Evaluating Code Clone Detection and Management: A Comprehensive Comparison among different Techniques and Tools along with some Effective Future Directions

Soily Ghosh Sneha
American International University-Bangladesh
Dhaka, Bangladesh
soilyghoshsneha@gmail.com

Sadia Niha
American International University Bangladesh
Dhaka, Bangladesh
nihaislam202@gmail.com

Dr. Md. Manzurul Hasan
American International University-Bangladesh
Dhaka, Bangladesh
manzurul@aiub.edu

Abstract

In today's software development, Code copying is a major issue in that adds to the workload and can degrade products. The best methods for identifying and dealing with copied code are examined in this study. We initially start by defining code copying, outlining the many types of copied code, and discussing the impact it has on software quality. Next, we examine how various tools and techniques can identify copied code and how to use them to locate and organize code that is identical. The study's second part focuses on controlling copied code and identifies the most effective techniques and resources for this task. Given that not all copied code can be handled in the same way, this paper aims to provide insights on the optimal course of action for various types of duplicated code. Our goal is to enhance software quality and make it easier to manage by linking the process of identifying copied code with solutions.

CCS Concepts

• **Software and its engineering → Clone maintenance system**; • **Software notation and tools**; • **Software maintenance tools**;

Keywords

Code Clone, Software Maintenance, Code fragment, Clone tools.

ACM Reference Format:
Soily Ghosh Sneha, Sadia Niha, and Dr. Md. Manzurul Hasan. 2024. Evaluating Code Clone Detection and Management: A Comprehensive Comparison among different Techniques and Tools along with some Effective Future Directions. In *3rd International Conference on Computing Advancements (ICCA 2024), October 17–18, 2024, Dhaka, Bangladesh.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3723178.3723206

1 Introduction

Code cloning is a common practice for developers to take code used somewhere else and copy it for their own purposes and apply small changes. In general, they perform cut and paste operations. The technique of code reuse makes code reuse possible. Parts of code, whether modified or unmodified, which are reused fall under the clone of the original source. In a software business, you will copy or clone the code for several reasons, it has both good and bad significance [22]:

(1) **Time Savings**: Cloning saves time, especially under deadlines, by reusing tested code instead of creating new code.
(2) **Lack of Knowledge**: Unfamiliar developers may duplicate features they are unaware of in the source.
(3) **Avoiding Dependencies**: Replicating code can help minimize the impact and complexity of changes by avoiding dependencies.
(4) **Temporary Fix**: Cloning offers a temporary solution, frequently with the goal of refactoring later.
(5) **Code Breaking Fear**: To make changes to existing modules without running the risk of creating bugs, developers may duplicate code.
(6) **Perceived Uniqueness**: When there are similar solutions, developers may repeat code because they believe their problem is unique.
(7) **Organizational Issues**: In large teams, inadvertent duplication may result from a lack of cooperation.
(8) **Simplicity in Management**: Duplication may be easier than creating abstract components in complex projects.

A software's maintainability is a quality that makes it easier to modify, correct or update program, after delivery. The source code has been more important for the maintenance of the software. When you write a code well, it is easy to maintain. While maintainability can be measured, the cost is often high. Systems that are hard to maintain are costly and extensive. Code cloning and similar techniques might make the situation worse for the program involved. This frequently results in unnecessary or duplicate code. Attempting to fix the code later can make things more serious than they already are. An easily maintainable software system is essential since more than 80% of the total software cost is likely to be incurred on maintenance [9].
Copying existing code can be harmful to the quality of the software because it can increase code size and impair maintenance. When more than one IT person works on the same thing, we often get confusion. The aim of this survey paper is to examine the difficulties associated with software engineering code clone detection and management. After giving an overview of the several kinds of code cloning and how they affect software quality, we assess the state-of-the-art detection techniques and resources. In order to

ICCA 2024, Dhaka, Bangladesh

ACM ISBN 979-8-4007-1382-8/24/10
https://doi.org/10.1145/3723178.3723206

improve software maintainability and lower related expenses, our analysis attempts to provide useful tactics for handling code clones.

1.1 Research Objectives

The main objective of this study is to evaluate various tools and techniques for handling and identifying code clones. It assesses a few approaches to determine their advantages, disadvantages, and application to a specific situation. A comparison table can help developers, researchers, and practitioners in choosing the correct tools and techniques for their specifying needs. That's the crux of the study, which tackles some key questions:

(1) **How to detect and mitigate code clones?**
The question explores the various ways to both detect and manage code clones.
(2) **What are the differentiators for the approaches for different tools at different times?**
In this question, the effectiveness of each strategy is compared across environments and tools.
(3) **What are the benefits and drawbacks of each tool?**
This question identifies each tool's advantages and disadvantages.
(4) **In what situations is each tool best appropriate?**
This question specifies the conditions in which each tool performs best.
(5) **What are the dominant trends and possible avenues in this field?**
This inquiry looks at current developments and future directions in code clone detection and management.

By answering these questions, we intend to further code clone detection and management, which will enhance software quality and maintainability.

2 Background

2.1 Clone Classification and Code Similarity

Code segments that are extremely similar to or identical to one another are known as code clones. These may take place across several software projects or inside a single project. Code similarity measures the degree of similarity between these parts based on syntactic and semantic similarities (near-miss clones) or direct copying (precise clones). Clone classification frequently centers on a few types: [8]:

(1) **Exact Copies:** With the exception of minor adjustments like whitespace, formatting, or comments that have no bearing on how the code functions, exact copies are bits of code that are exactly the same [27].
(2) **Renamed Copies:** While the structure of these code parts is essentially the same, the variables, actual language, data types, formatting, and comments have been altered.
(3) **Modified Copies:** These are copied sections of code that have been modified, such as by adding or removing lines or rearranging existing lines.
(4) **Similar Functioning Copies:** Although these pieces of code perform the same function, their appearance is different due to their various writing styles [27].

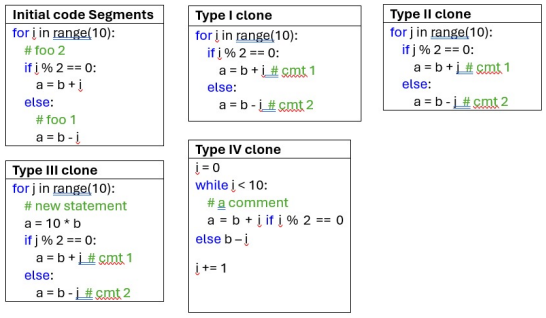

Figure 1: Examples of different type of code clones [31]

2.2 Primary Reasons for Code Similarity

There are several causes of code similarities, such as:

- **Code reuse:** To save time or duplicate functionality, developers may copy and paste functioning code segments.
- **Standardized Algorithms:** Some logic sequences or algorithms are frequently utilized in a variety of programs.
- **Architectural Patterns:** Code from several applications may share structural similarities as a result of comparable design patterns or architectural solutions.
- **Maintenance Procedures:** Similar features or defects may be introduced in multiple parts of a code base during the maintenance of software, which gives rise to similar segments of code.

2.3 A Refactoring Perspective on Clone Categorization

By identifying areas for code improvement, clone classification improves maintainability, lowers mistakes, and maximizes efficiency. Code structure is made simpler through refactoring without compromising functionality. Refactoring clones can be done by:

- **Extraction Techniques:** Similar blocks of code can sometimes be extracted into a separate method, which will decrease code duplication and allow its further reuse.
- **A Novel Solution to Data:** Using polymorphism or template programming, comparable processes can be handled more generically for Type IV clones.
- **Utilizing Design Patterns:** Similar code can be encapsulated within a class hierarchy using design patterns like Factory, Strategy, or Command, which encourage flexibility and reuse.

3 Related work

Roy et al.[23] In code cloning its difficult to highlight attention and pointing out that the developers continuously repeat those codes to maintain the strict deadlines and productive set goals which increases the code duplication. Lack of domain knowledge also contributes to this practice, which causes developers to duplicate well-known solutions. Repetitive patterns are a result of limitations in some programming languages, such as the lack of generics or inheritance. Because it becomes more difficult to manage duplicates and ensure consistency across clones, these methods make

maintenance more onerous. Because complicated Type III and IV clones are difficult to detect with current technologies, the study emphasizes the necessity of strong clone management systems.

Key elements of code clone management, such as discovery, analysis, annotation, documentation, tracking, evolution, and refactoring, were examined by Roy et al.[14] in 2014. The report provides a thorough review of existing approaches and identifies research gaps for better tactics by classifying clone refactoring and tracking methodologies and tools. In contrast to Roy et al.[17], this study offers a more thorough examination and comparison of tracking and refactoring techniques. In a 2006 study, Koschke et al.[16] examined the causes, effects, evolution, and refactoring techniques of code cloning. The study offers a thorough assessment by contrasting clone detection methods and technologies. It emphasizes how crucial it is to comprehend various strategies in order to properly handle code clones.

Roy et al.[14] presented a framework for classifying clone detection tools and methodologies. The techniques were categorized by editing situations that resulted in various clone types. In 2013, Rattan et al. [21] researched the effects of cloning on software quality, including management and visualization advantages, and compared clone detection tools. By restructuring the first two classes but eliminating partial clones, Yu and Ramaswamy et al.[30] classified Linux clones using CCFinderX as freestanding (39%), cross-cutting (24%), and partial (37%). Depending on the type and location of the clones, Schulze et al.[1] suggested either object-oriented or aspect-oriented refactoring, with AOR being preferred for scattered clones.

Sheneamer and Kalita [26] examined clone detection techniques, tools, and problems, whereas Kapdan et al. [7] investigated the capacity of clone detectors to detect structural clones. Recently, Roy and Cordy compared clone restructure and monitoring techniques in order to analyze clone detection benchmarks. In accordance with Gamma et al.'s strategic design pattern, which extended and streamlined code management, Balazinska et al.[1] automated method clone refactoring in JDK 1.1.5 using the Clone Re-engineering Tool (CloRT). RASE is an automatic clone eradication program developed by Meng et al.[25]. Common code fragments are identified by RASE, which also refactors by adding return objects, parameterizing differences, and generating classes and methods. It demonstrated the effectiveness of automated refactoring by successfully reworking clones in a significant percentage of tested method pairs and groups. An overview of related research work is shown in Table 1.

By giving a comprehensive review of cloning, including definitions, types, and its effects on software quality, this paper aids in the detection and control of code clones. We examine and contrast current instruments and approaches, emphasizing their advantages and disadvantages. In order to identify the most effective tools for various clone kinds, the study also looks at methods for handling cloned code. We provide insights to assist developers and organizations in enhancing the quality and maintainability of software by tying detection to useful management strategies. Through an analysis of current procedures and recommendations for better clone management, this study advances software development.

Table 1: Summary of Related Work

Study	Challenges Addressed	Contribution
Roy et al. [14, 23]	Code reuse due to tight deadlines, productivity metrics, lack of domain expertise, repetitive coding patterns, maintenance issues; Refactoring and tracking code clones, assessing current methods; Lack of a comprehensive framework for clone detection.	Root causes and maintenance challenges of code cloning; Compilation and categorization of clone refactoring studies, techniques, and tools; Framework for categorizing and comparing clone detection methods and tools.
Koschke et al. [16]	Reasons for cloning, consequences, evolution, refactoring methods.	comparison of tools and techniques for clone detection.
Rattan et al. [21]	Code clones' effects on software quality	A comparison is made between the impact of clone detection technologies on software quality.
Kapdan et al. [7]	Recognizing structural copies	The ability of clone detectors to identify structural clones.
Sheneamer and Kalita [26]	challenges with the methods and technologies used for clone detection today.	Review of clone detection methods, tools, and challenges.
Balazinska et al. [1]	Method difficulties, such as repetitive work and manual refactoring.	automation of the refactoring process using CloRT.
Meng et al. [25]	Identifying and repairing common code fragments.	development of RASE, a program for automatically removing code clones.

4 Research Methodology

4.1 Clone Management

A clone management system efficiently handles code clones—identical or similar fragments within or across systems. While code reuse has benefits, poor clone management can increase maintenance costs, cause inconsistent bug fixes, and add technical debt [22]. The system supports detecting, monitoring, documenting, and modifying clones, with strategies like refactoring, removing redundant copies, and documenting necessary clones. This approach mitigates cloning's downsides and improves code usability and reliability. Key components include detection, classification, tracking, visualization, refactoring support, and collaboration tools.

4.2 Impact of Cloning on Software Quality

Cloning software has an impact on quality factors like dependability and maintainability, increases technical debt, or results in long-term expenses from bad coding techniques. Larger clones reduce dependability, although cloned modules are 1.7 times more

dependable than non-cloned ones, according to a research [18] on a legacy system. Cloning complicates maintenance; cloned modules need more frequent revisions, especially as clone size increases. This evaluation highlights cloning's broader impacts, guiding sustainable software engineering practices.

4.3 Clone Management Activities

To lower maintenance expenses, mistakes, and inconsistencies, clone management supervises and maintains code clones, which are identical or comparable code pieces. **Detection**, which finds clones in the codebase, starts the workflow. The clone information are recorded by **Documentation** and then tracked over time by **Tracking**. **Visualization** uses graphics to make clones easier to grasp. After evaluating the impact of clones, **Analysis** generates **Recommendations** for management solutions. Through **Refactoring Operations**, **Refactoring Verification**, and **Refactoring Scheduling**, these tactics are implemented, confirmed, and scheduled. Lastly, **Prevention** seeks to reduce cloning in the future. The order and interconnectedness of these processes are shown by arrows.

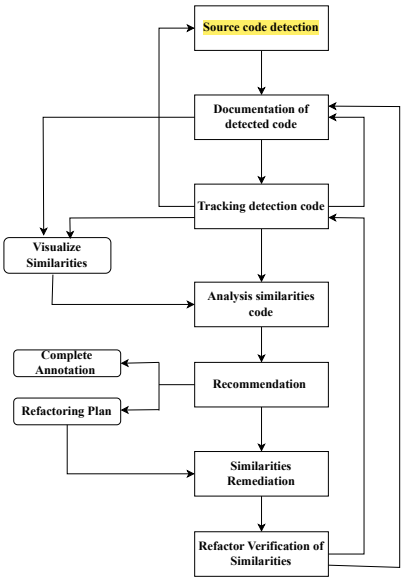

Figure 2: The process of working clone management operates[32]

4.4 Detection of Clones

Clone detection finds similar or repeating parts of software code, whether they are directly copied or only slightly altered. By pointing out places that can be simplified or eliminated, this preserves the efficiency and quality of the code.

4.5 Token-Based Approaches

Software token-based clone detection is a methodical procedure that finds code parts that are similar or identical throughout a big codebase. The first step in this process is to convert the source code into a series of tokens that stand in for the fundamental components of the code, including operators, keywords, and identifiers. Then, the tokens are checked against the presence of patterns that signify the existence of copied or cloned code. This approach successfully detects exact or near duplicate code fragments in sections of different software projects [5]:

(1) **Lexical Analysis:** From this, the original source text is segmented into basic items called tokens: Operators, literals, keywords, and identifiers are examples of such tokens created by lexical analysis. **Process:** It breaks down the source code linewise-to-token using lexer rules of the computer language under investigation. This step finalizes the syntactic components that constitute the logical structure of the code itself. All supplementary items such as white space and comments are discarded.

(2) **Token Transformation:** Tokens are to be standardized in order to recognize clones of analogous code structures with minimal variations (such as variable names or formatting) [23].
Process: Redundancy Transformation Rules: Because it is a sequence of tokens, a transformation is performed either by normalizing variable names changing literals into a general token or simplifying manifest expressions, making them less diverse. However, this doesn't affect the logic or functionality of the code as such. **Use of a Different Parameter:** To abstract further the code sequences and make it easier to match similar code snippets, even if they look different, one could use a placeholder token instead of an identifier or literal.

(3) **Finding-matches:** Match detection is considered finding identical or strikingly similar code segments in their tokenized form across a whole codebase. **Process:** Converted token sequences are analyzed to determine their relationships with possible similar pattern or sequence of tokenization. Approaches like suffix trees and hash-based methods can quickly find and catalog these matches. Every pair or collection of matching codes found gets recorded, typically with some reference to where they were found in the original code.

(4) **Remapping and Formatting:** Translating into the found matches from tokenized and manipulated code to restore the original context of the source code files. **Process:** Original lines and files of the source code are mapped with the positions of the clones in the modified token sequence. This is often carried out along with reinstating formatting and comments, to make the output readable and interpretable for the developers who want to use this information to rearrange the code.

(5) **Result and Evaluation:** To provide actionable insights from these cloned data to develop and project management. **Process:** Along with the information regarding the location and the amount of code duplication, the final report will include metrics or visuals (e. g., graphs or scatter plots) which will help to prioritize the refactoring efforts. This output may be useful in technical debt management, decision making for refactoring, and code maintenance.

4.6 Text-Based Approaches

A text-based process for clone detection identifies similar or duplicate source code from all those datasets. In the first step, a large number of source codes are collected. Normalize the format by preprocessing so that they can be compared with each other. The information is indexed to enable efficient search. To find related code fragments, queries are made on the index, often using tokenized code. Analysis finally seeks to identify potential clones using similarity metrics. This method inspects the source code text to discover exact and near-miss clones.

4.7 Metric-Based Approaches

It discovers similar code by retrieving metrics from programming entities such as functions, classes or statements. To assess the structure and behavior of code, some preliminary metrics are computed, which are reported on comparison of values to indicate objects that have like metrics, possibly as clone pairs; this subsequently helps in maintenance and optimization, usually by systematic discovery of code duplication through identification of simplification or reuse opportunities [8].

4.8 Abstract Syntax Tree (AST) Based Approaches

The first stage of the process of identifying code clones in the AST-based method involves a language-specific parser constructing an Abstract Syntax Tree (AST) that structurally represents the syntax of the program. The AST is subjected to a tree matching method, which looks for sub-trees that are identical or similar in order to detect clones. The corresponding portions of the source code are identified as a clone pair when two matching sub-trees are discovered. This technique efficiently detects duplicate code by examining the program's structural patterns.

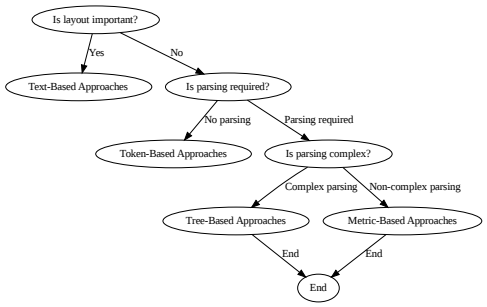

Figure 3: Evaluating Different Code Detection Approaches

Code clone detection approaches are compared by method and efficacy in the figure-3. Code arrangement is the main emphasis of **Text-based approaches**, which minimize false positives but overlook method-level variations. Though they may have false positives and detect beyond syntax, token-based approaches avoid parsing, adapt to new languages, and identify duplication well. Complex parsing is required for **Tree-based approaches** to provide high precision for syntactic clones that are suitable for refactoring. For function-level analysis, **Metric-based approaches** employ coarse-grained abstractions, which facilitate refactoring but lack statement-level information. Each method strikes a distinct balance between granularity, flexibility, and accuracy.

Code clone detection tools are categorized by approach (text-based, tree-based, token-based, and metric-based) in Table 2. This includes information on the tools' source code representation, supported languages, clone relation, and clone types found. **Text-based tools** (e.g., SDD, Duplo) detect Type-1 clones (exact copies), work with lines of code, and support languages like Java and C++. Abstract Syntax Trees (ASTs) are used by **Tree-based tools** (e.g., Deckard, CloneDr) to identify Types I–III clones in languages such as Java and C. **Token-based tools** (such as CP-Miner and scFinder) detect Type I-III clones and support many languages by representing code as tokens. **Metric-based tools** (e.g., Mayrand et al.) detect all clone types, from exact to highly modified copies, support Java and C++, and analyze functions and methods.

Working based some selected best tool suggested in table 3 because of:

Duploc that support, Java, C, C++, C#, and VB.NET are just a few of the many programming languages. Duploc would be a good option if you require a tool that can handle several languages. **CCFinderX** supports a number of languages, such as Cobol, C#, Java, C, and C++. Duploc is a complete option for token-based clone detection because of its support for several languages and its explicit handling of type-I, type-II, and type-III clone relations. **Deckard** and **cloneDr** Both are capable of handling C and Java. Nevertheless, cloneDr can identify type-I, type-II, and type-III clones and also supports C++ and Cobol. Therefore, if you are interested in a wider variety of clone kinds, cloneDr may be the more reliable tool in the tree-based category.

4.9 Useful resources for identifying code clones across various code bases

An overview of the best code clone detection tools for different project requirements is given in Table 4, with an emphasis on use case suitability and performance. This assessment improves the effectiveness of clone identification and management by assisting developers in choosing the optimal technologies for their requirements. Additional tool suggestions catered to different project requirements and scenarios are provided in Table 5.

5 Result and Discussion

A Comparison of Clone Detection Techniques Three main approaches are chosen for comparison: text-based, token-based, and tree-based approaches. Each method uses different strategies, with differing levels of precision and granularity, to find code duplication.

5.1 Suggested tool for Text-Based Approach: Duploc

The purpose of Duploc is to find code clones, or duplicates, in a codebase. Its primary attributes and benefits:

Table 2: Comparative Analysis of Tools

Approaches	Format for code layouts	Tool	Language Supported for the Tools	Types of Clone
Text Based	Lines	SDD [11]	Java, C++	Type-I
		Duploc [4]	Java, C, C++, C#, VB.NET	Type-I
		PMD-CPD [3] [19]	Java, C, C++, PHP, PLSQL, JSP, Ruby	Type-I
		Dude [29]	Java, C, C++, C#	Type-I
Tree Based	AST	Deckard [6]	Java, C	Type-I, Type-II, Type-III
		CloneDr [2]	Java, C, C++, Cobol	Type-I, Type-II, Type-III
		AST-Analyzer [13]	Java, C	Type-II, Type-III
		CodeDetector [10]	Java	Type-I, Type-II
Token Based	Tokens	ccFinder [5]	Java, C, C++	Type-I, Type-II
		SolidSDD [28]	Java, C, C++, C#	Type-I, Type-II, Type-III
		Siames [20]	Java	Type-I, Type-II, Type-III
		CP-Miner [12]	Java, C, C++	Type-I and Type-II
		CCFinderX	Java, C, C++, Cobol, C#	Type-I, Type-II, Type-III
		SourcererCC [24]	Java, C, C#	Type-I, Type-III
Metric Based	functions, classes or statements	Software metric such as Names, Layout, Expressions, Control Flow [15]	Java, C++	Type-I, Type-II, Type-III

Support for Language: It is appropriate for projects using various languages because it supports a large variety of programming languages. This adaptability is advantageous in a varied growth setting.

Table 3: Overview of selected best tool according to working techniques

Recommended Tool	Working Based Techniques
Duploc	Text Based
CCFinderX	Token Based
Deckard	Tree Based
cloneDr	Tree Based

Table 4: Suggested Tools for Code Clone Detection in Particular Specific Cases

Case of Use	Suggested Tool	Reason
Tool Suggested for Use Cases	Simian or CCFinderX	Excellent performance for simpler codebases with sufficient accuracy.
Large-scale, very accurate projects	Deckard	Accurate, if slower, for different types of clones.
Projects Requiring Functional Clone Detection	iClones	Slowest, but best at identifying functionally similar clones.
Projects That Require a Balance Between Accuracy and Speed	NiCad	Maintains a balance between text comparison and metrics analysis.

Table 5: Application-specific Tool Suggestions

Application Context	Dedicated Tool
Large Codebases	NiCad
High Accuracy Needs	Deckard
Small to Medium Project	Simian or CCFinderX

Method of Detection: Being a text-based tool, it probably checks code lines directly for duplication, which can be quicker and easier for some codebase types, especially when identifying Type I clones (identical copies minus comments and white space).
Usability: Because they don't involve parsing the source code into more intricate representations like tokens or abstract syntax trees (ASTs), text-based tools are frequently easier to use and are a useful option for rapid and simple analysis.

5.2 Suggested tools for Token-Based Approach: CCFinderX

CCFinderX examines the source code's tokens. The simplest components of a program are called tokens, and they include identifiers, operators, and keywords. Among its advantages are:

Strong Language Assistance: It is adaptable since it supports a number of languages, including less often used ones like Cobol.
Clone Types: It offers comprehensive clone detection capabilities by detecting type-I clones, which are exact copies; type-II clones, which are syntactically similar except for differences in identifiers, white space, literals, types, and comments; and type-III clones, which are copied fragments with additional modifications,

like added or removed statements.
Granularity: Token analysis allows CCFinderX to identify more detailed similarities between code fragments that text-based methods would overlook.

5.3 Suggested tools for Tree-Based Approaches: Deckard and CloneDR

Tree-based tools offer a representation of the hierarchical tree from the structure of the program employing the abstract syntax tree (AST) of the source code.

Deckard

Scalability: Deckard is designed to be both scalable and efficient so that it can detect clones in large software systems. **Clone Detection:** Both type-I and type-II clones can be detected by it. It is more reliable in identifying structural similarities than merely textual ones because to the use of AST. **Vectorization:** By vectorizing sub-trees in the AST, Deckard makes it possible to compare code segments effectively and helps identify related structures, even if they are not textually identical.

CloneDR

Extended Language and Clone Type Support: CloneDR supports C++ and Cobol in addition to Java and C. With its ability to identify type-I, type-II, and type-III clones, it offers a thorough examination across several similarity dimensions. **Precision:** CloneDR can accurately locate functional duplicates with the help of ASTs mechanism for structural clones that are not duplicate textually. **Reporting:** It tends to bring up very detailed reports about clones resulting and helping further studies and reworking into them.

A) The clone's stated output: There exist several techniques by which these tools incorporate detection and reporting of code clone methods which are provide in table-6. Basically, Duploc has been designed to detect exact clones by such a means as direct string comparison, as also is manifest from the fact that in it it shows identical code fragments along with the location information and similarity percentages. CCFinderX demonstrates its capacity to detect clones that are structurally similar but not textually identical by reporting clones by token sequences and providing clone type and similarity scores. Deckard and CloneDR provide comprehensive details about code structure and context by concentrating on structural similarities within ASTs.

B) Suitable: The complexity of the codebase and the kind of clones that need to be found determine which of these technologies are most appropriate. Duploc's lesser complexity makes it ideal for discovering precise clones or smaller codebases. Since CCFinderX tokenizes the code and checks token patterns, it is better suited for handling different clone kinds inside individual files. Deckard is scalable and appropriate for identifying intricate patterns in code because to its high complexity and AST comparison technique. Being the most sophisticated, CloneDR can handle a variety of languages and clone kinds, indicating that it is appropriate for thorough clone detection and refactoring across intricate and sizable codebases.

Table 6: An overview of the best tool chosen based on working approach

Tool	Output Description	Complexity Description
Duploc	Lists identical code segments along with their line numbers, file locations, and percentage of similarity.	Reduced complexity; makes use of straightforward hashing methods and direct string comparison. Ideal for precise clones or smaller codebases.
CCFinderX	Reports clones according to token sequences, along with the location in the code, clone type (Type 1, 2, and 3), and similarity score.	Tokenization and managing several clone kinds make it more complicated. involves comparing token patterns after source code has been parsed into tokens.
Deckard	Provides structural similarities in AST format together with context, such as surrounding functions or classes, locations, and a similarity score.	High level of complexity in AST sub-tree vector creation and comparison. involves mathematical computations for comparison as well as parsing into AST. scalable design.
CloneDR	Comprehensive reports that include locations, a structural sketch in the context of AST, matched constructions, differences, and recommendations for reworking.	Most intricate, requiring clone detection at different granularities, comparison transformations, and parsing into AST. manages a variety of clones and languages.

6 Some useful directions for the future

Developing a system that can identify clones and automatically recommend refactoring opportunities, perhaps with the help of artificial intelligence (AI) to recommend particular refactoring patterns depending on the circumstances around the clone occurrence.

A) Detection of Clone: Using approaches like textual, token-based, syntax tree-based, or semantic analysis, the system first finds code clones—segments of code that are identical or similar. **Clone Analysis for Refactoring:** By evaluating possible savings in complexity and expenses related to the refactoring process, the system evaluates found clones to ascertain whether refactoring is advantageous. **Benefits of Refactoring:** Calculates the complexity reduction and error-proneness decrease that would occur if the clone were refactored. **Refactoring Costs:** takes into account the work needed to validate and restructure the changes.

B) Refactoring Patterns Suggestion: The system uses AI to recommend suitable refactoring techniques according on the clones' kind and context. **Extraction Method:** It may be possible to extract

common code from several sites into a new method that is subsequently called from the original locations. **Methods of Pulling Up and Pushing Down:** For improved code reuse and hierarchy, functions that are similar across subclasses in object-oriented languages may be pushed down into one or more subclasses or pulled up into a parent class. **Pattern for the Template Method:** The template method design pattern may be used to rework similar methods that have the same steps but differ in some aspects of their implementation. **Set Up the Method:** By converting these constants into parameters, similar techniques that differ only by constant values can be parameterized.

C) Machine Learning Models: Historical refactoring data if examined, could help machine learning models identify the most successful pattern of refactoring and the clone types that tended to be refactored. **Training Information:** A dataset of code changes, including the before and after refactored versions of the code, is used to train the model. **Feature:** The clone types, code metrics, and previously used refactorings are the feature extraction metrics. **Model Training:** The best refactoring pattern for a particular kind of clone is predicted by supervised learning models. **Forecast and Recommendations:** The model predicts what would be the best refactoring pattern for new clones and recommends it to the developer.

7 Conclusion

The complex nature of code cloning has been explained by this thorough investigation, which has also outlined a range of management tools and detection strategies to accommodate various cloning situations. The subtleties of token, text, tree, and metric based detection approaches have been examined in this work. Each of these approaches has special benefits for locating and controlling code clones in various software systems. Our findings highlight how important good clone management is to preserving program quality and lowering technical debt. In addition to streamlining the clone detection process, the integration of sophisticated detection tools with IDEs and version control systems increases the effectiveness of ensuing refactoring initiatives. Future developments in machine learning algorithms and AI-powered technologies that can provide even more precise detection capabilities and more intelligent management solutions are expected to help the progress of clone detection and management. The creation of increasingly advanced tools that can manage big systems efficiently will be crucial as software systems continue to grow more complicated.

References

[1] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 2000. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 98–107.

[2] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.

[3] Shilpa Dang and Shahid Ahmad Wani. 2015. Performance evaluation of clone detection tools. *International Journal of Science and Research (IJSR)* (2015), 1903–1906.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 109–118.

[5] Katsuro Inoue. 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Annual report of Osaka University: academic achievement* 2001 (2002), 22–25.

[6] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[7] Mustafa Kapdan, Mehmet Aktas, and Melike Yigit. 2014. On the structural code clone detection problem: a survey and software metric based approach. In *Computational Science and Its Applications–ICCSA 2014: 14th International Conference, Guimarães, Portugal, June 30–July 3, 2014, Proceedings, Part V 14*. Springer, 492–507.

[8] Harpreet Kaur and Raman Maini. 2017. Performance Evaluation and Comparative Analysis of Code-Clone-Detection Techniques and Tools. *International Journal of Software Engineering and Its Applications* 11, 3 (2017), 31–50.

[9] Shahbaa I Khaleel and Ghassan Khaleel Al-Khatouni. 2023. A literature review for measuring maintainability of code clone. *Indonesian Journal of Electrical Engineering and Computer Science* 31, 2 (2023), 1118–1127.

[10] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.

[11] Seunghak Lee and Iryoung Jeong. 2005. SDD: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 140–141.

[12] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[13] Hongliang Liang and Lu Ai. 2021. AST-path based compare-aggregate network for code clone detection. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[14] Manishankar Mandal, Chanchal K Roy, and Kevin A Schneider. 2014. Automatic ranking of clones for refactoring through mining association rules. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 114–123.

[15] Mayrand, Leblanc, and Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*. IEEE, 244–253.

[16] Thilo Mende, Rainer Koschke, and Felix Beckwermert. 2009. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution: Research and Practice* 21, 2 (2009), 143–169.

[17] Manishankar Mondal, Chanchal K Roy, Md Saidur Rahman, Ripon K Saha, Jens Krinke, and Kevin A Schneider. 2012. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. 1227–1234.

[18] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*. IEEE, 87–94.

[19] Jaroslav Porubän et al. 2016. Preliminary report on empirical study of repeated fragments in internal documentation. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 1573–1576.

[20] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.

[21] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[22] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR* 541, 115 (2007), 64–68.

[23] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 18–33.

[24] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.

[25] Sandro Schulze and Martin Kuhlemann. 2009. Advanced analysis for code clone removal. (2009).

[26] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[27] Jeffrey Svajlenko and Chanchal K Roy. 2020. A survey on the evaluation of clone detection performance and benchmarking. *arXiv preprint arXiv:2006.15682* (2020).

[28] Lucian Voinea and Alexandru C Telea. 2014. Visual clone analysis with SolidSDD. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 79–82.

[29] Richard Wettel and Radu Marinescu. 2005. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 8–pp.

[30] Liguo Yu and Srini Ramaswamy. 2008. Improving modularity by refactoring code clones: A feasibility study on linux. *ACM SIGSOFT Software Engineering Notes* 33,

2 (2008), 1–5.

[31] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* (2023), 111796.

[32] Minhaz F Zibran and Chanchal K Roy. 2012. The road to software clone management: A survey. *Dept. Comput. Sci., Univ. of Saskatchewan, Saskatoon, SK, Tech. Rep* 3 (2012).