Towards Diverse Program Transformations for Program Simplification

HAIBO WANG, Concordia University, Canada
ZEZHONG XING, Southern University of Science and Technology, China
CHENGNIAN SUN, University of Waterloo, Canada
ZHENG WANG, University of Leeds, United Kingdom
SHIN HWEI TAN, Concordia University, Canada

By reducing the number of lines of code, program simplification reduces code complexity, improving software maintainability and code comprehension. While several existing techniques can be used for automatic program simplification, there is no consensus on the effectiveness of these approaches. We present the first study on how real-world developers simplify programs in open-source software projects. By analyzing 382 pull requests from 296 projects, we summarize the types of program transformations used, the motivations behind simplifications, and the set of program transformations that have not been covered by existing refactoring types. As a result of our study, we submitted eight bug reports to a widely used refactoring detection tool, RefactoringMiner, where seven were fixed. Our study also identifies gaps in applying existing approaches for automating program simplification and outlines the criteria for designing automatic program simplification techniques. In light of these observations, we propose SimpT5, a tool to automatically produce simplified programs that are semantically equivalent programs with reduced lines of code. SimpT5 is trained on our collected dataset of 92,485 simplified programs with two heuristics: (1) *modified line localization* that encodes lines changed in simplified programs, and (2) checkers that measure the quality of generated programs. Experimental results show that SimpT5 outperforms prior approaches in automating developer-induced program simplification.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

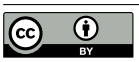
Additional Key Words and Phrases: Program Simplification, Program Transformation, Refactoring

ACM Reference Format:
Haibo Wang, Zezhong Xing, Chengnian Sun, Zheng Wang, and Shin Hwei Tan. 2025. Towards Diverse Program Transformations for Program Simplification. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE015 (July 2025), 23 pages. https://doi.org/10.1145/3715730

1 Introduction

Software systems have become increasingly more complex. To ease software maintenance, developers often dedicate significant time to simplify programs manually to reduce the code size or the number of lines of code (LOC) while preserving their functionalities — a process referred to as program simplification in this paper. In our study, developers mentioned in the pull requests that they are motivated to simplify programs to (1) clean up code, (2) improve readability, (3) reduce complexity, and (4) enhance reusability. However, as defects can be introduced when modifying programs by hand [21], manual program simplification is error-prone.

Authors' Contact Information: Haibo Wang, Concordia University, Montreal, Canada, haibo.wang@mail.concordia.ca; Zezhong Xing, Southern University of Science and Technology, Shenzhen, China, 12232384@mail.sustech.edu.cn; Chengnian Sun, University of Waterloo, Waterloo, Canada, cnsun@uwaterloo.ca; Zheng Wang, University of Leeds, Leeds, United Kingdom, z.wang5@leeds.ac.uk; Shin Hwei Tan, Concordia University, Montreal, Canada, shinhwei.tan@concordia.ca.



ACM 2994-970X/2025/7-ARTFSE015
https://doi.org/10.1145/3715730

Several techniques can be potentially used to automate program simplification: (1) rule-based approaches such as refactoring that restructures code via a predefined set of *syntactic program transformations* (transformations supported in refactoring engines or within the Martin Fowler's refactoring catalog [19] are usually called *refactoring types*), and (2) reduction-based techniques like delta debugging [80] that performs *semantic program simplification* by leveraging test executions to check if removing parts of the input program leads to smaller semantically-equivalent programs.

Program transformations that lead to reduced LOC play important roles in (1) reducing code size for software maintenance, and (2) program reduction or debloating component of existing automated techniques (e.g., debugging). The importance of removing unnecessary code for maintenance is illustrated in a recent study in Meta that showed it helps to respect users' privacy expectations, and allows engineers to work efficiently [56]. The framework proposed in the study used lines of code to measure code assets, and showed that removing unnecessary code enable the deletion of terabytes of data, saving over one megawatt of compute power as the corresponding processing pipelines can be removed to cut costs. Meanwhile, automated reduction techniques have showed promising results in producing reduced programs that can be used as inputs for various tasks, including debugging [79], test case simplification [80], and enhancing the understanding of features in code models [53]. Hence, understanding the set of program transformations that lead to reduced LOC is essential as it can potentially enhance these important tasks by producing more diverse forms of reduced programs.

Although refactoring practices [5] and deletion-based approaches [56, 80] have been extensively studied, there is rarely any study that focuses on obtaining a taxonomy of program transformations that lead to reduced LOC. This is mainly due to: (1) the misconception that program simplification is simply a subset of existing refactoring types (in fact, our study shows that some commonly used transformations for program simplification do not belong to any of the supported refactoring types. One example is the "Replace with equivalent API" transformation that replaces a block of code *Block* with an API performing similar task as the given code *Block*, which has not been supported by any tools as it is infeasible for rule-based tools to search for the corresponding equivalent block of code for the replacement), and (2) as deletion-based approaches directly remove redundant code, it is the most straightforward way to achieve the goal of obtaining reduced programs (in contrast, our evaluation shows that using diverse program transformations can achieve even more reductions than a deletion-only baseline).

To fill in the gaps of prior studies and techniques, we present the first study of the characteristics of developer-induced program simplification in open-source software (OSS) projects in GitHub. We called the transformations derived in our study "*developer-induced program simplification*" because (1) they are deemed to be program simplifications in the developers' perspective (commits with "simplify" keyword), and (2) developers explicitly include and discuss these simplifications in pull requests, indicating their importance. Our study investigated *the types of transformations used and the motivations behind developer-induced program simplification* by answering the questions below:

RQ1 What are the transformations frequently used in developer-induced program simplification?
RQ2 What are the motivations behind developer-induced program simplification in OSS projects?

As refactoring and developer-induced program simplification aim to improve code while preserving program semantics, we investigated the feasibility of reusing prior refactoring tools to automate developer-induced program simplification by designing the research questions below:

RQ3 What are the program transformations used in developer-induced program simplification that are covered by existing refactoring types?
RQ4 How effective are prior tools for refactoring detection and refactoring automation in supporting developer-induced program simplification?

Based on the findings of our study, we propose SimpT5, a tool that automates developer-induced program simplification by integrating (1) syntactic program transformations via a diverse set of transformations and (2) semantic program simplifications that use tests to check for test-equivalent relations. In summary, we made the following contributions:

- To the best of our knowledge, we present the first study of the program transformations used by developers and the motivations that drive developers during developer-induced program simplification in pull requests (PRs). The key findings of our study include (1) there are diverse types (26 types) of transformations used in developer-induced program simplification, (2) several frequently used types have not been covered by any prior approaches (e.g., replacing a code snippet with an equivalent API call, and simplifying expressions), (3) among the supported refactoring types, prior refactoring detection engine is still limited due to complex rule design, (4) prior auto-refactoring tools fail to automate many simplifications, indicating the need for a new tool with a richer set of transformations. As a result of our study, we submitted eight bug reports to RefactoringMiner [64] where seven have been fixed. Our proposed transformations have also been integrated in RefactoringMiner (No. 100-102 in [64]).
- We introduce SimpT5, a simplification framework based on pretrained large language model with two heuristics: (1) *modified line localization* (encoding the line-to-be-simplified into the models), (2) checkers that evaluate the quality of generated programs via static metrics observed in our study. Instead of generating the simplified programs directly without providing guidance in the location, our intuition behind modified line localization is that *some locations may have code smells which need simplifications* so encoding the location information may help in guiding the search for the program transformations to select for the simplification.
- We propose SimpliBench, a benchmark containing 37846 code simplification commits from 25022 open-source Java projects. We also derive a *valid* dataset from this benchmark, which contains 329 simplification commits across 307 projects. The *valid* dataset contains commits that are compilable with tests to validate the quality of generated programs. Our evaluation on SimpliBench shows that SimpT5 generates more simplified programs (that are exactly the same as ground truth) with higher quality than existing approaches.

2 Related Work and Problem Formulation

Table 1 highlights the differences between our work and prior techniques that produce programs of reduced sets. The problem of *automated program simplification* has been widely studied since 1965 [47]. We briefly discuss prior definitions of program simplification below:

Syntactic Program Simplification. According to [47], syntactic program simplification refers to *"simplifications which depend on the form of the program only, i.e. can be detected and proved to lead to an equivalent program without knowledge of what the program is supposed to do"*. This approach focuses on using a set of transformations known to preserve program equivalence. It is aligned with rule-based techniques (such as refactoring) in Table 1, where equivalent-preserving transformations are applied to enhance program design [19]. Therefore, we define ***syntactic program simplification*** as a way to obtain programs with fewer lines of code through semantic-preserving transformations. Typically, these techniques use static analysis to check for equivalence.

Refactoring, a representative of syntactic program simplification has been widely studied where most studies focus on the refactoring practice [50]. To reduce the time and effort in manual refactoring, many automated refactoring approaches have been proposed [43]. Several approaches use refactoring to support code review [66].

Semantic Program Simplification. Several automatic simplification techniques *rely on test executions (e.g., test outcome) to check for behavioral equivalence to produce smaller programs* (we

Table 1. Differences with relevant work that aim to produce reduced programs.

Approach	Simplification Goals	Transformations	Simplification Technique	Tests
Delta Debugging^D	Debugging, test input minimization, feature isolation	Deletion	Delta debugging [79, 80], hierarchical delta debugging [27, 44]	Need to run tests
Program Reduction^D	Test case minimization	Deleting/Replacing bug-irrelevant elements	Perses [62], Vulcan [78]	Need to run tests
Program Debloating^D	Reduce attack surfaces	Deletion	Combine with delta debugging [24], Dependency Analysis [52]	Need to run tests
Program Slicing^D	Debugging, testing	Deletion*	Dependency Analysis [6, 76]	May/May not run tests
Simplication in GP^R	Enforce explanability of models	Rule-based Transformations	Simplification Rules (e.g., algebraic, numerical) [36, 75]	May/May not run tests
Refactoring^R	Software maintenance	Rule-based Transformations	Rules based on refactoring types [66, 67, 80]	May/May not run tests
Our Work	Software maintenance	Diverse Transformations	Based on deep learning and LLMs	Need to run tests

Approaches marked with ^{D} denotes deletion-based approaches. Approaches marked with ^{R} denotes rule-based approaches.
*We consider program slicing as deletion-based as it removes statements without dependency from the original program.

call them ***semantic program simplification***) [1, 39, 79]. Semantic program simplification has been applied in many domains (e.g., simplifying event sequences [31], and deobfuscation [39]).

In terms of transformations used, these techniques either rely on deletions (delta debugging) or obtain a reduced list of statements by focusing on a slicing criterion (dynamic slicing). We call them deletion-based approaches in Table 1. Several techniques use limited types of transformation (e.g., mutation and crossover operators for pixel shader simplification [60], *remove* and *move* for debugging concurrent programs [30], replacing identifiers and subtrees of parse tree for program reduction [78]). These techniques usually rely on a variant of (1) delta debugging [79], and (2) dynamic slicing [1, 39]. Delta debugging increasingly deletes smaller parts of the input file and run tests to check if the simplified input changes the test outcome [79]. Dynamic slicing focuses on "all statements that actually affect the value of a variable occurrence for a given program input" [1]. Our work is different from dynamic slicing or its variants (e.g., observation-based slicing [6]) in that: (1) slicing-based techniques isolate feature-related elements (e.g., statements) under certain criteria, whereas we focus on the program itself to obtain a smaller program (reduced LOC); (2) we obtain various behavior-preserving transformation types that beyond mere deletion for simplification. *Our set of transformations can serve as the basis for providing more diverse candidates for delta debugging-based techniques to facilitate debugging.* Several researches focus on specific aspects of semantic program transformations, e.g., a prior work [16] studied the actual and potential usage of Java language features. Another study [54] mined API migration rules from PRs. Gil et al. [22] introduced an Eclipse plugin named Spartanizer that helps developers apply refactoring. Higo et al. [25] constructed a dataset of functionally equivalent Java methods using automated test generation techniques. Different from these techniques, we focus on transformations that could result in simplified programs.

Large Language Models for Program Transformation. Large language models (LLMs) have shown promising results in code-related tasks. Most prior learning-based approaches either focus on tasks like method name recommendations [49], code smell detection [2], bug fixing [81], reducing bug-triggering programs [82] (via five predefined transformations) or proposing new objectives to improve the effectiveness of pre-training for LLMs of code [9]. The most relevant techniques to us are the two general-purpose code transformation models: TufanoNMT [68] and AutoTransform [63]. Different from prior approaches, SimpT5 focuses on generating simplified programs with diverse transformations.

During our study, we needed a clear definition for what we consider a refactoring, and program simplification. We define them as follows:

Definition 1 (Refactoring). *Refactoring is a set of equivalent-preserving program transformations that rely on a predefined set of* ***syntactic*** *program transformations.*

Table 2. Taxonomy of program simplification in studied OSS projects.

Category	Sub-category	Description	Refactoring	PRs (#/%)	Total (#/%)
(T1) Control logic	(T1.1) Simplify method return	Simplify program logic related to method return value	Y	53/13.87	151/39.53
	(T1.2) Simplify boolean and algebraic expression	Simplify boolean and algebraic expression by rules	N	32/8.42	
	(T1.3) Use foreach in loop^{L}	Foreach loops can avoid potential off-by-one errors and reduce lines	Y	15/3.93	
	(T1.4) Merge conditional	Merge if statements that resulting action is the same	Y	13/3.40	
	(T1.5) Ternary conditional operator^{L}	Simplify conditional statements via ternary conditional operator	Y	11/2.88	
	(T1.6) Restructure conditional branches	Replace complex conditional branches by using enum, polymorphism, and etc.	Y	9/2.36	
	(T1.7) Replace with pipeline	Chain operations together by stream pipelining	Y	6/1.57	
	(T1.8) Replace variable with attribute	Replace variables in method with attributes	Y	6/1.57	
	(T1.9) Catching Multiple Exception Types^{L}	Merge multiple catch together if they contain duplicated code	Y	5/1.31	
	(T1.10) Change return type	Change method return to void and delete statements	Y	1/0.26	
(T2) Extraction	(T2.1) Extract method	Extract code blocks as methods to improve reusability	Y	75/19.63	120/31.41
	(T2.2) Extract variable	Extract common variables	Y	40/10.47	
	(T2.3) Consolidate duplicate conditional fragments	Pull up common head or pull down common tail of conditional to reduce duplication	Y	5/1.31	
(T3) Deletion	(T3.1) Remove unnecessary code	Remove duplicated or unneeded code	N	47/12.30	63/16.49
	(T3.2) Remove unused imports	Remove unused imports	N^{T}	11/2.88	
	(T3.3) Remove dead code	Remove dead code blocks that cannot be visited	Y	5/1.31	
(T4) API	(T4.1) Replace with equivalent API	Replace code block with semantically-equivalent APIs	N	62/16.23	62/16.23
(T5) Inline code	(T5.1) Inline variable	Inline temporary variables that are only used once	Y	38/9.95	47/12.30
	(T5.2) Inline method	Inline simple small method that are only used once	Y	9/2.36	
(T6) Lambda	(T6.1) Use lambda^{L}	Simplify using lambda expression	Y	44/11.52	44/11.52
(T7) Others	(T7.1) Use diamond operator^{L}	Simplify instantiation of generic class (since Java 1.7)	N^{T}	16/4.19	46/12.04
	(T7.2) Code style reformat	Use concise code style	N^{T}	9/2.36	
	(T7.3) Use constructor to initialize	Use class constructor to initialize the class properties	Y	7/1.83	
	(T7.4) Merge imports	Merge multiple imports from the same package	N^{T}	6/1.57	
	(T7.5) Replace with annotations^{L}	Replace code fragments with annotations which achieve the same functionality	Y	6/1.57	
	(T7.6) Try-with-resources^{L}	Use try-with-resources statement instead of finally block to close resources (since Java 1.7)	N^{T}	2/0.52	

L The superscript 'L' indicates the transformation uses language-specific feature (e.g., foreach, lambda, and diamond operator).
T The superscript 'T' indicates that although it is not a refactoring type, there is IDE plugin that supports the transformation.

As most refactoring engines only support making rule-based program transformations, we define refactoring in Def. 1 to be rule-based transformations, which correspond to the refactoring types. To obtain the predefined set of rule-based equivalent-preserving transformations, we refer to the refactoring types listed in RefactoringMiner [66] and Martin Fowler's refactoring catalog [19]. As our goal is to obtain a broader and diverse set of program transformations commonly used by open-source developers when simplifying code, our study in Section 3 investigated the developers' perspective of program simplification.

Definition 2 (Test equivalent). *Given a pair of programs (P, P') and a test suite T executable in P and P', P and P' are test-equivalent if $\forall t_k \in T$ where each test t_k takes as input inp_k and produces output out_k, $P(inp_k)=P'(inp_k)=out_k$ (i.e., for all tests in T, P and P' produce the same outputs).*

Although there are various granularity levels (e.g., statement, line, or character) used in prior techniques, we choose SLOC to represent simplified programs because it has been widely used in deletion-based [26, 79] and rule-based approaches [14].

3 Understanding Developer-Induced Program Simplification

In our study, we define *developer-induced program simplifications* as program simplifications that are based on **developers' perception** of program simplifications (i.e., transformations that the developers considered as a way to "simplify code" which lead to semantically-equivalent transformed programs, and the transformed programs should have less Source Lines of Code (SLOC)). We study developer-induced program simplification by manually inspecting commits within pull requests (PRs) in Java projects. We choose Java because it is one of the most popular programming languages. We focus on PRs as they contain detailed discussions to help us classify the simplification types and understand the motivations behind them.

3.1 Research Methodology

3.1.1 Mine Developer-Induced Program Simplification in PRs. We developed a crawler using the GitHub API [12] to search for PRs related to simplification. In our study, we use the keywords

"simplify" or its derived words "simplification", "simplified", together with "code" or "program" because we want to understand the developer's perspective of "code simplification" (according to our definition of developer-induced program simplification). We did not include other keywords related to existing approaches, such as "refactor", "clean up", "remove", "delete" as including these keywords may favor certain approaches (refactoring versus deletion-based), and may include irrelevant PRs. We also tried other keywords like "shorten", "reduce", and "shrink" but eventually excluded them as we observe that the results contained too many non-simplified PRs. After retrieving the top 1000 relevant PRs, two authors separately manually reviewed each one to exclude those changes that do not contain any Java file, do not focus on simplification, are irrelevant to code simplification, or do not reduce lines of code. Then, a meeting was held to resolve the conflict. This resulted in 382 PRs from 296 repositories.

3.1.2 Derive Taxonomy of Transformation Types. After reviewing simplification commits in the PRs, we developed a taxonomy through manual analysis of code changes in each PR using thematic analysis [10] – an approach that identifies patterns (or "themes") within data. To this end, we recruited two human raters to develop the taxonomy. Both raters were graduate students with over five years of Java programming experience. They followed the following steps independently, with conflicts resolved in meetings. First, we carefully reviewed PR titles, descriptions, and discussions to understand developers' motivations and the simplification process. We identified simplification commits by examining commit messages. If no explicit simplification-related keywords were found, we checked all PR commits. Next, we coded key "diff hunks" (i.e., code changes performing simplification) in each commit by describing their transformation types. We iteratively refined codes by reviewing related diff hunks and their context. We grouped key diff hunks based on codes, providing an overview of main edit actions and recurring patterns. Codes with similar meanings were aggregated into groups to derive broader themes. Finally, we reviewed and finalized the themes by providing clear definitions, then similar themes were merged to define the final set. After independent analysis, two authors meet to resolve the conflicts and finalize the themes. Specifically, we have two iterations: we achieved an initial Cohen's Kappa of 0.81, and after carefully discussion, disagreements were resolved, resulting in a final Cohen's Kappa of 1.0.

3.2 RQ1: Set of Transformations

Before following the steps of thematic analysis, we reviewed syntactic and semantic program simplification principles together with program refactoring from prior studies [47]. Table 2 lists the identified types of transformations across 382 PRs across 296 projects used in developer-induced program simplification. Note that a PR may contain multiple diff hunks where each diff hunk group may correspond to a particular transformation type, leading to multiple transformation types. The "Category" column in Table 2 describes the high-level types of program simplification, while the "Sub-category" column gives the specific categories. The last column (Category Total (#/%)) presents the total number and percentage of PRs that fit into a certain category.

In total, we identified seven main categories with 26 sub-categories. Among the seven categories, we observed that mere deletion only exists in 16.49% of the investigated PRs. This indicates that there *exists a missing gap in prior deletion-based approaches* (marked as "D" in Table 1), i.e., they can only fulfill the needs of up to 16.49% simplification. "Extract method" is the most commonly used transformation. This result is in line with prior studies that revealed that "extract method" is the most popular and most well-motivated refactoring among developers [59]. As our study shows that "extract method" is frequently used in developer-induced program simplification, it provides empirical evidence that developers still often perform "extract method" refactoring manually. However, prior study shows that manual refactoring often leads to mistakes [46].

Table 2 also shows that there are *simplification types that are Java language-specific* (try-with-resources, catching multiple exception types, and those marked as "L"). This indicates that developers prefer using language features to simplify programs, indicating that designers of future automated program simplification tools should incorporate these features.

Finding 1: Developer-induced program simplification includes 26 transformation types and deletion only exists in 16.49% of studied PRs. Two most widely used transformations are "Extract method" (19.63%) and "Replace with equivalent API" (16.23%). Some use language-specific features.

3.3 RQ2: Motivations Behind Simplifications

Instead of classifying the motivations based on code changes, we carefully read each PR's title, description, and discussion to analyze the motivations for simplification. In some PRs, developers only state "what" (changes that have been made) without "why" (the motivations), so we only keep PRs with clear descriptions of the motivations. We use the same procedures described in Section 3.1. This results in 79 out of 382 studied PRs.

Table 3. Motivations for simplifying programs.

Category	Description	GitHub PRs (#/%)
Cleanup code [79]	Remove unnecessary code	49/62.03
Readability [7, 45]	Improve readability	21/26.58
Complexity [8, 42]	Reduce complexity	7/8.86
Reusability	Reuse existing code	3/3.80

Table 3 shows the four main motivations, including "Readability" (26.58%), "Complexity" (8.86%), "Cleaning up code" (62.03%), and "Reusability" (3.80%). Note that one PR might contain multiple motivations. Compared to prior refactoring studies [34, 59], the main motivations of developer-induced program simplification are generally similar, with the focus being on removing duplication. We also observe from Table 3 that *most of the frequently mentioned motivations of developer-induced program simplification can be automatically measured using prior metrics*, indicating the promise of using these metrics to assess the quality of simplified programs.

Finding 2: The motivations that drive developer-induced program simplification are generally similar to those of refactoring, which include: (1) cleaning up code, (2) improving readability, (3) reducing complexity, and (4) enhancing reusability. Most motivations can be automatically measured using prior code metrics.

3.4 Supported Refactoring Types

Although there are several open-source refactoring detection tools (e.g., RefactoringMiner [66], RefDiff 2.0 [58]), we choose RefactoringMiner because it supports the greatest number of refactoring types (99 versus 15 types by RefDiff 2.0) considering the fact that RQ3 and RQ4 mainly focus on checking the supported refactoring types. We conducted two studies to analyze the overlap in program transformations between developer-induced program simplification and those supported by refactoring tools. First, we used RefactoringMiner [66] to identify if our transformations correspond to known refactorings, supplemented by Martin Fowler's refactoring catalog [19] for identifying refactoring-based program simplifications. Second, to understand the detection challenges, we ran RefactoringMiner on simplification commits and contacted its developers to confirm if each

transformation fits into currently supported refactoring types, understand detection challenges, and report detection failures.

3.4.1 Covered by Refactoring. Table 2 shows the overlapping ("Y") and non-overlapping transformations ("N") with refactoring types and those supported as IDE plugins ("T"). Notably, control-logic modifications (39.53%) are frequently used simplification types, but refactoring tools only support a subset. Additionally, 31.41% of the developer-induced program simplification involves extraction, a prevalent refactoring technique to deduplicate code and improve reusability.

3.4.2 Beyond Refactoring. Some simplifications do not align with well-established refactoring types. "Replace with equivalent API" is the second most frequently used transformation (16.23%) which replaces code snippets with semantically equivalent APIs. For example, instead of initiating an `Array` by traversing and setting up value for each element, developers simplify this procedure using `Arrays.fill()` method in `java.util.Arrays`. Except for the built-in methods in JDK, some popular Java libraries can be used to simplify programs (e.g., `StringUtils.isEmpty()` from the Apache Commons Lang library which includes helper utilities for the java.lang API). While prior studies on API recommendation [29] focus on suggesting APIs based on natural language descriptions, we found no technique that can automatically substitute code blocks with equivalent API calls. To replace code blocks with equivalent APIs from JDK or commonly used libraries, we could mine the usage patterns of the APIs, recommend potential usage by analyzing the context, and automate this process in a data-driven way. Based on the discussion with RefactoringMiner's developers, it can also be theoretically supported via combined operations that perform multi-project code clone detection, aggregate similar code snippets, extract methods, and then invoke these methods. When selecting appropriate APIs to replace, automated tools need to consider (1) the security of the used APIs [41] and their dependencies, and (2) the API misuse problem [83].

"Simplify boolean and algebraic expression" (8.42%) is another common yet unsupported simplification. For example, `false == co.isExpired()` can be simplified via the semantically-equivalent expression `!co.isExpired()`. In principle, we can simplify algebraic expressions by using rules like cancellation, commutativity, associativity, and design rules that enumerate all such semantically-equivalent expressions but it is time-consuming and impractical to craft these rules. The diamond operator (4.19%) has been introduced since Java 1.7. It allows omitting the repeated generic type arguments in the instantiation. For example, `Set<String> conditionKeys = new HashSet<String>();` can be simplified to `Set<String> conditionKeys = new HashSet<>();`. The try-with-resources (0.52%) statement is a Java feature with a try statement that declares one or more resources (e.g., file) that need to be released. For example, we can declare a `FileReader` without explicitly closing it inside a try statement like `try (FileReader fr = new FileReader(path))`.

Finding 3: While most simplifications (69%) are covered by prior refactoring types, uncovered ones include: "Replace with equivalent API" (16.2%), "Remove unnecessary code " (12.3%), "Simplify boolean and algebraic expression" (8.4%), "Use diamond operator" (4.2%), "Remove unused imports" (2.9%), "Code style reformat" (2.4%), and "Try-with-resources" (0.5%). Five are supported as IDE plugins.

Most newly discovered developer-induced program simplification cannot be easily supported by existing refactoring engines due to (1) missing API information ("Replace with equivalent API"), (2) a lack of exhaustive semantic-equivalent rules ('Simplify boolean and algebraic expression" and "Simplify non-control statement"), and (3) ambiguity regarding redundant code ("Remove unnecessary code"). This shows the need to design new tools to automate these transformations.

Table 4. Issues submitted to RefactoringMiner.

ID	Issue NO.	Transformation	Status
I-1	678	Inline Variable	Fixed
I-2	679	Invert Conditional	Fixed
I-3	680	Merge Conditional	Fixed
I-4	682	Replace Generic with Diamond	Fixed
I-5	683	Replace with Foreach Loop	Fixed
I-6	684	Replace Conditional with Ternary	Fixed
I-7	685	Wrap with Try-With-Resources	Fixed
I-8	681	Simplify Boolean Expression	Confirmed

For each commit, we also run RefactoringMiner to check if it can detect the transformations. Notably, we found several scenarios where RefactoringMiner fails to detect the transformations despite being in the list of supported refactoring types. For example, there exist cases where the "Inline Variable" refactoring interleaved with method return operation, which we classified as "Simplify method return" (13.87%). Although RefactoringMiner supports this transformation, its newest version (V3.0.4) and its Chrome extension (V2.0.4) both failed in certain cases. After analyzing the implementation of RefactoringMiner, we notice that refactoring that involves *non-pure refactorings* (interleaved with other transformations) causes the handcrafted rules of RefactoringMiner to fail. We also found cases where RefactoringMiner cannot detect the complete mapping for multi-line transformations. Although RefactoringMiner can detect the modification of the annotations, it fails to provide the mapping to the users if they want to know which part of the code is replaced by the added annotations. For example, Getter and Setter methods can be replaced with the annotations `@Getter` and `@Setter`. RefactoringMiner can only detect that the two annotations are added but cannot associate them together if there exist multiple code changes. RefactoringMiner also offers limited support for transformations with language-specific features (foreach and try-with-resources). We submitted issues to RefactoringMiner based on our study [73]. As shown in Table 4, among eight of our submitted issues, seven issues have been fixed (Finding 4), the other one issue (I-8) has been confirmed (RefactoringMiner's developers have labeled this issue as enhancement). Among our submitted issues, three issues (I-4, I-6, and I-7 in Table 4) related with Java language features are acknowledged by the RefactoringMiner developers that these are beyond typical refactoring types which aligns with Finding 3 and Finding 4. The developers have formally added supports for our proposed operations (No. 100-102 in [64]).

Finding 4: Among the supported types, RefactoringMiner is still limited in (1) handling non-pure refactorings, (2) providing a complete mapping for a multi-line transformation, and (3) handling language-specific features. This calls for redesigning the detection rules in RefactoringMiner.

3.4.3 Automation. We analyze the possibility of automating program simplification by checking the supported simplifications in two auto-refactoring tools [48, 65]. JDeodorant [65] is a code smell detection tool that automatically identifies and applies refactoring. It supports five code smells (Feature Envy, Type/State Checking, Long Method, God Class, and Duplicated Code) where "Long Method" can be resolved by "Extract Method" refactoring, and "Duplicated Code" problems can be resolved by "Extract Clone" refactoring. As our study shows that only 31.41% simplifications are related to extraction, only up to 31.41% simplifications can be recommended by JDeodorant in the best case. Meanwhile, the prior multi-objective approach supports 11 refactoring types [48] where three are simplification types.

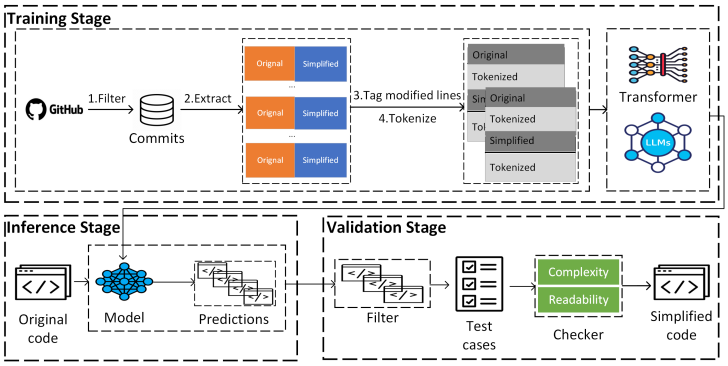


Fig. 1. Overall workflow of SimpT5.

After our analysis, we identify the following features: (1) the program token length before and after simplification are relatively short. For example, the program before simplification using "Replace with equivalent API" is usually a contiguous code snippet within a method scope, whereas the simplified program only contains a few lines of API invocations; (2) the simplification types have some recurring patterns; (3) massive simplification data can be mined from GitHub. Based on these features, we believe that *learning-based techniques can be used to automate program simplification.*

Finding 5: Auto-refactoring tools like JDeodorant and prior approach [48] can automate developer-induced program simplification but only support up to seven out of 26 types.

4 Automating Developer-Induced Program Simplification

In our study in Section 3, we manually analyzed if the transformed programs are semantically equivalent with the original program but manual analysis is impractical when designing an automated tool. Hence, we define *tool-supported program simplification*: given a program P, and a transformed program P', P' is a program produced via tool-supported program simplification of P if (1) P' is obtained via developer-induced program simplification, (2) P' contains less Source Lines of Code (SLOC) than P, and (3) P and P' are test-equivalent (defined in Def 2). The transformed program P' is obtained through a set of transformations, including (1) syntactic program transformations, and (2) semantic program simplifications. Based on our study, we design SimpT5, a program simplification framework that performs tool-supported program simplification. Figure 1 depicts the three-stage workflow of SimpT5: (1) training, (2) inference, and (3) validation.

Training and Test Data Collection. Although there are several refactoring datasets available [23, 38], Finding 3 in our study shows that some simplification types are not covered by existing refactoring. Hence, we propose SimpliBench, a new program simplification dataset that contains pairs of (P, P') with the original method P and its simplified version P'.

Select Commits. We use GH Archive [61] to collect the related commits between January 2012 and December 2021. We select 10 full years of data, which is the latest at the year of data collection (our project involves time-consuming manual efforts: literature reviews, analysis of transformation, and back-and-forth discussions with RefactoringMiner's developers, so the data collection started in 2022, and we excluded partial-year data from 2022). To select developer-induced program simplification commits based on developers' perception, we only keep commits with the keywords "simplify" or its derived words "simplification", "simplified", together with "code" or "program" in their commit messages. To remove commits irrelevant to program simplification, we randomly

Table 5. Statistics of SimpliBench.

Statistic	Whole	Valid
# of Projects	25,022	307
# of Method-pairs	92,485	404
Mean statement coverage of original method	-	81.2%
Mean SLOCs # of original method	17	16
Mean SLOCs # of simplified method	14	12

sample and check 200 commits, and conclude ten commit message anti-patterns: "config," "fix," "bug," "patch," "merge," "misspelling," "typo," "warning," "comment," and "doc". As we only focus on simplification for Java programs, we filter commits where the committed code changes contain no Java files. This results in 37846 commits from 26110 projects. To understand the noise this filtering mechanism can potentially introduce, we manually examined a randomly selected sample of 100 commits filtered using this approach and found 89 simplification-related commits. The manual analysis shows that the amount of noise (11%) is reasonable for ML training on large training data [35, 77]. Next, we split a commit based on *diff hunks*. As stated before, program simplification aims to reduce SLOCs while preserving the test-equivalent behavior of the generated program. Hence, our crawler automatically selects diff hunks that contain more deleted lines than added lines (code comments and blank lines in the diff hunk are excluded while counting SLOCs). We use JavaParser [70] to extract the method surrounding the diff hunk because (1) it contains a logical set of relevant variables and could give semantic information about the functionality; (2) compared to class, the method is relatively short with fewer tokens, which is more friendly for model training or tuning as maximum input and output sequence length is usually limited.

Data Preprocessing. After collecting the code samples, we then pre-process the raw source into pairs at method-level, where each training sample contains the original code and the simplified version. As shown in Figure 1, our code representation is based on the token sequence as prior large language models trained on code (LLMC). Like prior approaches [33, 40, 69], SimpT5 takes the tokenized original method as input and produces the simplified method as output. During pre-processing, we discard methods greater than 512 tokens which are beyond the maximum input sequence length of CodeT5 [74]. LLMC usually suffer from long sequence problems [28]: as the length of the input grows, the accuracy of LLMCs decreases, so we set max length of the input and output sequence to 512 following prior studies [28]. To remove duplication, we checked each method pair against other method pairs in the dataset, if two pairs are the same after tokenization, we remove the duplicates. Finally, our collected dataset contains 92,485 (original method, simplified method) pairs from 25022 projects. Based on a prior study [57], data-splitting strategies can significantly affect the model's performance. Hence, we use the most rigorous splitting strategy which randomly divides the projects into three partitions such that code from the same project exists only in one partition. This ensures that the trained model is tested on samples from *new, unseen* projects, i.e., we split the data into training, validation, and testing sets with a ratio of 8:1:1 at the project-level (80% of projects for training, 10% for validation, and 10% for testing).

Table 5 shows the statistics of SimpliBench. As some unmodified versions of projects in our dataset may have compilation errors or test failures, we divide the dataset into (1) *whole* (including all collected simplified method pairs) and (2) *valid* (a subset of the whole dataset where the unmodified projects can be compiled successfully and all tests within the test suite of the project pass). To prepare the *valid* dataset, we filter commits that (1) fail to compile using `mvn build` command, and (2) do not have any test. The key difference between the two is that in *whole*, some projects are

uncompilable or test fails so we can only evaluate whether patches are the same as ground-truths (PP), whereas in *valid* dataset, we can check for test-equivalence using the test suite of the project. As the statement coverage for the method under test (i.e., the original un-simplified method) is relatively high (mean 81.2%), we can rely on existing tests to check for test-equivalent relations.

4.1 Model Training and Tuning

Modified Line Localization. Inspired by APR approaches [18, 40] where fault localization is performed to identify the "buggy" lines in which the transformations are applied, we propose *modified line localization* to pinpoint the lines in the original program P to apply the program transformations to produce the simplified program P'. Our intuition is that *some locations (e.g., containing code smells) may provide hints for the needs of simplifications* so we proposed "modified line localization" to encode the location information in our model (our evaluation in Section 5.3 shows that encoding this information is more effective than without the line information). To encode simplified lines into our code representation, we marked the original and the simplified code lines with special tokens (<original>, </original>, <simplified>, </simplified>). If multiple *diff* hunks exist, we will add these tokens to each changed line at the beginning and the end. Specifically, we evaluate the effect of *perfect modified line localization* (i.e., the lines modified by the correct developer-induced program simplification) for program simplification, following prior APR evaluation [40]. To evaluate the impact of localization information on program simplification, we also design another code representation without modified line localization (i.e., without inserting special tokens) as a baseline comparison. In Section 5.3, our experiment shows that adding modified line localization helps improve the model's effectiveness.

Model Design. Given the nature of our task, we use an Encoder-Decoder-based LLMC [74] as it has been shown to be effective on code transformation tasks. We choose CodeT5 [74] as the LLMC because prior studies [28] showed that CodeT5 generally performed better than other LLMCs (e.g., CodeBERT) while having a reasonable model size. Our LLMC is trained through training samples and hard prompts [71]. By adding fixed natural language instructions to the model input, hard prompt uses task-specific knowledge learned during pre-training for the subsequent tuning stage. Specifically, we use the prompt "*Simplify the following java method: [X], the simplified version is: [Y]*" where [X] represents the original method and [Y] represents the simplified method. It is worthwhile to mention that we include the prompt for verifiability (i.e., SimpT5 uses all prompts fully automatically).

Filtering Unaltered Programs. Among the generated simplified programs, we notice that about 30% of them are the same as the original method (we call them *unaltered programs*). As unaltered programs are naturally semantically equivalent to the original programs and may mislead the validation stage, we filter out all unaltered programs before validation. Note that this step will not cause unfair comparison with other baselines because (1) we perform the same filtering for all deep-learning baselines that may generate unaltered programs, and (2) not filtering will be unfair because we will mistakenly overcount these as being equivalent.

4.2 Validation Stage

Validation is the key step in ensuring the quality of the generated simplified programs. Given an original program P, we consider a program as a simplified program P' if (1) P' contains less SLOCs than P, and (2) P' is test-equivalent with P. To ensure consistent programming style, we perform post-processing by reformatting original and simplified programs using IntelliJ IDEA. To check for the (1) condition, we compare the SLOCs of P and P' to ensure that the code size of P' is reduced. To check for the (2) condition, we first filter out simplified programs that do not compile and then run each simplified program against the entire test suite to check for test-equivalent. Based on

Finding 2, we designed a checker to measure the differences between the original and simplified programs in complexity and readability.

Cyclomatic Complexity. One of the main motivations for developer-induced program simplification is to reduce the complexity of a program. We use cyclomatic complexity [17, 42] to measure the complexity of a program. Cyclomatic complexity is a commonly used metric for measuring the number of linearly independent paths through a program's source code. Programs with lower cyclomatic complexity are generally easier to maintain.

Readability. We use cognitive complexity [45] to measure the readability of a program. Cognitive complexity measures how difficult it is for humans to read and understand a program. Programs with lower cognitive complexity are generally easier to understand.

5 Evaluation

We evaluate the effectiveness of SimpT5 by answering the questions below:

RQ5 How effective is SimpT5 compared to other tools?
RQ6 How effective is modified line localization in SimpT5?
RQ7 Among the correctly generated programs, what are the transformation types used to generate the simplified programs?
RQ8 What is the compilation success ratio, test-equivalent ratio, SLOC reduction, cyclomatic complexity reduction, and cognitive complexity reduction of the generated programs?

5.1 Experimental Setup

5.1.1 Implementation. We use the pre-trained CodeT5-base model and the corresponding tokenizer from Huggingface [13]. Our implementation of prompt-tuning is based on OpenPrompt [15]. We use the generic training strategy and parameter settings following the official implementation of CodeT5 [74]. Specifically, we set the learning rate to be $5e^{-5}$ and use the AdamW optimizer with a linear warmup to optimize the model. The training and validation batch size is 8. We tune the CodeT5-base model for 50 epochs. The maximum length of input and output text is set to 512. We set the beam size to 10. All experiments are conducted on Ubuntu 20.04 with 2x 24GB GeForce RTX 3090 GPUs. For the generated simplified programs, we first replace the original method with the simplified one, then compile the entire project under JDK1.8.0 and JDK11.0.15 during the validation step. When validating candidate programs, SimpT5 validates programs automatically where only the first one (if any) is found to be test-equivalent will be provided to users.

5.1.2 Evaluation Metrics. We use two metrics commonly used by learning-based tools to assess the quality of the generated programs: (1) *Perfect Prediction (PP)* [63, 68], and (2) CodeBLEU [55].

Perfect Prediction (PP): Perfect Prediction (PP) measures the ratio of generated programs that match the test samples' ground truth method A generated method is a PP if it is exactly the same as the developer-written after-simplification method.

CodeBLEU: CodeBLEU measures the similarity of the generated code to the ground truth. It combines the strength of BLEU [51] in n-gram matching and further checks syntax via Abstract Syntax Trees (AST) and code semantics via data flow analysis. Prior study [55] showed that CodeBLEU better correlates with developers' perception of code similarity than the BLEU metric.

5.1.3 Baselines. We compare SimpT5 with the baselines below:

Ideal Delta Debugging (IDD). As there are many variants of delta debugging [44, 72, 80], we adopt the ideal version of delta debugging IDD as a strong representative of these variants. Specifically, IDD (1) will only use deletions for transformations, and (2) can select the *correct* statements for

Table 6. Experimental results of deep-learning-based approaches.

Dataset	Code Rep.	TufanoNMT				AutoTransform				Vanilla Transformer				SimpT5			
		PP	CodeBLEU			PP	CodeBLEU			PP	CodeBLEU			PP	CodeBLEU		
		#/%	mean	median	st. dev.	#/%	mean	median	st. dev.	#/%	mean	median	st. dev.	#/%	mean	median	st. dev.
Whole	Raw	10/0.11	0.450	0.423	0.205	0/0.00	0.094	0.060	0.093	335/3.52	0.801	0.828	0.148	**1842/19.37**	**0.856**	**0.882**	0.136
(#9508)	Localized	14/0.15	0.411	0.385	0.199	0/0.00	0.084	0.053	0.088	2738/28.80	0.867	**0.911**	0.149	**2827/29.73**	**0.878**	**0.911**	0.128
Valid	Raw	0/0.00	0.460	0.449	0.195	0/0.00	0.086	0.053	0.089	2/0.50	0.317	0.302	0.172	**64/15.84**	**0.849**	**0.873**	0.132
(#404)	Localized	0/0.00	0.425	0.397	0.196	0/0.00	0.086	0.052	0.092	2/0.50	0.386	0.396	0.199	**123/30.45**	**0.880**	**0.910**	0.124

deletions (whenever simplifying the ground truth G requires only pure deletions, IDD can produce exactly the same program as G). IDD represents deletion-based approaches.

JDeodorant. This is an auto-refactoring tool [65] introduced in Section 3.4.3 that represents rule-based approaches.

TufanoNMT. This is an NMT-based sequence-to-sequence transformation model [68]. It tokenizes the input program into a sequence and translates it into a fixed code sequence. We choose TufanoNMT as it has shown promising results in performing refactoring-related transformations.

AutoTransform. This tool leverages a BPE scheme to handle new tokens and a Transformer-based NMT architecture to handle longer sequences [63].

Vanilla Transformer. As other baselines are not designed for this task, we trained a vanilla transformer from scratch on our whole dataset by reusing prior parameters [63].

5.2 [RQ5] Comparison with Prior Techniques

Table 6 shows the results for all the learning-based approaches. Compared to TufanoNMT and AutoTransform, SimpT5 produces the greatest number of perfect predictions (PPs) (around 30%) and code that are more similar to PPs (higher values of CodeBLEU). Compared to the prior evaluation of TufanoNMT [68] that perfectly predicts code transformations up to 36% for beam=10 without splitting the dataset by projects (may suffer from data leakage problem as similar tokens and code patterns may be shared within methods from the same project [57]), our code model gives comparable results while splitting dataset by projects, indicating the applicability of our results across different projects. Among all evaluated baselines, we also observe that the Vanilla Transformer with localization is the most effective: it produces more PPs, and programs that are similar to PPs (its median CodeBLEU is 0.911, which is the same as SimpT5 with localization). This indicate that training a model from scratch helps in producing more correctly simplified programs.

Table 7. Comparison results with IDD and JDeodorant.

Dataset	Perfect Prediction **PP** (#/%)		
	IDD	JDeodorant	SimpT5
Whole(#9508)	1913/20.12	-	2827/29.73
Valid(#404)	82/20.30	0/0.00	123/30.45

For the other baselines that are not based on deep learning (i.e., IDD and JDeodorant), we calculate the total number of PPs generated in the *valid* dataset. We did not run JDeodorant in the *whole* dataset as it requires compilation. Table 7 shows the results for the two approaches. Overall, SimpT5 outperforms IDD and JDeodorant by generating the greatest number of PPs. As JDeodorant only supported limited transformation types, it did not generate any PP for the *valid* dataset. Listing 1 [11] shows a perfect prediction (PP) generated by SimpT5 where other baselines fail to produce the PP. SimpT5 applies "Simplify method return" transformation, which inlines the declaration into a return statement, resulting in reduced lines of code.

```
public Collection<AuditRequestLog> getAuditRequestLogs(){
- Collection<AuditRequestLog> newList=repository.findAll();
- return newList;
+ return repository.findAll();}
```

Listing 1. A SimpT5 generated perfect prediction example that failed to be generated by other baselines.

5.3 [RQ6] Ablation Study for Effectiveness of Modified Line Localization

To investigate the effectiveness of the modified line localization in SimpT5, we conduct an ablation study to evaluate the number of PPs produced by SimpT5 with (*Localized*) and without (*Raw*) modified line localization. Overall, we can observe from *Localized* and *Raw* columns in Table 6 and Table 8 that line localization helps to guide SimpT5 in generating more PPs and increasing the diversity of the transformation types used. Notably, without localization, SimpT5 can only generate 64 PPs but can generate 59 (92%) more PPs after adding localization information. Listing 2 [4] gives an example showing the differences between program generated with and without line localization. Without modified line localization, SimpT5 would remove all the `System.out.println()` in the original method. With line localization in SimpT5 that marks the first two lines, SimpT5 can focus on simplifying these lines via "Use foreach in loop" transformation and generate PP. This example shows that *modified line localization guides SimpT5 into generating more diverse transformations.*

```
-<original>for(int i=0; i<numbers.size(); i++){</original>
-<original>Integer currentNum = numbers.get(i);</original>
+<simplified>for(Integer currentNum : numbers){</simplified>
 Integer otherNum = map.get(2020 - currentNum);
 if (otherNum != null) {
    System.out.println("this num = " + currentNum);
    System.out.println("other num = " + otherNum);
    int result = otherNum * currentNum;
    System.out.println("result = " + result);
    return result;    }}
```

Listing 2. Effectiveness of modified line localization.

5.4 [RQ7] Types of Transformations Used in Perfect Predictions

Table 8. Diversity of transformation types in perfect predictions (PPs) in *valid* dataset.

Tool	Transformation Type (type (#case with a specific type))		Total (#type; #case)	
	Raw	Localized	Raw	Localized
TufanoNMT	-	-	-	-
AutoTransform	-	-	-	-
Vanilla Transformer	T3 (1), T1.1 (1)	T3 (1), T1.1 (1)	2; 2	2; 2
SimpT5	T3 (27), T1.1 (14), T6.1 (5), T2.2 (4), T5.1 (3), T2.1 (2), T7.2 (2), T7.6 (2), T1.3 (2), T1.9 (1), T1.6 (1), T7.5 (1)	T3 (72), T1.1 (17), T2.2 (7), T1.10 (6), T5.1 (6), T1.3 (4), T2.1 (2), T6.1 (2),T7.6 (2), T7.2 (1), T1.9 (1), T1.6 (1), T7.5 (1), T4.1 (1)	12; 64	14; 123

In "Tx (y)", Tx denotes the transformation types in Table 2, and y denotes the number of PPs with type Tx.

To investigate the diversity of the transformations used in the correctly generated simplifications, we manually analyzed the types of transformations in the generated perfect predictions (PPs) by the deep learning baselines (we did not evaluate the diversity of DD^I and JDeodorant as these tools can only support limited types of transformations). The first two authors of this paper independently checked the transformations, and then they held a meeting to resolve any

disagreement. Table 8 shows the distribution of the transformation types in PPs generated by the deep learning baselines. In the "Transformation Type" columns, we use the labels in Table 2 to refer to the transformation type discovered in our study, whereas the last two columns represent the total number of transformation types (#type) used and the total number of perfect predictions generated by each approach for the *valid* dataset. As we observe that all automatically generated programs can be categorized using our taxonomy in Table 2, this shows that *our taxonomy of simplification is general.* Overall, SimpT5 generates the highest number of PPs using the most diverse set of transformations compared to other baselines. Specifically, it uses 12 (without localization) and 14 (with localization) different types of transformations to generate simplified programs. We also observe that Vanilla Transformer uses only deletion in both of the correctly generated simplified programs, making this baseline similar to DD^I, which performs only deletion correctly.

5.5 [RQ8] Compilation Success Ratio, Test-Equivalent Ratio, SLOC Reduction, Cyclomatic Complexity Reduction, and Cognitive Complexity Reduction of Generated Programs

Table 9. Quality of generated programs in *valid* dataset.

Tool	Compilation Success (#/%)		Test-equivalent (#/%)	
	Raw	Localized	Raw	Localized
TufanoNMT	22/5.45	10/6.19	16/3.96	7/1.73
AutoTransform	2/0.50	2/0.50	2/0.50	2/0.50
Vanilla Transformer	4/0.99	4/0.99	4/0.99	4/0.99
SimpT5	**70/17.33**	**143/35.40**	**63/15.59**	**126/31.19**

To evaluate the quality of the generated programs in the *valid* dataset, we calculate *compilation success rate* (i.e., the ratio of the generated simplified programs that are not unaltered and compilable among all programs in the valid dataset) and *test-equivalent rate* (i.e., the ratio of the generated altered simplified programs where the generated program and the corresponding original program are test-equivalent according to Definition 2 in the valid dataset). To check for test-equivalence, we replace the original method with the simplified method, compile the whole project, and then run the test suites. Table 9 shows the compilation success rate and test-equivalent rate on our valid dataset for all the deep-learning-based baselines. Overall, the programs generated by SimpT5 are of higher quality in terms of compilation success rate and test-equivalent rate. Specifically, SimpT5 performs better with line localization, which is almost double the ratio for compilation success rate (35.40%) and test-equivalent rate (31.19%). Compared to other baselines, SimpT5 could generate simplified programs that are free of syntax errors and behavior-preserving (with respect to tests).

We also measure the quality of the simplified programs generated by SimpT5 using three metrics described in Section 4.2: SLOC (Source Lines of Code), Cyclomatic Complexity, and Cognitive Complexity. We calculate the number of PPs and test-equivalent programs. As other baselines only generate very few PPs and test-equivalent programs (e.g., Vanilla Transformer only generates 4 test-equivalent programs and 2 PPs), we only compare with ideal delta debugging (IDD) which generates 82 (20.30%) PPs. Table 10 provides the results for these metrics. Specifically, for the mean and median reduction ratio (e.g., reduced SLOC divided by original method SLOC), SimpT5 performs better on the raw data (18.13%) and localized data (27.27%), respectively. Compared to IDD, our results show that using *a diverse set of program transformations can achieve even more reductions* because SimpT5 often correctly selects deletion-based transformations (as shown in Table 8) to perform the reduction, and some transformations (e.g., replace anonymous with lambda) could

Table 10. Simplification results with respect to SLOC, Complexity, and Cognitive Complexity in *valid* dataset.

Metric	Type	IDD	SimpT5			
			Raw		Localized	
		PP	PP	TE	PP	TE
SLOC	mean	-13.64%	-18.13%	-14.27%	-15.92%	-17.43%
	median	-23.08%	-18.75%	-20.00%	-27.27%	-18.18%
Complexity	mean	-7.52%	-2.05%	-8.20%	-10.65%	-15.97%
	median	-0.00%	-0.00%	-0.00%	-0.00%	-20.00%
Cognitive Complexity (Readability)	mean	-8.87%	-3.67%	-11.37%	-13.73%	-18.28%
	median	-0.00%	-100%	-0.00%	-0.00%	-50.00%

SLOC: source lines of code; PP: perfect prediction; TE: test-equivalent. We only compare with ideal delta debugging (IDD) as other baselines produce too few (<50) TE/PPs.

reduce more SLOCs than just removing a few lines of unused statements. For the complexity and readability, SimpT5 achieves better results for the test-equivalent programs on the localized data because SimpT5 not only applies deletion for simplification but also uses other transformations to reconstruct code, making it more readable (↓ cognitive complexity) and less complex (↓ complexity).

Are SimpT5's Generated Test-Equivalent Programs Useful? Table 9 shows that SimpT5 produces 143 out of 404 programs that compile successfully for localized version (SimpT5 can filter the remaining 261 programs that lead to compilation errors). For those 126 test-equivalent programs, the complexity metric can filter 1 (i.e., simplified program leading to a higher cyclomatic complexity), readability metric can filter 8 simplifications which increase the cognitive complexity, resulting in 117 final results. Among these 117 test-equivalent programs, our manual analysis shows that 105 (90%) are semantically equivalent to the PPs (i.e., they use different transformations). There are only 12 out of 117 programs that are not semantically equivalent to PPs due to insufficient test cases (12/117≈10% false positives). In future, directed test generation techniques can be used to eliminate these false positives. We also observe 58 cases where SimpT5 produces PPs but fails to compile as it can only make method-level changes. For example, Listing 3 shows that SimpT5 correctly predicts the "Extract method" by replacing relevant code with the `assertHasAndNotNull(playlist_id)` method call but does not currently support the new method creation at lines 1–3. In future, we plan to integrate SimpT5 with a refactoring engine (e.g., Eclipse LTK [20]) to produce the complete "Extract method" refactoring. Theoretically, a rule-based approach like JDeodorant can produce the perfect prediction but our experiments show that JDeodorant with Deckard [32] as the clone detection tool with default configuration (minT is set to 30 or 50, stride ranges from 2 to inf, and similarity ranged between 0.9 and 1.0) for performing "Extract Clone Refactoring" cannot generate the PP in Listing 3. This is because the code to be extracted in Listing 3 has only relatively small number of tokens. Tuning the configurations of clone detection tools may improve the results but a lot of false positives may be generated and it is challenging to filter all these false positives.

```
1  + public void assertHasAndNotNull(String str) {
2  +  assert(str != null);
3  +  assert(!str.equals("")); }
4  public Builder playlist_id(final String playlist_id) {
5  -  assert (playlist_id != null);
6  -  assert (!playlist_id.equals(""));
7  +  assertHasAndNotNull(playlist_id);
8     return setPathParameter("playlist_id", playlist_id);}
```

Listing 3. Example where SimpT5 produces PP but fails to compile as it does not support method creation.

6 Threats to Validity

We identify the following threats to validity of this paper:

External. The set of transformations used by developers found in our study merely represents transformations that frequently occurred in commits (i.e., we do not claim that our identified set is "complete"). We mitigate this by studying developer-induced program simplification from two aspects: (1) PRs in the study, and (2) commits in SimpliBench. Our study mainly focuses on method-level transformations. Although we observe class-level and component-level transformations, we did not classify them as they are usually interleaved with many other classes, making it difficult to isolate and map into well-known types. As we only focus on Java open-source projects, the findings may not generalize beyond Java and other closed-source projects.

Internal. Our scripts and tool may have bugs that can affect our results. To mitigate this, we open-source all data, source code and scripts.

Conclusion. Conclusion threats include (1) overfitting of our dataset and (2) subjectivity of manual analysis. We mitigate (1) by (a) de-duplicating to remove identical methods, and (b) ensuring that the training and testing datasets use data from different projects. We reduce (2) by cross-validating between two annotators.

7 Implications

Based on our study and evaluation, we discuss the implications for developers and researchers.

Implication for Developers. Our study identifies a set of commonly used transformations for developer-induced program simplification and developers' motivations. Based on our study's two most common transformations ("Extract method" and "Replace with equivalent API"), we observed that developers tend to simplify code by adding new methods or invoking existing API (Finding 1). Despite sharing the same goals of getting a reduced program, some IDE-supported simplifications need to be invoked under different menus (Finding 1). As IDE users find it difficult to find these simplifications [3, 37], IDE plugin developers should consider *enhancing usability by aggregating all transformations* sharing the same goal of simplification. For example, most IDEs support "Remove unused imports" as an "Organize imports", whereas "Use diamond operator" is available only under "Java inspection" in IntelliJ. Our study of the motivations behind program simplification shows that developers *use existing static metrics to evaluate the quality of simplified programs* (Finding 2). While many simplification types are covered by prior refactoring types (Finding 3), these tools only support limited types (Finding 5), which means that developers can only use prior tools (e.g., JDeodorant) for the few type. Hence, SimpT5 can help relieve the burden of manual simplification.

Implication for Researchers. Our study and program simplification framework lay the foundation for research in three promising directions. First, our study *provides the key criteria that drive the design of future automated tools for developer-induced program simplification.* Such tools should (1) have a richer set of program transformations (Finding 1), (2) incorporate language-specific features (Finding 1), (3) use prior static metrics (e.g., SLOCs, readability and cyclomatic complexity) to check for code quality (Finding 2). Our proposed framework and SimpliBench lay the first step in promoting future research in this direction. Second, *there exist limitations when adapting prior tools for detecting and automating developer-induced program simplification.* As developer-induced program simplification involves diverse types of transformations, our study reveals that several widely used transformations, e.g., replacing with equivalent API and simplifying expression, have not been automated (Finding 1). To enhance the usability of IDE-supported simplifications, it is worthwhile to enhance UI design to help users identify transformations that share the same goals of simplifications (Finding 3). For the supported refactoring types, prior tools fail to detect them due to the limitations in the hand-crafted rules used for detection (Finding 4) and limited supported types

by auto-refactoring tools (Finding 5). Third, our study serves as a preliminary study to *motivate future research on using a richer set of transformations for improving deletion-based approaches* (e.g., program reduction). As deletion-based approaches are used in many domains (Section 2), it is worthwhile to investigate applying more human-like transformations to improve these approaches. Our results show that SimpT5 can produce high-quality programs with diverse transformations.

8 Conclusion

We present the first study of developer-induced program simplification in OSS projects, focusing on the transformation types, developers' motivations, and the transformations covered by prior refactoring types. Our study reveals gaps in applying prior approaches for detecting and automating developer-induced program simplification and outlines the criteria for designing automatic simplification techniques. We also discovered bugs in RefactoringMiner in which we have reported eight bugs where seven are fixed. Based on our study, we propose SimpT5, an automated simplification framework that learns from the training part of our dataset SimpliBench using modified line localization and checkers to ensure the quality of the generated programs. Our results show that SimpT5 is more effective than prior approaches in producing more correctly simplified programs.

9 Data Availability

The data and code associated with this work are available at [73].

Acknowledgments

We sincerely thank Professor Nikolaos Tsantalis for his help in identifying the refactoring. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants RGPIN-2024-04301 and the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X018202/1 and EP/X037304/1.

References

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. *SIGPLAN Not.* 25, 6 (June 1990), 246–256. https://doi.org/10.1145/93548.93576

[2] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138. https://doi.org/10.1016/j.infsof.2018.12.009

[3] Michael Berry. 2011. *Convert existing generics to diamond syntax.* stackoverflow. https://stackoverflow.com/questions/6796545/convert-existing-generics-to-diamond-syntax

[4] Michael Berry. 2011. *Foreach example.* stackoverflow. https://github.com/ggavriilidis/adventOfCodeChallenge2020/commit/e56363b6d409e491506b084e7319c9bc82895cd6

[5] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. 2005. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM'05).* 27–36. https://doi.org/10.1109/ICSM.2005.27

[6] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2015. ORBS and the limits of static slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM).* 1–10. https://doi.org/10.1109/SCAM.2015.7335396

[7] Raymond P.L. Buse and Westley R. Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08).* Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/1390630.1390647

[8] G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18).* Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/3194164.3194186

[9] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022).* Association for Computing Machinery, New York, NY, USA, 18–30. https://doi.org/10.1145/3540250.3549162

[10] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284. https://doi.org/10.1109/ESEM.2011.36

[11] danilopez. 2019. *Perfect prediction example*. GitHub. https://github.com/Verdoso/VersioningDemo/commit/6afa0093e24fbb26f4597fdbbcef22714e3d4aa8

[12] GitHub Developers. 2022. *GitHub API*. GitHub. https://developer.github.com/v3/

[13] Hugging Face Devoloper. 2024. *Huggingface*. Hugging Face. https://huggingface.co/models

[14] Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring sequential Java code for concurrency via concurrent libraries. In *2009 IEEE 31st International Conference on Software Engineering*. publisher, address, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

[15] Ning Ding and et. al. 2021. *OpenPrompt*. GitHub. https://github.com/thunlp/OpenPrompt

[16] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[17] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic Complexity. *IEEE Software* 33, 6 (2016), 27–29. https://doi.org/10.1109/MS.2016.147

[18] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, address, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[19] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[20] Leif Frenzel. 2005. *Eclipse LTK*. Eclipse. https://www.eclipse.org/articles/Article-LTK/ltk.html

[21] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *2012 34th International Conference on Software Engineering (ICSE)*. publisher, address, 211–221. https://doi.org/10.1109/ICSE.2012.6227192

[22] Yossi Gil and Matteo Orrù. 2017. The Spartanizer: Massive automatic refactoring. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 477–481. https://doi.org/10.1109/SANER.2017.7884657

[23] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology* 95 (2018), 313–327. https://doi.org/10.1016/j.infsof.2017.11.012

[24] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[25] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. 2022. Constructing dataset of functionally equivalent Java methods using automated test generation techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 682–686. https://doi.org/10.1145/3524842.3528015

[26] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation* (Seattle, WA, USA) *(A-TEST 2016)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/2994291.2994296

[27] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. publisher, address, 194–203. https://doi.org/10.1109/ICSME.2017.26

[28] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181

[29] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/3238147.3238191

[30] Nicholas Jalbert and Koushik Sen. 2010. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 57–66. https://doi.org/10.1145/1882291.1882302

[31] Bo Jiang, Yuxuan Wu, Teng Li, and Wing Kwong Chan. 2017. Simplydroid: efficient event sequence simplification for Android application. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. publisher, address, 297–307. https://doi.org/10.1109/ASE.2017.8115643

[32] 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)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[33] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[34] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[35] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 481–490. https://doi.org/10.1145/1985793.1985859

[36] David Kinzett, Mengjie Zhang, and Mark Johnston. 2008. Using Numerical Simplification to Control Bloat in Genetic Programming. In *Simulated Evolution and Learning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 493–502. https://doi.org/10.1007/978-3-540-89694-4_50

[37] Steve Kuo. 2012. *IntelliJ Organize Imports*. stackoverflow. https://stackoverflow.com/questions/8608710/intellij-organize-imports

[38] István Kádár, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 599–603. https://doi.org/10.1109/SANER.2016.42

[39] Gen Lu and Saumya Debray. 2012. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*. 31–40. https://doi.org/10.1109/SERE.2012.13

[40] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[41] Neil Madden. 2020. *API security in action*. Simon and Schuster.

[42] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[43] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. 2015. Does automated refactoring obviate systematic editing?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 392–402. https://doi.org/10.1109/ICSE.2015.58

[44] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[45] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. 2020. An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3382494.3410636

[46] Emerson Murphy-Hill and Andrew P. Black. 2008. Breaking the barriers to successful refactoring: observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 421–430. https://doi.org/10.1145/1368088.1368146

[47] Jürg Nievergelt. 1965. On the automatic simplification of computer programs. *Commun. ACM* 8, 6 (1965), 366–370. https://doi.org/10.1145/364955.364963

[48] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 23 (jun 2016), 53 pages. https://doi.org/10.1145/2932631

[49] Saeed Parsa, Morteza Zakeri-Nasrabadi, Masoud Ekhtiarzadeh, and Mohammad Ramezani. 2023. Method name recommendation based on source code metrics. *Journal of Computer Languages* 74 (2023), 101177. https://doi.org/10.1016/j.cola.2022.101177

[50] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical*

Software Engineering 27, 1 (2022), 1–43. https://doi.org/10.1007/s10664-021-10045-x

[51] Matt Post. 2018. A Call for Clarity in Reporting BLEU Scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Mark Fishel, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Christof Monz, Matteo Negri, Aurélie Névéol, Mariana Neves, Matt Post, Lucia Specia, Marco Turchi, and Karin Verspoor (Eds.). Association for Computational Linguistics, Brussels, Belgium, 186–191. https://doi.org/10.18653/v1/W18-6319

[52] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[53] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3520312.3534869

[54] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1516–1528. https://doi.org/10.1109/ASE56229.2023.00117

[55] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv e-prints*, Article arXiv:2009.10297 (Sept. 2020), arXiv:2009.10297 pages. https://doi.org/10.48550/arXiv.2009.10297 arXiv:2009.10297 [cs.SE]

[56] Will Shackleton, Katriel Cohn-Gordon, Peter C Rigby, Rui Abreu, James Gill, Nachiappan Nagappan, Karim Nakad, Ioannis Papagiannis, Luke Petre, Giorgi Megreli, et al. 2023. Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. publisher, address, 1705–1715. https://doi.org/10.1145/3611643.3613871

[57] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1597–1608. https://doi.org/10.1145/3510003.3510060

[58] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802. https://doi.org/10.1109/TSE.2020.2968072

[59] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[60] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic Programming for Shader Simplification. *ACM Trans. Graph.* 30, 6 (dec 2011), 1–12. https://doi.org/10.1145/2070781.2024186

[61] Open source developers. 2015. *GH Archive*. GitHub. https://www.gharchive.org/

[62] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

[63] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/3510003.3510067

[64] Nikolaos Tsantalis. 2024. *RefactoringMiner*. Concordia University. https://github.com/tsantalis/RefactoringMiner

[65] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 4–14. https://doi.org/10.1109/SANER.2018.8330192

[66] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[67] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[68] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes Via Neural Machine Translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 25–36. https://doi.org/10.1109/ICSE.2019.00021

[69] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages. https://doi.org/10.1145/3340544

[70] Danny van Bruggen. 2019. *JavaParser*. GitHub. https://github.com/javaparser/javaparser

[71] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 382–394. https://doi.org/10.1145/3540250.3549113

[72] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 881–892. https://doi.org/10.1145/3468264.3468625

[73] Haibo Wang. 2024. *Artifact*. Concordia University. https://anonymous.4open.science/r/Automated-Program-Simplification-CCC6/

[74] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. (Nov. 2021), 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[75] Phillip Wong and Mengjie Zhang. 2006. Algebraic simplification of GP programs during evolution. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (Seattle, Washington, USA) *(GECCO '06)*. Association for Computing Machinery, New York, NY, USA, 927–934. https://doi.org/10.1145/1143997.1144156

[76] W. Eric Wong, Hira Agrawal, and Xiangyu Zhang. 2023. Slicing-Based Techniques for Software Fault Localization. *Handbook of Software Fault Localization: Foundations and Advances* (2023), 135–200. https://doi.org/10.1002/9781119880929.ch3 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119880929.ch3

[77] Tong Xiao, Tian Xia, Yi Yang, Chang Huang, and Xiaogang Wang. 2015. Learning from massive noisy labeled data for image classification . In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2691–2699. https://doi.org/10.1109/CVPR.2015.7298885

[78] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (April 2023), 29 pages. https://doi.org/10.1145/3586049

[79] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 253–267. https://doi.org/10.1145/318774.318946

[80] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[81] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/3551349.3556955

[82] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 261–273. https://doi.org/10.1145/3650212.3652126

[83] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable? a study of API misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 886–896. https://doi.org/10.1145/3180155.3180260

Received 2024-09-07; accepted 2025-01-14