming projects in the university.

Bibliography

[1] W3, “W3.” http://www.w3.org/TR/ws-gloss/.

[2] M. G. E. Jerry Swan and J. 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.

[3] W. W. C. L. G. 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.

[4] GitHub, “Github.” https://github.com/.

[5] PivotalTracker, “Pivotaltracker.” http://www.pivotaltracker.com/.

[6] S. Famuyide, “Moscow: Requirements prioritization technique,”

[7] T. U. of Glasgow’s Raspberry Pi Project, “The university of glasgow’s raspberry pi project.” http://raspberrypicloud.wordpress.com/.

[8] R. Pi, “Raspberry pi.” http://www.raspberrypi.org/.

[9] WebPy, “Webpy.” http://webpy.org/.

[10] CherryPy, “Cherrypy.” http://www.cherrypy.org/.

[11] Flask, “Flask.” http://flask.pocoo.org/.

[12] Bottle, “Bottle.” http://bottlepy.org/docs/dev/index.html.

[13] Django, “Django.” https://www.djangoproject.com/.

[14] PYevolve, “Pyevolve.” http://pyevolve.sourceforge.net/.

[15] pySTEP, “pystep.” http://pystep.sourceforge.net/.

[16] PyGene, “Pygene.” http://freenet.mcnabhosting.com/python/pygene/.

[17] deap, “deap.” http://deap.gel.ulaval.ca/doc/default/index.html.

[18] lxml, “lxml.” http://lxml.de/.

[19] requests, “requests.” http://docs.python-requests.org/en/latest/.

[20] WolframAlpha, “Wolframalpha.” http://www.wolframalpha.com/.

[21] G. Kouzmov, “Darwin tutorial.” https://github.com/MrJew/darwin/blob/master/README.md.

[22] G. Kouzmov, “Surveymonkey.” https://www.surveymonkey.com/s/3GZH57M.