

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER ONE

INTRODUCTION

1.1 Overview

Through the software development life cycle a series of changes need to be accomplished. These changes are required because of many reasons such as; enhancement, adaption, and maintenance or fixing the program defects (Bieman, *et al*, 2003). From these changes and results we can say the software is infinitely flexible (Koru.2005). However, changes must be considered as major risk elements, since they may impact time and cost (Koru & Liu, 2007). In addition, change-proneness of the software may lead to specific important quality issues (Bieman, *et al*, 2003).

The change history of software code provides useful information about the evolution of programs. This information helps us to understand the overall picture of the system evolution starting from design phase ending with maintainability phase (Al-khiaty.2009).

Software quality is a serious issue to consider, since software is entering in all life details starting from simple industries like children toys ending to industries like airplane.

1.2 Dealing with Quality Problems

To deal with the quality problems we need to study how can we test and measure the source code itself. The results from these studies and measurements provide useful information that can help in solving such quality problems.

1.2.1 Dynamic Testing

Dynamic testing or analysis focuses in accomplishing customer requests by supporting all requirements and functionalities by the software as a final product (Lochmann & Goeb, 2011).

Software testing tools are programs that try to find errors, defects, bugs, failures, etc. in the evaluated software products. Those different terms are, sometime, different based on the level and the nature of the errors. The errors are unexpected behavior of the system. The defects refer to the many problems related to software products, either external behavior or internal features, but a fault in a program which causes the program to perform in an unintended or unanticipated manner. The failure that means the system does not deliver a service as expected by it is user. The output of each test case in a testing process is one of two: pass or fail. The designer of the test cases defines the inputs for each test case along with expected outputs. On the execution, test cases are executed and actual results are compared with expected results. For those failed test cases (i.e. expected result is different from the actual result), a debugging process further starts to see why those test cases produce incorrect outputs or results. Errors can be syntax, semantic, functional, and non-functional. Errors may stop the compilation process or may not and only cause different or unexpected behavior from those defined by users.

1.2.2 Metrics

Studying class characteristics and identifying their attributes in terms of changes is very useful in the maintenance process. Consequently, this will make project manager and team to give more attention to the possibility of changes in classes during the

project life cycle (Bieman, *et al*, 2003). Here where the importance of measuring software metrics takes place.

1.2.3 Source Code Analysis Tools (static testing)

Many quality aspects can be identified by using metrics. Thus, software metrics are tools to measure one or more code attributes (EKLÖF.2011).

Source code analysis (SCA) tools are used to check the source code for attributes such: number of lines of code or any other static metrics of the code. Examples of such static metrics include: Lines Of Code (LOC), size, and complexity. It can be applied after the code is written which means that it may help us to learn about the code and possibly catch defects before testing phase. Although SCA cannot find all kinds of defects, it can be considered as an efficient tool in terms of cost and time (EKLÖF.2011). SCA tools are usually applied automatically with the least amount of effort and time from the users or testers side.

1.3 Sample of Source Code Analysis Tools

In this section, we will list some tool examples that are applied on the source code specially those that we used in our experimental studies.

1.3.1 StyleCop

StyleCop is an open source static SCA tool from Microsoft that checks .NET code for conformance of several design guidelines defined based on Microsoft's .NET Framework (CodePlex.2011). StyleCop analyzes the code in order to apply a set of rules which can be classified into several categories such as (CodePlex.2011): Naming, maintainability, documentation, ordering, readability, spacing, and layout. Table 1.1 shows a sample of some warnings and their classification.

Table 1.1: A sample SCA warning classification

Warnings	Categories
The spacing around an operator symbol is incorrect.	Spacing
The call to channel should only use the 'base.' prefix if the item is declared virtual in the base class and an override is defined in the local class. Otherwise, prefix the call with this rather than base.	Readability
All using directives must be placed inside of the namespace	Ordering
Method names begin with an upper-case letter.	Naming Rules
The class must have an access modifier	Maintainability
A statement containing curly brackets must not be placed on a single line. The opening and closing curly brackets must each be placed on their own line.	Layout
The constructor must have a documentation header.	Documentation

1.3.2 JustCode

JustCode is another example of SCA tools. There are some JustCode features that include (Telerik.2011): On-the-fly Code analysis, code navigation and search, refactoring, quick fixes, coding assistant and hints. JustCode executes its code analysis by applying custom inspections. There are several inspections that can be performed by JustCode. Examples include (Telerik.2011): Identical if and else clauses, obsolete casts, empty statements, assignments with no effect, unused private members, unused parameters, variables, namespaces, or statements. Figure 1.1 shows a sample of SCA output from JustCode.

```
public int Foo()
{
    return "bar";
}
// C#: An instance of type "string" cannot be returned by a method of type "int"
```

Errors – by default Just Code underlines errors with a red line

1.3.3 FxCop

FxCop is another example of SCA tools. FxCop is an application that resolves assembly codes after the source codes are compiled, and notifies information about the code assemblies, such as security improvements, possible design, performance and localization (MSDN, 2013).

FxCop is intentional for class library developers. But, anyone making applications that should conform to the .NET Framework best exercises will benefit. Also, FxCop is useful as a pedagogical tool for people who are uncommon with the .NET Framework Design Guidelines or who are fresh to the .NET Framework (MSDN, 2013).

FxCop is developed to be fully merged into the Systems Development Life Cycle (SDLC) and is distributed as both a command-line tool (FxCopCmd.exe) appropriate for integrated with Microsoft Visual Studio or usage as part of automated build processes .NET as an exterior tool. And a fully distinguished application that has a Graphical User Interface (GUI) (FxCop.exe) for interactive work (MSDN, 2013).

1.4 Problem Statement

Static source code analysis tools are software programs that are used to evaluate programs statistically and evaluate certain characteristics based on predefined quality standards. Unlike software testing where expected output will be (pass or fail) based on the conformance of expected outcome with the actual outcome. In SCA, the output will be one of three classes: error, warning or information.

Criteria are defined for what standard or typical program should be or should have. Based on those standards, a subject code is evaluated depending on the level of

conformance or violation of a standard, one of the three classes (i.e. error, warning, or information) is defined to show some quality aspects of the evaluated software.

First, we have evaluated several selected free and commercial SCA tools for the purpose of comparing, correlating and assessing the results. Our focus is on the warning class of issues as it is considered as a vague class between errors and information where many developers underestimate or ignore warning signs.

Second, we have evaluated the relations and the correlation between SCA reported warnings. Extensive statistical analyses from all evaluated SCA tools are conducted to evaluate the ability of warning reports by SCA tools to predict bugs or defects.

Based on those relations from the different SCA tools, we have first listed the important characteristics from all warning classes that were significant to bugs or defects.

Moreover, we have proposed enhancements on SCA and developed a tool to consider the major warning classes that showed high defect predictability values. The last goal that we have performed is to evaluate the correlations between data from software metrics tools and SCA tools.

1.5 Research Objectives

Based on the problem statement, we defined three major objectives that are accomplished in this thesis:

- Extensively evaluate several selected free and commercial SCA tools for the purpose of comparing, correlating and assessing the reported information. Expected outcome has included statistical data from several

open source evaluated projects that show all classes of warnings collected from the selected SCA tools. Moreover, the similarities and differences between the SCA tools will be shown.

- Evaluate the inconsistency of results and the kind of warnings that may vary from one experiment to another given the same tool and tested source code. Expected output have data and reports with inconsistency between reported warnings in the tools when apply these tools more than one run or test.
- Proposed enhancements on SCA and developed a tool to consider the major warning classes that showed high defect predictability values. Expected output is a tool or, for the least, a framework for the relevant and important SCA warning information combined from all evaluated SCA tools and possibly adding new warning classes discovered through this thesis and evaluate the correlations between data from software metrics tools and SCA tools.

1.6 Research Importance

Software quality tools are used to assess quality of software through all development stages. However, there is a little public information about test evaluation of the accuracy and value of the warning that are reported from some of these tools (Ayewah, *et al*, 2007).

By using static SCA tools we can study the architecture of the source code packages (EKLÖF.2011). Therefore, we have tested several codes downloaded from SourceForge.NET to evaluate the value of different warning messages in that code

project and see if such warning messages can correlate with bug or defect data collected from the source codes.

1.7 Thesis Structure

The following chapters of this thesis are organized as the following: Chapter two presents related studies to software quality. Chapter three presents static code analysis tools. Chapter four shows the research goals and approaches. Chapter five presents experimental results and analysis. Chapter six describes how to use the proposed tool. Chapter seven presents the conclusions and future work.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER TWO

RELATED WORKS

This chapter is a literature survey of the previous work that search in the history of software metrics, software analyzing, and software maintainability in order to enhance the quality and maintainability even after the product released.

It is divided into four sections starting with first section that describe software metrics their importance as attributes of software, and their role in facilitating software maintainability. Second section describes software quality. The Third section considers testing and SCA tools. Finally fourth section is dealing with software maintainability and changes as the final step in the software development life cycle.

2.1 Software Metrics and Class Change Proneness

Studying software metrics class characteristics and identifying their attributes in term of changes is very useful in the maintenance process. Consequently, this will make encourage project manager and his team to give more attention to the possibility of changes in classes during the project life cycle (Bieman, *et al*, 2003). Here where the importance of measuring software metrics take place.

According to Girba et al. (2004), their approach depends on the changes in the evolution of the Object Oriented (OO) software system by providing historical measurement study. The study focuses on the change in the history of a class by observing the change in the nature of methods in different versions, that means they measure the change by using one main code attribute (number of methods) add or remove method to certain class. Form the number of methods metrics can be derived

another two different metrics, the Latest Evolution Of Number Of Methods (LENOM) and the Earliest Evolution Of Number Of Methods (EENOM). By these two metrics the change in size inside each class over the software history different versions can be known and changes here focus only on the number of methods that added or removed from each class over different releases.

Koru and Liu (2007) focus on change-prone classes by providing tree-based model that shows the class characteristics, they test Pareto's law for the open source code programs which state that 80% of code changes are centered at 20% of the classes. They mainly searched in how to identify change-prone classes and their characteristics by trying to observe the change of set of static metrics of a group of products with different releases of an open source project, they prove the validity and applicability of Pareto's law for open source programs, they also provide useful guidance in development and maintenance of large-scale open source programs.

According to Basten and Klint (2009), finding and discovering the facts from a source code is an important step while software analysis is done. Several experiments are done and found that extracting facts from any source code then writing them in a large wide of programming languages; it will lead to hard working and error prone. Because of these reasons they developed a new technique which called DeFacto. It is language-parametric analysis software for fact extraction from the software source code.

According to Bieman, et al. (2003), four research questions were treated. The first research question was about visualization and identification of change-prone sets of classes in an object-oriented framework. The second research question was to do with differentiating change-prone clusters from local change-proneness of classes. Also this method was displaying how to determine the degree to which classes are change-prone

both in their interplays with others and locally. This method was applied to a considerable case study. For this case study, in response to the third research question that which modifies interplays between classes do not necessarily imitate functional interplays in the resolve of the framework. This which can have a diversity of causes. An example would be refinements of specific factors such as performance. Performance refinements may trigger concurrent alterations in classes that otherwise do not react with each other. On the other hand, in response to fourth research question, cluster change-proneness versus local was visualized through the alter-architecture graph and paralleled it to the design graph. We also differentiated between alter-prone clusters of classes which did not include in patterns and those which are included. The visualization was straightforward and simple and driven by the alteration measures that were identified. Future work in this field involves the representation of other measurements such as size of box symbolizing size of class, utilizing of color, and covers of alter-architecture versus rational architecture.

According to Romano and Pinzger (2011), interfaces declare contracts that are denoted to stay stable during the development of a software framework while the concrete classes implementation (a subclass class can be instantiated that implements all the missing functionality) is more likely to alter. This guide to another evolutionary demeanor of interfaces paralleled to concrete classes. This behavior was experimentally examined with the C&K metrics that are broadly utilized to estimate the implementation quality of interfaces and classes. The outcomes of the study with two Hibernate projects and eight Eclipse plug-in and indicate that, the Interface Usage Cohesion (IUC) metric shows a more powerful connection with the number of fine-grained Source Code Changes (SCC) than the C&K metrics when stratified to interfaces, also The IUC metric

can ameliorate the performance of foretelling models in categorizing Java interfaces into two categories, change-prone and not change-prone.

According to Romano et al. (2012), Anti-patterns have been defined to mean “poor” solutions to resolve and perform problems. Previous researches have indicated that classes impacted by anti-patterns are more change-prone than classes that did not impact by anti-patterns. A deeper premeditation was provided into which anti-patterns direct to which kinds of alterations in Java classes. The change-proneness of these classes was analyzed taking in consideration 40 kinds of (SCC) derived from the version control depository of 16 Java open-source frameworks. Classes impacted by anti-patterns alter more repeatedly along the development of a framework; Classes impacted by the SwissArmyKnife, ComplexClass, and SpaghettiCode anti-patterns are more probable to be altered than classes impacted by other anti-patterns in addition that, specific anti-patterns lead to specific kinds of source code alterations, like as Application Programming Interface (API) alterations are more probable to be shown in classes impacted by the SwissArmyKnife, ComplexClass, and SpaghettiCode anti-patterns.

Shatnawi and Li (2008) investigated three publications of the Eclipse project and detected that although several software metrics can still prognosticate class fault proneness in three errors - acuteness categories, the thoroughness of the prognosis minimized from publications to publications. Moreover, the Researchers detected that the prognosis cannot be utilized to construct a software metrics paradigm to recognize fault-prone classes with admissible accuracy. SHATNAWI's findings propose that as a software develops, the utilize of certain usually utilized metrics to recognize which classes are more prone to faults turns into increasingly complicated.

Zhou and Leung (2006) utilized machine learning concept and logic regression method to experimentally examine the advantage of metrics of Object-oriented analysis and design (OOAD), particularly, a subset of the Chidamber and Kemerer metrics suite- composed of six metrics numbered for each class - in prophesying error-proneness when taking error acuteness into consideration. The findings depend on a public domain National Aeronautics and Space Administration (NASA) data set, show that 1) statistically, many of these Object-oriented design metrics are affined to class error-proneness across fault acuteness, and 2) the prognosis capabilities of the examined metrics extremely based on the acuteness of faults. More specifically, these Object-oriented design metrics are capable to divine low acuteness errors in error-prone classes better than high acuteness errors in error-prone classes.

Gyimothy et al. (2005) calculate the metrics of object-oriented design given by Chidamber and Kemerer metrics suite to explain how error-proneness discovery of the source software code of the electronic mail and open sourceWeb suite called Mozilla Application Suite can be achieved. The researchers examined the values acquired against the faults number detected in its error database — referred to as Bugzilla — utilizing machine learning concept and logic regression method to prove and examine the utility of these object oriented metrics for error-proneness prediction. The researchers also paralleled the various versions metrics of Mozilla to realize and examine how the divined error-proneness of the software altered during its development period.

2.2 Testing and Source Code Analysis Tools

EKLÖF (2011) conclude that, developing complicated software productions necessarily introduces flaws. Most of these defects can be grasped during testing phases in the software development process, with the assistance of test cases or code reviews. Furthermore, it is concluded that, static code analysis must be utilized pending the implementation stage, during test analysis and during integration testing – a type of testing is used to test software interfaces and interactions that occur between the software components.

According to Ayewah et al. (2007), the research focused on evaluating the accuracy and value of warnings that the analysis tools usually report as a result. They examined the FindBug as a software analysis tool that find defects in Java programs. They discussed different kinds of warnings generated and their classifications in to false positive, trivial bugs, and serious bugs. They also tried to answer many questions such as why the static analysis tools defect true but not important bugs. They report their experiments from integrating static analysis in to the software development process at Google.

According to Zheng et al. (2006), defects and failure reports that are the result of static analysis tools applied over three selected industrial programs were proposed, they found that: Static analysis tools are good choice for detecting software faults and defects in term of time. Static analysis tools are perfect for improving current versions to new releases of software by focus on complicated, operational, and algorithmic defects. From their statistical result analysis they found the number of defects can indicate for the nature of the problem and these statistical tools can work together with other fault-defect software for producing high quality software.

Lucia et al. (2010) using Eclipse plug-in as a static analysis tool to extract the design pattern from an object oriented source code, to perform design pattern recovery and behavioral analysis and monitoring. According to Sharif and Maletic (2010) recovering source code design pattern one of the most important steps for program maintenance since it gives important information that could help in understanding the semantic and logic together with system design which helps in system documentation and system redesigning.

According to Black et al. (2010), no amount of correction and analysis can give software product high levels of correctness, quality, security, or other serious properties. Successful choices of platform, programming language, are more important than reactive efforts. Notwithstanding code inspection or testing (dynamic analysis) has benefits. Testing has the feature of check the behavior of the code in execution. By contrast, only static analysis can be anticipated to detect malignant trapdoors. Executable or binary code analysis averts suppositions about source code semantics or compilation.

According to Mahmood et al. (2010), some programmers depend on software testing stage to find existing errors and bugs in the software. The inherent obstacle of testing that it endeavors at verification of software requirements rather than detecting bugs and errors in the software. The same thing happened with the quality assurance of the software which checks the software product under different status rather than finding new bugs in the software. So there is a need to use security at early phase of software development process. One of the most effective and popular method to fulfill this goal is manual code review, but this mode is considered costly and needs specialized knowledge in software implementation stage. One of the alternative and most applicable methods is to perform static code analysis utilizing certain tool at an

early phase of software development process. Static code analysis method can ameliorate performance as well as better usage of software resources with respect to effort and time. Furthermore, there are several commercial as well as open source tools used for this goal. Each one of these tools uses various technique and ways for static code analysis. One of the latent issues with static code analysis method is the ability to reduce the pseudo alarms, and to correctly distinguish the existing code-related vulnerabilities.

According to Abraham et al. (2012), during the coding stage, engineering groups either automatically or manually transform the design documents code, in other words, the code is written in this phase. The application of techniques for testing and verification in this stage is described code investigation and the objective is to generate robust code by proving the absence of bugs such as execution errors. This can be achieved with formal techniques combined with static code analysis — programmers can utilize static code analysis tools to test that the software is free of findable execution errors. On the other hand, the author found that testing phases –that are performed during software development process – may flop to find some bugs unless comprehensive and tiring testing is used. Furthermore, many errors stay in the software after the verification and testing processes were accomplished. These defects remain because comprehensive testing is usually not practical. Other methods must be utilized to remove remaining bugs, such as using of static code analysis tools to ameliorate quality of code.

2.3 Software Quality

In quality problems we need to study how we can test and measure the source code itself, the results from these studies and measurements provide useful information helps in solving such quality problems.

According to Bieman et al. (2001), the focus was on assessing the software design's quality and developing software, by specifying the relationships between design structure and other quality factors such as reusability, maintainability, testability, and adaptability. They studied the architectural design of the class in order to predict future class changes and analyze 39 commercial object oriented software systems by using set of static metrics. They found that there are three kinds of classes that are the most change-prone on the system over time, the large class, the class that inherited as a super class, and the class that participate in the design patterns.

As illustrated by Lochmann and Goeb (2011), a common foundation aimed to give information about disciplines and facilitates tracing a certain code, and global framework describes all concepts related to software quality were searched. They provide a general quality model in order to describe different attributes related to quality, relying to activities related to quality such as maintainability and usability, the model can be integrated with all standard concept, quality models, guidelines, and statics code checkers rules. They showed that the quality model could describe the interrelations of disciplines such as software requirements and test reaching to software quality.

As illustrated by Deissenboeck et al. (2007) the quality model criticisms analyzed them as a result of unclear definition of quality models and describe their purposes and usage scenarios. Critique of current models was used as general

requirements to evaluate, and improve the existing models or even develop new enhanced models from scratch. They introduced three clear definitions for quality model as a concept to reflect the importance of quality model's purposes as follows: Quality model: a model to describe, assess, and predict quality. Quality Meta model: a model with rules needed to build a specific quality models. Quality modeling framework: a framework define, evaluate and improve quality.

As illustrated by Deissenboeck et al. (2009) they propose 2-dimensions model of maintainability that to which studies the system from maintainability perspective. They separate the maintenance activities from the system properties to identify the quality criteria and allow justifying their independencies, which helps to view the quality model in a structures design used in industrial project environments. Their model construction based on an explicit quality Meta model, which made the system more systematic and preciseness. The applicability of the model is confirmed by applying the model over a case study, they created a model of the maintainability of MATLAB Simulink models to use it frequently in model-based development of embedded systems.

As illustrated by Khaddaj and Horgan (2005), the traditional quality models used hierarchical techniques with restricted domain of factors that define quality, so they introduced a new model for handling software quality confirmation that dealing with the problems of old approaches and come with new factors of quality as common measurement instrument that can determine and analyze quality factors in technological enhancement way. Their approach was more flexible, since it can be extended to satisfy user requirement and add more details derived from the customer need.

According to Kuhn et al. (2006), they presented a new technique called Semantic Clustering based on Latent Semantic Clustering and Indexing to gather the

lingual information in the source code that use a same vocabularies. After that, they interpreted them in order to detect and discover what is the notion of the source code and to support program understanding by retrieve the topics including the same vocabulary. Simply, this process is done by a number of steps, beginning by comparing all topics together then they tied them by links. According to the first two steps, tables are drawn automatically according to retrieval data. After that, visualization is applied over the system to describe how they are divided.

According to Drake (1996), a project requires actual measures from actual data. If software productivity factors were not really understood, then it will not be known how to ameliorate the development processes. None of process "standards" the present or suggested software quality compels the details of the "software development process", thus an actual software process must be constructed from within. But, even when achieved for the proper reasons, will turn on established rules and demeanor patterns and will turn the status in quo. More significantly, it will turn on the perceptions and brains of people. The significance of the people must be recognized in the process. Higher productivity and amended quality can be accomplished by tapping their concealed strengths. The suitable use of statistical and metrics techniques can help to supply support to the software development teams and measure progress , whilst at the same time improving quality , alleviating risk, and minimizing cost.

According to Jones (2012), In order to create righteous economic patterns of software maintenance, development, and quality control it is imperative to have rigorous measurements that use rigorous metrics. The industry cannot endure the errors and gaps of poor metrics like as technical debt, cost per defect, and lines of code. The integration of function point metrics integrated with Defect Removal Efficiency

metrics (DRE) can view the actual cost of quality and clarify the fact that obtaining high quality is the most cost-effective method to construct software.

Engelbertink (2010) was presented six methods to economize software maintenance costs. These are often relied on experimental studies. The Omnext CARE idea was described, a workable solution for establishing continual incremental amelioration and so decreasing software maintenance costs. Moreover, the unique state-of-the-art abilities of Omnext's Source2VALUE technology were described.

2.4 Maintenance

Maintainability one of six characteristics refers to quality, analysis, test, and check stability and changeability of quality models. It is one of the major software costs that concerned during the development life cycle of the software (Al-khiaty.2009).

As illustrated by Riaz et al. (1993), measuring and assessing the quality metrics of maintainability were their research interests, they introduced a clear definition to distinguish between software maintenance, and software maintainability as maintenance vs. maintainability reflect the process vs. quality metrics which in turn reflect the cost vs. quality metrics measurement respectively. Their focus was on the maintenance because of its impact in improving and avoiding future defects, and its role in reducing the total cost and time consuming during the whole software development life cycle stages. To predict and distinguish future improvement activities they used the systematic review to generate set of questions that could help to provide more details about the whole domain and suggest that there is a relationship between the software maintainability estimation and models.

According to English et al. (2009), maintainability is one of the most important factors that helps to save time and resources in the long term periods, they studied

maintenance by examine part of source code of any program that expected to have defects so it need to change. They did experimental study to have information about number, different faults and the desired changes that need to be applied on part of code. They use both Pareto's law and Kemerer metrics for analyzing the information and identifying classes that most likely to change, respectively.

According to Bernstein et al. (2007), a new technique discovered to predict the defects in any software in order to write bug-free software. They discussed that the temporal features of the data is able to prediction performance, also they used the non-linear models to discover the relationship between the defects and features which it may hidden. As a result of maintain the reliability of the prediction. They depended on an automated feature selection algorithm called tree-based induction, in order to predict the location of defect, and to predict a number of bugs.

According to Canfora and Cimitile (2000), Object technology has become growingly common in these days and the most of the new frameworks are presently being evolved with an object-oriented technology. Among the essential reasons for using an object-oriented technology is consolidated modifiability, and thus simpler maintenance. This is obtained through notions such as dynamic binding, classes, inheritance, information hiding, and polymorphism. But, there is no enough data that experimentally show the effect of object-oriented technology on maintenance.

According to Edberg et al. (2012), many sides of software maintenance operations are badly understood in spite of the fact that the plurality of resources for software development in most organisms is dedicated to maintenance. EDBERG's study showed that single developer perceptions and differences have a far greater effect in the selection of a maintenance methodology than is the situation for the selection of a

formal (initial) software development methodology. Participants in the study systematically profited private maintenance methodologies that were unified from elements of various initial development methodologies. Finding that initial education and training robustly impacted the expansion of these personal maintenance techniques (methodologies).

Xiong et al. (2011) looked into the stochastic demeanors of maintenance activities and operation of software frameworks. The demeanors are depicted under the frame of the Non-homogeneous Continuous Time Markov Chain (NHCTMC). Then the cost brought in by nonexistent time is examined. Discussing how to minimize the effect of unavailability by the optimality of maintenance policy is resolved and altering maintenance policy. A cost model is suggested for the objective of quantitative analysis. In addition, rate-based simulation is performed to simplify the research.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER THREE

STATIC CODE ANALYSIS TOOLS

3.1 Static Code Analysis Tools

Static SCA tools are software programs that collect information from the source code with the goal of trying to verify all potential tracks within a software program without executing the program. Certain characteristics will be statistically evaluated, based on predefined quality standards. A static code analysis tool should be able to efficiently locate faults such as under flow or over flow in an arithmetic operation, out-of-bounds array accesses, memory allocation errors and conflict code fragments that may go unnoticed during dynamic tests.

In the phases of software development can apply static code analysis in the early phases. And can be applied to the code are incomplete and incorrect, as there are no test cases must be. Unlike software testing where expected output will be (pass or fail) based on the conformance of expected outcome with the actual outcome. In SCA tools, the output will be one of three classes: error, warning or information.

In computer technology, a software bug is a fault in a computer program that causes an unexpected result or blocks it from operating properly. Some bugs may only impact a program under specific situations. Others may be more critical and cause the software code to be unsteady or even unusable. A simple failure in the code can cause serious problems. For example, if the programmer fortuitously wrote a code to add two numbers to each other when it should multiply them, the remnant of the code will give a

wrong result. Next sections are description of the three class of information that is provided by SCA tools.

3.1.1 Warnings

According to Slaughter and Delwiche (1995), Warning messages from diagnostic messages warns construction that cannot be wrong, but that is decisive, or indicate that there is potential for future errors in the program. Warnings are less terrible than errors. Although some professional programmers, try to decrease the number of warnings, sometimes the situations that result in warnings are not serious. Other situations may indicate serious problems which, if unfixed, will render the results valueless. All warnings should be checked to judge their seriousness.

In other words, warning is an issue with your program, happened when the compiler hits a statement that is valid but probably not what you meant. Warnings are not errors - the compiler can ignore them- and do not break compilation or block the compiler from generating code.

Although the warnings should not be ignored, they are not something serious enough to actually prevent the program from compiling. Usually, compiler warnings are hints that something might go erroneous at runtime. A typical mistake might be made that the compiler knows about. A popular example is using the assignment operator, '=', instead the equality operator, '==', inside a while statement. Other example is using variables that might not have been initialized.

Nevertheless, compiler warnings aren't going to halt the program working (unless program is told to treat warnings as errors), thus they are perhaps a bit less complicated than errors. In other words, the warning is a code statement looks suspect

and can be ignored. However, warnings usually indicate that a code statement is incorrect with the input file.

3.1.2 Errors

According to Slaughter and Delwiche (1995), errors are program statements that are definitely wrong, and that deny the compiler from finishing the compilation of the compiled program. These include lines or statements that are usually missing semicolons, spelling errors, or incorrect syntax. For instance, Lines of code in Java or C# should have (;). The compiler errors will always contain a line number at which the error was discovered. These types of syntax errors are called compiler errors.

There is another type of errors called Linker errors. Unlike compiler errors, errors are problems with the link determine the definition of structures, global variables, functions, or categories that were used, but did not know, in the source code file. Mostly, we will be link errors of the form "could not find a definition of X".

Generally, the compilation process will start with a chain of compiler warnings and errors, once all of them have been fixed, and then linker errors will be presented.

3.1.3 Information

Information messages are messages that will be generated when a function or a variable is declared while they are not used in the program. These messages inform about the status of the program such the number of records.

3.2 What can Static Code Analysis Accomplish?

According to Vink and BV (2010), the key reasons for using static code analysis are twofold. The first is to minimize the costs and time of developing source code. The second is to increase revenue and decrease business risk by supplying reliable and responsible software to customers. Static code analysis is used to forcefully direct the code in a way as to be readable, less prone to mistakes and reliable on future tests. This also impacts the verification of the source code after it is ready, minimizing the number of errors found in additional implementations of the source code.

According to Gomes et al. (2009), static code analysis is used to analyze of computer software which is accomplished without the execution of the codes, as contrary to dynamic analysis or testing - codes. Commonly, the analysis of computer software is performed on some version of the object code and in the other states on the source code. Programmers make little faults all the time, like an additional parenthesis here, a missing semicolon there, and so on. Most of the time these errors are illegal and will be rejected by the compiler. The compiler observes the error then the programmer repairs the code mistakes , However, to most safety vulnerabilities this is a rapid scenario of feedback and response which is not usually applied.

Static code analysis is used to recognize many common programming problems before a software program is released. Static analysis endeavors to check the text of a code statically, without trying to execute it. In theory, static analysis tools can check either a source code of a program or a program's compiled form.

According to Gomes et al (2009), static code analysis can be done using automated tools or manual reviewing. Static analysis tools are more efficient than manual reviews because they are faster. Programs can be evaluated much more

repeatedly, and some of the knowledge is encapsulated in the static code analysis tools required to perform this kind of code analysis.

Programmers may depend on a compiler to apply the finer points of programming language syntax. A perfect static analysis tool can effectively apply the tool without being conscious of the finer points of the more hard to detect bugs. Moreover, examining process for bugs is complicated because they often occur in hard-to-reach cases or exist in uncommon circumstances.

Static analysis tools can look more black corners of the program with fewer hiccups than dynamic code analysis, which requires the implementation of the code. On the other hand, static analysis has also the possibility of their application before the source code up to the level of completion of writing the code can test the glory of the application.

3.3 Analysis and Comparison: Source Code Analysis Tools

In this section, three SCA tools specially designed for Microsoft .NET programming language will be analyzed. These are FxCop, StyleCop, and JustCode. FxCop and StyleCop are Open-source products, but JustCode is a commercial product.

3.3.1 Analysis: Source Code Analysis Tools

A key difference between StyleCop and FxCop is that StyleCop analyzes C# source code, and cannot analyze another .NET language source codes. On the other hand, FxCop works for any .NET programming language after the source codes are compiled. StyleCop is interested in how C# source code looks, provides programmers with an efficient way to follow C# coding standards, focused on code style, comments, naming convention, spacing, etc.

FxCop focuses on how the .NET framework classes are used. It concentrates on the Microsoft Design Guidelines and analyzes the code seeking possible security and performance issues. In other words, FxCop and StyleCop are related; they complement each other, because each tool executes some different code analysis tasks. Despite their different rules, StyleCop can be compared with FxCop in that both are used as SCA tools.

3.3.1.1 StyleCop Tool

StyleCop is an open source static SCA tool for Visual Studio produced by Microsoft that checks C# source code to determine if it is correctly formatted. StyleCop analyze the code in order to enforce a set of styles and consistency rules which are classified into the following categories:

- Spacing
- Readability
- Ordering
- Naming
- Maintainability
- Layout
- Documentation

StyleCop includes both command line and graphical user interface versions of the tool. It is also possible to create new StyleCop rules to be used.

- **Spacing Rules**

Spacing rules apply spacing requirements around symbols and keywords in the source code. Table 3.1 shows examples of spacing rules.

Table3.1: Spacing rules and examples

Warnings	Example of code
The spacing around the keyword 'for' is invalid.	<code>for(int row = 0; row < bitmap.Height; row ++)</code>
Invalid spacing around the semicolon.	<code>Public CommonUtils.HistoryListInputHistory {get; set;}</code>
The spacing around the symbol '!=' is invalid.	<code>if(oBuffer!=null)</code>
The documentation header line must start with a single space.	<code>///loop through all connected chatters and invoke their</code>
The comment must start with a single space	<code>//for WPF Dispatcher</code>
The preprocessor type keyword must not be preceded by a space	<code># region InteropServices.Marshal methods</code>
Invalid spacing around the opening parenthesis	<code>if(oBuffer!=null)</code>
Invalid spacing around the closing parenthesis	<code>if(hDIB!=(int)0)</code>
Invalid spacing around the opening square bracket	<code>dsBooks.Tables ["Authors"].DefaultView</code>
Invalid spacing around the closing square bracket	<code>Trim(new char[] {\})</code>
Invalid spacing around the opening curly bracket	<code>Trim(new char[] {\})</code>
Invalid spacing around the closing curly bracket	<code>Trim(new char[] {\})</code>
Invalid spacing around the closing attribute bracket	<code>[DllImport("olepro32.dll", CharSet =CharSet.Unicode, ExactSpelling=true)]</code>
Invalid spacing around the negative sign	<code>(Filename,-1,null,null,0)</code>

The code contains multiple spaces in a row. Only one space is needed	<code>Application.StartupPath + "\\errors.log"</code>
Tabs are not allowed. Use spaces instead	<code>currentSession.Abort();</code>

All warnings in the Table 3.1 will be explained, one by one, the first one is “The spacing around the keyword 'for' is invalid”. This warning is caused when the spacing around a C# keyword is improper. There are some C# keywords that must always be followed by a single space, compared with other keywords must not be followed by any space.

These C# keywords must always be followed by a single space: `stackalloc`, `catch`, `foreach`, `from`, `group`, `if`, `where`, `in`, `fixed`, `for`, `into`, `join`, `let`, `lock`, `return`, `select`, `orderby`, `switch`, `throw`, `using`, `while`, `yield`. Compared with following keywords must not be followed by any space: `default`, `checked`, `sizeof`, `unchecked`, `typeof`. The new keyword should be followed by a space or not depend on the code sentence, always should be followed by a space unless it is used to create a new array. In this case no space should be between the opening array bracket and the new keyword.

On the other hand, the second warning is “Invalid spacing around the semicolon”. This warning is resulted from an incorrect spacing around a semicolon. Unless the semicolon is the last character on the code line, A single space should be constantly preceded it, and unless it is the first character on the code line, it shouldn't be preceded by any whitespace. In order to solve a contravention of this rule, the

semicolon should not be preceded by any space, and it should be followed by a single space.

According to the following warning in from Table 3.1, “The preprocessor type keyword must not be preceded by a space”. This warning is caused when a C# preprocessor-type keyword is preceded by a space. For example:

```
# region InteropServices.Marshal methods
```

The warning “Invalid spacing around the opening parenthesis” is resulted from an incorrect space for an opening parenthesis inside a C# code statement. If an opening parenthesis is the first character on the line, or it is preceded by certain C# keywords such as while, if, or for, it should be preceded by any whitespace. When an opening parenthesis is preceded by an operator symbol inside an expression, a whitespace is permitted to be followed by an opening parenthesis. Moreover, “Invalid spacing around the opening square bracket” warning is result from incorrect space for a closing parenthesis inside a C# code statement.

A closing parenthesis should never be preceded by whitespaces. In most cases, a closing parenthesis should be followed by a single space, unless the closing parenthesis comes at the end of a cast, or the closing parenthesis is followed by certain types of operator symbols, such as positive signs, negative signs, and colons.

The warning “Invalid spacing around the opening square bracket” is resulted from incorrectly spaced for an opening square bracket inside a C# code statement; then Whitespace is preceded or followed by an opening square bracket inside a statement. A whitespace must be followed by an opening square bracket just in these suitcases, if it is the first or last character on the line; moreover, when the spacing around a closing square bracket is spaced incorrectly, the “Invalid spacing around the closing square

bracket” warning occurred. If a closing square bracket is the first character on the line, a space must be followed by a closing square bracket.

The warning “Invalid spacing around the opening curly bracket” is resulted from incorrectly spaced for an opening curly bracket inside a C# element, when this occurs a single whitespace must all the time become before An opening curly bracket, nevertheless, a single whitespace must be followed by An opening curly bracket in two conditions: if it is the first character on the line, or when an opening parenthesis is followed by An opening curly bracket, and in this suitcase there shouldn't be space between the parenthesis and the curly bracket. A space must be always preceded by an opening curly bracket, but if an opening curly bracket is the last character on the line, a space must not be preceded by an opening curly bracket.

The warning “A closing curly bracket within a C# element is not spaced correctly.” is resulted from a wrong spacing around a closing curly bracket, when this occurs a singular space must all the time be preceded by a closing curly bracket; however, a space must not be preceded by a closing curly bracket if and only if a closing curly bracket is the last character on the line, or if a comma, a semicolon, or a closing parenthesis is preceded by a closing curly bracket.

The warning “Invalid spacing around the closing attribute bracket.” is resulted from a wrong space around a closing attribute bracket. If the bracket is the first character on the line, a space must be followed by a closing attribute bracket.

The warning “Invalid spacing around the negative sign” is resulted from a wrong space around a negative sign. If a negative sign is the first character on the line preceded by an opening square bracket, or a parenthesis, a single space must not all the time be followed by a negative sign.

The warning “Tabs are not allowed. Use spaces instead” is resulted from consisting of a tab character in the code. Based on the editor that are used to display the code, the extent of the tab character can be alternated, and as a result for that Tabs have not to be used inside C# code, and this is one of the reasons that can cause spacing and indexing of the code differ from the original intention of the developers, and in some cases the reader may be seen the code difficult. Intended for these reasons Tabs should not be used and are not permitted, and four spaces should be contained in every indentation level. This will pledge that the code appears similar, and no affair which editor is being used in order to display the code.

The warning “The comment must start with a single space” is resulted from not started a single-line comment in a C# code file with a single space, and at what time a single-line comment does not begin with a single space, the contravention of this rules happen. For example:

```
private void Method1()
{
    //for WPF Dispatcher
    // for WPF Dispatcher
}
```

The comments should start with a single space after the forward slashes:

```
private void Method1()
{
    //for WPF Dispatcher
    //for WPF Dispatcher
}
```

Exclusion to this rule occurs when the comment is being used to comment out a line of the code program. In this case, the space can be deleted if the comment starts with four forward slashes to denote out-commented code. Such as:

```
private void Method1()
{
    ///int x = 2;
    ///return x;
}
```

The warning “Invalid spacing around the negative sign” is resulted from an incorrect space around an operator inside a C# code file, these operators kinds have to be cuddled by a single space on one of the sides: arithmetic operators, relational operator, logical operators, lambda operators, conditional operators, colons, assignment operators, and shift operators. For example:

```
(Filename,-1,null,null,0)
```

The warning “The documentation header line must start with a single space” is caused when a code line within a documentation header does not begin with a single space. For example:

```
///loop through all connected chatters and invoke their
```

The header lines should start with a space after the three leading slashes:

```
/// loop through all connected chatters and invoke their
```

- **Readability Rules**

These types of Rules are used to guarantee that the code is readable and well-formatted. Table 3.2 shows examples of readability rules.

Table3.2: Readability rules and examples

Warnings	Example of code
Calls to members from a base class should not begin with 'base.'	<code>return base.Channel.BeginJoin(name, callback, asyncState)</code>
Prefix local calls with this	<code>AbortProxy()</code>
The code contains an extra semicolon	<code>};</code>
A line may only contain a single statement	<code>catch (TimeoutException) { }</code>
A comment may not be placed within the bracketed statement	<code>else // small flake</code> <code>{</code>
All method parameters must be placed on the same line	<code>(Settings.Option.LogFileName, FileMode.Append)</code>
The comment is empty. Add text to the comment or remove it	<code>//</code>
Use the built-in type alias 'int' rather than Int32 or System.Int32.	<code>Int32[] baudRates</code>
Use string.Empty rather than ""	<code>(titleAttribute.Title != "")</code>

The warning “A comment may not be placed within the bracketed statement” is caused when a C# line code includes a comment between the opening curly bracket and the declaration of the statement. For example:

```
else // small flake
{
```

The comment can be placed or within the body of the block:

```
else
{
    // small flake
```

Or can be placed above the statement:

```
// small flake
else
{
```

The warning “A line may only contain a single statement” is caused when a single line Within a C# code contains more than one statement. For example:

```
catch (TimeoutException) { }
```

So that, each statement must begin on a new line.

The warning “The code contains an extra semicolon” is caused when The C# code contains an additional semicolon. For example:

```
};
```

This results in an empty statement in the code.

The warning “Calls to members from a base class should not begin with ‘base.’” is caused when a call to a class member (functions) from an inherited class (base class) starts with ‘base.’, furthermore the local class does not include an implementation or override of the member (function). For example:

```
return base.Channel.BeginJoin(name, callback, asyncState)
```

The warning “Prefix local calls with this” is resulted when a call to an instance member of a child class or a parent class that is involved in a C# code doesn't start with 'this'. Elimination to this rule occurs at the time there is a local (child) take priority over the parent class element, and the code means to identify the parent class element

directly, avoiding the local (child) priority over the parent class element. In this situation the call can be beginning with 'base.' rather than 'this.'.

The warning "All method parameters must be placed on the same line" is caused when the parameters to a C# declaration or indexer or method call each on a separate line or are not all on the same line. For example:

```
(Settings.Option.LogFileName,  
 FileMode.Append)
```

The parameters must all be placed on the same line:

```
(Settings.Option.LogFileName,FileMode.Append)
```

The warning "The comment is empty. Add text to the comment or remove it" is caused when the C# code includes a C# comment which does not contain any comment text.

The warning "Use string.Empty rather than """" is caused when the C# code contains a hard-coded empty string. For example:

```
(titleAttribute.Title != "")
```

This will cause an empty string was embedded into the compiled code by the compiler. So that rather than using an empty string, use the static string.Empty field to represent it, like this:

```
(titleAttribute.Title != string.Empty)
```

The warning "Use the built-in type alias 'int' rather than Int32 or System.Int32." is caused when the code uses one of the basic C# types anywhere in the code, but does not use the built-in alias for the type. For example:

```
Int32[] baudRates
```


Rather than using the fully-qualified type name, the alias for this type should always be used:

```
int[] baudRates
```

- **Ordering Rules**

These types of Rules are used to apply a standard ordering scheme for code program contents. Table 3.3 shows ordering rules.

Table3.3: Ordering rules

Warnings
All using directives must be placed inside of the namespace
All methods must be placed after all fields
All private fields must be placed after all public fields
All constant must be placed before all non-constant
Using directives must be sorted alphabetically by the namespaces

The warning “Using directives must be classified alphabetically through the namespaces” is happened at the time the used of orders in the file of a C# program are alphabetically not prepared. Organizing the used orders alphabetically has the ability to make the code easier to read and cleaner.

The warning “All constant must be placed before all non-constant” is caused when a constant field is placed below a non-constant field. Non-Constants fields must be placed below constants fields.

The warning “A get accessor appears after a set accessor” is resulted from the appearance of a set accessor before A get accessor inside indexer or a property. An infringement of this rule happens when a set accessor is located before a get accessor inside indexer or a property.

The warning “All using directives must be placed inside the namespace” is resulted when a C# using directive or a using-alias directive comes into view outside the elements of namespace; if not the C# code involve any elements of a namespace.

There are slight dissimilarities among insertion using directives outside of the namespace, rather than inside the elements of a namespace, including:

1. Placing the using directives within the namespace removes compiler embarrassment between contradicting types.
2. When various namespaces are placed within a single file, placing using directives within the namespace elements fields aliases and references.

- ***Naming Rules***

These types of Rules are used to enforce naming requirements for types, members and variables. Table 3.4 shows examples of naming rules.

Table3.4: Naming rules and examples

Warnings	Example of code
method names begin with an upper-case letter: button1_Click	<code>private void button1_Click(object sender, EventArgs e)</code>
The variable name 'iCount' begins with a prefix that looks like Hungarian notation	<code>int iCount</code>
Variable names must start with a lower-case letter	<code>(string Section, string Key)</code>
Public and internal fields must start with an upper-	<code>public string path</code>

case letter	
Field names must not start with an underscore	<code>SerialPort_serialPort</code>

The warning “Public and internal fields must start with an upper-case letter” is caused when public or internal field name in C# program does start with a lower character. If the variable field or name is intended to exchange the name of an item connected with Win32 or COM, and therefore need to begin with a character of lowercase, rest the field or variable inside a particular NativeMethods class. A NativeMethods class is any class which contains a name ending in NativeMethods, and is intended as a placeholder for Win32 or COM wrappers. StyleCop will ignore this infringement if the item is set inside a NativeMethods class. For example:

`public string path`

this public field must start with an upper-case letter , like this :

`public string Path`

The warning “Variable names must start with a lower-case letter” is resulted when the name of a field in C# or variable does not begin in a character with a lower-case. On the other side, non-private read-only and static read-only fields have all the time to start in a character with an uppercase, at the same time as private read-only fields have to start in a character with a lowercase. In addition, internal or public fields have all the time to start in character with an uppercase. For example:

`(string Section, string Key)`

variable names (Section, Key) must begin with a lowercase character, like this:

`(string section, string key)`

The warning “Field names must not start with an underscore” is caused when a field name in C# starts with an under strike. By default, StyleCop disallows the usage of m_, underscores, etc. For example:

```
SerialPort _serialPort
```

field names must not begin with an under strike, like this:

```
SerialPort serialPort
```

The warning “The variable name 'iCount' begins with a prefix that looks like Hungarian notation” is resulted when the name of a field or variable in C# code operates Hungarian notation. The usage of Hungarian notation has been predominant in C++ code; nevertheless the inclination in C# is to use more denominative, longer indication for variables, which are not based on the type of the variable but describe the reasons of using variable.

In addition, new source code editors such as Visual Studio make it easier to identify kind information for a field or variable, by hovering the mouse pointer over the variable name. This minimizes the requirement for Hungarian notation.

StyleCop presumes that a variable name that starts with one or two lowercase characters followed by an uppercase character is making utilization of Hungarian notation. It is probable to declare specific prefixes as legal, in which situation they will be disregarded. Such as a variable which has name “onExecute” will seem to StyleCop to be utilizing Hungarian notation, when in fact it is not. So, the on prefix should be gestured as an acceptable prefix.

The warning “method names begin with an upper-case letter” is resulted when the name of exact kinds of a C# code component does not begin in character with an uppercase. These kinds of components are supposed to use character with an uppercase

as the initial character of the component name: enums, structs, delegates, namespaces, properties, classes, events, and methods.

In addition, any field which is marked with the const attribute, public, or internal should start with an uppercase character. Read-only Non-private Fields must also be named utilizing an uppercase character.

- **Maintainability Rules**

These types of Rules are used to improve code maintainability. Table 3.5 shows examples of maintainability rules.

Table3.5: Maintainability rules and examples

Warnings	Example of code
The line contains unnecessary parenthesis	<code>double percentFailed = (numErrorFiles / numFilesProcessed)</code>
The class must have an access modifier	<code>static class Program</code>
Fields must be declared with private access. Use properties to expose fields	<code>public static string estimatingMessage</code>
A C# code file contains more than one unique class	_____

The warning “The line contains unnecessary parenthesis” is caused when a C# statement have parenthesis in which there is no need for them and are supposed to be erased. It is probable in C# code to put parenthesis in the region of any kind of expression, clause, or statement, and in many suitcases use of parenthesis that have the ability to advance the readability of the C# code; however, the use of parenthesis in an excessive way make it more difficult to maintain and read the code, and it may have the contradictory result. For example, the following line of C# code contains unnecessary:

```
double percentFailed = (numErrorFiles / numFilesProcessed)
```

The extra parenthesis can be deleted without affecting the readability of the code:

```
double percentFailed = numErrorFiles / numFilesProcessed
```

The warning “The class must have an access modifier” is caused when the access modifier for the component of a C# code for instance a class has not been identified in a clear way. In C# language the components are allowed to be identified with no need for an access modifier (public, private). An access level will be unexpectedly specified to the component of this situation by C#, depending on the type of component. An access modifier is demanded for this rule to be identified in a clear way for each component. This takes out the demands for the reader to make assumptions about the program of C#, improving the readability of the C# code.

The warning “Fields must be declared with private access. Use properties to expose fields” is caused when a field within a C# code class has a non-private access modifier such as public. For example:

```
public static string estimatingMessage
```

The warning “A C# code file contains more than one unique class” is resulted when the file of a C# program involve more than one single class. The class name in a file is supposed to be replicated by the name of the file, and each class is supposed to be placed in its own file in order to elevate the maintainability of long-term of the code. If the other components are supported to the class or referred to the class. It is probable to place other elements inside the same file the as enums as class, delegates, etc. moreover, it is probable to place deferent sections - of the same fractional class - inside the same file.

- **Layout Rules**

These types of Rules are used to enforce code line spacing and layout. Table 3.6 shows layout rules.

Table3.6: Layout rules

Warnings
If a statement spans multiple lines, the closing curly bracket must be placed on its own line
A statement containing curly brackets must not be placed on a single line
The body of the if statement must be wrapped in opening and closing curly brackets
The code must not contain multiple blank lines in a row
A closing curly bracket must not be preceded by a blank line
Statements or elements wrapped in curly brackets must be followed by a blank line
Adjacent elements must be separated by a blank line
The code file has blank lines at the end

The warning “The body of the “if statement” must to be wrapped in an opening and closing curly brackets” is resulted when the opening and closing curly brackets for a statement that are blocked has been missed. Some types of statements is might be facultatively involved curly brackets like if, for, and while statements In C# language.

For example:

```
if (true)
    return this.value;
```

This if-statement was written without curly brackets, although this is valid in C#, StyleCop always needs the curly brackets to be written, to increase the maintainability and readability of the C# code.

When the curly brackets are missed, it is probable to be an error in the code by writing another statement within the if-statement block. For example:

```
if (true)
    this.value = 2;
    return this.value;
```

The warning “A closing curly bracket must not be preceded by a blank line” is caused when a blank line precede a closing curly bracket within a C# expression, element, or statement.

The warning “The code must not contain multiple blank lines in a row” is resulted when a multiple blank lines are involved in the C# code in one row. Blank lines are demanded by StyleCop in particular suitcases and they are prevented in other suitcases in order to enhance the code readability, and as a result for that the readability and recognition of unfamiliar code can be improved.

The warning “A statement containing curly brackets must not be placed on a single line” is caused when a C# statement including closing and opening curly brackets is written on a one line. For example:

```
public object Func()
{
    lock (this) { return value; } }
```

The warning “If a statement spans multiple lines, the closing curly bracket must be placed on its own line” is caused when the closing or opening curly bracket within a C# expression, statement, or element is not located on its own line.

The warning “Statements or elements wrapped in curly brackets must be followed by a blank line” is caused when a closing curly bracket is not followed by a blank line.

The warning “Adjacent elements must be separated by a blank line” is caused when there is no blank line between two adjacent elements.

The warning “Adjacent the code file has blank lines at the end” is caused when there are blank lines at the end of the code. StyleCop needs no blank lines at the end of codes, to improve the layout of the code.

- **Documentation Rules**

These types of Rules are used to verify the formatting and the content of C# code documentation. Table 3.7 shows documentation rules.

Table3.7: Documentation rules

Warnings
A C# code element is missing a documentation header
The partial class element must have a documentation header containing either a summary tag or a content tag
The enumeration sub-item must have a documentation header
The Xml within a C# element’s document header is badly formed
A C# method, constructor, delegate or indexer element is missing documentation for one or more of its parameters
The documentation describing the parameters to a C# method, constructor, delegate or indexer element does not match the actual parameters on the element
A <param> tag within a C# element’s documentation header is empty
The documentation text within a C# property’s <summary> tag does not

match the accessors within the property
A section of the Xml header documentation for a C# element does not contain any whitespace between words
A C# code file is missing a standard file header
The Xml documentation header for a C# constructor does not contain the appropriate summary text
A section within the Xml documentation header for a C# element contains blank lines

The warning “The Xml within a C# element’s document header is shaped in a bad way” is resulted when the Xml inside the file header of a C# component cannot be analyzed and it is formed in a bad way. This may occur if the Xml involves characters that are invalid or if a closing tag is being lost by an Xml node. Throughout the use of Xml documentation headers, C# syntax introduces a method for inserting documentation for classes and components immediately into the C# code.

The warning “The partial class element must have a documentation header containing either a summary tag or a content tag” is caused when the header is empty, or if a C# partial element is entirely missing a documentation header.

3.3.1.2 JustCode Tool

- **Naming Rules**

There are some warnings will be considered in this section, the first one is related to naming reasons, is occur when a namespace name matches the name of the project, while the name of the project is “ChatService” but the name of the namespace is

“Chatters”, so that to avoid this type of warning should be converted the namespace name to “ChatService”.

Another type of warning- related to naming reasons - is occur when an interface name does not resemble the file name, to avoid this type of warning should be changed the file name to be similar to interface name.

According to the naming convention, if the access modifier is “public” then the first letter should be capital. A violation of this rule occurs when the field name do not begin with an upper-case letter, to solve it should be capitalized the first letter, and if the access modifier is “private” then the first letter of the field name should be lowercase and the first letter of the method name should be uppercase. A violation of this rule occurs when the field name does not begin with lowercase letter. These kinds of elements should use an uppercase character as the first character of the element name: public, namespaces, internal, classes, enums, and const structs.

- ***Readability Rules***

There is a warning occurs when there is an extra semicolon within the code, this results in an empty statement in the code. To fix a transgression of this rule, the unneeded semicolon should be removed.

- ***Using Rules***

There is also a warning occurs when there is a directive within the file which was never used by any element in the project, such as collection, generic directive. Another warning occurs when there is a variable, method, parameter has been declared, and however, they are not used in the program. Also there is a warning occurs when there is a field used however it is not initialized in the program.

3.3.1.3 Fxcop Tool

▪ *Naming Rules*

The warning “Identifiers should be spelled correctly” is caused when an Identifier is not understood by the “Microsoft spelling checker library”. In other words, the personal words that make an identifier are abbreviated or are not spelled correctly. This rule analyzes the identifier into parts and investigates the spelling of each part. The parsing algorithm depends on the following rules:

- Upper-case characters begin a new token. Such as, MyNameIsJoe divides into "My", "Name", "Is", "Joe".
- For multiple Upper-case characters, the last Upper-case characters begin a new token. Such as, GUIEditor divides into “GUI”, "Editor".
- Trailing and Leading apostrophes are deleted. Such as, 'sender' divides into “sender”.
- Underscores mean the end of a token and are deleted. Such as, Hello_world divides into “Hello”, "world".
- Embedded ampersands are deleted. Such as, for&mat divides into "format".

The warning “Resource strings should be spelled correctly” is caused when a resource string includes one or more words that are not understood by the “Microsoft spelling checker library”. This rule divides the resource string into terms, dividing compound words, and investigates the spelling of each term/token. In other words, the personal words that make a resource string should be spelled rightly, and should not be abbreviated.

- ***Performance Rules***

The warning “Avoid unused private fields” is caused when a private field in the program exists but is not utilized by any code track. For example, declaring the field 'PluginFamily._policy', but it are never used or are only ever assigned to.

The warning “Initialize reference type static fields inline” is caused when a reference type states an explicit static constructor. When a type states a frank static constructor, the Just-In-Time (JIT) compiler adds a test to each instance constructor and static method of the type to make certain that the static constructor was already called. Static initialization is elicited when an instance of the type is made or when any static member is accessed. But, static initialization is not elicited if a variable is declared of the type but do not utilize it.

The warning “Properties should not return arrays” is caused when a protected or public property in a public type returns an array. An Array returned by protected or public properties - even if the property is read-only - are not write-protected. To save the array tamper-proof, a copy of the array must be returned by the property.

The warning “Remove unused locals” is caused when a local variable is declared within a method but the method does not utilize the variable except perhaps as the recipient of an assignment statement. For dissection by this rule, the analyzed assembly must be constructed with debugging information and the associated program database PDB file must be existed.

3.3.2 A Comparison Between The Tools

After carrying out the analysis on the data which generated when 40 project codes – every project code contain at least 10 files - were applied on these tools (StyleCop, JustCode, FxCop), and after collect the results in a dataset, then each XSL file contains at least 500 warnings, after the analysis, it is concluded that the StyleCop tool has seven types of warnings: layout, documentation, ordering, naming, readability, spacing, and maintainability. On the other hand, the JustCode tool has three types of warnings: naming, usage, and readability. And FxCop has many types of warnings, but 3 types were considered on this study: naming, performance, and usage.

Firstly, we will compare between the results, this comparison related to the types of warnings, according to the naming warning, if we look at the table 5 and compare the results to the naming warning –in JustCode and FxCop tools - it is concluded that JustCode and StyleCop tools both contain the same warning “name does not match the naming convention” this according to the JustCode tool, but StyleCop there are many rules but all of them considered as one rule in just code tool. As for FxCop, this rule does not exist.

On the other hand, there are many differences between these tools such as, there is a rule in StyleCop rule that say, “field names must not start with an underscore”, and this rule does not exist in JustCode and FxCop. As for JustCode, the first three rules are similar, and they say that the element name within C# code does not match the files and project name, but as for FxCop the first two rules are different from other tools results.

As for the usage rule, this rule exists only in JustCode and FxCop tools but does not exist in StyleCop tool, in JustCode tool; there are some rules such as:

- 1- Field is only assigned.
- 2- Variable is only assigned.
- 3- Unused method.
- 4- Unused parameter.

These rules are similar to a rule in FxCop tool, “review unused parameter”.

As for the difference between the JustCode and FxCop tools, in JustCode there are two rules:

- 1- This cast is not required.
- 2- Field is never assigned.

In FxCop tool, the rules are:

- 1- Do not call overridable method in constructors.
- 2- Do not ignore method results.

As for readability rule, there is a similarity between StyleCop and JustCode, in the rule that say “the code contains an extra semicolon” this rule in StyleCop tool and in “this empty statement may be not intended here” this rule in JustCode tool. Also there is a rule exists in JustCode tool but not in StyleCop that say “field can be made read only”, however, FxCop tool does not contain readability rules. And there are 5 rules exist in StyleCop but not exist in JustCode.

As for performance rules, this rules exist in FxCop tool but does not exists in both JustCode and StyleCop tools. Also there are rules exist in StyleCop tool but does not exists in JustCode and FxCop tools, they are:

- 1- Spacing.
- 2- Ordering.
- 3- Maintainability.

- 4- Layout.
- 5- Documentation.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER FOUR

RESEARCH GOALS AND APPROACHES

In this chapter, we have described the major goals specified to guide the experiments. We have also described research approaches or steps taken in trying to test our proposed research approach.

As we have mentioned in previous chapter earlier, this project focuses on evaluating source code metric tools, based on their strengths and weaknesses. Particularly, we focused on two major SCA tools: MS StyleCop and JustCode. Both are popular and evaluate the different classes of warning we described earlier. Figure 4.1 summarizes the research procedures.

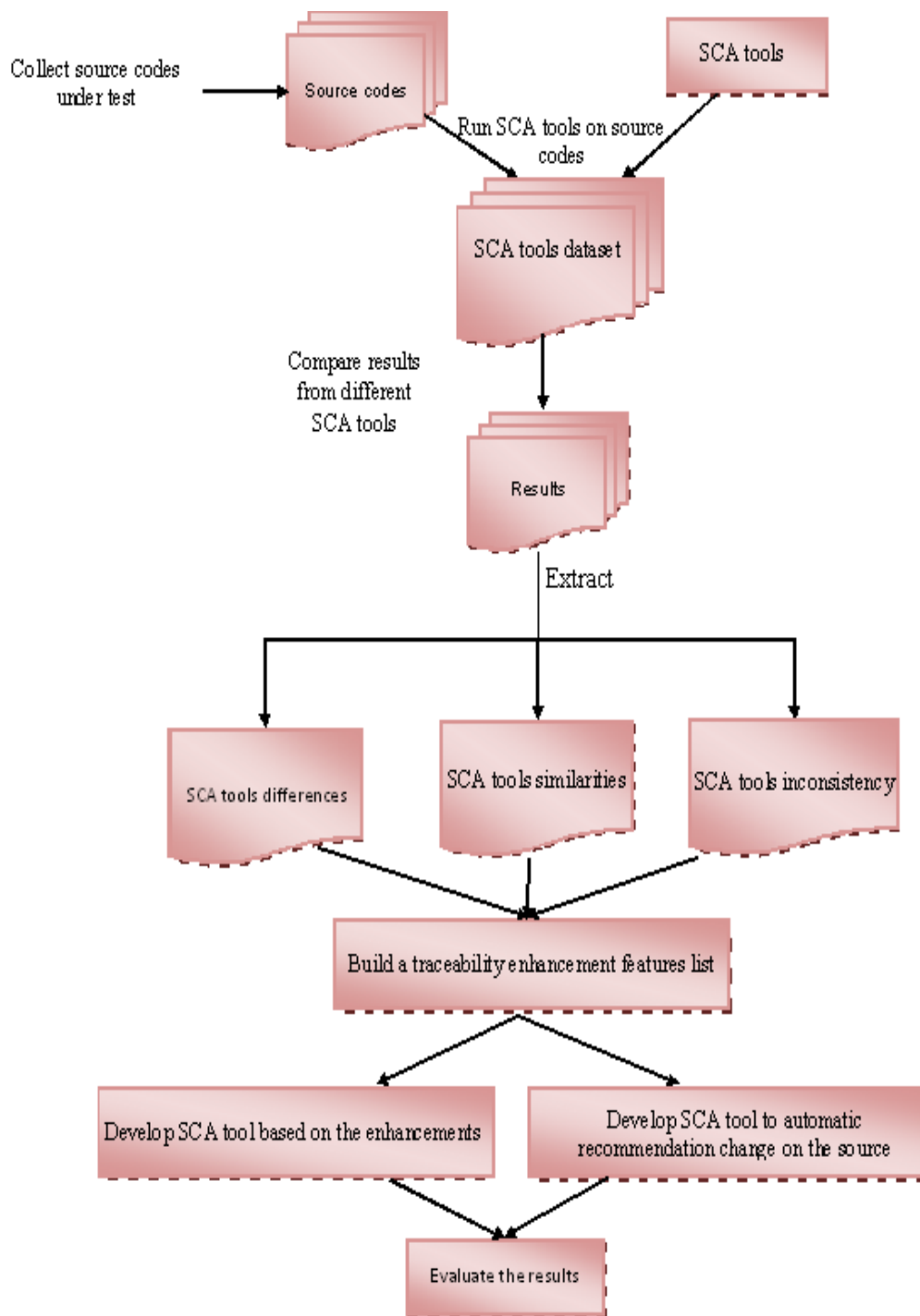


Figure 4.1: Methodology phases

The methodology includes the following six main steps:

1. In initial assessment of these tools, we noticed some differences in the results or the warnings that come from each tool for the same source code. This was one of the first problems triggered that we decided to investigate thoroughly.
2. We have developed our own SCA tool. We hope that this tool can overcome some of the weaknesses of the two evaluated tools: StyleCop and JustCode. We may not be able to solve all open issues, especially as we noticed that some issues are open not because the tools cannot solve them but because they are also open in the software development community. For example, in defining relations and their limits: parent-child, relations- by extensions or transitive relations, visibility issues. This was a second major task that will be evaluated in this thesis.
3. We also noticed that the same tool may give different number of warnings for the same source code if tested or run more than one time. Such inconsistency of results need to be evaluated and the kind of warnings that may vary from one experiment to another given the same tool and tested source code. This was a third major task that will be evaluated in this thesis.
4. Another important aspect that we have focused in our developed tool is the automatic implementation of suggested warnings and their solutions. Some SCA tools do not propose solutions. Some tools propose solutions with some problems. Tools may not have the ability to apply and evaluate applying proposed solutions for the warnings. We will tackle this issue in our developed tool, by proposing and applying warning possible solutions.
5. Of course, there is a step of evaluation for our own tool using different class files and possible codes.

6. We have compared the developed tool with StylCop and Just code.

4.1 Differences in the Results that come from each Tool for the Same Source Code

As mentioned earlier, the major goal in this section is to compare and conduct an assessment of selected SCA tools. In order to do this, the experimental study is performed with two open-source code-projects, implemented in C#. Table 4.1 shows an overview of the evaluated source codes. The first code-project is “Chatters” that contains 15 files and the total number of LOC is 2506 lines. On the other hand, “Design” code-project contains 23 files, and the total number of LOC is 2702 lines.

Table 4.1: An overview of the projects

Name	LOC	Files
Chatters	2506	15
Design	2702	23

After applying these SCA tools on the selected open-source code-projects, some differences were observed in the warnings that were generated from each tool for the same code-project. This was one of the issues that we determined to investigate thoroughly. Firstly, the “Chatters” project was resolved. According to StyleCop tool, the total number of warnings was generated is 555. However, according to JustCode tool, the total number of warnings was generated is only 103, as for, the “Design” project. According to StyleCop tool, the total number of warnings was generated equal 921. However, according to JustCode tool, the total number of warnings was generated is only 95.

As for, the distribution of warnings on the class warnings; this distribution is different from one project to another, this issue was expected, Table 4.2 and Table 4.3

show this distribution, and it is observed that, in Table 4.3 no readability warning was generated.

Table 4.2: Distribution the Chatters warnings on classes of warning

StyleCop Class Warnings	Chatters Project		JustCode Class Warnings
Naming	20	53	Naming
Readability	141	3	Readability
Maintainability	22	47	Usage
Spacing	58	—————	—————
Ordering	124	—————	—————
Layout	77	—————	—————
Documentation	112	—————	—————
Total	555	103	—————

Table 4.3: Distribution the Design warnings on classes of warning

StyleCop Class Warnings	Design Project		JustCode Class Warnings
Naming	65	69	Naming
Readability	302	—————	Readability
Maintainability	9	26	Usage
Spacing	—————	—————	—————
Ordering	240	—————	—————
Layout	75	—————	—————
Documentation	230	—————	—————
Total	921	95	—————

The results presented in the Table 4.2 and the Table 4.3 show the differences between numbers of class warnings in each of SCA tools after these tools on different

code projects. In Table 4.2 show that the number of spacing warnings equal to 58, but the number of spacing warning in the Table 4.3 equal to 0. From this result we return to the code of Design Project and we found that the line was not any warning because it did not achieve any of the rules of the spacing warnings.

The results presented in the Table 4.4 and the Table 4.5 show the similarities and the differences respectively between the results that were generated from applying the two SCA tools on the two code-projects. Table 4.4 shows the results were generated from applying the two SCA tools on Chatters project.

Table 4.4: Result from applying SCA tools on Chatters project

Example Code Chatters	JustCode Recommendation	StyleCop Recommendation
<code>public MessageType msgType</code>	<code>public MessageType MsgType</code>	<code>public MessageType MsgType</code>
<code>private string imageURL</code>	<code>private string imageUrl</code>	—————
<code>void lblExit_MouseDown</code>	<code>void lblExitMouseDown</code>	—————
<code>namespace Chatters</code>	<code>namespace ChatService</code>	—————
<code>public enum CallBackType</code>	Move Type to Another File	—————
<code>private static Object syncObj</code>	<code>private static readonly Object syncObj</code>	—————
<code>using System.Collections;</code>	Remove unused using	—————
<code>Receive(e.person.Name, e.message);</code>	<code>this.Receive(e.person.Name, e.message);</code>	<code>this.Receive(e.person.Name, e.message);</code>
<code>public string message=""</code> ;	—————	<code>public string message = ""</code> ;
<code>class Program</code>	—————	<code>public class Program</code>
<code>public Person person;</code>	—————	<code>private Person person;</code>

Table 4.5 shows the results were generated from applying Design project.

Table 4.5: Result from applying SCA tools on Design project

Example Code Design	JustCode Recommendation	StyleCop Recommendation
<code>void okButton_Click</code>	<code>void OkButtonClick</code>	<code>void OkButton_Click</code>
<code>private bool _accepted</code>	<code>private bool accepted</code>	<code>private bool accepted</code>
<code>interface Searchable</code>	<code>interface ISearchable</code>	<code>interface ISearchable</code>
<code>void webBrowser1</code>	—————	<code>public void webBrowser1</code>
<code>SetBackgroundColor(BackColor)</code>	—————	<code>SetBackgroundColor(this.BackColor)</code>
<code>interface SearchableBro</code>	<code>interface SearchDialog</code>	—————
<code>partial class SearchDialog</code>	—————	<code>public partial class SearchDialog</code>

As observed from the results, as for the “Naming Warning”, the two used SCA tools have some rules, such as “public field” name must start with a capital letter. In addition, underscore must be removed from field names, as for the function name, the tools presumes that it must start with a capital letter. But the difference between them, that JustCode disallows the underscores, on the other hand, StyleCop allows them. For example as shown in Table 4.5:

`void okButton_Click`

As for JustCode recommendation (underscore was removed, the first letter was capitalized):

`void OkButtonClick`

And as for StyleCop Recommendation (Underscore was not removed, the first letter was capitalized):

`void OkButton_Click.`

As for changing the type name, such as interface and namespace, JustCode requires to change the namespace name – or any type name such as class, struct, interface and enumeration – to match file name or folder directory name, or transfer it to a file that commensurate with it, such as “Chatters” namespace, should be converted to “ChatService”. As shown in Table 4.4:

namespace Chatters

As for JustCode Recommendation:

namespace ChatService.

As for using warning, this warning does not exist in the StyleCop tool, but in JustCode, this rules requires deleting unused using system, such as using System.Collections;

It was not used by any element in the project, so as for JustCode recommendation, it should be removed.

As for readability rules, private fields – according only to JustCode recommendation – must be followed by readonly keyword, as example shows in Table 4.4:

private static Object syncObj

As for JustCode recommendation (as shown below, it was added after static keyword directly):

private static readonly Object syncObj

In the existence of the access modifiers to each element and the field must be private, this rule is found only in StyleCop, and there are some examples in the both Tables, such as shown in Table 4.4:

`class Program`

As for StyleCop recommendation:

```
public class Program
```

Another example also shown in Table 4.5:

```
partial class SearchDialog
```

As for StyleCop recommendation:

```
private Person person;
```

4.2 Weaknesses of the Two Evaluated Tools

There are some weaknesses of SCA tools such as: generating false positive results, continuous inability to find configuration problems; because they are not represented in the code, difficulty to confirm that an identified security problem is a practical vulnerability. Many of SCA tools have difficulty analyzing source codes that cannot be compiled, and many types of security weaknesses are very hard to locate automatically, such as access control problems, authentication issues, etc.

The two SCA tools (JustCode, StyleCop) are applied on “MarsMission” project, and then some warnings (rules) were observed that may be due to some expected errors. As shown in table 4.6 and table 4.7 respectively, those show some of these warnings and tools recommendations.

Table 4.6 shows the process of applying JustCode Recommendation, and what are the results after applying this process.

Table 4.6: Example code MarsMission and JustCode recommendation

No.	Example Code MarsMission	JustCode Recommendation
1	<code>public int intWidth</code>	<code>public int IntWidth</code>
2	<code>struct udtWordImageLine</code>	Rename the file name to <code>udtWordImageLine</code>
3	<code>struct udtChemSymbols</code>	Rename the file name to <code>udtChemSymbols</code>
4	<code>Point ptRotateCopy;</code>	Field 'ptRotateCopy' is never assigned
5	<code>using Mars_Mission;</code>	

As shown, JustCode recommends capitalizing the first letter in “intWidth” field to be “IntWidth”, and so this capitalization process may cause an error, if there is another variable or field has the same name “IntWidth”. In other words, C# language is case-sensitive this mean that word “intWidth” is not the same as its first-capital spelling, “IntWidth”. They are totally different identifiers. If there is already a field its name is “IntWidth” in the code, and then the JustCode capitalizes the first letter in “intWidth” field to be “IntWidth”, and then they will be two variables with the same name.

As for the recommendations 2 and 3 in Table 4.6, JustCode recommends renaming the file name that contains a structure to the structure name, but if the field contains two structures, this causes an error or confusion.

As for the fourth, JustCode recommends initializing any declared field, but the “ptRotateCopy” is an object, objects in C# can be declared and not necessary to be initialized; because there is a default constructor to initialize the data members in the classes. Thus this recommendation is not correct or accurate.

The final recommendation in Table 4.7, JustCode recommends renaming all the namespaces in the files to the solution or folder name which contains these files.

Table 4.7 shows some code-lines that could not be recognized or recommended by StyleCop after applying StyleCop on “MarsMission” project.

Table 4.7: Example code MarsMission and StyleCop recommendation

Example Code MarsMission	StyleCop Recommendation
<code>public const string conMasterLimbName</code>	No recommendation for the field, but the StyleCop say the field must have a private.
<code>public const Int32 ULW_COLORKEY = 0x00000001;</code>	No recommendation for the field, but the StyleCop say the field must have a private.
<code>public class classReport</code>	StyleCop cannot discover more than one class
<code>class classSetNumImagesPerQuarterRotation</code>	No recommendation for the class, but the StyleCop say the element must have an access modifier.

As mentioned above, StyleCop recommends converting all non-private fields to private access modifier, however, the first row in the Table 4.7 shows an example; field has public access modifier, and there is no recommendation for the field to be private.

As mentioned above, if there is more than one class in the same file, and this class is not partial, then StyleCop will recommend that there is more than one class in the same file, but it did not recommend it in this example.

And finally, there must be a recommendation when the access modifier for a C# code element such as a class has not been explicitly defined. However, the example in the Table 4.7, there is no recommendation for the class.

It should be mentioned that are discovered in two ways, the first one is the process of application of what was recommended by JustCode by using the same tool. The second way is the process of comparing the result which is obtained from applying the StyleCop tool on the project with the result which is acquired from applying our own developed tool in the project.

4.3 Inconsistency Issue

This term refers to the result which is acquired from applying a specific SCA tool on a project, must not be changed from time to time; in other words, if a specific source code was applied many times on an SCA tool. Number of warnings should be always the same.

On the other hand, the number of warnings -which is acquired from applying a specific SCA tool on a project – must be equal the total number of warnings – which is acquired from applying this tool on the files that consist this project. Another Consistency issue is that, some SCA tools allow to alter the recommendations automatically. However, these recommendations are incorrect in the tool warnings.

The first and second issues that were mentioned previously do not exist in StyleCop tool. Figures 4.2 and 4.3 below show the differences between the results which are obtained from applying a StyleCop tool on the same project many times. Figures 4.2 shows the results which are obtained the first time, as evident in the Figure, the number of generated warnings were 1374.

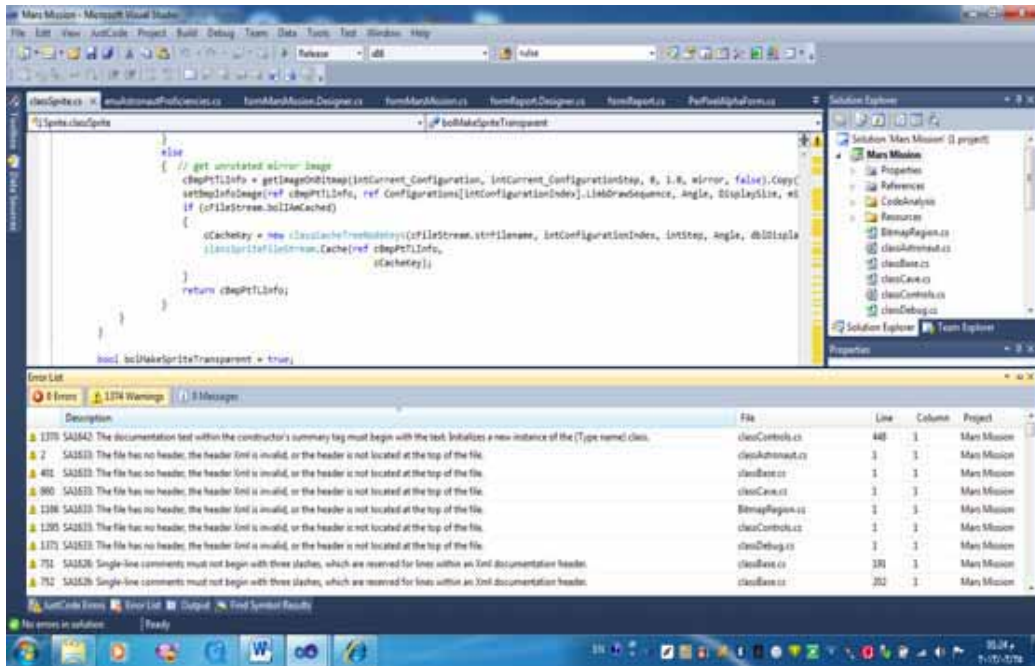


Figure 4.2: Generated warnings from StyleCop tool

After applying this source project on StyleCop many times, the number of warnings becomes 1002. Figure 4.3 shows a reason that may lead to the difference between the numbers of the warnings; StyleCop repeated the same warnings many times, such as the example below:

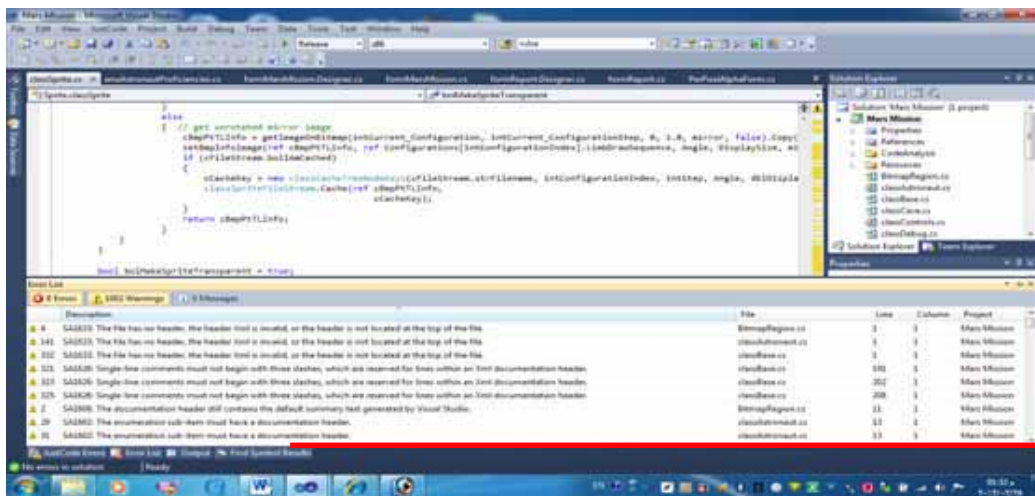
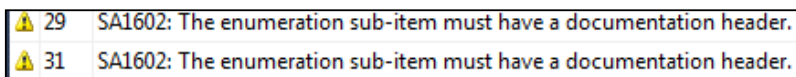


Figure 4.3: Repeated warnings from StyleCop tool

The second issue of StyleCop is the number of warnings -which are acquired from applying them on the project – is not equal to the total number of warnings – which is acquired from applying it on the files that consist this project.

The third inconsistency issue is related to JustCode, JustCode can alter the original code, to apply its recommendations, which is done by pressing on recommendation, after it is pressed, the user can then see the source code, and two options will be given. Making adjustments and changes will be then allowed. However, the problem is that options will be given to make alteration on a code line. This alteration is given in JustCode recommendations incorrect. In other words, make alteration on a code line is allowed, but this line was listed in the JustCode warnings list but incorrect. Figure 4.4 shows the inconsistency between the recommendations and alterations.

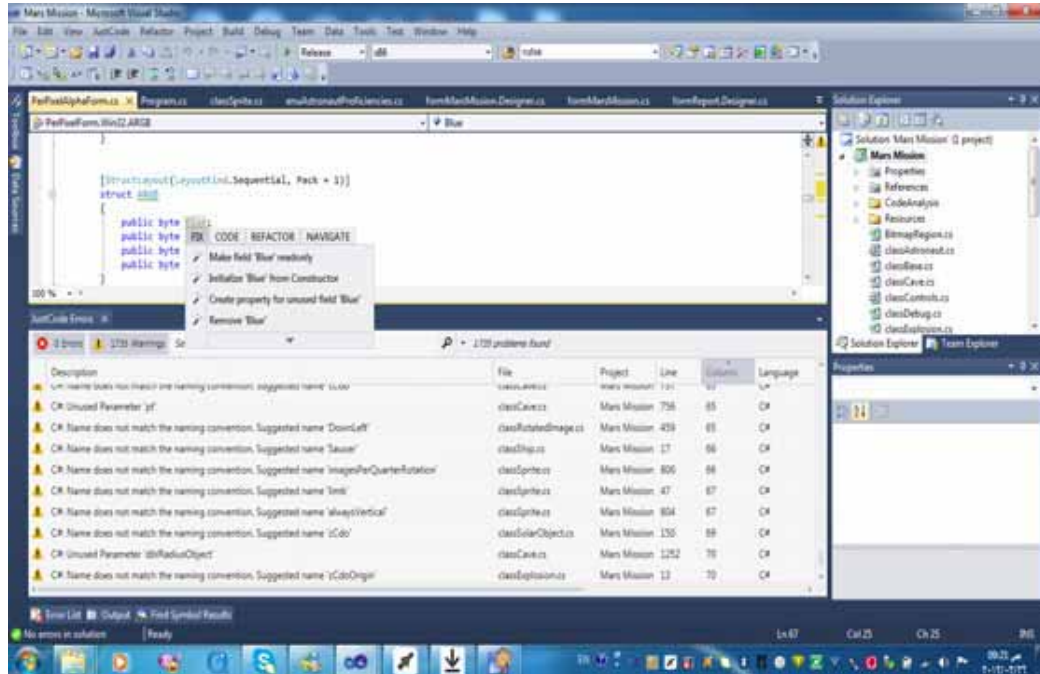


Figure 4.4: Inconsistency between the JustCode recommendations and alterations

4.4 Tool Implementation

This section will describe the process following to develop our own SCA tool. This will be presented in three major activities as typical software developed process: requirements and implementation.

4.4.1 Requirements

The features were achieved by our SCA tool:

- 1- Detecting warnings and give the recommendation.
- 2- Applying the automatic update and changes on the code.

As for the first feature in details, this tool is used to discover the warnings which are classified or related to four categories; maintainability, naming, ordering, layout.

As for the process of the application of the recommendations on the code; in other words, if there is any recommendation in some code line, after pressing on alteration button then the alteration will be applied on the code specifically for the alteration related to two categories; maintainability, naming.

4.4.2 Implementation

- Button maintainability:

Pseudo code:

- WHILE not End Of File (EOF)
 - IF the file extension is .cs
 - Read a code line
 - Split the line to list of strings
 - Check the line is not contain a comment
 - Extract the next word from the line
 - Match the list of string with the rule

- Show the updated code in text2
 - ENDF
- ENDWHILE

- Naming Button:

Pseudo code:

- WHILE not End Of File (EOF)
 - Read a code line
 - Check for “{” and “}” in the line
 - Split the line to list of strings
 - Match the list of string with the rule
 - Match the elements of the generated list with C# keywords, such as, class, namespace, and others
 - Check of condition to match rules
 - Print the warnings based on the checked condition and the condition in text1
 - Update the code to Match the detected warnings, and put the updated code in text2
- ENDWHILE

- Ordering Button:

Pseudo code:

- WHILE not End Of File (EOF)
 - Read a code line
 - Check for “{” and “}” in the line
 - Split the line to list of strings
 - Match the list of string with the rule
 - Match the elements of the generated list with C# keywords, such as , class , namespace , and others
 - Check of condition to match rules
 - Print the warnings based on the checked condition and the condition in text1

- ENDWHILE

- Layout Button:

Pseudo code:

- WHILE not End Of File (EOF)

- Read a code line
- Check for “{” and “}” in the line
- Split the line to list of strings
- Match the list of string with the rule
- Match the elements of the generated list with C# keywords, such as class, namespace, and others
- Check of condition to match rules
- Print the warnings based on the checked condition and the condition in text1

- ENDWHILE

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER FIVE

EXPERIMENTAL RESULTS AND ANALYSIS

In this chapter and based on the developed SCA tool and the previously described an approach, an experiment will be conducted to evaluate the developed SCA tool.

For each one of the projects of Software Under Test (SUT), the warnings based on the SCA tool have been extracted. Those warnings are according to the classes or categories mentioned, in section 4.4. Forty project codes were utilized for this experimental study. The sizes of tested projects vary based on the number of classes or files in each project or software.

5.1 SCA Tool Warnings Extraction

This section is divided into three subsections. In the first subsection the maintainability warnings will be extracted. In the second subsection the naming warnings will be extracted. Finally in the third subsection the ordering warnings will be extracted.

Table 5.1 shows all rules of warnings and examples, which related to the four kinds mentioned earlier.

Table 5.1: All rules of warnings and description

Category	Name of rule	Description
Maintainability Rules	Access Modifier Must Be Declared	The class must have an access modifier.
	Fields Must Be Private	Fields must be declared with private access. Use properties to expose fields.
	File May Only Contain A Single Class	A C# document may only contain a single class at the root level unless all of the classes are partial and are of the same type.
	File May Only Contain A Single Namespace	A C# code file contains more than one namespace.
Naming Rules	Field Names Must Not Contain Underscore	Field names must not contain underscore <code>m_cfgFilename</code> .
	Field Names Must Not Begin With Underscore	Field name must not begin with an underscore: <code>_cancel</code> .
	Accessible Fields Must Begin With Upper Case Letter	Public, internal, and const field names must start with an upper-case letter: <code>intStandardCaveCellDepth</code> .
	Interface Names Must Begin With I	The name of interface does not begin with the capital letter I <code>ISearchableBrowser</code> .
	Element Must Begin With Upper Case Letter	Method names begin with an upper-case letter: <code>convertButton_Click</code> .
	Field Names Must Begin With Lower Case Letter	The name of a field or variable in C# does not begin with a lower-case letter.

Ordering Rules	Using Directives Must Be Placed Within Namespace	Using directives System must be placed within a namespace.
	Constants Must Appear Before Fields	All constant and readonly private fields must be placed before all non-constants, non-readonly private fields.
	Protected Must Come Before Internal	The access modifier internal must come after the protected keyword in the element declaration.
Layout Rules	Curly Brackets For Multi Line Statements Must Not Share Line	If a statement spans multiple lines, the closing curly bracket must be placed on its own line.
	Statement Must Not Be On A Single Line	A C# statement containing opening and closing curly brackets is written completely on a single line.

5.1.1 Warnings Extraction

The maintainability warnings were extracted using our own SCA tool. Several warning types were extracted. Extracted maintainability warnings are divided into many kinds. They include the following types as examples:

- 1- The elements of C# code must have an access modifier.
- 2- The field must be declared with private access modifier.
- 3- The file must contain only one class.
- 4- The file must contain only one namespace.

As for the first warning in the list above “The elements of C# code must have an access modifier”, the elements were extracted. Those are: class, interface, enum, struct, constructor, field, method, and property.

Table 5.2 shows some examples of maintainability warnings, which related to the four kinds mentioned earlier.

Table 5.2: Maintainability warnings extraction examples

Maintainability Warning	Project	File	Line
The class must have an access modifier.	Clipz	Program.cs	9
The method must have an access modifier.	Clipz	Program.cs	15
The property must have an access modifier.	Calculator	MainWindow.xaml.cs	29
The struct must have an access modifier.	Calculator	GraphForm.xaml.cs	271
The Enumeration must have an access modifier	Mars Mission	classAstronaut.cs	13
The field must have an access modifier	Termie	CommPort.cs	29
A C# document may only contain a single class at the root level unless all of the classes are partial and are of the same type.	Mars Mission	classSprite.cs	1690
Fields must be declared with private access. Use properties to expose fields.	Mars Mission	classControls.cs	399

After applying the SCA warning modification on our own developed tool on the project that contain 9 files, it is found that the number of maintainability warnings are total of 315 on all 9 files.

As for the first warning in the Table above, “The class must have an access modifier” warning was found in the Clipz project, in “Program.cs” file at line 9. The line code is:

```
static class Program
```

Notice that, this line code needs an access modifier as will be explained later.

As for the second warning in the Table above, “The method must have an access modifier.” warning was found in the Clipz project, in “Program.cs” file at line 15. The line code is:

```
static void main()
```

As for the third warning in the Table above, “The property must have an access modifier.” warning was found in the Calculator project, in “MainWindow.xaml.cs” file at line 29. The line code is:

```
string CFgfiename
```

As for the fourth warning in the Table above, “The struct must have an access modifier.” warning was found in the Calculator project, in “GraphForm.xaml.cs” file at line 271. The line code is:

```
struct Sample
```

As for the fifth warning in the Table above, “The Enumeration must have an access modifier” warning was found in the Mars Mission project, in “classAstronaut.cs” file at line 13. The line code is:

```
enum enuAstronautProficiencies
```


As an example for the sixth warning in the Table above, “The field must have an access modifier” warning was found in the Termie project, in “CommPort.cs” file at line 29. The line code is:

```
SerialPort _serialPort;
```

As an example for the seventh warning in the Table above, “A C# document may only contain a single class at the root level unless all of the classes are partial and are of the same type.” warning was found in the Mars Mission project, in “classSprite.cs” file at line 1690.

As an example for the final warning in the Table above, “Fields must be declared with private access. Use properties to expose fields.” warning was found in the Mars Mission project, in “classControls.cs” file at line 399. The line code is:

```
public string _strText;
```

5.1.2 Naming Warnings Extraction

In order to find the naming warnings, we have several rules, which validate naming warnings, such as:

- 1- The name of the following components must always begin with an uppercase letter: namespace, class, method, enum, struct, delegate, property, interface, private, public, internal, and const.
- 2- The name of field must not contain an underscore.
- 3- The name of field must not start with an underscore

The naming warnings were extracted using our own SCA tool. We found a large group of warnings. Table 5.3 shows examples of those warnings.

Table 5.3: Naming warnings extraction examples

Naming Warning	Project	File	Line
Field names must not contain underscore m_cfgFilename.	Calculator	MainWindow.xaml.cs	27
Public, internal, and const field names must start with an upper-case letter: intStandardCaveCellDepth.	Mars Mission	classCave.cs	12
Field name must not begin with an underscore: _cancel.	Design	Editor.cs	15
Method names begin with an upper-case letter: convertButton_Click.	Code Colorizer	Page.xaml.cs	29
Class names must begin with upper case letter: dbImageBox.	MyControlSamples	dbImageBox.cs	68
The name of interface does not begin with the capital letter I SearchableBrowser.	Design	SearchDialog.cs	51
Enum names must begin with an upper case letter: enuShipModels.	Mars Mission	classShip.cs	17

As shown in the Table 5.3, if the field name has a private access modifier, then it must begin with a lowercase letter and not contain an underscore, or start with an underscore, such as:

```
string m_cfgFilename = string.Empty;
```

```
private bool _cancel=false;
```

As for the warning “The name of elements must always begin with an uppercase letter.” the elements are: method, class, enum, public, internal, and const field.

Code Examples:

```
void convertButton_click()
```

```
public class dbImageBox
```

```
public enum enushipcondition
```

```
public static int intstandardCarecellDepth;
```

As for the warning “The name of interface does not begin with the capital letter I.” this warning recommends that, the name of interface should begin with a capital letter I.

5.1.3 Ordering and Layout Warnings Extraction

The ordering and layout warnings were extracted using our own SCA tool. The tool extracted several types of this warning. It is found that, ordering and layout warnings are divided to several types such as:

- 1- All constants and read-only private fields must be placed before all non-constants, non-read-only private fields.
- 2- The access modifier keyword must come before the static keyword in the element declaration.
- 3- The access modifier; internal must come after the protected keyword in the element declaration.
- 4- The using directive must be placed in a namespace
- 5- The curly bracket must be placed on its own line, if a statement spans multiple lines.
- 6- If a statement contains opening and closing bracket in one line, statement is written completely on a single line.

Table 5.4 below shows some examples of ordering and layout warnings.

Table 5.4: Ordering and Layout warnings extraction examples

Ordering and Layout Warning	Project	File	Line
Using directives System must be placed within a namespace.	Design	TextInsertForm.cs	1
All constant and readonly private fields must be placed before all non-constants, non-readonly private fields.	cwTab	DoubleBuffer.cs	264
The access modifier internal must come after the protected keyword in the element declaration.	FishTank_src	FishAnimation.cs	9
If a statement spans multiple lines, the closing curly bracket must be placed on its own line.	Mars Mission	BitmapRegion.cs	14
A C# statement containing opening and closing curly brackets is written completely on a single line.	Mars Mission	classAstronaut.cs	295

As for the first warning in the Table 5.4, “The using directive “System” must be placed within a namespace” warning was found in the Design project, in “TextInsertForm.cs” file at line 1. The line code is:

```
Using system;
```

Using directive was not placed within a namespace, so this warning is appeared.

As for the second warning in the table above, “All constants and readonly private fields must be placed before all non-constants, non-readonly private fields.” warning was found in the cwTab project, in “DoubleBuffer.cs” file at line 264. The line code is:

```
public static readonly int
```

```
public const int
```

As mentioned above, all constants and read-only private fields must be placed before all non-constants, non-read-only private fields.

Another example:

```
private int x;  
static private int a;
```

As for the first warning in the table above, “The access modifier internal must come after the protected keyword in the element declaration.” warning was found in the FishTank_src project, in “FishAnimation.cs” file at line 9. The line code is:

```
internal protected int x;
```

The keyword protected should be preceding the keyword internal.

As for the last two warnings in the table above, they are related to layout warnings. They are recommending that the opening and closing curly brackets must each be placed on their own line.

5.2 The Automatic Modification of Proposed Warnings on Tested Code

From previous studies, analysis, and comparisons which we were carried out and applied on some SCA tools, such as JustCode and StyleCop, we noticed some differences between these tools, which were previously mentioned.

It is noticed that, there is a key difference between JustCode and StyleCop; the JustCode allows the programmer to modify or alter the original code to match the recommendations, using “fix” option. On the other hand, StyleCop recommendations are more comprehensive than JustCode recommendations.

Hence, we thought develop an option in our tool that can compromise between the StyleCop and JustCode properties. These tool recommendations are comprehensive as StyleCop tool. It should also allow the programmer to modify or alter the original code to match the recommendations as JustCode tool. It is worth mentioning that, the number warnings in our developed tool is less than StyleCop warnings as we did not

include evaluating all types of warnings. However, it allows modifying the original code as an option similar to JustCode.

In this section, the process of the automatic modification on the code will be shown, on maintainability and naming warnings.

5.2.1 Maintainability Recommendations Automatic Modification

According to Table 5.1; the process of the automatic modification on the code will be done according to the recommendations that are related to the specific code line and it follows to the maintainability rules.

Figure 5.1 shows a sample of the process of code modification.

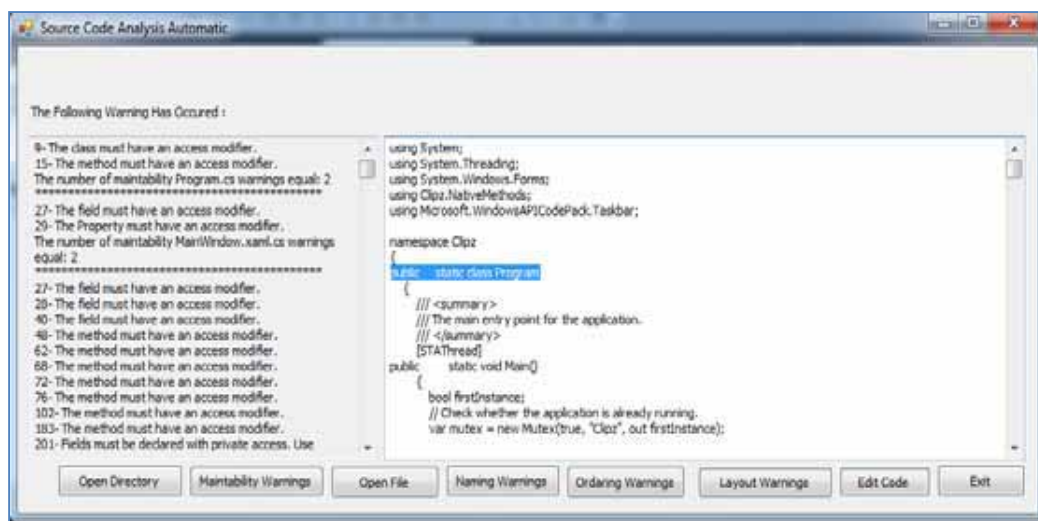


Figure 5.1: The process of automatic modification on the class element

It can be noticed from the Figure 5.1 above; that after the warning was detected by the developed tool. The tool recommends that the class must have an access modifier (public), the tool modified the code at this line, and showed the modified code line in the other box, “public” keyword was added, as follow:

`public static class Program`

Figure 5.2 below shows how the tool detects the warning the “method must have an access modifier” and update the code.

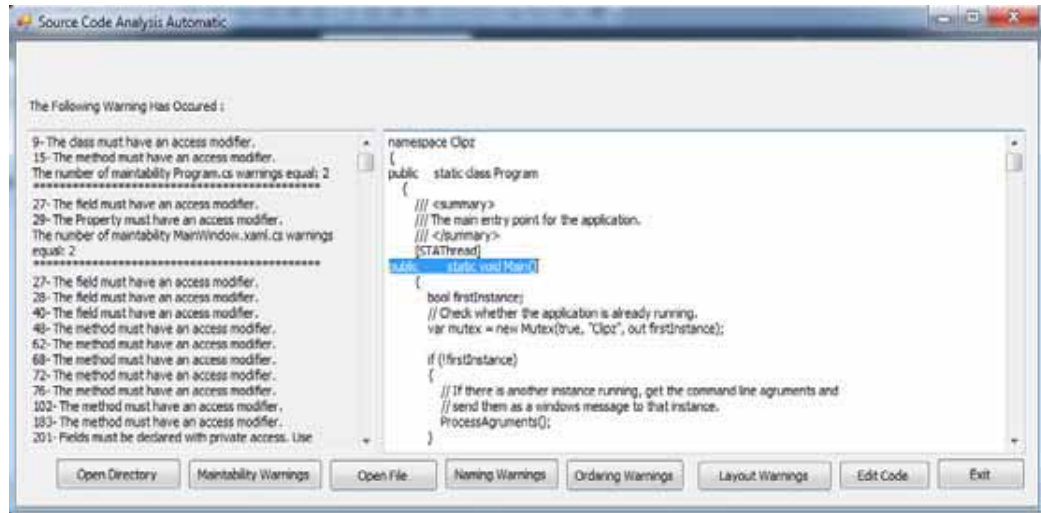


Figure 5.2: The process of automatic modification on the method element

As shown in the Figure 5.2, the tool recommends adding an access modifier at line 15, and then allows modifying the code at this line, so the method then has a comprehensive declaration:

```
public static void Main()
```

The Figure 5.3 shows the process on the code in the “MainWindow.xaml.cs” File; at line 29 which indicates that the property must have an access modifier.

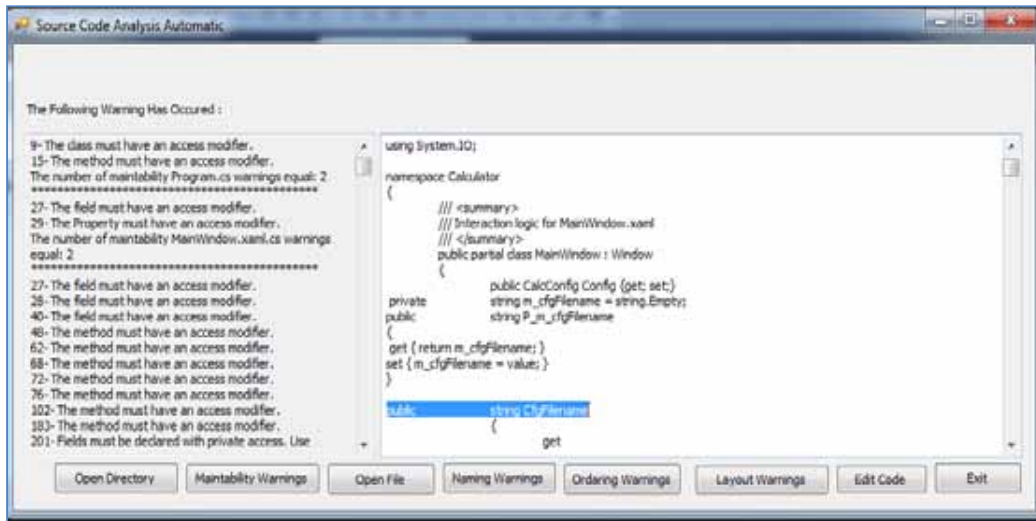


Figure 5.3: The process of automatic modification on the property element

As can be seen in in the Figure 5.3, the tool modified the code at line 29; “public” keyword was added.

Figure 5.4 below shows that the tool recommends that “The struct must have an access modifier”, the Figure 5.4 shows the modification on the code.

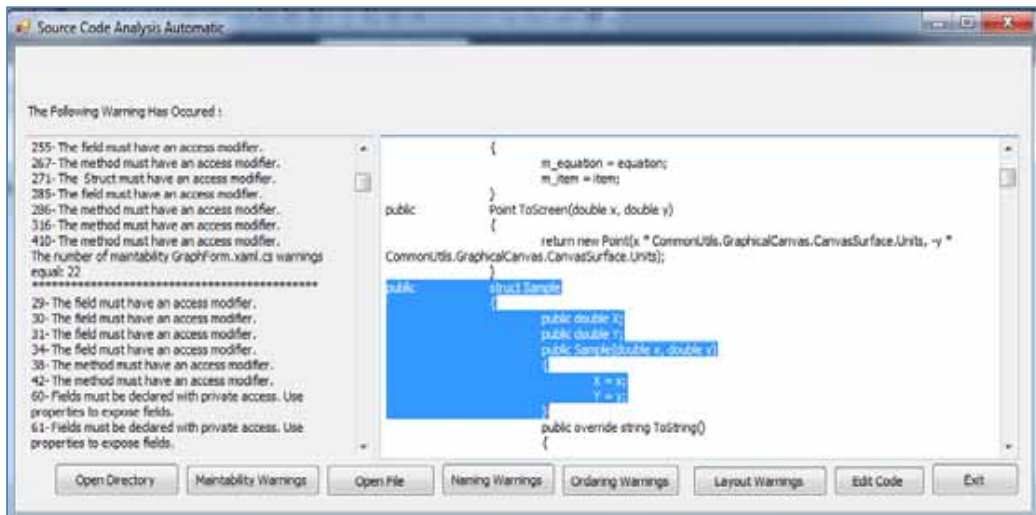


Figure 5.4: The process of automatic modification on the struct element

Figure 5.4 shows that the tool modified the code and “public” keyword was added, so the line code becomes:

```
public struct Sample
```

As shown in Figure 5.5 at line 13 in “classAstronaut.cs” File, there is a recommendation demonstrates that “The enumeration must have an access modifier”. Though, by looking at the box in Figure 5.5 which includes the shadowed code, the tool modified the code by adding “public” keyword, for example:

```
public enum enuAstronautProficiencies
```

As shown in the Figure 5.5 below the code was modified by adding access modifier to enum.

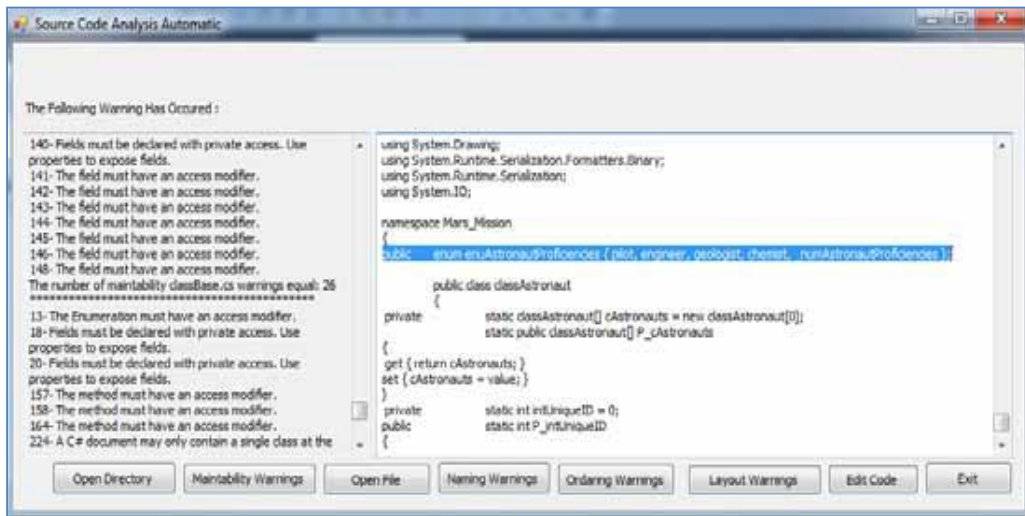


Figure 5.5: The process of automatic modification on the enum element

As for the warning “The field must have an access modifier”. The tool recommends adding an access modifier “private”.

As shown in the Figure 5.6 below, the tool adds a private access modifier, and then adds auto-implemented properties (private class accessed via get and set properties).

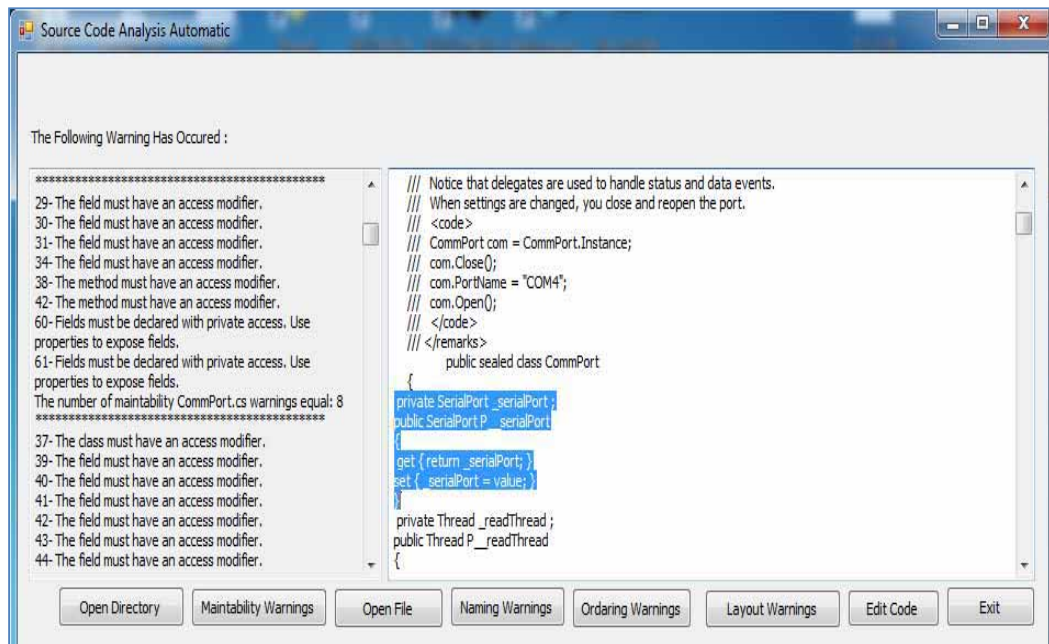


Figure 5.6: The process of automatic modification on the private field element

As shown in Figure 5.6 above, the tool adds a private access modifier, and then it declares setter and getter methods.

We described the warning “the field must have a private access modifier” in other words, if the access modifier is not private (i.e. public, protected, or internal), then it should be converted to private, then the property access modifier should be the same as after the modification.

5.2.2 Naming Recommendations Automatic Modification

This section will rely on the Table 5.2 in section 5.1 in the process of automatic code modification that is implemented in the tool.

Figure 5.7 shows how the tool modifies the code which is the result after implementing recommendations which states that: “public, internal, and const field names must start with an upper-case letter”.

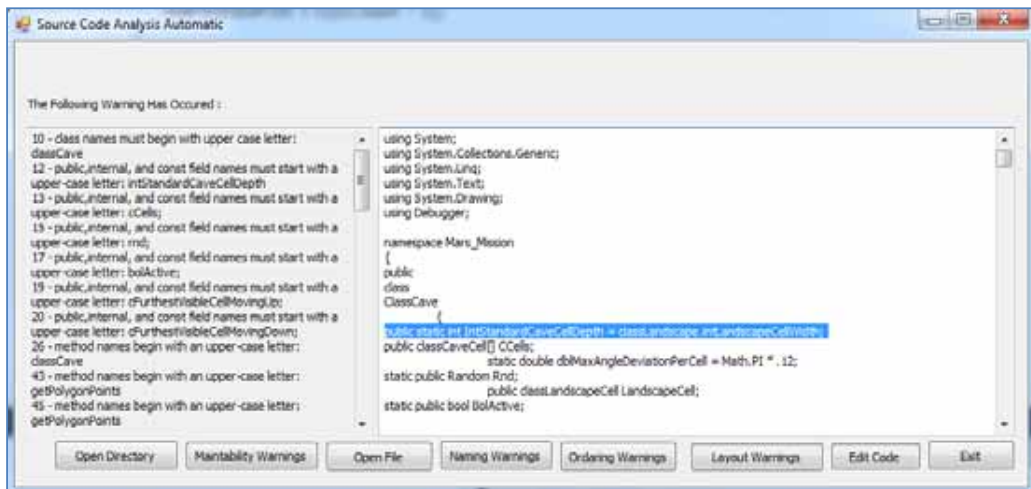


Figure 5.7: Modification on the non-private field element in naming warnings

As seen in Figure 5.7, the “intStandardCaveCellDepth” field becomes “IntStandardCaveCellDepth”, depending on the previous recommendation. Also, class, method, and enum names must begin with an upper case letter.

As for the Figure 5.8, it asserts that the name of interface must begin with the capital letter “I”.

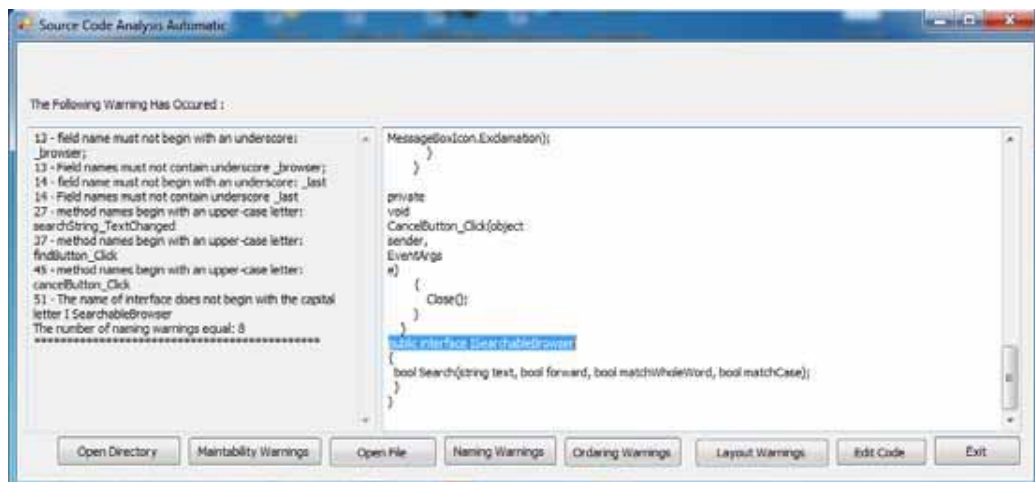


Figure 5.8: Modification on the interface element in naming warnings

As noticed from the figure above, the “SearchableBrowser” interface becomes “ISearchableBrowser”, depending on the previous recommendation, as follows:
public interface ISearchableBrowser

Applying the recommendation which results from the warning “field name must not begin with an underscore”, the tool modifies the code. Thus, underscore -that starts a field name- will be removed. As shown in the Figure 5.9 below:

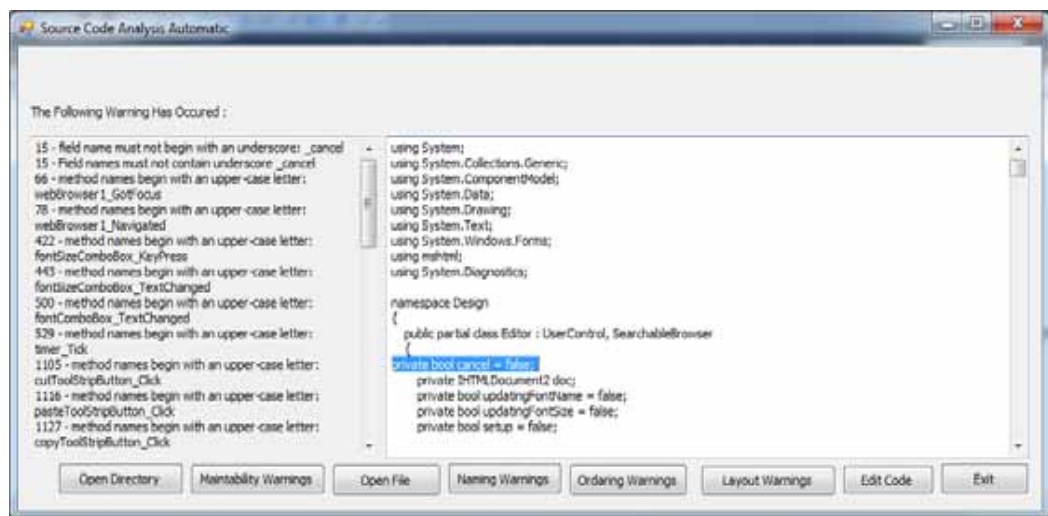


Figure 5.9: Modification on the field element that starts with underscore in naming warnings

5.3 Comparing Between Our Own Tool and Other SCA Tools

This section will focus on the differences and similarities between our own tool and other SCA tools, specifically JustCode and StyleCop tools.

Table 5.5 shows the comparing between our own tool and other SCA tools such as JustCode and StyleCop tools.

Table 5.5: Comparing between our own tool and other SCA tools

Category	StyleCop	JustCode	Our Own tool
Maintainability rules	Access Modifier Must Be Declared	—————	Access Modifier Must Be Declared
	Fields Must Be Private	—————	Fields Must Be Private
	File May Only Contain A Single Class	—————	File May Only Contain A Single Class
	File May Only Contain A Single Namespace	—————	File May Only Contain A Single Namespace
Naming Rules	Field Names Must Not Contain Underscore	Name does not match the naming convention	Field Names Must Not Contain Underscore
	Field Names Must Begin With Lower Case Letter	Name does not match the naming convention	Field Names Must Begin With Lower Case Letter
	Field Names Must Not Begin With Underscore	Name does not match the naming convention	Field Names Must Not Begin With Underscore
	Accessible Fields Must Begin With Upper Case Letter	Name does not match the naming convention	Accessible Fields Must Begin With Upper Case Letter
	Interface Names Must Begin With I	Name does not match the naming convention	Interface Names Must Begin With I
	Element Must Begin With Upper Case Letter	Name does not match the naming convention	Element Must Begin With Upper Case Letter
Ordering Rules	Using Directives Must Be Placed Within Namespace	—————	Using Directives Must Be Placed Within Namespace
	Constants Must Appear Before Fields	—————	Constants Must Appear Before Fields
	Protected Must Come Before Internal	—————	Protected Must Come Before Internal

Layout Rules	Curly Brackets For Multi Line Statements Must Not Share Line	—————	Curly Brackets For Multi Line Statements Must Not Share Line
	Statement Must Not Be On A Single Line	—————	Statement Must Not Be On A Single Line
Automatic Code	Change on code manual	Change on code automatic	Change on code automatic
Number of Warnings	Changeable	Fixed	Fixed

It is noticed that, the number of JustCode recommendations is few. However, JustCode allows the programmer to modify or alter the original code to match the recommendations. On the other hand, the number of StyleCop recommendations is more than JustCode recommendations, while it does not have the option of automatic updates.

As mentioned in section 3.3, there are StyleCop recommendations -such as maintainability that do not exist in JustCode. Hence, we thought to design an automatic SCA tool, that modifies the code based on specific rules.

In this section, a comparison will be done between the results that were obtained from our own tool and StyleCop tool, in terms of accuracy in finding warnings, and compare them in terms of inconsistency between the results, or in terms of the ability to modify the code as JustCode tool.

Firstly, we are going to discuss the differences and similarities in the process of giving the warnings. As for the similarities, as noticed from the Figures 5.10 and 5.11, our own tool and StyleCop results in the same warning at the same line for the same tested file or class.

Description	File	Line	Column	Project
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	27	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	28	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	40	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	233	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	254	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	255	1	Calculator
SA1400: The field must have an access modifier.	GraphForm.xaml.cs	285	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	48	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	62	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	68	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	72	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	76	1	Calculator
SA1400: The method must have an access modifier.	GraphForm.xaml.cs	94	1	Calculator

Figure 5.10: StyleCop tool warnings results

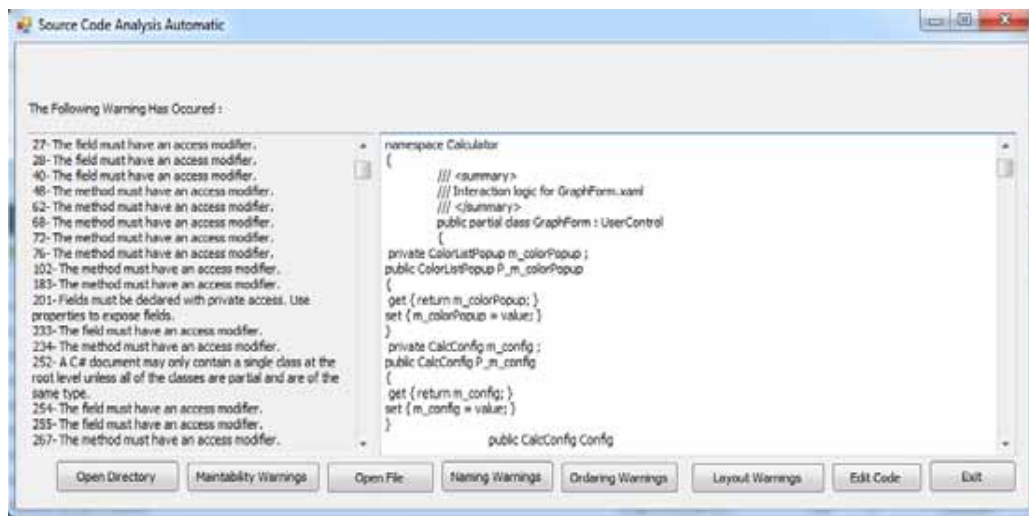


Figure 5.11: Our tool warnings results

As noticed in the Figure 5.10, StyleCop tool detected a group of warnings related to maintainability, they are:

- The field must have an access modifier.
- The method must have an access modifier.

In the file “GraphForm.xaml.cs” at lines 27, 28, 40, 233, etc. as shown in Figure 5.11, our own tool detects the same warnings at the same lines for the same tested code.

We will then evaluate the differences between our own tool and Style Cop, for the process of locating the code line which contains the modification. Figure 5.12 shows the results that are obtained from applying StyleCop on the MarsMission project. Figure 5.13 shows the results that are obtained from applying our own tool on the MarsMission project.

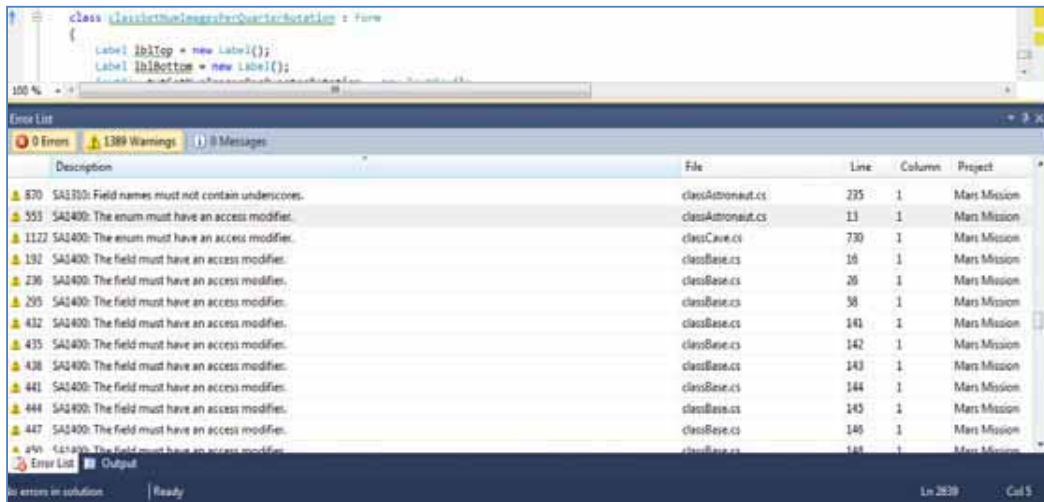


Figure 5.12: Apply StyleCop on MarsMission project

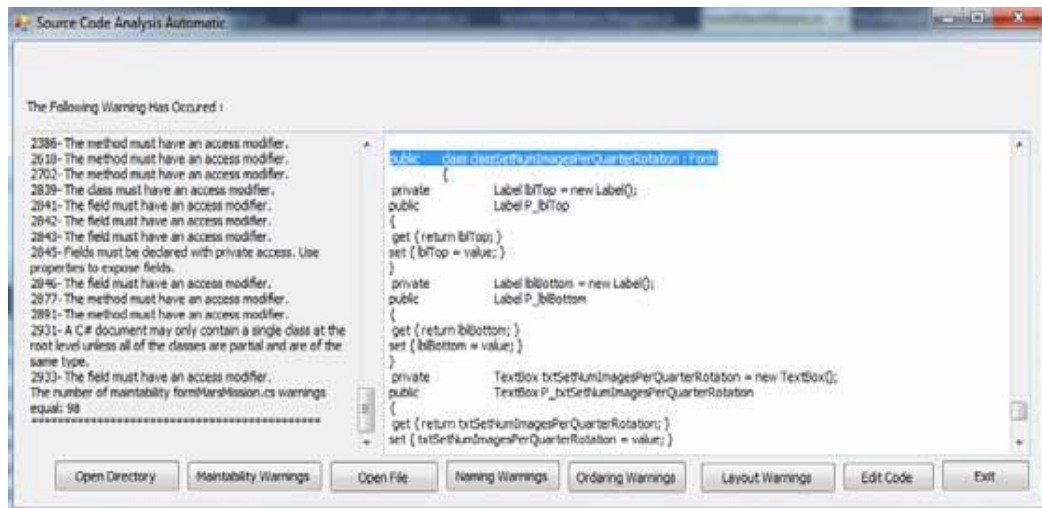


Figure 5.13: Apply our own tool on MarsMission project

As noticed from Figure 5.12, there is a class declaration called classSetNumImagesPerQuarterRotation at line 2839. However, the StyleCop did not

recommend that “The class must have an access modifier”. On the other hand, as shown in the Figure 5.13, our own tool recommend that “The class must have an access modifier” in the left box, and in the right box. It modified the code by adding a public access modifier. Such issue may need to be investigated thoroughly to be generalized.

As for the inconsistency issue, it is found that, the number of warnings may differ from in JustCode when running the tool more than once in the same code. An example of this is shown in Figure 5.12 the number of warnings is 1389, but as noted in Figure 5.14 the number of warnings is 1017 when running the tool another time. We evaluated our tool on several codes through running it several times on each code and results were always consistent.

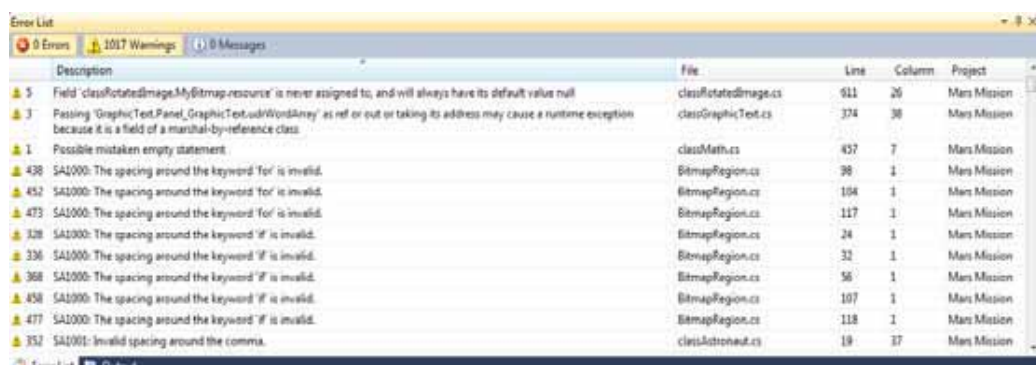


Figure 5.14: Applying StyleCop on MarsMission project in second run

As for the process of inaccurate modification on the code by JustCode tool; in other words, a field has a public access modifier that is converted to read-only by JustCode, this modification assumed to allow public access to the field without allowing it to be changed. However, this may be considered as changing the code improperly.

We used (setters and getters) as an alternative (Figure 5.15) which does not modify the field visibility scope.

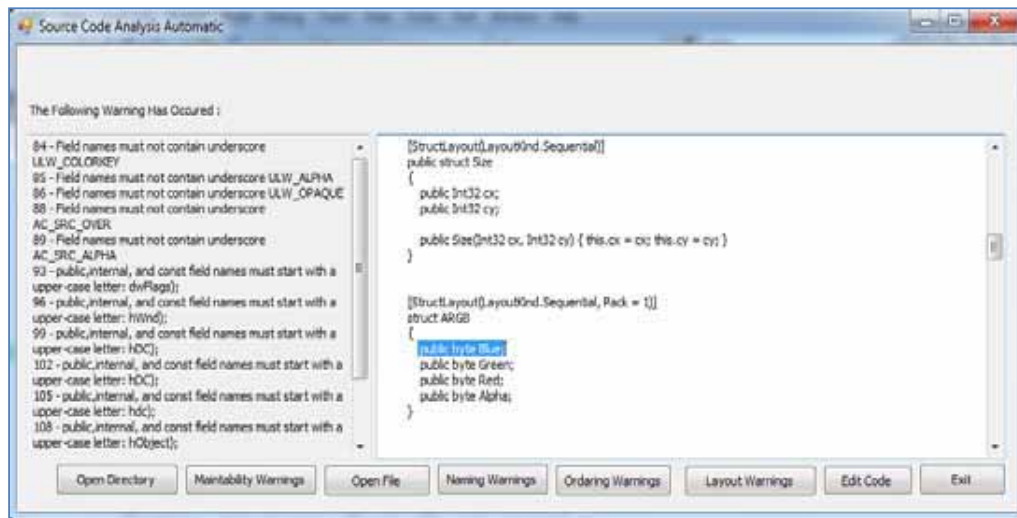


Figure 5.15: Public field issue

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER SIX

HOW TO USE THE DEVELOPED TOOL

In this chapter, we will explain how our own tool works to detect the warnings from tested source codes. Each warning class will be mentioned or explained how to select a file or folder, then how to give a recommendation, and how to modify the code according to the given recommendations.

Figure 6.1 shows the main menu or the graphical user interface of our own tool.

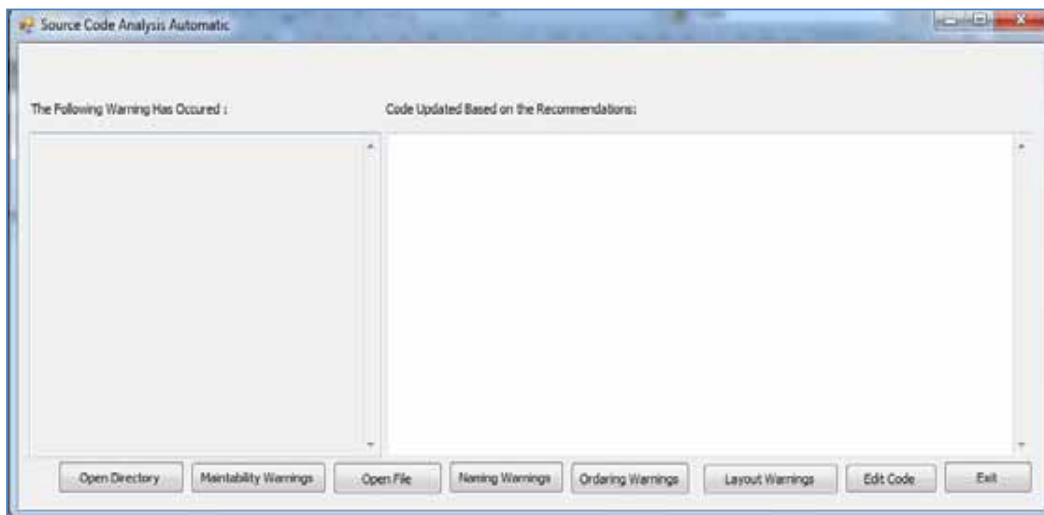


Figure 6.1: Main menu of our SCA tool

The essential components of this screen were discussed or mentioned in section 4.4.

6.1 Maintainability Warnings Extraction

To detect the warnings and give recommendations by pressing on "Open Directory" button, this button allows the user to choose a folder that consists of a group of CS files. Figure 6.2 shows how to select a folder using "Browse For Folder" dialog.

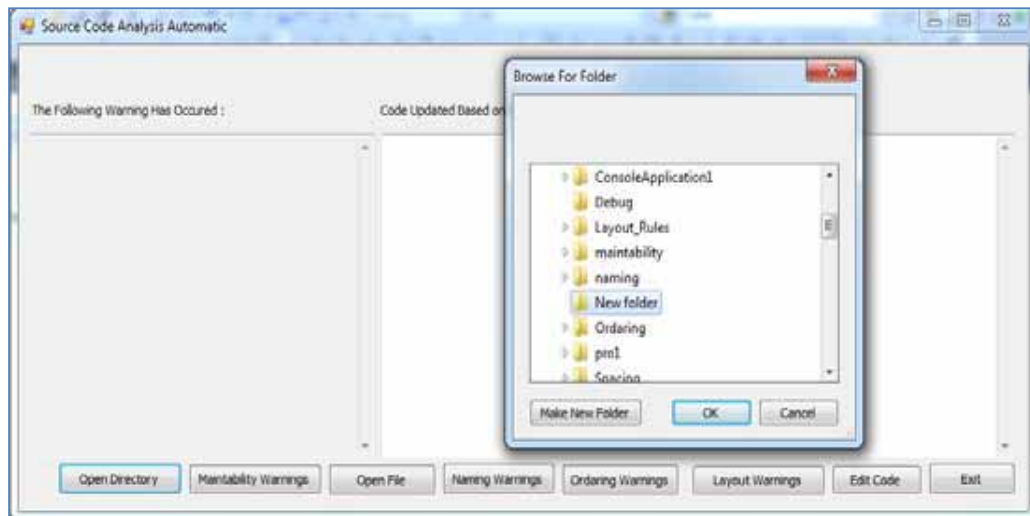


Figure 6.2: Browse for folder dialog

To proceed, user should click on "Maintainability Warnings" button then the results will be shown as in Figure 6.3 below. Figure 6.3 shows the obtained results after clicking on "Maintainability Warnings".

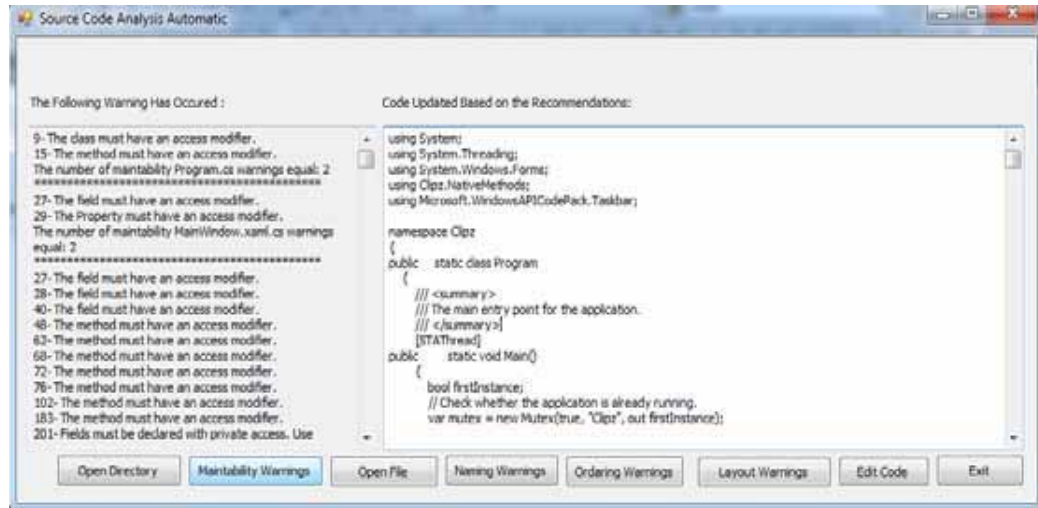


Figure 6.3: Results after clicking on "Maintainability Warnings"

6.2 Naming Warnings Extraction

In this section, we will explain how to apply the naming rules on the source code to generate recommendations.

Figure 6.4 shows the process of choosing a file then applying the rules.

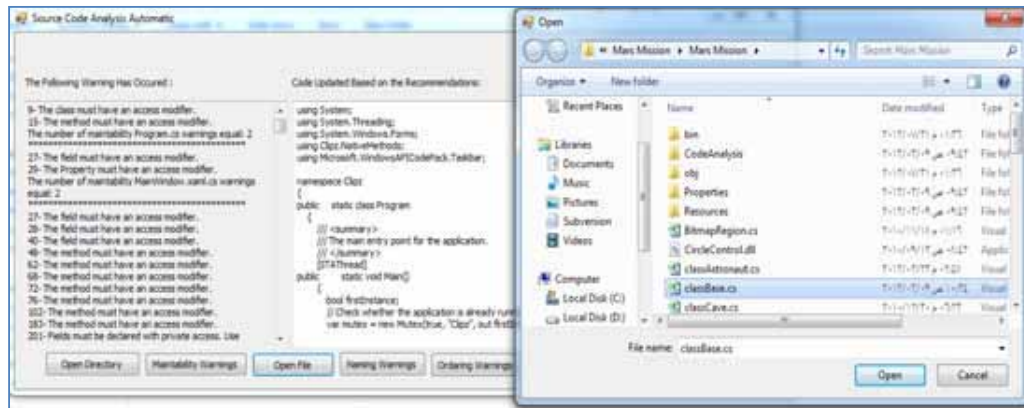


Figure 6.4: The process of choosing a file

Then, press on "Naming Warnings" button, in order to apply the rules on the selected code, as shown in the Figure 6.5 below. Figure 6.5 shows the obtained naming warnings and the modification on the code.

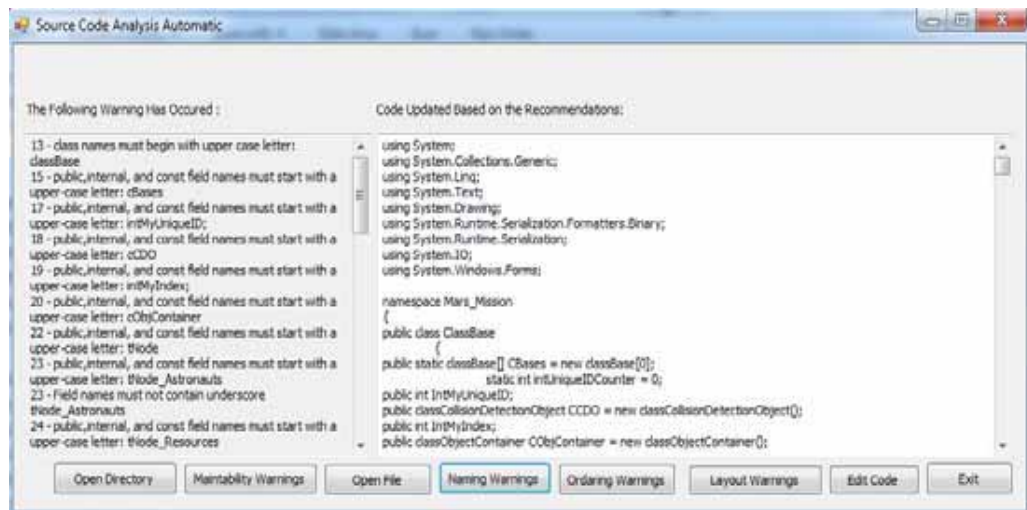


Figure 6.5: The obtained naming warnings and the modification on the source code

6.3 Ordering and Layout Warnings Extraction

According to these types of warnings, in order to apply them on the source code select a file using "Open File" button, as mentioned in "Naming Warnings" applying section , then click on "Ordering Warnings" or "Layout Warnings", then the rules will be applied , as shown in Figures 6.6 and 6.7 below.

Figure 6.6 shows the process of applying "Ordering Rules" on the source code, and the results were shown in the first box, but there is no modification on the code.

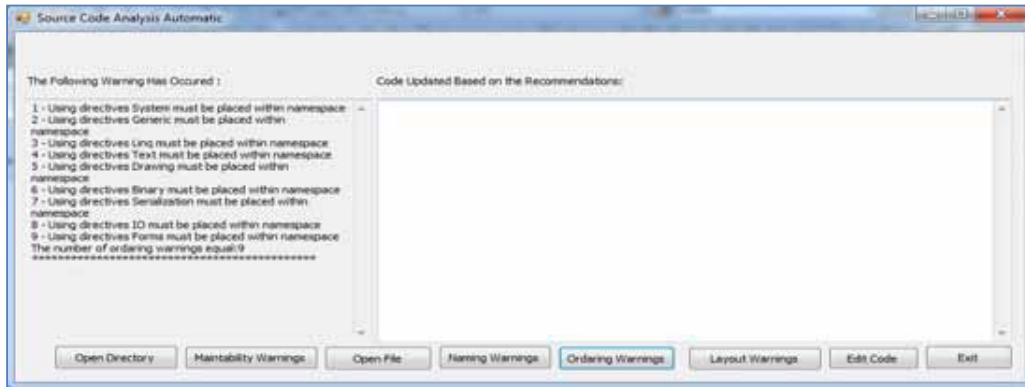


Figure 6.6: Applying "Ordering Rules" on the source code

Figure 6.7 shows the recommendations that were obtained after applying "Layout Rules" on the source code.

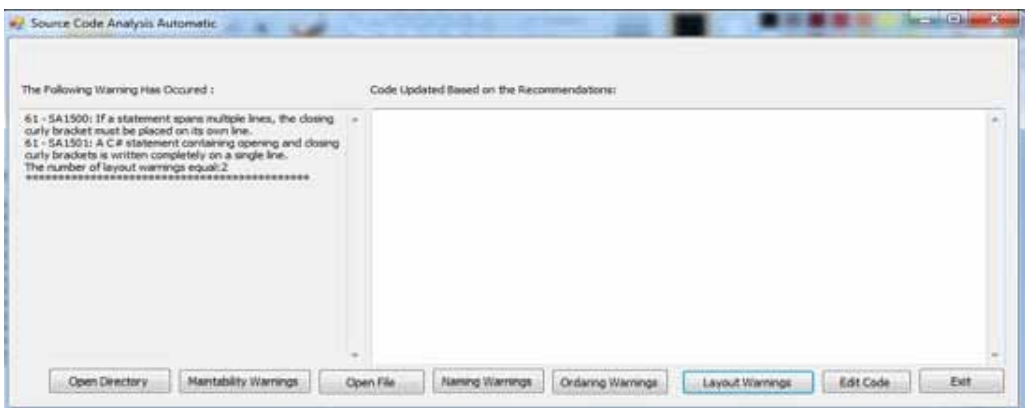


Figure 6.7: Applying "Layout Rules" on the source code

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

CHAPTER SEVEN

CONCLUSIONS AND FUTURE WORK

This chapter consists of two sections the first section will explain the conclusion of the thesis, while the second will show the future work.

7.1 Conclusions

Massive increase in quantity of software integration generates growing demand for programmers and their productivity, on the other hand, hiring additional programmers is expensive and ineffective, especially when the system is in execution time or is indivisible due to the complexity of modern software, and in order to found a more viable solution is a tool support, this led to growing interest on the tool that based on source code analysis.

Our SCA tool recommends some of warnings that directed to programmers to prevent the occurrence of errors.

There are four main SCA tools warnings were studied in this thesis: maintainability warnings, naming warnings, ordering warnings, and layout warnings.

The main goal of this study is to detect the warnings classes, and the process of automatic the modification on the source code based on the recommendations.

7.2 The Limitations and Weaknesses

As in most theses and studies, difficulties were encountered. I encountered many difficulties also the limited time that resulting in inability to detects all the warnings in our tool, because some warning needs a lot of time.

Moreover, the process of modification on the code needs to apply all the warnings, and each warning has a group of rules.

One of the difficulties that were encountered during the warnings detecting, that the warning is given or recommended based on a specific rule, and this rule rely on the C# element definition or declaration, such as there are many way to write a field declaration, so that many rules should be written to field declaration, and declaration nested class.

7.3 Future Work

We plan to extend this work in the future to include the following three areas:

1. Enhancing this work by including all the classes' warnings of source code to get the best prediction of error.
2. Extent the work of develop tool in the process of the modification on the code to include all the detected warnings in the develop tool.
3. As for the process of the modification, assurance that did not lead to the existence of real errors.
4. Adding a new feature to the tool to allow multiple options to the modifying on the code, as JustCode.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

ABSTRACT

Software testing is one of the most comprehensive phases in the software projects where it takes a huge amount of time and resources. Testing however is not only the process that occurs after the implementation executing the program looking for faults to fix them. There are many supplementary testing activities that may occur within the testing stage or outside that lead to the same goal as of testing to improve the developed software and reduce effort required to use it, test it, maintain or update it. One of those supplementary testing activities is source code analysis. Source code analysis focuses largely on warnings, not errors, where such warnings indicate possible violation of naming standards or best practices. Such violation may in future leads to errors and hence should be handed early. Source Code Analysis (SCA) tools such as: MS StyleCop and JustCode have been developed to help developed areas in their code that should be improved or modified to eliminate the display if warnings. Some of those tools are integrated with programming languages environments and compilers. The main objective of this thesis is to propose and develop an SCA tool that can improve some of the limitations in the evaluated SCA tools. In order to achieve our main objective, we first conducted an evaluation or assessment case study looking for limitations and weaknesses in the existing evaluated SCA tools. Based on such initial assessment and comparison, a list of candidate requirements for the new SCA tool is assembled. The developed or assembled tool can perform the following tasks: Detect several categories of warnings, propose solutions to remove those warnings and automatically apply those warnings if the user or the developer wants to do so. The main contribution of this thesis is the development of a new SCA tool that can override some of the limitations of the evaluated SCA tools. The new tool tried to take the good options of both tools and bypass or avoid their limitations. Results showed that, based on the four warning categories that we focused on, our tool showed better results in overcoming some of the inconsistency problems or problems related to the automatic implementation of recommended corrections.

Key Words: source code analysis tools, static code analysis tool, maintainability warnings, software testing, software quality.

المخلص

اختبار البرمجيات هو واحد من أكثر المراحل الشاملة في مشاريع البرمجيات، حيث يستغرق قدرا كبيرا من الوقت والموارد. لكنه ليس فقط العملية التي تحدث بعد تنفيذ البرنامج للبحث عن أخطاء لإصلاحها. هناك العديد من أنشطة الاختبار التكميلية التي قد تحدث داخل أو خارج المرحلة التي تؤدي إلى نفس الهدف من الإختبار لتحسين البرمجيات المطورة وتقليل الجهد المطلوب لاستخدامه، واختباره، والحفاظ عليه أو تحديثه. واحدة من أنشطة الاختبار التكميلي هو تحليل البرامج. ركز تحليل البرامج بشكل كبير على التحذيرات، لا الأخطاء، حيث تشير مثل هذه التحذيرات إلى انتهاك محتمل لمقاييس التسمية أو أفضل الممارسات. مثل هذا الانتهاك قد يؤدي إلى أخطاء، وبالتالي يجب أن يتم السيطرة عليها مبكرا. برامج تحليل البرامج مثل StyleCop و JustCode تطورت للمساعدة في عملية تطوير المساحات في التعليمات البرمجية الخاصة بهم التي ينبغي تحسينها أو تعديلها للقضاء على مكان ظهور التحذيرات. بعض هذه البرامج يتم دمجها مع بيئة لغات البرمجة والمترجم. الهدف الرئيسي لهذه الأطروحة هو اقتراح وتطوير أدوات SCA التي يمكن أن تحسن بعض القيود في أدوات SCA. من أجل تحقيق هدفنا الرئيسي، أجرينا أولا "تقييم أو دراسة حالة تقييم للبحث عن القيود ونقاط الضعف في أدوات SCA. وبناء على هذا التقييم الأولي والمقارنة، تم تجميع قائمة من المتطلبات المرشحة للتنفيذ في الأداة الجديدة التي قمنا ببنائها. أهم الوظائف التي احتوت عليها الأداة التي قمنا بتصميمها وبنائها: تستطيع الأداة الكشف عن عدة فئات من التحذيرات مثل الأدوات التي قيمناها مع بعض التحسينات، واقتراح حلول لإزالة تلك التحذيرات وتطبيق الحلول تلقائيا لتلك التحذيرات إذا كان المستخدم أو المطور يريد أن يفعل ذلك. إذن فإن المساهمة الرئيسية لهذه الأطروحة هي استحداث أداة SCA جديدة والتي لديها قدره على تجاوز بعض المشاكل في الأدوات المقيمه سلفا. استنادا إلى الفئات الأربعة من أنواع التحذيرات التي قمنا بتقييمها ومعالجتها تلقائيا أثبتت الأداة الجديدة التي قمنا ببنائها قدرة أفضل من الأدوات المقيمه لهذا الغرض أو أظهرت دمج بين حسنات الأدوات———— بين المقيمتين.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

TABLE OF CONTENTS

<u>Contents</u>	<u>Page</u>
ACKNOWLEDGEMENT	I
TABLE OF CONTENTS	II
LIST OF FIGURES	V
LIST OF TABLES	VII
LIST OF ABBREVIATIONS	VIII
ABSTRACT	IX
I. INTRODUCTION	1
1.1 Overview	1
1.2 Dealing with Quality Problems	1
1.2.1 Dynamic Testing	2
1.2.2 Metrics	2
1.2.3 Source Code Analysis Tools (static testing)	3
1.3 Sample of Source Code Analysis Tools	3
1.3.1 StyleCop	3
1.3.2 JustCode	4
1.3.3 FxCop	5
1.4 Problem Statement	5
1.5 Research Objective	6
1.6 Research Importance	7
1.7 Thesis Structure	8
II. RELATED WORK	9
2.1 Software Metrics and Class Change Proneness	9
2.2 Testing and Source Code Analysis Tools	14
2.3 Software Quality	17

2.4 Maintenance	20
III. STATIC CODE ANALYSIS TOOLS	23
3.1 Static Code Analysis Tools	23
3.1.1 Warnings	24
3.1.2 Errors	25
3.1.3 Information	25
3.2 What can Static Code Analysis Accomplish?	26
3.3 Analysis and Comparison: Source Code Analysis Tools	27
3.3.1 Analysis: Source Code Analysis Tools	27
3.3.1.1 StyleCop Tool	28
3.3.1.2 JustCode Tool	47
3.3.1.3 FxCop Tool	49
3.3.2 A Comparison between the Tools	51
IV. RESEARCH GOALS AND APPROACHES	54
4.1 Differences in the Results that come from each Tool for the Same Source Code	57
4.2 Weaknesses of the Two Evaluated Tools	62
4.3 Inconsistency Issue	65
4.4 Tool Implementation	68
4.4.1 Requirements	68
4.4.2 Implementation	68
V. EXPERIMENTAL RESULTS AND ANALYSIS	71
5.1 Source Code Analysis Tool Warnings Extraction	71
5.1.1 Maintainability Warnings Extraction	73
5.1.2 Naming Warnings Extraction	76
5.1.3 Ordering and Layout Warnings Extraction	78
5.2 The Automatic Modification of Proposed Warnings on Tested	80

Code	
5.2.1 Maintainability Recommendations Automatic Modification	81
5.2.2 Naming Recommendations Automatic Modification	85
5.3 Comparing Between Our Own Tool and Other SCA Tools	87
VI. HOW TO USE THE DEVELOPED TOOL	94
6.1 Maintainability Warnings Extraction	94
6.2 Naming Warnings Extraction	96
6.3 Ordering and Layout Warnings Extraction	97
VII. CONCLUSIONS AND FUTURE WORK	98
7.1 Conclusions	98
7.2 The Limitations and Weaknesses	99
7.3 Future Work	99
VIII. REFERENCES	100

LIST OF FIGURES

<u>Figures</u>	<u>Title</u>	<u>Page</u>
Figure 4.1:	Methodology phases	55
Figure 4.2:	Generated warnings from StyleCop tool	66
Figure 4.3:	Repeated warnings from StyleCop tool	66
Figure 4.4:	Inconsistency between the JustCode recommendations and alterations	67
Figure 5.1:	The process of automatic modification on the class element	81
Figure 5.2:	The process of automatic modification on the method element	82
Figure 5.3:	The process of automatic modification on the property element	83
Figure 5.4:	The process of automatic modification on the struct element	83
Figure 5.5:	The process of automatic modification on the enum element	84
Figure 5.6:	The process of automatic modification on the private field element	85
Figure 5.7:	Modification on the non-private field element in naming warnings	86
Figure 5.8:	Modification on the interface element in naming warnings	86
Figure 5.9:	Modification on the field element that starts with underscore in naming warnings	87
Figure 5.10:	StyleCop tool warnings results	90
Figure 5.11:	Our tool warnings results	90
Figure 5.12:	Apply StyleCop on MarsMission project	91
Figure 5.13:	Apply our own tool on MarsMission project	91
Figure 5.14:	Applying StyleCop on MarsMission project in second run	92
Figure 5.15:	Public field issue	93

Figure 6.1:	Main menu of our SCA tool	94
Figure 6.2:	Browse for folder dialog	95
Figure 6.3:	Results after clicking on "Maintainability Warnings"	95
Figure 6.4:	The process of choosing a file	96
Figure 6.5:	The obtained naming warnings and the modification on the source code	96
Figure 6.6:	Applying "Ordering Rules" on the source code	97
Figure 6.7:	Applying "Layout Rules" on the source code	97

LIST OF TABLES

<u>Tables</u>	<u>Title</u>	<u>Page</u>
Table 1.1:	A sample SCA warning classification	4
Table 3.1:	Spacing rules and examples	29
Table 3.2:	Readability rules and examples	35
Table 3.3:	Ordering rules	38
Table 3.4:	Naming rules and examples	39
Table 3.5:	Maintainability rules and examples	42
Table 3.6:	Layout rules	44
Table 3.7:	Documentation rules	46
Table 4.1:	An overview of the projects	57
Table 4.2:	Distribution the Chatters warnings on classes of warning	58
Table 4.3:	Distribution the Design warnings on classes of warning	58
Table 4.4:	Result from applying SCA tools on Chatters project	59
Table 4.5:	Result from applying SCA tools on Design project	60
Table 4.6:	Example code MarsMission and JustCode recommendation	63
Table 4.7:	Example code MarsMission and StyleCop recommendation	64
Table 5.1:	All rules of warnings and description	72
Table 5.2:	Maintainability warnings extraction examples	74
Table 5.3:	Naming warnings extraction examples	77
Table 5.4:	Ordering and Layout warnings extraction examples	79
Table 5.5:	Comparing between our own tool and other SCA tools	88

LIST OF ABBREVIATIONS

API	Application Programming Interface
DRE	Defect Removal Efficiency
EENOM	Earliest Evolution Of Number Of Methods
GUI	Graphical User Interface
IUC	Interface Usage Cohesion
JIT	Just In Time
LENOM	Latest Evolution of Number Of Methods
LOC	Lines Of Code
NASA	National Aeronautics and Space Administration
NHCTMC	Non-homogeneous Continuous Time Markov Chain
OO	Object Oriented
OOAD	Object-oriented analysis and design
SUT	Software Under Test
SCA	Source Code Analysis
SCC	fine-grained Source Code Changes
SDLC	Systems Development Life Cycle

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction

by:

Qosai Mwafeq AL-Zoubi

Supervisor:

Professor Dr. Bilal A. H. Abul-Huda

Co-Supervisor:

Dr. Izzat Alsmadi

Computer Information Systems Department

Yarmouk University

May 05, 2013

**Traceability Enhancements on Source Code Analysis
Tools to Improve Software Defects Prediction**

By:

Qosai Mwafeq AL-Zoubi

B.Sc. Computer Information Systems, Yarmouk University, 2010

A thesis submitted in partial fulfillment of the requirements for the
degree of Master of Computer Information Systems Department,
Yarmouk University, Irbid, Jordan


Approved by:

Bilal A. H. Abul-Huda  **Chairman**

Full Professor of Computer Information Systems, Yarmouk University

Izzat M. Alsmadi  **Co-supervisor**

Associate Professor of Computer Information Systems, Yarmouk University

Ahmad A. Saifan  **Member**

Assistant Professor of Computer Information Systems, Yarmouk University

Fawaz A. AL Zaghoul  **Member**

Full Professor of Computer Information Systems, The University of Jordan

MAY 05, 2013

ACKNOWLEDGMENT

I would like to thank Allah for giving me the patience to work hard and overcome my research obstacles.

Foremost, I would like to express my sincere gratitude to my advisors Professor Dr. Bilal A. H. Abul-Huda and Dr. Izzat Alsmadi for the continuous support of my Master study and research, for their patience, motivation, enthusiasm, and immense knowledge. The guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisors and mentors for my Master study.

Besides my advisors, I would like to thank the rest of my thesis committee: Dr. Ahmad Saifan, and Professor Dr. Fawaz AL Zaghoul, for their encouragement, insightful comments, and hard questions.

My thanks to my friends for their honest friendship, care, and for being kind to provide help and support.

I am deeply and forever indebted to my parents and my wife for their love, support and encouragement throughout my entire life.

Qosai AL_zoubi

MAY 05, 2013

TABLE OF CONTENTS

<u>Contents</u>	<u>Page</u>
ACKNOWLEDGEMENT	I
TABLE OF CONTENTS	II
LIST OF FIGURES	V
LIST OF TABLES	VII
LIST OF ABBREVIATIONS	VIII
ABSTRACT	IX
I. INTRODUCTION	1
1.1 Overview	1
1.2 Dealing with Quality Problems	1
1.2.1 Dynamic Testing	2
1.2.2 Metrics	2
1.2.3 Source Code Analysis Tools (static testing)	3
1.3 Sample of Source Code Analysis Tools	3
1.3.1 StyleCop	3
1.3.2 JustCode	4
1.3.3 FxCop	5
1.4 Problem Statement	5
1.5 Research Objective	6
1.6 Research Importance	7
1.7 Thesis Structure	8
II. RELATED WORK	9
2.1 Software Metrics and Class Change Proneness	9
2.2 Testing and Source Code Analysis Tools	14
2.3 Software Quality	17

2.4 Maintenance	20
III. STATIC CODE ANALYSIS TOOLS	23
3.1 Static Code Analysis Tools	23
3.1.1 Warnings	24
3.1.2 Errors	25
3.1.3 Information	25
3.2 What can Static Code Analysis Accomplish?	26
3.3 Analysis and Comparison: Source Code Analysis Tools	27
3.3.1 Analysis: Source Code Analysis Tools	27
3.3.1.1 StyleCop Tool	28
3.3.1.2 JustCode Tool	47
3.3.1.3 FxCop Tool	49
3.3.2 A Comparison between the Tools	51
IV. RESEARCH GOALS AND APPROACHES	54
4.1 Differences in the Results that come from each Tool for the Same Source Code	57
4.2 Weaknesses of the Two Evaluated Tools	62
4.3 Inconsistency Issue	65
4.4 Tool Implementation	68
4.4.1 Requirements	68
4.4.2 Implementation	68
V. EXPERIMENTAL RESULTS AND ANALYSIS	71
5.1 Source Code Analysis Tool Warnings Extraction	71
5.1.1 Maintainability Warnings Extraction	73
5.1.2 Naming Warnings Extraction	76
5.1.3 Ordering and Layout Warnings Extraction	78
5.2 The Automatic Modification of Proposed Warnings on Tested	80

Code	
5.2.1 Maintainability Recommendations Automatic Modification	81
5.2.2 Naming Recommendations Automatic Modification	85
5.3 Comparing Between Our Own Tool and Other SCA Tools	87
VI. HOW TO USE THE DEVELOPED TOOL	94
6.1 Maintainability Warnings Extraction	94
6.2 Naming Warnings Extraction	96
6.3 Ordering and Layout Warnings Extraction	97
VII. CONCLUSIONS AND FUTURE WORK	98
7.1 Conclusions	98
7.2 The Limitations and Weaknesses	99
7.3 Future Work	99
VIII. REFERENCES	100

LIST OF FIGURES

<u>Figures</u>	<u>Title</u>	<u>Page</u>
Figure 4.1:	Methodology phases	55
Figure 4.2:	Generated warnings from StyleCop tool	66
Figure 4.3:	Repeated warnings from StyleCop tool	66
Figure 4.4:	Inconsistency between the JustCode recommendations and alterations	67
Figure 5.1:	The process of automatic modification on the class element	81
Figure 5.2:	The process of automatic modification on the method element	82
Figure 5.3:	The process of automatic modification on the property element	83
Figure 5.4:	The process of automatic modification on the struct element	83
Figure 5.5:	The process of automatic modification on the enum element	84
Figure 5.6:	The process of automatic modification on the private field element	85
Figure 5.7:	Modification on the non-private field element in naming warnings	86
Figure 5.8:	Modification on the interface element in naming warnings	86
Figure 5.9:	Modification on the field element that starts with underscore in naming warnings	87
Figure 5.10:	StyleCop tool warnings results	90
Figure 5.11:	Our tool warnings results	90
Figure 5.12:	Apply StyleCop on MarsMission project	91
Figure 5.13:	Apply our own tool on MarsMission project	91
Figure 5.14:	Applying StyleCop on MarsMission project in second run	92
Figure 5.15:	Public field issue	93

Figure 6.1:	Main menu of our SCA tool	94
Figure 6.2:	Browse for folder dialog	95
Figure 6.3:	Results after clicking on "Maintainability Warnings"	95
Figure 6.4:	The process of choosing a file	96
Figure 6.5:	The obtained naming warnings and the modification on the source code	96
Figure 6.6:	Applying "Ordering Rules" on the source code	97
Figure 6.7:	Applying "Layout Rules" on the source code	97

LIST OF TABLES

<u>Tables</u>	<u>Title</u>	<u>Page</u>
Table 1.1:	A sample SCA warning classification	4
Table 3.1:	Spacing rules and examples	29
Table 3.2:	Readability rules and examples	35
Table 3.3:	Ordering rules	38
Table 3.4:	Naming rules and examples	39
Table 3.5:	Maintainability rules and examples	42
Table 3.6:	Layout rules	44
Table 3.7:	Documentation rules	46
Table 4.1:	An overview of the projects	57
Table 4.2:	Distribution the Chatters warnings on classes of warning	58
Table 4.3:	Distribution the Design warnings on classes of warning	58
Table 4.4:	Result from applying SCA tools on Chatters project	59
Table 4.5:	Result from applying SCA tools on Design project	60
Table 4.6:	Example code MarsMission and JustCode recommendation	63
Table 4.7:	Example code MarsMission and StyleCop recommendation	64
Table 5.1:	All rules of warnings and description	72
Table 5.2:	Maintainability warnings extraction examples	74
Table 5.3:	Naming warnings extraction examples	77
Table 5.4:	Ordering and Layout warnings extraction examples	79
Table 5.5:	Comparing between our own tool and other SCA tools	88

LIST OF ABBREVIATIONS

API	Application Programming Interface
DRE	Defect Removal Efficiency
EENOM	Earliest Evolution Of Number Of Methods
GUI	Graphical User Interface
IUC	Interface Usage Cohesion
JIT	Just In Time
LENOM	Latest Evolution of Number Of Methods
LOC	Lines Of Code
NASA	National Aeronautics and Space Administration
NHCTMC	Non-homogeneous Continuous Time Markov Chain
OO	Object Oriented
OOAD	Object-oriented analysis and design
SUT	Software Under Test
SCA	Source Code Analysis
SCC	fine-grained Source Code Changes
SDLC	Systems Development Life Cycle

ABSTRACT

Software testing is one of the most comprehensive phases in the software projects where it takes a huge amount of time and resources. Testing however is not only the process that occurs after the implementation executing the program looking for faults to fix them. There are many supplementary testing activities that may occur within the testing stage or outside that lead to the same goal as of testing to improve the developed software and reduce effort required to use it, test it, maintain or update it. One of those supplementary testing activities is source code analysis. Source code analysis focuses largely on warnings, not errors, where such warnings indicate possible violation of naming standards or best practices. Such violation may in future leads to errors and hence should be handed early. Source Code Analysis (SCA) tools such as: MS StyleCop and JustCode have been developed to help developed areas in their code that should be improved or modified to eliminate the display if warnings. Some of those tools are integrated with programming languages environments and compilers. The main objective of this thesis is to propose and develop an SCA tool that can improve some of the limitations in the evaluated SCA tools. In order to achieve our main objective, we first conducted an evaluation or assessment case study looking for limitations and weaknesses in the existing evaluated SCA tools. Based on such initial assessment and comparison, a list of candidate requirements for the new SCA tool is assembled. The developed or assembled tool can perform the following tasks: Detect several categories of warnings, propose solutions to remove those warnings and automatically apply those warnings if the user or the developer wants to do so. The main contribution of this thesis is the development of a new SCA tool that can override some of the limitations of the evaluated SCA tools. The new tool tried to take the good options of both tools and bypass or avoid their limitations. Results showed that, based on the four warning categories that we focused on, our tool showed better results in overcoming some of the inconsistency problems or problems related to the automatic implementation of recommended corrections.

Key Words: source code analysis tools, static code analysis tool, maintainability warnings, software testing, software quality.

المخلص

اختبار البرمجيات هو واحد من أكثر المراحل الشاملة في مشاريع البرمجيات، حيث يستغرق قدرا كبيرا من الوقت والموارد. لكنه ليس فقط العملية التي تحدث بعد تنفيذ البرنامج للبحث عن أخطاء لإصلاحها. هناك العديد من أنشطة الاختبار التكميلية التي قد تحدث داخل أو خارج المرحلة التي تؤدي إلى نفس الهدف من الإختبار لتحسين البرمجيات المطورة وتقليل الجهد المطلوب لاستخدامه، واختباره، والحفاظ عليه أو تحديثه. واحدة من أنشطة الاختبار التكميلي هو تحليل البرامج. ركز تحليل البرامج بشكل كبير على التحذيرات، لا الأخطاء، حيث تشير مثل هذه التحذيرات إلى انتهاك محتمل لمقاييس التسمية أو أفضل الممارسات. مثل هذا الانتهاك قد يؤدي إلى أخطاء، وبالتالي يجب أن يتم السيطرة عليها مبكرا. برامج تحليل البرامج مثل StyleCop و JustCode تطورت للمساعدة في عملية تطوير المساحات في التعليمات البرمجية الخاصة بهم التي ينبغي تحسينها أو تعديلها للقضاء على مكان ظهور التحذيرات. بعض هذه البرامج يتم دمجها مع بيئة لغات البرمجة والمترجم. الهدف الرئيسي لهذه الأطروحة هو اقتراح وتطوير أدوات SCA التي يمكن أن تحسن بعض القيود في أدوات SCA. من أجل تحقيق هدفنا الرئيسي، أجرينا أولا "تقييم أو دراسة حالة تقييم للبحث عن القيود ونقاط الضعف في أدوات SCA. وبناء على هذا التقييم الأولي والمقارنة، تم تجميع قائمة من المتطلبات المرشحة للتنفيذ في الأداة الجديدة التي قمنا ببنائها. أهم الوظائف التي احتوت عليها الأداة التي قمنا بتصميمها وبنائها: تستطيع الأداة الكشف عن عدة فئات من التحذيرات مثل الأدوات التي قيمناها مع بعض التحسينات، واقتراح حلول لإزالة تلك التحذيرات وتطبيق الحلول تلقائيا لتلك التحذيرات إذا كان المستخدم أو المطور يريد أن يفعل ذلك. إذن فإن المساهمة الرئيسية لهذه الأطروحة هي استحداث أداة SCA جديدة والتي لديها قدره على تجاوز بعض المشاكل في الأدوات المقيمه سلفا. استنادا إلى الفئات الأربعة من أنواع التحذيرات التي قمنا بتقييمها ومعالجتها تلقائيا أثبتت الأداة الجديدة التي قمنا ببنائها قدرة أفضل من الأدوات المقيمه لهذا الغرض أو أظهرت دمج بين حسنات الأدوات———— بين المقيمتين.

CHAPTER ONE

INTRODUCTION

1.1 Overview

Through the software development life cycle a series of changes need to be accomplished. These changes are required because of many reasons such as; enhancement, adaption, and maintenance or fixing the program defects (Bieman, *et al*, 2003). From these changes and results we can say the software is infinitely flexible (Koru.2005). However, changes must be considered as major risk elements, since they may impact time and cost (Koru & Liu, 2007). In addition, change-proneness of the software may lead to specific important quality issues (Bieman, *et al*, 2003).

The change history of software code provides useful information about the evolution of programs. This information helps us to understand the overall picture of the system evolution starting from design phase ending with maintainability phase (Al-khiaty.2009).

Software quality is a serious issue to consider, since software is entering in all life details starting from simple industries like children toys ending to industries like airplane.

1.2 Dealing with Quality Problems

To deal with the quality problems we need to study how can we test and measure the source code itself. The results from these studies and measurements provide useful information that can help in solving such quality problems.

1.2.1 Dynamic Testing

Dynamic testing or analysis focuses in accomplishing customer requests by supporting all requirements and functionalities by the software as a final product (Lochmann & Goeb, 2011).

Software testing tools are programs that try to find errors, defects, bugs, failures, etc. in the evaluated software products. Those different terms are, sometime, different based on the level and the nature of the errors. The errors are unexpected behavior of the system. The defects refer to the many problems related to software products, either external behavior or internal features, but a fault in a program which causes the program to perform in an unintended or unanticipated manner. The failure that means the system does not deliver a service as expected by it is user. The output of each test case in a testing process is one of two: pass or fail. The designer of the test cases defines the inputs for each test case along with expected outputs. On the execution, test cases are executed and actual results are compared with expected results. For those failed test cases (i.e. expected result is different from the actual result), a debugging process further starts to see why those test cases produce incorrect outputs or results. Errors can be syntax, semantic, functional, and non-functional. Errors may stop the compilation process or may not and only cause different or unexpected behavior from those defined by users.

1.2.2 Metrics

Studying class characteristics and identifying their attributes in terms of changes is very useful in the maintenance process. Consequently, this will make project manager and team to give more attention to the possibility of changes in classes during the

project life cycle (Bieman, *et al*, 2003). Here where the importance of measuring software metrics takes place.

1.2.3 Source Code Analysis Tools (static testing)

Many quality aspects can be identified by using metrics. Thus, software metrics are tools to measure one or more code attributes (EKLÖF.2011).

Source code analysis (SCA) tools are used to check the source code for attributes such: number of lines of code or any other static metrics of the code. Examples of such static metrics include: Lines Of Code (LOC), size, and complexity. It can be applied after the code is written which means that it may help us to learn about the code and possibly catch defects before testing phase. Although SCA cannot find all kinds of defects, it can be considered as an efficient tool in terms of cost and time (EKLÖF.2011). SCA tools are usually applied automatically with the least amount of effort and time from the users or testers side.

1.3 Sample of Source Code Analysis Tools

In this section, we will list some tool examples that are applied on the source code specially those that we used in our experimental studies.

1.3.1 StyleCop

StyleCop is an open source static SCA tool from Microsoft that checks .NET code for conformance of several design guidelines defined based on Microsoft's .NET Framework (CodePlex.2011). StyleCop analyzes the code in order to apply a set of rules which can be classified into several categories such as (CodePlex.2011): Naming, maintainability, documentation, ordering, readability, spacing, and layout. Table 1.1 shows a sample of some warnings and their classification.

Table 1.1: A sample SCA warning classification

Warnings	Categories
The spacing around an operator symbol is incorrect.	Spacing
The call to channel should only use the 'base.' prefix if the item is declared virtual in the base class and an override is defined in the local class. Otherwise, prefix the call with this rather than base.	Readability
All using directives must be placed inside of the namespace	Ordering
Method names begin with an upper-case letter.	Naming Rules
The class must have an access modifier	Maintainability
A statement containing curly brackets must not be placed on a single line. The opening and closing curly brackets must each be placed on their own line.	Layout
The constructor must have a documentation header.	Documentation

1.3.2 JustCode

JustCode is another example of SCA tools. There are some JustCode features that include (Telerik.2011): On-the-fly Code analysis, code navigation and search, refactoring, quick fixes, coding assistant and hints. JustCode executes its code analysis by applying custom inspections. There are several inspections that can be performed by JustCode. Examples include (Telerik.2011): Identical if and else clauses, obsolete casts, empty statements, assignments with no effect, unused private members, unused parameters, variables, namespaces, or statements. Figure 1.1 shows a sample of SCA output from JustCode.

```
public int Foo()
{
    return "bar";
}
// C#: An instance of type "string" cannot be returned by a method of type "int"
```

Errors – by default Just Code underlines errors with a red line

1.3.3 FxCop

FxCop is another example of SCA tools. FxCop is an application that resolves assembly codes after the source codes are compiled, and notifies information about the code assemblies, such as security improvements, possible design, performance and localization (MSDN, 2013).

FxCop is intentional for class library developers. But, anyone making applications that should conform to the .NET Framework best exercises will benefit. Also, FxCop is useful as a pedagogical tool for people who are uncommon with the .NET Framework Design Guidelines or who are fresh to the .NET Framework (MSDN, 2013).

FxCop is developed to be fully merged into the Systems Development Life Cycle (SDLC) and is distributed as both a command-line tool (FxCopCmd.exe) appropriate for integrated with Microsoft Visual Studio or usage as part of automated build processes .NET as an exterior tool. And a fully distinguished application that has a Graphical User Interface (GUI) (FxCop.exe) for interactive work (MSDN, 2013).

1.4 Problem Statement

Static source code analysis tools are software programs that are used to evaluate programs statistically and evaluate certain characteristics based on predefined quality standards. Unlike software testing where expected output will be (pass or fail) based on the conformance of expected outcome with the actual outcome. In SCA, the output will be one of three classes: error, warning or information.

Criteria are defined for what standard or typical program should be or should have. Based on those standards, a subject code is evaluated depending on the level of

conformance or violation of a standard, one of the three classes (i.e. error, warning, or information) is defined to show some quality aspects of the evaluated software.

First, we have evaluated several selected free and commercial SCA tools for the purpose of comparing, correlating and assessing the results. Our focus is on the warning class of issues as it is considered as a vague class between errors and information where many developers underestimate or ignore warning signs.

Second, we have evaluated the relations and the correlation between SCA reported warnings. Extensive statistical analyses from all evaluated SCA tools are conducted to evaluate the ability of warning reports by SCA tools to predict bugs or defects.

Based on those relations from the different SCA tools, we have first listed the important characteristics from all warning classes that were significant to bugs or defects.

Moreover, we have proposed enhancements on SCA and developed a tool to consider the major warning classes that showed high defect predictability values. The last goal that we have performed is to evaluate the correlations between data from software metrics tools and SCA tools.

1.5 Research Objectives

Based on the problem statement, we defined three major objectives that are accomplished in this thesis:

- Extensively evaluate several selected free and commercial SCA tools for the purpose of comparing, correlating and assessing the reported information. Expected outcome has included statistical data from several

open source evaluated projects that show all classes of warnings collected from the selected SCA tools. Moreover, the similarities and differences between the SCA tools will be shown.

- Evaluate the inconsistency of results and the kind of warnings that may vary from one experiment to another given the same tool and tested source code. Expected output have data and reports with inconsistency between reported warnings in the tools when apply these tools more than one run or test.
- Proposed enhancements on SCA and developed a tool to consider the major warning classes that showed high defect predictability values. Expected output is a tool or, for the least, a framework for the relevant and important SCA warning information combined from all evaluated SCA tools and possibly adding new warning classes discovered through this thesis and evaluate the correlations between data from software metrics tools and SCA tools.

1.6 Research Importance

Software quality tools are used to assess quality of software through all development stages. However, there is a little public information about test evaluation of the accuracy and value of the warning that are reported from some of these tools (Ayewah, *et al*, 2007).

By using static SCA tools we can study the architecture of the source code packages (EKLÖF.2011). Therefore, we have tested several codes downloaded from SourceForge.NET to evaluate the value of different warning messages in that code

project and see if such warning messages can correlate with bug or defect data collected from the source codes.

1.7 Thesis Structure

The following chapters of this thesis are organized as the following: Chapter two presents related studies to software quality. Chapter three presents static code analysis tools. Chapter four shows the research goals and approaches. Chapter five presents experimental results and analysis. Chapter six describes how to use the proposed tool. Chapter seven presents the conclusions and future work.

CHAPTER TWO

RELATED WORKS

This chapter is a literature survey of the previous work that search in the history of software metrics, software analyzing, and software maintainability in order to enhance the quality and maintainability even after the product released.

It is divided into four sections starting with first section that describe software metrics their importance as attributes of software, and their role in facilitating software maintainability. Second section describes software quality. The Third section considers testing and SCA tools. Finally fourth section is dealing with software maintainability and changes as the final step in the software development life cycle.

2.1 Software Metrics and Class Change Proneness

Studying software metrics class characteristics and identifying their attributes in term of changes is very useful in the maintenance process. Consequently, this will make encourage project manager and his team to give more attention to the possibility of changes in classes during the project life cycle (Bieman, *et al*, 2003). Here where the importance of measuring software metrics take place.

According to Girba et al. (2004), their approach depends on the changes in the evolution of the Object Oriented (OO) software system by providing historical measurement study. The study focuses on the change in the history of a class by observing the change in the nature of methods in different versions, that means they measure the change by using one main code attribute (number of methods) add or remove method to certain class. Form the number of methods metrics can be derived

another two different metrics, the Latest Evolution Of Number Of Methods (LENOM) and the Earliest Evolution Of Number Of Methods (EENOM). By these two metrics the change in size inside each class over the software history different versions can be known and changes here focus only on the number of methods that added or removed from each class over different releases.

Koru and Liu (2007) focus on change-prone classes by providing tree-based model that shows the class characteristics, they test Pareto's law for the open source code programs which state that 80% of code changes are centered at 20% of the classes. They mainly searched in how to identify change-prone classes and their characteristics by trying to observe the change of set of static metrics of a group of products with different releases of an open source project, they prove the validity and applicability of Pareto's law for open source programs, they also provide useful guidance in development and maintenance of large-scale open source programs.

According to Basten and Klint (2009), finding and discovering the facts from a source code is an important step while software analysis is done. Several experiments are done and found that extracting facts from any source code then writing them in a large wide of programming languages; it will lead to hard working and error prone. Because of these reasons they developed a new technique which called DeFacto. It is language-parametric analysis software for fact extraction from the software source code.

According to Bieman, et al. (2003), four research questions were treated. The first research question was about visualization and identification of change-prone sets of classes in an object-oriented framework. The second research question was to do with differentiating change-prone clusters from local change-proneness of classes. Also this method was displaying how to determine the degree to which classes are change-prone

both in their interplays with others and locally. This method was applied to a considerable case study. For this case study, in response to the third research question that which modifies interplays between classes do not necessarily imitate functional interplays in the resolve of the framework. This which can have a diversity of causes. An example would be refinements of specific factors such as performance. Performance refinements may trigger concurrent alterations in classes that otherwise do not react with each other. On the other hand, in response to fourth research question, cluster change-proneness versus local was visualized through the alter-architecture graph and paralleled it to the design graph. We also differentiated between alter-prone clusters of classes which did not include in patterns and those which are included. The visualization was straightforward and simple and driven by the alteration measures that were identified. Future work in this field involves the representation of other measurements such as size of box symbolizing size of class, utilizing of color, and covers of alter-architecture versus rational architecture.

According to Romano and Pinzger (2011), interfaces declare contracts that are denoted to stay stable during the development of a software framework while the concrete classes implementation (a subclass class can be instantiated that implements all the missing functionality) is more likely to alter. This guide to another evolutionary demeanor of interfaces paralleled to concrete classes. This behavior was experimentally examined with the C&K metrics that are broadly utilized to estimate the implementation quality of interfaces and classes. The outcomes of the study with two Hibernate projects and eight Eclipse plug-in and indicate that, the Interface Usage Cohesion (IUC) metric shows a more powerful connection with the number of fine-grained Source Code Changes (SCC) than the C&K metrics when stratified to interfaces, also The IUC metric

can ameliorate the performance of foretelling models in categorizing Java interfaces into two categories, change-prone and not change-prone.

According to Romano et al. (2012), Anti-patterns have been defined to mean “poor” solutions to resolve and perform problems. Previous researches have indicated that classes impacted by anti-patterns are more change-prone than classes that did not impact by anti-patterns. A deeper premeditation was provided into which anti-patterns direct to which kinds of alterations in Java classes. The change-proneness of these classes was analyzed taking in consideration 40 kinds of (SCC) derived from the version control depository of 16 Java open-source frameworks. Classes impacted by anti-patterns alter more repeatedly along the development of a framework; Classes impacted by the SwissArmyKnife, ComplexClass, and SpaghettiCode anti-patterns are more probable to be altered than classes impacted by other anti-patterns in addition that, specific anti-patterns lead to specific kinds of source code alterations, like as Application Programming Interface (API) alterations are more probable to be shown in classes impacted by the SwissArmyKnife, ComplexClass, and SpaghettiCode anti-patterns.

Shatnawi and Li (2008) investigated three publications of the Eclipse project and detected that although several software metrics can still prognosticate class fault proneness in three errors - acuteness categories, the thoroughness of the prognosis minimized from publications to publications. Moreover, the Researchers detected that the prognosis cannot be utilized to construct a software metrics paradigm to recognize fault-prone classes with admissible accuracy. SHATNAWI's findings propose that as a software develops, the utilize of certain usually utilized metrics to recognize which classes are more prone to faults turns into increasingly complicated.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

REFERENCES

- Abraham, J. and Friedman, J. 2012. *Building Confidence in the Quality and Reliability of Critical Software*. CrossTalk.
- Al-Khiaty, Mojeeb 2009. *Software evolution metrics for object-oriented software changeability prediction*. Department Computer Science King Fahd University of Petroleum and Minerals, Saudi Arabia.
- Ayewah, N., Pugh, W., Morgenthaler, J., Penix, J. and Zhou, Y. 2007. *Evaluating Static Analysis Defect Warnings on Production Software*. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07), San Diego, California, USA. Pp 1-8.
- Basten, H. J. S and Klint, P. 2009. DeFacto: Language-Parametric Fact Extraction from Source Code. *Springer-Verlag Berlin Heidelberg*. Pp 265-284.
- Bernstein, A., Ekanayake, J. and Pinzger, M. 2007. *Improving Defect Prediction Using Temporal Features and Non Linear Models*. In Proceedings of Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting (IWPSE '07). Dubrovnik, Croatia. Pp 11-18.
- Bieman, J., Andrews, A. and Yang, H. 2003. *Understanding Change-proneness in OO Software through Visualization*. In Proceedings of the 11th IEEE International Workshop on Program (IWPC). Portland, OR, USA. Pp 44-53.

- Bieman, J., Jain, D. and Yang, H. 2001. *OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study*. In Proceedings of IEEE International Conference on Software Maintenance, Florence Pp 580–589.
- Black, P., Kass, M., Koo, M. and Fong, E. 2011. *Source Code Security Analysis Tool Functional Specification Version 1.1*. Software and Systems Division.
- Canfora, G. and Cimitile, A. 2000. *Software Maintenance*. University of Sannio, Faculty of Engineering at Benevento, Italy.
- Deissenboeck, F., Juergens, E., Lochmann, K. and Wagner, S. 2009. *Software Quality Model: Purposes, Usage Scenarios and Requirements*. In Proceedings of Software Quality, 2009. WOSQ '09. ICSE Workshop on, Vancouver, Canada. Pp 9-14.
- Deissenboeck, F., Wagner, S., Teuchert, S. and Girard, J.-F. 2007. *An Activity-Based Quality Model for Maintainability*. In Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM). Paris, France. Pp 184-193.
- Drake, T. 1996. *Measuring Software Quality: A Case Study*. *Journal of IEEE Computer*. 29 (11): 78-87.
- Edberg, D., Ivanova, P. and Kuechler, W. 2012. Methodology Mashups: An Exploration of Processes Used to Maintain Software. *Journal of Management Information Systems*. 28 (2): 271- 304.
- EKLÖF, RICKARD 2011. *Improving software Development with Static Code Analysis in a Traceable Environment*. Master's Thesis at Machine Design.

- Engelbertink, F, VOGT, H. 2010. *How to save on software maintenance costs*.
- English, M., Exton, C., Rigon, I. and Cleary, B. 2009. *Fault Detection and Prediction in an Open-Source Software Project*. In Proceedings of the 5th International Conference on Predictor Models in Software Engineering Article. New York, USA.
- Girba, T., Ducasse, S. and Lanza, M. 2004. *Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes*. In Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 40–49.
- Gomes, I., Morgado, P., Gomes, T. and Moreira, R. 2009. *An overview on the static code analysis approach in software development*. Tech. rep., Faculdade de Engenharia da Universidade do Porto.
- Gyimothy, T., Ferenc, R. and Siket, I. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *Journal IEEE Transactions on Software Engineering*. 31(10): 897-910.
- Jones, C. 2012. *Software Quality Metrics: Three Harmful Metrics and Two Helpful Metrics*.
- Khaddaj, S. and Horgan, G. 2005. A Proposed Adaptable Quality Model for Software Quality Assurance. *Journal of Computer Sciences* 1(4): 482-487.
- Koru, G., and Liu, H. 2007. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*. 80(1): 63–73.

- Koru, G. and Tian, J. 2005. Comparing High Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products. *IEEE Transactions on Software Engineering*. 31 (8): 625-642.
- Kuhn, A., Ducasse, S. and Girba, T. 2007. Semantic Clustering: Identifying Topics in Source Code. *Journal on Information Systems and Technologies* 49(3): 230-243.
- Lochmann, K. and Goeb, A. 2011. *A Unifying Model for Software Quality*. In Proceedings of the 8th international workshop on Software quality (WoSQ'11), Szeged, Hungary. Pp 3-10.
- Lucia, A., Deufemia, V., Gravino, C. and Risi, M. 2010. *An Eclipse plug-in for the Detection of Design Pattern Instances through Static and Dynamic Analysis*. In Proceedings of 26th IEEE International Conference on Software Maintenance (ICEM). Timisoara, Romania. Pp 1-6.
- Mahmood, Waqas and Akhtar, Muhammad 2010. Validation of Machine Learning and Visualization based Static Code Analysis Technique. Master Thesis Computer Science.
- Riaz, M., Mendes, E. and Tempero, E. 2009. *A Systematic Review of Software Maintainability Prediction and Metrics*. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09). IEEE Computer Society Washington, DC, USA. Pp. 367-377.
- Romano, D. and Pinzger, M. 2011. *Using source code metrics to predict change-prone Java interfaces*. In Proceedings of 27th International Conference on Software Maintenance (ICSM'11), IEEE Computer Society, Washington, DC, USA. Pp 303-312.

- Romano, D., Raila, P., Pinzger, M. and Khomh, F. 2012. *Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes*. In Proceedings of 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada.
- Sharif, B. and Maletic, J. 2010. *The Effects of Layout on Detecting the Role of Design Patterns*. In Proceedings of the 2010 23rd IEEE Conference on Software Engineering Education and Training (CSEET '10). Pp 41-48.
- Shatnawi, R. and Li, W. 2008. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software*. 81(11): 1868-1882.
- Slaughter, S. and Delwiche, L. 1995. *Errors, Warnings, and Notes (Oh My) A Practical Guide to Debugging SAS Programs*. University of California.
- Vink, G. and BV, A. 2010. Static Code Analysis (SCA) Standardization Efforts & Integration in the Software Development Flow.
- Xiong, C, Xie, M. and Ng, S. 2011. Optimal software maintenance policy considering unavailable time. *Journal of Software Maintenance and Evolution: Research and Practice*. 23(1): 21-33.
- Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. and Vouk, M. 2006. On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering* 32(4): 240-253.
- Zhou, Y. and Leung, H. 2006. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *Journal IEEE Transactions on Software Engineering*. 32(10): 771-789.

CODEPLEX, 2006. *Project Hosting for Open Source Software*. Retrieved October, 8, 2011 from the World Wide Web: <http://stylecop.codeplex.com/>.

MSDN, 2012. *FxCop*. Retrieved March, 10, 2013 from the World Wide Web: <http://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx>.

TELERIK, 2002. *JustCode*. Retrieved October, 15, 2011 from the World Wide Web: <http://www.telerik.com/products/justcode.aspx>.

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction	العنوان:
Al Zoubi, Qosai Mwafeq	المؤلف الرئيسي:
Abu Alhuda, Bilal A. H., Al Smadi, Izzat M.(Advisor, Co-Advisor)	مؤلفين آخرين:
2013	التاريخ الميلادي:
إربد	موقع:
1 - 105	الصفحات:
743131	رقم MD:
رسائل جامعية	نوع المحتوى:
English	اللغة:
رسالة ماجستير	الدرجة العلمية:
جامعة اليرموك	الجامعة:
كلية تكنولوجيا المعلومات وعلوم الحاسوب	الكلية:
الاردن	الدولة:
Dissertations	قواعد المعلومات:
هندسة الحاسوب، البرمجيات، برامج الحاسوب	مواضيع:
https://search.mandumah.com/Record/743131	رابط:

Traceability Enhancements on Source Code Analysis Tools to Improve Software Defects Prediction

by:

Qosai Mwafeq AL-Zoubi

Supervisor:

Professor Dr. Bilal A. H. Abul-Huda

Co-Supervisor:

Dr. Izzat Alsmadi

Computer Information Systems Department

Yarmouk University

May 05, 2013