FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

An Empirical Investigation of Source Code Metrics and FindBugs Warnings

André Amarante dos Santos Cunha

Master in Informatics and Computing Engineering

Supervisor: Rui Maranhão (PhD)

Second Supervisor: Martin Pinzger (PhD)

28th June, 2010

An Empirical Investigation of Source Code Metrics and FindBugs Warnings

André Amarante dos Santos Cunha

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

31st July, 2010

Abstract

Static software checking tools are useful as an additional automated software inspection step that can easily be integrated in the development cycle and assist in creating secure, reliable and high quality code.

However, an often quoted disadvantage of these tools is that they generate an overly large number of warnings, including many false positives (warnings that do not indicate real bugs) due to the approximate analysis techniques. Therefore, programmers have to spend a considerable amount of time on screening out real bugs from a large number of reported warnings, which is time-consuming and inefficient. This information overload can easily hinder the potential benefits of such tools. The tools' warning prioritization is little help in focusing on important warnings: the maximum possible precision by selecting high priority warning instances.

In this thesis project, we conducted an empirical investigation on the issues detected by a popular bug-finding tool, FindBugs, on 2 open-source Java projects, Tomcat and Axis. We investigate the correlation between source code metrics and warning categories as used by the Java FindBugs tool. This lays the basics for developing a more advanced model for ranking warnings to aid developers in identifying and fixing most severe warnings first.

Resumo

As ferramentas de verificação estática de software são úteis como um passo complementar à inspecção automática de software que pode ser facilmente integrado no ciclo de desenvolvimento e ajudar a produzir código seguro, fidedigno e de alta qualidade.

No entanto, uma desvantagem frequentemente mencionada destas ferramentas é que geram uma grande quantidade de *warnings*, incluindo vários falsos positivos (*warnings* que não indicam *bugs* reais) devido à imprecisão das técnicas de análise. Acontece, portanto, que os programadores terão de perder muito tempo a identificar quais são os *bugs* reais a partir de um longo relatório de *warnings*. Esta informação excessiva pode dificultar os potenciais benefícios deste tipo de ferramentas. O seu sistema de prioridades é uma pequena ajuda para os programadores se focarem nos *warnings* mais importantes: a máxima precisão, ao seleccionar os *warnings* de maior prioridade.

Nesta tese, conduzimos uma investigação empírica sobre os *warnings* detectados por uma ferramenta de localização de *bugs* popular, FindBugs, em 2 projectos Java *open-source*, Tomcat e Axis. Investigamos as correlações entre métricas do código fonte e categorias dos *warnings* como usadas pela ferramenta Java FindBugs. Isto estabelece uma base para desenvolver um modelo mais avançado para priorizar *warnings* de forma a ajudar os programadores a identificar e corrigir os *warnings* mais importantes em primeiro lugar.

Acknowledgements

I would like to take this opportunity to thank my supervisors Rui and Martin for helping and guiding me during this process.

I would also like to say thanks to all of the friends I have met in here and i hope they will always remain like that.

One special thanks to my parents, Isabel and José, for providing me this chance which was probably the best experience of my life. I am also grateful of my aunt Teresa and my grandparents Fernando and Celina for all of their concern relating my well being.

Finally, my best regards to my brother Rui Pedro who has always supported me and to all of my friends in Portugal, specially to my friend Marta for motivating and helping me in every way as possible.

André Amarante dos Santos Cunha
Delft, the Netherlands
June 28, 2010

Contents

List of Figures

List of Tables

LIST OF TABLES

Abbreviations

CSV	Comma-separated values
IDE	Integrated development environment
JRE	Java Runtime Environment
JDK	Java Development Kit
OO	Object-oriented
PASW	Predictive Analytics SoftWare
SCM	Software configuration management
SOAP	Simple Object Access Protocol
SVN	Subversion
W3C	World Wide Web Consortium
WWW	*World Wide Web*
XML	Extensible Markup Language

ABBREVIATIONS

Chapter 1

Introduction

This chapter introduces all aspects that form the context, motivations and goals of this thesis project. We also describe the main terms and the background used throughout this dissertation. We present the problem statements as well as an overview of contributions. Finally, it is presented the outline used in this document.

1.1 Context

This thesis research is inserted in Software Engineering group, more properly, the Software Testing and Quality area. This work is a contribution to the field of automated inspection tools warning's prioritizing. The full process of researching, results analysis and report writing was carried out in the Department of Software Technology - Software Engineering Research Group of the Faculty Electrical Engineering, Mathematics and Computer Science on Delft University of Technology in Delft, the Netherlands [1].

1.2 Motivation and goals

Software inspection [Fag76] is widely recognized as an effective technique to assess and improve software quality and reduce the number of defects [Rus91, WBM96, PSMV98, WAP+02]. Software inspection involves carefully examining the code, design, and documentation of software and checking them for aspects that are known to be potentially problematic based on past experience.

It is generally accepted that the cost of repairing a defect is much lower when that defect is found early in the development cycle. One of the advantages of software inspection

[1] `www.st.ewi.tudelft.nl`

is that the software can be analyzed even before it is tested. Therefore, potential problems are identified and can be solved early, when it is still cheap to fix them.

In this thesis, we focus on tools that perform automatic code inspection. Such tools allow early (and repeated) detection of defects and anomalies which helps to ensure software quality, security and reliability. Most defect detection techniques are built around static analysis of the code. In addition, various dedicated static program analysis tools are available that assist in defect detection and writing reliable and secure code.

However, such static analyses come with a price: in the case that the algorithm cannot ascertain whether the source code at a given location obeys a desired property or not, it will make the safest approximation and issue a warning, regardless of the correctness. This conservative behavior can therefore lead to false positives, incorrectly signaling a problem with the code. Kremenek and Engler [KE03] observed that program analysis tools typically have false positive rates ranging between 30–100%. In addition, the increased scrutiny with which the code is examined can lead to an explosion in the number of warnings generated, especially when the tool is introduced later in the development process or during maintenance, when a significant code base already exists.

To cope with the large number of warnings, users resort to all kinds of (manual) filtering processes, often based on the perceived impact of the underlying fault. Even worse, this information overload often results in complete rejection of the tool, especially in cases where the first defects reported by the tool turn out to be false positives.

In this thesis, we aim at helping user of automated code inspection tools to deal with this information overload. We investigate the correlation between source code metrics and warning categories as used by the Java FindBugs [cit10f] tool. This lays the basics for developing a more advanced model for ranking warnings to aid developers in identifying and fixing the most severe warnings first .

1.3 Coding standards

Coding standards [cit10d] are a set of guidelines for a specific programming language that recommend a programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and etc. These are not enforced by compilers. As a result, not following some or all the rules has no impact on the executable programs created from the source code.

These standards are becoming more popular because it makes sure that the software quality is secured, enhancing its maintainability, portability and reliability. The rules within these standards are, usually, based on the experts' opinions, which have years of experience relating a certain language in different contexts. MISRA C:2004 [cit04] is one of the most used software development standard for the C programming language.

1.4 Automated inspection tools

Coding standards have become increasingly popular as a mean to ensure software quality throughout the development process. Different tools have appeared, along the years, with the intent of automating the checking of rules in a standard in order to help developers locating potentially difficult or problematic areas in the source code using techniques such as syntactic pattern matching, data flow analyses, type systems, model checking and theorem proving. Many of these tools check for the same kinds of programming mistakes. There are commercial and academic tools which generally come with their own sets of rules, but can often be adapted allowing that custom standards can be also checked automatically. Two examples of these kind of tools for Java programming language are FindBugs and PMD [2] [cit10g].

1.4.1 FindBugs

Next, we briefly introduce the background of FindBugs and the structure of error reports. FindBugs, one of the most popular static analysis tools, is becoming widely used in Java community [3]. FindBugs implements a set of bug detectors for a variety of common bug patterns (code idioms that are likely to be errors), and uses them to find a significant number of bugs in real-world applications and libraries. For instance, FindBugs plays an essential role in the development of Java programs in Google.

Bug patterns in FindBugs are mainly divided into eight categories, that is Bad Practice, Correctness, Experimental, Malicious code vulnerability, Multithreaded correctness, Performance, Security, and Style. For those experienced users, they can define new bug patterns according to their needs. According to the report structure of FindBugs, each bug category includes many bug kinds and each bug kind consists of several bug patterns. Then, we present a brief category description relating the ones which meaning is not evident:

Correctness bug Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. It strives for a low false positive rate.

Bad Practice Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize. It strives to make this analysis accurate, although some groups may not care about some of the bad practices.

Style Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and

[2]PMD developers don't know the meaning of the letters. They just think the letters sound good together.

[3]As of July 2008, FindBugs has been downloaded more than 700,000 times.

redundant null check of value known to be null. There are more false positives accepted.

Experimental These results are from running FindBugs 1.2.0 at standard effort level. These do not include any low priority warnings or any warnings about vulnerabilities to malicious code. Although FindBugs developers have (repeatedly) manually audited the results, they haven't manually filtered out false positives from these warnings, so that we can get a feeling for the quality of the warnings generated by FindBugs.

1.5 Source code metrics

We best manage what we can measure. A source code metric is a measure of some property of a piece of software source code. It is very difficult to satisfactorily define or measure "how much" software there is in a program. The practical utility of software measurements has thus been limited to narrow domains where they include:

- Size
- Complexity
- Quality

Measurement enables the organization to improve the software process; assists in planning, tracking and controlling the software project and assess the quality of the software thus produced. It is the measure of such specific attributes of the process, project and product that are used to compute the software metrics. Metrics are analyzed and they provide a dashboard to the management on the overall health of the process, project and product. Generally, the validation of the metrics is a continuous process spanning multiple projects. The kind of metrics employed generally account for whether the quality requirements have been achieved or are likely to be achieved during the software development process. As a quality assurance process, a metric is needed to be revalidated every time it is used. In general, for most software quality assurance systems the common software metrics that are checked for improvement are the Source lines of code, cyclomatic complexity of the code, number of classes and interfaces, cohesion and coupling between the modules etc.

Software Quality Metrics focus on the process, project and product. By analyzing the metrics the organization can take corrective action to fix those areas in the process, project or product which are the cause of the software defects.

1.5.1 State Of Flow Eclipse Metrics

This Eclipse [cith] plugin calculates various metrics for source code during build cycles and warns, via the Problems view, of 'range violations' for each metric. This allows to stay continuously aware of the health of the code base. We may also export the metrics to HTML for public display or to CSV or XML format for further analysis. This export can be performed from within Eclipse or using an Ant task.

The *method's* metrics addressed in this thesis are the following:

Cyclomatic Complexity This metric is an indication of the number of 'linear' segments in a method (i.e. sections of code with no branches). It can also be used to indicate the psychological complexity of a method [citp].

Feature Envy Feature Envy occurs when a method is more interested in the features (methods and fields) of other classes than its own [citj].

Lines of Code in Method This measure indicates the number of lines a method occupies. A line is determined by the presence of a newline character. It is a very basic measure of size and is susceptible to variation purely on the basis of different formatting styles [citn].

Number of Locals in Scope This metric is an indication of the maximum number of local variables in scope at any point in a method. The idea of the metric is that a large Number Of Locals In Scope increases complexity and reduces comprehensibility [citu].

Number of Levels This metric is an indication of the maximum number of levels of nesting in a method. The idea of the metric is that a large Number Of Levels increases complexity and reduces comprehensibility. All methods that have a large Number Of Levels can be simplified by extracting private methods or by creating a Method Object, raising the level of abstraction [citt].

Number of Parameters This metric is a count of the number of parameters to a method. Methods with a large Number Of Parameters often indicate that classes are missing from the model. Most methods that have a large Number Of Parameters can be simplified by grouping parameters into a number of related sets and making classes from those sets. This leads to an increase in maintainability [citv].

Number of Statements This metric represents the number of statements in a method. A large number of statements is not in itself a bad thing, it does suggest the possibility of extracting methods which gives related groups of statements a name and therefore increases the level of abstraction and deepens semantics [citw].

The *type's* metrics addressed in this thesis are the following:

Efferent Couplings This metric is a measure of the number of types the class being measured 'knows' about. In short, all types referred to anywhere within the source of the measured class. A large efferent coupling can indicate that a class is unfocussed and also may indicate brittleness, since it depends on the stability of all the types to which it is coupled. Efferent coupling can be reduced by extracing classes from the original class (i.e. decomposing the class into smaller classes) [citi].

Lack of Cohesion in Methods Cohesion is an important concept in OO programming. It indicates whether a class represents a single abstraction or multiple abstractions. The idea is that if a class represents more than one abstraction, it should be refactored into more than one class, each of which represents a single abstraction. We have selected four definitions of lack of cohesion. That of Chidamber and Kemerer, that of Henderson and Sellers, Total Correlation and Pairwise field irrelation [citm].

Number of Fields This metric represents the number of fields in a class. Although a large number of fields is not necessarily an indication of bad code, it does suggest the possibility of grouping fields together and extracting classes, taking appropriate methods as well [cits].

Weighted Methods Per Class This metric is the sum of cyclomatic complexities of methods defined in a class. It therefore represents the cyclomatic 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 [web].

1.6 Contributions

The main propose of this study is to answer the following research questions:

- *Is there any correlation between source code metrics and FindBugs warnings' categories?*
- *Are the same correlations significant across projects?*

This thesis makes the following **contributions**:

- *A correlation between source code metrics and warning categories approach for developing a more advanced model for ranking warnings.* This includes an approach to observe the warnings output by one bug-finding tool for two subject programs, collecting its source code metrics, and a study of statistics analysis (Section 3.3).

- *Analysis of empirical data.* The results are presented in Section 5.3: it contains two parts: (1) analysis of Tomcat system, in Section 5.3.1; (2) analysis of Axis system in Section 5.3.2; further empirical data includes histograms and descriptive statistics of source code metrics for both projects in Subections 5.3.1.1 and 5.3.2.1 respectively. We will also present histograms and descriptive statistics for FindBugs warnings' categories for the projects in Subsections 5.3.1.2 and 5.3.2.2 respectively. Ultimetaly, in the Subsections 5.3.1.3 and 5.3.2.3, we have studied the relationship between the source code metrics and warnings' categories for Tomcat and Axis.

1.7 Thesis outline

This thesis is organized as follows. Besides the introduction, it has more 5 chapters. The Chapter 2 is dedicated to present some other research teams who have worked in areas related to our research. Our work is partially based on some of their contributions.

In the Chapter 3, we describe the purpose and introduce the problem. Next, we describe the approach used to investigate the research questions.

Implementation is described in Chapter 4. We explain how to use and setup Eclipse, FindBugs and State of Flow Eclipse metrics to collect all the information needed. Next, we explain the algorithm used to merge all that data. Finally we go into detail of statistical analysis of source-code metrics and warnings' categories.

In the Chapter 5, we report on the experiments to provide the answers to our research questions. It is presented the experimental setup and a short description of the projects that we choose to perform the experiments. Ultimately, we discuss the threats to validity of the experimental results.

Finally, in chapter 6, we present the final conclusions of the research, along the main contributions of this project and future lines of work.

Chapter 2

Related Work

The main focus of this chapter is to present the actual literature about this thesis' subject. We present a summary that includes the most relevant researches done about this matter. We also make brief references to prior work whenever that is in order.

2.1 Issues of MISRA Standard

The idea of a safer subset of a language, the precept on which the MISRA coding standard is based, was promoted by Hatton [Hat95]. In [Hat03], he assesses a number of coding standards, introducing the signal to noise ratio for coding standards, based on the difference between measured violation rates and known average fault rates. He assessed MISRA C 2004 in [Hat07], arguing that the update was no real improvement over the original standard, and "both versions of the MISRA C standard are too noisy to be of any real use".

2.2 Automated inspection tools

Software inspection tools have been studied widely. Zitser et al. evaluated several open source static analyzers with respect to their ability to find known exploitable buffer overflows in open source code [ZLL04]. Engler et al. evaluate the warnings of their defect detection technique in [ECC01].

In [RAF04], it was examined the results of applying five tools, specifically Bandera, ESC/Java 2, FindBugs, JLint and PMT, to a variety of Java programs over different checking tasks. Thereby it was possible to cross-check their bug reports and warnings. In experimental results it was showed that none of the tools can fully replace one-another, and indeed the tools often find non-overlapping bugs (mostly warnings are distinct). There is also no correlation of warning counts between pairs of tools. Therefore, it was proposed a meta-tool which combines the output of the tools together, looking for particular lines of code, methods, and classes that many tools warn about. Thus it enables to precisely identify false positives and false negatives. Summarizing, this meta-tool automatically combine and correlate their output. It was concluded that the main difficulty in using the tools is simply the quantity of output (mostly because of false positives).

Wagner et al. compared results of defect detection tools with those of code reviews and software testing [WJK$^+$05]. Their main finding was that bug detection tools mostly find different types of defects than testing, but find a subset of the types found by code reviews. Warning types detected by a tool are analyzed more thoroughly than in code reviews.

Heckman et al. proposed a benchmark and procedures for the evaluation of software inspection prioritization and classification techniques, focused at Java programs [HW08].

2.3 Static profiling

Static profiling is used in a number of compiler optimizations or worst-case execution time (WCET) analyses. By analyzing program structure, a prediction is made as to which portions of the program will be most frequently visited during execution. Since this heavily depends upon branching behavior, some means of branch prediction is needed. This can range from simple and computationally cheap heuristics to expensive data flow based analyses such as constant propagation [KR00] [SRH95] [Bin94] or symbolic range propagation [VCH96] [BE96] [Pat95]. However, only [WL94] [WMGH94] compute a complete static profile.

2.4 Prioritizing results statically

In recent years, many solutions have been proposed to reduce the number of inspected violations and, instead, emphasis on the most relevant ones, according to some criterion.

The classical approach most automated inspection tools use for prioritizing and filtering results is to classify the results based on several levels (statically). Such levels are associated with the type of defects detected; they are obvious of the actual code that is being analyzed and of the location or frequency of a given defect. Therefore, the ordering and filtering that can be achieved while using this technique is rather crude.

Kremenek and Engler [KE03], proposed Z-ranking, a statistical approach to reduce the number of false positives due to inaccurate static analysis. With that technique it is possible to rank the output of static analyses tools so that more important warnings will tend to be ranked more highly. Z-Ranking is intended to rank the output of a particular bug checker. They prioritize checks (warning categories) using the frequency of check results.

Software checkers output success (indicating a successful check of an invariant) or failure (the invariant did not hold). If the ratio of successes to failures is high, the failures are assumed to be real bugs. If we apply this idea to warning categories, a warning category that has fewer warning instances is important.

In [BM06], Boogerd and Moonen presented ELAN, a technique that helps the user to prioritize the information generated by a software inspection tool. It is based on a demand-driven computation of the likelihood that execution reaches the locations for which warnings are reported using static profiling. Their goal was to help the user of automated inspection tools to deal with the information overload. Instead of focusing on improving a particular defect detection technique in order to reduce false positives, they strive for a generic prioritization approach that can apply the results of any software inspection tool and assists the user in selecting the most relevant warnings. However, a location which has low execution likelihood or warnings at the location could be important. In fact, severe bugs in lines with low chance of execution are more difficult to detect. Summarizing they use execution likelihood analysis to prioritize warning instances. For each warning location, they compute the execution likelihood of the location. If the location is very likely to be executed, the warning gets a high priority. If the location is less likely to be executed, the warning gets a low priority.

2.5 Bug-fixes approach

2.5.1 Software Configuration Management

Software configuration management (SCM) [cit10h] is the task of tracking and controlling changes in the software. It is responsible for the management of changes to documents, programs, and other information stored as computer files. It is most commonly used in software development. Configuration management practices include revision control and the establishment of baselines.

In recent years, many approaches have been proposed that benefit from the combination of data present in SCM systems and issue databases. Applications range from an examination of bug characteristics [LTW$^+$06], techniques for automatic identification of bug- introducing changes [SZZ05, KZPW06], bug-solving effort estimation [WPZZ07], prioritizing software inspection warnings [KE07] prediction of fault-prone locations in the source [KE07].

Williams and Hollingsworth [WH05] use software change histories to improve existing bug-finding tools. When a function returns a value, using the value without checking it may be a potential bug. The problem is that there are too many false positives if a bug-finding tool warns all source code that uses unchecked return values. To remove the false positives, Williams and Hollingsworth use the software histories and find what kinds of function return values must be checked. They leverage the software history to remove false positives. However, they only focus on the small sets bug patterns such as return value checking and it is not generic to all warnings.

Li et al. analyze and classify fault characteristics in two large, representative open-source projects based on the data in their SCM systems [LTW+06]. Rather than using software inspection results they interpret log messages in the SCM.

2.5.2 Revision Control Systems

Revision control system [citx] is an important component of Software Configuration management. This system can record the history of sources files and documents keeping track of all work and all changes in a set of files. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged. This way, it allows several developers (potentially widely separated in space and/or time) to collaborate. One of the most currently used RVS is SVN (Subversion) [cit10i] which is a mostly-compatible successor of the widely used CVS (Concurrent Versions System) [cit10e].

Fischer et al. [FPG03] introduced an approach in order to populate a release history database that combines version data with bug tracking data and adds missing data not covered by version control systems such as merge points. Then simple queries can be applied to the structured data to obtain meaningful views showing the evolution of a software project. Such views enable more accurate reasoning of evolutionary aspects and facilitate the anticipation of software evolution. They demonstrate their approach on a large Open Source project Mozilla that offers great opportunities to compare results and validate their approach.

Spacco et al. [SHP06] observed FindBugs warning categories across software versions measuring lifetimes of warning categories, warning number rates and the degree if decay using warning numbers. They said it is important to be able to track the occurrence of each potential defect over multiple versions of a software artifact under study to determine when warnings reported in multiple versions of the software all correspond the same underlying issue. One motivation for this capability is to remember decisions about code that has been reviewed and found to be safe despite the occurrence of a warning. Another motivation is constructing warning deltas between versions, showing which warnings are new, which have persisted, and which have disappeared. This allows reviewers to focus their efforts on inspecting new warnings. Finally, tracking warnings through a series of software versions reveals where potential defects are introduced and fixed, and how long they persist, exposing interesting trends and patterns. They discussed two different techniques they have implemented in FindBugs for tracking defects across versions, discussed their relative merits and how they can be incorporated into the software development process, and discuss the results of tracking defect warnings across Sun's Java runtime library.

Kim et al. [KPW06] presented a bug finding algorithm using bug fix memories: a project-specific bug and fix knowledge base developed by analyzing the history of bug fixes. A bug finding tool, BugMem, implements the algorithm. Bug fix memories use a learning process, so the bug patterns are project-specific and these can be detected. The algorithm and tool are assessed by evaluating if real bugs and fixes in project histories can be found in the bug fix memories. Source code repositories such as CVS and Subversion are contains knowledge that can be used to discriminate between good and bad source code.

Basalaj takes an alternative viewpoint and studies single versions from 18 different projects Basalaj [Bas05]. These are used to compute two rankings, one based on warnings generated by QA C++, and one based on known fault data. For 12 out of 900 warning types, a positive rank correlation between the two can be observed . Wagner et al. [WDA$^+$08] evaluated two Java defect detection tools on two different software projects. They investigated whether inspection tools were able to detect defects occurring in the field. Their study could not confirm this possibility for their two projects.

2.5.3 Defect Tracking Systems

Defect Tracking System [cit10a] is a software application designed to help quality assurance allowing individual or groups of developers to keep track of outstanding bugs in their product effectively. A major component of a bug tracking system is a database that records facts about known bugs. Facts may include the time that a bug was reported, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and any programmers who may be working on fixing it . Typical bug tracking systems support the concept of the life cycle for a bug which is tracked through status assigned to the bug. A bug tracking system should allow administrators to configure permissions based on status, move the bug to another status, or delete the bug. The system should also allow administrators to configure the bug statuses and to what status a bug in a particular status can be moved. Some systems will e-mail interested parties, such as the submitter and assigned programmers, when new records are added or the status changes [cit10b]. Bugzilla [cit10c] is one of the most favorite system between hundreds of organizations across the globe.

2.5.4 Bug-introducing changes

A bug-introducing change is the modification in which bug was injected into the software. Subsequently, a bug-fix change exists when a developer modifies the project's source code and repairs the bug.

Software evolution research leverages the history of changes and bug reports that accretes over tine in SCM systems and bug tracking systems to improve the understanding of how a project has grown. It offers the possibility that by examining the history of changes made to a software project, we might better understand patterns of bug introducing, and raise developer awareness that they are working on risky sections of a project.

Kim et al. [KZPW06] presented algorithms to automatically and accurately identify bug-introducing changes. Bug-introducing changes are important information for understanding properties of bugs, mining bug prone change patterns, and predicting future bugs. They remove false positives and false negatives by using annotation graphs, by ignoring non-semantic source code changes and outlier fixes.

2.5.5 History-Based Rule Selection

This approach has the purpose to use software and issue archives to link standard violations to known bugs. That information can then be used to address two practical issues in using a coding standard: which rules to adhere to and how to rank violations of those rules focusing on the historical fault likelihood to prioritize violations.

In [KE07], Kim and Ernst discovered that using a historical fault likelihood to prioritize warnings improved accuracy on fault detection. Their goal is to propose a new, program-specific prioritization that more effectively directs developers to errors. The new history-based warning prioritization (HWP) is obtained by mining the software change history for removed warnings during bug fixes. A version control system indicates when each file is changed. A software change can be classified as a fix-change or a non-fix change. A fix-change is a change that fixes a bug or other problem. A non-fix change is a change that does not fix a bug, such as a feature addition or refactoring. The underlying intuition is that if warnings from a category are eliminated by fix-changes, the warnings are important. Suppose that during development, a bug-finding tool would issue a warning instance from the Overflow category. If a developer finds the underlying problem and fixes it, the warning is probably important. They do not assume the software developer is necessarily using the bug-finding tool. On the other hand, if a warning instance is not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing. Using this intuition, they set a weight for each warning category to represent its importance. The weight of a category is proportional to the number of warning instances from that category that are eliminated by a change, with fix-changes contributing more to the weight than non-fix changes. The weight of each category determines its priority.

2.6 Summary

Coding standards and automated inspection tools help to prevent the introduction of faults in software. However, one of the main issues with these tools is that they are notorious for producing an overload of non-conformace warnings revealing about 30% of false positives [KAYE04]. Manual inspection of these violations adds a significant overhead for developers. Therefore, it is needed a trade-off algorithm which identifies the warnings to resolve. The algorithm should identify the "critical" violations that will most likely lead to an error hence need to be addressed first.

There are several approaches that may be used in order to resolve two practical issues related to the use of coding standards: which rules should be accepted and how to rank these rules' violations.

Chapter 3

Research questions and approach

In Section 3.1, we describe the thesis problem in more detail. In Section 5.3 it is presented the research questions. Finally, in Section 3.3, we expose an abstract explanation of the approach adopted.

3.1 Purpose and Problem

Current bug-finding tools report a high number of false positives warnings: most warnings do not indicate real bugs. Usually bug-finding tools assign important warnings high priority. However, the prioritization of warnings tends to be ineffective leaving it to developers to select the warnings to be addressed first.

In this thesis project, we investigate the correlation between source code metrics and warning categories used by one Java bug-finding tool. Therefore, we observe the warnings output by one bug-finding tool for two subject programs. We also collect source code metrics from these systems. Ultimately, we study the statistics analysis.

This lays the basics for developing a more advanced model for ranking warnings to aid developers in identifying and fixing the most severe warning first.

3.2 Research Question

The main propose of this study is to answer the following research questions:

- *Is there any correlation between source code metrics and FindBugs warnings' categories?*

  More specifically, we wanted to know if a type (class) with a certain metric, for instance, high cyclomatic complexity, can lead to warnings of a particular category, for example performance.

- *Are the same correlations significant across projects?*

  Although we chose two cases, their results may be different. Answering this question this will tell us whether we can select a generic set of correlations, important for either all projects or a certain type of project.

The approach used to address this problem can be found in section 3.3.

3.3 Approach

In order to provide the answers to the research questions, we have used two open-source Java applications as case studies, Tomcat and Axis.

We have started by running the Eclipse IDE to import the projects. We choose Eclipse because it is one of the most used IDE for Java language and has an extensible plugin system.

We have used FindBugs as bug-finding tool to compute the warnings. It is one of the most used bug-finding tool in software engineering community and it has a great plugin for Eclipse. It has also a standard version with command line, ant, and Swing interfaces but it is harder to setup.

State Of Flow Eclipse Metrics is the metric collector we have chosen. As we prefer to work with eclipse plugins due its simplicity we have not so many options. However, this one gives the most important metrics of a project in a very useful way.

We have also developed a script which filters and merges all the desired data (source code metrics and warnings) by class.

Finally, we use that data as input to PASW Statistics tool. We present histograms and descriptive statistics for source code metrics and warnings' categories. It is also studied the relationship between the source code metrics and warnings' categories.

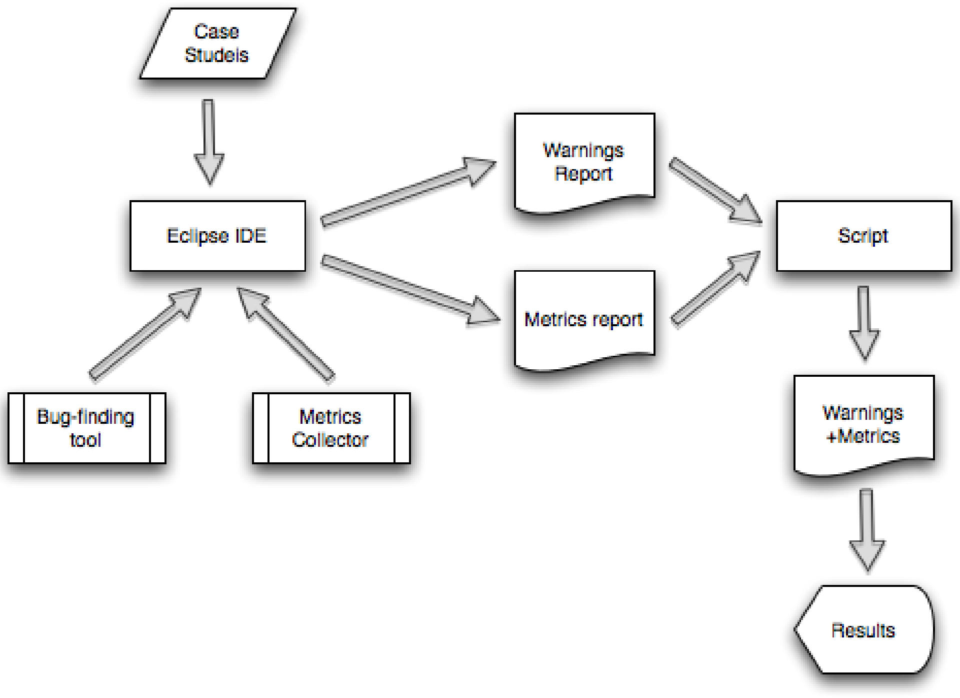


Figure 3.1: Approach workflow

3.4 Summary

Current tools to check coding standards report a high number of false positives warnings. The warnings' prioritization also tends to be ineffective leaving it to developers to select the warnings to be addressed first. In this thesis project, we investigate the correlation between source code metrics and warning categories used by one Java bug-finding tool. This lays the basics for developing a more advanced model for ranking warnings to aid developers in identifying and fixing the most severe warning first.

The main propose of this study is to answer the following research questions:

- *Is there any correlation between source code metrics and FindBugs warnings' categories?*
- *Are the same correlations significant across projects?*

The Figure 3.1 shows the experimental flow adopted in our study. We use two open-source Java applications as case studies, Tomcat, and Axis. We adopt Eclipse IDE to import the projects. We use the eclipse plugin of the FindBugs tool to compute the warnings. We also use a eclipse plugin to collect the source code metrics (State Of Flow Eclipse Metrics). We develop a script which merges all the source code metrics' and warnings' data by class. Finally, we use that data as input to the statistics tool.

Chapter 4

Implementation

This chapter is dedicated to the method's presentation used in the implementation of the approach explained in Section 3.3.

We have used two open-source Java applications as case studies. So, we start by downloading the source code distributions of each system, Tomcat [citb], and Axis[citd]. The next step is to run Eclipse IDE[citf] and create a new Java project from existing source using the previous download.

4.1 Findbugs

The FindBugs[cit10f] Eclipse plugin allows FindBugs to be used within the Eclipse IDE. To use the FindBugs Plugin for Eclipse, it is needed Eclipse 3.3 or later, and JRE/JDK [citl] 1.5 or later.

To get started, right click on a Java project in Package Explorer, and select the option labeled "Find Bugs". FindBugs will run, and problem markers (displayed in source windows, and also in the Eclipse Problems view) will point to locations in code which have been identified as potential instances of bug patterns.

We can also run FindBugs on existing java archives (jar, ear, zip, war etc). Simply create an empty Java project and attach archives to the project classpath. Having that, we can now right click the archive node in Package Explorer and select the option labeled "Find Bugs". If we additionally configure the source code locations for the binaries, FindBugs will also link the generated warnings to the right source files.

In our study, we need the warnings report in XML format:

1. In Eclipse IDE, go to Project Explorer View, **right click over the project**.
2. Click on **Find Bugs**, and then Click on **Save XML**.

3. Select the export directory and **save** the file with the name: **warnings.xml**.

It creates one file named warnings.xml with all the warnings data. This file is used as input for the script which filters and merges the desired data.

4.2 State of Flow Eclipse Metrics

This Eclipse plugin calculates various metrics for the code during build cycles and warns us, via the Problems view, of 'range violations' for each metric. This allows us to stay continuously aware of the health of the code base. We may also export the metrics to HTML for public display or to CSV or XML format for further analysis. This export can be performed from within Eclipse or using an Ant task.

In our study we need to export the source code metrics to a CSV file:

1. In Eclipse IDE, go to Project Explorer View and **right click over the project**.
2. Click on **Export**.
3. Click on **Other**, Click on **metrics**, and then click **Next**.
4. Check only the box **Export CSV**, select the export directory, and then click on **Finish**.

It creates 2 CSV files with source code metrics, one containing the methods' metrics (methods.csv), and the other one with types metrics (types.csv). These files are also used as input for the script which filters and merges the desired data.

4.3 Merging algorithm

We have developed a python script which filters and merges the methods' metrics (methods.csv), types' metrics (types.csv), and warnings' data (warnings.xml) by class into one CSV file. It stores all classes' names, methods' metrics, classes' metrics, number of occurrences of each warning type, and number of occurrences of each warning category.

The script defines 3 classes:

ClassMetrics This is the master class which stores the data relative to each class. Each of this classes' objects store the required data by project's classes (methods' metrics, types' metrics, number of occurrences of warnings' types and categories). Figure 4.1 presents a source code snippet of this class declaration.

```
class ClassMetrics:
        def __init__(self, package, name, method, line, m1,...,m14:
                self.package  = package
                self.name     = name
                self.methods  = [Method(method, line, m1,..., m7)]
                self.warnings = []
                self.warningTypes = []
                self.warningCategories = []
                self.m8 = m8
                ...
                self.m14=  m14

        def attributes(self):
                if self.package  == '':
                        stringPackageAndType = self.name
                else:
                        stringPackageAndType = self.package + '.' +
                        self.name
                ...
                stringClassMetrics   =',' + str(max_m1) + ','
                +... + str(max_m7)
                ...
                stringTypeMetrics = ',' + self.m8 + ',' +...+ ','
                + self.m14

                for i, nw in enumerate(self.warningTypes):
                        stringWarningTypesCount =
                        stringWarningTypesCount + ',' + str(nw)

                for i, nw in enumerate(self.warningCategories):
                        stringWarningCategoriesCount =
                        stringWarningCategoriesCount + ',' + str(nw)

                for i, w in enumerate(self.warnings):
                        stringWarnings = stringWarnings + w.attributes()

                return stringPackageAndType +...+
                stringWarningCategoriesCount + '\n'
```

Figure 4.1: Script's Class declaration - ClassMetrics

Method This class stores the data about each method. It has the attributes *name*, *line*, *m1*, *m2*, *m3*, *m4*, *m5*, *m6*, *m7*. The attribute *name* is the method's name, the *line* attribute is the line of code where the method begin and from attributes *m1* to *m7* are the methods' metrics. It is presented in Figure 4.2 a source code snippet of this class declaration.

Warning This class stores the warnings' data. It has the attributes *type*, *priority*, *abrev*, and *category*. The *type* attribute stores the type of warning, the *priority* stores the warning priority, the *abrev* stores the abbreviature and finally, the *category* attribute stores the category of the warning. We present in Figure 4.3 a source code snippet of this class declaration.

```
class Method:
        def __init__(self, name, line, m1,...,m7):
                self.name = name
                self.line = line
                self.m1   = int(m1)
                ...
                self.m7   = int(m7)

        def attributes(self):
                return self.name + ',' +...+ str(self.m7)
```

Figure 4.2: Script's Class declaration - Method

```
class Warning:
        def __init__(self, type, priority, abrev, category):
                self.type     = type
                self.priority = priority
                self.abrev    = abrev
                self.category = category

        def attributes(self):
                return self.type + ',' +...+  self.category + '\n'
```

Figure 4.3: Script's Class declaration - Warning

The script body starts by reading the file *methods.csv* line by line in which each line has the method's data. So, we extract the class information contained in that file and we identify all the classes. If the class already exists in array, we add the method and its metrics. Otherwise, if the class does not exist in array, we add the class. In Figure 4.4, we present a source code snippet which does the parsing of the *methods.csv* file.

Next, we read the file *types.csv* line by line in which each line has the types' (classes') data. So, we extract the class information contained in that file and we identify all the classes. It searches for the corresponding class. If the class already exists in, we add the metrics. Otherwise, if the class does not exist, we create a new class and add the metrics. Figure 4.5 presents a source code fragment which does the parsing of the *types.csv* file.

Afterwards, we read the file *warnings.xml* line by line. We extract the class information contained in that file and we identify all the classes. We search for the corresponding class and create a new warning object type. If there is not any warning of that type, we add this warning to the warning type list. We do the same procedure for the warning categories. Therefore, for each project's class, we count the number of occurrences of each warning type and each warning category. It is presented in Figure 4.6 a source code fragment which does the matching between warning's data and respective class.

```
for i, line in enumerate(cMetricsLines):
        if line.find('PACKAGE') == -1:
                cp = line.split(',')
                if cp[3] == '': cp[3] = '0'
                ...
                if cp[9] == '': cp[9] = '0'
                if cp[10].strip('\n') == '': cp[10] = '0'

                index = pinpoint(cp[0], cp[1], ClassMetricsList)
                if index != -1:
                        ClassMetricsList[index].methods.append
                        (Method(cp[2],...,cp[10]))
                else:
                        cm = ClassMetrics(cp[0],..., cp[10],
                        '0', '0', '0', '0', '0', '0', '0')
                        ClassMetricsList.append(cm)
```

Figure 4.4: Script's Body - methods.csv parsing

```
for i, line in enumerate(typesMetricsLines):
        if line.find('PACKAGE') == -1:
                cp = line.split(',')
                if cp[3] == '': cp[3] = '0'
                ...
                if cp[8] == '': cp[8] = '0'
                if cp[9].strip('\n') == '': cp[9] = '0'
                if cp[1].find('$'):
                        cp[1] = cp[1].replace('$', '.')
                index = pinpoint(cp[0], cp[1], ClassMetricsList)
                        if index != -1:
                        if int(cp[3]) > int(ClassMetricsList[index].m8):
                         ClassMetricsList[index].m8  = cp[3]
                        ...
                        if int(cp[9]) > int(ClassMetricsList[index].m14):
                         ClassMetricsList[index].m14  = cp[9]
                else:
                        cm = ClassMetrics(cp[0], cp[1], '', '0',...,
                        cp[3],..., cp[9])
                        if cm.name.find('$'):
                                cm.name = cm.name.replace('$', '.')
                        ClassMetricsList.append(cm)
                        index= pinpoint(cm.package, cm.name,
                        ClassMetricsList)
```

Figure 4.5: Script's Body - types.csv parsing

```
index = pinpoint(cpackage, cname, ClassMetricsList)

if index != -1 and ClassMetricsList[index].package == cpackage:
        w = Warning(cp[3].split('=')[1].strip('\"'),
        cp[4].split('=')[1].strip('\"'), cp[5].split('=')
        [1].strip('\"'), cp[6].split('=')[1].strip('\"'))
...
else:
for j, wt in enumerate(WarningTypesList):
        if cp[3].split('=')[1].strip('\"') == wt:
                break
```

Figure 4.6: Script's Body - warnings matching

We calculate the maximum value of each method's metrics. We store these maximum values in variables from *max_m1* to *max_m7*. We compute the maximum values of each methods' metric. We present in Figure 4.7 a source code fragment of how the maximum value of each method's metrics is calculated.

```
for i, m in enumerate(self.methods):
        stringMethodsMetrics = stringMethodsMetrics + m.attributes()

        if m.m1 > max_m1:
                max_m1 = m.m1
        ...
        if m.m7 > max_m7:
                max_m7 = m.m7
```

Figure 4.7: Script - Maximum value of each methods' metrics calculation

We compute the header of the output file (metrics+warnings.csv) dynamically, depending on metrics order, warning types, and warning categories in which execution time. Layout of the header: package.class,methods' metrics,types' metrics,warnings' types, warnings' categories. Figure 4.8 presents a source code fragment of the header's computation.

```
methods_header = line.replace('METHOD,LINE,', '')
methods_header = methods_header.replace('PACKAGE,', '').strip('\n')

types_header = line.replace('PACKAGE,TYPE,LINE', '').strip('\n')

for i, wt in enumerate(WarningTypesList):
        warningTypesHeader = warningTypesHeader + ',' + wt

for i, wt in enumerate(WarningCategoriesList):
        warningCategoriesHeader = warningCategoriesHeader + ',' + wt

open('metrics+warnings.csv', 'a').write('\%s\%s\%s\%s\n' \%
(methods_header, types_header, warningTypesHeader, warningCategoriesHeader))
```

Figure 4.8: Script - Header computation

Finally we print to metrics+warnings.csv file the data collected. In Figure 4.9, we present a source code fragment of how the results' output is printed to a file.

```
for i, cm in enumerate(ClassMetricsList):
        while len(cm.warningTypes) < len(WarningTypesList):
                 cm.warningTypes.append(0)

        while len(cm.warningCategories) < len(WarningCategoriesList):
                cm.warningCategories.append(0)

        open('metrics+warnings.csv', 'a').write(cm.attributes())
```

Figure 4.9: Script - Print the results into metrics+warnings.csv file

4.4 Statistical Analysis

We start by running PASW Statistics tool [citk]. Next, we explain step by step how to import the CSV file generated by the previous script to the Statistics tool:

1. We begin by opening the CSV data file generated by the previous script (**File**, **Open**, **Data**, and select the file **metrics+warnings.csv**).
2. It opens the text import wizard. We choose that the text file **does not match a predefined format** and select **Continue**.
3. In the next screen we select that **variables are delimited by a specific character**, **the variable names are included at the top of file**, and select **Continue**.
4. Now, just press **Continue**.
5. We select **Comma as the only delimiter between variables** and **none text qualifier**, then just click **Continue**.
6. Now, we have to define the data format of the variables: The first one (**Type**) is a **String with 128 characters**, all the **remaining variables** are **Numeric**, and click **Done**.
7. Finally, just press **Done** again.

In PASW Statistics Data Editor, we select **Variable View** and we have to change the Measure parameters of the variables. While the **Type** is a **Nominal** one, **the source code metrics and Warning Categories** are **Scale** [citr]. At this moment, the statistics tool has all the required data for the analysis.

We have done charts (histograms) for all the source code metrics:

1. Click on **Graphs** Menu, and then **Chart Builder**.
2. We select **Histogram** and drag it to the preview area. We drag the desired metric variable to the XX axis.

We have performed this procedure for all source code metrics.

We have also studied the descriptive statistics for source code metrics:

- In PASW, we go to **Analyze** menu, **Descriptive Statistics**, and then **Frequencies**.
- We select the **metrics variables** to be analyzed. We also sign the checkbox to **display frequency tables** and then click in **Statistics** button.
- In the new dialog window, we select **Median**, **Maximum**.

Considering the descriptive statistics for the warning categories, the procedure is the same as the previous for source code metrics except that after click on statistics button we select **Median**, **Maximum** and **Sum**.

Most classes do not have warnings from any category resulting in too much zero-values. Hence, for each warning category, we filter the cases with 0 occurrences:

1. Go to **Data** Menu and then **Select Cases**. In the new dialog window, we select "**If condition is satisfied**" and next click the **If** button. Now we select **one warning category** and write "**WC > 0**", where WC is the desired warning category.
2. Now, we do the **warning category histogram** as already described for source code metrics.

   We repeat this procedure for all warning categories.

Finally we have studied the correlation between source code metrics and warning categories using Spearman correlation analysis:

1. We go to **Analyze** menu, then **correlate**, and finally **bivariate**.
2. In the next screen, we select **all the source code metrics** and **one warning category**; We use **Spearman correlation coefficients** and **Two-tailed test of significance**. We also check the item "**Flag significant correlations**.

We repeat this procedure for all warning cateogories.

4.5 Workflow of experiments

Our approach to provide the answer to the research questions comprises 5 stages:

1. Import the case studies to Eclipse IDE
2. Warnings' computation
3. Metrics' computation
4. Merging all desired data
5. Statistics analysis

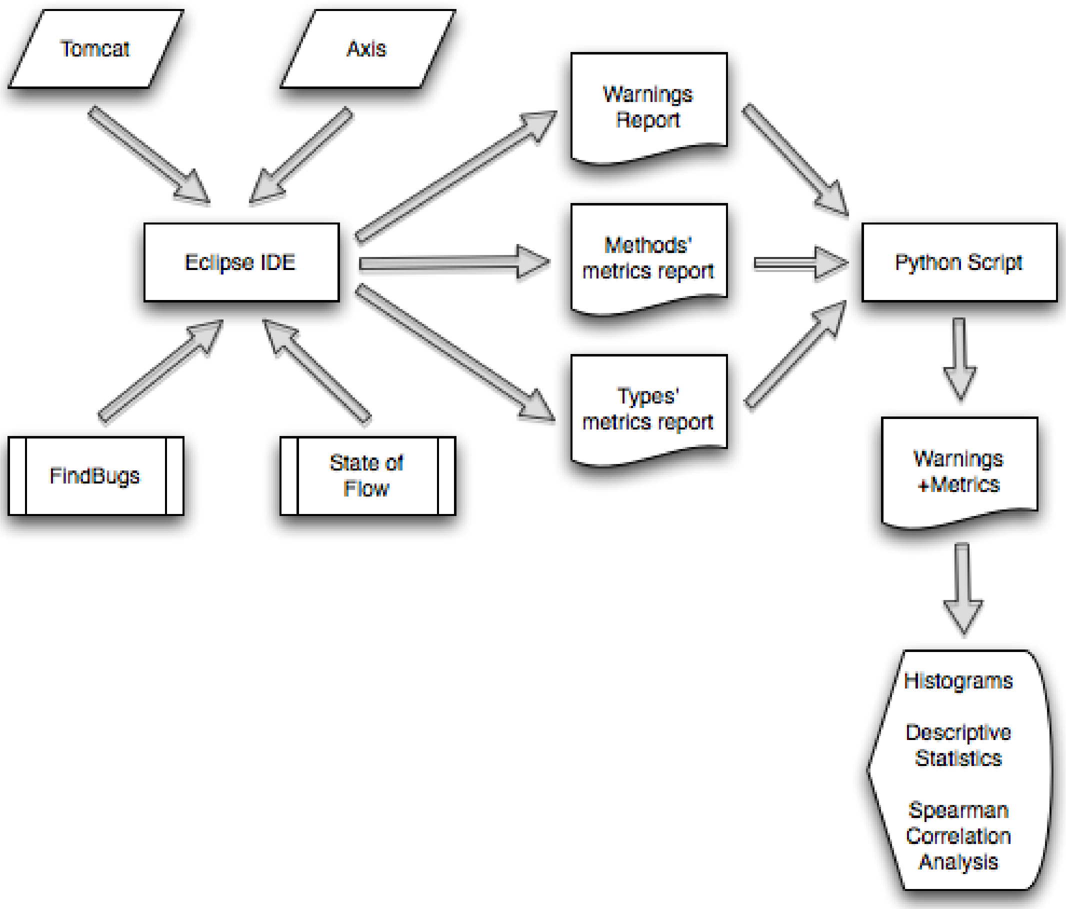


Figure 4.10: Implementation workflow

4.6 Summary

We have used two open-source Java applications as case studies, Tomcat, and Axis. We have started by downloading the source code distribution and running the Eclipse IDE to import the project.

We have used the eclipse plugin of the FindBugs to compute the warnings. We export the warnings report into a XML file.

State of Flow Eclipse metrics plugin is used to compute the source code metrics. These metrics were exported into 2 CSV files - one containing all the methods' metrics and the other one, the types' (classes') metrics. In our study we focus on class metrics. Therefore, we have selected the value of each class's metric by choosing the maximum occurrence of it throughout all methods of the class.

We have also developed a python script which filters and merges all the desired data by class into one CSV file. It stores all classes' names, methods' metrics, classes' metrics, number of occurrences of each warning type, and number of occurrences of each warning category.

Finally, we use the last CSV file as input to the PASW Statistics tool. We present histograms and descriptive statistics for source code metrics and warnings' categories. It is also studied the relationship between the source code metrics and warnings' categories using Spearman correlation analysis.

Chapter 5

Empirical Evaluation

This is the core chapter of this thesis in which we report on the experiments to provide the answers to our research questions. It is divided into 5 sections. In section 5.1, we present the experimental setup. In section 5.2, we do a short description of the projects that we have chosen to perform our experiments. Results are analyzed in section 5.3, and the answers to the research questions are outlined in section 5.4. In the last section 5.5, we discuss the threats to validity of the experimental results.

5.1 Experiment setup

The goal of the evaluation is to provide the answers to the following two research questions:

- *Is there any correlation between source code metrics and FindBugs [cit10f] warnings' categories?*
- *Are the same correlations significant across projects?*

For this we have used two open-source Java applications, Tomcat [cita] and Axis [citz]. While Tomcat is a widely used web application server, Axis is an implementation of the SOAP [1] submission to W3C.

We have adopted Eclipse IDE [cith] to import these projects. We have computed the set of warnings using the eclipse plugin of the FindBugs tool and this warnings' data has been exported into a XML file. We have also computed the source code metrics using the State Of Flow Eclipse Metrics (plugin) [citg].

[1] SOAP is a lightweight protocol for exchanging structured information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.

These metrics were exported into 2 CSV files - one containing all the methods' metrics and the other one, the types' (classes') metrics. In our study we focus on class metrics. Therefore, we have selected the value of each class's metric by choosing the maximum value of it throughout all methods of the class.

We have also developed a python script which filters and merges all the desired data by class into one CSV file. It stores all class's names, methods' metrics, classes' metrics, number of occurrences of each warning type, and number of occurrences of each warning category.

Finally, we have used the last CSV file as input to the PASW (former SPSS) Statistics tool. We present histograms and descriptive statistics for source code metrics and warnings' categories. We have also studied the relationship between the source code metrics and warnings' categories using Spearman correlation analysis.

5.2 Description of the subject systems

In our experiment, we have worked with two open-source Java applications as our subject systems. Apache Tomcat (version 6.0.26) is a widely used web application server. It is an implementation of the Java Servlet and JavaServer Pages technologies. Axis (version 1.4) is an implementation of the SOAP (Simple Object Access Protocol) submission to W3C.

We have selected different types of projects as subject systems to cover a broader range of bug patterns [2] and types of bugs.

The Table 5.1 shows the subject project's name (Project), number of lines of code in total (LoC), and total number of warnings (# Warnings). Both projects are large in size but only have rather few warnings. This might stem from the fact both projects have many users and therefore are well maintained.

Project	**LoC**	**# Classes**	**# Warnings**
Tomcat	215045	1687	1584
Axis	104925	840	891

Table 5.1: Case studies inspection results

5.3 Results

This section presents the results computed with the PASW statistics tool on the merged Tomcat and Axis data set.

[2]Bug patterns are code idioms that often result in errors. In other words, they are error-prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes.

It contains two different parts. First, we take a more detailed look at the Tomcat system in section 5.3.1, and then at the Axis in section 5.3.2.

In the subsections 5.3.1.1 and 5.3.2.1, we present the study of source code metrics' distribution for Tomcat and Axis respectively. We show histograms and descriptive statistics.

Methods' Metrics In our study we focus on class metrics. Therefore, we have selected the value of each class's metric by choosing the maximum value of it throughout all methods of the class. We study the following metrics:

- **NOL** - Number of Levels
- **NOP** - Number of Parameters
- **FE** - Feature Envy
- **NOS** - Number of Statements
- **CC** - Cyclomatic Complexity
- **LOCm** - Lines of Code in method
- **NLS** - Number of Locals In Scope

Types' Metrics We study the following metrics:

- **NOF** - Number of Fields
- **LCPFI** - Lack of Cohesion in Methods (Pairwise Field Irrelation)
- **LCTC** - Lack of Cohesion in Methods (Total Correlation
- **LCHS** - Lack of Cohesion in Methods (Henderson-Sellers)
- **LCCK** - Lack of Cohesion in Methods (Chidamber & Kemerer)
- **WMC** - Weighted methods per class
- **Ce** - Efferent Couplings

In the subsections 5.3.1.2 and 5.3.2.2, we present the study of FindBugs warnings' distribution for Tomcat and Axis systems respectively. We show histograms and descriptive statistics.

In the subsections 5.3.1.3 and 5.3.2.3, we study the relationship between the source code metrics and warnings' categories for Tomcat and Axis respectively. We use Spearman (Nonparametric) correlation analysis. The correlation coefficients values can vary between -1 and 1. Negative coefficients mean that there is a negative proportion between variables, in other words, the higher one of the variable's values is, the lower is the other one. Near zero values mean that there is no relation between the variables. And finally, positive coefficients mean positive proportions in such way that as higher is the value of one of those variables, the higher is the value of the other.

5.3.1 Tomcat

5.3.1.1 Source code metrics

In this subsection we present the study of source code metrics' distribution. We show histograms and descriptive statistics. Tomcat has a total of 1687 classes and 1584 warnings. We start by analyzing methods' metrics followed by types' metrics.

Methods' Metrics The Table 5.2 shows the descriptive statistics (median and maximum) of the methods' source code metrics. The median is low for all metrics because for most of the methods the metrics have value 0. This might happen because Tomcat is a large system which makes use of a lot of abstract methods. Only a few number of methods reach the highest metrics' values.

	NOL	NOP	FE	NOS	CC	LOCm	NLS
Median	2.00	2.00	0.00	8.00	2.00	13.00	0.00
Max	30	12	5	346	136	481	43

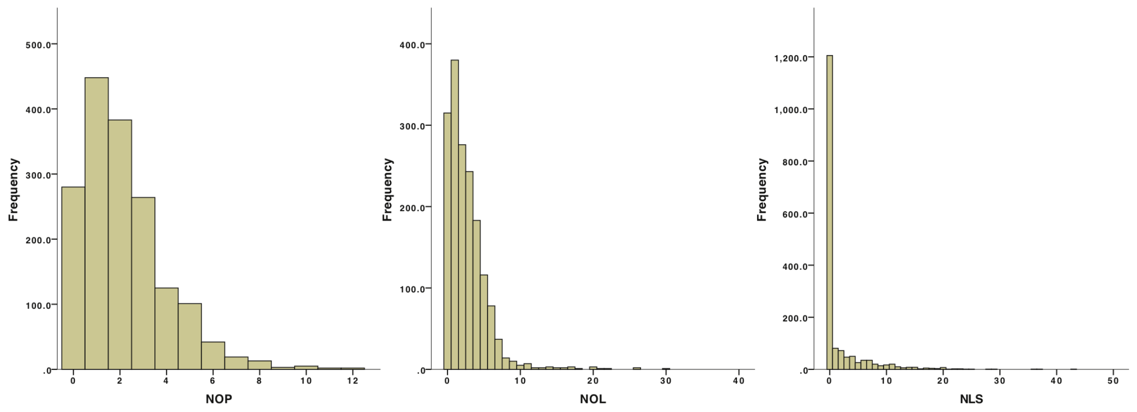
Table 5.2: Median and maximum values of methods' source code metrics for Tomcat

The histogram depicted by Figure 5.1a shows the distribution of the values for the maximum number of parameters metric (NOP). The distribution of values is positively skewed. The large majority of classes, namely 1375 out 1687 (81.5%), contain methods with less than 4 parameters. 280 classes (16.6%) contain methods without parameters. Only few classes (5.1%) contain methods with more than 5 parameters.

The histogram depicted by Figure 5.1b shows the distribution of the values for the maximum number of levels metric (NOL). The distribution of values is also positively skewed. The large majority of classes, namely 1513 out 1687 (89.7%), contain methods with less than 6 levels. 315 classes (18.7%) contain methods without levels. Only few classes (3.5%) contain methods with more than 7 levels.

The histogram depicted by Figure 5.1c shows the distribution of the values for the maximum number of locals in scope metric (NLS). The large majority of classes, namely 1455 out 1687 (86.2%), contain methods with less than 5 levels in scope. 1205 classes

(71.4%) contain methods without levels in scope. Only few classes (3.9%) contain methods with more than 11 locals in scope.



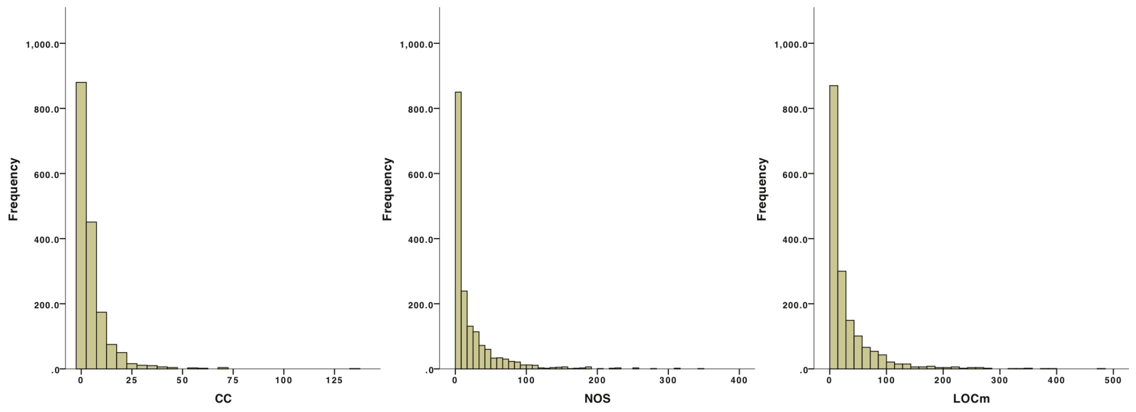
(a) Number of Parameters (b) Number of Levels (c) Number of Locals in Scope

Figure 5.1: Histograms of Number of Parameters, Levels, and Locals in Scope metrics (Tomcat)

The histogram depicted by Figure 5.2a shows the distribution of the values for the maximum cyclomatic complexity metric (CC). The distribution of values is positively skewed. The large majority of classes, namely 1221 out 1687 (72.4%), contain methods with less than 6 cyclomatic complexity. 315 classes (18.7%) contain methods without cyclomatic complexity. Only few classes (4.6%) contain methods with more than 20 cyclomatic complexity.

The histogram depicted by Figure 5.2b shows the distribution of the values for the maximum number of statements metric (NOS). The distribution of values is also positively skewed. The large majority of classes, namely 1294 out 1687 (76.7%), contain methods with less than 30 statements. 346 classes (20.5%) contain methods without statements. Only few classes (4.0%) contain methods with more than 100 statements.

The histogram depicted by Figure 5.2c shows the distribution of the values for the maximum lines of code metric (LOCm). The distribution of values is also positively skewed. The large majority of classes, namely 1367 out 1687 (81.1%), contain methods with less than 50 lines of code. 373 classes (22.1%) contain methods without lines of code. Only few classes (6.1%) contain methods with more than 100 lines of code.



(a) Cyclomatic Complexity (b) Number of Statements (c) Lines of Code in Method

Figure 5.2: Histograms of Cyclomatic Complexity, Number of Statements and Lines of Code metrics (Tomcat)

The histogram depicted by Figure 5.3 shows the distribution of the values for the maximum feature envy metric (FE). The large majority of classes, namely 1685 out 1687 (99.9%), contain methods with less than 3 feature envy problems. 1671 classes (99.1%) contain methods without feature envy's problems.

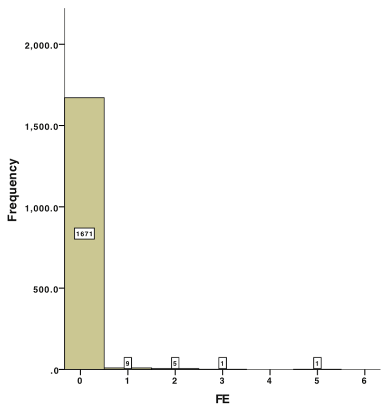


Figure 5.3: Histogram of Feature Envy metric (Tomcat)

Types' Metrics The Table 5.3 shows the descriptive statistics (median and maximum) of the types' source code metrics. The median is low for all metrics because for most of the types the metrics have value 0. This might happen because Tomcat is a large system which makes use of a lot of interface classes. Only a few number of classes reach the highest metrics' values.

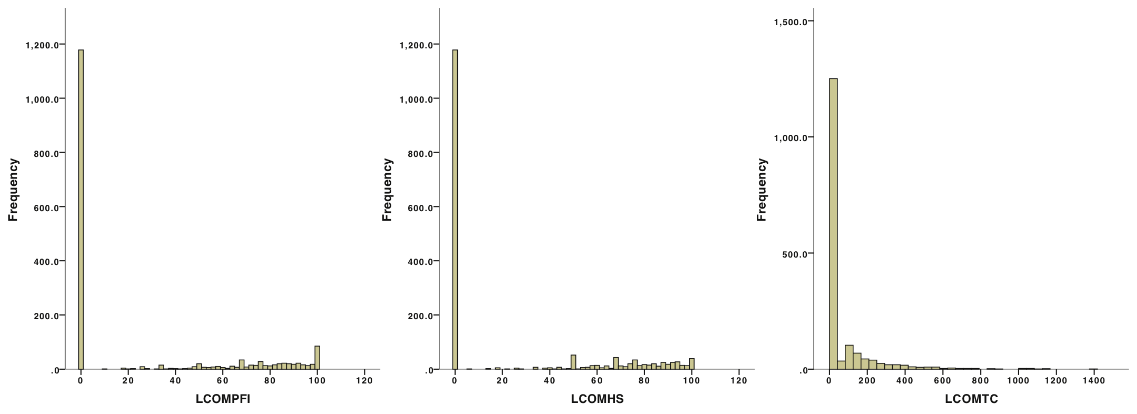
	NOF	LCPFI	LCTC	LCHS	LCCK	WMC	Ce
Median	1.00	0.00	0.00	0.00	0.00	6.00	6.00
Max.	104	100	1405	100	9025	689	99

Table 5.3: Median and maximum values of types' source code metrics for Tomcat

The histogram depicted by Figure 5.4a shows the distribution of the values for the Lack of Cohesion in Methods - Pairwise Fields Irrelation metric (LCPFI). This metric is measured in percentage. The large majority of classes, namely 1236 out 1687 (73.3%), contain less than 50% of the methods with lack of cohesion. 1178 classes (69.8%) contain only methods without lack of cohesion. Only few classes (14.8%) contain more than 80% of the methods with lack of cohesion.

The histogram depicted by Figure 5.4b shows the distribution of the values for the Lack of Cohesion in Methods - Henderson-Sellers metric (LCHS). This metric is also measured in percentage. This distribution is similar to the previous one and it has also the same number of classes with methods without lack of cohesion.

The histogram depicted by Figure 5.4c shows the distribution of the values for the Lack of Cohesion in Methods - Total Correlation metric (LCTC). The large majority of classes, namely 1492 out 1687 (88.4%), contain less than 200 lack of cohesion. 1213 classes (71.9%) contain only methods without lack of cohesion. Only few classes (1.7%) contain lack of cohesion greater than 600.



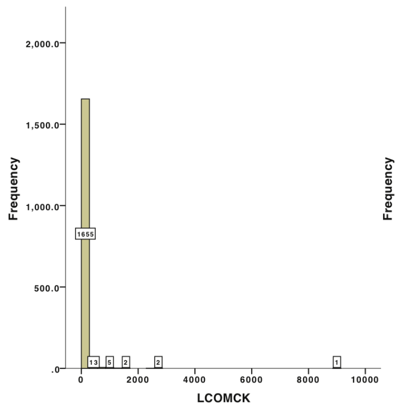
(a) Pairwise Field Irrelation (b) Henderson-Sellers (c) Total Correlation

Figure 5.4: Histograms of Lack of Cohesion in Methods (PFI, HS, TC) metric (Tomcat)

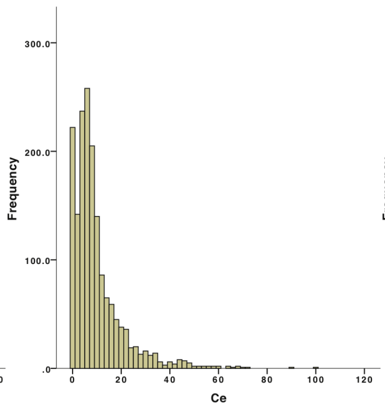
The histogram depicted by Figure 5.5a shows the distribution of the values for the Lack of Cohesion in Methods - Chidamber & Kemerer metric (LCCK). The large majority of classes, namely 1602 out 1687 (95.0%), contain less than 50 lack of cohesion. 1437 classes (85.2%) contain only methods without lack of cohesion. Only few classes (2.4%) contain lack of cohesion greater than 200.

The histogram depicted by Figure 5.5b shows the distribution of the values for the Efferent Couplings metric (Ce). The large majority of classes, namely 1132 out 1687 (67.1%), contain less than 10 efferent couplings. 222 classes (13.2%) have not efferent couplings. Only few classes (1.1%) contain more than 50 efferent couplings.

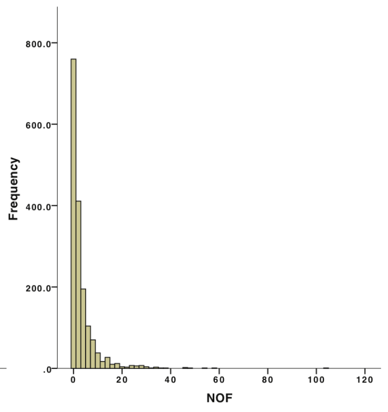
The histogram depicted by Figure 5.5c shows the distribution of the values for the number of fields metric (NOF). The large majority of classes, namely 1366 out 1687 (81.0%), contain less than 5 fields. 760 classes (45.1%) have not fields. Only few classes (6.5%) contain more than 10 fields.



(a) Henderson-Sellers



(b) Efferent Couplings Histogram



(c) Number of Levels

Figure 5.5: Histograms of Lack of Cohesion in methods (CK), Efferent Couplings and Number of Levels metrics (Tomcat)

The histogram depicted by Figure 5.6 shows the distribution of the values for the weighted methods per class metric (WMC). The large majority of classes, namely 1222 out 1687 (72.4%), contain less than 20 weighted methods. 313 classes (18.6%) have not weighted methods. Only few classes (10%) contain more than 64 weighted methods.

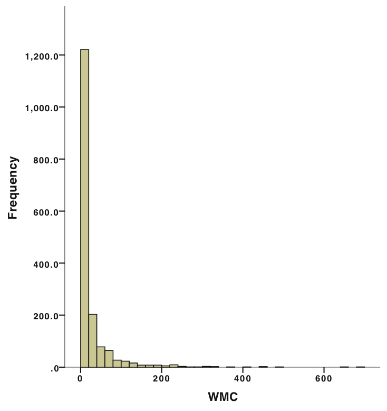


Figure 5.6: Histogram of Weighted Methods Per Class metric (Tomcat)

5.3.1.2 Warning's categories

In this subsection we present the study of FindBugs warnings' distribution for Tomcat system. We show histograms and descriptive statistics.

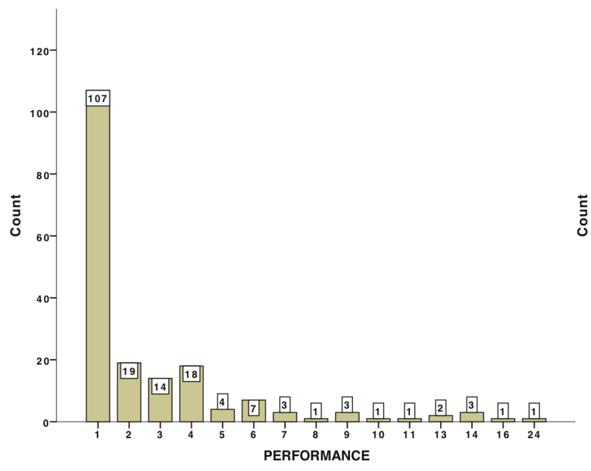
The Table 5.4 shows the descriptive statistics (median, maximum, sum and rate) of the warnings' categories. There are a total of 1584 warnings in Tomcat. The median is 0 for all warnings' categories because most classes don't have warnings. Therefore, we only select the classes which have warnings. Warning categories include Security (Sec), Performance (Perf), Bad Practice (Bad P.), Multithreaded Correctness (MT Corr), Malicious Code vulnerability (M. Code), Correctness (Corr), Style (Style), and Experimental (Exp).

	Sec	**Perf**	**Bad P.**	**MT Corr**	**M. Code**	**Corr**	**Style**	**Exp**
Median	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Max.	2	24	31	13	13	5	13	1
Sum	6	506	219	121	452	83	180	17
Rate	0.4%	32.1%	13.9%	7.7%	28.5%	5.3%	11.4%	1.0%

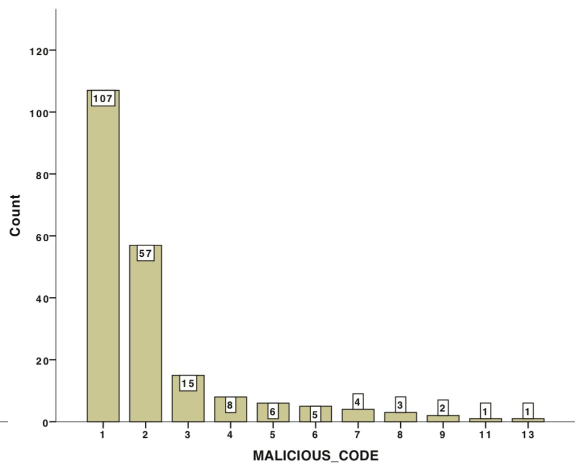
Table 5.4: Median, maximum, total and rate values of warnings' categories (Tomcat)

The histogram depicted by Figure 5.7a shows the distribution of the values for the performance warnings. The distribution of values is positively skewed. FindBugs reports 506 performance warnings in 185 classes (11.0%).

The histogram depicted by Figure 5.7b shows the distribution of the values for the malicious code vulnerability warnings. The distribution of values is also positively skewed. FindBugs reports 452 malicious code vulnerability warnings in 209 classes (12.4%)



(a) Performance

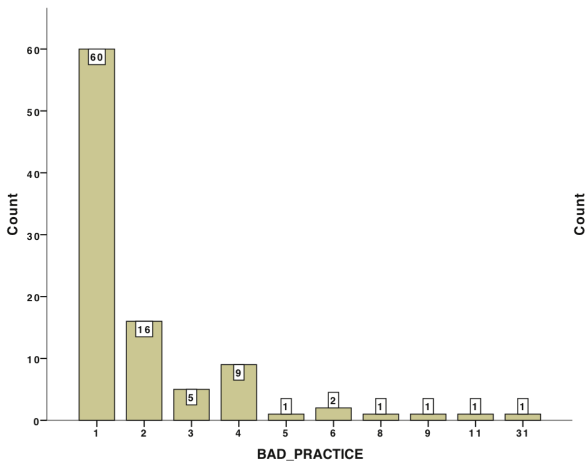


(b) Malicious Code

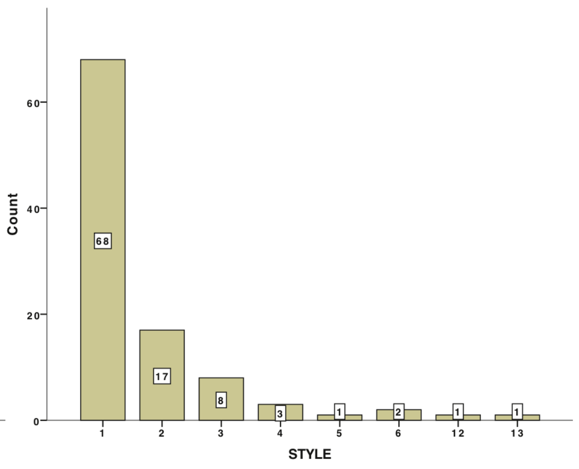
Figure 5.7: Histograms of Performance and Malicious Code warnings' categories (Tomcat)

The histogram depicted by Figure 5.8a shows the distribution of the values for the bad practice warnings. The distribution of values is also positively skewed. FindBugs reports 219 bad practice warnings in 97 classes (5.7%).

The histogram depicted by Figure 5.8b shows the distribution of the values for the style warnings. The distribution of values is also positively skewed. FindBugs reports 180 style warnings in 101 classes (6.0%).



(a) Bad Practice

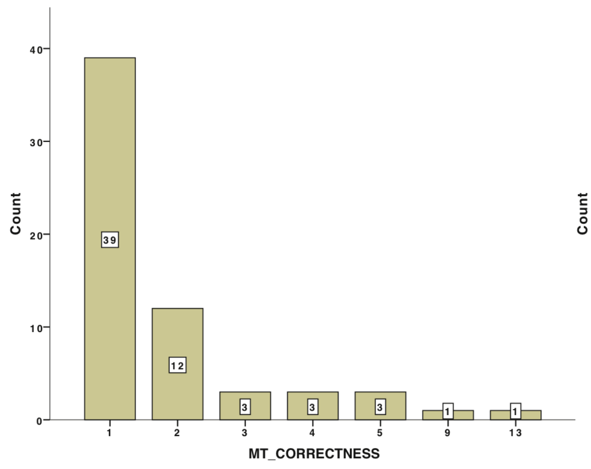


(b) Style

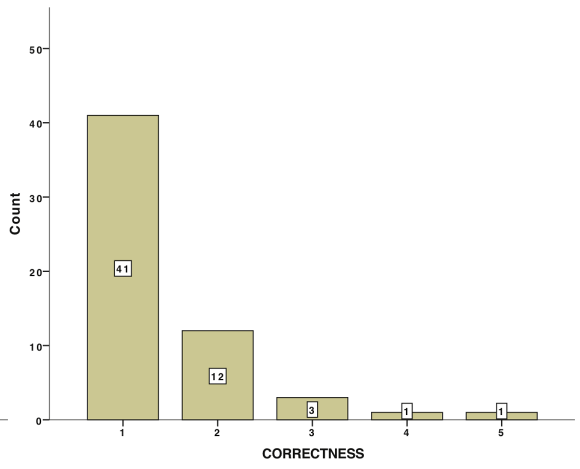
Figure 5.8: Histograms of Bad Practice and Style warnings' categories (Tomcat)

The histogram depicted by Figure 5.9a shows the distribution of the values for the Multithreaded correctness warnings. The distribution of values is also positively skewed. FindBugs reports 121 Multithreaded correctness warnings in 62 classes (3.7%).

The histogram depicted by Figure 5.9b shows the distribution of the values for the correctness warnings. The distribution of values is also positively skewed. FindBugs reports 83 correctness warnings in 58 classes (3.6%).



(a) MT Correctness

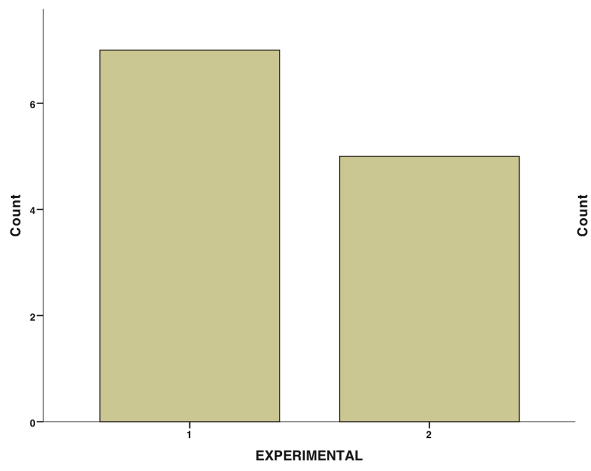


(b) Correctness

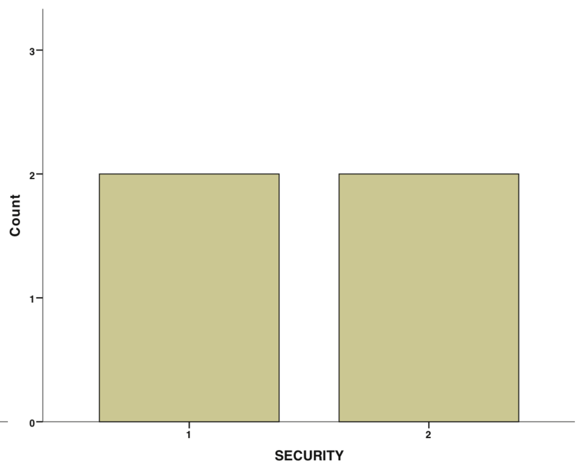
Figure 5.9: Histograms of MT Correctness and Correctness warnings' categories (Tomcat)

The histogram depicted by Figure 5.10a shows the distribution of the values for the experimental warnings. FindBugs reports 17 experimental warnings in 12 classes (0.8%). There are 7 classes with 1 experimental warning and 2 classes with 5 experimental warnings.

The histogram depicted by Figure 5.10b shows the distribution of the values for the security warnings. FindBugs reports 6 security warnings in 4 classes (0.2%). There are 2 classes with 2 security warnings and 2 classes with 1 security warning.



(a) Experimental



(b) Security

Figure 5.10: Histograms of Experimental and Security warnings' categories (Tomcat)

5.3.1.3 Relationship between source code metrics and warning's categories

The Tables 5.5 and 5.6 show the correlations' coefficients between source code metrics and warnings' categories. The orange colored cells mean that there is a correlation and the cyan ones designate an indication of a possible correlation.

Most classes do not have warnings from any category resulting in too much zero-values. Hence, for each warning category, we compute the Spearman correlation only for classes that contain at least 1 warning.

	NOL	**NOP**	**FE**	**NOS**	**CC**	**LOCm**	**NLS**
Correctness	-0.050	-0.118		0.112	-0.006	0.129	0.158
Security	0.000	-0.707		0.000	0.000	0.000	0.000
Performance	0.442**	0.057	-0.060	0.508**	0.469**	0.487**	0.390**
Bad Practice	0.229*	0.194	-0.077	0.356**	0.343**	0.396**	0.275**
MT Correctness	0.210	-0.100		0.188	0.247	0.188	0.102
Malicious Code	0.047	0.011		0.122	0.054	0.091	0.099
Style	0.228*	0.157		0.235*	0.203*	0.267**	0.258**
Experimental	0.615*	0.025		0.392	0.467	0.367	0.640*

Table 5.5: Spearman correlation between methods' metrics and warnings' categories (Tomcat)

	NOF	**LCPFI**	**LCTC**	**LCHS**	**LCCK**	**WMC**	**Ce**
Correctness	0.043	-0.190	0.094	-0.078	-0.187	-0.094	0.047
Security	0.894	0.000	0.894	0.000	0.894	0.447	0.000
Performance	0.344**	0.409**	0.404**	0.423**	0.341**	0.538**	0.491**
Bad Practice	0.299**	0.250*	0.177	0.245*	0.350**	0.400**	0.374**
MT Correctness	0.196	-0.012	0.248	0.055	0.022	0.148	0.098
Malicious Code	0.222**	0.145*	0.211**	0.233**	0.198**	0.167*	0.050
Style	0.096	0.094	0.129	0.056	0.071	0.305**	0.263**
Experimental	0.247	0.395	0.271	0.247	0.078	0.270	0.393

Table 5.6: Spearman correlation between types' metrics and warnings' categories (Tomcat)

* Correlation is significant at the 0.05 level (2-tailed).

* Correlation is significant at the 0.01 level (2-tailed).

- **Methods' metrics and warnings' categories**
  - An High *number of parameters* may decrease project's maintainability. Nevertheless, this metric has not a relationship with any warning category.
  - As few classes have *feature envy* problem, this metric should also not be considered for correlation analysis.

- A large *Number Of Levels* increases source code complexity. This metric has a significant correlation (coefficient 0.442) with Performance warnings meaning that classes containing higher number of levels, have more performance warnings. This makes sense since source code complexity may decrease program's performance. An high number of levels also reduces comprehensibility and this metric has an indication of possible correlation (coefficient 0.228) with Style and Bad Practice warnings.
- An higher *number of statements* is not in itself a bad thing. It does suggest the possibility of increasing the level of abstraction and deepens semantics. However, in this project, this metric has a relationship with performance warnings (correlation coefficient 0.508). It means that classes with many statements are related to the decrease of the program's performance or these warnings are most false-positives. The number of statements metric has also a relationship with Bad Practice and Style warnings which is understandable due the lower level of abstraction.
- *Cyclomatic Complexity* metric is correlated with Performance and Bad Practice warning categories (0.469 and 0.396 correlation coefficients respectively). This result is interesting because high values of cyclomatic complexity actually decrease the program's performance.
- Long methods tend to be more warning-prone and complex than shorter ones. Therefore, it is understandable that *Lines of Codes in methods* metric has a correlation with Performance and Bad Practice warnings (correlation coefficients 0.487 and 0.396 respectively).
- A large *Number Of Locals In Scope* increases complexity and reduces comprehensibility. This metric has significative correlation with Performance, Bad Practice and Style (correlation coefficients 0.390, 0.227 and 0.258 respectively) warning categories which also makes sense.

• **Types' metrics and warnings' categories**

- Although a large *number of fields* is not necessarily an indication of bad code, it does suggest the possibility improving the semantics of the object model. However, in this project, this metric has a relationship with performance warnings (correlation coefficient 0.344). It means that classes with many fields may decrease program's performance or these warnings are most false-positives. Number of fields metric has also a relationship with Bad Practice and Malicious code vulnerabilities warnings which is understandable due the lower level of abstraction.

  - *Cohesion* indicates whether a class represents a single abstraction or multiple abstractions. All four lack of cohesion metric definitions are related with Performance and Malicious Code warning categories.
  - *Weighted Methods Per Class* metric represents the complexity of a class as a whole. Classes with a large Weighted Methods Per Class value can often be refactored into two or more classes. This metric has a strong correlation with Performance warning category (correlation coefficient 0.538). It has also a relationship with Bad Practice and Style Warnings.
  - *Efferent Couplings* A large efferent coupling can indicate that a class is unfocussed and also may indicate brittleness, since it depends on the stability of all the types to which it is coupled. It has correlation with Performance, Bad Practice and Style warning categories.

- **Warnings' categories and source-code metrics**
  - There are some high correlation coefficients for *security* warning category. Still, by having only 6 warnings of this category, the correlation analysis is not significant.
  - The *Experimental* warning category is also not considered due the low number of occurrences of this kind of warning.
  - The *Correctness* warnings have low correlation coefficients (< 0.2) for all source metrics. Consequently, there is no relationship between this warning category and any source code metric.
  - The *performance* warnings are the most frequent ones (32%). This warning category has correlation coefficients higher than 0.3 for most source code metrics (12 out 14). Only feature envy and number of parameters metrics are not related with performance warnings. The number of parameters metric has almost zero correlation coefficient. It makes sense since a class has as more parameters, it may not decrease the program's performance.
  - The *Bad practice* warnings have significant correlation with 8 source code metrics (NOS, CC, LOCm, NLS, NOF, LCOMCK, WMC, and Ce).
  - The *Multithreaded Correctness* warnings have low correlation coefficients ($<$ 0.25) for all source metrics. Consequently, there is no relationship between this warning category and any source code metric.
  - The *Malicious Code Vulnerability* warning category has a relationship with some types' metrics (Lack of cohesion and number of fields)
  - The *Style* warning category has a significant correlation coefficients for LOCm, NLS, WMC, and Ce source code metrics. It has also an indication of possible correlation for NOL, NOS, and CC metrics.

5.3.2 Axis

5.3.2.1 Source code metrics

In this subsection we present the study of source code metrics' distribution. We present histograms and descriptive statistics. Axis has a total of 840 classes and 891 warnings. We start by analyzing methods' metrics followed by types' metrics.

Methods' Metrics The Table 5.7 shows the descriptive statistics (median and maximum) of the methods' source code metrics. The median is low for all metrics because for most of the methods the metrics have value 0. This might happen because Axis is a large system which makes use of a lot of abstract methods. Only a few number of methods reach the highest metrics' values.

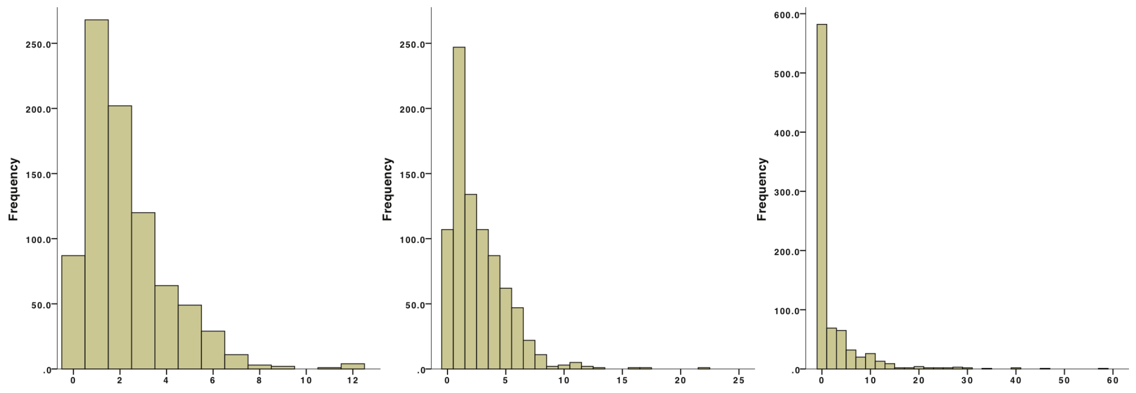
	NOL	**NOP**	**FE**	**NOS**	**CC**	**LOCm**	**NLS**
Median	2.00	2.00	0.00	8.00	2.00	15.00	0.00
Max.	22	12	4	314	60	452	57

Table 5.7: Median and maximum values of methods' source code metrics for Axis

The histogram depicted by Figure 5.11a shows the distribution of the values for the maximum number of parameters metric (NOP). The distribution of values is positively skewed. The large majority of classes, namely 677 out 840 (80.6%), contain methods with less than 4 parameters. 87 classes (10.4%) contain methods without parameters. Only few classes (2.5%) contain methods with more than 6 parameters.

The histogram depicted by Figure 5.11b shows the distribution of the values for the maximum number of levels metric (NOL). The distribution of values is also positively skewed. The large majority of classes, namely 682 out 840 (81.2%), contain methods with less than 5 levels. 107 classes (12.7%) contain methods without levels. Only few classes (3.2%) contain methods with more than 7 levels.

The histogram depicted by Figure 5.11c shows the distribution of the values for the maximum number of locals in scope metric (NLS). The large majority of classes, namely 716 out 840 (85.2%), contain methods with less than 5 levels in scope. 582 classes (69.3%) contain methods without levels in scope. Only few classes (2.9%) contain methods with more than 14 locals in scope.



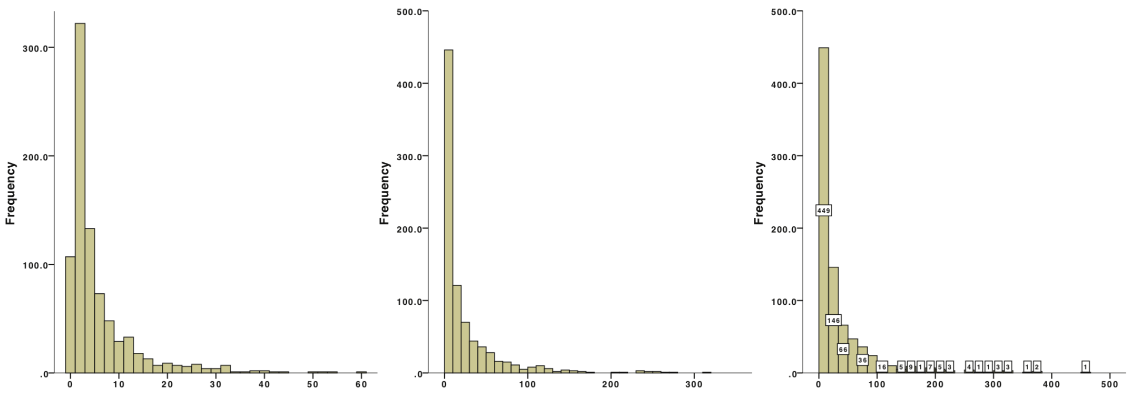
(a) Number of Parameters (b) Number of Levels (c) Number of Locals in Scope

Figure 5.11: Histograms of Number of Parameters, Levels, and Locals in Scope metrics (Axis)

The histogram depicted by Figure 5.12a shows the distribution of the values for the maximum cyclomatic complexity metric (CC). The distribution of values is positively skewed. The large majority of classes, namely 562 out 840 (66.9%), contain methods with less than 5 cyclomatic complexity. 107 classes (12.7%) contain methods without cyclomatic complexity. Only few classes (7.6%) contain methods with more than 16 cyclomatic complexity.

The histogram depicted by Figure 5.12b shows the distribution of the values for the maximum number of statements metric (NOS). The distribution of values is also positively skewed. The large majority of classes, namely 579 out 840 (68.9%), contain methods with less than 21 statements. 115 classes (13.7%) contain methods without statements. Only few classes (5.5%) contain methods with more than 100 statements.

The histogram depicted by Figure 5.12c shows the distribution of the values for the maximum lines of code metric (LOCm). The distribution of values is also positively skewed. The large majority of classes, namely 579 out 840 (68.9%), contain methods with less than 30 lines of code. 125 classes (14.9%) contain methods without lines of code. Only few classes (8.6%) contain methods with more than 100 lines of code.



(a) Cyclomatic Complexity (b) Number of Statements (c) Lines of Code in Method

Figure 5.12: Histograms of Cyclomatic Complexity, Number of Statements and Lines of Code metrics (Axis)

The histogram depicted by Figure 5.13 shows the distribution of the values for the maximum feature envy metric (FE). The large majority of classes, namely 836 out 840 (99.5%), contain methods with less than 2 feature envy problems. 827 classes (98.5%) contain methods without feature envy's problems.

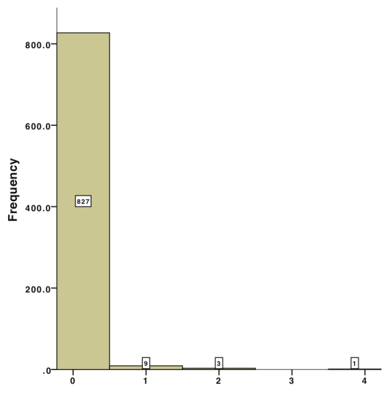


Figure 5.13: Histogram of Feature Envy metric (Axis)

Types' Metrics The Table 5.8 shows the descriptive statistics (median and maximum) of the types' source code metrics. The median is low for all metrics because for most of the types the metrics have value 0. This might happen because Axis is a large system which makes use of a lot of interface classes. Only a few number of classes reach the highest metrics' values.

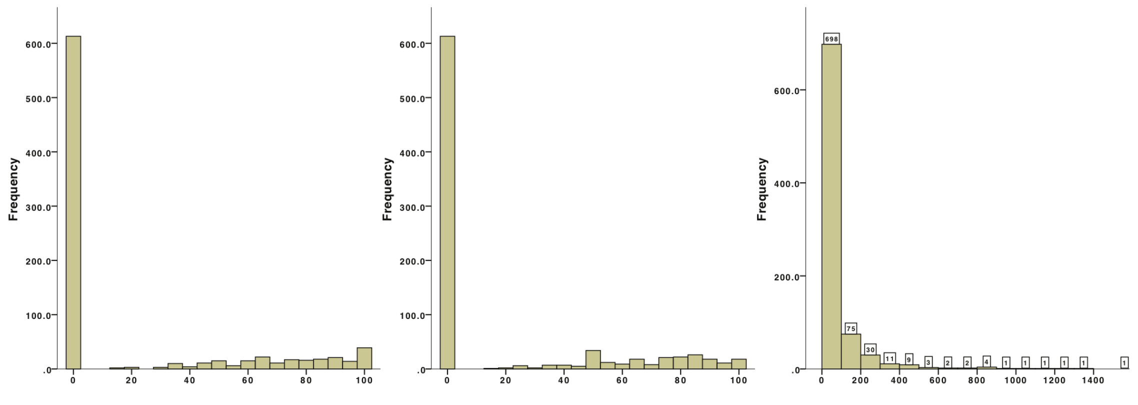
	NOF	LCPFI	LCTC	LCHS	LCCK	WMC	Ce
Median	1.00	0.00	0.00	0.00	0.00	6.00	7.00
Max.	48	100	2230	100	1647	459	94

Table 5.8: Median and maximum values of types' source code metrics for Axis

The histogram depicted by Figure 5.14a shows the distribution of the values for the Lack of Cohesion in Methods - Pairwise Fields Irrelation metric (LCPFI). This metric is measured in percentage. The large majority of classes, namely 633 out 840 (75.4%), contain less than 40% of the methods with lack of cohesion. 613 classes (73.0%) contain only methods without lack of cohesion. Only few classes (15.6%) contain more than 70% of the methods with lack of cohesion.

The histogram depicted by Figure 5.14b shows the distribution of the values for the Lack of Cohesion in Methods - Henderson-Sellers metric (LCHS). This metric is also measured in percentage. This distribution is similar to the previous one and it has also the same number of classes with methods without lack of cohesion.

The histogram depicted by Figure 5.14c shows the distribution of the values for the Lack of Cohesion in Methods - Total Correlation metric (LCTC). The large majority of classes, namely 803 out 840 (95.6%), contain less than 300 lack of cohesion. 642 classes (76.4%) contain only methods without lack of cohesion. Only few classes (2.0%) contain lack of cohesion greater than 500.



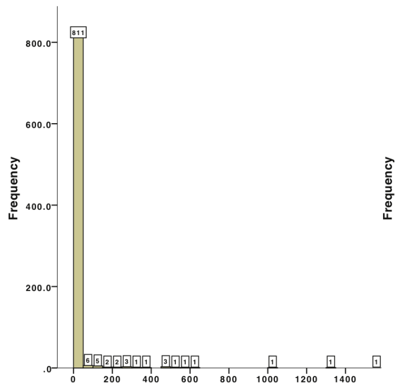
(a) Pairwise Field Irrelation (b) Henderson-Sellers (c) Total Correlation

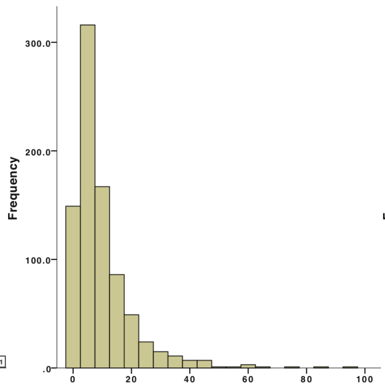
Figure 5.14: Histograms of Lack of Cohesion in Methods (PFI, HS, TC) metric (Axis)

The histogram depicted by Figure 5.15a shows the distribution of the values for the Lack of Cohesion in Methods - Chidamber & Kemerer metric (LCCK). The large majority of classes, namely 817 out 840 (97.3%), contain less than 100 lack of cohesion. 731 classes (87.0%) contain only methods without lack of cohesion. Only few classes (1.0%) contain lack of cohesion greater than 400.

The histogram depicted by Figure 5.15b shows the distribution of the values for the Efferent Couplings metric (Ce). The distribution of values is positively skewed. The large majority of classes, namely 680 out 840 (80.8%), contain less than 15 efferent couplings. 96 classes (11.4%) have not efferent couplings. Only few classes (1.4%) contain more than 45 efferent couplings.

The histogram depicted by Figure 5.15c shows the distribution of the values for the number of fields metric (NOF). The distribution of values is also positively skewed. The large majority of classes, namely 742 out 840 (88.3%), contain less than 5 fields. 405 classes (48.2%) have not fields. Only few classes (4.7%) contain more than 10 fields.


(a) Henderson-Sellers


(b) Efferent Couplings Histogram

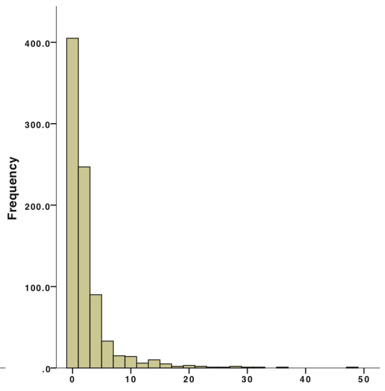

(c) Number of Levels

Figure 5.15: Histograms of Lack of Cohesion in methods (CK), Efferent Couplings and Number of Levels metrics (Axis)

The histogram depicted by Figure 5.16 shows the distribution of the values for the weighted methods per class metric (WMC). The distribution of values is positively skewed. The large majority of classes, namely 695 out 840 (82.7%), contain less than 30 weighted methods. 107 classes (12.7%) have not weighted methods. Only few classes (3.7%) contain more than 100 weighted methods.

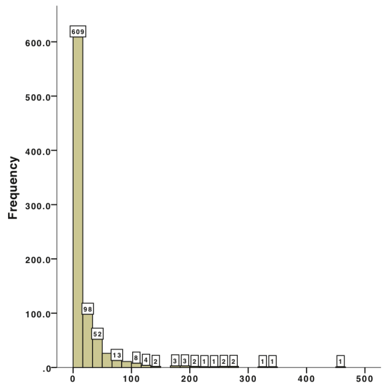


Figure 5.16: Histogram of Weighted Methods Per Class metric (Axis)

5.3.2.2 Warning's categories

In this subsection we present the study of FindBugs warnings' distribution for Axis system. We show histograms and descriptive statistics.

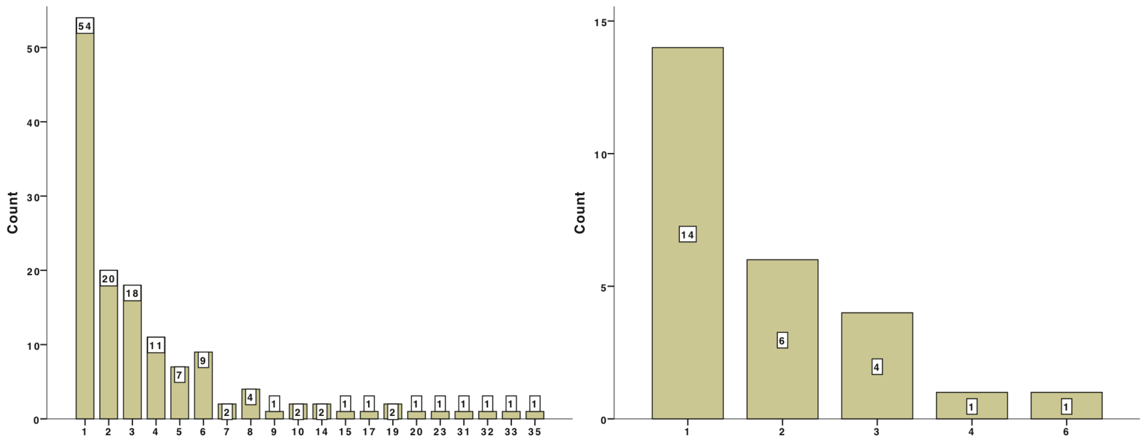
The Table 5.9 shows the descriptive statistics (median, maximum, sum and rate) of the warnings' categories. There are a total of 840 warnings in Axis. The median is 0 for all warnings' categories because most classes don't have warnings. Therefore, we only select the classes which have warnings. Warning categories include Performance (Perf), Bad Practice (Bad P.), Multithreaded Correctness (MT Corr), Malicious Code vulnerability (M. Code), Correctness (Corr), Style (Style), and Experimental (Exp).

	Perf	Bad P.	MT Corr	M. Code	Corr	Style	Exp
Median	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Max.	35	12	4	6	5	2	1
Sum	628	112	10	48	18	72	3
Rate	70.5%	12.6%	1.1%	5.4%	2.0%	8.1%	0.3%

Table 5.9: Median, maximum, total and rate values of warnings' categories (Axis)

The histogram depicted by Figure 5.17a shows the distribution of the values for the performance warnings. The distribution of values is positively skewed. FindBugs reports 628 performance warnings in 140 classes (16.7%).

The histogram depicted by Figure 5.17b shows the distribution of the values for the malicious code vulnerability warnings. The distribution of values is also positively skewed. FindBugs reports 48 malicious code vulnerability warnings in 26 classes (3.1%)



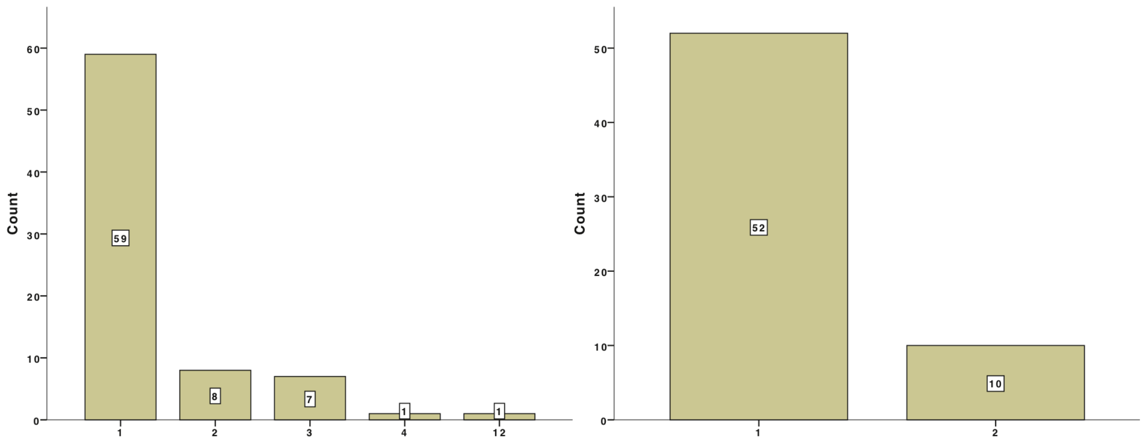
(a) Performance

(b) Malicious Code

Figure 5.17: Histograms of Performance and Malicious Code warnings' categories (Axis)

The histogram depicted by Figure 5.18a shows the distribution of the values for the bad practice warnings. The distribution of values is also positively skewed. FindBugs reports 112 bad practice warnings in 76 classes (9.0%).

The histogram depicted by Figure 5.18b shows the distribution of the values for the style warnings. The distribution of values is also positively skewed. FindBugs reports 72 style warnings in 62 classes (7.4%). There are 52 classes with just 1 style warning and 10 classes with 2 style warnings.

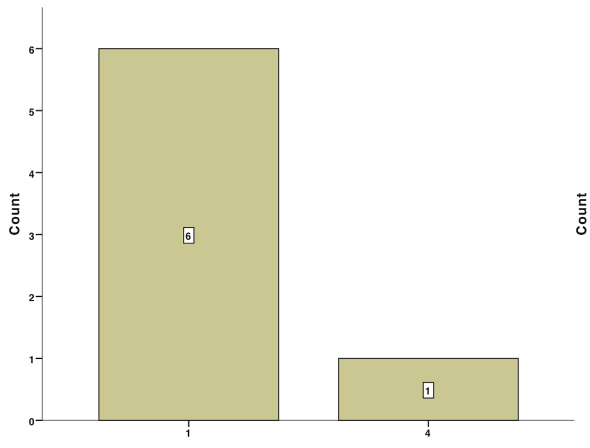


(a) Bad Practice (b) Style

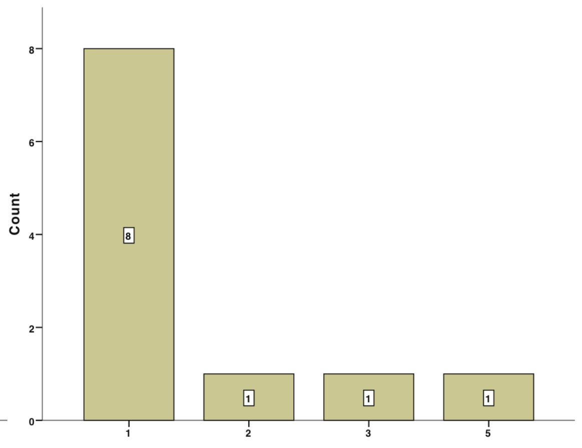
Figure 5.18: Histograms of Bad Practice and Style warnings' categories (Axis)

The histogram depicted by Figure 5.19a shows the distribution of the values for the Multithreaded correctness warnings. The distribution of values is also positively skewed. FindBugs reports 10 Multithreaded correctness warnings in 7 classes (0.8%). There are 6 classes with one experimental warning and 1 class with 4 experimental warnings.

The histogram depicted by Figure 5.19b shows the distribution of the values for the correctness warnings. The distribution of values is also positively skewed. FindBugs reports 18 correctness warnings in 11 classes (1.3%).



(a) MT Correctness



(b) Correctness

Figure 5.19: Histograms of MT Correctness and Correctness warnings' categories (Axis)

The histogram depicted by Figure 5.20 shows the distribution of the values for the experimental warnings. FindBugs reports 3 experimental warnings in 3 classes (0.4%) and each class has just one experimental warning

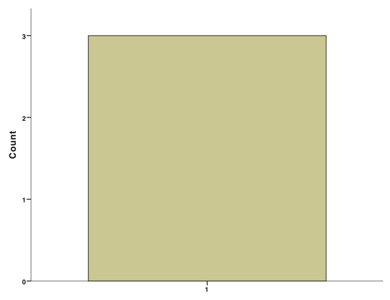


Figure 5.20: Histogram of Experimental Warnings (Axis)

5.3.2.3 Relationship between source code metrics and warning's categories

The Tables 5.10 and 5.11 show the correlations' coefficients between source code metrics and warnings' categories. The orange colored cells mean that there is a correlation and the cyan ones designate an indication of a possible correlation.

Most classes do not have warnings from any category resulting in too much zero-values. Hence, for each warning category, we compute the Spearman correlation only for classes that contain at least one warning.

	NOL	NOP	FE	NOS	CC	LOCm	NLS
Correctness	0.256	-0.121		0.104	0.061	-0.023	0.801**
Performance	0.482**	0.487**	-0.142	0.458**	0.505**	0.471**	0.356**
Bad Practice	0.103	-0.030	–0.087	0.110	0.173	0.125	0.216
MT Correctness	0.208	0.113		0.309	0.204	0.000	0.107
Malicious Code	0.221	0.123		0.234	0.210	0.128	0.008
Style	0.262*	0.165		0.258*	0.319*	0.277*	0.450**
Experimental							

Table 5.10: Spearman Correlation between methods' metrics and warnings' categories (Axis)

	NOF	LCPFI	LCTC	LCHS	LCCK	WMC	Ce
Correctness	0.645*	0.473	0.845**	0.599	0.189	0.330	0.439
Performance	0.475**	0.445**	0.434**	0.473**	0.390**	0.540**	0.467**
Bad Practice	0.254*	0.336**	0.337**	0.283*	0.264*	0.222	0.140
MT Correctness	0.206	0.412	0.206	0.624	0.635	0.204	0.424
Malicious Code	0.124	0.334	0.092	0.217	0.249	0.379	0.151
Style	0.098	0.118	0.139	0.115	-0.036	0.335**	0.305**
Experimental							

Table 5.11: Spearman Correlation between types' metrics and warnings' categories (Axis)

* Correlation is significant at the 0.05 level (2-tailed).

* Correlation is significant at the 0.01 level (2-tailed).

- **Methods' metrics and warnings' categories**

  – Classes with high *number of parameters* may decrease project's maintainability. However, there is a relationship between the number of parameters and performance warnings (coefficient 0.487).

  – As few classes have *feature envy* problem, this metric should also not be considered for correlation analysis.

  – A large *Number Of Levels* increases source code complexity. This metric has a significant correlation (coefficient 0.482) with Performance warnings meaning that classes containing higher number of levels usually have more performance warnings. This makes sense since source code complexity might may program's performance. Higher number of levels also reduces comprehensibility and this metric has an indication of possible correlation (coefficient 0.228) with Style warnings.

  - An higher *number of statements* is not in itself a bad thing. It does suggest the possibility of increasing the level of abstraction and deepens semantics. However, in this project, this metric has a relationship with performance warnings (correlation coefficient 0.458). It means that classes with many statements have usually more performance warnings. Number of statements metric has also a relationship with Style warnings which is understandable due the lower level of abstraction.
  - The *Cyclomatic Complexity* metric is correlated with Performance and Style warning categories. This result is interesting because high values of cyclomatic complexity actually decrease the program's performance. Therefore we should avoid methods with high cyclomatic complexity.
  - Long methods tend to be more warning-prone and complex than shorter ones. Therefore, it is understandable that *Lines of Codes in methods* metric has a correlation with Performance and Style warnings.
  - A large *Number Of Locals In Scope* increases complexity and reduces comprehensibility. This metric has significative correlation with Performance, Style warning categories which also makes sense. Number of Locals in Scope metric has a **strong** relationship with Correctness warnings category (correlation coefficient 0.801)

- **Types' metrics and warnings' categories**
  - Although a large *number of fields* is not necessarily an indication of bad code, it does suggest the possibility of improving the semantics of the object model. However, in this project, this metric has a relationship with performance warnings (correlation coefficient 0.475). It means that classes with many fields may decrease program's performance or these warnings are most false-positives. Number of fields metric has also a relationship with Bad Practice warnings which is understandable due the lower level of abstraction. This metric also indicates a possible correlation with Correctness warning category (coefficient 0.645).
  - *Cohesion* indicates whether a class represents a single abstraction or multiple abstractions. All four lack of cohesion metric definitions are related with Performance and Bad Practice warning categories. There is also a **strong** correlation between this metric and Correctness warning cateogry.
  - *Weighted Methods Per Class* metric represents the complexity of a class as a whole. Classes with a large Weighted Methods can often be refactored into two or more classes. This metric has a **strong** correlation with Performance

warning category (correlation coefficient 0.540). It has also a relationship with Style Warnings.

- A large *Efferent Couplings* can indicate that a class is unfocussed and also may indicate brittleness, since it depends on the stability of all the types to which it is coupled. It has correlation with Performance, and Style warning categories.

- **Warnings' categories and source-code metrics**
  - *Experimental* warning category is not considered due the low number of occurrences (3) of this kind of warning.
  - *Correctness* warnings have high correlation coefficients with Number of Locals in Scope and Lack of Cohesion in Methods (Total Correlation) metrics.
  - The *performance* warnings are the most frequent ones (70.5%). This warning category has correlation coefficients higher than 0.35 for most source code metrics (13 out 14). Only feature envy metric are not related with performance warnings.
  - *Bad practice* warnings have correlation with the four lack of cohesion metrics and also Number of fields.
  - *Multithreaded Correctness* warnings have low number of occurrences (10). Consequently, it is not possible to evaluate a relationship between this warning category and any source code metric.
  - *Malicious Code Vulnerability* warning category have not a relationship with any source code metric.
  - *Style* warnings high correlation coefficients for Number of Locals in Scope and Weighted Method Per Class metrics. It has also an indication of a possible correlation with NOL, NOS, CC, LOCm and Ce metrics.

5.4 Answers to research questions

Based on the results from the Tomcat and Axis case studies, we can give the following answers to our research questions:

- *Is there any correlation between source code metrics and FindBugs [cit10f] warnings' categories?*
  The answer to this research question is yes. The highest correlation (0.4-0.54) is between Performance and source code metrics. Furthermore, there is significant but low correlation between Bad Practice and Style warnings and some of the other source code metrics.

- *Are the same correlations significant across projects?*
  The results show that this is only partly true for Tomcat and Axis: the result sets for Tomcat and Axis systems are similar. Performance warning category has significative correlation coefficients for all source code metrics except Number of Parameters and Feature Envy metrics. Style warnings have also a relationship with Number of Statements and Weighted Methods Per Class metrics. We have also found that some metrics such as Feature Envy and Number of Parameters are not useful to study this correlation. Security, Multithreaded, and Experimental warning categories don't have a correlation with any metric for both case studies.

5.5 Threats to Validity

Like any empirical evaluation, there are some threats to validity which must be taken into consideration in our experiment. Though we use Tomcat and Axis as our case studies, it does not cover all bug patterns in FindBugs. The subject programs might not be representative. It is better to choose several various large-scale projects as case studies set to cover as many bug patterns as possible to improve the correlation analysis. Since we only evaluated the correlation analysis on two cases, and their results differed, we cannot extract a generic set of correlations for other projects.

Some developers may use FindBugs in their development cycle which may influence the results outcome. It is also possible that some programmers in different projects make, find or remove certain types of errors.

Moreover, we do not manually inspect error reports in Tomcat, and Axis. We do not find out the false positives and true error reports. A solution is to collect feedback from the developers of the case studies because their designations of error reports are accurate.

There are two further major factors that were not under our control: (1) the application domain; and (2) the development team.

It is possible to apply our approach to many other projects since they are written in Java and importable/compilable with Eclipse IDE.

5.6 Summary

To perform our experiments, we have worked with 2 Java open-source systems, Tomcat and Axis.

We have adopted Eclipse IDE to import these projects. We have computed the set of warnings using eclipse plugin of the FindBugs tool. We have also computed the source code metrics using State Of Flow Eclipse Metrics (plugin). We have developed a python script which filters and merges all the desired data by class. We have done the statistics analysis using PASW Statistics tool.

We found that most source code metrics used in our study are correlated with Performance and Style warning categories. The result sets for Tomcat and Axis systems are similar. Performance warning category has significative correlation coefficients for all source code metrics except Number of Parameters and Feature Envy metrics. Style warnings have also a relationship with Number of Statements and Weighted Methods Per Class metrics. We have also found that some metrics such as Feature Envy and Number of Parameters are not useful to study this correlation. Security, Multithreaded, and Experimental warning categories don't have a correlation with any metric for both case studies.

Although we have used two case studies, it does not cover all bug patterns in FindBugs. Also, the subject programs may not be representative. Some developers may have used FindBugs in their development cycle which may influence the results outcome. We do not manually inspect error reports in Tomcat and Axis to find out the false positives and true error reports. It is possible to apply our approach to many other projects since they are written in Java and importable/compilable with Eclipse IDE.

Chapter 6

Conclusions and Future Work

In Section 6.1, it is presented our contributions as well as a summary of our findings. Finally, in Section 6.2, we present the guidelines for the future work.

6.1 Conclusions

In this thesis, we have discussed an approach that studies the correlation between source code metrics and warnings' categories. We have applied this approach to two open-source projects, Tomcat and Axis. We have computed the set of warnings using eclipse plugin of the FindBugs tool. We have also computed the source code metrics using State Of Flow Eclipse Metrics (plugin). We have done the statistics analysis using PASW Statistics tool. The results from our two case studies indicate that Performance warning category and most source code metrics have the highest correlation (0.4-0.54). There is also a significant but low correlation between Bad Practice and Style warnings and some of the other source code metrics.

This thesis makes the following **contributions**:

- *A correlation between source code metrics and warning categories approach for developing a more advanced model for ranking warnings*. This includes an approach to observe the warnings output by one bug-finding tool for two subject programs, collecting its source code metrics, and a study of statistics analysis (Section 3.3).
- *Analysis of empirical data*. The results are presented in Section 5.3: it contains two parts: (1) analysis of Tomcat system, in Section 5.3.1; (2) analysis of Axis system in Section 5.3.2; further empirical data includes histograms and descriptive statistics of source code metrics for both projects in Subections 5.3.1.1 and 5.3.2.1 respectively.

We have also presented histograms and descriptive statistics for FindBugs warnings' categories for the projects in Subsections 5.3.1.2 and 5.3.2.2 respectively. Ultimetaly, in the Subsections 5.3.1.3 and 5.3.2.3, we have studied the relationship between the source code metrics and warnings' categories for Tomcat and Axis.

Application of results: Current tools to check coding standards report a high number of false positives warnings leaving it to developers to select the warnings to be addressed first. This thesis aims to resolve the issues of rule selection and violation ranking. We investigate the correlation between source code metrics and warning categories as used by the Java FindBugs tool. This lays the basics for developing a more advanced model for ranking warnings to aid developers in identifying and fixing the most severe warnings first.

6.2 Future Work

Future work (that is actually already ongoing) includes using additional case studies, computing different source code metrics, other kind of statistics analysis' studies, and experiment some ideas to improve our approach:

- We have used two open-source Java applications, Tomcat [cita] and Axis [citz]. To be able to further generalize the results, our approach should be applied to more systems. So, we are preparing a larger case study which may include AspectJ [citc], Columba [cite], Lucene [cito], and Scarab [city] systems.
- We have computed 14 source code metrics using the State Of Flow Eclipse Metrics (plugin) [citg]. However, it would be interesting to collect and study additional source code metrics such as Number of children, Depth of inheritance tree, Number of interfaces, and Afferent Coupling (from metrics plugin [citq]).
- We have also studied the relationship between source code metrics and warnings' categories using Spearman correlation analysis. We may apply more advanced statistics analysis techniques, such as decision tree analysis.
- We want to experiment a number of ideas to further improve our approach by studying particular warnings' types inside warnings' categories. We may also apply our approach more deeply using methods instead of classes.
- We want to explore a different approach using Test Coverage: Warnings inside source code which have high test coverage should have lower priority than warnings within source code having small test coverage. Ultimately, we may also combine and apply both approaches: source code metrics and test coverage.

References

[Bas05] W. Basalaj. Correlation between coding standards compliance and software quality. *IEE Seminar Digests*, 2005(11311):46–46, 2005.

[BE96] William Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, London, UK, 1996. Springer-Verlag.

[Bin94] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 374–388, London, UK, 1994. Springer-Verlag.

[BM06] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 149–160, Washington, DC, USA, 2006. IEEE Computer Society.

[cita] Apache tomcat. "http://tomcat.apache.org".

[citb] Apache tomcat - apache tomcat 6 downloads. "http://tomcat.apache.org/download-60.cgi".

[citc] The aspectj project. "http://www.eclipse.org/aspectj/".

[citd] Axis download. "http://apache.mirrors.webazilla.nl/ws/axis/1_4/".

[cite] Columba. "http://sourceforge.net/projects/columba/".

[citf] Eclipse downloads. "http://www.eclipse.org/downloads/".

[citg] Eclipse metrics plugin - state of flow. "http://eclipse-metrics.sourceforge.net".

[cith] Eclipse.org. "http://www.eclipse.org/".

[citi] Efferent couplings. "http://eclipse-metrics.sourceforge.net/descriptions/EfferentCouplings.html".

[citj] Feature envy. "http://www8.cs.umu.se/~lucasl/llamametrics/descriptions/FeatureEnvy.html".

[citk] Ibm spss statistics. "http://www.spss.com/statistics/".

[citl] Java se downloads. "http://java.sun.com/javase/downloads/index.jsp".

[citm] Lack of cohesion in methods. "http://eclipse-metrics.sourceforge.net/descriptions/LackOfCohesionInMethods.html".

[citn] Lines of code. "http://eclipse-metrics.sourceforge.net/descriptions/LinesOfCode.html".

[cito] Lucene. "http://lucene.apache.org/".

[citp] Mccabe's cyclomatic complexity. "http://eclipse-metrics.sourceforge.net/descriptions/CyclomaticComplexity.html".

[citq] Metrics 1.3.6. "http://metrics.sourceforge.net/".

[citr] Nominal, ordinal and scale. "http://www.spsslog.com/2006/05/03/nominal-ordinal-and-scale/".

[cits] Number of fields. "http://eclipse-metrics.sourceforge.net/descriptions/NumberOfFields.html".

[citt] Number of levels. "http://eclipse-metrics.sourceforge.net/descriptions/NumberOfLevels.html".

[citu] Number of locals in scope. "http://eclipse-metrics.sourceforge.net/descriptions/NumberOfLocals.html".

[citv] Number of parameters. "http://eclipse-metrics.sourceforge.net/descriptions/NumberOfParameters.html".

[citw] Number of statements. "http://eclipse-metrics.sourceforge.net/descriptions/NumberOfStatements.html".

[citx] Revision control - wikipedia. "http://en.wikipedia.org/wiki/Revision_control".

[city] scarab.tigris.org. "http://scarab.tigris.org/".

[citz] Webservices - axis. "http://ws.apache.org/axis".

[cit04] Guidelines for the use of the c language in critical systems. "http://www.misra-c.com/", 2004.

[cit10a] Bug report - docforge programming wiki. "http://docforge.com/wiki/Bug_report", April 2010.

[cit10b] Bug tracking system - wikipedia. "http://en.wikipedia.org/wiki/Bug_tracking_system", April 2010.

[cit10c] Bugzilla defect tracking ssytem. "http://www.bugzilla.org/", April 2010.

[cit10d] Code conventions - wikipedia. "http://en.wikipedia.org/wiki/Code_conventions/", April 2010.

[cit10e] Cvs - open source version control. "http://www.nongnu.org/cvs//", April 2010.

[cit10f] Findbugs - findbugs in java programs. "http://findbugs.sourceforge.net/", April 2010.

[cit10g] Pmd. "http://pmd.sourceforge.net/", April 2010.

[cit10h] Software configuration management - wikipedia. "http://en.wikipedia.org/wiki/Software_configuration_management", April 2010.

[cit10i] Subversion. "http://subversion.tigris.org/", April 2010.

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, 1976.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[Hat95] Les Hatton. *Safer C: Developing Software for in High-Integrity and Safety-Critical Systems*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[Hat03] Les Hatton. Safer language subsets: an overview and a case history. *MISRA C Accepted by Information and Software Technology*, 46:465–472, 2003.

[Hat07] Les Hatton. Language subsetting in an industrial context: A comparison of misra c 1998 and misra c 2004. *Inf. Softw. Technol.*, 49(5):475–482, 2007.

[HW08] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50, New York, NY, USA, 2008. ACM.

[KAYE04] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, New York, NY, USA, 2004. ACM.

[KE03] Ted Kremenek and Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *SAS'03: Proceedings of the 10th international conference on Static analysis*, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag.

[KE07] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[KPW06] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006. ACM.

[KR00] Jens Knoop and Oliver Rüthing. Constant propagation on the value graph: Simple constants and beyond. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 94–109, London, UK, 2000. Springer-Verlag.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[LTW$^+$06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM.

[Pat95] Jason R.C. Patterson. Accurate static branch prediction by value range propagation. Technical report, Australia, 1995.

[PSMV98] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.*, 7(1):41–79, 1998.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[Rus91] Glen W. Russell. Experience with inspection in ultralarge-scale development. *IEEE Softw.*, 8(1):25–31, 1991.

[SHP06] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM.

[SRH95] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 651–665, London, UK, 1995. Springer-Verlag.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[VCH96] Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized constant propagation: A study in c. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 74–90, London, UK, 1996. Springer-Verlag.

[WAP+02] Claes Wohlin, Aybuke Aurum, Håkan Petersson, Forrest Shull, and Marcus Ciolkowski. Software inspection benchmarking - a qualitative and quantitative comparative opportunity. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 118, Washington, DC, USA, 2002. IEEE Computer Society.

[WBM96] David A. Wheeler, Bill Brykczynski, and Reginald N. Meeson, Jr., editors. *Software Inspection: An Industry Best Practice for Defect Detection and Removal*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[WDA+08] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[web] Weighted methods per class. "http://eclipse-metrics.sourceforge.net/descriptions/WeightedMethodsPerClass.html".

[WH05] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[WJK+05] Stefan Wagner, Jan Jürjens, Claudia Koller, Peter Trischberger, and Technische Universität München. Comparing bug finding tools with reviews and tests. In *In Proc. 17th International Conference on Testing of Communicating Systems (TestCom'05), volume 3502 of LNCS*, pages 40–55. Springer, 2005.

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM.

[WMGH94] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1994. ACM.

[WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM.

Appendix A

Tools instalation

A.1 FindBugs

To install the FindBugs plugin:

1. In Eclipse, click on **Help -> Software Update** -> Find and Install...
2. Choose the **Search for new features to install** option, and click **Next**.
3. Click **New Remote Site**.
4. Enter the following:
   - **Name**: FindBugs update site
   - **URL**: one of the following (note: no final slash on the url)
     - **http://findbugs.cs.umd.edu/eclipse** for official releases
     - **http://findbugs.cs.umd.edu/eclipse-candidate** for candidate releases and official releases
     - **http://findbugs.cs.umd.edu/eclipse-daily** for all releases, including developmental ones

   and click **OK**.
5. "FindBugs update site" should appear under **Sites to include in search**.
   Click the checkbox next to it to select it, and click **Finish**.
6. We should see **FindBugs Feature** under **Select features to install**.
   (We may have to click on one or two triangles to make it visible in the tree.)
   Select the checkbox next to it and click next.
7. Select the **I accept** option to accept the license and click **Next**.
8. We have to make sure the location is correct where we are installing it. The default (workspace) should be fine. Click **Finish**.
9. The plugin is not digitally signed. Go ahead and install it anyway.
10. Click **Yes** to make Eclipse restart itself.

A.2 State of Flow Eclipse Metrics

To install the Eclipse Metrics Plugin:

1. In Eclipse, click on **Help -> Software Update** -> Find and Install...
2. Choose the **Search for new features to install** option, and click **Next**.
3. Click **New Remote Site**.
4. Enter the following:
   - **Name**: State of Flow update site
   - **URL**: one of the following (note: no final slash on the url)
     - **http://www.stateofflow.com/UpdateSite**

   and click **OK**.
5. "State of Flow update site" should appear under **Sites to include in search**.
   Click the checkbox next to it to select it, and click **Finish**.
6. We should see **State of Flow Feature** under **Select features to install**.
   (We may have to click on one or two triangles to make it visible in the tree.)
   Select the checkbox next to it and click next.
7. Select the **I accept** option to accept the license and click **Next**.
8. We have to make sure the location is correct where we are installing it. The default (workspace) should be fine. Click **Finish**.
9. The plugin is not digitally signed. Go ahead and install it anyway.
10. Click **Yes** to make Eclipse restart itself.

Appendix B

Merging Algorithm

Due to the fact that python indentation is line oriented, one statement must be completely in the same line. Therefore, for display purposes only, we have broken the lines 15 and 16; 103 and 104; 181 and 182; 184 and 185; 219 and 220; 258 and 259; 305 and 306; 321 and 322, which should be merged before trying to run the script.

B.1 script.py

```
1  #!/usr/bin/python -u
2
3  import sys
4  import os, os.path
5
6  ######################################## script.py ######################################
7  #
8  # This script mergess the metrics data and warning data contained in different sources
   #
9  #
10 ########################################################################################
11
12 #———————————————————————————— Class declaration ————————————————————————————#
13
14 class ClassMetrics:
15   def __init__(self, package, name, method, line, m1, m2, m3, m4, m5, m6, m7,
16   m8, m9, m10, m11, m12, m13, m14):
17     self.package  = package # Package
18     self.name     = name  # Type (Class Name)
19     self.methods  = [Method(method, line, m1, m2, m3, m4, m5, m6, m7)]  # Methods objects
20     self.warnings = []  # List of Warning objects type
21     self.warningTypes = []  # List of Warning Types objects type
22     self.warningCategories = [] # List of Warning categories Types objects type
23     self.m8 = m8  # Metric from types-metrics.csv file
24     self.m9 = m9
25     self.m10 = m10
26     self.m11 = m11
27     self.m12 = m12
28     self.m13 = m13
29     self.m14=  m14
30
31   def attributes(self):
32     if self.package  == '': # if has NO package
33       stringPackageAndType = self.name  # String with only Class name
```

```
    else: # if has package
      # String with Package and Class
      stringPackageAndType = self.package + '.' + self.name

    stringMethodsMetrics = '' # String with Methods object type (method name and metrics)
    stringWarnings = '' # String with Warnings object type (type,..., category)

    max_m1  = 0 # Max Methods metric 1
    max_m2  = 0
    max_m3  = 0
    max_m4  = 0
    max_m5  = 0
    max_m6  = 0
    max_m7  = 0 # Max Methods metric 7

    for i, m in enumerate(self.methods):
      # stores on a string all attributes of each method
      stringMethodsMetrics = stringMethodsMetrics + m.attributes()
##
# Gets the maximum values of each methods metric

      if m.m1 > max_m1:
        max_m1 = m.m1

      if m.m2 > max_m2:
        max_m2 = m.m2

      if m.m3 > max_m3:
        max_m3 = m.m3

      if m.m4 > max_m4:
        max_m4 = m.m4

      if m.m5 > max_m5:
        max_m5 = m.m5

      if m.m6 > max_m6:
        max_m6 = m.m6

      if m.m7 > max_m7:
        max_m7 = m.m7
##
#
    stringClassMetrics__  = ',' + str(max_m1) + ',' + str(max_m2) + ',' + str(max_m3) + ','
    stringClassMetrics_ = stringClassMetrics__ + str(max_m4) + ',' + str(max_m5) + ','
    # String with Type Metrics (Maximum value of each metric)
    stringClassMetrics   = stringClassMetrics_ + str(max_m6) + ',' + str(max_m7)

    # For each  warning type it calculates the number of occurences
    stringWarningTypesCount = ''

    # For each  warning category it calculates the number of occurences
    stringWarningCategoriesCount = ''

    stringTypeMetrics_ = ',' + self.m8 + ',' + self.m9 + ',' + self.m10 + ',' + self.m11 + ','
    stringTypeMetrics  = stringTypeMetrics_ + self.m12 + ',' + self.m13 + ',' + self.m14

    for i, nw in enumerate(self.warningTypes):
      # stores warning types of the class
      stringWarningTypesCount = stringWarningTypesCount + ',' + str(nw)

    for i, nw in enumerate(self.warningCategories):
      # stores warning categories of the class
      stringWarningCategoriesCount = stringWarningCategoriesCount + ',' + str(nw)

    for i, w in enumerate(self.warnings):
      # stores the attributes of each warning
      stringWarnings = stringWarnings + w.attributes()

    return stringPackageAndType + stringClassMetrics.strip('\n') +
    stringTypeMetrics.strip('\n')+stringWarningTypesCount+stringWarningCategoriesCount+'\n'

class Method:
  def __init__(self, name, line, m1, m2, m3, m4, m5, m6, m7):
    self.name = name    # Method Name
    self.line = line    # Line
    self.m1  = int(m1)
    self.m2  = int(m2)
    self.m3  = int(m3)
    self.m4  = int(m4)
    self.m5  = int(m5)
    self.m6  = int(m6)
    self.m7  = int(m7)

  def attributes(self):
    string =  self.name + ',' + self.line + ',' + str(self.m1) + ','
    string1 = string + str(self.m2) + ',' + str(self.m3) + ','
    string2 =  string1 + str(self.m4) + ',' + str(self.m5) + ','
    # Methods nome and respective metrics
    return string2 + str(self.m6) + ',' + str(self.m7)

class Warning:
  def __init__(self, type, priority, abrev, category):
    self.type     = type      # Type of Warning
    self.priority = priority  # Priority
    self.abrev    = abrev     # Abreviature
    self.category = category  # Category

  def attributes(self):
    return self.type + ',' + self.priority + ',' + self.abrev + ',' + self.category + '\n'

#---------------------------------- Function definition -----------------------------------#
# Locates the object with name cname in the list of object cmList
def pinpoint(cpkg, cname, cmList):
  for i, cm in enumerate(cmList):
    # cname Class Name ; cmList Class Metrics List
    if cname == cm.name and cm.package==cpkg:
      return i
  return -1

# Locates the string name in the list (used for warning types and categories)
def pinpointWt(name, list):
  for i, wt in enumerate(list):
    if name == wt:
      return i
  return -1

#---------------------------------- Variable declaration ----------------------------------#

cMetricsLines= open('methods-metrics.csv', 'r').readlines()
```

```
warningLines = open('warnings.xml', 'r').readlines()
typesMetricsLines = open('types-metrics.csv')

ClassMetricsList   = []
WarningTypesList   = []
WarningCategoriesList   = []
warningTypesHeader = ''
warningCategoriesHeader = ''
#------------------------------- Script body -------------------------------#

for i, line in enumerate(cMetricsLines):
  if line.find('PACKAGE') == -1:  # Skips the first line
    cp = line.split(',')
    #if cp[2] == '': cp[2] = '0'  # method
    if cp[3] == '': cp[3] = '0'   # line
    if cp[4] == '': cp[4] = '0'   # if metric is empty fills with 0
    if cp[5] == '': cp[5] = '0'
    if cp[6] == '': cp[6] = '0'
    if cp[7] == '': cp[7] = '0'
    if cp[8] == '': cp[8] = '0'
    if cp[9] == '': cp[9] = '0'
    if cp[10].strip('\n') == '': cp[10] = '0'

    # checks if the class stated in current line was already stored in ClassMetricsList
    index = pinpoint(cp[0], cp[1], ClassMetricsList)
    # if the class was already stored creates a new method instance
    if index != -1:
      ClassMetricsList[index].methods.append(Method(cp[2], cp[3], cp[4], cp[5],cp[6],
      cp[7], cp[8], cp[9], cp[10]))
    else:
      cm = ClassMetrics(cp[0], cp[1],cp[2], cp[3], cp[4], cp[5], cp[6], cp[7],
      cp[8], cp[9], cp[10], '0', '0', '0', '0', '0', '0', '0')
      # if not it stores the cclass stated on the current line in the list
      ClassMetricsList.append(cm)

  else: # first line of methods.csv file
    methods_header = line.replace('METHOD,LINE,', '')
    methods_header = methods_header.replace('PACKAGE,', '').strip('\n')

for i, line in enumerate(typesMetricsLines):
  if line.find('PACKAGE') == -1:  # Skips the first line
    cp = line.split(',')
    if cp[3] == '': cp[3] = '0' # if metric is empty fills with 0
    if cp[4] == '': cp[4] = '0'
    if cp[5] == '': cp[5] = '0'
    if cp[6] == '': cp[6] = '0'
    if cp[7] == '': cp[7] = '0'
    if cp[8] == '': cp[8] = '0'
    if cp[9].strip('\n') == '': cp[9] = '0'

    if cp[1].find('$'):
      cp[1] = cp[1].replace('$', '.')

    # checks if the class stated in current line was already stored in ClassMetricsList
    index = pinpoint(cp[0], cp[1], ClassMetricsList)
    if index != -1:  # if the class was already stored creates a new type instance
      if int(cp[3]) > int(ClassMetricsList[index].m8):  ClassMetricsList[index].m8
 = cp[3]
      if int(cp[4]) > int(ClassMetricsList[index].m9):  ClassMetricsList[index].m9
 = cp[4]
```

```
      if int(cp[5]) > int(ClassMetricsList[index].m10):  ClassMetricsList[index].m10 = cp[5]
      if int(cp[6]) > int(ClassMetricsList[index].m11):  ClassMetricsList[index].m11 = cp[6]
      if int(cp[7]) > int(ClassMetricsList[index].m12):  ClassMetricsList[index].m12 = cp[7]
      if int(cp[8]) > int(ClassMetricsList[index].m13):  ClassMetricsList[index].m13 = cp[8]
      if int(cp[9]) > int(ClassMetricsList[index].m14):  ClassMetricsList[index].m14 = cp[9]

    else:
      cm = ClassMetrics(cp[0], cp[1], '', '0', '0', '0', '0', '0', '0', '0', '0',
      cp[3], cp[4], cp[5], cp[6], cp[7], cp[8], cp[9])
      if cm.name.find('$'):
        cm.name = cm.name.replace('$', '.')
      # if not it stores the cclass stated on the current line in the list
      ClassMetricsList.append(cm)
      index= pinpoint(cm.package, cm.name, ClassMetricsList)

  else: # first line of types-metrics.csv file
    types_header = line.replace('PACKAGE,TYPE,LINE', '').strip('\n')

print len(ClassMetricsList)

for i, line in enumerate(warningLines):
  if line.find('<BugInstance') != -1:  # if the line has the string <BugInstance
    cp = line.split('␣')
    # On the next line removes the residual characters
    packageCp = warningLines[i+1].replace('<Class␣classname="', '').replace('\">\n', '')
    packageCp = packageCp.strip('␣')  #### This fixes the bug
    packageCp = packageCp.split('.')  #### This fixes the bug
    cname = packageCp[len(packageCp)-1] # gets the class name
    cpackage = ''           # package name

    for i, p in enumerate(packageCp):  # build package name
      if i == len(packageCp) -1: break
      if i == 0:  # if has package name
        cpackage = p
      else:
        cpackage = cpackage + '.' + p

    if cname.find('$')!= -1:          # Methods with $ in warnings.xml file
      scname= cname.split('$')
      cname = scname[0] + '.' + scname[1]

    # We can have more than one warning by class
    index = pinpoint(cpackage, cname, ClassMetricsList)

    if index != -1 and ClassMetricsList[index].package == cpackage:
      # Parses the file and creates a new warning object with exctracted information
      w = Warning(cp[3].split('=')[1].strip('\"'), cp[4].split('=')[1].strip('\"'),
      cp[5].split('=')[1].strip('\"'), cp[6].split('=')[1].strip('\"'))

      if len(ClassMetricsList[index].warningTypes) == 0:
        acc = 0
        while acc < len(WarningTypesList):
          ClassMetricsList[index].warningTypes.append(0)
          acc = acc +1

```

```python
      if len(ClassMetricsList[index].warningCategories) == 0:
        acc = 0
        while acc < len(WarningCategoriesList):
          ClassMetricsList[index].warningCategories.append(0)
          acc = acc +1

      if WarningTypesList.count(cp[3].split('=')[1].strip('\"')) == 0:
        WarningTypesList.append(cp[3].split('=')[1].strip('\"'))
        ind = pinpointWt(cp[3].split('=')[1].strip('\"'), WarningTypesList)
        acc = len(ClassMetricsList[index].warningTypes)
        while acc < ind:
          ClassMetricsList[index].warningTypes.append(0)
          acc = acc +1

        ClassMetricsList[index].warningTypes.append(1)

      else:
        for j, wt in enumerate(WarningTypesList):
          if cp[3].split('=')[1].strip('\"') == wt:
            break

        ClassMetricsList[index].warningTypes[j] = ClassMetricsList[index].warningTypes[j] +1

      if WarningCategoriesList.count(cp[6].split('=')[1].strip('\"')) == 0:
        WarningCategoriesList.append(cp[6].split('=')[1].strip('\"'))
        ind = pinpointWt(cp[6].split('=')[1].strip('\"'), WarningCategoriesList)
        acc = len(ClassMetricsList[index].warningCategories)
        while acc < ind:
          ClassMetricsList[index].warningCategories.append(0)
          acc = acc +1

        ClassMetricsList[index].warningCategories.append(1)

      else:
        for j, wt in enumerate(WarningCategoriesList):
          if cp[6].split('=')[1].strip('\"') == wt:
            break

        ClassMetricsList[index].warningCategories[j] =
        ClassMetricsList[index].warningCategories[j] +1

      ClassMetricsList[index].warnings.append(w)

for i, cm in enumerate(ClassMetricsList):
  cm.attributes()

#ClassMetricsList.sort(key=lambda cm:cm.max_ml, reverse=True) # type order by ml

for i, wt in enumerate(WarningTypesList):
  warningTypesHeader = warningTypesHeader + ',' + wt

for i, wt in enumerate(WarningCategoriesList):
  warningCategoriesHeader = warningCategoriesHeader + ',' + wt

open('metrics+warnings.csv', 'a').write('%s%s%s%s\n'
% (methods_header, types_header, warningTypesHeader, warningCategoriesHeader))

for i, cm in enumerate(ClassMetricsList):
  #cm.methods.sort(key=lambda m:m.ml, reverse=True) # methods order by ml
  while len(cm.warningTypes) < len(WarningTypesList):
```

```
            cm.warningTypes.append(0)

    while len(cm.warningCategories) < len(WarningCategoriesList):
        cm.warningCategories.append(0)

    open('metrics+warnings.csv', 'a').write(cm.attributes())
```