Journal of Software Engineering, Information and Communication Technology (SEICT)

Journal homepage: https://ejournal.upi.edu/index.php/SEICT

Comparison Towards Different Methods of Software Debugging

Akwan Cakra Tajimalela[1], Ramandha Putra Suryahadi[2], Rizka Alfadilla[3], Indira Syawanodya[4], Iqbal Tawakal[5]

[1,2,3,4] Software Engineering Study Program, Universitas Pendidikan Indonesia, Indonesia

Correspondence: E-mail: akwancak20@upi.edu

ABSTRACT

To ensure that software works as expected, debugging is essential. This article discusses several types of debugging, focusing on Visual Studio Code Debugger, print statement debugging, and static code analysis. Each method is demonstrated through a simple yet comprehensive program that determines whether a selected number is a prime number. We conclude that the Visual Studio Code Debugger offers powerful features but requires prior experience with Visual Studio Code. Print statement debugging is straightforward but can become cumbersome in more complex programs. Static code analysis, particularly with ESLint, does not directly uncover "bugs" in the traditional sense but evaluates code style, security, and good programming practices. Combining these three methods can yield optimal results in the debugging and inspection process, depending on the specific needs and context of the project.

ARTICLE INFO

Article History:
Submitted/Received 04 June 2024
First Revised 06 November 2024
Accepted 12 December 2024
First Available online 12 December 2024
Publication Date 12 December 2024

Keyword:
Software,
Debugging,
Visual Studio Code,
Programming,
Coding.

1. INTRODUCTION

Debugging and inspection are two crucial and inseparable components in software development. The importance of these activities cannot be overstated, as they significantly impact the quality and reliability of software products. Testing and debugging are essential phases in the software development lifecycle, but they are also the most expensive and exhausting phases (Campos, et al., 2012). Moreover, in software development, we need to clarify where inspections are essential because the more inspections are conducted, the less burden debugging places on testing operations, thereby increasing the efficiency and quality of software development (Fargan, 1999). In other words, the more effective the inspections, the less time and effort required for debugging, which significantly improves the overall efficiency and quality of software development.

However, if there are errors or bugs that escape the inspection phase, it is expected that these errors will be found and corrected during subsequent testing. Bugs, or errors in software, can have very detrimental effects on quality and reliability (Wahyuningsih, 2023). These issues can cause a range of problems, from minor glitches and performance hindrances to severe system crashes and security vulnerabilities. Consequently, the presence of bugs can increase maintenance costs, extend development time, and damage the reputation of the software provider (Azhar & Rochimah, 2016). Therefore, if bugs escape inspection, efficient debugging practices are crucial to mitigate these risks and ensure the delivery of reliable and accurate software solutions.

The importance of debugging and inspection lies in their ability to ensure that the software meets the required quality and reliability standards. From the outset, we need to implement effective debugging techniques to identify and fix bugs early on, thus preventing negative impacts on software performance and user experience (Wahid, et al., 2015). Therefore, evaluating the effectiveness of various debugging techniques becomes an interesting endeavor. This evaluation is important to optimize the process of identifying and fixing bugs. Various debugging tools and strategies, such as Visual Studio Code Debugger, Print statement debugging, and Static code analysis, each offer different advantages and challenges.

This article will present a comparative analysis of these various debugging tools and strategies to highlight the strengths and weaknesses of each in the context of debugging. By understanding the pros and cons of each tool and strategy, developers can make better decisions about which tools and strategies are most appropriate for different scenarios, ultimately enhancing debugging efficiency and improving overall software quality.

2. LITERATURE REVIEW

2.1. Debugging

Debugging is crucial for preventing software functionality issues. It is the essential process of identifying and fixing errors or defects in a software system to ensure its proper functioning (Strauss, 2023; Srivastva & Dhir, 2017). The goal is to ensure that the program or system operates as expected. However, debugging is not just about fixing code; it is also about understanding why bugs occur and finding ways to prevent them in the future. The debugging process typically involves six steps: reproduce the conditions, find the bug, determine the root cause, fix the bug, test to validate the fix, and document the process (IBM, 2023).

2.2. Inspection

Software inspection is a crucial process in software engineering that involves reviews by trained peers to identify defects in work products using a specified methodology (Anderson

& Teitelbaum, 2001). This inspection aims to reduce the expected costs of software failures throughout the product lifecycle. The goal of software inspection is to embed quality into the software product from the beginning, rather than addressing quality issues later in the development cycle, as the cost of fixing defects increases significantly when detected later. Various methods and tools are used in software inspections, such as analyzing output files to ensure consistency in requirements, design, testing, and source code, generating traceability matrices, and creating reports on inspection results (Laitenberger, 2002). Additionally, advancements like software inspection tools for remote diagnostics and unmanned vehicle control contribute to simplifying the maintenance and installation processes in software inspections.

2.3. Bugs in Software

Bugs can be found in various parts of software, from source code to system design, and can affect multiple operational aspects of the software. A bug is a system error that causes a mismatch between user expectations and actual results (Liang & Hartanto, 2022). Bugs typically arise from programming errors, and resolving them quickly is essential to avoid disrupting users' business processes. Even after the software is completed, bugs are often present (Hadiprakoso, 2020). Managing bugs in software requires a systematic and organized approach. This process involves early detection, accurate classification, prompt resolution, and effective prevention. The time required to fix a bug can vary depending on its complexity (Azhar & Rochimah, 2016). By implementing best practices in software development and rigorous testing, the number and impact of bugs can be minimized, thereby enhancing the overall quality and reliability of the software.

2.4. Types of Bugs in Software

Bugs in software can be categorized into various types based on their characteristics and impact on the software. Here are explanations of some commonly encountered types of bugs:

1) Functional Bug

This type of bug affects the functionality of the software (Whittaker, 2012). An example is when a feature does not function as intended or produces incorrect output. Functional bugs can cause failures in the software's main operations and are often a top priority for fixing.

2) Logic Bug

Logic bugs occur when there is an error in the programming logic. This means that the software may run without technical errors, but the results do not match expectations due to incorrect algorithm or logic decisions (Martin, 2008). Examples include calculation errors or incorrect conditions in if-else statements.

3) Syntax Bug

Syntax bugs occur due to errors in code writing that violate the syntax rules of the programming language used (Kernighan & Pike, 1999). Examples include writing errors in variables, missing parentheses, or disallowed characters. These bugs are usually easy to find and fix by the compiler or interpreter.

2.5. Debugging Tools and Strategies

Debugging has several tools and strategies to choose from. Many variations of strategy are used in order to gain the best result with the best approach. While VSCode Debugger, IntelliJ IDEA, and any other debugging tools are meant to help software developers in the debugging processes. Here are some of the tools and strategies for debugging.

A. **Tools**

   Visual Studio Code, Chrome DevTools, ReSharper, PyCharm Debugger, Xcode, Android Studio, dbForge SQL Tools, Telerik Fiddler, Eclipse Debugging Tools, WinDbg (Windows Debugger), intelliJ IDEA, Sauce Labs

B. **Strategies**

   Brute force method, rubber duck debugging, bug clustering, cause elimination method, backtracking, program slicing, binary searching, static analysis

1) Visual Studio Code Debugger

The Visual Studio Code Debugger is a built-in debugger in Visual Studio Code that provides various debugging features such as breakpoints, step in/over/out, watch expressions, and more. Users can use this Debugger to track the flow of the program, analyze variable values, and detect bugs (Paul Ballard).

2) Print Statement Debugging

Print Statement Debugging is a debugging method that involves adding print statements to the code to print variable values or specific points in the program flow. This helps developers track the execution flow of the program and check variable values (McConnell, 2004).

3) Static Code Analysis

Static Code Analysis is a method that involves using tools or services to analyze code statically without having to execute it. These tools check the code for common errors, code style, and security issues (Martin, 2008).

3. METHODS

While there are numerous debugging processes, here we have selected some of the most common methods and compared them. In this study, our aim is to comprehensively evaluate these selected debugging methods, analyzing their effectiveness, efficiency, and practicality in identifying and resolving software bugs. Through individual tests on each method, we aim to assess their performance in bug identification and resolution. This comprehensive examination seeks to uncover the strengths and weaknesses of each method, aiding informed decision-making when selecting the most suitable debugging approach for diverse software development scenarios.

3.1. Code to be Tested

The code used is simple enough to be understood by readers of various skill levels, yet complex enough to encompass various debugging aspects. This code uses JavaScript as its language and implements two functions: "isPrime" to check if a number is prime, and "calculateAverage" to compute the average of a list of numbers. This code contains functional and logical bugs in it, to test three different debugging methods.

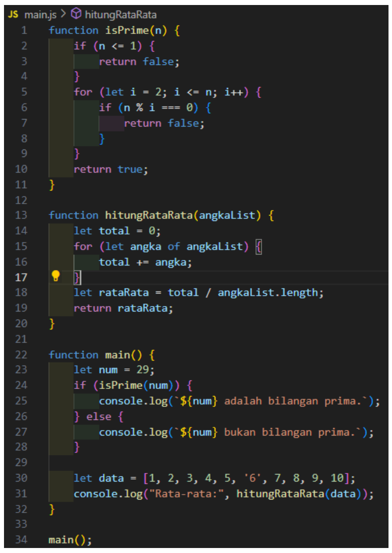

```
main.js > hitungRataRata
function isPrime(n) {
    if (n <= 1) {
        return false;
    }
    for (let i = 2; i <= n; i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return true;
}

function hitungRataRata(angkaList) {
    let total = 0;
    for (let angka of angkaList) {
        total += angka;
    }
    let rataRata = total / angkaList.length;
    return rataRata;
}

function main() {
    let num = 29;
    if (isPrime(num)) {
        console.log(`${num} adalah bilangan prima.`);
    } else {
        console.log(`${num} bukan bilangan prima.`);
    }

    let data = [1, 2, 3, 4, 5, '6', 7, 8, 9, 10];
    console.log("Rata-rata:", hitungRataRata(data));
}

main();
```

Figure 1. Test Code.

This code allows for demonstrating various common errors that may occur in everyday programming, such as logic errors in loops and data type errors in mathematical operations. It provides real examples of how debugging tools and strategies can be applied to fix these errors. These functions can also be easily tested using Visual Studio Code Debugger, print statements, and static code analysis, facilitating clear and structured comparisons between the three debugging methods.

3.2. Implementation Visual Studio Code Debugger

Add breakpoints by clicking on the left side of the line numbers in the code in Visual Studio Code. These breakpoints mark points where code execution pauses temporarily, allowing inspection of variable values and program flow at that point. Once breakpoints are set, click the green "Run" button in the top left corner of the debugging window or press "F5". This will start code execution and stop at the first encountered breakpoint.

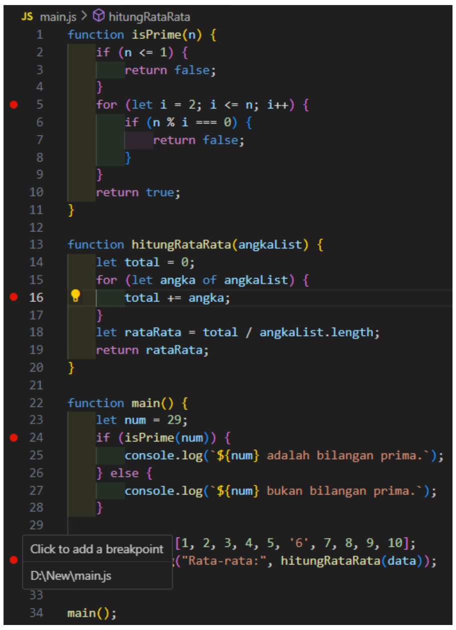
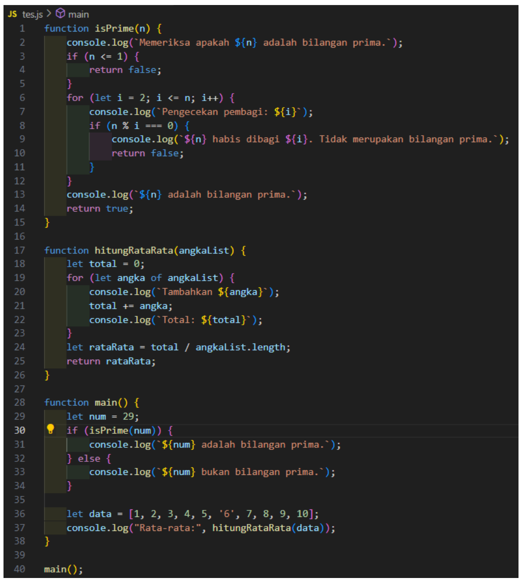

```
main.js > hitungRataRata
function isPrime(n) {
    if (n <= 1) {
        return false;
    }
    for (let i = 2; i <= n; i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return true;
}

function hitungRataRata(angkaList) {
    let total = 0;
    for (let angka of angkaList) {
        total += angka;
    }
    let rataRata = total / angkaList.length;
    return rataRata;
}

function main() {
    let num = 29;
    if (isPrime(num)) {
        console.log(`${num} adalah bilangan prima.`);
    } else {
        console.log(`${num} bukan bilangan prima.`);
    }

                [1, 2, 3, 4, 5, '6', 7, 8, 9, 10];
                ("Rata-rata:", hitungRataRata(data));

main();
```

Figure 2. Visual Studio Code Debugger Test.

When execution pauses at a breakpoint, inspect variable values by hovering over the variable or viewing the "Variables" panel in the debugging sidebar. This allows verification of whether variable values match expectations at a particular point in execution. By using "Step Over", "Step Into", and "Step Out", we can step through the code, observe execution flow, and inspect variable values at each step. This helps identify where errors occur, such as incorrect variable values or program logic.

3.3. Implementation Print Statement Debugging

Debugging using print statements is one of the simple yet effective methods for identifying errors in code. Identify critical parts of the code that need to be checked for values or execution flow. Add print statements at the identified points. These print statements will print variable values or specific messages to the console during code execution.



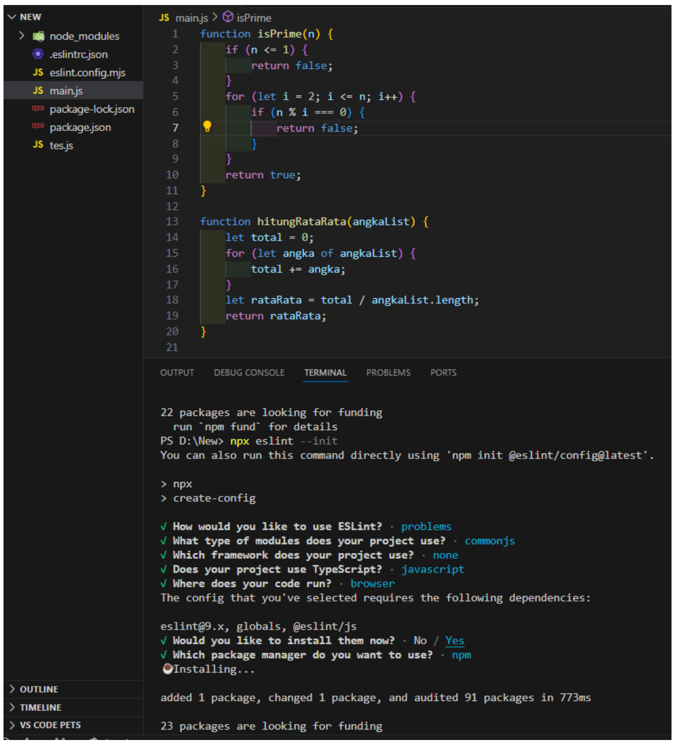
```javascript
function isPrime(n) {
    console.log(`Memeriksa apakah ${n} adalah bilangan prima.`);
    if (n <= 1) {
        return false;
    }
    for (let i = 2; i <= n; i++) {
        console.log(`Pengecekan pembagi: ${i}`);
        if (n % i === 0) {
            console.log(`${n} habis dibagi ${i}. Tidak merupakan bilangan prima.`);
            return false;
        }
    }
    console.log(`${n} adalah bilangan prima.`);
    return true;
}

function hitungRataRata(angkaList) {
    let total = 0;
    for (let angka of angkaList) {
        console.log(`Tambahkan ${angka}`);
        total += angka;
        console.log(`Total: ${total}`);
    }
    let rataRata = total / angkaList.length;
    return rataRata;
}

function main() {
    let num = 29;
    if (isPrime(num)) {
        console.log(`${num} adalah bilangan prima.`);
    } else {
        console.log(`${num} bukan bilangan prima.`);
    }

    let data = [1, 2, 3, 4, 5, '6', 7, 8, 9, 10];
    console.log("Rata-rata:", hitungRataRata(data));
}

main();
```

Figure 3. Print Statement Debugging Test.

Run the code as usual. The print statements will print the specified information to the console when execution reaches the marked points. After running the code and observing the output from print statements, analyze the results. Note the printed variable values and messages to check if they match expectations.

3.4. Implementation Static Code Analysis

The implementation of Static Code Analysis involves using tools or services that automatically check program code to identify potential errors, code style violations, or security issues. Choose the tool or service to perform static code analysis, here we use ESLint to test JavaScript programming language. Install the static code analysis tool ESLint by running the command npm install "eslint --save-dev". Configure the tool according to the project's needs. This may involve creating a configuration file such as ".eslintrc.json" for ESLint.



```javascript
function isPrime(n) {
    if (n <= 1) {
        return false;
    }
    for (let i = 2; i <= n; i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return true;
}

function hitungRataRata(angkaList) {
    let total = 0;
    for (let angka of angkaList) {
        total += angka;
    }
    let rataRata = total / angkaList.length;
    return rataRata;
}
```

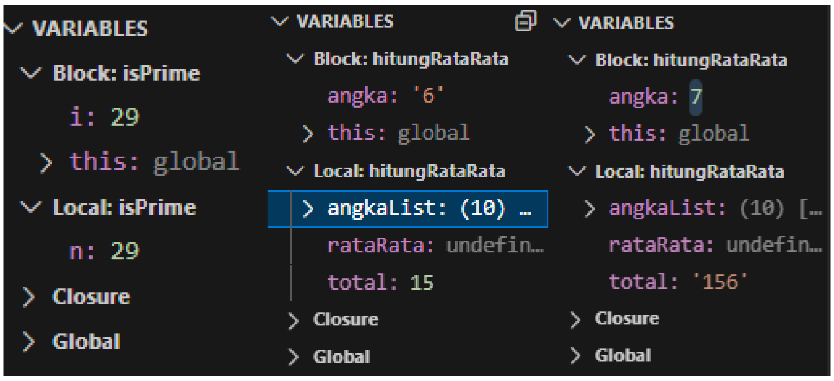
```
22 packages are looking for funding
  run `npm fund` for details
PS D:\New> npx eslint --init
You can also run this command directly using 'npm init @eslint/config@latest'.

> npx
> create-config

√ How would you like to use ESLint? · problems
√ What type of modules does your project use? · commonjs
√ Which framework does your project use? · none
√ Does your project use TypeScript? · javascript
√ Where does your code run? · browser
The config that you've selected requires the following dependencies:

eslint@9.x, globals, @eslint/js
√ Would you like to install them now? · No / Yes
√ Which package manager do you want to use? · npm
Installing...

added 1 package, changed 1 package, and audited 91 packages in 773ms

23 packages are looking for funding
```

Figure 4. Static Code Analysis Test.

Use the command or script "eslint ." to analyze all files in the current directory. The code analysis will identify potential errors, code style violations, or security issues in your code. Once the analysis is complete, check the output or report generated by the code analysis tool. Typically, the output will include a list of errors or warnings with brief descriptions of each issue found. Interpret the analysis results to understand the issues found and their locations in the code.

4. RESULTS AND DISCUSSION

As seen in **Figure 5**, the Visual Studio Code Debugger facilitates detailed examination of code execution flow, aiding in the identification of logical errors and incorrect variable values. In the "isPrime" function, it is found that the loop continues until "i = 29", causing a bug or logical error. **Figure 5** also shows that in the "calculateAverage" function, there is also an error or bug when the total of 15 is added to '6', resulting in '156'. It is evident that a data type change from integer to string has occurred.



```
VARIABLES
  Block: isPrime
    i: 29
  > this: global
  Local: isPrime
    n: 29
> Closure
> Global
```

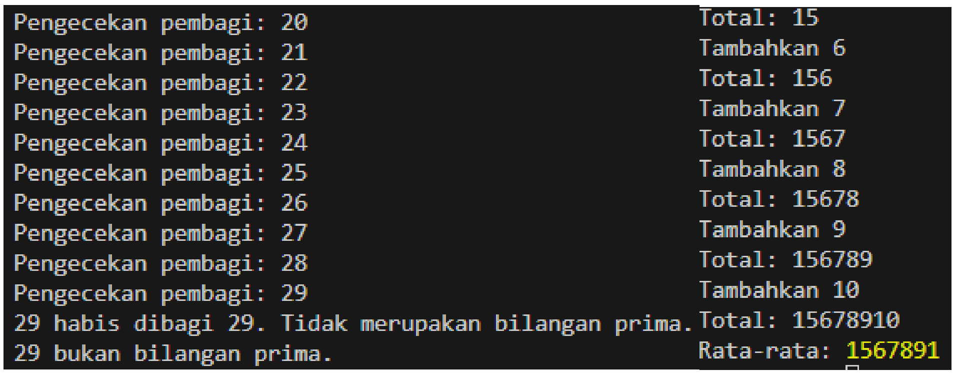
```
VARIABLES
  Block: hitungRataRata
    angka: '6'
  > this: global
  Local: hitungRataRata
  > angkaList: (10) …
    rataRata: undefin…
    total: 15
> Closure
> Global
```

```
VARIABLES
  Block: hitungRataRata
    angka: 7
  > this: global
  Local: hitungRataRata
  > angkaList: (10) […
    rataRata: undefin…
    total: '156'
> Closure
> Global
```

Figure 5. Hasil Tes Visual Studio Code Debugger.

The debugger significantly enhances debugging efficiency by providing step-by-step execution and real-time variable inspection. Developers can quickly navigate through the code, identify anomalies, and fix issues rapidly, reducing the time spent fixing bugs. However, its practicality may be influenced by the developer's familiarity with the debugging environment. Proficient Visual Studio Code users find this method very practical, but those less accustomed may require additional learning time.

By strategically placing print statements, developers can monitor variable values and program flow, aiding in error detection such as incorrect variable assignments or unexpected program behaviour. From **Figure 6**, we can see that the checking process stops at the divisor 29, whereas it should stop at the previous divisor. This indicates a bug in the "isPrime" function. The bug is that the for loop should stop when it reaches a value greater than the square root of n. Figure 6 also shows that the "calculateAverage" function also encounters a bug with a variable whose value disrupts subsequent calculations.

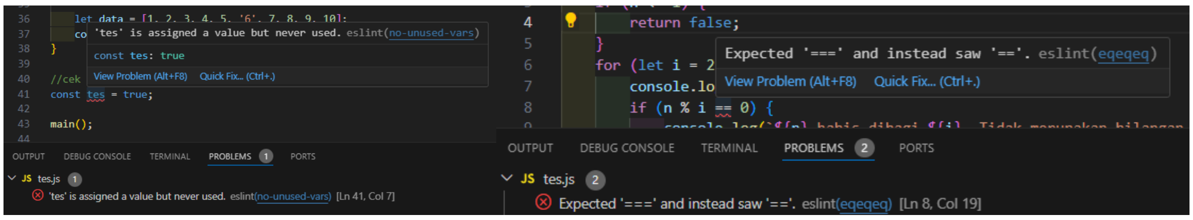


```
Pengecekan pembagi: 20
Pengecekan pembagi: 21
Pengecekan pembagi: 22
Pengecekan pembagi: 23
Pengecekan pembagi: 24
Pengecekan pembagi: 25
Pengecekan pembagi: 26
Pengecekan pembagi: 27
Pengecekan pembagi: 28
Pengecekan pembagi: 29
29 habis dibagi 29. Tidak merupakan bilangan prima.
29 bukan bilangan prima.
```

```
Total: 15
Tambahkan 6
Total: 156
Tambahkan 7
Total: 1567
Tambahkan 8
Total: 15678
Tambahkan 9
Total: 156789
Tambahkan 10
Total: 15678910
Rata-rata: 1567891
```

Figure 6. Hasil Tes Print Statement

Although debugging with print statements is easy to implement, its efficiency can be affected by the need for manual code modifications and re-execution to observe output. Compared to debugging using specialized tools like Visual Studio Code Debugger, debugging with print statements may be less efficient for complex or extensive code. Its practicality diminishes in larger projects where extensive print statements make the code difficult to read.



```
    let data = [1, 2, 3, 4, 5, '6', 7, 8, 9, 10];
    co
}

//cek
const tes = true;

main();
```

'tes' is assigned a value but never used. eslint(no-unused-vars)
const tes: true
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

OUTPUT DEBUG CONSOLE TERMINAL PROBLEMS 1 PORTS
JS tes.js 1
'tes' is assigned a value but never used. eslint(no-unused-vars) [Ln 41, Col 7]

```
        return false;
}
for (let i = 2
    console.lo
    if (n % i == 0) {
```

Expected '===' and instead saw '=='. eslint(eqeqeq)
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

OUTPUT DEBUG CONSOLE TERMINAL PROBLEMS 2 PORTS
JS tes.js 2
Expected '===' and instead saw '=='. eslint(eqeqeq) [Ln 8, Col 19]

Figure 7. Eksperimen Tes Static Code Analysis Test

Static code analysis using ESLint on the provided test code does not directly uncover "bugs" in the traditional sense, as ESLint focuses on evaluating code style, security, and good programming practices, rather than specific logic errors that can cause undesired behavior. Static code analysis using ESLint requires more time due to the initial configuration process required before analysis can be performed. Although the initial ESLint configuration can be time-consuming, this investment is usually worthwhile for its long-term benefits. While ESLint does not detect the two bugs intentionally placed in the test code, further experimentation is conducted by creating an unused variable and changing the comparison operator.

The results, as seen in **Figure 7**, show that ESLint detects errors in both changes. First, ESLint warns of an unused variable, which is just clutter. Then, ESLint warns that using "=="

can lead to unexpected results because JavaScript will automatically perform type conversion to match values. Although ESLint may not find bugs in the traditional sense in this code, its use remains beneficial for improving code quality and consistency, as well as preventing potential issues or vulnerabilities that may occur in the future.

Overall, the choice of debugging method should consider factors such as code complexity, developer familiarity, and project needs. While each method has its advantages and disadvantages, combining several approaches can provide a comprehensive debugging strategy, maximizing effectiveness in identifying and resolving bugs in software development projects.

5. CONCLUSION

In conclusion, debugging and inspection are integral components of the software development lifecycle, significantly impacting the quality and reliability of software products. The comparative analysis of various debugging tools and strategies in this study highlights the diverse approaches available to developers for identifying and resolving software bugs. The Visual Studio Code Debugger, Print Statement Debugging, and Static Code Analysis each present unique advantages and challenges, making them suitable for different scenarios. The Visual Studio Code Debugger offers a robust environment for detailed code examination, enabling developers to trace execution flow and inspect variable states effectively. This method is highly efficient for identifying logical errors and provides a powerful toolset for step-by-step debugging, although it may require a learning curve for those unfamiliar with the environment.

Print Statement Debugging, while straightforward and easy to implement, offers a more manual approach to debugging. It is particularly useful for quickly checking variable values and program flow but can become cumbersome in large codebases due to the need for extensive print statements and potential cluttering of the code. Static Code Analysis, exemplified by tools like ESLint, focuses on maintaining code quality and adherence to best practices rather than directly identifying logic errors. While it may not catch specific bugs in logic, it is invaluable for detecting code style issues, potential security vulnerabilities, and maintaining overall code health. The initial configuration effort is offset by the long-term benefits of consistent and high-quality code.

Overall, the study underscores the importance of selecting appropriate debugging strategies based on the specific requirements of a project. A combination of these methods can provide a comprehensive approach to debugging, leveraging the strengths of each to enhance software development efficiency and produce reliable, high-quality software. By understanding the strengths and weaknesses of each tool and strategy, developers can make informed decisions, ultimately leading to more effective and efficient debugging processes.

6. AUTHORS' NOTE

The authors state that there are no conflicts of interest related to the publication of this article. The authors also confirm that the paper is free from plagiarism.

7. REFERENCES

Anderson, P., and Teitelbaum, T. (2001, July). Software inspection using codesurfer. In Workshop on Inspection in Software Engineering (CAV 2001).

Azhar, N. F., and Rochimah, S. (2016). Memprediksi Waktu Memperbaiki Bug dari Laporan Bug Menggunakan Klasifikasi Random Forest. Jurnal Sistem dan Informatika (JSI), 11(1), 156-164.

Campos, J., Riboira, A., Perez, A., and Abreu, R. (2012, September). Gzoltar: an eclipse plug-in for testing and debugging. In Proceedings of the 27th IEEE/ACM international conference on automated software engineering (pp. 378-381).

Fagan, M. E. (1999). Design and code inspections to reduce errors in program development. IBM Systems Journal, 38(2.3), 258-287.

Hadiprakoso, R. B. (2020). Rekayasa Perangkat Lunak. Rbh.

Kernighan, B. W., and Pike, R. (1999). The Practice of Programming (Revised Edition). Addison-Wesley.

Laitenberger, O. (2002). A survey of software inspection technologies. In Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies (pp. 517-555).

Liang, S., and Hartanto, Y. (2022). Implementasi Bug Tracking System dengan Metodologi Scrum dan Algoritma Cosine Similarity. JURIKOM (Jurnal Riset Komputer), 9(1), 24-32.

Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

McConnell, S. (2004). Code Complete (2nd Edition). Microsoft Press.

Srivastva, S., and Dhir, S. (2017, April). Debugging approaches on various software processing levels. In 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA) (Vol. 2, pp. 302-306). IEEE.

Strauss, D. (2023). Debugging Your Code. In: Getting Started with Visual Studio 2022. Apress, Berkeley, CA.

Wahid, N. A., Sarwo, S., and Setiawan, A. W. (2015). Implementasi Dan Analisa Jaringan Saraf Tiruan Dengan Feature Normalization Dan Principal Component Analysis Untuk Digit Classifier. Petir, 8(1), 521808.

Wahyuningsih, S. S. (2023). Identifikasi Atribut Tingkat Lebih Tinggi untuk Prediksi Umur Bug. Jurnal Kolaboratif Sains, 6(3), 164-180.

What Is Debugging? | IBM. (2023, August). Ibm.com.

Whittaker, J. (2012). How Google Tests Software. Addison-Wesley.