2023 IEEE 8th International Conference for Convergence in Technology (I2CT)
Pune, India. Apr 7-9, 2023

Static Code Analyser to Enhance Developer Productivity

D.R.I. Peiris
Sri Lanka Institute of Information Technology
MSc (EAD)
ms21911712@my.slit.lk

Dr. Nuwan Kodagoda
Sri Lanka Institute of Information Technology
Associate Dean Faculty of Computing
nuwan.k@sliit.lk

2023 IEEE 8th International Conference for Convergence in Technology (I2CT) | 979-8-3503-3401-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/I2CT57861.2023.10126395

***Abstract*—One of the main metrics important to software development is productivity. The measure of productivity helps a organization/ individual or team identify how well their employees/ software runs and how they could improve. In this research paper, a new software is proposed along with a background study of how improving code quality will improve developers productivity and efficiency. The software proposed to aid developer productivity is a custom built static code analyser using python and it's rule set has been tailored through carefully selected research papers and with feedback from experts of the software industry.**

***Index Terms*—static analysis, productivity, code quality, readability**

I. Introduction

Good code quality is a fundamental need in today's modern world. Since source code is used universally to develop every software that runs on an electronic device, it is essential that source code must be well written and maintained in order to handle updates and enhancements. Quality code transforms into better productivity as well. As productivity is a key measure in the software industry[4] resulting in valuable business decisions, a quality code will result in better readability and stability, and in the case of code transferability to a new developer, it will save time due to easy to read code resulting in enhancing productivity. Due to the ease of readability and higher productivity it will lead to a better mental health for developers, resulting in a much healthier life style.[4] Furthermore higher productivity figures will enhance the organisations revenue, as developers will spend less time on projects and will take up another task faster than before. In this resulting chain of connections, with leads to higher revenue and better mental health the root is always a simple aspect such as having simple, clear and efficient code.

Although there are many tools currently existing in the tools to develop code such intelli-sense, wizards, dialogs shortcuts etc. in other IDE's, these impact only a specific set of scenarios such as syntax errors and indentation. Through this research a main aim is to handle specific set of scenarios/issues taken straight from the hands of experts in the software industry as well as specific scenarios/issues which can be found in specific research papers. Furthermore, these static code analysers are quite difficult to setup as one should have extensive knowledge in setting up SDK's and other dependency applications in order to make them execute.

A quality code can be defined as one that is well written, well tested and well stored. In this research we will be looking into the aspect of well written code, and how to contribute to the software industry using static code analysers to improve existing code. Since this research studies upon tackling a tailored set of scenarios and targeted languages taken from existing research papers and the software industry, the general idea and development is to create the rule set required for a static code analyser based on the research conducted.

II. Research Objectives

A. Static Code Analysis

In order to measure code quality, there are many metrics such as security, maintainability, clarity, portability etc. Since this research focuses on developing a static code analyser, the topics of clarity, re-usability and maintainability will be explored. [4]

Static analysis best defined is debugging and examining source code before a program is executed where as static code analysis is static analysis run across a set of predefined rules predefined by another program or user.[7] Static code analysis is highly customizable and different authors could code their own set of rules per organisation as to how to name variables and enhance any specific coding standards.

For an example at a software industry I worked at, when declaring variables an underscore was essential at the end of the variable. Using a custom static code analyser their teams had written a rule set to require an underscore and it was suggested in the NetBeans IDE when declaring variables. The analyser would error out the text and ask the developer to append and underscore to the end of the variable in order to deploy the code. In this research too, static code analysers will be built to give feedback to developers based on the feedback received from the quiz and through research papers.

B. Build a Questionnaire

Using a very effective method of information collection, the first step would be the create up a quiz/ questionnaire and get feedback from different developers. The sample size was around 25 senior/lead developers from different software organisations to get feedback on the most common problems they've seen with developers and why it impacts the quality of code. This sample size will randomise different developers with different levels of skills. Although the sample size is small it is very important to make sure the questions were short, simple and on point due to the candidates being busy with their current work and other lifestyles. Furthermore, COVID and other restrictions prevented travelling and meeting experts in person to get a more concrete opinion. After few discussions verbally with my own colleagues it was decided to limit it to two questions and go for an rank based answer style to save time and increase engagement.

C. Development and Validation

After the feedback is collected, a development will be performed to target different scenarios based on information gathered. Furthermore different other research papers will be read and an attempt will be made to solve what other authors have noted as code issues. A validation process needs to be run in order to verify the results of the development which will be explained further down and finally a conclusion will be given based on the findings of the paper and if the new proposed software is successful in enhancing developer productivity.

III. Literature Review

A. Open-source Software in Class: Students' Common Mistakes

Authors Zhewei, Yang and Edward have written a highly detailed paper on common source code mistakes.[1] Here they target mainly open source software where they take a live audience of over 700 students and have manually studied the students course projects. They further have summarised it down to thirteen common mistakes they have found and have provided suggestions to overcome these issues and improve code quality. The biggest common mistake they have found is in code comments, students are not using proper code comments in their code which leads to unreadable code and difficulty to understand.

However one could argue that too much code comments would make the code more unreadable as the length of the code will increase significantly. Code comments should therefore be used quite sparse and when absolutely necessary. The authors have also stated that not using switch statements are also a common issue they see which leads to lengthy code which results software developers being unproductive.

The authors have provided suggestions to improve this code and in one suggestion they state that using third party code analyser tools would be helpful for these students. However, a third party tool with such capabilities does not exist yet, and the program presented in this paper has the ability to fix if else statements by notifying the developer to change to switch-case.

B. Twenty-Five Most Common Mistakes with Real-Time Software Development

David B Stewart, a university professor has worked in the software industry for a long period of time and as a chief technical officer as well.[2] He lists down 25 most common issues he has noted during his career and helps the reader to understand what are the most important issues. In an example he provides, he states that he notices students writing large if - else statements when they can use the ternary operator and reduce four lines to one single line. This issue has been highlighted by the experts in the industry when collecting feedback to build the static code analyser.

The author continues to elaborate on very common and simple mistakes such as attitude issues to students not willing to accept a different answer, writing large unnecessary loops and sometimes not even understanding what the root cause is. A overall very well written paper and the reader can get a good idea on updated software issues in the industry.

C. Impact of Programming Features on Code Readability

Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi and Maryan Yatim are the authors of the above paper where they discuss about code readability.[4] They too find readability having a connection to re-usability, maintainability, reliability, complexity, and portability metrics. In their research they follow a similar approach where they approach experts and ask them to rank a source code on readability and classify different factors they find into positive and negative points. After their live tests were conducted, they came to the conclusion that they noticed factors that improve readability are:

- Having meaningful function names
- Using comments

Where as the below were found to reduce readability

- Using arithmetic formulas
- Nested loops
- Recursive functions

The authors note that using switch statements, if-else statements or for loops did not have a major impact on readability however one could argue that these do impact readability if not used correctly in the right scenario. For an example, using if statements when a ternary operator could be used is an issue noted by David B Stewart in his paper above.

D. An Empirical Study Assessing Source Code Readability in Comprehension

Authors John C. Johnson, Sergio Lubo and others have performed a study on source code and how should be written in a minimized way to make it easier for others to read and understand to improve productivity.[5] They conduct a controlled test to check and identify loops and nested loops to get an idea if they impact readability or not. The test was conducted with over 200 participants and reading through different source code material. The authors have come to a conclusion after the study that with reduction of nested loops it does improve the readability score and the understanding confidence of a developer, they also conclude that do-while statements do not have a significant impact on source code readability. A unexpected outcome out of the research was that the authors found out that better knowledge of the English language in a participant is also a factor in improving source code issues.

E. Code Readability Testing, an Empirical Study

Author Todd Sedano conducts a study on how much readability impacts software maintainability and understanding.[6] He argues that it's quite difficult for subsequent developers to read and understand and they are likely to introduce more bugs, extend the codes older and original functionality and create more issues than before. It can be agreed with this statement as the author provides enough evidence to his claim and other authors too have agreed with the notion that readability will have an impact on software development

The author conducts a study with 21 masters students and follow code readability testing in four different sessions. Then a questionnaire was provided to evaluate the students perspectives. The author concludes that by performing code readability testing they were able to improve the source code readability score and the students too had an easy understanding of the new code. Motivation was boosted among the students to continue writing readable code. This is a subtle but a very successful approach in helping students improve their skills and talents. When put in the correct path the students will follow through the things they learnt and apply them well in the future. However they might forget such rules and standards in the future, which is why a tool would help catch such mistakes and help them revise their knowledge.

IV. Research Methodology

A. The Questionnaire

Now finalized, the questionnaire consists of a few simple questions which have been designed by reading different other research papers and gaining knowledge from them to what are the most common issues found in developers. The answers can be weighted as it will be required to the candidates to rank the given answers in the order that they have noticed is the most frequent. As an example in the paper[2] it was noted by the author that using multiple if else if statements will lead to longer code and make it unreadable in some scenarios. For this the author has suggested using switch case as an alternative to better developer practise as it enhances the code readability and shortens the code. Using the knowledge gained from the related work and discussions with colleagues the following questions were sent out to the candidates using a questionnaire made by Google Forms.

The first question asked is as follows - Rank the most common seen issues in code review process.

1) Using nested for loops (performance impact)
2) Using multiple if else if statements (lengthy code)
3) Using single if else statement (ternary operator can be used)
4) Using while loop with counter (could use for loop)
5) Indentation issues
6) Unused imports
7) Comments (complicated code with less explanation)

On the second question, candidates were asked to select the most used language they see these issues in new employees and when they are mentoring new employees.

Once the averages are calculated, the common consensus will be taken to the top three scenarios and building static code analysers to the top three requested languages. One of the main goals of this research would be to create static code analysers from different languages and govern it all using some sort of user interface, with the intention being to make static analysis a worry free, setup free, user friendly application accessible to all sorts of developers.

As an additional aspect to this research to increase the depth and validity, different research papers have been thoroughly read to understand common code issues in the modern world. In the literature review section many papers published by different authors have been analysed to the issues they have seen and found in developers and this research aims to cater some of these hand picked scenarios as well to build rule sets.

After a thorough research it was decided to use Abstract Syntax Trees to build the analyser. AST's(Abstract Syntax Trees) are very powerful tools, they can identify and change source code at will. However they are rather used quite infrequently. This research also aims to improve the knowledge of the reader through by showing the powerful tool set of AST's for natural language processing.

B. Developing the Static Code Analyser

In analysing the information received for the second question, out of 25 candidates the top three languages taken out were Java, Python and C# respectively. For this research due to the time restriction, it was decided to build a static code analyser for python as the first step and then carry out building static code analysers for other languages in the future. Python was selected over java due to it's friendly nature towards machine learning and ready availability to building a static code analysers.

Python has inbuilt libraries which support AST's and could be used to develop the rule set, build and parse trees. Java on the other hand needs third party libraries in order to scan and build a syntax tree. This research will be using pythons inbuilt library .AST for creating and parsing through code given.

Abstract Syntax Tree is a kind of a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. Being the fundamental way of how code works, AST's can sketch the code which is fed to them into a tree and each node will match the depth of the source code entered.[8] AST's work in 3 ways which are

- Lexical analysis - In this step, code which is written and submitted will be converted into a set of tokens. The identifiers, literals, punctuators are all categorised and made into code tokens.
- Syntax analysis - The tokens are now made into a AST. The tokens are now categorised into different types of statements such as expression statements, programs, call expressions etc.
- Code generation - Here we can use the AST to print the tree, generate the new code or parse down the tree to manipulate code or alter it. It is in this final step that this

research focuses as the use of the AST is to scan entered code, identify mistakes and point them out to the user.

Using AST's, it was possible to read and understand the code submitted through a file, process it, generate a tree and give the output as required for the developer to tune their work and make their code more efficient. A python program was written using the knowledge on AST's to write the set of rules and print statements if the rules are matched in any scenario. The scenarios covered in this python program were picked from the questionnaire and other research papers.

In analysing the results received by the quiz which catered over 25 candidates,

The top three results from the survey were the below issues in the collected feedback. It is important to consider these scenarios through the static code analyser as this is what the majority in the industry demand.

1) Using multiple if else statements (lengthy code)
2) Using single if else statement (ternary operator can be used)
3) Unused imports

Furthermore authors of paper[1] too state that not using switch case is a very common mistake seen in a set of developers where an experiment was conducted to find common issues in source code. Therefore we can arrive at the conclusion that the majority of the experts in the software industry and research paper authors agree upon that switch case statements should be normalised over lengthy if else statements and if else statements are one of the main reasons for lengthy source code. Using the above information, the python code was written to cover the below scenarios which are :

1) Using nested for loops (performance impact) - Research paper
2) Using multiple if else if statements (lengthy code) - Research paper
3) Using single if else statement (ternary operator can be used) - Questionnaire
4) Using while loop with counter (could use for loop) - Questionnaire
5) Unused imports - Questionnaire

The program was written with five different classes and one main class to execute all the code at once. Each class contains a rule set and analysis part to detect if a certain scenario is found. If found, the program will alert the user with a statement giving some information with the line number of where the issue was detected with the message and file name.

The code has been carefully written and tested, to make sure the explicit scenarios are found to reduce chances of a mismatch. As an example for a good hit, in the scenario where the user would be alerted to use the ternary operator, it was a necessary fact to make sure that only one line of code exists in both the if and else statements or else it is not possible to use the ternary operator. Once unit testing was completed the next step was to find and present the analyser to a set of experts for testing.

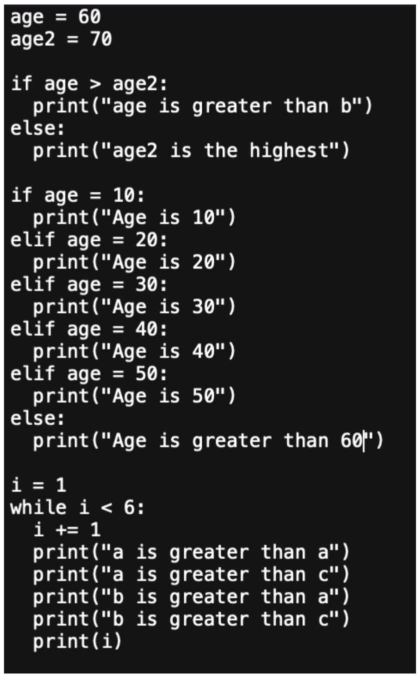
V. Testing & Evaluation

Once the program was successfully self tested and verified it was necessary to validate the findings and if the program did actually improve the quality of code and eventually improve productivity of developers. It was decided to perform this in two methods,

1) Send the sample code to selected set of experts and ask them to identify issues they notice and rank the readability
2) Send the corrected code by using suggestions from the program to the experts and ask them to re-score the readability

In the first step, it was necessary to reduce the sample size and keep things simple for reasons mentioned before. Around 5 experts were hand picked and met in person for collecting information. The sample picked were experts with over 5 years of experience in the software industry (own senior / lead colleagues) and the sample code was sent to them to identify issues seen by them. The sample code consisted of quite a number of issues which consist of issues that can be suggested from the program

A code snippet of the sample code is seen below, which is taken from websites and personal experience with new developers. It contains of a simple if statement, a badly used while loop and has some nested for loops as well. It was needed to not make it too lengthy as the experts time spent on this was a valuable time and should not be taken for granted. The sample code was mixed with issues which can be predicted from the program however it did consist of other issues as well which cannot be predicted by the program yet.


```
age = 60
age2 = 70

if age > age2:
  print("age is greater than b")
else:
  print("age2 is the highest")

if age = 10:
  print("Age is 10")
elif age = 20:
  print("Age is 20")
elif age = 30:
  print("Age is 30")
elif age = 40:
  print("Age is 40")
elif age = 50:
  print("Age is 50")
else:
  print("Age is greater than 60")

i = 1
while i < 6:
  i += 1
  print("a is greater than a")
  print("a is greater than c")
  print("b is greater than a")
  print("b is greater than c")
  print(i)
```

Fig. 1. Sample bad code screenshot

Before development, a mini oral interview was conducted with the experts. The experts common opinion when asked about the static analyser was that it is a good approach towards correcting common mistakes seen throughout their daily life. They noted that common mistakes are made ranging from new comers to even heavy experienced veterans in the industry, so a static analyser with a good rule set would cater their needs and save time needed to perform code reviews. The experts requested to only provide suggestions through the analyser as if the analyser were to fix issues as well, developers will not learn and might take the application for granted. This will not entice healthy learning and the developers will be more lazy expecting software to fix their mistakes. When corrections are suggested developers will have to put in the effort to carefully understand the mistake they have done and correct it. As self learning is the best way to learn the probability of making the mistake is thus reduced.

Then the experts were then asked to asked to identify issues seen and provide feedback which was collected and analysed. The results are given below in a summarised format. As for the scale used, slider scales and likert scales were compared and finally slider scales were used to get a more finely grained approach with the results since each point helps in contributing to the average more finely. Likert scales somewhat take a better response for questions that cannot be answered with a fine score so it was not suitable for this situation.

The feedback and the comments has been summarized below and given in the table along with the readability score

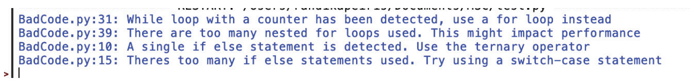
TABLE I
OLD FEEDBACK TABLE

Senior 1	Senior 2	Senior 3	Lead 1	Lead 2
6/10	5/10	6/10	6/10	4/10

- Senior 1- Nested loops usage is too much, for loops may impact on performance. Code is lengthy, but can be read due to proper indentation
- Senior 2- Use switch case for the if statements. Can use ternary operator for the single if else statement.
- Senior 3- For loops usage is too much, use switch statements instead of if else
- Lead 1- If statements need to be switched out with switch case, label the for loops for better definitions, the while loop with counter is unnecessary and can be switched with for loop
- Lead 2- Can use switch case instead of if else, easy to read due to indentation but this is because of python language

With the above results and an average of 54% out of 100, the results are on low average and it proves that the code does not match the standards the experts or industry seeks and its not a well written code. As through past work we can easily come to the conclusion that productivity is lowered with low readability, we can further add that if code is not comprise and not efficient, it will lead to another developer to spend a longer time reading and understanding the code which will impact productivity in a negative manner.

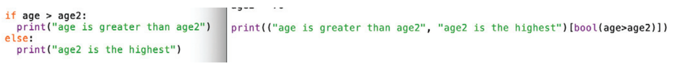
Now with these results, the python program will be run over the bad sample code and it worked as expected. The program was able to identify scenarios met and identified by the experts. A screenshot of the hits seen can be seen below



```
BadCode.py:31: While loop with a counter has been detected, use a for loop instead
BadCode.py:39: There are too many nested for loops used. This might impact performance
BadCode.py:10: A single if else statement is detected. Use the ternary operator
BadCode.py:15: Theres too many if else statements used. Try using a switch—case statement
>|
```

Fig. 2. Analyser Findings

After using the suggestions from the analyser to improve and fix the code, it was clear that the length of the code too did reduce. As an example we can see that the previous if condition with four lines has been reduced to one single line.



```
if age > age2:
  print("age is greater than age2")
else:
  print("age2 is the highest")
```

```
print(("age is greater than age2", "age2 is the highest")[bool(age>age2)])
```

Fig. 3. Old vs New code

Next, it was necessary to validate through expert opinions and get feedback if the measures calculated had been improved. For this the new code was again taken to the same set of experts and asked to rank in a slider scale. The new feedback scores are given below with the experts opinion in a summary.

TABLE II
NEW FEEDBACK TABLE

Senior 1	Senior 2	Senior 3	Lead 1	Lead 2
8/10	8/10	7/10	7/10	8/10

- Senior 1- Good improvement over the original code!
- Senior 2- Code quality has improved. Still some other aspects to improve.
- Senior 3- Very good improvement, still has room to improve
- Lead 1- A good improvement can be seen with compared with the original code. However switching if conditions with ternary operator might lengthen code horizontally.
- Lead 2- There is a very noticeable improvement over the original code. Very good job done by the analyser.

With the average now at 76% and almost a 25% increase in the scores, we can clearly see the new program has successfully improved the old source code by quite a number. Feedback was taken as well on other factors that could be improved and majority of the feedback was due to the situation which were not covered by the program such as indentation issues and tab spaces and one issue highlighted was that switching out a lengthy string in an if condition to a ternary operator might vertically lengthen the code reducing readability. Therefore an enhancement will have to be made to check the string length as well when asking the user to switch to ternary operator.

Now that the code's readability has improved, the results are clear that the developers productivity has been improved, and will also enhance the readability of transferring the source code to another developer. Furthermore, since the program lessens the line amount of the code, the compilation time is quicker by milliseconds as confirmed in another verification test.

In order to conduct a verification test, the code was run using python's time library to compute how much time is taken to compile and execute the code. The above is done by starting a timer at the beginning of the code and then calculate the time taken to compile the code is taken and reduced after all the code is executed. In the old code with issues the compilation time was roughly 10 seconds (taken by an average of executing the code 5 times) and in the improved version of the code it was roughly around 9 seconds, an improvement of 1 second.

Although this may not seem as a massive gain through performance, in the industry where thousands of lines of code are written in day to day basis the time and performance gain can be exponential. However, this compilation improvement might be due to other uncontrollable factors such as current RAM and CPU usage, therefore it's difficult to put a number on the percentage of improvement. Tests could be conducted in a more tightly controlled environment to ensure accuracy.

VI. Future Work

As of now the program can successfully detect a few tailored scenarios based on the industry and research papers. However the issues one can research on in the software industry is limitless. For the next scenarios to improve this algorithm, issues such as tightly coupled code, collusion could be considered. Furthermore, Artificial Intelligence and/or Machine Learning based learning could be integrated to enhance the value of this software by sniffing issues using prediction based learning. More research needs to be done on how to use machine learning or AI algorithms to create rule sets, but this will take a much longer time.

Another aspect currently in development is a Web UI to host the analyzers. Once work on the initial top requested analyzers are completed, an angular based web application is proposed to be developed to expose the algorithm to the public. Users will be able to select the language needed through a combo box and they will have the ability to copy paste code or else drag and drop source code files in to scan for issues using their preferred language. There are also few suggestions from the experts to automatically select the language based on the file inputted so that it will be further convenient to the user.

VII. Conclusion

With the results in hand it is clear that a static code analyser tailored to the industry requirements can contribute in improving source code quality, productivity and ultimately revenue. With improvements and more development, this analyzer could be a game changer in minimizing code review process and improving developer health conditions. Quality code should be a fundamental must in today's software industry and we as developers, should always keep good code standards in our daily developments.

References

[1] Hu, Z., Song, Y., Gehringer, E. F. (2018, May). "Open-source software in class: students' common mistakes." In Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (pp. 40-48)..

[2] Stewart, D. B. "Twenty-five most common mistakes with real-time software development." In Proceedings of the 1999 Embedded Systems Conference (ESC'99), vol. 141. 1999.

[3] Barbosa, L. F., Pinto, V. H., de Souza, A. L. O. T., Pinto, G. "To What Extent Cognitive-Driven Development Improves Code Readability?." In ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 238-248. 2022.

[4] Tashtoush, Y., Odat, Z., Alsmadi, I. M., Yatim, M. (2013). "Impact of programming features on code readability".

[5] Johnson, J., Lubo, S., Yedla, N., Aponte, J., Sharif, B. . "An empirical study assessing source code readability in comprehension." In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), (pp. 513-523). IEEE, 2019.

[6] Sedano, T. "Code readability testing, an empirical study." In 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET), (pp. 111-117). IEEE, 2016.

[7] Bardas, A. G. (2010). "Static code analysis". Journal of Information Systems Operations Management, 4(2), 99-107.

[8] Neamtiu, I., Foster, J. S., Hicks, M. "Understanding source code evolution using abstract syntax tree matching." In Proceedings of the 2005 international workshop on Mining software repositories, (pp. 1-5). 2005.