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

This page contains the text of your abstract.

Contents

Chapter 1

Introduction

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.

In this thesis, we introduce a module for analysing of quality of programs written in the Erlang programing language. Our goal is to analyse projects and then prepare the results of the analysis, so it can be added to the RefactorErl static analysis framework for Erlang.

Chapter 2

Software metrics

This chapter presents the introduction to the software metrics as the key point for evaluation of productivity and quality of the software development product. There is also classification of the software metrics, description the different measurements of software metrics in general and the most popular software metrics tools.

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 software characteristics which are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance, planning work items, 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.

Within the software development process, there are many metrics that are all related to each other. Metrics are related to the four functions of management:

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

The goal of tracking and analyzing software metrics is to determine the quality of the current product or process, improve that quality and predict the quality once the software development project is complete. On a more granular level, software development managers are trying to:

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

These goals can be achieved by providing information and clarity throughout the organization about complex software development projects. Metrics are an important component of quality assurance, management, debugging, performance, and estimating costs,and they're valuable for both developers and development team leaders:

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

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

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 are measures of software project and are used to monitor and control the project. These metrics measures the complexity of the software design size of the final program number of pages of documentation produced. They enable a software project manager to:

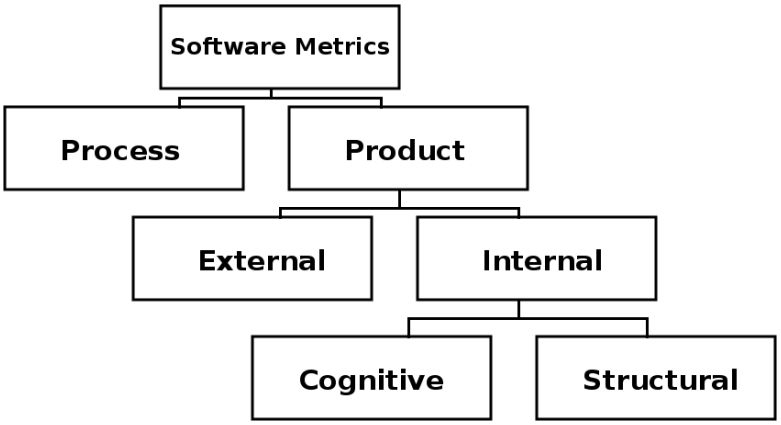

Figure 2.1: Classification of software metrics.

- minimize the development time by making the adjustments necessary to avoid delays and potential problems and risks.
- assess product quality on an ongoing basis and modify the technical approach to improve quality.

Product metrics are measures of the software product of any stage of its development,from requirements to installed system. Product metrics may measure: the complexity of the software design, the size of the final program, the number of pages of documentation produsedd. 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 trus" Hence,software metrics have been used to measure a wide range of software engineering 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 known as Source Lines of Code - SLOC) is a metric generally used to evaluate a software program or codebase according to its size. It shows how many lines of source code there is in the application, namespace, class or method. LoC can be used to: check the size of code units and estimate the size of project. LOC is the popular and simplest one. There are two major types of SLOC measures:

Physical SLOC (LOC) Physical SLOC is a count of lines in the text of the program's source code including comment.

Logical SLOC(LLOC) Logical LOC attempts to measure 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 (OO)

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

Number Of Methods (NOM)

The Number Of Methods metric is used to calculate 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 is a number of immediate subclasses subordinated to a class in the class hierarchy. NOC counts the number of subclasses belonging to a class.

NOC measures the breadth of a class hierarchy, where maximum DIT measures the depth. Depth is generally better than breadth, since it promotes reuse of methods through inheritance. NOC and DIT are closely related. Inheritance levels can be added to increase the depth and reduce the breadth.

A high NOC, a large number of child classes, can indicate several things:

- High reuse of base class. Inheritance is a form of reuse.
- Base class may require more testing.
- Improper abstraction of the parent class.
- Misuse of sub-classing. In such a case, it may be necessary to group related classes and introduce another level of inheritance.

A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design. A redesign may be required.

Not all classes should have the same number of sub-classes. Classes higher up in the hierarchy should have more sub-classes then those lower down.

Weighted Methods per Class (WMC)

This metric is the sum of complexities of methods defined in a class. It therefore represents the complexity of a class as a whole and this measure can be used to indicate the development and maintenance effort for the class. Classes with a large Weighted Methods Per Class value can often be refactored into two or more classes.

Coupling Between Object classes (CBO)

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

- Method call.
- Class extends.

- Properties or parameters.
- Method arguments or return types.
- Variables in methods

Coupling between classes is required for a system to do useful work, but excessive coupling makes the system more difficult to maitain and reuse.At project or package level, this metric provides the average number of classes used per class.

Depth of Inheritance Tree (DIT)

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

The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. Inheritance is a tool to manage complexity, really, not to not increase it. As a positive factor, deep trees promote reuse because of method inheritance.

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

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior
- Deeper trees constitute greater design complexity, since more methods and classes are involved
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods

Response For a Class (RFC) The Response for Class (RFC) metric is the total number of methods that can potentially be executed in response to a message received by an object of a class. This number is the sum of the methods of the class, and all distinct methods are invoked directly within the class methods. Additionally, inherited methods are counted, but overridden methods are not, because only one method of a particular signature will always be available to an object of a given class.

A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated. A worst case value for possible responses will assist in appropriate allocation of testing time.

2.3.3 Complexity metrics

Complexity is an important aspect for software quality assessment and must be appropriately addressed in service-oriented architecture[3]. Cyclomatic complexity one of the most difficult software metrics for understanding.

McCabe's Cyclomatic Complexity (MVG)

McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring 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's a structural metrics which measures inter-module complexities.

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

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

Fan-in and fan-out metrics reflect structure dependency [4]. This structural metrics were first defined by Henry. This metrics can be applied both at module level and function level. This metrics just puts 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 restriction on number of Fan-Out a function has to avoid cluttering your function but is not a widely accepted practice.

2.3.5 Cohesion metrics

Cohesion is an important software quality attribute and high cohesion is one of characteristics of well-structured software design [5]. Cohesion metrics analyze the connection between methods of a class. Module cohesion indicates relatedness in the functionality of a software module [6].

Lack of Cohesion in Methods (LCOM)

LCOM measures the amount of cohesiveness present, how well a system has been 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:

- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.

- Lack of cohesion implies classes should probably be split into two or more subclasses.
- Any measure of disparateness of methods helps identify flaws in the design of classes.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

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 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.

2.4 Software Metric Tools

There are number of software metrics have been developed and numerous tools exist to collect the metrics from program representations. This large variety of tools allows a user to select the tool best suited as per the use requirements for example it's handling, tool support ,cost etc. This is assumed that the metrics computed by the metric tools are 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 to access a software system according to the metrics by extracting the required entities from the software ad 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 follows language: Java(source or byte code), metrics: well known object oriented metrics on class level, license: freely available.

2.4.1 CCCC

It is a little command-line tool that generates metrics from the source code of a C or C++ project. The output of the tool is a simple HTML website with information about all your

sources. It generates reports on various metrics including lines of code (LOC) and metrics proposed by Chidamber&Kemererand Henry&Kafura. CCCC has been developed as freeware, and is released in source code form. Users are encouraged to compile the program themselves, and to modify the source to reflect their preferences and interests.

CCCC will process each of the files specified on the command line (using standard wildcard processing were appropriate. For each file, named, CCCC will examine the extension of the filename, and if the extension is recognized as indicating a supported language, the appropriate parser will run on the file. As each file is parsed, recognition of certain constructs will cause records to be written into an internal database. When all files have been processed, a report on the contents of the internal database will be generated in HTML format. By default the main summary HTML report is generated to the file cccc.htm in a subdirectory called .cccc of the the current working directory, with detailed reports on each module (i.e. C++ or Java class) identified by the analysis run).

In addition to the summary and detailed HTML reports, the run will cause generation of corresponding summary and detailed reports in XML format, and a further file called cccc.db to be created. cccc.db will contain a dump of the internal database of the program in a format delimited with the character '@' (chosen because it is one of the few characters which cannot legally appear in C/C++ non -comment source code).

The report contains a number of tables identifying the modules in the files submitted and covering:

1. Measures of the procedural volume and complexity of each module and its functions;.
2. Measures of the number and type of the relationships each module is a party to either as a client or a supplier.
3. Identification of any parts of the source code submitted which the program failed to parse.
4. A summary report over the whole body of code processed of the measures identified above.

This tool can measure the following metrics:

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

2.4.2 Chidamber&Kemerer

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

This tool can measure the following metrics:

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

2.4.3 Analyst4j

It is based on the Eclipse platform and available as a standalone Rich Client Applications or as an Eclipse IDE plug in. It features search, metrics,analyzing quality,report generating for Java programming. Analyst4j tool are most popular to findout the quality related metrics. This tool is based on Chidamber&Kemerer metrics.

This tool can measure the following metrics:

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

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. OOMeter has a rich collection of object-oriented software metrics. It is an Eclipse plug in. It provides an SQL like querying language for object-oriented code which allows to search for bugs, measure code metrics 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.

This tool can measure the following metrics:

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

2.4.5 Eclipse Metrics plugin 1.3.6

This is an open source metrics calculation and dependency analyzer a metrics plugin for Eclipse IDE. 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:

- Lines of Code (LOC).
- Weighted Methods per Class(WMC).
- Depth of Inheritance Tree (DIT).
- Numbe r Of Children(NOC).
- 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 ID.E It is a open source tool. It calculates various metrics during build cycles and warns via the problem view of metrics range violations.

This tool can measure the following metrics:

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

2.4.7 Semmle

Semmle is an analysis platform that produces detailed analyses of the code base for one or more software projects. For each project that it analyzes, it measures artifacts against rules that check for good practice. Analysis is scheduled to occur on a regular basis. As part of this process a copy of the source code is checked out of the repository for analysis. The code, and related artifacts, is checked against rules, defined using queries, to identify any alerts. In addition, metrics are calculated and data may be imported from third-party systems used by your company. A database is created, containing detailed information about the artifacts and each alert.

This tool can measure the following metrics:

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

2.5 Measuring functional languages

In previous section has been described sofrware 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. 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 [7].

Dissimilarity between functional and imperative languages are in such features as 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 insted of cohesion and 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 be- come applicable to functional languages [7].

There are general metrics are acceptable for functional languages:

- **Branches of recursion**.This metric allows to measure 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 behaviours.

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

Chapter 3

Sofrware metrics in erlang

In this chapter we give a brief summary of Erlang and RefactorErl tool.

3.1 An introductory glimpse at the Erlang programming language

Erlang is a functional language and accompanying runtime designed for highly parallel, scalable applications requiring high uptime. Erlang is a programming language designed for developing robust systems of programs that can be distributed among different computers in a network.Erlang is similar to Java in that it uses a virtual machine and 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 begin 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, enclosed in double quotes: "This is a string.".
4. Atoms. An atom stands for itself. It begins with a lowercase letter and is composed of letters, digits, and underscores, or it is any string enclosed in single quotes: atom1, 'Atom 2'.
5. Lists, which are a comma-separated sequence of values enclosed in brackets: [abc, 123,"pigs in a tree"].

6. Tuples, which are a comma-separated sequence of values enclosed in braces: abc, 123,"pigs in a tree".
7. Records, which 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, enclosed in double angle brackets: «0, 255, 128, 128», «"hello"», «X:3,Y:7, Z:6». Binaries are sequences of bits; the number of bits in a binary must be a multiple of 8.
9. References are globally unique values.
10. Process identifiers (Pids) are the "names" of processes.

Figure 3.1features 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 framework

RefactorErl [8, 9] 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 programstructure you do not alter its behaviour. RefactorErl was built to refactor Erlang programs.

The main focus of RefactorErl is to support daily code comprehension tasks of Erlang developers [10]. It can analyse 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 [10] . 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 coloms 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 modules functions. The proper metrics adjusted in a list can be implemented in the desired number and order [**?**].

line_of_code

The number of the 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 [**?**].

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 average number of characters in a cluster, or in the whole source text [**?**].

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 [**?**].

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 afte comma	module/function
is tail recursive	function
mcCabe	module/function
otp used	module

number_of_macros

This metric gives the number of defined macros in the concrete module, or modules. It is also possible to enquire the number of implemented macros in a module [**?**].

number_of_records

This metric gives the number of defined records in a module. It is also possible to enquire the number of implemented records in a module [**?**].

included_files

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

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) [**?**].

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 [**?**].

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 [**?**].

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 [**?**].

cohesion

The number of call-paths of functions that call eachother. 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 [**?**].

function_sum

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

max_depth_of_calling

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

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 embedd [**?**].

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 [**?**].

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 [**?**].

number_of_funclauses

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

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 [**?**].

calls_for_function

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

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) [**?**].

number_of_funexpr

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

number_of_messpass

In 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 [**?**].

fun_return_points

The metric gives the number of the functions possible return points (or the functions of the given module) [**?**].

average_size

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

max_length_of_line

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

average_length_of_line

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

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 [**?**].

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 semanctic query language, the result is converted to tail_rec, non_tail_rec or non_rec atom [**?**].

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 [**?**].

otp_used

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

3.3 Metric visualisation module for WEB2 interface of RefactorErl

The section describes the main concept and details of the developed module for RefactorErl static analyzis 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 to analyze git repositories with Erlang code files. The component is built on RefactorErl static analyzis tool and actively uses its feature of calculating different metrics of Erlang modules and functions. It has very convinient to use 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 plots can be saved for future usage as a pictures in PNG format.

The main feautures of the software are:

- Drawing plots which show 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. Visualisation helps with finding patterns and improving of 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 a pictures the saveSvgAsPng library is used.

NVD3

Inspired by Mike Bostock and currently maintained by a team of frontend software engineers at Novus Partners, NVD3 is another quality D3 based JavaScript library.Allowing you to create beautiful reusable charts in your web applications.

It has great features for visualizing data with lovely charts such as the box-plot, sunburst and candlestick charts. If you are looking for tons of functionality in a JavaScript charting library, NVD3 is the one to look out for.

AngularJs

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

AngularJS (also written as Angular.js) is a JavaScript-based open-source front-end web application framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications.

The AngularJS framework works by first reading the Hypertext Markup Language (HTML) page, which has additional custom tag attributes embedded into it. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved 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 [11].

JSON

Data which stored in Dets tables transforms into JSON objects when Erlang code interact with JavaScript code. JSON (JavaScript Object Notation) is a lightweight data-interchange format. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

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

saveSvgAsPng

This small library is used for saving SVG plots as a 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 metrics tab as shown on Figure 3.2

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

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

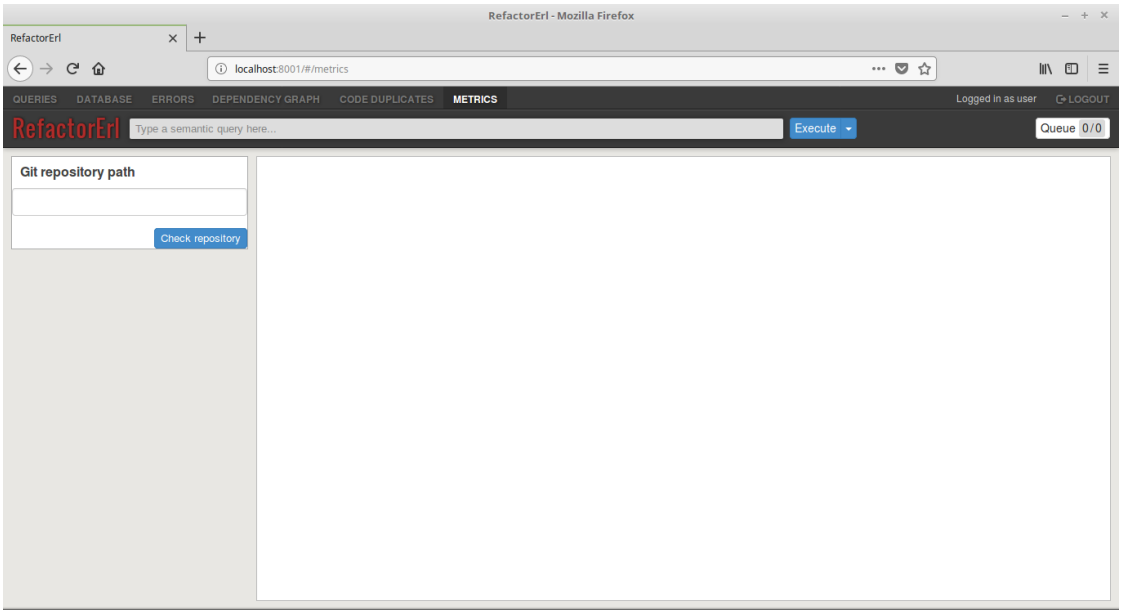

Figure 3.2: Component interface.

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

After analyzis is done the menu with choosing parameters of drawing the plot will be available as shown at Figure 3.6.

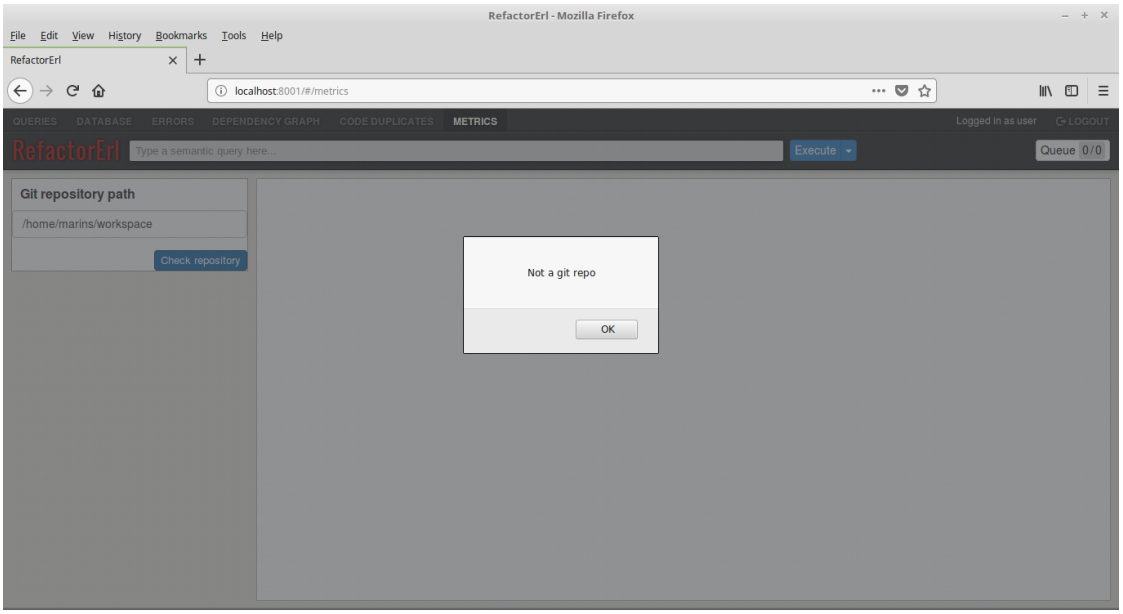

Figure 3.3: Alert shown because the provided folder is not a valid repository.

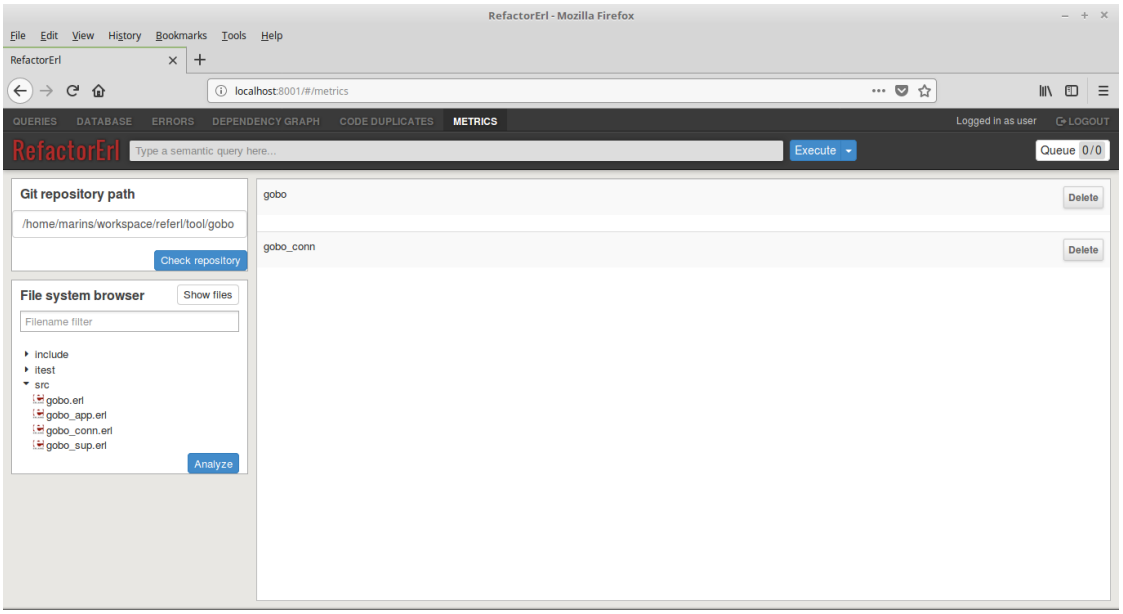

Figure 3.4: Repository tree.

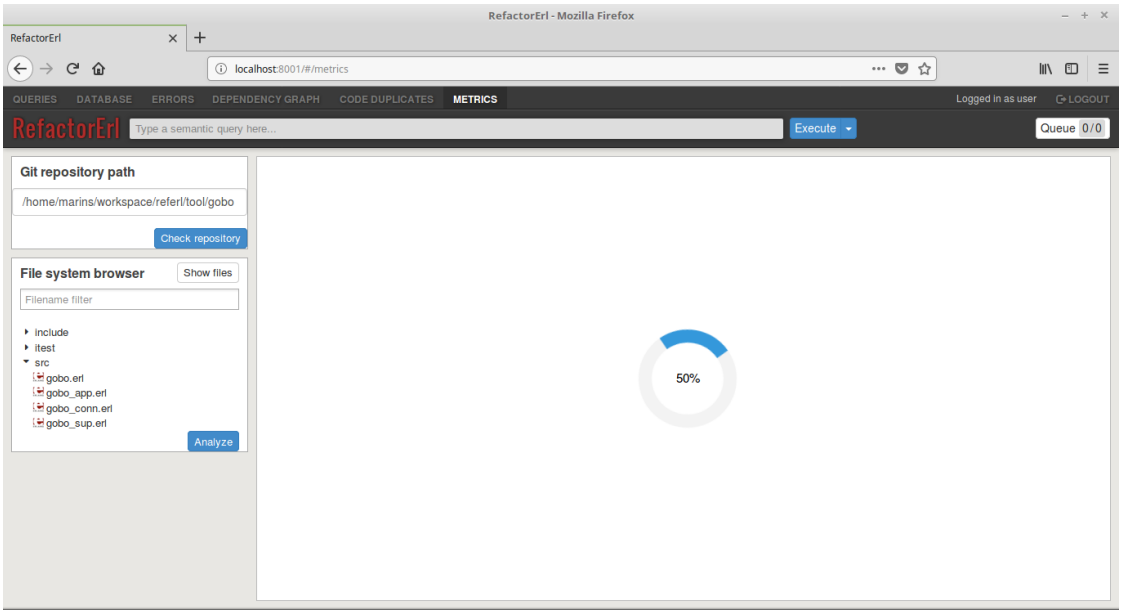

Figure 3.5: The process of repository analyzing.

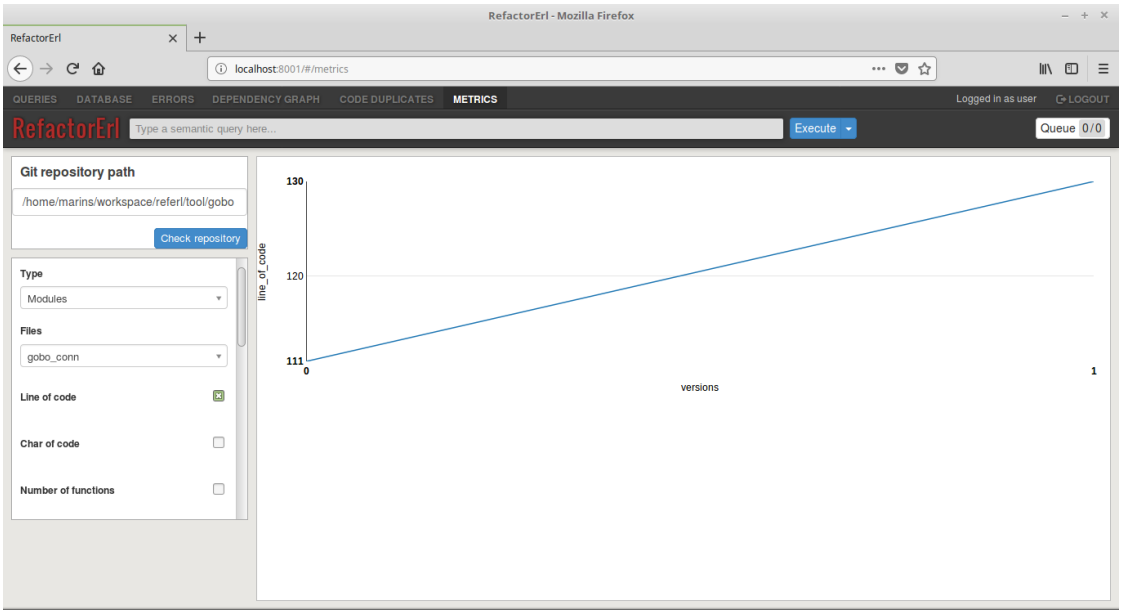

Figure 3.6: The example of plot.

Chapter 4

Measurements and findings

The aim of this chapter is to test and analyze projects from git by using developed module for RefactorErl. The 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 custom predicate.
- Find with coerce equality, find with custom predicate.

This project has just only one source code file with 68 commits.

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 on Figure 4.2.

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

Average length of line was not stabilize until arround 35th version with gradually decreasing from 50 symbols to 38 symbols.

As we can see on Figure 4.5 and Figure 4.6 there are not defined macros and record in the whole iron project.

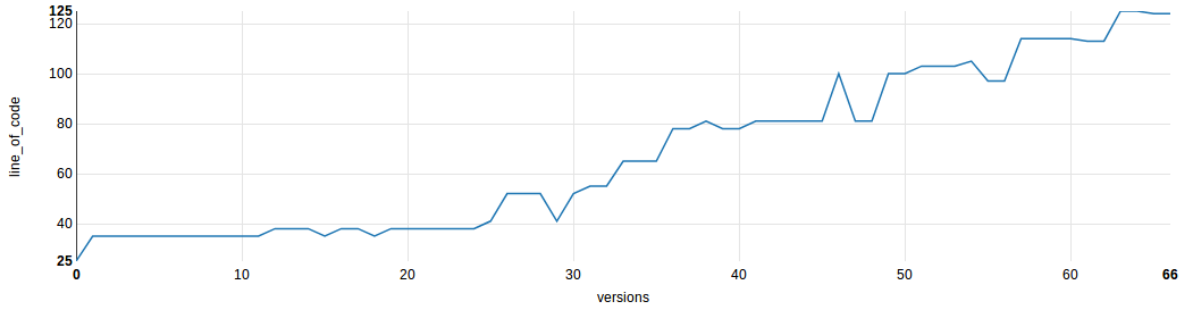

Figure 4.1: Effective Line of code for fe.erl file.

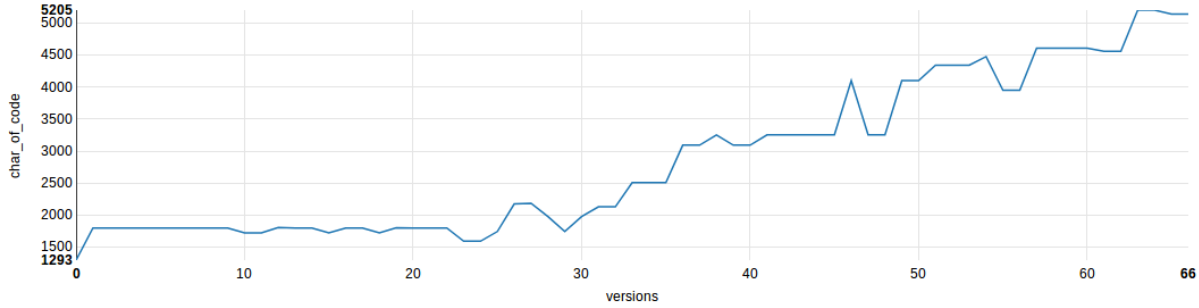

Figure 4.2: Char of code for fe.erl file.

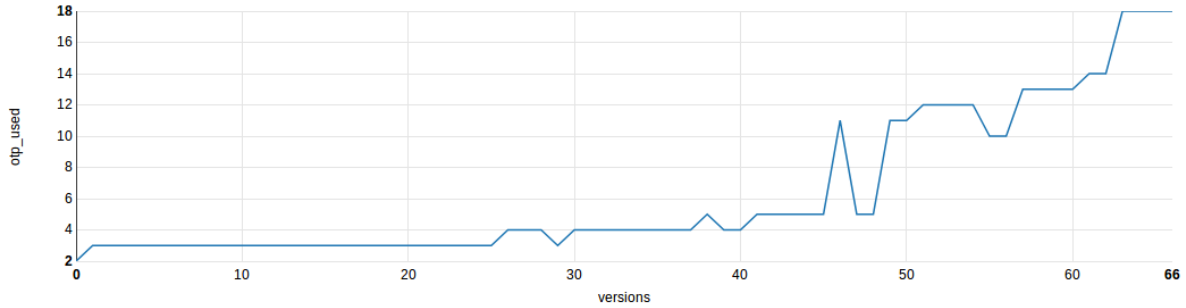

Figure 4.3: Otp used for fe.erl file.

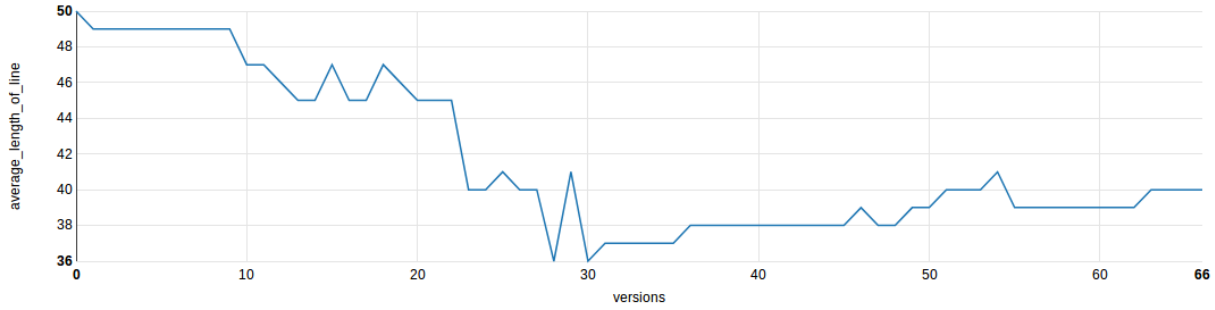

Figure 4.4: Average length of line for fe.erl file.

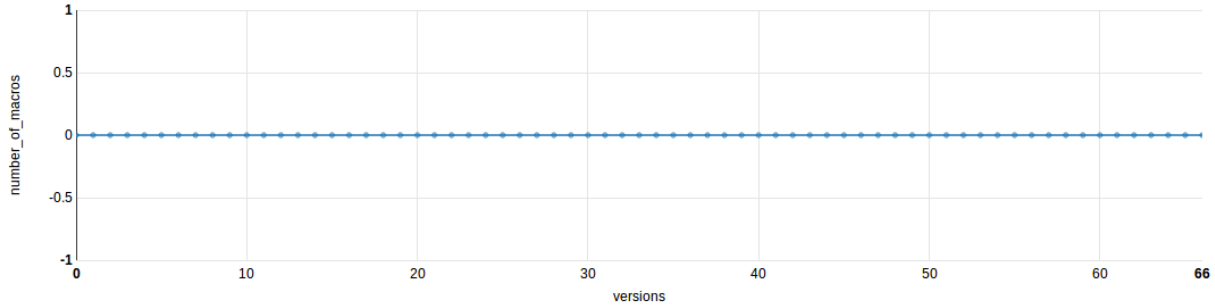

Figure 4.5: Number of macros for fe.erl file.

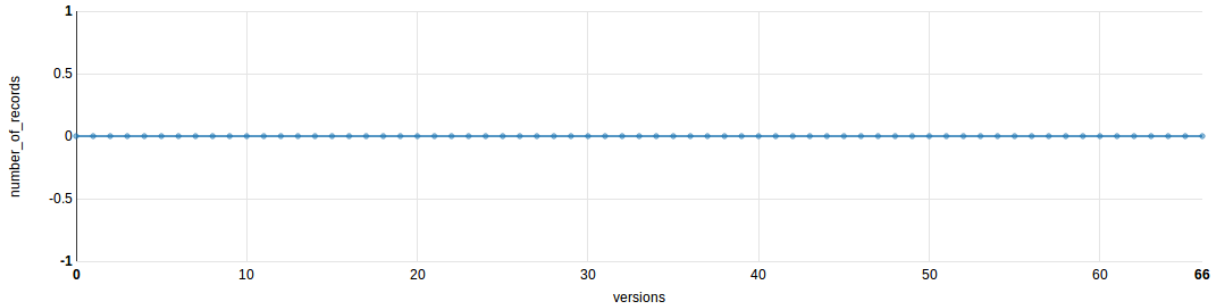

Figure 4.6: Number of records for fe.erl file.

4.2 Erlang chat

This project is multi user chat written in Erlang. It has nine source code files and 45 commits.

Chapter 5

Related work

Nowadays,the need of the visualization of software quality metrics have 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 analyzis.

5.1 Open Source tool "METRIX"

This tool can 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].

Different diagrams enable the user to visualize numeric data. Graphics like line charts, scatter plots and histograms are common. There are two less-common classes of diagrams:

- The radar plots, also called Kiviat diagrams.
- The city map diagram, mostly used to represent the cartography of cities.

One specific feature of METRIX is to use those two types of visualization for constructing signature and cartography ofsource codes.

Kiviat diagrams The Kiviat diagram visualizes information through polar coordinates. The distance between the point and the origin is associated to the value user want to represent, while the angle between two points is constant, this constant is uninformative and calculated to uniformly distribute the different points. Moreover, all metric points are linked together, making

a plain polygon, which shapes the specificity of the data that user wants to represent. Of course, this diagram is not adequate to represent only one or two metrics, it requires at least three values to be pertinent. Values may be of the same metrics, representing the metric measured on different parts of the source code. Values may be of very different metrics, computed with heterogeneous units. That enables the user to merge multivariate data on the same diagram. On Kiviat diagrams, values must be strictly positive.

In figure5.1 shows an example of two Kiviat diagrams. This diagrams represent coding different metric values for two functions.

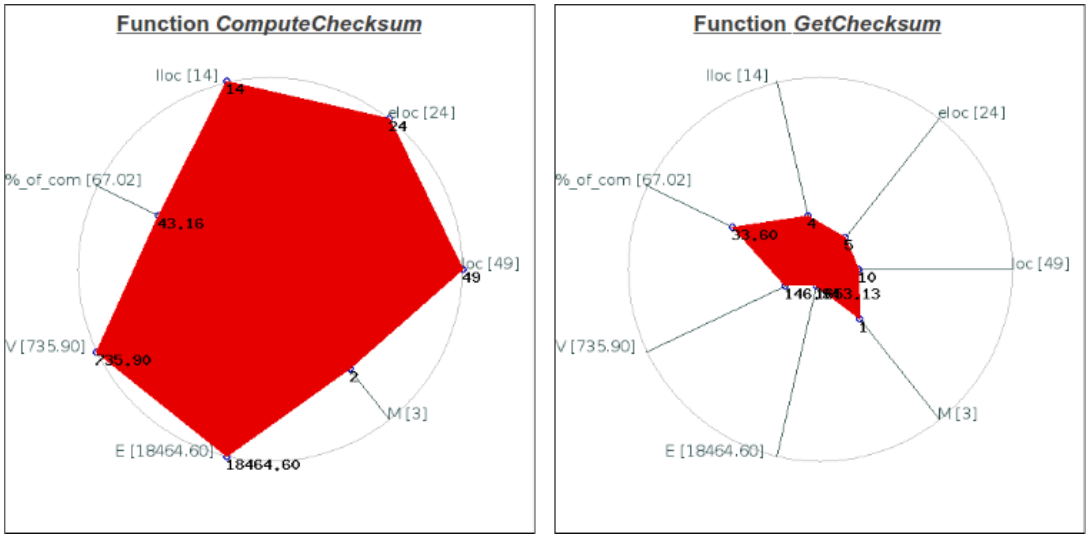

Figure 5.1: An example of Kiviat diagrams.

City map diagrams This diagram is mainly used in urbanism, to represent cities and their buildings in three dimensions. As a software project can contain a very large number of source files, code functions and code variables, but this visualization diagram adequately suits for complexity visualization issue.

The treemap diagram is used in computing to represent in two dimensions source code metrics with rectangles. 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 a n–tuple of numeric values.

This tool also has a graphical user interface with a single Window containing three tabs.

The first tab called "Calc" and enables the user to indicate the source file(s) to evaluate and to parameterize the metrics to compute. It avoids using hand-start scripts with a prompt, but the open source aspect of our tool allows advanced users to (re)use our measuring scripts manually and/or to integrate them into other software.

The second tab called "Csv" presents the numeric values as a list, with a tab per function and per file. The user may export these values to a spreadsheet application, in order to represent the values with common visualization graphs, like scatter plots, line plots, etc.

The third tab called “Plots” and provides a way to visualize the city map diagrams for the source code, the signatures of functions and the comparisons between functions through radar/Kiviat plots. This tab enables the user to change the default behavior of the tool, for example to modify the ceiling and floor to insert color into the data, to adapt the placement of the buildings in the city map, etc.

This tool may generate a report in the form of a LaTeX file, so the user can use it to produce a PDF file. This report summarizes all the values measured on the project. Each function and each file make a section of this report. In each section, numerical values are coupled with the signature of the function (as a Kiviat diagram). 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.

5.2 Sextant

Sextant is a Java source-code analysis tool under development at the University of Nebraska at Omaha (UNO) [12]. Sextant represents a non-trivial extension of the TL system (a general-purpose program transformation system) specialized to the domain of the Java programming language.

One of the primary design goals of Sextant is to provide a tool facilitating specification and visualization of custom analyzis rules (e.g.,domain specific or even application specific analyzis rules).

Sextant analysis rules are based on information drawn from several software models. There are two models of central importance. The first model, a syntactic model,is the parse tree of the source code. Parse trees correspond to compilation units (i.e., Java files) and are generated using GLR parser technology provided by the TL system. Parse trees are well-suited to analysis and manipulation through standard primitives provided by program transformation systems such as matching and generic traversal.

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

Information in the CAG is made accessible to Sextant’s transformation-based analysis rules through two mechanisms.The first mechanism is a positional system that establishes a relation between parse (sub)trees and corresponding contexts within the CAG. It is this relation that makes it possible to correctly resolve references to types, fields, local variables, methods, and constructors during the course of generic traversals which are a key enabling mechanism within program transformation systems.

The second mechanism is a library of semantic queries which, thanks to the positioning system, can be accessed during the course of transformation. Functionality presently provided by this library includes things like:

- Resolving a reference.
- Determining the type of a reference.
- Determining whether one type is a subtype of another.
- Determining whether one declaration shadows or overrides.

Sextant provides table and set datatypes for collecting information associated with analysis rules. These constructs are suitable for storing information related to a wide variety of custom metrics.

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.

Sextant can generate software models which can be visualized using third party tools such as Cytoscape, TreeMap, and GraphViz. Sextant can output the CAG of a targeted code base in a JavaScript JSON format. This dot-JSON file can then be loaded into Cytoscape, an open source platform providing extensive and sophisticated capabilities for visualizing large complex networks (i.e., graph structures). Similarly, metrics derived from tables and sets can be output in CSV format and viewed using TreeMap [3], and parse trees can be output as dot-files and subsequently viewed via GraphViz.

The view in Figure5.2 shows an example of represents a coloring of dependencies on the unsupported features.Purple nodes have direct dependencies on unsupported features while nodes colored orange have indirect dependencies on unsupported features.

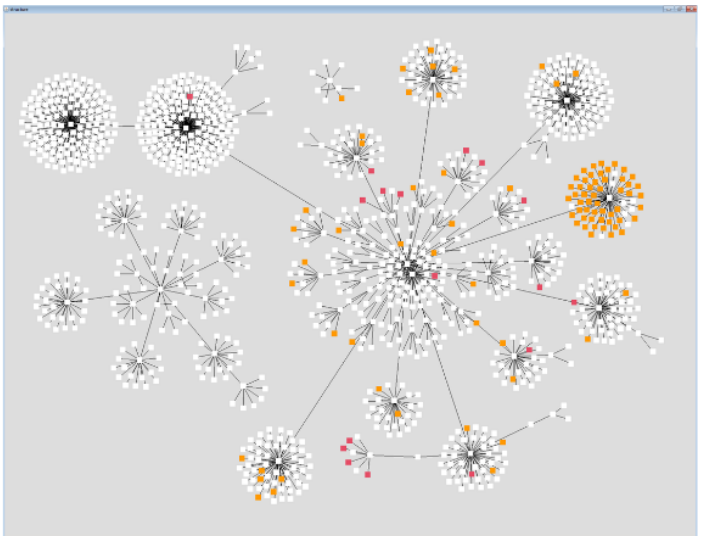

Figure 5.2: An example of using Sextant tool.

5.3 NDepend

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

NDepend computes a lot of 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 measures the complexity of a type or a method by calculating the number of branching points in code. NDepend has two metrics for cohesion. Relational Cohesion is an assembly level metric that measure the average number of internal relationships per type. Lack of Cohesion Of Methods (LCOM) measures the cohesiveness of a type. A type is maximally cohesive if all methods use all instance fields.

NDepend uses a visualization tool called a Treemap. NDepend comes with a dashboard to quickly visualize all application metrics s shown in Figure5.3. The dashboard is available both in the Visual Studio extension.For each metric, the dashboard shows the diff since baseline. It also shows if the metric value gets better (in green) or wort (in red).

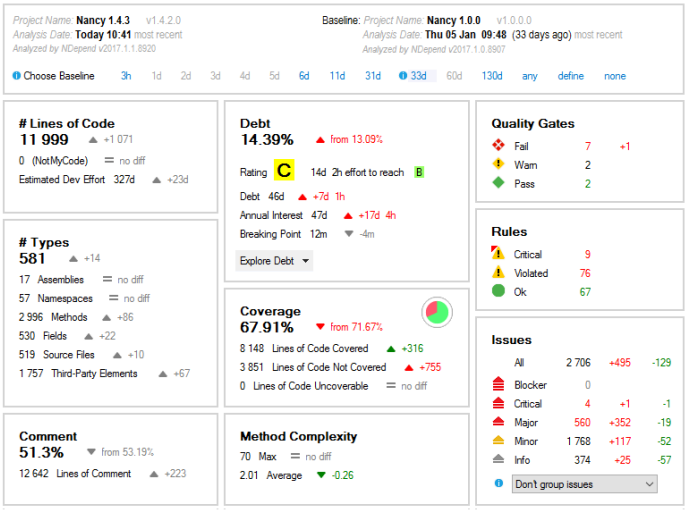

Figure 5.3: An example of using the dashboard.

The Figure5.4 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 contains methods and fields.

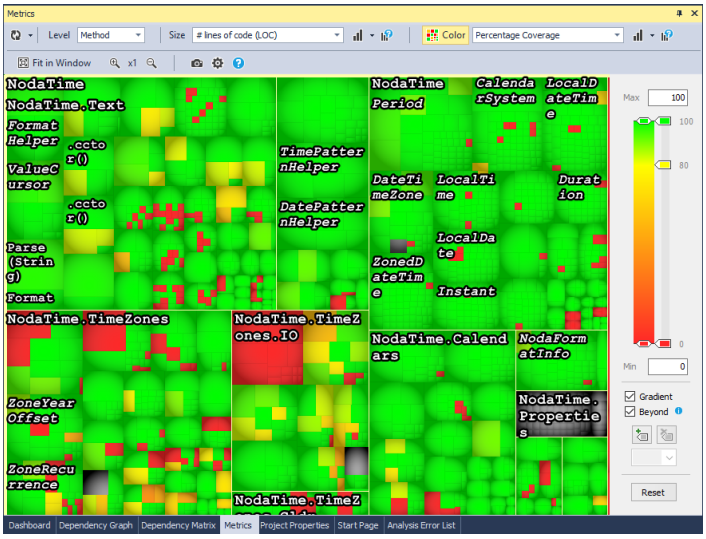

Figure 5.4: An example of using the colored treemap.

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 [13].

This tool executes static code analysis and after that createsa report for helping a developers to find and fix bugs. PVS-Studio has a wide range check methods. It helps to find misprints and Copy-Paste errors. The main value of static analysis is in its regular use, so that errors are identified and fixed at the earliest stages [13].

PVS-Studio runs from the command line. The analysis results can be save as HTML with full source code navigation. 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. Error statistics can be viewed in Excel [13].

This tool has an online reference guide concerning all the diagnostics available in the program, on the web site and documentation.

PVS-Studio ha an integration with open source platform SonarQube designed for continuous analysis and measurement of code quality.

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

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

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

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

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