Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Analysing the changes of software metric values with RefactorErl

Supervisor:
Melinda Tóth
Assistant lecturer

Author:
Marina Konoreva
Computer Science MSc
2. year

Budapest, 2019

Abstract

Nowadays, the requirements for development speed and software quality are significantly increasing. The use of a flexible architecture and various design techniques certainly can improve the quality of development, but formal quality criteria, such as code metrics, showing the quantitative characteristics of a software system in various dimensions, still remain relevant. In evaluating the duration and complexity of software development, various metrics are used. Software metrics and their visualization are two important features of measurement systems.

The goal of my thesis work is to analyse open source software with RefactorErl,log the changes of metric value, visualize it and conclude the findings.

Contents

Chapter 1

Introduction

Requirements for the quality of the product being developed have rapidly increased in recent years. From the beginning of software product development, developers have been striving to monitor quality. Software metrics and their visualization are two important features of measurement systems.

In this thesis, we introduce framework for analysing of quality of programs written in the Erlang programing language which built on the top of the RefactorErl static analysis tool. Our goal is to analyse projects and then prepare the results of the measurements.

Erlang is a functional language designed for highly parallel, scalable applications requiring high uptime. Several industrial and open source products were implemented in Erlang, therefore tools to measure the complexity, quality of the source code are highly desirable.

RefactorErl is a static source code analysis and transformation tool for Erlang providing several software metrics. The tool is developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. Among the features of RefactorErl is included a metric query language which can support Erlang developers in everyday tasks such as program comprehension, debugging, finding relationships among program parts, etc. Software metrics provide a means to extract useful and measurable information about the structure of a software system. The results of these evaluation methods can be used to indicate which parts of a software system need to be re-engineered.

This thesis is organized as follows: Chapter 2 is needed for understanding the main concepts of software metrics. Chapter 3 covers the necessary background on Erlang programming language, RefactorErl tool, defined metrics in this tool and description developed framework. Chapter 4 illustrates measurements and findings of some projects. Chapter 5 describes some related works. Chapter 6 consists conclusion about completed work.

Chapter 2

Software metrics

This chapter presents the introduction to the software metrics as the important tool for evaluation of quality and productivity of the software development product. Also the classification of the software metrics, description the different measurements of software metrics in general and the most popular software metrics tools are included in this chapter.

2.1 Definition of Software Metrics

Software metrics are the attributes of the software systems that deals with the measurements of the software product and process by which it is developed [1].

A software metric is a measure of characteristics of software which are countable or quantifiable. The importance of software metrics is valuable for many reasons, including planning work items, measuring software performance, measuring productivity, and many other uses.

Software developers must recognize the principles of software metrics that involve cost,schedule find quality goals, quantitative goals, comparison of plans with actual performance throughout development, monitoring data trends for indication of likely problems, metrics presentation, and investigation of data values.

There are some metrics within the process of software development, that are all related to each other. Metrics can be related to the four functions of management:

- Controlling.
- Planning.
- Improving.
- Organising.

The aim of analyzing and tracking software metrics is to find out the quality of the particular product or process, enhance its quality and forecast the quality once the software development project is done. On a more detailed level, software development managers try to:

- Manage workloads.
- Increase return on investment (ROI).
- Reduce overtime.
- Identify areas of improvement.
- Reduce costs.

These goals can be polished by providing information and clarity overall the organization about complex software development projects. Metrics are an essential value of quality assurance, performance, management, debugging, and estimating costs, and they are valuable for both development team leaders and developers:

- Teams of software developers can use software metrics to interact between the status of software development projects, pinpoint and address issues, and observe, improve on, and manage their workflow better.
- Managers of software can use software metrics to track, identify, communicate and prioritize any issues to foster better team productivity. This permits effective management and allows prioritization and assessment of problems within software development projects. The sooner managers can find software problems, the easier and less-expensive the process of troubleshooting.

Software metrics provide an assessment of the impact of decisions made through the process of development of software projects. This helps managers prioritize and assess performance goals and objectives.

2.2 Classification of software metrics

Software metrics are broadly classified as product metrics and process metrics as shown in Figure 2.1 [2].

Process metrics are numerical values that depict a software process such as the amount of time require to debug a module [2]. They are measures of the software development process, such as: overall development time and type of methodology used. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to longterm software process improvement.

Product metrics can be used to analyze and check the software project. These type of metrics calculates the complexity of the software design size of the final program number of pages of documentation produced. They allow a software project manager to do the following:

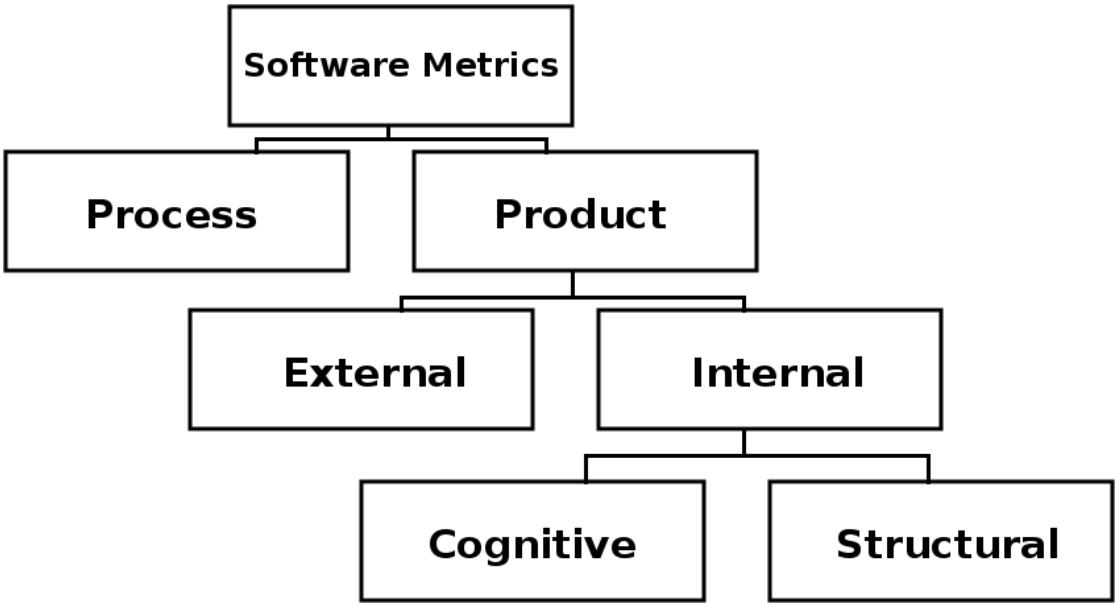


Figure 2.1: Classification of software metrics.

- decrease the development time by making the necessary modifications to avoid delays and potential problems and risks.
- project quality assessment and change the technical approach to get better quality.

Product metrics are measures of the software product of any stage of its development,from requirements to installed system [2]. Product metrics may measure: the complexity of the software design, the size of the final program, the number of pages of documentation produced. Product metrics can be internal or external. External attributes of an entity can be measured only with respect to how the entity relates with the environment and therefore can be measured only indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine and user. Productivity, an external attribute of a person, clearly depends on many factors such as the kind of process and the quality of the software delivered. Internal product metrics can be measured only based on the entity and therefore the measures are direct. For example, size is an internal attribute of any software document.

Internal product metrics are subdivided in two categories: cognitive complexity metrics and structural complexity metrics. Cognitive complexity metrics measure the effort required by developers to understand a system. They aim at discovering the cause of the complexity, which requires understanding human mental processes and details of the software system under development. Structural complexity metrics use the interactions within and among modules to measure a system's complexity. One of the oldest and most commonly used structural complexity metrics is the number of source lines of code.

Another classification of software metrics is as follows:

1. Objective metrics.
2. Subjective metrics

Objective metrics always results in identical values for a given metric as measured by two or more qualified observers. Whereas subjective metrics are those that even qualified observers may measure different values for a given metric since their subjective judgment is involved in arriving at the measured value.

2.3 Types Of Software Metrics

It is now apparent that software metrics are important in software engineering. Symons stated that "a reliable and credible method for measuring the software development cycle is needed that has a reasonable theoretica1 basis and that produces results that practitioners can trust" [3]. Hence, software metrics were used to measure a wide range of software developing activities.

2.3.1 Size-Oriented metrics

Size-oriented metrics are used to analyze the quality of software.

Lines of Code (LOC)

Lines Of Code (also possible SLOC - Source Lines of Code) is a metric generally used to measure a software program or codebase according to its size. It represents how many lines of source code exist in the application, class, methord or namespace. LoC can be used for: checking the size of code units and estimating the size of project. LOC is the simplest one but very popular. There are two major types of SLOC measures:

Physical SLOC (LOC) Physical SLOC is a number of lines in the source code of the program including comment.

Logical SLOC (LLOC) Logical LOC tries to calculate the number of "statements", but their specific definitions are tied to specific computer languages.

Physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical LOC is less sensitive to formatting and style conventions. Unfortunately, SLOC measures are often stated without giving their definition, and logical LOC can often be significantly different from physical SLOC.

2.3.2 Object-oriented metrics

Chidamber and Kemerer have specified several metrics for object-oriented designs [4]. All of these metrics are referred not to the whole system, but the separate class.

Number Of Methods (NOM)

The Number Of Methods metric is used to measure the average count of all class operations per class. A class must have some, but not an excessive number of operations. This information is useful when identifying a lack of primitiveness in class operations (inhibiting re-use), and in classes which are little more than data types.

Number Of Children (NOC)

NOC metric measures the number of subclasses which belong to a class. This metric calculates the class hierarchy.

NOC metric is closely relevant with DIT metric, which is more better because it supports for reusing methods through inheritance. The first metric calculates the number of child classes, DIT measures the depth of the class. Inheritance levels can be used to gain the depth and reduce the breadth.

A high number of NOC means the following:

- Wrong abstraction of the parent class.
- The reuse of the base class is high. The form of reuse is inheritance.
- Wrong using of subclasses. In this case, it is needed to introduce a new level of inheritance with newly grouped related classes.
- Base class needed to be more tested.

Classes high hierarchy have more subclasses then classes with low hierarchy. The number of children gives an idea of the potential influence a class has on the design [4].

Weighted Methods per Class (WMC)

This metric calculates a sum of complexities all defined methods in a class. WMC shows the complexity of whole class. This measure helps to indicate the development and maintenance effort for the class. Classes with a large number of WMC can often be refactored into several classes.

A class with a high value of WMC and a high number of NOC indicates complexity at the top of the class hierarchy. A sign of poor design is the potential impact of the base class on a large number of subclasses.

Coupling Between Object classes (CBO)

CBO classes metric demonstrates the number of classes coupled to a given class. This coupling can happen through:

- Properties or parameters.

- Method call.
- Method arguments or return types.
- Class extends.
- Variables in methods

Coupling among classes is crucial for a system to do useful work, but redundant coupling makes the system more complicated to maintain and reuse. At the project or package level, this metric displays the average number of classes used per class. A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be [4].

Depth of Inheritance Tree (DIT)

Depth of Inheritance Tree (DIT) measures the maximum length of a path from the current class to the root class in the inheritance structure of a system. DIT calculates how many super-classes can affect a class. DIT is applicable only to object-oriented systems.

If a class is on the deep level in the hierarchy, the more methods and variables it tends to inherit, what makes it more complex. Deep trees indicate big complexity of the design. Inheritance is a key for complexity managing, really, not for its increasing. As a positive factor, deep trees promote reuse because of method inheritance. Deeper trees constitute greater design complexity, since more methods and classes are involved [4].

C&K suggested the following consequences based on the depth of inheritance:

- Deeper trees establish large design complexity, since more classes and methods are involved
- If a class is on the deep level in the hierarchy, the more methods and variables it tends to inherit, what makes it more complex to foresee its behavior
- If a particular class is on the deep level in the hierarchy, there is a great chance of the possible reuse of inherited methods

Response For a Class (RFC)

The Response for Class (RFC) metric measures the total number of methods that can probably be executed as a response to a message received by some object of a class. This number calculates as the sum of the methods of the class, and all distinct methods are called directly within the class methods. Additionally, it counts inherited methods, but not overridden methods, because only one method of a particular signature will always be accessible for an object of a given class.

A large RFC is an indicator of more faults. Classes that have a high RFC are more complex and more difficult to understand. Testing and debugging for these classes is also complicated. A worst-case value for possible responses will assist in the appropriate allocation of testing time. If a large number of methods can be invoked on response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the pan of the tester [4].

2.3.3 Complexity metrics

Complexity is an important aspect for software quality assessment and must be appropriately addressed in service-oriented architecture [5]. One of the key aims of complexity metrics is to predict modules that are fault-prone post-release [6]. These metrics are one of the most difficult software metrics for understanding.

McCabe's Cyclomatic Complexity (MVG)

To determine the complexity of a software, McCabe suggests a "mathematical technique that will provide a quantitative basis for modularisation and allow us to identifY software modules that will be difficult to test or maintain" [7]. McCabe's cyclomatic complexity [8] is a software quality metric that shows the complexity of a software program. Complexity is inferred by summarizing the number of linearly independent paths through the program. The higher the number the more complex the code.

A pragmatic approximation to this can be found by counting language keywords and operators which introduce extra decision outcomes.

2.3.4 Structural Metrics

Fan-In and Fan-Out metrics (FIN and FOUT)

It is a structural metrics which measures inter-module complexities.

Fan-out Is the number of modules that are called by a given module.

Fan-in Is the number of modules that call a given module.

Fan-out and fan-in metrics reflect structure dependency [9]. These structural metrics were first defined by Henry. These metrics can be applied both at the module level and function level. These metrics just put a number on how complex is interlinking of different modules or functions. Unlike Cyclomatic complexity, you cannot put a number and say it cannot go beyond this number. This is used just to size up how difficult it will be to replace a function or module in your application and how changes to a function or module can impact other functions or modules. Sometimes you can put the restriction on the number of Fan-Out.

2.3.5 Cohesion metrics

Cohesion is an important software quality attribute and high cohesion is one of the characteristics of well-structured software design [10]. Cohesion metrics analyze the connection between the methods of a class. Module cohesion indicates relatedness in the functionality of a software module [11].

Lack of Cohesion in Methods (LCOM)

LCOM calculates the number of cohesiveness present, how well a system was designed and how complex a class is. LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero. LCOM is probably the most controversial and argued over of the C&K metrics.

C&K's rationale for the LCOM method was as follows:

- Lack of cohesion implies classes should probably be split into two or more subclasses.
- The cohesiveness of methods within a class is desirable since it promotes encapsulation.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.
- Any measure of disparateness of methods helps identify flaws in the design of classes.

Although there is a fair amount of debate about how to calculate LCOM and it features in a lot of metrics sets an increasing number of researchers to suggest that it is not a particularly useful metric. Perhaps this is also reflected in there being a fair amount of debate about how to calculate LCOM but very little on how to interpret it and how it fits in with other metrics.

Tight and Loose Class Cohesion (TCC and LCC)

TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) metrics measure the relative number of directly-connected pairs of methods and the relative number of directly- or indirectly-connected pairs of methods. The Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a low cohesion indicate errors in the design.

TCC considers two methods to be connected if they share the use of at least one attribute. A method uses an attribute if the attribute appears in the method's body or the method invokes directly or indirectly another method that has the attribute in its body. The higher TCC and LCC, the more cohesive and thus better the class.

In this Section, we focused our attention on all possible software metrics. In the next Chapter, we will define metrics which are used to describe various aspects of Erlang projects.

2.4 Software Metric Tools

There are a lot of software metrics have been developed and numerous tools exist to gather the metrics from program representations. This large number of tools allows a user to choose the tool best suited for user requirements, for example, its handling, tool support, cost etc. This is accepted that the metrics computed by the metric tools are the same for all the metric tools. One can think of a software metric tool as a program which implements a set of software metrics

definitions. It allows accessing a software system according to the metrics by extracting the required entities from the software and providing the corresponding metric values. There are some criteria for selecting the proper metric tools as the availability of the software tools can make confusion. One such criterion is that the tools must have to calculate any form of software metrics. Majority metric tools are available for Java programs. Many tools are just code counting tool, they basically count the variants of the lines of code (LOC) metric. The specific criteria areas follow language: Java (source or bytecode), metrics: well-known object-oriented metrics on class level, license: freely available.

2.4.1 CCCC

CCCC is a small command-line tool which generates metrics from C or C++ project's source code [12]. The tool outputs a simple HTML website with information about all your sources. It generates reports on different metrics, for example, lines of code (LOC) and metrics suggested by Chidamber&Kemererand Henry&Kafura. CCCC is distributed as a freeware and is released in the form of source code. Users have to compile the program on their own and to modify the source code to reflect their interests and preferences.

CCCC can process every file which was provided via the command line. It is possible to use standard wildcard process. For every file, CCCC will check the filename's extension, and if the extension is recognized as one from supported languages, the particular parser will parse this file. As every file is parsed, certain constructs recognition will cause records to be inserted into an internal database. After all files have been processed, the software will generate the report in HTML format. This report is depended on on the contents of the internal database. By default settings the main HTML report is produced to the file cccc.htm in a subfolder called .cccc of the current working folder. It includes detailed reports on every module (for example, C++ or Java class) detected by the analysis run.

As an addition to HTML reports and summary, the run of the program will generate the corresponding summary and reports in XML format. Also, the file called cccc.db will be created. This file will represent a dump of the internal database of the software in a special format with '@' symbol as a delimiter. It is chosen because this symbol is one of the few which can not appear in C/C++ source code.

The report consists of a number of tables which identify the modules in the submitted files and covering:

1. Measures the number and the relationships type.
2. Measures the procedural volume and complexity and functions of every module.
3. A summary report over the whole codebase processed of the measures described above.
4. Identification of any parts of the source code submitted which the program can not parse.

This tool can measure the following metrics:

- Fan-In Fan-Out (FIN and FOUT).
- Lines of Code (LOC).
- Number Of Children (NOC).
- Weighted Methods per Class (WMC).
- McCabe's Cyclomatic Complexity (MVG).
- Number Of Methods (NOM).

2.4.2 Chidamber&Kemerer

The program counts Chidamber and Kemerer object-oriented metrics by introspection the byte-code of compiled Java files [13]. It is an open source command line tool. The program counts the following six metrics for each class, and displays them on its standard output, following the class's name.

This tool can measure the following metrics:

- Depth of Inheritance Tree (DIT).
- Weighted Methods per Class (WMC).
- Numbe r Of Children (NOC).
- Coupling Between Object classes (CBO).
- Lack of Cohesion in Methods (LCOM).
- Response For a Class (RFC).

2.4.3 Analyst4j

Analyst4j is built on the Eclipse platform and can be downloaded as a standalone Rich Client Application or also as an Eclipse IDE plugin [14]. Its features are search, metrics analyzing, quality analyzing, report generating for Java programming. Analyst4j software is most popular to find out the quality-related metrics. This tool is based on Chidamber&Kemerer metrics.

This tool can measure the following metrics:

- Weighted Methods per Class (WMC).
- Lines of Code (LOC).
- Coupling Between Object classes (CBO).
- Depth of Inheritance Tree (DIT).

- Response For a Class (RFC).
- Number Of Children (NOC).
- Lack of Cohesion in Methods (LCOM).
- Number Of Methods (NOM).

2.4.4 OOMeter

OOMeter is a software metric tool for measuring the quality attributes of Java and C# source code and UML models, stored in XMI format [15]. OOMeter has a rich collection of object-oriented software metrics. This is the Eclipse plugin. It provides a querying language for object-oriented code similar to SQL which allows to search for measure code metrics, bugs etc.

OOMeter provides an interface for users to define custom metrics through Java classes that implement a certain interface. It supports export of metric results to a number of formats, including XML, HTML, delimited text, Microsoft Excel, etc [16].

This tool can measure the following metrics:

- Weighted Methods per Class (WMC).
- Lines of Code (LOC).
- Coupling Between Object classes (CBO).
- Depth of Inheritance Tree (DIT).
- Response For a Class (RFC).
- Numbe r Of Children (NOC).
- Tight Class Cohesion (TCC).
- Lack of Cohesion in Methods (LCOM).

2.4.5 Eclipse Metrics plugin 1.3.6

This is an open source dependency analyzer and metrics calculation plugin for Eclipse IDE [17]. The plugin is also provided integrated as an EasyEclipse package. The plugin computes the various metrics and displays it in the integrated view.

This tool can measure the following metrics:

- Weighted Methods per Class (WMC).
- Lines of Code (LOC).

- Numbe r Of Children (NOC).
- Depth of Inheritance Tree (DIT).
- Number Of Methods (NOM).

2.4.6 Eclipse Metrics plugin 3.4

The eclipse plugin 3.4 developed by Lance Walton is also integrated with Eclipse and is available for all Java projects developed using the IDE [18]. It is an open source tool. It counts various metrics in the moment of build cycles and shows warnings via the problem view of metrics range violations.

This tool can measure the following metrics:

- Weighted Methods per Class (WMC).
- Lines of Code (LOC).
- Lack of Cohesion in Methods (LCOM).
- Depth of Inheritance Tree (DIT).

2.4.7 Semmle

Semmle is the platform for analyzing that produces a detailed report of the code base for one or more software projects [19]. For every project that it analyzes, it calculates artifacts against rules that check for good practice. Analysis can be scheduled to run on a regular basis. The copy of the source code is checked out from the repository for analysis as part of this process. The code, and related artifacts is checked against rules, defined using queries, to identify any alerts. Finally, metrics are calculated and data can be imported from third-party systems used by your company. A database is created, containing detailed information about the artifacts and every alert.

This tool can measure the following metrics:

- Lack of Cohesion in Methods (LCOM).
- Depth of Inheritance Tree (DIT).
- Number Of Methods (NOM).
- Numbe r Of Children (NOC).
- Response For a Class (RFC).

2.5 Measuring functional languages

In the previous section has been described software metrics for object-oriented languages. However, software metrics developed for imperative and object-oriented languages can also be used for measuring in functional programming languages like Erlang and Haskell.

Some of the measurement techniques from imperative and object-oriented languages may transfer quite cleanly to functional languages, for instance the path count metric which counts the number of execution paths through a piece of program code, but some of the more advanced features of functional programming languages may contribute to the complexity of a program in ways that are not considered by traditional imperative or object oriented metrics [20].

We can use the same metrics because several constructs as a class, a module, and a library are similar. All of this structures can be consider like collections of functions. If the chosen metric does not take the distinctive properties of these constructs into account (variables, method overrides, dynamic binding, visibility etc.), then it can be applied to these apparently diverse constructs [21].

The dissimilarity between functional and imperative languages are in the difference in the level of nesting of blocks and control structures, in several ways of connecting certain functions (for example, data flow and call graph), inheritance instead of cohesion and simple cardinality metrics(lines of code, char of code).

Another difference functional programming languages from imperative languages is there are some constructs and properties that can be used only in functional programming languages as list comprehensions, pattern matching, referential transparency of pure functions, currying, laziness of expression evaluation.

While these features raise the expressive power of functional languages, most of the existing complexity metrics require some changes before they become applicable to functional languages [21].

There are general metrics are acceptable for functional languages:

- **Branches of recursion**. This metric allows measuring how many times did the function call itself
- **Fun expressions and message passing constructs**.
- **Return points of a function**.
- There is possible to calculate metrics on a single clause of a function.
- There is possible to calculate metrics on a single clause of a function.
- **Otp used**. This metric allows measure OTP behaviors.

In the next chapter will be described all developed metrics for Erlang in details.

Chapter 3

Software metrics in Erlang

In this chapter, we present a developed framework which allows to analyze Erlang programs and visualize calculated metrics. The first two sections give a brief summary of Erlang, RefactorErl static analysis tool and developed metrics.

3.1 An introductory glimpse at the Erlang programming language

Erlang is a functional programming language which provides runtime designed for highly parallel, scalable software requiring high uptime. It is designed for development of robust systems that can be distributed between many different computers in a network. Erlang is kinda of similar to Java in the case that it uses a virtual machine and also supports multithreading.

Variables Erlang provides dynamic data types, allowing programmers to develop system components (such as message dispatchers) that do not care what type of data they are handling and others that strongly enforce data type restrictions or that decide how to act based on the type of data they receive. Variables must start with a capital letter or an underscore, and are composed of letters, digits, and underscores.

Data types Erlang has:

1. Integers of unlimited size.
2. Floats.
3. Strings, placed within double quotes: "It is some string.".
4. Atoms. An atom is an element by itself. It starts with a lowercase letter and is built of letters, digits, and underscores, or it is any string placed within single quotes: atom1, 'Atom 2'.

5. Lists are a comma-separated sequence of some values placed within brackets: [abc, 123, "It is some string"].
6. Tuples are a comma-separated sequence of some values placed within braces: abc, 123, "It is some string".
7. Records are not a separate data type but are just tuples with keys associated with each value. They are declared in a file and defined (given specific values) in the program.
8. Binaries are placed within double angle brackets: «0, 128, 128, 255», «"It is some string"», «X:7, Y:5, Z:1». Binaries are series of bits; the number of bits in a binary has to be a multiple of 8.
9. References are globally unique values.
10. Pids stand for process identifiers which are the "names" of processes.

Figure 3.1 features the source of a small Erlang program called example that demonstrated recursive list manipulation.

```
1   -module(example).
2   -export([max/1, min/1, sum/1]).
3
4   %% Find the maximum of a list.
5   max([H|T]) -> max2(T, H).
6   max2([], Max) -> Max;
7   max2([H|T], Max) when H > Max -> max2(T, H);
8   max2([_|T], Max) -> max2(T, Max).
9
10
11  %% Find the minimum of a list.
12  min([H|T]) -> min2(T,H).
13  min2([], Min) -> Min;
14  min2([H|T], Min) when H < Min -> min2(T,H);
15  min2([_|T], Min) -> min2(T, Min).
16
17  %% Find the sum of all the elements of a list.
18  sum(L) -> sum(L,0).
19  sum([], Sum) -> Sum;
20  sum([H|T], Sum) -> sum(T, H+Sum).
```

Figure 3.1: A simple module in Erlang.

Erlang source files consist of a section containing meta-information about the module represented by the file (all functions in Erlang must be defined in modules.), and a list of functions that are either exposed to the users of this module (with the -export attribute), or are only defined for internal use inside the module.

A subtle element of all three functions is that every function needs to have an initial value to start counting with. In the case of `sum/2`, we use 0, as we're doing addition, and given `X = X + 0`, the value is neutral, so we can't mess up the calculation by starting there. If we were doing multiplication, we would use 1 given `X = X * 1`.

The functions `min/1` and `max/1` can't have a default starting value. If the list were only negative numbers and we started at 0, the answer would be wrong. So we need to use the first element of the list as a starting point.

3.2 The RefactorErl static analysis tool

RefactorErl [22, 23] is an open-source static source code analyzer and transformer tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. The phrase "refactoring" means a preserving source code transformation, so while you change the program structure you do not alter its behavior. RefactorErl was built to refactor Erlang programs.

The main focus of RefactorErl is to support daily code comprehension tasks of Erlang developers [24]. It can analyze the structure of the refactored program - based on the syntactic rules of the underlying programming language - and it can also collect and use semantical information about the source code.

3.2.1 Metrics in the RefactorErl

A metric query language is incorporated into RefactorErl [24]. Metric queries can be executed from the console interface or can be used as properties in semantic query language which is available from every interface.

Table 3.1 shows all the implemented metrics in RefactorErl tool. There are two columns in this table: the first column gives the information about the name of the metric, and the second column shows the for which node the metric is available.

module_sum

The sum of the chosen complexity structure metrics measured on the functions of the module. The proper metrics adjusted in a list can be implemented in the desired number and order [25].

line_of_code

The number of lines of part of the text, function, or module. The number of empty lines is not included in the sum. As the number of lines can be measured on more functions, or modules and the system is capable of returning the sum of these, the number of lines of the whole loaded program text can be enquired [25].

char_of_code

The number of characters in a program script. This metric is capable of measuring both the codes of functions and modules and with the help of aggregating functions we can enquire the total and the average number of characters in a cluster, or in the whole source text [25].

Table 3.1: Implemented metrics in RefactorErl

Name of the metric	Node type
module sum	module
line of code	module/function
char of code	module/function
number of fun	module
number of macros	module
number of records	module
included files	module
imported modules	module
number of funpath	module
function calls in	module
function calls out	module
cohesion	module
function sum	function
max depth of calling	module/function
max depth of cases	module/function
min depth of cases	module/function
max depth of structs	module/function
number of funclauses	module/function
branches of recursion	module/function
calls for function	function
calls from function	function
number of funexpr	module/function
number of messpass	module/function
fun return points	module/function
average size	module/function
max length of line	module/function
no space after comma	module/function
is tail recursive	function
mcCabe	module/function
otp used	module

number_of_fun

This metric gives the number of functions implemented in the concrete module, but it does not contain the number of non-defined functions in the module [25].

number_of_macros

This metric gives the number of defined macros in the concrete module or modules. It is also possible to inquire the number of implemented macros in a module [25].

number_of_records

This metric gives the number of defined records in a module. It is also possible to inquire the

number of implemented records in a module [25].

included_files

This metric gives the number of visible header files in a module [25].

imported_modules

This metric gives the number of imported modules used in a concrete module. The metric does not contain the number of qualified calls (calls that have the following form: module:function) [25].

number_of_funpath

The total number of function paths in a module. The metric, besides the number of internal function links, also contains the number of external paths or the number of paths that lead outward from the module. It is very similar to the metric called cohesion [25].

function_calls_in

Gives the number of function calls into a module from other modules. It can not be implemented to measure a concrete function. For that, we use the calls_for/1 function [25].

function_calls_out

Gives the number of every function call from a module towards other modules. It can not be implemented to measure a concrete function. For that, we use the calls_from/1 function [25].

cohesion

The number of call-paths of functions that call each other. By call-path we mean that an f1 function calls f2 (e.g. f1()->f2().). If f2 also calls f1, then the two calls still count as one call-path [25].

function_sum

The sum calculated from the functions complexity metrics characterizes the complexity of the function. It can be calculated using various metrics together [25].

max_depth_of_calling

The length of function call-chains, namely the chain with the maximum depth [25].

max_depth_of_cases

Gives the maximum of case control structures embedded in case of a concrete function (how deeply are the case control structures embedded). In case of a module, it measures the same regarding all the functions in the module. Measuring does not break in case of case expressions, namely when the case is not embedded [25].

min_depth_of_cases

Gives the minimum of the maximums of case control structures embedded in case of a concrete function (how deeply are the case control structures embedded). In case of a module it measures the same regarding all the functions in the module. Measuring does not break in case of case expressions, namely when the case is not embedded into a case structure. However, the following embedding does not increase the sum [25].

max_depth_of_structs

Gives the maximum of structures embedded in function (how deeply are the block, case, fun, if, receive, try control structures embedded). In case of a module it measures the same regarding all the functions in the module [25].

number_of_funclauses

Gives the number of functions clauses. Counts all distinct branches, but does not add the functions having the same name, but different arity, to the sum [25].

branches_of_recursion

Gives the number of a certain function's branches, how many times a function calls itself, and not the number of clauses it has besides definition [25].

calls_for_function

This metric gives the number of calls for a concrete function. It is not equivalent to the number of other functions calling the function, because all of these other functions can refer to the measured one more than once [25].

calls_from_function

This metric gives the number of calls from a certain function, namely how many times does a function refer to another one (the result includes recursive calls as well) [25].

number_of_funexpr

Gives the number of function expressions in a module. It does not measure the call of function expressions, only their initiation [25].

number_of_messpass

In the case of functions, it measures the number of code snippets implementing messages from a function, while in case of modules it measures the total number of messages in all of the modules functions [25].

fun_return_points

The metric gives the number of the functions possible return points (or the functions of the given

module) [25].

average_size

The average value of the given complexity metrics (e.g. Average branches_of_recursion calculated from the functions of the given module) [25].

max_length_of_line

It gives the length of the longest line of the given module or function [25].

average_length_of_line

It gives the average length of the lines within the given module or function [25].

no_space_afte_comma

It gives the number of cases when there are not any whitespaces after a comma or a semicolon in the given module's or function's text [25].

is_tail_recursive

It returns with 1 if the given function is tail recursive; with 0, if it is recursive, but not tail recursive; and -1 if it is not a recursive function (direct and indirect recursions are also examined). If we use this metric from the semantic query language, the result is converted to tail_rec, non_tail_rec or non_rec atom [25].

mcCabe

McCabe cyclomatic complexity metric. We define it based on the control flow graph of the functions with the number of different execution paths of a function, namely the number of different outputs of the function [25].

otp_used

Gives the number of OTP callback modules used in modules [25].

3.3 Metric visualisation module for WEB2 interface of RefactorErl

The section describes the main concept and details of the developed framework for RefactorErl static analysis tool. The first part introduces the main features of the software. The next part describes which tools were used in the developing process of the module. In the last part, we can read how to use the software step-by-step.

3.3.1 Description of the software

The program which was developed allows analyzing git repositories with Erlang code files. The component is built on RefactorErl static analysis tool and actively uses its feature of calculating different metrics of Erlang modules and functions. It has convenient user-friendly web interface with repository structure as a folder tree and a canvas where plots can be observed. These plots contain information how particular metric was changing with repository evolution. The plot can be saved for future use as a picture in PNG format.

The main features of the software are:

- Drawing plots which show the change of metrics with software evolving from version to version.
- Analyzing modules and functions separately.
- Choosing separate files for the analyzing.
- User-friendly web interface.

The main focus of the project is to help Erlang developers with analyzing their projects using plots. Visualization helps with finding patterns and improving the code quality.

3.3.2 Used tools

The interface of the program is the AngularJS component which uses NVD3 library for plot rendering. Metrics data is stored in DETS tables at the stage of calculation in Erlang code. This data passes as JSON objects when Erlang function communicates with javascript code. For saving plots as pictures the saveSvgAsPng library is used.

NVD3

This library currently under development of a team of software engineers at Novus Partners company. NVD3 is the D3 based JavaScript library. It allows creating beautiful and reusable charts in web applications.

It has wonderful features for data visualization with nice-looking charts such as the usual box-plot, line and bar charts and fancier candlestick and sunburst charts. If you need a lot of functionality in a JavaScript chart plotting library, NVD3 is the very nice option for your project.

AngularJs

The new component was created for the existing system using this javascript web framework.

AngularJS (also written as Angular.js) is an open-source JavaScript-based front-end web application framework mainly developed by Google and by a community of individuals and corporations

to solve many of the challenges which can be observed in the development of single-page applications.

The AngularJS framework starts his work by reading the Hypertext Markup Language (HTML) page, which has extra tag attributes embedded into it. Angular transforms these attributes as directives to bind input or output parts of the page to a model that is defined by standard JavaScript variables. The values of these JavaScript variables can be manually set in the code, or fetched from static or dynamic JSON resources.

DETS tables

Data of calculated metrics for modules and functions stored in two different tables: mods_ metrics and funs_metrics.

DETS is Disk Erlang Term Storage. DETS tables store tuples, with access to the elements given through a key field in the tuple. The tables are implemented using hash tables and binary trees, with different representations providing different kinds of collections [26].

JSON

Data, which stored in Dets tables, transforms into JSON objects when Erlang code interact with JavaScript code. JSON (JavaScript Object Notation) is a simple format of data-interchange. It is a text format that is absolutely language independent but it uses a convention that is familiar to programmers of the C-family of languages which includes C, C++, C#, Java, JavaScript, Python, Perl, and others. This property makes JSON an ideal language of data-interchange.

JSON is built on two data structures:

- A collection of name/value pairs. In various languages, this is realized as an object, struct, record, hash table, dictionary, associative array or keyed list.
- An ordered list of values. Usually, this is realized as an array, list, vector, or sequence.

saveSvgAsPng

This small library is used for saving SVG plots as PNG pictures. Despite its small size, it has a lot of different options such a choosing particular background, font, scale etc.

3.3.3 The typical workflow

For using the component WEB2 interface should be run. It can be done with this command executed in RefactorErl shell:

```
ri:start_web2([{yaws_path, PATH-TO-YAWS}]).
```

This command will run WEB2 interface available on localhost:8001 by default. After loginning the component can be accessed in the metrics tab as shown in Figure 3.2.

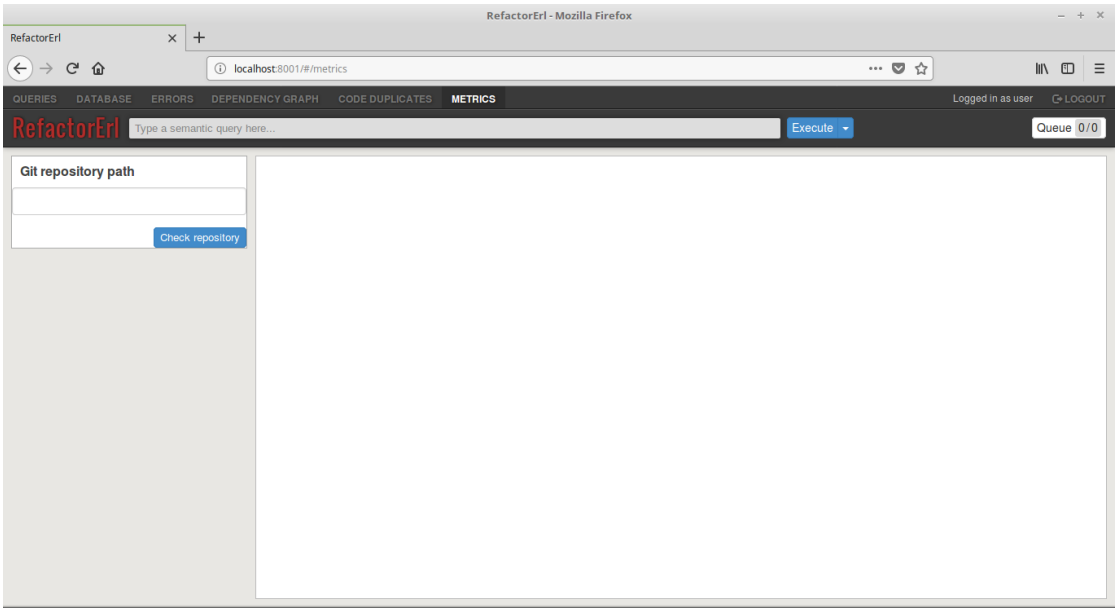


Figure 3.2: Component interface.

The path to git repository should be provided in "Git repository path" input. After clicking the "Check repository" button the folder will be analyzed. If it is not valid the alert Figure 3.3 will be shown.

In another case, the folder tree will be available where separate files can be chosen for future analyzing as shown in Figure 3.4. Also, there is a possibility to delete some files chosen by mistake.

The final step is pressing the "Analyze" button. It will start calculating metrics for all versions of the repository and progress will be shown on screen Figure 3.5.

After analysis is done the menu with choosing parameters of drawing the plot will be available as shown at Figure 3.6. It consists of two selection lists. The first one is the list where we can choose the type of item for plots drawing (module or function). Depends on the chosen type the select boxes with available metrics will differ.

For modules: module sum, line of code, char of code, number of fun, number of macros, number of records, included files, imported modules, number of funpath, function calls in, function calls out, cohesion, max depth of calling, max depth of cases, min depth of cases, max depth of structs, number of funclauses, branches of recursion, number of funexpr, number of messpass, fun return points, average size, max length of line, no space after comma, mcCabe, otp used.

For functions: line of code, char of code, function sum, max depth of calling, max depth of cases, min depth of cases, max depth of structs, number of funclauses, branches of recursion, calls for function, calls from function, number of funexpr, number of messpass, fun return points, average size, max length of line, no space after comma, is tail recursive, mcCabe. Another list is the list of all items which can be chosen for metrics plot drawing.

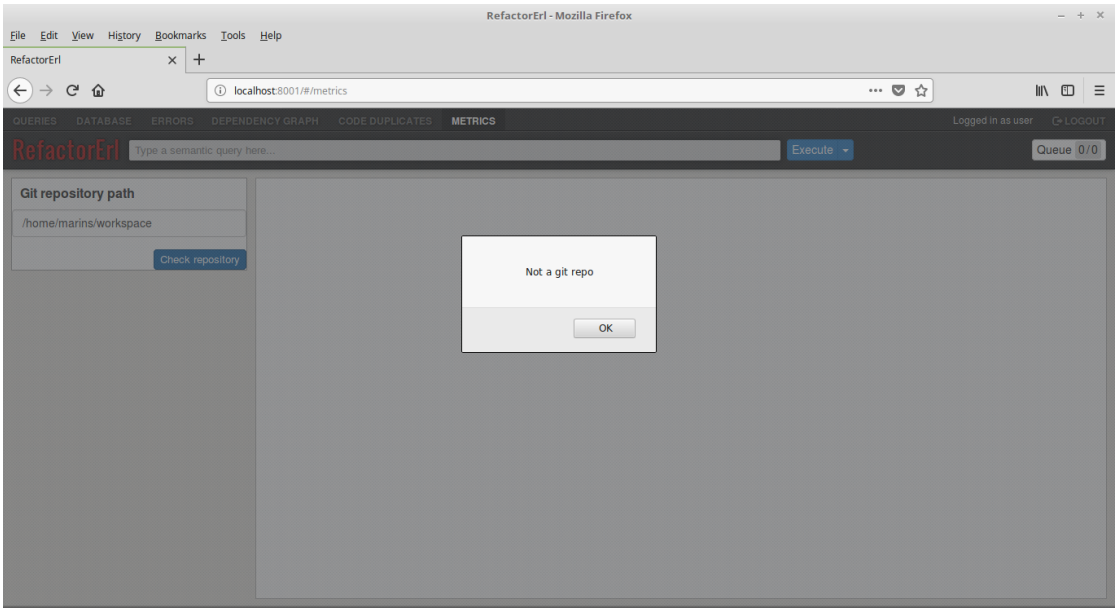


Figure 3.3: Alert showed because the provided folder is not a valid repository.

After the plot is shown the "Save" button appears which can be pressed to save SVG plot in a PNG file.

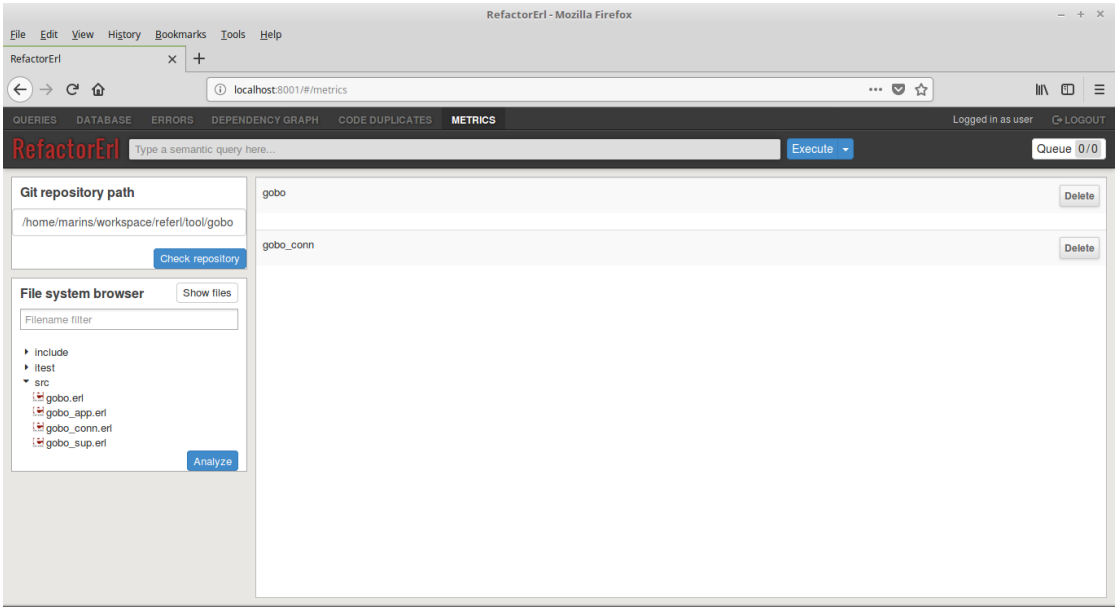


Figure 3.4: Repository tree.

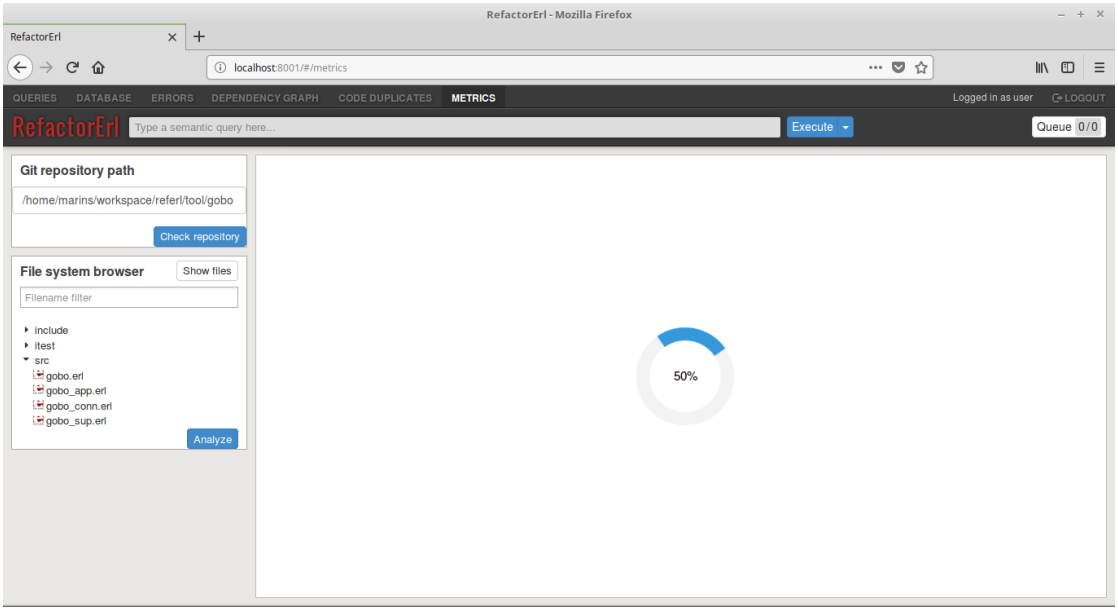


Figure 3.5: The process of repository analyzing.

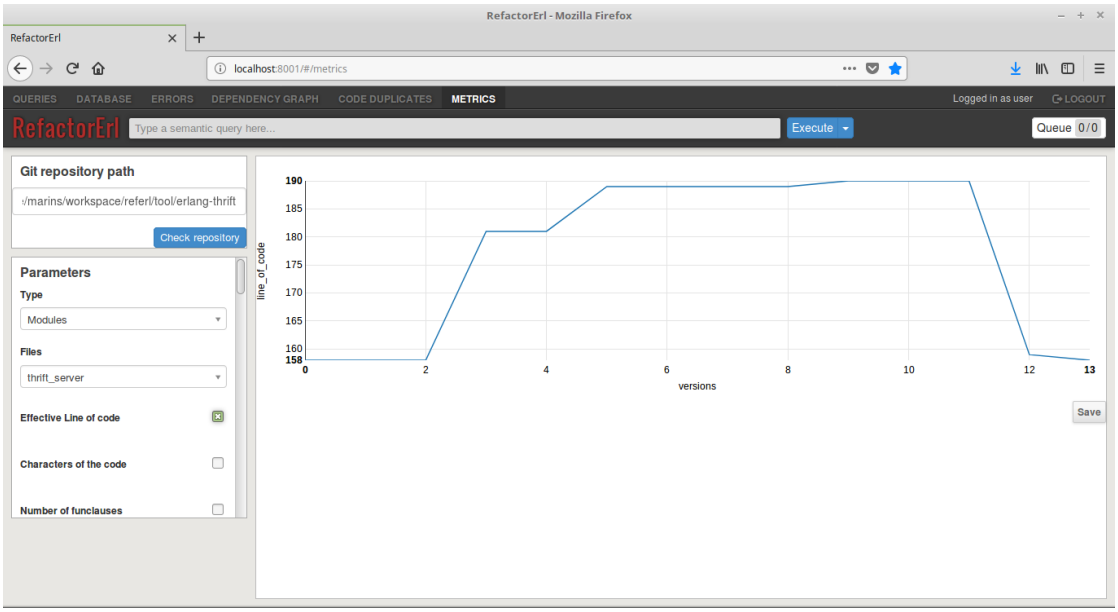


Figure 3.6: The example of plot.

Chapter 4

Measurements and findings

The aim of this chapter is to test and analyze projects from git by using the developed module for RefactorErl. Git commit logs hold all the change history. This is an excellent source to observe trends and patterns about projects. All projects selected for the experiment are written in Erlang and have more than 40 commits.

4.1 Iron

This project is functional Erlang Toolkit. Iron is released under the MIT license. It can do the foolowing:

- Count with coerce equality, count with a custom predicate.
- Find with coerce equality, find with a custom predicate.

This project has just only one source code file with 68 commits. The link to the repository is `https://github.com/elementerl/iron`.

We can see that with the version number increase the line of code number and char of code number also grow on Figure 4.1 and in Figure 4.2.

As shown in Figure 4.3 developer started to use otp library after 45th version.

The Figure 4.4 shows an overview of the evolution of the overall McCabe cyclomatic complexity. We can observe that the complexity keeps increasing.

There was a spike in 45th version, because some functions were called to an other module, however author removed changes in 46th version. There was also an insignificant drop in complexity in 28th version and a little increase in 29th version. The complexity decreases when the extracted

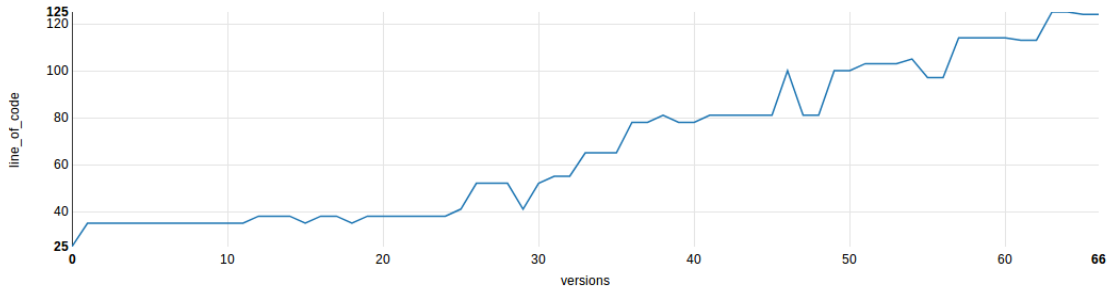


Figure 4.1: Effective Line of code for fe.erl file.

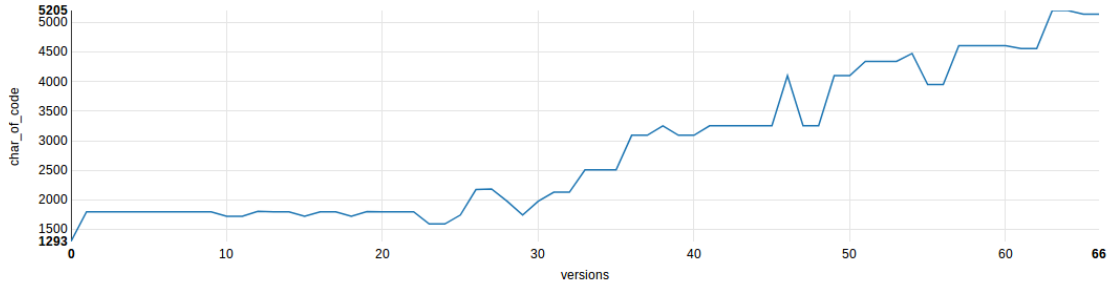


Figure 4.2: Char of code for fe.erl file.

piece of code occurred more than one time and the complexity of the function is more than one [27].

In Figure 4.5 we can see that the metric **max_depth_of_cases** is 1, but before it was 0, therefore developer stopped using case inside another case for this version of the software. We can find the same trend on plot with **mcCabe** metric where it appears as decreasing of the program complexity.

The average length of the line was not stabilized until 35th version with gradually decreasing from 50 symbols to 38 symbols in Figure 4.6.

As we can see in Figure 4.7 and Figure 4.8 there are not defined macroses and records in the whole iron project.

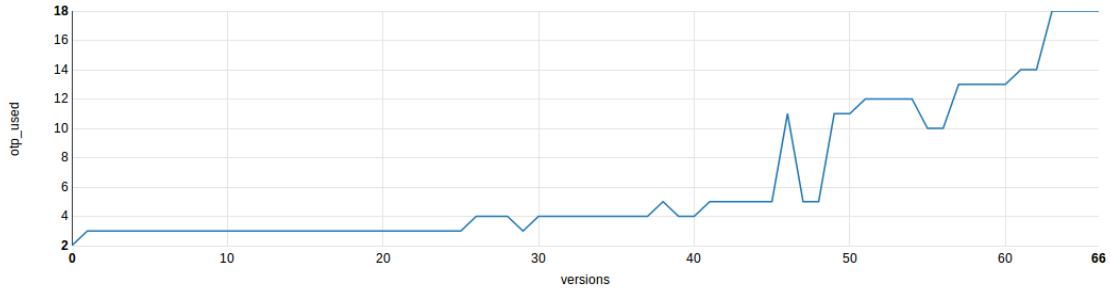


Figure 4.3: Otp used for fe.erl file.

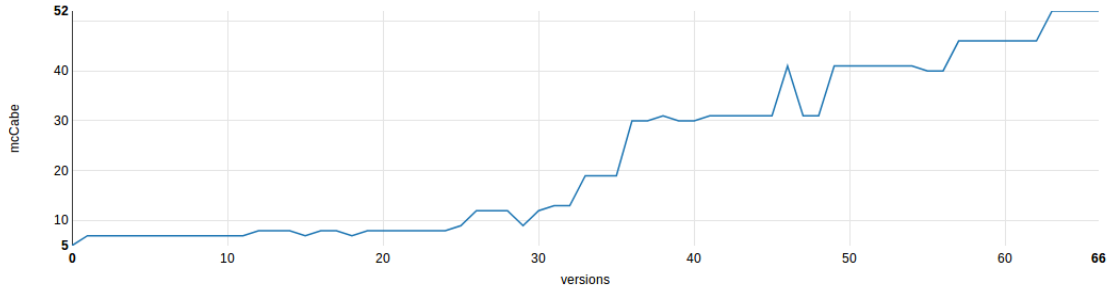


Figure 4.4: McCabe cyclomatic complexity metric for fe.erl file.

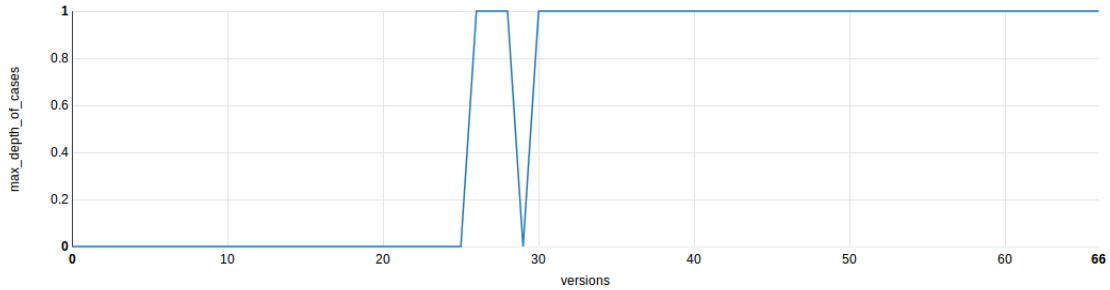


Figure 4.5: Max depth of cases for fe.erl file.

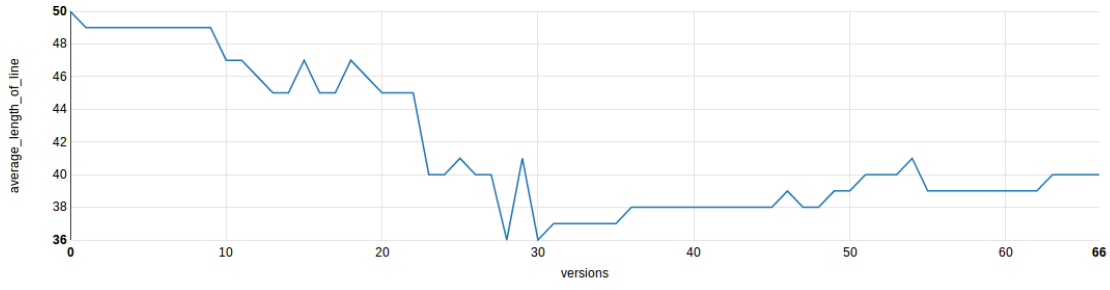


Figure 4.6: Average length of line for fe.erl file.

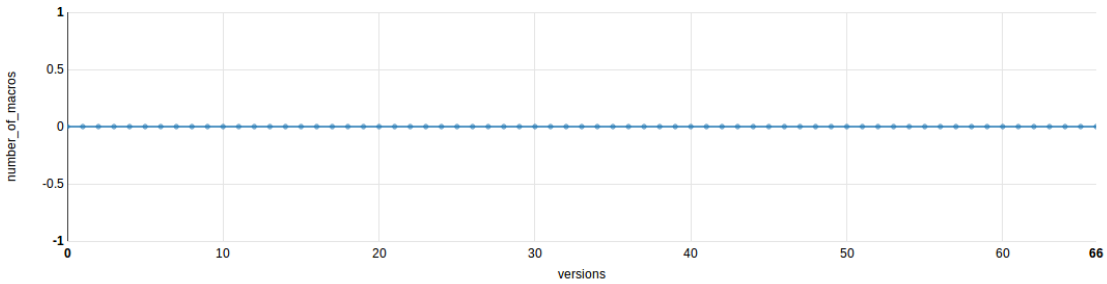


Figure 4.7: Number of macros for fe.erl file.

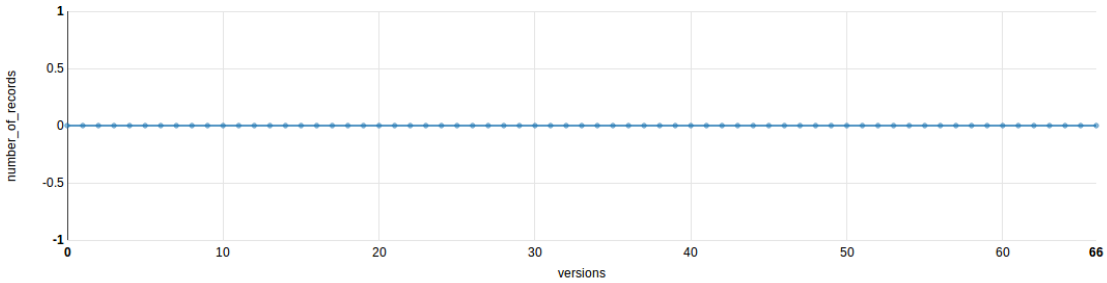


Figure 4.8: Number of records for fe.erl file.

4.2 Erlang chat

This project is multi-user chat written in Erlang. It has nine source code files and 45 commits. The link to the repository is `https://github.com/bildeyko/erlangChat`.

For this project, we analyzed the module **websocket_handler.erl**. This module consists of 6 functions.

We can see an increasing number of lines of code in Figure 4.9 and an increasing number of characters in a program text in Figure 4.10.

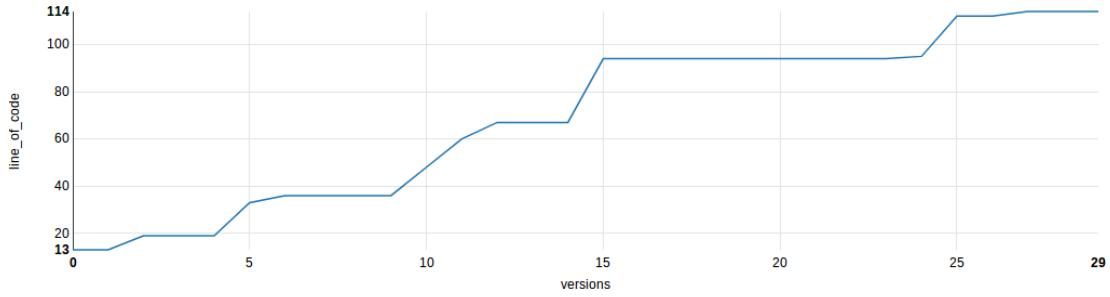


Figure 4.9: Effective Line of code for module websocket_handler.erl.

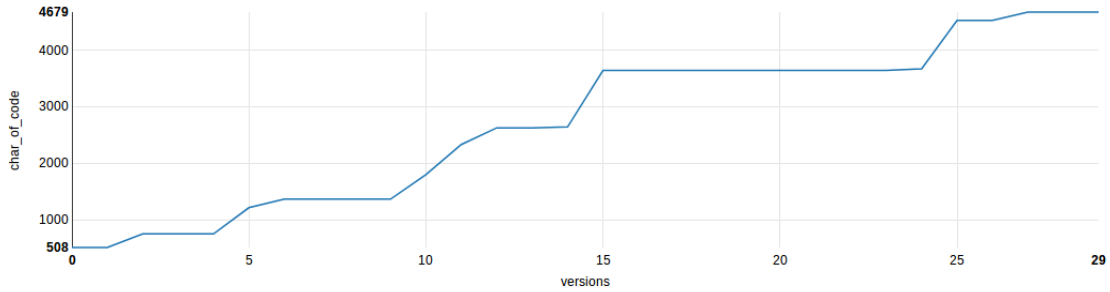


Figure 4.10: Characters of the code for module websocket_handler.erl.

In Figure 4.11 we can see that the author started using message passing from the 9th version of his software.

As shown in Figure 4.12 there was increase in the length of the longest line of code in 9th version.

McCabe's cyclomatic complexity metric measurement guarantees that developers are sensitive to the fact that programs with high McCabe numbers, for example, more than 10 are likely to be hard for understanding and accordingly have a higher probability of defects containing within the code base. The tested module has the cyclomatic complexity number which increased to 30 in the last versions as shown in Figure 4.13.

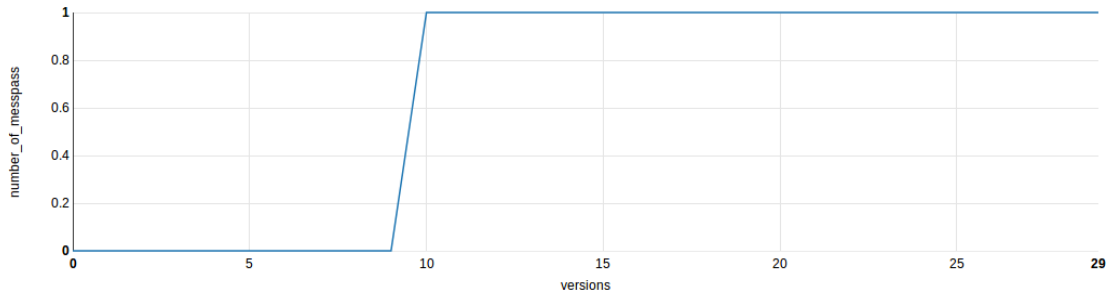


Figure 4.11: The number of message passing for module websocket_handler.erl.

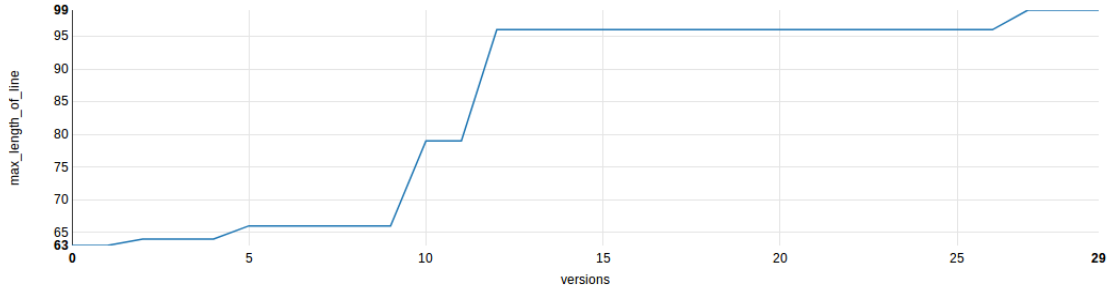


Figure 4.12: Max length of line for module websocket_handler.erl.

In Figure 4.14 shows that developer used otp library functions but after 9th version reconsidered to use them.

The developed framework allows measuring and visualizing metrics for a module and also for each function in the module. For example, in this module developer use message passing from the 9th version as we mentioned above. The Figure 4.15 shows that there was discovered function in which the author actively used message passing.

Visualizing of metrics helps to find which functions have been changed, added or deleted. As shown in the Figure 4.15 the developer slightly changed the function **terminate/3** by adding two lines of code in the 9th version.

To summarize findings we can assume that most of these changes were done in the 9th version.

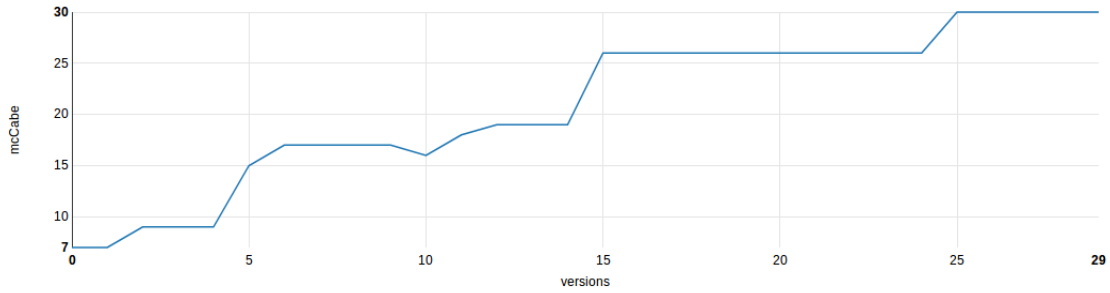


Figure 4.13: McCabe cyclomatic complexity metric for module websocket_handler.erl.

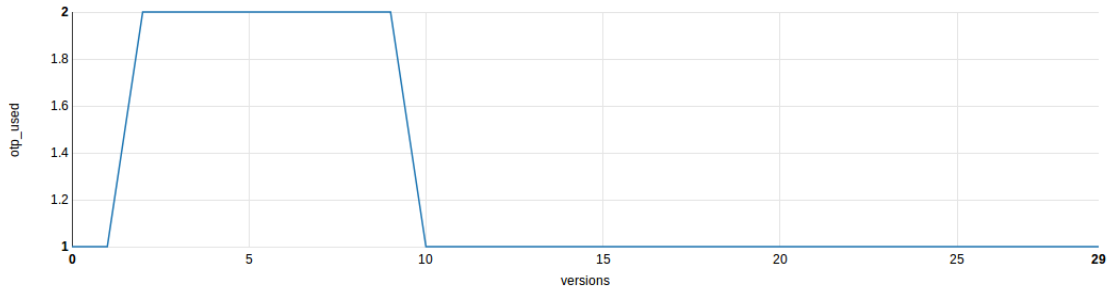


Figure 4.14: Otp used for websocket_handler.erl.

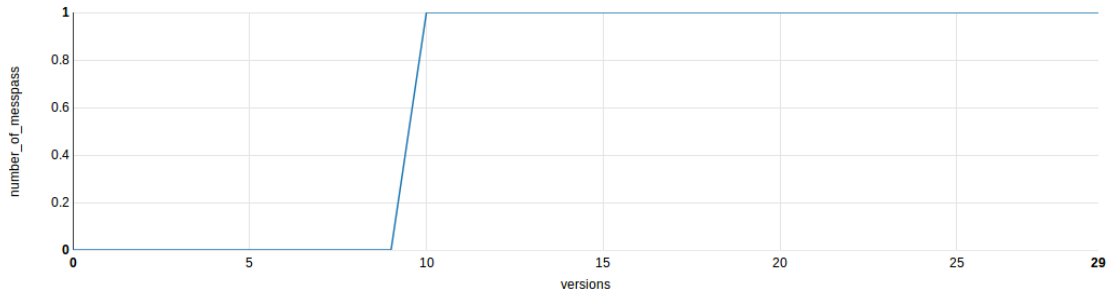


Figure 4.15: Number of message passing for function websocket_handle/3.

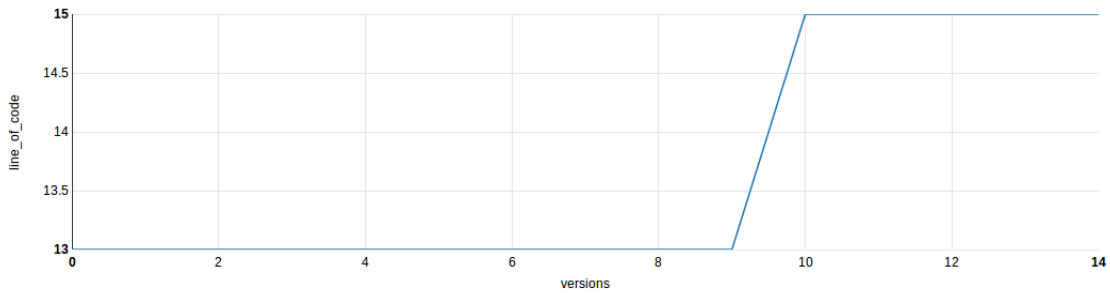


Figure 4.16: Effective lines of code for function terminate_/3.

4.3 prx

This project is an Erlang library for Unix process management and system programming tasks. Code from all project is divided into 4 modules.

The project provides:

- Reliable operating system process management by mapping Erlang processes to a hierarchy of system processes.
- Beam-friendly interface for system calls and other POSIX operation.
- Operations for processes isolation like jails and containers.
- An interface for separation operations with privileges for processes restriction.

The link to the repository is `https://github.com/msantos/prx`. This project has 201 commits.

For this project, we tested the module **prx.erl**.

As in previous two experiments, at the beginning, we started to measure the **LOC** metric. This metric helps us to see the changes all of the project from over time. This result is shown in Figure 4.17.

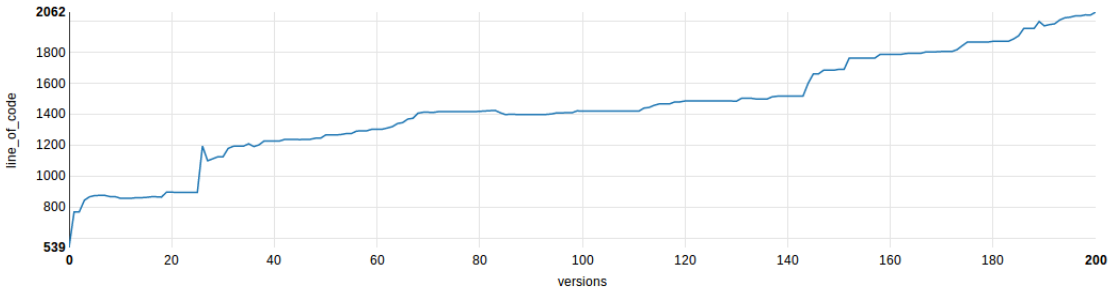


Figure 4.17: Effective Line of code for module prx.erl.

Another important metric is **cohesion** metric. Modules with large cohesion number is preferable because high cohesion is associated with many desirable attributes of software such as robustness, reusability, reliability, and understandability. Otherwise, low cohesion is associated with undesirable attributes, for example, being difficult to reuse, maintain, test, or even understand. The Figure 4.18 and shows the decreasing of the calculated metric.

When developer started to use otp library in this project in 80th version, as we can see in Figure 4.20, the McCabe cyclomatic complexity metric was rapidly increased in Figure 4.19. If complexity is increasing dramatically between versions, it is an indication of logic being added.

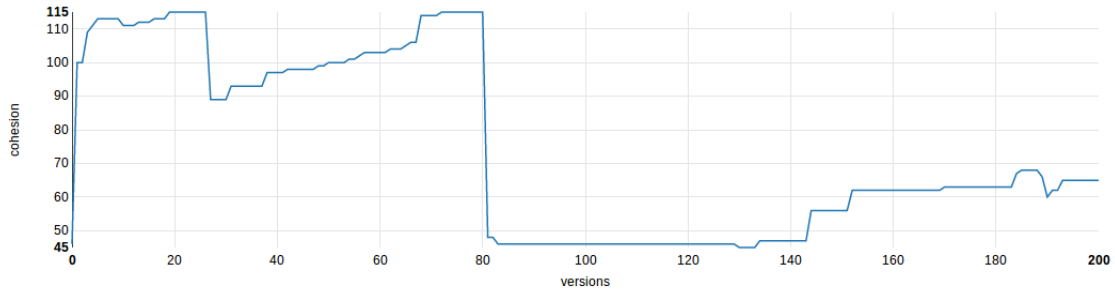


Figure 4.18: The cohesion of the module prx.erl.

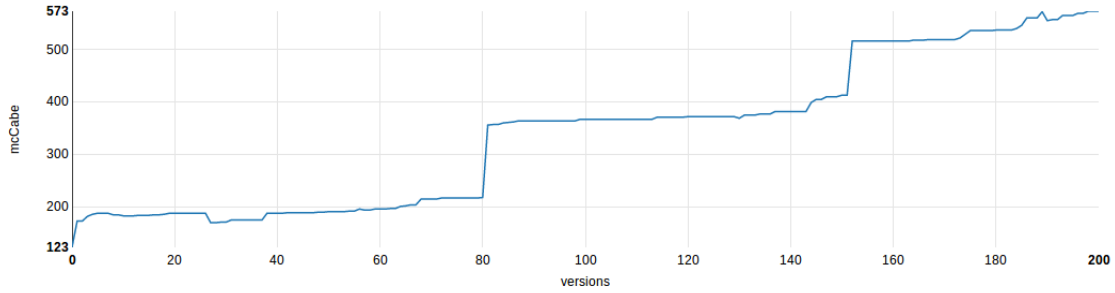


Figure 4.19: McCabe for the module prx.erl.

When we are comparing results of cohesion and McCabe cyclomatic complexity metrics, we can conclude that low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

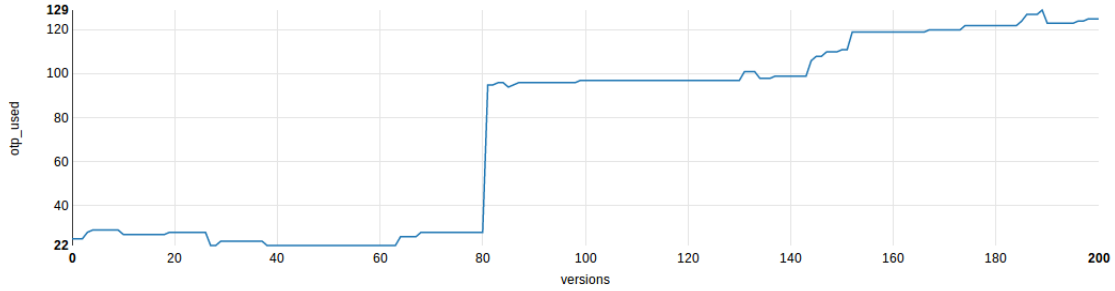


Figure 4.20: OTP used for the module prx.erl.

Apart from analyzing the changes of the module we also can observe the changes of functions. The Figure 4.21 shows that the function **find/2** was used only in one version and later was renamed or deleted.

One of the features of functional programming languages is the presence of tail recursion. It is a special form of recursion where the last operation of a function is a recursive call [28]. The

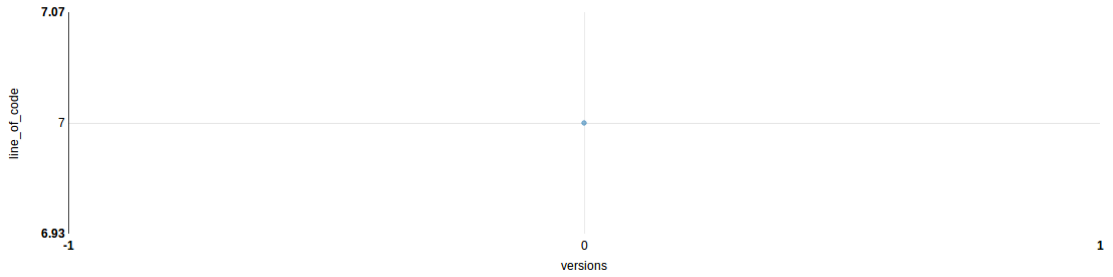


Figure 4.21: Effective Line of code for function find/2.

metric **is_tail_recursive** returns with 1, if the given function is tail recursive; with 0, if it is recursive, but not tail recursive and -1 if it is not a recursive function. As shown in Figure 4.22 we can see that developer got rid of this function from 37th version until 40th version.

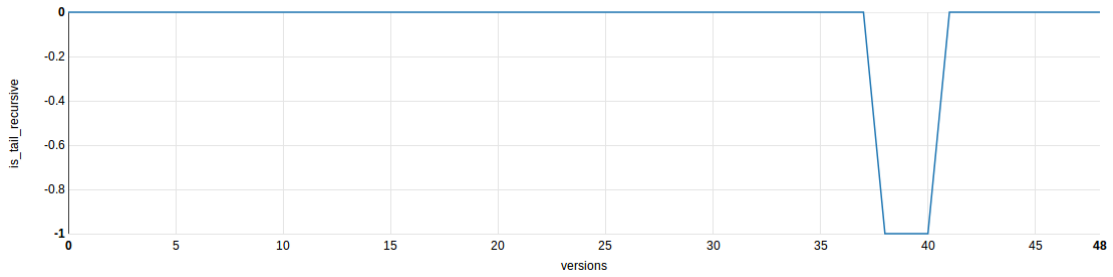


Figure 4.22: Is tail recursive metric for function filter/2.

Also we can calculate how many times a function calls itself by using **branches_of_recursion** metric for all module. The Figure 4.23 illustrates that there were created more recursive functions after 140th version.

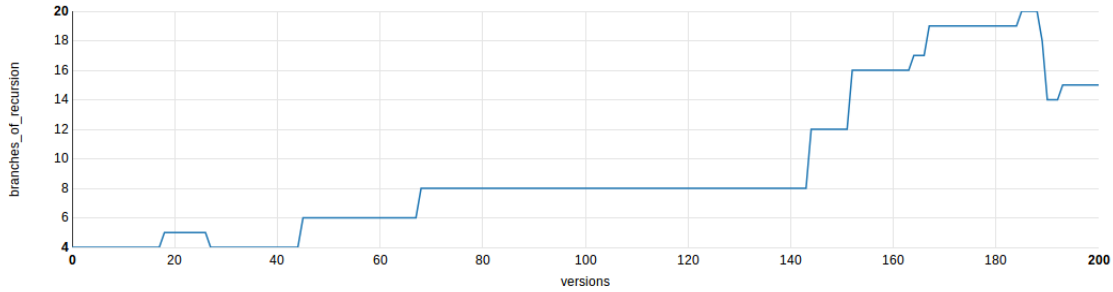


Figure 4.23: Branches of recursion for module prx.erl.

In RefactorErl there is a **function_sum** metric which can be calculated using these metrics together: **line_of_code**, **char_of_code**, **number_of_funclauses**, **branches_of_recursion**, **mcCabe**, **calls_for_function**, **calls_from_function**, **fun_return_points**, **number_of_messpass**. Metrics **calls_for_function** and **calls_from_function** available only

for functions. We can notice the dependency of **function_sum** (Figure 4.24) on two metrics changes: number of lines of code (Figure 4.26) and calls from the function (Figure 4.25).

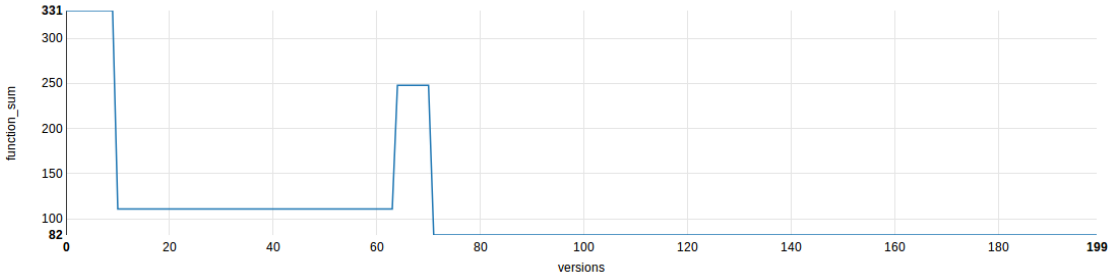


Figure 4.24: Function sum function task/4.

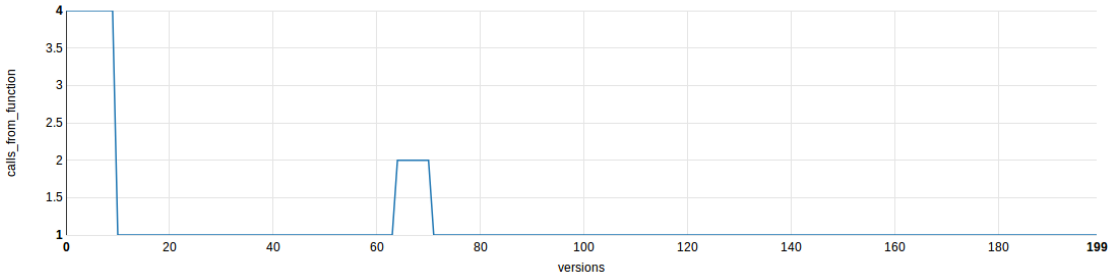


Figure 4.25: Calls from the function for function task/4.

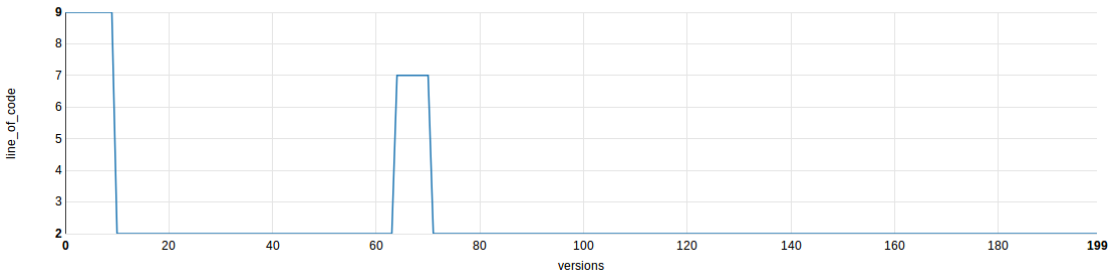


Figure 4.26: Effective Line of code for function task/4.

Chapter 5

Related work

Nowadays, the need for the visualization of software quality metrics has been rapidly increased. Software metrics help developers and companies to check and analyze information about the performance, quality of code and cost of software data. it helps to find out and fix errors in the early stages of development.

In this chapter will be described some tools for visualization of software quality metrics and tool for code analysis.

5.1 Open Source tool "METRIX"

This tool can compute different software quality metrics. METRIX is able to evaluate software written in C and ADA languages and many metrics can be considered for software evaluation (the different metrics will be described in detail in the next section [1].

The user can use different types of diagrams for metrics visualization: histograms, scatter plots and line charts. This tool allows to use two not common classes of diagrams:

- The first is Kiviat diagrams, also known as the radar plots.
- The second is the city map diagram.

One specific feature of METRIX is to use those two types of visualization for constructing signature and cartography of source codes [1].

Kiviat diagrams The Kiviat diagram uses polar coordinates for visualization. The value which user wants to represent associated with the distance between the point and the origin. The angle between two points does not change because it is constant. This constant is calculated to uniformly distribute the different points. Linked together metric points make a plain polygon,

which forms the data specifics of represented values. This type of diagram can be used only for visualization of more than three values of calculated metric. All values must be strictly positive. The user can merge data in one diagram. Also, this tool allows drawing the diagram for one or several metrics together.

In Figure 5.1 shows an example of two Kiviat diagrams. These diagrams represent different metric values for two functions.

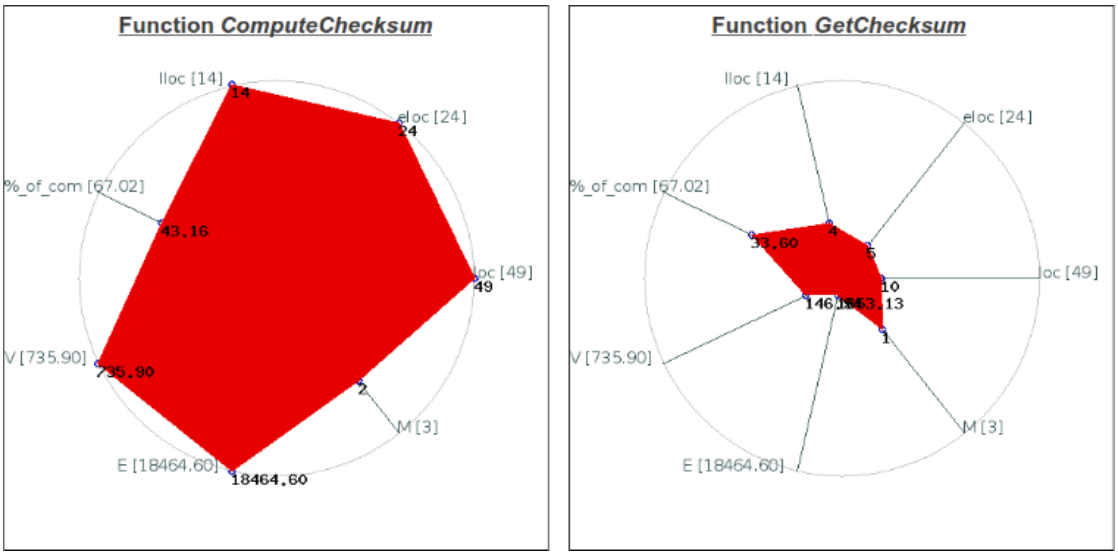


Figure 5.1: An example of Kiviat diagrams.

City map diagrams Usually this type of diagrams used for representing cities with buildings in three dimensions. City map diagrams also suit for visualization of software metrics of the project with a large number of source files, functions and variables. Source code metrics are represented by rectangles by using the treemap diagram. Some authors used variants of this class of visualization diagrams to represent values with esthetic considerations. City map diagrams are indeed multi-parameters diagrams: each “building” is associated with an n-tuple of numeric values [1].

This tool also has a graphical user interface with a single Window containing three tabs: “Calc”, “Csv” and “Plots”.

In the first "Calc" tab user chose source files for evaluation and also chose metrics which neede to be measured.It allows to users to use hand-start scripts with a prompt. The open source aspect of the tool allows users to repeat the measuring scripts manually or to integrate them into other software.

In the second “Csv” tab user can find a list of numeric values and chose which values will be used for function or for the file. There is an option for exporting calculated values to a spreadsheet application for drawing graphs, like scatter plots, line plots, etc.

In the third “Plots” tab the user can build the city map diagrams for the source code. The user

can change the settings of the tool like the modification of placement of the buildings in the city map or make the data colored, etc.

This tool supports making a report in the form of a LaTeX file where each function and each file make a section of this report. User can use PDF file for analisis. This report includes all the values measured on the project. Each file produces three views of the associated city map diagram, as well as different Kiviat diagrams for the different metrics measured on the functions of the file [1].

5.2 Sextant

Sextant is a Java source-code analysis tool under development at the University of Nebraska at Omaha (UNO) [29]. This software is a complex extension of the TL system (a general-purpose program transformation system) created particularly for the Java programming language domain.

The main design goal of Sextant is providing a tool facilitating specification and visualization of custom analysis rules, which can be domain-specific or moreover application specific analysis rules.

Analysis rules of Sextant are based on information fetched from different software models. There are two main models of central importance. The first model is a syntactic model. It is the source code parse tree. Parse trees conform to compilation units which represented by Java files and are generated with use of GLR parser technology provided by the TL system. Parse trees are well-fitted for analyzing and manipulating through standard primitives provided by program transformation systems, for example, by matching and generic traversal.

The second model is a compound attributed graph (CAG). It is a semantic model which captures subtype, structural, and reference dependencies among the constructors, methods, fields, packages and types. The CAG also links an attribute list with every node and edge.

CAG information is accessible to Sextant's transformation-based analysis rules via two mechanisms. The first mechanism is a positional system which organizes a relation between contexts within the CAG and corresponding parse (sub)trees. This relation makes possible correctly resolving references to constructors, methods, types, fields, and local variables, during the process of generic traversals which are the key enabling mechanism in program transformation systems.

The second mechanism is a library of semantic queries. These queries can be accessed even in the middle of transformation course. Functionality which provided by this library contains things like:

- Reference resolving.
- Reference type determining.
- Determining declaration shadowing or overriding.
- Determining whether one type is a subtype of another.

Sextant stores table and set types for information collecting with relation to analysis rules. These constructs can be used for storing information related to a custom metrics wide variety.

Sextant is open-ended with respect to the definition of metrics – any source-code analysis rule can be interpreted as a metric, be they PMD-style rules focusing on violations of coding conventions or rules such as those specified by FindBugs that are more semantic in nature [29].

Sextant can do software models generation. These models can be visualized using other tools such as GraphViz, Cytoscape and TreeMap. Sextant can produce the CAG of the code base in a JavaScript object notation format (JSON). This JSON file can be loaded into Cytoscape, an open source platform which provides extensive and sophisticated capabilities for large complex networks visualization, for example, graph structures. The same way, metrics derived from sets and tables can be produced in CSV format and viewed with use of TreeMap. Parse trees can be output as dot-files and later viewed with use of GraphViz.

The view in Figure 5.2 shows an example of represents a coloring of dependencies on the unsupported features. Nodes colored orange have indirect dependencies on unsupported features while purple nodes have direct dependencies on unsupported features.

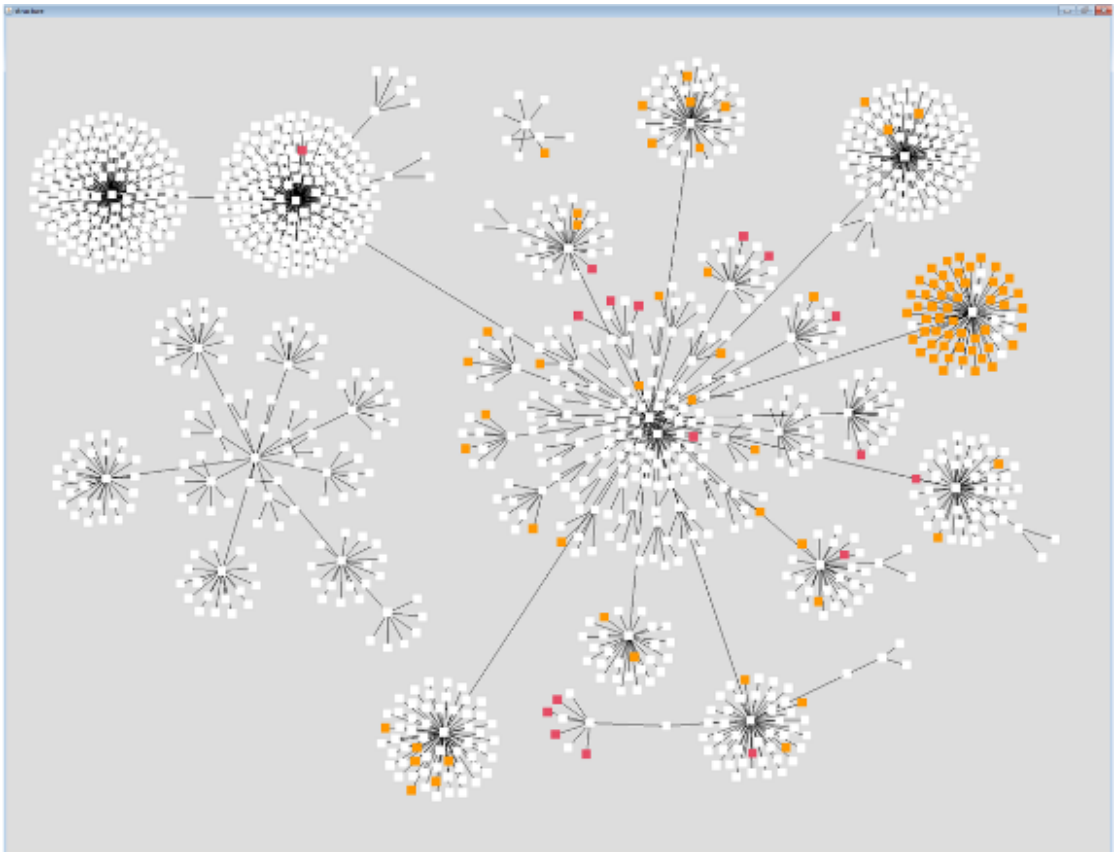


Figure 5.2: An example of using Sextant tool.

5.3 NDepend

NDepend is a static analysis tool for .NET managed code. The tool can measure a large number of code metrics, allowing to visualize dependencies using directed graphs and dependency matrix.

NDepend computes many size-related metrics: number of lines of code, number of assemblies, number of types, number of methods, etc. For measuring complexity, NDepend uses Cyclomatic Complexity. This metric calculates the complexity of a method or a type by calculating how many branching points it contains in the code.

NDepend has two metrics for cohesion. Relational Cohesion is an assembly level metric that calculates the average number of internal relationships for each type. Lack of Cohesion Of Methods (LCOM) measures a type cohesiveness. A type is maximally cohesive if every method use all instance fields.

NDepend has a tool for visualization called a Treemap. Also, NDepend has a dashboard for quick visualization of all application metrics. It's shown in Figure 5.3. The dashboard is available as the extension for Visual Studio. The dashboard shows the diff since baseline for every metric. It also displays if the metric value becomes better (in green) or worse (in red).

The Figure 5.4 shows metric visualization using the colored treemap.

The tree structure used in NDepend treemap is the usual code hierarchy:

- .NET assemblies contain namespaces.
- Namespaces contain types.
- Types contain methods and fields.

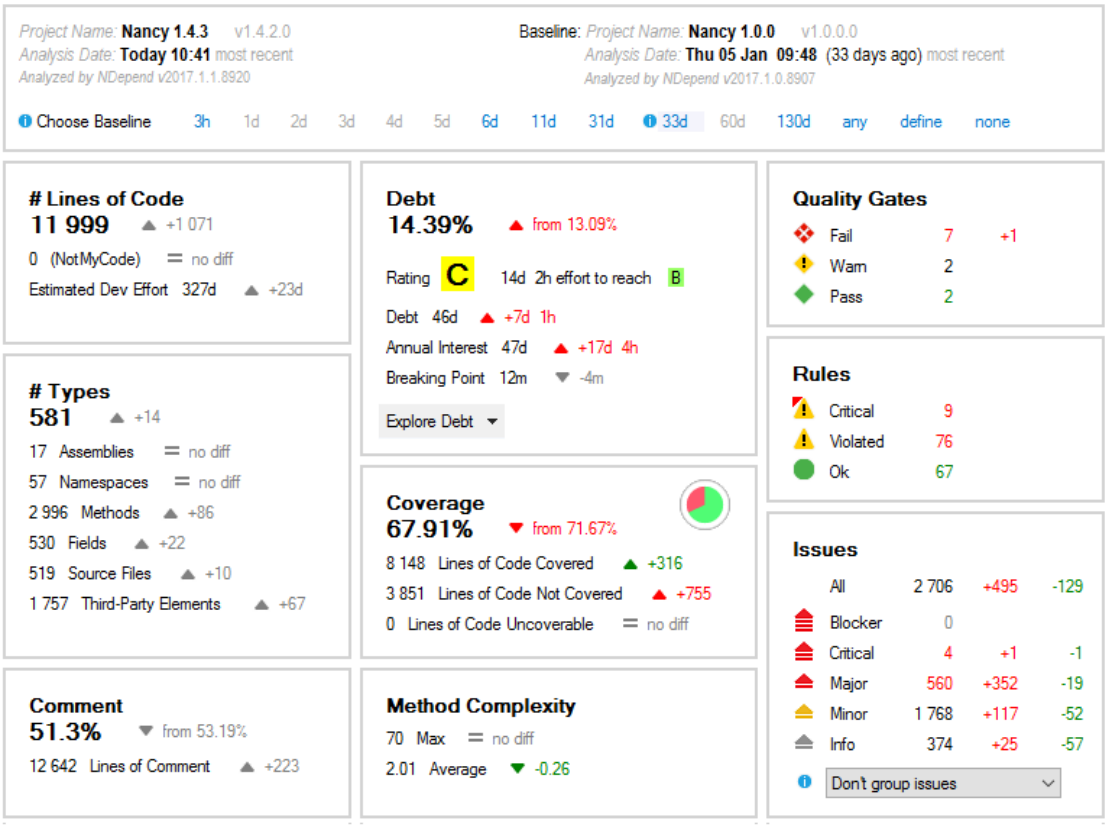


Figure 5.3: An example of using the dashboard.

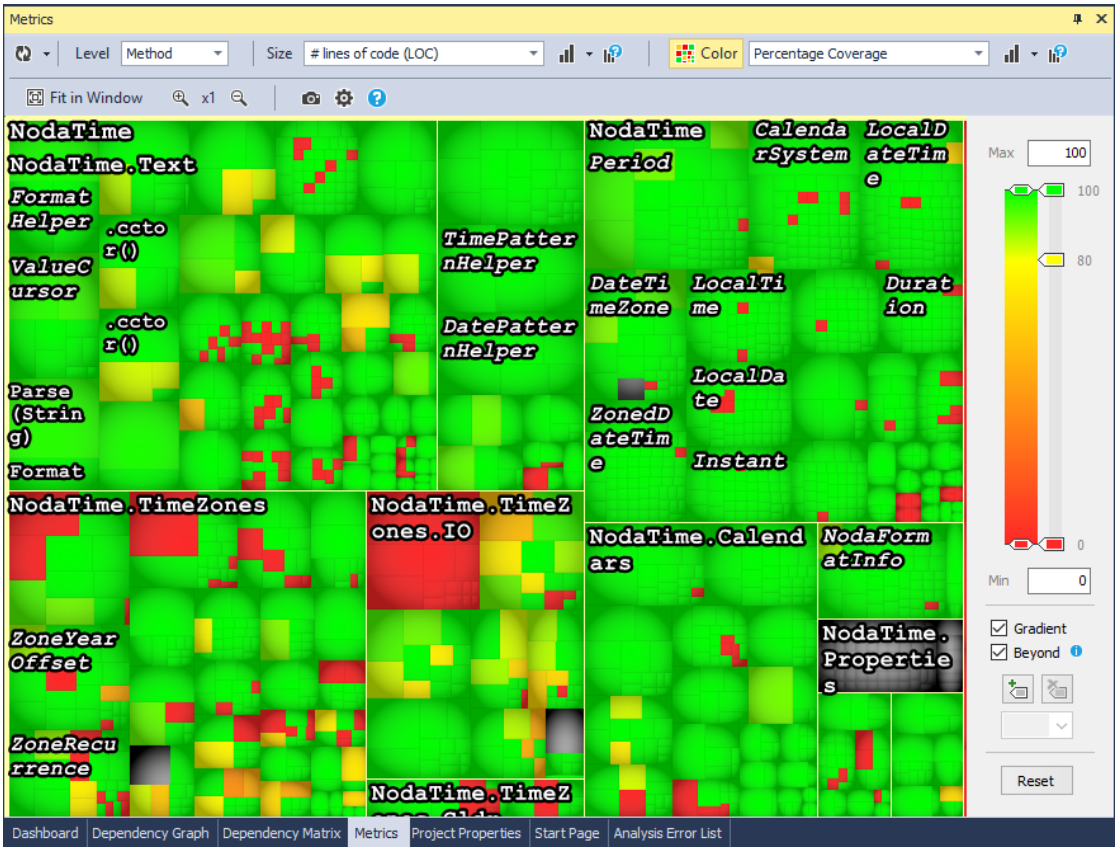


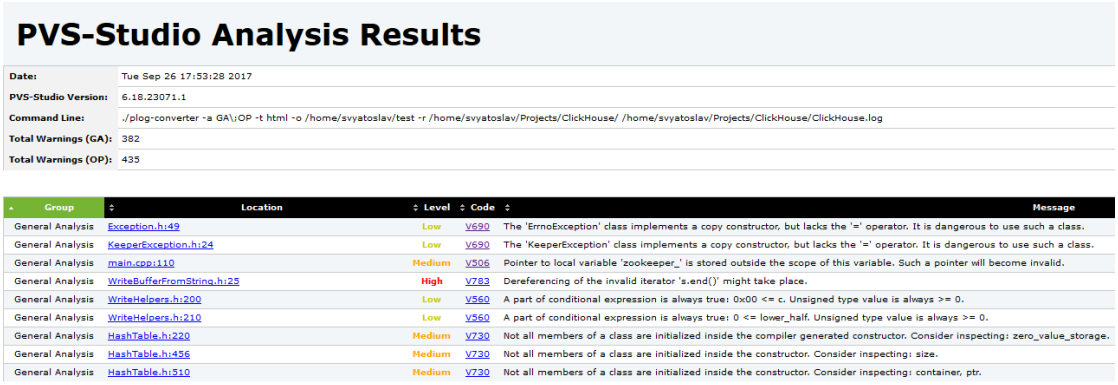
Figure 5.4: An example of using the colored treemap.

5.4 PVS-Studio

PVS-Studio is a tool for detecting bugs and security weaknesses in the source code of programs, written in C, C++, and C#. It works in Windows, Linux, and macOS environment [30]. The user can use online reference manual concerning all the diagnostics methods available in the program and on the website.

This tool executes static code analysis. The generated report helps developers in debugging. PVS-Studio has large number of methods for checking. These methods help to find typing and copy-paste mistakes. The main value of static analysis is in its regular use so that errors are identified and fixed at the earliest stages [30].

PVS-Studio can work on a Linux environment on Windows. It can be run from the command or can be integrated into Visual Studio development environment. The results of the analysis are presented in the HTML document which allows full navigation through the source code. The Figure 5.5 shows an example of an HTML-report.



PVS-Studio Analysis Results

Date:	Tue Sep 26 17:53:28 2017
PVS-Studio Version:	6.18.23071.1
Command Line:	./plog-converter -a GA\;OP -t html -o /home/svyatoslav/test -r /home/svyatoslav/Projects/ClickHouse/ /home/svyatoslav/Projects/ClickHouse/ClickHouse.log
Total Warnings (GA):	382
Total Warnings (OP):	435

Group	Location	Level	Code	Message
General Analysis	Exception.h:49	Low	V690	The 'ErrnoException' class implements a copy constructor, but lacks the '=' operator. It is dangerous to use such a class.
General Analysis	KeeperException.h:24	Low	V690	The 'KeeperException' class implements a copy constructor, but lacks the '=' operator. It is dangerous to use such a class.
General Analysis	main.cpp:110	Medium	V506	Pointer to local variable 'zookeeper_' is stored outside the scope of this variable. Such a pointer will become invalid.
General Analysis	WriteBufferFromString.h:25	High	V783	Dereferencing of the invalid iterator 's.end()' might take place.
General Analysis	WriteHelpers.h:200	Low	V560	A part of conditional expression is always true: 0x00 <= c. Unsigned type value is always >= 0.
General Analysis	WriteHelpers.h:210	Low	V560	A part of conditional expression is always true: 0 <= lower_half. Unsigned type value is always >= 0.
General Analysis	HashTable.h:220	Medium	V730	Not all members of a class are initialized inside the compiler generated constructor. Consider inspecting: zero_value_storage.
General Analysis	HashTable.h:456	Medium	V730	Not all members of a class are initialized inside the constructor. Consider inspecting: size.
General Analysis	HashTable.h:510	Medium	V730	Not all members of a class are initialized inside the constructor. Consider inspecting: container, ptr.

Figure 5.5: Main page of an HTML-report.

It is possible to not include files from the analysis by name, folder or mask; to run the analysis on the files modified during the last N days [30]. PVS-Studio has an integration with open source platform SonarQube. This platform was designed for uninterrupted analysis and evaluation of the quality of code.

Chapter 6

Conclusion

The main goal of this thesis, as stated at the beginning of this work, is to analyze the quality of programs written in the Erlang programing language by using RefactorErl tool. To accomplish that goal, it became necessary to provide a general overview of Erlang, RefactorErl static analysis tool and developed metrics. In this thesis, the software quality, software metrics and some of the tools for measuring software quality metrics have been studied and analyzed.

Erlang is a functional language designed for highly parallel, scalable applications requiring high uptime. RefactorErl is a static source code analysis and transformation tool for Erlang providing several software metrics. The tool is developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. Among the features of RefactorErl is included a metric query language which can support Erlang developers in everyday tasks such as program comprehension, debugging, finding relationships among program parts, etc.

In this thesis, we presented a developed framework which analyzes git repositories with Erlang code files. The new component is built on RefactorErl static analysis tool and actively uses its feature of calculating different metrics of Erlang modules and functions. This framework allows drawing plots which show change of metrics with software evolving from version to version. The plots can be saved for future usage as a pictures in PNG format.

The main focus of the component is to help Erlang developers with analyzing their projects using plots. Also, it was important to test the developed component on some projects and after that to analyze the measurements. For this purpose have been chosen three different projects from git. The experimental results allow finding changes (increasing the line of code number, char of code number, using otp library and etc.) in code. Visualisation helps with finding patterns and improving of the code quality.

It is safe to say that the main goals of this thesis were successfully achieved. The usage of software metrics is within an organization and its usage is expected to have a beneficial effect on software organizations by making software quality more visible.

Bibliography

[1] Léo Sartre Antoine Varet, Nicolas Larrieu. Metrix: a new tool to evaluate the quality of software source codes. AIAA Infotech@Aerospace Conference, 2013.

[2] Nenad Medvidovic André van der Hoek, Ebru Dincel. Using service utilization metrics to assess and improve product line architectures. Proceedings of the 9th International Symposium on Software Metrics, 2003.

[3] C Symons. Management : measure for measure's sake. Computer Weekly, pp. 16, July 1992.

[4] S.R. Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. EEE Transactions on Software Engineering 20,6(1994), 1992.

[5] Simon Thompson Zhang Qingqing. Complexity metrics for service-oriented systems. Second International Symposium on Knowledge Acquisition and Modeling, 2009.

[6] Martin Neil Norman E Fenton. Software metrics: Roadmap. The Future of Software Engineering. ACM Press, New York, 2000.

[7] M. Shepperd. A critique of cyclomatic complexity as a software metric. Software Engineering Journal, pp. 30 - 36, March 1988.

[8] T.J. McCabe. A complexity measure. IEEE Transactions on Software Engineering, 2(4), 1976.

[9] LI Qing-shan WANG Yu-ying. Dynamic fan-in and fan-out metrics for program comprehension. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, August 2009.

[10] Maryam Hooshyar Habib Izadkhah. Class cohesion metrics for software engineering: A critical review. Computer Science Journal of Moldova, vol. 25, no.1(73), 2017.

[11] R. S. Pressman. Software engineering: a practitioner's approach. Palgrave Macmillan, 2005.

[12] C and c++ code counter. `http://sourceforge.net/projects/cccc.`

[13] A tool for calculating chidamber and kemerer java metrics. `https://www.spinellis.gr/sw/ckjm/doc/indexw.html.`

[14] Analyst4j. `http://www.codeswat.com.`

[15] S.M. Khan J.S. Alghamdi, R.A. Rufai. Oometer: A software quality assurance tool. Ninth European Conference on Software Maintenance and Reengineering, March 2005.

[16] Oometer software metrics tool. www.ccse.kfupm.edu.sa/~oometer/oometer.

[17] Eclipse metrics plugin 1.3.6. http://www.easyeclipse.org/site/plugins/metrics.html.

[18] Eclipse metrics plugin 3.4. http://eclipse-metrics.sourceforge.net/.

[19] Semmle. https://semmle.com/.

[20] Dr.K.Krishnamoorthy V.Thangadurai, Dr.K.P.Yadav. A study of software functional programming and measurement. International Journal of Innovative Research in Science, Engineering and Technology, ISSN: 2319-8753, November 2012.

[21] Robert Kitlei Roland Kiraly. Application of complexity metrics in functional languages. STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume LV, Number 1, 2010.

[22] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[23] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[24] Refactorerl homepage. http://plc.inf.elte.hu/erlang/.

[25] Metric queries. http://pnyf.inf.elte.hu/trac/refactorerl/wiki/MetricQuery.

[26] Li Xinke Francesco Cesarini. Erlang programming. Journal of Shanghai University (English Edition), Volume 11, Issue 5, pp 474–479, Oknober 2007.

[27] Serge Demeyer Quinten David Soetens. Studying the effect of refactorings:a complexity metrics perspective. 2010 Seventh International Conference on the Quality of Information and Communications Technology, December 2010.

[28] Paul E. Black. Dictionary of algorithms and data structures. https://xlinux.nist.gov/dads/, August 2008.

[29] Jonathan Guerrero Victor Winter, Carl Reinke. Sextant: A tool to specify and visualize software metrics for java source-code. International Workshop on Emerging Trends in Software Metric, 2013.

[30] Pvs-studio analyzer. https://www.viva64.com/en/pvs-studio/.