MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei
Facultatea Calculatore, Informatică și Microelectronică
Departamentul Inginerie Software și Automatică

Admis la susținere
Șef departament: Fiodorov I., dr. conf. univ.

_____
„ __ ” _____ 2024

Analiza și implementarea metodelor și tehnicilor de proiectare a unui analizor de cod static

Proiect de master

Student: _____ Nani Victor, IS-221M

Coordinator: _____ Zaharia Gabriel, asis. univ.

Consultant: _____ Catruc Mariana, lect. univ.

Chișinău, 2024

Rezumat

Uneltele folosite pentru analiză statică de cod sunt esențiale și, în același timp, prezintă multe elemente care încă necesită să fie explorate în lumea modernă a ingineriei software. Acestea sunt folosite din ce în ce mai mult de către dezvoltatorii software, indiferent de magnitudinea proiectului, dar totuși doar un număr mic de dezvoltatori cunoaște, de fapt, cum funcționează aceste unelte uimitoare. Tocmai acesta este unul din scopurile lucrării, și anume să prezinte principiile de funcționare care definesc un analizor de cod static. Acest articol descrie structurile fundamentale care reprezintă baza unui analizor, precum și modurile de implementare ale acestora, atât din punct de vedere teoretic, cât și din punct de vedere a codului. Unul din beneficiile care survin la implementarea unui analizor de cod static este numărul redus de tehnologii necesare implementării. La nivel inițial, abordarea unui inginer care încearcă să dezvolte o unelță asemănătoare este strict algoritmică - practică încearcă să găsească soluții pentru implementarea mecanismelor de analiză a codului. Java a fost limbajul de programarea ales pentru implementare și toate soluțiile prezentate în acest raport sunt în corespondență directă cu Java. Acest raport are drept scop enunțarea problemelor care reprezintă premisele dezvoltării unei unelte de analiză statică a codului, stadiul curent de dezvoltare, rezultatele cercetării, precum și determinarea soluțiilor.

Capitolul introductiv va ghida cititorii prin fundamentele ce descriu subiectul ales. Următorul capitol va descrie problemele ce au condus la dezvoltarea proiectului respectiv și va prezenta statutul actual de dezvoltare. În continuare, vor fi descrise tehnologiile utilizate în cadrul dezvoltării acestui proiect, urmat de un capitol dedicat procesului de cercetare. O continuare a capitolului de cercetare este capitolul de prezentare practică a soluției, care prevede definirea aspectelor practice și pașii necesari pentru dezvoltarea componentei software. Întreg raportul va fi rezumat în capitolul dedicat concluziilor.

Cuvinte cheie— analizor de cod static, motivație, cercetare, soluții

Abstract

Static code analysis tools are as essential, as they are uncharted in the modern world of software engineering. They are used more and more by software developers regardless of the magnitude of the project. Yet, a very small number of engineers do actually know how such a magnificient tool works under the hood. This is one of the purposes of the following thesis, namely to expose the principles that define a static code analyzer. The article describes the fundamental structures that represent the basis of the analyzer, as well as describing a few solutions to implement these structures, theoretically and code-wise. One of the benefits a developer may find while implementing an analysis tool is that not a lot of technologies are used during the process. At inception level, the approach taken by an engineer trying to develop such a tool is strictly algorithmic - basically trying to find different algorithms to create the mechanics of the analyzer. Java was the language of choice when it came to implementation and all the solutions exposed in this paper will be in connection with Java code. This report will guide you through the problems that acted as premises for building static code analysis tools, the actual stage of development, the research results and will guide you through the solution as well.

The introductory chapter will guide the readers through the foundations of the selected subject. The next chapter will establish the problems which led to development of this project and present the state of the art. Next, the used technologies on developing the project will be presented, followed by a dedicated chapter to the research process, where the acquired knowledge and materials about the subject will be described. A sequel to the research chapter will be the solution chapter, where the practical aspects and the steps required for development will be presented. The entire article will be summarized in the conclusions chapter.

Keywords— static code analyzer, motivation, research, solution

Contents

INTRODUCTION

The software engineering has taken the world by storm in the last decades. In a short amount of time, it has managed to incorporate itself in any possible, being predominantly used in fields like economy, architecture, design, other engineering fields and even medicine. This is a milestone that was only a dream some time ago. Since its inception, the software world has proven itself to be extremely useful, automating a lot of work and speeding up the delivery of results. For all that, nothing of such a magnitude comes without a price. The fundamental unit in the engineering world is the code. Taking into consideration the growth and development rate of the software technologies and frameworks, one of the challenges that developers face is code maintenance. This challenge generates a series of problems on their own, which requires a dedicated tool to aid the software engineers facing them. These altogether have created a new area to explore and the results are presented in this report. In the next few paragraphs, an overview and the aim of the project will be presented.

The purpose of this project is to analyze in depth a static code analysis tool, which is a tool that analyzes source code without executing the code. Static code analyzers are designed to review bodies of source code (at the programming language level) or compiled code (at the machine language level) to identify poor coding practices. This report will present in details the architecture of a static code analyzer, including the way it works under the hood.

Being a project associated with master's degree, a research practice was conducted under a supervised institution and specialized staff in order to gather specific details about the static code analyzer and their place in the software engineering world. Likewise, the proper use of such a tool was explored in depth, gathering real case scenarios when we should use an analysis tool and more important, how to use it and how to interpret the results obtained after the processing of source code. However, since there is a not a lot of information on how to actually implement such a tool from a starting to and ending point, a hands-on approach was needed in order to further dive into the essence of an analyzer.

Static analysis involves a method of analyzing without the actual execution of a program. As opposed to a compiler, which transforms the source code into machine code, a static analyzer does not affect in any ways the source code. The purpose of an analyzer is to acquire information about the program and to assure, with a certain degree of probability, that it adheres to industry standards. In essence, it will check if the program does not contain code zones that may cause unexpected behavior, if it respects the structuring mode of the code, if it possesses comprehensiveness etc.

After the analysis is performed, it presents a report that contains the acquired results together with different metrics that determine the quality of the written code. In case the code presents different malformations, the analyzer will present potential solutions, depending on the problem. For example, if a certain function or method is defined, but is not used anywhere in the program, the usual solution presented by the tool will be the removal of the code block. Therefore, we have a point of view from a different perspective of computer programs, and, in consequence, the potential to improve code quality is amplified.

Despite the advantages presented by a static code analyzer tool, it also presents some disad-

vantages as well, which may affect the programmer's overall vision on code and it may provide some information that does not perfectly describe a certain code portion. One the most frequently provided information by a code analyzer is the abuse of termination statements in a loop. Loop instructions encountered in a program are *for*, *while* and *do... while*, which repeat the instructions in a block of code a certain number of times. These blocks of code can be relatively simple or moderately complex. In the later situations, we may encounter multiple instructions that have to be executed based on certain conditions. To avoid the repeated evaluation of many conditions, programmers use termination instructions, such as *break* or *continue*, to stop the execution of certain instructions when a base condition is not satisfied, for example. Using these instructions may increase performance, because the execution of redundant instructions in certain contexts is avoided. However, some analyzers, such as *SonarQube*, alert the situations where these termination instructions are used twice or more. It is certain that abusing these instructions may suggest the improper repartition of code in methods or functions. Nevertheless, there are situations when utilizing them, even more times, is welcomed. Therefore, one of the things that should be adjusted on a static code analyzer is detailed evaluation of the context, to make sure if utilizing the termination instructions many times is welcomed or not. This is one of the multiple things when an analyzer has space for improvement.

The presented solution in this report is not unique, but it follows the generally valid principles, which can be adapted into any solution one finds fit for the project. At the same time, the solution is not complete and it was not intended to be. The aim is to investigate and to provide a practical inception of a static code analyzer, so that the readers or people searching for practical solutions will have an idea emphasized by actual examples, which can be used as a reference point to construct their own tool. The solution has not reached the point where it can be used to effectively analyze a piece of code, but it reached the phase where it is already capable of producing the required structures of data (*data* being the processed source code) by the tool in order to start the evaluation of the code and come up with some statistics regarding code quality.

To sum up, the project presents the necessary theoretical aspects, as well as some hands-on practices and examples in order to kick start one's adventure into exploring the world of static code analyzers. The exposure of information regarding this topic is done in a comprehensive manner to fit everyone's level of knowledge.

Bibliography

[1] Khatiwada, Saket; Tushev, Miroslav; Mahmoud, Anas. *Just enough semantics: An information theoretic approach for IR-based software bug localization.* Information and Software Technology. 93: 45–57, 2018-01-01.

[2] Patrick Briand, Martin Brochet, Thierry Cambois, Emmanuel Coutenceau, Olivier Guetta, Daniel Mainberte, Frederic Mondot, Patrick Munier, Loic Noury, Philippe Spozio, Frederic Retailleau. *Software Quality Objectives for Source Code*. Proceedings: *Embedded Real Time Software and Systems 2010 Conference*, ERTS2010.org, Toulouse, France, 2015-06-04.

[3] *Sonar*. Methods and Tools. Vol. 18, no. 1. 2010-03-01. pp. 40–46. ISSN 1661-402X. Retrieved November 9, 2023.

[4] Campell/Papapetrou, Ann/Patroklos. *Sonar (SonarQube) in action*. Greenwich, Connecticut, USA: Manning Publications. p. 350. ISBN 978-1617290954, 2013.

[5] *The future of TypeScript on ESLint*. ESLint - Pluggable JavaScript linter. Retrieved November 9, 2023.

[6] Brian Chess, Jacob West. *Secure Programming with Static Analysis*. Pearson Education, 2007.

[7] Darko Stefanović et al 2021 IOP Conf. Ser.: Mater. Sci. Eng. 1163 012012. *Identification of strategies over tools for static code analysis.*

[8] Akash Nagaraj*, Bishesh Sinha, Mukund Sood, Yash Mathur, Sanchika Gupta, Dinkar Sitaram. *Learning Algorithms in Static Analysis of Web Applications*. Department of Computer Science, PES University.

[9] Tim Moses, Elina Petrovna. *Static Analysis: new emerging algorithms*. BitBrainery University, London – UK, Department of Security, 2013.

[10] Hannes Tribus. *Static Code Features for a Machine Learning based Inspection*. MSE-2010-16.

[11] Parasoft. Igor Kirilenko. *Can AI/ML Encourage Devs to Adopt Static Analysis Testing?*. December 1, 2022. Retrieved September 27, 2023.

[12] Medium. Katie Wanders. *Revolutionizing Code Quality with AI-Based Static Code Analysis Tools*. October 27, 2023. Retrieved November 1, 2023.

[13] *Write once, run anywhere?*. Computer Weekly. May 2, 2002. Archived from the original on August 13, 2021. Retrieved November 13, 2023.

[14] Cucumber. Dan North. *Introducing BDD*. Retrieved November 14, 2023.

[15] *Behavior-Driven Development*. Archived from the original on 1 September 2015. Retrieved November 14, 2023.

[16] *Introduction to Behavior-Driven Development*. Keogh, Liz (2009-09-07). SkillsMatter. Retrieved November 14, 2023.

[17] John Ferguson Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2015.