UNIVERSITY OF CALIFORNIA

Los Angeles

Leveraging Program Commonalities and Variations for Systematic Software Development and Maintenance

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Tianyi Zhang

2019

ABSTRACT OF THE DISSERTATION

Leveraging Program Commonalities and Variations for Systematic Software Development and Maintenance

by

Tianyi Zhang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Miryung Kim, Chair

Software is becoming increasingly pervasive and complex. During software development and maintenance, developers often make ad hoc decisions based on local program contexts and their own experience only, which may increase technical debt and raise unknown consequences as software evolves. This dissertation explores several opportunities to guide software development and maintenance by exploiting and visualizing the commonalities and variations among similar programs. Our hypothesis is that unveiling what has and has not been done in other similar program contexts can help developers make more systematic decisions, explore implementation alternatives, and reduce the risk of unknown consequences.

The inherent repetitiveness in software systems provides a solid foundation for identifying and exploiting similar code. This dissertation first presents two approaches that leverage the syntactic similarity and repetitiveness in local codebases to improve code reviews and software testing. First, in contrast to inspecting local program edits line by line, Critics enables developers to reason about related edits scattered across multiple files by summarizing similar edits and detecting inconsistencies in similar locations. Second, Grafter boosts test coverage by transplanting test cases between similar programs and helps developers discern the behavioral similarity and differences between these programs via differential testing.

This dissertation further extends the idea of systematic software development from local codebases to *the open world*, by exploiting the large and growing body of successful open-source projects on the Internet. In particular, we present three approaches that analyze a massive number of open-source projects and provide systematic guidance based on software reuse and adaptation patterns in the open-source community. First, ExampleCheck mines common API usage patterns from 380K GitHub projects and contrasts online code examples with these patterns to alert developers about potential API misuse during opportunistic code reuse. Second, to help developers comprehensively understand the variety of API usage at scale, Examplore aligns and super-imposes hundreds of code examples into a single code skeleton with statistical distributions of distinct API usage features among these examples. Finally, ExampleStack guides developers to adapt a code example to a target project by detecting similar code fragments in GitHub and by illuminating possible variations of this example with respect to its GitHub counterparts.

The viability and usability of these techniques are evaluated by their application to large-scale projects as well as within-subjects user studies. First, in a user study with twelve Java developers, Critics helps developers identify 47% more edit mistakes with 32% less time than a line-level program diff tool. Second, Grafter transplants test cases in three open-source projects with 94% success rate and improves the statement and branch coverage by 22% and 16% respectively. Third, ExampleCheck reduces the risk of bug propagation by detecting API usage violations in almost one third of code examples in a popular Q&A forum, Stack Overflow. These API usage violations can potentially lead to software anomalies such as program crashes and resource leaks in target programs. Fourth, in a study with sixteen Java developers, Examplore helps developers understand the distribution of API usage patterns and answer common API usage questions accurately with concrete details. Finally, in another study with sixteen Java developers, ExampleStack helps developers identify more adaptation opportunities about code safety and logic customization, rather than shallow adaptations such as variable renaming. These key findings demonstrate that discovering and representing the commonalities and variations in similar contexts helps a developer achieve better program comprehension, discover more design alternatives, and capture more errors in different software development and maintenance tasks.

The dissertation of Tianyi Zhang is approved.

Todd D. Millstein

Jens Palsberg

Westley Weimer

Miryung Kim, Committee Chair

University of California, Los Angeles

2019

To my mom and dad

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

When I looked back at my PhD, I felt nothing but humble and grateful. Six years is a long time. Yet I still remembered all the ups and downs along the path, as well as all the encouragement and support from others. I felt extremely lucky to be surrounded by so many brilliant minds and to be given so many opportunities to grow and learn during this journey.

I would like to express my deepest gratitude to my PhD advisor, Miryung Kim. Choosing Miryung as my advisor and later transferring to UCLA to continue working with her are two of the best decisions I ever made in my life. Miryung not only taught me how to do rigorous research, but also trained me to be a critical and independent thinker. I am also deeply indebted to her hands-on guidance on writing and presentation. Her encouragement and constructive criticism motivated me to continuously horn my skills and think deeply about my research. I am really grateful to Miryung's profound belief in my capability and unwavering support. From her, I learned how to be a thoughtful mentor and leader, which I wish to pass on to others in the future.

I would like to extend my sincere thanks to my collaborators, Elena Glassman, Björn Hartmann, Cristina Lopes, Mihir Mathur, Joseph Pinedo, Hridesh Rajan, Anastasia Reinhart, Aishwarya Sivaraman, Myoungkyu Song, Ganesha Upadyaya, Guy Van Den Broeck, Di Yang. I would not be able to finish all the work without their help and support. In particular, Myoungkyu led me through my first research project. I felt very grateful to have such a reliable mentor at the beginning of my PhD. Elena and Björn brought me into the HCI community, which significantly expanded my horizons and opened up new opportunities. I really appreciate their feedback from the human factors perspective.

Many thanks to my thesis committee, Todd Millstein, Jens Palsberg, and Westley Weimer. Thank you for those offline discussions and spending substantial time reading my dissertation, attending talks, and giving valuable and constructive suggestions.

I did a wonderful internship at Microsoft Research in Summer 2015. I must thank my three mentors, Mike Barnett, Christian Bird, Shuvendu Lahiri for the internship opportunity, as well as their continuous support after my internship.

I very much appreciate several people for their continuous advice and support through my PhD. I wish to thank Todd Millstein for his help and care since I moved to UCLA. I often solicited a second opinion from Todd when I felt confused and indecisive about research and life. I wish to thank Na Meng, Baishakhi Ray, Lingming Zhang, Yuqun Zhang, and James Xi Zheng for their advice on both my research and career plan at different stages of my PhD. I wish to thank Pei Wu, Yixue Zhao, and Xinyu Wang for always listening and sharing.

I really enjoyed discussing research ideas and hanging out with my lab mates and friends at UCLA, especially John Bender, Bobby Bruce, Brett Chalabian, Muhao Chen, Muhammad Ali Gulzar, Yuan He, Christian Kalhauge, Siva Kesava, Shaghayegh Mardani, Zeina Migeed, Saswat Padhi, Lana Ramji, Aishwarya Sivaraman, Jason Teoh, Pei Wu, Diyu Zhou. Special thanks to Bobby Bruce and Pei Wu for sharing comments on my defense talk and dissertation.

Though I only spent one year at UT Austin, I established close friendship with Jinru Hua, Rui Qiu, Kaiyuan Wang, Xingyu Wang, Lingming Zhang, Yuqun Zhang, James Xi Zheng. Our friendship continues to grow over the long distance after I moved to LA and never fades away. I also want to thank Professor Sarfraz Khurshid and Professor Christien Julien for their help and advice in the first year of my PhD. I could not quickly get acquainted to the drastically different life and culture in the US without the kind help from professors and friends at UT Austin.

I am very grateful to my parents for their constant support and encouragement. My dad always encourages me to jump out of my comfort zone and chase ambitious dreams, while my mom often reminds me to take it easy and enjoy life. This forms a harmonic balance that kept me moving forward without stressing out or slacking off during my PhD. I also want to thank Linghui Zhou for her love and support. When I pulled all-nighters before deadlines, she often stayed up with me and encouraged me. I could not go through those hard times without her encouragement and moral support.

Finally, I am really thankful for the sunny weather in Southern California as I often get depressed on rainy days. I truly am.

VITA

Sept. 2014–present	Teaching/Research Assistant, Computer Science Department, UCLA.
Jun 2015–Sept 2015	Research Intern, Microsoft Research, Redmond, WA
May 2014–Aug 2014	Quality Engineer Intern, Salesforce.com, San Francisco, CA
Aug 2013–May 2014	Teaching/Research Assistant, Department of Electrical and Computer Engineering, The University of Texas at Austin
Jul 2013	B.S. (Computer Science), Huazhong University of Science and Technology, Wuhan, China.
Jul 2012–Sept 2012	Software Development Engineer Intern, Microsoft, Beijing.

PUBLICATIONS

Analyzing and Supporting Adaptation of Online Code Examples. Tianyi Zhang, Di Yang, Crista Lopes, Miryung Kim. In Proceedings of the 41th International Conference on Software Engineering, 12 pages, IEEE, 2019

Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, Miryung Kim. In Proceedings of the 40th International Conference on Software Engineering, pages 886-896, ACM, 2018

Visualizing API Usage Examples at Scale. Elena Glassman∗, Tianyi Zhang∗, Bjoern Hartmann, Miryung Kim. In Proceedings of the 2018 CHI Conference on Human Factors in

Computing Systems, pages 580-591, ACM, 2018.
∗The two lead authors contributed equally to the work as part of an equal collaboration between both institutions.

Automated Transplantation and Differential Testing for Clones. Tianyi Zhang, Miryung Kim. In Proceedings of the 39th International Conference on Software Engineering, pages 665-676, IEEE, 2017.

Interactive Code Review for Systematic Changes. Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, Miryung Kim. In Proceedings of the 37th International Conference on Software Engineering, Volume 1, pages 111-122. IEEE, 2015.

Active Inductive Logic Programming for Code Search. Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, Miryung Kim. In Proceedings of the 41th International Conference on Software Engineering, 12 pages, IEEE, 2019

Augmenting Stack Overflow with API Usage Patterns Mined from GitHub. Anastasia Reinhardt, Tianyi Zhang, Mihir Mathur, Miryung Kim. In Proceedings of the 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Research Demonstration Track, ACM, 2018.

Grafter: Transplantation and Differential Testing for Clones. Tianyi Zhang, Miryung Kim. In Proceedings of the 40th International Conference on Software Engineering, Poster Track, pages 422-423, ACM, 2018.

Critics: An Interactive Code Review Tool for Searching and Inspecting Systematic Changes. Tianyi Zhang, Myoungkyu Song, Miryung Kim. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Research Demonstration Track, pages 755-758, ACM, 2014.

CHAPTER 1

Introduction

Building and maintaining software is a costly process. Developers need to tackle the complexity and uncertainty of software requirements while delivering highly functional software products on time. As software evolves and drifts away from its original design, developers need to spend significant maintenance effort to ensure that software quality does not degrade.

To reduce software development effort, developers tend to reuse existing software components rather than building them from scratch. Open-source software development has significantly enriched software reuse opportunities by making an enormous volume of source code and libraries available online [217]. One common form of software reuse is to copy and paste existing code fragments somewhere else to implement desired functionality [107, 133, 136, 167, 197, 208]. For instance, an analysis of 25 projects at NASA shows that 32% of modules in these projects are reused from prior projects [208]. Another common form of software reuse is to reuse mature and well-tested functionality exposed by library APIs. With the ever-increasing number of external libraries and the variety of functionality they provide, library APIs now become a fundamental building block of software systems and play a central role in modern software development.

Although software reuse has long been advocated for increasing development productivity and decreasing software defects [46,133,143], it is known to be undisciplined in practice, with ad hoc decisions about which software components (e.g., code fragments, library APIs) to reuse and how to reuse them. For instance, code reuse is often followed by specific adaptations of copied code to fit target scenarios [101,136,140,208]. Such adaptations are typically done manually and are thus prone to subtle edit mistakes, especially if developers do not fully understand the original code and its context. Previous studies show that copy-and-paste errors are one of the major causes of operating system bugs [53, 111, 140].

The problem is exacerbated when developers search and reuse code from the Web [48, 81, 202, 212, 238]. Since online code snippets are written for illustration purposes only, they are often incomplete and inadequate for production code. Some of these snippets even follow insecure coding practices [73, 161] or use deprecated APIs [268]. Hence, reusing online code snippets without proper adaptations can potentially decrease the quality of production code and cause unknown consequences. A recent study shows that 29% of security-related code snippets in a popular Q&A website, Stack Overflow, are insecure and have been potentially reused in over one million Android apps on Google Play, which raises a big security concern [73]. Ideally, developers would thoroughly examine the pros and cons of different code examples and pick the one that suits a target scenario the best. However, in practice, developers only examine a handful of these examples and return to their own code due to limited time and attention [48, 49, 218]. Currently, there is no easy way for developers to understand the gist of different code examples at scale.

Both forms of software reuse intentionally and unintentionally result in a large portion of highly similar code in software systems, e.g., 29% of lines of code in JDK, 23% in Linux, which is known as code clones [23, 36, 43, 121, 154, 199]. The existence of code clones significantly increases maintenance effort as software evolves: developers must keep track of program changes in similar locations, port critical patches as needed, and also avoid unintentional inconsistencies. Ray et al. analyze software patches in the family of BSD operating systems and find that over 10% of patches in each system are ported from other systems [187]. Such edits are often similar but not identical due to subtle variations (e.g., variable renaming) in their program contexts. Manually applying these edits are tedious and error-prone. Juegens et al. inspect 1427 clone groups in five large systems and find that over half of these clones contain inconsistent edits, among which over a quarter are introduced unintentionally and thus lead to bugs, confirmed by developers of these systems [114].

Developers cannot easily reason about similar edits scattered across multiple files or identify unintentional inconsistencies due to a lack of tool support. Existing code review tools only compute line-level differences per file. As a result, developers can only inspect program edits line by line and focus on local contexts, without a global understanding about missing edits or subtle inconsistencies in other similar locations [64]. We analyze 52 clone pairs in three open-source projects and find that, in 46% of clone pairs, only one clone is tested, while its counterpart is not. In absence of test cases, there is no easy way to examine the behavioral consistency of these similar locations.

This dissertation seeks to reduce the risk of ad hoc decisions and avoid unintentional inconsistencies by exploiting and visualizing the commonalities and variations among similar programs. We believe that by unveiling what has and has not been done in similar program contexts, developers can gain a deeper understanding about the program under investigation, make more systematic decisions, and avoid subtle edit mistakes during software development and maintenance.

1.1 Thesis Statement

This thesis draws inspiration from Linus's law, which states that "*given enough eyeballs, all bugs are shallow*" [190]. In other words, presenting the same code to multiple developers can effectively detect software defects and improve code quality. However, there is a lack of "qualified eyeballs"—developers may not have a deep understanding of the source code and may not audit every line of code due to limited attention.

In this dissertation, we pose and explore a complementary hypothesis to Linus's law—*presenting multiple similar programs to the same developer can enhance program comprehension, avoid potential inconsistencies, and identify better implementation alternatives in software development.* This hypothesis is also supported by learning theories in cognitive science, such as analogical learning [134] and variation theory [153]: displaying and contrasting multiple examples prompts re-evaluation of a human subject's own beliefs and understanding of a concept, and helps the human subject generalize the concept to new contexts.

Both the inherent repetitiveness in local codebases and the large body of open-source projects provide many opportunities for identifying and exploiting similar programs. In particular, we focus on two kinds of similar programs—code fragments with similar syntactic structures and code fragments with similar API usage. To verify the overarching hypothesis, we design five intelligent systems that discover and represent the commonalities and variations among similar programs, and investigate five separate research questions in three software development and maintenance tasks—code reviews, software testing, and opportunistic code reuse.

RQ1: Can we improve code review effectiveness by precisely identifying similar program locations and detecting edit inconsistencies among these locations?

RQ2: Can we improve testing effectiveness by reusing test cases among similar code fragments and examining behavioral similarities and differences among these fragments?

RQ3: Can we efficiently mine representative API usage patterns from massive code corpora and help developers identify potential API misuse in online code examples?

RQ4: Can we help developers grasp a comprehensive view of using an API by visualizing commonalities and variations among hundreds of API usage code examples at scale?

RQ5: Can we enable developers to identify adaptation opportunities and write complete and robust code by displaying and contrasting similar code in GitHub?

As analogical learning [134] and variation theory [153] point out, it is necessary to display sufficient and unbiased variations among concrete examples to help human subjects generalize a concept and understand what may vary. However, superficial differences among those examples may in turn hinder the human capability of drawing connections between these examples. There are similar challenges when presenting multiple similar programs to developers. These challenges are exacerbated when analyzing and representing a massive number of relevant code examples.

- *How do we define an abstraction of concrete programs that accurately represent desired program properties while abstracting away superficial syntactic details?* In this dissertation, we design different program abstractions, including interactive change templates for summarizing similar edits, de-facto interfaces of code clones for test transplantation, and structured API call sequences and code skeletons for opportunistic code reuse, in order to flexibly express different program properties of interest (e.g., program structures, internal program states, API usage features).

- *How do we design efficient algorithms to distill the essence of similar programs while preserving meaningful variations to detect potential inconsistencies and recommend implementation alternatives?* We use control-flow and data-flow analysis to identify relevant program features that we need to match and contrast, and to eliminate irrelevant ones to reduce both the computational and visual complexity. On top of that, we leverage AST-based tree matching, fine-grained differential testing, data mining and clustering techniques to identify commonalities and variations among similar programs while accommodating different program abstractions.

- *How do we intuitively represent a large number of relevant code examples to help developers easily understand the essence of similar programs and their variations at scale?* To scale to massive code corpora such as GitHub, we leverage a distributed software mining infrastructure and a scalable clone detector to identify similar programs. We further design efficient user-centric visualizations to illuminate commonalities and variations among the many similar programs and allow developers to drill down to concrete details via interaction. To address the variety of program expressions in massive code corpora, we canonicalize variable names and use a SMT solver to prove the semantic equivalence of different expressions.

In the next two sections, we will give an overview about the five techniques, propose sub-hypothesis for each work, and briefly explain how we verify each sub-hypothesis through evaluation. The first two techniques exploit syntactically similar code fragments in local codebases, while the other three techniques mine and analyze similar programs in a large collection of open-source projects.

1.2 Leveraging Inherent Repetitiveness in Software Systems

The effectiveness of software maintenance can be significantly improved by enabling developers to reason about similar edits scattered across multiple locations. We build two techniques and demonstrate the benefits of leveraging syntactic similar code fragments in two common software maintenance tasks—code reviews and software testing.

1.2.1 Interactive Code Review for Similar Program Edits

Peer code reviews require developers to manually inspect program changes to identify edit mistakes and optimization opportunities. However, the effectiveness of code reviews is much hindered when related program changes are scattered across multiple files [64]. Despite the proliferation of code review tools (e.g., Phabricator,[1] Gerrit,[2] CodeFlow,[3] Crucible,[4] Review Board[5]), all these tools compute line-level differences only and cannot enable developers to reason about similar, related program edits in multiple files. A case study of a large commercial software shows that 75% of program changes in a software revision can be classified as similar but not identical edits to multiple locations [125]. As a result, developers have to manually inspect these changes line by line to answer questions such as "what other code locations are changed similar to this change?" and "are there any other locations that are similar to this code but are not updated?"

We present Critics, an interactive approach for inspecting such similar edits during peer code reviews. Given a specified change, Critics extracts the surrounding control-flow and data-flow context of the change and creates a context-aware change template. This approach models the change template as Abstract Syntax Tree (AST) edits with data-flow and control-flow dependencies and allows reviewers to iteratively customize the template by parameterizing its content and excluding certain statements. Critics then matches the customized template against the rest of the codebase to summarize similar edits and locate potential inconsistent or missing edits. A reviewer can iteratively refine the template based

[1]http://phabricator.org

[2]http://code.google.com/p/gerrit/

[3]http://visualstudioextensions.vlasovstudio.com/2012/01/06/codeflow-code-review-tool-for-visual-studio/

[4]https://www.atlassian.com/software/crucible

[5]https://www.reviewboard.org/

on previous search results until she is satisfied with the final result. Our hypothesis is:

Hypothesis (H1). By allowing developers to interactively construct and refine an abstract change template, we can accurately summarize similar edits in a diff patch and help developers efficiently identify inconsistent and missing edits during code reviews.

To evaluate this hypothesis, we conduct two user studies. First, six professional developers at Salesforce.com use Critics to investigate the patches authored by their own team. All participants find Critics helpful for inspecting system-wide changes and noticing oversight errors. They would like Critics to be integrated into their current code review environment. This also attests to the fact that Critics is a mature tool that scales to an industry project and can be easily adopted by professional developers. Second, we recruit twelve participants to review diff patches using Critics and Eclipse diff. Participants using Critics answer questions about similar edits 47.3% more correctly with 31.9% less time during code review tasks, in comparison to the baseline use of Eclipse diff. These results show that Critics should improve developer productivity in inspecting similar edits during peer code reviews.

1.2.2 Automated Test Transplantation between Similar Programs

Once Critics identifies program edits that may cause inconsistencies, developers may want to examine how these inconsistencies affect the runtime behavior in similar code locations. For instance, in the user study of Critics, one Salesforce developer expressed the desire to run regression testing to examine the behavioral consistency after applying similar edits to multiple locations, but could not do so due to a lack of test cases. More broadly speaking, developers may want to investigate the semantics change of a reused code fragment with respect to the original code after copying and adapting it to a target program [74].

We develop a test transplantation and differential testing framework called Grafter to help developers examine the behavioral difference between two similar programs (i.e., clones). Given a target code fragment and a reference code fragment, Grafter automatically transplants the target code in place of the reference code so that the target code can be exercised by the test of the reference code. We use def-use analysis to expose the de-facto interfaces of two clones and design code transformation and data propagation rules to reconcile variations in their surrounding contexts. To detect behavioral divergence between two similar programs, Grafter compares both their test results (i.e., *test-level comparison*) and intermediate program states (i.e., *state-level comparison*). Our hypothesis is:

Hypothesis (H2). By exposing de-facto interfaces of two similar programs and properly handling their variations via automated code transformation and data propagation, we can automatically transplant test cases between these programs, boost test coverage, and help developers discern their runtime behavior differences and potential discrepancies.

To verify our hypothesis, we apply Grafter on 52 pairs of code clones in three open-source projects. All of these clones are code fragments inside a method without a well-defined interface and also have variations compared to their counterparts. In 24 clone pairs, only one clone is tested, while its counterpart is not. Grafter successfully reuses tests in 94% of clone pairs without inducing build errors, demonstrating its automated test transplantation capability. The statement and branch coverage of the cloned regions is improved from 50% and 67% to 72% and 83% respectively. To examine the robustness of Grafter, we automatically insert faults using a mutation testing tool [116] and check for behavioral consistency using Grafter. Compared with a static cloning bug finder [111], Grafter detects 31% more mutants using the test-level comparison and 2X more using the state-level comparison. This result indicates that Grafter should effectively complement static cloning bug finders.

1.3 Discovering Common Practices in Open Source Communities

The large and growing body of successful open-source projects on the Internet opens up new opportunities for identifying similar programs in the open source community. The availability of such *Big Code* suggests a new, data-driven approach to develop and maintain software: why not let developers make decisions based on common practices and possible alternatives in the open source community? The scale of available code online is massive (e.g., millions of open-source repositories in GitHub), which poses a significant challenge to efficiently mine, analyze, and visualize such massive data. We build three techniques that efficiently analyze similar programs mined from hundreds of thousands of GitHub projects.

1.3.1 Mining Common API Usage Patterns from Massive Code Corpora

The first technique is ExampleCheck, an API usage mining framework that extracts common API patterns from 380K Java repositories on GitHub. Code-sharing websites such as GitHub implicitly document a variety of programming idioms in open-source communities. We focus on mining API usage patterns, since modern software systems are composed of API calls to external libraries and frameworks. While APIs are designed to encapsulate internal implementation details for ease of code reuse, APIs expose rich semantics and usage constraints, which makes them challenging to use correctly in practice [63, 165, 195].

The intuition of ExampleCheck is that common API usage followed by many other developers in a large number of open-source projects may represent a desirable way of using an API. To capture rich semantics of API usage, we define a new program abstraction that retains not only the temporal ordering of API calls, but also exception handling logic, control structures, and guard conditions of API calls, while abstracting away superficial details such as variable names. To handle the variety of project-specific details in a large corpus, we use program slicing to eliminate extraneous program statements that are unrelated to the API of interest and use an SMT solver to prove the semantic equivalence of various expressions.

Hypothesis (H3). We can learn representative API usage patterns by mining rich API usage semantics from hundreds of thousands of open-source projects and unifying the variety of semantically equivalent expressions using a SMT solver.

We conduct two experiments to verify this hypothesis. First, an evaluation on a state-of-the-art API benchmark [26] shows that ExampleCheck can learn correct API usage patterns with 80% precision and 91% recall in 10 minutes. Compared with mining from a small curated corpus of 7K GitHub projects, mining from 380K GitHub projects improves the precision and recall by 31% and 45% respectively, demonstrating the advantage of mining from massive code corpora. Second, since developers often resort to online code examples for filling their program needs [48, 202, 238], we further demonstrate the usefulness of ExampleCheck by checking for API usage violations in a popular Q&A forum, Stack Overflow. A large-scale analysis of 220K SO posts shows that 31% of these posts contain potential API misuse that could produce unexpected behavior such as program crashes and resource leaks. ExampleCheck can effectively identify valid API misuse with 72% precision and prevent bug propagation when reusing code examples from Stack Overflow.

1.3.2 Visualizing Common and Uncommon API Usage at Scale

Though ExampleCheck considers frequent API usage patterns in a large corpus as correct API usage, some infrequent API usage may still be semantically correct in certain scenarios. Therefore, it is valuable for developers to explore different API usage in a diverse set of usage scenarios. However, since there is often a large number of relevant code examples on the Internet, it is prohibitively time consuming for developers to understand the commonalities and variations among them, while being able to drill down to concrete details.

We design an interactive visualization system called Examplore to help developers explore hundreds of thousands of code examples simultaneously. The key enabler of this visualization is to define an abstract API usage skeleton that summarizes a variety of API usage features, including initializations, enclosing control structures, guard conditions, and other method calls before and after invoking the given API method, etc. Examplore aligns and super-imposes hundreds of code examples into such a skeleton and displays the statistical distribution of different API usage features among these examples. Developers can select desired API usage features in the skeleton and drill down to concrete code examples with selected features.

Hypothesis (H4). By visualizing distinct API usage features along with their statistical distributions in an abstract API usage skeleton, developers can grasp a comprehensive view of various API usage scenarios in a large collection of relevant code examples at scale.

To verify this hypothesis, we recruit sixteen Java developers and ask them to learn new APIs by either searching relevant code examples, tutorials, forum posts online, or exploring a hundred relevant code examples mined from GitHub using Examplore. We designed a set of API usage questions to assess how much API usage knowledge participants acquire in each condition. Participants using Examplore answer questions about API usage more accurately and comprehensively, while participants using the baseline web search often answer questions just based on one example they find or by guessing. Examplore also helps a user explore how other developers have used an unfamiliar API and increases her confidence about correctly using an API in her own program context. These results indicate that Examplore complements existing online resources and tools by providing a bird's eye view of common and uncommon ways of using an API in a developer community.

1.3.3 Adapting Code Examples with Similar GitHub Counterparts

The API misuse study in Stack Overflow implies that online code examples are often incomplete and inadequate for developers' local program contexts. Adaptation of these examples is necessary to integrate them to production code. As a consequence, the process of adapting online code examples is done over and over again, by multiple developers independently.

To draw a deeper understanding about how developers adapt online code examples, we first perform a large-scale empirical study about the nature and extent of adaptations and variations of SO code snippets. We construct a comprehensive dataset linking SO code snippets to their GitHub counterparts based on clone detection, time stamp analysis, and explicit URL references. We then qualitatively inspect 400 SO examples and their GitHub counterparts and develop a taxonomy of 24 adaptation types. Using this taxonomy, we build an automated adaptation analysis technique on top of GumTree to classify the entire dataset into these types. Our quantitative analysis shows that the same type of adaptations and variations appears repetitively among different GitHub clones of the same SO example, and variation patterns resemble adaptation patterns.

Based on insights of the quantitative analysis, we build a Chrome extension called ExampleStack that guides developers to adapt and repair online code examples. Given a SO code example, ExampleStack detects similar code fragments in 50K high-quality GitHub projects and lifts an adaptation-aware template based on the commonalities and variations between the SO example and its GitHub counterparts. The lifted template highlights which parts of the SO example remain constant and illuminates the hot spots where most changes occur.

Hypothesis (H5). Displaying commonalities and variations in similar GitHub counterparts of a Stack Overflow code example can help developers understand how to adapt the example differently and avoid common pitfalls during code reuse.

To verify this hypothesis, we conduct a within-subjects user study with sixteen Java developers and ask each developer to perform two code reuse tasks with and without ExampleStack respectively. The result shows that by seeing commonalities and variations in similar GitHub counterparts of a code example, participants focus more on code safety and logic customization during code reuse, resulting in more complete and robust code. By contrast, participants in the control group make more shallow adaptations such as variable renaming. In a post survey, participants report that ExampleStack increases their confidence about the given SO example, and helps them grasp a more comprehensive view about how to reuse the example differently and avoid common pitfalls.

1.4 Contributions

This dissertation makes the following contributions.

- We proposed the idea of systematic software development and maintenance by discovering and representing commonalities and variations among similar programs. We provided tool support for code reviews, software testing, and opportunistic code reuse, and demonstrated the applicability and usefulness of these tools through quantitative experiments and user studies.

- We pioneered the research direction of applying sophisticated symbolic reasoning and visualization to massive code copora at scale. We built a system that performs program slicing and SMT-based semantic equivalence checking in hundreds of thousands of GitHub projects, and also designed user-centric interfaces for visualizing a large collection of relevant code examples mined from massive code corpora.

- We constructed two comprehensive datasets and conducted two large-scale studies that deepened the understanding about how prevalent API usage violations occur in online code examples and how developers may adapt online code examples in practice. These studies further motivated the design of two Chrome extensions. We released these datasets and tools to support the exploration of alternative tool designs and other related research in the community.

1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 describes related work on code clones, opportunistic code reuse, API usage mining and visualization. Chapter 3 presents the interactive code review technique for similar program edits. Chapters 4 presents the automated test transplantation and differential testing framework for similar programs. Chapter 5 presents the API usage mining and API misuse detection framework and the subsequent API misuse study on Stack Overflow. Chapter 6 presents the interactive visualization system. Chapter 7 presents the empirical study of common adaptation and variation patterns of online code examples and the tool support for guiding the adaptation of a code example by visualizing commonalities and variations in its GitHub counterpart. Chapter 8 concludes this dissertation and outlines avenues of future work.

CHAPTER 2

Related Work

This chapter discusses previous work related to the contributions of this dissertation. Section 2.1 gives an overview about the literature of code cloning, including empirical studies about the presence and evolution of code clones and existing techniques for detecting, updating, and refactoring code clones. Section 2.2 discusses about empirical studies of opportunistic code reuse and tool support for searching relevant code snippets and integrating them to target programs. Section 2.3 discusses about related work in mining software repositories, with a particular focus on API usage mining and visualization. Section 2.4 elaborates state-of-the-art interface design for exploring large collections of concrete examples in both the software engineering (SE) and human-computer interaction (HCI) communities. Sections 2.5, 2.6, and 2.7 briefly discuss related work in modern code reviews, differential testing, and software transplantation respectively.

2.1 Software Repetitiveness and Code Clones

2.1.1 Empirical Studies of the Presence and Evolution of Code Clones

Code clones are common in software systems, which provides a solid foundation for identifying and exploiting similar programs in this dissertation [23, 36, 43, 121, 154, 199]. Kamiya et al. conducted two case studies in JDK and Linux using a token-based clone detector [121]. They found that 29% of lines of code in JDK and 23% in Linux contained code clones. Roy and Cordy analyzed function clones in ten C projects and seven Java projects [199]. They found that on average 15% of the C files and 46% of the Java files contained exact function clones. Java had a higher percentage of clones due to a large number of accessor methods in

Java programs that are not present in C.

In addition to code duplication in individual codebases, cross-project clones are also pervasive in the large body of open-source projects [80, 84, 146, 260]. Lopes et al. analyzed file-level duplication in millions of non-forked GitHub projects [146]. Given such massive code corpora, they found that a staggering number of of source files (70%) in GitHub contained clones of other files somewhere else. Gabel and Su studied the uniqueness of software by analyzing code clones across a large collection of 6,000 SourceForge projects at any granularity rather than just file-level duplication [80]. They found a large portion of small-scale syntactic redundancy at levels of one to seven lines of source code, indicating a general lack of uniqueness in software systems at a fine granularity. These results imply a high probability of identifying similar programs at different granularities from the enormous volume of shared code in the open source community.

The existence of code clones significantly increases software maintenance effort. For instance, when applying bug fixes or performing code refactoring to one location, developers must apply similar edits to other similar locations consistently. Several studies investigate the evolution of code clones by tracing the change histories of code clones [31, 126, 132, 228]. Kim et al. presented an empirical study of code clone genealogies in two Java open-source projects [126]. The authors found that 36% to 38% code clones were changed consistently, thus requiring developers to apply similar edits repetitively to multiple locations. The authors also found that, though many clones were volatile—disappearing after an average of eight commits, those long-lived clones cannot be easily removed using standard refactoring methods. Thummalapenta et al. investigated the change propagation patterns among code clones in four Java and C systems [228]. They found that the majority of clones were either changed consistently or diverged intentionally, while about 16% of clones involved late change propagation—missing edits were applied repetitively in later commits. Tool support for effectively managing and maintaining is much needed to reduce manual effort of applying similar edits and ensure the evolution consistency of code clones. In the next section, we will elaborate existing techniques for detecting, inspecting, editing, and refactoring code clones and discuss how Critics and Grafter contribute to this line of research.

2.1.2 Existing Tool Support for Maintaining Code Clones

2.1.2.1 Clone Detection

As surveyed in [198], a variety of clone detection techniques [2,36,43,70,78,109,113,121,128, 130,131,139,141,144,149,151,154,200] have been proposed and evaluated since the early 90s. Based on the level of analysis applied to the source code, these techniques can be roughly classified to four main categories—*text based*, *token based*, *syntax based*, and *semantics based.*

Text-based approaches treat source code as text and directly compares the text similarity between two programs with little source code normalization [2, 113, 139, 149, 151, 200]. For instance, Johnson proposed n approach that applies a string hashing function to code fragments and then identify those code fragments with the same hash values as clones [113]. Though text-based clone detectors is language-agnostic, they cannot detect programs with similar code structures and functionality while varying a lot in identifiers and code comments.

Token-based approaches utilize a lexical analyzer to tokenize source code and parameterize identifier tokens such as variable names and constant values, in order to detect similar programs with small variations in identifiers [36, 121, 141, 206, 246]. CCFinder is a well-known token-based clone detector [121]. In addition to standard identifier parameterization, CCFinder also leverages a set of transformation rules (e.g., removing initialization lists) as a preprocessing step to eliminate superficial syntactic details in source code, before performing a token-by-token comparison.

Syntax-based approaches parse source code to abstract syntax trees (ASTs) and then identify similar subtrees using tree matching algorithms [43, 70, 130] or using metric-based clustering algorithms [59, 109]. For instance, Deckard encodes syntax trees to numerical vectors and clusters these vectors based on their euclidean distance [109]. The clustering algorithm in Deckard is optimized by locality sensitive hashing (LSH) [58], which generates the same hash value for vectors within a given euclidean distance. In Chapter 4, Grafter leverages Deckard to identify similar programs in software codebases but can be easily extended to support a new clone detector by writing a simple parser to interpret the clone detection result.

Semantics-based approaches use static program analysis to identify more precise semantic information in source code than simply comparing syntactic similarity [78,128,131,144]. For instance, Gabel et al. proposed to represent programs as program dependency graphs (PDGs) to model the control flow and data flow in a program and extended the vector generation phase of Deckard using PDGs rather than ASTs to detect semantic clones.

Recently, several new techniques have been proposed to detect clones in ultra-large code corpora [206] or to detect near-miss clones that are previously difficult to find [205, 246]. SourcererCC (SCC) is a token-based clone detector that scales to hundreds of thousands of open-source projects. SCC achieves this by selecting and indexing a subset of tokens in a code block (i.e., partial indexing) and builds an inverted index mapping between tokens and code blocks. To detect clone candidates, SCC queries the inverted index mapping and exploits the ordering of tokens to filter plausible clones. In Chapter 7, ExampleStack leverages SourcererCC [206] to scale the clone detection between 312K Stack Overflow posts and 51K high-quality GitHub repositories with at least five stars. CCAligner aims to detect cloned code with many inserted or deleted statements in the middle (i.e., large-gaped clones). It applies a sliding window to match inner blocks between two target code fragments and uses asymmetric similarity coefficient to measure the overall similarity of all matched code blocks in these two fragments [246]. Oreo detects near-miss clones with less than 70% token-level similarity by extracting action-related metrics (e.g., array accesses, function calls) and training a deep learning model that predicates clones based on these metrics [205].

2.1.2.2 Clone Tracking and Inconsistency Detection

During the evolution of code clones, inconsistent edits may occur, causing software bugs. For instance, Juegens et al. manually inspected 1427 clone groups in five large systems and found that over half of these clones contained inconsistent edits, among which over a quarter were introduced unintentionally and were confirmed as bugs by developers of these systems [114].

Many techniques have been proposed to track program edits on code clones and automatically detect potential inconsistencies [37, 62, 140, 145, 188]. Duala-Ekoko and Robillard proposed a technique for tracking clones in evolving software, where their technique notifies developers of modifications to clone regions and supports the simultaneous editing of clone regions [62]. Bakota et al. presented an approach that matches clone instances across program versions using supervised machine learning and detects four kinds of clone smells—*vanished clone instances*, *newly introduced clones*, *clone instances that are moved between clone groups*, and *late propagations of clone edits* [37]. CP-Miner detects copy-pasted code via frequent subsequence mining and then matches identifiers (e.g., variable names, function calls) to identify renaming inconsistencies [140]. Jiang et al. presented an approach that detects three types of cloning inconsistencies—*renaming mistakes*, *control-flow construct inconsistencies*, and *conditional predicate inconsistencies* [111]. In an empirical study of software patches in the family of BSD operating systems (e.g., FreeBSD, OpenBSD, NetBSD), Ray et al. found that over 10% of the patches in each system were ported from other systems [187]. The authors further categorized four common types of inconsistencies in these ported patches and proposed an automated approach to detect the control-flow and data-flow inconsistencies in the surrounding context of ported patches [188].

These techniques often report a lot of potential clone inconsistencies that are not real bugs, which is time-consuming for manual examination. To reduce the false positive rate, Lo et al. proposed an interactive approach that only shows a subset of detected clone inconsistencies each time and incorporates user feedback to incrementally refine the inconsistency report [145]. In Chapter 3, we present an interactive approach called Critics that allows a developer to actively express a desired program edit as an abstract change template and continuously refine the template based on identified missing or inconsistent edits in other similar locations. Furthermore, all existing inconsistency detection techniques only leverage static analysis to detect potential inconsistencies between clones. In Chapter 4, we present the first dynamic approach called Grafter that automatically transplants code clones to reuse test and examines the behavior consistency between clones via differential testing.

Origin analysis traces the merging and splitting of code fragments across versions by matching their similarity in terms of variable names, expressions, and function calls [87, 88, 237, 270]. The goal of origin analysis is to reconstruct the evolution history of code fragments and help developers understand structural changes during evolution. It does not monitor similar edits or detect inconsistencies in a group of code clones.

2.1.2.3 Automating Systematic Edits

Due to subtle variations in the program contexts of similar code locations, edits on these locations are often similar but not identical, which is coined as *systematic edits* by Kim and Notkin [125]. Kim and Notkin then proposed an automated approach called LSDiff that automatically infers and summarizes such similar but not identical edits as logic rules. In a case study of a large commercial software, they found that 75% of structural changes in a software revision were systematic edits.

Applying such similar but not identical edits is tedious and error-prone. Several techniques are proposed to automatically apply such similar edits [29, 159, 160, 169, 196]. Andersen et al. presented a generic patch inference algorithm, which takes a set of example program transformations and generates a generic patch to automate similar edits to multiple locations [29]. LibSync automatically updates the usage of an API by learning API usage adaptation patterns based on updates in other callsites of the same API [169]. SYDIT learns an abstract change template by generalizing all identifiers in a given program change and automatically applies similar changes by concretizing the abstract template to other similar locations [159]. LASE extends SYDIT by only generalizing the variations among multiple similar program changes [160]. However, these approaches do not provide users the flexibility to interactively customize change templates. In the comparison between Critics and LASE (Section 3.6.3), we show that the interactive feature of Critics allows users to achieve comparable or higher accuracy within a few iterations.

2.1.2.4 Clone Removal Refactoring

To eliminate the side effects caused by code clones, a number of techniques are proposed to automatically remove code clones [38, 104, 115, 158, 225, 236]. Some of them refactor clones using either the Strategy design pattern or the Form Template Method design pattern [38, 104, 115, 225]. Hence, these techniques are only applicable to clones in a restricted scope. Meng et al. extended prior work by introducing more transformation rules to handle clone variations in types, methods, variables, and expressions [158]. Given a set of systematic edits (i.e., similar edits to multiple locations), their technique extracts the surrounding common parts of these edits and parameterizes their variations. Tsantalis et al. presented another clone removal approach that leverages lambda expressions introduced in Java 8 to factorize code clones [236]. Compared with Meng et al.'s technique, this approach takes as input the results of clone detection tools, rather than relying on systematic edits. Considering the difficulty of clone removal tasks, both techniques achieve promising results in their evaluations, with 58% success rate of clone removal in their benchmarks. However, both techniques do not consider other factors such as readability and maintainability of the refactored code after clone removal.

On the other hand, several empirical studies find that removing clones is not necessary nor beneficial [32,86,122,124]. Based on case studies of Linux kernel and Apache Web server, Kapser and Godfrey presented eight cloning patterns, describing the underlying motivation, benefits, and issues of each cloning pattern [122]. They argued that code duplication seemed to be a reasonable and even beneficial choice in many situations. We share a similar thought that the existence of code clones is not necessarily harmful, as long as there is effective tool support for managing and maintaining code clones. In Chapter 3 and Chapter 4, we demonstrate that designing tools to help developers understand the syntactic and behavioral similarity and differences between code clones can enhance developers' understanding of clones and efficiently identify potential inconsistencies in code reviews and testing.

2.2 Opportunistic Code Reuse

2.2.1 Code Reuse from Local Codebases

One major cause of software repetitiveness and code cloning is copying, pasting, and adapting existing code fragments in developers' own codebases. An in situ observational study showed that 85% of the new classes developed in the one-week observation period were copied and pasted from existing code, followed by heavy modification [136]. Selby analyzed 25 projects at NASA and found that 32% of modules in these projects were reused from prior projects [208], among which 47% required modification from their original form. In a survey with 12 developers in the industry, all developers said that they often copied and pasted code (usually 4 to 50 lines of code) when prototyping new features or incorporating functionality from existing projects [101]. As developers repeatedly explained, the main motivation behind this was to save time—"reusing code is quicker and easier than starting from scratch"—and to increase the reliability of their code, since developers wanted to "leveraging existing testing".

On the other hand, some clones are created unintentionally or independently by different developers. Al-Ekram et al. analyzed code clones across different projects and found that a large number of cross-project clones were resulted from similar usage of APIs such as GUI toolkits and libraries [23]. Though unintentional code cloning does not induce any causal relationship between similar programs, we find it is still valuable to present similar yet independent implementations to developers. For instance, in the user study of ExampleStack (Chapter 7), displaying similar GitHub clones of a Stack Overflow example reminded developers of critical safety checks and logic customization opportunities they may otherwise miss during opportunistic code reuse, even though some of those clones may come from independent but similar implementations.

2.2.2 Code Reuse from the Web

As the Internet accumulates an enormous volume of source code, the code reuse workflow has gradually shifted from local codebases towards the Internet [39, 48, 81, 99, 164, 202, 212, 238, 255]. Developers often search code snippets shared in programming websites such as Stack Overflow to fulfill their own programming needs, e.g., learning new APIs, locating code snippets with desired functionality. Sim et al. conducted a lab study with 36 graduate students to evaluate the effectiveness of different code retrieval techniques [212]. In the demographic survey, 50% of participants reported to search code online frequently and 39%

reported to search occasionally. Sadowski et al. analyzed the search logs generated by 27 Google developers over two weeks [202]. They found that developers issued an average of 12 code search queries to the Web per weekday, while back to the 90s, developers mostly searched within their own codebases using `grep` or other built-in search tools in editors [213].

Brandt et al. conducted a lab study to understand how developers search and leverage online resources during software development [48]. The authors recruited 20 students at Stanford and asked them to prototype a Web chat room application. The authors observed that all participants searched and browsed external resources on the Web extensively. Participants typically started with searching for relevant tutorials and then used the code examples in these tutorials as the scaffold of their own implementation. Baltes et al. surveyed Stack Overflow users to study the usage and attribution of code snippets in Stack Overflow [39]. Among 122 respondents, 79% said they copied code from Stack Overflow no more than a month ago, and 39% not more than a week ago. However, half of them (49%) just copied the code without attributing the original SO post, while the others added a source code comment with a link to the original SO post. Wu et al. analyzed 289 GitHub files that contained a Stack Overflow link in code comments, and found that 44% of GitHub files involved modification varied from simple refactoring to complete reimplementation. They further surveyed Stack Overflow users to investigate the barriers of reusing code from Stack Overflow. Among 453 respondents, 65% explained that the reused code should be adapted accordingly to fit the target context, 44% found it difficult to understand some code snippets in Stack Overflow, and 32% complained about the low code quality in Stack Overflow.

Previous studies have investigated the quality of online code snippets from different perspectives, including compilability [223, 259], comprehensiveness [235], obsolete API usage [268], and security [73, 161]. Subramanian and Holmes analyzed 39K Android code snippets in Stack Overflow and found that 83% of them were incomplete snippets without class or method declarations, which cannot be accepted by standard compilers [223]. Yang et al. performed a more comprehensive analysis of 3M Stack Overflow code snippets in different programming languages [259]. They found that Java and C# had the least complete code snippets in Stack Overflow, with only 4% and 16% parsable and 1% and 0.1% runnable in each language. Treude and Robillard conducted a survey to investigate comprehension difficulty of code examples in Stack Overflow [235]. The responses from 321 GitHub users indicated that less than half of the SO examples were self-explanatory and the main issues included incomplete code, code quality, missing rationale, code organization, clutter, naming issues, and missing domain information. Zhou et al. found that 86 of 200 accepted posts on Stack Overflow used deprecated APIs but only 3 of them were reported by other programmers [268]. Fischer et al. investigated security-related code snippets in Stack Overflow and found that 29% of them were insecure [73]. They further applied clone detection to check for similar code between Stack Overflow and Android applications on Google Play and found that these insecure code snippets may have been copied to over one million Android applications. While our study in Chapter 7 also indicates the limitation of code snippets in Stack Overflow, our study focuses on API usage violations that may lead to unexpected behavior such as program crashes and resource leaks by contrasting SO code examples against frequent API usage mined from massive corpora. Our results strongly motivate the need of systematically augmenting Stack Overflow and helping developers to assess code examples of interest with quantitative evidence about how many GitHub snippets follow (or do not follow) related API usage patterns.

Despite the wide usage of Stack Overflow, most developers are not aware of the SO licensing terms nor attribute to the code reused from SO [28, 39, 255]. An et al. used a clone detector, NiCad [200], to identify duplicated code between 399 Android applications and 2M Android code snippets in Stack Overflow [28]. They found exact copies of SO code snippets in 62 different Android applications, among which 60 applications did not attribute to the original SO posts in Android. According to the survey with 122 SO users in [39], almost one half developers admitted copying code from SO without attribution and two thirds were not aware of the SO licensing implications. Based on these findings, when analyzing adaptation patterns of online snippets in Chapter 7, we carefully construct a comprehensive dataset of reused code, including both explicitly attributed SO examples and potentially reused ones using multiple complementary methods for quality control—clone detection, time stamp analysis, and explicit references.

Prior work has also investigated participation barriers in online programming communities [77, 219, 240, 241]. For instance, in Stack Overflow, negative feedback and hostile criticism can dissuade novice developers from participating in online discussions in the community [219]. Vasilescu et al. find that, compared to traditional mailing lists, Stack Overflow tends to be a relatively "unhealthy" community, in which women disengage sooner although their activity levels are comparable to mens [240]. To improve the engagement of Stack Overflow users, Ford et al. propose to pair novice programmers with experienced mentors [76]. Instead of assigning mentors based on seniority only, this thesis also suggests connecting Stack Overflow users and GitHub developers if they write similar code or use similar APIs, in order to enhance peer learning and assessment in online programming communities.

2.2.3 Tool Support for Searching and Integrating Code Snippets

One important activity in opportunistic code reuse is to locate a desired code fragment to reuse. Many code search techniques have been proposed to expand code search capability beyond keywords only, by utilizing the structural and semantic information in code examples [35,47,66,100,123,137,156,157,191,204,214,215,221,229]. StrathCona searches relevant code examples by matching program contexts [100]. Wang et al. proposed a dependency-based code search technique that represents source code as dependency graphs to capture control-flow and data-flow dependencies in a program, and matches a given search query against program dependence graphs [247]. XSnippet allows a user to search based on object instantiation using type hierarchy information from a given example [204]. S^6 uses a combination of test cases, method signatures, and natural language descriptions to find relevant code examples [191]. Sourcerer provides an SQL database of the source code and metadata in 18K open-source projects, where search queries are formed as standard SQL statements [35]. Boa is a MapReduce-like distributed software mining infrastructure that provides AST traversal primitives over 380K GitHub repositories in Java [66]. In Chapter 5, we leverage Boa to optimize the code search and program slicing phases of the API usage mining framework, since Boa provides an expressive query language and a high-performance infrastructure that scales to hundreds of thousands of projects on GitHub.

Due to the incompleteness of online code snippets, many of them cannot be accepted by compilers or program analysis techniques. Several techniques have been proposed to resolve compilation errors and ambiguous dependencies in incomplete online code snippets [55, 194, 227, 266]. Both ACE [194] and BAKER [224] resolve ambiguous types and method calls in online code snippets. BAKER resolves types and method calls based an *oracle*—a pre-defined set of known Java elements—while ACE analyzes the surrounding texts and code blocks of a given code snippet to find "hints" to resolve types. In Chapter 5, we leverage the oracle-based approach in BAKER to parse incomplete code snippets and resolve types and method calls when analyzing online code snippets. We choose BAKER over ACE because BAKER has a higher accuracy. DEPRECATION WATCHER [268] is a Chrome extension that detects deprecated API usages and gives warnings on Stack Overflow. In Chapter 5, we present a Chrome extension that proactively detects API usage violations in SO posts using common patterns mined from 380K GitHub projects and suggests corresponding fixes to developers when they are browsing Stack Overflow posts.

Previous support for reusing code from Stack Overflow mostly focuses on helping developers locate relevant posts or snippets from the IDE [34, 178, 179, 253]. For example, Prompter retrieves related SO discussions based on the program context in Eclipse [179]. SnipMatch supports light-weight code integration by renaming variables in a SO snippet based on corresponding variables in a target program [253]. Code correspondence techniques [54, 101] match code elements (e.g., variables, methods) to decide which code to copy, rename, or delete during copying and pasting. In Chapter 7, we focus on analyzing the common adaptation and variations patterns of online code examples to draw a deeper understanding about how developers adapt a code example to a target program in practice. Our insight is that, by displaying similar code in real-world projects with their variations, developers can better understand limitations of curated examples and recognize what other developers often change when reusing the same example to real software systems.

On the other hand, many techniques have been proposed to utilize online code snippets for different development activities. SISE automatically augments API documentations by assembling sentences from relevant Stack Overflow discussions [234]. Iyer et al. used texts and code snippets on Stack Overflow to train a neural network for code comment generation [106]. Chen et al. presented an automated bug detection approach by identifying code fragments that are syntactically similar to buggy code snippets posted on Stack Overflow via clone detection [52]. QACrashFix automatically generates patches to fix program exceptions by searching for solutions of similar exception traces on Stack Overflow [83]. All these techniques assume that online code snippets have high quality and can provide insights to guide the automated approaches. However, previous studies and our own work show that online code snippets have many quality issues that may either increase the difficulty of analyzing these snippets due to ambiguous types and incomplete programs, or downgrade the code generated based on these snippets due to API usage violations and security issues.

2.3 Mining and Visualizing API Usage

Mining software repositories is a well-established and active field in Software Engineering. The essential idea is to mine the large amount of source code and metadata such as commit logs in open-source code repositories to gain valuable information and use this information to enhance software tools and processes. Its history dates back to the work by Zimmerman et al. [269], which mines association rules of program changes from software version histories and recommends code elements potentially related to a given change task. In this section, we will focus on prior work about API usage mining, which is closely related to our work in Chapter 5 and Chapter 6, which mines, analyzes, and visualizes API usage patterns in 380K GitHub projects.

There is a large body of literature in mining implicit programming rules, API usage, and temporal properties of API calls [50, 51, 92, 93, 142, 150, 163, 170, 183, 231, 244]. GrouMiner models programs as graphs and performs frequent subgraph mining to find API usage patterns [170]. Buse and Weimer proposed an API usage mining technique that models programs as graphs and clusters these graphs using the k-medoid algorithm [50]. Gruska et al. proposed an API usage mining technique that extracts API call sequences from concrete programs and performs formal concept analysis [82] to identify API methods that are frequently invoked together [92]. Many other specification mining techniques are dedicated to inferring temporal properties of API calls [27, 69, 79, 180, 181, 248]. UP-Miner mines frequent sequence patterns but does not retain control constructs and guard conditions in API usage patterns [244]. Several techniques [142, 163, 231] model programs as item sets and infer pairwise programming rules using frequent itemset mining [90], which does not consider temporal ordering or guard conditions of API calls.

The advent of software forges such as GitHub and BitBucket makes millions of open-source repositories accessible to software developers and researchers. In recent years, there is an increasing interest in mining such "Big Code" to detect software defects and vulnerabilities, enhance program comprehension, and enable automated software construction and repair. The term "Big Code" is first coined by the Darpa MUSE program[1], in order to initiate a new research thrust by treating code as data and applying data analytics to massive code corpora to enhance software quality and robustness. However, such massive code corpora imposes significant challenges to scale previous software mining techniques to millions of open-source repositories. To our best knowledge, the largest code corpus used by previous API usage mining techniques is a collection of 6,000 Linux projects [92].

In Chapter 5, we present a scalable technique called ExampleCheck that mines common API usage patterns from 380K Java projects in GitHub, which is several orders of magnitude larger than prior work. ExampleCheck mines not only API call ordering but also guard conditions using predicate mining. Ramanathan et al. [184] and Nguyen et al. [168] are the only two predicate mining techniques. Ramanathan et al. applied inter-procedure data-flow analysis to collect all predicates related to a call site and then used frequent itemset mining to find common predicates. Ramanathan et al. only mined a single project and did not handle semantically equivalent predicates in different forms. Nguyen et al. improved upon Ramanathan et al. by normalizing predicates using several rewriting heuristics. ExampleCheck differs from these two techniques by formalizing the predicate equivalence problem as a satisfiability problem and leveraging a SMT solver to group logically equivalent

[1]http://science.dodlive.mil/2014/03/21/darpas-muse-mining-big-code/

predicates during predicate mining.

Only a few API usage mining techniques provide support for visualizing the results of mined API usage patterns but they do not focus on how to effectively visualize concrete supporting examples together. For example, Buse and Weimer synthesize human-readable API usage code based on mined graph patterns [50]. GrouMiner applies a similar graph-based mining algorithm and un-parses mined graph patterns to generate corresponding source code [170]. UP-Miner mines frequent method call sequences and visualizes call sequence patterns as probability graphs [244]. All these techniques visualize individual API usage patterns separately without enabling developers to reason about the similarity and differences among different API usage patterns. Furthermore, none of these techniques provides traceability to concrete examples that illustrate these patterns. Instead, Examplore (Chapter 6) visualizes a variety of API usage features along with their statistical distributions in a single synthetic code skeleton, and provides a navigation model to allow users to understand the correspondence between abstract API usage features and concrete examples. ExampleStack (Chapter 7) further extends Examplore by automatically lifting a template from arbitrary similar code fragments, not limited to those with similar API usage nor requiring a pre-defined code skeleton. We achieve this by designing a novel template construction approach that retains the unchanged part among multiple similar programs, clusters the program differences in the same locations, and abstracts away changed locations with *holes*.

2.4 Interfaces for Exploring Collections of Concrete Examples

2.4.1 Exploring and Visualizing Multiple Code Instances

Despite the growing interest of mining a large collection of open-source repositories, there is no easy way for a user to understand the commonalities and variances among a large number of relevant code examples. Most code search and mining techniques present multiple code instances in a flat structure (e.g., a ranked list) without providing an efficient way of exploring these code instances. Program differencing techniques align two programs and highlights their differences in a side-by-side view. However, these techniques can only compare a pair of programs at a time. If there are more than two programs under investigation, developers have to investigate what remains constant and what varies between every two programs, and mentally cluster the differences of every two programs to understand the commonalities and variations among all programs.

Lin et al. proposed a novel program differencing tool called MCIDiff that aligns multiple similar code fragments (i.e., instances of code clones) horizontally in a row and highlights their variances. However, horizontally aligning hundreds of relevant code examples mined from a large code corpus can significantly increase the visual complexity. Developers still have to scroll back and forth between different examples in MCIDiff and mentally interpret what has been changed and what remains constant among these examples.

OverCode represents how hundreds or thousands of students independently implement the same function in massive programming classes [85]. Given a large number of raw functions written by students, OverCode canonicalizes variable names and reformats the raw implementation to a standardized format. OverCode then clusters similar function implementations using both static analysis and dynamic analysis and displays the variations among different clusters along with the number of corresponding raw functions in the corpus. OverCode does not directly carry over to code examples collected outside a massive classroom, where developers are not all implementing the exact same function.

There are two major motivations for visualizing a large collection of code examples for API usage (Chapter 6). First, developers often express the desire for exploring multiple code examples for API learning. Buse and Weimer conducted a survey with 150 industrial developers and found that "the best documentation must show all different ways to use something, so it's helpful in all cases" [252]. The respondents in another survey also expressed a desire to examine multiple examples to investigate alternative uses [195]. This is also confirmed by several quantitative analysis of search logs. Hoffmann et al. analyzed the search logs on MSN and found that 34% of the search queries had a goal of finding and learning APIs [99]. Of those, 18% include the word "sample" or "example" in the query. Montandon et al. instrumented the Android API documentation platform and found that Android developers often searched for concrete code examples in Android documentations [164].

Second, as discussed in Section 2.2.2, individual code examples may suffer from insecure coding practices [73,161], unchecked obsolete usage [268], and comprehension difficulty [235]. Therefore, it is beneficial to get a comprehensive view of API usage and understand the pros and cons of different code examples. One might expect developers to perform thorough investigation on multiple code examples to learn an unfamiliar API. In practice, however, developers often examine only a few search results due to limited time and attention. In [48], Brandt et al. observed that participants typically clicked several search results and then judged their quality by rapidly skimming. Duala-Ekoko and Robillard observed that participants often backtracked when browsing search results, due to irrelevant or uninteresting information in search results [63]. More specifically, Starke et al. showed that developers rarely looked beyond five examples when searching for code examples to complete a programming task [218]. These results indicate that the code exploration process is often limited to a few search results, leaving a large portion of foraged information unexplored. To guide users to explore a large number of code examples simultaneously, EXAMPLORE constructs a code skeleton with statistical distributions of individual API usage features as a navigation model.

2.4.2 Exploring and Visualizing Other Complex Objects

The HCI community has a broad interest of visualizing large collections of objects such as Photoshop tutorials and text documents, not just limited to source code.

Both Sifter [174] and Delta [129] operate on sequences of image manipulation commands in Photoshop. Pavel et al. created an interface for browsing the variation and consistency across large collections of Photoshop tutorials, focusing in particular on sequences of invoked Photoshop commands [174]. Kong et al. addressed a similar problem by presenting different linked views, including lists, side-by-side comparisons between a few sequences, and clusters [129]. Unlike code or text, images can be easily consumed at a glance, so these systems use thumbnails to make a long reading comparison task easy.

Both WordTree [249] and WordSeer [209] operate on *text*, while EXAMPLORE attempts to translate similar insights to *code*. WordTree uses alignment, counts, deduplication, and dependence to visualize the *N+1*-word sequences containing the *N*-word long sequences simultaneously for a user-chosen root word. Similar to WordTree, Examplore captures dependencies across the multi-dimensional space of code examples. For example, when a user selects a particular feature option in the code skeleton, Examplore dynamically updates the counts of remaining feature options so that they are conditioned on the selected option, revealing the statistical distribution of co-occurring feature options. WordSeer infers the grammatical structure of natural language documents in a given corpus. It leverages the inferred grammatical structure to power grammatical search, where users can query for, e.g., *who (or what) is described as "cruel" in North American Antebellum slave narratives?* The result is a ranked list, with counts, of the different extracted entities described in one or more narratives as "cruel"; we adapt this display in Examplore to show the distribution of options for an API usage feature such as the guard condition of an API call.

2.5 Modern Code Reviews and Program Change Comprehension

Rigby et al. investigated code review practices in open source development and found that developers could understand small, logical, coherent units of code changes better rather than large, unrelated changes [193]. In another study, Rigby et al. found general principles of code review practices and elaborated the benefits of code reviews [192]. Bacchelli and Bird studied modern code review practices and found that a key challenge of modern code review is a lacking tool support for program change comprehension [33]. Tao et al. studied the challenges of comprehending program changes during code reviews and concluded that code review tools must support the capability to divide a large chunk of code changes into sub logical groupings and to filter non-essential changes [226]. These findings motivate the design of Critics. Barnett et al. developed a static analysis technique to help developers to understand program changes during code reviews [40]. Their technique decomposes a large diff patch to small program changes and cluster relevant changes using dependency analysis. While our work in Chapter 3 shares the same goal of assisting program change comprehension in code reviews, we focus on summarizing similar changes via interactive change template construction and identifying unintentional edit inconsistencies in a diff patch.

RefFinder summarizes refactoring edits in a diff patch as logic facts using using pre-defined refactoring rules [182]. LSdiff infers the latent patterns of similar edits and summarizes them as logic rules [125]. However, LSdiff supports only coarse-grained analysis at the level of method calls and field accesses. Furthermore, LSDiff does not leverage any human input to guide the inference process and thus often discovers a large amount of rules in an inefficient top-down manner, while discarding most of them in a post-processing step. In contrast, Critics allows a user to interactively refine an abstract diff template to be used.

2.6 Differential Testing

The cost of manually constructing test cases is high, which makes automated test generation very appealing. However, it is difficult to define the oracle of automatically generated tests without prior knowledge of expected outputs, which is known as the test oracle problem [42]. Differential testing addresses this oracle problem by examining test outputs of comparable systems [57, 91, 135, 155]. For example, Groce et al. randomly simulated different kinds of system failures and compared the fault tolerance capability of a Flash file system at JPL with heavily tested and widely available file systems, including Solaris file system, Cygwin, and EXT3 and tmpfs on Linux, as reference implementations [91]. Daniel et al. developed a differential testing technique that automatically generates Java programs to test code refactoring engines and identifies inconsistencies among the output programs generated by different refactoring engines [57]. All these techniques require similar programs under test to have the same interface for ease of comparison, whereas the real-world code clones often are sub-method code fragments without a clear interface. In Chapter 4, we explore the feasibility of automatically transplanting test cases between syntactically similar code fragments by using def-use analysis to expose their de-facto interfaces clones.

Several differential testing techniques use a *record-and-replay* approach to execute different programs under the same test environment. For example, Saff et al. presented an automated approach that generates unit tests from system tests by recording interactions such as method calls and referenced objects and by loading serialized interactions to replay the recorded execution [203]. Instead of serializing actual objects, Orso and Kennedy extended Saff et al. by only recording unique object ids in a reference map and inserting stub code to look up the reference map [171]. Elbaum et al. presented a unit test generation technique that detects heap objects reachable from a given method using k-bound analysis [185] and serializes reachable objects before and after the execution as the pre-state and post-state [68]. Diffut allows simultaneous execution and comparison of corresponding methods between different program revisions [256]. These techniques target regression testing scenarios and assume that identifier names stay unchanged between revisions. Unlike these techniques, Grafter handles name and type variations between code clones to enable differential testing.

2.7 Software Transplantation

In Chapter 4, the way Grafter grafts clones for test reuse resembles software transplantation techniques. Petke et al. used code grafting and genetic programming to specialize miniSAT for high performance combinatorial interaction testing [176]. Harman et al. introduced a *grow-and-graft* approach that transplants new functionality into an existing system [96]. This approach requires developers to provide hints regarding where to capture functionality from and how to constrain search space. μSCALPEL transplants arbitrary functionality from a donor system to a target system [41]. It requires organ entry and implantation points, similar to how Grafter requires donor and recipient clones. μSCALPEL first over-approximates graftable code through slicing and reduces and adapts it through genetic programming. To guide a generic search algorithm, μSCALPEL requires the existence of test suites at both the origin program and the target program. In contrast, Grafter does not require both clones to be already tested by existing tests. In Grafter, organ extraction and adaptation is not a search problem, rather a deterministic process guided by syntactic resemblance and its goal is to reveal behavior differences between clones at runtime. The Java type-safe grafting technology presented in our paper may have the potential to be used for automated code reuse.

Genprog transplants code from one location to another for automated repair [251]. Genprog relies on existing tests as oracles to validate the repair candidates. Thus, it may not be able to find repair solutions effectively, when a test suite is inadequate. A recent study shows that using human-written tests to guide automated repair leads to higher quality patches than using automatically generated tests [216]. By reusing human-written tests and boosting test coverage for similar code, GRAFTER may help improve test oracle guided repairs. Sidiroglou-Douskos et al. presented Code Phage, a system for automatically transferring input validation checks [211]. Code Phage requires an error generating input and a normal input. Through instrumented execution, they obtain a symbolic expression tree encoding a portable check. GRAFTER can be used to transplant code fragments with arbitrary lines of code, not limited to input validation checks.

Skipper copies related portions of a test suite when developers reuse source code [148]. It determines relevant test cases and transforms them to fit the target system. Skipper is built on Gilligan [101, 102] and requires users to provide a *pragmatic reuse plan* to establish the mapping of reused entities (e.g., classes, methods) from the original system to the target system and guide the transformation process. Skipper also assumes that the reused code is full-feature clones at the level of methods and classes. GRAFTER differs from Skipper in three perspectives. First, GRAFTER supports sub-method level clones without explicit interfaces. Second, GRAFTER does not require a reuse plan but rather leverages the syntactic resemblance between clones to guide the grafting process. Third, Skipper may require manual adjustments to fix compilation errors when the reuse plan is incomplete. In contrast, GRAFTER is fully automated using transplantation rules (Section 4.3.2) and data propagation heuristics (Section 4.3.3).

CHAPTER 3

Interactive Code Review for Similar Program Edits

This chapter presents an interactive approach that searches for similar code locations and identify potential edit mistakes. Though code duplications and redundancies in local codebases have long been believed to be harmful, we demonstrate that allowing developers to interactively express a desired change template and reason about commonalities and variations among similar edits can significantly improve maintenance efficiency and avoid unintentional edit inconsistencies during peer code reviews.

3.1 Introduction

Code reviews are one of the most important quality assurance activities in software development [22, 65, 71, 250]. According to a recent study, developers spend a significant amount of time and effort to comprehend code changes during peer code reviews [226]. When the information required to inspect code changes is distributed across multiple files, developers find it difficult to inspect a diff patch [64].

Popular code review tools only compute differences per file, which enforces developers to read changed lines file by file and focus on local program contexts only. Clone detection, code search, and matching approaches [121, 140, 152, 247] can locate similar code fragments, but they do not summarize *similar edits* nor report *change anomalies* in a diff patch. These approaches also do not empower users to interactively investigate systematic changes, as they do not give users the control to iteratively generalize the search template. LSdiff [125] automatically summarizes coarse-grained structural differences, but naively enumerates all possible systematic change patterns as rules. This leads to the issue of poor scalability and

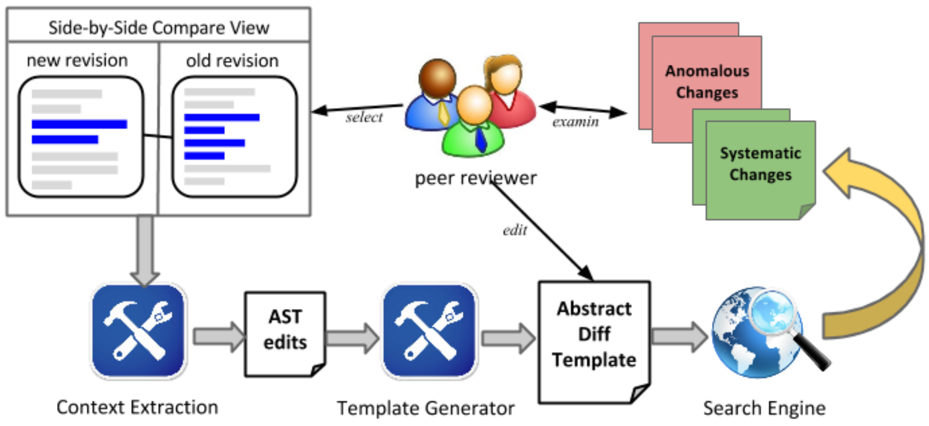


Figure 3.1: An overview of Critics workflow

a high rate of false positives.

We design Critics, a new approach for interactively searching and inspecting similar edits during peer code reviews. Figure 3.1 gives an overview of Critics. By selecting a certain region of a diff patch, a reviewer can specify the program change she wants to investigate. Critics then extracts a change template by including the surrounding unchanged code parts that the selected change depends on. A reviewer can further customize the template to enable flexible code matching. For example, a reviewer can parameterize variable names, types, and method calls so that the template can be matched with a location with different identifier names. Using the customized template, Critics searches the entire codebase to identify similar edits and report potential edit inconsistencies that violate the template. A reviewer can incrementally refine the template based on previous search results, until she is confident that no more similar edits or edit mistakes can be found.

We evaluated Critics through two user studies and a comparison with an automated template construction approach [160]. In the first study, we interviewed six professional developers at Salesforce.com to solicit feedback about the usability of Critics in an industrial setting. Each participant first used Critics to investigate diff patches mined from the version history of their own codebase, and then shared insights about code review challenges at Salesforce.com and whether and how Critics could help. Salesforce developers confirmed that they indeed faced the difficulty of reviewing similar edits due to code redundancy in their codebase. All of them found CRITICS helpful by allowing them to search for similar edits and reporting edit mistakes. They also mentioned that the interactive code search feature in CRITICS was beneficial for developer onboarding in their team, where novices could efficiently explore the codebase using desired code patterns. In the second study, we quantified the effectiveness of CRITICS by conducting a within-subjects user study with twelve students. Each student performed two code review tasks, one using CRITICS and the other using the default code review features in Eclipse. The evaluation result showed that participants answered code review questions 47% more correctly and 32% faster on average with the assistance of CRITICS, in comparison to the baseline. Finally, we compared the code search accuracy of CRITICS with a state-of-the-art approach called LASE [160]. We found that in five out of six cases, interactively customizing a change template using CRITICS could achieve the same or higher accuracy than LASE within an average of four iterations, demonstrating the advantage of interactive template construction over automated template construction.

The rest of this chapter is organized as follows. Section 3.2 illustrates the code review process in CRITICS using a real-world diff patch from Eclipse. Section 3.3 describes template construction and refinement in CRITICS. Section 3.4 describes tree-based code matching and inconsistency detection. Section 3.5 demonstrates tool features in the Eclipse plug-in of CRITICS. Section 3.6 describes the evaluation. Section 3.7 discusses the threats to validity.

3.2 Motivating Example

This section overviews CRITICS with an example drawn from the Eclipse Standard Widget Toolkit (SWT) project. SWT is an open source widget toolkit with 400K lines of source code over 1000 files. This example is based on a diff patch at revision 13516, as shown in Figure 3.2. The patch is adapted and simplified for presentation purposes.

Suppose Alice updates the program to use the new `sendEvent` API. Barry needs to review Alice's changes to ensure all locations using `sendEvent` are updated correctly and to check if there is any location that Alice forgot to change. The diff patch authored by Alice is over 450 lines of changes distributed across 42 different locations.

```
int keyDownEvent (int wParam, int lParam) {
- ExpandItem item = items [focusIndex];
  switch (wParam) {
  case OS.VK_SPACE:
  case OS.VK_RETURN:
    Event event = new Event ();
-   event.item = item;
-   sendEvent(true, event);
+   event.item = focusItem;
+   sendEvent(focusItem.expanded ? COLLAPSE:EXPAND,
     event);
+   refreshItem(focusItem);
    ...
}
```

(a) A changed region selected by Barry

```
int keyPressedEvent (int wParam, int lParam) {
  ExpandItem item = items [focusIndex];
  switch (wParam) {
  case OS.VK_SPACE:
  case OS.VK_RETURN:
    Event event = new Event ();
    event.item = item;
    sendEvent(true, event);
   ...
}
```

(b) Code location with exactly the same context but missing the update

```
int keyReleaseEvent (int wParam, int lParam) {
- ExpandItem item = items [focusIndex];
  switch (wParam) {
  case OS.GDK_RETURN:
  case OS.GDK_SPACE:
    Event ev = new Event ();
-   ev.item = item;
-   sendEvent(true, ev);
+   ev.item = focusItem;
+   sendEvent(focusItem.expanded ? COLLAPSE:EXPAND,
     ev);
+   refreshItem(focusItem);
   ...
}
```

(c) A similar but not identical change using a different variable name, `ev`, instead of `event`

```
int buttonUpEvent (int wParam, int lParam) {
- ExpandItem item = items [focusIndex];
  if (lParam == HOVER) {
    Event bEvent = new Event ();
-   bEvent.item = item;
-   sendEvent(true, bEvent);
+   bEvent.item = focusItem;
+   sendEvent(focusItem.expanded ? EXPAND:COLLAPSE,
     bEvent);
+   refreshItem(focusItem);
    ...
}
```

(d) Inconsistent change by mistakenly swapping two expressions, `EXPAND` and `COLLAPSE`

Figure 3.2: Simplified examples of similar and consistent changes, inconsistent changes, and missing updates. Code deletions are marked with '-' and additions are marked with '+'.

In order to find incorrect edits, Barry needs to inspect line level differences file by file. In particular, to identify missing updates, he must also inspect unchanged code as well, since the original diff patch does not show what did *not* change. The following shows how Barry may iteratively use CRITICS to inspect similar changes and to detect potential missing or inconsistent updates.

Iteration 1. Suppose Barry first inspects changes in the `keyDownEvent` method in Figure 3.2(a). He wonders whether there are other methods that are changed similarly to `keyDownEvent`. So he selects the changed code in the diff patch. Given the selected change, CRITICS identifies the *change context*—unchanged, surrounding code relevant to these edits in terms of control and data dependences, which further serves as an anchor to locate missing updates during the searching process. So the default template generated by CRITICS consists of both the initial edit selection and the change context. Using the template, CRITICS locates code that matches the change context but is missing the update, shown in Figure 3.2(b).

Iteration 2. After examining the search result in the first iteration, Barry wants to explore further since he suspects other locations may use different identifier names. To match similar but not identical changes, CRITICS allows Barry to generalize the change template by parameterizing type, variable, and method names. So Barry generalizes the variable name `event` and searches again. This time, the location in Figure 3.2(c) is summarized although it uses a different variable name, `ev`.

Iteration 3. CRITICS includes the change context such as the `switch` and `case` statements from lines 3 to 5 in Figure 3.2(a) in the current template. Barry wonders if there are similar changes in different control-flow contexts such as a `for` loop or an `if-else` branch. He excludes the switch statement. Using the new refined template, CRITICS locates `buttonUpEvent` in Figure 3.2(d). This location uses an `if` statement instead of a `switch` statement. However, CRITICS flags this location as a potential inconsistent change, since Alice mistakenly swapped the two expressions, `EXPAND` and `COLLAPSE`. Such mistake is usually hard for the reviewer to detect during code inspection.

3.3 Change Template Construction and Refinement

CRITICS provides a novel integration of program differencing and pattern-based interactive code search to help developers note inconsistent or missing changes during peer code reviews. It consists of the following three phases. Phase I takes a user specified change region and extracts the relevant context. Phase II allows developers to customize the change template by interactively generalizing its content. Phase III matches a template against the codebase to summarize similar changes and to detect potential anomalies. The reviewer can investigate the diff patch and achieve the desired result by iteratively refining the change template (Phase II) and searching change locations (Phase III).

3.3.1 Program Context Extraction

Critics parses the selected changed fragments into Abstract Syntax Tree (AST) edits and extracts the change context—surrounding unchanged code on which the selected edits are control and data dependent by performing static intra procedural slicing [103]. It selects all upstream dependent AST nodes based on a transitive relation within a method. The context could indicate where edits should be applied and serve as an anchor to locate similar edits and to identify potential mistakes.

- Data dependence: AST node n_j is data dependent on node n_i, if node n_j uses a variable whose value is defined in node n_i. For example, by analyzing data dependencies between edits and surrounding unchanged code, Critics includes a variable declaration at line 6, whose variable `event` is referenced from the deleted line 7 in Figure 3.2(a).
- Control dependence: AST node n_j is control dependent on node n_i, if node n_j may or may not execute depending on a decision made by node n_i. For example, the `switch` and `case` statements from lines 3 to 5 in Figure 3.2(a) are included since the execution of the changed code depends on these control predicates.

3.3.2 Change Template Customization

Critics creates a default change template, including the initial selected fragments and the change context. A reviewer can customize the template by generalizing its content and to iteratively refine the template.

Parameterizing Identifiers. Critics allows a developer to parameterize type, variable, and method names, so that they can be regarded as equivalent to different identifiers during the matching process. Suppose that there is a statement `char[] data = foo()` in the change template. By parameterizing the variable name, `data`, Critics automatically propagates the parameterization to all statements referencing `data` and this statement can be matched to any other statements in the form of `char[] $V1 = foo()` where `$V1` represents any variable name.

Excluding Statements. CRITICS allows a user to exclude certain statements in the change template. Specially, by excluding contextual statements, CRITICS is able to find similar changes in multiple contexts. An excluded statement is mapped to a parameter `$EXCLUDED` in a generalized change template. For example, by converting `switch(x)` to `$EXCLUDED`, it can match `if(x == 1114)` in line 3 in Figure 3.2(d).

3.4 Interactive Code Search and Anomaly Detection

Given a customized change template, CRITICS matches the template against the entire codebase and detects inconsistent or missing edits in other similar code locations.

3.4.1 Tree-based Program Matching

Tree Matching. CRITICS parses all Java methods in a codebase to abstract syntax trees and searches for similar subtrees by matching the change template against other methods in the codebase. CRITICS applies an efficient and worst-case optimal tree matching algorithm, Robust Tree Edit Distance (RTED) [175], which combines the strengths of Zhang's algorithm [261] and Demaine's algorithm [61]. Zhang's algorithm is efficient for trees with $O(log\ n)$ depth but has the worst-case time complexity $O(n^4)$. Demaine's algorithm has a better worst-case time complexity $O(n^3)$ but runs into the worst case frequently. RTED recursively decomposes the input trees into sub-forests, either removing the leftmost or the rightmost root node. It then computes the tree edit distance recursively by finding structural alignment. RTED then provides a list of matching node pairs with node edit operations that transform one tree into another.

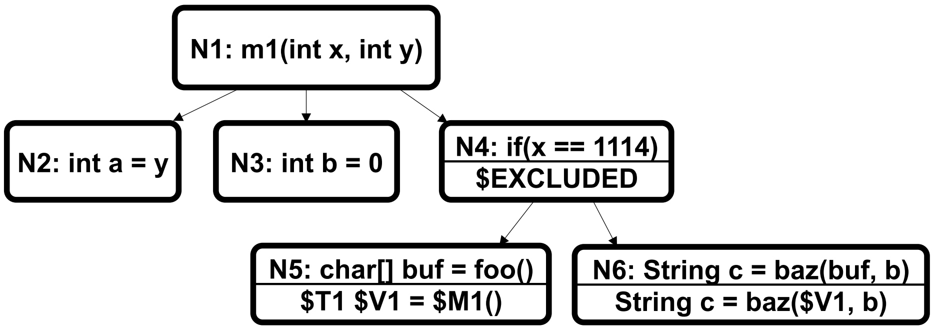
The original RTED algorithm finds node-level alignment in a flexible manner, producing many false positives. Therefore, CRITICS further computes token-level alignment between two matching nodes. If the token labels are exactly the same, CRITICS considers them to be equivalent. While matching labels, we match the parameterized names such as `$V1` in the query tree with any concrete name in the target tree to support flexible matching. For example, CRITICS inspects a node pair that RTED aligns by calculating the minimum edit distance, as described in line 7 in Algorithm 1. The `TokenMatch` procedure checks equivalence between two AST nodes in terms of token-level string values (line 14). Given a list of token level matches, CRITICS checks whether a parameterized name is mapped to a concrete name. Suppose that RTED aligns two nodes: "`$T $V = $M(y)`" and "`int x = foo(y)`." CRITICS produces token level alignment: {("`$T`", '`int`"), ("`$V`", " `x`"), ("`$M(y)`", "`foo(y)`")}. Because these token level mappings are allowed via explicit parameterization in the previous step, CRITICS considers the aligned two nodes as identical and continues to check the next aligned pair.

As another adaptation to RTED, CRITICS checks whether there is an excluded node in the list of the aligned nodes computed by RTED. If RTED aligns an `$EXCLUDED` node with another node, CRITICS allows such matching, as described in line 11 in Algorithm 1. Suppose that CRITICS takes a node pair `switch(x)` in a query tree and `if(x == y)` in a target tree. If the node `switch(x)` is excluded by a user, CRITICS matches the two nodes. Figure 3.3 shows an example of node level and token level alignment.

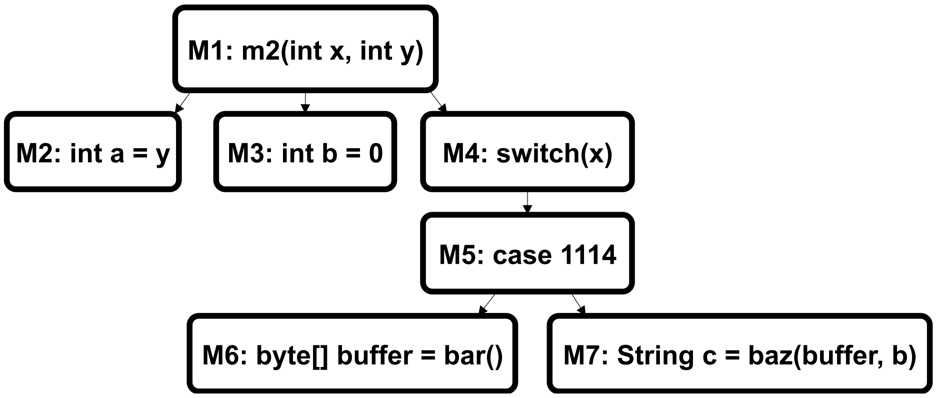
CRITICS improves the performance of search by caching relevant data to reduce search load. CRITICS maps an identifier name to a set of source files using the identifier name and stores the mappings in a hash table. Before running RTED, CRITICS inspects each identifier name in a query tree and identifies a set of files using the same set of identifier names by looking up the hash table. Then, it only scans through the searched source files to avoid unnecessary matching.

3.4.2 Change Summarization and Anomaly Detection

Each template consists of a before state and an after state. The *before state* refers to code before edits. Conversely, the *after state* refers to the code after edits. Using the tree matching algorithm, CRITICS finds two sets of similar subtrees, matching the old and the new version respectively. If a method matches the before state, but not the after state, it implies that the programmer either made an incorrect edit or forgot to update the code. Similarly, if a method matches the after state but not the before state, CRITICS reports it as an anomaly



(a) A query tree.



(b) A target tree matched with the above query tree.

Figure 3.3: RTED matches nodes, such as ($\texttt{N}_1$, $\texttt{M}_1$), ($\texttt{N}_2$, $\texttt{M}_2$), ($\texttt{N}_3$, $\texttt{M}_3$), ($\texttt{N}_4$, $\texttt{M}_4$), ($\texttt{N}_5$, $\texttt{M}_6$), and ($\texttt{N}_6$, $\texttt{M}_7$), and CRITICS matches tokens in the labels of two matched nodes $\texttt{N}_5$ and $\texttt{M}_6$, such as (“`$T1`”, “`byte[]`”), (“`$V1`”, “`buffer`”), and (“`$M2`”, “`bar`”).

as well, because similar edits are made to different contexts. We report two types of change anomalies: (1) *inconsistent changes*, where edits are applied but partially incorrect and (2) *missing updates*, where the required edits are completely missing. This feature of detecting change anomalies distinguishes CRITICS from other pattern mining and anomaly detection approaches that work with a single program version as opposed to a diff patch.

Algorithm 1: Searching similar subtrees.

```
Input  : Let AST be an Abstract Syntax Tree for a program.
Input  : Let QT be a query tree from a customized change template.
Output: Let MTs be a collection of the matched subtrees.

Algorithm searchSimilarSubtrees(QT)
1    MTs := $\emptyset$;
2    foreach $node_i$ from AST do
3        $t$ := getSubtree($node_i$);
         /* $t$ is a target tree */
4        if RTED.match(QT, $t$) $\equiv$ TRUE then
5            $nodePairs := \emptyset$;
6            $nodePairs := nodePairs \cup \{(n_i, m_i)$ — $n_i \in$ QT,
               $m_i \in t$, where $(n_i, m_i)$ is a pair of nodes that
               RTED matches and aligns.$\}$;
7            if tokenMatch($nodePairs$) $\equiv$ TRUE then
8                MTs := MTs $\cup \{t\}$;
9    return MTs;

Procedure tokenMatch($nodePairs$)
10   foreach $pair_i \in nodePairs$ do
11       if $pair_i.n$ is excluded then
12           continue;
13       if match($pair_i.n.label$, $pair_i.m.label$) $\equiv$ FALSE then
             /* $pair_i.n.label$ is different from $pair_i.m.label$. */
14           $tokenPairs := \emptyset$;
15           $tokenPairs := tokenPairs \cup \{(t_j, u_j)$ —
               $t_j \in pair_i.n.label$, $u_j \in pair_i.m.label$, where
               $(t_j, u_j)$ is a pair of a deleted token and an ins-
               erted token that CRITICS matches and aligns.$\}$;
16           if $\forall t_j \in tokenPairs$, $t_j$ is parameterized then
17               continue;
             else
18               return FALSE;
19   return TRUE;
```

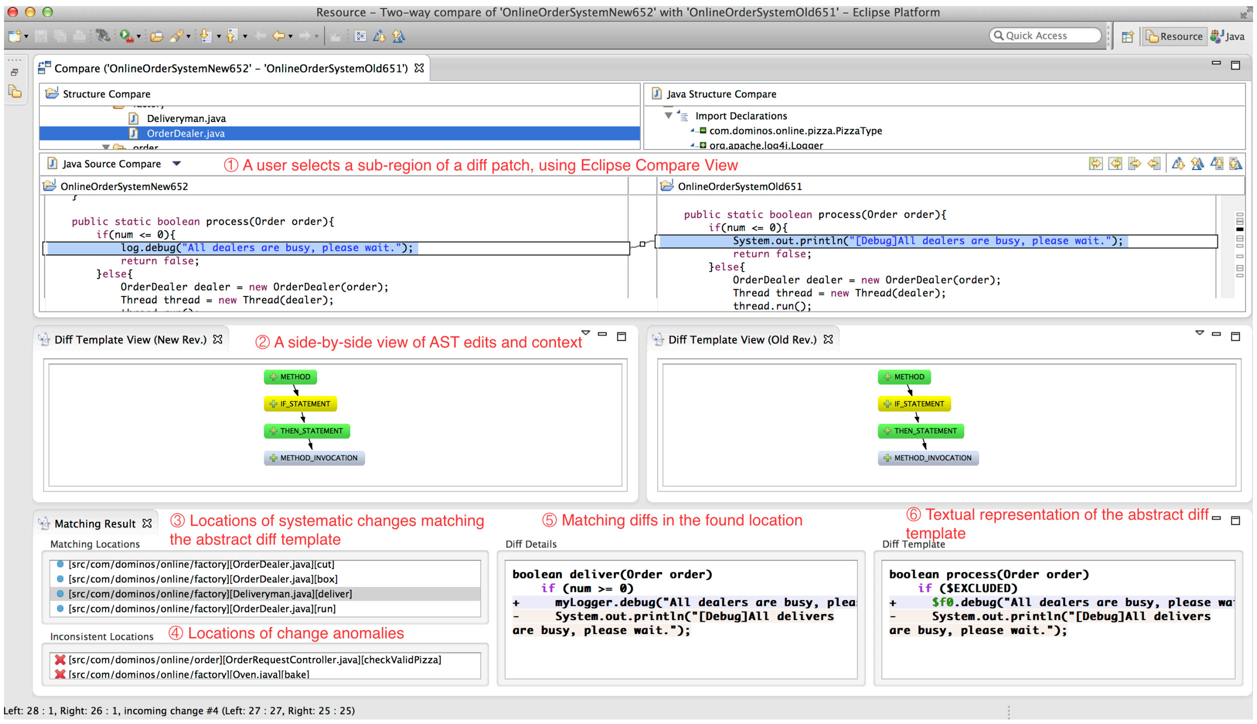
3.5 Implementation and Tool Features

Critics is instantiated as an Eclipse plug-in. Both the tool and its source code are available online.[1] Our implementation leverages ChangeDistiller [75] to compute AST edits, Crystal[2]

[1]https://github.com/tianyi-zhang/Critics

[2]https://code.google.com/p/crystalsaf/

for data and control flow program analysis, and RTED [175] for computing tree edit distance. Now we illustrate the tool features in Eclipse plug-in of CRITICS.


Resource – Two-way compare of 'OnlineOrderSystemNew652' with 'OnlineOrderSystemOld651' – Eclipse Platform
Quick Access
Resource
Java
Compare ('OnlineOrderSystemNew652' – 'OnlineOrderSystemOld651')
Structure Compare
Deliveryman.java
OrderDealer.java
order
Java Structure Compare
Import Declarations
com.dominos.online.pizza.PizzaType
org.apache.log4j.Logger
Java Source Compare
① A user selects a sub-region of a diff patch, using Eclipse Compare View
OnlineOrderSystemNew652
OnlineOrderSystemOld651
Diff Template View (New Rev.)
② A side-by-side view of AST edits and context
Diff Template View (Old Rev.)
METHOD
IF_STATEMENT
THEN_STATEMENT
METHOD_INVOCATION
METHOD
IF_STATEMENT
THEN_STATEMENT
METHOD_INVOCATION
Matching Result
③ Locations of systematic changes matching the abstract diff template
⑤ Matching diffs in the found location
⑥ Textual representation of the abstract diff template
Matching Locations
Diff Details
Diff Template
Inconsistent Locations
④ Locations of change anomalies

```
public static boolean process(Order order){
    if(num <= 0){
        log.debug("All dealers are busy, please wait.");
        return false;
    }else{
        OrderDealer dealer = new OrderDealer(order);
        Thread thread = new Thread(dealer);
```

```
public static boolean process(Order order){
    if(num <= 0){
        System.out.println("[Debug]All dealers are busy, please wait.");
        return false;
    }else{
        OrderDealer dealer = new OrderDealer(order);
        Thread thread = new Thread(dealer);
        thread.run();
```

Matching Locations:

- [src/com/dominos/online/factory][OrderDealer.java][cut]
- [src/com/dominos/online/factory][OrderDealer.java][box]
- [src/com/dominos/online/factory][Deliveryman.java][deliver]
- [src/com/dominos/online/factory][OrderDealer.java][run]

Inconsistent Locations:

- [src/com/dominos/online/order][OrderRequestController.java][checkValidPizza]
- [src/com/dominos/online/factory][Oven.java][bake]

Diff Details:

```
boolean deliver(Order order)
    if (num >= 0)
+       myLogger.debug("All dealers are busy, plea
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

Diff Template:

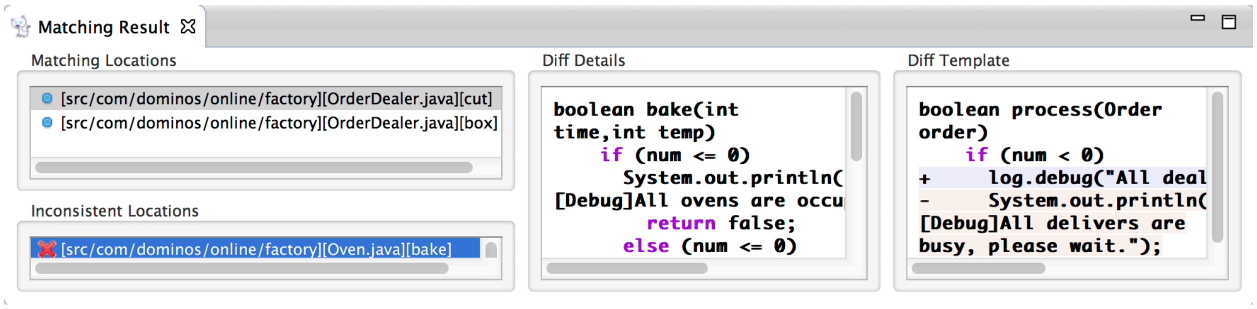
```
boolean process(Order order)
    if ($EXCLUDED)
+       $f0.debug("All dealers are busy, please wa
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

Left: 28 : 1, Right: 26 : 1, incoming change #4 (Left: 27 : 27, Right: 25 : 25)

Figure 3.4: A screen snapshot of CRITICS Eclipse Plug-in and its features.

Suppose Barry and Alice are developing an online sales system for a pizza store. Suppose Barry is conducting a code review of a diff patch authored by Alice. The check-in message says, “update to use log4j for log management.” Barry wants to check that Alice refactored all locations that print log messages to the console, so that these locations can use Apache log4j APIs instead. Without CRITICS, Barry must inspect each changed location one after another because existing *diff* displays only line-level differences per file. Furthermore, he has to search the entire codebase to ensure that Alice did not miss anything, because missing updates do not appear in the diff patch. This manual reviewing process not only requires the deep knowledge of the codebase but also is tedious and error prone.

Eclipse Compare View. Barry first inspects a region of changed code in method `OrderDealer` using the **Eclipse Compare View** (see ① in Figure 3.4). In this method, Alice updated the original `System.out.println` statement with log4j API `debug`. Barry checked this change and now he wonders if Alice updated all other similar locations correctly. To



```
[src/com/dominos/online/factory][OrderDealer.java][cut]
[src/com/dominos/online/factory][OrderDealer.java][box]
```

```
[src/com/dominos/online/factory][Oven.java][bake]
```

```
boolean bake(int
time,int temp)
    if (num <= 0)
      System.out.println(
[Debug]All ovens are occu
        return false;
      else (num <= 0)
```

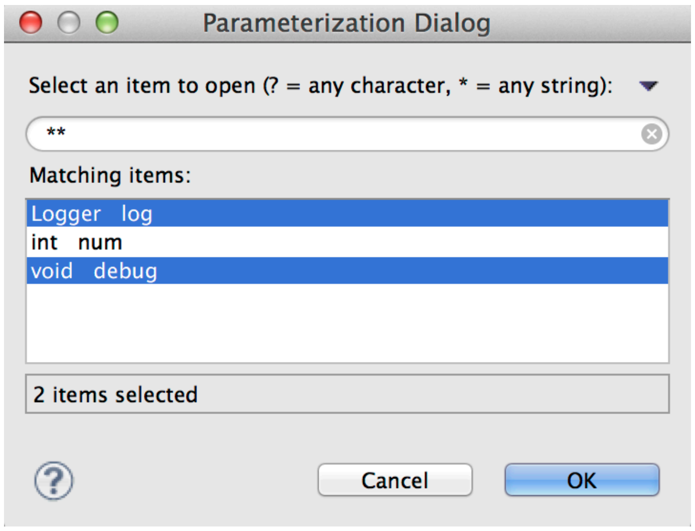
```
boolean process(Order
order)
    if (num < 0)
+     log.debug("All deal
-     System.out.println(
[Debug]All delivers are
busy, please wait.");
```

Figure 3.5: Searching results and diff details.

automate this process, he selects the changed code in both old and new versions and then provides the selected code as an input to CRITICS for further search.

Diff Template View. CRITICS abstracts and visualizes the selected change as an abstract *diff template*. Barry can review and customize the template in a side-by-side **Diff Template View** (see ② in Figure 3.4). The diff template serves as a change pattern for searching similar edits. In the template, CRITICS includes *change context*—unchanged, surrounding statements relevant to the selected change in the template. In this example, CRITICS includes an `if` statement because the changed code is executed only if the `if` condition is satisfied. Nodes with light blue color refer to statements that the user originally selected. Yellow nodes represent statements that the change is control dependent on. Green nodes represent parent nodes of the selected code. Orange nodes represent statements that the change is data dependent on. Barry can preview the textual template in the **Textual Diff Template View** (see ⑥ in Figure 3.4).

Matching Result View. Based on the diff template, CRITICS identifies similar changes and locates anomalies. It reports them in the **Matching Result View**. If the syntactic differences match the diff template in both the old and new versions, CRITICS summarizes this location as *similar change* in **Matching Locations** (see ③ in Figure 3.4). If the target code matches the old version but does not match the new version, such unpairing is reported as *anomalies* in **Inconsistent Locations** (see ④ in Figure 3.4). In the first attempt, Barry does not edit the template and searches matching locations. CRITICS summarizes locations that are identical to the template and reports those violations against the template, as shown in Figure 3.5. The `bake` method is detected as a possible anomaly, because it shares the same


Parameterization Dialog

Select an item to open (? = any character, * = any string):

**

Matching items:

Logger log
int num
void debug

2 items selected

Figure 3.6: Dialog for parameterizing type, method, and identifier names in an abstract diff template.

context in the old revision but Alice did not update this method.

Diff Details View. When Barry clicks an individual change location in the **Matching Result View**, the corresponding differences are presented in the **Diff Details View** (see ⑤ in Figure 3.4). Changed code is highlighted in this view, and inserted code is marked with '+', while deletion is marked with '-'. By comparing contents in the **Diff Details View** and those in the **Diff Template View**, Barry can quickly figure out why each location is identified as a similar change or reported as an anomaly, without navigating different files back and forth. If he wants to drill down into the source code and double clicks a location, CRITICS redirects him to the change location in the **Eclipse Compare View**.

Template Refinement and Search. To match similar but not identical changes, Barry can generalize identifiers in the diff template, including type, variable, and method names, as shown in Figure 3.6. When he generalizes variable `log` and method `debug`, CRITICS locates method `deliver` which uses variable `myLogger` and invokes method `error` instead of `log` and `debug`. Barry further excludes a context node, an `if` statement, in the template by double clicking the node. This time CRITICS reports a change within a `while` loop in method `run` in the **Matching Result View** in Figure 3.4. Barry can progressively explore the diff patch and search for similar changes till he is confident that all locations are updated correctly.

3.6 Evaluation

We evaluated Critics using two different methods. First, we conducted a user study with professional software engineers to understand how Critics can help them during code reviews. Engineers at Salesforce.com used Critics to investigate the real patches found in their version history, authored by their own team. This study emulates the realistic code review scenarios and solicits authentic feedback on the use of Critics in the real world. Second, we conducted a lab study at the University of Texas at Austin, where twelve participants investigated diff patches using both Critics and Eclipse diff and search. We selected Eclipse diff and search as a baseline, because they are default features in Eclipse. We cannot use existing clone-based search tools as a baseline, because they are not designed for inspecting diff patches and thus participants cannot use them without adapting the tools to inspect diff patches. These two evaluation methods (hands-on trials followed by semi-structured interviews and a controlled experiment using human subjects) complement each other by assessing the benefits of Critics both qualitatively and quantitatively.

3.6.1 User Study with Professional Developers at Salesforce.com

We recruited six participants from Salesforce.com. The participants included two software developers, three quality engineers, and a project manager from the same team. This team develops a platform for other teams to process and manage big data stored in the cloud. The participant names and the product name are anonymized.

All six participants had at least three years of Java development experience in industry. Five reported that they conduct code reviews at least weekly, using Collaborator, a default code review tool at Salesforce.[3] Although one manager said he seldom reviews others' changes, we still interviewed him, because he could provide valuable feedback from a man-

[3]http://smartbear.com/products/software-development/code-review/

ager's perspective. Table 3.1 shows the demographic information about the six participants.

Table 3.1: The demographic information of study participants

Subject	Role	Gender	Age	Java Experience	Code Review Frequency
1	Developers	Male	21-30	4	Weekly
2	Quality Engineer	Female	21-30	3	Weekly
3	Manager	Male	41-50	4	Seldom
4	Quality Engineer	Male	21-30	5	Weekly
5	Quality Engineer	Female	31-40	10	Weekly
6	Developers	Male	41-50	14	Daily

In terms of a study procedure, we first gave a presentation to introduce Critics's features to the participants. This presentation included a twenty-minute live demo of how to use Critics Eclipse plug-in. To get accurate and comprehensive feedback, participants were then asked to use CRITICS to investigate one of the four diff patches authored by their colleagues. This could simulate hands-on experience of using Critics in a real world setting, because the participants reviewed patches from their own system.

The four patches came directly from the version history of the Salesforce codebase that they currently work on. We selected the patches that include similar changes to multiple files, because the goal of Critics is to help developers examine similar changes and find potential anomalies. Table 3.2 describes the associated commit log descriptions, the size of the patches in terms of changed lines of code, and the number of changed files from the actual version history. While we do not disclose the size of the Salesforce codebase for confidentiality, we report that Critics is a mature tool that scales to an industrial-scale project and the participants did not have any problems running Critics on their codebase and patches.

For individual participants, the hands-on use of Critics lasted about 20 to 30 minutes. Afterward, we conducted a semi-structured interview to solicit their feedback on the utility of Critics. The advantage of semi-structured interviews is that they are flexible enough to allow unforeseen types of information to be recorded [207]. The interviews were audio-recorded and transcribed later for further analysis. The interview questions are described

Table 3.2: Diff patches from Salesforce.com used for the study

ID	Commit Description	Change Size (LOC)	# of Changed Files
1	Refactor test cases by moving bean maps to utils classes	743	22
2	Refactor the API to get versioned field values by passing the version context as a parameter	943	34
3	Refactor tests by using try-with-resources statements to ensure resource objects are released after program exits	484	10
4	Update common search tests by getting versioned test data	2224	12

below.

- What kind of challenges do you face when you conduct code reviews?
- In which situation, do you think Critics can help improve code reviews in your team?
- Would you like to have Critics be integrated with the code review tool you are currently using?
- How do you like or dislike Critics?

The interview results are organized by the questions raised during the interviews.

What kind of challenges do you face when you conduct peer code review? Collaborator allows developers to upload, compare, and comment patches during code reviews. However, participants find it hard to review similar changes, since Collaborator only highlights differences on the uploaded patches, lacking the ability to identify underlying similar change patterns and pinpoint overlooked mistakes.

"Since REST APIs across different versions generally share similar code snippets, refactoring on versioned APIs often involves similar changes. Unfortunately, these changes are not always exactly the same, including subtle differences in different locations."

"It is hard for us to find missing updates, especially if the reviewer is not familiar with the codebase. So we totally depend on regression testing to check if there is any location we

forgot to change, assuming it (the overlooked change) will break test cases. But, honestly, it does not work very well."

In which situations do you think Critics can help improve code reviews in your team? The participants mentioned that CRITICS can help them inspect system-wide changes, so that they do not need to manually walk through each changed location line by line. They also discussed that code reviews are usually assigned to senior developers and consequently piled up on them, since they are familiar with the codebase and are more likely to notice oversight errors. They believed that the interactive search process of CRITICS is an efficient method for novices to perform code reviews, relieving the burden of senior developers and spreading knowledge between team members.

"Because currently in our company, reviewers only ensure the logic correctness and coding style in uploaded patches. They barely check if there is any missing update, unless a reviewer is very familiar with the codebase and knows where the developer should update. That is also why we always assign code reviews to senior developers in the scrum team. The feature in your tool can free us from piling code review tasks on our senior developers, since it can do the inspection automatically without requiring deep knowledge of the codebase."

*"*CRITICS *would be helpful to check some API updates in our projects. For example, an API from one team is updated and the old API is deprecated. Since people only change the locations they know and reviewers usually do not intentionally check unchanged areas, we cannot guarantee all locations are updated as expected. So using* CRITICS *could help us find out all the locations that need to be updated in the early stage so that we do not need to wait for regression testing or even worse, the customer to tell us if there is any place that we updated incorrectly or forgot to update."*

Would you like to have Critics be integrated with your current code review tool? All six participants provided strong positive answers and believed that it would be useful to have CRITICS integrated to their code review tool, COLLABORATOR.

"Definitely. It makes sense to integrate it with COLLABORATOR*, since it will save a lot of time for code review."*

"Of course. Currently COLLABORATOR *only highlights the changed location in a very naive way. A feature like extracting and visualizing the change context can help us better understand the change itself as well as find some underlying change patterns between related changes."*

How do you like or dislike Critics? They thought CRITICS would be a good time saving tool for code reviews. Four participants replied that they like the search feature a lot because of its flexibility and interactivity compared with existing textual search. Two participants shared the UI is not very intuitive at a first glance and it took some time for them to grasp the UI.

"I like it since it is a great time saving tool for code review and I think its ability to find similar changes can be useful in our work."

"It will be more interesting if you can provide the change skeleton by default in the tree graph and enable users to expand a node to see details if they want to."

In summary, after using CRITICS to investigate their own team's patches, participants told us that CRITICS can improve developer productivity in code reviews and should be integrated to COLLABORATOR. Professional engineers encounter challenges when reviewing system-wide code changes. Currently, in their work environment, they barely have any reliable mechanism to guarantee all locations are correctly modified. Participants think CRITICS would help them detect unnoticed locations. Its interactive search feature also makes it easier for less experienced developers to use the tool. All participants strongly affirmed that they would like to have CRITICS's features as a part of their current code review environment.

3.6.2 Lab Study: Comparison with Eclipse Diff and Search

We conducted a user study with 12 participants to further evaluate the efficiency and usability of CRITICS.

- RQ1: How accurately does a reviewer locate similar changes with CRITICS in comparison to Eclipse diff and search?

Table 3.3: The description of two patches and corresponding questions for code review tasks.

	Version	Change Description	Similar Change	Inconsistent Change	Missing Update	Size (LOC)

Patch1 (Simple)	JDT 9800 vs. JDT 9801	initiate a variable in a for loop instead of using a hashmap	getTrailingComments(ASTNode)
			getLeadingComments(ASTNode)
			getExtendedEnd(ASTNode)
				getExtendedStartPosition(ASTNode)	getComments(ASTNode)
					getCommentsRange(ASTNode)
						190

	Q1. Given the change in the method `getTrailingComments`, what other methods containing similar changes can you find? Count the number. A. 0 B. 1 C. 2 D. 3 or more Answer: C. `getLeadingComments` and `getExtendedEnd`. Q2. Which of the following methods contains inconsistent changes compared with the change in `getTrailingComments`? A. `storeTrailingComments` B. `getExtendedEnd` C. `getLeadingComments` D. `getExtendedStartPosition` Answer: It uses a wrong expression, `i<=this.leadingPtr` instead of `range==null && i<=this.trailingPtr`. Q3. How many methods share context similar to the change in `getTrailingComment` but missed the similar update? A. 0 B. 1 C. 2 D. 3 or more Answer: C. `getComments` and `getCommentsRange`.					
	Version	Change Description	Similar Change	Inconsistent Change	Missing Update	Size (LOC)

Patch2 (Complex)	JDT 10610 vs. JDT 10611	extract the logic of unicode traitement to a method	getNextChar()
			getNextCharAsDigit()
			getNextToken()
			...
			9 locations in total
				getNextCharAsJavaIdentiferPart()	jumpOverMethodBody()
					getNextChar(char, char)
					getNextToken()
					... 11 locations in total,
					all located in another file
						680

	Q1. Given the change in the method `getNextChar`, what other methods containing similar changes can you find? A. 0 B. 1-5 C. 6-9 D. 10 or more Answer: C. `getNextCharAsDigit()` and `getNextToken()`, ... 8 methods in total Q2. Which of the following methods contains inconsistent change compared with the change in `getNextChar`? A. `getNextCharAsDigit` B. `getNextCharAsJavaIdentiferPart` C. `jumOverMethodBody` D. `jumpOverUnicodeWhiteSpace` Answer: It invokes a wrong method, `jumpOverMethodBody` instead of `getNextUnicodeChar`. Q3. How many methods share context similar to the change in `getNextChar` but missed the similar update? A. 0 B. 1-5 C. 6-9 D. 10 or more Answer: D.`jumpOverMethodBody`, `getNextToken` ... 11 methods in total.					

- RQ2: How correctly can a reviewer detect change anomalies with Critics in comparison to Eclipse diff and search?
- RQ3: How much time can a reviewer save in a code review task when using Critics?

In this study, we used counterbalancing to control the order effect. Each participant carried out two different code review tasks, Patch 1 and Patch 2, once using Critics and once with Eclipse diff and search. In this section, we refer to the setting of Eclipse without Critics as *diff* in short. Patch 1 is a simple patch with 190 changed lines, and Patch 2 is a complex patch with 680 changed lines. Both the order of the assigned tools and the order of the assigned tasks were randomized to mitigate the learning effect. Table 3.3 describes the patches in terms of data source, patch size (LOC), change description as well as the number of methods that contain similar changes, inconsistent changes, and missing updates. It also describes the user study questions for each patch.

Four of the twelve participants were electrical and computer engineering undergraduate students and the other eight were all graduate students in software engineering. All participants had at least one year experience in using the Eclipse IDE. All but one participant had code review experience with diff tools such as Eclipse diff and Git/SVN diff. Participation was voluntary with no compensation offered.

Prior to each study session, all participants were given a twenty-minute tutorial to learn how to use Critics. We gave them a live demo about inspecting a diff patch with Critics. Participants first answered two warm-up questions about the assigned diff patch. Then they were given a time to inspect a diff patch and answer three questions about similar changes. All study tasks concern answering questions about similar changes, because the goal of Critics is to support inspection of similar changes, not all types of code changes. The questions required participants to identify methods that were changed similarly to a given location and to search for potential anomalous locations where similar edits were incorrect or completely missing. The three questions are described in Table 3.3. Though these questions were initially designed as multiple-choice questions for ease of quantification, we asked participants to explicitly identify individual method locations during code reviews.

Participants were also encouraged to speak aloud during each task. Table 3.4 shows the percentage of correct answers for each tool. To measure efficiency, we recorded the task completion time from when participants started to inspect source code to the time when they submitted the answers.

At the end of the user study, each participant was asked to complete a post study survey to evaluate their experience with CRITICS and Eclipse diff. First, they were asked to rate CRITICS and Eclipse diff separately on the aspects of *relevance*, *clarity*, and *usefulness* for locating similar changes and detecting anomalies. The survey also included open-ended questions to solicit qualitative feedback on how the users like or dislike CRITICS and the suggestions for improving CRITICS.

Table 3.4: Average correctness of participants' answers with and without CRITICS. For example, (2/6) means that two out of six participants answered the question correctly.

	Q1 (Similar Changes)		Q2 (Inconsistent Changes)		Q3 (Missing Updates)		Time	
	CRITICS	Diff	CRITICS	Diff	CRITICS	Diff	CRITICS	Diff
Patch 1 (Simple)	100% (6/6)	50% (3/6)	100% (6/6)	33% (2/6)	100% (6/6)	83% (5/6)	0:18:32	0:20:24
Patch 2 (Complex)	67% (4/6)	50% (3/6)	100% (6/6)	83% (5/6)	83% (5/6)	33% (2/6)	0:20:20	0:30:53

Identifying Similar Changes. Overall, participants using CRITICS found 13% more similar locations than those using Eclipse diff and search. However, paired t-test failed to reject the null hypothesis since the mean difference between two conditions was not statistically significant (t=1.56, df=11, p-value=0.1462). This was mainly because, in the control condition, one pariticipant (P1) failed to find any similar locations within the given time, while the rest participants could find most similar locations (91.5% on average) using the text search feature in Eclipse. P1 explained that it was too hard to find the right keywords to find other similar locations using text search. After removing this outlier data, the mean difference between two groups was 5%, which was statistically significant (t=2.2473, df=10, p-value=0.0484). Despite the small difference, through interactive template customization and AST-based tree matching, CRITICS helped participants accurately find those locations that are often hard to be found using text search. In particular, 10 out of 12 participants found all similar locations correctly using CRITICS, while only 5 participants found all similar locations using text search. Depending on the choice of keywords, text search may return too many or too few locations. Therefore, participants often stopped refining their keywords once they found several but not all similar locations. However, by examining consistent and inconsistent locations compared with a search template, participants using Critics quickly re-evaluated their own understanding of similar locations and decided which code parts to parameterize in the template. Compared with the simple patch (Patch 1), the complex patch (Patch 2) required more template configuration and search iterations in Critics, and participants often stopped refining the template after two or three iterations. At that point, the customized template was still not general enough to find all similar changes.

Detecting Inconsistent Changes. Participants using Critics found 42% more inconsistent changes than those using Eclipse diff and search (paired t-test, t=2.80, df=11, p-value=0.01718). In particular, all 12 participants using Critics found all expected inconsistent change locations in both simple and complex patches, as opposed to 7 out of 12 with Eclipse diff. Manually identifying subtle inconsistencies among similar locations was challenging, since Eclipse diff was only capable of computing pairwise program differences and thus requires developers to go over each change manually. Critics automatically detected inconsistent changes by contrasting different locations with a given search template.

Detecting Missing Updates. Compared with using Eclipse diff and search, participants found 21% more locations that developers forgot to update using Critics (paired t-test, t=2.234, df=11, p-value=0.04719). 11 out of 12 participants pinpointed all missing updates correctly with Critics, while only 4 out of 12 found missing updates with Eclipse diff. We observed that Eclipse diff was comparable to Critics, when inspecting a simple, small patch (Patch 1 with 160 line changes), while participants could locate a missed update more accurately when using Critics than Eclipse diff for the complex one (Patch 2 with 680 line changes). This was mainly because, when using text search, it was much easier to identify proper search keywords from a small diff patch, while more difficult to select proper keywords from a large, complex patch to look for places that were overlooked to update.

Task Completion Time. Participants saved 6 minutes and 13 seconds with Critics on average, completing the tasks 32% faster than Eclipse diff. However, the mean difference was not statistically significant (paired t-test, t=-1.69, df=s11, p-value=0.1189). Because

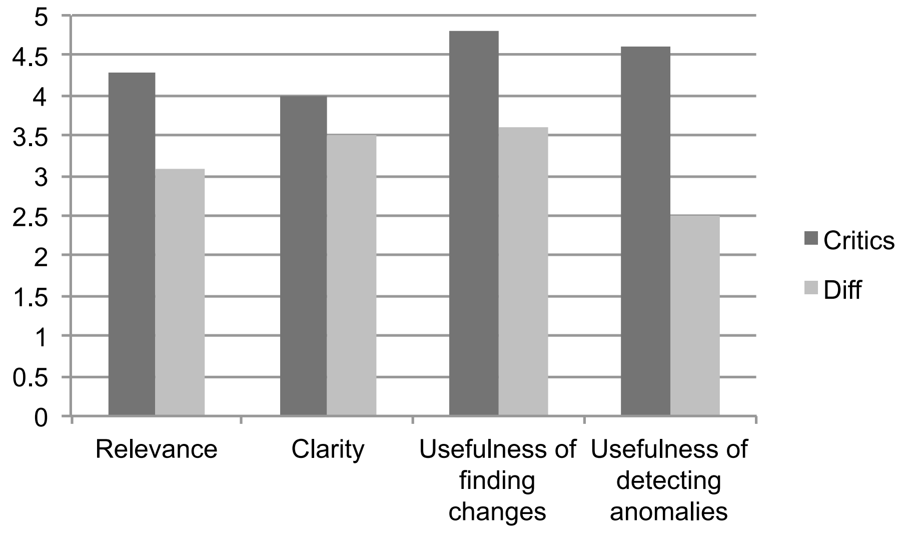


Figure 3.7: User Ratings for Critics and Eclipse Diff & Search (p<0.05 except for clarity)

four participants took longer time to find out which identifiers or statements to parameterize using Critics. As pointed out in the post survey, participants would like Critics to provide hints about which content in a diff template to parameterize. For the simple patch, Critics reduced task completion time by 9% on average, 48% at most. But for the complex patch, it reduced time by 34% on average, 60% at most. Consistent with the goal of Critics to support investigation of similar changes, it was more useful when a patch consists of a large amount of scattered similar edits.

User Feedback. Figure 3.7 shows likert-scale ratings from the post survey. Compared with Eclipse diff, participants gave higher ratings to Critics regarding the relevance of identified similar locations (paired t-test: t=4, df=11, p-value=0.002), usefulness of locating similar changes (paired t-test: t=6.665, df = 11, p-value=3.54e-05), and usefulness of detecting anomalies (t=8.3731, df=11, p-value=4.224e-06). However, the mean difference of ratings between Critics and Eclipse diff was not statistically significnat regarding the UI clarity (paired t-test: t = 1.603, df = 11, p-value = 0.137).

We solicited qualitative feedback from participants to further understand ther ratings. They appreciated that Critics reduces the effort to investigate similar changes, especially in a large system. Using Critics, they only needed to inspect one location, as opposed to reading changed lines file by file without having the global context of what they are reviewing.

"I like the way it (Critics) automatically identifies possible similar edits that I could miss and detects anomalous changes. It really speeds up the code review process."

However, opinions were divided on the usability of Critics's UI. Three participants mentioned that its user interface is not intuitive and would benefit from extra visual options or instructions. They also suggested that Critics should provide configuration hints, e.g., which identifier should be generalized.

"It would be much more straightforward if Critics *gave some hints about which identifiers should be generalized. Currently it seems totally depends on developer's sense."*

"It would be more straightforward for me if separating views for missed and inconsistent changes and only displaying the windows related to code review."

Overall, participants found that, compared with Eclipse diff & search, Critics identified more relevant locations and was more useful in terms of locating similar edits and detecting edit mistakes. They believed that Critics could complement the use of diff during inspection of similar changes.

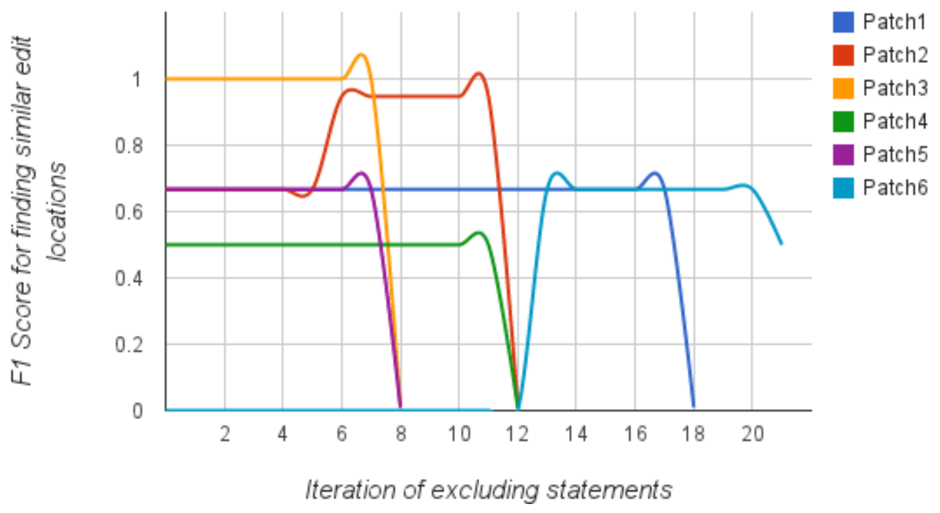
3.6.3 Comparison with LASE

Lase [160] automatatically applies similar edits by searching for locations and applying custom edits to individual locations. It requires multiple change examples as input to generate abstract transformation. It is challenging to directly compare Critics with Lase, because Lase's template generation requires multiple examples apriori and is fixed, while Critics is an interactive tool that a human can iteratively configure a template. Therefore, we simulate a human-driven template configuration process in Critics. From the lab study described in the previous section, we find that users follow common patterns while interactively generalizing the selected edit content and context. They usually generalize one identifier or statement at a time and re-run the search; if the search result degrades, they undo the generalization and try a different identifier or statement. In other words, their generalization strategy is similar to the typical *greedy search*. When generalizing identifiers, users first generalize a variable with a long name rather than a short one. When excluding statements, users prefer to exclude the context node on which a change is *control dependent*, such as `if` and `for`. We encode these patterns in a test script to simulate the interactive use of CRITICS. Then we compare LASE's accuracy with CRITICS's accuracy at each iteration.

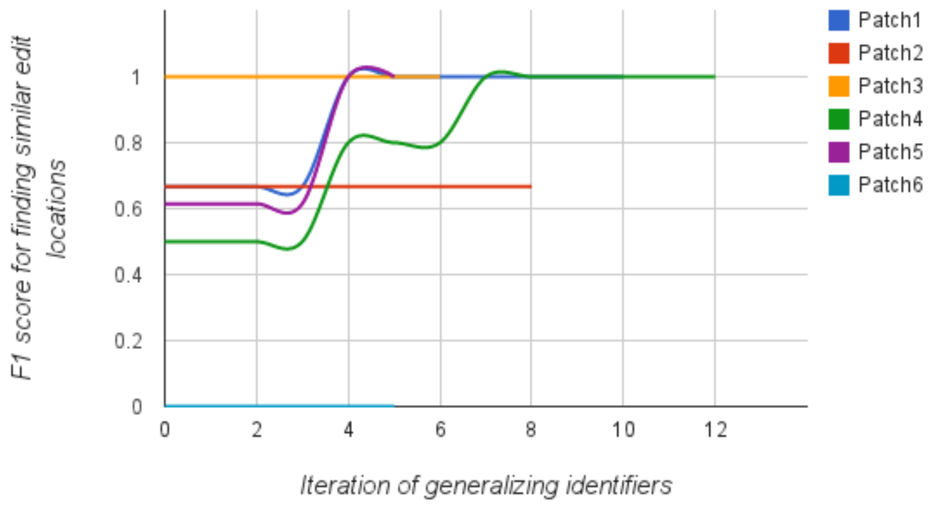
The oracle test suite is drawn from the similar edits identified by Park et al. [172] in Eclipse JDT and Eclipse SWT and consists six sets of similar changes. In this test suite, the patch size ranges from 190 to 680 lines of edits. The number of locations with similar changes ranges from three to ten locations. The first two changed locations are used as input examples to LASE, using the same approach described in Meng et al. [160]. Figure 3.8 describes the accuracy variation in CRITICS's simulation. Figure 3.8a represents F_1 score (a harmonic mean of precision and recall) for finding similar changes while varying the number of excluded nodes. Figure 3.8b represents F_1 score for finding similar changes, while varying the number of generalized identifiers. Table 3.5 shows the comparison of search accuracy between CRITICS and LASE, including the iteration numbers till CRITICS achieves the best result and the average execution time for each iteration. In five out of six cases, CRITICS achieves the same or higher accuracy than LASE within a few iterations, showing the benefit of interactive template configuration as opposed to fixed template configuration.

Table 3.5: Comparison between CRITICS and LASE

	CRITICS				LASE	
	Precision	Recall	Iteration	Time (sec)	Precision	Recall
Patch 1	1	1	4	1.66	1	1
Patch 2	1	0.9	6	8.95	0.92	0.75
Patch 3	1	1	0	13.52	1	1
Patch 4	1	1	7	71.98	1	0.33
Patch 5	1	1	4	6.86	1	1
Patch 6	1	0.33	3	1.47	1	1
Average	1	0.87	4	17.41	0.99	0.84



(a) F1 score for finding similar edit locations by excluding statements.



(b) F1 score for finding similar edit locations by parameterizing identifiers.

Figure 3.8: F1 score on change template customization.

3.7 Threats to Validity

In terms of *construct validity*, in our lab study, we measured the correctness of the answers and the time taken to answer questions to measure developer productivity in inspecting similar changes. Other measures such as the number of potential bugs detected could be used to measure developer productivity for peer code reviews. In our user study, we used both large and small patches and counterbalanced the order and task assignment to mitigate learning effect. Because the goal of Critics is to help inspect similar changes, the questions mainly pertain to the questions about similar scattered changes, not general program comprehension questions.

In terms of *external validity*, in our lab study, twelve student developers were not familiar with Eclipse JDT, from where patches are drawn. The lab study may not generalize to professional developers who are familiar with their codebase. To overcome this limitation, in our user study at Salesforce.com, six engineers investigated the patches from their own system.

The study at Salesforce is a qualitative study based on six interviews. Because of the qualitative nature of the study, we do not make any quantitative statements about how much productivity gain Critics can provide in comparison to their current code review tool, Collaborator. The study was conducted only in one company. We do not believe this is a significant limitation because the background of the participants and the code review practice at Salesforce.com are similar to other large software companies. In the comparison with Lase, our test suite of similar changes includes only patches from Park et al.'s data set [172] and may not generalize to projects other than Eclipse JDT and SWT.

To mitigate *internal validity*, in our lab study, before the participants started the task, we asked them to inspect the change example first and answer two questions to calibrate their understanding. The first question required them to choose true or false about detailed statements about the change to ensure that they have carefully inspected the example. The second question required them to identify changes similar to the given example. The warm up questions helped them better understand change similarity.

3.8 Summary

This chapter describes an interactive code review approach called Critics that summarizes similar program edits in a diff patch and identifies edit inconsistencies in other similar locations. Compared with existing code search and clone inconsistency detection techniques, Critics allows developers to express a program change of interest as an abstract change template and to incrementally refine the template based on previous search results. A user study at Salesforce.com shows that Critics scales to an industry-scale project and can be easily adopted by professional developers. Another user study with twelve students shows that, in code review tasks, participants using Critics found more similar edits and detected more edit mistakes in less time, compared with using code review features in Eclipse. The advantage of interactive template construction is further demonstrated by a comparison with an automated template construction approach, where Critics achieves the same or higher code search accuracy in most cases.

Critics detects syntactic inconsistencies among similar edits but it is not capable of investigating runtime behavioral discrepancy caused by such inconsistent edits. In the next chapter, we propose a complementary approach, Grafter to reuse tests and compare runtime behaviors between clones via code transplantation and differential testing.

CHAPTER 4

Automated Test Transplantation for Similar Programs

This chapter presents Grafter, a differential testing technique to reuse tests among similar code and examine behavioral similarities and differences. Compared with Critics, which detects syntactic inconsistencies among similar program locations, Grafter further enables developers to investigate behavioral discrepancies among these locations.

4.1 Introduction

When copying and pasting code fragments, developers express the desire to examine and contrast runtime behavior of clones [74, 101]. Furthermore, in the user study of Critics, Salesforce developers said "we totally depend on regression testing to check if there is any location we forgot to change, assuming it will break test cases." However, regression testing may not work well due to a lack of test cases. Our manual analysis of 56 pair of three open-source projects shows that, in 46% of clone pairs, only one clone is tested by existing tests, but not its counterpart (to be detailed in Section 4.5). No existing techniques can help programmers reason about runtime behavior differences of clones, especially when clones are not identical and when clones are not tested. In the absence of test cases, developers can only resort to static analysis techniques to examine clones [111, 140, 177, 188, 263], but these techniques are limited to finding only pre-defined types of cloning bugs such as renaming mistakes or control-flow and data-flow inconsistencies.

Given a pair of clones and an existing test suite, Grafter helps developers examine the behavioral differences between these clones by exercising them using the same test. Test reuse for clones is challenging because clones may appear in the middle of a method without a well-defined interface (i.e., explicit input arguments and return type), which also makes it hard to directly adapt test for reuse. Such intra-method clones are often found by widely-used clone detectors such as Deckard [111] or CCFinder [121]. Grafter identifies input and output parameters of a clone to expose its de-facto interface and then grafts one clone in place of its counterpart to exercise the grafted clone using the same test.

Similar to how organ transplantation may bring incompatibility issues between a donor and its recipient, a grafted clone may not fit the context of the target program due to variations in clone content. For example, if a clone uses variables or calls methods that are not defined in the context of its counterpart, simply copying a clone in place of another will lead to compilation errors. To ensure *type safety* during grafting, Grafter performs inter-procedural analysis to identify variations in referenced variables and methods. It then adapts the grafted clone using five transplantation rules to handle the variations in referenced variables, types, and method calls. Finally, it synthesizes stub code to propagate input data to the grafted clone and then transfers intermediate outputs back to the recipient. Grafter supports differential testing at two levels: test outcomes (i.e., *test-level comparison*) and intermediate program states (i.e., *state-level comparison*). During differential testing, Grafter does not assume that all clones should behave similarly nor considers that all behavioral differences indicate bugs. In fact, a prior study on clone genealogies [126] indicates that many syntactically similar clones are used in different contexts and have intended behavioral differences. The purpose of differential testing in Grafter is rather to illuminate and expose behavioral differences at a fine-grained level *automatically* and *concretely* by pinpointing which variables' states differ in which test.

We evaluate Grafter on 52 pairs of nonidentical clones from three open-source projects: Apache Ant, Java-APNS, and Apache XML Security. Grafter successfully grafts and reuses tests in 49 out of 52 pairs of clones without inducing compilation errors. Successfully reusing tests in 94% of the cases is significant, because currently no techniques enable test reuse for nonidentical clones appearing in the middle of a method. Grafter inserts up to 33 lines of stub code (6 on average) to ensure type safety during grafting, indicating that code transplantation and data propagation in Grafter are not trivial. To assess its fault detection capability, we systematically seed 361 mutants as artificial faults using the Major mutation framework [116]. We use Jiang et al.'s static cloning bug finder [111] as a baseline for comparison. By noticing runtime behavioral discrepancies, Grafter is more robust at detecting injected mutants than Jiang et al.—31% more using the test-level comparison and almost 2X more using the state-level comparison. Grafter's state-level comparison also narrows down the number of variables to inspect to three variables on average. Therefore, Grafter should complement static cloning bug finders by enabling runtime behavior comparison. Our grafting technology may also have potential to assist code reuse and repair [41, 96, 176, 251].

The rest of this chapter is organized as follows. Section 4.2 illustrates a motivating example. Section 4.3 describes how Grafter reuses tests from its counterpart clone by grafting a clone. Section 4.5 describes the evaluation of Grafter and comparison to Jiang et al. Section 4.6 discusses threats to validity.

4.2 Motivating Example

This section motivates Grafter using an example based on Apache Ant. The change scenario is constructed by us to illustrate the difficulty of catching cloning bugs. Figure 4.1 shows the pair of inconsistently edited clones, one from the `setIncludes` method in the `Copy` class (lines 6-15 in Figure 4.1a) and the other from the `setExcludes` method in the `Delete` class (lines 6-15 in Figure 4.1b). These clones are syntactically similar but not identical—the left program uses a field `includes` of type `IncludePatternSet` while the right program uses a field `excludes` of type `ExcludePatternSet`. The `Copy` class implements the task of copying files matching the specified file pattern(s). On the other hand, `Delete` removes files that do not match the pattern(s). Methods `setIncludes` and `setExcludes` both split the input string by a comma and add each pattern to a pattern set, `includes` and `excludes` respectively. Figure 4.2 shows a test case, `testCopy`, which creates a `Copy` object, specifies two copied file patterns as a string `"src/*.java, test/*.java"`, and then checks if all java files in the `src` folder and the `test` folder are copied to a target directory. However, the

Delete class is not tested by any existing test.

```
1  public class Copy extends Task{
2   private IncludePatternSet includes;
3
4   public void setIncludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7  -    StringTokenizer tok=new StringTokenizer(
        patterns,",");
8  -    while(tok.hasMoreTokens()){
9  -      includes.addPattern(tok.next());
10 -    }
11 +    String[] tokens = StringUtils.split(patterns, "
        ,");
12 +    for(String tok : tokens){
13 +      includes.addPattern(tok);
14 +    }
15    }
16  }
17   ...
18 }
19
20 public class IncludePatternSet{
21  public Set<String> set;
22  public void addPattern(String s) { set.add(s); }
23   ...
24 }
```

(a) Correctly edited clone in the Copy class

```
1  public class Delete extends Task{
2   private ExcludePatternSet excludes;
3
4   public void setExcludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7  -    StringTokenizer tok=new StringTokenizer(
        patterns,",");
8  -    while(tok.hasMoreTokens()){
9  -      excludes.addPattern(tok.next());
10 -    }
11 +    String[] tokens = StringUtils.split(patterns, "
        .");
12 +    for(String tok : tokens){
13 +      excludes.addPattern(tok);
14 +    }
15    }
16  }
17   ...
18 }
19
20 public class ExcludePatternSet{
21  public Set<String> set;
22  public void addPattern(String s) { set.add(s); }
23   ...
24 }
```

(b) Inconsistently edited clone in the Delete class

Figure 4.1: Similar edits to update the use of StringTokenizer API to StringUtils.split in Copy and Delete.

```
@Test
public void testCopy(){
  Task copyTask = FileUtils.createTask(FileUtils.COPY);
  ...
  copyTask.setIncludes("src/*.java, test/*.java");
  JobHandler.fireEvent(copyTask);
  assertTrue(checkFileCopied());
}
```

Figure 4.2: A test case for the Copy class.

StringTokenizer is a legacy class and its usage is now discouraged in new code. Therefore, Alice updates the use of StringTokenizer API to StringUtils.split in both Copy and Delete in Figure 4.1. However, she accidentally changes the separator from ‘,’ to ‘.’ in Delete (line 11 in Figure 4.1b). Such mistake is difficult to notice during manual inspection, as these programs are similar but not identical. An existing cloning bug finder by Jiang et al. would fail to find the mistake, as it checks for only three pre-defined cloning

```
1    public class Copy extends Task{
2      private IncludePatternSet includes;
3  +   private ExcludePatternSet excludes;
4
5      public void setIncludes(String patterns){
6        ...
7        /* this is stub code inserted for data transfer*/
8  +     ExcludePatternSet excludes_save = excludes;
9  +     excludes = new ExcludePatternSet();
10 +     excludes.set = includes.set;
11
12       /* the original code is replaced with the grafted code
              from setExcludes*/
13 -     if(patterns != null && patterns.length() > 0){
14 -       String[] tokens = StringUtils.split(patterns, ",");
15 -       for(String tok : tokens){
16 -         includes.addPattern(tok);
17 -       }
18 -     }                                      Original Clone (Deleted)
19 +     if(patterns != null && patterns.length() > 0){
20 +       String[] tokens = StringUtils.split(patterns, ".");
21 +       for(String tok : tokens){
22 +         excludes.addPattern(tok);
23 +       }
24 +     }                                      Grafted Clone (Inserted)
25
26       /* this is stub code inserted for data transfer*/
27 +     includes.set = excludes.set;
28 +     excludes = excludes_save;
29     }
30   }
```

Figure 4.3: Grafter grafts the clone in `Delete` (lines 19-24) in place of the original clone in `Copy` (lines 13-18) for test reuse. Grafter inserts stub code (highlighted in yellow).

bug types via static analysis [111]: renaming mistakes, control construct inconsistency, and conditional predicate inconsistency. Accidentally replacing the separator does not belong to any of the pre-defined cloning bug types.

To reuse the same test `testCopy` for `Delete`, Grafter grafts the clone from `Delete` in place of the original clone in `Copy`, as shown in Figure 4.3. As the grafted code uses an undefined variable `excludes`, Grafter also ports its declaration to `Copy.java`. Grafter ensures that the grafted clone receives the same input data by populating `excludes` with the value of `includes` (lines 8-10) and transfers the value of `excludes` back to `includes` (lines 27-28). Therefore, the value of `excludes` can flow into the same assertion check of the original test. Additional stub code generated by Grafter is highlighted in yellow in Figure 4.3.

After grafting, Grafter then runs `testCopy` on both clones and finds that the test now fails on `Delete`, because the string is not split properly. To help Alice further diagnose failure symptoms, Grafter shows that `tokens` has a list { `"src/*.java"`, `"test/*.java"`} in `Copy` but { `"src/*"`, `"java, test/*"`, `"java"` } in `Delete` due to a wrong split. Grafter also shows that this difference has propagated to corresponding variables `includes` and `excludes`.

4.3 Automated Code Transplantation and Differential Testing

Grafter takes a clone pair and an existing test suite as input and grafts a clone from the donor to the recipient to make the test(s) of the recipient executable for the donor. A *donor* program is the source of grafting and a *recipient* program is the target of grafting. Grafter does not require input clones to be identical. Clones could be the output of an existing clone detector [109, 121] or be supplied by a human. For example, lines 6-15 in setIncludes and lines 6-15 in setExcludes are clones found by Deckard [111] in Figure 4.1. Delete.java is the *donor* program and Copy.java is the *recipient* program, as Alice wants to reuse the test of Copy.java for Delete.java.

Grafter works in four phases. Grafter first analyzes variations in local variables, fields, and method call targets referenced by clones and their subroutines (Phase I). It also matches corresponding variables at the entry and exit(s) of clones, which is used for generating stub code and performing differential testing in later phases. Grafter ports the donor clone to replace its counterpart clone and declares undefined identifiers (Phase II). To feed the same test input into the grafted clone, Grafter populates the input data to newly ported variables and transfers the intermediate output of the grafted clone back to the test for examination (Phase III). Finally, it runs the same test on both clones and compares test outcomes and the intermediate states of corresponding variables at the exit(s) of clones. We use Figure 4.1 as a running example throughout this section.

4.3.1 Variation Identification

The goal of Phase I is to identify mappings between method call targets, local variables, and fields at the entry and exit(s) of each clone. Grafter leverages inter-procedural analysis to find identifiers referenced by each clone and its subroutines. It then determines which referenced identifiers are defined in the donor but not in the recipient.

There are three goals with respect to finding variable mappings at the entry and exit(s) of each clone. First, we need to identify variables used by the donor clone but not defined in the recipient clone, so Grafter can port their declarations in Phase II to ensure type safety and avoid compilation errors. Second, we need to decide the data flowing in and out of the clone at the entry and exit(s), so we can insert stub code to populate values between corresponding variables in Phase III. Third, we compare the states of corresponding variables at clone exit(s) for fine-grained differential testing in Phase IV.

These goals are achieved by capturing the *consumed* variables at the entry of the clone region and the *affected* variables at the exit(s) of the clone region in the control flow graph. A variable is *consumed* by a clone if it is used but not defined within the clone. A variable is *affected* by a clone if its value could be potentially updated by the clone. The consumed variables are associated with the data flowing into the clone and the affected variables are associated with the updated data flowing out of the clone. Grafter performs a combination of def-use analysis and scope analysis to identify consumed and affected variables.

Given a clone F and its container method M and class C, *consumed variables* at the clone's entry can be approximated:

$$\mathit{Consumed}(F, M, C) = (\mathit{Def}(C) \cup \mathit{Def}(M) \setminus \mathit{Def}(F)) \cap \mathit{Use}(F)$$

Similarly, given a clone F, affected variables at an exit point P can be approximated as following:

$$\mathit{Affected}(F, P) = \mathit{Use}(F) \cap \mathit{In\text{-}Scope}(P)$$

To assist the derivation of *consumed* and *affected* variables, we define three functions. $Def(F)$ returns the set of variables declared within the fragment F. $Use(F)$ returns the set of variables used within the fragment. $In\text{-}Scope(P)$ returns the set of variables at a program point P. The set of affected variables is an over-approximation of the variables that could be updated by a clone at runtime. This set may include variables only read but not mutated by the clone. However, it is guaranteed to include all variables potentially updated by the clone, thus capturing all data flowing out of it.

Consider `setIncludes` in Figure 4.1. Figure 4.4 shows an inter-procedural control flow graph. Nodes represent corresponding program statements, solid edges represent control flow, and dashed edges represent method invocation. The gray nodes in Figure 4.4 represent the clone region F (line 6 and lines 11-15 in Figure 4.1) in `setIncludes`. The CFG edge entering the clone region is labeled with `<entry>` and the two edges exiting the clone region are labeled with `<exit1>` and `<exit2>`. Each CFG node is labeled with the variables defined and used within the corresponding statement. For example, $Def(F)$ includes `tokens` and `tok`. Variable `patterns` is not included because it is declared as a method parameter in Figure 4.1, which is before the clone region F (line 6 and lines 11-15). $Use(F)$ returns `includes`, `patterns`, `tokens`, and `tok`. The figure does not show the scope of individual variables but we associate each variable with its scope and visibility. For example, *In-Scope*$(< exit_2 >)$ returns `patterns` and `tokens`. Putting these definitions together, the resulting set of consumed variables at the entry of the clone is {`includes`, `patterns`}. The resulting sets of affected variables at the two exit edges are the same: {`includes`, `patterns`}.

By comparing the two sets of consumed variables, {`includes, patterns`} and {`excludes, patterns`} at the entry of clones using name similarity, we find that `includes` and `excludes` are corresponding variables. Therefore, GRAFTER knows that it must port the declaration statement of the field `excludes`. The name similarity is computed using the Levenshtein distance [1], i.e., the minimum number of single-character insertions, deletions, or substitutions required to change one string into the other. The lower the distance is, the more similar two field names are. This mapping information is used to guide the process of redirecting data into the grafted clone and back to the recipient in Phase III. For example, GRAFTER populates the value of `includes` to `excludes` at the entry and transfers the updates on `excludes` back to `includes` at the exit (to be detailed in Section 4.3.3).

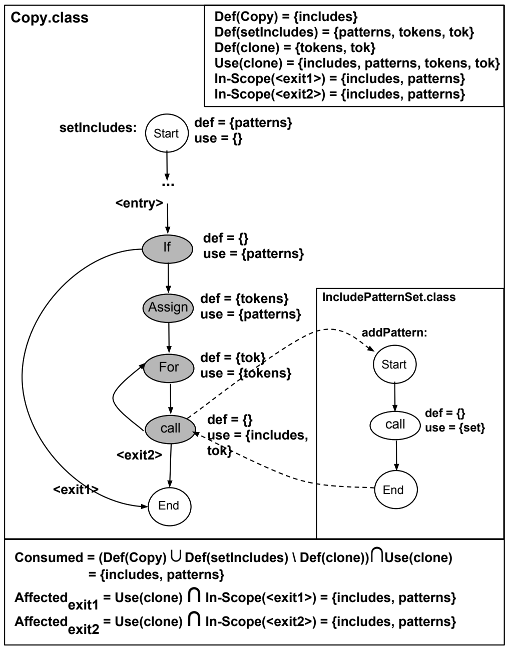

Copy.class
Def(Copy) = {includes}
Def(setIncludes) = {patterns, tokens, tok}
Def(clone) = {tokens, tok}
Use(clone) = {includes, patterns, tokens, tok}
In-Scope(<exit1>) = {includes, patterns}
In-Scope(<exit2>) = {includes, patterns}
setIncludes:
Start
def = {patterns}
use = {}
...
<entry>
If
def = {}
use = {patterns}
Assign
def = {tokens}
use = {patterns}
For
def = {tok}
use = {tokens}
call
def = {}
use = {includes, tok}
<exit2>
<exit1>
End
IncludePatternSet.class
addPattern:
Start
call
def = {}
use = {set}
End
Consumed = (Def(Copy) ∪ Def(setIncludes) \ Def(clone)) ∩ Use(clone)
= {includes, patterns}
Affected_exit1 = Use(clone) ∩ In-Scope(<exit1>) = {includes, patterns}
Affected_exit2 = Use(clone) ∩ In-Scope(<exit2>) = {includes, patterns}

Figure 4.4: An inter-procedural control flow graph for the `setIncludes` method in Figure 4.1. The CFG nodes in the clone region are colored in gray. There is one `entry` edge and two `exit` edges of the clone.

4.3.2 Code Transplantation

Simply copying and pasting a clone in place of its counterpart in the recipient could lead to compilation errors due to variations in clone content. Phase II applies five transplantation rules to ensure type safety during grafting. The rules are *sound* in the sense that the resulting grafted code is guaranteed to compile. To ensure type safety, our rules do not convert objects, if their types are not castable or structurally equivalent. We conservatively choose not to graft clones referencing such unrelated types and give the user a warning instead.

Rule 1. Handle Variable Name Variations. If the grafted clone uses a variable undefined in the recipient, Grafter moves its definition from the donor to the recipient. Consider Figure 4.1 where `setIncludes` and `setExcludes` use different variables, `includes` and `excludes`. When grafting the clone in `setExcludes` to `setIncludes`, Grafter adds the definition of `excludes` in line 3 of `Copy.java` in Figure 4.3. In particular, if the grafted clone uses a variable that has already been defined with a different type in the recipient, Grafter still ports the definition but renames it and all its references by appending `_graft` to avoid a naming conflict.

Rule 2. Handle Method Call Variations. If the grafted clone calls a method undefined in the recipient, Grafter ports its definition from the donor to the recipient. Similar to the rule above, Grafter renames it if it leads to a naming conflict.

Rule 3. Handle Variable Type Variations. If the grafted clone uses a different type compared with its counterpart, Grafter generates stub code to convert the object type to the one compatible with the recipient. For example, `includes` and `excludes` in Figure 4.1 have different types, `IncludePatternSet` and `ExcludePatternSet`. Simply assigning `includes` to `excludes` leads to a type error. Thus, Grafter preserves the original value of `excludes` in line 8, creates a new `ExcludePatternSet` instance in line 9, and populates the field `sets` from the `IncludePatternSet` object to the `ExcludePatternSet` object in line 10 in Figure 4.3.

Rule 4. Handle Expression Type Variations. The data type of an expression can be different based on the variables, operators, and method targets used in the expression. Such variation can cause type incompatibility in the returned object if it appears in the `return` statement. Grafter first decomposes the return statement `return X;` into two statements, one storing the expression value to a temporary variable `Type temp = X;` and the other returning the temporary value `return temp;`. Grafter applies the Variable Type Variation rule above on `temp` to convert its type to a compatible type in the recipient.

Rule 5. Handle Recursive Calls. If both container methods in the donor and recipient have recursive calls in the clone region, Grafter updates the recursive call targets in the grafted clone.

4.3.3 Data Propagation

In medicine, surgeons reattach blood vessels to ensure the blood in the recipient flows correctly to the vessels of the transplanted organ. Similarly, Grafter adds stub code to ensure that (1) newly declared variables consume the same input data as their counterparts in the recipient and (2) the updated values flow back to the same test oracle.

Algorithm 2: Heuristics for transfering variable values

Input : Let `v1` and `v2` be a pair of mapped variables. In this algorithm, each variable symbol is an abstraction, containing the name, type, and field information, which guides the generation of stub code.

Output: Let `code` be the stub code to tranfer the value of `v1` to `v2`. It starts with an empty string and ends with a sequence of statements generated using a few heuristics.

```
Algorithm transfer(v1, v2)
    code := “”
    if v1.name == v2.name then
        return “”
    if v1.type == v2.type or v1.type is castable to v2.type then
        return “v2.name = v1.name;”
    if v1.type structurally equivalent to v2.type then
        code += “v2.name = new v2.type();”
        match := stableMatching (v1.fields, v2.fields)
        foreach $f_i$, $f_j$ in match do
            code += transfer ($f_i$, $f_j$)
        return code
Procedure stableMatching($s_1$, $s_2$)
    match := ∅
    unmatch := $s_2$
    while unmatch is not empty do
        $f_2$ := next field in unmatch
        foreach $f_1$ in $s_1$ do
            if $f_1$.type == $f_2$.type or $f_1$.type is castable to $f_2$.type or $f_1$.type is structurally equivalent to $f_2$.type then
                if $f_1 \in$ match.keys then
                    $f'_2$ := match.get($f_1$)
                    $d_1$ := levenshteindistance ($f_1$.name, $f_2$.name)
                    $d_2$ := levenshteindistance ($f_1$.name, $f'_2$.name)
                    if $d_1 < d_2$ then
                        match.put($f_1$, $f_2$)
                        unmatch.add($f'_2$)
                else
                    match.put($f_1$, $f_2$)
    return match
```

Given each mapped variable pair v_1 and v_2 in Phase II, Grafter generates stub code to propagate the value of v_2 to v_1 at the entry of the clone and to transfer the updated value of v_1 back to v_2 at the exit. In Algorithm 2, the main function, `transfer`, takes two variables v_1 and v_2 as input and produces a sequence of program statements for data propagation. The symbols v_1 and v_2 in Algorithm 2 abstract their variable name, type, and field information.

Heuristic A. Given two variables v_1 and v_2 with the same name and type, there is no need to propagate the value from v_1 to v_2. In Figure 4.1, both clones use the method parameter `patterns` and the references to `patterns` in the grafted code are automatically resolved to the same parameter in the recipient. Algorithm 2 returns an empty string in this case.

Heuristic B. Given two variables v_1 and v_2 of the same type or castable types due to subtyping, the value of v_2 can be directly assigned to v_1 without inducing type casting errors. Algorithm 2 adds an assignment statement.

Heuristic C. Given v_1 of type t_1 and v_2 of type t_2, if t_1 and t_2 are *structurally equivalent*, we propagate corresponding sub fields from v_2 to v_1. Two types are structurally equivalent if (1) they have the same number of fields, and (2) for each field in one type, there exists a field in another type that has either the same or structurally equivalent type. For example, `IncludePatternSet` and `ExcludePatternSet` are structurally equivalent because both have only one sub-field `set` of type `Set<String>` in Figure 4.1. To propagate data at the clone entry, Grafter first preserves the original `ExcludePatternSet` object in line 8, creates a new `ExcludePatternSet` instance in line 9, and then populates the field `set` from the `IncludePatternSet`'s `set` field in line 10 in Figure 4.3. At the clone exit, the updates on `excludes` are transferred to `includes` by setting field `set` in line 27 and the original reference is restored to `excludes` in line 28.

Because Grafter allows the fields of structurally equivalent types to have different names and orders, Grafter identifies n-to-n sub-field matching. This problem can be viewed as a stable marriage problem (SMP) and is solved using the Gale-Shapley algorithm [3]. The `stableMatching` procedure in Algorithm 2 establishes field mappings based on type compatibility and name similarity. The `stableMatching` procedure takes two sets of fields, $\texttt{s}_1$ and $\texttt{s}_2$ as input. It creates an empty map `match` and adds all fields in $\texttt{s}_2$ to `unmatch`. For each field $\texttt{f}_2$ in `unmatch`, Grafter compares it with any field $\texttt{f}_1$ in $\texttt{s}_1$. If $\texttt{f}_1$ and $\texttt{f}_2$ have the same or structurally equivalent types and their name similarity is greater than the current match $\texttt{f}_1$ to $\texttt{f}_2'$ (if any), $\texttt{f}_2$ is a better match than $\texttt{f}_2'$. Grafter puts a mapped pair ($\texttt{f}_2$, $\texttt{f}_1$) to `match` and adds $\texttt{f}_2'$ to `unmatch`. This process continues until `unmatch` is empty.

Heuristic D. If a data type or its field is an array or a collection such as `List` or `Set` of the same or structurally equivalent type, Grafter synthesizes a loop, in which each iteration populates corresponding elements using Algorithm 2.

4.3.4 Fine-grained Differential Testing

Grafter supports behavior comparison at two levels.

Test Level Comparison. Grafter runs the same test on two clones and compares the test outcomes. If a test succeeds on one clone but fails on the other, behavior divergence is noted.

State Level Comparison. Grafter runs the same test on two clones and compares the intermediate program states for *affected* variables at the exit(s) of the clones. Grafter instruments code clones to capture the updated program states at the exit(s) of clones. Grafter uses the XStream library[1] to serialize the program states of affected variables in an XML format. Then it checks if two clones update corresponding variables with the same values. State-level comparison is more sensitive than test outcome comparison.

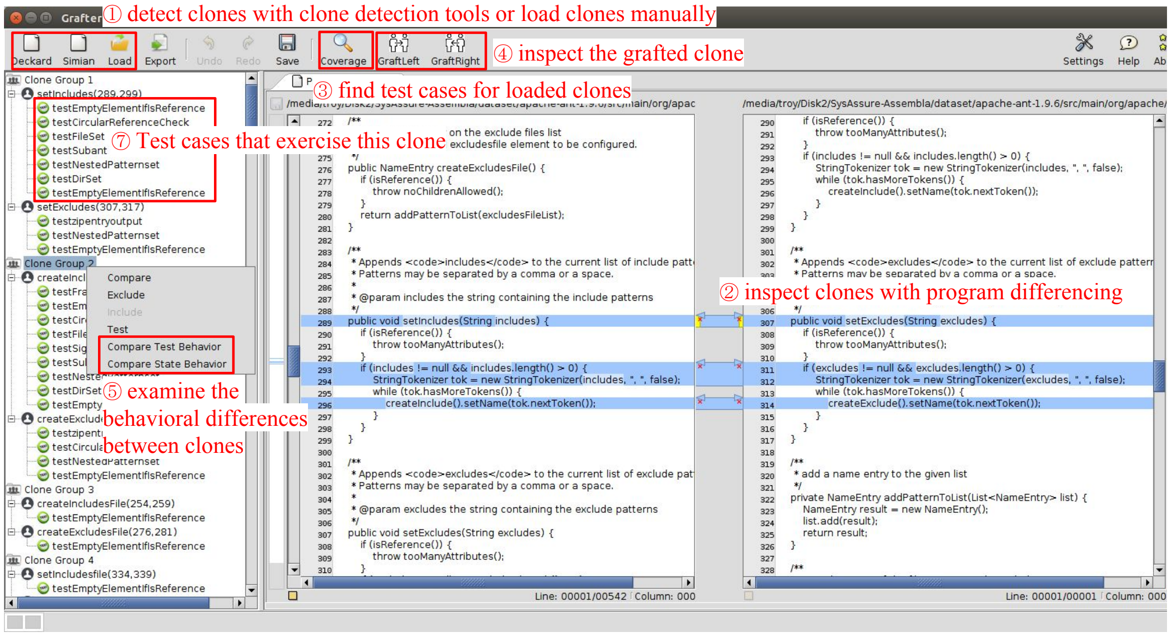
Grafter is publicly available with our experiment dataset.[2] Its GUI allows users to experiment with clone grafting and revert edits after examining runtime behavior. Therefore, the inserted stub code is not permanent and does not need to be comprehended by users. Using GUI, users can easily discard tests that do not preserve the desired semantics.

4.4 Tool Support and Demonstration

This section explains how a user may use Grafter by demonstrating UI features and corresponding screen snapshots. A user can load clones found by an off-the-shelf clone detector and mark clones for further runtime behavior investigation by associating clones with the test coverage. A user can also experiment with grafting and inspect the stub code generated

[1]http://x-stream.github.io/

[2]http://web.cs.ucla.edu/~tianyi.zhang/grafter.html


Grafter
① detect clones with clone detection tools or load clones manually
Deckard
Simian
Load
Export
Undo
Redo
Save
Coverage
GraftLeft
GraftRight
④ inspect the grafted clone
Settings
Help
③ find test cases for loaded clones
⑦ Test cases that exercise this clone
② inspect clones with program differencing
⑤ examine the behavioral differences between clones
Clone Group 1
setIncludes(289,299)
testEmptyElementIfIsReference
testCircularReferenceCheck
testFileSet
testSubant
testNestedPatternset
testDirSet
testEmptyElementIfIsReference
setExcludes(307,317)
testzipentryoutput
testNestedPatternset
testEmptyElementIfIsReference
Clone Group 2
Compare
Exclude
Include
Test
Compare Test Behavior
Compare State Behavior
testNestedPatternset
testEmptyElementIfIsReference
Clone Group 3
createIncludesFile(254,259)
testEmptyElementIfIsReference
createExcludesFile(276,281)
testEmptyElementIfIsReference
Clone Group 4
setIncludesfile(334,339)
testEmptyElementIfIsReference
Line: 00001/00542 Column: 000
Line: 00001/00001 Column: 000

```
/media/troy/Disk2/SysAssure-Assembla/dataset/apache-ant-1.9.6/src/main/org/apac
/**
                     on the exclude files list
                     excludesfile element to be configured.
 */
public NameEntry createExcludesFile() {
    if (isReference()) {
        throw noChildrenAllowed();
    }
    return addPatternToList(excludesFileList);
}

/**
 * Appends <code>includes</code> to the current list of include patt
 * Patterns may be separated by a comma or a space.
 *
 * @param includes the string containing the include patterns
 */
public void setIncludes(String includes) {
    if (isReference()) {
        throw tooManyAttributes();
    }
    if (includes != null && includes.length() > 0) {
        StringTokenizer tok = new StringTokenizer(includes, ", ", false);
        while (tok.hasMoreTokens()) {
            createInclude().setName(tok.nextToken());
        }
    }
}

/**
 * Appends <code>excludes</code> to the current list of exclude pat
 * Patterns may be separated by a comma or a space.
 *
 * @param excludes the string containing the exclude patterns
 */
public void setExcludes(String excludes) {
    if (isReference()) {
        throw tooManyAttributes();
    }
```

```
/media/troy/Disk2/SysAssure-Assembla/dataset/apache-ant-1.9.6/src/main/org/apache/
    if (isReference()) {
        throw tooManyAttributes();
    }
    if (includes != null && includes.length() > 0) {
        StringTokenizer tok = new StringTokenizer(includes, ", ", false);
        while (tok.hasMoreTokens()) {
            createInclude().setName(tok.nextToken());
        }
    }
}

/**
 * Appends <code>excludes</code> to the current list of exclude patterr
 * Patterns may be separated by a comma or a space.
 */
public void setExcludes(String excludes) {
    if (isReference()) {
        throw tooManyAttributes();
    }
    if (excludes != null && excludes.length() > 0) {
        StringTokenizer tok = new StringTokenizer(excludes, ", ", false);
        while (tok.hasMoreTokens()) {
            createExclude().setName(tok.nextToken());
        }
    }
}

/**
 * add a name entry to the given list
 */
private NameEntry addPatternToList(List<NameEntry> list) {
    NameEntry result = new NameEntry();
    list.add(result);
    return result;
}

/**
```

Figure 4.5: The screenshot of Grafter

for grafting before running differential testing. GRAFTER visualizes the results of differential testing at the level of test outcomes and program states. GRAFTER is built on top of an open-source program differencing tool, JMeld.[3] To run GRAFTER, a user needs to first specify the source and test folders in a project as well as the build commands in a configuration file. GRAFTER is run as a stand-alone Java desktop application.

```
java -jar grafter.jar /path/to/your/config/file
```

Step 1: Load Clones to Grafter

To ease the effort of detecting clones in a large code base, GRAFTER integrates two widely used off-the-shelf clone detectors, Deckard and Simian (① in Figure 4.5). Deckard parses source code to abstract syntax trees (ASTs) and detects similar code via tree comparison [109], while Simian is a commercial tool that detects duplicated code via text comparison [2]. A user can also specify the clones of interest in an `XML` file and load these clones manually by clicking Load.

[3]https://github.com/albfan/jmeld

As noted by previous studies [44, 201], clone detection tools may emit a large number of uninteresting clones—clones that are textually similar but are trivial to investigate. Grafter proactively detects such trivial clones using several heuristics, so that a user can easily ignore these clones if: (1) the clones are in comments or declaration statements such as a sequence of field declarations, (2) clones are not syntactically complete, or (3) clones are from the same method. Grafter also marks clones in test files, since Grafter aims to contrast the runtime behavior of clones in functional code.

Step 2: Inspect Clones Side-by-Side

A user can inspect the textual differences of loaded clones by right-clicking a clone group and selecting Compare (② in Figure 4.5). We customized JMeld to compute the differences between only the clone regions in two Java files. So a user only needs to focus on the clone regions, instead of the entire file. Each clone in the group is named in the format of "*(start line number, end line number)*" by default. A user can update the clone region or rename it with the method name by double-clicking its label.

During the manual inspection, if a user considers a clone group to be trivial, she can filter out the clone group by right-clicking the group label and selecting Exclude. Similarly, if a user finds a clone group is mistakenly excluded, she can add the group back by right-clicking the group label and selecting Include.

Step 3: Find Test Cases of Clones

In a large project with many clones and test files, a user may find it difficult to manually locate the corresponding test cases for each detected clone. Grafter facilitates test coverage analysis by automatically locating relevant test cases for each clone. Grafter instruments each clone to log its call stack trace and gathers the test cases that appear in the trace. When a user clicks the Coverage button (③ in Figure 4.5), Grafter automatically detects and updates the test cases that exercise each loaded clone (⑦ in Figure 4.5). Note that Grafter requires at least one clone in a clone group to be executed by one or more tests in order to re-run the same test(s) on its counterpart clones. If none of the clones in a group is examined by a test case, Grafter is not able to examine their behavior.

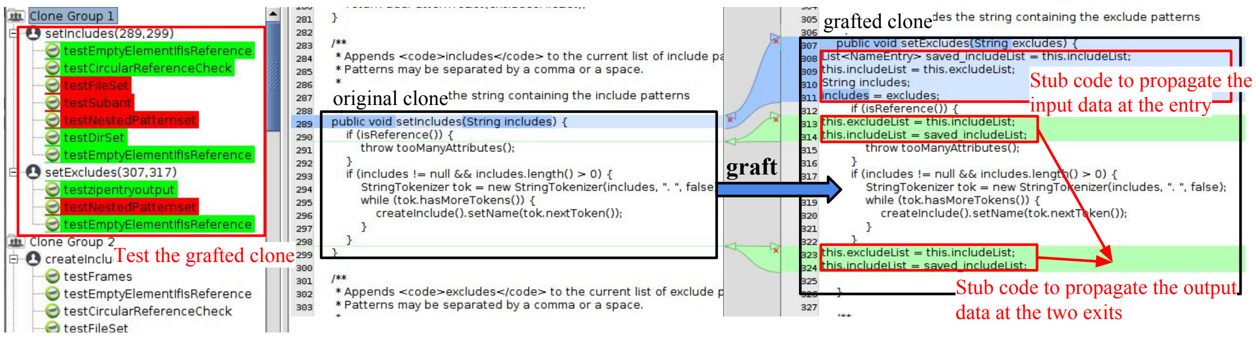
Step 4: Experiment with Clone Transplantation

Note that the stub code inserted during grafting is not a permanent edit and does not need to be comprehended by a developer, since it is temporarily inserted to run the same tests for differential testing purposes. Nevertheless, Grafter's GUI allows a user to experiment with clone grafting and to examine the grafted code. For example, a user can graft the left clone to replace the right one and view the grafted code in the *diff* view by clicking Graft Left in the top menu bar (④ in Figure 4.5). A user can further test the grafted code by right-clicking the clone group and selecting Test. The passed test cases will be highlighted in green, while the failed test cases will be highlighted in red, as shown in Figure 4.6. A user can undo and redo the previous grafting by clicking the Undo and Redo buttons on the top menu bar respectively.

Step 5: Examine Runtime Behavioral Differences

A user can contrast and examine the behavior of clones by right-clicking a clone group and selecting Compare Test Behavior or Compare State Behavior (⑤ in Figure 4.5). Grafter handles one pair of clones at a time. If there are more than two clones in a group, Grafter will compare every pair of clones in the group. The test-level comparison contrasts the test outcomes, as shown in Figure 4.7. The first column shows the names of test cases. The second and third columns show whether each clone passes or fails the corresponding test case. The last column shows the comparison result. Green means both clones are consistent in terms of test outcomes and red means their test outcomes are different.

The state-level comparison contrasts the internal state values of corresponding variables in two clones, as shown in Figure 4.8. The first and third columns show the names of corresponding variables, and the second and fourth columns show the variable values in the format of XML. Grafter prints the value of an object in XML using *XStream*. A user


Test the grafted clone
original clone
graft
grafted clone
Stub code to propagate the input data at the entry
Stub code to propagate the output data at the two exits

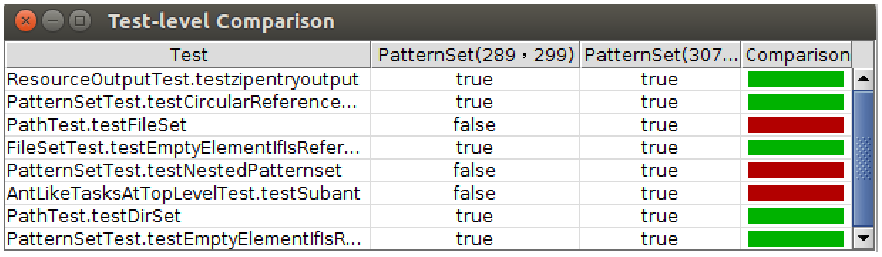
```
Clone Group 1
  setIncludes(289,299)
    testEmptyElementIfIsReference
    testCircularReferenceCheck
    testFileSet
    testSubant
    testNestedPatternset
    testDirSet
    testEmptyElementIfIsReference
  setExcludes(307,317)
    testzipentryoutput
    testNestedPatternset
    testEmptyElementIfIsReference
Clone Group 2
  createInclu
    testFrames
    testEmptyElementIfIsReference
    testCircularReferenceCheck
    testFileSet
```

```
}

/**
 * Appends <code>includes</code> to the current list of include pa
 * Patterns may be separated by a comma or a space.
 *
            the string containing the include patterns

public void setIncludes(String includes) {
    if (isReference()) {
        throw tooManyAttributes();
    }
    if (includes != null && includes.length() > 0) {
        StringTokenizer tok = new StringTokenizer(includes, ". ", false
        while (tok.hasMoreTokens()) {
            createInclude().setName(tok.nextToken());
        }
    }
}

/**
 * Appends <code>excludes</code> to the current list of exclude p
 * Patterns may be separated by a comma or a space.
```

```
            des the string containing the exclude patterns

    public void setExcludes(String excludes) {
List<NameEntry> saved_includeList = this.includeList;
this.includeList = this.excludeList;
String includes;
includes = excludes;
        if (isReference()) {
this.excludeList = this.includeList;
this.includeList = saved_includeList;
            throw tooManyAttributes();
        }
        if (includes != null && includes.length() > 0) {
            StringTokenizer tok = new StringTokenizer(includes, ". ", false);
            while (tok.hasMoreTokens()) {
                createInclude().setName(tok.nextToken());
            }
        }
this.excludeList = this.includeList;
this.includeList = saved_includeList;

    }
```

Figure 4.6: Experimenting the clone transplantation. A user can view and test the grafted clone. Green means the test succeeds. Red means the test fails.

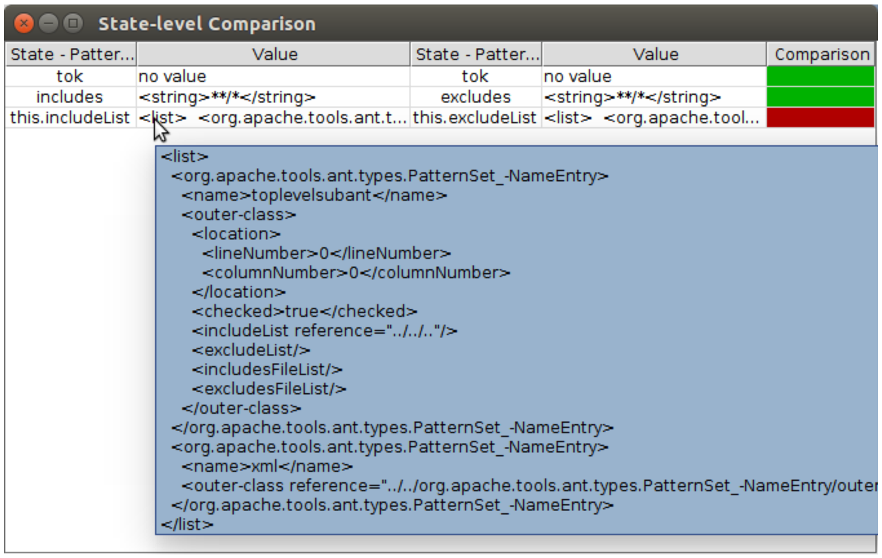
can view the complete XML representation of an object by hovering the mouse over the corresponding cell.



Test-level Comparison

Test	PatternSet(289 , 299)	PatternSet(307...	Comparison
ResourceOutputTest.testzipentryoutput	true	true	🟩
PatternSetTest.testCircularReference...	true	true	🟩
PathTest.testFileSet	false	true	🟥
FileSetTest.testEmptyElementIfIsRefer...	true	true	🟩
PatternSetTest.testNestedPatternset	false	true	🟥
AntLikeTasksAtTopLevelTest.testSubant	false	true	🟥
PathTest.testDirSet	true	true	🟩
PatternSetTest.testEmptyElementIfIsR...	true	true	🟩

Figure 4.7: The test outcome level behavior comparison for clones



State-level Comparison

State - Patter...	Value	State - Patter...	Value	Comparison
tok	no value	tok	no value	🟩
includes	<string>**/*</string>	excludes	<string>**/*</string>	🟩
this.includeList	<list> <org.apache.tools.ant.t...	this.excludeList	<list> <org.apache.tool...	🟥

```
<list>
  <org.apache.tools.ant.types.PatternSet_-NameEntry>
    <name>toplevelsubant</name>
    <outer-class>
      <location>
        <lineNumber>0</lineNumber>
        <columnNumber>0</columnNumber>
      </location>
      <checked>true</checked>
      <includeList reference="../../.."/>
      <excludeList/>
      <includesFileList/>
      <excludesFileList/>
    </outer-class>
  </org.apache.tools.ant.types.PatternSet_-NameEntry>
  <org.apache.tools.ant.types.PatternSet_-NameEntry>
    <name>xml</name>
    <outer-class reference="../../org.apache.tools.ant.types.PatternSet_-NameEntry/outer
  </org.apache.tools.ant.types.PatternSet_-NameEntry>
</list>
```

Figure 4.8: The state-level behavior comparison for clones. A user can hover the mouse to view the complete XML representation of an object.

4.5 Evaluation

Our evaluation investigates three research questions.

- RQ1: How successful is GRAFTER in transplanting code?
- RQ2: How does GRAFTER compare with a static cloning bug finder in terms of detecting behavioral differences?
- RQ3: How robust is GRAFTER in detecting unexpected behavioral differences caused by program faults?

We use DECKARD [109] to find intra-method clones (i.e., clone fragments appearing in the middle of a method) from three open-source projects. Apache Ant is a software build framework. Java-APNS is a Java client for the apple push notification service. Apache XML Security is a XML signature and encryption library. Ant and XML Security are well-known large projects with regression test suites. In Table 4.1, LOC shows the size in terms of lines of code. Test# shows the number of JUnit test cases. Branch Coverage and Statement Coverage show branch and statement coverage respectively, both measured by JaCoCo.[4] Pair# shows the number of selected clone pairs.

Because GRAFTER's goal is to reuse tests for nonidentical clones, we include clones meeting the following criteria in our dataset: (1) each clone pair must have at least one clone exercised by some tests, (2) clones must not be identical, because grafting is trivial for identical clones, (3) each clone must have more than one line, (4) each clone must not appear in uninteresting areas such as import statements and comments. These are not restrictions on GRAFTER's applicability, rather we target cases where GRAFTER is designed to help (e.g., tests exist for reuse) and where grafting clone is hard (e.g., clones with variations and without well-defined interfaces).

Table 4.1: Subject Programs

Subject	LOC	Test#	Branch Coverage	Satement Coverage	Clone Pair#
ant-1.9.6	267,048	1,864	45%	50%	18
Java-APNS-1.0.0	8,362	103	59%	67%	7
xmlsec-2.0.5	121,594	396	59%	65%	27

4.5.1 Grafting Capability

We use GRAFTER to graft clones in each pair in both directions. A pair of clones is considered successfully grafted if and only if there is no compilation error in both directions. Table 4.2 shows 52 clone pairs in our dataset. Column Type shows types of code clones in our dataset based on a well-known clone taxonomy [59, 201]. Type I clones refer to identical code fragments. Type II clones refer to syntactically identical fragments except for variations in names, types, method call targets, constants, white spaces, and comments. Type III clones refer to copied code with added and deleted statements. Because variations in variable names, types, method calls, and constants are all grouped as Type II clones, we enumerate individual kinds of variations in column Variation. Since grafting identical code is trivial, our evaluation focuses on Type II and III clones not to artificially inflate our results.

Column Tested indicates whether both clones are covered by an existing test suite (i.e., *full*) or only one of the two is covered (i.e., *partial*). Success shows whether GRAFTER successfully grafts clones without inducing compilation errors. Δ shows lines of stub code inserted by GRAFTER and Prpg shows the kinds of heuristics applied for data propagation, described in Section 4.3.3. Branch and Stmt show the branch and statement coverage over the cloned regions in each pair before and after test reuse respectively.

GRAFTER successfully grafts code in 49 out of 52 pairs (94%). In 3 cases, GRAFTER rejects transplantation, because it does not transform objects, unless they are hierarchically related or structurally equivalent to ensure type safety. In Table 4.2, the corresponding rows are marked with ✗ and —. On average, 6 lines of stub code is inserted. Transplantation Rule#1 is needed to define used but undefined variables in grafted code to handle variable

[4]http://www.eclemma.org/jacoco/

Table 4.2: Evaluation Benchmark

ID	Type	Variation	Tested	Graft			Branch		Stmt		Behavior Comparison			Mutation		
				Success	Δ	Prpg	Before	After	Before	After	Test	State	Jiang	Test	State	Jiang
1	II	var, call	full	✓	9	B	75%	75%	100%	100%	0/8	0/4	✗	10/28	12/28	12/28
2	II	var	full	✓	8	B	100%	100%	100%	100%	0/10	0/2	✗	4/4	4/4	4/4
3	II	var	full	✓	8	B	50%	50%	75%	75%	0/1	0/2	✗	2/4	2/4	4/4
4	II	var, call	full	✓	9	B	50%	50%	75%	75%	0/1	0/3	✗	2/6	2/6	6/6
5	II	var, call	full	✓	10	A, B	100%	100%	100%	100%	0/13	2/7	✗	4/8	4/8	8/8
6	II	var	full	✓	6	B	100%	100%	100%	100%	0/38	2/3	✗	12/12	12/12	0/12
7	II	var	full	✓	23	B	100%	100%	100%	100%	0/38	5/5	✗	20/22	20/22	6/22
8	II	var	full	✓	8	B	50%	50%	88%	88%	0/36	2/2	✗	2/6	6/6	4/6
9	II	type, call	full	✓	16	A	33%	66%	50%	80%	2/3	4/7	✗	—	—	—
10	II	var	full	✓	3	A, B	88%	100%	93%	100%	0/37	8/9	✗	13/30	30/30	12/30
11	II	var	full	✓	7	B	100%	100%	100%	100%	0/14	3/5	✗	4/4	4/4	4/4
12	II	call	full	✓	0	B	50%	50%	75%	75%	60/157	3/4	✗	—	—	—
13	II	var, type, call, lit	full	✓	6	A, C	63%	63%	100%	100%	1/1	10/11	✗	—	—	—
14	II	var, type, call, lit	full	✓	6	A, C	63%	63%	100%	100%	1/1	10/11	✗	—	—	—
15	II	type, call	full	✗	—	—	33%	—	45%	—	—	—	—	—	—	—
16	II	var, type, call	full	✓	4	A, B	100%	100%	100%	100%	15/45	2/7	✗	—	—	—
17	II	var, type	full	✓	3	B	25%	25%	14%	14%	54/54	4/5	✓	—	—	—
18	II	var	full	✓	6	B	75%	100%	75%	100%	0/116	0/3	✗	4/6	4/6	0/6
19	II	lit	full	✓	0	A	25%	25%	17%	17%	0/1	1/4	✗	2/6	2/6	6/6
20	II	type, lit	full	✓	0	A	66%	66%	80%	80%	4/4	2/2	✗	—	—	—
21	II	lit	full	✓	0	A	75%	75%	83%	83%	0/2	0/4	✗	2/6	2/6	4/6
22	II	var	full	✓	9	B	50%	50%	71%	71%	0/307	0/5	✗	2/24	4/24	6/24
23	II	call	full	✓	0	A	100%	100%	100%	100%	160/168	2/3	✗	—	—	—
24	II	type, lit	full	✓	11	A, C, D	70%	70%	77%	77%	1/1	1/4	✗	—	—	—
Type II (full)				23/24 (96%)	7		65%	68%	80%	84%	9/23 (39%)	16/23 (70%)	1/23 (4%)	83/166 (50%)	108/166 (65%)	76/166 (46%)
25	II	var	partial	✓	6	B	50%	100%	50%	100%	0/1	1/29	✗	2/4	4/4	4/4
26	II	var	partial	✓	6	B	50%	100%	50%	100%	0/1	1/29	✗	2/4	3/4	4/4
27	II	var	partial	✓	6	B	50%	100%	50%	100%	0/1	1/29	✗	3/4	4/4	0/4
28	II	lit	partial	✓	0	A	25%	50%	42%	84%	0/1	2/2	✗	4/12	12/12	4/12
29	II	var	partial	✓	6	B	50%	100%	50%	100%	0/4	1/3	✗	2/2	2/2	0/2
30	II	var, type	partial	✓	6	B	50%	100%	50%	100%	0/4	1/3	✓	2/2	2/2	1/2
31	II	var, type	partial	✓	6	B	50%	100%	50%	100%	0/4	1/3	✓	2/2	2/2	1/2
32	II	var	partial	✓	3	A, B	50%	100%	50%	100%	0/5	0/1	✗	2/2	2/2	2/2
33	II	var, lit	partial	✓	3	B	25%	50%	50%	100%	0/1	1/2	✗	0/8	2/8	2/8
34	II	type, call	partial	✓	0	A	25%	50%	50%	100%	0/21	5/7	✗	6/20	7/20	4/20
35	II	var, type, call	partial	✗	—	—	50%	—	50%	—	—	—	—	—	—	—
36	II	var, type, call, lit	partial	✗	—	—	25%	—	50%	—	—	—	—	—	—	—
37	II	var, lit, call	partial	✓	4	B	25%	50%	50%	100%	2/2	1/2	✗	—	—	—
38	II	var, lit	partial	✓	3	B	25%	50%	50%	100%	0/1	1/2	✗	1/8	6/8	2/8
Type II (partial)				12/14 (86%)	4		34%	68%	49%	98%	1/12 (8%)	11/12 (92%)	2/12 (17%)	26/68 (38%)	46/68 (68%)	24/68 (36%)
Type II Total				35/38 (92%)	6		59%	68%	73%	87%	10/35 (29%)	27/35 (77%)	3/35 (9%)	109/234 (47%)	154/234 (66%)	100/234 (43%)
39	III	call, extra	full	✓	2	A	100%	100%	100%	100%	10/21	4/6	✓	22/26	22/26	10/26
40	III	call, lit, extra	full	✓	33	A	38%	38%	68%	68%	1/3	2/8	✓	—	—	—
41	III	type, extra	full	✓	0	A	70%	70%	100%	100%	0/4	6/11	✓	18/30	23/30	8/30
42	III	var, extra	full	✓	0	A	51%	68%	70%	81%	33/156	8/13	✓	—	—	—
Type III (full)				4/4 (100%)	9		54%	65%	77%	84%	3/4 (75%)	4/4 (100%)	4/4 (100%)	40/56 (71%)	45/56 (80%)	18/56 (32%)
43	III	var, call, extra	partial	✓	32	B	36%	64%	50%	88%	0/3	1/5	✓	15/29	29/29	7/29
44	III	call, extra	partial	✓	8	A	33%	66%	43%	86%	2/2	3/8	✓	—	—	—
45	III	var, extra	partial	✓	10	B	20%	40%	20%	40%	14/14	2/3	✓	—	—	—
46	III	var, extra	partial	✓	6	B	25%	50%	25%	50%	14/14	2/3	✓	—	—	—
47	III	call, extra	partial	✓	1	A	25%	50%	46%	100%	0/2	4/5	✓	3/38	3/38	0/38
48	III	extra	partial	✓	1	A	25%	50%	30%	70%	0/4	1/4	✓	0/4	0/4	2/4
49	III	var, lit, extra	partial	✓	4	A, B	17%	50%	30%	70%	4/4	4/4	✗	—	—	—
50	III	var, lit, extra	partial	✓	4	A, B	17%	50%	30%	70%	4/4	5/5	✓	—	—	—
51	III	var, lit, extra	partial	✓	1	A	17%	50%	30%	70%	4/4	5/5	✓	—	—	—
52	III	lit, extra	partial	✓	3	A	17%	50%	30%	70%	1/1	3/3	✓	—	—	—
Type III (partial)				10/10 (100%)	7		27%	49%	36%	68%	7/10 (70%)	10/10 (100%)	9/10 (90%)	18/71 (25%)	32/71 (45%)	9/71 (13%)
Type III Total				14/14 (100%)	8		39%	60%	56%	81%	10/14 (71%)	14/14 (100%)	13/14 (93%)	58/127 (46%)	77/127 (61%)	27/127 (21%)
Type II & III Total				49/52 (94%)	6		50%	72%	67%	83%	18/47 (38%)	39/47 (83%)	14/47 (29%)	167/361 (46%)	231/361 (64%)	127/361 (35%)

name variations in 38 cases. Furthermore, Heuristic A is applied in these cases to ensure that the input data is properly propagated from a matched variable in the target location to the newly defined variable. In 3 of these 38 cases, Transplantation Rule#3 is successfully applied to convert objects with different types to ensure type safety, since matched variables have different types between clones. Heuristic C is further applied in these cases to propagate data between corresponding subfields among these types. In 3 cases, Transplantation Rule#2 is needed to port method definitions to resolve compilation errors due to method call variations between clones. In 11 cases, Transplantation Rule#4 is needed to introduce temporary variables to handle expression type variations between code clones. In 2 cases, Transplantation Rule#5 is applied to handle recursive calls. Heuristic D is applied to synthesizes a loop to propagate data between two arrays of structurally equivalent objects in Pair#24. Even when the mapped variables have the same type and name, additional stub code may be required, when one clone references extra identifiers undefined in another context (e.g., Pairs#47, 48, 52). In some cases, the generated stub code is over 30 lines long, indicating that naïve cut and paste is definitely inadequate for ensuring type safety and data transfer. Grafter automates this complex stub code generation.

By reusing tests between clones, Grafter roughly doubles the statement coverage and the branch coverage of partially tested clone pairs. For partially tested Type II clone pairs, statement coverage improves from 49% to 98% and branch coverage improves from 34% to 68%. For partially tested Type III clone pairs, we observe similar improvement (36% to 68% and 27% to 49%). For fully tested clones pairs, statement coverage is still improved by augmenting tests.

4.5.2 Behavior Comparison Capability

We use Grafter to detect behavioral differences on the 49 pairs of successfully grafted clones. We hypothesize that by noticing fine-grained behavior differences at runtime, Grafter can detect potential cloning bugs more effectively than Jiang et al. [111] that detect three predefined bug types:

```
if(l_stride!=NULL){
  mps_cdiv_q(X1,X1, l_stride ->value);
}
```

```
if(l_stride!=NULL){
  mps_cdiv_q(X1,X1, r_stride ->value);
}
```

(a) Renaming mistake.

```
if(cmd_type==READ_M2){
  msgbuf[xa_count*3]=0;
  msg(DBG_XA1, ...);
}
```

```
for(i=0;i<count;i++){
  msgbuf[i*3]=0;
  msg(DBG_SQ1, ...);
}
```

(b) Control-flow construct inconsistency.

```
if(length>=9&& strcmp (
    buffer,"EESOXSCSI",9){
  buffer+=9;
  length+=9;
}
```

```
if(length>=11&& strncmp (
   buffer,"CUMANNASCSI2"){
  buffer+=11;
  length+=11;
}
```

(c) Conditional predicate inconsistency.

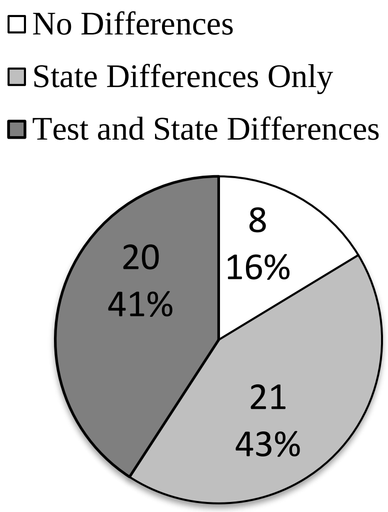
Figure 4.9: Three examples of cloning bugs by Jiang et al.

- Rename Mistake: in Figure 4.9a, the right clone performs a null check on `l_stride` but then dereferences `r_stride` due to a renaming mistake.
- Control-flow Construct Inconsistency: in Figure 4.9b, the left clone is enclosed in an `if` statement, while the right clone is enclosed in a `for` loop.
- Conditional Predicate Inconsistency: in Figure 4.9c, though both clones are in `if` branches, the `if` predicates are different: one calls `strncmp` which takes three arguments, while the other calls `strcmp` which takes two arguments.

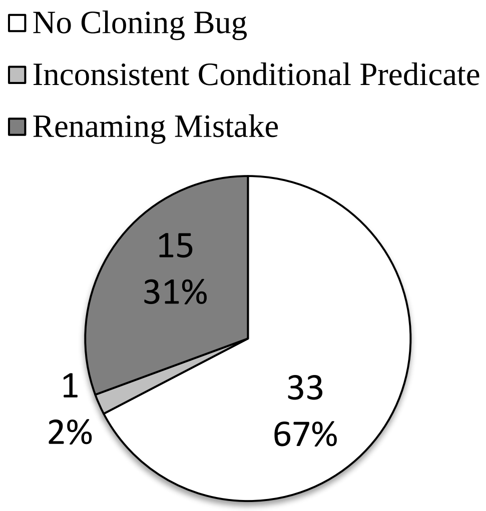
In Table 4.2, Behavior Comparison shows whether GRAFTER detects behavioral differences in each clone pair. Test shows how many tests exhibit different test outcomes. State shows how many variables exhibit state differences using GRAFTER's state-level comparison. Jiang shows whether Jiang et al. detect cloning bugs ✓or not ✗.

GRAFTER detects test-level differences in 20 pairs of clones and detects state-level differences in 41 pairs. On the other hand, Jiang et al. detect differences only in 16 pairs because they ignore behavioral differences at runtime caused by using different types and calling different methods. For example, pair#9 from Apache Ant in Figure 4.11 uses different object types, `TarFileSet` and `ZipFileSet`. Given the same input variables `p` and `o`, the clones enter different branches due to different runtime type checks (i.e., `instanceof` predicates). Because these runtime checks are syntactically isomorphic and there are no renaming mistake, Jiang et al. report no inconsistency. 13 of 15 renaming mistakes detected by Jiang et al. are in Type III clones, because Jiang et al. compare unique identifiers in each clone to detect renaming mistakes and added statements often lead to extra variable counts. In other words, by definition, they consider almost all Type III clones as cloning bugs.

Figure 4.10a shows that state-level difference is noted in 84% of pairs, while test outcome difference is noted in 41% of pairs. State-level comparison being more sensitive than test-level comparison is expected, because some program states are not examined by test oracle checking. As GRAFTER focuses its comparison scope to only affected variables, the size of state-level comparison is manageable, three variables on average.



(a) GRAFTER



(b) Jiang et al.

Figure 4.10: Comparison between GRAFTER and Jiang et al.

```java
File: /org/apache/tools/ant/types/TarFileSet.java
protected AbstractFileSet getRef(Project p){
  dieOnCircularReference();
  Object o = getRefid().getReferencedObject(p);
  if(o instanceof TarFileSet){
    return (AbstractFileSet)o;
  }else if (o instanceof FileSet){
    TarFileSet zfs = new TarFileSet((FileSet)o);
    configureFileSet(zfs);
    return zfs;
  }else{
    throw new Exception(..);
  }
}
```

```java
File: /org/apache/tools/ant/types/ZipFileSet.java
protected AbstractFileSet getRef(Project p){
  dieOnCircularReference();
  Object o = getRefid().getReferencedObject(p);
  if(o instanceof ZipFileSet){
    return (AbstractFileSet)o;
  }else if (o instanceof FileSet){
    ZipFileSet zfs = new ZipFileSet((FileSet)o);
    configureFileSet(zfs);
    return zfs;
  }else{
    throw new Exception(..);
  }
}
```

Figure 4.11: Type II clones (Pair#9) where GRAFTER detects behavioral differences and Jiang et al do not.

4.5.3 Fault Detection Robustness

To systematically assess the robustness of GRAFTER in detecting unexpected behavioral differences caused by program faults, we use the MAJOR mutation framework to inject 361 mutants into 30 pairs of clones. The 19 pairs where the test-level comparison already exhibits differences without adding mutants are marked with — and excluded from the study in order not to over-inflate our results. Each mutant represents an artificial cloning bug and it is injected to only one clone in each pair. We then use GRAFTER to check whether behavioral difference is exhibited at runtime. Table 4.3 shows eight kinds of mutants injected by MAJOR. A mutant is detected by GRAFTER's test-level comparison, if GRAFTER exposes test outcome differences in one or more tests after injecting the mutant. A mutant is detected by GRAFTER's state-level comparison, if there is an affected variable with a state value different from its corresponding variable's state value.

In Table 4.2, columns in Mutation show the mutation experiment results. Test shows how many mutants are detected using GRAFTER's test-level comparison. For example, 10/28 indicates that 10 out of 28 mutants are detected using GRAFTER's test-level comparison. Similarly, State shows how many mutants are detected using GRAFTER's state-level comparison while Jiang shows how many mutants are detected by Jiang et al.

Overall, GRAFTER detects 167 mutants (46%) using the test-level comparison and 231 mutants (64%) using the state-level comparison. This finding that the state-level comparison

Table 4.3: 8 kinds of mutants injected by Major

Operator	Description	Example
AOR	Arithmetic operator replacement	$a + b \rightarrow a - b$
LOR	Logical operator replacement	$a \wedge b \rightarrow a \vert b$
COR	Conditional operator replacement	$a \vee b \rightarrow a\&\&b$
ROR	Relational operator replacement	$a == b \rightarrow a> = b$
SOR	Shift operator replacement	$a>>b \rightarrow a<<b$
ORU	Operator replacement unary	$\neg a \rightarrow \sim a$
STD	Statement deletion operator: delete (omit) a single statement	foo(a, b) $\rightarrow$ // foo (a, b)
LVR	Literal value replacement: replace by a positive value, a negative value or zero	0 $\rightarrow$ 1 0 $\rightarrow$ -1

is more sensitive to seeded mutants than the test-level comparison is consistent with the literature of strong and weak mutation testing [105,117,254]. Jiang et al. detect 127 mutants (35%) only, as shown in Figure 4.12. Grafter outperforms Jiang et al. by detecting 31% more mutants at the test level and almost twice more at the state level. Its mutant detection ability is similar for both Type II and III clones.

Figure 4.13 shows Grafter is less biased than Jiang et al. when detecting different kinds of mutants. Jiang et al. detect 60% of COR mutants and 44% of STD mutants but less than 20% in other mutant types. This is because Jiang et al. only detect three pre-defined types of cloning bugs—removed statements (STD mutants) often flag renaming mistakes, and COR mutants flag inconsistent conditional predicate updates. Because many removed statements do not affect program states examined by test oracles, Grafter's test-level comparison detects fewer STD mutants than Jiang et al. Grafter does not detect mutants in eight AOR mutants, because they are all injected in an untested branch in pair#22. Jiang et al. do not detect these AOR mutants because they ignore inconsistencies in arithmetic operators. In summary, our experiment shows that Grafter can complement a static cloning bug finder via test reuse and differential testing.

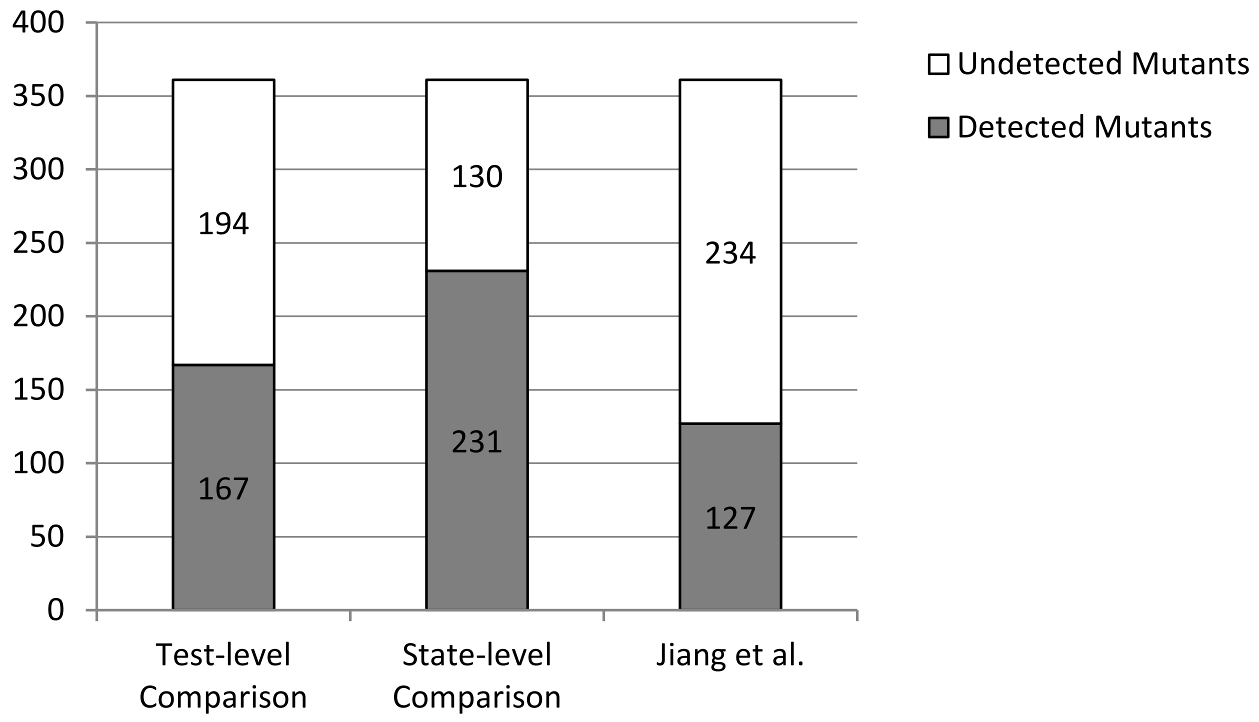


Figure 4.12: Mutant detection

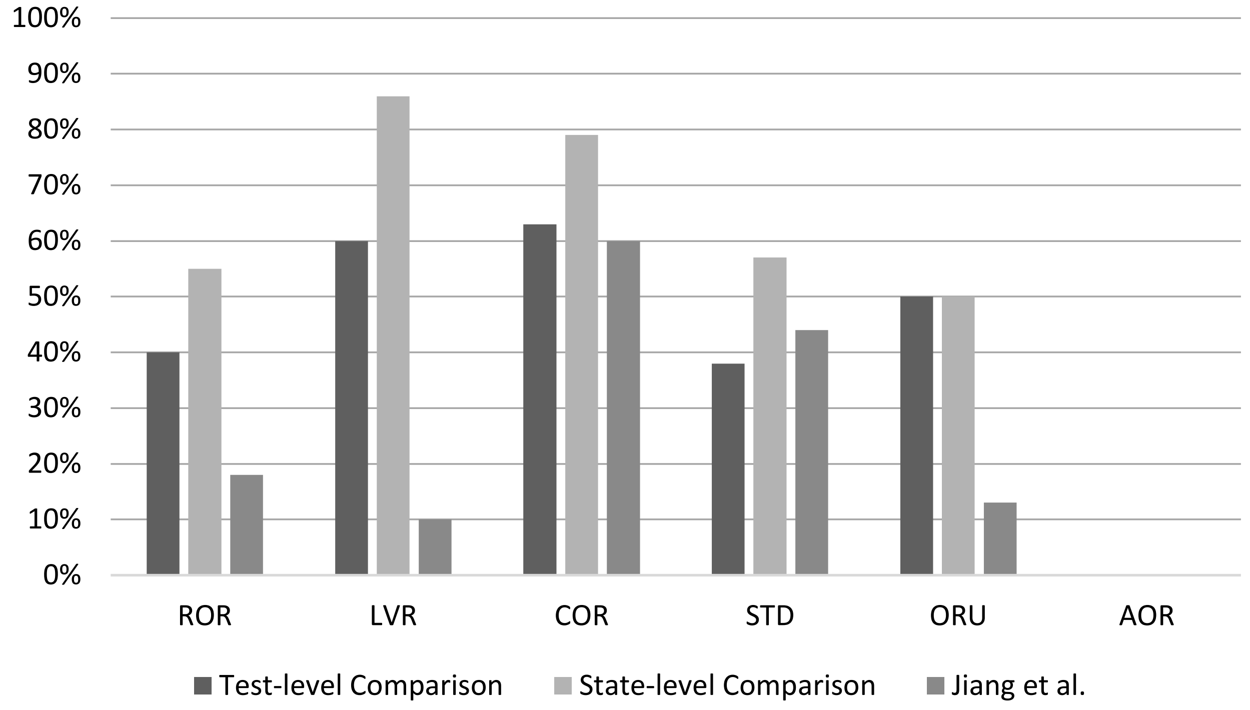


Figure 4.13: Mutant killing ratio for different mutant kinds

4.6 Threats to Validity

In terms of *external validity*, since clones in our study are found using an AST-based clone detector [109], our dataset does not include Type IV clones—functionally similar code without any syntactic resemblance. For Type IV clones, a programmer may need to provide the correspondence between variables to enable differential testing. In Section 4.5.3, we use mutants as a replacement for real faults to assess the robustness of Grafter. Recent studies [30, 119] find a strong correlation between mutants and real faults, so our results should generalize to real faults.

In terms of *internal validity*, like other dynamic approaches, Grafter's capability to expose behavioral differences is affected by test coverage and quality. If an existing test covers only some branches within a clone, Grafter may not expose behavioral differences in uncovered code. However, Grafter is still useful for boosting test coverage through code transplantation. Grafter's transplantation is guided by the syntactic resemblance of input and output variables. Grafter matches variables based on name and type similarity. This heuristic works well for real-world clones in our evaluation, which can be attributed to the fact that these clones are intra-project clones and developers in the same project may follow similar naming conventions. However, manual adjustments may be needed when variables have significantly different names. Experimentation with cross-project clones and alternative matching heuristics [54, 159] remain as future work.

In terms of *construct validity*, Grafter conservatively chooses not to graft clones referencing unrelated types—not castable nor structurally equivalent. This limit can be overcome by allowing programmers to provide user-defined type transformation functions. Grafter grafts clones rather than tests. Transplanting tests could have the advantage of minimizing impact on the functionality under test. Extending Grafter to transplant tests remains as future work. In Section 4.5.2, we do not assume that all clones should behave similarly at runtime nor we argue that all behavioral differences indicate cloning bugs. Rather, Grafter helps detect behavioral differences concretely and automatically. Therefore, it is necessary for the authors of clones to confirm whether detected behavioral differences are intended or represent potential bugs. Assessing if the generated tests are valuable to the authors remains as future work.

4.7 Summary

This chapter presents the first approach called Grafter that enables developers to reuse the same test between similar programs and examine their behavioral similarity and differences. Existing test reuse and differential testing techniques are much restricted to comparable functions with clear input-output interfaces. By contrast, Grafter is capable of handling arbitrary code fragments by exposing their de-facto interfaces via def-use analysis and handling their variations using a set of code transformation and data propagation rules. The evaluation on 52 non-identical clones shows that Grafter achieves 94% success rate in test reuse and transplantation, and helps developers identify up to 2X more seeded bugs than an existing static approach.

Both Critics and Grafter leverage similarities and differences among code clones in local codebases, but do not harness similar programs in the large body of open-source projects available on the Internet. The next three chapters will present three techniques that facilitate software development by mining, analyzing, and visualizing API usage patterns and code adaptation patterns learned from hundreds of thousands of GitHub projects.

CHAPTER 5

Mining Common API Usage Patterns from Massive Code Corpora

The availability of the large and growing body of open-source projects suggests a new, data-driven approach for software development: why not let the statistical properties of programs estimated over massive code corpora influence the development of software? In this chapter, we focus on API usage correctness in software products, since a common task in modern software development is to learn how to correctly and effectively use existing APIs. In partcilar, we present the first scalable approach that harnesses the power of such *Big Code* by mining common API usage patterns from 380K GitHub projects.

5.1 Introduction

Library APIs are becoming the fundamental building blocks in modern software development. Programmers reuse existing functionalities in well-tested libraries and frameworks by stitching API calls together, rather than building everything from scratch. Online Q&A forums such as Stack Overflow have a large number of curated code examples [202, 238]. Though such curated examples can serve as a good starting point, they could potentially impact the quality of production code, when integrated to a target application verbatim. Recently, Fischer et al. find that 29% of security-related snippets in Stack Overflow are insecure and these snippets could have been reused by over 1 million Android apps on Google play, which raises a big security concern [73]. Previous studies have also investigated the quality of online code examples in terms of compilability [223, 259], unchecked obsolete usage [268], and comprehension issues [235]. However, none of these studies have investigated the reliability of online code examples in terms of API usage correctness. There is also no tool support to help developers easily recognize unreliable code examples in online Q&A forums.

This chapter aims to assess the reliability of code examples on Stack Overflow by contrasting them against desirable API usage patterns mined from GitHub. Our insight is that commonly recurring API usage from a large code corpus may represent a desirable pattern that a programmer can use to assess or enhance code examples on Stack Overflow. The corpus should be large enough to provide sufficient API usage examples and to mine representative API usage patterns. We also believe that quantifying how many snippets are similar (or related but not similar) to a given example can improve developers' confidence about whether to trust the example as is.

Therefore, we design an API usage mining technology, ExampleCheck that scales to over 380K GitHub repositories without sacrificing the *fidelity* and *expressiveness* of the underlying API usage representation. By leveraging an ultra-large-scale software mining infrastructure [66, 239], ExampleCheck efficiently searches over GitHub and retrieves an average of 55144 code snippets for a given API within 10 minutes. We perform program slicing to remove statements that are not related to the given API, which improves accuracy in the mining process (Section 5.4). We combine frequent subsequence mining and SMT-based guard condition mining to retain important API usage features, including the temporal ordering of related API calls, enclosing control structures, and guard conditions that protect an API call. In terms of our study scope, we target 100 Java and Android APIs that are frequently discussed on Stack Overflow. We then inspect all patterns learned by ExampleCheck, create a data set of 180 desirable API usage patterns for the 100 APIs, and study the extent of API misuse in Stack Overflow.

Out of 217,818 SO posts relevant to our API data set, 31% contain potential API misuse that could produce symptoms such as program crashes, resource leaks, and incomplete actions. Such API misuse is caused by three main reasons—*missing control constructs*, *missing or incorrect order of API calls*, and *incorrect guard conditions*. Database, crypto, and networking APIs are often misused, since they often require observing the ordering between multiple calls and complex exception handling logic. Though programmers often put more trust on highly voted posts in Stack Overflow, we do not observe a strong positive nor negative correlation between the number of votes and the reliability of Stack Overflow posts in terms of API usage correctness. This observation suggests that votes alone should not be used as the single indicator of the quality of Stack Overflow posts. Our study provides empirical evidence about the prevalence and severity of API misuse in online Q&A posts and indicates that Stack Overflow needs another mechanism that helps users to understand the limitation of existing curated examples. We propose a Chrome extension that suggests desirable or alternative API usage for a given Stack Overflow code example, along with supporting concrete examples mined from GitHub.

The rest of this chapter is organized as follows. Section 5.2 motives our work with a code reuse scenario on Stack Overflow. Section 5.3 presents an automated pattern inference approach to facilitate API misuse detection for arbitrary APIs by mining usage patterns from massive software corpora. Section 5.5 describes the empirical study of API misuses on Stack Overflow. Section 5.7 discusses the threats to validity.

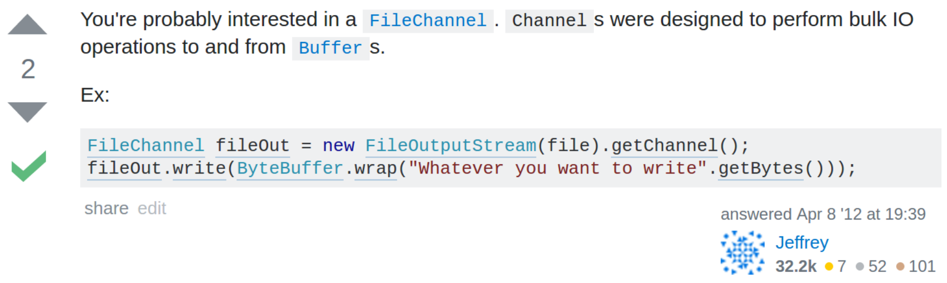
5.2 Motivating Examples

Suppose Alice wants to write data to a file using `FileChannel`. Alice searches on Stack Overflow and finds two code examples, both of which are accepted as correct answers and upvoted by other programmers, as shown in Figure 5.1. Though such curated examples can serve as a good starting point for API investigation, both examples have API usage violations that may induce unexpected behavior in real applications. If Alice puts too much trust on the given example as is, she may inadvertently follow less ideal API usage.

The first post in Figure 5.1a does not call `FileChannel.close` to close the channel. If Alice copies this example to a program that does not heavily access new file resources, this example may behave properly, because OS will clean up unmanaged file resources eventually

[1]http://stackoverflow.com/questions/10065852

[2]http://stackoverflow.com/questions/10506546



2

You're probably interested in a `FileChannel`. `Channel`s were designed to perform bulk IO operations to and from `Buffer`s.

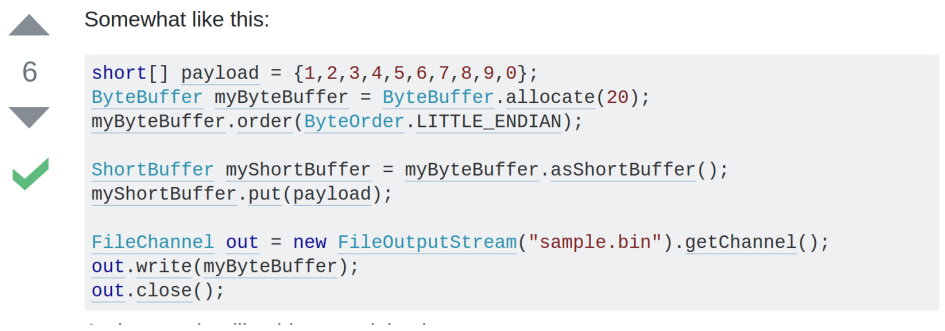
Ex:

```
FileChannel fileOut = new FileOutputStream(file).getChannel();
fileOut.write(ByteBuffer.wrap("Whatever you want to write".getBytes()));
```

share edit

answered Apr 8 '12 at 19:39
Jeffrey
32.2k ●7 ●52 ●101

(a) An example that does not close FileChannel properly[1]



6

Somewhat like this:

```
short[] payload = {1,2,3,4,5,6,7,8,9,0};
ByteBuffer myByteBuffer = ByteBuffer.allocate(20);
myByteBuffer.order(ByteOrder.LITTLE_ENDIAN);

ShortBuffer myShortBuffer = myByteBuffer.asShortBuffer();
myShortBuffer.put(payload);

FileChannel out = new FileOutputStream("sample.bin").getChannel();
out.write(myByteBuffer);
out.close();
```

(b) An example that misses exception handling[2]

Figure 5.1: Two code examples about how to write data to a file using `FileChannel` on Stack Overflow

after the program exits. However, if Alice reuses the example in a long-running program with heavy IO, such lingering file resources may cause file handle leaks. Since most operating systems limit the number of opened files, unclosed file streams can eventually run out of file handle resources [232]. Alice may also lose cached data in the file stream, if she uses `File-Channel` to write a big volume of data but forgots to flush or close the channel.

Even though the second example in Figure 5.1b calls `FileChannel.close`, it does not handle the potential exceptions thrown by `FileChannel.write`. Calling `write` could throw `ClosedChannelException`, if the channel is already closed. If Alice uses `FileChannel` in a concurrent program where multiple threads attempt to access the same channel, `Asynchronous-CloseException` will occur if one thread closes the channel, while another thread is still writing data.

As a novice programmer, Alice may not easily recognize the potential limitation of given Stack Overflow examples. In this case, our approach ExampleCheck scans over 380K GitHub repositories and finds 2230 GitHub snippets that also call `FileChannel.write`. ExampleCheck then learns two common usage patterns from these relevant GitHub snippets. The mostly frequent usage supported by 1829 code snippets on GitHub indicates that a method call to `write()` must be contained inside a `try` and `catch` block. Another frequent usage supported by 1267 GitHub snippets indicates that `write` must be followed by `close`. By comparing code snippets in Figures 5.1a and 5.1b against these two API usage patterns, Alice may consider adding a missing call to `close` and an exception handling block during the example integration and adaptation.

5.3 Scalable API Usage Mining on 380K GitHub Projects

As it is difficult to know desirable or alternative API usage a priori, we design an API usage mining approach, called ExampleCheck that scales to massive code corpora such as GitHub. We then inspect the results manually and construct a data set of desirable API usage to be used for the Stack Overflow study in Section 5.5.

Given an API method of interest, ExampleCheck takes three phases to infer API usage.

$$
\begin{aligned}
\mathit{sequence} &:= \epsilon \mid \mathit{call}\ ;\ \mathit{sequence} \\
&\quad \mid \mathit{structure}\ \{\ ;\ \mathit{sequence}\ ;\ \}\ ;\ \mathit{sequence} \\
\mathit{call} &:= \mathit{name}(t_1, ...t_n)@\mathit{condition} \\
\mathit{structure} &:= \texttt{if} \mid \texttt{else} \mid \texttt{loop} \mid \texttt{try} \mid \texttt{catch}(t) \mid \texttt{finally} \\
\mathit{condition} &:= \text{boolean expression} \\
\mathit{name} &:= \text{method name} \\
t &:= \text{argument type} \mid \text{exception type} \mid *
\end{aligned}
$$

Figure 5.2: Grammar of Structured API Call Sequences

In Phase 1, given an API method of interest, ExampleCheck searches GitHub snippets that call the given API method, removes irrelevant statements via program slicing, and extracts API call sequences. In Phase 2, ExampleCheck finds common subsequences from individual sequences of API calls. In Phase 3, to retain conditions under which each API can be invoked, ExampleCheck mines guard conditions associated with individual API calls. In order to accurately estimate the frequency of unique guard conditions, ExampleCheck uses a SMT solver, Z3 [60], to check the semantic equivalence of guard conditions, instead of considering the syntactic similarity between them only. We manually inspect all inferred patterns to construct the data set of desirable API usage. This data set is used to report potential API misuse in the Stack Overflow posts in our study discussed in Section 5.5.

5.3.1 Structured Call Sequence Extraction and Slicing on GitHub

Given an API method of interest, ExampleCheck searches individual code snippets invoking the same method in the GitHub corpora. ExampleCheck scans 380,125 Java repositories on GitHub, collected on September 2015. To filter out low-quality GitHub repositories, we only consider repositories with at least 100 revisions and 2 contributors. To scale code search to massive corpora, ExampleCheck leverages a distributed software mining infrastructure [66] to traverse the abstract syntax trees (ASTs) of Java files. ExampleCheck visits every AST method and looks for a method invocation of the API of interest. Figure 5.3 shows a code snippet retrieved from GitHub for the `File.createNewFile` API. This snippet creates a property file, if it does not exist by calling `createNewFile` (line 18).

To extract the essence of API usage, ExampleCheck models each code snippet as a structured call sequence, which abstracts away certain syntactic details such variable names, but still retains the temporal ordering, control structures, and guard conditions of API calls in a compact manner. Figure 5.2 defines the grammar of our API usage representation. A structured call sequence consists of relevant control structures and API calls, separated by the delimiter ";". This delimiter is is a separator in our pattern grammar in Figure 5.2, not a semi-colon for ending each statement in Java. We resolve the argument types of each API call to distinguish method overloading. In certain cases, the argument consists of a complex expression such as `write(e.getFormat())`, where the partial program analysis may not be able to resolve the corresponding type. In that case, we represent unresolved types with $*$, which can be matched with any other types in the following mining phases. Each API call is associated with a guard condition that protects its usage or `true`, if it is not guarded by any condition. Catch blocks are also annotated with the corresponding exception types. We normalize a catch block with multiple exception types such as `catch (IOException | SQLException){...}` to multiple catch blocks with a single exception type such as `catch (IOException){...} catch (SQLException){...}`.

ExampleCheck builds the control flow graph of a GitHub snippet and identifies related control structures [25]. A control structure is related to the given API call, if there exists a path between the two and the API call is not post-dominated by the control structure. For instance, the API call to `createNewFile` (line 18) is control dependent on the `if` statements at lines 2 and 17 in Figure 5.3. From each control structure, we lift the contained predicate. This process is a pre-cursor for mining a common guard condition that protects each API method call in Phase 3. We use the conjunction of the lifted predicates in all relevant control structures. If an API call is in the false branch of a control structure, we negate the predicate when constructing the guard. In Figure 5.3, since `createNewFile` is in the false branch of the `if` statement at line 2 and the true branch of the `if` statement at line 17, its guard condition is `temp.equals("props.txt") && !file.exists()`. The process of lifting control predicates can be further improved via symbolic execution to account for the effect of

```java
 1 void initInterfaceProperties(String temp, File dDir) {
 2  if(!temp.equals("props.txt")) {
 3   log.error("Wrong Template.");
 4   return;
 5  }
 6  // load default properties
 7  FileInputStream in = new FileInputStream(temp);
 8  Properties prop = new Properties();
 9  prop.load(in);
10  // init properties
11  prop.set("interface", PROPERTIES.INTERFACE);
12  prop.set("uri", PROPERTIES.URI);
13  prop.set("version", PROPERTIES.VERSION);
14  // write to the property file
15  String fPath=dDir.getAbosulatePath()+"/interface.prop";
16  File file = new File(fPath);
17  if(!file.exists()) {
18   file.createNewFile();
19  }
20  FileOutputStream out = new FileOutputStream(file);
21  prop.store(out, null);
22  in.close();
23 }
```

Figure 5.3: This method is extracted as an example of File.createNewFile from the GitHub copora. Program slicing only retains the underlined statements when k bound is set to 1, since they have direct control or data dependences on the focal API call to createNewFile at line 18.

program statement before an API call. Project-specific predicates and variable names used in the guard conditions are later generalized in Phase 3 to unify equivalent guards regardless of project-specific details.

ExampleCheck performs intra-procedural program slicing [252] to filter out any statements not related to the API method of interest. For example, `Properties` API calls in Figure 5.3 should be removed, since they are irrelevant to `createNewFile`. During this process, ExampleCheck uses both backward and forward slicing to identify data-dependent statements up to k hops. Setting k to 1 retains only immediately dependent API calls in the call sequence, while setting k to ∞ includes all transitively dependent API calls. For instance, the `Properties` APIs such as `load` (line 9) and `set` (lines 11-13) are transitively dependent on `createNewFile` through variables `file`, `out`, and `prop`. Table 5.1 shows the call sequences extracted from Figure 5.3 with different k bounds. By removing irrelevant statements, program slicing significantly reduces the mining effort and also improves the mining

Table 5.1: Structured call sequences sliced using k bounds. Guard conditions and argument types are omitted for presentation purposes.

Bound	Variables	Structured Call Sequence
k=1	file	new File; if {; createNewFile; }; new FileOutputStream
k=2	file, fPath, out	getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store
k=3	file, fPath, out, prop	new Properties; load; set; set; set; getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store
k=∞	file, fPath, out, prop, in, temp	new FileInputStream; new Properties; load; set; set; set; getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store; close
No Slicing	file, fPath, out, prop, in, temp, log	if {; debug; }; new FileInputStream; new Properties; getAbsolutePath; load; set; set; set; new File; if {; createNewFile; }; new FileOutputStream; store; close

precision. Setting k to 1 leads to best performance empirically (discussed in Section 5.7).

5.3.2 Frequent Subsequence Mining

Given a set of structured call sequences from Phase 1, ExampleCheck finds common subsequences using BIDE [245]. Computing the common subsequence is widely practiced in the literature of API usage mining [229,230,244,267] and has the benefit of filtering out API calls pertinent to only a few outlier examples. In this phase, ExampleCheck focuses on mining the temporal ordering of API calls only. The task of mining a common guard condition is done in Phase 3 instead. BIDE mines *frequent closed sequences* above a given minimum support threshold σ. A sequence is a frequent closed sequence, if it occurs frequently above the given threshold and there is no super-sequence with the same support. When matching API signature, ExampleCheck matches $*$ with any other types in the same position in an API call. For example, `write(int,*)` can be matched with `write(int,String)` but will not be matched with `write(String,int)`. ExampleCheck ranks a list of sequence patterns based on the number of supporting GitHub examples, which we call support. ExampleCheck filters invalid sequence patterns that do not follow the grammar in Figure 5.2, as frequent sub-sequence mining can find invalid patterns with unbalanced brackets such as "`foo@true; }; }`".

5.3.3 Guard Condition Mining

Given a common subsequence from Phase 2, ExampleCheck mines the common guard condition of each API call in the sequence. The rationale is that each method call in the common subsequence may have a guard to ensure that the constituent API call does not lead to a failure. Therefore, ExampleCheck collects all guard conditions from each call from Phase 1 and clusters them based on semantic equivalence. The guard conditions extracted from GitHub often contain project-specific predicates and variable names. Therefore, ExampleCheck first abstracts away such syntactic details before clustering guard conditions. For each guard condition from Phase 1, ExampleCheck removes project-specific predicates by substituting them with `true`. This ensures that the generalized guard condition is still implied by the original guard after removing project-specific predicates. A predicate in a guard condition is considered project-specific or irrelevant to an API call if it does not mention the receiver object or input arguments of the given API call. In Figure 5.3, the identified guard condition of `file.createNewFile()` (line 18) is `temp.equals("props.txt") && !file.exists()`. Its first predicate `temp.equals("props.txt")` is considered irrelevant to `file.createNewFile()`, since the predicate does not check the receiver object of `createNewFile`. As a result, the guard condition is transformed to `true && !file.exists()` to generalize the irrelevant predicate. In addition, since each code snippet may use different variable names, we normalize these names in the guard conditions. ExampleCheck uses `rcv` and `argi` as the symbolic names of the receiver and the i-th input argument. For instance, the second predicte `!file.exists()` is normalized to `!rcv.exists()`, since the `file` variable is the receiver of `createNewFile`.

Table 5.2 illustrates how we canonicalize guard conditions of `String.substring`. This method takes an integer index as input and returns a substring that begins from the given index. The third guard condition in Column Guard contains a project-specific predicate,

Table 5.2: Example guard conditions of `String.substring`. API Call shows three example call sites. Guard shows the guard condition associated with each call site. Generalized shows the guard conditions after eliminating project-specific predicates. Symbolized shows the guard conditions after symbolizing variable names.

API Call	Guard	Generalized	Symbolized
s.substring(start)	start>=0 && start<=s.length()	start>=0 && start<=s.length()	arg0>=0 && arg0<=rcv.length()
log.substring(index)	-1<index && index<log.length()+1	-1<index && index<log.length()+1	-1<arg0 && arg0<rcv.length()+1
f.substring(f.indexOf("/"))	dir!=null && f.indexOf("/")>=0 && f.indexOf("/")<=f.length()	true && f.indexOf("/")>=0 && f.indexOf("/")<=f.length()	true && arg0>=0 && arg0<=rcv.length()

`dir!=null`. Since such predicate is not related to `String.substring`'s arguments or receiver object, ExampleCheck substitutes `dir!=null` with `true`, as shown in Column Generalized. All three examples name the receiver object differently—`s`, `log`, and `f` respectively. ExampleCheck replaces them with a unique symbol, `rcv`. Similarly, ExampleCheck replaces the input argument with `arg0`, as shown in Column Symbolized.

ExampleCheck initializes each cluster with each canonicalized guard. In the following clustering process, ExampleCheck checks the equivalence of every pair of clusters and merges them with if the guards are logically equivalent, until no more clusters can be merged. At the end, we count the number of guard conditions in each cluster as frequency. In a large corpus, the same logic predicate can be expressed in multiple ways. ExampleCheck checks the semantic equivalence of guard conditions, instead of syntactic similarity only. ExampleCheck formalizes the equivalence of two guard conditions as a satisfiability problem:

$$p \Leftrightarrow q \text{ is valid iff. } \neg((\neg p \vee q) \wedge (p \vee \neg q)) \text{ is unsatisfiable.}$$

ExampleCheck uses a SMT solver, Z3 [60] to check the logical equivalence between two guards during the merging process. As Z3 only supports primitive types, ExampleCheck declares variables of unsupported data types as integer variables and substitutes constants such as `null` with integers in Z3 queries. In addition, ExampleCheck substitutes API calls in a predicate to symbolic variables based on their return types. Compared with prior work [168], ExampleCheck is capable of proving the semantic equivalence of arbitrary predicates regardless of their syntactic similarity. For example, the symbolized guards of the first two examples in Table 5.2 are equivalent, even though they are expressed in different ways, `(-1<arg0 && arg0<rcv.length()+1)` and (0¡=arg0 && arg0¡=rcv.length()) respectively. Prior work [168] cannot reason about the equivalence between `-1<arg0` and `0<=arg0`. However, ExampleCheck groups these logically equivalent predicates into the same cluster using the integer theorem prover in Z3.

If ExampleCheck identifies a sequence pattern containing multiple guard patterns for each API call, ExampleCheck enumerates different guards for each API and ranks these patterns by the number of supporting code examples in the corpora. Similar to the subsequence mining in Phase 2, ExampleCheck uses a minimum support threshold θ to filter infrequent guard conditions.

We bootstrap ExampleCheck with both the sequence mining threshold σ and the guard condition mining threshold θ set to 0.5, which means sequence and guard condition patterns are reported, only if more than half of relevant GitHub snippets include them. If ExampleCheck learns no patterns with these initial thresholds, we gradually decrease both thresholds by 0.1 till finding patterns. If the mining process does not terminate after 2 hours due to too many candidate patterns, we kill the process and increase both thresholds by 0.1 accordingly. This threshold adjustment method is empirically effective to achieve a good precision (73%).

5.4 Evaluation of the API Usage Mining Framework

5.4.1 Pattern Mining Accuracy

We systematically evaluate the pattern mining accuracy of ExampleCheck using the groundtruth patterns of 30 Java API methods in MUBench [26]. These patterns are constructed based on existing bug datasets [56,97,118], previous literature [67,83], and API misuse reported by

Table 5.3: Pattern mining accuracy with different settings. No SMT and Small Corpus are run with k set to 1.

Setting	Precision (%)			Recall (%)			Rank
	Top 3	Top 5	Top 10	Top 3	Top 5	Top 10	
k = 1	79	80	79	85	91	94	3
k = 2	77	79	79	84	92	95	4
k = 3	77	80	78	84	91	94	4
k = ∞	71	74	73	77	91	94	4
No Slicing	65	65	67	73	81	89.0	9
Small Corpus	49	49	49	42	46	53	6
No SMT	78	79	79	81	88	92	3

professional developers. We do not include APIs that have no references on Stack Overflow and those with project-specific misuse related to the logic of a client program.

Table 5.3 describes the pattern mining accuracy of ExampleCheck in various settings. When setting the dependency bound to one, enabling SMT solving, and considering top five patterns, ExampleCheck learns expected patterns for all 30 API methods from massive code corpora with 80% precision and 91% recall. When considering top 10 patterns, the recall increases gradually to 94% while the precision does not vary much. Several patterns in the ground truth of MUBench occur infrequently on Github and therefore cannot be easily inferred within top 10 patterns. For instance, `PrintWriter.close` should be called in `finally` to ensure that the buffered contents are written to the stream in case an exception occurs in the middle of execution. However, among all code examples that call `PrintWriter.close` on Github, only 17% call this API in `finally`. On the other hand, ExampleCheck often learns patterns that occur frequently on Github but are not included in the ground truth of MUBench. However, patterns not in the ground truth are not necessarily useless. Table 5.4 shows examples of inferred patterns that are useful but not in our ground truth. For instance, `TypedArray` is allocated from a static pool to store the layout attributes whenever a new application view is created in Android. The corresponding pattern in our benchmark checks for missing exception handling when retrieving attributes from `TypedArray` with invalid indices, which is confirmed on Github.[3] In another perspective, `TypedArray` should be recycled immediately to avoid resource leak and GC overhead, as mentioned in the documentation.[4] This pattern is commonly practiced on Github and inferred by ExampleCheck (ranked #1).

Table 5.4: Examples of useful patterns mined from GitHub but not included in MUBench

API	Pattern	Rank	Description
TypedArray.getString	getString(1)@true; recycle(0)@true	1	Recycle TypedArray after use
RandomFileAccess.read	try {; read(0)@true; }; catch {; }	1	Catch I/O exceptions
PrintWriter.write	write(1)@true; flush(0)@true	4	Flush the stream after write
JsonElement.getAsString	getAsString(0)@rcv!=null	1	Check if JsonElement is null
InputStream.read	read(0)@true; if {; }	1	Checks if the end of the stream has been reached
SQLiteDatabase.query	query(7)@true; If {; }	2	Check if query returns null

To demonstrate the benefit of mining massive code corpora, we curate a small code corpus that contains 7,899 randomly selected GitHub projects. ExampleCheck only learns expected patterns for 19 of 30 APIs from the small corpus with 49% precision and 46% recall in top five patterns. The average rank of expected patterns is 6 in the patterns inferred from the small corpus vs. 3 from the massive corpora. ExampleCheck learns patterns with low accuracy and ranking in the small corpus for three reasons. In 3 cases, ExampleCheck does not find any relevant examples in the small corpus. In 6 cases, ExampleCheck finds several examples but none of them contain the correct API usage. In 2 cases, ExampleCheck finds several examples but the correct usage occurs very infrequently. Unless there is an efficient way of creating a corpus of high-quality code examples, we believe mining massive code corpora is a more general and accurate approach to infer patterns for arbitrary APIs than mining a pre-defined code corpus.

Even though bounding dependency analysis with lower bounds may lead to incomplete sequences with less API calls, varying the dependency bound k does not change the accuracy too much. However, compared with unbounded analysis, filtering weakly dependent API

[3]https://github.com/chrisjenx/Calligraphy/issues/41

[4]https://developer.android.com/reference/android/content/res/TypedArray.html

calls can improve the precision and recall slightly. This is because including these weakly dependent API calls may introduce additional patterns of no interest. Furthermore, mining call sequences without removing any irrelevant calls will introduce more noise during pattern mining, which degrades the precision and recall by around 13% and 10% respectively.

We hypothesize that SMT-based precondition mining should infer API preconditions more accurately, since SMT solving is capable of merging semantically equivalent preconditions which cannot be easily addressed via heuristic-based mining. However, we observe that enabling SMT solving only increases the precision and recall slightly. This is due to two reasons. First, only 9 of the 30 APIs in our benchmark require preconditions. Disabling SMT solving in the precondition mining will not affect the other 21 APIs. Second, after disabling SMT solving, the previously mined preconditions are dismissed in only 2 APIs. SMT solving is mostly useful to merge equivalent predicates that are expressed in multiple ways. However, the expected preconditions in our benchmark are simple and tend to be expressed uniquely.

5.4.2 Scalability

For each API, ExampleCheck first searches relevant code examples in the GitHub corpora and extracts sliced call sequences using Boa. Table 5.5 summarizes the search time and the number of relevant code examples across the 30 APIs. ExampleCheck finds an average of 32k code examples for each API, indicating that massive code corpora can provide sufficient examples to learn patterns from. On average, ExampleCheck takes around 10 minutes to search for relevant code examples on GitHub. Dependency analysis and sequence slicing do not impose much overhead during code search as the code search time does not vary much with different dependency analysis settings.

To demonstrate that ExampleCheck scales to a large number of code examples, we create datasets with different numbers of examples for each API and run ExampleCheck on these datasets with various settings. For APIs that have a small number of examples, we repeat existing examples to create larger datasets. The experiments run on a single machine

Table 5.5: GitHub code search using Boa

	Code Search Time (min)					Example#
	k = 1	k = 2	k = 3	k = ∞	No Slicing	
Average	9m 58s	9m 59s	10m 9s	10m 10s	10m 16s	32,678
Median	9m 41s	9m 43s	9m 52s	9m 49s	9m 48s	11,405
Max	14m 44s	12m 45s	14m 37s	16m 49s	15m 14s	294,569
Min	8m 35s	8m 26s	8m 45s	8m 38s	8m 40s	376

with 2.93GHz dual core processor and 8GB DDR3 RAM. We run ExampleCheck on each dataset of each API five times and compute the average execution time. Figure 5.4a shows the execution time that ExampleCheck takes to infer patterns on different datasets with the default thresholds (σ=0.5, θ=0.5) but different dependency analysis bounds. Compared with the unbounded dependency analysis (k=∞) which retains all dependent API calls in a sliced sequence, the bounded analysis speeds up ExampleCheck by 3.3X by only retaining immediately dependent API calls (k=1). This is because bounded analysis creates shorter API call sequences by removing weakly dependent API calls. In contrast, ExampleCheck runs up to 4.6X slower without removing any irrelevant API calls (no slicing). Figure 5.4b shows the pattern inference time with k set to 1 but different minimum support thresholds of sequence mining and precondition mining. ExampleCheck slows down significantly as the thresholds goes below 0.5. On average, ExampleCheck is capable of inferring patterns within 10 minutes with thresholds above 0.3.

5.5 A Study of API Misuse in Stack Overflow

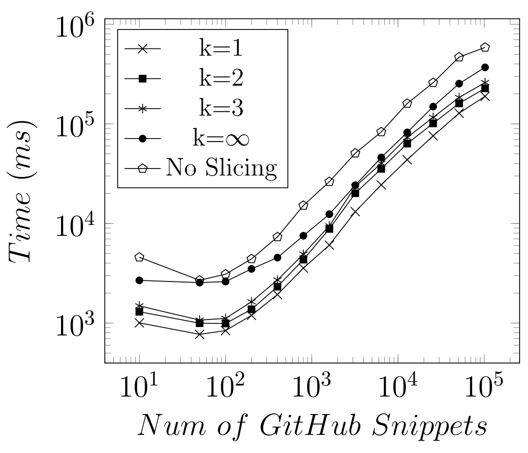
5.5.1 Data Collection

In terms of API scope, we target 100 popular Java APIs. From the Stack Overflow dump taken in October 2016,[5] we scan and parse all Java code snippets and extract API method calls. We rank the API methods based on frequency and remove trivial ones such as `System.out.println`. As a result, we select 70 frequently used API methods on Stack

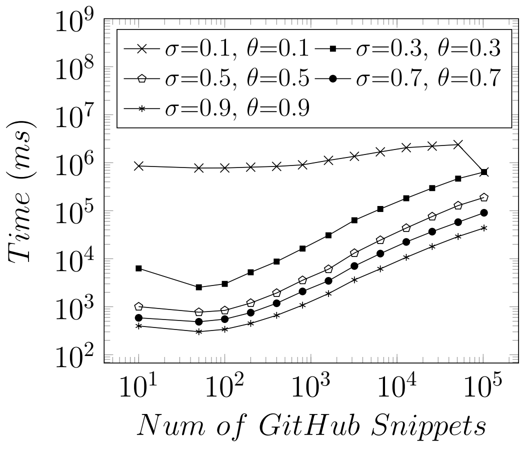
[5]https://archive.org/details/stackexchange, accessed on Oct 17, 2016.

Table 5.6: 25 samples of manually validated API usage patterns after GitHub code mining.

Domain	API	Pattern	Support	Description
Collection	ArrayList.get	loop {; get(int)@arg0<rcv.size(); }	31254	check if the index is out of bounds
	Iterator.next	iterator()@true; loop {; next()@rcv.hasNext(); }	218962	check if more elements exist to avoid `NoSuchElementException`
IO	File.createNewFile	if {; createNewFile()@!rcv.exists(); } ♣	5493	check if the file exists before creating it
	File.mkdir	mkdirs()@true	26343	call `mkdirs` instead, which also create non-existent parent directories
	FileChannel.write	try {; write(ByteBuffer)@true; close()@true; }; catch(IOException) {, } ♣	1267	close the `FileChannel` after writing to avoid resource leak
	PrintWriter.write	try {; write(String)@true; close()@true; }; catch(Exception) {; }	2473	close the `PrintWriter` after writing to avoid resource leak
String	StringTokenizer.nextToken	nextToken()@rcv.hasMoreTokens() ♣	36179	check if more tokens exist to avoid `NoSuchElementException`
	Scanner.nextLine	loop {; nextLine()@rcv.hasNextLine(); }	2510	check if more lines exist to avoid `NoSuchElementException`
	String.charAt	charAt(int)@arg0<rcv.length()	27597	check if the index is out of bounds
Regex	Matcher.find	matcher(String)@true; find()@true	5851	call `matcher` to create a `Matcher` instance first
	Matcher.group	if {; group(int)@rcv.find(); }	16447	check if there is a match first to avoid `IllegalStateException`
Database	SQLiteDatabase.query	query(String,String[],String,String[],String,String,String)@true; close()@true	5563	close the cursor returned by `query` to avoid resource leak
	ResultSet.getString	try {; getString(String)@rcv.next(); }; catch(Exception) {; }	18933	check if more results exist to avoid `SQLException`
	Cursor.close	finally {; close()@rcv!=null; }	15732	call `close` in a finally block
Android	Activity.setContentView	onCreate(Bundle)@true; setContentView(View)@true	56321	always call `super.onCreate` first to avoid exceptions
	TypedArray.getString	getString(int)@true; recycle()@true	2126	recycle the `TypedArray` so it can be reused by a later call
	SharedPreferences.Editor.edit	edit()@true; commit()@true	9650	commit the changes to SharedPreferences
	ApplicationInfo.loadIcon	getPackageManager()@true; loadIcon(PackageManager)@true	400	get a `PackageManager` as the argument of `load`
Crypto	Mac.doFinal	try {; getInstance(String)@true; getBytes()@true; doFinal(byte[])@true; }; catch(Exception) {; }	474	get a `Mac` instance first and convert the input to bytes
	MessageDigest.digest	getInstance(String)@true; digest()@true	7048	get a `MessageDigest` instance first
Network	Jsoup.connect	try {; connect(String)@true; get()@true; }; catch(IOException); { } ♣	376	call `get` to fetch the web content
	URL.openConnection	try {; new URL(String)@true; openConnection()@true; }; catch(Exception) {; }	19056	create a `URL` object first and handle potential exceptions
	HttpClient.execute	new HttpGet(String)@true; execute(HttpGet)@true	2536	create a `HttpGet` object as the argument of `execute`
Swing	SwingUtilities.invokeLater	new Runnable()@true; invokeLater(Runnable)@true	20406	create a Runnable object as the argument of invokeLater
	JFrame.setPreferredSize	setPreferredSize(Dimension)@true, pack()@true	394	call `pack` to update the `JFrame` with the preferred size



(a) Dependency Analysis Bound



(b) Minimum Support Threshold

Figure 5.4: Pattern mining performance

Overflow. They are in diverse domains, including Android, Collection, document processing (e.g., String, XML, JSON), graphical user interface (e.g., swing), IO, cryptography, security, Java runtime (e.g. Thread, Process), database, networking, date, and time. The rest 30 APIs come from an API misuse benchmark, MUBench [26], after we exclude those patterns without corresponding SO posts and those that cannot be generalized to other projects.

ExampleCheck scans over 380K GitHub projects and finds an average of 55144 relevant code snippets for each API method, ranging from 211 to 450,358 snippets. This result indicates that massive corpora can provide sufficient code snippets to learn API usage patterns from. ExampleCheck infers 245 API usage patterns for the 100 APIs in our study scope. This initial set of patterns may include invalid or incorrect patterns. Therefore, we manually inspect the 245 inferred patterns carefully and exclude incorrect ones based on online documentation and pattern frequencies. The overall precision is 73%, resulting in 180 validated, correct patterns that we can use for the empirical study in Section 5.5. These 180 validated patterns cover 85 of the 100 API methods. The rest 15 API methods do not converge to any API usage patterns that can be confirmed by online documentation, since they are simple to use and do not require additional guard conditions or additional API calls. For example, `System.nanoTime` can be used stand-alone to obtain the current system time. Even though these 15 API methods do not have any patterns, we still include them in the scope of Stack

Overflow study, since they represent a category of simple API methods that programmers are less likely to make mistakes.

During the inspection process, each pattern is annotated as either *alternative* or *required*. A code snippet should satisfy one of alternative patterns and must satisfy all required patterns. For example, ExampleCheck learns `firstKey()@rcv.size()>0` and `firstKey()@-!rcv.isEmpty()` for `SortedMap.firstKey`. Both patterns ensure that a sorted map is not empty before getting the first key to avoid `NoSuchElementException`. They are considered alternative to each other. As an example of required patterns, programmers must handle potential `IOException`, when reading from a stream (e.g., `FileChannel`), and close it to avoid resource leaks.

Table 5.6 shows 25 samples of validated API patterns in 9 domains. Alternative patterns are marked with ♣. Column Description describes each pattern. For instance, `TypedArray` is allocated from a static pool to store the layout attributes, whenever a new application view is created in Android. It should be recycled immediately to avoid resource leaks and GC overhead, as mentioned in the JavaDoc.[6] This pattern is supported by 2126 of 3348 related snippets in GitHub and inferred by ExampleCheck (ranked #1). The entire data set of API usage patterns for all 100 APIs and the list of SO posts with potential API usage violations are publicly available.[7]

We collect all Stack Overflow posts relevant to the 100 Java APIs in our study scope from the Stack Overflow data dump. We extract code examples in the markdown `<code>` from SO posts with the `Java` tag and consider code examples in the answer posts only, since code appearing in the question posts is buggy and rarely used as examples. We gather additional information associated with each post, including view counts, vote scores (i.e., upvotes minus downvotes), and whether a post is accepted as a correct answer.

Previous studies have shown that online code snippets are often unparsable [223,259] and contain ambiguous API elements [55] due to the incompleteness of these snippets. Exam-

[6]https://developer.android.com/reference/android/content/res/TypedArray.html

[7]http://web.cs.ucla.edu/~tianyi.zhang/examplecheck.html

pleCheck leverages a state-of-the-art partial program parsing and type resolution technique to handle these incomplete snippets, whose accuracy of API resolution is reported to be 97% [224]. Code examples that call overridden APIs or ambiguous APIs (i.e., APIs with the same name but from different Java classes) are filtered by checking the argument and receiver types respectively. In total, we find 217,818 SO posts with code examples for the 100 APIs in our study scope. Each post has 7644 view counts on average.

ExampleCheck checks whether the structured call sequence of a Stack Overflow code example is *subsumed* by the desirable API usage in the pattern set. A structured call sequence s is subsumed by a pattern p, only if p is a subsequence of s and the guard condition of each API call in s implies the guard of the corresponding API call in p. During this subsumption checking process, the guard conditions in Stack Overflow code examples are generalized in the same manner before checking logical implication using Z3. For a SO post with multiple method-level code snippets, ExampleCheck inlines invoked methods before extracting the structured call sequence in order to emulate a lightweight inter-procedural analysis.

5.5.2 Manual Inspection of Stack Overflow

To check whether Stack Overflow posts with potential API misuse reported by ExampleCheck indeed suggest undesirable API usage, the first and the third authors manually check 400 random samples of SO posts with reported API usage violations. We read the text descriptions and comments of each post and check whether the surrounding narrative discusses how to prevent the violated pattern. If there are multiple code snippets in a post, we first combine them all together and check them as a single code example. We also account for aliasing during code inspection. We examine whether the reported API usage violation could produce any potential behavior anomaly, such as program crashes and resource leaks on a contrived input data or program state and whether such anomaly could have been eliminated by following the desirable pattern. For short posts, this inspection takes about 5 minutes each. For longer posts with a big chunk of code or multiple code fragments, it takes around 15 to 20 minutes. To reduce subjectivity, the two authors inspect these posts independently.

The initial inter-rater agreement is 0.84, measured by Cohen's kappa coefficient [242]. The two authors resolve disagreements on all but two posts, and the kappa coefficient after the discussion is 0.99. The two authors disagree how helpful reported violations are in two posts, where API usage violations in these posts are either clarified in surrounding natural language explanations or mentioned in post comments.

True Positive 289 out of 400 inspected Stack Overflow posts (72%) contain real API misuse, confirmed by both authors. For instance, the following example demonstrates how to retrieve records from `SQLiteDatabase` using `Cursor` but forgets to close the database connection at the end.[8] Programmers should always close the connection to release all its resources. Otherwise, it may quickly run out of memory, when retrieving a large volume of data from the database frequently.

```java
public ArrayList<UserInfo> get_user_by_id(String id) {
    ArrayList<UserInfo> listUserInfo = new ArrayList<UserInfo>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.query(...);

    if (cursor != null) {
        while (cursor.moveToNext()) {
          UserInfo userInfo = new UserInfo();
          userInfo.setAppId(cursor.getString(cursor.getColumnIndex(COLUMN_APP_ID)));
          // HERE YOU CAN MULTIPLE RECORD AND ADD TO LIST
          listUserInfo.add(userInfo);
        }
    }
    return listUserInfo;
}
```

In many cases, a code example may function well with some crafted input data, even though it does not follow desirable API usage. For example, programmers should check whether the return value of `String.indexOf` is negative to avoid `IndexOutOfBoundsExcep-`

[8]https://stackoverflow.com/questions/31531250

tion. The example below does not follow this practice, but still works well with a hard-coded constant, text.[9] One can argue that the input data is hard-coded for illustration purposes only, as the role of Stack Overflow post is to provide a starting point rather than teaching complete details of correct API usage. However, if a programmer reuses this code example and replaces the hard-coded text with a function call reading from a html file, the reused code may crash if the html document does not have an expected element. Therefore, it is still beneficial to inform the users about desirable usage and potential pitfalls, especially for a novice programmer who may not be familiar with the given API.

```
String text = "<img src=\"mysrc\" width=\"128\" height=\"92\" border=\"0\" alt=\"alt\"
    /><p><strong>";
text = text.substring(text.indexOf("src=\""));
text = text.substring("src=\"".length());
text = text.substring(0, text.indexOf("\""));
System.out.println(text);
```

False Positive ExampleCheck mistakenly detects API misuse in 64 posts. The majority reason is that ExampleCheck checks for API misuse via a sequence comparison without deep knowledge of its specification, which is not sufficient in 56 posts. For instance, the following SO post calls substring (line 5) without explicitly checking whether the start index (index+1) is not a negative number and the end index (strValue.length()) is not greater than the length of the string.[10] While ExampleCheck warns potential API misuse, according to JDK specifications, indexOf never returns a negative integer $\leq$ -2. Thus, the following code is still safe, because index+1 is guaranteed to be non-negative. Similarly, strValue.length() returns the string's length, which cannot be out of bounds. Such cases require having detailed specifications, such as the return value of indexOf() is always $\geq$1.

```
1 public String getDecimalFractions(BigDecimal value) {
2     String strValue = value.toPlainString();
3     int index = strValue.indexOf(".");
```

[9]https://stackoverflow.com/questions/12742734

[10]http://stackoverflow.com/questions/7473462

```
    if(index != -1) {
        return strValue.substring(index+1, strValue.length());
    }
    return "0";
}
```

Second, 36 false positives are correct but infrequent alternatives. ExampleCheck does not learn these alternative usage patterns, because they do not commonly appear in GitHub. For example, programmers should first call `new SimpleDateFormat` to instantiate `SimpleDateFormat` with a valid date pattern and then call `format`, which is supported by 18,977 related GitHub snippets. An alternative way is to instantiate `SimpleDateFormat` by calling `getInstance`, as shown in the following SO post.[11] This alternative usage is supported by 360 GitHub snippets and therefore not inferred by ExampleCheck due to its low frequency.

```
... some other code...
public String toString() {
    Calendar c = new GregorianCalendar();
    c.set(Calendar.DAY_OF_WEEK, this.toCalendar());
    SimpleDateFormat sdf=(SimpleDateFormat)SimpleDateFormat.getInstance();
    sdf.applyPattern("EEEEEEEEEE");
    return sdf.format(c.getTime());
}
```

In some SO posts, users explicitly state in surrounding natural language text that the given code example must be improved during integration or adaptation. The following example shows how to load a `Class` instance by name and then cast the class.[12] The author of this post comments that "*be aware that this might throw several Exceptions, e.g. if the class defined by the string does not exist or if `AnotherClass.classMethod()` doesn't return an instance of the class you want to cast to.*" ExampleCheck still flags the post because of a missing exception handling, since the desirable API usage is not reflected in the embedded code. However, it is certainly possible that SO users will read both the code and surrounding

[11]https://stackoverflow.com/questions/2243850

[12]https://stackoverflow.com/questions/4650708

text carefully and investigate how to handle edge cases narrated in the text.

```java
Class<?> myclass = Class.forName("myClass_t");
myClass_t myVar = (myClass_t)myclass.cast(AnotherClass.classMethod());
```

Sometimes, Stack Overflow users split a single code example into multiple fragments and provide step-by-step explanation, which is considered as a better way of answering questions in Stack Overflow [166]. ExampleCheck may report API misuse if two related API calls are split in different code fragments.[13] This can be addressed by stitching these snippets together during analysis.

5.5.3 Is API Misuse Prevalent on Stack Overflow?

ExampleCheck detects potential API misuse in 66,897 (31%) out of 217,818 Stack Overflow posts in our study. We manually label each API pattern with its corresponding domain as well as the consequence of each possible violation. Then we write scripts to categorize reported violations based on their domains and based on their consequences. Figure 5.5 shows the prevalence of API misuse from different domains. Database, IO, and network APIs are often misused, since they often require to handle potential runtime exceptions and close underlying streams to release resources properly at the end. Similarly, many cryptography related posts are flagged as unreliable, due to unhandled exceptions. Stack Overflow posts on string and text manipulation often forget to check the validity of input data (e.g., whether the input string is empty) or return values (e.g., whether the returned character index is -1).

Among posts with potential API misuse reported by ExampleCheck, 76% could potentially lead to program crashes, e.g., unhandled runtime exceptions. 18% could lead to incomplete action, e.g., not completing a transaction after modifying resources in Android, or not calling `setVisible` after modifying the look and feel of a swing GUI widget. 2% could lead to resource leaks in operating systems, e.g., not closing a stream. We fully acknowledge that not all detected violations could lead to bugs when ported to a target application. To

[13]https://stackoverflow.com/questions/11552754

accurately assess the runtime impact of SO code examples, one must systematically integrate these examples to real-world target applications and run regression tests.

Many SO examples aim to answer a particular programming question. Therefore, authors of these examples may assume SO users who posted these questions already know about the used APIs and may not include complete details of desirable API usage. However, given that each post has 7,644 view counts on average, some users may not have similar background knowledge. Especially for novice programmers, it may be useful to show extra tips about desirable API usage evidenced by a large number of GitHub code snippets. We also find that SO posts with API misuse are more frequently viewed than those posts without API misuse, 8365 vs. 7276 on average. Therefore, there is an opportunity to help users consider better or alternative API usage mined from massive corpora, when they stumble upon SO posts with potential API misuse.

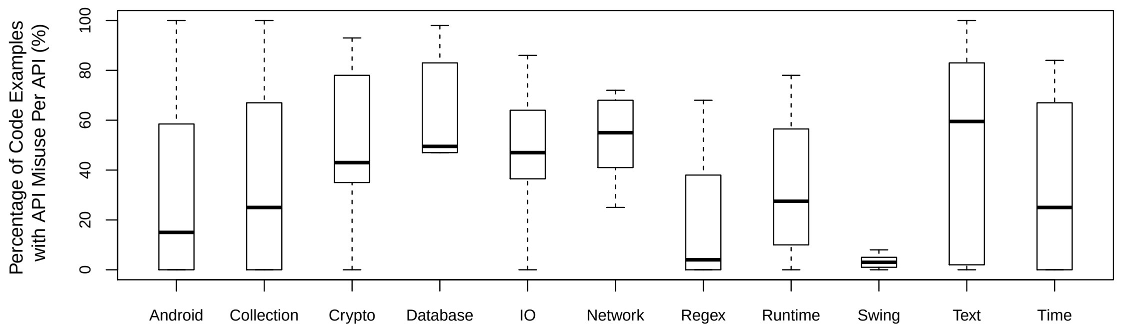


Figure 5.5: API Misuse Comparison between Different Domains

5.5.4 Are highly voted posts more reliable?

Stack Overflow allows users to upvote and downvote a post to indicate the applicability and usefulness of the post. Therefore, votes are often considered the main quality metric of Stack Overflow examples [166]. However, we find that highly voted posts are not necessarily more reliable in terms of correct API usage. Figure 5.6 shows the percentage of SO posts with different vote scores that are detected with at least one API usage violation. We perform a linear regression on the vote score and the percentage of unreliable examples, as shown by the red line in Figure 5.6. We do not observe a strong positive or negative correlation between the vote of a post and its reliability in terms of API misuse. A previous study shows that concise code examples and detailed step-by-step explanations are two key factors of highly voted Stack Overflow posts [166]. Our manual inspection confirms that many unreliable examples are simplified to operate on crafted input data for illustration purposes only (Section 5.5.2). Such curated examples are not sufficient for various input data and usage scenarios in real software systems, especially for handling corner cases. Therefore, votes alone should not be used as a single indicator of the quality of online code examples. To improve the quality of curated examples, Stack Overflow needs another mechanism that helps developers understand the limitation of existing examples and decide how to integrate the given example to production code.

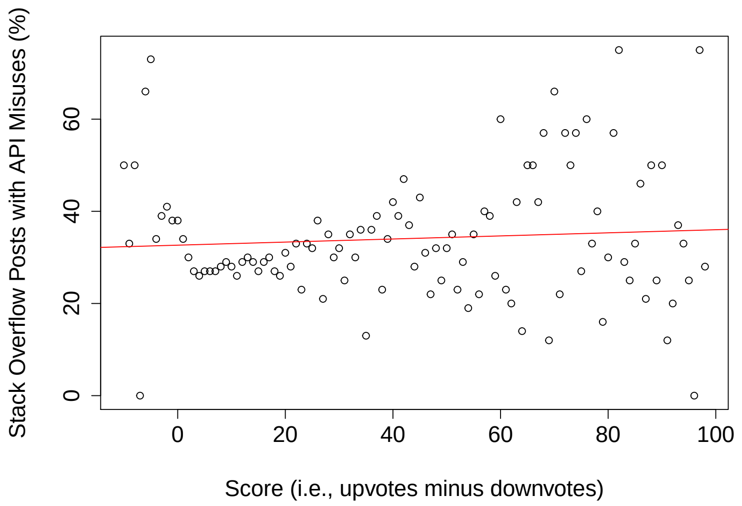


Figure 5.6: API Misuse Comparison between Code Examples with Different Vote Scores on Stack Overflow

5.5.5 What are the characteristics of API misuse?

We classify the detected API usage violations into three categories based on the required edits to correct the violations.

Missing Control Constructs. Many APIs should be used in a specific control-flow context to avoid unexpected behavior. This type of API usage violations can be further split based on the type of missing control constructs.

Missing exception handling. If an API may throw an exception, the thrown exception should either be caught and handled a `try-catch` block or be declared in the method header. In total, we find 17,432 code examples that do not handle exceptions properly. For example, `Integer.parseInt` may throw `NumberFormatException` if the string does not contain a parsable integer. The following example will crash, if a user enters an invalid integer.[14] A good practice is to surround `parseInt` with a try-catch block to handle the potential exception. Unlike *checked exceptions* such as `IOException`, runtime exceptions such as `NumberFormatException` will not be checked at compile time. In such cases, it would be helpful to inform users about which runtime exceptions must be handled based on common exception handling usage in GitHub.

```java
Scanner scanner = new Scanner(System.in);
System.out.print("Enter Number of Students:\t");
int numStudents = Integer.parseInt(scanner.nextLine());
```

Missing if checks. Some APIs may return erroneous values such as null pointers, which must be checked properly to avoid crashing the succeeding execution. For example, `TypedArray.getString` may return null, if the given attribute is not defined in the style resource of an Android application. Therefore, the return value, `customFont` must be checked before passing it as an argument of `setCustomFont` (line 6) to avoid `NullPointerException`, which is violated by the following Stack Overflow example.[15]

```java
1 public class TextViewPlus extends TextView {
2   ... some other code ...
3   private void setCustomFont(Context ctx, AttributeSet attrs) {
4       TypedArray a = ctx.obtainStyledAttributes(attrs, R.styleable.TextViewPlus);
5       String customFont = a.getString(R.styleable.TextViewPlus_customFont);
6       setCustomFont(ctx, customFont);
7       a.recycle();
8    }
9 }
```

[14]https://stackoverflow.com/questions/3137481

[15]https://stackoverflow.com/questions/7197867

Missing finally. Clean-up APIs such as `close` should be invoked in a finally block in case an exception occurs before invoking those APIs. 83% of Stack Overflow examples that call `Cursor.close` does not call it in a finally block, shown in the following.[16] `Cursor.close` may be skipped, if `getString` (line 5) throws an exception.

```
1 Cursor emails = contentResolver.query(Email.CONTENT_URI,...);
2 while (emails.moveToNext()) {
3     String email = emails.getString(emails.getColumnIndex(Email.DATA));
4     break;
5 }
6 emails.close();
```

Missing or Incorrect Order of API calls. In certain cases, multiple APIs should be called together in a specific order to achieve desired functionality. Missing or incorrect order of such API calls can lead to unexpected behavior. For example, developers must call `flip`, `rewind`, or `position` to reset the internal cursor of `ByteBuffer` back to the previous position to read the buffered data properly. The following SO example could throw `BufferUnderflowException`, if the internal cursor already reached the upper bound of the buffer after the `put` operation at line 2.[17] Without resetting the internal cursor, the next `getInt` operation at line 3 would start reading from the upper bound, which is prohibited. We find 7,956 posts that either misses a critical API call or calls APIs in an incorrect order.

```
1 ByteBuffer bb = ByteBuffer.allocate(4);
2 bb.put(newArgb);
3 int i = bb.getInt();
```

Incorrect Guard Conditions. Many APIs should be invoked under the correct guard condition to avoid runtime exceptions. For instance, programmers should check whether a sorted map is empty with a guard like `map.size()>0` or `!map.isEmpty()` before calling `firstKey` (API#9) on the map. However, the following calls `firstKey` on an empty map

[16]https://stackoverflow.com/questions/31427468

[17]http://stackoverflow.com/questions/12100651

without a guard, leading to `NoSuchElementException`.[18] Surprisingly, this example is accepted as the correct answer and also upvoted by six other developers on Stack Overflow. We find 12,791 posts with incorrect guard conditions.

```
TreeMap map = new TreeMap();
//OR SortedMap map = new TreeMap()
map.firstKey();
```

5.6 Augmenting Stack Overflow with API Usage Patterns mined from GitHub

The study results in the previous section indicate that even highly voted and frequently viewed SO posts do not necessarily follow desirable API usage. There is an opportunity to help developers consider better or alternative API usage that is mined from massive corpora and is supported by thousands of GitHub snippets. Certainly, the goal of Stack Overflow is to provide 'quick snippets' and not to share complete details of API usage or present compilable, runnable code. Rather, Stack Overflow often serves the purpose of providing a starting point and helping the user to grasp the gist of how the API works by omitting associated details such as which guard conditions to check and which runtime exceptions to handle. Nevertheless, it would be useful for a user to see related API usage along with concrete examples substantiating the desirable API usage, when the user is browsing the given SO post. Such information may reduce the effort of integrating, adapting, and testing the given code example. In this section, we present a Chrome extension that we design to augment a given SO post with mined API usage patterns.

5.6.1 Tool Features and Implementations

This section describes the tool implementation details of ExampleCheck. Figure 5.7 shows the architecture of ExampleCheck. The API usage mining process is computed offline

[18]http://stackoverflow.com/questions/21983867

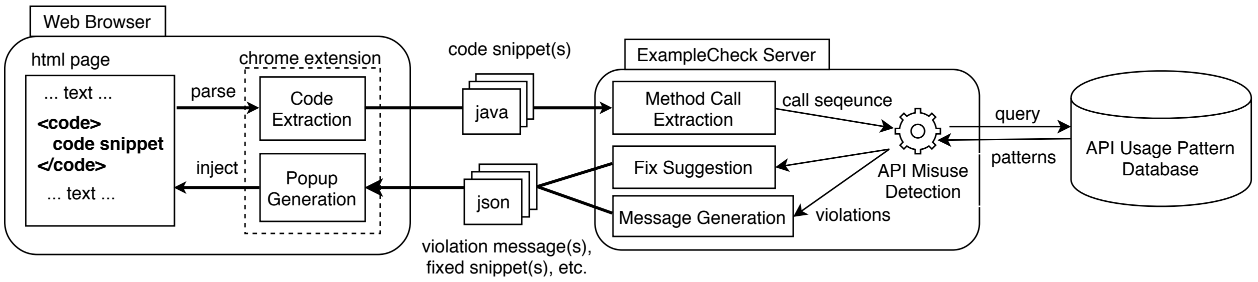


Figure 5.7: An overview of ExampleCheck's architecture

and the resulting patterns are stored in a database. The technical details and evaluation of API usage mining technique is presented in our ICSE 2018 paper [264]. When a user loads a Stack Overflow page in the Chrome browser, the Chrome extension extracts code snippets within `<code>` tags in answer posts, and sends them to the back-end server. The back end then detects API usage violations in a snippet and synthesizes warning messages and corresponding fixes. For each misused method call in a snippet, the Chrome extension generates a pop-up window using the Bootstrap popover plug-in[19] to inform the user about the API misuse information.

API Usage Mining and the Resulting Pattern Set. Our mining technique in [264] leverages a distributed software mining infrastructure [66] to search over the corpus of 380K GitHub projects. Given an API method of interest, it identifies code fragments that use the same method in the GitHub corpus and performs program slicing to remove statements that are not related to the given method. Then it combines frequent subsequence mining and SMT-based guard condition mining to retain important API usage features, including the temporal ordering of related API calls, enclosing control structures, and guard conditions that protect an API call. We evaluated the mining technique using 30 API methods from MUBench [26]. Our mining technique has 80% precision and 91% recall, when considering top 5 patterns for each API method.

In Section 5.5, we mined API usage patterns of 100 popular Java API methods and carefully inspected 245 inferred patterns based on online documentation. As a result, we curated a dataset of 180 validated, correct patterns for API misuse detection, which covers API us-

[19]https://www.w3schools.com/bootstrap/bootstrap_popover.asp

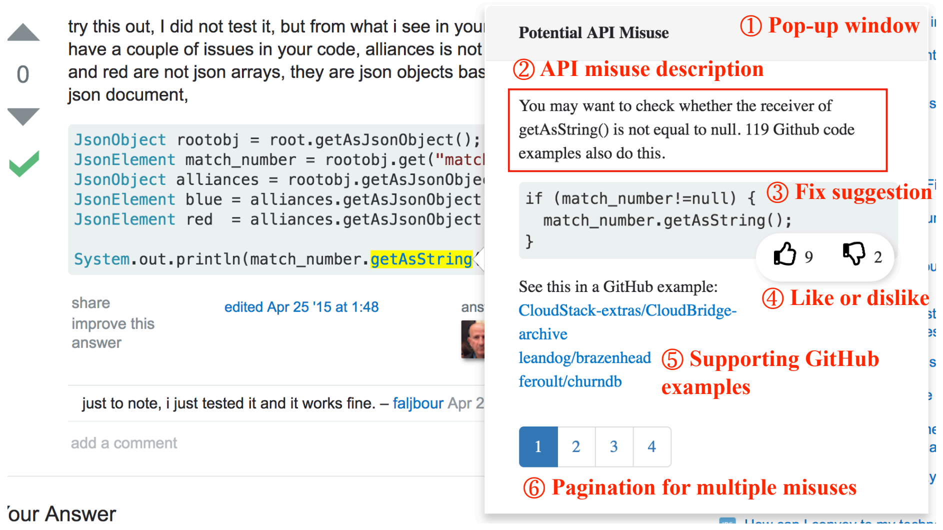
ages shown in 217K SO posts in Java. These patterns are represented as API call sequences with surrounding control constructs. Each API call is also annotated with its argument types and guard conditions. For example, one pattern, `loop {; get(int)@arg0<rcv.size(); }`, checks if the index is out of bounds when calling the `get` method on an `ArrayList` object.

API Misuse Detection. Given a code snippet sent from the browser, the server first extracts the API call sequence from the snippet. We use a partial program analysis and type resolution technique [224] to parse incomplete snippets and resolve ambiguous types. If a SO snippet has multiple methods, ExampleCheck inlines the call sequence of an invoked method into the sequence of the caller to emulate a lightweight inter-procedural analysis. ExampleCheck then queries the pattern database for the API calls present in each API call sequence. Given an API call sequence and an API usage pattern, it checks whether (1) the API calls and control constructs in the sequence follow the same temporal order in the pattern, and (2) the guard condition of an API call in the sequence implies the guard of the corresponding API call in the pattern. ExampleCheck uses a SMT solver, Z3 [60], to check whether one guard condition implies another. ExampleCheck is capable of detecting three types of API usage violations—*missing control constructs*, *missing or incorrect order of API call*, and *incorrect guard condition*.

Table 5.7: Warning message templates for different types of API usage violations. <?> and <before/after> are instantiated based on API usage violations and correct patterns. The digits in the last column are the SO post ids of the warning examples.

Violation Type	Description Template	Example Warning Message
Missing/Incorrect Order of API calls	You may want to call <?> **<before/after>** calling <?>	You may want to call `TypedArray.recycle()` after calling `TypedArray.getString()`. [35784171]
Missing Control Constructs	You may want to call the API method <?> in <?>	You may want to call `Cursor.close()` in a finally block. [31427468]
Missing Try-Catch	You may want to handle the potential <?> exception thrown by <?> using a try-catch block	You may want to handle the potential `SQLException` thrown by `PreparedStatement.setString()` using a try-catch block. [11183042]
Incorrect Guard Conditions	You may want to check whether <?> is true before calling <?>	You may want to check whether `iterator.hasNext()` is true. [25789601]

Warning Message Generation. Given an API usage violation and the correct pattern, ExampleCheck generates a warning message that describes the violation in natural language text. Table 5.7 shows the warning message templates for different types of API usage violations. In each template, <?> is instantiated with the corresponding API calls or control constructs based on the detected API usage violation and the correct pattern.



try this out, I did not test it, but from what i see in you
have a couple of issues in your code, alliances is not
and red are not json arrays, they are json objects bas
json document,

```
JsonObject rootobj = root.getAsJsonObject();
JsonElement match_number = rootobj.get("matc
JsonObject alliances = rootobj.getAsJsonObje
JsonElement blue = alliances.getAsJsonObject
JsonElement red  = alliances.getAsJsonObject

System.out.println(match_number.getAsString
```

just to note, i just tested it and it works fine. – faljbour Apr 2

Potential API Misuse

You may want to check whether the receiver of getAsString() is not equal to null. 119 Github code examples also do this.

```
if (match_number!=null) {
  match_number.getAsString();
}
```

See this in a GitHub example:
CloudStack-extras/CloudBridge-archive
leandog/brazenhead
feroult/churndb

Figure 5.8: The ExampleCheck Chrome extension that augments Stack Overflow with API misuse warnings. The pop-up window alerts that `match_number` can be `null` if the requested `JSON` attribute does not exist and will crash the program by throwing `NullPointerException` when `getAsString` is called on it.

<before/after> is instantiated based on the relative order of the two API calls in the correct pattern. The warning messages also describe which exception types are not handled in the snippets detected with *missing try-catch* violations. To help users understand the prevalence of a recommended API usage pattern, the warning message also quantifies how many other code fragments follow the same pattern in GitHub.

Fix Suggestion. ExampleCheck further suggests a correct way of using an API method by synthesizing a readable fixed snippet based on the original SO snippet. ExampleCheck first matches each API call in the recommended API usage pattern with the given SO snippet. If an API call is matched, ExampleCheck reuses the same receiver object and arguments of the corresponding API call from the original SO snippet in the synthesized snippet. Otherwise, ExampleCheck names the receiver and arguments based on their types. For example, if the receiver type of an unmatched API call (i.e., a *missing-API-call* violation) is `File`, ExampleCheck names the receiver object as `file`, the lower case of the receiver type. In this way, ExampleCheck reduces the mental gap for switching between the original SO post and the recommended snippet.

5.6.2 Demonstration Scenario

Suppose Alice wants to read attribute values from a `JSON` message using Google's Gson library. Alice searches online and finds a related Stack Overflow post with an illustrative code example, as shown in Figure 5.8.[20] Though this post is accepted as a correct answer, it does not properly use the `JsonElement.getAsString` method, which gets the `string` value of a `JSON` element. For example, if the requested attribute does not exist in the `JSON` message, the preceding API call, `JsonObject.get` will return `null`, which consequently leads to `NullPointException` when calling `getAsString` on the returned object. If Alice puts too much trust on this example of the SO post, she may inadvertently follow an unreliable solution, which might lead to runtime errors in some corner cases.

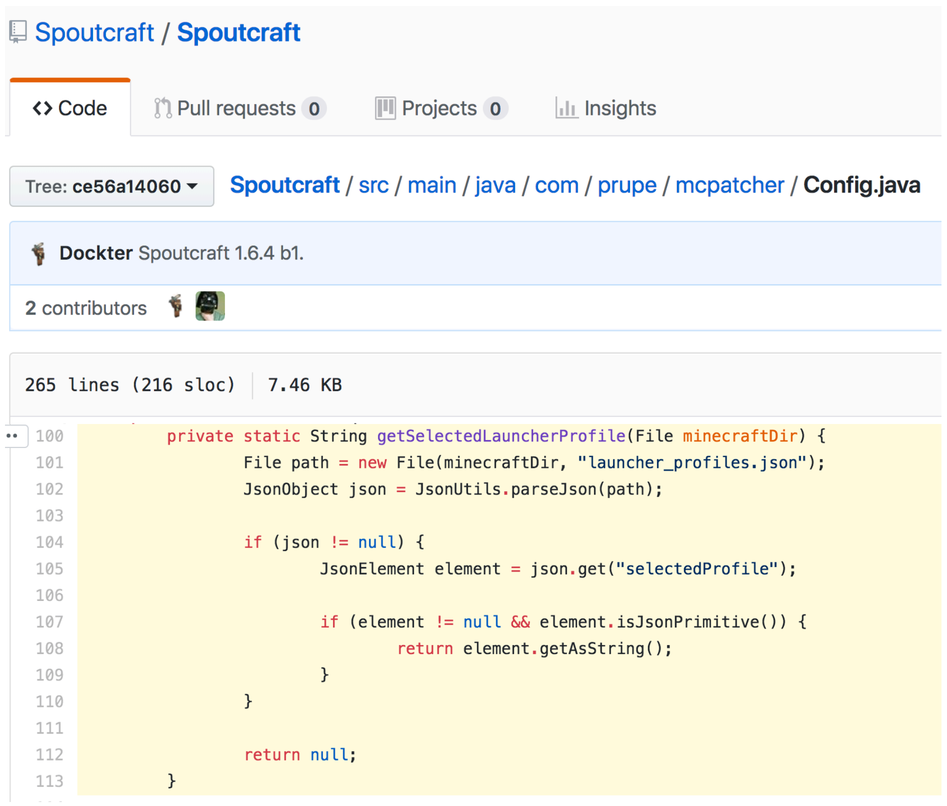
Alice cannot easily recognize the potential limitation of the given SO post, unless she manually investigates other similar code examples. ExampleCheck frees Alice from this manual investigation labor by contrasting a Stack Overflow post with common API usage patterns mined from over 380K GitHub repositories. ExampleCheck then highlights the potential API usage violations in the Stack Overflow post. When Alice clicks on a highlighted API call, ExampleCheck generates a pop-up window with detailed descriptions about the API usage violation, as shown in Figure 5.8.

API misuse description. To help Alice understand a detected API usage violation, ExampleCheck translates the violation to a natural language description (① in Figure 5.8). From the warning message, Alice learns that she should check whether the `JsonElement` object is `null` before calling `getAsString`. ExampleCheck also displays a message that 119 GitHub examples also follow this usage pattern. Such quantification can provide additional evidence about how many real-world examples are different from the given SO snippet.

Fix suggestion. ExampleCheck further sketches how to correct the violation in the original SO post, as shown in ③ in Figure 5.8. This fix is an embodiment of the correct API usage pattern in the context of the SO post. To reduce the gap between the fix and the

[20]https://stackoverflow.com/questions/29860000

[21]https://goo.gl/YHo1UM



Spoutcraft / Spoutcraft

Spoutcraft / src / main / java / com / prupe / mcpatcher / Config.java

Dockter Spoutcraft 1.6.4 b1.

2 contributors

265 lines (216 sloc) | 7.46 KB

```
	private static String getSelectedLauncherProfile(File minecraftDir) {
		File path = new File(minecraftDir, "launcher_profiles.json");
		JsonObject json = JsonUtils.parseJson(path);

		if (json != null) {
			JsonElement element = json.get("selectedProfile");

			if (element != null && element.isJsonPrimitive()) {
				return element.getAsString();
			}
		}

		return null;
	}
```

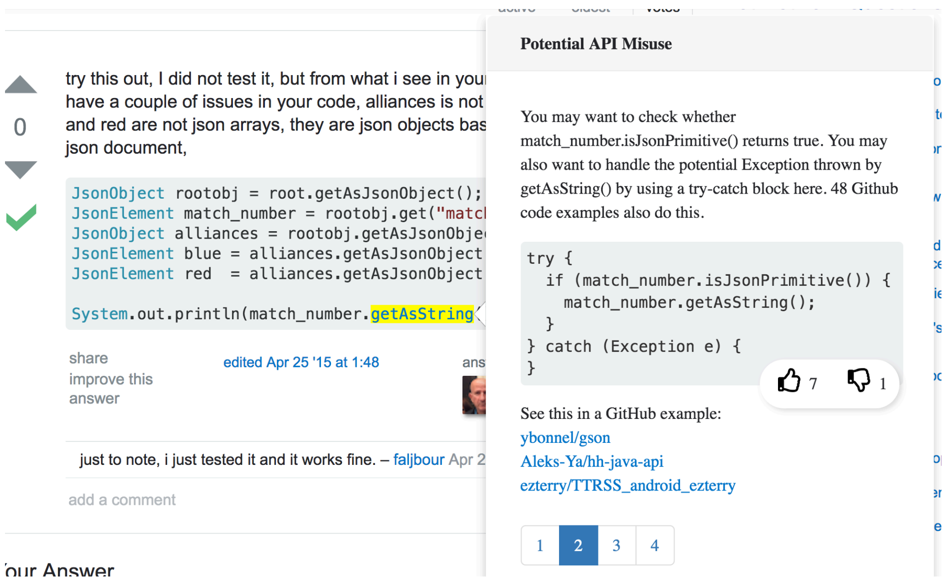
Figure 5.9: A programmer can view a concrete code example from GitHub that follows a correct API usage pattern, when clicking on a GitHub example link in the pop-up window.[21]

original post, ExampleCheck reuses the same variable names in the original SO posts to generate a suggestion with improved API usage. For example, the `JsonElement` variable in the generated example is named as the same variable, `match_number` in the original post.

Linking GitHub examples. To help Alice understand how the same API method is used in real-world projects, ExampleCheck provides several GitHub examples that follow the suggested API usage pattern (⑤ in Figure 5.8). Alice is curious about how others use `JsonElement.getAsString`. When she clicks on the link of the first GitHub example, ExampleCheck redirects Alice to a GitHub page and automatically scrolls down to the Java method where `JsonElement.getAsString` is called, as shown in Figure 5.9. Compared with the simplified SO example in Figure 5.8, this GitHub code is more carefully constructed with multiple `if` checks. For example, it not only checks whether the `JsonElement` object is `null`, but also checks whether it is a primitive type to avoid `ClassCastException` before calling `getAsString`. By providing the traceability to concrete code examples in GitHub, Alice could gain a more comprehensive view of correct API usage in production code, which may not be illustrated in simplified code examples in Stack Overflow.

User feedback. After investigating the concrete example in GitHub, Alice finds it necessary to perform a `null` check. She upvotes the pattern by clicking on the "thumbs-up" button to notify other users that this detected violation is helpful (④ in Figure 5.8). Alice also finds that her decision resonates with the majority of ExampleCheck users, since nine users also upvoted this violation.

Multiple API usage violations. If a method call in a SO post violates multiple API usage patterns, ExampleCheck displays them in separate pages in a pop-up window. These pages are first ranked by the vote score (i.e., upvotes minus downvotes) of each violated pattern, and then by the number of GitHub examples that support a pattern if two patterns have the same vote score. As shown in ⑥ in Figure 5.8, the method call, `getAsString` violates four API usage patterns. Figure 5.10 shows the second violated pattern and suggests Alice to check whether the `JsonElement` object represents a `JSON` primitive value before calling `getAsString`. Otherwise, `getAsString` will throw `ClassCastException`. ExampleCheck also suggests Alice to wrap `getAsString` with a `try-catch` block to handle potential excep-


try this out, I did not test it, but from what i see in your
have a couple of issues in your code, alliances is not
and red are not json arrays, they are json objects bas
json document,

```
JsonObject rootobj = root.getAsJsonObject();
JsonElement match_number = rootobj.get("matc
JsonObject alliances = rootobj.getAsJsonObje
JsonElement blue = alliances.getAsJsonObject
JsonElement red  = alliances.getAsJsonObject

System.out.println(match_number.getAsString
```

just to note, i just tested it and it works fine. – faljbour Apr 2

Potential API Misuse

You may want to check whether match_number.isJsonPrimitive() returns true. You may also want to handle the potential Exception thrown by getAsString() by using a try-catch block here. 48 Github code examples also do this.

```
try {
  if (match_number.isJsonPrimitive()) {
    match_number.getAsString();
  }
} catch (Exception e) {
}
```

See this in a GitHub example:
ybonnel/gson
Aleks-Ya/hh-java-api
ezterry/TTRSS_android_ezterry

Figure 5.10: Another API usage warning that reminds programmers to check whether the `JsonElement` object represents a `JSON` primitive value by calling `isJsonPrimitive`. It also suggests to catch potential exceptions thrown by `getAsString`.

tions. This pattern is supported by 48 GitHub examples.

5.7 Threats to Validity

In terms of *external validity*, our study is limited to 100 Java APIs that frequently appear in Stack Overflow and thus may not generalize to other Java APIs or different languages. Our scope is limited to code snippets found on Stack Overflow. Other types of online resources such as programming blogs and other Q&A forums may have better curated examples. ExampleCheck may overlook or mis-identify API misuses due to the limitations discussed in Section 5.5.2. According to the manual inspection of 400 sampled SO posts with detected API usage violations, ExampleCheck detects API misuse with 72% precision (Section 5.5.2). While the precision is rather low, ExampleCheck could be still useful in the case of false positives, since the goal of ExampleCheck is not to discard SO posts with potential API violations, but rather to suggest desirable or alternative API usage details to the users.

In terms of *construct validity*, we only assess the reliablity of online code examples in terms of API misuses. The reliability of online code examples can be affected by other issues such as logic bugs and performance bugs. However, it is still inspiring to demonstrate the prevalence and severity of API misuses in online code examples. This empirical study can be extended beyond API misuses by integrating online code examples to real-world programs and assessing their runtime impacts.

5.8 Summary

This chapter presents ExampleCheck, the first API usage mining technique that scales to 380K Java projects on GitHub. ExampleCheck captures a variety of API usage semantics including control structures, exception handling logic, guard conditions, and the temporal ordering of API calls. Using the common API usage patterns mined from GitHub, we conduct an empirical study of 220K Stack Overflow posts and find that almost 31% of these posts have potential API usage violations that could produce anomalies such as program crashes and resource leaks if reused to a target program *as-is*.

Certainly, the purpose of Stack Overflow is to provide a starting point for investigation and its code examples do not necessarily include all details of how to reuse the given code. However, for novice developers, it may be useful to show extra tips about desirable API usage evidenced by a large number of GitHub snippets. Based on this insight, we further develop a Chrome extension that warns about API misuse in online code snippets and suggests corresponding fixes in a browser. Our work provides a foundation for enriching and enhancing code snippets in a collaborative Q&A forum by contrasting them against frequent usage patterns learned from massive code corpora.

CHAPTER 6

Visualizing Common and Uncommon API Usage at Scale

The previous chapter focuses on mining frequent API usage patterns from massive code corpora. Given the variety of API usage scenarios in practice, some infrequent API usage may still be semantically correct in specific contexts. It is valuable for developers to explore different API usage in a diverse set of usage scenarios. However, there is no easy way for developers to understand the commonalities and variations in a large number of relevant code examples. In this chapter, we present an interactive visualization for exploring a large collection of code examples mined from open-source repositories without sacrificing the ability to drill down to concrete details.

6.1 Introduction

Learning how to correctly and effectively use existing APIs is a common task—and a core challenge—in software development. The greatest obstacle to learning an API is "*insufficient or inadequate examples.*" [195] Official documentation is typically dominated by textual descriptions and explanations, often lacking concrete code examples that illustrate API usage. Tutorials and blog posts walk developers through simplified code examples but often without demonstrating alternative uses of an API, which programmers frequently desire when learning unfamiliar APIs [50, 63, 195]. Code-sharing sites like GitHub hold the promise of documenting all the common and uncommon ways of using an API in practice, including many alternative usage scenarios that are not typically shown in curated examples. However, given the large amount of code available online, it is challenging for developers to efficiently browse the enormous volume of search results. In practice, programmers often investigate a handful of search results and return to their own code due to limited time and attention [48, 63, 218]. Prior work has shown that individual code examples may suffer from API usage violations [264], insecure coding practices [73], unchecked obsolete usage [268], and comprehension difficulties [235]. Therefore, inspecting a few examples may leave out critical safety checks or desirable usage scenarios.

In the software engineering community, there is a growing interest in leveraging a large collection of open source repositories—so called *Big Code*—to automatically infer API usage patterns from massive corpora [50,170,244,267]. However, these API usage mining techniques provide limited support to help programmers explore concrete code examples from which API usage patterns are inferred, and understand the commonalities and variances across different uses. To bridge the gap, we aim to visualize hundreds of concrete code examples mined from massive code corpora in a way that reveals their commonalities and variances, and design a navigation model to guide the exploration of these examples. We draw motivation from prior work on visualizing large corpora of related documents, e.g., student coding assignments [85], text [209,249], and image manipulation tutorials [174], to pose the following research question: *How might we extract, align, canonicalize, and display large numbers of usage examples for a given API?*

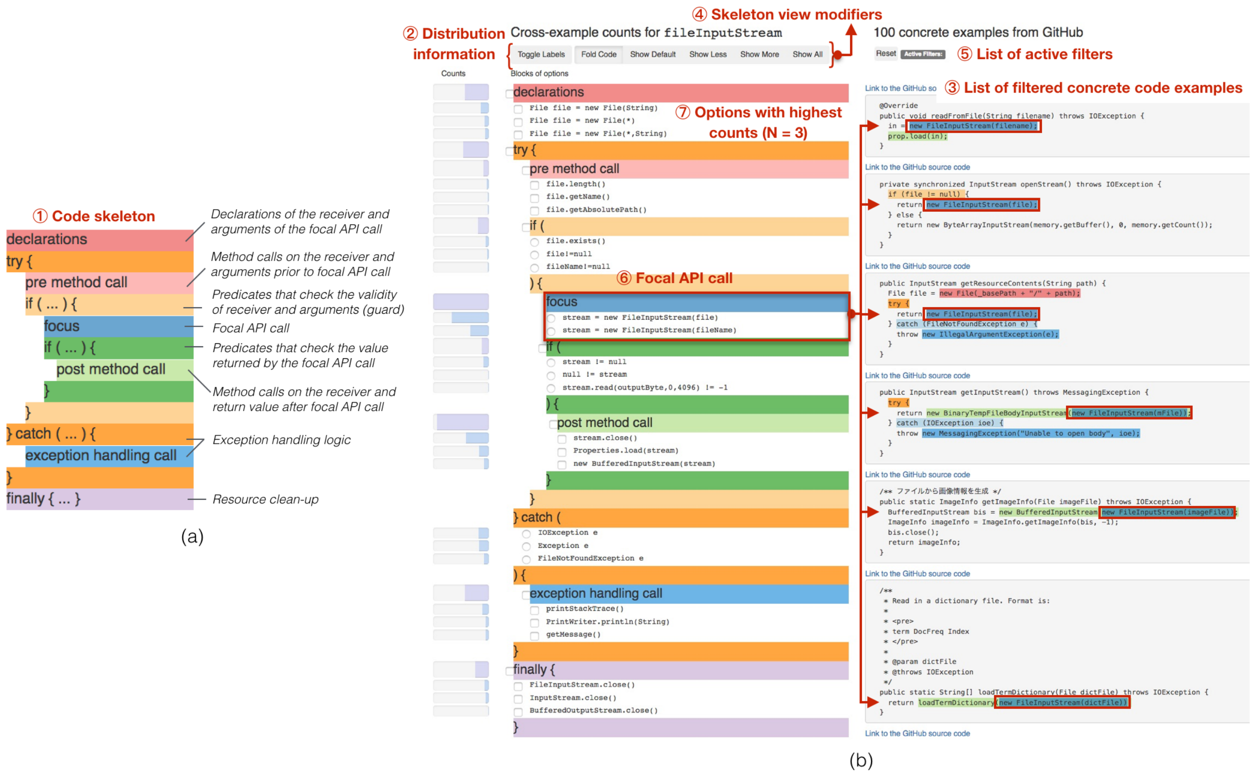
In this chapter, we introduce a novel interactive visualization and navigation technique called Examplore that (1) gives a bird's-eye view of common and uncommon ways in which a community of developers uses an API and (2) allows developers to quickly filter a corpus for concrete code examples that exhibit these various uses. It operates on hundreds of code examples of a given API method, which can be automatically mined from open-source projects or proprietary codebases. It is designed to supplement existing resources: for example, while Stack Overflow can provide explanations and discussions, Examplore provides quantitative information about how an API call is used in the wild.

Examplore instantiates a synthetic code skeleton that captures a variety of API usage features, including initializations, enclosing control structures, guard conditions, and other method calls before and after invoking the given API method, etc. This skeleton is designed to be general: it is grounded in how API design is taught in software engineering curricula and how API mining researchers conceptualize their tasks [63, 127, 195]. Examplore visualizes the statistical distribution of each API usage feature in the skeleton to provide quantitative evidence of each feature in the corpus. The user can select one or more features in the skeleton and, by dynamically filtering mined code examples from the corpus, drill down to concrete, supporting code examples. Color-coordinated highlighting makes it easier for users to recognize the correspondence between features in the skeleton and code segments within each example.

We conducted a within-subjects lab study where we asked sixteen Java programmers of various levels of expertise to answer questions about the usage of particular Java APIs based on either (1) searching online for relevant code examples, blogs, and forum posts or (2) using Examplore to explore one hundred API usage examples mined from GitHub. On average, participants answered significantly more API usage questions correctly, with more concrete details, using Examplore. This suggests that Examplore helps users grasp a more comprehensive view of API usage than online search. In a post survey, the majority of participants (13/16) found Examplore to be more helpful for answering API usage questions than online search, and when using Examplore, their median level of confidence in their answers was higher.

Our contributions are:

- a method for generating an interactive visualization of a distribution of code examples for a given API call
- an implementation of this interactive visualization for a set of Java and Android API calls
- a within-subjects lab study that shows how this interactive visualization may fill an important role in developers' programming workflows as they use unfamiliar APIs.


① Code skeleton
declarations
try {
pre method call
if (...) {
focus
if (...) {
post method call
}
}
} catch (...) {
exception handling call
}
finally { ... }
Declarations of the receiver and arguments of the focal API call
Method calls on the receiver and arguments prior to focal API call
Predicates that check the validity of receiver and arguments (guard)
Focal API call
Predicates that check the value returned by the focal API call
Method calls on the receiver and return value after focal API call
Exception handling logic
Resource clean-up
(a)
② Distribution information
④ Skeleton view modifiers
Toggle Labels
Fold Code
Show Default
Show Less
Show More
Show All
Counts
Blocks of options
⑦ Options with highest counts (N = 3)
⑥ Focal API call
⑤ List of active filters
Reset
Active Filters:
③ List of filtered concrete code examples
(b)

Cross-example counts for `fileInputStream`

```
declarations
  File file = new File(String)
  File file = new File(*)
  File file = new File(*,String)
try {
  pre method call
    file.length()
    file.getName()
    file.getAbsolutePath()
  if (
    file.exists()
    file!=null
    fileName!=null
  ) {
    focus
      stream = new FileInputStream(file)
      stream = new FileInputStream(fileName)
    if (
      stream != null
      null != stream
      stream.read(outputByte,0,4096) != -1
    ) {
      post method call
        stream.close()
        Properties.load(stream)
        new BufferedInputStream(stream)
    }
  }
} catch (
  IOException e
  Exception e
  FileNotFoundException e
) {
  exception handling call
    printStackTrace()
    PrintWriter.println(String)
    getMessage()
}
finally {
  FileInputStream.close()
  InputStream.close()
  BufferedOutputStream.close()
}
```

100 concrete examples from GitHub

Link to the GitHub so

```
@Override
public void readFromFile(String filename) throws IOException {
  in = new FileInputStream(filename);
  prop.load(in);
}
```

Link to the GitHub source code

```
private synchronized InputStream openStream() throws IOException {
  if (file != null) {
    return new FileInputStream(file);
  } else {
    return new ByteArrayInputStream(memory.getBuffer(), 0, memory.getCount());
  }
}
```

Link to the GitHub source code

```
public InputStream getResourceContents(String path) {
  File file = new File(_basePath + "/" + path);
  try {
    return new FileInputStream(file);
  } catch (FileNotFoundException e) {
    throw new IllegalArgumentException(e);
  }
}
```

Link to the GitHub source code

```
public InputStream getInputStream() throws MessagingException {
  try {
    return new BinaryTempFileBodyInputStream(new FileInputStream(mFile));
  } catch (IOException ioe) {
    throw new MessagingException("Unable to open body", ioe);
  }
}
```

Link to the GitHub source code

```
/** ファイルから画像情報を生成 */
public static ImageInfo getImageInfo(File imageFile) throws IOException {
  BufferedInputStream bis = new BufferedInputStream(new FileInputStream(imageFile));
  ImageInfo imageInfo = ImageInfo.getImageInfo(bis, -1);
  bis.close();
  return imageInfo;
}
```

Link to the GitHub source code

```
/**
 * Read in a dictionary file. Format is:
 *
 * <pre>
 * term DocFreq Index
 * </pre>
 *
 * @param dictFile
 * @throws IOException
 */
public static String[] loadTermDictionary(File dictFile) throws IOException {
  return loadTermDictionary(new FileInputStream(dictFile));
}
```

Link to the GitHub source code

Figure 6.1: (a) The layout of our general code skeleton to demonstrate different aspects of API usage. The skeleton structure corresponds to API design practice. (b) A screen snapshot of Examplore and its features, derived from 100 mined code examples for `FileInputStream`. The first six of those 100 code examples are visible on the right.

6.2 Constructing Synthetic API Usage Skeleton

To visualize and navigate a collection of code examples in the order of hundreds or thousands, we introduce the concept of a synthetic code skeleton, which summarizes a variety of API usage features in one view for ease of exploration. Its design is inspired by previous studies on the challenges and obstacles of learning unfamiliar APIs. Duala-Ekoko and Robillard argue that a user must understand *dependent* code segments—object construction, error handling, and interaction with other API methods—related to an API method of interest [63]. Ko et al. found that programmers must be aware of how to use several low-level APIs together (i.e., *a coordination barrier*); how to invoke a specific API method with valid arguments; and how to handle the effects of the method (i.e., *a use barrier*) [127]. Figure 6.1(a) shows the layout of the code skeleton in the Examplore interface.

The skeleton is composed of the following seven API usage features that can co-occur with a common *focal* API method call that is of interest to the user:

1. **Declarations** Prior to calling the focal API method, programmers may construct a receiver object and initialize method arguments.

2. **Pre-focal method calls** Developers may need to configure the program state of the receiver object or arguments by calling other methods before the focal API method call. For example, before calling `Cipher.doFinal` to encrypt or decrypt a message, programmers must call `Cipher.init` to set the operation mode and key. Otherwise, `doFinal` will throw `IllegalStateException`, indicating that the cipher has not been configured.

3. **Guard** Developers often need to check an appropriate guard condition before the focal API call. For example, before calling `Iterator.next`, programmers can check that `Iterator.hasNext` returns `true` to make sure another element exists, before calling `Iterator.next` to retrieve it.

4. **Return value check** Developers often need to read the return value of the focal API method call. For example, `Activity.findViewById(id)` returns `null` if the `id` argument is not valid. For API methods that may return invalid objects or error codes, programmers must check the return value before using it to avoid exceptions.

5. **Post-focal method calls** Developers may make follow-up method calls on the receiver object or the return value after calling the focal API method. For example, after calling `Activity.findViewById` to retrieve a view from an Android application, programmers may commonly call additional methods on the returned view, like `setVisibility` or `setBackground`, to update its rendering.

6. **Exception handling** For API methods that may throw exceptions, programmers may consider which exception types to handle and how these exceptions are handled in a `try-catch` block.

7. **Resource management** Many Java API methods manipulate different types of resources, e.g., files, streams, sockets, and database connections. Such resources must be freed to avoid resource leaks. A common practice in Java is to clean up these resources in a `finally` block to ensure these resources are freed, even in case of errors.

This skeleton design targets API usage in Java. All the components of the skeleton are standard aspects of Java API design and usage known to the software engineering community [63,127,195]. In other words, the skeleton is the reification of domain knowledge among those who design, teach about, and do research on Java APIs.

This skeleton can be generalized to similar languages like C++ and C#. Some components captured by the skeleton, e.g., conditional predicates guarding the execution of an API call, are expected to generalize to many other languages. Additional components may be necessary to capture API usage features in other programming paradigms, e.g., functional programming.

6.3 Usage Scenario: Interacting with Code Distribution

Examplore is designed to help programmers understand the common and uncommon usage patterns of a given API call. Let's consider Victor, a developer who wants to learn how to use FileInputStream objects in Java. Examplore shows one hundred code examples mined from GitHub that include at least one call to construct a FileInputStream object.

The right half of the screen shows all mined examples, sorted from shortest to longest. Victor can quickly pick out the FileInputStream constructor in each example because they are each highlighted with the same blue color as the header of the focal API section of the code skeleton (⑥ in Figure 6.1). Each section of the skeleton has a distinct heading color, which is used to highlight the corresponding concrete code segments in each code example, e.g., initializing declarations in red, guards in light orange. This is designed to reduce the cognitive load of parsing lots of code, and allows Victor to more easily identify the purpose of different portions of code within each example.

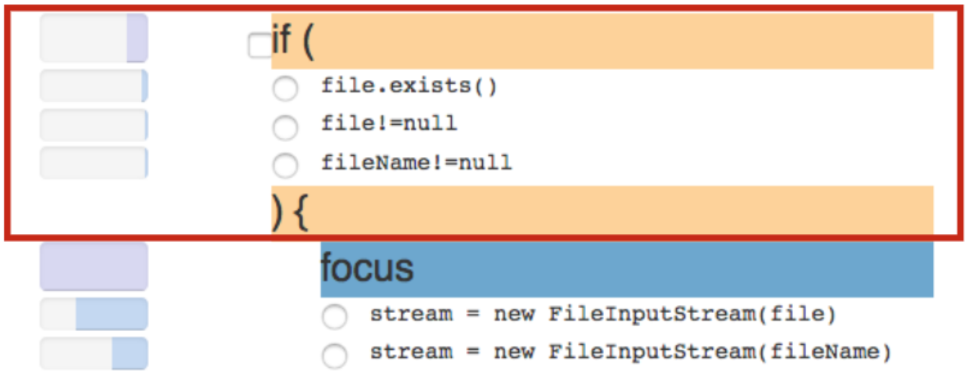
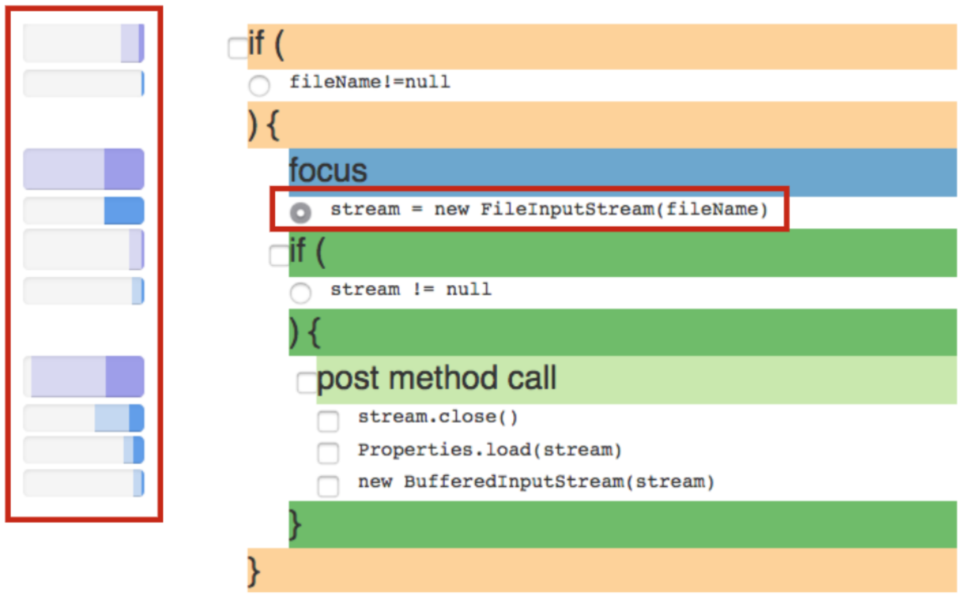

if (
file.exists()
file!=null
fileName!=null
) {
focus
stream = new FileInputStream(file)
stream = new FileInputStream(fileName)

Figure 6.2: As revealed by EXAMPLORE, programmers most often guard this API call by checking first if the argument exists.

EXAMPLORE reveals, by default, the top three most common options for each section of the skeleton (⑦ in Figure 6.1). Victor notices that, based on the relative lengths of the blue bars aligned with each option for calling FileInputStream, passing a File object as the argument is twice as likely as passing `fileName`, a String. By looking at the guard condition options within the `if` section in Figure 6.2, Victor can see how other programmers typically protect `FileInputStream` from receiving an invalid argument. He can also tell, by the small size of the blue bars aligned with these expressions, that these most popular guards are still not used frequently, overall. If he wants to see more or fewer options per skeleton section, he can click the “Show More” or “Show Less” buttons, or explore the long tail of the corpus by clicking “Show All” (④ in Figure 6.1).

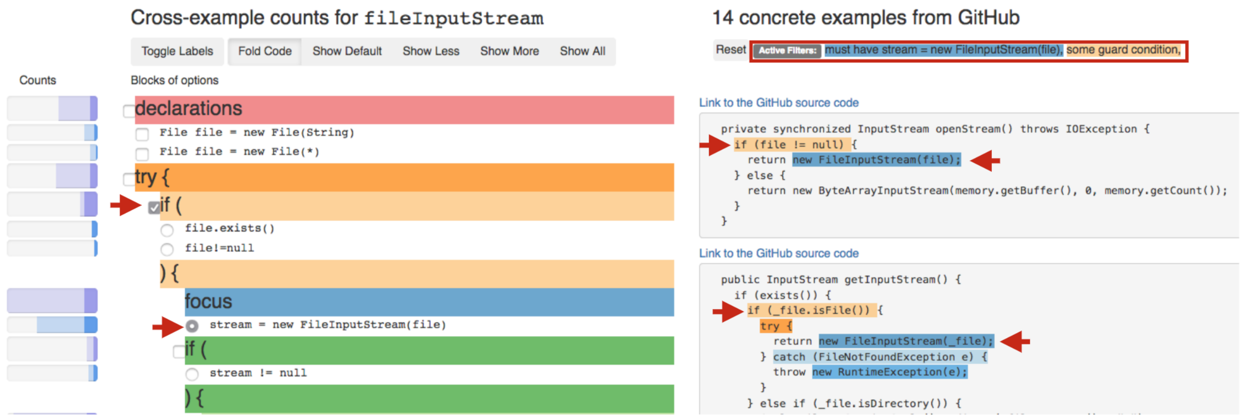
Victor is interested in exploring and better understanding the less common `FileInputStream` constructor, which takes a `String` argument representing a file name. Victor clicks on the radio button next to `stream = new FileInputStream(fileName)`. The active filters (⑤ in Figure 6.1) are updated and the right-hand side of the screen now only lists code examples that construct a `FileInputStream` with a `String`.

Feature options in the skeleton view are pruned and updated based on Victor’s selection (Figure 6.3). Since the selected `FileInputStream` constructor takes a `String` argument instead of a `File` object, the options that declare and initialize the `File` object disappear. The counts of the remaining co-occurring options are affected: the total, unfiltered counts



```
if (
  fileName!=null
) {
  focus
    stream = new FileInputStream(fileName)
  if (
    stream != null
  ) {
    post method call
      stream.close()
      Properties.load(stream)
      new BufferedInputStream(stream)
  }
}
```

Figure 6.3: The bars now show total counts (pastel) and counts conditioned on filtering for the selected option (darker), `stream = new FileInputStream(fileName)`. Options that do not co-occur with the selected option are hidden.



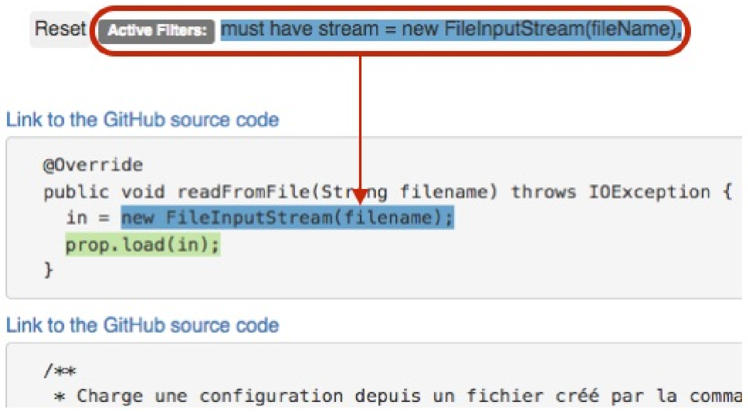
Cross-example counts for fileInputStream
Toggle Labels
Fold Code
Show Default
Show Less
Show More
Show All
Counts
Blocks of options
14 concrete examples from GitHub
Reset
Active Filters:
must have stream = new FileInputStream(file),
some guard condition,
Link to the GitHub source code

```
declarations
  File file = new File(String)
  File file = new File(*)
try {
  if (
    file.exists()
    file!=null
  ) {
    focus
      stream = new FileInputStream(file)
    if (
      stream != null
    ) {
```

```
private synchronized InputStream openStream() throws IOException {
  if (file != null) {
    return new FileInputStream(file);
  } else {
    return new ByteArrayInputStream(memory.getBuffer(), 0, memory.getCount());
  }
}
```

```
public InputStream getInputStream() {
  if (exists()) {
    if (_file.isFile()) {
      try {
        return new FileInputStream(_file);
      } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
      }
    } else if (_file.isDirectory()) {
```

Figure 6.4: A screen snapshot taken while answering the question "What guards do programmers in this dataset use to protect `stream = new FileInputStream(file)`?" The red arrows point to the user's selections and corresponding filtered code elements that answer this question.


Reset
Active Filters:
must have stream = new FileInputStream(fileName),
Link to the GitHub source code
Link to the GitHub source code

```
@Override
public void readFromFile(String filename) throws IOException {
  in = new FileInputStream(filename);
  prop.load(in);
}
```

```
/**
 * Charge une configuration depuis un fichier créé par la comma
```

Figure 6.5: The active filters are updated and the code examples in the list reflect Victor's selection of the skeleton option `stream = new FileInputStream(fileName)`.

shown in pastel bars are unchanged, but darker bars are super-imposed, showing the new counts for the subset of examples in the corpus that construct `FileInputStream` with a `String`.

Victor realizes that there is one place in his project where it will be a hassle to get a file name. He will need to use the other version of `FileInputStream` constructor that takes a `File` object instead. He wonders what guards other programmers use to prevent problems when constructing a `FileInputStream` this way. As shown in Figure 6.4, by clicking on the radio button next to `stream = new FileInputStream(file)` and the check box for the enclosing `if` block, he filters the skeleton options and concrete code examples down to just those with the guards he is interested in. He clicks "Show All" to see all the guard options in the corpus, from the most common guards like `file.exists()` to more unusual guards like `file.isFile()`. He was not aware that the `File` object has an `isFile()` method. He scrolls through a few of the concrete code examples on the right-hand side of the screen to confirm that he understands how these guard conditions are expressed in other programmers' code, and then continues his task of creating well-guarded `FileInputStream` objects in his own code.

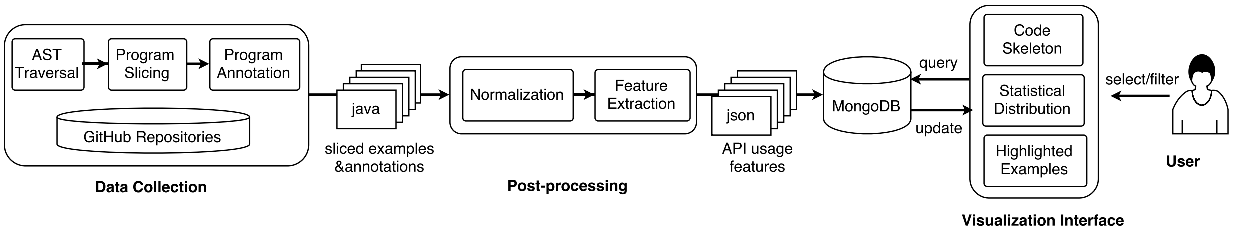


Figure 6.6: Examplore system architecture.

6.4 System Architecture and Implementation

Examplore retrieves and visualizes hundreds of usage examples for a given API call of interest in three phases, shown in Figure 6.6. In the Data Collection phase, Examplore leverages the API usage mining framework in ExampleCheck [264] to crawl 380K GitHub repositories and retrieve a large number of code examples that include at least one call to the API method call of interest. In the Post-processing phase, Examplore analyzes the code examples, labels the segments of code that correspond to each API usage feature in the skeleton, and then extracts and canonicalizes those segments of code to populate the options for each feature in a MongoDB database. In the Visualization phase, Examplore renders the code skeleton, including the canonicalized options for each feature and their distribution in the corpus, and highlights the code segments within each mined code example from which the canonicalized options were extracted. The user interacts with the visualization by selecting features and specific options to filter by.

6.4.1 Data Collection

Here, we briefly summarize the mining process in [?] to describe the format of resulting code example data used in Examplore. Given an API method of interest, the mining process first traverses the abstract syntax trees of Java files and locates all methods invoking the given API method by leveraging ultra-large-scale software repository analysis infrastructure [66]. For each scanned method, the mining technique uses program slicing to remove code statements irrelevant to the API method of interest. For example, when the API method of interest is the constructor of `FileInputStream` on line 12 in Figure 6.7, only underlined statements

```java
 1 @RequestMapping(method = RequestMethod.POST)
 2 public void download(String fName, HttpServletResponse response, HttpSession session) {
 3  if (fName == null) {
 4    log.error("Invalid File Name");
 5    return;
 6  }
 7  String path = session.getServletContext().getRealPath("/")+fName;
 8  response.setContentType("application/stream");
 9  response.setHeader("Content-Disposition", "attachment;filename=" + fName);
10  File file = new File(path);
11  try {
12    FileInputStream in = new FileInputStream(file);
13    ServletOutputStream out = response.getOutputStream();
14    byte[] outputByte = new byte[4096];
15
16    while (in.read(outputByte, 0, 4096) != -1) {
17      out.write(outputByte, 0, 4096);
18    }
19  } catch (FileNotFoundException e) {
20    e.printStackTrace();
21  } catch (IOException e) {
22    e.printStackTrace();
23  } finally {
24    in.close();
25    out.flush();
26    out.close();
27  }
28 }
```

Figure 6.7: This method is extracted as an example of `FileInputStream` from the GitHub corpus. Only the underlined statements and expressions have data dependences on the focal API call to `new FileInputStream` at line 12.

and expressions in lines 10, 16, and 24 are retained, as these have direct data dependences on the focal API call at line 12. In addition to filtering relevant statements based on direct data dependences, EXAMPLORE also identifies enclosing control structures such as `try-catch` blocks and `if` statements relevant to the focal API call. A control structure is related to a given API call if there exists a path between the two and the API call is not post-dominated by the control structure [25]. In Figure 6.7, the API call to `new FileInputStream` (line 12) is related to the enclosing `try-catch-finally` block at lines 11 and 19-27 and the preceding `if` statement at line 3. Such control structure information is used to extract API usage features about guard conditions, return value checks, and exception handling. In each scanned code example, each variable or object name is annotated with its static type information, which EXAMPLORE uses when canonicalizing variable names within the code skeleton.

6.4.2 Post-processing

EXAMPLORE normalizes the retrieved set of code examples into a canonical form so that the user can easily view relevant API usage features without the need to handle different syntactic structures and different concrete variable names. Concrete options for each API usage feature are stored in a MongoDB database so that the front end can construct a database query and update the interface based on user selections.

Normalization of Chained Calls. To help developers easily recognize a sequence of method calls, EXAMPLORE rewrites chained method calls for readability. Specifically, it separates chained method calls to different method calls by introducing temporary variables that store the intermediate results. For example, `new FileInputStream(new File(path)).read(...)` is rewritten to `file = new File(path); fileInputStream = new FileInputStream(file); fileInputStream.read(...);`.

Canonicalizing Variable Names. To reduce the cognitive effort of recognizing semantically similar variables that are named differently in different examples, EXAMPLORE renames the arguments of the focal API call based on the corresponding parameter names declared in the official Javadoc documentation so that all variable names follow the same naming convention.

The rest of the variables are renamed based on their static types. For example, if the type of the receiver object is `File`, we rename its object name to be `file`, the lower CamelCase of the receiver type.

Consider the example in Figure 6.7 where the constructor `FileInputStream(File)` is the focal API call. The following list describes the concrete code segments corresponding to different API usage features:

- Declarations: `File file = new File(path)` at line 10.
- Pre-focal method calls: none.
- Guard: the negation of `fName == null` at line 3.
- Return value check: none.
- Post-focal method calls: `in.read(outputByte,0,4096)!=-1` at line 16.
- Exception handling: `e.printStackTrace()` for handling `FileNotFoundException` at lines 19-20 and `IOException` at lines 21-22.
- Resource management: `in.close()` at line 24.

6.4.3 Visualization

For each API usage feature, Examplore records the start and end character indices of the corresponding code for color highlighting. Examplore queries the MongoDB database and instantiates the synthetic code skeleton with canonicalized options extracted from GitHub code examples and distributions of counts accumulated across the corpus. When a user selects particular options in the skeleton, the front end queries MongoDB and updates the interface accordingly.

6.5 User Study

We conducted a within-subjects study with sixteen Java programmers to evaluate whether participants could grasp a more comprehensive view of API usage using Examplore, in comparison to a realistic baseline of searching online for code examples, which is commonly used in real-world programming workflows [48, 202]. We designed a set of API usage questions, shown in Table 6.1, to assess how much knowledge about API usage participants could extract from Examplore or online search for a given API method. Questions Q1-7 were derived from the commonly asked API usage questions identified in prior work [63]. Q8 asked participants to inspect and critique a curated code example from Stack Overflow. This question was designed to evaluate whether users were capable of making comprehensive judgments about the quality of a given code example after exploring a large number of examples using Examplore, inspired by Brandt et al.'s observation that programmers typically opened several programming tutorials in different browser tabs and judged their quality by rapidly skimming [48].

6.5.1 API Methods

Programmers often behave differently when searching online to learn a new concept compared to when they are reminding themselves about the details in a familiar concept [48]. Similarly, we anticipated that programmers might apply different exploration strategies when answering API usage questions about familiar and unfamiliar APIs. To capture a spectrum of behaviors, we chose three API methods with which programmers might have varying levels of familiarity:

1. `Map.get` is a commonly used Java method that retrieves the value of a given key from a data structure that stores data as key and value pairs.

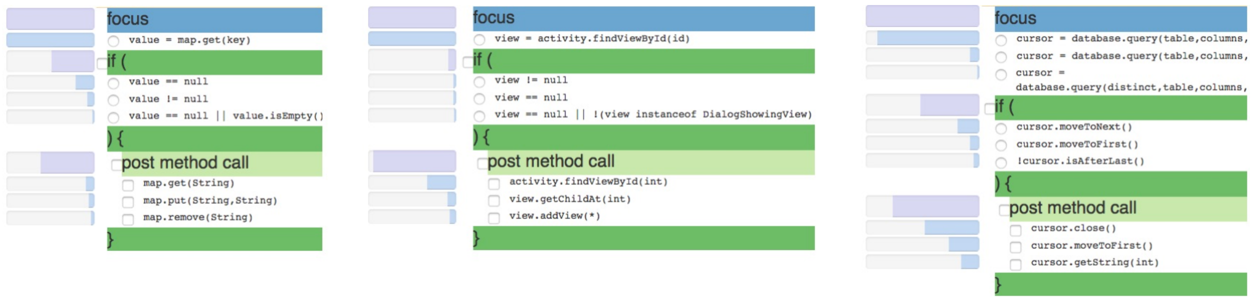
2. `Activity.findViewById` is an Android method that gets a specific view (e.g., button, text area) from an Android application.

3. `SQLiteDatabase.query` is a database query method that constructs a SQL command from the given parameters and queries a database.

API Usage Questions
Q1. How do I create or initialize the receiver object so I can call this API method? Describe multiple ways, if possible.
Q2. How do I create or initialize the arguments so I can call this API method? Describe multiple ways, if possible.
Q3. What other API methods, if any, would be reasonable to call before calling this API method?
Q4. What, if anything, would be reasonable to check before calling this API method?
Q5. What, if anything, would be reasonable to check after calling this API method?
Q6. How do programmers handle the return value of this API method?
Q7. What are the exceptions that programmers catch and how do programmers handle potential exceptions? Please indicate none if this API method does not throw any exception.
Q8. How might you modify this code example on Stack Overflow if you were going to copy and paste it into your own solution to the original prompt?

Table 6.1: Study task questions for participants to answer for each assigned API method. Q1-7 are derived from commonly asked API usage questions identified by [63]. Q8 prompts the participant to critique a curated code example from Stack Overflow.

Figure 6.8 shows cropped screenshots of how Examplore rendered each of these APIs.



```
focus
value = map.get(key)
if (
value == null
value != null
value == null || value.isEmpty()
) {
post method call
map.get(String)
map.put(String,String)
map.remove(String)
}
```

(a)

```
focus
view = activity.findViewById(id)
if (
view != null
view == null
view == null || !(view instanceof DialogShowingView)
) {
post method call
activity.findViewById(int)
view.getChildAt(int)
view.addView(*)
}
```

(b)

```
focus
cursor = database.query(table,columns,
cursor = database.query(table,columns,
cursor =
database.query(distinct,table,columns,
if (
cursor.moveToNext()
cursor.moveToFirst()
!cursor.isAfterLast()
) {
post method call
cursor.close()
cursor.moveToFirst()
cursor.getString(int)
}
```

(c)

Figure 6.8: Cropped screenshots of how Examplore renders each of the three APIs included in the study: (a) `Map.get` (b) `Activity.findViewById` (c) `SQLiteDatabase.query`.

6.5.2 Participants

We recruited sixteen Computer Science students from UC Berkeley through the EECS department mailing list. Eleven participants (69%) were undergraduate students and the other five (31%) were graduate students. Since our study task required participants to read code examples in Java and answer questions about Java APIs, we only included students who had taken at least one Java class. Participants had a diverse background in Java programming, including one participant with one semester of Java programming, four with one year, ten with two to five years, and one with over five years. Two students were teaching assistants for an object-oriented programming language course. Prior to the study, twelve participants (75%) had used `Map` or similar data structures, six (38%) had used `SQLiteDatabase.query` or similar database query methods, and only three (19%) had used `Activity.findViewById`.

6.5.3 Methodology

We conducted a 50-min user study with each participant. Note that our study follows a within-subjects design and both the order of the assigned conditions (using online search or Examplore to answer API usage questions) and which of the three API methods were assigned in each condition (`Map.get`, `Activity.findViewById`, or `SQLiteDatabase.query`) were counterbalanced across participants through random assignment.

1. **Training session (15 min)** We first walked the participant through a short list of relevant Java concepts and terminology, such as receiver objects and guards. Then we walked participants through each user interface feature and answered participants' questions about both the concepts and the interface.
2. **Code exploration task 1 (15 min)** The participant was given basic information about one of the three API methods and asked to answer API usage questions Q1-8 by exploring code examples using the assigned tool, either online search or Examplore.
3. **Code exploration task 2 (15 min)** The participant was given basic information about another one of the three API methods and asked to answer API usage questions

Q1-8 by exploring code examples using the tool (Examplore or online search) that they did not use in the previous task.

4. **Post survey (5 min)** At the end of the session, participants answered questions about their experience using each tool and the usability of individual user interface features in Examplore.

In the control condition, participants were allowed to search for code examples in any online learning resources, e.g., documentations, tutorial blogs, Q&A forums, and GitHub repositories, using any search engines in a web browser. In the experimental condition, participants used Examplore to explore one hundred code examples that were pre-loaded into the system.

Some of the API usage questions have multiple possible correct answers. Before each code exploration task, we reminded participants that they had 15 minutes to complete the API usage questions and that they should aim for thoroughness (i.e., list multiple correct answers if they exist) instead of speed when answering these questions.

6.6 Results

6.6.1 Quantitative Analysis

6.6.1.1 Answering Commonly Asked API Usage Questions

We manually assessed the participants' answers to Q1-8. An answer was considered concrete if it contained a code segment, e.g., `map.containsKey(key)`, or it was specific, e.g., "check whether the key exists." As a counter example, a vague answer to the question about how programmers handle the return value of `Map.get` (Q6) was, "[other programmers] do something with the return value [of `Map.get`]." We considered a concrete solution to be correct if it could be confirmed by the official documentation, blogs, or concrete code examples.

Table 6.2 shows statistics about participants' correct answers to Q1-7 when using online search or the Examplore tool. We find that the effects of using Examplore are both

	`Map`		`Activity`		`SQLiteDatabase`		Overall	
	Tool	Search	Tool	Search	Tool	Search	Tool	Search
Ave. # of Q's answered correctly	5.0	6.0	6.3	5.0	6.6	3.8	6.0	4.6
Ave. total # of correct answers	8.2	6.0	12.5	4.7	14.6	5.4	11.8	5.7
Ave. # of correct answers per Q	1.6	1.2	2.0	1.1	2.2	1.4	1.8	1.2

Table 6.2: Statistics about participants' correct answers to Q1-7. **Search** refers to participants in the control condition, and **Tool** refers to those using Examplore.

meaningful in size and statistically significant: Users gave, on average, correct answers to 6 out of 7 API usage questions using Examplore vs. 4.6 questions using the baseline of online search. This mean difference of 1.3 questions out of 7 is statistically significant (paired t-test: t=3.02, df=15, p-value=0.0086).

Screencasts of the user study sessions reveal that participants in the control condition often answered API usage questions just based on one example they found or by guessing. In contrast, Examplore users interacted with the code skeleton and investigated many individual examples that were relevant to the question. This may explain why, in Table 6.2, Examplore users gave, on average, twice as many correct answers to Q1-7 as baseline users (11.8 vs. 5.7, paired t-test: t=3.84, df=15, p-value=0.0016).

Participants using online search provided almost twice as many vague answers as participants using Examplore. When answering Q6 (*How do programmers handle the return value of this API method?*), two participants using online search were unable to find any examples that check the return value of `Activity.findViewById`, while all participants gave the correct answer using Examplore.

6.6.1.2 Critiquing Stack Overflow Answers

Q8 asked participants to critique a code example from Stack Overflow based on other relevant code examples they explored in the study. Regardless of whether participants had just used Examplore or online search, fourteen participants (88%) gave valid suggestions to improve the Stack Overflow posts. The majority of critiques (80%) written by participants using Examplore were about safety checks, e.g., how to handle potential exceptions in a `try-catch` block. When using online search, the majority of participants (57%) suggested how to customize and style the code example for better readability, e.g., adapting types and parameters when reused to a new client program, renaming variables, and indenting code.

6.6.1.3 Post Survey Responses

In the post survey, 13 participants (81%) found Examplore to be more helpful for answering API usage questions than online search. The distribution of their responses on a 7-point scale is shown in Figure 6.9. The median level of confidence that participants had in their answers was higher when using Examplore (5 vs. 4 on a 7-point scale, shown in Figure 6.10). Figure 6.11 suggests that Examplore's representation of the commonalities and differences across 100 code examples is more helpful than overwhelming (5 vs. 3.5 on a 7-point scale).

One source of participants' accuracy, thoroughness, and confidence when using Examplore appears to be the data itself, presented in structured form: P16 wrote, "[Examplore] provided structure to learning about API. This structure guides functionality while still showing variety of use. The frequency of [each option] shows me if I am looking at a random corner case or something commonly used." However, explanations in natural language are still valued. For example, two participants requested textual explanations alongside concrete code examples. P7 stated that, "although I definitely took longer with the online search, I felt more confident in knowing what I was doing because I had access to Stack Overflow explanations."

6.6.2 Qualitative Analysis

We coded participants' free responses in the post survey for common recurring patterns. By far the most popular interface feature named in their free responses (13/16) was the ability to filter for specific API usage aspects of code examples, e.g., declarations, guards, and co-occurring API calls. The second most popular feature (4/16) was the ability to explore many examples simultaneously in a summarized form. The long tail of responses included appreciation for the ease of finding relevant examples (3/16), the use of color to label different parts of each code example (2/16), being able to perceive and retrieve a variety of examples within a skeleton (2/16), which also gave structure to learning (2/16) and counts to indicate common practices (1/16).

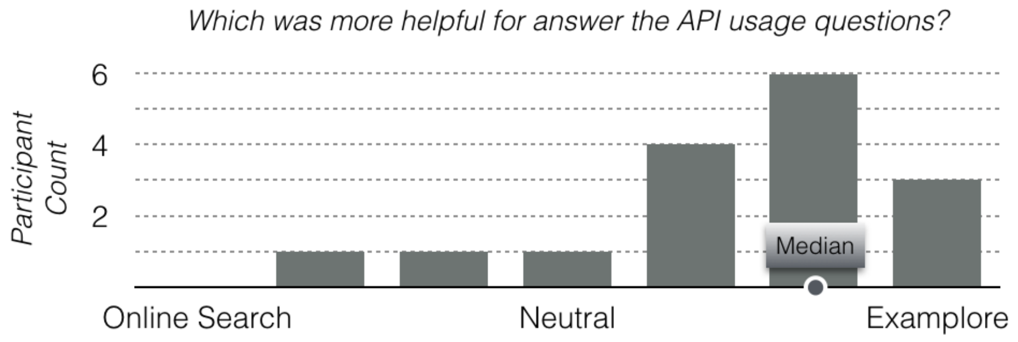


Figure 6.9: The majority of participants found EXAMPLORE more helpful for answering API usage questions.

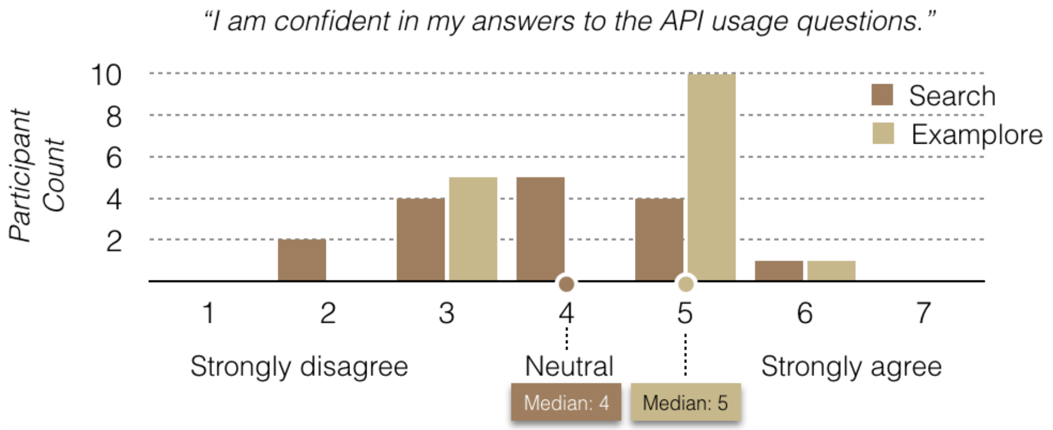


Figure 6.10: When using EXAMPLORE, participants had more confidence in their answers to API usage questions.

Several critical aspects of EXAMPLORE were highlighted by their absence in the control condition, i.e., online search. Nearly half (7/16) wrote that they wished traditional search had better filtering mechanisms, like EXAMPLORE provided, so that participants could retrieve more consistently relevant results and/or filter on a more fine-grained basis. A quarter of participants (4/16) complained that they had to mentally parse code examples from the

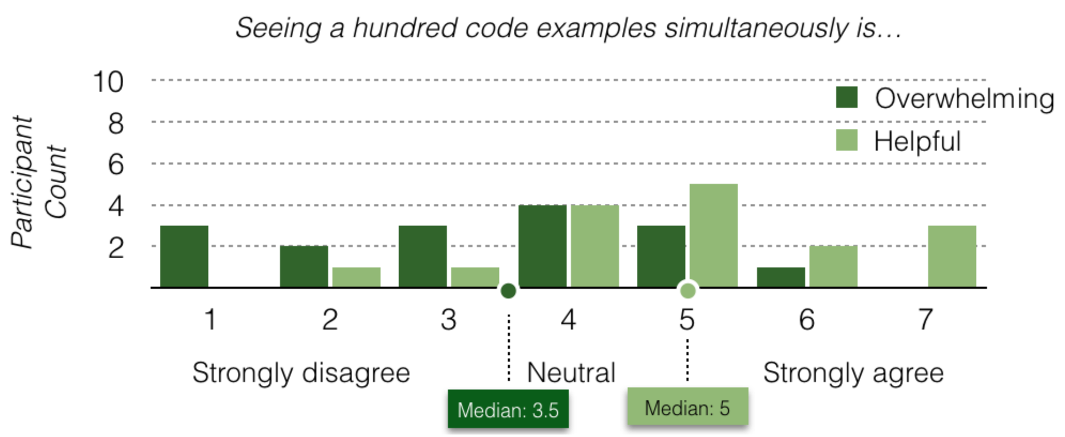


Figure 6.11: Participants' median level of agreement with the statement that Examplore's high-level view of API usage examples was helpful was higher than their median level of agreement with a statement that it was overwhelming.

on-line search results. Three participants complained that they cannot easily assess how common and uncommon the code examples found through Google or GitHub searches are: P3 wrote, "One thing that is important is 'best practice' which you might not get from reading random code online, so if I had a way to know what is common and uncommon, that would be useful." One participant pointed out that Google and GitHub searches did not make it easy to view multiple examples at once: while it was relatively easy to spot the use of the API call of interest in each code example, "it was hard to find the specific instances of API usage categories other than the Focus because the examples would use different names for different variables."

Participants did point out several areas where the interface could be improved. Half of the participants stated that the interface was confusing and hard to learn. Three of the sixteen participants felt confused or distracted by the many colors used to highlight different parts of the code examples that corresponded with the skeleton. Participants wished for not just filtering but search capabilities in the interface, and for textual explanation to be paired with the code, like the curated and explained examples in many online search results. Two participants asked for a more explicit indicator of code example quality, beyond frequency counts.

The final question of the post survey asked participants to write about how Examplore could fit into their programming workflows. Without any prior questions or prompting about API learning, nearly half the participants (7/16) wrote that they would use Examplore to explore and learn how to use an unfamiliar API. For example, P4 wrote "[U]sing [Examplore] to search for usage of unfamiliar methods could be very helpful." One quarter of the participants mentioned Examplore would be helpful to augment the code browsing mechanism in Q&A sites like Stack Overflow. Two participants wrote specifically about using Examplore to learn about design alternatives for an API, regardless of their prior familiarity with it. Two participants mentioned that they could consult it for certain specific questions, e.g., what exceptions are commonly caught. Finally, one participant pointed out that they could use Examplore to remind them of uncommon usage cases. Several participants asked the experimenter, after submitting their post survey answers, whether Examplore would be made publicly available, expressing a sincere desire to use it in the future.

6.7 Threats to Validity

Our study explored if Examplore can help users explore and understand how an API method is used. The results show that, when using Examplore instead of online search engines, users can answer more API usage questions correctly, with more confidence, concrete details, and alternative correct answers.

The Examplore interface appears to be most beneficial when learning and exploring unfamiliar APIs. Participants expressed, in free-response survey answers, the desire to use Examplore to explore unfamiliar APIs in the future. Also, the benefits of using Examplore described in Table 6.2 are most pronounced for the APIs that participants were, in aggregate, least familiar with: `Activity.findViewById` and `SQLiteDatabase.query`. In contrast, for the API that most participants were already familiar with, `Map.get`, participants answered one less question correctly on average, compared to online search. Existing online search provided a familiar and flexible search interface as well as the ability to access learning resources with textual explanations such as Stack Overflow posts, documentation, and blog posts. Even so, participants still provided two more concrete solutions on average, when using Examplore for `Map.get`, indicating that Examplore can still be helpful to provide a more comprehensive view of API usage even for familiar APIs.

The study results suggest that programmers can develop a more comprehensive understanding of API usage by exploring a large collection of code examples visualized using Examplore than by searching for relevant examples online. However, there is a trade-off between the Examplore interface's expressive power and its visual complexity. We have a lot of information about how APIs are used, but showing all of it at once can be overwhelming. More research is needed in making sure the most common use cases are answered in a visually simple and easy-to-interpret manner, while still supporting more complex investigations. This could, for example, be achieved through progressive disclosure or other UI design patterns.

Examplore does not require all mined code examples to be bug-free. We expect that inadequate examples occur less frequently in the majority of mined code, i.e., the "wisdom of the crowd," but we do not currently guard against stale examples or low-quality examples. Possible ways of detecting stale examples in the future include analyzing metadata and scanning for outdated method signatures. Even if all examples in the corpus are of equally high quality, sorting the concrete code examples by length, as the interface currently does, is not necessarily what programmers want. Alternative sorting criteria could include metrics like GitHub stars, number of contributors, and build status. We will surface these signals in the future user interface so users have additional information scent when judging quality.

6.8 Conclusion

This chapter presents Examplore, an interactive visualization for exploring a large collection of code examples mined from open-source repositories at scale. Examplore is the first approach that is capable of visualizing hundreds of relevant code examples in a single view without sacrificing the ability to drill down to concrete details. The key enabler of Examplore is a synthetic API usage skeleton that demonstrates distinct API usage features with statistical distributions for canonicalized statements and structures enclosing an API call.

We implement this interactive visualization for a set of Java APIs and find that, in a lab study, Examplore helps users answer significantly more API usage questions correctly and comprehensively, and explore how other programmers have used an unfamiliar API.

CHAPTER 7

Analyzing and Supporting Adaptation of Online Code Examples

The API misuse study in Chapter 5 implies that online code examples are often incomplete and inadequate for developers' local program contexts. Adaptation of these examples is necessary to integrate them to production code. As a consequence, the process of adapting online code examples is done over and over again, by multiple developers independently. This chapter presents a large-scale empirical study about the nature and extent of adaptations and variations of code snippets in Stack Overflow, serving as the basis for a tool that helps integrate these online code examples in a target context in an interactive manner.

7.1 Introduction

Nowadays, a common way of quickly accomplishing programming tasks is to search and reuse code examples in online Q&A forums such as Stack Overflow (SO) [48, 81, 238]. A case study at Google shows that developers issue an average of twelve code search queries per weekday [202]. As of July 2018, Stack Overflow has accumulated 26M answers to 16M programming questions. Copying code examples from Stack Overflow is common [39] and adapting them to fit a target program is recognized as a top barrier when reusing code from Stack Overflow [255]. SO examples are created for illustration purposes, which can serve as a good starting point. However, these examples may be insufficient to be ported to a production environment, as previous studies find that SO examples may suffer from API usage violations [264], insecure coding practices [73], unchecked obsolete usage [268], and incomplete code fragments [235]. Hence, developers may have to manually adapt code examples when importing them into their own projects.

Our goal is to investigate the common adaptation types and their frequencies in online code examples, such as those found in Stack Overflow, which are used by a large number of software developers around the world. To study how they are adopted and adapted in real projects, we contrast them against similar code fragments in GitHub projects. The insights gained from this study could inform the design of tools for helping developers adapt code snippets they find in Q&A sites. In this chapter, we describe one such tool we developed, ExampleStack, which works as a Chrome extension.

In broad strokes, the design and main results of our study are as follows. We link SO examples to GitHub counterparts using multiple complementary filters. First, we quality-control GitHub data by removing forked projects and selecting projects with at least five stars. Second, we perform clone detection [206] between 312K SO posts and 51K non-forked GitHub projects to ensure that SO examples are similar to GitHub counterparts. Third, we perform timestamp analysis to ensure that GitHub counterparts are created later than the SO examples. Fourth, we look for explicit URL references from GitHub counterparts to SO examples by matching the post ID. As the result, we construct a comprehensive dataset of *variations* and *adaptations*.

When we use all four filters above, we find only 629 SO examples with GitHub counterparts. Recent studies find that very few developers explicitly attribute to the original SO post when reusing code from Stack Overflow [28,39,255]. Therefore, we use this resulting set of 629 SO examples as an *under-approximation* of SO code reuse and call it an *adaptations* dataset. If we apply only the first three filters above, we find 14,124 SO examples with GitHub counterparts that represent potential code reuse from SO to GitHub. While this set does not necessarily imply any causality or intentional code reuse, it still demonstrates the kinds of common variations between SO examples and their GitHub counterparts, which developers might want to consider during code reuse. Therefore, we consider this second dataset as an *over-approximation* of SO code reuse, and call it simply a *variations* dataset.

We randomly select 200 clone pairs from each dataset and manually examine the program differences between SO examples and their GitHub counterparts. Based on the manual inspection insights, we construct an adaptation taxonomy with 6 high-level categories and 24 specialized types. We then develop an automated adaptation analysis technique built on top of GumTree [72] to categorize syntactic program differences into different adaptation types. The precision and recall of this technique are 98% and 96% respectively. This technique allows us to quantify the extent of common adaptations and variations in each dataset. The analysis shows that both the adaptations and variations between SO examples and their GitHub counterparts are prevalent and non-trivial. It also highlights several adaptation types such as type conversion, handling potential exceptions, and adding `if` checks, which are frequently performed yet not automated by existing code integration techniques [54, 253].

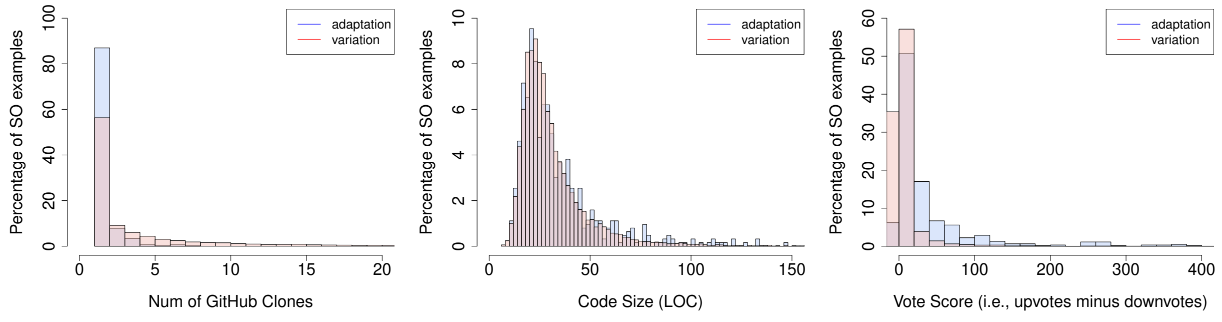
Building on this adaptation analysis technique, we develop a Chrome extension called ExampleStack to guide developers in adapting and customizing online code examples to their own contexts. For a given SO example, ExampleStack shows a list of similar code snippets in GitHub and also lifts an adaptation-aware template from those snippets by identifying common, unchanged code, and also the hot spots where most changes happen. Developers can interact and customize these lifted templates by selecting desired options to fill in the hot spots. We conduct a user study with sixteen developers to investigate whether ExampleStack inspires them with new adaptations that they may otherwise ignore during code reuse. Our key finding is that participants using ExampleStack focus more on adaptations about code safety (e.g., adding an if check) and logic customization, while participants without ExampleStack make more shallow adaptations such as variable renaming. In the post survey, participants find ExampleStack help them easily reach consensus on how to reuse a code example, by seeing the commonalities and variations between the example and its GitHub counterparts. Participants also feel more confident after seeing how other GitHub developers use similar code in different contexts, which one participant describes as "*asynchronous pair programming.*"

The rest of the chapter is organized as follows. Section 7.2 describes the data collection pipeline and compares the characteristics of the two datasets. Section 7.3 describes the adaptation taxonomy development and an automated adaptation analysis technique.

Section 7.4 describes the quantitative analysis of adaptations and variations. Section 7.5 explains the design and implementation of ExampleStack. Section 7.6 describes a user study that evaluates the usefulness of ExampleStack. Section 7.7 discusses threats to validity, and Section 7.8 concludes the paper.

7.2 Data Collection: Linking Stack Overflow to GitHub

This section describes the data collection pipeline. Due to the large portion of unattributed SO examples in GitHub [28, 39, 255], it is challenging to construct a complete set of reused code from SO to GitHub. To overcome this limitation, we apply four quality-control filters to *underapproximate* and *overapproximate* code examples reused from SO to GitHub, resulting in two complementary datasets.



(a) The number of clones (b) The code size (c) The vote score

Figure 7.1: Comparison between SO examples in the adaptation dataset and the variation dataset

GitHub project selection and deduplication. Since GitHub has many toy projects that do not adequately reflect software engineering practices [120], we only consider GitHub projects that have at least five stars. To account for internal duplication in GitHub [146], we choose non-fork projects only and further remove duplicated GitHub files using the same file hashing method as in [146], since such file duplication may skew our analysis. As a result, we download 50,826 non-forked Java repositories with at least five stars from GitTorrent [89]. After deduplication, 5,825,727 distinct Java files remain.

Detecting GitHub candidates for SO snippets. From the SO dump taken in October

2016 [19], we extract 312,219 answer posts that have `java` or `android` tags and also contain code snippets in the `<code>` markdown. We consider code snippets in answer posts only, since snippets in question posts are rarely used as examples. Then we use a token-based clone detector, SourcererCC (SCC) [206] to find similar code between 5.8M distinct Java files and 312K SO posts. We choose SCC because it has high precision and recall and also scales to a large code corpus. Since SO snippets are often free-standing statements [223, 259], we parse and tokenize them using a customized Java parser [224]. Prior work finds that larger SO snippets have more meaningful clones in GitHub [260]. Hence, we choose to study SO examples with no less than 50 tokens, not including code comments, Java keywords, and delimiters. We set the similarity threshold to 70% since it yields the best precision and recall on multiple clone benchmarks [206]. We cannot set it to 100% since SCC will then only retain exact copies and exclude those adapted code. We run SCC on a server machine with 116 cores and 256G RAM. It takes 24 hours to complete, resulting in 21,207 SO methods that have one or more similar code fragments (i.e., clones) in GitHub.

Timestamp analysis. If the GitHub clone of a SO example is created before the SO post, we consider it unlikely to be reused from SO and remove it from our dataset. To identify the creation date of a GitHub clone, we write a script to retrieve the Git commit history of the file and match the clone snippet against each file revision. We use the timestamp of the earliest matched file revision as the creation time of a GitHub clone. As a result, 7,083 SO examples (33%) are excluded since all their GitHub clones are committed before the SO posts.

Scanning explicitly attributed SO examples. Despite the large portion of unattributed SO examples, it is certainly possible to scan GitHub clones for explicit references such as SO links in code comments to confirm whether a clone is copied from SO. If the SO link in a GitHub clone points to a question post instead of an answer post, we check whether the corresponding SO example is from any of its answer posts by matching the post ID. We find 629 explicitly referenced SO examples.

Overapproximating and underapproximating reused code. We use the set of 629 explicitly attributed SO examples as an *underapproximation* of reused code from SO to

GitHub, which we call an *adaptation* dataset. We consider the 14,124 SO examples after timestamp analysis as an *overapproximation* of potentially reused code, which we call a *variation* dataset. Figure 7.1 compares the characteristics of these two datasets of SO examples in terms of the number of GitHub clones, code size, and vote score (i.e., upvotes minus downvotes). Since developers do not often attribute SO code examples, explicitly referenced SO examples have a median of one GitHub clone only, while SO examples have a median of two clones in the variation dataset. Both sets of SO examples have similar length, 26 vs. 25 lines of code in median. However, SO examples from the adaptation dataset have significantly more upvotes than the variation dataset: 16 vs. 1 in median. In the following sections, we inspect, analyze, and quantify the adaptations and variations evidenced by both datasets.

7.3 Adaptation Type Analysis

7.3.1 Manual Inspection and Adaptation Taxonomy

To get insights into adaptations and variations of SO examples, we randomly sample SO examples and their GitHub counterparts from each dataset and inspect their program differences using GumTree [72]. Below, we use "adaptations" to refer both adaptations and variations for simplicity.

The first and the last authors jointly labeled these SO examples with adaptation descriptions and grouped the edits with similar descriptions to identify common adaptation types. We initially inspected 90 samples from each dataset and had already observed convergent adaptation types. We continued to inspect more and stopped after inspecting 200 samples from each dataset, since the list of adaptation types was converging. This is a typical procedure in qualitative analysis [45]. The two authors then discussed with the other authors to refine the adaptation types. Finally, we built a taxonomy of 24 adaptation types in 6 high-level categories, as shown in Table 7.1.

Code Hardening. This category includes four adaptation types that strengthen SO examples in a target project. *Insert a conditional* adds an `if` statement that checks for corner

Table 7.1: Common adaptation types, categorization, and implementation

Category	Adaptation Type	Rule
Code Hardening	Add a conditional	Insert(t_1, t_2, i) ∧ NodeType(t_1, IfStatement)
	Insert a final modifier	Insert(t_1, t_2, i) ∧ NodeType(t_1, Modifier) ∧ NodeValue(t_1, final)
	Handle a new exception type	Exception(e, GH) ∧ ¬Exception(e, SO)
	Clean up unmanaged resources (e.g. close a stream)	(LocalCall(m, GH) ∨ InstanceCall(m, GH)) ∧ ¬LocalCall(m, SO) ∧ ¬InstanceCall(m, SO) ∧ isCleanMethod(m)
Resolve Compilation Errors	Declare an undeclared variable	Insert(t_1, t_2, i) ∧ NodeType(t_1, VariableDeclaration) ∧ NodeValue(t_1, v) ∧ Use(v, SO) ∧ ¬Def(v, SO)
	Specify a target of method invocation	InstanceCall(m, GH) ∧ LocalCall(m, SO)
	Remove undeclared variables or local method calls	(Use(v, SO) ∧ ¬Def(v, SO) ∧ ¬Use(v, GH)) ∨ (LocalCall(m, SO) ∧ ¬LocalCall(m, GH) ∧ ¬InstanceCall(m, GH))
Exception Handling	Insert/delete a try-catch block	(Insert(t_1, t_2, i) ∨ Delete(t_1)) ∧ NodeType(t_1, TryStatement)
	Insert/delete a thrown exception in a method header	Changed(t_1) ∧ NodeType(t_1, Type) ∧ Parent(t_2, t_1) ∧ NodeType(t_2, MethodDeclaration) ∧ NodeValue(t_1, t) ∧ isExceptionType(t)
	Update the exception type	Update(t_1, t_2) ∧ NodeType(t_1, SimpleType) ∧ NodeType(t_2, SimpleType) ∧ NodeValue(t_1, v_1) ∧ isExceptionType(v_1) ∧ NodeValue(t_2, v_2) ∧ isExceptionType(v_2)
	Change statements in a catch block	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, CatchClause)
	Change statements in a finally block	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, FinallyBlock)
Logic Customization	Change a method call	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, MethodInvocation)
	Update a constant value	Update(t_1, t_2) ∧ NodeType(t_1, Literal) ∧ NodeType(t_2, Literal)
	Change a conditional expression	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ (NodeType(t_2, IfCondition) ∨ NodeType(t_2, LoopCondition) ∨ NodeType(t_2, SwitchCase))
	Change the type of a variable	Update(t_1, t_2) ∧ NodeType(t_1, Type) ∧ NodeType(t_2, Type)
Refactoring	Rename a variable/field/method	Update(t_1, t_2) ∧ NodeType(t_1, Name)
	Replace hardcoded constant values with variables	Delete(t_1) ∧ NodeType(t_1, Literal) ∧Insert(t_1, t_2, i) ∧ NodeType(t_1, Name) ∧ Match(t_1, t_2)
	Inline a field	Delete(t_1) ∧ NodeType(t_1, Name) ∧Insert(t_1, t_2, i) ∧ NodeType(t_1, Literal) ∧ Match(t_1, t_2)
Miscellaneous	Change access modifiers	Changed(t_1) ∧ NodeType(t_1, Modifier) ∧ NodeValue(t_1, v) ∧ v ∈ {private, public, protected, static}
	Change a log/print statement	Changed(t_1) ∧ NodeType(t_1, MethodInvocation) ∧ NodeValue(t_1, m) ∧ isLogMethod(m)
	Style reformatting (i.e., inserting/deleting curly braces)	Changed(t_1) ∧ NodeType(t_1, Block) ∧ Parent(t_2, t_1) ∧ ¬Changed(t_2) ∧ Child(t_3, t_1) ∧ ¬Changed(t_3)
	Change Java annotations	Changed(t_1) ∧ NodeType(t_1, Annotation)
	Change code comments	Changed(t_1) ∧ NodeType(t_1, Comment)

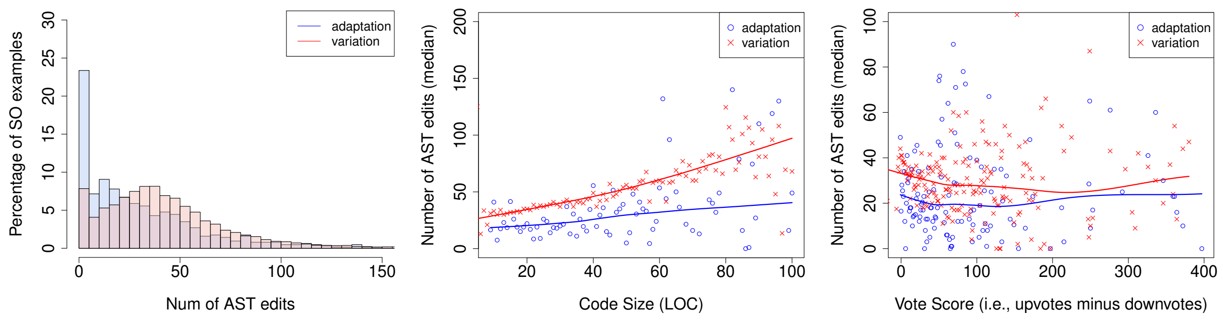
GumTree Edit Operation	Syntactic Predicate	Semantic Predicate
Insert(t_1, t_2, i) inserts a new tree node t_1 as the i-th child of t_2 in the AST of a GitHub snippet.	**NodeType**(t_1, X) checks if the node type of t_1 is X.	**Exception**(e, P) checks if e is an exception caught in a `catch` clause or thrown in a method header in program P.
	NodeValue(t_1, v) checks if the corresponding source code of node t_1 is v.	**LocalCall**(m, P) checks if m is a local method call in program P.
Delete(t) removes the tree node t from the AST of a SO example.	**Match**(t_1, t_2) checks if t_1 and t_2 are matched based on surrounding nodes regardless of node types.	**InstanceCall**(m, P) checks if m is an instance call in program P.
	Parent(t_1, t_2) checks if t_1 is the parent of t_2 in the AST.	**Def**(v, P) checks if variable v is defined in program P.
Update(t_1, t_2) updates the tree node t_1 in a SO example with t_2 in the GitHub counterpart.	**Ancestor**(t_1, t_2) checks if t_1 is the ancestor of t_2 in the AST.	**Use**(v, P) checks if variable v is used in program P.
	Child(t_1, t_2) checks if t_1 is the child of t_2.	**IsExceptionType**(X) checks if X contains "Exception".
Move(t_1, t_2, i) moves an existing node t_1 in the AST of a SO example as the i-th child of t_2 in the GitHub counterpart.	**Changed**(t_1) is a shorthand for **Insert**(t_1, t_2, i) ∨ **Delete**(t_1) ∨ **Update**(t_1, t_2) ∨ **Move**(t_1, t_2), which checks any edit operation on t_1.	**IsLogMethod**(X) checks if X is one of the predefined log methods, e.g., log, println, error, etc.
		IsCleanMethod(X) checks if X is one of the predefined resource clean-up methods, e.g., close, recycle, dispose, etc.

cases or protects code from invalid input data such as `null` or an out-of-bound index. *Insert a final modifier* enforces that a variable is only initialized once and the assigned value or reference is never changed, which is generally recommended for clear design and better performance due to static inlining. *Handle a new exception* improves the reliability of a code example by handling any missing exceptions, since exception handling is often omitted in examples in SO [264]. *Clean up unmanaged resources* helps release unneeded resources such as file streams and web sockets to avoid resource leaks [233].

Resolve Compilation Errors. SO examples are often incomplete with undefined variables and method calls [55, 259]. *Declare an undeclared variable* inserts a statement to declare an unknown variable. *Specify a target of method invocation* resolves an undefined method call by specifying the receiver object of that call. In an example about getting CPU usage [4], one comment complains the example does not compile due to an unknown method call, `getOperatingSystemMXBean`. Another suggests to preface the method call with an instance, `ManagementFactory`, which is also evidenced by its GitHub counterpart [14]. Sometimes, statements that use undefined variables and method calls are simply deleted.

Exception Handling. This category represents changes of the exception handling logic in `catch/finally` blocks and `throws` clauses. One common change is to customize the actions in a `catch` block, e.g., printing a short error message instead of the entire stack trace. Some developers handle exceptions locally rather than throwing them in method headers. For example, while the SO example [8] throws a generic `Exception` in the `addLibraryPath` method, its GitHub clone [12] enumerates all possible exceptions such as `SecurityException` and `IllegalArgumentException` in a `try-catch` block. By contrast, propagating the exceptions to upstream by adding `throws` in the method header is another way to handle the exceptions.

Logic Customization. Customizing the functionality of a code example to fit a target project is a common and broad category. We categorize logic changes to four basic types. *Change a method call* includes any edits in a method call, e.g., adding or removing a method call, changing its arguments or receiver, etc. *Update a constant value* changes a constant value such as the thread sleep time to another value. *Change a conditional expression* includes any edits on the condition expression of an `if` statement, a `loop`, or a `switch case`.



(a) Distribution of AST edits (b) Code size vs. AST edits (c) Vote score vs. AST edits

Figure 7.2: Code size (LOC) and vote scores on the number of AST edits in a SO example

Update a type name replaces a variable type or a method return type with another type. For example, `String` and `StringBuffer` appear in multiple SO examples, and a faster type, `StringBuilder`, is used in their GitHub clones instead. Such type replacement often involves extra changes such as updating method calls to fit the replaced type or adding method calls to convert one type to another. For example, instead of returning `InetAddress` in a SO example [7], its GitHub clone [10] returns `String` and thus converts the IP address object to its string format using a new `Formatter` API.

Refactoring. 31% of inspected GitHub counterparts use a method or variable name different from the SO example. Instead of `slider` in a SO example [5], `timeSlider` is used in one GitHub counterpart [11] and `volumnSlider` is used in another counterpart [9]. Because SO examples often use hardcoded constant values for illustration purposes, GitHub counterparts may use variables instead of hardcoded constants. However, sometimes, a GitHub counterpart such as [13] does the opposite by inlining the values of two constant fields, `BUFFER_SIZE` and `KB`, since these fields do not appear along with the copied method, `downloadWithHttpClient` [6].

Miscellaneous. Adaptation types in this category do not have a significant impact on the reliability and functionality of a SO example. However, several interesting cases are still worth noting. In 91 inspected examples, GitHub counterparts include comments to explain the reused code. Sometimes, annotations such as `@NotNull` or `@DroidSafe` appear in GitHub counterparts to document the constraints of code.

7.3.2 Automated Adaptation Categorization

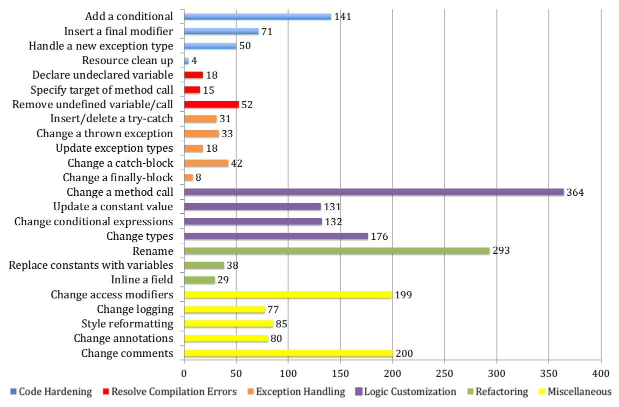
Based on the manual inspection, we build a rule-based classification technique that automatically categorizes AST edit operations generated by GumTree to different adaptation types. GumTree supports four edit operations—**insert**, **delete**, **update**, and **move**, described in Column GumTree Edit Operation in Table 7.1. Given a set of AST edits, our technique leverages both syntactic and semantic rules to categorize the edits to 24 adaptation types. Column Rule in Table 7.1 describes the implementation logic of categorizing each adaptation type.

Syntactic-based Rules. 16 adaptation types are detected based on syntactic information, e.g., edit operation types, AST node types and values, etc. Column Syntactic Predicate defines such syntactic information, which is obtained using the built-in functions provided by GumTree. For example, the rule *insert a final modifier* checks for an edit operation that inserts a `Modifier` node whose value is `final` in a GitHub clone.

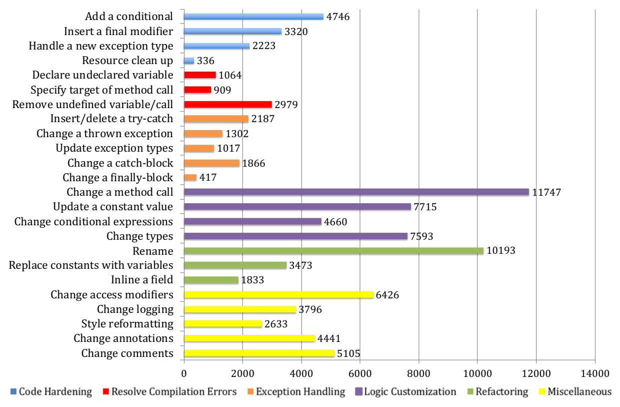
Semantic-based Rules. 8 adaptation types require leveraging semantic information to be detected (Column Semantic Predicate). For example, the rule *declare an undeclared variable* checks for an edit operation that inserts a `VariableDeclaration` node in the GitHub counterpart and the variable name is *used* but not *defined* in the SO example. Our technique traverses ASTs to gather such semantic information. For example, our AST visitor keeps track of all declared variables when visiting a `VariableDeclaration` AST node, and all used variables when visiting a `Name` node.

7.3.3 Accuracy of Adaptation Categorization

We randomly sampled another 100 SO examples and their GitHub clones to evaluate our automated categorization technique. To reduce bias, the second author who was not involved in the previous manual inspection labeled the adaptation types in this validation set. The ground truth contains 449 manually labeled adaptation types in 100 examples. Overall, our technique infers 440 adaptation types with 98% precision and 96% recall. In 80% of SO examples, our technique infers all adaptation types correctly. In another 20% of SO



(a) Adaptations: 629 explicitly attributed SO examples



(b) Variations: 14,124 potentially reused SO examples

Figure 7.3: Frequencies of categorized adaptation types in two datasets

examples, it infers some but not all expected adaptation types.

Our technique infers incorrect or missing adaptation types for two main reasons. First, our technique only considers 24 common adaptation types in Table 7.1 but does not handle infrequent ones such as refactoring using lambda expressions and rewriting `++i` to `i++`. Second, GumTree may generate sub-optimal edit scripts with unnecessary edit operations in about 5% of file pairs, according to [72]. In such cases, our technique may mistakenly report incorrect adaptation types.

7.4 An Empirical Study of Common Adaptations of Stack Overflow Code Examples

7.4.1 How many edits are potentially required to adapt a SO example?

We apply the adaptation categorization technique to quantify the extent of adaptions and variations in the two datasets. We measure AST edits between a SO example and its GitHub counterpart. If a SO code example has multiple GitHub counterparts, we use the average number. Overall, 13,595 SO examples (96%) in the variation dataset include a median of 39 AST edits (mean 47). 556 SO examples (88%) in the adaptation dataset include a median of 23 AST edits (mean 33). Figure 7.2a compares the distribution of AST edits in these two datasets. In both datasets, most SO examples have variations from their counterparts, indicating that integrating them to production code may require some type of adaptations.

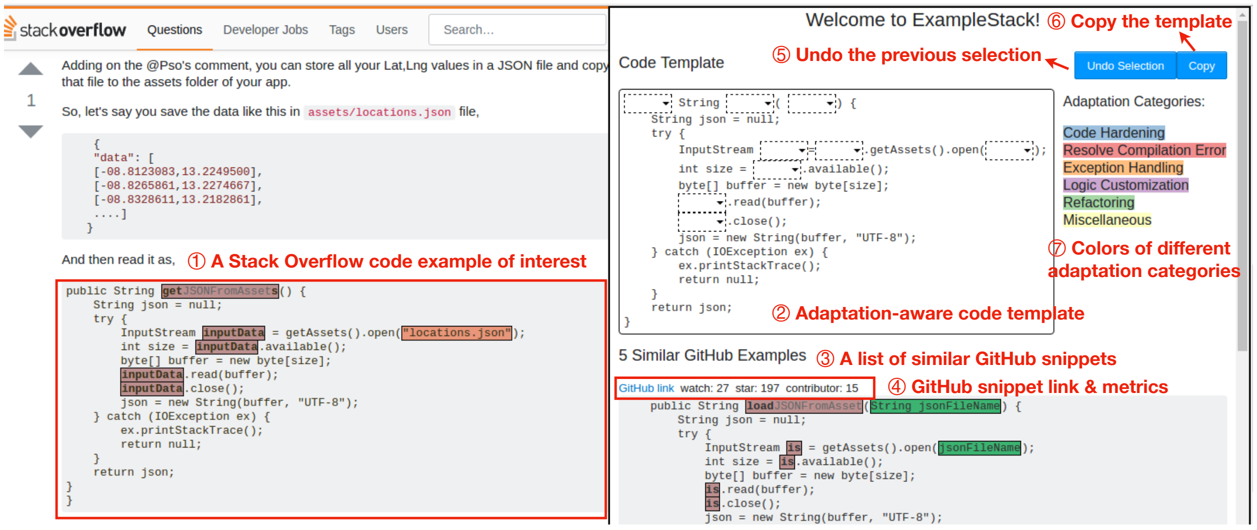
Figure 7.2b shows the median number of AST edits in SO examples with different lines of code. We perform a non-parametric local regression [210] on the example size and the number of AST edits. As shown by the two lines in Figure 7.2b, there is a strong positive correlation between the number of AST edits and the SO example size in both datasets—long SO examples have more adaptations than short examples.

Stack Overflow users can vote a post to indicate the applicability and usefulness of the post. Therefore, votes are often considered as the main quality metric of SO examples [166]. Figure 7.2c shows the median number of AST edits in SO examples with different vote scores. Although the adaptation dataset has significantly higher votes than the variation dataset (Figure 7.1c), there is no strong positive or negative correlation between the AST edit and the vote score in both sets. This implies that highly voted SO examples do not necessarily require fewer adaptations than those with low vote scores.

7.4.2 What are common adaptation and variation types?

Figure 7.3 compares the frequencies of the 24 categorized adaptation types (Column Adaptation Type in Table 7.1) for the adaptation and variation datasets. If a SO code example has multiple GitHub counterparts, we only consider the distinct types among all GitHub counterparts to avoid the inflation caused by repetitive variations among different counterparts. The frequency distribution is consistent in most adaptation types between the two datasets, indicating that *variation patterns resemble adaptation patterns.* Participants in the user study (Section 7.6) also appreciate being able to see variations in similar GitHub code, since "it highlights the best practices followed by the community and prioritizes the changes that I should make first," as P5 explained.

In both datasets, the most frequent adaptation type is *change a method call* in the logic customization category. Other logic customization types also occur frequently. This


stackoverflow
Questions
Developer Jobs
Tags
Users
Search...
1
Welcome to ExampleStack!
⑥ Copy the template
⑤ Undo the previous selection
Undo Selection
Copy
Code Template
① A Stack Overflow code example of interest
② Adaptation-aware code template
③ A list of similar GitHub snippets
④ GitHub snippet link & metrics
⑦ Colors of different adaptation categories

Adding on the @Pso's comment, you can store all your Lat,Lng values in a JSON file and copy that file to the assets folder of your app.

So, let's say you save the data like this in `assets/locations.json` file,

```
    {
    "data": [
    [-08.8123083,13.2249500],
    [-08.8265861,13.2274667],
    [-08.8328611,13.2182861],
    ....]
   }
```

And then read it as,

```
public String getJSONFromAssets() {
    String json = null;
    try {
        InputStream inputData = getAssets().open("locations.json");
        int size = inputData.available();
        byte[] buffer = new byte[size];
        inputData.read(buffer);
        inputData.close();
        json = new String(buffer, "UTF-8");
    } catch (IOException ex) {
        ex.printStackTrace();
        return null;
    }
    return json;
}
}
```

```
      String      (      ) {
    String json = null;
    try {
        InputStream      =     .getAssets().open(     );
        int size =     .available();
        byte[] buffer = new byte[size];
             .read(buffer);
             .close();
        json = new String(buffer, "UTF-8");
    } catch (IOException ex) {
        ex.printStackTrace();
        return null;
    }
    return json;
}
```

Adaptation Categories:

Code Hardening
Resolve Compilation Error
Exception Handling
Logic Customization
Refactoring
Miscellaneous

5 Similar GitHub Examples

GitHub link watch: 27 star: 197 contributor: 15

```
    public String loadJSONFromAsset(String jsonFileName) {
        String json = null;
        try {
            InputStream is = getAssets().open(jsonFileName);
            int size = is.available();
            byte[] buffer = new byte[size];
            is.read(buffer);
            is.close();
            json = new String(buffer, "UTF-8");
```

Figure 7.4: In the lifted template, common unchanged code is retained, while adapted regions are abstracted with *hot spots*.

is because SO examples are often designed for illustration purposes with contrived usage scenarios and input data, and thus require further logic customization. *Rename* is the second most common adaptation type. It is frequently performed to make variable and method names more readable for the specific context of a GitHub counterpart. 35% and 14% of SO examples in the variation dataset and the adaptation dataset respectively include undefined variables or local method calls, leading to compilation errors. The majority of these compilation errors (60% and 61% respectively) could be resolved by simply removing the statements using these undefined variables or method calls. 34% and 22% of SO examples in the two datasets respectively include new conditionals (e.g., an `if` check) to handle corner cases or reject invalid input data.

To understand whether the same type of adaptations appears repetitively on the same SO example, we count the number of adaptation types shared by different GitHub counterparts. Multiple clones of the same SO example share at least one same adaptation type in the 70% of the adaptation dataset and 74% of the variation dataset. In other words, *the same type of adaptations is recurring among different GitHub counterparts.*

7.5 Tool Support and Implementation

Based on insights of the adaptation analysis, we build a Chrome extension called ExampleStack that visualizes similar GitHub code fragments alongside a SO code example and allows a user to explore variations of the SO example in an adaptation-aware code template.

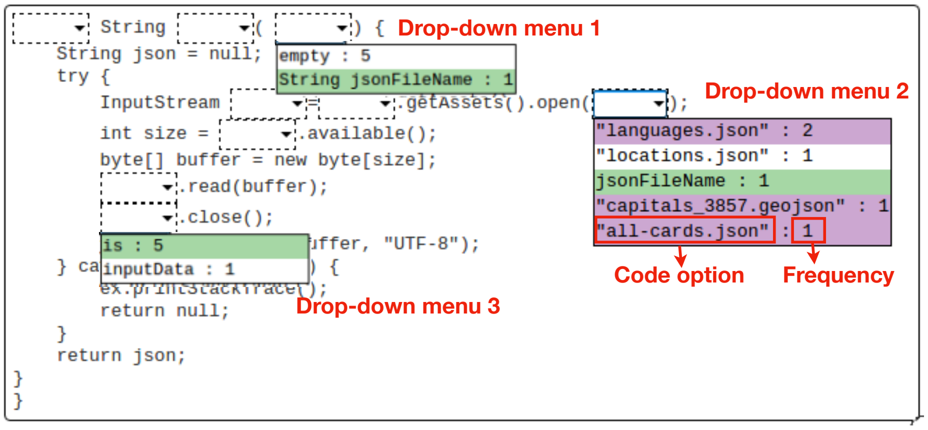
7.5.1 ExampleStack Tool Features

Suppose Alice is new to Android and she wants to read some `json` data from the asset folder of her Android application. Alice finds a SO code example [18] that reads geometric data from a specific file, `locations.json` (① in Figure 7.4). ExampleStack helps Alice by detecting other similar snippets in real-world Android projects and by visualizing the hot spots where adaptations and variations occur.

Browse GitHub counterparts with differences. Given the SO example, ExampleStack displays five similar GitHub snippets and highlights their variations to the SO example (③ in Figure 7.4). It also surfaces the GitHub link and reputation metrics of the GitHub repository, including the number of stars, contributors, and watches (④ in Figure 7.4). By default, it ranks GitHub counterparts by the number of stars.

View hot spots with code options. ExampleStack lifts a code template to illuminate unchanged code parts, while abstracting modified code as *hot spots* to be filled in (② in Figure 7.4). The lifted template provides a bird's-eye view and serves as a navigation model to explore a variety of code options used to customize the code example. In Figure 7.5, Alice can click on each hot spot and view the code options along with their frequencies in a drop-down menu. Code options are highlighted in six distinct colors according to their underlying adaptation intent (⑦ in Figure 7.4). For example, the second drop-down menu in Figure 7.5 indicates that two GitHub snippets replace `locations.json` to `languages.json` to read the language asset resources for supporting multiple languages. This variation is represented as *update a constant value* in the *logic customization* category.

Fill in hot spots with auto-selection. Instead of hardcoding the asset file name, Alice wants to make her program more general—being able to read asset files with any



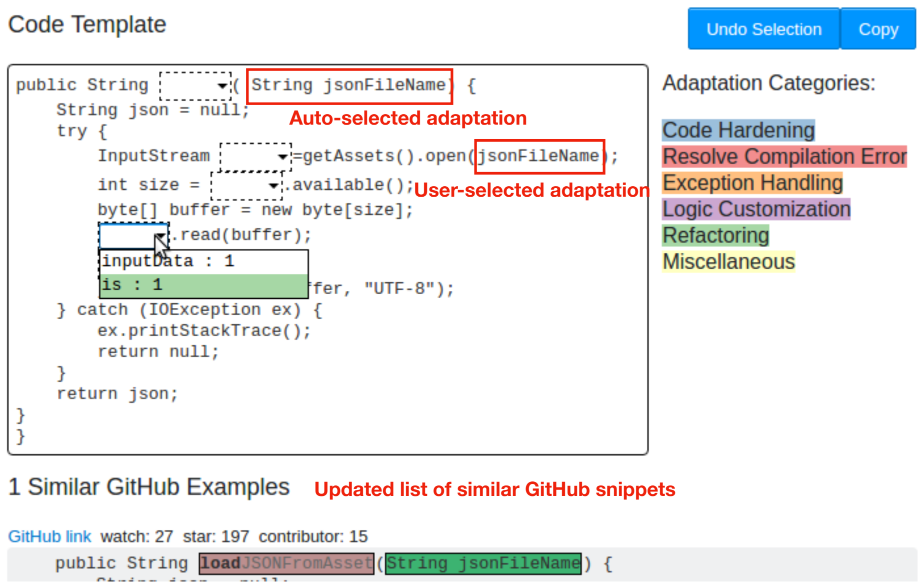
```
        String        (         ) {
    String json = null;
    try {
        InputStream        =        .getAssets().open(        );
        int size =        .available();
        byte[] buffer = new byte[size];
               .read(buffer);
               .close();
                         uffer, "UTF-8");
    } ca                 ) {
        ex.printStackTrace();
        return null;
    }
    return json;
}
}
```

Figure 7.5: Alice can click on a hot spot and view potential code options colored based on their underlying adaptation type.

given file name. Therefore, Alice selects the code option, `jsonFileName`, in the second drop-down menu in Figure 7.5, which generalizes the hardcoded file name to a variable. ExampleStack automatically selects another code option, `String jsonFileName`, in the first drop-down menu in Figure 7.5, since this code option declares the `jsonFileName` variable as the method parameter. This auto-selection feature is enabled by *def-use* analysis, which correlates code options based on the definitions and uses of variables (Section 7.5.2). By automatically relating code options in a template, Alice does not have to manually click through multiple drop-down menus to figure out how to avoid compilation errors. Figure 7.6 shows the customized template based on the selected `jsonFileName` option. The list of GitHub counterparts and the frequencies of other code options are also updated accordingly based on user selection. Alice can undo the previous selection (⑤ in Figure 7.4) or copy the customized template to her clipboard (⑥ in Figure 7.4).

7.5.2 Template Construction

Diff generating and pruning. To lift an adaptation-aware code template of a SO code example, ExampleStack first computes the AST differences between the SO example and each GitHub clone using GumTree. ExampleStack prunes the edit operations by filtering out *inner* operations that modify the children of other modified nodes. For example, if an insert operation inserts an AST node whose parent is also inserted by another insert, the



```
public String ( String jsonFileName) {
    String json = null;
    try {
        InputStream =getAssets().open(jsonFileName);
        int size = .available();
        byte[] buffer = new byte[size];
        .read(buffer);
        inputData : 1
        is : 1
                          fer, "UTF-8");
    } catch (IOException ex) {
        ex.printStackTrace();
        return null;
    }
    return json;
}
}
```

```
    public String loadJSONFromAsset(String jsonFileName) {
```

Figure 7.6: ExampleStack automatically updates the code template based on user selection.

first inner insert will be removed, since its edit is entailed by the second outer insert. Given the resulting tree edits, ExampleStack keeps track of the change regions in the SO example and how each region is changed.

Diff grouping. ExampleStack groups change regions to decide where to place hot spots in a SO example and what code options to display in a hot spot. If two change regions are the same, they are grouped together. If two change regions overlap, ExampleStack merges the overlapping change locations into a bigger region enclosing both and groups them together. For example, consider a diff that changes `a=`$\boxed{\texttt{b}}$ to `a=`$\boxed{\texttt{b+c}}$, and another diff that completely changes $\boxed{\texttt{a=b}}$ to $\boxed{\texttt{o.foo()}}$. Simply abstracting the changed code in these two diffs without any alignment will overlay two hot spots in the template, $\boxed{\texttt{a=}\boxed{\texttt{b}}}$ and the smaller diff is shadowed by the bigger diff in visualization. ExampleStack avoids this conflict by re-calibrating the first change region from `a=`$\boxed{\texttt{b}}$ to $\boxed{\texttt{a=b}}$.

Option generating and highlighting. For each group of change regions, ExampleStack replaces the corresponding location in the SO example with a hot spot and attaches a drop-down menu. ExampleStack displays both the original content in the SO example and contents of the matched GitHub snippet regions as options in each drop-down menu. ExampleStack then uses the adaptation categorization technique to detect the underlying

Table 7.2: Code reuse tasks and user study results

ID	Desired Function & SO Example	LOC	Clone#	Control			Experiment		
				Assignment	Adaptation	Time(s)	Assignment	Adaptation	Time(s)
Task I	Calculate the geographic distance between two GPS coordinates [20]	12	2	P5-A	refactor(5), logic(1)	458	P2-A	harden(1), logic(1), misc(2)	870
				P7-A	refactor(1), logic(2), misc(1)	900	P3-B	refactor(6), logic(4), misc(3)	900
				P12-B	refactor(2), harden(1)	900	P10-B	refactor(5), logic(2), misc(1)	366
				P16-B	refactor(7)	727	P15-A	refactor(10), logic(14), misc(3)	842
Task II	get the relative path between two files [16]	74	2	P3-A	refactor(5), logic(1), exception(2), misc(3)	900	P1-B	refactor(3), harden(1), logic(2)	640
				P8-A	harden(1)	900	P6-A	harden(4), logic(3)	900
				P11-B	none	621	P9-A	harden(4), logic(2)	900
				P15-B	refactor(13), harden(1), logic(5), exception(1), misc(1)	863	P13-B	refactor(3), logic(2), exception(1), misc(1)	900
Task III	encode a byte array to a hexadecimal string [15]	12	17	P1-A	refactor(5), harden(1)	652	P4-A	refactor(5), harden(1), misc(1)	667
				P6-B	refactor(1), misc(1)	900	P8-B	refactor(2), harden(1), misc(2)	548
				P9-B	harden(1), logic(1)	635	P12-A	refactor(3), harden(2), misc(1)	748
				P13-A	refactor(3), misc(1)	900	P14-B	refactor(3), harden(1), misc(1)	700
Task IV	add animation to an Android view [17]	29	4	P2-B	refactor(3), logic(1)	441	P5-B	refactor(1), logic(3)	478
				P4-B	refactor(1), compile(1), misc(1)	900	P7-B	refactor(2), compile(3), logic(3)	887
				P10-A	refactor(3), logic(5)	900	P11-A	refactor(1), logic(3)	617
				P14-A	refactor(2), logic(4)	862	P16-A	refactor(6), logic(4), misc(1)	773

adaptation types of code options. We use six distinct background colors to illuminate the categories in Table 7.1, which makes it easier for developers to recognize different intent. The color scheme is generated using ColorBrewer [21] to ensure the primary visual differences between different categories in the template.

ExampleStack successfully lifts code templates in all 14,124 SO examples. On average, a lifted template has 81 lines of code (median 41) with 13 hot spots (median 12) to fill in. On average, 4 code options (median 2) is displayed in the drop-down menu of each hot spot.

7.6 User Study

We conducted a within-subjects user study with sixteen Java programmers to evaluate the usefulness of ExampleStack. We emailed students in a graduate-level Software Engineering class and research labs in the CS department at UCLA. We did a background survey and excluded volunteers with no Java experience, since our study tasks required users to read code examples in Java. Fourteen participants were graduate students and two were undergraduate students. Eleven participants had two to five years of Java experience, while the other five were novice programmers with one-year Java experience, showing a good mix of different levels of Java programming experience.

In each study session, we first gave a fifteen-minute tutorial of our tool. Participants then did two code reuse tasks with and without ExampleStack. When not using our tool (i.e., the control condition), participants were allowed to search online for other code examples, which is commonly done in real-world programming workflow [48]. To mitigate learning effects, the order of assigned conditions and tasks were counterbalanced across participants through random assignment. In each task, we asked participants to mark which parts of a SO code example they would like to change and explain how they would change. We did not require participants to fully integrate a code example to a target program or make it compile, since our goal was to investigate whether ExampleStack could inspire developers with new adaptations that they may otherwise ignore, rather than automated code integration. Each task was stopped after fifteen minutes. At the end, we did a post survey to solicit feedback.

Table 7.2 describes the four code reuse tasks and also the user study results. Column Assignment in each condition shows the participant ID and the task order. "P5-A" means the task was done by the fifth participant as her first task. Column Adaptation shows the number of different types of adaptations each participant made. Overall, participants using ExampleStack made three times more code hardening adaptations (15 vs. 5) and twice more logic customization adaptations (43 vs. 20), considering more edge cases and different usage scenarios. For instance, in Task III, all users in the experimental group added a null check for the input byte array after seeing other GitHub examples, while only one user in the control group did so. P14 wrote, "*I would have completely forgotten about the null check without seeing it in a couple of examples.*" On average, participants using ExampleStack made more adaptations (8.0 vs. 5.5) in more diverse categories (2.8 vs. 2.2). Wilcoxon signed-rank tests indicate that the mean differences in adaptation numbers and categories are both statistically significant (p=0.042 and p=0.009). We do not argue that making more adaptations are always better. Instead, we want to emphasize that, by seeing commonalities and variations in similar GitHub code, participants focus more on code safety and logic customization, instead of making shallow adaptations such as variable renaming only. The average task completion time is 725 seconds (SD=186) and 770 seconds (SD=185) with and without ExampleStack. We do not claim ExampleStack saves code reuse time, since it is designed as an informative tool when developers browse online code examples, rather than providing direct code integration support in an IDE. Figure 7.7 shows the code templates generated by ExampleStack, not including the one in Task II due to its length (79 lines).

How do you like or dislike viewing similar GitHub code alongside a SO example? In the post survey, all participants found it very useful to see similar GitHub code for three main reasons. First, viewing the commonality among similar code examples helped users quickly understand the essence of a code example. P6 described this as "*the fast path to reach consensus on a particular operation.*" Second, the GitHub variants reminded users of some points they may otherwise miss. Third, participants felt more confident of a SO example after seeing how similar code was used in GitHub repositories. P9 stated that, "*[it is] reassuring to know that the same code is used in production systems and to know the common pitfalls.*"

How do you like or dislike interacting with a code template? Participants liked the code template, since it showed the essence of a code example and made it easier to see subtle changes, especially in lengthy code examples. Participants also found displaying the frequency count of different adaptations very useful. P5 explained, "*it highlights the best practices followed by the community and also prioritizes the changes that I should make first.*" However, we also observed that, when there were only a few GitHub counterparts, some participants inspected individual GitHub counterparts directly rather than interacting with the code template.

How do you like or dislike color-coding different adaptation types? Though the majority of participants confirmed the usefulness of this feature, six participants felt confused or distracted by the color scheme, since it was difficult to remember these colors during navigation. Three of them considered some adaptations (e.g., renaming) trivial and suggested to allow users to hide adaptations of no interest to avoid distraction.

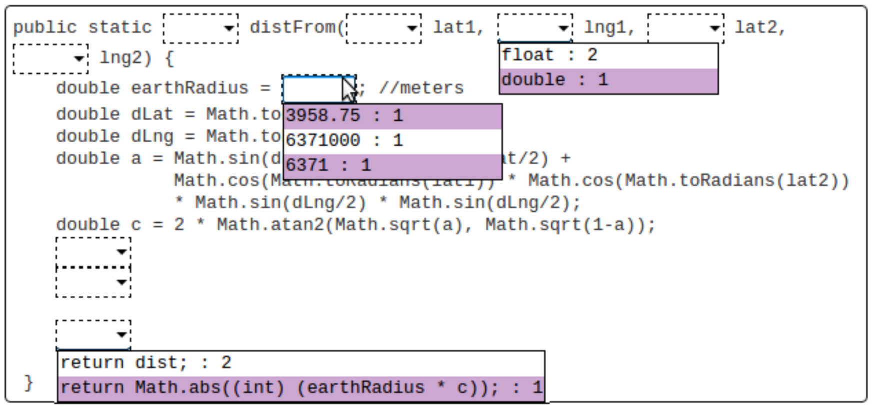
When would you use* ExampleStack*? Six participants would like to use ExampleStack when learning APIs, since it provided multiple GitHub code fragments that use the same API in different contexts with critical safety checks and exception handling. Five participants mentioned that ExampleStack would be most useful for a lengthy example. P4 wrote, "*the tool is very useful when the code is longer and hard to spot what to change at a glance.*" Two participants wanted to use ExampleStack to identify missing points and assess different solutions, when writing a large-scale robust project.

In addition, P15 and P16 suggested to display similar code based on semantic similarity rather than just syntactic similarity, in order to find alternative implementations and potential optimization opportunities. P13 suggested to add indicators about whether a SO example is compilable or not.

7.7 Threats to Validity

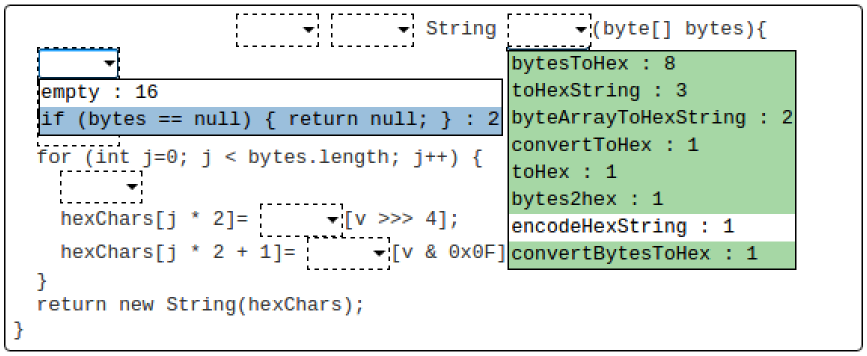
In terms of *internal validity*, our variation dataset may include coincidental clones, since GitHub developers may write code with similar functionality as a SO example. To mitigate this issue, we compare their timestamps and remove those GitHub clones that are created before the corresponding SO examples. We further create an adaptation set with explicitly attributed SO examples and compare the analysis results of both datasets for cross-validation. Figure 7.3 shows that the distribution of common adaptation patterns is similar between these two datasets. It would be valuable and useful to guide code adaptation by identifying the commonalities and variations between similar code, even for clones coming from independent but similar implementations.

In terms of *external validity*, when identifying common adaptation types, we follow the standard qualitative analysis procedure [45] to continuously inspect more samples till the insights are converging. However, we may still miss some adaptation types due to the small sample size. To mitigate this issue, the second author who was not involved in the manual inspection further manually labeled 100 more samples to validate the adaptation taxonomy (Section 7.3.3). In addition, user study participants may not be representative of real Stack Overflow users. To mitigate this issue, we recruit both novice and experienced developers who use Stack Overflow on a regular basis. To generalize our findings to industrial settings,



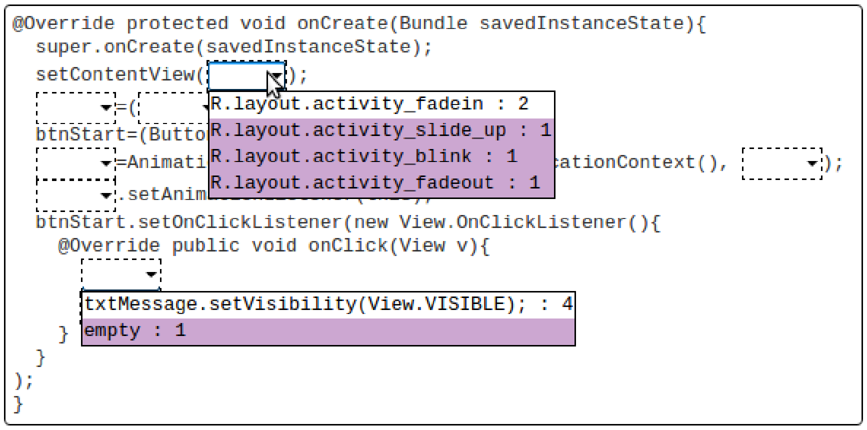
```
public static        distFrom(       lat1,        lng1,        lat2,
       lng2) {                                 float : 2
    double earthRadius =        ; //meters     double : 1
    double dLat = Math.to 3958.75 : 1
    double dLng = Math.to 6371000 : 1
    double a = Math.sin(d 6371 : 1          t/2) +
               Math.cos(M                   * Math.cos(Math.toRadians(lat2))
               * Math.sin(dLng/2) * Math.sin(dLng/2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));




    return dist; : 2
 }  return Math.abs((int) (earthRadius * c)); : 1
```

(a) Compute distance between two coordinates [20]



```
                            String        (byte[] bytes){
                                           bytesToHex : 8
  empty : 16                               toHexString : 3
  if (bytes == null) { return null; } : 2  byteArrayToHexString : 2
  for (int j=0; j < bytes.length; j++) {   convertToHex : 1
                                           toHex : 1
    hexChars[j * 2]=       [v >>> 4];      bytes2hex : 1
    hexChars[j * 2 + 1]=       [v & 0x0F]  encodeHexString : 1
  }                                        convertBytesToHex : 1
  return new String(hexChars);
}
```

(b) Encode byte array to a hex string [15]



```
@Override protected void onCreate(Bundle savedInstanceState){
  super.onCreate(savedInstanceState);
  setContentView(       );
        =(       R.layout.activity_fadein : 2
  btnStart=(Butto R.layout.activity_slide_up : 1
        =Animati R.layout.activity_blink : 1    ationContext(),        );
                 R.layout.activity_fadeout : 1
        .setAnim
  btnStart.setOnClickListener(new View.OnClickListener(){
    @Override public void onClick(View v){

      txtMessage.setVisibility(View.VISIBLE); : 4
    } empty : 1
  }
);
}
```

(c) Add animation to an Android view [17]

Figure 7.7: ExampleStack code template examples

further studies with professional developers are needed.

In terms of *construct validity*, in the user study, we only measure whether ExampleStack inspires participants to identify and describe adaptation opportunities. We do not ask participants to fully integrate a SO example to a target program nor make it compile. Therefore, our finding does not imply time reduction in code integration.

7.8 Summary

This chapter provides a comprehensive analysis of common adaptation and variation patterns of online code examples by both overapproximating and underapproximating reused code from Stack Overflow to GitHub. The key takeaway is that the same type of adaptations and variations appears repetitively among different GitHub clones of the same SO example, and variation patterns resemble adaptation patterns. This implies that different GitHub developers may apply similar adaptations to the same example over and over again independently. This further motivates the design of ExampleStack, a Chrome extension that guides developers in adapting online code examples by unveiling the commonalities and variations of similar past adaptations. A user study with sixteen developers demonstrates that by viewing the commonalities and variations in similar GitHub counterparts, developers identify more adaptation opportunities related to code safety and logic customization, resulting in more complete and robust code.

CHAPTER 8

Conclusion and Future Work

As software becomes ubiquitous and complex, developers often reuse existing software components (e.g., existing code fragments, library APIs) to build their own applications. The advent of code-sharing websites such as GitHub has significantly enriched software reuse opportunities by making a large number of open-source projects available online. As a result, many code fragments with similar implementation or API usage are shared within individual codebases and across different projects. There is a great opportunity to leverage these similar programs to guide software development and maintenance tasks, helping developers make systematic decisions based on what has and has not been done in other similar contexts.

This dissertation explores several opportunities to facilitate common tasks in developer workflow by identifying and analyzing similar code in local codebases and open-source projects. In Critics, we demonstrate that, by summarizing similar program edits and identifying inconsistencies among these edits, developers finish code review tasks with less time and find more edit mistakes. Using Grafter, developers can further investigate runtime behavior similarities between similar code and detect suspicious behavior discrepancies. ExampleCheck unveils common ways of using an API by mining representative API usage patterns from 380K GitHub projects. A large-scale study on Stack Overflow demonstrates that ExampleCheck can effectively identify code examples that deviate from common practices in open-source communities and prevent bug propagation during opportunistic code reuse. Examplore visualizes commonalities and variations among a large number of API usage examples in a synthetic code skeleton, providing a population-level view of API usage along with their frequencies. ExampleCheck enables developers to identify code adaptation opportunities and write more robust and complete code by displaying and contrasting similar code in GitHub. All together, we demonstrate that presenting multiple similar programs and illuminating their commonalities and variations can help developers avoid unintentional inconsistencies, identify better implementation alternatives, and get a deeper understanding of the program under investigation.

In the following two sections, we will summarize the insights we have learnt and discuss future directions.

8.1 Insights of Displaying and Contrasting Similar Programs

Insight 1. In order to effectively leverage similar programs, it is essential to first define a proper program abstraction to capture desired program similarity. In this thesis, we demonstrate applications that leverage two kinds of similar programs—code fragments with similar syntactic structures and code fragments with similar API usage. The general idea of displaying and contrasting similar code for systematic software development can be further applied to other kinds of similar programs. For example, research on automated program repair may concern more about common edit patterns, e.g., applying a null check on a variable before using it. Therefore, a similarity metric on edit operations may be more appropriate to identify common fix patterns than syntactic similarities in program contexts. Other research may also want to use input-output values or execution traces to model programs that have similar runtime behaviors. For each kind of similarity definition, it is essential to design a proper program abstraction to capture desired similarity while eliminating superficial variations. For example, when searching for syntactically similar code, abstract syntax tree (AST) can faithfully represent code elements and structures. However, for learning similar API usage, structured API call sequence is more appropriate than AST by unifying similar API usage in different syntactic expressions, e.g., calling multiple API methods in a single statement by chaining them together vs. calling multiple API methods in separate statements.

Insight 2. Allowing developers to interactively specify desired code and provide partial feedback can effectively guide automated program analysis and improve

the accuracy of analysis results. In the absence of formal specifications or test oracles, it is hard to guarantee that variations in similar code represent alternative implementations worth exploring or unintentional inconsistencies that lead to bugs. In fact, this is a general problem in code recommendation and bug detection tools—developers are often overwhelmed with a large number of false positives, which hinders the adoption of these tools [112]. Prior work mostly focuses on reducing false positives by designing effective ranking and filtering heuristics. However, in this thesis, we demonstrate the effectiveness of incorporating user feedback to refine analysis results. We explore two different kinds of interaction designs. First, in CRITICS, we allow developers to construct a search template from one concrete example and incrementally customize the template based on previous search results. In a follow-on work of CRITICS, we design a novel user interaction paradigm to further reduce the manual effort of editing a template, where developers only need to label some positive and negative examples and the template will be automatically refined via active learning [215]. Second, in EXAMPLORE and EXAMPLESTACK, we automatically lift a template from multiple code examples but allow developers to interact with the template to drill down to concrete details. The former interaction is more suitable for specifying and searching similar code with complex patterns, while the latter is more appropriate for visualizing and exploring a large volume of similar code that is already present.

Insight 3. It is important to combine statistical methods with semantic reasoning in big code analysis. When learning coding idioms from massive code corpora, we need to reason about program semantics rather than simply treating source code as sequences of tokens. For example, to analyze common API usage in thousands of software projects written by different developers, it is important to eliminate project-specific details and cluster semantically equivalent expressions with subtle syntactic variations. The experiment of EXAMPLECHECK shows that, by eliminating extraneous program statements that are unrelated to the focal API in a large code corpus, program slicing improves pattern mining precision and recall by 15% and 10% respectively, and improves mining performance by 4.5X.

Many mining techniques only apply statistical methods such as the n-gram model to infer patterns of surface structures such as tokens and syntax [24, 95, 98, 106, 186, 189]. Therefore, these techniques can only learn statistical correlations between code elements rather than semantic relationships between them. Some pattern inference techniques perform static analysis such as program slicing and symbolic execution to reason about program semantics [50, 248]. However, static analysis is computationally expensive and thus does not scale to a large collection of open-source projects. In this thesis, we make the first attempt to apply sophisticated program analysis on massive code corpora by harnessing the power of distributed computation. Specifically, we leverage AST traversal primitives in a distributed software mining infrastructure [66] to implement the program slicing algorithm. The process of searching for similar programs and performing program slicing takes no more than 15 minutes (10 minutes on average) over 380K GitHub projects in a cluster of eleven machines.

Insight 4. Helping developers understand the gist of similar programs is as important as identifying and analyzing similar programs. Writing robust code based on reference implementation depends on the capability to understand similar code in different program scenarios. Pairwise comparison is a ubiquitous way to contrast programs. However, through user studies, we find that pairwise comparison is cognitively overloading when there are more than three similar programs. Developers tend to only inspect the first few examples and skip the rest. Prior studies on code search also make similar observations—developers often rapidly examine a handful of search results and return to their own code due to limited time and attention [48, 218]. Therefore, it is necessary to provide concise, interactive visualizations to represent commonalities and variations among similar code at scale. In ExampleCheck, we summarize common API usage as pattern rules. Though such rule-based representation can concisely express the gist of common API usage, it cannot easily convey the variations among different patterns. Therefore, in Examplore, we design an alternative visualization by clustering distinct API usage features extracted from concrete examples and synthesizing a code skeleton with all of these features and their frequencies. To generalize the visualization from API usage to program constructs in arbitrary code, in ExampleStack, we design a visualization that lifts a code template from similar code fragments by retaining common, unchanged code while abstracting away variations as "holes" in the template.

8.2 Future Directions

Learning Infrequent but Semantically Correct Patterns The central thesis of frequency-based mining approaches is that a pattern that occurs more frequently is more likely to be representative and correct. However, this may not always be the case. For example, the majority way of using a cryptographic API in open-source communities may be insecure, e.g., not using a strong encryption mode such as RSA [165]. In the manual analysis of 400 sampled API usage violations detected by ExampleCheck, we find that about 9% of these violations are false positives due to correct but infrequent API usage. In Examplore, we present one solution by visualizing all distinct API usage features with their statistical distributions in the population of relevant code examples, regardless of their frequencies. Instead of assuming all code is equally likely to be correct, Le Goues and Weimer propose to use code quality metrics (e.g., code churn, cyclomatic complexity) to assign more weights to patterns learned from code with high quality [138]. Given the abundance of online learning resources, we can also triangulate inferred patterns with multiple information sources, e.g., correlating patterns inferred from massive code corpora with discussions in Stack Overflow. For security APIs, we may want to assign more weights to API usage confirmed by official documentation and security-related technical blogs, though they may occur less frequently in open-source projects.

Bridging Probabilistic Reasoning with Semantic Reasoning Given its capability to generalize from examples and handle noises, statistical methods such as machine learning have great potential to infer sophisticated program patterns that cannot be easily identified by clustering or frequency-based mining approaches. However, unlike traditional formal, logic-based program analysis, which often represents programs in a symbolic form, machine learning research represents programs with continuous representations. As a result, probabilistic models often lack the guarantee that formal methods often provide, e.g., producing invalid code that does not satisfy semantic constraints in software programs. Existing techniques simply apply semantic constraints on the outputs of a model to filter out invalid outputs [147, 189]. By contrast, enabling statistical methods to reason with symbolic constraints may learn more correct program patterns and produce better results. Since symbolic reasoning methods are not differentiable, we cannot direct add symbolic reasoning to a machine learning pipeline. One possible solution is to translate a semantic constraint as a differentiable loss function and minimize the loss function to satisfy the given constraint during model training. Xu et al. propose *semantic loss functions* to enforce symbolic domain knowledge in deep learning [257]. However, their approach is restricted to simple logic constraints constituted of propositional variables and logical operators ($\wedge$, $\vee$, etc). Therefore, it is is intractable for complicated constraints expressed in formal methods. Exploring how to extend semantic loss functions to effectively express program constraints remains an interesting future direction.

Scaling Programming to the Globe Twenty years ago, developers often write code by themselves or search local codebases for similar code to reuse. Nowadays, they have more opportunities to observe and learn from similar code written by other developers, given the abundance of tutorials, discussions, and code snippets shared on the Internet. There are also many opportunities to design effective tool support that aggregates and disseminates programming knowledge at a global scale. For example, the cost of developing test cases is high, which makes test reuse among similar programs appealing. The availability of big code will significantly increase the opportunities of identifying a diverse set of test cases that examine similar functionality. In GRAFTER, we have explored how to reuse test cases between similar code in local codebases via code transplantation. However, new challenges may arise when reusing test cases across different projects. For instance, similar programs from different projects may have more syntactic variations than similar code from the same codebase, e.g., using different libraries with similar functionality. Though a number of techniques have been proposed to detect semantically similar programs in code search and clone detection [110, 173, 220, 222, 265], it is unclear whether these techniques scale to massive code corpora. Furthermore, we need to design new code transformation and data propagation rules to handle a diverse set of variations that have not been seen in within-project clones, e.g., transferring data between objects with similar data structures but from different libraries.

Another intriguing application is to guide programs synthesis by harnessing the power of big code. Existing program synthesis techniques focus on synthesizing small programs in a specific domain by designing a domain-specific language (DSL) and synthesize programs from the scratch using the DSL [94,108,162,243,258,262]. However, online Q&A forums such as Stack Overflow host a large number of small code snippets, which can serve as the basis for composing complete programs. Therefore, there is a chance that we can automatically locate related snippets that implement individual operations, glue them together, and then let developers make the final tweaks to achieve the desired functionality. Several challenges must be addressed to achieve this goal. First, online code snippets are often designed for illustration purposes only and thus may not have concrete details such as variable declarations and exception handling logic, which should be filled in automatically. Second, many of these snippets are not designed to work together. Therefore, we need to design a set of transformation rules to glue multiple snippets together and reconcile potential inconsistencies between them. However, it is hard to design such transformation rules without a deep understanding about how real developers perform similar tasks in practice. For example, do developers often concatenate two snippets one after another or do they often interleave them together? Third, given a partial specification (e.g., keyword descriptions, input/output types, test cases) from a developer, how do we efficiently locate the underlying atomic operations and corresponding code snippets?

In summary, programming is not just a logical activity that designs algorithms and data structures and encodes them in a formal notation. It also involves cognitive and social processes where developers read and learn from code written by others. As the Internet accumulates an enormous volume of source code and the computing power grows, more and more resources become available for harnessing the power of big code. Therefore, we envision a data-driven programming paradigm that enables developers to write robust code by demonstrating how other developers write similar code in open-source communities. This thesis demonstrates that, by identifying and contrasting similar code in different contexts, developers can avoid potential programming mistakes and identify better, alternative implementations that may otherwise be overlooked. We believe such a data-driven paradigm will play an important role in modern software development and can be further applied to many other software development tasks such as program repair and code completion.

REFERENCES

[1] Levenshtein distance wikipedia. https://en.wikipedia.org/wiki/Levenshtein_distance.

[2] Simian - similarity analyser. http://www.harukizaemon.com/simian/. Accessed: 2017-11-15.

[3] Stable marriage problem (smp) wikipedia. https://en.wikipedia.org/wiki/Stable_marriage_problem.

[4] *Get OS-level system information*, 2008. https://stackoverflow.com/questions/61727.

[5] *JSlider question: Position after leftclick*, 2009. https://stackoverflow.com/questions/518672.

[6] *Youtube data API : Get access to media stream and play (JAVA)*, 2011. https://stackoverflow.com/questions/4834369.

[7] *How to get IP address of the device from code?*, 2012. https://stackoverflow.com/questions/7899226.

[8] *Adding new paths for native libraries at runtime in Java*, 2013. https://stackoverflow.com/questions/15409446.

[9] *Another GitHub clone about JSlide*, 2014. https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java#L343-L353.

[10] *A GitHub clone about how to get IP address from an Android device.*, 2014. https://github.com/kalpeshp0310/GoogleNews/blob/master/app/src/main/java/com/kalpesh/googlenews/utils/Utils.java#L24-L39.

[11] *A GitHub clone about JSlide*, 2014. https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java#L329-L339.

[12] *A GitHub clone that adds new paths for native libraries at runtime in Java*, 2014. https://github.com/armint/firesight-java/blob/master/src/main/java/org/firepick/firesight/utils/SharedLibLoader.java#L131-L153.

[13] *A GitHub clone that downloads videos from YouTube*, 2014. https://github.com/instance01/YoutubeDownloaderScript/blob/master/IYoutubeDownloader.java#L148-L193.

[14] *A GitHub clone that gets the CPU usage*, 2014. https://github.com/jomis/nomads/blob/master/nomads-framework/src/main/java/at/ac/tuwien/dsg/utilities/PerformanceMonitor.java#L44-L63.

[15] *How to use SHA-256 with Android*, 2014. https://stackoverflow.com/questions/25803281.

[16] *Construct a relative path in Java from two absolute paths*, 2015. https://stackoverflow.com/questions/3054692.

[17] *How can I add animations to existing UI components?*, 2015. https://stackoverflow.com/questions/33464536.

[18] *Add Lat and Long to ArrayList*, 2016. https://stackoverflow.com/questions/37273871.

[19] *Stack Overflow data dump*, 2016. https://archive.org/details/stackexchange, accessed on Oct 17, 2016.

[20] *Calculate distance in meters when you know longitude and latitude in java*, 2017. https://stackoverflow.com/questions/837957.

[21] *ColorBrewer: Color Advice for Maps*, 2018. http://colorbrewer2.org.

[22] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: An effective verification process. *IEEE software*, 6(3):31–36, 1989.

[23] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, page 10 pp., nov. 2005.

[24] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[25] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[26] S. Amani, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. Mubench: a benchmark for api-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 464–467. ACM, 2016.

[27] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[28] L. An, O. Mlouki, F. Khomh, and G. Antoniol. Stack overflow: a code laundering platform? In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 283–293. IEEE, 2017.

[29] J. Andersen. *Semantic Patch Inference.* Ph.D. Disseratation, University of Copenhagen, Copenhagen, Nov. 2009. Adviser-Julia L. Lawall.

[30] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Proc. of ICSE*, pages 402–411, 2005.

[31] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[32] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[33] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[34] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 26–30. IEEE Press, 2012.

[35] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4. IEEE Computer Society, 2009.

[36] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[37] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *2007 IEEE International Conference on Software Maintenance*, pages 24–33. IEEE, 2007.

[38] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 326, Washington, DC, USA, 1999. IEEE Computer Society.

[39] S. Baltes and S. Diehl. Usage and attribution of stack overflow code snippets in github projects. *arXiv preprint arXiv:1802.02938*, 2018.

[40] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE Press, 2015.

[41] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM.

[42] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[43] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[44] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.

[45] B. L. Berg, H. Lune, and H. Lune. *Qualitative Research Methods for the Social Sciences*, volume 5. Pearson Boston, MA, 2004.

[46] B. Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

[47] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[48] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[49] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Writing code to prototype, ideate, and discover. *IEEE software*, 26(5):18–24, 2009.

[50] R. P. Buse and W. Weimer. Synthesizing api usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 782–792. IEEE, 2012.

[51] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *Software Engineering, IEEE Transactions on*, 34(5):579–596, Sept 2008.

[52] F. Chen and S. Kim. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 320–332. ACM, 2015.

[53] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[54] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225. ACM, 2008.

[55] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, 2012.

[56] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[57] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[58] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[59] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995.

[60] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[61] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Automata, languages and programming*, pages 146–157. Springer, 2007.

[62] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[63] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar apis: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*, pages 266–276. IEEE Press, 2012.

[64] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 467–476, New York, NY, USA, 2000. ACM. code inspection, code review, object-oriented, delocalized.

[65] A. Dunsmore, M. Roper, and M. Wood. Practical code inspection techniques for object-oriented systems: an experimental comparison. *IEEE software*, 20(4):21–29, 2003.

[66] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[67] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[68] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264. ACM, 2006.

[69] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[70] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

[71] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999. code inspection, checklist.

[72] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[73] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 121–136. IEEE, 2017.

[74] G. Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60, 1987.

[75] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[76] D. Ford, K. Lustig, J. Banks, and C. Parnin. We don't do that here: How collaborative editing with mentors improves engagement in social q&a communities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, page 608. ACM, 2018.

[77] D. Ford, J. Smith, P. J. Guo, and C. Parnin. Paradise unplugged: Identifying barriers for female participation on stack overflow. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 846–857. ACM, 2016.

[78] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008.

[79] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 15–24. ACM, 2010.

[80] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[81] R. E. Gallardo-Valencia and S. Elliott Sim. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 49–52. IEEE Computer Society, 2009.

[82] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations.* Springer Science & Business Media, 2012.

[83] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing q&a sites (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 307–318. IEEE, 2015.

[84] M. Gharehyazie, B. Ray, and V. Filkov. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 291–301. IEEE, 2017.

[85] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[86] N. Göde. Clone removal: Fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones*, pages 33–40. ACM, 2010.

[87] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proceedings of the international workshop on Principles of software evolution*, pages 117–119. ACM, 2002.

[88] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[89] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE, 2012.

[90] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, volume 90, 2003.

[91] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631. IEEE, 2007.

[92] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130. ACM, 2010.

[93] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[94] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *ACM SIGPLAN Notices*, volume 46, pages 62–73. ACM, 2011.

[95] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[96] M. Harman, Y. Jia, and W. B. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *International Symposium on Search Based Software Engineering*, pages 247–252. Springer, 2014.

[97] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[98] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[99] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22. ACM, 2007.

[100] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[101] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, pages 447–457. IEEE Computer Society, 2007.

[102] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology*, 21(4):20:1–20:44, Nov. 2012.

[103] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[104] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 53–62. IEEE, 2012.

[105] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[106] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 2073–2083.

[107] S. Jansen, S. Brinkkemper, I. Hunink, and C. Demir. Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE software*, 25(6), 2008.

[108] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[109] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[110] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009.

[111] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.

[112] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[113] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[114] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[115] N. Juillerat and B. Hirsbrunner. Toward an implementation of the" form template method" refactoring. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 81–90. IEEE, 2007.

[116] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436. ACM, 2014.

[117] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.

[118] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[119] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing. In *Proc. of FSE*, pages 654–665, 2014.

[120] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[121] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[122] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[123] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. F a c o y: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957. ACM, 2018.

[124] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, New York, NY, USA, 2006. ACM.

[125] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319. IEEE Computer Society, 2009.

[126] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[127] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[128] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[129] N. Kong, T. Grossman, B. Hartmann, M. Agrawala, and G. Fitzmaurice. Delta: a tool for representing and comparing workflows. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1027–1036. ACM, 2012.

[130] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[131] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[132] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[133] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[134] K. J. Kurtz, C.-H. Miao, and D. Gentner. Learning by analogical bootstrapping. *The Journal of the Learning Sciences*, 10(4):417–446, 2001.

[135] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Testing of Communicating Systems*, pages 19–38. Springer, 2006.

[136] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. *ACM SIGCHI Bulletin*, 20(SI):69–73, 1989.

[137] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 476–482. ACM, 2009.

[138] C. Le Goues and W. Weimer. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering*, 38(1):175–190, 2011.

[139] S. Lee and I. Jeong. Sdd: High performance code clone detection system for large scale source code. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 140–141, New York, NY, USA, 2005. ACM.

[140] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[141] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[142] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[143] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, (5):23–30, 1994.

[144] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.

[145] D. Lo, L. Jiang, A. Budi, et al. Active refinement of clone anomaly reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 397–407. IEEE Press, 2012.

[146] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1:28, 2017.

[147] C. Maddison and D. Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657, 2014.

[148] S. Makady and R. J. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice & Experience*, 43(9):1039–1070, Sept. 2013.

[149] U. Manber et al. Finding similar files in a large file system. In *Usenix Winter*, volume 94, pages 1–10, 1994.

[150] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[151] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[152] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM.

[153] F. Marton and S. Booth. *Learning and awareness.* Routledge, 2013.

[154] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

[155] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[156] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[157] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120. IEEE, 2011.

[158] N. Meng, L. Hua, M. Kim, and K. 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.

[159] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI '11*, pages 329–342, San Jose, CA, 2011. ACM.

[160] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[161] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 372–383. IEEE, 2018.

[162] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of The 30th International Conference on Machine Learning*, pages 187–195, 2013.

[163] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*, pages 2–25. Springer, 2010.

[164] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 401–408. IEEE, 2013.

[165] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946. ACM, 2016.

[166] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 25–34. IEEE, 2012.

[167] C. Ncube, P. Oberndorf, and A. W. Kark. Opportunistic software systems development: making systems from what's available. *IEEE Software*, 25(6):38–41, 2008.

[168] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–177. ACM, 2014.

[169] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. Nguyen. A graph-based approach to api usage adaptation. In *OOPSLA '10: Proceedings of the 2010 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications*, page 10, New York, NY, USA, 2010. ACM.

[170] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, New York, NY, USA, 2009. ACM.

[171] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[172] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *MSR '12: The 9th IEEE Working Conference on Mining Software Repositories*, pages 40–49, 2012.

[173] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *ACM SIGPLAN Notices*, volume 49, pages 811–828. ACM, 2014.

[174] A. Pavel, F. Berthouzoz, B. Hartmann, and M. Agrawala. Browsing and analyzing the command-level structure of large collections of image manipulation tutorials. *Citeseer, Tech. Rep.*, 2013.

[175] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[176] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[177] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. ACM.

[178] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[179] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[180] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 371–382. IEEE Computer Society, 2009.

[181] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE Press, 2012.

[182] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1 –10, 2010.

[183] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: synthesizing what i mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 357–367. ACM, 2016.

[184] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, volume 42, pages 123–134. ACM, 2007.

[185] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *International Conference on Compiler Construction*, pages 39–56. Springer, 2004.

[186] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439. IEEE, 2016.

[187] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 53. ACM, 2012.

[188] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 367–377. IEEE Press, 2013.

[189] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[190] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[191] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[192] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[193] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 541–550, New York, NY, USA, 2008. ACM.

[194] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841. IEEE Press, 2013.

[195] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6), 2009.

[196] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press, 2017.

[197] M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[198] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[199] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 81–90. IEEE, 2008.

[200] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[201] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[202] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.

[203] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123. ACM, 2005.

[204] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[205] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365. ACM, 2018.

[206] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[207] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.

[208] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, 2005.

[209] A. Shrikumar. *Designing an Exploratory Text Analysis Tool for Humanities and Social Sciences Research*. University of California, Berkeley, 2013.

[210] W. M. Shyu, E. Grosse, and W. S. Cleveland. Local regression models. In *Statistical models in S*, pages 309–376. Routledge, 2017.

[211] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI '15: Proceedings of the 36th ACM SIGPLAN Conference on Programming language design and implementation*, volume 50, pages 43–54. ACM, 2015.

[212] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):4, 2011.

[213] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–. IBM Press, 1997.

[214] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, 2018.

[215] A. Sivaraman, T. Zhang, G. V. d. Broeck, and M. Kim. Active inductive logic programming for code search. page 12 pages, 2019.

[216] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[217] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE software*, 21(1):28, 2004.

[218] J. Starke, C. Luce, and J. Sillito. Working with search results. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 53–56. IEEE Computer Society, 2009.

[219] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*, pages 1379–1392. ACM, 2015.

[220] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26, 2014.

[221] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 195–202. IEEE, 2006.

[222] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 702–714. ACM, 2016.

[223] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 85–88. IEEE Press, 2013.

[224] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[225] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.

[226] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[227] V. Terragni, Y. Liu, and S.-C. Cheung. Csnippex: automated synthesis of compilable code snippets from q&a sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 118–129. ACM, 2016.

[228] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[229] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[230] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, pages 496–506. IEEE Computer Society, 2009.

[231] S. Thummalapenta and T. Xie. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering*, 18(3):293, 2011.

[232] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA, 2010. ACM.

[233] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 535–544. ACM, 2010.

[234] C. Treude and M. P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, pages 392–403. ACM, 2016.

[235] C. Treude and M. P. Robillard. Understanding stack overflow code fragments. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[236] N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 60–70. IEEE Press, 2017.

[237] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 127–136. IEEE, 2002.

[238] M. Umarji, S. E. Sim, and C. Lopes. Archetypal internet-scale source code searching. In *IFIP International Conference on Open Source Systems*, pages 257–263. Springer, 2008.

[239] G. Upadhyaya and H. Rajan. Collective program analysis. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018.

[240] B. Vasilescu, A. Capiluppi, and A. Serebrenik. Gender, representation and online participation: A quantitative study. *Interacting with Computers*, 26(5):488–511, 2013.

[241] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pages 3789–3798. ACM, 2015.

[242] A. J. Viera, J. M. Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363, 2005.

[243] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *ACM SIGPLAN Notices*, volume 52, pages 452–466. ACM, 2017.

[244] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[245] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 19(8), 2007.

[246] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077. ACM, 2018.

[247] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 457–466, New York, NY, USA, 2010. ACM.

[248] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44. ACM, 2007.

[249] M. Wattenberg and F. B. Viégas. The word tree, an interactive visual concordance. *IEEE transactions on visualization and computer graphics*, 14(6), 2008.

[250] K. E. Weigers. *Peer reviews in software: a practical guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[251] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[252] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[253] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 219–228. ACM, 2012.

[254] M. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 152–158. IEEE, 1988.

[255] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, pages 1–37, 2018.

[256] T. Xie, K. Taneja, S. Kale, and D. Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 5. IEEE Computer Society, 2007.

[257] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International Conference on Machine Learning*, pages 5498–5507, 2018.

[258] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.

[259] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 391–402. ACM, 2016.

[260] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in github: any snippets there? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 280–290. IEEE, 2017.

[261] 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.

[262] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234. IEEE, 2013.

[263] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proceedings of 37th IEEE/ACM International Conference on Software Engineering. IEEE*, 2015.

[264] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 886–896. IEEE, 2018.

[265] G. Zhao and J. Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151. ACM, 2018.

[266] H. Zhong and X. Wang. Boosting complete-code tool for partial program. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 671–681. IEEE Press, 2017.

[267] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

[268] J. Zhou and R. J. Walker. Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277. ACM, 2016.

[269] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[270] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 146. IEEE, 2003.