

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Learning Syntactic Program Transformations from Examples

Reudismam Rolim de Sousa

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Gustavo Araújo Soares, and Rohit Gheyi

(Orientadores)

Campina Grande, Paraíba, Brasil

©Reudismam Rolim de Sousa, 2 de Agosto de 2018

S7251

Sousa, Reudismam Rolim de.

Learning syntactic program transformations from examples /
Reudismam Rolim de Sousa. - Campina Grande-PB, 2018.
159 f. : il. color.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de
Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.

"Orientação: Prof. Dr. Gustavo Araújo Soares, Prof. Dr. Rohit Gheyi".

Referências.

1. Program Transformation. 2. Quick Fix. 3. Program Synthesis. 4.
Tutoring Systems. 5. Refactoring. I. Soares, Gustavo Araújo. II. Gheyi,
Rohit. III. Título.

CDU 004.41(043)

Resumo

Ferramentas como ErrorProne, ReSharper e PMD ajudam os programadores a detectar e/ou remover automaticamente vários padrões de códigos suspeitos, possíveis bugs ou estilo de código incorreto. Essas regras podem ser expressas como *quick fixes* que detectam e reescrevem padrões de código indesejados. No entanto, estender seus catálogos de regras é complexo e demorado. Nesse contexto, os programadores podem querer executar uma edição repetitiva automaticamente para melhorar sua produtividade, mas as ferramentas disponíveis não a suportam. Além disso, os projetistas de ferramentas podem querer identificar regras úteis para automatizarem. Fenômeno semelhante ocorre em sistemas de tutoria inteligente, onde os instrutores escrevem transformações complicadas que descrevem "falhas comuns" para consertar submissões semelhantes de estudantes a tarefas de programação. Nesta tese, apresentamos duas técnicas. REFAZER, uma técnica para gerar automaticamente transformações de programa. Também propomos REVISAR, nossa técnica para aprender quick fixes em repositórios. Nós instanciamos e avaliamos REFAZER em dois domínios. Primeiro, dados exemplos de edições de código dos alunos para corrigir submissões de tarefas incorretas, aprendemos transformações para corrigir envios de outros alunos com falhas semelhantes. Em nossa avaliação em quatro tarefas de programação de setecentos e vinte alunos, nossa técnica ajudou a corrigir submissões incorretas para 87% dos alunos. No segundo domínio, usamos edições de código repetitivas aplicadas por desenvolvedores ao mesmo projeto para sintetizar a transformação de programa que aplica essas edições a outros locais no código. Em nossa avaliação em 56 cenários de edições repetitivas de três grandes projetos de código aberto em C#, REFAZER aprendeu a transformação pretendida em 84% dos casos e usou apenas 2.9 exemplos em média. Para avaliar REVISAR, selecionamos 9 projetos e REVISAR aprendeu 920 transformações entre projetos. Atuamos como projetistas de ferramentas, inspecionamos as 381 transformações mais comuns e classificamos 32 como *quick fixes*. Para avaliar a qualidade das quick fixes, realizamos uma *survey* com 164 programadores de 124 projetos, com os 10 quick fixes que apareceram em mais projetos. Os programadores suportaram 9 (90%) quick fixes. Enviamos 20 *pull requests* aplicando quick fixes em 9 projetos e, no momento da escrita, os programadores apoiaram 17 (85%) e aceitaram 10 delas.

Abstract

Tools such as ErrorProne, ReSharper, and PMD help programmers by automatically detecting and/or removing several suspicious code patterns, potential bugs, or instances of bad code style. These rules could be expressed as quick fixes that detect and rewrite unwanted code patterns. However, extending their catalogs of rules is complex and time-consuming. In this context, programmers may want to perform a repetitive edit into their code automatically to improve their productivity, but available tools do not support it. In addition, tool designers may want to identify rules helpful to be automated. A similar phenomenon appears in intelligent tutoring systems where instructors have to write cumbersome code transformations that describe “common faults” to fix similar student submissions to programming assignments. In this thesis, we present two techniques. REFAZER, a technique for automatically generating program transformations. We also propose REVISAR, our technique for learning quick fixes from code repositories. We instantiate and evaluate REFAZER in two domains. First, given examples of code edits used by students to fix incorrect programming assignment submissions, we learn program transformations that can fix other students’ submissions with similar faults. In our evaluation conducted on four programming tasks performed by seven hundred and twenty students, our technique helped to fix incorrect submissions for 87% of the students. In the second domain, we use repetitive code edits applied by developers to the same project to synthesize a program transformation that applies these edits to other locations in the code. In our evaluation conducted on 56 scenarios of repetitive edits taken from three large C# open-source projects, REFAZER learns the intended program transformation in 84% of the cases and using only 2.9 examples on average. To evaluate REVISAR, we select 9 projects, and REVISAR learns 920 transformations across projects. We acted as tool designers, inspected the most common 381 transformations and classified 32 as quick fixes. To assess the quality of the quick fixes, we performed a survey with 164 programmers from 124 projects, showing the 10 quick fixes that appeared in most projects. Programmers supported 9 (90%) quick fixes. We submitted 20 pull requests applying our quick fixes to 9 projects and, at the time of the writing, programmers supported 17 (85%) and accepted 10 of them.

Agradecimentos

Gostaria de agradecer a Deus por me guiar nessa jornada;
aos meus pais Maria do Socorro Rolim de Sousa e Deusimar Antonio de Sousa;
ao meu filho Davi Lukas Oliveira Rolim e a minha esposa Luciana Oliveira Rolim;
aos meus familiares;
aos meus orientadores Gustavo Soares e Rohit Gheyi, pelos ensinamentos durante o doutorado;
aos membros da minha banca de doutorado Alessandro Garcia, Loris D'Antoni, Tiago Massoni e Everton Alves, pelas valiosas sugestões;
aos colegas do *Software Productivity Group* - SPG;
e a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e a Coordenação de Pós-graduação em Informática (COPIN).

Contents

1	Introduction	1
1.1	Problem	2
1.1.1	Automating edits	2
1.1.2	Learning quick fixes	5
1.2	Solution	7
1.2.1	Automating edits	8
1.2.2	Learning quick fixes	9
1.3	Evaluation	10
1.3.1	REFAZER	11
1.3.2	REVISAR	12
1.4	Contributions	13
1.5	Organization	13
2	Background	15
2.1	Program synthesis	15
2.1.1	Inductive programming	16
2.1.2	PROSE	17
2.2	Program transformation	25
2.2.1	Zhang and Shasha	26
2.2.2	Chawathe et al.	29
2.2.3	ChangeDistiller	31
2.2.4	GumTree	33

3	State-of-the-art	36
3.1	Motivation	36
3.1.1	Review questions	37
3.2	Review methods	38
3.2.1	Search strategy	38
3.2.2	Search string	40
3.2.3	Selection methodology	40
3.2.4	Quality assessment	41
3.2.5	Inclusion and exclusion criteria	41
3.2.6	Information extraction	42
3.2.7	Execution	44
3.3	Results	44
3.4	Discussion	45
3.4.1	Programming by examples	45
3.4.2	Linked editing	46
3.4.3	API usage	47
3.4.4	Bug fixing	49
3.4.5	Complex refactoring	50
3.4.6	Complex transformations	52
3.4.7	Validation	53
4	REFAZER	54
4.1	A DSL for AST transformations	55
4.1.1	Synthesis algorithm	59
4.1.2	Ranking	60
4.2	Evaluation	61
4.2.1	Fixing introductory programming assignments	62
4.3	Results and discussion	65
4.3.1	Applying repetitive edits to open-source C# projects	68
4.3.2	Threats to validity	72
4.4	Answer to the researches questions	73

4.5	Concluding remarks	74
5	REVISAR	75
5.1	Extracting concrete AST edits	75
5.2	Learning transformations	78
5.3	Clustering concrete edits	82
5.4	Effectiveness of REVISAR	84
5.5	Usefulness of the edit patterns learned by REVISAR	86
5.5.1	Catalog of quick fixes	87
5.5.2	A survey of programmers' opinion about our catalog	91
5.5.3	Pull Requests	96
5.5.4	Threats to validity	98
5.6	Related Work	98
5.7	Concluding remarks	101
6	Conclusion	102
6.1	Follow up approaches	104
6.2	Future work	104
A	Search string for each datasource	128
A.1	ACM search string for RQ 01	128
A.1.1	ACM search string based on title	128
A.1.2	ACM search string based on abstract	129
A.1.3	ACM search string based on keywords	129
A.2	Scopus search string for RQ 01	129
A.3	Engineering village search string for RQ 01	130
A.4	ScienceDirect search string for RQ 01	130
A.5	IEEE search string for RQ 01	130
A.5.1	IEEE search string based on abstract	130
A.5.2	IEEE search string based on title and keywords	131
B	Catalog	132
B.1	String to character	132

B.2	Prefer string literal equals method	133
B.3	Avoid using FileInputStream/FileOutputStream	133
B.4	Use valueOf instead wrapper constructor	133
B.5	Use collection isEmpty	134
B.6	StringBuffer to StringBuilder	135
B.7	Infer type in generic instance creation	135
B.8	Remove raw type	136
B.9	Field, parameter, local variable could be final	136
B.10	Avoid using strings to represent paths	137
B.11	Prefer Class<?>	137
B.12	Use variadic functions	138
B.13	Prefer string constant equals method	138
B.14	Add @Nullable for parameters that accept null	139
B.15	Use primitive type termination in literals	139
B.16	Promote inner class	140
B.17	Cannot use casts or instanceof with parameterized types	141
B.18	Use Stopwatch	141
B.19	Use HH in SimpleDateFormat	141
B.20	Remove this for unambiguous variables	142
B.21	Use modern log framework instead of Log4J	143
B.22	Remove redundant cast	143
B.23	Use Vector methods compatible with Java Collections interface	144
B.24	Replace Google Truth.FailureStrategy to Google Truth.FailureMetadata	145
B.25	Remove default value for default initializer	145
B.26	Update Calcite library	146
B.27	Use trace instead of debug, when want to log precise data	146
B.28	If an error is thrown, use Throwable instead of Exception	147
B.29	Return a boolean instead of void	147
B.30	Remove non-reachable @SuppressWarnings	148
B.31	Use fail instead of assertTrue(false)	148
B.32	Do not use new as argument of a method call.	149

C Survey

150

Lista de Símbolos

API - Application Programming Interface

AST - Abstract Syntactic Tree

CFG - Context-free grammar

CS - Computer Science

DSL - Domain-Specific Language

IDE - Integrated Development Environment

IP - Inductive Programming

LCS - Longest Common Subsequence

LHS - Left-hand side

MOOC - Massive Open Online Courses

PBE - Programming-by-example

RHS - Right-hand side

PS - Primary Study

RQ - Research Question

SD - Standard Deviation

SR - Systematic Review

VS - Version Space

VSA - Version Space Algebra

TA - Teacher Assistant

ω_F - Witness Function

List of Figures

1.1	An example of a common fault made by different students, two similar edits that can fix different programs, and a program transformation that captures both edits.	4
1.2	Repetitive edits applied to the Roslyn source code to perform a refactoring.	5
1.3	Concrete edits applied to the Apache Ant.	7
1.4	quick fix for the concrete edits in Figure 1.3.	10
2.1	Normalizing names by examples using FlashFill.	17
2.2	Grammar for <code>SubStr</code> operator.	19
2.3	Data for the extract last name problem.	21
2.4	Version space algebra for the extract last name problem.	22
2.5	Example of a forest for an AST.	27
2.6	Example of leaf most descendants. In this example, the node number 1 is leaf most descendants of nodes number 1, 3, and 9.	27
2.7	Trees to compute edit script, showing key roots.	28
2.8	An example of a match for a small tree.	32
2.9	An example where a node x in T_1 matches multiples nodes in T_2	32
3.1	Number of PS for each event.	44
4.1	The workflow of REFAZER. It receives an example-based specification of edits as input and returns a set of transformations that satisfy the examples.	55
4.2	A core DSL \mathcal{L}_T for describing AST transformations. <i>kind</i> ranges over possible AST kinds of the underlying programming language, and <i>value</i> ranges over all possible ASTs. <i>s</i> and <i>k</i> range over strings and integers, respectively.	55

4.3	An example of a synthesized transformation and its application to a C# program, which results in a list of edits.	57
4.4	Analysis of the first time REFAZER can fix a student submission for the 50 students with most attempts for two benchmark problems. Blue: submissions that might be avoided by showing feedback from a fix generated by REFAZER.	67
4.5	Issue submitted to NuGet project to use the method (ExtendedSqlAzureExecutionStrategy.ExecuteNew) consistently.	71
5.1	REVISAR's work-flow.	75
5.2	Before-after version for the first edited line of code.	76
5.3	Concrete edits and their input-output templates.	78
5.4	Incompatible concrete edits.	81
5.5	Distribution of concrete edits per edit pattern.	85
5.6	Preference of programmers for each quick fix.	93
5.7	Distribution of answers for tools.	94
B.1	Quick Fix 1: String to character	132
B.2	Quick Fix 2: Prefer string literal equals method	133
B.3	Quick Fix 3: Avoid using FileInputStream/FileOutputStream	134
B.4	Quick Fix 4: Use valueOf instead wrapper constructor	134
B.5	Quick Fix 5: Use collection isEmpty	135
B.6	Quick Fix 7: StringBuffer to StringBuilder	135
B.7	Quick Fix 7: Allow type inference for generic instance creation	136
B.8	Quick Fix 8: Remove raw type	136
B.9	Quick Fix 9: Field, parameter, local variable could be final	137
B.10	Quick Fix 10: Use Path to represent file path	137
B.11	Quick Fix 11: Prefer Class<?>	138
B.12	Quick Fix 12: Use variadic functions	138
B.13	Quick Fix 13: Prefer string literal equals method	139
B.14	Quick Fix 14: Add @Nullable for parameters that accept null	139
B.15	An example of a numeric problem associated with numeric literal use.	140
B.16	Quick Fix 15: Use primitive type termination in literals	140

B.17 Quick Fix 16: Promote inner class	140
B.18 Quick Fix 17: Use collection <code>isEmpty</code>	141
B.19 Quick Fix 18: Use <code>Stopwatch</code>	142
B.20 Quick Fix 19: Use <code>HH</code> in <code>SimpleDateFormat</code>	142
B.21 Quick Fix 20: Remove this for unambiguous variables.	143
B.22 Quick Fix 21: Use modern log framework instead of <code>Log4J</code>	143
B.23 Quick Fix 22: Remove redundant cast.	144
B.24 Quick Fix 23: Use <code>Vector</code> methods compatible with <code>Java Collections interface</code> .144	
B.25 Quick Fix 24: Replace <code>Google Truth.FailureStrategy</code> to <code>Google Truth.FailureMetadata</code>	145
B.26 Quick Fix 25: Remove default value for default initializer.	145
B.27 Quick Fix 26: Update <code>Calcite</code> library.	146
B.28 Quick Fix 27: Use <code>trace</code> instead of <code>debug</code> , when want to log precise data. . .	147
B.29 Quick Fix 28: If an error is thrown, use <code>Throwable</code> instead of <code>Exception</code> .147	
B.30 Quick Fix 29: Return a <code>boolean</code> instead of <code>void</code>	148
B.31 Quick Fix 30: Remove non-reachable <code>@SuppressWarnings</code>	148
B.32 Quick Fix 31: If an error is throw, use <code>Throwable</code> instead of <code>Exception</code> . 149	
B.33 Quick Fix 32: Do not use <code>new</code> as argument of a method call.	149

List of Tables

3.1	Studies from the exploratory study.	38
3.2	Synonyms and alternative spelling.	39
3.3	Criteria for our systematic review.	39
3.4	Quality assessment score.	41
3.5	Inclusion and exclusion criteria.	42
3.6	Default information collected from each primary study.	43
4.1	Our benchmarks and incorrect student submissions.	65
4.2	Summary of results for RQ1. “Incorrect submissions” = mean (SD) of submissions per student; “Students” = % of students with solution fixed by REFAZER; “Submissions” = mean (SD) of submissions required to find the fix.	66
4.3	Summary of results for RQ2.	67
4.4	Evaluation Summary. Scope = scope of the transformation; Ex. = examples; Dev. = locations modified by developers; REFAZER = locations modified by REFAZER. Outcomes: ✓ = it performed the same edits as the developers; ★ = it performed more edits than the developers (manually validated as correct); ✗ = it performed incorrect edits; “—” = it did not synthesize a transformation.	69
5.1	Projects used to detect edit patterns.	85
5.2	quick fixes. Number of projects associated to each quick fix. Number of edits that generate the edit pattern corresponding to the quick fix and number of revisions associated to the edit pattern.	88
5.3	Projects used to submit pull requests.	96

List of Algorithms

1	Semantic function $\text{SubStr}(x, pos_1, pos_2)$	20
2	Semantic function for operator $\text{AbsPos}(x, k)$	20
3	Semantic function $\text{RelPos}(x, r_1, r_2, k)$	20
4	Witness function to first parameter of operator $\text{SubStr}(x, pos, pos)$	23
5	Witness function to second parameter of operator $\text{SubStr}(x, pos, pos)$	23
6	Witness function to first parameter of operator $\text{AbsPos}(x, k)$	23
7	Witness function to first parameter of operator $\text{RelPos}(x, r, r, k)$	24
8	Witness function to second parameter of operator $\text{RelPos}(x, r, r, k)$	24
9	Witness function to third parameter of operator $\text{RelPos}(x, r, r, k)$	24
10	Ranking function for $\text{SubStr}(score_x, score_{pos_1}, score_{pos_2})$	25
11	Ranking function for $\text{AbsPos}(score_x, score_{pos_k})$	25
12	Ranking function for $\text{RelPos}(score_x, scorer_1, scorer_2, score_k)$	25
13	Zhang and Shasha algorithm.	28
14	Algorithm to compute treedist	29
15	Matching Chawathe et al. algorithm.	30
16	Edit Script Chawathe et al. algorithm.	31
17	GumTree top-down algorithm.	35
18	GumTree bottom-up algorithm.	35
19	Backpropagation procedure for the DSL operator $\text{Transformation}(rule_1, \dots, rule_n)$	61
20	Greedy clustering.	82

Chapter 1

Introduction

As software evolves, programmers edit the source code to add features, fix bugs, or refactor it. Many such *edits* have already been performed in the past by the same programmers in a different codebase location, or by other programmers in a different program/codebase. For instance, to apply an API update, a programmer needs to locate all references to the old API and consistently replace them with the new API [83, 148]. As another example, in programming courses, student submissions that exhibit the same fault often need similar fixes. For large classes such as *Massive Open Online Courses* (MOOCs), manually providing feedback to different students is an unfeasible burden on the teaching staff.

Since applying repetitive edits manually is tedious and error-prone, programmers often strive to automate them. The space of tools for automation of repetitive code edits contains Integrated Development Environments (IDEs), static analyzers, and various domain-specific engines. IDEs, such as Visual Studio [87] or Eclipse [35], include features that automate some code *transformations*, such as adding boilerplate code (e.g., equality comparisons) and code refactoring (e.g., *Rename*, *Extract Method*). Static analyzers, such as ReSharper [51], Coverity [6], ErrorProne [39], and Clang-tidy [38] automate the removal of suspicious code patterns, potential bugs, and verbose code fragments. In an education context, AutoGrader [135] uses a set of transformations provided by an instructor to fix common faults in introductory programming assignments.

Other tools do not apply repetitive edits but make programmers aware of code patterns that could be problematic. For instance, static analyzers such as Checkstyle [14], PMD [120], and FindBugs [4] warn about a bad code pattern, allowing programmers to fix them before they

cause problems later in the software evolution. For example, PMD can detect instances where the method `size` is used to check whether a list is empty and proposes to replace such instance with the method `isEmpty`. For the majority of collections, these two ways to check emptiness are equivalent, but for other certain collections—e.g., `ConcurrentSkipListSet`—computing the size of a list is not a constant-time operation [105]. We refer to this kind of edit patterns as *quick fixes*.

1.1 Problem

All aforementioned tool families rely on predefined catalogs of recognized rules, which are hard to extend. Furthermore, since the number of possible rules for a language such as Java is huge, it is unfeasible to enumerate all possible rules that could be useful for programmers. Thus, the catalog of rules of code analysis tools may be incomplete.

In this context, there are two problems (i) programmers may want to perform a repetitive edit into their code automatically to improve their productivity, but available tools do not support it, and (ii) it is difficult for designers of code analysis tools to identify rules that could be helpful to be automated. In the first problem, we want to provide a few examples since programmers may not like to provide many examples to perform transformations. In the second, it is needed to identify what programmers are performing on their source code to identify transformations that would be likely to be helpful for them.

We start by describing the first problem in Section 1.1.1. Then, we describe the problem of discovering quick fixes to be included in the catalog of rules of code analysis tools in Section 1.1.2.

1.1.1 Automating edits

Our key observation is that code edits gathered from repositories and version control history constitute *input-output examples* for learning program transformations.

The main challenge of example-based learning lies in abstracting concrete code edits into classes of *transformations* representing these edits. For instance, Figure 1.1a shows sim-

ilar edits performed by different students to fix the same fault in their submissions for a programming assignment. Although the edits share some structure, they involve different expressions and variables. Therefore, a transformation should partially abstract these edits as in Figure 1.1d.

However, examples are highly ambiguous, and many different transformations may satisfy them. For instance, replacing `<name>` by `<exp>` in the transformation will still satisfy the examples in Figure 1.1a. In general, learning either the most specific or the most general transformation is undesirable, as they are likely to respectively produce false negative or false positive edits on unseen programs. Thus, we need to (i) learn and store a *set* of consistent transformations efficiently, and (ii) rank them with respect to their trade-offs between over-generalization and over-specialization. Here, a consistent transformation is the one that produces the intended output of each input example. To resolve these challenges, we leverage state-of-the-art software engineering research to learn such transformations automatically using *Inductive Programming* (IP), or *Programming-by-Example* (PBE) [44], which has been successfully applied to many domains, such as text transformation [42], data cleaning [66], and layout transformation [26].

To motivate the problem, we start by describing two motivating examples of repetitive program transformations. First, we present a motivating example on the domain of programming courses. Then, we present a motivating example on the domain of applying repetitive edits to open-source C# projects.

Fixing programming assignment submissions

In introductory programming courses, assignments are often graded using a test suite, provided by the instructors. However, many students struggle to understand the fault in their code when a test fails. To provide more detailed feedback (e.g., fault location or its description), teachers typically compile a *rubric* of common types of faults and detect them with simple checks. With a large variety of possible faults, manually implementing these checks can be laborious and error-prone.

However, many faults are common and exhibit themselves in numerous unrelated student submissions. Consider the Python code in Figure 1.1a. It describes two submission attempts to solve a programming assignment in the course “The Structure and Interpretation of Computer

```

1 def product(n, term):
2     total, k = 1, 1
3     while k<=n:
4 -     total = total*k
5 +     total = total*term(k)
6         k = k+1
7     return total

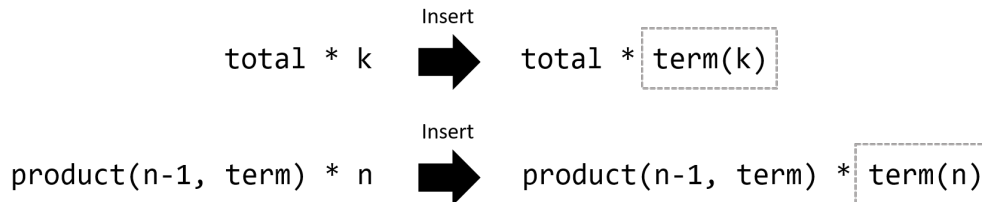
```

```

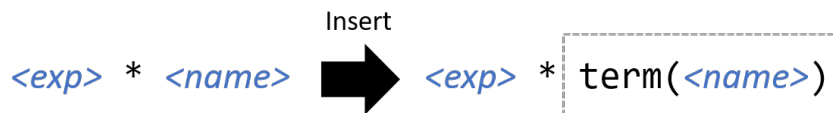
1 def product(n, term):
2     if (n==1):
3         return 1
4 -     return product(n-1, term)*n
5 +     return product(n-1,
6         term)*term(n)

```

(a) An edit applied by a student to fix the program. (b) An edit applied by another student fixing the same fault.



(c) Similar tree edits applied in (a) and (b), respectively. Each edit inserts a concrete subtree to the right-hand side of the * operator. The two edits share the same structure but involve different variables and expressions.



(d) A rewrite rule that captures the two edits in (a) and (b).

Figure 1.1: An example of a common fault made by different students, two similar edits that can fix different programs, and a program transformation that captures both edits.

Programs” (CS61A) at the University of California, Berkeley,¹ an introductory programming class with more than 1,500 enrolled students. In this assignment, the student is asked to write a program that computes the product of the first n terms, where `term` is a function. The original code, which includes line 4 instead of line 5, is an incorrect submission for this assignment, and the subsequent student submission fixes it by replacing line 4 with line 5. Notably, the fault illustrated in Figure 1.1 was a common fault affecting more than one hundred students in the Spring semester of 2016, and Figure 1.1b shows a recursive algorithm proposed by a different student with the same fault.

To alleviate the burden of compiling manual feedback, we propose to automatically learn the rubric checks from the student submissions. Existing tools for such automatic learning [82, 83] cannot generate a transformation that is general enough to represent both the edits shown in Figure 1.1c due to their limited forms of abstraction.

¹<http://cs61a.org/>

Repetitive codebase edits

The second motivating example is found in Roslyn, Microsoft’s library for compilation and code analysis for C# and VB.NET. Consider the edits shown in Figure 1.2, where, for every comparison instance with an object returned by the method `CSharpKind`, the programmer replaces the `==` operator with an invocation of the new method `IsKind`, and passes the right-hand side expression as the method’s argument. Such refactoring is beyond abilities of IDEs due to its context sensitivity.

When we analyzed the revision [8c14644](https://github.com/dotnet/roslyn/commit/8c14644)² in the Roslyn repository, we observed that the programmer applied this edit to 26 locations in the source code. However, the transformation generated by our solution (Chapter 5) applied this edit to 718 more locations. After we presented the results to the Roslyn programmers, they confirmed that the locations should have been covered in the original revision.

```
1 - while (receiver.CSharpKind() == SyntaxKind.ParenthesizedExpression)
2 + while (receiver.IsKind(SyntaxKind.ParenthesizedExpression))
3
4 - foreach (var m in modifiers) {
5 -     if (m.CSharpKind() == modifier) return true;
6 - }
7 + foreach (var m in modifiers) {
8 +     if (m.IsKind(modifier)) return true;
9 + }
```

Figure 1.2: Repetitive edits applied to the Roslyn source code to perform a refactoring.

1.1.2 Learning quick fixes

So far, we describe the problem of allowing programmers to perform edits that are not included in the catalog of code analysis tools. Now, we describe the problem of discovering quick fixes.

As discussed, the code analysis tools rely on a predefined catalog of quick fixes (usually expressed as rules), each used to detect and potentially fix a pattern. This catalog has to be updated often due to the addition of new language features (e.g., new constructs in new versions of Java), new style guidelines, or simply due to the discovery of new bad patterns. However, the task of coming up with what transformations are useful, what transformations

²<https://github.com/dotnet/roslyn/commit/8c14644>

are common, and integrating these transformations in the code analysis tools is challenging, time-consuming, and is currently performed in an ad-hoc fashion—i.e., new rules for quick fixes are added on a by-need basis. For these tools, some rules are detected by analyzing bug reports. For instance, the null pointer exception detector in FindBugs tool was inspired by a bug in Eclipse 2.1.0 (bug number 35769). Other rules are identified by other programmers and submitted as pull requests for analysis. For instance, ErrorProne often receives pull requests to improve its catalog.

The lack of a systematic way of discovering new quick fixes makes it hard for code analyzers to stay up-to-date with the latest code practices and language features.

In this problem, we also have the challenge of abstracting concrete code edits into classes of transformations representing these edits and the challenge of examples' ambiguity since different transformations may satisfy them (see Section 1.1.1). However, we have additional challenges (i) we want to learn the examples in an unsupervised way without programmers' guidance, and (ii) we want to scale since we may have to analyze many examples to learn the transformations.

Our key observation is that we can “discover” quick fixes by observing how programmers modify code in real repositories with large user bases. In particular, a transformation that is performed by many programmers across many projects is likely to reveal a good quick fix.

We start by describing a motivating example of an edit that occurs across projects.

Motivating example

Consider the edit applied to the code in Figure 1.3. The edit was performed in the Apache Ant source code.³ In the presented pattern, the original code contains three expressions of the form `x.equals("str")` that compare a variable `x` of type `String` to a string literal `"str"`. Since the variable `x` may contain a `null` value, evaluating this expression may cause a `NullPointerException`. In this particular revision, a programmer from this project addresses the issue by exchanging the order of the arguments of the `equals` method—i.e.,

³<https://github.com/apache/ant/commit/b7d1e9b>

by calling the method on the string literal. This edit fixes the issue since the `equals` checks whether the parameter is `null`. This type of edit is a common one, and we discovered it occurs in three industrial open source projects across GitHub repositories: Apache Ant, Apache Hive, and Google ExoPlayer. Given that the pattern appears in such large repositories, it makes sense to assume that it could also be useful to other programmers who may not know about it. Despite its usefulness, a quick fix rule for this edit is not included in the catalog of largely used code analysis tools, such as ErrorProne, FindBugs, and PMD. Remarkably, even though the edit is applied in Google repositories, this pattern does not appear in ErrorProne, a code analyzer developed by Google that is internally run on the Google's code base.

```
1 //...
2 - } else if (args[i].equals("--launchdiag")) {
3 + } else if ("--launchdiag".equals(args[i])) {
4     launchDiag = true;
5 - } else if (args[i].equals("--noclasspath")
6 -     || args[i].equals("--noclasspath")) {
7 + } else if ("--noclasspath".equals(args[i])
8 +     || "--noclasspath".equals(args[i])) {
9     noClassPath = true;
10 - } else if (args[i].equals("-main")) {
11 + } else if ("-main".equals(args[i])) {
12 //...
```

Figure 1.3: Concrete edits applied to the Apache Ant.

1.2 Solution

We now describe two solutions to deal with the problem of the incompleteness of code analysis tools. In Section 1.2.1, we describe our solution to perform repetitive edits that are not in the catalog of code analysis tools. To solve this problem, we want to provide a few examples since programmers may not like to provide many examples to perform transformations. Moreover, few examples increase the number of transformations that satisfy them. We need to search this space to identify the correct one. The most general or the most specific transformation is undesirable.

In Section 1.2.2, we describe our solution to discover transformations that could be helpful to be included in the catalog of code analysis tools. To solve this problem, we have to compare a huge number of edits to identify the ones that describe transformations. In the previous

problem, we usually have few examples that are instances of the same repetitive edit. Here, the edits interchange repetitive edits that could come from different programmers, revisions, and projects. Therefore, many different combinations of concrete edits will need to be compared to identify the ones that describe the same transformation. Therefore, maintaining a space of possible transformations would not scale for that problem. In this problem, we focus on finding the least general transformation, and we rely on the assumption that we have enough examples to abstract concrete edits into the correct transformations.

1.2.1 Automating edits

To deal with the problem of allowing programmers to perform repetitive edits that are not in the catalog of code analysis tools, we propose REFAZER,⁴ an IP technique for synthesizing program transformations from examples. REFAZER is based on the PROSE [121] Inductive Programming framework. This framework has been successfully used in the data wrangling domain. For instance, FlashFill and FlashExtract are examples of techniques that use this framework. FlashFill is a feature in Microsoft Excel 2013 that synthesizes string transformation macros from input-output examples while FlashExtract is a tool for data extraction from semi-structured text files, deployed in Microsoft PowerShell for Windows 10.

We specify a *Domain-Specific Language* (DSL) that describes a rich space of program transformations that commonly occur in practice. In our DSL, a program transformation is defined as a sequence of distinct *rewrite rules* applied to the *Abstract Syntax Tree* (AST). Each rewrite rule matches some subtrees of the given AST and outputs modified versions of these subtrees. For instance, in REFAZER, the transformation in Figure 1.1 is described as a rewrite rule shown in Figure 1.1d. This rewrite rule pattern matches any subtree of the program's AST whose root is a `*` operation with a variable as the second operand and inserts a `term` application on top of that variable. Notice that the rewrite rule abstracts both the variable name and the first operand of the `*` operator. Additionally, we specify constraints for our DSL operators based on the input-output examples to reduce the search space of transformations, allowing PROSE to efficiently synthesize them. Finally, we define functions to rank the synthesized transformations based on their DSL structure.

⁴<http://www.dsc.ufcg.edu.br/~spg/refazer/>

Additionally, we use synthesis algorithms to create the set of programs following this DSL. The synthesis problem starts with an example-based specification that contains inputs and desired outputs. REFAZER uses PROSE, which synthesizes a set of programs following the DSL structure. All programs in this set are consistent with the example-based specification. The methodology used by PROSE is backpropagation [121]. In this methodology, a synthesis problem for a rule in the DSL is reduced to several subproblems for each parameter of this rule, which are then solved recursively. The backpropagation algorithm uses modular operator-specific annotations called *witness functions*. Although PROSE includes many generic operators such as `Filter` and `Map`, most operators in the DSL are domain-specific. Therefore, the synthesis designer has to have non-trivial domain-specific insight to create *witness functions* for the backpropagation procedure.

Finally, we define ranking algorithms. The number of transformations learned could be huge ($> 10^{20}$). Even though the learned transformations are all consistent with the input-output examples, some of these program transformations may be incorrect to unseen instances. Thus, we need to favor the transformations that have a high likelihood of it being correct in general.

1.2.2 Learning quick fixes

So far, we described a solution to allow programmers to perform edits that are not included in the catalog of code analysis tools. Now, we describe a solution for learning quick fixes that could be useful to be added to the catalog of code analysis tools.

To resolve this problem, we propose REVISAR,⁵ a technique for learning quick fixes from code repositories. Given code repositories as input, REVISAR identifies transformations by comparing consecutive revisions in the revision histories. The most common transformations—i.e., those performed across multiple projects—can then be inspected to detect useful ones and add the corresponding rules to code analyzers. For example, REVISAR was able to *automatically* analyze the concrete edits in Figure 1.3 and generate the quick fix in Figure 1.4. We also sent pull requests applying this quick fix to other parts of the code in the Apache Ant and Google ExoPlayer projects, and these pull requests were accepted.

⁵<https://reudismam.github.io/Revisar/>

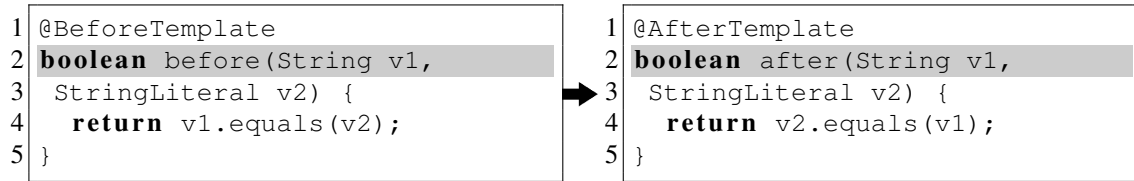


Figure 1.4: quick fix for the concrete edits in Figure 1.3.

Since we are interested in detecting patterns appearing across projects and revisions, REVISAR has to analyze large amounts of code, a task that requires efficient and precise algorithms. REVISAR focuses on edits performed to individual code locations and uses GumTree [31], a tree edit distance algorithm, to efficiently extract concrete abstract-syntax-tree edits from individual pairs of revisions—i.e., sequences of tree operations such as insert, delete, and update. REVISAR then uses a greedy algorithm to detect subsets of concrete edits that can be described using the same transformation. To perform this last task efficiently, REVISAR uses a variant of a technique called *anti-unification* [63], which is commonly used in inductive logic programming. Given a set of concrete edits, the anti-unification algorithm finds the least general generalization of the two edits—i.e., the largest pattern shared by the edits.

Revisar follows Refaster [148] syntax (see Figure 1.4). The left-hand side (i.e., annotated with `@BeforeTemplate`) matches nodes to be edited. In the presented transformation, the before version matches a call to the `equals` method where `v1` is a `String` and `v2` is a string literal. The after version (i.e., annotated with `@AfterTemplate`) is also a call to the `equals` method, but exchanges the order of the `equals` method elements.

1.3 Evaluation

We now describe the evaluation of our techniques. We start with the evaluation of REFAZER. Then, we describe the evaluation of REVISAR.

1.3.1 REFAZER

We evaluated REFAZER in two domains: learning transformations to fix submissions for introductory programming assignments and learning transformations to apply repetitive edits to large codebases.

Our first experiment is motivated by large student enrollments in Computer Science courses, e.g., in MOOCs, where automatically grading student submissions and providing personalized feedback is challenging. In this experiment, we mine submissions to programming assignments to collect examples of edits applied by students to fix their code. We then use these examples to synthesize transformations, and we try using the learned transformations to fix any new students' submissions with similar types of faults. We say that a submission is "fixed" if it passes the set of tests provided by course instructors. Synthesized fixes can then be used for grading or turned into hints to help students locate faults and correct misconceptions. In our evaluation conducted on 4 programming tasks performed by 720 students, REFAZER synthesizes transformations that fix incorrect submissions for 87% of the students.

Our second experiment is motivated by the fact that certain repetitive tasks occurring during software evolution, such as complex forms of code refactoring, are beyond the capabilities of current IDEs and have to be performed manually [47, 62]. In this experiment, we use repetitive code edits applied by programmers to the same project to synthesize a program transformation that can be applied to other locations in the code. We performed a study on three popular open-source C# projects (Roslyn [128], Entity Framework [27], and NuGet [102]) to identify and characterize repetitive code transformations. In our evaluation conducted on 56 scenarios of repetitive edits, REFAZER learns the intended program transformation in 84% of the cases using 2.9 examples on average. The learned transformations are applied to as many as 60 program locations. Moreover, in 16 cases REFAZER synthesized transformations on more program locations than the ones present in our dataset, thus suggesting potentially missed locations to the programmers.

So far, we describe the REFAZER evaluation. Now, we talk about REVISAR evaluation.

1.3.2 REVISAR

We evaluated the effectiveness of REVISAR and used it to mine transformations from nine popular Java GitHub projects, each one containing between 1,062 and 13,790 revisions. REVISAR was able to learn 28,842 transformations, which covered 158,906 out of 240,612 edits. Several patterns were learned from edits appearing in different revisions and different projects. We manually analyzed the results and saw that, most of the times, REVISAR correctly learned the patterns intended by the programmers. In some cases REVISAR learned overly specific patterns—e.g., it did not abstract a variable.

We then acted as developers of code analyzers and inspected the transformations learned by REVISAR to find the ones that describe quick fixes. First, we focused on the following research question: **(RQ1)** Can REVISAR be used to discover common quick fixes? Of the patterns learned by REVISAR, we manually inspected the 381 ones that occurred across most projects. After merging similar patterns and removing patterns that did not represent quick fixes (e.g., renaming `object` to `obj`), we derived a catalog of 32 quick fixes—i.e., edits that, based on the Java literature, improved the quality of the code in terms of correctness/performance/readability.

Since the creation of our catalog involved manual selection, we wanted to know the opinion of other programmers about the quick fixes discovered by REVISAR. Hence we asked **(RQ2)** Are programmers interested in applying quick fixes discovered by REVISAR? To answer this question, we performed two studies. First, we conducted an online survey with 164 (8.2% response rate on 2,000 survey request) randomly selected programmers from 124 different projects, including projects from Google, Facebook, and Apache Foundation. In this survey, we showed the programmers 10 out of the 32 quick fixes in our catalog. Our selection included both quick fixes missed and included in code analyzer tools. In this way, we could assess whether programmers that use code analyzers agreed with the usefulness of the quick fixes applied by the analyzer. Overall, programmers supported 9 (90%) quick fixes in our catalog. To better answer **RQ2**, we submitted 20 pull requests to 12 distinct projects, performing the quick fixes that were approved in the survey. In 10/20 (50%) cases the programmers found the quick fix to be useful and accepted the pull request. In 7 of the 10 remaining cases, programmers supported the quick fixes, but did not merge them because they requested further discussion or the quick fixes were not performed on critical locations

in their code—i.e., the programmers already had checks in place to avoid the effect of the bad code patterns. The remaining 3 pull requests were ignored. Remarkably, following one of our pull requests, *the developers of the code analyzer PMD added one of our rules to their library.*

1.4 Contributions

This thesis makes the following contributions:

- REFAZER, a novel technique that leverages state-of-the-art IP methodology to efficiently solve the problem of synthesizing transformations from examples (Chapter 4);
- An evaluation of REFAZER in the context of learning fixes for students' submissions to introductory programming assignments (Section 4.2.1);
- An evaluation of REFAZER in the context of learning transformations to apply repetitive edits in open-source industrial C# code (Section 4.3.1);
- REVISAR, a fully automatic novel and unsupervised technique for mining code repositories to identify repetitive Java transformations (Chapter 5);
- A comprehensive evaluation of REVISAR on nine large GitHub code repositories where REVISAR learned 920 new edits across projects, many of which were confirmed to be of good quality (Section 5.4);
- An evaluation of ten of our transformations with programmers from open source repositories (Section 5.5.2).

1.5 Organization

We organize this thesis as follows: In Chapter 2, we present the background to understand this thesis. We explain the concept of program synthesis and inductive programming synthesis. We also show a state-of-the-art framework in domain-specific program synthesis. Additionally, we describe the concept of program transformation and state-of-the-art algorithms to compute

edit operations in trees. In Chapter 3, we discuss a state-of-the-art of techniques to do transformations in software systems. In Chapter 4, we show REFAZER, our technique to learn syntactic program transformations based on examples. We also discuss the evaluation of this technique. In Chapter 5, we explain REVISAR, our technique for learning quick fixes from repositories along with its evaluation. In Chapter 6, we present the conclusion of this thesis. Finally, in Appendix A, we present the search string to select the works in our state-of-the-art. In Appendix B, we describe the catalog of transformations identified by REVISAR. In Appendix C, we show the survey used to evaluate REVISAR.

Chapter 2

Background

In this chapter, we present the background to understand this work. In Section 2.1, we present program synthesis, inductive programming, and domain-specific inductive synthesis. We also present a meta-framework for domain-specific synthesis by examples. In Section 2.2, we overview program transformations and present state-of-the-art techniques in program transformation. Finally, in Section 3, we present the state-of-the-art related to program transformation in software engineering.

2.1 Program synthesis

Program synthesis [41] is the process of learning programs specified in some form (e.g., logical specification and input-output examples). It can be seen as a search problem to find the desired program given a specification. In this search problem, specification is a mechanism in which users define their intent — i.e., define the task to be automated. The search space bounds possible generated programs. Here, a challenge is how to define a search space that is expressive enough to do intended tasks but restricted enough for efficient synthesis [44]. Moreover, search techniques explore search space to find the desired program.

Program synthesis can roughly be classified in deductive or inductive synthesis. Deductive synthesis uses theorem-proving and the intent is given by declarative and logical specification [76, 121]. The advantage of deductive synthesis is performance since it only relies on valid programs (i.e., programs following the specification) and does not backpropagate with invalid programs. Nevertheless, it requires specialized knowledge to write specifications [121].

Whereas, inductive synthesis, more known as inductive programming (IP), uses input-output examples to learn programs. The advantage of inductive programming is specification since examples are, usually, easy to provide. Nevertheless, examples are an incomplete specification. Thus, the number of programs consistent with the examples could be huge. In this work, we use examples to synthesize program transformations. Thus, we focus on inductive programming and discuss areas related to this kind of synthesis.

2.1.1 Inductive programming

Inductive programming, also known as Programming-by-Example (PBE), denotes a sub-area of program synthesis in which the specification is given by examples [43]. It has been an active research area in the AI and HCI communities for over a decade [71]. IP techniques have recently been developed for various domains including interactive synthesis of parsers [68], imperative data structure manipulations [136], and network policies [154]. Furthermore, IP has been extensively applied to the domain of data wrangling: a process that converts data between formats (e.g., FlashExtract [66] and FlashFill).

An advantage of inductive programming is that it allows users with limited or no background in programming to do complex tasks. For instance, 99% [43] of the population lack programming background but do tedious, time-consuming, and repetitive tasks (e.g., format spreadsheet cells). To do such tasks, users often use specialized forums. They post questions (usually with examples) related to the problem, and an expert provides a program to do the target task. IP can play a role of a specialist, helping users in their tasks. An example of a technique that has been helping users to do tedious and time-consuming tasks in spreadsheets using inductive programming is FlashFill [42], which is an IP technique for doing string transformations using input-output examples. For instance, Figure 2.1 illustrates a task that normalizes names in an input column using examples. The user only needs to give a few examples in the output column; FlashFill induces a program transformation to fill the other cells of the output column automatically.

Nevertheless, examples are an incomplete specification. Thus, the number of correct programs in the examples could be huge. Here, a challenge is how to efficiently store and search for the program that is correct to seen inputs (i.e., examples) and also has a higher probability to be correct on unseen inputs.

	A	B
1	Input	Output
2	Ana Trujillo	Trujillo, A.
3	Antonio Moreno	Moreno, A.
4	Thomas Hardy	Hardy, T.
5	Christina Berglund	Berglund, C.

Figure 2.1: Normalizing names by examples using FlashFill.

Domain-specific inductive synthesis

Domain-specific inductive synthesis is a sub-area of inductive programming that learns programs based on a domain-specific language (DSL). In contrast to a general-purpose language (GPL) (e.g., Java), a DSL focuses on a specific application domain [86]. Thus, it is optimized to that domain. Nevertheless, designing a domain-specific language synthesizer is a non-trivial task. It often requires four tasks [44]:

- **Problem definition:** identify the domain of tasks to automate and collect real scenarios of these tasks;
- **Domain-specific language:** propose a DSL expressive enough to do real scenarios of repetitive tasks but restricted enough for efficient learning from examples;
- **Synthesis:** design algorithms to synthesize DSL programs from examples. Synthesis algorithm may return programs that match examples;
- **Ranking:** design ranking algorithm to rank programs returned by the synthesizer.

For instance, consider the Java language; although developers could think in many ways to write a program that does the task of normalizing names (Figure 2.1). The space of possible programs to explore in Java is prohibitive. Thus, Java is expressive for that task, but a synthesizer is unable to learn a program in a feasible time.

Next, we describe a meta-framework that allows the development of applications in domain-specific inductive synthesis.

2.1.2 PROSE

Microsoft Program Synthesis using Examples (PROSE) denotes a meta-framework for domain-specific inductive synthesis. Problems in domain-specific inductive synthesis often share

properties (e.g., algorithms, operators, and rules). Nevertheless, synthesis is time-consuming: besides designers must know the domain, the DSL is coupled with implementation. Thus, small changes in the DSL may require non-trivial changes in implementation. PROSE allows synthesis by providing infrastructure to learn, rank, and run programs.

In PROSE, an application designer defines a DSL for desired tasks. The synthesis problem is given by a *spec* φ , which contains a set of program inputs and constraints on desired program’s outputs on these inputs (e.g., examples of these outputs). PROSE synthesizes a set of programs in the DSL that are consistent with φ , combining *deduction*, *search*, and *ranking*:

- Deduction is a top-down walk over the DSL grammar, which iteratively *backpropagates* the spec φ on the desired program to necessary specs on the subexpressions of this program. In other words, it reduces the synthesis problem to smaller synthesis subproblems using a divide-and-conquer dynamic programming algorithm over the desired program’s structure.
- Search is an enumerative algorithm, which iteratively constructs candidate subexpressions in the grammar and verifies them for compliance with the spec φ [2].
- Ranking is a process of picking the most robust program from the synthesized set of programs that are consistent with φ . Because examples are highly ambiguous, such a set may contain up to 10^{20} programs [121], and quickly eliminating undesirable candidates is paramount for a user-friendly experience.

Besides the DSL, synthesis designers define backpropagation procedures (*witness functions*), semantic algorithms (*semantic functions*), and ranking strategies (*ranking functions*) for each DSL operator.

- *Witness functions*. In PROSE, a *witness function* ω_F is a backpropagation procedure, which, given a spec φ on a desired program on kind $F(e)$, deduces a necessary (or even sufficient) spec $\varphi_e = \omega_F(\varphi)$ on its subexpression e .¹ Witness functions enable efficient top-down synthesis algorithms of PROSE.

¹Another view on witness functions ω_F is that they simply implement *inverse semantics* of F , or a generalization of inverse semantics w.r.t. some *constraints* on the output of F instead of just its *value*.

- *Ranking functions.* Since example-based specifications are incomplete, the synthesized abstract transformation may not do the desired transformation on other input programs. We specify *ranking functions* that rank a transformation based on its robustness (i.e., likelihood of it being correct in general).

Although PROSE supports some operators (e.g., *Filter* and *Map*), synthesis designers must define witness, semantics, ranking functions for most of the DSL operators. PROSE divides synthesis problem in smaller sub-problem over a version space algebra (VSA).

DSL syntax

A DSL defines the syntax of possible programs. Listing 2.2 shows the DSL for FlashFill `SubStr` operator. A DSL is written in a context-free grammar (CFG) and is composed by symbols and operators. Each DSL line defines a rule that associates a symbol on the left-hand side (LHS) to operators in the right-hand side (RHS). For instance, the `expr` symbol is defined in terms of operator `SubStr`. In this DSL, `SubStr` extracts a substring between position expressions p_1 and p_2 . Position p is either an absolute or a relative position to the k^{th} occurrence of a position sequence defined by the regex pair (r_1, r_2) . Position sequence (r_1, r_2) is the set of positions p in a string such that the left side of p matches with r_1 (a prefix), and the right side of p matches with r_2 (a suffix)— see Section 2.1.2 for an example.

```

expr      ::= SubStr(x, pos, pos)
pos       ::= AbsPos(x, k) | RelPos(x, r1, r2, k)
k         ::= int
r         ::= regex

```

Figure 2.2: Grammar for `SubStr` operator.

Semantic functions

The semantic functions define behavior for DSL operator. For example, `SubStr` semantic receives input string x and two positions pos_1 and pos_2 and returns the substring of x that starts at the pos_1 and ends at pos_2 (Algorithm 1). `AbsPos` semantic functions return the

string index k if $k \geq 0$ or $(length(x) + k + 1)$, otherwise (Algorithm 2). Algorithm 3 illustrates semantic function for `RelPos` operator. It returns the k^{th} index in x that has r_1 from the left and r_2 from the right.

```
1: result += x.SubStr(pos1, pos2)
2: return result
```

Algorithm 1: Semantic function `SubStr(x, pos1, pos2)`.

```
1: result += k >= 0 ? k : (length(x) + k + 1)
2: return result
```

Algorithm 2: Semantic function for operator `AbsPos(x, k)`.

```
1: matches += all indexes in x that matches r1 from left and r2 from right
2: result += matches[k]
3: return result
```

Algorithm 3: Semantic function `RelPos(x, r1, r2, k)`.

Version space algebra

In this section, we present the structure used by PROSE to efficiently store programs, a version space algebra [65] (VSA). This structure consists of version spaces (VS), which represent a set of subprograms/subexpressions of a program that satisfies the examples from the most specific subprogram/subexpression to the most general one. To form a new VS, algebra operators such as join and union combine the subprograms in a VS. Moreover, a VSA can be seen in a tree-like structure where nodes represent subprograms or a program if the node is the root of the tree. Each child node represents the VS for a subexpression of an operator. Since algebra operators (e.g., Cartesian product) can be applied to children of a VS to obtain a new VS, a VSA store an exponential number of programs. Nevertheless, the space of possible programs is polynomial. Union and Join operators are defined as follows:

Union operator Let v_1 and v_2 be version spaces with equal domain and range $d(v_1) = d(v_2)$ and $r(v_1) = r(v_2)$. Union operator returns $v_1 \cup v_2$.

Join operator Let v_1 and v_2 be version spaces. Let $E_1 = \{e_i\}_{i=1}^{|E_1|}$ be training examples (i, o) to learn v_1 and let $E_2 = \{e_i\}_{i=1}^{|E_2|}$ be training examples (i, o) to learn v_2 . Join operator returns Cartesian product $v_1 \bowtie v_2$, such that v_1 satisfies E_1 and v_2 satisfies E_2 .

To illustrate a VSA, consider the following problem: given people names, extract the last name, illustrated in Figure 2.3. In Figure 2.4, we present the VSA that represents programs that are consistent with the two first cells in Figure 2.3 as examples. These two examples generate twelve programs, which are represented succinctly in the VSA. `SubStr` version space joins first and second position version space. This version space is composed by the Cartesian product (i.e., join operation) of the first and second parameter of the `SubStr` operator. The first position returns four programs and the second position returns three. Thus, this version space generated $4 \times 3 = 12$ programs. Version space for each position unites `AbsPos` and `RelPos` version space. The VSA for the first position (i.e., `AbsPos` | `RelPos`) generated four programs. This VSA space is composed by the union of the programs for the possible `pos` operators (i.e., `AbsPos` on the left and `RelPos` on the right). The `AbsPos` operator returns an empty set because there is no k such that k matches the first position to extract the last name from the first cell and matches the first position to extract the last name from the second cell. Whereas, `RelPos` identifies four possible patterns to match the first position both on the first cell and the second cell. For instance, the first pattern in this set matches with a position that contains space (s^+) on the left and word (w^+) on the right. The VSA to extract the second position undergoes the same process. This version space unites `AbsPos` and `RelPos`. The `AbsPos` version space contains a single element that contains a $k = -1$ to match the end of the string. Additionally, `RelPos` contains two patterns to match the end of the string. Given the first element in this VSA, we have program `SubStr(x, RelPos(x, s+, w+, 1), AbsPos(x, -1))`, which extracts the substring between the first occurrence of pattern `[s+, w+]` and string end.

	A	B
1	Input	Output
2	Ana Trujillo	Trujillo
3	Antonio Moreno	Moreno
4	Thomas Hardy	Hardy
5	Christina Berglund	Berglund

Figure 2.3: Data for the extract last name problem.

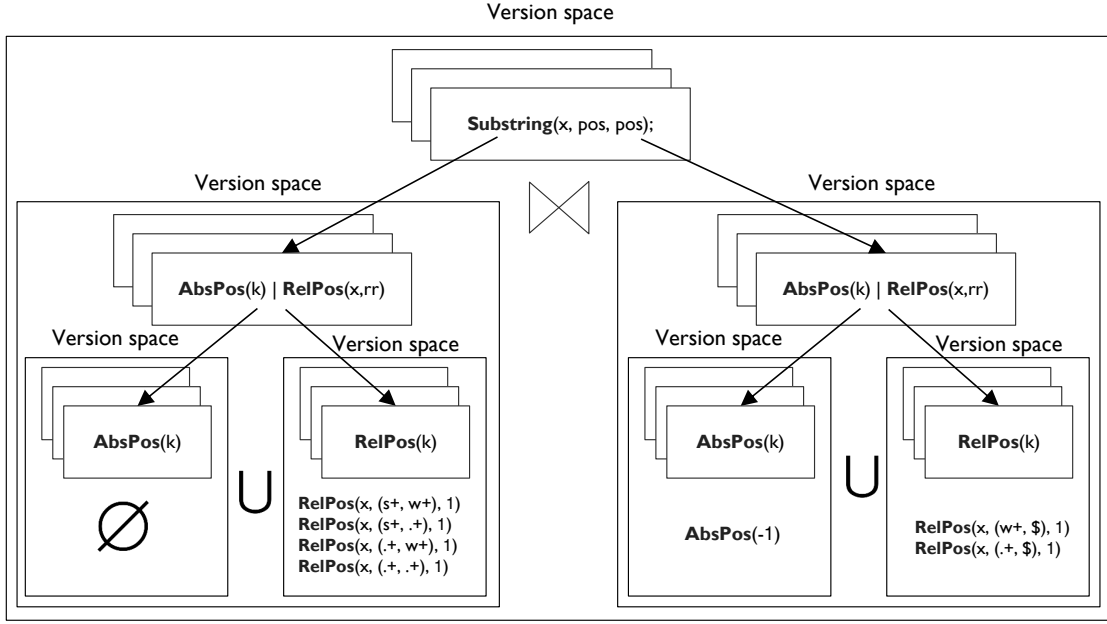


Figure 2.4: Version space algebra for the extract last name problem.

Witness functions

A witness function ω_F is a backpropagation procedure, which, given a spec φ on a desired program on kind $F(e)$, deduces a necessary (or even sufficient) spec $\varphi_e = \omega_F(\varphi)$ on its subexpression e . For example, the `SubStr` operator (Figure 2.2) requires two ω_{FS} , one for each parameter/subexpression of the `SubStr` operator. Each ω_F deduces the specification for `SubStr` to the specification to learn its correspondent parameter/subexpression. For instance, Algorithm 4 shows ω_F for the first position. This ω_F requires an example-based spec φ for the `SubStr` operator of the form (P_i, P_o) where P_i denotes the input string (e.g., the first cell in Figure 2.3 column A) and P_o denotes the output string (e.g., the first cell in Fig 2.3 column B). The goal of this ω_F is to deduce the specification for the first position expression given the specification for learning a `SubStr` operator. The first position expression requires a position. Thus, the ω_F for the first position generates input-output examples of the form (P_i, P_o) where P_i is the input string and P_o is a position, which is the desired specification for the first subexpression of the `SubStr` operator. To this end, ω_F for first position of `SubStr` operator evaluates all occurrences of P_o on P_i (line 4) and returns start index for each occurrence (line 6). Similarly, Algorithm 5 shows ω_F for second position. Instead of returning start index, it returns final occurrence index.

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   occurrences := all matches of  $P_o$  in  $P_i$ 
5:   for all occurrence  $\in$  occurrences do
6:     examples +=  $P_i.$ IndexOf(occurrence)
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 4: Witness function to first parameter of operator $\text{SubStr}(x, \text{pos}, \text{pos})$.

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   occurrences := all matches of  $P_o$  in  $P_i$ 
5:   for all occurrence  $\in$  occurrences do
6:     examples +=  $P_i.$ IndexOf(occurrence) + length(occurrence)
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 5: Witness function to second parameter of operator $\text{SubStr}(x, \text{pos}, \text{pos})$.

Furthermore, Algorithm 6 shows ω_F for AbsPos operator. The goal of this ω_F is to deduce the specification for the k parameter. It requires an example-based spec φ of the form (P_i, P_o) where P_i denotes the input string and P_o is a list of positions inside input, returned by previous ω_F (Algorithm 4). This ω_F evaluates to the k index inside input from left to right and vice-versa (line 6).

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   positions :=  $P_o$ 
5:   for all pos  $\in$  positions do
6:     examples +=  $\text{pos} \cup \text{pos} - \text{length}(P_i) - 1$ 
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 6: Witness function to first parameter of operator $\text{AbsPos}(x, k)$.

Algorithm 7 shows ω_F for first parameter of RelPos operator. The goal of this algorithm is to deduce the specification for left side regex r . Similar to the previous ω_F , it requires an example-based spec φ of the form (P_i, P_o) where P_i denotes the input string and P_o is a list of

positions inside input. This ω_F outputs a regex list that matches P_i from the left of position pos (line 6). Similarly, Algorithm 8 shows ω_F for second parameter. Instead of returning a regex list that matches the left of pos , it returns a regex list that matches right of pos . Finally, Algorithm 9 shows ω_F from third parameter of $RelPos$ operator.

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   positions :=  $P_o$ 
5:   for all  $pos \in positions$  do
6:     examples += regex list that match left of  $pos$  on  $P_i$ 
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 7: Witness function to first parameter of operator $RelPos(x, r, r, k)$.

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   positions :=  $P_o$ 
5:   for all  $pos \in positions$  do
6:     examples += regex list that match right of  $pos$  on  $P_i$ 
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 8: Witness function to second parameter of operator $RelPos(x, r, r, k)$.

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all  $(P_i, P_o)$  in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   positions :=  $P_o$ 
5:   for all  $pos \in positions$  do
6:     examples += index of  $pos$  in positions
7:   result[ $P_i$ ] += examples
8: return result

```

Algorithm 9: Witness function to third parameter of operator $RelPos(x, r, r, k)$.

Ranking functions

Since example-based specifications are incomplete, the number of transformations learned could be huge ($> 10^{20}$). Although all the learned transformations edit correctly the provided

input-output examples, some of these transformations may incorrectly edit instances lacked in the input-output example specification (i.e., the program does not follow the user’s intent). Thus, we need to favor the transformations that are robust (i.e., likelihood of it being correct in general). The goal of the *ranking functions* is to select robust transformations from the program space. To illustrate this concept, we present ranking functions for the FlashFill DSL in Algorithm 10, Algorithm 11, and Algorithm 12. These ranking functions compute their score recursively based on the score for ranking functions for their subexpressions. For simplicity, ranking function for `ABSPos` returns 0 and ranking function for `RELPos` returns 1. Furthermore, the `SubStr` operator returns the sum of the score of its subexpressions. The intuition here is that programs that compute the position relative to a pattern have a higher probability of getting the correct position for unseen input than programs that just return a position. Although we show ranking functions that compute the score recursively based on the score of the subexpressions of an operator, ranking functions also could be computed based on the properties of the generated program. To design of ranking functions, we could recur to experts or compute these values empirically.

```
1: result += scorepos1 + scorepos2)
2: return result
```

Algorithm 10: Ranking function for `SubStr` (*score_x*, *scorepos*₁, *scorepos*₂).

```
1: result := 0
2: return result
```

Algorithm 11: Ranking function for `ABSPos` (*score_x*, *scorepos_k*).

```
1: result := 1
2: return result
```

Algorithm 12: Ranking function for `RELPos` (*score_x*, *scorer*₁, *scorer*₂, *score_k*).

2.2 Program transformation

In this section, we overview program transformation. In addition, we present four state-of-the-art algorithms for representing tree edits. In Section 2.2.1, we present Zhang and Sasha

algorithm. In Section 2.2.2, we present Chawathe et al. algorithm. In Section 2.2.3, we present ChangeDistiller. Finally, in Section 2.2.4, we present GumTree.

A transformation is a structure that abstracts edits in the source code, allowing it to be applied in other contexts during the software development. For example, modern IDEs, include in their refactoring suites a set of transformations to help developers to edit the source code, such as rename and extract method refactorings. Usually, a transformation could be described by the edit operations that are done in a tree to get another tree: a process known as computing the edit script. An edit script is a mechanism that computes the difference between versions of the same tree. Since an AST is a tree that represents source code, an edit script can be seen as the problem to compute the difference between two ASTs. The edit script is composed by a list of edit operations. In general, the edit script problem aims to minimize edit operations that when applied to source AST (T_1) produce target AST (T_2). Basic edit operations are the following:

- **Insert(x, p, k)**: Insert a leaf node x on parent p at k^{th} position;
- **Delete(x)**: Delete a leaf node x from source tree;
- **Update(x, w)**: Update a leaf node x to leaf node w ;
- **Move(x, p, x)**: Move subtree x to be k^{th} child of parent p .

In general, the edit script problem could be divided into subproblems of mapping nodes from T_1 and T_2 and finding edit operations that produces T_2 when applied to T_1 [13]. State-of-the-art techniques to compute edit scripts include Zhang and Shasha (Section 2.2.1), Chawathe et al. [13], ChangeDistiller [33], and GumTree [31].

2.2.1 Zhang and Shasha

Zhang and Shasha [155] is an algorithm that computes minimum edit scripts of ordered trees. An ordered tree is a tree that preserves the left-to-right order of sibling nodes. Since an AST preserves the left-to-right order of sibling nodes, this algorithm may be used to compute the minimum edit script for two ASTs. Zhang and Shasha extend the edit script problem of string domain to the edit script problem for tree domain. Edit script problem for string domain is

limited to tree domain since strings do not preserve the ancestor-descendant relationship, but trees do. To compute edit script for ASTs, this algorithm compares ordered forest from AST nodes. An ordered forest is a structure that maintains root nodes from two node positions i and j when AST is traversed in some order — e.g., post-ordering adopted in Zhang and Shasha. For instance, Figure 2.5 (right) shows the forest computed from position 1 to 7 from left AST. Along the algorithm, Zhang and Shasha maintain a list l of the leaf most descendant

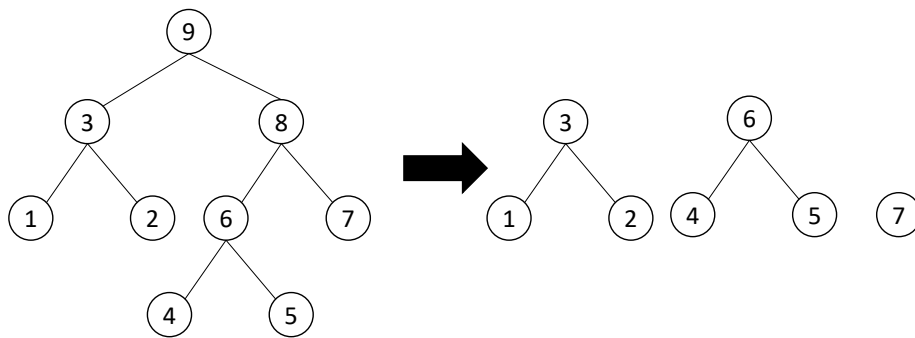


Figure 2.5: Example of a forest for an AST.

of subtree root at node x and a list *keyroots* of key roots. For instance, Figure 2.6 illustrates the leaf most descendant from nodes 1, 3, and 9. Moreover, a node x is a key root if there is no k such that $k > x$ and $l(x) = l(k)$, formally: $keyroot(x) = \{\nexists k \in k > x \mid l(x) = l(k)\}$. Figure 2.7 shows key roots from before/after ASTs.

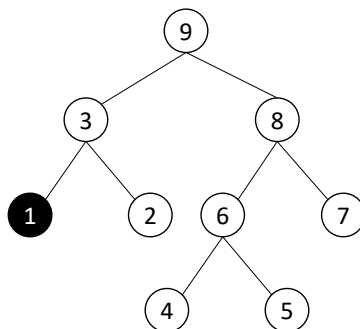


Figure 2.6: Example of leaf most descendants. In this example, the node number 1 is leaf most descendants of nodes number 1, 3, and 9.

The algorithm goal is to minimize edit operations to convert AST T_1 to T_2 . In this process, it associates costs for edit operations (i.e., insert, delete, and update) and uses

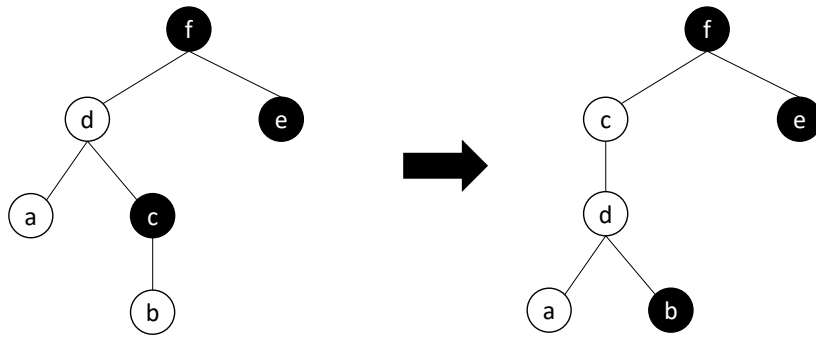


Figure 2.7: Trees to compute edit script, showing key roots.

dynamic programming to compute edit script. Algorithm 13 shows the main algorithm. Line 1 processes list l and $keyroots$. At line 4, it calls Algorithm 14 to build $tdist$ table. In Algorithm 14, lines 1 – 5 compute $fdist$ three base cases. The first base case occurs when T_1 and T_2 are empty. Thus, no operations are learned. The second base case occurs when T_1 contains nodes and T_2 is empty. In this case, the minimum edit script contains only delete operations. The last base case occurs when T_1 is empty and T_2 contains nodes. Here, edit script contains only insert operations. At line 9, if i_1 is an AST and j_1 is an AST, $fdist$ table is updated with the cheapest operation (i.e., delete, insert or update, respectively). Otherwise, the update operation bases on $tdist$ table and $fdist$ will be the cheapest of these three operations. Given $tdist$ table, edit script is computed using Levenshtein [16] algorithm.

Input: Trees T_1 and T_2

Output: Tree dist table $tdist$

- 1: Preprocessing {compute l and $keyroots$ }
- 2: **for all** $i \in keyroots(T_1)$ **do**
- 3: **for all** $j \in keyroots(T_2)$ **do**
- 4: $treedist(i, j)$

Algorithm 13: Zhang and Shasha algorithm.

Input: $i \in \text{keyroots}(T_1)$ and $j \in \text{keyroots}(T_2)$

Output: Tree distance table $tdist$

```

1:  $fdist(\emptyset, \emptyset) = 0$  {preprocess forest dist  $fdist$ .}
2: for all  $i_1 \in \text{range}(l(i)..i)$  do
3:    $fdist(T_1[l(i)..i_1], \emptyset) = fdist(T_1[l(i)..i_1 - 1], \emptyset) + \gamma(T_1[i_1] \rightarrow \Lambda)$ 
4: for all  $j_1 \in \text{range}(l(j)..j)$  do
5:    $fdist(\emptyset, T_2[l(j)..j_1]) = fdist(\emptyset, T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])$ 
6: for all  $i_1 \in \text{range}(l(i)..i)$  do
7:   for all  $j_1 \in \text{range}(l(j)..j)$  do
8:     if  $i_1$  is a tree and  $j_1$  is a tree then
9:        $fdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$ 
10:          $fdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \rightarrow \Lambda)$ 
11:          $fdist(T_1[l(i)..i_1], T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])$ 
12:          $fdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1 - 1]) + \gamma(T_1[i_1] \rightarrow T_2[j_1])\}$ 
13:        $tdist[i_1, j_1] = fdist(T_1[l(i)..i_1], T_2[l(j)..j_1])$ 
14:     else
15:        $fdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$ 
16:          $fdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \rightarrow \Lambda)$ 
17:          $fdist(T_1[l(i)..i_1], T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])$ 
18:          $fdist(T_1[l(i)..l(i_1) - 1], T_2[l(j)..l(j_1) - 1]) + tdist(i_1, j_1)\}$ 

```

Algorithm 14: Algorithm to compute $treedist$.

2.2.2 Chawathe et al.

Chawathe et al. [13] is a traditional algorithm for computing edit script. It divides this problem on the subproblem of finding a matching and the subproblem of computing a minimum edit script. The matching problem bases on the leaf nodes similarity and non-leaf nodes similarity:

Leaf matching : Given leaf nodes $x \in T_1$ and $y \in T_2$. Nodes (x, y) match if their labels match $l(x) = l(y)$ and their similarity exceeds a threshold λ , formally $sim(x, y) > \lambda$, satisfying $0 \leq \lambda \leq 1$.

Non-leaf matching : Given non-leaf nodes $x \in T_1$ and $y \in T_2$. Nodes $(x, y) \in M$ match if their labels match $l(x) = l(y)$ and their similarity exceeds a threshold λ , formally $\frac{\text{common}(x, y)}{\max(|x|, |y|)} > \lambda$, satisfying $0.5 \leq \lambda \leq 1$. $\text{common}(x, y)$ denotes x and y common nodes.

Algorithm 16 shows matching algorithm. Line 1 is a preprocessing step that sets mapping M to empty. The algorithm matches nodes based on their labels. For each label l , line 3 chains T_1 and T_2 (i.e., traverse nodes in order). Lines 4 – 5 use longest common subsequence algorithm (LCS) to match nodes from T_1 and T_2 based on label. The matched nodes accordingly with LCS are added to M . The aim is to compute the minimum edit script. Thus, the

LCS provides a configuration that maximizes the matching between T_1 and T_2 . The more similar are T_1 and T_2 , the less the number of edit operations. Lines 6 – 8 process each node that remains unmatched based on leaf and non-leaf similarity. This matching is based on the first matching of a node x in T_1 with a node y in T_2 . For a good matching, Chawathe et al. require that, for any leaf $x \in T_1$ and $y \in T_2$, x match at most a node y and vice-versa. When a node match multiple nodes, the matching may be made with the incorrect node and the edit script may contain more edit operations than the minimum to convert source AST T_1 to target T_2 . In Section 2.2.3, we discuss a case where this assumption fails.

Input: Trees T_1 and T_2

Output: Matching M

```

1:  $M = \emptyset$ 
2: for all  $l \in labels$  do
3:    $S_1 = chain(T_1, l)$ ,  $S_2 = chain(T_2, l)$ 
4:    $lcs = LCS(S_1, S_2, equal)$ 
5:   for all  $(x, y) \in lcs$  do add  $(x, y)$  to  $M$ 
6:   for all  $x \notin M$  and  $x \in S_1$  do
7:     if  $y \notin M$  and  $y \in S_2$  and  $equal(x, y)$  then
8:       add  $(x, y)$  to  $M$ 

```

Algorithm 15: Matching Chawathe et al. algorithm.

Edit script algorithm (Algorithm 15) uses M to compute edit operations. Line 1 is a preprocessing step that sets edits E to empty. Lines 2 – 14 denote the first phase of the algorithm that computes insert, update, and move edit operations. In this first phase, line 2 traverses T_2 in breadth first search. Line 6 adds an inserted operation if x is unmatched. The rationality for this operation is that if a node is lacked in T_2 , it must be inserted in T_1 to produce T_2 . If the value of two matched nodes differs, line 10 adds an update operation. The reason for this operation is that if two nodes match together, but differ in their values, an update operation must be added to T_1 to produce the corresponding value of the node in T_2 . Line 13 adds a move operation when node parents do not match. The rationality for this operation is that if the parent of x and the parent of y do not match, but x and y do match, the node x must be moved to some location in T_1 such that their parent matches. The second phase of the algorithm computes delete operations. For each unmatched node $x \in T_1$, lines 15 – 17 add a delete operation. The rationality for this operation is that if a node x is present on T_1 , but not present on T_2 it must be removed from T_1 to produce T_2 . At the end, T_1 and T_2 are isomorphic since each operation is applied to T_1 besides being added to E .

Input: Trees T_1 and T_2 and M

Output: Edit Script E

```

1:  $M' = M, E = \emptyset$ 
2: for all  $x \in T_2$  breadth first search do
3:    $y = p(x)$  and  $z$  is the partner of  $y$  in  $M'$ 
4:   if  $x$  has no partner in  $M'$  then
5:      $k = \text{FindPos}(x)$  and create a node  $w$  based on  $x$ 
6:     add  $\text{Insert}(w, z, k)$  to  $E$ , add  $(w, x)$  to  $M'$ , and apply  $\text{Insert}(w, z, k)$  to  $T_1$ 
7:   else
8:      $w =$  the partner of  $x$  in  $M'$  and  $v =$  the partner of  $p(w)$  in  $M'$ 
9:     if  $v(x)$  differs from  $v(w)$  then
10:      add  $\text{Update}(w, v(x))$  to  $E$  and apply  $\text{Update}(w, v(x))$  to  $T_1$ 
11:     if  $(y, v) \notin M$  then
12:        $k = \text{FindPos}(x)$  and  $z$  is the partner of  $y$  in  $M'$ 
13:       add  $\text{Move}(w, z, k)$  to  $E$  and apply  $\text{Move}(w, z, k)$  to  $T_1$ 
14:     Align children  $(w, x)$ 
15: for all  $w \in T_2$  post-order traversal do
16:   if  $w$  has no partner in  $M'$  then
17:     add  $\text{Delete}(w)$  to  $E$  and apply  $\text{Delete}(w)$  to  $T_1$ 

```

Algorithm 16: Edit Script Chawathe et al. algorithm.

2.2.3 ChangeDistiller

ChangeDistiller [13] is an algorithm to extract fine-grained changes for ASTs. It identifies fine-grained source code changes based on a taxonomy of code changes [32]. This taxonomy contains 35 kinds of transformations for object-oriented programming. This algorithm improves Chawathe et al. algorithm for computing edit script for ASTs. Since Chawathe et al. is an algorithm applicable for all kinds of trees, it may present suboptimal results when applied to ASTs. One reason for this phenomenon is that Chawathe et al. quality depends on similarity functions and a threshold that are difficult to adjust to adequate for all kind of trees. Thus, in some kinds of trees Chawathe et al. often do not compute the minimum edit script. For instance, in small trees, the mismatch on leaves may be propagated to inner nodes, leading to a mismatch in high-level nodes. Figure 2.8 shows a good matching that Chawathe et al. is unable to compute using a threshold of 0.6. To compute the mapping, Chawathe et al. traverses in bottom-up from left to right (i.e., in order traversal). Here, `foo.GetHuga()` matches in tree T_2 , but `foo.doNothing()` does not. The dissimilarity from the bottom nodes is propagated to the inner nodes. Using the non-leaf similarity function, the `Then` does not match since $\frac{\text{common}(\text{Then}, \text{Then})}{\max(2, 2)} = \frac{1}{2} = 0.5$. The same occurs with the `If` statement. Another scenario where Chawathe et al. produces suboptimal results is when assumption 1 fails (i.e.,

one node matches multiple nodes.). Figure 2.9 shows an example where a node matches multiple nodes. Nodes are matched from the left to the right, thus *Node 1* and *Node 3* matches, and *Node 2* is discarded.

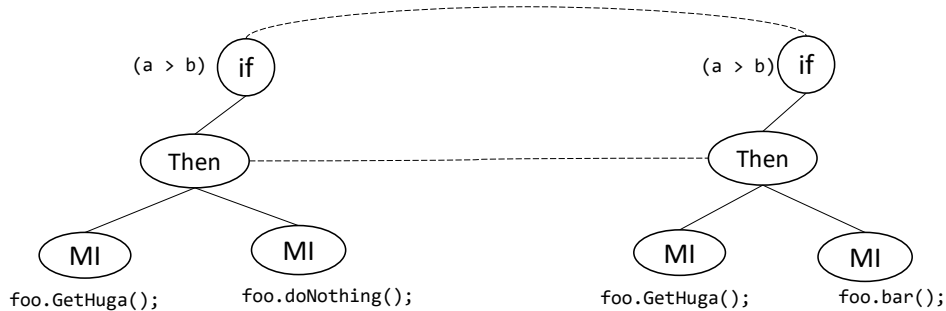


Figure 2.8: An example of a match for a small tree.

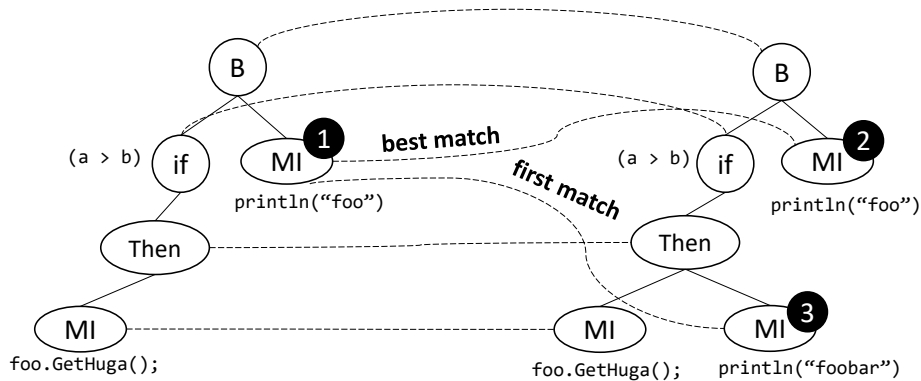


Figure 2.9: An example where a node x in T_1 matches multiples nodes in T_2 .

To improve Chawathe et al., ChangeDistiller extends Chawathe et al. in four aspects. First, it customizes leaf node matching. Additionally, it customizes inner node matching. It also introduces the concept of best match. Finally, it uses a dynamic threshold based on the number of node descendants.

To customize leaf node match, ChangeDistiller compares strings based on bi-gram similarity (Eq. 2.1) instead of string similarity based on Levenshtein algorithm (Eq. 2.2). Bi-grams uses a sliding window over a string for each pair of characters (e.g., “ve”, “er”, “rt”, “ti”,

“ic”, “ca”, “al” is the bi-gram of `vertical`). The key idea to use bi-gram is that developers often change word orders (e.g., `verticalDrawAction` to `DrawVerticalAction`). For this kind of change, Levenshtein distance $D(x, y)$ is not suitable since it is based on LCS. The more different the strings, the more operations to convert strings. The LCS from `verticalDrawAction` to `DrawVerticalAction` is `verticalAction`. Thus, four deletes are needed to remove `draw` and four inserts are needed to write `draw`. Using $f = 0.6$, `verticalDrawAction` and `DrawVerticalAction` do not match using Levenshtein similarity.

$$sim_{ngram} = \frac{2 * |ngram(x) \cap ngram(y)|}{|ngram(x) \cup ngram(y)|} \quad (2.1)$$

$$sim(x, y)_{Lev} = 1.0 - D(x, y) / \max(|x|, |y|) \quad (2.2)$$

To compare inner nodes, Chawathe et al. only uses leaf node descendants. Although this strategy is suitable for flat trees (e.g., \LaTeX document), in software engineering inner node descendants also play a role on similarity. Thus, to customize inner node matching, ChangeDistiller uses both inner and leaf node descendants using Dice function 2.3:

$$sim_{Dice} = \frac{2 * |nodes(x) \cap nodes(y)|}{|nodes(x) \cup nodes(y)|} \quad (2.3)$$

When a node x in T_1 matches multiple nodes in T_2 , ChangeDistiller uses the best matching instead of the first match. In addition, ChangeDistiller uses a dynamic threshold based on tree size ($f = 0.6$ if $n > 4$ and $f = 0.4$, otherwise).

Edit operations are based on a taxonomy of code changes for object-oriented programming. This taxonomy includes 35 distinct changes for object-oriented programming.

2.2.4 GumTree

GrumTree [31] is an edit script algorithm based on developers’ viewpoint of source code edits. Developers often want an edit script that reflects their edits instead of the shortest edit script [31]. GumTree focuses on mapping. It applies Chawathe et al. to compute edit operations. Mapping is divided into *top-down* and *bottom-up* phases: the top-down phase

finds isomorphic subtrees by decreasing height. The bottom-up phase compares nodes based on common descendants from top-down phase. This mapping aligns with developers way to edit source code. First, they search for large pieces of matching code. Then, they search for precise differences in matched nodes [31].

Algorithm 17 shows the top-down algorithm. It includes two priority queues based on height (L_1 for T_1 and L_2 for T_2). Line 1 empties mapping M and auxiliary mapping A . A maintains nodes from T_1 or T_2 that match multiples nodes. Line 2 pushes root of T_1 and T_2 to its priority queue. Lines 3 to 18 execute while L_1 and L_2 contain nodes with height larger than a threshold. If height of L_1 peek is larger than height of L_2 peek, descendants of L_1 peek are added to L_1 . If L_2 peek is larger than L_1 peek, descendants of L_2 peek are added to L_2 . Method *open* adds descendants of a node to the priority queue. The *pop* function returns all the node with height in the head of the priority queue (line 10). For pair of nodes (t_1, t_2) with same height from L_1 and L_2 , line 12 verifies whether t_1 and t_2 are isomorphic. If t_1 or t_2 matches a single node, all the pairs of isomorphic nodes of $s(t_1)$ and $s(t_2)$ to M . Otherwise, (t_1, t_2) are added to A . Line 17 and line 18 add descendants of unmatched nodes to L_1 and L_2 , respectively. Lines 20 – 23 add pair of isomorphic nodes (t_1, t_2) from A with the highest similarity accordingly with Dice function.

Algorithm 18 shows bottom-up algorithm. Line 1 iterates from T_1 nodes that are unmatched and contain matched descendants $s(t_1)$ nodes. Line 2 selects the node $t_2 \in T_2$ with the highest Dice. Line 4 adds (t_1, t_2) to M if their dice exceeds a threshold. Line 6 collects the mapping of a minimum edit script algorithm without move (GumTree adopts RTED [119]). If nodes contain the same label and are unmatched, GumTree adds (t_a, t_b) to M .

Input: Trees T_1 and T_2 , priority queues L_1 and L_2

Output: Matching M

```

1:  $M = \emptyset, A = \emptyset$ 
2:  $push(root(T_1), L_1), push(root(T_2), L_2)$ 
3: while  $min(peekMax(L_1), peekMax(L_2)) > minHeight$  do
4:   if  $peekMax(L_1) \neq peekMax(L_2)$  then
5:     if  $peekMax(L_1) > peekMax(L_2)$  then
6:       for all  $t \in pop(L_1)$  do  $open(t, L_1)$ 
7:     else
8:       for all  $t \in pop(L_2)$  do  $open(t, L_2)$ 
9:   else
10:     $H_1 = pop(L_1), H_2 = pop(L_2)$ 
11:    for all  $(t_1, t_2) \in H_1 \times H_2$  do
12:      if  $isomorphic(t_1, t_2)$  then
13:        if  $\exists t_x \in T_2 \mid isomorphic(t_1, t_x) \wedge t_x \neq t_2$  or  $\exists t_x \in T_1 \mid isomorphic(t_x, t_2) \wedge t_x \neq t_1$  then
14:           $add(A, (t_1, t_2))$ 
15:        else
16:           $add$  all pair of isomorphic nodes in  $s(t_1)$  and  $s(t_2)$  to  $M$ 
17:          for all  $t_1 \in H_1 \mid (t_1, t_x) \notin A \cup M$  do  $open(t_1, L_1)$ 
18:          for all  $t_2 \in H_2 \mid (t_x, t_2) \notin A \cup M$  do  $open(t_2, L_2)$ 
19:           $sort(t_1, t_2) \in A$  using  $dice(parent(t_1), parent(t_2), M)$ 
20:          while  $size(A) \neq 0$  do
21:             $(t_1, t_2) = remove(A, 0)$ 
22:             $add$  all pair of isomorphic nodes in  $s(t_1)$  and  $s(t_2)$  to  $M$ 
23:             $A = A \setminus \{(t_1, t_x \in A)\}, A = A \setminus \{(t_x, t_2 \in A)\}$ 

```

Algorithm 17: GumTree top-down algorithm.

Input: Trees T_1 and T_2 and M

Output: Matching M

```

1: for all  $t_1 \in T_1 \mid t_1 \notin M \wedge t_1$  has matched children, in post-order do
2:    $t_2 = candidate(t_1, M)$ 
3:   if  $t_2 \neq null$  and  $Dice(t_1, t_2, M) > minDice$  then
4:      $add(M, (t_1, t_2))$ 
5:     if  $max(|s(t_1)|, |s(t_2)|) < maxSize$  then
6:        $R = opt(t_1, t_2)$ 
7:       for all  $(t_a, t_b) \in R$  do
8:         if  $t_a, t_b \notin M \wedge l(t_a) = l(t_b)$  then
9:            $add(M, (t_a, t_b))$ 

```

Algorithm 18: GumTree bottom-up algorithm.

Chapter 3

State-of-the-art

In this chapter, we present the state-of-the-art about program transformations. This state-of-the-art is collected using a systematic review. In Section 3.1, we present the background, motivation, goal, and research question of this state-of-the-art. In Section 3.2, we present the review methods, including our research strategy, search string, selection methodology, quality assessment, inclusion and exclusion criteria, extracted information, and execution. In Section 3.3, we present the results of our systematic review. Finally, in section 3.4, we discuss the works in our state-of-the-art.

3.1 Motivation

During software evolution, developers may do repetitive edits [58,91,96,124,144]. These edits are identical or similar to different codebase locations, which may occur as developers add features, refactor, or fix a bug. An example of them is an API replacement where developers must find old API usages and replace it by the new one. Since some repetitive edits are recurring, IDEs automate, in the refactoring suites, common transformations (e.g., rename refactoring). A transformation generalizes developer concrete edits in an abstraction to be applied to other codebase locations.

However, developers still do many transformations absent in IDEs. These edits are often done manually, which is time-consuming and error-prone. To help developers to apply repetitive edits, some techniques were proposed. Nguyen et al. [94] present a recommender to automate repetitive edits based on properties of these edits. Meng et al. [83] propose

a technique that uses two or more examples of edited methods to learn a transformation that locates and edits target methods of the repetitive edit. Raychev et al. [125] propose an approach that uses refactorings from IDEs to compose complex refactorings.

Researchers also reviewed techniques for repetitive edits. Monperrus [89] write a bibliography on automatic software repair. Zibran and Roy [156] survey clone management. Park et al. [118] present an empirical study on supplementary bug fixing. Mens and Tourwé [85] and Pakdeetrakulwong et al. [117] present techniques to recommend source code edits. These works focus on specific transformations, and many of them are not conducted in a systematic way. Furthermore, Kim et al. [58] study techniques to automatize repetitive edits, but this study is restricted to a few pre-selected papers.

In this section, we present a systematic review of techniques to do repetitive edits. These edits could occur horizontally (i.e., across commits/revisions) or vertically (i.e., on a single commit/revision).

Our goal is to characterize techniques to transform software code on perspective of repetitive edits. Formally, we have:

Goal : *analyze techniques to transform source code for the purpose of characterizing them with respect to technique properties from the point of view of software developers in the context of software evolution.*

3.1.1 Review questions

We base our review on Kitchenham and Charters [60] guidelines to do systematic reviews in software engineering. Our research question is the following:

- **RQ 01**: what are the techniques to do transformations in software engineering?

We collect studies to directly answer our RQ (i.e., primary studies). In addition, we collect studies that review primary studies (i.e., secondary studies) such as systematic literature review, mapping study, and surveys.

3.2 Review methods

We first conduct an exploratory study. This study aims to select studies to base our review. It helps to define places to search for studies (i.e., databases) and terms to build our search string (e.g., major terms and synonyms). Table 3.1 shows works on this exploratory study.

Table 3.1: Studies from the exploratory study.

Reference	Name
Nguyen et al. [94]	A Study of Repetitiveness of Code Changes in Software Evolution
Ray et al. [124]	The Uniqueness of Changes: Characteristics and Applications
Meng et al. [83]	LASE: Locating and Applying Systematic Edits by Learning from Examples
Robbes and Lanza [126]	Example-based Program Transformation
Raychev et al. [125]	Refactoring with Synthesis,
Kim and Meng [58]	Recommending Program Transformations to Automate Repetitive Software Changes
Nguyen et al. [98]	A Graph-Based Approach to API Usage Adaptation
Bravenboer et al. [8]	Stratego/XT 0.17. A language and toolset for program transformation
Cordy [15]	Source Transformation, Analysis and Generation in TXL
Boshernitsan et al. [58]	Aligning Development Tools with the Way Programmers Think about Code Changes
Toomim et al. [144]	Managing Duplicated Code with Linked Editing

3.2.1 Search strategy

To select studies for our SLR, we apply the following methodology:

- a) We derive from population, intervention, and outcome major terms for our search string.
- b) We identify alternative spelling and synonym for major terms.
- c) We identify databases to apply the search string.
- d) We build a search string for each database.

- e) We do the search.
- f) We do backward snowballing (i.e., analyze works in reference section) and forward snowballing (i.e., analyze who cited this work). For backward snowballing, we analyze related works. Based on Kitchenham Brereton [61], for forward snowballing, we select critical works (10% of total) based on number of citations by year and collect works citing it.

Regard the item **a)**, search terms derive from population, intervention, and outcome. Population is who/what intervention affects. Intervention is what is being studied, and outcome denotes observed results. Our population is software engineers (target of transformation techniques). Intervention is techniques to do transformation, and outcome is technique transformations. To answer the **b)**, we use our exploratory study to identify synonyms and alternative spelling, which are shown in Table 3.2.

Table 3.2: Synonyms and alternative spelling.

Value	Alternative Spelling and Synonyms
Software engineer	Developer, programmer
Systematic change	Repetitive edit, repetitive change, non-unique change, duplicate change, systematic edit, linked edit, refactoring
Technique	Tool, recommender, algorithm, synthesizer

For item **c)**, we identify databases. Based on [61], we choose ACM and IEEE along with ScienceDirect, Engineering Village, and Scopus. Table 3.3 shows these databases.

Table 3.3: Criteria for our systematic review.

Feature	Value
Language	English
Search method	The work must be available online.
Databases	IEEE Explore ACM Digital Library ScienceDirect Engineering Village Scopus
SR evaluation	Author
Publication year	After 2005

3.2.2 Search string

Search strings may vary among databases. Default search string for our SLR is the following:

(“software engineering” OR developer OR programmer) AND (“systematic change” OR “program repair” OR “API usage adaptation” OR “Code change” OR “program change” OR “program transformation” OR “script transformation” OR “source transformation” OR “code transformation” OR “repetitive change” OR “non-unique change” OR “duplicated change” OR “duplicated code” OR “systematic edit” OR “linked edit” OR “refactoring” OR “bug-fix”) AND (technique OR tool OR recommender OR algorithm OR synthesizer)

For each database, we search based on title, abstract, and meta-data. If databases lack these characteristics, we search based on the full text (if allowed) or default (otherwise).

- **IEEE Explorer:** we use advanced search and do three searches based on title, abstract, and meta-data. This database restricts search string to 15 terms. Since we have more than 15 terms, we do different search and concatenate results.
- **ACM Digital Library:** we use advanced search and do three searches based on title, abstract, and meta-data.
- **ScienceDirect:** we use advanced search. we select “Computer Science” for area and do the search.
- **Engineering Village:** we use advanced search. We select “Computer Science” for area and do the search.
- **Scopus:** we access search area and do the search.

3.2.3 Selection methodology

We use the StArt [64], a tool for conducting systematic review. We divided our methodology in selection and extraction stage as follows. For each database, we searched based on database syntax (see Section 3.2.2). On selection stage, we analyze title and abstract. If the work fits an exclusion criterion, we reject; otherwise, we accept it. On extraction stage, we scan the work. If work fits an exclusion criterion, we reject; otherwise, we analyze full text and classify it. Finally, we write the report based on accepted PSs.

3.2.4 Quality assessment

We base on Kitchenham and Charters [60] quality assessment. We remove works focused on automatic program repair since extensive study focuses on it [89] that by itself can be a systematic review. We also discard works on clone removal. In addition, we remove works on transformations for a too specific problem (stylized edits) [83]. For instance, works in concurrent transformations [9, 131, 149], removal of security issues [53, 75], refactoring of web applications [95], refactoring to aspects [77, 145, 146], refactoring C macros [79, 113], refactoring product lines [132], among others. Table 3.4 shows our quality criteria.

Table 3.4: Quality assessment score.

Id	Criterion	Score	Score response
Empirical study			
01	Clear, unambiguous		Yes = 1, No = 0
02	Full work (≥ 8 pages)		Yes = 1, No = 0
03	Besides Fowler catalog		Yes = 1, No = 0
04	Automate transformations		Yes = 1, No = 0
05	Many transformations		Yes = 1, No = 0
Theoretical study			
06	Clear		Yes = 1, No = 0
07	Well referenced		Yes = 1, No = 0
Total quality score			

3.2.5 Inclusion and exclusion criteria

We are interested in works related to software engineering, answering our RQ. The inclusion and exclusion criteria for our systematic review can be found on Table 3.5. In particular, we are interested in systematic change studies. Works that apply identical or similar transformation throughout the source code.

Table 3.5: Inclusion and exclusion criteria.

Feature	Value
Inclusion criteria	a) The work answers our research question b) The study has a high quality
Exclusion criteria	a) The work does not deal with systematic changes b) The work is external to software engineering c) The document does not make part of a conference or journal d) The document is unavailable online
Study type	Cases studies, empirical studies, and second studies
Duplicate studies	Select the most recent version

3.2.6 Information extraction

From PSs, besides the standard data (e.g., title, author, and proceeding) shown in Table 3.6, we collect data to answer our RQ. For secondary studies, we collect PSs that contributes to answer our RQ.

To answer our RQ, we collect:

1. **Locations:** we collect location strategy (i.e., manual, semi-automatic, or automatic).
2. **Transformation:** we collect edit strategy (i.e., manual, semi-automatic, or automatic).
3. **Location template:** we collect template learning for location (i.e., manual, semi-automatic, or automatic).
4. **Transformation template:** we collect template for transformation (i.e., manual, semi-automatic, or automatic).
5. **Transformation scope:** we collect transformation scope (e.g., method, API, and refactoring).
6. **Manipulation:** we collect technique manipulation (i.e., text, syntax, semantic).
7. **Language:** we collect target language (e.g., Java, C#, C).
8. **Input:** we collect input (e.g., input-output examples, history, before-after API).
9. **Output:** we collect output (e.g., modification list, modified source code).
10. **Learning:** we collect learning (i.e., on-line, off-line).

11. **Reuse:** we collect whether transformation could be reused.
12. **Catalog-bounded:** we collect whether transformation catalogs bound techniques.
13. **Abstraction:** we collect elements abstracted.
14. **Interaction:** we collect developers' interaction model.
15. **Technique scope:** we collect technique scope (e.g., method body, attribute, or class).
16. **Context:** we collect whether technique models context.
17. **Similarity:** we collect whether technique does similar or identical transformations.
18. **Multiple or single:** we collect whether technique works on single or multiple files.

For example-based techniques, we collect:

19. **Number of examples:** we collect number of examples.
20. **Example selection methodology:** we collect example selection methodology.

If any feature is absent, we set NP value. If feature could not be identified, we set a NA value.

Table 3.6: Default information collected from each primary study.

Item	Description
Item type	Study type (e.g., journal, conference)
Title	Study title
Author(s)	Study author(s)
Publication	Study series
Volume	Study volume
Issue	Study issue
Pages	Study pages
Year	Study year

3.2.7 Execution

Form 1¹ shows extracted data for each PS in the final list of works. In the selection stage, we selected 4,741 works. From these works, 2,849 were rejected, 1,319 were duplicated, and 573 were accepted. From the 573 works for the extraction stage, 362 were accepted, 173 were rejected, and 40 were duplicated. Due to the large number of accepted works (573), we apply further filtering. First, we reject works based on publication year. We only accepted works published after 2005. Second, we reject works based on quality criteria. We only accept works with maximum quality score. Finally, we end up with 51 PSs.

3.3 Results

We select 51 works on total. Figure 3.1 summarizes PSs based on events.

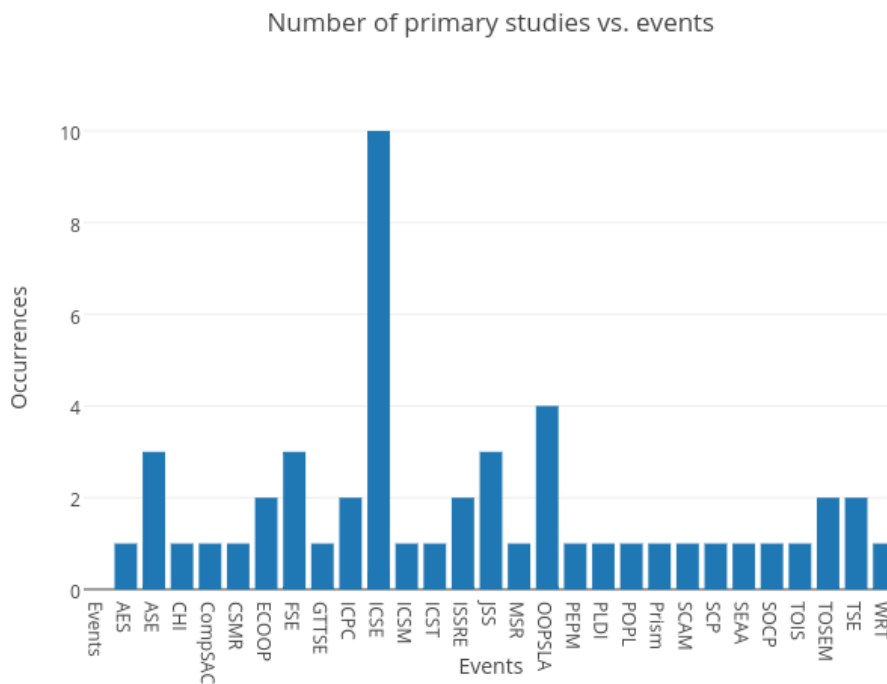


Figure 3.1: Number of PS for each event.

¹<https://goo.gl/wBV7dR>

3.4 Discussion

In this section, we present the primary studies selected to be included in the list of works for our systematic review. Techniques to automate transformations range from PBE (Section 3.4.1), linked editing (Section 3.4.2), API usage (Section 3.4.3), bug fixing (Section 3.4.4), complex refactoring (Section 3.4.5), and complex transformations (Section 3.4.6).

3.4.1 Programming by examples

PBE techniques use developers' edits or modification from repositories to learn transformations to edit other code locations.

Automatic transformation techniques

Lase [83] is a technique for doing repetitive edits using examples. Developers give two or more edited methods as examples, and Lase creates a context-aware abstract transformation that could be applied to other locations in codebase. It uses clone detection and dependence analysis to identify methods where transformation should be applied. Andersen and Lawall [3] present *spdiff*, a technique that extracts common edits from two versions of source files. Then, it learns a transformation that could be applied to other codebase locations. The key idea of Meng et al. [81] is to remove code clones from the source code based on examples. Their technique (Rase) combines refactorings (e.g., extract method, add parameter, introduce exit label) to extract and remove similar code. It extracts common code from clones and creates a new method that abstract clones, removing them from code. Kessentini et al. [55] propose a technique to remove design defect from the source code. Their technique receives defect examples and quality metrics to learn rules to refactor source code.

Edit-based techniques

Some techniques require that developers indicate locations where transformations could be applied. Sydit [82] receives one or more edited methods as examples and learned a transformation to apply to locations indicated by developers. Santos et al. [130] propose MacroRecorder, a technique that records developers code edit events from IDE to reproduce them in other code locations. MacroRecorder semi-automatically learns a transformation and allows the developers to customize the transformation.

Semi-automatic techniques

Some techniques help developers to write their own transformations. Boshernitsan et al. [7] present iXj, a technique that semi-automatically allows developers to specify transformations. Based on mental models of developer interaction, they developed a visual tool that allows developers to specify transformations. Rosemari [142] is a semi-automatic technique that migrates programs for annotated version of a framework based on examples. It infers refactoring rules from two versions of a single class provided by developers. Developers configure transformations inferred by Rosemari, which refactor the source code automatically.

Refactoring

Other techniques use developers' edits on IDE to recommend refactorings. BeneFactor [37] is a technique that detects whether developers are applying a refactoring and can finish the refactoring to developers. BeneFactor uses developers' edits as example and automatically detects whether developers are applying a refactoring. If this occurs, BeneFactor identify the refactoring and can finish the refactoring.

3.4.2 Linked editing

Managing clones is expensive. The main bottleneck relies in update code clones. Linked editing techniques offer strategies to update clones. JSync [97] and Clever [100] are techniques for clone management. These techniques detect code clones, detect edits in clones, and notify developers for clone inconsistencies. CloneTrack [22] is a technique that uses clone region descriptors (CRDs) to management of code clones. CRDs combine syntactic, structural, and lexical information to represent code clones in codebase. CloneTrack uses output of a clone detection technique and produces CRDs to represent clone regions. Developers can specify clones to track, and the technique notifies developers for modification and supports update in these clones. The key idea of Wit et al. [20] is to keep track copy and paste operations as clone relations. Their technique (CloneBoard) detects when developers edit a clone relation and shows clones in the relation. It semi-automatically supports update of clones in this relation.

3.4.3 API usage

Source code often uses external API providers. These providers should maintain external APIs unchanged, but sometimes it does not happen [153]. Thus, clients need to update usage to conform new API. In addition, developers often need code completion systems to guide them to use an API. IDEs provide code completion systems, but IDEs' recommendations often rank suggestions based on alphabetic order, which does not fit API context.

DSL-based API adaptation

To help clients to adapt API, providers may give tools to automate it (e.g., Microsoft wizard, which helps in migrating Visual Basic to Visual Basic.Net [70]). However, providing a transformation tool is difficult, and developers often migrate API manually. To help developers, some studies use transformation DSL. Nita and Notkin [101] present twinning, a technique to update API by mapping usage between two codebase versions. Developers write rules describing correspondences usage in these versions to update code. Patl [147] is a DSL that allows developers to write rules for a transformation where many invocations from old API are replaced by many invocations to new API (many-to-many mapping). The key idea for Li et al. [70] is a type-safe transformation DSL for API migration. Their technique (Swin) avoids type bugs in transformed program. This language updates code where one method invocation in old API usage is replaced by one or multiple invocations in the new API usage (one-to-many mapping). Wu et al. [151] present CMIT, a transformation language for API adaptation. CMIT allows transformation where a sequence of method invocation is considered together during transformation (compositional-mapping).

Automatic and semi-automatic API adaptation

To reduce the burden of writing transformations since developers have to learn how to use a DSL, some techniques analyze code usage from new and old API to update API. Nguyen et al. [98] present LibSync, a technique that recommends locations and edits for API adaptation. LibSync uses clients that migrated the API to learn transformations. LibSync compares two library versions to identify edits, extract templates for the usage, and build patterns of API usage. Based on these patterns, it recommends API adaptation. The key idea of Dagenais and Robillard [18] is to suggest updates based on adaptations of the own framework. Their technique (SemDiff) analyzes framework source code, identifies edits (e.g., method additions

and deletions) and, based on these edits and methods removed from framework, recommends replacements. Similarly, Xing and Stroulia [153] present DiffCatchUp, a technique that recognizes API changes and support client update based on examples from framework source code. The key idea of Cossette et al. [17] is to help developers to migrate API when framework updates are undocumented or incomplete. In this case, developers must discover how to use new API. They propose Umani, a technique that discovers replacements for breaking APIs based on structural similarity from new API and old API. Other techniques are semi-automatic. Kapur et al. [54] propose Trident, a technique for API migration. After selection a code fragment, developers configure refactorings and Trident shows all locations following the same pattern. Then, developers select locations that need to be edit and configure transformation to be applied.

Code completion based on statistical models

Another way to help developers in API usage is using code-completion. Some techniques use high repetitiveness of code edits [36,47] to build statistical models for code completion. Nguyen et al. [92] present APIRec, a technique that recommends API usage based on repetitiveness of code edits. APIRec is trained in a corpus of repetitive edits, learn a statistical model based on the co-occurrence of fine-grained changes, and recommends API calls based on edit context. GraLan and ASTLan [93] are graph-based and AST-based techniques that recommend the next element (e.g., API method, field access, and related control units) based on statistical models of code usage. These techniques are based on n-gram, which is a statistical model that assumes a sequence from right to left, and next elements depend only on current context. They consider source code properties that are difficult model in n-gram (i.e., API usage often lacks order, repetitive code often interleaves with project specific code, and elements could be apart from one another).

Mining repository for code completion

Other works mines repositories to recommend code completion. Nguyen et al. [94] mined repositories to identify characteristics of repetitive edits, focusing in three dimensions: size, type, and general/fixing edits. They build a recommender system based on these characteristics. Ray et al. [124] studied edits characteristics of unique/repetitive edit from repository (e.g., extent of unique edits, who introduce unique edits, and where the edits take place). They propose two recommender systems. The first recommends previous edits when developers

select a fragment of code to edit. The second recommends edits that co-occurred when developers do a repetitive edit.

Code completion based on standard machine learning algorithm

Other techniques use standard machine techniques in code completion. Bruch et al. [11] present three code completion systems based on repositories using K-nearest neighbor algorithm. These systems search for code fragments that use variables of the same type of variable developers want a recommendation. Each code completion system recommends based on a specific metric: frequency of method calls, rule mining, and the closest code fragment.

3.4.4 Bug fixing

Bugs are frequent along software development. Although developers try to remove bugs from source code, many bugs persist in systems bug reports. Human resources are often insufficient to remove all bugs, even for well-known bugs (e.g., Windows 2000 was shipped with thousands of well-known bugs) [56]. Bug fixing techniques help developers to remove bugs from code.

Recurring bug fixing

Some bugs re-occur [59]. Thus, developers may fix these bugs similarly. To help developers, some techniques were proposed. Nguyen et al. [99] present FixWizard, a semi-automatic technique to fix recurring bugs based on properties of recurring fixes (e.g., why, where, and how fixes occur and how to detect fixes). FixWizard identifies target locations, detects recurring fixes, and recommends fixing edits. BugFix [50] is a technique that uses association rules to help developers to fix a bug. This tool suggests fixes by analyzing historic of bug fixing scenarios. Association rules map debugging scenarios with respective fixing scenarios. BugFix uses these rules to suggest a ranked list of fixes. The key idea of Kim et al. [56] is to use human-written patches to repair programs. They analyze 62,652 human-written and identified ten common fixing templates. Their technique (Par), detects bug locations using a fault location technique [67] and uses these fixing templates to repair program. Liu et al. [72] propose R2Fix, a technique for automatic program repair that learns patches from bug reports. R2Fix analyzes bug reports, identifies common bugs, and recommends program repairs. This technique contains a bug classifier, a parameter extractor that extracts patterns based on bug reports, and a patch generator. Long and Rinard [74] present Prophet, a technique for

automatic program repair that learns a probabilistic application-independent model of correct code from a set of successful human patches.

Semi-automatic techniques for bug fixing

Some techniques are semi-automatic. Xia and Lo [152] present SupLocator, a technique that recommends target methods of supplementary bugs fixings. To identify target methods, this technique uses six edit relationships: method invocation, method containment, inheritance, co-change, context similarity, and name similarity. SupLocator builds a graph for each relationship and uses a genetic algorithm to recommend target methods of supplementary fixes. PatternBuild is a semiautomatic technique to help developers to resolve bugs that propagate over codebase. PatternBuild receives a bug version of project and a version with one or more fixed instances. PatternBuild allows developers to specify a bug pattern based on bug version and a fixing pattern based on fixed version. This technique searches for locations target of fixing based on the bug pattern. It shows these locations to developers that can resolve bug using the fixing pattern. Sun et al. [141] present a semi-automatic technique that helps developers to fix a bug that propagates over codebase. This technique offers basic rules to fix bugs based on three type of bug fixing usage: precondition, which requires checking input parameters before calling the function; post-condition, which requires checking return after calling the function; and call-pair, which requires that two functions be called together in a particular order. Developers can compose complex rules based on these rules. This technique detects rule violations and suggests fixing.

3.4.5 Complex refactoring

IDEs provide refactorings, but developers often refactor manually. Besides knowing that refactoring automatically is faster than by hand, developers need to know that the refactoring exists, the refactoring name, and how to apply it [34]. In addition, IDEs refactoring suites are often limited. For complex refactorings, developers have to refactor manually or implement their own refactoring [69]. Refactoring by hand is error-prone and time-consuming. Specifying a refactoring requires knowing to represent a program and available APIs.

Search-based techniques

Some works search over a space of complex refactoring, selecting a refactoring that optimizes

a function. Recon [90] is a technique that formulates refactorings as an optimization problem to find best refactoring sequence for source code. Recon uses developers' context and meta-heuristics to compute a refactoring sequence that affects elements in developer's context. Based on interaction traces, Recon generated an abstract model of source code. This technique detects anti-patterns and searches for refactoring sequences to fix these anti-patterns. O'Keefe and Cinnéide [103] present a technique that formulates refactorings as a search problem. Based on design quality function, this technique selects the best refactoring candidate over a space of design refactorings.

Multi-objective refactoring

Traditional refactorings aim to improve code quality by changing internal structure, but maintaining external behavior or refactored program. Some studies investigate other criteria besides code quality. Ouni et al. [111] present an approach that recommends a refactoring sequence based on multi-objective criteria. Over a space of refactoring sequences, this approach searches for a refactoring sequence that improves design quality, preserves design coherence and consistency of refactored program, minimizes code edits, and maximizes consistency with developers' history. Ouni et al. [112] propose a multi-objective approach to recommend a refactoring sequence based on three criteria: minimize code-smells, maximize use of developers' history, and preserve construct semantics. The key idea of Ouni et al. [110] is to use recorded edits from developers to recommend a refactoring sequence. They propose a multi-objective approach to find the best refactoring sequence that maximizes use of past refactorings, minimizes semantic errors, and minimizes defects.

Program language for complex refactoring

Some researchers allows developers to write their own refactoring using a DSL. Li and Thompson [69] present a framework that allows developers to write their own complex refactorings. The framework is built on Wrangler refactoring tool. Developers write refactorings using Erlang syntax, making it suitable for developers familiar with Erlang. RefaCola [139] is a refactoring DSL that allows writing refactoring with constraint. The key idea of Ruhroth et al. [129] is to allow developers to write refactoring for any language that follows Backus-Naur-Form. They present $\text{Re}\mathcal{L}$, a domain specific language for refactoring that allows developers to specify refactoring and execute refactorings in source code.

Learning refactoring from developer edits

Some works use developers' edits to learn complex refactorings. Raychev et al. [125] propose ReSynth, a technique that synthesizes a refactoring sequence. Developers active ReSynth, edit code base, and call ReSynth to complete refactoring. Given before/after version, ReSynth computes edits to synthesize a refactoring sequence. ReSynth requires developers to provide a successors function describing how to refactor a program. WitchDoctor [34] is a technique that detects a refactoring while developers edit code and recommends a refactoring before developers complete refactoring. The key idea of Hayashi et al. [45] is to learn a refactoring sequence based on developers' edits on repositories. This technique identifies locations and applies refactoring automatically.

Complex refactoring

Some techniques analyze source code to identify complex refactoring. Alkhalid et al. [1] propose a technique that clusters code at function/method level to suggest refactoring. This technique reprocesses code using an Adaptive K-nearest Neighbor to cluster source code. This technique shows clusters for developers that could apply extract method to refactoring code. R3 [57] is a technique that encodes Java entities in a database and maps refactorings in database to the source code. R3 compose complex refactoring as a sequence of primitive refactorings. Refactorings are applied to database. Source code is edited when developers modify pretty-printing ASTs mapped on database.

3.4.6 Complex transformations

Besides refactoring, developers may apply other transformations (e.g., add a feature, or fix a bug). Some techniques allow developers to apply complex transformations. Some researchers use a DSL for complex transformations. Cordy [15] present TXL, a DSL for source transformations. TXL requires specification of DSL structure and transformation rules. Based on these rules, TXL applies transformations and outputs transformed versions. Stratego/XT [8] is an infrastructure for program transformations. Stratego/XT uses Stratego, a DSL for program transformations and XT, an environment for development of transformations. Erwig and Ren [28] present a DSL for program transformation. They show language elements and an application of this DSL to update lambda calculus programs.

3.4.7 Validation

In this section, we present the validation process of our systematic review. To validate our protocol, we conduct a pilot study following our protocol. We used this pilot study to review of the search string and databases. We have a set of works relevant to our study in the folder Relevant Papers². Our study must select this set of works. If work in this folder is not returned, we added this work to our exploratory study, refine our search string and do a new search. We do this process until we select all works in relevant works folder.

²goo.gl/W4IKEr

Chapter 4

REFAZER

In this chapter, we describe our technique for synthesizing program transformations from input-output examples. REFAZER builds on PROSE (Section 2.1.2), a framework for program synthesis from examples and under-specifications as explained in Section 2.1.2. In this framework, application designers create a DSL for the specific domain. Given a *spec* φ with inputs and constraints on the desired outputs, PROSE synthesizes programs in the DSL that are consistent with φ , combining *deduction*, *search*, and *ranking*.

REFAZER consists of three main components, which are illustrated in Figure 4.1:

- *A DSL for describing program transformations.* Its operators partially abstract the edits provided as examples. The DSL is expressive enough to capture common transformations but restrictive enough to allow efficient synthesis.
- *Witness functions.* In PROSE, a *witness function* ω_F is a backpropagation procedure, which, given a spec φ on a desired program on kind $F(e)$, deduces a necessary (or even sufficient) spec $\varphi_e = \omega_F(\varphi)$ on its subexpression e . Witness functions enable efficient top-down synthesis algorithms in PROSE.¹
- *Ranking functions.* Since example-based specifications are incomplete, the synthesized abstract transformation may not perform the desired transformation on other input programs. We specify *ranking functions* that rank a transformation based on its robustness (i.e., likelihood of it being correct in general).

¹Another view on witness functions ω_F is that they simply implement *inverse semantics* of F , or a generalization of inverse semantics w.r.t. some *constraints* on the output of F instead of just its *value*.

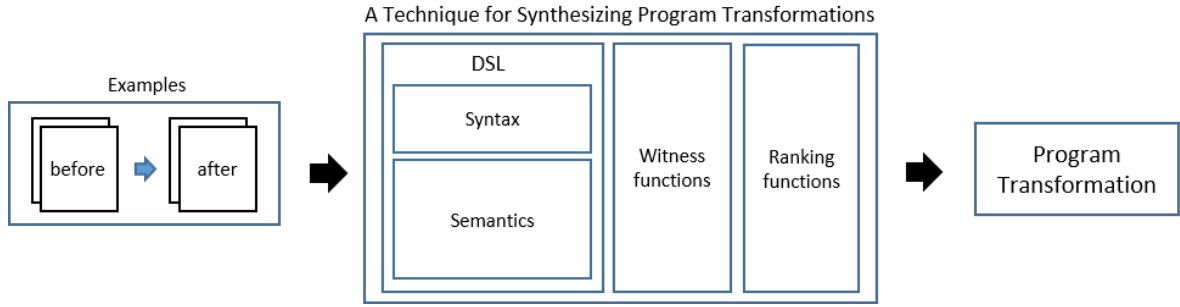


Figure 4.1: The workflow of REFAZER. It receives an example-based specification of edits as input and returns a set of transformations that satisfy the examples.

4.1 A DSL for AST transformations

In this section, we present our DSL for program transformations, hereinafter denoted \mathcal{L}_T . It is based on tree edit operators (e.g., `Insert`, `Delete`, `Update`), list processing operators (`Filter`, `Map`), and pattern-matching operators on trees. The syntax of \mathcal{L}_T is formally given in Figure 4.2.

```

transformation ::= Transformation(rule1, ..., rulen)
rule           ::= Map( $\lambda x \rightarrow$  operation, locations)
locations      ::= Filter( $\lambda x \rightarrow$  Match(x, match), AllNodes())
match         ::= Context(pattern, path)
pattern       ::= token | Pattern(token, pattern1, ..., patternn)
token        ::= Concrete(kind, value) | Abstract(kind)
path         ::= Absolute(s) | Relative(token, k)
operation     ::= Insert(x, ast, k) | Delete(x, ref)
               | Update(x, ast) | Prepend(x, ast)
ast          ::= const | ref
const        ::= ConstNode(kind, value, ast1, ..., astn)
ref         ::= Reference(x, match, k)

```

Figure 4.2: A core DSL \mathcal{L}_T for describing AST transformations. *kind* ranges over possible AST kinds of the underlying programming language, and *value* ranges over all possible ASTs. *s* and *k* range over strings and integers, respectively.

A transformation T on an AST is a list of *rewrite rules* (or simply “rules”) r_1, \dots, r_n . Each rule r_i specifies an *operation* O_i that should be applied to some set of *locations* in the

input AST. The locations are chosen by *filtering* all nodes within the input AST w.r.t. a *pattern-matching predicate*.

Given an input AST P , each rewrite rule r produces a *list of concrete edits* that may be applied to the AST. Each such edit is a replacement of some node in P with a new node. This set of edits is typically an overapproximation of the desired transformation result on the AST; the precise method for applying the edits is domain-specific (e.g., based on verification via unit testing). We discuss the application procedures for our studied domains in Chapter 4.2. In the rest of this subsection, we focus on the semantics of the rules that suggest the edits.

A rewrite rule consists of two parts: a *location expression* and an *operation*. A location expression is a `Filter` operator on a set of sub-nodes of a given AST. Its predicate $\lambda x \rightarrow \text{Match}(x, \text{Context}(\text{pattern}, \text{path}))$ matches each sub-node x with a *pattern expression*.

Patterns

A pattern expression `Context(pattern, path)` checks the *context* of the node x against a given *pattern*. Here *pattern* is a combination of `Concrete` tokens (which match a concrete AST) and `Abstract` tokens (which match only the AST kind, such as `IfStatement`). In addition, a *path expression* specifies the expected position of x in the context that is described by *pattern*, using a notation similar to XPath [150]. This allows for a rich variety of possible pattern-matching expressions, constraining the ancestors or the descendants of the desired locations in the input AST.

Example 1. Figure 4.3 shows a transformation that describes our running example from Figure 1.2. This transformation contains one rewrite rule. Its location expression is

$$\text{Filter}(\lambda x \rightarrow \text{Context}(\pi, \text{Absolute}(""))))$$

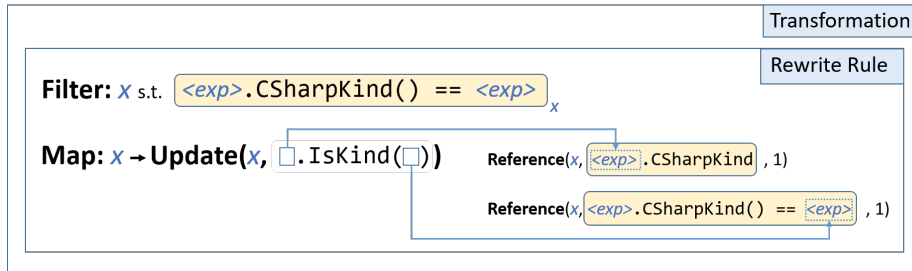
where

$$\pi = \text{Pattern}(\boxed{==}, \text{Pattern}(\boxed{.}, t_e, t_m), t_e)$$

$$t_e = \text{Abstract}(\boxed{\langle \text{exp} \rangle})$$

$$t_m = \text{Concrete}(\boxed{\langle \text{call} \rangle}, \text{"CSharpKind()"})$$

The path expression `Absolute("")` specifies that the expected position of a location x in π should be at the root – that is, the pattern π should match the node x itself.



(a) A synthesized AST transformation.

```

1 while (receiver.CSharpKind() ==
2   SyntaxKind.ParenthesizedExpression) {...}
3 ...
4 foreach (var m in modifiers) {
5   if (m.CSharpKind() == modifier) {return true;}
6 };
7 ...
8 if (r.Parent.CSharpKind() ==
9   SyntaxKind.WhileStatement) {...}

```

(b) A C# program used as an input to the transformation.



(c) A list of edits produced after instantiating (a) to (b).

Figure 4.3: An example of a synthesized transformation and its application to a C# program, which results in a list of edits.

Operations

Given a list of locations selected by the `Filter` operator, a rewrite rule applies an *operation* to each of them. An operation O takes as input an AST x and performs one of the standard tree edit procedures [119, 155] on it:

- Insert some fresh AST as the k^{th} child of x ;
- Delete some sub-node from x ;
- Update x with some fresh AST;
- Prepend some fresh AST as the preceding sibling of x .

An operation creates fresh ASTs using a combination of *constant ASTs* `ConstNode` and *reference ASTs* `Reference`, extracted from the location node x . Reference extraction uses the same pattern-matching language, described above. In particular, it can match over the ancestors or descendants of the desired reference. Thus, the semantics of reference extraction `Reference(x, Context(pattern, path), k)` is:

1. Find all nodes in x s.t. their surrounding context matches *pattern*, and they are located at *path* within that context;
2. Out of all such nodes, extract the k^{th} one.

Example 2. For our running example from Figure 1.2, the desired rewrite rule applies the following operation to all nodes selected by the location expression from Example 1:

$$\text{Update}(x, \text{ConstNode}(\langle \text{call} \rangle, \text{"IsKind"}, \ell_1, \ell_2))$$

where

$$\ell_1 = \text{Reference}(x, \text{Context}(\pi_1, s_1), 1)$$

$$\ell_2 = \text{Reference}(x, \text{Context}(\pi_2, s_2), 1)$$

$$\pi_1 = \text{Pattern}(\boxed{\cdot}, t_e, t_m)$$

$$\pi_2 = \text{Pattern}(\boxed{==}, \text{Pattern}(\boxed{\cdot}, t_e, t_m), t_e)$$

$$s_1 = \text{Absolute}(\text{"1"}) \quad s_2 = \text{Absolute}(\text{"2"})$$

and t_e and t_m are defined in Example 1. This operation updates the selected location x with a fresh call to `IsKind`, performed on the extracted receiver AST from x , and with the extracted right-hand side AST from x as its argument.

4.1.1 Synthesis algorithm

We now describe our algorithm for synthesizing AST transformations from input-output examples. Formally, it solves the following problem: given an example-based spec φ , find a transformation $T \in \mathcal{L}_T$ that is consistent with all examples $(P_i, P_o) \in \varphi$. We denote this problem as $T \models \varphi$. A transformation T is consistent with φ if and only if applying T to P_i produces the concrete edits that yield P_o from P_i .

Recall that the core methodology of PBE in PROSE is *deductive synthesis* (i.e., *backpropagation*). In PROSE, a problem of kind $F(T_1, T_2) \models \varphi$ is reduced to several subproblems of kinds $T_1 \models \varphi_1$ and $T_2 \models \varphi_2$, which are then solved recursively. Here φ_1 and φ_2 are fresh specs, which constitute necessary (or even sufficient) constraints on the subexpressions T_1 and T_2 in order for the entire expression $F(T_1, T_2)$ to satisfy φ . In other words, the examples on an operator F are backpropagated to examples on the parameters of F .

As discussed previously, the backpropagation algorithm relies on a number of modular operator-specific annotations called *witness functions*. Even though PROSE includes many generic operators with universal witness functions out of the box (e.g., `list-processing Filter`), most operators in \mathcal{L}_T are domain-specific, and therefore require non-trivial domain-specific insight to enable backpropagation. The key part of this process is the witness function for the top-level `Transformation` operator.

The operator `Transformation($rule_1, \dots, rule_n$)` takes as input a list of rewrite rules and produces a transformation that, on a given AST, applies these rewrite rules in all applicable locations, producing a list of edits. The backpropagation problem for it is stated in reverse: given examples φ of edits performed on a given AST, find necessary constraints on the rewrite rules $rule_1, \dots, rule_n$ in the desired transformation.

The main challenges that lie in backpropagation for `Transformation` are:

1. Given an input-output example (P_i, P_o) , which often represents the entire codebase/namespace/class, find examples of individual edits in the AST of P_i ;

2. Partition the edits into clusters, deducing which of them were obtained by applying the same rewrite rule;
3. For each cluster, build a set of operation examples for the corresponding rewrite rule.

Finding individual edits

We resolve challenge 1 by calculating *tree edit distance* between P_i and P_o . Note that the state-of-the-art Zhang-Shasha tree edit distance algorithm [155] manipulates single nodes, whereas our operations (and, consequently, examples of their behavior) manipulate whole subtrees. Thus, to construct proper examples for operations in \mathcal{L}_T , we group tree edits computed by the distance algorithm into connected components. A connected component of node edits represents a single edit operation over a subtree.

Partitioning into rewrite rules

To identify subtree edits that were performed by the same rewrite rule, we use the DB-SCAN [29] clustering algorithm to partition edits by similarity. Here we conjecture that components with similar edit distances constitute examples of the same rewrite rule.

Algorithm 19 describes the steps performed by the witness function for `Transformation`. Lines 2-6 perform the steps described above: computing tree edit distance and clustering the connected components of edits. Then, in lines 7-11, for each similar component, we extract the topmost operation to create an example for the corresponding rewrite rule. This example contains the subtree where the operation was applied in the input AST and the resulting subtree in the output AST.

4.1.2 Ranking

The last component of REFAZER is a ranking function for transformations synthesized by the backpropagation algorithm. Since \mathcal{L}_T typically contains many thousands of ambiguous programs that are all consistent with a given example-based spec, we must disambiguate among them. Our ranking function selects a transformation that is more likely to be robust on unseen ASTs – that is, avoid false positive and false negative matches. REFAZER selects the transformation on the top even if other transformations contain the same score. The ranking is based on the following principles:

Input: Example-based spec φ

```

1: result := a dictionary for storing examples for each input
2: for all ( $P_i, P_o$ ) in  $\varphi$  do
3:   examples := empty list of refined examples for edits
4:   operations := TREEEDITDISTANCE( $P_i, P_o$ )
5:   components := CONNECTEDCOMPONENTS(operations)
6:   connectedOpsByEdits := DBSCAN(components)
7:   for all connectedOps  $\in$  connectedOpsByEdits do
8:     ruleExamples := MAP(connectedOps,
        $\lambda ops \rightarrow$  create a single concrete operation based on ops)
9:     examples += ruleExamples
10:  result[ $P_i$ ] += examples
11: return result

```

Algorithm 19: Backpropagation procedure for the DSL operator `Transformation($rule_1, \dots, rule_n$)`.

- Favor Reference over ConstNode: a transformation that reuses a node from the input AST is more likely to satisfy the intent than one that constructs a constant AST.
- Favor patterns with non-root paths (i.e., patterns that consider surrounding context of a location). A transformation that selects its locations based on surrounding context is less likely to generate false positives.
- Among patterns with non-empty context, favor the shorter ones. Even though context helps prevent underfitting (i.e., false positive matches), over-specializing to large contexts may lead to overfitting (i.e., false negative matches).

We defined these heuristics based on the literature. For instance, Gulwani [42] defines a ranking strategy that favors reusing input data (i.e., SubStr operator) instead of constants (i.e., ConstStr). On the other hand, Nguyen [98] points out that the context helps to create more accurate code refactorings.

4.2 Evaluation

In this section, we present two empirical studies to evaluate REFAZER. First, we present an empirical study on learning transformations for fixing student submissions to introductory Python programming assignments (Section 4.2.1). Then, we present an evaluation of

REFAZER on learning transformations to apply repetitive edits to open-source C# projects (Section 4.3.1).

4.2.1 Fixing introductory programming assignments

In this study, we use REFAZER to learn transformations that describe how students modify an incorrect program to obtain a correct one. We then measure how often the learned transformations can be used to fix other students' incorrect submissions. Transformations that apply across students are valuable because they can be used to generate hints to students on how to fix bugs in their code; alternatively, they can also help Teaching Assistants (TAs) with writing better manual feedback.

Our goal is to investigate both the overall effectiveness of our technique, and to what extent learned transformations in an education scenario are problem-specific, or general in nature. Formally, we have:

Goal : *analyze REFAZER for the purpose of fixing bugs in students' programming assignments with respect to effectiveness and the extent the learned transformations are problem-specific or general in nature from the point of view of teachers and TAs in the context of education scenario.*

If most transformations are general purpose, instructors might be able to provide them manually, once. However, if most transformations are problem-specific, automated techniques such as REFAZER will be especially valuable. Concretely, we address the following research questions:

RQ1 How often can transformations learned from student code edits be used to fix incorrect code of other students in the *same* programming assignment?

Given a programming assignment, students can submit multiple solutions until get all test cases passing. We measure for how many of the students, REFAZER can find transformation to fix the incorrect submission before the student submit the correct one, and how many submissions were required to find the fix for each student.

RQ2 How often can transformations learned from student code edits be used to fix incorrect code of other students who are solving a *different* programming assignment?

Transformations learned for one programming assignment can potentially be useful for another assignment. To answer this research question, we evaluate how often we can fix student solutions by using transformations learned from other assignments.

Experiment units

We collected data from the introductory programming course CS61A at UC Berkeley. As many as 1,500 students enroll in this course per semester, which has led the instructors to adopt solutions common to MOOCs such as autograders. For each homework problem, the teachers provide a black-box test suite and the students use these tests to check the correctness of their programs. The system logs a submission whenever the student runs the provided tests for a homework assignment. This log thus provides a history of all submissions. Our benchmark comprises four assigned problems (see Table 4.1).

The *Product* assignment asks the students to write a product function `product(n, term)` function that returns `term(1) * ... * term(n)`. In the *Accumulate* assignment, students have to write a more general function that works for other kinds of operations besides `product`, `accumulate(combiner, base, n, term)` that returns the result of combining the first `n` terms in a sequence and `base` is the base case. The terms to be combined are `term(1)`, `term(2)`, ..., `term(n)` and `combiner` is a two-argument commutative function. On the other hand, The *Repeated* assignment asks students to implement a function `repeated(f, n)(x)` returns `f(f(...f(x)...))`, where `f` is applied `n` times. `repeated(f, n)` returns another function that can then be applied to another `x` argument. Moreover, the `G(n)` function return $\sum_{i=1}^3 i \cdot G(n - i)$.

For each problem, students had to implement a single function in Python. We filtered the log data to focus on students who had at least one incorrect submission, which is required to learn a transformation from incorrect to correct state. We analyzed 21,781 incorrect submissions from up to 720 students.

Experimental procedure

For each problem, each student in the data set submitted one or more incorrect submissions and, eventually, a correct one. We used the last incorrect submission and the correct one as input-output examples to synthesize a program transformation and used the synthesized transformation to attempt fixing other student submissions. By selecting a pair of incorrect and correct submissions, we learn a transformation that changes the state of the program from

incorrect to correct, fixing existing faults in the code. Students may have applied additional edits, such as refactorings, though. The transformation thus may contain unnecessary rules to fix the code. By learning from the last incorrect submission, we increase the likelihood of learning a transformation that is focused on fixing the existing faults. We left for future work, an investigation of the results when we choose a pair of incorrect and correct submission with the minimum number of edits between them.

The experiments were performed on a PC with a Core i7 processor and 16GB of RAM, running Windows 10.

Research method

For RQ1, we considered two scenarios: *Batch* and *Incremental*. In the *Batch* scenario, for each assignment, we synthesize transformations for all but one student in the data set and use them to fix the incorrect submissions of the remaining student, in a leave-one-out cross-validation – i.e., we attempt to fix the submission of a student using only transformations learned using submissions of other students. This scenario simulates the situation in which instructors have data from one or more previous semesters. In the *Incremental* scenario, we sort our data set by submission time and try to fix a submission using only transformations learned from earlier timestamps. This scenario simulates the situation in which instructors lack previous data. Here the technique effectiveness increases over time. For RQ2, we use all transformations learned in *Batch* from one assignment to attempt to fix the submissions for another assignment. We selected four combinations of assignments for this experiment, and we evaluated the submissions of up to four hundred students in each assignment.

We used the teacher-provided test suites to check whether a program was fixed. Therefore, our technique relies on test cases for evaluating the correctness of fixed programs. While reliance on tests is a fundamental limitation, when fixes are reviewed in an interactive setting, our technique can be used to discover the need for more test cases for particular assignments.

In general, each synthesized rule in the transformation may apply to many locations in the code. In our experiments, we apply each synthesized transformation to at most five hundred combinations of locations. For instance, a transformation could add a single statement to a code location or could perform many edits at multiple code locations of the student programming assignment. If a transformation can be applied to further locations, we abort and proceed to the next transformation.

Table 4.1: Our benchmarks and incorrect student submissions.

	Assignment	Students	Incorrect submissions
Product	product of the first n terms	549	3,218
Accumulate	fold-left of the first n terms	668	6,410
Repeated	function composition, depth n	720	9,924
G	$G(n) = \sum_{i=1}^3 i \cdot G(n - i)$	379	2,229

4.3 Results and discussion

In the *Batch* scenario, REFAZER generated fixes for 87% of the students. While, on average, students took 8.7 submissions to finish the assignment, the transformations learned using REFAZER fixed the student submissions after an average of 5.2 submissions. In the *Incremental* scenario, REFAZER generated fixes for 44% of the students and required, on average, 6.8 submissions to find a fix. The results suggest that the technique can be useful even in the absence of data from previous semesters but using existing data can double its effectiveness. Table 4.2 summarizes the results for both scenarios.

Although we only used students' last incorrect submissions and their corresponding correct submissions as examples for learning transformations, we could find a transformation to fix student solutions 3.5 submissions before the last incorrect submission, on average. This result suggests that REFAZER can be used to provide feedback to help students before they know how to arrive at a correct solution themselves. In addition, providing feedback about mistakes can be more important for students struggled with their assignments. Figure 4.4 shows the 50 students with the highest number of submissions for the two hardest assignments. Each column shows chronological submissions for one student, with the earliest submissions at the top and the eventual correct submission at the bottom. Red indicates an incorrect submission; blue shows the first time REFAZER was able to fix the student's code (we only show the earliest time and do not re-test subsequent incorrect submissions). As we can see in the charts, students took dozens (up to 148) submissions. In many cases, REFAZER provided a fix after the student attempted half of the submissions. For the students where REFAZER was not able to learn a program transformation that fixes their programming assignment, the other students' submissions did not provide hints on how to fix them. It could be caused by student programming style or the student was getting far away from the correct answer.

Table 4.2: Summary of results for RQ1. “Incorrect submissions” = mean (SD) of submissions per student; “Students” = % of students with solution fixed by REFAZER; “Submissions” = mean (SD) of submissions required to find the fix.

Assignment	Incorrect submissions	Batch		Incremental	
		Students	Submissions	Students	Submissions
Product	5.3 (8.2)	501 (91%)	3.57 (6.1)	247 (45%)	4.1 (6.7)
Accumulate	8.9 (10.5)	608 (91%)	5.4 (7.9)	253 (38%)	7.5 (9.8)
Repeated	12.7 (15.3)	580 (81%)	8 (10.3)	340 (47%)	9.6 (11.5)
G	5.5 (9.4)	319 (84%)	1.4 (1.7)	174 (46%)	4.1 (7)
Total	8.7 (12)	2,008 (87%)	5.2 (8.1)	1,014 (44%)	6.8 (9.7)

The transformations learned by REFAZER contain edits with different granularity, ranging from edits to single nodes in the AST, e.g., updating a constant, to edits that add multiple statements, such as adding a base case, a return statement, or even replacing an iterative solution by a recursive one. On average, the tree edit distance between the AST of an incorrect submission and the fixed one was 4.9 ($\sigma = 5.1$). However, REFAZER did learn some larger fixes. The maximum tree edit distance was 45. We left for future work, an investigation on the impact of the number of edits on the usefulness of the program transformation. We also noticed transformations containing multiple rules that represent multiple mistakes in the code.

Most learned transformations are not useful among different programming assignments

Using transformations learned from other assignments we fixed submissions for 7-24% of the students, which suggests that most transformations are problem-specific and not common among different assignments, as shown in Table 4.3. Column Original Assignment shows the assignments where REFAZER learned the transformations, and Target Assignment shows the assignments where the transformations were applied to. Accumulate was the assignment with most fixed submissions. The Accumulate function is a generalization of the Product function, used to learn the transformations, which may be the reason for the higher number of fixed faults. In general, the results suggest that different assignments exhibit different fault patterns; therefore, problem-specific training corpora are needed. This finding also suggests that other automatic grading tools that use a fixed or user-provided fault rubric (e.g., AutoGrader [135]) are not likely to work on arbitrary types of assignments.

Qualitative feedback from teaching assistants

To validate the quality of the learned transformations, we built a user interface that allows one

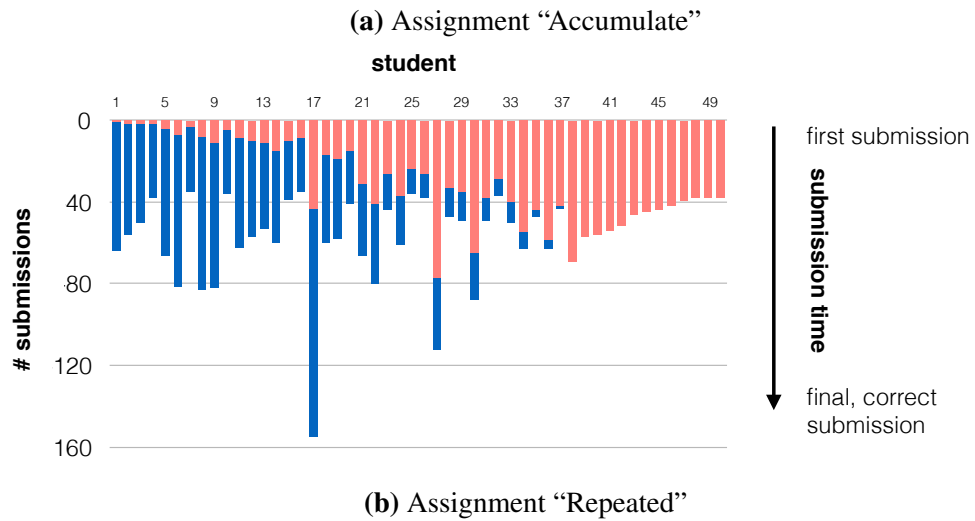
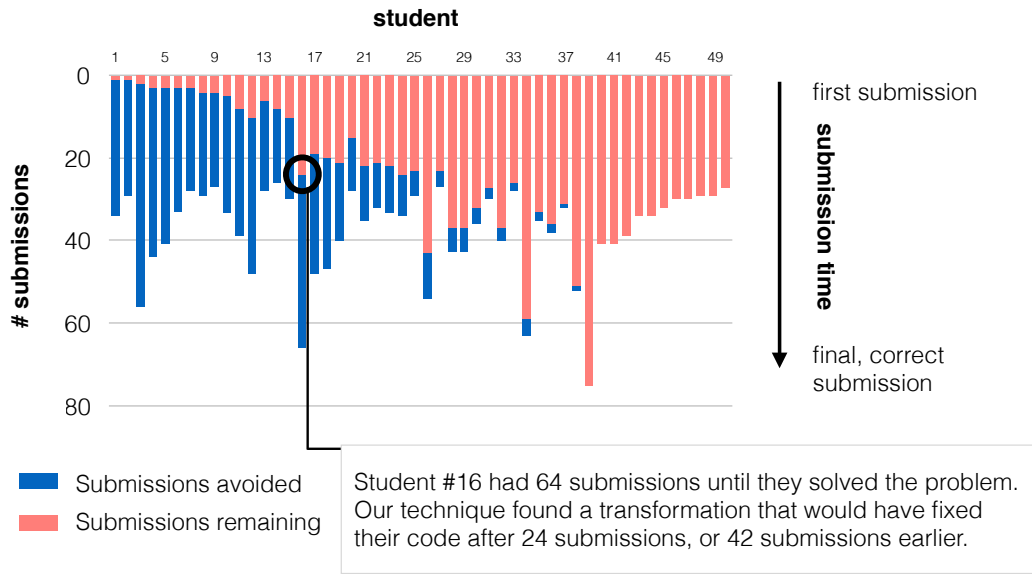


Figure 4.4: Analysis of the first time REFAZER can fix a student submission for the 50 students with most attempts for two benchmark problems. Blue: submissions that might be avoided by showing feedback from a fix generated by REFAZER.

Table 4.3: Summary of results for RQ2.

Original Assignment	Target Assignment	Helped students
Product	G	28 out of 379 (7%)
Product	Accumulate	94 out of 400 (24%)
Product	Repeated	43 out of 400 (11%)
Accumulate	G	33 out of 379 (9%)

to explore, for each transformation, the incorrect submissions that can be fixed with it. We asked a TA of the CS61a course to analyze the fixes found using REFAZER. The TA confirmed that fixes were generally appropriate, but also reported some issues. First, a single syntactic transformation may represent multiple distinct mistakes. For instance, a transformation that changes a literal to 1 was related to a bug in the stopping condition of a while loop in one student’s code; and also to a bug in the initial value of a multiplication which would always result in 0 in another student’s code. In this case, the TA found it hard to provide a meaningful description of the fault beyond “replace 0 with 1”. If fixes are used to generate feedback, TAs will need additional tools to merge or split clusters of student submissions using approaches such as the one performed in MistakeBrowser [46], a mixed-initiative approach, that uses teacher expertise to better leverage the fixes produced by REFAZER. Finally, some fixed programs passed the tests but the TA noticed some faults remained due to missing tests.

4.3.1 Applying repetitive edits to open-source C# projects

In this study, we use REFAZER to learn transformations that describe simple edits that have to be applied to many locations of a C# codebase. We then measure how often the learned transformation is the intended one and whether it is correctly applied to all the required code locations. Formally, we have:

Goal : *analyze REFAZER for the purpose of performing repetitive edits with respect to identify how often the learned transformation is the intended one and whether it is correctly applied to all the required code locations from the point of view of developers in the context of software evolution.*

Concretely, we address the following question:

RQ3 Can REFAZER synthesize transformations with repetitive edits to large open-source projects? We learn transformations based on developers edits in the source code, and we use REFAZER to perform the edits to other locations in the source code. We compare the edits performed by REFAZER with the edits performed by the developers.

Benchmark

We manually inspected 404 revisions from three open-source projects: Roslyn, Entity

Framework, and NuGet. The projects’ sizes range from 150,000 to 1,500,000 lines of code. Starting from the most recent revision, we inspected each revision *diff* files—i.e., the code before and after the edits. If similar edits appeared three or more times into the same commit, we classified the edit as repetitive. We identified 56 scenarios: 27 in Roslyn, 12 in Entity Framework, and 17 in NuGet.

The number of edited locations in each scenario ranges from 3 to 60 (*median* = 5). Each project contains at least one scenario with more than 19 edited locations. In 14 (25%) out of the 56 scenarios, there are edited locations in more than one file, which is harder for developers to handle correctly. Finally, in 39 (70%) out of the 56 scenarios, the edits are complex and context-dependent, meaning that a simple search/replace strategy is not enough to correctly apply the edits to all the necessary locations. For instance, the transformation that replaces the equals methods that use `CSharpKind()` to use `IsKind()` could not be performed with a copy and paste because the edits are not identical, requiring the abstraction and manipulation of some subtrees of the input-output examples. We measured the size of the edits used as input-output examples by calculating the tree edit distance between the input AST and the output AST. On average, the distance was 13.0 ($\sigma = 13.6$). The maximum distance was 76.

Experimental setup

We selected the examples for REFAZER as follows. First we randomly sort the edits described in the diff information. Next we incrementally add them as examples to REFAZER: after each example, we check whether the learned transformation correctly applies subsequent edits. If the transformation misses an edit or incorrectly applies it according to the diff information, we take the first discrepancy in the edit list and provide it as the next example. If the transformation applies edits not presented in the diff, we manually inspect them to check whether the developer missed a location or the locations were incorrectly edited. The experiments were performed on a PC with a Core i7 processor and 16GB of RAM, running Windows 10.

Table 4.4: Evaluation Summary. Scope = scope of the transformation; Ex. = examples; Dev. = locations modified by developers; REFAZER = locations modified by REFAZER. Outcomes: ✓ = it performed the same edits as the developers; ★ = it performed more edits than the developers (manually validated as correct); ✗ = it performed incorrect edits; “—” = it did not synthesize a transformation.

Id	Project	Scope	Ex.	Dev.	REFAZER	Outcome
1	EF	Single file	2	13	13	✓
2	EF	Single file	5	10	10	✓
3	EF	Multiple files	3	19	20	★
4	EF	Single file	3	4	4	✓
5	EF	Single file	3	3	3	✓
6	EF	Single file	2	3	3	✓
7	EF	Single file	2	4	10	★
8	EF	Multiple files	2	8	8	✓
9	EF	Single file	2	3	3	✓
10	EF	Single file	2	12	12	✓
11	EF	Multiple files	4	5	5	✓
12	EF	Single file	2	3	3	✓
13	NuGet	Single file	2	4	4	✓
14	NuGet	Multiple files	2	4	16	✗
15	NuGet	Single file	3	3	3	✓
16	NuGet	Multiple files	2	31	88	★
17	NuGet	Single file	3	3	3	✓
18	NuGet	Multiple files	4	8	14	✗
19	NuGet	Single file	4	14	43	✗
20	NuGet	Single file	2	4	4	✓
21	NuGet	Multiple files	5	5	13	✗
22	NuGet	Single file	3	3	3	✓
23	NuGet	Single file	3	5	5	✓
24	NuGet	Single file	2	3	3	✓
25	NuGet	Single file	3	4	4	✓
26	NuGet	Single file	2	9	32	✗
27	NuGet	Single file	3	4	4	✓
28	NuGet	Multiple files	3	4	10	★
29	NuGet	Multiple files	4	12	79	✗
30	NuGet	Single file	2	3	21	★
31	Roslyn	Multiple files	5	7	7	✓
32	Roslyn	Multiple files	3	17	17	✓
33	Roslyn	Single file	3	6	6	✓
34	Roslyn	Single file	2	9	9	✓
35	Roslyn	Multiple files	3	26	744	★
36	Roslyn	Single file	2	4	4	✓
37	Roslyn	Single file	4	4	4	✓
38	Roslyn	Single file	8	14	14	✓
39	Roslyn	Single file	2	60	—	—
40	Roslyn	Single file	3	8	8	✓
41	Roslyn	Multiple files	3	15	15	✓
42	Roslyn	Single file	2	7	7	✓
43	Roslyn	Single file	5	13	14	✗
44	Roslyn	Single file	2	12	12	✓
45	Roslyn	Single file	2	4	4	✓
46	Roslyn	Single file	2	5	5	✓
47	Roslyn	Single file	2	3	3	✓
48	Roslyn	Single file	4	11	11	✓
49	Roslyn	Single file	2	5	5	✓
50	Roslyn	Single file	2	3	5	★
51	Roslyn	Single file	2	5	5	✓
52	Roslyn	Single file	2	3	3	✓
53	Roslyn	Single file	4	6	6	✓
54	Roslyn	Multiple files	2	15	49	✗
55	Roslyn	Single file	3	4	4	✓

56	Roslyn	Single file	2	4	4	✓
----	--------	-------------	---	---	---	---

Results

Table 4.4 summarizes our results. REFAZER synthesized transformations for 55 out of 56 scenarios. In one case, Refazer is not able to learn the transformation. This occurs because the edits in the diff are not consistent. For the same input, the developers provide distinct outputs. In the case Refazer where not able to learn the transformation. In 40 (71%) scenarios, the synthesized transformations applied the same edits as developers, whereas, in 16 scenarios, the transformations applied more edits than developers. We manually inspected these scenarios, and conclude that seven transformations were correct (i.e., developers missed some edits). We reported them to the developers of the respective projects. As of this writing, they confirmed three of these scenarios. In one of them, developers merged our pull request. In another one, they are not planning to spend time changing the code because the non-changed locations did not cause issues in the tests. We show this transformation on Figure 4.5. In the last scenario, developers confirmed that other locations may need similar edits, but they did not inform us whether they will apply them.

```

1 - c => Assert.Throws<DataException>(() =>
2 -   c().InnerException.ValidateMessage("CommitFailed"),
3 + c => Assert.Throws<DataException>(() =>
4 +   ExtendedSqlAzureExecutionStrategy.ExecuteNew(c) .
5 +   InnerException.ValidateMessage("CommitFailed"),
6
7 - c => Assert.Throws<CommitFailedException>(() =>
8 -   c().ValidateMessage("CommitFailed"),
9 + c => Assert.Throws<CommitFailedException>(() =>
10 +   ExtendedSqlAzureExecutionStrategy.ExecuteNew(c) .
11 +   ValidateMessage("CommitFailed"),

```

Figure 4.5: Issue submitted to NuGet project to use the method (`ExtendedSqlAzureExecutionStrategy.ExecuteNew`) consistently.

In nine scenarios (16%), additional edits were incorrect and revealed two limitations of the current DSL. First, some edits require further analysis to identify locations to apply them. For example, in scenario 18, developers replaced the full name of a type by its simple name. However, in some classes, this edit conflicted with another type name. The second limitation relates to our tree pattern matching. Some examples produced templates that were too general. For example, if two nodes have different numbers of children, we can currently only match

them based on their types. To support this kind of pattern, we plan to include additional predicates in our DSL such as `Contains`, which does not consider the entire list of children, but checks if any of the children match a specific pattern.

Our technique, on average, required 2.9 examples to synthesize all transformations in a diff. The number of required examples may vary based on example selection. Additionally, changes in the ranking functions preferring more general patterns over more restrictive ones can also influence the number of examples.

4.3.2 Threats to validity

Construct validity

With respect to construct validity, our initial baseline is the diff information between revisions. Some repetitive edits may have been performed across multiple revisions, or developers may not have changed all possible code fragments. Therefore, some similar edits in each scenario may be unconsidered. To reduce this threat, we manually inspect REFAZER's additional edits. Additionally, we select the scenarios of repetitive edits manually. This selection may favor our technique. To reduce this bias, two researches analyzed these revisions.

Internal validity

Concerning internal validity, example selection may affect the number of examples needed to perform transformations. To avoid bias, we randomly selected the examples. Additionally, we performed this experiment three times and did not find significant changes in the required number of examples.

External validity

For the education domain evaluation, we use four programming assignments to learn fixes for students' submissions. This number of programming assignments could be small. To reduce this threat, we select assignments that have a large number of students enrolled (up to 720 students). Although related to the same programming assignment, students may solve their programming assignments in many ways, such as recursive or interactive solutions. Thus, these programming assignments may represent a large number of transformations. In addition, we complement these edits with repetitive edit scenarios.

In the open source project evaluation, we select repetitive edits scenarios for three open source projects. This number may not reflect all repetitive edits that developers could perform.

To reduce this bias, we select projects from different domains: compilers, database, and library management. Additionally, our database of repetitive edits is small. To reduce this bias, we select a number of repetitive scenarios consistent with those from the literature [82, 83]. In addition, we include edits from two distinct domains: education and open source projects.

4.4 Answer to the researches questions

RQ1 How often can transformations learned from student code edits be used to fix incorrect code of other students in the *same* programming assignment?

In the *Batch* scenario, REFAZER generated fixes for 87% of the students. On average students used 8.7 submissions to finish the assignment, and REFAZER fixed the student submissions after 5.2 submissions on average. In the *Incremental* scenario, REFAZER generated fixes for 44% of the students. REFAZER required, on average, 6.8 submissions to find a fix.

RQ2 How often can transformations learned from student code edits be used to fix incorrect code of other students who are solving a *different* programming assignment?

Using transformations learned from other assignments, we fixed submissions for 7-24% of the students. This suggests that most transformations are problem-specific and not common among different assignments.

RQ3 Can REFAZER synthesize transformations with repetitive edits to large open-source projects?

REFAZER synthesized transformations for 55 out of 56 scenarios. In 71% of the scenarios, the synthesized transformations applied the same edits as developers. However, in 16 scenarios, the transformations applied more edits than developers. We conclude that seven transformations were correct. We reported them to the developers of the respective projects, and they confirmed three of these scenarios.

4.5 Concluding remarks

We presented REVISAR, a technique for synthesizing syntactic transformations from examples. Given a set of examples consisting of program edits, REVISAR synthesizes a transformation that is consistent with the examples. Our synthesizer builds on the state-of-the-art program synthesis engine PROSE. To enable it, we develop (i) a novel DSL for representing program transformations, (ii) domain-specific constraints for the DSL operators, which reduce the space of search for transformations, and (iii) ranking functions for transformation robustness, based on the structure of the synthesized transformations. We evaluated REVISAR on two applications: synthesizing transformations that describe how students “fix” their programming assignments and synthesizing transformations that apply repetitive edits to large codebases. Our technique learned transformations that automatically fixed the program submissions of 87% of the students participating in a large UC Berkeley class and it learned the transformations necessary to apply the correct code edits for 84% of the repetitive tasks we extracted from three large code repositories.

Currently, REFAZER learns transformations for C# and Python, but it can be extended to support other programming languages by changing the language compiler/interpreter. We left for future work an investigation of how it can be adapted to other programming paradigms such as logic and functional.

Chapter 5

REVISAR

We now describe REVISAR, our technique for learning quick fixes from code repositories. Given repositories as input, REVISAR: (i) Identifies concrete code edits by comparing pairs of consecutive revisions (Section 5.1), and (ii) Clusters edits into sets that can be described using the same transformation (Section 5.2 and 5.3), Figure 5.1 shows the work-flow of REVISAR. Before describing the details of REVISAR, we clarify that, since we are interested in detecting transformations that are common between different projects, in this thesis, we only focus on individual transformations—i.e., we do not capture multiple patterns that may depend on each other. Individual transformations are also the most common cases of quick fixes in code analyzers. Our infrastructure can be extended to accommodate more complex transformations in the future.

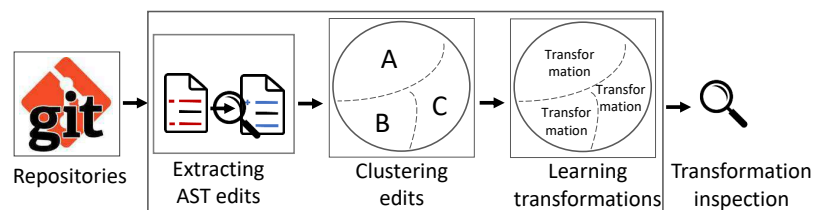


Figure 5.1: REVISAR’s work-flow.

5.1 Extracting concrete AST edits

The initial input of REVISAR is a set $revs = \{R_1, \dots, R_n\}$ where each R_i is a revision history $r_1 r_2 \dots r_k$ from a different project (i.e., a sequence of revisions). For each pair (r_i, r_{i+1}) of

consecutive revisions, REVISAR analyzes the differences between the Abstract Syntax Trees (ASTs)¹ of r_i and r_{i+1} and uses a Tree Edit Distance (TED) algorithm to identify a set of tree edits $\{e_1, e_2, \dots, e_l\}$ that, when applied to the AST t_i corresponding to r_i yields the AST t_{i+1} corresponding to r_{i+1} . In our setting, an edit is one of the following:

insert(x, p, k): insert a leaf node x as k^{th} child of parent p . The current children at positions $\geq k$ are shifted one position to the right.

delete(x): delete a leaf node x from source tree. The deletion may cause new nodes to become leaves when all their children are deleted. Therefore we can delete a whole tree through repeated bottom-up deletions.

update(x, w): replace a leaf node x by a leaf node w .

move(x, p, k): move tree x to be the k^{th} child of parent p . The current children at positions $\geq k$ are shifted one position to the right.

Given a tree t , let $s(t)$ be the set of nodes in the tree t . Intuitively, solving the TED problem amounts to identifying a partial mapping $M : s(t) \mapsto s(t')$ between source tree t and target tree t' nodes. The mapping can then be used to detect which nodes are preserved by the edit and in which positions they appear. When a node $n \in s(t)$ is not mapped to any $n' \in s(t')$, then n was deleted.

Of the many tools that are available for computing tree edits over Java source code, REVISAR builds on GumTree [31], a tool that focuses on finding edits that are representative of those intended by programmers instead of just finding the smallest possible set of tree edits. To give an example of edits computed by GumTree, let's look at the first two lines in Figure 1.3. Figure 5.2 illustrates the ASTs corresponding to `args[i].equals("--launchdiag")` and `--launchdiag.equals(args[i])`, respectively.



Figure 5.2: Before-after version for the first edited line of code.

¹REVISAR uses Eclipse JDT [25] to extract partial type annotations of the ASTs. In our implementation, we use these type annotations to create a richer AST with type information—i.e., every node has a child describing its type. For simplicity, we omit this detail in the rest of the section.

To produce the modified version of the code in line 2 at Figure 1.3, GumTree learned four edits:

```
insert("-launchdiag", MethodInvo, 0)

insert(equals, MethodInvo, 1)

delete(equals)

delete("-launchdiag")
```

Basically, these edits move the string literal `"-launchdiag"` and the `equals` node so that they appear in front of `args[i]`. When run on the code in Figure 1.3, GumTree also learns edits for the other lines of code.

For our purposes, the edits computed by GumTree are at too low granularity since they modify nodes instead of expressions. Concretely, we are interested in detecting edits to entire subtrees—e.g., an edit to a method invocation `args[i].equals("-launchdiag")` instead of individual nodes inside it. REVISAR identifies subtree-level edits by grouping edits that belong to the same connected components. Concretely, REVISAR identifies connected components by analyzing the parent-child and siblings relationships between the nodes appearing in the tree edits. Two edits e_1 and e_2 belong to the same component if (i) the two nodes x_1 and x_2 modified by e_1 and e_2 have the same parent, or (ii) the x_1 (resp. x_2) is the parent of x_2 (resp. x_1). For instance, the previously shown edits are associated to two nodes $x_1 = \text{"-launchdiag"}$ and $x_2 = \text{equals}$. These nodes are connected since they have the same parent node—i.e., the method invocation.

Once the connected components are identified, REVISAR can use them to identify tree-to-tree mappings between subtrees inside the original and modified trees. We call this mapping a *concrete edit*. A concrete edit is a pair (i, o) consisting of two components (i) the tree i in the original version of the program, and (ii) the tree o in the modified version of the program. This last step completes the first phase of our algorithm, which, given a set of revisions $\{R_1, \dots, R_n\}$, outputs a set of concrete edits $\{(i_1, o_1), \dots, (i_k, o_k)\}$.

5.2 Learning transformations

Once REVISAR has identified concrete edits—i.e., pairs of trees $\{(i_1, o_1) \dots (i_n, o_n)\}$ —it tries to group “similar” concrete edits and generate a transformation consistent with all the edits in each group. An *transformation* is a rule $r = \tau_i \mapsto \tau_o$ with two components: (i) the template τ_i , which is used to decide whether a subtree t in the code can be transformed using the rule r , (ii) the template τ_o , which describes how the tree matching τ_i should be transformed by r . In this section, we focus on computing r from a set of concrete edits $\{(i_1, o_1) \dots (i_n, o_n)\}$. We assume that we are already given a group of concrete edits that can be described using the same transformation. We will discuss our clustering algorithm for creating the groups in the next section.

A template τ is an AST where leaves can also be holes (variables) and a tree t matches the template τ if there exists a way to assign concrete values to the holes in τ and obtain t —denoted $t \in L(\tau)$. Given a template τ over a set of holes H , we use α to denote a substitution from H to concrete trees and $\alpha(\tau)$ to denote the application of the substitution α to the holes in τ . Figure 5.3 shows the first two concrete edits from Figure 1.3 and the templates τ_i and τ_o describing the transformation obtained from these examples. Here, the template τ_i matches any expression calling the method `equals` with first argument `args[i]` and any possible second argument. The two substitutions $\alpha_1 = \{?_1 = \text{"-launchdiag"}\}$ and $\alpha_2 = \{?_1 = \text{"-noclasspath"}\}$ yield the expressions $\alpha_1(\tau_i) = \text{args[i].equals("-launchdiag")}$ and $\alpha_2(\tau_i) = \text{args[i].equals("-noclasspath")}$, respectively. The template τ_o is similar to τ_i and note that the hole $?_1$ appearing in τ_o is the same as the one appearing in τ_i .

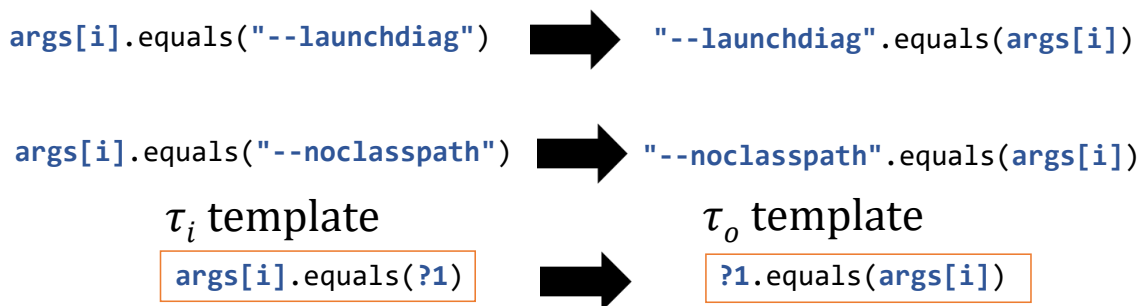


Figure 5.3: Concrete edits and their input-output templates.

In the rest of the section, we describe how we obtain the transformation from the concrete edits. We start by describing how the input template τ_i is computed. Our goal is to compute a template τ_i such that every AST i_j can match the template τ_i —i.e., $i_j \in L(\tau_i)$. In general, the same set of ASTs can be matched by multiple different templates, which could contain different numbers of nodes and holes. Typically, a template with more concrete nodes and fewer holes is more precise—i.e., will match fewer concrete ASTs—whereas a template with few concrete nodes will be more general—e.g., `?1.equals(?2)` is more general than `arg[i].equals(?1)`. Among the possible templates, we want the *least general template*, which preserves the maximum common nodes for a given set of trees. The idea is to preserve the maximum amount of shared information between the concrete edits. Even when an edit is too specific, we will obtain the desired template when provided with appropriate concrete edits. In our running example, when encountering an expression of the form `x.equals("abc")`, we will obtain the desired, more general template `?1.equals(?2)`.

Remarkably, the problem we just described is tightly related to the notion of anti-unification often used in logic programming [5]. Given two trees t_1 and t_2 , the anti-unification algorithm produces the least general template τ for which there exist substitutions α_1 and α_2 such that $\alpha_1(\tau) = t_1$ and $\alpha_2(\tau) = t_2$. Among anti-unification algorithm, we use Baumgartner and Kutsia [5], which runs in $O(n^2)$. Using this algorithm, we can generate the least general templates τ_i and τ_o that are consistent with the input and output trees in the concrete edits. For now, the two templates will have distinct sets of holes.

At this point, we have the template for the inputs τ_i and the template for the output τ_o . However, REVISAR needs to analyze whether these templates describe an edit pattern $r = \tau_i \mapsto \tau_o$ —i.e., whether there is a way to map the holes of τ_i to the ones of τ_o . For instance, let's look at Figure 5.4. Although we can learn templates τ_i and τ_o , these templates cannot describe an edit pattern since it is impossible to come up with a mapping between the holes of the two templates that is consistent with all the substitutions—i.e. in this case, the substitution for `?1` in τ_i is incompatible with the substitution for `?2` in τ_o (i.e., `"-noclasspath"` is mapped to `"-main"`, but the content of these substituting trees differs).

Definition 1. Given a set of concrete edits $S = \{(i_1, o_1), \dots, (i_n, o_n)\}$, we say that a transformation $r = \tau_i \mapsto \tau_o$, is consistent with S if: (i) the set of holes in τ_o is a subset of the set the

holes appearing in τ_i (ii) for every (i_k, o_k) there exists a substitution α_k such that $\alpha_k(\tau_i) = i_k$ and $\alpha_k(\tau_o) = o_k$.

To illustrate the definition, the concrete edits in Figure 5.3 form a valid transformation because the hole $?_1$ maps to the same concrete nodes with the same value in τ_i and τ_o . Therefore, computing a transformation simply boils down to finding a mapping between the holes in the templates τ_i and τ_o obtained from the anti-unification algorithm. This mapping can be computed by finding, for every hole $?_2$ in τ_o , a hole $?_1$ in τ_i that applies the same substitution with respect to all the concrete edits. Since the templates are the least general, if no such mapping exists, there exists no transformation consistent with the concrete edits.

Theorem 5.2.1 (Soundness and Completeness). Given a set of concrete edits $S = \{(i_1, o_1), \dots, (i_n, o_n)\}$, REVISAR returns a transformation $r = \tau_i \mapsto \tau_o$ consistent with S if and only if some transformation $r' = \tau'_i \mapsto \tau'_o$ consistent with S exists.

Proof. By construction, REVISAR only returns transformations consistent with S . To prove completeness, we show that if there exists a transformation r' consistent with S , REVISAR will also find some transformation r consistent with S . Let $r' = \tau'_i \mapsto \tau'_o$ be a transformation consistent with S and let $\{\alpha_1, \dots, \alpha_n\}$ be the substitution for the two templates. The template τ'_i (resp. τ'_o) has to be more general than τ_i (resp. τ_o) or equal to it because a less general template would not be consistent with the examples due to the properties of anti-unification. Let $?_j$ be a hole in τ'_i and let $t_j = \alpha_j(?_j)$ be the tree assigned by the j -th substitution for every $j \leq n$. Let $t(?_1, \dots, ?_k)$ be the tree obtained by anti-unifying all the t_j , where the tree contains holes $?_1, \dots, ?_k$ as leaves. If $t(?_1, \dots, ?_k)$ is just a single hole, the hole $?$ will also appear in the same position in the template obtained from REVISAR through anti-unification (i.e., this hole position is the least general). If $t(?_1, \dots, ?_k)$ contains concrete nodes (not holes), we can replace $?$ in τ'_i with $t(?_1, \dots, ?_k)$ and obtain a template consistent with S . Similarly, we can change or substitute to account for the new introduced holes. Now we have new substitutions α'_j for all the new holes in the modified template and can do the same for τ'_o . To complete our proof we need to show that there exists a mapping between the holes in the two new templates. If a hole $?$ appearing in both input and output templates was expanded to $t(?_1, \dots, ?_k)$, the mapping obtained from the new extended substitutions will still be a correct one. This concludes the proof. \square

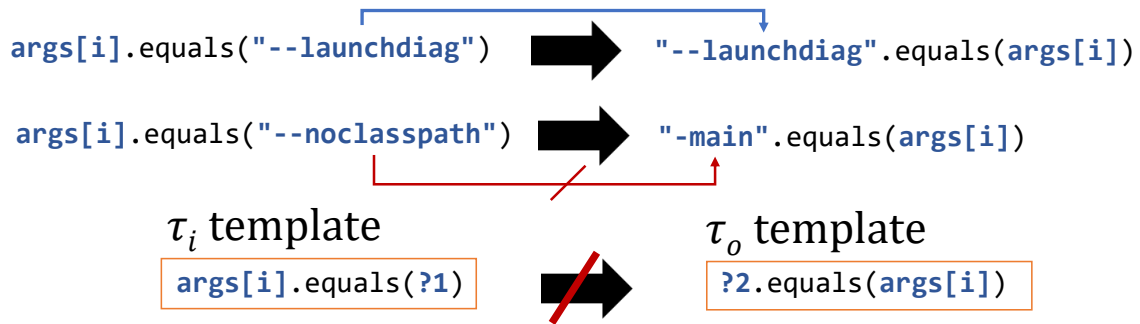


Figure 5.4: Incompatible concrete edits.

Producing executable transformations Refaster [148] is a subset of the code analyzer ErrorProne [39] from Google for describing simple transformations for Java expressions. When permitted by the syntax of Refaster, REVISAR automatically compiles transformations to Refaster rules, which are user-readable and can be executed in ErrorProne. A Refaster transformation is described using (i) a before template to pattern-match target locations, and (ii) an after template to specify how these locations are transformed. More practically, a Refaster transformation is a compilable class with multiple methods. All methods contain the same return type and list of arguments. One method is annotated with `@AfterTemplate` and the other methods are annotated with `@BeforeTemplate`. For each transformation $r = \tau_i \mapsto \tau_o$ generated by REVISAR, we use the template τ_i to generate the `@BeforeTemplate` and the output τ_o to generate an `@AfterTemplate`. The holes in a template are represented as Java variables and are added as parameters for the before and after methods. As Refaster uses Java instead of a domain-specific language, rules are easy to understand for any Java programmer.

Like any tool, Refaster has its limitations. In particular, it cannot describe transformations that require knowing the AST type of a node. For instance, the transformation Figure 1.4 requires knowing that an AST node is a literal besides knowing that its type is `String`. This edit cannot be represented exactly in Refaster. In addition, Refaster can only modify expressions that appear inside a method body and it is not suitable to express fixes to expressions outside a method body (e.g., global field declarations.).

```

Input: Set of templates  $\{(i_1, o_1) \dots (i_n, o_n)\}$ 
1: clusters =  $\emptyset$ 
2: for all  $(i_k, o_k)$  in  $\{(i_1, o_1) \dots (i_n, o_n)\}$  do
3:   cluster_candidates =  $\emptyset$ 
4:   for all cluster  $c$  in clusters do
5:      $r = \tau_i \mapsto \tau_o := \text{compute\_template}((i_k, o_k), c)$ 
6:     if  $r$  is a transformation then
7:       cluster_candidates.add( $c$ )
8:        $cost_c := \text{add\_cost}((i_k, o_k), c)$ 
9:     best = argmin(cluster_candidates)
10:  if a best cluster exists then
11:    best.add( $(i_k, o_k)$ )
12:  else
13:    clusters.append(new cluster( $(i_k, o_k)$ ))
14: return clusters

```

Algorithm 20: Greedy clustering.

5.3 Clustering concrete edits

In this section, we show how REVISAR groups concrete edits into clusters that share the same transformation. REVISAR’s clustering algorithm receives a set of concrete edits $\{(i_1, o_1) \dots (i_n, o_n)\}$ and uses a greedy algorithm (Algorithm 20). The clustering algorithm starts with an empty set of clusters (Line 1). Then, for each concrete edit (i_k, o_k) and for each cluster c , REVISAR checks, using the algorithm from Section 5.2, if adding (i_k, o_k) to the concrete edits of cluster c gives a transformation. When this happens, the cluster c is added to a set of cluster candidates and the cost of adding (i_k, o_k) to c is computed (Lines 2–8). REVISAR then adds (i_k, o_k) to candidate cluster of minimum cost, or it creates a new cluster with just (i_k, o_k) if no candidate exists (Lines 9–13). The cost of this algorithm is $O(n^2)$ since we have to perform a linear search for all the concrete edits, and another linear search for each cluster to identify in which one each concrete edit should be included. The number of clusters is always less than the number of concrete edits.

The cost of adding an edit (i_k, o_k) to cluster c is computed as follows. Let τ be the template for the cluster c . To compute the cost, REVISAR anti-unifies τ and the tree in the concrete edit we are trying to cluster. Let α_k be the substitution for the result of the anti-unification and let $\alpha_k(?_i)$ be the tree substituting hole $?_i$. We define the size of a tree as the number of leaf nodes inside it, which intuitively captures the number of names and constants in the AST. We denote the cost of an anti-unification as the sum of the sizes

of each $\alpha_k(?_i)$ minus the total number of holes (the same metric proposed by Bulychev et al. [12] in the context of clone detection). Intuitively, we want the sizes of substitutions to be small—i.e., we prefer more specific templates. For instance, assume we have a cluster consisting of a single tree `args[i].equals("-launchdiag")`. Upon receiving a new tree `args[i].equals("-noclasspath")`, anti-unifying the two trees yields the template `args[i].equals(?_1)` with substitutions $\alpha_1=\{?_1="-launchdiag"\}$ and $\alpha_2=\{?_1="-noclasspath"\}$. We have concrete nodes `"-launchdiag"` and `"-noclasspath"` each of size one, and for a single hole $?_1$. The cost will be $2 - 1 = 1$. The final cost of adding an edit to a cluster is the sum of the cost to anti-unify τ_i and i_k and the cost to anti-unify τ_o and o_k .

Predicting promising clusters When analyzing large repositories, the total number of concrete edits may be very large and it will be unfeasible to compare all the possible edits to compute the clusters. To address this problem, REVISAR only cluster concrete edits which are “likely” to produce a transformation. In particular, REVISAR uses *d-caps* [30, 63, 96], an effective technique for identifying repetitive edits. Given a number $d \geq 1$ and a template τ , a *d-cap* is a tree-like structure obtained by replacing all subtrees of depth d and left nodes in the template τ with holes. The *d-cap* works as a hash-index for sets of potential clusters. For instance, let’s look at the left-hand side of Figure 5.2, which is the tree representation of the node `args[i].equals("-launchdiag")`. A 1-cap replaces all the nodes at depth one with holes. For our example, `args[i]`, `equals`, and `"-launchdiag"` will be replaced with holes, outputting the following *d-cap* $?_1 \cdot ?_2 (?_3)$. REVISAR uses the *d-caps* as a pre-step in Algorithm 20. For all concrete edits that have the same *d-cap* for the input tree i_k and for the output tree o_k , REVISAR uses the clustering algorithm described in Section 5.3 to compute the clusters for all concrete edits in these *d-cap*. This heuristic makes our clustering algorithm practical as it avoids considering all possible example combinations. However, it also comes at the cost of sacrificing completeness—i.e., two concrete edits for which there exists a common transformation might be placed in different clusters. We will further discuss this limitation in the rest of the thesis.

5.4 Effectiveness of REVISAR

In this section, we evaluate how effective REVISAR is at learning edit patterns from software repositories.

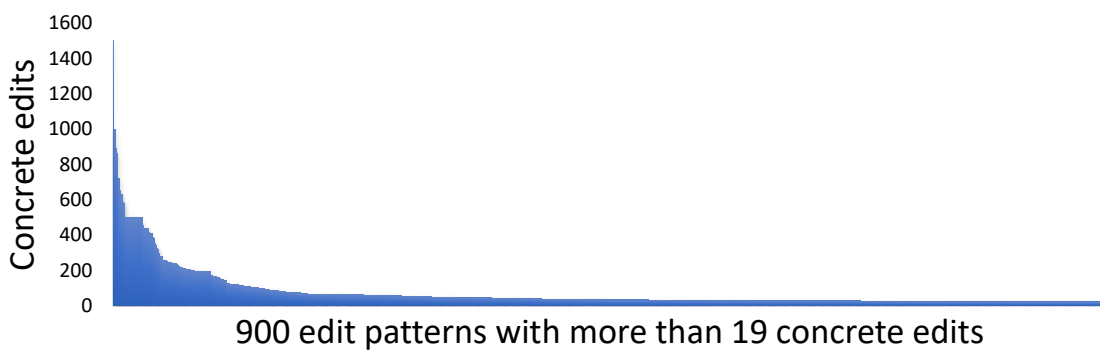
Benchmarks and Results We selected 9 popular GitHub Java projects (Table 5.1) from a list of previously studied projects [133]. The project selection influences quantity and quality of discovered patterns. We select mature popular projects that have long history of edits, experienced developers who detect problems during code reviews, and several collaborators (avg. 89.11) with different levels of expertise (low expertise collaborators likely submit pull-requests that need quick fixes). Analyzing many projects is prohibitive given our resources, thus we selected a subset of projects with various sizes/domains. We favored projects containing between 1,000 and 15,000 revisions, so that we had a sample large enough to identify many patterns but not too big due to the time required to evaluate all revisions. The projects have size ranging from 24,753 to 1,119,579 lines of code. In total, the sample contains 43,113 revisions. We performed the experiments on a PC running Windows 10 with a processor Core i7 and 16GB of RAM. We obtained the revision histories of each repository using JGit [52]. As described in § 5, REVISAR uses GumTree to compute concrete edits. REVISAR took 5 days to analyze the 9 projects (approximately 10 seconds per revision). This time is dominated by the time to checkout revisions, a process that can be done incrementally for future projects.

It took us about three days to manually analyze the 381 edit patterns. For each pattern, we checked whether it was useful, and if it was already known through online resources. Since we focused on small quick fixes, checking whether it was a repeated pattern was usually fast. Checking usefulness was also fast since the rationales of the patterns were well described on the online documentation. Regarding packaging, when patterns were supported by the Refaster syntax, integrating a pattern in Refaster took 10-30 minutes. 17 edits could not be integrated due to the limitations of Refaster.

REVISAR found 240,612 edits which were clustered in 105,705 clusters. 28,842 of these clusters contained more than one edit, which allowed REVISAR to generalize the examples to an edit pattern. The chart in Figure 5.5 shows the number of edits per pattern for edit patterns containing 20 or more edits. The most common pattern has 1,503 concrete edits.

Table 5.1: Projects used to detect edit patterns.

Projects	Edits	LOCs	Revisions
Hive	94,921	1,119,579	11,467
Ant	49,680	137,203	13,790
Guava	28,784	325,902	4,633
Drill	26,173	350,756	2,902
ExoPlayer	20,726	85,305	3,875
Giraph	8,836	99,274	1,062
Gson	4,435	24,753	1,393
Truth	3,857	27,427	1,137
ErrorProne	3,200	116,023	2,854
Total:	240,612	2,286,222	43,113

**Figure 5.5:** Distribution of concrete edits per edit pattern.

REVISAR identified 28,842 common edit patterns in 43,113 revisions from 9 GitHub Java projects.

Discussion We manually investigated some of the learned patterns to analyze on how well REVISAR generalized concrete edits. Many patterns indeed describe logical edits made by programmers (further discussed in the next section). However, we also found instances of overly specific patterns. First, although the *d-cap* analysis helped predicting clusters, we observed that some clusters that were deemed different due to the *d-cap* analysis could still be merged together to yield a single edit pattern. As a result, some edit patterns were over-specific. For instance, in the `StringBuffer` to `StringBuilder` quick fix, both the left-hand side and the right-hand side could not become a hole without losing the edit pattern. We also found multiple patterns that represented similar conceptual patterns. As expected, we did not observe overly general patterns, which were prevented by the anti-unification algorithm. Finally, we inspected edits that REVISAR could not generalize to an edit pattern. From the sample we analyzed, we observed these edits either required a more complex edit-pattern format than the one we consider in this thesis or were too specific for the context where they were applied, the latter being the more common case.

5.5 Usefulness of the edit patterns learned by REVISAR

In this section, we present a mixed study to evaluate how programmers perceive the edits learned by REVISAR. In particular, we focus on the following research questions:

RQ1: Can REVISAR be used to discover common quick fixes?

RQ2: Are developers interested in applying quick fixes discovered by REVISAR?

First, discovering common new quick fixes will help in extending the catalog of quick fixes in existing code analyzers. Second, understanding the usefulness of common quick fixes that are already present in existing code analyzers can help the programmers of these tools prioritize their work as well as motivate programmers in adopting such tools. In § 5.5.1, we describe our study to identify the most common edits performed by programmers according

to REVISAR and then present two studies in Sections 5.5.2 and 5.5.3 to answer our second research question.

5.5.1 Catalog of quick fixes

In this experiment, we acted as developers of code analyzers and inspected the edit patterns learned by REVISAR to find which ones were quick fixes. So far, of the patterns learned by REVISAR, we manually inspected the top-381 ones that occurred across most projects since they were more likely to be of interest to general programmers. We performed this analysis to identify edits that could be helpful for general programmers, but our technique also could be used to train new programmers to follow the code standard of a particular company. For instance, by investigating the quick fixes for a particular project/company. These clusters cover more than 8,173 concrete edits (i.e., our patterns cluster many concrete edits, thus simplifying their analysis). We only analyzed 381 out of 28,000+ due to time constraints. To classify an edit pattern as a quick fix, we analyzed whether the edit pattern had a clear motivation and these motivation could be of interest for general developers. Since REVISAR does not learn dependencies between edit patterns, some of the single edit patterns did not have a clear meaning because they were part of a larger edit pattern. Additionally, we found many renaming operations (e.g., renaming `obj` to `object`), which improve readability, but are specific to certain variable names. Since these last patterns are not applicable to general programs we did not include them in our catalog. Finally, we merged patterns that represent the same quick fix. At the time of writing, we derived a catalog of 32 quick fixes from the edit patterns discovered by REVISAR. For 15/32 quick fixes, REVISAR was also able to generate an executable `ErrorProne` rule. Next we present further details for the 10 patterns we used in our survey. 9/32 fixes are covered by existing tools.

1. *String to character* In Java, we can represent a character both as a `String` or a `character`. For operations such as appending a value to a `StringBuffer`, representing the value as a `character` improves performance (see Quick Fix 1). This edit improved the performance of some operations in the Guava project by 10-25% [122].

2. *Prefer string literal equals method* (discussed Section 1) Invoking `equals` on a `null` variable causes a `NullPointerException` When comparing the value in a string variable

Table 5.2: quick fixes. Number of projects associated to each quick fix. Number of edits that generate the edit pattern corresponding to the quick fix and number of revisions associated to the edit pattern.

Quick Fix	Projects	Edits	revisions
01	3	61	9
02	3	78	7
03	3	17	4
04	2	3	2
05	3	11	5
06	3	75	7
07	7	172	10
08	3	102	5
09	7	863	90
10	4	15	4
11	3	10	3
12	6	45	8
13	3	21	4
14	4	37	11
15	4	4	4
16	8	171	29
17	4	24	4
18	3	7	4
19	2	3	2
20	5	145	121
21	4	10	8
22	2	3	3
23	5	44	11
24	2	4	3
25	7	237	100
26	2	2	2
27	3	22	10
28	3	7	5
29	6	18	10
30	4	18	9
31	3	12	4
32	2	2	2
Average:	3,84	70,10	15,62

<pre> 1 StringBuffer sb = 2 new StringBuffer(); 3 sb.append("a"); </pre>	→	<pre> 1 StringBuffer sb = 2 new StringBuffer(); 3 sb.append('a'); </pre>
--	---	--

Quick Fix 1: String to character.

to a string literal, programmers can overcome this exception by invoking the `equals` method on the string literal since the `String` `equals` method checks whether the parameter is `null`. Quick Fix 2 shows an edit to use the `equals` method of a string literal.

<pre> 1 void bar(String s) { 2 if (s.equals("str")) { 3 //... </pre>	➔	<pre> 1 void bar(String s) { 2 if ("str".equals(s)) { 3 // ... </pre>
--	---	---

Quick Fix 2: Prefer string literal `equals` method.

3. **Avoid `FileInputStream` and `FileOutputStream`** These classes override the `finalize` method. As a result, their objects are only cleaned when the garbage collector performs a sweep [23]. Since Java 7, programmers can use `Files.newInputStream/Files.newOutputStream` to improve performance as recommended in this Java JDK bug-report [49]. Quick Fix 3 shows an edit to use `Files.newOutputStream`.

<pre> 1 void bar(String p, 2 byte[] content) 3 throws IOException { 4 FileOutputStream os = 5 new FileOutputStream(p); 6 //... </pre>	➔	<pre> 1 void bar(String p, 2 byte[] content) 3 throws IOException { 4 OutputStream os=Files. 5 newOutputStream(Paths.get(p)); 6 //... </pre>
---	---	--

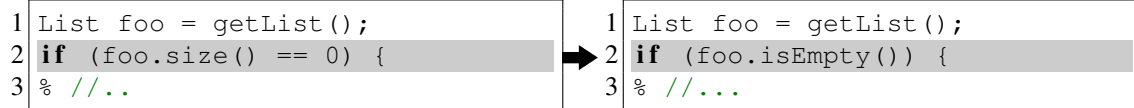
Quick Fix 3: Avoid `FileInputStream/FileOutputStream`.

4. **Use `valueOf` instead wrapper constructor** When creating a new `Integer`, one can use the `valueOf` method or the constructor. The method `valueOf` can improve performance since it caches frequently requested values [106] (Quick Fix 4). This edit pattern is included in the Sonar catalog of rules. The same edit also applies for types such as `Byte`, `Short`, and `Long`, etc.

<pre> 1 Integer a = 2 new Integer(1); </pre>	➔	<pre> 1 Integer a = 2 Integer.valueOf(1); </pre>
--	---	--

Quick Fix 4: Use `valueOf` instead wrapper constructor.

5. **Use collection `isEmpty` method** Using the method `isEmpty` to check whether a collection is empty is preferred to checking that the size of the collection is 0. For most collections these constructions are equivalent, but for others computing the size could be expensive. For instance, in the class `ConcurrentSkipListSet`, the `size` method is not constant-time [105]. This edit pattern is included in the PMD catalog of rules (Quick Fix 5).



```

1 List foo = getList();
2 if (foo.size() == 0) {
3 % //...

```

→

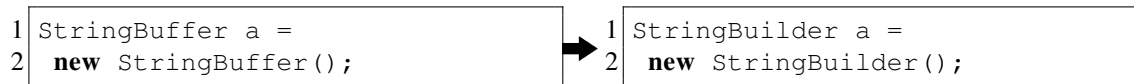
```

1 List foo = getList();
2 if (foo.isEmpty()) {
3 % //...

```

Quick Fix 5: Use collection `isEmpty` method.

6. *StringBuffer to StringBuilder* These classes have the same API, but `StringBuilder` is not synchronized. Since synchronization is rarely used [107], `StringBuilder` offers high performance and is designed to replace `StringBuffer` in single threaded contexts [107]. Since this edit pattern requires knowing whether the applications uses synchronization, it does not appear in any code analyzer. Quick Fix 6 shows an edit to use `StringBuilder`.



```

1 StringBuffer a =
2 new StringBuffer();

```

→

```

1 StringBuilder a =
2 new StringBuilder();

```

Quick Fix 6: `StringBuffer` to `StringBuilder`.

7. *Infer type in generic instance creation* Since Java 7, programmers can replace type parameters to invoke the constructor of a generic class with an empty set (`<>`), diamond operator [108] and allow inference of type parameters by the context. This edit ensures the use of generic instead of the deprecated raw types [109]. The benefit of the diamond operator is clarity since it is more concise(Quick Fix 7).



```

1 List<String> a = new
2 ArrayList<String>();

```

→

```

1 List<String> a = new
2 ArrayList<>();

```

Quick Fix 7: Infer type in generic instance creation.

8. *Remove raw type* A raw type is a generic type without type parameters, which was used in version of Java prior to 5.0 and is allowed to ensure compatibility with pre-generics code. Since type parameters of raw types are unchecked, the catch of unsafe code is deferred to runtime [109] and the Java compiler issues warnings for them [109]. Quick Fix 8 shows an edit that removes a raw type.



```

1 List<String> a = new
2 ArrayList();

```

→

```

1 List<String> a = new
2 ArrayList<>();

```

Quick Fix 8: Remove raw type.

9. Field, parameter, and variable could be final The `final` modifier can be used in fields, parameters, and local variables to indicate they cannot be re-assigned [123]. This edit improves clarity and it helps with debugging since it shows what values will change at runtime. In addition, it allows the compiler and virtual machine to optimize the code [123]. The edit pattern that adds the `final` modifier is included in PMD catalog of rules [120]. IDEs such as Eclipse [35] and NetBeans [48] can be configured to perform this edit automatically on saving. Quick Fix 9 adds the `final` modifier to a local variable.

```
1 String a = "a"; → 1 final String a = "a";
```

Quick Fix 9: Field, parameter, and variable could be final.

10. Avoid using strings to represent paths Programmers sometimes use `String` to represent a file system path even though some classes are specifically designed for this task—e.g., `java.nio.Path`. In this cases, it is useful to change the type of the variable to `Path`. First, strings can be combined in an undisciplined way, which can lead to invalid paths. Second, different operating systems use different file separators, which can cause bugs. Since detecting this pattern requires a non-trivial analysis, code analyzers do not include it in their rules. Quick Fix 10 shows an edit to use a specific class to represent a path.

```
1 private String path; → 1 private Path path;
```

Quick Fix 10: Avoid using strings to represent paths.

5.5.2 A survey of programmers' opinion about our catalog

We conducted a survey to assess programmers' opinions about the catalog of the edit patterns presented in Section 5.5.1. We only evaluated 10/32 in our survey to keep the survey small and increase the completion rate. We believe there are many other useful patterns in the set and we plan to build visualization tools to help us browse them.

Survey planning To recruit participants, we sent e-mails to 2,000 programmers randomly selected from 124 popular GitHub Java projects, including projects from Google, Facebook and Apache Foundation. We divided the survey into two main sections. The first section asks

one question for each of the 10 edit patterns. Each question shows a code pattern on the left-hand side (A) and one on the right-hand side (B). One of the two (either (A) or (B)) is the good and the other the bad code pattern. The 10 edits include both edit patterns missed and included in code analyzer tools. In this way, we can verify aspects such as whether programmers that use code analyzer tools that include a pattern still choose the bad version of it. For each question, we asked for a preference using a Likert Scale with options: strongly prefer (A), prefer (A), it does not matter, prefer (B), and strongly prefer (B). We used text boxes after each question to allow programmers to explain their choices. All programmers answered the same questions in the same order. The second section asks general information: (i) How many of the code patterns shown in the first section have you used in your code before? (ii) How would you rate your knowledge of the effects caused by using the different code patterns? and (iii) How long have you been working/have worked with Java? Furthermore, we ask programmers what code analyzers they are familiar with among Checkstyle, PMD, FindBugs, Coverity, ErrorProne, and IntelliJ IDE; we added a text box for other tools.

Quantitative analysis Overall, 164 (8.2%) programmers completed the survey. Most of them (72%) have more than five years of experience, and we refer to these programmers as *experts*. In addition, 75.5% of programmers in general used 6 or more of the code patterns in their code before. Programmers also indicate a high or very high knowledge of the effects caused by different code patterns in 72.4% of the cases. We refer to these programmers as *informed programmers*.

The results of the survey are shown in Figure 5.6. For all edit patterns, the difference between the preferences (sum of prefer and strongly prefer) for the two patterns is statistically significant using Wilcoxon signed-rank test with 95% significance (p-value = 0.05). For nine edit patterns, programmers prefer or strongly prefer the good code pattern suggested by our edit pattern (90%). The only exception is the pattern *String to character*, for which programmers tend to prefer the `String` to the `character` version.

For most quick fixes (90%), programmers agree with the suggested edits.

When looking only at experts' answers, we observe the same results for all patterns but the first one, for which we do not find a statistical difference. However, when considering

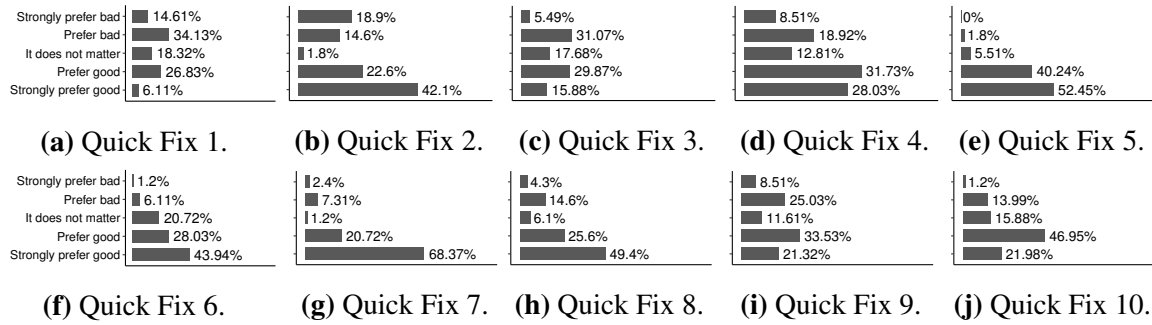


Figure 5.6: Preference of programmers for each quick fix.

only answers from non-experts, we cannot find statistical differences for Quick Fixes 2, 3, 4, and 9—i.e., the answers of experts differ from the answers of non-experts. This result supports our hypothesis that experts have a common wisdom that other programmers (e.g., non-experts) lack. We observe similar results between informed programmers and uninformed programmers, which supports the hypothesis that the participants’ preferences are related to the impact the code pattern has in the quality of their code.

Experts and informed programmers tend to prefer better code patterns compared to non-experts and uninformed programmers.

Most programmers use IDEs to detect edit patterns and perform quick fixes (82.5%). Most of them use IntelliJ (72.5%). Programmers also mention code analyzer tools, such as Checkstyle (50.3%), Sonar (49.7%), FindBugs (42.5%), and PMD (30.7%). Most programmers use multiple tools to detect edit patterns (59%). Only 0.6% of the programmers uses a single code analyzer, and no programmer uses Checkstyle or PMD as the only code analyzer. This indicates that programmers often use multiple tools to maintain their code. In addition, only 17% of the programmers indicate the use of the IDE alone without specifying any code analysis tool integrated with it. IDEs are not typically designated for detecting patterns. Figure 5.7 shows the distribution of the answers for tools.

A minority of the programmers (12%) does not indicate any tool, which hints at a low knowledge of the implications of using some different patterns. In fact, 60% of the users who did not know any tool selects average to very low knowledge of the effect of edit patterns. This is a large number since, in general, 72.4% of the total programmers indicates the contrary—i.e., a high or very high knowledge of the implications of the different patterns. We also

observe that programmers who do not use any tools are less experienced Java programmers (65% experts instead of 72%). For programmers who do not use any tool, the difference in answers for Quick Fixes 1, 2, 3, 4, 9, and 10 is not statistically significant. Additionally, for programmers that use only the IDE, the difference for Quick Fixes 1, 2, 3, 4, and 9 is not statistically significant. This result suggests that the IDE alone without any additional integrated code analysis is not the most suitable tool to detect edit patterns.

Programmers who use more tool support to detect and apply code patterns tend to prefer better code patterns compared to non-experts and uninformed programmers.

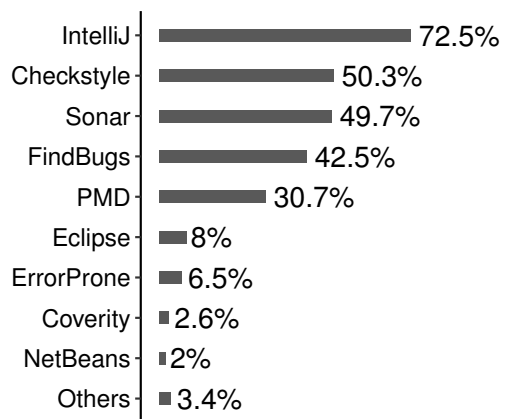


Figure 5.7: Distribution of answers for tools.

Qualitative analysis To understand additional reasons behind why programmers preferred one code pattern over another, we analyzed the explanations they wrote.

We observe some programmers favor readability over other benefits that edit patterns may provide, which leads them to not choose the code patterns suggested by our edit patterns. For instance, in edit pattern *String to character*, a participant answered:

“B is technically more efficient but A is likely more consistent with all your other [c]ode (where you rarely deal in characters) so let the compiler sort the optimization out” (sic).

For the edit *Field, parameter, and variable could be final*, programmers also indicate that having `final` everywhere can be noisy, as we can see in the answer of one of the participants.

“Although it is definitely preferable for all locals to be effectively final, explicitly marking them as such adds a lot of noise.”

Programmers may favor readability of their original code over other benefits provided by edit patterns.

We also observe that programmers may prefer a good code pattern without fully understanding its benefits. For instance, for the edit *Use valueOf instead wrapper constructor* a participant mentions:

“Prefer constructors to standard objects, though truth be told idk [I Don’t Know] what `valueOf` does beside return a new `Integer`” (sic).

In fact, `valueOf` does not return a new `Integer`, but caches frequently requested values. This means some programmers are unaware of the effects of some code patterns. Among the programmers that use a tool that detects this pattern, *23.19% still choose the wrong option*. Some programmers suggest `Integer a = 1`, which the compiler translates to use `valueOf` [104].

For the edit *Avoid FileInputStream and FileOutputStream*, programmers indicate other reasons for choosing the good pattern. Most programmers indicate that the modified version is better because it uses a general type `OutputStream` instead of `FileOutputStream`.

Some programmers prefer some code patterns for different reasons than those the patterns are designed for.

Finally, we observe static analyzers plays an important role in teaching good practices to programmers. A participant wrote:

“I forget why, but I’m pretty sure that B is better. PMD enforces it at my company”.

Static analysis tools can teach programmers to recognize good code patterns.

5.5.3 Pull Requests

To further assess the perceived usefulness of our edit patterns, we submitted pull requests to identify whether programmers want to perform the edit in their code.

Planning We selected 12 Java code projects from different domains such as databases, servers, and library management. We also chose both traditional software projects and projects for code analysis tools such as PMD, Checkstyle, and ErrorProne. Five of the projects were chosen from the 9 projects we learned patterns from. The full list of selected projects is shown in Table 5.3. From the 10 edits in our survey, we selected 6 of them. We selected

Table 5.3: Projects used to submit pull requests.

Project	Edit Pattern	Status
VoltDB	Quick Fix 8	Open
Ant	Quick Fix 4	Merged
Ant	Quick Fix 6	Open
Ant	Quick Fix 5	Merged
Ant	Quick Fix 2	Merged
Ant	Quick Fix 1	Open
Drill	Quick Fix 4	Merged
Hive	Quick Fix 4	Open
Checkstyle	Quick Fix 6	Merged
ErrorProne	Quick Fix 4	Merged
ErrorProne	Quick Fix 2	Rejected
ExoPlayer	Quick Fix 2	Merged
Libgdx	Quick Fix 6	Open
Libgdx	Quick Fix 2	Rejected
PMD	Quick Fix 5	Merged
PMD	Quick Fix 1	Merged
RStudio	Quick Fix 5	Merged
RStudio	Quick Fix 1	Rejected
Sonarqube	Quick Fix 1	Open
Spotbugs	Quick Fix 6	Rejected

edit patterns to include both edit patterns that appear in existing code analysis tools (3) and edits that do not appear in any tool (3). In this process, we selected edit patterns that were likely to appear in open source code repositories. We submitted pull requests related to these patterns to the selected projects. For each edit patterns, we performed up to 4 pull requests. Since we wanted to receive a quick feedback from programmers about the edit patterns, we applied them to just a subset of locations per project since a large number of edits would have required a careful analysis and code review. We targeted 20 locations but if a single file contained more than 20 possible locations, we applied the edit pattern to all of them.

Results and Discussion At the time of writing, programmers found 10/20 (50%) of our pull requests useful and accepted them. Of the accepted requests, 6 were for patterns appearing in existing code analysis tools and 4 were for patterns that do not appear in any tool. Both traditional software projects and projects for code analysis tools accepted some pull requests. For the latter category, ErrorProne and Checkstyle accepted one pull request each, and PMD accepted two. Curiously, some of the pull requests fixed issues that their own static analysis tool could have detected. This may indicate issues in how programmers run static analysis tools. For instance, some pull requests fixed issues in test projects, which may not be inspected often by programmers and tools. Remarkably, the programmers of PMD found the *string to character* edit, which we applied to their code, to be particularly useful and *they will include it in PMD 6.2.0 as a new rule*.

Programmers rejected 4/20 (20%) of our pull requests, but they showed support for the edit patterns we proposed. In one case, our pull request intend to improve performance, but programmers indicated that the edits were not performed on critical locations and they will instead welcome edits to critical paths. In other two cases, we performed an edit to prevent a `NullPointerException`, but the locations where we performed the edits already included `null` checks. In the last case, we performed an edit to improve performance for Java code that is later translated to JavaScript. Although the edit improves performance in Java, the same does not apply to other languages.

For 3/20 (15%) of our pull requests, programmers supported our edits, but the pull requests are still open. In one case, the programmers required us to provide a test suite to demonstrate the gain in performance. In another case, we edited the code to use `StringBuilder`, which improves performance but is not thread-safe. Thus, we need further analysis to decide what locations are safe and can be edited. In the last case, programmer reacted positively to our edit but have not accepted the pull request yet. Finally, 3/20 (15%) of our pull requests do not receive answers yet.

Programmers were positive towards 70% of the pull requests and one of the edit patterns was integrated in PMD.

5.5.4 Threats to validity

With respect to construct validity, our edit patterns are based on the edits programmers perform in open source repositories. These programmers may not be experts and their edits may not reflect good edit patterns. To reduce this threat, we validated the edit patterns learned from the source code using resources such as programming communities and the literature (e.g., Eckel [24]). Concerning internal validity, the edit patterns are classified manually, which may impact on the number of edits classified as useful. To reduce this threat, we evaluate our edit patterns using different validation approaches. We performed a survey to retrieve the opinion of programmers along GitHub, and we submitted pull requests applying our edit patterns to source code repositories. In addition, we select a sample in our catalog of edit patterns to evaluate both the survey and to submit pull requests. The choice of the edit patterns may affect the opinion of the programmers. To reduce this threat, we select edit patterns considering different aspects (e.g., edit included or missed by code analysis tools). Concerning external validity, we selected 9 projects to extract the code edit patterns. We also selected 12 projects to submit pull requests. These projects may not be representative of all the projects on GitHub. To reduce this threat, we selected projects from different domains such as databases, servers, and library management. When we performed the pull request, we selected both projects that we have used to extract the edit patterns or not. In this way, we could validate the technique not only with programmers that have seen the edit patterns before, but also general with general programmers. In addition, we only select a sample of learned code edit patterns. This may not reflect all edit patterns that could be useful to programmers. To reduce this threat, we select edit patterns that occur across more projects.

5.6 Related Work

Discovering code patterns Many systems have been recently proposed to mine code transformations and edit patterns from code. Molderez et al. [88] learn AST-level tree transformations that are used to automate repetitive code edits. It learns rules in the form of tree edits. Since the same pattern can be described with different tree edits, two patterns that are deemed equivalent by REVISAR can be deemed nonequivalent in [31]. Moreover, it abstracts the list of tree edits of a transformation into a set of edits, losing information that is important

when automating the transformation as done by REVISAR. Finally, the edits learned by it are not publicly available and were not evaluated through a survey. Brown et al. [10] learn token-level syntactic transformations—e.g., delete a variable or insert an integer literal—from online code commits and use them to generate new candidate mutations for mutation testing. Unlike REVISAR, can only mine token level transformations over a predefined set of syntactic constructs and cannot unify across multiple concrete edits. Negara et al. [91] mine code interactions directly from the IDE to detect repetitive edit patterns and finds 10 new refactoring patterns. REVISAR does not use continuous interaction data from an IDE and only makes use of public data available on online repositories. Despite this fact, REVISAR discovered (at least) 9 new useful quick fixes. Other tools [73, 138] mine fine-grained repair templates from StackOverflow the Defect4j bug data-set. REVISAR extracts edit patterns from online repositories and not blogs or repair data sets. Combining the two techniques to discover what patterns are useful is an interesting research direction. In summary, our paper differs from prior work in that (i) REVISAR mines new edit patterns in a sound and complete fashion (with respect to the syntax of edit patterns), (ii) by analyzing the edit patterns REVISAR mined, we discovered quick fixes and assessed their usefulness through a formal evaluation.

Some techniques use repetitiveness of code edits [36, 47] to build statistical models for code completion. APIRec [92] is trained on a corpus of repetitive edits to learn a statistical model based on the co-occurrence of fine-grained changes. APIRec can then recommend API calls based on edit context. GraLan and ASTLan [93] are graph-based and AST-based techniques that recommend the next token based on statistical models. Our technique analyzes edit patterns from unsupervised data and not from curated data. Moreover, we perform a formal study of the usefulness of quick fixes we discovered.

Learning transformations from examples Several techniques use user-given examples to learn repetitive code edits for refactoring [3, 83, 127], for removing code clones [81], for removing defects from code [55], and for learning ways to fix command-line errors [19]. All these techniques rely on the user-given examples and all the given examples should describe the same intended transformation. This extra information allows the tools to perform more informed types of rule extraction. Instead, REVISAR uses fully unsupervised learning and

receives concrete edits as input that describe different transformations. Moreover, REVISAR is sound and complete for our edit-pattern format thanks to the use of anti-unification.

Program repair Several tools for program repair learn useful fixing strategies by mining code. FixWizard [99] is a semi-automated technique for detecting and fixing common bugs. FixWizard identifies target locations, detects fixes, and recommends fixing edits. BugFix [50] uses association rules to help to fix a bug. An association rule is a machine learning technique to discover interesting relations among variables. It suggests fixes by analyzing historic of bug fixing scenarios. Association rules map debugging scenarios to fixing scenarios. It uses these rules to suggest a list of fixes. Kim et al. [56] use human-written patches to repair programs. They analyze 62,652 human-written patches and identified ten common fixing templates. Its technique detects bug locations using a fault location technique [67] and uses these fixing templates to repair program. Liu et al. [72] propose R2Fix, a technique that learns patches from bug reports. R2Fix analysis bug reports, identify common bugs, and recommends program repairs. This technique contains a bug classifier, a parameter extractor that extracts pattern based on bug reports, and a patch generator. Long and Rinard [74] present Prophet, a technique for automatic program repair that learns a probabilistic application-independent model of correct code from a set of successful human patches. All these tools either rely on a predefined set of patches or learn patches from supervised data—e.g., learn how to fix a `NullPointerException` by mining all concrete edits that were performed to resolve that type of exception. Unlike these techniques, we analyze fully unsupervised set of concrete code edits and we propose a sound and complete technique for mining edit patterns. Moreover, REVISAR learns arbitrary quick fixes that can improve code quality and not just ones that are used to repair buggy code. Since we do not have a notion of correct edit pattern (i.e., there is no bug to fix), we also analyze the usefulness of the learned quick fixes through a comprehensive evaluation and user study, a component that is not necessary for the code transformations used in program repair.

5.7 Concluding remarks

We presented REVISAR, a technique for automatically discovering common Java code edit patterns in online code repositories. REVISAR *(i)* identifies edits by comparing pairs of revisions in input revision histories, *(ii)* clusters edits into sets that can be abstracted into the same edit pattern, and *(iii)* for each cluster, learns an edit pattern. Finally, when possible, it compiles the edit pattern to executable code in the Google's ErrorProne extension of the Java compiler. We used REVISAR to mine quick fixes from nine popular Java projects from GitHub and REVISAR successfully learned 920 edit patterns that appeared in more than one projects. So far, upon manual inspection, we identified 32 quick fixes. To assess whether programmers would like to apply these quick fixes to their code, we performed an online survey with 164 programmers showing 10 of our quick fixes. Overall, programmers supported 90% of our quick fixes. For such quick fixes, we issued pull requests in various repositories and 50% were accepted so far. Moreover, *PMD will include one of the quick fixes we discovered in their next release.*

Chapter 6

Conclusion

Tools such as ErrorProne [39], ReSharper [51], Coverity [6], FindBugs [4], PMD [120], and Checkstyle [14] help programmers by automatically detecting and/or removing several suspicious code patterns, potential bugs, or instances of bad code style. However, extending these catalogs of transformations is complex and time-consuming. In this context, programmers may want to perform a repetitive edit into their code automatically to improve their productivity, but available tools do not support it. In addition, designers of code analysis tools may want to identify rules that could be helpful to be automated.

To deal with the problem of allowing programmers to perform repetitive edits that are not in the catalog of code analysis tools, we presented REFAZER, a technique for synthesizing syntactic transformations from examples. Given a set of examples consisting of program edits REFAZER synthesizes a transformation that is consistent with the examples. Our synthesizer builds on the state-of-the-art program synthesis engine PROSE. To enable it, we develop (i) a novel DSL for representing program transformations, (ii) domain-specific constraints for the DSL operators, which reduce the space of search for transformations, and (iii) ranking functions for transformation robustness, based on the structure of the synthesized transformations. We evaluated REFAZER in two applications: synthesizing transformations that describe how students “fix” their programming assignments and synthesizing transformations that apply repetitive edits to large codebases. Our technique learned transformations that automatically fixed the program submissions of 87% of the students participating in a large UC Berkeley class and it learned the transformations necessary to apply the correct code edits for 84% of the repetitive tasks we extracted from three large code repositories.

In addition to being a useful tool, REFAZER makes two novel achievements in PBE. First, it is the first application of backpropagation-based PBE methodology to a domain unrelated to data wrangling or string manipulation. Second, in its domain it takes a step towards development of fully unsupervised PBE, as it automates extraction of input-output examples from the datasets (that is, students' submissions or programmers' modifications). We hope that with our future work on incorporating flow analyses into witness functions, REFAZER will become the first major application of inductive programming that leverages research developments from the entire field of software engineering.

To discover useful edit patterns that could be useful to be added to the catalog of code analysis tools, we propose REVISAR, a technique for automatically discovering common Java code edit patterns in online code repositories. Given code repositories as input, REVISAR identifies simple edit patterns by comparing consecutive revisions in the revision histories. The most common edit patterns—i.e., those performed across multiple projects—can then be inspected to detect useful ones and add the corresponding rules to code analyzers.

Since we are interested in detecting patterns appearing across projects and revisions, REVISAR has to analyze large amounts of code, a task that requires efficient and precise algorithms. REVISAR focuses on edits performed to individual code locations and uses GumTree [31], a tree edit distance algorithm, to efficiently extract concrete abstract-syntax-tree edits from individual pairs of revisions—i.e., sequences of tree operations such as insert, delete, and update. REVISAR then uses a greedy algorithm to detect subsets of concrete edits that can be described using the same edit pattern. To perform this last task efficiently, REVISAR uses a variant of a technique called anti-unification [63], which is commonly used in inductive logic programming. Given a set of concrete edits, the anti-unification algorithm finds the least general generalization of the two edits—i.e., the largest pattern shared by the edits. Thanks to this formal result, REVISAR is sound and complete with respect to the rule format it adopts—i.e., if there is a rule consistent with a given set of concrete edits, REVISAR will find it.

6.1 Follow up approaches

In this thesis, we focus our evaluation on how a transformation learned from programming assignments can help students with incorrect solutions. Some works use program synthesis to help teachers to generate hints. Although REFAZER is capable of fixing students bugs automatically, it lacks the domain knowledge of a teacher and can, therefore, generate functionally correct but stylistically bad fixes. To address this limitation, MistakeBrowser [46], a mixed-initiative approach, uses teacher expertise to better leverage the fixes produced by REFAZER. MistakeBrowser clusters incorrect submissions for the student submission history. Each cluster is based on common bugs among students. Then, teachers provide hints for the incorrect submission, and MistakeBrowser propagates teachers hints to the incorrect submissions in the same cluster. MistakeBrowser requires data about the programming assignment submissions from the previous semesters. Sometimes, instructors lack this information. In this case, teachers can write examples to correct incorrect submission, and use REFAZER to fix the students' programming assignments. This is the idea of FixPropagator [46]. This technique can learn code transformations from teachers' bug fixes for correcting students' submissions. The teachers provide examples of bug fixings for students' programming assignments and provide hints for the provided fixes. Then, Refazer learns programming transformations to correct the bug in the students' programming assignments and propagates the fixes along with the feedback to other students with similar incorrect submission. The last tool is TraceDiff [137], which helps students to debug the code. It shows the trace of the incorrect and correct versions of the code. Thus, students can understand why the code is incorrect.

6.2 Future work

In this section, we present the future work to be done after this thesis. In this thesis, we use positive examples (i.e., examples of what programmers want to perform) to synthesize program transformations. Another kind of example that could also be used along the synthesis process is negative examples (i.e., examples of edits that programmers do not want to perform). Different from positive examples, negative examples specify what users do not want. Some

techniques use negative examples to bound the space of possible transformations. For instance, FlashExtract uses negative examples to remove incorrect transformations from synthesized list of transformations. All transformations that produce an output that is in the list of negative examples given the input are removed from the transformation list. As another example, FlashProg interacts with end-users that may provide negative examples to get to the desired transformation. However, these studies investigate the use of negative examples on the data-wrangling domain.

These results from the data-wrangling domain suggest that negative examples also could be beneficial in the process of learning program transformations for source code. For instance, if programmers want to perform a repetitive edit, but they have already performed this transformation manually to some locations in the source code, they probably will not want that this transformation be performed to edited locations. In this scenario, programmers may provide locations where they already have applied the transformation as negative examples, and REFAZER could use this information to refine synthesized program transformations list.

In this future work, we want to use the support to negative examples of the PROSE framework and modify our witness functions to accept this kind of specification. In this process, we will allow programmers to give negative examples implicit or explicit. To detect implicit negative examples, we want to use some heuristics. For instance, FlashExtract analyzes the examples provided by programmers. If programmers provide two or more examples, FlashExtract automatically adds the regions between the examples as a negative example to refine the transformation list, so that any transformation that selects regions between these examples is excluded. The explicit negative example are provided by programmers before the program transformations are learned or after the best-ranked program transformation shows potential case of repetitive edits to programmers. In the last case, programmers could select some of these edits as negative examples to refine the program transformations.

Moreover, we identified cases where REFAZER was unable to perform the desired transformations. As future work, we plan to increase the expressivity of our tree pattern expressions to avoid selecting incorrect locations due to over-generalization. Currently, the generalization is based on rigid alignment between nodes in two trees. For instance, if two nodes have the same parent, but have different numbers of children, we can currently only match them with respect to their type, which may be too general. To support this kind of pattern, we plan to

include additional predicates in our DSL such as `Contains`, which does not consider the entire list of children, but checks if any of the children match a specific pattern.

Additionally, we aim at investigating the use of control-flow and data-flow analysis for identifying the context of the transformation. For instance, `REFAZER` may introduce a name into the source code, but this node may also be defined in the source code. With data and control-flow analysis, we want to analyze the context where `REFAZER` is performing the transformation to avoid compilation errors due to a situation like this one.

We also plan to extend our DSL to support type analysis and preconditions for each transformation. Currently, `REFAZER` only uses the name and label of the nodes in the source code. We also want to investigate the type of the expression. For instance, the programmer may want to perform a transformation, but he only wants to perform this transformation to certain overloaded method. Without type checking, we are unable to perform this transformation. Thus, we want to add type checking to our language to analyze the type of the expression.

Additionally, we want to investigate the use of variables. Some pattern re-occurs within the same transformation. If we can abstract this occurrence in a variable operator, we can only match this tree a single type. The other matches may reuse the previously learned variable. If the examples to learn a transformation were so distinct among themselves, we cannot learn a transformation. Furthermore, the examples may diverge in terms of applied edits; that is, one example may have an update operation and a delete operation, and the other example may have just an update operation. We will extend the DSL operators to group examples according to their similarities and learn transformations for each group of examples separately.

Although `REFAZER` is capable of fixing student bugs automatically, it lacks the domain knowledge of a teacher and can therefore generate functionally correct but stylistically bad fixes. To address this limitation, we have recently built a mixed-initiative approach that uses teacher expertise to better leverage the fixes produced by `REFAZER` [46]. Teachers can write feedback about an incorrect submission or a cluster of incorrect submissions. The system propagates the feedback to all other students who made the same mistake.

In this thesis, we have the history of submissions of each student for a programming assignment. For each student who completed the assignment and had at least one incorrect submission, we select the last pair of submissions, the incorrect and the correct ones, and we use it as examples of bug fixes. However, we do not investigate the learning of bug fixes from

non-consecutive pair of submissions. In this case, students may have applied additional edits, such as refactorings. Although learning from the last incorrect submission, we increase the likelihood of learning a transformation that is focused on fixing the existing faults, learning from non-consecutive submissions, we increase the number of learned rules and, therefore, can fix the bug of with a few number students' submissions. Thus, in this future work, we want to perform an evaluation of learning larger transformations from earlier incorrect submissions to correct submissions, and how these transformations can help to fix larger conceptual faults in the code.

We also want to improve our ranking functions. In this thesis, we use some heuristics for ranking the programming transformations, such as preferring some operators over others. In future, we want to use machine learning techniques for learning the ranking models automatically. In this direction, some techniques were proposed. Singh and Gulwani [134] present a machine learning technique for ranking transformations. Since the examples are a kind of incomplete specification, the number of learned transformations could be huge. Thus, evaluating all transformations against the ranking functions may be prohibitive. In this scenario, they propose a gradient descent approach to rank the program transformation at the VSA level. Menon et al. [84] present a probabilistic model that ranks programs based on the feature of the provided input-output examples. These works focus on the data-wrangling domain. As future work, we want to investigate the use of machine learning techniques to rank our program transformations.

Additionally, we want to investigate how programmers could interact with REFAZER/REVISAR. In thesis, we focus on learning transformations and leave the interaction models as future work. Among the studies that investigate this problem, FlashProg [78] learns transformations and when two transformations produce different results for the same input, it shows to the user cases in which these two transformations differ and ask him which one produces the desired result. This work focuses on the data-wrangling domain. Another problem is how to present the result of program transformations to programmers. In software engineering, the works focus on the development of techniques. The interaction model in these techniques is minimal or nonexistent. In this work, we want to investigate how programmers could interactive with our technique. We want to perform a formative study showing different

interaction models to programmers. Using the data from this formative study, we aim to develop a interaction model for REFAZER.

Regarding REVISAR, our work mine transformations that can be integrated in code analyzers and opens many research directions. Currently our technique requires the programmers of code analyzers to manually investigate what edit patterns are useful while it would be desirable to design an automated approach for checking usefulness based on crowd-sourcing (e.g., see [140]) or on mining StackOverflow blog posts (e.g., see [73]).

Additionally, REVISAR tackles individual edits, but it would desirable to extend it to more complex types of edit patterns. We plan to draw inspiration from the program repair literature and consider concepts such as contexts and dependencies between edits. For instance, when programmers change the source code to use `List` instead of `Vector`, they also need to edit specific `Vector` usage to the `List` usage. For example, the `Vector` API contains methods (e.g., `addElement`) that required to be changed to their correspondent methods in the `List` API (e.g., `add`). In this case, the dependencies among edits is needed to preserve the type correctness of the program. To do so, we could we type constraints that had been used in the refactoring domain to express the type correctness of the program [143]. The type constraints specify a set of rules that defines when the program is type correct. When some rules are broken, it implies that a set of additional transformations must be performed to make the program type correct again.

Furthermore, since our goal is to discover patterns that exploit unknown features of programming language, it would be beneficial to mine edit patterns for other programming languages, such as C# and Python. Last, our study covers only a limited set of repositories and it will be interesting to see if analyzing different repositories yields new quick fixes.

Bibliography

- [1] Abdulaziz Alkhalid, Mohammad Alshayeb, and Sabri Mahmoud. Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. *International Journal of Advances in Software Engineering*, 41(10-11):1160–1178, October 2010.
- [2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design*, FMCAD '13, pages 1–8, Piscataway, NJ, USA, 2013. IEEE Press.
- [3] J. Andersen and J. L. Lawall. Generic patch inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 337–346, Washington, DC, USA, 2008. IEEE Computer Society.
- [4] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, September 2008.
- [5] Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. *Information and Computation*, 255:262 – 286, 2017.
- [6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Halleem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [7] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *Proceedings of the*

- SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 567–576, New York, NY, USA, 2007. ACM.
- [8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.
- [9] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. Paraphrasing: Generating parallel programs using refactoring. In *Proceedings of the 10th International Symposium on Formal Methods for Components and Objects*, FMCO '11, pages 237–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [10] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 511–522, New York, NY, USA, 2017. ACM.
- [11] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.
- [12] Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the 3rd International Workshop On Software Clones*, IWSC '09, pages 1–6, Kaiserslautern, Germany, 2009. Fraunhofer IESE.
- [13] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.
- [14] Checkstyle. Checkstyle project. At <http://checkstyle.sourceforge.net/>, 2018.

- [15] James R. Cordy. Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 1–11, New York, NY, USA, 2006. ACM.
- [16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [17] Bradley Cossette, Robert Walker, and Rylan Cottrell. Using structural generalization to discover replacement functionality for API evolution, 2014.
- [18] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.
- [19] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 582–592, New York, NY, USA, 2017. ACM.
- [20] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, ICSM '09, pages 169–178, Piscataway, NJ, USA, 2009. IEEE Press.
- [21] Harvey M. Deitel and Paul J. Deitel. *Java How to Program (6th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.
- [22] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20(1):3:1–3:31, July 2010.
- [23] DZone. Fileinputstream / fileoutputstream considered harmful, 2017. At <https://dzone.com/articles/fileinputstream-fileoutputstream-considered-harmful>. Accessed in 2017, December 19.

-
- [24] Bruce Eckel. *Thinking in Java (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
- [25] Eclipse. Eclipse jdt. At <https://www.eclipse.org/jdt/>, 2018.
- [26] Darren Edge, Sumit Gulwani, Natasa Milic-Frayling, Mohammad Raza, Reza Adhitya Saputra, Chao Wang, and Koji Yatani. Mixed-initiative approaches to global editing in slideware. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3503–3512. ACM, 2015.
- [27] EntityFramework. Entity Framework 6. At <https://entityframework.codeplex.com/>.
- [28] Martin Erwig and Deling Ren. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67(2-3):199–222, July 2007.
- [29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2th International Conference on Knowledge Discovery and Data Mining, KDD '96*, pages 226–231. AAAI Press, 1996.
- [30] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. In *Proceedings of 14th Working Conference on Reverse Engineering, WCRE '07*, pages 150–159, Piscataway, NJ, USA, 2007. IEEE Press.
- [31] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering, ASE '14*, pages 313–324, New York, NY, USA, 2014. ACM.
- [32] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC '06*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.
- [33] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

- [34] Stephen R. Foster, William G. Griswold, and Sorin Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 222–232, Piscataway, NJ, USA, 2012. IEEE Press.
- [35] The Eclipse Foundation. Eclipse. At <https://eclipse.org/>, 2018.
- [36] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, pages 147–156, New York, NY, USA, 2010. ACM.
- [37] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.
- [38] Google. Clang-tidy. At <http://clang.llvm.org/extra/clang-tidy/>, 2016.
- [39] Google. Error-prone. At <http://errorprone.info/>, 2016.
- [40] Google Guava. Class stopwatch, 2017. At <https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Stopwatch.html>. Accessed in 2017, December 19.
- [41] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP '10*, pages 13–24, New York, NY, USA, 2010. ACM.
- [42] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11*, pages 317–330, New York, NY, USA, 2011. ACM.
- [43] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference Automated Reasoning, IJCAR '16*, pages 9–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

- [44] Summit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.
- [45] Shinpei Hayashi, Motoshi Saeki, and Masahito Kurihara. Supporting refactoring activities using histories of program modification. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 89-D(4):1403–1412, 2006.
- [46] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D’Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale, L@S ’17*, pages 89–98, New York, NY, USA, 2017. ACM.
- [47] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering, ICSE ’12*, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.
- [48] NetBeans IDE. Netbeans ide. At <https://netbeans.org/>, 2018.
- [49] Java JDK. Relax fileinputstream/fileoutputstream requirement to use finalize, 2017. At <https://bugs.openjdk.java.net/browse/JDK-8187325>. Accessed in 2017, December 19.
- [50] D. Jeffrey, M. Feng, Neelam Gupta, and R. Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension, ICPC ’09*, pages 70–79, Piscataway, NJ, USA, 2009. IEEE Press.
- [51] JetBrains. ReSharper. At <https://www.jetbrains.com/resharper/>, 2018.
- [52] JGit. Eclipse jgit. At <https://www.eclipse.org/jgit/>, 2018.
- [53] Richard Joiner, Thomas Reps, Somesh Jha, Mohan Dhawan, and Vinod Ganapathy. Efficient runtime-enforcement techniques for policy weaving. In *Proceedings of the 22Nd*

- ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '14*, pages 224–234, New York, NY, USA, 2014. ACM.
- [54] Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring references for library migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 726–738, New York, NY, USA, 2010. ACM.
- [55] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design defects detection and correction by example. In *Proceedings of the 19th IEEE International Conference on Program Comprehension, ICPC '11*, pages 81–90, Piscataway, NJ, USA, 2011. IEEE Press.
- [56] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering, ICSE '13*, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.
- [57] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 1145–1156, New York, NY, USA, 2016. ACM.
- [58] Miryung Kim and Na Meng. *Recommending Program Transformations to Automate Repetitive Software Changes*, pages 1–31. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
- [59] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE '14*, pages 35–45, New York, NY, USA, 2006. ACM.
- [60] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering. In *Evidence-Based Software Engineering. Technical Report, EBSE*, pages 1–65, 2007.

-
- [61] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 55(12):2049–2075, 2013.
- [62] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.
- [63] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *Journal of Automated Reasoning*, 52(2):155–190, February 2014.
- [64] LaPES. Start: State of the art through systematic review. At http://lapes.dc.ufscar.br/tools/start_tool, 2017.
- [65] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Journal of Machine Learning*, 53(1-2):111–156, October 2003.
- [66] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 542–553, New York, NY, USA, 2014. ACM.
- [67] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.
- [68] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 565–574, 2015.
- [69] Huiqing Li and Simon Thompson. Let’s make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12*, pages 32–39, New York, NY, USA, 2012. ACM.

- [70] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe Java program adaptation between apis. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15*, pages 91–102, New York, NY, USA, 2015. ACM.
- [71] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [72] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *6th IEEE International Conference on Software Testing, Verification and Validation, ICST '13*, pages 282–291, Piscataway, NJ, USA, 2013. IEEE Press.
- [73] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *25th edition of IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER '18*, page to appear, 2018.
- [74] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 51(1):298–312, 2016.
- [75] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 711–722, 2016.
- [76] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.
- [77] Marius Marin, Arie Deursen, Leon Moonen, and Robin Rijst. An integrated cross-cutting concern migration strategy and its semi-automated application to JHotDraw. *Automated Software Engineering*, 16(2):323–356, 2009.
- [78] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual*

- ACM Symposium on User Interface Software & Technology*, UIST '15, pages 291–301, New York, NY, USA, 2015. ACM.
- [79] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, and Balduino Fonseca dos Santos Neto. A catalogue of refactorings to remove incomplete annotations. *The Journal of Universal Computer Science*, 20(5):746–771, 2014.
- [80] Mathias Mehrmann. Apache ant, 2002. https://bz.apache.org/bugzilla/show_bug.cgi?id=11582.
- [81] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 392–402, Piscataway, NJ, USA, 2015. IEEE Press.
- [82] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 329–342, New York, NY, USA, 2011. ACM.
- [83] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.
- [84] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning*, ICML'13, pages 187–195. JMLR.org, 2013.
- [85] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.
- [86] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

-
- [87] Microsoft. Visual Studio. At <https://www.visualstudio.com>, 2016.
- [88] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 248–256, Piscataway, NJ, USA, 2017. IEEE Press.
- [89] Martin Monperrus. Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille, 2015.
- [90] Rodrigo Morales, Z ephyrin Soh, Foutse Khomh, Giuliano Antoniol, and Francisco Chicano. On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software*, pages –, 2016.
- [91] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 803–813, New York, NY, USA, 2014. ACM.
- [92] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA, 2016. ACM.
- [93] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.
- [94] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering*, ASE '17, pages 180–190, 2013.

- [95] H. A. Nguyen, H. V. Nguyen, T. T. Nguyen, and T. N. Nguyen. Output-oriented refactoring in php-based dynamic web applications. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 150–159, Sept 2013.
- [96] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of 28th the IEEE/ACM International Conference on Automated Software Engineering, ASE '13*, pages 180–190, New York, NY, USA, 2013.
- [97] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.
- [98] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 302–321, New York, NY, USA, 2010. ACM.
- [99] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering - Volume 1, ICSE '10*, pages 315–324, New York, NY, USA, 2010. ACM.
- [100] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Clone-aware configuration management. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 123–134, Washington, DC, USA, 2009. IEEE Computer Society.
- [101] Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32Nd IEEE/ACM International Conference on Software Engineering - Volume 1, ICSE '10*, pages 205–214, New York, NY, USA, 2010. ACM.
- [102] NuGet. NuGet 2. At <https://github.com/nuget/nuget2>.

- [103] Mark O’Keeffe and Mel í Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, April 2008.
- [104] Oracle. Autoboxing and unboxing. At <https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>. Accessed in 2017, December 19, 2017.
- [105] Oracle. Class `concurrentskiplistset<e>`, 2017. At <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html>. Accessed in 2017, December 19.
- [106] Oracle. Class `float`, 2017. At <https://docs.oracle.com/javase/9/docs/api/java/lang/Float.html>. Accessed in 2017, December 19.
- [107] Oracle. Class `stringbuilder`, 2017. At <https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>. Accessed in 2017, December 19.
- [108] Oracle. Type inference for generic instance creation, 2017. At <https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html>. Accessed in 2017, December 19.
- [109] Oracle Java Documentation. Raw types, 2017. At <https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html>. Accessed in 2017, December 19.
- [110] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. Search-based refactoring using recorded code changes. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR ’13, pages 221–230, Washington, DC, USA, 2013. IEEE Computer Society.
- [111] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 25(3):23:1–23:53, 16.

- [112] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi. Improving multi-objective code-smells correction using development history. *J. Syst. Softw.*, 105(C):18–39, July 2015.
- [113] Jeffrey L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring c with macros. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '14*, pages 75–85, New York, NY, USA, 2014. ACM.
- [114] Stack Overflow. @nullable annotation usage, 2017. At <https://stackoverflow.com/questions/14076296/nullable-annotation-usage>. Accessed in 2017, December 19.
- [115] Stack Overflow. When should i use “this” in a class?, 2017. At <https://stackoverflow.com/questions/2411270/when-should-i-use-this-in-a-class>. Accessed in 2017, December 19.
- [116] Stack Overflow. When should i use “this” in a class?, 2017. At <https://logback.qos.ch/>. Accessed in 2017, December 19.
- [117] U. Pakdeetrakulwong, P. Wongthongtham, and W. V. Siricharoen. Recommendation systems for software engineering: A survey from software development life cycle phase perspective. In *The 9th International Conference for Internet Technology and Secured Transactions, (ICITST '14)*, pages 137–142, 2014.
- [118] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. An empirical study of supplementary bug fixes. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12*, pages 40–49, Piscataway, NJ, USA, 2012. IEEE Press.
- [119] Mateusz Pawlik and Nikolaus Augsten. RTED: A robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.
- [120] PMD. PMD: an extensible cross-language static code analyzer. At <https://pmd.github.io/>, 2018.

- [121] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 542–553, New York, NY, USA, 2015. ACM.
- [122] Chris Povirk. Google guava, 2012. <https://github.com/google/guava/commit/8f48177>.
- [123] Java Practices. Use final liberally, 2017. At <http://www.javapractices.com/topic/TopicAction.do?Id=23>. Accessed in 2017, December 19.
- [124] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The uniqueness of changes: Characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 34–44, Piscataway, NJ, USA, 2015. IEEE Press.
- [125] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. Refactoring with synthesis. In *Proceedings of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 339–354, New York, NY, USA, 2013. ACM.
- [126] Romain Robbes and Michele Lanza. Example-based program transformation. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *MoDELS '08*, pages 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [127] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.
- [128] Roslyn. Eclipse java development tools (jdt). At <https://github.com/dotnet/roslyn>, 2011.

-
- [129] T. Ruhroth, H. Wehrheim, and S. Ziegert. Rel: A generic refactoring language for specification and execution. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 83–90, 2011.
- [130] G. Santos, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente. Recording and re-playing system specific, source code transformations. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM '15*, pages 221–230, Piscataway, NJ, USA, 2015. IEEE Press.
- [131] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 71–80, New York, NY, USA, 2011. ACM.
- [132] Sandro Schulze, Malte Lochau, and Saskia Brunswig. Implementing refactorings for FOP: Lessons learned and challenges ahead. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development, FOSD '13*, pages 33–40, New York, NY, USA, 2013. ACM.
- [133] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '16*, pages 858–870, New York, NY, USA, 2016. ACM.
- [134] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Proceedings of the 27th International Conference on Computer Aided Verification, CAV '15*, pages 398–414, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [135] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 48(6):15–26, 2013.

- [136] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 289–299, 2011.
- [137] Gustavo Soares, Ryo Suzuki, Björn Hartmann, Andrew Head, Elena Glassman, Loris D’Antoni, and Ruan Reis. Tracediff: Debugging unexpected behavior in code assignments with synthesized corrections, 2017.
- [138] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *25th edition of IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER ’18, page to appear, 2018.
- [139] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A refactoring constraint language and its application to Eiffel. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP’11, pages 255–280, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [140] Kathryn T. Stolee and Sebastian Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM ’10, pages 35:1–35:4, New York, NY, USA, 2010. ACM.
- [141] Boya Sun, Ray-Yaung Chang, Xianghao Chen, and Andy Podgurski. Automated support for propagating bug fixes. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE ’08, pages 187–196, Washington, DC, USA, 2008. IEEE Computer Society.
- [142] Wesley Tansey and Eli Tilevich. Annotation refactoring: Inferring upgrade transformations for legacy applications. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA ’08, pages 295–312, New York, NY, USA, 2008. ACM.

- [143] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):9:1–9:47, May 2011.
- [144] M. Toomim, A. Begel, and S.L. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing, VL-HCC '14*, pages 173–180, Piscataway, NJ, USA, 2004. IEEE Press.
- [145] Santiago A. Vidal and Claudia A. Marcos. Building an expert system to assist system refactorization. *Expert Systems with Applications*, 39(3):3810–3816, February 2012.
- [146] Santiago A. Vidal and Claudia A. Marcos. Toward automated refactoring of crosscutting concerns into aspects. *Journal of Systems and Software*, 86(6):1482 – 1497, 2013.
- [147] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming Programs between APIs with Many-to-Many Mappings. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [148] Louis Wasserman. Scalable, example-based refactorings with Refaster. In *Proceedings of the 6th ACM Workshop on Workshop on Refactoring Tools, WRT '13*, pages 25–28, New York, NY, USA, 2013. ACM.
- [149] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, pages 173–182, New York, NY, USA, 2009. ACM.
- [150] World Wide Web Consortium. XPath. At <https://www.w3.org/TR/xpath/>. Accessed in 2017, May 12, 1999.
- [151] Ling Wu, Qian Wu, Guangtai Liang, Qianxiang Wang, and Zhi Jin. Transforming code with compositional mappings for API-library switching. In *39th IEEE Annual*

Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2, pages 316–325, 2015.

- [152] Xin Xia and David Lo. An effective change recommendation approach for supplementary bug fixes. *Automated Software Engineering*, pages 1–44, 2016.
- [153] Zhenchang Xing and Eleni Stroulia. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.
- [154] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. NetEgg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 20:1–20:7, 2014.
- [155] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.
- [156] Minhaz F. Zibran and Chanchal K. Roy. The road to software clone management: A survey. *Technical report*, pages 1–68, 2012.

Appendix A

Search string for each datasource

In this Chapter, we present the search string for each datasource. In Section [A.1](#), we present the search strings for ACM. In Section [A.2](#), we present the search string for Scopus. In Section [A.3](#), we present the search string for Engineering village. In Section [A.4](#), we present the search string for ScienceDirect. Finally, we Section [A.5](#), we present the search string for IEEE.

A.1 ACM search string for RQ 01

We divide the search string for ACM into three search strings based on work title (Section [A.1.1](#)), abstract (Section [A.1.2](#)), and keywords (Section [A.1.3](#)).

A.1.1 ACM search string based on title

```
"query": acmdlTitle:("software engineering" OR developer OR programmer) AND acmdlTitle:("systematic change" OR "program repair" OR "API usage adaptation" OR "Code change" OR "program change" OR "program transformation" OR "script transformation" OR "source transformation" OR "code transformation" OR "repetitive change" OR "non-unique change" OR "duplicated change" OR "duplicated code" OR "systematic edit" OR "linked edit" OR "refactoring" OR "bug-fix") AND acmdlTitle:(technique OR tool OR recommender OR algorithm OR synthesizer)
```

```
"filter": owners.owner=HOSTED
```


A.1.2 ACM search string based on abstract

"query": recordAbstract:("software engineering" OR developer OR programmer) AND recordAbstract:("systematic change" OR "program repair" OR "API usage adaptation" OR "Code change" OR "program change" OR "program transformation" OR "script transformation" OR "source transformation" OR "code transformation" OR "repetitive change" OR "non-unique change" OR "duplicated change" OR "duplicated code" OR "systematic edit" OR "linked edit" OR "refactoring" OR "bug-fix") AND recordAbstract:(technique OR tool OR recommender OR algorithm OR synthesizer)

"filter": owners.owner=HOSTED

A.1.3 ACM search string based on keywords

"query": keywords.author.keyword:("software engineering" OR developer OR programmer) AND keywords.author.keyword:("systematic change" OR "program repair" OR "API usage adaptation" OR "Code change" OR "program change" OR "program transformation" OR "script transformation" OR "source transformation" OR "code transformation" OR "repetitive change" OR "non-unique change" OR "duplicated change" OR "duplicated code" OR "systematic edit" OR "linked edit" OR "refactoring" OR "bug-fix") AND keywords.author.keyword:(technique OR tool OR recommender OR algorithm OR synthesizer)

"filter": owners.owner=HOSTED

A.2 Scopus search string for RQ 01

TITLE-ABS-KEY ("software engineering" OR developer OR programmer) AND TITLE-ABS-KEY ("systematic change" OR "program repair" OR "API usage adaptation" OR "Code change" OR "program change" OR "program transformation" OR "script transformation" OR "source transformation" OR "code transformation" OR "repetitive change" OR "non-unique change" OR "duplicated change" OR "duplicated code" OR "systematic edit" OR "linked edit" OR "refactoring" OR "bug-fix") AND TITLE-ABS-KEY (technique OR tool OR recommender OR algorithm OR synthesizer) AND (LIMIT-TO (SUBJAREA , "COMP"))

A.3 Engineering village search string for RQ 01

((("software engineering" OR developer OR programmer) AND ("systematic change" OR "program repair" OR "API usage adaptation" OR "Code change" OR "program change" OR "program transformation" OR "script transformation" OR "source transformation" OR "code transformation" OR "repetitive change" OR "non-unique change" OR "duplicated change" OR "duplicated code" OR "systematic edit" OR "linked edit" OR "refactoring" OR "bug-fix") AND (technique OR tool OR recommender OR algorithm OR synthesizer) WN KY))

A.4 ScienceDirect search string for RQ 01

TITLE-ABSTR-KEY("software engineering" or developer or programmer) and TITLE-ABSTR-KEY("systematic change" or "program repair" or "API usage adaptation" or "Code change" or "program change" or "program transformation" or "script transformation" or "source transformation" or "code transformation" or "repetitive change" or "non-unique change" or "duplicated change" or "duplicated code" or "systematic edit" or "linked edit" or "refactoring" or "bug-fix") and TITLE-ABSTR-KEY (technique or tool or recommender or algorithm or synthesizer)[All Sources(Computer Science)]

A.5 IEEE search string for RQ 01

We divide the search string for IEEE into two search strings based on work abstract (Section [A.5.1](#)) and title and keywords (Section [A.5.2](#)).

A.5.1 IEEE search string based on abstract

We break our default search string in three search strings because the database allows a maximum of 15 search terms and the search string have more terms that this number.

("Abstract":.QT.software engineering.QT. OR "Abstract":developer OR "Abstract":programmer) AND ("Abstract":.QT.systematic change.QT. OR "Abstract":.QT.program repair.QT. OR "Abstract":.QT.API usage adaptation.QT. OR "Abstract":.QT.code change.QT. OR ."Abstract":QT.program change.QT. OR "Ab-

stract":.QT.program transformation.QT. OR "Abstract":.QT.script transformation.QT.) AND ("Abstract":technique OR "Abstract":tool OR "Abstract":recommender OR "Abstract":algorithm OR "Abstract":synthesizer)

("Abstract":.QT.software engineering.QT. OR "Abstract":developer OR "Abstract":programmer) AND ("Abstract":.QT.source transformation.QT. OR "Abstract":.QT.code transformation.QT. OR "Abstract":.QT.repetitive change.QT. OR "Abstract":.QT.non-unique change.QT. OR "Abstract":.QT.duplicated change.QT. OR "Abstract":.QT.duplicated code.QT. OR "Abstract":.QT.systematic edit.QT.) AND ("Abstract":technique OR "Abstract":tool OR "Abstract":recommender OR "Abstract":algorithm OR "Abstract":synthesizer)

("Abstract":.QT.software engineering.QT. OR "Abstract":developer OR "Abstract":programmer) AND ("Abstract":.QT.linked edit.QT. OR "Abstract":.QT.refactoring.QT. OR "Abstract":.QT.bug-fix.QT.) AND ("Abstract":technique OR "Abstract":tool OR "Abstract":recommender OR "Abstract":algorithm OR "Abstract":synthesizer)

A.5.2 IEEE search string based on title and keywords

The search string for document title and keywords follows the same pattern from the abstract (Section A.5.1). The only difference is the substitution of the keyword *Abstract* by *Document Title* and *Author Keywords*, respectively.

Appendix B

Catalog

In this section, we present a catalog of common code edits that programmers perform in Java. We now describe REVISAR, our technique for automatically discovering common Java code edits in code repositories. Given repositories as input, REVISAR: (i) Identifies concrete code edits by comparing pairs of consecutive revisions, (ii) Clusters edits into sets that can be described using the same code transformation and learns an abstract transformation for each cluster.

B.1 String to character

In Java, we can represent a character both as a `String` or a `character`. For operations such as appending a value to a `StringBuffer`, representing the value as a character improves performance. This edit improved the performance of some operations in the Guava project by 10-25% [122]. Figure B.1 shows an example of an edit that changes the code from the string notation to the character notation.

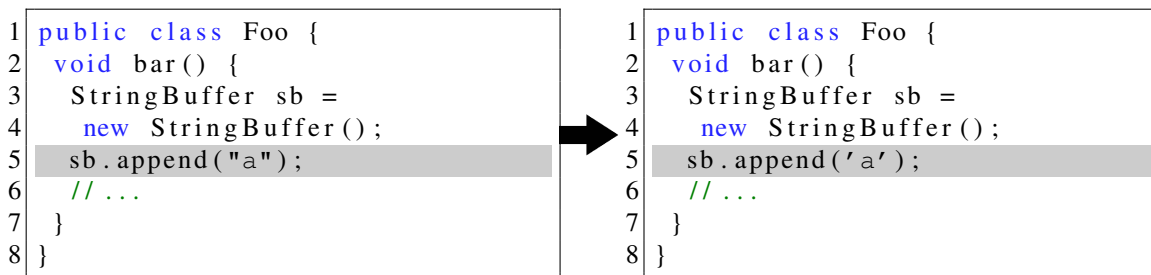
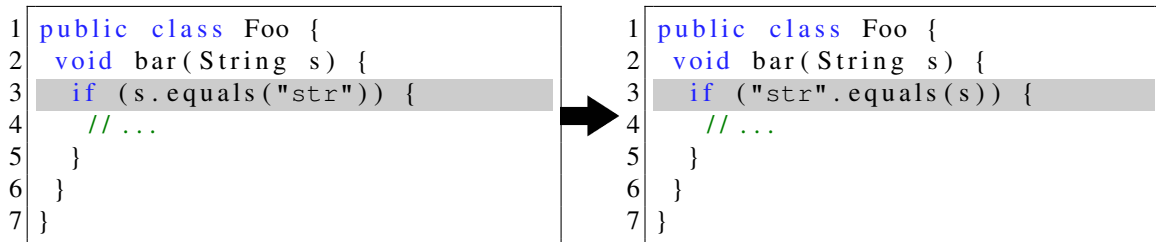


Figure B.1: Quick Fix 1: String to character

B.2 Prefer string literal equals method

Invoking `equals` on a null variable causes a `NullPointerException`. When comparing the value in a string variable to a string literal, programmers can overcome this exception by invoking the `equals` method on the string literal since the `String` `equals` method checks whether the parameter is null. Figure B.2 shows an edit that change a code snippet to use the `equals` method of a string literal.



```
1 public class Foo {
2   void bar(String s) {
3     if (s.equals("str")) {
4       // ...
5     }
6   }
7 }
```

```
1 public class Foo {
2   void bar(String s) {
3     if ("str".equals(s)) {
4       // ...
5     }
6   }
7 }
```

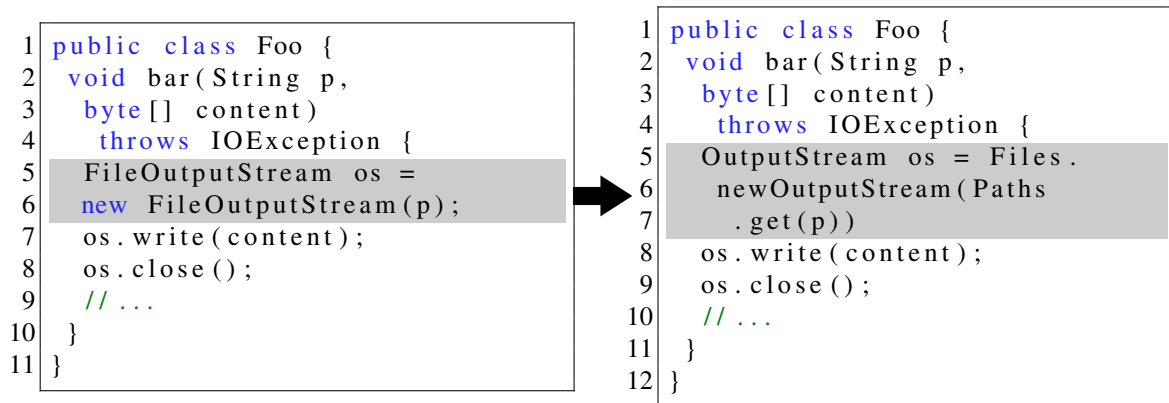
Figure B.2: Quick Fix 2: Prefer string literal equals method

B.3 Avoid using `FileInputStream/FileOutputStream`

These classes override the `finalize` method. As a result, their objects are only cleaned when the garbage collector performs a sweep [23]. Since Java 7, programmers can use `Files.newInputStream/Files.newOutputStream` to improve performance as recommended in this Java JDK bug-report [49]. Figure B.3 shows an example of an edit that changes a code snippet that creates an object of the type `FileOutputStream` to use the `Files.newOutputStream` method.

B.4 Use `valueOf` instead wrapper constructor

When creating a new `Integer`, one can use the `valueOf` method or the constructor. The method `valueOf` can improve performance since it caches frequently requested values [106]. This edit pattern is included in the Sonar catalog of rules. The same edit also applies for types such as `Byte`, `Short`, and `Long`, etc. Figure B.4 shows an example of an edit that changes a code snippet that uses the constructor of the wrapper of a primitive type to use the `valueOf`

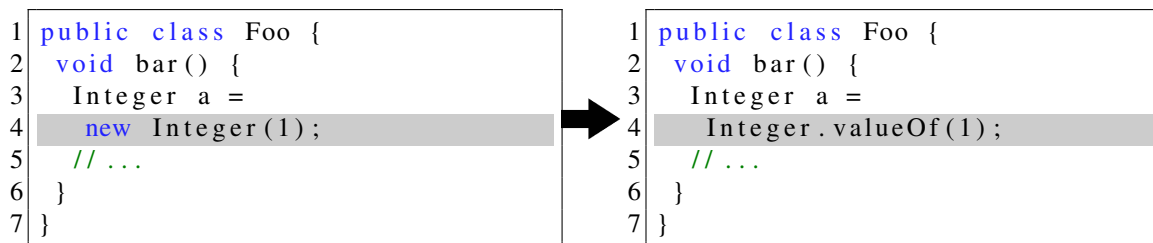


```
1 public class Foo {
2   void bar(String p,
3     byte[] content)
4     throws IOException {
5     FileOutputStream os =
6     new FileOutputStream(p);
7     os.write(content);
8     os.close();
9     // ...
10  }
11 }
```

```
1 public class Foo {
2   void bar(String p,
3     byte[] content)
4     throws IOException {
5     OutputStream os = Files.
6     newOutputStream(Paths
7     .get(p))
8     os.write(content);
9     os.close();
10    // ...
11  }
12 }
```

Figure B.3: Quick Fix 3: Avoid using FileInputStream/FileOutputStream

method. The example in this figure shows an edit in the variable of type `Integer`, but the same applies for other wrapper types such as `Float`, `Long`, `Double`, among others.



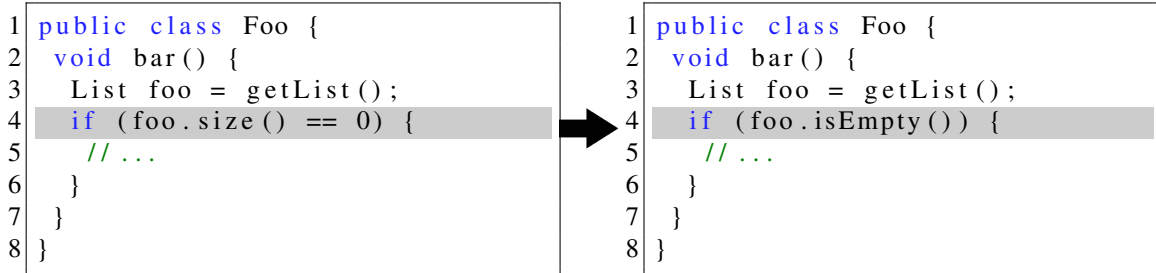
```
1 public class Foo {
2   void bar() {
3     Integer a =
4     new Integer(1);
5     // ...
6   }
7 }
```

```
1 public class Foo {
2   void bar() {
3     Integer a =
4     Integer.valueOf(1);
5     // ...
6   }
7 }
```

Figure B.4: Quick Fix 4: Use valueOf instead wrapper constructor

B.5 Use collection isEmpty

Using the method `isEmpty` to check whether a collection is empty is preferred to checking that the size of the collection is 0. For most collections these constructions are equivalent, but for others computing the size could be expensive. For instance, in the class `ConcurrentSkipListSet`, the `size` method is not constant-time [105]. This edit pattern is included in the PMD catalog of rules. Figure B.5 shows an example an edit that changes the an code snippet to use the `isEmpty` method.



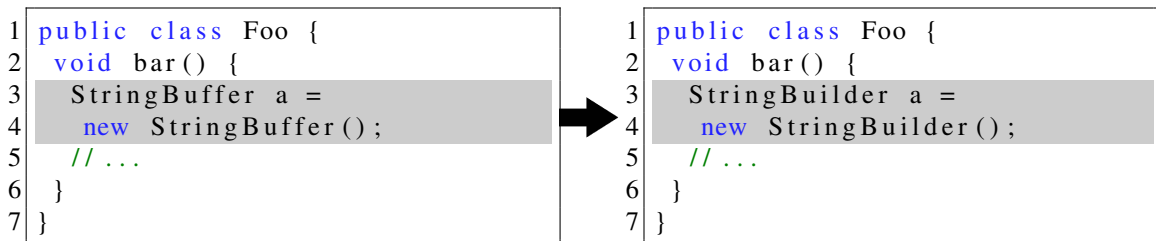
```
1 public class Foo {
2   void bar() {
3     List foo = getList();
4     if (foo.size() == 0) {
5       // ...
6     }
7   }
8 }

1 public class Foo {
2   void bar() {
3     List foo = getList();
4     if (foo.isEmpty()) {
5       // ...
6     }
7   }
8 }
```

Figure B.5: Quick Fix 5: Use collection isEmpty

B.6 StringBuffer to StringBuilder

These classes have the same API, but `StringBuilder` is not synchronized. Since synchronization is rarely used [107], `StringBuilder` offers high performance and is designed to replace `StringBuffer` in single threaded contexts [107]. Since this edit pattern requires knowing whether the applications uses synchronization, it does not appear in any code analyzer. Figure B.6 shows an example of an edit that changes the code from the use of the `StringBuffer` type to the use of the `StringBuilder` type.



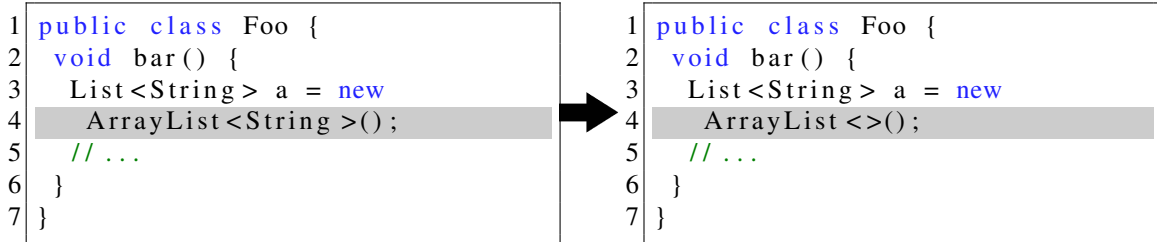
```
1 public class Foo {
2   void bar() {
3     StringBuffer a =
4     new StringBuffer();
5     // ...
6   }
7 }

1 public class Foo {
2   void bar() {
3     StringBuilder a =
4     new StringBuilder();
5     // ...
6   }
7 }
```

Figure B.6: Quick Fix 7: StringBuffer to StringBuilder

B.7 Infer type in generic instance creation

Since Java 7, programmers can replace type parameters to invoke the constructor of a generic class with an empty set (`<>`), diamond operator [108] and allow inference of type parameters by the context. This edit ensures the use of generic instead of the deprecated raw types [109]. The benefit of the diamond operator is clarity since it is more concise. Figure B.7 shows an example of an edit that change a code snippet to use the diamond operator in a variable declaration. Instead of using the type parameter `<String>`, programmers can use the diamond to invoke the constructor of `List` generic class.



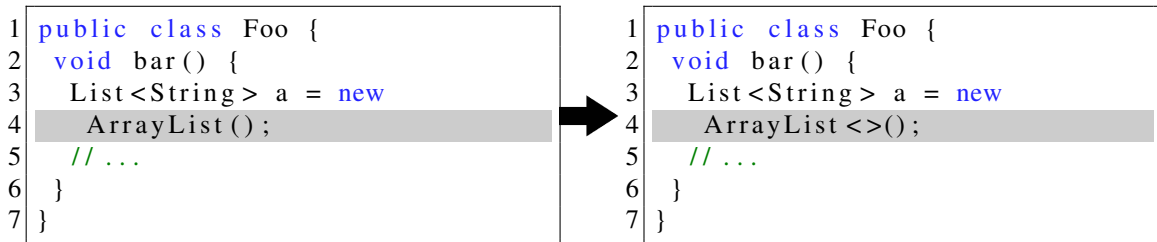
```
1 public class Foo {
2     void bar() {
3         List<String> a = new
4         ArrayList<String>();
5         // ...
6     }
7 }
```

```
1 public class Foo {
2     void bar() {
3         List<String> a = new
4         ArrayList<>();
5         // ...
6     }
7 }
```

Figure B.7: Quick Fix 7: Allow type inference for generic instance creation

B.8 Remove raw type

A raw type is a generic type without type parameters, which was used in version of Java prior to 5.0 and is allowed to ensure compatibility with pre-generics code. Since type parameters of raw types are unchecked, the catch of unsafe code is deferred to runtime [109] and the Java compiler issues warnings for them [109]. Figure B.8 shows an example of an edit that change the code to remove an use of a raw type.



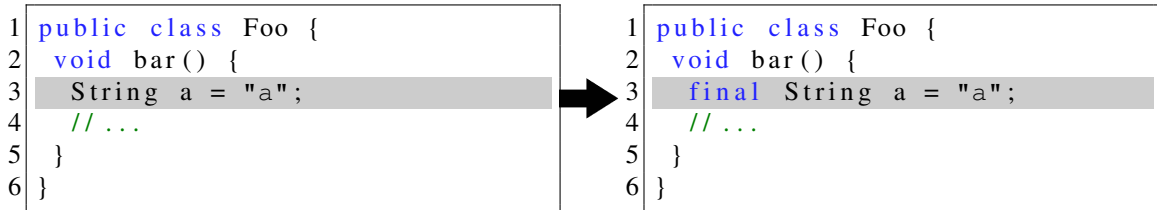
```
1 public class Foo {
2     void bar() {
3         List<String> a = new
4         ArrayList();
5         // ...
6     }
7 }
```

```
1 public class Foo {
2     void bar() {
3         List<String> a = new
4         ArrayList<>();
5         // ...
6     }
7 }
```

Figure B.8: Quick Fix 8: Remove raw type

B.9 Field, parameter, local variable could be final

The `final` modifier can be used in fields, parameters, and local variables to indicate they cannot be re-assigned [123]. This edit improves clarity and it helps with debugging since it shows what values will change at runtime. In addition, it allows the compiler and virtual machine to optimize the code [123]. The edit pattern that adds the `final` modifier is included in PMD catalog of rules [120]. IDEs such as Eclipse [35] and NetBeans [48] can be configured to perform this edit automatically on saving. Figure B.9 shows an example of an edit that adds the final modifier to a local variable. Variable `a` is assigned a single time, so it can be declared final.



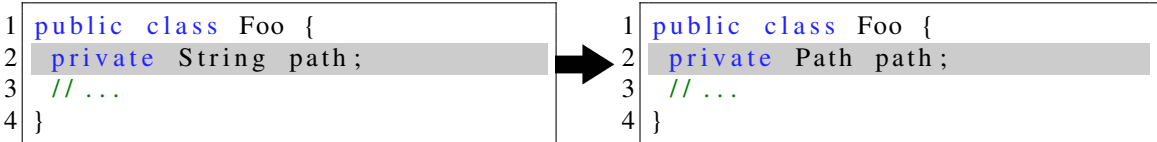
```
1 public class Foo {
2   void bar() {
3     String a = "a";
4     // ...
5   }
6 }
```

```
1 public class Foo {
2   void bar() {
3     final String a = "a";
4     // ...
5   }
6 }
```

Figure B.9: Quick Fix 9: Field, parameter, local variable could be final

B.10 Avoid using strings to represent paths

Programmers sometimes use `String` to represent a file system path even though some classes are specifically designed for this task—e.g., `java.nio.Path`. In this cases, it is useful to change the type of the variable to `Path`. First, strings can be combined in an undisciplined way, which can lead to invalid paths. Second, different operating systems use different file separators, which can cause bugs. Since this pattern is very hard to detect, code analyzers do not include it in their rules. Figure B.10 shows an example of an edit that changes a code snippet that uses a string to represent a path to the use of a specific class for this purpose.



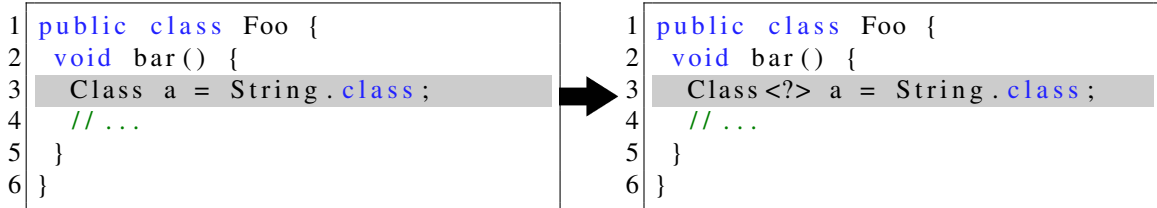
```
1 public class Foo {
2   private String path;
3   // ...
4 }
```

```
1 public class Foo {
2   private Path path;
3   // ...
4 }
```

Figure B.10: Quick Fix 10: Use `Path` to represent file path

B.11 Prefer `Class<?>`

Java prefers `Class<?>` over plain `Class` although these constructions are equivalent [24]. The benefit of `Class<?>` is clarity since programmers explicitly indicates that they are aware of not using a prior to Java 5 Java construction. The Java compiler generates a warning on the use of `Class`. Figure B.11 shows an edit that change the code to use `Class<?>`.



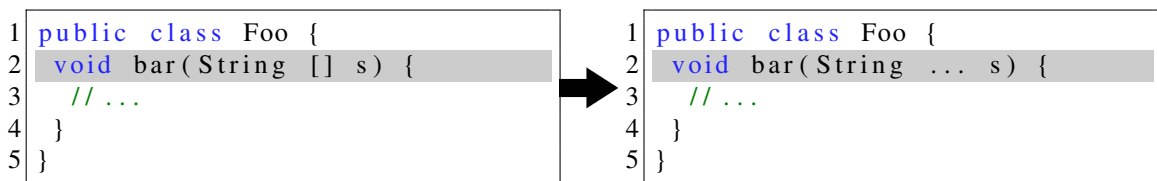
```
1 public class Foo {
2   void bar() {
3     Class a = String.class;
4     // ...
5   }
6 }
```

```
1 public class Foo {
2   void bar() {
3     Class<?> a = String.class;
4     // ...
5   }
6 }
```

Figure B.11: Quick Fix 11: Prefer `Class<?>`

B.12 Use variadic functions

Variadic functions denote functions that use variable-length arguments (varargs) [21]. This feature was introduced in Java 5 to indicate that the method receives zero or more arguments. Prior to Java 5, if a method receives a variable number of arguments, programmers have to create overload method for each number of arguments or to pass an array of arguments to the method. The benefit of using varargs is simplicity since programmers do not need to create overload methods and use the same notation independently of the number of arguments. For the compiler perspective, the method receives an array as parameter. Figure B.12 shows an example of an edit that change the code from the use of the array notation to use the use of the Variadic functions.



```
1 public class Foo {
2   void bar(String [] s) {
3     // ...
4   }
5 }
```

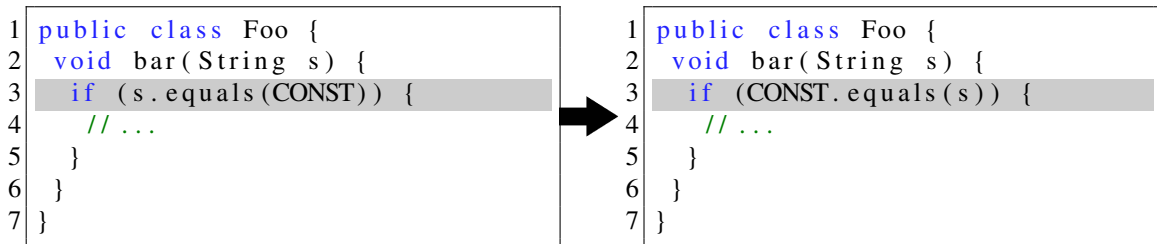
```
1 public class Foo {
2   void bar(String ... s) {
3     // ...
4   }
5 }
```

Figure B.12: Quick Fix 12: Use variadic functions

B.13 Prefer string constant equals method

The equals method is widely used in software development. Some usages can cause `NullPointerException` due to the right-hand side of the equals method object reference being `null`. When using the equals method to compare some variable to a string constant, programmers could overcome `null` point errors by allowing the string constant to call the equals method because a constant is rarely `null`. Since Java `String` equals method checks for `null`, we do not need to check for `null` explicitly when calling the

equals method of a string literal. Figure B.13 shows an example of an edit that change an code snippet to use the equals method of a string constant.



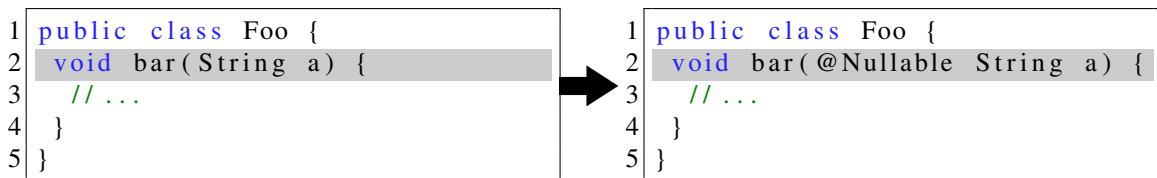
```
1 public class Foo {
2   void bar(String s) {
3     if (s.equals(CONST)) {
4       // ...
5     }
6   }
7 }
```

```
1 public class Foo {
2   void bar(String s) {
3     if (CONST.equals(s)) {
4       // ...
5     }
6   }
7 }
```

Figure B.13: Quick Fix 13: Prefer string literal equals method

B.14 Add @Nullable for parameters that accept null

In Java, programmers could annotate a parameter with `@Nullable` to indicate that the method, as well as all its overrides, accepts `null` for that parameter. This annotation helps code analyzer tools such as FindBugs and Checker Framework to better analyze the code [114]. Figure B.14 shows an example of an edit to a code snippet to use the `@Nullable` annotation.



```
1 public class Foo {
2   void bar(String a) {
3     // ...
4   }
5 }
```

```
1 public class Foo {
2   void bar(@Nullable String a) {
3     // ...
4   }
5 }
```

Figure B.14: Quick Fix 14: Add @Nullable for parameters that accept null

B.15 Use primitive type termination in literals

In Java, all non-float point literals are evaluated to `int` and all float point literals are evaluated to `float`. This behaviour could lead to numeric errors. For instance, consider this example took from a development forum (Figure B.15). In this example, `a` receives the value `-1846840301` and `b` receives the value `370370362962963` although they intent

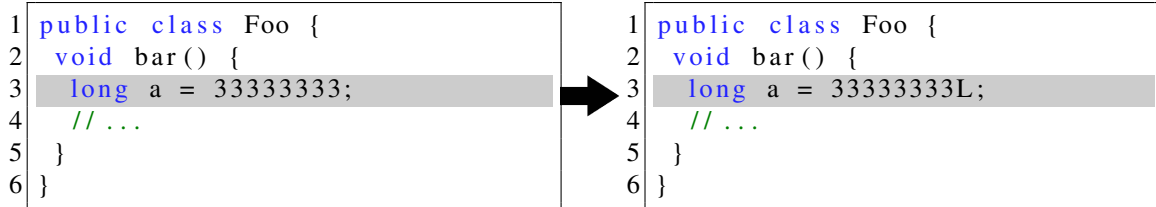
to represent the same value. Figure B.16 shows an example of an edit that change a code snippet to use the primitive type letter termination.

```

1 long a = 33333333 * 11111111; // overflows
2 long b = 33333333L * 11111111L;
3 System.out.println("a= "+new BigDecimal(a));
4 System.out.println("b= "+new BigDecimal(b));
5 System.out.println("a == b is " + (a == b));

```

Figure B.15: An example of a numeric problem associated with numeric literal use.



```

1 public class Foo {
2 void bar() {
3 long a = 333333333;
4 // ...
5 }
6 }

```

```

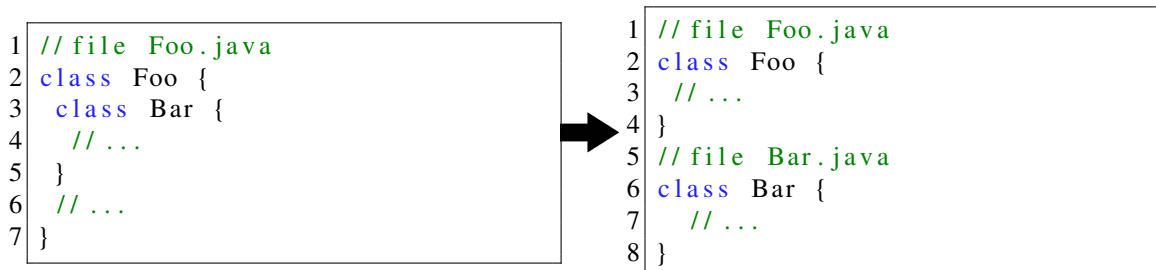
1 public class Foo {
2 void bar() {
3 long a = 333333333L;
4 // ...
5 }
6 }

```

Figure B.16: Quick Fix 15: Use primitive type termination in literals

B.16 Promote inner class

Inner classes are an important feature of Java. As a design principle classes should be declared final. If programmers want to re-use an inner class, it may be the case to promote the inner class to a regular class. Figure B.17 shows an example that removes the use an inner class to use two distinct classes.



```

1 // file Foo.java
2 class Foo {
3 class Bar {
4 // ...
5 }
6 // ...
7 }

```

```

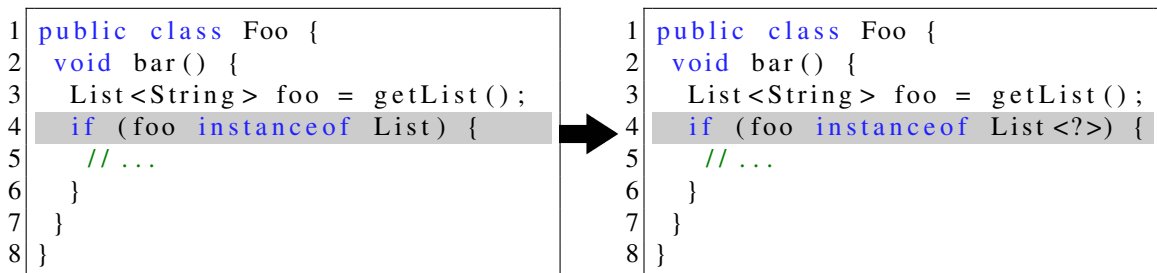
1 // file Foo.java
2 class Foo {
3 // ...
4 }
5 // file Bar.java
6 class Bar {
7 // ...
8 }

```

Figure B.17: Quick Fix 16: Promote inner class

B.17 Cannot use casts or instanceof with parameterized types

Java compiler erases all type parameters in generic code. Therefore, it is not possible to verify for a generic type which parameterized type is being used at run-time. Trying use `instanceof` to check a generic type will generate a compiler-time error. To perform this verification, Java recommends the use of an unbounded wildcard `<?>` to verify the generic type. Programmer also could use a raw type instead of an unbounded wildcard `<?>`. Figure B.18 shows an example an edit to a code snippet that changes a code snippet that uses `instanceof` with a raw type to use the unbounded wildcard.



```
1 public class Foo {
2     void bar() {
3         List<String> foo = getList();
4         if (foo instanceof List) {
5             // ...
6         }
7     }
8 }
```

```
1 public class Foo {
2     void bar() {
3         List<String> foo = getList();
4         if (foo instanceof List<?>) {
5             // ...
6         }
7     }
8 }
```

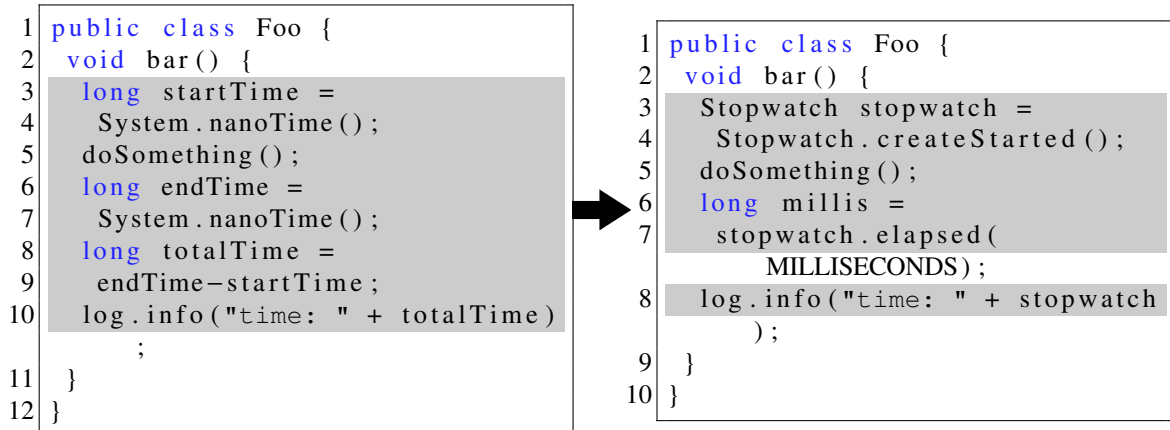
Figure B.18: Quick Fix 17: Use collection isEmpty

B.18 Use Stopwatch

Java programmers often measure time using `System.nanoTime()`. An alternative approach for this task is to use a `Stopwatch`. The latter has some benefits: programmers can substitute the source that measures time to improve performance. Second, the result of a `Stopwatch` is more interpretable than the `System.nanoTime`, which meaning can only be understood when compared to another call to `System.nanoTime` [40]. Figure B.19 shows an example of an edit that change a code snippet to use a `Stopwatch`.

B.19 Use HH in SimpleDateFormat

We could configure a `SimpleDateFormat` in different ways. Some configurations may lead to bugs. For instance, it causes a bug in the Ant project [80]. In some module of



```

1 public class Foo {
2   void bar() {
3     long startTime =
4       System.nanoTime();
5     doSomething();
6     long endTime =
7       System.nanoTime();
8     long totalTime =
9       endTime - startTime;
10    log.info("time: " + totalTime)
11    ;
12  }
}

```

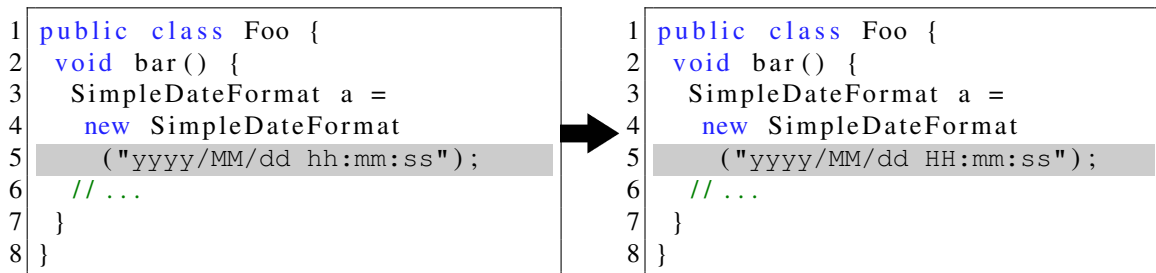
```

1 public class Foo {
2   void bar() {
3     Stopwatch stopwatch =
4       Stopwatch.createStarted();
5     doSomething();
6     long millis =
7       stopwatch.elapsed(
8         MILLISECONDS);
9     log.info("time: " + stopwatch
10    );
11  }
12 }

```

Figure B.19: Quick Fix 18: Use Stopwatch

this project, `SimpleDateFormat` uses the 12h-format (hh). As a result, sometimes the chronological order of revisions is broken since a revision at 14:20 (becomes 02:20), which is before a revision at 07:00. Change the `SimpleDateFormat` instance from hh:mm to 24h-format (HH:mm) fix the bug. Figure B.20 shows an example of an edit that changes a code snippet that uses the 12h-format to use the HH 24th-format.



```

1 public class Foo {
2   void bar() {
3     SimpleDateFormat a =
4     new SimpleDateFormat
5     ("yyyy/MM/dd hh:mm:ss");
6     // ...
7   }
8 }

```

```

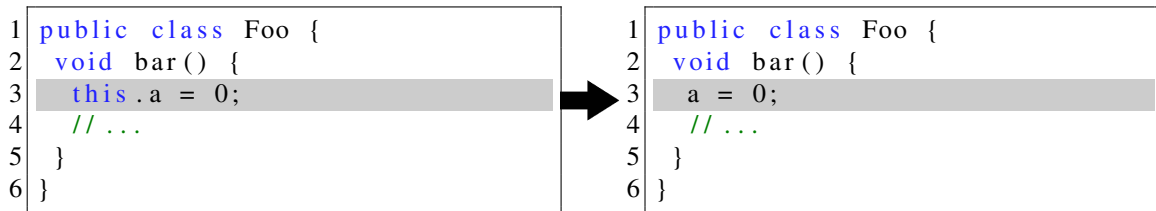
1 public class Foo {
2   void bar() {
3     SimpleDateFormat a =
4     new SimpleDateFormat
5     ("yyyy/MM/dd HH:mm:ss");
6     // ...
7   }
8 }

```

Figure B.20: Quick Fix 19: Use HH in SimpleDateFormat

B.20 Remove this for unambiguous variables

Basically, the `this` keyword could be used in three situations (i) in a get and set method when the method already defines a parameter with the same name (ii) when it is need to pass the current version of the object for other classes (iii) when it is need to call alternative constructor of a class hierarchy [115]. Out of these three situations the `this` keyword could be removed. Figure B.21 shows an example of an edit to remove the `this` keyword.



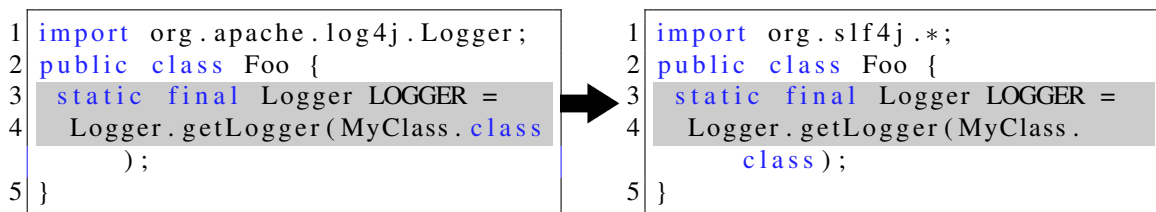
```
1 public class Foo {
2 void bar() {
3 this.a = 0;
4 // ...
5 }
6 }

1 public class Foo {
2 void bar() {
3 a = 0;
4 // ...
5 }
6 }
```

Figure B.21: Quick Fix 20: Remove this for unambiguous variables.

B.21 Use modern log framework instead of Log4J

The Log4j project has been deprecated. Instead of using this logging framework, it is suggested to use modern frameworks such as SLF4J and Logback [116]. Figure B.22 shows an edit that changes the usage of Log4j to use SLF4J framework.



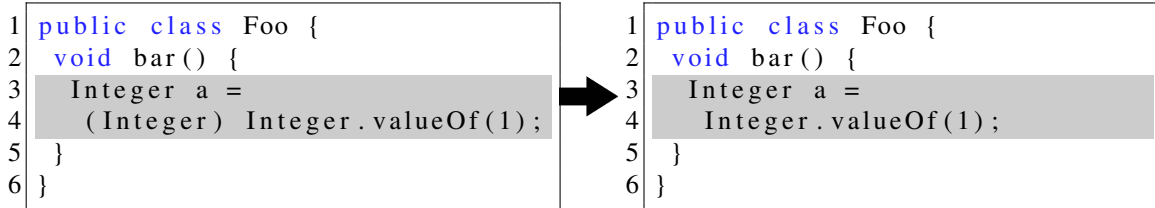
```
1 import org.apache.log4j.Logger;
2 public class Foo {
3 static final Logger LOGGER =
4 Logger.getLogger(MyClass.class);
5 }

1 import org.slf4j.*;
2 public class Foo {
3 static final Logger LOGGER =
4 Logger.getLogger(MyClass.class);
5 }
```

Figure B.22: Quick Fix 21: Use modern log framework instead of Log4J.

B.22 Remove redundant cast

Some casts could be unnecessary and could be removed to make the source code more concise. Given the importance of this edit, IDEs such as Eclipse allow programmers to remove redundant cast automatically. Figure B.23 shows an example of an edit to remove a redundant cast.



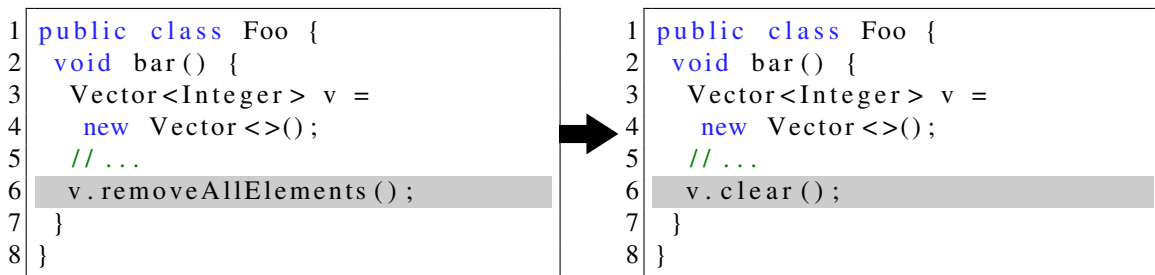
```
1 public class Foo {
2 void bar() {
3 Integer a =
4 (Integer) Integer.valueOf(1);
5 }
6 }

1 public class Foo {
2 void bar() {
3 Integer a =
4 Integer.valueOf(1);
5 }
6 }
```

Figure B.23: Quick Fix 22: Remove redundant cast.

B.23 Use Vector methods compatible with Java Collections interface

Since Java 4, the `Vector` type defines methods with the same interface of the `List` type. Changing the code to use the method compatible with the `List` type improves maintenance of the source code. For instance, `Vector` offers the method `removeAllElements()` to remove all the elements in it, but it also provides the method `clear`, which is identical in functionality. In the future, if programmers decide to replace the use of the `Vector` to use `ArrayList`, the other locations will have to be modified accordingly when using `removeAllElements`, but no modification is needed when the code uses `clear`. Figure B.24 shows an edit to update the `Vector` method to use one compatible with the `List` type.



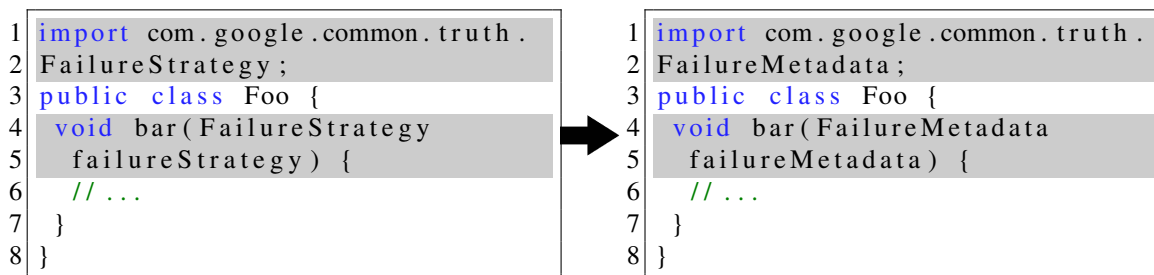
```
1 public class Foo {
2 void bar() {
3 Vector<Integer> v =
4 new Vector<>();
5 // ...
6 v.removeAllElements();
7 }
8 }

1 public class Foo {
2 void bar() {
3 Vector<Integer> v =
4 new Vector<>();
5 // ...
6 v.clear();
7 }
8 }
```

Figure B.24: Quick Fix 23: Use Vector methods compatible with Java Collections interface.

B.24 Replace Google Truth.FailureStrategy to Google Truth.FailureMetadata

In Google Truth, the `FailureMetadata` was designated to replace the use of `FailureStrategy`. The use of `FailureMetadata` removes some of the problems in `FailureStrategy`. It also enables new features introduced in Google Truth. Figure B.25 shows an edit that replaces the use of `FailureStrategy` to `FailureMetadata`.



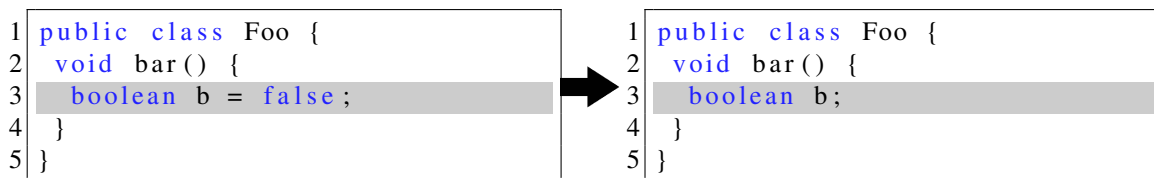
```
1 import com.google.common.truth .
2 FailureStrategy;
3 public class Foo {
4 void bar(FailureStrategy
5 failureStrategy) {
6 // ...
7 }
8 }
```

```
1 import com.google.common.truth .
2 FailureMetadata;
3 public class Foo {
4 void bar(FailureMetadata
5 failureMetadata) {
6 // ...
7 }
8 }
```

Figure B.25: Quick Fix 24: Replace Google Truth.FailureStrategy to Google Truth.FailureMetadata.

B.25 Remove default value for default initializer

The types in Java have default initializers. For instance, when we have an object of type `boolean`, it is initialized by default with `false`. We can also initialize a `boolean` variable explicitly with a `false` value, but as the type already has its default value as `false`, the explicit initialization is redundant and should be removed. Figure B.26 shows an example of an edit to remove the value, which is already provided by the compiler.



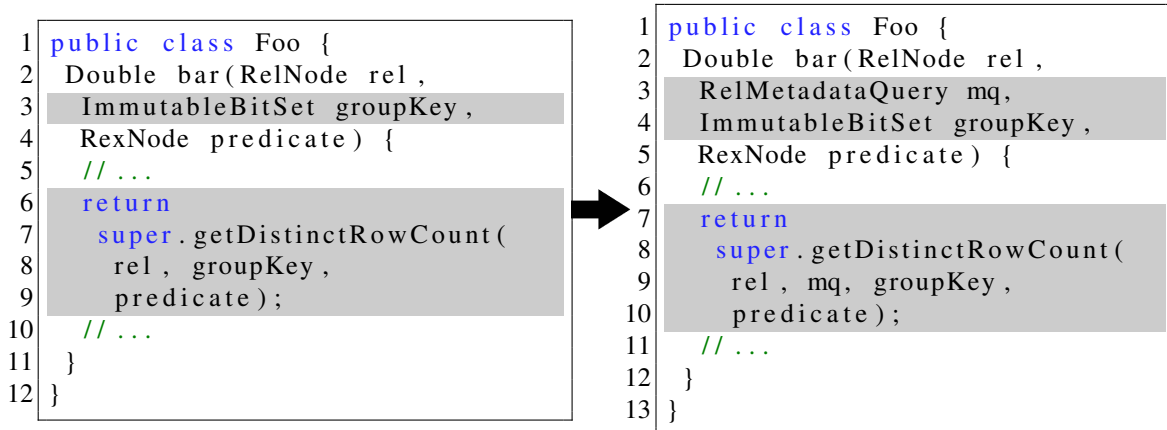
```
1 public class Foo {
2 void bar() {
3 boolean b = false;
4 }
5 }
```

```
1 public class Foo {
2 void bar() {
3 boolean b;
4 }
5 }
```

Figure B.26: Quick Fix 25: Remove default value for default initializer.

B.26 Update Calcite library

The versions of Calcite previous to 1.4.0 may cause stack overflow error. This bug is caused by some method in this library that calls itself recursively. To fix this bug programmer need to update this library to 1.4.0 or later version.



```
1 public class Foo {
2   Double bar(RelNode rel ,
3     ImmutableBitSet groupKey ,
4     RexNode predicate) {
5     // ...
6     return
7     super.getDistinctRowCount(
8       rel , groupKey ,
9       predicate);
10    // ...
11  }
12 }
```

```
1 public class Foo {
2   Double bar(RelNode rel ,
3     RelMetadataQuery mq,
4     ImmutableBitSet groupKey ,
5     RexNode predicate) {
6     // ...
7     return
8     super.getDistinctRowCount(
9       rel , mq, groupKey ,
10      predicate);
11    // ...
12  }
13 }
```

Figure B.27: Quick Fix 26: Update Calcite library.

B.27 Use trace instead of debug, when want to log precise data

When we are logging the application with Log4J, SLF4J, and other logging frameworks, we can decide to use the debug or trace options. The trace option provides more data about problems that could occur in the source code. It could be helpful to identify the presence of some bugs. However, it has a negative impact on the performance since we have more data to store and to process. Therefore, when analyzing bugs in the source code, it could be helpful to use the trace option instead of the debug option. Figure B.28 shows an example of an edit that replaces the use of the `debug` option to the use of the `trace` option.

```
1 public class Foo {
2 void bar() {
3 if (log.isDebugEnabled()) {
4 log.debug("some text");
5 }
6 }
7 }
```

```
1 public class Foo {
2 void bar() {
3 if (log.isTraceEnabled()) {
4 log.trace("some text");
5 }
6 }
7 }
```

Figure B.28: Quick Fix 27: Use trace instead of debug, when want to log precise data.

B.28 If an error is thrown, use *Throwable* instead of *Exception*

In Java, we can have problems caused by an `Error` or an `Exception`. Therefore, it needs to deal with them appropriately. When we also want to capture errors besides exception, we should use the specific types or use general classes such as `Throwable`, which captures both errors and exceptions. Figure B.29 shows an edit to replace the use of an `Exception` to use `Throwable`.

```
1 public class Foo {
2 void bar() {
3 try {
4 // ...
5 } catch (Exception e) {
6 // ...
7 }
8 }
9 }
```

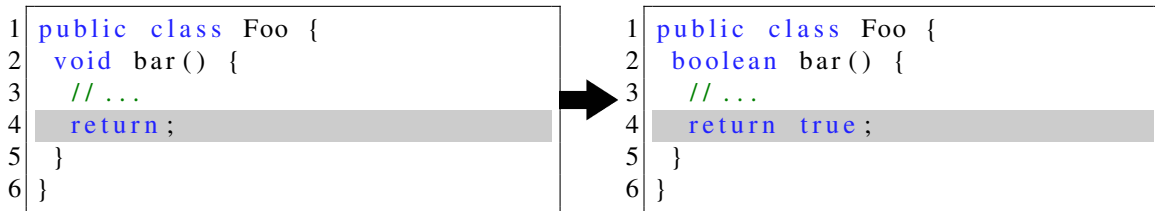
```
1 public class Foo {
2 void bar() {
3 try {
4 // ...
5 } catch (Throwable e) {
6 // ...
7 }
8 }
9 }
```

Figure B.29: Quick Fix 28: If an error is thrown, use `Throwable` instead of `Exception`.

B.29 Return a `boolean` instead of `void`

In cases of method that returns nothing (`void`), it may be desirable to return a `boolean` status of the result of running the method (i.e., `true` in case the method executes successfully

and `false` in case we have some problem). Figure B.30 shows an example of an edit to return a `boolean` instead of `void`.



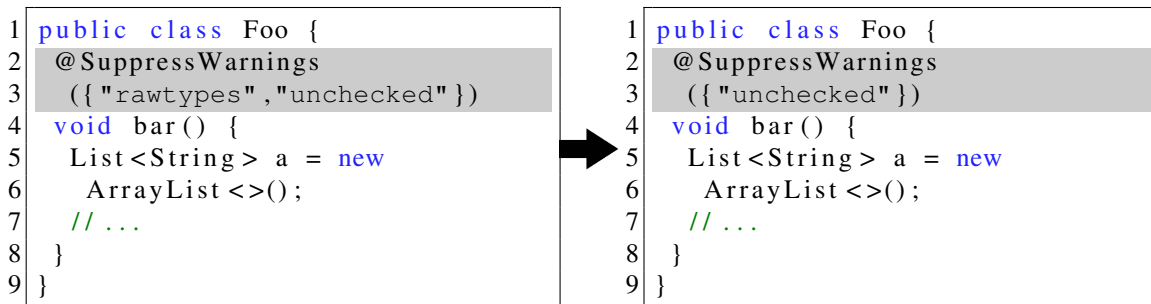
```
1 public class Foo {
2 void bar() {
3 // ...
4 return;
5 }
6 }
```

```
1 public class Foo {
2 boolean bar() {
3 // ...
4 return true;
5 }
6 }
```

Figure B.30: Quick Fix 29: Return a `boolean` instead of `void`.

B.30 Remove non-reachable @SuppressWarnings

In Java, we could use the `@SuppressWarnings` annotation to control the warnings generated by the compiler. As the software evolves, the code that caused the compiler to generate some warning could end up being removed. Therefore, this annotation needs to be removed to reflect the state of the code. Figure B.31 shows an edit to remove the `@SuppressWarnings` annotation.



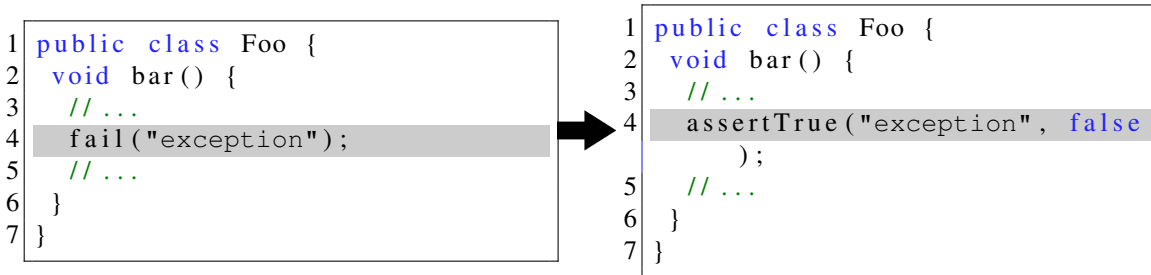
```
1 public class Foo {
2 @SuppressWarnings
3 ({ "rawtypes", "unchecked" })
4 void bar() {
5 List<String> a = new
6 ArrayList<>();
7 // ...
8 }
9 }
```

```
1 public class Foo {
2 @SuppressWarnings
3 ({ "unchecked" })
4 void bar() {
5 List<String> a = new
6 ArrayList<>();
7 // ...
8 }
9 }
```

Figure B.31: Quick Fix 30: Remove non-reachable `@SuppressWarnings`

B.31 Use `fail` instead of `assertTrue(false)`

Programmers can use both `fail` or `assertTrue(false)` to evaluate a test. These constructions are equivalent in functionality, but `fail` describes the intent more clearly. Figure B.32 shows an edit to use `fail` instead of `assertTrue(false)`.



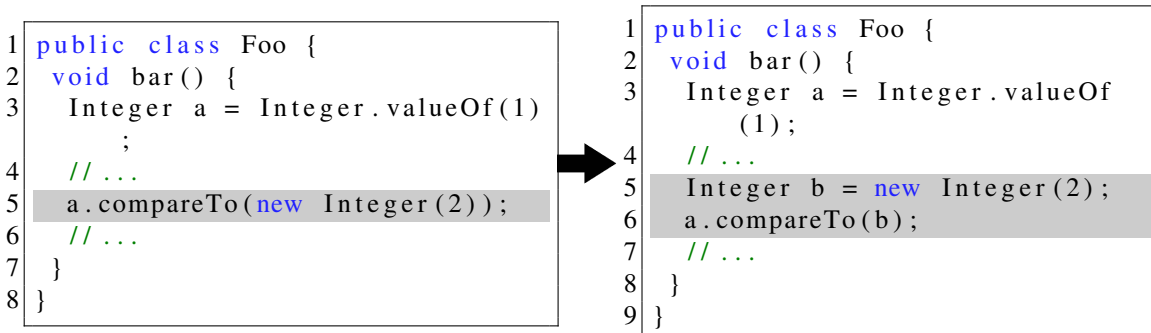
```
1 public class Foo {
2   void bar() {
3     // ...
4     fail("exception");
5     // ...
6   }
7 }
```

```
1 public class Foo {
2   void bar() {
3     // ...
4     assertTrue("exception", false);
5     // ...
6   }
7 }
```

Figure B.32: Quick Fix 31: If an error is throw, use `Throwable` instead of `Exception`.

B.32 Do not use new as argument of a method call.

We could both create a new instance of a class and reuse this instance along the code or we use the new operators without storing them in a variable. The first options improve readability of the code by making it more concise. Figure B.33 shows an edit to use an object stored in a variable.



```
1 public class Foo {
2   void bar() {
3     Integer a = Integer.valueOf(1);
4     // ...
5     a.compareTo(new Integer(2));
6     // ...
7   }
8 }
```

```
1 public class Foo {
2   void bar() {
3     Integer a = Integer.valueOf(1);
4     // ...
5     Integer b = new Integer(2);
6     a.compareTo(b);
7     // ...
8   }
9 }
```

Figure B.33: Quick Fix 32: Do not use `new` as argument of a method call.

Appendix C

Survey

You are being invited to participate in this survey because you have contributed in some way to a public repository. To answer this survey, you should take around 5-10 minutes. The goal of this survey is to analyze developers' perception of small-grained code patterns in source code. A code pattern denotes a usage that improves some aspects of the source code, such as enhance code style, remove error-prone code, and increase performance.

It consists of three parts:

- (i) Choose the code snippet that you prefer (10 questions)
- (ii) Participant Background (4 questions)
- (iii) Additional Comments (2 questions)

All the data from this survey is anonymous and we might report them in academic publications.

Section 1

1 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        StringBuffer sb = new StringBuffer();  
        sb.append("a");  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar() {  
        StringBuffer sb = new StringBuffer();  
        sb.append('a');  
        // blah  
    }  
}
```

(B)

-
- Strongly prefer (A)
 - Prefer (A)
 - It does not matter
 - Prefer (B)
 - Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

2 - Which code snippet do you prefer?

```
public class Foo {  
    void bar(String str) {  
        if (str.equals("string")) {  
            // blah  
        }  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar(String str) {  
        if ("string".equals(str)) {  
            // blah  
        }  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

3 - Which code snippet do you prefer?

```
public class Foo {  
    public void bar(String path,  
        byte[] content) throws IOException {  
        try (OutputStream os = Files.  
            newOutputStream(Paths.get(path))) {  
            os.write(content);  
        }  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    public void bar(String path,  
        byte[] content) throws IOException {  
        try (FileOutputStream os = new  
            FileOutputStream(path)) {  
            os.write(content);  
        }  
        // blah  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

4 - Which code snippet do you prefer?

- Strongly prefer (A)


```
public class Foo {  
    void bar() {  
        Integer a = new Integer(1);  
        // blah  
    }  
}
```

OR

```
public class Foo {  
    void bar() {  
        Integer a = Integer.valueOf(1);  
        // blah  
    }  
}
```

(A)

(B)

- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

5 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        List foo = getList();  
        if (foo.size() == 0) {  
            // blah  
        }  
    }  
}
```

OR

```
public class Foo {  
    void bar() {  
        List foo = getList();  
        if (foo.isEmpty()) {  
            // blah  
        }  
    }  
}
```

(A)

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter

- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

6 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        StringBuilder a = new StringBuilder();  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar() {  
        StringBuffer a = new StringBuffer();  
        // blah  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

7 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        Map<String, List<String>> a = new  
        HashMap<>();  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar() {  
        Map<String, List<String>> a = new  
        HashMap<String, List<String>>();  
        // blah  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

8 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        List<Integer> integers = new  
        LinkedList<>();  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar() {  
        List<Integer> integers = new  
        LinkedList();  
        // blah  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)

- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

9 - Which code snippet do you prefer?

```
public class Foo {  
    void bar() {  
        final String a = "a";  
        // blah  
    }  
}
```

(A)

OR

```
public class Foo {  
    void bar() {  
        String a = "a";  
        // blah  
    }  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

10 - Which code snippet do you prefer?

```
public class Foo {  
    private String path;  
    // blah  
}
```

(A)

OR

```
public class Foo {  
    private Path path;  
    // blah  
}
```

(B)

- Strongly prefer (A)
- Prefer (A)
- It does not matter
- Prefer (B)
- Strongly prefer (B)

Please, let us know if you have any additional comments about this Java code pattern.

Section 2 — Background Information

11 - How many of the code snippets shown in questions 1-10 have you used in your code before?

- 0
- 1-2
- 3-5

6-10

12 - How would you rate your knowledge of the effects caused by using the different code snippets (i.e., (A) vs (B)) shown in questions 1-10?

Very high

High

Average

Very low

13 - How long have you been working/have worked with Java?

Less than a year

1-2 years

3-5 years

More than five years

14 - Do you use any tool(s) to identify opportunities of using code patterns shown in snippet (A) or (B) in your code? Which one(s)?

Checkstyle

PMD

FindBugs

Sonar

Coverity

ErrorProne

IntelliJ

Other

Section 3 — Additional Comments

15 - Please, let us know if you have any additional comments about Java code patterns.

16 - If you would like to receive the results of our survey, please include your email address.
