UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GABRIEL BARUFI VERAS

Supporting Swarm Debugging in interpreted programming languages

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marcelo Soares Pimenta
Co-orientador: Prof. Dr. Fabio Petrillo

Porto Alegre
2018

ACKNOWLEDGEMENTS

I want to thank my wife with all my heart for all her support and to encourage me to always be my better self. I want to thank my family to put me on the path of knowledge since my early age. And also thank my advisors Fábio Petrillo and Marcelo Pimenta for the patience and understanding to finish this journey.

ABSTRACT

Debugging a program takes time: nearly a third of the time spent in development is debugging and it seems that there is a strong correlation between the time of the first breakpoint and the time necessary to the debugging activity. The model of Swarm Debugging presents as being able to transfer the knowledge acquired in many sessions of debugging activity to future developers that would work in that same program. The model of Swarm Debugging was originally evaluated using the Java programming language being run over the Eclipse integrated development environment. This work evaluates the Swarm Debugging on the context of interpreted programming languages. Interpreted programming languages have been increasing in popularity and seven of the twenty most popular programming languages are interpreted programming languages. The meta-model of the concepts used in the Swarm Debugging is mapped to features described in the documentation of interpreted programming languages to demonstrate the possibility of supporting the Swarm Debugging in interpreted programming languages. Finally, the Firebug, an extension for Firefox web browser capable of debugging JavaScript language, and the PyDev, a plug-in to support Python language for the Eclipse integrated development environment, were changed put in practice the concepts developed in this work.

Keywords: Interactive debugging. Interpreted programming language. Crowd software engineering. Software maintenance. Software engineering.

Suporte à Depuração em Enxame em linguagens de programação interpretadas

RESUMO

Depurar um programa leva tempo: quase um terço do tempo gasto no desenvolvimento é depuração e parece haver uma forte correlação entre o tempo até o primeiro ponto de parada e o tempo necessário para a atividade de depuração. O modelo de Depuração em Enxame apresenta como sendo capaz de transferir o conhecimento adquirido em muitas sessões de atividade de depuração para futuros desenvolvedores que virão a trabalhar no mesmo programa. O modelo de Depuração em Enxame foi originalmente avaliado usando a linguagem de programação Java sendo executada sobre o ambiente de desenvolvimento integrado do Eclipse. Este trabalho avalia a Depuração em Enxame no contexto de linguagens de programação interpretadas. Linguagens de programação interpretadas têm aumentado em popularidade e sete das vinte linguagens de programação mais populares são linguagens de programação interpretadas. O meta-modelo dos conceitos usados na Depuração em Enxame é mapeado para recursos descritos na documentação de linguagens de programação interpretadas para demonstrar a possibilidade de suportar a Depuração em Enxame em linguagens de programação interpretadas. Finalmente, o Firebug, uma extensão para o navegador Firefox capaz de depurar a linguagem JavaScript, e o PyDev, um plug-in de suporte à linguagem Python para o ambiente de desenvolvimento integrado do Eclipse, foram alterados colocando em prática os conceitos desenvolvidos neste trabalho.

Palavras-chaves: depuração interativa, linguagem de programação interpretada. engenharia de software em multidão, manutenção de software, engenharia de software.

LIST OF FIGURES

LIST OF TABLES

LISTA DE ABREVIATURAS E SIGLAS

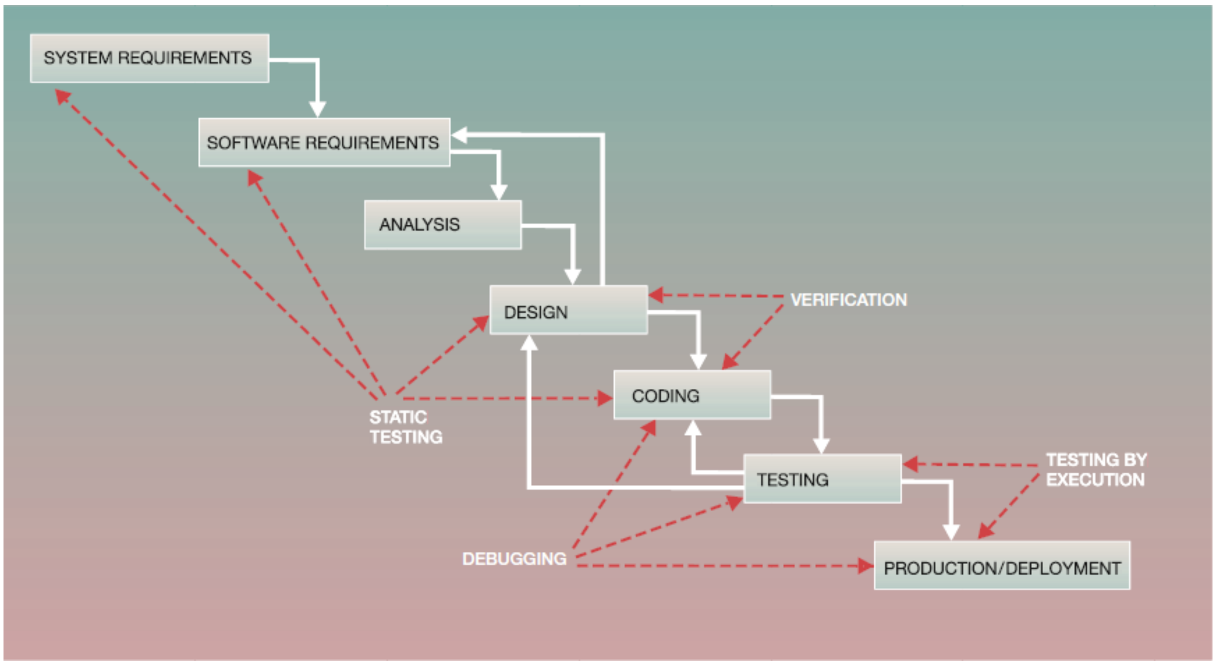
ECMA	European Association for Standardizing Information and Communication Systems
IDE	Integrated Development Environment
REST	Representational State Transfer
XML	Extensible Markup Language
HTTP	Hypertext Transfer Protocol
SDK	Software Development Kit
URL	Uniform Resource Locator

CONTENTS

1 INTRODUCTION

Software development is a huge task that vastly increases in complexity as the software being developed grows (BROOKS, 1987). The process of software development has some basic activities that are agnostic of the methodology of software development being used. Of these activities, debugging, testing and verification are essential to assure that the software being develop is reliable, that is, the software is performing accordingly to its specification bounded by the limitations of its environment (HAILPERN, 2002).

Figure 1.1 – Typical software development process



Source: Hailpern (2002, p. 6).

While debugging is an activity of finding and fixing defects (or bugs) that prevent the proper operations of software programs, debugger is a tool that helps the developer to analyze the source code while it runs one step at a time, trying to understand it and find a place to put breakpoints. The developers need to acquire runtime information and frequently execute the application using the debugger to understand the program, spending nearly a third of their time in this debugging activity (PETRILLO, 2015a).

The Swarm Debugger renders the activity of debugging more efficient with this transfer of knowledge among developers. The model of Swarm Debugging presents as being able to transfer the knowledge acquired in many sessions of debugging activity by gathering data from these debugging activities and presenting it through meaningful visualizations to future developers that would work in that same program. The work of Petrillo et al. (2016a) also demonstrated that there is no correlation between the numbers of invocations or the numbers of toggled breakpoints and elapsed task time. On the other hand, it also demonstrated that there is a strong correlation between the task's first breakpoint and the elapsed task time in the form of an inverse proportion. Additionally, a questionnaire among the test subjects showed a support to the usefulness of the Swarm Debugging visualization features. The model of Swarm Debugging was originally evaluated using the Java programming language being run over the Eclipse integrated development environment (PETRILLO, 2016b).

Interpreted programming languages have been increasing in popularity year after year in the last decades owing to its features that help rapid development of small programs (OUSTERHOUT, 1998). Differently from Java programming language, interpreted programming languages are usually of higher level and weakly typed which increases the productivity of the programmer although it runs slower than compiled programming languages causing them to be largely used in system integration. According to the Institute of Electrical and Electronics Engineers (CASS, 2018), seven of the twenty most popular programming languages are interpreted programming languages. Additionally, there are proposals that interpreted programming languages like Python be the first programming language of computer science courses (ZELLE, 1999). Also, the Web is increasing in interactivity thanks to interpreted programming languages embedded in the web pages (O'REILLY, 2005). As any other software development, the development of web pages also includes the exercise of debugging the source code directly on the web page using a web development tool.

This work evaluates the Swarm Debugging on the context of interpreted programming languages. The concepts of the meta-model of the Swarm Debugging are mapped to features described in the documentation of interpreted programming languages to demonstrate the possibility of supporting the Swarm Debugging in interpreted programming languages. The Python and JavaScript programming languages were chosen as they are among the most popular programming languages according various sources. A compiling of the twenty most popular programming languages according to the IEEE Spectrum (CASS, 2018), the TIOBE Programming Community index (TIOBE SOFTWARE BV, 2018) and the PYPL Index (CARBONNELLE, 2018) includes the two interpreted programming languages used in this work plus the fact that Python is the most popular programming languages in two indexes (IEEE and PYPL) and JavaScript is the third most popular programming language in one index (PYPL). Table 2.1 provides the 20 most popular programming languages by each index.

Finally, the Firebug, an extension for Firefox web browser capable of debugging JavaScript language, and the PyDev, a plug-in to support Python language for the Eclipse integrated development environment, were changed put in practice the concepts developed in this work.

Table 2.1 – Most popular programming languages in August 2018

Index	*IEEE*	*TIOBE*	*PYPL*
1	Python	Java	Python
2	C++	C	Java
3	C	C++	JavaScript
4	Java	Python	C#
5	C#	Visual Basic .NET	PHP
6	PHP	C#	C/C++
7	R	PHP	R
8	JavaScript	JavaScript	Objective-C
9	Go	SQL	Swift
10	Assembly	Assembly	Matlab
11	Matlab	Swift	Ruby
12	Scala	Delphi/Object Pascal	TypeScript
13	Ruby	Matlab	VBA
14	HTML	Objective-C	Visual Basic
15	Arduino	Ruby	Scala
16	Shell	Perl	Kotlin
17	Perl	Go	Go
18	Swift	R	Perl
19	Processing	Visual Basic	Lua
20	Lua	PL/SQL	Rust

1.1 Main Objective

The rising importance of the interpreted programming languages provides the first step to expand the Swarm Debugging approach into different paradigms of programming languages. If all meta-concepts of the Swarm Debugging could be mapped to a feature of a target language, then we could say that the Swarm Debugging can support the programming languages providing all advantages of the Swarm Debugging to the debugging activities in the programming language. This work tries to find this mapping of the meta-concepts of the Swarm Debugging into the features of the interpreted programming languages.

1.2 Secondary Objectives

In addition to mapping the features of interpreted programming languages to the meta-concepts of the Swarm Debugging, this work also evaluates the implementation of a version of the Swarm Debug Tracer to a debugging tool of the JavaScript and Python programming tool. Namely, the debugging tools changed to have the Swarm Debug Tracer embedded in them was the Firebug web browser extension and the PyDev Eclipse IDE plug-in.

1.3 Structure of this work

This work is divided in six chapters:

- **Chapter 2** presents the main concepts needed to understand the objectives of this work.
- **Chapter 3** provides the proposal of how to support the approach of Swarm Debugging in interpreted programming languages.
- **Chapter 4** provides the proposal of how to support the approach of Swarm Debugging in the JavaScript programming language.
- **Chapter 5** provides the proposal of how to support the approach of Swarm Debugging in the Python programming language.
- **Chapter 6** presents the implementations of two test of concept for the JavaScript and the Python programming language.
- **Chapter 7** gives the final remarks on this work including contributions, limitations of this work and future works.

2 BACKGROUND

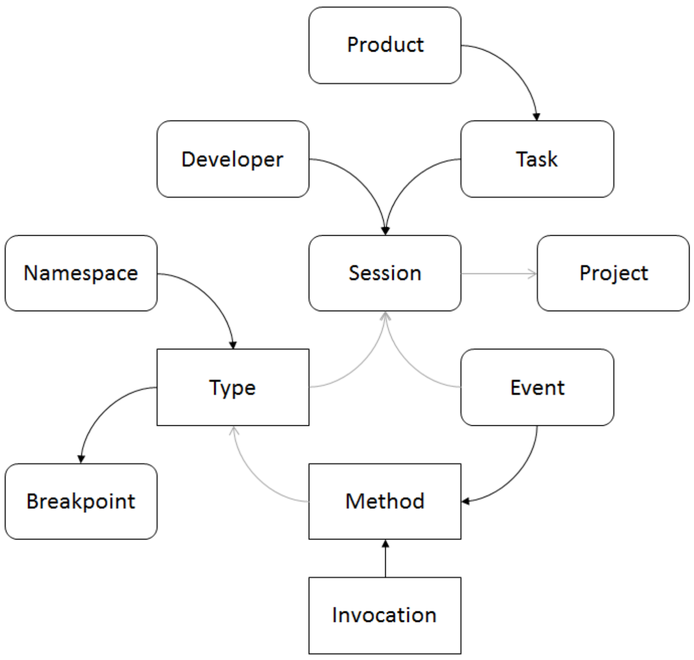
This section presents three main topics that should be clear before explaining how to support Swarm Debugging in interpreted programming languages. These are: how Swarm Debugging works, how the JavaScript programming language works and how the Python programming language works.

2.1 Swarm Debugging

The Swarm Debugging concept tries to improve software development activities like interactive debugging using on the concept of swarm intelligence (PETRILLO, 2016b). The Swarm Debugging concept works by collecting the actions of several developers during sessions of interactive debugging and transform this data into knowledge through visualizations and searching tools designed to be shared among other developers.

The meta-model representing the central concepts of the Swarm Debugger is shown in figure 2.1. The concepts of the Swarm Debugger data-model comprise the main object-oriented concepts like Type and Method plus the necessary associations between them like Invocation and Access.

Figure 2.1 – The Swarm Debugger meta-model



Source: Petrillo (2016a, p. 59).

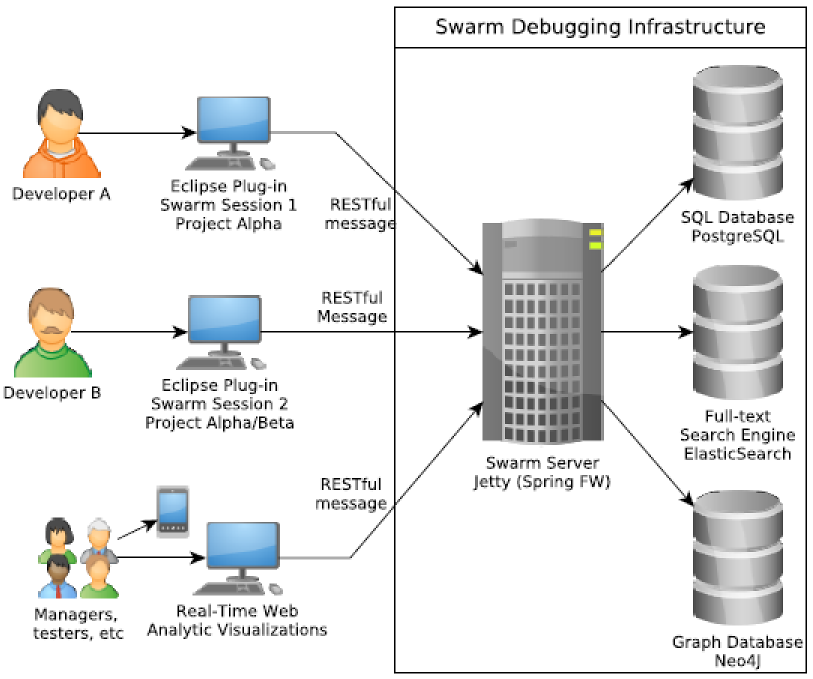
The Swarm Debugging approach to software development is supported by an infrastructure (called Swarm Debugger Infrastructure) containing tools for store and analyze debugging data created by the debugging activities of developers while using the Eclipse IDE and its integrated debugger. The Swarm Debugger Infrastructure is organized in three main modules: tracer, services and views.

The Swarm Debug Tracer is an Eclipse plug-in responsible to register the interactive debugging process. Using this plug-in, developers can authenticate their credentials and create a debug session that will register the developer's actions through the debugging process. When the developer starts the debugging process, he or she usually mark one point in the source code where the program should stop to be analyzed step-by-step by the developer: This mark in the source code is called *breakpoint*. During the step-by-step analysis, the developer can choose to follow the program execution inside the method being called by the code snippet currently being analyzed: This action of following the execution inside the method is called *step into*. The Swarm Debugger Tracer is developed to only take breakpoints and step into as relevant events following the approach of Fleming et al. (FLEMING, 2013).

The Swarm Debug Services (Figure 2.2) provides an infrastructure to store and share debugging data from and between developers which are composed by the following services: Swarm RESTful API, SQL Query Console, Full-text Search Engine, Dashboard Service, Graph Querying Console. For this work, only the Swarm RESTful API is extensively used as it is the connection point between the Swarm Debugger Tracer and the rest of the Swarm Debugger Infrastructure.

An instance of the Swarm Debug Services receives messages sent from the Swam Debugger Tracer. Then the Swarm Debug Services stores the received messages in three specialized persistence mechanisms: an SQL database (*PostgreSQL*), a full-text search engine (*ElasticSearch*) and a graph database (*Neo4J*). The three persistence mechanisms use a similar set of concepts to define the semantics of the Swarm Debugger Tracer messages.

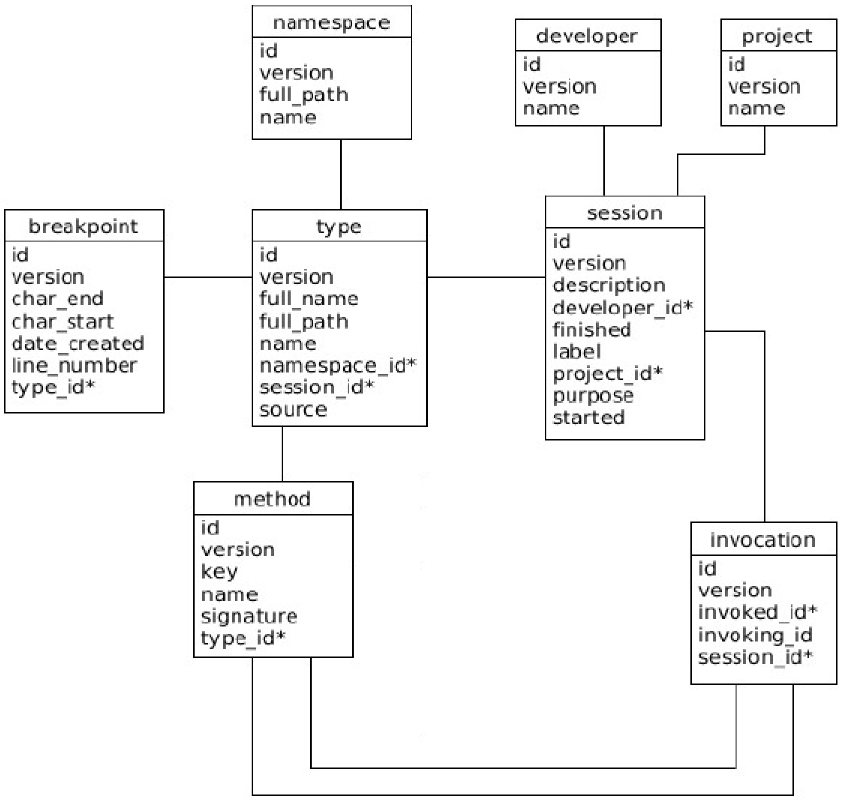
Figure 2.2 – The Swarm Debugger Services architecture


Swarm Debugging Infrastructure
Developer A
Eclipse Plug-in Swarm Session 1 Project Alpha
RESTful message
Developer B
Eclipse Plug-in Swarm Session 2 Project Alpha/Beta
RESTful Message
Managers, testers, etc
Real-Time Web Analytic Visualizations
RESTful message
Swarm Server Jetty (Spring FW)
SQL Database PostgreSQL
Full-text Search Engine ElasticSearch
Graph Database Neo4J

Source: Petrillo (2016a, p. 60).

Figure 2.3 presents these concepts using an entity-relationship model.

Figure 2.3 – The Swarm Debugger metadata



Source: Petrillo (2016a, p. 60).

Each entity has the following definition:

- **Developer** is the user of the plug-in.
- **Product** is the software project which is being debugged.
- **Task** is the reference to the debugging purpose.
- **Session** is the Swarm Debugger session.
- **Type** is the class or interface contained in the project.
- **Method** is the calling method or method being called.
- **Namespace** is the container of types.
- **Invocation** is the link between the method calling and the method being called.
- **Breakpoint** is the information of breakpoints created by the user.

2.2 The JavaScript language

JavaScript was defined in 1997 by Brendan Eich from Netscape Communication Corp. for use with the version 2.0 of its Navigator web browser in 1995. A year later, Microsoft Corp. added support to JavaScript in the version 3.0 of its Internet Explorer web browser. Netscape Communication Corp. submitted JavaScript to Ecma International to set a standard specification in 1996 and, in 1997, Ecma International published the first standard version of the JavaScript specification under the name of ECMAScript (ECMA INTERNATIONAL, 2018).

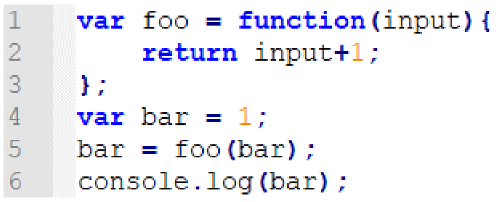
This work follows the specification of the ECMAScript but is limited to the features present in the JavaScript implementation of the most used browsers as provided by Mozilla Developer Network (MOZILLA DEVELOPER NETWORK, 2018). For reference, the most used web browsers are Google Chrome, Apple Safari, Mozilla Firefox and Microsoft Internet Explorer (WIKIMEDIA FOUNDATION, 2018).

2.2.1 Lexical environment

A lexical environment is a specification type (meta-values that are used within algorithms to describe the semantics) used to define the association of *Identifiers* to specific variables and functions based upon the lexical nesting structure of JavaScript code. There are three types of Lexical Environment: *global environment*, *module environment* and *function environment*.

A *global environment* is a lexical environment which does not have an outer environment. A *module environment* is a lexical environment that contains the bindings for the top-level declarations of a Module. A *function environment* is a lexical environment that corresponds to the invocation of an JavaScript function object.

Figure 2.4 – Lexical environment in JavaScript



```
var foo = function(input){
    return input+1;
};
var bar = 1;
bar = foo(bar);
console.log(bar);
```

In the example of Figure 2.4, *foo* and *bar* is defined in the global environment while input in defined in the function environment.

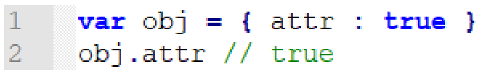
2.2.1 Modules

Modules are not covered in this work because this functionality is not yet supported in any implementation of JavaScript of the most used web browsers (MOZILLA DEVELOPER NETWORK, 2018) although there is definition for modules in the ECMAScript specification.

2.2.1 Objects

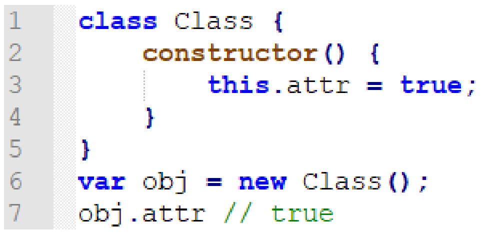
In the JavaScript programming language, objects are not strictly based on classes as much as the Java programming language. There is more than one way to create an object like using the literal notation (Figure 2.5) or the class constructor (Figure 2.6).

Figure 2.5 – Creating a class using literal notation in JavaScript



```
var obj = { attr : true }
obj.attr // true
```

Figure 2.6 – Creating a class using class constructor in JavaScript

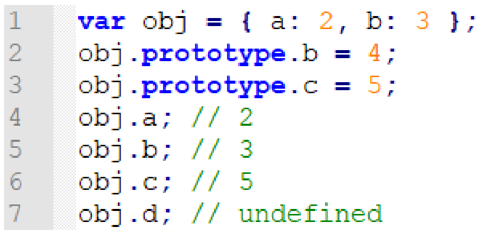


```
class Class {
    constructor() {
        this.attr = true;
    }
}
var obj = new Class();
obj.attr // true
```

All objects are logically collections of properties, but there are multiple forms of objects that differ in their semantics for accessing and manipulating their properties. A property of an object can be explained as a variable or another object that is attached to the object.

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property (referred to as [[Prototype]]) which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. Null has no prototype by definition and acts as the final link in this prototype chain. Nearly all objects in JavaScript are instances of the *Object* object which sits on the top of a prototype chain.

Figure 2.7 – Prototypes in JavaScript



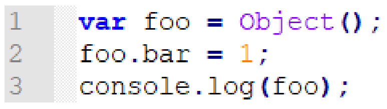
```
var obj = { a: 2, b: 3 };
obj.prototype.b = 4;
obj.prototype.c = 5;
obj.a; // 2
obj.b; // 3
obj.c; // 5
obj.d; // undefined
```

In the example of Figure 2.7: *obj* has the property *a* therefore *obj.a* has value 2; *obj* has the property *b* therefor *obj.b* has value 3 despite its prototype also having the property *b*; *obj* does not have the property *c* but its prototype has the property *c*} therefore *obj.c* has value 5; finally *obj* does not have the property *d* neither its prototype has the property *d* therefore *obj.d* has *undefined* value.

2.2.1.1 Built-in objects

There are certain built-in objects available whenever an JavaScript Script or Module begins execution. The unique *global* object which is part of the lexical environment of the executing program and is created before control enters any execution context. Others built-in objects are accessible as initial properties of the global object or indirectly as properties of accessible built-in objects.

Figure 2.8 – Built-in objects in JavaScript



```
var foo = Object();
foo.bar = 1;
console.log(foo);
```

In the example of Figure 2.8, *Object* is a property of the *global* object. Although there is not an explicit reference to the *global* object, every variable, function, etc. defined in the global environment is a property of the *global* object.

2.2.1.2 Function objects

In JavaScript, function objects encapsulate parameterized JavaScript code closed over a lexical environment and support the dynamic evaluation of that code. A function object in JavaScript is an ordinary object and has the same internal slots and the same internal methods as any other ordinary object. All JavaScript function objects have the [[Call]] internal method defined here. JavaScript functions that are also constructors have in addition the [[Construct]] internal method. Anonymous functions objects that do not have a contextual name associated with them by this specification do not have a name own property but instead inherit the name property of %FunctionPrototype%.

Figure 2.9 – Function objects in JavaScript

```
function square(number) {
    return number * number;
}
```

2.2.1.3 Class Objects

Classes in JavaScript works almost equals to any other object-oriented programming language that also have classes. The example in Figure 2.9 shows one way of creating an object by calling the class constructor:

Figure 2.10 – Class objects in JavaScript

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
var p = new Rectangle();
```

2.3 The Python language

Python is an interpreted dynamic programming language developed by Guido van Rossum in early 1990s that uses the object-oriented programming paradigm. As an interpreted programming language, Python does not need to be translated to machine language to be executed and, as a dynamic programming language, Python executes, at run time, actions that would otherwise be executed during compilation (VAN ROSSUM, 2003).

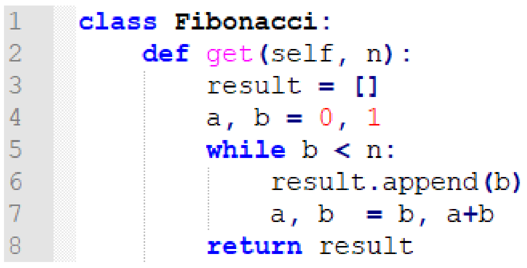
In the Python programming language, all data is an *object* composed of *identity*, *type* and *value*. The identity is an integer number that never changes once the object is created. The type determines the domains of a given object and what operations it supports. The value of objects may be changeable or not, thus objects whose values can be changed are called *mutable* and objects whose values cannot be changed are called *immutable*.

2.3.1 Modules

Modules are files containing source code written in the Python programming language and its name is the name of the file without the *.py* suffix. Modules can be imported using the *import* statement creating an object that enables the reference to objects declared inside the module.

Consider the *fibonacci.py* module file:

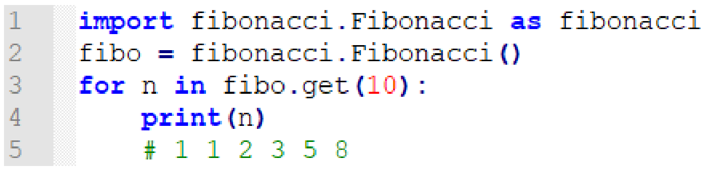
Figure 2.11 – Function objects in Python – *fibonacci.py* file



```
class Fibonacci:
    def get(self, n):
        result = []
        a, b = 0, 1
        while b < n:
            result.append(b)
            a, b  = b, a+b
        return result
```

Now, the *fibonacci.py* module file can be used as following:

Figure 2.12 – Function objects in Python – importing *fibonacci.py* file



```
import fibonacci.Fibonacci as fibonacci
fibo = fibonacci.Fibonacci()
for n in fibo.get(10):
    print(n)
    # 1 1 2 3 5 8
```

2.3.2 Packages

A collection of multiple modules can be organized hierarchically inside a folder to create packages and sub-packages. The package and each of its sub-packages must have an initialization file called *__init__.py* to it be considered a package. Using this system, it is possible to import the package, modules inside the package or even specific objects of a given module.

Figure 2.13 – Package structure for a generic sound library

```
sound/                      // Top-level package
    __init__.py             // Initialize the sound package
    formats/                // Subpackage for file format conversions
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/                // Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/                // Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

The following piece of code shows how it is possible to use a specific module of a package or an effect from the sound library based on the example package given above.

Figure 2.14 – Importing a specific module of a package in Python

```python
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

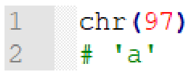
2.3.3 Callable types

The Python programming language defines some types that have the function call operation. This operation enables the type to behave as a function that executes a piece of code. The Python programming language defines the following callable types: built-in functions, user-defined functions, generator functions, lambda expressions, instance methods, built-in methods, classes and class instances.

2.3.3.1 Built-in functions

Build-in functions are all function objects defined by the implementation of the interpreter of the Python programming language. Examples include the *chr()* function which returns the Unicode character of an integer.

Figure 2.15 – Built-in functions in Python

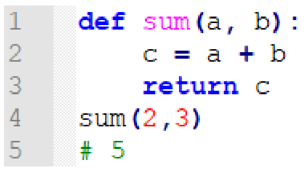


```python
chr(97)
# 'a'
```

2.3.3.2 User-defined functions

User-defined functions are callable objects in form of functions that can be defined by the user. The following piece of code examples how to create user-defined functions.

Figure 2.16 – User-defined functions in Python



```python
def sum(a, b):
    c = a + b
    return c
sum(2,3)
# 5
```

2.3.3.3 Generator functions

Generator functions are any function object that uses the *yield* statement thus creating an iterator. When the generator function is called and reached the yield statement, the program flow stops the function and returns an iterator. Any further call to the generator function resumes the execution of the generator function from the former yield statement.

Figure 2.17 – Generator functions in Python

```python
def range(start,end):
    now = start
    while now <= end:
        yield now
        now += 1
for i in range(1,9):
    print(i)
    # prints all integers from 1 to 9
```

2.3.3.4 Lambda expressions

Lambda expressions are just like functions except defining a lambda expression create an anonymous function that have only one statement. The following piece of code shows lambda expressions in the example of a function that takes another function and a list as argument and apply the function to each element of the list.

Figure 2.18 – Lambda expressions in Python

```python
domain = [1,2,3,4]
def apply(f,l):
    return list(f(x) for x in l)
apply(lambda x: x*x, domain)
# [1,4,9,16]
```

2.3.3.5 Classes

Classes are objects that when called returns a new instance object of that class. Optionally, classes can have a special *__init__* method that will be executed when a class is called to create an instance object.

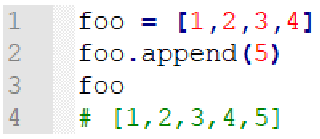
Figure 2.19 – Callable classes in Python

```python
class MyFoo:
    name = "unnamed"
    def __init__(self, newName):
        self.name = newName
    def getName(self):
        return self.name
myName = MyFoo("foo")
myName.getName()
# foo
```

2.3.3.6 Built-in methods

Build-in methods are all methods pertaining to objects and defined by the implementation of the interpreter of the Python programming language. Examples include the *list.append()* of any *list* object as presented in the piece of code below.

Figure 2.20 – Built-in methods in Python

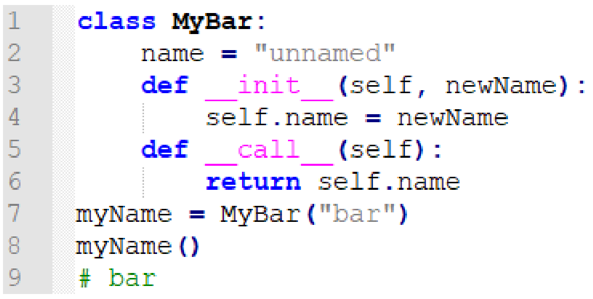


```
foo = [1,2,3,4]
foo.append(5)
foo
# [1,2,3,4,5]
```

2.3.3.7 Class instances

Class instances can be callable by having the special __*call*__ method implemented. The __*call*__ method will be executed whenever the class instance is called like it was an function or method.

Figure 2.21 – Class instances in Python

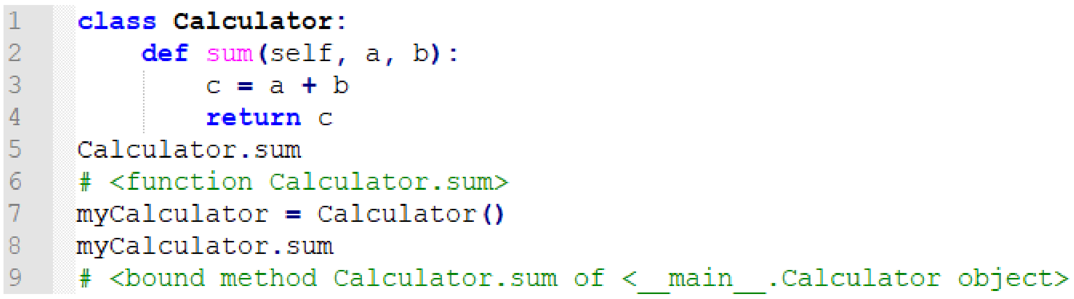


```
class MyBar:
    name = "unnamed"
    def __init__(self, newName):
        self.name = newName
    def __call__(self):
        return self.name
myName = MyBar("bar")
myName()
# bar
```

2.3.3.8 Instance methods

Instance methods are objects that connects a class, a function object and an instance object. The following piece of code shows that accessing the method from the class returns a function but accessing the method from the class instance returns a method bound to the instance object.

Figure 2.22 – Instance methods in Python



```
class Calculator:
    def sum(self, a, b):
        c = a + b
        return c
Calculator.sum
# <function Calculator.sum>
myCalculator = Calculator()
myCalculator.sum
# <bound method Calculator.sum of <__main__.Calculator object>
```

3 SWARM DEBUGGING IN INTERPRETED PROGRAMMING LANGUAGES

To implement the Swarm Debug Tracer for interpreted programming languages, it is necessary that the Swarm Debugging meta-model supports concepts that are intrinsic to interpreted programming languages. Therefore, this section compares the key concepts of interpreted programming languages with the concepts of Swarm Debugging meta-model defined by Petrillo (2016a).

The Swarm Debugging meta-model makes mention to source-code versioning in the meta-model model but accepts it as limitation that should be addresses in future works. The meta-model proposed in this work does not make references to versioning and therefore focus is kept on defining features necessary for supporting interpreted programming languages in the Swarm Debugging.

3.1 Namespace meta-concept

The proposed Namespace meta-concept is the same as the path of the file in the file system because namespaces already work in that manner in most programming languages. For example, the namespace is a facade to the file path of the source code in the Java programming language. The JavaScript running in web browsers does not support packages yet and packages is just a way to unify a collection of modules in Python. In the end, packages boil down to files and folders in most languages.

Table 3.1 presents the proposed Namespace meta-concept.

Table 3.1 – Proposed Namespace meta-concept

Namespace:	
id	internal id
full_path	path of the file within the system
name	name of the script file

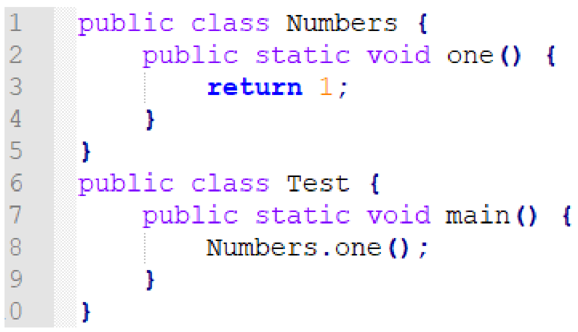
3.2 Object, Method and Call meta-concepts

The previous sections of this work have shown that interpreted programming languages uses the object-oriented programming paradigm much less based on classes than Java which was the main programming language used to conceive the Swarm Debugging. In the Java programming language, all methods belong to a class therefore it makes sense that

Swarm Debugging follow these steps and stated that all methods had a reference to class in the meta-model.

The source code in Figure 3.1 was written in the Java programming language and exemplifies a typical case of a method calling another method. In this example, the *main* method of *Test* class calls the *one* method of *Numbers* class.

Figure 3.1 – Method calling in Java



```java
public class Numbers {
    public static void one() {
        return 1;
    }
}
public class Test {
    public static void main() {
        Numbers.one();
    }
}
```

On the other hand, interpreted programming languages have a trend to use objects as its main concept instead of classes and to have objects that can be executed as if they were a method. At least, those are trends found in the studied languages which are also the most popular interpreted programming languages. The first mentioned trend means that everything is an object and its attributes are also objects (except when it is a built-in data type instead of an object). In programming languages based mainly in objects, there is no need to declare a class then instantiate it in an object as objects could be created directly. The second trend mentioned means that some objects are of types that can be called in the same way as class methods without the requirement that the called object be a method belonging to a class.

Finally, to better represent the object-oriented paradigm that is used by all programming languages mentioned, Type should be called Object as it refers objects more so than anything else.

Table 3.2 presents the proposed Object meta-concept.

Table 3.2 – Proposed Object meta-concept

Object:	
id	internal id
name	name of object if there is one
namespace_id	namespace of the object
session_id	session where this object appeared

Table 3.3 presents the proposed Method meta-concept.

Table 3.3 – Proposed Method meta-concept

Method:	
id	internal id
name	name of method if there is one
signature	defined parameters of the method
object_id	object that owns the method

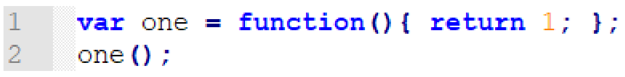
Table 3.4 presents the proposed Call meta-concept.

Table 3.4 – Proposed Call meta-concept

Call:	
id	internal id
caller_id	method that calls
called_id	method being called
session_id	session where this call happened

The following examples written in the JavaScript programming language illustrates the nesting of objects with some of them also being callable objects. Tables representing the resulting meta-data of the debugging session of each example is presented together with the source code.

Figure 3.2 – A callable object named *one*



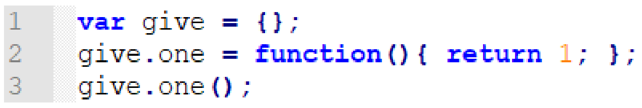
```
1  var one = function(){ return 1; };
2  one();
```

In the example of Figure 3.2, it is made a step from the *__main__* function of the *script.js* object into the *one* function of the *script.js* object in line 2. This debugging session results in the meta-data contained in Table 3.5.

Table 3.5 – Resulting meta-data model of a callable object

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	__main__	1	1	1	2
		2	number	2			

Figure 3.3 – A callable object named *one* owned by an object named *give*



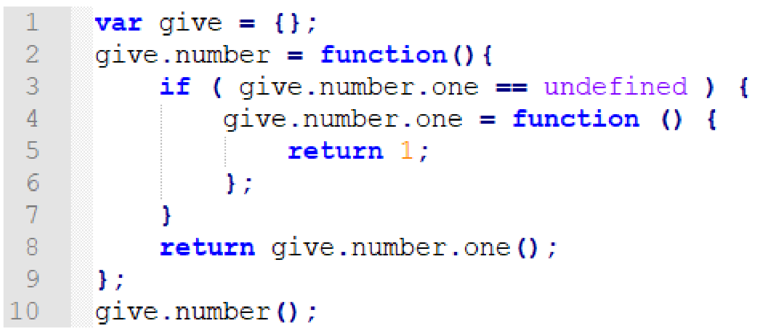
```
1  var give = {};
2  give.one = function(){ return 1; };
3  give.one();
```

In the example of Figure 3.3, it is made a step from the *__main__* function of the *script.js* object into the *one* function of the *give* object in line 3. This debugging session results in the meta-data contained in Table 3.6.

Table 3.6 – Resulting meta-data model of a callable object owned by an object

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	__main__	1	1	1	2
2	give	2	one	2			

Figure 3.4 – A callable object *one* owned by a callable object *number* which in turn is owned by an object *give*



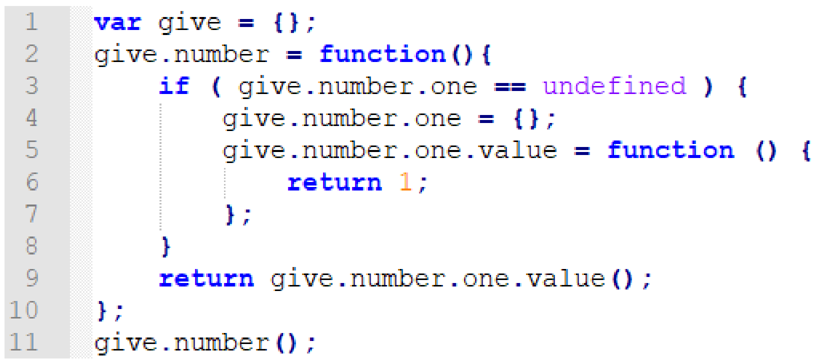
```
1  var give = {};
2  give.number = function(){
3      if ( give.number.one == undefined ) {
4          give.number.one = function () {
5              return 1;
6          };
7      }
8      return give.number.one();
9  };
10 give.number();
```

In the example of figure 3.4, it is made a step from the *__main__* function of the *script.js* object into the *number* function of the *give* object in line 10 then a step from the *number* function of the *give* object into the *one* function of the *number* object in line 8. This debugging session results in the meta-data contained in Table 3.7.

Table 3.7 - Resulting meta-data model of a callable object owned by a callable object owned by a callable object

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	__main__	1	1	1	2
2	give	2	number	2	2	2	3
3	number	3	one	3			

Figure 3.5 – A callable object *value* owned by an object *one* and this object *one* is owned by a callable object *number* which in turn is owned by an object *give*



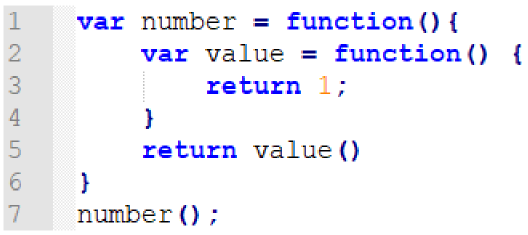
```
var give = {};
give.number = function(){
    if ( give.number.one == undefined ) {
        give.number.one = {};
        give.number.one.value = function () {
            return 1;
        };
    }
    return give.number.one.value();
};
give.number();
```

In the example of Figure 3.5, it is made a step from the *__main__* function of the *script.js* object into the *number* function of the *give* object in line 11 then a step from the *number* function of the *give* object into the *value* function of the *one* object in line 9. This debugging session results in the meta-data contained in Table 3.8.

Table 3.8 – Resulting meta-data model of a callable object owned by an object owned by a callable object owned an object

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	__main__	1	1	1	2
2	give	2	number	2	2	2	3
3	one	3	value	3			

Figure 3.6 – A callable object named *value* not owned but declared inside another callable object named *number*



```
1 var number = function(){
2     var value = function() {
3         return 1;
4     }
5     return value()
6 }
7 number();
```

In the example of Figure 3.6, it is made a step from the *__main__* function of the *script.js* object into the *number* function of the *script.js* object in line 7 then a step from the *number* function of the *script.js* object into the *value* function of declared inside the *number* object in line 1. In cases like this, the function is created referencing the object where it was created. This debugging session results in the meta-data contained in Table 3.9.

Table 3.9 – Resulting meta-data model of step-into a function declared inside another function

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	__main__	1	1	1	2
2	number	2	number	1	2	2	3
		3	value	3			

The previous examples also showed another characteristic of interpreted languages and how they are handled in this work: the program starts to run from the first line of the script file. The name of the script file is recorded as the object and the function is a special predefined entry (e.g. *__main__* and *script.js* in the previous examples).

The lambda expressions and anonymous functions are another case where special entries on the meta-concept should be defined. The following example provides a lambda expression written in the Python programming language.

Figure 3.7 – Example of a callable object using lambda expression

```
1   domain = [1,2,3,4]
2   def apply(f,l):
3       return list(f(x) for x in l)
4   apply(lambda x: x*x, domain)
5   # [1,4,9,16]
```

In the example of Figure 3.7, a step is made from the *apply* function of the *script.py* object into the *__lambda__* function of the *apply* object in line 3. This debugging session results in the meta-data contained in Table 3.10.

Table 3.10 – Resulting meta-data model of step-into a lambda expression

Object		Function			Call		
id	Name	id	name	id	id	caller_id	called_id
1	script.js	1	apply	1	1	1	2
2	Apply	2	__lambda__	1			

Figure 3.8 – Recursive function that calculates the Fibonacci numbers

```
1   def fibonacci(n):
2       if n <= 2:
3           l = [0,1]
4           return l[0:n]
5       if n >= 3:
6           l = fib(n-1)
7           l.append( l[-1] + l[-2] )
8           return l
9   fibonacci(10)
10  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Recursion is an important feature of many programming languages. In the example of Figure 3.8, a step is made from the *fibonacci* function of the *script.py* object into the *fibonacci* function of the *script.py* object in line 6 then another step-into in the same line 6. This debugging session results in the meta-data contained in Table 3.11.

Table 3.11 – Resulting meta-data of step-into a recursive function

Object		Function			Call		
id	name	id	name	id	id	caller_id	called_id
1	script.js	1	fibonacci	1	1	1	1
					2	1	1

3.3 Breakpoint meta-concept

In the Swarm Debugging as proposed by Petrillo (2016a), every breakpoint is set inside a class following the logic of the Java programming language. In interpreted programming languages, breakpoints need an entry specially defined for when the breakpoint is set outside an object. This usually happens when the breakpoint is set in a line of the script file that does not correspond to any object in the source code.

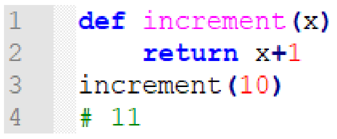
Table 3.12 presents the proposed Breakpoint meta-concept:

Table 3.12 – Breakpoint meta-concept

Breakpoint:	
id	internal id
date_created	creation date of the breakpoint
line_number	line number inside the script
object_id	object where the breakpoint was created

Source: Veras (2018, p. 31).

Figure 3.9 – A breakpoint set in the global environment



```
1  def increment(x)
2      return x+1
3  increment(10)
4  # 11
```

In the example of Figure 3.9, a breakpoint is set in line 3 which does not reference the inside of any object. In this case, the breakpoint is defined as being set in the *script.py* object. This debugging session results in the meta-data contained in Table 3.13:

Table 3.13 – Resulting meta-data of setting a breakpoint in the main script scope

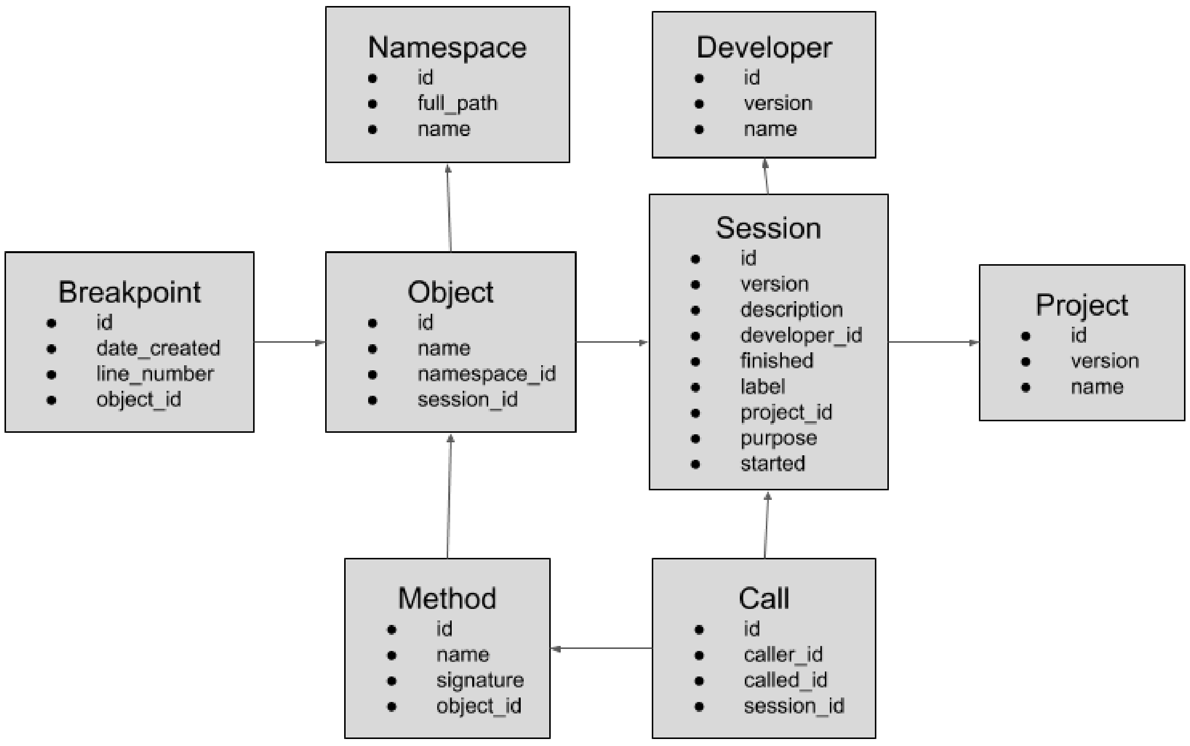
Object		Breakpoint		
id	name	id	line	obj_id
1	script.js	1	4	1

Source: Veras (2018, p. 31).

3.4 Proposed Swarm Debugging meta-model

No changes are proposed to the Session, Developer and Project meta-concepts because they are not related to the programming language being used in the Swarm Debugging.

Figure 3.10 – The Swarm Debugging meta-model proposed in this work



The Figure 3.10 presents the Swarm Debugging meta-concept as proposed by this work. Comparing Figure 2.4 with Figure 3.10, it shows that there is not much difference between the original meta-concept and the new meta-concept being proposed. The largest difference between the Swarm Debugging as initially proposed by Petrillo (2016a) and the Swarm Debugging as proposed in this work is the new set rules necessary to create entries when it is not clear how to define some features of interpreted programming languages in the original meta-model of the Swarm Debugging. This uncertainty exists because the core differences between the Java programming language and the many interpreted programming languages where the former is strongly based on classes and the latter are usually more based on objects.

4 SWARM DEBUGGING IN JAVASCRIPT

The JavaScript programming languages may differ in some points from other interpreted languages. One of these points is the concept of namespace, module or package may it be the definition or the inclusion of it as a feature of the language. Other points are how to handle breakpoints and calls involving the top-level scope and the many ways objects can be created.

4.1 Definitions for the Namespace meta-concept in JavaScript

The Swarm Debugging has a container of types called Namespace in its meta-model. This concept of Namespace is like what is called Package in the Java programming language and Module in the JavaScript programming language. Unfortunately, translating Module to Namespace has no practical use because the Modules of JavaScript has no implementation in any version of JavaScript that runs in web browsers currently (MOZILLA DEVELOPER NETWORK, 2018). One work-around to keep using Namespace in a meaningful way is to translate the file path of the script file being run to Namespace.

4.2 Definitions for the Breakpoint and Call meta-concepts in JavaScript

The Swarm Debugging presupposes that every breakpoint is created inside a Type and that Types have names. This is false in JavaScript as the program starts from the first line of a script file and breakpoints inside the starting global environment of the program is not considered to be inside a class, method or anything else. Furthermore, any method calls made from the global environment will not have a Type to reference as a source of the method call. A special Type could be used to indicate the global environment when a breakpoint in created or a method is called.

4.3 Definitions for the Object and Method meta-concepts in JavaScript

The Types as specified by the Swarm Debugging acts as a meta-concept for classes and interfaces, but one could write an object-oriented program without declare a single class and interfaces does not even exist in the JavaScript programming language. That happens because JavaScript is weakly typed, its objects are not strictly based on classes and can be created directly without being an instance of a class or type. Additionally, JavaScript objects can be anonymous which means that their identification for use in the Swarm Debugging is even more hampered.

Figure 4.1 – Anonymous functions in JavaScript

```
(function() { console.log("Hello World"); })();
```

The JavaScript source code in Figure 4.1 creates an anonymous function object and then call it. After the function runs, the object vanishes from the global environment as it is not referenced anymore. When debugging this line of code and calling for a Step Into event, there is no source Type as it is a call made from the global environment and the target Type is an anonymous function. Having a name field in Type loses its significance entirely when many anonymous objects starts to appear in the source code.

Functions in JavaScript are just like any other object except they have a [[Call]] internal method that dynamically evaluates a runnable piece of code when the function object is called. To translate JavaScript function calls to be used in the Swarm Debugging, the function object will be considered an Object and the [[Call]] internal method will be recorded as the method being called when the function object is called.

Figure 4.2 – Dynamic functions in JavaScript

```
function mainfunc (func){
  this[func].apply(null, Array.prototype.slice.call(arguments, 1));
}
function target(a) {
  alert(a)
}
mainfunc("target", "Hello");
```

The JavaScript source code in Figure 4.2 is an example of how to implement a dynamic function object with dynamic parameters. Despite functions objects and methods having a signature defining its parameters, they can be omitted, or additional parameters can be passed when calling a function or method. The fact that parameters of a function object are not set on stone lessens the meaning of having a signature field in the Method meta-concept although it does not hinder it either.

5 SWARM DEBUGGING IN PYTHON

The Python programming languages may differ in some points from other interpreted languages. One of these points is the concept of namespace, module or package may it be the definition or the inclusion of it as a feature of the language. Other points are how to handle breakpoints and calls involving the top-level scope and the many ways objects can be created.

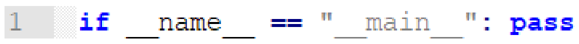
5.1 Definitions for the Namespace meta-concept in Python

Modules and packages have the same name of its folder in the Python programming language thus they may be converted into the namespaces of the Swarm Debugging as they have similar functions of the packages of the Java programming language. This approach also is very similar to the approach being used in the JavaScript programming language thus the concept that represents the namespace in Java, JavaScript and Python programming languages are nearly the same after all.

5.2 Definitions for the Breakpoint and Call meta-concepts in Python

In the Python programming language, there is no main class or main method: The program starts to run from the beginning of the script file as is usual in other interpreted programming languages that uses script files. In the Python programming language, this top-level scope has the reserved name __*main*__ and the script program can check if it is running from this top-level scope by checking the __*name*__ global variable as presented in the source code of Figure 5.1.

Figure 5.1 – Check for the main scope in Python

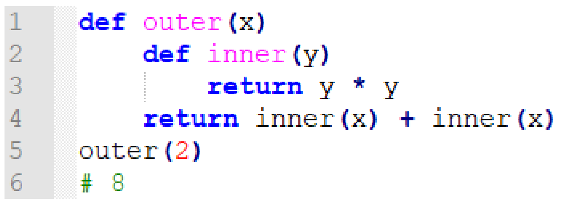


```
if __name__ == "__main__": pass
```

Every breakpoint being created in the top-level scope of a module will be defined as if being created in an object with the same name of the module based on behavior presented in Figure 5.1 that was defined by the documentation of the Python programming language.

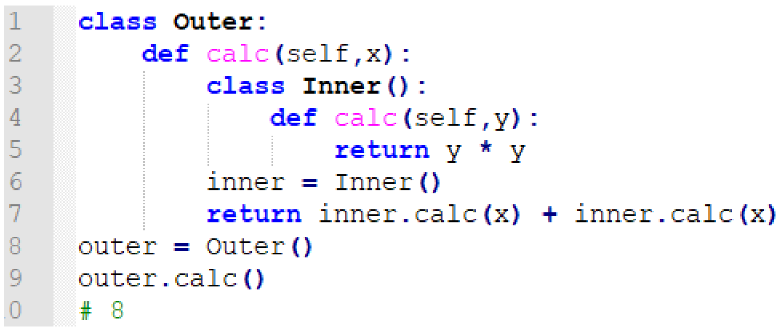
5.3 Definitions for the Object and Method meta-concepts in Python

Figure 5.2 – A function declared inside a function in Python



```
1  def outer(x)
2      def inner(y)
3          return y * y
4      return inner(x) + inner(x)
5  outer(2)
6  # 8
```

Figure 5.3 – A class declared inside the method of a class

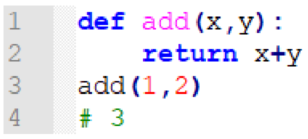


```
1  class Outer:
2      def calc(self,x):
3          class Inner():
4              def calc(self,y):
5                  return y * y
6          inner = Inner()
7          return inner.calc(x) + inner.calc(x)
8  outer = Outer()
9  outer.calc()
0  # 8
```

Despite functions in the Python programming language being very similar to functions in other object-oriented programming languages, the Python programming language offers the possibility to define methods, functions and classes inside any function or method. The source code in Figure 5.2 and Figure 5.3 provide examples of a function declared inside a function and a class declared inside the method of a class, respectively.

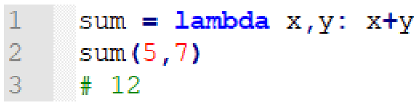
In the source code of Figure 5.2 and Figure 5.3, the objects being called from the top-level scope of a script or module are not being called by another object in lines 5 and 9, respectively. Call of objects directly from the execution scope should create calls with the caller object_id representing the module scope. Other callable types of the Python programming language are functions, lambda expressions and classes with the *__call__* special method. Figures 5.4, 5.5 and 5.6 provide an example of each type.

Figure 5.4 – Callable function in Python



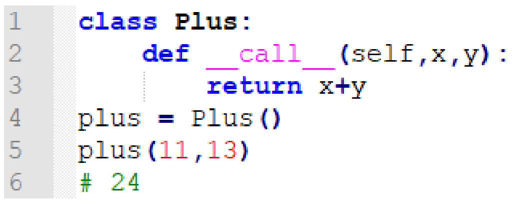
```
1  def add(x,y):
2      return x+y
3  add(1,2)
4  # 3
```

Figure 5.5 – Callable lambda expression in Python



```
1  sum = lambda x,y: x+y
2  sum(5,7)
3  # 12
```

Figure 5.6 – Callable class instance in Python



```python
class Plus:
    def __call__(self,x,y):
        return x+y
plus = Plus()
plus(11,13)
# 24
```

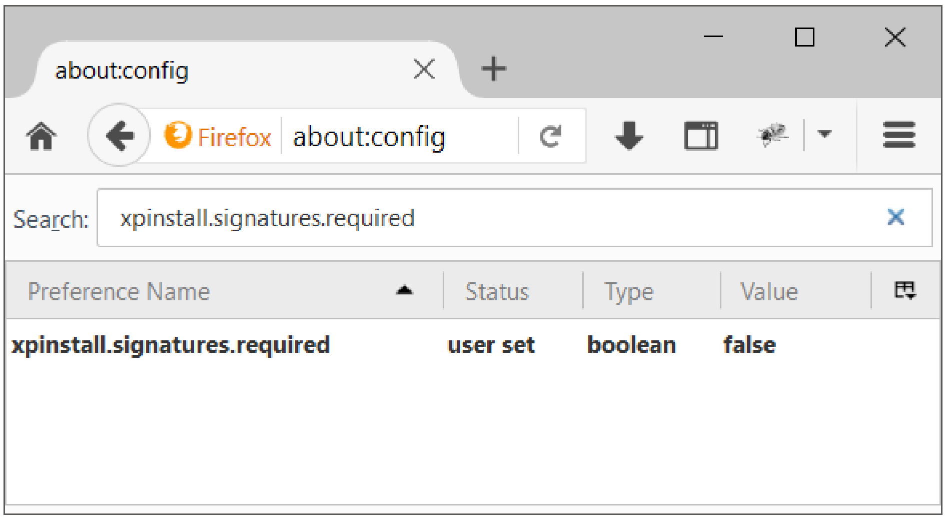
6 IMPLEMENTATION TESTS

New versions of the Swarm Debugger Tracer were made for the JavaScript and Python programming languages by changing the source code of a debugging tool for each of the two interpreted programming languages. The source code of the Firebug web browser extension was changed to support Swarm Debugging in the JavaScript programming language and the source code of the PyDev plug-in was changed to supporting Swarm Debugging in the Python programming language. Both Firebug web browser extension and PyDev plug-in provide tools to execute debugging activities in their respective interpreted programming language and both are an open source project which enabled modifications directly in their source code.

6.1 Implementing Swarm Debugger Tracer in the Firebug extension

The Firebug is a web browser extension for the Mozilla Firefox web browser that provides many tools needed in web development. Among its features, the Firebug web browser extension supports interactive debugging of the JavaScript source code that is running on the web pages. The complete setup used in the implementation test was the Firefox web browser version 56.0 and the Firebug extension version 2.0.17.

Figure 6.1 – Firefox preference changes to install custom extensions

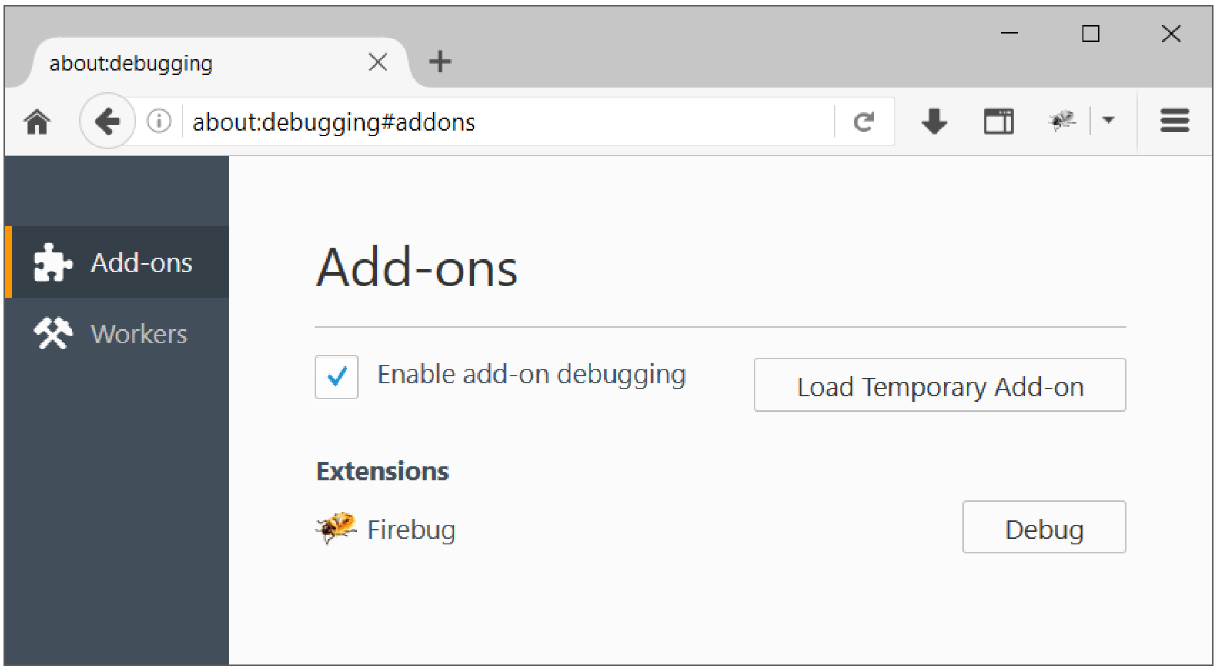


Search: xpinstall.signatures.required

Preference Name	Status	Type	Value
xpinstall.signatures.required	user set	boolean	false

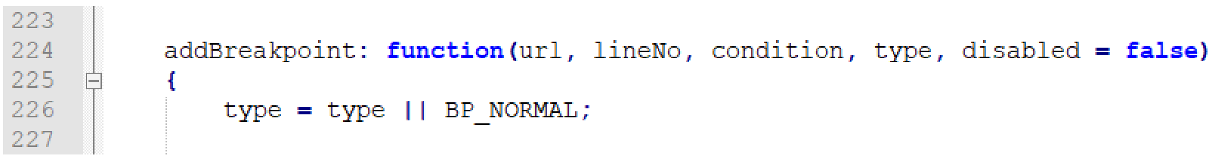
For security reasons, it is necessary to set the value of the *xpinstall.signatures.required* preference to *false* in the Firefox configuration (*about:config*) to install a custom extension as demonstrated in Figure 6.1.

Figure 6.2 – Firefox extension debugging



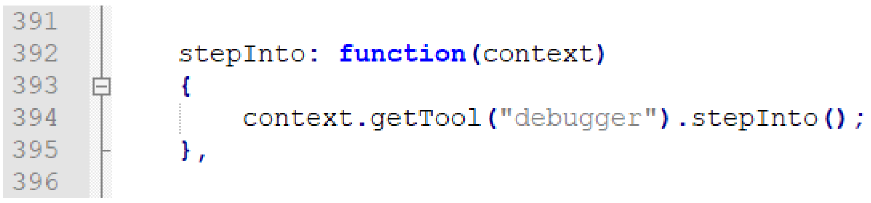
Firefox extensions are capable of being debugged as show in Figure 6.2 thus answering the question: “who debugs the debugger tool?”.

Figure 6.3 – Main function in Firebug for adding breakpoints



```
223
224     addBreakpoint: function(url, lineNo, condition, type, disabled = false)
225     {
226         type = type || BP_NORMAL;
227
```

Figure 6.4 – Main function in firebug for stepping into a function

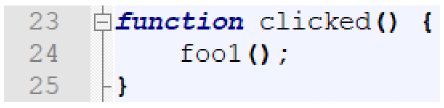


```
stepInto: function(context)
{
    context.getTool("debugger").stepInto();
},
```

The source code of the Firebug extension was analyzed and changed to watch these debugging activities, catching the developer interactions and sending them to the Swarm Debug Services. All the information about the developer interaction is sent to the Swarm Debug Services via RESTful messages during the debugging activities. The function that catches the event of adding a breakpoint in found in line 224 of */firebug/debugger/breakpoints/breakpointStore.js* file (Figure 6.3) and the function that catches the event of stepping into a function is found in line 391 of */firebug/debugger/debugger.js* file (Figure 6.4). It is then used an *XMLHttpRequest* in each function to send the developer interaction information to the Swarm Debug Services.

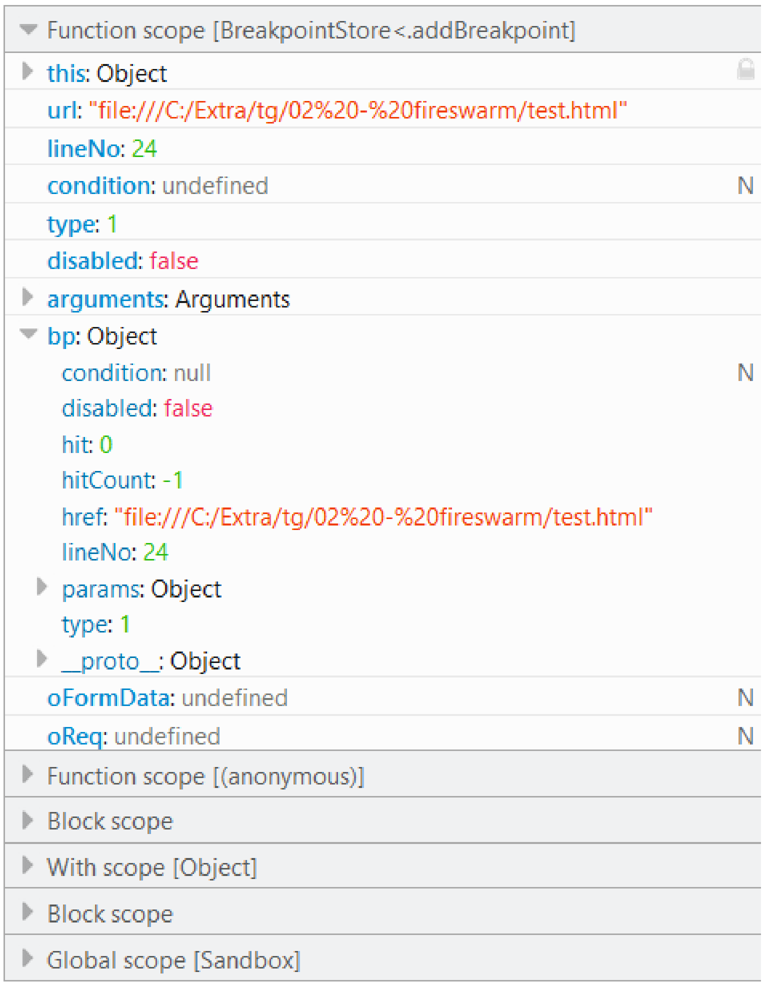
However, there is limitations to this approach. The debugger tool of the Firebug extension does not have any syntactic information about the program. When a breakpoint is set, the debugger tool does not know if the breakpoint was set inside a class method or function.

Figure 6.5 – Source code used in Firebug to test breakpoints



```
23  function clicked() {
24      foo1();
25  }
```

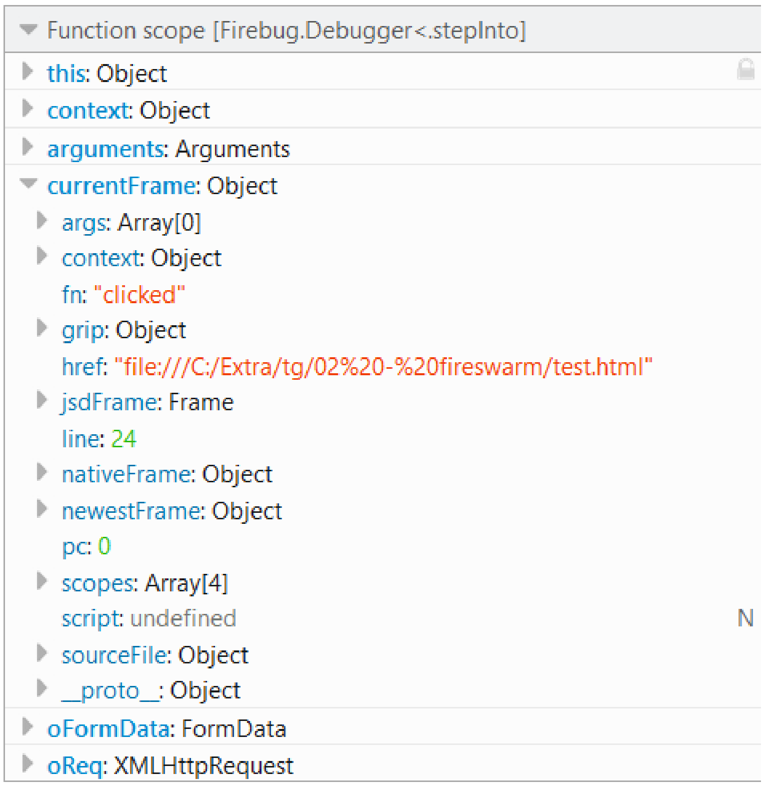
Figure 6.6 – Function scope when adding a breakpoint in Firebug



```
Function scope [BreakpointStore<.addBreakpoint]
  this: Object
  url: "file:///C:/Extra/tg/02%20-%20fireswarm/test.html"
  lineNo: 24
  condition: undefined                                    N
  type: 1
  disabled: false
  arguments: Arguments
  bp: Object
    condition: null                                       N
    disabled: false
    hit: 0
    hitCount: -1
    href: "file:///C:/Extra/tg/02%20-%20fireswarm/test.html"
    lineNo: 24
    params: Object
    type: 1
    __proto__: Object
  oFormData: undefined                                    N
  oReq: undefined                                         N
Function scope [(anonymous)]
Block scope
With scope [Object]
Block scope
Global scope [Sandbox]
```

In Figure 6.5, if a breakpoint is set on line 24 of the source code, it is not possible to know that the breakpoint is set inside a function or class method because there is no information about the syntax of the program inside the debugger tool of the Firebug web browser extension when a breakpoint is set. Figure 6.6 shows that the available information inside the debugger tool only regards to which line of which file the breakpoint was set.

Figure 6.7 – Function scope when stepping into a function in Firebug



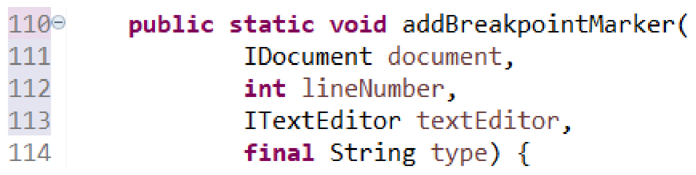
```
Function scope [Firebug.Debugger<.stepInto]
  this: Object
  context: Object
  arguments: Arguments
  currentFrame: Object
    args: Array[0]
    context: Object
    fn: "clicked"
    grip: Object
    href: "file:///C:/Extra/tg/02%20-%20fireswarm/test.html"
    jsdFrame: Frame
    line: 24
    nativeFrame: Object
    newestFrame: Object
    pc: 0
    scopes: Array[4]
    script: undefined                                          N
    sourceFile: Object
    __proto__: Object
  oFormData: FormData
  oReq: XMLHttpRequest
```

Another limitation to this approach happens during the interactive debugging of stepping into a function or class method. The step into activity does not receives information to where it is going in the source code. This happens because the debugging tool of the Firebug web browser extension and the JavaScript interpreter of the Firefox web browser are not the same thing. Only the JavaScript interpreter of the Firefox web browser knows the state of the script execution while the debugging tool just wait for when the JavaScript interpreter warns that it stopped running the code without give the reason for why exactly it stopped like when it found a breakpoint, made step into, got paused, etc.

6.2 Implementing Swarm Debugger Tracer in the PyDev plug-in

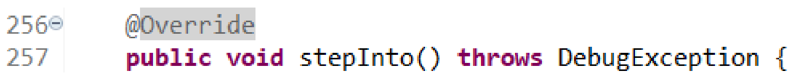
The PyDev is an open source plug-in for the Eclipse IDE to support software development in the Python programming language. The PyDev plug-in supports debugging activities in the Python programming language directly in the Eclipse IDE. The complete setup in the implementation test was the Eclipse SDK version Photon (4.8), the PyDev version 6.5.0 and the Python interpreter version 3.7.0.

Figure 6.8 – Main function in PyDev for adding breakpoints



```
110     public static void addBreakpointMarker(
111             IDocument document,
112             int lineNumber,
113             ITextEditor textEditor,
114             final String type) {
```

Figure 6.9 – Main function in PyDev for stepping into a function

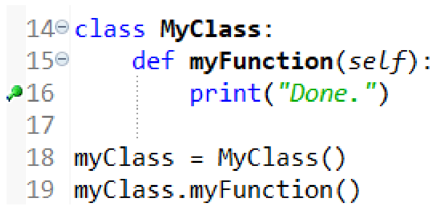


```
256     @Override
257     public void stepInto() throws DebugException {
```

The PyDev plug-in source code was changed in the in the same way as the Firebug extension source code, it watches the developer while in debugging activities and reports these activities to the Swarm Debug Services via RESTful messages during the debugging activities. The class method that catches the event of adding a breakpoint in found in line 110 of */org/python/pydev/debug/ui/actions/PyBreakpointRulerAction.java* file (Figure 6.8) and the class method that catches the event of stepping into a function is found in line 257 of */org/python/pydev/debug/model/PyStackFrame.java* file (Figure 6.9). It is then used an *HttpURLRequest* in each function to send the developer interaction information to the Swarm Debug Services.

Unfortunately, the PyDev plug-in also has the same problem of the Firebug web browser extension regarding setting breakpoints within class methods and functions. When a breakpoint is set, the debugger tool does not know if the breakpoint was set inside a class method or function.

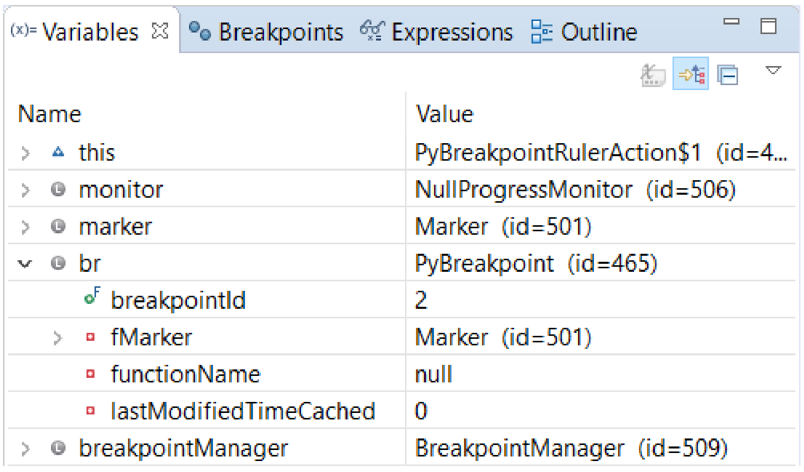
Figure 6.10 – Source code used in PyDev to test breakpoints



```
class MyClass:
    def myFunction(self):
        print("Done.")

myClass = MyClass()
myClass.myFunction()
```

Figure 6.11 – Variables when adding a breakpoint in PyDev



Name	Value
this	PyBreakpointRulerAction$1 (id=4...
monitor	NullProgressMonitor (id=506)
marker	Marker (id=501)
br	PyBreakpoint (id=465)
breakpointId	2
fMarker	Marker (id=501)
functionName	null
lastModifiedTimeCached	0
breakpointManager	BreakpointManager (id=509)

In Figure 6.10, if a breakpoint is set on line 16 of the source code, the PyDev plug-in does not have the necessary syntactic information to tell that the breakpoint is set inside a class method or function. Figure 6.11 shows that the *functionName* variable does not have any information and probably it could have some information.

Lastly, there is a difference in what is the information of the Namespace when setting a breakpoint in a line and when stepping into a function. The PyDev plug-in defines the namespace of a breakpoint as the Eclipse project followed by the Python packages then the script file of the module. Meanwhile the PyDev defines the namespace of a stepping into starting from the root of the file system all the way up to the script file.

Figure 6.12 – Python project files and folders

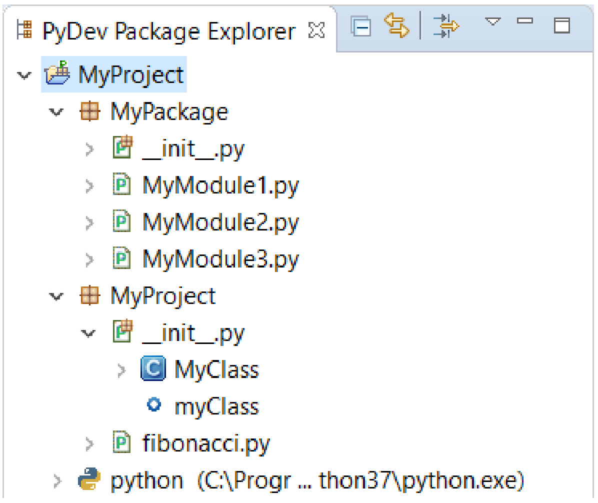
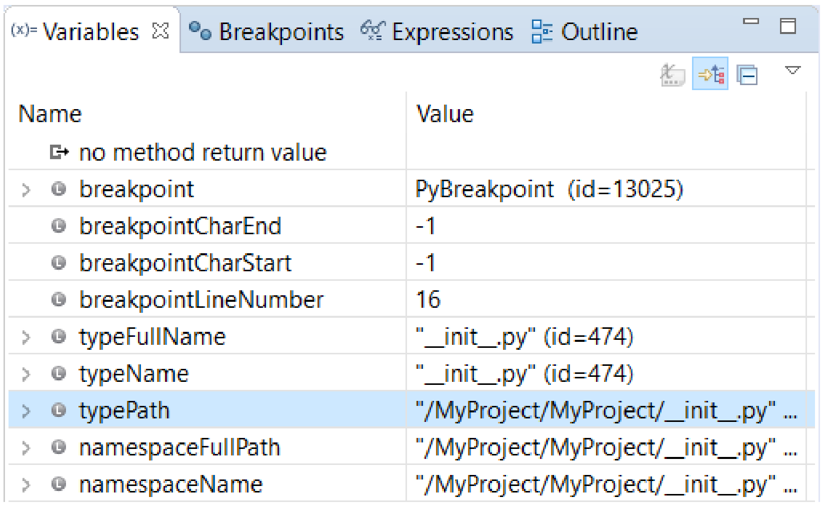
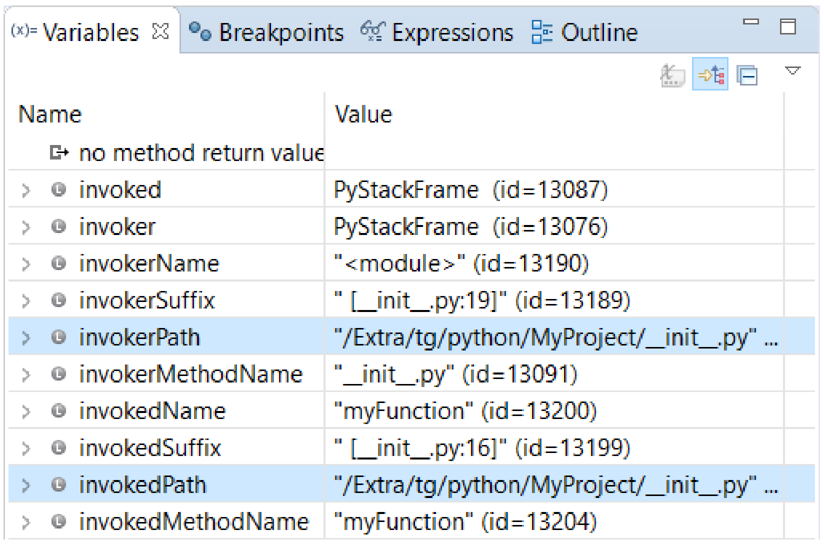


Figure 6.13 – Variables when adding a breakpoint in PyDev



Name	Value
no method return value	
breakpoint	PyBreakpoint (id=13025)
breakpointCharEnd	-1
breakpointCharStart	-1
breakpointLineNumber	16
typeFullName	"__init__.py" (id=474)
typeName	"__init__.py" (id=474)
typePath	"/MyProject/MyProject/__init__.py" ...
namespaceFullPath	"/MyProject/MyProject/__init__.py" ...
namespaceName	"/MyProject/MyProject/__init__.py" ...

Figure 6.14 – Variables when stepping into a function in PyDev



Name	Value
no method return value	
invoked	PyStackFrame (id=13087)
invoker	PyStackFrame (id=13076)
invokerName	"<module>" (id=13190)
invokerSuffix	" [__init__.py:19]" (id=13189)
invokerPath	"/Extra/tg/python/MyProject/__init__.py" ...
invokerMethodName	"__init__.py" (id=13091)
invokedName	"myFunction" (id=13200)
invokedSuffix	" [__init__.py:16]" (id=13199)
invokedPath	"/Extra/tg/python/MyProject/__init__.py" ...
invokedMethodName	"myFunction" (id=13204)

The Figure 6.12 presents the folders and files for an example project. The *MyProject* project is set to the *C:/Extra/tg/python* folder and inside the folder there is two packages: *MyPackage* and *MyProject*. In Figure 6.13, the namespace for a breakpoint set inside the *__init__.py* module inside the *MyProject* package is very different from the namespace for a stepping into made in a function call in the same script file as shows in 6.14.

7 CONCLUSION

Debugging activities are essential and make a large portion of the time spent in software development. The Swarm Debugging approach to software development provides a way to keep the knowledge gathered during debugging sessions for future developers who will work in the same source code. Analyzing the documentation of the JavaScript and Python programming languages, it seems possible to support the Swarm Debugging in these interpreted programming languages.

7.1 Summary of contributions

The key meta-concepts of the Swarm Debugging were mapped to features of interpreted programming languages, however it was necessary to define beforehand the behavior of some edge cases where the Java programming language and interpreted programming languages differs.

Two versions of the Swarm Debug Tracer are developed for use with the JavaScript and Python programming languages on top of the Firebug extension and the PyDev plug-in respectively.

7.2 Limitations of this work

Despite being theoretically possible to support Swarm Debugging in interpreted programming languages, the viability of this endeavor is compromised by a lack of lexical and syntactic information about the program during debugging activity in the analyzed debugging tools. As such, it was not possible to test the effectiveness of Swarm Debugging with end users in the native environment of interpreted programming languages.

7.3 Future works

There are multiple paths open for future work. One is to improve the debugging tools used to bring lexical and syntactic information into the target development environment, so it can be used in the Swarm Debug Tracer and other tools. Another path in future works is to analyze other debugging tools to see if some of them has all the necessary lexical and syntactic information necessary to support the development of a Swarm Debug Tracer in them. Lastly, there is another programming paradigms yet to support Swarm Debugging as the functional programming paradigm with programming languages such Lisp and Haskell.

REFERENCES

BROOKS, F. P. No silver bullet: Essence and accidents of software engineering. **IEEE Computer**, v. 20, n. 4, p. 10–19, April 1987.

HAILPERN, Brent; SANTHANAM, Padmanabhan. Software debugging, testing, and verification. **IBM Systems Journal**, v. 41, n. 1, p. 4-12, 2002.

PETRILLO, Fabio et al. Swarm debugging: towards a shared debugging knowledge. **III Workshop on Software Visualization, Evolution, and Maintenance (VEM)**, 2015.

PETRILLO, F. **Swarm debugging: the collective debugging intelligence of the crowd**. 2016. 125 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2016.

O'REILLY, TIM. **What is web 2.0**: Design Patterns and Business Models for the Next Generation of Software. 2018. Available in: <https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html>. Accessed in: 20 ago. 2018.

OUSTERHOUT, John K. Scripting: Higher level programming for the 21st century. **Computer**, v. 31, n. 3, p. 23-30, 1998.

CASS, Stephen. Interactive: The top programming languages 2018. **IEEE Spectrum**, 2018. Available in: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018>. Accessed in: 20 Nov. 2018

ZELLE, John M. Python as a first language. In: **Proceedings of 13th Annual Midwest Computer Conference**. 1999. p. 145.

TIOBE SOFTWARE BV. **TIOBE Programming Community index**, 2018. Available in: <https://www.tiobe.com/tiobe-index/>. Accessed in: 20 ago. 2018.

CARBONNELLE, Pierre. **PYPL PopularitY of Programming Language**, 2018. Available in: < http://pypl.github.io/PYPL.html>. Accessed in: 20 ago. 2018.

PETRILLO, Fabio et al. Towards understanding interactive debugging. In: **Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on**. IEEE, 2016. p. 152-163.

FLEMING, Scott D. et al. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 22, n. 2, p. 14, 2013.

ECMA INTERNATIONAL. **Standard ECMA-262**: ECMAScript 2015 Language Specification, 9th ed. [S.l.:s.n], 2018.

MOZILLA DEVELOPER NETWORK. **JavaScript Reference**, 2018. Available in: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/>. Accessed in: 20 ago. 2018.

WIKIMEDIA FOUNDATION. **Web browser data usage**, 2018. Available in: <https://analytics.wikimedia.org/dashboards/browsers/#all-sites-by-browser>. Accessed in: 20 ago. 2018.

VAN ROSSUM, Guido; DRAKE, Fred L. **Python language reference manual**. United Kingdom: Network Theory, 2003.