electronics

MDPI

Article

Mutation-Based Approach to Supporting Human–Machine Pair Inspection

Yujun Dai [1], Shaoying Liu [1,*] and Haiyi Liu [1,2]

[1] Graduate School of Advanced Science and Engineering, Hiroshima University, Higashihiroshima 739-8511, Japan; d201609@hiroshima-u.ac.jp (Y.D.); liuhaiyi@jssnu.edu.cn (H.L.)
[2] School of Computer Engineering, Jiangsu Second Normal University, Nanjing 211200, China
* Correspondence: sliu@hiroshima-u.ac.jp

Abstract: Human–machine pair inspection refers to a technique that supports programmers and machines working together as a "pair" in source code inspection tasks. The machine provides guidance, while the programmer performs the inspection based on this guidance. Although programmers are often best suited to inspect their own code due to familiarity, overconfidence may lead them to overlook important details. This study introduces a novel mutation-based human–machine pair inspection method, which is designed to direct the programmer's attention to specific code components by applying targeted mutations. We assess the effectiveness of code inspections by analyzing the programmer's corrections of these mutations. Our approach involves defining mutation operators for each keyword in the program based on historical defects, developing mutation rules based on program keywords and a strategy for automatically generating mutants, and designing a code comparison strategy to quantitatively evaluate code inspection quality. Through a controlled experiment, we demonstrate the effectiveness of mutation-based human–machine pair inspection in aiding programmers during the inspection process.

Keywords: software inspection; human–machine pair inspection; software faults; software reliability; mutation testing

check for updates

Academic Editors: Jun-Ho Huh and Yeong-Seok Seo

Received: 20 December 2024
Revised: 15 January 2025
Accepted: 17 January 2025
Published: 19 January 2025

Citation: Dai, Y.; Liu, S.; Liu, H. Mutation-Based Approach to Supporting Human–Machine Pair Inspection. *Electronics* **2025**, *14*, 382. https://doi.org/10.3390/electronics14020382

1. Introduction

Code inspection is an activity in which software developers evaluate code submitted by others to improve the quality of software [1]. Fagan [2] proposed a structured process for code inspections in 1976, and subsequent research [3] and practice [4] have proven that code inspections can effectively identify potential errors and security risks in the code, thereby improving software engineering quality. Code inspections have become a crucial part of the development process, serving as a key step before integrating newly written code into the project's main codebase [5].

As computer software systems continue to grow in scale, the complexity of code structures is increasing accordingly. This trend presents two major challenges: first, complex software systems are prone to more potential errors; and second, the complexity of the code makes it challenging for reviewers to fully grasp the context of the code during inspection, leading to a decline in the quality of code inspection. In modern software companies, dedicated test teams are commonly employed to manage code inspections. While this approach introduces external perspectives and centralized expertise, it also incurs significant time and resource costs [6]. Furthermore, test teams may lack the in-depth understanding of the code context that developers possess. In agile development and DevOps workflows, code inspections are often performed by peers in the development team rather than dedicated test teams. Although automated tools are available to assist reviewers in understanding and inspecting submitted source code [7], the core responsibility of code inspection still lies with the programmer. Additionally, the effectiveness of code inspection heavily depends on the reviewer's expertise, attitude, and personality traits [8]. As companies strive to optimize the software development lifecycle, addressing these challenges has become increasingly critical.

Programmers have not only an in-depth understanding of the code that they develop but also specialized knowledge regarding the projects that they are involved in. This makes guiding them through self-inspections a promising approach to addressing existing challenges in code inspection. To this end, we proposed the human–machine pair inspection (HMPI) approach in our previous work [9], aiming to leverage both human expertise and automated tools for more efficient and effective code inspections. However, programmers' excessive confidence in the code that they write may lead them to overlook potential errors. Consequently, guiding programmers to effectively inspect their code and quantitatively assess the quality of code inspections has become a challenge worth investigating. A feasible approach to address this challenge is mutating the source code written by programmers to generate mutants and then evaluating the quality of code inspection by quantifying the extent to which programmers repair these mutants. Specifically, this study aims to address the following research questions:

- RQ1. How can an automated approach effectively attract programmers' attention during the HMPI process to mitigate the overconfidence caused by familiarity with their own code?
- RQ2. How can a mutation-based method be utilized to guide programmers in identifying potential issues in their code during the inspection process?
- RQ3. What metrics can be designed to quantitatively evaluate the effectiveness of code inspections by programmers?

At present, among the existing methods for ensuring software quality through code modification, mutation testing and software fault injection are two widely discussed approaches. Both involve injecting faults into the source code [10]. Fault injection has been widely used to evaluate fault tolerance mechanisms and to assess the impact of faults on computer systems [11]. Unlike fault injection, the purpose of mutation testing is to increase the effectiveness of test cases in identifying faults, while software fault injection is applied following other testing and verification activities (including mutation testing) in order to assess the effectiveness of fault tolerance algorithms and mechanisms. What distinguishes our approach is that we aim to guide programmers in inspecting their code through mutation (fault injection). Our research seeks to draw the attention of the inspector (who is also the programmer) to specific components of the code by strategically inserting faults during the coding process. These deliberate fault insertions aim to guide the inspector towards conducting a more thorough and focused code inspection. In this study, we propose the mutation-based HMPI approach (MB-HMPI), which effectively contributes to ensuring the correctness of programs throughout the entire development process.

In general, this study makes the following contributions:

- An automated MB-HMPI method that leverages both human expertise and automated tools is proposed to improve code inspections.
- This framework is the first to use mutants to guide programmers in code inspection, enhancing code quality and the effectiveness of inspections.
- Two novel metrics—the Mutation Detection Rate (MDR) and Code Modification Rate (CMR)—are proposed to quantitatively assess the quality of a programmer's code inspection. The MDR measures the effectiveness of identifying injected mutants, while

the CMR evaluates the extent of code modifications made by the programmer to address identified issues.

The remainder of this paper is organized as follows: Section 2 reviews the related literature, providing context for this study, and introduces the motivations and strengths of the proposed approach. Section 3 presents a detailed overview and an in-depth explanation of the MB-HMPI methodology, including its key components and implementation. Section 4 reports the case studies and experimental results, followed by a comprehensive discussion of the study's implications, limitations, and potential directions for future research. Finally, Section 5 summarizes the findings and contributions of this study, concluding the paper.

2. Background

In this section, we describe the current state of code inspection, identify challenges and limitations in existing approaches, review mutation testing and fault injection techniques, and explain the motivations and strengths of our method.

2.1. Related Work

In the early stages of code inspection, peer review involved programmers submitting their code to reviewers, who evaluated it based on their own software engineering expertise and knowledge to identify potential defects. While this process allows for the integration of external insights and experience to enhance the quality of code, it also incurs significant human resource costs. Furthermore, variations in reviewers' knowledge, skill levels, and diligence can lead to inconsistencies in the quality of code inspections. To address these challenges, much of the existing research has focused on developing static code analysis tools to assist reviewers, aiming to reduce the required manual effort while improving inspection quality [12,13]. Recent studies have explored the automation of code review processes using machine learning [14] and pre-trained models [15,16]. However, the most significant factors influencing the efficiency and effectiveness of code inspections are often the reviewer's familiarity with the code's context, their coding proficiency, and the thoroughness of their review [17,18]. As programmers are typically the most familiar with the code context, exploring mechanisms to guide them in conducting self-inspections during the coding process has emerged as a promising area for further investigation.

To the best of our knowledge, there is no research in the literature focused on techniques with the same purpose as our MB-HMPI. However, in software engineering, the technique of modifying source code based on specific rules and executing these modifications to collect runtime data remains a focal point for enhancing the security and reliability of software systems. This technique is typically combined with testing methods, with the most common applications being mutation testing [19] and fault injection [20]. Although these techniques differ from our MB-HMPI in many respects, they seem to share the similar idea of taking advantage of deliberately modified code for defect detection. Therefore, in this subsection, we focus on the review of mutation testing and fault injection techniques.

Mutation testing can generally be divided into three steps: generating reasonable mutants based on the given program, testing the mutants using test cases, and finally evaluating the effectiveness of the test suite through analyzing the test results. The step of generating reasonable mutants based on the source code is closely related to our current work. Loise et al. [21] have attempted to address security issues by utilizing a strategy for the generation of reasonable mutants. Similar studies include the work of Nanavati et al. [22], as well as its extension by Wu et al. [23], where they attempted to simulate memory faults using mutants. Garvin and Cohen [24] conducted a study on real faults in open-source projects and subsequently designed unique mutants for feature interaction faults. Although these works use mutation to discover bugs in programs and improve the reliability of software systems, they still fall within the scope of mutation testing. This approach requires significant computation to remove redundant mutants, as well as complex strategies to generate test cases and execute them, which are computationally intensive [19]. In contrast, our approach involves modifying the source code to assist programmers in code inspection. Therefore, our strategy for generating mutants focuses more on creating mutants that can help programmers to identify errors in their own code during code inspection, rather than designing mutants with specific types of bugs that are beneficial for testing.

Another approach to ensuring system reliability through modifications to the program's source code is known as software fault injection (SFI). Among these, the subfield most relevant to this study is code change injection. In this method, the modifications made to the program's source code are intended to simulate potential faults that may occur during system operation, thereby evaluating the system's response to specified faults during its operation. The Fault Injection and Monitoring Environment (FIME) tool [25], proposed by Kao et al., was the first to address the issue of which faults should be injected in SFI. Its extended version, the DEFINE tool [26], expanded this approach to distributed environments. Similar works include G-SWFIT, the NFTAPE framework [27], and State-Driven Workload Generation [28]. Regarding the question of when faults resulting from code changes should be injected, Daran et al. [29] have suggested that the injection should occur before the code execution. Ng et al. [30] further extended this conclusion by proposing a method for injecting faults during code execution.

However, the strategy of code change injection primarily focuses on evaluating the system's fault tolerance, making the faults generated for SFI more targeted, and placing greater emphasis on the representativeness of the faults and their impact during system operation. In contrast, code inspection is usually conducted during the system development phase. Therefore, the objective of our research differs significantly from previous studies.

2.2. Motivations and Strengths

Errors in software development can be categorized based on different stages of development into compile-time errors, logical errors, resource management errors, environmental errors, and integration errors. HMPI primarily focuses on inspecting the source code during the programming process, which makes it challenging to detect environmental errors that emerge during program execution. As a result, we exclude environmental and integration errors when defining the mutation strategy. Additionally, as compile-time errors are typically detected during the compilation process, they are not the focus of our research.

Unlike traditional mutation testing, which primarily aims to evaluate the effectiveness of test cases through generating mutants with specific types of bugs, our approach focuses on assisting developers during the code inspection process. This distinction enables us to design mutants that directly guide programmers in identifying and inspecting logical errors, security vulnerabilities, and resource management issues. Through integrating targeted mutant generation into the programming process, our methodology eliminates the need for computationally intensive tasks such as removing redundant mutants or generating exhaustive test cases. Additionally, in contrast to software fault injection, which is often applied during runtime to evaluate fault tolerance, our strategy emphasizes the early stages of software development, allowing programmers to address critical issues proactively during code inspection. Compared to existing approaches, such as mutation testing and SFI, our method demonstrates significant improvements in guiding programmers to identify and rectify critical flaws in their own code. Our mutation-based strategy enhances the thoroughness of code inspections through reducing the overconfidence of programmers, which may lead to them overlook potential issues in their code. This not only improves the overall quality of code, but also minimizes the time and effort required for peer review.

In summary, the proposed MB-HMPI approach leverages deliberate fault injection to focus programmers' attention on areas prone to errors, offering a novel methodology for improving the effectiveness and efficiency of code inspections. This contribution significantly advances the field by addressing the limitations of existing techniques and providing a practical framework for enhancing the reliability of software during development.

3. Methodology

In this section, we first introduce the fundamental techniques required for MB-HMPI, followed by an overview of the MB-HMPI approach. Subsequently, we detail the mutation strategy as the foundational step in the process, explaining how mutants are constructed. Building on this, we outline the method for automating mutant generation, ensuring efficiency and scalability in the application of the mutation strategy. Finally, we propose a code comparison strategy to evaluate the effectiveness of programmers' self-inspections, guiding the decision on whether to continue coding or inspect a newly generated mutant.

3.1. Foundational Techniques

The MB-HMPI proposed in this study is established based on the initial HMPI (proposed in our previous publication), mutation testing, and the tree edit distance. In this subsection, we provide a brief introduction to these three techniques.

3.1.1. Human–Machine Pair Inspection

During the peer review process, when reviewers inspect complex and large amounts of code, it can sometimes be challenging to fully comprehend all the details of the code being reviewed due to a lack of background information or domain knowledge [31]. Self-inspection of the code by programmers can overcome the aforementioned issue. However, due to programmers' excessive familiarity and confidence in the code they have constructed [32], the effectiveness of code self-inspection may be compromised by challenges such as the absence of external perspectives and cognitive biases [33]. In order to mitigate these deficiencies of existing inspection techniques, we proposed human–machine pair inspection (HMPI) [9] in our previous work, which aims to provide a technique supporting human–machine pair programming (HMPP) [34] technology. In HMPP, a programmer and computer work together to construct a program, where the programmer plays the role of a driver and the computer plays the role of an observer. Similarly, the process of HMPI involves the computer pointing out where to inspect in the code (usually high-risk code) and generating a checklist, whereas the programmer carries out the inspection based on the checklist. The main process is illustrated in Figure 1.

Specifically, our proposed HMPI utilizes cognitive complexity [35] to guide programmers to inspect their own code. The process consists of six major steps: Pre-process the recorded data, generate the intermediate representation (IR) and control flow graph (CFG), calculate the cognitive complexity, generate the inspection syntax graph, establish the checklist, and analyze the code to detect defects. The detailed steps of HMPI are illustrated in Figure 2. It is worth noting that this method leverages complexity theory to guide the machine in identifying and recommending potentially high-risk code segments for inspection. This feature of HMPI directs programmers to inspect code areas that, theoretically, are more prone to issues, thereby enhancing the efficiency of the inspection process. However, this approach primarily addresses the question of which areas programmers should focus on during inspections.

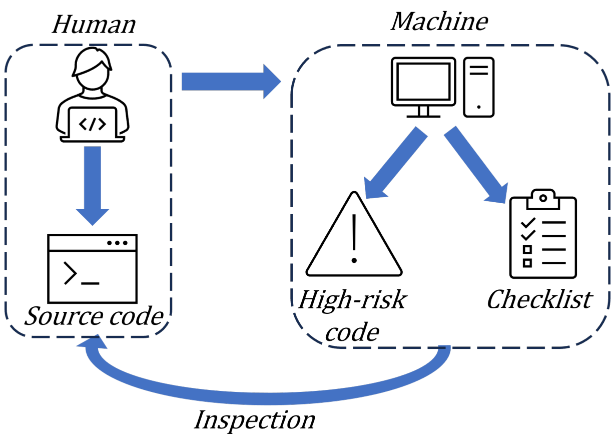

Figure 1. Overview of the human–machine pair inspection process.

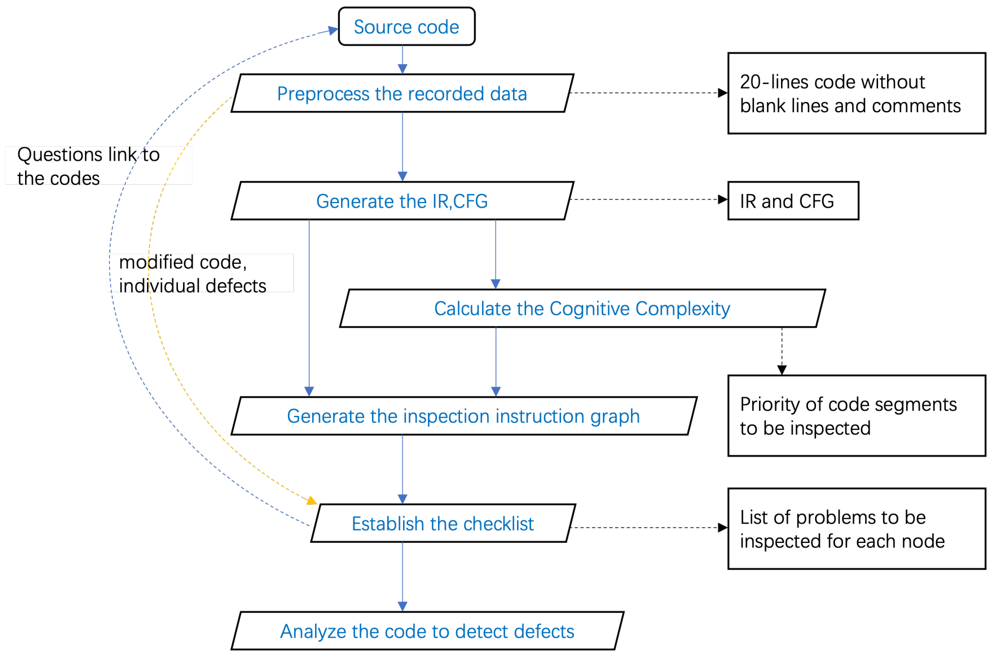
Source code
Preprocess the recorded data
20-lines code without blank lines and comments
Questions link to the codes
Generate the IR,CFG
IR and CFG
modified code, individual defects
Calculate the Cognitive Complexity
Generate the inspection instruction graph
Priority of code segments to be inspected
Establish the checklist
List of problems to be inspected for each node
Analyze the code to detect defects

Figure 2. Detailed steps of the human–machine pair inspection process.

As a result, the HMPI approach has limitations in effectively drawing the attention of the inspector to error-prone sections of code and lacks a reliable means to quantify the quality of programmers' code inspections. These shortcomings can lead to inefficiencies, particularly in large-scale software engineering projects. Therefore, we propose to use mutation to further support HMPI. The method proposed in this study does not strictly follow the original HMPI workflow, but instead adopts its overarching concept. Detailed steps of the proposed method are described in the subsequent sections.

3.1.2. Mutation Testing

Mutation testing involves assessing the effectiveness of test cases by running them on faulty versions of a program, known as **mutants** [36]. These mutants can either be manually introduced or automatically generated using a set of **mutation operators**, which are predefined rules for making changes in the code. **Equivalent mutants** [37] are syntactically different but semantically identical to the original software, performing the same function, and cannot be distinguished or detected by any test cases, meaning that they cannot be "killed". **Hard-to-kill mutants** are non-equivalent mutants that can only be detected and eliminated by a small number of highly specific test cases. These mutants are valuable as they help to enhance the fault detection capabilities of test cases. The effectiveness of the test cases is determined by how many mutants they "kill", referred to as **killed mutants**, where the output of the mutant differs from the original program for at least one test case. Consequently, test cases are crafted to eliminate as many mutants as possible.

This method is grounded in two key theoretical principles: the **competent programmer hypothesis** and the **coupling effect**. The competent programmer hypothesis assumes that programmers tend to write code that is nearly correct, meaning that most faults are relatively minor and can be corrected by small changes to the code. These small changes are represented by mutants. On the other hand, the coupling effect posits that test cases that are effective in detecting these small faults (mutants) are also likely to detect more complex faults in the program. Together, these principles provide the theoretical foundation for generating mutants and evaluating the effectiveness of tests. The process of mutation testing is shown in Figure 3. For the source code, a set of mutants is selected according to the mutation strategy. Problematic mutants—such as equivalent ones and redundant mutants, which are semantically distinct but subsumed by others—need to be removed. After forming the final set of mutants, a mutation-based test suite is generated and executed against them. In this phase, test cases are designed (either manually or automatically) to target and potentially kill the mutants. These test cases are then executed with all the mutants in order to assess their effectiveness in detecting and eliminating the introduced mutations.

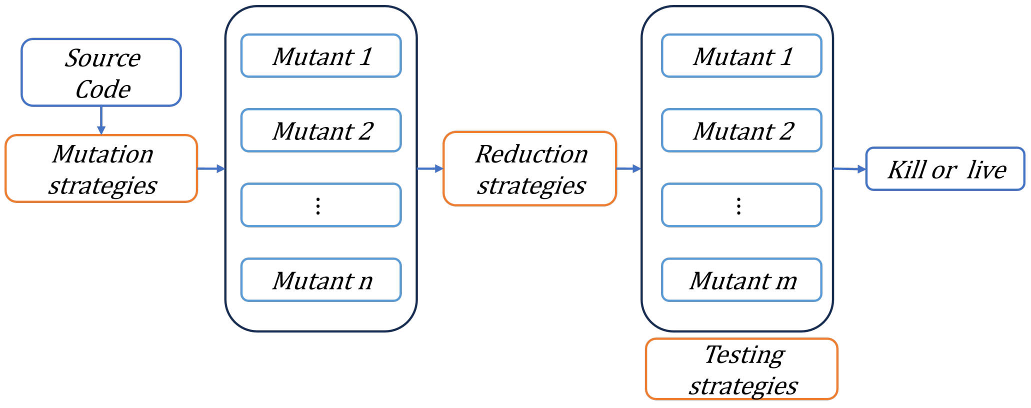

Figure 3. Mutation testing process.

A long-standing issue in mutation testing is the large number of mutants generated by mutation operators, which can overwhelm the testing process. Ideally, mutation testing should concentrate on "hard-to-kill" mutants, as these are more likely to lead to high-quality test cases and prevent the unnecessary increase in test executions caused by trivial mutants. To address this challenge, researchers have explored the development of a sufficient set of mutation operators—namely, those capable of producing a smaller number of mutants while maintaining high test effectiveness—as well as strategies for mutant sampling and clustering [19].

3.1.3. Tree Edit Distance

The tree edit distance (TED) [38] is a specialized metric for measuring the structural similarity between two tree-structured data representations. It calculates the minimum cost of a sequence of edit operations needed to transform one tree into another. Common edit operations include node insertion, deletion, and substitution, each associated with a pre-defined cost. TED is particularly suited for hierarchical data structures, such as syntax trees, XML documents, or abstract syntax trees, due to its ability to capture structural differences effectively and efficiently [39].

Definition 1 (Tree Edit Distance (TED))**.** *The tree edit distance between two rooted trees t_1 and t_2, denoted as TED (t_1, t_2), is defined as*

$$TED\,(t_1, t_2) = \min_{(e_1,\ldots,e_k)\in P(t_1,t_2)} \sum_{i=1}^{k} c(e_i). \tag{1}$$

where $P\,(t_1, t_2)$ represents the set of all possible edit paths (i.e., sequences of edit operations) that transform t_1 into t_2, and $c(e_i) \geq 0$ is the cost associated with each tree edit operation e_i. The cost function $c(e)$ can be customized based on the specific application in order to reflect the significance of different types of modifications.

Compared to the more general graph edit distance (GED), which is NP-hard, TED algorithms are computationally more efficient due to the hierarchical constraints of trees. Many TED algorithms achieve polynomial-time complexity, making TED a practical choice for analyzing structured data [40]. In this study, we use the TED to compare abstract syntax trees, as it provides an effective balance between computational feasibility and structural similarity assessment.

3.2. General Workflow

The main purpose of the proposed MB-HMPI is to utilize mutation operators to simulate potential errors that may exist in the program. This helps programmers to identify potential risks that may occur during the program development process when inspecting the mutated versions. The proposed method consists of two phases: the first phase involves the mutant generation algorithm, while the second phase focuses on the rules for mutant inspection. Its organization process is shown in Figure 4.

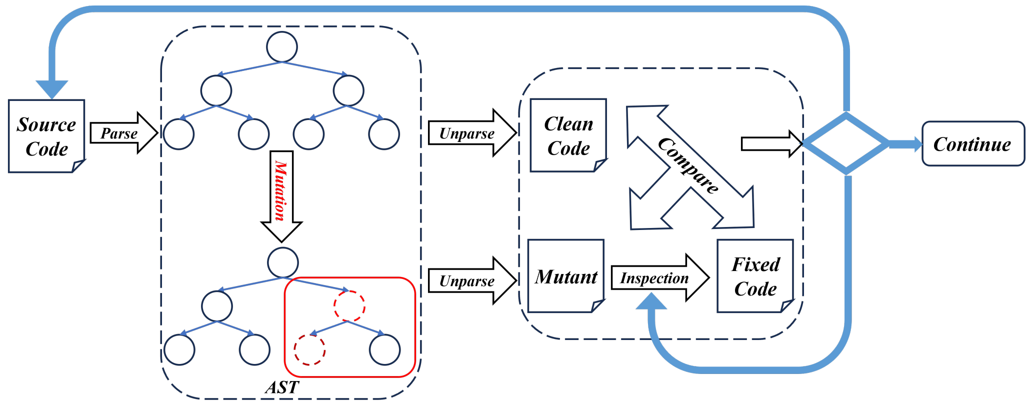

Figure 4. Overview of the mutation-based human–machine pair inspection method.

In the first phase of the MB-HMPI method, the source code is parsed into an abstract syntax tree (AST). This approach offers two key advantages. First, it facilitates monitoring of the structure of the source code. Second, by manipulating the AST, pre-defined mutation strategies can be more easily implemented to automatically generate mutants. The specific steps of the first phase consist of three parts:

- The AST is traversed according to a given set of rules, and the nodes requiring modification are identified.
- The identified nodes are modified according to the specified patterns, as illustrated in the red box in Figure 4.
- The modified AST is recompiled into mutants, which are then provided to programmers for inspection and evaluation.

The detailed implementation of mutant generation is discussed in Section 3.4.

In the second phase, the mutant code is inspected by programmers, who then submit the fixed code. The fixed code is compared against both the original clean code and the mutant code, resulting in one of the three possible outcomes:

- The inspection passes, and the programmers continue with development.
- The fixed code undergoes another round of mutation, generating new mutated code for further inspection by the programmers.
- The inspection fails, and the programmers continue inspecting the mutated code. The specific comparison strategy and implementation method are provided in Section 3.5.

3.3. Mutation Model

The main purpose of generating mutants in MB-HMPI is to guide programmers in identifying mutations within the mutants, thereby improving the efficiency of code inspection and uncovering potential errors in the code. Therefore, the injected mutations should reflect as many typical errors often introduced to source code as possible. To achieve this, we propose a defect-based mutation operator setting strategy and a mutation rule setting strategy.

3.3.1. Defect-Based Mutation Operators

We propose a strategy for designing mutation operators based on empirical patterns of historical real-world errors, aimed at guiding programmers in conducting more effective code inspections. This strategy involves two key steps: identifying and summarizing the real defects that have occurred historically, along with how they were fixed. Through extracting and analyzing the modified segments of the code, we can derive meaningful patterns that inform the systematic design of mutation operators, enabling them to capture and simulate real-world error scenarios effectively. In modern software development, peer review is frequently performed on version control platforms that support distributed collaboration through mechanisms such as pull requests. Pull requests encapsulate detailed information about code modifications, with "Diffs" being fundamental components that visualize changes line-by-line, thereby streamlining the review process and enhancing collaborative discussions. Historical defect data in the code can be obtained through pull requests in GitHub. Through analyzing defect-related Diff files in GitHub, it is possible to manually summarize the mutation operators corresponding to different keywords in the program.

As shown in Table 1, as the structure of a program is determined by keywords, linking keywords to mutation operators establishes a relationship between the program structure and mutation operators. First, let KW be the set of all keywords in a programming language and let the mapping $\phi : P \to P'$ represent a mutation operator, where P is the original program and P' is the modified program. Here, ϕ is derived manually by summarizing the **Diff** file. Secondly, we construct a set of mutation operators for each keyword, denoted as $MO_k = \{\phi_{1k}, \ldots, \phi_{sk}\}$, where the k belongs to the keyword set KW and s represents the number of mutation operators defined for the keyword k. Here, the values of $|MO_i|$ and $|MO_j|$ can be different for different keywords i and j.

Table 1. Fundamental constructs, operators, and keywords used in programming.

Construct	Keywords	Operators
Sequence Structure	- Import-from Keyword - As Keyword - Def Keyword - With-as Keyword	- Arithmetic Operators - Bitwise Operators - Assignment Operators

Table 1. *Cont.*

Construct	Keywords	Operators
Selection Structure	- If-else Keyword - Try-except Keyword - Assert Keyword	- Comparison Operators - Logical Operators - The Operators of Sequence Structure
Iteration Structure	- For Keyword - While Keyword - Break-continue Control Keywords	- Membership Operators - The Operators of Selection

To clarify the above strategy, we take the keywords **For** and **If** as examples. Specific mutation operators can be assigned to each keyword, as shown in Table 2. For the keyword **For**, its corresponding mutation operators can be represented as the set MO_f, where $|MO_f| = 3$. Similarly, for the keyword **If**, its corresponding mutation operators can be represented as the set MO_i, where $|MO_i| = 3$.

Table 2. Examples of keyword-to-mutation operator mappings.

Keywords	Mutations	Example
For	- Boundary Change - Body Statement - Control Variable	- $range(1,2) \rightarrow range(1,4)$ - $delete\ break$ - $for\ i\ in\ List \rightarrow for\ j\ in\ List$
If-else	- Condition Logic - Condition Value - Control Flow	- $\&\& \rightarrow \|\|$ - $if(x > 5) \rightarrow if(true)$ - $add\ return$

If a large number of mutation operators are assigned to each keyword and applied together to generate source code mutants, it may result in the code becoming unrecognizable. Therefore, it is necessary to manually establish rules for the use of mutation operators based on different code structures.

3.3.2. Mutation Rule

Unlike the purpose of generating mutants in mutation testing, our approach aims to guide programmers in inspecting code. Instead of generating a large number of random mutants for a piece of source code, we generate a single, unique mutant. If defined mutation operators are used to randomly modify the code, the resulting mutants may differ significantly from the original code. Such mutants are unlikely to effectively guide programmers in performing code self-inspection and fail to achieve the intended purpose. To achieve this, we define specific mutation rules tailored to different code structures.

These rules are defined based on the structure of the code. Consider the following *Example_rule*: if the control structure of a given source code segment contains two nested *FOR* loops, with an *IF* condition inside the inner loop, the boundary change mutation operator is applied to the first level *FOR* loop in that code segment. Applying the above rule to the code in Listing 1, the MB-HMPI will generate the mutant in Listing 2.

Listing 1. Original code.

```python
for i in range(1, 10):
    for j in range(1, 5):
        if j % 2 == 0:
            print(f"i: {i}, j: {j}")
```

Listing 2. The mutant.

```python
for i in range(1, 100):
    for j in range(1, 5):
        if j % 2 == 0:
            print(f"i: {i}, j: {j}")
```

Many similar rules can also be defined. To implement rule matching within code, the AST of the source code is traversed to extract a nested tree, which is structured according to the keywords that define the code's architecture. We define a set of nested trees $T = \{t_1, t_2, \ldots, t_n\}$, where n is the number of code nested tree templates we have defined. For each nested tree, we define an If-Then rule, denoted as r_i: If the nested structure of the source code matches t_i, then apply the mutation operators in the set X_i to the source code. We define the set X_i as

$$X_i = \bigcup_{k \in K} \{\phi_{jk} \mid j \in J_k\},$$

where $K = \{k_1, k_2, \ldots, k_c\} \subseteq KA = \{k_1, k_2, \ldots, k_w\}$, with w being the total number of keywords and c being the number of keywords that need to be modified in the nested structure; furthermore, $J_k \subseteq MO_k$ is the set of mutation operators for the keyword in the nested structure t_i.

To better understand the description above, we continue using the code structure described in $Example\_rule$ as an example. Here, $KA = \{For, If\}$ represents the set of two keywords defining the nested structure, while $K = \{For\}$ specifies the keyword that needs to be mutated according to the rule; and $MO_k = \{Boundary\ Change, Body\ Statement, Control\ Variable\}$ represents all mutation operators applicable to the For keyword, whereas $J_k = \{Boundary\ Change\}$ denotes the restricted set of mutation operators allowed by the $Example\_rule$. The nested rule described in $Example\_rule$ corresponds to the set of applicable mutation operators $X = \{Boundary\ Change\}$.

3.4. Automatic Mutant Generation

To ensure the feasibility of this strategy without human intervention, we propose an automated AST-based mutant generation method. We detail the process of generating mutants for code inspection from two key perspectives: the location (Where) and the type of error (What).

Where: First, convert the source code into an abstract syntax tree (AST). Next, traverse the AST to obtain the nested structures between specified keywords. Finally, compare the identified nested structures with the pre-defined nested tree structure. If a match is found, return the AST node that needs to be modified. For specific details, see Algorithm 1.

What: We apply the pre-defined mutation operator for each structure to the AST nodes that need modification, as identified in the location step. After mutation, the mutated AST is converted back into source code, which becomes the mutant. For specific details, see Algorithm 2.

Algorithm 1: Location.

```
Input: Source_Code, StructureSet
Output: nodes_to_modify
Nodes = AST.parse(Source_Code)
Function Extract_Structure(Node, Level):
    ControlFlowNodes ← keywords;
    Structure ← [ ];
    if Node in ControlFlowNodes then
        Append (Node_Type_Name, Level) to Structure;
    end
    for Child in Children of Node do
        Structure ← Structure + Extract_Structure(Child, Level + 1);
    end
    return Structure;
Function CompareStructures(Structure, StructureSet):
    for S in StructureSet do
        if S matches Structure then
            return nodes_to_modify;
        end
    end
```

Algorithm 2: Mutant.

```
Input: Source Code
Output: Mutant
begin
    AST = Parse (Source Code);
    nodes_to_modify = Location (AST);
    for AST.node in AST do
        if AST.node in nodes_to_modify then
            AST.node = Fix (AST.node, Operator);
        end
    end
    Mutant = Unparse (AST);
    return Mutant;
end
```

3.5. Code Comparison Strategy

The mutants generated by MB-HMPI are presented to programmers for inspection, requiring them to manually restore the original code from the mutants. The extent to which programmers restore the mutants is used to evaluate the effectiveness of their code self-inspection. Specifically, when the programmer fixes a mutant, we obtain the fixed code. Then, the fixed code is compared with both the mutant and the source code to calculate the modification ratio. This allows us to quantify the effectiveness of the programmer's code self-inspection based on the differences between the codes. To clarify the evaluation process, we first introduce the following key definitions.

Definition 2 (Mutation Detection Rate (MDR))**.** *Let T_1 be the AST of the fixed code, and T_2 be the AST of the mutant. If Substitution(T_1, T_2) represents the total cost of node substitutions in the tree edit distance and $|M|$ is the number of mutations in the mutant, then the MDR is defined as*

$$MDR = \frac{Substitution(T_1, T_2)}{|M|}. \tag{2}$$

Definition 3 (Code Modification Rate (CMR))**.** *Let T_1 be the AST of the fixed code, and T_3 be the AST of the clean code. If $Insert(T_1, T_3)$ represents the total sum of node insertion costs in the tree edit distance, $Deletions(T_1, T_3)$ represents the total sum of node deletion costs in the tree edit distance, and $|T_3|$ represents the total number of nodes in T_3, then the CMR is defined as*

$$CMR = \frac{Insert(T_1, T_3) + Deletions(T_1, T_3)}{|T_3|}. \tag{3}$$

Definition 2 aims to measure how many of the inserted mutations the programmer identified and corrected during the code inspection. If the modification ratio is too small, MB-HMPI will assume that the programmer did not perform a thorough code inspection. Conversely, if the modification ratio is relatively large, it will be considered that the programmer's inspection was effective. In contrast, Definition 3 measures whether these mutations prompted the programmer to modify the previously written source code. If the modification ratio is too large, MB-HMPI will assume that the programmer identified potential issues in the code during the inspection process. On the other hand, if the modification ratio is too small, it suggests that the inspection process did not uncover potential issues in the source code. However, during the code inspection, the programmer may modify both the inserted mutations from MB-HMPI and the source code simultaneously. Therefore, we designed Table 3, which presents a method to balance both metrics.

Table 3. Comparison strategy.

	$MDR \geq TM\%$	$MDR < TM\%$
$CMR \geq TC\%$	Re-mutation	Re-mutation
$CMR < TC\%$	Continue	Inspection

In Table 3, TM% is the threshold of the MDR manually set by the programmer and TC% is the threshold of the CMR manually set by the user. When the CMR exceeds TC%, the MB-HMPI considers that the programmer has made significant code modifications after the inspection. In this case, regardless of the MDR value, our algorithm will re-insert mutations into the modified code and generate a new mutant. If the CMR is less than the given TC%, our algorithm considers that the programmer has not made substantial modifications to the code and, in this case, the size of the MDR will determine whether the programmer's inspection passes.

4. Results and Discussion

In this section, we present two case studies. Case study 1 is used to elaborate on how generating mutants of the source code guides programmers in discovering potential errors in the source code, while case study 2 provides a detailed demonstration of how our method generates mutants. We then evaluate the effectiveness of our MB-HMPI method through a series of controlled experiments, subsequently addressing the research questions outlined in the Introduction. Finally, we provide a comprehensive discussion of the study's implications, limitations, and potential future directions.

4.1. Case Study 1

To explain our algorithm, we begin by addressing the following specific requirement: Given an integer array and a sliding window of size k, the window moves from the leftmost side of the array to the rightmost, shifting one position to the right at each step. In each step, the task is to return the maximum value within the current window. To illustrate and clarify the development of our approach, we first present an initial implementation that contains potential errors (out-of-bounds), serving as the basis for further refinement and discussion.

- **Step 1: Generate the mutant**

  The code in Listing 3 is an implementation of the above requirements. However, in the loop on line 8, `for j in range(1, k)` is used to find the maximum value within the sliding window. If the array length is insufficient to accommodate the specified window size, it may lead to out-of-bounds access on `arr[i + j]`. Although the check `k > len(arr)` on line 2 ensures that the window size is valid, this check does not account for all potential out-of-bounds cases.

Listing 3. Source code containing potential errors.

```
1  def maxSlidingWindow(arr, k):
2      if k <= 0 or k > len(arr):
3          return None
4      max = []
5      for i in range(len(arr) - k + 1):
6      # for i in range(len(arr) - k + 2):
7          windowMax = arr[i]
8          for j in range(1, k):
9              if arr[i + j] > windowMax:
10                 windowMax = arr[i + j]
11         max.append(windowMax)
12     return max
```

As the control statement nesting structure from **lines 5** to **10** in Listing 3 is consistent with the structure in Listing 1 referenced in Section 3.3.2, we apply boundary operators to the first `for` loop in Listing 3—namely, `for i in range(len(arr) - k + 1)`—according to the mutation rule bound to Listing 1. Finally, the mutant generated by MB-HMPI only needs to replace **line 5** in Listing 3 with the content from **line 6**.

- **Step 2: Code inspection of the mutant**

  The programmer reviews the mutant generated after modifying Listing 3, and there are two possible outcomes. The first possible outcome is that the mutant is modified back to the original code in Listing 3. This situation corresponds to "**continue**" in Table 3, where MB-HMPI will consider the code in Listing 3 to be correct. The second possible outcome is that the programmer identifies a potential error in Listing 3. Through adding the conditional statement in **line 8** and incorporating error handling, the code in Listing 4 written by the programmer can prevent out-of-bounds read errors.

Listing 4. The code after inserting mutants.

```
1  def maxSlidingWindow(arr, k):
2      if k <= 0 or k > len(arr):
3          return None
4      max = []
5      for i in range(len(arr) - k + 1):
6      # for i in range(len(arr) - k + 20000):
7          windowMax = arr[i]
8          for j in range(1, k):
9              if i + j < len(arr):
10                 if arr[i + j] > windowMax:
11                     windowMax = arr[i + j]
12             else:
13                 print(f"Error: Accessing out of bounds at index {i + j}.")
14                 return None
15         max.append(windowMax)
16     return max
```

- **Step 3: Re-mutation**

After the programmer re-writes the code in Listing 3, resulting in Listing 4, MB-HMPI needs to mutate Listing 4 again, according to the rules in Table 3. As the control statement nesting structure from **lines 5 to 14** in Listing 4 is still consistent with the structure in Listing 1 referenced in Section 3.3.2, we apply boundary operators to the first `for` loop in Listing 4; namely, `for i in range(len(arr) - k + 1)`. Suppose this time that we change 1 $\rightarrow$ 20,000. In this case, the mutant generated by MB-HMPI should replace **line 5** in Listing 4 with the content from **line 6**. Actually, as this implementation uses a Brute Force Algorithm, it typically has a high time complexity due to a lack of optimization. Although it can produce correct results for small test cases, the computation time often becomes very long as the problem size increases. When the programmer reviews the mutant generated after modifying Listing 4, there is a certain probability that they will identify a potential issue with the algorithm's high complexity in Listing 4. As a result, they may apply a queue optimization technique, leading to the code shown in Listing 5.

Listing 5. Optimized code.

```
def maxSlidingWindow(arr, k):
    deque = collections.deque()
    max, n = [], len(arr)
    for i, j in zip(range(1 - k, n + 1 - k), range(n)):
        if i > 0 and deque[0] == arr[i - 1]:
            deque.popleft()
        while deque and deque[-1] < arr[j]:
            deque.pop()
        deque.append(arr[j])
        if i >= 0:
            max.append(deque[0])
    return max
```

4.2. Case Study 2

In this subsection, we progressively demonstrate the steps for automatically generating mutants using Algorithms 1 and 2. For the source code in Listing 6, to facilitate automatic monitoring of the code structure and mutant generation, we first convert the Python code into an abstract syntax tree.

Listing 6. Python code for parsing.

```
def process_number(num, target, increment):
    if num > target:
        while num > target:
            if num % 2 == 0:
                num -= 2
            else:
                num -= 1
    else:
        while num < target:
            if num + increment > target:
                increment = target - num
            num += increment
    return num
```

As shown in Figure 5, through performing a depth-first traversal of the abstract syntax tree using Algorithm 1, we can accurately identify the **if** and **while** keywords and determine that the code in Listing 6 exhibits a three-level nested structure. As source code with higher complexity is more likely to contain potential errors, we have established a rule: if the source code contains multiple levels of nesting, we only mutate the first level. In the case study, Algorithm 1 returned the $if(num > target)$ node in the AST. Next, as shown in Figure 6, we used a pre-defined mutation operator to modify the $if(num > target)$ node, as detailed in Algorithm 2.

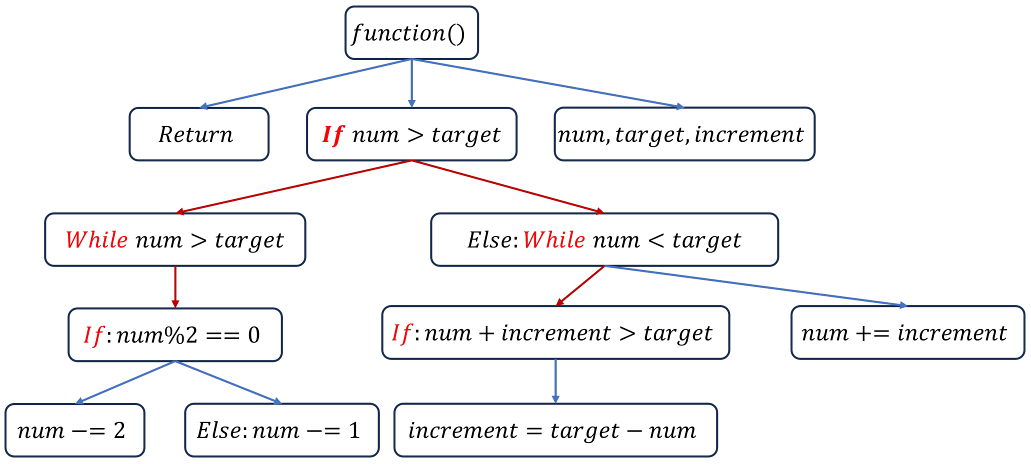

Figure 5. AST for Python code.

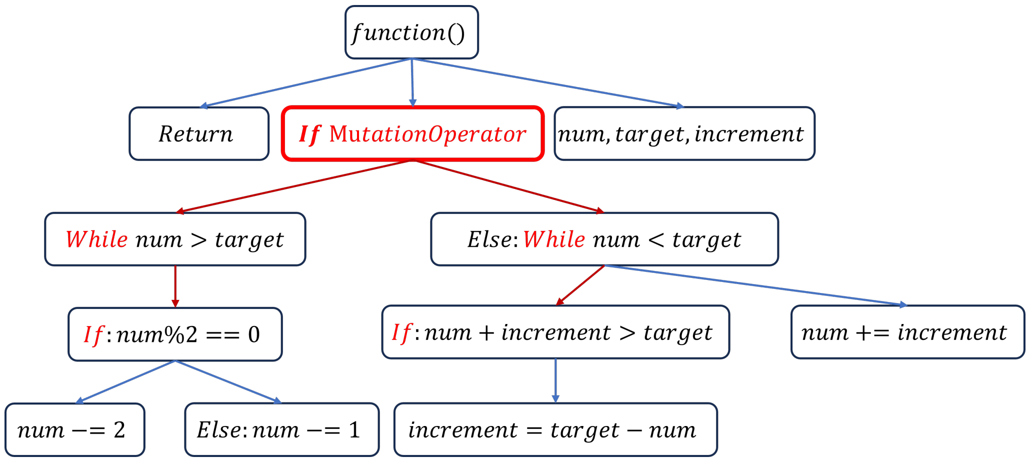

Figure 6. AST for Python code with mutation.

4.3. Experiments

To validate the proposed approach, we conducted a comparative evaluation against the popular inspection technique known as checklist-based reading (CBR) [41]. To measure the impact of this approach, we recruited four postgraduate students in the field of software engineering (labeled as 1, 2, 3, and 4) who met the following criteria: (1) High proficiency in Python programming, (2) familiarity with checklist-based reading method, and (3) the ability to understand and manually replicate the process of MB-HMPI. Prior to the experiment, the participants underwent comprehensive training that consisted of three phases: (1) Theoretical instruction, where the principles of the checklist-based reading method and the MB-HMPI approach were explained in detail, including the rationale and design of mutation operators; (2) demonstration sessions, in which we showcased the step-by-step application of both the checklist-based reading method and the use of mutation operators to guide code inspection, simulating the inspection process; and (3) hands-on practice, where participants independently applied the checklist-based reading method and the MB-HMPI method to example code under supervision, followed by feedback and refinement of their understanding. This structured training ensured that participants not only understood the checklist-based reading method and machine-assisted MB-HMPI process and could simulate their functionality, but also demonstrated their ability to independently apply the MB-HMPI method for effective code inspection.

The experiment was conducted in three phases using the Python 3 programming language, executed on the LeetCode online programming platform. The participants were introduced to the platform's features and usage prior to the experiment. LeetCode was chosen for its ability to control coding difficulty, track code execution time, measure lines of code, monitor pass rates, and provide integrated timing functions. For the experiment, each participant independently selected a set of problems from LeetCode's "hard" category, in order to ensure consistency in the difficulty and complexity of the problems. A total of 18 problems (numbered 1 through 18) with a difficulty of "hard" were randomly selected from LeetCode by all participants. These problems were chosen to reflect sufficient algorithmic complexity and real-world applicability.

Phase 1: Evaluation of programming without code inspection

In the first phase, we tested the participants' effectiveness in programming independently without code inspection. Each participant worked individually to complete problems 1 and 2 selected from LeetCode without code inspection. For each task, the following metrics were recorded:

- Programming Time: The total time spent by the programmer in writing the code.
- Execution duration: The time taken for the code to run during testing.
- Submission count: The number of times the programmer submitted the code to the system for verification.

Phase 2: Evaluation of CBR Method

In the second phase, we tested the effectiveness of traditional inspection methods. Participants were organized into pairs to perform specific programming and inspection tasks. Participant 1 was tasked with programming solutions to problems 3 and 4, after which participant 2 inspected the code. This pair was designated as group A. Participant 3 was tasked with programming solutions to problems 5 and 6, followed by participant 4 inspecting the code. This pair was designated as group B. After the initial tasks were completed, the roles within each group were reversed: Participant 2 assumed the programming role for problems 7 and 8, while participant 1 became the inspector. This role-reversed pair was termed group A′. Participant 4 programmed solutions to problems 9 and 10, with participant 3 as the inspector. This pair was designated as Group B′. For each task, the following metrics were recorded: Programming time, inspection time (the time taken by the inspector to evaluate and provide feedback on the code), execution duration, and submission count. For a comprehensive overview, refer to Table 4.

Phase 3: Evaluation of MB-HMPI

In the last phase, we tested the effectiveness of our proposed method. The participants were paired into two groups, working together to solve the remaining eight problems selected from LeetCode. The group collaboration and naming method was similar to that in phase 2, except that the task of the inspector in phase 2 was to simulate the machine's role through inserting mutants and guiding the inspection process, as well as recording relevant data; specifically, participants 1 and 3 focused on coding, while participants 2 and 4 simulated the machine's role. According to Section 3.3, ten mutants were introduced into the code. Upon completion, the roles were switched: participants 2 and 4 took over coding, while participants 1 and 3 simulated the machine's functionality. Thus, group A completed programming problems 11 and 12, group B handled problems 13 and 14, group A′ worked on problems 15 and 16, and group B′ was responsible for problems 17 and 18. As the time taken by the tool to insert mutants is negligible in practice, the time spent by participants simulating this function did not count toward the programming time. For each task, the following metrics were recorded: Programming time, execution duration, and submission count.

The specific participants, their assigned tasks, the problems completed (selected from LeetCode), and the data recorded are summarized in Table 4.

Table 4. Task allocation in the experimental setup.

Phase	Group	Participant	Problem	Task	Recorded data
1	1 2 3 4	1 2 3 4	1, 2 1, 2 1, 2 1, 2	Programming independently without code inspection	• Programming Time • Execution Duration • Submission Count
2	A	1 2	3, 4	Programming Inspection	• Programming Time • Inspection Time • Execution Duration • Submission Count
	B	3 4	5, 6	Programming Inspection	
	A′	1 2	7, 8	Inspection Programming	
	B′	3 4	9, 10	Inspection Programming	
3	A	1 2	11, 12	Programming Machine's role	• Programming Time • Execution Duration • Submission Count
	B	3 4	13, 14	Programming Machine's role	
	A′	1 2	15, 16	Machine's role Programming	
	B′	3 4	17, 18	Machine's role Programming	

4.4. Experimental Results

The first phase primarily evaluated the participants' independent programming capabilities, serving as a baseline for subsequent method comparisons. Table 5 presents the programming time, execution duration, and the number of submissions required for each participant to complete different coding tasks. Shorter programming times indicate higher programming efficiency, while shorter execution durations and fewer submissions reflect higher code quality. Based on these metrics, the participants' coding proficiency can be ranked as follows: $4 > 2 > 1 > 3$.

Table 5. Experimental results of programming without inspection.

Participant	Problem	Programming Time (min)	Execution Duration (ms)	Submission Count
1	1	32	37	7
	2	25	31	5
2	1	27	19	4
	2	20	24	2
3	1	43	56	9
	2	36	47	7
4	1	20	18	3
	2	16	17	2

In the second phase, after introducing CBR, a comparison with the first phase (where no inspections were conducted) revealed certain trends in the data presented in Table 6. The programming time increased as participants needed to make modifications based on inspection feedback, leading to a significant increase in the overall development time (including both programming and inspection). However, the number of submissions decreased slightly, indicating only a limited improvement in code quality. Even the most skilled participant 4 did not significantly reduce the number of submissions, likely due to their limited familiarity with others' code.

Table 6. Experimental results with CBR method.

Group	Programmer	Problem	Programming Time (min)	Inspection Time (min)	Execution Duration (ms)	Submission Count
A	1	3	28	13	32	5
		4	34	11	35	5
B	3	5	46	10	46	7
		6	40	9	38	6
A′	2	7	24	15	18	3
		8	31	14	20	2
B′	4	9	19	16	17	2
		10	21	15	14	2

In the third phase, our approach involved introducing mutants to facilitate code inspection. Consequently, the programming time increased due to the additional time spent on self-inspection. However, as shown in Table 7, the programming time (including inspection) using our proposed MB-HMPI method was relatively shorter compared to the CBR inspection method in Table 6. This reduction is mainly attributed to the programmers' familiarity with their own code, which reduces the time spent inspecting it. Additionally, the number of submissions was lower than with the traditional method, indicating higher code quality. Overall, our method effectively enhanced the inspection process, leading to improved development efficiency and code quality.

Table 7. Experimental results with MB-HMPI method.

Group	Programmer	Problem	Programming Time (min)	Execution Duration (ms)	Submission Count
A	1	11	36	30	4
		12	40	33	3
B	3	13	45	36	5
		14	57	43	6
A′	2	15	32	19	2
		16	37	17	2
B′	4	17	28	12	1
		18	30	15	2

4.5. Responses to Research Questions

The research questions presented in the Introduction are addressed as follows:

- RQ1. How can an automated approach effectively attract programmers' attention during the HMPI process to mitigate the overconfidence caused by familiarity with their own code?
- A1. Our proposed MB-HMPI approach leverages human expertise through integrating mutation-based techniques to strategically guide programmers' attention to specific components of the code. Utilizing mutation operators derived from historical defects and applying automated AST-based mutant generation, the approach ensures that code inspection remains focused, efficient, and less prone to oversights.

- RQ2. How can a mutation-based method be utilized to guide programmers in identifying potential issues in their code during the inspection process?
- A2. The mutation-based method generates mutants that simulate common logical and structural errors, enabling programmers to identify potential flaws during the inspection process. Through combining deliberate fault injection and structured inspection strategies, programmers are guided toward uncovering hidden issues in their code that might otherwise be missed.
- RQ3. What metrics can be designed to quantitatively evaluate the effectiveness of code inspections by programmers?
- A3. We introduce two novel metrics: the Mutation Detection Rate (MDR) and Code Modification Rate (CMR). The MDR measures the programmer's ability to detect and repair inserted mutations, while the CMR evaluates the extent to which programmers refine the original source code based on the inspection process. These metrics provide a robust quantitative assessment of the effectiveness of code inspections conducted through MB-HMPI.

4.6. Implications, Limitations, and Future Prospects

Implications: The proposed MB-HMPI approach significantly advances the field of code inspection by leveraging mutation-based strategies to enhance the ability of programmers to identify and rectify defects in their own code. This method reduces reliance on external reviewers, thereby saving time and resources, while also empowering programmers to conduct more thorough and focused inspections. Furthermore, the introduction of metrics (MDR and CMR) offers a systematic framework for evaluating the effectiveness of code inspections, which can be adapted to various programming contexts. These implications underscore the potential of MB-HMPI to improve software quality and reliability throughout the development process.

Limitations: Despite its promising contributions, the MB-HMPI framework has certain limitations. The reliance on manually designed mutation operators introduces variability and limits the generalizability of the approach across different programming languages and domains. Additionally, while the method partially automates code inspection, it still requires significant human intervention to customize mutation operators and manage mutants, which may hinder scalability in large-scale industrial projects. These limitations highlight areas where further refinement is needed to enhance the framework's adaptability and automation.

Future Prospects: Future research could address these limitations by leveraging advanced machine learning techniques, such as deep learning and natural language processing, to automatically extract mutation patterns from historical defect data in version control systems. For instance, models trained on historical defect data could identify common patterns and create tailored mutants for specific programming scenarios. Additionally, integrating MB-HMPI with AI-driven static analysis tools could further expand its defect detection capabilities. Extending the framework to support multiple programming languages and incorporating dynamic analysis techniques would also broaden its applicability. By pursuing these directions, the MB-HMPI approach could evolve into a comprehensive solution for improving code quality and inspection efficiency. Further refinements could extend its applicability and impact, making it a valuable tool for both academic research and practical software development.

5. Conclusions

The mutation-based human–machine pair inspection (MB-HMPI) approach introduced in this study represents a novel and systematic method to enhance the code inspection processes of programmers. Leveraging mutation operators tailored to historical defect patterns and integrating automated tools with programmer expertise, MB-HMPI effectively addresses common challenges in code inspection, such as programmer overconfidence and oversight of potential errors.

This study comprehensively described the framework and methodology of MB-HMPI, including the selection and application of mutation operators, the automated generation of mutants, and the development of metrics—including the Mutation Detection Rate (MDR) and Code Modification Rate (CMR)—for quantitative assessments of inspection effectiveness, as well as the way in which the guidance is used by the programmer to carry out inspections. Two case studies demonstrated the process of applying our MB-HMPI. The results of controlled experiments further demonstrated the method's potential to improve the thoroughness of code inspections while reducing inspection time.

However, this study also acknowledges certain limitations, such as the dependency on manually defined mutation operators and scalability challenges in large-scale applications. Future work should aim to overcome these limitations by incorporating advanced machine learning techniques to automate mutant generation and enhance the adaptability of MB-HMPI across different programming languages and domains. By addressing these limitations, MB-HMPI can evolve into a comprehensive solution that not only improves the quality and efficiency of code inspections but also establishes a robust framework for advancing research and practice in the context of software engineering.

Author Contributions: Conceptualization, S.L.; Methodology, Y.D. and S.L.; Software, H.L.; Investigation, Y.D. and H.L.; Data curation, Y.D. and H.L.; Writing—original draft, Y.D.; Writing—review & editing, S.L., Y.D. and H.L.; Supervision, S.L.; Funding acquisition, Y.D., S.L. and H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by JST, the establishment of university fellowships towards the creation of science technology innovation, Grant Number JPMJFS2129.

Data Availability Statement: Data are contained within the article.

Acknowledgments: We extend our gratitude to all participants who contributed to the experiments conducted in this study for their efforts and cooperation in fulfilling their tasks.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Badampudi, D.; Unterkalmsteiner, M.; Britto, R. Modern code reviews—Survey of literature and practice. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–61. [CrossRef]
2. Fagan, M. Design and code inspections to reduce errors in program development. In *Software Pioneers: Contributions to Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 575–607.
3. Ackerman, A.F.; Buchwald, L.S.; Lewski, F.H. Software inspections: An effective verification process. *IEEE Softw.* **1989**, *6*, 31–36. [CrossRef]
4. Sadowski, C.; Söderberg, E.; Church, L.; Sipko, M.; Bacchelli, A. Modern code review: A case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, 27 May–3 June 2018; Paulisch, F., Bosch, J., Eds.; ACM: New York, NY, USA, 2018; pp. 181–190. [CrossRef]
5. Wessel, M.S.; Serebrenik, A.; Wiese, I.; Steinmacher, I.; Gerosa, M.A. Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, 28 September–2 October 2020; pp. 1–11. [CrossRef]
6. McIntosh, S.; Kamei, Y.; Adams, B.; Hassan, A.E. The impact of code review coverage and code review participation on software quality: A case study of the qt, VTK, and ITK projects. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 192–201.
7. Wessel, M.S.; Serebrenik, A.; Wiese, I.; Steinmacher, I.; Gerosa, M.A. What to Expect from Code Review Bots on GitHub?: A Survey with OSS Maintainers. In Proceedings of the 34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, 19–23 October 2020; Cavalcante, E., Dantas, F., Batista, T., Eds.; ACM: New York, NY, USA, 2020; pp. 457–462. [CrossRef]

8. Wurzel Gonçalves, P.; Calikli, G.; Serebrenik, A.; Bacchelli, A. Competencies for code review. *Proc. ACM Hum.-Comput. Interact.* **2023**, *7*, 1–33. [CrossRef]
9. Dai, Y.; Liu, S. Applying Cognitive Complexity to Checklist-Based Human-Machine Pair Inspection. In Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021—Companion, Hainan Island, China, 6–10 December 2021; pp. 314–318. [CrossRef]
10. Natella, R.; Cotroneo, D.; Madeira, H.S. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv. (CSUR)* **2016**, *48*, 1–55. [CrossRef]
11. Duraes, J.A.; Madeira, H.S. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.* **2006**, *32*, 849–867. [CrossRef]
12. Charoenwet, W.; Thongtanunam, P.; Pham, V.; Treude, C. An Empirical Study of Static Analysis Tools for Secure Code Review. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, 16–20 September 2024; Christakis, M., Pradel, M., Eds.; ACM: New York, NY, USA, 2024; pp. 691–703. [CrossRef]
13. Singh, D.; Sekar, V.R.; Stolee, K.T.; Johnson, B. Evaluating how static analysis tools can reduce code review effort. In Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, 11–14 October 2017; Henley, A.Z., Rogers, P., Sarma, A., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2017; pp. 101–105. [CrossRef]
14. Tufano, R.; Pascarella, L.; Tufano, M.; Poshyvanyk, D.; Bavota, G. Towards automating code review activities. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 163–174.
15. Li, Z.; Lu, S.; Guo, D.; Duan, N.; Jannu, S.; Jenks, G.; Majumder, D.; Green, J.; Svyatkovskiy, A.; Fu, S.; et al. Automating code review activities by large-scale pre-training. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 1035–1047.
16. Tufano, R.; Masiero, S.; Mastropaolo, A.; Pascarella, L.; Poshyvanyk, D.; Bavota, G. Using pre-trained models to boost code review automation. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 2291–2302.
17. Bacchelli, A.; Bird, C. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 712–721.
18. Davila, N.; Nunes, I. A systematic literature review and taxonomy of modern code review. *J. Syst. Softw.* **2021**, *177*, 110951. [CrossRef]
19. Papadakis, M.; Kintis, M.; Zhang, J.; Jia, Y.; Le Traon, Y.; Harman, M. Mutation testing advances: An analysis and survey. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2019; Volume 112, pp. 275–378.
20. Ziade, H.; Ayoubi, R.A.; Velazco, R. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* **2004**, *1*, 171–186.
21. Loise, T.; Devroey, X.; Perrouin, G.; Papadakis, M.; Heymans, P. Towards Security-Aware Mutation Testing. In Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, 13–17 March 2017; IEEE Computer Society: Piscataway, NJ, USA, 2017; pp. 97–102. [CrossRef]
22. Nanavati, J.; Wu, F.; Harman, M.; Jia, Y.; Krinke, J. Mutation testing of memory-related operators. In Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Graz, Austria, 3–17 April 2015; IEEE Computer Society: Piscataway, NJ, USA, 2015; pp. 1–10. [CrossRef]
23. Wu, F.; Nanavati, J.; Harman, M.; Jia, Y.; Krinke, J. Memory mutation testing. *Inf. Softw. Technol.* **2017**, *81*, 97–111. [CrossRef]
24. Garvin, B.J.; Cohen, M.B. Feature Interaction Faults Revisited: An Exploratory Study. In Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, 29 November–2 December 2011; Dohi, T., Cukic, B., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2011; pp. 90–99. [CrossRef]
25. Kao, W.I.; Iyer, R.K.; Tang, D. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Trans. Softw. Eng.* **1993**, *19*, 1105–1118. [CrossRef]
26. Kao, W.L.; Iyer, R. DEFINE: A distributed fault injection and monitoring environment. In Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, College Station, TX, USA, 12–14 June 1996; pp. 252–259.
27. Stott, D.T.; Floering, B.; Burke, D.; Kalbarczpk, Z.; Iyer, R.K. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In Proceedings of the IEEE International Computer Performance and Dependability Symposium, IPDS 2000, Chicago, IL, USA, 27–29 March 2000; pp. 91–100.
28. Cotroneo, D.; Natella, R.; Russo, S.; Scippacercola, F. State-driven testing of distributed systems. In Proceedings of the Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, 16–18 December 2013; Proceedings 17; Springer: Berlin/Heidelberg, Germany, 2013; pp. 114–128.
29. Daran, M.; Thévenod-Fosse, P. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Softw. Eng. Notes* **1996**, *21*, 158–171. [CrossRef]
30. Ng, W.T.; Chen, P.M. The design and verification of the Rio file cache. *IEEE Trans. Comput.* **2001**, *50*, 322–337. [CrossRef]

31. Thongtanunam, P.; Tantithamthavorn, C.; Kula, R.G.; Yoshida, N.; Iida, H.; Matsumoto, K. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. In Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, 2–6 March 2015; Guéhéneuc, Y., Adams, B., Serebrenik, A., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2015; pp. 141–150. [CrossRef]
32. Ruangwan, S.; Thongtanunam, P.; Ihara, A.; Matsumoto, K. The impact of human factors on the participation decision of reviewers in modern code review. *Empir. Softw. Eng.* **2019**, *24*, 973–1016. [CrossRef]
33. Mohanani, R.; Salman, I.; Turhan, B.; Rodríguez, P.; Ralph, P. Cognitive biases in software engineering: A systematic mapping study. *IEEE Trans. Softw. Eng.* **2018**, *46*, 1318–1339. [CrossRef]
34. Liu, S. Software Construction Monitoring and Predicting for Human-Machine Pair Programming. In Proceedings of the Structured Object-Oriented Formal Language and Method—8th International Workshop, SOFL+MSVL 2018, Gold Coast, QLD, Australia, 16 November 2018; Duan, Z., Liu, S., Tian, C., Nagoya, F., Eds.; Revised Selected Papers; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2018; Volume 11392, pp. 3–20. [CrossRef]
35. Campbell, G.A. Cognitive Complexity—A new way of measuring understandability. *SonarSource SA* **2018**, 10.
36. Jia, Y.; Harman, M. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **2010**, *37*, 649–678. [CrossRef]
37. Offutt, A.J.; Pan, J. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test. Verif. Reliab.* **1997**, *7*, 165–192. [CrossRef]
38. Tai, K.C. The tree-to-tree correction problem. *J. ACM (JACM)* **1979**, *26*, 422–433. [CrossRef]
39. Zhang, K.; Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.* **1989**, *18*, 1245–1262. [CrossRef]
40. Zhang, K. A constrained edit distance between unordered labeled trees. *Algorithmica* **1996**, *15*, 205–222. [CrossRef]
41. Oladele, R.; Adedayo, H. On empirical comparison of checklist-based reading and adhoc reading for code inspection. *Int. J. Comput. Appl.* **2014**, *87*. [CrossRef]